Flaskage Documentation

Release 0.3

Fotis Gimian

May 04, 2014

Contents

Welcome to Flaskage's documentation. The Flaskage template attempts to bring together all the best packages and integrate them with Flask so you have a full stack MVC (or MTV) structure ready-to-use.

Flaskage takes a lot of inspiration from frameworks like Ruby on Rails, Play and Laravel and doesn't attempt to re-create the Django way of working.

In addition to providing a template, this project also intends to document common workflows with the template and serve as a good general point of reference.

Some of the existing templates and projects which inspired and influenced this project are:

- overholt
- flasky
- flask_website
- beancounter
- flaskr-bdd
- fbone
- cookiecutter-flask
- flask-chassis

CHAPTER 1

User's Guide

This part of the documentation focuses on using Flaskage along with common patterns that you can use as a reference while developing your application.

1.1 Overview

Flaskage incorporates a large number of features and integrates almost everything you will ever need to build a large web application.

1.1.1 Features

So what makes Flaskage unique? A few little things:

- **Clean and simple**: Although I'm designing Flaskage for medium to larger sized projects, I wanted to ensure that the boilerplate code was kept to a minimum. Instead, I've opted for comments which provide examples of what you can do rather than starting to write an application in the template.
- **One directory per function**: It seems that most templates are inspired by the way that Django separates apps, whereby each component of the larger web application has its own models, views, templates and static files. I personally feel that this layout doesn't make sense. Instead, I prefer a structure more similar to the Play or Ruby on Rails which keeps views in one directory, models in one directory and so on. The ability to split views and models into multiple files is an absolute must which is also part of Flaskage's design.
- **Full integration of Flask-Assets**: This template has been designed for use with Coffeescript, LESS (in particular Twitter's Bootstrap CSS framework) or any other pre-processor you may have in mind. Furthermore, Flaskage keeps all such uncompiled files neatly in an assets directory.
- **Database migrations**: Flaskage integrates Flask-Migrate and is ready for database migrations which can be invoked via management commands.
- **Switchable configurations**: With a simple command line switch, you can run the development server under any environment you wish (development, production or testing). Further to this, you can set a default environment for your app to run in via a variable in the config module or define your own custom config environments.
- **Flake8 integration**: You can check that your syntax is valid and that your coding style follows the PEP8 standard with a simple management command.
- **Clean client-side library integration**: Flaskage uses Bower and symlinks to cleanly integrate Twitter Bootstrap and jQuery with the ability to seamlessly upgrade these components when necessary and avoid duplication of the original source code in your Git repository.

- **More robust development server**: Using flask-failsafe, the development server won't crash each time small errors are made while coding.
- **Travis Integration**: Test case integration with Travis is provided out of the box.
- **Powerful test tools**: Integrated use of nose, Coverage.py, factory_boy and fake-factory.
- **Python 3 ready**: I have only chosen extensions which work across Python 2.6, 2.7 and 3.3 so that you're future-proof if and when you decide to move to a Python 3 environment.
- **PyJade integration (optional)**: If you choose to, you may use PyJade to write your templates. This is a far less verbose language than regular HTML.
- **Behaviour-driven development (optional)**: Integrated use of behave, splinter and selenium for fully featured behaviour-driven development.

1.1.2 Project Structure

Flaskage is structured as shown below:

```
|-- app
|   |-- assets
|   |-- models
|   |-- static
|   |-- templates
|   '-- views
|-- db
|   '-- migrations
|-- features
|   '-- steps
|-- lib
|-- test
|   |-- factories
|   |-- lib
|   |-- models
|   '-- views
|-- vendor
|   '-- assets
|-- .bowerrc
|-- .travis.yml
|-- bower.json
|-- config.py
|-- manage.py
|-- requirements.txt
|-- setup.cfg
'-- wsgi.py
```

The purpose of each file and directory are as follows:

- **app**: Main web application directory with app initialisation
  - **assets**: Pre-compiled script and stylsheet assets
  - **models**: Database model definitions
  - **static**: Static files such as CSS, Javascript and images
  - **templates**: Jinja2 templates for presentation
  - **views**: Views and related forms that provide business logic for each page
- **db**: Database related code and binaries

    - **migrations**: The generated Alembic database migrations
- **features**: Feature definitions in the Gherkin language to be used for BDD
    - **steps**: Test code which validates that each feature works as expected
- **lib**: Supporting libraries you have developed for your web application
- **test**: Unit tests for testing your web application
    - **factories**: factories created using factory_boy that are used to create model instances
    - **lib**: Unit tests which test libraries
    - **models**: Unit tests which test models
    - **views**: Unit tests which test views
- **vendor**: Vendor provided libraries
    - **assets**: Third-party Javascript and Stylsheet assets (via Bower)
- **.bowerrc**: Overrides the default installation directory for Bower components
- **.travis.yml**: The Travis configuration for the application
- **bower.json**: Vendor provided client-side package requirements
- **config.py**: Configuration for development, production and test environments
- **manage.py**: Management interface and command registrations
- **requirements.txt**: Python package requirements
- **setup.cfg**: General package configuration (used for nose)
- **wsgi.py**: Exposes the production application to your production WSGI server

1.2 Installation

Flaskage is tested against the most common Linux distributions and Python versions to ensure seamless usage. Please follow the steps below to prepare your operating system and Python environment.

1.2.1 Preparing Your Operating System

Flaskage supports the following Linux operating systems:

- Debian 6 (Squeeze)
- Debian 7 (Wheezy)
- Ubuntu Server 10.04 LTS (Lucid Lynx)
- Ubuntu Server 12.04 LTS (Precise Pangolin)
- Ubuntu Server 14.04 LTS (Trusty Tahr)
- CentOS 5.x
- CentOS 6.x
- Red Hat Enterprise Linux 5.x
- Red Hat Enterprise Linux 6.x

Note: Although Flaskage is designed to work on Python 3.3 and above, it is recommended that you develop projects on Python 2.7 until the Python 3 ecosystem further matures.

Python 3 really hasn't presented any compelling advantages over its prior version and has added some complexity around aspects of the language which many (including me) feel is unnecessary.

You'll need to install some pre-requisites to ensure that all Python packages install correctly.

If you're running Python 2.6 or 2.7 Debian or Ubuntu Server:

```
$ sudo apt-get install gcc python-dev git
```

If you're running CentOS 6.x or Red Hat Enterprise Linux 6.x:

```
$ sudo yum install gcc python-devel git
```

If you're running CentOS 5.x or Red Hat Enterprise Linux 5.x, you'll need to use the python26 package from EPEL. You may then install required dependencies as follows:

```
$ sudo rpm -Uvh http://download.fedoraproject.org/pub/epel/5/i386/epel-release-5-4.noarch.rpm
$ sudo yum install gcc python26 python26-devel
```

1.2.2 Preparing Your Python Environment

Flaskage supports the following Python versions:

- CPython 2.6
- CPython 2.7
- CPython 3.3
- CPython 3.4
- PyPy 2.2

If you haven't already, you'll need to first install setuptools, pip and virtualenv as follows:

```
$ curl https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py | sudo python
$ easy_install pip
$ pip install virtualenv
```

Create a virtualenv and install the Flaskage packages:

```
$ mkdir ~/.virtualenv
$ virtualenv ~/.virtualenv/flaskage
$ source ~/.virtualenv/flaskage/bin/activate
$ pip install git+git://github.com/fgimian/flaskage.git
```

This will make an executable script named **flaskage** available which allows you to create a new project and generate scaffolding. More on that to follow in the next section.

1.2.3 Installing Node.js Components

In order to use Bower, LESS, Clean CSS, Coffeescript and UglifyJS, we need to install the necessary modules on our system via Node.js.

