COM661 Full Stack Strategies and Development

BE09. Modular Design with Flask Blueprints

Aims

- To introduce Blueprints as a technique for developing modular Flask applications
- To develop a strategy for assigning endpoints to Blueprints
- To introduce a simple multi-function application using Blueprints
- To demonstrate the refactoring of a single-file application using Blueprints
- To present techniques for handling shared resources in multi-file applications

Table of Contents

9.1 Flask Blueprints

To this point, our back-end application has been constructed as a single code file, containing all of the application scaffold such as database initialization and Flask infrastructure, as well as the route management, user authentication, and API endpoint definition. For smaller applications, this will work fine, but as our application size increases, so the large single code file becomes more difficult to keep track of and maintain. Additionally, were we have components that we want to reuse in other applications, such as user authentication, or a shopping cart, then a single file structure makes it awkward to extract these and re-deploy them elsewhere.

To promote application modularity and code re-use, Flask provides a concept known as **blueprints** to support common patterns across applications. Blueprints can greatly simplify how large applications are structured and make it much easier to share components across applications. In this section we examine the structure of blueprints by considering how a simple API application could be re-structured into separate modules.

9.1.1 A Basic Application

Let's take a simple API application with three endpoints that demonstrate different ways in which Flask routes can be defined as follows

`GET /`	a root endpoint that will display a simple message
`GET /hello`	an endpoint specified by a static path that prints the message "Hello World"
`GET /hello/<name>`	an endpoint with a variable parameter that prints a personalized message to the person identified by name.

We would normally implement this as a simple application as shown in the following code box.

Do it now!	Create a new project folder called ***blueprint*** and prepare it in the usual way by creating a new virtual environment and installing Flask. Inside the blueprint project, create the file ***app.py*** with code as shown in the code box overleaf. Run the application, and present URLs mapping to the three supported routes in a web browser to ensure that it is working as expected. Verify that you receive output such as that shown in Figures 9.1 to 9.3.

File: app.py

```
from flask import Flask

app = Flask(__name__)

@app.route("/", methods=['GET'])
def index():
    return "Index of Hello World"

@app.route("/hello", methods=['GET'])
def hello():
    return "Hello World!"

@app.route("/hello/<string:name>", methods=['GET'])
def hello_name(name):
    return "Hello " + name

if __name__ == '__main__':
    app.run(debug=True)
```

