Tethys Platform Documentation

Release 2.1.3

Nathan Swain

Feb 08, 2020

CONTENTS

Last Updated: December 2018

Tethys is a platform that can be used to develop and host environmental web apps. It includes a suite of free and open source software (FOSS) that has been carefully selected to address the unique development needs of environmental web apps. Tethys web apps are developed using a Python software development kit (SDK) which includes programmatic links to each software component. Tethys Platform is powered by the Django Python web framework giving it a solid web foundation with excellent security and performance. Refer to the *Features* article for an overview of the features of Tethys Platform.

Important: Tethys Platform 2.1.3 has arrived! Check out the *What's New* article for a description of the new features and changes.

Warning: Python 2 support is officially deprecated in this release. It will no longer be supported in the next release of Tethys Platform. Migrate now!

CHAPTER

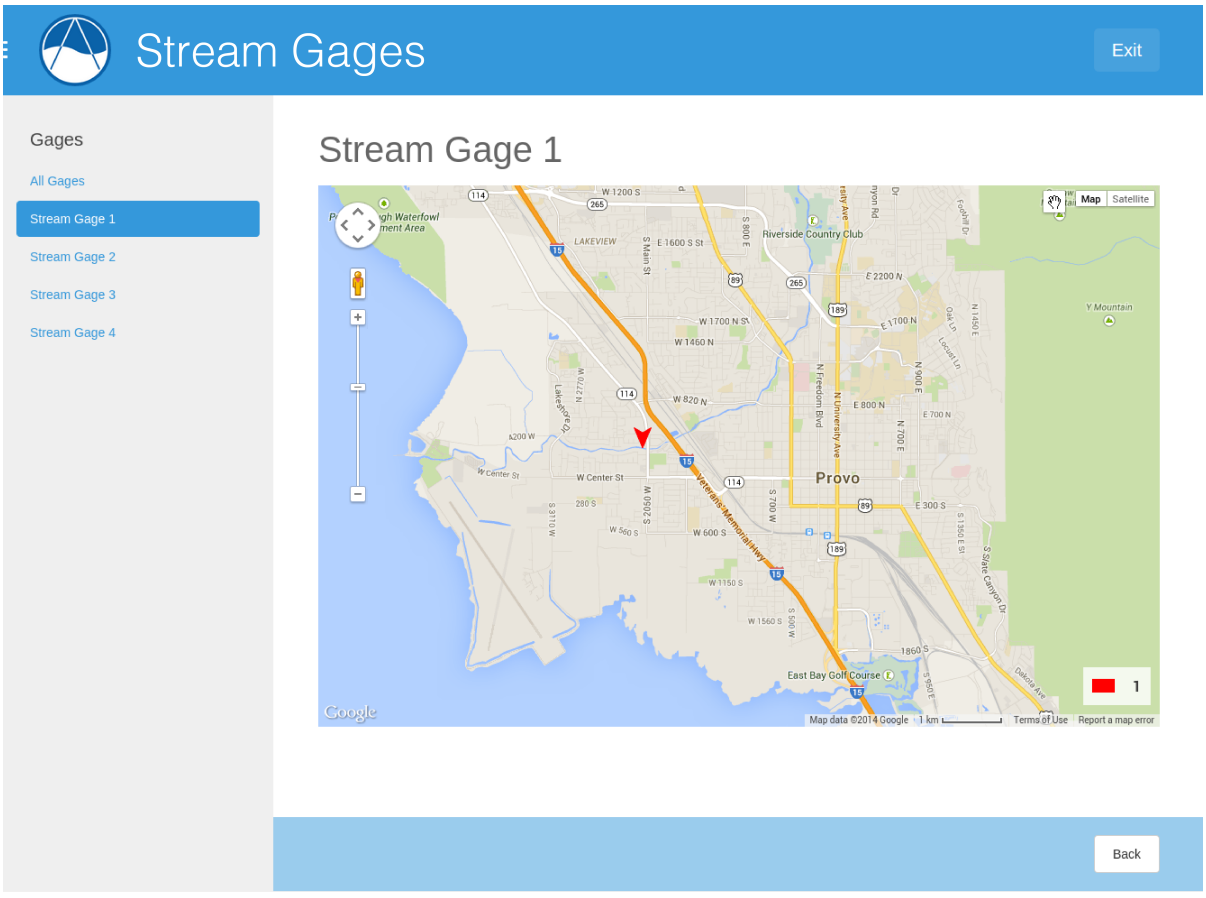
ONE

CONTENTS

1.1 Features

Last Updated: May 28, 2015

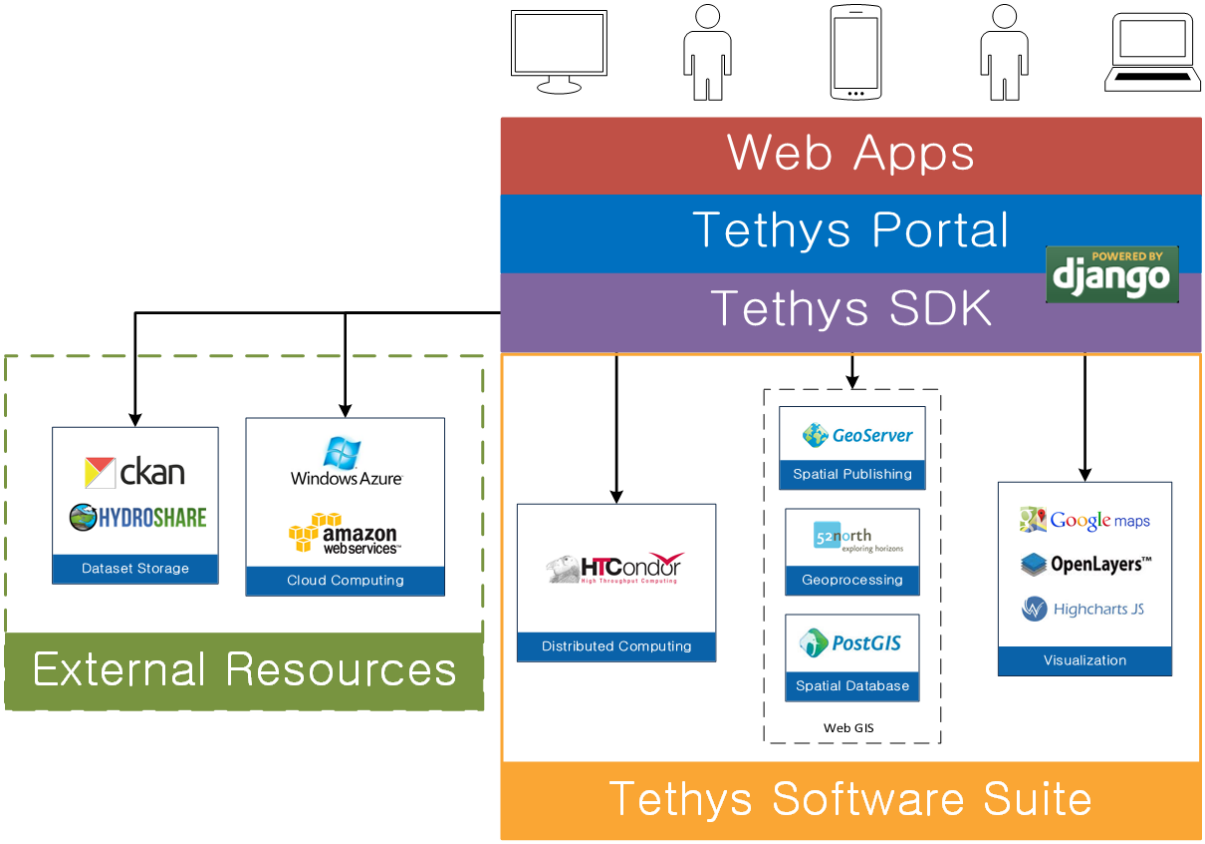
Tethys is a platform that can be used to develop and host engaging, interactive water resources web applications or web apps. It includes a suite of free and open source software (FOSS) that has been carefully selected to address the unique development needs of water resources web apps. Tethys web apps are developed using a Python software development kit (SDK) which includes programmatic links to each software component. Tethys Platform is powered by the Django Python web framework giving it a solid web foundation with excellent security and performance.



Tethys platform can be used to create engaging, interactive web apps for water resources.

1.1.1 Software Suite

Tethys Platform provides a suite of free and open source software. Included in the *Software Suite* is PostgreSQL with the PostGIS extension for spatial database storage, GeoServer for spatial data publishing, and 52 North WPS for geoprocessing. Tethys also provides Gizmos for inserting OpenLayers and Google Maps for interactive spatial data visualizations in your web apps. The *Software Suite* also includes HTCondor for managing distributed computing resources and scheduling computing jobs.



Tethys Platform include software to meet water resources web app development needs.

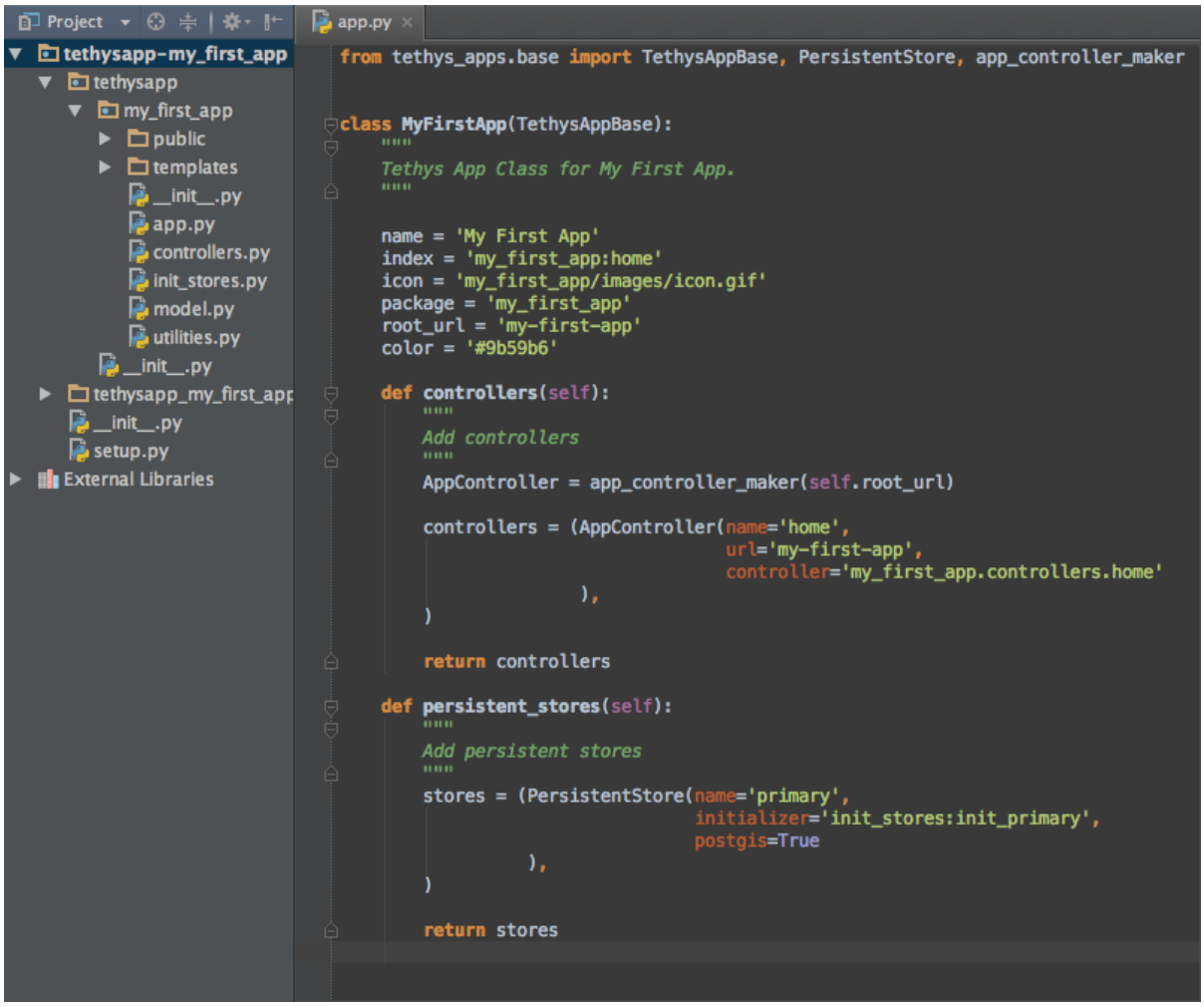
Note: Read more about the Software Suite by reading the *Software Suite* documentation.

1.1.2 Python Software Development Kit

Tethys web apps are developed with the Python programming language and a *Software Development Kit* (SDK). Tethys web apps projects are organized using a model-view-controller (MVC) approach. The SDK provides Python module links to each software component of the Tethys Platform, making the functionality of each software easy to incorporate each in your web apps. In addition, you can use all of the Python modules that you are accustomed to using in your scientific Python scripts to power your web apps.

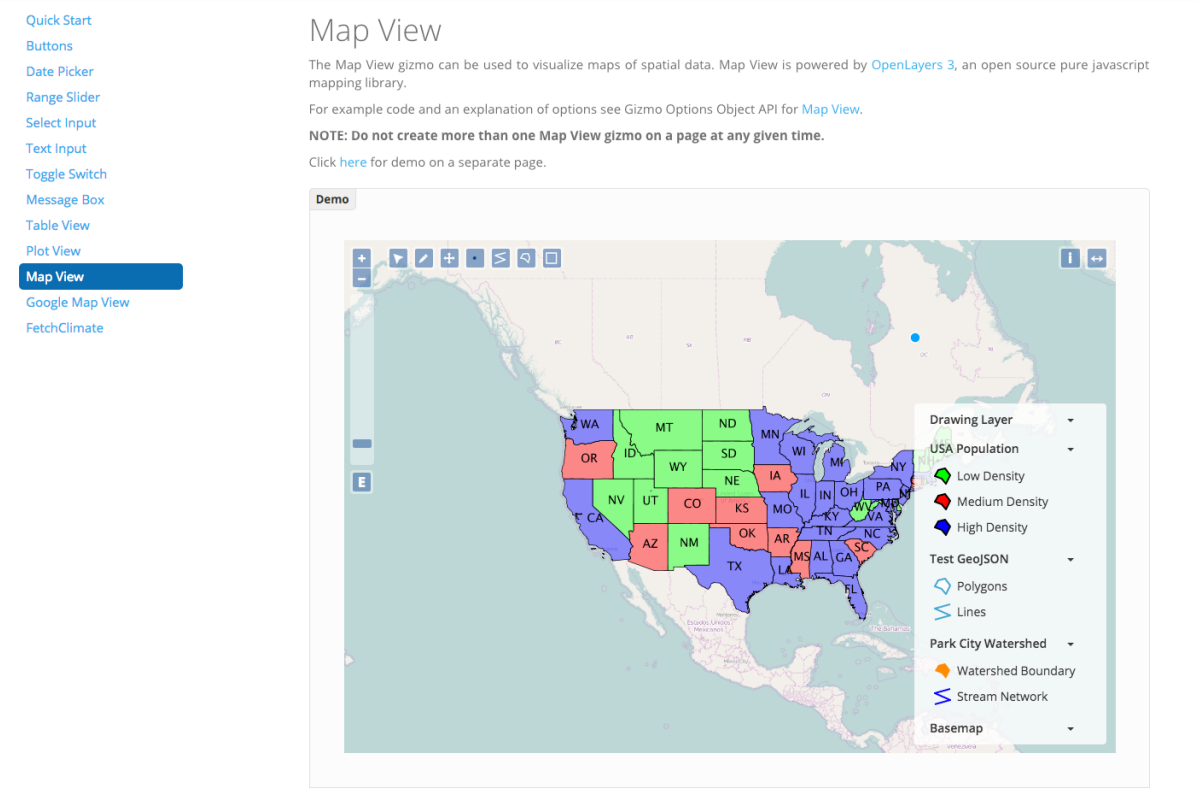
Tethys web apps are developed using Python and the Tethys SDK.

Note: Read more about the Tethys SDK by reading the *Software Development Kit* documentation.


Project
tethysapp-my_first_app
tethysapp
my_first_app
public
templates
__init__.py
app.py
controllers.py
init_stores.py
model.py
utilities.py
__init__.py
tethysapp_my_first_app
__init__.py
setup.py
External Libraries
app.py
from tethys_apps.base import TethysAppBase, PersistentStore, app_controller_maker
class MyFirstApp(TethysAppBase):
"""
Tethys App Class for My First App.
"""
name = 'My First App'
index = 'my_first_app:home'
icon = 'my_first_app/images/icon.gif'
package = 'my_first_app'
root_url = 'my-first-app'
color = '#9b59b6'
def controllers(self):
"""
Add controllers
"""
AppController = app_controller_maker(self.root_url)
controllers = (AppController(name='home',
url='my-first-app',
controller='my_first_app.controllers.home'
),
)
return controllers
def persistent_stores(self):
"""
Add persistent stores
"""
stores = (PersistentStore(name='primary',
initializer='init_stores:init_primary',
postgis=True
),
)
return stores

1.1.3 Templating and Gizmos

Tethys SDK takes advantage of the Django template system so you can build dynamic pages for your web app while writing less HTML. It also provides a series of modular user interface elements called Gizmos. With only a few lines of code you can add range sliders, toggle switches, auto completes, interactive maps, and dynamic plots to your web app.



Insert common user interface elements like date pickers, maps, and plots with minimal coding.

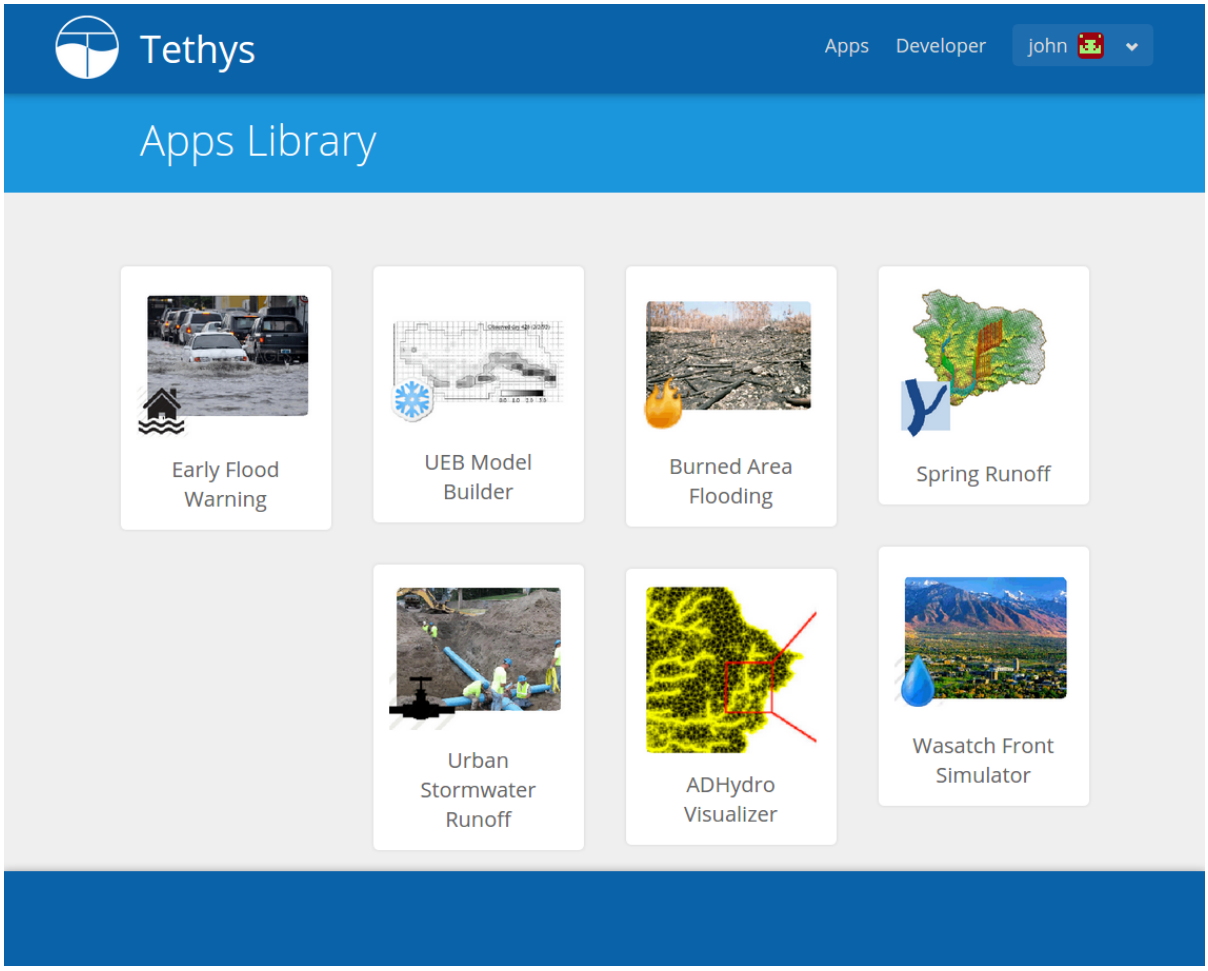
Note: Read more about templating and Gizmo by reading the *App Templating API* and the *Template Gizmos API* documentation.

1.1.4 Tethys Portal

Tethys Platform includes a modern web portal built on Django that is used to host web apps called *Tethys Portal*. It provides the core website functionality that is often taken for granted in modern web applications including a user account system with with a password reset mechanism for forgotten passwords. It provides an administrator backend that can be used to manage user accounts, permissions, link to elements of the software suite, and customize the instance.

The portal also includes landing page that can be used to showcase the capabilities of the Tethys Platform instance and an app library page that serves as the access point for installed apps. The homepage and theme of Tethys Portal are customizable allowing organizations to re-brand it to meet the their needs.

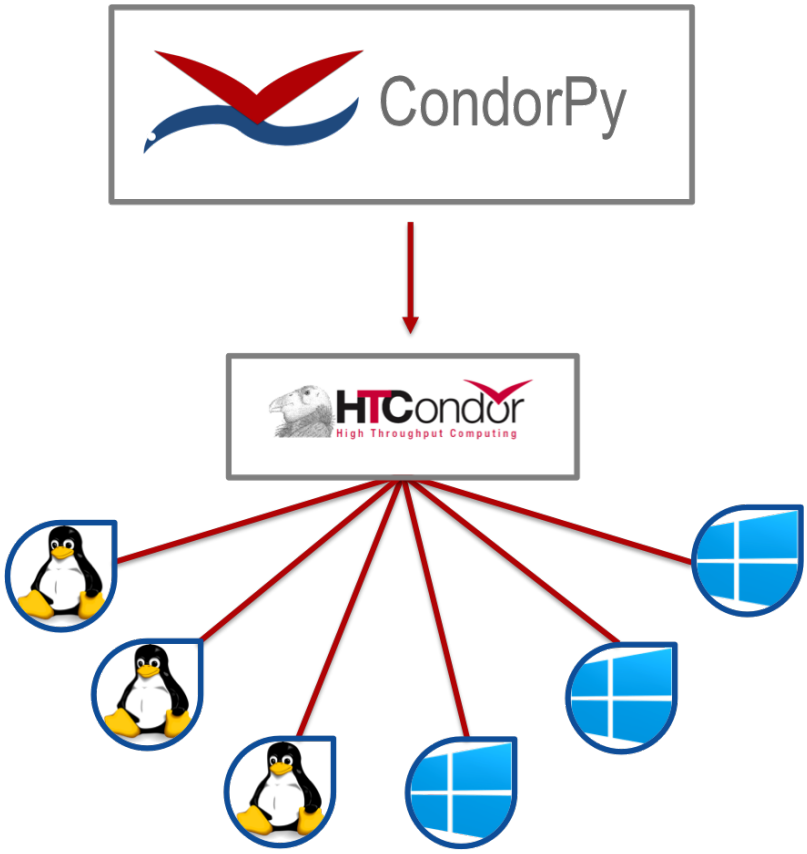
Browse available web apps using the Apps Library.


Tethys
Apps
Developer
john
Apps Library
Early Flood Warning
UEB Model Builder
Burned Area Flooding
Spring Runoff
Urban Stormwater Runoff
ADHydro Visualizer
Wasatch Front Simulator

Note: Read more about the Tethys Portal by reading the *Tethys Portal* documentation.

1.1.5 Computing

Tethys Platform includes Python modules that allow you to provision and run computing jobs in distributed computing environments. With CondorPy you can define your computing jobs and submit them to distributed computing environments provided by HTCondor.



CondorPy enables computing jobs to be created and submitted to a HTCondor computing pool.

HTCondor provides a way to make use of the idle computing power that is already available in your office.

Note: To learn more, read the *Jobs API* and the *Compute API*.

1.1.6 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1135482

1.2 What's New

Last Updated: December 2018

Refer to this article for information about each new release of Tethys Platform.

1.2.1 Release 2.1.3

Python 3 Support

- Python 3 officially supported in Tethys Platform.
- Python 2 support officially deprecated and will be dropped when Tethys Platform 3.0 is released.
- Tethys needs to migrate to Python 3 only so we can upgrade to Django 2.0, which only supports Python 3.

Important: Migrate your apps to Python 3. After Tethys Platform 3.0 is released, Python 2 will no longer be supported by Tethys Platform.

100% Unit Test Coverage

- Tests pass in Python 2 and Python 3.
- Unit tests cover 100% of testable code.
- Code base linted using flake8 to enforce PEP-8 and other Python coding best practices.
- Automated test execution on Travis-CI and Stickler-CI whenever a Pull Request is submitted.
- Added badges to the README to display build/testing, coverage, and docs status on github repository.
- All of this will lead to increased stability in this and future releases.

See: Tethys Platform Repo for build and coverage information.

Tethys Extensions

- Customize Tethys Platform functionality.
- Create your own gizmos.
- Centralize app logic that is common to multiple apps in an extension.

See: *Tethys Extensions API*

Map View Gizmo

- Added support for many more basemaps.
- Added Esri, Stamen, CartoDB.
- Support for custom XYZ services as basemaps.
- User can set OpenLayers version.
- Uses jsdelivr to load custom versions (see: https://cdn.jsdelivr.net/npm/openlayers)
- Default OpenLayers version updated to 5.3.0.

See: *Map View*

Class-based Controllers

- Added `TethysController` to SDK to support class-based views in Tethys apps.
- Inherits from django `View` class.
- Includes `as_controller` method, which is a thin wrapper around `as_view` method to better match Tethys terminology.
- UrlMaps can take class-based Views as the controller argument: `MyClassBasedController.as_controller(...)`
- More to come in the future.

See: Django Class-based views to get started.

Partial Install Options

- The Tethys Platform installation scripts now allow for partial installation.
- Install in existing Conda environment or against existing database.
- Upgrade using the install script!
- Linux and Mac only.

See: *Installation on Linux and Mac OSX* and *Upgrade to 2.1.3*

Commandline Interface

- New commands to manage app settings and services.
- `tethys app_settings` - List settings for an app.
- `tethys services` - List, create, and remove Tethys services (only supports persistent store services and spatial dataset services for now).
- `tethys link` - Link/Assign a Tethys service to a corresponding app setting.
- `tethys schedulers` - List, create, and remove job Schedulers.
- `tethys manage sync` - Sync app and extensions with Tethys database without a full Tethys start.

See: *app_settings <app_name>*, *services <subcommand> [<subsubcommand> | options]*, *link <service_identifier> <app_setting_identifier>*, *schedulers <subcommand>*, and *manage <subcommand> [options]*

Dockerfile

- New Dockerfile for Tethys Platform.
- Use it to build Docker images.
- Use it as a base for your own Docker images that have your apps installed.
- Includes supporting salt files.
- Dockerfile has been optimized to minimize the size of the produced image.
- Threading is enabled in the Docker container.

See: Docker Documentation to learn how to use Docker in your workflows.

API Tokens for Users

- API tokens are automatically generated for users when they are created.
- Use User API tokens to access protected REST API views.

Documentation

- Added SSL setup instruction to Production Installation (see: *4. Setup SSL (https) on the Tethys and Geoserver (Recommended)*)

Bugs

- Fixed grammar in forget password link.
- Refactored various methods and decorators to use new way of using Django methods `is_authenticated` and `is_anonymous`.
- Fixed bug with Gizmos that was preventing errors from being displayed when in debug mode.
- Fixed various bugs with uninstalling apps and extensions.
- Fixed bugs with get_persistent_store_setting methods.
- Fixed a naming conflict in the SelectInput gizmo.
- Fixed numerous bugs identified by new tests.

1.2.2 Prior Release Notes

Prior Release Notes

Last Updated: December 2017

Information about prior releases is shown here.

Release 2.0.0

Powered by Miniconda Environment

- Tethys Platform is now installed in a Miniconda environment.
- Using the Miniconda includes Conda, an open source Python package management system
- Conda can be used to install Python dependencies as well as system dependencies
- Installing packages like GDAL or NetCDF4 are as easy as `conda install gdal`
- Conda is cross platform: it works on Windows, Linux, and MacOS

See: Miniconda and Conda

Cross Platform Support

- Develop natively on Windows, Mac, or Linux!
- No more virtual machines.
- Be careful with your paths.

See: *Installation*

Installation Scripts

- Completely automated installation of Tethys
- Scripts provided for Mac, Linux, and Windows.

See: *Installation*

Python 3

- Experimental Python 3 Support in 2.0.0
- Tethys Dataset Services is not completely Python 3 compatible
- Use `--python-version 3` option on the installation script
- Python 2 support will be dropped in version 2.1

See: *Installation*

Templating API

- Leaner, updated theme for app base template.
- New `header_buttons` block for adding custom buttons to app header.

See: *App Templating API*

App Settings

- Developers can create App Settings, which are configured in the admin interface of the Tethys Portal.
- Types of settings that can be created include Custom Settings, Persistent Store Settings, Dataset Service Settings, Spatial Dataset Service Settings, and Web Processing Service Settings.
- The way Tethys Services are allocated to apps is now done through App Settings.
- All apps using the Persistent Stores APIs, Dataset Services APIs, or Web Processing Services APIs prior to version 2.0.0 will need to be refactored to use the new App settings approach.

See: *App Settings API*

Commandline Interface

- Added `tethys list` command that lists installed apps.
- Completely overhauled scaffold command that works cross-platform.
- New options for scaffold command that allow automatically accepting the defaults and overwriting project if it already exists.

See: *list* and *scaffold <name>*

Tutorials

- Brand new Getting Started Tutorial
- Demonstration of most Tethys SDK APIs

See: *Getting Started*

Gizmos

- New way to call them
- New load dependencies Method
- Updated select_gizmo to allow Select2 options to be passed in.

See: *Template Gizmos API*

Map View

- Updated OpenLayers libraries to version 4.0
- Fixes to make MapView compatible with Internet Explorer
- Can configure styling of MVDraw overlay layer
- New editable attribute for MVLayers to lock layers from being edited
- Added data attribute to MVLayer to allow passing custom attributes with layers for use in custom JavaScript
- A basemap switcher tool is now enabled on the map with the capability to configure multiple basemaps, including turning the basemap off.
- Added the ability to customize some styles of vector MVLayers.

See: *Map View*

Esri Map View

- New map Gizmo that uses ArcGIS for JavaScript API.

See: *ESRI Map*

Plotly View and Bokeh View Gizmos

- True open source options for plotting in Tethys

See: *Bokeh View* and *Plotly View*

DataTable View Gizmos

- Interactive table gizmo based on Data Tables.

See: *DataTable View*

Security

- Sessions will now timeout and log user out after period of inactivity.
- When user closes browser, they are automatically logged out now.
- Expiration times can be configured in settings.

See: *Tethys Platform Settings*

HydroShare OAuth Backend and Helper Function

- Refactor default HydroShare OAuth backend; Token refresh is available; Add backends for HydroShare-beta and HydroShare-playground.
- Include hs_restclient library in requirements.txt; Provide a helper function to help initialize the `hs` object based on HydroShare social account.
- Update python-social-auth to 0.2.21.

See: *Social Authentication*

Bugs

- Fixed issue where `tethys uninstall <app>` command was not uninstalling fully.

Release 1.4.0

App Permissions

- There is now a formalized mechanism for creating permissions for apps.
- It includes a *permission_required* decorator for controllers and a *has_permission* method for checking permissions within controllers.

Tags for Apps

- Apps can be assigned tags via the "tags" property in app.py.
- App tags can be overriden by portal admins using the `Installed Apps` settings in the admin portal.
- If there are more than 5 app tiles in the apps library, a list of buttons, one for each tag, will be displayed at the top of the Apps Library page.
- Clicking on one of the tag buttons, will filter the list of displayed apps to only those with the selected tag.

Terms and Conditions Management

- Portal Admins can now manage and enforce portal-wide terms and conditions and other legal documents.
- Documents are added via the admin interface of the portal.
- Documents can be versioned and dates at which they become active can be set.
- Once the date passes, all users will be prompted to accept the terms of the new documents.

GeoServer

- The GeoServer docker was updated to version 2.8.3
- It can be configured to run in clustered mode (multiple instances of GeoServer running inside the container) for greater stability and performance
- Several extensions are now included:
  - JMS Clustering
  - Flow Control
  - CSS Styles
  - NetCDF
  - NetCDF Output
  - GDAL WCS Output
  - Image Pyramid

Tethys Docker CLI

- Modified behaviour of "-c" option to accept a list of containers names so that commands can be performed on subsets of the containers
- Improved behaviour of "start" and "stop" commands such that they will start/stop all installed containers if some are not installed
- Improved behaviour of the "remove" command to skip containers that are not installed

Select2 Gizmo

- Updated the Select2 Gizmo libraries to version 4.0.
- Not changes should be necessary for basic usage of the Select2 Gizmo.
- If you are using advanced features of Select2, you will likely need to migrate some of your code.
- Refer to https://select2.github.io/announcements-4.0.html#migrating-from-select2-35 for migration help.

MapView Gizmo

- New JavaScript API endpoints for the MapView.
- Use the *TETHYS_MAP_VIEW.getSelectInteraction()* method to have more control over items that are selected.
- MVLayer Select Features now supports selection of vector layers in addition to the WMS Layers.
- Added support for images in the legend including support for GeoServer GetLegendGraphic requests.

PlotView Gizmo

- New JavaScript API endpoints for initializing PlotViews dynamically.

Workflow Job Type

- New Condor Workflow provides a way to run a group of jobs (which can have hierarchical relationships) as a single job.
- The hierarchical relationships are defined as parent-child relationships between jobs.
- As part of this addition the original Condor Job type was refactored and, while backwards compatibility is maintained in version 1.4, several aspects of how job templates are defined have been deprecated.

Testing Framework

- New Tethys CLI command to run tests on Tethys and apps.
- Tethys SDK now provides a TethysTestCase to streamlines app testing.
- Persistent stores is supported in testing.
- Tethys App Scaffold now includes testing module with example test code.

Installation

- Installation Instructions for Ubuntu 16.04

Bug Fixes

- Fixed an issue with URL mapping that was masking true errors with contollers (see: Issue #177)
- Fixed an issue with syncstores that use the string version of the path to the intializer function (see: Issue #185)
- Fixed an issue with syncstores that would cause it to fail the first time (see: Issue #194)

Release 1.3.0

Tethys Portal

- Open account signup disabled by default
- New setting in *settings.py* that allows open signup to be enabled

Map View

- Feature selection enabled for ImageWMS layers
- Clicking on features highlights them when enabled
- Callback functions can be defined in JavaScript to trap on the feature selection change event
- Custom styles can be applied to highlighted features
- Basemap can be disabled
- Layer attributes can be set in MVLayer (e.g. visibility and opacity)
- Updated to use OpenLayers 3.10.1

Plot View

- D3 plotting implemented as a free alternative to Highcharts for line plot, pie plot, scatter plot, bar plot, and timeseries plot.

Spatial Dataset Services

- Upgraded gsconfig dependency to version 1.0.0
- Provide two new methods on the geoserver engine to create SQL views and simplify the process of linking PostGIS databases with GeoServer.

App Feedback

- Places button on all app pages that activates a feedback form
- Sends app-users comments to specified developer emails
- Includes user and app specific information

Handoff

- Handoff Manager now available, which can be used from controllers to handoff from one app to another on the same Tethys portal (without having to use the REST API)
- The way handoff handler controllers are specified was changed to be consistent with other controllers

Jobs Table Gizmo

- The refresh interval for job status and runtime is configurable

Social Authentication

- Support for HydroShare added

Dynamic Persistent Stores

- Persistent stores can now be created dynamically (at runtime)
- Helper methods to list persistent stores for the app and check whether a store exists.

App Descriptions

- Apps now feature optional descriptions.
- An information icon appears on the app icon when descriptions are available.
- When the information icon is clicked on the description is shown.

Bugs

- Missing initial value parameter was added to the select and select2 gizmos.
- Addressed several cases of mixed content warnings when running behind HTTPS.
- The disconnect social account buttons are now disabled if your account doesn't have a password or there is only one social account associated with the account.
- Fixed issues with some of the documentation not being generated.
- Fixed styling issues that made the Message Box gizmo unusable.
- Normalized references to controllers, persistent store initializers, and handoff handler functions.
- Various docs typos were fixed.

Release 1.2.0

Social Authentication

- Social login supported
- Google, LinkedIn, and Facebook
- HydroShare coming soon
- New controls on User Profile page to manage social accounts

D3 Plotting Gizmos

- D3 alternatives for all the HighCharts plot views
- Use the same plot objects to define both types of charts
- Simplified and generalized the mechanism for declaring plot views

Job Manager Gizmo

- New Gizmo that will show the status of jobs running with the Job Manager

Workspaces

- SDK methods for creating and managing workspaces for apps
- List files and directories in workspace directory
- Clear and remove files and directories in workspace

Handoff

- Use handoff to launch one app from another
- Pass arguments via GET parameters that can be used to retrieve data from the sender app

Video Tutorials

- New video tutorials have been created
- The videos highlight working with different software suite elements
- CKAN, GeoServer, PostGIS
- Advanced user input forms
- Advanced Mapping and Plotting Gizmos

New Location for Tethys SDK

- Tethys SDK methods centralized to a new convenient package: tethys_sdk

Persistent Stores Changes

- Moved the get_persistent_stores_engine() method to the TethysAppBase class.
- To call the method import your *app class* and call it on the class.
- The old get_persistent_stores_engine() method has been flagged for deprecation.

Command Line Interface

- New management commands including `createsuperuser`, `collectworkspaces`, and `collectall`
- Modified behavior of `syncdb` management command, which now makes and then applies migrations.

Release 1.1.0

Gizmos

- Options objects for configuring gizmos
- Many improvements to Map View
  - Improved layer support including GeoJSON, KML, WMS services, and ArcGIS REST services
  - Added a mechanism for creating legends
  - Added drawing capabilities
  - Upgraded to OpenLayers version 3.5.0
- New objects for simplifying Highcharts plot creation
  - HighChartsLinePlot
  - HighChartsScatterPlot
  - HighChartsPolarPlot
  - HighChartsPiePlot
  - HighChartsBarPlot
  - HighChartsTimeSeries
  - HighChartsAreaRange
- Added the ability to draw a box on Google Map View

Tethys Portal Features

- Reset forgotten passwords
- Bypass the home page and redirect to apps library
- Rename the apps library page title
- The two mobile menus were combined into a single mobile menu
- Dataset Services and Web Processing Services admin settings combined into a single category called Tethys Services
- Added "Powered by Tethys Platform" attribution to footer

Job Manager

- Provides a unified interface for all apps to create submit and monitor computing jobs
- Abstracts the CondorPy module to provide a higher-level interface with computing jobs
- Allows definition of job templates in the app.py module of apps projects

Documentation Updates

- Added documentation about the Software Suite and the relationship between each software component and the APIs in the SDK is provided
- Documentation for manual download and installation of Docker images
- Added system requirements to documentation

Bug Fixes

- Naming new app projects during scaffolding is more robust
- Fixed bugs with fetch climate Gizmo
- Addressed issue caused by usernames that included periods (.) and other characters
- Made header more responsive to long names to prevent header from wrapping and obscuring controls
- Fixed bug with tethys gen apache command
- Addressed bug that occurred when naming WPS services with uppercase letters

Other

- Added parameter of UrlMap that can be used to specify custom regular expressions for URL search patterns
- Added validation to service engines
- Custom collectstatic command that automatically symbolically links the public/static directories of Tethys apps to the static directory
- Added "list" methods for dataset services and web processing services to allow app developers to list all available services registered on the Tethys Portal instance

1.3 Installation

Last Updated: July 1, 2016

This section describes how to install Tethys Platform. Installation instructions are provided for Ubuntu.

1.3.1 System Requirements

Last Updated: April 20, 2015

Use these guidelines as a starting point for installing Tethys Platform as a stand alone environment:

- Processor: 4 CPU Cores
- RAM: 4 GB
- Hard Disk: 10 GB

Caution: The stand alone configuration should be used primarily for development purposes. It is not recommended that you use a stand alone configuration for production installations. See the *Production Installation* documentation for system requirements of a production installation.

1.3.2 Installation on Linux and Mac OSX

Last Updated: April 2017

Use these instructions to install a development environment on Mac OSX or Linux systems.

Tip: To install and use Tethys Platform, you will need to be familiar with using the command line/terminal. For a quick introduction to the command line, see the *Terminal Quick Guide* article.

1. Download and Run the Installation Script

Run the following commands from a terminal to download and run the Tethys Platform install script.

For systems with *wget* (most Linux distributions):

```
wget https://raw.githubusercontent.com/tethysplatform/tethys/release/scripts/
install_tethys.sh
bash install_tethys.sh -b release
```

For Systems with *curl* (e.g. Mac OSX and CentOS):

```
curl https://raw.githubusercontent.com/tethysplatform/tethys/release/scripts/
install_tethys.sh -o ./install_tethys.sh
bash install_tethys.sh -b release
```

Install Script Options

You can customize your tethys installation by passing command line options to the installation script. The available options can be listed by running:

```
$ bash install_tethys.sh --help
```

Each option is also descriped here:

- ***-t, --tethys-home <PATH>*:** Path for tethys home directory. Default is ~/tethys.
- ***-s, --tethys-src <PATH>*:** Path to the tethys source directory. Default is ${TETHYS_HOME}/src.
- ***-a, --allowed-host <HOST>*:** Hostname or IP address on which to serve Tethys. Default is 127.0.0.1.
- ***-p, --port <PORT>*:** Port on which to serve Tethys. Default is 8000.
- ***-b, --branch <BRANCH_NAME>*:** Branch to checkout from version control. Default is 'release'.
- ***-c, --conda-home <PATH>*:** Path to conda home directory where Miniconda will be installed, or to an existing installation of Miniconda. Default is ${TETHYS_HOME}/miniconda.

  Tip: The conda home path cannot contain spaces. If the the tethys home path contains spaces then the *--conda-home* option must be specified and point to a path without spaces.

- ***-n, --conda-env-name <NAME>*:** Name for tethys conda environment. Default is 'tethys'.
- ***--python-version <PYTHON_VERSION>* (deprecated):** Main python version to install tethys environment into (2-deprecated or 3). Default is 3. .. note:

  ```
  Support for Python 2 is deprecated and will be dropped in Tethys␣
  ↪version 3.0.
  ```

- ***--db-username <USERNAME>*:** Username that the tethys database server will use. Default is 'tethys_default'.
- ***--db-password <PASSWORD>*:** Password that the tethys database server will use. Default is 'pass'.
- ***--db-super-username <USERNAME>*:** Username for super user on the tethys database server. Default is 'tethys_super'.
- ***--db-super-password <PASSWORD>*:** Password for super user on the tethys database server. Default is 'pass'.
- ***--db-port <PORT>*:** Port that the tethys database server will use. Default is 5436.
- ***--db-dir <PATH>*:** Path where the local PostgreSQL database will be created. Default is ${TETHYS_HOME}/psql.
- ***-S, --superuser <USERNAME>*:** Tethys super user name. Default is 'admin'.
- ***-E, --superuser-email <EMAIL>*:** Tethys super user email. Default is ''.
- ***-P, --superuser-pass <PASSWORD>*:** Tethys super user password. Default is 'pass'.
- ***--skip-tethys-install*:** Flag to skip the Tethys installation so that the Docker installation or production installation can be added to an existing Tethys installation.

Tip: If conda home is not in the default location then the *--conda-home* options must also be specified with this option.

- ***--partial-tethys-install <FLAGS>*:** List of flags to indicate which steps of the installation to do.

  Flags:

  - *m* - Install Miniconda
  - *r* - Clone Tethys repository (the *--tethys-src* option is required if you omit this flag).
  - *c* - Checkout the branch specified by the option *--branch* (specifying the flag *r* will also trigger this flag)
  - *e* - Create Conda environment
  - *s* - Create *settings.py* file
  - *d* - Create a local database server
  - *i* - Initialize database server with the Tethys database (specifying the flag *d* will also trigger this flag)
  - *u* - Add a Tethys Portal Super User to the user database (specifying the flag *d* will also trigger this flag)
  - *a* - Create activation/deactivation scripts for the Tethys Conda environment
  - *t* - Create the *t* alias to activate the Tethys Conda environment

  For example, if you already have Miniconda installed and you have the repository cloned and have generated a *settings.py* file, but you want to use the install script to:

  - create a conda environment,
  - setup a local database server,
  - create the conda activation/deactivation scripts, and
  - create the *t* shortcut,

  then you can run the following command:

```
bash install_tethys.sh --partial-tethys-install edat
```

Warning: If *--skip-tethys-install* is used then this option will be ignored.

- ***--install-docker*:** Flag to include Docker installation as part of the install script (Linux only). See *2. Install Docker (OPTIONAL)* for more details.
- ***--docker-options <OPTIONS>*:** Command line options to pass to the *tethys docker init* call if --install-docker is used. Default is "'-d'".

Tip: The value for the *--docker-options* option must have nested quotes. For example "'-d -c geoserver'" or '"-d -c geoserver"'.

- ***--production*** Flag to install Tethys in a production configuration.

- ***--configure-selinux*** Flag to perform configuration of SELinux for production installation. (Linux only).
- ***-x*:** Flag to turn on shell command echoing.
- ***-h, --help*:** Print this help information.

Here is an example of calling the installation script with customized options:

```
$ bash install_tethys.sh -t ~/Workspace/tethys -a localhost -p 8005 -c ~/
↪miniconda3 --db-username tethys_db_user --db-password db_user_pass --db-
↪port 5437 -S tethys -E email@example.com -P tpass
```

The installation script may take several minutes to run. Once it is completed you will need to activate the new conda environment so you can start the Tethys development server. This is most easily done using an alias created by the install script. To enable the alias you need to open a new terminal or re-run the `.bashrc` (Linux) or `.bash_profile` (Mac) file.

For Linux:

```
$ . ~/.bashrc
```

For Mac:

```
   $ . ~/.bash_profile

You can then activate the Tethys conda environment and start the Tethys development␣
↪server by running::

   $ t
   (tethys) $ tethys manage start
```

or simply just:

```
$ t
(tethys) $ tms
```

Tip: The installation script adds several environmental variables and aliases to help make using Tethys easier. Most of them are active only while the tethys conda environment is activated, however one alias to activate the tethys conda environment was added to your *.bashrc* or *bash_profile* file in your home directory and should be available from any terminal session:

- *t*: Alias to activate the tethys conda environment. It is a shortcut for the command *source <CONDA_HOME>/bin/activate tethys* where <CONDA_HOME> is the value of the *--conda-home* option that was passed to the install script.

The following environmental variables are available once the tethys conda environment is activated:

- ***TETHYS_HOME*:** The directory where the Tethys source code and other Tethys resources are. It is set from the value of the *--tethys-home* option that was passed to the install script.
- ***TETHYS_PORT*:** The port that the Tethys development server will be served on. Set from the *--port* option.
- ***TETHYS_DB_PORT*:** The port that the Tethys local database server is running on. Set from the *--db-port* option.

Also, the following aliases are available:

- ***tethys_start_db*:** Starts the local Tethys database server
- ***tstartdb*:** Another alias for *tethys_start_db*

- ***tethys_stop_db*:** Stops the localTethys database server
- ***tstopdb*:** Another alias for *tethys_stop_db*
- ***tms*:** An alias to start the Tethys development server. It calls the command *tethys manage start -p <HOST>:${TETHYS_PORT}* where *<HOST>* is the value of the *--allowed-host* option that was passed to the install script and *${TETHYS_PORT}* is the value of the environmental variable which is set from the *--port* option of the install script.
- ***tstart*:** Combines the *tethys_start_db* and the *tms* commands.

When installing Tethys in production mode the following additional environmental variables and aliases are added:

- ***NGINX_USER*:** The name of the Nginx user.
- ***NGINX_HOME*:** The home directory of the Nginx user.
- ***tethys_user_own*:** Changes ownership of relevant files to the current user by running the command *sudo chown -R ${USER} ${TETHYS_HOME}/src ${NGINX_HOME}/tethys*.
- ***tuo*:** Another alias for *tethys_user_own*
- ***tethys_server_own*:** Reverses the effects of *tethys_user_own* by changing ownership back to the Nginx user.
- ***tso*:** Another alias for *tethys_server_own*

  When you start up a new terminal there are three steps to get the Tethys development server running again:

  1. Activate the Tethys conda environment
  2. Start the Tethys database server
  3. start the Tethys development server

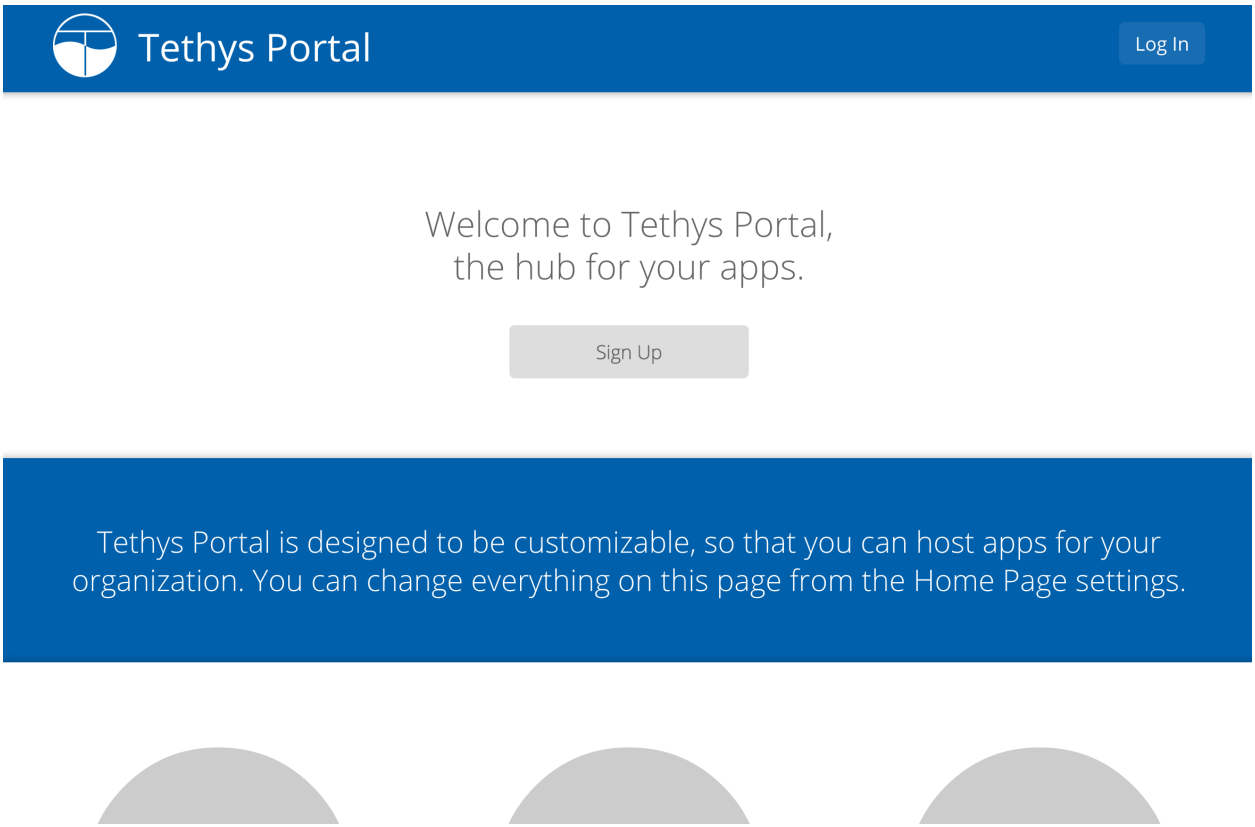
Using the supplied aliases, starting the Tethys development server from a fresh terminal can be done with the following two commands:

```
$ t
(tethys) $ tstart
```

Congratulations! You now have Tethys Platform running a in a development server on your machine. Tethys Platform provides a web interface that is called the Tethys Portal. You can access your Tethys Portal by opening http://localhost:8000/ (or if you provided custom host and port options to the install script then it will be *<HOST>:<PORT>*) in a new tab in your web browser.

To log in, use the credentials that you specified with the *-S* or *--superuser* and the *-P* or *--superuser-pass* options. If you did not specify these options then the default credentials are:

- username: *admin*
- password: *pass*



2. Install Docker (OPTIONAL)

To facilitate leveraging the full capabilities of Tethys Platform Docker containers are provided to allow the *Software Suite* to be easily installed. To use these containers you must first install Docker. The Tethys installation script `install_tethys.sh` will support installing the community edition of Docker on several Linux distributions. To install Docker when installing Tethys then add the *--install-docker* option. You can also add the *--docker-options* options to pass options to the *tethys docker init* command (see the *docker <subcommand> [options]* section of the *Command Line Interface* documentation).

To install Docker on other systems or to install the enterprise edition of Docker please refer to the Docker installation documentation

Use the following Tethys command to start the Docker containers.

```
tethys docker start
```

You are now ready to link your Tethys Portal with the Docker containers using the web admin interface. Follow the *Web Admin Setup* tutorial to finish setting up your Tethys Platform.

If you would like to test the Docker containers, see *Test Docker Containers*.

3. Customize Settings (OPTIONAL)

The Tethys installation script created a settings file called `settings.py` in the directory `$TETHYS_HOME/src/tethys_apps`. The installation script has defined the most essential settings that will allow the Tethys development server to function based on the options that were passed to the script or based on the default values of those options. If you would like to further customize the settings then open the `settings.py` file and make any desired changes. Refer to the *Tethys Platform Settings* documentation for a description of each of the settings.

1.3.3 Installation on Windows

Last Updated: April 2017

Use these instructions to install a development environment on Windows systems.

1. Download the Installation Script and the Miniconda Installation Executable

a. Download the Tethys installation batch script by right clicking on the following link and selecting *Save link as...*: install_tethys.bat

b. Download the Miniconda installer from the Conda site or by clicking on the following link: https://repo.continuum.io/miniconda/Miniconda3-latest-Windows-x86_64.exe

2. Run the Tethys Installation Batch Script

As long as the `install_tethys.bat` and the `Miniconda3-latest-Windows-x86_64.exe` files are in the same directory you can simply double click the `install_tethys.bat` to perform a default installation. To pass in custom options to the installation scrip you must run the script for the command prompt:

```
install_tethys.bat -b release
```

Note: You can customize your tethys installation by passing command line options to the installation script. The available options can be listed by running:

```
> install_tethys.bat --help
```

Each option is also descriped here:

- ***-t, --tethys-home <PATH>*:** Path for tethys home directory. Default is C:%HOMEPATH%tethys.
- ***-s, --tethys-src <PATH>*:** Path for tethys source directory. Default is %TETHYS_HOME%src.
- ***-a, --allowed-host <HOST>*:** Hostname or IP address on which to serve Tethys. Default is 127.0.0.1.
- ***-p, --port <PORT>*:** Port on which to serve Tethys. Default is 8000.
- ***-b, --branch <BRANCH_NAME>*:** Branch to checkout from version control. Default is 'release'.
- ***-c, --conda-home <PATH>*:** Path to conda home directory where Miniconda will be installed, or to an existing installation of Miniconda. Default is %TETHYS_HOME%miniconda.

  Tip: The conda home path cannot contain spaces. If the the tethys home path contains spaces then the *--conda-home* option must be specified and point to a path without spaces.

- ***-C, --conda-exe <PATH>*:** Path to Miniconda installer executable. Default is '.Miniconda3-latest-Windows-x86_64.exe'.

- ***-n, --conda-env-name <NAME>*:** Name for tethys conda environment. Default is 'tethys'.
- ***--python-version <PYTHON_VERSION>* (deprecated):** Main python version to install tethys environment into (2-deprecated or 3). Default is 3. .. note:

```
Support for Python 2 is deprecated and will be dropped in Tethys version 3.0.
```

- ***--db-username <USERNAME>*:** Username that the tethys database server will use. Default is 'tethys_default'.
- ***--db-password <PASSWORD>*:** Password that the tethys database server will use. Default is 'pass'.
- ***--db-port <PORT>*:** Port that the tethys database server will use. Default is 5436.
- ***-S, --superuser <USERNAME>*:** Tethys super user name. Default is 'admin'.
- ***-E, --superuser-email <EMAIL>*:** Tethys super user email. Default is ''.
- ***-P, --superuser-pass <PASSWORD>*:** Tethys super user password. Default is 'pass'.
- ***-x*:** Flag to echo all commands.
- ***-h, --help*:** Print this help information.

Here is an example of calling the installation script with customized options:

```
> install_tethys.bat -t C:\tethys -a localhost -p 8005 -c C:\Miniconda3 --db-username␣
↪tethys_db_user --db-password db_user_pass --db-port 5437 -S tethys -E email@example.
↪com -P tpass
```

The installation script may take several minutes to run. Once it is completed the new conda environment will be left activated so you can start the Tethys development server by running:

```
(tethys) > tethys manage start
```

or simply just:

```
(tethys) > tms
```

Tip: The installation script adds several environmental variables and aliases to help make using Tethys easier, which are active only while the tethys conda environment is activated. To facilitate activating the environment a batch file was added to the TETHYS_HOME directory called `tethys_cmd.bat`. Double clicking that file will open a command prompt with the tethys conda environment activated.

The following environmental variables are available once the tethys conda environment is activated:

- ***TETHYS_HOME*:** The directory where the Tethys source code and other Tethys resources are. It is set from the value of the *--tethys-home* option that was passed to the install script.
- ***TETHYS_PORT*:** The port that the Tethys development server will be served on. Set from the *--port* option.
- ***TETHYS_DB_PORT*:** The port that the Tethys local database server is running on. Set from the *--db-port* option.

Also, the following aliases are available:

- ***tethys_start_db*:** Starts the local Tethys database server
- ***tstartdb*:** Another alias for *tethys_start_db*
- ***tethys_stop_db*:** Stops the localTethys database server
- ***tstopdb*:** Another alias for *tethys_stop_db*

- ***tms*:** An alias to start the Tethys development server. It calls the command *tethys manage start -p <HOST>:${TETHYS_PORT}* where *<HOST>* is the value of the *--allowed-host* option that was passed to the install script and *${TETHYS_PORT}* is the value of the environmental variable which is set from the *--port* option of the install script.
- ***tstart*:** Combines the *tethys_start_db* and the *tms* commands.

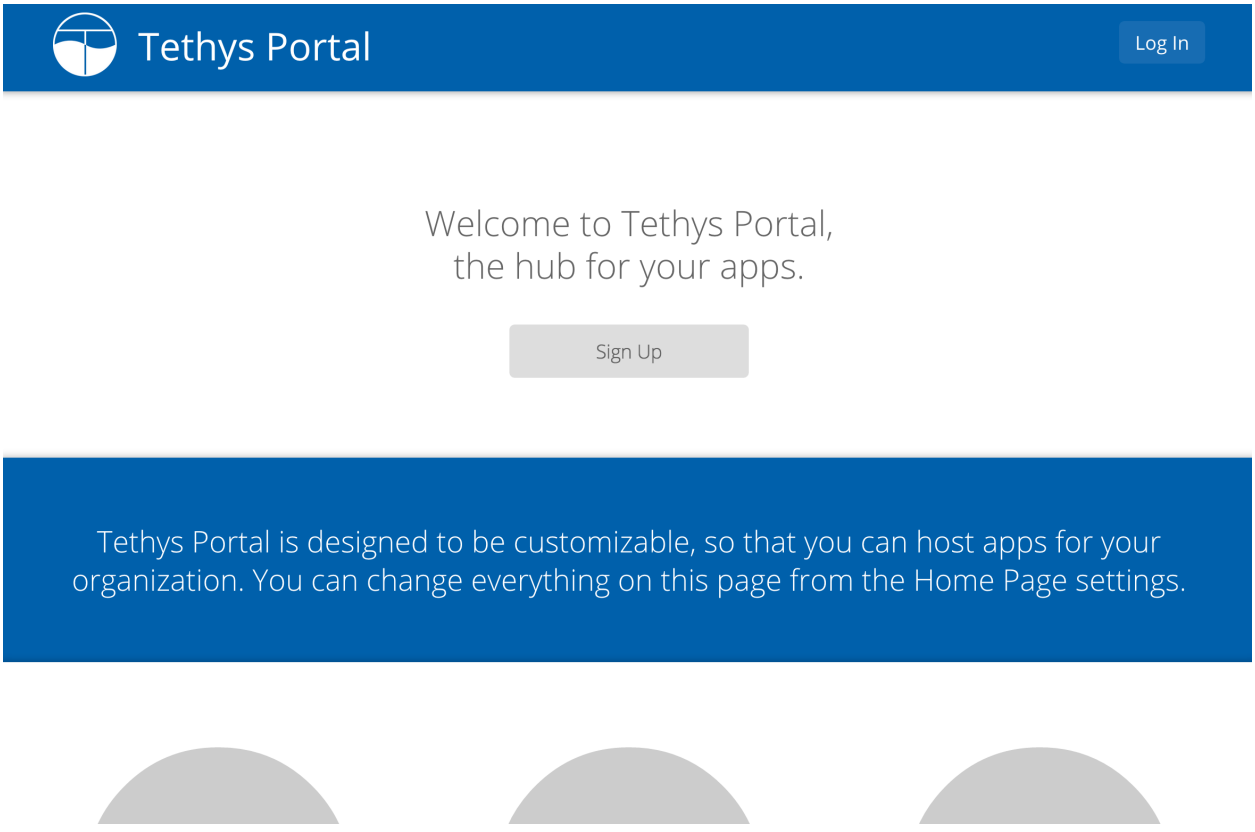
  When you start up a new terminal there are three steps to get the Tethys development server running again:

  1. Activate the Tethys conda environment
  2. Start the Tethys database server
  3. start the Tethys development server

Using the supplied aliases, starting the Tethys development server can be done by running the `tethys_cmd.bat` file and then executing the following command:

```
(tethys) > tstart
```

Congratulations! You now have Tethys Platform running a in a development server on your machine. Tethys Platform provides a web interface that is called the Tethys Portal. You can access your Tethys Portal by opening http://localhost:8000/ (or if you provided custom host and port options to the install script then it will be *<HOST>:<PORT>*) in a new tab in your web browser.



To log in, use the credentials that you specified with the *-S* or *--superuser* and the *-P* or *--superuser-pass* options. If you did not specify these options then the default credentials are:

- username: *admin*
- password: *pass*

2. Install Docker (OPTIONAL)

To facilitate leveraging the full capabilities of Tethys Platform Docker containers are provided to allow the *Software Suite* to be easily installed. To use these containers you must first install Docker. To install Docker on Windows please refer to the Docker installation documentation

Use the following Tethys command to start the Docker containers.

```
tethys docker start
```

You are now ready to link your Tethys Portal with the Docker containers using the web admin interface. Follow the *Web Admin Setup* tutorial to finish setting up your Tethys Platform.

If you would like to test the Docker containers, see *Test Docker Containers*.

3. Customize Settings (OPTIONAL)

The Tethys installation script created a settings file called `settings.py` in the directory `$TETHYS_HOME/src/tethys_apps`. The installation script has defined the most essential settings that will allow the Tethys development server to function based on the options that were passed to the script or based on the default values of those options. If you would like to further customize the settings then open the `settings.py` file and make any desired changes. Refer to the *Tethys Platform Settings* documentation for a description of each of the settings.

1.3.4 Tethys Platform Settings

Note: COMING SOON: An explanation of all the settings in setting.py!

Todo: describe each of the settings in `settings.py`

1.3.5 Web Admin Setup

Last Updated: February 2, 2015

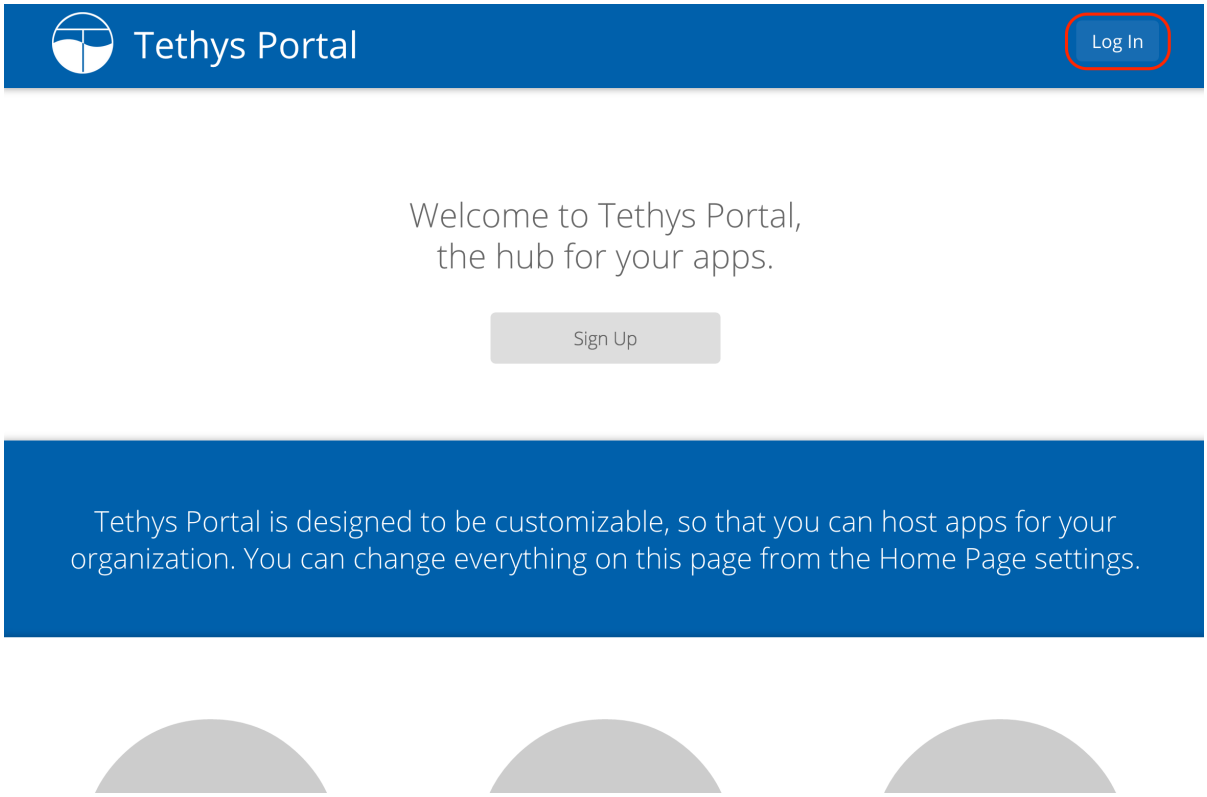
The final step required to setup your Tethys Platform is to link it to the software that is running in the Docker containers. This is done using the Tethys Portal Admin console.

1. Access Tethys Portal Admin Console

The Tethys Portal Admin Console is only accessible to users with administrator rights. When you installed Tethys Platform, you created superuser. Use these credentials to log in for the first time.

Tip: When installing Tethys with the installation script the superuser credentials would have been specified with the *-S* or *--superuser* and the *-P* or *--superuser-pass* options. If you did not specify these options then the default username and password is *admin* and *pass*.

a. Use the "Log In" link on the Tethys Portal homepage to log in as an administrator.



b. Select "Site Admin" from the user drop down menu.

You will now see the Tethys Portal Web Admin Console. The Web Admin console can be used to manage user accounts, customize the homepage of your Tethys Portal, and configure the software included in Tethys Platform. Take a moment to familiarize yourself with the different options that are available in the Web Admin.

2. Link to 52 North WPS Docker

The built in 52 North Web Processing Service (WPS) is provided as one mechanism for Geoprocessing in apps. It exposes the GRASS GIS and Sextante geoprocessing libraries as web services. See *Web Processing Services API* documentation for more details about how to use 52 North WPS processing in apps. Complete the following steps to link Tethys with the 52 North WPS:

a. Select "Web Processing Services" from the options listed on the Tethys Portal Admin Console.

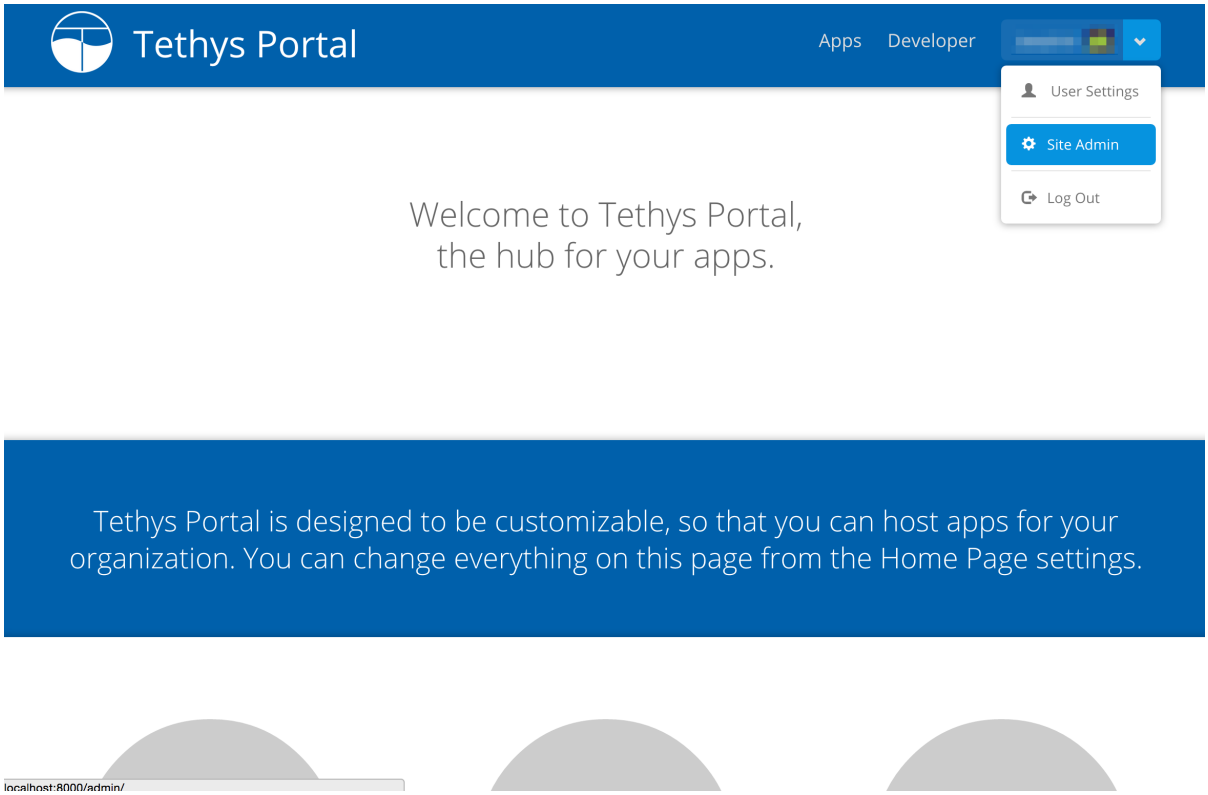
b. Click on the "Add Web Processing Service" button to create a new link to the web processing service.

c. Provide a unique name for the web processing service.

d. Provide an endpoint to the 52 North WPS that is running in Docker. The endpoint is a URL pointing to the WPS API. The endpoint will be of the form:

```
http://<host>:<port>/wps/WebProcessingService
```

Execute the following command in the terminal to determine the endpoint for the built-in 52 North server:




```
(tethys)$ tethys docker ip
...
52 North WPS:
  Host: 192.168.59.103
  Port: 8282
  Endpoint: http://192.168.59.103:8282/wps/WebProcessingService
```

When you are done you will have something similar to this:

e. Press "Save" to save the WPS configuration.

3. Link to GeoServer

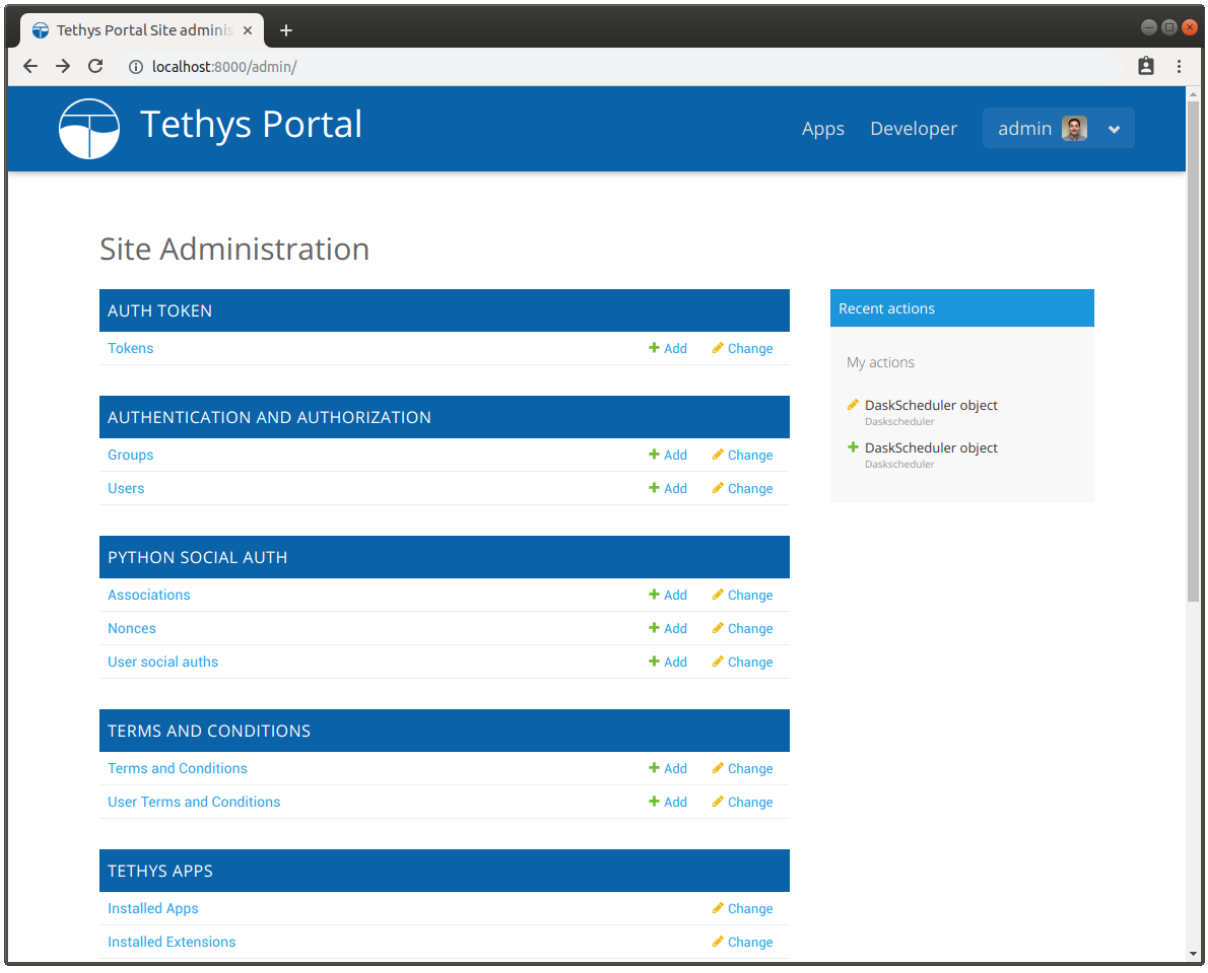
Tethys Platform provides GeoServer as a built-in Spatial Dataset Service. Spatial Dataset Services can be used by apps to publish Shapefiles and other spatial files as web resources. See *Spatial Dataset Services API* documentation for how to use Spatial Dataset Services in apps. To link your Tethys Platform to the built-in GeoServer or an external Spatial Dataset Service, complete the following steps:

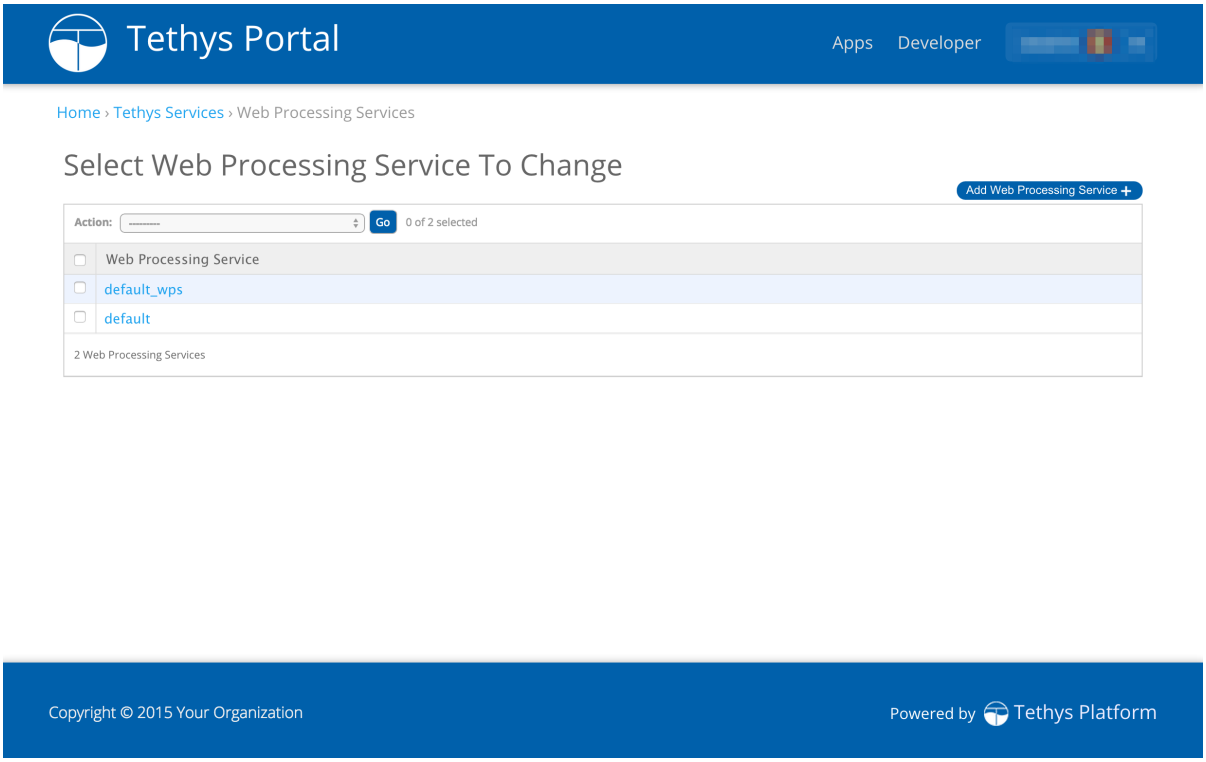
a. Select "Spatial Dataset Services" from the options listed on the Tethys Portal Admin Console.

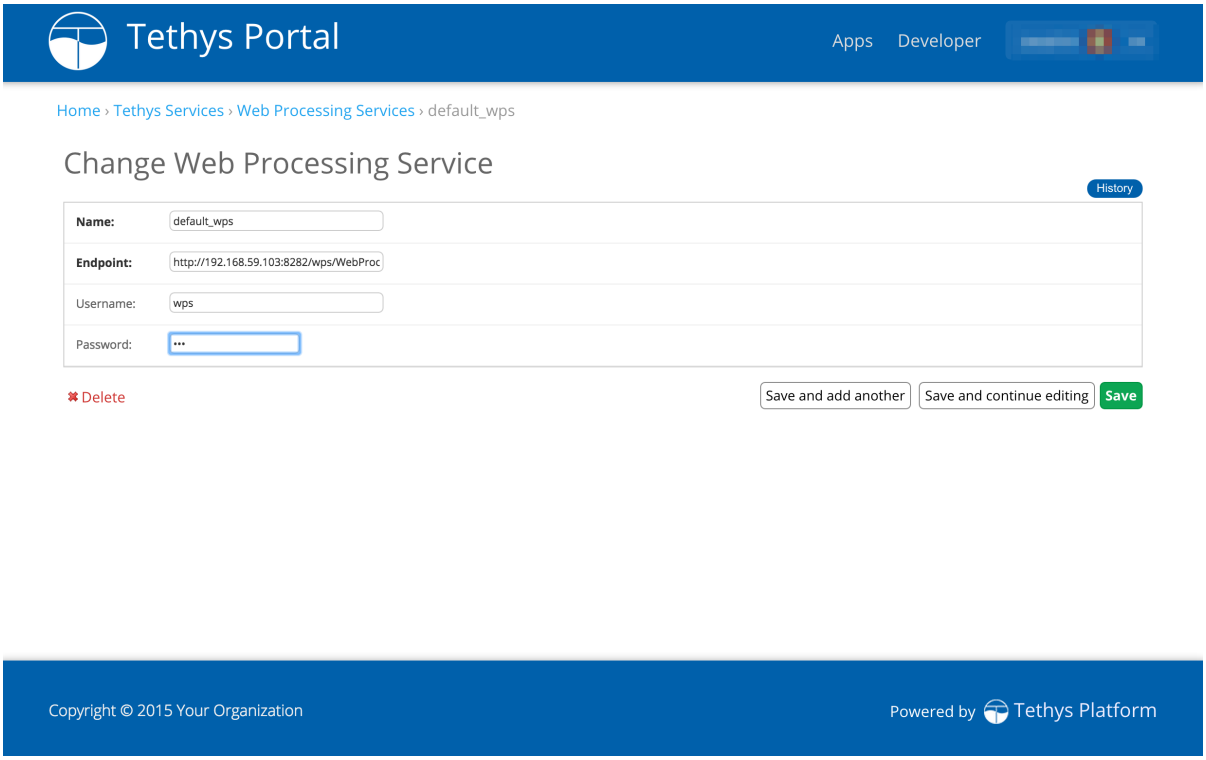
b. Click on the "Add Spatial Dataset Service" button to create a new spatial dataset service.