Firstly, ensure that your system has the latest Node.js installed and then run the following:

```
$ [sudo] npm install -g bower less clean-css coffee-script uglify-js
```

Note: If your Node.js installation is global and owned by root, you'll need to run the command above using sudo.

1.3 Creating Your Project

Now that we have Flaskage environment setup, we can create our first project and generated some basic boilerplate code using scaffolding.

1.3.1 Creating a New Project

To create a new project, simply run the following:

```
$ flaskage new <project_name>
```

Note: The project name must be in lowercase underscore format, much like regular Python variable names.

Alternatively, you may provide a full path to your new application too:

```
$ flaskage new ~/webapps/<project_name>
```

This will generate a new project structure for you.

Please follow the provided instructions to prepare your project for running.

Once you have completed installing all the necessary components, you may start the Flask development server using the following command as described in the instructions:

```
./manage.py server
```

This will run the development server on loopback address which will mean that it will only be available for viewing by your development server.

If you wish to make the website available to other machines on the same network, then start the development server as follows:

```
./manage.py server -t 0.0.0.0
```

1.3.2 Generating Scaffolds

A typical Flaskage web application is made up of the following components:

- Blueprints (application components) and their related templates
- Helpers designed to provide utility functions or classes for the application
- Javascript and Stylsheet assets (using Coffeescript and LESS respectively) intended to be used for particular blueprints
- Database models
- Libraries designed to provide generic non-application specific utility functions or classes for the application and other applications in future

You may generate the required boilerplate code for each of these components using the **flaskage** command.

Please ensure that you follow all on-screen instructions to activate the new component in your application.

Note: In all examples below, the name must be in lowercase underscore format.

To generate a new blueprint:

```
flaskage generate blueprint <blueprint_name>
```

To generate a new helper:

```
flaskage generate helper <helper_name>
```

To generate a new set of assets:

```
flaskage generate asset <asset_name>
```

To generate a new model:

```
flaskage generate model <model_name> [<column> <column> ...]
```

Furthermore, you can specify the columns and their types of your model which generates the required model code and the respective factory too. The code uses a simple system to guess the sort of faker that should be used for the factory, and so the naming of the columns will alter the outcome.

Each column definition should be specified in the following format:

```
<column> ::= <name>[:<type>[,<length>][:<modifier>,<modifier>...]]
```

- **Name**: The name of the column in lowercase underscore format
- **Type (optional; defaults to string)**: The data type of the column
  - integer (or int)
  - decimal
  - float
  - boolean (or bool)
  - date
  - time
  - datetime
  - binary (or bin)
  - string (or str)
  - text
- **Length (optional; only applies to string, text & binary)**: The required length of the column
- **Modifiers (optional)**: Any modifiers relating to the column
  - index
  - primary
  - required
  - unique

Note: If no primary key is specified, a primary key **integer** column named **id** will be created for you.

Let's go through an example:

```
flaskage generate model user name:string,100 email:string:index created:datetime
```

This would generate the following model:

```
class User(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100))
    email = db.Column(db.String, index=True)
    created = db.Column(db.DateTime)
```

And one more example:

```
flaskage generate model person email::primary name::index dob:datetime:required
```

Notice that I haven't specified **string** explicitly above. Leaving the type blank assumes the default type but allows you to then specify modifiers.

This would generate the following model:

```
class Person(db.Model, CRUDMixin):
    email = db.Column(db.String, primary_key=True)
    name = db.Column(db.String, index=True)
    dob = db.Column(db.DateTime, nullable=False)
```

And this time, let's also check out the factory that was generated for us:

```
class PersonFactory(SQLAlchemyModelFactory):
    FACTORY_FOR = Person
    FACTORY_SESSION = db.session

    email = factory.LazyAttribute(lambda a: fake.email())
    name = factory.LazyAttribute(lambda a: fake.name())
    dob = factory.LazyAttribute(lambda a: fake.date_time())
```

Notice how Flaskage chose the correct faker for each column here!