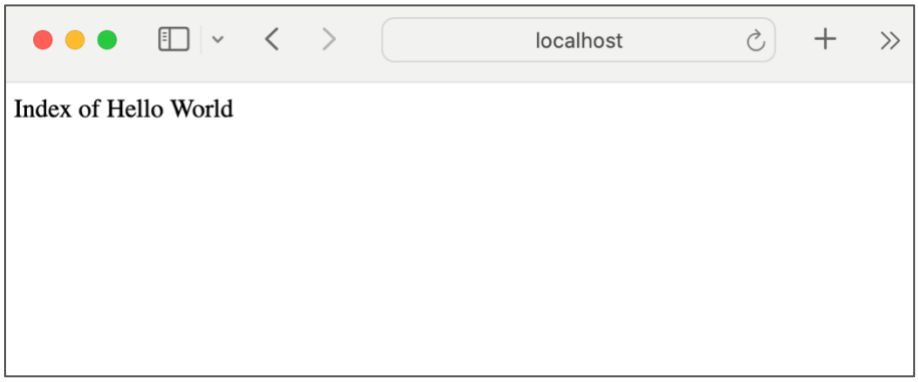

Figure 9.1 Response from http://localhost:5000/

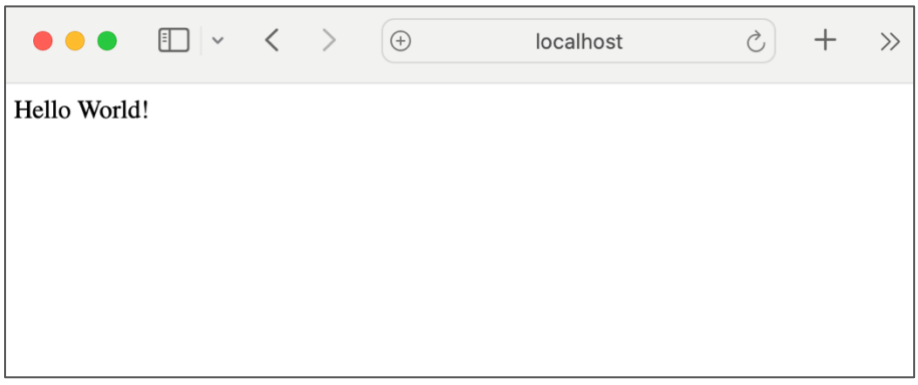

Figure 9.2 Response from http://localhost:5000/hello

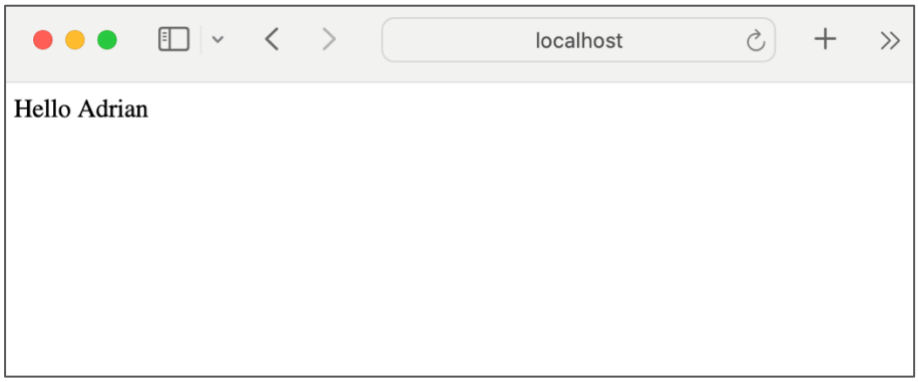

Figure 9.3 Response from http://localhost:5000/hello/Adrian

9.1.2 Refactoring the Application

As a first small example in using blueprints, we will now refactor (re-organise) our code to separate the API endpoint logic from the main application body. A Flask application will often contain many blueprints, so it is common to organize these in a ***blueprints*** sub-folder, giving a structure such as shown in Figure 9.4, below.

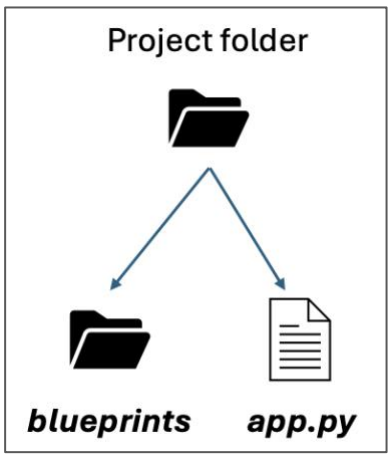

Figure 9.4 Blueprints folder structure

Next, we require a sub-folder inside ***blueprints*** to hold our blueprints. Initially, we only require a single blueprint that we will call ***helloworld***, so we create a ***helloworld*** folder inside ***blueprints*** and create a new (empty) Python file ***helloworld.py*** inside that folder, resulting in the structure shown in Figure 9.5, below.

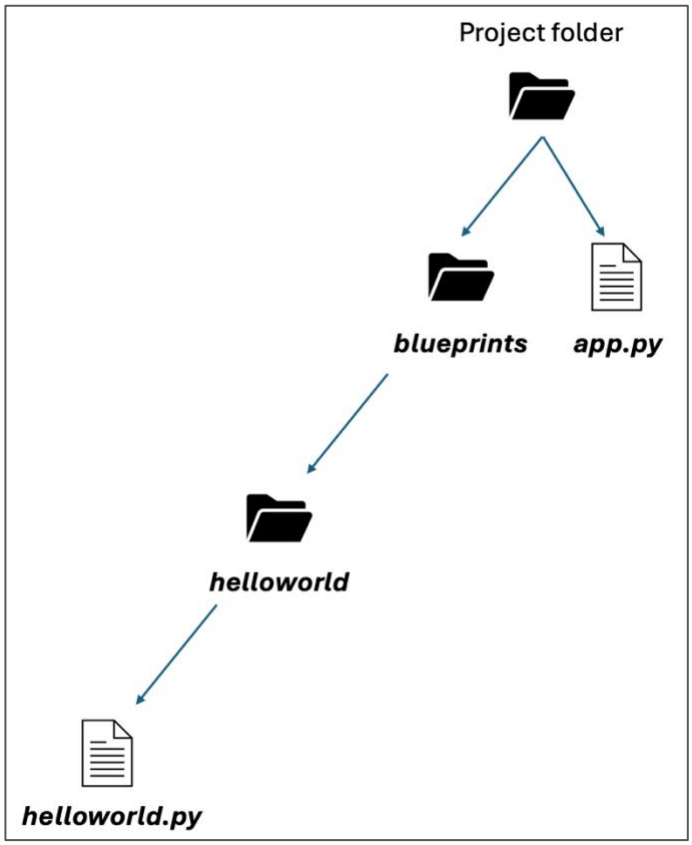

*Figure 9.5 Structure with **helloworld** Blueprint*

Do it now!	Prepare for code refactoring by creating the folder and file structure shown in Figure 9.5 within your blueprint application.

