Tethys Platform Documentation

Release 1.4.0

Nathan Swain

June 20, 2017

Contents

Last Updated: December 12, 2016

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 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. Refer to the *Features* article for an overview of the features of Tethys Platform.

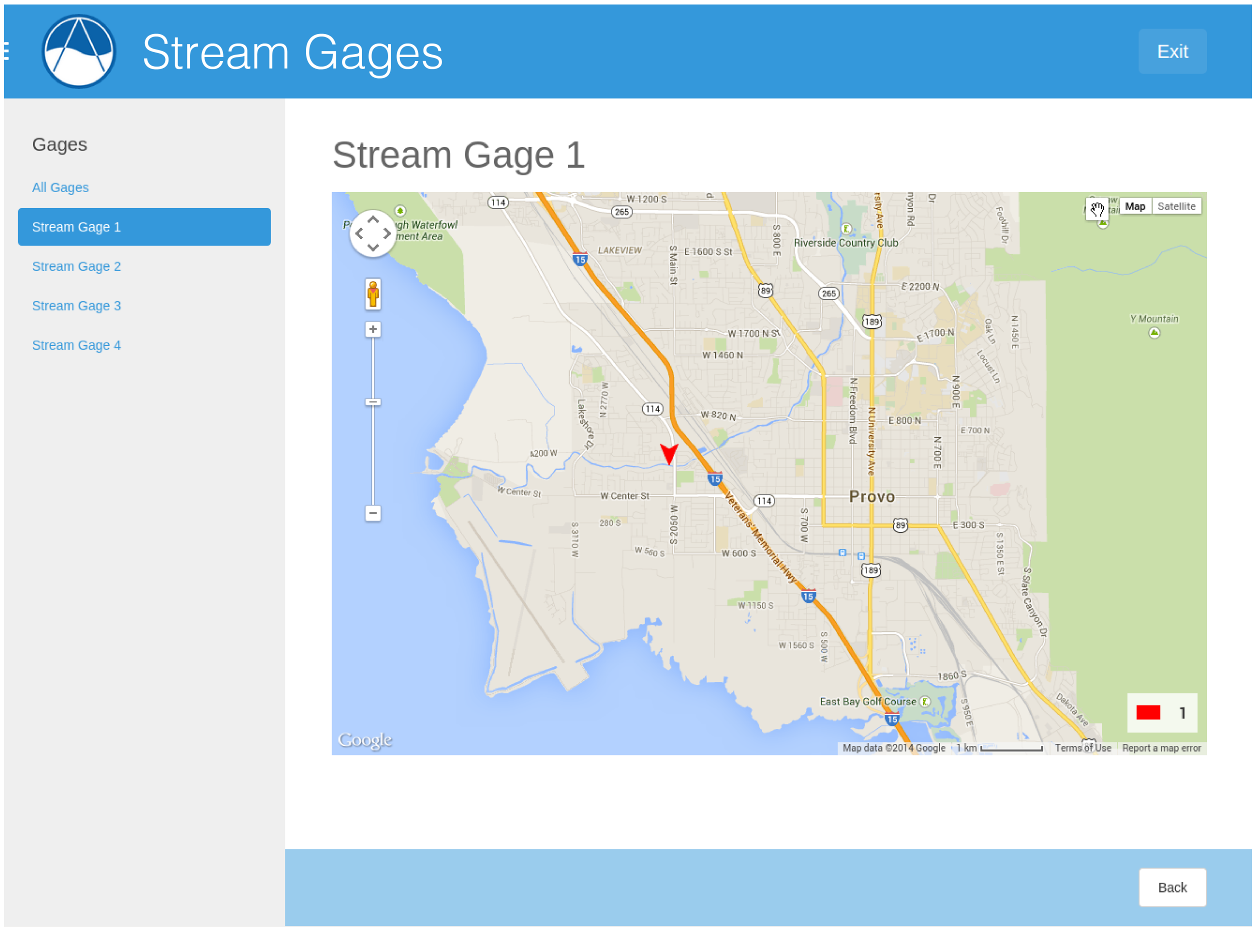
Important: Tethys Platform 1.4.0 has arrived! Check out the *What's New* article for a description of the new features and changes.

Contents

Features

Last Updated: May 28, 2015

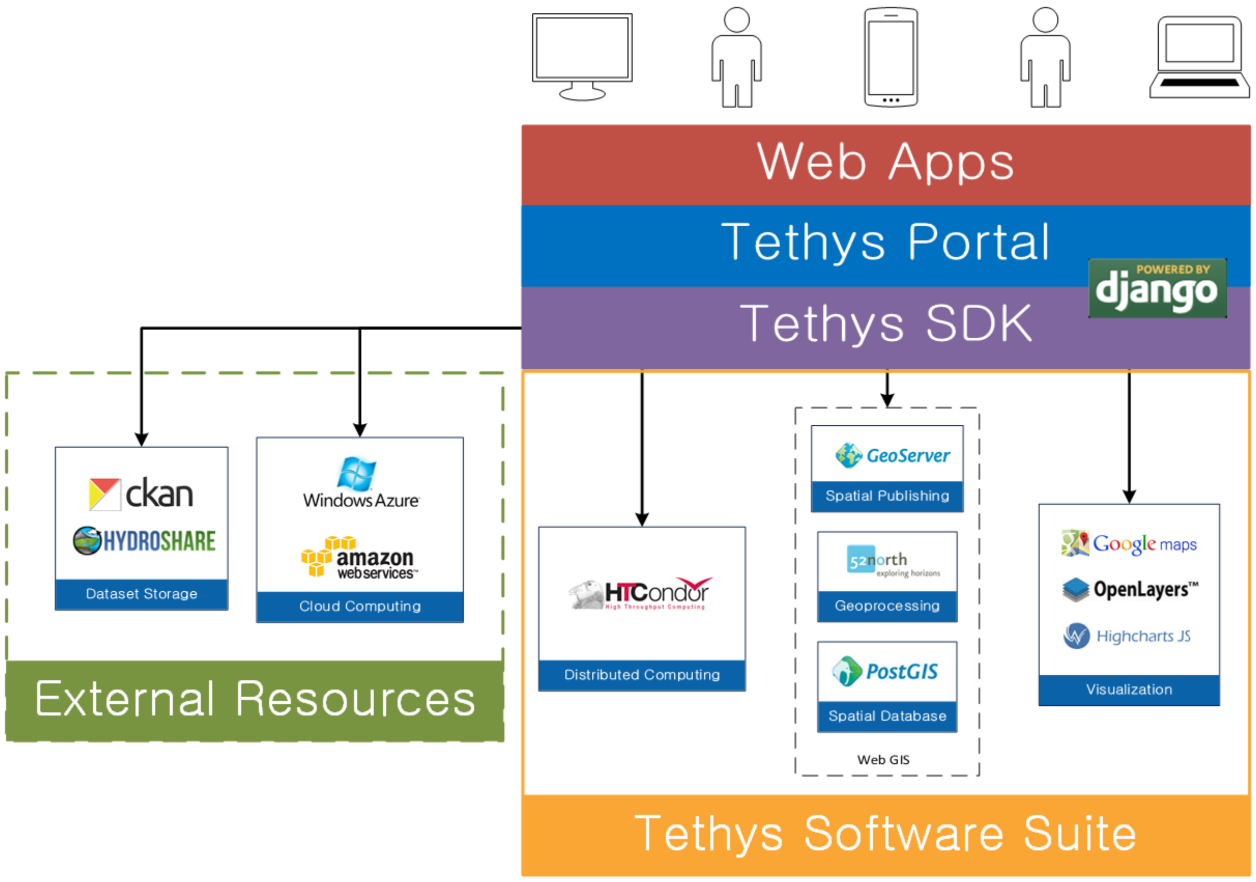
Tethys is a platform that can be used to develop and host engaging, interactive water resources web applications or web apps. It includes a suite of free and open source software (FOSS) that has been carefully selected to address the unique development needs of water resources web apps. Tethys web apps 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.

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.



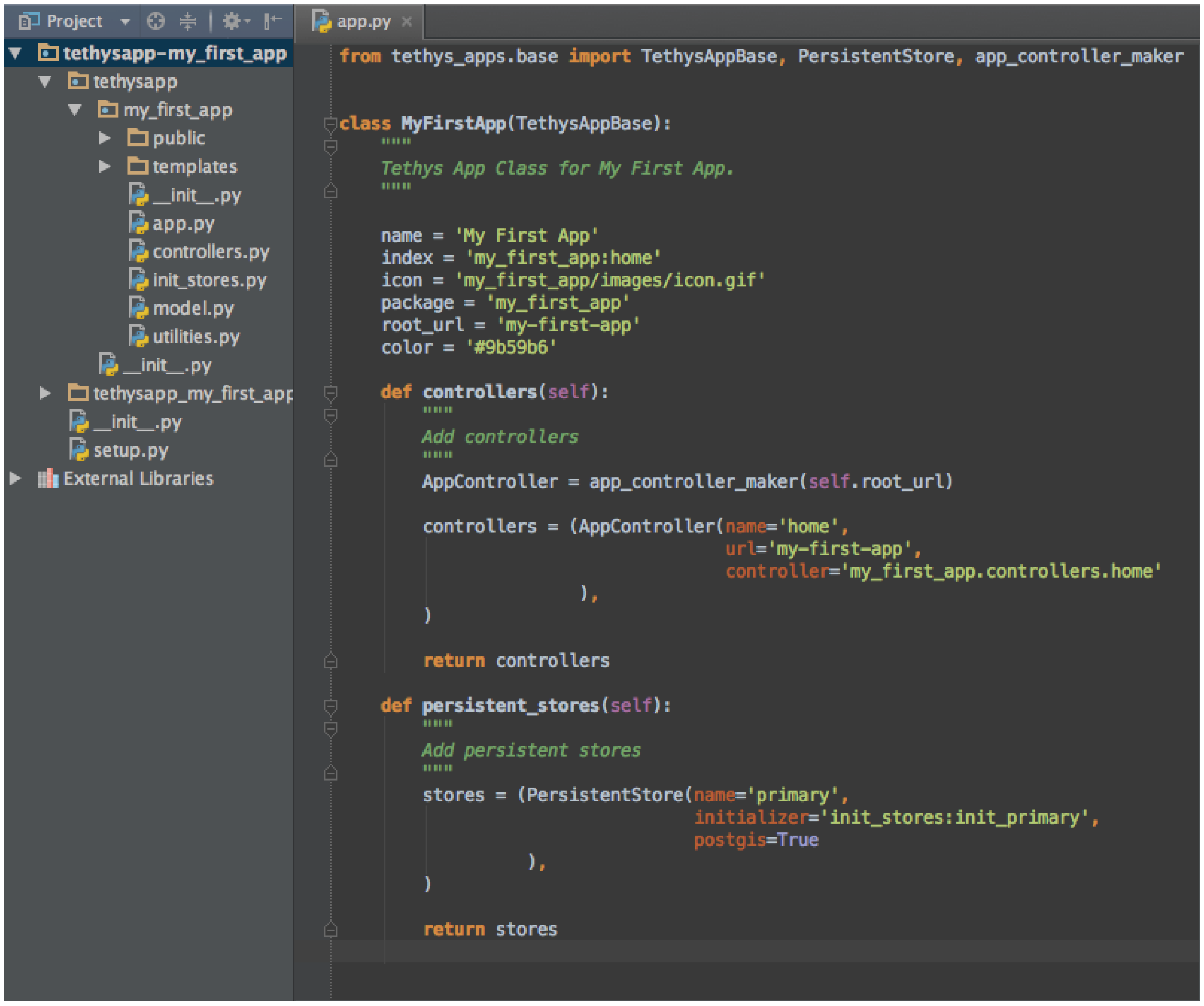
Tethys Platform include software to meet water resources web app development needs.

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

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.

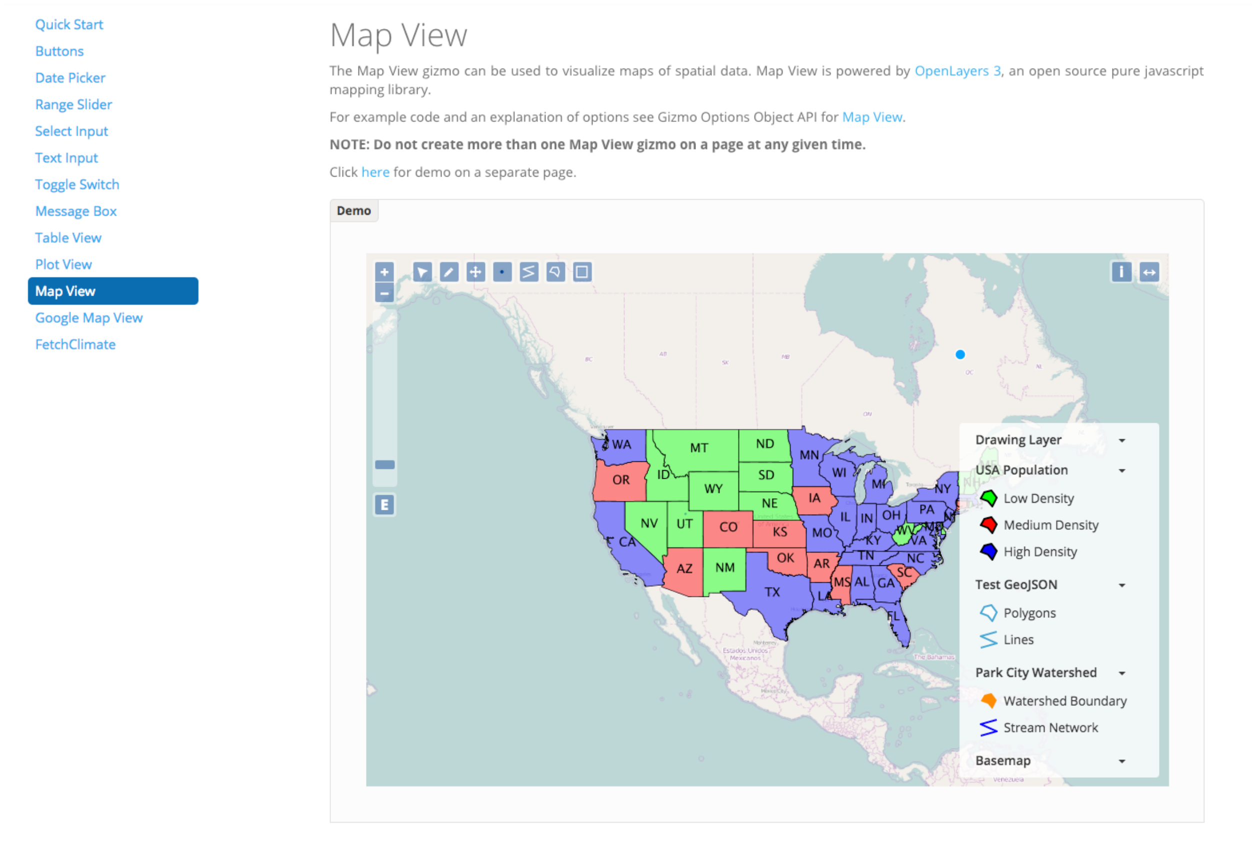


```python
from tethys_apps.base import TethysAppBase, PersistentStore, app_controller_maker


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
```

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

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.



Map View

The Map View gizmo can be used to visualize maps of spatial data. Map View is powered by OpenLayers 3, an open source pure javascript mapping library.

For example code and an explanation of options see Gizmo Options Object API for Map View.

NOTE: Do not create more than one Map View gizmo on a page at any given time.

Click here for demo on a separate page.

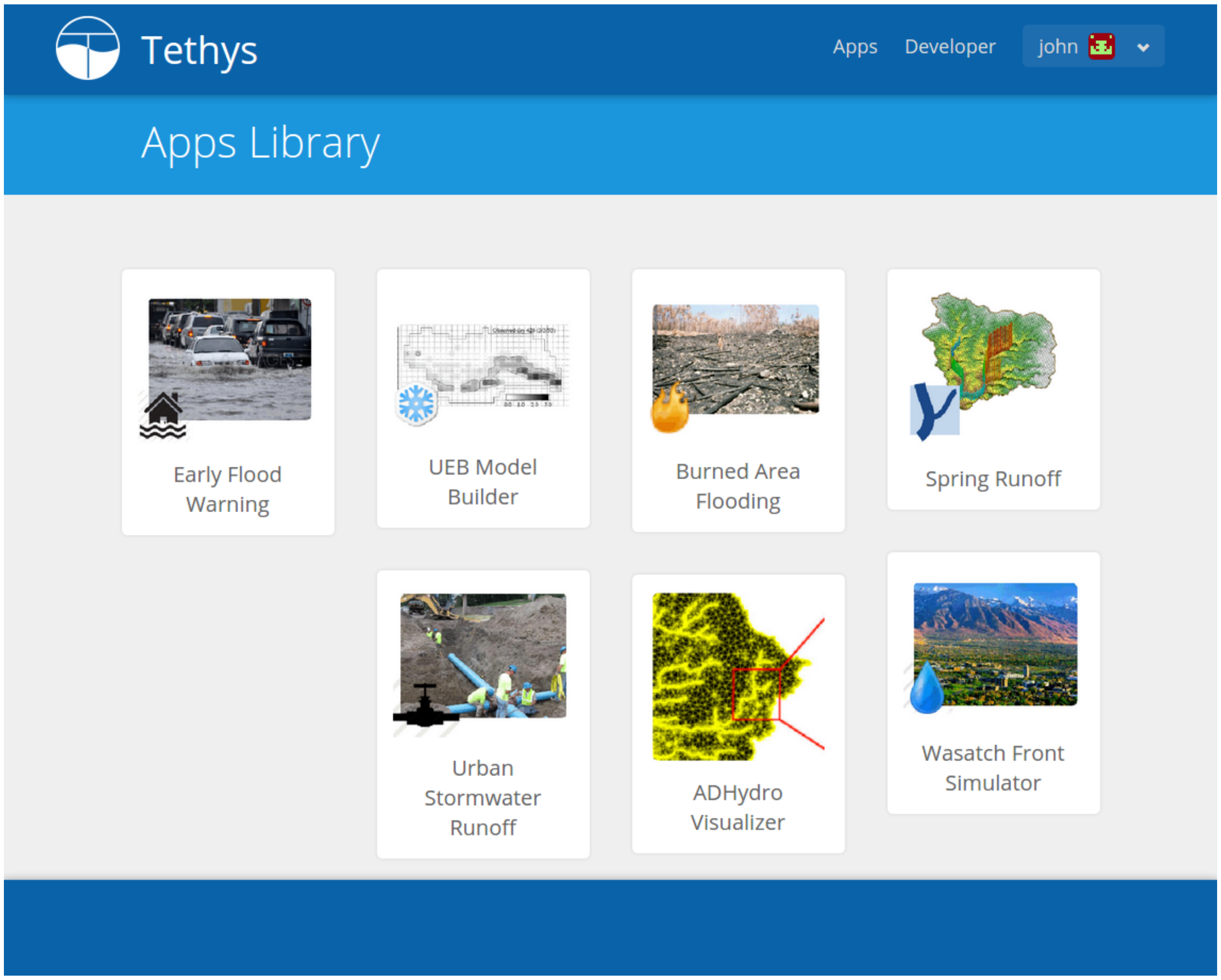
Insert common user interface elements like date pickers, maps, and plots with minimal coding.

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

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.

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

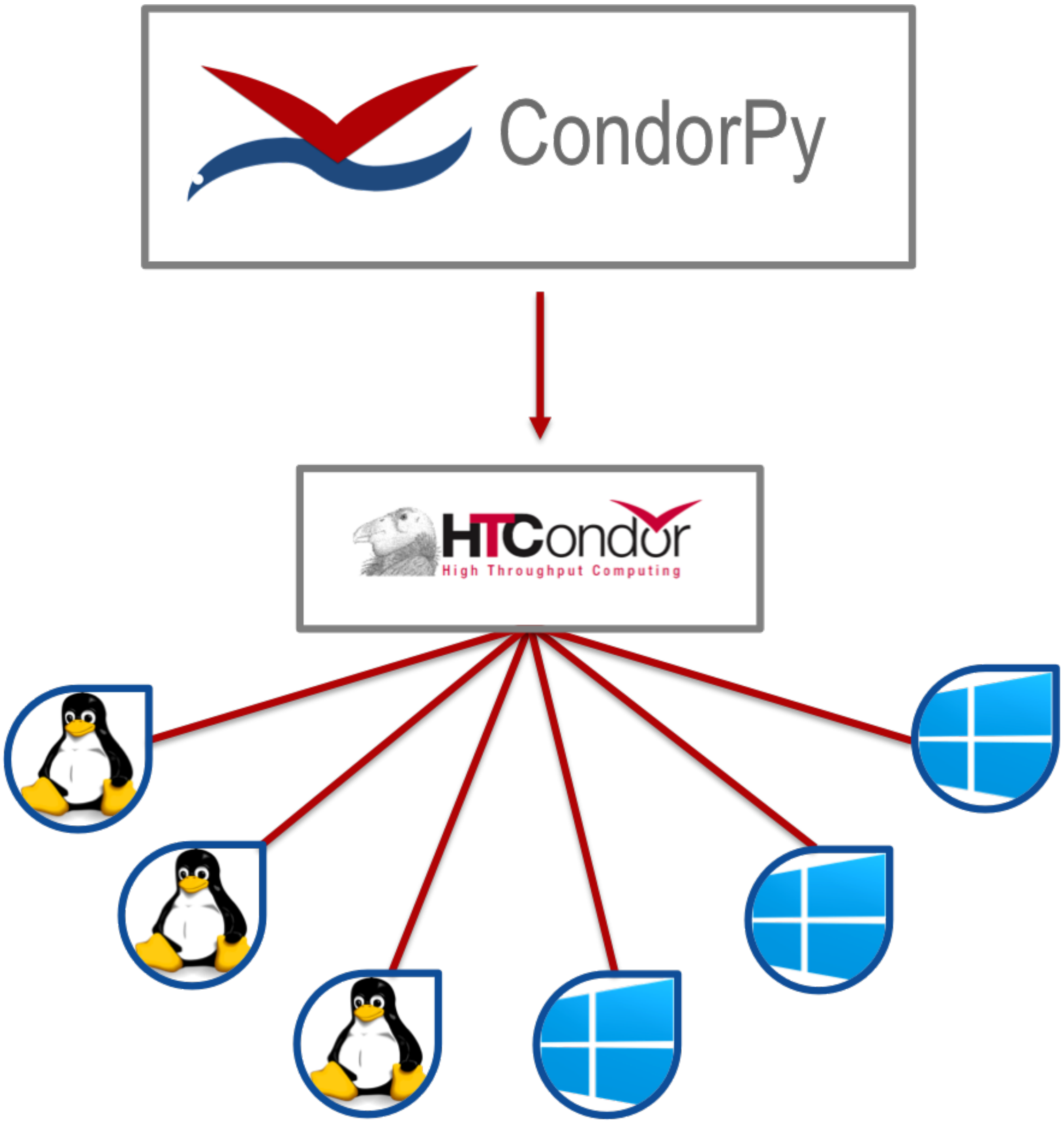
Computing

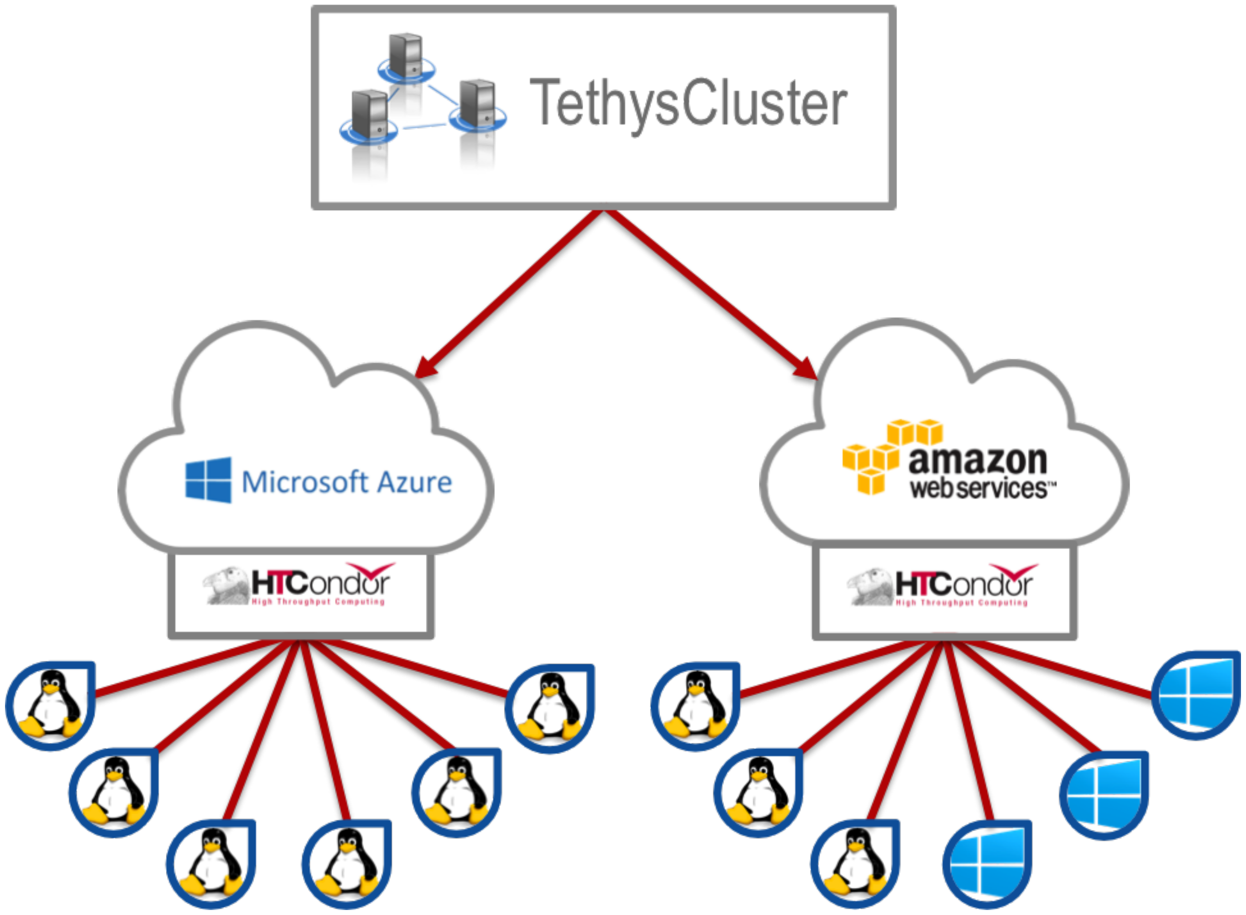
Tethys Platform includes Python modules that allow you to provision and run computing jobs in distributed computing environments. With CondorPy you can define your computing jobs and submit them to distributed computing environments provided by HTCondor.

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. Alternatively, TethysCluster enables you to provision scalable computing resources in the cloud using commercial services like Amazon AWS and Microsoft Azure.

TethysCluster makes it easy to scale your computing resources using commercial cloud services.


CondorPy
HTCondor
High Throughput Computing


TethysCluster
Microsoft Azure
amazon web services
HTCondor High Throughput Computing
HTCondor High Throughput Computing

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

Acknowledgements

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

What's New

Last Updated: December 1, 2016

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

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.

See: *Permissions API*

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.

See: *Manage Terms and Conditions*

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

See: `software_suite/geoserver`

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

See: *docker <subcommand> [options]*

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.

See: *Select Input*

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.

See: *Map View*

PlotView Gizmo

- New JavaScript API endpoints for initializing PlotViews dynamically.

See: *Plot View*

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.

See: *Condor Workflow Job Type*

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.

See: *Testing API* and *test [options]*

Installation

- Installation Instructions for Ubuntu 16.04

See: *Installation on 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)

Prior Release Notes

Prior Release Notes

Last Updated: May 28, 2016

Information about prior releases is shown here.

Release 1.3.0

Tethys Portal

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

See: *Customize*

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

See: *Map View*

Plot View

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

See: *Plot View*

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.

See: *GeoServer Spatial Dataset Engine Reference*

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

See: *App Feedback*

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

See: *Handoff API*

Jobs Table Gizmo

- The refresh interval for job status and runtime is configurable

See: *Jobs Table*

Social Authentication

- Support for HydroShare added

See: *Social Authentication*

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.

See: *Persistent Stores API*

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.

See: *App Base Class API*

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

See: *Social Authentication*

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

See: *Plot View*

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

See: *Workspaces API*

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

See: *Handoff API*

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

See: *Video Tutorials*

New Location for Tethys SDK

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

See: *Software Development Kit*

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.

See: *Persistent Stores API*

Command Line Interface

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

See: *Command Line Interface*

Release 1.1.0

Gizmos

- Options objects for configuring gizmos (see *Template Gizmos API* for more details).
- Many improvements to Map View (see *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 (see *Plot View*)
  - 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

Installation

Last Updated: July 1, 2016

This section describes how to install Tethys Platform. Installation instructions are provided for Ubuntu 14.04 and 16.04.

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.

Installation on Ubuntu 14.04

Last Updated: December 12, 2016

Warning: These installation instructions have been tested for Ubuntu 14.04 only. It is likely that you will encounter problems if you try to use these instructions on any other Linux distribution (e.g. RedHat, CentOS) or even other versions of Ubuntu. The current release of Ubuntu is 16.04, so use the Alternative Downloads page to download and install Ubuntu 14.04.

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.

Also, check to make sure that your installation of Ubuntu = version 14.04. The following steps are likely not to work with other versions.

1. Install the Dependencies

1. Install most of the dependencies via **apt-get**. Open a terminal and execute the following commands:

```
$ sudo apt-get update
$ sudo apt-get install python-dev python-pip python-virtualenv libpq-dev libxml2-dev libxslt1-de
```

   You may be prompted to enter your password to authorize the installation of these packages. If you are prompted about the disk space that will be used to install the dependencies, enter `Y` and press `Enter` to continue.

2. Finish the Docker Installation

There are a few additional steps that need to be completed to finish the installation of Docker.

1. Execute the following command to finish the installation of Docker:

```
$ source /etc/bash_completion.d/docker
```

   Note: If running this command and those that follow with **docker** doesn't work, try substituting **docker.io** instead.

2. Add your user to the Docker group. This is necessary to use the Tethys Docker commandline tools. In a command prompt execute:

```
$ sudo gpasswd -a ${USER} docker
$ sudo service docker restart
$ gnome-session-quit --logout
```

3. Select **log out** and then **log back in** to make the changes take effect.

Important: **DO NOT FORGET PART C!** Be sure to logout of Ubuntu and log back in before you continue. You will not be able to complete the installation without completing this step.

Warning: Adding a user to the Docker group is the equivalent of declaring a user as root. See Giving non-root access for more details.

3. Create Virtual Environment and Install Tethys Platform

Python virtual environments are used to create isolated Python installations to avoid conflicts with dependencies of other Python applications on the same system. The following commands should be executed in a terminal.

1. Create a *Python virtual environment* and activate it:

   ```
   $ sudo mkdir -p /usr/lib/tethys
   $ sudo chown `whoami` /usr/lib/tethys
   $ virtualenv --no-site-packages /usr/lib/tethys
   $ . /usr/lib/tethys/bin/activate
   ```

Hint: You may be tempted to enter single quotes around the *whoami* directive above, but those characters are actually grave accent characters: **`**. This key is usually located to the left of the `1` key or in that vicinity.

Important: The final command above activates the Python virtual environment for Tethys. You will know the virtual environment is active, because the name of it will appear in parenthesis in front of your terminal cursor:

```
(tethys) $ _
```

The Tethys virtual environment must remain active for the entire installation. If you need to logout or close the terminal in the middle of the installation, you will need to reactivate the virtual environment. This can be done at anytime by executing the following command (don't forget the dot):

```
$ . /usr/lib/tethys/bin/activate
```

If you get tired of typing `.  /usr/lib/tethys/bin/activate` to activate your virtual environment, you can add an alias to your `.bashrc` file:

```
$ echo "alias t='. /usr/lib/tethys/bin/activate'" >> ~/.bashrc
```

Close your terminal window and reopen it to effect the changes. Now, to activate your virtual environment all you have to do is use the alias `t`:

```
  $ t
(tethys) $ _
```

2. Install Tethys Platform into the virtual environment with the following command:

   ```
   (tethys) $ git clone https://github.com/tethysplatform/tethys /usr/lib/tethys/src
   ```

Tip: If you would like to install a different version of Tethys Platform, you can use git to checkout the tagged release branch. For example, to checkout version 1.0.0:

```
$ cd /usr/lib/tethys/src
$ git checkout tags/1.0.0
```

For a list of all tagged releases, see Tethys Platform Releases. Depending on the version you intend to install, you may need to delete your entire virtual environment (i.e.: the `/usr/lib/tethys` directory) to start fresh.

3. Install the Python modules that Tethys requires:

   ```
   (tethys) $ pip install --upgrade -r /usr/lib/tethys/src/requirements.txt
   (tethys) $ python /usr/lib/tethys/src/setup.py develop
   ```

4. Restart the Python virtual environment:

```
(tethys) $ deactivate
         $ . /usr/lib/tethys/bin/activate
```

4. Install Tethys Software Suite Docker Containers

Execute the following Tethys commands using the **tethys** *Command Line Interface* to initialize the Docker containers:

```
(tethys) $ tethys docker init
```

You will be prompted to enter various parameters needed to customize your instance of the software. **Take note of the usernames and passwords that you specify**. You will need them to complete the installation.

Tip: Running into errors with this command? Make sure you have completed all of step 2, including part c.

Occasionally, you may encounter an error due to poor internet connection. Run the `tethys docker init` command repeatedly. It will pick up where it left off and eventually lead to success. When in doubt, try, try again.

5. Start the Docker Containers

Use the following Tethys command to start the Docker containers:

```
(tethys) $ tethys docker start
```

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

6. Create Settings File and Configure Settings

In the next steps you will configure your Tethys Platform and link it to each of the software in the software suite. Create a new settings file for your Tethys Platform installation using the **tethys** *Command Line Interface*. Execute the following command in the terminal:

```
(tethys) $ tethys gen settings -d /usr/lib/tethys/src/tethys_apps
```

This will create a file called `settings.py` in the directory `/usr/lib/tethys/src/tethys_apps`. As the name suggests, the `settings.py` file contains all of the settings for the Tethys Platform. There are a few settings that need to be configured in this file.

Note: The `usr` directory is located in the root directory which can be accessed using a file browser and selecting `Computer` from the menu on the left.

Open the `settings.py` file that you just created (`/usr/lib/tethys/src/tethys_apps/settings.py`) in a text editor and modify the following settings appropriately.

1. Run the following command to obtain the host and port for Docker running the database (PostGIS). You will need these in the following steps:

   ```
   (tethys) $ tethys docker ip
   ```

2. Replace the password for the main Tethys Portal database, **tethys_default**, with the password you created in the previous step. Also make sure that the host and port match those given from the `tethys docker ip` command (PostGIS). This is done by changing the values of the PASSWORD, HOST, and PORT parameters of the DATABASES setting:

```
DATABASES = {
  'default': {
      'ENGINE': 'django.db.backends.postgresql_psycopg2',
      'NAME': 'tethys_default',
      'USER': 'tethys_default',
      'PASSWORD': 'pass',
      'HOST': '127.0.0.1',
      'PORT': '5435'
      }
}
```

3. Find the TETHYS_DATABASES setting near the bottom of the file and set the PASSWORD parameters with the passwords that you created in the previous step. If necessary, also change the HOST and PORT to match the host and port given by the `tethys docker ip` command for the database (PostGIS):

```
TETHYS_DATABASES = {
    'tethys_db_manager': {
        'NAME': 'tethys_db_manager',
        'USER': 'tethys_db_manager',
        'PASSWORD': 'pass',
        'HOST': '127.0.0.1',
        'PORT': '5435'
    },
    'tethys_super': {
        'NAME': 'tethys_super',
        'USER': 'tethys_super',
        'PASSWORD': 'pass',
        'HOST': '127.0.0.1',
        'PORT': '5435'
    }
}
```

4. Setup social authentication

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

5. Save your changes and close the `settings.py` file.

7. Create Database Tables

Execute the following command to initialize the database tables:

```
(tethys) $ tethys manage syncdb
```

8. Create a Superuser

Create a superuser/website administrator for your Tethys Portal:

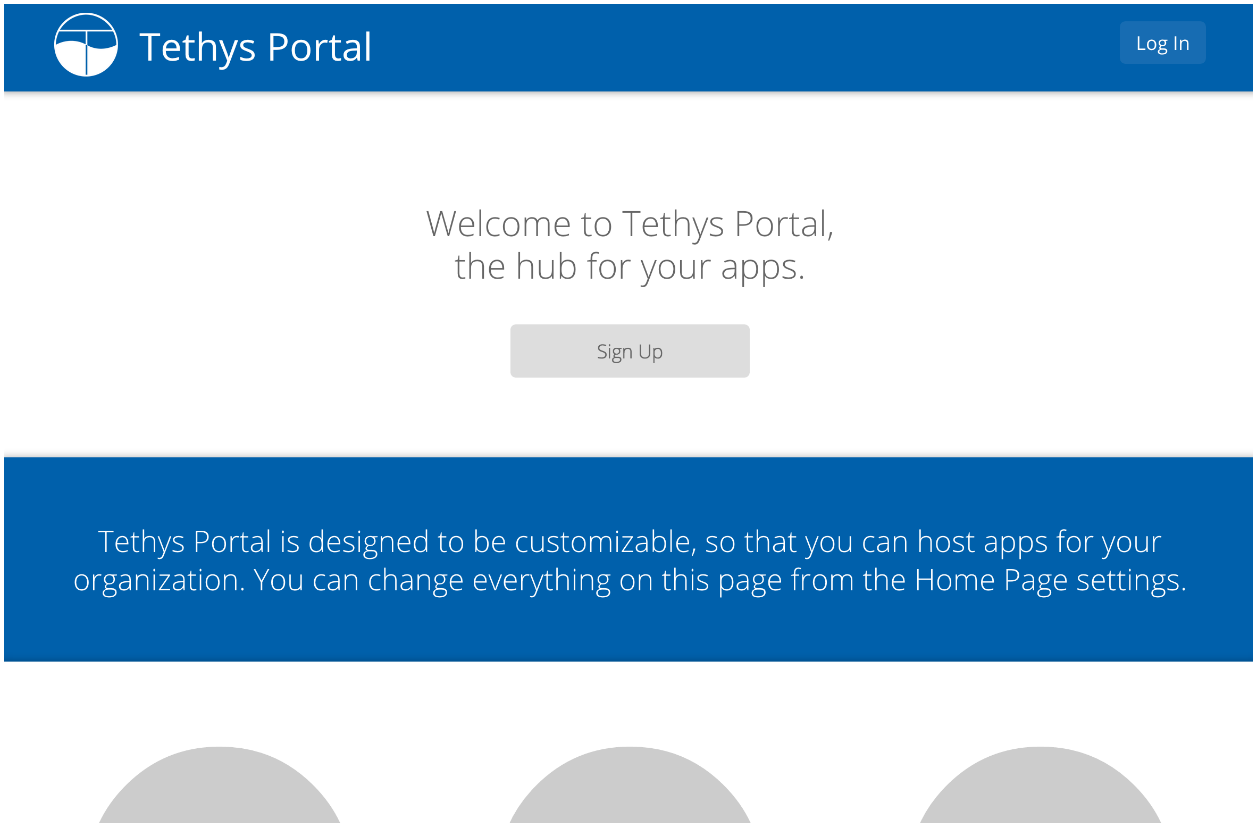
```
(tethys) $ tethys manage createsuperuser
```

9. Start up the Django Development Server

You are now ready to start the development server and view your instance of Tethys Platform. The website that ships with Tethys Platform is called *Tethys Portal*. In the terminal, execute the following command to start the development server:

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

Open http://localhost:8000/ in a new tab in your web browser and you should see the default *Tethys Portal* landing page.



Welcome to Tethys Portal, the hub for your apps.

Tethys Portal is designed to be customizable, so that you can host apps for your organization. You can change everything on this page from the Home Page settings.

9. Web Admin Setup

You are now ready to configure your Tethys Platform installation using the web admin interface. Follow the *Web Admin Setup* instructions to finish setting up your Tethys Platform.

Installation on Ubuntu 16.04

Last Updated: July 1, 2016

Warning: These installation instructions have been tested for Ubuntu 16.04 only. It is likely that you will encounter problems if you try to use these instructions on any other Linux distribution (e.g. RedHat, CentOS) or even other versions of Ubuntu.

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.

Also, check to make sure that your installation of Ubuntu = version 16.04. The following steps are likely not to work with other versions.

1. Install the Dependencies

1. Install most of the dependencies via **apt-get**. Open a terminal and execute the following commands:

   ```
   $ sudo apt-get update
   $ sudo apt-get install python-dev python-pip python-virtualenv libpq-dev libxml2-dev libxslt1-de
   ```

   You may be prompted to enter your password to authorize the installation of these packages. If you are prompted about the disk space that will be used to install the dependencies, enter `Y` and press `Enter` to continue.

2. Install Docker

Docker needs to be installed to install the Tethys Software Suite. These instructions are adapted from the Installation on Ubuntu Docker tutorial and the How to Install and Use Docker on Ubuntu 16.04 Digital Ocean tutorial.

1. Add the GPG key for the official Docker repository:

   ```
   $ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys 58118E89F3A91289
   ```

2. Add the Docker repository to APT sources:

   ```
   $ echo "deb https://apt.dockerproject.org/repo ubuntu-xenial main" | sudo tee /etc/apt/sources.l
   ```

3. Update APT sources again and install Docker engine:

   ```
   $ sudo apt-get update
   $ sudo apt-get install -y docker-engine
   ```

4. Add your user to the Docker group. This is necessary to use the Tethys Docker commandline tools. In a command prompt execute:

   ```
   $ sudo gpasswd -a ${USER} docker
   $ sudo service docker restart
   $ gnome-session-quit --logout
   ```

5. Select **log out** and then **log back in** to make the changes take effect.

Important: DO NOT FORGET PART E! Be sure to logout of Ubuntu and log back in before you continue. You will not be able to complete the installation without completing this step.

Warning: Adding a user to the Docker group is the equivalent of declaring a user as root. See Giving non-root access for more details.

3. Create Virtual Environment and Install Tethys Platform

Python virtual environments are used to create isolated Python installations to avoid conflicts with dependencies of other Python applications on the same system. The following commands should be executed in a terminal.

1. Create a *Python virtual environment* and activate it:

   ```
   $ sudo mkdir -p /usr/lib/tethys
   $ sudo chown `whoami` /usr/lib/tethys
   $ virtualenv --no-site-packages /usr/lib/tethys
   $ . /usr/lib/tethys/bin/activate
   ```

Hint: You may be tempted to enter single quotes around the *whoami* directive above, but those characters are actually grave accent characters: `. This key is usually located to the left of the 1 key or in that vicinity.

Important: The final command above activates the Python virtual environment for Tethys. You will know the virtual environment is active, because the name of it will appear in parenthesis in front of your terminal cursor:

```
(tethys) $ _
```

The Tethys virtual environment must remain active for the entire installation. If you need to logout or close the terminal in the middle of the installation, you will need to reactivate the virtual environment. This can be done at anytime by executing the following command (don't forget the dot):

```
$ . /usr/lib/tethys/bin/activate
```

If you get tired of typing `. /usr/lib/tethys/bin/activate` to activate your virtual environment, you can add an alias to your `.bashrc` file:

```
$ echo "alias t='. /usr/lib/tethys/bin/activate'" >> ~/.bashrc
```

Close your terminal window and reopen it to effect the changes. Now, to activate your virtual environment all you have to do is use the alias `t`:

```
  $ t
(tethys) $ _
```

2. Install Tethys Platform into the virtual environment with the following command:

```
(tethys) $ git clone https://github.com/tethysplatform/tethys /usr/lib/tethys/src
```

Tip: If you would like to install a different version of Tethys Platform, you can use git to checkout the tagged release branch. For example, to checkout version 1.0.0:

```
$ cd /usr/lib/tethys/src
$ git checkout tags/1.0.0
```

For a list of all tagged releases, see Tethys Platform Releases. Depending on the version you intend to install, you may need to delete your entire virtual environment (i.e.: the `/usr/lib/tethys` directory) to start fresh.

3. Install the Python modules that Tethys requires:

```
(tethys) $ pip install --upgrade -r /usr/lib/tethys/src/requirements.txt
(tethys) $ python /usr/lib/tethys/src/setup.py develop
```

4. Restart the Python virtual environment:

```
(tethys) $ deactivate
         $ . /usr/lib/tethys/bin/activate
```

4. Install Tethys Software Suite Docker Containers

Execute the following Tethys commands using the **tethys** *Command Line Interface* to initialize the Docker containers:

```
(tethys) $ tethys docker init
```

You will be prompted to enter various parameters needed to customize your instance of the software. **Take note of the usernames and passwords that you specify**. You will need them to complete the installation.

Tip: Running into errors with this command? Make sure you have completed all of step 2, including part c.

Occasionally, you may encounter an error due to poor internet connection. Run the `tethys docker init` command repeatedly. It will pick up where it left off and eventually lead to success. When in doubt, try, try again.

5. Start the Docker Containers

Use the following Tethys command to start the Database Docker container for the next steps:

```
(tethys) $ tethys docker start -c postgis
```

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

6. Create Settings File and Configure Settings

In the next steps you will configure your Tethys Platform and link it to each of the software in the software suite. Create a new settings file for your Tethys Platform installation using the **tethys** *Command Line Interface*. Execute the following command in the terminal:

```
(tethys) $ tethys gen settings -d /usr/lib/tethys/src/tethys_apps
```

This will create a file called `settings.py` in the directory `/usr/lib/tethys/src/tethys_apps`. As the name suggests, the `settings.py` file contains all of the settings for the Tethys Platform. There are a few settings that need to be configured in this file.

Note: The `usr` directory is located in the root directory which can be accessed using a file browser and selecting `Computer` from the menu on the left.

Open the `settings.py` file that you just created (`/usr/lib/tethys/src/tethys_apps/settings.py`) in a text editor and modify the following settings appropriately.

