How to test Express.js REST API with Cypress

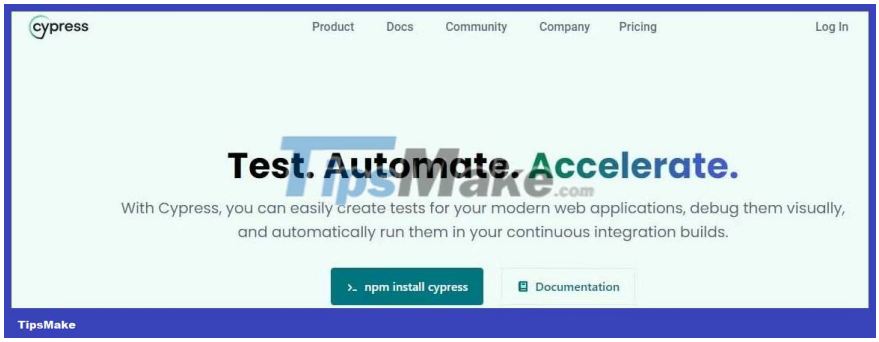
Cypress is great for front-end testing, but it can also effectively test APIs. Cypress allows you to write comprehensive tests that cover the entire work cycle of your web application.

Cypress is a popular testing framework specifically for JavaScript applications. Although it is primarily designed to test UI elements and interactions with interface elements in the browser, it still tests APIs well. You can use this framework to test RESTful AI via HTTP queries and validate responses.

Cypress allows you to write comprehensive tests that cover the entire work cycle of your web application.

Instructions for testing API with Cypress

Cypress helps you verify that the API is working as expected. This process typically includes testing API endpoints, input data, and HTTP responses. You can verify integration with any external services, and confirm debugging mechanisms work correctly.



Test APIs to ensure they are efficient, reliable, and meet the needs of the applications that depend on them. It helps identify and correct errors early, preventing problems from occurring during production.

Cypress is a great UI testing tool, used by several popular JavaScript frameworks. The ability to create and test HTTP queries makes it equally effective in testing APIs.

```
npm install cypress --save-dev
```

Cypress performs the above task by using Node.js as the engine for creating HTTP queries and processing responses.

Create the RESTT Express.js API

To get started, create an Express web server, and install this package in the project:

```
npm install cors
```

Next, add the Cypress package to the project:

```
npm install cypress --save-dev
```

Finally, update the package.json file to include this test script:

```
"test": "npx cypress open"
```

Identify API controls

In real cases, you will make API calls to read and write data from a database or external API. However, in this example, you will simulate and test API calls by adding and fetching user data from an array.

In the root of the project directory, create the file **controllers/userControllers.js** , and add the following code:

First define a registerUser controller function that will manage the user's registration path. It will retrieve user data from the query body, create a new user object and add it to the **users** array . If this process is successful, it will respond with status code 201 and notify that it has registered this user.

```
const users = []; exports.registerUser = async (req, res) => { const { username,
```

Add a second function - **getUsers** - to retrieve user data from the array, and return it as a JSON response.

```
exports.getUsers = async (req, res) => { try { res.json(users); } catch (error)
```

Finally, you can also simulate login attempts. In the same file, add this code to check if the provided username and password match any user data in the users **array** :

```
exports.loginUser = async (req, res) => { const { username, password } = req.body
```

Define API roadmap

To define routes for your Express REST API, create a **routes/userRoutes.js** file in the root directory, and add this code to it:

```
const express = require('express'); const router = express.Router(); const userC
```

Update the Server.js file

Update the server.js file to configure the API as follows:

```
const express = require('express'); const cors = require('cors'); const app = exp
```

Set up the test environment

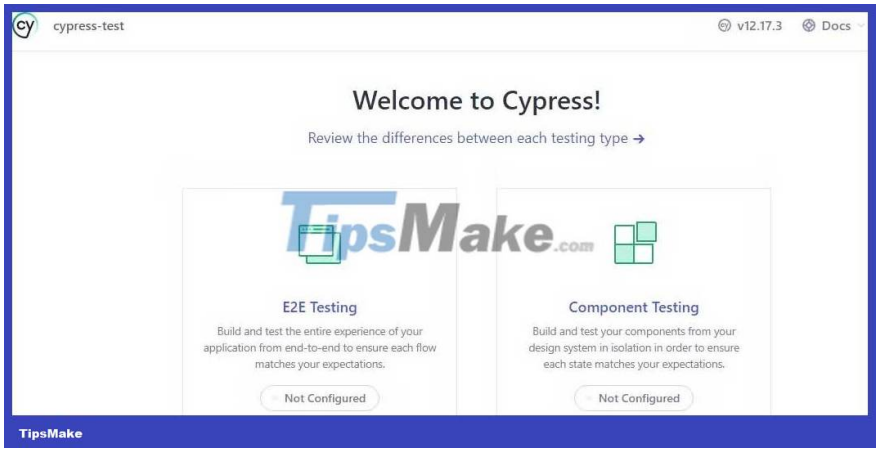
With the testing API in place, you're ready to set up your testing environment. Start the programming server with this terminal command:

```
node server.js
```

