

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Introduction

Course Description

How is this course valuable to a Full Stack Engineer (FSE)?

- Web Services provide a standard way to make functionality available to client-side developers, independent of programming language and other implementation details
- Web Services are used in many applications at Fidelity
- RESTful web services provide a lightweight means of providing data in response to requests sent via HTTP
- This will be very useful in web applications where JavaScript will send a request to a RESTful service and will display the data that is returned from the service

Course Outline

- Chapter 1 Building RESTful Services
- Chapter 2 Designing RESTful Services
- Chapter 3 Testing RESTful Services
- Chapter 4 Securing RESTful Web Services
- Chapter 5 Cloud Design Patterns
- Chapter 6 Node.js
- Chapter 7 Node.js and Express
- Chapter 8 Testing Node with Jasmine
- Chapter 9 Service Virtualization

Course Outline

- Chapter 10 Apigee Edge
- Chapter 11 The FidZulu Mini Project
- Chapter 12 Testing with Cucumber.js
- Chapter 13 Server-Side JavaScript Programming
- Chapter 14 Functional and Reactive Programming in JavaScript
- Appendix A Building RESTful Services with JAX-RS

Course Objectives

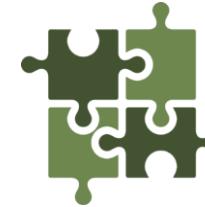
In this course, we will:

- Solve common programming problems by using design patterns
- Design and build RESTful web services
- Use JAX-RS and Spring Boot to create RESTful web services written in Java
- Use Node.js to execute RESTful services written in JavaScript
- Use some advanced JavaScript programming techniques

Key Deliverables



Course Notes



Project Work



There is a Skills Assessment for this course

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 1: Building RESTful Services

Chapter Overview

In this chapter, we will explore:

- Understanding what a RESTful web service is
- Building RESTful services with Spring Boot
- Returning HTTP status codes

Chapter Concepts

RESTful Web Services

What Is Spring Boot?

Building RESTful Services with Spring Boot

HTTP Response Codes

Chapter Summary

Web Application vs. Web Service

- Web applications are meant for use by human users
 - Visual representation of information, interactive
 - Multiple steps (often stateful)

The image shows a travel search interface with tabs for Flight, Hotel, Car, and Vacation. The Flight tab is selected. It displays two flight options:

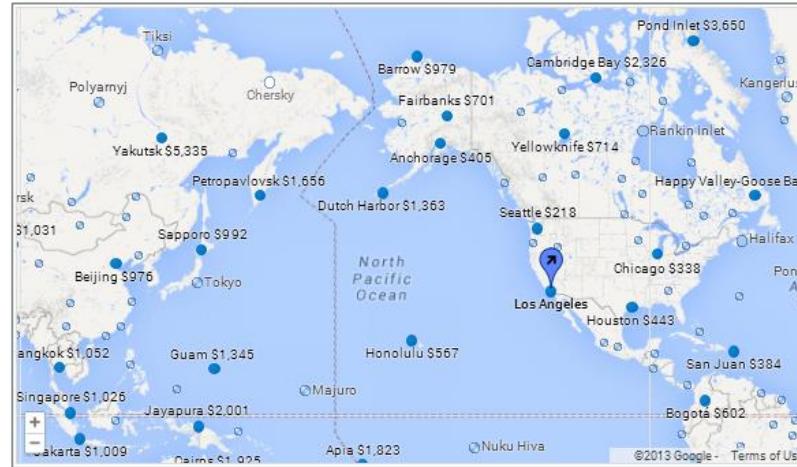
Flight Details	Flight 1	Flight 2
From:	Helsinki, Finland (HE)	
To:	Johannesburg ZA (JNB)	
Depart:	1:00 p.m. Sun., Nov. 16, 2014 Helsinki, Finland (HEL)	2:35 p.m. Sun., Nov. 16, 2014 Munich, Germany (MUC)
Arrive:		
Price:	\$1,264	
Select		

Below the flights, there are two sections for date selection:

- Search Specific Dates:** Depart Date: 11/16/2014, Time: Anytime; Return Date: 11/26/2014, Time: Anytime.
- My Dates are Flexible:** Depart Date: 11/16/2014, Time: Anytime; Return Date: 11/26/2014, Time: Anytime.

