Flask has three main dependencies. The routing, debugging, and Web Server Gateway Interface (WSGI) subsystems come from [Werkzeug](http://werkzeug.pocoo.org/); the template support is provided by [Jinja2](http://jinja.pocoo.org/); and the command-line integration comes from [Click](http://click.pocoo.org). These dependencies are all authored by Armin Ronacher, the author of Flask.

When the virtual environment is activated, its python interpreter location is added to the PATH variable. When deactivated, its removed. Pip is the python package manager.

Create virtual environment: python –m venv <name>

Activate virtual environment: <name>\scripts\activate deactivate the venv: deactivate

In linux: source <name>/bin/activate

To install packages: pip install <packagename>

To list installed packaged in the environment: pip freeze

App = Flask(\_\_name\_\_) // The \_\_name\_\_ argument that is passed to the Flask application constructor is a source of confusion among new Flask developers. Flask uses this argument to determine the location of the application, which in turn allows it to locate other files that are part of the application, such as images and templates.

The association between a URL and the function that handles it is called a route.

A common use of decorators is to register functions as handler functions to be invoked when certain events occur.

Flask also offers a more traditional way to set up the application routes with the app.add\_url\_rule() method, which in its most basic form takes three arguments: the URL, the endpoint name, and the view function. The following example uses app.add\_url\_rule() to register an index() function that is equivalent to the one shown previously:

def index():

return '<h1>Hello World!</h1>'

app.add\_url\_rule('/', 'index', index)

Functions like index() that handle application URLs are called view functions.

The dynamic components in routes are strings by default but can also be of different types. For example, the route /user/<int:id> would match only URLs that have an integer in the id dynamic segment, such as /user/123. Flask supports the types string, int, float, and path for routes. The path type is a special string type that can include forward slashes, unlike the string type.

Flask applications include a development web server that can be started with the flask run command. This command looks for the name of the Python script that contains the application instance in the FLASK\_APP environment variable.

(venv) $ **set FLASK\_APP=hello.py**

(venv) $ **flask run**

\* Serving Flask app "hello"

\* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

Flask applications can optionally be executed in debug mode. In this mode, two very convenient modules of the development server called the reloader and the debugger are enabled by default.

The --host argument is particularly useful because it tells the web server what network interface to listen to for connections from clients. By default, Flask’s development web server listens for connections on localhost, so only connections originating from the computer running the server are accepted. The following command makes the web server listen for connections on the public network interface, enabling other computers in the same network to connect as well:

(venv) $ **flask run --host 0.0.0.0**

\* Serving Flask app "hello"

\* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

There are two contexts in Flask: the application context and the request context.

| **Variable name** | **Context** | **Description** |
| --- | --- | --- |
| current\_app | Application context | The application instance for the active application. |
| g | Application context | An object that the application can use for temporary storage during the handling of a request. This variable is reset with each request. |
| request | Request context | The request object, which encapsulates the contents of an HTTP request sent by the client. |
| session | Request context | The user session, a dictionary that the application can use to store values that are “remembered” between requests. |

>>> **app\_ctx = app.app\_context()**

>>> **app\_ctx.push()**

>>> **current\_app.name**

'hello'

>>> **app\_ctx.pop()**

---

>>> **app.url\_map**

The / and /user/<name> routes were defined by the app.route decorators in the application. The /static/<filename> route is a special route added by Flask to give access to static files.

| Table 2-2. Flask request object | |
| --- | --- |
| **Attribute or Method** | **Description** |
| form | A dictionary with all the form fields submitted with the request. |
| args | A dictionary with all the arguments passed in the query string of the URL. |
| values | A dictionary that combines the values in form and args. |
| cookies | A dictionary with all the cookies included in the request. |
| headers | A dictionary with all the HTTP headers included in the request. |
| files | A dictionary with all the file uploads included with the request. |
| get\_data() | Returns the buffered data from the request body. |
| get\_json() | Returns a Python dictionary with the parsed JSON included in the body of the request. |
| blueprint | The name of the Flask blueprint that is handling the request. |
| endpoint | The name of the Flask endpoint that is handling the request. Flask uses the name of the view function as the endpoint name for a route. |
| method | The HTTP request method, such as GET or POST. |
| scheme | The URL scheme (http or https). |
| is\_secure() | Returns True if the request came through a secure (HTTPS) connection. |
| host | The host defined in the request, including the port number if given by the client. |
| path | The path portion of the URL. |
| query\_string | The query string portion of the URL, as a raw binary value. |
| full\_path | The path and query string portions of the URL. |
| url | The complete URL requested by the client. |
| base\_url | Same as url, but without the query string component. |
| remote\_addr | The IP address of the client. |
| Environ | The raw WSGI environment dictionary for the request. |

Request hooks are implemented as decorators. These are the four hooks supported by Flask:

before\_request

Registers a function to run before each request.

before\_first\_request

Registers a function to run only before the first request is handled. This can be a convenient way to add server initialization tasks.

after\_request

Registers a function to run after each request, but only if no unhandled exceptions occurred.

teardown\_request

Registers a function to run after each request, even if unhandled exceptions occurred.

The make\_response() function takes one, two, or three arguments, the same values that can be returned from a view function, and returns an equivalent response object.

| Table 2-3. Flask response object | |
| --- | --- |
| **Attribute or Method** | **Description** |
| status\_code | The numeric HTTP status code |
| headers | A dictionary-like object with all the headers that will be sent with the response |
| set\_cookie() | Adds a cookie to the response |
| delete\_cookie() | Removes a cookie |
| content\_length | The length of the response body |
| content\_type | The media type of the response body |
| set\_data() | Sets the response body as a string or bytes value |
| get\_data() | Gets the response body |

There is a special type of response called a redirect. This response does not include a page document; it just gives the browser a new URL to navigate to.