Now that the file and folder structure is in place, we can move the code that services the **helloworld** endpoints from ***app.py*** into the new file ***blueprints/helloworld/helloworld.py***. This leaves the code distributed across the two files as follows.

File: app.py

```
from flask import Flask

app = Flask(__name__)

if __name__ == '__main__':
    app.run(debug=True)
```

File: blueprints/helloworld/helloworld.py

```
@app.route("/", methods=['GET'])
def index():
    return "Index of Hello World"

@app.route("/hello", methods=['GET'])
def hello():
    return "Hello World!"

@app.route("/hello/<string:name>", methods=['GET'])
def hello_name(name):
    return "Hello " + name
```

The next step is to turn the functionality within **helloworld.py** into a Blueprint which could then be potentially attached to any other Flask application. We do this by importing the **`Blueprint`** class from the **`flask`** library and creating a new Blueprint object, providing it with the name by which it will be known. Finally, we update each route definition to specify that the endpoints are routes of the blueprint rather than of the top-level application.

File: blueprints/helloworld/helloworld.py

```
from flask import Blueprint

helloworld_bp = Blueprint("helloworld_bp", __name__)

@helloworld_bp.route("/", methods=['GET'])
def index():
    return "Index of Hello World"

@helloworld_bp.route("/hello", methods=['GET'])
def hello():
    return "Hello World!"

@helloworld_bp.route("/hello/<string:name>", methods=['GET'])
def hello_name(name):
    return "Hello " + name
```

With the blueprint defined and available for use, we can now import it and register it with our main application by using the **`register_blueprint()`** method of the Flask object that we have called **`app`**.

File: app.py

```
from flask import Flask
from blueprints.helloworld.helloworld import helloworld_bp

app = Flask(__name__)

app.register_blueprint(helloworld_bp)

if __name__ == '__main__':
    app.run(debug=True)
```

Do it now!	Refactor your code as shown in the code boxes for *app.py* and ***helloworld.py*** above and ensure that the application still runs as before, generating output as shown earlier in Figures 9.1 to 9.3.

9.1.3 Adding a Second Blueprint

We will demonstrate the addition of a second blueprint to our application by specifying a simple calculator function with endpoints to add and subtract a pair of integer values provided as parameters within the URL, such that the route **/add/1/2** will return the result of the sum 1+2 and the route **/subtract/3/2** will return the result of the sum 3-2.

This blueprint will be called **`calc_bp`** and will be specified as shown in the code box below.

File: blueprints/calculator/calculator.py

```
from flask import Blueprint

calc_bp = Blueprint("calc_bp", __name__)

@calc_bp.route("/add/<int:x>/<int:y>", methods=['GET'])
def add(x, y):
    return str(x + y)

@calc_bp.route("/subtract/<int:x>/<int:y>", methods=['GET'])
def subtract(x, y):
    return str(x - y)
```

Do it now!	Add a new folder ***calculator*** within the ***blueprints*** sub-folder and create the file ***calculator.py*** with code as shown above.

Now, we can register the calculator blueprint with the application by repeating the import and register process as previously.

File: app.py

```
from flask import Flask
from blueprints.helloworld.helloworld import helloworld_bp
from blueprints.calculator.calculator import calc_bp

app = Flask(__name__)

app.register_blueprint(helloworld_bp)
app.register_blueprint(calc_bp)

if __name__ == '__main__':
    app.run(debug=True)
```

Do it now!	Register the calculator blueprint with the application and test it. Make sure that all of the previous Hello World functionality is still working as expected, and that you also now have addition and subtraction operations available as shown in Figure 9.6 and 9.7, below.

