# Setup a Vue+flask

## How to build a Single Page App with Flask and Vue.js

The following is a step-by-step walkthrough of how to set up a basic CRUD app with Vue and Flask. We'll start by scaffolding a new Vue application with the Vue CLI and then move on to performing the basic CRUD operations through a back-end RESTful API powered by Python and Flask.

### Objectives

By the end of this tutorial, you will be able to:

1. Explain what Flask is
2. Explain what Vue is and how it compares to other UI libraries and front-end frameworks like React and Angular
3. Scaffold a Vue project using the Vue CLI
4. Create and render Vue components in the browser
5. Create a Single Page Application (SPA) with Vue components
6. Connect a Vue application to a Flask back-end
7. Develop a RESTful API with Flask
8. Style Vue Components with Bootstrap
9. Use the Vue Router to create routes and render components

*Main dependencies:*

- Vue v2.6.10
- Vue CLI v3.7.0
- Node v12.1.0
- npm v6.9.0
- Flask v1.0.2
- Python v3.7.3

*Final app*:

### What is Flask?

Flask is a simple, yet powerful micro web framework for Python, perfect for building RESTful APIs. Like Sinatra (Ruby) and Express (Node), it's minimal and flexible, so you can start small and build up to a more complex app as needed.

### 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. That said, compared to React and Angular, it's much more

approachable, so beginners can get up and running quickly. It's also just as powerful, so it provides all the features you'll need to create modern front-end applications.

## Flask Setup

Begin by creating a new project directory:

```
$ mkdir flask-vue-crud
$ cd flask-vue-crud
```

Within "flask-vue-crud", create a new directory called "server". Then, create and activate a virtual environment inside the "server" directory:

```
$ python3.7 -m venv env
$ source env/bin/activate
```

Install Flask along with the Flask-CORS extension:

```
(env)$ pip install Flask==1.0.2 Flask-Cors==3.0.7
```

Add an app.py file to the newly created "server" directory:

```
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
@app.route('/ping', methods=['GET'])
def ping_pong():
    return jsonify('pong!')


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

Why do we need Flask-CORS? In order to make cross-origin requests

Run the app:

```
(env)$ python app.py
```

To test, point your browser at http://localhost:5000/ping. You should see:

```
"pong!"
```

Back in the terminal, press Ctrl+C to kill the server and then navigate back to the project root. With that, let's turn our attention to the front-end and get Vue set up.

## Vue Setup

We'll be using the powerful Vue CLI to generate a customized project boilerplate.

Install it globally:

```
$ npm install -g @vue/cli@3.7.0
```

Then, within "flask-vue-crud", run the following command to initialize a new Vue project called `client`:

```
$ vue create client
```

This will require you to answer a few questions about the project. Use the down arrow key to highlight "Manually select features", and then press enter. Next, you'll need to select the features you'd like to install. For this tutorial, select "Babel", "Router", and "Linter / Formatter" like so:

```
Vue CLI v3.7.0
? Please pick a preset: Manually select features
? Check the features needed for your project:
 ◉ Babel
 ◯ TypeScript
 ◯ Progressive Web App (PWA) Support
❯◉ Router
 ◯ Vuex
 ◯ CSS Pre-processors
 ◉ Linter / Formatter
 ◯ Unit Testing
 ◯ E2E Testing
```

Use the history mode for the router. Select "ESLint + Airbnb config" for the linter and "Lint on save". Finally, select the "In package.json" option so that configuration is placed in the *package.json* file instead of in separate configuration files.

You should see something similar to:

```
Vue CLI v3.7.0
? Please pick a preset: Manually select features
? Check the features needed for your project: Babel, Router, Linter
? Use history mode for router? Yes
? Pick a linter / formatter config: Airbnb
? Pick additional lint features: Lint on save
? Where do you prefer placing config for Babel, PostCSS, ESLint, etc.? In package.json
? Save this as a preset for future projects? (y/N) No
```

Take a quick look at the generated project structure. It may seem like a lot, but we'll *only* be dealing with the files and folders in the "src" folder along with the *index.html* file found in the "public" folder.

The *index.html* file is the starting point of our Vue application.

