Dipartimento di Ingegneria Informatica
Automatica e Gestionale Antonio Ruberti

Sapienza
Università di Roma

Software Engineering

Developing RESTful APIs with Python and Flask

JSON

Luciana Silo
Software Engineering

SAPIENZA
UNIVERSITÀ DI ROMA

What is JSON?

- JSON stands for **JavaScript Object Notation**
- It is a lightweight format for storing and transporting data
- JSON is often used when data is sent from a server to a web page
- It is is *"self-describing"* and easy to understand

A little bit of history

- JSON was initially developed as a format for communicating between JavaScript clients and back-end servers
- It quickly gained popularity as a **human-readable format** that front-end programmers could use to communicate with the back end using a terse, standardized format
- Developers also discovered that JSON was *very flexible*: you could **add**, **remove**, and **update** fields ad hoc

JSON Example

- This example defines an employees object: an array of 3 employee records (objects):

-
  ```
  {
  "employees":[
      {"firstName":"John", "lastName":"Doe"},
      {"firstName":"Anna", "lastName":"Smith"},
      {"firstName":"Peter", "lastName":"Jones"}
  ]
  }
  ```

JSON Syntax Rules

- Data is in ***name/value*** pairs
- Data is separated by ***commas***
- ***Curly braces*** hold objects
- ***Square brackets*** hold arrays

JavaScript Object Notation

- The JSON format is syntactically identical to the code for creating JavaScript objects
- Because of this similarity, a JavaScript program can easily convert JSON data into native JavaScript objects
- The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only
  - code for reading and generating JSON data can be written in *any programming language*

JSON Data - A Name and a Value

- JSON data is written as **name/value pairs**, just like JavaScript object properties

- A name/value pair consists of a *field name* (in double quotes), followed by a colon, followed by a *value*:

```
"firstName":"John"
```

- JSON names require double quotes, JavaScript names do not

JSON Example

- This example is a JSON string:

```
'{"name":"John", "age":30, "car":null}'
```

- It defines an object with **3 properties**:
    - name
    - age
    - car

- Each property has a **value**

JSON Objects

- JSON objects are written inside **curly braces**
- Just like in JavaScript, objects can contain multiple *name/value pairs*:

```
{"firstName":"John", "lastName":"Doe"}
```

JSON Arrays

- JSON arrays are written inside **square brackets**
- Just like in JavaScript, an array can contain objects:

```
"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
```

- In the example above, the object "employees" is an array
  - it contains **three objects**
  - each object is a **record of a person** (with a first name and a last name)

Python JSON

- JSON is a *syntax* for storing and exchanging data
- JSON is *text*, written with JavaScript object notation
- Python has a **built-in package** called `json`, which can be used to work with JSON data
- Import the json module:

```
import json
```

Parse JSON - Convert from JSON to Python

- If you have a JSON string, you can parse it by using the `json.loads()` method:
    - the result will be a Python dictionary

- Example:

```
import json
# some JSON:
x =  '{ "name":"John", "age":30, "city":"New York"}'

# parse x:
y = json.loads(x)

# the result is a Python dictionary:
print(y["age"])
```

Convert from Python to JSON

- If you have a Python object, you can convert it into a JSON string by using the `json.dumps()` method

-
  ```
  import json
      # a Python object (dict):
      x = {
        "name": "John",
        "age": 30,
        "city": "New York"
      }
      # convert into JSON:
      y = json.dumps(x)
      # the result is a JSON string:
      print(y)
  ```

Convert from Python to JSON

- You can convert Python objects of the following types, into JSON strings: *dict, list, tuple, string, int, float, True, False, None*

- Convert Python objects into JSON strings, and print the values:

```
import json
print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

Format the Result

Convert a Python object containing all the legal data types:

```
import json
x = {
  "name": "John",
  "age": 30,
  "married": True,
  "divorced": False,
  "children": ("Ann","Billy"),
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}
      print(json.dumps(x))
```

Format the Result

- The code before prints a JSON string, but it is not very easy to read, with no indentations and line breaks
- The `json.dumps()` method has parameters to make it easier to read the result:
  - use the indent parameter to define the numbers of indents:

