



U.S. DEPARTMENT OF  
**ENERGY**

PNNL-24318

Prepared for the U.S. Department of Energy  
under Contract DE-AC05-76RL01830

# OpenEIS: Developer Guide

RG Lutes  
JN Haack  
KE Monson  
P Sharma

CC Neubauer  
BJ Carpenter  
CH Allwardt  
BA Akyol

March 2015



**Pacific Northwest**  
NATIONAL LABORATORY

*Proudly Operated by **Battelle** Since 1965*

## DISCLAIMER

United States Government. Neither the United States Government nor any agency thereof, nor Battelle Memorial Institute, nor any of their employees, makes **any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.** Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or Battelle Memorial Institute. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

PACIFIC NORTHWEST NATIONAL LABORATORY

*operated by*

BATTELLE

*for the*

UNITED STATES DEPARTMENT OF ENERGY

*under Contract DE-AC05-76RL01830*

Printed in the United States of America

Available to DOE and DOE contractors from the

Office of Scientific and Technical Information,

P.O. Box 62, Oak Ridge, TN 37831-0062;

ph: (865) 576-8401, fax: (865) 576-5728

email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service,  
U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161

ph: (800) 553-6847, fax: (703) 605-6900

email: orders@ntis.fedworld.gov

online ordering: <http://www.ntis.gov/ordering.htm>



This document was printed on recycled paper.

(8/00)

## OpenEIS: Developer Guide

RG Lutes  
CC Neubauer  
JN Haack  
BJ Carpenter  
KE Monson  
CH Allwardt  
P Sharma  
BA Akyol

March 2015

Prepared for  
U.S. Department of Energy  
under Contract DE-AC05-76RL01830

Pacific Northwest National Laboratory  
Richland, Washington 99352

## Summary

The Department of Energy's (DOE's) Building Technologies Office (BTO) is supporting the development of an open-source software tool for analyzing building energy and operational data: OpenEIS (open energy information system). This tool addresses the problems of both owners of building data and developers of tools to analyze this data. Building owners and managers have data but lack the tools to analyze it while tool developers lack data in a common format to ease development of reusable data analysis tools.

OpenEIS provides a platform that allows those with data to map their data to a common format which then allows researchers to develop analysis tools against this format instead of some device/vendor specific format. OpenEIS also provides application developers with the services necessary to work with the data and produce visualizations based on their analysis.

This document is intended for developers of applications and explains the mechanisms for building analysis applications, accessing data, and displaying data using a visualization from the included library. A brief introduction to the visualizations can be used as a jumping off point for developers familiar with JavaScript to produce their own. Several example applications are included which can be used along with this document to implement algorithms for performing energy data analysis.

## Table of Contents

|  |    |
|--|----|
| Summary .....  | iv |
| 1 Introduction .....   | 1  |
| 1.1 Document Background.....   | 2  |
| 1.2 Document Conventions .....   | 2  |
| 2 Installation of the OpenEIS .....  | 3  |
| 2.1 Project Software Requirements .....  | 3  |
| 2.2 Building the Project.....  | 4  |
| 2.2.1 Building the Project / Bootstrap OpenEIS .....   | 4  |
| 2.2.2 Installing Numpy (Windows Only).....   | 5  |
| 2.3 Install Eclipse and Eclipse Plug-ins .....   | 6  |
| 2.3.1 Installation of PyDev Plug-in for Eclipse.....   | 6  |
| 2.3.2 Install PyDev Alternate Steps .....  | 7  |
| 2.3.3 Installation of EGit Plug-in for Eclipse.....  | 11 |
| 3 Project Configuration .....  | 14 |
| 3.1 Importing the OpenEIS into Eclipse .....   | 14 |
| 3.1.1 Import OpenEIS into Eclipse from an Existing Local Repository (Previously<br>Downloaded OpenEIS Project) ..... | 14 |
| 3.1.2 Import New OpenEIS Project from GitHub.....  | 17 |
| 3.2 Configuring PyDev .....  | 21 |
| 4 Running the OpenEIS Server .....   | 27 |
| 4.1 Running Applications in Command Line.....  | 28 |
| 4.2 Running Tests.....   | 35 |
| 5 OpenEIS Sensor Data and Server Interaction.....  | 40 |
| 5.1 Sensor Data and Naming Convention .....  | 40 |
| 5.1.1 Data Mapping (sensormap-schema.json) .....   | 40 |
| 5.1.2 Sensor Data and Associated Units (units.json).....   | 40 |
| 5.1.3 Sensor and Object Definition (general_definition.json).....  | 42 |
| 5.2 Database .....   | 44 |
| 5.2.1 User Account and Project Organization Models .....   | 44 |
| 5.2.2 Data Mapping and Ingestion Models .....  | 44 |
| 5.2.3 Analysis and Results Models .....  | 45 |
| 5.3 RESTful Interface .....  | 45 |
| 5.3.1 Hypertext Transfer Protocol Status Codes .....   | 45 |
| 5.3.2 Views.....   | 46 |
| 5.4 OpenEIS Server API .....   | 47 |

|       |  |    |
|-------|--|----|
| 5.4.1 | API Root.....  | 47 |
| 5.4.2 | API Projects Page.....   | 48 |
| 5.4.3 | API Files Page.....  | 49 |
| 5.4.4 | API Sensor Data Maps Page .....  | 50 |
| 5.4.5 | API Datasets Page .....  | 52 |
| 5.4.6 | API Authentication Page.....   | 54 |
| 5.4.7 | Analyses Page .....  | 54 |
| 5.4.8 | Example Interaction with RESTful API.....  | 55 |
| 5.5   | OpenEIS Server Commands.....   | 58 |
| 5.5.1 | Authorization Commands .....   | 59 |
| 5.5.2 | Django Commands .....  | 59 |
| 5.5.3 | django_pytest Commands .....   | 61 |
| 5.5.4 | Project Commands .....   | 61 |
| 5.5.5 | Server Commands .....  | 61 |
| 5.5.6 | Session Commands .....   | 62 |
| 5.5.7 | staticfiles.....   | 62 |
| 5.6   | Data Manipulation Filters.....   | 62 |
| 5.6.1 | OpenEIS Filters .....  | 62 |
| 5.6.2 | Example Filters .....  | 63 |
| 6     | Creating Applications .....  | 68 |
| 6.1   | Application Interaction with the Database .....                                      | 68 |
| 6.1.1 | Application Interactions with the OpenEIS Database: Retrieving Input Data .....      | 69 |
| 6.1.2 | Application Interactions with the OpenEIS Database: Uploading Analysis Results ..... | 71 |
| 6.2   | Driven Applications .....  | 72 |
| 6.2.1 | Example Driven Application.....  | 73 |
| 6.2.2 | Results Class .....  | 81 |
| 6.3   | Driver Applications .....  | 82 |
| 6.3.1 | Example Driver Application .....   | 82 |
| 7     | OpenEIS Report Elements/Visualizations .....   | 92 |
| 7.1   | Creating a Stock Visualization .....   | 92 |
| 7.1.1 | On Server.....   | 92 |
| 7.1.2 | In Client.....   | 92 |
| 7.2   | Included Visualizations .....  | 92 |
| 7.3   | Adding Visualizations .....  | 93 |
| 7.3.1 | On Server.....   | 93 |
| 7.3.2 | In Client.....   | 93 |
| 7.3.3 | Build, Run, and View .....   | 94 |
| 8     | Additional Support .....   | 95 |



## Figures

|  |    |
|--|----|
| Figure 1: Command Prompt after Successfully Cloning OpenEIS Project from GitHub.....                         | 4  |
| Figure 2: Command Prompt after Successful Completion of OpenEIS Bootstrap .....                              | 5  |
| Figure 3: Launching Eclipse and Selecting a Workspace .....  | 6  |
| Figure 4: Installing PyDev Plug-in for Eclipse .....   | 7  |
| Figure 5: Installing PyDev Plug-in for Eclipse (continued).....  | 8  |
| Figure 6: Installing PyDev Plug-in for Eclipse (continued).....  | 9  |
| Figure 7: Installing PyDev Plug-in for Eclipse (continued).....  | 10 |
| Figure 8: Installing PyDev Plug-in for Eclipse (continued).....  | 11 |
| Figure 9: Installing Eclipse EGit Plug-in .....  | 11 |
| Figure 10: Installing Eclipse EGit Plug-in (continued).....  | 12 |
| Figure 11: Installing Eclipse EGit Plug-in (continued).....  | 13 |
| Figure 12: Importing OpenEIS with Eclipse from Local Source.....   | 14 |
| Figure 13: Importing OpenEIS with Eclipse from Local Source (continued) .....                                | 15 |
| Figure 14: Importing OpenEIS with Eclipse from Local Source (continued) .....                                | 15 |
| Figure 15: Importing OpenEIS with Eclipse from Local Source (continued) .....                                | 16 |
| Figure 16: Importing OpenEIS with Eclipse from Local Source (continued) .....                                | 16 |
| Figure 17: Importing OpenEIS with Eclipse from Local Source (continued) .....                                | 17 |
| Figure 18: Checking Out OpenEIS with Eclipse from GitHub.....  | 18 |
| Figure 19: Checking Out OpenEIS with Eclipse from GitHub (continued) .....                                   | 18 |
| Figure 20: Checking Out OpenEIS with Eclipse GitHub (continued) .....  | 19 |
| Figure 21: Checking Out OpenEIS with Eclipse from GitHub (continued) .....                                   | 19 |
| Figure 22: <i>Checking Out OpenEIS with Eclipse from GitHub (continued)</i> .....                            | 20 |
| Figure 23: Checking Out OpenEIS with Eclipse from GitHub (continued) .....                                   | 20 |
| Figure 24: Checking Out OpenEIS with Eclipse from GitHub (continued) .....                                   | 21 |
| Figure 25: Configuring PyDev.....  | 22 |
| Figure 26: Configuring PyDev (continued) .....   | 22 |
| Figure 27: Configuring PyDev (continued) .....   | 23 |
| Figure 28: Configuring PyDev (continued) .....   | 23 |
| Figure 29: Configuring PyDev (continued) .....   | 24 |
| Figure 30: Set OpenEIS as Django Project.....  | 25 |
| Figure 31: Configure Django Settings .....   | 26 |
| Figure 32: Running the OpenEIS.....  | 27 |
| Figure 33: Running an application in the command line – Creating a data map .....                            | 29 |
| Figure 34: Running an application in the command line - Creating the application configuration<br>file ..... | 30 |
| Figure 35: Running an application in the command line – Accessing data set information using<br>API.....     | 31 |
| Figure 36: Running an application in the command line – Finished Configuration File.....                     | 33 |

|  |    |
|--|----|
| Figure 37: Running an application in the command line – Eclipse Run Configurations Main (Windows OS) ..... | 34 |
| Figure 38: Running an application in the command line – Eclipse Run Configurations Arguments               | 35 |
| Figure 39: Global PyUnit test configuration.....   | 36 |
| Figure 40: Run Configuration for individual tests .....  | 37 |
| Figure 41: Running the tests .....   | 38 |
| Figure 42: Console output from executing test_greenbutton.py test .....                                    | 39 |
| Figure 43: PyUnit output from executing test_greenbutton.py test .....                                     | 39 |
| Figure 44: API Root.....   | 48 |
| Figure 45: API project page .....  | 48 |
| Figure 46: API project page (continued).....   | 49 |
| Figure 47: API File List .....   | 50 |
| Figure 48: API data map (top) .....  | 51 |
| Figure 49: API data map (bottom) .....   | 52 |
| Figure 50: API dataset page .....  | 53 |
| Figure 51: API analyses page – application configuration .....   | 55 |
| Figure 52: Creating a new Driven Application .....   | 73 |
| Figure 53: Creating a new Driven Application from the Project Explorer .....                               | 74 |
| Figure 54: Creating a new Driven Application (continued).....  | 75 |
| Figure 55: Creating an example driver application.....   | 83 |
| Figure 56: Creating a new Driver Application from the PyDev Package Explorer .....                         | 83 |
| Figure 57: Creating a new Driver Application (continued) .....   | 84 |

# 1 Introduction

OpenEIS (open energy information system) is an open-source software tool for analyzing building energy and operational data to identify improvement opportunities. Continuous monitoring and analysis can increase whole building energy efficiency by up to 20%. However, most building managers and operators do not have cost-effective access to commercial tools and algorithms for identifying potential savings. Conversely, diagnostic methods developed by the Department of Energy's National Laboratories, by university researchers, and by publicly funded research projects do not have a common distribution path by which to put new tools in the hands of energy managers.

In response, OpenEIS was designed to provide standard methods for authoring, sharing, testing, using, and improving algorithms for operational building energy efficiency. The OpenEIS strategy is to get the market to validate and implement state-of-the-art analytical and diagnostic algorithms. This, in turn, should create market demand for control system manufacturers and integrators serving small and medium commercial customers, as well as for commercial tool offerings.

One of the largest obstacles to data analytics (including but not limited to building energy and efficiency related analysis) is overcoming incomplete and non-uniform raw performance or consumption data. Few (if any) tools allow a user to merge data from multiple sources (with possible gaps in the data) and obtain one uniform dataset. OpenEIS provides this functionality with multiple aggregation filters for use in merging data, aggregating trend data from high sampling frequency to a lower sampling frequency, and other manipulations to create datasets suitable for direct analysis.

Although OpenEIS was initially developed for building systems (air-handling units (AHUs), packaged rooftop air conditioners (RTUs), chilled water distribution systems, hot water distribution systems, and zone terminal-box and lighting systems), it can be extended to include analysis tools for other types of systems and devices (i.e., nearly any device or system where data is trended). OpenEIS is compatible with most operating systems and can be run on Windows, Mac, and Linux operating systems. OpenEIS can also be deployed as a service in the Cloud.

Initially, several building energy-efficiency applications were developed by Lawrence Berkeley National Laboratory (LBNL) and Pacific Northwest National Laboratory (PNNL), and included as part of the OpenEIS software package. These "seed" applications serve as a starting point with the hope that new applications would be added by possible commercial and/or academic developers. This document is intended to serve as a guide for developers who wish to create or extend applications available with the OpenEIS.

## 1.1 Document Background

This document is targeted at both developers building applications on top of OpenEIS and those wishing to understand/modify the supporting code itself. It is highly recommended that the companion guide (OpenEIS: Users Guide) be read first to get background on terminology, workflow, intended use, etc. of the platform.

## 1.2 Document Conventions

The following typographical conventions are used in this document:

- A class or function name will be in Consolas 10 point bold font. For example:
  - **Myclass** inherits from the **Baseclass**.
- A file or application name will be Times New Roman 12 point bold font. For example:
  - This code can be found in the **example.py** file.
- Terminal commands (command prompt) will be Times New Roman 12 point bold font and will be called out in text as terminal commands.
- File paths will be Times New Roman 12 point italic font. For example:
  - The file **example.py** is located at: *C:\Users\<USER NAME>\openeis\*
- User supplied parameters will be Times New Roman 12 point regular font. For example:
  - level – logging level of the message of a message.
- Python Code blocks will be Consolas 10 point font and are color coded using the default schema from the Eclipse Integrated Development Environment plugin PyDev.

The default PyDev color code is as follows:

- Code – color
- Decorator – color
- Numbers – color
- Keywords – color
- Strings – color
- Comments – color
- Mathematical operators – color

## 2 Installation of the OpenEIS

This section will detail the steps required to install the OpenEIS. This includes installation of OpenEIS software dependencies, bootstrapping the OpenEIS (building the project), and installation of an integrated development environment. Links to software dependencies and supplemental information are provided, where applicable.

### 2.1 Project Software Requirements

The OpenEIS can be used on nearly any operating system (OS) (i.e., Windows, Mac, or Linux). OpenEIS is written in Python, which is a general use, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. Many of the applications and tools developed for the OpenEIS are written in Python. The OpenEIS requires Python 3.3 or greater. For Windows, Python is available from the following site:

<https://www.python.org/downloads/>

Python is typically included in most Linux distributions (e.g., Ubuntu) or one can install Python with the Linux distributions respective package manager.

Git is a distributed revision control system (source code control) with an emphasis on speed, data integrity, and support for distributed, non-linear work flows. Since its introduction in 2005, Git has become the most widely adopted version control system for software development.

Installation of Git is recommended but not required. Git allows one to pull and incorporate the latest updates and code in the OpenEIS without the need to reconfigure, rebuild, or recreate applications. Also, developers can use Git to incorporate their applications into the OpenEIS and make them available to the public.

Git is available from the following site:

<http://git-scm.com/downloads>

On a Linux OS, Git can be installed with the Linux distribution's respective package manager.

While an Integrated Development Environment (IDE) is not required, it can be very useful for developing applications or modifying OpenEIS itself. Eclipse is a powerful open source IDE that can be used for development of OpenEIS tools and applications. Eclipse contains a base Workspace and an extensible plug-in system for customizing the environment. Through various plug-ins, Eclipse may also be used to develop applications in many other programming languages (e.g., C, C++, Fortran, Java, JavaScript, Perl, PHP, Prolog, R, Ruby, etc.).

Eclipse is available from the following site:

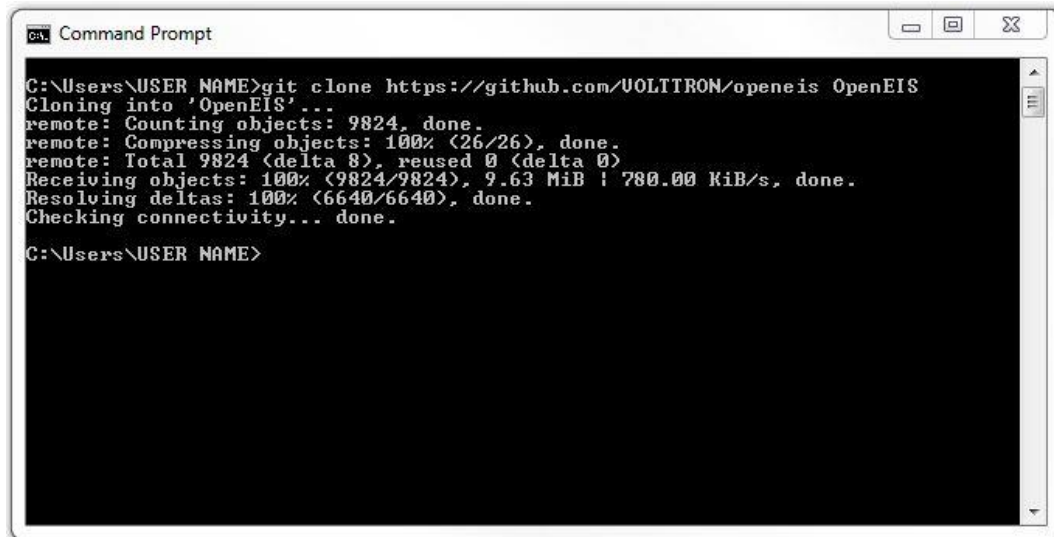
<https://eclipse.org/downloads/>

Eclipse requires a Java Runtime Environment (JRE). Instructions for installing a JRE are provided on the Eclipse website.

The OpenEIS project is hosted on GitHub. A GitHub account is not required to work with OpenEIS unless you want to contribute back to the project. A GitHub account can be created by visiting <https://github.com> and selecting “Sign up for GitHub.” To contribute to the OpenEIS, create a pull request at <https://github.com/VOLTTRON/openeis>. The development team will review the contributions.

## 2.2 Building the Project

After the OpenEIS project dependencies are installed, the OpenEIS project must be downloaded (use Git to clone the project or download the OpenEIS as a zip file) and built. The OpenEIS clone URL is <https://github.com/VOLTTRON/openeis>. Figure 1 shows the command prompt after successfully cloning the OpenEIS project.



```
C:\Users\USER NAME>git clone https://github.com/VOLTTRON/openeis OpenEIS
Cloning into 'OpenEIS'...
remote: Counting objects: 9824, done.
remote: Compressing objects: 100% (26/26), done.
remote: Total 9824 (delta 8), reused 0 (delta 0)
Receiving objects: 100% (9824/9824), 9.63 MiB | 780.00 KiB/s, done.
Resolving deltas: 100% (6640/6640), done.
Checking connectivity... done.
C:\Users\USER NAME>
```

Figure 1: Command Prompt after Successfully Cloning OpenEIS Project from GitHub

### 2.2.1 Building the Project / Bootstrap OpenEIS

We recommend creating a directory for the project; this guide will assume the project is located at and file paths will be relative to this directory:

`C:\Users\<USER NAME>\OpenEIS\`

- Open a command prompt (terminal) and navigate to this directory (OpenEIS directory). Run the following command:
  - **python bootstrap.py**
    - Note: if you have multiple versions of Python installed, you may need to use “python3 bootstrap.py”

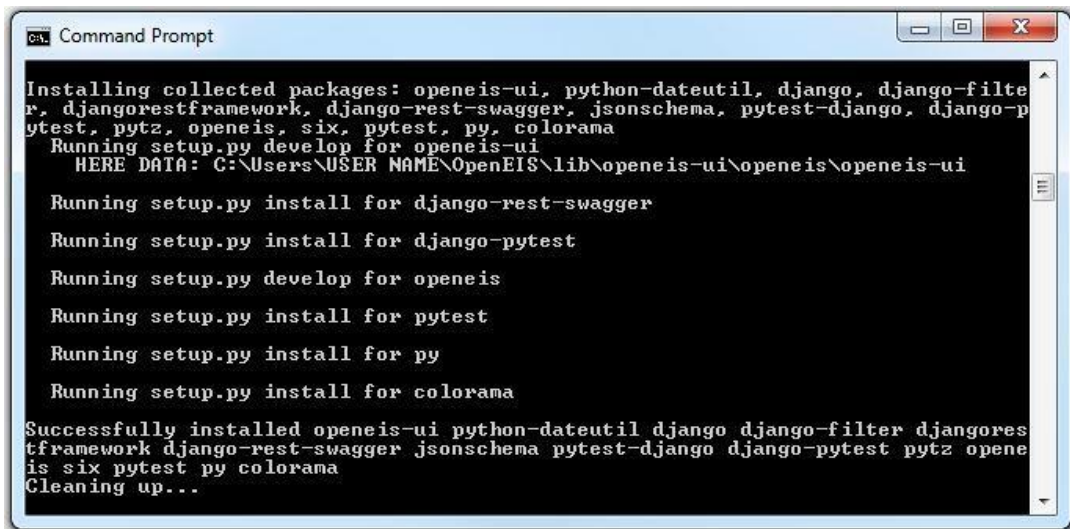
If the command prompt output displays the following message:

- Python is not recognized as an internal or external command, operable program or batch file.

Then, either Python is not installed or the Python interpreter was not added to the system path. Typically, this happens automatically upon installation of Python. Instructions on manually adding the Python interpreter to the system path can be found at:

<https://docs.python.org/2/using/windows.html>

If the bootstrap completes successfully, the command prompt should appear similar to Figure 2 with the last line of text in the prompt displaying “Cleaning up...”



```
Command Prompt

Installing collected packages: openeis-ui, python-dateutil, django, django-filter, djangorestframework, django-rest-swagger, jsonschema, pytest-django, django-pytest, pytz, openeis, six, pytest, py, colorama
Running setup.py develop for openeis-ui
HERE DATA: C:\Users\USER NAME\OpenEIS\lib\openeis-ui\openeis\openeis-ui

Running setup.py install for django-rest-swagger
Running setup.py install for django-pytest
Running setup.py develop for openeis
Running setup.py install for pytest
Running setup.py install for py
Running setup.py install for colorama

Successfully installed openeis-ui python-dateutil django django-filter djangorestframework django-rest-swagger jsonschema pytest-django django-pytest pytz openeis six pytest py colorama
Cleaning up...
```

Figure 2: Command Prompt after Successful Completion of OpenEIS Bootstrap

## 2.2.2 Installing Numpy (Windows Only)

The installation of Numpy on Windows is not as easily accomplished as it is on Unix type systems. This section shows how to install Numpy on a Windows OpenEIS environment.

1. Download a prepackaged wheel (.whl) file from <http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy> (make sure to match the Numpy version with the Python interpreter installed on your system). The rest of this section assumes you downloaded the Numpy wheel to:

`C:\Users\<USER NAME>\OpenEIS\`

2. Open a command prompt (terminal) and navigate to the OpenEIS directory.
3. Activate the OpenEIS (execute the following command from the terminal):

➤ `.\env\Scripts\activate`

4. Install Numpy into the OpenEIS (execute the following command from the terminal):

➤ `pip install numpy.whl`

Where “numpy.whl” is the file name of the Numpy wheel downloaded in step 1 (current section).

## 2.3 Install Eclipse and Eclipse Plug-ins

Download the latest version of Eclipse. Extract the provided zip file to the desired installation directory. When launching Eclipse for the first time, one will be prompted to select a Workspace location, as shown in Figure 3. The Workspace is the location on your system where Eclipse stores one or more projects. You can also have multiple Workspaces, each with one or more projects.

Pydev and EGit are plug-ins (software add-ons) that add increased functionality to Eclipse and make development within OpenEIS more convenient.

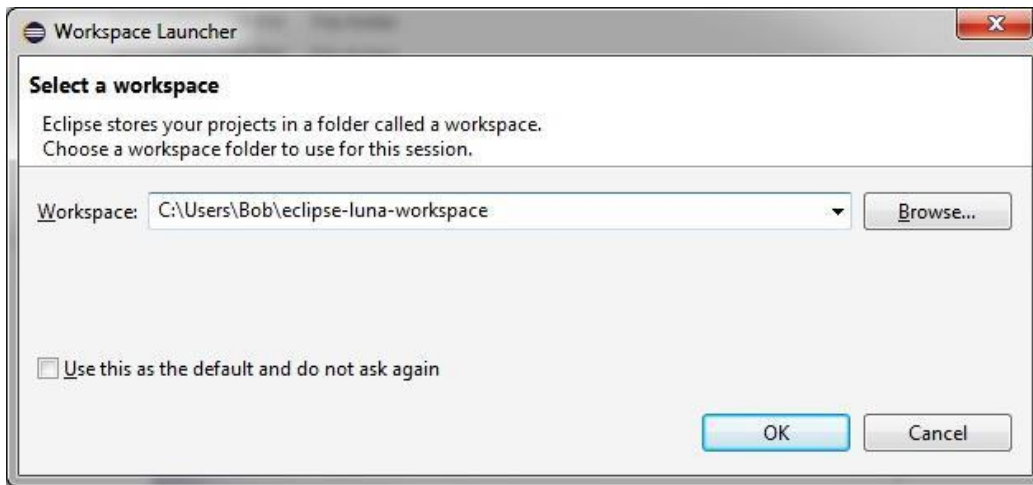


Figure 3: Launching Eclipse and Selecting a Workspace

### 2.3.1 Installation of PyDev Plug-in for Eclipse

PyDev is a third party plug-in for Eclipse. It is an IDE used for programming in Python and supports code refactoring, graphical debugging, code analysis and many other features. More information on PyDev is available from the following site:

<http://pydev.org>

To install the PyDev plug-in, in Eclipse select:

➤ Help -> Eclipse Marketplace...

- In the search bar, enter: PyDev
- Select PyDev from the list of search results, and click “Install”.

On the next screen, accept any license agreements and finish installing PyDev.

### 2.3.2 Install PyDev Alternate Steps

After installing Eclipse, you should add the PyDev plug-in to the environment. In Eclipse select:

- Help -> Install New Software (Figure 4).

