Recipe API Lab: Exploring APIs with Swagger

Goal: Use a working Recipe API (already running on your computer) to explore how APIs behave. You will practice sending requests, looking at responses, and noticing how the URL, HTTP verb, and JSON all work together. You will not write any code in this lab; everything is done through Swagger UI.

By the end of this lab, you should feel more comfortable with the basic API "verbs" (GET, POST, PUT, DELETE) and with reading and sending JSON.

1. Setup: Open the Recipe API in Swagger

1. Make sure your Recipe API project is running (your instructor will show you how).

2. Open your browser and go to the Swagger UI URL given by your instructor (for example, something like http://localhost:8080/swagger-ui).

3. Find the section that lists the endpoints for recipes (look for something like "Recipe" or "/recipes").

In Swagger, each endpoint usually shows:

- • The HTTP verb (GET, POST, PUT, DELETE, etc.)
- • The URL path (for example: /recipes or /recipes/{id})
- • Any parameters the endpoint needs (for example: an id, or extra options)
- • The request body (for example: JSON you send to the server)
- • The possible responses (status codes and response body)

2. Predict What Each Endpoint Does

Before you try anything, look at the list of Recipe endpoints in Swagger and make a prediction about what each one does. You do not have to be perfect. This is just to get you thinking.

HTTP Verb + Path (from Swagger)	What I think this does (before trying)	Notes after trying it

Fill in at least a few of the rows above using the endpoints you see in Swagger. Later, you can come back and write notes about what they really did.

3. Creating New Recipes (Sending Data to the Server)

In this section you will send data to the server to create new recipes. You will use a POST endpoint in Swagger.

1. In Swagger, find the endpoint that creates a new recipe (look for a POST endpoint, for example POST /recipes).

2. Click on that endpoint to expand it. Then click the "Try it out" button.

3. You should see a box where you can type or paste JSON. Paste in a new recipe, for example:

```
{
 "name": "Peanut Butter Sandwich",
 "ingredients": "2 slices of bread, peanut butter, jelly",
 "instructions": "Spread peanut butter and jelly on bread. Put slices together."
}
```

4. Click the "Execute" button to send the request to the server.

5. Look at the response section that appears below. Answer these in your own notes:

- • What HTTP status code do you see?
- • Do you see JSON coming back from the server?
- • Does the JSON include an id that you did not send?

6. Repeat steps 3–5 to create at least two more different recipes. Try using different names and ingredients.

4. Reading Recipes Back (Getting Data from the Server)

Now you will see how to ask the server for data that is already stored. You will use GET endpoints in Swagger.

A. Get all recipes

1. Find the GET endpoint that returns a list of recipes (for example, GET /recipes).

2. Expand it, click "Try it out," then click "Execute."

3. Look at the JSON that comes back. In your notes, think about:

- • Does it come back as a single object or as a list/array?
- • How many recipes do you see?
- • Do you see the ones you created earlier?

B. Get a single recipe by id

Some endpoints have a piece of the URL that changes, such as /recipes/{id}. In Swagger, this usually appears as a box where you can type a value (for example, an id number). This is sometimes called a “path parameter” or “URL parameter.”

1. Find the GET endpoint that looks like it gets a single recipe (for example, GET /recipes/{id}).

2. Expand it and click “Try it out.” You should see a place to enter an id.

3. First, use GET /recipes (from part A) to find an id that actually exists.

4. Put that id into the box in GET /recipes/{id}, then click “Execute.”

5. Look at the response. In your notes, consider:

- • What URL did Swagger show for this request (look for the full URL under “Request URL”)?
- • Does the JSON now show just one recipe?

C. Try an id that does not exist

1. In the same GET /recipes/{id} endpoint, try putting in an id that you know does not exist (for example, a very large number).

2. Click “Execute” again.

3. Look at the response. In your notes, think about:

- • Did the status code change?
- • What is the server telling you?

5. Exploring URL Parts and Parameters

In Swagger, some endpoints may have extra options or boxes you can fill in, such as id, page, size, or others. These are called parameters. There are two common kinds to look for in this lab:

- • Parameters that become part of the path, like /recipes/3
- • Parameters that appear after a question mark in the URL, like /recipes?page=0&size=5

Do this exploration:

1. Look at any endpoint in Swagger that has a box labeled with something like id.

- • When you change the number and click “Execute,” how does the Request URL change?

2. If you see any parameters that show up as “query” in Swagger (for example, page, size, or search):

- • Try changing those values and see how the results change.
- • Notice how the Request URL changes when you change these options.

3. In your own words, write down how a parameter that is part of the path (like /recipes/3) feels different from an option that shows up after a question mark (like ?page=0&size=5).

6. What Happens When You Send Bad Data?

The Recipe API has some rules about what data is allowed. For example, the name, ingredients, and instructions are required fields. In this section, you will experiment with sending “bad” data and observe how the server responds.

1. Go back to the POST endpoint for creating a recipe. Click “Try it out.”

2. This time, try sending JSON with missing or empty fields, for example with an empty name or missing ingredients.

3. Click “Execute.” In your notes, observe:

- • What HTTP status code do you see now?
- • Does the server send back any error messages in the JSON?
- • How do these messages relate to the rules your instructor showed you for the Recipe fields?

7. Updating an Existing Recipe

In this section you will change an existing recipe using a PUT endpoint. Updating usually means: “keep the same thing, but with new data.”

1. Use GET /recipes to pick one recipe to update. Write down its id and current name in your notes.

2. Find the endpoint that updates a recipe (for example, PUT /recipes/{id}).

3. Click “Try it out,” enter the id of the recipe you chose, and look at the example JSON body.

4. Change the JSON body so that at least one field is different (for example, change the name or instructions).

5. Click “Execute.”

6. Use GET /recipes/{id} to look up the same recipe again. In your notes, observe:

- • Did the data actually change?
- • How does this feel different from creating a brand new recipe with POST?

8. Deleting a Recipe

Now you will remove a recipe from the server using a DELETE endpoint. Deleting means the data is gone from the list the API returns.

1. Use GET /recipes to find one recipe you are willing to delete. Note its id in your notes.

2. Find the DELETE endpoint (for example, DELETE /recipes/{id}).

3. Click "Try it out," enter the id of the recipe you chose, and click "Execute."

4. Use GET /recipes/{id} for the same id you just deleted.

5. Use GET /recipes again to see the full list. In your notes, think about:

- • What happened to the recipe you deleted?
- • What status code do you get when you try to get it by id now?

9. Reflection (Write in Your Own Words)

Answer these questions in your own words. You do not need to use perfect technical vocabulary yet; the goal is to see what you understood from the lab.

1. 1. For each of these, write which verb and which kind of endpoint you used in this lab:

- • Create a new recipe
- • Read one recipe
- • Read many recipes
- • Update a recipe
- • Delete a recipe

2. 2. In your own words, describe how the URL changed when you:

- • Asked for all recipes
- • Asked for a single recipe by id

3. 3. Write a short explanation of what you think a "parameter" is in the context of an API.
4. 4. Imagine a mobile app that lets users browse recipes. Write a few sentences explaining how that app could use the same API you used today.