```
json.dumps(x, indent=4)
```

- The `json.dumps()` method has parameters to order the keys in the result:
  - use the `sort_keys` parameter to specify if the result should be sorted or not:

```
json.dumps(x, indent=4, sort_keys=True)
```

FLASK

What is Flask?

- **Flask** is a web application framework written in *Python*

- It allows you to develop web applications **easily**

- It is based on the *Werkzeg WSGI* toolkit and the *Jinja2* template engine
    - the **Web Server Gateway Interface** (WSGI) is the specification of a common interface between web servers and web applications
    - **Werkzeug** is a WSGI toolkit that implements requests, response objects, and utility functions
    - **jinja2** is a popular template engine for Python

Why is Flask a good web framework choice?

- Flask is very **Pythonic**
- It’s easy to get started with Flask, because it doesn’t have a huge learning curve
- It’s very *explicit*, this increases **readability**

Prerequisites

- Install *Python 3*
- Install *Pip*
- Install *Flask*, using pip
  - `pip install flask`

First Program: *hello.py*

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello_world():
    return "Hello, World!"
```

- To run it, execute the following command: `flask --app hello run`
- To reach the application open a browser and navigate to http://127.0.0.1:5000/ or by issuing `curl http://127.0.0.1:5000/`

book.py

- Small example with data about books represented as a list of dictionaries
- Each dictionary has for each book: ID number, title, author and year of publication

```
from flask import Flask, jsonify, request

app = Flask(__name__)

app.config["DEBUG"] =True


books = [

    {'id': 0,

     'title': 'Endymion',

     'author': 'John Keats',

     'published': '1818'},

    {'id': 1,

     'title': 'Romeo and Juliet',

     'author': 'William Shakespeare',

     'published': '1597'},

    {'id': 2,

     'title': 'The great Gatsby',

     'author': 'Francis Scott Fitzgerald',

     'published': '1925'}

]
```

book.py - GET

```
@app.route('/books', methods=['GET'])

def get_all_books():

    return jsonify(books)

@app.route('/books/<int:id>', methods=['GET'])


def get_books_id(id):

    for book in books:

        if book['id'] == id:

            return jsonify(books[id])

    return f'Product with ID {id} not found', 404
```

- To get all books:

```
curl -i
http://localhost:5000/books
```

- To get a specific book:

```
curl -i
http://localhost:5000/books/1
```

book.py - POST

```
@app.route('/books', methods=['POST'])
def add_books():
    data = request.json
    new_id = max([book['id'] for book in books])+1
    new_book = {'id': new_id,
   'title': data['title'],
                'author': data['author'],
                'published': data['published']
                }
    books.append(new_book)
    return jsonify(new_book)
```

- To add a new book:

```
curl -X POST -H "Content-Type: application/json" -d '{ "id": 3, "title": "The Old Man and the Sea", "author": "Ernest Hemingway", "published": "1952" }' http://localhost:5000/books
```

book.py - PUT

```
@app.route('/books/<int:id>', methods=['PUT'])
def update_books(id):
    change_data = request.json
    for book in books:
        if book['id'] == id:
            book['title'] = change_data['title']
            book['author'] = change_data['author']
            book['published'] = change_data['published']
            return jsonify(book), 200
    return f'Book with id {id} not found', 404
```

- To update a book:

```
curl -X PUT -H "Content-Type:
application/json" -d '{ "id":
3, "title": "The Old Man and
the Sea", "author": "Ernest
Hemingway", "published": "1955"
}'
http://localhost:5000/books/3
```

`book.py` - DELETE

```
@app.route('/books/<int:id>',methods=['DELETE'])
def remove_books(id):
    for book in books:
        if book['id'] == id:
            books.remove(book)
            return f'Product with ID {id} removed', 200
    return f'Product with ID {id} not found', 404
```

- To delete a book:

```
curl -i -X DELETE
http://localhost:5000/books /3
```

CREATING DATABASE

Flask-SQLAlchemy

- **Flask-SQLAlchemy** is an extension for Flask that adds support for SQLAlchemy to your application
- **SQLAlchemy** is the Python SQL toolkit and Object Relational Mapper that gives application developers the *full power* and *flexibility* of SQL
- Install:
  - `pip install Flask-SQLAlchemy`