Web Application vs. Web Service (continued)

- Web services are meant for use by automated applications
 - Machine-parseable representation of information
 - Often stateless
- Web services are “behind the scenes” and are not readily apparent
 - How do you think this “mashup” at <https://www.google.com/flights> is created?



SOAP Web Services

- A SOAP-based web service
 - Advertises a WSDL interface
 - Contains everything needed to communicate with that service
 - All the request and response messages are in XML
 - Interoperable
 - Many different clients can communicate with the service
 - The data is wrapped inside of an XML SOAP message
 - Requests
 - Responses
 - Not the most convenient format for many clients
 - Ajax
 - JavaScript

A Simpler Web Service

■ REST-based web services

- Communicate with a simpler format than SOAP-based web services
 - Simple XML
 - JavaScript Object Notation (JSON)
- No Web Service Definition Language (WSDL) interface
 - Can provide a Web Application Definition Language (WADL) interface
- Parameters often passed as part of the URL

REST-Based Web Services

Problem

- Implementing a Service-Oriented Architecture requires:
 - Interoperable, loosely coupled ways of invoking services
 - Enterprise systems consist of many applications
 - Developed using many technologies
 - How to support communication between disparate systems?

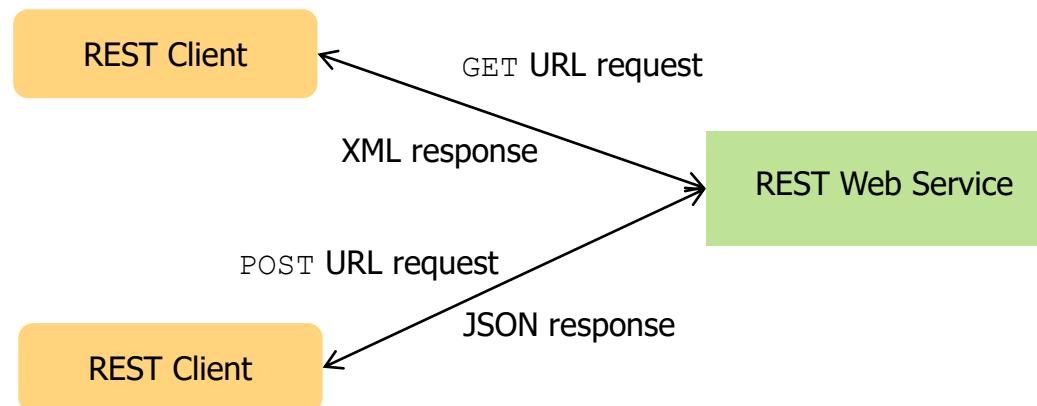
Solution

- Use HTTP as base protocol
 - Mature, standard protocol supported by all languages and operating systems
 - HTTP traffic is allowed through most firewalls
- REST-based Web Services are a form of SOA that:
 - Use HTTP as base protocol
 - Use HTTP verbs to obtain and manipulate resources
 - GET, POST, PUT, DELETE
 - Exchange simple messages
 - In XML, JSON, etc.

REST-Based Web Services (continued)

■ REST communication

- Based on HTTP requests
- Response can be in various formats
 - XML
 - JSON





HANDS-ON
EXERCISE

30 min

Exercise 1.1: Exploring the Time Service

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

RESTful Web Services

What Is Spring Boot?