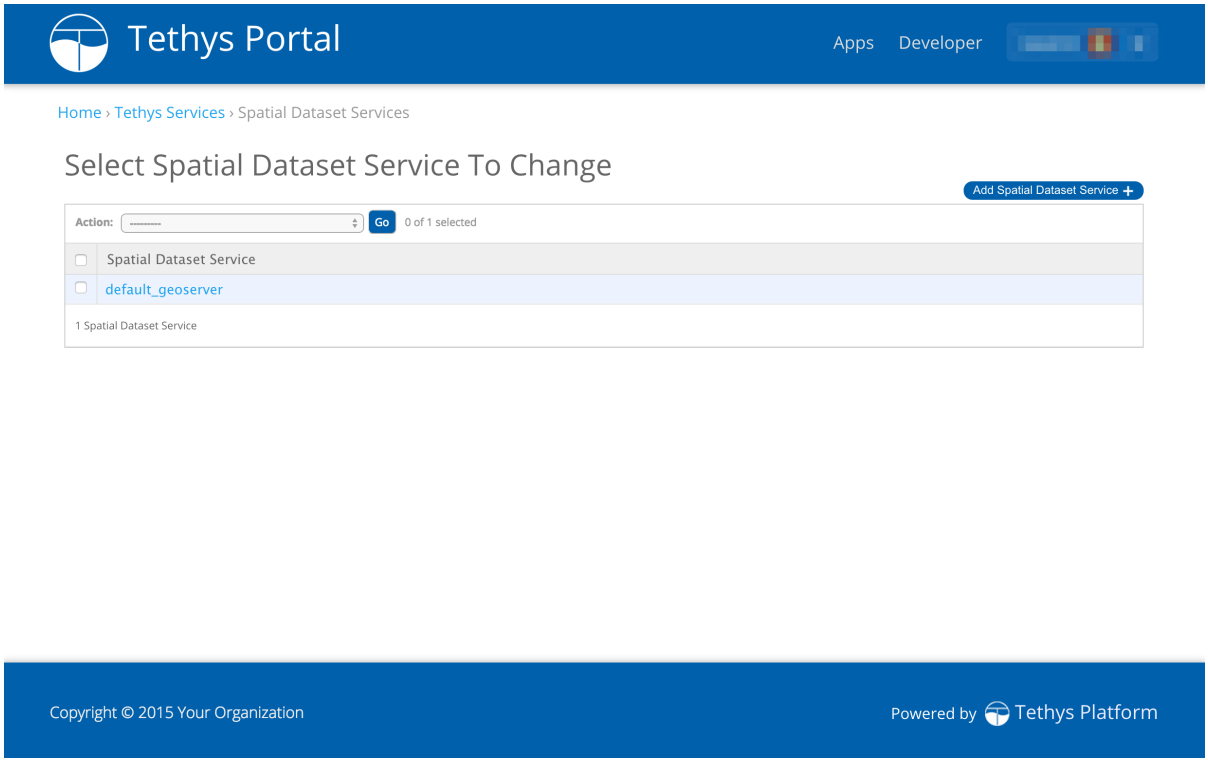
c. Provide a unique name for the spatial dataset service.

d. Select *"GeoServer"* as the engine and provide an endpoint to the Spatial Dataset Service. The endpoint is a URL pointing to the API of the Spatial Dataset Service. For GeoServers, this endpoint is of the form:


Tethys Portal Site adminis
localhost:8000/admin/
Tethys Portal
Apps
Developer
admin
Site Administration
AUTH TOKEN
Tokens
Add
Change
AUTHENTICATION AND AUTHORIZATION
Groups
Users
PYTHON SOCIAL AUTH
Associations
Nonces
User social auths
TERMS AND CONDITIONS
Terms and Conditions
User Terms and Conditions
TETHYS APPS
Installed Apps
Installed Extensions
Recent actions
My actions
DaskScheduler object
Dasksceduler


Tethys Portal
Apps
Developer
Home › Tethys Services › Web Processing Services
Select Web Processing Service To Change
Add Web Processing Service +
Action:
Go
0 of 2 selected
Web Processing Service
default_wps
default
2 Web Processing Services
Copyright © 2015 Your Organization
Powered by Tethys Platform


Tethys Portal
Apps
Developer
Home › Tethys Services › Web Processing Services › default_wps
Change Web Processing Service
History
Name:
default_wps
Endpoint:
http://192.168.59.103:8282/wps/WebProc
Username:
wps
Password:
Delete
Save and add another
Save and continue editing
Save
Copyright © 2015 Your Organization
Powered by Tethys Platform



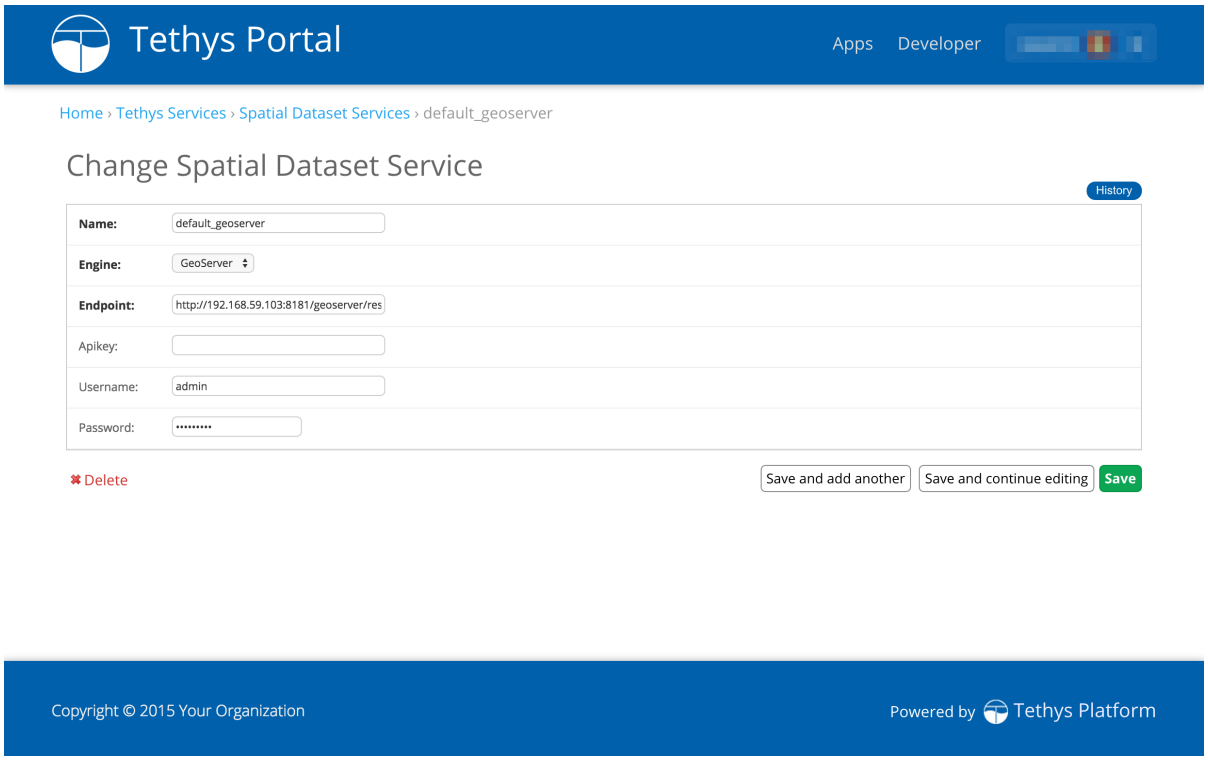

```
http://<host>:<port>/geoserver/rest
```

Execute the following command in the terminal to determine the endpoint for the built-in GeoServer:

```
(tethys)$ tethys docker ip
...
GeoServer:
  Host: 127.0.0.1
  Port: 8181
  Endpoint: http://127.0.0.1:8181/geoserver/rest
...
```

e. Specify either the username or password of your GeoServer as well. The default GeoServer username and password are *"admin"* and *"geoserver"*, respectively. When you are done you will have something similar to this:

f. Press "Save" to save the Spatial Dataset Service configuration.



4. Link to Dataset Services

Optionally, you may wish to link to external Dataset Services such as CKAN and HydroShare. Dataset Services can be used by apps as data stores and data sources. See *Dataset Services API* documentation for how to use Dataset Services in apps. Complete the following steps for each dataset service you wish to link to:

a. Select "Dataset Services" from the options listed on the Tethys Portal Admin Console.

b. Click on the "Add Dataset Service" button to create a new link to the dataset service.

c. Provide a unique name for the dataset service.

d. Select the appropriate engine and provide an endpoint to the Dataset Service. The endpoint is a URL pointing to the dataset service API. For example, the endpoint for a CKAN dataset service would be of the form

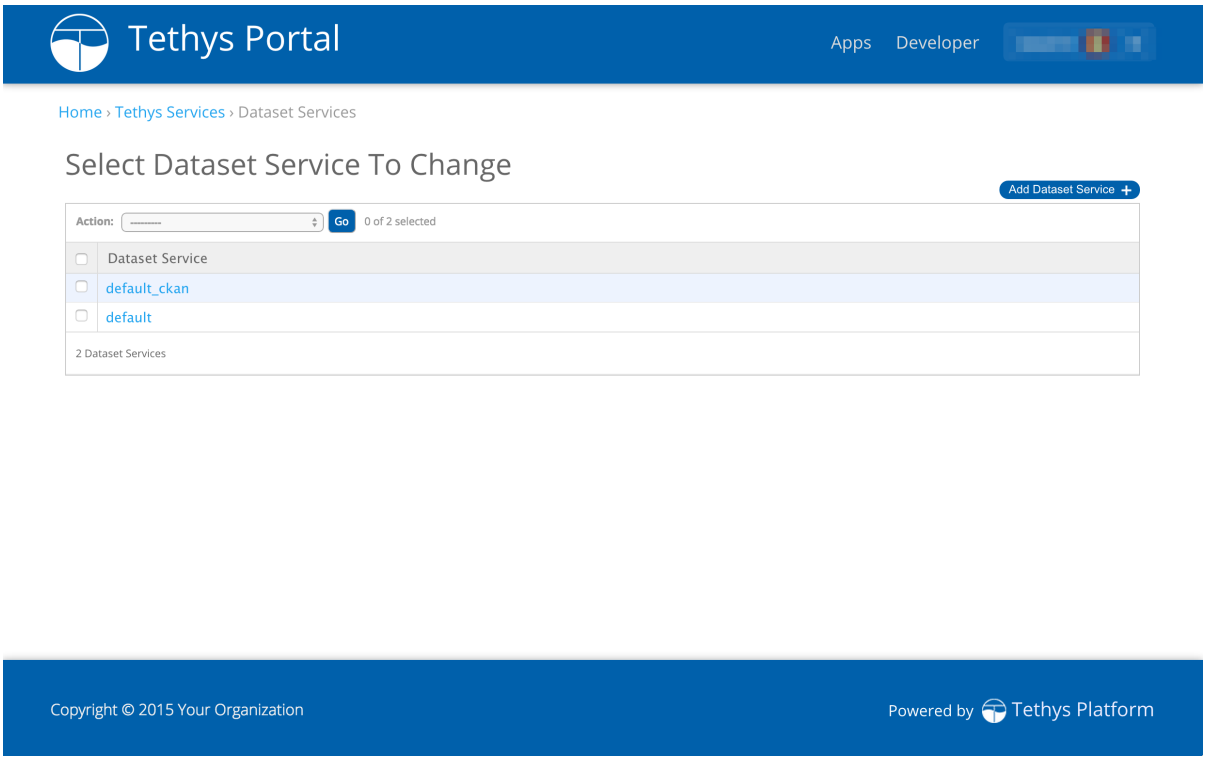
```
http://<host>:<port>/api/3/action
```

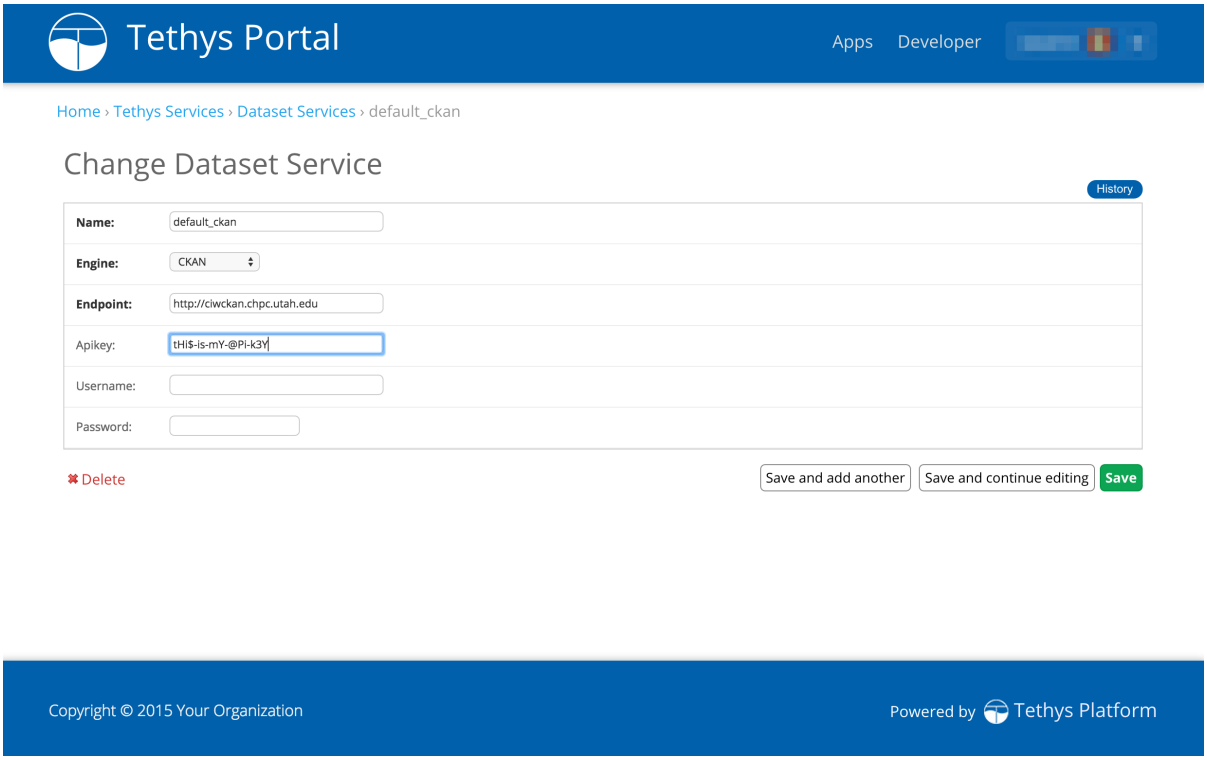
If authentication is required, specify either the API Key or username or password as well. When you are done you will have something similar to this:

Tip: When linking Tethys to a CKAN dataset service, an API Key is required. All user accounts are issued an API key. To access the API Key log into the CKAN site where you have an account and browse to your user profiles. The API key will be listed as a private attribute of your user profile.

---

e. Press "Save" to save the Dataset Service configuration.


Tethys Portal
Apps
Developer
Home › Tethys Services › Dataset Services
Select Dataset Service To Change
Add Dataset Service
Action:
Go
0 of 2 selected
Dataset Service
default_ckan
default
2 Dataset Services
Copyright © 2015 Your Organization
Powered by Tethys Platform

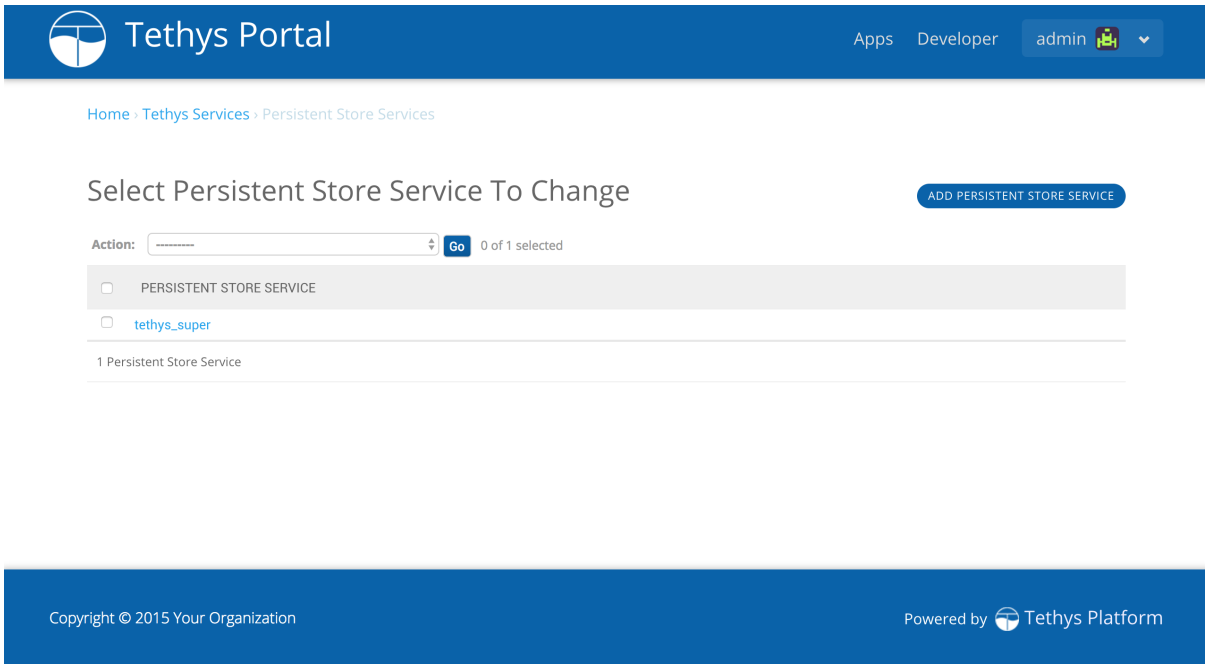

Tethys Portal
Apps
Developer
Home › Tethys Services › Dataset Services › default_ckan
Change Dataset Service
History
Name:
default_ckan
Engine:
CKAN
Endpoint:
http://ciwckan.chpc.utah.edu
Apikey:
tHi$-is-mY-@Pi-k3Y
Username:
Password:
Delete
Save and add another
Save and continue editing
Save
Copyright © 2015 Your Organization
Powered by Tethys Platform

5. Link to Persistent Store Services

Optionally, you may wish to link to external Persistent Store Services such as PostgreSQL. Persistent Store Services can be used by apps as data stores and data sources. See *Persistent Stores API* documentation for how to use Persistent Store Services in apps. Complete the following steps for each service you wish to link to:

a. Select "Persistent Store Services" from the options listed on the Tethys Portal Admin Console.

b. Click on the "Add Persistent Store Service" button to create a new link to the persistent store service.



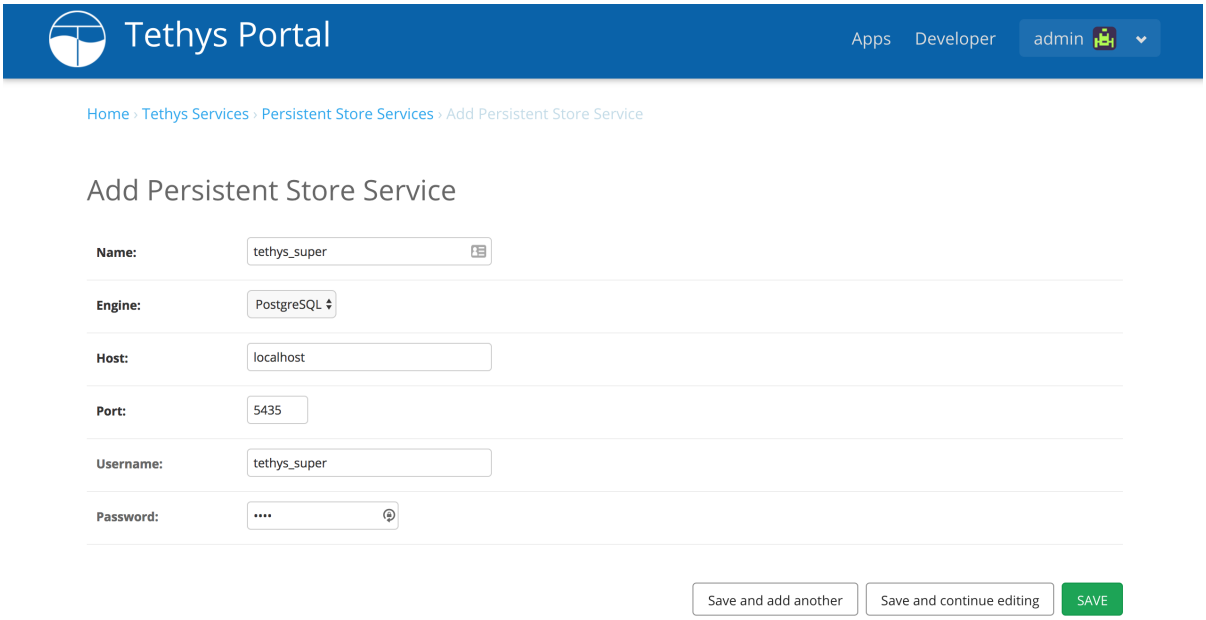
c. Provide a unique name for the persistent store service.

d. Select the appropriate engine and provide an endpoint, enter the host and port, and the username and password of the database user that will be used to authenticate with the service. When you are done you will have something similar to this:

e. Press "Save" to save the Persistent Store Service configuration.

What's Next?

Head over to *Getting Started* and create your first app. You can also check out the *Software Development Kit* documentation to familiarize yourself with all the features that are available.



1.3.6 Upgrade to 2.1.3

Last Updated: December 2018

This document provides a recommendation for how to upgrade Tethys Platform from the last release version. If you have not updated Tethys Platform to the last release version previously, please revisit the documentation for that version and follow those upgrade instructions first.

1. Activate Tethys environment and start your Tethys Database:

```
. t
tstartdb
```

2. Backup your `settings.py` (Note: If you do not backup your `settings.py` you will be prompted to overwrite your settings file during upgrade):

```
mv $TETHYS_HOME/src/tethys_portal/settings.py $TETHYS_HOME/src/tethys_portal/settings_
↪20.py
```

Caution: Don't forget to copy any settings from the backup settings script (`settings.py_bak`) to the new settings script. Common settings that need to be copied include:

- DEBUG
- ALLOWED_HOSTS
- DATABASES, TETHYS_DATABASES
- STATIC_ROOT, TETHYS_WORKSPACES_ROOT
- EMAIL_HOST, EMAIL_PORT, EMAIL_HOST_USER, EMAIL_HOST_PASSWORD, EMAIL_USE_TLS, DEFAULT_FROM_EMAIL
- SOCIAL_OAUTH_XXXX_KEY, SOCIAL_OAUTH_XXXX_SECRET
- BYPASS_TETHYS_HOME_PAGE

3. (Optional) If you want the new environment to be called `tethys` remove the old environment:

```
conda activate base
conda env remove -n tethys
```

Tip: If these commands don't work, you may need to update your conda installation:

```
conda update conda -n root -c defaults
```

4. Download and execute the new install tethys script with the following options (Note: if you removed your previous tethys environment, then you can omit the `-n tethys21` option to have the new environment called `tethys`):

```
wget https://raw.githubusercontent.com/tethysplatform/tethys/release/scripts/
install_tethys.sh
bash install_tethys.sh -b release --partial-tethys-install cieast -n tethys21
```

5. (Optional) If you have a locally installed database server then you need to downgrade postgresql to the version that the database was created with. If it was created by the 2.0 Tethys install script then it was created with postgresql version 9.5. (Note: be sure to open a new terminal so that the newly created tethys environment is activated):

```
t
conda install -c conda-forge postgresql=9.5
```

Tip: These instructions assume your previous installation was done using the install script with the default configuration. If you used any custom options when installing the environment initially, you will need to specify those same options. For an explanation of the installation script options, see: *Install Script Options*.

1.3.7 Production Installation

Last Updated: August 12, 2015

The following instructions can be used to install Tethys Platform on a production server.

System Requirements

Last Updated: April 18, 2015

Tethys Platform is composed of several software components, each of which has the potential of using a copious amount of computing resources (see Figure 1). We recommend distributing the software components across several servers to optimize the use of computing resources and improve performance of Tethys Platform. Specifically, we recommend having a separate server for each of the following components:

- Tethys Portal
- PostgreSQL with PostGIS
- GeoServer
- 52 North WPS
- HTCondor

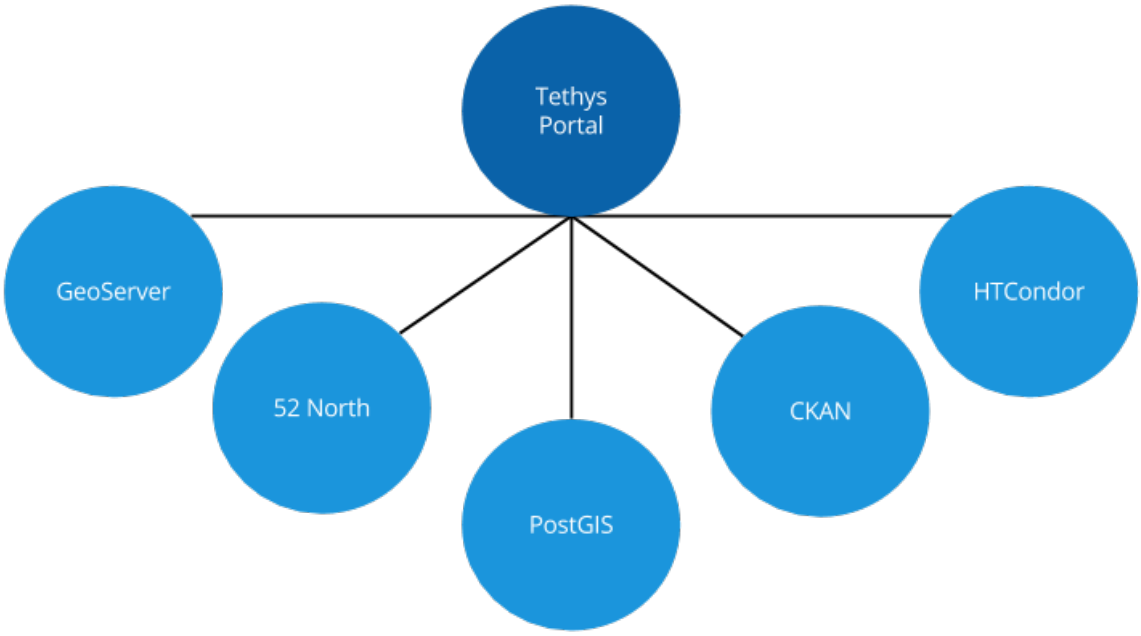


Fig. 1: **Figure 1.** Tethys Platform consists of several software components that should be hosted on separate servers in a production environment.

- CKAN

The following requirements should be interpreted as minimum guidelines. It is likely you will need to expand storage, RAM, or processors as you add more apps. Each instance of Tethys Platform will need to be fine tuned depending to fit the requirements of the apps that it is serving.

Tethys Portal

Tethys Portal is a Django web application. It needs to be able to handle requests from many users meaning it will need processors and memory. Apps should be designed to offload data storage onto one of the data storage nodes (CKAN, database, GeoServer) to prevent the Tethys Portal server from get bogged down with file reads and writes.

- Processor: 2 CPU Cores @ 2 GHz each
- RAM: 4 GB
- Hard Disk: 20 GB

GeoServer

GeoServer is used to render maps and spatial data. It performs operations like coordinate transformations and format conversions on the fly, so it needs a decent amount of processing power and RAM. It also requires storage for the datasets that it is serving.

- Processor: 4 CPU Cores @ 2 GHz each
- RAM: 8 GB
- Hard Disk: 500 GB +

52 North WPS

52 North WPS is a geoprocessing service provider and as such will require processing power.

- Processors: 4 CPU Cores @ 2 GHz each
- RAM: 8 GB
- Hard Disk: 100 GB

PostgreSQL with PostGIS

PostgreSQL is a database server and it should be optimized for storage. The PostGIS extension also provide the server with geoprocessing capabilities, which may require more processing power than recommended here.

- Processors: 4 CPU Cores @ 2 GHz each
- RAM: 4 GB
- Hard Disk: 500 GB +

Production Installation

Last Updated: June, 2017

This article will provide an overview of how to install Tethys Portal in a production setup ready to host apps. Currently production installation of Tethys is only supported on Linux. Some parts of these instructions are optimized for Ubuntu 16.04, though installation on other Linux distributions will be similar.

1. Install Tethys Portal

Follow the default *Installation on Linux and Mac OSX* instructions to install Tethys Portal with the following considerations

- Make sure to checkout the correct branch. The master branch provides the latest stable release.
- Assign strong passwords to the database users.
- A new settings file with production specific settings will be generated and will overwrite an existing settings file. If you want to preserve existing settings make sure to rename or move the existing `settings.py` file.
- Optionally, Follow the *Distributed Configuration* instructions to install Docker and the components of the software suite on separate servers.

For a production installation the installation script should be run with all of the following settings:

```
$ bash install_tethys.sh -allowed-host <YOUR_SERVERS_HOSTNAME> --db-username <SECURE_
↪DB_USERNAME> --db-password <SECURE_DB_PASSWORD> --db-port <PORT_FOR_YOUR_DB_SERVER>␣
↪--superuser <PORTAL_ADMIN_USERNAME> --superuser-email <PORTAL_ADMIN_EMAIL> --
↪superuser-pass <PORTAL_ADMIN_PASSWORD> --production
```

Note: The parameters indicated with angle brackets <> should be replaced with appropriate values for your production server.

2. Customize Production Settings

A new `settings.py` file will have been generated during the production installation specifically for a production environment. Notwithstanding, the internet is a hostile environment and you need to take every precaution to make sure your Tethys Platform installation is secure. Django provides a Deployment Checklist that points out critical settings. You should review this checklist carefully before launching your site. Follow the process described below to review and edit settings. Only a few examples are included here, but be sure to review and update any settings that are needed to provide a secure production server environment.

Open the `settings.py` module for editing using `vim` or another text editor:

```
sudo vim $TETHYS_HOME/src/tethys_apps/settings.py
```

Press `i` to start editing and change settings as necessary for your production environment. Some settings you may want to customize include:

a. Social authentication settings

   If you wish to enable social authentication capabilities in your Tethys Portal, follow the *Social Authentication* instructions.

b. Email settings

   If you would like to enable resetting passwords then an email server needs to be configured. See the next section for details.

Press `ESC` to exit `INSERT` mode and then press `:x` and `ENTER` to save changes and exit.

Important: Review the Deployment Checklist carefully.

---

3. Setup Email Capabilities (optional)

Tethys Platform provides a mechanism for resetting forgotten passwords that requires email capabilities, for which we recommend using Postfix. Install Postfix as follows:

```
sudo apt-get install postfix
```

When prompted select "Internet Site". You will then be prompted to enter you Fully Qualified Domain Name (FQDN) for your server. This is the domain name of the server you are installing Tethys Platform on. For example:

```
foo.example.org
```

Next, configure Postfix by opening its configuration file:

```
sudo vim /etc/postfix/main.cf
```

Press `i` to start editing, find the *myhostname* parameter, and change it to point at your FQDN:

```
myhostname = foo.example.org
```

Find the *mynetworks* parameter and verify that it is set as follows:

```
mynetworks = 127.0.0.0/8 [::ffff:127.0.0.0]/104 [::1]/128
```

Press `ESC` to exit `INSERT` mode and then press `:x` and `ENTER` to save changes and exit. Finally, restart the Postfix service to apply the changes:

```
sudo service postfix restart
```

Several email settings need to be configured for the forget password functionality to work properly. The following exampled illustrates how to setup email in the `settings.py` file.

```
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'localhost'
EMAIL_PORT = 25
EMAIL_HOST_USER = ''
EMAIL_HOST_PASSWORD = ''
EMAIL_USE_TLS = False
DEFAULT_FROM_EMAIL = 'Example <noreply@exmaple.com>'
```

For more information about setting up email capabilities for Tethys Platform, refer to the Sending email documentation.

For an excellent guide on setting up Postfix on Ubuntu, refer to How To Install and Setup Postfix on Ubuntu 14.04.

4. Setup SSL (https) on the Tethys and Geoserver (Recommended)

SSL is the standard technology for establishing a secured connection between a web server and a browser. In order to create a secured connection, an SSL certificate and key are needed. An SSL certificate is simply a paragraph with letters and numbers that acts similar to a password. When users visit your website via https this certificate is verified and if it matches, then a connecton is established. An SSL certificate can be self-signed, or purchased from a Certificate Authority. Some of the top certificate authorities include: Digicert, VertiSign, GeoTrust, Comodo, Thawte, GoDaddy, and Nework Solutions. If your instance of Tethys is part of a larger organization, contact your IT to determine if an agreement with one of these authorities already exists.

Once a certificate is obtained, it needs to be referenced in the Nginx configuration, which is the web server that Tethys uses in production. The configuration file can be found at:

```
/home/<username>/tethys/src/tethys_portal/tethys_nginx.conf
```

The file should look something like this:

```
# tethys_nginx.conf

# the upstream component nginx needs to connect to
upstream django {
    server unix://run/uwsgi/tethys.sock; # for a file socket
}
# configuration of the server
server {
    # the port your site will be served on
    listen      80;
    # the domain name it will serve for
    server_name <domain-name>; # substitute your machine's IP address or FQDN
    charset     utf-8;

    # max upload size
    client_max_body_size 75M;   # adjust to taste

    # Tethys Workspaces
    location /workspaces  {
        internal;
        alias /home/<username>/tethys/workspaces;  # your Tethys workspaces files -␣
↪amend as required
```

(continues on next page)

(continued from previous page)

```
    }

    location /static {
        alias /home/<username>/tethys/static; # your Tethys static files - amend as␣
↪required
    }

    # Finally, send all non-media requests to the Django server.
    location / {
        uwsgi_pass  django;
        include /etc/nginx/uwsgi_params;
    }
}
```

If you need your site to be accessible through both secured (https) and non-secured (http) connections, you will need a server block for each type of connection. Otherwise just edit the existing block.

Make a copy of the existing non-secured server block and paste it below the original. Then modify it as shown below:

```
server {

listen   443;

ssl    on;
ssl_certificate    /home/<username>/tethys/ssl/your_domain_name.pem; (or bundle.crt)
ssl_certificate_key    /home/<username>/tethys/ssl/your_domain_name.key;


# the domain name it will serve for
server_name <domain-name>; # substitute your machine's IP address or FQDN
charset     utf-8;

# max upload size
client_max_body_size 75M;   # adjust to taste

# Tethys Workspaces
location /workspaces  {
    internal;
    alias /home/<username>/tethys/workspaces;  # your Tethys workspaces files - amend␣
↪as required
}

location /static {
    alias /home/<username>/tethys/static; # your Tethys static files - amend as␣
↪required
}

# Finally, send all non-media requests to the Django server.
location / {
    uwsgi_pass  django;
    include /etc/nginx/uwsgi_params;
}
```

Note: SSL works on port 443, hence the server block above listens on 443 instead of 80

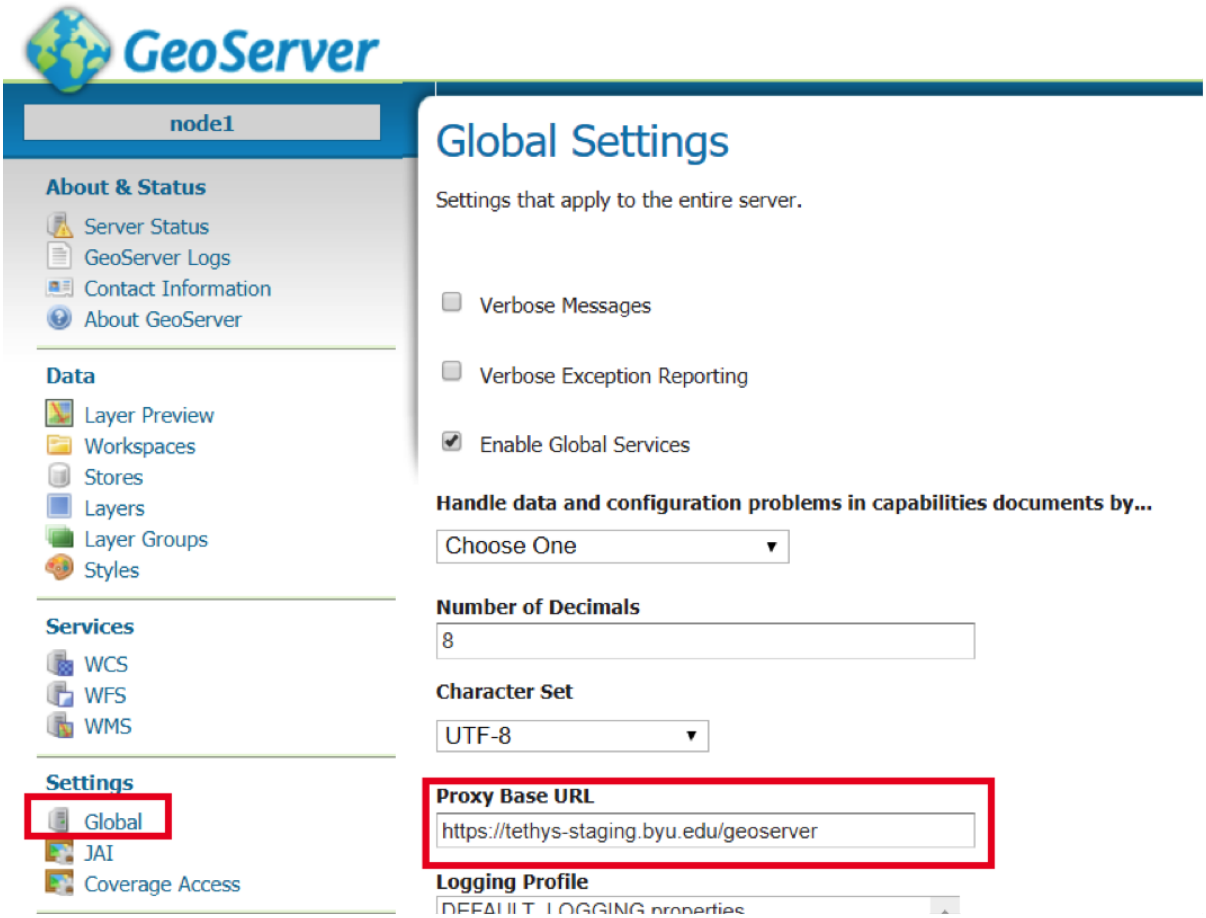
Geoserver SSL

A secured server can only communicate with other secured servers. Therefore to allow the secured Tethys Portal to communicate with Geoserver, the latter needs to be secured as well. To do this, add the following location at the end of your server block.

```
server {

listen   443;

ssl    on;
ssl_certificate    /home/<username>/tethys/ssl/your_domain_name.pem; (or bundle.crt)
ssl_certificate_key    /home/<username>/tethys/ssl/your_domain_name.key;


# the domain name it will serve for
server_name <domain-name>; # substitute your machine's IP address or FQDN
charset     utf-8;

# max upload size
client_max_body_size 75M;   # adjust to taste

# Tethys Workspaces
location /workspaces  {
    internal;
    alias /home/<username>/tethys/workspaces;  # your Tethys workspaces files - amend␣
↪as required
}

location /static {
    alias /home/<username>/tethys/static; # your Tethys static files - amend as␣
↪required
}

# Finally, send all non-media requests to the Django server.
location / {
    uwsgi_pass  django;
    include /etc/nginx/uwsgi_params;
}

#Geoserver
location /geoserver {
      proxy_pass http://127.0.0.1:8181/geoserver;
}
```

Next, go to your Geoserver web interface (http://domain-name:8181/geoserver/web), sign in, and set the **Proxy Base URL** in Global settings to:

```
https://<domain-name>/geoserver
```



Finally, restart uWSGI and Nginx services to effect the changes:

```
$ sudo systemctl restart tethys.uwsgi.service
$ sudo systemctl restart nginx
```

Tip: Use the alias *trestart* as a shortcut to doing the final step.

The portal should now be accessible from: https://domain-name

Geoserver should now be accessible from: https://domain-name/geoserver

Note: Notice that the Geoserver port (8181) is not necessary once the proxy is configured

5. Install Apps

Download and install any apps that you want to host using this installation of Tethys Platform. For more information see: *Installing Apps in Production*.

Todo: **Troubleshooting**: Here we try to provide some guidance on some of the most commonly encountered issues. If you are experiencing problems and can't find a solution here then please post a question on the Tethys Platform Forum.

Installing Apps in Production

Last Updated: June, 2017

Installing apps in a Tethys Platform configured for production can be challenging. Most of the difficulties arise, because Tethys is served by Nginx in production and all the files need to be owned by the Nginx user. The following instructions for installing apps in a production environment are provided to aid administrators of a Tethys Portal.

1. Change the Ownership of Files to the Current User

During the production installation any Tethys related files were change to be owned by the Nginx user. To make any changes on the server it is easiest to change the ownership back to the current user. This is easily done with an alias that was created in the tethys environment during the production installation process:

```
$ t
(tethys) $ tethys_user_own
```

3. Download App Source Code

You will need to copy the source code of the app to the server. There are many methods for accomplishing this, but one way is to create a repository for your code in GitHub. To download the source from GitHub, clone it as follows:

```
$ cd $TETHYS_HOME/apps/
$ sudo git clone https://github.com/username/tethysapp-my_first_app.git
```

Tip: Substitute "username" for your GitHub username or organization and substitute "tethysapp-my_first_app" for the name of the repository with your app source code.

---

4. Install the App

Execute the setup script (`setup.py`) with the `install` command to make Python aware of the app and install any of its dependencies:

```
(tethys) $ cd cd $TETHYS_HOME/apps/tethysapp-my_first_app
(tethys) $ python setup.py install
```

5. Collect and Static Files and Workspaces

The static files and files in app workspaces are hosted by Nginx, which necessitates collecting all of the static files to a single directory and all workspaces to another single directory. These directory is configured through the `STATIC_ROOT` and `TETHYS_WORKSPACES_ROOT` setting in the `settings.py` file. Collect the static files and workspaces with this command:

```
(tethys) $ tethys manage collectall
```

6. Change the Ownership of Files to the Nginx User

The Nginx user must own any files that Nginx is serving. This includes the source files, static files, and any workspaces that your app may have. The following alias will accomplish the change in ownership that is required:

```
(tethys) $ tethys_server_own
```

7. Restart uWSGI and Nginx

Restart uWSGI and Nginx services to effect the changes:

```
$ sudo systemctl restart tethys.uwsgi.service
$ sudo systemctl restart nginx
```

Tip: Use the alias *trestart* as a shortcut to doing both steps 6 and 7.

8. Configure App Settings

Set all required settings on the app settings page in the Tethys Portal admin pages (see *Administrator Pages*).

9. Initialize Persistent Stores

If your app requires a database via the persistent stores API, you will need to initialize it:

```
$ t
(tethys) $ tethys syncstores all
```

Distributed Configuration

Last Updated: April 18, 2015

The Tethys Docker images can be used to easily install each of the software components of Tethys Platform on separate servers. However, you will not be able to use the Tethys commandline tools to install the Dockers as you do during development. The following article describes how to deploy each software component using the native Docker API.

Install Docker on Each Server

After you have provisioned servers for each of the Tethys software components, install Docker on each using the appropriate Docker installation instructions. Docker provides installation instructions for most major types of servers.

GeoServer Docker Deployment

Pull the Docker image for GeoServer using the following command:

```
$ sudo docker pull ciwater/geoserver
```

After the image has been pulled, run a new Docker container as follows:

```
$ sudo docker run -d -p 80:8080 --restart=always --name geoserver ciwater/geoserver
```

Refer to the Docker Run Reference for an explanation of each parameter. To summarize, this will start the container as a background process on port 80, with the restart policy set to always restart the container after a system reboot, and with an appropriate name.

More information about the GeoServer Docker can be found on the Docker Registry:

https://registry.hub.docker.com/u/ciwater/geoserver/

Important: The admin username and password can only be changed using the web admin interface. Be sure to log into GeoServer and change the admin password using the web interface. The default username and password are *admin* and *geoserver*, respectively.

Backup

PostgreSQL with PostGIS Docker Deployment

Pull the Docker image for PostgreSQL with PostGIS using the following command:

```
$ sudo docker pull ciwater/postgis
```

The PostgreSQL with PostGIS Docker automatically initializes with the three database users that are needed for Tethys Platform:

- tethys_default
- tethys_db_manager
- tethys_super

The default password for each is "pass". For production, you will obviously want to change these passwords. Do so using the appropriate environmental variable:

- -e TETHYS_DEFAULT_PASS=<TETHYS_DEFAULT_PASS>
- -e TETHYS_DB_MANAGER_PASS=<TETHYS_DB_MANAGER_PASS>
- -e TETHYS_SUPER_PASS=<TETHYS_SUPER_PASS>

Here is an example of how to use the environmental variables to set passwords when starting a container:

```
$ sudo docker run -d -p 80:5432 -e TETHYS_DEFAULT_PASS="pass" -e TETHYS_DB_MANAGER_
↪PASS="pass" -e TETHYS_SUPER_PASS="pass" --restart=always --name postgis ciwater/
↪postgis
```

Refer to the Docker Run Reference for an explanation of each parameter. To summarize, this will start the container as a background process on port 80, with the restart policy set to always restart the container after a system reboot, and with an appropriate name. It also set the passwords for each database at startup.

More information about the PostgreSQL with PostGIS Docker can be found on the Docker Registry:

https://registry.hub.docker.com/u/ciwater/postgis/

Important: Set strong passwords for each database user for a production system.

Backup

52 North WPS Docker Deployment

Pull the Docker image for 52 North WPS using the following command:

```
$ sudo docker pull ciwater/n52wps
```

After the image has been pulled, run a new Docker container as follows:

```
$ sudo docker run -d -p 80:8080 -e USERNAME="foo" -e PASSWORD="bar" --restart=always -
↪-name n52wps ciwater/n52wps
```

Refer to the Docker Run Reference for an explanation of each parameter. To summarize, this will start the container as a background process on port 80, with the restart policy set to always restart the container after a system reboot, and with an appropriate name. It also sets the username and password for the admin user.

You may pass several environmental variables to set the service metadata and the admin username and password:

- -e USERNAME=<ADMIN_USERNAME>
- -e PASSWORD=<ADMIN_PASSWORD>
- -e NAME=<INDIVIDUAL_NAME>
- -e POSITION=<POSITION_NAME>
- -e PHONE=<VOICE>
- -e FAX=<FACSIMILE>
- -e ADDRESS=<DELIVERY_POINT>
- -e CITY=<CITY>
- -e STATE=<ADMINISTRATIVE_AREA>
- -e POSTAL_CODE=<POSTAL_CODE>
- -e COUNTRY=<COUNTRY>
- -e EMAIL=<ELECTRONIC_MAIL_ADDRESS>

Here is an example of how to use the environmental variables to set metadata when starting a container:

```
$ sudo docker run -d -p 80:8080 -e USERNAME="foo" -e PASSWORD="bar" -e NAME="Roger" -
↪e COUNTRY="USA"  --restart=always --name n52wps ciwater/n52wps
```

More information about the 52 North WPS Docker can be found on the Docker Registry:

https://registry.hub.docker.com/u/ciwater/n52wps/

Important: Set strong passwords for the admin user for a production system.

Maintaining Docker Containers

This section briefly describes some of the common maintenance tasks for Docker containers. Refer to the Docker Documentation for a full description of Docker.

Status

You can view the status of containers using the following commands:

```
# Running containers
$ sudo docker ps

# All containers
$ sudo docker ps -a
```

Start and Stop

Docker containers can be stopped and started using the names assigned to them. For example, to stop and start a Docker named "postgis":

```
$ sudo docker stop postgis
$ sudo docker start postgis
```

Attach to Container

You can attach to running containers to give you a command prompt to the container. This is useful for checking logs or modifying configuration of the Docker manually. For example, to attach to a container named "postgis":

```
$ sudo docker exec --rm -it postgis bash
```

1.4 Tutorials

Last Updated: May 2017

Use the following tutorials to learn the basics about Tethys Platform.

1.4.1 Getting Started

Last Updated: June 2017

This tutorial will walk you through the steps of setting up a new Tethys App project using Tethys Platform. If you have not already installed Tethys Platform, follow the *Installation* documentation and then return.

You will need to use the command line/terminal to manage your app and run the development server. It is highly recommended that you read the *Terminal Quick Guide* article for some tips if you are new to command line before you get started.

New Tethys App Project

Last Updated: June 2017

Tethys Platform provides an easy way to create new app projects called a scaffold. The scaffold generates a Tethys app project with the minimum files and the folder structure that is required (see *App Project Structure*).

Tip: You will need to use the command line/terminal to manage your app and run the development server. See the *Terminal Quick Guide* article for some tips if you are new to command line.

---

1. Generate Scaffold

To generate a new app using the scaffold, open a terminal, press `CTRL-C` to stop the development server if it is still running, and execute the following commands:

Linux and Mac:

```
         $ t
(tethys) $ mkdir ~/tethysdev
(tethys) $ cd ~/tethysdev
(tethys) $ tethys scaffold dam_inventory
```

Windows:

Locate the `tethys_cmd.bat` (in the TETHYS_HOME directory) and double-click it to open a new command windows with the Tethys environment activated. Then run the following commands:

```
(tethys) $ mkdir C:%HOMEPATH%\tethysdev
(tethys) $ cd %HOMEPATH%\tethysdev
(tethys) $ tethys scaffold dam_inventory
```

Tip: Windows Users: If you have admin rights on your computer, it is even better to right-click on the `tethys_cmd.bat` and select **Run as Administrator**.

You will be prompted to enter metadata about your app such as, proper name, version, author, and description. All of these metadata are optional. You can accept the default value by pressing enter, repeatedly.

In a file browser change into your `Home` directory and open the `tethysdev` directory. If the scaffolding worked, you should see a directory called `tethysapp-dam_inventory`. All of the source code for your app is located in this directory. For more information about the app project structure, see *App Project Structure*.

2. Development Installation

Now that you have a new Tethys app project, you need to install the app on your development Tethys Portal. In a terminal, change into the `tethysapp-dam_inventory` directory and execute the **`python setup.py develop`** command. Be sure to activate the Tethys *Python conda environment* if it is not already activated (see line 1 of the first code block):

Linux and Mac:

```
(tethys) $ cd ~/tethysdev/tethysapp-dam_inventory
(tethys) $ python setup.py develop
```

Windows:

```
(tethys) $ cd C:%HOMEPATH%\tethysdev\tethysapp-dam_inventory
(tethys) $ python setup.py develop
```

Tip: Windows Users: If you get an error when running `python setup.py develop`, then you have insufficient permissions to install your app in development mode. Either try opening the `tethys_cmd.bat` as an administrator and run the commands again, or run `python setup.py install`. The disadvantage to this method is that each time you want Tethys to reflect changes to your app code, you will need to run `python setup.py install` again.

---

3. View Your New App

Use start up the development server:

```
(tethys) $ tethys manage start
```

OR use the *tms* alias:

```
(tethys) $ tms
```

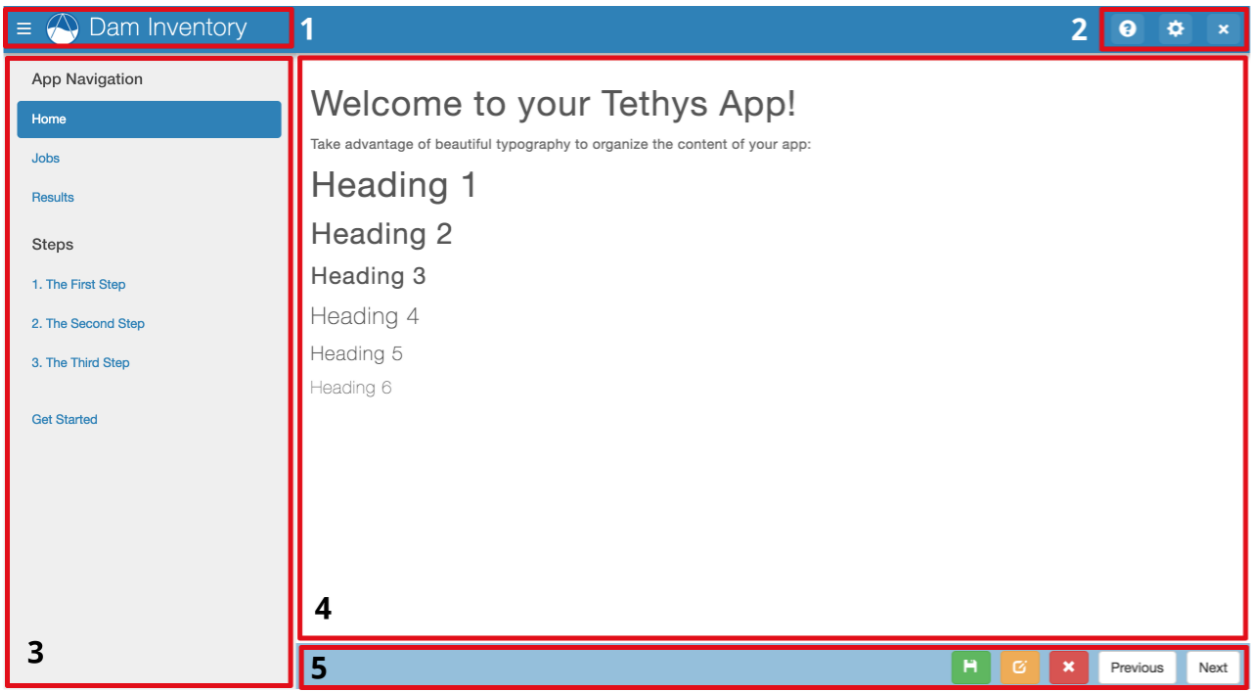
Note: If you get errors related to Tethys not being able to connect to the database, start the database by running:

```
(tethys) $ tstartdb
```

You can also stop the Tethys database by running:

```
(tethys) $tstopdb
```

Browse to http://127.0.0.1:8000/apps in a web browser. If all has gone well, you should see your app listed on the app library page. Exploring your new app won't take long, because there is only one page. Familiarize yourself with different parts of the app interface (see below).



Parts of a Tethys app interface: (1) app navigation toggle and app branding; (2) exit button, settings, button, and custom buttons; (3) app navigation, (4) app content, and (5) app actions.

Tip: To stop the development server press `CTRL-C`.

---

4. App Project Paths

Throughout the tutorial, you will be asked to open various files. Most of the files will be located in your *app package* directory which shares the name of your app: "dam_inventory". If you generated your scaffold exactly as above, this directory will be located at the following path:

```
# Linux and Mac
~/tethysdev/tethysapp-dam_inventory/tethysapp/dam_inventory/

# Windows
C:%HOMEPATH%\tethysdev\tethysapp-dam_inventory\tethys_app\dam_inventory\
```

For convenience, all paths in the following tutorials will be given relative to the *app package* directory. For example:

```
# This path:
~/tethysdev/tethysapp-dam_inventory/tethysapp/dam_inventory/controllers.py

# Will be referred to as:
controllers.py
```

Tip: As you explore the contents of your app project, you will notice that many of the directories have files named `__init__.py`. Though many of these files are empty, they are important and should not be deleted. They inform Python that the containing directory is a Python package. Python packages and their contents can be imported in Python scripts. Removing the `__init__.py` files will result in breaking import statements and it could make some

of your code inaccessible. Similarly, if you add a directory to your project that contains Python modules and you would like them to be made available to your code, add a `__init__.py` file to the directory to make it a package.

Beginner Concepts

Last Updated: June 2017

This tutorial introduces important concepts for first-time or beginner Tethys developers. The topics covered include:

- The App Class
- Model View Controller
- Introduction to the View
- Django Template Language
- Introduction to the Controller
- Templating Gizmos
- Custom Styles and CSS
- Linking Between Pages
- App Navigation

1. App Class

The app class, located in `app.py` is the primary configuration file for Tethys apps. All app classes inherit from the `TethysAppBase` class.

a. Open `app.py` in your favorite Python IDE or text editor.

b. Change the theme color of your app by changing the value of the `color` property of the `DamInventory` class. Use a site like color-hex to find an appropriate hexadecimal RGB color.

c. You can also change the icon of your app. Find a new image online (square images work best) and save it in the `public/images/` directory of your app. Then change the value of the `icon` property of the `DamInventory` class to match the name of the image.

Tip: For more details about the app class, see the *App Base Class API*.

Warning: DO NOT change the value of the `index`, `package`, or `root_url` properties of your app class unless you know what you are doing. Doing so will break your app.

2. App Settings

Other settings for your app can be configured in the app settings. App settings are stored in a database, meaning they can be changed dynamically by the administrator of your portal. Some app settings, like Name and Description, correspond with properties in the `app.py` script.

a. To access the app settings, click on the Settings button (gear icon) at the top right-hand corner of your app.

b. Change the Name and Description of your app by changing their respective values on the app settings page. Press the `Save` button, located at the bottom of the app settings page. Then navigate back to your app to see the changes.

You can also create custom settings for your app that can be configured on the app settings page:

b. Open the `app.py` and add the `custom_settings()` method to the `DamInventory` class. Don't for get to import `CustomSetting`:

```
from tethys_sdk.app_settings import CustomSetting

...

class DamInventory(TethysAppBase):
    """
    Tethys app class for Dam Inventory.
    """
    ...

    def custom_settings(self):
        """
        Example custom_settings method.
        """
        custom_settings = (
            CustomSetting(
                name='max_dams',
                type=CustomSetting.TYPE_INTEGER,
                description='Maximum number of dams that can be created␣
↪in the app.',
                required=False
            ),
        )
        return custom_settings
```

Warning: Ellipsis in code blocks in Tethys tutorials indicate code that is not shown for brevity. When there are ellipsis in the code, DO NOT COPY AND PASTE THE BLOCK VERBATIM.

c. Save changes to `app.py` then restart your development server (press `CTRL-C` to stop followed by the `tms` command to start it again).

d. Navigate to the settings page of your app and scroll down to the **Custom Settings** section and you should see an entry for the `max_dams` settings. Enter a value and save changes to the setting. You will learn how to use this custom setting in the app later on in the tutorial.

Tip: For more information about app settings, see the *App Settings API*.

3. Model View Controller

Tethys apps are developed using the *Model View Controller* (MVC) software architecture pattern. Model refers to the data model and associated code, View refers to the representations of the data, and Controller refers of the code that coordinates data from the Model for rendering in the View. In Tethys apps, the Model is usually an SQL database or files and the code for accessing them, the Views are most often the templates or HTML files, and Controllers are implemented as Python functions.

Tip: For more information about the MVC pattern, see *Key Concepts*.

---

4. Views

Views for Tethys apps are constructed using the standard web programming tools: HTML, JavaScript, and CSS. Additionally, Tethys Platform provides the Django Python templating language allowing you to insert Python code into your HTML documents. The result is dynamic, reusable templates for the web pages of your app.

a. Open `/templates/dam_inventory/home.html` and replace it's contents with the following:

```
{% extends "dam_inventory/base.html" %}
{% load tethys_gizmos %}

{% block app_content %}
  {% gizmo dam_inventory_map %}
{% endblock %}

{% block app_actions %}
  {% gizmo add_dam_button %}
{% endblock %}
```

Tip: Django TemplateLanguage: If you are familiar with HTML, the contents of this file may seem strange. That's because the file is actually a Django template, which contains special syntax (i.e.: `{% ... %}` and `{{ ... }}` to make the template dynamic. Django templates can contain variables, filters, and tags.

Variables. Variables are denoted by double curly brace syntax like this: `{{ variable }}`. Template variables are replaced by the value of the variable. Dot notation can be used access attributes of a variable, keys of dictionaries, and items in lists: `{{ my_object.attribute }}`, `{{ my_dict.key }}`, and `{{ my_list.3 }}`.

Filters. Variables can be modified by filters which look like this: `{{ variable|filter:argument }}`. Filters perform modifying functions on variable output such as formatting dates, formatting numbers, changing the letter case, and concatenating multiple variables.

Tags. Tags use curly-brace-percent-sign syntax like this: `{% tag %}`. Tags perform many different functions including creating text, controlling flow, or loading external information to be used in the app. Some commonly used tags include `for`, `if`, `block`, and `extends`.

Blocks. The block tags in the Tethys templates coorespond with different areas in the app. For example, any HTML written inside the `app_content` block will render in the app content area of the app.

For a better explanation of the Django Template Language and the blocks available in Tethys apps see the *App Templating API*.

---

5. Controllers

Controllers consist of a Python function that takes a `request` object as an argument. The `request` object contains all the information about the incoming request including any data being passed to the server, information about the user that is logged in, and the HTTP headers. Each controller function is associated with one view or template. Any variable assigned to the `context` variable in a controller becomes a variable on the template specified in the `render` function.

a. Open `controllers.py` and define the `dam_inventory_map` and `add_dam_button` gizmos in your home controller. Open `controllers.py` and change the `home` controller function as follows:

```
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
from tethys_sdk.gizmos import MapView, Button


@login_required()
def home(request):
    """
    Controller for the app home page.
    """

    dam_inventory_map = MapView(
        height='100%',
        width='100%',
        layers=[],
        basemap='OpenStreetMap',
    )


    add_dam_button = Button(
        display_text='Add Dam',
        name='add-dam-button',
        icon='glyphicon glyphicon-plus',
        style='success'
    )

    context = {
        'dam_inventory_map': dam_inventory_map,
        'add_dam_button': add_dam_button
    }

    return render(request, 'dam_inventory/home.html', context)
```

b. Save your changes to `controllers.py` and `home.html` and refresh the page to view the map.

Tip: Gizmos: The `home.html` template used a Tethys template tag, `gizmo`, to insert a map and a button with only one line of code: `{% gizmo dam_inventory_map %}`. Gizmo tags require one argument, an object that defines the options for the gizmo. These gizmo options must be defined in the controller for that view. In the example above we define the options objects for the two gizmos on the `home.html` template and pass them to the template through the context dictionary.

For more details on the Map View or Button Gizmos see: *Map View* and *Button and Button Group* For more information about Gizmos in general see the *Template Gizmos API*.

---

6. Custom Styles

It would look nicer if the map gizmo filled the entire app content area. To do this, we will need to add custom CSS or style rules to remove the padding around the `inner-app-content` area.

a. Create a new file `/public/css/map.css` and add the following contents:

```
#inner-app-content {
    padding: 0;
}

#app-content, #inner-app-content, #map_view_outer_container {
    height: 100%;
}
```

b. Load the styles on the `/templates/dam_inventory/home.html` template by adding a link to the `public/css/map.css` to it. To do this add `staticfiles` to the load statement at the top of the template and add the `styles` block to the end of the file:

```
{% load tethys_gizmos staticfiles %}

...

{% block styles %}
    {{ block.super }}
    <link href="{% static 'dam_inventory/css/map.css' %}" rel="stylesheet"/>
{% endblock %}
```

c. Save your changes to `map.css` and `home.html` and refresh the page to view the changes. The map should fill the content area now. Notice how the map dynamically resizes if the screen size changes.

Important: Don't forget the `{{ block.super }}`! The `{{ block.super }}` statement loads all previously loaded styles in this block. If you forget the `{{ block.super }}`, it will result in a broken page with no styles applied.

---

7. Create a New Page

Creating a new page in your app consists of three steps: (1) create a new template, (2) add a new controller to `controllers.py`, and (3) add a new `UrlMap` to the `app.py`.

a. Create a new file `/templates/dam_inventory/add_dam.html` and add the following contents:

```
{% extends "dam_inventory/base.html" %}
```

This is the simplest template you can create in a Tethys app, which amounts to a blank Tethys app page. You must still extend the `base.html` to retain the styling of an app page.

b. Create a new controller function called `add_dam` at the bottom of the `controllers.py`:

```
@login_required()
def add_dam(request):
    """
    Controller for the Add Dam page.
    """
```

(continues on next page)

(continued from previous page)

```
    context = {}
    return render(request, 'dam_inventory/add_dam.html', context)
```

This is the most basic controller function you can write: a function that accepts an argument called `request` and a return value that is the result of the `render` function. The `render` function renders the Django template into valid HTML using the `request` and `context` provided.

c. Create a new URL Map for the `add_dam` controller in the `url_maps` method of App Class in `app.py`:

```
class DamInventory(TethysAppBase):
    """
    Tethys app class for Dam Inventory.
    """
    ...

    def url_maps(self):
        """
        Add controllers
        """
        UrlMap = url_map_maker(self.root_url)

        url_maps = (
            UrlMap(
                name='home',
                url='dam-inventory',
                controller='dam_inventory.controllers.home'
            ),
            UrlMap(
                name='add_dam',
                url='dam-inventory/dams/add',
                controller='dam_inventory.controllers.add_dam'
            ),
        )

        return url_maps
```

A `UrlMap` is an object that maps a URL for your app to controller function that should handle requests to that URL.

d. At this point you should be able to access the new page by entering its URL (http://localhost:8000/apps/dam-inventory/dams/add/) into the address bar of your browser. It is not a very exciting page, because it is blank.

Tip: New Page Pattern: Adding new pages is an exercise of the Model View Controller pattern. Generally, the steps are:

- Modify the model as necessary to support the data for the new page
- Create a new HTML template
- Create a new controller function
- Add a new `UrlMap` in `app.py`

---

8. Link to New Page

Finally, you can also link to the page from another page using a button.

a. Modify the `add_dam_button` on the Home page to link to the newly created page (don't forget the import):

```
from django.shortcuts import reverse

...

@login_required()
def home(request):
    ...

    add_dam_button = Button(
        display_text='Add Dam',
        name='add-dam-button',
        icon='glyphicon glyphicon-plus',
        style='success',
        href=reverse('dam_inventory:add_dam')
    )
```

9. Build Out New Page

a. Modify the `template/dam_inventory/add_dam.html` with a title in the app content area and add `Add` and `Cancel` buttons to the app actions area:

```
{% extends "dam_inventory/base.html" %}
{% load tethys_gizmos %}

{% block app_content %}
  <h1>Add Dam</h1>
{% endblock %}

{% block app_actions %}
  {% gizmo cancel_button %}
  {% gizmo add_button %}
{% endblock %}
```

b. Define the options for the `Add` and `Cancel` button gizmos in the `add_app` controller in `controllers.py`. Also add the variables to the context so they are available to the template:

```
@login_required()
def add_dam(request):
    """
    Controller for the Add Dam page.
    """
    add_button = Button(
        display_text='Add',
        name='add-button',
        icon='glyphicon glyphicon-plus',
        style='success'
    )

    cancel_button = Button(
        display_text='Cancel',
```

(continues on next page)

(continued from previous page)

```
        name='cancel-button',
        href=reverse('dam_inventory:home')
    )

    context = {
        'add_button': add_button,
        'cancel_button': cancel_button,
    }

    return render(request, 'dam_inventory/add_dam.html', context)
```

10. Customize Navigation

Now that there are two pages in the app, we should modify the app navigation to have links to the **Home** and **Add Dam** pages.

a. Open `/templates/dam_inventory/base.html` and replace the `app_navigation_items` block:

```
{% block app_navigation_items %}
  <li class="title">Navigation</li>
  <li class="active"><a href="{% url 'dam_inventory:home' %}">Home</a></li>
  <li class=""><a href="{% url 'dam_inventory:add_dam' %}">Add Dam</a></li>
{% endblock %}
```

Notice that the **Home** link in the app navigation is always highlighed, even if you are on the **Add Dam** page. The highlight is controlled by adding the `active` class to the appropriate navigation link. We can get the navigation to highlight appropriately using the following pattern.

b. Modify `app_navigation_items` block in `/templates/dam_inventory/base.html` to dynamically highlight active link:

```
{% block app_navigation_items %}
  {% url 'dam_inventory:home' as home_url %}
  {% url 'dam_inventory:add_dam' as add_dam_url %}
  <li class="title">Navigation</li>
  <li class="{% if request.path == home_url %}active{% endif %}"><a href="{{ home_url␣
↪}}">Home</a></li>
  <li class="{% if request.path == add_dam_url %}active{% endif %}"><a href="{{ add_
↪dam_url }}">Add Dam</a></li>
{% endblock %}
```

The `url` tag is used in templates to lookup URLs using the name of the UrlMap, namespaced by the app package name (i.e.: `namespace:url_map_name`). We assign the urls to two variables, `home_url` and `add_dam_url`, using the `as` operator in the `url` tag. Then we wrap the `active` class of each navigation link in an `if` tag. If the expression given to an `if` tag evaluates to true, then the content of the `if` tag is rendered, otherwise it is left blank. In this case the result is that the `active` class is only added to link of the page we are visiting.

11. Solution

This concludes the Beginner Tutorial. You can view the solution on GitHub at https://github.com/tethysplatform/tethysapp-dam_inventory or clone it as follows:

```
$ git clone https://github.com/tethysplatform/tethysapp-dam_inventory.git
$ cd tethysapp-dam_inventory
$ git checkout beginner-solution
```

Intermediate Concepts

Last Updated: June 2017

This tutorial introduces intermediate concepts for Tethys developers. The topics covered include:

- HTML Forms and User Input
- Handling Form Submissions in Controllers
- Form Validation Pattern
- Introduction to the Model
- File IO and Workspaces
- Intermediate Template Gizmos
- Review of Model View Controller
- Spatial Inputs in Forms
- Rendering Spatial Data on the Map View Gizmo

0. Start From Beginner Solution (Optional)

If you wish to use the beginner solution of the last tutorial as a starting point:

```
$ git clone https://github.com/tethysplatform/tethysapp-dam_inventory.git
$ cd tethysapp-dam_inventory
$ git checkout beginner-solution
```

1. Forms and User Input

HTML forms are the primary mechanism for obtaining input from users of your app. In the next few sections, you'll learn how to create forms in the template and process the data submitted through the form in the controller. For this example, we'll create a form for adding new dams to the inventory.

a. Add a form to the Add Dam page by modifying the `/templates/dam_inventory/add_dam.html` template as follows:

```
{% extends "dam_inventory/base.html" %}
{% load tethys_gizmos %}

{% block app_content %}
  <h1>Add Dam</h1>
  <form id="add-dam-form" method="post">
```

(continues on next page)

(continued from previous page)

```
    {% csrf_token %}
    {% gizmo name_input %}
    {% gizmo owner_input %}
    {% gizmo river_input %}
    {% gizmo date_built_input %}
  </form>
{% endblock %}

{% block app_actions %}
  {% gizmo cancel_button %}
  {% gizmo add_button %}
{% endblock %}
```

The form is composed of the the HTML `<form>` tag and various input gizmos inside it. We'll use the `add_button` gizmo to submit the form. Also note the use of the `csrf_token` tag in the form. This is a security precaution that is required to be included in all the forms of your app (see the Cross Site Forgery protection article in the Django documentation for more details).

Also note that the `method` attribute of the `<form>` element is set to `post`. This means the form will use the POST HTTP method to submit and transmit the data to the server. For an introduction to HTTP methods, see The Definitive Guide to GET vs POST.

b. Define the options for the form gizmos in the controller and change the `add_button` gizmo to be a submit button for the form in the `add_dam` controller:

```
from tethys_sdk.gizmos import TextInput, DatePicker, SelectInput

...

@login_required()
def add_dam(request):
    """
    Controller for the Add Dam page.
    """
    # Define form gizmos
    name_input = TextInput(
        display_text='Name',
        name='name'
    )

    owner_input = SelectInput(
        display_text='Owner',
        name='owner',
        multiple=False,
        options=[('Reclamation', 'Reclamation'), ('Army Corp', 'Army Corp'), ('Other',
↪ 'Other')],
        initial=['Reclamation']
    )

    river_input = TextInput(
        display_text='River',
        name='river',
        placeholder='e.g.: Mississippi River'
    )

    date_built = DatePicker(
        name='date-built',
```

(continues on next page)

(continued from previous page)

```
        display_text='Date Built',
        autoclose=True,
        format='MM d, yyyy',
        start_view='decade',
        today_button=True,
        initial='February 15, 2017'
    )

    add_button = Button(
        display_text='Add',
        name='add-button',
        icon='glyphicon glyphicon-plus',
        style='success',
        attributes={'form': 'add-dam-form'},
        submit=True
    )

    cancel_button = Button(
        display_text='Cancel',
        name='cancel-button',
        href=reverse('dam_inventory:home')
    )

    context = {
        'name_input': name_input,
        'owner_input': owner_input,
        'river_input': river_input,
        'date_built_input': date_built,
        'add_button': add_button,
        'cancel_button': cancel_button,
    }

    return render(request, 'dam_inventory/add_dam.html', context)
```

2. Handle Form Submission

At this point the form will be functional, but the app is not doing anything with the data when the user submits the form. In this section we'll implement a pattern for handling the form submission and validating the form.

a. Change the `add_dam` controller to handle the form data using the form validation pattern:

```
from django.shortcuts import redirect
from django.contrib import messages

...

@login_required()
def add_dam(request):
    """
    Controller for the Add Dam page.
    """
    # Default Values
    name = ''
    owner = 'Reclamation'
    river = ''
```

(continues on next page)

(continued from previous page)

```
    date_built = ''

    # Errors
    name_error = ''
    owner_error = ''
    river_error = ''
    date_error = ''

    # Handle form submission
    if request.POST and 'add-button' in request.POST:
        # Get values
        has_errors = False
        name = request.POST.get('name', None)
        owner = request.POST.get('owner', None)
        river = request.POST.get('river', None)
        date_built = request.POST.get('date-built', None)

        # Validate
        if not name:
            has_errors = True
            name_error = 'Name is required.'

        if not owner:
            has_errors = True
            owner_error = 'Owner is required.'

        if not river:
            has_errors = True
            river_error = 'River is required.'

        if not date_built:
            has_errors = True
            date_error = 'Date Built is required.'

        if not has_errors:
            # Do stuff here
            return redirect(reverse('dam_inventory:home'))

        messages.error(request, "Please fix errors.")

    # Define form gizmos
    name_input = TextInput(
        display_text='Name',
        name='name',
        initial=name,
        error=name_error
    )

    owner_input = SelectInput(
        display_text='Owner',
        name='owner',
        multiple=False,
        options=[('Reclamation', 'Reclamation'), ('Army Corp', 'Army Corp'), ('Other',
↪ 'Other')],
        initial=owner,
        error=owner_error
    )
```

(continues on next page)

(continued from previous page)

```
    river_input = TextInput(
        display_text='River',
        name='river',
        placeholder='e.g.: Mississippi River',
        initial=river,
        error=river_error
    )

    date_built = DatePicker(
        name='date-built',
        display_text='Date Built',
        autoclose=True,
        format='MM d, yyyy',
        start_view='decade',
        today_button=True,
        initial=date_built,
        error=date_error
    )
    ...
```

Tip: Form Validation Pattern: The example above implements a common pattern for handling and validating form input. Generally, the steps are:

1. **define a "value" variable for each input in the form and assign it the initial value for the input**
2. **define an "error" variable for each input to handle error messages and initially set them to the empty string**
3. **check to see if the form is submitted and if the form has been submitted:**
    a. extract the value of each input from the GET or POST parameters and overwrite the appropriate value variable from step 1
    b. validate the value of each input, assigning an error message (if any) to the appropriate error variable from step 2 for each input with errors.
    c. if there are no errors, save or process the data, and then redirect to a different page
    d. if there are errors continue on and re-render from with error messages
4. **define all gizmos and variables used to populate the form:**
    a. pass the value variable created in step 1 to the `initial` argument of the corresponding gizmo
    b. pass the error variable created in step 2 to the `error` argument of the corresponding gizmo
5. **render the page, passing all gizmos to the template through the context**

3. Create the Model and File IO

Now that we are able to get information about new dams to add to the dam inventory from the user, we need to persist the data to some sort of database. It's time to create the Model for the app.

In this tutorial we will start with a file database model to illustrate how to work with files in Tethys apps. In the *Advanced Concepts* tutorial we will convert this file database model to an SQL database model. Here is an overview of the file-based model:

- One text file will be created per dam
- The name of the file will be the id of the dam (e.g.: *a1e26591-d6bb-4194-b4a7-1222fe0195fd.json*)
- The files will be stored in the **app workspace** (a directory provided by the app for storing files).
- Each file will contain a single JSON object with the following structure:

```
{
  "id": "a1e26591-d6bb-4194-b4a7-1222fe0195fd",
  "name": "Deer Creek",
  "owner": "Reclamation",
  "river": "Provo River",
  "date_built": "June 16, 2017"
}
```

Tip: For more information on file workspaces see the *Workspaces API*.

Warning: File database models can be problematic for web applications, especially in a production environment. We recommend using and SQL or other database that can handle concurrent requests and heavy traffic.

a. Open `model.py` and add a new function called `add_new_dam`:

```
import os
import uuid
import json
from .app import DamInventory as app


def add_new_dam(name, owner, river, date_built):
    """
    Persist new dam.
    """
    # Serialize data to json
    new_dam_id = uuid.uuid4()
    dam_dict = {
        'id': str(new_dam_id),
        'name': name,
        'owner': owner,
        'river': river,
        'date_built': date_built
    }

    dam_json = json.dumps(dam_dict)

    # Write to file in app_workspace/dams/{{uuid}}.json
```

(continues on next page)

(continued from previous page)

```
    # Make dams dir if it doesn't exist
    app_workspace = app.get_app_workspace()
    dams_dir = os.path.join(app_workspace.path, 'dams')
    if not os.path.exists(dams_dir):
        os.mkdir(dams_dir)

    # Name of the file is its id
    file_name = str(new_dam_id) + '.json'
    file_path = os.path.join(dams_dir, file_name)

    # Write json
    with open(file_path, 'w') as f:
        f.write(dam_json)
```

b. Modify `add_dam` controller to use the new `add_new_dam` model function to persist the dam data:

```
from .model import add_new_dam

...

@login_required()
def add_dam(request):
    """
    Controller for the Add Dam page.
    """
    ...

    # Handle form submission
    if request.POST and 'add-button' in request.POST:

        ...

        if not has_errors:
            add_new_dam(name=name, owner=owner, river=river, date_built=date_built)
            return redirect(reverse('dam_inventory:home'))

        ...
```

c. Use the Add Dam page to add several dams for the Dam Inventory app.

d. Navigate to `workspaces/app_workspace/dams` to see the JSON files that are being written.

4. Develop Table View Page

Todo: Implement with table view rather than raw HTML table

Now that the data is being persisted in our make-shift inventory database, let's create useful views of the data in our inventory. First, we'll create a new page that lists all of the dams in our inventory database in a table, which will provide a good review of Model View Controller:

a. Open `models.py` and add a model method for listing the dams called `get_all_dams`:

```
def get_all_dams():
    """
```

(continues on next page)

(continued from previous page)

```
    Get all persisted dams.
    """
    # Write to file in app_workspace/dams/{{uuid}}.json
    # Make dams dir if it doesn't exist
    app_workspace = app.get_app_workspace()
    dams_dir = os.path.join(app_workspace.path, 'dams')
    if not os.path.exists(dams_dir):
        os.mkdir(dams_dir)

    dams = []

    # Open each file and convert contents to python objects
    for dam_json in os.listdir(dams_dir):
        # Make sure we are only looking at json files
        if '.json' not in dam_json:
            continue

        dam_json_path = os.path.join(dams_dir, dam_json)
        with open(dam_json_path, 'r') as f:
            dam_dict = json.loads(f.readlines()[0])
            dams.append(dam_dict)

    return dams
```

b. Add a new template `/templates/dam_inventory/list_dams.html` with the following contents:

```
{% extends "dam_inventory/base.html" %}
{% load tethys_gizmos %}

{% block app_content %}
  <h1>Dams</h1>
  {% gizmo dams_table %}
{% endblock %}
```

c. Create a new controller function in `controllers.py` called `list_dams`:

```
from tethys_sdk.gizmos import DataTableView
from .model import get_all_dams

...

@login_required()
def list_dams(request):
    """
    Show all dams in a table view.
    """
    dams = get_all_dams()
    table_rows = []

    for dam in dams:
        table_rows.append(
            (
                dam['name'], dam['owner'],
                dam['river'], dam['date_built']
            )
        )
```

(continues on next page)

(continued from previous page)

```
    dams_table = DataTableView(
        column_names=('Name', 'Owner', 'River', 'Date Built'),
        rows=table_rows,
        searching=False,
        orderClasses=False,
        lengthMenu=[ [10, 25, 50, -1], [10, 25, 50, "All"] ],
    )

    context = {
        'dams_table': dams_table
    }

    return render(request, 'dam_inventory/list_dams.html', context)
```

d. Create a new URL Map in the `app.py` for the new `list_dams` controller:

```
class DamInventory(TethysAppBase):
    """
    Tethys app class for Dam Inventory.
    """
    ...

    def url_maps(self):
        """
        Add controllers
        """
        UrlMap = url_map_maker(self.root_url)

        url_maps = (
            ...

            UrlMap(
                name='dams',
                url='dam-inventory/dams',
                controller='dam_inventory.controllers.list_dams'
            ),
        )

        return url_maps
```

e. Open `/templates/dam_inventory/base.html` and add navigation links for the List View page:

```
{% block app_navigation_items %}
  {% url 'dam_inventory:home' as home_url %}
  {% url 'dam_inventory:add_dam' as add_dam_url %}
  {% url 'dam_inventory:dams' as list_dam_url %}
  <li class="title">Navigation</li>
  <li class="{% if request.path == home_url %}active{% endif %}"><a href="{{ home_url␣
↪}}">Home</a></li>
  <li class="{% if request.path == list_dam_url %}active{% endif %}"><a href="{{ list_
↪dam_url }}">Dams</a></li>
  <li class="{% if request.path == add_dam_url %}active{% endif %}"><a href="{{ add_
↪dam_url }}">Add Dam</a></li>
{% endblock %}
```

5. Spatial Input with Forms

In this section, we'll add a Map View gizmo to the Add Dam form to allow users to provide the location of the dam as another attribute.

a. Open `/templates/dam_inventory/add_dam.html` and add the `location_input` gizmo to the form:

```
{% extends "dam_inventory/base.html" %}
{% load tethys_gizmos %}

{% block app_content %}
  <h1>Add Dam</h1>
  <form id="add-dam-form" method="post">
    {% csrf_token %}
    <div class="form-group{% if location_error %} has-error{% endif %}">
      <label class="control-label">Location</label>
      {% gizmo location_input %}
      {% if location_error %}<p class="help-block">{{ location_error }}</p>{% endif %}
    </div>
    {% gizmo name_input %}
    {% gizmo owner_input %}
    {% gizmo river_input %}
    {% gizmo date_built_input %}
  </form>
{% endblock %}

{% block app_actions %}
  {% gizmo add_button %}
  {% gizmo cancel_button %}
{% endblock %}
```

b. Add the definition of the `location_input` gizmo and validation code to the `add_dam` controller in `controllers.py`:

```
from tethys_sdk.gizmos import MVDraw, MVView

...

@login_required()
def add_dam(request):
    """
    Controller for the Add Dam page.
    """
    # Default Values
    location = ''
    ...

    # Errors
    location_error = ''
    ...

    # Handle form submission
    if request.POST and 'add-button' in request.POST:
        # Get values
        has_errors = False
        location = request.POST.get('geometry', None)
        ...
```

(continues on next page)

(continued from previous page)

```
        # Validate
        if not location:
            has_errors = True
            location_error = 'Location is required.'

        ...

        if not has_errors:
            add_new_dam(location=location, name=name, owner=owner, river=river, date_
↪built=date_built)
            return redirect(reverse('dam_inventory:home'))

        messages.error(request, "Please fix errors.")

    # Define form gizmos
    initial_view = MVView(
        projection='EPSG:4326',
        center=[-98.6, 39.8],
        zoom=3.5
    )

    drawing_options = MVDraw(
        controls=['Modify', 'Delete', 'Move', 'Point'],
        initial='Point',
        output_format='GeoJSON',
        point_color='#FF0000'
    )

    location_input = MapView(
        height='300px',
        width='100%',
        basemap='OpenStreetMap',
        draw=drawing_options,
        view=initial_view
    )

    ...

    context = {
        'location_input': location_input,
        'location_error': location_error,
        ...
    }

    return render(request, 'dam_inventory/add_dam.html', context)
```

c. Modify the `add_new_dam` Model Method to store spatial data:

```
def add_new_dam(location, name, owner, river, date_built):
    """
    Persist new dam.
    """
    # Convert GeoJSON to Python dictionary
    location_dict = json.loads(location)

    # Serialize data to json
```

(continues on next page)

(continued from previous page)

```
    new_dam_id = uuid.uuid4()
    dam_dict = {
        'id': str(new_dam_id),
        'location': location_dict['geometries'][0],
        'name': name,
        'owner': owner,
        'river': river,
        'date_built': date_built
    }

    dam_json = json.dumps(dam_dict)

    # Write to file in app_workspace/dams/{{uuid}}.json
    # Make dams dir if it doesn't exist
    app_workspace = app.get_app_workspace()
    dams_dir = os.path.join(app_workspace.path, 'dams')
    if not os.path.exists(dams_dir):
        os.mkdir(dams_dir)

    # Name of the file is its id
    file_name = str(new_dam_id) + '.json'
    file_path = os.path.join(dams_dir, file_name)

    # Write json
    with open(file_path, 'w') as f:
        f.write(dam_json)
```

d. Navigate to `workspaces/app_workspace/dams` and delete all JSON files now that the model has changed, so that all the files will be consistent.

e. Create several new entries using the updated Add Dam form.

