Fidelity INVESTMENTS

DEVELOPING RESTFUL SERVICES

EXERCISE MANUAL

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 Advanced Rest Client (ARC) to communicate with the service and view the response.

Your instructor will tell you how to import the Eclipse projects for this course. When you import all of the projects in the course zip file, Eclipse will be busy opening, examining, and validating all of the projects. It is best to let Eclipse 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 Eclipse.

Time: 30 minutes
Format: Programming exercise

1. Open the `TimeService` project.

2. Examine the sources in `com.fidelity.restservices`.
   a. Note the URL associated with each web method.

3. Launch the TimeServiceApplication in the `com.fidelity.timeservice` package by running it as a Java application.
   a. Note the port that Tomcat is listening on.

4. Enter the URL for the web method for the current time zone information in your favorite browser.
   a. View the response in the browser.

   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:
   - http://localhost:8080/time/IST is ok.
   - Not http://localhost:8080/time/America/New_York.

   The list of legal Java timezone names is at https://garygregory.wordpress.com/2013/06/18/what-are-the-java-timezone-ids/.

5. View the response from the web service.
   a. Notice that the response is in JSON format.

6. Use the Advanced Rest Client (or Postman) 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.

7. 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 1.2: Creating a RESTful Service

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

Time: 30 minutes
Format: Programming exercise

1. Open the `LibraryService` project.

2. Create a RESTful service by defining the two methods in `LibraryService.java`.
   a. The `LibraryService` 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 Java application.

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

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

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 service 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 web method to return an HTTP status code of `NO_CONTENT` if there are no Books in the database.

3. Modify the web method to return an error code if the DAO throws an exception.
   a. You should decide what is the appropriate error code to return.

4. Use ARC to verify this new functionality.

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 Ribbit site:
   https://ribbit.fmr.com/groups/ecc-cloud-cafe
   a. What does the Cloud Café offer?
   b. Where is it located?

2. In your browser, visit the Confluence documentation at the following link:
   https://itec-confluence.fmr.com/display/AP119867/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 the moment, 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. Use ARC to verify the service methods are working as expected.

Challenge: (if you have time)

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

6. In particular, the service should support a query for all Products, including both Gadgets and Widgets.

Exercise 2.3: 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.
   a. *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 2.4: Debugging a RESTful Service

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

Time: 30 minutes
Format: Programming exercise

1. Open the `BuggyService` project.

Configure the BuggyService to run in debug mode

2. In the Project explorer, right-click the BuggyService and choose **Run As ... / Run Configurations**.

3. Select the **Arguments** tab.

4. In the VM arguments text area, enter the following text:

```
-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=5005
```

5. Click the **Apply** button.

6. Click the **Run** button.
   a. The BuggyService application should now run in debug mode
   b. It is listening for a remote debugger to attach to the application.

Configure a remote debugger to attach to the BuggyService

7. On the Run menu, choose **Debug Configurations ...**

8. Select **Remote Java Application**.

9. Enter `RemoteDebugger` in the Name text box.

10. Use the Browse button to select the BuggyService as the Project.

11. Enter `5005` in the Port text box.

12. Click the **Apply** button.

13. Click the **Debug** button.
    a. The remote debugger should now be attached to the BuggyService application.

Use Eclipse to debug the BuggyService

14. Set a breakpoint in the web service method.

15. In your favorite browser, enter the URL that corresponds to the web service method.
    a. Eclipse should highlight the breakpoint.
    b. Use the Eclipse debugging environment to find the error(s).
    c. Correct it (them).

Shutting down the debugging environment

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

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

This page intentionally left blank.

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 a unit test. You will test the business service with a unit test where you will mock the data access object that the business service depends on. Once the unit tests are passing, you will then create an integration test to verify that the business service and data access object perform correctly together.

Time: 30 minutes
Format: Programming exercise

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

2. Open the WarehouseDaoMyBatisImplUnitTest 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.
   a. And, of course, get a green bar.

5. Open the WarehouseBusinessServiceUnitTest class.

6. Run the tests in this class.
   a. Again, some of the tests have not been implemented.

7. Complete the tests that are not implemented.
   a. And, of course, get a green bar.

8. Open the WarehouseBusinessServiceIntegrationTest class.
   a. Guess what? Some of the tests have not been implemented.

9. You know what to do.

10. 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 `WarehouseServiceWebLayerTest`.

3. Complete the test methods.

4. Run this test as a JUnit test.

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

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 `WarehouseServiceTestRestTemplateTest`.