Another special response is issued with the abort() function, which is used for error handling. The following example returns status code 404 if the id dynamic argument given in the URL does not represent a valid user:

from flask import abort

@app.route('/user/<id>')

def get\_user(id):

user = load\_user(id)

if not user:

abort(404)

return '<h1>Hello, {}</h1>'.format(user.name)

Note that abort() does not return control back to the function because it raises an exception.

Flask is designed to be extended.

These two types of tasks are formally called business logic and presentation logic, of view functions. Moving the presentation logic into templates helps improve the maintainability of the application.

The process that replaces the variables with actual values and returns a final response string is called rendering. For the task of rendering templates, Flask uses a powerful template engine called Jinja2.

The function render\_template() provided by Flask integrates the Jinja2 template engine with the application. This function takes the filename of the template as its first argument. Any additional arguments are key-value pairs that represent actual values for variables referenced in the template.

Jinja2 recognizes variables of any type, even complex types such as lists, dictionaries, and objects.

Variables can be modified with filters, which are added after the variable name with a pipe character as separator.

Hello, {{ name|capitalize }}

Jinja2 also supports macros, which are similar to functions in Python code.

Jinja2 also supports macros, which are similar to functions in Python code. For example:

{% macro render\_comment(comment) %}

<li>{{ comment }}</li>

{% endmacro %}

<ul>

{% for comment in comments %}

{{ render\_comment(comment) }}

{% endfor %}

</ul>

To make macros more reusable, they can be stored in standalone files that are then imported from all the templates that need them:

{% import 'macros.html' as macros %}

<ul>

{% for comment in comments %}

{{ macros.render\_comment(comment) }}

{% endfor %}

</ul>

Portions of template code that need to be repeated in several places can be stored in a separate file and included from all the templates to avoid repetition:

{% include 'common.html' %}

[Bootstrap](http://getbootstrap.com) is an open-source web browser framework from Twitter that provides user interface components that help create clean and attractive web pages that are compatible with all modern web browsers used on desktop and mobile platforms. Bootstrap is a client-side framework.

Flask extensions are initialized at the same time the application instance is created.

| **Block name** | **Description** |
| --- | --- |
| doc | The entire HTML document |
| html\_attribs | Attributes inside the <html> tag |
| html | The contents of the <html> tag |
| head | The contents of the <head> tag |
| title | The contents of the <title> tag |
| metas | The list of <meta> tags |
| styles | CSS definitions |
| body\_attribs | Attributes inside the <body> tag |
| body | The contents of the <body> tag |
| navbar | User-defined navigation bar |
| content | User-defined page content |
| scripts | JavaScript declarations at the bottom of the document |

The two most common error codes are 404, triggered when the client requests a page or route that is not known, and 500, triggered when there is an unhandled exception in the application.

@app.errorhandler(404)

def page\_not\_found(e):

return render\_template('404.html'), 404

@app.errorhandler(500)

def internal\_server\_error(e):

return render\_template('500.html'), 500

Flask provides the url\_for() helper function, which generates URLs from the information stored in the application’s URL map.

Calling url\_for('index', \_external=True) would instead return an absolute URL, which in this example is <http://localhost:5000/>.

Relative URLs are sufficient when generating links that connect the different routes of the application. Absolute URLs are necessary only for links that will be used outside of the web browser, such as when sending links by email.

Flask-Moment for localization of dates and time.

There is an excellent open source library written in JavaScript that renders dates and times in the browser called Moment.js. Flask-Moment is an extension for Flask applications that makes the integration of Moment.js into Jinja2 templates very easy.

{% block scripts %}

{{ super() }}

{{ moment.include\_moment() }}

{{ moment.locale('es') }} # TO SET LANGUAGE TO SPANISH

{% endblock %}

Flask-Moment implements the format(), fromNow(), fromTime(), calendar(), valueOf(), and unix() methods from Moment.js. Consult the Moment.js documentation to learn about all the formatting options offered by this library.

Web Forms

The Flask-WTF extension makes working with web forms a much more pleasant experience. This extension is a Flask integration wrapper around the framework-agnostic WTForms package.

Unlike most other extensions, Flask-WTF does not need to be initialized at the application level, but it expects the application to have a secret key configured. A secret key is a string with any random and unique content that is used as an encryption or signing key to improve the security of the application in several ways. Flask uses this key to protect the contents of the user session against tampering.

Flask-WTF requires a secret key to be configured in the application because this key is part of the mechanism the extension uses to protect all forms against cross-site request forgery (CSRF) attacks. A CSRF attack occurs when a malicious website sends requests to the application server on which the user is currently logged in. Flask-WTF generates security tokens for all forms and stores them in the user session, which is protected with a cryptographic signature generated from the secret key.

Form Classes

When using Flask-WTF, each web form is represented in the server by a class that inherits from the class FlaskForm. The class defines the list of fields in the form, each represented by an object. Each field object can have one or more *validators* attached. A validator is a function that checks whether the data submitted by the user is valid.

| Table 4-1. WTForms standard HTML fields | |
| --- | --- |
| **Field type** | **Description** |
| BooleanField | Checkbox with True and False values |
| DateField | Text field that accepts a datetime.date value in a given format |
| DateTimeField | Text field that accepts a datetime.datetime value in a given format |
| DecimalField | Text field that accepts a decimal.Decimal value |
| FileField | File upload field |
| HiddenField | Hidden text field |
| MultipleFileField | Multiple file upload field |
| FieldList | List of fields of a given type |
| FloatField | Text field that accepts a floating-point value |
| FormField | Form embedded as a field in a container form |
| IntegerField | Text field that accepts an integer value |
| PasswordField | Password text field |
| RadioField | List of radio buttons |
| SelectField | Drop-down list of choices |
| SelectMultipleField | Drop-down list of choices with multiple selection |
| SubmitField | Form submission button |
| StringField | Text field |
| TextAreaField | Multiple-line text field |

The list of WTForms built-in validators is shown in [Table 4-2](https://learning.oreilly.com/library/view/flask-web-development/9781491991725/ch04.html#ch01_wtforms_validators).