Building RESTful Services with Spring Boot

HTTP Response Codes

Chapter Summary

Spring Boot

- Provides “an opinionated view”¹ of how to build a Spring application
- Minimizes Spring configuration for many applications
- Makes it easy to create a stand-alone Spring-based application
 - That “just runs”

1. <https://spring.io/projects/spring-boot>

Spring Boot Features

- Simplifies the creation of stand-alone Spring applications
 - Provides “opinionated” starter configuration options
- Can embed a web server such as Tomcat
 - No need to deploy a war file to an external web server
- Provides production features
 - Health checks
 - Application metrics
- No code generation required
- No XML configuration required

How Does Spring Boot Work?

- It is opinionated
 - Makes reasonable default configuration settings
 - Example: a Spring Boot web application embeds the Tomcat web server
- It is customizable
 - You can modify the Maven pom file
 - Override the “reasonable default” setting with your own setting value

Starters

- Spring Boot Starter
 - A set of dependencies for a particular type of application
- Some popular starters
 - spring-boot-starter-web
 - Used to build RESTful web services
 - Uses Spring MVC and embedded Tomcat
 - spring-boot-starter-jersey
 - Used to build RESTful web services
 - Uses Apache Jersey (JAX-RS) instead of Spring MVC
 - spring-boot-starter-jdbc
 - Used for JDBC connection pooling
 - Uses Tomcat's JDBC connection pool implementation
- There are many more starters available:
 - <https://github.com/spring-projects/spring-boot/tree/master/spring-boot-project/spring-boot-starters>

Spring Initializr

- An even better approach to creating a Spring Boot project
 - <https://start.spring.io/>
- The website lets you choose options to “bootstrap your application”
 - Creates a Maven (or Gradle) Spring Boot project
 - Includes a Maven `pom.xml`
- Allows you to choose project dependencies
 - Developer tools
 - Web
 - Security
 - SQL/NO SQL
 - Testing
 - Cloud

AutoConfiguration

- Spring Boot can automatically configure your application
 - Based on the jar files in the application's classpath
 - And how the Spring managed beans are defined
- Spring Boot will examine the jar files in the classpath
 - Forms an opinion on how to configure some behavior
 - Example: if the H2 database jar file is in the classpath, and no other DataSource beans are defined, the application will be configured with an in-memory database
- Spring Boot will examine the Spring managed bean definitions
 - Example: if a JPA bean is annotated with `@Entity`, the application will be configured to use JPA without the need for a `persistence.xml` file

JPA: Java Persistence API

The All-in-One Jar File

- Spring Boot aims to create an application that “just runs”
- The application is packaged into a single, executable jar file
 - The “uber” jar file
 - All of the application dependencies are included in this jar file
- The application is launched with a command like the following:
 - `java -jar PATH_TO_EXECUTABLE/HelloWorld.jar`

Nested Jar Files

- The all-in-one jar file contains other “nested” jar files
- Java does not provide a standard way to load nested jar files!
- Spring Boot defines a special jar file layout¹
 - Library jar files are contained in the all-in-one executable jar file
- Spring Boot uses a special class loader at runtime
 - Loads the classes contained in the nested jar files

1. <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar>

Chapter Concepts

RESTful Web Services

What Is Spring Boot?

Building RESTful Services with Spring Boot

HTTP Response Codes

Chapter Summary

Configuration

- Here is where the Spring Boot starters help us get jump started
- A typical RESTful web service project has many dependencies
 - Spring MVC or JAX-RS
 - Tomcat
 - Jackson
 - Etc.
- We can use the `spring-boot-starter-web` to simplify this

Maven Dependencies

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

The RestController

- The RESTful service is a Java class
 - Annotated with `@RestController`
- The web service methods are annotated
 - With `@RequestMapping`
- These are Spring annotations
 - Not JAX-RS annotations

The RESTful Service GET Request

```
@RequestMapping(value = "/widgets", method=RequestMethod.GET)
public ResponseEntity<List<Widget>> queryAllWidgets() {
    List<Widget> widgets = null;

    try {
        widgets = dao.getAllWidgets();
    } catch (RuntimeException e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body(widgets);
    }
    if(widgets == null) {
        return ResponseEntity.noContent().build();
    }
    return ResponseEntity.ok(widgets);
}
```

