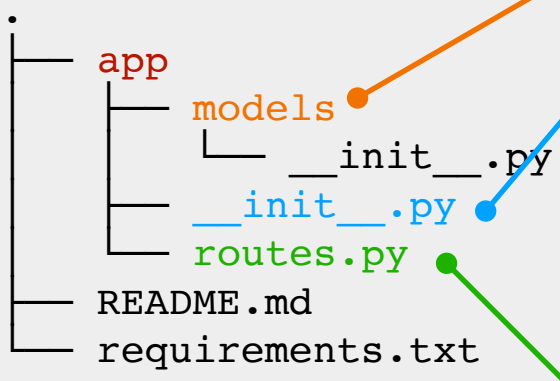
Building an API with Flask

Flask Setup

- [ ] Fork project
- [ ] Git clone
- [ ] Environment setup
  - [ ] % python3 -m venv venv
  - [ ] % source venv/bin/activate # activate environment
- [ ] pip install -r requirements.txt # install project requirements in the environment
- [ ] Running the server
  - [ ] % flask run
  - [ ] % FLASK_ENV=development flask run # optional: run flask in debug mode

Suggested Project Structure:



- The `app/models` directory will hold our data models. We will learn Data models later
- The app/__init__.py File This is the same file we have used to mark a folder as a package! While we often leave this file blank, a common Flask pattern is to define the start-up logic for the Flask server in this file.
  - The start-up logic is responsible for locating and applying any app configuration, and getting the server ready to receive requests.
  - Configurations to the app can include things like:
    - Where's the location of our database?,"
    - "How do we load different data models, the objects that represent our data?,"
    - or "How can we set up template views, called Blueprints?"
- routes.py The responsibility of this file is to define the endpoints.

Dev Workflow for Flask Development:

1. cd into a project root folder
2. Activate a virtual environment
3. Check git status
4. Start the server
5. Cycle frequently among:
   - Writing code
   - Checking git statuses and making git commits
   - Debugging with Postman, server logs, VS Code, and more
6. Stop the server
7. Deactivate the virtual environment

ABOUT THIS STRUCTURE

To make a rainbow, you need to start at red, go through all the warm colors, then go through all the cool colors, until you get to purple. This cheat sheet and the suggested program structure is organized with that same concept in mind. Our app is red and the app contents will contain all the colors to get us to the purple database. Along the way, we will need to do a thing called migration with our code. This means writing code that follows some rules (in various locations of the directory - like rainbow sprinkles) that connect everything together.

Defining Endpoints with Blueprint

Flask Hello Books: A Walk-through

Defining Endpoints with Blueprint

`Blueprint` is a Flask class that provides a pattern for grouping related **routes** (endpoints). Flask will often refer to these routes using the word "view" due to Flask having the potential of sending HTML views. However, we will be sending back JSON.

Creating a Blueprint in `app/routes.py`

Our code will import / use `Blueprint` from `flask`.

In `app/routes.py`:

1. Remember to import `Blueprint`

```
from flask import Blueprint
hello_world_bp = Blueprint("hello_world", __name__)
```

```
.
├── app
│   ├── models
│   │   └── __init__.py
│   ├── __init__.py
│   └── routes.py
├── README.md
└── requirements.txt
```

2. Local variable that holds our blueprint instance. Whenever we need to use this Blueprint to define a route, well use `hello_world_bp`

3. The syntax to initiate a `Blueprint()` object

4. The first argument is a string that will be used to identify this `Blueprint` from the Flask server logs (in the terminal). Use a name related to the data being served or the functionality being provided.

5. The second argument is almost always the special Python variable `__name__`, which the blueprint uses to figure out certain aspects of routing.

Registering a Blueprint in `app/__init__.py`

Every time we instantiate a new `Blueprint`, Flask requires us to register it with `app`. We need to tell the `app` that it should use the endpoints from `hello_world_bp` for its routing.

[[*Where* we register the `Blueprint` depends on the project.]]. But for the Hello Books project, the place to do this is in `app/__init__.py`, inside the `create_app()` function.

