Flask-Diamond Documentation

Release 0.5.1

Ian Dennis Miller

November 02, 2017

Contents

Flask-Diamond is a batteries-included Python Flask framework, sortof like Django but radically decomposable. **Flask-Diamond** offers some opinions about network information systems that process data. Using **Flask-Diamond**, you can scaffold a working application with sensible defaults, then easily override those defaults to meet your own goals. **Flask-Diamond** provides a shared vocabulary that helps teams coordinate as they scale up to develop multiple Flask applications while maintaining good code reuse and learning transfer. **Flask-Diamond** goes beyond a "project scaffold" by providing a complete architecture and team solution, including documentation, tutorials, and other learning support.

CHAPTER 1

Overview

A Flask-Diamond application consists of *facets*, which are common facilities that many applications eventually need to provide. The *facets* provided by Flask-Diamond include:

- account management
- administrative access
- databases
- Model object CRUD
- email
- testing
- documentation
- deployment
- and more

Usage

The following steps will create a new Flask-Diamond application.

```
pip install Flask-Diamond
mkdir my-application
cd my-application
flask-diamond scaffold app
make install docs test db server
```

Please read the Introduction Documentation to get started.

Documentation

http://flask-diamond.org

Quick Start Screencast

Length: 3:17

CHAPTER 2

Get Started

Start here. These introductory readings will teach you how to install Flask-Diamond and how to read the rest of the Flask-Diamond documentation.

Quick Start

Flask-Diamond installs in a Python environment with *virtualenv*. Please see the *System Requirements* for information about installation pre-requisites.

Screencast

Length: 3:13

Install Flask-Diamond

Create a *virtualenv* for your application and install Flask-Diamond.

```
mkdir my-application
cd my-application
mkvirtualenv -a . my-application
pip install Flask-Diamond
```

If any of these steps do not work, review the *System Requirements* and ensure everything is installed.

Scaffold a new Flask-Diamond application

Enter the virtual environment and scaffold a new Flask-Diamond application. For this *Quick Start*, just use the default options.

```
workon my-application
flask-diamond scaffold app
```

Use it!

Test the installation to ensure everything is installed correctly:

```
make test
```

Create the application database:

```
make db
```

Start the application server:

```
make server
```

You now have a server running at http://127.0.0.1:5000. Depending on your OS, the following command will open your application in a browser.

```
open http://localhost:5000
```

Login with the following account details:

- username: **admin@example.com**
- password: **the simple_password specified during scaffolding**

Next Steps

Now that you have scaffolded your first Flask-Diamond application, you can read about how to *Learn Flask-Diamond* by using the online documentation and materials. If you prefer to learn by experimenting with a live example, see *Planets Tutorial* for a hands-on introduction to an working Flask-Diamond application.

Learn Flask-Diamond

The best way to learn Flask-Diamond is to read the *developers-guide*; contained within is a growing list of documents that explain various aspects of the Flask-Diamond approach. To help new learners focus on the fundamentals, read the following documents first.

- *Quick Start* shows you how to scaffold a new application. You can be up and running in just a few minutes.
- *Project Initialization and Scaffolding* offers many options for you to customize your application. Read this document to learn more about answering the questions during scaffolding.
- *Application Facets* describes the use of Flask-Diamond's *facets* for customizing your application's behavior.
- *Writing an Application with Flask-Diamond* provides examples and describes an approach for designing and programming an application that achieves your goals.
- *MVC: Model, View, Controller* is a more advanced document that describes the Flask-Diamond architecture. Model-View-Controller (MVC) is widely used in software engineering to write applications that provide a user interface. Once you understand how to implement MVC using Flask-Diamond, you will be able to write applications for a wide range of domains.

Learning Python

If you are relatively new to Python, you may find that there are advanced computer science principles that Flask-Diamond requires some knowledge of. In particular, Flask-Diamond applications are heavily Object Oriented and make extensive use of Inheritance.

There are many tutorials online that can help you to learn about these fundamentals. In particular, I recommend the following:

- Dive Into Python
- Learn Python the Hard Way

System Requirements

Flask-Diamond requires some software to be installed in order to function. Once you have installed these requirements, you can follow the *Quick Start* to start your first project. The following packages should be installed globally, as the superuser, for all users on the system to access.

- Python 2.7.x or 3.4 and above.
- Python development libraries (i.e. header files for compiling C code)
- pip
- virtualenv
- virtualenvwrapper

The following sections describe the process for installing these requirements on various systems. In each of the following examples, it is assumed you will be using a root account (or some other privileged account).

If you do not have root access, then refer to the section *Unprivileged Installation* for information about creating a virtualenv in your user account.

Debian/Ubuntu

Flask-Diamond installs cleanly on Debian and Ubuntu systems released after 2011.

```
apt-get install python python-dev python-pip build-essential
apt-get install sqlite-dev
pip install --upgrade pip
pip install --upgrade virtualenv
pip install virtualenvwrapper
```

Redhat

Flask-Diamond can be installed on RedHat, but ensure your package manager is installing Python 2.7; as of August 2015, RHEL provides an older version.

```
yum install python python-devel python-pip
yum install sqlite-devel
pip install --upgrade pip
pip install --upgrade virtualenv
pip install virtualenvwrapper
```

OSX with Homebrew

Flask-Diamond installs pretty easily on OSX with Homebrew. Make sure you are using the *admin* user for this process, just like a normal Homebrew operation.

```
brew install python --universal --framework
brew install pyenv-virtualenv
brew install pyenv-virtualenvwrapper
brew install sqlite
pip install --upgrade pip
```

Windows with Cygwin

Note: Have you done this install successfully? Please share your process as a comment on Issue 8.

Here are a few resources to get you started:

- http://www.pdxpixel.com/blog/setting-up-python-and-virtualenv-windows-cygwin/
- http://atbrox.com/2009/09/21/how-to-get-pipvirtualenvfabric-working-on-cygwin/
- http://anythingsimple.blogspot.ca/2010/04/using-pip-virtualenv-and.html
- http://stackoverflow.com/questions/2173963/how-do-i-get-virtualenvwrapper-and-cygwin-to-co-operate

Unprivileged Installation

Sometimes, you do not have root access to the system. It is still possible to use Flask-Diamond, but the installation process is slightly different because it does not use virtualenvwrapper. Instead, you will create your virtualenv directly and use the *activate* macro to work on it.

```
curl -O https://raw.github.com/pypa/virtualenv/master/virtualenv.py
python virtualenv.py my-diamond-app
source my-diamond-app/bin/activate
pip install Flask-Diamond
```

CHAPTER 3

Basics

Learn how to create a new project. This section introduces the Flask-Diamond philosophy and the first steps for making an application.

Planets Tutorial

The Planets Tutorial will introduce Flask-Diamond by demonstrating a simple web application that manages Planets and Satellites in a database. When you are done with this tutorial, you will understand a powerful pattern for using Flask-Diamond to develop applications. This tutorial assumes you have already installed the *System Requirements*.

Screencast

Length: 11:03

Overview

The Planets Tutorial consists of two parts:

1. This document, which contains instructions and explanations
2. The Planets code scaffold, which can be installed from the command line

Simply follow the tutorial and we'll install everything, then use it to demonstrate some useful principles about Flask-Diamond applications.

During the tutorial, we will accomplish the following:

1. Scaffold a working "Planets" application
2. Use the application to create a few Planets and Moons in the application database
3. Alter the Model code to make the GUI behave differently
4. Introduce several important files that are used to structure a Flask-Diamond application

Setup for Tutorial

The following commands will prepare you to begin the tutorial:

```
mkdir planets
cd planets
mkvirtualenv -a . planets
pip install Flask-Diamond ipython
flask-diamond scaffold app
```

For a more detailed explanation of these commands, please see the *Quick Start*. When you run `flask-diamond scaffold app`, you will be prompted to answer some questions. Give your application and module the name of `planets` during scaffolding. For all other questions, you can accept the default answers by pressing Enter for each question.

Tutorial Exercises

Now, we'll use a special scaffold called `tutorial-planets` for the rest of this tutorial. The `tutorial-planets` scaffold places some example models and views into the application structure.

```
flask-diamond scaffold tutorial-planets
make test db server
open http://localhost:5000
```

Your web browser should now display the `planets` application. If you ran into problems, please review the *Quick Start* to ensure you have all the requirements installed and working.

The Planets application comes with two object classes:

1. *Planets*, which are large celestial bodies that orbit something like a star
2. *Satellites*, which are little bodies orbiting planets

We will use these classes to model our solar system.

Create Planet Earth

The first thing to do is enter the application shell. The `make shell` command enters the application context, connects to the database, and starts an interactive shell that will allow us to interact with our application.

```
make shell
```

Enter the following commands to create Earth.

```
from planets import models
earth = models.Planet.create(name="Earth", mass=100.0)
```

We have provided two parameters to the Planet.create method: *name* and *mass*. These model parameters come from the Planet model class definition, which we will investigate in the next section.

Also create Earth's lunar body, the Moon.

```
moon = models.Satellite.create(name="Moon", mass=25.0, planet=earth)
```

Take note of the additional parameter: *planet*. The application database now contains the Earth and the Moon.

Inspect Models

Using a text editor, inspect the files in `planets/models`.

- `__init__.py` proxies all model classes

- `planet.py` contains definition of Planet class
- `satellite.py` contains definition of Satellite class

The *Planet* model enables us to capture the name and mass of a planet in the application database. The *Satellite* model is similar to the Planet model, but it also includes a foreign key relationship so that satellites may belong to planets. See *Facet: Database* for more about how to write model classes.

Administration GUI

Log in to GUI

Using a web browser, connect to the application server in a new tab. If you used the default scaffolding settings, your application server is online at http://localhost:5000/.

First, log in as `admin@example.com` using randomly generated password. The development password can be recovered from `Makefile`.

Create Mars

Now that you have logged in, create a new Planet called Mars using the GUI. Choose *Admin* from the drop-down menu at the top of the screen. Select the *Planet* model from the menu. Once the Planets List View has loaded, click *Create* to make a new planet. Use the following values:

- name: Mars
- mass: 90.0

Create Phobos

Repeat this process to create a new Satellite using the menus.

- name: Phobos
- mass: 10.0
- planet: Mars

However, you will run into trouble when you try to set the planet to Mars. To fix this, open the file `models/planet.py`. Add a function called `__str__` within the Planet class:

```
def __str__(self):
    return self.name
```

With the string representation function in place, try to create Phobos again. You will now be able to select *Mars* from the drop-down.

Inspect `__init__.py`

The last step in this tutorial is to look at the most important file of all, `__init__.py`, which controls every aspect of your application. Using a text editor, inspect the file `planets/__init__.py`. Flask-Diamond applications mostly follow Flask's `create_app()` pattern. If you are not yet familiar with Flask applications, read *Writing an Application with Flask-Diamond*.

blueprints facet

Take a look at `init_blueprints`, which registers two blueprints that provide basic administrative functionality to your application. To add new views to your application, you will extend this function to register your own blueprints.

administration facet

Finally, look at `init_administration`, which adds a `ModelView` for Planets and Satellites. When you create new models in your application, if you wish to edit those models using the GUI, you will need to add those new models to `init_administration`.

Tutorial Conclusion

To recap this tutorial, we covered the following:

- scaffold a new application
- use data model to create objects in our database
- use web GUI to create even more objects
- edit the data model code to add functionality
- inspect `__init__.py` to learn how applications are controlled

These fundamental ideas are common to many applications. Of course, this tutorial is just an introduction. Each of these topics has many more readings that will help you learn to master the facets of your application.

Next steps

- *Application Facets* describes the use of Flask-Diamond's *facets* for customizing your application's behavior.
- *Writing an Application with Flask-Diamond* provides examples and describes an approach for designing and programming an application that achieves your goals.
- *MVC: Model, View, Controller* is a more advanced document that describes the Flask-Diamond architecture. Model-View-Controller (MVC) is widely used in software engineering to write applications that provide a user interface. Once you understand how to implement MVC using Flask-Diamond, you will be able to write applications for a wide range of domains.

Project Initialization and Scaffolding

The quickest way to start a new Flask-Diamond project is to use the scaffolding system. Scaffolding applies a series of file templates that result in a starter application that can be further customized. This document discusses the configuration questions that are used during scaffolding.

As described in the *Quick Start*, the basic process looks like this:

```
mkdir my-application
cd my-application
mkvirtualenv -a . my-application
pip install Flask-Diamond
flask-diamond scaffold app
make install docs test db server
```

About Scaffolding

When you invoke `flask-diamond`, a template is automatically applied. Using mr.bob, a brief set of questions are used to populate the templates with variables. When you answer these questions, your choices are stored in a file called `.mrbob.ini` that is located in the root folder of your project.

Template questions

This template is suitable for supporting many types of Python projects. It will create a basic directory structure that provide requirements management, documentation, build, deployment, and other basic project needs. This step of the scaffolding process asks the following questions:

1. What is the name of the application (i.e. username, daemon name, etc)?

Example: `Flask-Diamond`

The application name is a human-readable name that will be used in documentation, on websites, and when talking about the project. This should follow conventions that are established within your community. In the case of **Flask-Diamond**, the project name tries to follow the naming pattern established by other Flask extensions like Flask-Admin, Flask-Security, and Flask-RESTful.

2. What is the name of the module (can be different from the application name)?

Example: `flask_diamond`

The module name is a PEP8 styled name. In the case of Flask-Diamond, the module name should be lowercase and hyphens become underscores. Thus, Flask-Diamond becomes **flask_diamond**.

3. What is the short description for this project?

Example: `Flask-Diamond is a batteries-included Flask framework.`

This question corresponds to the `setuptools.setup(description="")` variable in the project setup.py file. The short description is intended to appear as a brief summary in PyPI.

4. What is the long description for this project?

Example: `Flask-Diamond is a batteries-included Flask framework.  Easily scaffold a working application with sensible defaults, then override the defaults to customize it for your goals.`

This question corresponds to the `setuptools.setup(long_description="")` variable in the project setup.py file. In practice, this usually ends up being multiple paragraphs.