3. Complete the test methods.

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

This page intentionally left blank.

Chapter 4: Securing RESTful Web Services

Exercise 4.1: Using OAuth2 for Authorization

In this exercise, you will use OAuth2 for authorization in a Spring Boot-based RESTful service.

Time: 20 minutes
Format: Individual or pair programming

1. Open the `OAuth2` Eclipse project.

2. Examine the code in the `AuthorizationServerConfiguration` class.

3. Examine the code in the `WebSecurityConfiguration` class.

4. Run the project as a Spring Boot application.

5. Test the application using a tool like Postman or ARC.
   a. Send a `POST` request to http://localhost:8080/oauth/token
   b. Choose Basic Authentication.
      i. Enter "`client`" for the username
      ii. Enter "`password`" for the password
   c. Send the following key/value pairs in the body of the request (use `x-www-form-urlencoded`):
      i. `password / secret`
      ii. `username / user`
      iii. `grant_type / password`
   d. The response should return JSON including an "`access_token`" property.

6. Examine the code in the `ResourceServerConfiguration` class.

7. Send a `GET` request to http://localhost:8080/public
   a. The response should be visible with a status of 200.

8. Send a `GET` request to http://localhost:8080/private
   a. The response should return a status of 401 unauthorized.

9. Send a `GET` request to http://localhost:8080/private with the following Header set:
   a. `authorization / bearer {access token}`
      i. *Note:* authorization is the key. The value is `bearer {access token}`.
      ii. Where `{access token}` is the "access token" property value obtained earlier.
   b. The response should now be a 200 with data returned.

10. Send a `GET` request to http://localhost:8080/admin
    a. The response should return a status of 403 Forbidden since we do not have admin privileges.

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 `Bookservice`.

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.

2. In the Project Explorer, right-click the `pom.xml` file.
   a. Choose **Maven | Add Dependency**
   b. In the **Add Dependency** windows, type the following values:
      i. Group Id: `com.amazonaws`
      ii. Artifact Id: `aws-lambda-java-core`
      iii. Version: `1.1.0`

3. Examine the Greeter and Greetings classes.

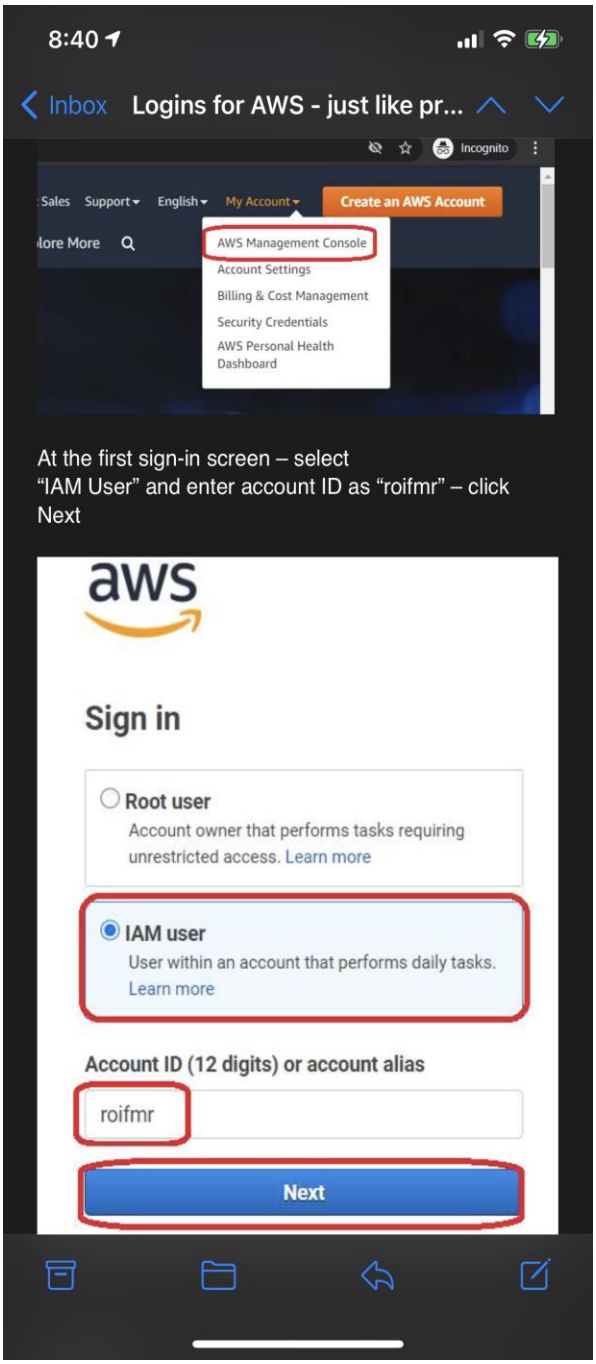
4. Build the project:
   a. Right-click the project.
   b. Choose **Run As | Maven Build ...**
   c. In the **Edit Configuration** window, type `package` in the **Goals** box.
   d. Choose **Run**.