6. Render Spatial Data on Map

Finally, we'll add logic to the home controller to display all of the dams in our dam inventory on the map.

a. Modify the `home` controller in `controllers.py` to map the list of dams:

```
from tethys_sdk.gizmos import MVLayer

...

@login_required()
def home(request):
    """
    Controller for the app home page.
    """
    # Get list of dams and create dams MVLayer:
    dams = get_all_dams()
    features = []
    lat_list = []
    lng_list = []

    # Define GeoJSON Features
    for dam in dams:
        dam_location = dam.pop('location')
```

(continues on next page)

(continued from previous page)

```
        lat_list.append(dam_location['coordinates'][1])
        lng_list.append(dam_location['coordinates'][0])

        dam_feature = {
            'type': 'Feature',
            'geometry': {
                'type': dam_location['type'],
                'coordinates': dam_location['coordinates'],
            }
        }

        features.append(dam_feature)

    # Define GeoJSON FeatureCollection
    dams_feature_collection = {
        'type': 'FeatureCollection',
        'crs': {
            'type': 'name',
            'properties': {
                'name': 'EPSG:4326'
            }
        },
        'features': features
    }

    # Create a Map View Layer
    dams_layer = MVLayer(
        source='GeoJSON',
        options=dams_feature_collection,
        legend_title='Dams',
        layer_options={
            'style': {
                'image': {
                    'circle': {
                        'radius': 10,
                        'fill': {'color':  '#d84e1f'},
                        'stroke': {'color': '#ffffff', 'width': 1},
                    }
                }
            }
        }
    )

    # Define view centered on dam locations
    try:
        view_center = [sum(lng_list) / float(len(lng_list)), sum(lat_list) /␣
↪float(len(lat_list))]
    except ZeroDivisionError:
        view_center = [-98.6, 39.8]

    view_options = MVView(
        projection='EPSG:4326',
        center=view_center,
        zoom=4.5,
        maxZoom=18,
        minZoom=2
    )
```

(continues on next page)

(continued from previous page)

```
    dam_inventory_map = MapView(
        height='100%',
        width='100%',
        layers=[dams_layer],
        basemap='OpenStreetMap',
        view=view_options
    )

    add_dam_button = Button(
        display_text='Add Dam',
        name='add-dam-button',
        icon='glyphicon glyphicon-plus',
        style='success',
        href=reverse('dam_inventory:add_dam')
    )

    context = {
        'dam_inventory_map': dam_inventory_map,
        'add_dam_button': add_dam_button
    }

    return render(request, 'dam_inventory/home.html', context)
```

7. Solution

This concludes the Intermediate Tutorial. You can view the solution on GitHub at https://github.com/tethysplatform/tethysapp-dam_inventory or clone it as follows:

```
$ mkdir ~/tethysdev
$ cd ~/tethysdev
$ git clone https://github.com/tethysplatform/tethysapp-dam_inventory.git
$ cd tethysapp-dam_inventory
$ git checkout intermediate-solution
```

Advanced Concepts

Last Updated: June 2017

This tutorial introduces advanced concepts for Tethys developers. The topics covered include:

- Tethys Services API
- PersistentStores API
- Gizmo JavaScript APIs
- JavaScript and AJAX
- Permissions API
- Advanced HTML forms - File Upload
- Plotting Gizmos

0. Start From Intermediate Solution (Optional)

If you wish to use the intermediate solution as a starting point:

```
$ git clone https://github.com/tethysplatform/tethysapp-dam_inventory.git
$ cd tethysapp-dam_inventory
$ git checkout intermediate-solution
```

1. Persistent Store Database

In the *Intermediate Concepts* tutorial we implemented a file-based database as the persisting mechanism for the app. However, simple file based databases typically don't perform well in a web application environment, because of the possibility of many concurrent requests trying to access the file. In this section we'll refactor the Model to use an SQL database, rather than files.

a. Open the `app.py` and define a new `PersistentStoreDatabaseSetting` by adding the `persistent_store_settings` method to your app class:

```
from tethys_sdk.app_settings import PersistentStoreDatabaseSetting

class DamInventory(TethysAppBase):
    """
    Tethys app class for Dam Inventory.
    """
    ...
    def persistent_store_settings(self):
        """
        Define Persistent Store Settings.
        """
        ps_settings = (
            PersistentStoreDatabaseSetting(
                name='primary_db',
                description='primary database',
                initializer='dam_inventory.model.init_primary_db',
                required=True
            ),
        )

        return ps_settings
```

Tethys provides the library SQLAlchemy as an interface with SQL databases. SQLAlchemy provides an Object Relational Mapper (ORM) API, which allows data models to be defined using Python and an object-oriented approach. With SQLAlchemy, you can harness the power of SQL databases without writing SQL. As a primer to SQLAlchemy ORM, we highly recommend you complete the Object Relational Tutorial.

b. Define a table called `dams` by creating a new class in `model.py` called `Dam`:

```
import json
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, Float, String
from sqlalchemy.orm import sessionmaker

from .app import DamInventory as app

Base = declarative_base()
```

(continues on next page)

(continued from previous page)

```
# SQLAlchemy ORM definition for the dams table
class Dam(Base):
    """
    SQLAlchemy Dam DB Model
    """
    __tablename__ = 'dams'

    # Columns
    id = Column(Integer, primary_key=True)
    latitude = Column(Float)
    longitude = Column(Float)
    name = Column(String)
    owner = Column(String)
    river = Column(String)
    date_built = Column(String)
```

Tip: SQLAlchemy Data Models: Each class in an SQLAlchemy data model defines a table in the database. The model you defined above consists of a single table called "dams", as denoted by the `__tablename__` property of the `Dam` class. The `Dam` class inherits from a `Base` class that we created in the previous lines from the `declarative_base` function. This inheritance notifies SQLAlchemy that the `Dam` class is part of the data model.

The class defines seven other properties that are instances of SQLAlchemy `Column` class: *id*, *latitude*, *longitude*, *name*, *owner*, *river*, *date_built*. These properties define the columns of the "dams" table. The column type and options are defined by the arguments passed to the `Column` class. For example, the *latitude* column is of type `Float` while the *id* column is of type `Integer`. The `id` column is flagged as the primary key for the table. IDs will be generated for each object when they are committed.

This class is not only used to define the tables for your persistent store, it is also used to create new entries and query the database.

For more information on Persistent Stores, see: *Persistent Stores API*.

c. Refactor the `add_new_dam` and `get_all_dams` functions in `model.py` to use the SQL database instead of the files:

```
def add_new_dam(location, name, owner, river, date_built):
    """
    Persist new dam.
    """
    # Convert GeoJSON to Python dictionary
    location_dict = json.loads(location)
    location_geometry = location_dict['geometries'][0]
    longitude = location_geometry['coordinates'][0]
    latitude = location_geometry['coordinates'][1]

    # Create new Dam record
    new_dam = Dam(
        latitude=latitude,
        longitude=longitude,
        name=name,
        owner=owner,
        river=river,
        date_built=date_built
    )
```

(continues on next page)

(continued from previous page)

```
    # Get connection/session to database
    Session = app.get_persistent_store_database('primary_db', as_sessionmaker=True)
    session = Session()

    # Add the new dam record to the session
    session.add(new_dam)

    # Commit the session and close the connection
    session.commit()
    session.close()


def get_all_dams():
    """
    Get all persisted dams.
    """
    # Get connection/session to database
    Session = app.get_persistent_store_database('primary_db', as_sessionmaker=True)
    session = Session()

    # Query for all dam records
    dams = session.query(Dam).all()
    session.close()

    return dams
```

Important: Don't forget to close your `session` objects when you are done. Eventually you will run out of connections to the database if you don't, which will cause unsightly errors.

d. Create a new function called `init_primary_db` at the bottom of `model.py`. This function is used to initialize the data database by creating the tables and adding any initial data.

```
def init_primary_db(engine, first_time):
    """
    Initializer for the primary database.
    """
    # Create all the tables
    Base.metadata.create_all(engine)

    # Add data
    if first_time:
        # Make session
        Session = sessionmaker(bind=engine)
        session = Session()

        # Initialize database with two dams
        dam1 = Dam(
            latitude=40.406624,
            longitude=-111.529133,
            name="Deer Creek",
            owner="Reclamation",
            river="Provo River",
            date_built="April 12, 1993"
        )
```

(continues on next page)

(continued from previous page)

```
        dam2 = Dam(
            latitude=40.598168,
            longitude=-111.424055,
            name="Jordanelle",
            owner="Reclamation",
            river="Provo River",
            date_built="1941"
        )

        # Add the dams to the session, commit, and close
        session.add(dam1)
        session.add(dam2)
        session.commit()
        session.close()
```

e. Refactor `home` controller in `controllers.py` to use new model objects:

```
@login_required()
def home(request):
    """
    Controller for the app home page.
    """
    # Get list of dams and create dams MVLayer:
    dams = get_all_dams()
    features = []
    lat_list = []
    lng_list = []

    for dam in dams:
        lat_list.append(dam.latitude)
        lng_list.append(dam.longitude)

        dam_feature = {
            'type': 'Feature',
            'geometry': {
                'type': 'Point',
                'coordinates': [dam.longitude, dam.latitude],

            },
            'properties': {
                'id': dam.id,
                'name': dam.name,
                'owner': dam.owner,
                'river': dam.river,
                'date_built': dam.date_built
            }
        }
        features.append(dam_feature)

    ...
```

f. Refactor the `list_dams` controller to use the new model objects:

```
@login_required()
def list_dams(request):
    """
```

(continues on next page)

(continued from previous page)

```
    Show all dams in a table view.
    """
    dams = get_all_dams()
    table_rows = []

    for dam in dams:
        table_rows.append(
            (
                dam.name, dam.owner,
                dam.river, dam.date_built
            )
        )

    ...
```

g. Add **Persistent Store Service** to Tethys Portal:

   a. Go to Tethys Portal Home in a web browser (e.g. http://localhost:8000/apps/)

   b. Select **Site Admin** from the drop down next to your username.

   c. Scroll down to **Tethys Services** section and select **Persistent Store Services** link.

   d. Click on the **Add Persistent Store Service** button.

   e. Give the **Persistent Store Service** a name and fill out the connection information.

Important: The username and password for the persistent store service must be a superuser to use spatial persistent stores.

---

h. Assign **Persistent Store Service** to Dam Inventory App:

   a. Go to Tethys Portal Home in a web browser (e.g. http://localhost:8000/apps/)

   b. Select **Site Admin** from the drop down next to your username.

   c. Scroll down to **Tethys Apps** section and select **Installed App** link.

   d. Select the **Dam Inventory** link.

   e. Scroll down to the **Persistent Store Database Settings** section.

   f. Assign the **Persistent Store Service** that you created in Step 4 to the **primary_db**.

i. Execute **syncstores** command to initialize Persistent Store database:

```
(tethys) $ tethys syncstores dam_inventory
```

2. Use Custom Settings

In the *Beginner Concepts* tutorial, we created a custom setting named *max_dams*. In this section, we'll show you how to use the custom setting in one of your controllers.

a. Modify the *add_dam* controller, such that it won't add a new dam if the *max_dams* limit has been reached:

```
from .model import Dam
from .app import DamInventory as app
```

(continues on next page)

(continued from previous page)

```
...

@login_required()
def add_dam(request):
    """
    Controller for the Add Dam page.
    """

    ...

    # Handle form submission
    if request.POST and 'add-button' in request.POST:

        ...

        if not has_errors:
            # Get value of max_dams custom setting
            max_dams = app.get_custom_setting('max_dams')

            # Query database for count of dams
            Session = app.get_persistent_store_database('primary_db', as_
↪sessionmaker=True)
            session = Session()
            num_dams = session.query(Dam).count()

            # Only add the dam if we have not exceed max_dams
            if num_dams < max_dams:
                add_new_dam(location=location, name=name, owner=owner, river=river,␣
↪date_built=date_built)
            else:
                messages.warning(request, 'Unable to add dam "{0}", because the␣
↪inventory is full.'.format(name))

            return redirect(reverse('dam_inventory:home'))

        messages.error(request, "Please fix errors.")

    ...
```

Tip: For more information on app settings, see *App Settings API*.

3. Use JavaScript APIs

JavaScript is the programming language that is used to program web browsers. You can use JavaScript in you Tethys apps to enrich the user experience and add dynamic effects. Many of the Tethys Gizmos include JavaScript APIs to allow you to access the underlying JavaScript objects and library to customize them. In this section, we'll use the JavaScript API of the Map View gizmo to add pop-ups to the map whenever the users clicks on one of the dams.

a. Modify the MVLayer in the `home` controller to make the layer selectable:

```
...

dams_layer = MVLayer(
```

(continues on next page)

(continued from previous page)

```
    ...

    feature_selection=True
)

...
```

b. Create a new file called `/public/js/map.js` and add the following contents:

```
$(function()
{
    // Create new Overlay with the #popup element
    var popup = new ol.Overlay({
        element: document.getElementById('popup')
    });

    // Get the Open Layers map object from the Tethys MapView
    var map = TETHYS_MAP_VIEW.getMap();

    // Get the Select Interaction from the Tethys MapView
    var select_interaction = TETHYS_MAP_VIEW.getSelectInteraction();

    // Add the popup overlay to the map
    map.addOverlay(popup);

    // When selected, call function to display properties
    select_interaction.getFeatures().on('change:length', function(e)
    {
        var popup_element = popup.getElement();

        if (e.target.getArray().length > 0)
        {
            // this means there is at least 1 feature selected
            var selected_feature = e.target.item(0); // 1st feature in Collection

            // Get coordinates of the point to set position of the popup
            var coordinates = selected_feature.getGeometry().getCoordinates();

            var popup_content = '<div class="dam-popup">' +
                                    '<p><b>' + selected_feature.get('name') + '</b></
↪p>' +
                                    '<table class="table  table-condensed">' +
                                        '<tr>' +
                                            '<th>Owner:</th>' +
                                            '<td>' + selected_feature.get('owner') + '
↪</td>' +
                                        '</tr>' +
                                        '<tr>' +
                                            '<th>River:</th>' +
                                            '<td>' + selected_feature.get('river') + '
↪</td>' +
                                        '</tr>' +
                                        '<tr>' +
                                            '<th>Date Built:</th>' +
                                            '<td>' + selected_feature.get('date_built
↪') + '</td>' +
```

(continues on next page)

(continued from previous page)

```
                                        '</tr>' +
                                    '</table>' +
                                '</div>';

            // Clean up last popup and reinitialize
            $(popup_element).popover('destroy');

            // Delay arbitrarily to wait for previous popover to
            // be deleted before showing new popover.
            setTimeout(function() {
                popup.setPosition(coordinates);

                $(popup_element).popover({
                  'placement': 'top',
                  'animation': true,
                  'html': true,
                  'content': popup_content
                });

                $(popup_element).popover('show');
            }, 500);
        } else {
            // remove pop up when selecting nothing on the map
            $(popup_element).popover('destroy');
        }
    });
});
```

c. Open `/templates/dam_inventory/home.html`, add a new `div` element to the `app_content` area of the page with an id `popup`, and load the `map.js` script to the bottom of the page:

```
...

{% block app_content %}
  {% gizmo dam_inventory_map %}
  <div id="popup"></div>
{% endblock %}

...

{% block scripts %}
  {{ block.super }}
  <script src="{% static 'dam_inventory/js/map.js' %}" type="text/javascript"></
↪script>
{% endblock %}
```

d. Open `public/css/map.css` and add the following contents:

```
...

.popover-content {
    width: 240px;
}
```

4. App Permissions

By default, any user logged into the app can access any part of it. You may want to restrict access to certain areas of the app to privileged users. This can be done using the *Permissions API*. Let's modify the app so that only admin users of the app can add dams to the app.

a. Define permissions for the app by adding the `permissions` method to the app class in the `app.py`:

```
...

from tethys_sdk.permissions import Permission, PermissionGroup

class DamInventory(TethysAppBase):
    """
    Tethys app class for Dam Inventory.
    """
    ...

    def permissions(self):
        """
        Define permissions for the app.
        """
        add_dams = Permission(
            name='add_dams',
            description='Add dams to inventory'
        )

        admin = PermissionGroup(
            name='admin',
            permissions=(add_dams,)
        )

        permissions = (admin,)

        return permissions
```

b. Protect the Add Dam view with the `add_dams` permission by replacing the `login_required` decorator with the `permission_required` decorator to the `add_dams` controller:

```
from tethys_sdk.permissions import permission_required

...

@permission_required('add_dams')
def add_dam(request):
    """
    Controller for the Add Dam page.
    """
    ...
```

c. Add a context variable called `can_add_dams` to the context of each controller with the value of the return value of the `has_permission` function:

```
from tethys_sdk.permissions import has_permission

@login_required()
def home(request):
```

(continues on next page)

(continued from previous page)

```
    """
    Controller for the app home page.
    """
    ...

    context = {
        ...
        'can_add_dams': has_permission(request, 'add_dams')
    }

    return render(request, 'dam_inventory/home.html', context)


@permission_required('add_dams')
def add_dam(request):
    """
    Controller for the Add Dam page.
    """
    ...

    context = {
        ...
        'can_add_dams': has_permission(request, 'add_dams')
    }

    return render(request, 'dam_inventory/add_dam.html', context)


@login_required()
def list_dams(request):
    """
    Show all dams in a table view.
    """
    dams = get_all_dams()
    context = {
        ...
        'can_add_dams': has_permission(request, 'add_dams')
    }
    return render(request, 'dam_inventory/list_dams.html', context)
```

d. Use the `can_add_dams` variable to determine whether to show or hide the navigation link to the Add Dam View in `base.html`:

```
{% block app_navigation_items %}
  ...
  <li class="{% if request.path == home_url %}active{% endif %}"><a href="{{ home_url␣
↪}}">Home</a></li>
  <li class="{% if request.path == list_dam_url %}active{% endif %}"><a href="{{ list_
↪dam_url }}">Dams</a></li>
  {% if can_add_dams %}
  <li class="{% if request.path == add_dam_url %}active{% endif %}"><a href="{{ add_
↪dam_url }}">Add Dam</a></li>
  {% endif %}
{% endblock %}
```

e. Use the `can_add_dams` variable to determine whether to show or hide the "Add Dam" button in `home.html`:

```
{% block app_actions %}
  {% if can_add_dams %}
    {% gizmo add_dam_button %}
  {% endif %}
{% endblock %}
```

f. The `admin` user of Tethys is a superuser and has all permissions. To test the permissions, create two new users: one with the `admin` permissions group and one without it. Then login with these users:

   a. Go to Tethys Portal Home in a web browser (e.g. http://localhost:8000/apps/)

   b. Select **Site Admin** from the drop down next to your username.

   c. Scroll to the **Authentication and Authorization** section.

   d. Select the **Users** link.

   e. Press the **Add User** button.

   f. Enter "diadmin" as the username and enter a password. Take note of the password for later.

   g. Press the **Save** button.

   h. Scroll down to the **Groups** section.

   i. Select the **dam_inventory:admin** group and press the right arrow to add the user to that group.

   j. Press the **Save** button.

   k. Repeat steps e-f for user named "diviewer". DO NOT add "diviewer" user to any groups.

   l. Press the **Save** button.

g. Log in each user. If the permission has been applied correctly, "diviewer" should not be able to see the Add Dam link and should be redirected if the Add Dam view is linked to directly. "diadmin" should be able to add dams.

Tip: For more details on Permissions, see: *Permissions API*.

Todo: Split into another tutorial here?

---

5. Persistent Store Related Tables

Add Flood Hydrograph table

a. Define two new tables to `models.py` for storing the hydrograph and hydrograph points. Also, establish relationships between the tables. Each dam will have only one hydrograph and each hydrograph can have multiple hydrograph points.

```
from sqlalchemy import ForeignKey
from sqlalchemy.orm import relationship

...

class Dam(Base):
    """
    SQLAlchemy Dam DB Model
```

(continues on next page)

(continued from previous page)

```
    """
    ...

    # Relationships
    hydrograph = relationship('Hydrograph', back_populates='dam', uselist=False)


class Hydrograph(Base):
    """
    SQLAlchemy Hydrograph DB Model
    """
    __tablename__ = 'hydrographs'

    # Columns
    id = Column(Integer, primary_key=True)
    dam_id = Column(ForeignKey('dams.id'))

    # Relationships
    dam = relationship('Dam', back_populates='hydrograph')
    points = relationship('HydrographPoint', back_populates='hydrograph')


class HydrographPoint(Base):
    """
    SQLAlchemy Hydrograph Point DB Model
    """
    __tablename__ = 'hydrograph_points'

    # Columns
    id = Column(Integer, primary_key=True)
    hydrograph_id = Column(ForeignKey('hydrographs.id'))
    time = Column(Integer)  #: hours
    flow = Column(Float)  #: cfs

    # Relationships
    hydrograph = relationship('Hydrograph', back_populates='points')
```

b. Execute **syncstores** command again to add the new tables to the database:

```
(tethys) $ tethys syncstores dam_inventory
```

6. File Upload

CSV File Upload Create new page for uploading the hydrograph.

a. New Model function

```
def assign_hydrograph_to_dam(dam_id, hydrograph_file):
    """
    Parse hydrograph file and add to database, assigning to appropriate dam.
    """
    # Parse file
    hydro_points = []

    try:
```

(continues on next page)

(continued from previous page)

```
        for line in hydrograph_file:
            sline = line.split(',')

            try:
                time = int(sline[0])
                flow = float(sline[1])
                hydro_points.append(HydrographPoint(time=time, flow=flow))
            except ValueError:
                continue

        if len(hydro_points) > 0:
            Session = app.get_persistent_store_database('primary_db', as_
↪sessionmaker=True)
            session = Session()

            # Get dam object
            dam = session.query(Dam).get(int(dam_id))

            # Overwrite old hydrograph
            hydrograph = dam.hydrograph

            # Create new hydrograph if not assigned already
            if not hydrograph:
                hydrograph = Hydrograph()
                dam.hydrograph = hydrograph

            # Remove old points if any
            for hydro_point in hydrograph.points:
                session.delete(hydro_point)

            # Assign points to hydrograph
            hydrograph.points = hydro_points

            # Persist to database
            session.commit()
            session.close()

    except Exception as e:
        # Careful not to hide error. At the very least log it to the console
        print(e)
        return False

    return True
```

b. New Template: `assign_hydrograph.html`

```
{% extends "dam_inventory/base.html" %}
{% load tethys_gizmos %}

{% block app_content %}
  <h1>Assign Hydrograph</h1>
  <p>Select a dam and a hydrograph file to assign to that dam. The file should be a␣
↪csv with two columns: time (hours) and flow (cfs).</p>
  <form id="add-hydrograph-form" method="post" enctype="multipart/form-data">
    {% csrf_token %}
    {% gizmo dam_select_input %}
```

(continues on next page)

(continued from previous page)

```
    <div class="form-group{% if hydrograph_file_error %} has-error{% endif %}">
      <label class="control-label">Hydrograph File</label>
      <input type="file" name="hydrograph-file">
      {% if hydrograph_file_error %}<p class="help-block">{{ hydrograph_file_error }}
↪</p>{% endif %}
    </div>
  </form>
{% endblock %}

{% block app_actions %}
  {% gizmo cancel_button %}
  {% gizmo add_button %}
{% endblock %}
```

c. New Controller

```
from .model import assign_hydrograph_to_dam

...

@login_required()
def assign_hydrograph(request):
    """
    Controller for the Add Hydrograph page.
    """
    # Get dams from database
    Session = app.get_persistent_store_database('primary_db', as_sessionmaker=True)
    session = Session()
    all_dams = session.query(Dam).all()

    # Defaults
    dam_select_options = [(dam.name, dam.id) for dam in all_dams]
    selected_dam = None
    hydrograph_file = None

    # Errors
    dam_select_errors = ''
    hydrograph_file_error = ''

    # Case where the form has been submitted
    if request.POST and 'add-button' in request.POST:
        # Get Values
        has_errors = False
        selected_dam = request.POST.get('dam-select', None)

        if not selected_dam:
            has_errors = True
            dam_select_errors = 'Dam is Required.'

        # Get File
        if request.FILES and 'hydrograph-file' in request.FILES:
            # Get a list of the files
            hydrograph_file = request.FILES.getlist('hydrograph-file')

        if not hydrograph_file and len(hydrograph_file) > 0:
            has_errors = True
            hydrograph_file_error = 'Hydrograph File is Required.'
```

(continues on next page)

(continued from previous page)

```
        if not has_errors:
            # Process file here
            success = assign_hydrograph_to_dam(selected_dam, hydrograph_file[0])

            # Provide feedback to user
            if success:
                messages.info(request, 'Successfully assigned hydrograph.')
            else:
                messages.info(request, 'Unable to assign hydrograph. Please try again.
↪')
            return redirect(reverse('dam_inventory:home'))

        messages.error(request, "Please fix errors.")

    dam_select_input = SelectInput(
        display_text='Dam',
        name='dam-select',
        multiple=False,
        options=dam_select_options,
        initial=selected_dam,
        error=dam_select_errors
    )

    add_button = Button(
        display_text='Add',
        name='add-button',
        icon='glyphicon glyphicon-plus',
        style='success',
        attributes={'form': 'add-hydrograph-form'},
        submit=True
    )

    cancel_button = Button(
        display_text='Cancel',
        name='cancel-button',
        href=reverse('dam_inventory:home')
    )

    context = {
        'dam_select_input': dam_select_input,
        'hydrograph_file_error': hydrograph_file_error,
        'add_button': add_button,
        'cancel_button': cancel_button,
        'can_add_dams': has_permission(request, 'add_dams')
    }

    session.close()

    return render(request, 'dam_inventory/assign_hydrograph.html', context)
```

d. New UrlMap

```
class DamInventory(TethysAppBase):
    """
    Tethys app class for Dam Inventory.
    """
```

(continues on next page)

(continued from previous page)

```
    ...

    def url_maps(self):
        """
        Add controllers
        """
        UrlMap = url_map_maker(self.root_url)

        url_maps = (

            ...

            UrlMap(
                name='assign_hydrograph',
                url='dam-inventory/hydrographs/assign',
                controller='dam_inventory.controllers.assign_hydrograph'
            ),
        )

        return url_maps
```

d. Update navigation

```
{% block app_navigation_items %}
  <li class="title">App Navigation</li>
  ...
  {% url 'dam_inventory:assign_hydrograph' as assign_hydrograph_url %}
  ...
  <li class="{% if request.path == assign_hydrograph_url %}active{% endif %}"><a href=
↪"{{ assign_hydrograph_url }}">Assign Hydrograph</a></li>
{% endblock %}
```

f. Test upload with these files:

`Sample Hydrograph CSVs`

7. URL Variables and Plotting

Create a new page with hydrograph plotted for selected Dam

a. Create Template `hydrograph.html`

```
{% extends "dam_inventory/base.html" %}
{% load tethys_gizmos %}

{% block app_navigation_items %}
  <li class="title">App Navigation</li>
  <li class=""><a href="{% url 'dam_inventory:dams' %}">Back</a></li>
{% endblock %}

{% block app_content %}
  {% gizmo hydrograph_plot %}
{% endblock %}
```

b. Create `helpers.py`

```
from plotly import graph_objs as go
from tethys_gizmos.gizmo_options import PlotlyView

from tethysapp.dam_inventory.app import DamInventory as app
from tethysapp.dam_inventory.model import Hydrograph


def create_hydrograph(hydrograph_id, height='520px', width='100%'):
    """
    Generates a plotly view of a hydrograph.
    """
    # Get objects from database
    Session = app.get_persistent_store_database('primary_db', as_sessionmaker=True)
    session = Session()
    hydrograph = session.query(Hydrograph).get(int(hydrograph_id))
    dam = hydrograph.dam
    time = []
    flow = []
    for hydro_point in hydrograph.points:
        time.append(hydro_point.time)
        flow.append(hydro_point.flow)

    # Build up Plotly plot
    hydrograph_go = go.Scatter(
        x=time,
        y=flow,
        name='Hydrograph for {0}'.format(dam.name),
        line={'color': '#0080ff', 'width': 4, 'shape': 'spline'},
    )
    data = [hydrograph_go]
    layout = {
        'title': 'Hydrograph for {0}'.format(dam.name),
        'xaxis': {'title': 'Time (hr)'},
        'yaxis': {'title': 'Flow (cfs)'},
    }
    figure = {'data': data, 'layout': layout}
    hydrograph_plot = PlotlyView(figure, height=height, width=width)
    session.close()
    return hydrograph_plot
```

c. Create Controller

```
from .helpers import create_hydrograph

...

@login_required()
def hydrograph(request, hydrograph_id):
    """
    Controller for the Hydrograph Page.
    """
    hydrograph_plot = create_hydrograph(hydrograph_id)

    context = {
        'hydrograph_plot': hydrograph_plot,
        'can_add_dams': has_permission(request, 'add_dams')
    }
    return render(request, 'dam_inventory/hydrograph.html', context)
```

Tip: For more information about plotting in Tethys apps, see *Plotly View*, *Bokeh View*, and *Plot View*.

d. Add UrlMap with URL Variable

```
class DamInventory(TethysAppBase):
    """
    Tethys app class for Dam Inventory.
    """
    ...

    def url_maps(self):
        """
        Add controllers
        """
        UrlMap = url_map_maker(self.root_url)

        url_maps = (
            ...

            UrlMap(
                name='hydrograph',
                url='dam-inventory/hydrographs/{hydrograph_id}',
                controller='dam_inventory.controllers.hydrograph'
            ),
        )

        return url_maps
```

e. Modify `list_dams` controller:

```
.. todo::

    Find a way to get links into data tables view
```

8. Dynamic Hydrograph Plot in Pop-Ups

Add Hydrographs to pop-ups if they exist.

a. Add Plotly Gizmo dependency to `home.html`:

```
{% extends "dam_inventory/base.html" %}
{% load tethys_gizmos staticfiles %}

{% block import_gizmos %}
  {% import_gizmo_dependency plotly_view %}
{% endblock %}

...
```

b. Create a template for the AJAX plot (`hydrograph_ajax.html`)

```
{% load tethys_gizmos %}

{% if hydrograph_plot %}
```

(continues on next page)

(continued from previous page)

```
  {% gizmo hydrograph_plot %}
{% endif %}
```

c. Create an AJAX controller `hydrograph_ajax`

```
@login_required()
def hydrograph_ajax(request, dam_id):
    """
    Controller for the Hydrograph Page.
    """
    # Get dams from database
    Session = app.get_persistent_store_database('primary_db', as_sessionmaker=True)
    session = Session()
    dam = session.query(Dam).get(int(dam_id))

    if dam.hydrograph:
        hydrograph_plot = create_hydrograph(dam.hydrograph.id, height='300px')
    else:
        hydrograph_plot = None

    context = {
        'hydrograph_plot': hydrograph_plot,
    }

    session.close()
    return render(request, 'dam_inventory/hydrograph_ajax.html', context)
```

d. Create an AJAX UrlMap

```
class DamInventory(TethysAppBase):
    """
    Tethys app class for Dam Inventory.
    """
    ...

    def url_maps(self):
        """
        Add controllers
        """
        UrlMap = url_map_maker(self.root_url)

        url_maps = (
            ...

            UrlMap(
                name='hydrograph_ajax',
                url='dam-inventory/hydrographs/{dam_id}/ajax',
                controller='dam_inventory.controllers.hydrograph_ajax'
            ),
        )

        return url_maps
```

e. Load the plot dynamically using JavaScript and AJAX (modify `map.js`)

```
$(function()
{
```

(continues on next page)

(continued from previous page)

```
    // Create new Overlay with the #popup element
    var popup = new ol.Overlay({
        element: document.getElementById('popup')
    });

    // Get the Open Layers map object from the Tethys MapView
    var map = TETHYS_MAP_VIEW.getMap();

    // Get the Select Interaction from the Tethys MapView
    var select_interaction = TETHYS_MAP_VIEW.getSelectInteraction();

    // Add the popup overlay to the map
    map.addOverlay(popup);

    // When selected, call function to display properties
    select_interaction.getFeatures().on('change:length', function(e)
    {
        var popup_element = popup.getElement();

        if (e.target.getArray().length > 0)
        {
            // this means there is at least 1 feature selected
            var selected_feature = e.target.item(0); // 1st feature in Collection

            // Get coordinates of the point to set position of the popup
            var coordinates = selected_feature.getGeometry().getCoordinates();

            // Load hydrograph dynamically with AJAX
            $.ajax({
                url: '/apps/dam-inventory/hydrographs/' + selected_feature.get('id')␣
↪+ '/ajax/',
                method: 'GET',
                success: function(plot_html) {
                    var popup_content = '<div class="dam-popup">' +
                        '<p><b>' + selected_feature.get('name') + '</b></p>' +
                        '<table class="table  table-condensed">' +
                            '<tr>' +
                                '<th>Owner:</th>' +
                                '<td>' + selected_feature.get('owner') + '</td>' +
                            '</tr>' +
                            '<tr>' +
                                '<th>River:</th>' +
                                '<td>' + selected_feature.get('river') + '</td>' +
                            '</tr>' +
                            '<tr>' +
                                '<th>Date Built:</th>' +
                                '<td>' + selected_feature.get('date_built') + '</td>'␣
↪+
                            '</tr>' +
                        '</table>' +
                        plot_html +
                    '</div>';

                    // Clean up last popup and reinitialize
                    $(popup_element).popover('destroy');

                    // Delay arbitrarily to wait for previous popover to
```

(continues on next page)

(continued from previous page)

```
                    // be deleted before showing new popover.
                    setTimeout(function() {
                        popup.setPosition(coordinates);

                        $(popup_element).popover({
                          'placement': 'top',
                          'animation': true,
                          'html': true,
                          'content': popup_content
                        });

                        $(popup_element).popover('show');
                    }, 500);
                }
            });

        } else {
            // remove pop up when selecting nothing on the map
            $(popup_element).popover('destroy');
        }
    });
});
```

f. Update `map.css`:

```
.popover-content {
    width: 400px;
    max-height: 300px;
    overflow-y: auto;
}

.popover {
    max-width: none;
}

#inner-app-content {
    padding: 0;
}

#app-content, #inner-app-content, #map_view_outer_container {
    height: 100%;
}
```

9. Solution

This concludes the Advanced Tutorial. You can view the solution on GitHub at https://github.com/tethysplatform/tethysapp-dam_inventory or clone it as follows:

```
$ git clone https://github.com/tethysplatform/tethysapp-dam_inventory.git
$ cd tethysapp-dam_inventory
$ git checkout advanced-solution
```

1.4.2 Spatial Dataset Services and GeoServer

Last Updated: May 2017

This tutorial will walk you through the steps of working with GeoServer in a Tethys App project using Tethys Platform. If you have not already installed Tethys Platform, follow the *Installation* documentation and then return.

This tutorial will make use of the *GeoServer Docker* and the *Spatial Dataset Services API*.

The solution to this tutorial is available at: https://github.com/swainn/tethysapp-geoserver_app

Start and Register

Last Updated: May 2017

Scaffold New App

Create a new app, but don't install it yet:

```
$ t
(tethys) $ tethys scaffold geoserver_app
```

Create Spatial Dataset Service Setting

Open the `app.py` and add the following method to the `GeoserverApp` class:

```
from tethys_sdk.app_settings import SpatialDatasetServiceSetting

class GeoserverApp(TethysAppBase):
    """
    Tethys app class for Geoserver App.
    """
    ...

    def spatial_dataset_service_settings(self):
        """
        Example spatial_dataset_service_settings method.
        """
        sds_settings = (
            SpatialDatasetServiceSetting(
                name='main_geoserver',
                description='spatial dataset service for app to use',
                engine=SpatialDatasetServiceSetting.GEOSERVER,
                required=True,
            ),
        )

        return sds_settings
```

Install GeoServer and Start Tethys Development Server

```
(tethys) $ cd tethysapp-geoserver_app
(tethys) $ python setup.py develop
(tethys) $ tms
```

Start GeoServer

If you are using the Docker containers, start up your *GeoServer Docker* container:

```
(tethys) $ tethys docker start -c geoserver
```

Otherwise ensure that you have GeoServer installed and running. Refer to the GeoServer Installation Guide for system specific instructions.

Create GeoServer Spatial Dataset Service

Register the GeoServer with Tethys Portal admin page by creating a Spatial Dataset Service:

1. Select the "Site Admin" link from the drop down menu by your username.
2. Scroll to the "Tethys Services" section and select the "Spatial Dataset Services" link.
3. Create a new Spatial Dataset Service named "primary_geoserver" of type GeoServer.
4. Enter the endpoint and public endpoint as the same (e.g.: http://localhost:8181/geoserver/rest/ if using Docker or http://localhost:8080/geoserver/rest/ if using a default installation of GeoServer).
5. Fill out the username and password (default username and password is "admin" and "geoserver", respectively.
6. No API Key is required.
7. Press "Save".

Assign Spatial Dataset Service to App Setting

Assign the "primary_geoserver" Spatial Dataset Service to the "main_geoserver" setting for the app.

1. Select the "Site Admin" link from the drop down menu by your username.
2. Scroll to the "Tethys Apps" section and select the "Installed Apps" link.
3. Select the "Geoserver App" link.
4. Scroll down to the "Spatial Dataset Service Settings" section and assign the "primary_geoserver" to the Spatial Dataset Service property of the "main_geoserver" setting for the app.
5. Press "Save".

Tip:

If you don't see the "main_geoserver" setting in the "Spatial Dataset Service Settings" section try restarting the Tethys development server. If it still doesn't show up, then stop the Tethys development server, uninstall the app, reinstall it, and start the Tethys server again:

```
(tethys) $ tethys uninstall geoserver_app
(tethys) $ cd tethysapp-geoserver_app
(tethys) $ python setup.py develop
(tethys) $ tms
```

GeoServer Web Admin Interface

Explore the GeoServer web admin interface by visiting link: http://\T1\textless{}host\T1\textgreater{}:\T1\textless{}port\T1\textgreater{}/geoserver/web/.

Download Test Files

Download the sample shapefiles that you will use to test your app:

`geoserver_app_data.zip`

Upload Shapefile

Last Updated: May 2017

Add Form to Home Page

Replace the contents of the existing `home.html` template with:

```
{% extends "geoserver_app/base.html" %}

{% block app_content %}
  <h1>Upload a Shapefile</h1>
  <form action="" method="post" enctype="multipart/form-data">.
    {% csrf_token %}
    <div class="form-group">
        <label for="fileInput">Shapefiles</label>
        <input name="files" type="file" multiple class="form-control" id="fileInput"␣
↪placeholder="Shapefiles">
    </div>
    <input name="submit" type="submit" class="btn btn-default">
  </form>
{% endblock %}
```

Handle File Upload in Home Controller

Replace the contents of `controllers.py` module with the following:

```
import random
import string

from django.shortcuts import render
from django.contrib.auth.decorators import login_required
```

(continues on next page)

(continued from previous page)

```
from tethys_sdk.gizmos import *
from .app import GeoserverApp as app


WORKSPACE = 'geoserver_app'
GEOSERVER_URI = 'http://www.example.com/geoserver-app'


@login_required
def home(request):
    """
    Controller for the app home page.
    """
    # Retrieve a geoserver engine
    geoserver_engine = app.get_spatial_dataset_service(name='main_geoserver', as_
↪engine=True)

    # Check for workspace and create workspace for app if it doesn't exist
    response = geoserver_engine.list_workspaces()

    if response['success']:
        workspaces = response['result']

        if WORKSPACE not in workspaces:
            geoserver_engine.create_workspace(workspace_id=WORKSPACE, uri=GEOSERVER_
↪URI)

    # Case where the form has been submitted
    if request.POST and 'submit' in request.POST:
        # Verify files are included with the form
        if request.FILES and 'files' in request.FILES:
            # Get a list of the files
            file_list = request.FILES.getlist('files')

            # Upload shapefile
            store = ''.join(random.choice(string.ascii_lowercase + string.digits) for␣
↪_ in range(6))
            store_id = WORKSPACE + ':' + store
            geoserver_engine.create_shapefile_resource(
                store_id=store_id,
                shapefile_upload=file_list,
                overwrite=True
            )

    context = {}

    return render(request, 'geoserver_app/home.html', context)
```

Test Shapefile Upload

Go to the home page of your app located at http://localhost:8000/apps/geoserver-app/. You should see a form with a file input ("Browse" button or similar) and a submit button. To test this page, select the "Browse" button and upload one of the shapefiles from the data that you downloaded earlier. Remember that for the shapefile to be valid, you need to select at least the files with the extensions "shp", "shx", and "dbf". Press submit to upload the files.

Use the GeoServer web admin interface (http://localhost:8181/geoserver/web/) to verify that the layers were successfully uploaded. Look for layers belonging to the workspace 'geoserver_app'.

Map GeoServer Layers

Last Updated: May 2017

Map Page UrlMap

Add a new `UrlMap` to the `url_maps` method of the `app.py` module:

```
UrlMap(
    name='map',
    url='geoserver-app/map',
    controller='geoserver_app.controllers.map'
),
```

Map Page Controller

Add a new controller to the `controller.py` module:

```
@login_required
def map(request):
    """
    Controller for the map page
    """
    geoserver_engine = app.get_spatial_dataset_service(name='main_geoserver', as_
↪engine=True)

    options = []

    response = geoserver_engine.list_layers(with_properties=False)

    if response['success']:
        for layer in response['result']:
            options.append((layer.title(), layer))

    select_options = SelectInput(
        display_text='Choose Layer',
        name='layer',
        multiple=False,
        options=options
    )

    map_layers = []
```

(continues on next page)

(continued from previous page)

```
    if request.POST and 'layer' in request.POST:
        selected_layer = request.POST['layer']
        legend_title = selected_layer.title()

        geoserver_layer = MVLayer(
            source='ImageWMS',
            options={
                'url': 'http://localhost:8181/geoserver/wms',
                'params': {'LAYERS': selected_layer},
                'serverType': 'geoserver'
            },
            legend_title=legend_title,
            legend_extent=[-114, 36.5, -109, 42.5],
            legend_classes=[
                MVLegendClass('polygon', 'County', fill='#999999'),
        ])

        map_layers.append(geoserver_layer)


    view_options = MVView(
        projection='EPSG:4326',
        center=[-100, 40],
        zoom=4,
        maxZoom=18,
        minZoom=2
    )

    map_options = MapView(
        height='500px',
        width='100%',
        layers=map_layers,
        legend=True,
        view=view_options
    )

    context = {'map_options': map_options,
               'select_options': select_options}

    return render(request, 'geoserver_app/map.html', context)
```

Map Page Template

Create a new `map.html` template in your template directory and add the following contents:

```
{% extends "geoserver_app/base.html" %}
{% load tethys_gizmos %}

{% block app_content %}
    <h1>GeoServer Layers</h1>
    <form method="post">
        {% csrf_token %}
        {% gizmo select_input select_options %}
        <input name="submit" type="submit" value="Update" class="btn btn-default">
    </form>
```

(continues on next page)

(continued from previous page)

```
    {% gizmo map_options %}
{% endblock %}
```

Test Map Page

Navigate to the map page (http://localhost:8000/apps/geoserver-app/map/). Use the select box to select a layer to display on the map. Press the submit button to effect the change.

Spatial Input

Last Updated: May 2017

Spatial Input Page UrlMap

Add a new `UrlMap` to the `url_maps` method of the `app.py` module:

```
UrlMap(
    name='draw',
    url='geoserver-app/draw',
    controller='geoserver_app.controllers.draw'
),
```

Spatial Input Controller

Add a new controller to the `controller.py` module:

```
@login_required
def draw(request):
    drawing_options = MVDraw(
        controls=['Modify', 'Move', 'Point', 'LineString', 'Polygon', 'Box'],
        initial='Polygon'
    )

    map_options = MapView(
        height='450px',
        width='100%',
        layers=[],
        draw=drawing_options
    )

    geometry = ''

    if request.POST and 'geometry' in request.POST:
        geometry = request.POST['geometry']

    context = {'map_options': map_options,
               'geometry': geometry}

    return render(request, 'geoserver_app/draw.html', context)
```

Spatial Input Template

Create a new `draw.html` template in your template directory and add the following contents:

```
{% extends "geoserver_app/base.html" %}
{% load tethys_gizmos %}

{% block app_content %}
    <h1>Draw on the Map</h1>

    {% if geometry %}
        <p>{{ geometry }}</p>
    {% endif %}

    <form method="post">
        {% csrf_token %}
        {% gizmo map_view map_options %}
        <input name="submit" type="submit">
    </form>
{% endblock %}
```

Add Navigation Links

Replace the `app_navigation_items` block of the `base.html` template with:

```
{% block app_navigation_items %}
  <li class="title">App Navigation</li>
  {% url 'geoserver_app:home' as home_url %}
  {% url 'geoserver_app:map' as map_url %}
  {% url 'geoserver_app:draw' as draw_url %}
  <li class="{% if request.path == home_url %}active{% endif %}"><a href="{{ home_url␣
↪}}">Upload Shapefile</a></li>
  <li class="{% if request.path == map_url %}active{% endif %}"><a href="{{ map_url }}
↪">GeoServer Layers</a></li>
  <li class="{% if request.path == draw_url %}active{% endif %}"><a href="{{ draw_url␣
↪}}">Draw</a></li>
{% endblock %}
```

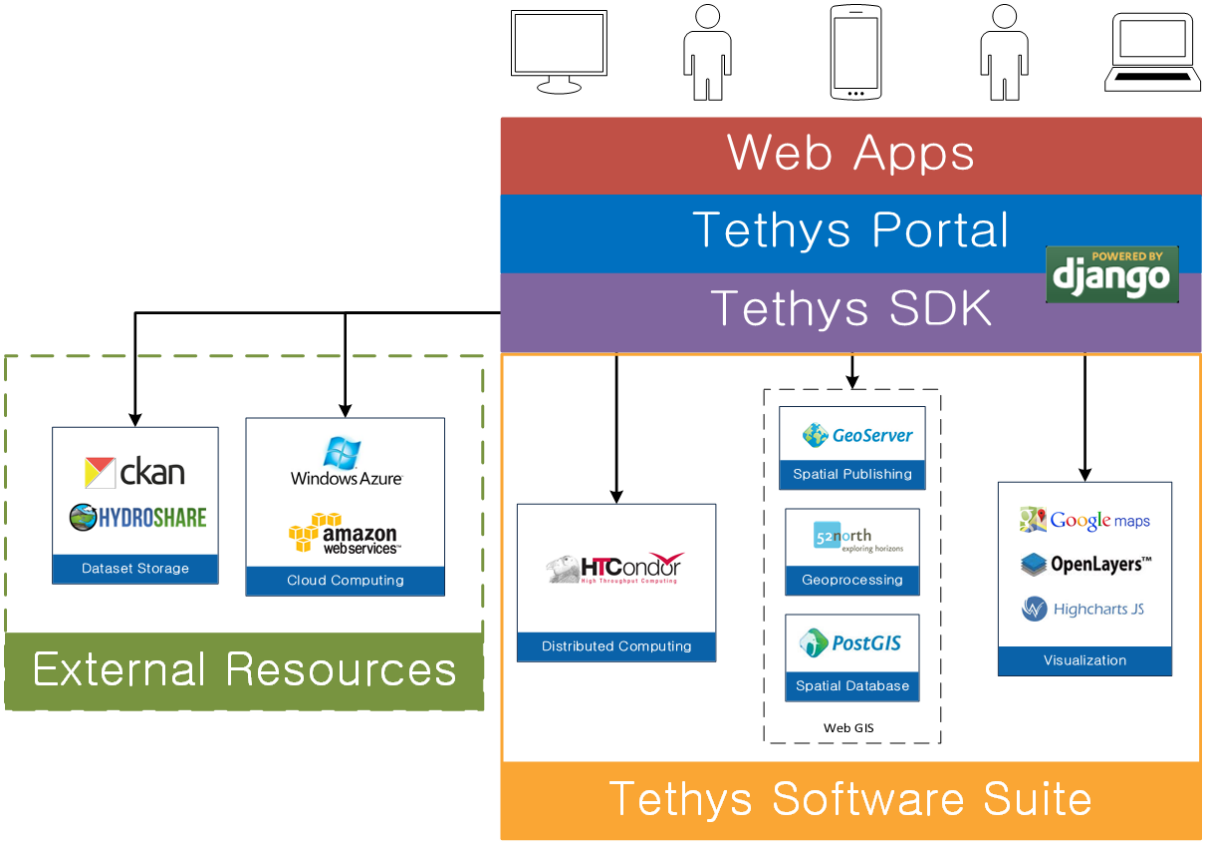
Test Spatial Input Page

Navigate to the spatial input page using the "Draw" link in your navigation (http://localhost:8000/apps/geoserver-app/draw/). Use the drawing controls to add features to the map, then press the submit button. The GeoJSON encoded spatial data should be displayed when the page refreshes.

1.5 Software Suite

Last Updated: May 18, 2016

The Software Suite is the component of Tethys Platform that provides access to resources and functionality that are commonly required to develop water resources web apps. The primary motivation of creating the Tethys Software Suite was overcome the hurdle associated with selecting a GIS software stack to support spatial capabilities in apps. Some of the more specialized needs for water resources app development arise from the spatial data components of the models that are used in the apps. Distributed hydrologic models, for example, are parameterized using raster or vector layers such as land use maps, digital elevation models, and rainfall intensity grids.



A majority of the software projects included in the software suite are web GIS projects that can be used to acquire, modify, store, visualize, and analyze spatial data, but Tethys Software Suite also includes other software projects to address computing and visualization needs of water resources web apps. This article will describe the components included in the Tethys Software Suite in terms of the functionality provided by the software.

1.5.1 Spatial Database Storage

Tethys Software Suite includes the PostgreSQL database with PostGIS, a spatial database extension, to provide spatial data storage capabilities for Tethys web apps. PostGIS adds spatial column types including raster, geometry, and geography. The extension also provides database functions for basic analysis of GIS objects.

To use a PostgreSQL database in your app use the *Persistent Stores API*. To use a spatially enabled database with PostGIS use the *Spatial Persistent Stores API*.

1.5.2 Map Publishing

Tethys Software Suite provides GeoServer for publishing spatial data as web services. GeoServer is used to publish common spatial files such as Shapefiles and GeoTIFFs in web-friendly formats.

To use the map publishing capabilities of GeoServer in your app refer to the *GeoServer Docker* documentation and use the *Spatial Dataset Services API*.

1.5.3 Geoprocessing

52°North Web Processing Service (WPS) is included in Tethys Software Suite as one means for supporting geoprocessing needs in water resources web app development. It can be linked with geoprocessing libraries such as GRASS, Sextante, and ArcGIS® Server for out-of-the-box geoprocessing capabilities.

The PostGIS extension, included in the software suite, can also provide geoprocessing capabilities on data that is stored in a spatially-enabled database. PostGIS includes SQL geoprocessing functions for splicing, dicing, morphing, reclassifying, and collecting/unioning raster and vector types. It also includes functions for vectorizing rasters, clipping rasters with vectors, and running stats on rasters by geometric region.

To use 52°North WPS or other WPS geoprocessing services in your app use the *Web Processing Services API*.

1.5.4 Visualization

OpenLayers 3 is a JavaScript web-mapping client library for rendering interactive maps on a web page. It is capable of displaying 2D maps of OGC web services and a myriad of other spatial formats and sources including GeoJSON, KML, GML, TopoJSON, ArcGIS REST, and XYZ.

To use an OpenLayers map in your app use the **Map View Gizmo** of the *Template Gizmos API*.

Google Maps™ provides the ability to render spatial data in a 2D mapping environment similar to OpenLayers, but it only supports displaying data in KML formats and data that are added via JavaScript API. Both maps provide a mechanism for drawing on the map for user input.

To use an OpenLayers map in your app use the **Google Map View Gizmo** of the *Template Gizmos API*.

Plotting capabilities are provided by Highcharts, a JavaScript library created by Highsoft AS. The plots created using Highcharts are interactive with hovering effects, pan and zoom capabilities, and the ability to export the plots as images.

To use an OpenLayers map in your app use the **Plot View Gizmo** of the *Template Gizmos API*.

1.5.5 Distributed Computing

To facilitate the large-scale computing that is often required by water resources applications, Tethys Software Suite leverages the computing management middleware HTCondor. HTCondor is both a resource management and a job scheduling software.

To use the HTCondor and the computing capabilities in your app use the *Jobs API* and the *Compute API*.

1.5.6 File Dataset Storage

Tethys Software Suite does not include software for handling flat file storage. However, Tethys SDK provides APIs for working with CKAN and HydroShare to address flat file storage needs. Descriptions of CKAN and HydroShare are provided here for convenience.

CKAN is an open source data sharing platform that streamlines publishing, sharing, finding, and using data. There is no central CKAN hub or portal, rather data publishers setup their own instance of CKAN to host the data for their organization.

HydroShare is an online hydrologic model and data sharing portal being developed by CUAHSI. It builds on the sharing capabilities of CUAHSI's Hydrologic Information System by adding support for sharing models and using social media functionality.

To use a CKAN instance for flat file storage in your app use the *Dataset Services API*. HydroShare is not fully supported at this time, but when it is you will use the *Dataset Services API* to access HydroShare resources.

1.5.7 Docker Installation

Tethys Software Suite uses Docker virtual container system to simplify the installation of some elements. Docker images are created and used to create containers, which are essentially stripped down virtual machines running only the software included in the image. Unlike virtual machines, the Docker containers do not partition the resources of your computer (processors, RAM, storage), but instead run as processes with full access to the resources of the computer.

Three Docker images are provided as part of Tethys Software Suite including:

- PostgreSQL with PostGIS
- 52° North WPS
- GeoServer.

The installation procedure for each software has been encapsulated in a Docker image reducing the installation procedure to three simple steps:

1. Install Docker
2. Download the Docker images
3. Deploy the Docker images as containers

1.5.8 SDK Relationships

Tethys Platform provides a software development kit (SDK) that provides application programming interfaces (APIs) for interacting with each of the software included in teh Software Suite. The appropriate APIs are referenced in each section above, but a summary table of the relationship between the Software Suite and the SDK is provided as a reference.

Software	API	Functionality
PostgreSQL	*Persistent Stores API*	SQL Database Storage
PostGIS	*Spatial Persistent Stores API*	Spatial Database Storage and Geoprocessing
GeoServer	*Spatial Dataset Services API*	Spatial File Publishing
52° North WPS	*Web Processing Services API*	Geoprocessing Services
OpenLayers, Google Maps, HighCharts	*Template Gizmos API*	Spatial and Tabular Visualization
HTCondor	*Compute API* and *Jobs API*	Computing and Job Management
CKAN, HydroShare	*Dataset Services API*	Flat File Storage

1.5.9 References

GeoServer Docker

Last Updated: December 10, 2016

Features

- The GeoServer docker was updated to version 2.8.3
- It can be configured to run in clustered mode (multiple instances of GeoServer running inside the container) for greater stability and performance
- Several extensions are now included:
  - JMS Clustering
  - Flow Control
  - CSS Styles
  - NetCDF
  - NetCDF Output
  - GDAL WCS Output
  - Image Pyramid

Installation

Installing the GeoServer Docker is done using the Tethys Command Line Interface (see: *docker <subcommand> [options]*). To install it, open a terminal, activate the Tethys virtual environment, and execute the command:

```
$ . /usr/lib/tethys/bin/activate
$ tethys docker init -c geoserver
```

This command will initiate the download of the GeoServer Docker image. Once the image finishes downloading it will be used to create a Docker container and you will be prompted to configure it. Here is a brief explanation of each of the configuration options:

- **GeoServer Instances Enabled**: This is the number of parallel running GeoServer's to start up when the docker starts. All of the GeoServer instances share the same directory and remain synced via the JMS clustering extension (see: JMS Clustering Documentation). Access to the instances is automatically load balanced via NGINX. The load-balanced cluster of GeoServers is accessed using port 8181 and this should be used as the endpoint for you GeoServer docker. You will notice that the identifier of the node appears in the top left corner of the GeoServer admin page. When accessing the admin page of the cluster using port 8181, you will always be redirected to the first node. Any changes to this node will be synced to the other nodes, so usually it will be sufficient to administer the GeoServer this way. However, you can access the admin pages of each node directly using ports 8081-8084, respectively, for troubleshooting.
- **GeoServer Instances with REST Enabled**: The number of running GeoServer instances that have the REST API enabled. Tethys Dataset Services uses the rest API to manage data (create, read, update, delete) in GeoServer. It is a good idea to leave a few of your GeoServer nodes as read-only (REST disabled) to retain access to GeoServer data even when it is processing data. To configure it this way, be sure this number is less than the number of enabled GeoServer nodes.
- **Control Flow Options**: The control flow extenion allows you to limit the number of requests that are allowed to run simultaneously, placing any excess requests into a queue to be executed late. This prevents your GeoServer from becoming overwhelmed during heavy traffic periods. There are two ways to configure control flow during setup:
  1. Automatically derive flow control options based on the number of cores of your computer (recommended for development or inexperienced developers)
  2. Explicitly set several of the most useful options (useful for a production installation and more experienced developers)

  Note: If you bind the geoserver data directory to the host machine (highly recommended), you can edit these options by editing the `controlflow.properties` file which is located in the geoserver data directory. Refer to the Control Flow documentation for more details (see: Control Flow Documentation).

- **Max Timeout**: The amount of time in seconds to wait before terminating a request.
- **Min and Max Memory**: The amount of memory to allocate as heap space for each GeoServer instance. It is usually a good idea to set the min to be the same as the max to avoid the overhead of allocating additional memory if it is needed. 2 GB per instance is probably the maximum you will need for this and the default of 1 GB is likely to be sufficient for many installations.

  Warning: BE CAREFUL WITH THIS. If you set the min memory to be 2 GB per instance and 4 instances enabled, GeoServer will try to allocate 8GB of memory. If your machine doesn't have 8GB of memory, it will get overwhelmed and lock down.

- **Bind the GeoServer data directory to the Host** (HIGHLY RECOMMENDED): This allows you to mount one of the directories on your machine into the docker container. Long story short, this will give you direct access to the GeoServer data directory outside of the docker container. This is useful if you want to configure your controlflow.properties, add data directly to the data directory, or view the files that were uploaded for debugging. The GeoServer docker container will automatically add the demo data to this directory after starting up the first time.

> **Warning:** If the directory that you are binding to doesn't exist or you don't have permission to write to it, the setup operation may fail. To be safe you should create the directory before hand and ensure you can write to it.

Migrate to New GeoServer Docker

Use these instructions to migrate the data in a GeoServer 2.7.0 Docker to a newer version. You can see the version of GeoServer is displayed on the main admin page of GeoServer.

1. Extract data from GeoServer docker (the container that Tethys creates for GeoServer is named tethys_geoserver)

```
$ mkdir ~/backup
$ cd ~/backup
$ docker run --rm --volumes-from tethys_geoserver -v $(pwd):/backup␣
↪ubuntu:14.04 tar cvf /backup/backup.tar /var/lib/geoserver/data
```

2. Rename old GeoServer docker container as a backup and verify that it was renamed

```
$ docker rename tethys_geoserver tethys_geoserver_bak
$ docker ps -a
```

3. Pull new docker container (only in Tethys versions 1.4.0+)

```
$ . /usr/lib/tethys/bin/activate
(tethys) $ tethys docker init
```

4. Respond to the prompts to configure the new GeoServer container, which can be configured to run in a clustered mode (refer to the explanation of the configuration parameters in the installation instructions).
5. After the new GeoServer installs, start it up and visit the admin page (http://localhost:8181/geoserver) to make sure it is working properly. This also adds the data from the GeoServer to the data directory on the host, so DON'T SKIP THIS STEP. When you are done stop the GeoServer docker.

```
(tethys) $ tethys docker start -c geoserver
(tethys) $ tethys docker stop -c geoserver
```

6. Browse to the directory where you bound the GeoServer data directory (default is /usr/lib/tethys/geoserver):

```
$ cd /usr/lib/tethys/geoserver
$ ls -alh data/
```

7. You should see the contents of the data directory for the GeoServer docker container. Notice that everything is owned by root. This is because the container runs with the root user. To restore the data from your old container, you will need to delete the contents of this directory and copy over the the data in the tar file in ~/backup.

```
$ sudo rm -rf data/
$ cp ~/backup/backup.tar .
```

(continues on next page)

(continued from previous page)

```
$ tar xvf backup.tar --strip 3
$ rm backup.tar
```

8. Listing the contents of data again, you should see the data restored from your previous GeoServer docker:

```
$ ls -alh data/
```

9. Start up the GeoServer container again.

```
(tethys) $ tethys docker start -c geoserver
```

10. The layer preview and some other features of GeoServer will not work properly until you set the Proxy Base URL due to the clustered configuration of the GeoServer. Navigate to *Settings* > *Global* and locate the Proxy Base URL field and enter the external URL of your GeoServer (e.g.: http://localhost:8181/geoserver).

Note: Logging in as admin: sometimes it doesn't work the first time (or second, third or forth for that matter). Try, try again until it works.

---

11. Once you are confident that the data has been successfully migrated from the old GeoServer container to the new one, you should delete the old GeoServer container:

```
$ docker rm tethys_geoserver_bak
```

1.6 Software Development Kit

Last Updated: February 22, 2018

The Tethys Platform provides a Python Software Development Kit (SDK) to make it easier to incorporate the functionality of the various supported software packages into apps. The SDK is includes an Application Programming Interface (API) for each of the major software components of Tethys Platform. This section contains the documentation for each API that is included in the SDK:

1.6.1 Command Line Interface

Last Updated: November 18, 2014

The Tethys Command Line Interface (CLI) provides several commands that are used for managing Tethys Platform and Tethys apps. The *Python conda environment* must be activated to use the command line tools. This can be done using the following command:

```
$ ./usr/lib/tethys/bin/activate
```

The following article provides and explanation for each command provided by Tethys CLI.

Usage

```
$ tethys <command> [options]
```

Options

- **-h, --help**: Request the help information for Tethys CLI or any command.

Commands

scaffold <name>

This command is used to create new Tethys app projects via the scaffold provided by Tethys Platform. You will be presented with several interactive prompts requesting metadata information that can be included with the app. The new app project will be created in the current working directory of your terminal.

Arguments:

- **name**: The name of the new Tethys app project to create. Only lowercase letters, numbers, and underscores are allowed.

Optional Arguments:

- **-t TEMPLATE, --template TEMPLATE**: Name of app template to use.
- **-e EXTENSION, --extension EXTENSION**: Name of extension template to use [UNDER DEVELOPMENT].
- **-d, --defaults**: Run command, accepting default values automatically.
- **-o, --overwrite**: Attempt to overwrite project automatically if it already exists.

Examples:

```
$ tethys scaffold my_first_app
```

gen <type>

Aids the installation of Tethys by automating the creation of supporting files.

Arguments:

- **type**: The type of object to generate. Either "settings" or "apache".
  - *settings*: When this type of object is specified, **gen** will generate a new `settings.py` file. It generates the `settings.py` with a new `SECRET_KEY` each time it is run.
  - *apache*: When this type of object is specified **gen** will generate a new `apache.conf` file. This file is used to configure Tethys Platform in a production environment.

Optional Arguments:

- **-d DIRECTORY, --directory DIRECTORY**: Destination directory for the generated object.

Examples:

```
$ tethys gen settings
$ tethys gen settings -d /path/to/destination
$ tethys gen apache
$ tethys gen apache -d /path/to/destination
```

manage <subcommand> [options]

This command contains several subcommands that are used to help manage Tethys Platform.

Arguments:

- **subcommand**: The management command to run.
  - *start*: Start the Django development server. Wrapper for `manage.py runserver`.
  - *syncdb*: Initialize the database during installation. Wrapper for `manage.py syncdb`.
  - *sync*: Sync installed apps and extensions with the TethysApp database.
  - *collectstatic*: Link app and extension static/public directories to STATIC_ROOT directory and then run Django's collectstatic command. Preprocessor and wrapper for `manage.py collectstatic`.
  - *collectworkspaces*: Link app workspace directories to TETHYS_WORKSPACES_ROOT directory.
  - *collectall*: Convenience command for running both *collectstatic* and *collectworkspaces*.
  - *createsuperuser*: Create a new superuser/website admin for your Tethys Portal.

Optional Arguments:

- **-p PORT, --port PORT**: Port on which to start the development server. Default port is 8000.
- **-m MANAGE, --manage MANAGE**: Absolute path to `manage.py` file for Tethys Platform installation if different than default.

Examples:

```
# Start the development server
$ tethys manage start
$ tethys manage start -p 8888

# Sync the database
$ tethys manage syncdb

# Sync installed apps with the TethysApp database.
$ tethys manage sync

# Collect static files
$ tethys manage collectstatic

# Collect workspaces
$ tethys manage collectworkspaces

# Collect static files and workspaces
$ tethys manage collectall

# Create a new superuser
$ tethys manage createsuperuser
```

syncstores <app_name, app_name...> [options]

Management command for Persistent Stores. To learn more about persistent stores see *Persistent Stores API*.

Arguments:

- **app_name**: Name of one or more apps to target when performing persistent store sync OR "all" to sync all persistent stores on this Tethys Platform instance.

Optional Arguments:

- **-r, --refresh**: Drop databases prior to performing persistent store sync resulting in a refreshed database.
- **-f, --firsttime**: All initialization functions will be executed with the `first_time` parameter set to `True`.
- **-d DATABASE, --database DATABASE**: Name of the persistent store database to target.
- **-m MANAGE, --manage MANAGE**: Absolute path to `manage.py` file for Tethys Platform installation if different than default.

Examples:

```
# Sync all persistent store databases for one app
$ tethys syncstores my_first_app

# Sync all persistent store databases for multiple apps
$ tethys syncstores my_first_app my_second_app yet_another_app

# Sync all persistent store databases for all apps
$ tethys syncstores all

# Sync a specific persistent store database for an app
$ tethys syncstores my_first_app -d example_db

# Sync persistent store databases with a specific name for all apps
$ tethys syncstores all -d example_db

# Sync all persistent store databases for an app and force first_time to True
$ tethys syncstores my_first_app -f

# Refresh all persistent store databases for an app
$ tethys syncstores my_first_app -r
```

list

Use this command to list all installed apps and extensions.

Examples:

```
$ tethys list
```

uninstall <app>

Use this command to uninstall apps and extensions.

Arguments:

- **name**: Name the app or extension to uninstall.

Optional Arguments: * **-e, --extension**: Flag used to indicate that the item being uninstalled is an extension.

Examples:

```
# Uninstall my_first_app
$ tethys uninstall my_first_app

# Uninstall extension
$ tethys uninstall -e my_extension
```

docker <subcommand> [options]

Management commands for the Tethys Docker containers. To learn more about Docker, see What is Docker?.

Arguments:

- **subcommand**: The docker command to run. One of the following:
  - *init*: Initialize the Tethys Dockers including, starting Boot2Docker if applicable, pulling the Docker images, and installing/creating the Docker containers.
  - *start*: Start the Docker containers.
  - *stop*: Stop the Docker containers.
  - *restart*: Restart the Docker containers.
  - *status*: Display status of each Docker container.
  - *update*: Pull the latest version of the Docker images.
  - *remove*: Remove a Docker images.
  - *ip*: Display host, port, and endpoint of each Docker container.

Optional Arguments:

- **-d, --defaults**: Install Docker containers with default values (will not prompt for input). Only applicable to *init* subcommand.
- **-c {postgis, geoserver, wps} [{postgis, geoserver, wps} ...], --containers {postgis, geoserver, wps} [{postgis, geoserver, wps} ...]**: Execute subcommand only on the container(s) specified.
- **-b, --boot2docker**: Also stop Boot2Docker when *stop* subcommand is called with this option.

Examples:

```
# Initialize Tethys Dockers
$ tethys docker init

# Initialize with Default Parameters
$ tethys docker init -d

# Start all Tethys Dockers
```

(continues on next page)

(continued from previous page)

```
$ tethys docker start

# Start only PostGIS Docker
$ tethys docker start -c postgis

# Start PostGIS and GeoServer Docker
$ tethys docker start -c postgis geoserver

# Stop Tethys Dockers
$ tethys docker stop

# Stop Tethys Dockers and Boot2Docker if applicable
$ tethys docker stop -b

# Update Tethys Docker Images
$ tethys docker update

# Remove Tethys Docker Images
$ tethys docker remove

# View Status of Tethys Dockers
$ tethys docker status

# View Host and Port Info
$ tethys docker ip
```

test [options]

Management commands for running tests for Tethys Platform and Tethys Apps. See *Testing API*.

Optional Arguments:

- **-c, --coverage**: Run coverage with tests and output report to console.
- **-C, --coverage-html**: Run coverage with tests and output html formatted report.
- **-u, --unit**: Run only unit tests.
- **-g, --gui**: Run only gui tests. Mutually exclusive with -u. If both flags are set, then -u takes precedence.
- **-f FILE, --file FILE**: File or directory to run test in. If a directory, recursively searches for tests starting at this directory. Overrides -g and -u.

Examples:

```
# Run all tests
tethys test

# Run all unit tests with coverage report
tethys test -u -c

# Run all gui tests
tethys test -g

# Run tests for a single app
tethys test -f tethys_apps.tethysapp.my_first_app
```

app_settings <app_name>

This command is used to list the Persistent Store and Spatial Dataset Settings that an app has requested.

Arguments:

- **app_name**: Name of app for which Settings will be listed

Optional Arguments:

- **-p --persistent**: A flag indicating that only Persistent Store Settings should be listed
- **-s --spatial**: A flag indicating that only Spatial Dataset Settings should be listed

Examples:

```
$ tethys app_settings my_first_app
```

services <subcommand> [<subsubcommand> | options]

This command is used to interact with Tethys Services from the command line, rather than the App Admin interface.

Arguments:

- **subcommand**: The services command to run. One of the following:
  - *list*: List all existing Tethys Services (Persistent Store and Spatial Dataset Services)
  - ***create*: Create a new Tethys Service**
    - **subcommand: The service type to create**
      - ***persistent*: Create a new Persistent Store Service Arguments:**

        -n, --name: A unique name to identify the service being created

        -c, --connection: The connection endpoint associated with this service, in the form "<username>:<password>@<host>:<port>"
      - ***spatial*: Create a new Spatial Dataset Service Arguments:**

        -n, --name: A unique name to identify the service being created

        -c, --connection: The connection endpoint associated with this service, in the form "<username>:<password>@<protocol>//<host>:<port>"

        Optional Arguments:

        -p, --public-endpoint: The public-facing endpoint of the Service, if different than what was provided with the "--connection" argument, in the form "<protocol>//<host>:<port>".

        -k, --apikey: The API key, if any, required to establish a connection.
  - ***remove*: Remove a Tethys Service**
    - **subcommand: The service type to remove**
      - ***persistent*: Remove a Persistent Store Service Arguments:** * **service_uid**: A unique identifier of the Service to be removed, which can either be the database ID, or the service name
      - ***spatial*: Remove a Spatial Dataset Service Arguments:** * **service_uid**: A unique identifier of the Service to be removed, which can either be the database ID, or the service name

Examples:

```
# List all Tethys Services
$ tethys services list

# List only Spatial Dataset Tethys Services
$ tethys services list -s

# List only Persistent Store Tethys Services
$ tethys services list -p

# Create a new Spatial Dataset Tethys Service

$ tethys services create spatial -n my_spatial_service -c my_username:my_
↪password@http://127.0.0.1:8081 -p https://mypublicdomain.com -k mysecretapikey

# Create a new Persistent Store Tethys Service
$ tethys services create persistent -n my_persistent_service -c my_username:my_
↪password@http://127.0.0.1:8081

# Remove a Spatial Dataset Tethys Service
$ tethys services remove my_spatial_service

# Remove a Persistent Store Tethys Service
$ tethys services remove my_persistent_service
```

link <service_identifier> <app_setting_identifier>

This command is used to link a Tethys Service with a TethysApp Setting

Arguments:

- **service_identifier**: An identifier of the Tethys Service being linked, of the form "<service_type>:<service_uid>", where <service_type> can be either "spatial" or "persistent", and <service_uid> must be either the database ID or name of the Tethys Service.
- **app_setting_identifier**: An identifier of the TethysApp Setting being linked, of the form "<app_package>:<setting_type>:<setting_uid>", where <setting_type> must be one of "ds_spatial," "ps_connection", or "ps_database" and <setting_uid> can be either the database ID or name of the TethysApp Setting.

Examples:

```
# Link a Persistent Store Service to a Persistent Store Connection Setting
$ tethys link persistent:my_persistent_service my_first_app:ps_connection:my_ps_
↪connection

# Link a Persistent Store Service to a Persistent Store Database Setting
$ tethys link persistent:my_persistent_service my_first_app:ps_database:my_ps_
↪connection

# Link a Spatial Dataset Service to a Spatial Dataset Service Setting
$ tethys link spatial:my_spatial_service my_first_app:ds_spatial:my_spatial_connection
```

schedulers <subcommand>

This command is used to interact with Schedulers from the command line, rather than through the App Admin interface

Arguments:

- **subcommand**: The schedulers command to run. One of the following:
  - *list*: List all existing Schedulers
  - ***create*: Create a new Scheduler Arguments:** * **-n, --name**: A unique name to identify the Scheduler being created * **-d, --endpoint**: The endpoint of the remote host the Scheduler will connect with in the form <protocol>//<host>" * **-u, --username**: The username that will be used to connect to the remote endpoint" **Optional Arguments:** * **-p, --password**: The password associated with the username (required if "-f (--private-key-path)" not specified. * **-f, --private-key-path**: The path to the private ssh key file (required if "-p (--password)" not specified. * **-k, --private-key-pass**: The password to the private ssh key file (only meaningful if "-f (--private-key-path)" is specified.
  - ***remove*: Remove a Scheduler Arguments:** * **scheduler_name**: The unique name of the Scheduler being removed.

Examples:

```
# List all Schedulers
$ tethys schedulers list

# Create a new scheduler
$ tethys schedulers create -n my_scheduler -e http://127.0.0.1 -u my_username -p my_
↪password

# Remove a scheduler
$ tethys schedulers remove my_scheduler
```

1.6.2 App Base Class API

Last Updated: May 2017

Tethys apps are configured via the *app class*, which is contained in the *app configuration file* (`app.py`) of the *app project*. The *app class* must inherit from the `TethysAppBase` to be recognized by Tethys. The following article contains the API documentation for the `TethysAppBase` class.

Properties

class tethys_apps.base.**TethysAppBase**

Base class used to define the app class for Tethys apps.

name

Name of the app.

Type string

index

Lookup term for the index URL of the app.

Type string

icon

Location of the image to use for the app icon.

Type string

package
Name of the app package.
Type string

root_url
Root URL of the app.
Type string

color
App theme color as RGB hexadecimal.
Type string

description
Description of the app.
Type string

tag
A string for filtering apps.
Type string

enable_feedback
Shows feedback button on all app pages.
Type boolean

feedback_emails
A list of emails corresponding to where submitted feedback forms are sent.
Type list

Override Methods

`TethysAppBase.`**`url_maps`**`()`
Override this method to define the URL Maps for your app. Your `UrlMap` objects must be created from a `UrlMap` class that is bound to the `root_url` of your app. Use the `url_map_maker()` function to create the bound `UrlMap` class. If you generate your app project from the scaffold, this will be done automatically.
Returns A list or tuple of `UrlMap` objects.
Return type iterable
Example:

```
from tethys_sdk.base import url_map_maker

class MyFirstApp(TethysAppBase):

    def url_maps(self):
        """
        Example url_maps method.
        """
        # Create UrlMap class that is bound to the root url.
        UrlMap = url_map_maker(self.root_url)

        url_maps = (UrlMap(name='home',
                           url='my-first-app',
                           controller='my_first_app.controllers.home',
                           ),
        )

        return url_maps
```

TethysAppBase.**permissions**()

Override this method to define permissions for your app.

Returns A list or tuple of `Permission` or `PermissionGroup` objects.

Return type iterable

Example:

```
from tethys_sdk.permissions import Permission, PermissionGroup

class MyFirstApp(TethysAppBase):

    def permissions(self):
        """
        Example permissions method.
        """
        # Viewer Permissions
        view_map = Permission(
            name='view_map',
            description='View map'
        )

        delete_projects = Permission(
            name='delete_projects',
            description='Delete projects'
        )

        create_projects = Permission(
            name='create_projects',
            description='Create projects'
        )

        admin = PermissionGroup(
            name='admin',
            permissions=(delete_projects, create_projects)
        )


        permissions = (admin, view_map)

        return permissions
```

TethysAppBase.**custom_settings**()

Override this method to define custom settings for use in your app.

Returns A list or tuple of `CustomSetting` objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import CustomSetting

class MyFirstApp(TethysAppBase):

    def custom_settings(self):
        """
        Example custom_settings method.
        """
        custom_settings = (
            CustomSetting(
                name='default_name',
```

(continues on next page)

(continued from previous page)

```
                type=CustomSetting.TYPE_STRING
                description='Default model name.',
                required=True
            ),
            CustomSetting(
                name='max_count',
                type=CustomSetting.TYPE_INTEGER,
                description='Maximum allowed count in a method.',
                required=False
            ),
            CustomSetting(
                name='change_factor',
                type=CustomSetting.TYPE_FLOAT,
                description='Change factor that is applied to some process.',
                required=True
            ),
            CustomSetting(
                name='enable_feature',
                type=CustomSetting.TYPE_BOOLEAN,
                description='Enable this feature when True.',
                required=True
            )
        )

        return custom_settings
```

TethysAppBase.**persistent_store_settings**()

Override this method to define a persistent store service connections and databases for your app.

Returns A list or tuple of PersistentStoreDatabaseSetting or PersistentStoreConnectionSetting objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import PersistentStoreDatabaseSetting,␣
↪PersistentStoreConnectionSetting

class MyFirstApp(TethysAppBase):

    def persistent_store_settings(self):
        """
        Example persistent_store_settings method.
        """

        ps_settings = (
            # Connection only, no database
            PersistentStoreConnectionSetting(
                name='primary',
                description='Connection with superuser role needed.',
                required=True
            ),
            # Connection only, no database
            PersistentStoreConnectionSetting(
                name='creator',
                description='Create database role only.',
                required=False
            ),
```

(continues on next page)

(continued from previous page)

```
            # Spatial database
            PersistentStoreDatabaseSetting(
                name='spatial_db',
                description='for storing important spatial stuff',
                required=True,
                initializer='appsettings.model.init_spatial_db',
                spatial=True,
            ),
            # Non-spatial database
            PersistentStoreDatabaseSetting(
                name='temp_db',
                description='for storing temporary stuff',
                required=False,
                initializer='appsettings.model.init_temp_db',
                spatial=False,
            )
        )

        return ps_settings
```

TethysAppBase.**dataset_service_settings**()

Override this method to define dataset service connections for use in your app.

Returns A list or tuple of DatasetServiceSetting objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import DatasetServiceSetting

class MyFirstApp(TethysAppBase):

    def dataset_service_settings(self):
        """
        Example dataset_service_settings method.
        """
        ds_settings = (
            DatasetServiceSetting(
                name='primary_ckan',
                description='Primary CKAN service for app to use.',
                engine=DatasetServiceSetting.CKAN,
                required=True,
            ),
            DatasetServiceSetting(
                name='hydroshare',
                description='HydroShare service for app to use.',
                engine=DatasetServiceSetting.HYDROSHARE,
                required=False
            )
        )

        return ds_settings
```

TethysAppBase.**spatial_dataset_service_settings**()

Override this method to define spatial dataset service connections for use in your app.

Returns A list or tuple of SpatialDatasetServiceSetting objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import SpatialDatasetServiceSetting

class MyFirstApp(TethysAppBase):

    def spatial_dataset_service_settings(self):
        """
        Example spatial_dataset_service_settings method.
        """
        sds_settings = (
            SpatialDatasetServiceSetting(
                name='primary_geoserver',
                description='spatial dataset service for app to use',
                engine=SpatialDatasetServiceSetting.GEOSERVER,
                required=True,
            ),
        )

        return sds_settings
```

TethysAppBase.**web_processing_service_settings**()

Override this method to define web processing service connections for use in your app.