| Table 4-2. WTForms validators | |
| --- | --- |
| **Validator** | **Description** |
| DataRequired | Validates that the field contains data after type conversion |
| Email | Validates an email address |
| EqualTo | Compares the values of two fields; useful when requesting a password to be entered twice for confirmation |
| InputRequired | Validates that the field contains data before type conversion |
| IPAddress | Validates an IPv4 network address |
| Length | Validates the length of the string entered |
| MacAddress | Validates a MAC address |
| NumberRange | Validates that the value entered is within a numeric range |
| Optional | Allows empty input in the field, skipping additional validators |
| Regexp | Validates the input against a regular expression |
| URL | Validates a URL |
| UUID | Validates a UUID |
| AnyOf | Validates that the input is one of a list of possible values |
| NoneOf | Validates that the input is none of a list of possible values |

The Flask-Bootstrap extension provides a high-level helper function that renders an entire Flask-WTF form using Bootstrap’s predefined form styles, all with a single call. Using Flask-Bootstrap, the previous form can be rendered as follows:

{% import "bootstrap/wtf.html" as wtf %}

{{ wtf.quick\_form(form) }}

The validate\_on\_submit() method of the form returns True when the form was submitted and the data was accepted by all the field validators.

Post/Redirect/Get pattern

Applications can “remember” things from one request to the next by storing them in the user session, a private storage that is available to each connected client.

By default, user sessions are stored in client-side cookies that are cryptographically signed using the configured secret key. Any tampering with the cookie content would render the signature invalid, thus invalidating the session.

The first and only required argument to url\_for() is the endpoint name, the internal name each route has. By default, the endpoint of a route is the name of the view function attached to it.

Flask makes a get\_flashed\_messages() function available to templates to retrieve the messages and render them, as shown in

Calling flash() is not enough to get messages displayed; the templates used by the application need to render these messages. The best place to render flashed messages is the base template, because that will enable these messages in all pages.

##### Example 4-7. templates/base.html: rendering of flashed messages

{% block content %}

<div class="container">

{% for message in get\_flashed\_messages() %}

<div class="alert alert-warning">

<button type="button" class="close" data-dismiss="alert">&times;</button>

{{ message }}

</div>

{% endfor %}

Databases:

The most commonly used databases for web applications are those based on the relational model, also called SQL databases in reference to the Structured Query Language they use. But in recent years document-oriented and key-value databases, informally known together as NoSQL databases, have become popular alternatives.

The graphical style of representing the structure of a database is called an entity-relationship diagram

There are also a number of database abstraction layer packages, such as SQLAlchemy or MongoEngine, that allow you to work at a higher level with regular Python objects instead of database entities such as tables, documents, or query languages.

Abstraction layers, also called object-relational mappers (ORMs) or object-document mappers (ODMs), provide transparent conversion of high-level object-oriented operations into low-level database instructions.

SQLAlchemy is a powerful relational database framework that supports several database backends. It offers a high-level ORM and low-level access to the database’s native SQL functionality.

| Table 5-1. Flask-SQLAlchemy database URLs | |
| --- | --- |
| **Database engine** | **URL** |
| MySQL | *mysql://username:password@hostname/database* |
| Postgres | *postgresql://username:password@hostname/database* |
| SQLite (Linux, macOS) | *sqlite:////absolute/path/to/database* |
| SQLite (Windows) | *sqlite:///c:/absolute/path/to/database* |

SQLite databases do not have a server, so *hostname*, *username*, and *password* are omitted and *database* is the filename on disk for the database.

##### Example 5-1. hello.py: database configuration

import os

from flask\_sqlalchemy import SQLAlchemy

basedir = os.path.abspath(os.path.dirname(\_\_file\_\_))

app = Flask(\_\_name\_\_)

app.config['SQLALCHEMY\_DATABASE\_URI'] =\

'sqlite:///' + os.path.join(basedir, 'data.sqlite')

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

db = SQLAlchemy(app)

| Table 5-2. Most common SQLAlchemy column types | | |
| --- | --- | --- |
| **Type name** | **Python type** | **Description** |
| Integer | int | Regular integer, typically 32 bits |
| SmallInteger | int | Short-range integer, typically 16 bits |
| BigInteger | int or long | Unlimited precision integer |
| Float | float | Floating-point number |
| Numeric | decimal.Decimal | Fixed-point number |
| String | str | Variable-length string |
| Text | str | Variable-length string, optimized for large or unbounded length |
| Unicode | unicode | Variable-length Unicode string |
| UnicodeText | unicode | Variable-length Unicode string, optimized for large or unbounded length |
| Boolean | bool | Boolean value |
| Date | datetime.date | Date value |
| Time | datetime.time | Time value |
| DateTime | datetime.datetime | Date and time value |
| Interval | datetime.timedelta | Time interval |
| Enum | str | List of string values |
| PickleType | Any Python object | Automatic Pickle serialization |
| LargeBinary | str | Binary blob |

The remaining arguments to db.Column specify configuration options for each attribute. [Table 5-3](https://learning.oreilly.com/library/view/flask-web-development/9781491991725/ch05.html#ch02_column_options) lists some of the options available.

| Table 5-3. Most common SQLAlchemy column options | |
| --- | --- |
| **Option name** | **Description** |
| primary\_key | If set to True, the column is the table’s primary key. |
| unique | If set to True, do not allow duplicate values for this column. |
| index | If set to True, create an index for this column, so that queries are more efficient. |
| nullable | If set to True, allow empty values for this column. If set to False, the column will not allow null values. |
| default | Define a default value for the column. |

Flask-SQLAlchemy requires all models to define a primary key column, which is commonly named id.