5. Who is the author of this software?

Example: `Ian Dennis Miller`

This question corresponds to the `setuptools.setup(author="")` variable in the project setup.py file. It's a name. Your name.

6. What is the author's contact email?

Example: `iandennismiller@gmail.com`

This question corresponds to the `setuptools.setup(author_email="")` variable in the project setup.py file. This is your public contact information.

7. What is the author's contact URL?

Example: `http://flask-diamond.org`

This question corresponds to the `setuptools.setup(url="")` variable in the project setup.py file. If you have a website that provides support for your project, put it here. In the case of Flask-Diamond, there are lots of resources on the official website, including a link to the project issue tracker. As a result, Flask-Diamond uses the project URL as the contact URL.

8. Which port will the daemon listen on?

Example: `5000`

Your application will run as an HTTP service that listens on the port provided here. Thus, if you answer `8000` you will be able to connect to your application at `http://localhost:8000/admin`.

Automatically generated scaffolding fields

There are some steps in the scaffolding process that are automatically generated for you because they involve random processes.

1. What is the secret key?

Example: `\x83.RH\xdc@\x0fu\xb5o\xcd\xf5\xc4\xd5\xb12\xc2M\xca\x96\xc8\xbf\xeb\xde`

Flask uses a secret key to seed certain cryptographic functions. To generate a suitable random string for the secret key, use the following Python code:

```
python -c 'import os; print(repr(os.urandom(24)))'
```

2. What is the hash_salt?

Example: `t52ybrp0oOGHkQEZ`

Flask uses a hash salt for password storage. To generate a suitable random string for the hash salt, use the following Python code:

```
python -c 'import string as s, random as r; \
    print repr("".join(r.choice(s.letters+s.digits) for _ in range(16)))'
```

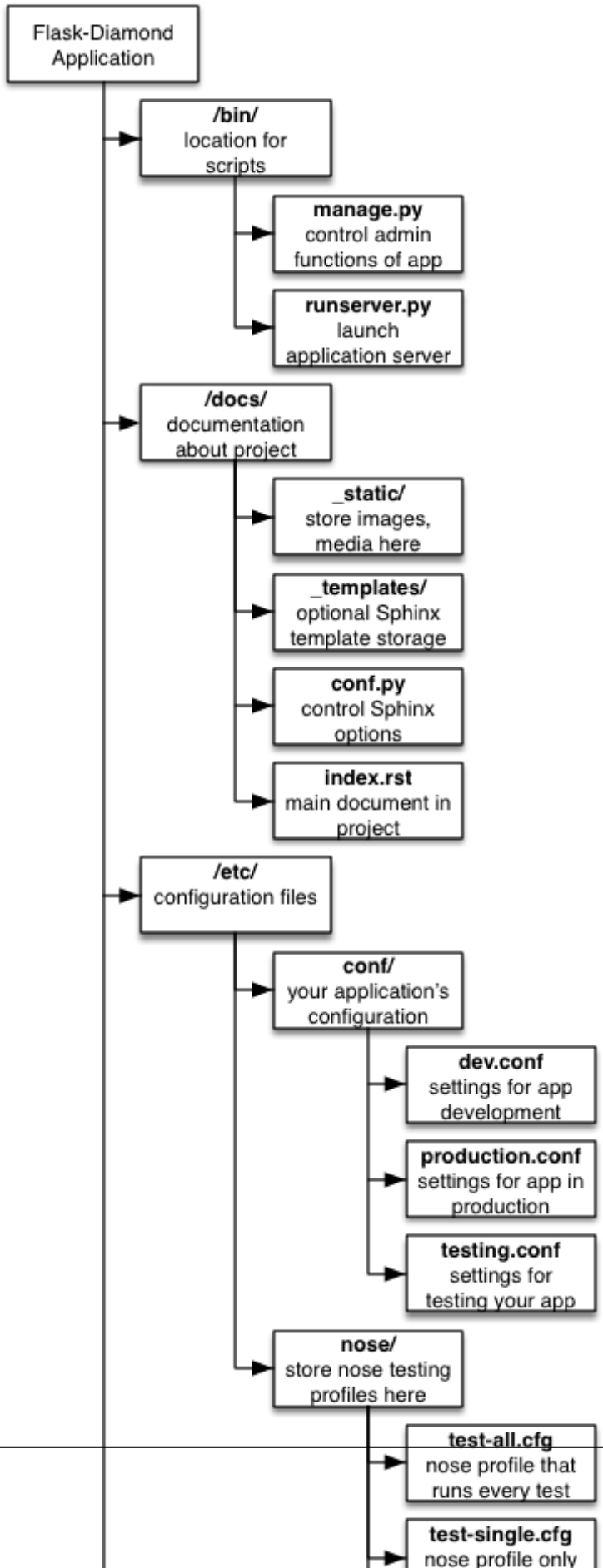
3. What is the simple_password?

Example: `abc`

Flask-Diamond can simplify the creation of testing accounts during development. One of the default accounts has administrative privileges, so a simple password is used to protect the account. This password is just 3 letters long by default, so it should never be used anywhere but during development. To ensure this weak password is not used in production, the `Makefile` is hardcoded to use the development configuration, only.

Diagram of a Flask-Diamond Scaffold

When a new Flask-Diamond project is scaffolded, it will generate files in roughly the following layout. The `SVG scaffold diagram` is also available for download.


Flask-Diamond Application
/bin/ location for scripts
manage.py control admin functions of app
runserver.py launch application server
/docs/ documentation about project
_static/ store images, media here
_templates/ optional Sphinx template storage
conf.py control Sphinx options
index.rst main document in project
/etc/ configuration files
conf/ your application's configuration
dev.conf settings for app development
production.conf settings for app in production
testing.conf settings for testing your app
nose/ store nose testing profiles here
test-all.cfg nose profile that runs every test
test-single.cfg nose profile only

Writing an Application with Flask-Diamond

By default, Flask-Diamond is ready to run when you initially scaffold a new application. This document describes the customization process, which transforms the default application into something tailored to your goals. We will start by discussing the default start-up routine for all Flask-Diamond applications, and then talk about modifying start-up to change its behavior.

A basic Flask-Diamond Example

The following example can be generated from a freshly scaffolded project by following the *Project Initialization and Scaffolding* document with the project name `MyDiamondApp`. Notice that the `MyDiamondApp` class inherits from `Diamond`, which gives the new project a lot of functionality "out of the box."

```
from flask.ext.diamond import Diamond, db
from flask.ext.diamond.facets.administration import AdminView, AdminModelView
app_instance = None
from . import models

class MyDiamondApp(Diamond):
    def init_blueprints(self):
        self.super("blueprints")
        from .views.administration.modelviews import adminbaseview
        self.app.register_blueprint(adminbaseview)

def create_app():
    global application
    if not application:
        application = Diamond()
        application.facet("configuration")
        application.facet("logs")
        application.facet("database")
        application.facet("marshalling")
        application.facet("accounts")
        application.facet("blueprints")
        application.facet("signals")
        application.facet("forms")
        application.facet("error_handlers")
        application.facet("request_handlers")
        application.facet("administration")
        # application.facet("rest")
        # application.facet("webassets")
        # application.facet("email")
        # application.facet("debugger")
        # application.facet("task_queue")
    return application.app
```

Customization with Inheritance

In the basic example above, notice the function `blueprints()`. This function handles the initialization of Flask blueprints. By overloading [1] this functions (and others) within your application, it is possible to customize the start-up behavior of any subsystem. Every subsystem in the Flask-Diamond start-up can be configured. For a complete list of the functions you can overload, refer to the **extensions** argument in the *Diamond object* documentation.

[1] "Overloading" is the process of creating a function with the same name as a function in the class you're inheriting from. In the example above, we have overloaded `administration()` and `blueprints()`.

Enable or Disable Functionality

Flask-Diamond functionality is initialized using extensions. In case you wish to disable certain functionality, you may simply omit that initialization step from the extensions list. For example, to enable REST functionality, omit email initialization from the extensions passed to create_app(). Many extensions can be enabled/disabled with ease, including rest, webassets, email, debugger, and task_queue.

Other extensions, like blueprints, are fundamental to the behavior of a Flask application, and are therefore trickier to disable in Flask-Diamond. In those cases, it may be better to override the initialization function using inheritance.

Inheritance also permits functionality to be disabled. For example, to completely disable email, it is possible to overload the email startup with a function that does nothing. It looks like this:

```
def init_email(self):
    pass # do nothing
```

Application start-up

Flask-Diamond applications initialize the following facets during startup:

1. `flask_diamond.facets.configuration`
2. `flask_diamond.facets.logs`
3. `flask_diamond.facets.database`
4. `flask_diamond.facets.accounts`
5. `flask_diamond.facets.blueprints`
6. `flask_diamond.facets.signals`
7. `flask_diamond.facets.forms`
8. `flask_diamond.facets.handlers`
9. `flask_diamond.facets.administration`
10. `flask_diamond.facets.rest`
11. `flask_diamond.facets.webassets`
12. `flask_diamond.facets.email`
13. `flask_diamond.facets.debugger`
14. `flask_diamond.facets.task_queue`

See *Application Facets* for an overview of the specific facets that ship with Flask-Diamond.

Extending the Scaffold

The scaffold files are a starting point, and you will probably end up creating many new files in the course of writing your application. You can think about the scaffold as being sortof similar to inheritance; if you want to change one of the default files, just overwrite it with your own. By customizing the scaffold, you can easily create new models, views, security views, administration views, API endpoints, and more.

Additional scaffolds are distributed along with Flask-Diamond. They are stored in `$VIRTUAL_ENV/share/skels` and can be applied manually using `mr.bob`. Additional scaffolds describe common patterns for using Views and Models.

It is recommended to stick with the directory structure in the beginning. As with anything, you are free to change the structure, but if you learn how to work within it, your applications will be easier to maintain and deploy - especially when you have dozens of Flask-Diamond applications to manage!

Further Reading

Several guides have been created to discuss Flask-Diamond application building in greater detail:

- *Facet: Database*
- *Facet: Administration*
- *Facet: Blueprints*

Philosophy of Flask-Diamond

Flask-Diamond provides a path that can guide your thought and development. Flask-Diamond is the road that leads to other ideas.

The following principles serve to guide the development of Flask-Diamond.

Inheritance

Flask-Diamond is primarily configured via Pythonic Inheritance. Your main application object will inherit from Flask-Diamond, and any functions you wish to customize must be overridden in order to change their behavior.

Economy

There are tons of libraries wrapped up in Flask-Diamond. If something has been done well by somebody else, then it is always preferable to leverage that work instead of re-building an unnecessary component.

Stable

Flask-Diamond versions are tied closely to specific versions of third-party libraries. When you stick with a specific version of Flask-Diamond, you can lock the version of third-party libraries too. The goal is to prevent any requirements from changing accidentally so that your application will be more resistant to code rot and API breakage.

Decomposable

Because third-party libraries are constantly changing, it is sometimes desirable to upgrade just one library without upgrading anything else. Flask-Diamond is built atop the Flask ecosystem, which is architecturally decomposable. Thus, extensions may be upgraded individually.

Data-centric

Flask-Diamond was originally built to support the research objectives of Ian Dennis Miller. Through his work on memes and social networks, Ian regularly needed to build lightweight API access to big data sets. Due to the precise nature of the SQL queries needed, a powerful standalone ORM like SQLAlchemy was preferred over the Django

approach of bundling the ORM with the web platform. Thus, Flask-Diamond is suitable for projects that require raw access to SQL queries.

Sensible

Flask-Diamond is pretty sensible about the defaults it presents. A lot of decisions have already been made in the service of delivering a functioning application out-of-the-box. The goal is to enable a focus upon the unique parts of your application, rather than the common tasks that most applications need anyway.

CHAPTER 4

Developers Guide

Learn common development patterns.

MVC: Model, View, Controller

Model-View-Controller (MVC) is a popular architecture for designing applications that have a user interface. At its heart, MVC is a collection of software design patterns that provide a vocabulary for designing your application. When you "speak MVC," other people who also know MVC will understand what you are saying.

The MVC vocabulary consists of:

- **Models**: a way for talking about data
- **Views**: a way for talking about user interfaces
- **Controllers**: a way for talking about program logic

This document presents an overview of Model-View-Controller and links to more detailed documentation that discusses these ideas in greater detail.

Model

A model is usually named after a noun. A model is a data representation of something that exists, and just about anything that exists can be modeled.

Entities and Relationships

To model a solar system, you'd start with a model of Planets and Satellites, which are the *entities* we will be dealing with. A planet can have many satellites, so there is a relationship between our entities. Using nothing more than the idea of "Planets" and "Satellites", you can go a long way towards modeling a solar system.

A complete data model consists of entities and the relationships between those entities.

- **Entities**: An Entity is a type of object. Entities have attributes, which are characteristics of the Entity.

In *Planets Tutorial*, a Planet is a an Entity and so is a Satellite. An attribute of a Planet is its "mass"; the mass of a planet is stored in the data model alongside the name of the planet. Since a planet can have a name, *name* is therefore also an attribute of a Planet. - **Relationships**: Entities can affect one another through relationships. In `tutorial-planet`, a Planet can have many Satellites. Since a Planet can have many Satellites, we call this a "one-to-many" Relationship. There are also one-to-one and many-to-many relationships.

A model can therefore be described using an Entity-Relationship Diagram, which shows all of the types of objects, their attributes, and the way entities relate to one another. Read more about *Facet: Database* for a more detailed discussion and code examples.

A Philosophy of Models

A model might be a very simple representation of a real thing, or the model might be very detailed. A model of an entire country's economy might require lots of detail, whereas a model of a school district might be relatively simpler.

A model is in some ways a platonic ideal of the actual domain being modeled. While things in the "real world" are irregular in an uncountable number of ways, our models are perfectly regular. Since models are stored in a database, all of the model attributes can be lined up nicely into rows and columns. Tidy!