To generate a new library:

```
flaskage generate lib <library_name>
```

Note: When using the flaskage command, you need not type the full command in order to execute it. In almost all cases, simply typing the first letter of the command will suffice.

For example, the following are equivalent:

```
flaskage generate blueprint <blueprint_name>
flaskage g blueprint <blueprint_name>
flaskage g b <blueprint_name>
```

1.4 Configuring Your Project

Flaskage makes it easy to maintain a hierarchical configuration that can work in each of your environments. The base class named **Config** is common across all environments. Classes which inherit from this object can then define certain

customisations which relate to each specific site.

1.4.1 Generating Secret Keys

Open **config.py** and start by generating some good secret keys for each of your environments. To do this, simply open a Python shell and run the following several times to generate multiple secret keys.

```
>>> import os
>>> os.urandom(24)
```

Place the contents of each randomly generated key in the **SECRET_KEY** config parameter for each environment.

1.4.2 Database Configuration

Next up, configure your database via the **SQLALCHEMY_DATABASE_URI** variable. Please find several examples below:

For **PostgreSQL**:

```
SQLALCHEMY_DATABASE_URI = 'postgresql://user:pass@host/dbname'
```

You'll need to install the psycopg2 package to enable this database.

```
pip install psycopg2
```

Note: In general, PostgreSQL provides superior performance and features to MySQL and is therefore the preferred database for use with Flaskage.

For **MySQL**:

```
SQLALCHEMY_DATABASE_URI = 'mysql://user:pass@host/dbname'
```

You'll need to install the MySQL-python package to enable this database.

```
pip install MySQL-python
```

For **SQLite**:

```
SQLALCHEMY_DATABASE_URI = (
    'sqlite:///' + os.path.join(Config.PROJECT_ROOT, 'dbname.db')
)
```

Please ensure that you add any newly installed package in your **requirements.txt** file. You may list your installed packages as follows:

```
pip freeze
```

Important: It is highly recommended you setup multiple databases on your chosen database engine and use it to power all environments. For example, if you choose to use PostgreSQL, then also install it on your dev server and setup databases for both dev and test. This ensures that your will seamlessly work in production as each database has its own quirks.

1.4.3 Managing Configuration Environments

As you'll notice, by default Flaskage provides a production, development and testing configuration out of the box.

If you would like to add an additional configuration environment, simply create a new class which inherits from **Config** similar to that below:

```
class StagingConfig(Config):
    SQLALCHEMY_DATABASE_URI = 'postgresql://flaskage:pass123@localhost/stagingdb'
```

You'll then need to add the config to the **AVAILABLE_CONFIGS** global dict so that it may be used.

```
AVAILABLE_CONFIGS = {
    'production': 'config.ProductionConfig',
    'development': 'config.DevelopmentConfig',
    'testing': 'config.TestingConfig',
    'staging': 'config.StagingConfig'
}
```

While running the development server, a default configuration is loaded if not specified by the user. You may change the default config by updating the **DEFAULT_CONFIG** global variable as shown below:

```
DEFAULT_CONFIG = 'staging'
```

1.5 Troubleshooting

Flaskage comes with several simple helpers for troubleshooting as you develop your application.

1.5.1 The Shell

A pre-configured shell using the incredible ipython may be invoked using the following command:

```
./manage.py shell
```

Within this shell, several variables will automatically be made available:

- The Flask application object (app)
- The Flask-SQLAlchemy database object (db)
- All model classes (e.g. User, Post .etc)
- All factory classes (e.g. UserFactory, PostFactory .etc)

The shell is a great place to try out queries or troubleshoot your application.

1.5.2 The Python Debugger

TODO

1.5.3 URL Mappings

Unlike frameworks like Django and Rails, Flask allows you to configure your routing right next to each view which is the preferred way of working in Flaskage.