| Table 5-4. Common SQLAlchemy relationship options | |
| --- | --- |
| **Option name** | **Description** |
| backref | Add a back reference in the other model in the relationship. |
| primaryjoin | Specify the join condition between the two models explicitly. This is necessary only for ambiguous relationships. |
| lazy | Specify how the related items are to be loaded. Possible values are select (items are loaded on demand the first time they are accessed), immediate (items are loaded when the source object is loaded), joined (items are loaded immediately, but as a join), subquery (items are loaded immediately, but as a subquery), noload (items are never loaded), and dynamic (instead of loading the items, the query that can load them is given). |
| uselist | If set to False, use a scalar instead of a list. A **scalar** value refers to a single value |
| order\_by | Specify the ordering used for the items in the relationship. |
| secondary | Specify the name of the association table to use in many-to-many relationships. |
| secondaryjoin | Specify the secondary join condition for many-to-many relationships when SQLAlchemy cannot determine it on its own. |
|  |  |

Flask shell. The db.create\_all() function locates all the subclasses of db.Model and creates corresponding tables in the database for them

The brute-force solution to update existing database tables to a different schema is to remove the old tables first:

>>> **db.drop\_all()**

>>> **db.create\_all()**

Changes to the database are managed through a database *session*, which Flask-SQLAlchemy provides as db.session. To prepare objects to be written to the database, they must be added to the session:

>>> **db.session.add(admin\_role) // or**

>>> **db.session.add\_all([admin\_role, mod\_role, user\_role,**

... **user\_john, user\_susan, user\_david])**

To write the objects to the database, the session needs to be *committed* by calling its commit() method:

>>> **db.session.commit()**

The db.session database session is not related to the Flask session object. Database sessions are also called transactions.

A database session can also be *rolled back*. If db.session.rollback() is called, any objects that were added to the database session are restored to the state they have in the database.

It is also possible to inspect the native SQL query that SQLAlchemy generates for a given query by converting the query object to a string:

>>> **str(User.query.filter\_by(role=user\_role))**

| Table 5-5. Common SQLAlchemy query filters | | | |
| --- | --- | --- | --- |
| **Option** | **Description** | | |
| filter() | Returns a new query that adds an additional filter to the original query | | |
| filter\_by() | Returns a new query that adds an additional equality filter to the original query | | |
| limit() | Returns a new query that limits the number of results of the original query to the given number | | |
| offset() | Returns a new query that applies an offset into the list of results of the original query | | |
| order\_by() | Returns a new query that sorts the results of the original query according to the given criteria | | |
| group\_by() | Returns a new query that groups the results of the original query according to the given criteria | | |
| Table 5-6. Most common SQLAlchemy query executors | | |
| **Option** | | **Description** |
| all() | | Returns all the results of a query as a list |
| first() | | Returns the first result of a query, or None if there are no results |
| first\_or\_404() | | Returns the first result of a query, or aborts the request and sends a 404 error as the response if there are no results |
| get() | | Returns the row that matches the given primary key, or None if no matching row is found |
| get\_or\_404() | | Returns the row that matches the given primary key or, if the key is not found, aborts the request and sends a 404 error as the response |
| count() | | Returns the result count of the query |
| paginate() | | Returns a Pagination object that contains the specified range of results |

To add objects to the import list, a shell context processor must be created and registered with the app.shell\_context\_processor decorator.

Adding a shell context

@app.shell\_context\_processor

def make\_shell\_context():

return dict(db=db, User=User, Role=Role)

The shell context processor function returns a dictionary that includes the database instance and the models. The flask shell command will import these items automatically into the shell, in addition to app, which is imported by default

Flask-Migrate:

Database migration framework. In the same way source code version control tools keep track of changes to source code files, a database migration framework keeps track of changes to a database schema, allowing incremental changes to be applied.

The developer of SQLAlchemy has written a migration framework called [Alembic](http://bit.ly/alembic-doc), but instead of using Alembic directly, Flask applications can use the [Flask-Migrate](http://bit.ly/fl-migrate) extension, a lightweight Alembic wrapper that integrates it with the flask command.

When you work on a new project, you can add support for database migrations with the init subcommand:(venv) $ flask db init

In Alembic, a database migration is represented by a migration script. This script has two functions called upgrade() and downgrade(). The upgrade() function applies the database changes that are part of the migration, and the downgrade() function removes them. This ability to add and remove changes means, Alembic can reconfigure a database to any point in the change history.Alembic migrations can be created manually or automatically using the revision and migrate commands, respectively. A manual migration creates a migration skeleton script with empty upgrade() and downgrade() functions that need to be implemented by the developer using directives exposed by Alembic’s Operations object. An automatic migration attempts to generate the code for the upgrade() and downgrade() functions by looking for differences between the model definitions and the current state of the database.

Flask db migrate // flask db upgrade The flask db migrate subcommand creates an automatic migration script:

Another option is to skip the flask db upgrade and instead mark the existing database as upgraded using the flask db stamp command.

**pip install flask-mail**

The extension connects to a Simple Mail Transfer Protocol (SMTP) server and passes emails to it for delivery. If no configuration is given, Flask-Mail connects to *localhost* at port 25 and sends email without authentication.

| Table 6-1. Flask-Mail SMTP server configuration keys | | |
| --- | --- | --- |
| **Key** | **Default** | **Description** |
| MAIL\_SERVER | *localhost* | Hostname or IP address of the email server |
| MAIL\_PORT | 25 | Port of the email server |
| MAIL\_USE\_TLS | False | Enable Transport Layer Security (TLS) security |
| MAIL\_USE\_SSL | False | Enable Secure Sockets Layer (SSL) security |
| MAIL\_USERNAME | None | Mail account username |
| MAIL\_PASSWORD | None | Mail account password |

Large application structure:

Unlike most other web frameworks, Flask does not impose a specific organization for large projects; the way to structure the application is left entirely to the developer.