```
<!DOCTYPE html>
<html lang="en">
  <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>client</title>
  </head>
  <body>
    <noscript>
      <strong>
        We're sorry but client 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>
```

Take note of the `<div>` element with an `id` of `app`. This is a placeholder that Vue will use to attach the generated HTML and CSS to produce the UI.

Turn your attention to the folders inside the "src" folder:

```
├── App.vue
├── assets
│   └── logo.png
├── components
│   └── HelloWorld.vue
├── main.js
├── router.js
└── views
    ├── About.vue
    └── Home.vue
```

Breakdown:

| Name | Purpose |
| --- | --- |
| main.js | app entry point, which loads and initializes Vue along with the root component |
| App.vue | Root component, which is the starting point from which all other components will be rendered |
| "components" | where UI components are stored |
| router.js | where URLS are defined and mapped to components |
| "views" | where UI components that are tied to the router are stored |
| "assets" | where static assets, like images and fonts, are stored |

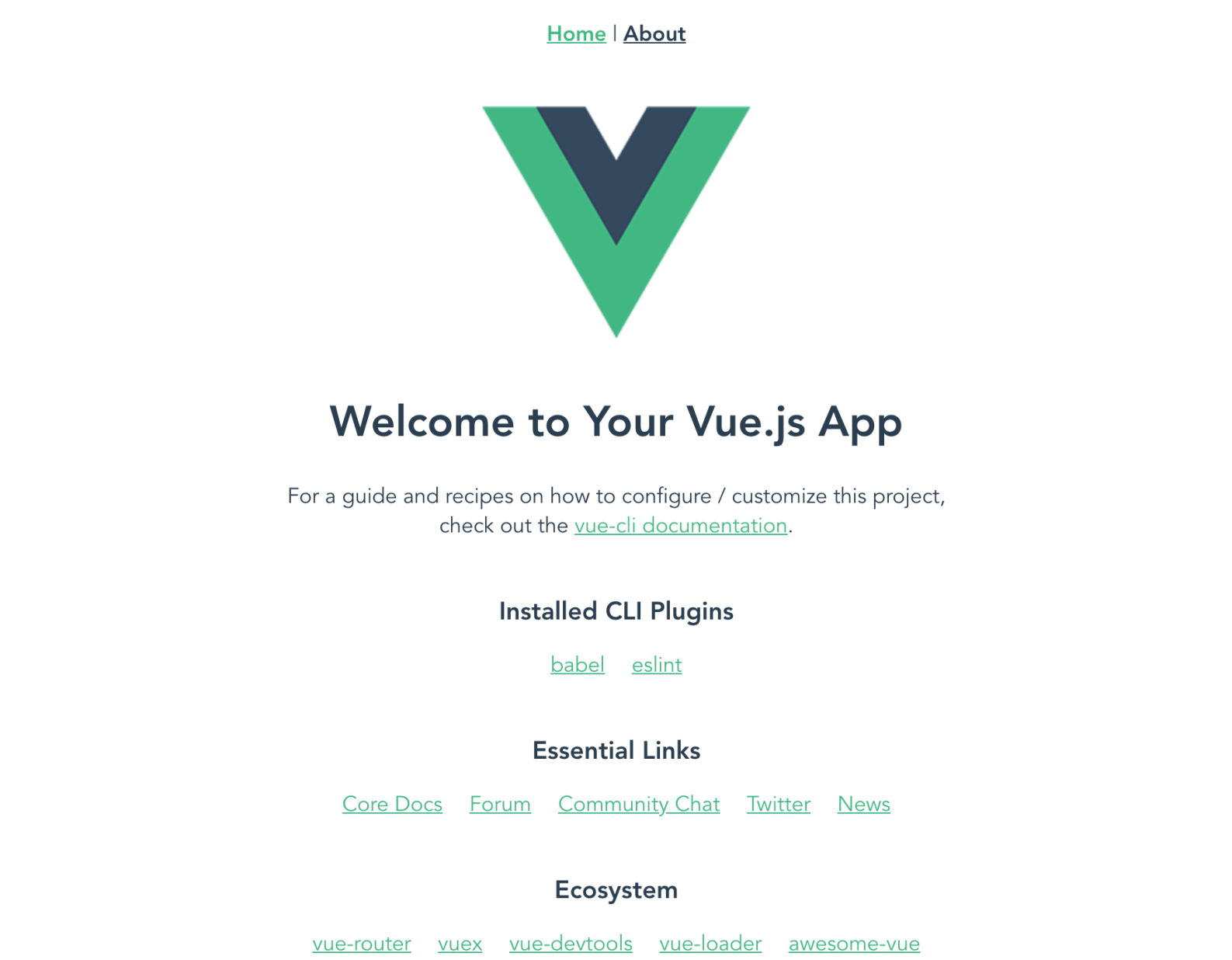
Review the *client/src/components/HelloWorld.vue* file. This is a Single File component, which is broken up into three different sections:

1. *template*: for component-specific HTML
2. *script*: where the component logic is implemented via JavaScript
3. *style*: for CSS styles

Fire up the development server:

```
$ cd client
$ npm run serve
```

Navigate to http://localhost:8080 in the browser of your choice. You should see the following:




To simplify things, remove the "views" folder. Then, add a new component to the "client/src/components" folder called *Ping.vue*:

```
<template>
  <div>
    <p>{{ msg }}</p>
  </div>
</template>

<script>
export default {
  name: 'Ping',
  data() {
    return {
      msg: 'Hello!',
    };
  },
```

```
};
</script>
```

Update *client/src/router.js* to map '/ping' to the `Ping` component like so:

```
import Vue from 'vue';
import Router from 'vue-router';
import Ping from './components/Ping.vue';

Vue.use(Router);

export default new Router({
  mode: 'history',
  base: process.env.BASE_URL,
  routes: [
    {
      path: '/ping',
      name: 'Ping',
      component: Ping,
    }
  ],
});
```

Finally, within *client/src/App.vue*, remove the navigation:

```
<template>
  <div id="app">
    <router-view/>
  </div>
</template>
```

You should now see `Hello!` in the browser at http://localhost:8080/ping.

To connect the client-side Vue app with the back-end Flask app, we can use the axios library to send AJAX requests. Start by installing it:

```
$ npm install axios@0.18.0 --save
```

Update the `script` section of the component, in *Ping.vue*, like so:

```
<script>
import axios from 'axios';

export default {
  name: 'Ping',
  data() {
    return {
      msg: '',
    };
  },
  methods: {
    getMessage() {
      const path = 'http://localhost:5000/ping';
      axios.get(path)
        .then((res) => {
          this.msg = res.data;
        })
        .catch((error) => {
          // eslint-disable-next-line
          console.error(error);
        });
    },
  },
  created() {
```

```
    this.getMessage();
  },
};
</script>
```

Fire up the Flask app in a new terminal window. You should see `pong!` in the browser. Essentially, when a response is returned from the back-end, we set `msg` to the value of `data` from the response object.

## Bootstrap Setup

Next, let's add Bootstrap, a popular CSS framework, to the app so we can quickly add some style.

Install:

```
$ npm install bootstrap@4.3.1 --save
```

> Ignore the warnings for jquery and popper.js. Do NOT add either to your project. More on this later.

Import the Bootstrap styles to *client/src/main.js*:

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

Vue.config.productionTip = false;

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

Update the `style` section in *client/src/App.vue*:

```
<style>
#app {
  margin-top: 60px
}
</style>
```

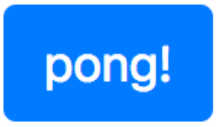
Ensure Bootstrap is wired up correctly by using a Button and Container in the `Ping` component:

```
<template>
  <div class="container">
    <button type="button" class="btn btn-primary">{{ msg }}</button>
  </div>
</template>
```

Run the dev server:

```
$ npm run serve
```

You should see:




Next, add a new component called `Books` in a new file called *Books.vue*:

```
<template>
  <div class="container">
    <p>books</p>
  </div>
</template>
```

Update the router:

```
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,
    },
    {
      path: '/ping',
      name: 'Ping',
      component: Ping,
    },
  ],
});
```

Test:

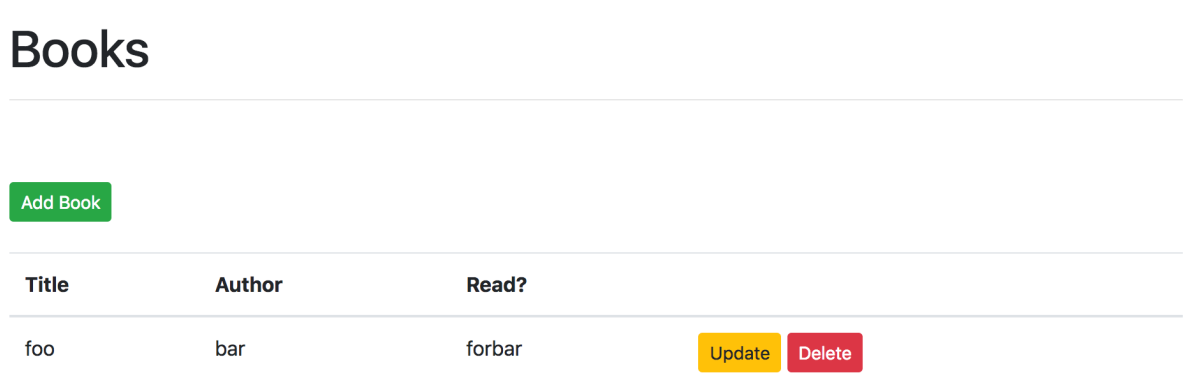
1. http://localhost:8080
2. http://localhost:8080/ping

Finally, let's add a quick, Bootstrap-styled table to the `Books` 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">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>
```

You should now see:




Now we can start building out the functionality of our CRUD app.

## What are we building?

Our goal is to 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.

We'll also set up a front-end application with Vue that consumes the back-end API:

> This tutorial only deals with the happy path. Handling errors is a separate exercise. Check your understanding and add proper error handling on both the front and back-end.

## GET Route

### Server

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

```
BOOKS = [
    {
        'title': 'On the Road',
        'author': 'Jack Kerouac',
        'read': True
    },
    {
        'title': 'Harry Potter and the Philosopher\'s Stone',
        'author': 'J. K. Rowling',
        'read': False
    },
    {
        'title': 'Green Eggs and Ham',
        'author': 'Dr. Seuss',
        'read': True
    }
]
```

Add the route handler:

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

Run the Flask app, if it's not already running, and then manually test out the route at http://localhost:5000/books.

> Looking for an extra challenge? Write an automated test for this. Review this resource for more info on testing a Flask app.

### 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">Title</th>
              <th scope="col">Author</th>
              <th scope="col">Read?</th>
              <th></th>
            </tr>
          </thead>
          <tbody>
            <tr v-for="(book, index) in books" :key="index">
              <td>{{ book.title }}</td>
              <td>{{ book.author }}</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>
```

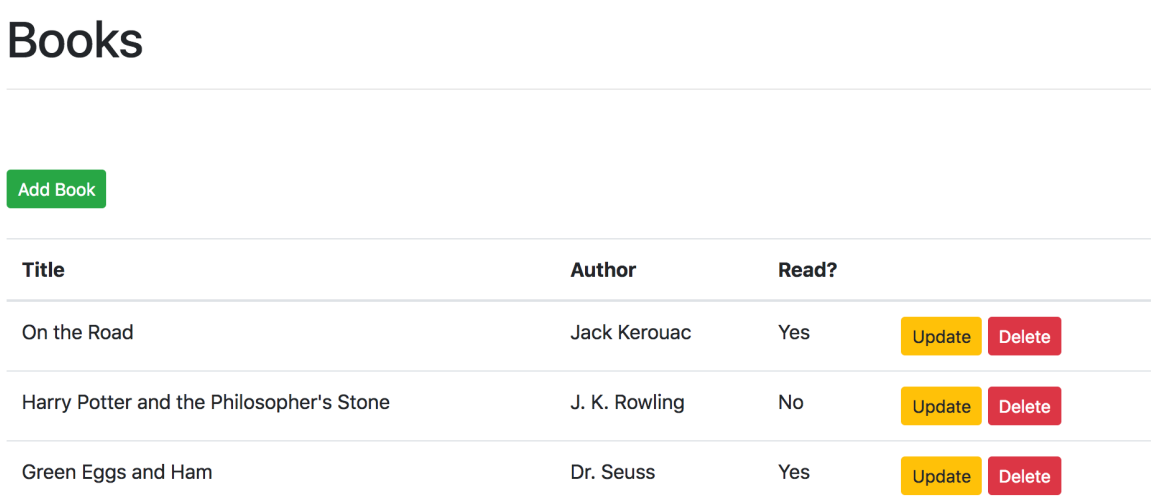
```
                  <button type="button" class="btn btn-danger btn-sm">Delete</button>
                </div>
              </td>
            </tr>
          </tbody>
        </table>
      </div>
    </div>
  </div>
</template>