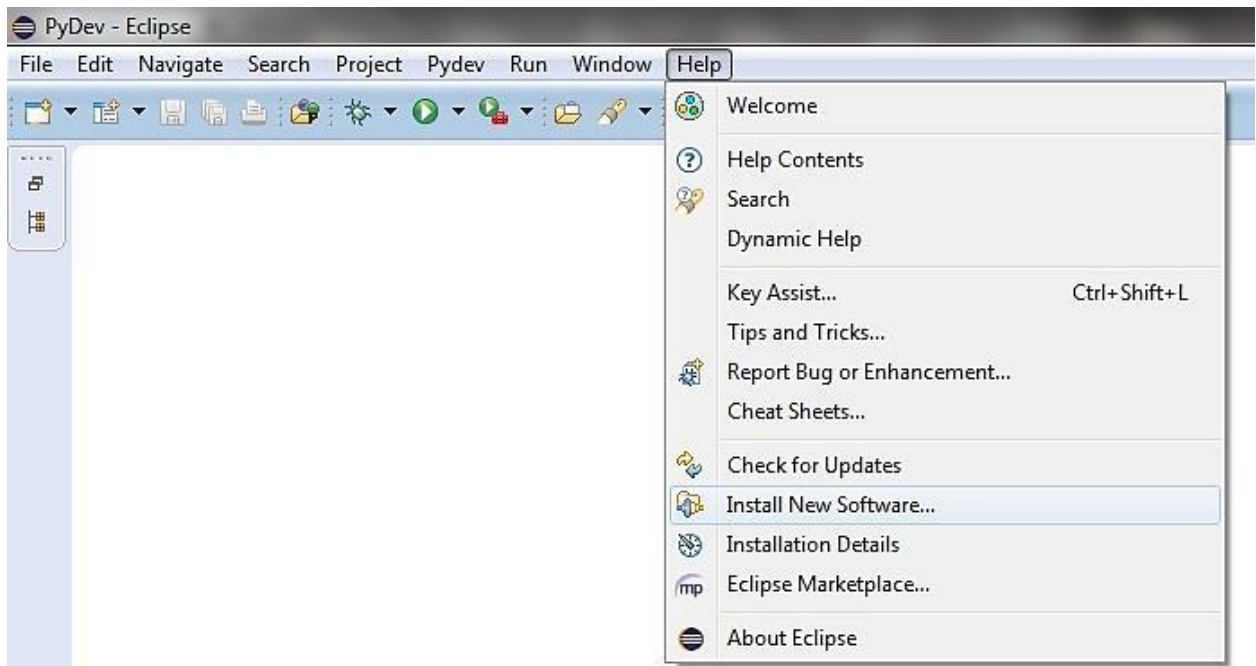


Figure 4: Installing PyDev Plug-in for Eclipse

- Click on the "Add" button, as shown in Figure 5.

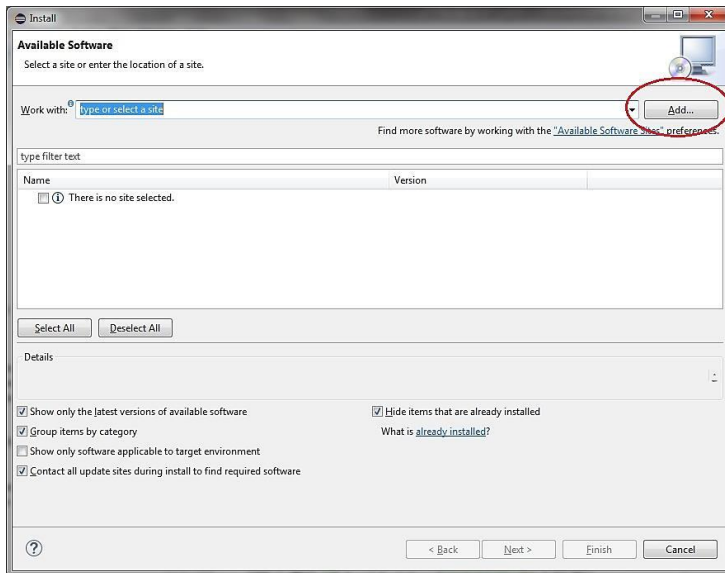


Figure 5: Installing PyDev Plug-in for Eclipse (continued)

- As shown in Figure 6, enter the following:
- For name use: PyDev
  - For location: <http://pydev.org/updates>

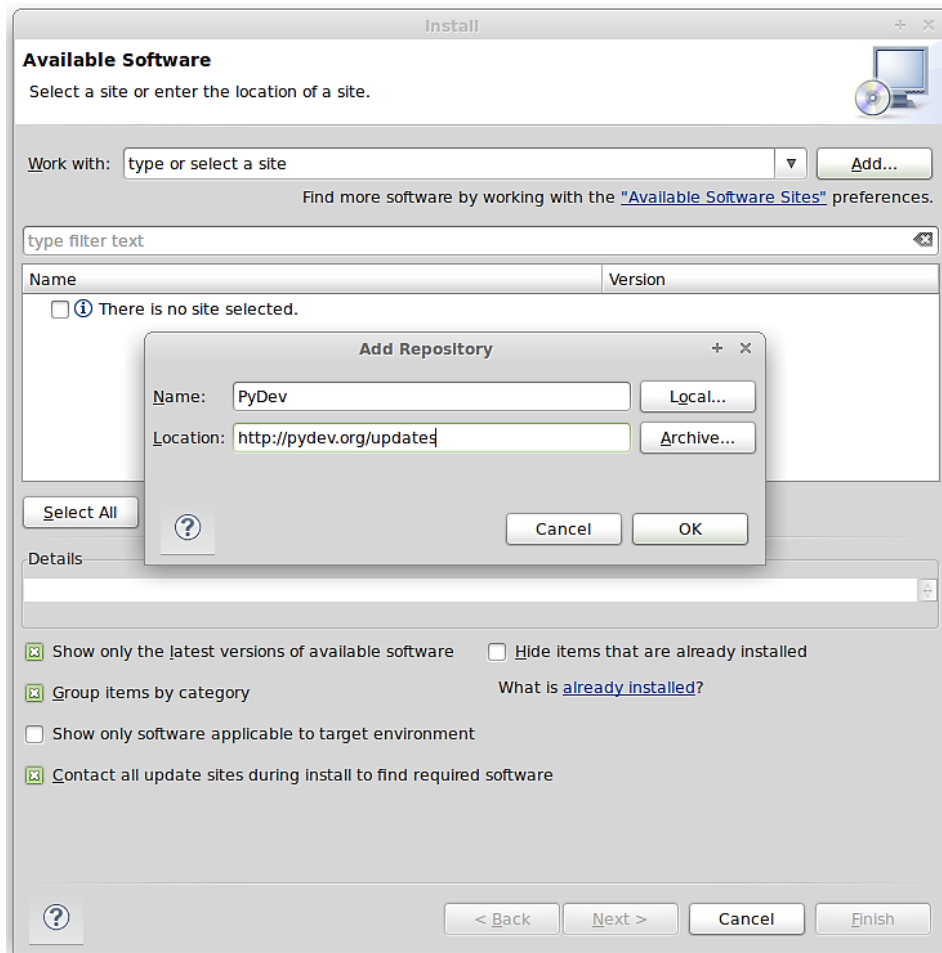


Figure 6: Installing PyDev Plug-in for Eclipse (continued)

A list of available PyDev software will display.

- Select PyDev for Eclipse (PyDev Mylyn Integration is optional) -> Next (Figure 7).

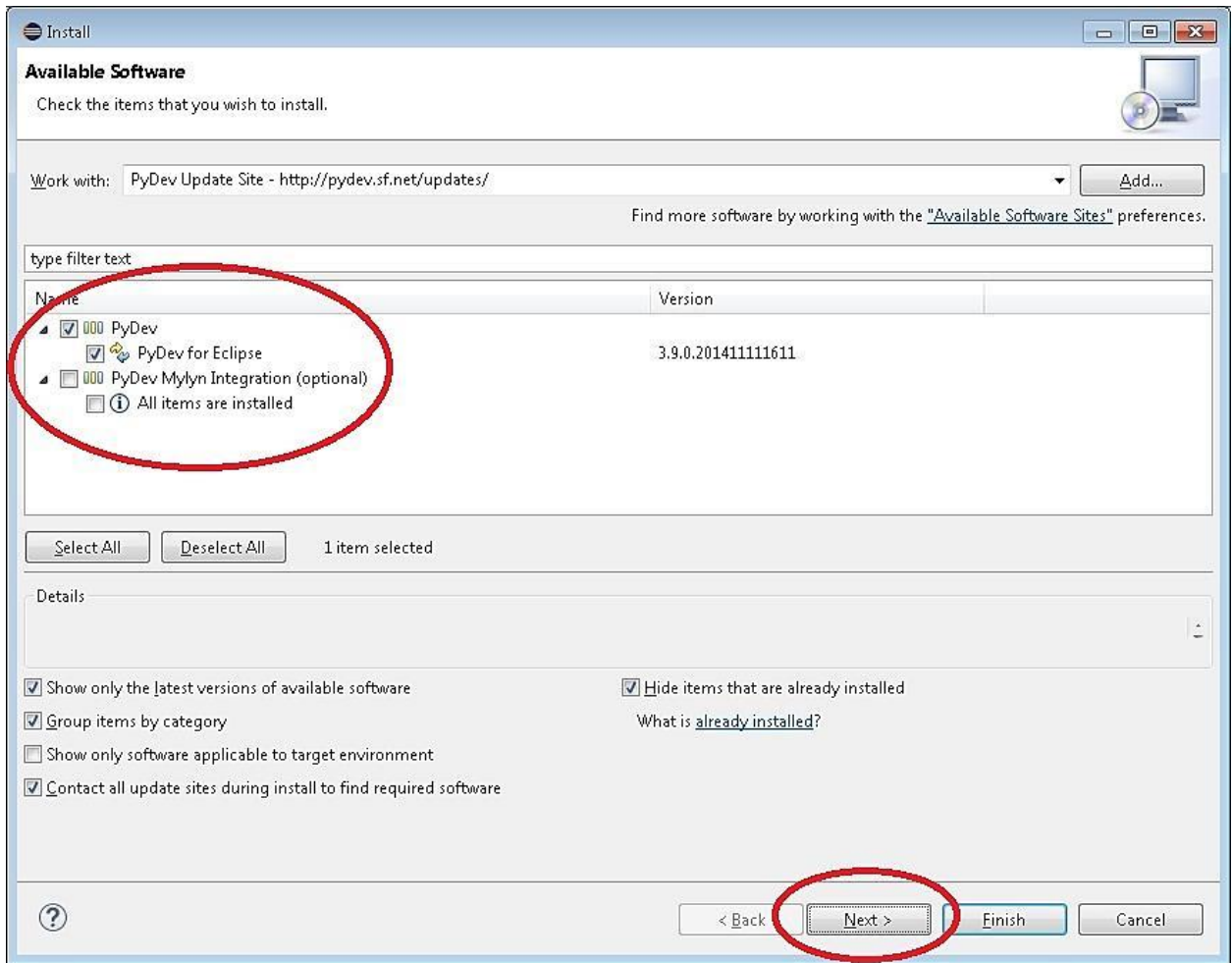


Figure 7: Installing PyDev Plug-in for Eclipse (continued)

- Accept the license agreement and finish installing PyDev (Figure 8).

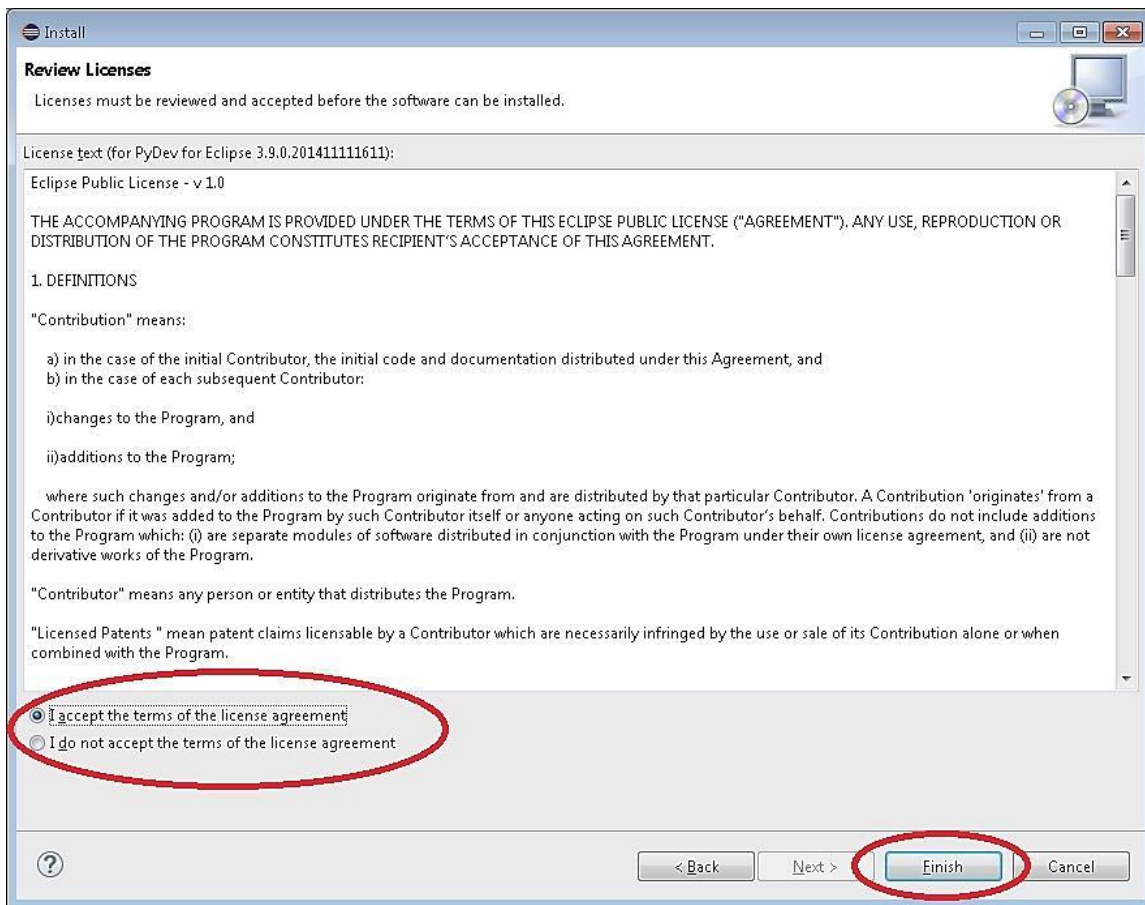


Figure 8: Installing PyDev Plug-in for Eclipse (continued)

### 2.3.3 Installation of EGit Plug-in for Eclipse

There is a plug-in available for Eclipse that makes development more convenient (note: you must have Git (Sections 2.1) installed on the system and have built the project (Section 2.2.1).

- As shown in Figure 9:
  - Select: Help -> Install New Software

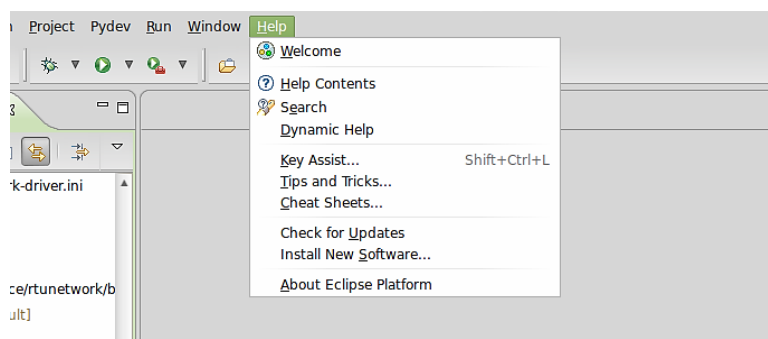


Figure 9: Installing Eclipse EGit Plug-in

- Click on the "Add" button, as shown in Figure 10.

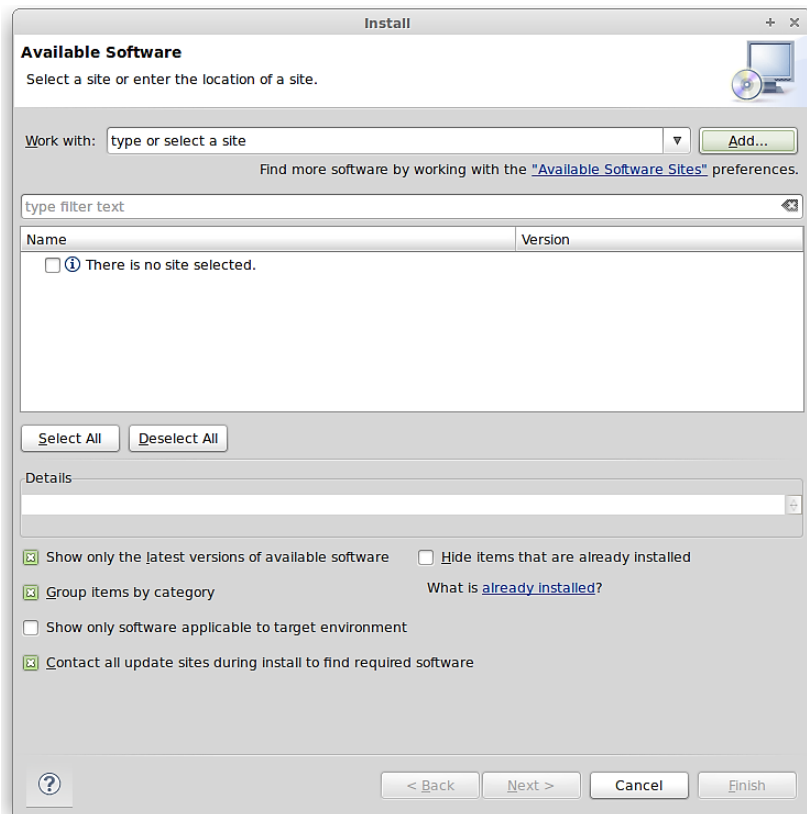


Figure 10: Installing Eclipse EGit Plug-in (continued)

- As shown in Figure 11, enter the following:
  - For name use: EGit
  - For location: <http://download.eclipse.org/egit/updates>

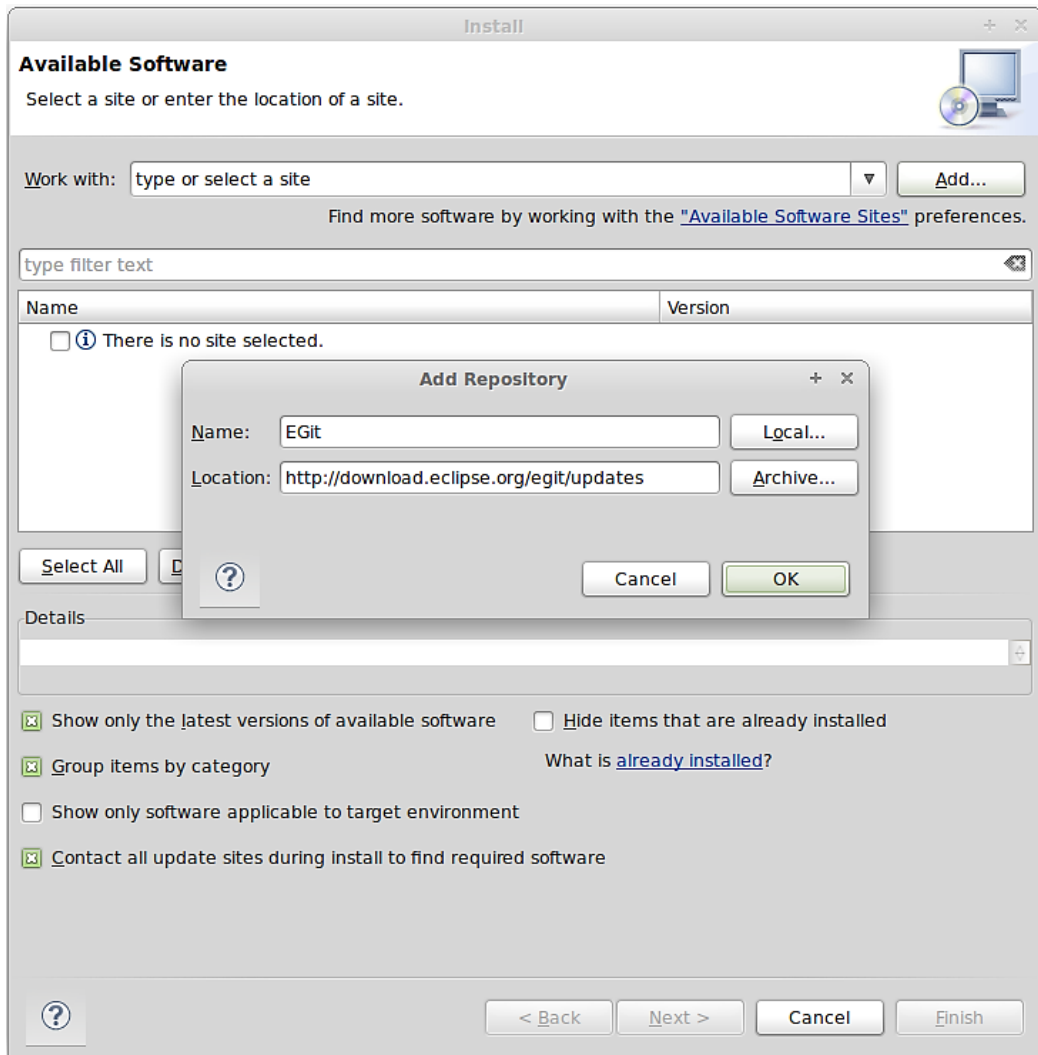


Figure 11: Installing Eclipse EGit Plug-in (continued)

- After hitting OK, check the Select All button.
- Click Next -> Agree to Terms -> Finish
- Allow Eclipse to restart

## 3 Project Configuration

After the OpenEIS software requirements have been installed, the project has been built (bootstrapped), and Eclipse and the Eclipse plug-ins have been installed (Section 2.1, Section 2.2, and Section 2.3, respectively) the OpenEIS project can be imported into Eclipse and the development environment can be configured.

### 3.1 Importing the OpenEIS into Eclipse

OpenEIS can be imported into Eclipse from an existing OpenEIS project (OpenEIS was previously checked out from GitHub) or a new download from GitHub.

#### 3.1.1 Import OpenEIS into Eclipse from an Existing Local Repository (Previously Downloaded OpenEIS Project)

To import an existing OpenEIS project into Eclipse, the following steps should be followed:

1. Select File, then import, as shown in Figure 12

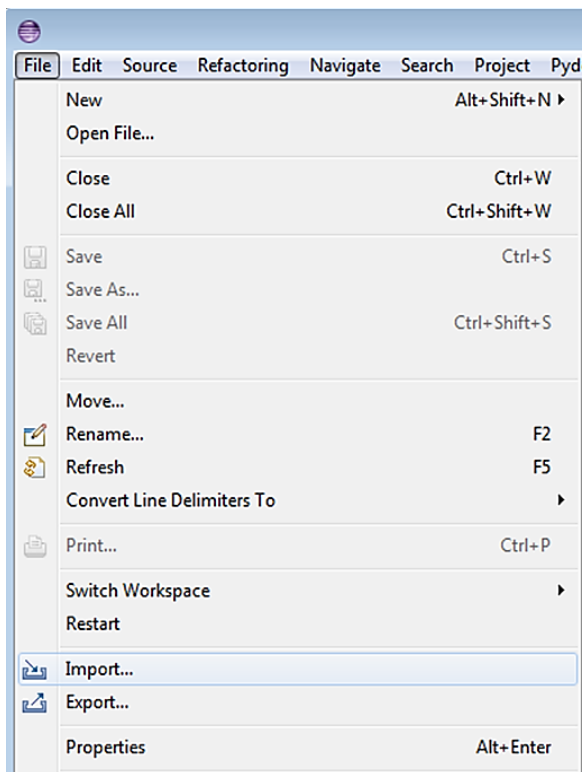


Figure 12: Importing OpenEIS with Eclipse from Local Source

2. Select Git -> Projects from Git, then click the Next button (Figure 13)

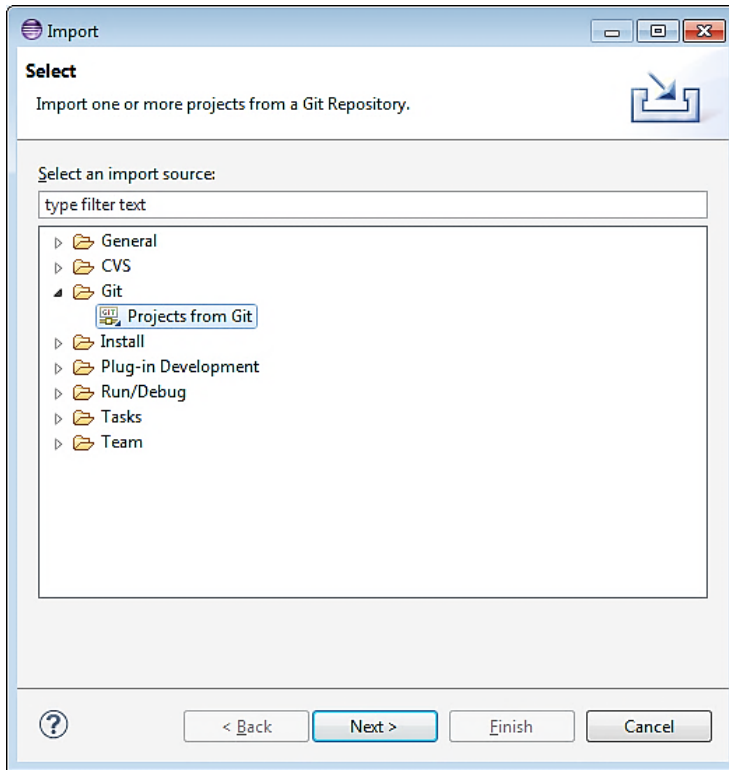


Figure 13: Importing OpenEIS with Eclipse from Local Source (continued)

3. As shown in Figure 14:

- Select Existing local repository -> Next ->

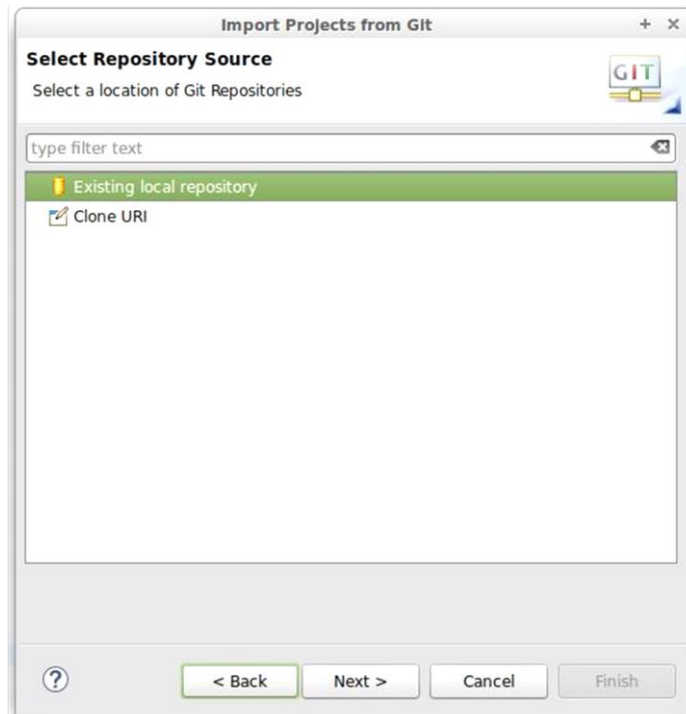


Figure 14: Importing OpenEIS with Eclipse from Local Source (continued)

4. Select Add (Figure 15)

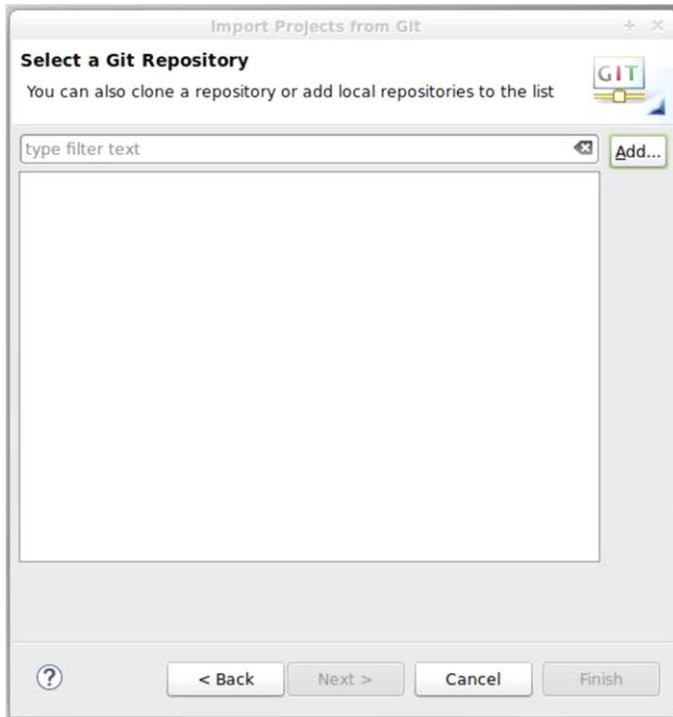


Figure 15: Importing OpenEIS with Eclipse from Local Source (continued)