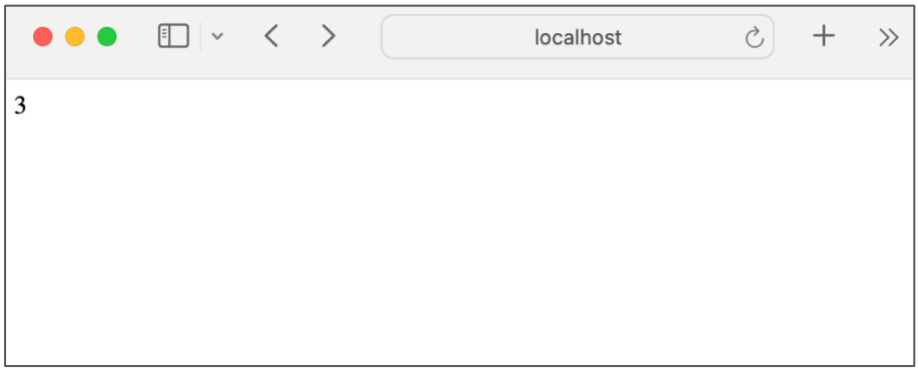

Figure 9.6 Response from http://localhost:5000/add/1/2

Figure 9.7 Response from localhost:5000/subtract/3/2

9.1.4 Resolving Naming Conflicts

One potential issue when combining Blueprints from various sources to make new applications is the conflict that arises when different Blueprints have used the same URL in a route definition. Consider the case where we add a root path to the calculator blueprint as highlighted in the code box below.

File: blueprints/calculator/calculator.py

```
from flask import Blueprint

calc_bp = Blueprint("calc_bp", __name__)

@calc_bp.route("/", methods=['GET'])
def index():
    return "Index of Calculator"

@calc_bp.route("/add/<int:x>/<int:y>", methods=['GET'])
def add(x, y):
    return str(x + y)

@calc_bp.route("/subtract/<int:x>/<int:y>", methods=['GET'])
def subtract(x, y):
    return str(x - y)
```

When we add this route to the calculator and run the application, we now find that the URL ***http://localhost:5000/*** matches both the root path to HelloWorld and the root path to the Calculator. As the HelloWorld blueprint is the first to be registered with the app, it is its route definition that takes priority and that of the Calculator blueprint is unavailable.

Do it now!	Add the additional endpoint to the Calculator blueprint as shown above and verify that when you attempt to navigate to the URL http://localhost:5000/ it is the matching route of the first blueprint to be registered that is serviced.

Flask provides a means of resolving such conflicts by providing for an option **`url_prefix`** parameter in the **`register_blueprint()`** method. This allows us to guarantee the uniqueness of all routes in our application by specifying a path that will be prepended to the route specified for the endpoint.

For example, in ***app.py***, we might provide a **`url_prefix`** for the Calculator blueprint as follows.

File: app.py

```
from flask import Flask
from blueprints.helloworld.helloworld import helloworld_bp
from blueprints.calculator.calculator import calc_bp

app = Flask(__name__)

app.register_blueprint(helloworld_bp)
app.register_blueprint(calc_bp, url_prefix='/calculator')

if __name__ == '__main__':
    app.run(debug=True)
```

By this, the routes defined within the Calculator blueprint as

`/`

`/add/<x>/<y>`

`/subtract/<x>/<y>`

will now be matched instead by the URLs

`/calculator`

`/calculator``/add/<x>/<y>`

`/calculator``/subtract/<x>/<y>`

Do it now!	Add the **`url_prefix`** parameter to the statement that registers the Calculator blueprint as shown above. Run the application and verify that the calculator functionality is now available at the ***/calculator/…*** URLs and that the calculator index is now available as shown in Figure 9.8, below.

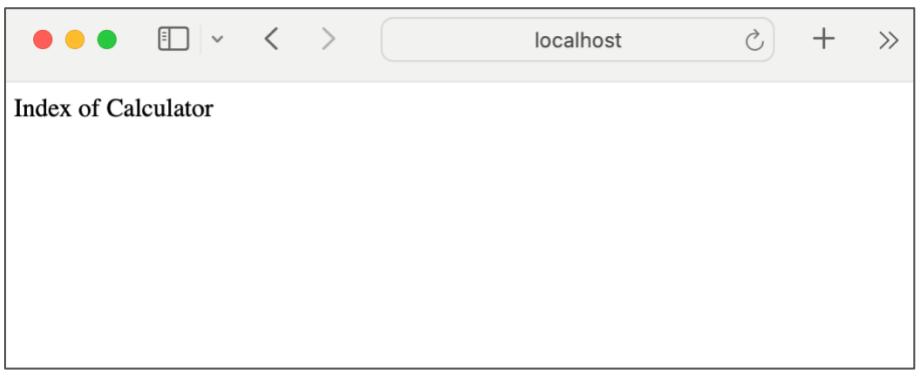

Figure 9.8 Response from http://localhost:5000/calculator

9.1.5 Inter-Blueprint Communication

One advantage of maintaining all endpoint definitions within a single application file is that any part of the application can be reached from any other. For example, if a user in an online cinema booking application makes a request to buy a seat from ***POST /cinema/booking***, the booking logic may determine that they are now eligible for a special offer and redirect them instead to the endpoint ***POST /cinema/bonusoffer***. However, when these endpoints are in different blueprints, they are no longer visible to each other.