<script>
import axios from 'axios';

export default {
  data() {
    return {
      books: [],
    };
  },
  methods: {
    getBooks() {
      const path = 'http://localhost:5000/books';
      axios.get(path)
        .then((res) => {
          this.books = res.data.books;
        })
        .catch((error) => {
          // eslint-disable-next-line
          console.error(error);
        });
    },
  },
  created() {
    this.getBooks();
  },
};
</script>
```

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

In the template, we iterated through the list of books via the v-for directive, creating a new table row on each iteration. The index value is used as the key. Finally, v-if is then used to render either `Yes` or `No`, indicating whether the user has read the book or not.




## Bootstrap Vue

In the next section, we'll use a modal to add a new book. We'll add on the Bootstrap Vue library for this, which provides a set of Vue components styled with Bootstrap-based HTML and CSS.

> Why Bootstrap Vue? Bootstrap's Modal component uses jQuery, which you should avoid using with Vue together in the same project since Vue uses the Virtual Dom to update the DOM. In other words, if you did use jQuery to manipulate the DOM, Vue would not know about it. At the very least, if you absolutely need to use jQuery, do not use Vue and jQuery together on the same DOM elements.

Install:

```
$ npm install bootstrap-vue@2.0.0-rc.19 --save
```

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

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

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:

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

Update the imports:

```
from flask import Flask, jsonify, request
```

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

```
$ curl -X POST http://localhost:5000/books -d \
  '{"title": "1Q84", "author": "Haruki Murakami", "read": "true"}' \
  -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/booksendpoint.

## Client

On the client-side, let's add that modal now 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-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>
    <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-read-group">
      <b-form-checkbox-group v-model="addBookForm.read" id="form-checks">
        <b-form-checkbox value="true">Read?</b-form-checkbox>
      </b-form-checkbox-group>
    </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>
```

Add this just before the closing `div` tag. Take a quick look at the code. `v-model` is a directive used to bind input values back to the state. You'll see this in action shortly.

Update the `script` section:

```
<script>
import axios from 'axios';

export default {
```

```
  data() {
    return {
      books: [],
      addBookForm: {
        title: '',
        author: '',
        read: [],
      },
    };
  },
  methods: {
    getBooks() {
      const path = 'http://localhost:5000/books';
      axios.get(path)
        .then((res) => {
          this.books = res.data.books;
        })
        .catch((error) => {
          // eslint-disable-next-line
          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.title = '';
      this.addBookForm.author = '';
      this.addBookForm.read = [];
    },
    onSubmit(evt) {
      evt.preventDefault();
      this.$refs.addBookModal.hide();
      let read = false;
      if (this.addBookForm.read[0]) read = true;
      const payload = {
        title: this.addBookForm.title,
        author: this.addBookForm.author,
        read, // property shorthand
      };
      this.addBook(payload);
      this.initForm();
    },
    onReset(evt) {
      evt.preventDefault();
      this.$refs.addBookModal.hide();
      this.initForm();
    },
  },
  created() {
    this.getBooks();
  },
};
</script>
```

What's happening here?

1. `addBookForm` is bound to the form inputs via, again, `v-model`. When one is updated, the other will be updated as well, in other words. This is called two-way binding. Think about the ramifications of this. Do you think this makes state management easier or harder? How do React and Angular handle this? In my opinion, two-way binding (along with mutability) makes Vue much more approachable than React, but less scaleable in the long run.

2. `onSubmit` is fired when the user submits the form successfully. On submit, we prevent the normal browser behavior (`evt.preventDefault()`), close the modal (`this.$refs.addBookModal.hide()`), fire the `addBook` method, and clear the form (`initForm()`).
3. `addBook` sends a POST request to `/books` to add a new book.

Finally, update 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>
```