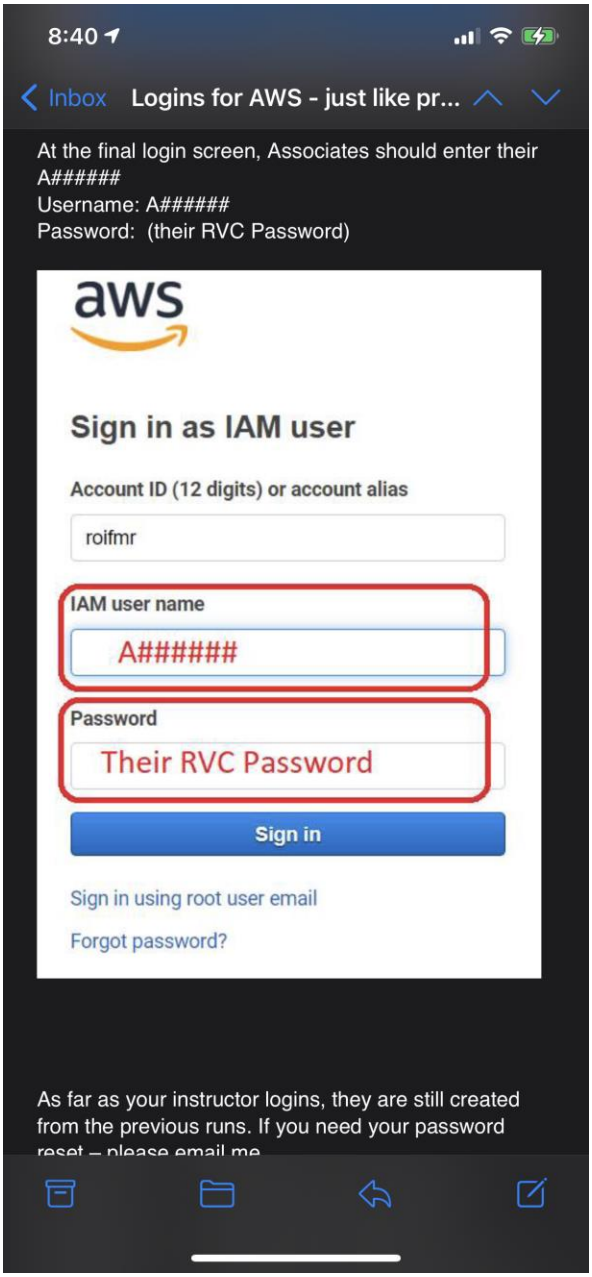
5. Right-click the `pom.xml` file.
   a. Choose **Maven | Add Plugin**.
   b. In the **Add Plugin** window, type the following values:
      i. Group Id: `org.apache.maven.plugins`
      ii. Artifact Id: `maven-shade-plugin`
      iii. Version: `2.3`

6. Right-click the project.
   a. Choose **Run As | Maven Build ...**
   b. In the **Edit Configuration** windows, type `package shade:shade` in the **Goals** box.
   c. Choose **Run**.
   d. This will build the standalone jar file that contains all of the required code and libraries.

7. The standalone jar file (which is the deployment package) is in the `/target` subdirectory.

8. Open the AWS Lambda Console.
   a. Log in with the credentials supplied by your instructor.





9. Click the **Create Function** button.

10. Type `A######LambdaGreeter` in the **Function name** text box.

11. Choose **Java 8** as the Runtime.

12. Click the **Create Function** button.

13. In the **Configuration** window that appears, select **Upload a .zip or .jar file** option in the **Code entry type** select box.

14. Click the **Upload** button and select the jar file that was created earlier (in the `/target` subdirectory).
    a. Under **Function code**, select **Action** and change to **Upload a .zip or .jar file**.
    b. Make sure you upload the shade version of the jar file that you created earlier.