Database setup

- Create a new Python file called *`main.py`*
- After create Flask server, setup SQLAlchemy in a Flask project and wrap our Flask app variable in a new SQLAlchemy object
- Setup also `SQLALCHEMY_DATABASE_URI` in our flask app configuration to specify which database we want to use and how to access it

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db'
db = SQLAlchemy(app)

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

Database model

- A **model** is a representation of your database, where you can *store*, *fetch*, and *manipulate* your data from it
- It represents a single **table/collection**
- Every book has:
  - an **id** property which is a *primary key field*
  - a **title**, an **author** and **year** of publication field, that are an ordinary string field with maximum length defined

```
class Book(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    author = db.Column(db.String(100), nullable=False)
    published = db.Column(db.String(100), nullable=False)
```

Database model

- Setup the schema for our model, parsing book object(s) into a JSON response
- Make use of `flask_marshmallow` package, an integration layer for Flask and marshmallow
    - **marshmallow** is a Python library that converts complex data types to native Python data types and vice versa
    - how install it: `pip install flask-marshmallow`

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_marshmallow import Marshmallow
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db'
db = SQLAlchemy(app)
ma = Marshmallow(app)
```

Database model

- Create a new marshmallow schema based on Book model
- In this schema choose what fields to expose to users
- If your model has some sensitive data, you may want to exclude it here
- Instantiate it in `books_schema` for serialize an array of books, otherwise use `book_schema`

```
class BookSchema(ma.Schema):
    class Meta:
        fields = ("id", "title", "author", "published")
        model = Book
book_schema = BookSchema()
books_schema = BookSchema(many=True)
```

RESTful Routes

- Define RESTful handler, using `Flask-RESTful` package, a set of tools that help us to construct a RESTful routes with object-oriented design
  - Install it: `pip install flask-restful`
- Setup Flask-RESTful extension to get up and running in Flask server

```
from flask_restful import Api, Resource # new

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///test.db'
db = SQLAlchemy(app)
api = Api(app)
```

RESTful Routes

- Create a new RESTful resource with a **GET method** and make query to fetch all books with Book model
- Use `books_schema` to serialize the data from database and return it as a response to the client
- Register resource by using `api.add_resource` method and define the route endpoint

```
class BookListResource(Resource):
    def get(self):
        books = Book.query.all()
        return books_schema.dump(books)

api.add_resource(BookListResource, '/books')
```

- Start the server, send a request to **`/books`** endpoint, and you will get an empty array

```
curl http://localhost:5000/books
```

RESTful Routes

- Create a new **POST method**, instantiate a new post object with the request data, and save the record to the database.
  - return the post data as the response to the client.

```
def post(self):
    new_book = Book(id=request.json['id'], title=request.json['title'],
     author=request.json['author'], published=request.json['published'])
    db.session.add(new_book)
    db.session.commit()
    return book_schema.dump(new_book)
api.add_resource(BookListResource, '/books')
```

- Send a **POST request** with a book data

```
curl -X POST -H "Content-Type: application/json" -d ‘{ "id": 0, "title":
"The Old Man and the Sea", "author": "Ernest Hemingway", "published":
"1952" }’ http://localhost:5000/books
```

RESTful Routes

- Define **GET request** that, instead of querying all posts, fetch a single post with the given id
  - if it not exist, it will raise a *404 error*

```
class BookResource(Resource):
    def get(self, book_id):
        book = Book.query.get_or_404(book_id)
        return book_schema.dump(book)

api.add_resource(BookResource, '/books/<int:book_id>')
```

- To get the book with a specific id

```
curl http://localhost:5000/books/1
```

RESTful Routes

- In the **PUT method**, get the post object if exist, then update the properties which defined in the request body (`request.json`)
- For this reason check both properties with `in` expression
  - save the changes to the database by using the `db.session.commit()` and send the update data to the client

