Fidelity INVESTMENTS

DEVELOPING RESTFUL SERVICES
EXERCISE MANUAL

ROITRAINING
MAXIMIZE YOUR TRAINING INVESTMENT

Fidelity LEAP
Technology Immersion Program

This page intentionally left blank.

Table of Contents

Chapter 1: Building RESTful Services

Exercise 1.1: Exploring the Time Service

In this exercise, you will explore an existing RESTful service. You will use a browser to send a request to the service and view the response. You will also use the Insomnia application to communicate with the service and view the response.

It is strongly recommended that you use the Spring Tool Suite (STS) for the projects in this course. STS is based on Eclipse but contains many additions that are specifically designed to provide support for Spring-based projects.

Your instructor will tell you how to import the exercise projects for this course. When you import all the projects in the course zip file, STS will be busy opening, examining, and validating all of the projects. It is best to let STS finish its work before starting to work on the exercise. If you open a project too quickly, you may be faced with a distressingly large number of squiggly red complaints from STS.

If you get a prompt for Marketplace solution available when opening your project, feel free to choose **Show Solutions** and **Install** the top solution.

Time: 30 minutes
Format: Programming exercise

1. Open the `TimeService` project.

2. Examine the source file for the REST controller in `com.fidelity.restcontroller`.
   a. Note the URL associated with each web method.

3. Launch the `TimeZoneServiceApplication` in the `com.fidelity` package by running it as a Spring Boot app (if you're using STS) or as a Java application (if you're using Eclipse).
   a. In the console output, note the port that Tomcat is listening on.

4. Use the Insomnia application to communicate with the TimeService (if you are asked to update to the current version of Insomnia, feel free to do so).
   a. Create a new request with Ctrl-N
   b. Send a `GET` request to the time service URL for the current time.
   c. View the HTTP response status code and JSON response body.

5. Send GET requests for the current time in several different time zones.
   a. Enter the URL for the web method for the current time zone information.
   b. View the response body.

   Hint:
   - http://localhost:8080/time?timezone=America/New_York
   - http://localhost:8080/time?timezone=America/Chicago
   - http://localhost:8080/time?timezone=IST

   Notes:
   If there is not a "/" in the time zone name, you can send a request without a query parameter:
   - OK: http://localhost:8080/time/IST
   - Not OK: http://localhost:8080/time/America/New_York

   The list of legal Java time zone names is at https://docs.oracle.com/middleware/12211/wcs/tag-ref/MISC/TimeZones.html.

6. Try sending a `GET` request for the time for an invalid time zone ID.
   a. How does the web service respond?

Exercise 1.2: Creating a RESTful API

In this exercise, you will create a RESTful API and test it with Insomnia.

Time: 30 minutes
Format: Programming exercise

1. Open the `LibraryService` project.

2. Create a RESTful service by completing the implementation of the two REST controller methods in `LibraryController.java`.
   a. The `LibraryController` should call on the `MockLibraryDao` for the book data.
   b. The data should be returned in either XML or JSON format.

3. Run the `LibraryServiceApplication` as a Spring Boot application (if you're using STS) or as a Java application (if you're using Eclipse).

4. Test the `LibraryService` using Insomnia.
   a. Verify that the service returns either XML or JSON depending on the value of the `Accept` request header.

Bonus Exercise (to be attempted if time permits)

5. Add a method to the `LibaryController` that inserts a book into the library.
   a. Modify `LibraryDao` and `MockLibraryDao` as required to support the new method.

Exercise 1.3: Which Status Code to Use?

In this exercise, you will research which HTTP status code should be returned in various situations.

Time: 20 minutes
Format: Research and class discussion exercise

1. Visit the URL in the Course Notes or do a web search to decide what HTTP status code to return when problems occur. (https://www.codetinkerer.com/2015/12/04/choosing-an-http-status-code.html)

2. When would you return a 500-level status code?

3. When would you return a 400-level status code?

4. What status codes can you return that do NOT indicate an error (besides the boring 200)?

5. What is the difference between a 204 and a 404 response?

6. Be ready to discuss with the class.

Exercise 1.4: Returning a Status Code

In this exercise, you will modify your RESTful API to return proper HTTP status codes.

Time: 30 minutes
Format: Programming exercise

1. Continue working with the `LibraryService` project that you worked with in a previous exercise.

2. Modify the method that gets all books to return an HTTP status code of `NO_CONTENT` if there are no Books in the database.
   a. Use Insomnia to verify this new functionality.

3. Modify the method that queries a book by ID to return an HTTP status code of `NO_CONTENT` if that ID is not in the database. Test the method with Insomnia.

4. Modify the controller methods to return an HTTP error status code if a DAO method throws an exception.
   a. *Hint:* set an error status code by throwing an instance of a subclass of Spring's `ResponseStatusException`, as described earlier in the Chapter 1 course notes.
   b. What is the appropriate error code to return in this situation?
   c. Test this new functionality with Insomnia.

Bonus Exercise (to be attempted if time permits)

5. Revise the API to ensure that if a request to look up a book has an ID shorter than 10 characters, the response has an error status code.
   a. Add the validation code to the DAO method, not the controller method. The DAO method should throw an `IllegalArgumentException` if the input ID is not valid.
   b. In the controller method, catch the `IllegalArgumentException` and set the HTTP response status code.
   c. What is an appropriate HTTP response code for a request that has invalid data?

6. If you haven't completed the Bonus Exercise section of Exercise 1.2, do that now. Then modify the method that adds a book, so it returns an error status if the input book is not fully populated.
   a. As before, add the validation code to the DAO, not the controller.

This page intentionally left blank.

Chapter 2: Designing RESTful Services

Exercise 2.1: Research Spring Boot at Fidelity

The use of Spring Boot at Fidelity was discussed in class. In this exercise, you will learn more about developing software with Spring Boot by reading the Spring Boot Reference Application section of the Confluence documentation.

Time: 30 minutes
Format: Research and report done in pairs

1. In your browser, visit the following site:
   a. https://teams.microsoft.com/l/team/19%3a776314441292463fbeb63f33c2cdc006%40thread.skype/conversations?groupId=8437d6a7-20ce-4853-8141-1e4a2ae913d6&tenantId=7521acbc-a68c-41e5-a975-1cf83066dd19
   b. What does the Cloud Café offer?
   c. Where is it located?

2. In your browser, visit the Confluence documentation at the following link: https://confluence.fmr.com/display/EAP/Springboot+Reference+Application

3. With your partner, choose an entry in the "Dependencies to Avoid Security Vulnerabilities" section that lists a CVE entry in the last column.

4. Research that CVE issue.

5. Prepare to discuss with the class:
   a. Which Artifact ID did you choose?
   b. What CVE issue is related to that artifact?
   c. What is the issue that is documented in that CVE?

Exercise 2.2: Designing a RESTful Service API

In this exercise, you will design a RESTful web service API for a service that will manage the items stored in a Warehouse. At present, there are only Widgets and Gadgets stored in the Warehouse.

The service should provide an API to manage Widgets. In particular, the service should support querying, adding, modifying, and removing Widgets in the Warehouse.

Time: 40 minutes
Format: Programming exercise

1. Open the `WarehouseService` project.

2. Design the API for the RESTful service that will manage Widgets.

3. Be sure to return proper HTTP status codes.

4. Work using TDD. In the next chapter, you'll learn how to write automated test cases for RESTful APIs. For now, you can use TDD for manual testing. For example, to implement the API method that gets all widgets:
   a. Use Insomnia to send a GET request to the URL you have chosen.
   b. Verify that the request fails, e.g., the response status is 404 (Red).
   c. Open `WarehouseController.java` and complete the steps in the TODO comments.
   d. Implement a method in the controller class that returns the expected JSON and HTTP status code.
   e. Use Insomnia to verify the service method works as expected (Green).
   f. Now look for opportunities to Refactor your code to improve the design: eliminate duplicate code by extracting private helper methods; replace literal numbers and strings with constants; etc.
   g. Once you get the first method working, use the same TDD strategy to implement another method: send a new HTTP request and confirm that it fails; implement a new method; verify the new request now succeeds; refactor the code to improve the structure and design.

Bonus Exercise (to be attempted if time permits)

5. Extend the service to also manage Gadgets.
    a. The service should support querying, adding, modifying, and removing Gadget information.

6. Extend the service to support a query for all Products, including both Gadgets and Widgets.
    a. Work using TDD: start with a failing Insomnia test (Red).
    b. Make the fewest modifications possible to get the new controller method working (Green).
    c. After verifying the new method works, refactor your code to remove duplication and improve the structure (Refactor).

Optional Exercise 2.3: Deploying a Spring Boot RESTful Service into Docker

In this exercise, you will deploy a Spring Boot-based RESTful service into a Docker container.

Time: 20 minutes
Format: Programming exercise

1. In Eclipse, open the `DockerWarehouseService` project.

2. Examine the `Dockerfile`.
   a. This file controls how the Docker image will be created.

3. Build a Docker image for this project by doing the following steps:
   a. Right-click the project (in the Project Explorer window) and select **Run As | Maven build ...**
   b. In the **Goals** text box, type: `clean package`
   c. Then click **Run**.