Returns A list or tuple of WebProcessingServiceSetting objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import WebProcessingServiceSetting

class MyFirstApp(TethysAppBase):

    def web_processing_service_settings(self):
        """
        Example wps_services method.
        """
        wps_services = (
            WebProcessingServiceSetting(
                name='primary_52n',
                description='WPS service for app to use',
                required=True,
            ),
        )

        return wps_services
```

TethysAppBase.**handoff_handlers**()

Override this method to define handoff handlers for use in your app.

Returns A list or tuple of HandoffHandler objects.

Return type iterable

Example:

```
from tethys_sdk.handoff import HandoffHandler

class MyFirstApp(TethysAppBase):

    def handoff_handlers(self):
        """
        Example handoff_handlers method.
        """
```

(continues on next page)

(continued from previous page)

```
        handoff_handlers = (
            HandoffHandlers(
                name='example',
                handler='my_first_app.controllers.my_handler'
            ),
        )

        return handoff_handlers
```

TethysAppBase.**job_templates**()

Override this method to define job templates to easily create and submit jobs in your app.

Returns A list or tuple of `JobTemplate` objects.

Return type iterable

Example:

```
from tethys_sdk.jobs import CondorJobTemplate
from tethys_sdk.compute import list_schedulers

class MyFirstApp(TethysAppBase):

    def job_templates(cls):
        """
        Example job_templates method.
        """
        my_scheduler = list_schedulers()[0]

        job_templates = (CondorJobTemplate(name='example',
                                           parameters={'executable': '$(APP_
↪WORKSPACE)/example_exe.py',
                                                       'condorpy_template_name':
↪'vanilla_transfer_files',
                                                       'attributes': {'transfer_
↪input_files': ('../input_1.in', '../input_2.in'),
                                                                      'transfer_
↪output_files': ('example_output1.out', 'example_output2.out'),
                                                                     },
                                                       'scheduler': my_scheduler,
                                                       'remote_input_files': ('
↪$(APP_WORKSPACE)/example_exe.py', '$(APP_WORKSPACE)/input_1.in', '$(USER_
↪WORKSPACE)/input_2.in'),
                                                      }
                                          ),
                         )

        return job_templates
```

Class Methods

classmethod TethysAppBase.**get_custom_setting**(*name*)

Retrieves the value of a CustomSetting for the app.

Parameters **name** (*str*) -- The name of the CustomSetting as defined in the app.py.

Returns Value of the CustomSetting or None if no value assigned.

Return type variable

Example:

```
from my_first_app.app import MyFirstApp as app

max_count = app.get_custom_setting('max_count')
```

classmethod TethysAppBase.**get_persistent_store_connection**(*name*, *as_url=False*, *as_sessionmaker=False*)

Gets an SQLAlchemy Engine or URL object for the named persistent store connection.

Parameters

- **name** (*string*) -- Name of the PersistentStoreConnectionSetting as defined in app.py.
- **as_url** (*bool*) -- Return SQLAlchemy URL object instead of engine object if True. Defaults to False.
- **as_sessionmaker** (*bool*) -- Returns SessionMaker class bound to the engine if True. Defaults to False.

Returns An SQLAlchemy Engine or URL object for the persistent store requested.

Return type sqlalchemy.Engine or sqlalchemy.URL

Example:

```
from my_first_app.app import MyFirstApp as app

conn_engine = app.get_persistent_store_connection('primary')
conn_url = app.get_persistent_store_connection('primary', as_url=True)
SessionMaker = app.get_persistent_store_database('primary', as_sessionmaker=True)
session = SessionMaker()
```

classmethod TethysAppBase.**get_persistent_store_database**(*name*, *as_url=False*, *as_sessionmaker=False*)

Gets an SQLAlchemy Engine or URL object for the named persistent store database given.

Parameters

- **name** (*string*) -- Name of the PersistentStoreConnectionSetting as defined in app.py.
- **as_url** (*bool*) -- Return SQLAlchemy URL object instead of engine object if True. Defaults to False.
- **as_sessionmaker** (*bool*) -- Returns SessionMaker class bound to the engine if True. Defaults to False.

Returns An SQLAlchemy Engine or URL object for the persistent store requested.

Return type sqlalchemy.Engine or sqlalchemy.URL

Example:

```
from my_first_app.app import MyFirstApp as app

db_engine = app.get_persistent_store_database('example_db')
db_url = app.get_persistent_store_database('example_db', as_url=True)
SessionMaker = app.get_persistent_store_database('example_db', as_
↪sessionmaker=True)
session = SessionMaker()
```

classmethod TethysAppBase.**get_dataset_service**(*name*, *as_public_endpoint=False*, *as_endpoint=False*, *as_engine=False*)

Retrieves dataset service engine assigned to named DatasetServiceSetting for the app.

Parameters

- **name** (*str*) -- name fo the DatasetServiceSetting as defined in the app.py.
- **as_endpoint** (*bool*) -- Returns endpoint url string if True, Defaults to False.
- **as_public_endpoint** (*bool*) -- Returns public endpoint url string if True. Defaults to False.
- **as_engine** (*bool*) -- Returns tethys_dataset_services.engine of appropriate type if True. Defaults to False.

Returns DatasetService assigned to setting if no other options are specified.

Return type DatasetService

Example:

```
from my_first_app.app import MyFirstApp as app

ckan_engine = app.get_dataset_service('primary_ckan', as_engine=True)
```

classmethod TethysAppBase.**get_spatial_dataset_service**(*name*, *as_public_endpoint=False*, *as_endpoint=False*, *as_wms=False*, *as_wfs=False*, *as_engine=False*)

Retrieves spatial dataset service engine assigned to named SpatialDatasetServiceSetting for the app.

Parameters

- **name** (*str*) -- name fo the SpatialDatasetServiceSetting as defined in the app.py.
- **as_endpoint** (*bool*) -- Returns endpoint url string if True, Defaults to False.
- **as_public_endpoint** (*bool*) -- Returns public endpoint url string if True. Defaults to False.
- **as_wfs** (*bool*) -- Returns OGC-WFS enpdoint url for spatial dataset service if True. Defaults to False.
- **as_wms** (*bool*) -- Returns OGC-WMS enpdoint url for spatial dataset service if True. Defaults to False.
- **as_engine** (*bool*) -- Returns tethys_dataset_services.engine of appropriate type if True. Defaults to False.

Returns SpatialDatasetService assigned to setting if no other options are specified.

Return type SpatialDatasetService

Example:

```
from my_first_app.app import MyFirstApp as app

geoserver_engine = app.get_spatial_dataset_engine('primary_geoserver', as_
↪engine=True)
```

classmethod TethysAppBase.**get_web_processing_service**(*name*, *as_public_endpoint=False*, *as_endpoint=False*, *as_engine=False*)

Retrieves web processing service engine assigned to named WebProcessingServiceSetting for the app.

Parameters

- **name** (*str*) -- name fo the WebProcessingServiceSetting as defined in the app.py.
- **as_endpoint** (*bool*) -- Returns endpoint url string if True, Defaults to False.
- **as_public_endpoint** (*bool*) -- Returns public endpoint url string if True. Defaults to False.
- **as_engine** (*bool*) -- Returns owslib.wps.WebProcessingService engine if True.

Defaults to False.

Returns WpsService assigned to setting if no other options are specified.

Return type WpsService

Example:

```
from my_first_app.app import MyFirstApp as app

wps_engine = app.get_web_processing_service('primary_52n')
```

classmethod TethysAppBase.**get_handoff_manager**()

Get the HandoffManager for the app.

classmethod TethysAppBase.**get_job_manager**()

Get the JobManager for the app.

classmethod TethysAppBase.**get_app_workspace**()

Get the file workspace (directory) for the app.

Returns An object representing the workspace.

Return type *tethys_apps.base.TethysWorkspace*

Example:

```
import os
from my_first_app.app import MyFirstApp as app

def a_controller(request):
    """
    Example controller that uses get_app_workspace() method.
    """
    # Retrieve the workspace
    app_workspace = app.get_app_workspace()
    new_file_path = os.path.join(app_workspace.path, 'new_file.txt')

    with open(new_file_path, 'w') as a_file:
        a_file.write('...')

    context = {}

    return render(request, 'my_first_app/template.html', context)
```

classmethod TethysAppBase.**get_user_workspace**(*user*)

Get the file workspace (directory) for the given User.

Parameters **user** (*User or HttpRequest*) -- User or request object.

Returns An object representing the workspace.

Return type *tethys_apps.base.TethysWorkspace*

Example:

```
import os
from my_first_app.app import MyFirstApp as app

def a_controller(request):
    """
    Example controller that uses get_user_workspace() method.
    """
    # Retrieve the workspace
    user_workspace = app.get_user_workspace(request.user)
    new_file_path = os.path.join(user_workspace.path, 'new_file.txt')
```

(continues on next page)

(continued from previous page)

```
    with open(new_file_path, 'w') as a_file:
        a_file.write('...')

    context = {}

    return render(request, 'my_first_app/template.html', context)
```

classmethod TethysAppBase.**list_persistent_store_connections**()

Returns a list of existing persistent store connections for this app.

Returns A list of persistent store connection names.

Return type list

Example:

```
from my_first_app.app import MyFirstApp as app

ps_connections = app.list_persistent_store_connections()
```

classmethod TethysAppBase.**list_persistent_store_databases**(*dynamic_only=False*, *static_only=False*)

Returns a list of existing persistent store databases for the app.

Parameters

- **dynamic_only** (*bool*) -- only persistent store created dynamically if True. Defaults to False.
- **static_only** (*bool*) -- only static persistent stores if True. Defaults to False.

Returns A list of all persistent store database names for the app.

Return type list

Example:

```
from my_first_app.app import MyFirstApp as app

ps_databases = app.list_persistent_store_databases()
```

classmethod TethysAppBase.**persistent_store_exists**(*name*)

Returns True if a persistent store with the given name exists for the app.

Parameters **name** (*string*) -- Name of the persistent store database to check.

Returns True if persistent store exists.

Return type bool

Example:

```
from my_first_app.app import MyFirstApp as app

result = app.persistent_store_exists('example_db')

if result:
    engine = app.get_persistent_store_engine('example_db')
```

classmethod TethysAppBase.**create_persistent_store**(*db_name*, *connection_name*, *spatial=False*, *initializer=''*, *refresh=False*, *force_first_time=False*)

Creates a new persistent store database for the app. This method is idempotent.

Parameters

- **db_name** (*string*) -- Name of the persistent store that will be created.
- **connection_name** (*string|None*) -- Name of persistent store connection or None if creating a test copy of an existing persistent store (only while in the testing

environment)
- **spatial** (*bool*) -- Enable spatial extension on the database being created when True. Connection must have superuser role. Defaults to False.
- **initializer** (*string*) -- Dot-notation path to initializer function (e.g.: 'my_first_app.models.init_db').
- **refresh** (*bool*) -- Drop database if it exists and create again when True. Defaults to False.
- **force_first_time** (*bool*) -- Call initializer function with "first_time" parameter forced to True, even if this is not the first time intializing the persistent store database. Defaults to False.

Returns True if successful.

Return type bool

Example:

```
from my_first_app.app import MyFirstApp as app

result = app.create_persistent_store('example_db', 'primary')

if result:
    engine = app.get_persistent_store_engine('example_db')
```

classmethod TethysAppBase.**drop_persistent_store**(*name*)

Drop a persistent store database for the app. This method is idempotent.

Parameters **name** (*string*) -- Name of the persistent store to be dropped.

Returns True if successful.

Return type bool

Example:

```
from my_first_app.app import MyFirstApp as app

result = app.drop_persistent_store('example_db')

if result:
    # App database 'example_db' was successfully destroyed and no longer exists
    pass
```

1.6.3 App Templating API

Last Updated: May 2017

The pages of a Tethys app are created using the Django template language. This provides an overview of important Django templating concepts and introduces the base templates that are provided to make templating easier.

Django Templating Concepts

The Django template language allows you to create dynamic HTML templates and minmizes the amount of HTML you need to write for your app pages. This section will provide a crash course in Django template language basics, but we highly recommend a review of the Django Template Language documentation.

Tip: Review the Django Template Language to get a better grasp on templating in Tethys.

Variables

In Django templates, variables are denoted by double curly brace syntax: `{{ variable }}`. The variable expression will be replaced by the value of the variable. Dot notation can be used access attributes of a variable: `{{ variable.attribute }}`.

Examples:

```
# Examples of Django template variable syntax
{{ variable }}

# Access items in a list or tuple using dot notation
{{ list.0 }}

# Access items in a dictionary using dot notation
{{ dict.key }}

# Access attributes of objects using dot notation
{{ object.attribute }}
```

Hint: See Django template Variables documentation for more information.

Filters

Variables can be modified by filters which look like this: `{{ variable|filter:argument }}`. Filters perform modifying functions on variable output such as formatting dates, formatting numbers, changing the letter case, and concatenating multiple variables.

Examples:

```
# The default filter can be used to print a default value when the variable is falsy
{{ variable|default:"nothing" }}

# The join filter can be used to join a list with a the separator given
{{ list|join:", " }}
```

Hint: Refer to the Django Filter Reference for a full list of the filters available.

Tags

Tags use curly brace percent sign syntax like this: `{% tag %}`. Tags perform many different functions including creating text, controlling flow, or loading external information to be used in the app. Some commonly used tags include `for`, `if`, `block`, and `extends`.

Examples:

```
# The if tag only prints its contents when the condition evaluates to True
{% if name %}
    <h1>Hello, {{ name }}!</h1>
{% else %}
    <h1>Welcome!</h1>
```

(continues on next page)

(continued from previous page)

```
{% endif %}

#  The for tag can be used to loop through iterables printing its contents on each␣
↪iteration
<ul>
  {% for item in item_list %}
    <li>{{ item }}</li>
  {% endfor %}
</ul>

# The block tag is used to override the contents of the block of a parent template
{% block example %}
  <p>I just overrode the contents of the "example" block with this paragraph.</p>
{% endblock %}
```

Hint: See the Django Tag Reference for a complete list of tags that Django provides.

Template Inheritance

One of the advantages of using the Django template language is that it provides a method for child templates to extend parent templates, which can reduce the amount of HTML you need to write. Template inheritance is accomplished using two tags, `extends` and `block`. Parent templates provide `blocks` of content that can be overridden by child templates. Child templates can extend parent templates by using the `extends` tag. Calling the `block` tag of a parent template in a child template will override any content in that `block` tag with the content in the child template.

Hint: The Django Template Inheritance documentation provides an excellent example that illustrates how inheritance works.

Base Templates

There are two layers of templates provided for Tethys app development. The `app_base.html` template provides the HTML skeleton for all Tethys app templates, which includes the base HTML structural elements (e.g.: `<html>`, `<head>`, and `<body>` elements), the base style sheets and JavaScript libraries, and many blocks for customization. All Tethys app projects also include a `base.html` template that inherits from the `app_base.html` template.

App developers are encouraged to use the `base.html` file as the base template for all of their templates, rather than extending the `app_base.html` file directly. The `base.html` template is easier to work with, because it includes only the blocks that will be used most often from the `app_base.html` template. However, all of the blocks that are available from `app_base.html` template will also be available for use in the `base.html` template and any templates that extend it.

Many of the blocks in the template correspond with different portions of the app interface. Figure 1 provides a graphical explanation of these blocks. An explanation of all the blocks provided in the `app_base.html` and `base.html` templates can be found in the section that follows.

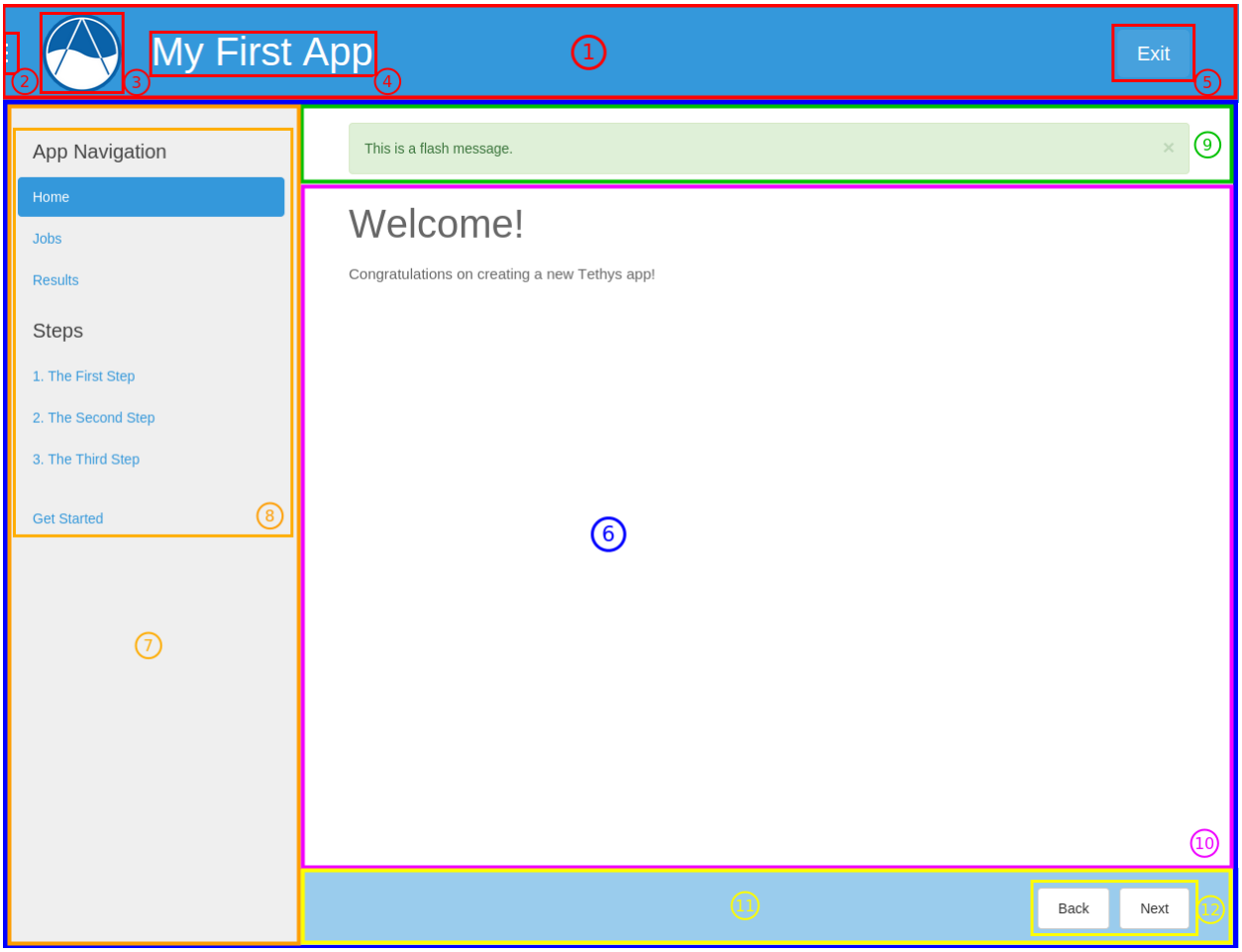


Fig. 2: **Figure 1.** Illustration of the blocks that correspond with app interface elements as follows:

1. app_header_override
2. app_navigation_toggle_override
3. app_icon_override, app_icon
4. app_title_override, app_title
5. exit_button_override
6. app_content_override
7. app_navigation_override
8. app_navigation, app_navigation_items
9. flash
10. app_content
11. app_actions_override
12. app_actions

Blocks

This section provides an explanation of the blocks are available for use in child templates of either the `app_base.html` or the `base.html` templates.

htmltag

Override the `<html>` element open tag.

Example:

```
{% block htmltag %}<html lang="es">{% endblock %}
```

headtag

Add attributes to the `<head>` element.

Example:

```
{% block headtag %}style="display: block;"{% endblock %}
```

meta

Override or append `<meta>` elements to the `<head>` element. To append to existing elements, use `block.super`.

Example:

```
{% block meta %}
  {{ block.super }}
  <meta name="description" value="My website description" />
{% endblock %}
```

title

Change title for the page. The title is used as metadata for the site and shows up in the browser in tabs and bookmark names.

Example:

```
{% block title %}{{ block.super }} - My Sub Title{% endblock %}
```

links

Add content before the stylesheets such as rss feeds and favicons. Use `block.super` to preserve the default favicon or override completely to specify custom favicon.

Example:

```
{% block links %}
  <link rel="shortcut icon" href="/path/to/favicon.ico" />
{% endblock %}
```

import_gizmos

The import_gizmos block allows you register gizmos to be added to your page so that the dependencies load properly.

Example:

```
{% block import_gizmos %}
  {% import_gizmo_dependency map_view %}
{% endblock %}
```

styles

Add additional stylesheets to the page. Use `block.super` to preserve the existing styles for the app (recommended) or override completely to use your own custom stylesheets.

Example:

```
{% block styles %}
  {{ block.super }}
  <link href="/path/to/styles.css" rel="stylesheet" />
{% endblock %}
```

global_scripts

Add JavaScript libraries that need to be loaded prior to the page being loaded. This is a good block to use for libraries that are referenced globally. The global libraries included as global scripts by default are JQuery and Bootstrap. Use `block.super` to preserve the default global libraries.

Example:

```
{% block global_scripts %}
  {{ block.super }}
  <script src="/path/to/script.js" type="text/javascript"></script>
{% endblock %}
```

bodytag

Add attributes to the `body` element.

Example:

```
{% block bodytag %}class="a-class" onload="run_this();"{% endblock %}
```

app_content_wrapper_override

Override the app content structure completely. The app content wrapper contains all content in the `<body>` element other than the scripts. Use this block to override all of the app template structure completely.

Override Eliminates:

app_header_override, app_navigation_toggle_override, app_icon_override, app_icon, app_title_override, app_title, exit_button_override, app_content_override, flash, app_navigation_override, app_navigation, app_navigation_items, app_content, app_actions_override, app_actions.

Example:

```
{% block app_content_wrapper_override %}
  <div>
    <p>My custom content</p>
  </div>
{% endblock %}
```

app_header_override

Override the app header completely including any wrapping elements. Useful for creating a custom header for your app.

Override Eliminates:

app_navigation_toggle_override, app_icon_override, app_icon, app_title_override, app_title, exit_button_override

app_navigation_toggle_override

Override the app navigation toggle button. This is useful if you want to create an app that does not include the navigation pane. Use this to remove the navigation toggle button as well.

Example:

```
{% block app_navigation_toggle_override %}{% endblock %}
```

app_icon_override

Override the app icon in the header completely including any wrapping elements.

Override Eliminates:

app_icon

app_icon

Override the app icon `<img>` element in the header.

Example:

```
{% block app_icon %}<img src="/path/to/icon.png">{% endblock %}
```

app_title_override

Override the app title in the header completely including any wrapping elements.

Override Eliminates:

app_title

app_title

Override the app title element in the header.

Example:

```
{% block app_title %}My App Title{% endblock %}
```

header_buttons_override

Override all the header buttons on the right-hand side of the header (settings button, exit button, and header buttons).

header_buttons

Use this block to add custom buttons to the app header. Use an anchor/link tag for the button and wrap it in a `div` with the class `header-button`. For buttons with the Bootstrap glyphicons, add the <`glyphicon-button` class to the wrapper element as well.

Example:

```
{% block header_buttons %}
  <div class="header-button glyphicon-button">
    <a href="{% url my_first_app:another_page %}"><span class="glyphicon glyphicon-
↪question-sign"></span></a>
  </div>
{% endblock %}
```

exit_button_override

Override the exit button completely including any wrapping elements.

app_content_override

Override only the app content area while preserving the header. The navigation and actions areas will also be overridden.

Override Eliminates:

flash, app_navigation_override, app_navigation, app_navigation_items, app_content, app_actions_override, app_actions

flash

Override the flash messaging capabilities. Flash messages are used to display dismissible messages to the user using the Django messaging capabilities. Override if you would like to implement your own messaging system or eliminate functionality all together.

app_navigation_override

Override the app navigation elements including any wrapping elements.

Override Eliminates:

app_navigation, app_navigation_items

app_navigation

Override the app navigation container. The default container for navigation is an unordered list. Use this block to override the unordered list for custom navigation.

Override Eliminates:

app_navigation_items

app_navigation_items

Override or append to the app navigation list. These should be `<li>` elements.

app_content

Add content to the app content area. This should be the primary block used to add content to the app.

Example:

```
{% block app_content %}
  <p>Content for my app.</p>
{% endblock %}
```

after_app_content

Use this block for adding elements after the app content such as Bootstrap modals (Bootstrap modals will not work properly if they are placed in the main `app_content` block).

Example:

```
{% block after_app_content %}
  {% gizmo my_modal %}
{% endblock %}
```

app_actions_override

Override app content elements including any wrapping elements.

app_actions

Override or append actions to the action area. These are typically buttons or links. The actions are floated right, so they need to be listed in right to left order.

Example:

```
{% block app_actions %}
  <a href="" class="btn btn-default">Next</a>
  <a href="" class="btn btn-default">Back</a>
{% endblock %}
```

scripts

Add additional JavaScripts to the page. Use `block.super` to preserve the existing scripts for the app (recommended) or override completely to use your own custom scripts.

Example:

```
{% block scripts %}
  {{ block.super }}
  <script href="/path/to/script.js" type="text/javascript"></script>
{% endblock %}
```

app_base.html

This section provides the complete contents of the `app_base.html` template. It is meant to be used as a reference for app developers, so they can be aware of the HTML structure underlying their app templates.

```
{% load staticfiles tethys_gizmos %}
<!DOCTYPE html>

{% block htmltag %}
<!--[if IE 7]> <html lang="en" class="ie ie7"> <![endif]-->
<!--[if IE 8]> <html lang="en"  class="ie ie8"> <![endif]-->
<!--[if IE 9]> <html lang="en"  class="ie9"> <![endif]-->
```

(continues on next page)

(continued from previous page)

```
<!--[if gt IE 8]><!--> <html lang="en" > <!--<![endif]-->
{% endblock %}

  <head {% block headtag %}{% endblock %}>

    {% block meta %}
      <meta charset="utf-8" />
      <meta http-equiv="X-UA-Compatible" content="IE=edge">
      <meta name="viewport" content="width=device-width, initial-scale=1">
      <meta name="generator" content="Django" />
    {% endblock %}

    <title>
      {% if site_globals.site_title %}
         {{ site_globals.site_title }}
      {% elif site_globals.brand_text %}
        {{ site_globals.brand_text }}
      {% else %}
        Tethys
      {% endif %}
      {% block title %}{% endblock %}
    </title>

    {% block links %}
      {% if site_globals.favicon %}
        <link rel="shortcut icon" href="{{ site_globals.favicon }}" />
      {% endif %}
    {% endblock %}

    {% block styles %}
      <link href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css"␣
↪rel="stylesheet" />
      <link href="{% static 'tethys_apps/css/app_base.css' %}" rel="stylesheet" />
    {% endblock %}

    {% block global_scripts %}
      <script src="//code.jquery.com/jquery-2.1.1.min.js" type="text/javascript"></
↪script>
      <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"␣
↪type="text/javascript"></script>
    {% endblock %}

  </head>

  <body {% block bodytag %}{% endblock %}>

    {% block app_content_wrapper_override %}
      <div id="app-content-wrapper" class="show-nav">

        {% block app_header_override %}
          <div id="app-header" class="clearfix">
            <div class="tethys-app-header" style="background: {{ tethys_app.
↪color|default:'#1b95dc' }};">

              {% block app-navigation-toggle-override %}
                <a href="javascript:void(0);" class="toggle-nav">
                  <div></div>
```

(continues on next page)

(continued from previous page)

```
                <div></div>
                <div></div>
              </a>
            {% endblock %}

            {% block app_icon_override %}
              <div class="icon-wrapper">
                {% block app_icon %}<img src="{% static tethys_app.icon %}">{%␣
↪endblock %}
              </div>
            {% endblock %}

            {% block app_title_override %}
              <div class="app-title-wrapper">
                <span class="app-title">{% block app_title %}{{ tethys_app.name }}{
↪% endblock %}</span>
              </div>
            {% endblock %}

            {% block exit_button_override %}
              <div class="exit-button">
                <a href="javascript:void(0);" onclick="TETHYS_APP_BASE.exit_app('{%␣
↪url 'app_library' %}');">Exit</a>
              </div>
            {% endblock %}
          </div>
        </div>
      {% endblock %}

      {% block app_content_override %}
        <div id="app-content">

          {% block flash %}
            {% if messages %}
              <div class="flash-messages">

                {% for message in messages %}
                  <div class="alert {% if message.tags %}{{ message.tags }}{% endif
↪%} alert-dismissible" role="alert">
                    <button type="button" class="close" data-dismiss="alert">
                      <span aria-hidden="true">&times;</span>
                      <span class="sr-only">Close</span>
                    </button>
                    {{ message }}
                  </div>
                {% endfor %}
              </div>
            {% endif %}
          {% endblock %}

          {% block app_navigation_override %}
            <div id="app-navigation">
              {% block app_navigation %}
                <ul class="nav nav-pills nav-stacked">
                  {% block app_navigation_items %}{% endblock %}
                </ul>
              {% endblock %}
```

(continues on next page)

(continued from previous page)

```
            </div>
          {% endblock %}

          <div id="inner-app-content">
            {% block app_content %}{% endblock %}

            {# App actions are fixed to the bottom #}
            {% block app_actions_override %}
              <div id="app-actions">
                {% block app_actions %}{% endblock %}
              </div>
            {% endblock %}
          </div>
        </div>
      {% endblock %}
    </div>
  {% endblock %}

  {% block scripts %}
    <script src="{% static 'tethys_apps/vendor/cookies.js' %}" type="text/javascript
↪"></script>
    <script src="{% static 'tethys_apps/js/app_base.js' %}" type="text/javascript">
↪</script>
    {% gizmo_dependencies %}
  {% endblock %}
 </body>
</html>
```

base.html

The `base.html` is the base template that is used directly by app templates. This file is generated in all new Tethys app projects that are created using the scaffold. The contents are provided here for reference.

All of the blocks provided by the `base.html` template are inherited from the `app_base.html` template. The `base.html` template is intended to be a simplified version of the `app_base.html` template, providing only the the blocks that should be used in a default app configuration. However, the blocks that are excluded from the `base.html` template can be used by advanced Tethys app developers who wish customize parts or all of the app template structure.

See the *Blocks* section for an explanation of each block.

```
{% extends "tethys_apps/app_base.html" %}

{% load staticfiles %}

{% block title %}- {{ tethys_app.name }}{% endblock %}

{% block styles %}
  {{ block.super }}
  <link href="{% static 'new_template_app/css/main.css' %}" rel="stylesheet"/>
{% endblock %}

{% block app_icon %}
  {# The path you provided in your app.py is accessible through the tethys_app.icon␣
↪context variable #}
```

(continues on next page)

(continued from previous page)

```
  <img src="{% static tethys_app.icon %}">
{% endblock %}

{# The name you provided in your app.py is accessible through the tethys_app.name␣
↪context variable #}
{% block app_title %}{{ tethys_app.name }}{% endblock %}

{% block app_navigation_items %}
  <li class="title">App Navigation</li>
  <li class="active"><a href="">Home</a></li>
  <li><a href="">Jobs</a></li>
  <li><a href="">Results</a></li>
  <li class="title">Steps</li>
  <li><a href="">1. The First Step</a></li>
  <li><a href="">2. The Second Step</a></li>
  <li><a href="">3. The Third Step</a></li>
  <li class="separator"></li>
  <li><a href="">Get Started</a></li>
{% endblock %}

{% block app_content %}
{% endblock %}

{% block app_actions %}
{% endblock %}

{% block scripts %}
  {{ block.super }}
  <script src="{% static 'new_template_app/js/main.js' %}" type="text/javascript"></
↪script>
{% endblock %}
```

1.6.4 App Settings API

Last Updated: May 2017

The App Settings API allows developers to create settings for their apps that can be configured in the admin interface of the Tethys Portal in which the app is installed. Examples of what App Settings could be used for include enabling or disabling functionality, assigning constant values or assumptions that are used throughout the app, or customizing the look and feel of the app. App Settings are only be accessible by Tethys Portal administrators in production, so they should be thought of as global settings for the app that are not customizable on a user by user basis.

As of Tethys Platform 2.0.0, Tethys Services such as databases and map servers are configured through App Settings. Tethys Service App Settings can be thought of as sockets for a particular type of Tethys Service (e.g. PostGIS database, GeoServer, CKAN). Tethys Portal administrators can "plug-in" the appropriate type of Tethys Service from the pool of Tethys Services during installation and setup of the app in the portal. This slight paradigm shift gives Tethys Portal administrators more control over the resources they manage for a Portal instance and how they are distributed across the apps.

Custom Settings

Custom Settings are used to create scalar-valued settings for an app. Four basic types of values are supported including `string`, `boolean`, `integer`, and `float`. Create Custom Settings by implementing the `custom_setting()` method in your *app class*. This method should return a list of *CustomSettings* objects:

`TethysAppBase.`**`custom_settings`**`()`

Override this method to define custom settings for use in your app.

Returns A list or tuple of `CustomSetting` objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import CustomSetting

class MyFirstApp(TethysAppBase):

    def custom_settings(self):
        """
        Example custom_settings method.
        """
        custom_settings = (
            CustomSetting(
                name='default_name',
                type=CustomSetting.TYPE_STRING
                description='Default model name.',
                required=True
            ),
            CustomSetting(
                name='max_count',
                type=CustomSetting.TYPE_INTEGER,
                description='Maximum allowed count in a method.',
                required=False
            ),
            CustomSetting(
                name='change_factor',
                type=CustomSetting.TYPE_FLOAT,
                description='Change factor that is applied to some process.',
                required=True
            ),
            CustomSetting(
                name='enable_feature',
                type=CustomSetting.TYPE_BOOLEAN,
                description='Enable this feature when True.',
                required=True
            )
        )

        return custom_settings
```

To retrieve the value of a Custom Setting, import your *app class* and call the `get_custom_setting()` class method:

`classmethod` `TethysAppBase.`**`get_custom_setting`**`(`*`name`*`)`

Retrieves the value of a CustomSetting for the app.

Parameters **`name`** (*`str`*) -- The name of the CustomSetting as defined in the app.py.

Returns Value of the CustomSetting or None if no value assigned.

Return type variable

Example:

```
from my_first_app.app import MyFirstApp as app

max_count = app.get_custom_setting('max_count')
```

Persistent Store Settings

Persistent Store Settings are used to request databases and connections to database servers for use in your app (e.g. PostgreSQL, PostGIS). Create Persistent Store Settings by implementing the `persistent_store_settings()` method in your *app class*. This method should return a list of *PersistentStoreConnectionSetting* and *PersistentStoreDatabaseSetting* objects:

`TethysAppBase.`**`persistent_store_settings`**`()`

Override this method to define a persistent store service connections and databases for your app.

Returns A list or tuple of `PersistentStoreDatabaseSetting` or `PersistentStoreConnectionSetting` objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import PersistentStoreDatabaseSetting,␣
↪PersistentStoreConnectionSetting

class MyFirstApp(TethysAppBase):

    def persistent_store_settings(self):
        """
        Example persistent_store_settings method.
        """

        ps_settings = (
            # Connection only, no database
            PersistentStoreConnectionSetting(
                name='primary',
                description='Connection with superuser role needed.',
                required=True
            ),
            # Connection only, no database
            PersistentStoreConnectionSetting(
                name='creator',
                description='Create database role only.',
                required=False
            ),
            # Spatial database
            PersistentStoreDatabaseSetting(
                name='spatial_db',
                description='for storing important spatial stuff',
                required=True,
                initializer='appsettings.model.init_spatial_db',
                spatial=True,
            ),
            # Non-spatial database
            PersistentStoreDatabaseSetting(
                name='temp_db',
                description='for storing temporary stuff',
                required=False,
                initializer='appsettings.model.init_temp_db',
```

(continues on next page)

(continued from previous page)

```
                spatial=False,
            )
        )

        return ps_settings
```

To retrieve a connection to a Persistent Store, import your *app class* and call either the `get_persistent_store_database()` or `get_persistent_store_connection` class methods:

classmethod TethysAppBase.**get_persistent_store_database**(*name*, *as_url=False*, *as_sessionmaker=False*)

Gets an SQLAlchemy Engine or URL object for the named persistent store database given.

Parameters

- **name** (*string*) -- Name of the PersistentStoreConnectionSetting as defined in app.py.
- **as_url** (*bool*) -- Return SQLAlchemy URL object instead of engine object if True. Defaults to False.
- **as_sessionmaker** (*bool*) -- Returns SessionMaker class bound to the engine if True. Defaults to False.

Returns An SQLAlchemy Engine or URL object for the persistent store requested.

Return type sqlalchemy.Engine or sqlalchemy.URL

Example:

```
from my_first_app.app import MyFirstApp as app

db_engine = app.get_persistent_store_database('example_db')
db_url = app.get_persistent_store_database('example_db', as_url=True)
SessionMaker = app.get_persistent_store_database('example_db', as_
↪sessionmaker=True)
session = SessionMaker()
```

classmethod TethysAppBase.**get_persistent_store_connection**(*name*, *as_url=False*, *as_sessionmaker=False*)

Gets an SQLAlchemy Engine or URL object for the named persistent store connection.

Parameters

- **name** (*string*) -- Name of the PersistentStoreConnectionSetting as defined in app.py.
- **as_url** (*bool*) -- Return SQLAlchemy URL object instead of engine object if True. Defaults to False.
- **as_sessionmaker** (*bool*) -- Returns SessionMaker class bound to the engine if True. Defaults to False.

Returns An SQLAlchemy Engine or URL object for the persistent store requested.

Return type sqlalchemy.Engine or sqlalchemy.URL

Example:

```
from my_first_app.app import MyFirstApp as app

conn_engine = app.get_persistent_store_connection('primary')
conn_url = app.get_persistent_store_connection('primary', as_url=True)
SessionMaker = app.get_persistent_store_database('primary', as_sessionmaker=True)
session = SessionMaker()
```

Tip: See the *Persistent Stores API* and the *Spatial Persistent Stores API* for more details on how to use Persistent Stores in your apps.

Dataset Service Settings

Dataset Service Settings are used to request specific types of dataset services for use in your app (e.g. CKAN, HydroShare). Create Dataset Service Settings by implementing the `dataset_service_settings()` method in your *app class*. This method should return a list of *DatasetServiceSetting* objects:

`TethysAppBase.`**`dataset_service_settings`**`()`

Override this method to define dataset service connections for use in your app.

Returns A list or tuple of `DatasetServiceSetting` objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import DatasetServiceSetting

class MyFirstApp(TethysAppBase):

    def dataset_service_settings(self):
        """
        Example dataset_service_settings method.
        """
        ds_settings = (
            DatasetServiceSetting(
                name='primary_ckan',
                description='Primary CKAN service for app to use.',
                engine=DatasetServiceSetting.CKAN,
                required=True,
            ),
            DatasetServiceSetting(
                name='hydroshare',
                description='HydroShare service for app to use.',
                engine=DatasetServiceSetting.HYDROSHARE,
                required=False
            )
        )

        return ds_settings
```

To retrieve a connection to a Dataset Service, import your *app class* and call the `get_dataset_service()` class method:

`classmethod` `TethysAppBase.`**`get_dataset_service`**`(`*name*, *as_public_endpoint=False*, *as_endpoint=False*, *as_engine=False*`)`

Retrieves dataset service engine assigned to named DatasetServiceSetting for the app.

Parameters

- **`name`** (*`str`*) -- name fo the DatasetServiceSetting as defined in the app.py.
- **`as_endpoint`** (*`bool`*) -- Returns endpoint url string if True, Defaults to False.
- **`as_public_endpoint`** (*`bool`*) -- Returns public endpoint url string if True. Defaults to False.
- **`as_engine`** (*`bool`*) -- Returns tethys_dataset_services.engine of appropriate type if True. Defaults to False.

Returns DatasetService assigned to setting if no other options are specified.

Return type DatasetService

Example:

```
from my_first_app.app import MyFirstApp as app

ckan_engine = app.get_dataset_service('primary_ckan', as_engine=True)
```

Tip: See the *Dataset Services API* for more details on how to use Dataset Services in your apps.

Spatial Dataset Service Settings

Spatial Dataset Service Settings are used to request specific types of spatial dataset services for use in your app (e.g. geoserver). Create Spatial Dataset Service Settings by implementing the `spatial_dataset_service_settings()` method in your *app class*. This method should return a list of *SpatialDatasetServiceSetting* objects:

`TethysAppBase.`**`spatial_dataset_service_settings`**`()`

Override this method to define spatial dataset service connections for use in your app.

Returns A list or tuple of `SpatialDatasetServiceSetting` objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import SpatialDatasetServiceSetting

class MyFirstApp(TethysAppBase):

    def spatial_dataset_service_settings(self):
        """
        Example spatial_dataset_service_settings method.
        """
        sds_settings = (
            SpatialDatasetServiceSetting(
                name='primary_geoserver',
                description='spatial dataset service for app to use',
                engine=SpatialDatasetServiceSetting.GEOSERVER,
                required=True,
            ),
        )

        return sds_settings
```

To retrieve a connection to a Spatial Dataset Service, import your *app class* and call the `get_spatial_dataset_service()` class method:

`classmethod` `TethysAppBase.`**`get_spatial_dataset_service`**`(`*name*, *as_public_endpoint=False*, *as_endpoint=False*, *as_wms=False*, *as_wfs=False*, *as_engine=False*`)`

Retrieves spatial dataset service engine assigned to named SpatialDatasetServiceSetting for the app.

Parameters

- **`name`** (*`str`*) -- name fo the SpatialDatasetServiceSetting as defined in the app.py.
- **`as_endpoint`** (*`bool`*) -- Returns endpoint url string if True, Defaults to False.
- **`as_public_endpoint`** (*`bool`*) -- Returns public endpoint url string if True. Defaults to False.
- **`as_wfs`** (*`bool`*) -- Returns OGC-WFS enpdoint url for spatial dataset service if True. Defaults to False.
- **`as_wms`** (*`bool`*) -- Returns OGC-WMS enpdoint url for spatial dataset service if True. Defaults to False.

- **as_engine** (*bool*) -- Returns tethys_dataset_services.engine of appropriate type if True. Defaults to False.

Returns SpatialDatasetService assigned to setting if no other options are specified.

Return type SpatialDatasetService

Example:

```
from my_first_app.app import MyFirstApp as app

geoserver_engine = app.get_spatial_dataset_engine('primary_geoserver', as_
↪engine=True)
```

Tip: See the *Spatial Dataset Services API* for more details on how to use Spatial Dataset Services in your apps.

Web Processing Service Settings

Web Processing Service Settings are used to request specific types of dataset services for use in your app (e.g. CKAN, HydroShare). Create Web Processing Service Settings by implementing the `web_processing_service_settings()` method in your *app class*. This method should return a list of *WebProcessingServiceSetting* objects:

`TethysAppBase.`**`web_processing_service_settings`**`()`

Override this method to define web processing service connections for use in your app.

Returns A list or tuple of `WebProcessingServiceSetting` objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import WebProcessingServiceSetting

class MyFirstApp(TethysAppBase):

    def web_processing_service_settings(self):
        """
        Example wps_services method.
        """
        wps_services = (
            WebProcessingServiceSetting(
                name='primary_52n',
                description='WPS service for app to use',
                required=True,
            ),
        )

        return wps_services
```

To retrieve a connection to a Web Processing Service, import your *app class* and call the `get_web_processing_service()` class method:

classmethod `TethysAppBase.`**`get_web_processing_service`**`(`*name*, *as_public_endpoint=False*, *as_endpoint=False*, *as_engine=False*`)`

Retrieves web processing service engine assigned to named WebProcessingServiceSetting for the app.

Parameters

- **name** (*str*) -- name fo the WebProcessingServiceSetting as defined in the app.py.
- **as_endpoint** (*bool*) -- Returns endpoint url string if True, Defaults to False.

- **as_public_endpoint** (*bool*) -- Returns public endpoint url string if True. Defaults to False.
- **as_engine** (*bool*) -- Returns owslib.wps.WebProcessingService engine if True. Defaults to False.

Returns WpsService assigned to setting if no other options are specified.

Return type WpsService

Example:

```
from my_first_app.app import MyFirstApp as app

wps_engine = app.get_web_processing_service('primary_52n')
```

Tip: See the *Web Processing Services API* for more details on how to use Dataset Services in your apps.

API Documentation

Settings Objects

class tethys_sdk.app_settings.**CustomSetting**(**args*, ***kwargs*)

Used to define a Custom Setting.

name

Unique name used to identify the setting.

Type str

type

The type of the custom setting. Either CustomSetting.TYPE_STRING, CustomSetting.TYPE_INTEGER, CustomSetting.TYPE_FLOAT, CustomSetting.TYPE_BOOLEAN

Type enum

description

Short description of the setting.

Type str

required

A value will be required if True.

Type bool

Example:

```
from tethys_sdk.app_settings import CustomSetting

default_name_setting = CustomSetting(
    name='default_name',
    type=CustomSetting.TYPE_STRING
    description='Default model name.',
    required=True
)

max_count_setting = CustomSetting(
    name='max_count',
    type=CustomSetting.TYPE_INTEGER,
    description='Maximum allowed count in a method.',
    required=False
```

(continues on next page)

(continued from previous page)

```
)

change_factor_setting = CustomSetting(
    name='change_factor',
    type=CustomSetting.TYPE_FLOAT,
    description='Change factor that is applied to some process.',
    required=True
)

enable_feature_setting = CustomSetting(
    name='enable_feature',
    type=CustomSetting.TYPE_BOOLEAN,
    description='Enable this feature when True.',
    required=True
)
```

class tethys_sdk.app_settings.**PersistentStoreConnectionSetting**(**args*, ***kwargs*)

Used to define a Peristent Store Connection Setting.

name
Unique name used to identify the setting.
Type str

description
Short description of the setting.
Type str

required
A value will be required if True.
Type bool

Example:

```
from tethys_sdk.app_settings import PersistentStoreConnectionSetting

primary_db_conn_setting = PersistentStoreConnectionSetting(
    name='primary',
    description='Connection with superuser role needed.',
    required=True
)
```

class tethys_sdk.app_settings.**PersistentStoreDatabaseSetting**(**args*, ***kwargs*)

Used to define a Peristent Store Database Setting.

name
Unique name used to identify the setting.
Type str

description
Short description of the setting.
Type str

initializer
Dot-notation path to function used to initialize the database.
Type str

spatial
Enable the PostGIS extension on the database during creation when True.

Type bool

required
A value will be required if True.
Type bool

Example:

```
from tethys_sdk.app_settings import PersistentStoreDatabaseSetting

spatial_db_setting = PersistentStoreDatabaseSetting(
    name='spatial_db',
    description='for storing important spatial stuff',
    required=True,
    initializer='appsettings.init_stores.init_spatial_db',
    spatial=True,
),

temp_db_setting = PersistentStoreDatabaseSetting(
    name='temp_db',
    description='for storing temporary stuff',
    required=False,
    initializer='appsettings.init_stores.init_temp_db',
    spatial=False,
)
```

class tethys_sdk.app_settings.**DatasetServiceSetting**(**args*, ***kwargs*)
Used to define a Dataset Service Setting.

name
Unique name used to identify the setting.
Type str

description
Short description of the setting.
Type str

engine
Either DatasetServiceSetting.CKAN or DatasetServiceSetting.HYDROSHARE
Type enum

required
A value will be required if True.
Type bool

Example:

```
from tethys_sdk.app_settings import DatasetServiceSetting

primary_ckan_setting = DatasetServiceSetting(
    name='primary_ckan',
    description='Primary CKAN service for app to use.',
    engine=DatasetServiceSetting.CKAN,
    required=True,
)

hydroshare_setting = DatasetServiceSetting(
    name='hydroshare',
    description='HydroShare service for app to use.',
```

(continues on next page)

(continued from previous page)

```
    engine=DatasetServiceSetting.HYDROSHARE,
    required=False
)
```

class tethys_sdk.app_settings.**SpatialDatasetServiceSetting**(**args*, ***kwargs*)

Used to define a Spatial Dataset Service Setting.

name

Unique name used to identify the setting.

Type str

description

Short description of the setting.

Type str

engine

Only SpatialDatasetServiceSetting.GEOSERVER at this time.

Type enum

required

A value will be required if True.

Type bool

Example:

```
from tethys_sdk.app_settings import SpatialDatasetServiceSetting

primary_geoserver_setting = SpatialDatasetServiceSetting(
    name='primary_geoserver',
    description='spatial dataset service for app to use',
    engine=SpatialDatasetServiceSetting.GEOSERVER,
    required=True,
)
```

class tethys_sdk.app_settings.**WebProcessingServiceSetting**(**args*, ***kwargs*)

Used to define a Web Processing Service Setting.

name

Unique name used to identify the setting.

Type str

description

Short description of the setting.

Type str

required

A value will be required if True.

Type bool

Example:

```
from tethys_sdk.app_settings import WebProcessingServiceSetting

primary_52n_setting = WebProcessingServiceSetting(
    name='primary_52n',
    description='WPS service for app to use',
    required=True,
)
```

Settings Declaration Methods

TethysAppBase.**custom_settings**()

Override this method to define custom settings for use in your app.

Returns A list or tuple of CustomSetting objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import CustomSetting

class MyFirstApp(TethysAppBase):

    def custom_settings(self):
        """
        Example custom_settings method.
        """
        custom_settings = (
            CustomSetting(
                name='default_name',
                type=CustomSetting.TYPE_STRING
                description='Default model name.',
                required=True
            ),
            CustomSetting(
                name='max_count',
                type=CustomSetting.TYPE_INTEGER,
                description='Maximum allowed count in a method.',
                required=False
            ),
            CustomSetting(
                name='change_factor',
                type=CustomSetting.TYPE_FLOAT,
                description='Change factor that is applied to some process.',
                required=True
            ),
            CustomSetting(
                name='enable_feature',
                type=CustomSetting.TYPE_BOOLEAN,
                description='Enable this feature when True.',
                required=True
            )
        )

        return custom_settings
```

TethysAppBase.**persistent_store_settings**()

Override this method to define a persistent store service connections and databases for your app.

Returns A list or tuple of PersistentStoreDatabaseSetting or PersistentStoreConnectionSetting objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import PersistentStoreDatabaseSetting,␣
↪PersistentStoreConnectionSetting

class MyFirstApp(TethysAppBase):
```

(continues on next page)

(continued from previous page)

```
    def persistent_store_settings(self):
        """
        Example persistent_store_settings method.
        """

        ps_settings = (
            # Connection only, no database
            PersistentStoreConnectionSetting(
                name='primary',
                description='Connection with superuser role needed.',
                required=True
            ),
            # Connection only, no database
            PersistentStoreConnectionSetting(
                name='creator',
                description='Create database role only.',
                required=False
            ),
            # Spatial database
            PersistentStoreDatabaseSetting(
                name='spatial_db',
                description='for storing important spatial stuff',
                required=True,
                initializer='appsettings.model.init_spatial_db',
                spatial=True,
            ),
            # Non-spatial database
            PersistentStoreDatabaseSetting(
                name='temp_db',
                description='for storing temporary stuff',
                required=False,
                initializer='appsettings.model.init_temp_db',
                spatial=False,
            )
        )

        return ps_settings
```

TethysAppBase.**dataset_service_settings**()

Override this method to define dataset service connections for use in your app.

Returns A list or tuple of DatasetServiceSetting objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import DatasetServiceSetting

class MyFirstApp(TethysAppBase):

    def dataset_service_settings(self):
        """
        Example dataset_service_settings method.
        """
        ds_settings = (
            DatasetServiceSetting(
                name='primary_ckan',
                description='Primary CKAN service for app to use.',
```

(continues on next page)

(continued from previous page)

```
                engine=DatasetServiceSetting.CKAN,
                required=True,
            ),
            DatasetServiceSetting(
                name='hydroshare',
                description='HydroShare service for app to use.',
                engine=DatasetServiceSetting.HYDROSHARE,
                required=False
            )
        )

        return ds_settings
```

TethysAppBase.**spatial_dataset_service_settings**()

Override this method to define spatial dataset service connections for use in your app.

Returns A list or tuple of SpatialDatasetServiceSetting objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import SpatialDatasetServiceSetting

class MyFirstApp(TethysAppBase):

    def spatial_dataset_service_settings(self):
        """
        Example spatial_dataset_service_settings method.
        """
        sds_settings = (
            SpatialDatasetServiceSetting(
                name='primary_geoserver',
                description='spatial dataset service for app to use',
                engine=SpatialDatasetServiceSetting.GEOSERVER,
                required=True,
            ),
        )

        return sds_settings
```

TethysAppBase.**web_processing_service_settings**()

Override this method to define web processing service connections for use in your app.

Returns A list or tuple of WebProcessingServiceSetting objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import WebProcessingServiceSetting

class MyFirstApp(TethysAppBase):

    def web_processing_service_settings(self):
        """
        Example wps_services method.
        """
        wps_services = (
            WebProcessingServiceSetting(
                name='primary_52n',
                description='WPS service for app to use',
```

(continues on next page)

(continued from previous page)

```
                required=True,
            ),
        )

        return wps_services
```

Settings Getter Methods

classmethod TethysAppBase.**get_custom_setting**(*name*)

Retrieves the value of a CustomSetting for the app.

Parameters **name** (*str*) -- The name of the CustomSetting as defined in the app.py.

Returns Value of the CustomSetting or None if no value assigned.

Return type variable

Example:

```
from my_first_app.app import MyFirstApp as app

max_count = app.get_custom_setting('max_count')
```

classmethod TethysAppBase.**get_persistent_store_database**(*name*, *as_url=False*, *as_sessionmaker=False*)

Gets an SQLAlchemy Engine or URL object for the named persistent store database given.

Parameters

- **name** (*string*) -- Name of the PersistentStoreConnectionSetting as defined in app.py.
- **as_url** (*bool*) -- Return SQLAlchemy URL object instead of engine object if True. Defaults to False.
- **as_sessionmaker** (*bool*) -- Returns SessionMaker class bound to the engine if True. Defaults to False.

Returns An SQLAlchemy Engine or URL object for the persistent store requested.

Return type sqlalchemy.Engine or sqlalchemy.URL

Example:

```
from my_first_app.app import MyFirstApp as app

db_engine = app.get_persistent_store_database('example_db')
db_url = app.get_persistent_store_database('example_db', as_url=True)
SessionMaker = app.get_persistent_store_database('example_db', as_
↪sessionmaker=True)
session = SessionMaker()
```

classmethod TethysAppBase.**get_persistent_store_connection**(*name*, *as_url=False*, *as_sessionmaker=False*)

Gets an SQLAlchemy Engine or URL object for the named persistent store connection.

Parameters

- **name** (*string*) -- Name of the PersistentStoreConnectionSetting as defined in app.py.
- **as_url** (*bool*) -- Return SQLAlchemy URL object instead of engine object if True. Defaults to False.
- **as_sessionmaker** (*bool*) -- Returns SessionMaker class bound to the engine if True. Defaults to False.

Returns An SQLAlchemy Engine or URL object for the persistent store requested.

Return type sqlalchemy.Engine or sqlalchemy.URL

Example:

```
from my_first_app.app import MyFirstApp as app

conn_engine = app.get_persistent_store_connection('primary')
conn_url = app.get_persistent_store_connection('primary', as_url=True)
SessionMaker = app.get_persistent_store_database('primary', as_sessionmaker=True)
session = SessionMaker()
```

classmethod TethysAppBase.**get_dataset_service**(*name*, *as_public_endpoint=False*, *as_endpoint=False*, *as_engine=False*)

Retrieves dataset service engine assigned to named DatasetServiceSetting for the app.

Parameters

- **name** (*str*) -- name fo the DatasetServiceSetting as defined in the app.py.
- **as_endpoint** (*bool*) -- Returns endpoint url string if True, Defaults to False.
- **as_public_endpoint** (*bool*) -- Returns public endpoint url string if True. Defaults to False.
- **as_engine** (*bool*) -- Returns tethys_dataset_services.engine of appropriate type if True. Defaults to False.

Returns DatasetService assigned to setting if no other options are specified.

Return type DatasetService

Example:

```
from my_first_app.app import MyFirstApp as app

ckan_engine = app.get_dataset_service('primary_ckan', as_engine=True)
```

classmethod TethysAppBase.**get_spatial_dataset_service**(*name*, *as_public_endpoint=False*, *as_endpoint=False*, *as_wms=False*, *as_wfs=False*, *as_engine=False*)

Retrieves spatial dataset service engine assigned to named SpatialDatasetServiceSetting for the app.

Parameters

- **name** (*str*) -- name fo the SpatialDatasetServiceSetting as defined in the app.py.
- **as_endpoint** (*bool*) -- Returns endpoint url string if True, Defaults to False.
- **as_public_endpoint** (*bool*) -- Returns public endpoint url string if True. Defaults to False.
- **as_wfs** (*bool*) -- Returns OGC-WFS enpdoint url for spatial dataset service if True. Defaults to False.
- **as_wms** (*bool*) -- Returns OGC-WMS enpdoint url for spatial dataset service if True. Defaults to False.
- **as_engine** (*bool*) -- Returns tethys_dataset_services.engine of appropriate type if True. Defaults to False.

Returns SpatialDatasetService assigned to setting if no other options are specified.

Return type SpatialDatasetService

Example:

```
from my_first_app.app import MyFirstApp as app

geoserver_engine = app.get_spatial_dataset_engine('primary_geoserver', as_
↪engine=True)
```

classmethod TethysAppBase.**get_web_processing_service**(*name*, *as_public_endpoint=False*, *as_endpoint=False*, *as_engine=False*)

Retrieves web processing service engine assigned to named WebProcessingServiceSetting for the app.

Parameters

- **name** (*str*) -- name fo the WebProcessingServiceSetting as defined in the app.py.
- **as_endpoint** (*bool*) -- Returns endpoint url string if True, Defaults to False.
- **as_public_endpoint** (*bool*) -- Returns public endpoint url string if True. Defaults to False.
- **as_engine** (*bool*) -- Returns owslib.wps.WebProcessingService engine if True. Defaults to False.

Returns WpsService assigned to setting if no other options are specified.

Return type WpsService

Example:

```
from my_first_app.app import MyFirstApp as app

wps_engine = app.get_web_processing_service('primary_52n')
```

1.6.5 Compute API

Last Updated: February 11, 2015

Distributed computing in Tethys Platform is made possible with HTCondor. Portal wide HTCondor computing resources are managed through the *Tethys Compute Admin Pages*. Accessing these resources in your app and configuring app specific resources is made possible through the Compute API.

See also:

For more information on HTCondor see Overview of HTCondor or the HTCondor User Manual.

Key Concepts

HTCondor is a job and resources management middleware. It can be used to create High-Throughput Computing (HTC) systems from diverse computing units including desktop computers or cloud-computing resources. These HTC systems are known as HTCondor pools or clusters. In Tethys the Python library TethysCluster is used to automatically provision HTCondor clusters on Amazon Web Services (AWS) or Microsoft Azure. Portal-wide clusters can be configured by the Tethys Portal admin using the *Tethys Compute Admin Pages*, or app-specific clusters can be configured in apps using the *ClusterManager*. To run jobs to a clusters, it must have a `Scheduler` configured. Portal-wide schedulers can also be configured by the Tethys Portal admin using the *Tethys Compute Admin Pages*, or app-specific schedulers can be set up through the Compute API.

See also:

To see how to configure a job with a `Scheduler` see the *Jobs API*.

Working with the Cluster Manager

The cluster manager can be used to create new computing clusters. It is accessed through the `get_cluster_manager` function.

```
from tethys_sdk.compute import get_cluster_manager

tethyscluster_config_file = '/path/to/TethysCluster/config/file'
cluster_manager = get_cluster_manager(tethyscluster_config_file)
```

For more information on how to use the cluster manager see the TethysCompute documentation

Working with Schedulers

Portal-wide schedulers can be accessed through the `list_schedulers` and the `get_scheduler` functions.

```
from tethys_sdk.compute import list_schedulers, get_scheduler

scheduler = list_schedulers()[0]

# this assumes the Tethys Portal administrator has created a scheduler named 'Default
↪'.
scheduler = get_scheduler('Default')
```

App-specific schedulers can be created with the `create_scheduler` function.

```
from tethys_sdk.compute import create_scheduler

scheduler = create_scheduler(name='my_app_scheduler', host='example.com', username=
↪'root', private_key_path='/path/to/private/key')
```

API Documentation

`tethys_sdk.compute.`**`list_schedulers`**`()`

Gets a list of all scheduler objects registered in the Tethys Portal

Returns List of Schedulers

`tethys_sdk.compute.`**`get_scheduler`**`(`*name*`)`

Gets the scheduler associated with the given name

Parameters **name** (*str*) -- The name of the scheduler to return

Returns The scheduler with the given name or None if no scheduler has the name given.

`tethys_sdk.compute.`**`create_scheduler`**`(`*name*, *host*, *username=None*, *password=None*, *private_key_path=None*, *private_key_pass=None*`)`

Creates a new scheduler

Parameters

- **name** (*str*) -- The name of the scheduler
- **host** (*str*) -- The hostname or IP address of the scheduler
- **username** (*str, optional*) -- The username to use when connecting to the scheduler
- **password** (*str, optional*) -- The password for the username
- **private_key_path** (*str, optional*) -- The path to the location of the SSH private key file
- **private_key_pass** (*str, optional*) -- The passphrase for the private key

Returns The newly created scheduler

Note: The newly created scheduler object is not committed to the database.

1.6.6 Handoff API

Last Updated: October 14, 2015

App developers can use Handoff to launch one app from another app or an external website. Handoff also provides a mechanism for passing data from the originator app to the target app. Using Handoff, apps can be strung together to form a workflow, allowing apps to become modular and specialized in their capabilities. If the handoff is initiated from an app then the HandoffManager can be used. Alternatively, there are REST endpoints (described below) that allow an app to be launched from an external site, but can also be used for app-to-app handoff.

As an example, consider an app called "Hydrograph Plotter" that plots hydrographs. We would like Hydrograph Plotter to be able to accept hydrograph CSV files from other apps so that it can be used as a generic hydrograph viewer. One way to do this would be to define a Handoff endpoint that accepts a URL to a CSV file. The Handoff handler would use that URL to download or pull the CSV file into the app and then redirect it to a page with a plot. The GET request/pull mechanism is used to get around the limitations associated with POST requests, which are required to push or upload files.

Create a Handoff Handler

The first step is to define a Handoff handler. The purpose of the Handoff handler is to handle the transfer data from the originator and then redirect the call to a page in the target app. It is implemented as a function that returns a URL or name of a view. For the example, the Handoff handler could be defined as follows:

```
import os
import requests
from .app import HydrographPlotter

def csv(request, csv_url):
    """
    Handoff handler for csv files.
    """
    # Get a filename in the current user's workspace
    user_workspace = HydrographPlotter.get_user_workspace(request.user)
    filename = os.path.join(user_workspace, 'hydrograph.csv')

    # Initiate a GET request on the CSV URL
    response = requests.get(csv_url, stream=True)

    # Stream content into a file
    with open(filename, 'w') as f:
        for chunk in response.iter_content(chunk_size=512):
            if chunk:
                f.write(chunk)

    return 'hydrograph_plotter:plot_csv'
```

This Handoff handler uses the `requests` library and the *Workspaces API* to download the file and store it in the current user's workspace. Then it returns the name of a controller called `plot_csv` to be redirected to. The `plot_csv` controller would need to know to look for a file in the current user's workspace and plot it.

Register Handoff Handler

The Handoff handler needs to be registered to make it available for other apps to use. This is done by adding the `handoff_handlers` method to the *app class*. This method needs to return a list or tuple of `HandoffHandler` objects.

```
from tethys_sdk.handoff import HandoffHandler

class HydrographPlotter(TethysAppBase):
    """
    Tethys app class for Hydrograph Plotter
    """
    ...

    def handoff_handlers(self):
        """
        Register some handoff handlers
        """
        handoff_handlers = (HandoffHandler(name='plot-csv',
                                           handler='hydrograph_plotter.handoff.csv'),
        )
        return handoff_handlers
```

Execute a Handoff

To execute a Handoff, the originator app or website needs to provide a link of the form:

```
http://<host>/handoff/<app_name>/<handler_name>/?param1=x&param2=y
```

Any parameters that need to be passed with the Handoff call are passed as query parameters on the URL. For our example, the URL would look something like this:

```
http://www.example.com/hydrograph-plotter/plot-csv/?csv_url=http://www.another.com/
↪url/to/file.csv
```

The URL must have query parameters for each argument defined in the Handoff handler function or it will throw an error. It will also throw an error if extra query parameters are provided that are not defined as arguments for the Handoff handler function.

View Handoff Endpoints for Apps

For convenience, a list of the available Handoff endpoints for an app can be viewed by visiting the URL:

```
http://<host>/handoff/<app_name>/
```

For our example, the URL would look like this:

```
http://www.example.com/handoff/hydrograph-plotter/
```

The output would look something like this:

```
[{"arguments": ["csv_url"], "name": "plot-csv"}]
```

HandoffManager

If a handoff is initiated from an app to another app on the same instance of Tethys then the HandoffManager can be used. This has several benefits including being able to being able to process the handoff in a controller and use Python to add logic or handle errors. Additionally, the HandoffManager will expose HandoffHandlers that are marked as "internal". An internal HanoffHandler can take advantage of the assumption that the both sides of the handoff are on the same system by, for example, using file paths and symbolic links rather than passing large files over the network.

A HandoffHandler can be marked as internal when it is registered in *app class*.

```
from tethys_sdk.handoff import HandoffHandler

class HydrographPlotter(TethysAppBase):
    """
    Tethys app class for Hydrograph Plotter
    """
    ...

    def handoff_handlers(self):
        """
        Register some handoff handlers
        """
        handoff_handlers = (HandoffHandler(name='internal-plot-csv',
                                           handler='hydrograph_plotter.handoff.csv_
↪internal',
                                           internal=True),
        )
        return handoff_handlers
```

An example of an internal HandoffHandler:

```
import os
import requests
from .app import HydrographPlotter

def csv_internal(request, path_to_csv):
    """
    Internal handoff handler for csv files.
    """
    # Get a filename in the current user's workspace
    user_workspace = HydrographPlotter.get_user_workspace(request.user)

    # Create symbolic link to the csv in the user's workspace
    src = path_to_csv
    dst = os.path.join(user_workspace, 'hydrograph.csv')

    try:
        os.symlink(src, dst)
    except OSError:
        pass

    return 'hydrograph_plotter:plot_csv'
```

An example of initiating a handoff with the HandoffManager from a controller:

```
def plot(request):
    handoff_manager = app.get_handoff_manager()
    app_name = 'hydrograph_plotter'
```

(continues on next page)

(continued from previous page)

```
    handler_name = 'internal-plot-csv'

    handler = handoff_manager.get_handler(handler_name, app_name)
    if handler:
        try:
            return redirect(handler(request, path_to_netcdf_file=file_path))
        except Exception, e:
            pass

return redirect(reverse('my_app:home', kwargs={'message': 'Hydrograph plotting is not␣
↪working.'}))
```

HandoffManager API

class tethys_apps.base.handoff.**HandoffManager**(*app*)

An object that is used to interact with HandoffHandlers.

app

Instance of a TethysAppBase object.

Type str

handlers

A list of HandoffHandlers registered in the app.

Type list[HandoffHandler]

valid_handlers

A filtered list of only the valid HandoffHandlers.

Type list[HandoffHandler]

get_capabilities(*app_name=None*, *external_only=False*, *jsonify=False*)

Gets a list of the valid handoff handlers.

Parameters

- **app_name** (*str, optional*) -- The name of another app whose capabilities should be listed. Defaults to None in which case the capabilities of the current app will be listed.
- **external_only** (*bool, optional*) -- If True only return handlers where the internal attribute is False. Default is False.
- **jsonify** (*bool, optional*) -- If True return the JSON representation of the handlers is used. Default is False.

Returns A list of valid HandoffHandler objects (or a JSON string if jsonify=True) representing the capabilities of app_name, or None if no app with app_name is found.

get_handler(*handler_name*, *app_name=None*)

Returns the HandoffHandler with name == handler_name.

Parameters

- **handler_name** (*str*) -- the name of a HandoffHandler object.
- **app_name** (*str, optional*) -- the name of the app with handler_name. Defaults to None in which case the current app will be used.

Returns A HandoffHandler object where the name attribute is equal to handler_name or None if no HandoffHandler with that name is found or no app with app_name is found.

handoff(*request*, *handler_name*, *app_name=None*, *external_only=True*, ***kwargs*)

Calls handler if it is not internal and if it exists for the app.

Parameters

- **request** (*HttpRequest*) -- The request object passed by the http call.
- **handler_name** (*str*) -- The name of the HandoffHandler object to handle the handoff. Must not be internal.
- **app_name** (*str, optional*) -- The name of another app where the handler should exist. Defaults to None in which case the current app will attempt to handle the handoff.
- ****kwargs** -- Key-value pairs to be passed on to the handler.

Returns HttpResponse object.

1.6.7 Jobs API

Last Updated: December 27, 2018

The Jobs API provides a way for your app to run asynchronous tasks (meaning that after starting a task you don't have to wait for it to finish before moving on). As an example, you may need to run a model that takes a long time (potentially hours or days) to complete. Using the Jobs API you can create a job that will run the model, and then leave it to run while your app moves on and does other stuff. You can check the job's status at any time, and when the job is done the Jobs API will help retrieve the results.

Key Concepts

To facilitate interacting with jobs asynchronously, the details of the jobs are stored in a database. The Jobs API provides a job manager to handle the details of working with the database, and provides a simple interface for creating and retrieving jobs. The Jobs API supports various types of jobs (see *Job Types*).

Job Manager

The Job Manager is used in your app to interact with the jobs database. It facilitates creating and querying jobs.

Using the Job Manager in your App

To use the Job Manager in your app you first need to import the TethysAppBase subclass from the app.py module:

```
from app import MyFirstApp as app
```

You can then get the job manager by calling the method `get_job_manager` on the app.

```
job_manager = app.get_job_manager()
```

You can now use the job manager to create a new job, or retrieve an existing job or jobs.

Creating and Executing a Job

To create a new job call the `create_job` method on the job manager. The required arguments are:

- `name`: A unique string identifying the job
- `user`: A user object, usually from the request argument: *request.user*
- `job_type`: A string specifying on of the supported job types (see *Job Types*)

Any other job attributes can also be passed in as *kwargs*.

```
# get the path to the app workspace to reference job files
app_workspace = app.get_app_workspace().path

# create a new job from the job manager
job = job_manager.create_job(
    name='myjob_{id}',  # required
    user=request.user,  # required
    job_type='CONDOR',  # required

    # any other properties can be passed in as kwargs
    attributes=dict(attribute1='attr1'),
    condorpy_template_name='vanilla_transfer_files',
    remote_input_files=(
        os.path.join(app_workspace, 'my_script.py'),
        os.path.join(app_workspace, 'input_1'),
        os.path.join(app_workspace, 'input_2')
    )
)

# properties can also be added after the job is created
job.extended_properties = {'one': 1, 'two': 2}

# each job type may provided methods to further specify the job
job.set_attribute('executable', 'my_script.py')

# save or execute the job
job.save()
# or
job.execute()
```

Before a controller returns a response the job must be saved or else all of the changes made to the job will be lost (executing the job automatically saves it). If submitting the job takes a long time (e.g. if a large amount of data has to be uploaded to a remote scheduler) then it may be best to use AJAX to execute the job.

Tip: The *Jobs Table Gizmo* has a built-in mechanism for submitting jobs with AJAX. If the *Jobs Table Gizmo* is used to submit the jobs then be sure to save the job after it is created.

Job Types

The Jobs API is designed to support multiple job types. Each job type provides a different framework and environment for executing jobs. When creating a new job you must specify its type by passing in the *job_type* argument. Currently the supported job types are:

- 'BASIC'
- 'CONDOR' or 'CONDORJOB'
- 'CONDORWORKFLOW'

Additional job attributes can be passed into the *create_job* method of the job manager or they can be specified after the job is instantiated. All jobs have a common set of attributes, and then each job type may add additional attributes.

The following attributes can be defined for all job types:

- `name` (string, required): a unique identifier for the job. This should not be confused with the job template name. The template name identifies a template from which jobs can be created and is set when the template is created. The job `name` attribute is defined when the job is created (see *Creating and Executing a Job*).
- `description` (string): a short description of the job.
- `workspace` (string): a path to a directory that will act as the workspace for the job. Each job type may interact with the workspace differently. By default the workspace is set to the user's workspace in the app that is creating the job.
- `extended_properties` (dict): a dictionary of additional properties that can be used to create custom job attributes.

All job types also have the following read-only attributes:

- `user` (User): the user who created the job.
- `label` (string): the package name of the Tethys App used to created the job.
- `creation_time` (datetime): the time the job was created.
- `execute_time` (datetime): the time that job execution was started.
- `start_time` (datetime):
- `completion_time` (datetime): the time that the job status changed to 'Complete'.
- `status` (string): a string representing the state of the job. Possible statuses are:
  - 'Pending'
  - 'Submitted'
  - 'Running'
  - 'Complete'
  - 'Error'
  - 'Aborted'
  - 'Various'*
  - 'Various-Complete'*

  *used for job types with multiple sub-jobs (e.g. CondorWorkflow).

Specific job types may define additional attributes. The following job types are available.

Basic Job Type

Last Updated: December 27, 2018

The Basic Job type is a sample job type for creating dummy jobs. It has all of the basic properties and methods of a job, but it doesn't have any mechanism for running jobs. It's primary purpose is for demonstration. There are no additional attributes for the BasicJob type other than the common set of job attributes.

Creating a Basic Job

To create a job call the `create_job` method on the job manager. The required parameters are `name`, `user` and `job_type`. Any other job attributes can also be passed in as *kwargs*.

```
# create a new job
job = job_manager.create_job(
    name='unique_job_name',
    user=request.user,
    template_name='BASIC',
    description='This is a sample basic job. It can't actually compute anything.',
    extended_properties={
        'app_spcific_property': 'default_value',
    }
)
```

Before a controller returns a response the job must be saved or else all of the changes made to the job will be lost (executing the job automatically saves it). If submitting the job takes a long time (e.g. if a large amount of data has to be uploaded to a remote scheduler) then it may be best to use AJAX to execute the job.

API Documentation

class tethys_compute.models.**BasicJob**(**args*, ***kwargs*)
Basic job type. Use this class as a model for subclassing TethysJob

class tethys_sdk.jobs.**BasicJobTemplate**(*name*, *parameters=None*)
DEPRECATED A subclass of JobTemplate with the `type` argument set to BasicJob.

Parameters

- **name** (*str*) -- Name to refer to the template.
- **parameters** (*dict*) -- A dictionary of parameters to pass to the BasicJob constructor.

Condor Job Type

Last Updated: December 27, 2018

The *Condor Job Type* (and *Condor Workflow Job Type*) enable the real power of the jobs API by combining it with the *Compute API*. This make it possible for jobs to be offloaded from the main web server to a scalable computing cluster, which in turn enables very large scale jobs to be processed.

See also:

The Condor Job and the Condor Workflow job types use the CondorPy library to submit jobs to HTCondor compute pools. For more information on CondorPy and HTCondor see the CondorPy documentation and specifically the Overview of HTCondor.

Creating a Condor Job

To create a job call the `create_job` method on the job manager. The required parameters are `name`, `user` and `job_type`. Any other job attributes can also be passed in as *kwargs*.

```
from tethys_sdk.compute import list_schedulers
from .app import MyApp as app

def some_controller(request):

    # get the path to the app workspace to reference job files
    app_workspace = app.get_app_workspace().path

    # create a new job from the job manager
    job = job_manager.create_job(
        name='myjob_{id}',  # required
        user=request.user,  # required
        job_type='CONDOR',  # required

        # any other properties can be passed in as kwargs
        attributes=dict(
            transfer_input_files=('../input_1', '../input_2'),
            transfer_output_files=('example_output1', example_output2),
        ),
        condorpy_template_name='vanilla_transfer_files',
        remote_input_files=(
            os.path.join(app_workspace, 'my_script.py'),
            os.path.join(app_workspace, 'input_1'),
            os.path.join(app_workspace, 'input_2')
        )
    )

    # properties can also be added after the job is created
    job.extended_properties = {'one': 1, 'two': 2}

    # each job type may provided methods to further specify the job
    job.set_attribute('executable', 'my_script.py')

    # get a scheduler for the job
    my_scheduler = list_schedulers()[0]
    job.scheduler = my_scheduler

    # save or execute the job
    job.save()
    # or
    job.execute()
```

Before a controller returns a response the job must be saved or else all of the changes made to the job will be lost (executing the job automatically saves it). If submitting the job takes a long time (e.g. if a large amount of data has to be uploaded to a remote scheduler) then it may be best to use AJAX to execute the job.

API Documentation

class tethys_compute.models.**CondorJob**(**args*, ***kwargs*)
CondorPy Job job type

class tethys_sdk.jobs.**CondorJobTemplate**(*name*, *job_description*, *scheduler=None*, ***kwargs*)
DEPRECATED A subclass of the JobTemplate with the `type` argument set to CondorJob.

Parameters

- **name** (*str*) -- Name to refer to the template.
- **job_description** (*CondorJobDescription*) -- An object containing the attributes for the condorpy job.
- **scheduler** (*Scheduler*) -- An object containing the connection information to submit the condorpy job remotely.

Condor Workflow Job Type

Last Updated: December 27, 2018

A Condor Workflow provides a way to run a group of jobs (which can have hierarchical relationships) as a single (Tethys) job. The hierarchical relationships are defined as parent-child relationships. For example, suppose a workflow is defined with three jobs: `JobA`, `JobB`, and `JobC`, which must be run in that order. These jobs would be defined with the following relationships: `JobA` is the parent of `JobB`, and `JobB` is the parent of `JobC`.

See also:

The Condor Workflow job type uses the CondorPy library to submit jobs to HTCondor compute pools. For more information on CondorPy and HTCondor see the CondorPy documentation and specifically the Overview of HTCondor.

Creating a Condor Workflow

Creating a Condor Workflow job involves 3 steps:

1. Create an empty Workflow job from the job manager.
2. Create the jobs that will make up the workflow with *CondorWorkflowJobNode*
3. Define the relationships among the nodes

```
from tethys_sdk.jobs import CondorWorkflowJobNode
from .app import MyApp as app

def some_controller(request):

    # get the path to the app workspace to reference job files
    app_workspace = app.get_app_workspace().path

    workflow = job_manager.create_job(
        name='MyWorkflowABC',
        user=request.user,
        job_type='CONDORWORKFLOW',
        scheduler=None,
    )
    workflow.save()

    job_a = CondorWorkflowJobNode(
```

(continues on next page)

(continued from previous page)

```
        name='JobA',
        workflow=workflow,
        condorpy_template_name='vanilla_transfer_files',
        remote_input_files=(
            os.path.join(app_workspace, 'my_script.py'),
            os.path.join(app_workspace, 'input_1'),
            os.path.join(app_workspace, 'input_2')
        ),
        attributes=dict(
            executable='my_script.py',
            transfer_input_files=('../input_1', '../input_2'),
            transfer_output_files=('example_output1', 'example_output2'),
        )
    )
    job_a.save()

    job_b = CondorWorkflowJobNode(
        name='JobB',
        workflow=workflow,
        condorpy_template_name='vanilla_transfer_files',
        remote_input_files=(
            os.path.join(app_workspace, 'my_script.py'),
            os.path.join(app_workspace, 'input_1'),
            os.path.join(app_workspace, 'input_2')
        ),
        attributes=dict(
            executable='my_script.py',
            transfer_input_files=('../input_1', '../input_2'),
            transfer_output_files=('example_output1', 'example_output2'),
        ),
    )
    job_b.save()

    job_c = CondorWorkflowJobNode(
        name='JobC',
        workflow=workflow,
        condorpy_template_name='vanilla_transfer_files',
        remote_input_files=(
            os.path.join(app_workspace, 'my_script.py'),
            os.path.join(app_workspace, 'input_1'),
            os.path.join(app_workspace, 'input_2')
        ),
        attributes=dict(
            executable='my_script.py',
            transfer_input_files=('../input_1', '../input_2'),
            transfer_output_files=('example_output1', 'example_output2'),
        ),
    )
    job_c.save()

    job_b.add_parent(job_a)
    job_c.add_parent(job_b)

    workflow.save()
    # or
    workflow.execute()
```

Note: The *CondorWorkflow* object must be saved before the *CondorWorkflowJobNode* objects can be instantiated, and the *CondorWorkflowJobNode* objects must be saved before you can define the relationships.

Before a controller returns a response the job must be saved or else all of the changes made to the job will be lost (executing the job automatically saves it). If submitting the job takes a long time (e.g. if a large amount of data has to be uploaded to a remote scheduler) then it may be best to use AJAX to execute the job.

API Documentation

class tethys_compute.models.**CondorWorkflow**(**args*, ***kwargs*)
CondorPy Workflow job type

class tethys_compute.models.**CondorWorkflowNode**(**args*, ***kwargs*)
Base class for CondorWorkflow Nodes

Parameters

- **name** (*str*) --
- **workflow** (*CondorWorkflow*) -- instance of a *CondorWorkflow* that node belongs to
- **parent_nodes** (*list*) -- list of *CondorWorkflowNode* objects that are prerequisites to this node
- **pre_script** (*str*) --
- **pre_script_args** (*str*) --
- **post_script** (*str*) --
- **post_script_args** (*str*) --
- **variables** (*dict*) --
- **priority** (*int*) --
- **category** (*str*) --
- **retry** (*int*) --
- **retry_unless_exit_value** (*int*) --
- **pre_skip** (*int*) --
- **abort_dag_on** (*int*) --
- **dir** (*str*) --
- **noop** (*bool*) --
- **done** (*bool*) --

For a description of the arguments see http://research.cs.wisc.edu/htcondor/manual/v8.6/2_10DAGMan_Applications.html

class tethys_compute.models.**CondorWorkflowJobNode**(**args*, ***kwargs*)
CondorWorkflow JOB type node

class tethys_sdk.jobs.**CondorWorkflowTemplate**(*name*, *parameters=None*, *jobs=None*, *max_jobs=None*, *config=''*, ***kwargs*)
DEPRECATED A subclass of the JobTemplate with the `type` argument set to CondorWorkflow.

Parameters

- **name** (*str*) -- Name to refer to the template.
- **parameters** (*dict, DEPRECATED*) -- A dictionary of key-value pairs. Each Job type defines the possible parameters.
- **jobs** (*list*) -- A list of CondorWorkflowJobTemplates.
- **max_jobs** (*dict, optional*) -- A dictionary of category-max_job pairs defining the maximum number of jobs that will run simultaneously from each category.
- **config** (*str, optional*) -- A path to a configuration file for the condorpy DAG.

class tethys_sdk.jobs.**CondorWorkflowJobTemplate**(*name*, *job_description*, ***kwargs*)
DEPRECATED A subclass of the CondorWorkflowNodeBaseTemplate with the `type` argument set to CondorWorkflowJobNode.

Parameters

- **name** (*str*) -- Name to refer to the template.
- **job_description** (*CondorJobDescription*) -- An instance of *CondorJobDescription* containing of key-value pairs of job attributes.

Retrieving Jobs

Two methods are provided to retrieve jobs: `list_jobs` and `get_job`. Jobs are automatically filtered by app. An optional `user` parameter can be passed in to these methods to further filter jobs by the user.

```
# get list of all jobs created in your app
job_manager.list_jobs()

# get list of all jobs created by current user in your app
job_manager.list_jobs(user=request.user)

# get job with id of 27
job_manager.get_job(job_id=27)

# get job with id of 27 only if it was created by current user
job_manager.get_job(job_id=27, user=request.user)
```

Caution: Be thoughtful about how you retrieve jobs. The user filter is provided to prevent unauthorized users from accessing jobs that don't belong to them.

Jobs Table Gizmo

The Jobs Table Gizmo facilitates job management through the web interface and is designed to be used in conjunction with the Job Manager. It can be configured to list any of the properties of the jobs, and will automatically update the job status, and provides buttons to run, delete, or view job results. The following code sample shows how to use the job manager to populate the jobs table:

```
job_manager = app.get_job_manager()

jobs = job_manager.list_jobs(request.user)

jobs_table_options = JobsTable(jobs=jobs,
                               column_fields=('id', 'description', 'run_time'),
                               hover=True,
                               striped=False,
                               bordered=False,
                               condensed=False,
                               results_url='my_first_app:results',
                               )
```

See also:

Jobs Table

Job Status Callback

Each job has a callback URL that will update the job's status. The URL is of the form:

```
http://<host>/update-job-status/<job_id>/
```

For example, a URL may look something like this:

```
http://example.com/update-job-status/27/
```

The output would look something like this:

```
{"success": true}
```

This URL can be retrieved from the job manager with the `get_job_status_callback_url` method, which requires a *request* object and the id of the job.

```
job_manager = app.get_job_manager()
callback_url = job_manager.get_job_status_callback_url(request, job_id)
```

API Documentation

class tethys_compute.job_manager.**JobManager**(*app*)

A manager for interacting with the Jobs database providing a simple interface creating and retrieving jobs.

Note: Each app creates its own instance of the JobManager. The `get_job_manager` method returns the app.

```
from app import MyApp as app

job_manager = app.get_job_manager()
```

---

create_job(*name*, *user*, *template_name=None*, *job_type=None*, ***kwargs*)

Creates a new job from a JobTemplate.

Parameters

- **name** (*str*) -- The name of the job.
- **user** (*User*) -- A User object for the user who creates the job.
- **job_type** (TethysJob) -- A subclass of TethysJob.
- ****kwargs** --

Returns A new job object of the type specified by job_type.

get_job(*job_id*, *user=None*, *filters=None*)

Gets a job by id.

Parameters

- **job_id** (*int*) -- The id of the job to get.
- **user** (*User, optional*) -- The user to filter the jobs by.

Returns A instance of a subclass of TethysJob if a job with job_id exists (and was created by user if the user argument is passed in).

get_job_status_callback_url(*request*, *job_id*)

Get the absolute url to call to update job status

list_jobs(*user=None*, *order_by='id'*, *filters=None*)
Lists all the jobs from current app for current user.

Parameters

- **user** (*User, optional*) -- The user to filter the jobs by. Default is None.
- **order_by** (*str, optional*) -- An expression to order jobs. Default is 'id'.
- **filters** (*dict, optional*) -- A list of key-value pairs to filter the jobs by. Default is None.

Returns A list of jobs created in the app (and by the user if the user argument is passed in).

class tethys_compute.models.**TethysJob**(**args*, ***kwargs*)
Base class for all job types. This is intended to be an abstract class that is not directly instantiated.

References

Condor Job Description

Last Updated: March 29, 2016

DEPRECATED

Both the *Condor Job Type* or the *Condor Workflow Job Type* facilitate running jobs with HTCondor using the CondorPy library, and both use `CondorJobDescription` objects which stores attributes used to initialize the CondorPy job. The `CondorJobDescription` accepts as parameters any HTCondor job attributes.

Note: In addition to any HTCondor job attributes, the `CondorJobDescription` also accepts a special parameter called `condorpy_template_name`. This parameter accepts a string naming a CondorPy Template, which is a pre-configured set of HTCondor job attributes. For more information about CondorPy templates and HTCondor job attributes see the CondorPy documentation.

Important: Perhaps the most confusing part about `CondorJobDescription` parameters is the file paths. Different parameters require that the paths be defined relative to different locations. For more information about how to define paths for HTCondor job attributes see the CondorPy documentation

---

Setting up a CondorJobDescription

```
from tethys_sdk.jobs import CondorJobDescription

...

    my_job_description = CondorJobDescription(condorpy_template_name='vanilla_
↪transfer_files',
                                              remote_input_files=('$(APP_WORKSPACE)/
↪my_script.py', '$(APP_WORKSPACE)/input_1', '$(USER_WORKSPACE)/input_2'),
                                              executable='my_script.py',
                                              transfer_input_files=('../input_1', '../
↪input_2'),
                                              transfer_output_files=('example_output1
↪', example_output2),
                                              )
```

(continues on next page)

(continued from previous page)

```
...
```

1.6.8 Permissions API

Last Updated: May 28, 2016

Permissions allow you to restrict access to certain features or content of your app. We recommend creating permissions for specific tasks or features of your app (e.g.: "Can view the map" or "Can delete projects") and then define groups of permissions to create the roles for you app.

Create Permissions and Permission Groups

Declare the `permissions` method in the app class and have it return a list or tuple of `Permission` and/or `PermissionGroup` objects. Permissions are synced everytime you start or restart the development server (i.e.: `tethys manage start`) or Apache server in production.

Once you have created permissions and permission groups for your app, they will be available for the Tethys Portal administrator to assign to users. See the *Auth Token* documentation for more details.

`TethysAppBase.`**`permissions`**`()`

Override this method to define permissions for your app.

Returns A list or tuple of `Permission` or `PermissionGroup` objects.

Return type iterable

Example:

```
from tethys_sdk.permissions import Permission, PermissionGroup

class MyFirstApp(TethysAppBase):

    def permissions(self):
        """
        Example permissions method.
        """
        # Viewer Permissions
        view_map = Permission(
            name='view_map',
            description='View map'
        )

        delete_projects = Permission(
            name='delete_projects',
            description='Delete projects'
        )

        create_projects = Permission(
            name='create_projects',
            description='Create projects'
        )

        admin = PermissionGroup(
            name='admin',
            permissions=(delete_projects, create_projects)
        )
```

(continues on next page)

(continued from previous page)

```
        permissions = (admin, view_map)

        return permissions
```

Permission and Permission Group Objects

class tethys_sdk.permissions.**Permission**(*name*, *description*)

Defines an object that represents a permission for an app.

name

The code name for the permission. Only numbers, letters, and underscores allowed.

Type string

description

Short description of the permission for the admin interface.

Type string

Example:

```
from tethys_sdk.permissions import Permission

create_projects = Permission(
    name='create_projects',
    description='Create projects'
)
```

class tethys_sdk.permissions.**PermissionGroup**(*name*, *permissions=[]*)

Defines an object that represents a permission group for an app.

name

The name for the group. Only numbers, letters, and underscores allowed.

Type string

permissions

A list or tuple of Permission objects.

Type iterable

Example:

```
from tethys_sdk.permissions import Permission, PermissionGroup

create_projects = Permission(
    name='create_projects',
    description='Create projects'
)

delete_projects = Permission(
    name='delete_projects',
    description='Delete projects'
)

admin = PermissionGroup(
    name='admin',
```

(continues on next page)

(continued from previous page)

```
    permissions=(create_projects, delete_projects)
)
```

Check Permission

Use the `has_permission` method to check whether the user of the current request has a permission.

`permissions.`**`has_permission`**(*perm*, *user=None*)

Returns True if the user of the given request has the given permission for the app. If a user object is provided, it is tested instead of the request user. The Request object is still required to derive the app context of the permission check.

Parameters

- **`request`** (*`Request`*) -- The current request object.
- **`perm`** (*`string`*) -- The name of the permission (e.g. 'create_things').
- **`user`** (*`django.contrib.auth.models.User`*) -- A user object to test instead of the user provided in the request.

Example:

```
from tethys_sdk.permissions import has_permission

def my_controller(request):
    """
    Example controller
    """

    can_create_projects = has_permission(request, 'create_projects')

    if can_create_projects:
        ...
```

Controller Decorator

Use the `permission_required` decorator to enforce permissions for an entire controller.

`permissions.`**`permission_required`**(***kwargs*)

Decorator for Tethys App controllers that checks whether a user has a permission.

Parameters

- **`*args`** -- Any number of permission names for the app (e.g. 'create_projects')
- **`**kwargs`** -- Any of keyword arguments specified below.

Valid Kwargs:

- **message: (string):** Override default message that is displayed to user when permission is denied. Default message is "We're sorry, but you are not allowed to perform this operation.".
- **raise_exception (bool):** Raise 403 error if True. Defaults to False.
- **use_or (bool):** When multiple permissions are provided and this is True, use OR comparison rather than AND comparison, which is default.

Example:

```
from tethys_sdk.permissions import permission_required

# Basic use
@permission_required('create_projects')
def my_controller(request):
```

(continues on next page)

(continued from previous page)

```
    """
    Example controller
    """
    ...

# Custom message when permission is denied
@permission_required('create_projects', message="You do not have permission to␣
↪create projects")
def my_controller(request):
    """
    Example controller
    """
    ...