##### Basic multiple-file Flask application structure

|-flasky

|-app/

|-templates/

|-static/

|-main/

|-\_\_init\_\_.py

|-errors.py

|-forms.py

|-views.py

|-\_\_init\_\_.py

|-email.py

|-models.py

|-migrations/

|-tests/

|-\_\_init\_\_.py

|-test\*.py

|-venv/

|-requirements.txt

|-config.py

|-flasky.py

##### Example 7-2. config.py: application configuration

import os

basedir = os.path.abspath(os.path.dirname(\_\_file\_\_))

class Config:

SECRET\_KEY = os.environ.get('SECRET\_KEY') or 'hard to guess string'

MAIL\_SERVER = os.environ.get('MAIL\_SERVER', 'smtp.googlemail.com')

MAIL\_PORT = int(os.environ.get('MAIL\_PORT', '587'))

MAIL\_USE\_TLS = os.environ.get('MAIL\_USE\_TLS', 'true').lower() in \

['true', 'on', '1']

MAIL\_USERNAME = os.environ.get('MAIL\_USERNAME')

MAIL\_PASSWORD = os.environ.get('MAIL\_PASSWORD')

FLASKY\_MAIL\_SUBJECT\_PREFIX = '[Flasky]'

FLASKY\_MAIL\_SENDER = 'Flasky Admin <flasky@example.com>'

FLASKY\_ADMIN = os.environ.get('FLASKY\_ADMIN')

SQLALCHEMY\_TRACK\_MODIFICATIONS = False

@staticmethod

def init\_app(app):

pass

class DevelopmentConfig(Config):

DEBUG = True

SQLALCHEMY\_DATABASE\_URI = os.environ.get('DEV\_DATABASE\_URL') or \

'sqlite:///' + os.path.join(basedir, 'data-dev.sqlite')

class TestingConfig(Config):

TESTING = True

SQLALCHEMY\_DATABASE\_URI = os.environ.get('TEST\_DATABASE\_URL') or \

'sqlite://'

class ProductionConfig(Config):

SQLALCHEMY\_DATABASE\_URI = os.environ.get('DATABASE\_URL') or \

'sqlite:///' + os.path.join(basedir, 'data.sqlite')

config = {

'development': DevelopmentConfig,

'testing': TestingConfig,

'production': ProductionConfig,

'default': DevelopmentConfig

}

The application package is where all the application code, templates, and static files live. It is called simply app here, though it can be given an application-specific name if desired. The templates and static directories are now part of the application package, so they are moved inside app. The database models and the email support functions are also moved inside this package, each in its own module, as app/models.py and app/email.py.

Unittest:

The tests are written using the standard unittest package from the Python standard library. The setUp() and tearDown() methods of the test case class run before and after each test, and any methods that have a name that begins with test\_ are executed as tests.

##### app/auth/\_\_init\_\_.py: authentication blueprint creation

from flask import Blueprint

auth = Blueprint('auth', \_\_name\_\_) // api = Blueprint('api', 'api', url\_prefix='/api') 1st argument is the blueprint name. 2nd arguments is important, blueprint package name, 3rd the blueprint api package relative path.

from . import views

---

votr = Flask(\_\_name\_\_)

votr.register\_blueprint(api)

User authentication:

Most applications need to keep track of who their users are. When users connect with an application, they authenticate with it, a process by which they make their identity known. Once the application knows who the user is, it can offer a customized experience.

This is the list of packages that will be used, and what they’re used for:

* Flask-Login: Management of user sessions for logged-in users
* Werkzeug: Password hashing and verification
* itsdangerous: Cryptographically secure token generation and verification

Werkzeug’s password hashing functions will be demonstrated. Other good choices for password hashing are [bcrypt](https://github.com/pyca/bcrypt/) and [Passlib](https://bitbucket.org/ecollins/passlib/wiki/Home).

| Flask-Login required items | |
| --- | --- |
| **Property/method** | **Description** |
| is\_authenticated | Must be True if the user has valid login credentials or False otherwise. |
| is\_active | Must be True if the user is allowed to log in or False otherwise. A False value can be used for disabled accounts. |
| is\_anonymous | Must always be False for regular users and True for a special user object that represents anonymous users. |
| get\_id() | Must return a unique identifier for the user, encoded as a Unicode string. |

These properties and methods can be implemented directly in the model class, but as an easier alternative Flask-Login provides a UserMixin class that has default implementations that are appropriate for most cases.

The login\_view attribute of the LoginManager object sets the endpoint for the login page. Flask-Login will redirect to the login page when an anonymous user tries to access a protected page.

Protecting Routes. To protect a route so that it can only be accessed by authenticated users, Flask-Login provides a login\_required decorator. An example of its usage follows:

from flask\_login import login\_required

@app.route('/secret')

@login\_required

def secret():

return 'Only authenticated users are allowed!'

The optional REMEMBER\_COOKIE\_DURATION configuration option can be used to change the default one-year duration for the remember cookie.

On a production server, the application must be made available over secure HTTP, so that login credentials and user sessions are always transmitted encrypted. Without secure HTTP, sensitive data can be intercepted during transit by an attacker.

## Understanding How Flask-Login Works

Flask-Login is a fairly small extension, but due to the many moving pieces involved in the authentication flow, Flask users often have trouble understanding how the extension works. The following is the sequence of operations that occur when a user logs in to the system:

1. The user navigates to http://localhost:5000/auth/login by clicking on the “Log In” link. The handler for this URL returns the login form template.
2. The user enters their username and password, and presses the Submit button. The same handler is invoked again, but now as a POST request instead of GET.
   1. The handler validates the credentials submitted with the form, and then invokes Flask-Login’s login\_user() function to log the user in.
   2. The login\_user() function writes the ID of the user to the user session as a string.
   3. The view function returns with a redirect to the home page.
3. The browser receives the redirect and requests the home page.
   1. The view function for the home page is invoked, and it triggers the rendering of the main Jinja2 template.
   2. During the rendering of the Jinja2 template, a reference to Flask-Login’s current\_user appears for the first time.
   3. The current\_user context variable does not have a value assigned for this request yet, so it invokes Flask-Login’s internal function \_get\_user() to find out who the user is.
   4. The \_get\_user() function checks if there is a user ID stored in the user session. If there isn’t one, it returns an instance of Flask-Login’s AnonymousUser. If there is an ID, it invokes the function that the application registered with the user\_loader decorator, with the ID as its argument.
   5. The application’s user\_loader handler reads the user from the database and returns it. Flask-Login assigns it to the current\_user context variable for the current request.
   6. The template receives the newly assigned value of current\_user.