4. Verify that the Spring Boot uber jar file is in the `target` folder.
   a. Right-click the **`target` folder | Refresh**.
   b. Note the name of the uber jar file: `DockerWarehouseService-1.0.0.jar`

5. Open a Command window as an Administrator.
   a. Double-click the desktop icon **CMD Prompt (as Admin)**.

6. In the command window, enter `D:` to switch to that drive.
   a. Then change to the directory for the `DockerWarehouseService` project.
   b. Don't remember where it is?
      i. In STS, right-click the project and choose **Show In | System Explorer**.

7. Build the Docker image by running the following command (don't forget the period at the end of the command):

```
docker build --tag warehouse/dockerwarehouse .
```

8. It make take several minutes to download the JRE image. If the download appears to hang, press ENTER in the command window a few times.

9. Run the Docker image by running the following command:

```
docker run -p 7777:8080 warehouse/dockerwarehouse
```

10. The application should now be available on `http://localhost:7777`.
    a. Verify this by using Insomnia to send a GET request to `http://localhost:7777/warehouse/widgets`.

11. Open another command prompt as administrator. Run a second Docker image by executing the following command:

```
docker run -p 8888:8080 warehouse/dockerwarehouse
```

12. You now have two Docker containers running your application.
    a. Verify this by using Insomnia to send a GET request to `http://localhost:8888/warehouse/widgets`.

13. Open a third command window as Administrator.

14. Run the following command in the new command window to view the list of Docker containers:

```
docker ps
```

15. To stop a Docker container, run the following command with the container ID or unique name displayed by the previous command:

```
docker stop container-id
```

Exercise 2.4: Research the Twelve-Factor App

Several of the features described in the Twelve-Factor App were discussed in class. In this exercise, you will explore some of the other features listed in the Twelve-Factor App.

Time: 30 minutes
Format: Research and report done in pairs

1. With your partner, decide what feature in the Twelve-Factor App you want to learn more about.
   a. Choose a feature that was not covered in class.

2. Research the feature.
   Note: https://12factor.net.

3. Prepare to discuss with the class
   a. What feature did you choose?
   b. Why did you choose that feature?
   c. Describe what the feature is.
   d. Why is important for software development?
   e. How does this relate to software development at Fidelity?

Exercise 2.5: Debugging a RESTful Service

In this exercise, you will debug a RESTful service and whip it into working order.

It is better to use the Spring Tool Suite (STS) to debug a RESTful web service. While it is possible to do this with Eclipse, it is much more work to establish a connection to the remote debugger.

Time: 30 minutes
Format: Programming exercise

1. Open the `BuggyService` project.

Use STS to debug the BuggyService

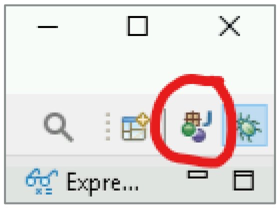
2. Set a breakpoint in the web service method.

3. Right-click the **BuggyService** project.
   a. Choose **Debug As | SpringBootApp**.
   b. This will launch the debugging session.

4. In Insomnia, send a GET request to the URL that corresponds to the web service method.
   a. STS will pause execution and highlight the breakpoint.
   b. Be sure to switch to the Debug perspective.
   c. Use the STS debugging environment to find the error(s).
   d. Correct it (them).

5. After you fix the error for the request to `/time/current`, trying sending a GET request to `/time/zone/CST`.
   a. Debug this error also.

6. When you are done debugging, you can switch from the Debug Perspective to the Java Perspective by clicking the "J" icon in the top-right corner of the STS toolbar.



Shutting down the debugging environment

7. Click the red square icon on the task bar in the Console window.
    a. This will shut down the Spring Boot application.

8. Right-click in the Console window and choose **Terminate/Disconnect All**.
    b. This will shut down the remote debugger.

Chapter 3: Testing RESTful Services

Exercise 3.1: Testing Back-End POJOs

In this exercise, you will continue working with the `WarehouseService` project from a previous exercise. You will test the data access object with an integration test. You will test the business service with a unit test where you will mock the DAO that the business service depends on. Once the separate tests for the DAO and business service are passing, you will then create an integration test to verify that the business service and DAO perform correctly together.

Time: 45 minutes
Format: Programming exercise

1. Continue working with the `WarehouseService` project.

2. Open the `WarehouseDaoMyBatisImplIntegrationTest` class.

3. Run the tests in this test class.
   a. You will notice that some of the tests have not been implemented.

4. Complete the tests that are not implemented by following the instructions in the TODO comments in the source code.
   a. And, of course, get a green bar.

5. Open the `WarehouseBusinessServicePojoUnitTest` class.
   a. Run the tests in this class. Again, some of the tests aren't implemented.

6. Complete the tests that are not implemented as instructed by the TODO comments.
   a. And, of course, get a green bar.

Bonus Exercise (to be attempted if time permits)

7. Open the `WarehouseBusinessServiceIntegrationTest` class.
   a. Guess what? None of the tests have been implemented.

8. You know what to do.

9. Now you should have that warm, fuzzy feeling that you get from that green bar.

Exercise 3.2: Testing the Web Layer

In this exercise, you will continue working with the `WarehouseService` project from the previous exercise. You will test the behavior of the RESTful service operating in the HTTP (web) layer.

Time: 30 minutes
Format: Programming exercise

1. Continue working with the `WarehouseService` project that you used in the previous exercise.

2. View the code in the `WarehouseControllerWebLayerTest`.

3. Complete the test methods by following the instructions in the TODO comments.

4. Run this test as a JUnit test.

5. Of course, you get the green bar, right?

Bonus Exercise (to be attempted if time permits)

6. Write web layer tests for the `LibraryController` in the `LibraryService` project from Chapter 1.
   a. If you didn't complete that project, use the `LibraryServiceSolution` project as a starting point.

7. Use TDD with web layer tests to implement additional RESTful operations in the `LibraryController`.
   a. Add a book.
   b. Delete a book.
   c. Update a book.

Exercise 3.3: End-to-End Testing with `TestRestTemplate`

In this exercise, you will continue working with the `WarehouseService` project from the previous exercise. You will use `TestRestTemplate` to further test the web service.

Time: 30 minutes
Format: Programming exercise

1. Continue working with the `WarehouseService` project that you used in the previous exercise.

2. View the code in the `WarehouseControllerE2eTest`.

3. Complete the test methods by following the instructions in the TODO comments.

4. Run this test as a JUnit test.
    a. Verify the tests pass.
    b. Green bars – goes without saying!

Bonus Exercise (to be attempted if time permits)

5. Write E2E tests for the `LibraryService` project from Chapter 1.
    a. If you didn't complete that project, use the `LibraryServiceSolution` project as a starting point.

6. Use TDD with E2E tests to implement additional RESTful operations in the `LibraryService`.
    a. Add a book.
    b. Delete a book.
    c. Update a book.

This page intentionally left blank.

Chapter 4: Securing RESTful Web Services

Exercise 4.1: Using OAuth2 for Authorization

In this exercise, we will study the interactions between a Spring Boot authorization server, a Spring Boot resource server, and an Angular client using the OAuth2 protocol.

Time: 30 minutes

The authorization server is deployed at localhost:8083. It is implemented with Keycloak, an open-source identity and access management framework. Spring Boot provides wrappers that make Keycloak easier to configure and use.

The resource server, deployed at localhost:8081, is our WidgetService with the addition of OAuth2 and CORS.

The client, deployed at localhost:8089, is a simple Angular front-end that communicates with both the authentication server and the resource server.

Summary of the authorization sequence:

1. The Angular client passes the user's credentials to the authorization server.
2. The authorization server verifies the user's credentials and responds with an access token, which is signed with the server's private key.
3. The client adds the access token to a GET request to the resource server.
4. The resource server validates the token using the authorization server's public key.
5. The resource server returns the requested resource to the client.

Exercise Steps

1. Copy `M:\FSE\RESTfulServices\OAuth2Exercise.zip` to `D:\`

2. Extract `OAuth2Exercise.zip` to `D:\`.

3. Open a new STS or Eclipse workspace and import `D:\Oauth2Exercise\oauth-authorization-server` using **File > Import > Maven > Existing Maven Projects**.
   a. The project is big and may take a few minutes to build. While it's building, continue to the next step.

4. In Visual Studio Code, open the folder `Oauth2Exercise\oauth-angular-client`. Build the client:

   ```
   npm install
   ```

5. Switch back to Eclipse and import `Oauth2Exercise\oauth-resource-server`.
   a. This is a stub WarehouseService with OAuth2 and CORS enabled.