The RESTful Service POST Request

```
@RequestMapping(value = "/widgets", method = RequestMethod.POST,
    produces = { "application/json" }, consumes = { "application/json" })
public DatabaseRequestResult insertWidget(@RequestBody Widget w) {
    int count = 0;
    try {
        count = dao.insertWidget(w);
    } catch (Exception e) {
        throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR,
                "Error communicating with the warehouse database", e);
    }

    if (count == 0) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST);
    }

    return new DatabaseRequestResult(count, "Insert");
}
```

The Data Model

- The Data Model is a POJO
 - Not a Spring managed bean
- An instance of this class will be returned by the service
 - Converted to JSON format

The Data Model (continued)

```
public class Product {  
    private String description;  
    private Integer id;  
    private BigDecimal price;  
  
    // constructors  
  
    // getters & setters  
}
```

```
public class Widget extends Product {  
    private int gears;  
    private int sprockets;  
  
    // constructors  
  
    // getters & setters  
}
```

The Application

- The application could be packaged as a war file
 - And deployed to a web server
- A simpler approach uses a stand-alone application
- Everything is packaged into an executable jar file
 - All the Java source
 - All the resource files
 - All the libraries the application depends on
- The application class uses the good old `main()` method
 - Calls the `run()` method of the `SpringApplication` class

The Application (continued)

```
@SpringBootApplication(scanBasePackages = { "com.fidelity.configuration" })  
public class Application {  
    public static void main(String[] args) {  
        SpringApplication.run(Application.class, args);  
    }  
}
```

The `@SpringBootApplication` Annotation

- `@SpringBootApplication` is a convenience annotation that adds all of the following:
 - `@Configuration`
 - Tags the class as a source of bean definitions for the application context
 - `@EnableAutoConfiguration`
 - Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings
- `@EnableWebMvc`
 - Spring Boot adds it automatically when it sees `spring-webmvc` on the classpath
 - This flags the application as a web application and activates key behaviors such as setting up a `DispatcherServlet`
- `@ComponentScan`
 - Tells Spring to look for other components, configurations, and services in the same package as the application
 - Allowing it to find the controllers

Running the Application

- You can run the application from the command line:

```
java -jar target/warehouse-rest-service-0.1.0.jar
```

- Or you can run it using Maven:

```
mvnw spring-boot:run
```

- Logging output will be displayed
- The service should start fairly quickly

Communicating with the Service

- Once the application has started:
 - The service will be listening for requests
- Use a browser to send the following request to the service:

<http://localhost:8080/warehouse/widgets>

The port can be set in
application.properties:
server.port=8080

- The response should be the following:

```
[ {"description": "Low Impact Widget", "id": 1, "price": 13, "gears": 2, "sprockets": 3}, ... ]
```

- Send a request with a request parameter:

<http://localhost:8080/warehouse/widgets/1>

- The response this time should be:

```
{"description": "Low Impact Widget", "id": 1, "price": 13, "gears": 2, "sprockets": 3}
```

Handling Path Parameters

- Can inject variables in the URL to a method:

```
@RequestMapping(value = "/widgets/{id}", method = RequestMethod.GET,  
    produces = MediaType.APPLICATION_JSON_VALUE)  
public Widget queryForWidget(@PathVariable int id) {  
    ...}
```

- Example of Path handled by the above method:

```
http://localhost:8080/warehouse/widgets/42
```

Query Parameters

- Query parameters can also be injected:

```
@RequestMapping(value = "/widgets", method = RequestMethod.GET)
public ResponseEntity<List<Widget>>
    queryAllWidgets(@RequestParam (value = "sortBy", required = false String sortColumn)
{
    ...
}
```

- Example of URL handled by the above method:

```
http://localhost:8080/warehouse/widgets?sortBy=id
```