The component should now look like this:

```
<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" v-b-modal.book-modal>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 v-for="(book, index) in books" :key="index">
              <td>{{ book.title }}</td>
              <td>{{ book.author }}</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>
    <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-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>
        <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-read-group">
          <b-form-checkbox-group v-model="addBookForm.read" id="form-checks">
            <b-form-checkbox value="true">Read?</b-form-checkbox>
          </b-form-checkbox-group>
        </b-form-group>
        <b-button-group>
          <b-button type="submit" variant="primary">Submit</b-button>
          <b-button type="reset" variant="danger">Reset</b-button>
        </b-button-group>
      </b-form>
    </b-modal>
  </div>
</template>

<script>
import axios from 'axios';

export default {
  data() {
    return {
      books: [],
      addBookForm: {
        title: '',
        author: '',
        read: [],
      },
    };
  },
  methods: {
    getBooks() {
      const path = 'http://localhost:5000/books';
      axios.get(path)
        .then((res) => {
          this.books = res.data.books;
        })
        .catch((error) => {
          // eslint-disable-next-line
          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.title = '';
      this.addBookForm.author = '';
      this.addBookForm.read = [];
    },
    onSubmit(evt) {
      evt.preventDefault();
      this.$refs.addBookModal.hide();
      let read = false;
      if (this.addBookForm.read[0]) read = true;
      const payload = {
        title: this.addBookForm.title,
        author: this.addBookForm.author,
        read, // property shorthand
      };
      this.addBook(payload);
      this.initForm();
    },
```

```
    onReset(evt) {
      evt.preventDefault();
      this.$refs.addBookModal.hide();
      this.initForm();
    },
  },
  created() {
    this.getBooks();
  },
};
</script>
```

Test it out! Try adding a book:

## Alert Component

Next, let's add an Alert component to display a message to the end user after a new book is added. We'll create a new component for this since it's likely that you'll use the functionality in a number of components.

Add a new file called *Alert.vue* to "client/src/components":

```
<template>
  <p>It works!</p>
</template>
```

Then, import it into the `script` section of the `Books` component and register the component:

```
<script>
import axios from 'axios';
import Alert from './Alert.vue';

...

export default {
  data() {
    return {
      books: [],
      addBookForm: {
        title: '',
        author: '',
        read: [],
      },
    };
  },
  components: {
    alert: Alert,
  },

  ...

};
</script>
```

Now, we can reference the new component in the `template` section:

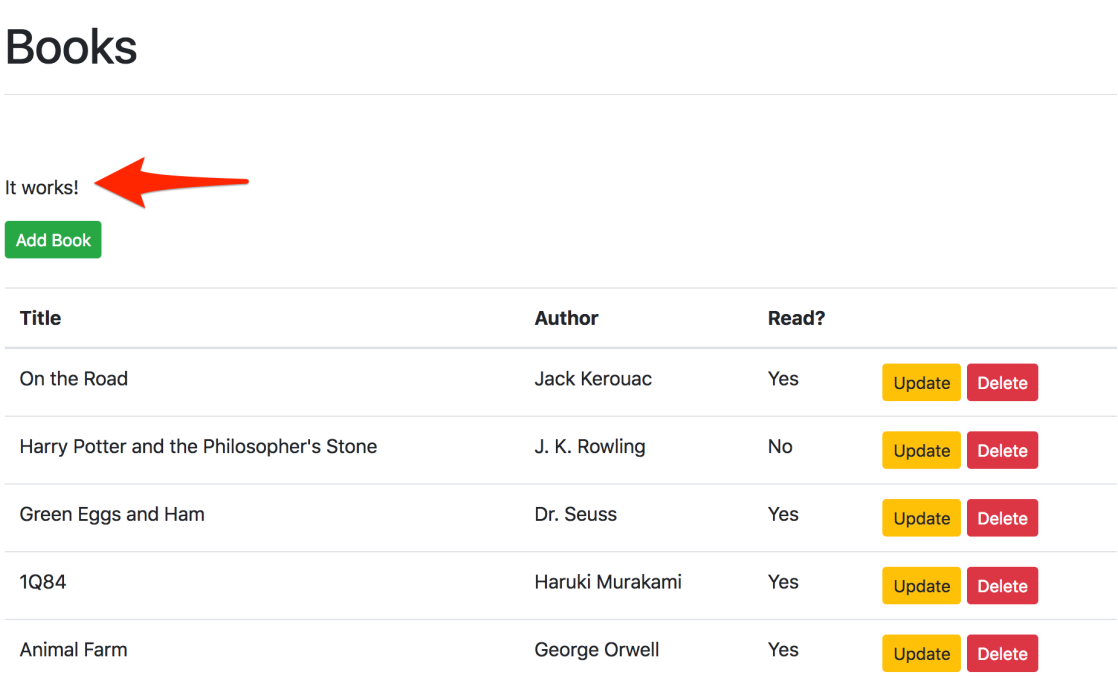
```
<template>
  <b-container>
    <b-row>
      <b-col col sm="10">
        <h1>Books</h1>
        <hr><br><br>
```

```
        <alert></alert>
        <button type="button" class="btn btn-success btn-sm" v-b-modal.book-modal>Add Book</button>

        ...

      </b-col>
    </b-row>
  </b-container>
</template>
```