1. Run the following command to obtain the host and port for Docker running the database (PostGIS). You will need these in the following steps:

   ```
   (tethys) $ tethys docker ip
   ```

2. Replace the password for the main Tethys Portal database, **tethys_default**, with the password you created in the previous step. Also make sure that the host and port match those given from the `tethys docker ip` command (PostGIS). This is done by changing the values of the PASSWORD, HOST, and PORT parameters of the DATABASES setting:

   ```
   DATABASES = {
     'default': {
         'ENGINE': 'django.db.backends.postgresql_psycopg2',
         'NAME': 'tethys_default',
         'USER': 'tethys_default',
         'PASSWORD': 'pass',
         'HOST': '127.0.0.1',
         'PORT': '5435'
         }
   }
   ```

3. Find the TETHYS_DATABASES setting near the bottom of the file and set the PASSWORD parameters with the passwords that you created in the previous step. If necessary, also change the HOST and PORT to match the host and port given by the `tethys docker ip` command for the database (PostGIS):

   ```
   TETHYS_DATABASES = {
       'tethys_db_manager': {
           'NAME': 'tethys_db_manager',
           'USER': 'tethys_db_manager',
           'PASSWORD': 'pass',
           'HOST': '127.0.0.1',
           'PORT': '5435'
       },
       'tethys_super': {
           'NAME': 'tethys_super',
           'USER': 'tethys_super',
           'PASSWORD': 'pass',
           'HOST': '127.0.0.1',
           'PORT': '5435'
       }
   }
   ```

4. Setup social authentication

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

5. Save your changes and close the `settings.py` file.

7. Create Database Tables

Execute the following command to initialize the database tables:

```
(tethys) $ tethys manage syncdb
```

8. Create a Superuser

Create a superuser/website administrator for your Tethys Portal:

```
(tethys) $ tethys manage createsuperuser
```

9. Start up the Django Development Server

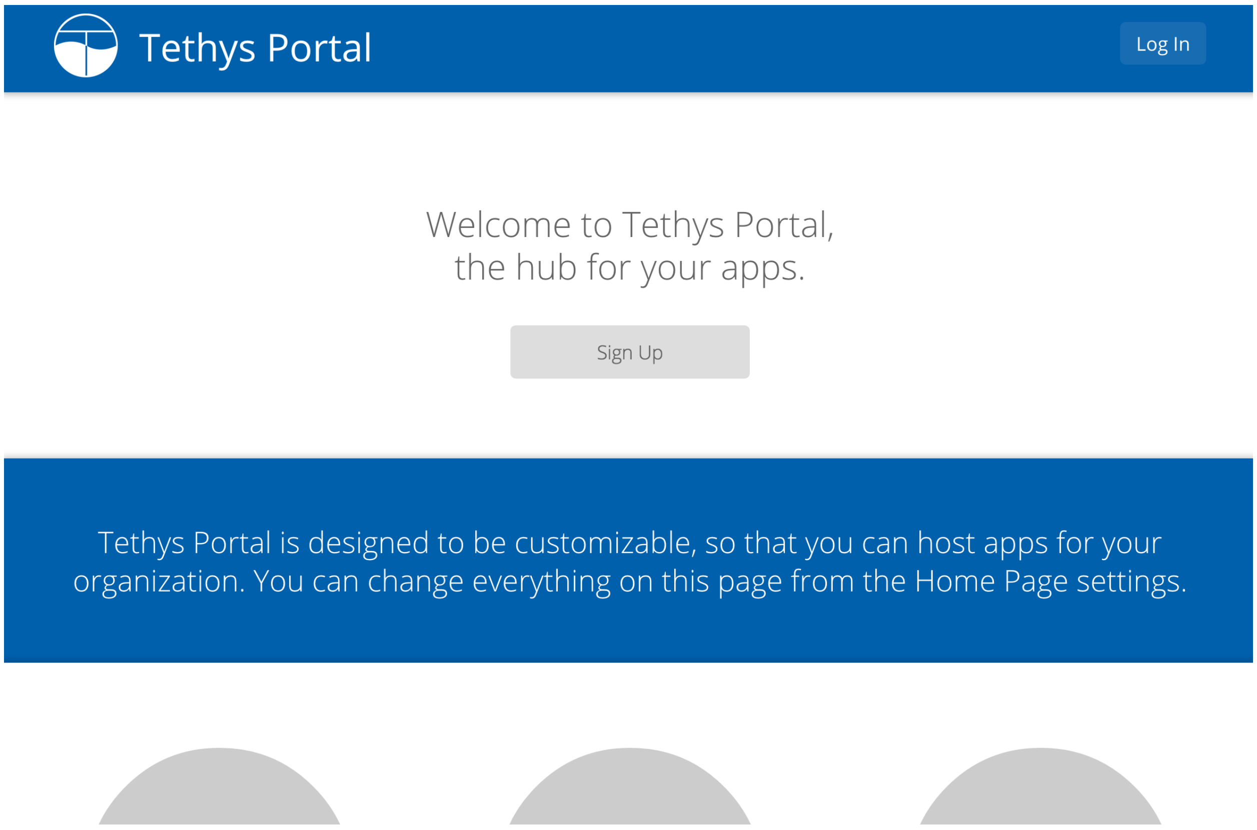
You are now ready to start the development server and view your instance of Tethys Platform. The website that ships with Tethys Platform is called *Tethys Portal*. In the terminal, execute the following command to start the development server:

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

Open http://localhost:8000/ in a new tab in your web browser and you should see the default *Tethys Portal* landing page.

9. Web Admin Setup

You are now ready to configure your Tethys Platform installation using the web admin interface. Follow the *Web Admin Setup* instructions to finish setting up your Tethys Platform.



Welcome to Tethys Portal,
the hub for your apps.

Tethys Portal is designed to be customizable, so that you can host apps for your organization. You can change everything on this page from the Home Page settings.

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.

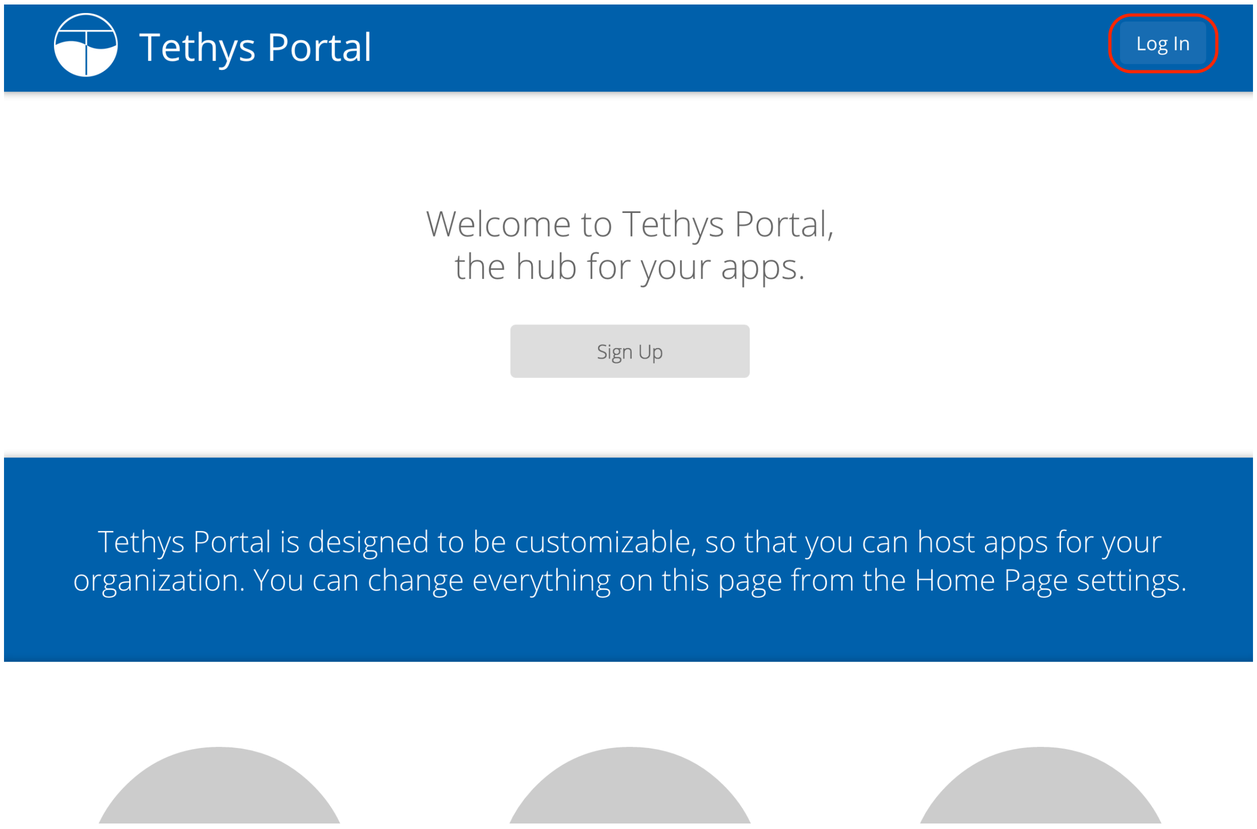
1. Access Tethys Portal Admin Console

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.

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

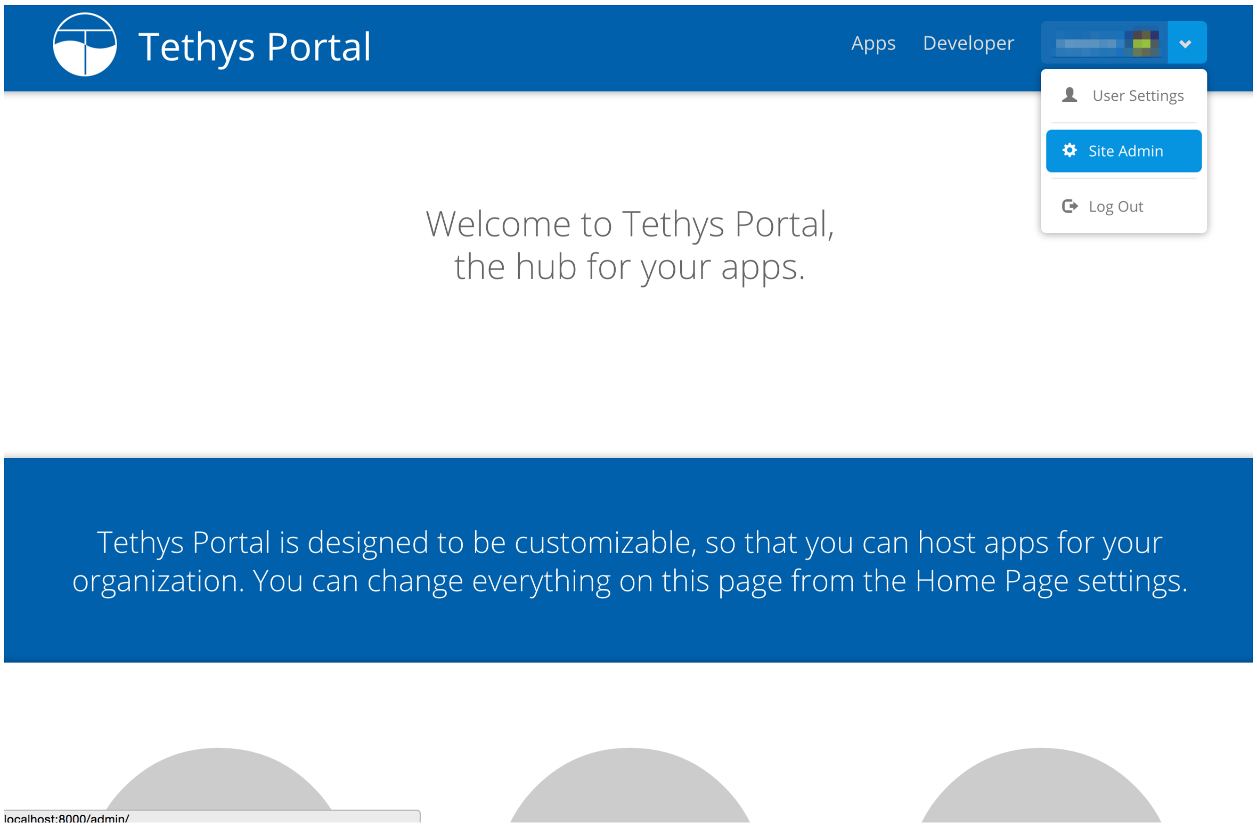
2. Select "Site Admin" from the user drop down menu.

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.



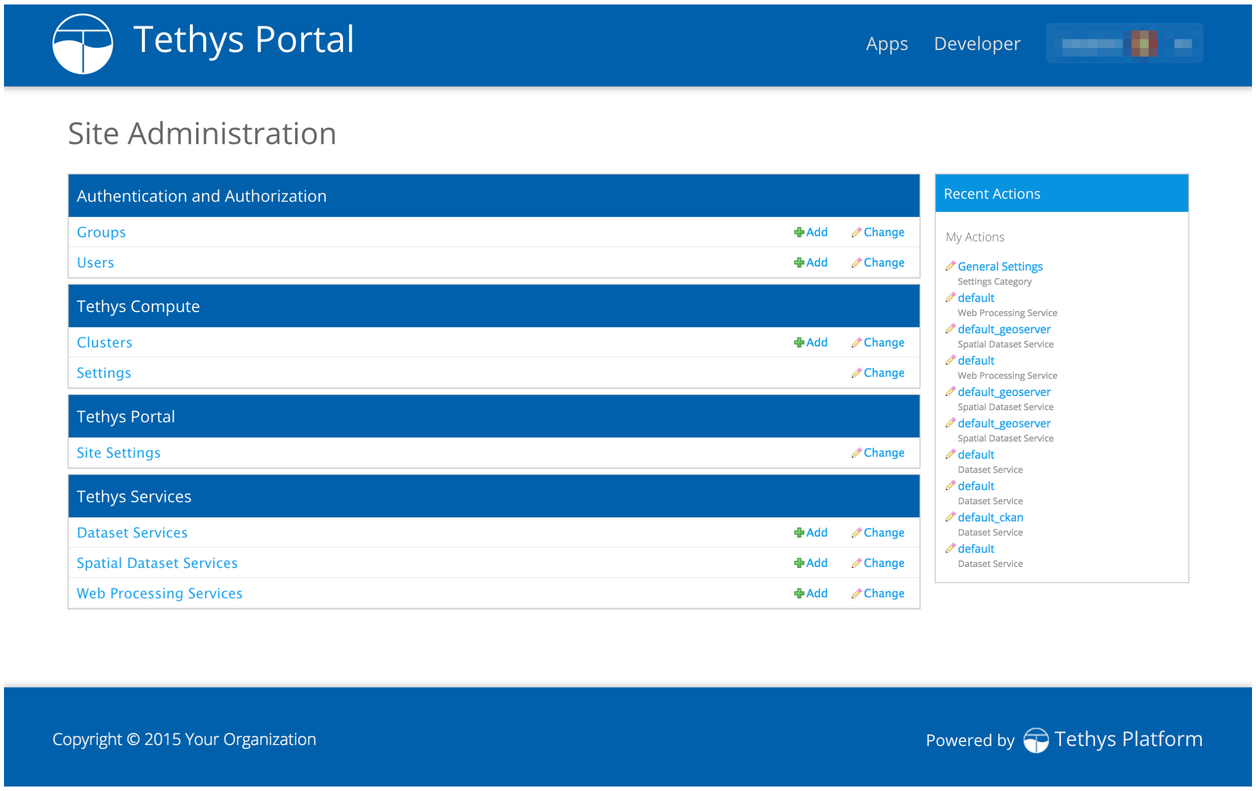
Welcome to Tethys Portal, the hub for your apps.

Tethys Portal is designed to be customizable, so that you can host apps for your organization. You can change everything on this page from the Home Page settings.



Welcome to Tethys Portal, the hub for your apps.

Tethys Portal is designed to be customizable, so that you can host apps for your organization. You can change everything on this page from the Home Page settings.


Tethys Portal

Apps Developer

Site Administration

Authentication and Authorization		
Groups	Add	Change
Users	Add	Change

Tethys Compute		
Clusters	Add	Change
Settings		Change

Tethys Portal		
Site Settings		Change

Tethys Services		
Dataset Services	Add	Change
Spatial Dataset Services	Add	Change
Web Processing Services	Add	Change

Recent Actions

My Actions

General Settings
Settings Category
default
Web Processing Service
default_geoserver
Spatial Dataset Service
default
Web Processing Service
default_geoserver
Spatial Dataset Service
default_geoserver
Spatial Dataset Service
default
Dataset Service
default
Dataset Service
default_ckan
Dataset Service
default
Dataset Service

Copyright © 2015 Your Organization

Powered by Tethys Platform

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:

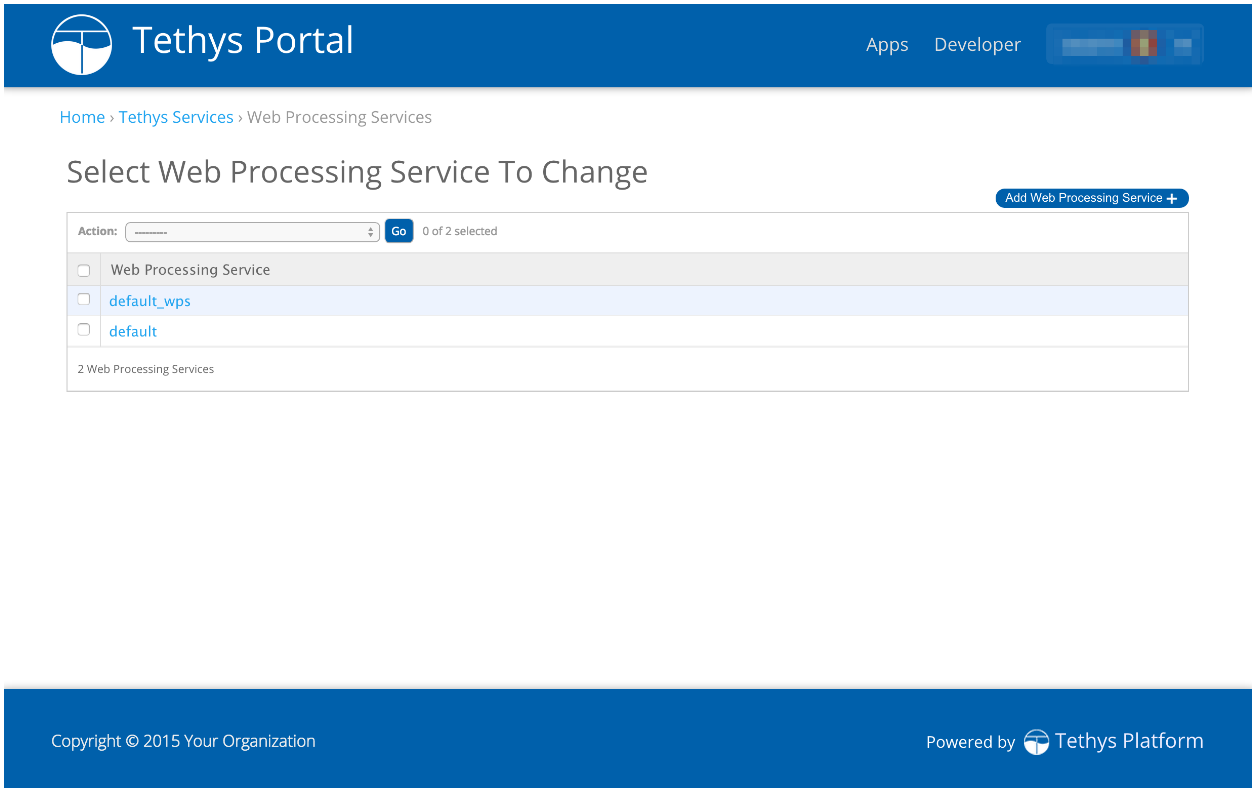
1. Select "Web Processing Services" from the options listed on the Tethys Portal Admin Console.
2. Click on the "Add Web Processing Service" button to create a new link to the web processing service.
3. Provide a unique name for the web processing service.
4. Provide an endpoint to the 52 North WPS that 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:
5. Press "Save" to save the WPS configuration.



Home › Tethys Services › Web Processing Services

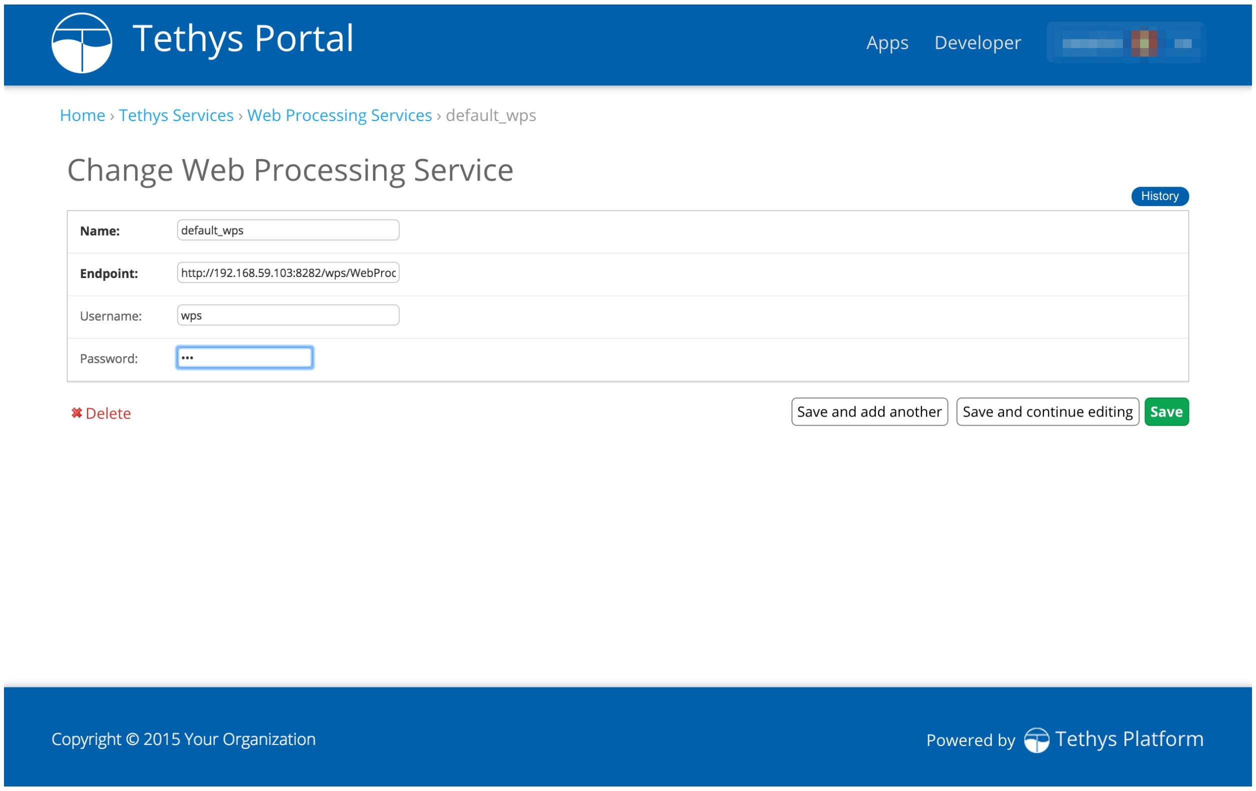
Select Web Processing Service To Change

0 of 2 selected

Web Processing Service
default_wps
default

2 Web Processing Services

Copyright © 2015 Your Organization



Home › Tethys Services › Web Processing Services › default_wps

Change Web Processing Service

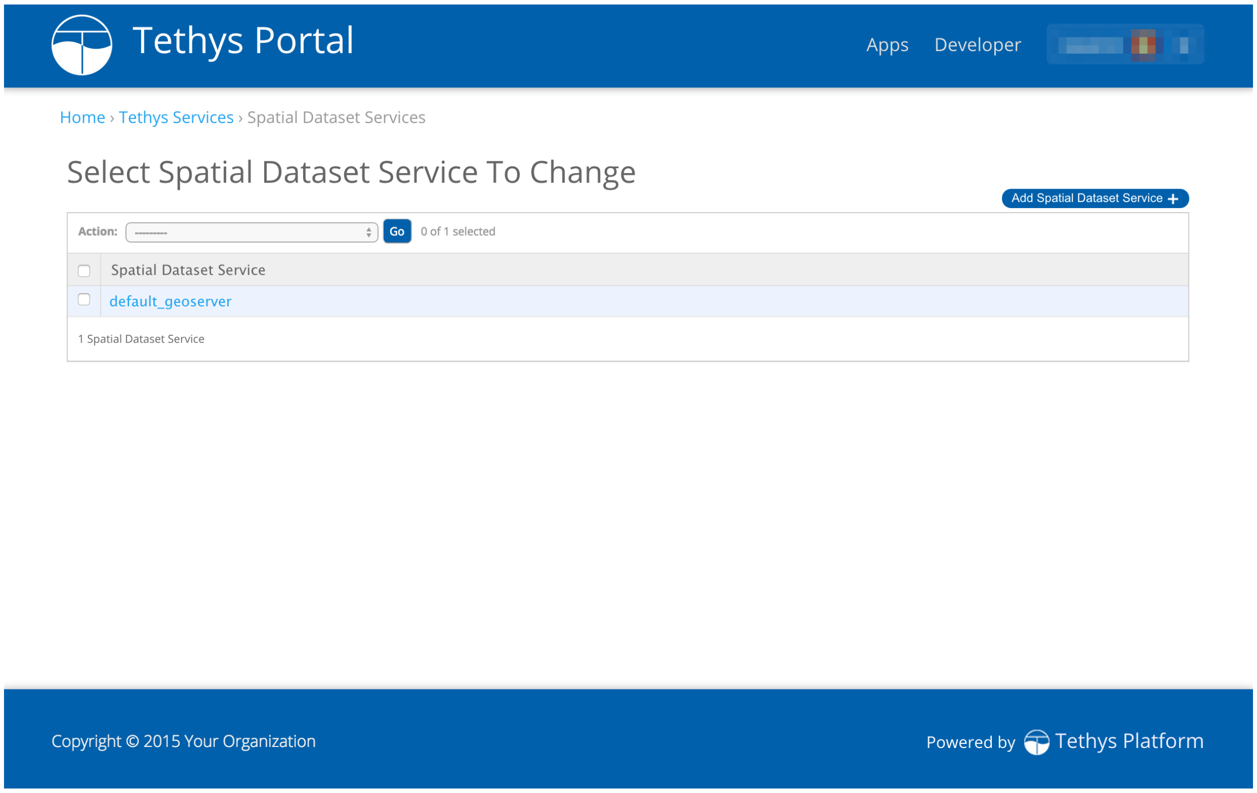
Name:	default_wps
Endpoint:	http://192.168.59.103:8282/wps/WebProc
Username:	wps
Password:	•••

Copyright © 2015 Your Organization

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:

1. Select "Spatial Dataset Services" from the options listed on the Tethys Portal Admin Console.
2. Click on the "Add Spatial Dataset Service" button to create a new spatial dataset service.


Tethys Portal
Apps
Developer
Home › Tethys Services › Spatial Dataset Services
Add Spatial Dataset Service +
Go
Copyright © 2015 Your Organization
Powered by Tethys Platform

Select Spatial Dataset Service To Change

Action: --------- Go 0 of 1 selected

	Spatial Dataset Service
☐	default_geoserver

1 Spatial Dataset Service

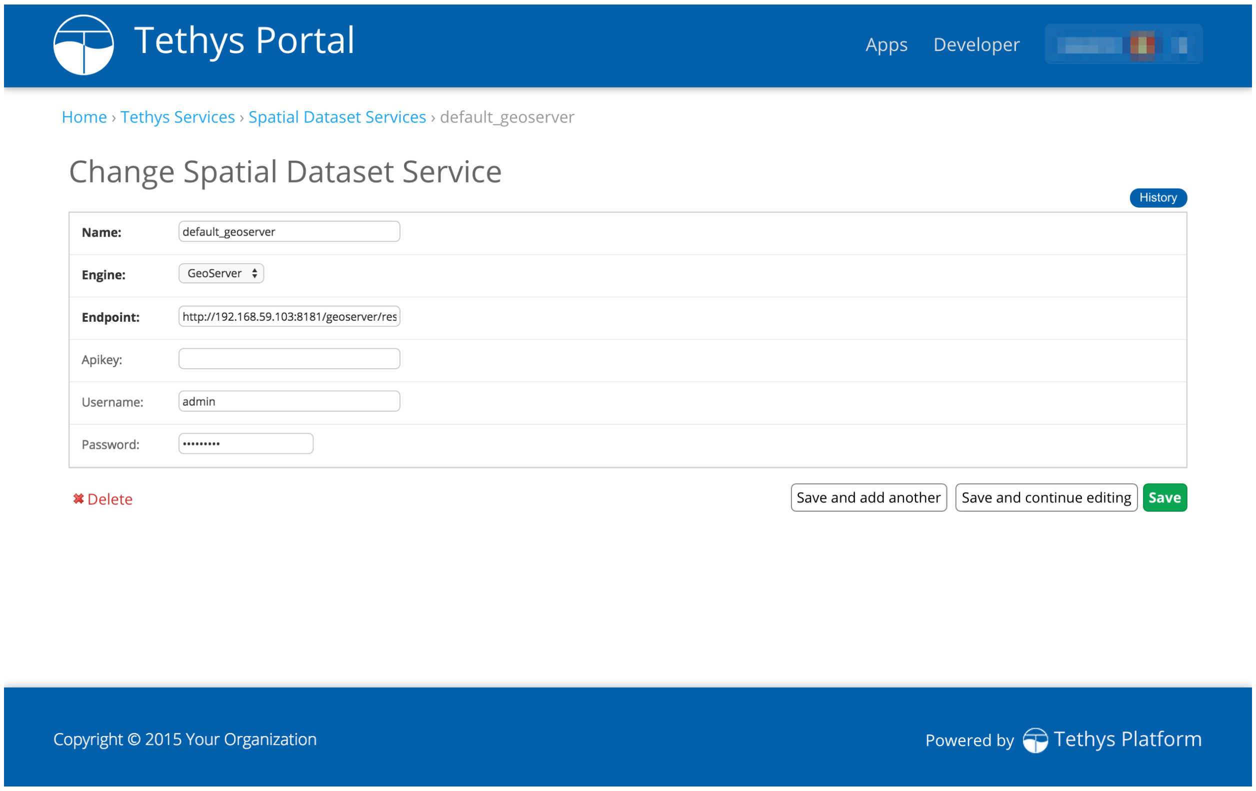
3. Provide a unique name for the spatial dataset service.
4. Select *"GeoServer"* as the engine and provide an endpoint to the Spatial Dataset Service. The endpoint is a URL pointing to the API of the Spatial Dataset Service. For GeoServers, this endpoint is of the form:

   ```
   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
   ...
   ```

5. 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:


Tethys Portal
Apps
Developer
Home › Tethys Services › Spatial Dataset Services › default_geoserver
History
Delete
Save and add another
Save and continue editing
Save
Copyright © 2015 Your Organization
Powered by Tethys Platform

Change Spatial Dataset Service

Name:	default_geoserver
Engine:	GeoServer
Endpoint:	http://192.168.59.103:8181/geoserver/res
Apikey:	
Username:	admin
Password:	•••••••••

6. 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:

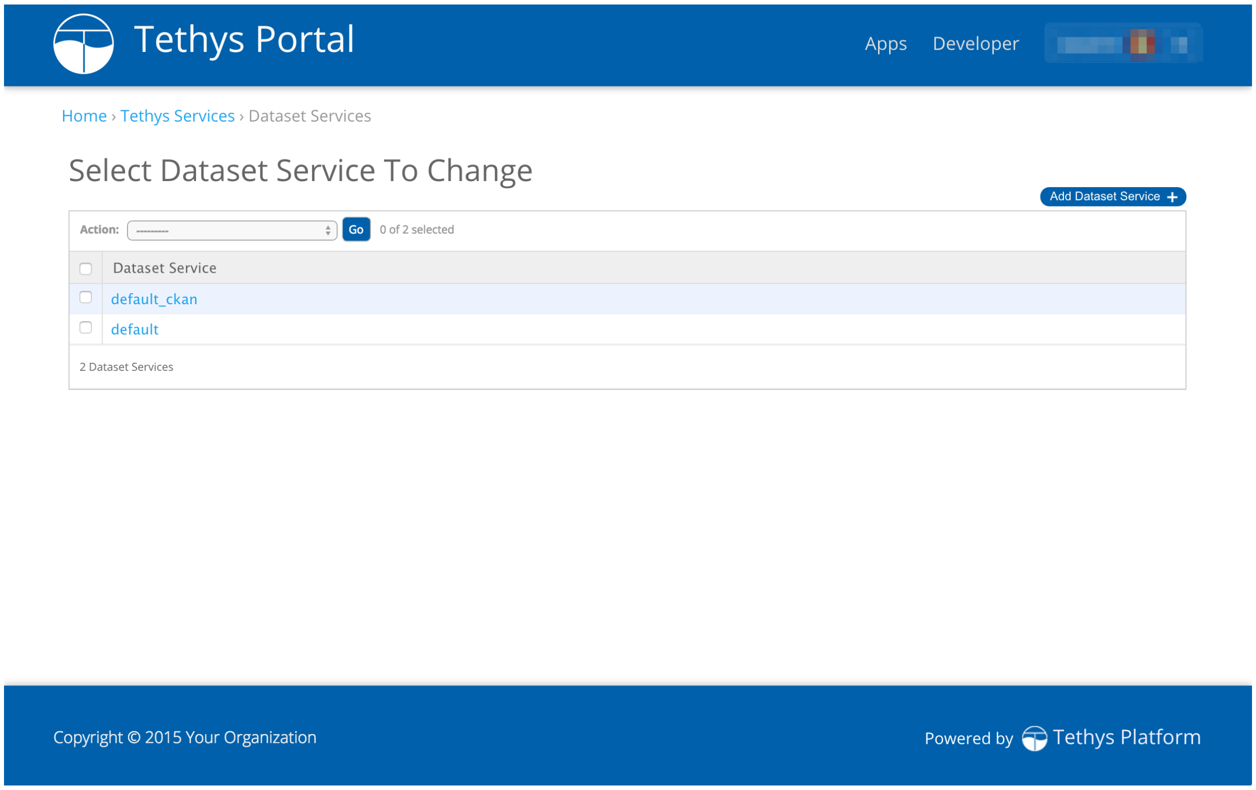
1. Select "Dataset Services" from the options listed on the Tethys Portal Admin Console.
2. Click on the "Add Dataset Service" button to create a new link to the dataset service.
3. Provide a unique name for the dataset service.
4. Select the appropriate engine and provide an endpoint to the Dataset 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.

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



Home › Tethys Services › Dataset Services

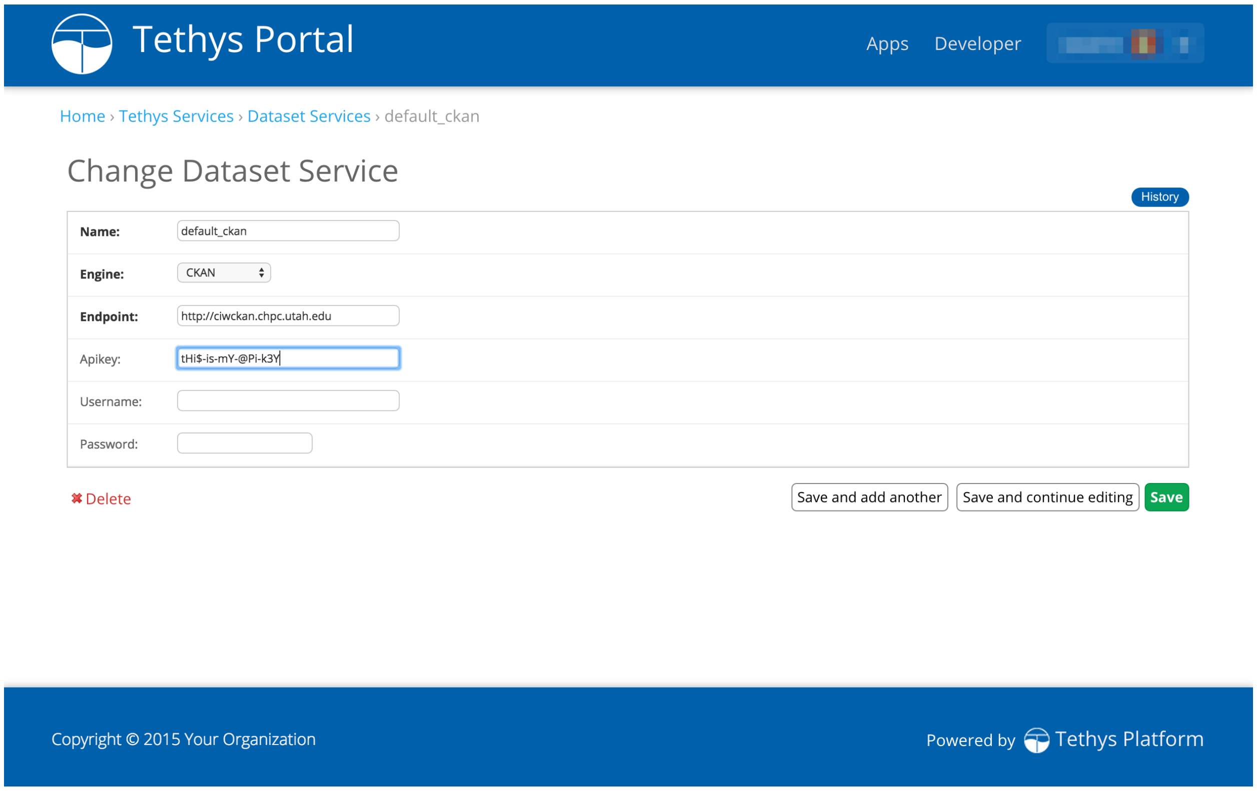
Select Dataset Service To Change

Action: --------- Go 0 of 2 selected

	Dataset Service
☐	default_ckan
☐	default

2 Dataset Services

Tethys Portal
Apps
Developer
Add Dataset Service +
Copyright © 2015 Your Organization
Powered by Tethys Platform



Home › Tethys Services › Dataset Services › default_ckan

Change Dataset Service

Name:	default_ckan
Engine:	CKAN
Endpoint:	http://ciwckan.chpc.utah.edu
Apikey:	tHi$-is-mY-@Pi-k3Y
Username:	
Password:	

Tethys Portal
Apps
Developer
History
Delete
Save and add another
Save and continue editing
Save
Copyright © 2015 Your Organization
Powered by Tethys Platform

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.

Upgrade from 1.3 to 1.4

Last Updated: December 1, 2016

1. Get the Latest Version

When you installed Tethys Platform you did so using it's remote Git repository on GitHub. To get the latest version of Tethys Platform, you will need to pull the latest changes from this repository:

```
$ cd /usr/lib/tethys/src
$ git pull origin master
```

2. Install Requirements and Run Setup Script

Install new dependencies and upgrade old ones:

```
         $ . /usr/lib/tethys/bin/activate
(tethys) $ pip install --upgrade -r /usr/lib/tethys/src/requirements.txt
(tethys) $ python /usr/lib/tethys/src/setup.py develop
```

3. Generate New Settings Script

Backup your old settings script (`settings.py`) and generate a new settings file to get the latest version of the settings. Then copy any settings (like database usernames and passwords) from the backed up settings script to the new settings script.

```
(tethys) $ mv /usr/lib/tethys/src/tethys_apps/settings.py /usr/lib/tethys/src/tethys_apps/settings.py
(tethys) $ tethys gen settings -d /usr/lib/tethys/src/tethys_apps
```

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

After you have copied these settings, you can delete or archive the backup settings script.

4. Sync the Database

Start the database docker if not already started and apply any changes to the database that may have been issued with the new release:

```
(tethys) $ tethys docker start -c postgis
(tethys) $ tethys manage syncdb
```

Note: For migration errors use:

```
$ cd ~/usr/lib/tethys/src
$ python manage.py makemigrations --merge
$ tethys manage syncdb
```

Tutorials

Last Updated: August 6, 2015

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

Getting Started

Last Updated: September 29, 2016

The getting started 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.

Create a New Tethys App Project

Last Updated: September 29, 2016

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*). In this tutorial you will start a new Tethys app project using the scaffold and install it into your Tethys Platform ready for development.

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.

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:

```
         $ . /usr/lib/tethys/bin/activate
(tethys) $ mkdir ~/tethysdev
(tethys) $ cd ~/tethysdev
(tethys) $ tethys scaffold my_first_app
```

The final command from the code block above is provided by the Tethys *Command Line Interface*. It will prompt you to enter metadata about your app such as, proper name, version, author, and description. All of these metadata are optional and you can accept the default value by pressing enter.

The commands you entered did the following tasks:

1. activated the Tethys *Python virtual environment*,
2. created a new directory called "tethysdev" in your home directory,
3. changed your working directory into the `tethysdev` directory, and
4. executed the **tethys scaffold** command to create the new app.

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-my_first_app`. All of the source code for your app is located in this directory. Open the `tethysapp-my_first_app` and explore the contents. The main directory of your app project, `my_first_app`, is located within a namespace directory called `tethysapp`. Each part of the app project will be explained throughout these tutorials. For more information about the app project structure, see *App Project Structure*.

Development Installation

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

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

View Your New App

Use start up the database Docker (postgis) and the development server:

```
(tethys) $ tethys docker start -c postgis
(tethys) $ tethys manage start
```

Browse to http://127.0.0.1:8000/apps. 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).

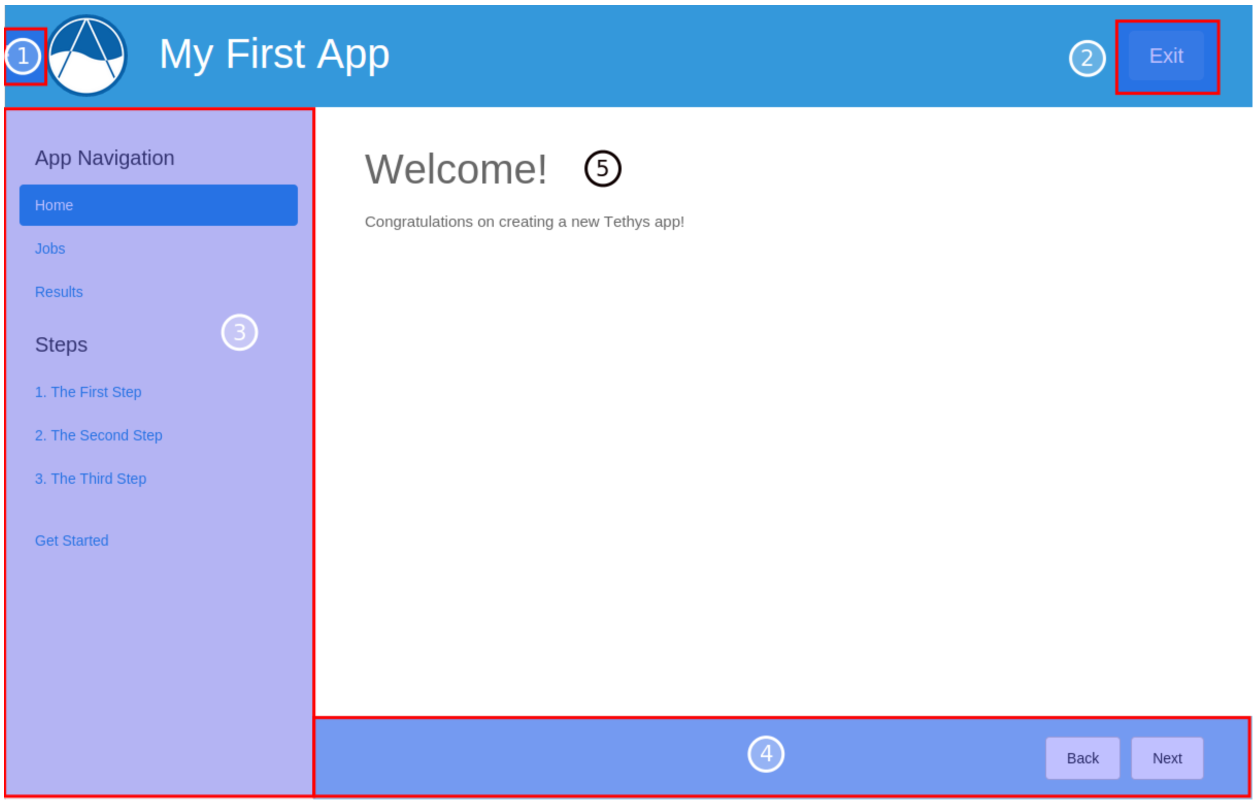
Parts of a Tethys app interface: (1) app navigation toggle, (2) exit button, (3) app navigation, (4) actions, and (5) app content.

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

```
(tethys) $ tethys docker stop
```

Model View Controller

Tethys apps are developed using the *Model View Controller* (MVC) software architecture pattern. Following the MVC pattern will make your app project easier to develop and maintain in the future. Most of the code in your app will fall into one of the three MVC categories. The Model represents the data of your app, the View is composed of the representation of the data, and the Controller consists of the logic to prepare the data for the view and any other logic your app needs. In the next few tutorials, you will be introduced to how the MVC development paradigm is used to develop Tethys apps. For more information about MVC, see *Key Concepts*.



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: "my_first_app". If you generated your scaffold exactly as above, this directory will be located at the following path:

```
# Path to App Package Directory
~/tethysdev/tethysapp-my_first_app/tethysapp/my_first_app/
```

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

```
# Relative App Package Directory Notation
my_first_app/controllers.py
```

Tip: As you explore the contents of your app project, you will notice that many of the directories have filed 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 could 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.

The Model and Persistent Stores

Last Updated: September 29, 2016

In this part of the tutorial you'll learn about the Model component of MVC development for Tethys apps. The Model represents the data of your app and the code used to manage it. The data of your app can take many forms. It can be generated on-the-fly and stored in Python data structures (e.g.: lists, dictionaries, and NumPy arrays), stored in databases, or contained in files via the *Dataset Services API*.

In this tutorial you will define your model using the *Persistent Stores API* to create a spatially enabled database for your app and you will learn how to use the SQLAlchemy object relational mapper (ORM) to create a data model for your app.

Register a Persistent Store

The Tethys Portal provides the *Persistent Stores API* to streamline the use of SQL databases in apps. To register a new *persistent store* database add the `persistent_stores()` method to your *app class*, which is located in your *app configuration file*. This method must return a list or tuple of `PersistentStore` objects.

Open the app configuration file for your app located at `my_first_app/app.py` in your favorite text editor. Import the `PersistentStore` object at the top of the file, add the `persistent_stores()` method to your app class, and save the changes:

```python
from tethys_sdk.base import TethysAppBase, url_map_maker
from tethys_sdk.stores import PersistentStore


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 = '#e74c3c'
    description = ''
    tags = ''
    enable_feedback = False
    feedback_emails = []


    def url_maps(self):
        """
        Add controllers
        """
        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

    def persistent_stores(self):
        """
        Add one or more persistent stores
        """
        stores = (PersistentStore(name='stream_gage_db',
                                  initializer='my_first_app.init_stores.init_stream_gage_db',
                                  spatial=True
```

```
                ),
        )

        return stores
```

A persistent store database will be created for each `PersistentStore` object that is returned by the `persistent_stores()` method of your *app class*. In this case, your app will have a persistent store named "stream_gage_db". The `initializer` argument points to a function that you will define in a later step. The `spatial` argument can be used to add spatial capabilities to your persistent store. Tethys Platform provides PostgreSQL databases for persistent stores and PostGIS for the spatial database capabilities.

Note: Read more about persistent stores in the *Persistent Stores API* documentation.