15. Enter `com.fidelity.lambda.Greeter::myHandler` in the **Handler** text box.

16. Click the **Save** button to save this as a Lambda function.

17. Test the Lambda function by clicking the **Test** button.

18. In the **Configure test event** window, replace the JSON text with a simple text string that will be passed as the name parameter.

19. Enter the name `helloEvent` in the **Event name** text box.

20. Click the **Create** button.

21. To run the test, click the **Test** button.

22. Expand the **Details** section to view the response from the Lambda function.

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: 15 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.

Exercise 6.2: Creating a Server

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

Time: 20 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`:

```
let http = require('http');
let 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: 20 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();
});write
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: 20 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`:

```
let fs = require('fs');
let http = require('http');
let 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: 20 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: (begin) **`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 `pi.js` program.

2. Explore more debugging capabilities on your own.

Exercise 7.2: Using the Express Module to Create a Website

In this exercise, you'll use Node Package Manger (npm) to install Express, generate scaffolding that uses jade as the view engine, and run the website in the browser.

Time: 30 minutes
Format: Programming exercise

1. In a terminal window, run the following commands to install express and express-generator:

   ```
   npm install express – g
   npm install express-generator -g
   ```

2. Navigate to the `RESTfulServices\Chapter7` folder in the command prompt and execute the following command to generate an Express site named `ExpressSite`:

   ```
   express ExpressSite
   ```

3. Change the command's location to the new `ExpressSite` folder that's now created.

4. Run the following command to install dependencies locally within the `ExpressSite` folder:

   ```
   npm install
   ```

5. Open the `app.js` file at the root of `ExpressSite` in an editor and take a moment to look through the code. Notice the following features:
   a. Multiple modules are used including Express, morgan (a logger), body-parser, path, and others.
   b. An Express app object is configured to listen for HTTP requests.
   c. The jade view engine is defined.
   d. Two routes are configured including `/` and `/users` and are associated with routing modules:

      ```
      app.use('/', indexRouter);
      app.use('/users', usersRouter);
      ```

6. Open the `routes/index.js` file in an editor. This file is referenced by `app.js` and used to handle the `/` route. Notice that the following code is included that handles rendering a Handlebars view named `index` and passing an object literal to the view that has a `title` property:

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
    res.render('index', { title: 'Express' });
});

module.exports = router;
```

7. Modify the code in `index.js` so that it includes a `name` property as shown next:

```
router.get('/', function(req, res, next) {
   res.render('index', { title: 'Express', name: 'Elon Musk' });
});
```

8. Save `index.js`.

9. Open the `views/layout.jade` file in an editor and take a moment to look through the jade code. This file contains the master layout for the website.

10. Open the `views/index.jade` file in an editor and change it to look like the following:

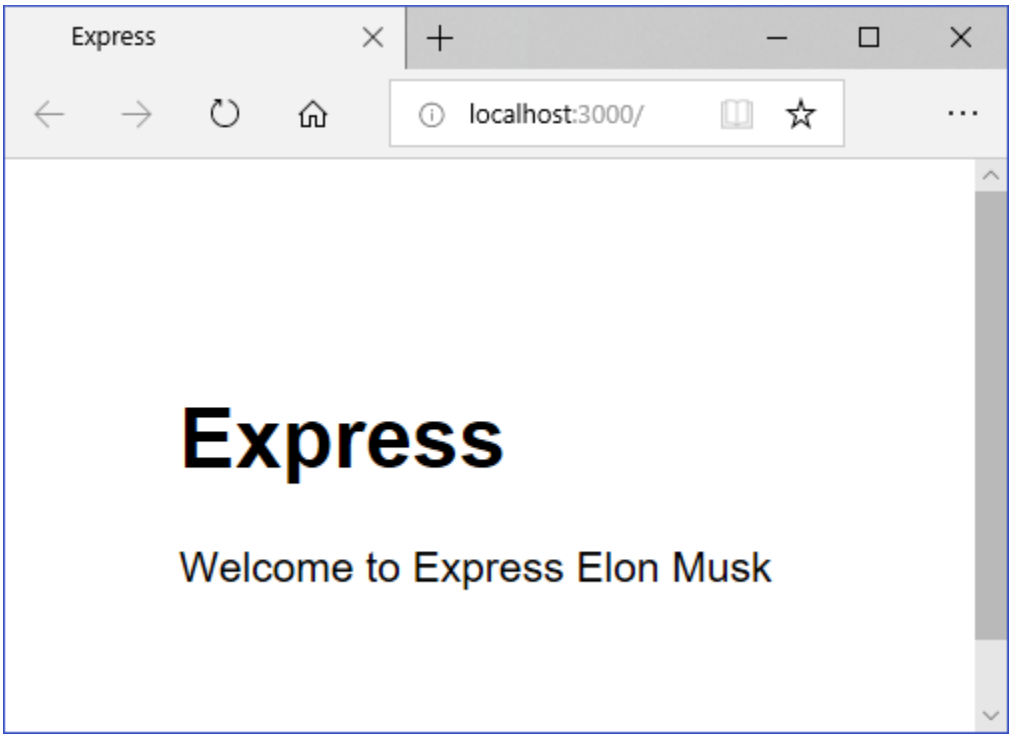
```
extends layout