Refresh the browser. You should now see:




Next, let's add the actual b-alert component to the template:

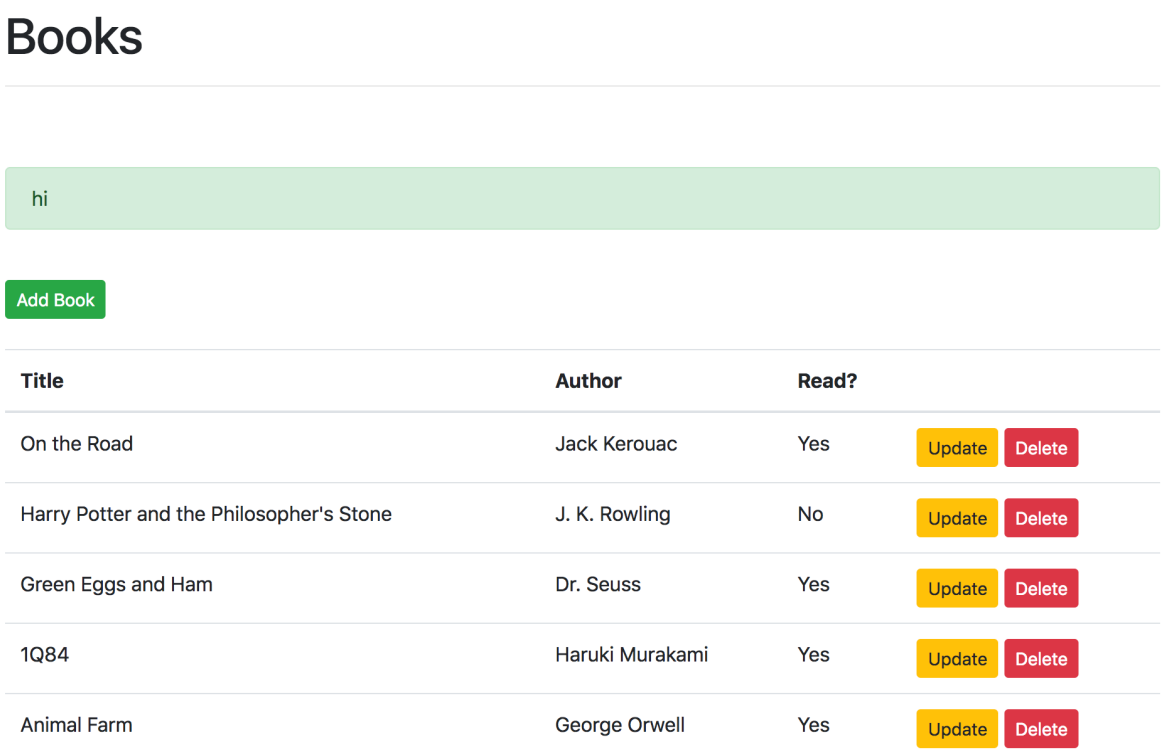
```
<template>
  <div>
    <b-alert variant="success" show>{{ message }}</b-alert>
    <br>
  </div>
</template>

<script>
export default {
  props: ['message'],
};
</script>
```

Take note of the props option in the `script` section. We can pass a message down from the parent component (`Books`) like so:

```
<alert message="hi"></alert>
```

Try this out:




To make it dynamic, so that a custom message is passed down, use a binding expression in *Books.vue*:

```
<alert :message="message"></alert>
```

Add the `message` to the `data` options, in *Books.vue* as well:

```
data() {
  return {
    books: [],
    addBookForm: {
      title: '',
      author: '',
      read: [],
    },
    message: '',
  };
},
```