Create an SQLAlchemy Data Model

After your database is created, you will need to create the tables that will store the data for your app. The plan for your database tables or schema is called a data model. SQLAlchemy provides an Object Relational Mapper (ORM) that allows you to create data models using Python code and issue queries using an object-oriented approach. In other words, you are able to harness the power of SQL databases without writing SQL. As a primer to SQLAlchemy ORM, we highly recommend you complete the Object Relational Tutorial.

In this step, you will use SQLAlchemy to create a data model for the tables that will store the data for your app. Open the `model.py` file located at `my_first_app/model.py` in a text editor.

First, add the following import statements to your `model.py` file:

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

from .app import MyFirstApp
```

Next, add these lines to your `model.py` file:

```
# DB Engine, sessionmaker and base
engine = MyFirstApp.get_persistent_store_engine('stream_gage_db')
SessionMaker = sessionmaker(bind=engine)
Base = declarative_base()
```

The `get_persistent_store_engine()` method that is used here accepts the name of a persistent store as an argument and returns an SQLAlchemy engine object. The engine object contains the connection information needed to connect to the persistent store database. Anytime you want to query or modify your persistent store data, you will do so with an SQLAlchemy `session` object. As the name implies, the `SessionMaker` can be used to create new `session` objects. The `Base` object is used in the next step when we define our data model. Add these lines to your `model.py` file:

```
# 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
```

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

The class defines four other properties that are SQLAlchemy `Column` objects: *id*, *latitude*, *longitude*, and *value*. These properties define the columns of the "stream_gages" table. 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. The `StreamGage` class also has a simple constructor method called `__init__()`.

This class is not only used to define the tables for your persistent store, it will also be used to create objects for interacting with your data.

Be sure to save the changes to `model.py` and close before proceeding.

Create an Initialization Function

Now that you have created a data model, the next step is to write a database initialization function. This function will be called during the initialization phase of your persistent store database and will be used to create the tables in your database and add any initial data that you may need in the database for your app to work.

Open the `my_first_app/init_stores.py` in a text editor. At the top of this file, import the `engine`, `SessionMaker`, `Base`, and `StreamGage` from your data model:

```
from .model import engine, SessionMaker, Base, StreamGage
```

Next, create a new function called `init_stream_gage_db()` with a single argument called `first_time` and the following code:

```
def init_stream_gage_db(first_time):
    """
    An example persistent store initializer function
    """
    # Create tables
    Base.metadata.create_all(engine)

    # Initial data
    if first_time:
        # Make session
        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)

        # Gage 3
        gage3 = StreamGage(latitude=40.23650788415366,
                           longitude=-111.73278093338013,
                           value=3)

        session.add(gage3)

        # Gage 4
        gage4 = StreamGage(latitude=40.242519244799816,
                           longitude=-111.68254852294922,
                           value=4)

        session.add(gage4)

        session.commit()
```

The `Base.metedata.create_all(engine)` line is all that is needed to create the tables in your persistent store database. Every class that inherits from the `Base` class is tracked by a `metadata` object. The `metadata.create_all()` method issues the SQL that is needed to create the tables associated with the `Base` class. Notice that you must give it the `engine` object for connection information.

The `first_time` parameter that is passed to all persistent store initialization functions 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. Notice the code that adds initial data to your persistent store database is wrapped in a conditional statement that uses the `first_time` parameter.

This initial data code adds four stream gages to your persistent store database. 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. To persist the new records to the persistent store database, the `session.commit()` method is called. You will learn how to query the persistent store database using SQLAlchemy in the *The Controller* tutorial.

Save your changes to `init_stores.py` and close before moving on.

Register Initialization Function

Recall that when you registered the persistent store in your app configuration file, you specified the `initializer` function for the persistent store. This argument accepts a string representing the path to the function using dot notation and a colon to delineate the function (e.g.: “app_name.module.function”). Check your *app configuration file* (`app.py`) to ensure the path to the initializer function is correct: `'my_first_app.init_stores.init_stream_gage_db'`.

Persistent Store Initialization

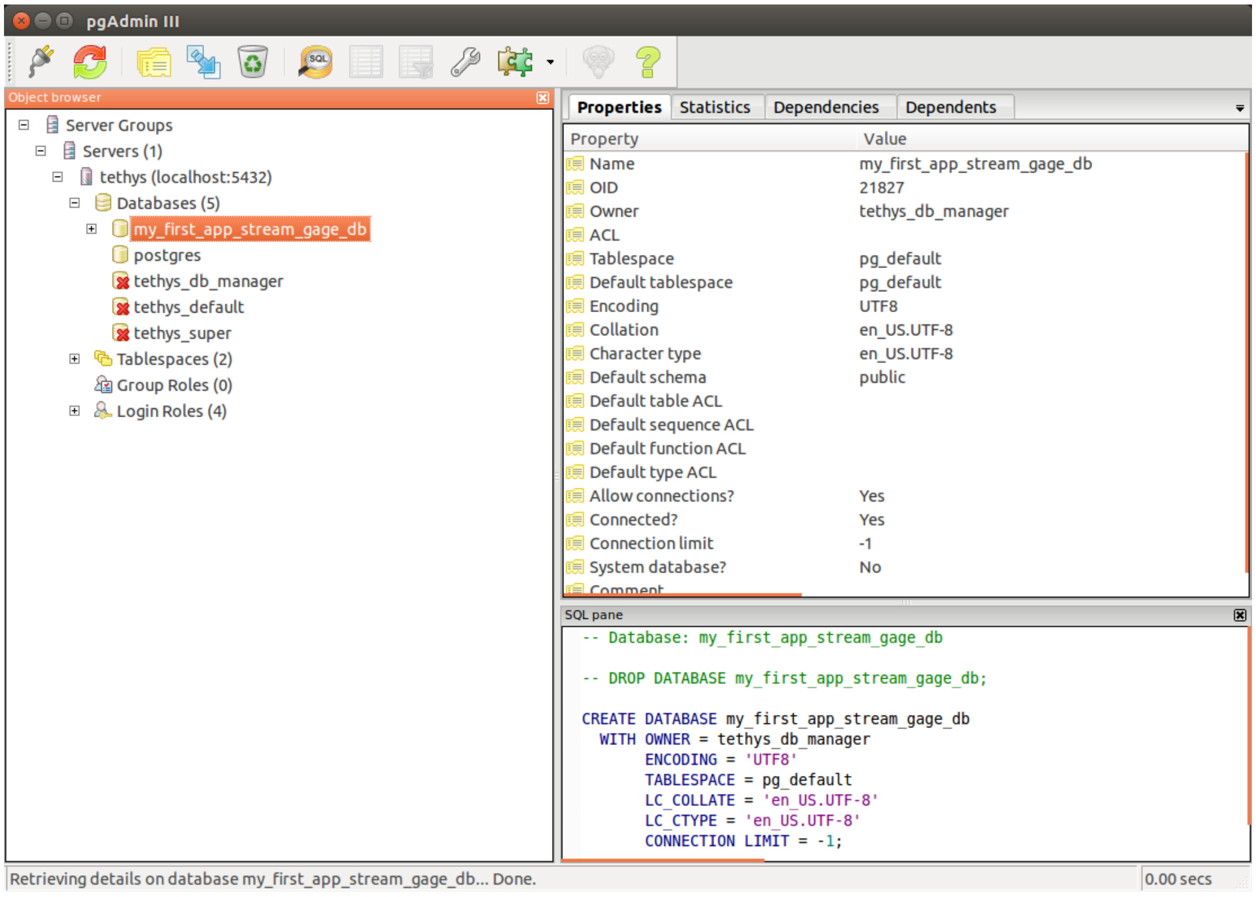
The Tethys command line utility provides a command for initializing persistent stores. Save all changes to the files you edited and stop your development server using `CTRL-C` if it is still running. It is possible that your server may have crashed during editing and is displaying errors; ignore these errors. Execute the following command in the terminal:

```
(tethys) $ tethys syncstores my_first_app
```

The database will be initialized and you will see text printed to the terminal that will indicate this:

```
Loading Tethys Apps...
Tethys Apps Loaded: my_first_app

Provisioning Persistent Stores...
Creating database "stream_gage_db" for app "my_first_app"...
Enabling PostGIS on database "stream_gage_db" for app "my_first_app"...
Initializing database "stream_gage_db" for app "my_first_app" using initializer "init_stream_gage_db
```

If you have a graphical database client like PGAdmin III, you may wish to connect to your PostgreSQL database server and confirm that the database was created. You can use the credentials for `tethys_super` database user that you defined during installation to connect to the database. The name of the database will be a combination of the name of your app and the name of the persistent store: (e.g.: my_first_app_stream_gage_db). For a more detailed explanation of connecting to your database using PGAdmin III, see the *PGAdmin III Tutorial*.



```
Server Groups
  Servers (1)
    tethys (localhost:5432)
      Databases (5)
        my_first_app_stream_gage_db
        postgres
        tethys_db_manager
        tethys_default
        tethys_super
      Tablespaces (2)
      Group Roles (0)
      Login Roles (4)
```

Property	Value
Name	my_first_app_stream_gage_db
OID	21827
Owner	tethys_db_manager
ACL	
Tablespace	pg_default
Default tablespace	pg_default
Encoding	UTF8
Collation	en_US.UTF-8
Character type	en_US.UTF-8
Default schema	public
Default table ACL	
Default sequence ACL	
Default function ACL	
Default type ACL	
Allow connections?	Yes
Connected?	Yes
Connection limit	-1
System database?	No
Comment	

```
-- Database: my_first_app_stream_gage_db

-- DROP DATABASE my_first_app_stream_gage_db;

CREATE DATABASE my_first_app_stream_gage_db
  WITH OWNER = tethys_db_manager
       ENCODING = 'UTF8'
       TABLESPACE = pg_default
       LC_COLLATE = 'en_US.UTF-8'
       LC_CTYPE = 'en_US.UTF-8'
       CONNECTION LIMIT = -1;
```

Retrieving details on database my_first_app_stream_gage_db... Done. 0.00 secs

Example of graphical database client PGAdmin III.

The View and Templating

Last Updated: September 29, 2016

In this section the View aspect of MVC will be introduced. The View consists of the representation or visualizations of your app's data and the user interface. 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, similar to how PHP is used. The result is dynamic, reusable templates for the web pages of your app.

In this tutorial you will add a view to your app for displaying the stream gages that are in your database on a Google Map.

Templating

The Django template language is a simple, but powerful templating language. This section will provide a crash course in Django template language basics, but we highly recommend a review of the Django Template Language documentation.

Browse to the your templates directory located at `my_first_app/templates/`. By convention, all the templates for your app are stored in a directory with the same name of your *app package* inside the templates directory (e.g.: `templates/my_first_app`). This will prevent potential conflicts with the templates of other apps. You will find two templates in this directory: `base.html` and `home.html`. Refer to these templates as the Django template concepts are introduced.

Variables, Filters, and Tags Django templates can contain variables, filters, and tags. 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: `{{ variable.attribute }}`.

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 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`.

Tip: For a better explanation of variables, filters and tags, see the *App Templating API*.

Template Inheritance One of the advantages of using the Django template language is that it provides a way for child templates to extend parent templates, which reduces 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 and specifying the template they which to inherit from. 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.

Tip: If you are unfamiliar with Django template inheritance, please review the Django Template Inheritance documentation before proceeding.

Base Template Tethys apps generated from the scaffold come with a `base.html` template which has the following contents:

```
{% extends "tethys_apps/app_base.html" %}

{% load staticfiles %}
{% block title %}- {{ tethys_app.name }}{% endblock %}

{% block styles %}
  {{ block.super }}
  <link href="{% static 'my_first_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
  <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 'my_first_app/js/main.js' %}" type="text/javascript"></script>
{% endblock %}
```

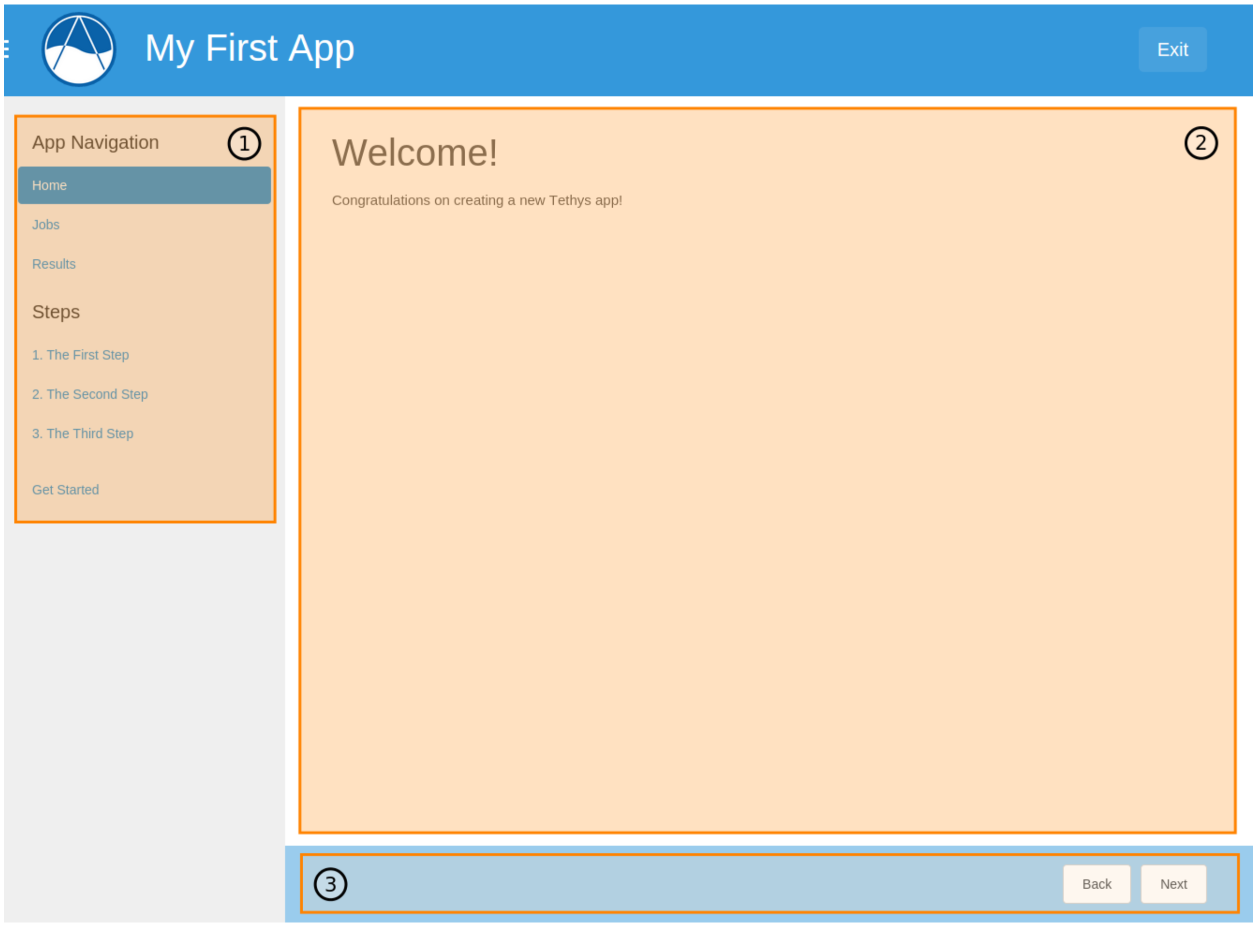
The `base.html` template is intended to be used as the parent template for all your app templates via the `extends` tag. It contains several `block` tags that your app templates can override or extend. The `block` tags you will use most often are `app_navigation_items`, `app_content`, and, `app_actions`. These blocks correspond with different parts of the app interface (shown in the figure below). As a rule, content that you would like to be present in all your templates should be included in the `base.html` template and content that is specific to a certain template should be included in that template.

The `block` tags of the `base.html` template correspond with different parts of the interface: (1) `app_navigation_items`, (2) `app_content`, and (3) `app_actions`.

Tip: For an explanation of the blocks in the `base.html` template see the *App Templating API*.

Public Files and Resources

Most apps will use files and resources that are static–meaning they do not need to be preprocessed before being served like templates do. Examples of these files include images, CSS files, and JavaScript files. Tethys Platform will automatically register static files that are located in the `public` directory of your app project. Use the `static` tag in


My First App
Exit
App Navigation
1
Home
Jobs
Results
Steps
1. The First Step
2. The Second Step
3. The Third Step
Get Started
Welcome!
2
Congratulations on creating a new Tethys app!
3
Back
Next

templates to load the resource URLs. The `base.html` template provides examples of how to use the `static` tag. See the Django documentation for the static tag for more details.

Caution: Any file stored in the public directory will be accesible to anyone. Be careful not to expose sensitive information.

Make a New Template

Now that you know the basics of templating, you will learn how to create new templates that extend the base template and use the `block` tags. Create a new template in your templates directory (`my_first_app/templates/my_first_app/`) and name it `map.html`. Open this file in a text editor and copy and paste the following code into it:

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

{% load tethys_gizmos %}

{% block app_content %}
  <h1>Stream Gages</h1>
  {% gizmo map_view map_options %}
{% endblock %}

{% block app_actions %}
  <a href="{% url 'my_first_app:home' %}" class="btn btn-default">Back</a>
{% endblock %}
```

The `map.html` template that you created extends the `base.html` template. It also overrides the `app_content`, *app_actions*‘, and `scripts` blocks. An action called “Back” is added to the `app_actions` block. It uses a new tag, the `url` tag, to provide a link back to the home page of the app. The `url` tag will be discussed in more detail in the *URL Mapping* tutorial.

The map is inserted into the `app_content` block using one of the Tethys Gizmos called `map_view`. Gizmos are an easy way to insert common user interface elements in to your templates with minimal code. The map is configured via a dictionary called `map_options`, which is defined in the controller. This will be discussed in the next tutorial. For more information on Gizmos, refer to the *Template Gizmos API* documentation.

The Controller

Last Updated: September 29, 2016

The Controller component of MVC will be discussed in this part of the tutorial. The job of the controller is to coordinate between the View and the Model. Often this means querying a database and transforming the data to a format that the view expects it to be in. The Controller also handles most of the application logic such as processing and validating form data or launching model runs. In a Tethys app, controllers are simple Python functions.

Django is used to implement Tethys controllers but they are called “views” in Django. The Writing Views documentation for Django is a good reference for Tethys controllers. Note that URL mapping is handled differently in Tethys app development than in Django development and will be discussed in the *URL Mapping* tutorial.

In this tutorial you will write a controller that will retrieve the data from your stream gage model and then pass it to the template that you created in the previous tutorial.

Make a New Controller

Recall that in *The Model and Persistent Stores* tutorial you created an SQLAlchemy data model to store information about stream gages. You also created an initialization function that loaded some dummy data into your database. You will now add some logic to your controller to retrieve this data and pass it to the template.

Open your `controllers.py` file located at `my_first_app/controllers.py`. This file should contain a function called `home`. This function is the controller for the home page of your app. All controller functions must accept a request object and they must return a response object. The request object contains information about the HTTP request, including any form data that is submitted (more on this later). There are several ways to return a response object, but the most common way is to use the `render()` function provided by Django. This function requires three arguments: the request object, the template to be rendered, and the context dictionary.

Add the following imports to the top of the file:

```python
from .model import SessionMaker, StreamGage
from tethys_sdk.gizmos import MapView, MVLayer, MVView
```

Then add a new controller function called `map` after the `home` function:

```python
@login_required()
def map(request):
    """
    Controller for map page.
    """
    # Create a session
    session = SessionMaker()

    # Query DB for gage objects
    gages = session.query(StreamGage).all()

    # Transform into GeoJSON format
    features = []

    for gage in gages:
        gage_feature = {
          'type': 'Feature',
          'geometry': {
            'type': 'Point',
            'coordinates': [gage.longitude, gage.latitude]
          }
        }

        features.append(gage_feature)

    geojson_gages = {
      'type': 'FeatureCollection',
      'crs': {
        'type': 'name',
        'properties': {
          'name': 'EPSG:4326'
        }
      },
      'features': features
    }

    # Define layer for Map View
    geojson_layer = MVLayer(source='GeoJSON',
                            options=geojson_gages,
                            legend_title='Provo Stream Gages',
                            legend_extent=[-111.74, 40.22, -111.67, 40.25])

    # Define initial view for Map View
    view_options = MVView(
        projection='EPSG:4326',
        center=[-111.70, 40.24],
        zoom=13,
        maxZoom=18,
        minZoom=2
    )

    # Configure the map
    map_options = MapView(height='500px',
                          width='100%',
                          layers=[geojson_layer],
                          view=view_options,
                          basemap='OpenStreetMap',
                          legend=True)

    # Pass variables to the template via the context dictionary
    context = {'map_options': map_options}

    return render(request, 'my_first_app/map.html', context)
```

The new `map` controller queries the persistent store for the stream gages, converts the data into GeoJSON format for the map, and configures the map options for the Map View Gizmo that is used in the template.

To query the database, an SQLAlchemy `session` object is needed. It is created using the `SessionMaker` object imported from the `model.py` file. Querying is accomplished by using the `query()` method on the `session` object. The result is a list of `StreamGage` objects representing the records in the database.

The map is capable of consuming spatial data in a few formats including GeoJSON, so the `map` controller handles the job of converting the data from the list of `StreamGage` objects to GeoJSON format.

The map Gizmo that is used in the `map.html` template requires a dictionary of configuration options called "map_options". This is created in the controller and the `input_overlays` option is used to give the GeoJSON formatted stream gage data to the map.

Next, a template context dictionary is defined that contains all of the variables that you wish to be available for use in the template.

Finally, the `render()` function is used to create the response object. It is in the `render()` function that you specify the template that is to be rendered by the controller. In this case, the `map.html` that you created in the last tutorial. Note that the path you provide to the template is relative to the template directory of your app: `my_first_app/map.html`.

Save `controllers.py` before going on.

URL Mapping

Last Updated: September 29, 2016

Whenever you create a new controller, you will also need to associate it with a URL by creating URL map for it. When a URL is requested, the controller that it is mapped to will be executed.

In this tutorial you will create a new URL map for the new `map` controller you created in the previous tutorial.

Map Controller to URL

Mapping a controller to a URL is performed in the *app configuration file* (`app.py`). Open your app configuration file located at `my_first_app/app.py`. Your *app class* will already have a method called `url_maps()`. This method must return a list or tuple of `UrlMap` objects. `UrlMap` objects require three attributes: `name`, `url`, and `controller`.

Your *app class* will already have one `UrlMap` for the home page called "home". Add a new `UrlMap` object for the `map` controller that you created in the previous step. Give it a name of "map", a url of "my-first-app/map", and a path to the controller of "my_first_app.controllers.map". The `url_maps()` method for your app should look something like this when you are done:

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

    url_maps = (UrlMap(name='home',
                       url='my-first-app',
                       controller='my_first_app.controllers.home'),
                UrlMap(name='map',
                       url='my-first-app/map',
                       controller='my_first_app.controllers.map'),
    )

    return url_maps
```

Important: All of the URL patterns for your app should begin with the base URL of your app (e.g.: 'my-first-app') to prevent conflicts with other apps.

Now that you have created the URL map for your new map page, you can create a link to it from the home page. Open the `home.html` template located at `my_first_app/templates/my_first_app/home.html`. Replace the `app_actions` template block with the following:

```
{% block app_actions %}
  <a href="{% url 'my_first_app:map' %}" class="btn btn-default">Go To Map</a>
{% endblock %}
```

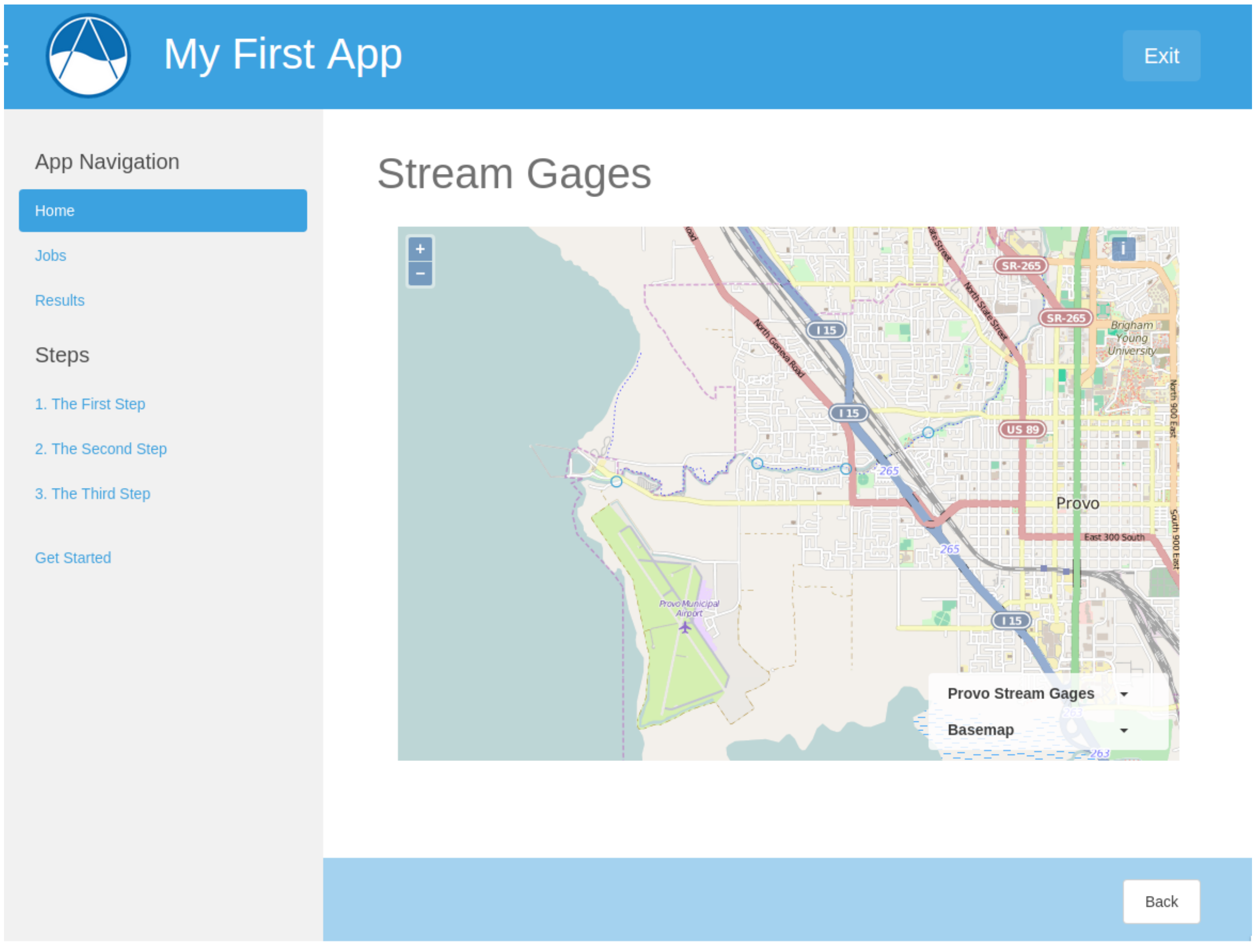
In this code, the `url` template tag is used to provide the url to the map page. It accepts a string with the following pattern: `"name_of_app:name_of_url_map"`. The advantage of using the `url` tag as opposed to hard coding the URL is that if the URL ever needs to be changed, you will only need to change it in your app configuration file and not in every template that references that URL.

View New Map Page

At this point, your app should be ready to run again. Save all changes in the files you edited and restart the development server using the `tethys manage start` command in the terminal (stop it using `CTRL-C` if necessary). Browse to your app home page at http://127.0.0.1:8000/apps/my-first-app. Use the "Go To Map" action to browse to your new map page. Click on the "Provo Stream Gages" layer in the legend to zoom to that layer. Your map page should look similar to this:

Advanced Concepts

Last Updated: September 29, 2016


My First App
Exit
App Navigation
Home
Jobs
Results
Steps
1. The First Step
2. The Second Step
3. The Third Step
Get Started
Stream Gages
Provo
Provo Stream Gages
Basemap
Back

The purpose of this tutorial will be to introduce some advanced concepts in Tethys app development. In the map page you created in the previous tutorials, you are able to view all of the stream gages on a map concurrently. In this tutorial you will add the ability to view individual stream gages on the map page. This will involve creating a new url map, new controller, and some modifications to the map template. This exercise will also serve as a good review of MVC development in Tethys Platform.

New URL Map and URL Variables

You can add variables to your URLs to make your controllers and web pages more dynamic. URL variables are denoted by single curly braces in the URL string like this: `/example/url/{variable}`. Open the `my_first_app/app.py` file in a text editor. Modify the `url_maps()` method by adding a new `UrlMap` object named "map_single" with a URL variable called "id". Your `url_maps()` method should look like this when you are done:

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

    url_maps = (UrlMap(name='home',
                       url='my-first-app',
                       controller='my_first_app.controllers.home'),
                UrlMap(name='map',
                       url='my-first-app/map',
                       controller='my_first_app.controllers.map'),
                UrlMap(name='map_single',
                       url='my-first-app/map/{id}',
                       controller='my_first_app.controllers.map_single'),
    )

    return url_maps
```

Note: The Django documentation on URL mapping will not be useful for Tethys apps. A different approach is used by Tethys that is easier to use than the Django method.

New Controller

Notice that the `map_single` `UrlMap` object points to a controller named "map_single". This controller doesn't exist yet, so we will need to create it. Open the `my_first_app/controllers.py` in a text editor and add the `map_single` controller function to it:

```
@login_required
def map_single(request, id):
    """
    Controller for map page.
    """
    # Create a session
    session = SessionMaker()

    # Query DB for gage objects
    gage = session.query(StreamGage).filter(StreamGage.id==id).one()

    # Transform into GeoJSON format
    gage_feature = {
      'type': 'Feature',
      'geometry': {
        'type': 'Point',
        'coordinates': [gage.longitude, gage.latitude]
      }
    }

    geojson_gages = {
      'type': 'FeatureCollection',
      'crs': {
        'type': 'name',
        'properties': {
          'name': 'EPSG:4326'
        }
      },
      'features': [gage_feature]
    }

    # Define layer for Map View
    geojson_layer = MVLayer(source='GeoJSON',
                            options=geojson_gages,
                            legend_title='Provo Stream Gages',
                            legend_extent=[-111.74, 40.22, -111.67, 40.25])

    # Define initial view for Map View
    view_options = MVView(
        projection='EPSG:4326',
        center=[-111.70, 40.24],
        zoom=13,
        maxZoom=18,
        minZoom=2
    )

    # Configure the map
    map_options = MapView(height='500px',
                          width='100%',
                          layers=[geojson_layer],
                          view=view_options,
                          basemap='OpenStreetMap',
                          legend=True)

    context = {'map_options': map_options,
               'gage_id': id}

    return render(request, 'my_first_app/map.html', context)
```

The `map_single` controller function is slightly different than the `map` controller you created earlier. It accepts an additional argument called “id”. The `id` URL variable value will be passed to the `map_single` controller making the `id` variable available for use in the controller logic.

Anytime you create a URL with variables in it, the variables need to be added to the arguments of the controller function it maps to.

The `map_single` controller is similar but different from the `map` controller you created earlier. The SQLAlchemy query searches for a single stream gage record using the `id` variable via the‘‘filter()‘‘ method. The stream gage data returned by the query is reformatted into GeoJSON format as before and the `map_options` for the Gizmo are defined.

The context is expanded to include the `id` variable, so that it will be available for use in the template. The same `map.html` template is being used by this controller as was used by the `map` controller. However, it will need to be modified slightly to make use of the new `gage_id` context variable.

Modify the Template

Open the `map.html` template located at `my_first_app/templates/my_first_app/map.html`. Modify the template so that it matches this:

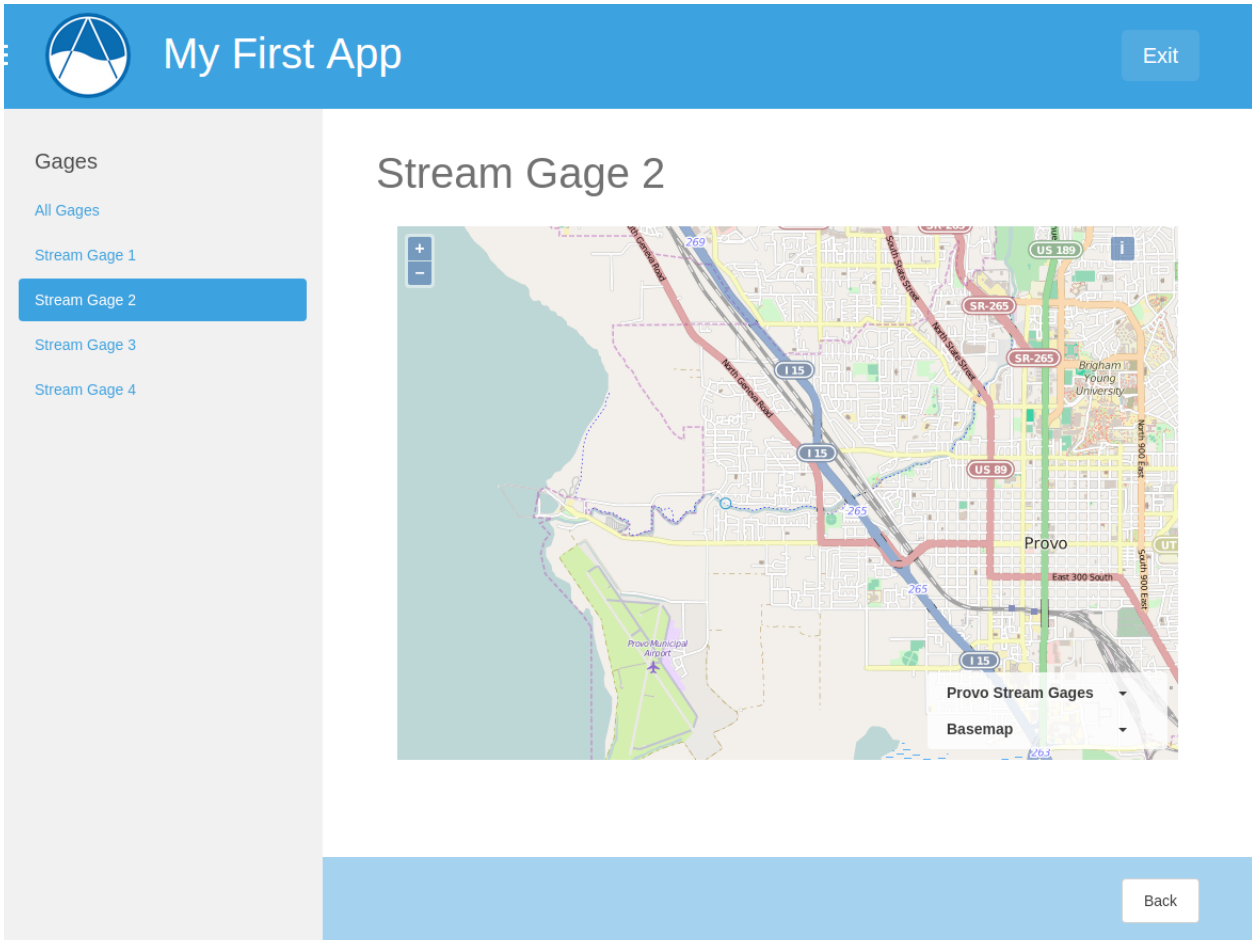
```
{% extends "my_first_app/base.html" %}

{% load tethys_gizmos %}

{% block app_navigation_items %}
  <li class="title">Gages</li>
  <li{% if not gage_id %} class="active"{% endif %}>
    <a href="{% url 'my_first_app:map' %}">All Gages</a>
  </li>
  <li{% if gage_id == '1' %} class="active"{% endif %}>
    <a href="{% url 'my_first_app:map_single' id=1 %}">Stream Gage 1</a>
  </li>
  <li{% if gage_id == '2' %} class="active"{% endif %}>
    <a href="{% url 'my_first_app:map_single' id=2 %}">Stream Gage 2</a>
  </li>
  <li{% if gage_id == '3' %} class="active"{% endif %}>
    <a href="{% url 'my_first_app:map_single' id=3 %}">Stream Gage 3</a>
  </li>
  <li{% if gage_id == '4' %} class="active"{% endif %}>
    <a href="{% url 'my_first_app:map_single' id=4 %}">Stream Gage 4</a>
  </li>
{% endblock %}

{% block app_content %}
  {% if gage_id %}
    <h1>Stream Gage {{gage_id}}</h1>
  {% else %}
    <h1>Stream Gages</h1>
  {% endif %}

  {% gizmo map_view map_options %}
{% endblock %}

{% block app_actions %}
  <a href="{% url 'my_first_app:home' %}" class="btn btn-default">Back</a>
{% endblock %}
```

There are two changes to the `map.html` template that are worth noting. First, the template now overrides the `app_navigation_block` to provide links for each of the stream gages in the navigation. The `if` template tag is used in each of the nav items to highlight the appropriate link based on the `gage_id`. Notice that all `if` tags must also end with a `endif` tag. The text between the two tags is displayed only if the conditional statement evaluates to `True`. The `href` for each link is provided using the `url`, but this time the `id` variable is also provided as an argument.

The other change to the template is the heading of the page (`<h1>`) is wrapped by `if`, `else`, and `endif` tags. The effect is to display “Stream Gage id#” when viewing only one stream gage and “Stream Gages” when viewing all of them.

View Updated Map Page

Just like that, you added a new page to your app using MVC. Save the changes to any files you edited and start up the development server using the `tethys manage start` command and browse to your app. Use the "Go To Map" action on the home page to browse to your new map page and use the options in the navigation pane to view the different gages. It should look like this (although you may need to pan and zoom some):



Variable URLs

Take note of the URL as you are viewing the different gages. You should see the ID of the current gage. For example, the URL for the gage with an ID of 1 would be http://127.0.0.1:8000/apps/my-first-app/map/1/. You can manually change the ID in the URL to request the gage with that ID. Visit this URL http://127.0.0.1:8000/apps/my-first-app/map/3/ and it will map the gage with ID 3.

Try this URL: http://127.0.0.1:8000/apps/my-first-app/map/100/. You should see a lovely error message, because you don't have a gage with ID 100 in the database. This uncovers a bug in your code that we won't take the time to fix in this tutorial. If this were a real app, you would need to handle the case when the ID doesn't match anything in the database so that it doesn't give you an error.

This exercise also exposes a vulnerability with using integer IDs in the URL–they can be guessed easily. For example if your app had a delete method, it would be very easy for an attacker to write a script that would increment through integers and call the delete method–effectively clearing your database. It would be a much better practice to use a UUID (see Universally unique identifier) or something similar for IDs.

User Input and Forms

Last Updated: September 29, 2016

Eventually you will need to request input from the user, which will involve working with HTML forms. In this tutorial, you'll learn how to create forms in your template and process the data submitted through the form in your controller.

New URL Map

The form will be created on a new page, which means you will need to create a new URL map and controller. Open your `my_first_app/app.py` and add a new `UrlMap` object called "echo_name" to the `url_maps()` method of your *app class*. The `url_maps()` method of your app class should look like this now:

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

    url_maps = (UrlMap(name='home',
                       url='my-first-app',
                       controller='my_first_app.controllers.home'),
                UrlMap(name='map',
                       url='my-first-app/map',
                       controller='my_first_app.controllers.map'),
                UrlMap(name='map_single',
                       url='my-first-app/map/{id}',
                       controller='my_first_app.controllers.map_single'),
                UrlMap(name='echo_name',
                       url='my-first-app/echo-name',
                       controller='my_first_app.controllers.echo_name'),
    )

    return url_maps
```

New Template

Create a new template called "echo_name.html" in your templates directory (`my_first_app/templates/my_first_app/echo_name.html`). Open the file and add the following contents:

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

{% load tethys_gizmos %}

{% block app_navigation_items %}
  <li class="active"><a href="{% url 'my_first_app:echo_name' %}">Name Echoer</a></li>
{% endblock %}

{% block app_content %}
  <form method="post">
      {% csrf_token %}
      {% gizmo text_input text_input_options %}
      <input type="submit" name="name-form-submit" class="btn btn-default">
  </form>
  {% if name %}
    <h1>Hello, {{ name }}!</h1>
  {% endif %}
{% endblock %}

{% block app_actions %}
  <a href="{% url 'my_first_app:home' %}" class="btn btn-default">Back</a>
{% endblock %}
```

The form is denoted by the HTML `<form>` tag and it contains a text input (created by a template Gizmo) and a submit button. Also note the use of the `csrf_token` tag. 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 HTTP method called POST to submit the data to the server. For an introduction to HTTP methods, see The Definitive Guide to GET vs POST.

New Controller

Now you need to create the `echo_name` controller function. First, add the following import statement to the top of `my_first_app/controllers.py`` file:

```
from tethys_sdk.gizmos import TextInput
```

Then add the following function to your `my_first_app/controllers.py` file:

```
@login_required
def echo_name(request):
    """
    Controller that will echo the name provided by the user via a form.
    """
    # Default value for name
    name = ''

    # Define Gizmo Options
    text_input_options = TextInput(display_text='Enter Name',
                                   name='name-input')

    # Check form data
    if request.POST and 'name-input' in request.POST:
       name = request.POST['name-input']

    # Create template context dictionary
    context = {'name': name,
               'text_input_options': text_input_options}

    return render(request, 'my_first_app/echo_name.html', context)
```

There are a few features to point out in this controller. First, the Gizmo options for the text input are defined in this controller via the `text_input_options` dictionary. The text input must have a name assigned to it for its value to be sent with the form data. In this case the name of the text input is "name-input". See the *Template Gizmos API*.

Next, the data that is submitted with HTML forms is returned through the `request` object. For forms submitted using the "post" method, the data will be accessible in the `request.POST` attribute. Similarly, form data submitted using the "get" method will be available via the `request.GET` attribute. Both `request.GET` and `request.POST` are dictionary like objects where the keys are the names of the fields from the form.

The controller contains logic that checks the `request.POST` for data with the name of the text input field, "name-input". If it exists (which it will after the user submits the form), the `name` variable is replaced with the value of the text input. The `name` variable is passed to template where it renders a nice greeting.