block content
  h1= title
  p Welcome to #{title} #{name}
```

11. Save `views/index.jade`.

12. Run the following in the command prompt to start the Express server:
    `npm start`

13. Navigate to http://localhost:3000 in the browser and you should see the home page of the Express site displayed. Notice that it includes the custom name property value defined in `index.js`.



Exercise 7.3: Using the Express Module to Create a RESTful Service

In this exercise, you'll use Node Package Manager (npm) to create a new Express application, change the port settings of the service, make contact information stored in the JSON file accessible via your service, and run the website in the browser.

Note: If you run out of time typing, you can use a text file containing the code in the `Chapter7` folder: `SE_NGL_RESTful_Exercise 7.3.txt`.

Time: 45 minutes
Format: Programming exercise

1. Navigate to the `RESTfulServices\Chapter7` folder in the command prompt.

2. Execute the following command to generate an Express site named `SimpleService`:

   ```
   express SimpleService
   ```

3. Change the command's location to the new `SimpleService` folder that's now created.

4. Run the following command to install dependencies locally within the `SimpleService` folder:

   ```
   npm install
   ```

5. Open the `app.js` file at the root of SimpleService in an editor and take a moment to look through the code. Notice the following features should be familiar:
   a. Multiple modules are used including express, morgan (a logger), body-parser, path, and others.
   b. An Express app object is configured to listen for HTTP requests.
   c. The jade view engine is defined.
   d. Two routes are configured including `/` and `/users` and are associated with routing modules:

      ```
      app.use('/', indexRouter);
      app.use('/users', usersRouter);
      ```

6. Open the bin/www file in an editor. This file contains the port settings and createSever function. Add following code sequences for better feedback when staring the server.

Change port to 8081

```
// change port number on line 15 to 8081:
var port = normalizePort(process.env.PORT || '8081');
app.set('port', port);
```

Add this line for demo purposes only

Add url variable

```
// replace line 28 with the following:
var url = '127.0.0.1';
server.listen(port, url);
console.log('listening to :' + url + ' and Port: ' + port);
```

Add this line for output purposes only

a. Create a new folder named `modules` in the SimpleServer project folder.
b. In the modules folder add a file called `contacts.js`.
c. Add the following code to contacts.js:

Function to import JSON file

```
const fs = require('fs');
let read_json_file = () => {
    let file = './data/contact.json';
    return fs.readFileSync(file);
}

exports.list = function() {
   return JSON.parse(read_json_file());
};
```

Public function to query contacts by generic arguments and values

```
exports.query_by_arg = (arg, value) => {
    let json_result = JSON.parse(read_json_file());
    // all addresses are stored in a "result" object
    let result = json_result.result;
    console.log("query by arg: " + arg + " " + value);
    for (let i = 0; i < result.length; i++) {
        let contact = result[i];
        if (contact[arg] === value) {
            return contact;
        }
    }
    return null;
};
```

d. Create a new folder named `data` in the SimpleService project folder.
e. Copy the `contact.json` file from the `RESTfulServices\Chapter7\Resources` folder into the `SimpleService\data` folder.

7. Open the `routes/index.js` file in an editor. This file is referenced by `app.js` and used to handle the `/` route.

   Note: Replace the first two lines with the following code:

```
const createError = require('http-errors');
const express = require('express');
const router = express.Router();
```

   Note: adding these two lines makes your contacts.js module accessible, and provides utilities for URL resolution and parsing to your project.