6. Start both servers. Be sure to wait until the auth server is running before starting the resource server:
   a. Run as Java application:
      i. `oauth-authorization-server > com.roifmr.auth.AuthorizationServerApp`
      ii. Wait until you see the following output in the console:
      Embedded Keycloak started: http://localhost:8083/auth to use keycloak
   b. Run as Java application:
      i. `oauth-resource-server > com.roifmr.resource.ResourceServerApp`
      ii. Wait until you see the following output in the console:
      Tomcat started on port(s): 8081 (http) with context path '/resource-server'

7. Switch back to Visual Studio Code and start the `oauth-angular-client` application:
   a. `npm start`
   b. The Angular client will later retrieve widget data by sending a GET request to the resource server at `http://localhost:8081/resource-server/warehouse/widgets`

8. On the desktop, double-click the **Fiddler Classic** icon to view the HTTP exchanges.
   a. Click the Filters tab > Request Headers > check **Show only if URL contains**
   b. Enter the following (note the hosts are separated by spaces):
   `localhost:8081 localhost:8083 localhost:8089`
   c. Switch to the Inspectors tab.

9. In Chrome, open **Developer Tools > Application > Storage > Cookies**

10. Now try to access the WarehouseService without authentication:
    a. Browse to `http://localhost:8081/resource-server/warehouse/widgets`
    b. Note the response: HTTP ERROR 401 (Unauthorized).

11. Next, browse to the Angular client at `http://localhost:8089`
    a. Click the **Login** button.

12. Switch to Fiddler and note the requests to localhost:8083 (the authorization server).
    a. Click the first request to URL `/auth/realms/roifmr/protocol/openid-connect/auth`
    b. In the response pane, click the Raw button on the upper right to see the entire HTTP response that was returned by the authorization server.
    c. Click the **Headers** buttons for the request and the response. Note the authorization cookies that are being exchanged.
    d. Switch to Chrome Developer Tools and note the setting of the cookies from localhost:8083.

13. In Chrome, sign into the Angular client:
    a. Login: `john@test.com`
    b. Password: `123`

14. In Fiddler, note the requests to localhost:8089 (Angular client).
    a. The Tunnel request (an encrypted HTTPS request) contains the access token.
    b. Later requests to localhost:8089/ include a cookie with the same access token.

15. In Chrome, note the `access_token` cookie associated with the Angular client.
    a. The client will include this access token in all requests to the Warehouse service.

16. In the Angular client, click **Get Widgets**.
    a. You'll see a table with several widgets, which means the client's request to `http://localhost:8081/resource-server/warehouse/widgets` was successful.

17. In Fiddler, find the request to localhost:8081 that contains an Authorization header.
    a. The header specifies Bearer (Token) Authentication. In this case, the access token is a JSON Web Token (JWT).
    b. We'll discuss JWTs in detail in the next section. For now, copy the access token by right-clicking the **Authorization header > Copy value only**
    c. Paste the copied JWT into an empty text file for use in the next lab exercise.

18. If you wait more than five minutes to click Get Widgets, the token expires, and you get a 401 error. Log out and log in again.

19. When you're finished experimenting with OAuth2, switch to the Angular client and click the **Logout** button.

20. Switch to STS and shut down the resource server and the authorization server.

Exercise 4.2: JSON Web Tokens (JWT)

In this exercise, you will explore the structure of JSON Web Tokens. You will enter data for the header, payload, and secret to generate a JWT. You will also enter a JWT and read the header, payload, and secret details.

Time: 20 minutes
Format: Exploration

1. Go to the following URL with your favorite browser:
   https://jwt.io/

2. The Debugger section of the page shows the encoded and decoded versions of a JWT.

3. Modify the payload.
   a. Enter your name.
   b. Observe that as you type your name, the encoded section is updated automatically.
   c. Add another entry to the payload.
      i. Such as "admin": true.

4. Add a strong secret to the Verify Signature section.
   a. How long does the secret need to be before it is not considered weak?

5. Choose the HS512 algorithm.
   a. Notice how the encoded JWT changes.

6. What is required to use the RS512 algorithm?

This page intentionally left blank.

Chapter 5: Cloud Design Patterns

Exercise 5.1: Researching Cloud Design Patterns

In this exercise, you will research cloud design patterns for particular problems. Your instructor will assign one of the areas listed below.

Time: 30 minutes
Format: Discussion in teams of two

Design Pattern Areas

- https://docs.microsoft.com/en-us/azure/architecture/patterns/
    - Availability
    - Resiliency
    - Performance and Scalability
    - Security
- https://patterns.arcitura.com/cloud-computing-patterns
    - Reliability, Resiliency, and Recovery
    - Monitoring, Provisioning, and Administration
    - Scalability
    - Cloud Service and Storage Security

Choosing a Cloud Design Pattern

1. Choose one of the cloud design patterns that appears in that area.
    a. Describe the problem that this pattern solves.
    b. Describe the pattern and how it solves the problem.
    c. Explain when you would use this pattern.

2. Be ready to discuss with the class.

How is this used or not used at Fidelity?

Exercise 5.2: Using the Circuit Breaker Pattern

In this exercise, you will implement the Circuit Breaker Pattern for a RESTful web service.

Time: 30 minutes
Format: Programming exercise

Using the CircuitBreaker Pattern

1. Open the `BookClient` project in Eclipse.

2. Open the `Bookstore` project in Eclipse.

3. Examine the code in the `Bookstore` project.
   a. This is a simple RESTful service that returns a string of book titles.

4. Run the `Bookstore` project as a Spring Boot application.
   a. Verify that the service starts successfully.

5. Examine the code in the `BookClient` project.
   a. This application uses a RESTful service that makes a call to the Bookstore service.
   b. The `BookClient` application uses a circuit breaker to provide a backup method in case the Bookstore service does not respond.

6. Run the `BookClient` project as a Spring Boot application.

7. Enter the following URL in your favorite browser:
   a. http://localhost:8080/to-read
   b. Verify that a list of book titles is displayed.

8. Now stop the Bookstore application.

9. Refresh the browser display.
   a. You should now see the data that is returned by the backup method.

10. Restart the `Bookstore`.

11. Refresh the browser
    a. The original list of book titles should once again be displayed.

Exercise 5.3: Creating and Deploying a Lambda Function

In this exercise, you will create and deploy a Lambda function.

Time: 30 minutes
Format: Programming exercise

Creating the Lambda Function

1. Open the `LambdaGreeter` project with Eclipse.
   - There will be compile errors initially because of missing dependencies in the `pom.xml` file.

2. In the Project Explorer, right-click the `pom.xml` file.
   - Choose **Maven | Add Dependency**
   - In the **Add Dependency** windows, type the following values:
     i. Group Id: `com.amazonaws`
     ii. Artifact Id: `aws-lambda-java-core`
     iii. Version: `1.2.2`
   - If Eclipse reports an error in `pom.xml`:
     i. Right-click `pom.xml` and choose **Maven | Update Project | OK**.
     ii. Close and re-open the `pom.xml` editor window.

3. Examine the `Greeter` and `Greetings` classes.

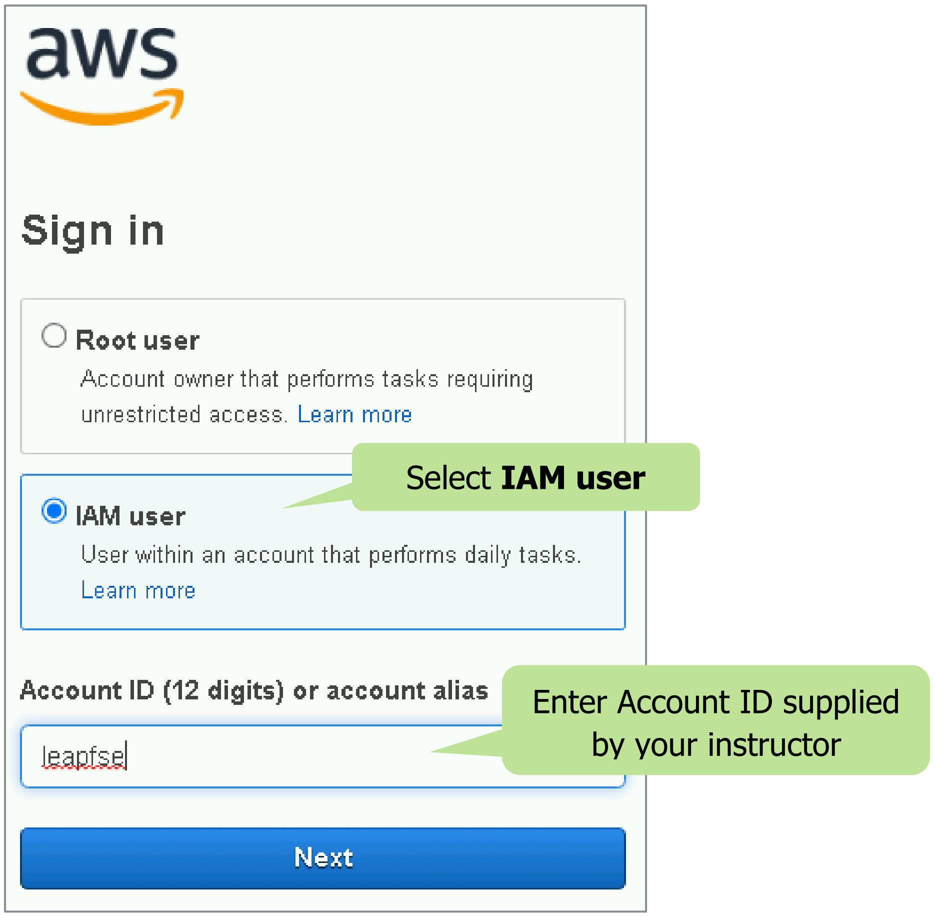
4. Build the project:
   - Right-click the project.
   - Choose **Run As | Maven Build …**
   - In the **Edit Configuration** window, type `package` in the **Goals** box.
   - Choose **Run**.
   - After the build is complete, right-click the project and select **Refresh**. Open the `target` subdirectory and note the JAR file created by the build process.