Link to New Page

Create a link to the new page from the home page using the `url` tag. Open the `my_first_app/templates/my_first_app/home.html` file and replace the contents with this:

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

{% block app_navigation_items %}
  <li><a href="{% url 'my_first_app:echo_name' %}">Name Echoer</a></li>
{% endblock %}

{% block app_content %}
  <h1>Welcome!</h1>
  <p>Congratulations on creating a new Tethys app!</p>
{% endblock %}

{% block app_actions %}
  <a href="{% url 'my_first_app:map' %}" class="btn btn-default">Go To Map</a>
{% endblock %}
```

View New Page

The app is ready to be tested. Run the **tethys manage start** command in the terminal and browse to your app. Use the "Name Echoer" link in the navigation to access the new page. Enter your name, press submit, and enjoy the greeting. Your new page should look something like this:

Distributing Apps

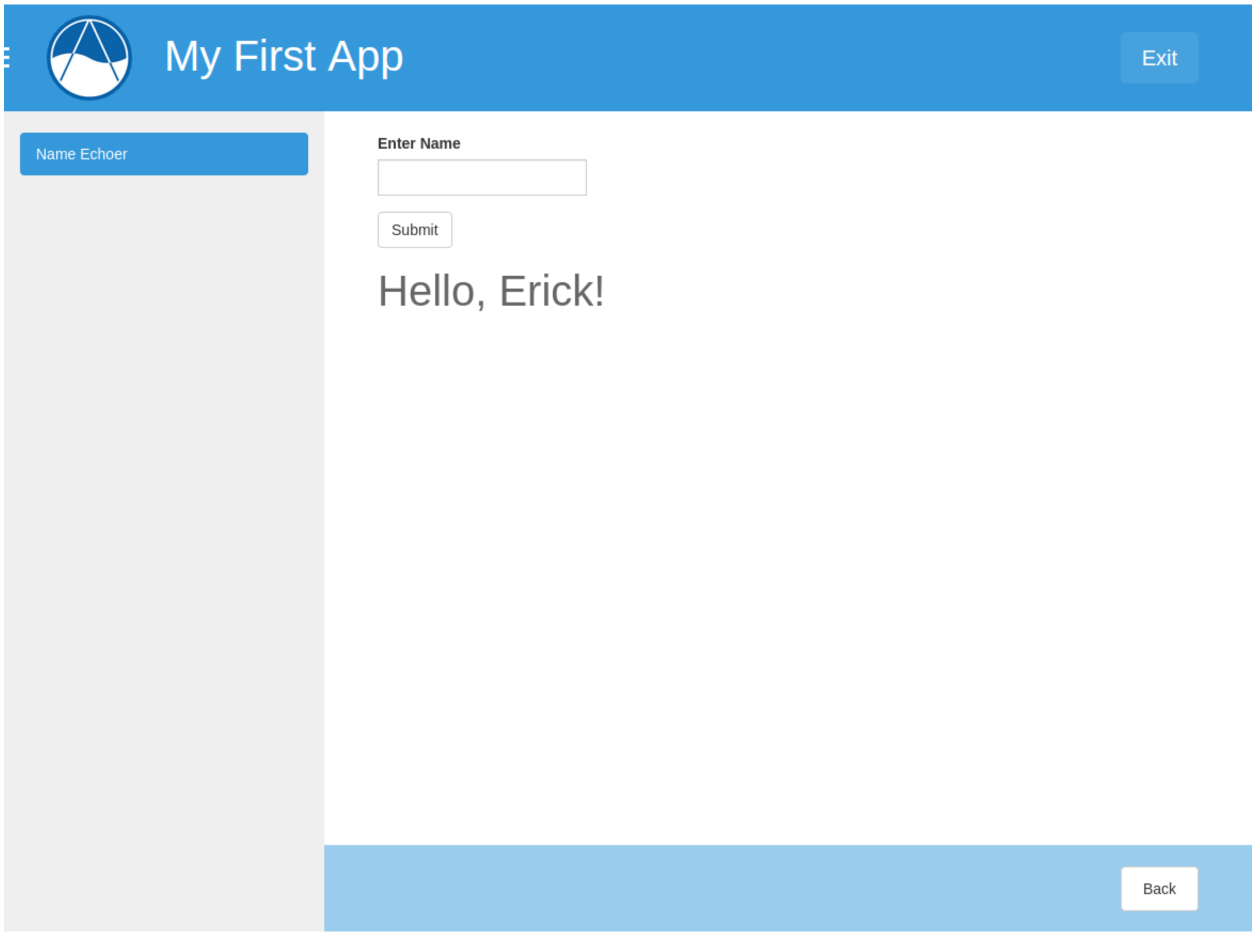
Last Updated: September 29, 2016

Once your app is complete, you will likely want to distribute it for others to use or at the very least install it in a production Tethys Platform environment. When you share your app with others, you will share the entire *release package*, which is the outermost directory of your *app project*. For these tutorials, your release package is called "tethysapp-my_first_app".

The release package contains the source code for your app and a *setup script* (`setup.py`). You may also wish to include a README file and a LICENSE file in this directory. The *setup script* can be used to streamline installation of your app and any Python dependencies it may have. You already used the *setup script* without realizing it in the *Create a New Tethys App Project* tutorial when you installed your app for the first time (this command: `python setup.py develop`). A brief introduction to the *setup script* will be provided in this tutorial.

Setup Script

When you generate your app using the scaffold, it will automatically generate a *setup script* (`setup.py`). Open the *setup script* for your app located at `~/tethysdev/tethysapp-my_first_app/setup.py`. It should look something like this:


My First App
Exit
Name Echoer
Enter Name
Submit
Hello, Erick!
Back

```python
import os
import sys
from setuptools import setup, find_packages
from tethys_apps.app_installation import custom_develop_command, custom_install_command

### Apps Definition ###
app_package = 'my_first_app'
release_package = 'tethysapp-' + app_package
app_class = 'my_first_app.app:MyFirstApp'
app_package_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'tethysapp', app_package)

### Python Dependencies ###
dependencies = []

setup(
    name=release_package,
    version='0.0',
    tags='',
    description='',
    long_description='',
    keywords='',
    author='',
    author_email='',
    url='',
    license='',
    packages=find_packages(exclude=['ez_setup', 'examples', 'tests']),
    namespace_packages=['tethysapp', 'tethysapp.' + app_package],
    include_package_data=True,
    zip_safe=False,
    install_requires=dependencies,
    cmdclass={
        'install': custom_install_command(app_package, app_package_dir, dependencies),
        'develop': custom_develop_command(app_package, app_package_dir, dependencies)
    }
)
```

As a general rule, you should never modify the parameters under the "Apps Definition" heading. These parameters are used by the *setup script* to find the source code for your app and changing their values could result in your app not working properly. If you use Python libraries that are external to your app or Tethys Platform, you will need add the library name to the `dependencies` list in the *setup script*. These libraries will automatically be installed when your app is installed.

The final part of the setup script makes a call to the `setup()` function that is provided by the `setuptools` library. You will see the metadata that you defined during the scaffold process listed here. As you release subsequent versions of your app, you may wish to increment the `version` parameter of this function.

Setup Script Installation

The setup script is used to install your app and there are two types of installation that can be performed: `install` and `develop`. The `install` type of installation hard copies the source code of your app into the `site-packages` directory of your Python installation. The `site-packages` directory is where Python keeps all of the code for external modules and libraries that have been installed.

This is the type of installation you would use for a completed app that is being installed in a production environment. To perform this type of installation, open a terminal, change into the *release package* directory of your app, and run the `install` command on the *setup script* as follows:

```
cd ~/tethysdev/tethysapp-my_first_app
python setup.py install
```

The `install` type of installation is not well suited for working with your app during development, because you would need to reinstall it (i.e.: run the commands above) every time you made a change to the app source code. This is why the `develop` type of installation exists. When an app is installed with the `develop` command, the source code for your app is only linked to the `site-packages` directory. This allows you to change your code and test the changes without reinstalling the app.

You already performed this type of installation on your app during the *Create a New Tethys App Project* tutorial. To perform this type of installation, open a terminal, change into the *release package* directory, and run the `develop` command on the *setup script* like so:

```
cd ~/tethysdev/tethysapp-my_first_app
python setup.py develop
```

Tip: For more information about `setuptools` and the *setup script*, see the Setuptools Documentation.

Spatial Dataset Services and GeoServer

Last Updated: September 30, 2016

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 `../software_suite/geoserver` and the *Spatial Dataset Services API*.

Start and Register

Last Updated: September 30, 2016

Start GeoServer Docker

Start up your `../../software_suite/geoserver` container:

```
$ tethys docker start -c geoserver
```

Register GeoServer Docker

Get the endpoint for your GeoServer Docker container:

```
$ tethys docker ip
```

Register the GeoServer with Tethys in the Portal admin page. Select the dropdown menu next to your username in the top right-hand corner of the screen and select the “Site Admin” link. Select the “Spatial Dataset Servics” link from the “Tethys Services” section and then press the “Add Spatial Dataset Service” button. Create a new Spatial Dataset Service named “default” of type GeoServer, enter the endpoint and public endpoint as the same from the print out in the terminal, and fill out the username and password.

Note: The default username and password for GeoServer is “admin” and “geoserver”, respectfully. You do not need to enter an API key.

GeoServer Web Admin Interface

Explore the GeoServer web admin interface by visiting link: http://localhost:8181/geoserver/web/.

Scaffold New App

Create a new app and install it:

```
$ tethys scaffold geoserver_app
$ cd tethysapp-geoserver_app
$ python setup.py develop
```

Download Test Files

Download the sample shapefiles that you will use to test your app:

`geoserver_app_data.zip`

Upload Shapefile

Last Updated: September 30, 2016

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="Sha
    </div>
    <input name="submit" type="submit" class="btn btn-default">
  </form>
{% endblock %}
```

Handle File Upload in Home Controller

Add these imports to the top of the `controllers.py` module:

```
import random
import string

from django.shortcuts import render
from django.contrib.auth.decorators import login_required

from tethys_sdk.gizmos import *
from tethys_sdk.services import get_spatial_dataset_engine
```

```
WORKSPACE = 'geoserver_app'
GEOSERVER_URI = 'http://www.example.com/geoserver-app'
```

Modify the `home()` controller so that it can handle the file upload event like so:

```
@login_required
def home(request):
    """
    Controller for the app home page.
    """
    # Retrieve a geoserver engine
    geoserver_engine = get_spatial_dataset_engine(name='default')

    # 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:
            response = 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
            response = geoserver_engine.create_shapefile_resource(
                        store_id=store_id,
                        shapefile_upload=file_list,
                        overwrite=True,
                        debug=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: September 30, 2016

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 = get_spatial_dataset_engine(name='default')

    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 = []

    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 action="" method="post">
        {% csrf_token %}
        {% gizmo select_input select_options %}
        <input name="submit" type="submit" value="Update" class="btn btn-default">
    </form>
    {% gizmo map_view 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: September 30, 2016

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):

    user = request.user
    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 action="" 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>
  <li><a href="{% url 'geoserver_app:home' %}">Upload Shapefile</a></li>
  <li><a href="{% url 'geoserver_app:map' %}">GeoServer Layers</a></li>
  <li><a href="{% url 'geoserver_app:draw' %}">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.

Dam Break Tutorial

Last Updated: April 12, 2016

Build the Dam Break app from scratch in this step by step tutorial. This tutorial was originally given during the CUAHSI 2015 conference on HydroInformatics in Tuscaloosa, AL. It provides a good overview of many of the features of Tethys Platform.

Get Started At: https://github.com/erdc-cm/tethysapp-dam_break/wiki

Video Tutorials

Last Updated: August 6, 2015

The following video tutorials provide more detailed demonstration of Tethys Platform's capabilities.

File Management, CKAN, and NetCDF Parsing

This tutorial was given during the National Flood Interoperability Experiment summer institute and demonstrates how to work with file dataset in Tethys apps including how to work with a CKAN server. Topics covered included the Dataset Storage API with CKAN, multi-stage forms, Highcharts plotting using the Template Gizmos API, and more.

View Solution Source Code: https://github.com/tethysplatform/tethysapp-ckan_app

GeoServer and Web Mapping

This tutorial was given during the National Flood Interoperability Experiment and demonstrates how to use GeoServer with the Web Mapping features of Tethys Platform. The topics covered include Spatial Dataset Services API, the Map View from the Tethys Gizmos API, and working with advanced user forms.

View Solution Source Code: https://github.com/tethysplatform/tethysapp-geoserver_app

PostGIS Databases and Geoprocessing

This training was given during the National Flood Interoperability Experiment and demonstrates how to create spatially-enabled databases and perform geoprocessing tasks using PostGIS. Topics covered include both Persistent Stores APIs, mapping with Template Gizmos, SQLAlchemy, GeoAlchemy, PostgreSQL, and PostGIS.

View Solution Source Code: https://github.com/tethysplatform/tethysapp-postgis_app

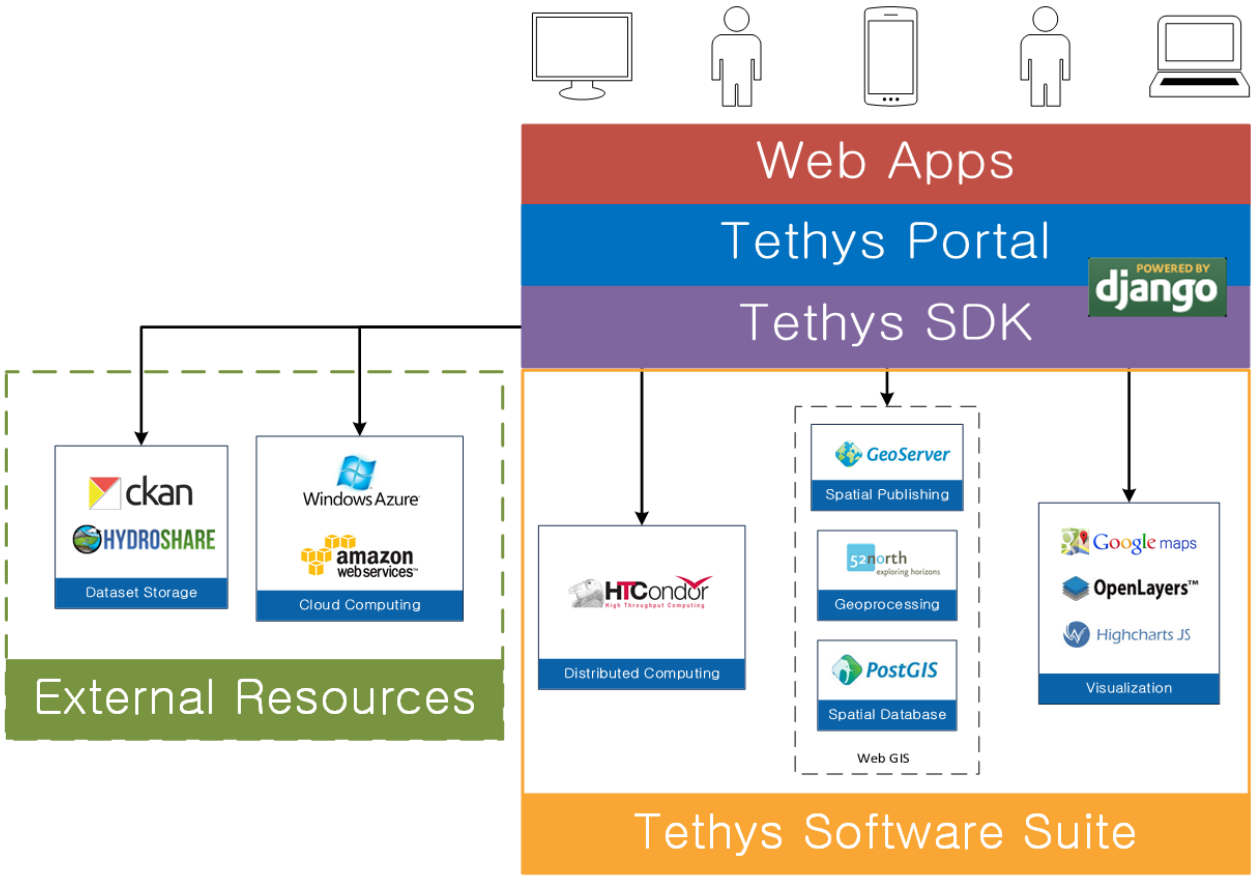
Intro to Git and Versioning

This tutorial was given during the National Flood Interoperability Experiment and provides a short introduction to versioning using Git and GitHub.

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.

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*.

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 `./software_suite/geoserver` documentation and use the *Spatial Dataset Services API*.

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*.

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*.

Distributed Computing

HTCondor
High Throughput 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*.

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.

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

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

Software Development Kit

Last Updated: May 11, 2016

The Tethys Platform provides a Python Software Development Kit (SDK) to make it easier to incorporate the functionality of the various supporting 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:

App Base Class API

Last Updated: May 11, 2016

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` class to be loaded properly into CKAN. The following article contains the API documentation for the `TethysAppBase` class.

class `tethys_apps.base.app_base.`**`TethysAppBase`**

Base class used to define the app class for Tethys apps.

`name`

string

Name of the app.

`index`

string

Lookup term for the index URL of the app.

`icon`

string

Location of the image to use for the app icon.

`package`

string

Name of the app package.

root_url
string

Root URL of the app.

color
string

App theme color as RGB hexadecimal.

description
string

Description of the app.

tag [string]
A string for filtering apps.

enable_feedback
boolean

Shows feedback button on all app pages.

feedback_emails
list

A list of emails corresponding to where submitted feedback forms are sent.

classmethod create_persistent_store(*persistent_store_name*, *spatial=False*)
Creates a new persistent store database for this app.

Parameters

- **persistent_store_name** (*string*) – Name of the persistent store that will be created.
- **spatial** (*bool*) – Enable spatial extension on the database being created.

Returns True if successful.

Return type bool

Example:

```
from .app import MyFirstApp

result = MyFirstApp.create_persistent_store('example_db')

if result:
    engine = MyFirstApp.get_persistent_store_engine('example_db')
```

classmethod destroy_persistent_store(*persistent_store_name*)
Destroys (drops) a persistent store database from this app.

Parameters **persistent_store_name** (*string*) – Name of the persistent store that will be created.

Returns True if successful.

Return type bool

Example:

```
from .app import MyFirstApp

result = MyFirstApp.destroy_persistent_store('example_db')
```

```
if result:
    # App database 'example_db' was successfuly destroyed and no longer exists
    pass
```

classmethod 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 .app import MyFirstApp

def a_controller(request):
    """
    Example controller that uses get_app_workspace() method.
    """
    # Retrieve the workspace
    app_workspace = MyFirstApp.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 get_handoff_manager()

Get the handoff manager for the app.

classmethod get_job_manager()

Get the job manager for the app

classmethod get_persistent_store_engine(*persistent_store_name*)

Creates an SQLAlchemy engine object for the app and persistent store given.

Parameters **persistent_store_name** (*string*) – Name of the persistent store for which to retrieve the engine.

Returns An SQLAlchemy engine object for the persistent store requested.

Return type object

Example:

```
from .app import MyFirstApp

engine = MyFirstApp.get_persistent_store_engine('example_db')
```

classmethod get_user_workspace(*user*)

Get the file workspace (directory) for a 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 .app import MyFirstApp

def a_controller(request):
    """
    Example controller that uses get_user_workspace() method.
    """
    # Retrieve the workspace
    user_workspace = MyFirstApp.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)
```

handoff_handlers()

Use 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

def handoff_handlers(self):
    """
    Example handoff_handlers method.
    """
    handoff_handlers = (HandoffHandlers(name='example',
                                        handler='my_first_app.controllers.my_handler'),
    )

    return handoff_handlers
```

job_templates()

Use 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

def job_templates(cls):
    """
    Example job_templates method.
    """
    my_scheduler = list_schedulers()[0]

    job_templates = (CondorJobTemplate(name='example',
                                       parameters={'executable': '$(APP_WORKSPACE)/example_e
```

```
                                                                'condorpy_template_name': 'vanilla_transf
                                                                'attributes': {'transfer_input_files': ('
                                                                               'transfer_output_files': (
                                                                              },
                                                                'scheduler': my_scheduler,
                                                                'remote_input_files': ('$(APP_WORKSPACE)/
                                                               }
                                                  ),
                                 )

        return job_templates
```

classmethod `list_persistent_stores()`

Returns a list of existing persistent stores for this app.

Returns A list of persistent store names.

Return type list

Example:

```
from .app import MyFirstApp

persistent_stores = MyFirstApp.list_persistent_stores()
```

`permissions()`

Use 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

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
```

classmethod persistent_store_exists(*persistent_store_name*)

Returns True if a persistent store with the given name exists for this app.

Parameters **persistent_store_name** (*string*) – Name of the persistent store that will be created.

Returns True if persistent store exists.

Return type bool

Example:

```
from .app import MyFirstApp

result = MyFirstApp.persistent_store_exists('example_db')

if result:
    engine = MyFirstApp.get_persistent_store_engine('example_db')
```

persistent_stores()

Define this method to register persistent store databases for your app. You may define up to 5 persistent stores for an app.

Returns A list or tuple of `PersistentStore` objects. A persistent store database will be created for each object returned.

Return type iterable

Example:

```
from tethys_sdk.stores import PersistentStore

def persistent_stores(self):
    """
    Example persistent_stores method.
    """

    stores = (PersistentStore(name='example_db',
                              initializer='init_stores:init_example_db',
                              spatial=True
              ),
    )

    return stores
```

url_maps()

Use 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

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
```

App Templating API

Last Updated: November 24, 2014

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>
{% 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.

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 %}
```

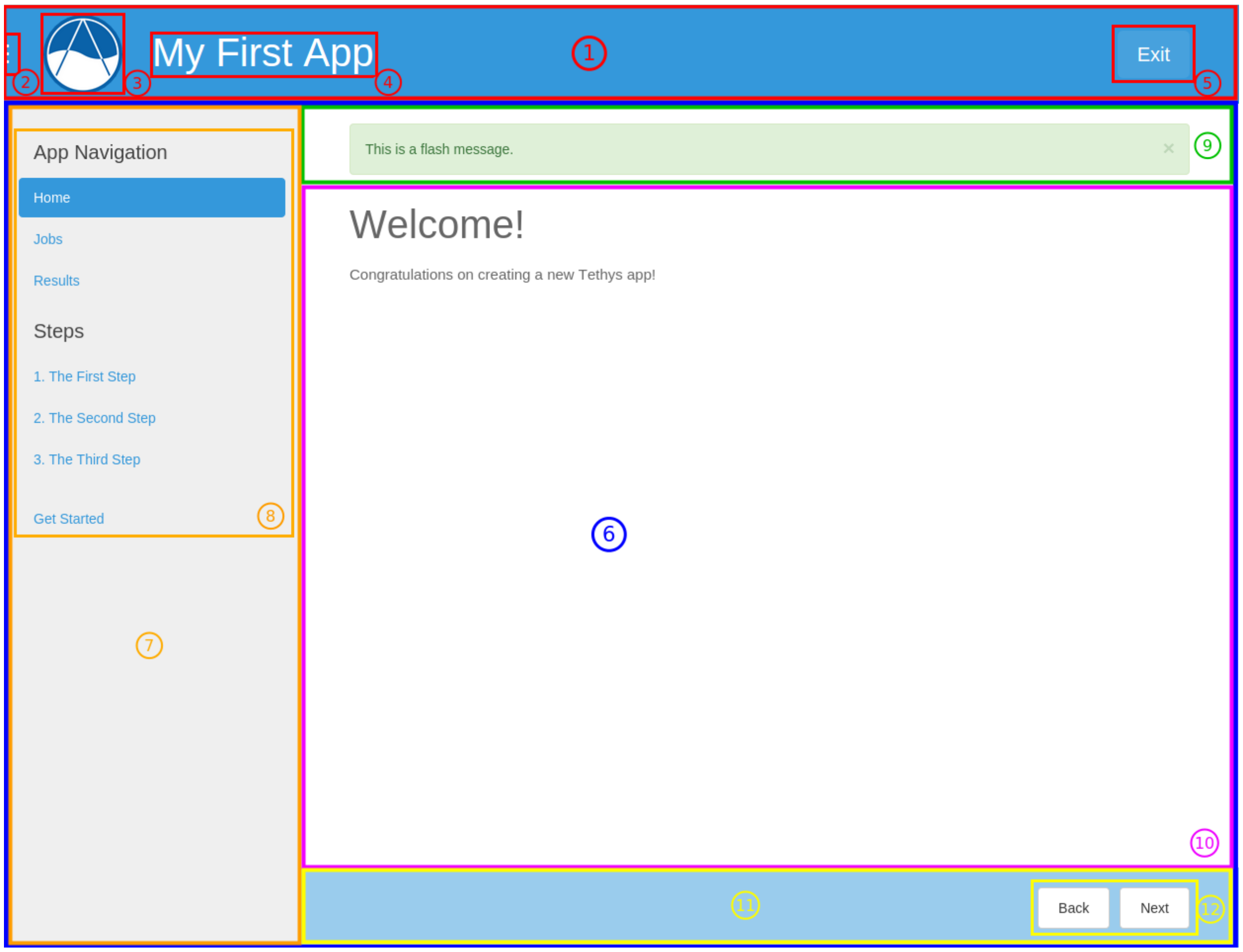


Figure 1.1: **Figure 1.** Illustration of the blocks that correspond with app interface elements as follows:

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

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 %}
```

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 %}
```

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 %}
```

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]-->
<!--[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/javascri
    {% 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>
                  <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 %}</s
                </div>
              {% endblock %}

              {% block exit_button_override %}
                <div class="exit-button">
                  <a href="javascript:void(0);" onclick="TETHYS_APP_BASE.exit_app('{% url 'app_librar
                </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-dismis
                      <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 %}
              </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
  <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 %}
```

Template Gizmos API

Last Updated: August 10, 2015

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. If you are not familiar with templating in Tethys apps, please review *The View and Templating* tutorial before proceeding.

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 Date Picker Gizmo. The basic workflow involves three steps:

1. Define gizmo options in the controller for the template
2. Load gizmo library in the template
3. Insert the gizmo tag in the template

A detailed description of each step follows.

1. Define Gizmo Options

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_options` with the appropriate options. Then we pass the object to the template via the `context` dictionary:

```
from tethys_sdk.gizmos import DatePicker

def example_controller(request):
    """
    Example of a controller that defines options for a Template Gizmo.
    """
    date_picker_options = DatePicker(name='data1',
                                     display_text='Date',
                                     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)
```

Note: The Gizmo Options objects are new as of version 1.1.0. Prior to this time, Gizmo options were defined using dictionaries. The dictionary parameterization of Gizmos is still supported, but will no longer be referenced in the documentation.

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

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 %}
```

3. Insert the Gizmo

The `gizmo` tag is used to insert the date picker anywhere in the template. The `gizmo` tag accepts two arguments: the name of the Gizmo to insert and a dictionary of configuration options for the Gizmo:

```
{% gizmo <name> <options> %}
```

For this example, the `date_picker` Gizmo is inserted and the `date_picker_options` object that was defined in the controller and passed to the template is provided:

```
{% gizmo date_picker date_picker_options %}
```

Rendered Gizmo

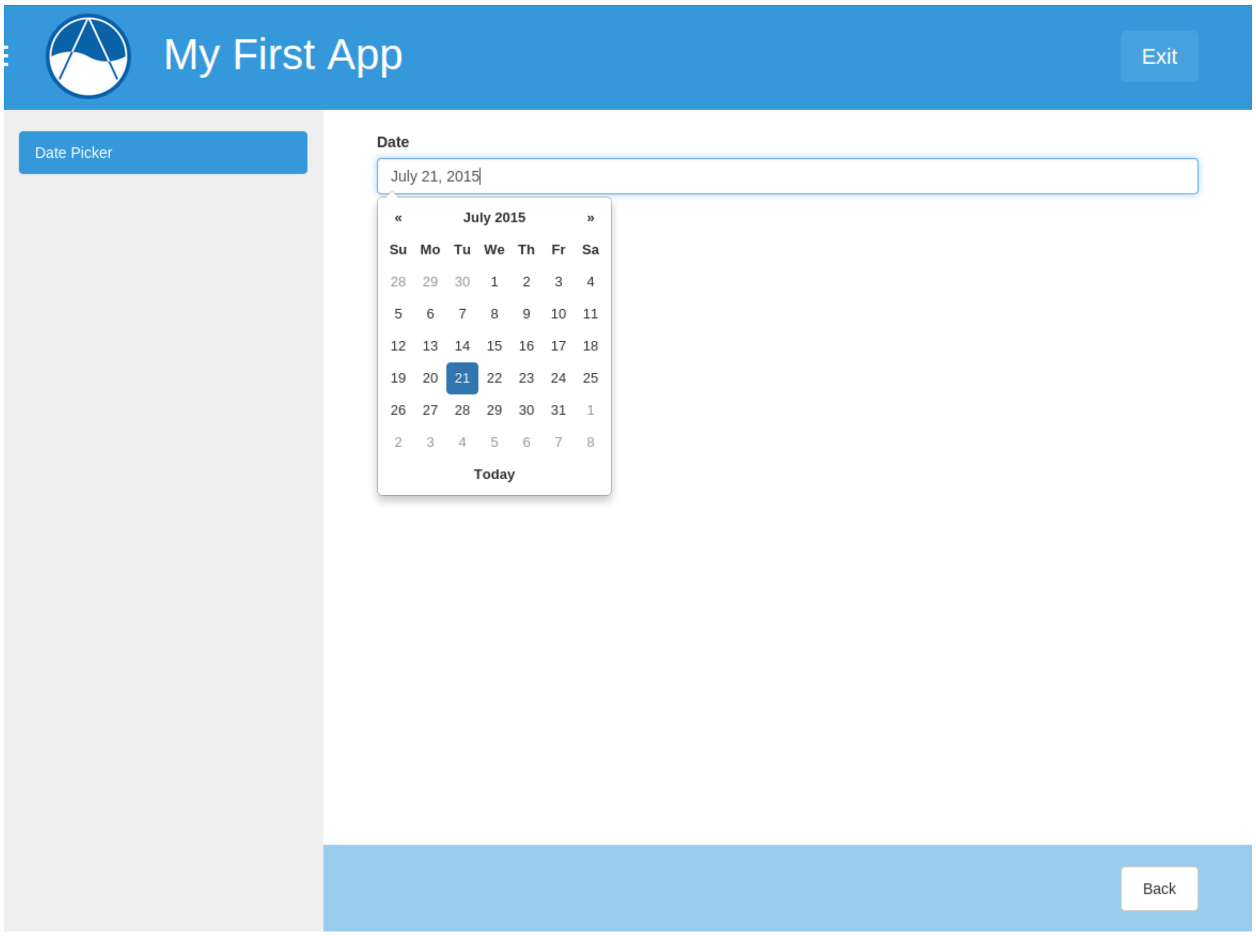
The Gizmo tag is replaced with the appropriate HTML, JavaScript, and CSS that is needed to render the Gizmo. In the example, the date picker is inserted at the location of the `gizmo` tag. The template with the rendered date picker would look something like this:

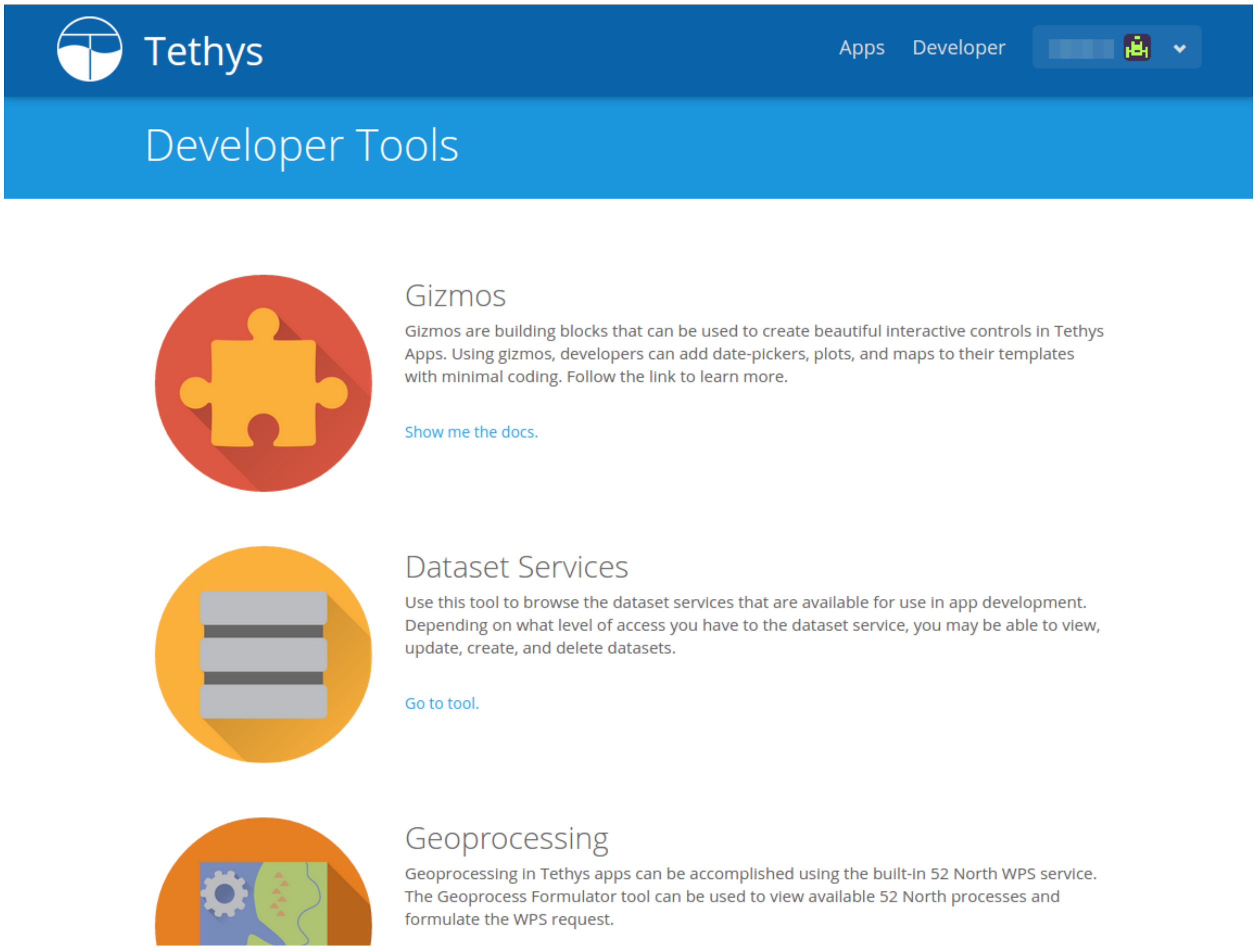
Gizmo Showcase

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.


My First App
Exit
Date Picker
Date
July 21, 2015
«
July 2015
»
Su Mo Tu We Th Fr Sa
28 29 30 1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31 1
2 3 4 5 6 7 8
Today
Back



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.

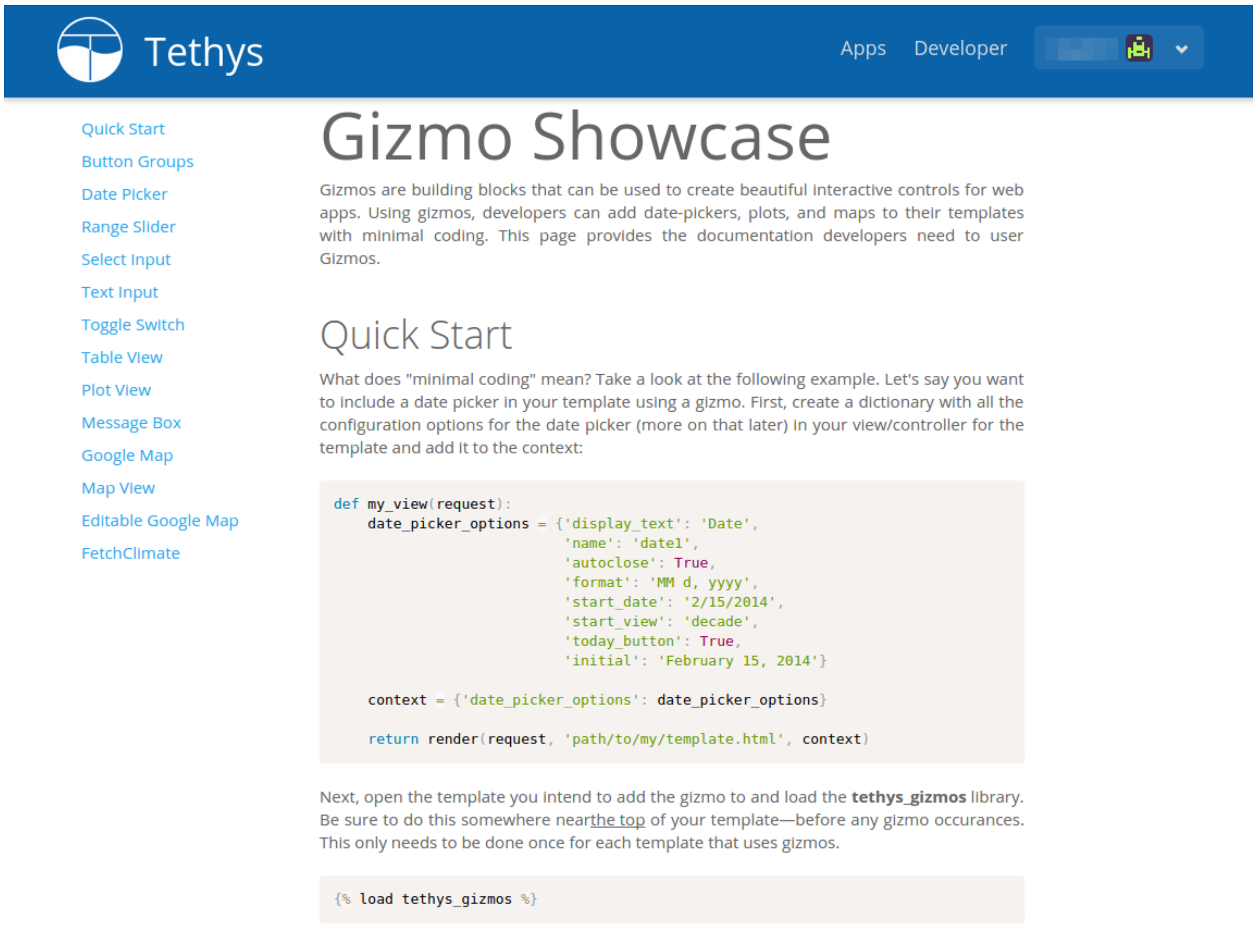
Dataset Services

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.



- 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 %}
```

Django Tag Reference

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:

- **name** (string or literal) - The name of the Gizmo to insert as either a string (e.g.: "date_picker") or a literal (e.g.: date_picker).
- **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.

Examples:

```
# With literal for name parameter
{% gizmo date_picker date_picker_options %}

# With string for name parameter
{% gizmo "date_picker" date_picker_options %}
```

gizmo_dependencies

Inserts the CSS and JavaScript dependencies at the location of the 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 or "js" for JavaScript dependencies.

Examples:

```
# No type parameter
{% gizmo_dependencies %}

# CSS only
{% gizmo_dependencies css %}

# JavaScript only
{% gizmo_dependencies js %}
```

Gizmos Options Objects

This section provides explanations of each of the Gizmo Options Objects available for configuring Gizmos. It also provide 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`
str

Display text that appears on the button.

`name`
str

Name of the input element that will be used for form submission.

`style`
str

Name of the input element that will be used for form submission.

`icon`
str

Name of a valid Twitter Bootstrap icon class (see the Bootstrap glyphicon reference).

`href`
str

Link for anchor type buttons.

`submit`
bool

Set this to true to make the button a submit type button for forms.

`disabled`
bool

Set the disabled state.

`attributes`
dict

A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).

`classes`
str

Additional classes to add to the primary HTML element (e.g. "example-class another-class").

Example:

```
# CONTROLLER

from tethys_sdk.gizmos import Button

# Single Button
single_button = Button(display_text='Click Me',
                       name='click_me_name',
                       attributes={"onclick": "alert(this.name);"},
                       submit=True)
```

```
# TEMPLATE

{% gizmo button single_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`
list, required

A list of dictionaries where each dictionary contains the options for a button.

`vertical`
bool

Set to true to have button group stack vertically.

`attributes`
str

A string representing additional HTML attributes to add to the primary element (e.g. "onclick=run_me();").

`classes`
str

Additional classes to add to the primary HTML element (e.g. "example-class another-class").

Example

```
# CONTROLLER

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)
```

```
# TEMPLATE

{% gizmo button_group horizontal_buttons %}
{% gizmo button_group 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
str, required

Name of the input element that will be used for form submission.

display_text
str

Display text for the label that accompanies date picker.

autoclose
bool

Set whether datepicker auto closes when a date is selected.

calendar_weeks
bool

Set whether calendar week numbers are shown on the left of the datepicker.

clear_button
bool

Set whether the clear button is displayed or not.

days_of_week_disabled
str

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').

end_date
str

Last date that can be selected. All other dates after this date are shown as disabled.

format
str

String representing date format. For valid formats see Bootstrap Datepicker documentation here.

min_view_mode
str

Set the minimum view mode. Possible values are 'days', 'months', 'years'.

multidate
int

Enables multi-selection of dates up to the number given.

start_date
str

First date that can be selected. All other dates before this date are shown as disabled.

start_view
str

View the date picker starts on. Valid values include 'month', 'year', and 'decade'.

today_button
bool

Set whether a today button is displayed or not.

today_highlight
bool

Set whether to highlight the current date.

week_start
int

Set the day the week starts on 0-6, where 0 is Sunday and 6 is Saturday.

initial
str

Initial date to appear in date picker.

disabled
bool

Disabled state of the date picker.

error
str

Error message for form validation.

attributes
dict

A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).

classes
str

Additional classes to add to the primary HTML element (e.g. "example-class another-class").

Example

```
# CONTROLLER

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.')

# TEMPLATE

{% gizmo date_picker date_picker %}
{% gizmo date_picker 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`
str

Display text for the label that accompanies slider

`name`
str, required

Name of the input element that will be used on form submission

`min`
int, required

Minimum value of range

`max`
int, required

Maximum value of range

`initial`
int, required

Initial value of slider

`step`
int, required

Increment between values in range

disabled
bool

Disabled state of the slider

error
str

Error message for form validation

attributes
dict

A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).

classes
str

Additional classes to add to the primary HTML element (e.g. "example-class another-class").

Example

```
# CONTROLLER
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.')

# TEMPLATE

{% gizmo range_slider slider1 %}
{% gizmo range_slider slider2 %}
```

Select Input

Last Updated: August 10, 2015

class `tethys_sdk.gizmos.`**`SelectInput`**(*name*, *display_text=''*, *initial=[]*, *multiple=False*, *original=False*, *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
str

Display text for the label that accompanies select input

name
str, required

Name of the input element that will be used for form submission

multiple
bool

If True, select input will be a multi-select

original
bool

If True, Select2 reference functionality will be turned off

options
list

List of tuples that represent the options and values of the select input

initial
list or str

List of keys or values that represent the initial selected values or a string representing a singular initial selected value.

disabled
bool

Disabled state of the select input

error
str

Error message for form validation

attributes
dict

A dictionary representing additional HTML attributes to add to the primary element (e.g. {“onclick”: “run_me();”}).

classes
str

Additional classes to add to the primary HTML element (e.g. “example-class another-class”).

Example

```
# CONTROLLER
from tethys_sdk.gizmos import SelectInput

select_input2 = SelectInput(display_text='Select2',
                            name='select1',
                            multiple=False,
                            options=[('One', '1'), ('Two', '2'), ('Three', '3')],
                            initial=['Three'],
                            original=['Two'])

select_input2_multiple = SelectInput(display_text='Select2 Multiple',
                                     name='select2',
                                     multiple=True,
                                     options=[('One', '1'), ('Two', '2'), ('Three', '3')],
                                     initial=['1', '2'])
```

```
select_input_multiple = SelectInput(display_text='Select Multiple',
                                    name='select2.1',
                                    multiple=True,
                                    original=True,
                                    options=[('One', '1'), ('Two', '2'), ('Three', '3')])

select_input2_error = SelectInput(display_text='Select2 Disabled',
                                  name='select3',
                                  multiple=False,
                                  options=[('One', '1'), ('Two', '2'), ('Three', '3')],
                                  disabled=True,
                                  error='Here is my error text')

# TEMPLATE

{% gizmo select_input select_input2 %}
{% gizmo select_input select_input2_multiple %}
{% gizmo select_input select_input_multiple %}
{% gizmo select_input select_input2_error %}
```

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`
str

Display text for the label that accompanies select input

`name`
str, required

Name of the input element that will be used for form submission

`initial`
str

The initial text that will appear in the text input when it loads

`placeholder`
str

Placeholder text is static text that displayed in the input when it is empty

`prepend`
str

Text that is prepended to the text input

`append`
str

Text that is appended to the text input

icon_prepend
str

The name of a valid Bootstrap v2.3 icon. The icon will be prepended to the input.

icon_append
str

The name of a valid Bootstrap v2.3 icon. The icon will be appended to the input.

disabled
bool

Disabled state of the select input

error
str

Error message for form validation

attributes
dict

A dictionary representing additional HTML attributes to add to the primary element (e.g. {“onclick”: “run_me();”}).

classes
str

Additional classes to add to the primary HTML element (e.g. “example-class another-class”).

Example

```
# CONTROLLER
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')

# TEMPLATE

{% gizmo text_input text_input %}
{% gizmo text_input 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`
str

Display text for the label that accompanies switch

`name`
str, required

Name of the input element that will be used for form submission

`on_label`
str

Text that appears in the "on" position of the switch

`off_label`
str

Text that appears in the "off" position of the switch

`on_style`
str

Color of the "on" position. Either: 'default', 'info', 'primary', 'success', 'warning', or 'danger'

`off_style`
str

Color of the "off" position. Either: 'default', 'info', 'primary', 'success', 'warning', or 'danger'

`size`
str

Size of the switch. Either: 'large', 'small', or 'mini'.

`initial`
bool

The initial position of the switch (True for "on" and False for "off")

`disabled`
bool

Disabled state of the switch

`error`
str

Error message for form validation

`attributes`
dict

A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).

`classes`
str

Additional classes to add to the primary HTML element (e.g. "example-class another-class").

Example

```
# CONTROLLER
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')

# TEMPLATE

{% gizmo toggle_switch toggle_switch %}
{% gizmo toggle_switch toggle_switch_styled %}
{% gizmo toggle_switch toggle_switch_disabled %}
```

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`
str, required

Unique name for the message box

`title`
str, required

Title that appears at the top of the message box

`message`
str

Message that will appear in the main body of the message box

`dismiss_button`
str

Title for the dismiss button (a.k.a.: the "Cancel" button)

affirmative_button
str

Title for the affirmative action button (a.k.a.: the "OK" button)

affirmative_attributes
str

Use this to place any html attributes on the affirmative button. (e.g.: 'href="/action" onclick="doSomething();"')

width
int

The width of the message box in pixels

attributes
dict

A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).

classes
str

Additional classes to add to the primary HTML element (e.g. "example-class another-class").

Example

```
# CONTROLLER
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);')

# TEMPLATE

<a href="#sampleModal" role="button" class="btn btn-success" data-toggle="modal">Show Message Bo
{% gizmo message_box message_box %}
```

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
tuple or list, required

A list/tuple of lists/tuples representing each row in the table.

column_names
tuple or list

A tuple or list of strings that represent the table columns names.

hover
bool

Illuminate rows on hover (does not work on striped tables)

striped
bool

Stripe rows

bordered
bool

Add borders and rounded corners

condensed
bool

A more tightly packed table

editable_columns
list or tuple

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.

row_ids
list or tuple

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.

attributes
dict

A dictionary representing additional HTML attributes to add to the primary element (e.g. {“onclick”: “run_me();”}).

classes
str

Additional classes to add to the primary HTML element (e.g. “example-class another-class”).

Example

```
# CONTROLLER
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])

# TEMPLATE

{% gizmo table_view table_view %}
{% gizmo table_view table_view_edit %}
```

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`
list, required

A list of series dictionaries.

`height`
str

Height of the plot element. Any valid css unit of length.

`width`
str

Width of the plot element. Any valid css unit of length.

`engine`
str

The plot engine to be used for rendering, either 'd3' or 'highcharts'. Defaults to 'd3'.

`title`
str

Title of the plot.

subtitle
str

Subtitle of the plot.

spline
bool

If True, lines are smoothed using a spline technique.

x_axis_title
str

Title of the x-axis.

x_axis_units
str

Units of the x-axis.

y_axis_title
str

Title of the y-axis.

y_axis_units
str

Units of the y-axis.

Example

```
# coding=utf-8

# CONTROLLER
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',
           '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]
           ]
       }
    ]
)

# TEMPLATE

{% gizmo plot_view 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
list, required

A list of series dictionaries.

height
str

Height of the plot element. Any valid css unit of length.

width
str

Width of the plot element. Any valid css unit of length.

engine
str

The plot engine to be used for rendering, either 'd3' or 'highcharts'. Defaults to 'd3'.

title
str

Title of the plot.

subtitle
str

Subtitle of the plot.

spline
bool

If True, lines are smoothed using a spline technique.

x_axis_title
str

Title of the x-axis.

x_axis_units
str

Units of the x-axis.

y_axis_title
str

Title of the y-axis.

y_axis_units
str

Units of the y-axis.

Example

```
# coding=utf-8

# CONTROLLER
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],
        [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
    ]
)

# TEMPLATE

{% gizmo plot_view 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`
list, required

A list of series dictionaries.

`height`
str

Height of the plot element. Any valid css unit of length.

`width`
str

Width of the plot element. Any valid css unit of length.

`engine`
str

The plot engine to be used for rendering, either 'd3' or 'highcharts'. Defaults to 'd3'.

`title`
str

Title of the plot.

`subtitle`
str

Subtitle of the plot.

categories
list

List of category names, one for each data point in the series.

Example

```
# coding=utf-8

# CONTROLLER
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'
      }
    ]
)

# TEMPLATE

{% gizmo plot_view 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
list, required

A list of series dictionaries.

height
str

Height of the plot element. Any valid css unit of length.

width
str

Width of the plot element. Any valid css unit of length.

engine
str

The plot engine to be used for rendering, either ‘d3’ or ‘highcharts’. Defaults to ‘d3’.

title
str

Title of the plot.

subtitle
str

Subtitle of the plot.

Example

```
# coding=utf-8

# CONTROLLER
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}
    ]
)

# TEMPLATE

{% gizmo plot_view 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, **kwargs*)

Bar Plot

Displays as either a bar or column chart.

series
list, required

A list of series dictionaries.

height
str

Height of the plot element. Any valid css unit of length.

width
str

Width of the plot element. Any valid css unit of length.

engine
str

The plot engine to be used for rendering, either ‘d3’ or ‘highcharts’. Defaults to ‘d3’.

title
str

Title of the plot.

subtitle
str

Subtitle of the plot.

horizontal
bool

If True, bars are displayed horizontally, otherwise they are displayed vertically.

categories
list

A list of category titles, one for each bar.

axis_title
str

Title of the axis.

axis_units
str

Units of the axis.

Example

```
# coding=utf-8

# CONTROLLER
from tethys_sdk.gizmos import BarPlot

bar_plot_view = BarPlot(
    height='500px',
    width='500px',
    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]
        }
    ]
)

# TEMPLATE

{% gizmo plot_view 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='', **kwargs*)

Use to create a timeseries plot visualization

`series`
list, required

A list of series dictionaries.

`height`
str

Height of the plot element. Any valid css unit of length.

`width`
str

Width of the plot element. Any valid css unit of length.