```
class BookResource(Resource):
    def put(self, book_id):
        book = Book.query.get_or_404(book_id)

        if 'id' in request.json:
            book.id = request.json['id']
        if 'title' in request.json:
            book.title = request.json['title']
        if 'author' in request.json:
            book.author =
                request.json['author']
        if 'published' in request.json:
            book.published =
                       request.json['published']
        db.session.commit()
        return book_schema.dump(book)
api.add_resource(BookResource,
                       '/books/<int:book_id>')
```

- Update book:

```
curl -X PUT -H "Content-Type: application/json"
-d '{ "id": 0, "title": "The Old Man and the
Sea", "author": "Ernest Hemingway", "published":
"1955" }' http://localhost:5000/books/0
```

RESTful Routes

- In the **DELETE method**, a specific object it is removed from the book object
- Save the changes and return nothing to the client (because there's nothing to show for)

```
class BookResource(Resource):
    def delete(self, book_id):
        book = Book.query.get_or_404(book_id)
        db.session.delete(book)
        db.session.commit()
        return '', 204

api.add_resource(BookResource, '/books/<int:book_id>')
```

- Delete book:

```
curl -i -X DELETE http://localhost:5000/books/0
```

CONNECTING DATABASE

Connecting API to a Database

- Suppose to have a **SQLite database** `book.db`
- Move it to the folder of your application
- book.db database has five columns: *id*, *published*, *author*, *title*, and *first_sentence*
  - each row represents one book

GET method

- `dict_factory` function defined returns items from the database as dictionaries rather than lists
- First, connect to the database using `sqlite3 library`
- Execute an SQL query to pull out all available data **(*)** from the books table of our database
    - this data is returned as **JSON**

```
def dict_factory(cursor, row):
    d = {}
    for idx, col in
        enumerate(cursor.description):
        d[col[0]] = row[idx]
    return d
```

```
@app.route('/books', methods=['GET'])
def books_all():
    conn = sqlite3.connect('books.db')
    conn.row_factory = dict_factory
    cur = conn.cursor()
    all_books = cur.execute('SELECT * FROM
books;').fetchall()

    return jsonify(all_books)
```

GET method

- Build an **SQL query** that will be used to find the requested information in the database
- A `to_filter list` is built: combined, the query and the the filters provided by the user will allow to pull the correct books from our database

```
@app.route('/books', methods=['GET'])
def api_filter():
    query_parameters = request.args
    id = query_parameters.get('id')
    published =
          query_parameters.get('published')
    author = query_parameters.get('author')
    query = "SELECT * FROM books WHERE"
    to_filter = []
    if id:
        query += ' id=? AND'
        to_filter.append(id)
    if published:
        query += ' published=? AND'
        to_filter.append(published)
```

```
    if author:
        query += ' author=? AND'
        to_filter.append(author)
    if not (id or published or author):
        return page_not_found(404)
    query = query[:-4] + ';'
    conn = sqlite3.connect('books.db')
    conn.row_factory = dict_factory
    cur = conn.cursor()
    results = cur.execute(query,
            to_filter).fetchall()
    return jsonify(results)
app.run()
    return d
```

DEVELOPING AN APPLICATION

Developing an application

- How to set up a basic **CRUD app** (create, read, update, and delete) with *Vue* and *Flask*

- First, create a new Vue application with the **Vue CLI**

- Then, perform the basic CRUD operations through a **back-end RESTful API** powered by Python and Flask

What is Flask?

- **Already know it!!**

- A brief recap:
  - Flask is a simple, yet powerful micro **web framework for Python**, perfect for building *RESTful APIs*
  - it is *minimal* and *flexible*, in such a way as to build up small or more complex app

What is Vue?

- **Vue** is an open-source JavaScript framework used for building user interfaces

- It adopted some of the best practices from *React* and *Angular*
  - compared to them it's much **more approachable**
  - beginners can get up and running quickly

- It is also **powerful**, so it provides all the features needed to create modern front-end applications

Flask Setup

- Create a new project directory:

  -
    ```
    mkdir flask-vue-crud
    cd flask-vue-crud
    ```
- Within "flask-vue-crud", create a new directory called "server"
  - `mkdir server`
- Install Flask along with the **Flask-CORS extension**:
  - `pip install flask`
  - `pip install flask-cors`