Then, within `addBook`, update the message:

```
addBook(payload) {
  const path = 'http://localhost:5000/books';
  axios.post(path, payload)
    .then(() => {
      this.getBooks();
      this.message = 'Book added!';
    })
    .catch((error) => {
      // eslint-disable-next-line
      console.log(error);
      this.getBooks();
    });
},
```

Finally, add a `v-if`, so the alert is only displayed if `showMessage` is true:

```
<alert :message=message v-if="showMessage"></alert>
```

Add `showMessage` to the `data`:

```
data() {
  return {
    books: [],
    addBookForm: {
      title: '',
      author: '',
      read: [],
    },
    message: '',
    showMessage: false,
  };
},
```

Update `addBook` again, setting `showMessage` to `true`:

```
addBook(payload) {
  const path = 'http://localhost:5000/books';
  axios.post(path, payload)
    .then(() => {
      this.getBooks();
      this.message = 'Book added!';
      this.showMessage = true;
    })
    .catch((error) => {
      // eslint-disable-next-line
      console.log(error);
      this.getBooks();
    });
},
```

Test it out!

# PUT Route

## Server

For updates, we'll need to use a unique identifier since we can't depend on the title to be unique. We can use `uuid` from the Python standard library

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

```
BOOKS = [
    {
        'id': uuid.uuid4().hex,
        'title': 'On the Road',
        'author': 'Jack Kerouac',
        'read': True
    },
    {
        'id': uuid.uuid4().hex,
        'title': 'Harry Potter and the Philosopher\'s Stone',
        'author': 'J. K. Rowling',
        'read': False
    },
```

```
    {
        'id': uuid.uuid4().hex,
        'title': 'Green Eggs and Ham',
        'author': 'Dr. Seuss',
        'read': True
    }
]
```

Don't forget the import:

```
import uuid
```

Refactor `all_books` to account for the unique id when a new book is added:

```
@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': uuid.uuid4().hex,
            'title': post_data.get('title'),
            'author': post_data.get('author'),
            'read': post_data.get('read')
        })
        response_object['message'] = 'Book added!'
    else:
        response_object['books'] = BOOKS
    return jsonify(response_object)
```

Add a new route handler:

```
@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': uuid.uuid4().hex,
            'title': post_data.get('title'),
            'author': post_data.get('author'),
            'read': post_data.get('read')
        })
        response_object['message'] = 'Book updated!'
    return jsonify(response_object)
```

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

Steps:

1. Add modal and form

2. Handle update button click
3. Wire up AJAX request
4. Alert user
5. Handle cancel button click

### (1) Add modal and form

First, add a new modal to the template, just below the first modal:

```
<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-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">
        <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-read-edit-group">
      <b-form-checkbox-group v-model="editForm.read" id="form-checks">
        <b-form-checkbox value="true">Read?</b-form-checkbox>
      </b-form-checkbox-group>
    </b-form-group>
    <b-button-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>
```

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

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

### (2) Handle update button click

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 = {
    title: this.editForm.title,
    author: this.editForm.author,
    read,
  };
  this.updateBook(payload, this.editForm.id);
},
```

### (3) 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();
    });
},
```

### (4) Alert user

Update `updateBook`:

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

### (5) Handle cancel button click

Add method:

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

Update `initForm`:

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

Make sure to review the code before moving on. Once done, test out the application. Ensure the modal is displayed on the button click and that the input values are populated correctly.

## 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': uuid.uuid4().hex,
            'title': post_data.get('title'),
            'author': post_data.get('author'),
            'read': post_data.get('read')
        })
        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 like so:

```
<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;
    })
    .catch((error) => {
      // eslint-disable-next-line
      console.error(error);
      this.getBooks();
    });
},
onDeleteBook(book) {
  this.removeBook(book.id);
},
```

Now, when the user clicks the delete button, the `onDeleteBook` method is fired, which, in turn, fires the `removeBook` method. This method sends the DELETE request to the back-end. When the response comes back, the alert message is displayed and `getBooks` is ran.

## Conclusion

This post covered the basics of setting up a CRUD app with Vue and Flask.

Check your understanding by reviewing the objectives from the beginning of this post and going through each of the challenges.

You can find the source code in the flask-vue-crud repo. Thanks for reading