5. Right-click the `pom.xml` file.
   - Choose **Maven | Add Plugin**.
   - In the **Add Plugin** window, type the following values:
     i. Group Id: `org.apache.maven.plugins`
     ii. Artifact Id: `maven-shade-plugin`
     iii. Omit the version number because Spring Boot will manage this plugin.

6. Right-click the project.
   - Choose **Run As | Maven Build ...**
   - In the **Edit Configuration** windows, type `package shade:shade` in the **Goals** box.
   - Choose **Run**.
   - After the build is complete, refresh the project again.

7. The new standalone "uber" JAR file (which is the deployment package) is in the `/target` subdirectory.

8. Open the AWS Lambda Console. (Be sure to use a browser in your RVC.)
   - Browse to `https://console.aws.amazon.com/`.
   - Select **IAM user**.
   - Log in with the credentials supplied by your instructor.



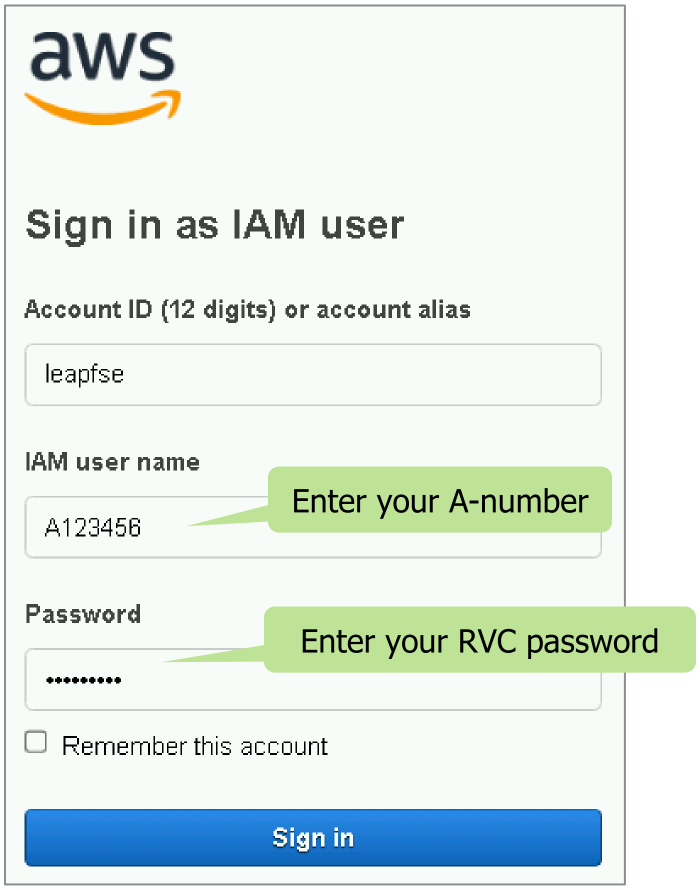
Sign in

○ Root user
Account owner that performs tasks requiring unrestricted access. Learn more

● IAM user
User within an account that performs daily tasks. Learn more

Account ID (12 digits) or account alias

leapfse



Sign in as IAM user

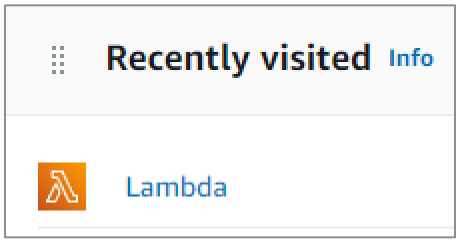
Account ID (12 digits) or account alias: leapfse

IAM user name: A123456

Password: ••••••••••

Remember this account

9. After you're signed in, click the **Lambda** link.



10. Click the **Create a Function** button.

11. In the **Function name** text box, type A######`LambdaGreeter`.

12. Choose **Java 11 (Corretto)** as the Runtime.

13. Click **Advanced settings**.

14. Select **Enable function URL**.

15. For the Auth type, select **NONE**.

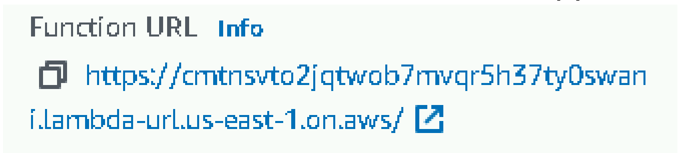
16. Click the **Create Function** button.

17. In the configuration window that appears, in the **Code source** section, select **Upload from | .zip or .jar file** option.
    - Ignore the warning "The code editor does not support…".

18. Click the **Upload** button and select the shaded JAR file that was created earlier (`/target/LambdaGreeter-1.0.0-shaded.jar`).
    - Click the **Save** button.
    - Ignore the warning "The deployment package…is too large."

19. Scroll down to the **Runtime settings** section and click the **Edit** button.
    - In the **Handler** text box, enter `com.fidelity.lambda.Greeter::handleRequest`.
    - Click the **Save** button.

20. In the **Function overview** section, copy the function URL:



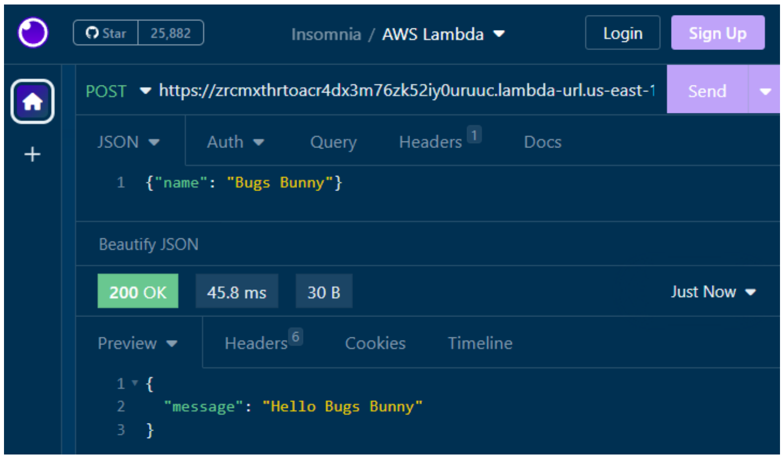
Function URL Info

https://cmtnsvto2jqtwob7mvqr5h37ty0swani.lambda-url.us-east-1.on.aws/

21. Your lambda function is now deployed and will respond to HTTP requests. Next, you will test your function with Insomnia.

Testing Your Lambda Function

22. Open Insomnia and send a POST request to your lambda's functions URL.
    - Select a body type of JSON.
    - Enter a JSON string with a `name` element and value.
    - Verify that the response has a status of 200 and a well-formed JSON body.



```
{"name": "Bugs Bunny"}
```

```
{
  "message": "Hello Bugs Bunny"
}
```

23. Experiment with returning different statuses, different messages, etc., from your lambda function.
    - Each time you modify your function, remember to rebuild the shaded JAR file, and upload it using the AWS Lambda console.
    - On an actual project, you would automate the upload step by using the AWS Command Line Interface (CLI) function `update-function-code` instead of the console GUI.

Bonus Exercise (to be attempted if time permits)

24. Copy the `Gadget` and `Product` classes from the Warehouse project to the current project's `com.fidelity.lambda` package.

25. Modify your lambda function:
    a. Use the `ObjectMapper` to read the value of a Gadget instance from the body of the JSON request.
    b. Decrease the Gadget's price by 10%
    c. Return the modified Gadget in the body of the response.
    Hint: the Gadget in the return value of the function will be converted to JSON automatically.

26. Rebuild the shaded JAR file, upload it with the AWS Lambda console, and test the function with Insomnia.

This page intentionally left blank.

Chapter 6: Node.js

Exercise 6.1: Hello World

In this exercise, you will create the obligatory first project for any new programming endeavor. In this case, you will create a very simple Node.js application that will display the standard greeting message.

Use Visual Studio Code for this and all other node exercises.

Time: 10 minutes
Format: Programming exercise

1. Navigate to the `RESTfulServices\Chapter6` folder.

2. Create a file named `hello.js`.

3. Type the following inside `hello.js`:

```
console.log("Node says Hello World");
```

4. Open a command window in `RESTfulServices\Chapter6`.

5. Type the following inside the command window:

```
node hello.js
```

6. Verify that the message is displayed in the command window.

7. You can also execute the program from Visual Studio Code: in the **Run** menu, select **Run Without Debugging**.

Exercise 6.2: Creating a Server

In this exercise, you will create a very simple server and run that server with Node.