However, if you ever wish to see a full mapping of all URLs and their associated view function, you may run:

```
./manage.py urls
```

1.5.4 Switching Environments

Any command that you invoke via manage.py will be run against the development environment by default. However, you may easily override this using the **-c** option as shown below:

```
./manage.py -c production shell
```

This example will run the shell using the production environment.

1.6 Database Design

Flaskage leverages the extremely powerful SQLAlchemy ORM for database interaction. SQLAlchemy can indeed by a little overwhelming when starting out, but it's worth it in the long run.

In most cases, it is suggested that you use scaffolding to generate your models so that they are created in the correct location and with the correct Flaskage naming conventions which have been carefully thought out. However, this page goes through all details of manual model design so that you can understand how everything works.

1.6.1 The CRUD Mixin

Unlike Django's ORM, SQLAlchemy has a slightly lower level API whereby updating records requires them to be added to a session and then the session committed.

Enter the **CRUDMixin** which provides some handy helpers that you can use on all your models. These include create, update, save and delete methods.

We'll be using the CRUDMixin in the examples that follow but use of this base class is entirely optional.

1.6.2 Model Definitions

It is recommended that you create one file per model in the models directory named the same name as the model in lowercase underscore notation.

All model class names and their related filename should be **singular**. For example, the model class named **BlogPost** should be in a file named **blog_post.py**.

So let's look at a basic model definition:

```
from datetime import datetime

from .. import db
from . import CRUDMixin

class User(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), nullable=False, unique=True)
    email = db.Column(db.String(100), unique=True)
    creation_date = db.Column(db.DateTime, default=datetime.utcnow)
```

So here, we demonstrate the main directives used when defining models using SQLAlechmy with Flask.

Note:

- When defining models with SQLAlchemy, you must explicitly create a primary key for each table (which usually will be named id and similar to the example above).

- In SQLAlchemy, columns are nullable by default which means that null values are allowed. If you wish to specify that the column must include a value at all times, then be sure to set nullable to False as we have here with the username.
- The default keyword argument takes in a callable function which generates the default value for a column if not specified by the user.

Once you have defined your models, you must register them in **app/models/__init__.py** by importing them in the following manner:

```
from .user import User  # noqa
```

This enables database migrations for the new model and makes it easier to import in views. The **noqa** directive in the comments tells Flake8 to ignore the import as it would otherwise raise a warning stating that the model was unused in that file.

Please see SQLAlchemy Declarative Documentation for more information.

Let's go through other column types that you'll commonly use in models:

```
class MyModel(db.Model, CRUDMixin):
    boolean_column = db.Column(db.Boolean)
    decimal_column = db.Column(db.Numeric)
    float_column = db.Column(db.Float)
    integer_column = db.Column(db.Integer)
    text_column = db.Column(db.Text)
```

Please see SQLAlchemy Column and Data Types Documentation for more information.

1.6.3 Relationship Definitions

Note: At the current time, Flaskage scaffolding doesn't support creation of relationships, so the foreign key fields and relationships will need to be added manually after you generate the model with all other fields.

One to many relationships may be defined as follows:

```
# Category table (in app/models/category.py)
class Category(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50))

# Post table (in app/models/post.py)
class Post(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80))
    body = db.Column(db.Text)
    pub_date = db.Column(db.DateTime, default=datetime.utcnow)

    # One to many relationship
    category_id = db.Column(db.Integer, db.ForeignKey('category.id'))
    category = db.relationship('Category', backref=db.backref('posts'))
```

The **backref** property specifies the member variable that will be used to access the related posts when working with a Category object.

