Server-Side APIs: Part 1 Exercise (Java)

Previously, you created a command-line application that made requests to an API server. In this exercise, you'll re-create the API server in Java to handle `GET` and `POST` requests.

Step One: Import project into IntelliJ and explore starting code

Import the server-side APIs part 1 exercise into IntelliJ. After you've imported the project, review the starting code.

Models

In the `model` package, there's an `Auction.java` model that has the same properties as the `Auction` class you've been working with.

DAO

In the `dao` package, there's a `MemoryAuctionDao.java` class that provides data access code. To reduce complexity, a static `List` substitutes for a DBMS. Also, there are some methods in there that you'll call from the controller.

Controllers

In the `controller` package, there's an `AuctionController.java` class that you'll work in today.

You'll create the methods for the API. The controller class already contains the necessary `@RestController` and `@RequestMapping` annotations. The value for `@RequestMapping` is `/auctions`, which is the base path for all the request mappings defined in this controller.

Tests

In `src/test/java/com/techelevator/auctions/controller`, you'll find the test class `AuctionControllerTest`, which has all of the tests for the methods you'll write today. After you complete each step, more tests pass.

If you also want to run the server and test with Postman or the browser, feel free to do so. However, your goal is to make sure all the unit tests pass.

Step Two: Implement the `list()` method

This method's purpose is to return a list of all auctions.

In `AuctionController.java`, create a method named `list()` that returns a `List<Auction>`. Then add the `@RequestMapping` annotation to have this method respond to `GET` requests for `/auctions`.

Look in `MemoryAuctionDao.java` for a method that returns all auctions.

If completed properly, the `listShouldReturnStatusOK()` and `listShouldReturnCorrectCount` tests pass.

Step Three: Implement the `get()` action

This method's purpose is to return a specific auction based on the value passed to it.

In `AuctionController.java`, create a method named `get()` that accepts an `int` and returns an `Auction`.

Add the `@RequestMapping` annotation to this method to respond to `GET` requests for `/auctions` with a number following it—for example, `/auctions/7`. You'll need to pass a value to the path to tell it to accept a dynamic parameter.

Look in `MemoryAuctionDao.java` for a method that returns a specific auction based upon an `int` that's passed to the access method.

If completed properly, the `getShouldReturnSingleAuction` and `getInvalidIdShouldReturnNothing` tests pass.

Step Four: Implement the `create()` action

This method's purpose is to add the auction that's passed to it.

In `AuctionController.java`, create a method named `create()` that accepts an `Auction` and returns an `Auction`.

Add the `@RequestMapping` annotation to have this method respond to `POST` requests for `/auctions`.

Look in `MemoryAuctionDao.java` for a method that creates an auction. The controller method passes the newly created auction back to the client.

If completed properly, the `createShouldAddNewAuction` and `createShouldThrowExceptionWhenRequestBodyDoesntExist` tests pass.

Step Five: Add searching by title

The purpose of this step is to enable searching by title. You'll modify the `list()` method to accept an optional query string parameter that returns all auctions with the search term in the title. The URL to use this would look like `/auctions?title_like=` with a search term following it.

In `AuctionController.java`, return to the `list()` action method. Add a `String` request parameter with the name `title_like`. You'll need to make this parameter optional, which means you set a default value for it in the parameter declaration. In this case, you want to set the default value to an empty string `""`.

Look in `MemoryAuctionDao.java` for a method that returns auctions that have titles containing a search term. Return that result in the controller method if `title_like` contains a value, otherwise return the full list like before.

If completed properly, the `searchByTitleShouldReturnList` and `searchByTitleExpectNone` tests pass.

Step Six: Add searching by price

The purpose of this step is to enable searching by price. You'll modify the `list()` method to accept an optional query string parameter that returns all auctions with the current bid less than or equal to the value

passed to it. The URL to use this would look like `/auctions?currentBid_lte=` with a numeric value following it.

In `AuctionController.java`, return to the `list()` action method. Add another optional parameter after `title_like`—this time a `double` with the name `currentBid_lte`. Set the default value to `0`. Using the declaration of `title_like` as a guide, figure out how to declare `currentBid_lte`.

Look in `MemoryAuctionDao.java` for a method that returns auctions based on prices being less than or equal to a certain amount. Return that result in the controller method if `currentBid_lte` is greater than zero.

If completed properly, the `searchByPriceShouldReturnList` and `searchByPriceExpectNone` tests pass.

Step Seven: Search by title and price

You might be thinking, "Wait, what if the client searches with both parameters?" There's another method in `MemoryAuctionDao.java` that returns search results for both parameters. Add a call to this method if a request has both parameters. In the controller method, determine when to call the search methods and when to call the method that returns the full list.

If completed properly, the `searchByTitleAndPriceExpectOne` and `searchByTitleAndPriceExpectNone` tests pass.

If you completed all of the steps correctly, all of your tests pass.