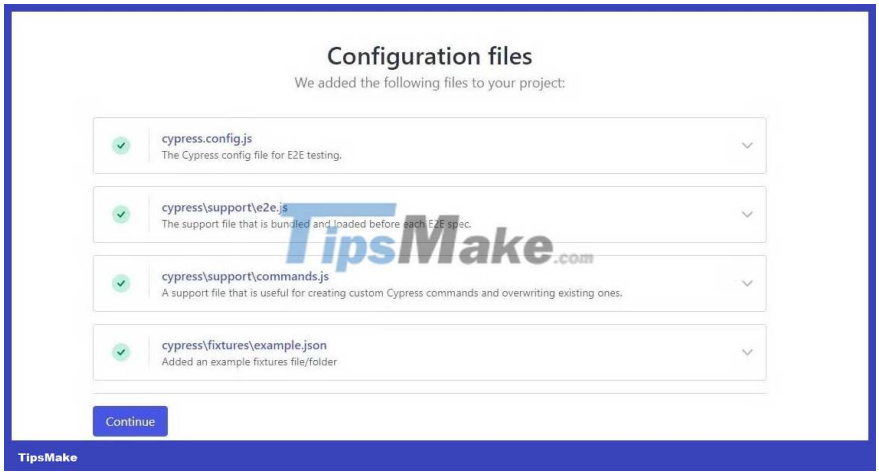
Next, run the test script command in a separate terminal:

```
npm run test
```

This command will open the Cypress desktop client, providing a testing environment. Once it opens, click the **E2E Testing** button . End-to-end testing processes ensure that you test the entire Express API, meaning Cypress will have access to the web server, routes, and related controller functions.



Next, click **Continue** to add the Cypress configuration file.



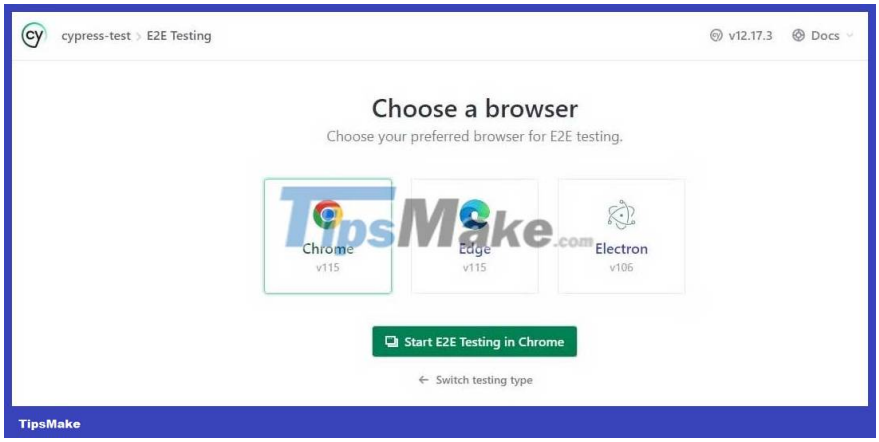
Once the setup is complete, you will see a new Cypress folder in the project. Cypress will also add a **cypress.config.js** file that contains configuration settings for your testing.

Go ahead and update the file to include the server base URL as follows:

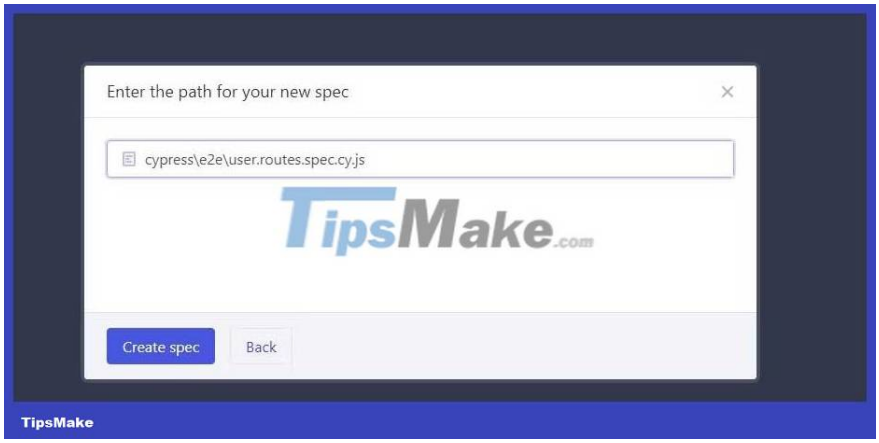
```
const { defineConfig } = require("cypress"); module.exports = defineConfig({ chro
```

Write test cases

Now you're ready to write some test cases. First, select the browser that Cypress will open to run the tests from the options available on the Cypress client.