Allowed Parameter Types

- Path and Query parameters can be used on:
 - String
 - All simple types (int, long, Date, ...)
- Other types can be handled by the use of a custom type converter

Raising Error Conditions

- Can raise HTTP error conditions by throwing a ResponseStatusException

```
@RequestMapping(value = "/widgets/{id}", method = RequestMethod.GET,
                 produces = MediaType.APPLICATION_JSON_VALUE)
public Widget queryForWidget(@PathVariable int id) {
    Widget widget= null;
    try {
        widget = dao.getWidget(id);
    } catch (RuntimeException e) {
        throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Backend issue",e);
    }
    if(widget == null) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Provide correct id");
    }
    return widget;
}
```

XML and JSON

- JSON is the standard format used with REST services
- Clients are often Dynamic Web Applications using JavaScript
- Jax-RS handles both JSON and XML data
 - Will marshal data to/from Java classes based on type of data
- If a client requires data from a service in a particular format, they should set the HTTP header on the request
 - Accept: application/json or Accept:application/xml
 - Service will automatically send data in the correct format
 - With no change to code
- For a service to accept data, client should set HTTP header indicating type of data being sent
 - Content-type: application/json or Content-type:application/xml

Setting the Accept Header in ARC

The screenshot shows the ARC application window. At the top, the title bar reads "Request". Below it, the "Method" is set to "GET" and the "Request URL" is "http://localhost:8082/warehouse/widgets". A "SEND" button is to the right. Under the "Parameters" section, the "Headers" tab is selected. It displays a table with one row: "Header name" is "Accept" and "Header value" is "application/json". There are "ADD HEADER" and "DELETE" buttons. The "Variables" tab is also visible. At the bottom, there's a toolbar with icons for file operations and a status message "Selected environment: Default".

Producing JSON or XML Example

- Consider the following service:

- Will produce XML or JSON based on HTTP header and data format

```
@RequestMapping(value = "/widgets/{id}", method = RequestMethod.GET,
                 produces = { "application/json", "application/xml" })
public Widget queryForWidget(@PathVariable int id) {
    Widget widget= null;
    try {
        widget = dao.getWidget(id);
    } catch (RuntimeException e) {
        throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Backend issue", e);
    }
    if(widget == null) {
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Provide correct id");
    }
    return widget;
}
```

Producing JSON or XML Example (continued)

- Following method will accept JSON or XML based on header set:

```
@RequestMapping(value="/widget", method=RequestMethod.POST,  
                 consumes = MediaType.APPLICATION_JSON_VALUE)  
public Widget addWidget (@RequestBody Widget widget) {  
    dao.insertWidget(widget);  
    return widget;  
}
```

Widget parameter will be created from
JSON data in the body of the message

```
@XmlRootElement  
public class Exhibit {  
    private String name;  
    private String artist;  
  
    // get/set methods  
}
```



HANDS-ON
EXERCISE

30 min

Exercise 1.2: Creating a RESTful Service

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

RESTful Web Services

What Is Spring Boot?

Building RESTful Services with Spring Boot

HTTP Response Codes

Chapter Summary

HTTP Status Codes

- The HTTP protocol defines meaningful status codes
 - Which can be returned from a RESTful service
- Using status codes can help service consumers
 - Determine how to understand the service response
 - Especially when errors occur
- What is status code 418?

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

HTTP Status Codes (continued)