`engine`
str

The plot engine to be used for rendering, either 'd3' or 'highcharts'. Defaults to 'd3'.

`title`
str

Title of the plot.

`subtitle`
str

Subtitle of the plot.

`y_axis_title`
str

Title of the axis.

`y_axis_units`
str

Units of the axis.

Example

```
# coding=utf-8

# CONTROLLER
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]
        ]
    }]
)

# TEMPLATE

{% gizmo plot_view 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`
list, required

A list of series dictionaries.

`height`
str

Height of the plot element. Any valid css unit of length.

width
str

Width of the plot element. Any valid css unit of length.

engine
str

The plot engine to be used for rendering, either 'd3' or 'highcharts'. Defaults to 'd3'.

title
str

Title of the plot.

subtitle
str

Subtitle of the plot.

y_axis_title
str

Title of the axis.

y_axis_units
str

Units of the axis.

Example

```
# coding=utf-8

# CONTROLLER
from tethys_sdk.gizmos import AreaRange

averages = [
    [datetime(2009, 7, 1), 21.5], [datetime(2009, 7, 2), 22.1], [datetime(2009, 7, 3), 23],
    [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
    [datetime(2009, 7, 4), 16.7, 30.7], [datetime(2009, 7, 5), 16.5, 25.0], [datetime(2009, 7, 6
    [datetime(2009, 7, 7), 13.5, 24.8], [datetime(2009, 7, 8), 10.5, 21.4], [datetime(2009, 7, 9
    [datetime(2009, 7, 10), 11.6, 21.8], [datetime(2009, 7, 11), 10.7, 23.7], [datetime(2009, 7,
    [datetime(2009, 7, 13), 11.6, 23.7], [datetime(2009, 7, 14), 11.8, 20.7], [datetime(2009, 7,
    [datetime(2009, 7, 16), 13.6, 19.6], [datetime(2009, 7, 17), 11.4, 22.6], [datetime(2009, 7,
    [datetime(2009, 7, 19), 14.2, 21.6], [datetime(2009, 7, 20), 13.1, 17.1], [datetime(2009, 7,
    [datetime(2009, 7, 22), 12.0, 20.8], [datetime(2009, 7, 23), 12.0, 17.1], [datetime(2009, 7,
    [datetime(2009, 7, 25), 12.4, 19.4], [datetime(2009, 7, 26), 12.6, 19.9], [datetime(2009, 7,
    [datetime(2009, 7, 28), 11.0, 19.3], [datetime(2009, 7, 29), 10.8, 17.8], [datetime(2009, 7,
    [datetime(2009, 7, 31), 10.8, 16.1]
]

area_range_plot_object = AreaRange(
    title='July Temperatures',
    y_axis_title='Temperature',
    y_axis_units='*C',
    series=[{
        'name': 'Temperature',
        'data': averages,
        'zIndex': 1,
        'marker': {
            'lineWidth': 2,
        }
    }, {
        'name': 'Range',
        'data': ranges,
        'type': 'arearange',
        'lineWidth': 0,
        'linkedTo': ':previous',
        'fillOpacity': 0.3,
        'zIndex': 0
    }]
)

area_range_plot = PlotView(_object=area_range_plot_object,
                           width='500px',
                           height='500px')

# TEMPLATE

{% gizmo plot_view area_range_plot %}
```

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 or import the JavaScript libraries in the main html template page.

For example:

```
{% block global_scripts %}
  {{ block.super }}
  <script src="/static/tethys_gizmos/vendor/highcharts/js/highcharts.js" type="text/javascript"></scr
  <script src="/static/tethys_gizmos/vendor/highcharts/js/highcharts-more.js" type="text/javascript">
{% endblock %}

...

{% block scripts %}
  {{ block.super }}
  <script src="/static/tethys_gizmos/js/plot_view.js" type="text/javascript"></script>
{% 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.
    """
    hydrograph = ... #insert data here

    ...

    # Configure the Hydrograph Plot View
    flood_plot = TimeSeries(
        title='Flood Hydrograph',
        y_axis_title='Flow',
        y_axis_units='cms',
        series=[
           {
               'name': 'Flood Hydrograph',
               'color': '#0066ff',
               'data': hydrograph,
           },
        ],
        width='500px',
        height='500px'
    )

    context = {'flood_plot': flood_plot}

    return render(request, 'dam_break/hydrograph_ajax.html', context)
```

2) A url map to the controller in app.py

```
...
    UrlMap(name='hydrograph_ajax',
           url='dam-break/map/hydrograph',
           controller='dam_break.controllers.hydrograph_ajax'),
...
```

3) A template for with the tethys gizmo (e.g. hydrograph_ajax.html)

```
{% load tethys_gizmos %}

{% gizmo highcharts_plot_view flood_plot %}
```

4) The AJAX call in the javascript

```
$(function() { //wait for page to load

    $.ajax({
        url: 'hydrograph',
        method: 'GET',
        data: {
            'peak_flow': 500, //example data to pass to the controller
        },
        success: function(data) {
            //Initialize Plot
            TETHYS_PLOT_VIEW.initHighChartsPlot($('.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();
```

Map View

class `tethys_sdk.gizmos.`**`MapView`**(*height='100%', width='100%', basemap='OpenStreetMap', 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 3, 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`
str

Height of the map element. Any valid css unit of length (e.g.: '500px'). Defaults to '520px'.

`width`
str

Width of the map element. Any valid css unit of length (e.g.: '100%'). Defaults to '100%'.

`basemap`
str or dict

The base map to dispaly: either 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.

`view`
MVView

An MVView object specifying the initial view or extent for the map.

`controls`
list

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.

layers
list

A list of MVLayer objects.

draw
MVDraw

An MVDraw object specifying the drawing options.

disable_basemap
bool

Render the map without a base map.

feature_selection
bool

A dictionary of global feature selection options. See below.

attributes
dict

A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).

classes
str

Additional classes to add to the primary HTML element (e.g. "example-class another-class").

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 three base maps supported by the Map View gizmo: OpenStreetMap, Bing, and MapQuest. Use the following links to learn about the additional options you can configure the base maps with:

- Bing: ol.source.BingMaps
- MapQuest: ol.source.MapQuest
- OpenStreetMap: ol.source.OSM

```
{'Bing': {'key': 'Ap|k3yheRE', 'imagerySet': 'Aerial'}}
```

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.

Example

```
# CONTROLLER
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',
        '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)', s
                            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='#0
                              MVLegendClass('polygon', 'Medium Density', fill='#ff0000', stroke=
                              MVLegendClass('polygon', 'High Density', fill='#0000ff', stroke='#
                          ])

# 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/servi
                        legend_title='ESRI USA Highway',
                        legend_extent=[-173, 17, -65, 72])

# 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, geoserver_layer, kml_layer, arc_gis_layer],
        view=view_options,
        basemap='OpenStreetMap',
        draw=drawing_options,
        legend=True
)

# TEMPLATE

{% gizmo map_view map_view_options %}
```

MVLayer

class `tethys_sdk.gizmos.`**`MVLayer`**(*source*, *options*, *legend_title*, *layer_options=None*, *legend_classes=None*, *legend_extent=None*, *legend_extent_projection='EPSG:4326'*, *feature_selection=False*, *geometry_attribute=None*)

MVLayer objects are used to define map layers for the Map View Gizmo.

`source`
str, required

The source or data type of the layer (e.g.: ImageWMS)

`options`
dict, required

A dictionary representation of the OpenLayers options object for ol.source.

`legend_title`
str, required

The human readable name of the layer that will be displayed in the legend.

`layer_options`
dict

A dictionary representation of the OpenLayers options object for ol.layer.

`feature_selection`
bool

Set to True to enable feature selection on this layer. Defaults to False.

`geometry_attribute`
str

The name of the attribute in the shapefile that describes the geometry

`legend_classes`
list

A list of MVLegendClass objects.

`legend_extent`
list

A list of four ordinates representing the extent that will be used on “zoom to layer”: [minx, miny, maxx, maxy].

`legend_extent_projection`
str

The EPSG projection of the extent coordinates. Defaults to “EPSG:4326”.

Example

```
# 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',
        '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)', s
                            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='#0
                              MVLegendClass('polygon', 'Medium Density', fill='#ff0000', stroke=
                              MVLegendClass('polygon', 'High Density', fill='#0000ff', stroke='#
                          ])

# 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
# to use GeoWebCaching capabilities. This is done by setting the TILESORIGIN parameter and speci
# 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,
                                              '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, 2003750
                                     '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/servi
                        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`

str, required

The type of feature to be represented by the legend class. Either 'point', 'line', 'polygon', or 'raster'.

`value`

str, required

The value or name of the legend class.

`fill`

str

Valid RGB color for the fill (e.g.: '#00ff00', 'rgba(0, 255, 0, 0.5)'). Required for 'point' or 'polygon' types.

`stoke`

str

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.

`ramp`

list

A list of hexidecimal RGB colors that will be used to construct a color ramp. Required for 'raster' types.

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`

str, required

The value or name of the legend class.

`image_url`

str, required

The url to the legend image.

Example

```
image_class = MVLegendImageClass(value='Cities',
                                 image_url='https://upload.wikimedia.org/wikipedia/commons/d/da/
                                 )
```

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`
str, required

The value or name of the legend class.

`geoserver_url`
str, required

The url to your geoserver (e.g. http://localhost:8181/geoserver).

`style`
str, required

The name of the geoserver style (e.g. green).

`layer`
str, required

The name of the geoserver layer (e.g. rivers).

`width`
int

The legend width (default is 20).

`height`
int

The legend height (default is 10).

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'*)

MVDraw objects are used to define the drawing options for Map View.

`controls`
list, required

List of drawing controls to add to the map. Valid options are 'Modify', 'Delete', 'Move', 'Point', 'LineString', 'Polygon' and 'Box'.

`initial`
str, required

Drawing control to be enabled initially. Must be included in the controls list.

`output_format`
str

Format to output to the hidden text area. Either 'WKT' (for Well Known Text format) or 'GeoJSON'. Defaults to 'GeoJSON'

Example

```
drawing_options = MVDraw(
    controls=['Modify', 'Delete', 'Move', 'Point', 'LineString', 'Polygon', 'Box'],
    initial='Point',
    output_format='WKT'
)
```

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`
str

Projection of the center coordinates given. This projection will be used to transform the coordinates into the default map projection (EPSG:3857).

`center`
list

An array with the coordinates of the center point of the initial view.

`zoom`
int or float

The zoom level for the initial view.

`maxZoom`
int or float

The maximum zoom level allowed. Defaults to 28.

`minZoom`
int or float

The minimum zoom level allowed. Defaults to 0.

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 3.10.1 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 3.10.1. 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,
                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) {
      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.

For example:

```
{% block styles %}
  {{ block.super }}
  <link rel="stylesheet" href="/static/tethys_gizmos/vendor/openlayers/ol.css"" type="text/css">
{% endblock %}

{% block global_scripts %}
  {{ block.super }}
  <script src="/static/tethys_gizmos/vendor/openlayers/ol.js" type="text/javascript"></script>
{% endblock %}

...

{% block scripts %}
  {{ block.super }}
  <script src="/static/tethys_gizmos/js/tethys_map_view.js" type="text/javascript"></script>
{% endblock %}
```

Four elements are required:

1) A controller for the AJAX call with a map view gizmo.

```python
@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',
                              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_view 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();
        }
    });

});
```

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=''*)

Google Map View

The Google Map View is similar to Map View, but it 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`
string, required

Height of map container in normal css units

`width`
string, required

Width of map container in normal css units

`maps_api_key`
string, required

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.

`reference_kml_action`
url string

The action that returns the background kml datasets. These datasets are used for reference only.

`drawing_types_enabled`
list of strings

A list of the types of geometries the user will be allowed to draw (POLYGONS, POINTS, POLYLINES).

`initial_drawing_mode`
string

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.

`output_format`
string

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.

`input_overlays`
PySON

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.

attributes
dict

A dictionary representing additional HTML attributes to add to the primary element (e.g. {“onclick”: “run_me();”}).

classes
str

Additional classes to add to the primary HTML element (e.g. “example-class another-class”).

Example

```
# CONTROLLER
from tethys_sdk.gizmos import GoogleMapView

google_map_view = 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')

# 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
                   'properties':{'id':2,'value':2}
                   },
                  {'type':'PolyLine',#
                   'wkt':'POLYLINE(-111.49123191833496 40.65003865742191, -111.49088859558105 40
                   'properties':{'id':3,'value':3}
                   }
                  ]
    }

google_map_view_options = {'height': '700px',
                           'width': '100%',
                           'maps_api_key': 'S0mEaPIk3y',
                           'drawing_types_enabled': ['POLYGONS', 'POINTS', 'POLYLINES'],
                           'initial_drawing_mode': 'POINTS',
                           'input_overlays': geo_json}

# 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,-1
                    "properties":{"id":2,"value":2}},
                   {"type":"LineString",
                    "coordinates":[[40.65003865742191,-111.49123191833496],[40.635319920747456,-1
                    "properties":{"id":3,"value":3}}
                   ]
         }

    google_map_view_options = {'height': '700px',
                               'width': '100%',
                               'maps_api_key': 'S0mEaPIk3y',
                               'drawing_types_enabled': ['POLYGONS', 'POINTS', 'POLYLINES'],
                               'initial_drawing_mode': 'POINTS',
                               'input_overlays': wkt_json}

    # TEMPLATE

    {% gizmo google_map_view 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`
tuple or list, required

A list/tuple of TethysJob objects.

`column_fields`
tuple or list, required

A tuple or list of strings that represent TethysJob object attributes to show in the columns.

`status_actions`
bool

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.

`run_btn`
bool

Add a button to run the job when job status is "Pending". Default is True.

`delete_btn`
bool

Add a button to delete jobs. Default is True.

`results_url`
str

A string representing the namespaced path to a controller to that displays job results (e.g. app_name:results_controller)

`hover`
bool

Illuminate rows on hover (does not work on striped tables)

`striped`
bool

Stripe rows

`bordered`
bool

Add borders and rounded corners

condensed
bool

A more tightly packed table

attributes
dict

A dictionary representing additional HTML attributes to add to the primary element (e.g. {"onclick": "run_me();"}).

classes
str

Additional classes to add to the primary HTML element (e.g. "example-class another-class").

refresh_interval
int

The refresh interval for the runtime and status fields in milliseconds. Default is 5000.

Example

```
# CONTROLLER
from tethys_apps.sdk.gizmos import JobsTable

jobs_table_options = JobsTable(
                                jobs=jobs,
                                column_fields=('id', 'name', 'description', 'creation_time', 'exe
                                hover=True,
                                striped=False,
                                bordered=False,
                                condensed=False,
                                results_url='app_name:results_controller',
                              )

# TEMPLATE

{% gizmo jobs_table jobs_table_options %}
```

Persistent Stores API

Last Updated: January 19, 2016

The Persistent Store API streamlines the use of SQL databases in Tethys apps. Using this API, you can provision up to 5 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. register a new persistent store in the *app configuration file*,
2. create a data model to define the table structure of the database,
3. write a persistent store initialization function, and
4. 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 Registration

Registering new *persistent stores* is accomplished by adding the `persistent_stores()` method to your *app class*, which is located in your *app configuration file* (`app.py`). This method should return a list or tuple of `PersistentStore` objects. For example:

```
from tethys_sdk.base import TethysAppBase, url_map_maker
from tethys_sdk.stores import PersistentStore


class MyFirstApp(TethysAppBase):
    """
    Tethys App Class for My First App.
    """
    ...

    def persistent_stores(self):
        """
        Add one or more persistent stores
        """
        stores = (PersistentStore(name='example_db',
                                  initializer='my_first_app.init_stores.init_example_db'
                  ),
        )

        return stores
```

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'`).

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 `Persistent Store` object to the tuple that is returned by the `persistent_stores()` method.

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
from sqlalchemy.orm import sessionmaker

from .app import MyFirstApp

# DB Engine, sessionmaker, and base
engine = MyFirstApp.get_persistent_store_engine('example_db')
SessionMaker = sessionmaker(bind=engine)
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_engine()` 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 Registration, you need only use the name you provided during registration to retrieve the engine using `get_persistent_store_engine()`.

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:

```python
from .model import engine, SessionMaker, Base, StreamGage

def init_example_db(first_time):
    """
    An example persistent store initializer function
    """
    # Create tables
    Base.metadata.create_all(engine)

    # Initial data
    if first_time:
        # Make session
        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, three methods were added to the app class that allow apps to create persistent stores at run time, list existing persistent stores, and check if a given persistent store exists. See the API documentation below for details.

API Documentation

classmethod `TethysAppBase.`**`get_persistent_store_engine`**`(`*persistent_store_name*`)`

Creates an SQLAlchemy engine object for the app and persistent store given.

Parameters **persistent_store_name** (*string*) – Name of the persistent store for which to retrieve the engine.

Returns An SQLAlchemy engine object for the persistent store requested.

Return type object

Example:

```
from .app import MyFirstApp

engine = MyFirstApp.get_persistent_store_engine('example_db')
```

`TethysAppBase.`**`persistent_stores`**`()`

Define this method to register persistent store databases for your app. You may define up to 5 persistent stores for an app.

Returns A list or tuple of `PersistentStore` objects. A persistent store database will be created for each object returned.

Return type iterable

Example:

```
from tethys_sdk.stores import PersistentStore

def persistent_stores(self):
    """
    Example persistent_stores method.
    """

    stores = (PersistentStore(name='example_db',
                              initializer='init_stores:init_example_db',
                              spatial=True
              ),
    )

    return stores
```

classmethod `TethysAppBase.`**`create_persistent_store`**`(`*persistent_store_name*, *spatial=False*`)`

Creates a new persistent store database for this app.

Parameters

- **persistent_store_name** (*string*) – Name of the persistent store that will be created.
- **spatial** (*bool*) – Enable spatial extension on the database being created.

Returns True if successful.

Return type bool

Example:

```
from .app import MyFirstApp

result = MyFirstApp.create_persistent_store('example_db')
```

```
if result:
    engine = MyFirstApp.get_persistent_store_engine('example_db')
```

classmethod `TethysAppBase.`**`list_persistent_stores`**`()`

Returns a list of existing persistent stores for this app.

Returns A list of persistent store names.

Return type list

Example:

```
from .app import MyFirstApp

persistent_stores = MyFirstApp.list_persistent_stores()
```

classmethod `TethysAppBase.`**`persistent_store_exists`**`(`*persistent_store_name*`)`

Returns True if a persistent store with the given name exists for this app.

Parameters **persistent_store_name** (*string*) – Name of the persistent store that will be created.

Returns True if persistent store exists.

Return type bool

Example:

```
from .app import MyFirstApp

result = MyFirstApp.persistent_store_exists('example_db')

if result:
    engine = MyFirstApp.get_persistent_store_engine('example_db')
```

class `tethys_sdk.stores.`**`PersistentStore`**`(`*name*, *initializer*, *spatial=False*, *postgis=False*`)`

An object that stores the registration data for a Tethys Persistent Store.

Parameters

- **name** (*string*) – The name of the persistent store.
- **initializer** (*string*) – Path to the initialization function for the persistent store. Use dot-notation with a colon delineating the function (e.g.: "foo.bar:function").
- **spatial** (*bool, optional*) – PostGIS spatial extension will be enabled on the persistent store if True. Defaults to False.
- **postgis** (*bool, deprecated*) – PostGIS spatial extension will be enabled on the persistent store if True. Defaults to False. Deprecated, use spatial instead.

Spatial Persistent Stores API

Last Updated: November 24, 2014

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.

Register Spatial Persistent Store

Registering spatially enabled persistent stores follows the same process as registering normal persistent stores. The only difference is that you will set the `spatial` attribute of the `PersistentStore` object to `True`:

```
from tethys_sdk.base import TethysAppBase, url_map_maker
from tethys_sdk.stores import PersistentStore


class MyFirstApp(TethysAppBase):
    """
    Tethys App Class for My First App.
    """
    ...

    def persistent_stores(self):
        """
        Add one or more persistent stores
        """
        stores = (PersistentStore(name='spatial_db',
                                  initializer='my_first_app.init_stores.init_spatial_db',
                                  spatial=True
                  ),
        )

        return stores
```

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. For this, Tethys Platform provides the GeoAlchemy2, which extends SQLAlchemy to support spatial columns and database functions. A data model that uses a `geometry` column type to store the points for stream gages may look like this:

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

from geoalchemy2 import Geometry

from .app import MyFirstApp

# Spatial DB Engine, sessiomaker, and base
spatial_engine = MyFirstApp.get_persistent_store_engine('spatial_db')
SpatialSessionMaker = sessionmaker(bind=spatial_engine)
SpatialBase = declarative_base()

# SQLAlchemy ORM definition for the spatial_stream_gages table
class SpatialStreamGage(SpatialBase):
    """
    Example of SQLAlchemy spatial DB model
    """
    __tablename__ = 'spatial_stream_gages'
    # Columns
    id = Column(Integer, primary_key=True)
    value = Column(Integer)
    geom = Column(Geometry('POINT'))

    def __init__(self, latitude, longitude, value):
        """
        Constructor for a gage
        """
        self.geom = '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 "geom". Notice that the constructor (`__init__.py`) takes the `latitude` and `longitude` provided and sets the value of the `geom` column to a string with a special format called Well Known Text. This is a common pattern when working with GeoAlchemy2 columns.

Important: This article only briefly introduces the concepts of working with GeoAlchemy2. It is highly recommended that you complete the GeoAlchemy ORM tutorial.

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 .model import spatial_engine, SpatialSessionMaker, SpatialBase, SpatialStreamGage

def init_spatial_db(first_time):
    """
    An example persistent store initializer function
    """
    # Create tables
    SpatialBase.metadata.create_all(spatial_engine)

    # Initial data
    if first_time:
        # Make session
        session = SpatialSessionMaker()

        # 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: August 5, 2015

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. CKAN is currently they only supported dataset service.

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

`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)

`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

`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)

`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)

`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)

`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)

`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

`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)

`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

`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)

`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.
- **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.
- **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.
- **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.
- **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.

`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.

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

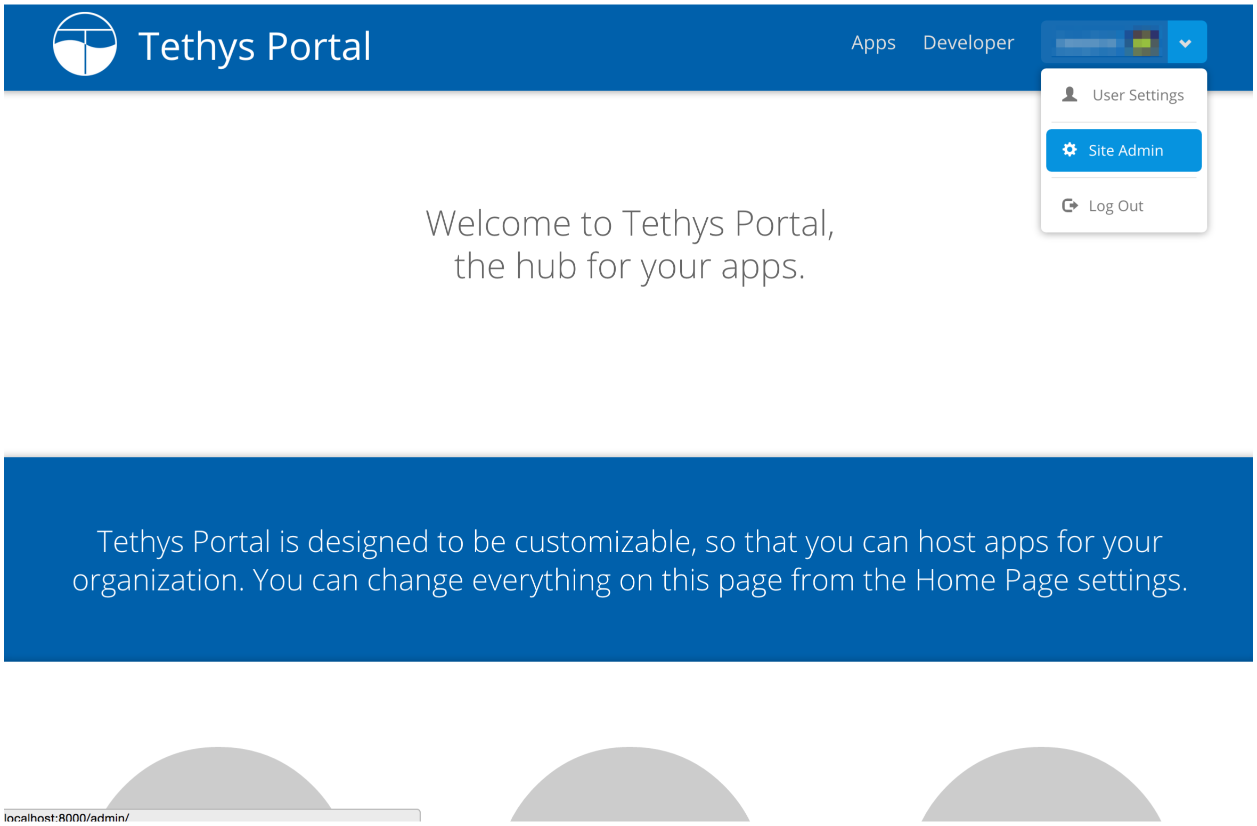
- **resource_id** (*string*) – The id of the resource that will be updated.
- **url** (*string, optional*) – URL of the resource that will be added to the dataset.
- **file** (*string, optional*) – Absolute path to a file to upload for the resource.
- **console** (*bool, optional*) – Pretty print the 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.

Register New Dataset Service

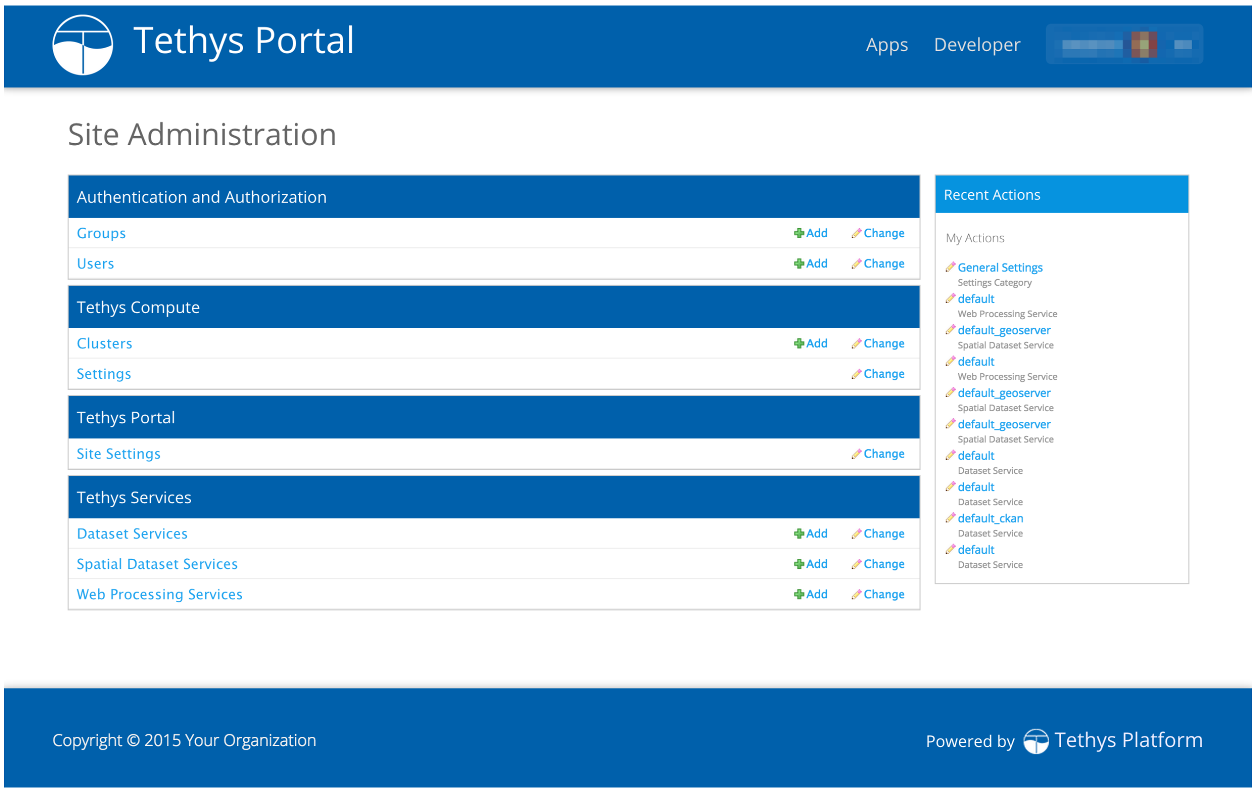
Registering new dataset services is performed through the System Admin Settings.

1. Login to your Tethys Platform instance as an administrator.
2. Select “Site Admin” from the user drop down menu.
3. Select “Dataset Services” from the “Tethys Services” section.



Welcome to Tethys Portal,
the hub for your apps.

Tethys Portal is designed to be customizable, so that you can host apps for your organization. You can change everything on this page from the Home Page settings.



Site Administration

Authentication and Authorization		
Groups	Add	Change
Users	Add	Change
Tethys Compute		
Clusters	Add	Change
Settings		Change
Tethys Portal		
Site Settings		Change
Tethys Services		
Dataset Services	Add	Change
Spatial Dataset Services	Add	Change
Web Processing Services	Add	Change

Recent Actions

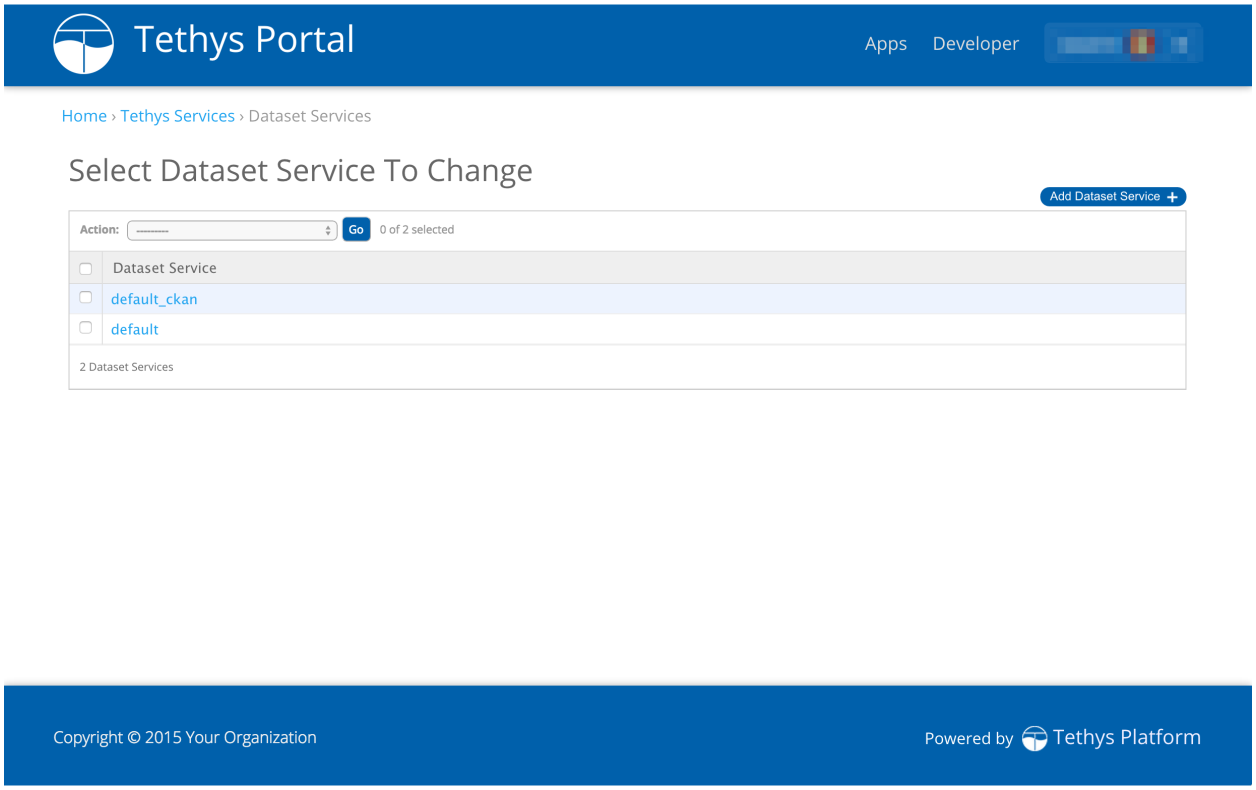
My Actions

- General Settings
  Settings Category
- default
  Web Processing Service
- default_geoserver
  Spatial Dataset Service
- default
  Web Processing Service
- default_geoserver
  Spatial Dataset Service
- default_geoserver
  Spatial Dataset Service
- default
  Dataset Service
- default
  Dataset Service
- default_ckan
  Dataset Service
- default
  Dataset Service

Copyright © 2015 Your Organization

Powered by Tethys Platform

4. Select an existing Dataset Service configuration from the list to edit it OR click on the "Add Dataset Service" button to create a new one.


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

Select Dataset Service To Change

Dataset Service
default_ckan
default

2 Dataset Services

5. Give the Dataset Service configuration a name, select an appropriate engine, and specify the endpoint. The name must be unique, because it is used to retrieve the Dataset Service connection object. The endpoint is a URL pointing to the Dataset Service API. Example endpoints for several different types of Dataset Services are shown below:

   ```
   # CKAN Endpoint URL
   http://www.example.com/api/3/action
   ```

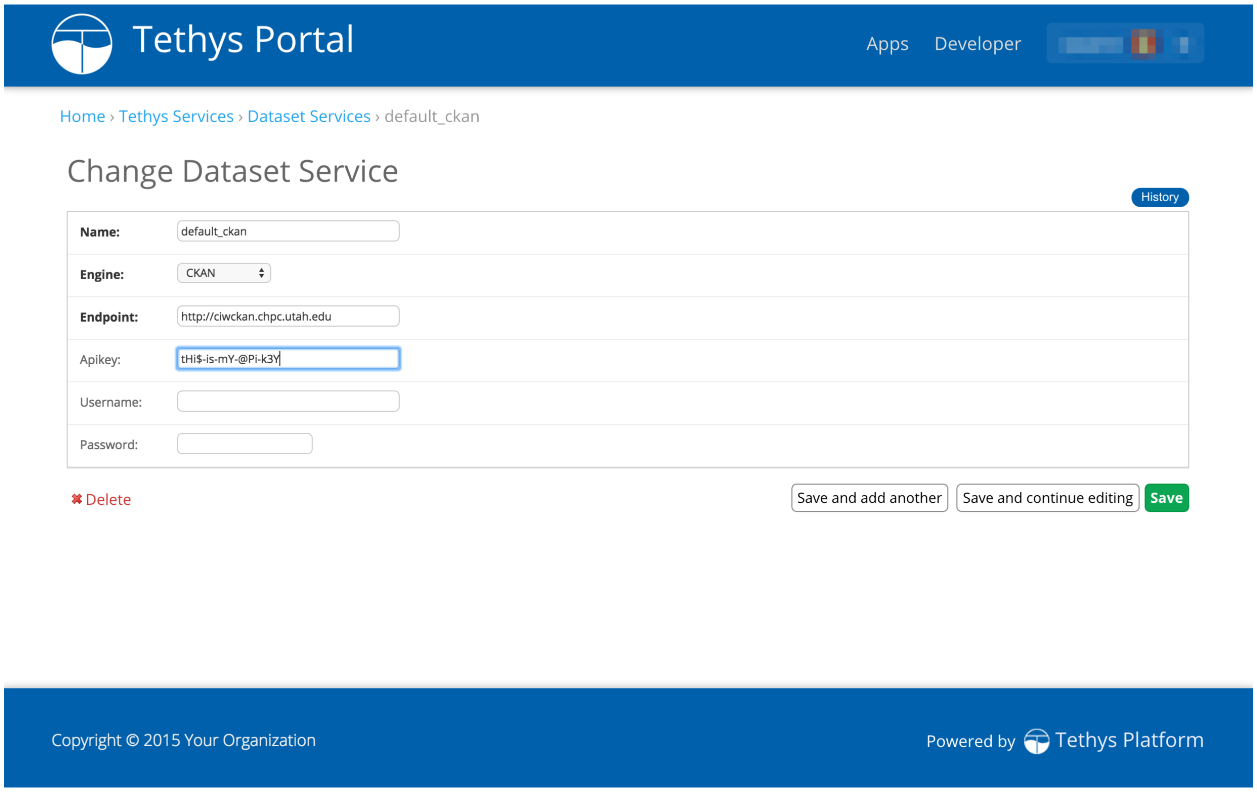
   If authentication is required, specify either the API key or the username and password.

   Note: 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 on which you have an account and browse to your user profile page. The API key will be listed there. Depending on the CKAN instance and the dataset, you may have full read-write access or you may have read-only access.

   When you are done, the form should look similar to this:

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

Note: Prior to version Tethys Platform 1.1.0, it was possible to register dataset services using a mechanism in the *app configuration file*. This mechanism has been deprecated due to security concerns.


Tethys Portal
Apps
Developer
Home › Tethys Services › Dataset Services › default_ckan
History
Delete
Save and add another
Save and continue editing
Save
Copyright © 2015 Your Organization
Powered by Tethys Platform

Change Dataset Service

Name:	default_ckan
Engine:	CKAN
Endpoint:	http://ciwckan.chpc.utah.edu
Apikey:	tHi$-is-mY-@Pi-k3Y
Username:	
Password:	

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

The Dataset Services API provides a convenience function for working with *dataset services* called `get_dataset_engine`. To retrieve and engine for a sitewide configuration, call `get_dataset_engine` with the name of the configuration:

```
from tethys_sdk.services import get_dataset_engine

dataset_engine = get_dataset_engine(name='example')
```

It will return the first service with a matching name or raise an exception if the service cannot be found with the given name. Alternatively, you may retrieve a list of all the dataset engine objects that are registered using the `list_dataset_engines` function:

```
from tethys_sdk.services import list_dataset_engines

dataset_engines = list_dataset_engines()
```

You can also create a `DatasetEngine` object directly without using the convenience function. This can be useful if you want to vary the credentials for dataset access frequently (e.g.: using user specific credentials). Simply import it and instantiate it with valid credentials:

```
from tethys_dataset_services.engines import CkanDatasetEngine

dataset_engine = CkanDatasetEngine(endpoint='http://www.example.com/api/3/action', apikey='a-R3llY-n1
```

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 tethys_sdk.services import get_dataset_engine

dataset_engine = get_dataset_engine(name='example')

result = dataset_engine.list_datasets()

if result['success']:
    dataset_list = result['result']

    for each 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

@ensure_oauth2('hydroshare')
def my_controller(request):
    """
    This is an example controller that uses the HydroShare API.
    """
    engine = get_dataset_engine('hydroshare', request=request)

    response = engine.list_datasets()

    context = {}

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

Spatial Dataset Services API

Last Updated: July 17, 2015

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). 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.

Powered by GeoServer

GeoServer powers the Spatial Dataset Service capabilities of Tethys Platform. It 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).

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

`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)

`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)

`SpatialDatasetEngine.`**`list_resources`**`()`
List all resources available from the spatial dataset service.

Returns Response dictionary

Return type (dict)

`SpatialDatasetEngine.`**`list_layers`**`()`
List all layers available from the spatial dataset service.

Returns Response dictionary

Return type (dict)

Update Methods

`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)

`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

`SpatialDatasetEngine.`**`delete_resource`**`(`*resource_id*`)`
Delete a resource.

Parameters **resource_id** (*string*) – Identifier of the resource to delete.

Returns Response dictionary

Return type (dict)

`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=True*)

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.
- **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.
- **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_file*, *coverage_type*, *coverage_name=None*, *overwrite=False*, *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.

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.
- **coverage_file** (*string*) – 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.
- **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'.
- **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.
- **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.

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.

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')

`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.
- **bounds** (*iterable*) – A tuple representing the bounding box of the layer group (e.g.: ('-74.02722', '-73.907005', '40.684221', '40.878178', 'EPSG:4326'))
- **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)

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.

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.
- **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.
- **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’)

# Without table

response = engine.create_postgis_resource(store_id=’workspace:store_name’, host=’localhost’, port=‘5432’, database=’database_name’, user=’user’, password=’pass’)

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.

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.
- **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.
- **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").
- **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** (*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").
- **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]+$'), ('simtime', '0:00:00', '^[w:]+$'))
- **debug** (*bool, optional*) – Pretty print the response dictionary to the console for debugging. Defaults to False.

Returns Response dictionary

Return type (dict)

Examples

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

- **create_style** (*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).
- **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*, *purge=False*, *recurse=False*, *debug=False*)

Delete a layer.

Parameters

- **layer_id** (*string*) – Identifier of the layer to delete.
- **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=None*, *purge=False*, *recurse=False*, *debug=False*)

Delete a resource.

Parameters

- **resource_id** (*string*) – Identifier of the resource to delete.
- **store** (*string, optional*) – Delete resource from this 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).
- **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*, *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").
- **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").
- **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=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").
- **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").
- **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='172.17.42.1'*)
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.
- **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.

- **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.
- **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` (*with_properties=False*, *debug=False*)
List the names of all styles available from the spatial dataset service.

Parameters

- **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)

`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.
- **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'])

`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'])

`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").
- **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

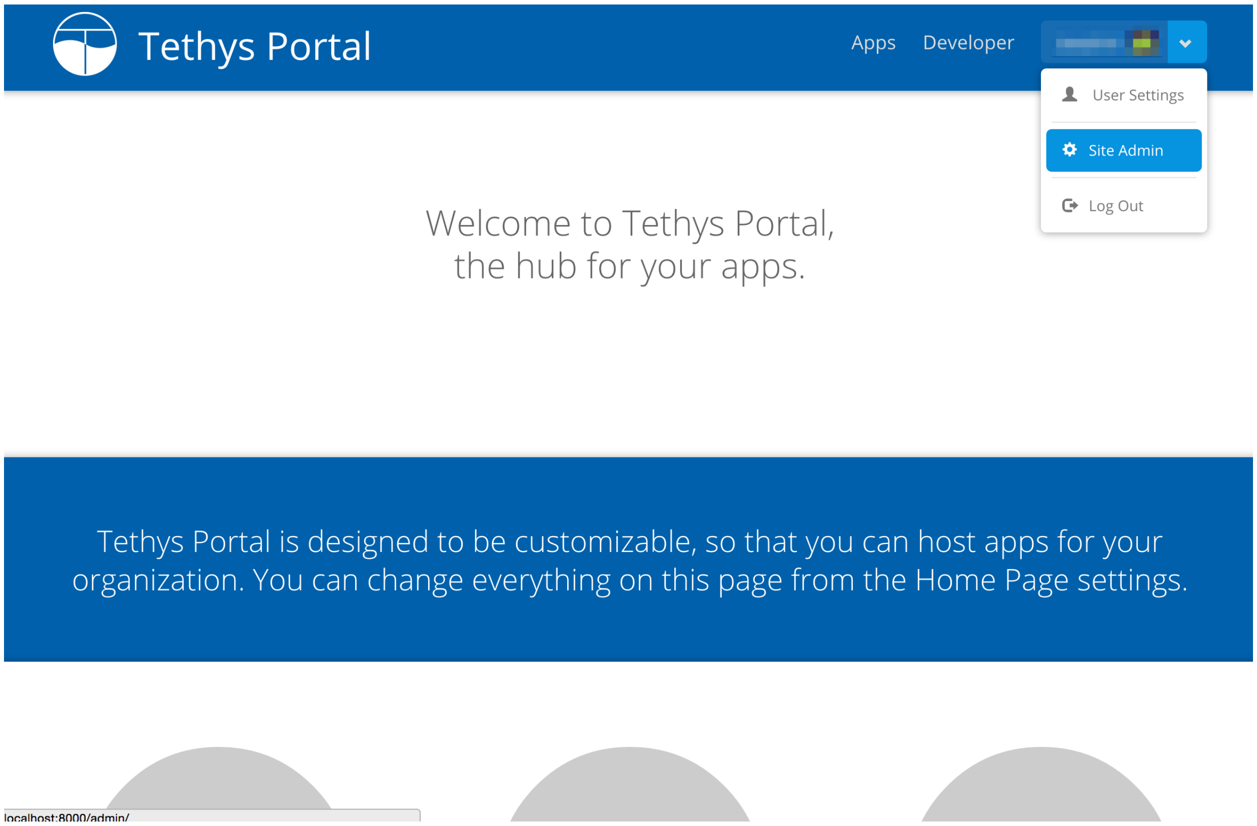
response = engine.update_resource(resource_id='workspace:resource_name', enabled=False, title='New Title')

`validate`()
Validate the GeoServer spatial dataset engine. Will throw and error if not valid.

Register New Spatial Dataset Services

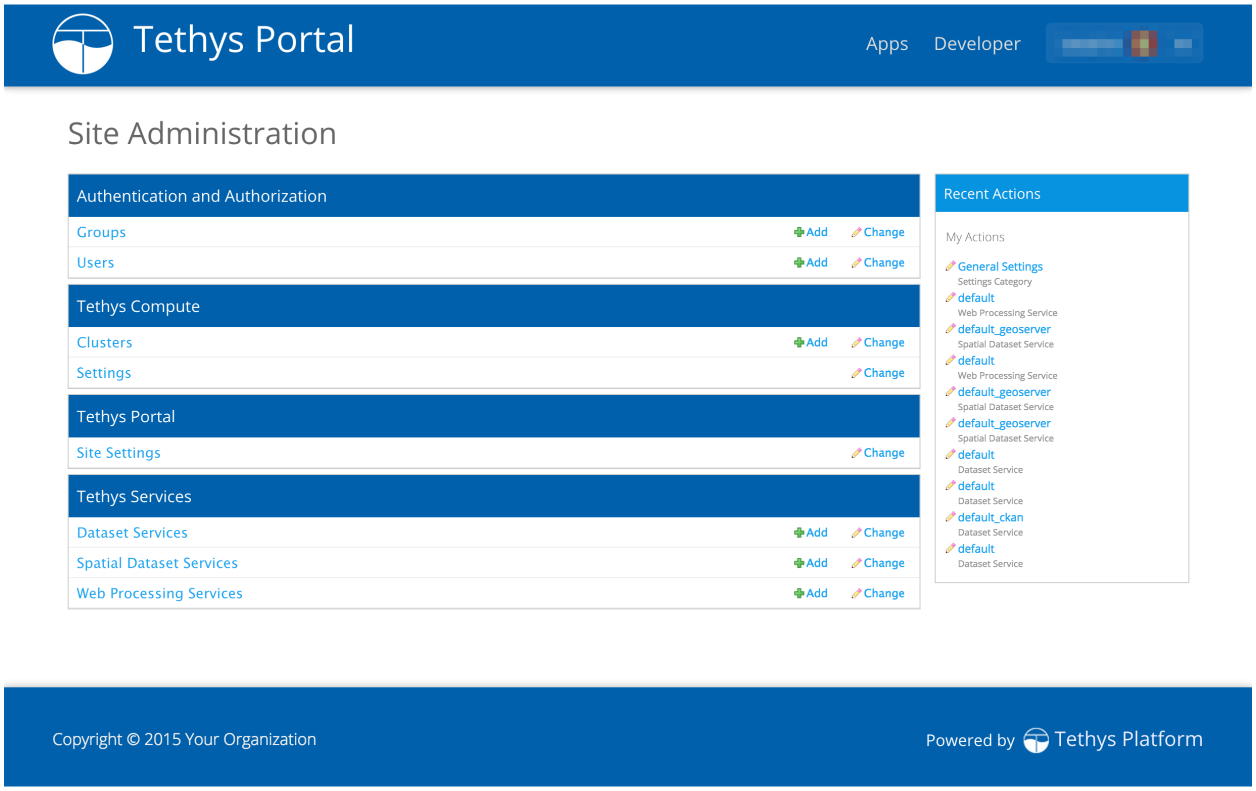
Registering new spatial dataset services is performed using the System Admin Settings.