Flask resolves this by providing **`redirect`** and **`url_for`** methods that invoke other endpoints in the application and automatically generates the URLs for those endpoints from the information gathered during the **`register_blueprint()`** process.

We can demonstrate this by adding an endpoint to the Calculator blueprint that redirects to an endpoint within the HelloWorld blueprint. In this example, the redirection is automatic, but we could easily use program logic to make the redirection subject to some condition or combination of conditions.

File: blueprints/calculator/calculator.py

```
from flask import Blueprint, redirect, url_for

calc_bp = Blueprint("calc_bp", __name__)

@calc_bp.route("/", methods=['GET'])
def index():
    return "Index of Calculator"

@calc_bp.route("/add/<int:x>/<int:y>", methods=['GET'])
def add(x, y):
    return str(x + y)

@calc_bp.route("/subtract/<int:x>/<int:y>", methods=['GET'])
def subtract(x, y):
    return str(x - y)

@calc_bp.route("/go_to_hello", methods=['GET'])
def go_to_hello():
    return redirect(url_for("helloworld_bp.hello"))
```

Here, we have added an new endpoint ***/calculator/go_to_hello*** which will invoke the **`go_to_hello()`** function. In this, the call to **`url_for()`** can be read as “get me the URL of

the endpoint associated with the function called **`hello()`** within the blueprint called **`helloworld_bp`**". The call to **`redirect()`** then makes an API request to that URL.

Do it now!	Make the changes to ***calculator.py*** as shown in the code box above. Then run the application and make sure that a request to ***/calculator/go_to_hello*** returns the output of a call to ***/hello***. You can see in the Python console how the request to ***/calculator/go_to_hello*** has generated HTTP Status Code 302 indicating redirection, before the call to ***/hello*** is made, as shown in Figure 9.9, below.

```
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 228-508-889
127.0.0.1 - - [20/Aug/2024 12:56:02] "GET /calculator/go_to_hello HTTP/1.1" 302 -
127.0.0.1 - - [20/Aug/2024 12:56:02] "GET /hello HTTP/1.1" 200 -
```

Figure 9.9 Redirection Generates Status Code 302

9.2 Case Study – Refactoring the Biz Directory

Having examined blueprints and their application in modular design, we can return to our Biz Directory example and apply blueprint modularization to the currently monolithic ***app.py***.

Do it now!	In preparation for refactoring the Biz Directory application, create a new application folder ***biz_bp***. Create a new virtual environment within this folder, activate it, and use **pip** to install **`flask`**, **`pymongo`**, **`pyjwt`** and **`bcrypt`**. Now copy your existing ***app.py*** from the Biz Directory into your new folder and check that the application is running as expected.

9.2.1 Identification and Implementation of Blueprints

Normally, we would aim to identify blueprints during the design process, but when refactoring a large application, the best approach is to identify the main actors in the system by considering the URLs used to identify routes. Remember that good RESTful design uses nouns rather than verbs in the URL, so it should be easy to quickly identify the main objects.

The list of endpoints in the Biz Directory application is presented in Table 9.1 below

Method	URL	Action
GET	/api/v1.0/businesses	Get all (or multiple) businesses
POST	/api/v1.0/businesses	Create a new business
GET	/api/v1.0/businesses/<id>	Get a specific business
PUT	/api/ v1.0/businesses/<id>	Update a specific business
DELETE	/api/ v1.0/businesses/<id>	Delete a specific business
GET	/api/ v1.0/businesses/<id>/reviews	Get all (or multiple) reviews for a specific business
POST	/api/ v1.0/businesses/<id>/reviews	Add a review for a specific business
GET	/api/ v1.0/businesses/<id>/reviews/<id>	Get a specific review for a specific business
PUT	/api/ v1.0/businesses/<id>/reviews/<id>	Update a specific review for a specific business
DELETE	/api/ v1.0/businesses/<id>/reviews/<id>	Delete a specific review
GET	/api/v1.0/businesses/login	User login
GET	/api/v1.0/businesses/logout	User logout

Table 9.1 Biz Directory Endpoints

One possible subdivision of the functionality is suggested by the colour coding in the table, which identifies groups as

Business-level endpoints those that deal with the collection of businesses or with individual businesses

Review-level endpoints those that deal with the collection of reviews for a specified business, or with individual reviews

User-level endpoints those that provide a user authentication service

This appears to be a reasonable sub-division of the functionality, so we can proceed to create our folder and file structure.