Time: 10 minutes
Format: Programming exercise

1. Return to the `RESTfulServices\Chapter6` folder.

2. Create a new file named `server.js`.

3. Type the following inside `server.js`:

```
const http = require('http');
const server = http.createServer((req, res) => {
    res.end('Hello World from the Server');
});
server.listen(8081);
```

4. Open a command window in the `RESTfulServices\Chapter6` folder.

5. Enter the following on the command line:

```
node server.js
```

6. Browse to http://localhost:8081 in any browser.
   a. Verify the message is displayed in the browser.

7. When you're done, use CTRL+C in the command window to close the server.

Exercise 6.3: Returning HTML

In this exercise, you will create a very simple server that returns HTML and run that server with Node.

Time: 10 minutes
Format: Programming exercise

1. Return to the `RESTfulServices\Chapter6` folder.

2. Create a new file named `serverHtml.js`.

3. Type the following inside `serverHtml.js`:

```
let http = require('http');
let server = http.createServer( (req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html'});
    res.write('<h1>Hello Node World</h1>');
    res.end();
});
server.listen(8081);
```

4. Use node to run `serverHtml.js`.

5. Using your browser, verify the server is operating as expected.

6. Use Chrome Developer Tools (push `F12` key or right-click on page and choose **Inspect**) to verify existence of `<h1>` object in the response from the server.

7. When you're done, use CTRL+C in the command window to close the server

Exercise 6.4: Using a Core Module

In this exercise, you will write a Node.js application that loads a core module and uses some of its functionality.

Time: 10 minutes
Format: Programming exercise

1. Create a new file `content.html` in the `RESTfulServices\Chapter6` folder.

2. Add HTML content that will define a simple (but complete) web page.

   Your page may look something like this:

```
<html>
      <head>
        <title>Who Moved My Cheese?&#153;</title>
      </head>
      <body>
        <header>
          <h1>A simple parable that reveals profound truths</h1>
        </header>
        <article>Dealing with Change in Work and Life</article>
        <footer>Copyright Sniff and Scurry &copy;</footer>
      </body>
</html>
```

3. Save the file.

4. Create a new file named `serverContent.js`.

5. Type the following inside `serverContent.js`:

```
const fs = require('fs');
const http = require('http');
const server = http.createServer( (req, res) => {
    fs.readFile('content.html', (err, fileData) => {
        res.writeHead(200, { 'Content-Type': 'text/html'});
        res.write(fileData);
        res.end();
    });
});
server.listen(8081);
```

6. Use node to run `serverContent.js`.

7. Then browse to http://localhost:8081.

8. Verify the html content defined in `content.html` is displayed in the browser.

9. When you're done, use CTRL+C in the command window to close the server.

Exercise 6.5: Using npm

In this exercise, you will use npm to create and initialize the `package.json` file for a new Node application.

Time: 10 minutes
Format: Programming exercise

1. Open a command window in the `RESTfulServices\Chapter6` folder.

2. Run the following command: `npm init`

3. Enter the bold text below at the appropriate prompts, using the `Return/Enter` key to accept defaults for all other prompts:

package name:	(chapter6) **`hellonpm`**
description:	**`A simple Node server`**
author:	**`Your Name`**

4. Examine the file `package.json` that has been generated in the folder. *Note:* the *start* script calls the command *node server.js*.

5. Let's add the express library: `npm install express`

6. Edit the `server.js` file.

7. Type the following inside `server.js`:

```
let express = require('express');
let app = express();

app.get('/', (req, res) => {
    res.send('Hello Express');
});

app.listen(8081, () => {
    console.log('App listening on port 8081');
});
```

8. Enter `npm start` at the command line.

9. Then in your browser view http://localhost:8081.

10. When you're done, use CTRL+C in the command window to close the server.

This page intentionally left blank.

Chapter 7: Node.js and Express

Exercise 7.1: Debugging Node.js

In this exercise, you will try out the Visual Studio Code debugging feature.

Time: 30 minutes
Format: Programming exercise

1. The instructor will demonstrate a simple debugging example using the `Node_JavaScript\Chapter7\pi.js` program.

2. Explore more debugging capabilities on your own.

Exercise 7.2: Creating a RESTful API with Express

In this exercise, you'll use Node.js and Express to implement a RESTful API.

Time: 30 minutes
Format: Programming exercise

1. In Visual Studio Code, open the `Node_JavaScript\RESTfulServices\Chapter7\SimpleService` folder.

2. Edit `package.json` and add the following line in the `"scripts"` element above the `"test"` element (don't forget the trailing comma):

   ```
   "start": "node src/service/contact-rest-controller.js",
   ```

3. Open a terminal window and run the following command to install Express:

   ```
   npm install express
   ```

4. Note that `npm` modified `package.json` by adding a dependency for Express.

5. Copy the file `dao/mock-product-dao.js` to `dao/mock-contact-dao.js`.

6. Edit `mock-contact-dao.js` and modify it as necessary so it works for contacts.

   Hint: use the contact information in `data/contacts.json`.

7. Copy the file `service/product-rest-controller.js` to `service/contact-rest-controller.js`.

8. Modify `contact-rest-controller.js` so it responds appropriately to the following HTTP requests:
   a. GET all contacts.
   b. GET one contact.
   c. POST one contact.

   Note: in the `addContact()` method, you don't need to test every property of the contact argument; just select a few required properties.

9. Start `SimpleService` by running `npm start`.

10. Test your API by sending it HTTP requests with Insomnia.

Bonus Exercise (to be attempted if time permits)

11. Modify `contact-rest-controller.js` so it responds appropriately to the following HTTP requests:
    a. PUT one contact.
    b. DELETE one contact.

Bonus Exercise (to be attempted if time permits)

In a more complex API, you'll need to configure many routes. So, a best practice is to keep the routing configuration in a separate module and import that module in the controller module.

In this bonus exercise, you'll refactor the routing configuration into a separate module and import that module in the controller's module:

12. Under the `src` directory, create a subdirectory named `routes`.

13. In the `routes` directory, create a file named `contact-routes.js`.

14. Make the following changes in `contact-routes.js`:
    a. Import Express as you did in `contact-rest-controller.js`.
    b. Define a global function named `router` that has one parameter named `controller`.
    c. Cut the `Router` configuration from `contact-rest-controller.js` and paste it into the `router()` function in `contact-routes.js`. (Be sure *not* to cut the call to `this.app.use('/', router)` from `contact-rest-controller.js`.)
    d. Make the following additional changes to `contact-routes.js`:
        i. In the `router()` function, change all references of `this` to `controller`.
        ii. Add a r`eturn` statement to `router()`:
        `return router;`
        iii. Add the following after the function definition to export the `router` function:
        `module.exports = router;`

e. Your completed router module will look something like this:

```
const express = require('express');

function router(controller) {
 const router = express.Router();
 router.get('/contacts',
            controller.getAllContacts.bind(controller));
 …
 return router;
}

module.exports = router;
```

15. Make the following changes in `contact-rest-controller.js`:
    a. Import your new module:
       `const router = require('../routes/contact-routes');`
    b. Change the Express app's router configuration to the following:
       `this.app.use('/', router(this));`

16. Restart the application and use Insomnia to verify the RESTful service works as before.

Exercise 7.3: Testing a RESTful API with Jasmine

In this exercise, you'll use Jasmine to implement tests for the RESTful API you created in the previous exercise.

Time: 30 minutes
Format: Programming exercise

1. In Visual Studio Code, open a terminal in the `Node_JavaScript\RESTfulServices\Chapter7\SimpleService` directory.

2. Run `npm install jasmine --save-dev`.

3. Generate a Jasmine configuration file by running the following command, which creates the `jasmine.json` file in a new subdirectory `spec/support`:

   ```
   npx jasmine init
   ```

4. Edit `package.json` and change the value of the `"test"` script to the following:

   ```
   "test": "jasmine spec/service/contact-rest-controller.spec.js"
   ```

5. Copy `spec/service/product-rest-controller.spec.js` to `spec/service/contact-rest-controller.spec.js`.

6. Edit `contact-rest-controller.spec.js` and make the following changes:
   a. Change the `require` statement so it loads `ContractRestController` from `../../src/service/contact-rest-controller`
   b. Rename the `testWidgets` const to `testContacts` (*Hint:* use a "smart" rename: right-click `testWidgets` and select Rename Symbol.)
   c. Delete the list of initial values for `testWidgets`.
   d. Edit `data/contacts.json`. Copy the contents of `contacts.json` to the `testContacts` value:

      ```
      const testContacts = [{
         "id": 1,
         "firstname": "Joe",
         ...
         "groups": [ "Dev" ]
      ];
      ```

7. Modify the statements in the `beforeEach()` callback:
   a. Edit the `mockDao` spy configuration to match the contact DAO's method names.
   b. Replace the call to the `ProductRestController` constructor with a call to the `ContactRestController` constructor.

   c. Change the name of the controller's DAO property.