1. Login to your Tethys Platform instance as an administrator.
2. Select "Site Admin" from the user drop down menu.
3. Select "Spatial Dataset Services" from the "Tethys Services" section.
4. Select an existing Spatial Dataset Service configuration from the list to edit it OR click on the "Add Spatial Dataset Service" button to create a new one.
5. Give the Spatial Dataset Service configuration a name, select an appropriate engine, specify the endpoint, and provide a username and password. The name of the configuration must be unique, because it is used to retrieve the Spatial Dataset Service connection object. The endpoint is a URL pointing to the GeoServer REST endpoint. This endpoint can be for **any** GeoServer. If you want to use the built-in GeoServer installation, you can obtain the endpoint by running `tethys docker ip` in a terminal:



Welcome to Tethys Portal, the hub for your apps.

Tethys Portal is designed to be customizable, so that you can host apps for your organization. You can change everything on this page from the Home Page settings.



Site Administration

Authentication and Authorization		
Groups	Add	Change
Users	Add	Change
Tethys Compute		
Clusters	Add	Change
Settings		Change
Tethys Portal		
Site Settings		Change
Tethys Services		
Dataset Services	Add	Change
Spatial Dataset Services	Add	Change
Web Processing Services	Add	Change

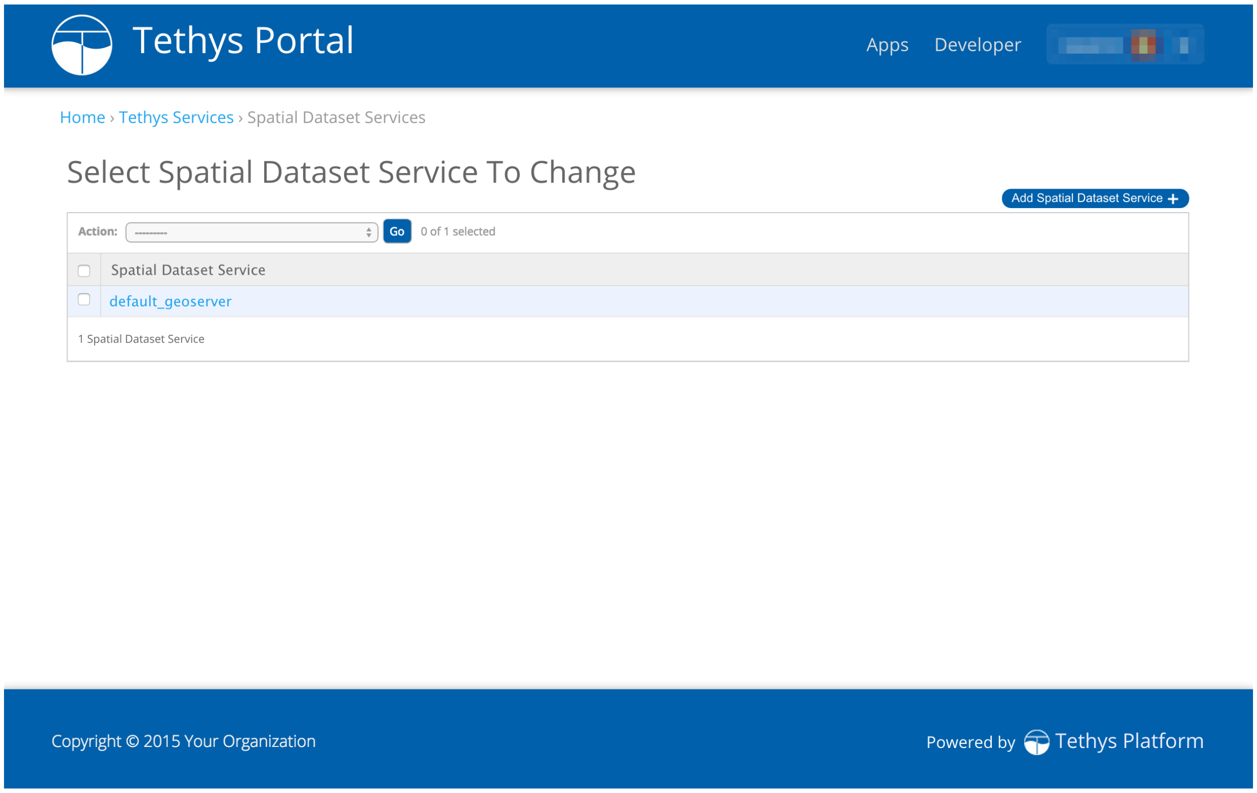
Recent Actions

My Actions

- General Settings — Settings Category
- default — Web Processing Service
- default_geoserver — Spatial Dataset Service
- default — Web Processing Service
- default_geoserver — Spatial Dataset Service
- default_geoserver — Spatial Dataset Service
- default — Dataset Service
- default — Dataset Service
- default_ckan — Dataset Service
- default — Dataset Service

Copyright © 2015 Your Organization

Powered by Tethys Platform


Tethys Portal
Apps
Developer
Home › Tethys Services › Spatial Dataset Services

Select Spatial Dataset Service To Change

Add Spatial Dataset Service +
Action:
Go
0 of 1 selected

	Spatial Dataset Service
☐	default_geoserver

1 Spatial Dataset Service

Copyright © 2015 Your Organization
Powered by Tethys Platform

```
$ tethys docker ip
...

GeoServer:
  Host: localhost
  Port: 8181
  Endpoint: http://localhost:8181/geoserver/rest

...
```

When you are done, your Spatial Dataset Service configuration should look similar to this:

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

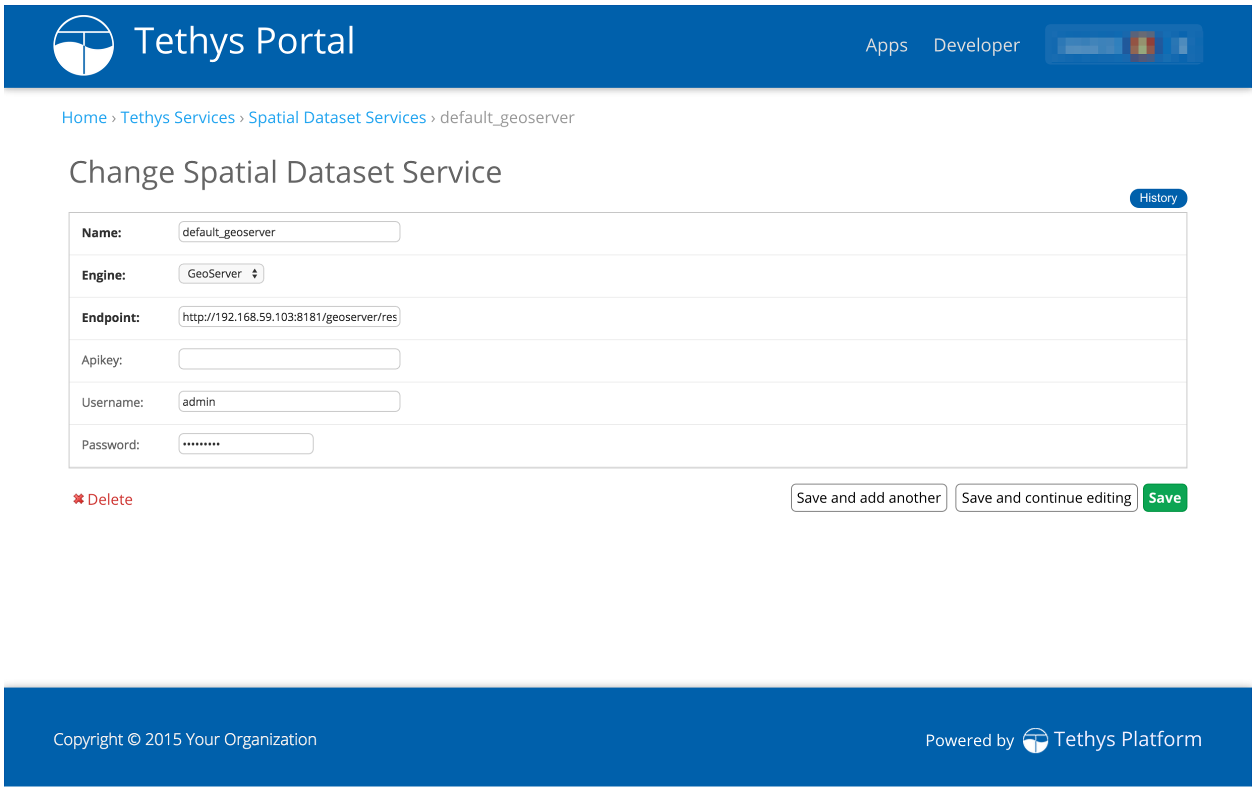
Note: Prior to version Tethys Platform 1.1.0, it was possible to register spatial dataset services using a mechanism in the *app configuration file*. This mechanism has been deprecated due to security concerns.

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

The Spatial Dataset Services API provides a convenience function called `get_spatial_dataset_engine`. To retrieve and engine for a sitewide configuration, call `get_spatial_dataset_engine` with the name of the configuration:



[redacted]

Home › Tethys Services › Spatial Dataset Services › default_geoserver

Change Spatial Dataset Service

Name:	default_geoserver
Engine:	GeoServer
Endpoint:	http://192.168.59.103:8181/geoserver/res
Apikey:	
Username:	admin
Password:	•••••••••

Copyright © 2015 Your Organization

```
from tethys_sdk.services import get_spatial_dataset_engine

dataset_engine = get_dataset_engine(name='example')
```

It will return the first service with a matching name or raise an exception if the service cannot be found with the given name. Alternatively, you may retrieve a list of all the spatial dataset engine objects that are registered using the `list_spatial_dataset_engines` function:

```
from tethys_sdk.services import list_spatial_dataset_engines

dataset_engines = list_spatial_dataset_engines()
```

You can also create a `SpatialDatasetEngine` object directly without using the convenience function. This can be useful if you want to vary the credentials for dataset access frequently (e.g.: using user specific credentials). Simply import it and instantiate it with valid credentials:

```
from tethys_dataset_services.engines import GeoServerSpatialDatasetEngine

spatial_dataset_engine = GeoServerSpatialDatasetEngine(endpoint='http://www.example.com/geoserver/res
```

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 tethys_sdk.services import get_spatial_dataset_engine

# First get an engine
engine = get_spatial_dataset_engine(name='example')

# 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_bas

# 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=
                    'geotiff8': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&reques
                    'geptiff': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&request
                    'gif': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&request=Get
                    'jpeg': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&request=Ge
                    'kml': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&request=Get
                    'kmz': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&request=Get
                    'openlayers': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&requ
                    'pdf': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&request=Get
                    'png': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&request=Get
                    'png8': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&request=Ge
                    'svg': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&request=Get
                    'tiff': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&request=Ge
                    'tiff8': 'http://localhost:8181/geoserver/wms?service=WMS&version=1.1.0&request=G
 '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 13, 2015

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 handled. 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. Refer to the *Installation* documentation to learn how to install Tethys Platform with 52 North WPS enabled.

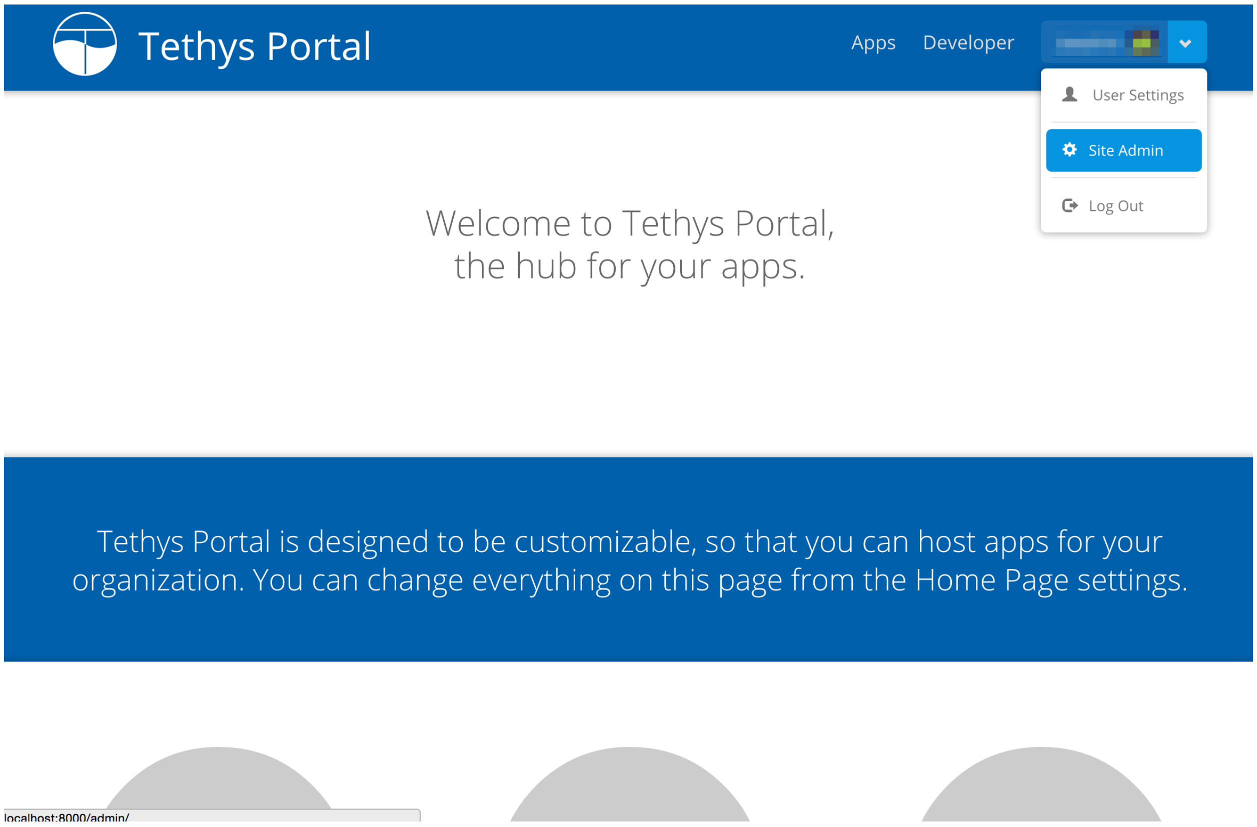
Configuring WPS Services

Before you can start using WPS services in your apps, you will need link your Tethys Platform to a valid WPS. This can be done either at a sitewide level or at an app specific level. When a WPS is configured at the sitewide level, all apps that are installed on that Tethys Platform instance will be able to access the WPS. When installed at an app specific level, the WPS will only be accessible to the app that it is linked to. The following sections will describe how to configure a WPS to be used at both of these levels.

Register New WPS Service

Sitewide configuration is performed using the System Admin Settings.

1. Login to your Tethys Platform instance as an administrator.
2. Select "Site Admin" from the user drop down menu.


Tethys Portal
Apps
Developer
User Settings
Site Admin
Log Out
Welcome to Tethys Portal,
the hub for your apps.
Tethys Portal is designed to be customizable, so that you can host apps for your organization. You can change everything on this page from the Home Page settings.
localhost:8000/admin/

3. Select "Web Processing Services" from the "Tethys Services" section.

4. Select an existing Web Processing Service configuration from the list to edit it OR click on the "Add Web Processing Service" button to create a new one.

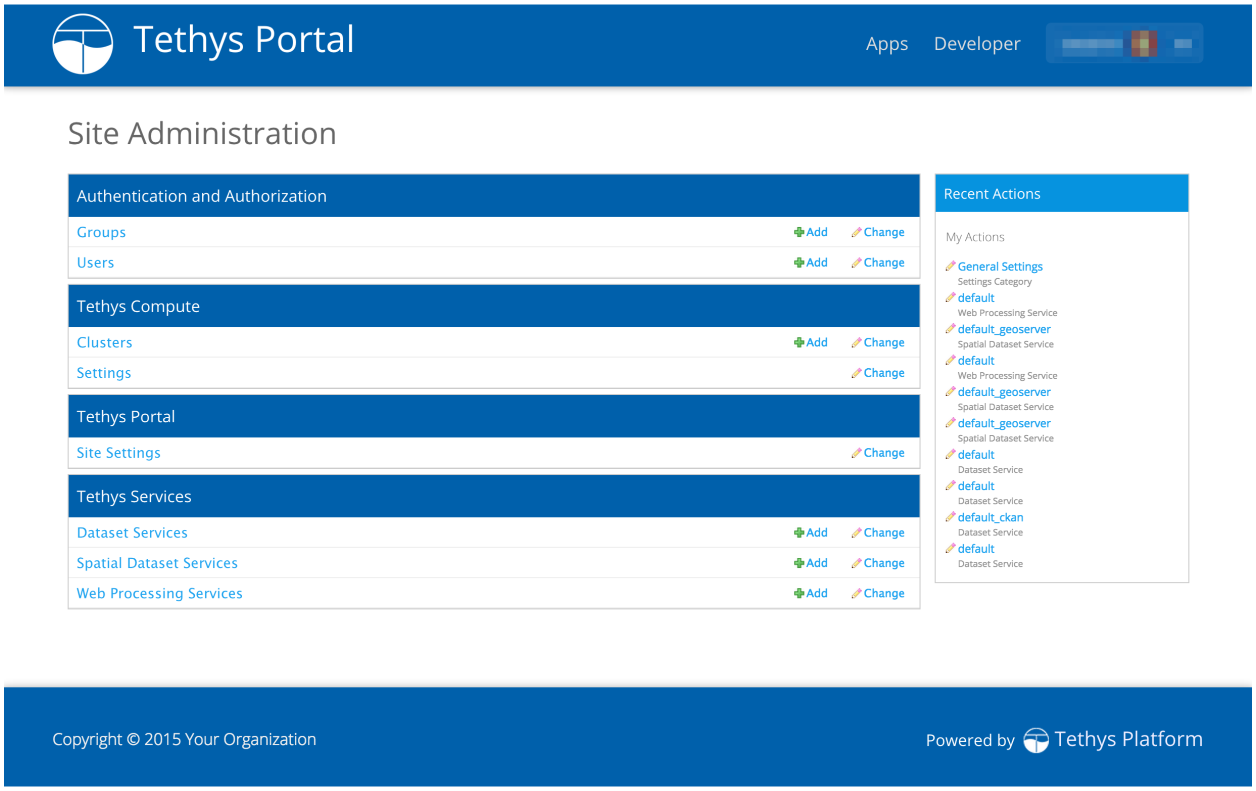
5. Give the Web Processing Service configuration a name and specify the endpoint. The name must be unique, because it is used to connect to the WPS. The endpoint is a URL pointing to the WPS. For example, the endpoint for the 52 North WPS demo server would be:

   ```
   http://geoprocessing.demo.52north.org:8080/wps/WebProcessingService
   ```

If authentication is required, specify the username and password.

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

Note: Prior to version Tethys Platform 1.1.0, it was possible to register WPS services using a mechanism in the *app configuration file*. This mechanism has been deprecated due to security concerns.



Tethys Portal — Apps — Developer

Site Administration

Authentication and Authorization		
Groups	Add	Change
Users	Add	Change

Tethys Compute		
Clusters	Add	Change
Settings		Change

Tethys Portal		
Site Settings		Change

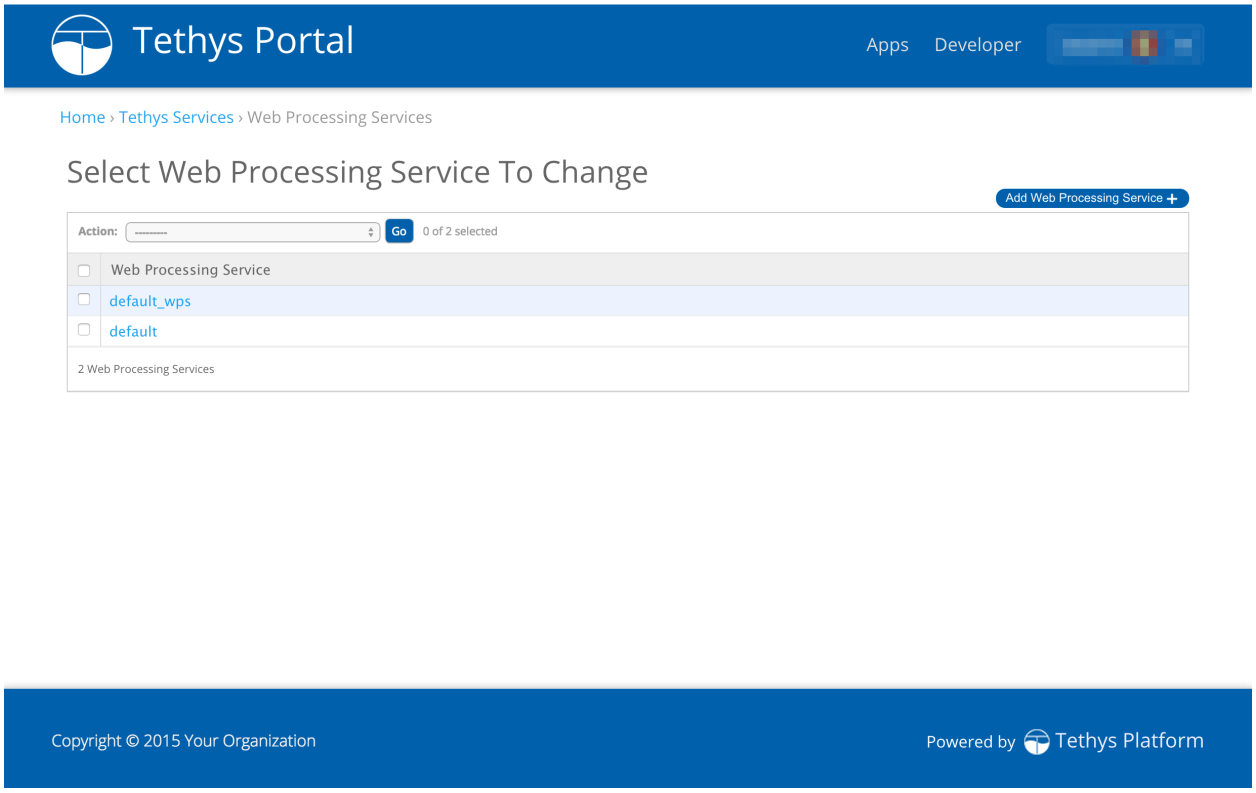
Tethys Services		
Dataset Services	Add	Change
Spatial Dataset Services	Add	Change
Web Processing Services	Add	Change

Recent Actions

My Actions

- General Settings
  Settings Category
- default
  Web Processing Service
- default_geoserver
  Spatial Dataset Service
- default
  Web Processing Service
- default_geoserver
  Spatial Dataset Service
- default_geoserver
  Spatial Dataset Service
- default
  Dataset Service
- default
  Dataset Service
- default_ckan
  Dataset Service
- default
  Dataset Service

Copyright © 2015 Your Organization — Powered by Tethys Platform



Tethys Portal — Apps — Developer

Home › Tethys Services › Web Processing Services

Select Web Processing Service To Change

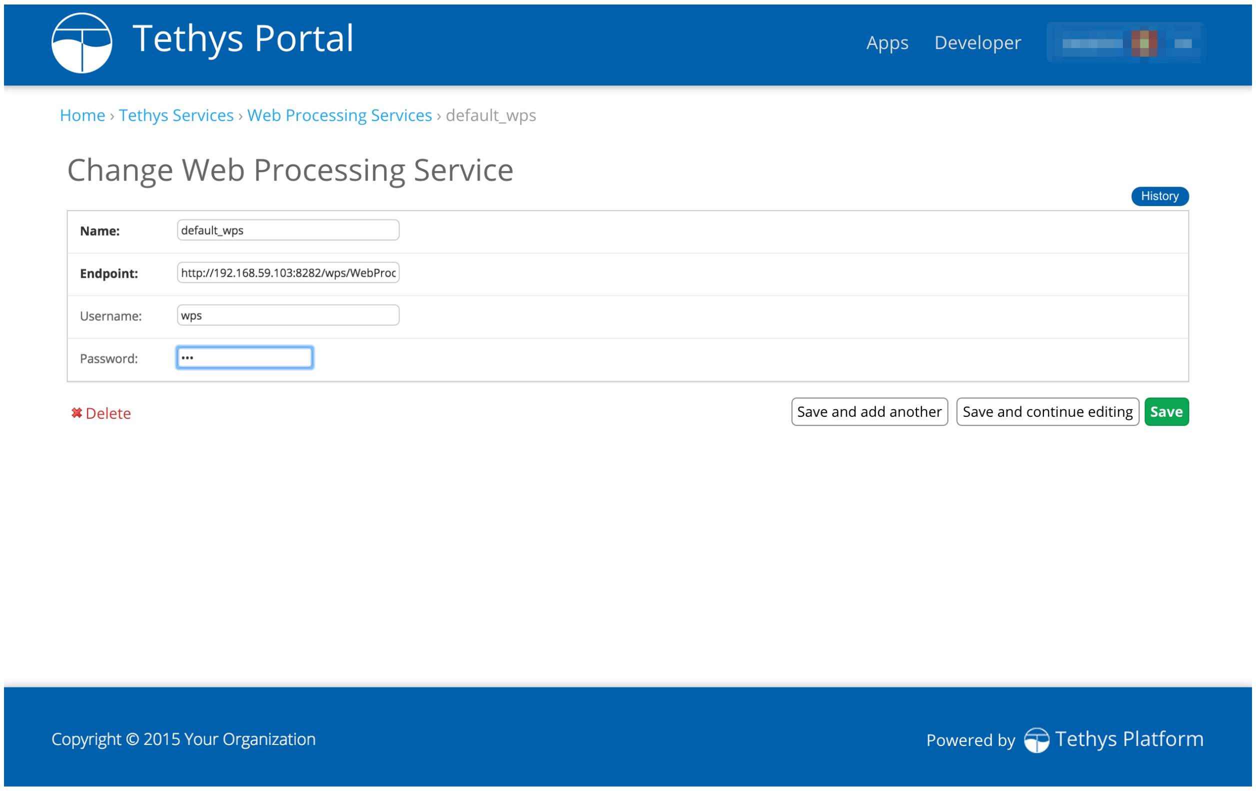
Add Web Processing Service +

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

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

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. The Web Processing Service API provides a convenience function for retrieving `owslib.wps.WebProcessingService` engine objects called `get_wps_service_engine`. Basic usage involves calling the function with the name of the WPS service that you wish to use. For example:

```
from tethys_sdk.services import get_wps_service_engine

wps_engine = get_wps_service_engine(name='example')
```

Alternatively, you may retrieve a list of all the dataset engine objects that are registered using the `list_wps_service_engines` function:

```
from tethys_sdk.services import list_wps_service_engines

wps_engines = list_wps_service_engines()
```

You can also 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.: using user specific credentials).

```
from owslib.wps import WebProcessingService

wps_engine = WebProcessingService('http://www.example.com/wps/WebProcessingService', verbose=False, s
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), (-10
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:

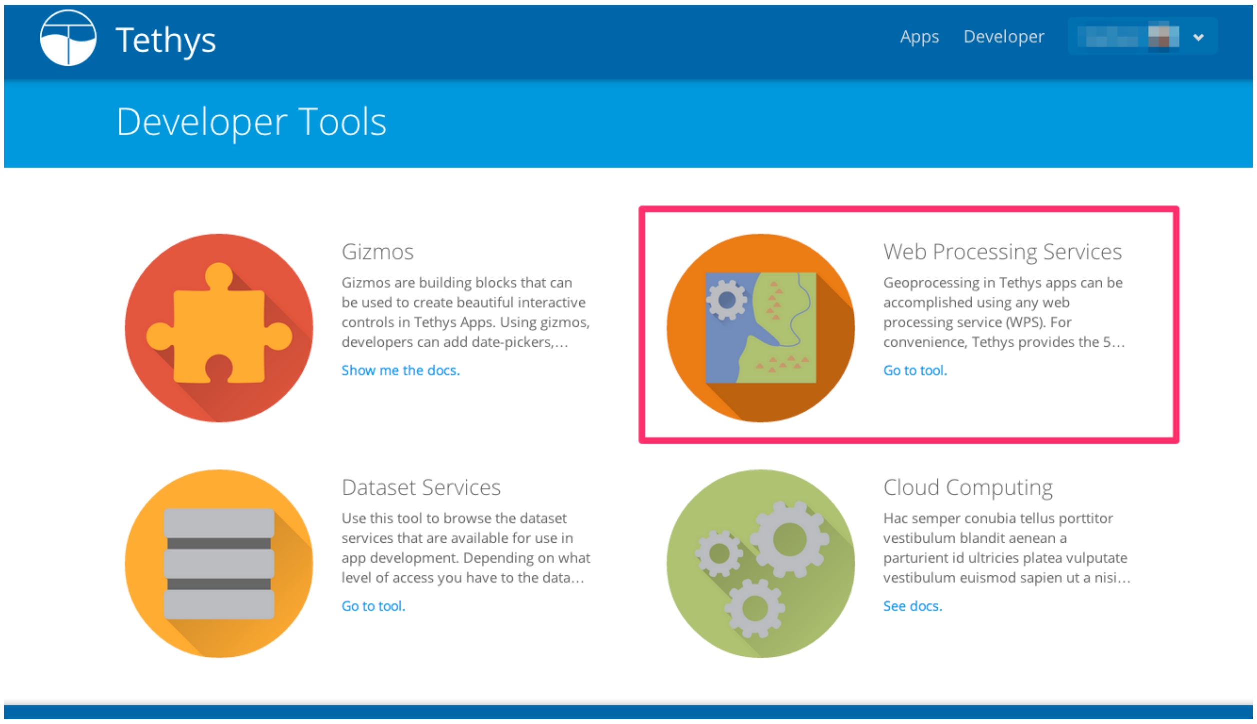
1. Browse to the Developer Tools page of your Tethys Platform by selecting the “Developer” link from the menu at the top of the page.
2. Select the tool titled “Web Processing Services”.
3. Select a WPS service from the list of services that are linked with your Tethys Instance. If no WPS services are 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.

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.



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,...

Show me the docs.

Web Processing Services

Geoprocessing in Tethys apps can be accomplished using any web processing service (WPS). For convenience, Tethys provides the 5...

Go to tool.

Dataset Services

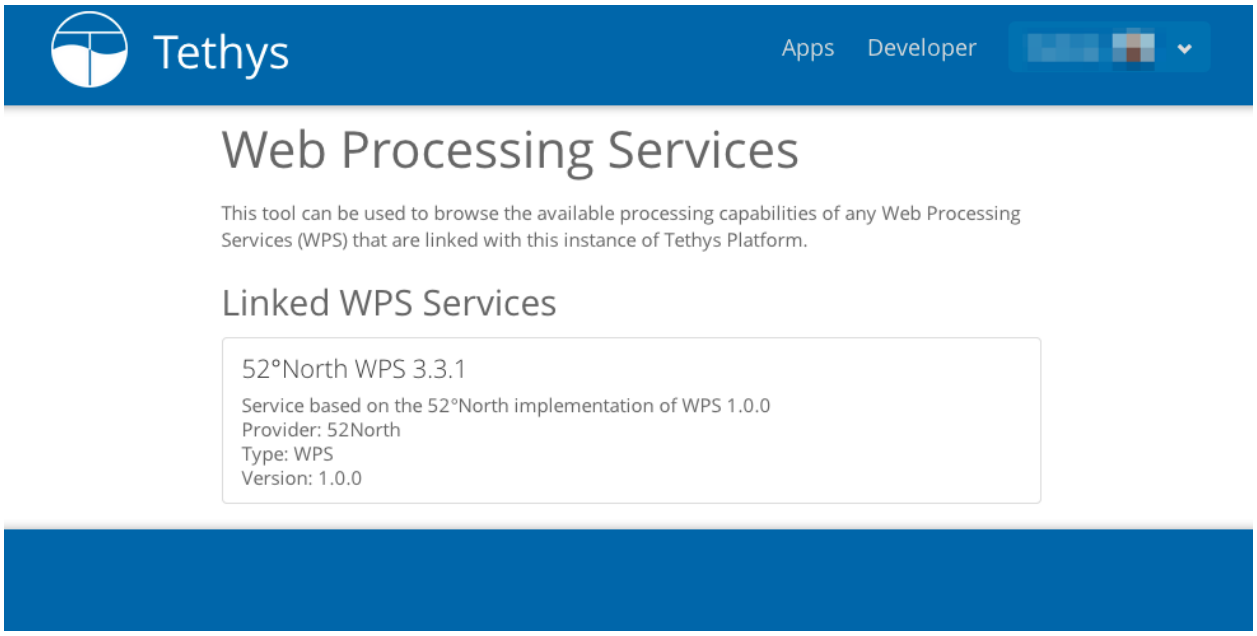
Use this tool to browse the dataset services that are available for use in app development. Depending on what level of access you have to the data...

Go to tool.

Cloud Computing

Hac semper conubia tellus porttitor vestibulum blandit aenean a parturient id ultricies platea vulputate vestibulum euismod sapien ut a nisi...

See docs.



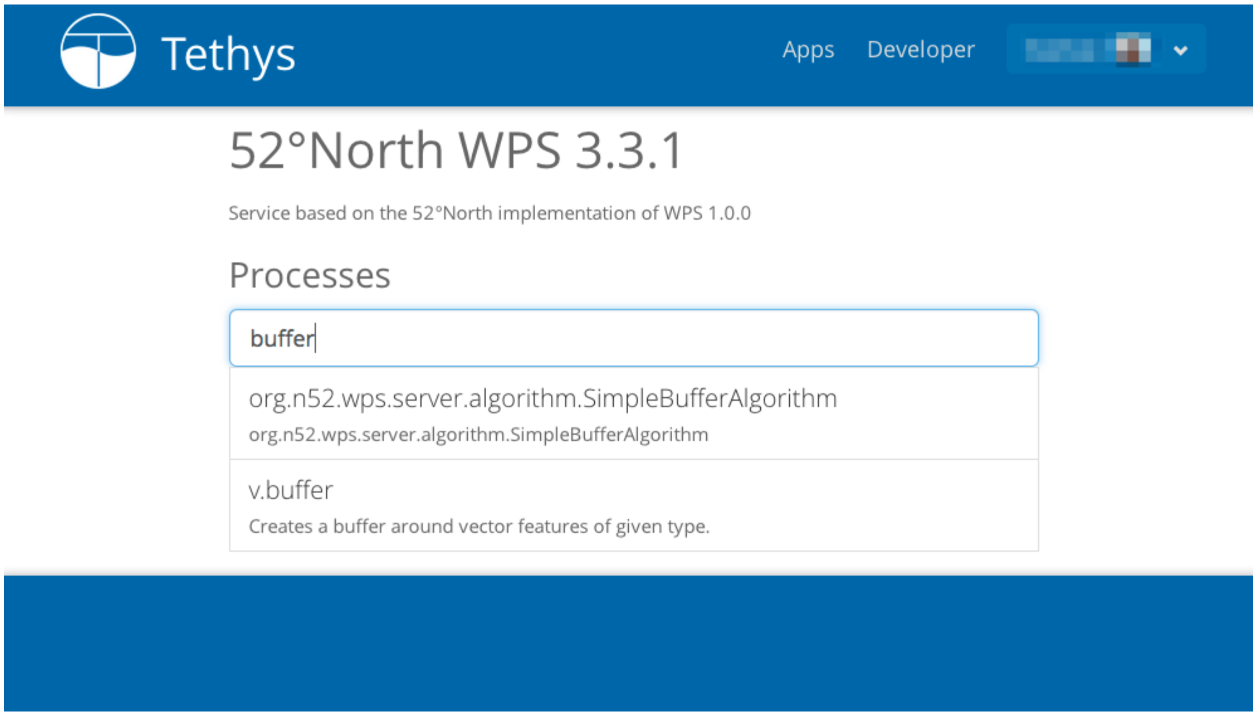
Web Processing Services

This tool can be used to browse the available processing capabilities of any Web Processing Services (WPS) that are linked with this instance of Tethys Platform.

Linked WPS Services

52°North WPS 3.3.1

Service based on the 52°North implementation of WPS 1.0.0
Provider: 52North
Type: WPS
Version: 1.0.0



Tethys Apps Developer [redacted]

52°North WPS 3.3.1

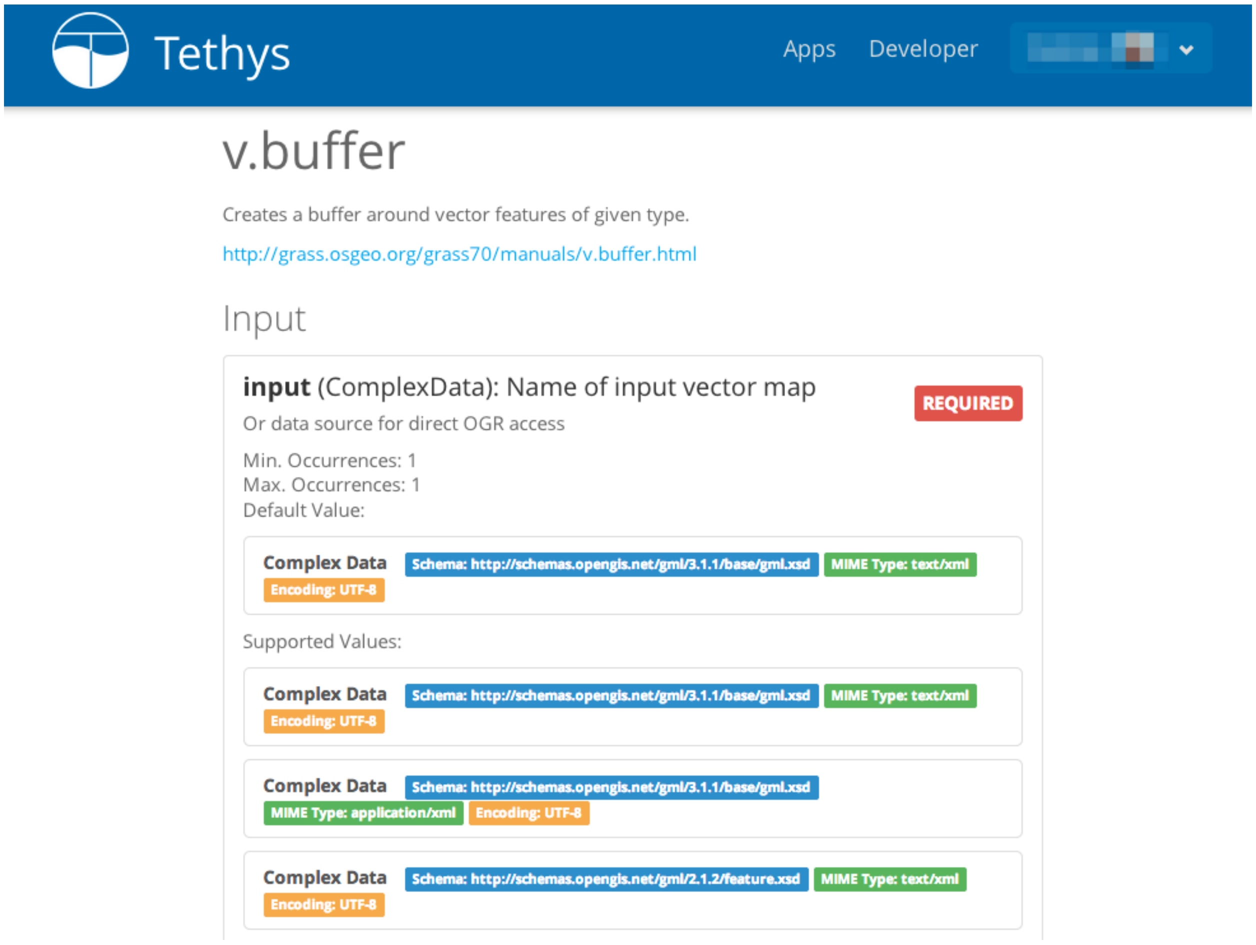
Service based on the 52°North implementation of WPS 1.0.0

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 [redacted]

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

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_ke
```

API Documentation

`tethys_sdk.compute.`**`get_cluster_manager`**(*config_file=None*, *cache=False*)

Factory for ClusterManager class that attempts to load AWS credentials from the TethysCluster config file. Returns a ClusterManager object if successful `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.

Jobs API

Last Updated: September 12, 2016

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, they 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 first step to creating a job is to define a job template. A job template is like a blue print that describes certain key characteristics about the job, such as the job type and where the job will be run. The job manager uses a job template to create a new job that has all of the attributes that were defined in the template. Once a job has been created from a template it can then be customized with any additional attributes that are needed for that specific job.

The Jobs API supports various types of jobs (see Job Types).

Note: The real power of the jobs API comes when it is combined 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. This is done through the *Condor Job Type* or the *Condor Workflow Job Type*.

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.

Defining Job Templates

To create jobs in an app you first need to define job templates. A job template specifies the type of job, and also defines all of the static attributes of the job that will be the same for all instances of that template. These attributes often include the names of the executable, input files, and output files. Job templates are defined in a method on the `TethysAppBase` subclass in `app.py` module. The following code sample shows how this is done:

```
from tethys_sdk.jobs import CondorJobTemplate, CondorJobDescription
from tethys_sdk.compute import list_schedulers

def job_templates(cls):
    """
    Example job_templates method.
    """
    my_scheduler = list_schedulers()[0]

    my_job_description = CondorJobDescription(condorpy_template_name='vanilla_transfer_files',
                                              remote_input_files=('$(APP_WORKSPACE)/my_script.py', '$
                                              executable='my_script.py',
                                              transfer_input_files=('../input_1', '../input_2'),
                                              transfer_output_files=('example_output1', example_outpu
                                              )

    job_templates = (CondorJobTemplate(name='example',
                                       job_description=my_job_description,
                                       scheduler=my_scheduler,
                                      ),
                    )

    return job_templates
```

Note: To define job templates the appropriate template class and any supporting classes must be imported from `tethys_sdk.jobs`. In this case the template class *CondorJobTemplate* is imported along with the supporting class *CondorJobDescription*.

There is a corresponding job template class for every job type. In this example the *CondorJobTemplate* class is used, which corresponds to the *Condor Job Type*. For a list of all possible job types see Job Types.

When instantiating any job template class there is a required `name` parameter, which is used used to identify the template to the job manager (see Using the Job Manager in your App). The template class for each job type may have additional required and/or optional parameters. In the above example the *job_description* and the *scheduler* parameters are required because the the *CondorJobTemplate* class is being instantiated. Job template classes may also support setting job attributes as parameters in the template. See the Job Types documentation for a list of acceptable parameters for the template class of each job type.

Warning: The generic template class `JobTemplate` allong with the dictionary `JOB_TYPES` have been used to define job templates in the past but are being deprecated in favor of job-type specific templates classes (e.g. *CondorJobTemplate* or *CondorWorkflowTemplate*).

Job Types

The Jobs API is designed to support multiple job types. Each job type provides a different framework and environment for executing jobs. To create a job of a particular job type, you must first create a job template from the template class corresponding to that job type (see Defining Job Templates). After the job template for the job type you want has been instantiated you can create a new job instance using the job manager (see Using the Job Manager in your App).

Once you have a newly created job from the job manager you can then customize the job by setting job attributes. 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 (see Workspaces).
- `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).

Note: Job template classes may support passing in job attributes as additional arguments. See the documentation for each job type for a list of acceptable parameters for each template class add if additional arguments are supported.

Specific job types may define additional attributes. The following job types are available.

Basic Job Type **Last Updated:** March 29, 2016

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. The only required parameter for the *BasicJobTemplate* class is `name`, but it also supports passing in other job attributes as additional arguments.

Setting up a BasicJobTemplate

```
from tethys_sdk.jobs import BasicJobTemplate

def job_templates(cls):
    """
    Example job_templates method with a BasicJob type.
    """

    job_templates = (BasicJobTemplate(name='example',
                                      description='This is a sample basic job. It can't actually comp
                                      extended_properties={'app_spcific_property': 'default_value',
                                                           'persistent_store_id': None,   # Will be de
                                                           }
                                      ),
                     )

    return job_templates
```

Creating and Customizing a Job To create a job call the `create_job` method on the job manager. The required parameters are `name`, `user` and `template_name`. 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='example', desc
```

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_sdk.jobs.`**`BasicJobTemplate`**(*name*, *parameters=None*)

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.

class `tethys_compute.models.`**`BasicJob`**(**args*, ***kwargs*)

Basic job type. Use this class as a model for subclassing TethysJob

Condor Job Type **Last Updated:** March 29, 2016

Setting up a CondorJobTemplate

```
from tethys_sdk.jobs import CondorJobTemplate, CondorJobDescription
from tethys_sdk.compute import list_schedulers

def job_templates(cls):
    """
    Example job_templates method.
    """
    my_scheduler = list_schedulers()[0]

    my_job_description = CondorJobDescription(condorpy_template_name='vanilla_transfer_files',
                                              remote_input_files=('$(APP_WORKSPACE)/my_script.py', '$
                                              executable='my_script.py',
                                              transfer_input_files=('../input_1', '../input_2'),
                                              transfer_output_files=('example_output1', example_outpu
                                              )

    job_templates = (CondorJobTemplate(name='example',
                                       job_description=my_job_description,
                                       scheduler=my_scheduler,
                                      ),
                    )

    return job_templates
```

Creating and Customizing a Job To create a job call the `create_job` method on the job manager. The required parameters are `name`, `user` and `template_name`. Any other job attributes can also be passed in as *kwargs*.

```
# create a new job
job = job_manager.create_job(name='job_name', user=request.user, template_name='example', descriptio

# customize the job using methods provided by the job type
job.set_attribute('arguments', 'input_2')

# 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_sdk.jobs.`**`CondorJobTemplate`**(*name*, *parameters=None*, *job_description=None*, *scheduler=None*, ***kwargs*)

A subclass of the JobTemplate with the `type` argument set to CondorJob.