Do it now!	Inside the application folder, create a new sub-folder called ***blueprints***. Inside ***blueprints***, create additional subfolders called ***businesses***, ***reviews*** and ***auth***. Now create (empty) Python files ***businesses.py***, ***reviews.py***, and ***auth.py*** inside the new bottom-level folders. The folder and file structure should be as illustrated in Figure 9.10.

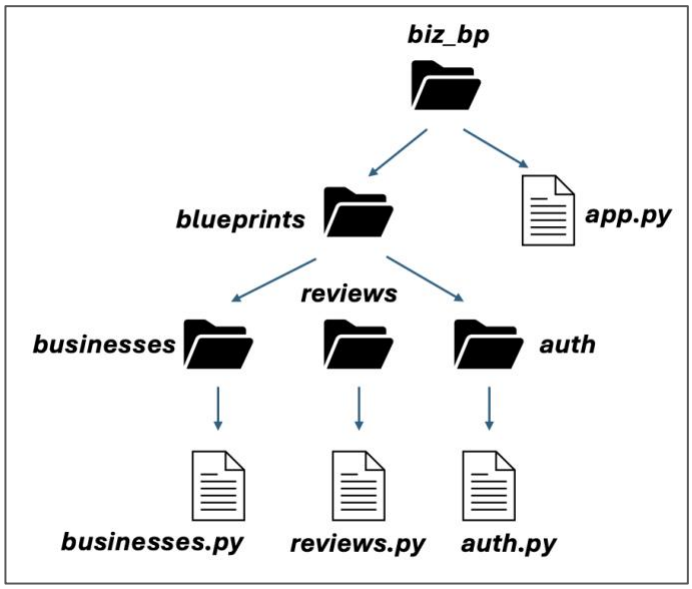

Figure 9.10 ***biz_bp*** *Folder and File Structure*

Now that the folder structure is created, we can begin to migrate functionality from ***app.py*** into the individual blueprint Python files in the way shown in Table 9.1.

Note:	As you move code from file to file, you may receive error indications from your code editor. This is to be expected as we have some modifications to perform to the code after it is moved.

Do it now!	Move the functionality from ***app.py*** to each of ***businesses.py***, ***reviews.py*** and ***auth.py*** in accordance with the allocation shown in Table 9.1. Remember that for each endpoint, you need to move **both** the `@app.route()` decorator and the associated function.

Now that the functionality has been moved, we need to package each of ***businesses.py***, ***reviews.py*** and ***auth.py*** as a Blueprint and register it with the main application. The Blueprint is created by adding a line to the top of each file and changing the `app.route()` decorator for each endpoint to use the blueprint name as follows.

File: blueprints/businesses/businesses.py.

```
from flask import Blueprint
businesses_bp = Blueprint("businesses_bp", __name__)

@businesses_bp.route("/api/v1.0/businesses", methods=["GET"])
def show_all_businesses():

...

# repeat for each endpoint in businesses.py
```

File: blueprints/reviews/reviews.py.

```
from flask import Blueprint
reviews_bp = Blueprint("reviews_bp", __name__)

@reviews_bp.route(
         "/api/v1.0/businesses/<string business_id>/reviews",
         methods=["POST"])
@jwt_required
def add_new_review(business_id):

...

# repeat for each endpoint in reviews.py
```

File: blueprints/auth/auth.py.

```
from flask import Blueprint
auth_bp = Blueprint("auth_bp", __name__)

@auth_bp.route("/api/v1.0/login", methods=["GET"])
def login():

...

# repeat for each endpoint in auth.py
```

We can now register our new blueprints with the main application by adding the highlighted code below to ***app.py***.

File: app.py

```
...

from blueprints.businesses.businesses import businesses_bp
from blueprints.reviews.reviews import reviews_bp
from blueprints.auth.auth import auth_bp

app = Flask(__name__)
app.config['secret_key'] = 'mysecret'
app.register_blueprint(businesses_bp)
app.register_blueprint(reviews_bp)
app.register_blueprint(auth_bp)

...
```

Do it now!	Specify each of ***businesses.py***, ***reviews.py*** and ***auth.py*** as a blueprint as described in the code boxes above. Then register each of the blueprints with the main application by adding the highlighted code to **app.py**.

9.2.2 Handling Shared Resources

Although all of the functionality has been distributed to the appropriate blueprints, the application is not yet ready to run. Among the functionality not yet considered are the **`@jwt_required`** and **`@admin_required`** decorators and while these are no longer required in ***app.py***, they are needed in each of the blueprint files.

The easiest way to make these available across multiple blueprints is to extract them into a separate python module which can then be imported where they are needed.