```
const contacts = require('../modules/contacts');
const url = require('url');
```

8. Modify the code in `routes/index.js` to add following code.

   Note: add this code to add a `GET` routing for the "`/contacts`" string added to your base URL. This will allow to retrieve all contacts or specific contacts when parameters are provided. In this case only, filtering by last name is implemented.

   Add the code just above the `module.exports = router;` statement in the `index.js` file.

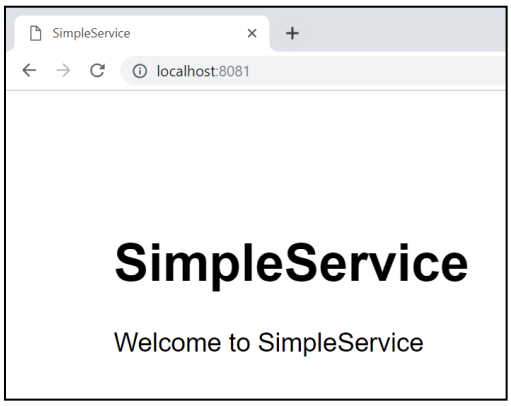
```
// can process any existing query parameters
//(e.g.:?firstname=John)
router.get('/contacts', (request, response, next) => {
  let get_params = url.parse(request.url, true).query;
  console.log('got into contacts');
  if (Object.keys(get_params).length == 0) {
    console.log('no params');
    response.setHeader('content-type', 'application/json');
    response.end(JSON.stringify(contacts.list()));
  } else {
    // get first parameter only
    let key = Object.keys(get_params)[0];
    console.log("First key is: " + key);
    let value = request.query[key];
    console.log('params ' + value);
    let result = contacts.query_by_arg(key, value);
    if (result) {
      response.setHeader('content-type', 'application/json');
      response.end(JSON.stringify(result));
    } else {
      next(createError(404));
    }
  }
});
// example for using path variable
router.get('/contact/:lastname', (request, response, next) => {
  const param = request.params.lastname;
  console.log('got into contact/:lastname ' + param);
  const result = contacts.query_by_arg(
    "lastname", param);
  if (result) {
    response.setHeader('content-type', 'application/json');
    response.end(JSON.stringify(result));
  } else {
    next(createError(404));
  }
});
```

9. Save `index.js`.

10. Run the following in the command prompt to start the Express server:

    ```
    npm start
    ```

11. Navigate to http://localhost:8081 in the browser and you should see the home page of the Express site displayed.



12. View the URL http://localhost:8081/contacts in your browser.
    a. Verify the contacts information is displayed.

13. View the URL http://localhost:8081/contacts?firstname=Joe in your browser.
    a. Verify that the contact information for Smith is displayed.

14. View the URL http://localhost:8081/contact/Smith in you browser
    a. Verify that the contact information for Smith is displayed.

Bonus Exercise (to be attempted if time permits)

15. Change the code in `routes/index.js` to allow service to process requests by some property other than last name. E.g.:
    a. `GET /contacts/lastname/Smith`
    b. `GET /contacts/firstname/Joe`

Exercise 7.4: Testing Express with Jasmine

In this exercise, you will use Jasmine to write tests for your Express application.

Time: 30 minutes
Format: Programming exercise

1. Create a new node.js project named `TestWithJasmine`.

2. Create the following folders in the project directory:
   a. `app`
   b. `spec`

3. Install dependencies by running the following command:

   ```
   npm install jasmine-node --save-dev
   ```

4. Install the request package by running the following command:

   ```
   npm install request --save
   ```

5. Install Express by running the following command:

   ```
   npm install express --save
   ```

6. Add test instructions to the `package.json` file

```
{
    "name": "jasminetest",
    "version": "0.0.0",
    "private": true,
    "scripts": {
      "start": "node ./bin/www",
      "test": "./node_modules/.bin/jasmine-node --captureExceptions spec"
    },
    "dependencies": {
      "cookie-parser": "~1.4.4",
      "debug": "~2.6.9",
      …
```

7. Create a Jasmine test file named `firstnodetest.spec.js` in the `spec` folder.

```
let request = require("request");

const base_url = "http://localhost:3000/";

console.log("Starting test");