Next, click the **Create new spec** button to create the test file, and provide a name. Then click **Create spec** .



Now open the file **cypress/fixtures/example.json** , and update its content with the user authentication information below. Fixture is a file that contains static test data that you can use in test cases.

```
{ "username": "testuser", "password": "password123" }
```

Cypress provides the **cy.request** method to make an HTTP request to the web server. You can use it to test different types of HTTP endpoints, which manage individual operations, including GET, POST, PUT, and DELETE.

To test the 3 API roadmap you defined from the beginning, start by describing a test case for the registration endpoint. This test case will verify that the endpoint is working correctly by successfully registering new users and confirming correct information.

Open the file **cypress/e2e/user.routes.spec.cy.js** and update its content with the following code:

```
describe('User Routes', () => { it('registers a new user', () => { cy.fixture('ex
```

In this test, Cypress will load the test data in the fixture file, and create POST queries to the specified endpoint with the data in the query body. If all assertions pass, the test case is successful. Otherwise, it will fail.

It's worth noting here that the syntax for Cypress testing is almost identical to the syntax used in the Mocha tests that Cypress has adopted.

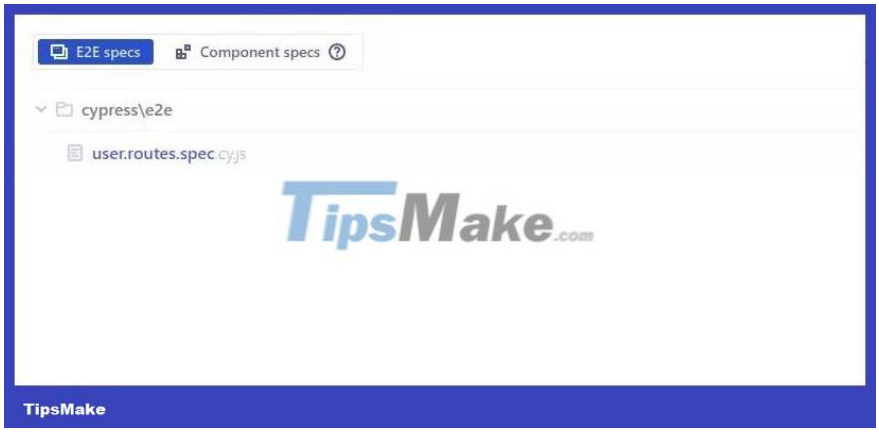
Now describe the test for the **users** route . This test will verify that this response contains user data when making queries to this endpoint. To achieve that, add the following code inside the test **describe** block .

```
 it('gets users data and the username matches test data', () => { cy.fixture('exa
```

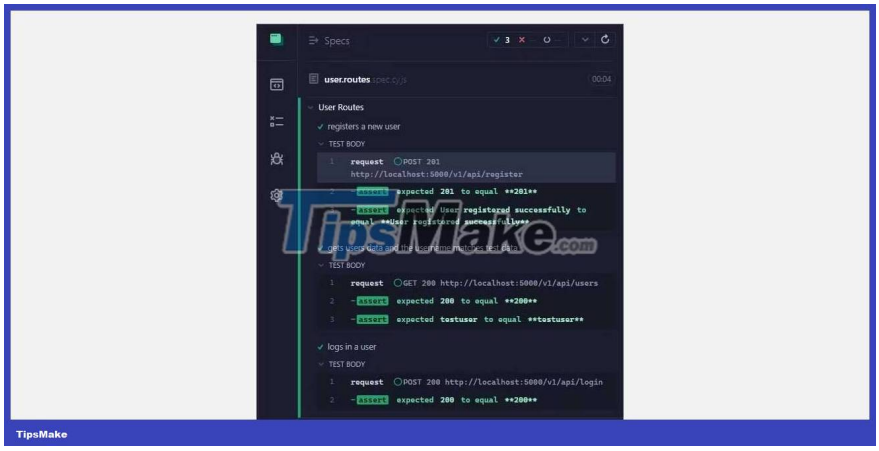
Finally, include a test case that will check the login endpoint and insert a response status of 200, indicating successful login.

```
 it('logs in a user', () => { cy.fixture('example').then((loginData) => { cy.requ
```

To run this test, return to your Cypress-managed browser instance and select the specific test file you want to run.



The Cypress test runner will run the tests and record the results, showing the pass or fail status of each test case.



The example above illustrates how you can test different routes and their corresponding controller functions to ensure their functionality and behavior work as expected.

Overall, Cypress is a great tool for testing web applications, including both front-end and back-end. Try it and feel the great benefits it brings!

You finished reading the article "**How to test Express.js REST API with Cypress**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.
