Exercises for Web API (POST)

In this exercise, you'll continue working with the application that displays topics and messages in those topics. The only major change to the application is it's refactored to use Vuex to manage state.

You'll use the application as an administrator where you can create, read, update, and delete both topics and messages.

Before you begin: Initialize the project

After opening the project folder in Visual Studio Code, open the **View** menu and click **Terminal**. Alternatively, you may press `Ctrl+`` on Windows and macOS. Next, run the command `npm install` to install any dependencies before working on the project.

To run the project, use `npm run serve`.

To test the project and verify completion, you can do either of the following:

- Type `npm run test:e2e` in Terminal to run the tests using the Cypress user interface. This displays the Cypress UI, which may provide you some extra help, like screenshots, when you're debugging a failed test. This method takes longer to run, however.
- Type `npm run test:e2e-headless` in Terminal to run the tests in "headless" mode. This mode doesn't show the Cypress UI—it just runs the tests and displays results in the console. Tests run significantly faster this way, but you don't get the additional support that the UI provides.

Part One: CRUD Topics

Create all of the service methods for topics in `src/services/TopicService.js`.

Step One: Create a new Topic

Add a new method to the service object that accepts a topic as an argument, performs a `POST` request to the URL `/topics`, and returns a `Promise`. Use Postman to perform a `POST` request to `/topics` and make sure the service endpoint works before moving on.

Next, open `src/components/CreateTopic.vue`. You'll see that the `saveTopic()` method is empty. You'll need to call the method you created in `TopicService`.

When the service returns a promise, check the status code to make sure it created the new topic (201), and then use the router to forward the user to the route named `Home`.

Step Two: Update a Topic

Add a new method to the service object that accepts a topic id and topic as arguments, performs a `PUT` request to the URL `/topics/:id`, and returns a `Promise`. Use Postman to perform a `PUT` request to `/topics/:id` and make sure the service endpoint works before moving on.

Open `src/components/UpdateTopic.vue`. You'll see that the `updateTopic()` is missing a call to the service. You'll need to call the method you just created in `TopicService`.

When the service returns a promise, check the status code to make sure everything was successful (200) and then use the router to forward the user to the route named `Home`.

Step Three: Delete a Topic

Add a new method to the service object that accepts a topic id, performs a `DELETE` request to the URL `/topics/:id`, and returns a `Promise`. Use Postman to perform a `DELETE` request to `/topics/:id` and make sure the service endpoint works before moving on.

Open `src/components/TopicList.vue`. You'll see that the `deleteTopic(id)` is empty. You'll need to call the method you created in `TopicService`.

When the service returns a promise, check the status code to make sure everything was successful (200), and call `this.getTopics()` to refresh the list of topics.

After this part is complete, all tests under `Part One: CRUD Topics` pass.

> Note: if your tests fail and you receive the error message "Too many elements found", refer to the section "*A note on the test runner*", near the end of this document.

Part Two: CRUD Messages

Create all of the service methods for messages in `src/services/MessageService.js`.

Step One: Create a new Message

Add a new method to the service object that accepts a message as an argument, performs a `POST` request to the URL `/messages`, and returns a `Promise`. Use Postman to perform a `POST` request to `/messages`, and make sure the service endpoint works before moving on.

Open `src/components/CreateMessage.vue`. You'll see that the `saveMessage()` method is empty. You'll need to call the method you created in `MessageService`.

When the service returns a promise, check the status code to make sure it created the new message (201), and then use the router to forward the user to the route named `Messages`, with parameter `id` set to `message.topicId`.

Step Two: Update a Message

Add a new method to the service object that accepts a message id and message as arguments, performs a `PUT` request to the URL `/messages/:id`, and returns a `Promise`. Use Postman to perform a `PUT` request to `/messages/:id`, and make sure the service endpoint works before moving on.

Open `src/components/UpdateMessage.vue`. You'll see that the `updateMessage()` is missing a call to the service. You'll need to call the method you created in `MessageService`.

When the service returns a promise, check the status code to make sure everything was successful (200), and then use the router to forward the user to the route named `Messages`, with parameter `id` set to `message.topicId`.

Step Three: Delete a Message

Add a new method to the service object that accepts a message id, performs a `DELETE` request to the URL `/messages/:id`, and returns a `Promise`. Use Postman to perform a `DELETE` request to `/messages/:id` and make sure the service endpoint is working before moving on.

Open `src/components/TopicDetails.vue`. You'll see that the `deleteMessage(id)` is empty. You need to call the method you created in `MessageService`. When the service returns a promise, check the status code to make sure everything was successful (200), and commit a mutation to the Vuex Store:

```
this.$store.commit("DELETE_MESSAGE", id);
```

After this part is complete, all tests under `Part Two: CRUD Messages` pass.

> Note: if your tests fail and you receive the error message "Too many elements found", refer to the following section, *"A note on the test runner"*.

A note on the test runner

The exercises and tests use an NPM package called json-server as the API server. At the start of each test run, the test script restores the "database" of json-server to a known starting state. Exactly when this database restore happens depend on how you run the tests:

- If you run in "headless" mode, every time you execute the `npm run test:e2e-headless` command, it restores the database. So if you have a test failure, and then you correct your code and re-run the tests, everything works well. The script restores the test data, then runs the tests.
- However, if you run the Cypress UI using the `npm run test:e2e` command, you have to keep in mind that the script restores the initial data only when you run that command. Inside the Cypress UI, you can click a test spec to re-run tests, but that doesn't restore the database.

If your tests fail and you receive the error message "Too many elements found", you may need to restore the test database. You can either:

- Close the Cypress UI and cancel it in Terminal using `Ctrl+C`. Then re-run `npm run test:e2e`. This restores the database and re-launches the UI.
- Alternatively, you can leave the Cypress UI running, open *another* Terminal window, and execute the command: `npm run restore-db`. This script restores the database, and the running json-server resets.