8. In the first nested `describe()` call (the first argument is "retrieve all widgets"), change the first `it()` call to `fit()` to "focus" on that one spec; i.e., when you run `npm test`, only that spec will be executed.

9. In the spec:
   a. Change the name of the `mockDao` method that is being configured.
   b. Change the controller method that is being called.

10. Run the spec and verify it passes: `npm test`.

    a. Note the reminder from Jasmine that not all specs were executed: "`Incomplete: fit()` or `fdescribe()` was found".

11. Now change the second `it()` to `fit()`. Modify the spec as necessary and confirm that both specs pass.

12. Change the two `fit()` calls back to `it()`. Examine the second nested `describe()`. Change the first `it()` call to `fit()`. Modify the spec as necessary to make it pass.

Bonus Exercise (to be attempted if time permits)

13. Continue working on the remaining specs. Make as many passes as possible in the remaining exercise time.

Bonus Exercise (to be attempted if time permits)

14. If you haven't completed the second bonus exercise step from Exercise 7.2 (creating a routing module), complete that step now.

Chapter 8: Promises and Testing Node with Jasmine

Exercise 8.1: Using JavaScript Promises

In this exercise, you will further explore the use of Promises in JavaScript. You'll make calls with asynchronous functions in the Axios module and handle the Promises returned by those functions. You'll call the Axios functions from Jasmine specs so you can easily verify that your HTTP calls are correct.

Time: 30 minutes
Format: Programming exercise

1. Make sure `SimpleService` is running and listening on port 3000.

2. Open `TestSimpleService` project in `Node_JavaScript\RESTfulServices\Chapter8` and edit the file `axios.spec.js`.

3. Your test suite needs a way to reset the mock database back to its initial state before each spec. To make that possible, the `SimpleService` controller defines a method `restart()`, which is bound to the HTTP OPTIONS method. (It uses OPTIONS to avoid confusion with the API's public CRUD operations.)

   a. Add the following call to `beforeEach()` inside the `describe()` callback:

   ```
   beforeEach(async () => {
       await axios.options(restartUrl);
   });
   ```

4. Complete the implementation of the spec for “GET all contacts”. Use the TODO comments in the source file as a guide.

5. Run `npm test` and confirm the spec is successful.

6. Complete the implementation of the spec for “GET one contact” by completing the tasks in the TODO comments. Run `npm test`.

7. Complete the implementation of the spec for “POST a new contact”. Confirm the spec is successful.

Bonus Exercise (to be attempted if time permits)

8. Complete the implementation of the specs for “PUT an existing contact” (updates an existing contact) and “DELETE one contact”.

Bonus Exercise (to be attempted if time permits)

9. Every Axios function can take an `config` argument that configures the HTTP request. For example, you can specify header values, query parameters, a timeout, etc. Your task: add appropriate HTTP headers to the Axios calls. For example:

```
const reqHeaders = {
      'Accept': 'application/json',
};
const resp = await axios.get(baseUrl, { headers: reqHeaders });
```

Bonus Exercise (to be attempted if time permits)

10. Open `axios.spec.js`.

11. Examine the methods under the BONUS TODO comment. They have replaced `async` and `await` with explicit calls to `Promise.then()` and `catch()`. Note that the code is considerably more complex than the equivalent code with `async` and `await`. If you are using ES8 (ECMAScript 2017) or later, you will probably prefer to work with `async` and `await` whenever possible.

Exercise 8.2: Calling a Back-End RESTful API

In this exercise, you'll add calls in your mid-tier Contact API to a back-end API implemented with Java. You'll replace calls to the DAO with Axios calls to the back-end service.

Time: 30 minutes
Format: Individual programming exercise

1. If `SimpleService` is running, shut it down with CTRL+C.
2. In STS or Eclipse, open the project `ContactApi` in the RESTful workspace.
3. Right-click `ContactApiApplication.java` in the `com.roifmr.contactapi` package and select **Run As | Java Application**.
    a. If you get compile errors in the project, right-click `pom.xml` and select **Maven > Update Project > OK**.
4. Use Insomnia to confirm that the back-end service is handling HTTP requests at `http://localhost:8080/backend-api/contacts`.
5. In Visual Studio Code, open the project `Node_JavaScript\RESTfulServices\Chapter8\SimpleServiceMidtier`.
6. Open a terminal in the `SimpleServiceMidtier` folder.
7. Install Axios: `npm install axios`
8. Edit `src/service/contact-rest-controller.js`.
9. Complete the first four TODO steps in the file (up to the TODO in the `getAllContacts()` method).
10. Start the Node.js service with `npm start`.
11. Use Insomnia to send a GET request to your service at `http://localhost:3000/contacts`.
    a. Confirm that your service returns the list of contacts it received from the Axios call to the back-end Java API.

12. Continue with the remaining TODO steps in `contact-rest-controller.js`. Complete one controller method at a time. After completing each method, restart and test your Node.js service:
    a. Stop the service with CTRL+C.
    b. Run `npm start`.
    c. Use Insomnia to send the appropriate HTTP request to your service at `http://localhost:3000/contacts`.
    d. Confirm that the service used Axios to send the correct request to the back-end Java API.

Bonus Exercise (to be attempted if time permits)

In this bonus exercise, you'll write integration tests for the Product CRUD API.

13. Open SQL Developer and review the contents of the WIDGET and GADGET tables in Oracle.

    a. The DAO in this bonus project interacts with these tables.

14. Shut down `SimpleServiceMidtier` with CTRL+C.

15. In Visual Studio Code, open the project `Node_JavaScript\RESTfulServices\Chapter8\NodeProductApi`.

16. Edit `package.json` and note the script `test-controller-integration`, which runs Jasmine to execute the controller integration tests.

17. Open a terminal in the `NodeProductApi` folder and run the following commands:
    ```
    npm install

    npm start
    ```

18. Verify the product service is handling requests by browsing to `http://localhost:3000/widgets`

19. Open another terminal for running your specs.

20. Edit `spec\controllers\product-rest-controller.integration.spec.js`.

21. Write integration tests by completing the TODO steps in the spec file. After completing each spec, run Jasmine and verify that your new spec passes:

    ```
    npm run test-controller-integration
    ```

Bonus Exercise (to be attempted if time permits)

In this bonus exercise, you'll refactor the product controller by replacing its dependencies on the DAO and the transaction manager with a single dependency on a business service. The business service implements the business use cases, which include defining the transaction boundaries. The controller will then have a single responsibility, which is to handle HTTP requests.

The integration tests you wrote in the previous step will function as a suite of regression tests. Because your regression suite thoroughly tests all the controller's code, you can fearlessly refactor your code to improve the design without worrying about introducing bugs.

22. Run Jasmine again to verify that the specs in your regression test suite all pass:
    ```
    npm run test-controller-integration
    ```

23. Open `src\business-service\business-service.js`.
    a. Note that it has dependencies on both a DAO and a transaction manager.

24. Refactor `ProductRestController` to delegate business logic to the business service:
    a. Add a `require()` call for the business-service module.
    b. Delete the `require()` calls for the transaction manager and DAO.
    c. Add a property for a `BusinessService` and initialize it by calling the BusinessService constructor.
    d. Remove the `ProductDao` and `TransactionManager` properties from the controller.
    e. Refactor the `ProductRestController` methods to call `BusinessService` methods instead of DAO methods.
    f. Because the `BusinessService` manages transactions, you can delete all transaction management from the `ProductRestController` methods.

25. Run the integration tests again and confirm that all specs still pass.
    a. As a result of the refactoring, the controller is now simpler, and it focuses only on handling HTTP requests.

26. Run the controller unit tests. Note that many of them now fail.
    a. This highlights a drawback of using mock objects in testing: any restructuring that alters a class’s dependencies will necessitate updates to the mock objects used in unit tests.
    b. Nonetheless, the benefits of testing with mock objects generally surpass this limitation.

Chapter 11: Service Virtualization

Exercise 11.1: Using Service Virtualization

In this exercise, you will use service virtualization in application development.

Time: 20 minutes
Format: Programming exercise

1. Open the `MessageService` STS project.

2. Run the `MessageService` project as a Spring Boot application.

3. Verify the `MessageService` is running correctly.
   a. View the following URL in your favorite browser:
      http://localhost:9080/message
   b. The response should be displayed in JSON format.

4. Open the `MessageClient` Eclipse project.

5. Run the `MessageClient` project as a Spring Boot application.

6. Verify the `MessageClient` is running correctly.
   a. View the following URL in your favorite browser:
      http://localhost:8080/invoke
   b. The response should be the same as in Step 3.

7. Use Hoverfly for service virtualization.

8. Open a command window.
   a. Change to the Hoverfly directory (`C:\Apps\Hoverfly`)
      Start Hoverfly by running the following command: `hoverctl start`

9. Switch Hoverfly to capture mode by running the following command:
   a. `hoverctl mode capture`
   b. Open the Hoverfly admin web page by viewing the following URL in any old browser:
      http://localhost:8888/dashboard

10. Refresh the request from the `MessageClient` several times.
    a. Hoverfly will capture the requests and the responses.
    b. The Capture counter will be updated in the Hoverfly admin web page.