Paradoxically, a model is always an imperfect representation of the thing it is modeling. The irregularities of the real world are difficult to capture using a model. The goal for good model creation is to isolate the parts of the model that are regular so as to reduce the number of exceptions to your model.

Sometimes, we talk about "domains" when we talk about models, because our models might be thematically related to one another. A domain might be something like *finance*, *gaming*, *email*, or any other broad category that people build applications for. To properly model a domain, we might talk to a "domain expert" to learn more about the kinds of models we are building.

View

A view is a user interface that can present data that comes from a model. In classic MVC, the model pushes data to the view, and the view knows how to update itself to display the data that was received from the model. Views can also contain input elements like buttons, fields, and sliders. When these input elements are activated, the Controller must decide how to respond. Views are often written as templates that have placeholders for data. For web programming, a View template is frequently written using HTML [1]. Read more about *Facet: Blueprints* for a more detailed discussion and code examples.

Controller

A controller responds to input by changing a view or model. A common type of controller is driven with a Graphical User Interface, which uses things like menus, fields, and buttons so that a human can click stuff to get things done. Read more about *Facet: Administration* for a more detailed discussion.

A different type of controller is an API, which is typically used by other software (rather than a human) to make the application do something. Read more about *Facet: REST* for a more detailed discussion.

CRUD: Create, Read, Update, Delete

Create Read Update Delete

CRUD stands for Create, Read, Update, and Delete. The CRUD pattern complements *Model-View-Controller* by providing a standard set of methods that can be applied to most types of models. CRUD acts like a simple API for models in Flask-Diamond. The four CRUD actions describe the life-cycle of a model object:

1. First, an object is **created** with certain attribute values.

[1] There's nothing inherently special about HTML templates for constructing Views. For example, Android Views can be constructed using Java. The point is that the idea of *views* can be generalized to any platform.

2. The object may be **read** to get the value of those object attributes.
3. The object values may be **updated** to update the model based on changes in the world.
4. Finally, the object is **deleted** to remove it from the database.

CRUDMixin

`flask_diamond.mixins.crud.CRUDMixin`, which has been adapted from Flask-Kit, will extend your model with functions for `create()`, `read()`, `update()`, and `delete()`. The default Flask-Diamond scaffold provides an example of CRUDMixin usage:

```
from flask.ext.diamond.utils.mixins import CRUDMixin
from flask.ext.diamond import db

class Planet(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
    mass = db.Column(db.Integer)
```

Notice that `class Planet()` inherits both `db.Model` and `CRUDMixin`. Now the following CRUD workflow is possible:

```
earth = Planet.create(name="earth", mass=200)
print "{name} has mass of {mass}".format(earth)
earth.update({'name': "Earth"})
print "Its name is {name}".format(myself)
earth.delete()
```

Flask-Admin CRUD

Flask-Admin provides Model CRUD functionality with its BaseModelView class. `BaseModelView` is an extremely powerful tool for rapidly implementing a web-based CRUD Graphical User Interface that makes it easy to create, read, update, and delete model objects using a web browser.

The following application instantiates a CRUD for the `Planet` model described above.

```
from flask_diamond import Diamond
from flask_diamond.administration import AdminModelView
from .models import User, Role, Planet

class my_diamond_app(Diamond):

    def init_administration(self):
        admin = self.super("administration", user=User, role=Role)
        admin.add_view(AdminModelView(
            Planet,
            db.session,
            name="Planet",
            category="Admin")
        )
```

Further Reading

- See *Facet: Database* for a more detailed examination of Models.

- See *Facet: Administration* for a more detailed examination of GUIs.

Database Schema Migration

Any time the data model is changed, the database must be updated so that it has the right columns. Any time a table or column is added, removed, or changed in any way, we say the database schema [2] has changed. We need to be sure the database schema always matches the model. This common task is called schema migration, and it is handled well by Alembic.

Flask-Diamond makes it quick and easy to rebuild the schema during development with `make db`, which will delete the development database and rebuild it with all new columns matching your models. However, it's a little drastic to always throw everything away and start from scratch. Furthermore, when the data in your database is important, it's actually impossible to throw it away and start over. Luckily, there's Flask-Migrate, which makes Alembic integration easy so we can handle migrations smoothly.

Introducing Migrations

Schema migrations, which are handled using Flask-Migrate, can be used to change in-production databases by recording the minimal set of steps will make your database schema match your model. Migrations are versioned, so it is possible to experiment with schemas before deploying them. It is also possible to roll back schemas in case something went wrong.

Creating a Migration

A database migration is potentially complex, so creating one takes several steps.

1. find the location of the dev database
2. create a new migration
3. rename the file with a short summary
4. edit the migration
5. test the migration
6. done

find the location of the dev database

We will work on the dev database, which should not have any important data in it.

```
grep SQLALCHEMY_DATABASE_URI etc/conf/dev.conf
```

create a new migration

This works by comparing python model to last migration version

```
make newmigration
```

[2] A database schema is a list of all the tables in a database, all the columns in those tables, and the data types for each column. Schemas are often expressed using SQL CREATE statements, which is a concise way of describing exactly which tables and columns need to exist.

This will create a new python script within the `migrations/versions` subdirectory of your application module directory. There will be two functions automatically created for you: `upgrade()` and `downgrade()`. The `upgrade()` function will individually add tables and columns to your old database schema until it matches your current model. The `downgrade()` function is the opposite. In this manner, it is possible to "roll back" to a previous schema.

rename the file with a short summary

```
cd ${MODULE}/migrations/versions
mv ${CHECKSUM}_.py ${CHECKSUM}_${SUMMARY}.py
```

edit the migration

- for sqlite, remove all drop_column() operations
- for sqlite, remove all create_foreign_key() operations

test the migration

- delete the database again (rm /tmp/daemon-dev.db)
- perform a new migration (make migrate)
- verify that it is empty (i.e. all tables are reflected)
- delete the empty migration file (rm ${MODULE}/migrations/versions/${CHECKSUM}_.py)

done

The migration is ready. It can now be applied in the production environment by specifying the configuration for the production system.

First, ensure you are using a configuration file that specifies the target database you will apply the migration to. Then, invoke the schema upgrade directly.

```
export SETTINGS=/etc/production.conf
manage.py db upgrade
```

An Example Migration File

It is easy to see how a migration uses SQLAlchemy directly to create tables if we examine an example. One of the migrations that ships with Flask-Diamond is in `flask_diamond/migrations/versions/20f04b9598da_flask-diamond-020.py`. In this case, the `upgrade()` function adds several new columns to the `user` table. The file looks like this:

```
"""update to flask-diamond 0.2.0

Revision ID: 20f04b9598da
Revises: cf0f5b45967
Create Date: 2015-02-07 22:54:24.608403

"""
```

```
# revision identifiers, used by Alembic.
revision = '20f04b9598da'
down_revision = 'cf0f5b45967'

from alembic import op
import sqlalchemy as sa


def upgrade():
    ### commands auto generated by Alembic - please adjust! ###
    op.add_column('user', sa.Column('current_login_at', sa.DateTime(), nullable=True))
    op.add_column('user', sa.Column('current_login_ip', sa.String(length=255), nullable=True))
    op.add_column('user', sa.Column('last_login_at', sa.DateTime(), nullable=True))
    op.add_column('user', sa.Column('last_login_ip', sa.String(length=255), nullable=True))
    op.add_column('user', sa.Column('login_count', sa.Integer(), nullable=True))
    ### end Alembic commands ###


def downgrade():
    ### commands auto generated by Alembic - please adjust! ###
    op.drop_column('user', 'login_count')
    op.drop_column('user', 'last_login_ip')
    op.drop_column('user', 'last_login_at')
    op.drop_column('user', 'current_login_ip')
    op.drop_column('user', 'current_login_at')
    ### end Alembic commands ###
```

Applying a Migration

To apply a migration to the development database, enter the virtualenv and run:

```
make migrate
```

This will inspect your database and automatically apply migrations, in order, until it is at the latest. By default, this applies the migration to your development database.

Migrations in Production

In order to affect the production database, you must set `SETTINGS` so that it points to your production configuration. Then, you must invoke Flask-Migrate explicitly, like so:

```
bin/manage.py db upgrade
```

Displaying a Migration as SQL

It can be helpful to inspect a migration before it is applied to the database. The following command will display a preview of the changes that will be made once a migration is applied:

```
bin/manage.py db upgrade --sql
```

Accessing Flask-Migrate directly

In fact, the full functionality of Flask-Migrate is easily available on the command line:

```
bin/manage.py db help
```

Version Control and Migrations

Because each migration has a unique checksum, and because each migration is in a separate file, it is easy to use a version control mechanism like `git` to closely control your schemas.

Documentation

As projects grow in size, the need to create documentation increases. The Sphinx Project has put forth a very robust Python documentation solution. As a result, Flask-Diamond provides a starter template for creating your own documentation using Sphinx.

Where are the files

You will find a starter set of documentation in the `/docs` folder of your project. The first file you need to edit is called `/docs/index.rst`, and from there build the table of contents to describe your project. Images and media may be placed in `/docs/_static`. These source files will be transformed into a documentation website that can be browsed online.

Sphinx

Sphinx is a structured document generator application. As a project author, you create documents that describe your project, and Sphinx will take care of the Table of Contents, links between documents, and all of the other parts that make it easy for somebody to navigate the documentation. Sphinx is particularly well suited to creating documentation websites, in which case Sphinx will automatically apply a theme to every page. The end result of using Sphinx is that your project will have an extremely functional online documentation website.

Restructured Text

Documentation in Sphinx is written using a markup language called ReStructured Text (quickref). While this language has some similarities to Markdown, it has many extensions that are suited to document structuring. As a result, Restructured Text makes it easy enough to manage links between pages, make references to specific Python classes, and add footnotes.

In my experience, the learning curve on Restructured Text is steeper than I expected. As a result, you need to persevere, but the payoff is totally worth it. The best place to start is with the ReStructured Text documentation from the Sphinx website.

autodoc

Sphinx can use the autodoc extension to look at your project's file structure and determine all of the classes in your project. This can be extremely useful for generating API documentation. Sphinx can also extract method and object signatures, which makes it easy to describe all of the parameters for everything.

docstrings

Sphinx autodoc is also able to scan your source code for comments that have been formatted for Sphinx using Restructured Text. Here is an example taken from the `flask_diamond.facets.accounts.init_accounts()` Flask-Diamond source code:

```
def init_accounts(self, app_models=None):
    """
    Initialize Security for application.

    :param kwargs: parameters that will be passed through to Flask-Security
    :type kwargs: dict
    :returns: None

    >>> def ext_security(self):
    >>>     super(MyApp, self).ext_security(confirm_register_form=CaptchaRegisterForm)
    """
```

The comment above will be used to create documentation by Sphinx. Special directives like `:param:` are used to specify details about the documentation that *autodoc* cannot determine on its own.

Building the documentation

In the project on the command line:

```
make docs
```

Output

The results of `make docs` will end up in `var/sphinx/build`. You can open that folder in your web browser to view the documentation.

ReadTheDocs

It is easy to integrate Flask-Diamond applications with readthedocs.org due to the use of Sphinx. A special file called `.readthedocs.txt` is included with the project scaffold. This file contains a reduced set of Python requirements that may make it easier for RTD to build your project.

Testing

Writing tests is an important part of building an application. Especially with dynamically typed languages such as Python, testing is one of the only ways to determine the correctness of your implementation. Python has built the concept of Unit Tests into the core library. *Unit Testing* is a technique for validating units of your program by making assertions about the behaviour of that code. If any assertions turn out to be false, then you know your code isn't correct.

Nose

In the interest of making test as automatic as possible, Flask-Diamond uses Nose for *test discovery*, which will find and run all your tests without having to create a test harness. As an application becomes more developed, it may become necessary to build a test harness for the application. However, at the start of application development, it is usually easier to use test discovery in order to iterate faster.

Nose will automatically search through the `/tests` folder for any files starting with the name "test" that have functions in them that also start with "test".

Running Tests

It is possible to invoke the testing subsystem from the command line. The following different testing methods are available in Flask-Diamond:

Run every test

Find every test and run it.

```
make test
```

Run individual tests

Run tests identified with decorator `@attr("single")`. These "single" tests can be used to make testing much faster by skipping all of the other tests.

```
make single
```

The following code snippet contains a simple test with the *single* attribute decorator applied to it.

```
from nose.plugins.attrib import attr
from flask.ext.testing import TestCase
from flask.ext.diamond.mixins.testing import DiamondTestCaseMixin

class BasicTestCase(DiamondTestCaseMixin, TestCase):
    @attr("single")
    def test_basic(self):
        assert True
```

Automatically run individual tests

This functionality will watch the project folder for files to change. When a file has changed, it will re-run tests identified with `@attr("single")`. This feature is designed to make it very quick to get feedback on the performance of your code.

```
make watch
```

xUnit

It is possible to automatically interpret the results of testing using the xUnit framework. Nose xUnit output can permit tools like Jenkins CI, Travis CI, or Atlassian Bamboo to capture the results of testing.

Libraries Included in Flask-Diamond

Flask-Diamond pulls from many different Flask extensions, which are all listed in this document. A significant factor for inclusion with Flask-Diamond is the existence of good documentation. Most of the following libraries therefore provide extensive documentation that can help you to understand everything in greater detail.

Database/Model

- Flask-SQLAlchemy: "Flask-SQLAlchemy is an extension for Flask that adds support for SQLAlchemy to your application. It requires SQLAlchemy 0.6 or higher. It aims to simplify using SQLAlchemy with Flask by providing useful defaults and extra helpers that make it easier to accomplish common tasks."
- Flask-Migrate: "Flask-Migrate is an extension that handles SQLAlchemy database migrations for Flask applications using Alembic. The database operations are provided as command line arguments for Flask-Script."
- Flask-Marshmallow: "Flask-Marshmallow is a thin integration layer for Flask (a Python web framework) and marshmallow (an object serialization/deserialization library) that adds additional features to marshmallow, including URL and Hyperlinks fields for HATEOAS-ready APIs. It also (optionally) integrates with Flask-SQLAlchemy."