In `app/__init__.py`:

```
from flask import Flask

def create_app(test_config=None):
    app = Flask(__name__)

    from .routes import hello_world_bp
    app.register_blueprint(hello_world_bp)

    return app
```

```
.
├── app
│   ├── models
│   │   └── __init__.py
│   ├── __init__.py
│   └── routes.py
├── README.md
└── requirements.txt
```

1. We are importing `hello_world_bp` into this module so we may use it in the next line.

2. We use `app`'s pre-defined function `register_blueprint()` to register the `hello_world_bp` Blueprint.

Defining an Endpoint in `app/routes.py`

Now that we have registered a Blueprint, which will help us organize our routes, we can create an endpoint in `app/routes.py`:

```
.
├── app
│   ├── models
│   │   └── __init__.py
│   ├── __init__.py
│   └── routes.py
├── README.md
└── requirements.txt
```

1. This decorator transforms the function that follows into an endpoint. Use the `.route()` instance method from our Blueprint instance.

```
@blueprint_name.route("/endpoint/path/here", methods=["GET"])
def endpoint_name():
    my_beautiful_response_body = "Hello, World!"
    return my_beautiful_response_body
```

2. Together these arguments define what type of request will be routed to this function. The first argument defines the path (or URL) of the request and the second argument defines a list of HTTP methods (or verbs) the request could have.

3. This function will execute whenever a request that matches the decorator is received. The function can be named whatever feels most appropriate.

4. Define a response body to return. We're using a local variable `my_beautiful_response_body` to hold a value

5. For each endpoint, we must *return* the HTTP response.

Endpoint Example #1: `/hello-world`

Here's the full example of a possible endpoint (with the local variable from earlier):

```
from flask import Blueprint

hello_world_bp = Blueprint("hello_world", __name__)

@hello_world_bp.route("/hello-world", methods=["GET"])
def say_hello_world():
    my_beautiful_response_body = "Hello, World!"
    return my_beautiful_response_body
```

Manually Testing the `/hello-world` Endpoint

While the Flask server is running, we can use Postman to send a `GET` request to `localhost:5000/hello-world`.

- confirm that we can send a request and get a response back with Postman.
- also confirm that we can send a request and get a response back in the browser.
- Finally, confirm that we can also see output in the server log.

Check the Server Logs

Each time we send an HTTP request to our server, we should see a new line appear in the server log.

Endpoint Example #2: `/hello/JSON`

Here's a second possible endpoint that also uses the Blueprint hello_world_bp:
This time the HTTP response body will be the following JSON-like dictionary:

```
{
  "name": "Ada Lovelace",
  "message": "Hello!",
  "hobbies": ["Fishing", "Swimming", "Watching Reality Shows"]
}
```

```
from flask import Blueprint

hello_world_bp = Blueprint("hello_world", __name__)

@hello_world_bp.route("/hello/JSON", methods=["GET"])
def say_hello_json():
    return {
        "name": "Ada Lovelace",
        "message": "Hello!",
        "hobbies": ["Fishing", "Swimming", "Watching Reality Shows"]
    }
```

Manually Testing the `/hello/JSON` Endpoint

- confirm that we can send a request and get a response back with Postman.
- also confirm that we can send a request and get a response back in the browser.
- Finally, confirm that we can also see output in the server log.

Models and Model Setup

Models and Model Setup

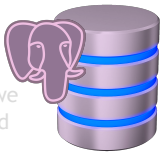
In software, a model is a representation of a single concept relevant to the application. This representation includes state and behavior.

Models in Flask

We will define models in our Flask application code. We will use the package `SQLAlchemy` and follow their patterns to define and use models. Terms like Model, Resource, Entity, and Class are similar because they are all "things" being shared in a set of endpoints.
Models in our Flask code will create a direct connection between the data modeled in our database, and the OOP Python code we can use in our back-end API

Models Are a Link Between a Database and Code

Our models are pieces of data that should be stored in a database. Their data should be persisted. To set up a model, we will need to:



STEP 1 **CREATE DATABASE** (Set up the database for the project in PostgresSQL)

```
psql -U postgres
CREATE DATABASE hello_books_development;
```

You won't enter data with SQL - we are building an app with Python and Flask to automatically enter data