The login\_required decorator builds on top of the current\_user context variable by only allowing the decorated view function to run when the expression current\_user.is\_authenticated is True. The logout\_user() function simply deletes the user ID from the user session.

When a form defines a method with the prefix validate\_ followed by the name of a field, the method is invoked in addition to any regularly defined validators.

Roles:

content moderators exist as well.

The difference between db.String and db.Text is that db.Text is a variable-length field and as such does not need a maximum length.

There are several Python packages that can be used to generate fake information. A fairly complete one is Faker, which is installed with pip:(venv) $ pip install faker

| Table 11-1. Flask-SQLAlchemy pagination object attributes | |
| --- | --- |
| **Attribute** | **Description** |
| items | The records in the current page |
| query | The source query that was paginated |
| page | The current page number |
| prev\_num | The previous page number |
| next\_num | The next page number |
| has\_next | True if there is a next page |
| has\_prev | True if there is a previous page |
| pages | The total number of pages for the query |
| per\_page | The number of items per page |
| total | The total number of items returned by the query |

The pagination object also has some methods, listed in [Table 11-2](https://learning.oreilly.com/library/view/flask-web-development/9781491991725/ch11.html#ch15_pagination_method_reference).

| Table 11-2. Flask-SQLAlchemy pagination object methods | |
| --- | --- |
| **Method** | **Description** |
| iter\_pages(left\_edge=2, left\_current=2, right\_current=5, right\_edge=2) | An iterator that returns the sequence of page numbers to display in a pagination widget. The list will have left\_edge pages on the left side, left\_current pages to the left of the current page, right\_current pages to the right of the current page, and right\_edge pages on the right side. For example, for page 50 of 100 this iterator configured with default values will return the following pages: 1, 2, None, 48, 49, 50, 51, 52, 53, 54, 55, None, 99, 100. A None value in the sequence indicates a gap in the sequence of pages. |
| prev() | A pagination object for the previous page. |
| next() | A pagination object for the next page |

Database relationship:

The one-to-many, many-to-one, and one-to-one relationships all have at least one side with a single entity, so the links between related records are implemented with foreign keys pointing to that one element.

Following is the code that represents the many-to-many relationship:

registrations = db.Table('registrations',

db.Column('student\_id', db.Integer, db.ForeignKey('students.id')),

db.Column('class\_id', db.Integer, db.ForeignKey('classes.id'))

)

class Student(db.Model):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String)

classes = db.relationship('Class',

secondary=registrations,

backref=db.backref('students', lazy='dynamic'),

lazy='dynamic')

class Class(db.Model):

id = db.Column(db.Integer, primary\_key=True)

name = db.Column(db.String)

A relationship in which both sides belong to the same table is said to be self-referential. In this case the entities on the left side of the relationship are users, which can be called the “followers.” The entities on the right side are also users, but these are the “followed” users. Conceptually, self-referential relationships are no different than regular relationships, but they are harder to think about.

The database operation that can do this is called a join. A join operation takes two or more tables and finds all the combinations of rows that satisfy a given condition. The resulting combined rows are inserted into a temporary table that is the result of the join.

# Basic Relationship Patterns

A quick walkthrough of the basic relational patterns.

The imports used for each of the following sections is as follows:

**from** **sqlalchemy** **import** Table, Column, Integer, ForeignKey

**from** **sqlalchemy.orm** **import** relationship

**from** **sqlalchemy.ext.declarative** **import** declarative\_base

Base = declarative\_base()

## One To Many

A one to many relationship places a foreign key on the child table referencing the parent. [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship) is then specified on the parent, as referencing a collection of items represented by the child:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer, ForeignKey('parent.id'))

To establish a bidirectional relationship in one-to-many, where the “reverse” side is a many to one, specify an additional [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship) and connect the two using the [relationship.back\_populates](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) parameter:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child", back\_populates="parent")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer, ForeignKey('parent.id'))

parent = relationship("Parent", back\_populates="children")

Child will get a parent attribute with many-to-one semantics.

Alternatively, the [relationship.backref](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) option may be used on a single [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship) instead of using [relationship.back\_populates](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship):

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child", backref="parent")

## Many To One

Many to one places a foreign key in the parent table referencing the child. [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship) is declared on the parent, where a new scalar-holding attribute will be created:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

child\_id = Column(Integer, ForeignKey('child.id'))

child = relationship("Child")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

Bidirectional behavior is achieved by adding a second [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship) and applying the [relationship.back\_populates](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) parameter in both directions:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

child\_id = Column(Integer, ForeignKey('child.id'))

child = relationship("Child", back\_populates="parents")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

parents = relationship("Parent", back\_populates="child")

Alternatively, the [relationship.backref](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) parameter may be applied to a single [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship), such as Parent.child:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

child\_id = Column(Integer, ForeignKey('child.id'))

child = relationship("Child", backref="parents")

## One To One

One To One is essentially a bidirectional relationship with a scalar attribute on both sides. To achieve this, the [relationship.uselist](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.uselist" \o "sqlalchemy.orm.relationship) flag indicates the placement of a scalar attribute instead of a collection on the “many” side of the relationship. To convert one-to-many into one-to-one:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

child = relationship("Child", uselist=**False**, back\_populates="parent")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

parent\_id = Column(Integer, ForeignKey('parent.id'))

parent = relationship("Parent", back\_populates="child")

Or for many-to-one:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

child\_id = Column(Integer, ForeignKey('child.id'))

child = relationship("Child", back\_populates="parent")

**class** **Child**(Base):

\_\_tablename\_\_ = 'child'

id = Column(Integer, primary\_key=**True**)

parent = relationship("Parent", back\_populates="child", uselist=**False**)