Development

- Flask-Testing: "The Flask-Testing extension provides unit testing utilities for Flask."
- Flask-DebugToolbar: "This extension adds a toolbar overlay to Flask applications containing useful information for debugging."
- Flask-DbShell: "The extension provides facilites for implementing Django-like ./manage.py dbshell command"
- Flask-Script: "The Flask-Script extension provides support for writing external scripts in Flask. This includes running a development server, a customised Python shell, scripts to set up your database, cronjobs, and other command-line tasks that belong outside the web application itself."

Security

- Flask-Security: "Flask-Security allows you to quickly add common security mechanisms to your Flask application."
- Flask-Login: "Flask-Login provides user session management for Flask. It handles the common tasks of logging in, logging out, and remembering your users' sessions over extended periods of time."

Doing Stuff

- Flask-Celery-Helper: "Celery support for Flask without breaking PyCharm inspections."
- Flask-Mail: "The Flask-Mail extension provides a simple interface to set up SMTP with your Flask application and to send messages from your views and scripts."

Interfaces

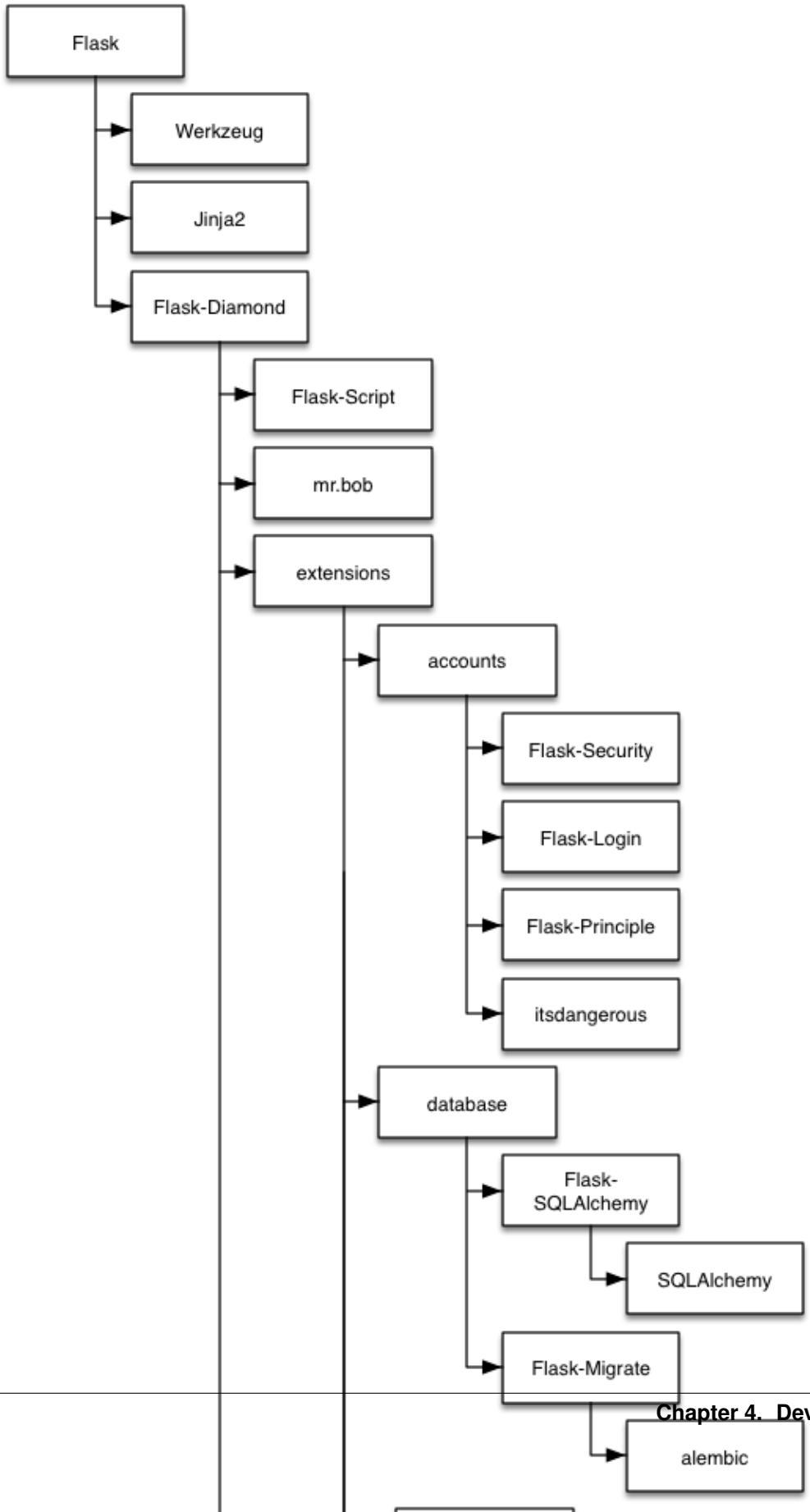
- Flask-Admin: "In a world of micro-services and APIs, Flask-Admin solves the boring problem of building an admin interface on top of an existing data model. With little effort, it lets you manage your web service's data through a user-friendly interface."
- Flask-RESTful: "Flask-RESTful is an extension for Flask that adds support for quickly building REST APIs. It is a lightweight abstraction that works with your existing ORM/libraries. Flask-RESTful encourages best practices with minimal setup. If you are familiar with Flask, Flask-RESTful should be easy to pick up."
- Flask-WTF: "Flask-WTF offers simple integration with WTForms."

Presentation

- Flask-Assets: "Flask-Assets helps you to integrate webassets into your Flask application."
- Flask-Markdown: "Flask-Markdown adds support for Markdown to your Flask application. There is little to no documentation for it, but it works just the same as markdown would normally."

Diagram of Libraries

Flask-Diamond is composed of many other libraries. The following diagram is an effort to describe how these modules are organized. The `SVG diagram of Flask-Diamond libraries` is also available for download.


Flask
Werkzeug
Jinja2
Flask-Diamond
Flask-Script
mr.bob
extensions
accounts
Flask-Security
Flask-Login
Flask-Principle
itsdangerous
database
Flask-
SQLAlchemy
SQLAlchemy
Flask-Migrate
alembic

CHAPTER 5

Facets Guide

This section describes the facets of Flask-Diamond.

Application Facets

A Flask-Diamond application consists of many components (called *facets*) that complement each other to form a coherent application. These facets are common to many applications, but they can be enabled and disabled individually depending on specific requirements. A Flask-Diamond application is customized by changing the way these facets behave.

List of Facets

Here are all of the facets currently shipping with Flask-Diamond:

1. `flask_diamond.facets.configuration`. the `$SETTINGS` environment variable is inspected and the file it points to is loaded.
2. `flask_diamond.facets.logs`. based on the configuration, write log messages to a file on the filesystem.
3. `flask_diamond.facets.database`. connect to a database and initialize the SQLAlchemy Object Relational Mapper (ORM)
4. `flask_diamond.facets.accounts`. manage users, roles, login, passwords, and other security things with Flask-Security.
5. `flask_diamond.facets.blueprints`. initialize your application's views (in the MVC sense), which are saved as "blueprints" in a Flask application.
6. `flask_diamond.facets.signals`. Flask provides a signals subsystem that your application can hook into to automate certain behaviors.
7. `flask_diamond.facets.forms`. initialize your application's form helpers, which may be global to the forms used in your application.
8. `flask_diamond.facets.handlers`. when something goes wrong, you may want to handle it (e.g. by displaying a 404 page). This is the place to create redirections or other custom request handlers that extend beyond views.
9. `flask_diamond.facets.administration`. a quick GUI using Flask-Admin with extensive Model support.
10. `flask_diamond.facets.rest`. provide a REST API using Flask-RESTful

11. `flask_diamond.facets.webassets`. it is possible to bundle assets like images, CSS, and javascript with your application. webassets simplifies some of this work.
12. `flask_diamond.facets.email`. send email with Flask-Mail
13. `flask_diamond.facets.debugger`. when the configuration specifies that `DEBUG = True`, the web interface will display a widget with extra debugging tools.
14. `flask_diamond.facets.task_queue`. provide a task queue using Celery

Next Steps

Now that you know what facets are, learn how to use them by *Writing an Application with Flask-Diamond*.

Facet: Accounts

User accounts are a common requirement for many applications. Another common requirement is *Roles*, which can be used to grant certain users access to specific functions. Both Users and Roles are provided in Flask-Diamond by Flask-Security, which provides a nice interface for controlling these models.

User and Role Models

Flask-Diamond is already set up with `flask_diamond.models.user` and `flask_diamond.models.role`, which provide everything needed for a simple setup. You will notice in your application that `models/__init__.py` then imports these classes from Flask-Diamond. In this manner, your application will incorporate these models into its own schema.

Overloading the User Model

With an application of moderate complexity, it may be necessary to store additional user data, and certain data may just be simple to put in the User model directly. The following code snippet describes an `init_security()` function that can be used to implement your own User model.

```
def init_security(self):
    self.super(app_models=models)
```

With that in place, ensure `models__init__.py` imports your own User model instead of importing from Flask-Diamond.

Flask-Admin Integration

You can use user accounts in lots of ways, but one common pattern for Users is to create a password-protected user interface. To make this easier, the Flask-Diamond scaffold includes templates in `views/administration/templates/security` that permit you to customize the look of login, password maintenance, account registration, and password reset. The templates already inherit from Flask-Admin, but by editing the templates in your project, you can customize them to other scenarios too.

Facet: Administration

A common pattern in application design is to apply *CRUD* to your *Model*, and then provide a Graphical User Interface for people to interact with the Model. Flask-Admin makes it very easy to create a basic interface with Create-Read-Update-Delete functionality, and provides a framework for designing much more sophisticated interfaces.

This document discusses a simple CRUD with Flask-Admin, and then extends the CRUD with additional functionality.

A Simple CRUD GUI

When Flask-Admin creates a GUI, it automatically discovers the Model Attributes and creates a web form containing fields for all the attributes. Flask-Admin provides the BaseModelView class as the foundation for building Views that a user can interact with to inspect and control a Model. Flask-Diamond provides AdminModelView, which applies an authentication requirement so that only users with the Admin role can access the View.

AdminModelViewExample

The following example imports a class called Planet (which is described in the *Model documentation*). Then, the Planet class is added to the GUI using the add_view() method.

```
from flask_diamond import Diamond
from flask_diamond.administration import AdminModelView
from .models import User, Role, Planet

class my_diamond_app(Diamond):

    def init_administration(self):
        admin = self.super("administration", user=User, role=Role)
        admin.add_view(AdminModelView(
            Planet,
            db.session,
            name="Planet",
            category="Admin")
        )
```

When the server is launched, it provides a GUI facility for applying CRUD operations to the Planet model. Due to the use of AdminModelView, the application will now require a password before allowing anybody to create, read, update, or delete any Planet object.

The *category* parameter causes the GUI to put this View into the menu bar beneath the heading "ModelAdmin". The *name* parameter indicates this link will be titled *Planet*. Now you can find the Planet CRUD by navigating through the menu to Models and then to Planet.

Multiple CRUD Views Inside BaseModelView

BaseModelView actually creates multiple views that provide CRUD functionality:

- **Create**: this view presents a form with all of the model attributes displayed as blank fields. When the fields are populated and the form is submitted, a new model object will be created.
- **List**: the List view displays a paginated list of all the objects of the Model. This view also contains widgets for editing and deleting objects.
- **Edit**: the Edit view is identical to the create view, except now the fields are populated by a specific model object. When the fields are changed, the model object can be updated.

- **Delete**: The Delete view simply confirms whether the user wants to delete an object.

Create-List-Edit-Delete corresponds directly to Create-Read-Update-Delete.

Extending the CRUD

Flask-Admin makes it pretty easy to add custom functionality through Python class inheritance [1]. In the *Model documentation*, the Planet Model provides a `bombard()` method that sends an asteroid at a planet. The following sections demonstrate how to expose the `bombard()` method and create a user interface widget for calling that method.

Exposing a new View

In addition to the basic CRUD views, new views can be created for doing other things with Models. Since the Planet class has been extended with a bombard() method, let's create a URL endpoint to send an asteroid at a planet.

```
from flask_diamond import Diamond
from flask_diamond.administration import AdminModelView
from flask_admin import expose
from .models import User, Role, Planet

class PlanetModelView(AdminModelView):
    @expose('/bombard/<planet_id>')
    def bombard(self, planet_id):
        the_planet = models.Planet.get(planet_id)
        the_planet.bombard(mass=10.0)
        return flask.redirect(flask.url_for('.list_view'))

class my_diamond_app(Diamond):
    def init_administration(self):
        admin = self.super("administration", user=User, role=Role)
        admin.add_view(PlanetModelView(
            models.Planet,
            db.session,
            name="Planet",
            category="Admin")
        )
```

Adding a Widget

One simple way to add functionality to the user interface is to use Flask-Admin's formatters to make a field into an interactive widget. This basic pattern is demonstrated by formatting Planet.mass with a "bombard" button:

```
import jinja2
from flask_diamond import Diamond
from flask_diamond.administration import AdminModelView
from flask_admin import expose
from .models import User, Role, Planet

class PlanetModelView(AdminModelView):
    def mass_formatter(self, context, model, name):
        mass_widget_template = "{0} <a href='{1}'>bombard!</a>"
        mass_widget = mass_widget_template.format(
            model.age,
```

[1] Incidentally, Python class inheritance is the same mechanism used by Flask-Diamond for customization. Inheritance is discussed further in *writing_an_application_with_flask-diamond*.

```
            flask.url_for(".bombard", planet_id=model.id)
        )
        return jinja2.Markup(mass_widget)

    column_formatters = {
        "age": mass_formatter,
    }
```

When these two *PlanetModelView* examples are combined, the result is a user interface that can bombard a planet with asteroids when clicked.