5. Select Browse -> navigate to the top-level OpenEIS directory and select OK -> select Finish (Figure 16)

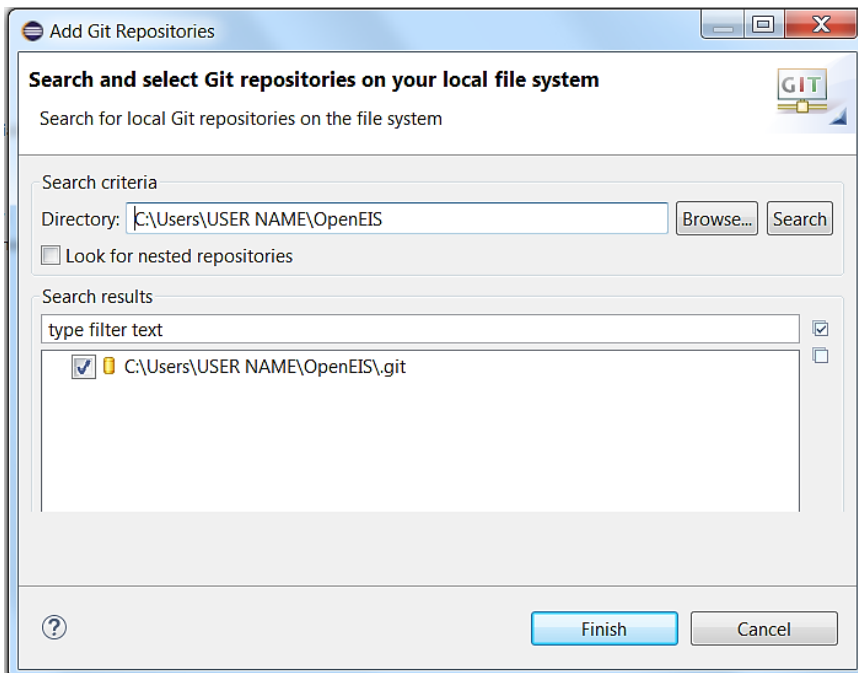


Figure 16: Importing OpenEIS with Eclipse from Local Source (continued)

6. Choose Import as general project and click Next -> Finish, the project will be imported into the work space (Figure 17)

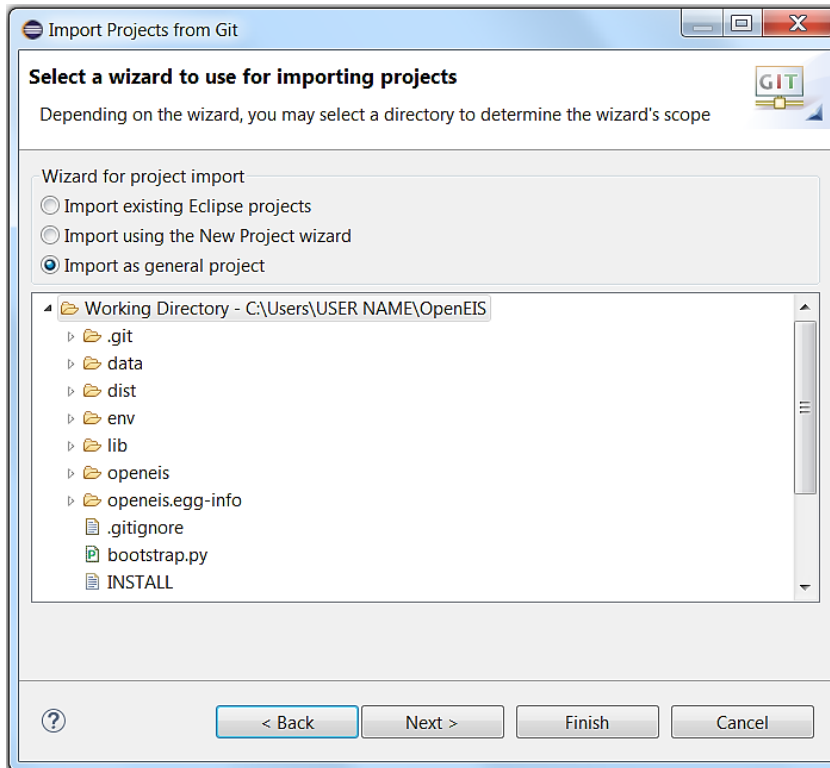


Figure 17: Importing OpenEIS with Eclipse from Local Source (continued)

### 3.1.2 Import New OpenEIS Project from GitHub

If one already has Eclipse installed and configured, then the user can import a new OpenEIS project directly from GitHub. After importing the project, one would need to bootstrap the project outside of Eclipse, as described in Section 2.2.1. To import a new OpenEIS project from GitHub into Eclipse use the following steps:

1. Select File, then Import (Figure 18)

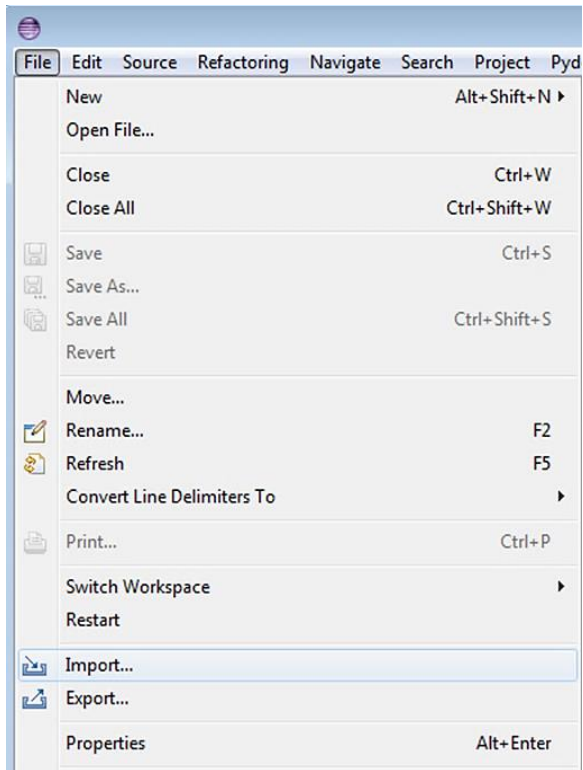


Figure 18: Checking Out OpenEIS with Eclipse from GitHub

2. Select Git -> Projects from Git, then click the Next button (Figure 19)

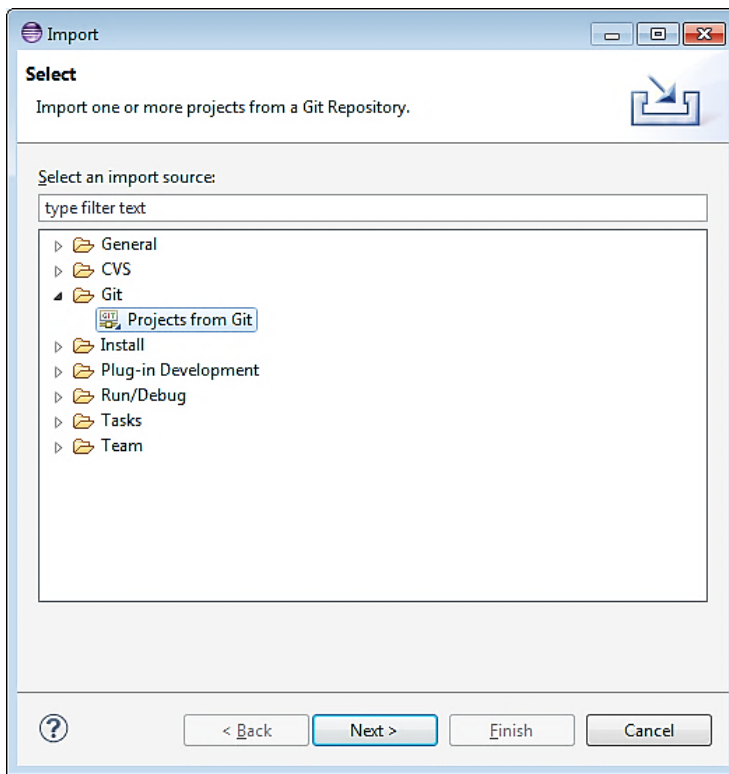


Figure 19: Checking Out OpenEIS with Eclipse from GitHub (continued)

3. As shown in Figure 20, select Clone URI -> Next >

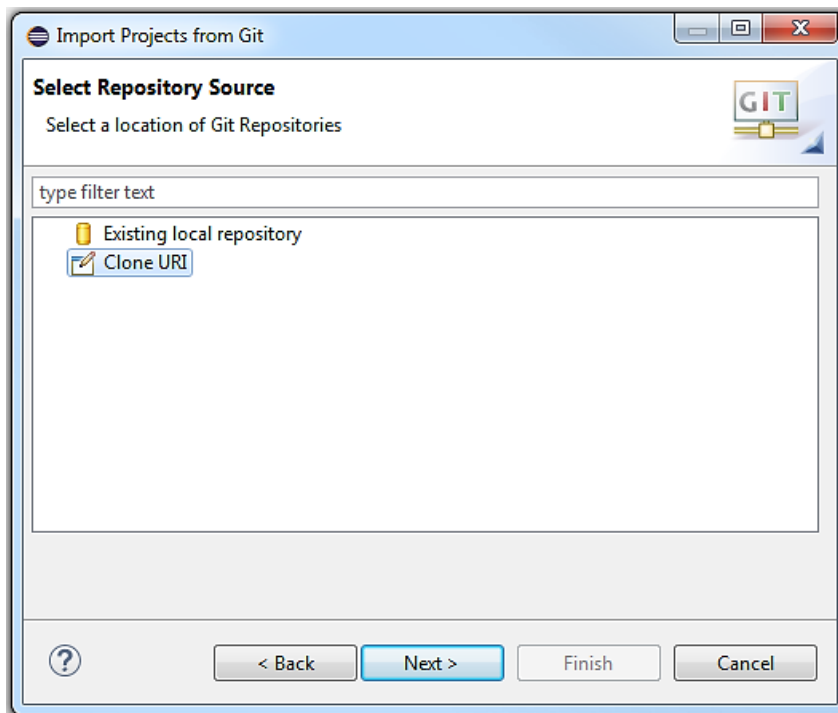


Figure 20: Checking Out OpenEIS with Eclipse GitHub (continued)

4. Fill in <https://github.com/VOLTTRON/openeis.git> for the URI, and use your GitHub account login (GitHub account username and password in the User and Password, as shown in Figure 21), select Next ->

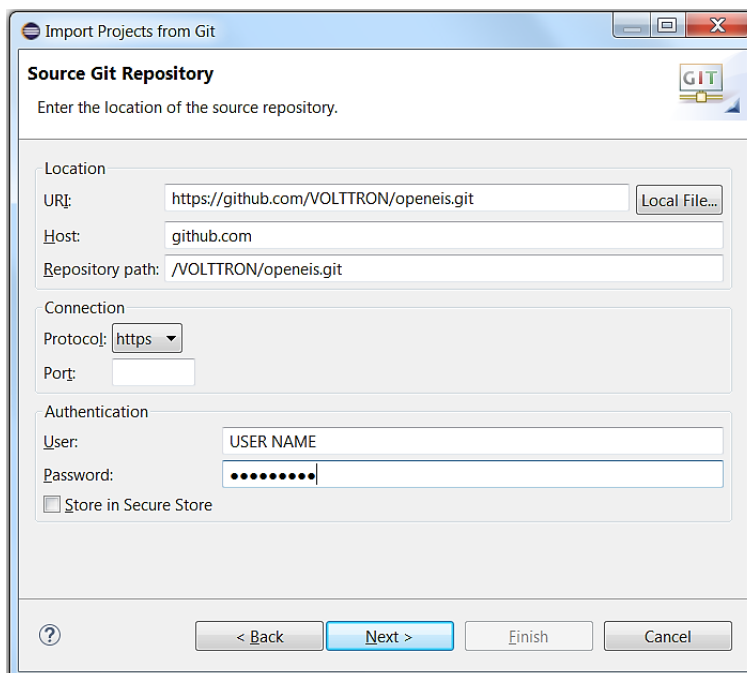


Figure 21: Checking Out OpenEIS with Eclipse from GitHub (continued)

5. Select the 2.x branch (Figure 22)

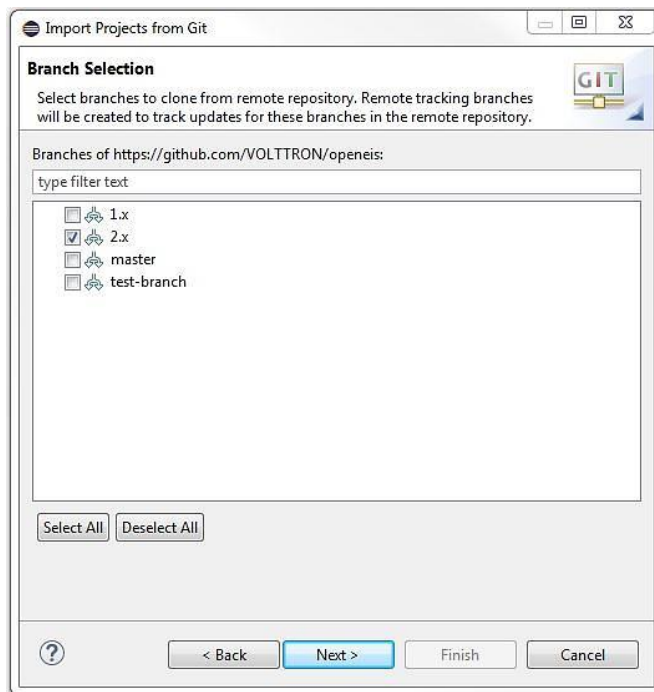


Figure 22: Checking Out OpenEIS with Eclipse from GitHub (continued)

6. Select a location to save the local repository (Figure 23)

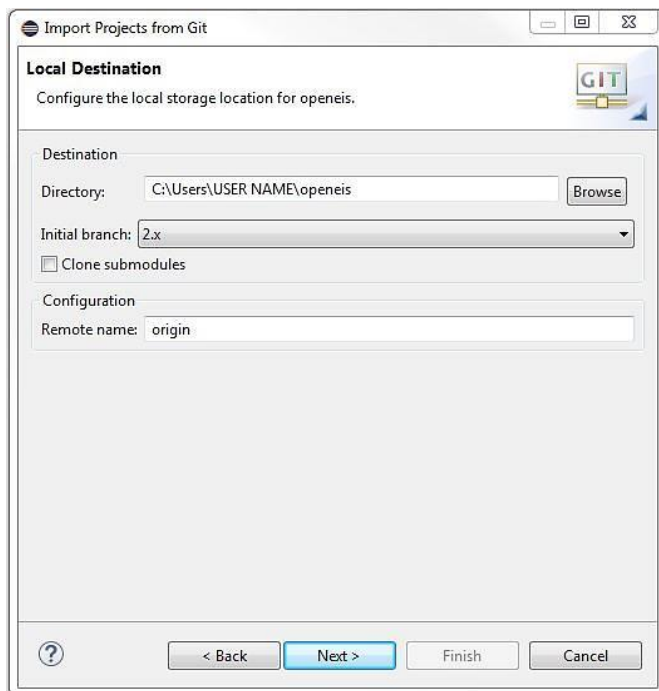


Figure 23: Checking Out OpenEIS with Eclipse from GitHub (continued)

7. Select Import as general project, select Next, then select Finish (Figure 24), the project will now be imported into the Workspace

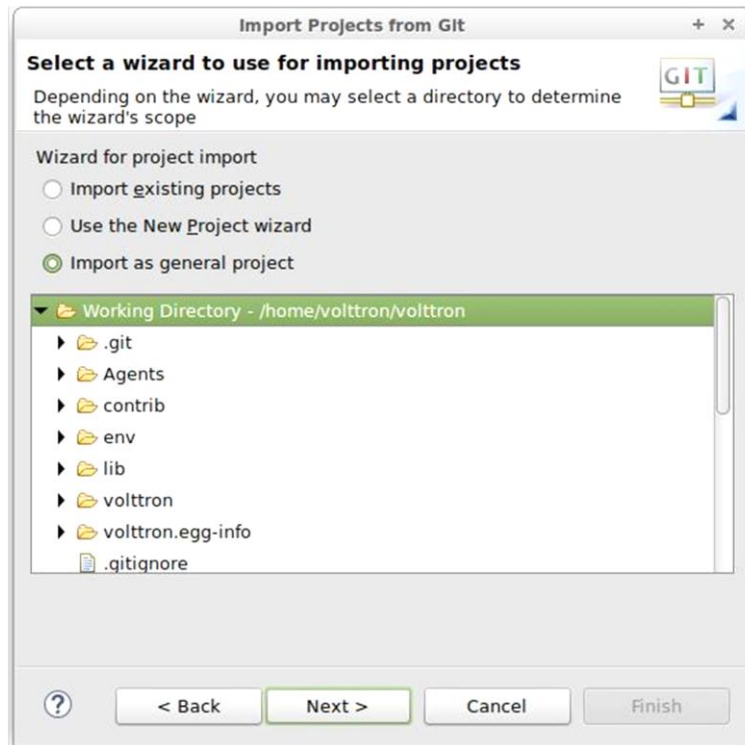


Figure 24: Checking Out OpenEIS with Eclipse from GitHub (continued)

The project must now be built (bootstrapped) outside Eclipse. Please follow the directions in Section 2.2.1. After building the file system outside Eclipse, right-click on the project name and select Refresh.

## 3.2 Configuring PyDev

PyDev must now be configured to use the OpenEIS project Python interpreter.

- In the Project/Package Explorer view on the left, right-click on the project, PyDev-> Set as PyDev Project (Figure 25)

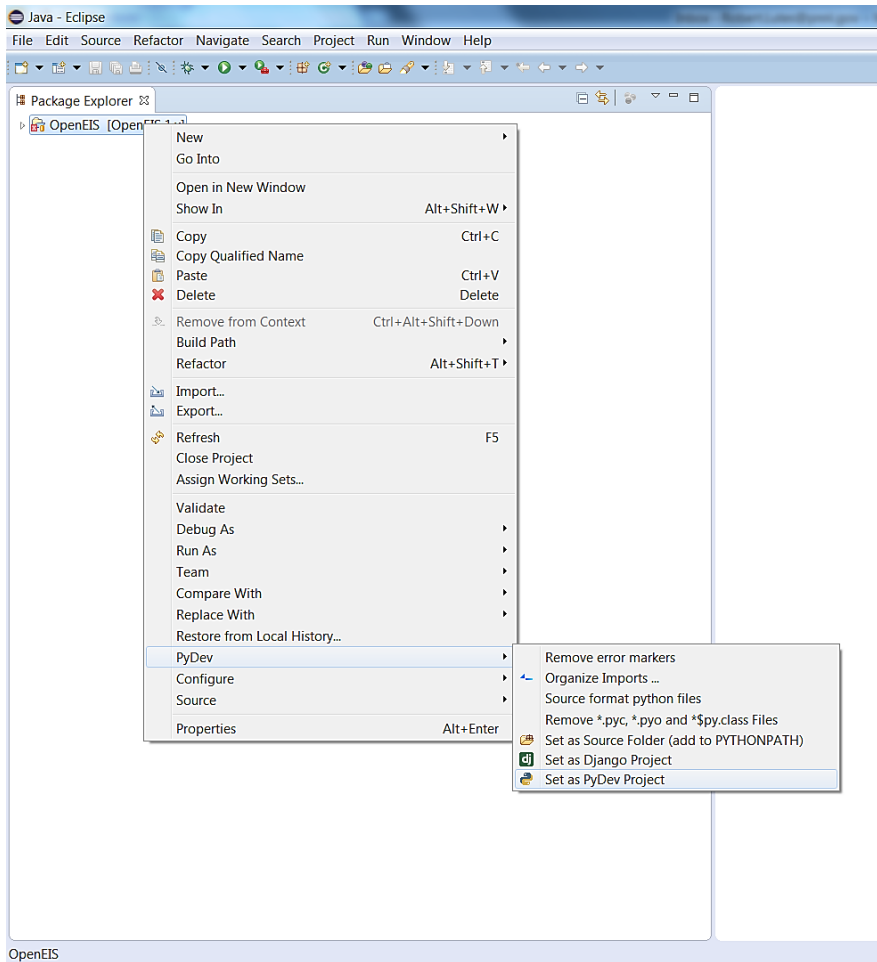


Figure 25: Configuring PyDev

- A selection box will appear, choose Manual Config (Figure 26)

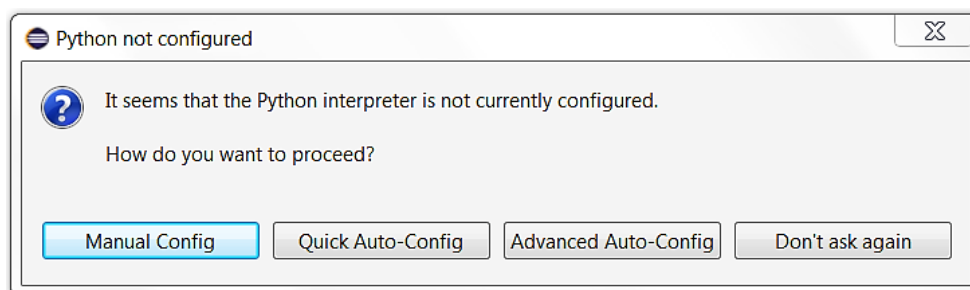


Figure 26: Configuring PyDev (continued)

- Select -> New...
  - For Interpreter Name enter: Python
  - Select browse -> navigate to the project Python interpreter : `./OpenEIS/env/Scripts/python.exe` -> select OK (Figure 27)

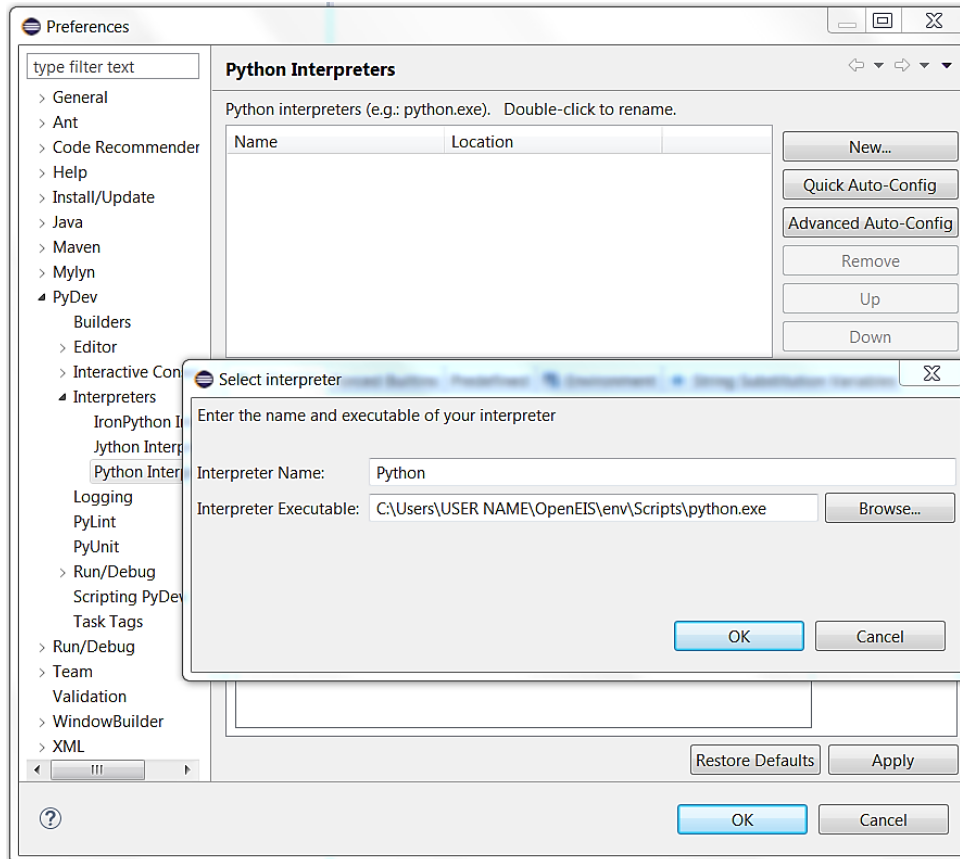


Figure 27: Configuring PyDev (continued)

- Choose Select All -> OK (Figure 28)

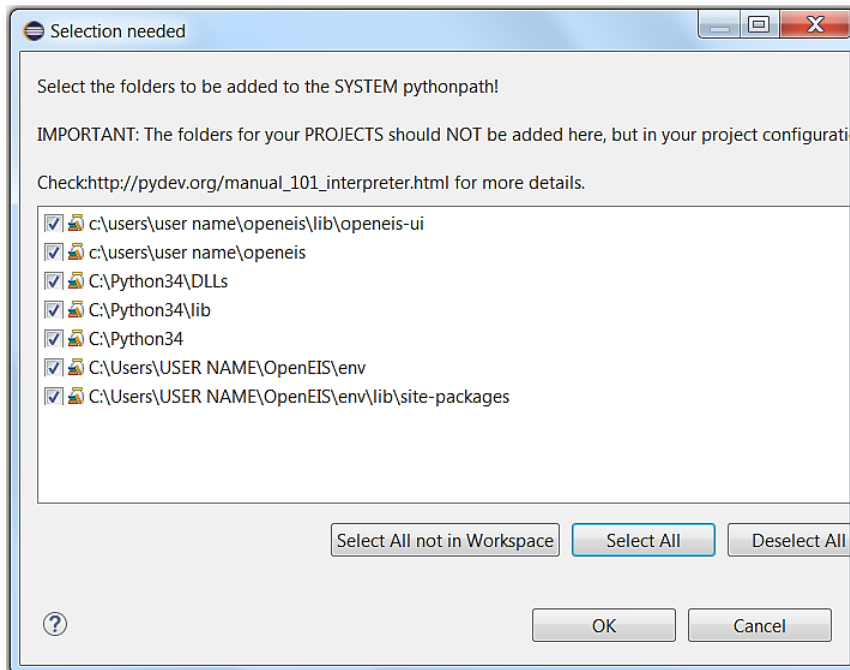


Figure 28: Configuring PyDev (continued)

- Choose Apply -> OK (Figure 29)

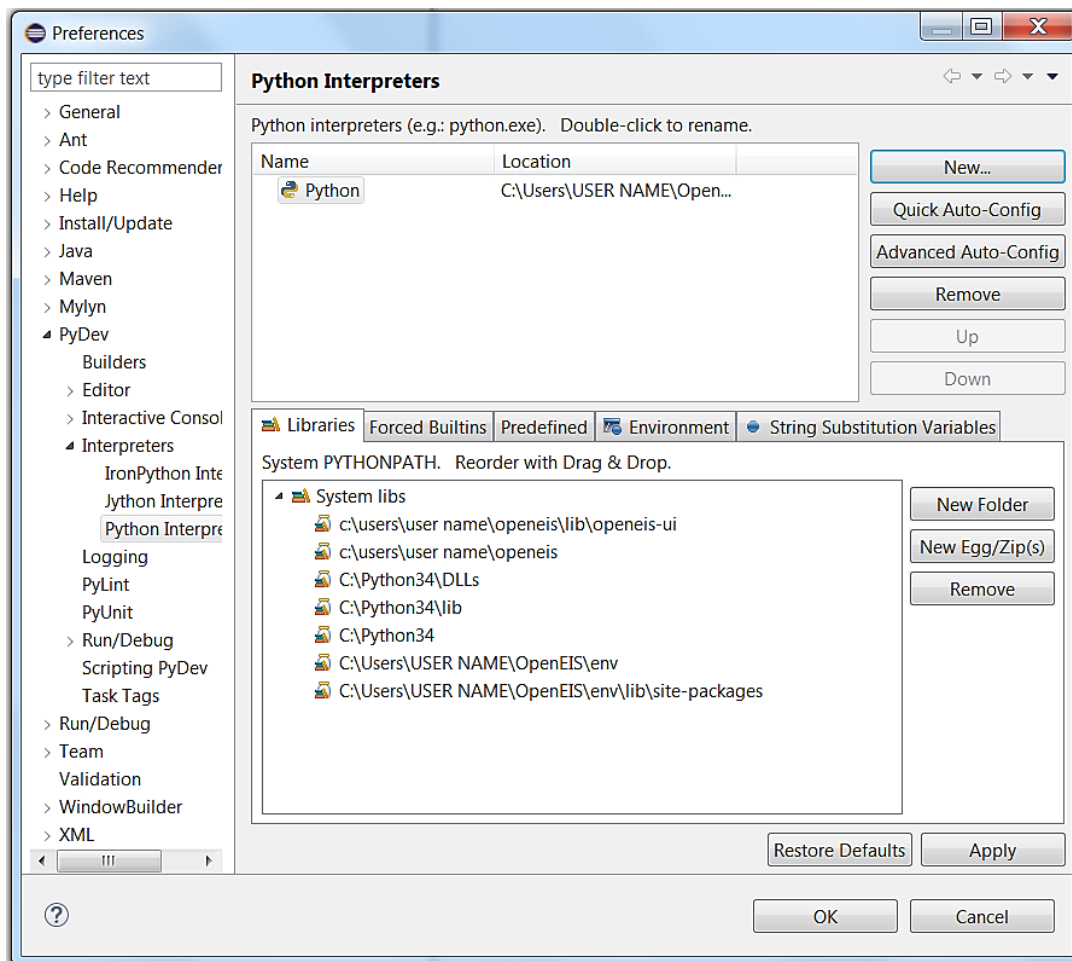


Figure 29: Configuring PyDev (continued)