Flask Setup

```
from flask import Flask, jsonify
from flask_cors import CORS
# configuration
DEBUG = True
# instantiate the app
app = Flask(__name__)
app.config.from_object(__name__)
# enable CORS
CORS(app, resources={r'/*': {'origins': '*'}})
# sanity check route
if __name__ == '__main__':
    app.run()
```

Vue Setup

- Use Vue CLI to generate a customized project boilerplate
- Install it globally:
  - `npm install -g @vue/cli@4.5.11`
- Then, within "flask-vue-crud", initialize a **new Vue project** called client
  - `vue create client`

- This will require you to answer a few questions about the project
  - press enter again to configure the project structure and install the dependencies

Vue Setup

- A lots of files and folders are created (deal only with the "src" folder)
- `index.html` file is the starting point of Vue application

```
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.title %> doesn't work properly
without JavaScript enabled. Please enable it to continue.</strong>
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

Vue Setup

- Fire up the development server

  -
    ```
    cd client
    npm run serve
    ```

- Navigate to http://localhost:8080 in the browser
  - the welcome page of **Vue.js app** is showed

Vue Setup

- To connect the client-side Vue app with the back-end Flask app, use the **axios library** to send *AJAX requests*
- Install it:
  - `npm install axios@0.21.1 --save`
- To import it in the code:
  - `import axios from 'axios'`

Bootstrap Setup

- Next, add **Bootstrap**, a popular *CSS framework* in order to quickly add some style

- Install:
  - `npm install bootstrap@4.6.0 --save`

- To import the **Bootstrap styles** in the code:
  - `import 'bootstrap/dist/css/bootstrap.css';`

Book.vue

- Add a new component to the "client/src/components" folder called **Book.vue:**

```
<template>
  <div class="container">
    <div class="row">
      <div class="col-sm-10">
        <h1>Books</h1>
        <hr><br><br>
        <button type="button" class="btn btn-success
        btn-sm">Add Book</button>
        <br><br>
        <table class="table table-hover">
          <thead>
            <tr>
              <th scope="col">Title</th>
              <th scope="col">Author</th>
              <th scope="col">Read?</th>
              <th></th>
            </tr>
          </thead>
          <tbody>
             <tr>
            <td>foo</td>
            <td>bar</td>
            <td>foobar</td>
            <td>
              <div class="btn-group" role="group">
                <button type="button" class="btn
                 btn-warning btn-sm">Update</button>
                <button type="button" class="btn
                 btn-danger btn-sm">Delete</button>
              </div>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
 </div>
</template>
```

Book.vue

- Update in the router folder `index.js`

```
import Vue from 'vue';
import Router from 'vue-router';
import Books from '../components/Books.vue';
import Ping from '../components/Ping.vue';
Vue.use(Router);
export default new Router({
  mode: 'history',
  base: process.env.BASE_URL,
  routes: [
    {
      path: '/',
      name: 'Books',
      component: Books,
    },
  ],
});
```

What are we building?

- **Goal:** design a **back-end RESTful API**, powered by Python and Flask, for a single resource: *Books*

- The API itself should follow **RESTful design principles**, using the basic *HTTP verbs*: **GET**, **POST**, **PUT**, and **DELETE**

- Set up a **front-end application** with **Vue** that consumes the back-end API

GET Route - Server

- Add a list of books to `server/app.py`:

```
BOOKS = [
    {
        'id': 0,
        'title': 'Endymion',
        'author': 'John Keats',
        'published': '1818'
    },
    {
        'id': 1,
        'title': 'Romeo and Juliet', ]
        'author': 'William Shakespea
        'published': '1597'

    },
```

```
{
    'id': 2,
    'title': 'The great Gatsby',
    'author': 'Francis Scott
                  Fitzgerald',
    'published': '1925'
 }
```

GET Route - Server

- Add the route handler:

```
@app.route('/books', methods=['GET'])
 def all_books():
        return jsonify({
              'status': 'success',
              'books': BOOKS
         })
```

- Run the Flask app and then manually test out the route at http://localhost:5000/books