ModelView Example

The following `AuthModelView` includes examples for overriding various fields within the model view. The full documentation for ModelView should be consulted for more information, but this example is intended to describe how that information may be applied within a Flask-Diamond project.

```
class PlanetAdmin(AuthModelView):

    edit_template = 'planet_edit.html'

    column_list = ("name", "mass")

    form_overrides = {
        "upload_buffer": FileUploadField
    }

    form_args = {
        'upload_buffer': {
            'label': 'Planet PDF',
            'base_path': "/tmp",
        }
    }
```

More Flask-Admin

Flask-Admin is really powerful, and the best way to learn more is by reading the Flask-Admin documentation.

Further Reading

- See *CRUD: Create, Read, Update, Delete*, which describes the Create-Read-Update-Delete pattern for Models.
- See *Facet: Database* for a more detailed examination of Models.

Facet: Blueprints

As is explained in *MVC: Model, View, Controller*, a *View* takes data from a Model and presents it (typically to a user). Often times, there are multiple Views of a data Model, and they may present different aspects of the Model. A common pattern for multiple views is the use of permissions to restrict functionality to users based upon their account role. The administrator may have a special *View* into the data that provides extra functionality that regular users do not have.

Views are frequently written using *templates*, which have placeholders for variables that may be filled in by the application. In Flask-Diamond, the Jinja templating language is used to generate HTML, javascript, CSS, and other

web-facing files that create a user interface. By populating the template with data from the Model, the result is that users can interact with Model data.

Jinja is a Templating Language

The rest of this document provides a summary of the key points about Jinja Views. The Jinja website provides a nice example of what Jinja looks like:

```
{% extends "layout.html" %}
{% block body %}
  <ul>
  {% for user in users %}
    <li><a href="{{ user.url }}">{{ user.username }}</a></li>
  {% endfor %}
  </ul>
{% endblock %}
```

A detailed discussion of templates is available from the Jinja templates documentation.

Variables in Jinja

A python expression within a Jinja template is denoted using double-curlies as `{{ }}`. In this manner, it is easy to reference variables by simply placing them inside double-curlies. In the example above, *user.username* will be replaced by the actual value of a user's username (e.g. "administrator"). If it were not inside the double-curlies, "user.username" would be printed verbatim (i.e. substitution is not performed unless inside double-curlies).

Statements in Jinja

A python statement (like an "if statement") is denoted in Jinja using `{% %}`. Using statements, it is possible to create dynamic templates that print different details based upon the data given to them. To use *Planets Tutorial* as an example, we could extend the template to print a special message when Earth is viewed.

```
{% if planet.name=='earth' %}
    Go Earthlings!
{% endif %}
```

Rendering a template with Flask

Flask provides an easy mechanism for working with templates that is called render_template(). When a template is rendered with data, the placeholders inside the template are replaced with the data. In the case of web applications, the result is typically an HTML document that presents the application's data model in a format that a human could use through a web browser.

To render the previous template, first save it to a file called `my_template.html` and place that in a directory called `templates`.

The code looks like:

```
my_data = {"users": [{"username": "administrator", "url", "http://example.com"}]}
return flask.render_template("my_template.html", **my_data)
```

Flask searches for templates by looking through a directory called "templates". This behaviour can be extended by using Blueprints, which are explained a little later in this document. More information about Flask and templates is available from the Flask templating documentation.

Routing a View

URLs are used within web applications to identify Views. In *Planets Tutorial*, the URL */planet_list* could return with a summary of all the planets in the database, and a URL like */planet/earth* could provide information about the named planet.

When building an *MVC* View for a web application, the Controller is responsible for actually routing the user to the View. MVC Web applications use the URL to connect with a view, such that a user can use their web browser to request */user/login_form* in order to log in to a website or view the */planet/earth* to view details about a planet called Earth. In Flask, the route() decorator is used to apply a route to a View function.

A simple route looks like:

```
@app.route('/')
def index():
    return "Hello world!"
```

The Site Map is used to determine all the routes that will be necessary for exposing a web application. Every View must be present within the Site Map, and there must be a unique URL for each View.

It is possible to look at a Flask-Diamond web application site map from within the application itself:

```
print flask.current_app.url_map
```

Flask Blueprints: a Collection of Views

Often times, there will be several related views and it makes sense to collect these together using a Flask Blueprint. When this happens, the views are collected into a URL subdirectory, and individual views are nested within that URL. In *Planets Tutorial*, there could be several views related to tracking moons (satellites), which could be collected into the */satellite* directory.

The Flask documentation demonstrates an extremely simple blueprint:

```
from flask import Blueprint, render_template, abort
from jinja2 import TemplateNotFound

simple_page = Blueprint('simple_page', __name__,
                        template_folder='templates')

@simple_page.route('/', defaults={'page': 'index'})
@simple_page.route('/<page>')
def show(page):
    try:
        return render_template('pages/%s.html' % page)
    except TemplateNotFound:
        abort(404)
```

This example demonstrates everything we have discussed so far:

- the `render_template` function
- the `template_folder` that contains Jinja templates
- routing the View to a URL with `route()`

The Flask documentation also explains how to register a blueprint with an application:

```
from flask import Flask
from planets.simple_page import simple_page
```

```
app = Flask(__name__)
app.register_blueprint(simple_page)
```

Views within Flask-Admin BaseModelView

In *Flask-Admin*, each BaseModelView is actually a Blueprint that provides views for *creating, reading, updating, and deleting* model objects. The BaseModelView template behaves much like a regular blueprint, except:

- expose() is used to "expose" a view inside BaseModelView instead of route()
- self.render() is used instead of render_template

In this manner, it becomes easy to extend a CRUD with custom methods that go beyond create, read, update, and delete.

Facet: Database

In Flask-Diamond, a Model is a way of reading and writing a database. If our application is a model of the solar system, then we're modeling planet objects in a database. A fundamental assumption of the *Model-View-Controller* architecture is that our application deals with objects, and our objects are modeled after the things our application deals with.

This document will demonstrate a model, then discuss some of the ways Flask-Diamond makes it easier to work with models.

A Basic Model

A model is actually written in Flask-Diamond using Python. Models are represented using SQLAlchemy, which is a very powerful Python library for working with databases. Since we are storing our models in a database, SQLAlchemy provides a strong foundation for getting the job done.

Let's create a model of a planet with a name and mass. Let us also create a model of a satellite that orbits a planet. The following example demonstrates one way this model might be accomplished. [2]

```
from flask_diamond import db
from flask_diamond.mixins.crud import CRUDMixin

class Planet(db.Model, CRUDMixin):
    "A Planet is a celestial body"
    id = db.Column(db.Integer(), primary_key=True)
    name = db.Column(db.String(80), unique=True)
    mass = db.Column(db.Float())
```

Now, we can model a Planet and a Satellite. Notice that satellites specify a relationship to a planet. Let's use our data model to create a few objects in our database.

```
from planets import models
earth = models.Planet.create(name="Earth", mass=100.0)
mars = models.Planet.create(name="Mars", mass=90.0)
```

[2] Note the use of CRUDMixin, which provides us with a create() method. For more information about CRUDMixin, see *CRUD: Create, Read, Update, Delete*.

Model Methods

Now let's assume our planets can be bombarded by asteroids with a certain mass. We can extend our planet model to incorporate this feature

```
class Planet(db.Model, CRUDMixin):
    "A Planet is a celestial body"
    id = db.Column(db.Integer(), primary_key=True)
    name = db.Column(db.String(80), unique=True)
    mass = db.Column(db.Float())

    def bombard(self, mass=1.0):
        self.mass += mass
        self.save()
```

Now we can send an asteroid at a planet and our data model will add the mass of the asteroid to the planet:

```
from planets import models
earth = models.Planet.create(name="Earth", mass=100.0)
earth.bombard(mass=15)
print(earth.mass)
```

Only The Model Controls the Data

It is important to contain any data-altering methods within your Model, or else there may be confusion down the road. For example, if there is code within the Controller that directly alters the data Model, you should move that code from the Controller to a new Model method that achieves the same task. Finally, in the controller, invoke the Model method.

SQLAlchemy

SQLAlchemy and Flask-SQLAlchemy are used to provide an Object Relation Mapper (ORM), which reads/writes a database and makes the data easy to access using Python. By using an ORM such as SQLAlchemy, it is possible to avoid many pitfalls of directly using SQL, such as SQL injections or schema mismatches. One of the best resources to learn about writing models is the SQLAlchemy documentation itself, which is both excellent and extensive.

Model Relationships with SQLAlchemy

The SQLAlchemy Basic Relationships document provides an excellent overview of different relationship patterns, including:

- One to Many
- Many to One
- One to One
- Many to Many
- Many to Many Association

To demonstrate a basic relationship, let's say a planet can have a satellite orbiting it:

```
from flask_diamond import db
from flask_diamond.mixins.crud import CRUDMixin

class Planet(db.Model, CRUDMixin):
    "A Planet is a celestial body"
```

```
    id = db.Column(db.Integer(), primary_key=True)
    name = db.Column(db.String(80), unique=True)
    mass = db.Column(db.Float())

class Satellite(db.Model, CRUDMixin):
    "A Satellite orbits a Planet"
    id = db.Column(db.Integer(), primary_key=True)
    name = db.Column(db.String(80), unique=True)
    mass = db.Column(db.Float())
    planet = db.relationship('Planet', backref=db.backref('satellites', lazy='dynamic'))
    planet_id = db.Column(db.Integer(), db.ForeignKey("planet.id"))
```

The following code example uses the classes above to create a planet called Earth with a moon.

```
from planets import models
earth = models.Planet.create(name="Earth", mass=100.0)
moon = models.Satellite.create(name="Moon", mass=25.0, planet=earth)
```

Querying with SQLAlchemy

Based on the Planet class, a simple query that finds a planet named "Earth" looks like:

```
earth = models.Planet.find(name="Earth")
print(earth.mass)
```

However, the SQLAlchemy Query API is extremely powerful, and its documentation is the authoritative source.

When the Model Changes

There is a close correspondence between the Model and the database tables. If an attribute is added to a model, then we need a new column in our database to store the values for this attribute. If the model changes, the database must also change. There are two ways of updating your database:

- **the clean slate**: delete the old database and creating a new one that reflects the latest changes to the model. This is accomplished with `make db` on the command line. It's easy and quick.
- **schema migrations**: analyze your updated model to determine what parts are different from your old database, and then add/remove those parts to a live database. This is tricky, but it is necessary for databases in production. Read more in *Database Schema Migration*.

As long as you are actively developing, it is recommended to use `make db` each time you update your model. However, when your application is live, you will need to read *Database Schema Migration* to learn about altering a production database.

Another model example

For the sake of illustration, the following is a recursive model that is able to link to itself, creating a friendship graph of individuals.

```
from flask_diamond import db
from flask_diamond.mixins.crud import CRUDMixin

class Individual(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(255))
```

```
    friend_id = db.Column(db.Integer, db.ForeignKey('individual.id'))
    friend = db.relationship('Individual',
        primaryjoin=('Individual.friend_id == Individual.id'),
        remote_side="Individual.id")

    def set_friend(self, obj):
        self.friend = obj
        self.save()

    def __str__(self):
        return self.name
```

Further Reading

- See *Database Schema Migration*, which describes how to evolve the application database along with its Model.
- See *CRUD: Create, Read, Update, Delete*, which describes the Create-Read-Update-Delete pattern for Models.
- See *Facet: Administration*, which explains how to create a GUI for interacting with Models.

Facet: Debugger

To simplify debugging, Flask-Diamond provides a basic setup for logging so that your application can write trace data to a file for your inspection. Flask-Diamond also integrates Flask-DebugToolbar to provide a powerful debugger shell directly in the interface.

Logs

Flask-Diamond creates a logging object during initialization. You can write log messages to this object with the following basic code:

```
flask.current_app.logger.debug("this is a debug-level message")

flask.current_app.logger.warn("important! this is a warning message")
```

The destination for log files is controlled in the configuration file. During development, `dev.conf` will write log messages to the var/logs path of your project.

Log Level

During debugging, it is sometimes useful to get extra information about your application while it is running. The *log level* is used to control how critical logging is. When the level is `DEBUG`, then all messages are printed - this can be verbose. When the level is `INFO`, then debugging messages are not printed at all, but general information messages are. When the level is `WARN`, then only important messages are printed. This level may be suitable for production.

DebugToolbar

Flask-DebugToolbar is a useful web gadget that can help developers to diagnose problems within the browser. In practice, you may end up doing most of your development in a terminal, but the Debug Toolbar can nevertheless be handy for casual projects.

In the application configuration file, whenever `DEBUG=True` the DebugToolbar will be active. By default, the development configuration enables debugtoolbar whereas the production configuration disables it. It is possible to completely remove the DebugToolbar by ensuring the `debugger` extension is not loaded at all during `flask_diamond.Diamond.init_app()`.

Facet: Email

Lots of applications need to send emails in the course of their operation. Email remains one of the best sources of *identity* on the Internet today. Flask-Mail is an easy way to integrate email-sending capabilities into your application.

Configuration

By default, `flask_diamond.facets.email` is already enabled in `flask_diamond.Diamond.init_app()`, so there is nothing special to do. The project created by `flask-diamond` has email commented out by default, so you will have to enable this in your project's `__init__.py` file.

The settings for the SMTP server are located in your configuration file. The relevant lines are these:

```
MAIL_SERVER = '127.0.0.1'
MAIL_PORT = 25
MAIL_USE_TLS = False
MAIL_USERNAME = None
MAIL_PASSWORD = None
```

You will need to set this configuration based on your particular email setup.

SMTP Server

In order for your application to be able to send emails, you will need access to an SMTP server. It is possible to use an email provider such as Gmail, but be aware that you won't be able to route a very high volume of email through a service like that. Often times, you can use a web hosting provider for higher volumes. Since every situation is different, you will need to track down your mail server configuration on your own.