- The project must be marked as a Django project inside of PyDev:
  - Right click the project in the PyDev Package Explorer -> PyDev -> Set as Django Project (Figure 320)

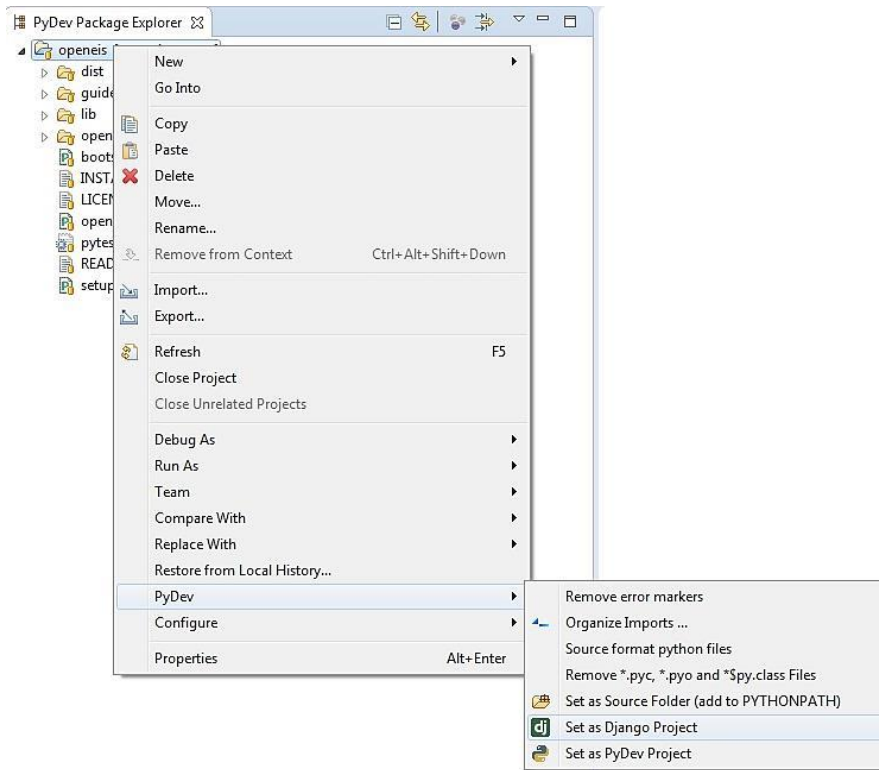


Figure 30: Set OpenEIS as Django Project

- Next, configure the Django settings:
  - Right click the project in the PyDev Package Explorer and select Properties  
Properties -> PyDev - Django
  - Select the PyDev – Django tab and configure the window, as show in Figure 31.
    - The Django settings are as follows:
      - A **DJANGO\_MANAGE\_LOCATION** string substitution variable must point to the project-relative location of manage.py
      - A **DJANGO\_SETTINGS\_MODULE** string substitution variable must contain the name of the settings module in that project

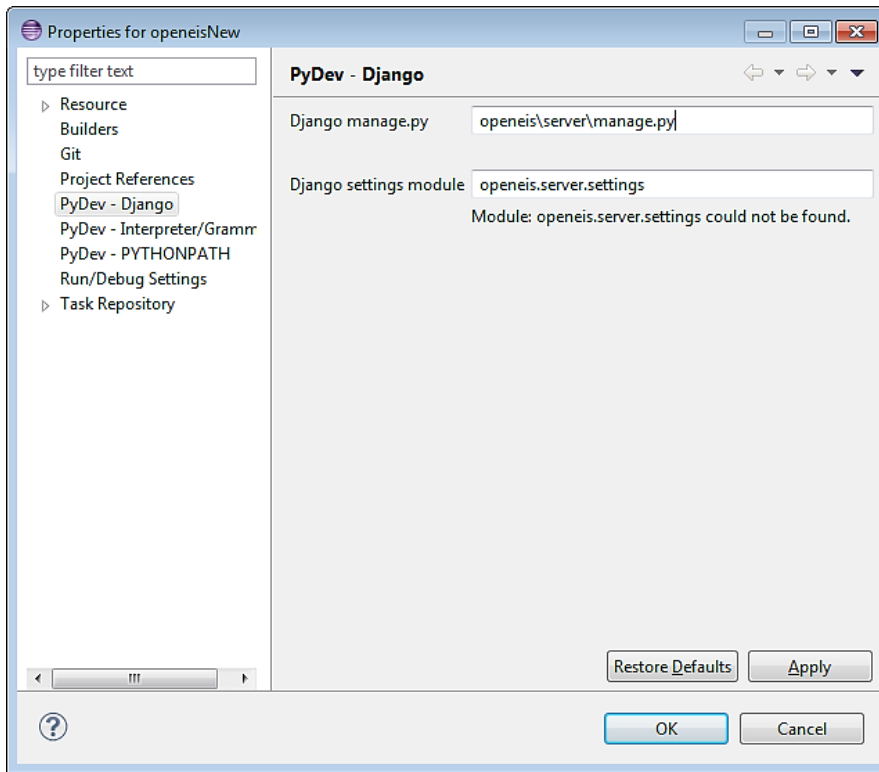


Figure 31: Configure Django Settings

## 4 Running the OpenEIS Server

Open a terminal and navigate to the OpenEIS base directory. There are two steps to start OpenEIS: activating the platform and running the server.

To activate the platform on Linux and Mac enter the following terminal commands:

- **`./env/Scripts/activate`**

Note the space after the period.

To activate the platform on Windows enter the following command in the command prompt:

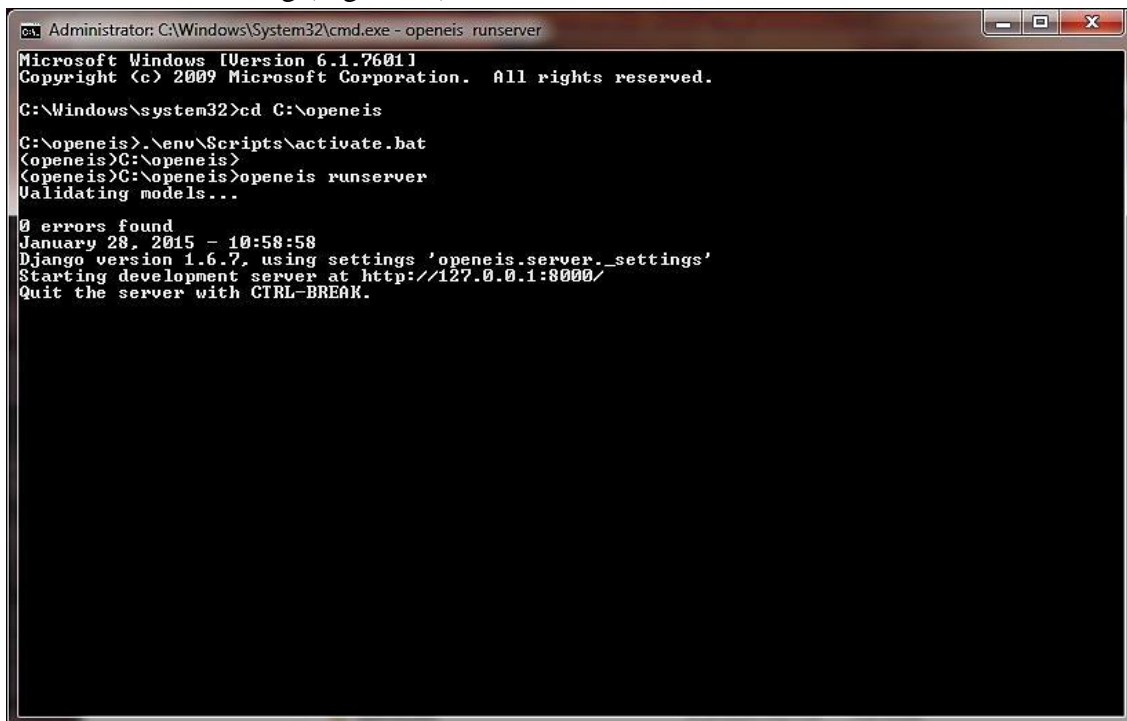
- **`.\env\Scripts\activate`**

On a successful activation, the command prompt will change to include “(openeis)” at the front.

Once the platform is activated, the server can be run. In the same terminal, run:

- **`openeis syncdb`**
- **`openeis runserver`**

The server will be running (Figure 32).



```
Administrator: C:\Windows\System32\cmd.exe - openeis runserver
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>cd C:\openeis
C:\openeis>.\env\Scripts\activate.bat
(openeis)C:\openeis>
(openeis)C:\openeis>openeis runserver
Validating models...

0 errors found
January 28, 2015 - 10:58:58
Django version 1.6.7, using settings 'openeis.server._settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Figure 32: Running the OpenEIS

## 4.1 Running Applications in Command Line

During application development, it is helpful to run applications in an environment that has debugging, syntax highlighting, and other useful development tools. This example will utilize Eclipse to run applications through the command line. The **example\_driver.py** application will be used to illustrate this process.

The steps to run an application in the command (via Eclipse) are as follows:

1. Ensure that the OpenEIS project has been activated and that the server is running (Section 4).
2. Upload time series data (input data for the application) and create a data map and dataset in the OpenEIS User Interface (OpenEIS UI) as follows:
  - Open a web browser and proceed to the OpenEIS server intended for application development (typically <http://localhost:8000>)
  - Create a new project (*OpenEIS: Users Guide* Section 4)
  - Upload the csv data file (*OpenEIS: Users Guide* Section 5)
  - Create a data map with the data uploaded in the previous step (*OpenEIS: Users Guide* Section 6)
  - Create a dataset by applying a data map to data uploaded to the OpenEIS (*OpenEIS: Users Guide* Section 7)
  - The process of creating an OpenEIS project, uploading metered data, creating a data map (map data to standardized OpenEIS names), and using data to run applications are documented, with step-by-step instructions, in the *OpenEIS: Users Guide*, which is packaged with the OpenEIS. After installation of the OpenEIS the user guide and sample data files used in the guide are located at:
    - OpenEIS installed from Microsoft Installer (.exe):
      - *docs/*
    - OpenEIS installed from GitHub:
      - *guides/*

- The **example\_driver** application code is located at:
  - `openeis/applications/example_driver.py`
- The **example\_driver** application requires building electricity usage data (OpenEIS standard name is WholeBuildingElectricity). The data file for this example is located at:
  - `guides/cli_example.csv`

As per the OpenEIS User Guide, create a data map similar to Figure 33. When completed the “New data map” page in the OpenEIS UI will appear similar to Figure 33.

PROJECTS / CLI EXAMPLE / NEW DATA MAP

## New data map

Add to data map: site building other

Building: cli\_example Rename Delete

Attributes

timezone Delete

"US/Pacific"

Add attribute

Sensors

WholeBuildingElectricity Delete

cli\_example.csv → Main Meter [kWh] (kilowatt\_hour)

Add sensor

Add under cli\_example: RTU AHU Zone Hot Water Distribution System Chilled Water Distribution System other

Data map name:  Preview Save Cancel

Figure 33: Running an application in the command line – Creating a data map

3. In `openeis/applications/` create a file named **cli\_config.ini**.

Any configurable parameters that would normally be specified by a user in the OpenEIS UI must be added to the configuration file.

4. Create three sections in the configuration file: `global_settings`, `application_config`, and `inputs` with each heading enclosed by square brackets (Figure 34).

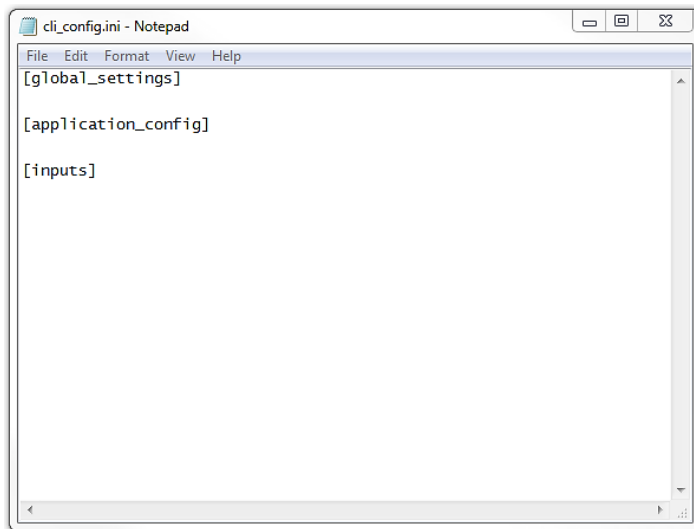


Figure 34: Running an application in the command line - Creating the application configuration file

5. In the `global_settings` section of the configuration file created in step 3, add the following lines:

```
application=example_driver
dataset_id=49
debug=true
```

`application` – parameter specifies the file name of the application, minus the file suffix.

`dataset_id` – corresponds to the value “id” for the dataset intended for use by the application (top of Figure 35 “id” is specified as 49).

`debug` – optional parameter, if set to “true” debug information will be output to file.

6. Open a browser and proceed to <http://yourserver/api/datasets> (when running the OpenEIS locally: <http://localhost:8000/api/datasets>).
7. Locate the dataset that was just created (the project and building were named “cli\_example” during the data mapping process). Figure 35 shows the dataset as accessed via the browser API.

```

"download_url": "http://localhost:8000/api/datasets/49/download",
"id": 49,
"project": 4,
"name": "cli_example - 2/10/15 2:42 PM",
"map": 50,
"start": "2015-02-10T22:42:37.654Z",
"end": "2015-02-10T22:42:38.875Z",
"datamap": {
  "files": {
    "0": {
      "timestamp": {
        "columns": [
          0
        ]
      },
      "signature": {
        "headers": [
          "Date",
          "Hillside OAT [F]",
          "Main Meter [kWh]",
          "Gas [kBtu]"
        ]
      }
    }
  },
  "sensors": {
    "cli_example/WholeBuildingElectricity": {
      "file": "0",
      "type": "WholeBuildingElectricity",
      "unit": "kilowatt_hour",
      "column": "Main Meter [kWh]"
    },
    "cli_example": {
      "attributes": {
        "timezone": "US/Pacific"
      },
      "level": "building"
    }
  },
  "version": 1
}

```

Figure 35: Running an application in the command line – Accessing data set information using API

8. Examine the **required\_input** method for the **example\_driver** application. For each key in the dictionary, one will need to specify an entry for a sensor corresponding to the dataset created in step 2.

```

@classmethod
def required_input(cls):
    #Called by UI

```

```

'''Returns a dictionary of required data.'''
return {
    'electricity': InputDescriptor('WholeBuildingElectricity',
                                   'Building Electricity')
}

```

Figure 35 shows the sensor path: “cli example/WholeBuildingElectricity”

9. Add the following line to the configuration file under the inputs section (this specify what data within the OpenEIS the application will have access to):

```
electricity=cli_example/WholeBuildingElectricity
```

10. Examine the **get\_config\_parameters** method. For each key in the dictionary, one will need to specify a configuration parameter in the configuration file.

```

@classmethod
def get_config_parameters(cls):
    # Called by UI
    return {
        "building_name": ConfigDescriptor(str, "Building Name", optional=True)
    }

```

The **example\_driver** application takes one configuration parameter “building\_name”, which is specified as a string (str). This parameter is optional.

11. Add the configuration parameters to the application\_config section of the configuration file, as shown in the following example (“cli\_example” is the building name input to the application for use in its analysis):

building\_name=“cli\_building”

The configuration file is now complete and should appear as follows (Figure 36):

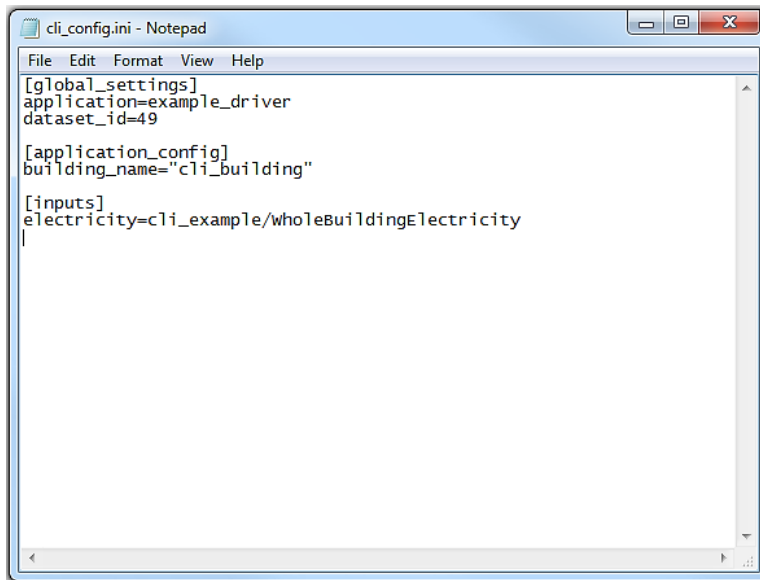


Figure 36: Running an application in the command line – Finished Configuration File

12. To run the application enter the following command from the OpenEIS directory:

- From Windows:

**python .\env\Scripts\openeis-script.py runapplication cli\_config.ini**

- From Linux or Mac:

**python . /env/bin/openeis-script.py runapplication cli\_config.ini**

To run in Eclipse:

- Navigate to *openeis\env\scripts\* directory (Windows) or the *openeis/bin/* (Linux or Mac)
- Right click **openeis-script.py**, select: Run As -> Run Configurations

- Figure 37 shows the Run Configurations window for a Windows PC. For a Linux or Mac the path to the Main Module will be different, as noted previously. On the Main tab, ensure that the Main Module is pointed **openeis-script.py**.
- Select the Arguments tab in the run configurations window.

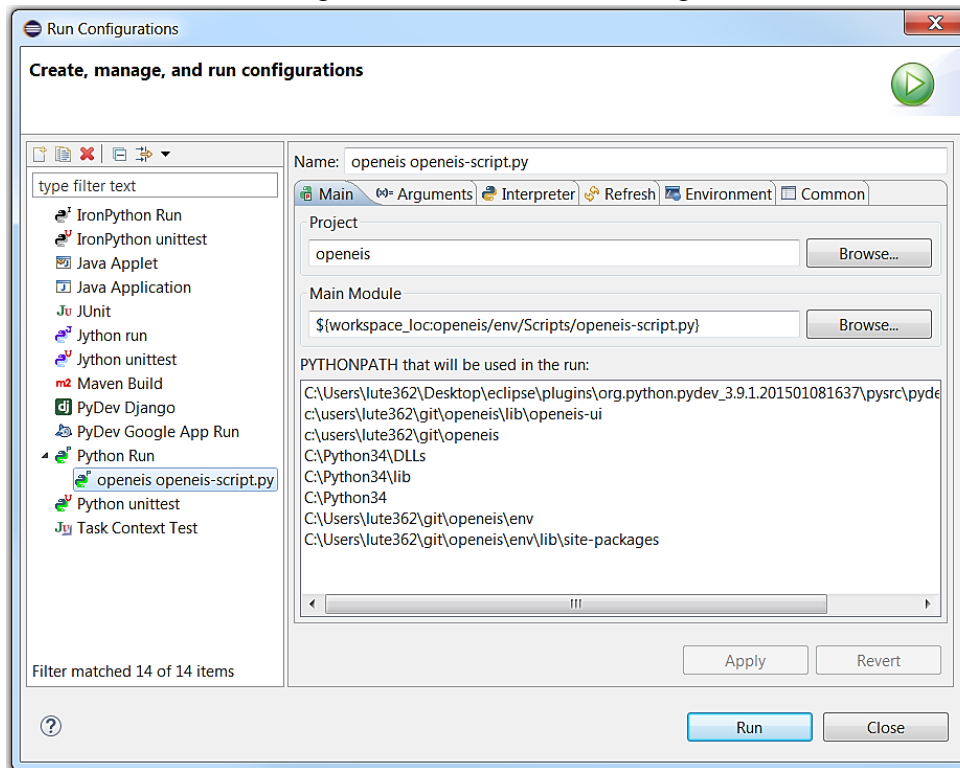


Figure 37: Running an application in the command line – Eclipse Run Configurations Main (Windows OS)

- In the Arguments tab under Working directory, select: Default
- In the Arguments tab in Program arguments, enter the following:

**runapplication openeis/applications/cli\_config.ini**

This argument specifies the path to the application configuration file.

- Click Run at the bottom of the Run Configurations window.
- The application results are viewable in the OpenEIS UI.

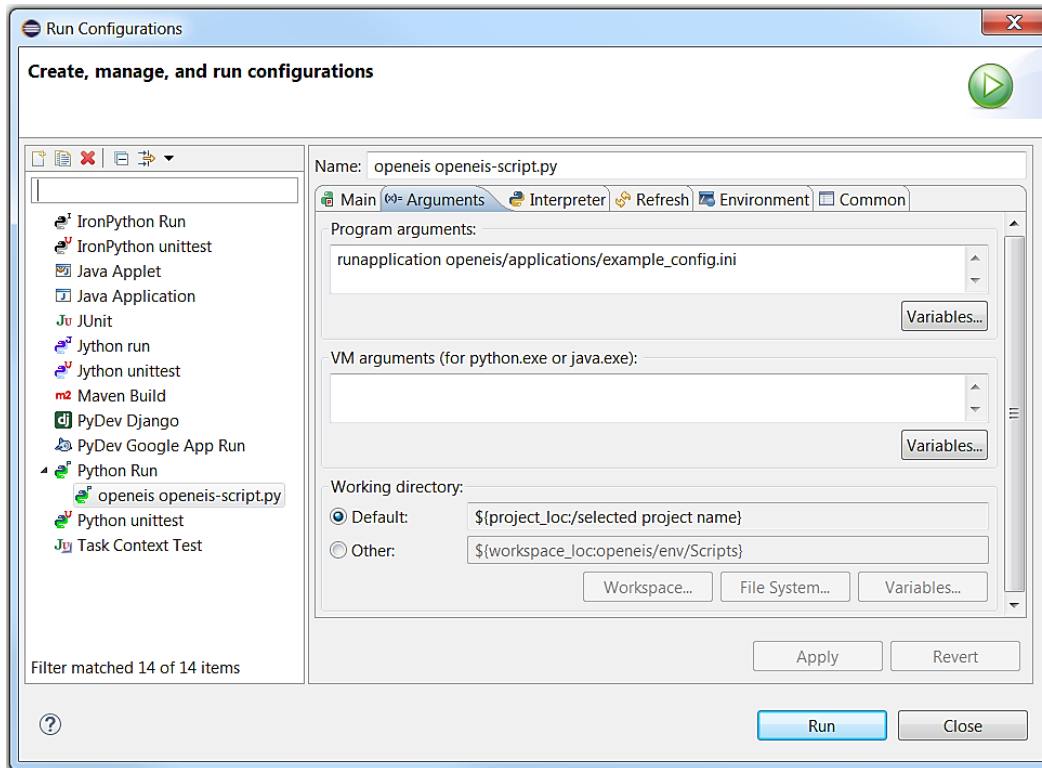


Figure 38: Running an application in the command line – Eclipse Run Configurations Arguments

## 4.2 Running Tests

To validate applications and changes to the main platform, unit tests should be written. The following section discusses how to build and run tests in the Eclipse IDE. These instructions are for running the tests in Eclipse, running on a Windows system. The steps for running the tests on a Linux system or Mac OS X may differ slightly.

OpenEIS uses pytest (<http://pytest.org/latest/>) for its testing environment. In order for tests to be run from the Eclipse environment, the py.test runner must be selected as the PyUnit Test Runner. This can be done globally from the menu (Window->Preferences->PyDev->PyUnit) or by configuring each Test Run Configuration.

Note: When configuring the PyUnit Test Runner in Eclipse, an optional parameter called verbosity is included by default. This parameter must be removed. Figure 39 and Figure 40.

Example of a PyUnit Test Run:

1. Open Eclipse
2. In the Project Explorer, navigate to *openeis/projects/tests/new\_tests/test\_greenbutton.py*
3. Right-click on *test\_greenbutton.py* in the Project Explorer

4. Select Run As - > Python unit-test

5. The console window will open and run the tests. The test output from a successful run will be similar to Figure 39.

```
===== test session starts =====  
platform win32 -- Python 3.4.1 -- py-1.4.25 -- pytest-2.6.3  
plugins: django  
collected 12 items  
  
test_greenbutton.py .....  
  