Do it now!	Create a new file ***decorators.py*** in the same folder as ***app.py*** and move the definitions of **`jwt_required()`** and **`admin_required()`** into this new file. Note that you will also need to copy the imports (but **NOT** the blueprint imports) and the database connection code into ***decorators.py***.

Following this, the code in ***decorators.py*** should look as shown in the following code box.

File: decorators.py

```
from flask import Flask, request, jsonify, make_response
from bson import ObjectId
from pymongo import MongoClient
import jwt
import datetime
from functools import wraps
import bcrypt

client = MongoClient("mongodb://127.0.0.1:27017")
db = client.bizDB      # select the database
businesses = db.biz    # select the collection name
users = db.users
blacklist = db.blacklist

def jwt_required(func):
    @wraps(func)
    def jwt_required_wrapper(*args, **kwargs):

...

def admin_required(func):
    @wraps(func)
    def admin_required_wrapper(*args, **kwargs):

...
```

We have one final adjustment to make to ***decorators.py*** before we can import it into the blueprints. When encoding and decoding the JWT token, the code makes reference to a configuration variable **`app.config['secret_key']`** that is initialised in ***app.py***. However, the application variable **`app`** is no longer in scope since it is only accessible from ***app.py***. We could simply re-define **`secret_key`** as a string variable in each file where it is needed, but a better solution is to create another top-level Python file where any global variables can be maintained.

Do it now!	Create a new file ***globals.py*** in the same folder as ***app.py*** and add a definition for a variable **`secret_key`** to hold whatever string value you want to use as your secret. For example, you might have a definition such as that shown below.

File: globals.py

```
secret_key = 'mysecret'
```

Now we can import ***globals.py*** into any other file that requires one of our global values and make use of the value by referring to the module in which it is defined. For example, ***decorators.py*** can be modified as shown below.

File: decorators.py

```
from flask import Flask, request, jsonify, make_response
from bson import ObjectId
from pymongo import MongoClient
import jwt
import datetime
from functools import wraps
import bcrypt
import globals

...

def jwt_required(func):
    @wraps(func)
    def jwt_required_wrapper(*args, **kwargs):

        ...

        try:
            data = jwt.decode(token,
                              globals.secret_key,
                              algorithms="HS256")

...
```

Do it now!

Update ***decorators.py*** to import the new ***globals.py*** file as shown above. Make sure that you update **both** instances of **`app.config['SECRET_KEY']`** to **`globals.secret_key`**.

Make similar changes in the auth blueprint file to import the globals and replace the instance of **`app.config['SECRET_KEY']`** found there.

You can also now remove the definition of **`app.config['SECRET_KEY']`** from ***app.py*** as it is no longer required.

Now, we can import the **`@jwt_required`** and **`@admin_required`** decorators into the each of the blueprints by adding the following **`import`** statement to the top of each file.

File: decorators/businesses/businesses.py

File: decorators/reviews/reviews.py

File: decorators/auth/auth.py

```
from flask import Blueprint
from decorators import jwt_required, admin_required

...
```

Do it now!	Update each of the blueprint files to import the decorators as shown in the code box above.

The final stage before running the application is to make sure that all of the imports originally made at the top of ***app.py*** are moved to the correct place, even if they need to be duplicated. First, if you examine the code for ***businesses.py***, ***reviews.py*** and ***auth.py***, you will see that each instance of the Flask elements **`request`**, **`make_response()`** and **`jsonify()`** are indicating an error as they are not imported into the files where they are used.

File: blueprints/businesses/businesses.py

File: blueprints/reviews/reviews.py

File: blueprints/auth/auth.py

```
from flask import Blueprint, request, make_response, jsonify

...
```

Now, if we examine the code for ***decorators.py***, we see that the **`Flask`** object is being imported even though it is never used, so we can remove that from the **`import`** statement leaving just the following statement.

File: decorators.py

```
from flask import request, make_response, jsonify

...
```

Finally regarding imports from Flask, we find that the only object needed in ***app.py*** is the **`Flask`** object itself, so the corresponding **`import`** statement there can be modified to the following.

File: app.py

```
from flask import Flask

...
```

Do it now!	Update the Flask imports across the **`businesses.py`**, **`reviews.py`**, **`auth.py`**, **`decorators.py`** and **`app.py`** files so that each only imports the facilities it requires.

The final modification required is to make the database functionality available in each file where it is needed. In order to increase maintainability, it is better to add the database connection statement to the globals, so that a change of database server or database name only requires an amendment in one place. This also requires that the **`pymongo`** import is added to the file ***globals.py*** as shown below.