Facet: REST

One of the most common ways to build an API is using a paradigm called REST. Flask-RESTful is a great library for Flask that simplifies the creation of a REST API. This document describes REST and how to implement it in a Flask-Diamond project.

REST in a Nutshell

Representational State Transfer, usually abbreviated REST, can be simplified as follows:

- The web consists of documents that are referred to by URL links.
- Each URL is a "thing". As a part of speech, it is a noun. For example: a URL points to a picture, a document, or a person's timeline.
- The language spoken by web browsers and web servers is called HTTP. This is the protocol for "hypertext".
- HTTP provides actions (verbs) like GET, PUT, DELETE, and POST.

- Using HTTP and URLs provides verbs and nouns that we can use to write statements, like "GET a Picture" or "DELETE a Document".

REST is a strong statement about the grammar of the web. It tells us where the nouns are and where the verbs are. If we learn to make APIs that are *RESTful*, then we can count on others to be able to use our APIs in natural ways to create complex expressions.

Examples of REST

Let's say you are building an API for a solar system and you need to manage planets for observation. Our API could then expose one URL endpoint for retrieving all planets: `/api/planet_list`. We could then add a planet to our solar system with an HTTP POST containing the following data:

```
{
    'name': 'Venus',
    'mass': '90.0'
}
```

Such a POST could be issued using javascript, from the command line (e.g. with `curl`), or using any HTTP client. That's the beauty of REST: if you work with the web, the web can work with you.

Flask-RESTful

It is easy to create RESTful APIs with Flask-RESTful, which is already integrated into Flask-Diamond.

The primary way to use Flask-RESTful is to create Resource objects. A Resource is a "thing" that your API will expose. You can easily add API Resources to your Flask-Diamond application by placing them into a function called *init_rest()*:

```
from flask_diamond import db
from .mixins import DiamondTestCase
from ..models import Planet

def init_rest():
    class PlanetResource(Resource):
        def get(self, name):
            return models.planet.find(name=name).dumps()

    rest.add_resource(PlanetResource, '/api/planet/<str:name>')
```

This simple example creates a resource called PlanetResource. Then, it specifies a way to handle the GET verb, which will result in retrieving a planet. Finally, the resource is exposed using a URL: `/api/planet/...`.

MarshmallowMixin

`flask_diamond.mixins.marshmallow.MarshmallowMixin` simplifies object marshalling, which is the process of mapping data to and from a serialization format like JSON. This functionality is provided by Marshmallow and Flask-Marshmallow.

Marshalling is useful because applications must frequently send model data across the Internet, and in order to do so, models are commonly translated into JSON or another format. For example, date and time objects are native Python objects that must be converted to a string in order for JSON to transmit them. Marshalling makes serialization and deserialization into a repeatable process.

Let's look at the following line of code:

```
return models.planet.get(id).dumps()
```

The *dumps()* at the end of the line will cause an ORM model object to be converted to JSON using Marshmallow. For more information, please see the documentation for Marshmallow and Flask-Marshmallow.

How not to REST

REST says we should not create URLs that imply actions; URLs must be things. This pattern is common in older websites. For example, you should not build an API with a URL called `/api/rename_planet` because that describes an action: changing a planet's name. REST says the only actions available are HTTP verbs, like GET and PUT.

You can think about it this way too: when a web client visits a URL, it usually issues the GET verb. It doesn't really make sense to `GET /api/rename_planet` because that's actually something you want to push to the server, not get from the server. However, early web developers tried to make this work using arguments, which resulted in operations like `GET /api/move_piece?from=A2&to=C3`. These became really long URLs that contained all the same information as the RESTful example, but which broke certain features of the Internet. For example, if a web spider visited that URL, the web server would incorrectly interpret that visit as a command to move a piece. This leads to chaos. Don't make URLs this way. Be RESTful, instead.

CHAPTER 6

Operation Guide

The User Guide topics are for managing and installing a Flask-Diamond application. In particular, IT Ops and Deployment Engineers will benefit from this section.

Requirements Management with `virtualenv`

The preferred way to manage your Python requirements is with virtualenv. The preferred way to manage virtualenvs is with virtualenvwrapper, which provides a regularized interface for creating and using virtualenvs.

When you install your Flask-Diamond application inside a virtualenv, you are able to "freeze" the state of your installed Python libraries. This way, your application will never suffer from broken requirements due to a system-wide upgrade. This document describes the typical workflow for using virtualenv to manage a Flask-Diamond project.

Pre-requisites

You need to install the minimum *System Requirements* in order to have the necessary tools for this process. It is assumed that you now have `workon` and `mkvirtualenv` available. If you cannot call these commands in a terminal, then you should double-check your *System Requirements*.

Making a New `virtualenv`

The way to initialize a new virtualenv is with `mkvirtualenv`, which is described on the mkvirtualenv documentation. The following example demonstrates creating a virtualenv called *my-diamond-app*.

```
$ mkvirtualenv my-diamond-app
New python executable in my-diamond-app/bin/python2.7
Also creating executable in my-diamond-app/bin/python
Installing setuptools, pip...done.
(my-diamond-app) $
```

Upon creating the virtualenv, it will be activated and your shell prompt changes to include your project name as a prefix. You can verify that you are inside the virtualenv by echoing an environment variable to the screen:

```
(my-diamond-app) $ echo $VIRTUAL_ENV
~/.virtualenvs/my-diamond-app
```

Using it

When you need to work on your project, you must activate your project's virtualenv using `workon`. After calling `workon`, your shell's search path will be updated so that scripts from your virtualenv will be called before any globally installed scripts.

```
$ workon my-diamond-app
(my-diamond-app) $
```

In order to leave the virtualenv and return to a normal shell, use `deactivate`.

```
(my-diamond-app) $ deactivate
$
```

`pip` and Installation

When you use `pip` inside your virtualenv, it will automatically install packages locally, instead of installing them at the system level. This way, it's easy to install anything without needing root access. In the following example, we are inside our virtualenv and we want to upgrade pip:

```
(my-diamond-app) $ pip install --force -U pip
Collecting pip
  Downloading pip-7.1.0-py2.py3-none-any.whl (1.1MB)
    100% || 1.1MB 172kB/s
Installing collected packages: pip
Successfully installed pip-7.1.0
(my-diamond-app) $
```

Freezing Python Requirements

`pip` provides a handy mechanism for printing all of the installed libraries, along with their versions. When called within your virtualenv, it produces a snapshot of all your project requirements so you can easily re-install them later.

```
(my-diamond-app) $ pip freeze
Jinja2==2.7.3
MarkupSafe==0.23
mr.bob==0.1.1
requests==2.7.0
virtualenv==1.11.6
virtualenvwrapper==4.2
(my-diamond-app) $
```

You can also store your requirements in a requirements file.

```
(my-diamond-app) $ pip freeze > requirements.txt
```

`Makefile` support

By default, Flask-Diamond provides `make install`, which will use requirements.txt to install your project's pre-requisites automatically.

Configuration Explanation

The use of configuration files permits your application to easily adapt to multiple environments. Often times, different systems will use different path structures, user accounts, database configurations, and TCP ports. Flask-Diamond implements the practices suggested by Flask 0.10, and by default stores these config files in the `./etc` folder of your project.

The SETTINGS Environment Variable

Flask-Diamond will load its configuration from whatever file is referenced by the `$SETTINGS` environment variable. You can use `$SETTINGS` to easily manage several profiles for your application. The following example demonstrates choosing the development profile stored in `dev.conf`.

```
export SETTINGS=$PWD/etc/conf/dev.conf
```

Another common way to control `$SETTINGS` is to use it as a prefix in front of a command. In the following example, the script `bin/manage.py` is invoked with the `dev.conf` profile to start the embedded HTTP server:

```
SETTINGS=$PWD/etc/conf/dev.conf bin/manage.py server
```

To start the server with the `production.conf` environment:

```
SETTINGS=$PWD/etc/conf/production.conf bin/manage.py server
```

Examples of Configurations

Flask-Diamond ships with a few configuration profiles to get you started:

dev.conf

The development environment is typically used on your developer workstation. You probably have root access to the machine. Any databases are probably temporary in nature, and exist mostly for testing purposes.

production.conf

The production environment is typically your front-facing web server (a "live server"). In this case, the application is probably not running as root. In fact, you may not even have root access to this machine. Thus, you must choose filenames for logging output that are owned by the application user's account. The production database is also likely to have different permissions, and unlike the development database, the production database probably has important information on it that you want to protect.

testing.conf

For testing purposes, there is a special configuration that writes to a temporary database that is created and destroyed during tests.

Makefile Support

If you inspect the `Makefile`, you will see that `$SETTINGS=$PWD/etc/conf/dev.conf` appears before most commands. Most cases will use `dev.conf` by default in order to protect against accidentally performing tasks upon the production database. Those prefixes are hardcoded so that a command like `make db` (which resets the database from scratch) cannot easily be applied to the production database.

Flask-Diamond Configuration Variables

By default, Flask-Diamond expects the following variables to be present within a configuration file.

Project

The project is configured with the following directives.

```
PROJECT_NAME = "Flask-Diamond"
PORT = 5000
LOG = "/tmp/dev.log"
LOG_LEVEL = "DEBUG"
SQLALCHEMY_DATABASE_URI = "sqlite:////tmp/flask-diamond-dev.db"
SECRET_KEY = "av^\x81\x03\xd7\xd1\xbd\x92~b\x00\xe8\xf7n9\x0e\xf8i\xdb\xba'\xa9\xea"
BASE_URL = "http://flask-diamond.org"
```

- `PROJECT_NAME`: the human-readable name of the project
- `PORT`: which TCP port will the HTTP server listen on?
- `LOG`: the filename to log messages to
- `LOG_LEVEL`: control the verbosity of logging output; DEBUG prints everything, INFO prints less, and WARN will only display problems.
- `SQLALCHEMY_DATABASE_URI`: a URI that points to your database. See Flask-SQLAlchemy for more examples.
- `SECRET_KEY`: a randomly-generated string that you created during scaffolding
- `BASE_URL`: the canonical name for your web application

Debugging

Debugging instructs the application to print extra information during operation. For example, there may be more verbose logging and it may be possible to inspect the application internals. All of this is helpful during development, but can be extremely dangerous in production.

```
DEBUG = False
DEBUG_TOOLBAR = True
DEBUG_TB_INTERCEPT_REDIRECTS = False
```

- `DEBUG`: when debugging is enabled, Flask produces tracebacks when an exception is encountered.
- `DEBUG_TOOLBAR`: whether to include Flask-DebugToolbar, which is a helpful in-browser debugging widget.
- `DEBUG_TB_INTERCEPT_REDIRECTS`: it is possible to inject the debug toolbar before URL redirects, which can be helpful for isolating routing problems.

Accounts and Security

Flask-Security provides an integrated platform of account security features, and Flask-Diamond incorporates most of its functionality. The following directives control Flask-Security.

```
SECURITY_PASSWORD_SALT = "aIf8ObrvtSTkIIGd"
SECURITY_POST_LOGIN_VIEW = "/admin"
SECURITY_PASSWORD_HASH = 'sha256_crypt'
SECURITY_URL_PREFIX = '/user'
SECURITY_CHANGEABLE = True
SECURITY_SEND_PASSWORD_CHANGE_EMAIL = False
SECURITY_CONFIRMABLE = False
SECURITY_REGISTERABLE = False
SECURITY_RECOVERABLE = False
SECURITY_TRACKABLE = True
SECURITY_EMAIL_SENDER = "accounts@flask-diamond.org"
```

- `SECURITY_PASSWORD_SALT`: The salt is a random string you generated during scaffolding. This is used to encrypt the password database.
- `SECURITY_POST_LOGIN_VIEW`: the name of the view to redirect to upon a successful login
- `SECURITY_PASSWORD_HASH`: the name of the hashing algorithm to use for passwords. **sha256_crypt** is recommended.
- `SECURITY_URL_PREFIX`: Change the URL prefix to make all account-related facilities appear as a subdirectory (like `/user`).
- `SECURITY_CHANGEABLE`: Can users change their own passwords?
- `SECURITY_SEND_PASSWORD_CHANGE_EMAIL`: Should users be notified by email when their password is changed?
- `SECURITY_CONFIRMABLE`: Must users confirm their email address in order to activate their account?
- `SECURITY_REGISTERABLE`: Is self-registration allowed?
- `SECURITY_RECOVERABLE`: Can a user reset their password if they have forgotten it?
- `SECURITY_TRACKABLE`: Does the User model include fields for recording User account history? By default, Flask-Diamond provides these fields. See the Flask-Security docs for more information about this.
- `SECURITY_EMAIL_SENDER`: What is the email address that security messages should be sent from?

ReCAPTCHA

Flask-Captcha provides a quick mechanism for ensuring your application is used by people instead of bots. You may recognize CAPTCHA as the squiggly letters and numbers that you must type into a text box. In order to get started with CAPTCHA and ReCAPTCHA, you must create a free account with their service.

```
RECAPTCHA_PUBLIC_KEY = '0000_00000000000000000000000000000000000000'
RECAPTCHA_PRIVATE_KEY = '0000_00000000000000000000000000000000000000'
```

- `RECAPTCHA_PUBLIC_KEY`: The ReCAPTCHA online service will provide you with a public key, which will be included with your web application.
- `RECAPTCHA_PRIVATE_KEY`: ReCAPTCHA also provides a private key, but this one must be kept secret. You will enter it in this configuration file, but nowhere else.

Flask-Mail

The simplest way for your application to send email is using Flask-Mail, which makes it pretty easy to create and send emails.

```
MAIL_SERVER = '127.0.0.1'
MAIL_PORT = 25
MAIL_USE_TLS = False
MAIL_USERNAME = None
MAIL_PASSWORD = None
```

- `MAIL_SERVER`: the hostname or IP address of your SMTP server
- `MAIL_PORT`: the port used by your SMTP server. Usually, this is 25 or 465.
- `MAIL_USE_TLS`: If the server supports or requires encryption (with TLS), then set this to *True*
- `MAIL_USERNAME`: If you must provide authentication information to your server in order to send email through it, then provide the username here.
- `MAIL_PASSWORD`: As with the username, provide the password here if it is required.