# Multiple permissions with AND comparison (must pass both permissions tests)
@permission_required('create_projects', 'delete_projects')
def my_controller(request):
    """
    Example controller
    """
    ...

# Multiple permissions with OR comparison (must pass at least one permissions␣
↪test)
@permission_required('create_projects', 'delete_projects', use_or=True)
def my_controller(request):
    """
    Example controller
    """
    ...

# Raise 403 exception rather than redirecting and displaying message (useful for␣
↪REST controllers).
@permission_required('create_projects', raise_exception=True)
def my_controller(request):
    """
    Example controller
    """
    ...
```

1.6.9 REST API

REST API's in Tethys Platform use token authentication (see: http://www.django-rest-framework.org/api-guide/authentication/#tokenauthentication).

You can find the API token for your user on the user management page (http://{[}HOST_Portal{]}/user/{[}username{]}/).

Example Url Map (app.py):

```
UrlMap(name='api_get_data',
       url='[your_app_name]/api/get_data',
       controller='[your_app_name].api.get_data')
```

Example API Controller (api.py):

```
from django.http import JsonResponse
from rest_framework.authentication import TokenAuthentication
from rest_framework.decorators import api_view, authentication_classes

@api_view(['GET'])
@authentication_classes((TokenAuthentication,))
def get_data(request):
    '''
    API Controller for getting data
    '''
    name = request.GET.get('name')
    data = {"name": name}
    return JsonResponse(data)
```

Example Accessing Data:

```
 >>> import requests
 >>> res = requests.get('http://[HOST_Portal]/apps/[your_app_name]/api/get_data?
↪name=oscar',
                       headers={'Authorization': 'Token asdfqwer1234'})
>>> da.text
'{"name": "oscar"}'
```

1.6.10 Template Gizmos API

Last Updated: May 2017

Template Gizmos are building blocks that can be used to create beautiful interactive controls for web apps. Using the Template Gizmos API, developers can add date-pickers, plots, and maps to their app pages with minimal coding. This article provides an overview of how to use Gizmos.

For a detailed explanation and code examples of each Gizmo, see the *Gizmos Options Objects* section.

Working with Gizmos

The best way to illustrate how to use Template Gizmos is to look at an example. The following example illustrates how to add a date picker to a page using the `DatePicker Gizmo`. The basic workflow involves three steps:

1. Define gizmo options object in the controller for the template
2. Load gizmo library in the template
3. Insert the gizmo tag in the template at the desired location

A detailed description of each step follows.

1. Define Gizmo Options Object in Controller

The first step is to import the appropriate options object and configure the Gizmo. This is performed in the controller of the template where the Gizmo will be used.

In this case, we import `DatePicker` and initialize a new object called `date_picker` with the desired options. Then we pass the object to the template via the `context` dictionary:

```
from tethys_sdk.gizmos import DatePicker

def gizmo_controller(request):
    """
    Example of a controller that defines options for a Template Gizmo.
    """
    date_picker = DatePicker(
        name='date',
        display_text='Date',
        autoclose=True,
        format='MM d, yyyy',
        start_date='2/15/2014',
        start_view='decade',
        today_button=True,
        initial='February 15, 2017'
    )

    context = {'date_picker': date_picker}

    return render(request, 'my_first_app/template.html', context)
```

The *Gizmos Options Objects* section provides detailed descriptions of each Gizmo option object, valid parameters, and examples of how to use them.

2. Load Gizmo Library in Template

Now near the top of the template where the Gizmo will be inserted, load the `tethys_gizmos` library using the Django load tag. This only needs to be done once per template:

```
{% load tethys_gizmos %}
```

4. Insert the Gizmo

The `gizmo` tag is used to insert the date picker anywhere in the template. The `gizmo` tag accepts a Gizmo object of configuration options for the Gizmo:

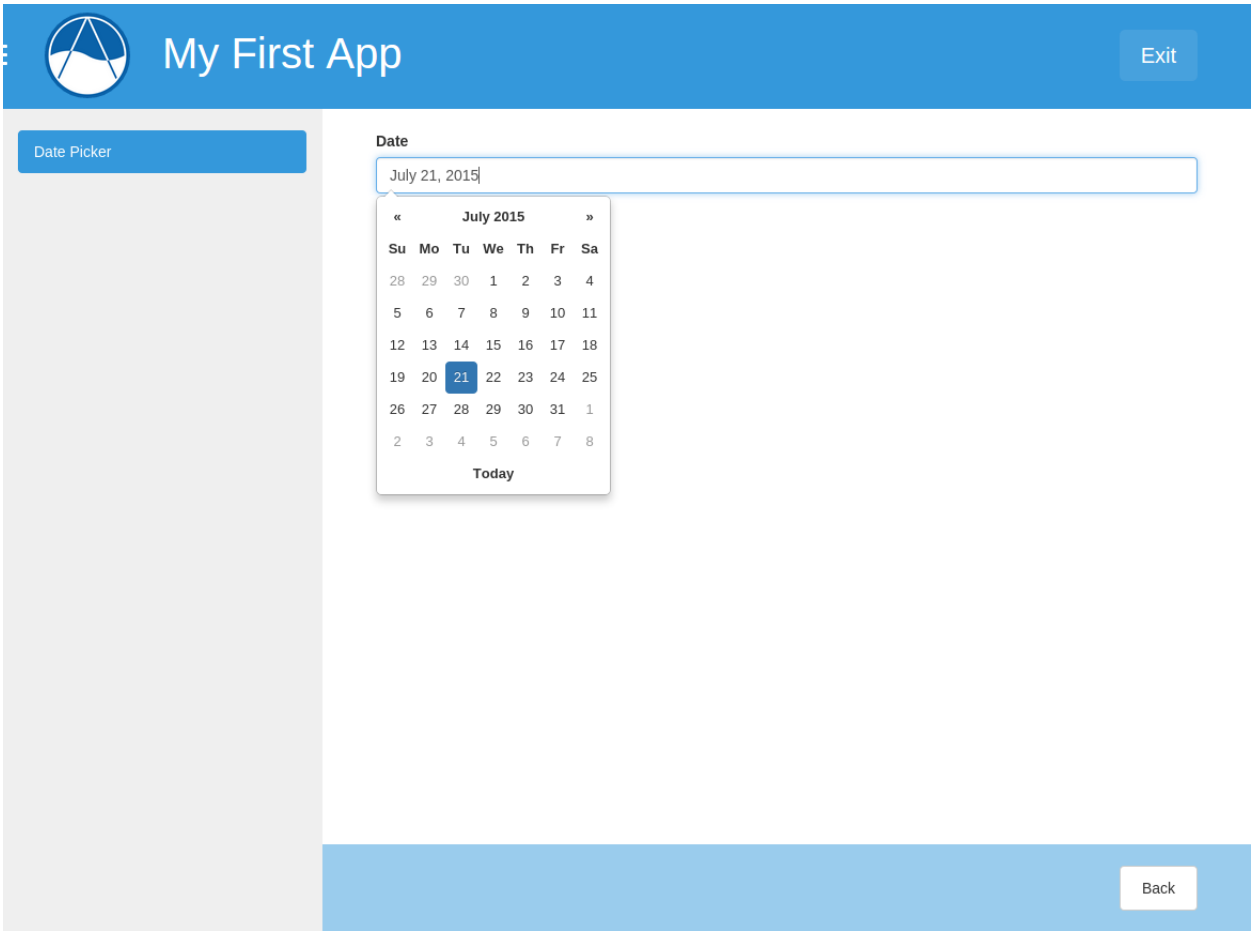
```
{% gizmo <options> %}
```

For this example, call the `gizmo` tag with the `date_picker` object that was defined in the controller and passed to the template as an argument:

```
{% gizmo date_picker %}
```

Rendered Gizmo

The Gizmo tag is replaced with the appropriate HTML, JavaScript, and CSS that is needed to render the Gizmo. In the example, the date picker is inserted at the location of the `gizmo` tag. The template with the rendered date picker would look something like this:

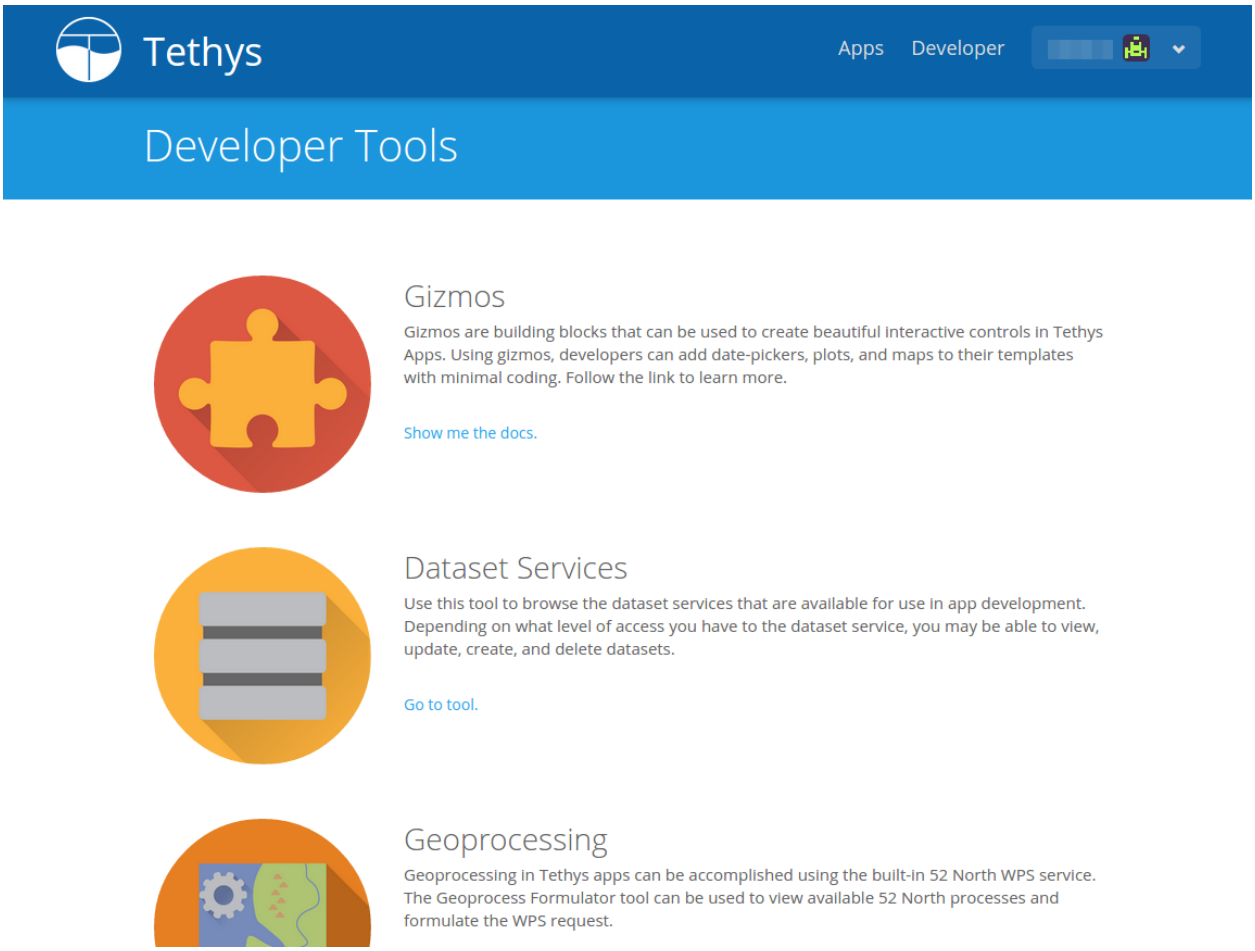


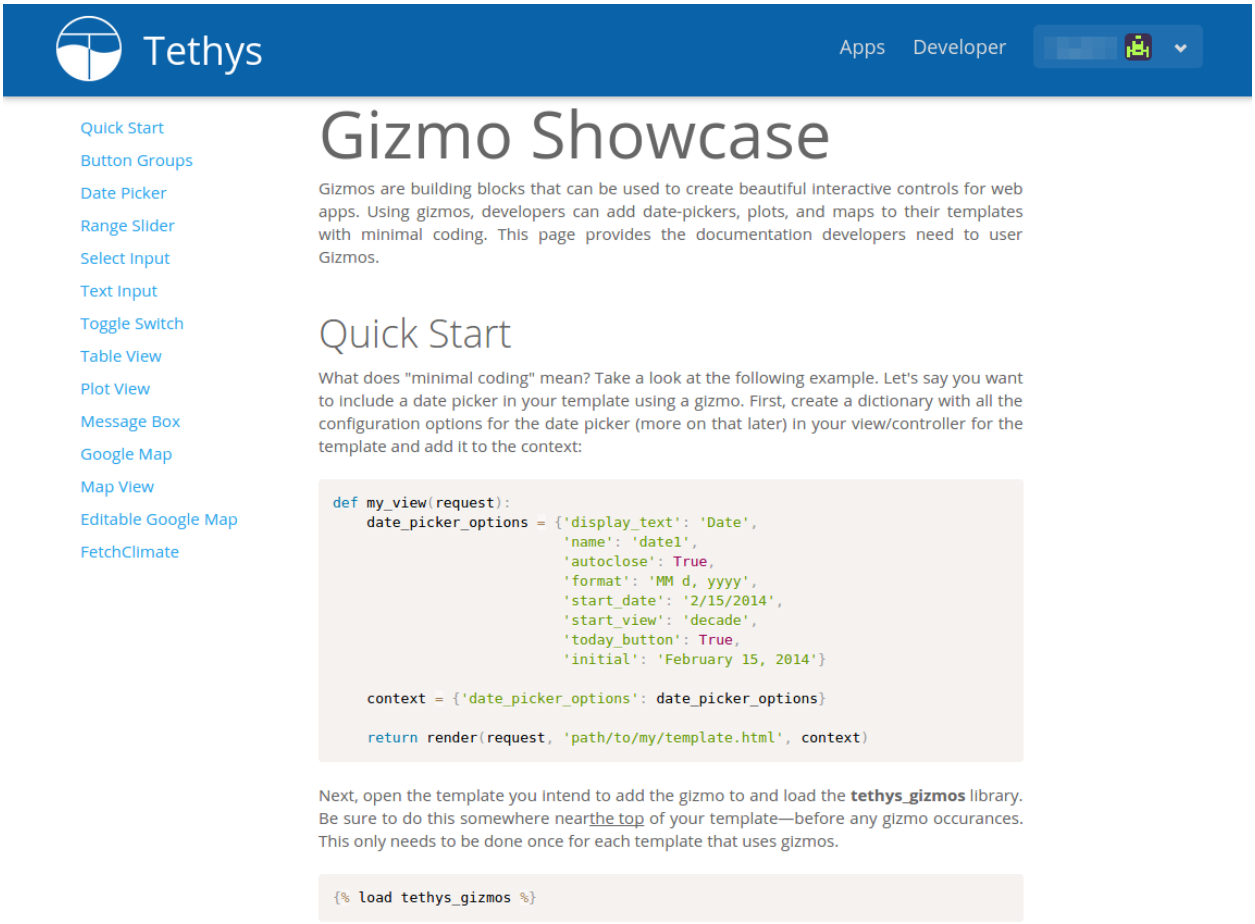
Gizmo Showcase

Live demos of each Gizmo is provided as a developer tool called "Gizmo Showcase". To access the Gizmo Showcase, start up your development server and navigate to the home page of your Tethys Portal at http://127.0.0.1:8000. Login and select the `Developer` link from the main navigation. This will bring up the Developer Tools page of your Tethys Portal:

Select the Gizmos developer tool and you will be brought to the Gizmo Showcase page:

For explanations the Gizmo Options objects and code examples, refer to the *Gizmos Options Objects* section.


Tethys
Apps
Developer
Developer Tools
Gizmos
Gizmos are building blocks that can be used to create beautiful interactive controls in Tethys Apps. Using gizmos, developers can add date-pickers, plots, and maps to their templates with minimal coding. Follow the link to learn more.
Show me the docs.
Dataset Services
Use this tool to browse the dataset services that are available for use in app development. Depending on what level of access you have to the dataset service, you may be able to view, update, create, and delete datasets.
Go to tool.
Geoprocessing
Geoprocessing in Tethys apps can be accomplished using the built-in 52 North WPS service. The Geoprocess Formulator tool can be used to view available 52 North processes and formulate the WPS request.


Tethys
Apps
Developer
Quick Start
Button Groups
Date Picker
Range Slider
Select Input
Text Input
Toggle Switch
Table View
Plot View
Message Box
Google Map
Map View
Editable Google Map
FetchClimate
Gizmo Showcase
Gizmos are building blocks that can be used to create beautiful interactive controls for web apps. Using gizmos, developers can add date-pickers, plots, and maps to their templates with minimal coding. This page provides the documentation developers need to user Gizmos.
Quick Start
What does "minimal coding" mean? Take a look at the following example. Let's say you want to include a date picker in your template using a gizmo. First, create a dictionary with all the configuration options for the date picker (more on that later) in your view/controller for the template and add it to the context:
def my_view(request):
date_picker_options = {'display_text': 'Date',
'name': 'date1',
'autoclose': True,
'format': 'MM d, yyyy',
'start_date': '2/15/2014',
'start_view': 'decade',
'today_button': True,
'initial': 'February 15, 2014'}
context = {'date_picker_options': date_picker_options}
return render(request, 'path/to/my/template.html', context)
Next, open the template you intend to add the gizmo to and load the tethys_gizmos library. Be sure to do this somewhere nearthe top of your template—before any gizmo occurances. This only needs to be done once for each template that uses gizmos.
{% load tethys_gizmos %}

API Documentation

This section contains a brief explanation of the template tags that power Gizmos. These are provided by the `tethys_gizmos` library that you load at the top of templates that use Gizmos.

gizmo

Inserts a Gizmo at the location of the tag.

Parameters:

- **options** (dict) - The configuration options for the Gizmo. The options are Gizmo specific. See the Gizmo Showcase documentation for descriptions of the options that are available.

Example:

```
{% gizmo date_picker %}
```

import_gizmo_dependency

Tells the `gizmo_dependencies` to load in the dependencies for the gizmo. This tag must be in the `import_gizmos` block. This is useful for pre-loading the dependencies of a gizmo that is not loaded on the initial page load (e.g.: loading the gizmos using AJAX after the initial page load).

Parameters:

- **name** (string or literal) - The name of the Gizmo for which to load dependencies as a string (e.g.: "date_picker") or a literal (e.g.: date_picker).

Note: You can get the name of the gizmo through the *gizmo_name* attribute of the gizmo object.

Controller Example:

```
from tethys_sdk.gizmos import DatePicker

def example_controller(request):
    """
    Example of a controller that defines options for a Template Gizmo.
    """
    context = {'date_picker_name': DatePicker.gizmo_name}

    return render(request, 'path/to/my/template.html', context)
```

Template Example:

```
{% block import_gizmos %}
    {% import_gizmo_dependency date_picker_name %}
{% endblock %}
```

gizmo_dependencies

Inserts of the CSS and JavaScript dependencies at the location of the tag for all gizmos loaded in the template with the **gizmo** tag. This tag must appear after all occurrences of the `gizmo` tag. In Tethys Apps, these depenencies are imported for you, so this tag is not required. For external Django projects that use the tethys_gizmos Django app, this tag is required.

Parameters:

- **type** (string or literal, optional) - The type of dependency to import. This parameter can be used to include the CSS and JavaScript dependencies at different locations in the template. Valid values include "css" for CSS dependencies, "global_css" for CSS library dependencies, "js" for JavaScript dependencies, and "global_js" for JavaScript library dependencies.

Examples:

```
# No type parameter
{% gizmo_dependencies %}

# CSS only
{% gizmo_dependencies global_css %}
{% gizmo_dependencies css %}

# JavaScript only
{% gizmo_dependencies global_js %}
{% gizmo_dependencies js %}
```

Gizmos Options Objects

This section provides explanations of each of the Gizmo Options Objects available for configuring Gizmos. It also provides code and usage examples for each object.

Button and Button Group

Last Updated: August 10, 2015

class tethys_sdk.gizmos.**Button**(*display_text=''*, *name=''*, *style=''*, *icon=''*, *href=''*, *submit=False*, *disabled=False*, *attributes={}*, *classes=''*)

display_text
Display text that appears on the button.
Type str

name
Name of the input element that will be used for form submission.
Type str

style
Name of the input element that will be used for form submission.
Type str

icon
Name of a valid Twitter Bootstrap icon class (see the Bootstrap glyphicon reference).
Type str

href
Link for anchor type buttons.

Type str

submit
Set this to true to make the button a submit type button for forms.
Type bool

disabled
Set the disabled state.
Type bool

attributes
A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).
Type dict

classes
Additional classes to add to the primary HTML element (e.g. "example-class another-class").
Type str

Controller Example

```
from tethys_sdk.gizmos import Button

# Single Buttons
map_button = Button(
    display_text='',
    name='map-button',
    icon='glyphicon glyphicon-globe',
    style='info',
    attributes={
        'data-toggle':'tooltip',
        'data-placement':'top',
        'title':'Map'
    }
)

save_button = Button(
    display_text='',
    name='save-button',
    icon='glyphicon glyphicon-floppy-disk',
    style='success',
    attributes={
        'data-toggle':'tooltip',
        'data-placement':'top',
        'title':'Save'
    }
)

edit_button = Button(
    display_text='',
    name='edit-button',
    icon='glyphicon glyphicon-edit',
    style='warning',
    attributes={
        'data-toggle':'tooltip',
        'data-placement':'top',
        'title':'Edit'
    }
)
```

(continues on next page)

(continued from previous page)

```
remove_button = Button(
    display_text='',
    name='remove-button',
    icon='glyphicon glyphicon-remove',
    style='danger',
    attributes={
        'data-toggle':'tooltip',
        'data-placement':'top',
        'title':'Remove'
    }
)

previous_button = Button(
    display_text='Previous',
    name='previous-button',
    attributes={
        'data-toggle':'tooltip',
        'data-placement':'top',
        'title':'Previous'
    }
)

next_button = Button(
    display_text='Next',
    name='next-button',
    attributes={
        'data-toggle':'tooltip',
        'data-placement':'top',
        'title':'Next'
    }
)

context = {
    'map_button': map_button,
    'save_button': save_button,
    'edit_button': edit_button,
    'remove_button': remove_button,
    'previous_button': previous_button,
    'next_button': next_button
}
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo map_button %}
{% gizmo save_button %}
{% gizmo edit_button %}
{% gizmo remove_button %}
{% gizmo previous_button %}
{% gizmo next_button %}
```

class tethys_sdk.gizmos.**ButtonGroup**(*buttons*, *vertical=False*, *attributes=''*, *classes=''*)

The button group gizmo can be used to generate a single button or a group of buttons. Groups of buttons can be stacked horizontally or vertically. For a single button, specify a button group with one button. This gizmo is a wrapper for Twitter Bootstrap buttons.

buttons
A list of dictionaries where each dictionary contains the options for a button.
Type list, required

vertical
Set to true to have button group stack vertically.
Type bool

attributes
A string representing additional HTML attributes to add to the primary element (e.g. "onclick=run_me();").
Type str

classes
Additional classes to add to the primary HTML element (e.g. "example-class another-class").
Type str

Controller Example

```
from tethys_sdk.gizmos import Button, ButtonGroup

# Horizontal Button Group
add_button = Button(display_text='Add',
                    icon='glyphicon glyphicon-plus',
                    style='success')
delete_button = Button(display_text='Delete',
                       icon='glyphicon glyphicon-trash',
                       disabled=True,
                       style='danger')
horizontal_buttons = ButtonGroup(buttons=[add_button, delete_button])

# Vertical Button Group
edit_button = Button(display_text='Edit',
                     icon='glyphicon glyphicon-wrench',
                     style='warning',
                     attributes='id=edit_button')
info_button = Button(display_text='Info',
                     icon='glyphicon glyphicon-question-sign',
                     style='info',
                     attributes='name=info')
apps_button = Button(display_text='Apps',
                     icon='glyphicon glyphicon-home',
                     href='/apps',
                     style='primary')
vertical_buttons = ButtonGroup(buttons=[edit_button, info_button, apps_button],␣
↪vertical=True)

context = {
            'horizontal_buttons': horizontal_buttons,
            'vertical_buttons': vertical_buttons,
          }
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo horizontal_buttons %}
{% gizmo vertical_buttons %}
```

Date Picker

Last Updated: August 10, 2015

class tethys_sdk.gizmos.**DatePicker**(*name*, *display_text=''*, *autoclose=False*, *calendar_weeks=False*, *clear_button=False*, *days_of_week_disabled=''*, *end_date=''*, *format=''*, *min_view_mode='days'*, *multidate=1*, *start_date=''*, *start_view='month'*, *today_button=False*, *today_highlight=False*, *week_start=0*, *initial=''*, *disabled=False*, *error=''*, *attributes={}*, *classes=''*)

Date pickers are used to make the input of dates streamlined and easy. Rather than typing the date, the user is presented with a calendar to select the date. This date picker was implemented using Bootstrap Datepicker.

name
Name of the input element that will be used for form submission.
Type str, required

display_text
Display text for the label that accompanies date picker.
Type str

autoclose
Set whether datepicker auto closes when a date is selected.
Type bool

calendar_weeks
Set whether calendar week numbers are shown on the left of the datepicker.
Type bool

clear_button
Set whether the clear button is displayed or not.
Type bool

days_of_week_disabled
Days of the week that are disabled 0-6 with 0 being Sunday and 6 being Saturday. Multiple days are comma separated (e.g.: '0,6').
Type str

end_date
Last date that can be selected. All other dates after this date are shown as disabled.
Type str

format
String representing date format. For valid formats see Bootstrap Datepicker documentation here.
Type str

min_view_mode
Set the minimum view mode. Possible values are 'days', 'months', 'years'.
Type str

multidate
Enables multi-selection of dates up to the number given.
Type int

start_date
First date that can be selected. All other dates before this date are shown as disabled.
Type str

start_view
View the date picker starts on. Valid values include 'month', 'year', and 'decade'.
Type str

today_button
Set whether a today button is displayed or not.
Type bool

today_highlight
Set whether to highlight the current date.
Type bool

week_start
Set the day the week starts on 0-6, where 0 is Sunday and 6 is Saturday.
Type int

initial
Initial date to appear in date picker.
Type str

disabled
Disabled state of the date picker.
Type bool

error
Error message for form validation.
Type str

attributes
A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).
Type dict

classes
Additional classes to add to the primary HTML element (e.g. "example-class another-class").
Type str

Controller Example

```
from tethys_sdk.gizmos import DatePicker

# Date Picker Options
date_picker = DatePicker(name='date1',
                         display_text='Date',
                         autoclose=True,
                         format='MM d, yyyy',
                         start_date='2/15/2014',
                         start_view='decade',
                         today_button=True,
                         initial='February 15, 2014')

date_picker_error = DatePicker(name='data2',
                               display_text='Date',
                               initial='10/2/2013',
                               disabled=True,
                               error='Here is my error text.')

context = {
            'date_picker': date_picker,
```

(continues on next page)

(continued from previous page)

```
            'date_picker_error': date_picker_error,
        }
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo date_picker %}
{% gizmo date_picker_error %}
```

Range Slider

Last Updated: August 10, 2015

class tethys_sdk.gizmos.**RangeSlider**(*name*, *min*, *max*, *initial*, *step*, *disabled=False*, *display_text=''*, *error=''*, *attributes={}*, *classes=''*)

Sliders can be used to request an input value from a range of possible values. A slider is configured with a dictionary of key-value options. The table below summarizes the options for sliders.

display_text
Display text for the label that accompanies slider
Type str

name
Name of the input element that will be used on form submission
Type str, required

min
Minimum value of range
Type int, required

max
Maximum value of range
Type int, required

initial
Initial value of slider
Type int, required

step
Increment between values in range
Type int, required

disabled
Disabled state of the slider
Type bool

error
Error message for form validation
Type str

attributes
A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).
Type dict

classes
Additional classes to add to the primary HTML element (e.g. "example-class another-class").

Type str

Example

```
from tethys_sdk.gizmos import RangeSlider

slider1 = RangeSlider(display_text='Slider 1',
                      name='slider1',
                      min=0,
                      max=100,
                      initial=50,
                      step=1)

slider2 = RangeSlider(display_text='Slider 2',
                      name='slider2',
                      min=0,
                      max=1,
                      initial=0.5,
                      step=0.1,
                      disabled=True,
                      error='Incorrect, please choose another value.')

context = {
            'slider1': slider1,
            'slider2': slider2,
          }
```

Template Example

```
{% gizmo slider1 %}
{% gizmo slider2 %}
```

Select Input

Last Updated: August 10, 2015

class tethys_sdk.gizmos.**SelectInput**(*name*, *display_text=''*, *initial=[]*, *multiple=False*, *original=False*, *select2_options=None*, *options=''*, *disabled=False*, *error=''*, *attributes={}*, *classes=''*)

Select inputs are used to select values from an given set of values. Use this gizmo to create select inputs and multi select inputs. This uses the Select2 functionality.

display_text
Display text for the label that accompanies select input
Type str

name
Name of the input element that will be used for form submission
Type str, required

multiple
If True, select input will be a multi-select
Type bool

original
If True, Select2 reference functionality will be turned off
Type bool

select2_options
Select2 options that will be passed when initializing the select2.
Type dict

options
List of tuples that represent the options and values of the select input
Type list

initial
List of keys or values that represent the initial selected values or a string representing a singular initial selected value.
Type list or str

disabled
Disabled state of the select input
Type bool

error
Error message for form validation
Type str

attributes
A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).
Type dict

classes
Additional classes to add to the primary HTML element (e.g. "example-class another-class").
Type str

Controller Example

```
from tethys_sdk.gizmos import SelectInput

select_input2 = SelectInput(display_text='Select2',
                            name='select2',
                            multiple=False,
                            options=[('One', '1'), ('Two', '2'), ('Three', '3')],
                            initial=['Three'],
                            select2_options={'placeholder': 'Select a number',
                                             'allowClear': True})

select_input2_multiple = SelectInput(display_text='Select2 Multiple',
                                     name='select21',
                                     multiple=True,
                                     options=[('One', '1'), ('Two', '2'), ('Three
↪', '3')],
                                     initial=['Two', 'One'])

select_input2_error = SelectInput(display_text='Select2 Disabled',
                                  name='select22',
                                  multiple=False,
                                  options=[('One', '1'), ('Two', '2'), ('Three',
↪'3')],
                                  disabled=True,
                                  error='Here is my error text')

select_input = SelectInput(display_text='Select',
                           name='select1',
```

(continues on next page)

(continued from previous page)

```
                            multiple=False,
                            original=True,
                            options=[('One', '1'), ('Two', '2'), ('Three', '3')],
                            initial=['Three'])

select_input_multiple = SelectInput(display_text='Select Multiple',
                                    name='select11',
                                    multiple=True,
                                    original=True,
                                    options=[('One', '1'), ('Two', '2'), ('Three',
↪ '3')])


context = {
            'select_input2': select_input2,
            'select_input2_multiple': select_input2_multiple,
            'select_input2_error': select_input2_error,
            'select_input': select_input,
            'select_input_multiple': select_input_multiple,
          }
```

Template Example

```
{% gizmo select_input2 %}
{% gizmo select_input2_multiple %}
{% gizmo select_input2_error %}
{% gizmo select_input %}
{% gizmo select_input_multiple %}
```

AJAX

Often dynamically loading in a select input can be useful. Here is a description of how to do so with the SelectInput gizmo.

Note: In order to use this, you will either need to use a `SelectInput` gizmo on the main page or register the dependencies in the main html template page using the `import_gizmo_dependency` tag with the `select_input` name in the `import_gizmos` block.

For example:

```
{% block import_gizmos %}
    {% import_gizmo_dependency select_input %}
{% endblock %}
```

Four elements are required:

1) A controller for the AJAX call with a SelectInput gizmo.

```
from tethys_sdk.gizmos import SelectInput

@login_required()
def select_input_ajax(request):
    """
```

(continues on next page)

(continued from previous page)

```
    Controller for the bokeh ajax request.
    """
    select_input2 = SelectInput(display_text='Select2',
                                name='select2',
                                multiple=False,
                                options=[('One', '1'), ('Two', '2'), ('Three', '3')],
                                initial=['Three'])

    context = {'select_input2': select_input2}

    return render(request, 'app_name/select_input_ajax.html', context)
```

2) A template for with the tethys gizmo (e.g. select_input_ajax.html)

```
{% load tethys_gizmos %}

{% gizmo select_input2 %}
```

3) A url map to the controller in app.py

```
...
    UrlMap(name='select_input_ajax',
           url='app_name/select',
           controller='app_name.controllers.select_input_ajax'),
...
```

4) The AJAX call in the javascript

Note: You only need to call the init function if you are using select2.

```
$(function() { //wait for page to load

    $.ajax({
        url: 'select',
        method: 'GET',
        success: function(data) {
            // add to page
            $("#select_div").html(data);
            // initialize if using select2
            $("#select_div").find('.select2').select2();
        }
    });

});
```

Text Input

Last Updated: August 10, 2015

class tethys_sdk.gizmos.**TextInput**(*name*, *display_text=''*, *initial=''*, *placeholder=''*, *prepend=''*, *append=''*, *icon_prepend=''*, *icon_append=''*, *disabled=False*, *error=''*, *attributes={}*, *classes=''*)

The text input gizmo makes it easy to add text inputs to your app that are styled similarly to the other input snippets.

display_text
Display text for the label that accompanies select input
Type str

name
Name of the input element that will be used for form submission
Type str, required

initial
The initial text that will appear in the text input when it loads
Type str

placeholder
Placeholder text is static text that displayed in the input when it is empty
Type str

prepend
Text that is prepended to the text input
Type str

append
Text that is appended to the text input
Type str

icon_prepend
The name of a valid Bootstrap v2.3 icon. The icon will be prepended to the input.
Type str

icon_append
The name of a valid Bootstrap v2.3 icon. The icon will be appended to the input.
Type str

disabled
Disabled state of the select input
Type bool

error
Error message for form validation
Type str

attributes
A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).
Type dict

classes
Additional classes to add to the primary HTML element (e.g. "example-class another-class").
Type str

Controller Example

```
from tethys_sdk.gizmos import TextInput

text_input = TextInput(display_text='Text',
                       name='inputAmount',
                       placeholder='e.g.: 10.00',
                       prepend='$')

text_error_input = TextInput(display_text='Text Error',
                             name='inputEmail',
                             initial='bob@example.com',
                             disabled=True,
                             icon_append='glyphicon glyphicon-envelope',
                             error='Here is my error text')

context = {
            'text_input': text_input,
            'text_error_input': text_error_input,
          }
```

Template Example

```
{% gizmo text_input %}
{% gizmo text_error_input %}
```

Toggle Switch

Last Updated: August 10, 2015

class tethys_sdk.gizmos.**ToggleSwitch**(*name*, *display_text=''*, *on_label='ON'*, *off_label='OFF'*, *on_style='primary'*, *off_style='default'*, *size='regular'*, *initial=False*, *disabled=False*, *error=''*, *attributes={}*, *classes=''*)

Toggle switches can be used as an alternative to check boxes for boolean or binomial input. Toggle switches are implemented using the excellent Bootstrap Switch reference project.

display_text
Display text for the label that accompanies switch
Type str

name
Name of the input element that will be used for form submission
Type str, required

on_label
Text that appears in the "on" position of the switch
Type str

off_label
Text that appears in the "off" position of the switch
Type str

on_style
Color of the "on" position. Either: 'default', 'info', 'primary', 'success', 'warning', or 'danger'
Type str

off_style
Color of the "off" position. Either: 'default', 'info', 'primary', 'success', 'warning', or 'danger'

Type str

size
Size of the switch. Either: 'large', 'small', or 'mini'.
Type str

initial
The initial position of the switch (True for "on" and False for "off")
Type bool

disabled
Disabled state of the switch
Type bool

error
Error message for form validation
Type str

attributes
A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).
Type dict

classes
Additional classes to add to the primary HTML element (e.g. "example-class another-class").
Type str

Controller Example

```
from tethys_sdk.gizmos import ToggleSwitch

toggle_switch = ToggleSwitch(display_text='Defualt Toggle',
                             name='toggle1')

toggle_switch_styled = ToggleSwitch(display_text='Styled Toggle',
                                    name='toggle2',
                                    on_label='Yes',
                                    off_label='No',
                                    on_style='success',
                                    off_style='danger',
                                    initial=True,
                                    size='large')

toggle_switch_disabled = ToggleSwitch(display_text='Disabled Toggle',
                                      name='toggle3',
                                      on_label='On',
                                      off_label='Off',
                                      on_style='success',
                                      off_style='warning',
                                      size='mini',
                                      initial=False,
                                      disabled=True,
                                      error='Here is my error text')

context = {
            'toggle_switch': toggle_switch,
            'toggle_switch_styled': toggle_switch_styled,
            'toggle_switch_disabled': toggle_switch_disabled,
          }
```

Template Example

```
{% gizmo toggle_switch %}
{% gizmo toggle_switch_styled %}
{% gizmo toggle_switch_disabled %}
```

AJAX

Often dynamically loading in the ToggleSwitch can be useful. Here is a description of how to do so with the ToggleSwitch gizmo.

Note: In order to use this, you will either need to use a `ToggleSwitch` gizmo on the main page or register the dependencies in the main html template page using the `import_gizmo_dependency` tag with the `toggle_switch` name in the `import_gizmos` block.

For example:

```
{% block import_gizmos %}
    {% import_gizmo_dependency toggle_switch %}
{% endblock %}
```

Four elements are required:

1) A controller for the AJAX call with a ToggleSwitch gizmo.

```
import json

@login_required()
def toggle_ajax(request):
    """
    Controller for the datatable ajax request.
    """

    toggle_switch = ToggleSwitch(display_text='Defualt Toggle',
                                 name='toggle1')

    context = {'toggle_switch': toggle_switch}

    return render(request, 'app_name/toggle_ajax.html', context)
```

2) A template for with the tethys gizmo (e.g. toggle_ajax.html)

```
{% load tethys_gizmos %}

{% gizmo toggle_switch %}
```

3) A url map to the controller in app.py

```
...
    UrlMap(name='toggle_ajax',
           url='app-name/toggle_ajax',
           controller='app_name.controllers.toggle_ajax'),
...
```

4) The AJAX call in the javascript

```
$(function() { //wait for page to load

    $.ajax({
        url: 'toggle_ajax',
        method: 'GET',
        success: function(data) {
            //add DataTable  to page
            $("#toggle_div").html(data);

            //Initialize DataTable
            $("#toggle_div").find('.bootstrap-switch').bootstrapSwitch();
         }
    });

});
```

Message Box

Last Updated: August 10, 2015

class tethys_sdk.gizmos.**MessageBox**(*name*, *title*, *message=''*, *dismiss_button='Cancel'*, *affirmative_button='Ok'*, *affirmative_attributes=''*, *width=560*, *attributes={}*, *classes=''*)

Message box gizmos can be used to display messages to users. These are especially useful for alerts and warning messages. The message box gizmo is implemented using Twitter Bootstrap's modal.

name
Unique name for the message box
Type str, required

title
Title that appears at the top of the message box
Type str, required

message
Message that will appear in the main body of the message box
Type str

dismiss_button
Title for the dismiss button (a.k.a.: the "Cancel" button)
Type str

affirmative_button
Title for the affirmative action button (a.k.a.: the "OK" button)
Type str

affirmative_attributes
Use this to place any html attributes on the affirmative button. (e.g.: 'href="/action" onclick="doSomething();"')
Type str

width
The width of the message box in pixels
Type int

attributes
A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).

Type dict

classes

Additional classes to add to the primary HTML element (e.g. "example-class another-class").

Type str

Controller Example

```
from tethys_sdk.gizmos import MessageBox

message_box = MessageBox(name='sampleModal',
                         title='Message Box Title',
                         message='Congratulations! This is a message box.',
                         dismiss_button='Nevermind',
                         affirmative_button='Proceed',
                         width=400,
                         affirmative_attributes='href=javascript:void(0);')

context = {
            'message_box': message_box,
          }
```

Template Example

```
{% load tethys_gizmos %}

<a href="#sampleModal" role="button" class="btn btn-success" data-toggle="modal">
↪Show Message Box</a>

{% block after_app_content %}
    {% gizmo message_box %}
{% endblock %}
```

Table View

Last Updated: August 10, 2015

class tethys_sdk.gizmos.**TableView**(*rows*, *column_names=''*, *hover=False*, *striped=False*, *bordered=False*, *condensed=False*, *editable_columns=''*, *row_ids=''*, *attributes={}*, *classes=''*)

Table views can be used to display tabular data. The table view gizmo can be configured to have columns that are editable. When used in this capacity, embed the table view in a form with a submit button.

rows

A list/tuple of lists/tuples representing each row in the table.

Type tuple or list, required

column_names

A tuple or list of strings that represent the table columns names.

Type tuple or list

hover

Illuminate rows on hover (does not work on striped tables)

Type bool

striped

Stripe rows

Type bool

bordered
Add borders and rounded corners
Type bool

condensed
A more tightly packed table
Type bool

editable_columns
A list or tuple with an entry for each column in the table. The entry is either False for non-editable columns or a string that will be used to create identifiers for the input column_fields in that column.
Type list or tuple

row_ids
A list or tuple of ids for each row in the table. These will be combined with the string in the editable_columns parameter to create unique identifiers for easy input field in the table. If not specified, each row will be assigned an integer value.
Type list or tuple

attributes
A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).
Type dict

classes
Additional classes to add to the primary HTML element (e.g. "example-class another-class").
Type str

Controller Example

```
from tethys_sdk.gizmos import TableView

table_view = TableView(column_names=('Name', 'Age', 'Job'),
                       rows=[('Bill', 30, 'contractor'),
                             ('Fred', 18, 'programmer'),
                             ('Bob', 26, 'boss')],
                       hover=True,
                       striped=False,
                       bordered=False,
                       condensed=False)

table_view_edit = TableView(column_names=('Name', 'Age', 'Job'),
                            rows=[('Bill', 30, 'contractor'),
                                  ('Fred', 18, 'programmer'),
                                  ('Bob', 26, 'boss')],
                            hover=True,
                            striped=True,
                            bordered=False,
                            condensed=False,
                            editable_columns=(False, 'ageInput', 'jobInput'),
                            row_ids=[21, 25, 31])

context = {
            'table_view': table_view,
            'table_view_edit': table_view_edit,
          }
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo table_view %}
{% gizmo table_view_edit %}
```

DataTable View

Last Updated: November 11, 2016

class tethys_sdk.gizmos.**DataTableView**(*rows*, *column_names*, *footer=False*, *attributes={}*, *classes=''*, ***kwargs*)

Table views can be used to display tabular data. The table view gizmo can be configured to have columns that are editable. When used in this capacity, embed the table view in a form with a submit button.

Note: The current version of DataTables in Tethys Platform is 1.10.12.

---

rows
A list/tuple of lists/tuples representing each row in the table.
Type tuple or list, required

column_names
A tuple or list of strings that represent the table columns names.
Type tuple or list

footer
If True, it will add the column names to the bottom of the table.
Type Optional[bool]

attributes
A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).
Type Optional[dict]

classes
Additional classes to add to the primary HTML element (e.g. "example-class another-class").
Type Optional[str]

****kwargs**
See https://datatables.net/reference/option.
Type DataTable Options

Regular Controller Example

```
from tethys_sdk.gizmos import DataTableView

datatable_default = DataTableView(column_names=('Name', 'Age', 'Job'),
                                  rows=[('Bill', 30, 'contractor'),
                                        ('Fred', 18, 'programmer'),
                                        ('Bob', 26, 'boss')],
                                  searching=False,
                                  orderClasses=False,
                                  lengthMenu=[ [10, 25, 50, -1], [10, 25, 50, "All
↪"] ],
                                  )

context = { 'datatable_view': datatable_default}
```

Regular Template Example

```
{% load tethys_gizmos %}

{% gizmo datatable_view %}
```

Note: You can also add extensions to the data table view as shown in the next example. To learn more about DataTable extensions, go to https://datatables.net/extensions/index.

ColReorder Controller Example

```
from tethys_sdk.gizmos import DataTableView

datatable_with_extension = DataTableView(column_names=('Name', 'Age', 'Job'),
                                         rows=[('Bill', 30, 'contractor'),
                                               ('Fred', 18, 'programmer'),
                                               ('Bob', 26, 'boss')],
                                         colReorder=True,
                                         )

context = { 'datatable_with_extension': datatable_with_extension}
```

ColReorder Template Example

```
{% load tethys_gizmos %}

#LOAD IN EXTENSION JAVASCRIPT/CSS
{% block global_scripts %}
  {{ block.super }}
   <script type="text/javascript" language="javascript" src="https://cdn.
↪datatables.net/colreorder/1.3.2/js/dataTables.colReorder.min.js"></script>
{% endblock %}

{% block styles %}
  {{ block.super }}
  <link rel="stylesheet" type="text/css" href="https://cdn.datatables.net/
↪colreorder/1.3.2/css/colReorder.dataTables.min.css">
{% endblock %}
#END LOAD IN EXTENSION JAVASCRIPT/CSS

{% gizmo datatable_with_extension %}
```

AJAX

Often dynamically loading in the DataTable can be useful. Here is a description of how to do so with the DataTableView gizmo.

Note: In order to use this, you will either need to use a `DataTableView` gizmo on the main page or register the dependencies in the main html template page using the `import_gizmo_dependency` tag with the `datatable_view` name in the `import_gizmos` block.

For example:

```
{% block import_gizmos %}
    {% import_gizmo_dependency datatable_view %}
{% endblock %}
```

Four elements are required:

1) A controller for the AJAX call with a DataTableView gizmo.

```
import json

@login_required()
def datatable_ajax(request):
    """
    Controller for the datatable ajax request.
    """

    searching = False
    if request.GET.get('searching') is not None:
        searching = json.loads(request.GET.get('searching'))
        if searching != True and searching != False:
            searching = False

    datatable_default = DataTableView(column_names=('Name', 'Age', 'Job'),
                                      rows=[('Bill', 30, 'contractor'),
                                            ('Fred', 18, 'programmer'),
                                            ('Bob', 26, 'boss')],
                                      searching=searching,
                                      orderClasses=False,
                                      lengthMenu=[ [10, 25, 50, -1], [10, 25, 50, "All
↪"] ],
                                      )

    context = {'datatable_options': datatable_default}

    return render(request, 'app_name/datatable_ajax.html', context)
```

2) A template for with the tethys gizmo (e.g. datatable_ajax.html)

```
{% load tethys_gizmos %}

{% gizmo datatable_options %}
```

3) A url map to the controller in app.py

```
...
    UrlMap(name='datatable_ajax',
           url='dam-break/datatable_ajax',
           controller='dam_break.controllers.datatable_ajax'),
...
```

4) The AJAX call in the javascript

```
$(function() { //wait for page to load

    $.ajax({
        url: 'datatable_ajax',
```

(continues on next page)

(continued from previous page)

```
        method: 'GET',
        data: {
            'searching': false, //example data to pass to the controller
        },
        success: function(data) {
            //add DataTable  to page
           $("#datatable_div").html(data);

            //Initialize DataTable
            $("#datatable_div").find('.data_table_gizmo_view').DataTable();
         }
    });

});
```

Plot View

Last Updated: August 10, 2015

Tethys Platform provides two interactive plotting engines: D3 and Highcharts. The Plot view options objects have been designed to be engine independent, meaning that you can configure a D3 plot using the same syntax as a Highcharts plot. This allows you to switch which plotting engine to use via configuration. This article describes each of the plot views that are available.

Warning: Highcharts is free-of-charge for certain applications (see: Highcharts JS Licensing). If you need a guaranteed fee-free solution, D3 is recommended.

Note: D3 plotting implemented for Line Plot, Pie Plot, Bar Plot, Scatter Plot, and Timeseries Plot.

Line Plot

class tethys_sdk.gizmos.**LinePlot**(*series*, *height='500px'*, *width='500px'*, *engine='d3'*, *title=''*, *subtitle=''*, *spline=False*, *x_axis_title=''*, *x_axis_units=''*, *y_axis_title=''*, *y_axis_units=''*, ***kwargs*)

Used to create line plot visualizations.

series
A list of series dictionaries.
Type list, required

height
Height of the plot element. Any valid css unit of length.
Type str

width
Width of the plot element. Any valid css unit of length.
Type str

engine
The plot engine to be used for rendering, either 'd3' or 'highcharts'. Defaults to 'd3'.
Type str

title
Title of the plot.
Type str

subtitle
Subtitle of the plot.
Type str

spline
If True, lines are smoothed using a spline technique.
Type bool

x_axis_title
Title of the x-axis.
Type str

x_axis_units
Units of the x-axis.
Type str

y_axis_title
Title of the y-axis.
Type str

y_axis_units
Units of the y-axis.
Type str

Controller Example

```
from tethys_sdk.gizmos import LinePlot

line_plot_view = LinePlot(
    height='500px',
    width='500px',
    engine='highcharts',
    title='Plot Title',
    subtitle='Plot Subtitle',
    spline=True,
    x_axis_title='Altitude',
    x_axis_units='km',
    y_axis_title='Temperature',
    y_axis_units='°C',
    series=[
       {
           'name': 'Air Temp',
           'color': '#0066ff',
           'marker': {'enabled': False},
           'data': [
               [0, 5], [10, -70],
               [20, -86.5], [30, -66.5],
               [40, -32.1],
               [50, -12.5], [60, -47.7],
               [70, -85.7], [80, -106.5]
           ]
       },
       {
           'name': 'Water Temp',
           'color': '#ff6600',
```

(continues on next page)

(continued from previous page)

```
            'data': [
                [0, 15], [10, -50],
                [20, -56.5], [30, -46.5],
                [40, -22.1],
                [50, -2.5], [60, -27.7],
                [70, -55.7], [80, -76.5]
            ]
        }
    ]
)

context = {
            'line_plot_view': line_plot_view,
          }
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo line_plot_view %}
```

Scatter Plot

class tethys_sdk.gizmos.**ScatterPlot**(*series=[]*, *height='500px'*, *width='500px'*, *engine='d3'*, *title=''*, *subtitle=''*, *x_axis_title=''*, *x_axis_units=''*, *y_axis_title=''*, *y_axis_units=''*, ***kwargs*)

Use to create a scatter plot visualization.

series
A list of series dictionaries.
Type list, required

height
Height of the plot element. Any valid css unit of length.
Type str

width
Width of the plot element. Any valid css unit of length.
Type str

engine
The plot engine to be used for rendering, either 'd3' or 'highcharts'. Defaults to 'd3'.
Type str

title
Title of the plot.
Type str

subtitle
Subtitle of the plot.
Type str

spline
If True, lines are smoothed using a spline technique.
Type bool

x_axis_title
Title of the x-axis.
Type str

x_axis_units
Units of the x-axis.
Type str

y_axis_title
Title of the y-axis.
Type str

y_axis_units
Units of the y-axis.
Type str

Controller Example

```
from tethys_sdk.gizmos import ScatterPlot

male_dataset = {
    'name': 'Male',
    'color': '#0066ff',
    'data': [
        [174.0, 65.6], [175.3, 71.8], [193.5, 80.7], [186.5, 72.6],
        [187.2, 78.8], [181.5, 74.8], [184.0, 86.4], [184.5, 78.4],
        [175.0, 62.0], [184.0, 81.6], [180.0, 76.6], [177.8, 83.6],
        [192.0, 90.0], [176.0, 74.6], [174.0, 71.0], [184.0, 79.6],
        [192.7, 93.8], [171.5, 70.0], [173.0, 72.4], [176.0, 85.9],
        [176.0, 78.8], [180.5, 77.8], [172.7, 66.2], [176.0, 86.4],
        [173.5, 81.8], [178.0, 89.6], [180.3, 82.8], [180.3, 76.4],
        [164.5, 63.2], [173.0, 60.9], [183.5, 74.8], [175.5, 70.0],
        [188.0, 72.4], [189.2, 84.1], [172.8, 69.1], [170.0, 59.5],
        [182.0, 67.2], [170.0, 61.3], [177.8, 68.6], [184.2, 80.1],
        [186.7, 87.8], [171.4, 84.7], [172.7, 73.4], [175.3, 72.1],
        [180.3, 82.6], [182.9, 88.7], [188.0, 84.1], [177.2, 94.1],
        [172.1, 74.9], [167.0, 59.1], [169.5, 75.6], [174.0, 86.2],
        [172.7, 75.3], [182.2, 87.1], [164.1, 55.2], [163.0, 57.0],
        [171.5, 61.4], [184.2, 76.8], [174.0, 86.8], [174.0, 72.2],
        [177.0, 71.6], [186.0, 84.8], [167.0, 68.2], [171.8, 66.1]
    ]
}

female_dataset = {
    'name': 'Female',
    'color': '#ff6600',
    'data': [
        [161.2, 51.6], [167.5, 59.0], [159.5, 49.2], [157.0, 63.0],
        [155.8, 53.6], [170.0, 59.0], [159.1, 47.6], [166.0, 69.8],
        [176.2, 66.8], [160.2, 75.2], [172.5, 55.2], [170.9, 54.2],
        [172.9, 62.5], [153.4, 42.0], [160.0, 50.0], [147.2, 49.8],
        [168.2, 49.2], [175.0, 73.2], [157.0, 47.8], [167.6, 68.8],
        [159.5, 50.6], [175.0, 82.5], [166.8, 57.2], [176.5, 87.8],
        [170.2, 72.8], [174.0, 54.5], [173.0, 59.8], [179.9, 67.3],
        [170.5, 67.8], [160.0, 47.0], [154.4, 46.2], [162.0, 55.0],
        [176.5, 83.0], [160.0, 54.4], [152.0, 45.8], [162.1, 53.6],
        [170.0, 73.2], [160.2, 52.1], [161.3, 67.9], [166.4, 56.6],
        [168.9, 62.3], [163.8, 58.5], [167.6, 54.5], [160.0, 50.2],
        [161.3, 60.3], [167.6, 58.3], [165.1, 56.2], [160.0, 50.2],
```

(continues on next page)

(continued from previous page)

```
        [170.0, 72.9], [157.5, 59.8], [167.6, 61.0], [160.7, 69.1],
        [163.2, 55.9], [152.4, 46.5], [157.5, 54.3], [168.3, 54.8],
        [180.3, 60.7], [165.5, 60.0], [165.0, 62.0], [164.5, 60.3]
    ]
}

scatter_plot_view = ScatterPlot(
    width='500px',
    height='500px',
    engine='highcharts',
    title='Scatter Plot',
    subtitle='Scatter Plot',
    x_axis_title='Height',
    x_axis_units='cm',
    y_axis_title='Weight',
    y_axis_units='kg',
    series=[
        male_dataset,
        female_dataset
    ]
)

context = {
            'scatter_plot_view': scatter_plot_view,
          }
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo scatter_plot_view %}
```

Polar Plot

class tethys_sdk.gizmos.**PolarPlot**(*series=[]*, *height='500px'*, *width='500px'*, *engine='d3'*, *title=''*, *subtitle=''*, *categories=[]*, ***kwargs*)

Use to create a polar plot visualization.

series
A list of series dictionaries.
Type list, required

height
Height of the plot element. Any valid css unit of length.
Type str

width
Width of the plot element. Any valid css unit of length.
Type str

engine
The plot engine to be used for rendering, either 'd3' or 'highcharts'. Defaults to 'd3'.
Type str

title
Title of the plot.

Type str

subtitle
Subtitle of the plot.
Type str

categories
List of category names, one for each data point in the series.
Type list

Controller Example

```
from tethys_sdk.gizmos import PolarPlot

web_plot = PolarPlot(
    height='500px',
    width='500px',
    engine='highcharts',
    title='Polar Chart',
    subtitle='Polar Chart',
    pane={
      'size': '80%'
    },
    categories=['Infiltration', 'Soil Moisture', 'Precipitation', 'Evaporation',
                'Roughness', 'Runoff', 'Permeability', 'Vegetation'],
    series=[
      {
          'name': 'Park City',
          'data': [0.2, 0.5, 0.1, 0.8, 0.2, 0.6, 0.8, 0.3],
          'pointPlacement': 'on'
      },
      {
          'name': 'Little Dell',
          'data': [0.8, 0.3, 0.2, 0.5, 0.1, 0.8, 0.2, 0.6],
          'pointPlacement': 'on'
      }
    ]
)

context = {
            'web_plot': web_plot,
          }
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo web_plot %}
```

Pie Plot

class tethys_sdk.gizmos.**PiePlot**(*series=[]*, *height='500px'*, *width='500px'*, *engine='d3'*, *title=''*, *subtitle=''*, ***kwargs*)

Use to create a pie plot visualization.

series
A list of series dictionaries.
Type list, required

height
Height of the plot element. Any valid css unit of length.
Type str

width
Width of the plot element. Any valid css unit of length.
Type str

engine
The plot engine to be used for rendering, either 'd3' or 'highcharts'. Defaults to 'd3'.
Type str

title
Title of the plot.
Type str

subtitle
Subtitle of the plot.
Type str

Controller Example

```
from tethys_sdk.gizmos import PieChart

pie_plot_view = PiePlot(
    height='500px',
    width='500px',
    engine='highcharts',
    title='Pie Chart',
    subtitle='Pie Chart',
    series=[
          {'name': 'Firefox', 'value': 45.0},
          {'name': 'IE', 'value': 26.8},
          {'name': 'Chrome', 'value': 12.8},
          {'name': 'Safari', 'value': 8.5},
          {'name': 'Opera', 'value': 8.5},
          {'name': 'Others', 'value': 0.7}
    ]
)

context = {
            'pie_plot_view': pie_plot_view,
          }
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo pie_plot_view %}
```

Bar Plot

class tethys_sdk.gizmos.**BarPlot**(*series=[]*, *height='500px'*, *width='500px'*, *engine='d3'*, *title=''*, *subtitle=''*, *horizontal=False*, *categories=[]*, *axis_title=''*, *axis_units=''*, *group_tools=True*, *y_min=0*, ***kwargs*)

Bar Plot

Displays as either a bar or column chart.

series
A list of series dictionaries.
Type list, required

height
Height of the plot element. Any valid css unit of length.
Type str

width
Width of the plot element. Any valid css unit of length.
Type str

engine
The plot engine to be used for rendering, either 'd3' or 'highcharts'. Defaults to 'd3'.
Type str

title
Title of the plot.
Type str

subtitle
Subtitle of the plot.
Type str

horizontal
If True, bars are displayed horizontally, otherwise they are displayed vertically.
Type bool

categories
A list of category titles, one for each bar.
Type list

axis_title
Title of the axis.
Type str

axis_units
Units of the axis.
Type str

y_min
Minimum value of y axis.
Type int,float

Controller Example

```
from tethys_sdk.gizmos import BarPlot

bar_plot_view = BarPlot(
    height='500px',
    width='500px',
```

(continues on next page)

(continued from previous page)

```
    engine='highcharts',
    title='Bar Chart',
    subtitle='Bar Chart',
    vertical=True,
    categories=[
        'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov
↪', 'Dec'
    ],
    axis_units='millions',
    axis_title='Population',
    series=[{
            'name': "Year 1800",
            'data': [100, 31, 635, 203, 275, 487, 872, 671, 736, 568, 487, 432]
        }, {
            'name': "Year 1900",
            'data': [133, 200, 947, 408, 682, 328, 917, 171, 482, 140, 176, 237]
        }, {
            'name': "Year 2000",
            'data': [764, 628, 300, 134, 678, 200, 781, 571, 773, 192, 836, 172]
        }, {
            'name': "Year 2008",
            'data': [973, 914, 500, 400, 349, 108, 372, 726, 638, 927, 621, 364]
        }
    ]
)

context = {
            'bar_plot_view': bar_plot_view,
          }
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo bar_plot_view %}
```

Time Series

class tethys_sdk.gizmos.**TimeSeries**(*series=[]*, *height='500px'*, *width='500px'*, *engine='d3'*, *title=''*, *subtitle=''*, *y_axis_title=''*, *y_axis_units=''*, *y_min=0*, ***kwargs*)

Use to create a timeseries plot visualization

series
A list of series dictionaries.
Type list, required

height
Height of the plot element. Any valid css unit of length.
Type str

width
Width of the plot element. Any valid css unit of length.
Type str

engine
The plot engine to be used for rendering, either 'd3' or 'highcharts'. Defaults to 'd3'.
Type str

title
Title of the plot.
Type str

subtitle
Subtitle of the plot.
Type str

y_axis_title
Title of the axis.
Type str

y_axis_units
Units of the axis.
Type str

Controller Example

```
from tethys_sdk.gizmos import TimeSeries

timeseries_plot = TimeSeries(
    height='500px',
    width='500px',
    engine='highcharts',
    title='Irregular Timeseries Plot',
    y_axis_title='Snow depth',
    y_axis_units='m',
    series=[{
        'name': 'Winter 2007-2008',
        'data': [
            [datetime(2008, 12, 2), 0.8],
            [datetime(2008, 12, 9), 0.6],
            [datetime(2008, 12, 16), 0.6],
            [datetime(2008, 12, 28), 0.67],
            [datetime(2009, 1, 1), 0.81],
            [datetime(2009, 1, 8), 0.78],
            [datetime(2009, 1, 12), 0.98],
            [datetime(2009, 1, 27), 1.84],
            [datetime(2009, 2, 10), 1.80],
            [datetime(2009, 2, 18), 1.80],
            [datetime(2009, 2, 24), 1.92],
            [datetime(2009, 3, 4), 2.49],
            [datetime(2009, 3, 11), 2.79],
            [datetime(2009, 3, 15), 2.73],
            [datetime(2009, 3, 25), 2.61],
            [datetime(2009, 4, 2), 2.76],
            [datetime(2009, 4, 6), 2.82],
            [datetime(2009, 4, 13), 2.8],
            [datetime(2009, 5, 3), 2.1],
            [datetime(2009, 5, 26), 1.1],
            [datetime(2009, 6, 9), 0.25],
            [datetime(2009, 6, 12), 0]
        ]
    }]
)
```

(continues on next page)

(continued from previous page)

```
context = {
            'timeseries_plot': timeseries_plot,
          }
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo timeseries_plot %}
```

Area Range

class tethys_sdk.gizmos.**AreaRange**(*series=[]*, *height='500px'*, *width='500px'*, *engine='d3'*, *title=''*, *subtitle=''*, *y_axis_title=''*, *y_axis_units=''*, ***kwargs*)

Use to create a area range plot visualization.

series
A list of series dictionaries.
Type list, required

height
Height of the plot element. Any valid css unit of length.
Type str

width
Width of the plot element. Any valid css unit of length.
Type str

engine
The plot engine to be used for rendering, either 'd3' or 'highcharts'. Defaults to 'd3'.
Type str

title
Title of the plot.
Type str

subtitle
Subtitle of the plot.
Type str

y_axis_title
Title of the axis.
Type str

y_axis_units
Units of the axis.
Type str

Controller Example

```
from tethys_sdk.gizmos import AreaRange

averages = [
    [datetime(2009, 7, 1), 21.5], [datetime(2009, 7, 2), 22.1], [datetime(2009, 7,
↪ 3), 23],
```

(continues on next page)

(continued from previous page)

```
    [datetime(2009, 7, 4), 23.8], [datetime(2009, 7, 5), 21.4], [datetime(2009, 7,
↪ 6), 21.3],
    [datetime(2009, 7, 7), 18.3], [datetime(2009, 7, 8), 15.4], [datetime(2009, 7,
↪ 9), 16.4],
    [datetime(2009, 7, 10), 17.7], [datetime(2009, 7, 11), 17.5], [datetime(2009,␣
↪7, 12), 17.6],
    [datetime(2009, 7, 13), 17.7], [datetime(2009, 7, 14), 16.8], [datetime(2009,␣
↪7, 15), 17.7],
    [datetime(2009, 7, 16), 16.3], [datetime(2009, 7, 17), 17.8], [datetime(2009,␣
↪7, 18), 18.1],
    [datetime(2009, 7, 19), 17.2], [datetime(2009, 7, 20), 14.4],
    [datetime(2009, 7, 21), 13.7], [datetime(2009, 7, 22), 15.7], [datetime(2009,␣
↪7, 23), 14.6],
    [datetime(2009, 7, 24), 15.3], [datetime(2009, 7, 25), 15.3], [datetime(2009,␣
↪7, 26), 15.8],
    [datetime(2009, 7, 27), 15.2], [datetime(2009, 7, 28), 14.8], [datetime(2009,␣
↪7, 29), 14.4],
    [datetime(2009, 7, 30), 15], [datetime(2009, 7, 31), 13.6]
]

ranges = [
    [datetime(2009, 7, 1), 14.3, 27.7], [datetime(2009, 7, 2), 14.5, 27.8],␣
↪[datetime(2009, 7, 3), 15.5, 29.6],
    [datetime(2009, 7, 4), 16.7, 30.7], [datetime(2009, 7, 5), 16.5, 25.0],␣
↪[datetime(2009, 7, 6), 17.8, 25.7],
    [datetime(2009, 7, 7), 13.5, 24.8], [datetime(2009, 7, 8), 10.5, 21.4],␣
↪[datetime(2009, 7, 9), 9.2, 23.8],
    [datetime(2009, 7, 10), 11.6, 21.8], [datetime(2009, 7, 11), 10.7, 23.7],␣
↪[datetime(2009, 7, 12), 11.0, 23.3],
    [datetime(2009, 7, 13), 11.6, 23.7], [datetime(2009, 7, 14), 11.8, 20.7],␣
↪[datetime(2009, 7, 15), 12.6, 22.4],
    [datetime(2009, 7, 16), 13.6, 19.6], [datetime(2009, 7, 17), 11.4, 22.6],␣
↪[datetime(2009, 7, 18), 13.2, 25.0],
    [datetime(2009, 7, 19), 14.2, 21.6], [datetime(2009, 7, 20), 13.1, 17.1],␣
↪[datetime(2009, 7, 21), 12.2, 15.5],
    [datetime(2009, 7, 22), 12.0, 20.8], [datetime(2009, 7, 23), 12.0, 17.1],␣
↪[datetime(2009, 7, 24), 12.7, 18.3],
    [datetime(2009, 7, 25), 12.4, 19.4], [datetime(2009, 7, 26), 12.6, 19.9],␣
↪[datetime(2009, 7, 27), 11.9, 20.2],
    [datetime(2009, 7, 28), 11.0, 19.3], [datetime(2009, 7, 29), 10.8, 17.8],␣
↪[datetime(2009, 7, 30), 11.8, 18.5],
    [datetime(2009, 7, 31), 10.8, 16.1]
]

area_range_plot_object = AreaRange(
    title='July Temperatures',
    y_axis_title='Temperature',
    y_axis_units='*C',
    width='500px',
    height='500px',
    series=[{
        'name': 'Temperature',
        'data': averages,
        'zIndex': 1,
        'marker': {
            'lineWidth': 2,
        }
```

(continues on next page)

(continued from previous page)

```
    }, {
        'name': 'Range',
        'data': ranges,
        'type': 'arearange',
        'lineWidth': 0,
        'linkedTo': ':previous',
        'fillOpacity': 0.3,
        'zIndex': 0
    }]
)


context = {
            'area_range_plot_object': area_range_plot_object,
          }
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo area_range_plot_object %}
```

JavaScript API

For advanced features, the JavaScript API can be used to interact with the HighCharts object that is generated by the Plot View JavaScript library.

TETHYS_PLOT_VIEW.initHighChartsPlot(jquery_element)

This method initializes a chart generated from an AJAX request. An example is demonstrated in the Dam Break javascript tutorial.

Note: In order to use this, you will either need to use a `PlotView` gizmo on the main page or register the dependencies in the main html template page using the `import_gizmo_dependency` tag with the `plot_view` name in the `import_gizmos` block.

For example:

```
{% block import_gizmos %}
    {% import_gizmo_dependency plot_view %}
{% endblock %}
```

Four elements are required:

1) A controller for the AJAX call with a plot view gizmo.

```
@login_required()
def hydrograph_ajax(request):
    """
    Controller for the hydrograph ajax request.
    """
```

(continues on next page)

(continued from previous page)

```
    # add a plot view gizmo
    flood_plot = TimeSeries(
                    ...
    )

    context = {'flood_plot': flood_plot}

    return render(request, 'dam_break/hydrograph_ajax.html', context)
```

2) A template for with the tethys gizmo (e.g. hydrograph_ajax.html)

```
{% load tethys_gizmos %}

{% gizmo flood_plot %}
```

3) A url map to the controller in app.py

```
...
    UrlMap(name='hydrograph_ajax',
           url='dam-break/map/hydrograph',
           controller='dam_break.controllers.hydrograph_ajax'),
...
```

4) The AJAX call in the javascript

```
$(function() { //wait for page to load

    $.ajax({
        url: 'hydrograph',
        method: 'GET',
        data: {
            'peak_flow': 500, //example data to pass to the controller
        },
        success: function(data) {
            // add plot to page
            $("#plot_view_div").html(data);

            //Initialize Plot
            TETHYS_PLOT_VIEW.initHighChartsPlot($('.highcharts-plot'));
        }
    });

});
```

Highcharts JavaScript API

The Highcharts plots can be modified via JavaScript by using jQuery to select the Highcharts div and calling the `highcharts()` method on it. This will return the JavaScript object that represents the plot, which can be modified using the Highcharts API.

```
var plot = $('#my-plot').highcharts();
```

Plotly View

Last Updated: November 11, 2016

class tethys_sdk.gizmos.**PlotlyView**(*plot_input*, *height='520px'*, *width='100%'*, *attributes=''*, *classes=''*, *divid=''*, *hidden=False*, *show_link=False*)

Simple options object for plotly view.

Note: Information about the Plotly API can be found at https://plot.ly/python.

---

plot_input
A plotly graph_objs to be plotted.
Type plotly graph_objs

height
Height of the plot element. Any valid css unit of length.
Type Optional[str]

width
Width of the plot element. Any valid css unit of length.
Type Optional[str]

attributes
Dictionary of attributed to add to the outer div.
Type Optional[dict]

classes
Space separated string of classes to add to the outer div.
Type Optional[str]

hidden
If True, the plot will be hidden. Default is False.
Type Optional[bool]

show_link
If True, the link to export plot to view in plotly is shown. Default is False.
Type Optional[bool]

Controller Code Basic Example:

```
from datetime import datetime
import plotly.graph_objs as go
from tethys_sdk.gizmos import PlotlyView

x = [datetime(year=2013, month=10, day=04),
     datetime(year=2013, month=11, day=05),
     datetime(year=2013, month=12, day=06)]

my_plotly_view = PlotlyView([go.Scatter(x=x, y=[1, 3, 6])])

context = {'plotly_view_input': my_plotly_view}
```

Controller Code Pandas Example:

```
import numpy as np
import pandas as pd
from tethys_sdk.gizmos import PlotlyView
```

(continues on next page)

(continued from previous page)

```
df = pd.DataFrame(np.random.randn(1000, 2), columns=['A', 'B']).cumsum()

my_plotly_view = PlotlyView(df.iplot(asFigure=True))

context = {'plotly_view_input': my_plotly_view}
```

Template Code:

```
{% load tethys_gizmos %}

{% gizmo plotly_view_input %}
```

AJAX

Often dynamically loading in plots can be useful. Here is a description of how to do so with PlotlyView.

Note: In order to use this, you will either need to use a `PlotlyView` gizmo on the main page or register the dependencies in the main html template page using the `import_gizmo_dependency` tag with the `plotly_view` name in the `import_gizmos` block.

For example:

```
{% block import_gizmos %}
    {% import_gizmo_dependency plotly_view %}
{% endblock %}
```

Four elements are required:

1) A controller for the AJAX call with a PlotlyView gizmo.

```
from datetime import datetime
import plotly.graph_objs as go
from tethys_sdk.gizmos import PlotlyView

@login_required()
def plotly_ajax(request):
    """
    Controller for the plotly ajax request.
    """
    x = [datetime(year=2013, month=10, day=04),
         datetime(year=2013, month=11, day=05),
         datetime(year=2013, month=12, day=06)]

    my_plotly_view = PlotlyView([go.Scatter(x=x, y=[1, 3, 6])])

    context = {'plotly_view_input': my_plotly_view}

    return render(request, 'app_name/plotly_ajax.html', context)
```

2) A template for with the tethys gizmo (e.g. plotly_ajax.html)

```
{% load tethys_gizmos %}

{% gizmo plotly_view_input %}
```

3) A url map to the controller in app.py

```
...
    UrlMap(name='plotly_ajax',
           url='app_name/plotly',
           controller='app_name.controllers.plotly_ajax'),
...
```

4) The AJAX call in the javascript

```
$(function() { //wait for page to load

    $.ajax({
        url: 'plotly',
        method: 'GET',
        data: {
            'plot_height': 500, //example data to pass to the controller
        },
        success: function(data) {
            // add plot to page
            $("#plotly_plot_div").html(data);

            //NOTE: IF USING MODAL/MESSAGE BOX NEED A TIMEOUT BEFORE DISPLAY
            //$('#modal_div').modal('show');
            //setTimeout(function(){
            //    $("#modal_div").find('.modal-body').html(data);
            //}, 100);

        }
    });

});
```

Bokeh View

Last Updated: November 11, 2016

class tethys_sdk.gizmos.**BokehView**(*plot_input*, *height='520px'*, *width='100%'*, *attributes=''*, *classes=''*, *divid=''*, *hidden=False*)

Simple options object for Bokeh plotting.

Note: For more information about Bokeh and for Python examples, see http://bokeh.pydata.org.

---

plot_input
A bokeh figure to be plotted.
Type bokeh figure

height
Height of the plot element. Any valid css unit of length.
Type Optional[str]

width
Width of the plot element. Any valid css unit of length.
Type Optional[str]

attributes
Dictionary of attributed to add to the outer div.
Type Optional[dict]

classes
Space separated string of classes to add to the outer div.
Type Optional[str]

hidden
If True, the plot will be hidden. Default is False.
Type Optional[bool]

Controller Code Example:

```
from tethys_sdk.gizmos import BokehView
from bokeh.plotting import figure

plot = figure(plot_height=300)
plot.circle([1,2], [3,4])
my_bokeh_view = BokehView(plot, height="300px")

context = {'bokeh_view_input': my_bokeh_view}
```

Template Code Example:

```
{% load tethys_gizmos %}

{% gizmo bokeh_view_input %}
```

AJAX

Often dynamically loading in plots can be useful. Here is a description of how to do so with Bokeh.

Note: In order to use this, you will either need to use a `BokehView` gizmo on the main page or register the dependencies in the main html template page using the `import_gizmo_dependency` tag with the `bokeh_view` name in the `import_gizmos` block.

For example:

```
{% block import_gizmos %}
    {% import_gizmo_dependency bokeh_view %}
{% endblock %}
```

Four elements are required:

1) A controller for the AJAX call with a BokehView gizmo.