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.
- **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.

class `tethys_compute.models.`**`CondorJob`**(**args*, ***kwargs*)

CondorPy Job job type

Condor Workflow Job Type **Last Updated:** March 29, 2016

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.

Setting up a CondorWorkflowTemplate Creating a *CondorWorkflowTemplate* involves 3 steps:

1. Define job descriptions for each of the sub-jobs using *CondorJobDescription* (see `condor_job_description`).
2. Create the sub-jobs and define relationships using *CondorWorkflowJobTemplate*.
3. Create the *CondorWorkflowTemplate*.

Note: The *CondorWorkflowJobTemplate* is similar to a *CondorJobTemplate* in that it represents a single HTCondor job and requires a *CondorJobDescription* to define the attributes of that job. However, unlike a *CondorJobTemplate* a *CondorWorkflowJobTemplate* cannot be run independently; it can only be part of a *CondorWorkflowTemplate*. Also, note that the *CondorWorkflowJobTemplate* has a *parents* parameter, which is used to define relationships between jobs.

The following code sample demonstrates how to set up a *CondorWorkflowTemplate*:

```
Example job_templates method with a CondorWorkflow type.
"""

job_a_description = CondorJobDescription(condorpy_template_name='vanilla_transfer_files',
                                         remote_input_files=('$(APP_WORKSPACE)/my_script.py', '$(APP_
                                         executable='my_script.py',
                                         transfer_input_files=('../input_1', '../input_2'),
                                         transfer_output_files=('example_output1', example_output2),
                                         )
job_b_description = CondorJobDescription(condorpy_template_name='vanilla_transfer_files',
                                         remote_input_files=('$(APP_WORKSPACE)/my_script.py', '$(APP_
                                         executable='my_script.py',
                                         transfer_input_files=('../input_1', '../input_2'),
                                         transfer_output_files=('example_output1', example_output2),
                                         )
job_c_description = CondorJobDescription(condorpy_template_name='vanilla_transfer_files',
                                         remote_input_files=('$(APP_WORKSPACE)/my_script.py', '$(APP_
                                         executable='my_script.py',
                                         transfer_input_files=('../input_1', '../input_2'),
                                         transfer_output_files=('example_output1', example_output2),
                                         )
job_a = CondorWorkflowJobTemplate(name='JobA',
                                  job_description=job_a_description,
                                  )
job_b = CondorWorkflowJobTemplate(name='JobB',
                                  job_description=job_b_description,
                                  parents=[job_a]
                                  )
job_c = CondorWorkflowJobTemplate(name='JobC',
                                  job_description=job_c_description,
                                  parents=[job_b]
```

```
                                          )
job_templates = (CondorWorkflowTemplate(name='WorkflowABC',
                                         job_list=[job_a, job_b, job_c],
                                         scheduler=None,
                                         ),
                 )
```

If the you want to use the same job both as part of a workflow and as a stand alone job then use the same job description in setting up the *CondorJobTemplate* and the *CondorWorkflowJobTemplate*. This process is demonstrated below:

```
from tethys_sdk.jobs import CondorJobTemplate, CondorWorkflowTemplate, CondorWorkflowJobTemplate, Con
from tethys_sdk.compute import list_schedulers

def job_templates(cls):
    """
    Example job_templates method with a CondorWorkflow type.
    """

    reusable_job_a_description = CondorJobDescription(condorpy_template_name='vanilla_transfer_files
                                                      remote_input_files=('$(APP_WORKSPACE)/my_script
                                                      executable='my_script.py',
                                                      transfer_input_files=('../input_1', '../input_
                                                      transfer_output_files=('example_output1', examp
                                                      )
    job_b1_description = CondorJobDescription(condorpy_template_name='vanilla_transfer_files',
                                              remote_input_files=('$(APP_WORKSPACE)/my_script.py', '$
                                              executable='my_script.py',
                                              transfer_input_files=('../input_1', '../input_2'),
                                              transfer_output_files=('example_output1', example_outpu
                                              )
    job_b2_description = CondorJobDescription(condorpy_template_name='vanilla_transfer_files',
                                              remote_input_files=('$(APP_WORKSPACE)/my_script.py', '$
                                              executable='my_script.py',
                                              transfer_input_files=('../input_1', '../input_2'),
                                              transfer_output_files=('example_output1', example_outpu
                                              )
    job_c_description = CondorJobDescription(condorpy_template_name='vanilla_transfer_files',
                                             remote_input_files=('$(APP_WORKSPACE)/my_script.py', '$
                                             executable='my_script.py',
                                             transfer_input_files=('../input_1', '../input_2'),
                                             transfer_output_files=('example_output1', example_output
                                             )
    job_a = CondorWorkflowJobTemplate(name='JobA',
                                      job_description=reusable_job_a_description,
                                      )
    job_b1 = CondorWorkflowJobTemplate(name='JobB1',
                                       job_description=reusable_job_a_description,
                                       parents=[job_a]
                                       )
    job_b2 = CondorWorkflowJobTemplate(name='JobB2',
                                       job_description=reusable_job_a_description,
                                       parents=[job_a]
                                       )
    job_c = CondorWorkflowJobTemplate(name='JobC',
                                      job_description=reusable_job_a_description,
                                      parents=[job_b1, job_b2]
                                      )
    job_templates = (CondorWorkflowTemplate(name='DiamondWorkflow',
                                            job_list=[job_a, job_b1, job_b2, job_c],
                                            scheduler=None,
                                            ),
                     CondorJobTemplate(name='JobAStandAlone',
                                       job_description=reusable_job_a_description,
                                       scheduler=None,
                                       ),
                     )
```

Creating and Customizing a Job To create a job call the `create_job` method on the job manager. The required parameters are `name`, `user` and `template_name`. Any other job attributes can also be passed in as *kwargs*.

```
# create a new job
job = job_manager.create_job(name='job_name', user=request.user, template_name='example', descriptio

# customize the job using methods provided by the job type
job.set_attribute('arguments', 'input_2')

# 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_sdk.jobs.`**`CondorWorkflowTemplate`**(*name*, *parameters=None*, *jobs=None*, *max_jobs=None*, *config=None*, ***kwargs*)

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*)

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.

class `tethys_compute.models.`**`CondorWorkflow`**(**args*, ***kwargs*)

CondorPy Workflow job type

class `tethys_compute.models.`**`CondorWorkflowJobNode`**`(*args, **kwargs)`
CondorWorkflow JOB type node

Workspaces

Each job has it's own workspace, which by default is set to the user's workspace in the app where the job is created. However, the job may need to reference files that are in other workspaces. To make it easier to interact with workspaces in job templates, two special variables are defined: `$(APP_WORKSPACE)` and `$(USER_WORKSPACE)`. These two variables are resolved to absolute paths when the job is created. These variables can only be used in job templates. To access the app's workspace and the user's workspace when working with a job in other places in your app use the *Workspaces API*.

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 job call the `create_job` method on the job manager. The required parameters are `name`, `user` and `template_name`. Any other job attributes can also be passed in as *kwargs*.

```
# create a new job
job = job_manager.create_job(name='job_name', user=request.user, template_name='example', descriptio

# customize the job using methods provided by the job type
job.set_attribute('arguments', 'input_2')

# 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.

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*, ***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.
- **template_name** (*str*) – The name of the JobTemplate from which to create the job.
- ****kwargs** –

Returns A new job object of the type specified by the JobTemplate

`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_sdk.jobs.`**`JobTemplate`**(*name*, *type=None*, *parameters=None*)

A template from which to create a job.

Parameters

- **name** (*str*) – Name to refer to the template.
- **type** (*TethysJob*) – A subclass of the TethysJob base class. Use the JOB_TYPE dictionary for possible values.
- **parameters** (*dict*) – A dictionary of key-value pairs. Each Job type defines the possible parameters.

class `tethys_sdk.jobs.`**`BasicJobTemplate`**(*name*, *parameters=None*)

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.

class `tethys_sdk.jobs.`**`CondorJobTemplate`**(*name*, *parameters=None*, *job_description=None*, *scheduler=None*, ***kwargs*)

A subclass of the JobTemplate with the `type` argument set to CondorJob.

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.
- **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.

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 .app import MyFirstApp

def a_controller(request):
    """
    Example controller that uses get_app_workspace() method.
    """
    # Retrieve the workspace
    app_workspace = MyFirstApp.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 a 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 .app import MyFirstApp

def a_controller(request):
    """
    Example controller that uses get_user_workspace() method.
    """
    # Retrieve the workspace
    user_workspace = MyFirstApp.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`

str

The absolute path to the workspace directory. Cannot be overwritten.

`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)

# Clear all but specified files and directories
workspace.clear(exclude=['file1.txt', '/full/path/to/directory1', 'directory2', '/full/path/
```

`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
```

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:

```python
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'
    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`
str

Instance of a TethysAppBase object.

`handlers`
list[HandoffHandler]

A list of HandoffHandlers registered in the app.

`valid_handlers`
list[HandoffHandler]

A filtered list of only the valid HandoffHandlers.

`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.

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 *Manage Users and Permissions* documentation for more details.

`TethysAppBase.`**`permissions`**`()`

Use 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

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
```

Permission and Permission Group Objects

class `tethys_sdk.permissions.`**`Permission`**(*name*, *description*)

Defines an object that represents a permission for an app.

`name`

string

The code name for the permission. Only numbers, letters, and underscores allowed.

`description`

string

Short description of the permission for the admin interface.

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`

string

The name for the group. Only numbers, letters, and underscores allowed.

`permissions`

iterable

A list or tuple of Permission objects.

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',
    permissions=(create_projects, delete_projects)
)
```

Check Permission

Use the `has_permission` method to check whether the user of the current request has a permission.

static `permissions.`**`has_permission`**`(`*request*, *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.

static `permissions.`**`permission_required`**`(`**args*, ***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):
    """
    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 controller
@permission_required('create_projects', raise_exception=True)
def my_controller(request):
    """
    Example controller
    """
    ...
```

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 virtual 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.

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. Either "start", "syncdb", or "collectstatic".
  - *start*: Starts the Django development server. Wrapper for `manage.py runserver`.
  - *syncdb*: Initialize the database during installation. Wrapper for `manage.py syncdb`.
  - *collectstatic*: Link app 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*.
  - *superuser*: 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

# 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
```

uninstall <app>

Use this command to uninstall apps.

Arguments:

- **app**: Name the app to uninstall.

Examples:

```
# Uninstall my_first_app
$ tethys uninstall my_first_app
```

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
$ 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
```

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:

```python
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):
        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. In settings.py make sure that the tethys_default database user is set to tethys_super:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'tethys_default',
        'USER': 'tethys_super',
        'PASSWORD': 'pass',
        'HOST': '127.0.0.1',
        'PORT': '5435'
    }
}
```

4. **Enter app-level `tethys test` command.** `(tethys)$ tethys test -f tethys_apps.tethysapp.<app_name(required)>.<folder_name>.<file_name>.<class_name>.<` `[-c/C]` Where `-c` tracks code coverage and prints out a report in the terminal, and `-C` does opens the report as an interactive HTML page in your browser

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.`**`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

Tethys Portal

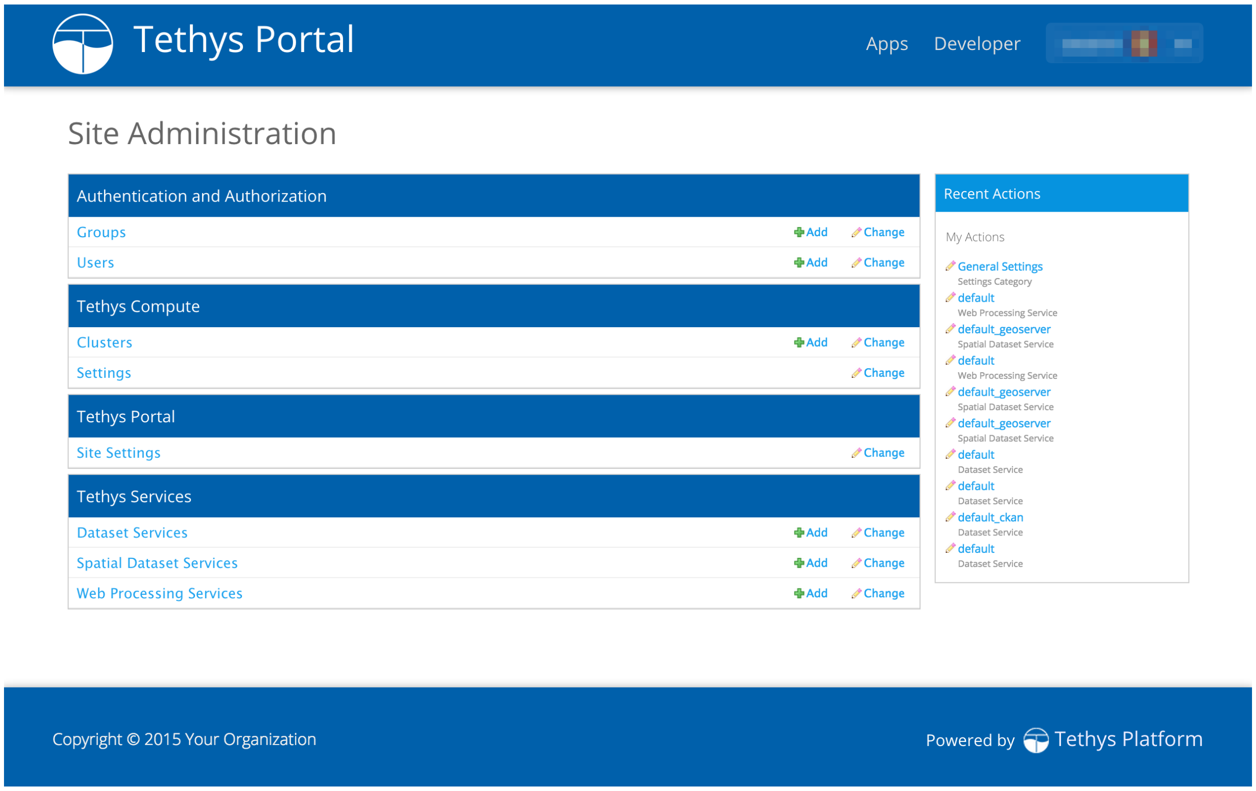
Last Updated: December 14, 2015

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.

Administrator Pages

Last Updated: August 4, 2015

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. 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.



Site Administration

Authentication and Authorization		
Groups	Add	Change
Users	Add	Change

Tethys Compute		
Clusters	Add	Change
Settings		Change

Tethys Portal		
Site Settings		Change

Tethys Services		
Dataset Services	Add	Change
Spatial Dataset Services	Add	Change
Web Processing Services	Add	Change

Recent Actions

My Actions

- General Settings — Settings Category
- default — Web Processing Service
- default_geoserver — Spatial Dataset Service
- default — Web Processing Service
- default_geoserver — Spatial Dataset Service
- default_geoserver — Spatial Dataset Service
- default — Dataset Service
- default — Dataset Service
- default_ckan — Dataset Service
- default — Dataset Service

Copyright © 2015 Your Organization

Powered by Tethys Platform

Figure 1. Administrator dashboard for Tethys Portal.

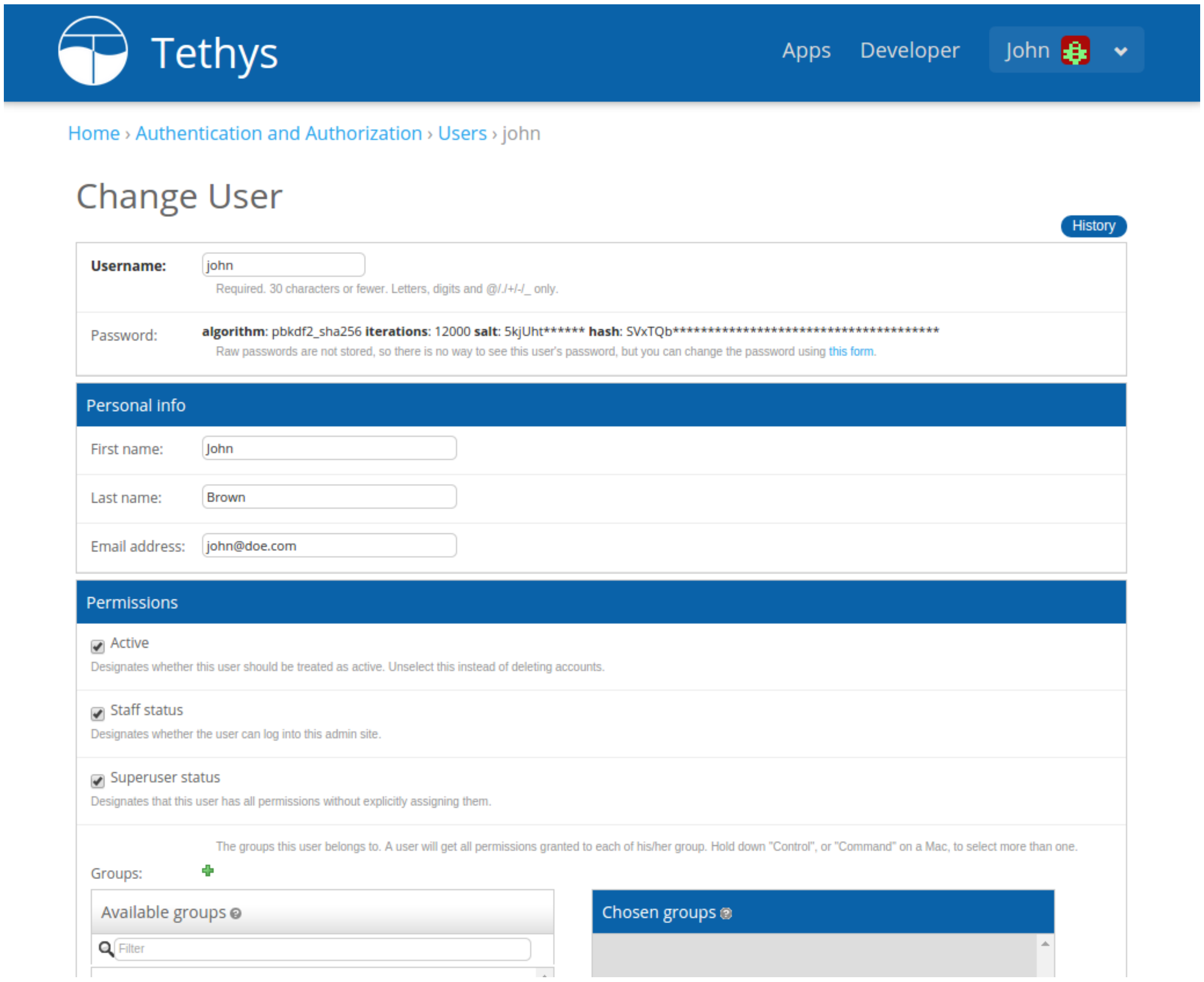
Note: If you did not create an administrator user during installation, run the following command in the terminal:

```
$ python /usr/lib/tethys/src/manage.py createsuperuser
```

Manage Users and Permissions

Permissions and users can be managed from the administrator dashboard using `Users` link under the `Authentication and Authorization` heading. Figure 4 shows an example of the user management page for a user named John.

Figure 4. User management for Tethys Portal.



Change User

Username: john

Required. 30 characters or fewer. Letters, digits and @/./+/-/_ only.

Password: **algorithm**: pbkdf2_sha256 **iterations**: 12000 **salt**: 5kjUht****** **hash**: SVxTQb**************************************

Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

Personal info

First name:	John
Last name:	Brown
Email address:	john@doe.com

Permissions

✓ Active

Designates whether this user should be treated as active. Unselect this instead of deleting accounts.

✓ Staff status

Designates whether the user can log into this admin site.

✓ Superuser status

Designates that this user has all permissions without explicitly assigning them.

Groups:

The groups this user belongs to. A user will get all permissions granted to each of his/her group. Hold down "Control", or "Command" on a Mac, to select more than one.

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.

Manage Tethys Services

The administrator pages provide a simple mechanism for linking to the other services of Tethys Platform. Use the `Spatial Dataset Services` link to connect your Tethys Portal to GeoServer, the `Dataset Services` link to connect to CKAN instances or HydroShare, or the `Web Processing Services` link to connect to WPS instances. For detailed instructions on how to perform each of these tasks, refer to the *Spatial Dataset Services API*, *Dataset Services API*, and *Web Processing Services API* documentation, respectively.

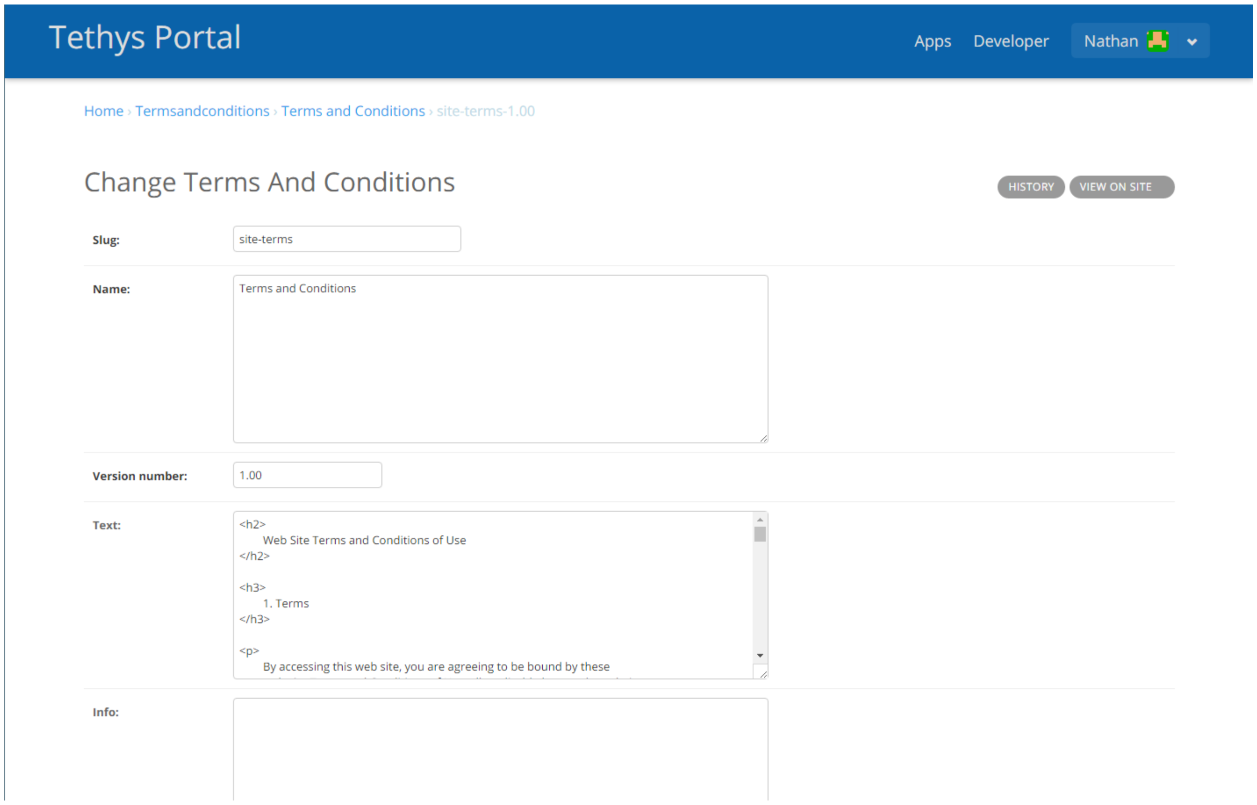
Manage Terms and Conditions

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 5). 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 5. Creating a new legal document using the terms and conditions feature.

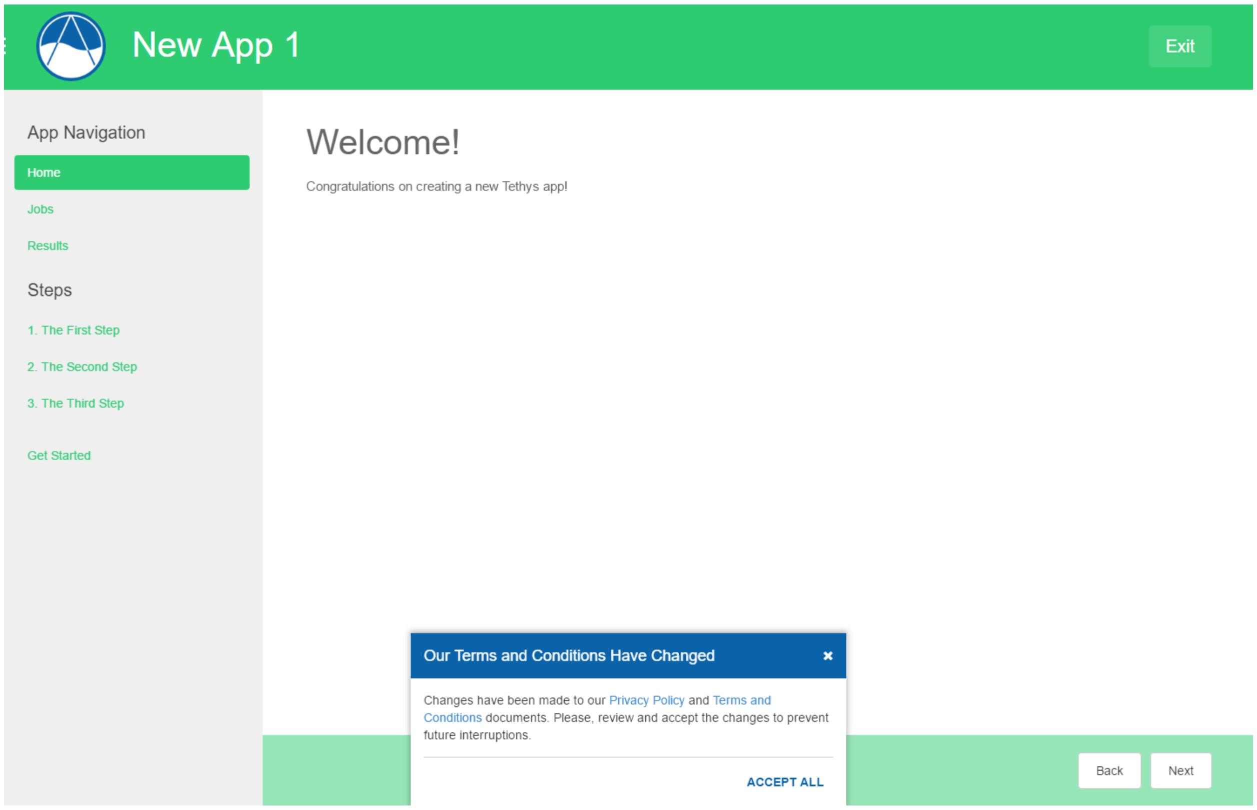
When a new document becomes active, users will be presented with a modal prompting them to review and accept the new terms and conditions (see Figure 6). The modal can be dismissed, but will reappear each time a page is refreshed until the user accepts the new versions of the legal documents.


Tethys Portal
Apps
Developer
Nathan
Home › Termsandconditions › Terms and Conditions › site-terms-1.00
HISTORY
VIEW ON SITE

Change Terms And Conditions

Slug:	site-terms
Name:	Terms and Conditions
Version number:	1.00
Text:	
Info:	

```
<h2>
    Web Site Terms and Conditions of Use
</h2>

<h3>
    1. Terms
</h3>

<p>
    By accessing this web site, you are agreeing to be bound by these
```



Welcome!

Congratulations on creating a new Tethys app!

Our Terms and Conditions Have Changed

Changes have been made to our Privacy Policy and Terms and Conditions documents. Please, review and accept the changes to prevent future interruptions.

ACCEPT ALL

Figure 6. Terms and conditions modal.

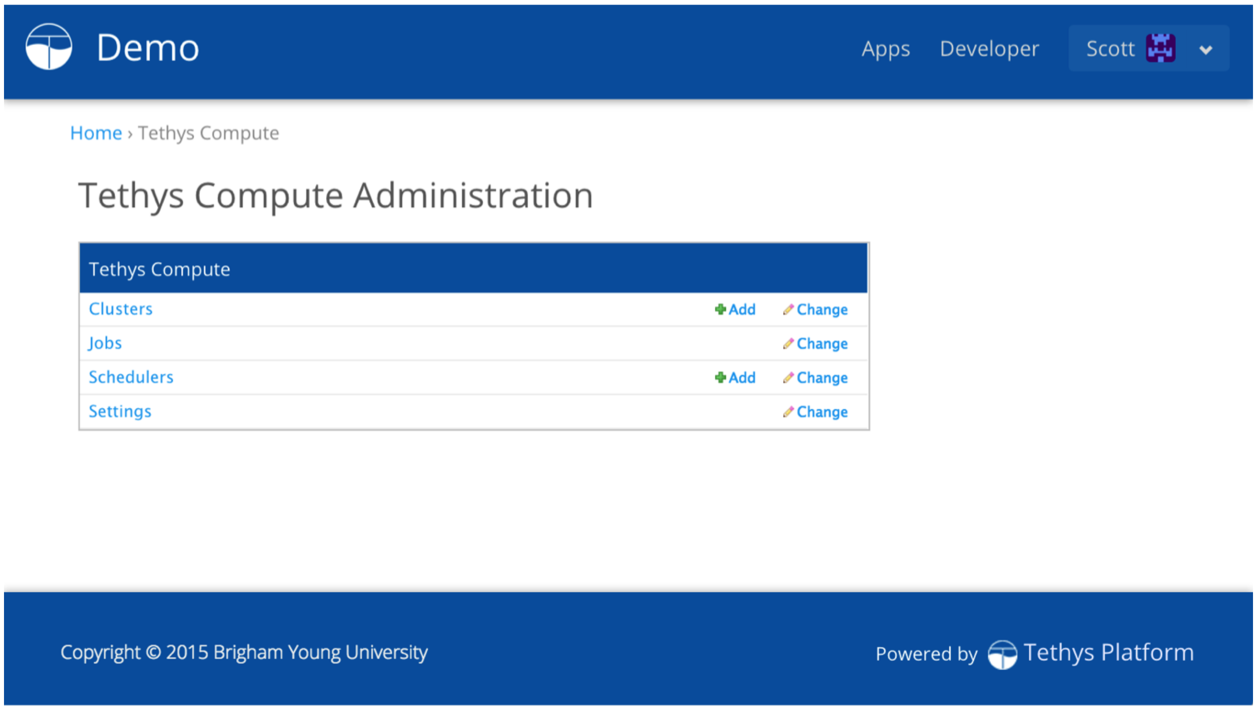
Manage Computing Resources

Computing resources can be managed using the `Tethys Compute` admin pages. Powered, by TethysCluster <http://www.tethysplatform.org/TethysCluster/>`_, these pages allow Tethys Portal administrators to spin up clusters of computing resources on either the Amazon or Microsoft Azure commercial clouds, and link local computing clusters that are managed with HTCondor. These computational These computational resources are accessed in apps through the *Jobs API* and the *Compute API*. For more detailed documentation refer to the links below.

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.

- Clusters
- Jobs
- Schedulers
- Settings



Tethys Compute Administration

Tethys Compute		
Clusters	Add	Change
Jobs		Change
Schedulers	Add	Change
Settings		Change

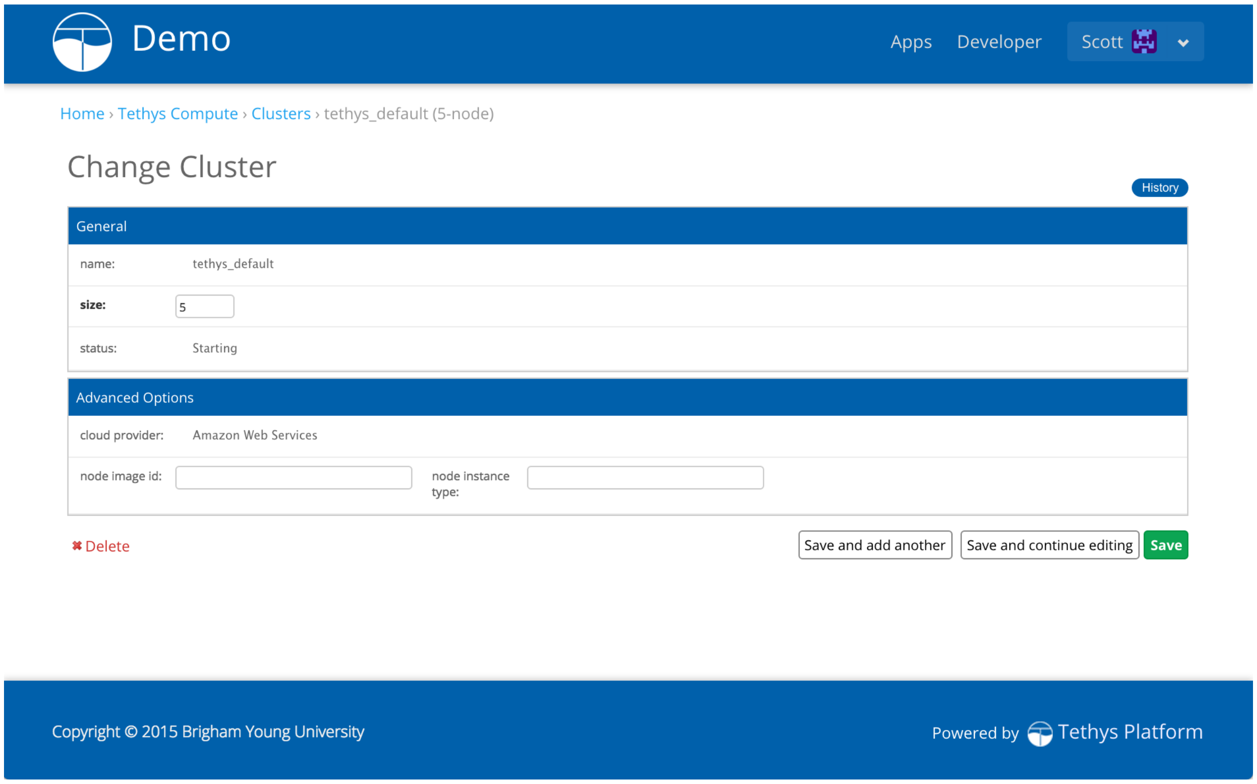
Copyright © 2015 Brigham Young University

Powered by Tethys Platform

Figure 1. Dashboard for Tethys Compute admin pages.

Clusters A cluster is a group of virtual machines (VMs) that are configured with HTCondor so that they provided a distributed computing environment. Each cluster is made up of a master node and zero to many worker nodes. The master node is responsible for assigning jobs to the worker nodes based on their availability and capability. Tethys Platform uses a Python module called TethysCluster to provision and manage clusters using commercial clouds. TethysCluster enables provisioning clusters using either Amazon Web Services (AWS) or Microsoft Azure.

When creating a new cluster there are only two required settings: *name* and size ; and four options settings: master image id, master instance type, node image id, and node instance type.


Demo
Apps
Developer
Scott
Home › Tethys Compute › Clusters › tethys_default (5-node)
History
Delete
Save and add another
Save and continue editing
Save
Copyright © 2015 Brigham Young University
Powered by Tethys Platform

Change Cluster

General

name:	tethys_default
size:	5
status:	Starting

Advanced Options

cloud provider:	Amazon Web Services		
node image id:		node instance type:	

Figure 2. Form for creating a new Cluster.

name The name can be any string, but must be unique among all of the clusters for a given cloud account. Therefore, if the same account credentials are used for two separate instances of Tethys Portal then the the two Tethys Portals may not both have a cluster with the same name.

size The size is the number of VMs the cluster will contain (including the master). The minimum is 1 and the maximum is determined by the limits on the cloud account being uses. The size may be changed after the cluster is created.

master image id The master image id refers to the image that the master node will be made from. When using AWS this would be the AMI ID (e.g. ami-38e4a750). When using Azure it is the name of the image (e.g. tc-ubuntu14). If left blank the master node will be created from the node image id.

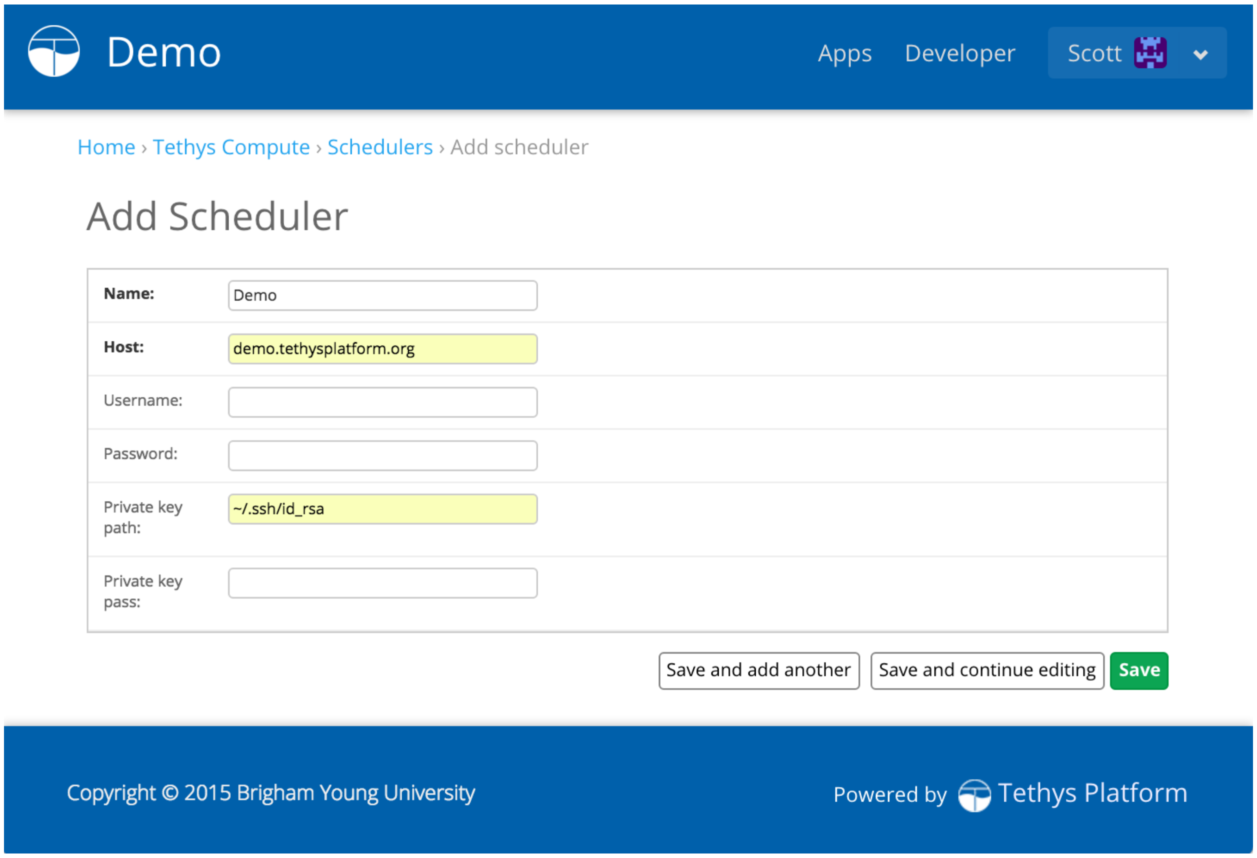
master instance type The master instance type refers to the VM type for the master node. For AWS this would be something like t2.small, m3.medium, etc. For Azure it would be Small, Large, A4, etc. If left blank then the master instance type will be the same as the node instance type. For default values refer to the Default Cluster setting below.

node image id The node image id refers to the image that the worker nodes will be made from. When using AWS this would be the AMI ID (e.g. ami-38e4a750). When using Azure it is the name of the image (e.g. tc-ubuntu14). If left blank the image id specified in the Default Cluster template will be used.

node instance type The node instance type refers to the VM type for the worker nodes. For AWS this would be something like t2.small, m3.medium, etc. For Azure it would be Small, Large, A4, etc. If left blank then the default value specified by the Default Cluster template will be used.

Jobs Jobs represent some sort of computation that is sent from an app to a cluster using the *Job Manager*. 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.


Demo
Apps
Developer
Scott
Home › Tethys Compute › Schedulers › Add scheduler
Add Scheduler

Name:	Demo
Host:	demo.tethysplatform.org
Username:	
Password:	
Private key path:	~/.ssh/id_rsa
Private key pass:	

Save and add another
Save and continue editing
Save
Copyright © 2015 Brigham Young University
Powered by Tethys Platform

Figure 3. Form for creating a new Scheduler.

Name A name to refer to the scheduler. Can be any string, but must be unique among schedulers.

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.

Settings Tethys Compute settings are divided into three sections: Azure Credentials, Amazon Credentials, and Cluster Management. The Azure and Amazon Credentials sections are used to store the cloud account credentials that will be used by Tethys Portal to create clusters. Both Azure and Amazon credentials may be added, however, Tethys Portal is only capable of using one cloud provider at a time. The cloud provider that will be used is determined by the Default Cluster setting in the Cluster Management section. In addition to the Default Cluster setting the Cluster Management section also holds settings for the scheduler server.

Azure Credentials This section contains settings for connecting to an Azure account. There are two required settings: Subscription ID and Certificate Path.

Subscription ID The Subscription ID is a unique identifier for your Azure subscription. For instructions on how to find your subscription id see this video.

Certificate Path The Certificate Path is the path to an SSL certificate file on the Tethys Portal server that is also registered in with your Azure subscription. View these instructions for help creating and uploading a certificate to the Microsoft Azure Management Portal.

Amazon Credentials This section contains settings for connecting to an Amazon Web Services (AWS) account.

AWS Access Key ID The AWS Access Key ID is a unique id for your IAM user. View these instructions for getting your Access Key ID and Secret Access Key.

AWS Secret Access Key The AWS Secret Access Key is like a password for the AWS account. It is associated with your Access Key ID, but is not viewable through the AWS Management Console. They only time a Secret Access Key can be retrieved is when it is created. View these instructions for getting your Access Key ID and Secret Access Key.

AWS User ID The AWS User ID is a unique 12-digit number that identifies the AWS account. This is different from the AWS Access Key ID which is associated with a specific IAM user within an AWS account.

Key Name The Key Name is the name of an SSH key pair that is uploaded to your AWS account. For more information see Amazon EC2 Key Pairs.

Key Location The Key Location is the path to the SSH private key on the Tethys Portal server. For more information see Amazon EC2 Key Pairs.

Cluster Management This section contains general settings for clusters.

Scheduler IP The ip address or host name of the global HTCondor scheduler server. This should be one of the nodes in a cluster.

Note: This setting is deprecated. Use the Schedulers options to set up schedulers now.

Scheduler Key Location The path to the private ssh key allowing passwordless ssh into the scheduler server. When a node in a cluster is used as the scheduler server then this will be the same as either the Key Location (for AWS) or the Certificate Path (for Azure).

Note: This setting is deprecated. Use the Schedulers options to set up schedulers now.

Default Cluster The template that will be used to create new clusters. This value also determines which cloud provider will be used to create clusters. Acceptable values are:

- *azure_default_cluster*
- *aws_default_cluster*

Azure Default Cluster

```
[cluster azure_default_cluster]
CLOUD_PROVIDER = Azure
CLUSTER_SIZE = 1
CLUSTER_SHELL = bash
NODE_IMAGE_ID = ami-3393a45a
NODE_INSTANCE_TYPE = m3.medium
```

AWS Default Cluster

```
[cluster aws_default_cluster]
CLOUD_PROVIDER = AWS
CLUSTER_SIZE = 1
CLUSTER_SHELL = bash
NODE_IMAGE_ID = tc-linux12-2
NODE_INSTANCE_TYPE = Small
```

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.

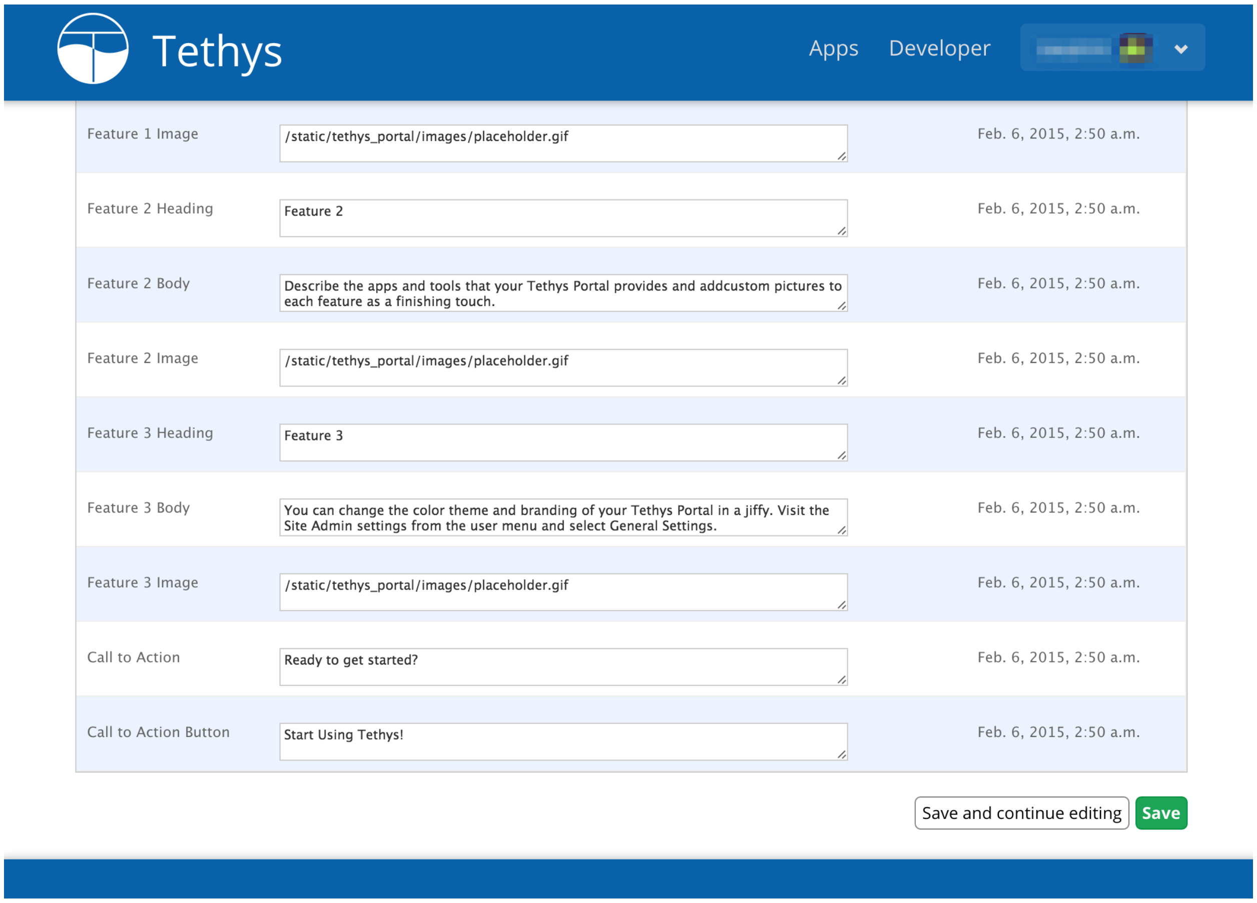
Figure 2. General settings for Tethys Portal.