Celery

Celery is a job queue that has been integrated into Flask-Diamond so that you create background tasks for any operations that take a while to complete. Typically, you will want your application to respond to requests within 100ms, but when this is not possible, you can achieve a rapid response by queueing the slow operation so that it executes separately. This way, it is still possible to respond to requests quickly enough that nobody will notice.

```
CELERY_BROKER_URL = 'sqla+sqlite:///var/db/celerydb.sqlite'
CELERY_RESULT_BACKEND = 'db+sqlite:///var/db/results.sqlite'
```

- `CELERY_BROKER_URL`: the URL pointing to a database connection. This is like the SQLAlchemy URI, but different enough that you should consult the documentation.
- `CELERY_RESULT_BACKEND`: Celery is able to store job results in a separate database, and for certain types of jobs, this is recommended. The URI here is similar to but different from the *CELERY_BROKER_URL*.

`Makefile` Explanation

The Makefile that ships with Flask-Diamond by default includes a number of targets that address several common tasks throughout the life cycle of a project. The way to use the Makefile is with the `make` command. Thus, to install the project with make, you'd invoke `make install`.

During development, the Makefile is one of the primary ways for you to interact with your project. You may find yourself running `make db server` or perhaps `make single` with some regularity. It is recommended to become familiar with the Flask-Diamond Makefile.

Integration

These targets control project builds.

- `install`: Use setup.py to install the project (typically inside a virtualenv).
- `clean`: Delete all of the temporary files created by `install`.
- `docs`: Use Sphinx to render the documentation in `etc/sphinx`.

Development

These targets are used to run the application with the `dev.conf` profile.

- `server`: Invoke the HTTP server in debug mode with `dev.conf`
- `shell`: Enter a python (or ipython) shell within the virtualenv.
- `notebook`: Use ipython notebook, if installed, to inspect the application.

Testing

These targets are used to run automated tests.

- `test`: Run all of the tests using nosetests.
- `single`: Run only tests marked with the `@attr("single")` decorator.
- `watch`: Enter a loop that watches for any project source code to be changed, and then automatically run any tests marked with the `@attr("single")` decorator.

Databases

These targets control the database. By default these use the `dev.conf` profile so as to avoid inadvertently changing the production database.

- `db`: drop the database, re-create the database, and populate the database with starter values.
- `newmigration`: when the data model schema has changed, use `newmigration` to create a new data migration.
- `migrate`: apply all data migrations, in order, until the database is up to date.

`manage.py` Explanation

Many applications will want to expose some functionality through a command line interface, and `bin/manage.py` provides an easy way to accomplish this. For example:

- certain administrative tasks could be triggered from the command line
- other tasks can be automated using a tool like `cron`

Many of the targets within the Flask-Diamond Makefile are actually wrappers for `manage.py`, but you can invoke it manually on the command line like so:

```
SETTINGS=$PWD/etc/conf/dev.conf bin/manage.py runserver
```

The following commands come with Flask-Diamond by default.

Commands

- `shell`: Launch the Python REPL (or iPython if installed) using Flask-Script. By default, the following objects will be imported into the namespace:
  - `app`: your app's Flask-Diamond object
  - `db`: your app's database object

  - `model`: your app's model

- `runserver`: Launch your application's HTTP server. When `runserver` is invoked, it will bind to `localhost`. The `PORT` your application listens on is defined in the *Configuration Explanation*.
- `publicserver`: Like `runserver` but public. This causes the server to bind to `0.0.0.0` so that remote hosts can connect to your application. This is intended for development purposes, and is not recommended for deployment. See *Web Services with WSGI* for more information about running a public web service.
- `db`: This command acts as the entry point for Flask-Migrate. The subcommands available, taken directly from the command output, are:
  - `upgrade`: Upgrade to a later version
  - `heads`: Show current available heads in the script directory
  - `show`: Show the revision denoted by the given symbol.
  - `migrate`: Alias for 'revision –autogenerate'
  - `stamp`: 'stamp' the revision table with the given revision; don't run any migrations
  - `current`: Display the current revision for each database.
  - `merge`: Merge two revisions together. Creates a new migration file
  - `init`: Generates a new migration
  - `downgrade`: Revert to a previous version
  - `branches`: Show current branch points
  - `history`: List changeset scripts in chronological order.
  - `revision`: Create a new revision file.
- `useradd`: Add a user to the users database via Flask-Security. This accepts the following arguments:
  - `email`: (required)
  - `password`: (required)
  - `admin`: *True/False* Should this user have the *Admin* role?
- `userdel`: Delete a user from the users database.
- `init_db`: Drop the existing database using Flask-SQLAlchemy and re-create it. This obviously destroys anything in the database, resetting it to its original state.

Web Services with WSGI

For deploying your application in a production environment, you will probably end up using a WSGI application server like uwsgi or gunicorn. By default, Flask-Diamond will install gunicorn as a requirement.

System Start-up

Under most circumstances, you will want to automatically run the application server when the host boots. This is going to be different for every host, but an example demonstrates launching gunicorn via Ubuntu upstart:

```
#!upstart
description "flask-diamond daemon"

env USER=flask-diamond
env SETTINGS=/etc/flask-diamond.conf

start on runlevel [2345]
stop on runlevel [06]

respawn

exec start-stop-daemon --start \
    --make-pidfile \
    --pidfile /var/run/$USER-daemon.pid \
    --chuid $USER \
    --exec /var/lib/$USER/.virtualenvs/$USER/bin/gunicorn -- \
        --workers 2 \
        --bind 0.0.0.0:5000 \
        --user $USER \
        --chdir /var/lib/$USER \
        --log-file /var/lib/$USER/gunicorn-error.log \
        --access-logfile /var/lib/$USER/gunicorn-access.log \
        --pid /var/run/$USER-daemon.pid \
        --daemon \
        flask_diamond.wsgi:app
```

This demonstrates several important principles:

- setting the configuration file that will control the application server
- launching the application server from inside the Python virtual environment

Reverse Proxy

You probably want to keep your application server behind a firewall, so a common pattern for deploying Flask-Diamond applications relies upon a reverse proxy. There is a good Digital Ocean tutorial for setting up a reverse proxy with either nginx or apache.

Puppet-Diamond

For scaling up deployment, it is recommended to use an automation solution like Puppet or Chef. Flask-Diamond is particularly easy to deploy with Puppet-Diamond, which will simplify the management of the host configurations as well as application deployment.

Embedded Server

Flask-Diamond provides an embedded web server with `bin/manage.py` and `bin/runserver.py` for simple deployments, like development and debugging. This is not recommended for production use. However, when paired with a reverse proxy, this is actually good enough to handle a surprising amount of traffic.

IT Operations with Fabric

There is frequently a split between the application development environment and the live deployment environment. When the application ends up running on a remote host while you are doing your work on a workstation, it can be helpful to simplify remote system operations. Flask-Diamond uses Fabric to make it pretty easy to invoke system functions from the command line.

Using Fabric

To use the Flask-Diamond Fabric functionality, navigate to the root directory of the project and issue the following command:

```
fab help
```

This will list all of the available commands.

Flask-Diamond Fabric Commands

By default, Flask-Diamond provides a `fabfile.py` with the scaffold with the following functionality:

- **rsync**: copy files directly from the working directory to the remote host
- **pull**: on the remote host, execute `git pull` in the application directory
- **setup**: run `make install` on the remote host
- **ipython**: enter an ipython shell on the remote host
- **shell**: open a bash shell on the remote host
- **restart**: restart the remote application server
- **nginx_restart**: restert the remote web server
- **logs**: view the application logs

Customizing Fabric

Because all systems are different, it is not too likely that all of the commands in `fabfile.py` will work. However, this at least provides a starting point.

CHAPTER 7

API Reference

API

This documentation is for Flask-Diamond 0.5.1.

Diamond object

class flask_diamond.**Diamond**(*name=None*)

A Diamond application.

Parameters **app** (*Flask*) – a Flask app that you created on your own

Returns None

facet(*extension_name*, **args*, ***kwargs*)

initialize an extension

super(*extension_name*, **args*, ***kwargs*)

invoke the initialization method for the superclass

ex: self.super("administration")

teardown(*exception*)

Remove any persistent connections during application context teardown.

Returns None

models

models.user

class flask_diamond.models.user.**User**(***kwargs*)

Bases: flask_sqlalchemy.Model, flask_security.core.UserMixin, flask_diamond.mixins.crud.CRUDMixin, flask_diamond.mixins.marshmallow.MarshmallowMixin

active

boolean – whether the user account is active

add_role(*role_name*)

update a User account so that it includes a new Role

Parameters **role_name** (*string*) – the name of the Role to add

confirm()
update a User account so that login is permitted

Returns None

confirmed_at
datetime – when the user account was confirmed

current_login_at
datetime – the time of the current login, if any

current_login_ip
string – the IP address of the current login

email
string – email address

id
integer – primary key

last_login_at
datetime – the time of the most recent login

last_login_ip
string – the IP address of the previous login

login_count
integer – the number of times this account been accessed

password
password – the users's password

classmethod register(*email*, *password*, *confirmed=False*, *roles=None*)
Create a new user account.

Parameters

- **email** (*string*) – the email address used to identify the account
- **password** (*string*) – the plaintext password for the account
- **confirmed** (*boolean*) – whether to confirm the account immediately
- **roles** (*list(string)*) – a list containing the names of the Roles for this User

roles

class flask_diamond.models.user.**UserSchema**(*obj=None*, *extra=None*, *only=None*, *exclude=None*, *prefix=u''*, *strict=False*, *many=False*, *skip_missing=False*, *context=None*)
Bases: flask_marshmallow.Schema

class Meta

additional = ('id', 'email', 'password', 'active', 'last_login_ip', 'current_login_ip', 'login_count')

dateformat = '%F %T %z'

models.role

class flask_diamond.models.role.**Role**(***kwargs*)
Bases: flask_sqlalchemy.Model, flask_security.core.RoleMixin, flask_diamond.mixins.crud.CRUDMixin, flask_diamond.mixins.marshmallow.MarshmallowMixin

For the purpose of access controls, Roles can be used to create collections of users and give them permissions as a group.

classmethod add_default_roles()
Create a basic set of users and roles

Returns None

description
string – a sentence describing the role

id
integer – primary key

name
string – what the role is called

class flask_diamond.models.role.**RoleSchema**(*obj=None*, *extra=None*, *only=None*, *exclude=None*, *prefix=u''*, *strict=False*, *many=False*, *skip_missing=False*, *context=None*)
Bases: flask_marshmallow.Schema

class Meta

additional = ('id', 'name', 'description')

mixins

mixins.crud

class flask_diamond.mixins.crud.**CRUDMixin**
Convenience functions for CRUD operations.

Adapted from flask-kit.

classmethod create(*_commit=True*, ***kwargs*)
Create a new object.

Parameters

- **commit** (*boolean*) – whether to commit the change immediately to the database
- **kwargs** (*dict*) – parameters corresponding to the new values

Returns the object that was created

delete(*_commit=True*)
Delete this object.

Parameters **commit** (*boolean*) – whether to commit the change immediately to the database

Returns whether the delete was successful

classmethod find(***kwargs*)
Find an object in the database with certain properties.

Parameters **kwargs** (*dict*) – the values of the object to find

Returns the object that was found, or else None

classmethod **find_or_create**(*_commit=True*, ***kwargs*)
Find an object or, if it does not exist, create it.

Parameters **kwargs** (*dict*) – the values of the object to find or create

Returns the object that was created

classmethod **get_by_id**(*id*)
Retrieve an object of this class from the database.

Parameters **id** (*integer*) – the id of the object to be retrieved

Returns the object that was retrieved

save(*_commit=True*)
Save this object to the database.

Parameters **commit** (*boolean*) – whether to commit the change immediately to the database

Returns the object that was saved

update(*_commit=True*, ***kwargs*)
Update this object with new values.

Parameters

- **commit** (*boolean*) – whether to commit the change immediately to the database
- **kwargs** (*dict*) – parameters corresponding to the new values

Returns the object that was updated

mixins.marshmallow

class flask_diamond.mixins.marshmallow.**MarshmallowMixin**

dump()
serialize the Model object as a python object

classmethod **dump_all**()
write all objects of Model class to an array of python objects

dumpf(*file_handle*)
write a Model object to file_handle as a JSON string

classmethod **dumpf_all**(*file_handle*)
write all objects of Model class to file_handle as JSON

dumps()
serialize the Model object as a JSON string

classmethod **dumps_all**()
write all objects of Model class to a JSON-encoded array

classmethod **load**(*python_obj*)
create a Model object from a python object

classmethod **load_all**(*python_objects*)
create objects of Model class from an array of python objects

classmethod loadf (*file_handle*)
create a Model object from a file_handle pointing to a JSON file

classmethod loadf_all (*file_handle*)
create objects of Model class from a file containing an array of JSON-encoded objects

classmethod loads (*buf*)
create a Model object from a JSON-encoded string

classmethod loads_all (*buf*)
create objects of Model class from a string containing an array of JSON-encoded objects

facets

facets.accounts

flask_diamond.facets.accounts.**init_accounts**(*self*, *user=None*, *role=None*, **args*, ***kwargs*)

Initialize Security for application.

Parameters **kwargs** (*dict*) – parameters that will be passed through to Flask-Security

Returns None

A number of common User account operations are provided by Flask- Security. This function is responsible for associating User models in the database with the Security object.

In case you need to override a Flask-Security form (as is the case with implementing CAPTCHA) then you must use super() from within your application and provide any arguments destined for Flask-Security.

```
>>> result = self.super("accounts", user=User, role=Role,
>>>     confirm_register_form=ExtendedRegisterForm)
```

facets.administration

flask_diamond.facets.administration.**init_administration**(*self*, *index_view=None*, *user=None*, *role=None*)

Initialize the Administrative GUI.