For example:

```
chosen_category = Category.get_by_id(5)
posts_in_category = chosen_category.posts
```

The relationship may also be specified on the other end if you like:

```
# Category table (in app/models/category.py)
class Category(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50))
    posts = db.relationship('Post', backref=db.backref('category'))

# Post table (in app/models/post.py)
class Post(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(80))
    body = db.Column(db.Text)
    pub_date = db.Column(db.DateTime, default=datetime.utcnow)

    # One to many relationship
    category_id = db.Column(db.Integer, db.ForeignKey('category.id'))
```

Many to many relationships may be defined as follows:

```
# User table (in app/models/user.py)
class User(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), nullable=False, unique=True)
    email = db.Column(db.String(100), unique=True)

# Relationship table (in app/models/relationships.py)
users_posts = db.Table(
    'users_posts',
    db.Column('user_id', db.Integer, db.ForeignKey('user.id')),
    db.Column('post_id', db.Integer, db.ForeignKey('post.id')))

# Post table (in app/models/post.py)
class Post(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
    ...
    # Many to many relationship
    users = db.relationship(
        'User', secondary=users_posts, backref=db.backref('posts')
    )
```

All many to many relationship tables should be placed in the file **relationships.py** under the **app/models** directory.

One to one relationships are achieved using the **uselist** flag as shown below:

```
class User(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
    profile_id = db.Column(db.Integer, db.ForeignKey('profile.id'))
    profile = db.relationship('Profile', db.backref=('user', uselist=False))

class Profile(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
```

Alternatively, the relationship may be reversed:

```
class User(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
```

```
    profile = db.relationship('Profile', uselist=False, backref='user')

class Profile(db.Model, CRUDMixin):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
```

Please see SQLAlchemy Relationship Configuration Documentation for more information.

1.6.4 Database Migrations

Thanks to alembic and the Flask-Migrate extension, Flaskage implements the ability to easily alter the database schema as the application evolves over time.

Each time you update or add models, you can generate a new migration by running the following in the root directory of your project:

```
./manage.py db migrate
```

This introspects the database and generates a new migration which will be placed in the **db/migrations** directory. Carefully review the migration to verify that it is correct and then update your database as follows:

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

Changes may be undone by downgrading the migration:

```
./manage.py db downgrade
```

1.7 Database Queries

Now that we've defined our models and used migrations to build our schema, we're ready to start manipulating our data.

1.7.1 CRUD Operations

In case you're unaware, CRUD stands for **create, read, update and delete** which are the common actions one would take with their models.

For any of these operations, be sure to first import the models you require:

```
from models import User
```

To **create** a new record for a model:

```
me = User(username='fgimian', name='Fotis')
me.save()
```

The auto-generated id will now be avalaible as **me.id**.

To **read** a record using the model's primary key:

```
the_user = User.query.get(5)
```

Further examples of read operations (select queries) and their respective SQL will follow below.

To **update** a record for a model:

```
the_user = User.query.get(5)
the_user.update(username='newusername')
```

And finally, to **delete** a record for a model:

```
the_user = User.query.get(5)
the_user.delete()
```

1.7.2 Select Queries

If you're like me and love your SQL, you'll understand how frustrating it can be to use a limited ORM. Luckily for us, we're using one of the most powerful ORMs out there and it can do just about anything that raw SQL can do with its expressive query API.

Selecting all rows in a table

```
SELECT *
FROM user
```

translates to:

```
users = User.query.all()
```

Basic filtering

```
SELECT *
FROM user
WHERE age = 18
```

translates to:

```
users = User.query.filter_by(age = 18).all()
```

You may also filter using other standard <, >, != operators:

```
SELECT *
FROM user
WHERE age < 18
AND age >= 5
```

translates to:

```
users = User.query.filter((User.age < 18) & (User.age >= 5)).all()
```

Obtaining a single record

```
SELECT *
FROM user
WHERE username = 'fgimian'
LIMIT 1
```

translates to:

```
me = User.query.filter_by(username = 'fgimian').first()
```

The **first** function will return **None** if no records were found. In the case that multiple records are found matching your filter, only the first row will be returned.