===== 12 passed in 13.66 seconds =====
```

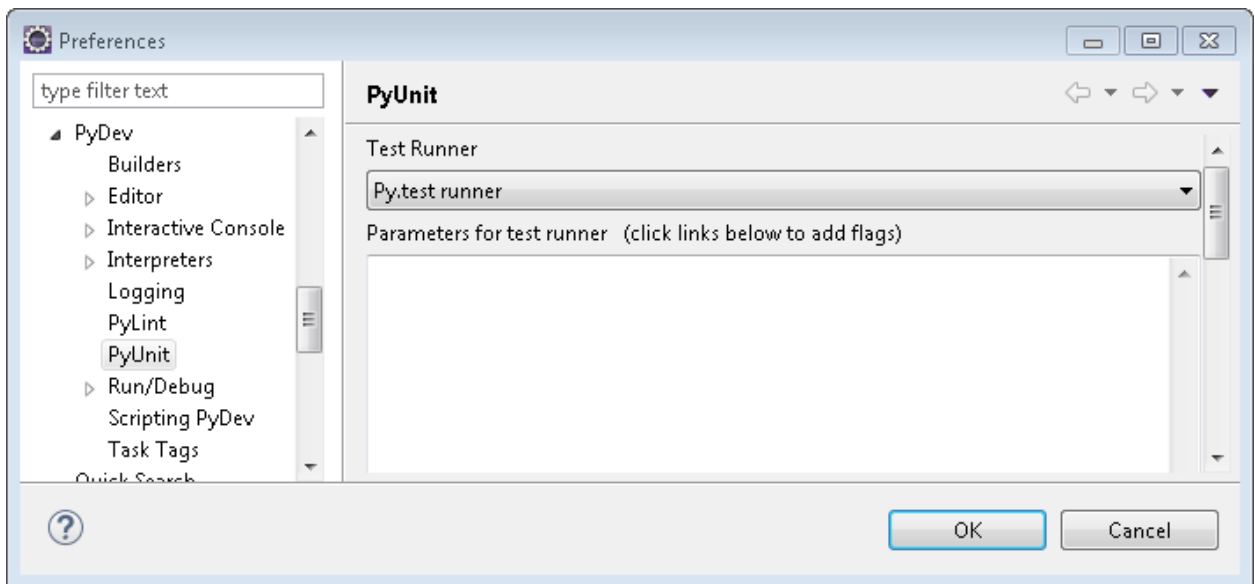


Figure 39: Global PyUnit test configuration

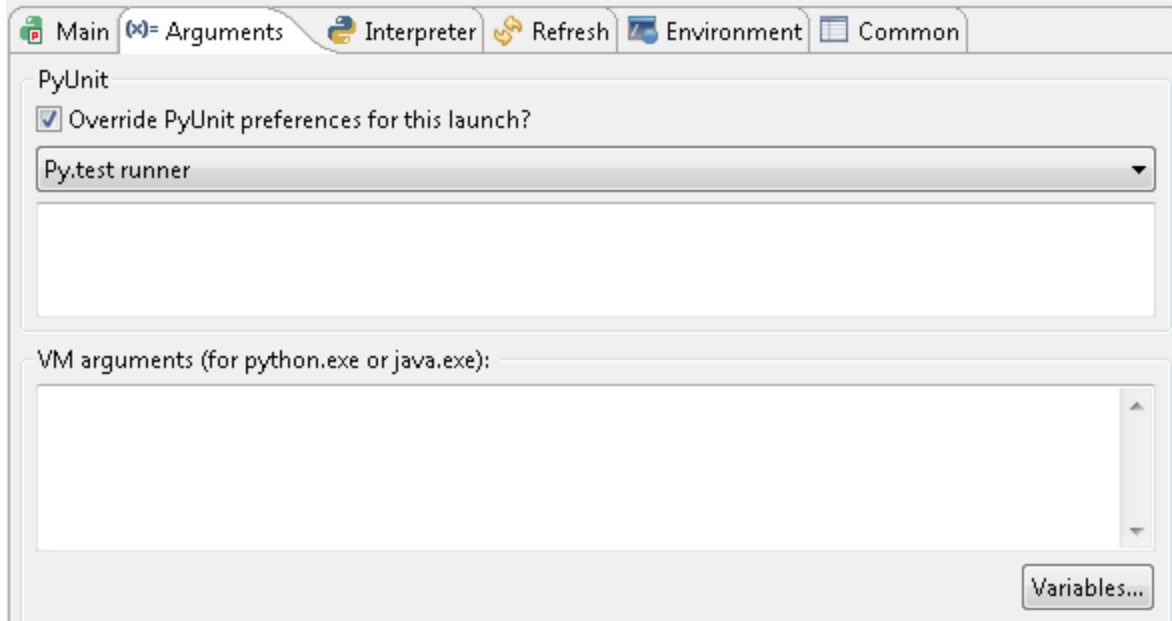


Figure 40: Run Configuration for individual tests

Use the Project Explorer in Eclipse to navigate to *OpenEIS/openeis/projects/tests/new\_tests/*. Right-click on a test file and select “Run As -> 2 Python unit-test” (Figure 41).

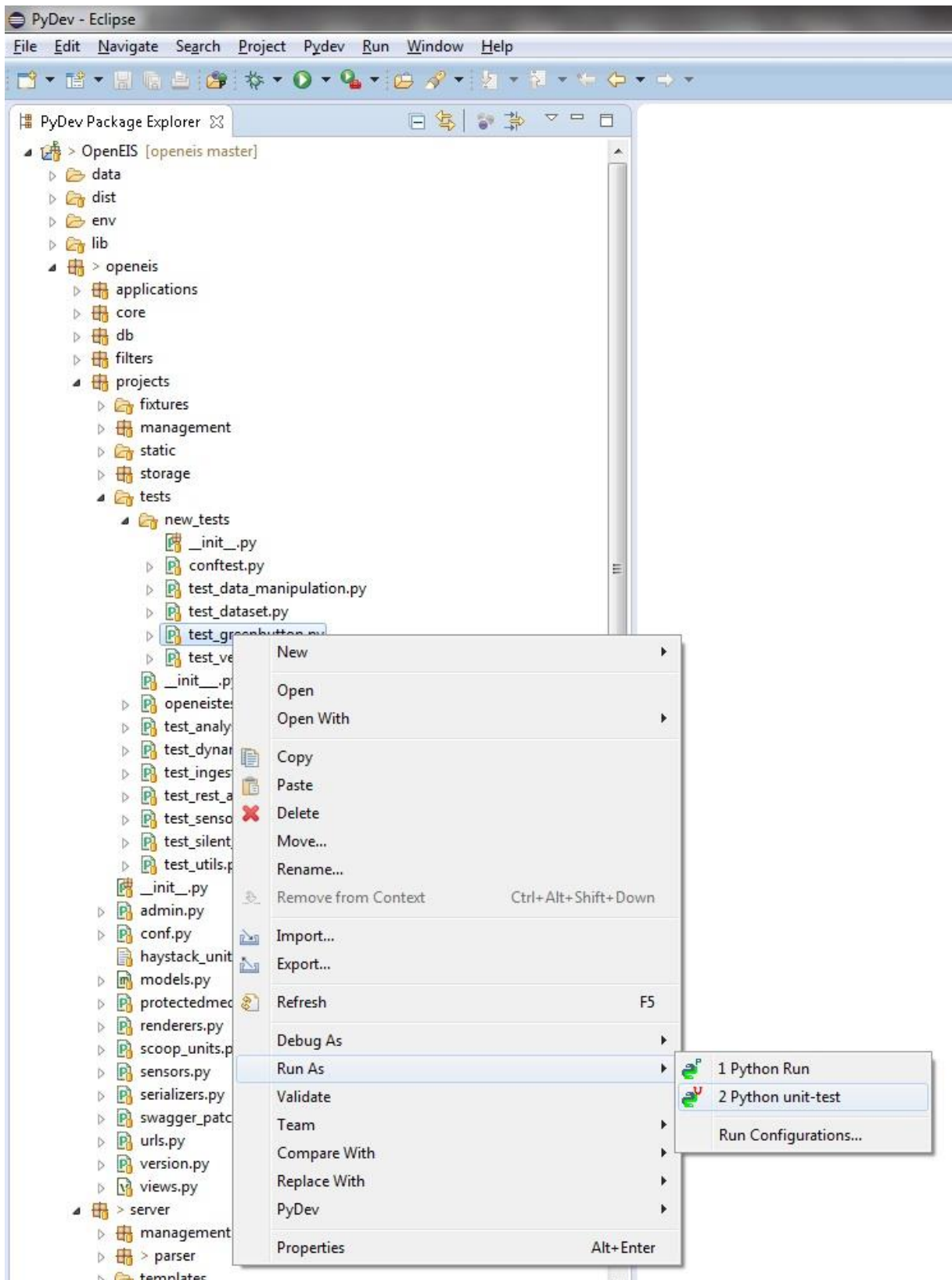
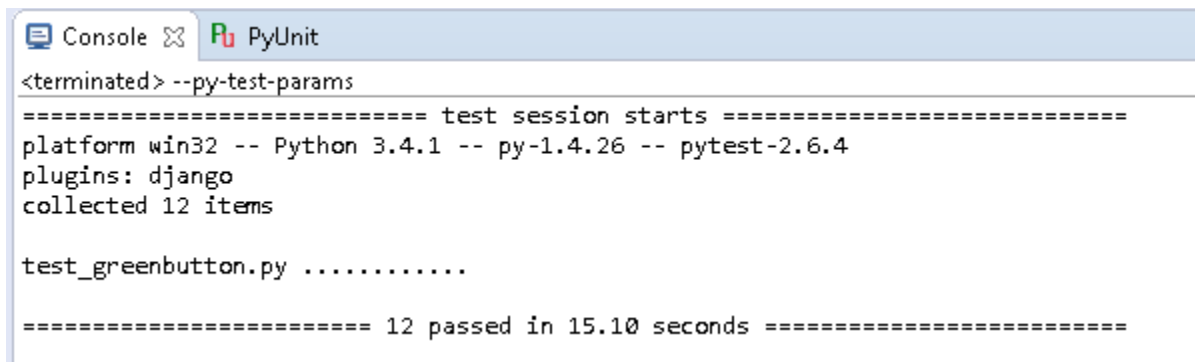


Figure 41: Running the tests

The output from executing the **test\_greenbutton.py** should be similar to Figures 42 and 43.



```
Console x PyUnit
<terminated> --py-test-params
===== test session starts =====
platform win32 -- Python 3.4.1 -- py-1.4.26 -- pytest-2.6.4
plugins: django
collected 12 items

test_greenbutton.py .....

===== 12 passed in 15.10 seconds =====
```

Figure 42: Console output from executing *test\_greenbutton.py* test

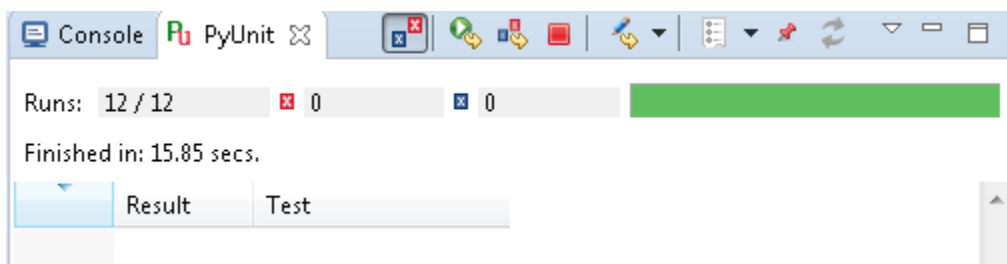


Figure 43: PyUnit output from executing *test\_greenbutton.py* test

## 5 OpenEIS Sensor Data and Server Interaction

The OpenEIS uses a SQL database (Sqlite3 by default but PostgreSQL is also supported) to store input data (sensor data), user OpenEIS account information, and application analysis. This section will explore the OpenEIS database and other related topics including:

- sensor data and naming convention
- database interaction
- RESTful interface
- OpenEIS Server API
- OpenEIS Server commands
- data manipulation filters

### 5.1 Sensor Data and Naming Convention

The OpenEIS has adopted a standard naming convention for sensor data. This allows users to map their data to standard OpenEIS names and utilize the OpenEIS analytic applications with greater ease. The OpenEIS will recognize applications data input requirements and what data is available (imported and mapped to standard names by user). If an application's data requirements are not met, the OpenEIS will not allow the application to be run.

The common naming schema will allow analytic developers to focus on development of more robust algorithms instead of worrying about data input or data naming conventions, a serious hurdle for the deployment of data analytics.

#### 5.1.1 Data Mapping (**sensormap-schema.json**)

The standard OpenEIS sensor naming convention and associated integration within the OpenEIS UI and database is implemented in three separate but intertwined text files: **sensormap-schema.json**, **units.json**, **general\_definitions.json**.

**sensormap-schema.json** maps the sensors and objects defined in **general\_definition.json** to the units defined in **units.json**. This file maps the sensors to the schema when a user selects a sensor type for a data map on the “Create new data map...” page in the UI.

An example of extending the **sensormap-schema.json** file is described in Section 5.1.3.

#### 5.1.2 Sensor Data and Associated Units (**units.json**)

The file **units.json** contains information on units for the various sensors available within the OpenEIS. The units in the file are grouped into categories such as length, mass, and temperature. For example, the length category contains definitions for centimeter, meter, kilometer, inch, foot, mile, and so on. When a sensor specifies a unit type in the **general\_definitions.json** file, that unit type must be defined in **units.json**. Changes to both these files will be automatically reflected in the UI without additional work. For example, if the dimensionless section in

**units.json** is modified to add a new unit, that change will be reflected in the UI dropdown for any sensor that specifies “dimensionless” units.

The following steps describe the process to add terawatt to the **units.json** file:

1. Open **units.json** (*openeis/projects/static/projects/json/units.json*) file with a text editor.
2. Navigate to the power section.
3. Add the following text to the file, then save and exit.

```
"terawatt": {  
  "key": "terawatt",  
  "value": "TW"  
},
```

Note: the entries in the file are alphabetized for convenience; this is not a requirement. Also, the final entry in the sensor\_list subsection should have the comma omitted.

4. Save the changes to the **units.json** file.
5. Open sensormap-schema.json (*openeis/projects/static/projects/json/sensormap-schema.json*) file with a text editor.
6. Navigate to the power section. This section contains an enum of power units.
7. Add the following text to the file. This text should be added between “milliwatt” and “tons\_refrigeration” in the power section enum:

```
"terawatt",
```

8. Stop the OpenEIS server (if it is running).
9. From the terminal, with the OpenEIS activated, enter the following command:

```
openeis syncdb
```

10. Restart the OpenEIS server (from terminal):

```
openeis runserver
```

11. “terawatt” should be available as a selection for unit with sensors that have a “unit\_type” of power.

12. To test this, create a new data map, add a new sensor, and select WholeBuildingPower from the “New sensor” dropdown. “terawatt” should be an option under the “Value unit” dropdown.
13. If the new unit is not available (in data map configuration in OpenEIS UI), clear the browser cache and retry steps 8 through steps 12.

### 5.1.3 Sensor and Object Definition (*general\_definition.json*)

The **general\_definition.json** file describes the available building systems (building, RTU, AHU, chilled water distribution system, hot water distribution system, and zone terminal-box) and the sensors associated with those systems within the OpenEIS. Although utilization of custom **general\_definition**, **sensormap-schema**, and **units** is possible, this change would necessitate modifying stock OpenEIS applications (applications that are packaged with OpenEIS and developed to use the standard sensor schema) to use the custom sensor schema. Extending the default schema by adding more sensors or devices does not require such modifications to stock OpenEIS applications.

The **general\_definition.json**, **sensormap-schema.json**, and **units.json** are located in the same directory, and modifying one will often necessitate changes in the other two files (the files are linked). These files are in the widely used and supported json<sup>1</sup> format.

The following steps describe the process to add “ParkingLotElectricity” to the default sensor schema:

1. Open **general\_definition.json** file  
(*openeis/projects/static/projects/json/general\_definition.json*) with a text editor.
2. Navigate to the building section.
3. Add the following text to the sensor\_list subsection. This adds the sensor to the list of sensors at the building level. If one had wanted to add the sensor to the AHU, for example, we would navigate to the AHU list and add the sensor there (instead or in addition):

"ParkingLotElectricity",

Note: The final entry in the sensor\_list subsection should have the comma omitted.

4. Navigate to the sensors section. This section contains the definitions of individual sensors.

---

<sup>1</sup> <http://www.json.org>

5. Add the following text to the sensors section of the file:

```
"ParkingLotElectricity": {  
    "data_type": "float",  
    "unit_type": "energy",  
    "sensor_name": "ParkingLotElectricity",  
    "sensor_type": "ParkingLotElectricity",  
    "default_aggregation": "Sum",  
    "default_fill": "LinearInterpolation"  
},
```

Note that the `unit_type` is “energy”, which indicates that the sensor will use units from the “energy” list, which is defined in **units.json**. **units.json** is described in Section 5.1.2.

Note: The final entry in the sensor section should have the comma omitted.

6. Open the **sensormap-schema.json** file (located in the same directory as the **general\_definition.json** file) with a text editor.
7. Because this sensor has a `unit_type` of energy, we need to add a reference to the sensor in the list of energy units. Navigate to the appropriate section (of **units.json**) by searching for the following text: `"#/definitions/units/energy"`
8. Immediately above the searched text is an enum (list) of sensor types such as `WholeBuildingElectricity`, `WholeBuildingGas`, etc. Add the following text to the enum:

```
"ParkingLotElectricity",
```

Note: The final entry in the sensor list should have the comma omitted.

9. Stop the OpenEIS server (if it is running).
10. From the terminal, with the OpenEIS activated, enter the following command:

```
openeis syncdb
```

11. Restart the OpenEIS server (from terminal):

```
openeis runserver
```

12. The “ParkingLotElectricity” sensor should now be available under building (device) when creating a data map.

13. If the new sensor is not available as a mappable point (in data map configuration in OpenEIS UI), clear the browser cache and retry steps 9 through steps 13.

## 5.2 Database

OpenEIS makes extensive use Django's object-relational mapper (ORM) to store most of its data. As such, the database models are found in *openeis/projects/models.py*, as prescribed by Django. Because the Django ORM is database agnostic, it is theoretically possible to use one of many relational database products, however, OpenEIS has only been tested with the Sqlite3 and PostgreSQL. OpenEIS is initially configured to use the Sqlite3 backend to simplify installation for developers, but PostgreSQL should be used in production or where performance is critical. Please refer to the Django documentation for more information on the ORM and for instructions on replacing the backend: <https://docs.djangoproject.com/en/1.6/#the-model-layer>.

The tables used by OpenEIS fall into three categories:

- User account and project organization
- Data mapping and ingestion
- Analysis and results.

These categories will be discussed in more detail in the next three sections.

### 5.2.1 User Account and Project Organization Models

OpenEIS extends the default Django user model to allow grouping users into one or more organizations. Organizations are not currently exposed to the user but can, however, be assigned in the Django admin interface. Adding a user to an organization is currently only useful for grouping related users together, but may provide the ability to share data between peers in a future release.

Each user may own zero or more projects. A project provides a way to group related data and is the root relation for the remaining table categories.

The model objects defining the tables in this category are Organization, Membership, Project, and AccountVerification. Please see *openeis/projects/models.py* for more details on each model.

### 5.2.2 Data Mapping and Ingestion Models

Data files must be uploaded to the server before any analysis may be performed. The uploading of data occurs in three stages: upload files, map files, and ingest files. The upload stage simply involves receiving a file from the client and saving it to the file system. Each saved file gets a record in the DataFile model, which includes some metadata including the timestamp column and time zone information. After uploading and defining the timestamp column, a file is ready to progress to the next stage.

Once a file is uploaded, the columns must be mapped to a standard set of point names, which each define their own data types and units. Mappings are stored in the DataMap model using a JSON encoded character field (JSONField). The JSON is checked against a schema to ensure its validity. Once the JSON is validated against a schema, the mapping is used to parse data from columns in the file and stored in the database as part of the ingest step.

Each data ingest is stored in the SensorIngest model. Each file ingested receives an entry in the SensorIngestFile model. And each sensor in the mapping is stored in Sensor model. The data for each sensor is stored in one of four models, depending on the data type: BooleanSensorData for boolean types, FloatSensorData for floats, IntegerSensorData for integers, and StringSensorData for strings.

Errors that occur during ingestion, such as parse errors, are stored in the SensorIngestLog model, from where they may be queried by the client and displayed to the user. Once the data is ingested into the tables, the custom SensorDataQuerySet queryset and SensorDataManager manager are used to perform queries, including truncating dates/times and aggregating the results.

### 5.2.3 Analysis and Results Models

Filters and analysis applications may be run against ingested data. While filtering produces new ingest records, application runs are stored in the Anaysis model. Because applications can define their output tables based on input, the results schema is not known in advance and the models, therefore, must be produced dynamically during application execution. Most of the dynamic table logic is found in *openeis/projects/storage/dynamictables.py*. The rest is handled by the AppOutput model. Dynamic tables are not managed by Django and must use triggers to propagate updates and deletes. See `delete_appoutputdata` and `sync_appoutputdata` in **models.py**.

Results with the same project and column type signatures will share tables to avoid the proliferation of dynamic tables.

Keys for shared analyses are stored in the SharedAnalysis model.

## 5.3 RESTful Interface

Representational State Transfer (REST) is a style of software architecture used in web services. Such a system or interface is termed RESTful. The OpenEIS Server provides a RESTful interface. The RESTful interface supports requests for retrieving (GET), adding (POST), editing (PUT), and deleting (DELETE) data. For more information on RESTful interfaces, see [http://en.wikipedia.org/wiki/Representational\\_state\\_transfer](http://en.wikipedia.org/wiki/Representational_state_transfer).

### 5.3.1 Hypertext Transfer Protocol Status Codes

Hypertext Transfer Protocol (HTTP) is a set of standards that facilitates exchange of information on the World Wide Web. When the OpenEIS server is running HTTP status codes are returned to provide information on the status of a request. HTTP status codes are grouped as follows:

- 1xx - Informational. Request received continuing to process.
- 2xx - Success. The server successfully processed the request.
- 3xx - Redirection. Further action is needed to fulfill the request. Often, these status codes are used for redirection
- 4xx - Client Error. Indicates an error on the client (user) side, such as specifying an invalid URL. Many people are familiar with the “404 Page not found” error. Other examples of 400 errors can be caused by invalid parameters specified in a request, or valid parameters with invalid ranges.
- 5xx - Server Error. Indicates that the server had an internal error when trying to process the request. These errors tend to be with the server itself, not with the request

For a detailed list of HTTP status codes see:

[http://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](http://en.wikipedia.org/wiki/List_of_HTTP_status_codes)

### 5.3.2 Views

The API calls in OpenEIS are backed by ViewSets, which can be found in the following project file:

➤ *openeis/projects/views.py*

Each API has a corresponding ViewSet. For instance, the File API has FileViewSet with functions defined for downloading the file, retrieving the first n lines of the file, and parsing the selected columns as timestamps. The following ViewSets are implemented.

ProjectViewSet – Lists all projects owned by the active user.

FileViewSet – Lists all files owned by the active user.

UserViewSet – Lists all active users.

AccountViewSet – Create, update, and delete user accounts.

DataMapViewSet – Manipulate data maps owned by the active user.

DataSetViewSet – Retrieves DataSet ingestion.

DataSetPreviewViewSet – Returns sample rows from a DataSet ingestion.

ApplicationViewSet – This ViewSet is used to return a list of OpenEIS Applications with inputs and parameters.

FilterViewSet – This ViewSet is used to return a list of OpenEIS Filters. For more information on filters see Section 5.6 *Data Manipulation Filters*.

VersionViewSet – This ViewSet is used for the OpenEIS version information.

AnalysisViewSet – Retrieve analyses owned by the active user.

SharedAnalysisViewSet – Retrieve shared analyses.

## 5.4 OpenEIS Server API

This section lists some useful API pages available under the OpenEIS server. After starting the server (Section 4), log in to enable the API pages. The API pages offer a way to make HTTP POST and GET requests without using the user interface. Additionally, you may also see past requests made through the UI. The POST requests allow one to put information on the server and database, while GET allows one to obtain information from the database.

Interaction with the OpenEIS database can also be accomplished through a RESTful API. An example of interaction with the OpenEIS database via the RESTful API interface is contained in Section 5.4.8 (RESTful example via Python code).

Most API pages have an "OPTIONS" and "GET" button near the top. The "OPTIONS" button displays a JSON object describing the API (e.g., its name, what it can parse, what data needs to be input, etc.). The "GET" button displays a JSON object showing the relevant information from the database. At the top of the field, there is a header showing: HTTP request code, content-type, and the allowed HTTP request methods, if the POST data can vary. The results of GET request can be displayed as formatted or unformatted JSON. To view the unformatted version, select "json" from the dropdown menu on the side of the "GET" button (selecting "api" gives the default, formatted version).

### 5.4.1 API Root

This is the root of the API. The API root is a central location where one can access the other API content (Figure 44). When running the OpenEIS server locally, the API root can be accessed at:

<http://localhost:8000/api/>

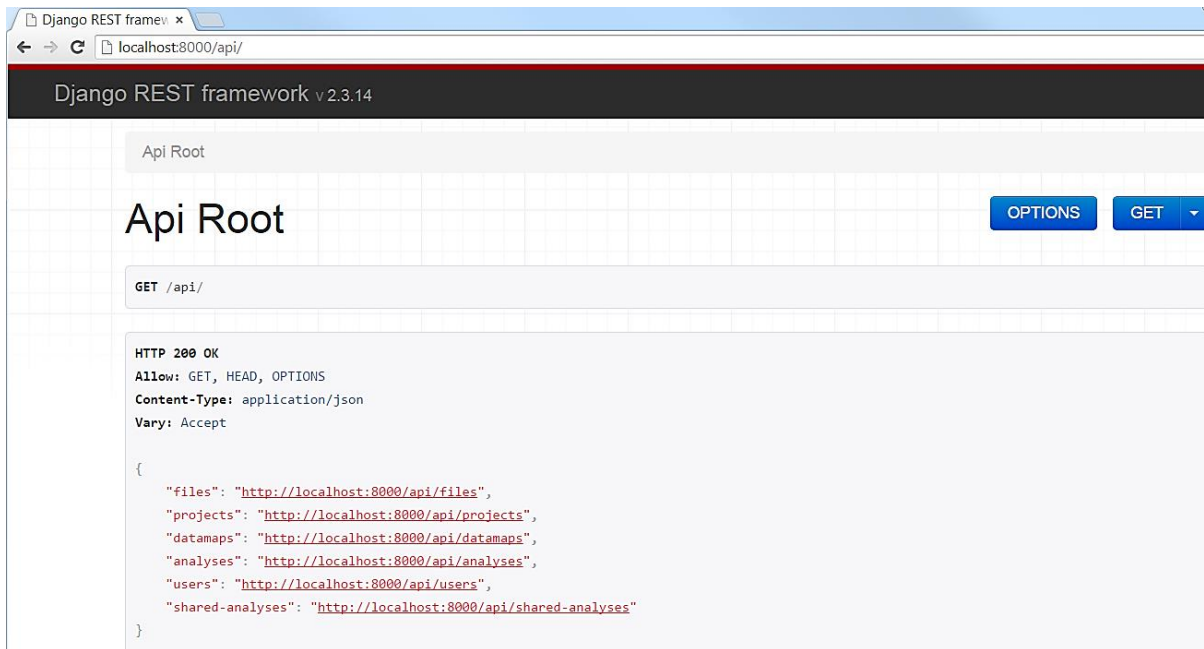


Figure 44: API Root

## 5.4.2 API Projects Page

This page lists all of the projects that are in the database (Figure 45):

<http://localhost:8000/api/projects>

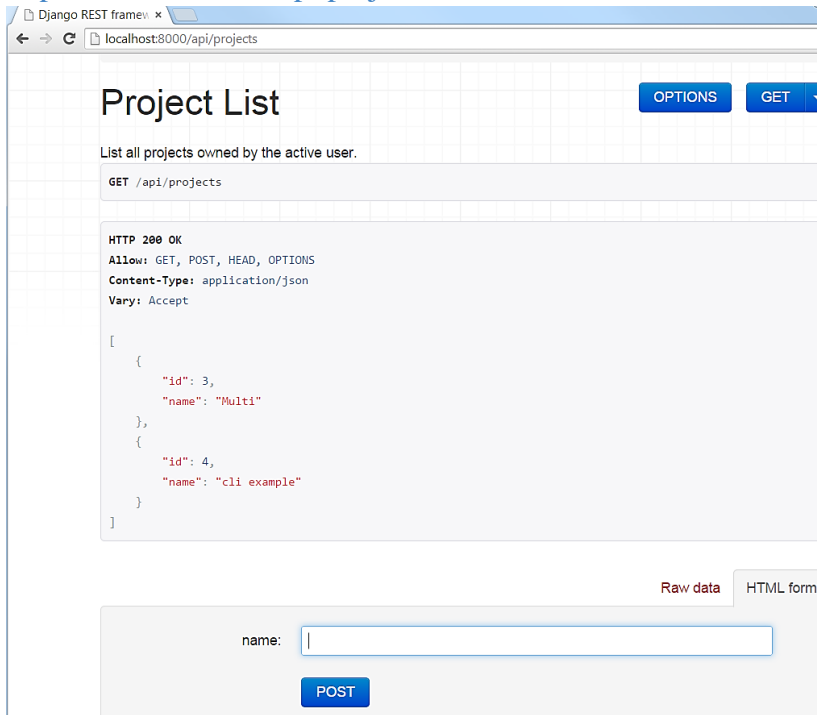


Figure 45: API project page

To create a new project, enter a name in the HTML form and click "POST". You can also put the name of your project under "Raw data" in the "name" field. Now you should see a JSON formatted entry that has an id associated with your project and the name you had given it (Figure 46).

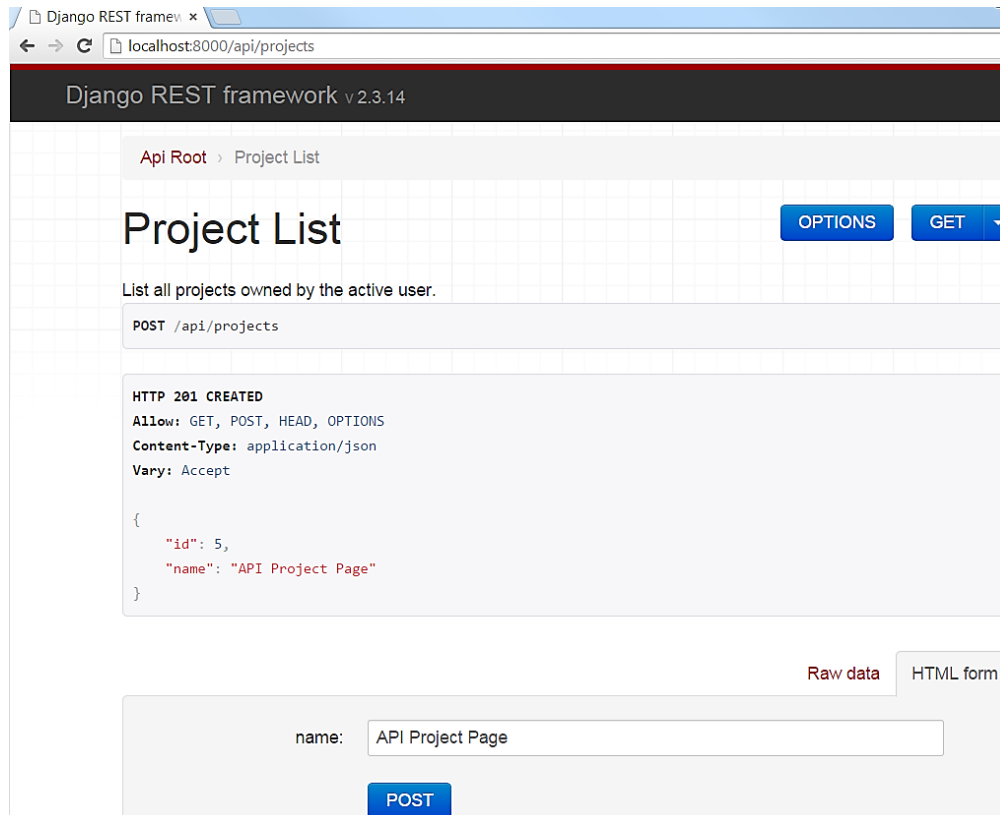


Figure 46: API project page (continued)

### 5.4.3 API Files Page

This page lists the files uploaded to the database. You may only GET with this page, so you must upload files through the user interface or use the RESTful API (Figure 47):

<http://localhost:8000/api/files>

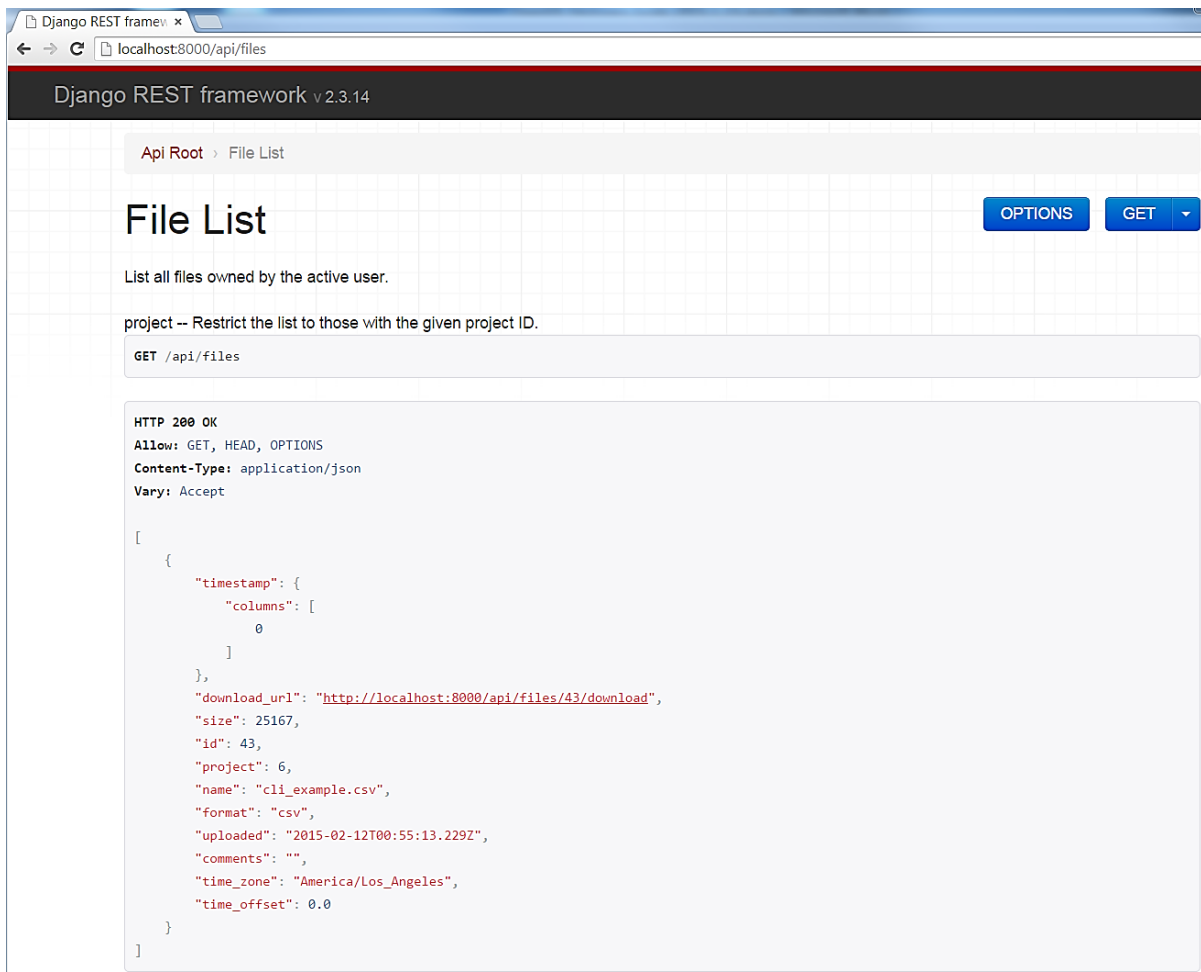


Figure 47: API File List

#### 5.4.4 API Sensor Data Maps Page

When running the OpenEIS server locally, the API data maps can be accessed at:

<http://localhost:8000/api/data maps>

You may POST your own sensor map in "map" field of the "HTML form" as well as the "Raw data". The sensor map you post must be in JSON format. It is easiest to first input a sensor map with the user interface and then copy the JSON format for input (Figure 48 and Figure 49).

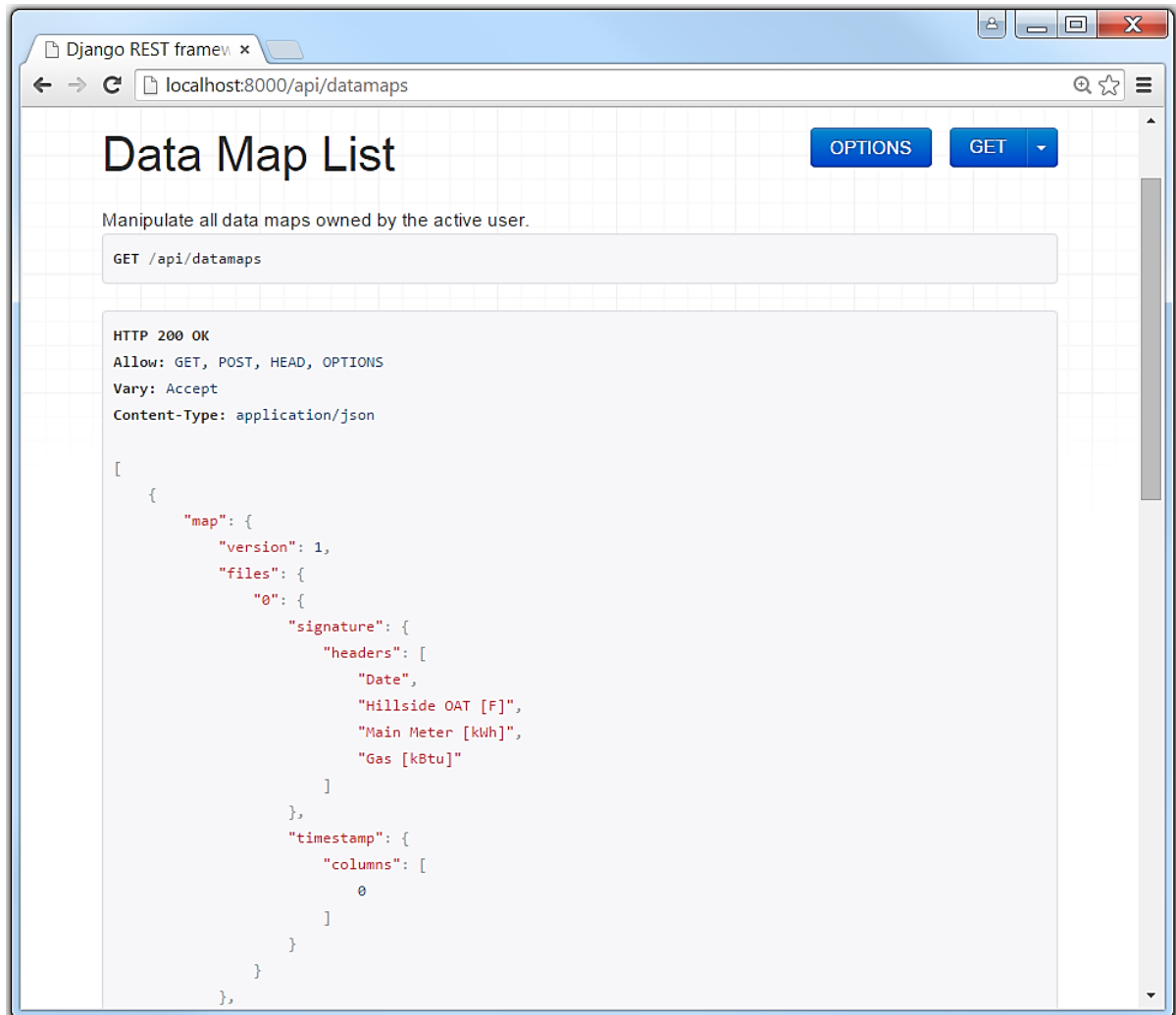


Figure 48: API data map (top)

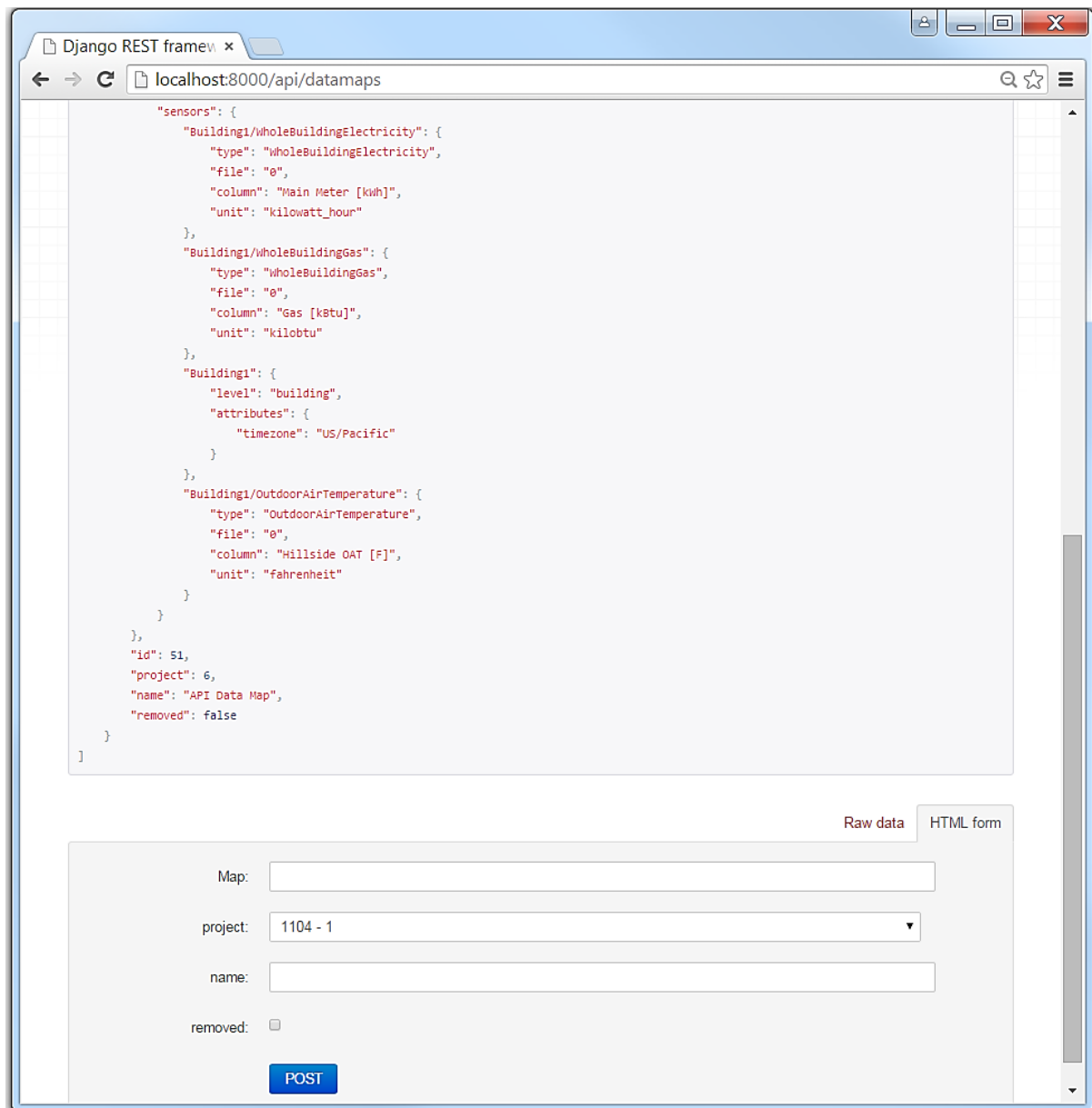


Figure 49: API data map (bottom)

#### 5.4.5 API Datasets Page

Information regarding datasets created within the OpenEIS (Figure 50) can be accessed at (when locally running OpenEIS server): <http://localhost:8000/api/datasets>

# Data Set List

GET /api/datasets

HTTP 200 OK

Allow: GET, POST, HEAD, OPTIONS

Vary: Accept

Content-Type: application/json