```
from tethys_sdk.gizmos import BokehView
from bokeh.plotting import figure

@login_required()
def bokeh_ajax(request):
    """
    Controller for the bokeh ajax request.
    """
```

(continues on next page)

(continued from previous page)

```
    plot = figure(plot_height=300)
    plot.circle([1,2], [3,4])
    my_bokeh_view = BokehView(plot, height="300px")

    context = {'bokeh_view_input': my_bokeh_view}

    return render(request, 'app_name/bokeh_ajax.html', context)
```

2) A template for with the tethys gizmo (e.g. bokeh_ajax.html)

```
{% load tethys_gizmos %}

{% gizmo bokeh_view_input %}
```

3) A url map to the controller in app.py

```
...
    UrlMap(name='bokeh_ajax',
           url='app_name/bokeh',
           controller='app_name.controllers.bokeh_ajax'),
...
```

4) The AJAX call in the javascript

```
$(function() { //wait for page to load

    $.ajax({
        url: 'bokeh',
        method: 'GET',
        data: {
            'plot_height': 500, //example data to pass to the controller
        },
        success: function(data) {
            // add plot to page
            $("#bokeh_plot_div").html(data);
        }
    });

});
```

Map View

class tethys_sdk.gizmos.**MapView**(*height='100%', width='100%', basemap=None, view={'center': [-100, 40], 'zoom': 2}, controls=[], layers=[], draw=None, legend=False, attributes={}, classes='', disable_basemap=False, feature_selection=None*)

The Map View gizmo can be used to produce interactive maps of spatial data. It is powered by OpenLayers, a free and open source pure javascript mapping library. It supports layers in a variety of different formats including WMS, Tiled WMS, GeoJSON, KML, and ArcGIS REST. It includes drawing capabilities and the ability to create a legend for the layers included in the map.

Shapes that are drawn on the map by users can be retrieved from the map via a hidden text field named 'geometry' and it is updated every time the map is changed. The text in the text field is a string representation of JSON. The geometry definition contained in this JSON can be formatted as either GeoJSON or Well Known Text. This can be configured via the output_format option of the MVDraw object. If the Map View is embedded in a form, the

geometry that is drawn on the map will automatically be submitted with the rest of the form via the hidden text field.

height
Height of the map element. Any valid css unit of length (e.g.: '500px'). Defaults to '520px'.
Type str

width
Width of the map element. Any valid css unit of length (e.g.: '100%'). Defaults to '100%'.
Type str

basemap
The base maps to display: choose from OpenStreetMap, MapQuest, or a Bing map. Valid values for the string option are: 'OpenStreetMap' and 'MapQuest'. If you wish to configure the base map with options, you must use the dictionary form. The dictionary form is required to use a Bing map, because an API key must be passed as an option. See below for more detail. A basemap switcher will automatically be provided if a list of more than one basemap is included.
Type str, dict or a list of strings and/or dicts

view
An MVView object specifying the initial view or extent for the map.
Type *MVView*

controls
A list of controls to add to the map. The list can be a list of strings or a list of dictionaries. Valid controls are ZoomSlider, Rotate, FullScreen, ScaleLine, ZoomToExtent, and 'MousePosition'. See below for more detail.
Type list

layers
A list of MVLayer objects.
Type list

draw
An MVDraw object specifying the drawing options.
Type *MVDraw*

disable_basemap
Render the map without a base map.
Type bool

feature_selection
A dictionary of global feature selection options. See below.
Type bool

attributes
A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).
Type dict

classes
Additional classes to add to the primary HTML element (e.g. "example-class another-class").
Type str

Options Dictionaries

Many of the options above will accept dictionaries with additional options. These dictionaries should be structured with a single key that is the name of the original option with a value of another dictionary containing the additional options. For example, to provide additional options for the 'ZoomToExtent' control, you would create a dictionary with key 'ZoomToExtent' and value of a dictionary with the additional options like this:

```
{'ZoomToExtent': {'projection': 'EPSG:4326', 'extent': [-135, 22, -55, 54]}}
```

Most of the additional options correspond with the options objects in the OpenLayers API. The following sections provide links to the OpenLayers objects that you can refer to when selecting the options.

Base Maps

There are several base maps supported by the Map View gizmo: *OpenStreetMap*, *Bing*, *Stamen*, *CartoDB*, and *ESRI*. All base maps can be specified as a string or as an options dictionary. When using an options dictionary all base maps map services accept the option *control_label*, which is used to specify the label to be used in the Base Map control. For example:

```
{'Bing': {'key': 'Ap|k3yheRE', 'imagerySet': 'Aerial', 'control_label': 'Bing␣
↪Aerial'}}
```

For additional options that can be provided to each base map service see the following links:

- OpenStreetMap: ol/source/OSM
- Bing: ol/source/BingMaps
- Stamen: ol/source/Stamen
- XYZ ol/source/XYZ

Note: The CartoDB and ESRI services are just pre-defined instances of the XYZ service. In addition to the standard XYZ options they have the following additional options:

CartoDB:

- *style*: The style of map. Possibilities are 'light' or 'dark'.
- *labels*: Boolean specifying whether or not to include labels.

ESRI:

- ***layer*: A string specifying which ESRI map to use. Possibilities are:**
  - NatGeo_World_Map
  - Ocean_Basemap
  - USA_Topo_Maps
  - World_Imagery
  - World_Physical_Map
  - World_Shaded_Relief
  - World_Street_Map
  - World_Terrain_Base
  - World_Topo_Map

Controls

Use the following links to learn about options for the different controls:

- FullScreen: ol.control.FullScreen
- MousePosition: ol.control.MousePosition
- Rotate: ol.control.Rotate
- ScaleLine: ol.control.ScaleLine
- ZoomSlider: ol.control.ZoomSlider
- ZoomToExtent: ol.control.ZoomToExtent

Feature Selection

The feature_selection dictionary contains global settings that can be used to modify the behavior of the feature selection functionality. An explanation of valid options follows:

- multiselect: Set to True to allow multiple features to be selected while holding the shift key on the keyboard. Defaults to False.
- sensitivity: Integer value that adjust the feature selection sensitivity. Defaults to 2.

Tip: OpenLayers Version

Currently, OpenLayers version 5.3.0 is used by default with the Map View gizmo. If you need a specific version of OpenLayers you can specify the version number using the *ol_version* class attribute on the *MapView* class:

```
MapView.ol_version = '4.6.5'
```

Any versions that are provided by https://www.jsdelivr.com/package/npm/openlayers can be specified.

Controller Example

```
from tethys_sdk.gizmos import MapView, MVDraw, MVView, MVLayer, MVLegendClass

# Define view options
view_options = MVView(
    projection='EPSG:4326',
    center=[-100, 40],
    zoom=3.5,
    maxZoom=18,
    minZoom=2
)

# Define drawing options
drawing_options = MVDraw(
    controls=['Modify', 'Delete', 'Move', 'Point', 'LineString', 'Polygon', 'Box
↪'],
    initial='Point',
    output_format='WKT'
)

# Define GeoJSON layer
geojson_object = {
  'type': 'FeatureCollection',
  'crs': {
    'type': 'name',
    'properties': {
      'name': 'EPSG:3857'
    }
  },
  'features': [
    {
      'type': 'Feature',
      'geometry': {
        'type': 'Point',
        'coordinates': [0, 0]
      }
    },
    {
      'type': 'Feature',
      'geometry': {
        'type': 'LineString',
```

(continues on next page)

(continued from previous page)

```
        'coordinates': [[4e6, -2e6], [8e6, 2e6]]
      }
    },
    {
      'type': 'Feature',
      'geometry': {
        'type': 'Polygon',
        'coordinates': [[[-5e6, -1e6], [-4e6, 1e6], [-3e6, -1e6]]]
      }
    }
  ]
}

# Define GeoJSON point layer
geojson_point_object = {
  'type': 'FeatureCollection',
  'crs': {
    'type': 'name',
    'properties': {
      'name': 'EPSG:3857'
    }
  },
  'features': [
    {
      'type': 'Feature',
      'geometry': {
        'type': 'Point',
        'coordinates': [0, 0]
      }
    },
  ]
}

geojson_layer = MVLayer(
    source='GeoJSON',
    options=geojson_object,
    legend_title='Test GeoJSON',
    legend_extent=[-46.7, -48.5, 74, 59],
    legend_classes=[
        MVLegendClass('polygon', 'Polygons', fill='rgba(255,255,255,0.8)', stroke=
↪'#3d9dcd'),
        MVLegendClass('line', 'Lines', stroke='#3d9dcd')
    ]
)

geojson_point_layer = MVLayer(
    source='GeoJSON',
    options=geojson_point_object,
    legend_title='Test GeoJSON',
    legend_extent=[-46.7, -48.5, 74, 59],
    legend_classes=[
        MVLegendClass('line', 'Lines', stroke='#3d9dcd')
    ],
    layer_options={
        'style': {
            'image': {
                'circle': {
```

(continues on next page)

(continued from previous page)

```
                    'radius': 10,
                    'fill': {'color':  '#d84e1f'},
                    'stroke': {'color': '#ffffff', 'width': 1},
                }
            }
        }
    }
)

# Define GeoServer Layer
geoserver_layer = MVLayer(
    source='ImageWMS',
    options={'url': 'http://192.168.59.103:8181/geoserver/wms',
           'params': {'LAYERS': 'topp:states'},
           'serverType': 'geoserver'},
    legend_title='USA Population',
    legend_extent=[-126, 24.5, -66.2, 49],
    legend_classes=[
        MVLegendClass('polygon', 'Low Density', fill='#00ff00', stroke='#000000'),
        MVLegendClass('polygon', 'Medium Density', fill='#ff0000', stroke='#000000
↪'),
        MVLegendClass('polygon', 'High Density', fill='#0000ff', stroke='#000000')
    ]
)

# Define KML Layer
kml_layer = MVLayer(
    source='KML',
    options={'url': '/static/tethys_gizmos/data/model.kml'},
    legend_title='Park City Watershed',
    legend_extent=[-111.60, 40.57, -111.43, 40.70],
    legend_classes=[
        MVLegendClass('polygon', 'Watershed Boundary', fill='#ff8000'),
        MVLegendClass('line', 'Stream Network', stroke='#0000ff'),
    ]
)

# Tiled ArcGIS REST Layer
arc_gis_layer = MVLayer(
    source='TileArcGISRest',
    options={'url': 'http://sampleserver1.arcgisonline.com/ArcGIS/rest/services/'␣
↪+ 'Specialty/ESRI_StateCityHighway_USA/MapServer'},
    legend_title='ESRI USA Highway',
    legend_extent=[-173, 17, -65, 72]
)

# Define base map options
esri_layer_names = [
    'NatGeo_World_Map',
    'Ocean_Basemap',
    'USA_Topo_Maps',
    'World_Imagery',
    'World_Physical_Map',
    'World_Shaded_Relief',
    'World_Street_Map',
    'World_Terrain_Base',
    'World_Topo_Map',
```

(continues on next page)

(continued from previous page)

```
]
esri_layers = [{'ESRI': {'layer': l}} for l in esri_layer_names]
basemaps = [
    'Stamen',
    {'Stamen': {'layer': 'toner', 'control_label': 'Black and White'}},
    {'Stamen': {'layer': 'watercolor'}},
    'OpenStreetMap',
    'CartoDB',
    {'CartoDB': {'style': 'dark'}},
    {'CartoDB': {'style': 'light', 'labels': False, 'control_label': 'CartoDB-
↪light-no-labels'}},
    {'XYZ': {'url': 'https://maps.wikimedia.org/osm-intl/{z}/{x}/{y}.png',
↪'control_label': 'Wikimedia'}}
    'ESRI',
]
basemaps.extend(esri_layers)

# Specify OpenLayers version
MapView.ol_version = '5.3.0'

# Define map view options
map_view_options = MapView(
        height='600px',
        width='100%',
        controls=['ZoomSlider', 'Rotate', 'FullScreen',
                  {'MousePosition': {'projection': 'EPSG:4326'}},
                  {'ZoomToExtent': {'projection': 'EPSG:4326', 'extent': [-130,␣
↪22, -65, 54]}}],
        layers=[geojson_layer, geojson_point_layer, geoserver_layer, kml_layer,␣
↪arc_gis_layer],
        view=view_options,
        basemap=basemaps,
        draw=drawing_options,
        legend=True
)

context = {'map_view_options': map_view_options}
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo map_view_options %}
```

MVLayer

class tethys_sdk.gizmos.**MVLayer**(*source*, *options*, *legend_title*, *layer_options=None*, *editable=True*, *legend_classes=None*, *legend_extent=None*, *legend_extent_projection='EPSG:4326'*, *feature_selection=False*, *geometry_attribute=None*, *data=None*)

MVLayer objects are used to define map layers for the Map View Gizmo.

source

The source or data type of the layer (e.g.: ImageWMS)

Type str, required

options
A dictionary representation of the OpenLayers options object for ol.source.
Type dict, required

legend_title
The human readable name of the layer that will be displayed in the legend.
Type str, required

layer_options
A dictionary representation of the OpenLayers options object for ol.layer.
Type dict

editable
If true the layer will be editable with the tethys_map_view drawing/editing tools.
Type bool

feature_selection
Set to True to enable feature selection on this layer. Defaults to False.
Type bool

geometry_attribute
The name of the attribute in the shapefile that describes the geometry
Type str

legend_classes
A list of MVLegendClass objects.
Type list

legend_extent
A list of four ordinates representing the extent that will be used on "zoom to layer": [minx, miny, maxx, maxy].
Type list

legend_extent_projection
The EPSG projection of the extent coordinates. Defaults to "EPSG:4326".
Type str

tethys_data
Dictionary representation of layer data
Type dict

Example

```
# Define GeoJSON layer
geojson_object = {
  'type': 'FeatureCollection',
  'crs': {
    'type': 'name',
    'properties': {
      'name': 'EPSG:3857'
    }
  },
  'features': [
    {
      'type': 'Feature',
      'geometry': {
        'type': 'Point',
        'coordinates': [0, 0]
      }
    },
```

(continues on next page)

(continued from previous page)

```
    {
      'type': 'Feature',
      'geometry': {
        'type': 'LineString',
        'coordinates': [[4e6, -2e6], [8e6, 2e6]]
      }
    },
    {
      'type': 'Feature',
      'geometry': {
        'type': 'Polygon',
        'coordinates': [[[-5e6, -1e6], [-4e6, 1e6], [-3e6, -1e6]]]
      }
    }
  ]
}

geojson_layer = MVLayer(source='GeoJSON',
                        options=geojson_object,
                        legend_title='Test GeoJSON',
                        legend_extent=[-46.7, -48.5, 74, 59],
                        legend_classes=[
                            MVLegendClass('polygon', 'Polygons', fill='rgba(255,
↪255,255,0.8)', stroke='#3d9dcd'),
                            MVLegendClass('line', 'Lines', stroke='#3d9dcd')
                        ])

# Define GeoServer Layer
geoserver_layer = MVLayer(source='ImageWMS',
                          options={'url': 'http://192.168.59.103:8181/geoserver/
↪wms',
                                   'params': {'LAYERS': 'topp:states'},
                                   'serverType': 'geoserver'},
                          legend_title='USA Population',
                          legend_extent=[-126, 24.5, -66.2, 49],
                          legend_classes=[
                              MVLegendClass('polygon', 'Low Density', fill='
↪#00ff00', stroke='#000000'),
                              MVLegendClass('polygon', 'Medium Density', fill='
↪#ff0000', stroke='#000000'),
                              MVLegendClass('polygon', 'High Density', fill='
↪#0000ff', stroke='#000000')
                          ])

# Define GeoServer Tile Layer with Custom tile grid
# The default EPSG:900913 gridset can be used with OpenLayers.
# You must ensure that OpenLayers requests tiles with the same gridset and origin␣
↪as the gridset GeoServer uses
# to use GeoWebCaching capabilities. This is done by setting the TILESORIGIN␣
↪parameter and specifying a custom tileGrid.
# Refer to OpenLayers API for ol.tilegrid.TileGrid for explanation and options.
# See: http://docs.geoserver.org/2.7.0/user/webadmin/tilecache/index.html
geoserver_layer = MVLayer(source='TileWMS',
                          options={'url': 'http://192.168.59.103:8181/geoserver/
↪wms',
                                   'params': {'LAYERS': 'topp:states',
                                              'TILED': True,
```

(continues on next page)

(continued from previous page)

```
                                        'TILESORIGIN': '0.0,0.0'},
                                'serverType': 'geoserver',
                                'tileGrid': {
                                'resolutions': [
                                    156543.03390625,
                                    78271.516953125,
                                    39135.7584765625,
                                    19567.87923828125,
                                    9783.939619140625,
                                    4891.9698095703125,
                                    2445.9849047851562,
                                    1222.9924523925781,
                                    611.4962261962891,
                                    305.74811309814453,
                                    152.87405654907226,
                                    76.43702827453613,
                                    38.218514137268066,
                                    19.109257068634033,
                                    9.554628534317017,
                                    4.777314267158508,
                                    2.388657133579254,
                                    1.194328566789627,
                                    0.5971642833948135,
                                    0.2985821416974068,
                                    0.1492910708487034,
                                    0.0746455354243517,
                                  ],
                                  'extent': [-20037508.34, -20037508.34,␣
↪20037508.34, 20037508.34],
                                  'origin': [0, 0],
                                  'tileSize': [256, 256]
                                }
                            },
                            legend_title='USA Population')

# Define KML Layer
kml_layer = MVLayer(source='KML',
                    options={'url': '/static/tethys_gizmos/data/model.kml'},
                    legend_title='Park City Watershed',
                    legend_extent=[-111.60, 40.57, -111.43, 40.70],
                    legend_classes=[
                        MVLegendClass('polygon', 'Watershed Boundary', fill='
↪#ff8000'),
                        MVLegendClass('line', 'Stream Network', stroke='#0000ff'),
                    ])

# Tiled ArcGIS REST Layer
arc_gis_layer = MVLayer(source='TileArcGISRest',
                        options={'url': 'http://sampleserver1.arcgisonline.com/
↪ArcGIS/rest/services/' + 'Specialty/ESRI_StateCityHighway_USA/MapServer'},
                        legend_title='ESRI USA Highway',
                        legend_extent=[-173, 17, -65, 72]),
```

MVLegendClass

class tethys_sdk.gizmos.**MVLegendClass**(*type*, *value*, *fill=''*, *stroke=''*, *ramp=[]*)

MVLegendClasses are used to define the classes listed in the legend.

type

The type of feature to be represented by the legend class. Either 'point', 'line', 'polygon', or 'raster'.

Type str, required

value

The value or name of the legend class.

Type str, required

fill

Valid RGB color for the fill (e.g.: '#00ff00', 'rgba(0, 255, 0, 0.5)'). Required for 'point' or 'polygon' types.

Type str

stoke

Valid RGB color for the stoke/line (e.g.: '#00ff00', 'rgba(0, 255, 0, 0.5)'). Required for 'line' types and optional for 'polygon' types.

Type str

ramp

A list of hexidecimal RGB colors that will be used to construct a color ramp. Required for 'raster' types.

Type list

Example

```
point_class = MVLegendClass(type='point', value='Cities', fill='#00ff00')
line_class = MVLegendClass(type='line', value='Roads', stroke='rbga(0,0,0,0.7)')
polygon_class = MVLegendClass(type='polygon', value='Lakes', stroke='#0000aa',␣
↪fill='#0000ff')
```

MVLegendImageClass

class tethys_sdk.gizmos.**MVLegendImageClass**(*value*, *image_url*)

MVLegendImageClasses are used to define the classes listed in the legend using a pre-generated image.

value

The value or name of the legend class.

Type str, required

image_url

The url to the legend image.

Type str, required

Example

```
image_class = MVLegendImageClass(value='Cities',
                                 image_url='https://upload.wikimedia.org/
↪wikipedia/commons/d/da/The_City_London.jpg'
                                 )
```

MVLegendGeoServerImageClass

class tethys_sdk.gizmos.**MVLegendGeoServerImageClass**(*value*, *geoserver_url*, *style*, *layer*, *width=20*, *height=10*)

MVLegendGeoServerImageClasses are used to define the classes listed in the legend using the GeoServer generated legend.

value
The value or name of the legend class.
Type str, required

geoserver_url
The url to your geoserver (e.g. http://localhost:8181/geoserver).
Type str, required

style
The name of the geoserver style (e.g. green).
Type str, required

layer
The name of the geoserver layer (e.g. rivers).
Type str, required

width
The legend width (default is 20).
Type int

height
The legend height (default is 10).
Type int

Example

```
image_class = MVLegendGeoServerImageClass(value='Cities',
                                          geoserver_url='http://localhost:8181/
↪geoserver',
                                          style='green',
                                          layer='rivers',
                                          width=20,
                                          height=10)
```

MVDraw

class tethys_sdk.gizmos.**MVDraw**(*controls*, *initial*, *output_format='GeoJSON'*, *line_color='#ffcc33'*, *fill_color='rgba(255, 255, 255, 0.2)'*, *point_color='#ffcc33'*)

MVDraw objects are used to define the drawing options for Map View.

controls
List of drawing controls to add to the map. Valid options are 'Modify', 'Delete', 'Move', 'Point', 'LineString', 'Polygon' and 'Box'.
Type list, required

initial
Drawing control to be enabled initially. Must be included in the controls list.
Type str, required

output_format
Format to output to the hidden text area. Either 'WKT' (for Well Known Text format) or 'GeoJSON'. Defaults to 'GeoJSON'
Type str

line_color
User control for customizing the stroke color of annotation objects
Type str

fill_color
User control for customizing the fill color of polygons (suggest rgba format for setting transparency)
Type str

point_color
User control for customizing the color of points
Type str

Example

```
drawing_options = MVDraw(
    controls=['Modify', 'Delete', 'Move', 'Point', 'LineString', 'Polygon', 'Box
↪'],
    initial='Point',
    output_format='GeoJSON',
    line_color='#663399',
    fill_color='rgba(255,255,255,0.2)',
    point_color='#663399'
)
```

MVView

class tethys_sdk.gizmos.**MVView**(*projection*, *center*, *zoom*, *maxZoom=28*, *minZoom=0*)
MVView objects are used to define the initial view of the Map View. The initial view is set by specifying a center and a zoom level.

projection
Projection of the center coordinates given. This projection will be used to transform the coordinates into the default map projection (EPSG:3857).
Type str

center
An array with the coordinates of the center point of the initial view.
Type list

zoom
The zoom level for the initial view.
Type int or float

maxZoom
The maximum zoom level allowed. Defaults to 28.
Type int or float

minZoom
The minimum zoom level allowed. Defaults to 0.
Type int or float

Example

```
view_options = MVView(
    projection='EPSG:4326',
    center=[-100, 40],
    zoom=3.5,
    maxZoom=18,
    minZoom=2
)
```

JavaScript API

For advanced features, the JavaScript API can be used to interact with the OpenLayers map object that is generated by the Map View JavaScript library.

TETHYS_MAP_VIEW.getMap()

This method returns the OpenLayers map object. You can use the OpenLayers Map API version 4.0.1 to perform operations on this object such as adding layers and custom controls.

```
$(function() { //wait for page to load

    var ol_map = TETHYS_MAP_VIEW.getMap();
    ol_map.addLayer(...);
    ol_map.setView(...);

});
```

Caution: The Map View Gizmo is powered by OpenLayers version 4.0.1. When referring to the OpenLayers documentation, ensure that you are browsing the correct version of documentation (see the URL of the documentation page).

TETHYS_MAP_VIEW.updateLegend()

This method can be used to update the legend after removing/adding layers to the map.

```
$(function() { //wait for page to load

    var ol_map = TETHYS_MAP_VIEW.getMap();
    ol_map.addLayer(...);
    TETHYS_MAP_VIEW.updateLegend();

});
```

TETHYS_MAP_VIEW.zoomToExtent(latlongextent)

This method can be used to set the view of the map to the extent provided. The extent is assumed to be given in the EPSG:4326 coordinate reference system.

```
$(function() { //wait for page to load

    var extent = [-109.49945001309617, 37.58047995600726, -109.44540360290348, 37.
↪679502621605735];
    TETHYS_MAP_VIEW.zoomToExtent(extent);
});
```

TETHYS_MAP_VIEW.clearSelection()

This method applies to the WMS layer feature selection functionality. Use this method to clear the current selection via JavaScript.

```
TETHYS_MAP_VIEW.clearSelection();
```

TETHYS_MAP_VIEW.overrideSelectionStyler(geometry_type, styler)

This method applies to the WMS layer feature selection functionality. This method can be used to override the default styling for the points, lines, and polygons selected feature layers.

- geometry_type (str): The type of the layer that the styler function will apply to. One of: 'points', 'lines', or 'polygons'.
- styler (func): A function that accepts two arguments, feature and resolution, and returns an array of valid ol.style objects.

```
function my_styler(feature, resolution) {
var image, properties;
    properties = feature.getProperties();

    // Default icon
    image = new ol.style.Circle({
        radius: 5,
        fill: new ol.style.Fill({
            color: 'red'
        })
    });

    if ('type' in properties) {
        if (properties.type === 'TANK') {
            image = new ol.style.RegularShape({
                fill: new ol.style.Fill({
                    color: SELECTED_NODE_COLOR
                }),
                stroke: new ol.style.Stroke({
                    color: 'white',
                    width: 1
                }),
                points: 4,
                radius: 14,
```

(continues on next page)

(continued from previous page)

```
                rotation: 0,
                angle: Math.PI / 4
            });

        }
        else if (properties.type === 'RESERVOIR') {
            image = new ol.style.RegularShape({
                fill: new ol.style.Fill({
                    color: SELECTED_NODE_COLOR
                }),
                stroke: new ol.style.Stroke({
                    color: 'white',
                    width: 1
                }),
                points: 3,
                radius: 14,
                rotation: 0,
                angle: 0
            });
        }
    }

    return [new ol.style.Style({image: image})];

}

TETHYS_MAP_VIEW.overrideSelectionStyler('points', my_styler);
```

TETHYS_MAP_VIEW.onSelectionChange(callback)

This method applies to the WMS layer feature selection functionality. The callback function provided will be called each time the feature selection is changed.

- callback (func): A function that accepts three arguments, points_layer, lines_layer, polygons_layer. These are handles on the OpenLayers layers that are rendering the selected features. The features are divided into three layers by type.

```
function my_callback(points_layer, lines_layer, polygons_layer) {
   console.log(points_layer);
}

TETHYS_MAP_VIEW.onSelectionChange(my_callback);
```

TETHYS_MAP_VIEW.getSelectInteraction()

This method applies to the WFS/GeoJSON/KML layer feature selection functionality.

```
$(function() { //wait for page to load

    var selection_interaction = TETHYS_MAP_VIEW.getSelectInteraction();

    //when selected, print feature to developers console
    selection_interaction.getFeatures().on('change:length', function(e) {
```

(continues on next page)

(continued from previous page)

```
      if (e.target.getArray().length > 0) {
        // this means there is at least 1 feature selected
        var selected_feature = e.target.item(0); // 1st feature in Collection
        console.log(selected_feature);

      }
    });

});
```

TETHYS_MAP_VIEW.reInitializeMap()

This method is intended for initializing a map generated from an AJAX request.

Caution: This method assumes there is only one and that there will only ever be one map on the page.

Note: In order to use this, you will either need to use a MapView gizmo or import the JavaScript/CSS libraries in the main html template page using the `register_gizmo_dependency` tag in the `register_gizmos` block.

For example:

```
{% block import_gizmos %}
    {% import_gizmo_dependency map_view %}
{% endblock %}
```

Four elements are required:

1) A controller for the AJAX call with a map view gizmo.

```
@login_required()
def dam_break_map_ajax(request):
    """
    Controller for the dam_break_map ajax request.
    """
    if request.GET:
        ...

        #get layers
        map_layer_list = ...

        # Define initial view for Map View
        view_options = MVView(
            projection='EPSG:4326',
            center=[(bbox[0]+bbox[2])/2.0, (bbox[1]+bbox[3])/2.0],
            zoom=10,
            maxZoom=18,
            minZoom=2,
        )

        # Configure the map
        map_options = MapView(height='500px',
```

(continues on next page)

(continued from previous page)

```
                                width='100%',
                                layers=map_layer_list,
                                controls=['FullScreen'],
                                view=view_options,
                                basemap='OpenStreetMap',
                                legend=True,
                                )

        context = { 'map_options': map_options }

        return render(request, 'dam_break_map_ajax/map_ajax.html', context)
```

2) A url map to the controller in app.py

```
...
    UrlMap(name='dam_break_map_ajax',
           url='dam-break/map/dam_break_map_ajax',
           controller='dam_break.controllers.dam_break_map_ajax'),
...
```

3) A template for with the tethys gizmo (e.g. map_ajax.html)

```
{% load tethys_gizmos %}

{% gizmo map_options %}
```

4) The AJAX call in the javascript

```
$(function() { //wait for page to load

    $.ajax({
        url: ajax_url,
        method: 'GET',
        data: ajax_data,
        success: function(data) {
            //add new map to map div
            $('#main_map_div').html(data);

            TETHYS_MAP_VIEW.reInitializeMap();
        }
    });

});
```

ESRI Map

Last Updated: February 15, 2017

class tethys_sdk.gizmos.**ESRIMap**(*height='100%', width='100%', basemap='topo', view={'center': [-100, 40], 'zoom': 2}, layers=[]*)

ESRI Map View

The ESRI Map View is similar to Map View, but it is powered by ArcGIS API for JavaScript 4.0

Attributes height(string, required): Height of map container in normal css units width(string, required): Width of map container in normal css units basemap(string, required): Basemap layer.

Values=[streets,satellite,hybrid,topo,gray,dark-gray,oceans, national-geographic,terrain,osm,dark-gray-vector,gray-vector,street-vector, topo-vector,streets-night-vector,streets-relief-vector,streets-navigation-vector] zoom(string,required): Zoom Level of the Basemap. view(EMView): An EVView object specifying the initial view or extent for the map

Example

```
# CONTROLLER
from tethys_sdk.gizmos import ESRIMapView

esri_map_view_options = {'height': '700px',
                         'width': '100%',
                         'basemap':'topo'}

# TEMPLATE

{% gizmo esri_map_view_options %}
```

EMLayer

class tethys_sdk.gizmos.**EMLayer**(*type*, *url*)

EMLayer objects are used to define map layers for the ESRI Map Gizmo.

type

The ESRI Layer Type (e.g.: FeatureLayer, ImageLayer)

Type str,required

url

The ESRI Layer WMS url

Type str,required

Example

```
#Define ArcGIS FeatureLayer
```

esri_feature_layer = EMLayer(type='FeatureLayer', url='http://geoserver.byu.edu/arcgis/rest/services/Alabama_Flood/Flood_45/MapServer/0')

#Define ArcGIS ImageLayer

esri_image_layer = EMLayer(type='ImageryLayer', url='https://sampleserver6.arcgisonline.com/arcgis/rest/services/NLCDLandCover2001/ImageServer')

EMView

s .. autoclass:: tethys_sdk.gizmos.EMView

Google Map View

Last Updated: August 10, 2015

class tethys_sdk.gizmos.**GoogleMapView**(*height, width, maps_api_key='', reference_kml_action='', drawing_types_enabled=[], initial_drawing_mode='', output_format='GEOJSON', input_overlays=[None], attributes={}, classes=''*)

The Google Map View is powered by Google Maps 3. It has the drawing library enabled to allow geospatial user input. An optional background dataset can be specified for reference, but only the shapes drawn by the user are returned (see Retrieving Shapes reference section).

Shapes that are drawn on the map by users can be retrieved from the map in two ways. A hidden text field named 'geometry' is updated every time the map is changed. The text in the text field is a string representation of JSON. The geometry can be formatted as either GeoJSON or Well Known Text. This can be configured by setting the output_format parameter. If the Google Map View is embedded in a form, the geometry that is drawn on the map will automatically be submitted with the rest of the form via the hidden text field.

Alternatively, the data can be extracted directly using the JavaScript API (see below).

height
Height of map container in normal css units
Type string, required

width
Width of map container in normal css units
Type string, required

maps_api_key
The Google Maps API key. If the API key is provided in the settings.py via the TETHYS_GIZMOS_GOOGLE_MAPS_API_KEY option, this parameter is not required.
Type string, required

reference_kml_action
The action that returns the background kml datasets. These datasets are used for reference only.
Type url string

drawing_types_enabled
A list of the types of geometries the user will be allowed to draw (POLYGONS, POINTS, POLYLINES).
Type list of strings

initial_drawing_mode
A string representing the drawing mode that will be enabled by default. Valid modes are: 'POLYGONS', 'POINTS', 'POLYLINES'. The mode used must be one of the drawing_types_enabled that the user is allowed to draw.
Type string

output_format
A string specifying the format of the string that is output by the editable map tool. Valid values are 'GEOJSON' for GeoJSON format or 'WKT' for Well Known Text Format.
Type string

input_overlays
A JavaScript-equivalent Python data structure representing GeoJSON or WktJSON containing the geometry and attributes to be added to the map as overlays (see example below). Only points, lines and polygons are supported.
Type PySON

attributes
A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick":

"run_me();"}).

Type dict

classes

Additional classes to add to the primary HTML element (e.g. "example-class another-class").

Type str

Controller Default Example

```
from tethys_sdk.gizmos import GoogleMapView

google_map_view_options = GoogleMapView(height='600px',
                                        width='100%',
                                        reference_kml_action=reverse('gizmos:get_
↪kml'),
                                        drawing_types_enabled=['POLYGONS', 'POINTS
↪', 'POLYLINES'],
                                        initial_drawing_mode='POINTS',
                                        output_format='WKT')

context = {
            'google_map_view_options': google_map_view_options,
          }
```

Controller GeoJSON Example

```
geo_json = {'type':'WKTGeometryCollection',
    'geometries':[
                  {'type':'Point',
                   'wkt':'POINT(-111.5123462677002 40.629197012613545)',
                   'properties':{'id':1,'value':1}
                   },
                  {'type':'Polygon',
                   'wkt':'POLYGON((-111.50153160095215 40.63193284946615, -111.
↪50101661682129 40.617210120505035, -111.48625373840332 40.623594711231775, -111.
↪49123191833496 40.63193284946615, -111.50153160095215 40.63193284946615))',
                   'properties':{'id':2,'value':2}
                   },
                  {'type':'PolyLine',#
                   'wkt':'POLYLINE(-111.49123191833496 40.65003865742191, -111.
↪49088859558105 40.635319920747456, -111.48127555847168 40.64912697157757, -111.
↪48024559020996 40.634668574229735)',
                   'properties':{'id':3,'value':3}
                   }
                  ]
    }

google_map_view_options = GoogleMapView(height='700px',
                                        width='100%',
                                        maps_api_key='S0mEaPIk3y',
                                        drawing_types_enabled=['POLYGONS', 'POINTS
↪', 'POLYLINES'],
                                        initial_drawing_mode='POINTS',
                                        input_overlays=geo_json)

context = {
            'google_map_view_options': google_map_view_options,
          }
```

Controller WKT Example

```
wkt_json = {"type":"GeometryCollection",
    "geometries":[
                  {"type":"Point",
                   "coordinates":[40.629197012613545,-111.5123462677002],
                   "properties":{"id":1,"value":1}},
                  {"type":"Polygon",
                   "coordinates":[[40.63193284946615,-111.50153160095215],[40.
↪617210120505035,-111.50101661682129],[40.623594711231775,-111.48625373840332],
↪[40.63193284946615,-111.49123191833496]],
                   "properties":{"id":2,"value":2}},
                  {"type":"LineString",
                   "coordinates":[[40.65003865742191,-111.49123191833496],[40.
↪635319920747456,-111.49088859558105],[40.64912697157757,-111.48127555847168],
↪[40.634668574229735,-111.48024559020996]],
                   "properties":{"id":3,"value":3}}
                  ]
    }

google_map_view_options = GoogleMapView(height='700px',
                                        width='100%',
                                        maps_api_key='S0mEaPIk3y',
                                        drawing_types_enabled=['POLYGONS', 'POINTS
↪', 'POLYLINES'],
                                        initial_drawing_mode='POINTS',
                                        input_overlays=wkt_json)

context = {
            'google_map_view_options': google_map_view_options,
          }
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo google_map_view_options %}
```

JavaScript API

For advanced features, the JavaScript API can be used to interact with the editable map. If you need capabilities beyond the scope of this API, we recommend using the Google Maps version 3 API to create your own map.

TETHYS_GOOGLE_MAP_VIEW.getMap()

This method returns the Google Map object for direct manipulation through JavaScript.

TETHYS_GOOGLE_MAP_VIEW.getGeoJson()

This method returns the GeoJSON object representing all of the overlays on the map.

TETHYS_GOOGLE_MAP_VIEW.getGeoJsonString()

This method returns a stringified GeoJSON object representing all of the overlays on the map.

TETHYS_GOOGLE_MAP_VIEW.getWktJson()

This method returns a Well Known Text JSON object representing all of the overlays on the map.

TETHYS_GOOGLE_MAP_VIEW.getWktJsonString()

This method returns a stringified Well Known Text JSON object representing all of the overlays on the map.

TETHYS_GOOGLE_MAP_VIEW.swapKmlService(kml_service)

Use this method to swap out the current reference kml layers for new ones.

- **kml_service** *(string)* = URL endpoint that returns a JSON object with a property called 'kml_link' that is an array of publicly accessible URLs to kml or kmz documents

TETHYS_GOOGLE_MAP_VIEW.swapOverlayService(overlay_service, clear_overlays)

Use this method to add new overlays to the map dynamically without reloading the page.

- **overlay_service** *(string)* = URL endpoint that returns a JSON object with a property called 'overlay_json' that has a value of a WKT or GeoJSON object in the same format as is used for input_overlays
- **clear_overlays** *(boolean)* = if true, will clear all overlays from the map prior to adding the new overlays. Otherwise all overlays will be retained.

Jobs Table

Last Updated: August 10, 2015

class tethys_sdk.gizmos.**JobsTable**(*jobs*, *column_fields*, *status_actions=True*, *run_btn=True*, *delete_btn=True*, *results_url=''*, *hover=False*, *striped=False*, *bordered=False*, *condensed=False*, *attributes={}*, *classes=''*, *refresh_interval=5000*, *delay_loading_status=True*)

A jobs table can be used to display users' jobs. The JobsTable gizmo takes the same formatting options as the table view gizmo, but does not allow the columns to be edited. Additional attributes for the jobs table allows for a dynamically updating status field, and action buttons.

jobs
A list/tuple of TethysJob objects.
Type tuple or list, required

column_fields
A tuple or list of strings that represent TethysJob object attributes to show in the columns.
Type tuple or list, required

status_actions
Add a column to the table to show dynamically updating status, and action buttons. If this is false then the values for run_btn, delete_btn, and results_url will be ignored. Default is True.
Type bool

run_btn
Add a button to run the job when job status is "Pending". Default is True.
Type bool

delete_btn
Add a button to delete jobs. Default is True.
Type bool

results_url
A string representing the namespaced path to a controller to that displays job results (e.g. app_name:results_controller)
Type str

hover
Illuminate rows on hover (does not work on striped tables)
Type bool

striped
Stripe rows
Type bool

bordered
Add borders and rounded corners
Type bool

condensed
A more tightly packed table
Type bool

attributes
A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).
Type dict

classes
Additional classes to add to the primary HTML element (e.g. "example-class another-class").
Type str

refresh_interval
The refresh interval for the runtime and status fields in milliseconds. Default is 5000.
Type int

Controller Example

```
from tethys_apps.sdk.gizmos import JobsTable

jobs_table_options = JobsTable(
```

(continues on next page)

(continued from previous page)

```
                                jobs=jobs,
                                column_fields=('id', 'name', 'description',
↪'creation_time', 'execute_time'),
                                hover=True,
                                striped=False,
                                bordered=False,
                                condensed=False,
                                results_url='app_name:results_controller',
                               )
context = {
            'jobs_table_options': jobs_table_options,
          }
```

Template Example

```
{% load tethys_gizmos %}

{% gizmo jobs_table_options %}
```

1.6.11 Testing API

Last Updated: November 18, 2016

Manually testing your app can be very time consuming, especially when modifying a simple line of code usually warrants retesting everything. To help automate and streamline the testing of your app, Tethys Platform provides you with a great starting point by providing the following:

1. A `tests` directory with a `tests.py` script within your app's default scaffold that contains well-commented sample testing code.
2. The Testing API which provides a helpful test class for setting up your app's testing environment.

Writing Tests

Tests should be written in a separate python script that is contained somewhere within your app's scaffold. By default, a `tests` directory already exists in the app-level directory and contains a `tests.py` script. Unless you have a good reason not to, it would be best to start writing your test code here.

As an example, if wanting to automate the testing of a the map controller in the "My First App" from the tutorials, the `tests.py` script might be modified to look like the following:

```
from tethys_sdk.testing import TethysTestCase
from ..app import MyFirstApp

class MapControllerTestCase(TethysTestCase):
    def set_up(self):
        self.create_test_persistent_stores_for_app(MyFirstApp)
        self.create_test_user(username="joe", email="joe@some_site.com", password=
↪"secret")
        self.c = self.get_test_client()

    def tear_down(self):
        self.destroy_test_persistent_stores_for_app(MyFirstApp)

    def test_success_and_context(self):
```

(continues on next page)

(continued from previous page)

```
        self.c.force_login(self.user)
        response = self.c.get('/apps/my-first-app/map/')

        # Check that the response returned successfully
        self.assertEqual(response.status_code, 200)

        # Check that the response returned the context variable
        self.assertIsNotNone(response.context['map_options'])
```

Tethys Platform leverages the native Django testing framework (which leverages the unittests Python module) to make writing tests for your app much simpler. While Tethys Platform encapsulates most of what is needed in its Testing API, it may still be necessary to refer to the Django and Python documentation for additional help while writing tests. Refer to their documentation here:

https://docs.djangoproject.com/en/1.9/topics/testing/overview/#writing-tests

https://docs.python.org/2.7/library/unittest.html#module-unittest

Running Tests

To run any tests at an app level:

1. Open a terminal
2. **Enter the Tethys Platform python environment:** `$ . /usr/lib/tethys/bin/activate`
3. **Enter app-level `tethys test` command.** `(tethys)$ tethys test -f tethys_apps.tethysapp.<app_name(required)>.<folder_name>.<file_name>.<class_name>.<function_name>`

More specifically:

To run all tests across an app: Test command: `(tethys)$ tethys test -f tethys_apps.tethysapp.<app_name>`

To run all tests within specific directory of an app: Test command: `(tethys)$ tethys test -f tethys_apps.tethysapp.<app_name>.<folder_name>`

And so forth... Thus, you can hone in on the exact tests that you want to run.

Note: Remember to append either `-c` or `-C` if you would like a coverage report at the end of the testing printed in your terminal, or opened in your browser as an interactive HTML page, respectively.

API Documentation

class tethys_apps.base.testing.testing.**TethysTestCase**(*methodName='runTest'*)

This class inherits from the Django TestCase class and is itself the class that is should be inherited from when creating test case classes within your app. Note that every specific test written within your custom class inheriting from this class must begin with the word "test" or it will not be executed during testing.

static create_test_persistent_stores_for_app(*app_class*)

Creates temporary persistent store databases for this app to be used in testing.

Parameters **app_class** -- The app class from the app's app.py module

Returns None

static create_test_superuser(*username*, *password*, *email=None*)
Creates and returns a temporary superuser to be used in testing
Parameters
- **username** (*string*) -- The username for the temporary test user
- **password** (*string*) -- The password for the temporary test user
- **email** (*string*) -- The email address for the temporary test user

Returns User object

static create_test_user(*username*, *password*, *email=None*)
Creates and returns temporary user to be used in testing
Parameters
- **username** (*string*) -- The username for the temporary test user
- **password** (*string*) -- The password for the temporary test user
- **email** (*string*) -- The email address for the temporary test user

Returns User object

static destroy_test_persistent_stores_for_app(*app_class*)
Destroys the temporary persistent store databases for this app that were used in testing.
Parameters **app_class** -- The app class from the app's app.py module
Returns None

static get_test_client()
Returns a Client object to be used to mimic a browser in testing
Returns Client object

set_up()
This method is to be overridden by the custom test case classes that inherit from the TethysTestCase class and is used to perform any set up that is applicable to every test function that is defined within the custom test class
Returns None

tear_down()
This method is to be overridden by the custom test case classes that inherit from the TethysTestCase class and is used to perform any tear down that is applicable to every test function that is defined within the custom test class. It is often used in conjunction with the "set_up" function to tear down what was setup therein.
Returns None

1.6.12 Tethys Extensions API

Last Updated: February 22, 2018

Tethys Extensions provide a way for app developers to extend Tethys Platform and modularize functionality that is used accross multiple apps. For example, models, templates and static resources that are used by multiple apps could be created in a Tethys Extension and imported/referenced in the apps that use that functionality. Developers can also create custom gizmos using Tethys Extensions. This section will provide an overview of how to develop Tethys Extensions and use them in apps.

Scaffold and Installation

Last Updated: February 22, 2018

Scaffolding an Extension

Scaffolding Tethys Extensions is done in the same way scaffolding of apps is performed. Just specify the extension option when scaffolding:

```
$ tethys scaffold -e my_first_extension
```

Installing an Extension

This will create a new directory called `tethysext-my_first_extension`. To install the extension for development into your Tethys Portal:

```
$ cd tethysext-my_first_extension
$ python setup.py develop
```

Alternatively, to install the extension on a production Tethys Portal:

```
$ cd tethysext-my_first_extension
$ python setup.py install
```

If the installation was successful, you should see something similar to this when Tethys Platform loads:

```
Loading Tethys Extensions...
Tethys Extensions Loaded: my_first_extension
```

You can also confirm the installation of an extension by navigating to the *Site Admin* page and selecting the `Installed Extensions` link under the `Tethys Apps` heading.

Uninstalling an Extension

An extension can be easily uninstalled using the `uninstall` command provided in the Tethys CLI:

```
$ tethys uninstall -e my_first_extension
```

Extension File Structure

Last Updated: February 22, 2018

The Tethys Extension file structure mimics that of Tethys Apps. Like apps, extensions have the `templates` and `public` directories and the `controllers.py` and `model.py` modules. These modules and directories are used in the same way as they are in apps.

There are some notable differences between apps and extensions, however. Rather than an `app.py` module, the configuration file for extensions is called `ext.py`. Like the `app.py`, the `ext.py` module contains a class that is used to configure the extension. Extensions also contain additional packages and directories such as the `gizmos` package and `templates/gizmos` directory.

Note: Although extensions and apps are similar, extension classes do not support as many operations as the app classes. For example, you cannot specify any service settings (persistent store, spatial dataset, etc.) for extensions, nor can you perform the syncstores on an extension. The capabilities of extensions will certainly grow over time, but some limitations are deliberate.

The sturcture of a freshly scaffolded extension should looks something like this:

```
tethysext-my_first_extension/
|-- tethysext/
|   |-- my_first_extension/
|   |   |-- gizmos/
|   |   |   |-- __init__.py
|   |   |-- public/
|   |   |   |-- js/
|   |   |   |   |-- main.js
|   |   |   |-- css/
|   |   |   |   |-- main.css
|   |   |-- templates/
|   |   |   |-- my_first_extension/
|   |   |   |-- gizmos/
|   |   |-- __init__.py
|   |   |-- contollers.py
|   |   |-- ext.py
|   |   |-- model.py
|   |-- __init__.py
|-- .gitignore
|-- setup.py
```

Templates and Static Files

Last Updated: February 22, 2018

Templates and static files in extensions can be used in other apps. The advantage to using templates and static files from extensions in your apps is that when you update the template or static file in the extension, all the apps that use them will automatically be updated. Just as with apps, store the templates in the `templates` directory and store the static files (css, js, images, etc.) in the `public` directory. Then reference the template or static file in your app's controllers and templates using the namespaced path.

For example, to use an extension template in one of your app's controllers:

```
def my_controller(request):
    """
    A controller in my app, not the extension.
    """
    ...
    return render(request, 'my_first_extension/a_template.html', context)
```

You can reference static files in your app's templates using the `static` tag, just as you would any other static resource:

```
{% load staticfiles %}

<link href="{% static 'my_first_extension/css/a_css_file.css' %}" rel="stylesheet">
<script src="{% static 'my_first_extension/js/a_js_file.css' %}" type="text/javascript
↪"></script>
<img src="{% static 'my_first_extension/images/an_image.png' %}">
```

UrlMaps and Controllers

Last Updated: February 22, 2018

Although `UrlMaps` and controllers defined in extensions are loaded, it is not recommended that you use them to load normal html pages. Rather, use `UrlMaps` in extensions to define REST endpoints that handle any dynamic calls used by your custom gizmos and templates. `UrlMaps` are defined in extensions in the `ext.py` in the same way that they are defined in apps:

```
from tethys_sdk.base import TethysExtensionBase
from tethys_sdk.base import url_map_maker


class MyFirstExtension(TethysExtensionBase):
    """
    Tethys extension class for My First Extension.
    """
    name = 'My First Extension'
    package = 'my_first_extension'
    root_url = 'my-first-extension'
    description = 'This is my first extension.'

    def url_maps():
        """
        Map controllers to URLs.
        """
        UrlMap = url_map_maker(self.root_url)

        return (
            UrlMap(
                name='get_data',
                url='my-first-extension/rest/get-data',
                controller='my_first_extension.controllers.get_data
            ),
        )
```

Models

Last Updated: February 22, 2018

Extensions are not able to be linked to databases, but they can be used to store SQLAlchemy models that are used by multiple apps. Define the SQLAlchemy model as you would normally:

```
import datetime
from sqlalchemy import Column
from sqlalchemy.types import Integer, String, DateTime
from sqlalchemy.ext.declarative import declarative_base


MyFirstExtensionBase = declarative_base()


class Project(MyFirstExtensionBase):
    """
    SQLAlchemy interface for projects table
    """
    __tablename__ = 'projects'
```

(continues on next page)

(continued from previous page)

```
    id = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(String)
    description = Column(String)
    date_created = Column(DateTime, default=datetime.datetime.utcnow)
```

To initialize the tables using a model defined in an extension, import the declarative base from the extension in the initializer function for the persistent store database you'd like to initialize:

```
from tethyext.my_first_extension.models import MyFirstExtensionBase


def init_primary_db(engine, first_time):
    """
    Initializer for the primary database.
    """
    # Create all the tables
    MyFirstExtensionBase.metadata.create_all(engine)
```

To use the extension models to query the database, import them from the extension and use like usual:

```
from tethysapp.my_first_app.app import MyFirstApp as app
from tethysext.my_first_extension.models import Project


def my_controller(request, project_id):
    """
    My app controller.
    """
    SessionMaker = app.get_persistent_store_database('primary_db', as_
↪sessionmaker=True)
    session = SessionMaker()
    project = session.query(Project).get(project_id)

    context = {
        'project': project
    }

    return render(request, 'my_first_app/some_template.html', context)
```

Custom Gizmos

Last Updated: February 22, 2018

Tethys Extensions can be used to create custom gizmos, which can then be used by any app in portals where the extension is installed. This document will provide a brief overview of how to create a gizmo.

Anatomy of a Gizmo

Gizmos are essentially mini-templates that can be embedded in other templates using the `gizmo` tag. They are composed of three primary components:

1. Gizmo Options Class
2. Gizmo Template
3. JavaScript and CSS Dependencies

Each component will be briefly introduced. To illustrate, we will show how a simplified version of the `SelectInput` gizmo could be implemented as a custom Gizmo in an extension.

Gizmo Organization

The files used to define custom gizmos must be organized in a specific way in your app extension. Each gizmo options class must be located in its own python module and the file should be located in the `gizmos` package of your extension. The template for the gizmo must an HTML file located within the `templates/gizmos/` folder of your extension.

Gizmo files must follow a specific naming convention: the python module containing the gizmo options class and the name of the gizmo template must have the same name as the gizmo. For example, if the name of the gizmo you are creating is `custom_select_input` then the name of the gizmo template would be `custom_select_input.html` and the name of the gizmo options module would be `custom_select_input.py`.

JavaScript and CSS dependencies should be stored in the `public` directory of your extension as usual or be publicly available from a CDN or similar. Dependencies stored locally can be organized however you prefer within the `public` directory.

Finally, you must import the gizmo options class in the `gizmos/__init__.py` module. Only Gizmos imported here will be accessible. For the custom select input example, the file structure would look something like this:

```
tethysext-my_first_extension/
|-- tethysext/
|   |-- my_first_extension/
|   |   |-- gizmos/
|   |   |   |-- custom_select_input.py
|   |   |-- public/
|   |   |   |-- gizmos/
|   |   |   |   |-- custom_select_input/
|   |   |   |   |   |-- custom_select_input.css
|   |   |   |   |   |-- custom_select_input.js
|   |   |-- templates/
|   |   |   |-- gizmos/
|   |   |   |   |-- custom_select_input.html
```

Important: Gizmo names must be globally unique within a portal. If a portal has two extensions that implement gizmos with the same name, they will conflict and likely not work properly.

Gizmo Options Class

A gizmo options class is a class that inherits from the `TethysGizmoOptions` base class. It can be thought of as the context for the gizmo template. Any property or attribute of the gizmo options class will be available as a variable in the Gizmo Template.

For the custom select input gizmo, create a new python module in the `gizmos` package called `custom_select_input.py` and add the following contents:

```
from tethys_sdk.gizmos import TethysGizmoOptions


class CustomSelectInput(TethysGizmoOptions):
    """
    Custom select input gizmo.
    """
    gizmo_name = 'custom_select_input'

    def __init__(self, name, display_text='', options=(), initial=(),␣
↪multiselect=False,
                 disabled=False, error='', **kwargs):
        """
        constructor
        """
        # Initialize parent
        super(CustomSelectInput, self).__init__(**kwargs)

        # Initialize Attributes
        self.name = name
        self.display_text = display_text
        self.options = options
        self.initial = initial
        self.multiselect = multiselect
        self.disabled = disabled
        self.error = error
```

It is important that `gizmo_name` property is the same as the name of the python module and template for the gizmo. Also, it is important to include `**kwargs` as an argument to your contstructor and use it to initialize the parent `TethysGizmoOptions` object. This will catch any of the parameters that are common to all gizmos like `attributes` and `classes`.

After defining the gizmo options class, import it in the `gizmos/__init__.py` module:

```
from custom_select_input import CustomSelectInput
```

Gizmo Template

Gizmo templates are similar to the templates used for Tethys apps, but much simpler.

For the custom select input gizmo, create a new template in the `templates/gizmos/` directory with the same name as your gizmo, `custom_select_input.html`, with the following contents:

```
{% load staticfiles %}

<div class="form-group {% if error %} has-error {% endif %}">
  {% if display_text %}
```

(continues on next page)

(continued from previous page)

```
    <label class="control-label" for="{{ name }}">{{ display_text }}</label>
  {% endif %}
  <select id="{{ name }}"
          name="{{ name }}"
          class="select2{% if classes %} {{ classes }}{% endif %}"
          {% if attributes %}
            {% for key, value in attributes.items %}
              {{ key }}="{{ value }}"
            {% endfor %}
          {% endif %}
          {% if multiselect %}multiple{% endif %}
          {% if disabled %}disabled{% endif %}>
    {% for option, value in options %}
      {% if option in initial or value in initial %}
        <option value="{{value}}" selected="selected">{{ option }}</option>
      {% else %}
        <option value="{{value}}">{{ option }}</option>
      {% endif %}
    {% endfor %}
  </select>
  {% if error %}
  <p class="help-block">{{ error }}</p>
  {% endif %}
</div>
```

The variables in this template are defined by the attributes of the gizmo options object. Notice how the `classes` and `attributes` variables are handled. It is a good idea to handle these variables for each of your gizmos, because most gizmos support them and developers will expect them.

JavaScript and CSS Dependencies

Some gizmos have JavaScript and/or CSS dependencies. The `TethysGizmoOptions` base class provides methods for specifying different types of dependencies:

- `get_vendor_js`: For vendor/3rd party javascript.
- `get_vendor_css`: For vendor/3rd party css.
- `get_gizmo_js`: For your custom javascript.
- `get_gizmo_css`: For your custom css.
- `get_tethys_gizmos_js`: For global gizmo javascript. Changing this could cause other gizmos to stop working. Best not to mess with it unless you know what you are doing.
- `get_tethys_gizmos_css`: For global gizmo css. Changing this could cause other gizmos to stop working. Best not to mess with it unless you know what you are doing.

Note: Tethys provides `Twitter Bootstrap` and `jQuery`, so you don't need to include these as gizmo dependencies.

The custom select input depends on the select2 libraries and some custom javascript and css. Create `custom_select_input.js` and `custom_select_input.css` in the `public/gizmos/custom_select_input/` directory, creating the directory as well. Add the following contents to each file:

Add this content to the `custom_select_input.css` file:

```
.select2 {
    width: 100%;
}
```

Add this content to the `custom_select_input.js` file:

```
$(document).ready(function() {
    $('.select2').select2();
});
```

Modify the gizmo options class to include these dependencies:

```
from tethys_sdk.gizmos import TethysGizmoOptions


class CustomSelectInput(TethysGizmoOptions):
    """
    Custom select input gizmo.
    """
    gizmo_name = 'custom_select_input'

    def __init__(self, name, display_text='', options=(), initial=(),␣
↪multiselect=False,
                 disabled=False, error='', **kwargs):
        """
        constructor
        """
        # Initialize parent
        super(CustomSelectInput, self).__init__(**kwargs)

        # Initialize Attributes
        self.name = name
        self.display_text = display_text
        self.options = options
        self.initial = initial
        self.multiselect = multiselect
        self.disabled = disabled
        self.error = error

    @staticmethod
    def get_vendor_js():
        """
        JavaScript vendor libraries.
        """
        return ('https://cdnjs.cloudflare.com/ajax/libs/select2/4.0.6-rc.0/js/select2.
↪min.js',)

    @staticmethod
    def get_vendor_css():
        """
        CSS vendor libraries.
        """
        return ('https://cdnjs.cloudflare.com/ajax/libs/select2/4.0.6-rc.0/css/
↪select2.min.css',)

    @staticmethod
```

(continues on next page)

(continued from previous page)

```
    def get_gizmo_js():
        """
        JavaScript specific to gizmo.
        """
        return ('my_first_extension/gizmos/custom_select_input/custom_select_input.js
↪',)

    @staticmethod
    def get_gizmo_css():
        """
        CSS specific to gizmo .
        """
        return ('my_first_extension/gizmos/custom_select_input/custom_select_input.css
↪',)
```

Using a Custom Gizmo

To use a custom gizmo in an app, import the gizmo options object from the extension and create a new instance fo the gizmo in the app controller. Then use it with the `gizmo` template tag as normal.

Import and create a new instance of the gizmo in your controller:

```
from tethysext.my_first_extension.gizmos import CustomSelectInput


def my_app_controller(request):
    """
    Example controller using extension gizmo
    """
    my_select = CustomSelectInput(
        name = 'my_select',
        display_text = 'Select One:',
        options = (('Option 1', '1'), ('Option 2', '2'), ('Option 3', '3')),
        initial = ('2')
    )

    context = {
        'my_select': my_select,
    }
    return render(request, 'my_first_app/a_template.html', context)
```

Then use the gizmo as usual in `a_template.html`:

1.6.13 Tethys Services APIs

Last Updated: May 2017

Tethys Services consists of several APIs that can be used to work with external data and processing services. Use the Persistent Store Services APIs to connect to SQL databases. Use the Dataset Services to consume file dataset services like CKAN or HydroShare. The Spatial Dataset Services can be used to connect to map servers like GeoServer and the Web Processing Services can be used to consume processing services such as those hosted by 52 North installations.1

Persistent Stores API

Last Updated: May 2017

The Persistent Store API streamlines the use of SQL databases in Tethys apps. Using this API, you can provision SQL databases for your app. The databases that will be created are PostgreSQL databases. Currently, no other databases are supported.

The process of creating a new persistent database can be summarized in the following steps:

1. create a new PersistentStoreDatabaseSetting in the *app configuration file*,
2. assign a PersistentStoreService to the PersistentStoreDatabaseSetting from the admin pages.
3. create a data model to define the table structure of the database,
4. write a persistent store initialization function, and
5. use the Tethys command line interface to create the persistent store.

More detailed descriptions of each step of the persistent store process will be discussed in this article.

Persistent Store Settings

Using *persistent stores* in your app is accomplished by adding the `persistent_store_settings()` method to your *app class*, which is located in your *app configuration file* (`app.py`). This method should return a list or tuple of `PersistentStoreDatabaseSetting` and/or `PersistentStoreConnectionSetting` objects. For example:

```
from tethys_sdk.base import TethysAppBase
from tethys_sdk.app_settings import PersistentStoreDatabaseSetting


class MyFirstApp(TethysAppBase):
    """
    Tethys App Class for My First App.
    """
    ...

    def persistent_store_settings(self):
        ps_settings = (
            PersistentStoreDatabaseSetting(
                name='example_db',
                description='Primary database for my_first_app.',
                initializer='my_first_app.model.init_example_db',
                required=True
            ),
        )

        return ps_settings
```

Caution: The ellipsis in the code block above indicates code that is not shown for brevity. **DO NOT COPY VERBATIM**.

In this example, a database called "example_db" would be created for this app. It would be initialized by a function called "init_example_db", which is located in a Python module called `init_stores.py`. Notice that the path to the initializer function is given using dot notation (e.g.: `'foo.bar.function'`).

Persistent store databases follow a specific naming convention that is a combination of the app name and the name that is provided during registration. For example, the database for the example above may have a name "my_first_app_example_db". To register another database, add another `PersistentStoreDatabaseSetting` object to the tuple that is returned by the `persistent_store_settings()` method.

Assign Persistent Store Service

The `PersistentStoreDatabaseSetting` can be thought of as a socket for a connection to a database. Before we can do anything with the `PersistentStoreDatabaseSetting` we need to "plug in" or assign a `PersistentStoreService` to the setting. The `PersistentStoreService` contains the connection information and can be used by multiple apps. Assigning a `PersistentStoreService` is done through the Admin Interface of Tethys Portal as follows:

1. Create `PersistentStoreService` if one does not already exist
    a. Access the Admin interface of Tethys Portal by clicking on the drop down menu next to your user name and selecting the "Site Admin" option.
    b. Scroll to the **Tethys Service** section of the Admin Interface and select the link titled **Persistent Store Services**.
    c. Click on the **Add Persistent Store Services** button.
    d. Fill in the connection information to the database server.
    e. Press the **Save** button to save the new `PersistentStoreService`.

    Tip: You do not need to create a new `PersistentStoreService` for each `PersistentStoreDatabaseSetting` or each app. Apps and `PersistentStoreDatabaseSettings` can share `PersistentStoreServices`.

2. Navigate to App Settings Page
    a. Return to the Home page of the Admin Interface using the **Home** link in the breadcrumbs or as you did in step 1a.
    b. Scroll to the **Tethys Apps** section of the Admin Interface and select the **Installed Apps** linke.
    c. Select the link for your app from the list of installed apps.
3. Assign `PersistentStoreService` to the appropriate `PersistentStoreDatabaseSetting`
    a. Scroll to the **Persistent Store Database Settings** section and locate the `PersistentStoreDatabaseSetting`.

    Note: If you don't see the `PersistentStoreDatabaseSetting` in the list, uninstall the app and reinstall it again.

    b. Assign the appropriate `PersistentStoreService` to your `PersistentStoreDatabaseSetting` using the drop down menu in the **Persistent Store Service** column.
    c. Press the **Save** button at the bottom of the page to save your changes.

Note: During development you will assign the `PersistentStoreService` setting yourself. However, when the app is installed in production, this steps is performed by the portal administrator upon installing your app, which may or may not be yourself.

Data Model Definition

The tables for a persistent store should be defined using an SQLAlchemy data model. The recommended location for data model code is `model.py` file that is generated with the scaffold. The following example illustrates what a typical SQLAlchemy data model may consist of:

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, Float

# DB Engine, sessionmaker, and base
Base = declarative_base()


# SQLAlchemy ORM definition for the stream_gages table
class StreamGage (Base):
    """
    Example SQLAlchemy DB Model
    """
    __tablename__ = 'stream_gages'

    # Columns
    id = Column(Integer, primary_key=True)
    latitude = Column(Float)
    longitude = Column(Float)
    value = Column(Integer)

    def __init__(self, latitude, longitude, value):
        """
        Constructor for a gage
        """
        self.latitude = latitude
        self.longitude = longitude
        self.value = value
```

Object Relational Mapping

Each class in an SQLAlchemy data model defines a table in the database. Each object instantiated using an SQLAlchemy class represent a row or record in the table. The contents of a table or multiple rows would be represented as a list of SQLAlchemy objects. This pattern for interacting between database tables using objects in code is called Object Relational Mapping or ORM.

The example above consists of a single table called "stream_gages", as denoted by the `__tablename__` property of the `StreamGage` class. The `StreamGage` class is defined as an SQLAlchemy data model class because it inherits from the `Base` class that was created in the previous lines using the `declarative_base()` function provided by SQLAlchemy. This inheritance makes SQLAlchemy aware of the `StreamGage` class is part of the data model. All tables belonging to the same data model should inherit from the same `Base` class.

The columns of tables defined using SQLAlchemy classes are defined by properties that contain `Column` objects. The class in the example above defines four columns for the "stream_gages" table: `id`, `latitude`, `longitude`, and

value. The column type and options are defined by the arguments passed to the Column constructor. For example, the latitude column is of type Float while the id column is of type Integer and is also flagged as the primary key for the table.

Engine Object

Anytime you wish to retrieve data from a persistent store database, you will need to connect to it. In SQLAlchemy, the connection to a database is provided via engine objects. You can retrieve the SQLAlchemy engine object for a persistent store database using the get_persistent_store_database() method of the *app class* provided by the Persistent Store API. The example above shows how the get_persistent_store_engine() function should be used. Provide the name of the persistent store to the function and it will return the engine object for that store.

Note: Although the full name of the persistent store database follows the app-database naming convention described in *Persistent Store Settings*, you need only use the name you provided when you created the setting to retrieve the engine using get_persistent_store_database().

Session Object

Database queries are issued using SQLAlchemy session objects. You need to create new session objects each time you perform a new set of queries (i.e.: in each controller). Creating session objects is done via a SessionMaker. In the example above, the SessionMaker is created using the sessionmaker() function provided by SQLAlchemy. The SessionMaker is bound to the engine object. This means that anytime a session is created using that SessionMaker it will automatically be connected to the database that the engine provides a connection to. You should create a SessionMaker for each persistent store that you create. An example of how to use session and SessionMaker objects is shown in the *Initialization Function* section.

SQLAlchemy ORM is a powerful tool for working with SQL databases. As a primer to SQLAlchemy ORM, we highly recommend you complete the Object Relational Tutorial.

Initialization Function

The code for initializing a persistent store database should be defined in an initialization function. The recommended location for initialization functions is the :file:init_stores.py file that is generated with the scaffold. In most cases, each persistent store should have it's own initialization function. The initialization function makes use of the SQLAlchemy data model to create the tables and load any initial data the database may need. The following example illustrates a typical initialization function for a persistent store database:

```
from sqlalchemy.orm import sessionmaker
from .model import Base, StreamGage


def init_example_db(engine, first_time):
    """
    An example persistent store initializer function
    """
    # Create tables
    Base.metadata.create_all(engine)

    # Initial data
```

(continues on next page)

(continued from previous page)

```
    if first_time:
        # Make session
        SessionMaker = sessionmaker(bind=engine)
        session = SessionMaker()

        # Gage 1
        gage1 = StreamGage(latitude=40.23812952992122,
                           longitude=-111.69585227966309,
                           value=1)

        session.add(gage1)

        # Gage 2
        gage2 = StreamGage(latitude=40.238784729316215,
                           longitude=-111.7101001739502,
                           value=2)

        session.add(gage2)

        session.commit()
        session.close()
```

Create Tables

The SQLAlchemy `Base` class defined in the data model is used to create the tables. Every class that inherits from the `Base` class is tracked by a `metadata` object. As the name implies, the `metadata` object collects metadata about each table defined by the classes in the data model. This information is used to create the tables when the `metadata.create_all()` method is called:

```
Base.metadata.create_all(engine)
```

Note: The `metadata.create_all()` method requires the `engine` object as an argument for connection information.

Initial Data

The initialization functions should also be used to add any initial data to persistent store databases. The `first_time` parameter is provided to all initialization functions as an aid to adding initial data. It is a boolean that is `True` if the function is being called after the tables have been created for the first time. This is provided as a mechanism for adding initial data only the first time the initialization function is run. Notice the code that adds initial data to the persistent store database in the example above is wrapped in a conditional statement that uses the `first_time` parameter.

Example SQLAlchemy Query

This initial data code uses an SQLAlchemy data model to add four stream gages to the persistent store database. A new `session` object is created using the `SessionMaker` that was defined in the model. Creating a new record in the database using SQLAlchemy is achieved by creating a new `StreamGage` object and adding it to the `session` object using the `session.add()` method. The `session.commit()` method is called, to persist the new records to the persistent store database. Finally, `session.close()` is called to free up the connection to the database.

Managing Persistent Stores

Persistent store management is handled via the **`syncstores`** command provided by the Tethys Command Line Interface (Tethys CLI). This command is used to create the persistent stores of apps during installation. It should also be used anytime you make changes to persistent store registration, data models, or initialization functions. For example, after performing the registration, creating the data model, and defining the initialization function in the example above, the **`syncstores`** command would need to be called from the command line to create the new persistent store:

```
$ tethys syncstores my_first_app
```

This command would create all the non-existent persistent stores that are registered for `my_first_app` and run the initialization functions for them. This is the most basic usage of the **`syncstores`** command. A detailed description of the **`syncstores`** command can be found in the *Command Line Interface* documentation.

Dynamic Persistent Store Provisioning

As of Tethys Platform 1.3.0, methods were added to the app class that allow apps to create persistent stores dynamically at run time, list existing persistent stores, and check if a given persistent store exists. See the API documentation below for details.

API Documentation

`TethysAppBase.`**`persistent_store_settings`**`()`

Override this method to define a persistent store service connections and databases for your app.

Returns A list or tuple of `PersistentStoreDatabaseSetting` or `PersistentStoreConnectionSetting` objects.

Return type iterable

Example:

```
from tethys_sdk.app_settings import PersistentStoreDatabaseSetting,␣
↪PersistentStoreConnectionSetting

class MyFirstApp(TethysAppBase):

    def persistent_store_settings(self):
        """
        Example persistent_store_settings method.
        """

        ps_settings = (
            # Connection only, no database
            PersistentStoreConnectionSetting(
```

(continues on next page)

(continued from previous page)

```
                name='primary',
                description='Connection with superuser role needed.',
                required=True
            ),
            # Connection only, no database
            PersistentStoreConnectionSetting(
                name='creator',
                description='Create database role only.',
                required=False
            ),
            # Spatial database
            PersistentStoreDatabaseSetting(
                name='spatial_db',
                description='for storing important spatial stuff',
                required=True,
                initializer='appsettings.model.init_spatial_db',
                spatial=True,
            ),
            # Non-spatial database
            PersistentStoreDatabaseSetting(
                name='temp_db',
                description='for storing temporary stuff',
                required=False,
                initializer='appsettings.model.init_temp_db',
                spatial=False,
            )
        )

        return ps_settings
```

classmethod TethysAppBase.**get_persistent_store_connection**(*name*, *as_url=False*, *as_sessionmaker=False*)

Gets an SQLAlchemy Engine or URL object for the named persistent store connection.

Parameters

- **name** (*string*) -- Name of the PersistentStoreConnectionSetting as defined in app.py.
- **as_url** (*bool*) -- Return SQLAlchemy URL object instead of engine object if True. Defaults to False.
- **as_sessionmaker** (*bool*) -- Returns SessionMaker class bound to the engine if True. Defaults to False.

Returns An SQLAlchemy Engine or URL object for the persistent store requested.

Return type sqlalchemy.Engine or sqlalchemy.URL

Example:

```
from my_first_app.app import MyFirstApp as app

conn_engine = app.get_persistent_store_connection('primary')
conn_url = app.get_persistent_store_connection('primary', as_url=True)
SessionMaker = app.get_persistent_store_database('primary', as_sessionmaker=True)
session = SessionMaker()
```

classmethod TethysAppBase.**get_persistent_store_database**(*name*, *as_url=False*, *as_sessionmaker=False*)

Gets an SQLAlchemy Engine or URL object for the named persistent store database given.

Parameters

- **name** (*string*) -- Name of the PersistentStoreConnectionSetting as defined in

app.py.

- **as_url** (*bool*) -- Return SQLAlchemy URL object instead of engine object if True. Defaults to False.
- **as_sessionmaker** (*bool*) -- Returns SessionMaker class bound to the engine if True. Defaults to False.

Returns An SQLAlchemy Engine or URL object for the persistent store requested.

Return type sqlalchemy.Engine or sqlalchemy.URL

Example:

```
from my_first_app.app import MyFirstApp as app

db_engine = app.get_persistent_store_database('example_db')
db_url = app.get_persistent_store_database('example_db', as_url=True)
SessionMaker = app.get_persistent_store_database('example_db', as_
↪sessionmaker=True)
session = SessionMaker()
```

classmethod TethysAppBase.**list_persistent_store_connections**()

Returns a list of existing persistent store connections for this app.

Returns A list of persistent store connection names.

Return type list

Example:

```
from my_first_app.app import MyFirstApp as app

ps_connections = app.list_persistent_store_connections()
```

classmethod TethysAppBase.**list_persistent_store_databases**(*dynamic_only=False*, *static_only=False*)

Returns a list of existing persistent store databases for the app.

Parameters

- **dynamic_only** (*bool*) -- only persistent store created dynamically if True. Defaults to False.
- **static_only** (*bool*) -- only static persistent stores if True. Defaults to False.

Returns A list of all persistent store database names for the app.

Return type list

Example:

```
from my_first_app.app import MyFirstApp as app

ps_databases = app.list_persistent_store_databases()
```

classmethod TethysAppBase.**persistent_store_exists**(*name*)

Returns True if a persistent store with the given name exists for the app.

Parameters **name** (*string*) -- Name of the persistent store database to check.

Returns True if persistent store exists.

Return type bool

Example:

```
from my_first_app.app import MyFirstApp as app

result = app.persistent_store_exists('example_db')

if result:
    engine = app.get_persistent_store_engine('example_db')
```

classmethod TethysAppBase.**create_persistent_store**(*db_name*, *connection_name*, *spatial=False*, *initializer=''*, *refresh=False*, *force_first_time=False*)

Creates a new persistent store database for the app. This method is idempotent.

Parameters

- **db_name** (*string*) -- Name of the persistent store that will be created.
- **connection_name** (*string|None*) -- Name of persistent store connection or None if creating a test copy of an existing persistent store (only while in the testing environment)
- **spatial** (*bool*) -- Enable spatial extension on the database being created when True. Connection must have superuser role. Defaults to False.
- **initializer** (*string*) -- Dot-notation path to initializer function (e.g.: 'my_first_app.models.init_db').
- **refresh** (*bool*) -- Drop database if it exists and create again when True. Defaults to False.
- **force_first_time** (*bool*) -- Call initializer function with "first_time" parameter forced to True, even if this is not the first time intializing the persistent store database. Defaults to False.

Returns True if successful.

Return type bool

Example:

```
from my_first_app.app import MyFirstApp as app

result = app.create_persistent_store('example_db', 'primary')

if result:
    engine = app.get_persistent_store_engine('example_db')
```

classmethod TethysAppBase.**drop_persistent_store**(*name*)

Drop a persistent store database for the app. This method is idempotent.

Parameters **name** (*string*) -- Name of the persistent store to be dropped.

Returns True if successful.

Return type bool

Example:

```
from my_first_app.app import MyFirstApp as app

result = app.drop_persistent_store('example_db')

if result:
    # App database 'example_db' was successfully destroyed and no longer exists
    pass
```

Spatial Persistent Stores API

Last Updated: May 2017

Persistent store databases can support spatial data types. The spatial capabilities are provided by the PostGIS extension for the PostgreSQL database. PostGIS extends the column types of PostgreSQL databases by adding `geometry`, `geography`, and `raster` types. PostGIS also provides hundreds of database functions that can be used to perform spatial operations on data stored in spatial columns. For more information on PostGIS, see http://www.postgis.net.

The following article details the the spatial capabilities of persistent stores in Tethys Platform. This article builds on the concepts and ideas introduced in the *Persistent Stores API* documentation. Please review it before continuing.

Spatial Persistent Store Settings

Registering spatially enabled persistent stores is the same process as registering normal persistent stores. The only difference is that you will set the `spatial` attribute of the `PersistentStoreDatabaseSetting` object to `True`:

```
from tethys_sdk.base import TethysAppBase
from tethys_sdk.app_settings import PersistentStoreDatabaseSetting


class MyFirstApp(TethysAppBase):
    """
    Tethys App Class for My First App.
    """
    ...

    def persistent_store_settings(self):
        ps_settings = (
            PersistentStoreDatabaseSetting(
                name='spatial_db',
                description='Primary spatially enabled database for my_first_app.',
                initializer='my_first_app.model.init_spatial_db',
                required=True,
                spatial=True
            ),
        )

        return ps_settings
```

Caution: The ellipsis in the code block above indicates code that is not shown for brevity. **DO NOT COPY VERBATIM**.

Adding Spatial Columns to Model

Working with the `raster`, `geometry`, and `geography` column types provided by PostGIS is not supported natively in SQLAlchemy. Tethys Platform includes GeoAlchemy2, which extends SQLAlchemy to support spatial columns and database functions. The following example illustrates how a data model could be developed using SQLAlchemy and GeoAlchemy2:

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer
from sqlalchemy.orm import sessionmaker

from geoalchemy2 import Geometry

# Spatial DB Engine, sessiomaker, and base
Base = declarative_base()

# SQLAlchemy ORM definition for the spatial_stream_gages table
class SpatialStreamGage(Base):
    """
    Example of SQLAlchemy spatial DB model
    """
    __tablename__ = 'stream_gages'

    # Columns
    id = Column(Integer, primary_key=True)
    value = Column(Integer)
    geometry = Column(Geometry('POINT'))

    def __init__(self, latitude, longitude, value):
        """
        Constructor for a gage
        """
        self.geometry = 'SRID=4326;POINT({0} {1})'.format(longitude, latitude)
        self.value = value
```

This data model is very similar to the data model defined in the *Persistent Stores API* documentation. Rather than using `Float` columns to store the latitude and longitude coordinates, the spatial data model uses a GeoAlchemy2 `Geometry` column called "geometry". Notice that the constructor (`__init__.py`) takes the `latitude` and `longitude` provided and sets the value of the `geometry` column to a string with a special format called Well Known Text. This is a common pattern when working with GeoAlchemy2 columns.

Initialization Function

Initializing spatial persistent stores is performed in exactly the same way as normal persistent stores. An initialization function for the example above, would look like this:

```
from sqlalchemy.orm import sessionmaker
from .model import Base, SpatialStreamGage

def init_spatial_db(engine, first_time):
    """
    An example persistent store initializer function
    """
    # Create tables
    Base.metadata.create_all(engine)
```

(continues on next page)

(continued from previous page)

```
    # Initial data
    if first_time:
        # Make session
        SessionMaker = sessionmaker(bind=engine)
        session = SessionMaker()

        # Gage 1
        gage1 = SpatialStreamGage(
            latitude=40.23812952992122,
            longitude=-111.69585227966309,
            value=1
        )

        session.add(gage1)

        # Gage 2
        gage2 = SpatialStreamGage(
            latitude=40.238784729316215,
            longitude=-111.7101001739502,
            value=2
        )

        session.add(gage2)

        session.commit()
        session.close()
```

Using Spatial Database Functions

One of the major advantages of storing spatial data in PostGIS is that the data is exposed to spatial querying. PostGIS includes over 400 database functions (not counting variants) that can be used to perform spatial operations on the data stored in the database. Refer to the Geometry Function Reference and the Raster Function Reference in the PostGIS documentation for more details.

GeoAlchemy2 makes it easy to use the spatial functions provided by PostGIS to perform spatial queries. For example, the `ST_Contains` function can be used to determine if one geometry is contained inside another geometry. To perform this operation on the spatial stream gage model would look something like this:

```
from sqlalchemy import func
from .model import SpatialStreamGage, SpatialSessionMaker

session = SpatialSessionMaker()
query = session.query(SpatialStreamGage).filter(
            func.ST_Contains('POLYGON((0 0,0 1,1 1,0 1,0 0))', SpatialStreamGage.geom)
            )
```

Important: This article only briefly introduces the concepts of working with GeoAlchemy2. It is highly recommended that you complete the GeoAlchemy ORM tutorial.

Dataset Services API

Last Updated: May 2017

Dataset services are web services external to Tethys Platform that can be used to store and publish file-based *datasets* (e.g.: text files, Excel files, zip archives, other model files). Tethys app developers can use the Dataset Services API to access *datasets* for use in their apps and publish any resulting *datasets* their apps may produce. Supported options include CKAN and HydroShare.

Key Concepts

Tethys Dataset Services API provides a standardized interface for interacting with *dataset services*. This means that you can use datasets from different sources without completely overhauling your code. Each of the supported *dataset services* provides a `DatasetEngine` object with the same methods. For example, all `DatasetEngine` objects have a method called `list_datasets()` that will have the same result, returning a list of the datasets that are available.

There are two important definitions that are applicable to *dataset services*: *dataset* and *resource*. A *resource* contains a single file or other object and the metadata associated with it. A *dataset* is a container for one or more resources.

Dataset Service Engine References

All `DatasetEngine` objects implement a minimum set of base methods. However, some `DatasetEngine` objects may include additional methods that are unique to that `DatasetEngine` and the arguments that each method accepts may vary slightly. Refer to the following references for the methods that are offered by each `DatasetEngine`.

Base Dataset Engine Reference

Last Updated: January 19, 2015

All `DatasetEngine` object provide a minimum set of methods for interacting with *datasets* and *resources*. Specifically, the methods allow the standard CRUD operations (Create, Read, Update, Delete) for both *datasets* and *resources*.

All `DatasetEngine` methods return a dictionary, often called the Response dictionary. The Response dictionary contains an item named 'success' that contains a boolean indicating whether the operation was successful or not. If 'success' is `True`, then the the dictionary will also have an item named 'result' that contains the result of the operation. If 'success' is `False`, then the Response dictionary will contain an item called 'error' with information about what went wrong.

The following reference provides a summary of the base methods and properties provided by all `DatasetEngine` objects.

Properties

`DatasetEngine.` **endpoint** (string): URL for the *dataset service* API endpoint.

`DatasetEngine.` **apikey** (string, optional): API key may be used for authorization.

`DatasetEngine.` **username** (string, optional): Username key may be used for authorization.

`DatasetEngine.` **password** (string, optional): Password key may be used for authorization.

`DatasetEngine.` **type** (string, readonly): Identifies the type of `DatasetEngine` object.

Create Methods

abstract `DatasetEngine.`**create_dataset**(*name*, ***kwargs*)

Create a new dataset.

Parameters

- **name** (*string*) -- Name of the dataset to create.
- ****kwargs** (*kwargs, optional*) -- Any number of additional keyword arguments.

Returns Response dictionary

Return type (dict)

abstract `DatasetEngine.`**create_resource**(*dataset_id*, *url=None*, *file=None*, ***kwargs*)

Create a new resource.

Parameters

- **dataset_id** (*string*) -- Identifier of the dataset to which the resource will be added.
- **url** (*string, optional*) -- URL of resource to associate with resource.
- **file** (*string, optional*) -- Path of file to upload as resource.
- ****kwargs** (*kwargs, optional*) -- Any number of additional keyword arguments.

Returns Response dictionary

Return type (dict)

Read Methods

abstract `DatasetEngine.`**get_dataset**(*dataset_id*, ***kwargs*)

Retrieve a dataset object.

Parameters

- **dataset_id** (*string*) -- Identifier of the dataset to retrieve.
- ****kwargs** (*kwargs, optional*) -- Any number of additional keyword arguments.

Returns Response dictionary

Return type (dict)

abstract `DatasetEngine.`**get_resource**(*resource_id*, ***kwargs*)

Retrieve a resource object.

Parameters

- **resource_id** (*string*) -- Identifier of the dataset to retrieve.
- ****kwargs** (*kwargs, optional*) -- Any number of additional keyword arguments.

Returns Response dictionary

Return type (dict)

abstract DatasetEngine.**search_datasets**(*query*, ***kwargs*)

Search for datasets that match a query.

Parameters

- **query** (*dict*) -- Key value pairs representing the fields and values of the datasets to be included.
- ****kwargs** -- Any number of additional keyword arguments.

Returns Response dictionary

Return type (dict)

abstract DatasetEngine.**search_resources**(*query*, ***kwargs*)

Search for resources that match a query.

Parameters

- **query** (*dict*) -- Key value pairs representing the fields and values of the resources to be included.
- ****kwargs** (*kwargs, optional*) -- Any number of additional keyword arguments.

Returns Response dictionary

Return type (dict)

abstract DatasetEngine.**list_datasets**(***kwargs*)

List all datasets available from the dataset service.

Parameters ****kwargs** (*kwargs, optional*) -- Any number of additional keyword arguments.

Returns Response dictionary

Return type (dict)

Update Methods

abstract DatasetEngine.**update_dataset**(*dataset_id*, ***kwargs*)

Update an existing dataset.

Parameters

- **dataset_id** (*string*) -- Identifier of the dataset to update.
- ****kwargs** (*kwargs, optional*) -- Any number of additional keyword arguments.

Returns Response dictionary

Return type (dict)

abstract DatasetEngine.**update_resource**(*resource_id*, *url=None*, *file=None*, ***kwargs*)

Update an existing resource.

Parameters

- **resource_id** (*string*) -- Identifier of the resource to update.
- **url** (*string*) -- URL of resource to associate with resource.
- **file** (*string*) -- Path of file to upload as resource.
- ****kwargs** (*kwargs, optional*) -- Any number of additional keyword arguments.

Returns Response dictionary

Return type (dict)

Delete Methods

abstract DatasetEngine.**delete_dataset**(*dataset_id*, ***kwargs*)

Delete a dataset.

Parameters

- **dataset_id** (*string*) -- Identifier of the dataset to delete.
- ****kwargs** (*kwargs, optional*) -- Any number of additional keyword arguments.

Returns Response dictionary

Return type (dict)

abstract DatasetEngine.**delete_resource**(*resource_id*, ***kwargs*)

Delete a resource.

Parameters

- **resource_id** (*string*) -- Identifier of the resource to delete.
- ****kwargs** (*kwargs, optional*) -- Any number of additional keyword arguments.

Returns Response dictionary

Return type (dict)

CKAN Dataset Engine Reference

Last Updated: January 19, 2015

The following reference provides a summary the class used to define the CkanDatasetEngine objects.

class tethys_dataset_services.engines.**CkanDatasetEngine**(*endpoint*, *apikey=None*, *username=None*, *password=None*)

Definition for CKAN Dataset Engine objects.

create_dataset(*name*, *console=False*, ***kwargs*)

Create a new CKAN dataset.

Wrapper for the CKAN package_create API method. See the CKAN API docs for this method to see applicable options (http://docs.ckan.org/en/ckan-2.2/api.html).

Parameters

- **name** (*string*) -- The id or name of the resource to retrieve.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see CKAN docs).

Returns The response dictionary or None if an error occurs.

create_resource(*dataset_id*, *url=None*, *file=None*, *console=False*, ***kwargs*)

Create a new CKAN resource.

Wrapper for the CKAN resource_create API method. See the CKAN API docs for this method to see applicable options (http://docs.ckan.org/en/ckan-2.2/api.html).

Parameters

- **dataset_id** (*string*) -- The id or name of the dataset to to which the resource will be added.
- **url** (*string, optional*) -- URL for the resource that will be added to the dataset.

- **file** (*string, optional*) -- Absolute path to a file to upload for the resource.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see CKAN docs).

Returns The response dictionary or None if an error occurs.

delete_dataset(*dataset_id*, *console=False*, *file=None*, ***kwargs*)

Delete CKAN dataset

Wrapper for the CKAN package_delete API method. See the CKAN API docs for this method to see applicable options (http://docs.ckan.org/en/ckan-2.2/api.html).

Parameters

- **dataset_id** (*string*) -- The id or name of the dataset to delete.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see CKAN docs).

Returns The response dictionary or None if an error occurs.

delete_resource(*resource_id*, *console=False*, ***kwargs*)

Delete CKAN resource

Wrapper for the CKAN resource_delete API method. See the CKAN API docs for this method to see applicable options (http://docs.ckan.org/en/ckan-2.2/api.html).

Parameters

- **resource_id** (*string*) -- The id of the resource to delete.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see CKAN docs).

Returns The response dictionary or None if an error occurs.

download_dataset(*dataset_id*, *location=None*, *console=False*, ***kwargs*)

Downloads all resources in a dataset

Description

Parameters

- **dataset_id** (*string*) -- The id of the dataset to download.
- **location** (*string, optional*) -- Path to the location for the resource to be downloaded. Default is a subdirectory in the current directory named after the dataset. # noqa: E501
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments to pass to the get_dataset method (see CKAN docs).

Returns A list of the files that were downloaded.

download_resouce(*resource_id*, *location=None*, *local_file_name=None*, *console=False*, ***kwargs*)

Deprecated alias for download_resource method for backwards compatibility (the old method was misspelled).

Description

Parameters

- **resource_id** (*string*) -- The id of the resource to download.
- **location** (*string, optional*) -- Path to the location for the resource to be downloaded. Defaults to current directory. # noqa: E501
- **local_file_name** (*string, optional*) -- Name for downloaded file.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments to pass to the get_resource method (see CKAN docs).

Returns Path and name of the downloaded file.

download_resource(*resource_id*, *location=None*, *local_file_name=None*, *console=False*, ***kwargs*)

Download a resource from a resource id

Description

Parameters

- **resource_id** (*string*) -- The id of the resource to download.
- **location** (*string, optional*) -- Path to the location for the resource to be downloaded. Defaults to current directory. # noqa: E501
- **local_file_name** (*string, optional*) -- Name for downloaded file.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments to pass to the get_resource method (see CKAN docs).

Returns Path and name of the downloaded file.

get_dataset(*dataset_id*, *console=False*, ***kwargs*)

Retrieve CKAN dataset

Wrapper for the CKAN package_show API method. See the CKAN API docs for this method to see applicable options (http://docs.ckan.org/en/ckan-2.2/api.html).

Parameters

- **dataset_id** (*string*) -- The id or name of the dataset to retrieve.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see CKAN docs).

Returns The response dictionary or None if an error occurs.

get_resource(*resource_id*, *console=False*, ***kwargs*)

Retrieve CKAN resource

Wrapper for the CKAN resource_show API method. See the CKAN API docs for this method to see applicable options (http://docs.ckan.org/en/ckan-2.2/api.html).

Parameters

- **resource_id** (*string*) -- The id of the resource to retrieve.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see CKAN docs).

Returns The response dictionary or None if an error occurs.

list_datasets(*with_resources=False*, *console=False*, ***kwargs*)

List CKAN datasets.

Wrapper for the CKAN package_list and current_package_list_with_resources API methods. See the CKAN API docs for these methods to see applicable options (http://docs.ckan.org/en/ckan-2.2/api.html).

Parameters

- **with_resources** (*bool, optional*) -- Return a list of dataset dictionaries. Defaults to False.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see CKAN docs).

Returns A list of dataset names or a list of dataset dictionaries if with_resources is true.

Return type list

search_datasets(*query=None*, *filtered_query=None*, *console=False*, ***kwargs*)

Search CKAN datasets that match a query.

Wrapper for the CKAN search_datasets API method. See the CKAN API docs for this methods to see applicable options (http://docs.ckan.org/en/ckan-2.2/api.html).

Parameters

- **query** (*dict, optional if filtered_query set*) -- Key value pairs representing field and values to search for.
- **filtered_query** (*dict, optional if filtered_query set*) -- Key value pairs representing field and values to search for. # noqa: E501
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see CKAN docs).

Returns The response dictionary or None if an error occurs.

search_resources(*query*, *console=False*, ***kwargs*)

Search CKAN resources that match a query.

Wrapper for the CKAN search_resources API method. See the CKAN API docs for this methods to see applicable options (http://docs.ckan.org/en/ckan-2.2/api.html).

Parameters

- **query** (*dict*) -- Key value pairs representing field and values to search for.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see CKAN docs).

Returns The response dictionary or None if an error occurs.

property type
CKAN Dataset Engine Type

update_dataset(*dataset_id*, *console=False*, ***kwargs*)
Update CKAN dataset

Wrapper for the CKAN package_update API method. See the CKAN API docs for this method to see applicable options (http://docs.ckan.org/en/ckan-2.2/api.html).

Parameters

- **dataset_id** (*string*) -- The id or name of the dataset to update.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see CKAN docs).

Returns The response dictionary or None if an error occurs.

update_resource(*resource_id*, *url=None*, *file=None*, *console=False*, ***kwargs*)
Update CKAN resource

Wrapper for the CKAN resource_update API method. See the CKAN API docs for this method to see applicable options (http://docs.ckan.org/en/ckan-2.2/api.html).

Parameters

- **resource_id** (*string*) -- The id of the resource that will be updated.
- **url** (*string, optional*) -- URL of the resource that will be added to the dataset.
- **file** (*string, optional*) -- Absolute path to a file to upload for the resource.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see CKAN docs).

Returns The response dictionary or None if an error occurs.

validate()
Validate CKAN dataset engine. Will throw an error if not valid.

HydroShare Dataset Engine Reference

Last Updated: August 5, 2015

Warning: Coming Soon!

The following reference provides a summary the class used to define the `HydroShareDatasetEngine` objects.

class tethys_dataset_services.engines.**HydroShareDatasetEngine**(*endpoint*, *apikey=None*, *username=None*, *password=None*)

Definition for HydroShare Dataset Engine objects.

create_dataset(*name*, *console=False*, ***kwargs*)
Create a new HydroShare resource.

Parameters

- **name** (*string*) -- The id or name of the resource to retrieve.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see HydroShare docs).

Returns The response dictionary or None if an error occurs.

create_resource(*dataset_id*, *url=None*, *file=None*, *console=False*, ***kwargs*)
Create a new HydroShare file

Parameters

- **dataset_id** (*string*) -- The id or name of the dataset to to which the resource will be added.
- **url** (*string, optional*) -- URL for the resource that will be added to the dataset.
- **file** (*string, optional*) -- Absolute path to a file to upload for the resource.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see HydroShare docs).

Returns The response dictionary or None if an error occurs.

delete_dataset(*dataset_id*, *console=False*, ***kwargs*)
Delete HydroShare resource

Parameters

- **dataset_id** (*string*) -- The id or name of the dataset to delete.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see HydroShare docs).

Returns The response dictionary or None if an error occurs.

delete_resource(*resource_id*, *console=False*, ***kwargs*)
Delete HydroShare file.

Parameters

- **resource_id** (*string*) -- The id of the resource to delete.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see HydroShare docs).

Returns The response dictionary or None if an error occurs.

get_dataset(*dataset_id*, *console=False*, ***kwargs*)
Retrieve HydroShare resource

Parameters

- **dataset_id** (*string*) -- The id or name of the dataset to retrieve.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see HydroShare docs).

Returns The response dictionary or None if an error occurs.

get_resource(*resource_id*, *console=False*, ***kwargs*)

Retrieve HydroShare file

Parameters

- **resource_id** (*string*) -- The id of the resource to retrieve.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see HydroShare docs).

Returns The response dictionary or None if an error occurs.

list_datasets(*with_resources=False*, *console=False*, ***kwargs*)

List HydroShare resources

Parameters

- **with_resources** (*bool, optional*) -- Return a list of dataset dictionaries. Defaults to False.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see HydroShare docs).

Returns A list of dataset names or a list of dataset dictionaries if with_resources is true.

Return type list

search_datasets(*query*, *console=False*, ***kwargs*)

Search HydroShare resources that match a query.

Parameters

- **query** (*dict*) -- Key value pairs representing field and values to search for.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see HydroShare docs).

Returns The response dictionary or None if an error occurs.

search_resources(*query*, *console=False*, ***kwargs*)

Search HydroShare files that match a query.

Parameters

- **query** (*dict*) -- Key value pairs representing field and values to search for.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see HydroShare docs).

Returns The response dictionary or None if an error occurs.

property type
HydroShare Dataset Engine Type

update_dataset(*dataset_id*, *console=False*, ***kwargs*)
Update HydroShare resource

Parameters

- **dataset_id** (*string*) -- The id or name of the dataset to update.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see HydroShare docs).

Returns The response dictionary or None if an error occurs.

update_resource(*resource_id*, *url=None*, *file=None*, *console=False*, ***kwargs*)
Update HydroShare file

Parameters

- **resource_id** (*string*) -- The id of the resource that will be updated.
- **url** (*string, optional*) -- URL of the resource that will be added to the dataset.
- **file** (*string, optional*) -- Absolute path to a file to upload for the resource.
- **console** (*bool, optional*) -- Pretty print the result to the console for debugging. Defaults to False.
- ****kwargs** -- Any number of optional keyword arguments for the method (see HydroShare docs).

Returns The response dictionary or None if an error occurs.

Dataset Service Settings

Using dataset services in your app is accomplished by adding the `dataset_service_settings()` method to your *app class*, which is located in your *app configuration file* (`app.py`). This method should return a list or tuple of `DatasetServiceSetting`. For example:

```
from tethys_sdk.app_settings import DatasetServiceSetting

class MyFirstApp(TethysAppBase):
    """
    Tethys App Class for My First App.
    """
    ...
    def dataset_service_settings(self):
        """
        Example dataset_service_settings method.
        """
        ds_settings = (
            DatasetServiceSetting(
                name='primary_ckan',
                description='Primary CKAN service for app to use.',
                engine=DatasetServiceSetting.CKAN,
                required=True,
            ),
```

(continues on next page)

(continued from previous page)

```
            DatasetServiceSetting(
                name='hydroshare',
                description='HydroShare service for app to use.',
                engine=DatasetServiceSetting.HYDROSHARE,
                required=False
            )
        )

        return ds_settings
```

Caution: The ellipsis in the code block above indicates code that is not shown for brevity. **DO NOT COPY VERBATIM**.

Assign Dataset Service

The `DatasetServiceSetting` can be thought of as a socket for a connection to a `DatasetService`. Before we can do anything with the `DatasetServiceSetting` we need to "plug in" or assign a `DatasetService` to the setting. The `DatasetService` contains the connection information and can be used by multiple apps. Assigning a `DatasetService` is done through the Admin Interface of Tethys Portal as follows:

1. Create `DatasetService` if one does not already exist
    a. Access the Admin interface of Tethys Portal by clicking on the drop down menu next to your user name and selecting the "Site Admin" option.
    b. Scroll to the **Tethys Service** section of the Admin Interface and select the link titled **Dataset Services**.
    c. Click on the **Add Dataset Service** button.
    d. Fill in the connection information to the database server.
    e. Press the **Save** button to save the new `DatasetService`.

    Tip: You do not need to create a new `DatasetService` for each `DatasetServiceSetting` or each app. Apps and `DatasetServiceSettings` can share `DatasetServices`.

2. Navigate to App Settings Page
    a. Return to the Home page of the Admin Interface using the **Home** link in the breadcrumbs or as you did in step 1a.
    b. Scroll to the **Tethys Apps** section of the Admin Interface and select the **Installed Apps** linke.
    c. Select the link for your app from the list of installed apps.
3. Assign `DatasetService` to the appropriate `DatasetServiceSetting`
    a. Scroll to the **Dataset Service Settings** section and locate the `DatasetServiceSetting`.

    Note: If you don't see the `DatasetServiceSetting` in the list, uninstall the app and reinstall it again.

b. Assign the appropriate `DatasetService` to your `DatasetServiceSetting` using the drop down menu in the **Dataset Service** column.

c. Press the **Save** button at the bottom of the page to save your changes.

Note: During development you will assign the `DatasetService` setting yourself. However, when the app is installed in production, this steps is performed by the portal administrator upon installing your app, which may or may not be yourself.

Working with Dataset Services

After dataset services have been properly configured, you can use the services to store and retrieve data for your apps. The process involves the following steps:

1. Get a Dataset Service Engine

Call the `get_dataset_service()` method of the app class to get a `DatasetEngine`:

```
from my_first_app.app import MyFirstApp as app

ckan_engine = app.get_dataset_service('primary_ckan', as_engine=True)
```

You can also create a `DatasetEngine` object directly. This can be useful if you want to vary the credentials for dataset access frequently (e.g.: using user specific credentials):

```
from tethys_dataset_services.engines import CkanDatasetEngine

dataset_engine = CkanDatasetEngine(endpoint='http://www.example.com/api/3/action',␣
↪apikey='a-R3llY-n1Ce-@Pi-keY')
```

Caution: Take care not to store API keys, usernames, or passwords in the source files of your app--especially if the source is made public. This could compromise the security of the dataset service.

2. Use the Dataset Service Engine

After you have a `DatasetEngine`, simply call the desired method on it. All `DatasetEngine` methods return a dictionary with an item named `'success'` that contains a boolean. If the operation was successful, the value of `'success'` will be `True`, otherwise it will be `False`. If the value of `'success'` is `True`, the dictionary will also contain an item named `'result'` that will contain the results. If it is `False`, the dictionary will contain an item named `'error'` that will contain information about the error that occurred. This can be used for debugging purposes as illustrated in the following example:

```
from my_first_app.app import MyFirstApp as app

dataset_engine = app.get_dataset_service('primary_ckan', as_engine=True)

result = dataset_engine.list_datasets()

if result['success']:
```

(continues on next page)

(continued from previous page)

```
    dataset_list = result['result']

    for dataset in dataset_list:
        print(dataset)
else:
    print(result['error'])
```

Use the dataset service engines references above for descriptions of the methods available and examples.

Note: The HydroShare dataset engine uses OAuth 2.0 to authenticate and authorize interactions with the HydroShare via the REST API. This requires passing the `request` object as one of the arguments in `get_dataset_engine()` method call. Also, to ensure the user is connected to HydroShare, app developers must use the `ensure_oauth2()` decorator on any controllers that use the HydroShare dataset engine. For example:

```
from tethys_sdk.services import get_dataset_engine, ensure_oauth2
from .app import MyFirstApp as app

@ensure_oauth2('hydroshare')
def my_controller(request):
    """
    This is an example controller that uses the HydroShare API.
    """
    engine = app.get_dataset_service('hydroshare', request=request)

    response = engine.list_datasets()

    context = {}

    return render(request, 'my_first_app/home.html', context)
```

Spatial Dataset Services API

Last Updated: May 2017

Spatial dataset services are web services that can be used to store and publish file-based *spatial datasets* (e.g.: Shapefile and GeoTiff). The spatial datasets published using spatial dataset services are made available in a variety of formats, many of which or more web friendly than the native format (e.g.: PNG, JPEG, GeoJSON, and KML).

One example of a spatial dataset service is GeoServer, which is capable of storing and serving vector and raster datasets in several popular formats including Shapefiles, GeoTiff, ArcGrid and others. GeoServer serves the data in a variety of formats via the Open Geospatial Consortium (OGC) standards including Web Feature Service (WFS), Web Map Service (WMS), and Web Coverage Service (WCS).

Tethys app developers can use this Spatial Dataset Services API to store and access :term:` spatial datasets` for use in their apps and publish any resulting *datasets* their apps may produce.

Key Concepts

There are quite a few concepts that you should understand before working with GeoServer and spatial dataset services. Definitions of each are provided here for quick reference.

Resources are the spatial datasets. These can vary in format ranging from a single file or multiple files to database tables depending on the type resource.

Feature Type: is a type of *resource* containing vector data or data consisting of discreet features such as points, lines, or polygons and any tables of attributes that describe the features.

Coverage: is a type of *resource* containing raster data or numeric gridded data.

Layers: are *resources* that have been published. Layers associate styles and other settings with the *resource* that are needed to generate maps of the *resource* via OGC services.

Layer Groups: are preset groups of *layers* that can be served as WMS services as though they were one *layer*.

Stores: represent repositories of spatial datasets such as database tables or directories of shapefiles. A *store* containing only *feature types* is called a **Data Store** and a *store* containing only *coverages* is called a **Coverage Store**.

Workspaces: are arbitrary groupings of data to help with organization of the data. It would be a good idea to store all of the spatial datasets for your app in a workspace resembling the name of your app to avoid conflicts with other apps.

Styles: are a set of rules that dictate how a *layer* will be rendered when accessed via WMS. A *layer* may be associated with many styles and a style may be associated with many *layers*. Styles on GeoServer are written in Styled Layer Descriptor (SLD) format.

Styled Layer Descriptor (SLD): An XML-based markup language that can be used to specify how spatial datasets should be rendered. See GeoServer's SLD Cookbook for a good primer on SLD.

Web Feature Service (WFS): An OGC standard for exchanging vector data (i.e.: feature types) over the internet. WFS can be used to not only query for the features (points, lines, and polygons) but also the attributes associated with the features.

Web Coverage Service (WCS): An OGC standard for exchanging raster data (i.e.: coverages) over the internet. WCS is roughly the equivalent of WFS but for *coverages*, access to the raw coverage information, not just the image.

Web Mapping Service (WMS): An OGC standard for generating and exchanging maps of spatial data over the internet. WMS can be used to compose maps of several different spatial dataset sources and formats.

Spatial Dataset Engine References

All `SpatialDatasetEngine` objects implement a minimum set of base methods. However, some `SpatialDatasetEngine` objects may include additional methods that are unique to that `SpatialDatasetEngine` implementation and the arguments that each method accepts may vary slightly. Refer to the following references for the methods that are offered by each `SpatailDatasetEngine`.

Base Spatial Dataset Engine Reference

Last Updated: January 30, 2015

All `SpatialDatasetEngine` objects provide a minimum set of methods for interacting with layers and resources. Specifically, the methods allow the standard CRUD operations (Create, Read, Update, Delete) for both layers and resources.

All `SpatialSpatialDatasetEngine` methods return a dictionary called the response dictionary. The Response dictionary contains an item named 'success' that is a boolean indicating whether the operation was successful or not. If 'success' is `True`, then the the dictionary will also have an item named 'result' that contains the result of the operation. If 'success' is `False`, then the Response dictionary will contain an item called 'error' with information about what went wrong.

The following reference provides a summary of the base methods and properties provided by all `SpatialDatasetEngine` objects.

Properties

`SpatialDatasetEngine.` **endpoint** (string): URL for the spatial dataset service API endpoint.

`SpatialDatasetEngine.` **apikey** (string, optional): API key may be used for authorization.

`SpatialDatasetEngine.` **username** (string, optional): Username key may be used for authorization.

`SpatialDatasetEngine.` **password** (string, optional): Password key may be used for authorization.

`SpatialDatasetEngine.` **type** (string, readonly): Identifies the type of `SpatialDatasetEngine` object.

Create Methods

`SpatialDatasetEngine.`**`create_resource`**(*resource_id*, ***kwargs*)

Create a new resource.

Parameters

- **`resource_id`** (*`string`*) -- Identifier of the resource to create.
- **`**kwargs`** (*`kwargs, optional`*) -- Any number of additional keyword arguments.

Returns Response dictionary

Return type (dict)

`SpatialDatasetEngine.`**`create_layer`**(*layer_id*)

Create a new layer.

Parameters **`layer_id`** (*`string`*) -- Identifier of the layer to create.

Returns Response dictionary

Return type (dict)

Read Methods

abstract SpatialDatasetEngine.**get_resource**(*resource_id*)
Retrieve a resource object.
Parameters **resource_id** (*string*) -- Identifier of the dataset to retrieve.
Returns Response dictionary
Return type (dict)

abstract SpatialDatasetEngine.**get_layer**(*layer_id*)
Retrieve a single layer object.
Parameters **layer_id** (*string*) -- Identifier of the layer to retrieve.
Returns Response dictionary
Return type (dict)

abstract SpatialDatasetEngine.**list_resources**()
List all resources available from the spatial dataset service.
Returns Response dictionary
Return type (dict)

abstract SpatialDatasetEngine.**list_layers**()
List all layers available from the spatial dataset service.
Returns Response dictionary
Return type (dict)

Update Methods

abstract SpatialDatasetEngine.**update_resource**(*resource_id*, ***kwargs*)
Update an existing resource.
Parameters
- **resource_id** (*string*) -- Identifier of the resource to update.
- ****kwargs** (*kwargs, optional*) -- Any number of additional keyword arguments.

Returns Response dictionary
Return type (dict)

abstract SpatialDatasetEngine.**update_layer**(*layer_id*, ***kwargs*)
Update an existing layer.
Parameters
- **layer_id** (*string*) -- Identifier of the layer to update.
- ****kwargs** (*kwargs, optional*) -- Any number of additional keyword arguments.

Returns Response dictionary
Return type (dict)

Delete Methods

abstract SpatialDatasetEngine.**delete_resource**(*resource_id*)
Delete a resource.
Parameters **resource_id** (*string*) -- Identifier of the resource to delete.
Returns Response dictionary
Return type (dict)

abstract SpatialDatasetEngine.**delete_layer**(*layer_id*)
Delete a layer.

Parameters **layer_id** (*string*) -- Identifier of the layer to delete.
Returns Response dictionary
Return type (dict)

GeoServer Spatial Dataset Engine Reference

Last Updated: January 30, 2015

The following reference provides a summary the class used to define the `GeoServerSpatialDatasetEngine` objects.

class tethys_dataset_services.engines.**GeoServerSpatialDatasetEngine**(*endpoint*, *apikey=None*, *username=None*, *password=None*)

Definition for GeoServer Dataset Engine objects.

add_table_to_postgis_store(*store_id*, *table*, *debug=False*)

Add an existing postgis table as a feature resource to a postgis store that already exists.

Parameters

- **store_id** (*string*) -- Identifier for the store to add the resource to. Can be a store name or a workspace name combination (e.g.: "name" or "workspace:name"). Note that the workspace must be an existing workspace. If no workspace is given, the default workspace will be assigned. # noqa: E501
- **table** (*string*) -- Name of existing table to add as a feature resource. A layer will automatically be created for this resource. Both the resource and the layer will share the same name as the table. # noqa: E501
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

response = engine.add_table_to_postgis_store(store_id='workspace:store_name', table='table_name')

create_coverage_resource(*store_id*, *coverage_type*, *coverage_file=None*, *coverage_upload=None*, *coverage_name=None*, *overwrite=False*, *query_after_success=True*, *debug=False*)

Use this method to add coverage resources to GeoServer.

This method will result in the creation of three items: a coverage store, a coverage resource, and a layer. If store_id references a store that does not exist, it will be created. Unless coverage_name is specified, the coverage resource and the subsequent layer will be created with the same name as the image file that is uploaded. # noqa: E501

Parameters

- **store_id** (*string*) -- Identifier for the store to add the image to or to be created. Can be a name or a workspace name combination (e.g.: "name" or "workspace:name"). Note that the workspace must be an existing workspace. If no workspace is given, the default workspace will be assigned. # noqa: E501

- **coverage_type** (*string*) -- Type of coverage that is being created. Valid values include: 'geotiff', 'worldimage', 'imagemosaic', 'imagepyramid', 'gtopo30', 'arcgrid', 'grassgrid', 'erdasimg', 'aig', 'gif', 'png', 'jpeg', 'tiff', 'dted', 'rpftoc', 'rst', 'nitf', 'envihdr', 'mrsid', 'ehdr', 'ecw', 'netcdf', 'erdasimg', 'jp2mrsid'. # noqa: E501
- **coverage_file** (*string, optional*) -- Path to the coverage image or zip archive. Most files will require a .prj file with the Well Known Text definition of the projection. Zip this file up with the image and send the archive. # noqa: E501
- **coverage_upload** (*FileUpload list, optional*) -- A list of Django FileUpload objects containing a coverage file and .prj file or archive that have been uploaded via multipart/form-data form. # noqa: E501
- **coverage_name** (*string*) -- Name of the coverage resource and subsequent layer that are created. If unspecified, these will match the name of the image file that is uploaded. # noqa: E501
- **overwrite** (*bool, optional*) -- Overwrite the file if it already exists.
- **charset** (*string, optional*) -- Specify the character encoding of the file being uploaded (e.g.: ISO-8559-1)
- **query_after_success** (*bool, optional*) -- Query geoserver for resource objects after successful upload. Defaults to True.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Note If the type coverage being uploaded includes multiple files (e.g.: image, world file, projecttion file), they must be uploaded as a zip archive. Otherwise upload the single file. # noqa: E501

Returns Response dictionary
Return type (dict)

Examples

coverage_file = '/path/to/geotiff/example.zip'

response = engine.create_coverage_resource(store_id='workspace:store_name', coverage_file=coverage_file, coverage_type='geotiff') # noqa: E501

create_layer_group(*layer_group_id*, *layers*, *styles*, *bounds=None*, *debug=False*)

Create a layer group. The number of layers and the number of styles must be the same.

Parameters

- **layer_group_id** (*string*) -- Identifier of the layer group to create.
- **layers** (*iterable*) -- A list of layer names to be added to the group. Must be the same length as the styles list.
- **styles** (*iterable*) -- A list of style names to associate with each layer in the group. Must be the same length as the layers list. # noqa: #501
- **bounds** (*iterable*) -- A tuple representing the bounding box of the layer group (e.g.: ('-74.02722', '-73.907005', '40.684221', '40.878178', 'EPSG:4326')) # noqa: #501
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Examples

layers = ('layer1', 'layer2')

styles = ('style1', 'style2')

bounds = ('-74.02722', '-73.907005', '40.684221', '40.878178', 'EPSG:4326')

response = engine.create_layer_group(layer_group_id='layer_group_name', layers=layers, styles=styles, bounds=bounds) # noqa: E501

create_postgis_feature_resource(*store_id*, *host*, *port*, *database*, *user*, *password*, *table=None*, *debug=False*)

Use this method to link an existing PostGIS database to GeoServer as a feature store. Note that this method only works for data in vector formats. # noqa: E501

Parameters

- **store_id** (*string*) -- Identifier for the store to add the resource to. Can be a store name or a workspace name combination (e.g.: "name" or "workspace:name"). Note that the workspace must be an existing workspace. If no workspace is given, the default workspace will be assigned. # noqa: E501
- **host** (*string*) -- Host of the PostGIS database (e.g.: 'www.example.com').
- **port** (*string*) -- Port of the PostGIS database (e.g.: '5432')
- **database** (*string*) -- Name of the database.
- **user** (*string*) -- Database user that has access to the database.
- **password** (*string*) -- Password of database user.
- **table** (*string, optional*) -- Name of existing table to add as a feature resource to the newly created feature store. A layer will automatically be created for the feature resource as well. Both the layer and the resource will share the same name as the table. # noqa: E501
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Examples

With Table

response = engine.create_postgis_feature_resource(store_id='workspace:store_name', table='table_name', host='localhost', port='5432', database='database_name', user='user', password='pass') # noqa: E501

Without table

response = engine.create_postgis_resource(store_id='workspace:store_name', host='localhost', port='5432', database='database_name', user='user', password='pass') # noqa: E501

create_shapefile_resource(*store_id*, *shapefile_base=None*, *shapefile_zip=None*, *shapefile_upload=None*, *overwrite=False*, *charset=None*, *debug=False*)

Use this method to add shapefile resources to GeoServer.

This method will result in the creation of three items: a feature type store, a feature type resource, and a layer. If store_id references a store that does not exist, it will be created. The feature type resource and the subsequent layer will be created with the same name as the feature type store. Provide shapefile with either shapefile_base, shapefile_zip, or shapefile_upload arguments. # noqa: E501

Parameters

- **store_id** (*string*) -- Identifier for the store to add the resource to. Can be a store name or a workspace name combination (e.g.: "name" or "workspace:name"). Note that the workspace must be an existing workspace. If no workspace is given, the default workspace will be assigned. # noqa: E501
- **shapefile_base** (*string, optional*) -- Path to shapefile base name (e.g.: "/path/base" for shapefile at "/path/base.shp")
- **shapefile_zip** (*string, optional*) -- Path to a zip file containing the shapefile and side cars.
- **shapefile_upload** (*FileUpload list, optional*) -- A list of Django FileUpload objects containing a shapefile and side cars that have been uploaded via multipart/form-data form. # noqa: E501
- **overwrite** (*bool, optional*) -- Overwrite the file if it already exists.
- **charset** (*string, optional*) -- Specify the character encoding of the file being uploaded (e.g.: ISO-8559-1)
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

For example.shp (path to file but omit the .shp extension)

shapefile_base = "/path/to/shapefile/example"

response = engine.create_shapefile_resource(store_id='workspace:store_name', shapefile_base=shapefile_base)

Using zip

shapefile_zip = "/path/to/shapefile/example.zip"

response = engine.create_shapefile_resource(store_id='workspace:store_name', shapefile_zip=shapefile_zip)

Using upload

file_list = request.FILES.getlist('files')

response = engine.create_shapefile_resource(store_id='workspace:store_name', shapefile_upload=file_list)

create_sql_view(*feature_type_name*, *postgis_store_id*, *sql*, *geometry_column*, *geometry_type*, *geometry_srid=4326*, *default_style_id=None*, *key_column=None*, *parameters=None*, *debug=False*)

Create a new feature type configured as an SQL view.

Parameters

- **feature_type_name** (*string*) -- Name of the feature type and layer to be created.
- **postgis_store_id** (*string*) -- Identifier of existing postgis store with tables that will be queried by the sql view. Can be a store name or a workspace-name combination (e.g.: "name" or "workspace:name"). # noqa: E501
- **sql** (*string*) -- SQL that will be used to construct the sql view / virtual table.
- **geometry_column** (*string*) -- Name of the geometry column.
- **geometry_type** (*string*) -- Type of the geometry column (e.g. "Point", "LineString", "Polygon").
- **geometry_srid** (*string, optional*) -- EPSG spatial reference id of the geometry column. Defaults to 4326.
- **default_style_id** (*string, optional*) -- Identifier of a style to assign as the default style. Can be a style name or a workspace-name combination (e.g.: "name" or "workspace:name"). # noqa: E501
- **key_column** (*string, optional*) -- The name of the key column.
- **parameters** (*iterable, optional*) -- A list/tuple of tuple-triplets representing parameters in the form (name, default, regex_validation), (e.g.: (('variable', 'pressure', '^[w]+$'), ('simtime', '0:00:00', '^[w:]+$')) # noqa: E501,W605
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Example:

```
sql = "SELECT name, value, geometry FROM pipes"

response = engine.create_sql_view(
    feature_type_name='my_feature_type',
    postgis_store_id='my_workspace:my_postgis_store',
    sql=sql,
    geometry_column='geometry',
    geometry_type='LineString',
    geometry_srid=32144,
    default_style_id='my_workspace:pipes',
    debug=True
)
```

create_style(*style_id*, *sld*, *overwrite=False*, *debug=False*)

Create a new SLD style object.

Parameters

- **style_id** (*string*) -- Identifier of the style to create.
- **sld** (*string*) -- Styled Layer Descriptor string
- **overwrite** (*bool, optional*) -- Overwrite if style already exists. Defaults to False.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Examples

sld = '/path/to/style.sld'

sld_file = open(sld, 'r')

response = engine.create_style(style_id='fred', sld=sld_file.read(), debug=True)

sld_file.close()

create_workspace(*workspace_id*, *uri*, *debug=False*)

Create a new workspace.

Parameters

- **workspace_id** (*string*) -- Identifier of the workspace to create. Must be unique.
- **uri** (*string*) -- URI associated with your project. Does not need to be a real web URL, just a unique identifier. One suggestion is to append the URL of your project with the name of the workspace (e.g.: http:www.example.com/workspace-name). # noqa: E501
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Examples

response = engine.create_workspace(workspace_id='workspace_name', uri='www.example.com/workspace_name')

delete_layer(*layer_id*, *store_id=None*, *purge=False*, *recurse=False*, *debug=False*)

Delete a layer.

Parameters

- **layer_id** (*string*) -- Identifier of the layer to delete. Can be a name or a workspace-name combination (e.g.: "name" or "workspace:name"). # noqa: E501
- **store_id** (*string, optional*) -- Return only resources belonging to a certain store.
- **purge** (*bool, optional*) -- Purge if True.
- **recurse** (*bool, optional*) -- Delete recursively if True (i.e: delete layer groups it belongs to).
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Examples

response = engine.delete_layer('workspace:layer_name')

delete_layer_group(*layer_group_id*, *purge=False*, *recurse=False*, *debug=False*)
Delete a layer group.

Parameters

- **layer_group_id** (*string*) -- Identifier of the layer group to delete.
- **purge** (*bool, optional*) -- Purge if True.
- **recurse** (*bool, optional*) -- Delete recursively if True.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

response = engine.delete_layer_group('layer_group_name')

delete_resource(*resource_id*, *store_id*, *purge=False*, *recurse=False*, *debug=False*)
Delete a resource.

Parameters

- **resource_id** (*string*) -- Identifier of the resource to delete. Can be a name or a workspace-name combination (e.g.: "name" or "workspace:name"). # noqa: E501
- **store_id** (*string*) -- Return only resources belonging to a certain store.
- **purge** (*bool, optional*) -- Purge if True.
- **recurse** (*bool, optional*) -- Delete recursively any dependencies if True (i.e.: layers or layer groups it belongs to). # noqa: E501
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

response = engine.delete_resource('workspace:resource_name')

delete_store(*store_id*, *purge=False*, *recurse=False*, *debug=False*)
Delete a store.

Parameters

- **store_id** (*string*) -- Identifier of the store to delete.
- **purge** (*bool, optional*) -- Purge if True.
- **recurse** (*bool, optional*) -- Delete recursively if True.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

response = engine.delete_store('workspace:store_name')

delete_style(*style_id*, *purge=False*, *recurse=False*, *debug=False*)
Delete a style.

Parameters

- **style_id** (*string*) -- Identifier of the style to delete.
- **purge** (*bool, optional*) -- Purge if True.
- **recurse** (*bool, optional*) -- Delete recursively if True.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

response = engine.delete_resource('style_name')

delete_workspace(*workspace_id*, *purge=False*, *recurse=False*, *debug=False*)
Delete a workspace.

Parameters

- **workspace_id** (*string*) -- Identifier of the workspace to delete.
- **purge** (*bool, optional*) -- Purge if True.
- **recurse** (*bool, optional*) -- Delete recursively if True.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

response = engine.delete_resource('workspace_name')

get_layer(*layer_id*, *store_id=None*, *debug=False*)
Retrieve a layer object.

Parameters

- **layer_id** (*string*) -- Identifier of the layer to retrieve. Can be a name or a workspace-name combination (e.g.: "name" or "workspace:name"). # noqa: E501
- **store_id** (*string, optional*) -- Return only resources belonging to a certain store.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

response = engine.get_layer('layer_name')

response = engine.get_layer('workspace_name:layer_name')

get_layer_group(*layer_group_id*, *debug=False*)

Retrieve a layer group object.

Parameters

- **layer_group_id** (*string*) -- Identifier of the layer group to retrieve. Can be a name or a workspace-name combination (e.g.: "name" or "workspace:name"). # noqa: E501
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Examples

response = engine.get_layer_group('layer_group_name')

response = engine.get_layer_group('workspace_name:layer_group_name')

get_resource(*resource_id*, *store_id=None*, *debug=False*)

Retrieve a resource object.

Parameters

- **resource_id** (*string*) -- Identifier of the resource to retrieve. Can be a name or a workspace-name combination (e.g.: "name" or "workspace:name"). # noqa: E501
- **store** (*string, optional*) -- Get resource from this store.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Examples

response = engine.get_resource('example_workspace:resource_name')

response = engine.get_resource('resource_name', store='example_store')

get_store(*store_id*, *debug=False*)

Retrieve a store object.

Parameters

- **store_id** (*string*) -- Identifier of the store to retrieve. Can be a name or a workspace-name combination (e.g.: "name" or "workspace:name"). # noqa: E501
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Examples

response = engine.get_store('store_name')

response = engine.get_store('workspace_name:store_name')

get_style(*style_id*, *debug=False*)

Retrieve a style object.

Parameters

- **style_id** (*string*) -- Identifier of the style to retrieve.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Examples

response = engine.get_style('style_name')

get_workspace(*workspace_id*, *debug=False*)

Retrieve a workspace object.

Parameters

- **workspace_id** (*string*) -- Identifier of the workspace to retrieve.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Examples

response = engine.get_workspace('workspace_name')

link_sqlalchemy_db_to_geoserver(*store_id*, *sqlalchemy_engine*, *docker=False*, *debug=False*, *docker_ip_address='172.17.0.1'*)

Helper function to simplify linking postgis databases to geoservers using the sqlalchemy engine object.

Parameters

- **store_id** (*string*) -- Identifier for the store to add the resource to. Can be a store name or a workspace name combination (e.g.: "name" or "workspace:name"). Note that the workspace must be an existing workspace. If no workspace is given, the default workspace will be assigned. # noqa: E501
- **sqlalchemy_engine** (*sqlalchemy_engine*) -- An SQLAlchemy engine object.
- **docker** (*bool, optional*) -- Set to True if the database and geoserver are running in a Docker container. Defaults to False. # noqa: E501
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.
- **docker_ip_address** (*str, optional*) -- Override the docker network ip address. Defaults to '172.17.41.1'.

Returns Response dictionary

Return type (dict)

list_layer_groups(*with_properties=False*, *debug=False*)
List the names of all layer groups available from the spatial dataset service.

Parameters

- **with_properties** (*bool, optional*) -- Return list of layer group dictionaries instead of a list of layer group names. # noqa: E501
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

response = engine.list_layer_groups()

response = engine.list_layer_groups(with_properties=True)

list_layers(*with_properties=False*, *debug=False*)
List names of all layers available from the spatial dataset service.

Parameters

- **with_properties** (*bool, optional*) -- Return list of layer dictionaries instead of a list of layer names.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

response = engine.list_layers()

response = engine.list_layers(with_properties=True)

list_resources(*with_properties=False*, *store=None*, *workspace=None*, *debug=False*)
List the names of all resources available from the spatial dataset service.

Parameters

- **with_properties** (*bool, optional*) -- Return list of resource dictionaries instead of a list of resource names.
- **store** (*string, optional*) -- Return only resources belonging to a certain store.
- **workspace** (*string, optional*) -- Return only resources belonging to a certain workspace.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

response = engine.list_resource()

response = engine.list_resource(store="example_store")

response = engine.list_resource(with_properties=True, workspace="example_workspace")

list_stores(*workspace=None*, *with_properties=False*, *debug=False*)
List the names of all stores available from the spatial dataset service.

Parameters

- **workspace** (*string, optional*) -- List long stores belonging to this workspace.
- **with_properties** (*bool, optional*) -- Return list of store dictionaries instead of a list of store names.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

response = engine.list_stores()

response = engine.list_stores(workspace='example_workspace", with_properties=True)

list_styles(*workspace=None*, *with_properties=False*, *debug=False*)
List the names of all styles available from the spatial dataset service.

Parameters

- **workspace** (*string*) -- Return only resources belonging to a certain workspace.
- **with_properties** (*bool, optional*) -- Return list of style dictionaries instead of a list of style names.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary
Return type (dict)

Examples

response = engine.list_styles()

response = engine.list_styles(with_properties=True)

list_workspaces(*with_properties=False*, *debug=False*)
List the names of all workspaces available from the spatial dataset service.

Parameters

- **with_properties** (*bool, optional*) -- Return list of workspace dictionaries instead of a list of workspace names.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Examples

response = engine.list_workspaces()

response = engine.list_workspaces(with_properties=True)

property type
GeoServer Spatial Dataset Type

update_layer(*layer_id*, *debug=False*, ***kwargs*)
Update an existing layer.

Parameters

- **layer_id** (*string*) -- Identifier of the layer to update. Can be a name or a workspace-name combination (e.g.: "name" or "workspace:name"). # noqa: E501
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.
- ****kwargs** (*kwargs, optional*) -- Key value pairs representing the attributes and values to change.

Returns Response dictionary
Return type (dict)

Examples

updated_layer = engine.update_layer(layer_id='workspace:layer_name', default_style='style1', styles=['style1', 'style2']) # noqa: E501

update_layer_group(*layer_group_id*, *debug=False*, ***kwargs*)
Update an existing layer. If modifying the layers, ensure the number of layers and the number of styles are the same.

Parameters

- **layer_group_id** (*string*) -- Identifier of the layer group to update.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.
- ****kwargs** (*kwargs, optional*) -- Key value pairs representing the attributes and values to change

Returns Response dictionary
Return type (dict)

Examples

updated_layer_group = engine.update_layer_group(layer_group_id='layer_group_name', layers=['layer1', 'layer2'], styles=['style1', 'style2']) # noqa: E501

update_resource(*resource_id*, *store=None*, *debug=False*, ***kwargs*)
Update an existing resource.

Parameters

- **resource_id** (*string*) -- Identifier of the resource to update. Can be a name or a workspace-name combination (e.g.: "name" or "workspace:name"). # noqa: E501
- **store** (*string, optional*) -- Update a resource in this store.
- **debug** (*bool, optional*) -- Pretty print the response dictionary to the console for debugging. Defaults to False.
- ****kwargs** (*kwargs, optional*) -- Key value pairs representing the attributes and values to change.

Returns Response dictionary
Return type (dict)

Examples

response = engine.update_resource(resource_id='workspace:resource_name', enabled=False, title='New Title')

validate()
Validate the GeoServer spatial dataset engine. Will throw and error if not valid.

Spatial Dataset Service Settings

Using dataset services in your app is accomplished by adding the `spatial_dataset_service_settings()` method to your *app class*, which is located in your *app configuration file* (`app.py`). This method should return a list or tuple of `SpatialDatasetServiceSetting`. For example:

```
from tethys_sdk.app_settings import SpatialDatasetServiceSetting

class MyFirstApp(TethysAppBase):
    """
    Tethys App Class for My First App.
    """
    ...
    def spatial_dataset_service_settings(self):
        """
        Example spatial_dataset_service_settings method.
        """
        sds_settings = (
            SpatialDatasetServiceSetting(
                name='primary_geoserver',
                description='spatial dataset service for app to use',
                engine=SpatialDatasetServiceSetting.GEOSERVER,
                required=True,
            ),
        )

        return sds_settings
```

Caution: The ellipsis in the code block above indicates code that is not shown for brevity. **DO NOT COPY VERBATIM**.

Assign Spatial Dataset Service

The `SpatialDatasetServiceSetting` can be thought of as a socket for a connection to a `SpatialDatasetService`. Before we can do anything with the `SpatialDatasetServiceSetting` we need to "plug in" or assign a `SpatialDatasetService` to the setting. The `SpatialDatasetService` contains the connection information and can be used by multiple apps. Assigning a `SpatialDatasetService` is done through the Admin Interface of Tethys Portal as follows:

1. Create `SpatialDatasetService` if one does not already exist
    a. Access the Admin interface of Tethys Portal by clicking on the drop down menu next to your user name and selecting the "Site Admin" option.
    b. Scroll to the **Tethys Service** section of the Admin Interface and select the link titled **Spatial Dataset Services**.
    c. Click on the **Add Spatial Dataset Service** button.
    d. Fill in the connection information to the database server.
    e. Press the **Save** button to save the new `SpatialDatasetService`.

    Tip: You do not need to create a new `SpatialDatasetService` for each `SpatialDatasetServiceSetting` or each app. Apps and `SpatialDatasetServiceSettings` can share `DatasetServices`.

2. Navigate to App Settings Page
    a. Return to the Home page of the Admin Interface using the **Home** link in the breadcrumbs or as you did in step 1a.
    b. Scroll to the **Tethys Apps** section of the Admin Interface and select the **Installed Apps** linke.
    c. Select the link for your app from the list of installed apps.
3. Assign `SpatialDatasetService` to the appropriate `SpatialDatasetServiceSetting`
    a. Scroll to the **Spatial Dataset Services Settings** section and locate the `SpatialDatasetServiceSetting`.

    Note: If you don't see the `SpatialDatasetServiceSetting` in the list, uninstall the app and reinstall it again.

    b. Assign the appropriate `SpatialDatasetService` to your `SpatialDatasetServiceSetting` using the drop down menu in the **Spatial Dataset Service** column.
    c. Press the **Save** button at the bottom of the page to save your changes.

Note: During development you will assign the `SpatialDatasetService` setting yourself. However, when the app is installed in production, this steps is performed by the portal administrator upon installing your app, which may or may not be yourself.

Working with Spatial Dataset Services

After spatial dataset services have been properly configured, you can use the services to store, publish, and retrieve data for your apps. This process typically involves the following steps:

1. Get a Spatial Dataset Engine

Call the `get_spatial_dataset_service()` method of the app class to get a `SpatialDatasetEngine`:

```
from my_first_app.app import MyFirstApp as app

geoserver_engine = app.get_spatial_dataset_engine('primary_geoserver', as_engine=True)
```

You can also create a `SpatialDatasetEngine` object directly. This can be useful if you want to vary the credentials for dataset access frequently (e.g.: using user specific credentials):

```
from tethys_dataset_services.engines import GeoServerSpatialDatasetEngine

spatial_dataset_engine = GeoServerSpatialDatasetEngine(endpoint='http://www.example.
↪com/geoserver/rest', username='admin', password='geoserver')
```

Caution: Take care not to store API keys, usernames, or passwords in the source files of your app--especially if the source code is made public. This could compromise the security of the spatial dataset service.

2. Use the Spatial Dataset Engine

After you have a `SpatialDatasetEngine` object, simply call the desired method on it. All `SpatialDatasetEngine` methods return a dictionary with an item named 'success' that contains a boolean. If the operation was successful, 'success' will be true, otherwise it will be false. If 'success' is true, the dictionary will have an item named 'result' that will contain the results. If it is false, the dictionary will have an item named 'error' that will contain information about the error that occurred. This can be very useful for debugging and error catching purposes.

Consider the following example for uploading a shapefile to spatial dataset services:

```
from my_first_app.app import MyFirstApp as app

# First get an engine
engine = app.get_spatial_dataset_engine('primary_geoserver', as_engine=True)

# Create a workspace named after our app
engine.create_workspace(workspace_id='my_app', uri='http://www.example.com/apps/my-app
↪')

# Path to shapefile base for foo.shp, side cars files (e.g.: .shx, .dbf) will be
# gathered in addition to the .shp file.
shapefile_base = '/path/to/foo'

# Notice the workspace in the store_id parameter
result = dataset_engine.create_shapefile_resource(store_id='my_app:foo', shapefile_
↪base=shapefile_base)
```

(continues on next page)

(continued from previous page)

```
# Check if it was successful
if not result['success']:
    raise
```

A new shapefile Data Store will be created called 'foo' in workspace 'my_app' and a resource will be created for the shapefile called 'foo'. A layer will also automatically be configured for the new shapefile resource.

Tip: When you are learning how to use the spatial dataset engine methods, run the commands with the debug parameter set to true. This will automatically pretty print the result dictionary to the console so that you can inspect its contents:

```
# Example method with debug option
engine.list_layers(debug=True)
```

3. Get OGC Web Service URL

Publishing the spatial dataset with a spatial dataset service would be pointless without using the service to render the data on a map. This can be done by querying the data using the OGC web services WFS, WCS, or WMS. The dictionary that is returned when retrieving layers, layer groups, or resources will include a key for appropriate OGC services for the object returned. Feature type resources will provide a "wfs" key, coverage resources will provide a "wcs" key, and layers and layergroups will provide a "wms" key. The value will be another dictionary of OGC queries for different endpoints. For example:

```
# Get a feature type layer
response = engine.get_layer(layer_id='sf:roads', debug=True)

# Response dictionary includes "wms" key with links to maps in various formats
{'result': {'advertised': True,
            'attribution': None,
            'catalog': 'http://localhost:8181/geoserver/',
            'default_style': 'simple_roads',
            'enabled': None,
            'href': 'http://localhost:8181/geoserver/rest/layers/sf%3Aroads.xml',
            'name': 'sf:roads',
            'resource': 'sf:roads',
            'resource_type': 'layer',
            'styles': ['sf:line'],
            'wms': {'georss': 'http://localhost:8181/geoserver/wms?service=WMS&
↪version=1.1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&
↪tiled=no&srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,
↪4928063.398014731&width=731&height=512&format=rss',
                    'geotiff8': 'http://localhost:8181/geoserver/wms?service=WMS&
↪version=1.1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&
↪tiled=no&srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,
↪4928063.398014731&width=731&height=512&format=image/geotiff8',
                    'geptiff': 'http://localhost:8181/geoserver/wms?service=WMS&
↪version=1.1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&
↪tiled=no&srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,
↪4928063.398014731&width=731&height=512&format=image/geotiff',
                    'gif': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.
↪1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&tiled=no&
↪srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,4928063.
↪398014731&width=731&height=512&format=image/gif',
```

(continues on next page)

(continued from previous page)

```
                    'jpeg': 'http://localhost:8181/geoserver/wms?service=WMS&
↪version=1.1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&
↪tiled=no&srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,
↪4928063.398014731&width=731&height=512&format=image/jpeg',
                    'kml': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.
↪1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&tiled=no&
↪srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,4928063.
↪398014731&width=731&height=512&format=kml',
                    'kmz': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.
↪1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&tiled=no&
↪srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,4928063.
↪398014731&width=731&height=512&format=kmz',
                    'openlayers': 'http://localhost:8181/geoserver/wms?service=WMS&
↪version=1.1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&
↪tiled=no&srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,
↪4928063.398014731&width=731&height=512&format=application/openlayers',
                    'pdf': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.
↪1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&tiled=no&
↪srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,4928063.
↪398014731&width=731&height=512&format=application/pdf',
                    'png': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.
↪1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&tiled=no&
↪srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,4928063.
↪398014731&width=731&height=512&format=image/png',
                    'png8': 'http://localhost:8181/geoserver/wms?service=WMS&
↪version=1.1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&
↪tiled=no&srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,
↪4928063.398014731&width=731&height=512&format=image/png8',
                    'svg': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.
↪1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&tiled=no&
↪srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,4928063.
↪398014731&width=731&height=512&format=image/svg',
                    'tiff': 'http://localhost:8181/geoserver/wms?service=WMS&
↪version=1.1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&
↪tiled=no&srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,
↪4928063.398014731&width=731&height=512&format=image/tiff',
                    'tiff8': 'http://localhost:8181/geoserver/wms?service=WMS&
↪version=1.1.0&request=GetMap&layers=sf:roads&styles=simple_roads&transparent=true&
↪tiled=no&srs=EPSG:26713&bbox=589434.8564686741,4914006.337837095,609527.2102150217,
↪4928063.398014731&width=731&height=512&format=image/tiff8'}},
 'success': True}
```

These links could be passed on to a web mapping client like OpenLayers or Google Maps to render the map interactively on a web page. Note that the OGC mapping services are very powerful and the links provided represent only a simple query. You can construct custom OGC URLs queries without much difficulty. For excellent primers on WFS, WCS, and WMS with GeoServer, visit these links:

- GeoServer Web Feature Service Overview
- GeoServer Web Coverage Service Overview
- GeoServer Web Map Service Overview

Web Processing Services API

Last Updated: May 2017

Web Processing Services (WPS) are web services that can be used perform geoprocessing and other processing activities for apps. The Open Geospatial Consortium (OGC) has created the WPS interface standard that provides rules for how inputs and outputs for processing services should be formatted. Using the Web Processing Services API, you will be able to provide processing capabilities for your apps using any service that conforms to the OGC WPS standard. For convenience, the 52 North WPS is provided as part of the Tethys Platform software suite.

Web Processing Service Settings

Using web processing services in your app is accomplised by adding the `web_processing_service_settings()` method to your *app class*, which is located in your *app configuration file* (`app.py`). This method should return a list or tuple of `WebProcessingServiceSetting` objects. For example:

```
from tethys_sdk.app_settings import WebProcessingServiceSetting

class MyFirstApp(TethysAppBase):
    """
    Tethys App Class for My First App.
    """
    ...
    def web_processing_service_settings(self):
        """
        Example wps_services method.
        """
        wps_services = (
            WebProcessingServiceSetting(
                name='primary_52n',
                description='WPS service for app to use',
                required=True,
            ),
        )

        return wps_services
```

Caution: The ellipsis in the code block above indicates code that is not shown for brevity. **DO NOT COPY VERBATIM**.

Assign Web Processing Service

The `WebProcessingServiceSetting` can be thought of as a socket for a connection to a WPS. Before we can do anything with the `WebProcessingServiceSetting` we need to "plug in" or assign a `WebProcessingService` to the setting. The `WebProcessingService` contains the connection information and can be used by multiple apps. Assigning a `WebProcessingService` is done through the Admin Interface of Tethys Portal as follows:

1. Create `WebProcessingService` if one does not already exist
    a. Access the Admin interface of Tethys Portal by clicking on the drop down menu next to your user name and selecting the "Site Admin" option.

b. Scroll to the **Tethys Service** section of the Admin Interface and select the link titled **Web Processing Services**.

c. Click on the **Add Web Processing Services** button.

d. Fill in the connection information to the WPS server.

e. Press the **Save** button to save the new `WebProcessingService`.

Tip: You do not need to create a new `WebProcessingService` for each `WebProcessingServiceSetting` or each app. Apps and `WebProcessingServiceSettings` can share `WebProcessingServices`.

---

2. Navigate to App Settings Page

   a. Return to the Home page of the Admin Interface using the **Home** link in the breadcrumbs or as you did in step 1a.

   b. Scroll to the **Tethys Apps** section of the Admin Interface and select the **Installed Apps** linke.

   c. Select the link for your app from the list of installed apps.

3. Assign `WebProcessingService` to the appropriate `WebProcessingServiceSetting`

   a. Scroll to the **Web Processing Service Settings** section and locate the `WebProcessingServiceSetting`.

Tip: If you don't see the `WebProcessingServiceSetting` in the list, uninstall the app and reinstall it again.

---

   b. Assign the appropriate `WebProcessingService` to your `WebProcessingServiceSetting` using the drop down menu in the **Web Processing Service** column.

   c. Press the **Save** button at the bottom of the page to save your changes.

Note: During development you will assign the `WebProcessingService` setting yourself. However, when the app is installed in production, this steps is performed by the portal administrator upon installing your app, which may or may not be yourself.

Working with WPS Services in Apps

The Web Processing Service API is powered by OWSLib, a Python client that can be used to interact with OGC web services. For detailed explanations the WPS client provided by OWSLib, refer to the OWSLib WPS Documentation. This article only provides a basic introduction to working with the OWSLib WPS client.

Get a WPS Engine

Anytime you wish to use a WPS service in an app, you will need to obtain an `owslib.wps.WebProcessingService` engine object. This can be done by calling the `get_web_processing_service()` method of the app class:

```
from my_first_app.app import MyFirstApp as app

wps_engine = app.get_web_processing_service('primary_52n', as_engine=True)
```

Alternatively, you can create an `owslib.wps.WebProcessingService` engine object directly without using the convenience function. This can be useful if you want to vary the credentials for WPS service access frequently (e.g.: to provide user specific credentials).

```
from owslib.wps import WebProcessingService

wps_engine = WebProcessingService('http://www.example.com/wps/WebProcessingService',␣
↪verbose=False, skip_caps=True)
wps_engine.getcapabilities()
```

Using the WPS Engine

After you have retrieved a valid `owslib.wps.WebProcessingService` engine object, you can use it execute process requests. The following example illustrates how to execute the GRASS buffer process on a 52 North WPS:

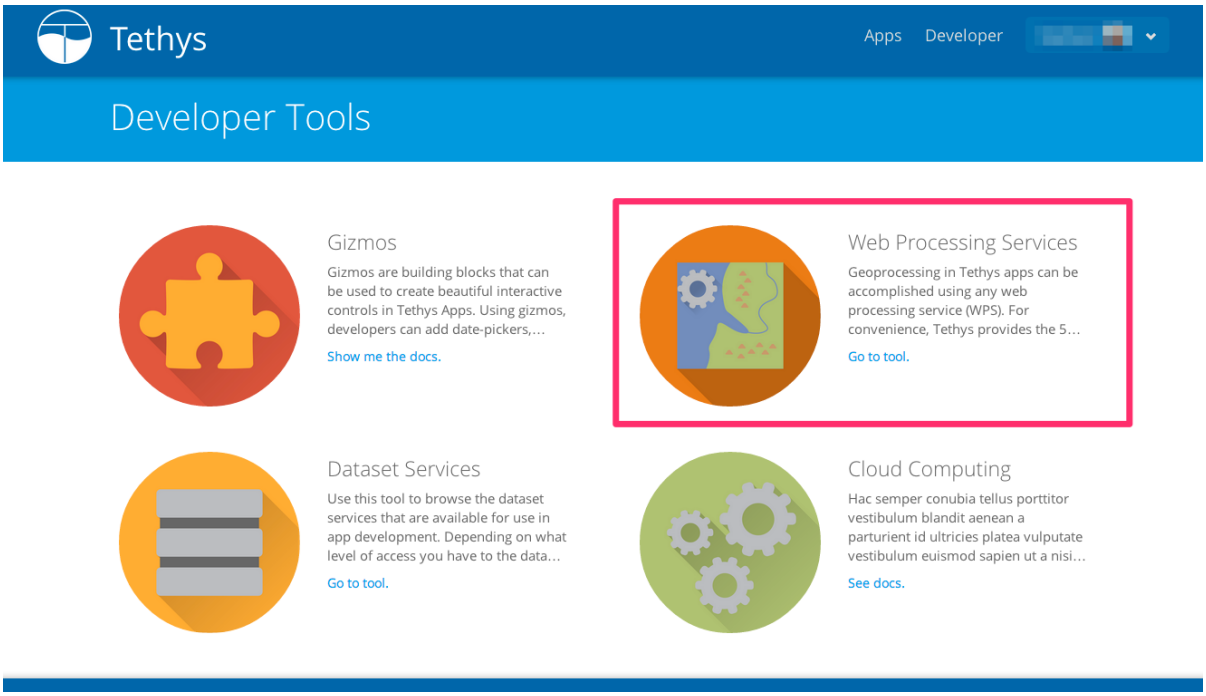
```
from owslib.wps import GMLMultiPolygonFeatureCollection

polygon = [(-102.8184, 39.5273), (-102.8184, 37.418), (-101.2363, 37.418), (-101.2363,
↪ 39.5273), (-102.8184, 39.5273)]
feature_collection = GMLMultiPolygonFeatureCollection( [polygon] )
process_id = 'v.buffer'
inputs = [ ('DISTANCE', 5.0),
           ('INPUT', feature_collection)
         ]
output = 'OUTPUT'
execution = wps_engine.execute(process_id, inputs, output)
monitorExecution(execution)
```

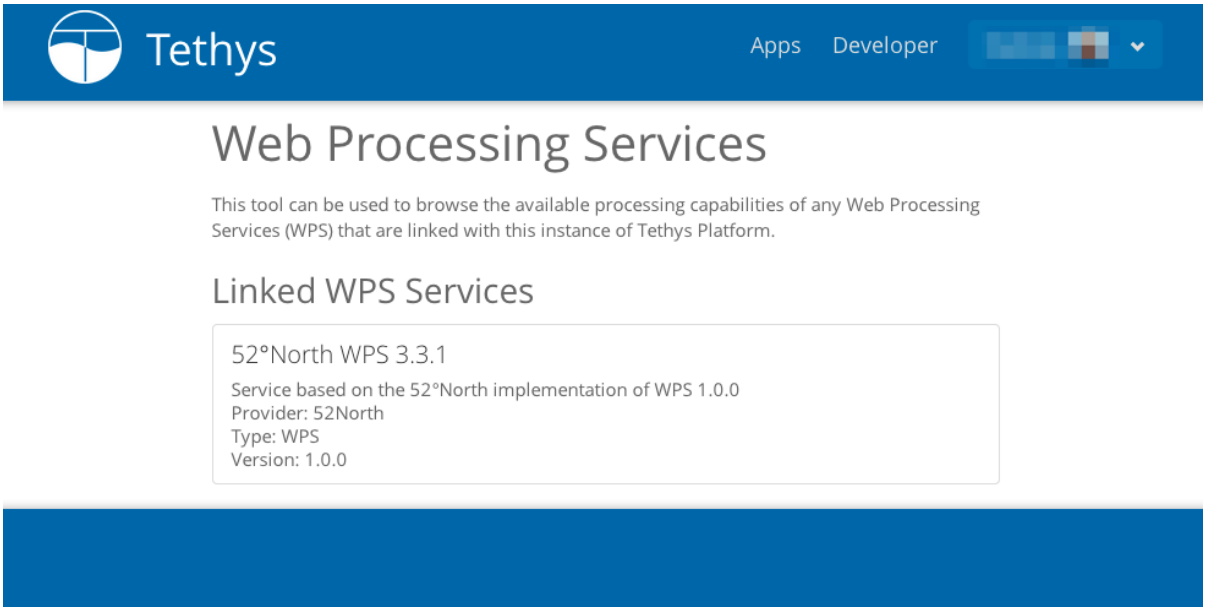
It is also possible to perform requests using data that are hosted on WFS servers, such as the GeoServer that is provided as part of the Tethys Platform software suite. See the OWSLib WPS Documentation for more details on how this is to be done.

Web Processing Service Developer Tool

Tethys Platform provides a developer tool that can be used to browse the sitewide WPS services and the processes that they provide. This tool is useful for formulating new process requests. To use the tool:

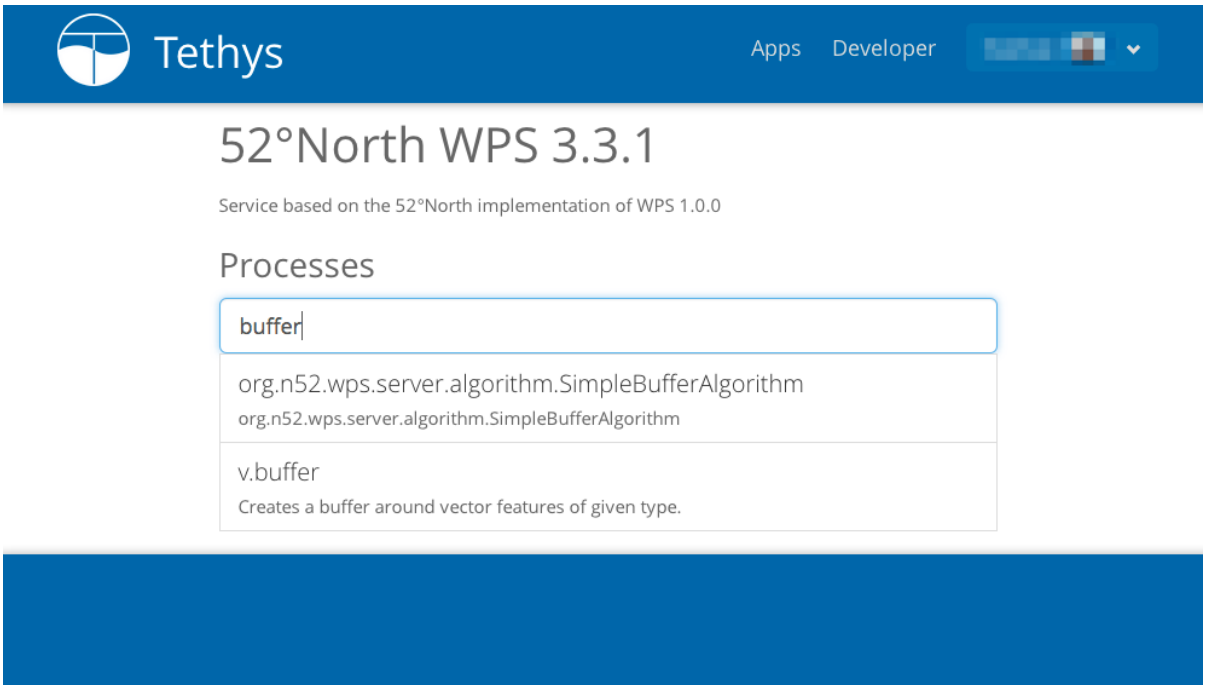
1. Browse to the Developer Tools page of your Tethys Platform by selecting the "Developer" link from the menu at the top of the page.
2. Select the tool titled "Web Processing Services".
3. Select a WPS service from the list of services that are linked with your Tethys Instance. If no WPS services are linked to your Tethys instance, follow the steps in Sitewide Configuration, above, to setup a WPS service.

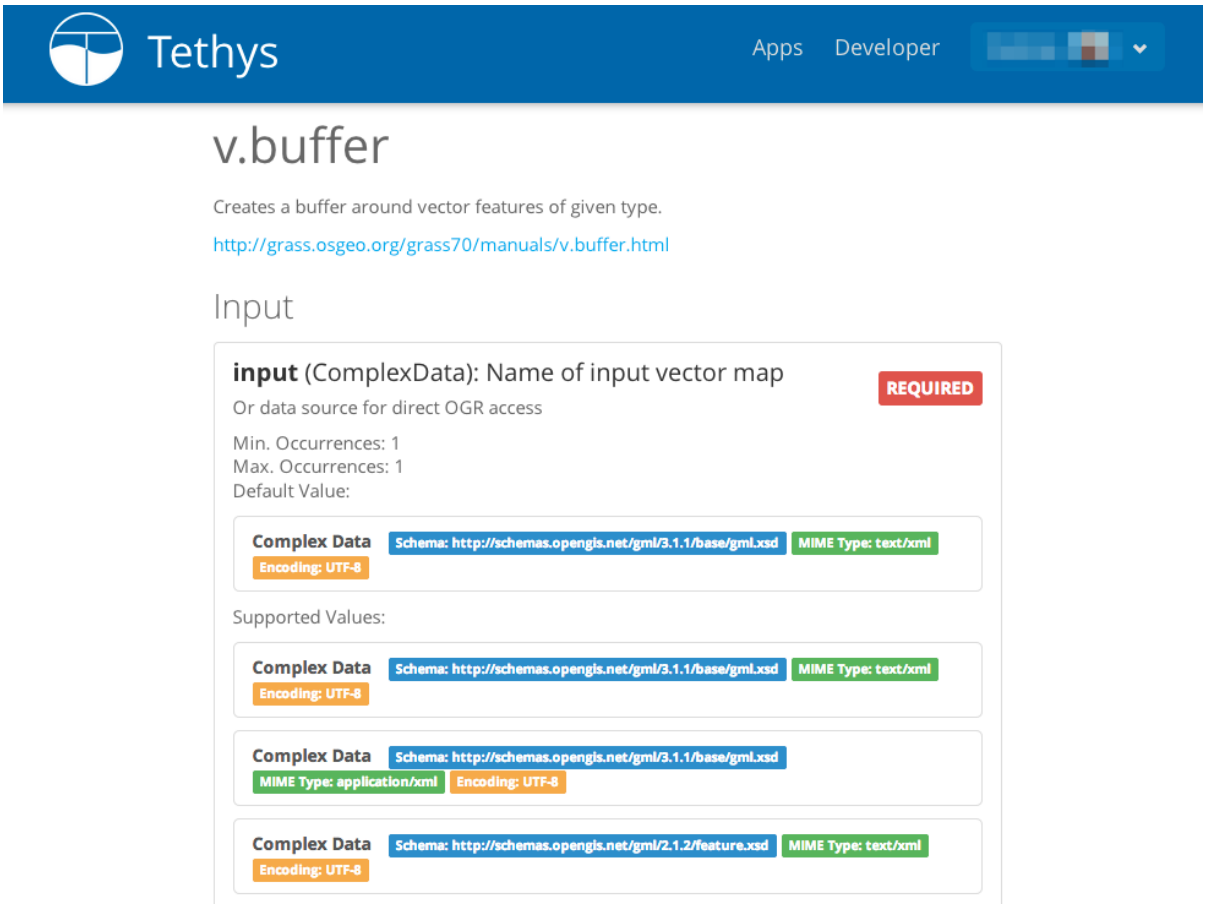




4. Select the process you wish to view.

A description of the process and the inputs and outputs will be displayed.


Tethys
Apps
Developer
52°North WPS 3.3.1
Service based on the 52°North implementation of WPS 1.0.0
Processes
buffer
org.n52.wps.server.algorithm.SimpleBufferAlgorithm
org.n52.wps.server.algorithm.SimpleBufferAlgorithm
v.buffer
Creates a buffer around vector features of given type.


Tethys
Apps
Developer
v.buffer
Creates a buffer around vector features of given type.
http://grass.osgeo.org/grass70/manuals/v.buffer.html
Input
input (ComplexData): Name of input vector map
REQUIRED
Or data source for direct OGR access
Min. Occurrences: 1
Max. Occurrences: 1
Default Value:
Complex Data
Schema: http://schemas.opengis.net/gml/3.1.1/base/gml.xsd
MIME Type: text/xml
Encoding: UTF-8
Supported Values:
Complex Data
Schema: http://schemas.opengis.net/gml/3.1.1/base/gml.xsd
MIME Type: text/xml
Encoding: UTF-8
Complex Data
Schema: http://schemas.opengis.net/gml/3.1.1/base/gml.xsd
MIME Type: application/xml
Encoding: UTF-8
Complex Data
Schema: http://schemas.opengis.net/gml/2.1.2/feature.xsd
MIME Type: text/xml
Encoding: UTF-8

1.6.14 Workspaces API

Last Updated: August 6, 2014

The Workspaces API makes it easy for you to create directories for storing files that your app operates on. This can be a tricky task for a web application, because of the multi-user, simultaneous-connection environment of the web. The Workspaces API provides a simple mechanism for creating and managing a global workspace for your app and individual workspaces for each user of your app to prevent unwanted overwrites and file lock conflicts.

Get a Workspace

The Workspaces API adds two methods to your *app class*, `get_app_workspace()` and `get_user_workspace()`, that can be used to retrieve with the global app workspace and the user workspaces, respectively. To use the Workspace API methods, import your *app class* from the *app configuration file* (`app.py`) and call the appropriate method on that class. Explanations of the methods and example usage follows.

get_app_workspace

classmethod TethysAppBase.**get_app_workspace**()

Get the file workspace (directory) for the app.

Returns An object representing the workspace.

Return type *tethys_apps.base.TethysWorkspace*

Example:

```
import os
from my_first_app.app import MyFirstApp as app

def a_controller(request):
    """
    Example controller that uses get_app_workspace() method.
    """
    # Retrieve the workspace
    app_workspace = app.get_app_workspace()
    new_file_path = os.path.join(app_workspace.path, 'new_file.txt')

    with open(new_file_path, 'w') as a_file:
        a_file.write('...')

    context = {}

    return render(request, 'my_first_app/template.html', context)
```

get_user_workspace

classmethod TethysAppBase.**get_user_workspace**(*user*)

Get the file workspace (directory) for the given User.

Parameters **user** (*User or HttpRequest*) -- User or request object.

Returns An object representing the workspace.

Return type *tethys_apps.base.TethysWorkspace*

Example:

```
import os
from my_first_app.app import MyFirstApp as app

def a_controller(request):
    """
    Example controller that uses get_user_workspace() method.
    """
    # Retrieve the workspace
    user_workspace = app.get_user_workspace(request.user)
    new_file_path = os.path.join(user_workspace.path, 'new_file.txt')

    with open(new_file_path, 'w') as a_file:
        a_file.write('...')

    context = {}

    return render(request, 'my_first_app/template.html', context)
```

Working with Workspaces

The two methods described above return a `TethysWorkspace` object that contains the path to the workspace and several convenience methods for working with the workspace. An explanation of the `TethysWorkspace` object and examples of it's usage is provided below.

TethysWorkspace Objects

class tethys_apps.base.**TethysWorkspace**(*path*)

Defines objects that represent file workspaces (directories) for apps and users.

path

The absolute path to the workspace directory. Cannot be overwritten.

Type str

clear(*exclude=[]*, *exclude_files=False*, *exclude_directories=False*)

Remove all files and directories in the workspace.

Parameters

- **exclude** (*iterable*) -- A list or tuple of file and directory names to exclude from clearing operation.
- **exclude_files** (*bool*) -- Excludes all files from clearing operation when True. Defaults to False.
- **exclude_directories** (*bool*) -- Excludes all directories from clearing operation when True. Defaults to False.

Examples:

```
# Clear everything
workspace.clear()

# Clear directories only
workspace.clear(exclude_files=True)

# Clear files only
workspace.clear(exclude_directories=True)
```

(continues on next page)

(continued from previous page)

```
# Clear all but specified files and directories
workspace.clear(exclude=['file1.txt', '/full/path/to/directory1', 'directory2
↪', '/full/path/to/file2.txt'])
```

directories(*full_path=False*)

Return a list of directories that are in the workspace.

Parameters **full_path** (*bool*) -- Returns list of directories with full path names when True. Defaults to False.

Returns A list of directories in the workspace.

Return type list

Examples:

```
# List directory names
workspace.directories()

# List full path directory names
workspace.directories(full_path=True)
```

files(*full_path=False*)

Return a list of files that are in the workspace.

Parameters **full_path** (*bool*) -- Returns list of files with full path names when True. Defaults to False.

Returns A list of files in the workspace.

Return type list

Examples:

```
# List file names
workspace.files()

# List full path file names
workspace.files(full_path=True)
```

remove(*item*)

Remove a file or directory from the workspace.

Parameters **item** (*str*) -- Name of the item to remove from the workspace.

Examples:

```
workspace.remove('file.txt')
workspace.remove('/full/path/to/file.txt')
workspace.remove('relative/path/to/file.txt')
workspace.remove('directory')
workspace.remove('/full/path/to/directory')
workspace.remove('relative/path/to/directory')
```

Note: Though you can specify relative paths, the `remove()` method will not allow you to back into other directories using "../" or similar notation. Futhermore, absolute paths given must contain the path of the workspace to be valid.

Centralize Workspaces

The Workspaces API includes a command, `collectworkspaces`, for moving all workspaces to a central location and symbolically linking them back to the app project directories. This is especially useful for production where the administrator may want to locate workspace content on a mounted drive to optimize storage. A brief explanation of how to use this command will follow. Refer to the *Command Line Interface* documentation for details about the `collectworkspaces` command.

Setting

To enable centralized workspaces create a directory for the workspaces and specify its path in the `settings.py` file using the `TETHYS_WORKSPACES_ROOT` setting.

```
TETHYS_WORKSPACES_ROOT = '/var/www/tethys/workspaces'
```

Command

Run the `collectworkspaces` command to automatically move all of the workspace directories to the `TETHYS_WORKSPACES_ROOT` directory and symbolically link them back. You will need to run this command each time you install new apps.

```
$ tethys manage collectworkspaces
```

Tip: A convenience command is provided called `collectall` that can be used to run both the `collectstatic` and the `collectworkspaces` commands:

```
$ tethys manage collectall
```

1.7 Tethys Portal

Last Updated: December 14, 2015

Tethys Portal is the Django web site provided by Tethys Platform that acts as the runtime environment for apps. It leverages the capabilities of Django to provide the core website functionality that is often taken for granted in modern web applications. A description of the primary capabilities of Tethys Portal is provided in this section.

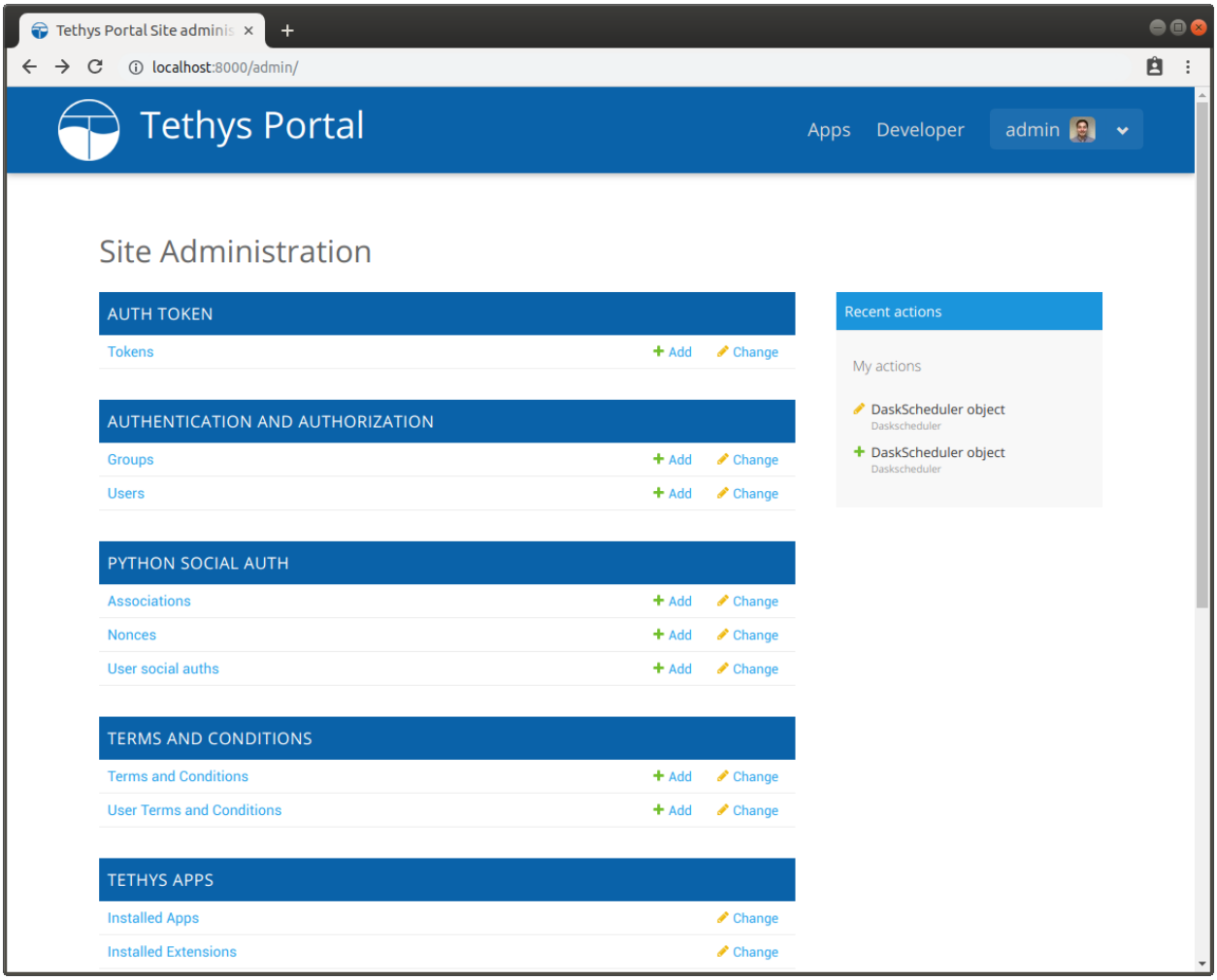
1.7.1 Administrator Pages

Last Updated: December 2018

Tethys Portal includes administration pages that can be used to manage the website (see Figure 1). The administration dashboard is only available to administrator users (staff users). You should have created a default administrator user when you installed Tethys Platform. If you are logged in as an administrator, you will be able to access the administrator dashboard by selecting the "Site Admin" option from the user drop down menu in the top right-hand corner of the page (when you are not in an app).

Figure 1. Administrator dashboard for Tethys Portal.

Note: If you did not create an administrator user during installation, run the following command in the terminal:


Tethys Portal Site adminis
localhost:8000/admin/
Tethys Portal
Apps
Developer
admin
Site Administration
AUTH TOKEN
Tokens
Add
Change
AUTHENTICATION AND AUTHORIZATION
Groups
Users
PYTHON SOCIAL AUTH
Associations
Nonces
User social auths
TERMS AND CONDITIONS
Terms and Conditions
User Terms and Conditions
TETHYS APPS
Installed Apps
Installed Extensions
Recent actions
My actions
DaskScheduler object
Daskscheduler