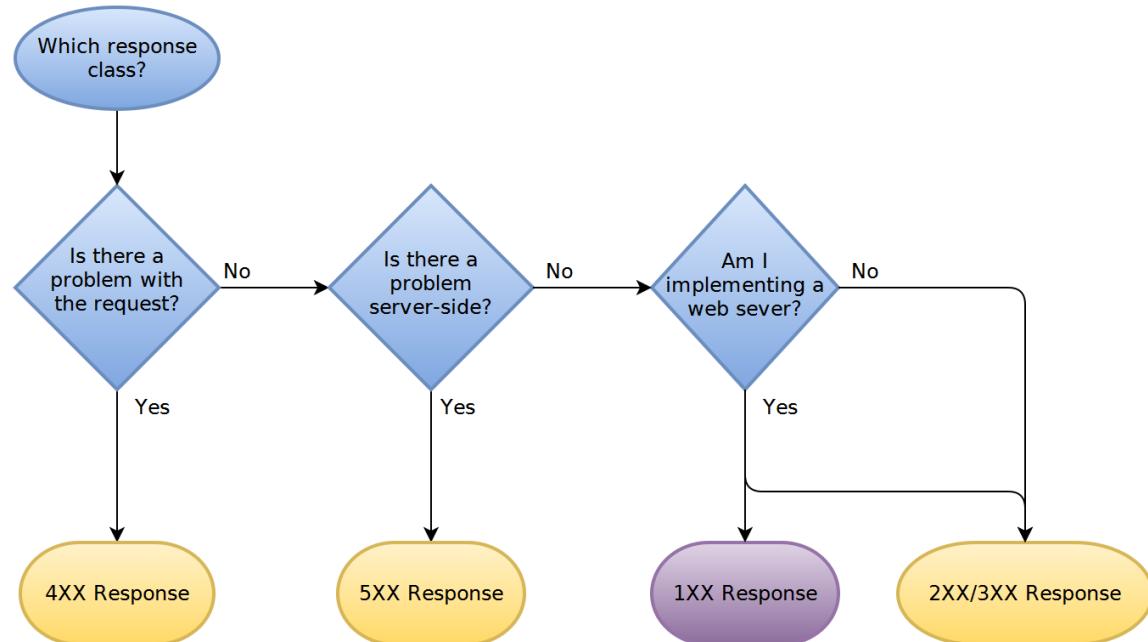
- 200 OK Response to a successful request
- 201 Created Response to POST that results in a resource creation
- 204 No Content Response to a successful request that does not return a body
- 400 Bad Request The request was malformed
- 401 Unauthorized Either invalid or missing authentication details in request
- 403 Forbidden User does not have access to the requested resource
- 404 Not Found We've all been here before
- 405 Method Not Allowed The HTTP method is not allowed for this user
- 410 Gone The resource is no longer available
- 418 ?

Why Should I Use a Status Code?

- They communicate to the RESTful client
 - When an exceptional event occurs
 - When some special behavior is required
- Many status codes represent situations that are worth handling with a special response
- Many widely used APIs are using them
 - A convention is being created
 - Following that convention makes it easier for users of your RESTful service
 - <https://gist.github.com/vkostyukov/32c84c0c01789425c29a>

What Status Should I Return?

- The following flowcharts answer this question
 - From <https://www.codetinkerer.com/2015/12/04/choosing-an-http-status-code.html>
- The flowcharts for each category of response are too big to fit on these slides
- Visit the above URL to see them





HANDS-ON
EXERCISE

Exercise 1.3: Which Status Code to Use?

20 min

- Follow the instructions in your Exercise Manual for this exercise

How to Return a Status Code

- Two main methods for returning an HTTP status code response
- Return a ResponseEntity object
 - That wraps your Java return object
 - And adds the status code
- Throw a ResponseStatusException
 - Can set the HTTP status code
 - And provide an error message

Using the ResponseEntity Object

- The service method returns a ResponseEntity object
 - Wraps the Java object to be returned
 - And sets the HTTP status code

```
@RequestMapping(value = "/widgets", method=RequestMethod.GET)
public ResponseEntity<List<Widget>> queryAllWidgets() {
    List<Widget> widgets = null;

    try {
        widgets = dao.getAllWidgets();
    } catch (RuntimeException e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body(widgets);
    }
    if(widgets == null) {
        return ResponseEntity.noContent().build();
    }
    return ResponseEntity.ok(widgets);
}
```