STEP 2 **Connect the Database and Flask**

- Configure the database into our app/__init__.py by providing a path to the database
- Connection string: `postgresql+psycopg2://postgres:postgres@localhost:5432/REPLACE_THIS_LAST_PART_WITH_DB_NAME`
- This *connection string* identifies where the database is, and how to connect to it.

```
app
    models
        __init__.py
        model.py
    __init__.py
    routes.py
README.md
requirements.txt
```

STEP 5 → **Migration Helper**

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate

db = SQLAlchemy()
migrate = Migrate()

def create_app(test_config=None):
    app = Flask(__name__)

    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
    app.config['SQLALCHEMY_DATABASE_URI'] = 'connection string goes here'

    db.init_app(app)
    migrate.init_app(app, db)

    return app
```

SQLAlchemy Migrate

STEP 2

Figure1

STEP 3 **Create the `model.py` File and Class**

- We will create a class for each model.
- The class defines the state and behavior

STEP 4 **Define our `Model` in Python code in `app/models` Directory**

- SQLAlchemy and Flask patterns encourage us to define each model as a class in its own file.
- In this file structure, we create a `models` folder inside of the `app` folder. Note the location of a, `Model`, in `app/models/model.py`. Note the `__init__.py` file to mark the folder as a package.
- `model.py` is a placeholder for each model (aka class). You may specify a different model (aka class) i.e. `app/models/book.py`
- Following the pattern of creating a file for every model, we can create a file for `Book`: `$ touch app/models/book.py`

```
app
    models
        __init__.py
        model.py
        book.py
    __init__.py
    routes.py
README.md
requirements.txt
```

- **Models Are Classes** That Inherit From **`db.Model`** ——Note: **db=SQLAlchemy** see Figure1
- Inside of `app/models/book.py`, we can define the `Book` model. **SQLAlchemy** provides a pattern for creating a class for our model that:
  - Connects the model with our SQL database, mapping attributes to table columns
  - Gives our Flask code the ability to work with `Book` instances, taking advantage of OOP
  - Enables us to access instances of `Book`; each instance corresponds to a row in our database

1. Give this file access to the SQLAlchemy db

2. Define & name a new class after our model. SQLAlchemy uses this class name as the name of the table it creates.

3. Our model will inherit from `db.Model`

```
from app import db

class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    title = db.Column(db.String)
    description = db.Column(db.String)
```

Create attributes, which SQLAlchemy will map to a column, with matching name and value

book.py

Database Migrations

STEP 5 **Make `Model`s Visible to the Flask Migration Helper** Note: see Figure1

- Creating the `model` file alone is not enough for Flask to generate instructions for modifying the database. We must ensure the Migration helper is able to find our `model`s and Ensure that Flask and **SQLAlchemy** are able to see our `model` code.

Our model is set up!

STEP 6 **Next we will use our Flask tools to generate the migrations that we can use to update our database**

One-time Setup: Initialize Migrations

```
(venv) $ flask db init
```

Remember to enter this one-time set up command in your terminal while the project environment is activated

Thanks Flask and SQLAlchemy! When we generate migrations the migration files automatically appear and are automatically placed in a new folder now, the migrations folder!

```
app
    mode
        __in
        model.py
        book.py
    __init__.py
    routes.py
Migrations
    alembic.ini
    env.py
    README
    script.py.mako
    versions
README.md
requirements.txt
```

Every-time Migrations After Each Model Change

```
(venv) $ flask db migrate -m "adds Book model"
```

Generate database migrations with this command. Run this command after every change to a file in the `models` folder.

When running this command ourselves, we should replace the `"adds Book model"` with a description relevant to our recent changes.

Every-time Apply Migrations After Each Model Change

```
(venv) $ flask db upgrade
```

Run this separate command to actually apply the generated migrations. Always run $ `flask db migrate` and $ `flask db upgrade` back-to-back in that order.

STEP 7 **Confirm Migrations in the Database**

Once you are in the Postgres interactive terminal, run \c hello_books_development; to connect to the database.
Check that it worked - list the database with \dt which should show use book, and another table called alembic_version which tracks our migrations.

```
   psql -U postgres
=# \c hello_books_development;

=# \dt
=# \d
```

Great Job Doing all that!