```
$ tethys manage createsuperuser
```

Auth Token

Tethys REST API tokens for individual users can be managed using the `Tokens` link under the `AUTH TOKEN` heading (see Figure 2).

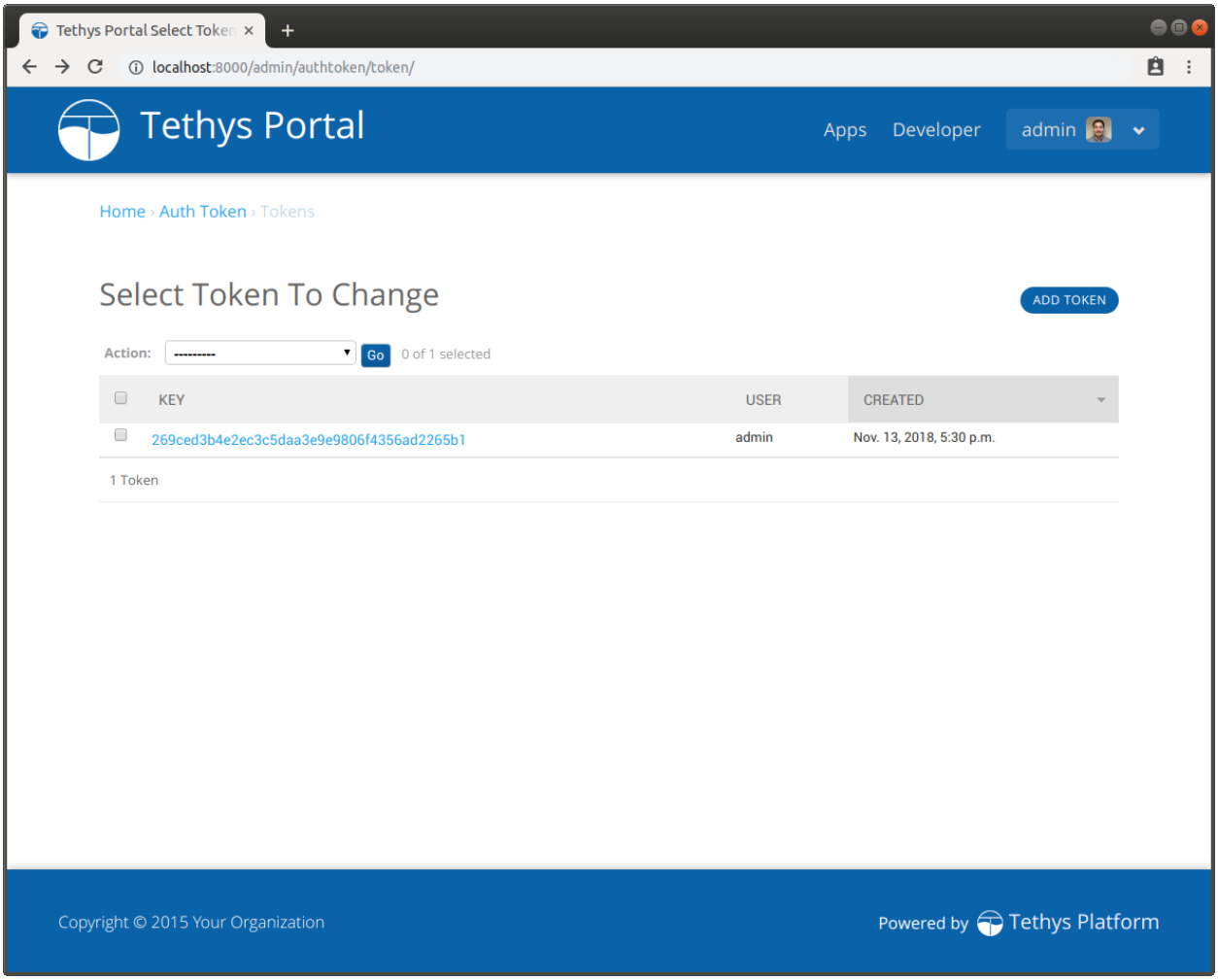


Figure 2. Auth Token management page for Tethys Portal.

Authentication and Authorization

Permissions and users can be managed from the administrator dashboard using `Users` link under the `AUTHENTICATION AND AUTHORIZATION` heading. Figure 3 shows an example of the user management page for a user named John.

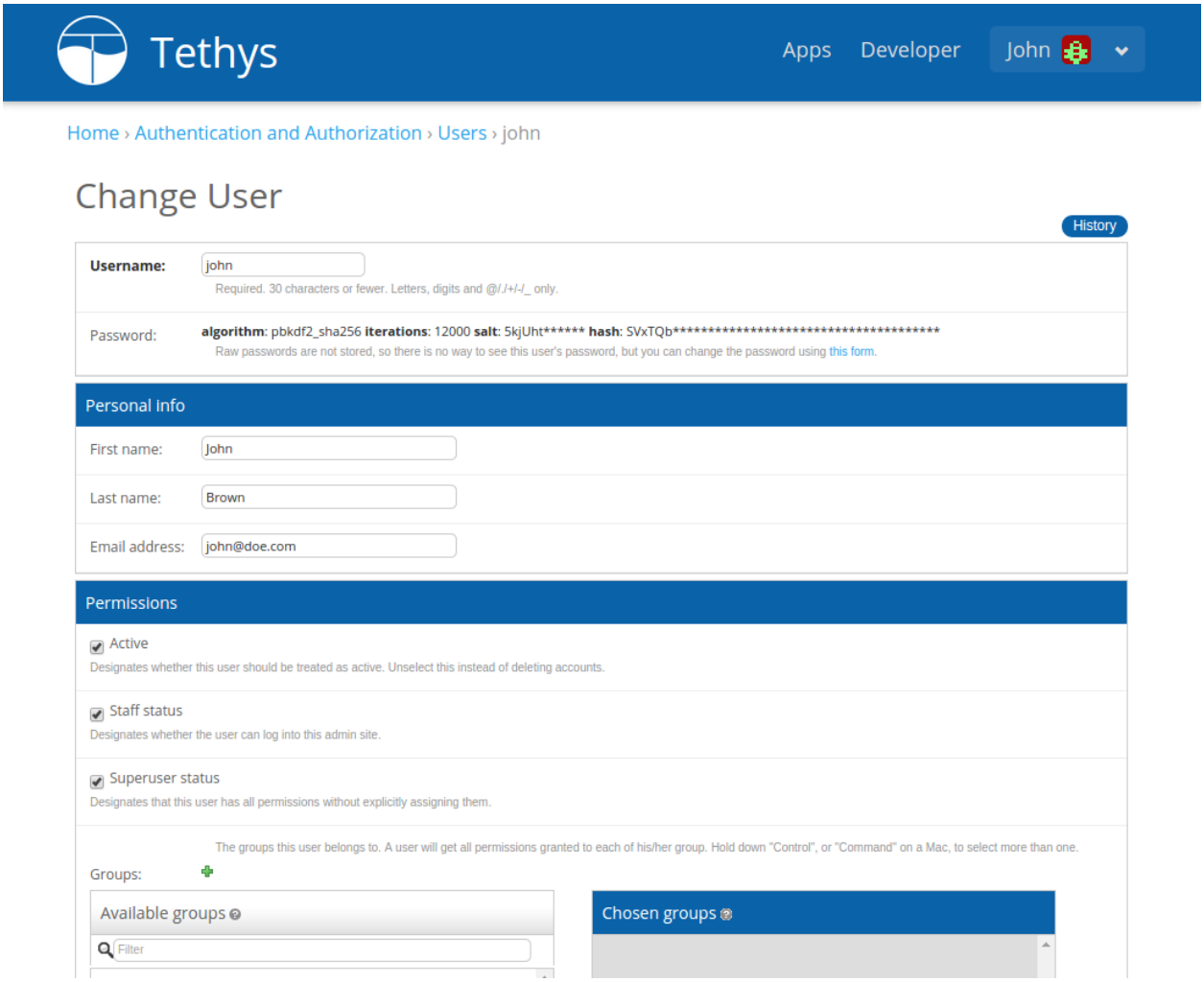


Figure 3. User management for Tethys Portal.

Assign App Permission Groups

To assign an app permission group to a user, select the desired user and locate the `Groups` dialog under the `Permissions` heading of the `Change User` page. All app permission groups will appear in the `Available Groups` list box. Assigning the permission group is done by moving the permission group to the `Chosen Groups` list box. Although the permissions may also appear in the `User Permissions` list box below, they cannot be properly assigned in the `Change User` dialog.

Assign App Permissions

To assign a singluar app permission to a user, return to the administrator dashboard and navigate to the `Installed Apps` link under the `Tethys Apps` heading. Select the link with the app name from the list. In the upper right corner of the `Change Tethys App` page click the `Object Permissions` button. On the `Object Permissions` page you can assign app specific permissions to a user by entering the username in the `User Identification` field and press the `Manage user` button. Incidentally, you can also manage the app permissions groups from the `Object Permisions` page, but changes will be overridden the next time the server restarts and permissions are synced from the app.

Note: Since assigning the individual app permissions is so difficult, we highly recommend that you use the app permission groups to group app permissions and then assign the permission groups to the users using the `Change User` page.

Anonymous User

The `AnonymousUser` can be used to assign permissions and permission groups to users who are not logged in. This means that you can define permissions for each feature of your app, but then assign them all to the `AnonymousUser` if you want the app to be publicly accessible.

Python Social Auth

Tethys leverages the excellent Python Social Auth to provide support for authenticating with popular servies such as Facebook, Google, LinkedIn, and HydroShare. The links under the `PYTHON SOCIAL AUTH` heading can be used to manually manage the social associations and data that is linked to users when they authenticate using Python Social Auth.

Tip: For more detailed information on using Python Social Auth in Tethys see the *Social Authentication* documentation.

Terms and Conditions

Portal administrators can manage and enforce portal wide terms and conditions and other legal documents via the administrator pages.

Use the `Terms and Conditions` link to create new legal documents (see Figure 4). To issue an update to a particular document, create a new entry with the same slug (e.g. 'site-terms'), but a different version number (e.g.: 1.10). This allows you to track multiple versions of the legal document and which users have accepted each. The document will not become active until the `Date active` field has been set and the date has past.

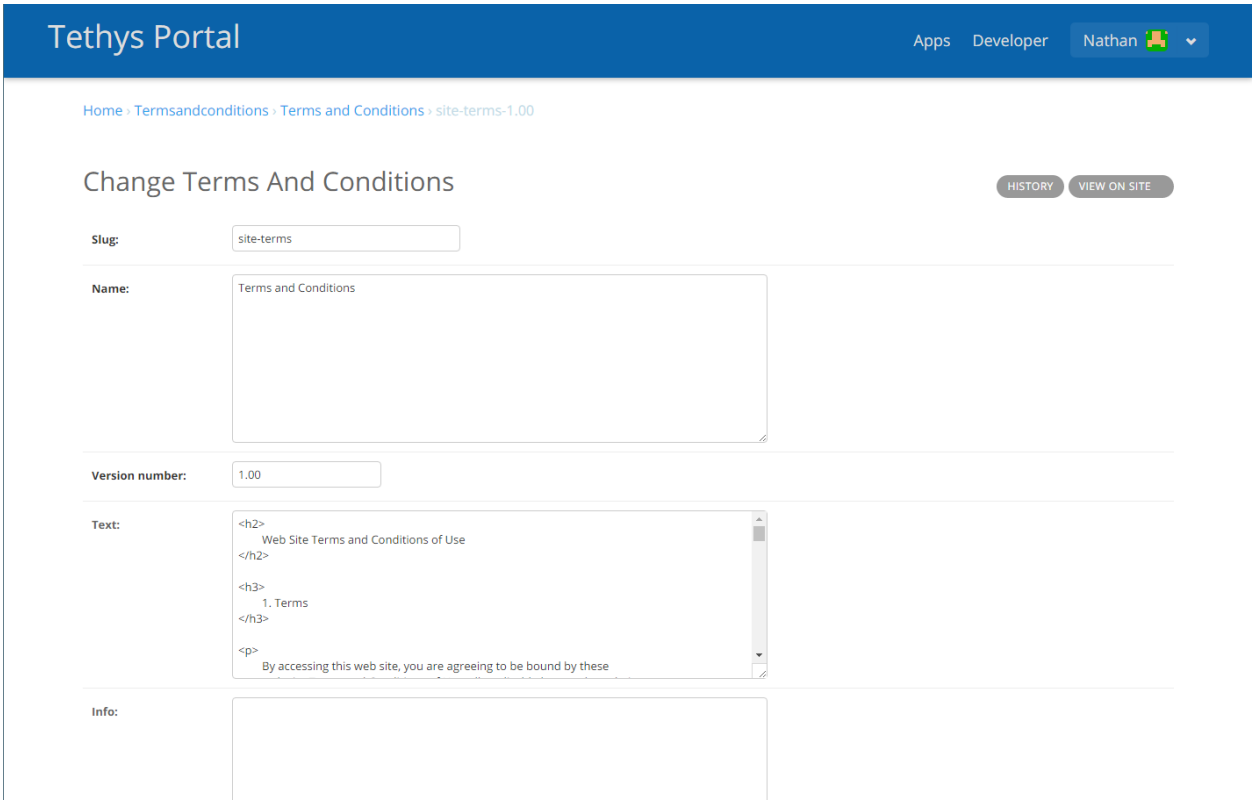


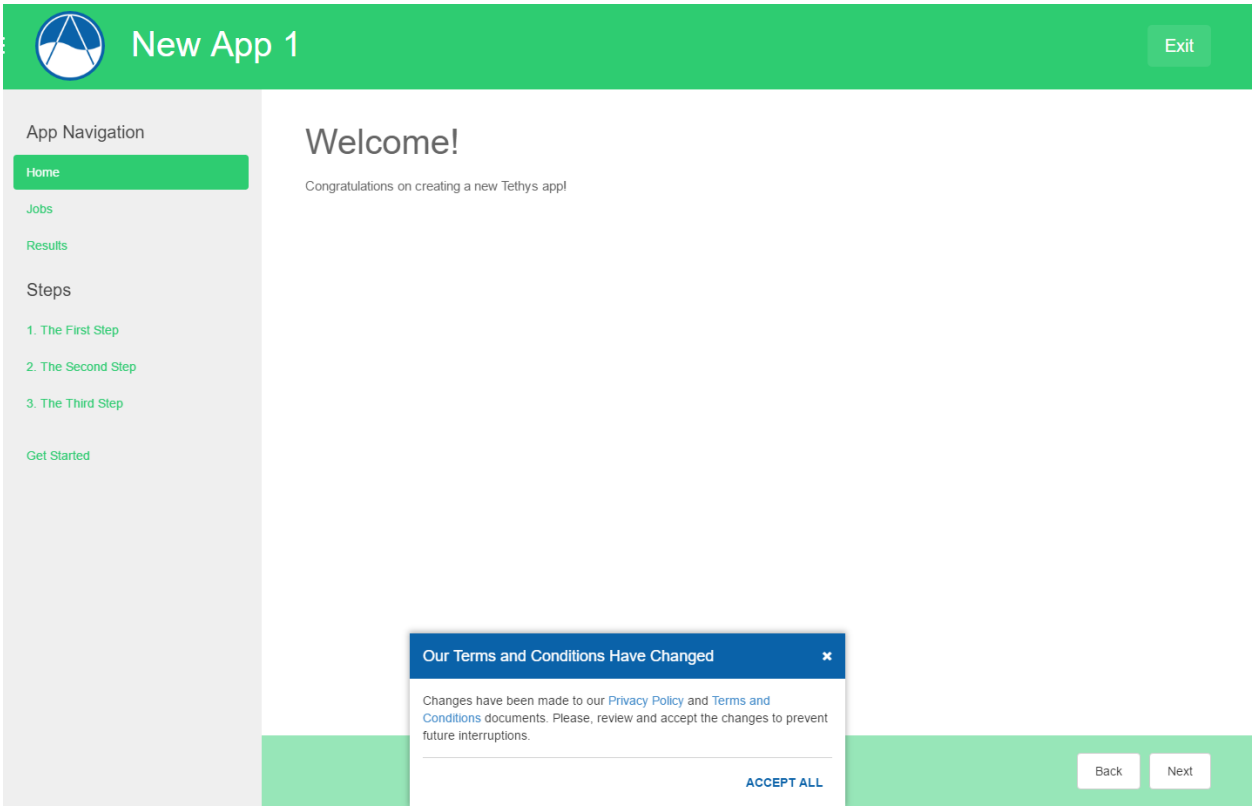
Figure 4. Creating a new legal document using the terms and conditions feature.

When a new document becomes active, users will be presented with a modal prompting them to review and accept the new terms and conditions (see Figure 5). The modal can be dismissed, but will reappear each time a page is refreshed until the user accepts the new versions of the legal documents. The `User Terms and Conditions` link shows a record of which users have accepted the terms and conditions.

Figure 5. Terms and conditions modal.

Tethys Apps

The links under the `TETHYS APPS` heading can be used to manage settings for installed apps and extensions. Clicking on the `Installed Apps` or `Installed Extensions` links will show a list of installed apps or extensions. Clicking on a link for an installed app or extension will bring you to the settings page for that app or extension. There are several different types of app settings: Common Settings, Custom Settings, and Service Settings.



Common Settings

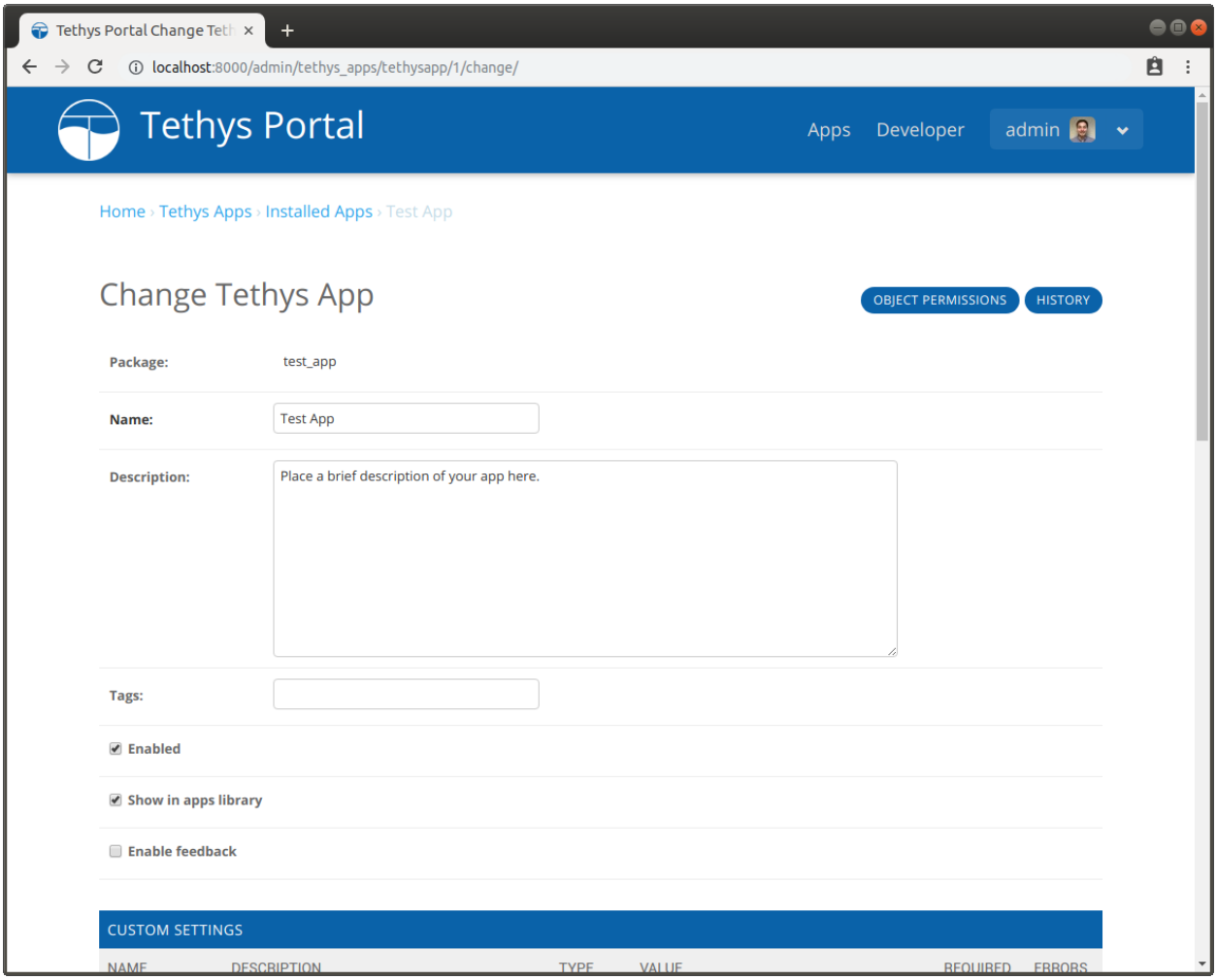
The Common Settings include those settings that are common to all apps or extension such as the `Name`, `Description`, `Tags`, `Enabled`, `Show in apps library`, and `Enable feedback` (see Figure 6). Many of these settings correspond with attributes of the term:*app class* and can be overridden by the portal administrator. Other control the visibility or accessibility of the app.

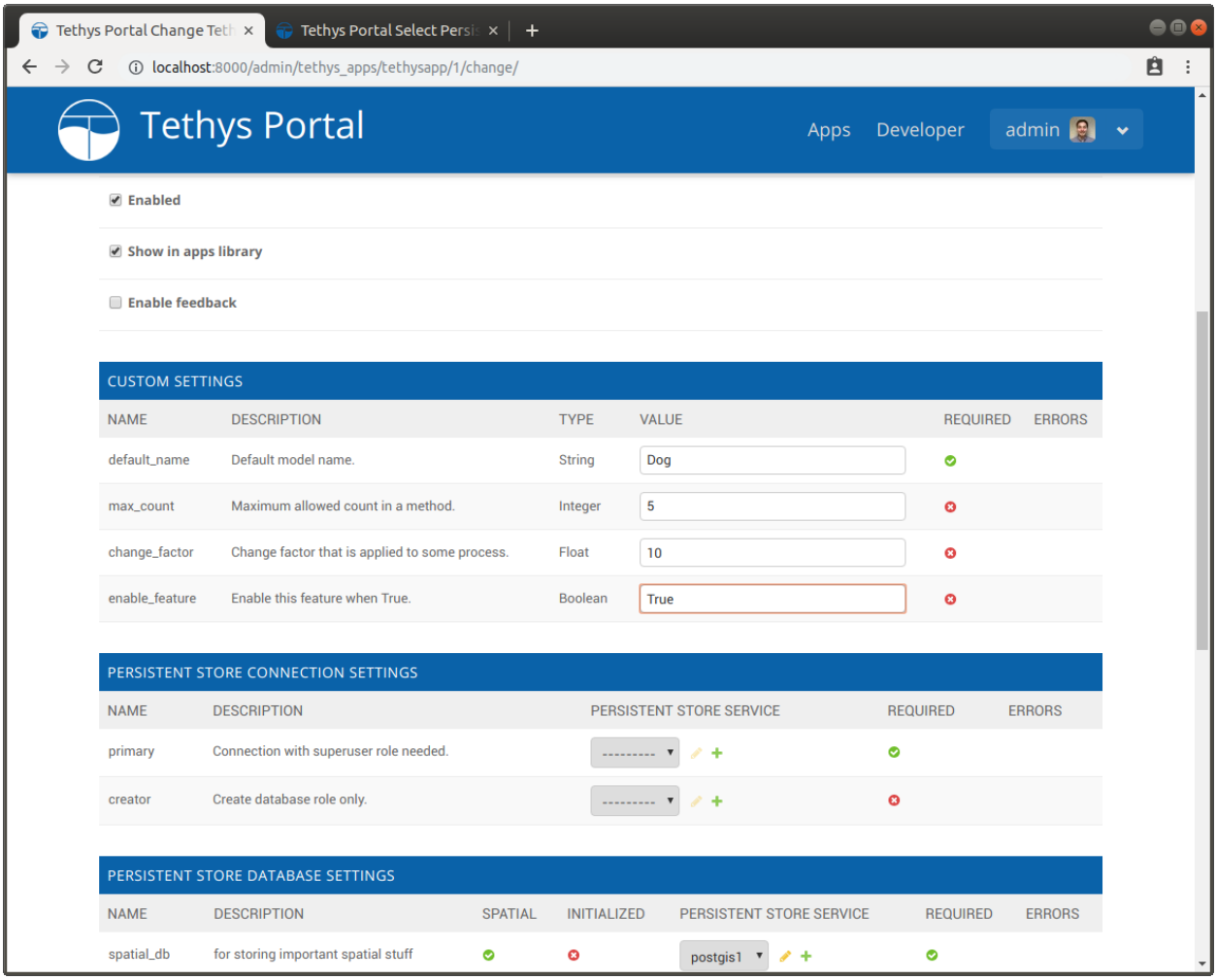
Figure 6. App settings page showing Common Settings.

Custom Settings

Custom Settings appear under the `CUSTOM SETTINGS` heading and are defined by the app developer (see Figure 7). Custom Settings have simple values such as strings, integers, floats, or booleans, but all are entered as text. For boolean type Custom Settings, type a valid boolean value such as `True` or `False`.

Figure 7. Custom Settings section of an app.


Tethys Portal Change Teth
localhost:8000/admin/tethys_apps/tethysapp/1/change/
Tethys Portal
Apps
Developer
admin
Home › Tethys Apps › Installed Apps › Test App
Change Tethys App
OBJECT PERMISSIONS
HISTORY
Package:
test_app
Name:
Test App
Description:
Place a brief description of your app here.
Tags:
Enabled
Show in apps library
Enable feedback
CUSTOM SETTINGS
NAME
DESCRIPTION
TYPE
VALUE
REQUIRED
ERRORS


Tethys Portal Change Tet
Tethys Portal Select Persi
localhost:8000/admin/tethys_apps/tethysapp/1/change/
Tethys Portal
Apps
Developer
admin
Enabled
Show in apps library
Enable feedback
CUSTOM SETTINGS
NAME
DESCRIPTION
TYPE
VALUE
REQUIRED
ERRORS
default_name
Default model name.
String
Dog
max_count
Maximum allowed count in a method.
Integer
5
change_factor
Change factor that is applied to some process.
Float
10
enable_feature
Enable this feature when True.
Boolean
True
PERSISTENT STORE CONNECTION SETTINGS
NAME
DESCRIPTION
PERSISTENT STORE SERVICE
REQUIRED
ERRORS
primary
Connection with superuser role needed.
----
creator
Create database role only.

PERSISTENT STORE DATABASE SETTINGS
NAME
DESCRIPTION
SPATIAL
INITIALIZED
PERSISTENT STORE SERVICE
REQUIRED
ERRORS
spatial_db
for storing important spatial stuff
postgis1

Service Settings

There are several different types of Service Settings including: `Persistent Store Connection Settings`, `Persistent Store Database Settings`, `Dataset Service Settings`, `Spatial Dataset Service Settings`, and `Web Processing Service Settings` (see Figure 8). These settings specify the types of services that the apps require. Use the drop down next to each Service Setting to assign a pre-registered `Tethys Service` to that app or use the *plus* button to create a new one.

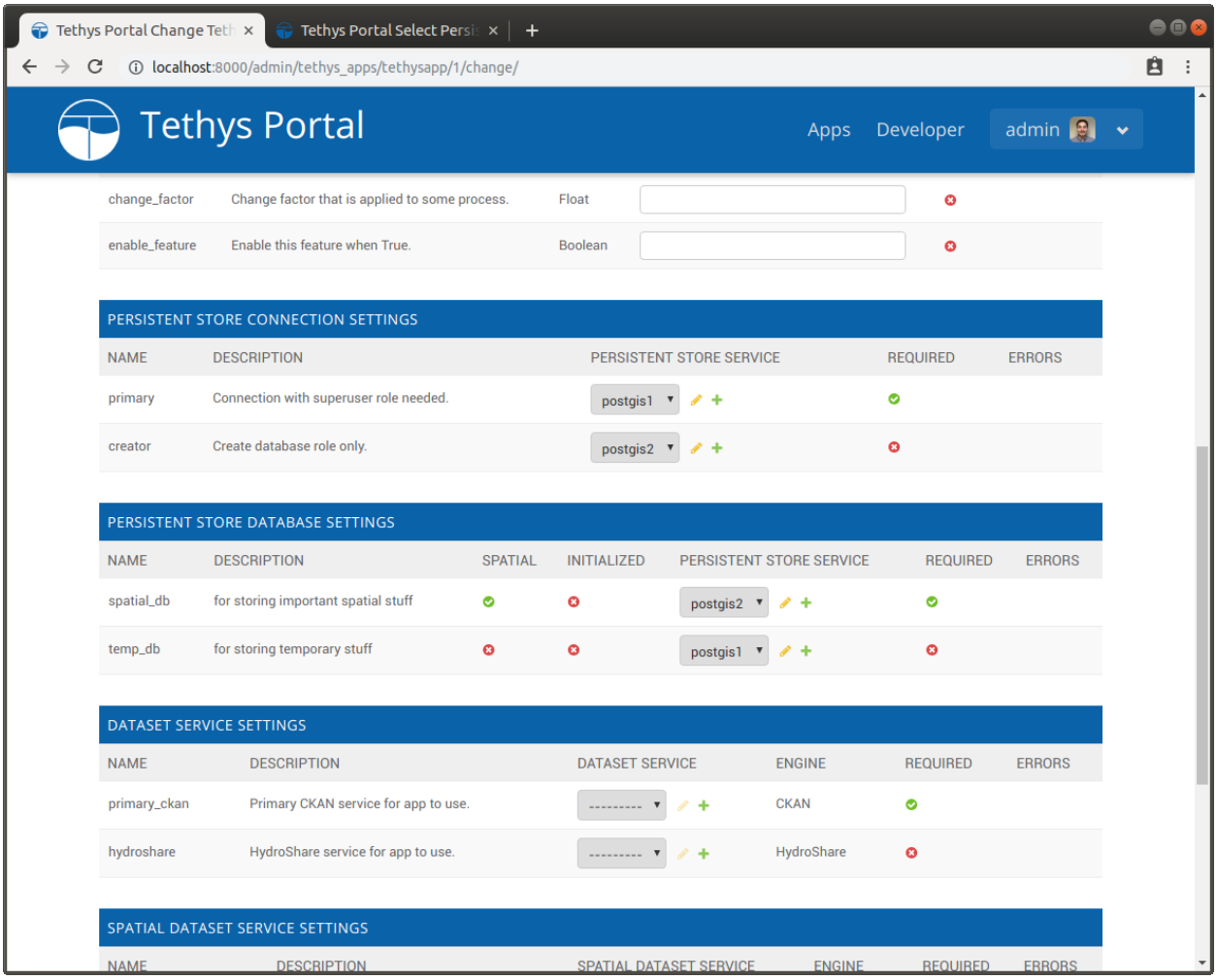


Figure 8. Service Settings sections of an app.

Tip: For information on how to define settings for your app see the *App Settings API* documentation. See *Tethys Services* for how to configure different `Tethys Services`.

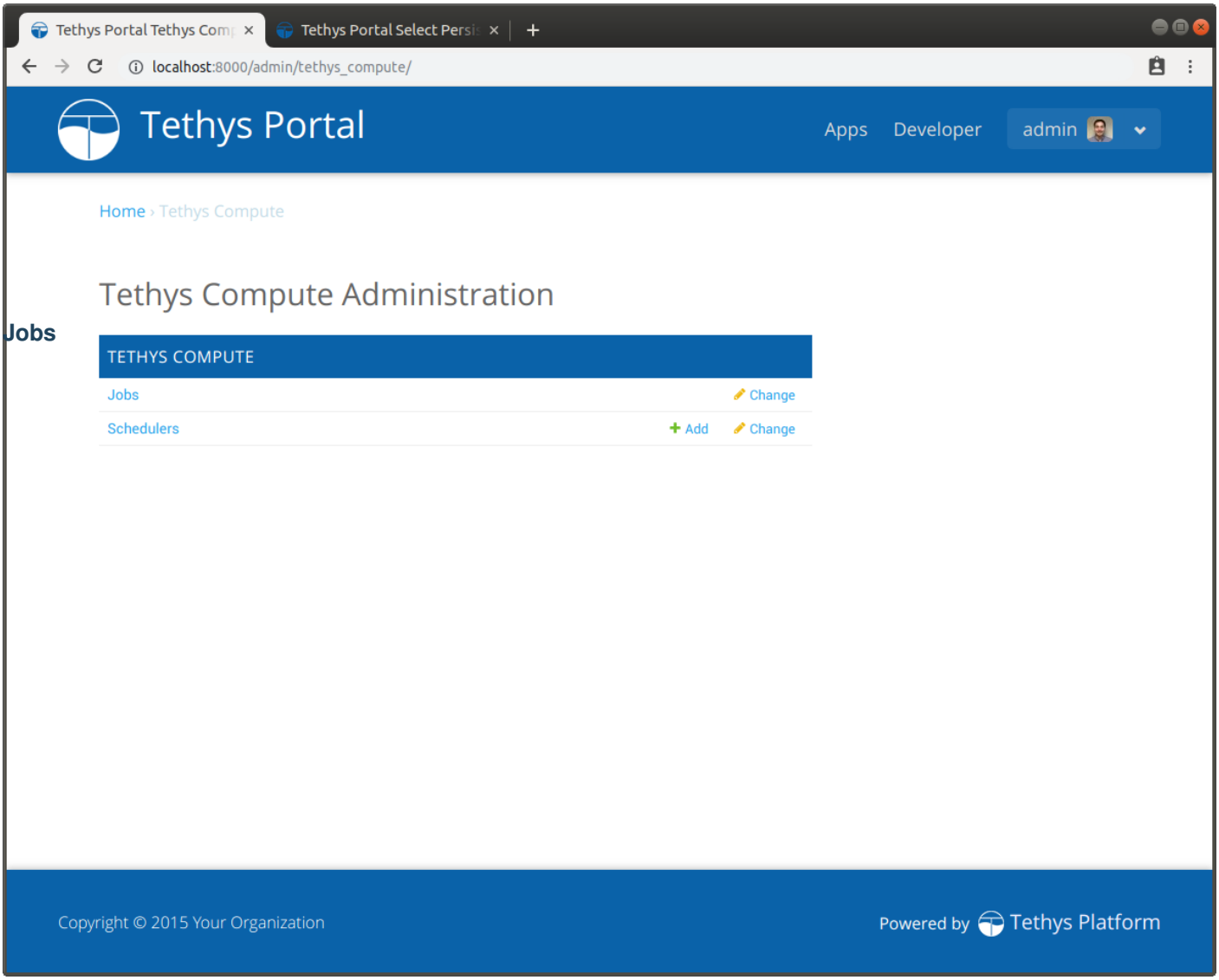
Tethys Compute

The links under the `TETHYS COMPUTE` heading can be used to manage `Jobs` and `Schedulers`:

Tethys Compute Admin Pages

The Tethys Compute settings in site admin allows an administrator to manage computing clusters, oversee jobs, configure schedulers, and configure settings for computing resources.

- *Jobs*
- *Schedulers*

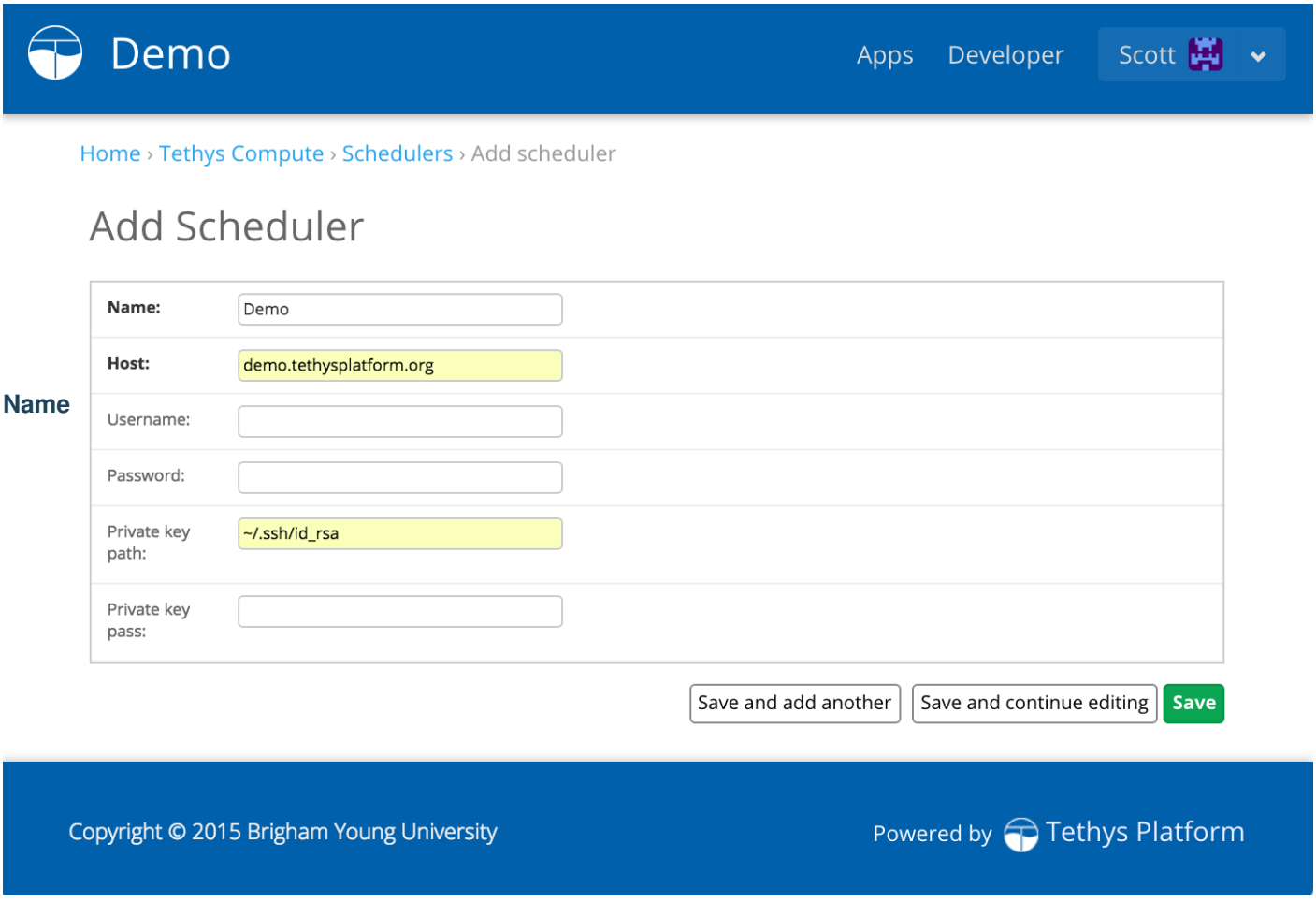


Jobs

ager. For each job that is created a database record is made to store some of the basic information about the job including: name, user, creation time, and status. The Jobs section in the Tethys Compute admin page allows for basic management of these database records. Jobs cannot be created in the admin pages, but they can be edited.

Schedulers

Schedulers are HTCondor nodes that have scheduling rights in the pool they belong to. Schedulers are needed for CondorJob types (see *Job Manager* documentation). When creating a new Scheduler there are two required settings: *Name* and *Host* ; an optional setting: *Username* ; and then two options for specifying authentication credentials: *Password* or *Private key path* and *Private key pass*.



Name

ulers.

Host

The fully qualified domain name (FQDN) or the IP address of the scheduler.

Username

The username that will be used to connect to the scheduler. The default username is 'root'.

Password

The password for the user specified by *Username* on the scheduler. Either a *Password* or a *Private key path* must be specified.

Private key path

The absolute path to the private key that is configured with the scheduler. Either a *Password* or a *Private key path* must be specified.

Note: The shortcut for the home directory: '~/' can be used and will be evaluated to the home directory of the Apache user.

Private key pass

The passphrase for the private key. If there is no passphrase then leave this field blank.

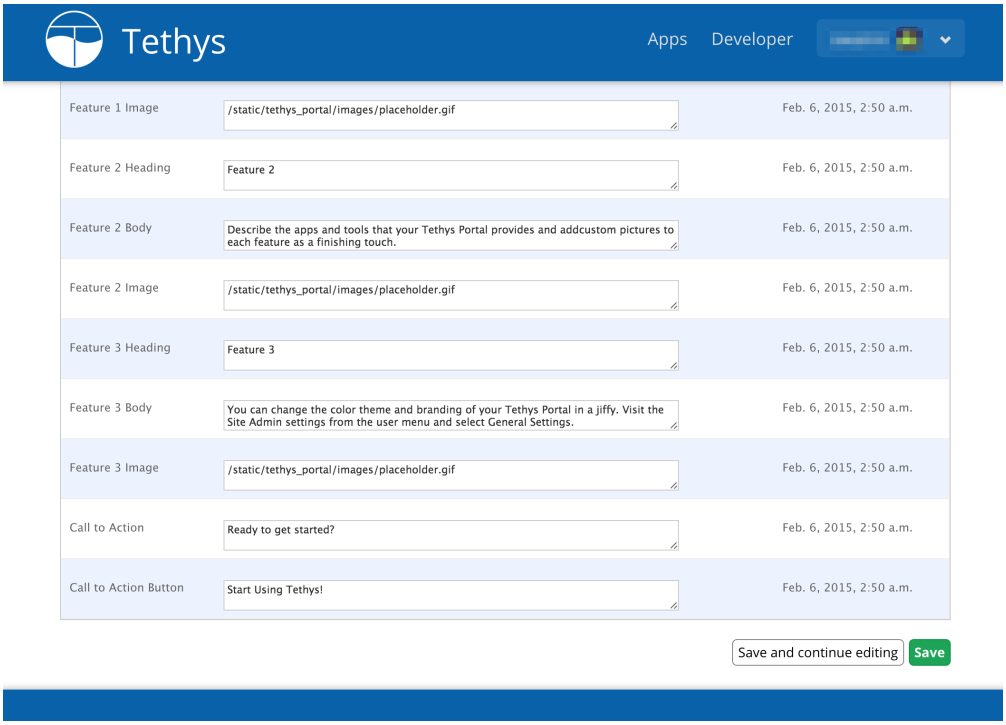
Tip: For more information on Tethys Jobs see the *Jobs API* and *Compute API* documentation.

Tethys Portal

The links under the `TETHYS PORTAL` heading can be used to customize the look of the Tethys Portal. For example, you can change the name, logo, and color theme of the portal (see Figure 9).

Figure 9. Home page settings for Tethys Portal.

Tip: For more information on customizing the Tethys Portal see the *Customize* documentation.



Tethys Services

The links under the `TETHYS SERVICES` heading can be used to register external services with Tethys Platform for use by apps and extensions. Use the `Spatial Dataset Services` link to register your Tethys Portal to GeoServer, the `Dataset Services` link to register to CKAN or HydroShare instances, the `Web Processing Services` link to register to WPS instances, or the `Persistent Store Services` link to register a database.

Tip: For detailed instructions on how to use each of these services in apps, refer to these docs:

- *Spatial Dataset Services API*
- *Dataset Services API*
- *Web Processing Services API*
- *Persistent Stores API*
- *Spatial Persistent Stores API*
- *Service Settings*

1.7.2 Customize

Last Updated: August 4, 2015

The content of Tethys Portal can be customized or rebranded to reflect your organization. To access these settings, login to Tethys Portal using an administrator account and select the `Site Settings` link under the `Tethys Portal` heading. Sitewide settings can be changed using the `General Settings` link and the content on the home page can be modified by using the `Home Page` link.

General Settings

The following settings can be used to modify global features of the site. Access the settings using the `Site Settings > General Settings` links on the admin pages.

Setting	Description
Site Title	Title of the web page that appears in browser tabs and bookmarks of the site.
Favicon	Path to the image that is used in browser tabs and bookmarks.
Brand Text	Title that appears in the header.
Brand Image	Logo or image that appears next to the title in the header.
Brand Image Height	Height to scale the brand image to.
Brand Image Width	Width to scale the brand image to.
Brand Image Padding	Adjust space above brand image to center it.
Apps Library Title	Title of the page that displays app icons.
Primary Color	Color that is used as the primary theme color (e.g.: #ff0000 or rgb(255,0,0)).
Secondary Color	Color that is used as the secondary theme color.
Primary Text Color	Color of the text appearing in the headers and footer.
Primary Text Hover Color	Hover color of the text appearing in the headers and footer (where applicable).
Secondary Text Color	Color of secondary text on the home page.
Secondary Text Hover Color	Hover color of the secondary text on the home page.
Background Color	Color of the background on the apps library page and other pages.
Footer Copyright	Copyright text that appears in the footer.

Figure 2. General settings for Tethys Portal.

Home Page Settings

The following settings can be used to modify the content on the home page. Access the settings using the `Site Settings > Home Page` links on the admin pages.

Setting	Description
Hero Text	Text that appears in the hero banner at the top of the home page.
Blurb Text	Text that appears in the blurb banner, which follows the hero banner.
Feature 1 Heading	Heading for 1st feature highlight.
Feature 1 Body	Body text for the 1st feature highlight.
Feature 1 Image	Path or url to image for the 1st feature highlight.
Feature 2 Heading	Heading for 2nd feature highlight.
Feature 2 Body	Body text for the 2nd feature highlight.
Feature 2 Image	Path or url to image for the 2nd feature highlight.
Feature 3 Heading	Heading for 3rd feature highlight.
Feature 3 Body	Body text for the 3rd feature highlight.
Feature 3 Image	Path or url to image for the 3rd feature highlight.
Call to Action	Text that appears in the call to action banner at the bottom of the page (only visible when user is not logged in).
Call to Action Button	Text that appears on the call to action button in the call to action banner (only visible when user is not logged in).

Bypass the Home Page

Tethys Portal can also be configured to bypass the home page. When this setting is applied, the root url will always redirect to the apps library page. This setting is modified in the `settings.py` script. Simply set the `BYPASS_TETHYS_HOME_PAGE` setting to `True` like so:

```
BYPASS_TETHYS_HOME_PAGE = True
```

Enable Open Signup

Prior to version 1.3.0, any visitor to a Tethys portal could signup for an account without administrator approval or in other words account signup was open. For version 1.3.0+ the open signup capability has been disabled by default for security reasons. To enable open signup, you must modify the `ENABLE_OPEN_SIGNUP` setting in the `settings.py` script:

```
ENABLE_OPEN_SIGNUP = True
```

1.7.3 Social Authentication

Last Updated: September 2019

Tethys Portal supports authenticating users with Google, Facebook, LinkedIn and HydroShare via the OAuth 2.0 method. The social authentication and authorization features have been implemented using the Python Social Auth module and the social buttons provided by the Social Buttons for Bootstrap. Social login is disabled by default, because enabling it requires registering your tethys portal instance with each provider.

Enable Social Login

Use the following instructions to setup social login for the providers you desire.

Caution: These instructions assume that you have generated a new settings file after upgrading to **Tethys Platform 1.2.0** or later. If this is not the case, please review the *Social Auth Settings* section.

Google

1. Create a Google Developer Account

   Follow these instructions to register your project and create a client ID: Setting Up OAuth 2.0. Provide the following as you setup OAuth2:

   a. Provide Authorized JavaScript Origins

      As a security precaution, Google will only accept authentication requests from the hosts listed in the `Authorized JavaScript Origins` box. Add the domain of your Tethys Portal to the list. Optionally, you may add a localhost domain to the list to be used during testing. For example, if the domain of your Tethys Portal is `www.example.org`, you would add the following entries:

      ```
      https://www.example.org
      http://localhost:8000
      ```

   b. Provide Authorized Redirect URIs

      You also need to provide the callback URI for Google to call once it has authenticated the user. This follows the pattern `http://<host>/oauth2/complete/google-oauth2/`. For a Tethys Portal at domain `www.example.org`:

      ```
      https://www.example.org/oauth2/complete/google-oauth2/
      https://localhost:8000/oauth2/complete/google-oauth2/
      ```

   Note: Some Google APIs are free to use up to a certain quota of hits. Be sure to familiarize yourself with the terms of use for each service.

2. Open `settings.py` script located in `$TETHYS_SRC/tethys_portal/settings.py`

   Add the `social_core.backends.google.GoogleOAuth2` backend to the `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.google.GoogleOAuth2',
    'django.contrib.auth.backends.ModelBackend',
)
```

Copy the `Client ID` and `Client secret` into the `SOCIAL_AUTH_GOOGLE_OAUTH2_KEY` and `SOCIAL_AUTH_GOOGLE_AUTH2_SECRET` settings, respectively:

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = '...'
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = '...'
```

References

For more detailed information about using Google social authentication see the following articles:

- Developer Console Help
- Google Identity Platform

Facebook

1. Create a Facebook Developer Account

   You will need a Facebook developer account to register your Tethys Portal with Facebook. To create an account, visit https://developers.facebook.com and sign in with a Facebook account.

2. Create a Facebook App

a. Point to `My Apps` and select `Create App`.

b. Fill out the form and press `Create App ID` button.

3. Setup OAuth

   a. Scroll down and locate the tile titled Facebook Login.

   b. Press the `Setup` button on the tile (or `Settings` if setup previously).

   c. If your Tethys Portal were hosted at `www.example.com`, you would enter the following for the Valid OAuth Redirect URIs field:

   ```
   https://www.example.org/oauth2/complete/facebook/
   ```

Note: Localhost domains are automatically enabled when the app is in development mode, so you don't need to add them for Facebook OAuth logins.

---

   d. Press the `Save Changes` button.

   c. Make the app public you wish by changing the toggle switch in the header from `Off` to `On`.

Note: The Facebook app must be public to allow Facebook authentication to non-localhost Tethys Portals.

---

4. Expand the `Settings` menu on the left and select `Basic`. Note the `App ID` and `App Secret`.

5. Open `settings.py` script located in `$TETHYS_SRC/tethys_portal/settings.py`

   Add the `social_core.backends.facebook.FacebookOAuth2` backend to the `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.facebook.FacebookOAuth2',
    'django.contrib.auth.backends.ModelBackend',
)
```

   Copy the `App ID` and `App Secret` to the `SOCIAL_AUTH_FACEBOOK_KEY` and `SOCIAL_AUTH_FACEBOOK_SECRET` settings, respectively:

```
SOCIAL_AUTH_FACEBOOK_KEY = '...'
SOCIAL_AUTH_FACEBOOK_SECRET = '...'
```

References

For more detailed information about using Facebook social authentication see the following articles:

- Facebook Login
- Facebook Login for the Web with the JavaScript SDK

LinkedIn

1. Create a LinkedIn Developer Account

   You will need a LinkedIn developer account to register your Tethys Portal with LinkedIn. To create an account, visit https://developer.linkedin.com/my-apps and sign in with a LinkedIn account.

2. Create a LinkedIn Application

a. Navigate back to https://www.linkedin.com/developers/apps, if necessary and press the `Create App` button.

b. Fill out the form and press `Create App`.

3. Open the **Auth** tab and note the `Client ID` and `Client Secret` for Step 5.

4. Setup OAuth

   a. Add the call back URLs under the **OAuth 2.0 settings** section. For example, if your Tethys Portal is hosted at the domain `www.example.org`:

```
https://www.example.org/oauth2/complete/linkedin-oauth2/
http://localhost:8000/oauth2/complete/linkedin-oauth2/
```

5. Open `settings.py` script located in `$TETHYS_SRC/tethys_portal/settings.py`

   Add the `social_core.backends.linkedin.LinkedinOAuth2` backend to the `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    ...
    'social_core.backends.linkedin.LinkedinOAuth2',
    'django.contrib.auth.backends.ModelBackend',
)
```

Copy the `Client ID` and `Client Secret` to the `SOCIAL_AUTH_LINKEDIN_OAUTH2_KEY` and `SOCIAL_AUTH_LINKEDIN_OAUTH2_SECRET` settings, respectively:

```
SOCIAL_AUTH_LINKEDIN_OAUTH2_KEY = '...'
SOCIAL_AUTH_LINKEDIN_OAUTH2_SECRET = '...'
```

References

For more detailed information about using LinkedIn social authentication see the following articles:

- LinkedIn: Authenticating with OAuth 2.0

HydroShare

1. Create a HydroShare Account

   You will need a HydroShare account to register your Tethys Portal with HydroShare. To create an account, visit https://www.hydroshare.org.

2. Create and setup a HydroShare Application

   a. Navigate to https://www.hydroshare.org/o/applications/register/.

   b. Name: Give this OAuth app a name. It is recommended to use the domain of your Tethys Portal instance as the name, like: www.my-tethys-portal.com

   c. Client id: Leave unchanged. Note this value for step 3.

   d. Client secret: Leave unchanged. Note this value for step 3.

   e. Client type: Select "Confidential".

   f. Authorization grant type: Select "Authorzation code".

   g. Redirect uris: Add the call back URLs. The protocol (http or https) that matches your Tethys Portal settings should be included in this url. For example:

```
if your Tethys Portal was located at the domain ``https://www.my-tethys-
↪portal.com``:
    https://www.my-tethys-portal.com/oauth2/complete/hydroshare/

if your Tethys Portal was on a local development machine:
    http://localhost:8000/oauth2/complete/hydroshare/
    or
    http://127.0.0.1:8000/oauth2/complete/hydroshare/
```

   h. Press the "Save" button.

3. Open `settings.py` script located in `$TETHYS_SRC/tethys_portal/settings.py`

   Add the `tethys_services.backends.hydroshare.HydroShareOAuth2` backend to the `AUTHENTICATION_BACKENDS` setting:

```
AUTHENTICATION_BACKENDS = (
    'tethys_services.backends.hydroshare.HydroShareOAuth2',
    ...
    'django.contrib.auth.backends.ModelBackend',
)
```

Assign the `Client id` and `Client secret` to the `SOCIAL_AUTH_HYDROSHARE_KEY` and `SOCIAL_AUTH_HYDROSHARE_SECRET` settings, respectively:

```
SOCIAL_AUTH_HYDROSHARE_KEY = '...'
SOCIAL_AUTH_HYDROSHARE_SECRET = '...'
```

4. Work with HydroShare in your app

   Once user has logged in Tethys through HydroShare OAuth, your app is ready to retrieve data from HydroShare on behalf of this HydroShare user using HydroShare REST API Client (hs_restclient). A helper function is provided to make this integration smoother.

```
# import helper function
from tethys_services.backends.hs_restclient_helper import get_oauth_hs

# your controller function
def home(request)

    # put codes in a 'try..except...' statement
    try:
        # pass in request object
        hs = get_oauth_hs(request)

        # your logic goes here. For example: list all HydroShare resources
        for resource in hs.getResourceList():
            print(resource)

    except Exception as e:
        # handle exceptions
        pass
```

5. (Optional) Link to a testing HydroShare instance

   The production HydroShare is located at https://www.hydroshare.org/. In some cases you may want to link your Tethys Portal to a testing HydroShare instance, like hydroshare-beta. Tethys already provides OAuth backends for hydroshare-beta and hydroshare-playground. To activate them, you need to go through steps 1-3 for each backend (replace www.hydroshare.org with the testing domain urls accordingly).

   At step 3:

   a. Append the following classes in `AUTHENTICATION_BACKENDS` settings:

      hydroshare-beta: `tethys_services.backends.hydroshare_beta.HydroShareBetaOAuth2`

      hydroshare-playground: `tethys_services.backends.hydroshare_playground.HydroSharePlaygroundOAuth2`

   b. Assign the `Client ID` and `Client Secret` to the following variables:

      hydroshare-beta: `SOCIAL_AUTH_HYDROSHARE_BETA_KEY`

      `SOCIAL_AUTH_HYDROSHARE_BETA_SECRET`

      hydroshare-playground: `SOCIAL_AUTH_HYDROSHARE_PLAYGROUND_KEY`

      `SOCIAL_AUTH_HYDROSHARE_PLAYGROUND_SECRET`

Note: To prevent any unexpected behavior in section (4), a Tethys account SHOULD NOT be

associated with multiple HydroShare social accounts.

References

For more detailed information about using HydroShare social authentication see the following articles:

- https://github.com/hydroshare/hydroshare/wiki/HydroShare-REST-API#oauth-20-support

Social Auth Settings

The following code snippet shows the settings in the `settings.py` that are relevant to social auth in Tethys Platform:

```
INSTALLED_APPS = (
    ...
    'social_django',
)

MIDDLEWARE_CLASSES = (
    ...
    'tethys_portal.middleware.TethysSocialAuthExceptionMiddleware',
)

AUTHENTICATION_BACKENDS = (
    'tethys_services.backends.hydroshare.HydroShareOAuth2',
    'social_core.backends.linkedin.LinkedinOAuth2',
    'social_core.backends.google.GoogleOAuth2',
    'social_core.backends.facebook.FacebookOAuth2',
    'django.contrib.auth.backends.ModelBackend',
    'guardian.backends.ObjectPermissionBackend',
)

TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'django.contrib.messages.context_processors.messages',
                'social_django.context_processors.backends',
                'social_django.context_processors.login_redirect',
                ...
            ],
            ...
        }
    }
]

# OAuth Settings
SOCIAL_AUTH_ADMIN_USER_SEARCH_FIELDS = ['username', 'first_name', 'email']
SOCIAL_AUTH_SLUGIFY_USERNAMES = True
SOCIAL_AUTH_LOGIN_REDIRECT_URL = '/apps/'
SOCIAL_AUTH_LOGIN_ERROR_URL = '/accounts/login/'

# OAuth Providers
## Google
```

(continues on next page)

(continued from previous page)

```
SOCIAL_AUTH_GOOGLE_OAUTH2_KEY = ''
SOCIAL_AUTH_GOOGLE_OAUTH2_SECRET = ''

## Facebook
SOCIAL_AUTH_FACEBOOK_KEY = ''
SOCIAL_AUTH_FACEBOOK_SECRET = ''
SOCIAL_AUTH_FACEBOOK_SCOPE = ['email']

## LinkedIn
SOCIAL_AUTH_LINKEDIN_OAUTH2_KEY = ''
SOCIAL_AUTH_LINKEDIN_OAUTH2_SECRET = ''

## HydroShare
SOCIAL_AUTH_HYDROSHARE_KEY = ''
SOCIAL_AUTH_HYDROSHARE_SECRET = ''
```

1.7.4 Developer Tools

Last Updated: August 4, 2015

Tethys provides a Developer Tools page that is accessible when you run Tethys in developer mode. Developer Tools contain documentation, code examples, and live demos of the features of various features of Tethys. Use it to learn how to add a map or a plot to your web app using Gizmos or browse the available geoprocessing capabilities and formulate geoprocessing requests interactively.

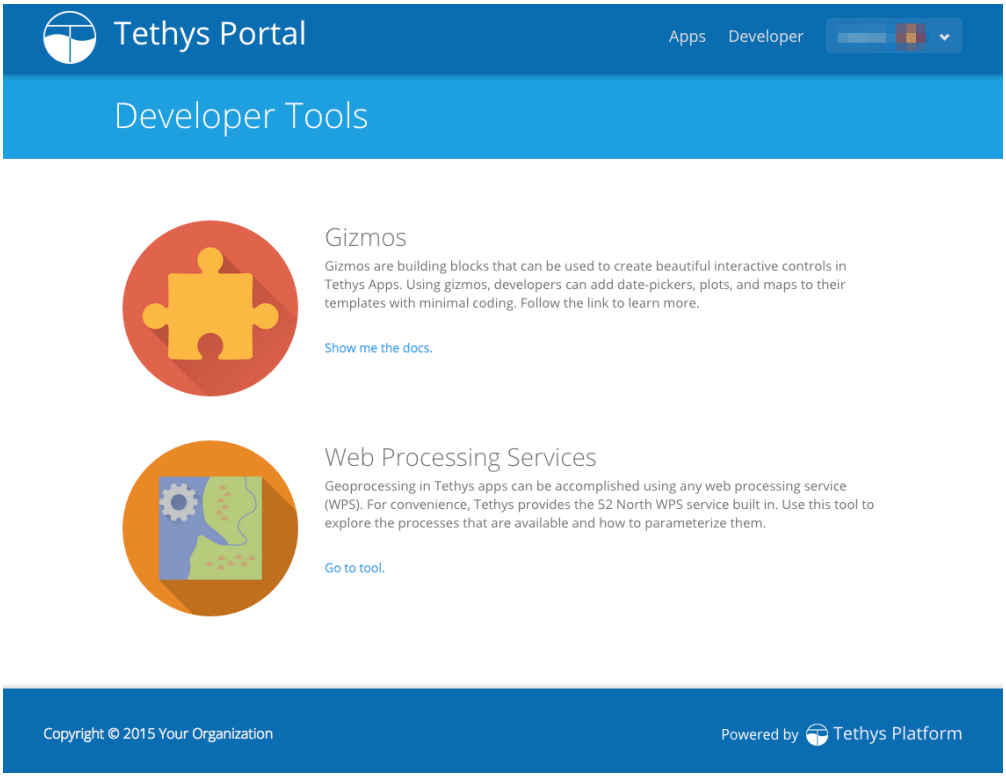


Figure 4. Use the Developer Tools page to assist you in development.

1.7.5 App Feedback

Last Updated: December 15, 2015

Tethys Portal includes a feature for enabling app feedback from the app-users. When activated, the feature shows a button on the bottom-left of each app page that activates a feedback form. The form is submitted to specified developers. The feature is supported starting in Tethys 1.3.0.

Enable Feedback

Use the following instructions to setup the feedback form on a Tethys app.

Add feedback properties to the *app configuration file* (`app.py`)

Open the *app configuration file* (`app.py`) found in the app installation directory using a text editor and add the following properties to the `TethysAppBase` class. The `feedback_emails` should correspond to specific app developers that desire feedback.

```
enable_feedback = True
feedback_emails = ['app_developer@emaildomain.com', 'another_app__
↪developer@emaildomain.com']
```

Note: The emails will only be sent if Step *3. Setup Email Capabilities (optional)* has been setup upon installing Tethys.

If either of the properties listed above are not defined or if `enable_feedback` is set to False, the feedback feature will not be available.

Example

```
class MyFirstApp(TethysAppBase):
  """
  Tethys app class for My First App.
  """

  name = 'My First App'
  index = 'my_first_app:home'
  icon = 'my_first_app/images/icon.gif'
  package = 'my_first_app'
  root_url = 'my-first-app'
  color = '#29ABE1'
  enable_feedback = True
  feedback_emails = ['developer@myfirstapp.com']
```

1.8 Source Code

Last Updated: August 11, 2015

The source code for Tethys Platform is contained in the following repositories:

- Tethys Platform
- Tethys Dockers
- Tethys Dataset Services
- TethysCluster
- CondorPy

1.9 Contribute

Last Updated: July 13, 2016

Tethys Platform is a growing Open Source project and we are always looking for developers who wish to contribute and help improve the platform. If you would like to contribute, join the discussion on the Tethys Forum and visit the Tethys Development Wiki on the Tethys Platform GitHub repository.

Resources:

- Tethys Forum
- Tethys Development Wiki

1.10 Supplementary

Last Updated: May 27, 2015

This section provides a list of miscellaneous reference material that can be used to help you understand Tethys Platform and Tethys app development in more detail.

1.10.1 Key Concepts

Last Updated: April 6, 2015

The purpose of this page is to provide an explanation of some of the key concepts of Tethys Platform. The concepts are only discussed briefly here to provide a basic overview. It is highly recommended that you visit the suggested resources to have a better understanding of these concepts, as developing apps in Tethys Platform relies heavily on them.

What is an App?

In the most basic sense, an app is a workflow. The purpose of an app is not provide an all-in-one solution, but rather to perform a narrowly focused task or set of tasks. For example, an app that works with hydrologic models might be focused on guiding the user to change the land use layer of a model, run the modified model, and compare the result with the original model results.

In terms of implementation, an app built with Tethys Platform or a Tethys app is a web app(as opposed to a mobile app). A Tethys Platform installation provides a website called the Tethys Portal that can be used to organize and access your apps. Tethys apps are technically extensions of the Tethys Portal web page, because when you create a Tethys app you will be adding additional web pages to the Tethys Portal web site. Tethys Platform is built on the Django Python web framework, so Tethys apps are also Django web apps--though Tethys Platform streamlines many aspects of Django web development. This is why the Django documentation is referred to often in the documentation for Tethys Platform.

Web Frameworks

Tethys Portal is built using the Django web framework. Understanding the difference between a static website and a dynamic website built with a web framework is important for app developers, because apps rely on web framework concepts.

Static web development consists of creating a series of HTML files--one for every page of the website. The files are organized using the server's file system and stored in some directory on the server that is accessible by the Internet. For a static site, the URL works very similar to how a file path works on an operating system. When a request is sent from the web browser to a server, the server locates the HTML file that the URL is requesting and returns it to the browser for the user to view.

The static method of developing web pages presents some problems for developers. For example, if a developer wants to include a consistent header and footer on every page of her website, she would end up duplicating the header and footer code many times (via copy and paste). As a result, static websites are more difficult to update and maintain, because changes need to be made wherever the code is duplicated. Developing a website in this way is error prone and can become prohibitive for large websites.

Web frameworks provide a way to develop websites using a programmatic approach. Instead of static HTML files, developers create generic reusable HTML templates. With a web framework, the developer can create one template file containing only the markup for the header and another template file for the footer. Now when the developer wants to include the header and footer in another page, she uses an import construct that references the header and footer templates. The header and footer markup is added dynamically to all the files that need it upon request by a template rendering program. Maintenance is much easier, because changes to the header and footer only need to be made in one place and the entire site will be updated. In this way, the site becomes dynamic. One type of software that makes it possible to create dynamic web pages is a web framework.

Web frameworks also handle requests differently than traditional web pages. When the user submits a request to the server, instead of looking up a file on the server at the directory implied by the URL, the request is handed to the web framework application. The web framework application processes the request and usually returns a web page that has been generated dynamically as the result. This type of web framework application is called a Common Gateway Interface (CGI) application; or if the application is a Python web framework, it is called a Web Server Gateway Interface (WSGI) application.

Model View Controller

The dynamic templating feature is only one aspect of what web frameworks offer. Many web frameworks use a software development pattern called Model View Controller (MVC). MVC is used to organize the code that is used to develop user interfaces into conceptual components. A brief explanation of each components is provided:

Model

The model represents the code that is used to store and retrieve data that is used in the web application. Most websites use SQL databases for persistent data storage, so the model is usually made up of a database model.

View

Views are used to represent the data to the end user. In a web applications views are the HTML pages that are generated. Views are typically generic, reusable, and oblivious to the origins of the data that fills them. This is possible because of templating languages that allow coders to create dynamic HTML web pages.

Controller

Controllers are used to orchestrate the interaction between the view and the model. They contain the logic for retrieving data from the database and transforming it into a format that is consumable by the view, because the model and the view never communicate directly. Controllers also handle the input from the user.

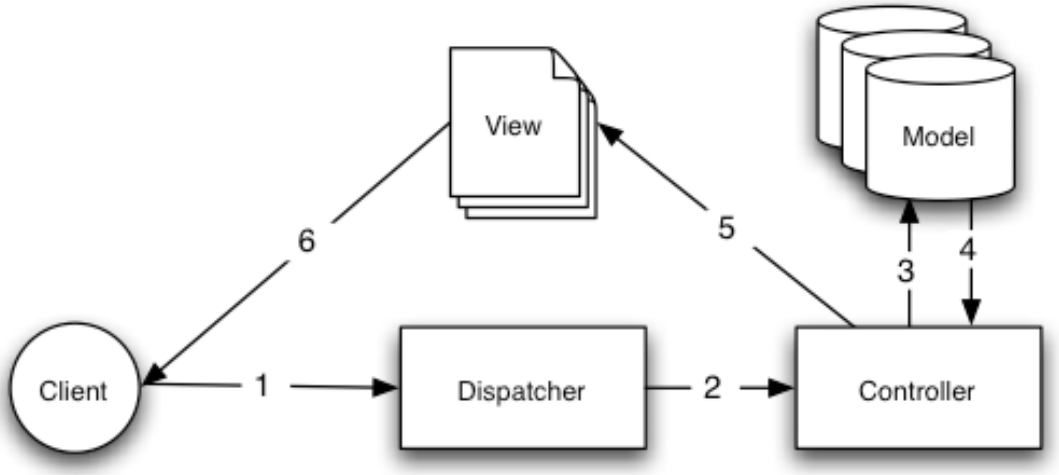


Fig. 3: A typical collaboration of MVC components (courtesy of Cake PHP Docs)

When a user submits a request, the web application (dispatcher in the Figure above) looks up the controller that is mapped to the URL and executes it. The controller may perform change or lookup data from the model after which it returns this data and a template to render. This is handed off to a template rendering utility that processes the template and generates HTML. The HTML is rendered for the user's viewing pleasure in the web browser or other client. The user sends another request, and the process repeats.

URL Design and REST Paradigm

The URL takes on a different meaning in dynamic websites than it does in a static website. In a static website, the URL maps to directories and files on the server. In a dynamic website, there are no static files (or at least very few) to map to. The web framework simply maps the URL to a controller and returns the result. Although the developer is free to use URL's in whatever manner they would like, it is recommended that some type of URL pattern should be used to make the website more maintainable.

We recommend developers use some form of the Representational State Transfer (REST) abstraction for creating meaningful URLs for apps. In a REST architecture for a website, the data of the website is referred to as resources. The current state of resources is presented to the user through some representation, for example, an HTML document. The user can interact with the resources through the actions of the controller. Examples of common actions on resources are create, read (view), update (edit), and delete, often referred to as CRUD. In true REST implementations, the CRUD operations are mapped to specific HTTP methods: POST, GET, PUT, and DELETE, respectively (see *HTTP Verbs*). In practice HTML only supports the POST and GET HTTP methods, so a pseudo-REST implementation is achieved via URL patterns.

For example, consider an app that is meant to provide information about a stream gages. In this case, the resources of the website may be stream gage records in a database. A potential URL for a page that shows a summary about a single stream gage record would be:

```
www.example.com/gages/1/show
```

The number "1" in the URL represents the stream gage record ID in the database. To show a page with the representation of another stream gage, the ID number could be changed. A generalization of this URL pattern could be represented as:

```
/gages/{id}/{action}
```

In this URL pattern, variables are represented using curly braces. The `{id}` variable in the URL represents the ID of a stream gage resource in our database and the `{action}` variable represents the action to perform on the stream gage resource. The `{action}` variable is used instead of HTTP methods to indicate which CRUD operation to perform on the resource. In the first example, the action "show" is used to perform the read operation. Often, the show action is the default action, so the URL could be shortened to:

```
www.example.com/gages/1
```

Similarly, a URL for a page the represents all of the stream gages in the database in a list could be represented by omitting the ID:

```
www.example.com/gages
```

URLs for each of the CRUD operations on the steram gage database could look like this:

```
# Create
www.example.com/gages/new

# Read One
www.example.com/gages/1

# Read All
www.example.com/gages

# Update
www.example.com/gages/1/edit
```

(continues on next page)

(continued from previous page)

```
# Delete
www.example.com/gages/1/delete
```

Before you dive into writing your app, you should take some time to design the URLs for the app. Define the resources for your app and the URLs that will be used to perform the CRUD operations on the resources.

Caution: The examples above used integer IDs for simplicity. However, using integer IDs in URLs is not recommended, because they are often incremented consecutively and can be easily guessed. For example, it would be very easy for an attacker to write a script that would increment through integer IDs and call the delete method on all your resources. A better option would be to use randomly assigned IDs such as a UUID.

HTTP Verbs

Anytime you type a URL into an address bar you are performing what is called a GET request. All of the above URLs are examples of implementing REST using only GET requests. GET is an example of an HTTP verb or method. There are quite a few HTTP verbs, but the other verbs pertinent to REST are POST, PUT, and DELETE. A truely RESTful design would make use of these HTTP verbs to implement the CRUD for the resources instead of using different key word actions. Consider our example from above. To read or view a dog resource, we use a GET request as before:

```
HTTP GET
www.example.com/dogs/1
```

However, to implement the create action for a dog resource, now we use the POST verb with the same url that we used for the read action:

```
HTTP POST
www.example.com/dogs/1
```

Similarly, to delete the dog resource we use the same URL as before but this time use the DELETE verb and to update or edit a dog resource, we use the PUT verb. Using this pattern, the URL becomes a unique resource identifier (URI) and the HTTP verbs dictate what action we will perform on the data. Unfortunately, HTML (which is the interface of HTTP) does not implement PUT or DELETE verbs in forms. In practice many RESTful sites use the "action" pattern for interacting with resources, because not all of the HTTP verbs are supported.

1.10.2 App Project Structure

Last Updated: November 17, 2014

The source code for a Tethys app project is organized in a specific file structure. Figure 1 illustrates the key components of a Tethys app project called "my_first_app". The top level package is called the *release package* and it contains the *app package* for the app and other files that are needed to distribute the app. The key components of the *release package* and the *app package* will be explained in this article.

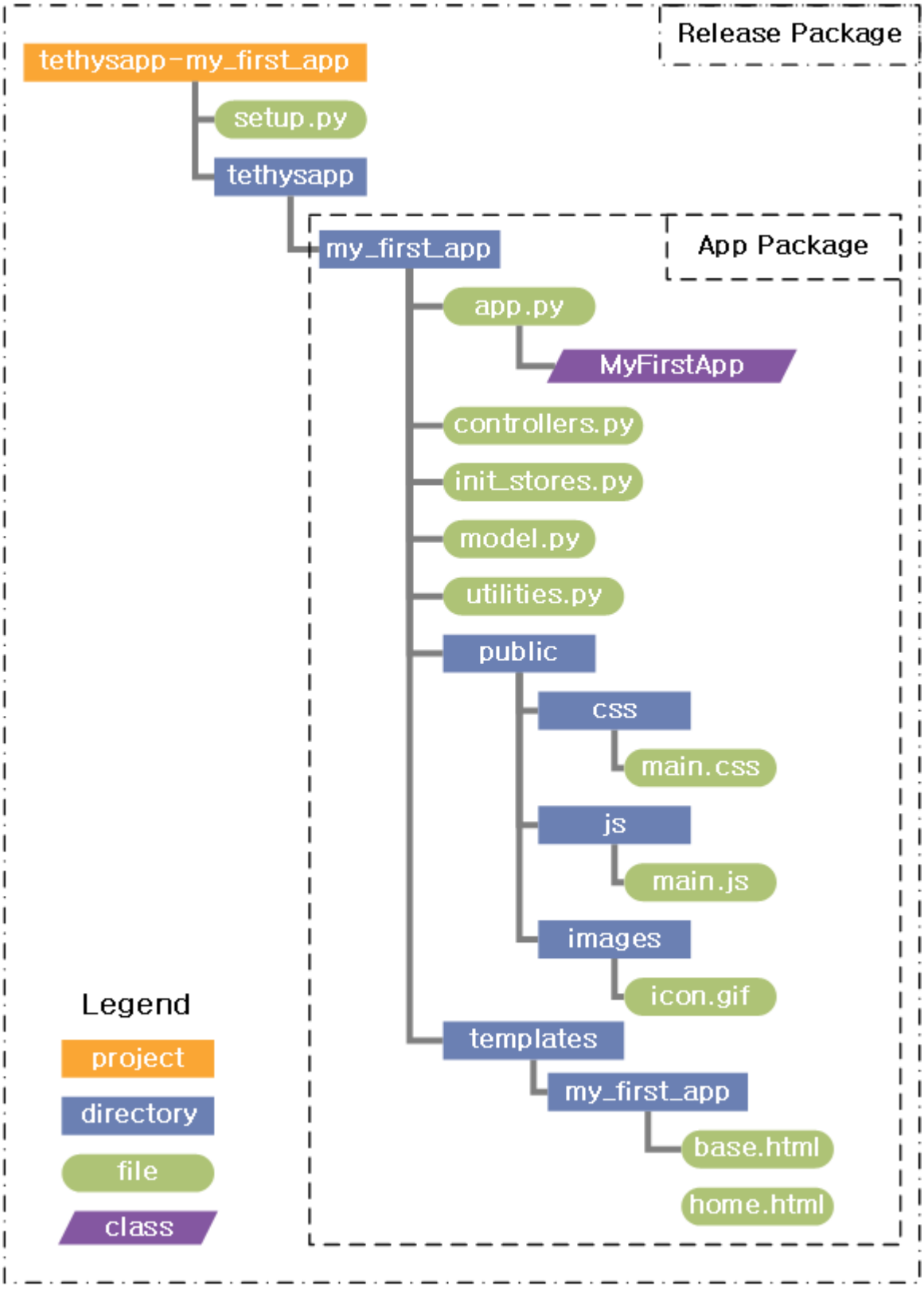


Fig. 4: **Figure 1. An example of a Tethys app project for an app named "my_first_app".**

Release Package

As the name suggests, the release package is the package that you will use to release and develop your app. The entire *release package* should be provided when you share your app with others.

The name of a *release package* follows a specific naming convention.The name of the directory should always start with "tethysapp-" followed by a *unique* name for the app. The name of the app may not have spaces, dashes, or other special characters (however, underscores are allowed). For example, Figure 1 shows the project structure for an app with name "my_first_app" and the name of the *release package* is "tethysapp-my_first_app".

The release package must contain a setup script (`setup.py`) and `tethysapp` namespace package at a minimum. This directory would also be a good place to put any accessory files for the app such as a README file or LICENSE file. No code that is required by the app to run should be in this directory.

The setup script to install your app and its dependencies. A basic setup script is generated as part of the scaffolding for a new app project. For more information on writing setup scripts refer to this article: Writing the Setup Script.

The `tethysapp` package is a Python namespace package. It provides a way to mimic the production environment during development of the app (i.e.: when the app is installed, it will reside in a namespace package called `tethysapp`). This package contains the *app package*, which has the same name as your app name by convention.

Caution: When you generate a new app project using the command line tool, you will notice that many of the directories contain a `__init__.py` file, many of which are empty. These are omitted in the diagram for simplicity. DO NOT DELETE THE `__init__.py` FILES. These files indicate to Python that the directories containing them are Python packages. Your app will not work properly without the `__init__.py` files.

The App Package

The *app package* contains all of the source code and resources that are needed by the Tethys Platform to run your app. The `model.py`, `templates`, and `controllers.py` modules and directories correspond with the Model View Controller approach that is used to build apps.

The data structures, classes, and methods that are used to define the data model `model.py` module. The `templates` directory contains all the Django HTML templates that are used to generate the views of the app. The `controllers.py` module contains Python files for each controller of the app. The `public` directory is used for static resources such as images, JavaScript and CSS files. The `app.py` file contains all the configuration parameters for the app.

To learn how to work with the files in the *app package*, see the *Getting Started* tutorial.

Naming Conventions

There are a few naming conventions that need to be followed to avoid conflicts with other apps. The more obvious one is the *app package* name. Like all Python modules, *app package* names must be unique.

All templates should be contained in a directory that shares the same name as the *app package* within the `templates` directory (see Figure 1). This ensures that when your app calls for a template like `home.html` it finds the correct one and not an `home.html` from another app.

1.10.3 Terminal Quick Guide

Last Updated: November 18, 2014

To install and use Tethys Platform, you will need to be familiar with using the command line/terminal. This guide provides tips and explanations of the most common features of command line that you will need to know to work with Tethys. For a more exhaustive reference, please review this excellent tutorial: Learn the Bash Command Line.

$

The "$" in code blocks means "run this in the terminal". This is usually done by typing the command or copying and pasting it into the terminal. When copying, don't copy the "$". Copy lines one at a time and press `enter` after each one to execute it. Note that some commands may prompt you for input.

~

The "~" is short hand for your `Home` directory. You will see this symbol most often in paths that extend from your `Home` directory. The shorthand is used because the path to the `Home` directory varies depending on your user name. For example, if your user name was "john", then the absolute path to your home directory would be something like `/home/john`.

sudo

Some operations on the commandline require authorization by a superuser or administrator. The `sudo` command is used to grant permission. This is done by prepending any command with `sudo`. You will be prompted for your password before you can continue.

```
sudo apt-get moo
```

Note: When you type passwords into the command line, the characters are not printed to the screen for security reasons. This can be unsettling, but type with faith and press enter.

---

cd

This command is used to change working directories on the command line. This is the equivalent of moving in and out of folders on a file browser.

mkdir

This command is used to make new directories.

chown

This command is used to change the owner of files or directories.

Copy and Paste

The keyboard shortcuts `CTRL-C` and `CTRL-V` do not do preform copy and paste in the terminal. Instead, use the shortcuts `CTRL-SHIFT-C` and `CTRL-SHIFT-V` to copy and paste.

1.10.4 Ubuntu Installation

Last Updated: November 17, 2014

Ubuntu Desktop can be downloaded at the Download Ubuntu page. There are three ways you can install Ubuntu on your computer. The first option is to overwrite whatever operating system you are running with Ubuntu.This can be done using either a USB or DVD. Use the Install Ubuntu instructions to do so (Note: these instructions are for Ubuntu 14.04, but they should work for Ubuntu 12.04 as well). This method is not usually preferable or recommended, because most users still want to retain use of their Windows or Mac operating systems. The next two options accomodate this need.

The second options is to install Ubuntu in a dual boot configuration. This will let you choose to either run Ubuntu or Windows/Mac OSX when you start your computer. Follow the instructions provided by Ubuntu for Windows Dual Boot if on a Windows computer or the Intel Mac Dual Boot if on a Mac computer.

The third option is to install Ubuntu as a virtual machine using virtualization software such as VirtualBox. If you are running Mac OSX you can also use VMWare or Parallels. Follow the instructions for creating a new Ubuntu virtual machine for the software you are running.

After installing Ubuntu, be sure to install any updates using the Update Manager and restart.

1.10.5 Test Docker Containers

If you would like, you may perform the following tests to ensure the containers are working properly.

Activate the virtual environment if you have not done so already and use the following Tethys command to start the Docker containers:

```
$ . /usr/lib/tethys/bin/activate
$ tethys docker start
```

Note: Although each Docker container seem to start instantaneously, it may take several minutes for the started containers to be fully up and running.

Use the following command in the terminal to obtain the ports that each software is running on:

```
$ tethys docker ip
```

You will be able to access each software on `localhost` at the appropriate port. For example, GeoServer and 52 North WPS both have web administrative interfaces. In a web browser, enter the following URLs replacing the `<port>` with the appropriate port number from the previous command:

```
# GeoServer
http://localhost:<port>/geoserver

# 52 North WPS
http://localhost:<port>/wps
```

With some luck, you should see the administrative page for each. Feel free to explore. You can login to the 52 North WPS admin site using the username and password you specified during installation or the defaults if you accepted those which are:

Default 52 North WPS Admin

- Username: wps
- Password: wps

You are not given the option of specifying a custom username and password for GeoServer, because it can only be done through the web interface. You may log into your GeoServer using the default username and password:

Default GeoServer Admin

- Username: admin
- Password: geoserver

The PostgreSQL database is installed with the database users and databases required by Tethys Platform: **tethys_default**, **tethys_db_manager**, and **tethys_super**. You set the passwords for each user during installation of the container. You can test the database by installing the PGAdmin III desktop client for PostgreSQL and using the credentials of the **tethys_super** database user to connect to it. For more detailed instructions on how to do this, see the *PGAdmin III Tutorial*.

1.10.6 PGAdmin III Tutorial

Last Updated: November 20, 2014

All of the SQL databases used in Tethys Platform are PostregSQL databases. An excellent graphical client for PostgreSQL. It is available for Windows, OSX, and many Linux distributions. Please visit the Download page to learn how to install it for your particular operating system. After it is installed, you can connect to your Tethys Platform databases by using the credentials for the `tethys_super` database user you defined during installation.

To create a new connection to your PostgreSQL database using PGAdmin:

1. Open PGAdmin III and click on the `Add New Connection` button.

2. In the New Server Registration dialog that appears, fill out the form with the appropriate credentials. Provide a meaningful name for the connection like "tethys". If you have installed PostgreSQL with the Docker containers, the host will be either `localhost` if you are on Linux or `192.168.59.103` if you are on Mac or Windows. Use the `tethys docker ip` command to get the port for PostgreSQL (PostGIS). Fill in the username as `tethys_super` and enter the password you gave the user during installation. Click `OK` to close the window.

3. To connect to the PostgreSQL database server, double-click on the "tethys" connection listed under the `Servers` dropdown menu. You will see a list of the databases on the server. Expand the menus to view each database. The tables will be located under `Schemas > public > Tables`.

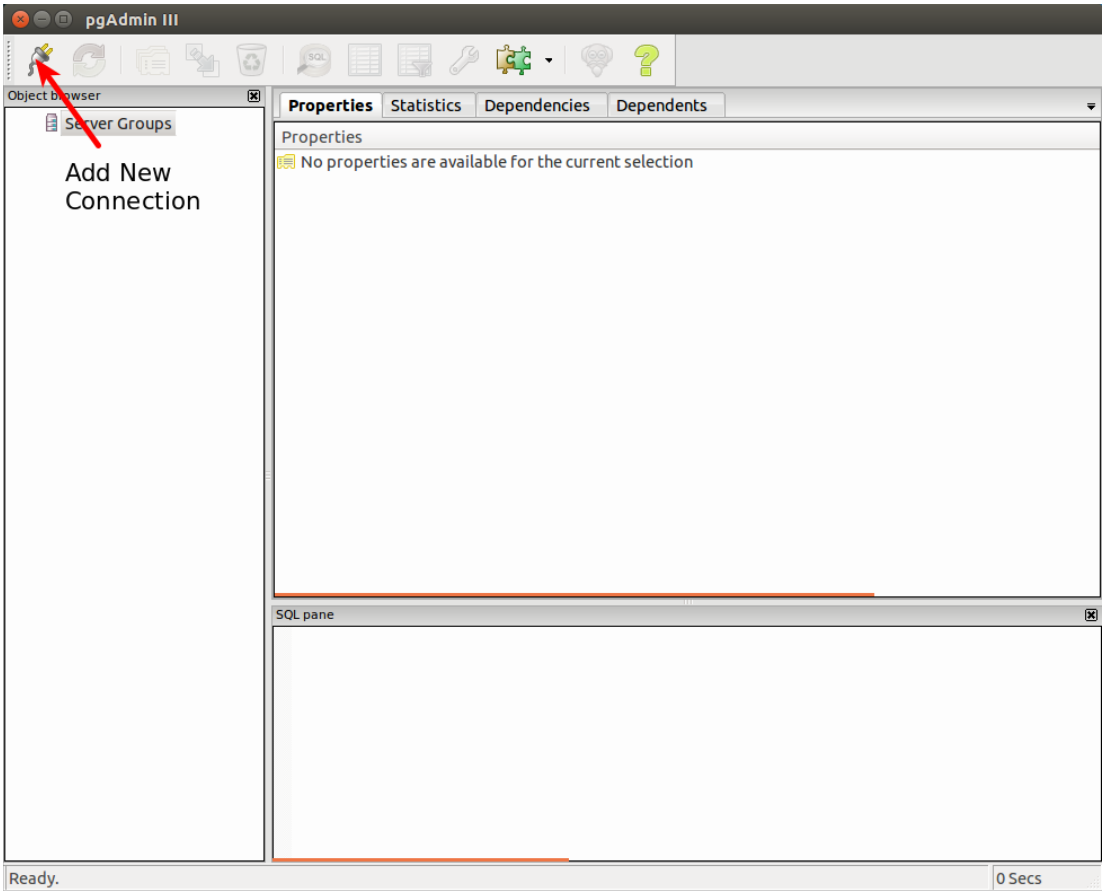


Fig. 5: **Figure 1.** Click the `Add New Connection` button.

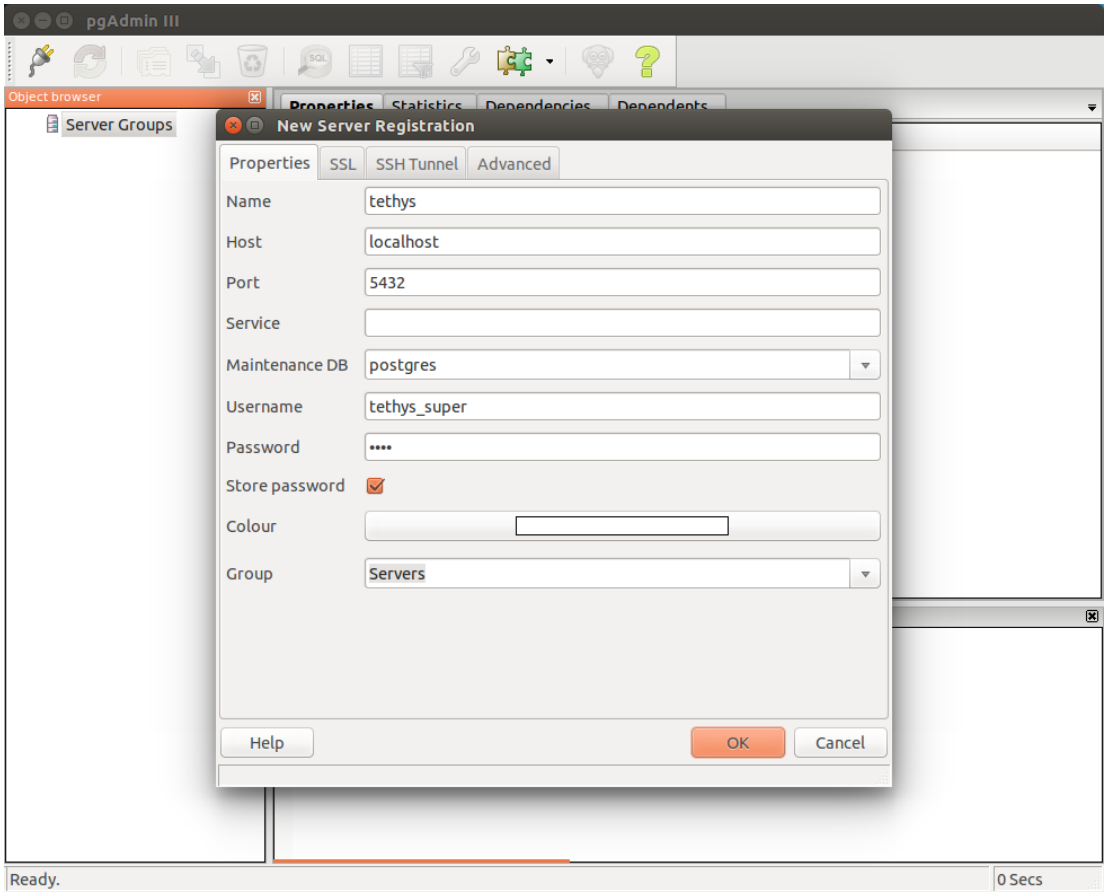


Fig. 6: **Figure 2.** Fill out the New Server Registration dialog.

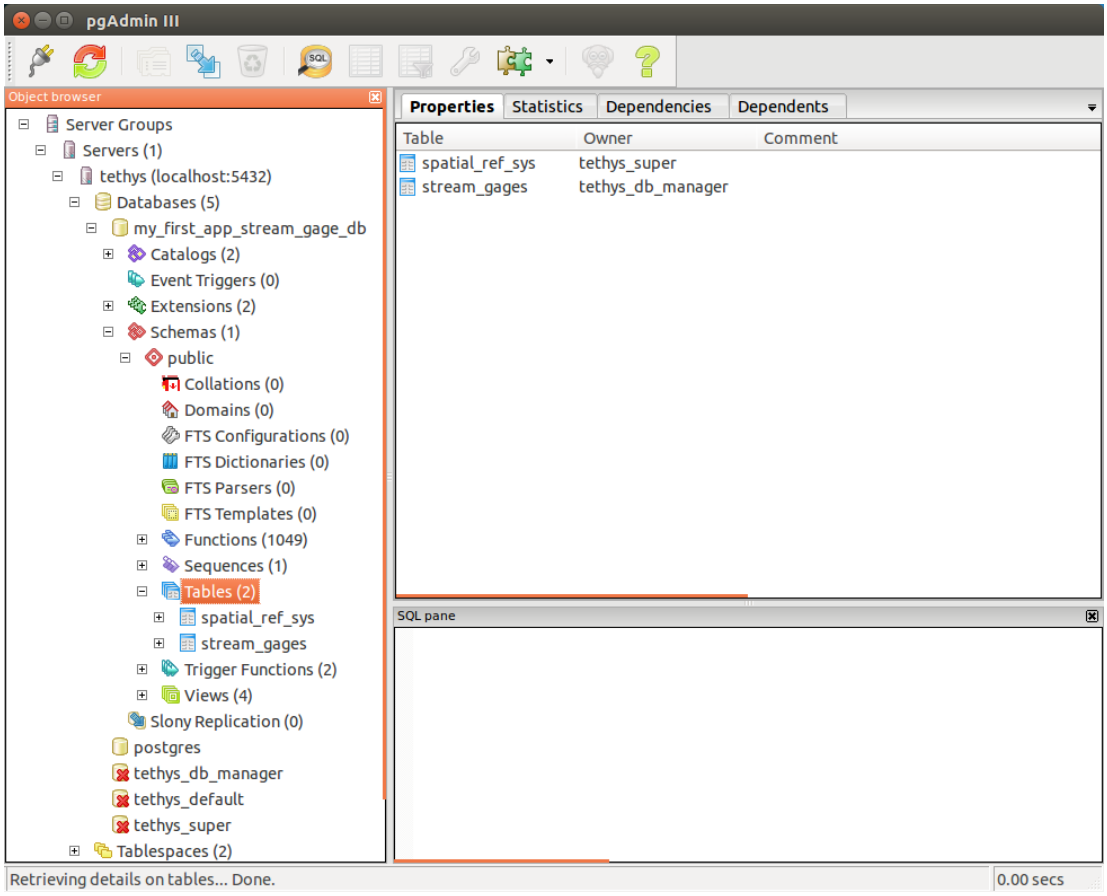


Fig. 7: **Figure 3.** Browse the databases using the graphical interface.

1.11 Summary of References

Last Updated: November 17, 2014

This page is provides a list of all the external resources referred too during the documentation for convenience.

Ubuntu

- Download Ubuntu
- Dual Boot Ubuntu with Windows PC
- Learn the Bash Command Line

Docker

- Docker virtualization container

Virtualization Options

- VMWare
- Parallels
- VirtualBox

Django

- Writing Views
- Django Template Language
- Django template Variables
- Django Filter Reference
- Django Tag Reference
- Django Template Inheritance
- Django static tag
- Cross Site Forgery protection

CKAN

- Install CKAN 2.2 from Source
- CKAN instances around the world
- FileStore Setup
- DataStore Setup
- Install on Other Operating Systems
- Actions API

IDEs

- How to Install Aptana on Ubuntu 12.04

SQLAlchemy

- SQLAlchemy
- Object Relational Tutorial

GeoAlchemy

- GeoAlchemy2
- GeoAlchemy ORM
- Well Known Text

Database Clients

- PGAdmin III

PostGIS

- PostGIS
- Geometry Function Reference
- Raster Function Reference

Google

- Obtaining an API Key

Python

- PyPI
- Setuptools Documentation
- Writing Setup Script
- Namespace

Production Installation

- WSGI
- modwsgi
- Deployment Checklist
- Deployment Checklist: STATIC_ROOT
- Deployment Checklist: SECRET_KEY
- Deployment Checklist: ALLOWED_HOSTS
- Deployment Checklist: STATIC_ROOT
- How to deploy with WSGI
-

Miscellaneous

- Universally unique identifier
- The Definitive Guide to GET vs POST

1.12 Glossary

Last Updated: May 2017

app class A class defined in the *app configuration file* that inherits from the `TethysAppBase` class provided by the Tethys Platform. For more details on the app class, see *App Base Class API*.

app configuration file A file located in the *app package* and called `app.py` by convention. This file contains the *app class* that is used to configure apps. For more details on the app configuration file, see *App Base Class API*.

app harvester An instance of the `SingletonAppHarvester` class. The app harvester collects information about each app and uses it to load Tethys apps.

app instance

app instances An instance of an *app class*.

app package

app packages A Python namespace package of a Tethys *app project* that contains all of the source code for an app. The app package is named the same as the app by convention. Refer to Figure 1 of *App Project Structure* for more information.

app project All of the source code for a Tethys app including the *release package* and the *app package*.

dataset

datasets A dataset is a container for one or more resources that are stored in a *dataset service*.

dataset service

dataset services A dataset service is a web service external to Tethys Platform that can be used to store and publish file-based datasets (e.g.: text files, Excel files, zip archives, other model files). See the *Dataset Services API* for more information.

Debian Debian is a type of Linux operating system and many Linux distributions are based on it including Ubuntu. See Linux Distributions for more information.

Gizmo

Gizmos Reusable view elements that can be inserted into a template using a single line of code. Examples include common GUI elements like buttons, toggle switches, and input fields as well as more complex elements like maps and plots. For more information on Gizmos, see *Template Gizmos API*.

Model View Controller The development pattern used to develop Tethys apps. The Model represents the data of the app, the View is composed of the representation of the data, and the Controller consists of the logic needed to prepare the data from the Model for the View and any other logic your app needs.

persistent store

persistent stores A persistent store is a database that can be automatically created for an app. See the *Persistent Stores API* for more information about persistent stores.

release package The top level Python namespace package of an *app project*. The release package contains the *setup script* and all the source for an app including the *app package*. Refer to Figure 1 of *App Project Structure* for more information.

resource

resources A resource is a file or other object and the associated metadata that is stored in a *dataset service*.

setup script A file located in the *release package* and called `setup.py` by convention. The setup script is used to automate the installation of apps.

spatial dataset

spatial datasets A spatial dataset is a file-based dataset that stores spatial data (e.g.: shapefiles, GeoTiff, ArcGrid, GRASS ASCII Grid).

virtual environment

Python conda environment An isolated Python installation. Many operating systems use the system Python installation to perform maintenance operations. Installing Tethys Platform in a conda environment prevents potential dependency conflicts. It also make it easier to install dependencies that have non-python dependencies (e.g. netCDF4, GDAL, arcgis)

wps service

wps services A WPS Service provides processes/geoprocesses as web services using the Open Geospatial Consortium Web Processing Service (WPS) standard.

CHAPTER

TWO

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1135482

CHAPTER

THREE

INDICES AND TABLES

- genindex
- search

INDEX

E

F

G

H

I

J

L

M

N

O

P

R

S

T

Z