11. Export the captured requests by running the following command:
    a. `hoverctl export simulatedMessages.json`

12. To import the captured requests, run the following command:
    a. `hoverctl import simulatedMessages.json`

13. Switch Hoverfly to simulate mode by running the following command:
    a. `hoverctl mode simulate`

14. Refresh the request from the `MessageClient` several times.
    a. Notice that the response is from the captured request file.
    b. The Simulate counter will be updated in the Hoverfly admin web page.

15. Stop the `MessageServer` application.

16. Refresh the request from the `MessageClient` several times.
    a. Verify that the response is displayed even though the `MessageServer` is no longer running.
    b. This illustrates the use of a simulator (aka virtual service).

Chapter 12: Testing with Cucumber.js

Exercise 12.1: Writing an Acceptance Test with Gherkin Syntax

Time: 10 minutes
Format: Paper exercise

Description:

1. In this exercise, you will write acceptance tests using Gherkin syntax.

2. Work as a team. If you're working onsite, form teams of two (client and developer).

3. The user goal is to purchase a new cell phone from Amazon.

4. First, develop the user story.

5. Next, describe the story in a feature file. Write the feature description using BDD syntax:

   As a *<role>*
   I want to *<task>*
   So *<business value>*

6. Write a scenario for an acceptance test using Gherkin's Given…When…Then syntax.

7. As with all types of testing, your acceptance tests should include multiple paths through the application. Keeping that in mind, write additional scenarios with different "Given's" or "When's".

Feature:

As a … I want to … So …

Scenario:

Given … When … Then

Exercise 12.2: Test Your Calculator

Time: 20 minutes
Format: Programming exercise

1. Navigate to the `RESTfulServices\Chapter12\calculator` folder.

2. First, open `calculator.js` and examine the `Calculator` class.
   a. This is the module you'll be testing.

3. Open `features/calculator.feature` and examine the feature that we need to implement.
   a. Note there is currently one simple scenario defined.

4. Open `step-definitions/calculator-steps.js`. This is your test suite.
   a. Note that the code is initially all commented out.

5. Open `package.json` and note that the "test" script runs Cucumber with your feature file and step file as inputs.

6. Open a terminal in the `calculator` folder and run `npm install`.

7. Run `npm test` before making any changes.
   a. Because Cucumber doesn't find any step definitions in `calculator-steps.js`, its output includes stubs of the function calls required to verify the scenario in `calculator.feature`.

8. In `calculator-steps.js`, uncomment the function calls on lines 1 – 18.

9. Run `npm test` again.
   a. Cucumber now finds Given/When/Then function calls in `calculator-steps.js` that match the Given/When/Then in `calculator.feature`, so the simple scenario passes.

10. Next, you'll add a scenario outline, which allows you to easily test multiple inputs and outputs without writing separate When/Then functions for each.
    a. In `calculator.feature`, uncomment the second scenario and the sample data (lines 11 – 26).
        i. Note that the column headers in the example data table (`num1, num2, total`) match placeholders in the scenario outline (`<num1>, <num2>, <total>`).
        ii. When Cucumber calls the When and Then callback functions, it passes values from the data table as arguments to the callback functions.

    b. In `calculator-steps.js`, comment out the When/Then for the simple scenario (lines 12 – 18).

    c. Uncomment the When/Then for the scenario outline (lines 20 – 26).

    d. Save both files.

    e. Run `npm test` and note that 10 scenarios now pass.

11. This technique of creating a table of example inputs and outputs is known as Specification by Example. It's a very useful tool for gathering and verifying requirements from all stakeholders of a project.

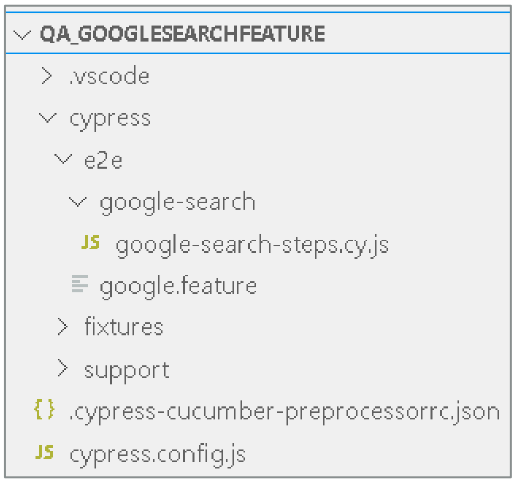
Demo 12.3: Who Is the CEO?

Time: 20 minutes
Format: Instructor-led demonstration

Description:
Build a Cucumber/Gherkin-based test to search in Google for the CEO of Fidelity and find Abigail Johnson.

1. Go through the steps as laid out in the course material.

2. Find the following files ready for you in the folder `RESTfulServices\Chapter12\QA_GoogleSearchFeature`



```
QA_GOOGLESEARCHFEATURE
  > .vscode
  v cypress
    v e2e
      v google-search
        google-search-steps.cy.js
      google.feature
    > fixtures
    > support
  .cypress-cucumber-preprocessorrc.json
  cypress.config.js
```

3. Don't forget to install all modules with: `npm install`

4. To fix potential issues run: `npm audit fix`

5. Then run your tests with: `npm test`

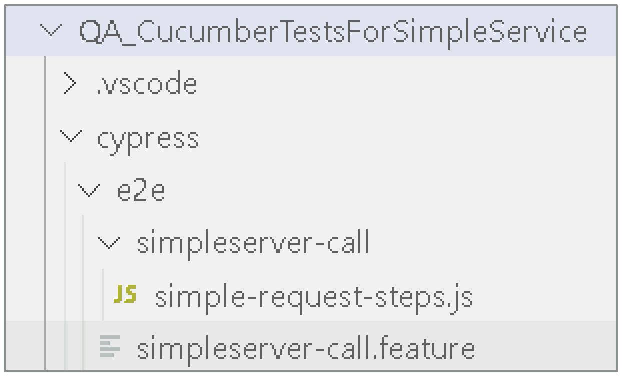
Exercise 12.4: Let's Test SimpleService BDD Style

Time: 30 minutes
Format: Programming exercise

Description:
Write Cucumber/Gherkin-based specs to test the `SimpleService` API.

1. Before testing, start the `SimpleService` that you will be testing.
   a. Open the `SimpleService` project in `Node_JavaScript\RestfulServicesSolutions\Chapter7`
   b. Open a terminal in the `SimpleService` folder and run these commands:
      i. `npm install`
      ii. `npm start`
   c. Verify the service is running by browsing to the following URLs:
      i. `http://localhost:3000/contacts`
      ii. `http://localhost:3000/contacts/2`

2. In Visual Studio Code, open the folder `RESTfulServices\Chapter12\QA_CucumberTestsForSimpleService`

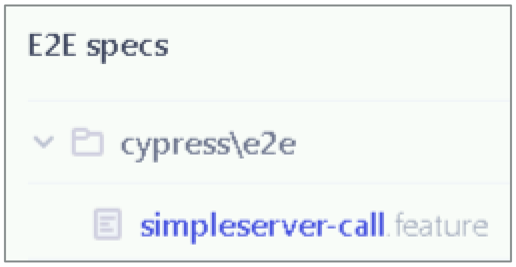


3. Open a terminal in `QA_CucumberTestsForSimpleService` and run `npm install`

4. Edit `cypress\e2e\simpleserver-call.feature`.
   a. Complete the TODO steps in file (ignore the BONUS TODO step for now).

5. Edit `cypress\e2e\simpleserver-call\simple-request-steps.js`.
   a. Complete the TODO steps in the file (ignore the BONUS TODO step for now).

6. Run your Cucumber test with: `npm test`
   a. The Cypress test runner will open a new Chrome window. Click the link to `simpleserver-call.feature` to run your Cucumber steps.



E2E specs

cypress\e2e

simpleserver-call.feature

   b. Verify that the spec passes.

Bonus Exercise (to be attempted if time permits)

7. Complete the BONUS TODO steps in `simpleserver-call.feature` and `simple-request-steps.js`.

8. Run `npm test` again and verify the new spec passes.

Chapter 13: Server-Side JavaScript Programming

Exercise 13.1: Factories

In this exercise, you will explore code and add variables and functions to subclasses.

Time: 30 minutes
Format: Individual programming exercise

The CarMaker example in the Course Notes defines static factory method that creates JavaScript objects based on the type argument that is passed to that method. The factory method creates a new CarMaker object and returns it.

It would be better to have the CarMaker create a Car object. In this exercise, you will do exactly that.