```
[
  {
    "files": [
      {
        "name": "0",
        "file": 43
      }
    ],
    "download_url": "http://localhost:8000/api/datasets/50/download",
    "id": 50,
    "project": 6,
    "name": "API Data Map - 2/11/15 4:56 PM",
    "map": 51,
    "start": "2015-02-12T00:56:47.724Z",
    "end": "2015-02-12T00:56:49.075Z",
    "datamap": {
      "version": 1,
      "files": {
        "0": {
          "signature": {
            "headers": [
              "Date",
              "Hillside OAT [F]",
              "Main Meter [kWh]",
              "Gas [kBtu]"
            ]
          },
          "timestamp": {
            "columns": [
              0
            ]
          }
        }
      }
    },
    "sensors": {
      "Building1/WholeBuildingElectricity": {
        "type": "WholeBuildingElectricity",
        "file": "0",
        "column": "Main Meter [kWh]",
        "unit": "kilowatt_hour"
      }
    }
  }
]
```

Figure 50: API dataset page

When running an application for the command line (Section 4.1) or from the API analyses page (Section 5.4.7), one will need the sensor path and the dataset id. In (Figure 50) the id is circled in blue at the top of the figure (“id”: 50,) and the sensor path is circled in blue towards the bottom of the figure (“Building1/WholeBuildingElectricity”).

#### 5.4.6 API Authentication Page

This page lists all of the usernames on the database right now. You may only GET from this page, so new users must be added through the user interface. When running the OpenEIS server locally, the authentication page can be accessed at:

<http://localhost:8000/api/auth>

#### 5.4.7 Analyses Page

This page can be used to run applications and perform analyses:

<http://localhost:8000/api/analyses>

The following fields contain information telling the OpenEIS what application to run, what data to use for analysis, and what configuration parameters to pass to the application.

- name – Name/tag of analysis being performed. This can include a date or other information to help identify the analysis results at a later time (API analyses used for the example shown in Figure 51).
- dataset – Using the dropdown menu, pick the dataset appropriate to the application.
- application – The name of the Python file that defines the application, but without ".py" (this should be the same entry as in the .ini file).
- configuration – The required configuration parameters for the application in a JSON format.

Hit POST. You should see output on the console, as well as new tables in the database. If the application creates a visualization of its analysis results, the visualization will also be available in the UI (Figure 51).

Raw data HTML form

name: API analyses

dataset: SensorIngest object - 50

application: example\_driver

configuration:

```
{
  "global_settings": {
    "application": "example_driver",
    "dataset_id": 50
  },
  "logging": {
    "log_file": "test.log",
    "log_level": "INFO"
  },
  "parameters": {
    "building_name": "cli_example"
  },
  "inputs": {
    "electricity": [
      "Building1/WholeBuildingElectricity"
    ]
  }
}
```

debug: ☒

POST

Figure 51: API analyses page – application configuration

#### 5.4.8 Example Interaction with RESTful API

The following is an example of interaction with the RESTful API using Python code. This coding example has detailed comments for each action performed in the code. This file is located at:

`openeis\projects\tests\restful_api_example.py`

The Requests module can be installed with the following command:

➤ **pip install requests**

Note that if this command is run when the OpenEIS platform is activated, it will use the pip and Python interpreter in OpenEIS and will not be installed for the rest of the system. The reverse is also true, if this command is run when the platform is not activated, it will be installed only for the system Python.

```
import requests
import json

# Setup server URL
LOCALHOST = 'http://localhost:8000'
HIGHROAD = 'https://<remote host url>'
```

```

host = HIGHROAD

# Setup user and password. This user account must exist
# on the server chosen as host above.
username = 'DevGuide'
password = 'DevPass'
auth = (username, password)

# Path to data file.
filename = 'C:/path/to/1Month_hourly.csv'

# Create a new project for our user, this is
# equivalent of using the Create button on the Projects screen
project_payload = {'name': 'resttest'}
response = requests.post('{host}/api/projects'.format(host=host),
                        auth=auth, data=project_payload, verify=False)

# Retrieve the project id from the response
project_id = response.json['id']

# Setup the file we will upload and include some metadata
file_meta = {'file': open(filename, 'rb'),
             'name': filename,
             'format': 'csv'}

# Post the file to the add_file endpoint. Equivalent of using the
# Upload File button in the UI.
filepost_response = requests.post('{host}/api/projects/{proj}/add_file'.
                                  format(host=host, proj=project_id),
                                  files=file_meta, auth=auth, verify=False)

# Retrieve the file id from the response
file_id = filepost_response.json['id']

# Set the timestamp column and timezone for the file. Uses an http PATCH
# Equivalent of Configure Timestamp on file
ts = {'timestamp': {'columns': [0]}, 'time_zone': 'US/Pacific'}

# Set content-type
headers = {'content-type': 'application/json'}
patch_response = requests.patch('{host}/api/files/{id}'.
                                format(host=host, id=file_id),
                                json.dumps(ts), auth=auth, headers=headers,
                                verify=False)

# Send data map to RESTful endpoint instead of creating it in UI
# requires project id, retrieved above
data_map = {
    'project': project_id,
    'name': 'restmap',
    'map': {
        'version': 1,
        'sensors': {
            'New building/WholeBuildingPower': {

```



```

        .format(host=host),
        json.dumps(dataset_request),
        auth=auth, headers=headers, verify=False)

# Retrieve dataset id from response
dataset_id = dataset_response.json['id']

# Now we will use the dataset as input to an application
# This replaces Run Analysis from UI. All the information that would be input
# in the analysis GUI must be replicated here
application_setup = {'application': 'heat_map',
                     'configuration': {
                         'parameters': {'building_name': 'MyBuilding'},
                         'inputs': {'load': [
                             'New building/WholeBuildingPower'
                         ]
                        }},
                     'dataset': dataset_id, 'debug': 'false',
                     'name': 'rest_dataset - Heat Map'}

# Post the application setup and the result
# should appear on the server for viewing.
application_response = requests.post('{host}/api/analyses'
                                     .format(host=host),
                                     json.dumps(application_setup),
                                     auth=auth, headers=headers, verify=False)

```

## 5.5 OpenEIS Server Commands

The OpenEIS has several server commands available to run from the command line. All commands, except those in the projects and server directories, are implemented from a Django command or a dependency. Many commands require additional command line arguments, such as a file name or path. The most commonly used command is **openeis runserver**, used to start the OpenEIS server. Another frequently used command is **openeis syncdb**, used to synchronize the database.

Remember to activate OpenEIS before running commands. In a terminal (command prompt), navigate to the base OpenEIS directory and enter the following commands:

- For Linux or Mac:

➤ **./env/bin/activate**

Note the space after the period

- For Windows:

➤ **.\env\Scripts\activate.bat**

Additional information on each command can be displayed by running the following commands (from the OpenEIS base directory with the project activated):

**`./env/bin/openeis help <COMMAND>`** (Linux or Mac)

**`.\env\scripts\openeis help <COMMAND>`** (Windows)

Following is a list of OpenEIS server commands, organized by the module that implements the command.

### 5.5.1 Authorization Commands

The Authorization Commands section contains commands for managing users and passwords. The authorization commands are as follows:

`changepassword` – Change a user's password for `django.contrib.auth`.

`createsuperuser` – Used to create a superuser.

### 5.5.2 Django Commands

The Django Commands section contains the majority of the Django commands. The Django commands are as follows:

`check` – Checks your configuration's compatibility with this version of Django.

`cleanup` – Can be run as a cronjob or directly to clean out expired sessions (only with the database backend at the moment).

`compilemessages` – Compiles `.po` files to `.mo` files for use with builtin gettext support.

`createcachetable` – Creates the table needed to use the SQL cache backend.

`dbshell` – Runs the command line client for specified database, or the default database if none is provided.

`diffsettings` – Displays differences between the current `settings.py` and Djangos.

`dumpdata` – Output the contents of the database as a fixture of the given format (using each model's default manager unless `--all` is specified).

`flush` – Returns the database to the state it was in immediately after `syncdb` was executed. This means that all data will be removed from the database, any post-synchronization handlers will be re-executed, and the `initial_data` fixture will be re-installed.

`inspectdb` – Introspects the database tables in the given database and outputs a Django model module.

loaddata – Installs the named fixture(s) in the database.

makemessages – Runs over the entire source tree of the current directory and pulls out all strings marked for translation. It creates (or updates) a message file in the conf/locale (in the Django tree) or locale (for projects and applications) directory.

runfcgi – Run this project as a fastcgi (or some other protocol supported by flup) application. To do this, the flup package from <http://www.saddi.com/software/flup/> is required.

shell – Runs a Python interactive interpreter. Tries to use IPython or bpython, if one of them is available.

sql – Prints the CREATE TABLE SQL statements for the given app name(s).

sqlall – Prints the CREATE TABLE, custom SQL and CREATE INDEX SQL statements for the given model module name(s).

sqlclear – Prints the DROP TABLE SQL statements for the given app name(s).

sqlcustom – Prints the custom table modifying SQL statements for the given app name(s).

sqldropindexes – Prints the DROP INDEX SQL statements for the given model module name(s).

sqlflush – Returns a list of the SQL statements required to return all tables in the database to the state they were in just after they were installed.

sqlindexes – Prints the CREATE INDEX SQL statements for the given model module name(s).

sqlinitialdata – RENAMED: see 'sqlcustom'

sqlsequencereset – Prints the SQL statements for resetting sequences for the given app name(s).

startapp – Creates a Django app directory structure for the given app name in the current directory or optionally in the given directory.

startproject – Creates a Django project directory structure for the given project name in the current directory or optionally in the given directory.

syncdb – Create the database tables for all apps in INSTALLED\_APPS for tables that haven't already been created.

testserver – Runs a development server with data from the given fixture(s).

validate – Validates all installed models.

### 5.5.3 django\_pytest Commands

The django\_pytest Commands section contains commands used for testing. The django\_pytest commands are as follows:

**pytest** – Discover and run tests in the specified modules or the current directory.

**test** – Discover and run tests in the specified modules or the current directory.

### 5.5.4 Project Commands

The Project Commands section contains commands specific to manipulating an OpenEIS project. The project commands are as follows:

**cleanprojectfiles** – Remove files orphaned by deleting database files and/or projects. When deleting uploaded files via the web interface, the database record is removed, but the files remain. This command finds files with no corresponding database record and removes them from the file system. It should be run occasionally on a personal installation or regularly from a cron job on a production system.

**cleardyntables** – Remove orphaned dynamic application output tables. Deleting analyses via the web interfaces removes the database records but leaves the tables. This command finds orphaned dynamic tables and drops them. This command probably does not need to be run on a personal installation, but should be run regularly on a production system.

**clearprojects** – Drop project tables to prepare for database update. A developer command to easily drop tables. It will erase all project data and remove tables. Use this command with extreme caution as data loss will occur. Because of the danger, the -f/--force argument is required to actually delete data.

**cloneproject** – Duplicate (clone) a project.

**manipulatedata** – Copy a dataset while performing filtering operations.

**runapplication** – Run an application from the command line.

### 5.5.5 Server Commands

The Server Commands section contains commands specific to the OpenEIS server. The server commands are as follows:

**linkstatic** – Link project and application static directories to static root. Similar to collectstatic command, except soft links are used instead of copying the files. Useful when the application and content servers are on the same system.

**localsettings** – Create openeis.local package with skeleton settings.

nginxconfig – Create nginx configuration file from project settings.

### 5.5.6 Session Commands

The Session Commands section contains commands specific to a session. The session commands are as follows:

clearsessions – Can be run as a cronjob or directly to clean out expired sessions (only with the database backend at the moment).

### 5.5.7 staticfiles

The staticfiles section contains commands specific to manipulating static files in OpenEIS.

collectstatic – Collect static files in a single location.

findstatic – Finds the absolute paths for the given static file(s).

runserver – Starts a lightweight Web server for development and also serves static files.

## 5.6 Data Manipulation Filters

Filters are a powerful tool for manipulating and merging data from building automation systems or other device loggers. Often, data from these sources can contain problematic data (e.g., missing data, data loggers trending at different frequencies, etc.), which make analysis more difficult and time consuming. The OpenEIS filters are intended to streamline the pre-processing of data by providing data manipulation filters to deal with common data issues.

The *OpenEIS: Users Guide*, Section 8 contains detailed instruction on the use of the data manipulation filters (from the perspective of an OpenEIS user).

### 5.6.1 OpenEIS Filters

New filters can be created by extending the appropriate class. Decorate<sup>2</sup> the filter with **@register\_column\_modifier** to make it available in the UI and command line utilities.

OpenEIS provides the following types of common filters that users can extend to create custom filters (located at: *openeis/filters/\_\_init\_\_.py* and *openeis/filters/common.py*):

- **BaseSimpleAggregate** – Aggregation filters group data, often to reduce the trending interval or align trending intervals for multiple data points. For an example of a **BaseSimpleAggregate** filter, see the **Average** filter (*openeis/filters/average.py*).

---

<sup>2</sup> <https://www.python.org/dev/peps/pep-0318/#on-the-name-decorator>

- **BaseSimpleNormalize** – Normalization filters ensure that data occurs at regular intervals and eliminate gaps in the data and extra data. For an example of a **BaseSimpleNormalize** filter, see the linear **Interpolation** filter (*openeis/filters/linear\_interpolation.py*).
- **SimpleRuleFilter** – This type of filter does not combine or normalize the data, just changes the value for a given timestamp. For an example of a **SimpleRuleFilter**, see the **RoundOff** (*openeis/filters/round\_off.py*).

All of the filter types inherit from **BaseFilter**. To create a filter, extend the appropriate filter class (based on type of filter being created) or **BaseFilter**.

**BaseFilter** is the parent class for all filters.

- **BaseSimpleAggregate**, **BaseSimpleNormalize**, and **SimpleRuleFilter** all inherit from the **BaseFilter**.

```
class BaseFilter(SelfDescriptorBaseClass,
                 ConfigDescriptorBaseClass,
                 metaclass=abc.ABCMeta):
    def __init__(self, parent=None):
        self.parent = parent

    @abc.abstractmethod
    def __iter__(self):
        pass

    @classmethod
    @abc.abstractmethod
    def filter_type(cls):
        pass
```

- **parent** – the parent filter, or filter that comes before this one in the chain of filters acting on a column of data. **self.parent** is set to this value for the child filter.
- **\_\_iter\_\_** – abstract method, must return an iterator that yields (Python datetime, value) tuples.

- A new filter will typically inherit from one or more of the Simple filters (e.g., **BaseSimpleAggregate**).

### 5.6.2 Example Filters

This section explores the data manipulation filters. Each type of “Simple” filter is explored by way of an example. The filters documented in this section are available by default in the OpenEIS (Section 5.6.1).

**SimpleRuleFilter** can be extended (to a child class) to create a rule filter, which modifies each value with a rule function, defined in the child class.