Client

- Update the component:

```
<template>
  <div class="container">
    <div class="row">
      <div class="col-sm-10">
        <h1>Books</h1>
        <hr><br><br>
        <button type="button" class="btn btn-success
          btn-sm">Add Book</button>
        <br><br>
        <table class="table table-hover">
          <thead>
            <tr>
              <th scope="col">Id</th>
              <th scope="col">Title</th>
              <th scope="col">Author</th>
              <th scope="col">Published</th>
              <th></th>
            </tr>
          </thead>
          <tbody>
            <tr v-for="(book, index) in books" :key="index">
              <td>{{ book.id }}</td>
              <td>{{ book.title }}</td>
              <td>{{ book.author }}</td>
              <td>{{ book.published }}</td>
              <td>
                <span v-if="book.read">Yes</span>
                <span v-else>No</span>
              </td>
              <td>
                <div class="btn-group" role="group">
                  <button type="button" class="btn
                  btn-warning btn-sm">Update</button>
                  <button type="button" class="btn
                  btn-danger btn-sm">Delete</button>
                </div>
              </td>
            </tr>
          </tbody>
        </table>
      </div>
    </div>
  </div>
</template>
<script>
```

Client

```
import axios from 'axios';
export default {
  data() {
    return {
      books: [],
    };
  },
  methods: {
    getBooks() {
      const path =
'http://localhost:5000/books';
      axios.get(path)
```

- The `getBooks()` method is called via the created lifecycle hook, which fetches the books from the back-end endpoint just set up

```
.then((res) => {
          this.books =
res.data.books;
        })
        .catch((error) => {
          //
eslint-disable-next-line
          console.error(error);
        });
    },
  },
  created() {
    this.getBooks();

  },
};
</script>
```

Client

- Enable the **Bootstrap Vue library** in `client/src/main.js`:

```
import BootstrapVue from 'bootstrap-vue';
import Vue from 'vue';
import App from './App.vue';
import router from './router';
import 'bootstrap/dist/css/bootstrap.css';

Vue.use(BootstrapVue);
Vue.config.productionTip = false;

new Vue({
  router,
  render: (h) => h(App),
}).$mount('#app');
```

POST Route - Server

- Update the existing route handler to handle **POST requests** for adding a new book:

```
from flask import Flask, jsonify, request
@app.route('/books', methods=['GET', 'POST'])
def all_books():
    response_object = {'status': 'success'}
    if request.method == 'POST':
        post_data = request.get_json()
        BOOKS.append({
            'id': post_data.get('id'),
            'title': post_data.get('title'),
            'author': post_data.get('author'),
            'published': post_data.get('published')
        })
        response_object['message'] = 'Book added!'
    else:
        response_object['books'] = BOOKS
    return jsonify(response_object)
```

POST Route - Server

- With the Flask server running, you can test the POST route in a new terminal tab:

  -
    ```
    curl -X POST http://localhost:5000/books -d \  '{"id": 1, "title": "1Q84", "author": "Haruki Murakami", "published": "2009"}' \  -H 'Content-Type: application/json'
    ```

- You should see:

  -
    ```
    {
      "message": "Book added!",
      "status": "success"
     }
    ```

- You should also see the new book in the response from the http://localhost:5000/books endpoint

Client

- On the client-side, define that modal for adding a new book to the `Books` component, starting with the HTML:

```
<b-modal ref="addBookModal"
         id="book-modal"
         title="Add a new book"
         hide-footer>
  <b-form @submit="onSubmit" @reset="onReset" class="w-100'
  <b-form-group id="form-id-group"
                label="Id:"
                label-for="form-id-input">
      <b-form-input id="form-id-input"
                    type="text"
                    v-model="addBookForm.id"
                    required
                    placeholder="Enter id">
      </b-form-input>
    </b-form-group>
```

```
<b-form-group id="form-title-group"
              label="Title:"
              label-for="form-title-input">

    <b-form-input id="form-title-input"
                  type="text”
                  v-model="addBookForm.title"
                  required
                  placeholder="Enter title">
    </b-form-input>
  </b-form-group>
```