File: globals.py

```
from pymongo import MongoClient

secret_key = 'mysecret'

client = MongoClient("mongodb://127.0.0.1:27017")
db = client.bizDB           # select the database
```

Next, we need to create the appropriate collection references in each of the files that holds the functionality. The **businesses** and **reviews** blueprints require access to the ***businesses*** collection, while the **auth** blueprint requires access to ***users*** and ***blacklist***, and the **decorators** definition requires access to **blacklist**. This is accomplished by the following modifications. In each case we need to **`import globals`** to the corresponding file and create a connection to the required collections.

File: blueprints/businesses/businesses.py

```
...

import globals

businesses_bp = Blueprint("businesses_bp", __name__)

businesses = globals.db.biz    # select the collection name
```

File: blueprints/reviews/reviews.py

```
...
import globals

reviews_bp = Blueprint("reviews_bp", __name__)

businesses = globals.db.biz
```

File: blueprints/auth/auth.py

```
...
import globals

auth_bp = Blueprint("auth_bp", __name__)

users = globals.db.users
blacklist = globals.db.blacklilst
```

File: decorators.py

```
...

import globals

blacklist = globals.db.blacklist
```

Do it now!	Ensure that each file has the appropriate database access by performing the updates shown in the code boxes above.

The final step is to tidy up the remaining imports so that all components have exactly the information they require, but no more. In the businesses and reviews blueprints, both

modules need access to the **`ObjectId()`** constructor that was imported from **bson**, so that import needs to be included in each file.

File: blueprints/businesses/businesses.py

File: blueprints/reviews/reviews.py

```
from flask import Blueprint, request, jsonify, make_response
from bson import ObjectId
from decorators import jwt_required, admin_required
import globals

...
```

The auth blueprint requires access to **`bcrypt`**, **`jwt`** and **`datetime`**, so we can copy those statements from ***app.py*** to ***auth.py***.

File: blueprints/auth/auth.py

```
from flask import Blueprint, request, make_response, jsonify
import bcrypt
import jwt
import datetime
from decorators import jwt_required, admin_required
import globals

...
```

Finally, we can remove unneeded imports from ***decorators.py*** and ***app.py***. Initially, we populated ***decorators.py*** by copying all imports from ***app.py***, but not all of these are needed, so we can reduce the import section as shown below.

File: decorators.py

```
from flask import request, jsonify, make_response
import jwt
from functools import wraps
import globals

blacklist = globals.db.blacklist
...
```

That only leaves ***app.py***, where all functionality has been moved into either a blueprint, the decorators file, or the globals. All that remains of ***app.py*** is shown below in its entirety.

File: app.py

```
from flask import Flask
from blueprints.businesses.businesses import businesses_bp
from blueprints.reviews.reviews import reviews_bp
from blueprints.auth.auth import auth_bp

app = Flask(__name__)
app.register_blueprint(businesses_bp)
app.register_blueprint(reviews_bp)
app.register_blueprint(auth_bp)

if __name__ == "__main__":
      app.run(debug=True)
```

Do it now!	Make the final updates to the imports as described in the above code boxes. Run the application and ensure that it is still working exactly as before.

9.3 Further Information

- https://flask.palletsprojects.com/en/3.0.x/blueprints/
  Modular Applications with Blueprints

- https://realpython.com/flask-blueprint/
  Using a Flask Blueprint to Architect your Applications

- https://www.freecodecamp.org/news/how-to-use-blueprints-to-organize-flask-apps/
  How to use Blueprints to Organise your Flask Apps

- https://www.youtube.com/watch?v=_LMiUOYDxzE
  Flask Blueprints Make Your Apps Modular And Professional (YouTube)

- https://blog.miguelgrinberg.com/post/the-flask-mega-tutorial-part-xv-a-better-application-structure
  The Flask Mega-Tutorial, Part XV: A Better Application Structure

- https://www.youtube.com/watch?v=WteIH6J9v64
  Flask Tutorial #10 - Blueprints & Using Multiple Python Files (YouTube)

- https://nickjanetakis.com/blog/flask-nested-blueprints-example
  Flask Nested Blueprints Example

- https://dev.to/blankgodd/creating-a-blueprint-based-flask-app-199h
  Build a Flask Application with Blueprints

- https://hackersandslackers.com/flask-blueprints/
  Organise Flask Apps with Blueprints

- https://www.youtube.com/watch?v=pjVhrIJFUEs
  Intro to Flask Blueprints (YouTube)

- https://www.youtube.com/watch?v=GZ51so48YnM
  Blueprints - Flask Tutorial Series #9 (YouTube)

- https://www.youtube.com/watch?v=hc5tC3TCs8g
  Flask Blueprints - Hello World (YouTube)