1. Work in the `RESTfulServices\Chapter13\CarFactory` directory.

2. Define a Car class that contains the following data:
   a. The number of seats
   b. A description of the car
   c. The number of doors

3. The Car class should also define the following functions, each of which should print a message to the console:
   a. start
   b. drive
   c. stop

4. Define some specific car models such as the following:
   a. VW Bug
   b. Jeep Cherokee
   c. Tesla Model S

5. Define a CarFactory class that can create a Car of any model type.

6. Write tests to verify the CarFactory operates correctly.

Exercise 13.2: Exploring Closures

In this exercise, you will create a Counter class that uses closures to modify the value of a private data field when public methods are called.

Time: 20 minutes
Format: Individual programming exercise

1. Work in the following folder `RESTfulServices\Chapter13\Closures`.

2. Define Counter class that contains the following data:
   a. A privateCounter that will store a numeric value.

3. The Counter class should define a private function:
   a. A function named modify that has one argument.
   b. The argument is added to the privateCounter.

4. The Counter class should return the following public methods:
   a. Increment
      i. Calls modify(1)
   b. Decrement
      i. Calls modify(-1)
   c. Value
      i. Returns the current value of privateCounter

5. Write tests to verify that the Counter operates correctly.

6. Verify that when two Counter objects are created, each one has its own privateCounter value that is independent of the other.

Optional Exercise 13.3: Iterators and Generators

In this exercise, you will create a generator that calculates the ratio successive terms in the Fibonacci sequence. The example below creates a generator that returns the terms of the Fibonacci sequence. Your task in this exercise is to modify that code to return the ratio of the private data fields that the Fibonacci generator uses. That is, return the ratio of fn2/fn1.

Time: 20 minutes
Format: Individual programming exercise

1. Work in the following folder `RESTfulServices\Chapter13\Generators.`

2. Create a new JavaScript file that defines the Fibonacci sequence generator that is defined in the Course Notes.

3. Write a test to verify that the generator works correctly.

4. Verify that the sequence can be reset.

5. Modify the generator to return the ratio of fn2/fn1.

6. Modify your tests to verify that the modified generator works correctly.

7. If you are not familiar with the golden ratio, do a quick web search to find out why this ratio is of interest.

```
let fibonacci = {
    [Symbol.iterator]() {
        let pre = 0, cur = 1;
        return {
            next() {
                [pre, cur] = [cur, pre + cur];
                return { done: false, value: cur };
            }
        };
    }
}

for (let n of fibonacci) {
    if (n > 1000)
        break;
    console.log(n);
}
```

Chapter 14: Functional and Reactive Programming in JavaScript

Exercise 14.1: Working with Arrays

In this exercise, you will explore more ways to work with arrays in JavaScript.

Time: 20 minutes
Format: Individual programming exercise

1. View the files in the `RESTfulServices\Chapter14\workingWithArrays` folder.

2. View the `workingWithArrays.html` file in your favorite browser.

3. Run the code by clicking the different buttons on the page.

4. Complete the code in the `workingWithArraysAndLists.js` file.

5. View the web page again in your browser.

6. Verify that the code you added works correctly.

Optional Exercise 14.2: Using Higher Order Functions

In this exercise, you will use higher order functions in JavaScript.

Time: 20 minutes
Format: Individual programming exercise

1. View the files in the `RESTfulServices\Chapter14\higherOrderFunctions` folder.

2. View the `higherOrderFunctions.html` file in your favorite browser.

3. Run the code by clicking the different buttons on the page.

4. Complete the code in the `higherOrderFunctions.js` file.

5. View the web page again in your browser.

6. Verify that the code you added works correctly.

Exercise 14.3: Using Function Composition

In this exercise, you will use function composition in JavaScript.

Time: 20 minutes
Format: Individual programming exercise

1. View the files in the `RESTfulServices\Chapter14\functionComposition` folder.

2. View the `functionComposition.html` file in a local web server.
   a. Right-click the `functionComposition.html` file in the Explorer window.
   b. Choose **Open with Live Server**.
   c. The `functionComposition.html` file will be displayed in your default browser hosted on `localhost (127.0.0.1:5500)`.

3. Run the code by clicking the different buttons on the page.

4. Complete the code in the `functionComposition.js` file.

5. View the web page again in the local web server.

6. Verify that the code you added works correctly.

Optional Exercise 14.4: Currying

In this exercise, you will explore the use of currying in JavaScript.

Time: 20 minutes
Format: Individual programming exercise

1. View the files in the `RESTfulServices\Chapter14\currying` folder.

2. View the `currying.html` file in a local web server.
   a. Right-click the `currying.html` file in the Explorer window.
   b. Choose **Open with Live Server**.
   c. The `currying.html` file will be displayed in your default browser hosted on `localhost (127.0.0.1:5500).`

3. Run the code by clicking the different buttons on the page.

4. Complete the code in the `currying.js` file.

5. View the web page again in the local web server.

6. Verify that the code you added works correctly.

Exercise 14.5: Using Observables

In this exercise, you will explore the use of Observables in JavaScript.

Time: 20 minutes
Format: Individual programming exercise

1. View the files in the `RESTfulServices\Chapter14\observables` folder.

2. View the `observables.html` file in a local web server.
   a. Right-click the `observables.html` file in the Explorer window.
   b. Choose **Open with Live Server**.
   c. The `observables.html` file will be displayed in your default browser hosted on `localhost (127.0.0.1:5500).`

3. Run the code by clicking the different buttons on the page.

4. Complete the code in the `observables.js` file.

5. View the web page again in the local web server.

6. Verify that the code you added works correctly.

This page intentionally left blank.

Appendix A: Building RESTful Services with JAX-RS

All the exercises in this section use JAX-RS without Spring Boot. The projects are in the `Using JAX-RS` folder.

Exercise A.1: Exploring the Time Service

In this exercise, you will explore an existing RESTful service. You will use a browser to send a request to the service and view the response. You will also use the Insomnia application to communicate with the service and view the response.

Time: 30 minutes
Format: Programming exercise

1. Import the `TimeService` project.

2. Examine the sources in `com.fidelity.restservices`.

3. Launch the TimeService application by running `index.jsp` on the Tomcat server inside Eclipse.
   a. You may have to create a new Tomcat Server in Eclipse.

4. Follow the link to the REST-based TimeService.
   a. View the response in the browser.

5. Copy the URL from the browser address box.

6. Using your favorite browser send the request to the web service.

7. View the response from the web service.
   a. Notice that the response is in XML format.

8. Use the Insomnia application to communicate with the TimeService.
   a. Enter the URL for the TimeService.
   b. Send a `GET` request
   c. View the response from the web service.

9. Try sending a `GET` request for the time for a specified time zone.
   a. Provide a valid time zone ID.
   b. View the response from the web service.
   c. Provide an invalid time zone ID.
   d. How does the web service respond?

Exercise A.2: Creating a RESTful Service

In this exercise, you will create a RESTful service and test it with the Insomnia.

Time: 30 minutes
Format: Programming exercise

1. Import the `ExhibitsService` project.

2. Create a RESTful service by defining the two methods in `ExhibitsService.java`.
   a. The `ExhibitsService` should call on the `MockExhibitDao` for the exhibit data.
   b. The data should be returned in JSON format.

3. Run the `ExhibitsService` on the Tomcat server.

4. Verify the `ExhibitsService` works as expected by using your favorite browser.

5. Run the service on Tomcat.

6. Test the `ExhibitsService` using your favorite browser.

7. Test the `ExhibitsService` using Insomnia.

Exercise A.3: Testing a RESTful Service

In this exercise, you will continue working with the project from the previous exercise. You will use JUnit to test the RESTful service as a plain old Java object. You will then use Insomnia to send HTTP messages to the RESTful service and examine the responses returned by the service.

Time: 40 minutes
Format: Programming exercise

1. Continue working with the `ExhibitsService` project that you used in the previous exercise.

2. Test the `ExhibitsService` as a POJO by writing a JUnit test.
   a. Verify the service methods return the expected Java objects.

3. Test the service using Spring support.
   a. Open the `ExhibitsServiceTest` source file.
      i. Examine the annotations on the test class.
      ii. Notice that the `ExhibitsService` is autowired into the test.
   b. Complete the testQueryAllExhibits test.
   c. Complete the testSearchFor test.
   d. Of course, you get the green bar, right?

4. Modify the `@Produces` annotation on the two web methods so that the methods will return either XML or JSON.
   a. The format of the data depends on the client request.

5. Test the service using Insomnia.
   a. Verify that the service will return either XML or JSON depending on the Accept header sent in the HTTP request.

Exercise A.4: Integrating Spring with JAX-RS

In this exercise, you will integrate Spring into a RESTful service. This will allow you to have Spring inject a dependency into the RESTful service.

Time: 30 minutes
Format: Programming exercise

1. Import the `WarehouseService` .

2. Open the `web.xml` file.

3. Examine the definition of the `SpringJaxrsServlet`.

4. Modify the `WarehouseService` to be a RESTful service that calls on the `WarehouseDAOMyBatisImpl` to communicate with the database.
   a. Define a web method that will return all of the widgets from the database.
   b. Define a web method that will return all of the gadgets from the database.

5. Test your service.
   a. First as a POJO with JUnit.
   b. Then with Insomnia.