To ensure that only a single result is present

```
SELECT *
FROM user
WHERE username = 'fgimian'
```

translates to:

```
me = User.query.filter_by(username = 'fgimian').one()
```

The **one** function will return a single row, however it will raise a **sqlalchemy.orm.exc.NoResultFound** exception if no records were found or a **sqlalchemy.orm.exc.MultipleResultsFound** exception if multiple records were found.

Filtering using an in statement:

```
SELECT *
FROM user
WHERE username IN ('fgimian', 'lonelycat')
```

translates to:

```
users = User.query.filter(User.username.in_(['fgimian', 'lonelycat'])).all()
```

Selecting particular columns only

```
SELECT username, email
FROM user
```

translates to:

```
users = User.query.with_entities(User.username, User.email).all()
```

Note: Using the **with_entitities** function returns a list of tuples instead of a list of objects as per the other queries.

Logical operators for use with filtering:

```
SELECT *
FROM user
WHERE (username = 'fgimian'
       OR username = 'lonelycat')
AND id < 3
```

```
users = User.query.filter(
    (User.username == 'fgimian') | (User.username == 'lonelycat') &
    (User.id < 3)
).all()
```

Applying functions to columns when selecting

```
SELECT upper(username)
FROM user
```

```
users = User.query.with_entities(db.func.upper(User.username)).all()
```

Ordering results

```
SELECT *
FROM user
ORDER BY age [DESC]
```

translates to:

```
users = User.query.order_by(User.age)
users_desc = User.query.order_by(User.age.desc())
```

Grouping by

```
SELECT age, count(*)
FROM user
GROUP BY age
HAVING count(*) > 5
```

translates to:

```
users = User.query.with_entities(
    User.age, db.func.count()
).group_by(User.age).having(db.func.count() > 5).all()
```

Aliasing columns

```
SELECT age, count(*) AS counter
FROM user
GROUP BY age
HAVING counter > 5
```

translates to:

```
users = User.query.with_entities(
    User.age, db.func.count().label('counter')
).group_by(User.age).having('counter > 5').all()
```

Sub-queries, ranking and extracting date components

```
SELECT name, used, date
FROM (SELECT name, value, date,
             rank() OVER (PARTITION BY name, EXTRACT(YEAR FROM date)
                          ORDER BY date DESC) AS ranking
      FROM bandwidth_usage) AS bwr
WHERE ranking = 1
```

translates to:

```
used_bw_subquery = BandwidthUsage.query.add_columns(
    db.func.rank().over(
        partition_by=[BandwidthUsage.name,
                      db.extract('year', BandwidthUsage.date)],
        order_by=BandwidthUsage.date.desc()
    ).label('ranking')
).subquery('bwr')

used_bw = BandwidthUsage.query.select_entity_from(used_bw_subquery).filter(
    bw_subquery.c.ranking == 1
)
```

Tip: When constructing such complex queries containing sub-quries, it's best to tackle the inner most query first and work your way outwards until you reach the main query.

1.8 Unit Testing Models

Flaskage includes a variety of powerful tools for testing your models including model factories and data fakers to easily generate mock objects.

1.8.1 Model Factories

Using the excellent factory_boy library combined with the powerful fake-factory, we can generate fake model objects that can be used during testing.

A factory for each model should be defined in the a file named model_factory.py in the **tests/fixtures** directory (e.g. blog_post_factory.py).

Here's an example of a factory boy factory that uses fake factory to fake data:

```
from app import db
from app.models import User

from faker import Factory
import factory
from factory.alchemy import SQLAlchemyModelFactory

fake = Factory.create()

class UserFactory(SQLAlchemyModelFactory):
    FACTORY_FOR = User
    FACTORY_SESSION = db.session

    username = factory.LazyAttribute(lambda a: fake.user_name())
    email = factory.LazyAttribute(lambda a: fake.email())
    password = factory.PostGenerationMethodCall(
        'set_password', 'password123'
    )
```

Please see the factory_boy docs and fake-factory docs for further information and examples.