Client

```
<b-form-group id="form-author-group"
              label="Author:"
              label-for="form-author-input">
      <b-form-input id="form-author-input"
                    type="text"
                    v-model="addBookForm.author"
                       required
                       placeholder="Enter author">
  </b-form-input>
  </b-form-group>
<b-form-group id="form-published-group"
                label="Published:"
                label-for="form-published-input">
      <b-form-input id="form-published-input"
                     type="text"
                     v-model="addBookForm.published"
                     required
                     placeholder="Enter published year">
      </b-form-input>
    </b-form-group>
```

```
    <b-button type="submit"
variant="primary">Submit</b-button>
    <b-button type="reset"
variant="danger">Reset</b-button>
  </b-form>
</b-modal>
```

Client

```
<script>
import axios from 'axios';
export default {
  data() {
    return {
      books: [],
      addBookForm: {
         id: '',
        title: '',
        author: '',
        published: '',
      },
    };
  },
  methods: {
    getBooks() {
      const path = 'http://localhost:5000/books';
      axios.get(path)
        .then((res) => {
          this.books = res.data.books;
        })
       .catch((error) => {
          console.error(error);
        });
    },
    addBook(payload) {
      const path = 'http://localhost:5000/books';
      axios.post(path, payload)
        .then(() => {
          this.getBooks();
        })
        .catch((error) => {
          // eslint-disable-next-line
          console.log(error);
          this.getBooks();
        });
    },
    initForm() {
      this.addBookForm.id = '';
      this.addBookForm.title = '';
      this.addBookForm.author = '';
      this.addBookForm.published = '';
    },
```

Client

```
onSubmit(evt) {
      evt.preventDefault();
      this.$refs.addBookModal.hide();

      if (this.addBookForm.read[0]);
      const payload = {
        id: this.addBookForm.id,
        title: this.addBookForm.title,
        author: this.addBookForm.author,
        published
      };
      this.addBook(payload);
      this.initForm();
    },
    onReset(evt) {
      evt.preventDefault();
      this.$refs.addBookModal.hide();
      this.initForm();
    },
  },
```

```
      created() {
        this.getBooks();
      },
  };
</script>
```

- `addBookForm` is two-way binding function, when one is updated, the other will be updated as well
- `onSubmit` function for when the user submits the form successfully
- `addBook` sends a POST request to `/books` to add a new book
- update also the **"Add Book"** button in the template so that the modal is displayed when the button is clicked:
  ```
  <button type="button" class="btn
  btn-success btn-sm"
  v-b-modal.book-modal>Add
  Book</button>
  ```

PUT Route - Server

- Update BOOKS in `server/app.py`:

```
@app.route('/books/<book_id>', methods=['PUT'])
def single_book(book_id):
    response_object = {'status': 'success'}
    if request.method == 'PUT':
        post_data = request.get_json()
        remove_book(book_id)
        BOOKS.append({
            'id': post_data.get('id'),
            'title': post_data.get('title'),
            'author': post_data.get('author'),
            'published': post_data.get('published')
        })
        response_object['message'] = 'Book updated!'
    return jsonify(response_object)
```

PUT Route - Server

- Add the helper:

```
def remove_book(book_id):
    for book in BOOKS:
        if book['id'] == book_id:
            BOOKS.remove(book)
            return True
    return False
```

Client - Add modal

- Add a new modal to the template:

```
<b-modal ref="editBookModal"
         id="book-update-modal"
         title="Update"
         hide-footer>
  <b-form @submit="onSubmitUpdate"
@reset="onResetUpdate"
   class="w-100">
<b-form-group id="form-id-edit-group"
                label="Id:"
                label-for="form-id-edit-input">
      <b-form-input id="form-id-edit-input"
                    type="text"
                    v-model="editForm.id"
                    required
                    placeholder="Enter id">
      </b-form-input>
    </b-form-group>
  <b-form-group id="form-title-edit-group"
                label="Title:"
label-for="form-title-edit-input">
      <b-form-input id="form-title-edit-input"
                    type="text"
                    v-model="editForm.title"
                    required
                    placeholder="Enter title">
      </b-form-input>
    </b-form-group>
    <b-form-group id="form-author-edit-group"
                  label="Author:"

label-for="form-author-edit-input">
```