```

class SimpleRuleFilter(BaseFilter, metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def rule(self, time, value):
        """Must return time, value pair."""

    def __iter__(self):
        def generator:
            for dt, value in self.parent:
                yield self.rule(dt, value)
        return generator

    @classmethod
    def filter_type(cls):
        return "other"

```

`__iter__` – abstract method, classes that inherit from **SimpleRuleFilter** must return an iterator that yields (datetime, value) tuples. Applies `self.rule` method (defined in child class) to each datetime, value associated with the sensor measurement being manipulated and returns a generator object.

The **RoundOff** filter serves as a good example of a rule filter:

- Declare the import statements:

```

from openeis.filters import SimpleRuleFilter, register_column_modifier
from openeis.core.descriptors import ConfigDescriptor, Descriptor

```

- Create the child class that inherits from **SimpleRuleFilter** (**SimpleRuleFilter** inherits from **BaseFilter**):

```

@register_column_modifier
class RoundOff(SimpleRuleFilter):
    """
    Round the value of a column to a specified number of places.
    """

    def __init__(self, places=0, **kwargs):
        super.__init__(**kwargs)
        self.places = places

```

The decorator **@register\_column\_modifier** makes the filter available in the UI. When an instance of **RoundOff** is instantiated, the class initializes a keyword argument `places` (input from UI or command line). The default value for `places` is “0”.

- Create the rule to modify the value of input data (this filter creates a rounding rule allowing the user to control the perceived precision of the input data):

```

def rule(self, time, value):
    return time, round(value, self.places)

```

This method inputs a datetime and sensor value. The function returns an unmodified datetime and a sensor value that has been rounded to ‘places’ past the decimal.

- Create information for UI:

```
@classmethod
def get_config_parameters(cls):
    description_ = 'Number of places to round to. \n'
    description += 'i.e. 2 will round to 1.12345 to 1.12. \n'
    description += 'i.e. 0 will round to 123.12345 to 123. \n'
    description += 'i.e. -2 will round to 1234.12345 to 1200.'
    return {
        'places': ConfigDescriptor(int, "Rounding Places",
                                   description=description,
                                   value_default=0)
    }

@classmethod
def get_self_descriptor(cls):
    name = 'Rounding Filter'
    desc = 'Round the value of a column to a specified number of places.'
    return Descriptor(name=name, description=desc)
```

**get\_config\_parameters** – informs the UI what values the class requires when instantiated. The parameter **places** will be supplied by user in the UI. If the user chooses to apply the **RoundOff** filter and does not supply a value for **places**, the value defaults to zero (drop all digit past the decimal point).

**get\_self\_descriptor** – supplies the UI with a description of the class (**RoundOff**) and its function to display to the user.

**BaseSimpleAggregate** simplifies the creation of filters that aggregate values over a period of time. If no values exist for a time period aggregation method, (**aggregate\_values**) is not called and the period is skipped.

The **Average** filter serves as a good example of an aggregation filter:

- Declare the import statements:

```
from openeis.filters.common import BaseSimpleAggregate, register_column_modifier
from openeis.core.descriptors import Descriptor
```

- Create the child class that inherits from **SimpleAggregationFilter** (**SimpleRuleFilter** inherits from **BaseFilter**):

```
@register_column_modifier
class Average(BaseSimpleAggregate):
    """
    Aggregate by averaging.
    """
    def aggregate_values(self, target_dt, value_pairs):
        return sum(value for _, value in value_pairs)/len(value_pairs)
```

```

@classmethod
def get_self_descriptor(cls):
    name = 'Average'
    desc = 'Aggregate by averaging.'
    return Descriptor(name=name, description=desc)

```

- A filter that inherits from **BaseSimpleAggregate** must implement the **get\_self\_descriptor** method, **aggregate\_values** method and optionally **get\_config\_parameters** and **filter\_type** methods.

**filter\_type** – returns string with a description of the filter. This method defaults to “aggregation”.

**aggregate\_values** – returns a single value derived from the **target\_dt** and **value\_list**. **value\_list** is guaranteed to always have at least one value in it.

**target\_dt** – the time stamp that will be associated with the returned value.

**value\_list** – a list of time stamp/value tuples from the period to aggregate.

**BaseSimpleNormalize** simplifies the creation of filters that normalize time stamp/values pairs to a uniform trending interval. Normalization only occurs between two existing values and only to a timestamp aligned to a regular interval. If a value already exists for a target timestamp, then the unaltered value is used and the child class’s normalization method (**calculate\_value**) method is not invoked.

The **LinearInterpolation** filter serves as a good example of a normalization filter:

- Declare the import statements:

```

from openeis.filters.common import BaseSimpleNormalize, register_column_modifier
from openeis.core.descriptors import Descriptor

```

- A filter that inherits from **BaseSimpleNormalize** must implement the **get\_self\_descriptor**, class’s **calculate\_value** methods and optionally **get\_config\_parameters** and **filter\_type**. The filter types return value defaults to **filter\_type** “fill”. **get\_config\_parameters** defaults to configuring the period to normalize on and whether or not to drop values that do not line up with the period.

```

@register_column_modifier
class LinearInterpolation(BaseSimpleNormalize):
    '''
    Normalize values to a specified time period'
    using Linear Interpolation to
    supply missing values.
    '''

```

```

def calculate_value(self, target_dt):
    x0 = self.previous_point[0]
    x1 = self.next_point[0]
    if x1 <= target_dt <= x0:
        raise RuntimeError('Tried to interpolate
                           'value during incorrect state.')
    y0 = self.previous_point[1]
    y1 = self.next_point[1]
    return target_dt, y0 + ((y1-y0)*((target_dt-x0)/(x1-x0)))

@classmethod
def get_self_descriptor(cls):
    name = 'Linear Interpolation'
    desc = ('Normalize values to a specified time period using '
           'Linear Interpolation to supply missing values.')
    return Descriptor(name=name, description=desc)

```

**calculate\_value** – function called to calculate the normalized value for target\_dt. The previous timestamp/value pair is stored in self.previous\_point and the next timestamp/value pair is stored in self.next\_point. These two points are used to perform the linear interpolation calculation and for the value that corresponds to the target\_dt.

## 6 Creating Applications

Data analytics and the development of effective analysis tools to utilize the ever-increasing amounts of trend data have seen significant growth in recent years. Intuitive, easy-to-understand, highly visual analysis tools have never been more important. The OpenEIS strategy is aimed at getting the market to validate and implement state-of-the-art analytical and diagnostic algorithms. The following section will give detailed instructions on how to develop an application in the OpenEIS.

Applications come in two flavors. They are either driver applications or driven applications. Both application types share some similarities. Both application types must implement the following methods:

- **\_\_init\_\_** – this method instantiates driver application class.
- **get\_self\_descriptor** – this method creates a name object and description object for display in the UI.
- **get\_config\_parameters** – this method declares and returns the required configuration parameters for the application.
- **required\_input** – this method describes the required input data for the application.
- **output\_format** – this method describes the format for application output (application results).
- **reports** – this method describes how to application result will be visualized (type of visualization: chart, graph) and describes components of the visualization (graph titles, axes, etc.).
- **execute** – this method is the core of the application; this is where the application uses data and create results.

First, application interaction with the OpenEIS database (i.e., obtaining data for analysis and uploading analysis results) will be detailed, with examples. Then, the requirements for building an OpenEIS application will be explained and both driver and driven applications will be examined.

### 6.1 Application Interaction with the Database

Applications utilize data that has been uploaded into the OpenEIS database. For information on uploading data, please refer to the *OpenEIS: User Guide*. The applications use the data and create meaningful results based on this analysis. These results are then uploaded to the OpenEIS database for use in creation of application visualizations (e.g., charts).

### 6.1.1 Application Interactions with the OpenEIS Database: Retrieving Input Data

Input for an application is available via a `DataBaseInput` object via `self.inp`. The interface to this object gives the application access to the data selected by the user for analysis.

- **get\_topics** – return a copy of the topic map (dictionary) used for this run of the application. The map will be in the following format:

```
Topic map: {'data name': ['building/device/point']}
```

For example, the topic map from Section 4.1 would appear as follows:

```
Topic map: {'electricity': [cli_example/WholeBuildingElectricity]}
```

The topic map is a Python dictionary that contains the data input sensor information as defined by the data map (dictionary value) and the application’s name for the data input sensor (dictionary key). 'data name' is declared in the **required\_input** method and is the key(s) in the dictionary returned by the **required\_input** method (more details on the **required\_input** method are contained in Sections 6.2.1 and 6.3.1). The **get\_topics** method is useful for extracting information related to an application’s analysis (e.g., site or building name) for reports and result visualizations (e.g., charts, graphs, etc.).

- **get\_topics\_meta** – return a copy of the metadata associated with the user selected topics.

The topic map allows an application to access metadata associated with the input data. For example, if the application has a “load” topic (typically building or device power), one can look up the units for this sensor measurement as follows:

```
base_topic = self.inp.get_topics
meta_topics = self.inp.get_topics_meta

load_unit = meta_topics['Load'][base_topic['Load'][0]]['unit']
```

`load_unit` will be a string indicating what unit was mapped with the sensor value (e.g., a typical unit for load might be ‘kilowatt’).

- **get\_tz\_for\_sensor(sensor\_topic)** – returns the time zone for the selected topic (i.e., `sensor_topic`). The `sensor_topic` argument required by the method is in string format.
- **localize\_sensor\_time(sensor\_topic, timestamp)** – return the timestamp (as a `datetime`<sup>3</sup> object) localized to the supplied `sensor_topic`’s time zone. The `sensor_topic` argument is in string format and the `timestamp` argument is a `datetime` object.

---

<sup>3</sup> <https://docs.python.org/3.4/library/datetime.html>

- `get_query_sets(group_name, order_by='time', filter_=None, exclude=None, wrap_for_merge=False, group_by=None, group_by_aggregation=None)` – This method returns a list of Django queryset objects for each column in the selected topic.
  - `order_by` – may be either “time” or “value”. Orders the results by the selected column, ascending.
  - `filter_` – filters out data that does not meet the supplied condition. The filter object is required to be in the form of a dictionary. The two values available to filter on are “time” and “value”. For example, `{'time_hour':1}` will filter out all values that are not between 1 AM and 2 AM.
  - `exclude` – this method works identically to `filter_` except all data that meets the supplied condition is excluded.
  - `wrap_for_merge` – this returns the result wrapped in a dictionary to be used as an argument for a **merge** call.
  - `group_by` – time period to group the result by for aggregation. May be one of “second”, “minute”, “hour”, “day”, “month”, “year”, or “all”. When “all” is specified, `group_by` will return a list of values instead of a list of querysets because there is nothing to iterate over.
  - `group_by_aggregation` – method used for grouping. This is a Django aggregation method such as **Min**, **Max**, **Avg**, or **Sum**. These modules must be imported from *django.db.models*. See <https://docs.djangoproject.com/en/1.7/topics/db/aggregation/> for more details on Django aggregation methods.
- `merge(*args, drop_partial_lines= True)` - this method merges one or more queryset by providing a generator<sup>4</sup> that iterates them in lockstep, returning the results of each step as a dictionary that maps a topic to a list of values for that time step. The timestamp is contained in the result and can be obtained via the “time” key, which maps to the timestamp of the result.
  - `drop_partial_lines` – If *True* the generator will leave out timestamps for which one or more sources did not provide a value. Otherwise, those time steps for which any queryset provided at least one value will be included, and querysets that did

---

<sup>4</sup> <https://wiki.python.org/moin/Generators>

not provide values for the time step will be given a “None” value for the time step.

It is common to aggregate data before merging and to drop partial lines (drop rows of data where one or more data fields is missing for that row) to eliminate the need to evaluate unaligned (non-uniform time series data) data. There is also a use case for merging the results of a single call to `get_query_sets` if the call can possibly result in a list with more than one queryset. In this case, `merge` will handle advancing the separate querysets.

For example, to obtain two lists of querysets (here `load_query` and `oat_query`, each set having one queryset), merge them, and drop partial lines. It should be noted that the initial query (using `get_query_sets`) returns data that is aggregated to hourly trend interval (aggregation method is averaging (`Avg`)):

```
load_query = self.inp.get_query_sets('load', group_by='hour',
                                     group_by_aggregation=Avg,
                                     exclude={'value':None},
                                     wrap_for_merge=True)
oat_query = self.inp.get_query_sets('oat', group_by='hour',
                                    group_by_aggregation=Avg,
                                    exclude={'value':None},
                                    wrap_for_merge=True)

merged_load_oat = self.inp.merge(load_query, oat_query)
for x in merged_load_oat:
    print(x['oat'][0], x['load'][0]) # A real application would do something useful
```

### 6.1.2 Application Interactions with the OpenEIS Database: Uploading Analysis Results

A `DataBaseOutput` object that is available to the application via `self.out` handles output for an application. The interface to this object gives the application a place to push analysis results. The format for the output data is specified in the applications `output_format` method. The following method is available to `output_format`:

- `insert_row(table_name, row_data)` - insert a row of data into a table requested by `output_format` in the format specified in `output_format`. The input parameter `row_data` must be a dictionary of column names (keys) to values.

For example, with the following output format:

```
{
    'Weather_Sensitivity': {
        'value':OutputDescriptor('string', weather_topic)
    },
    'Load_vs_OAT': {
        'oat':OutputDescriptor('float', oat_topic),
        'load':OutputDescriptor('float', load_topic)
    }
}
```

```
}
```

You could call `insert_row` as follows:

```
self.inp.insert_row('Load_vs_OAT', {'oat',78.5:, 'Load'120.0})
```

The following method can be called to log information (this can be especially helpful if debugging or testing):

- `log(msg, level=logging.DEBUG, timestamp=None)` - Log data for the application. The table to place this data is created automatically.
  - `msg` – the message to log, typically a string.
  - `level` – logging level of the message
  - `timestamp` – if specified, the logger will use the supplied timestamp. Without this argument, the current system time is used. This is meant for applications that need the log times to be somehow reconcilable with either the input or the output.

More information on the `output_format` method is available in the Section 6.2.1 and Section 6.3.1 on driven and driver applications, respectively.

## 6.2 Driven Applications

Driven applications and the framework for deploying driven applications were created to allow previously developed applications and analysis tools to be incorporated into the OpenEIS with as few modifications to the application as possible.

A developer creates or implements a driven application in the OpenEIS by extending the abstract class **DrivenApplicationBaseClass**. A driven application does not directly query for data from the OpenEIS database. The application is passed input data as a dictionary of key value pairs when its `run` method is called by the **DrivenApplicationBaseClass**.

The **Results** class (defined in `openeis/applications/__init__.py`) serves as a container for output returned by a driven application. This class also handles logging and database insertion for the driven application.

A driven application requires an additional method, the `run` method:

**run** – method where the driven application uses data and creates analysis results. The `run` method that is called by the **DrivenApplicationBaseClass**. The execute method for the driven application is in the **DrivenApplicationBaseClass**. This method “flattens” (creates dictionary

of key value pairs) the required input data and calls the driven application's **run** method with the data and current timestamp (as datetime) as inputs.

Each row of data used for analysis is passed to the run method (as well as the timestamp corresponding to that row of data). For example, if a dataset contained 100 rows of data, then the **run** method would be called 100 times; each time the dictionary of data would contain the respective row of input data.

### 6.2.1 Example Driven Application

This section will detail the creation of an example driver application.

1. Open an IDE (this example will utilize Eclipse) or text editor (e.g., notepad++) and create a new Python file in the applications directory. In the Eclipse menu, select:

➤ File -> New -> PyDev Module (Figure 52)

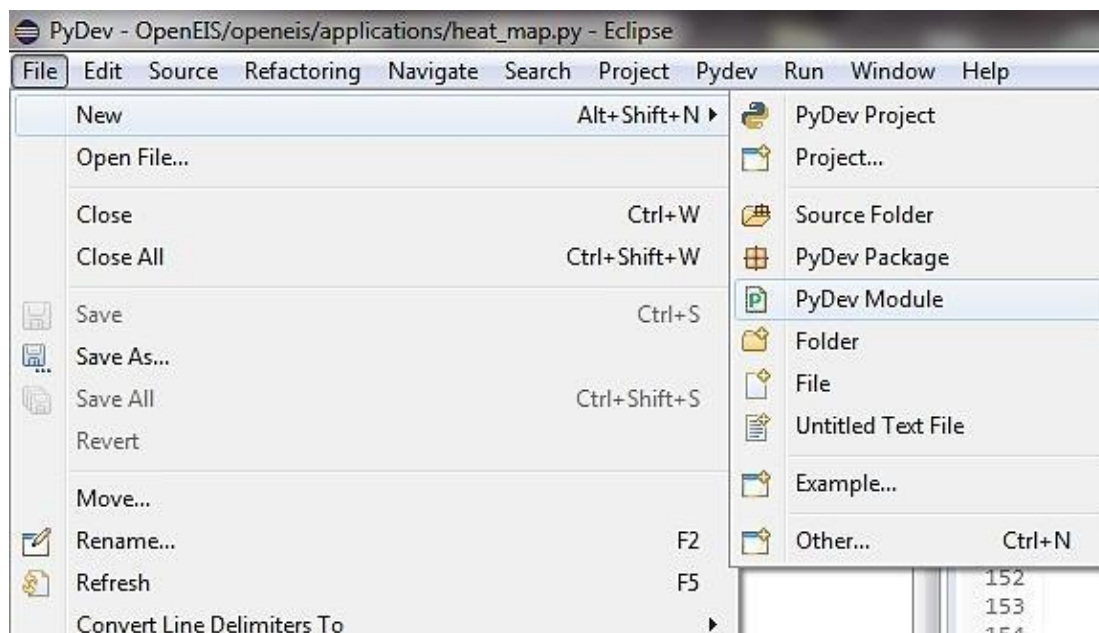


Figure 52: Creating a new Driven Application

Tip: Another way to do this in Eclipse is to right-click on the *openeis/applications* package in the project tree and in the context menu select:

New -> PyDev Module (Figure 53)

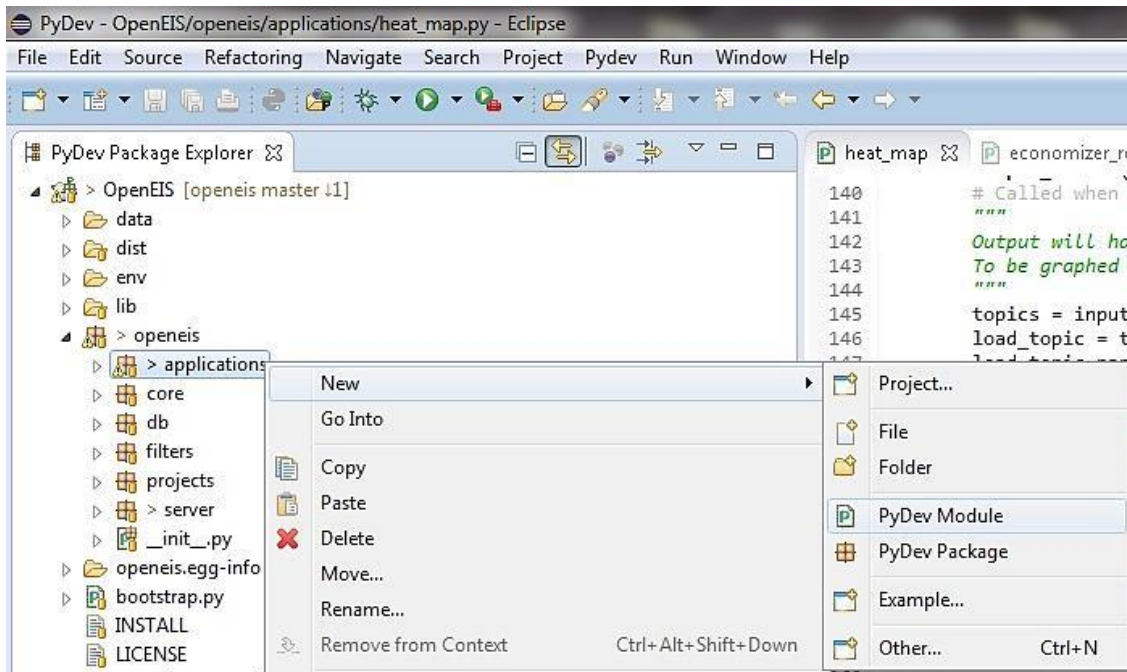


Figure 53: Creating a new Driven Application from the Project Explorer

2. On the new Python module screen, enter the following information:

Source Folder: /OpenEIS

Package: openeis.applications

The application name can be anything but should reflect the nature of the application. In this example, the application is named “example\_driven”. It is important that the application file is located in the *openeis/applications* directory.

- Click Finish (Figure 54)

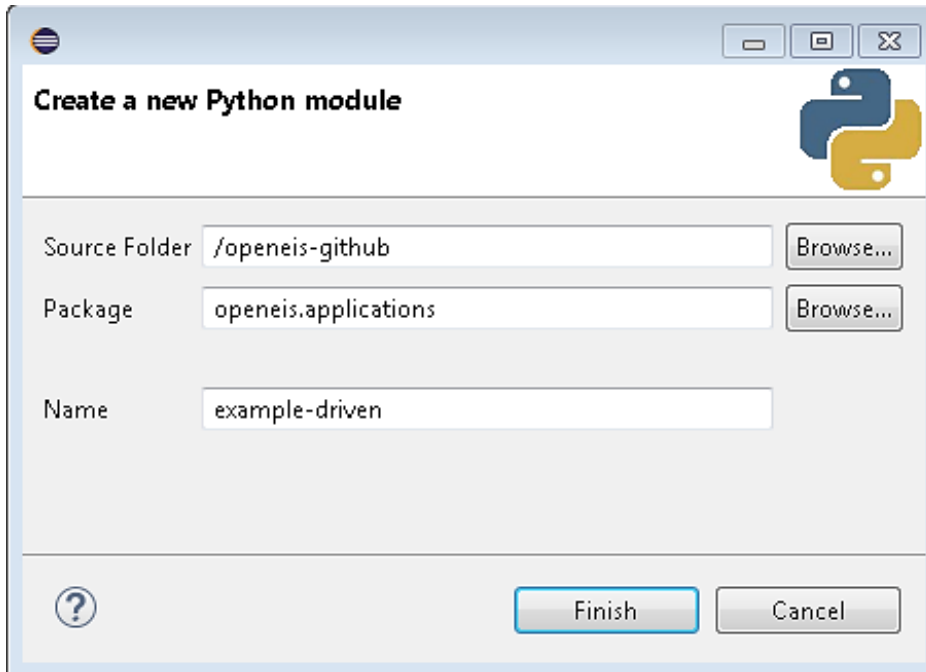


Figure 54: Creating a new Driven Application (continued)

3. Next, declare the modules to import for use by the application. The following import list is typical for driven applications:

```
import logging
from openeis.applications import (DrivenApplicationBaseClass,
                                   OutputDescriptor,
                                   ConfigDescriptor,
                                   InputDescriptor,
                                   Results,
                                   Descriptor,
                                   reports)
```

The logging<sup>5</sup> module is part of the Python standard library. Logging is a means of tracking events that happen when some software runs. The software’s developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message, which can optionally contain variable data.

The remaining imports are OpenEIS modules that are required and will be explained in detail as the “example\_driven” application is built.

4. In the file, create a new class<sup>6</sup> called **Application**. **Application** should inherit from the **DrivenApplicationBaseClass** (located at *openeis/applications/\_\_init\_\_.py*).

<sup>5</sup> <https://docs.python.org/3.4/howto/logging.html>

<sup>6</sup> <https://docs.python.org/3.4/tutorial/classes.html>

```
class Application(DrivenApplicationBaseClass):
```

A class named **Application** is required for all applications in OpenEIS. The class inherits from **DrivenApplicationBaseClass** (which in turn inherits from **DriverApplicationBaseClass**).

5. Create the `__init__` method:

```
def __init__(self, *args, building_sq_ft=-1,
             building_year_constructed=-1,
             building_name=None, **kwargs):
    """When applications extend this base class, they need to make
    use of any kwargs that were setup in get_config_parameters.
    """
    super.__init__(*args, **kwargs)
    self.default_building_name_used = False
    if building_sq_ft < 0:
        raise Exception("Invalid input for building_sq_ft")
    if building_year_constructed < 0:
        raise Exception("Invalid input for building_sq_ft")
    if building_name is None:
        building_name = "None supplied"
        self.default_building_name_used = True
    self.sq_ft = building_sq_ft
    self.building_year = building_year_constructed
    self.building_name = building_name
    self.first = True
    self.counter = 0
```

the `__init__` method has the default arguments (`*args` and `**kwargs`)<sup>7</sup> and any other configuration parameters (declared in `get_config_parameters`). The body of the method contains a call to `super`<sup>8</sup>, and some basic error handling for the configuration parameters.

6. Next, create the `get_self_descriptor` method:

```
@classmethod
def get_self_descriptor(cls):
    name = 'test_driven'
    desc = 'test_driven'
    return Descriptor(name=name, description=desc)
```

This method returns a Descriptor object. The Descriptor object should contain two keyword arguments, name and description. The name should be a human intelligible name for the application (or custom filter), and the description should be a short

---

<sup>7</sup> <https://freepythontips.wordpress.com/2013/08/04/args-and-kwargs-in-python-explained/>

<sup>8</sup> <https://docs.python.org/2/library/functions.html#super>

explanation of what the application does. This text is displayed to users in the OpenEIS UI and is how a user will identify the application.

7. Create the `get_config_parameters` method:

```
@classmethod
def get_config_parameters(cls):
    """Required application configuration
    parameters.
    """
    return {
        "building_sq_ft":
            ConfigDescriptor(float,
                             "Square footage",
                             value_min=200),
        "building_year_constructed":
            ConfigDescriptor(int,
                             "Construction Year",
                             value_min=1800,
                             value_max=2014),
        "building_name": ConfigDescriptor(str,
                                           "Building Name",
                                           optional=True)
    }
```

The `get_config_parameters` method returns a dictionary with information about an application's required configuration parameters. This method is called by the UI when constructing the application configuration page, the menu where users input configuration parameters for an application.

The `ConfigDescriptor` object contains the following arguments:

- `config_type` – Data type for the configuration parameter. In our example, the data type is a string (str). This parameter is required.
- `display_name` – Name displayed to user in the UI. In our example, the data type is a string ("Building Name"). This parameter is required.
- `description` – Text displayed to user as description of parameter. This parameter is optional.
- `optional` – Keyword argument indicating if the parameter is optional. The default value for optional argument is False. This parameter is optional.
- `value_default` – The default value for the configuration parameter. This value will automatically propagate into the application configuration menu in the OpenEIS UI. This point is optional; the default behavior is that there is no minimum default value for the configuration parameter. This parameter is optional.

- `value_min` – This will set a lower bound on the configuration parameter. The user will not be allowed to enter a smaller value for the configuration parameter. This point is optional; the default behavior is that there is no minimum value for the configuration parameter. This parameter is optional.
- `value_max` – This will set an upper bound on the configuration parameter. The user will not be allowed to enter a larger value for the configuration parameter. This point is optional; the default behavior is that there is no maximum value for the configuration parameter.
- `value_list` – A list of valid inputs for the configuration parameters. These values will appear in a drop down for the configuration parameters. The user will only be able to choose from these values. This point is optional; the default behavior is that there any value (with the correct `config_type`) can be used. This parameter is optional.

This method allows a user to understand and customize configuration parameters for an application. An OpenEIS user can configure an application, customizing the analysis and making the results more meaningful to them.

Please note:

- This method must be implemented because the **Application** class inherits from **ConfigDescriptorBaseClass**.
- Noteworthy, is that the **DriverApplicationBaseClass** defined in `openeis/applications/__init__.py` inherits from both **ConfigDescriptorBaseClass** and **SelfDescriptorBaseClass**. These descriptor classes are found in `openeis/core/descriptors.py`.

#### 8. Create the `required_input` method:

```
@classmethod
def required_input(cls):
    """Returns a dictionary of required data."""
    return {
        'OAT':
            InputDescriptor('OutdoorAirTemperature',
                           'Outdoor Temperature',
                           count_max=None),
        'Load': InputDescriptor('WholeBuildingElectricity',
                               'Building Load'),
        'natgas': InputDescriptor('NaturalGasEnergy', 'Natural Gas usage')
    }
```

The **required\_input**<sup>9</sup> method returns a dictionary with information on the required data to run the application. This method is called by the UI, which determines if the data required to run the application is present in the dataset of interest (dataset that is chosen for analysis).

The InputDescriptor is described in *openeis/core/descriptors.py* and takes the following parameters:

- **sensor\_type** – The sensor type is the name of the input sensor. This will be a string containing the OpenEIS name for a sensor. The UI uses this to verify that the required input data is present, and then the UI makes the application available to the user.
- **display\_name** – The display name is a human intelligible name shown in the OpenEIS UI to a user.

The **required\_input** method is called by the UI when staging applications to run. In this example, the application requires OutdoorAirTemperature, WholeBuildingEnergy, and NaturalGas sensors.

#### 9. Create the **output\_format** method:

```
@classmethod
def output_format(cls, input_object):
    """Describe output table format.
    """
    # super - call DrivenApplicationBaseClass parent
    # DriverApplicationBaseClass output_format method
    result = super.output_format(input_object)
    topics = input_object.get_topics
    # extract load from topic map: 'site/building/device/point'
    load_topic = topics['Load'][0]
    load_topic_parts = load_topic.split('/')
    output_topic_base = load_topic_parts[:-1]
    oat_topic = '/'.join(output_topic_base +
                        ['example_driven', 'OAT'])
    load_topic = '/'.join(output_topic_base +
                        ['example_driven', 'Load'])
    output_needs = {
        'example_driven': {
            'OAT': OutputDescriptor('float', oat_topic),
            'Load': OutputDescriptor('float', load_topic)
        }
    }
```

---

<sup>9</sup> **count\_min** – minimum number of required inputs for this sensor (currently importing more than one of any sensor is only supported in command line deployments).

**count\_max** – maximum number of required inputs for this sensor (currently importing more than one of any sensor is only supported in command line deployments).