As always, the [relationship.backref](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship) and [backref()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) functions may be used in lieu of the [relationship.back\_populates](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) approach; to specify uselist on a backref, use the [backref()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.backref" \o "sqlalchemy.orm.backref) function:

**from** **sqlalchemy.orm** **import** backref

**class** **Parent**(Base):

\_\_tablename\_\_ = 'parent'

id = Column(Integer, primary\_key=**True**)

child\_id = Column(Integer, ForeignKey('child.id'))

child = relationship("Child", backref=backref("parent", uselist=**False**))

## Many To Many

Many to Many adds an association table between two classes. The association table is indicated by the [relationship.secondary](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument to [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship). Usually, the [Table](https://docs.sqlalchemy.org/en/13/core/metadata.html#sqlalchemy.schema.Table) uses the [MetaData](https://docs.sqlalchemy.org/en/13/core/metadata.html" \l "sqlalchemy.schema.MetaData" \o "sqlalchemy.schema.MetaData) object associated with the declarative base class, so that the [ForeignKey](https://docs.sqlalchemy.org/en/13/core/constraints.html" \l "sqlalchemy.schema.ForeignKey" \o "sqlalchemy.schema.ForeignKey) directives can locate the remote tables with which to link:

association\_table = Table('association', Base.metadata,

Column('left\_id', Integer, ForeignKey('left.id')),

Column('right\_id', Integer, ForeignKey('right.id'))

)

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child",

secondary=association\_table)

**class** **Child**(Base):

\_\_tablename\_\_ = 'right'

id = Column(Integer, primary\_key=**True**)

For a bidirectional relationship, both sides of the relationship contain a collection. Specify using [relationship.back\_populates](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship), and for each [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship) specify the common association table:

association\_table = Table('association', Base.metadata,

Column('left\_id', Integer, ForeignKey('left.id')),

Column('right\_id', Integer, ForeignKey('right.id'))

)

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship(

"Child",

secondary=association\_table,

back\_populates="parents")

**class** **Child**(Base):

\_\_tablename\_\_ = 'right'

id = Column(Integer, primary\_key=**True**)

parents = relationship(

"Parent",

secondary=association\_table,

back\_populates="children")

When using the [relationship.backref](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship.params.backref) parameter instead of [relationship.back\_populates](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship), the backref will automatically use the same [relationship.secondary](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument for the reverse relationship:

association\_table = Table('association', Base.metadata,

Column('left\_id', Integer, ForeignKey('left.id')),

Column('right\_id', Integer, ForeignKey('right.id'))

)

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child",

secondary=association\_table,

backref="parents")

**class** **Child**(Base):

\_\_tablename\_\_ = 'right'

id = Column(Integer, primary\_key=**True**)

The [relationship.secondary](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument of [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship) also accepts a callable that returns the ultimate argument, which is evaluated only when mappers are first used. Using this, we can define the association\_table at a later point, as long as it’s available to the callable after all module initialization is complete:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child",

secondary=**lambda**: association\_table,

backref="parents")

With the declarative extension in use, the traditional “string name of the table” is accepted as well, matching the name of the table as stored in Base.metadata.tables:

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Child",

secondary="association",

backref="parents")

**Warning**

When passed as a Python-evaluable string, the [relationship.secondary](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument is interpreted using Python’s eval() function. **DO NOT PASS UNTRUSTED INPUT TO THIS STRING**. See [Evaluation of relationship arguments](https://docs.sqlalchemy.org/en/13/orm/extensions/declarative/relationships.html#declarative-relationship-eval) for details on declarative evaluation of [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship) arguments.

### Deleting Rows from the Many to Many Table

A behavior which is unique to the [relationship.secondary](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument to [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship) is that the [Table](https://docs.sqlalchemy.org/en/13/core/metadata.html#sqlalchemy.schema.Table) which is specified here is automatically subject to INSERT and DELETE statements, as objects are added or removed from the collection. There is **no need to delete from this table manually**. The act of removing a record from the collection will have the effect of the row being deleted on flush:

*# row will be deleted from the "secondary" table*

*# automatically*

myparent.children.remove(somechild)

A question which often arises is how the row in the “secondary” table can be deleted when the child object is handed directly to [Session.delete()](https://docs.sqlalchemy.org/en/13/orm/session_api.html" \l "sqlalchemy.orm.session.Session.delete" \o "sqlalchemy.orm.session.Session.delete):

session.delete(somechild)

There are several possibilities here:

* If there is a [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship) from Parent to Child, but there is **not** a reverse-relationship that links a particular Child to each Parent, SQLAlchemy will not have any awareness that when deleting this particular Child object, it needs to maintain the “secondary” table that links it to the Parent. No delete of the “secondary” table will occur.
* If there is a relationship that links a particular Child to each Parent, suppose it’s called Child.parents, SQLAlchemy by default will load in the Child.parents collection to locate all Parent objects, and remove each row from the “secondary” table which establishes this link. Note that this relationship does not need to be bidirectional; SQLAlchemy is strictly looking at every [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship) associated with the Child object being deleted.
* A higher performing option here is to use ON DELETE CASCADE directives with the foreign keys used by the database. Assuming the database supports this feature, the database itself can be made to automatically delete rows in the “secondary” table as referencing rows in “child” are deleted. SQLAlchemy can be instructed to forego actively loading in the Child.parents collection in this case using the [relationship.passive\_deletes](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.passive_deletes" \o "sqlalchemy.orm.relationship) directive on [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship); see [Using Passive Deletes](https://docs.sqlalchemy.org/en/13/orm/collections.html#passive-deletes) for more details on this.