Client - Add modal

```
<b-form-input id="form-author-edit-input"
                      type="text"
                      v-model="editForm.author"
                      required
                      placeholder="Enter author">
        </b-form-input>
      </b-form-group>
<b-form-group id="form-published-edit-group"
                label="Published:"
                label-for="form-published-edit-input">
      <b-form-input id="form-published-edit-input"
                    type="text"
                    v-model="editForm.published"
                    required
                    placeholder="Enter published year">
      </b-form-input>
    </b-form-group>
        <b-button type="submit" variant="primary">Update</b-button>
      <b-button type="reset" variant="danger">Cancel</b-button>
    </b-button-group>
  </b-form>
</b-modal>
```

Client - Add form and update button click

- Add the form state to the data part of the script section:

```
editForm: {
  id: '',
  title: '',
  author: '',
  published: [],
},
```

- Update the "update" button in the table:

```
<button
        type="button"
        class="btn btn-warning btn-sm"
        v-b-modal.book-update-modal
        @click="editBook(book)">
    Update
</button>
```

- Add a new method to update the values in editForm:

```
editBook(book) {
  this.editForm = book;
},
```

- Then, add a method to handle the form submit:

```
onSubmitUpdate(evt) {
  evt.preventDefault();
  this.$refs.editBookModal.hide();
  let read = false;
  if (this.editForm.read[0]) read = true;
  const payload = {
    id: this.editForm.id,
    title: this.editForm.title,
    author: this.editForm.author,
    published: this.editForm.published,
  };
  this.updateBook(payload, this.editForm.id);
},
```

Client - Wire up AJAX request

- Add a method to handle the form submit:

```
onSubmitUpdate(evt) {
  evt.preventDefault();
  this.$refs.editBookModal.hide();
  let read = false;
  if (this.editForm.read[0]) read =
true;
  const payload = {
    id: this.editForm.id,
    title: this.editForm.title,
    author: this.editForm.author,
    published:
this.editForm.published,
  };
  this.updateBook(payload,
this.editForm.id);
},
```

- Wire up AJAX request:

```
updateBook(payload, bookID) {
  const path =
`http://localhost:5000/books/${bookID}
`;
  axios.put(path, payload)
    .then(() => {
      this.getBooks();
    })
    .catch((error) => {
      // eslint-disable-next-line
      console.error(error);
      this.getBooks();
    });
},
```

Client - Handle cancel button click

- Add method:

```
onResetUpdate(evt) {
  evt.preventDefault();
  this.$refs.editBookModal.hide();
  this.initForm();
  this.getBooks();
},
```

- Update `initForm`:

```
initForm() {
  this.addBookForm.id = '';
  this.addBookForm.title = '';
  this.addBookForm.author = '';
  this.addBookForm.published = '';
  this.editForm.id = '';
  this.editForm.title = '';
  this.editForm.author = '';
  this.editForm.published = '';
},
```

DELETE Route - Server

- Update the route handler:

```
@app.route('/books/<book_id>', methods=['PUT', 'DELETE'])
def single_book(book_id):
    response_object = {'status': 'success'}
    if request.method == 'PUT':
        post_data = request.get_json()
        remove_book(book_id)
        BOOKS.append({
            'id': post_data.get('id'),
            'title': post_data.get('title'),
            'author': post_data.get('author'),
            'published': post_data.get('published')
        })
        response_object['message'] = 'Book updated!'
    if request.method == 'DELETE':
        remove_book(book_id)
        response_object['message'] = 'Book removed!'
    return jsonify(response_object)
```

Client

- Update the **"delete"** button:

```
<button
        type="button"
        class="btn btn-danger btn-sm"
        @click="onDeleteBook(book)">
    Delete
</button>
```

- Add the methods to handle the *button click* and then remove the book:

```
removeBook(bookID) {
  const path = `http://localhost:5000/books/${bookID}`;
  axios.delete(path)
    .then(() => {
      this.getBooks();
      this.message = 'Book removed!';
      this.showMessage = true;
this.removeBook(book.id);
    })
    .catch((error) => {
      // eslint-disable-next-line
```

```
          this.getBooks();
    });
  },
  onDeleteBook(book)
        {

   },
```