Parameters **index_view** (*AdminIndexView*) – the View that will act as the index page of the admin GUI.

Returns None

The administration GUI is substantially derived from Flask-Admin. When this function is called, it will instantiate blueprints so the application serves the admin GUI via the URL http://localhost/admin.

Typically, you will want to call this function even if you override it. The following example illustrates using super() to invoke this administration() function from within your own application.

```
>>> admin = super(MyApp, self).administration(
>>>     index_view=MyApp.modelviews.RedirectView(name="Home")
>>> )
```

facets.blueprints

flask_diamond.facets.blueprints.**init_blueprints**(*self*)
Initialize blueprints.

Returns None

By default, this function does nothing. Your application needs to overload this function in order to implement your View functionality. More information about blueprints can be found in the Flask documentation.

facets.configuration

`flask_diamond.facets.configuration.`**`init_configuration`**(*self*)
Load the application configuration from the `SETTINGS` environment variable.

Returns None

`SETTINGS` must contain a filename that points to the configuration file.

facets.database

`flask_diamond.facets.database.`**`init_database`**(*self*)
Initialize database

Returns None

Flask-Diamond assumes you are modelling your solution using an Entity- Relationship framework, and that the application will use a relational database (e.g. MySQL, Postgres, or SQlite3) for model persistence. Thus, SQLAlchemy and Flask- SQLalchemy are used for database operations.

Typically, this just works as long as `SQLALCHEMY_DATABASE_URI` is set correctly in the application configuration.

facets.debugger

`flask_diamond.facets.debugger.`**`init_debugger`**(*self*)
Initialize the DebugToolbar

Returns None

The DebugToolbar is a handy utility for debugging your application during development.

This function obeys the `DEBUG_TOOLBAR` configuration setting. Only if this value is explicitly set to True will the Debug Toolbarr run.

facets.email

`flask_diamond.facets.email.`**`init_email`**(*self*)
Initialize email facilities.

Returns None

Flask-Mail is a useful tool for creating and sending emails from within a Flask application. There are a number of configuration settings beginning with `MAIL_` that permit control over the SMTP credentials used to send email.

facets.forms

`flask_diamond.facets.forms.`**`add_helpers`**(*app*)
Create any Jinja2 helpers needed.

flask_diamond.facets.forms.**init_forms**(*self*)
WTForms helpers

Returns None

WTForms is a great library for using forms and Flask-WTF provides good integration with it. WTForms helpers enable you to add custom filters and other custom behaviours.

facets.handlers

flask_diamond.facets.handlers.**init_error_handlers**(*self*)
Initialize handlers for HTTP error events

Returns None

Flask is able to respond to HTTP error codes with custom behaviours. By default, it will redirect error 403 (forbidden) to the login page.

flask_diamond.facets.handlers.**init_request_handlers**(*self*)
request handlers

Returns None

Flask handles requests for URLs by scanning the URL path. Typically, any serious functionality will be collected into Views. However, this function is a chance to define a few simple utility URLs.

If in your application you want to disable the default handlers in Flask-Diamond, you can override them like this.

```
>>> def request_handlers(self):
>>>     pass
```

facets.logs

flask_diamond.facets.logs.**init_logs**(*self*)
Initialize a log file to collect messages.

Returns None

This file may be written to using

```
>>> flask.current_app.logger.info("message")
```

facets.marshalling

flask_diamond.facets.marshalling.**init_marshalling**(*self*)
Initialize Marshmallow.

Returns None

facets.rest

flask_diamond.facets.rest.**init_rest**(*self*, *api_map=None*)
Initialize REST API.

Returns None

By default, this function does nothing. Your application needs to overload this function in order to implement your REST API. More information about REST can be found in the documentation.

api_map is an optional function that can be responsible for setting up the API. This is usually accomplished with a series of add_resource() invocations. api_map must take one parameter, which is the Flask-Restful object managed by Flask-Diamond.

You will end up writing something like this in your application:

class PlanetResource(Resource):

def get(self, name): planet = Planet.find(name=name) if planet:

return(planet.dump())

def api_map(rest_extension): rest_extension.add_resource(PlanetResource, '/api/planet/<string:name>')

def create_app(): application.facet("rest", api_map=api_map)

facets.signals

`flask_diamond.facets.signals.`**`init_signals`**(*self*)
Initialize Flask signal handlers

Returns None

Flask provides a number of signals corresponding to things that happen during the operation of the application, which can also be thought of as events. It is possible to create signal handlers that will respond to these events with some behaviour.

facets.task_queue

`flask_diamond.facets.task_queue.`**`init_task_queue`**(*self*)
Initialize celery.

facets.webassets

`flask_diamond.facets.webassets.`**`init_webassets`**(*self*, *asset_map=None*)
Initialize web assets.

Returns None

webassets make it simpler to process and bundle CSS and Javascript assets. This can be baked into a Flask application using Flask-Assets

CHAPTER 8

Open Source Software

How to Contribute to the Project

Flask-Diamond welcomes contributions from everybody, and there are several ways you can help! The first step is always to clone the project repository using git. If you haven't done this yet, then do it now. Then, give the rest of this document a read for some specific ideas about contributing.

Implement your favourite features

If you know a feature you'd like to code in order to help with Flask-Diamond, then the easiest way to help is by submitting a pull request to Flask-Diamond on GitHub. From your perspective, there is no barrier preventing you from contributing to Flask-Diamond today.

Help with the documentation

Project documentation is one of the most important aspects of an open source project. You can help immensely by editing the current documentation for clarity. You can find the documentation in the Flask-Diamond repository, which you can modify by submitting a pull request.

Help with an existing issue

Another way you can contribute is by working directly upon issues that have been submitted by the community using the Issue Tracker. This is more advanced than simply implementing a new feature, because the issue may specify acceptance criteria. It is recommended that you coordinate with project members before working on an issue. The easiest way to contact the team is through the Issue Tracker itself, because each issue has a comment thread associated with it.

Submit your first pull request

The only requirement for your first pull request that you must add your name to the *Contributors List*, signifying your acceptance of the *Flask-Diamond Contributors Agreement*.

Become a project developer

If you have proven yourself to consistently deliver great work on Flask-Diamond, then you should join the development team! The team uses a Project Kanban to coordinate its development efforts. In order to join the team, you must be familiar with Agile-Diamond, which is the project management framework used for working on Flask-Diamond. Other than that, you must simply open an issue in the Issue Tracker requesting to become a team member.

Contributors

Flask-Diamond is a volunteer effort. This page discusses the Contributors Agreement and Contributors List. This page is also the official document for signing up as a contributor to the Flask-Diamond project. To learn more about how you can volunteer, please read *How to Contribute to the Project*.

Contributors Agreement

Contributors to the Flask-Diamond project affirm:

- Because the *License* chosen for publishing Flask-Diamond complies with the principles of Open Source Software; and
- Because works contributed to this project could become part of Flask-Diamond; and
- Because all contributions to the project are either 1) the original work of the contributor; or 2) include proper attribution to the original author as permitted by applicable Open Source licenses; and
- Because the Flask-Diamond copyright may be transferred to a qualified Open Source organization in the future; therefore

For the benefit of the Flask-Diamond project:

- Contributors will assign the rights to their contributed works to the the Flask-Diamond author, Ian Dennis Miller; and
- Contributors will add their name or well-known pseudonym to the *Contributors List*.

Contributors to the Flask-Diamond project signify acceptance of the Contributors Agreement by submitting a git pull request appending themselves to the *Contributors List*.

Contributors List

- iandennismiller

License

The MIT License (MIT)

Flask-Diamond

Change Log

This file contains a summary of major changes to Flask-Diamond.

0.5.1

2017-01-05

- fixed flask-diamond CLI python3 compatibility bug

0.5.0

2017-01-04

- updating flask-security to 1.7.5, flask to 0.11.1
- more robust accounts facet
- moving templates out of core library into skel
- renamed all intances of flask.ext.whatever as flask_whatever
- update rest API docs and skel
- refactor of cli
- migrate global junk into scaffolds
- consolidate Planets tutorial
- update documentation
- testing uses latest available flask-diamond version
- permit testing of core flask-diamond library without scaffolding
- screencast in readme
- simplify naming of documentation files

0.4.12

2016-11-14

- remove tutorial dependencies from basic tests

0.4.11

2016-11-14

- ensure user registration signal works

0.4.10

2016-11-14

- ensure skel references latest version of flask-diamond

0.4.9

2016-11-10

- add user name to GUI Create form

0.4.8

2016-11-10

- fix other python 3 errors

0.4.7

2016-11-05

- fix escaping problem with flask-diamond scaffold

0.4.6

2016-07-25

- update scaffolds to be compatible with py3

0.4.5

2016-07-25

- add example REST interface to scaffolds

0.4.4

2016-07-25

- fix documentation
- fix namespace conflict with REST api

0.4.3

2016-06-09

- permit any sufficiently modern version of Fabric3

0.4.2

2016-05-22

- using Fabric3 instead of Fabric

0.4.1

2016-05-21

- fixed typo in flask-diamond command line app

0.4.0

2016-05-21

- renamed extensions to "facets" (backwards incompatible change)
- new command line utility: "flask-diamond" (was diamond-scaffold.sh)
- enhanced code skeletons: example-models and example-views
- migrated many application components from Flask-Diamond proper into skeletons
- migrated all testing into code skeletons to test actual usage scenario
- deeper integration with Travis CI, including testing using code skeletons
- generally cleaned up the Flask-Diamond core

0.3.6

2016-05-17

- do not fail during setup if files cannot be copied

0.3.5

2016-05-03

- Python 3 compatibility
- added Travis CI
- added Planetary model for testing
- testing for marshmallow

0.3.4

2016-04-29

- adding username to User model schema

0.3.3

2016-04-29

- Python 3 compatibility
- refactor test mixin

0.3.2

2016-04-26

- Python 3 compatibility

0.3.1

2016-04-25

- Python 3 compatibility

0.3.0

2016-01-21

- a new application startup procedure based on extensions
- putting skels into project
- integrate wsgi launcher
- refine questions, add version to project config
- making readme generate from a template
- use alabaster doc theme
- refactor extensions into separate namespace
- added super method, fixing templates
- removing Individual from the skeleton
- refactoring mixins
- simplify views, create separate skel
- added change log document
- documented API creation
- diagram of libraries
- wrote sphinx docs
- wrote email document, testing document

- wrote about user accounts, project diagram
- wrote wsgi, fabric

0.2.16

2015-01-21

- putting skels into project

0.2.15

2015-08-05

- switch theme to be flask-esque
- adding new documentation stubs and restructuring TOC
- added build details to footer
- created quick-start
- stubbed several new documents
- gather git hash using a different command
- wrote scaffolding explanation
- wrote philosophy and some of the learning section
- starting GUIs with Flask-Admin
- remove sqlite from requirements for documentation build
- separate requirements from installation
- remove pysqlite2 requirement
- added relationship examples to models, rounded out gui examples
- finishing Views documentation
- update migration process

0.2.13

2015-07-30

- controlling documentation more closely
- migrating markdown documentation to sphinx
- inter-linking github, pypi, and readthedocs
- add resources to REST api before calling init_app

0.2.12

2015-07-30

- This release was used to debug packaging and documentation.

0.2.11

2015-07-30

- This release was used to debug packaging and documentation.

0.2.10

2015-07-29

- separate models into submodules
- remove backref on user roles to permit easier inheritance and overloading of the User model
- store requirements in separate file
- split documentation into smaller files

0.2.9

2015-07-08

- admin views can be turned off
- admin views can be toggled
- Create Dependencies.md

0.2.8

2015-06-01

- Update manage.py

0.2.7

2015-05-13

- include marshmallow mixin
- loads() from unmarshalled data
- load(), loads(), loadf()

0.2.6

2015-04-24

- hardcoding alembic because the latest version does not parse correctly in FlaskMigrate
- can disable admin views

0.2.5

2015-03-20

- useradd and userdel
- migrate conf files into subdir
- decent isolation of blueprints, but weirdness with security

0.2.4

2015-03-15

- bump flask-admin version
- fixed user create with password
- fixed layout of login page

0.2.3

2015-03-03

- mrbob

0.2.2

2015-03-03

- bump requirements
- reduce required libraries

0.2.1

2015-02-17

- delayed commit in CRUD
- default repr in CRUD
- bump flask script and SQLAlchemy

0.2.0

2015-02-07

- use latest Flask-Migrate==1.3.0
- move user management into user model
- remove unnecessary variables
- reorganize
- meta script helps keep skels aligned
- trying to get migrations neat

- working meta-build
- simpler test fixture
- using relative paths
- scaffolding util
- repair manifest
- fixing paths for databases
- tweak documentation
- automatically sync github pages with API documentation
- API more prominent
- autosync documentation
- include description in sphinx main document
- documented every method

0.1.10

2015-02-04

- freeze versions of other dependencies
- update docs

0.1.9

2015-01-25

- PEP8 for setup, migrate a few Flask libraries into the core

0.1.8

2014-11-19

- it is possible to contol the AdminIndexView during app creation

0.1.7

2014-06-29

- use new class instantiation for flask-mail

0.1.6

2014-06-23

- remove ipython dependency

0.1.5

2014-06-16

- more robust user creation
- admin object local to entire package
- update flask-admin dependency

0.1.3

2014-03-29

- do not require a specific version of distribute
- include webassets

0.1.2

2014-03-22

- correct auth mixin ordering
- load/save mixins

0.1.1

2014-03-20

- split error handlers and request handlers
- support changeable passwords
- removed hardcoded config options
- code annotation
- steps towards PEP8
- following Flask capitalization conventions
- account functions are behind /user URL
- CRUD create() may defer commit

0.1

2014-03-06

- Initial public release.

CHAPTER 9

Online Resources

- GitHub Project Page
- Issue Tracker
- Python Project on PyPI
- Diamond Methods

f

A

C

D

E

F

G

I

L

M

N

P

R

S

T

U