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).

Figure 3. Home page settings for Tethys Portal.



Feature 1 Image	/static/tethys_portal/images/placeholder.gif	Feb. 6, 2015, 2:50 a.m.
Feature 2 Heading	Feature 2	Feb. 6, 2015, 2:50 a.m.
Feature 2 Body	Describe the apps and tools that your Tethys Portal provides and addcustom pictures to each feature as a finishing touch.	Feb. 6, 2015, 2:50 a.m.
Feature 2 Image	/static/tethys_portal/images/placeholder.gif	Feb. 6, 2015, 2:50 a.m.
Feature 3 Heading	Feature 3	Feb. 6, 2015, 2:50 a.m.
Feature 3 Body	You can change the color theme and branding of your Tethys Portal in a jiffy. Visit the Site Admin settings from the user menu and select General Settings.	Feb. 6, 2015, 2:50 a.m.
Feature 3 Image	/static/tethys_portal/images/placeholder.gif	Feb. 6, 2015, 2:50 a.m.
Call to Action	Ready to get started?	Feb. 6, 2015, 2:50 a.m.
Call to Action Button	Start Using Tethys!	Feb. 6, 2015, 2:50 a.m.

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
```

Social Authentication

Last Updated: August 5, 2015

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

   You will need a Google developer account to register your Tethys Portal with Google. To create an account, visit https://developers.google.com and sign in with a Google account.

2. Create a New Project

   Use the Google Developer Console to create a new project.

3. Create a New Client ID

   After the project has been created, select the project and use the navigation on the left to go to `APIs & auth > Credentials` and press the `Create new Client ID` button in the OAuth section.

   1. Configure the Consent Screen

      In the window that appears, select `Web Application` and press `Configure consent screen`. The consent screen is what the user sees when they log into Tethys using their Google account. You need to provide information like the name of your Tethys Portal and your logo.

2. Provide Authorized 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
   ```

3. 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/
   ```

4. Press `Create Client ID` Button

   Take note the `Client ID` and `Client secret` that are assigned to your app for the next step.

3. Enable the Google+ API

   1. Use the navigation on the left to go to `APIs & auth > APIs`.
   2. Search for `Google+ API` and select it from the results.
   3. Click on the `Enable API` button to enable it.

   Note: Some Google APIs are free to use up to a certain quota of hits. Familiarize your self with the quotas for any APIs you use by selecting the API and viewing the `Quota` tab.

4. Open `settings.py` script located in `/usr/lib/tethys/src/tethys_apps/settings.py`

   Add the `social.backends.google.GoogleOAuth2` backend to the `AUTHENTICATION_BACKENDS` setting:

   ```
   AUTHENTICATION_BACKENDS = (
       ...
       'social.backends.google.GoogleOAuth2',
       'django.contrib.auth.backends.ModelBackend',
   )
   ```

   Assign the `Client ID` and `Client secret` to 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:

- Python Social Auth Supported Backends: Google
- 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.

   Point to `My Apps` and select `Become a Facebook Developer`. Click on `Register Now` and then accept the terms.

2. Create a Facebook App

1. Point to `My Apps` and select `Add a New App`.
2. Select the `Website` option.
3. Type the name of the new app in the text field and press the `Create New Facebook App ID` button from the drop down.
4. Choose a category and press `Create App ID`.
5. View the Quick Start tutorial if you wish or press the `Skip Quick Start` button to skip.

3. Note the `App ID` and `App Secret` for Step 5.

4. Setup OAuth

   1. Select `Settings` from the left navigation menu and add a `Contact Email` address.
   2. Click on the `Advanced` tab and add the callback URIs to the Valid OAuth redirect URIs field. For example, if my Tethys Portal was located at `www.example.org`:

      ```
      https://www.example.org/oauth2/complete/facebook/
      http://localhost:8000/oauth2/complete/facebook/
      ```

   3. Select `Status & Review` from the left navigation menu. Make the app public by changing the toggle switch to `Yes`.

   Note: The Facebook app must be public for you to allow anyone to authenticate using Facebook in your Tethys Portal. For testing, you can use the `Roles` menu item to add specific Facebook users that are allowed to authenticate when the app is in development mode.

5. Open `settings.py` script located in `/usr/lib/tethys/src/tethys_apps/settings.py`

   Add the `social.backends.facebook.FacebookOAuth2` backend to the `AUTHENTICATION_BACKENDS` setting:

   ```
   AUTHENTICATION_BACKENDS = (
       ...
       'social.backends.facebook.FacebookOAuth2',
       'django.contrib.auth.backends.ModelBackend',
   )
   ```

   Assign 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:

- Python Social Auth Supported Backends: Facebook

- 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

1. Navigate back to https://developer.linkedin.com/my-apps, if necessary and press the `Create Application` button.

2. Fill out the form and press `Submit`.

3. Note the `Client ID` and `Client Secret` for Step 5.

4. Setup OAuth

   1. Add the call back URLs under the OAuth 2.0 section. For example, if my Tethys Portal was located at the domain `www.example.org`:

      ```
      https://www.example.org/oauth2/complete/linkedin-oauth2/
      http://localhost:8000/oauth2/complete/linkedin-oauth2/
      ```

   2. Select `Settings` from the left navigation menu. Make the app public by selecting `Live` from the `Application Status` dropdown.

   Note: The LinkedIn app must be public for you to allow anyone to authenticate using LinkedIn in your Tethys Portal. For testing, you can use the `Roles` menu item to add specific LinkedIn users that are allowed to authenticate when the app is in development mode.

5. Open `settings.py` script located in `/usr/lib/tethys/src/tethys_apps/settings.py`

   Add the `social.backends.linkedin.LinkedinOAuth2` backend to the `AUTHENTICATION_BACKENDS` setting:

   ```
   AUTHENTICATION_BACKENDS = (
       ...
       'social.backends.linkedin.LinkedinOAuth2',
       'django.contrib.auth.backends.ModelBackend',
   )
   ```

   Assign 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:

- Python Social Auth Supported Backends: LinkedIn
- 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 a HydroShare Application

1. Navigate to https://www.hydroshare.org/o/applications/register/.
2. See Step 4 for instructions on Redirect URIs.
3. Fill out the form and press `Save`.

3. Note the `Client ID` and `Client Secret` for Step 5.
4. Setup OAuth

   1. Add the call back URLs under the Redirect URIs section. For example, if my Tethys Portal was located at the domain `www.example.org`:

      ```
      https://www.example.org/oauth2/complete/hydroshare/
      http://localhost:8000/oauth2/complete/hydroshare/
      ```

5. Open `settings.py` script located in `/usr/lib/tethys/src/tethys_apps/settings.py`

   Add the `social.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 = '...'
   ```

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

Social authentication requires Tethys Platform 1.2.0 or later. If you are using an older version of Tethys Platform, you will need to upgrade by following either the *Upgrade from 1.3 to 1.4* or the *Upgrade from 1.3 to 1.4* instructions. The `settings.py` script is unaffected by the upgrade. You will need to either generate a new `settings.py` script using `tethys gen settings` or add the following settings to your existing `settings.py` script to support social login.

```
INSTALLED_APPS = (
    ...
    'social.apps.django_app.default',
)

MIDDLEWARE_CLASSES = (
```

```
    ...
    'tethys_portal.middleware.TethysSocialAuthExceptionMiddleware',
)

TEMPLATE_CONTEXT_PROCESSORS = (
    ...
    'django.core.context_processors.request',
    'social.apps.django_app.context_processors.backends',
    'social.apps.django_app.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
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 = ''
```

Developer Tools

Last Updated: August 4, 2015

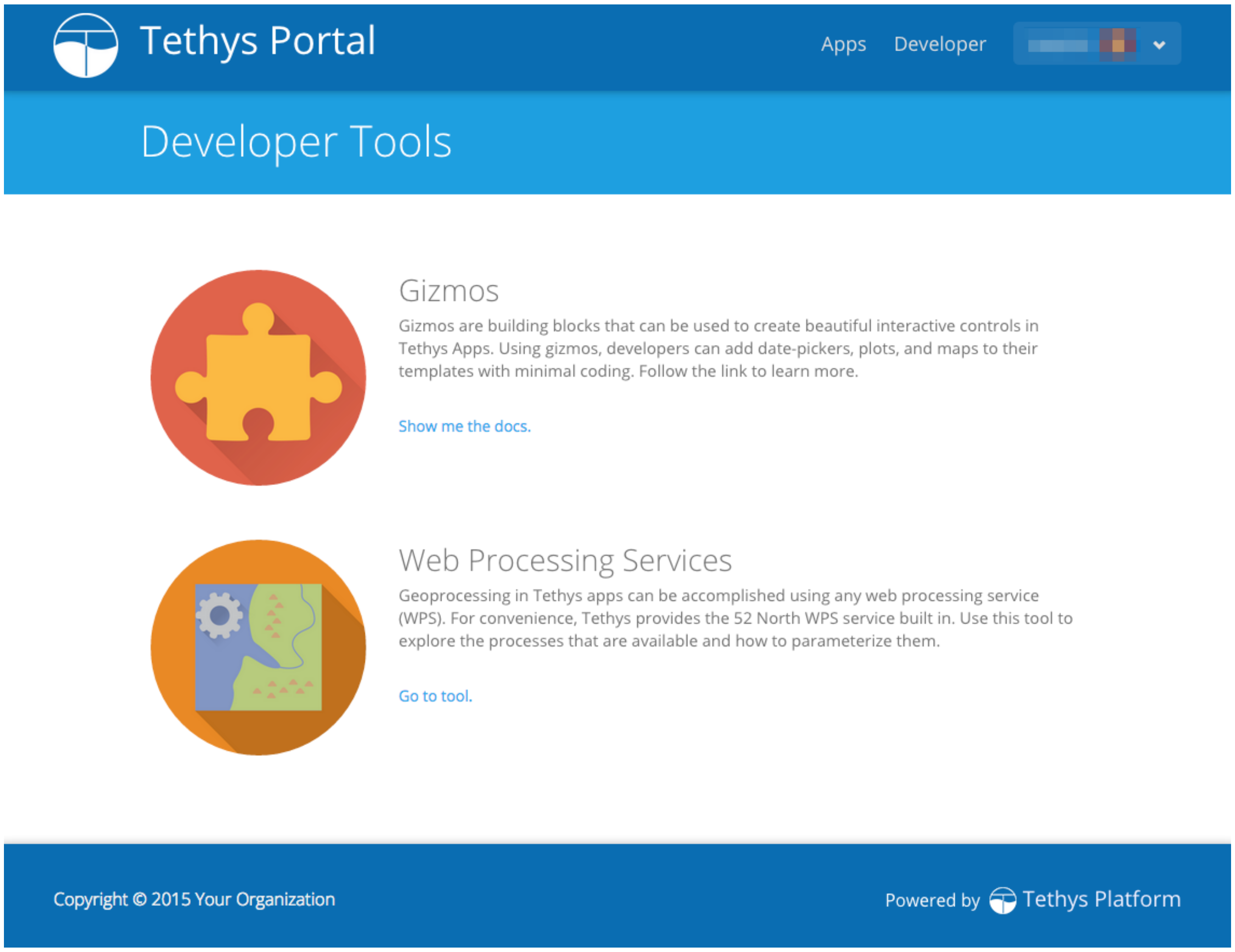
Tethys provides a Developer Tools page that is accessible when you run Tethys in developer mode. Developer Tools contain documentation, code examples, and live demos of the features of various features of Tethys. Use it to learn how to add a map or a plot to your web app using Gizmos or browse the available geoprocessing capabilities and formulate geoprocessing requests interactively.

Figure 4. Use the Developer Tools page to assist you in development.

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.



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.

Web Processing Services

Geoprocessing in Tethys apps can be accomplished using any web processing service (WPS). For convenience, Tethys provides the 52 North WPS service built in. Use this tool to explore the processes that are available and how to parameterize them.

Go to tool.

Copyright © 2015 Your Organization

Powered by Tethys Platform

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 *6. Setup Email Capabilities* 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']
```

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
- CKAN

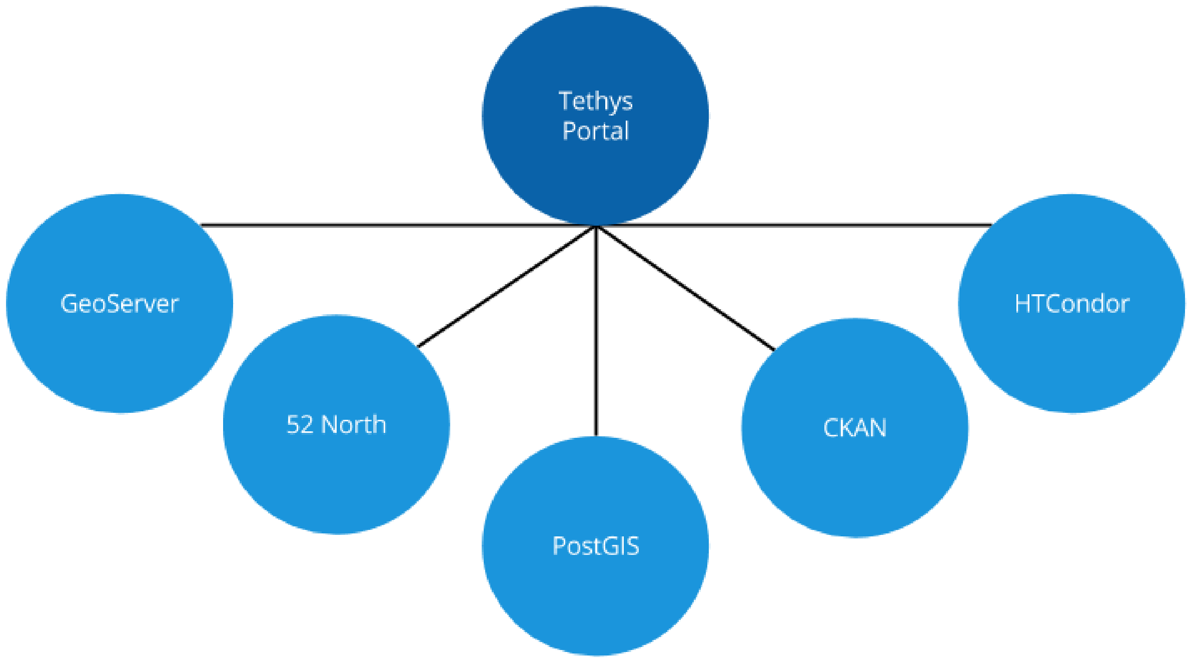


Figure 1.2: **Figure 1.** Tethys Platform consists of several software components that should be hosted on separate servers in a production environment.

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 on Ubuntu 14.04

Last Updated: August 11, 2015

This article will provide an overview of how to install Tethys Portal in a production setup ready to host apps. The recommended deployment platform for Python web projects is to use WSGI. The easiest and most stable way to deploy a WSGI application is with the modwsgi extension for the Apache Server. These instructions are optimized for Ubuntu 14.04 using Apache and modwsgi, though installation on other Linux distributions will be similar.

1. Install Tethys Portal

Follow the default `../installation/linux` instructions to install Tethys Portal with the following considerations

- Assign strong passwords to the database users.
- Create a new settings file, do not use the same file that you have been using in development.
- Optionally, Follow the *Distributed Configuration* instructions to install Docker and the components of the software suite on separate servers.

2. Install Apache and Dependencies

Install Apache and the `modwsgi` module if they are not installed already. In this tutorial, `vim` is used to edit file, however, you are welcome to use any text editor you are comfortable with.

```
$ sudo apt-get install apache2 libapache2-mod-wsgi vim
```

3. Make BASELINE Virtual Environment

An additional virtual environment needs to be created to use `modwsgi` in Apache. This virtual environment needs to be independent of the Tethys virtual environment and the system Python installation.

```
$ sudo mkdir -p /usr/local/pythonenv
$ sudo virtualenv --no-site-packages /usr/local/pythonenv/BASELINE
```

4. Set WSGI Python Home

Edit the Apache configuration to use the `BASELINE` environment as the home python for WSGI. Open `apache2.conf` using `vim` or another text editor:

```
$ sudo vim /etc/apache2/apache2.conf
```

To edit the file using `vim`, you need to be in `INSERT` mode. Press `i` to enter `INSERT` mode and add this line to the bottom of the `apache2.conf` file:

```
WSGIPythonHome /usr/local/pythonenv/BASELINE
```

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

5. Make Directories for Static Files and TethysCluster

When running Tethys Platform in development mode, the static files are automatically served by the development server. In a production environment the static files will need to be collected into one location and Apache or another server will need to be configured to serve these files (see Deployment Checklist: STATIC_ROOT). Since Apache will be serving Tethys Portal under Apache user (www-data) the TethysCluster home directory also needs to be created:

```
$ sudo mkdir /var/www/.tethyscluster && sudo mkdir -p /var/www/tethys/static
$ sudo chown `whoami` /var/www/tethys/static
```

6. Setup Email Capabilities

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
```

Django must be configured to use the postfix server. The next section will describe the Django settings that must be configured for the email server to work. For an excellent guide on setting up Postfix on Ubuntu, refer to How To Install and Setup Postfix on Ubuntu 14.04.

7. Set Secure Settings

Several settings need to be modified in the settings.py module to make the installation ready for a production environment. 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. As a minimum do the following:

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

```
$ sudo vim /usr/lib/tethys/src/tethys_apps/settings.py
```

Press i to start editing and change the following settings:

1. Create new secret key

   Create a new SECRET_KEY for the production installation of Tethys Platform. Do not use the same key you used during development and keep the key a secret. Take care not to store the settings.py file with the production secret key in a repository. Django outlines several suggestions for making the secret key more secure in the Deployment Checklist: SECRET_KEY documentation.

2. Turn off debugging

   Turn off the debugging settings by changing DEBUG and TEMPLATE_DEBUG to False. **You must never turn on debugging in a production environment.**

   ```
   DEBUG = False
   ```

3. Set the allowed hosts

   Allowed hosts must be set to a suitable value, usually a list of the names and aliases of the server that you are hosting Tethys Portal on (e.g.: "www.example.com"). Django will not work without a value set for the ALLOWED_HOSTS parameter when debugging is turned of. See the Deployment Checklist: ALLOWED_HOSTS for more information.

   ```
   ALLOWED_HOSTS = ['www.example.com']
   ```

4. Set the static root directory

   You must set the STATIC_ROOT settings to tell Django where to collect all of the static files. Set this setting to the directory that was created in the previous step (/var/www/tethys/static). See the Deployment Checklist: STATIC_ROOT for more details.

   ```
   STATIC_ROOT = '/var/www/tethys/static'
   ```

5. Set email settings

   Several email settings need to be configured for the forget password functionality to work properly. The following exampled illustrates how to setup email using the Postfix installation from above:

   ```
   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.

4. Setup social authentication

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

5. Configure workspaces (optional)

   If you would like all of the app workspace directories to be aggregated to a central location, create the directory and then specify it using the `TETHYS_WORKSPACES_ROOT` setting.

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

Important: Review the Deployment Checklist carefully.

8. Create Apache Site Configuration File

Create an Apache configuration for your Tethys Platform using the **gen** command and open the `tethys-default.conf` file that was generated using `vim`:

```
         $ sudo su
         $ . /usr/lib/tethys/bin/activate
(tethys) $ tethys gen apache -d /etc/apache2/sites-available
(tethys) $ vim /etc/apache2/sites-available/tethys-default.conf
(tethys) $ exit
```

Press `i` to enter `INSERT` mode and edit the file. Change the `ServerName` and `ServerAlias` to match the domain for your Tethys Portal. The `tethys-default.conf` will look similar to this when you are done:

```
<VirtualHost 0.0.0.0:80>
    ServerName example.net
    ServerAlias www.example.net

    Alias /static/ /var/www/tethys/static/

    <Directory /var/www/tethys/static/>
        Require all granted
    </Directory>

    WSGIScriptAlias / /usr/lib/tethys/src/tethys_portal/wsgi.py

    <Directory /usr/lib/tethys/src/tethys_portal>
        <Files wsgi.py>
            Require all granted
        </Files>
    </Directory>

    # Daemon config
    WSGIDaemonProcess tethys_default \
     python-path=/usr/lib/tethys/src/tethys_portal:/usr/lib/tethys/lib/python2.7/site-packages
    WSGIProcessGroup tethys_default
    # Logs
    ErrorLog /var/log/apache2/tethys_default.error.log
    CustomLog /var/log/apache2/tethys_default.custom.log combined
</VirtualHost>
```

There is a lot going on in this file, for more information about Django and WSGI review Django's How to deploy with WSGI documentation.

9. Install Apps

Download and install any apps that you want to host using this installation of Tethys Platform. It is recommended that you create a directory to store the source code for all of the apps that you install. The installation of each app may vary, but generally, an app can be installed as follows:

```
         $ sudo su
         $ . /usr/lib/tethys/bin/activate
(tethys) $ cd /path/to/tethysapp-my_first_app
(tethys) $ python setup.py install
(tethys) $ exit
```

10. Collect Static Files

The static files need to be collected into the directory that you created. Enter the following commands and enter "yes" if prompted:

```
         $ sudo su
         $ . /usr/lib/tethys/bin/activate
(tethys) $ tethys manage collectstatic
(tethys) $ exit
```

11. Collect Workspaces (optional)

If you configured a workspaces directory with the `TETHYS_WORKSPACES_ROOT` setting, you will need to run the following command to collect all the workspaces to that directory:

```
         $ sudo su
         $ . /usr/lib/tethys/bin/activate
(tethys) $ tethys manage collectworkspaces
(tethys) $ exit
```

12. Setup the Persistent Stores for Apps

After all the apps have been successfully installed, you will need to initialize the persistent stores for the apps:

```
         $ . /usr/lib/tethys/bin/activate
(tethys) $ tethys syncstores all
```

13. Transfer Ownership to Apache

When you are finished installing Tethys Portal, change the ownership of the source code and static files to be the Apache user (`www-data`):

```
$ sudo chown -R www-data:www-data /usr/lib/tethys/src /var/www/tethys/static /var/www/.tethyscluster
```

14. Enable Site and Restart Apache

Finally, you need to disable the default apache site, enable the Tethys Portal site, and reload Apache:

```
$ sudo a2dissite 000-default.conf && sudo a2ensite tethys-default.conf && sudo service apache2 reloa
```

Tip: To install additional apps after the initial setup of Tethys, you will follow the following process:

1. Change ownership of the `src` and `static` directories to your user using the patter in step 12 OR login as root user using `sudo su`.
2. Install apps, syncstores, collectstatic, and collectworkspaces as in steps 9-12.
3. Transfer ownership of files to Apache user as in step 13.
4. Reload the apache server using `sudo service apache2 reload`.

For more information see: *Installing Apps in Production*.

Installing Apps in Production

Last Updated: August 10, 2015

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

1. Create a Directory for App Source

Create a directory on your server that will store the source code for the apps that are installed on your server. For example:

```
$ sudo mkdir -p /var/www/tethys/apps/
```

2. 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 /var/www/tethys/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.

---

3. 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:

```
         $ sudo su
         $ . /usr/lib/tethys/bin/activate
(tethys) $ cd /var/www/tethys/apps/tethysapp-my_first_app
(tethys) $ python setup.py install
(tethys) $ exit
```

Tip: If you plan to execute the commands in steps 4 - 6, do not run the `exit` until after you have completed the commands in these steps. That way you will not need to run the `sudo su` and ‘‘ . /usr/lib/tethys/bin/activate‘‘ commands multiple times.

4. Collect and Static Files

The static files for apps are hosted by Apache, which necessitates collecting all of the static files to a single directory. This directory is configured through the `STATIC_ROOT` setting in the `settings.py` script. Collect the static files with this command:

```
         $ sudo su
         $ . /usr/lib/tethys/bin/activate
(tethys) $ tethys manage collectstatic
(tethys) $ exit
```

5. Collect Workspaces (optional)

As a means of optimizing storage on the server, the workspaces of apps can be collected to a central location. This location is configured through the `TETHYS_WORKSPACES_ROOT` setting in the `settings.py` script. Collect the workspaces with this command:

```
$ sudo su
         $ . /usr/lib/tethys/bin/activate
(tethys) $ tethys manage collectworkspaces
(tethys) $ exit
```

Tip: The `collectall` command provides a shortcut for running both `collectstatic` and `collectworkspaces` commands:

```
         $ sudo su
         $ . /usr/lib/tethys/bin/activate
(tethys) $ tethys manage collectall
(tethys) $ exit
```

6. Initialize Persistent Stores (optional)

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

```
         $ sudo su
         $ . /usr/lib/tethys/bin/activate
(tethys) $ tethys syncstores my_first_app
(tethys) $ exit
```

7. Change the Ownership of Files to the Apache User

The Apache user must own any files that it Apache is serving. This includes the source files, static files, and any workspaces that your app may have. Assuming the sources files, static files, and workspaces are all located in the `/var/www/tethys/` directory, the following command will accomplish the change in ownership that is required:

```
$ sudo chown -R www-data:www-data /var/www/tethys/ /usr/lib/tethys/src/tethys_apps/tethysapp/
```

Note: The name of the Apache user in RedHat or CentOS flavored systems is `apache`, not `www-data`.

8. Restart Apache

Restart Apache to effect the changes:

```
$ sudo service apache2 restart
```

Note: The command for managing Apache on CentOS or RedHat flavored systems is `httpd`. Restart as follows:

```
$ sudo service httpd restart
```

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 TE
```

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 c
```

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"
```

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
```

Upgrade from 1.3 to 1.4

Last Updated: December 1, 2016

1. Pull Repository

When you installed Tethys Platform you did so using it's remote Git repository on GitHub. To get the latest version of Tethys Platform, you will need to pull the latest changes from this repository:

```
$ sudo su
$ cd /usr/lib/tethys/src
$ git pull origin master
$ exit
```

2. Install Requirements and Run Setup Script

Install new dependencies and upgrade old ones:

```
         $ sudo su
         $ . /usr/lib/tethys/bin/activate
(tethys) $ pip install --upgrade -r /usr/lib/tethys/src/requirements.txt
(tethys) $ python /usr/lib/tethys/src/setup.py develop
(tethys) $ exit
```

3. Generate New Settings Script

Backup your old settings script (`settings.py`) and generate a new settings file to get the latest version of the settings. Then copy any settings (like database usernames and passwords) from the backed up settings script to the new settings script.

```
         $ sudo su
(tethys) $ mv /usr/lib/tethys/src/tethys_apps/settings.py /usr/lib/tethys/src/tethys_apps/settings.py
(tethys) $ tethys gen settings -d /usr/lib/tethys/src/tethys_apps
(tethys) $ exit
```

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

After you have copied these settings, you can delete the backup settings script.

4. Setup Social Authentication (optional)

If you would like to allow users to signup using their social credentials from Facebook, Google, LinkedIn, and/or HydroShare, follow the *Social Authentication* instructions.

5. Sync the Database

Start the database docker if not already started and apply any changes to the database that may have been issued with the new release:

```
         $ . /usr/lib/tethys/bin/activate
(tethys) $ tethys docker start -c postgis
(tethys) $ tethys manage syncdb
```

Note: For migration errors use:

```
$ cd ~/usr/lib/tethys/src
$ python manage.py makemigrations --merge
$ tethys manage syncdb
```

6. Collect Static Files

Collect the new static files and update the old ones:

```
         $ sudo su
(tethys) $ tethys manage collectstatic
(tethys) $ exit
```

7. Transfer Ownership to Apache

Assign ownership of Tethys Platform files and resources to the Apache user:

```
$ sudo chown -R www-data:www-data /usr/lib/tethys/src /var/www/tethys
```

Note: The name of the Apache user in RedHat or CentOS flavored systems is `apache`, not `www-data`.

8. Restart Apache

Restart Apache to effect the changes:

```
$ sudo service apache2 restart
```

Note: The command for managing Apache on CentOS or RedHat flavored systems is `httpd`. Restart as follows:

```
$ sudo service httpd restart
```

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

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

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.

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.

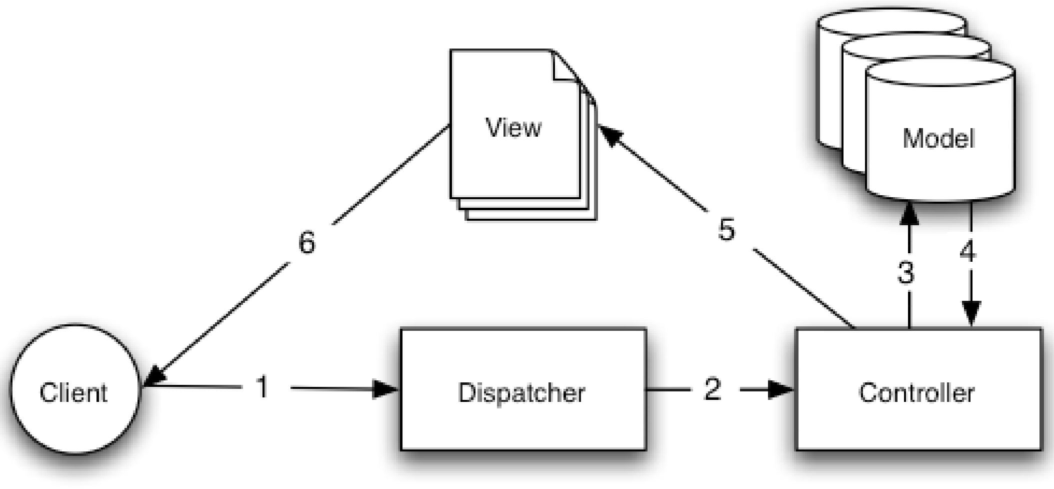


Figure 1.3: A typical collaboration of MVC components (courtesy of Cake PHP Docs)

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

# 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.

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.

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 the *Distributing Apps* tutorial and 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.

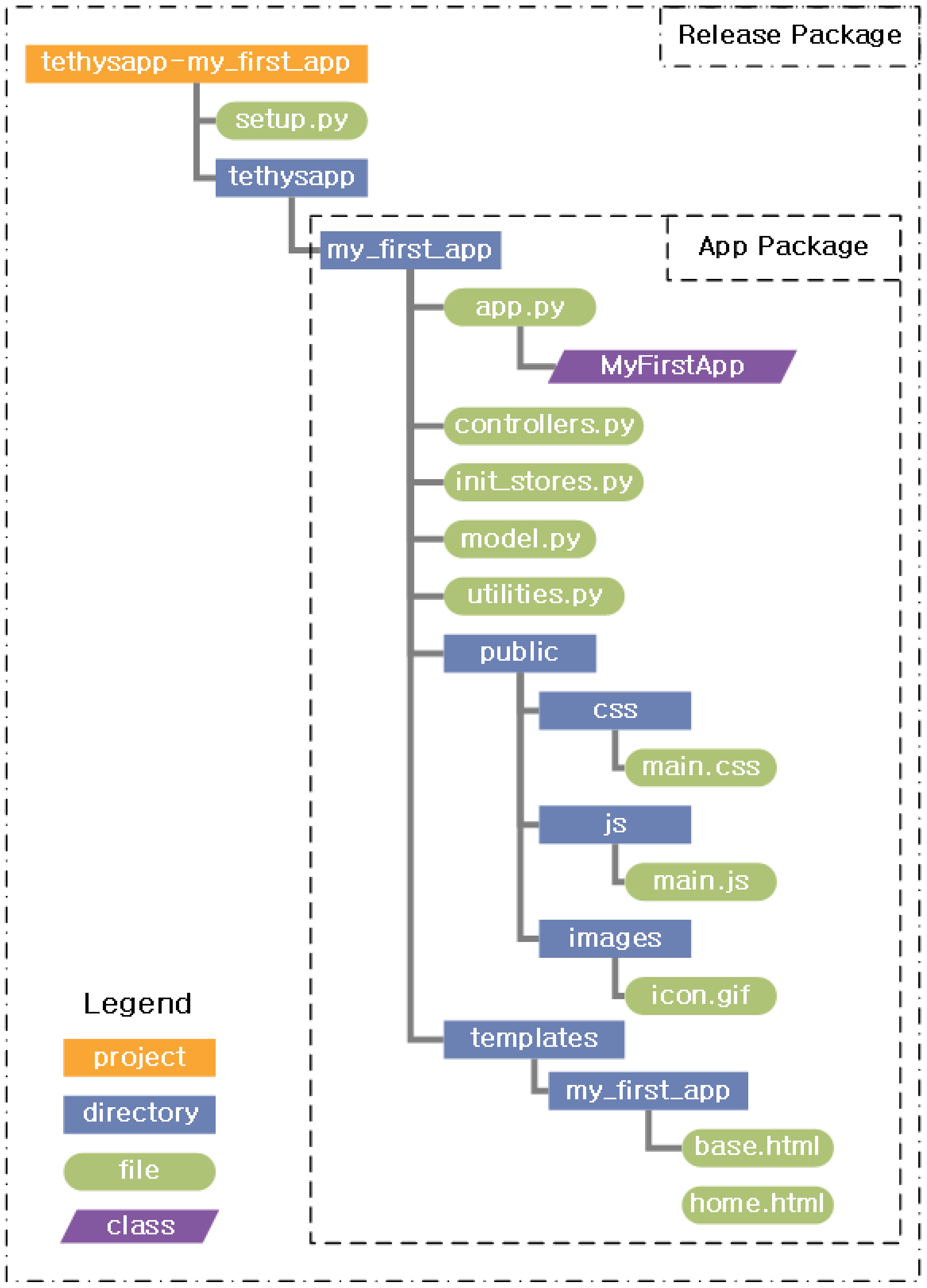


Figure 1.4: **Figure 1. An example of a Tethys app project for an app named "my_first_app".**

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.

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.

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.

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*.

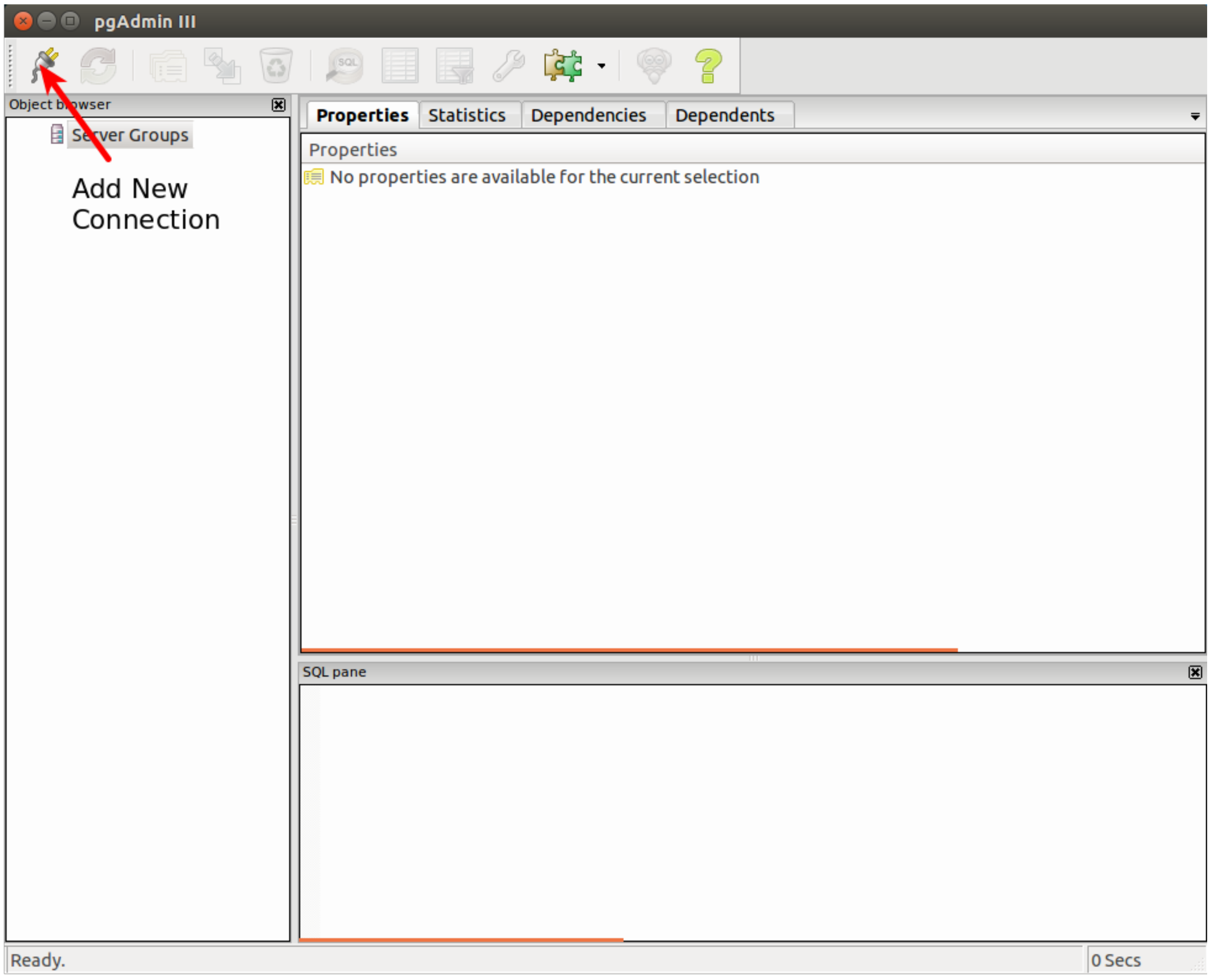
PGAdmin III Tutorial

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.

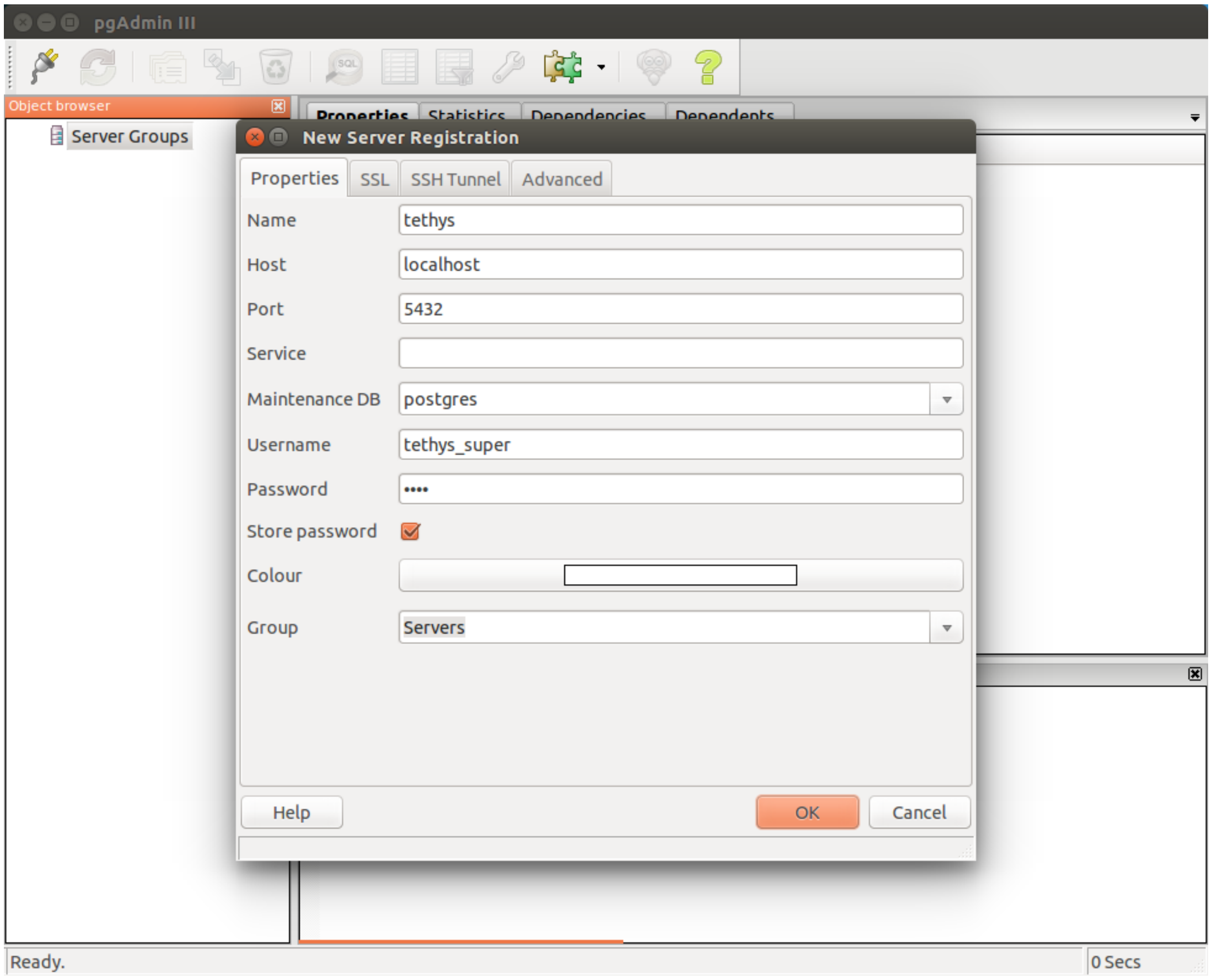


No properties are available for the current selection

Figure 1.5: **Figure 1.** Click the `Add New Connection` button.

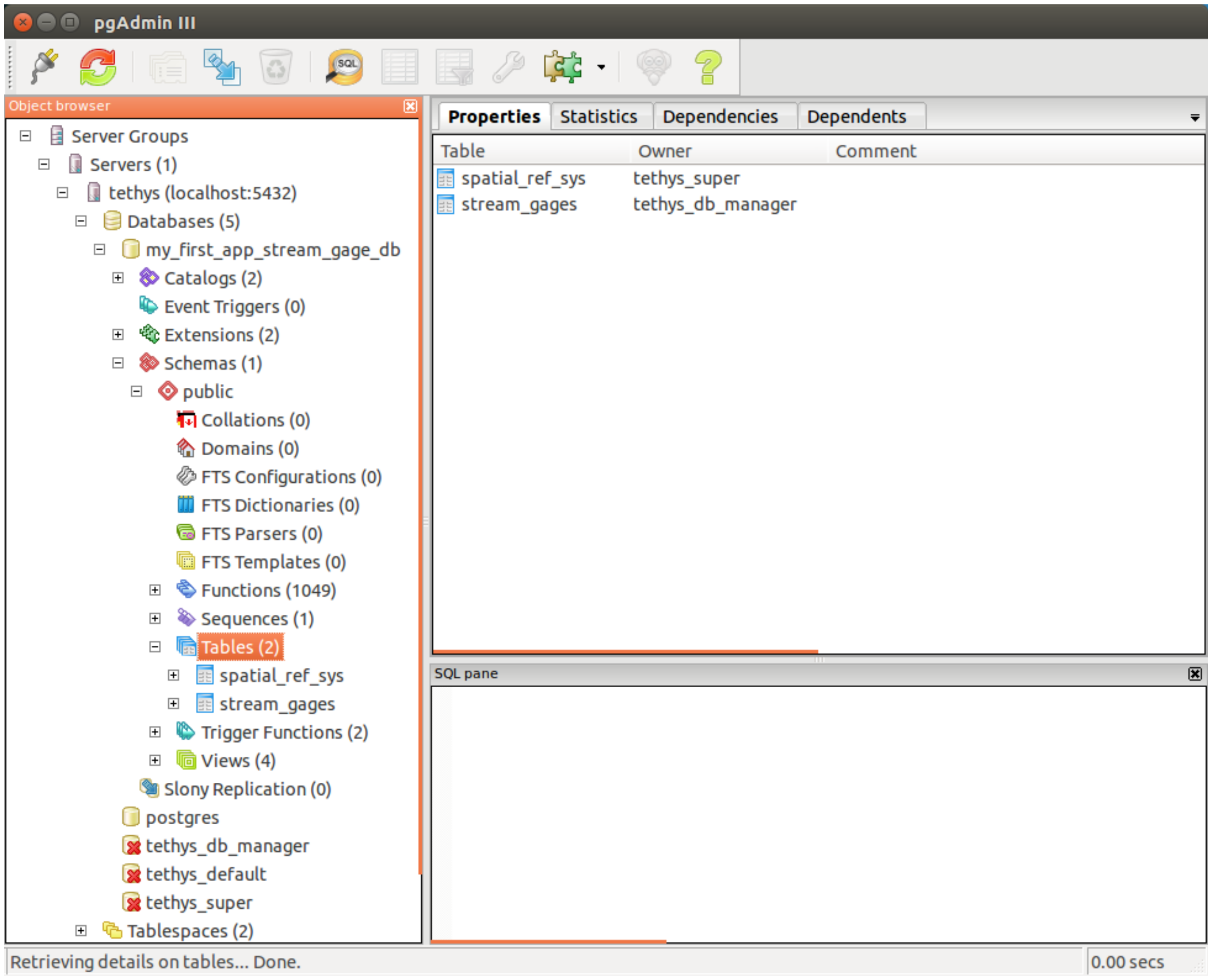
2. In the New Server Registration dialog that appears, fill out the form with the appropriate credentials. Provide a meaningful name for the connection like "tethys". If you have installed PostgreSQL with the Docker containers, the host will be either `localhost` if you are on Linux or `192.168.59.103` 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`.



Name	tethys
Host	localhost
Port	5432
Service	
Maintenance DB	postgres
Username	tethys_super
Password	••••
Store password	✓
Colour	
Group	Servers

Figure 1.6: **Figure 2.** Fill out the New Server Registration dialog.



Server Groups
Servers (1)
tethys (localhost:5432)
Databases (5)
my_first_app_stream_gage_db
Catalogs (2)
Event Triggers (0)
Extensions (2)
Schemas (1)
public
Collations (0)
Domains (0)
FTS Configurations (0)
FTS Dictionaries (0)
FTS Parsers (0)
FTS Templates (0)
Functions (1049)
Sequences (1)
Tables (2)
spatial_ref_sys
stream_gages
Trigger Functions (2)
Views (4)
Slony Replication (0)
postgres
tethys_db_manager
tethys_default
tethys_super
Tablespaces (2)

Table	Owner	Comment
spatial_ref_sys	tethys_super	
stream_gages	tethys_db_manager	

```
Retrieving details on tables... Done.    0.00 secs
```

Figure 1.7: **Figure 3.** Browse the databases using the graphical interface.

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

Glossary

Last Updated: November 18, 2014

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 Model and Persistent Stores* tutorial and 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. For more details see *Distributing 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 virtual environment An isolated Python installation. Many operating systems use the system Python installation to perform maintenance operations. Installing Tethys Platform in a virtual environment prevents potential dependency conflicts.

wps service, wps services A WPS Service provides processes/geoprocesses as web services using the Open Geospatial Consortium Web Processing Service (WPS) standard.

CHAPTER 2

Acknowledgements

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

CHAPTER 3

Indices and tables

- *genindex*
- *search*