```

# Driven apps. create commands table by default
# this update output to include this user defined
# table and return the resulting output_needs object.
result.update(output_needs)
return result

```

The **output\_format** method takes a database input\_object as an input and returns a dictionary containing output needs (i.e., what will the table, containing the application's analysis results, look like). The following list details some important aspects of the **output\_format** method:

10. Calling **get\_topics** method on the input\_object returns a dictionary or topic map.
11. The keys for this dictionary are the same as those specified in the **required\_inputs** method. For example, the dictionary topic\_map would contain three entries with the keys : *'OAT'*, *'Load'*, and *'natgas'*
12. The dictionary (topic map) contains values with the format “site/building/device/point” (e.g., *'OAT': 'site1/building1/AHU1/OutdoorAirTemperature'*).
13. Typically, this information is used to build an output\_needs dictionary, for example:

```

'table1': {
    'datetime': OutputDescriptor('string', date_topic),
    'analysis': OutputDescriptor('float', analysis_topic)
}

```

The date\_topic and analysis\_topic inputs to the **OutputDescriptor** would be in the format “site/building/device/table/output topic” for example:

```
'site1/building1/AHU1/table1/topic1'
```

This method is called by the UI when the application is staged.

14. Create the **reports** method:

```

@classmethod
def reports(cls, output_object):
    # Called by UI to create Viz
    """Describe how to present output to user
    Display this viz with these columns from this table
    """
    report = reports.Report('Report for Example Driven Application')

    text_blurb = reports.TextBlurb(text="Sample Text Blurb.")
    report.add_element(text_blurb)

    report_list = [report]
    return report_list

```

The **reports** method describes how the application will present its output (results) to the user. The UI calls this method in order to create the visualization that the user will see. This method describes the required inputs for several supported visualization (i.e., bar graph, pie chart, heat map, etc.).

15. Create the **run** method:

```
# method will receive time, datetime corresponding to inputs.
# input is a dictionary of key value pairs (1 row in a data set).
def run(self, time, inputs):
    # Instantiate instance of Results class.
    results = Results

    if self.first:
        results.log('First Post!', logging.INFO)
        self.first = False

    inputs['time'] = time
    # Use results object to store analysis results.
    results.insert_table_row('output', inputs)

    self.counter += 1
    results.command('/awesome/counter', self.counter)
    return results
```

This method is called on each time stamped row of data within a dataset. The required inputs to the run method are as follows (the variable names for the inputs to the **run** method may be changed as desired):

- inputs – dictionary of key (same keys that were used in the **required\_input** method) value (trend data) pairs.
- time – is a datetime object that corresponds to the timestamp associated with the row of data used in the inputs dictionary.

The run method must return a Results object. For more detail on **Results** class see: *opneis.applications.\_\_init\_\_.py*.

### 6.2.2 Results Class

When the **Results** class is instantiated, a “container” for a driven applications analysis results is created. The following is the contents of the **Results** class:

```
class Results:
    def __init__(self, terminate=False):
        self.commands = {}
        self.log_messages = []
        self._terminate = terminate
        self.table_output = defaultdict(list)
```

```

def command(self, point, value):
    self.commands[point]=value

def log(self, message, level=logging.DEBUG):
    self.log_messages.append((level, message))

def terminate(self, terminate):
    self._terminate = bool(terminate)

def insert_table_row(self, table, row):
    self.table_output[table].append(row)

```

The methods within the Results class store the analysis results and push these results to the OpenEIS database. The following is a description of each of these functions:

**command** – accepts a point (key) and value as input. The command method uses these inputs to create a dictionary that is subsequently written to a table (within OpenEIS). This function emulates device control. The key value pairs are the commands a device would receive (recommended commands) from the application (applications incorporated into the OpenEIS as driven applications were often developed to interact with real building equipment).

**log** – creates a log for the driven application to capture debug or other pertinent information.

**terminate** - when called the application will gracefully terminate (finish any pending analysis) and stop the application.

**insert\_table\_row** – accepts a point (key) and value as input. The **insert\_table\_row** method uses the input to create a dictionary that is subsequently written to a table (within OpenEIS). This is where the application stores data to create any reports (visualizations) for the application’s analysis results.

## 6.3 Driver Applications

Unlike driven applications, a driver application is not meant to emulate being run on live data. The driver application can access the database, does not use the **Results** class to store its analysis results, and it has more control over its invocation. A developer creates a driver application by extending the abstract class **DriverApplicationBaseClass**.

### 6.3.1 Example Driver Application

This section will detail the creation of an example driver application.

1. Open an IDE (this example will utilize Eclipse) or text editor (e.g., notepad++) and create a new python file in the applications directory. In the Eclipse menu, select:

➤ File -> New -> PyDev Module (Figure 55)

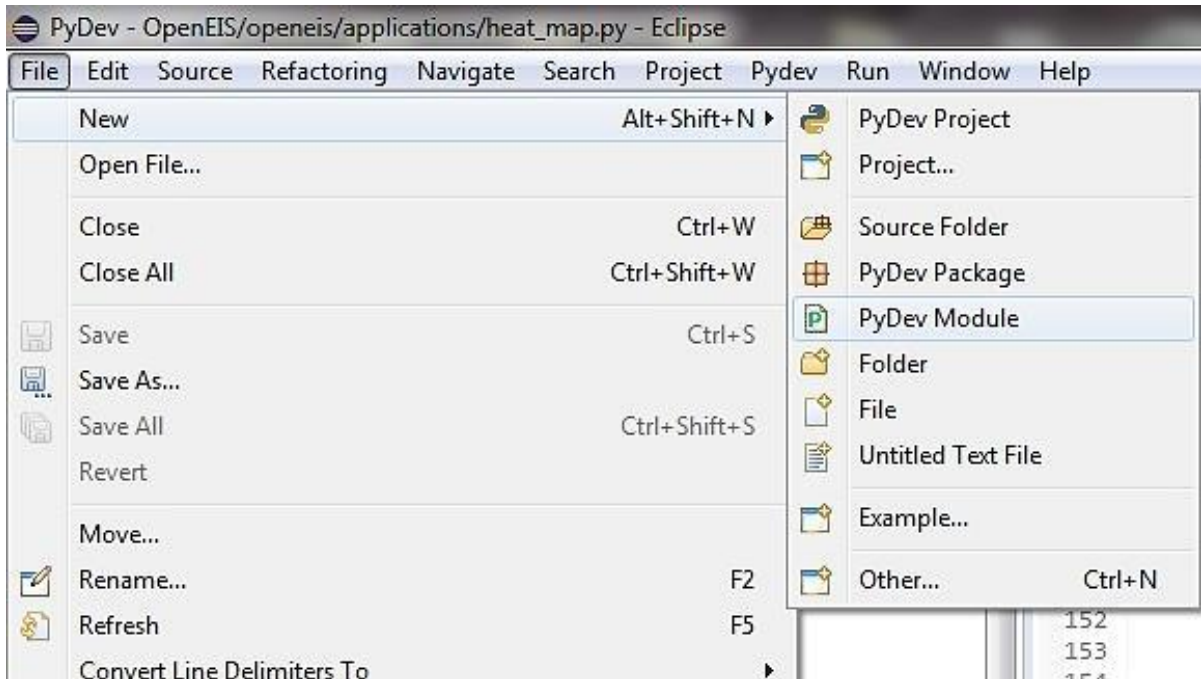


Figure 55: Creating an example driver application

Tip: Another way to do this in Eclipse is to right-click on the openeis.applications package in the PyDev Package Explorer and in the context menu select:

- New -> PyDev Module (Figure 56)

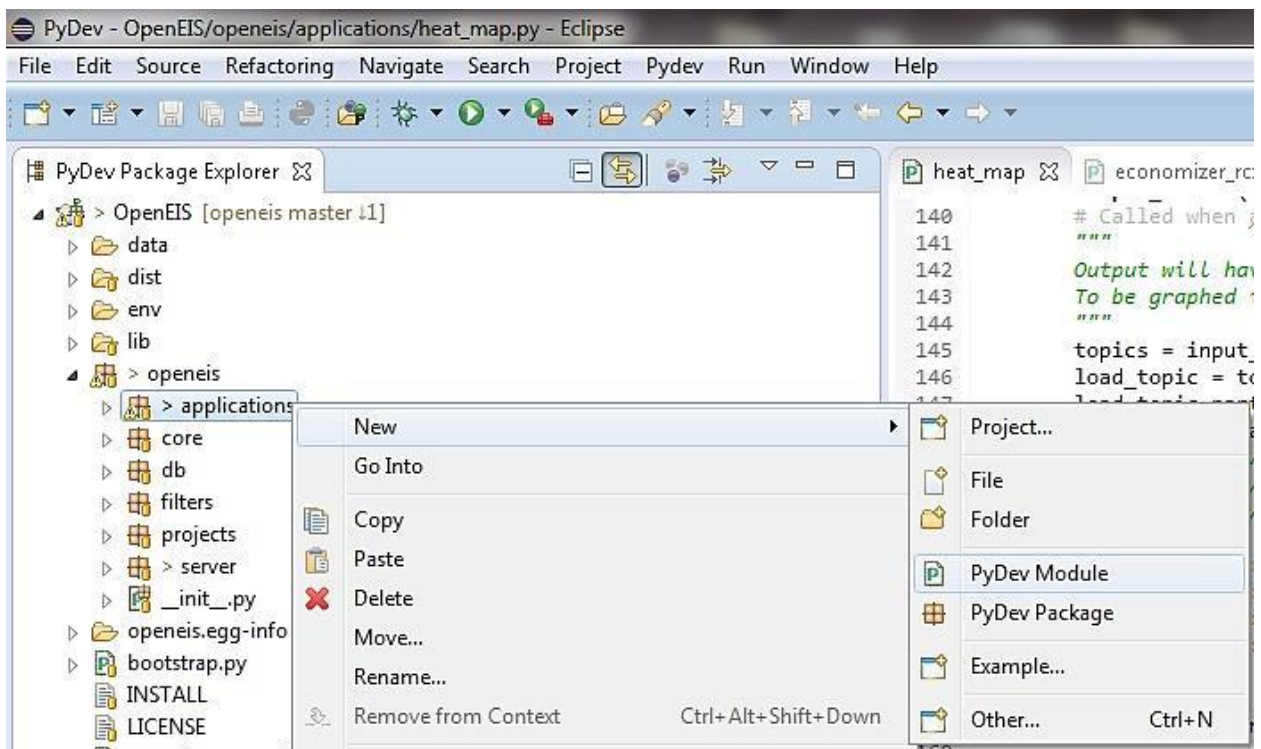


Figure 56: Creating a new Driver Application from the PyDev Package Explorer

2. On the new Python module screen, enter the following information (Figure 57):

Source Folder: /OpenEIS

Package: *openeis.applications*

The application name can be anything but should reflect the nature of the application. In this example, the application is named “example\_driver”. It is important that the application file is located in the *openeis/applications* directory.

- Click Finish

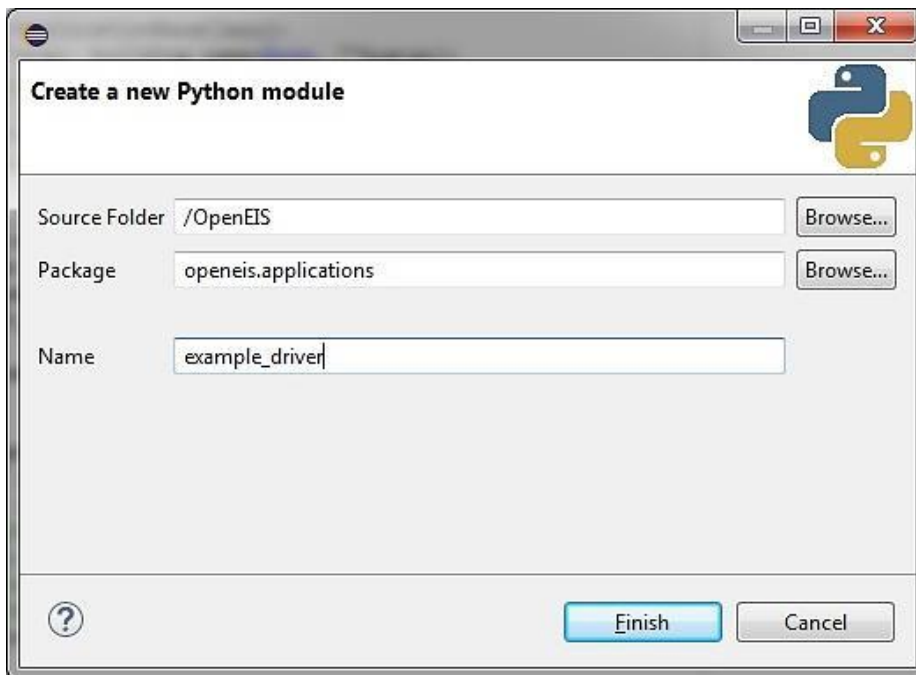


Figure 57: Creating a new Driver Application (continued)

3. Next, declare the modules to import for use by the application. The following import list is typical for driver application's:

```
import logging
import pytz
from django.db.models import Avg
from openeis.applications import (DriverApplicationBaseClass,
                                   OutputDescriptor,
                                   ConfigDescriptor,
                                   InputDescriptor,
                                   Descriptor,
                                   reports)
```

The logging module is part of the Python standard library. Logging is a means of tracking events that happen when some software runs. The software's developer adds logging calls to

their code to indicate that certain events have occurred. An event is described by a descriptive message, which can optionally contain variable data.

The `pytz` module allows accurate and cross platform time zone calculations using Python 2.4 or higher. It also solves the issue of ambiguous times at the end of daylight saving time, which you can read more about in the Python Library Reference

The remaining imports are OpenEIS modules that are required and will be explained in detail as the “`example_driver`” application is built.

`django.db.models`, contains an **Avg** function that will be used in the `execute` method. The remaining imports are for various OpenEIS modules.

4. In the file, create a new class called **Application**. **Application** should inherit from the **DriverApplicationBaseClass** (located at `openeis/applications/__init__.py`).

```
class Application(DriverApplicationBaseClass):
```

Applications within the OpenEIS are required to have an **Application** class.

5. Create the `__init__` method:

```
def __init__(self, *args, building_name=None, **kwargs):
    """
    When applications extend this base class, they need to make
    use of any kwargs that were setup in config_param
    """
    super.__init__(*args, **kwargs)
    self.default_building_name_used = False
    if building_name is None:
        building_name = 'None supplied'
        self.default_building_name_used = True

    self.building_name = building_name
```

This method has the default arguments (`*args` and `**kwargs`)<sup>10</sup>, and also takes `building_name` (building name) as a keyword argument. The body of the method contains a call to **super**<sup>11</sup> and some basic error handling for the `building_name` parameter.

6. Next, create the `get_self_descriptor` method:

```
@classmethod
def get_self_descriptor(cls):
    name = 'Example Driver Application - Electricity Map'
```

---

<sup>10</sup> <https://freepythontips.wordpress.com/2013/08/04/args-and-kwargs-in-python-explained/>

<sup>11</sup> <https://docs.python.org/2/library/functions.html#super>

```

desc = ('This is an example driver application. '
        'This method returns a Descriptor used by the UI on the run '
        'analysis screen. This Electricity Map is similar to Heat '
        'Map, however it calculates WholeBuildingElectricity')
return Descriptor(name=name, description=desc)

```

This method returns a **Descriptor** object. The **Descriptor** object should contain two keyword arguments, name and description. The name should be a human intelligible name for the application (or custom filter) and the description should be a short description of what the application does. This text is displayed to users in the OpenEIS UI and is how a user will identify the application.

Noteworthy is that the **DrivenApplicationBaseClass** defined in `/openeis/applications__init__.py` inherits from both **ConfigDescriptorBaseClass** and **SelfDescriptorBaseClass**. These descriptor classes are found in `openeis/core/descriptors.py`.

#### 7. Create the `get_config_parameters` method:

```

@classmethod
def get_config_parameters(cls):
    # Called by UI
    return {
        "building_name": ConfigDescriptor(str,
                                          "Building Name", optional=True)
    }

```

The `get_config_parameters` method returns a dictionary with information about an application's required configuration parameters. This method is called by the UI when constructing the application configuration page, the menu where users input configuration parameters for an application.

The `ConfigDescriptor` object contains the following arguments:

- `config_type` – data type for the configuration parameter. In our example, the data type is a string (str).
- `display_name` – Name displayed to user in the UI. In our example, the data type is a string ("*Building Name*").
- `description` – Text displayed to user as description of parameter. In our example, the data type is a string ("*Building Name*").
- `optional` – Keyword argument indicating if the parameter is optional. The default value for optional argument is False.

- `value_default` – the default value for the configuration parameter. This value will automatically propagate into the application configuration menu in the OpenEIS UI. This point is optional; the default behavior is there is no minimum default value for the configuration parameter.
- `value_min` – this will set a lower bound on the configuration parameter. The user will not be allowed to enter a smaller value for the configuration parameter. This point is optional; the default behavior is there is no minimum value for the configuration parameter.
- `value_max` – this will set an upper bound on the configuration parameter. The user will not be allowed to enter a larger value for the configuration parameter. This point is optional; the default behavior is there is no maximum value for the configuration parameter.
- `value_list` – a list of valid inputs for the configuration parameters. These values will appear in a dropdown for the configuration parameters. The user will only be able to choose from these values. This point is optional; the default behavior is that any value (with the correct `config_type`) can be used.

This method allows a user to understand and customize configuration parameters for an application. An OpenEIS user can interact with the application and through this configuration make the application more useful.

Please note:

- This method must be implemented because class **Application** inherits from **ConfigDescriptorBaseClass**.
- Noteworthy is that the **DriverApplicationBaseClass** defined in *openeis.applications.\_\_init\_\_.py* inherits from both **ConfigDescriptorBaseClass** and **SelfDescriptorBaseClass**. These descriptor classes are found in *openeis/core/descriptors.py*.

8. Create the `required_input` method:

```
@classmethod
def required_input(cls):
    '''Returns a dictionary of required data.'''
    return {
        'electricity': InputDescriptor('WholeBuildingElectricity',
                                       'Building Electricity')
```

```
}
```

The **required\_input**<sup>12</sup> method returns a dictionary with information on the required data to run the application.

The InputDescriptor is described in *openeis/core/descriptors.py* and takes the following parameters:

- **sensor\_type** – The sensor type is the name of the input sensor. This will be a string containing the OpenEIS name for a sensor. The UI uses this to verify that the required input data is present, then the UI makes the application available to the user.
- **display\_name** – The display name is a human intelligible name shown to a user in the OpenEIS UI.

This method is called by the UI to determine what input(s) are required to run the application. In this case, the application requires a WholeBuildingElectricity sensor.

#### 9. Create the **output\_format** method:

```
@classmethod
def output_format(cls, input_object):
    # Called when app is staged
    ...
    Output will have date, hour, and electricity use, used in a map later.
    ...

    topics = input_object.get_topics
    load_topic = topics['electricity'][0]
    load_topic_parts = load_topic.split('/')
    output_topic_base = load_topic_parts[:-1]
    date_topic = '/'.join(output_topic_base+['heatmap', 'date'])
    hour_topic = '/'.join(output_topic_base+['heatmap', 'time'])
    load_topic = '/'.join(output_topic_base+['heatmap', 'electricity'])
    output_needs = {
        'Heat_Map': {
            'date': OutputDescriptor('string', date_topic),
            'hour': OutputDescriptor('integer', hour_topic),
            'electricity': OutputDescriptor('float', load_topic)
        }
    }
    return output_needs
```

---

<sup>12</sup> **count\_min** – minimum number of required inputs for this sensor (currently importing more than one of any sensor is only supported in command line deployments).

**count\_max**– maximum number of required inputs for this sensor (currently importing more than one of any sensor is only supported in command line deployments).

The **output\_format** method takes a database input\_object as an input and returns a dictionary containing output needs (i.e., what will the table containing the application's analysis results look like). The following list details some important aspects of the **output\_format** method:

- Calling **get\_topics** method on the input\_object returns a dictionary or topic map.
- The keys for this dictionary are the same as those specified in the **required\_inputs** method. For example, the dictionary topic\_map would contain three entries with the keys : *'OAT'*, *'Load'*, and *'natgas'*
- The dictionary (topic map) contains values with the format “site/building/device/point” (e.g., *'OAT': 'site1/building1/AHU1/OutdoorAirTemperature'*).
- Typically, this information is used to build an output\_needs dictionary, for example:

```
'table1': {
    'datetime': OutputDescriptor('string', date_topic),
    'analysis': OutputDescriptor('string', analysis_topic)
}
```

The date\_topic and analysis\_topic inputs to the OutputDescriptor would be in the format “site/building/device/table/output topic” for example:

```
'site1/building1/AHU1/table1/topic1'
```

- This method is called by the UI when the application is staged.

10. Create the **reports** method:

```
def reports(self):
    """Describe how to present output to user
    Display this viz with these columns from this table

    display_elements is a list of display objects
    specifying viz and columns for that viz
    """
    report = reports.Report('Heat Map for Building Energy Electricity')
    text_blurb = reports.TextBlurb(text=("Analysis of the extent of a "
                                         "building's daily, weekly, and "
                                         "seasonal shut off."))
    report.add_element(text_blurb)
    heat_map = reports.HeatMap(table_name='Heat_Map',
                               x_column='hour',
                               y_column='date',
                               z_column='electricity',
                               x_label='Hour of the Day',
```

```

        y_label='Date',
        z_label='Building Energy [kWh]')
report.add_element(heat_map)
text_guide1 = reports.TextBlurb(text=('Horizontal banding '
'indicates shut off during '
'periodic days (e.g. weekends).''))

report.add_element(text_guide1)
text_guide2 = reports.TextBlurb(text="Unusual or unexplainable
\\"hot spots\\""
may indicate poor equipment
control.")
report.add_element(text_guide2)

text_guide3 = reports.TextBlurb(text="Vertical banding indicates
consistent"
daily scheduling of usage.")

report.add_element(text_guide3)
report_list = [report]
return report_list

```

The **reports** method describes how the application will present its output (results) to the user. The UI calls this method to create the visualization that the user will see. This method describes the required inputs for several supported visualizations (i.e., bar graph, pie chart, heat map, etc.). The following reports method describes the creation of a heat map (carpet plot).

The visualization is created by instantiating an instance of the reports class and adding elements by calling **report.add\_element**. For example, to add the heat map element:

```

heat_map = reports.HeatMap(table_name='Heat_Map',
    x_column='hour',
    y_column='date',
    z_column='electricity',
    x_label='Hour of the Day',
    y_label='Date',
    z_label='Building Energy [kWh]')

```

Then add the heat map to the report:

```
report.add_element(heat_map)
```

Similarly, for a bar graph, one would use: **bar\_graph = reports.BarChart**, and add the element with **report.add\_element(bar\_graph)**. The **TextBlurbs** contain descriptive text and are also added as elements. Finally, the entire report object is put into a list and returned.

For more information, the **Report** class is located at:

*openeis/applications/reports/\_\_init\_\_.py*

11. Create the **execute** method. This function should contain the application's core functionality. This is where the application uses data to create meaningful results.

```
def execute(self):
    """
    Example driver, take electricity
    load data, group and create carpet plot.
    """
    self.out.log("Starting application: heat map.",
                 logging.INFO)

    self.out.log("Querying database.", logging.INFO)
    loads = self.inp.get_query_sets('electricity', group_by='hour',
                                    group_by_aggregation=Avg,
                                    exclude={'value': None})

    base_topic = self.inp.get_topics

    self.out.log("Compiling the report table.", logging.INFO)
    for x in loads[0]:
        datevalue = (
            self.inp.localize_sensor_time(
                base_topic['electricity'][0], x[0]))

        self.out.insert_row("Heat_Map", {
            'date': datevalue.date,
            'hour': datevalue.hour,
            'electricity': x[1]
        })
    )
```

The **execute** method is called when the driver application is run. For this example, the first few statements use the built-in Python logging module to record some descriptive messages. From here, we get the electricity loads, grouped by hour, using **self.inp.get\_query\_sets** (see Section 6.1). Then we loop through the electricity loads and insert them into the heat map.

## 7 OpenEIS Report Elements/Visualizations

The OpenEIS allows developers to create visualizations from data analysis results. Several stock visualization are supported (i.e., tables, line plot, scatter plot, carpet plot, and bar chart). A simple visualization API allows user to declare visualization types and attributes (e.g., axes titles, etc.). This section will describe the elements of the visualization API (**Report** class).

Code for the UI is in a separate repository at: <https://github.com/VOLTTRON/openeis-ui>

The focus of this guide is on development of applications, but an overview of visualizations is provided as a starting point for those familiar with JavaScript.

### 7.1 Creating a Stock Visualization

This section will describe the steps necessary to create a stock (available in OpenEIS by default) visualization to represent application analysis results.

#### 7.1.1 On Server

Each report element is a sub-class of **openeis.applications.reports.ReportElement**. Class properties are serialized and returned by the API as part of analysis reports.

See: *openeis/applications/reports/\_\_init\_\_.py*

#### 7.1.2 In Client

Each report element of an analysis report is converted to HTML markup by the `analysisReport` directive.

Within the “link” function of the directive, there are three variables relevant to visualizations:

1. **element**: DOM element to which markup is to be appended
2. **scope.arReport**: the analysis report
3. **scope.arData**: all output data (tables) of the analysis, keyed by table name

The link function loops through the report elements of the analysis report (**scope.arReport.elements**), converts them to markup, and appends them to **element**.

See: *openeis-ui/src/directives/analysisReport.js*

### 7.2 Included Visualizations

OpenEIS includes some simple stock visualizations (in `analysisReport.js`) for use by applications:

- **TextBlurb** – Simple text element for use as a label
- **LinePlot** – Plots an XY dataset as a line
- **BarChart** – Plots an XY dataset as bars

- ScatterPlot – Produces a scatterplot of an XY dataset
- DatetimeScatterPlot – XY dataset with datetime
- HeatMap – Heatmap with XYZ dataset

## 7.3 Adding Visualizations

Adding a new visualization potentially requires changes to both the server and client. The server must have code for returning report elements and the client must have code for making use of it. The JavaScript defines how to produce the visualization (*analysisReport.js*). The Report classes on the server are filled out by applications to send to the JavaScript via calls the server.

### 7.3.1 On Server

1. Create sub-class of **openeis.applications.reports.ReportElement**.

```
class MyReportElement(ReportElement):
    def __init__(self, someArg, **kwargs):
        super.__init__(**kwargs)
        self.someProp = someArg
```

2. Use the created class in an application, in the “reports” method. (See “Creating the reports method” under Example Driven Application (Section 6.2.1) or Example Driver Application (Section 6.3.1).

For more information on reports and report elements, see *openeis/applications/reports/\_\_init\_\_.py*. A suitable application to test is heat map, *openeis/applications/heat\_map.py*.

### 7.3.2 In Client

1. In the *analysisReport* directive (*openeis-ui/src/directives/analysisReport.js*), add the created class name as a case in the switch block and append to **element** the markup necessary to display the report element, based on the element’s parameters and the data of the analysis report.

```
switch (reportElement.type) {
    ...
    case 'MyReportElement':
        var someText = angular.element('<div/>');

        someText.addClass('my-report-element');
        someText.text('The element says ' + reportElement.someProp);

        element.append(someText);
        break;
    ...
}
```

```
}
```

For text-based report elements, the markup may consist of basic HTML (e.g. TextBlurb, Table), but for graphical report elements, Scalable Vector Graphics (SVG) markup will need to be generated. D3.js (<http://d3js.org/>) is used by included visualizations and is highly recommended for generating SVG markup.

2. Add any necessary styling to *openeis-ui/src/scss/objects.analysis-report.scss*.

```
.my-report-element {  
  color: green;  
  font-weight: bold;  
}
```

### 7.3.3 Build, Run, and View

1. Follow the instructions in *openeis-ui/README.md* to install dependencies for building the UI and override the UI module bundled with OpenEIS.
2. Start continuous development build of UI. (i.e., “[openeis-ui] \$ grunt”)
3. Start OpenEIS server.
4. In browser, go to *http://localhost:8000/*
5. If necessary, create account and project, upload files, and create data map and dataset.
6. Run application utilizing your report element.
7. Open resulting analysis report.

## 8 Additional Support

This document is meant to give developers an introduction to writing applications for and extending the base OpenEIS system. Any questions or comments can be directed to the issue tracker at the project GitHub site: <https://github.com/VOLTTRON/openeis>. The development team can also be contacted at [volttron@pnnl.gov](mailto:volttron@pnnl.gov).