

# HTTP GET Labs

A series of labs working with HTTP's GET method

## Part 1: Building a controller

Making a basic URL available. Also setting up a class that we will modify over the next several labs.

Create a Java class in the `main` folder, under `microservices.labs` called `MemberController`. Remember to annotate it with the correct annotation `RestController`.

In `MemberController`, create a function, using the right annotations, that:

- Answers on the URL `/members/status`
- Returns a `String`
- The message can be simple "All members available" or something like that.

Start up your server, either from the command line or via Spring Tool Suite. Navigate to `http://localhost:8080/members/status` to see if the mapping worked.

## Part 2: Testing a controller

Automatically testing the URL we set up in the last lab.

Create a Java class in the `test` folder, under `microservices.labs` called `MemberControllerTest`. Add code for the following:

- The class should use `@SpringBootTest`
- Set up a random port for the server to run on
- Include a `TestRestTemplate`
- Write a test to check that `MemberController` returns the expected default message.

Run your test to see if it works!

## Part 3: A real object!

Let's integrate a real object into our microservice.

Create a class, `Member`. It should have the following properties:

- `id`: `long`
- `firstName`: `String`
- `lastName`: `String`
- `policyHolder`: `boolean`

Methods:

- Constructor signature: `Member(int, String, String, boolean)`

- There should be public getters for each of the properties
- You can generate `equals` and `hashCode` methods

In `MemberController`, add a mapping for `/members`. It should create a `Member` and return it as JSON. The `Member` values can be hard-coded at the moment.

Start your microservice and navigate to `http://localhost:8080/members` to see if it works!

## Part 4: Testing a real object

Wherein we add testing for the object mapping we just set up.

Back in `MemberControllerTest`, add a test which requests `/members` and checks to see if the appropriate object is returned.

Run your test(s) and see if they pass.

## Part 5: Working with lists

We are on our way to implementing typical microservice endpoints! In this lab, we will implement retrieving a list of values

### In `MemberController`

- Create an initializer function that defines a list or array of `Members`
- It's fine to hard-code them for now
- Add a mapping for `/members` which returns the list/array of `Members`
- Check the URL in a browser or Postman to see if it works

### In `MemberControllerTest`

- Create a test to validate that the `/members` URL returns an *array* of the appropriate size
- Create a second test to validate that the `/members` URL returns a `List<Member>` of the appropriate size
- Devise a test that will check that the returned value contains expected results
- How would you test to ensure that there were a certain number of `Members` whose `policyHolder` property was true?

## Part 6: Working with Query Parameters

Adding search capabilities to our microservice

### In `MemberController`

First, we will concentrate on searching on the `lastName` field:

- For the mapping `/members`, update the handling method to take a single query parameter `lastName`
  - The parameter should be optional
- Using the value in `lastName`, return a list/array of only the matching Members
- Think about what happens if there are no matching Members

#### **In MemberControllerTest**

- Create a test to validate the search on `lastName` returns the appropriate values

#### **Back in MemberController**

- How can we grab **all** the query parameters passed to `/members`?
- What would we do to update the code that returns the results?

#### **Back in MemberControllerTest**

- The previous test should still work!
- Add a test for searching on first name, standalone
- Add a test for searching on two or more parameters (first name & `lastName` or last name & policy holder, as examples)

## **Part 7: Working with URL parameters**

We can get a list of Members, what about fetching an individual member?

#### **In MemberController**

- Add a mapping so that `/members/5` will return the member with the id 5
- What happens when the mapping doesn't have a corresponding Member?
  - We'll deal with this in the next exercise
- Check the URL in a browser or Postman to see if it works

#### **In MemberControllerTest**

- Create a test to validate that `/members/5` returns an expected Member.
- Create another test to validate what happens when requesting an invalid member number
  - We will update this to be better in the next exercise

## **Part 8: Exception handling**