Note again, these behaviors are only relevant to the [relationship.secondary](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) option used with [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship). If dealing with association tables that are mapped explicitly and are not present in the [relationship.secondary](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) option of a relevant [relationship()](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html#sqlalchemy.orm.relationship), cascade rules can be used instead to automatically delete entities in reaction to a related entity being deleted - see [Cascades](https://docs.sqlalchemy.org/en/13/orm/cascades.html#unitofwork-cascades) for information on this feature.

## Association Object

The association object pattern is a variant on many-to-many: it’s used when your association table contains additional columns beyond those which are foreign keys to the left and right tables. Instead of using the [relationship.secondary](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.secondary" \o "sqlalchemy.orm.relationship) argument, you map a new class directly to the association table. The left side of the relationship references the association object via one-to-many, and the association class references the right side via many-to-one. Below we illustrate an association table mapped to the Association class which includes a column called extra\_data, which is a string value that is stored along with each association between Parent and Child:

**class** **Association**(Base):

\_\_tablename\_\_ = 'association'

left\_id = Column(Integer, ForeignKey('left.id'), primary\_key=**True**)

right\_id = Column(Integer, ForeignKey('right.id'), primary\_key=**True**)

extra\_data = Column(String(50))

child = relationship("Child")

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Association")

**class** **Child**(Base):

\_\_tablename\_\_ = 'right'

id = Column(Integer, primary\_key=**True**)

As always, the bidirectional version makes use of [relationship.back\_populates](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.back_populates" \o "sqlalchemy.orm.relationship) or [relationship.backref](https://docs.sqlalchemy.org/en/13/orm/relationship_api.html" \l "sqlalchemy.orm.relationship.params.backref" \o "sqlalchemy.orm.relationship):

**class** **Association**(Base):

\_\_tablename\_\_ = 'association'

left\_id = Column(Integer, ForeignKey('left.id'), primary\_key=**True**)

right\_id = Column(Integer, ForeignKey('right.id'), primary\_key=**True**)

extra\_data = Column(String(50))

child = relationship("Child", back\_populates="parents")

parent = relationship("Parent", back\_populates="children")

**class** **Parent**(Base):

\_\_tablename\_\_ = 'left'

id = Column(Integer, primary\_key=**True**)

children = relationship("Association", back\_populates="parent")

**class** **Child**(Base):

\_\_tablename\_\_ = 'right'

id = Column(Integer, primary\_key=**True**)

parents = relationship("Association", back\_populates="child")

Working with the association pattern in its direct form requires that child objects are associated with an association instance before being appended to the parent; similarly, access from parent to child goes through the association object:

*# create parent, append a child via association*

p = Parent()

a = Association(extra\_data="some data")

a.child = Child()

p.children.append(a)

*# iterate through child objects via association, including association*

*# attributes*

**for** assoc **in** p.children:

print(assoc.extra\_data)

print(assoc.child)

## **One To Many**

A One to Many relationship refers to parent with the help of a foreign key on the child table. relationship() is then specified on the parent, as referencing a collection of items represented by the child. The relationship.back\_populates parameter is used to establish a bidirectional relationship in one-to-many, where the “reverse” side is a many to one.

## **Many To One**

On the other hand, Many to One relationship places a foreign key in the parent table to refer to the child. relationship() is declared on the parent, where a new scalar-holding attribute will be created. Here again the relationship.back\_populates parameter is used for Bidirectionalbehaviour.

## **One To One**

One To One relationship is essentially a bidirectional relationship in nature. The uselist flag indicates the placement of a scalar attribute instead of a collection on the “many” side of the relationship. To convert one-to-many into one-to-one type of relation, set uselist parameter to false.

## **Many To Many**

Many to Many relationship is established by adding an association table related to two classes by defining attributes with their foreign keys. It is indicated by the secondary argument to relationship(). Usually, the Table uses the MetaData object associated with the declarative base class, so that the ForeignKey directives can locate the remote tables with which to link. The relationship.back\_populates parameter for each relationship() establishes a bidirectional relationship. Both sides of the relationship contain a collection.

| Table 14-2. HTTP response status codes typically returned by APIs | | |
| --- | --- | --- |
| **HTTP status code** | **Name** | **Description** |
| 200 | OK | The request was completed successfully. |
| 201 | Created | The request was completed successfully and a new resource was created as a result. |
| 202 | Accepted | The request was accepted for processing, but it is still in progress and will run asynchronously. |
| 204 | No Content | The request was completed successfully and there is no data to return in the response. |
| 400 | Bad Request | The request is invalid or inconsistent. |
| 401 | Unauthorized | The request does not include authentication information or the credentials provided are invalid. |
| 403 | Forbidden | The authentication credentials sent with the request are insufficient for the request. |
| 404 | Not Found | The resource referenced in the URL was not found. |
| 405 | Method Not Allowed | The method requested is not supported for the given resource. |
| 500 | Internal Server Error | An unexpected error occurred while processing the request. |

Unit test:

Python has an excellent code coverage tool appropriately called coverage. You can install it with pip:

(venv) $ **pip install coverage**

[Selenium](http://www.seleniumhq.org/) is a web browser automation tool that supports the most popular web browsers in the three major operating systems.

The Python interface for Selenium is installed with pip:

(venv) $ **pip install selenium**

* [Flask-Babel](http://bit.ly/fl-babel): Internationalization and localization support
* [Marshmallow](http://marshmallow.readthedocs.io/en/latest/): Serialization and deserialization of Python objects, useful for API resource representations
* [Celery](http://bit.ly/celery-doc): Task queue for processing background jobs
* [Frozen-Flask](http://bit.ly/flask-frozen): Conversion of a Flask application to a static website
* [Flask-DebugToolbar](http://bit.ly/flask-debug): In-browser debugging tools
* [Flask-Assets](http://bit.ly/fl-assets): Merging, minifying, and compiling of CSS and JavaScript assets
* [Flask-Session](https://pythonhosted.org/Flask-Session/): Alternative implementation of user sessions that use server-side storage
* [Flask-SocketIO](https://flask-socketio.readthedocs.io/): Socket.IO server implementation with support for WebSocket and long-polling

xcross