describe("First Node Test Server", () => {
     describe("GET /", () => {
          /* it("returns status code 200", (done) => {
               request.get(base_url, (error, response, body) => {
                    expect(response.statusCode).toBe(200);
                    done();
               });
          }); */
          it("returns Hello World", (done) => {
               request.get(base_url, (error, response, body) => {
                    expect(body).toBe("Hello World");
                    done();
               });
          });
     });
});
```

8. Add another test for the status code of 200.

9. Open a command window and `cd` to the project folder.

10. Run the test by running the following command:

    ```
    npm test
    ```

11. The test should fail (this is a good thing).

12. Create a file called `firstnodetest.js` in the `app` folder.

    ```
    var express= require('express');
    var app = express();
    app.get("/", (req, res) => {
        res.send("Hello World");
    });

    app.listen(3000, () => {
        console.log("Server listening to port 3000");
    });
    ```

13. Start the server by running the following command in the console:

    ```
    node app/firstnodetest.js
    ```

14. Verify your application is running properly by opening http://localhost:3000 in your favorite browser.
    a. You should see "Hello World" in the browser.

15. Rerun the test (open new terminal window first).
    a. Now the test should pass.
    b. If not, you have some debugging work to do!

Chapter 8: Testing Node with Jasmine

Exercise 8.1: Testing with Jasmine and Node

In this exercise, you'll produce various tests for your SimpleServer RESTful service.

Time: 45 minutes
Format: Programming exercise

1. Navigate to the `RESTfulServices\Chapter8\QA_SimpleServiceWithTests` folder.
2. Run `npm install` to load all the necessary `node_modules`.
3. Start service with `npm start`.
4. Open a separate terminal to run tests later.
5. Investigate the `spec` folder inside this directory.
   a. Explore the three test files:
      i. `contacts_unit_spec.js`
         1. `tests service implementation as basic unit test: some tests need to be completely added, others with TODOs`
      ii. `contactByParam_spec.js`
         1. `test HTML interface: some tests need to be completely added, others with TODOs`
      iii. `contacts_spec.js`
6. Run `npm test` to see which test passes and fails.
7. Complete the TODOs and add necessary code to make all tests pass.
8. Try to understand all tests. What other tests could be a good idea?

This page intentionally left blank.

Chapter 9: Service Virtualization

Exercise 9.1: Using Service Virtualization

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

Time: 20 minutes
Format: Programming exercise

1. Open the `MessageService` Eclipse 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 and start Hoverfly by running the following command:
   a. `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.

Chapter 12: Testing with Cucumber.js

Exercise 12.1: Writing an Acceptance Test with Gherkin Syntax

Time: 20 minutes
Format: Programming exercise

Description:

1. Form groups of two (client and developer). Develop the story first, then describe the story in a feature file.

2. The problem is to purchase a new cell phone form Amazon.

3. Write an acceptance test using Gherkin syntax.

Feature: As an Amazon customer, I want to purchase a specific cell phone on their webpage, so that I can run the latest software on it.

Scenario: Purchasing cell phone from amazon.com.

Given Access to the internet
And valid website to see phone
And valid item
And item in stock

When purchase phone

Then decrease inventory
and charge customer
and shipping order created

Exercise 12.2: Test Your Calculation

Time: 20 minutes
Format: Programming exercise

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

2. `calculator-steps.js` is initially all commented out, and there's just one very simple scenario in `calculator.feature`.

3. Run npm test before making any changes.
   a. Output shows what steps are needed.

4. In `calculator-steps.js`, uncomment lines 1 – 18.

5. Run npm install to install the required libraries.

6. Run npm test and note the simple scenario passes.

7. In `calculator-steps.js`, comment out the When/Then for the simple scenario (lines 8 – 14).

8. Uncomment the When/Then for the second scenario (lines 20 – 26).

9. In `calculator.feature`, uncomment the second scenario and the sample data (lines 11 – 26).

10. Save both files.

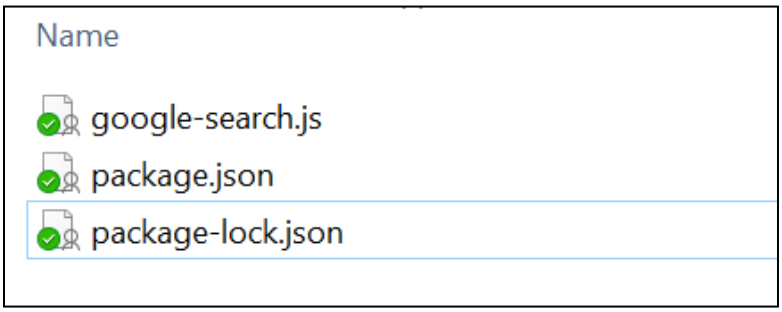
11. Run npm test and note that 10 scenarios now pass.

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.

- Go through the steps as laid out in the course material
- Find the following files ready for you in the `Exercise` folder `RESTfulServices\Chapter12\QA_GoogleSearchFeature`



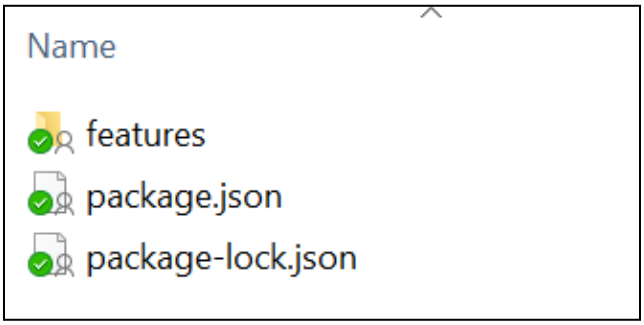
- Don't forget to install all modules with: `npm install`
- To fix potential issues run: `npm audit fix`
- Then Run test with: `npm test`

Exercise 12.4: Let's Test SimpleService BDD Style

Time: 30 minutes
Format: Programming exercise

Description:
Build a Cucumber/Gherkin-based test to test SimpleService server application.

- Go through the steps as laid out in the course material
- Add additional positive and negative tests for your service
- Before testing, start the `SimpleService` from the `QA_SimpleServiceWithTests` project in Chapter 8.
- Find the following files ready for you in the folder `RESTfulServices\Chapter12\QA_CucumberTestsForSimpleService`



- Don't forget to install all modules with: `npm install`
- To fix potential issues run: `npm audit fix`
- Then Run test with: `npm test`

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 actually 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.

Optional Exercise 14.3: Using JavaScript Promises

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

Time: 30 minutes
Format: Individual programming exercise

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

For the first section, you will replace usages of the deprecated `request` package with the `request-promise-native` package.

In the Bonus section, you will complete a JavaScript function that sends an asynchronous HTTP GET request and returns a Promise.

Time: 30 minutes
Format: Individual programming exercise

1. The Node.js `request` package is a popular HTTP client library. Visit the project's home page at https://www.npmjs.com/package/request.

   Note that the developers of `request` have deprecated it. They suggest that developers migrate to a more modern, Promise-based HTTP library.

2. The `request-promise-native` package is a replacement for `request`. Visit the project's home page at https://www.npmjs.com/package/request-promise-native.

3. Follow the installation instructions on the `request-promise-native` home page to install the package in your RVC.

   ```
   npm install request
   npm install request-promise-native
   ```

4. Visit https://www.npmjs.com/package/request-promise and search for "Cheat Sheet". You will see examples of sending HTTP GET requests to a JSON REST API.

5. Now that you have some background information, you will modify a script to use the `request-promise-native` package. The script communicates with a REST server that maintains a simple to-do list. Open a command prompt (or a new terminal window) and execute the following commands to start the REST server:

```
cd RESTfulServices\Chapter14\promises
npm install
npm install express
npm install cors
node todo_server.js
```

6. Verify the server is running by opening a web browser and navigating to http://localhost:9876/todos/1. You should see a to-do item in JSON format.

7. Next, you'll run a script that updates a to-do item. Open a second command prompt and execute the following commands:

```
cd RESTfulServices\Chapter14\promises
node send_request_for_json.js
```

8. Note the output from `send_request_for_json.js`.

9. Currently, `send_request_for_json.js` uses the deprecated `request` package. Now, you will modify `send_request_for_json.js` to use `request-promise-native`.
   a. Open the folder `RESTfulServices\Chapter14\promises` in VS Code.
   b. Edit `send_request_for_json.js`.
   c. Follow the instructions in the TODO comments.

10. Save your changes and execute `send_request_for_json.js`. Verify that the output is the same as before. If not, modify your code and run the script again.

Bonus Exercise (to be attempted if time permits)

11. View `RESTfulServices\Chapter14\promises\promises.html` in a local web server.
    a. Right-click the `promises.html` file in the Explorer window.
    b. Choose **Open with Live Server**.
    c. The `promises.html` file will be displayed in your default browser hosted on `localhost (127.0.0.1:5500)`.

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

13. Complete the code in the `promises.js` file.

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

15. Verify that the code you added works correctly.

Exercise 14.4: 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 14.5: 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 14.6: 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.

Appendix A: Building RESTful Services with JAX-RS

All of the exercises in this section use JAX-RS without Spring Boot. The projects are located 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 Advanced Rest Client (ARC) or Postman 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 Advanced Rest Client or Postman 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 Advanced Rest Client or Postman.

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 ARC or Postman.

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 the Advanced Rest Client or Postman 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 ARC.
   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 ARC or Postman.