Throwing a ResponseStatusException

- If the ResponseStatusException is thrown:
 - Returns an error message with the HTTP status code

```
@RequestMapping(value = "/widgets/{id}", method = RequestMethod.GET,
                 produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<Widget> queryForWidget(@PathVariable int id) {
    Widget widget= null;
    try {
        widget = dao.getWidget(id);
    } catch (RuntimeException e) {
        throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR,
                                         "Backend issue",e);
    }
    if(widget == null) {
        return ResponseEntity.noContent().build();
    }
    return widget;
}
```

Subclasses of ResponseStatusException

- There are several subclasses of ResponseStatusException:
 - ServerErrorException
 - For an HttpStatus.INTERNAL_SERVER_ERROR
 - ServerWebInputException
 - For errors that fit response status 400 (bad request)



HANDS-ON
EXERCISE

30 min

Exercise 1.4: Returning a Status Code

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

RESTful Web Services

What Is Spring Boot?

Building RESTful Services with Spring Boot

HTTP Response Codes

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Understanding what a RESTful web service is
- Building RESTful services with Spring Boot
- Returning HTTP status codes

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 2: Designing RESTful Services

Chapter Overview

In this chapter, we will explore:

- Designing an effective RESTful API
 - RESTful API guidelines
- Deploying a Spring Boot-based RESTful web service
- The Twelve-Factor App
- How Spring Boot is used at Fidelity
- How to debug a RESTful web service

Chapter Concepts

How Fidelity Uses Spring Boot

Designing an Effective RESTful API

Deploying a Spring Boot-Based RESTful Web Service

The Twelve-Factor App

Debugging a RESTful Web Service

Chapter Summary

Using Spring Boot at Fidelity

- Fidelity developers use Spring Boot to build RESTful web services
- Documentation for using Spring Boot is provided¹
- Sample applications are available

1. <https://itec-confluence.fmr.com/display/AP119867/Springboot-Reference+Application>

Prerequisites

1. The application must be compatible with JDK v1.8
2. Spring Boot version 1.5.4.RELEASE recommended
3. Spring version 4.3.9 recommended
4. The application complies with the Twelve-Factors App guidelines
5. Recommended build tool is Maven
6. Recommended IDE is Spring Tool Suite v3.9.0.RELEASE
7. Configuration credentials must be set up in Concourse Vault for CI/CD

Note: These versions may be outdated. Check with your business unit or tech lead for current versions.

Spring Boot Reference Applications

- There are three Spring Boot reference applications
 1. RESTful service that consumes another RESTful service
 2. RESTful service protected by OAuth2 which consumes another RESTful service
 3. RESTful service with database connectivity



Exercise 2.1: Research Spring Boot at Fidelity

30 min

- Follow the directions in your Exercise Manual for this exercise

Chapter Concepts

How Fidelity Uses Spring Boot

Designing an Effective RESTful API

Deploying a Spring Boot-Based RESTful Web Service

The Twelve-Factor App

Debugging a RESTful Web Service

Chapter Summary

RESTful API Guidelines

- RESTful services should be stateless
- Use RESTful URLs and actions
- Prefer returning JSON instead of XML
- Support versioning
- Use token-based authentication
- Include response headers that support caching
- Use HTTP Status codes effectively
- Consider using query parameters for filtering

These are discussed
on the following slides

RESTful Services Should Be Stateless

- A RESTful service API should be stateless
- Requests should not depend on cookies
 - Or sessions
- Services are much simpler
- Services are more efficient

User RESTful URLs

- RESTful principles were introduced in Roy Fielding's dissertation¹
- Separate your API into conceptual resources
 - Describe as nouns (not verbs)
- Use HTTP verbs to manipulate resources

GET /employees/42/email	Retrieve the email for employee 42
POST /employees/42	Create employee 42
PUT /employees/42	Replace the existing employee 42
PATCH /employees/42	Perform partial update of employee 42
DELETE /employees/42	Delete employee 42

1. Architectural Styles and the Design of Network-based Software Architectures, Roy Fielding
<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

Prefer JSON

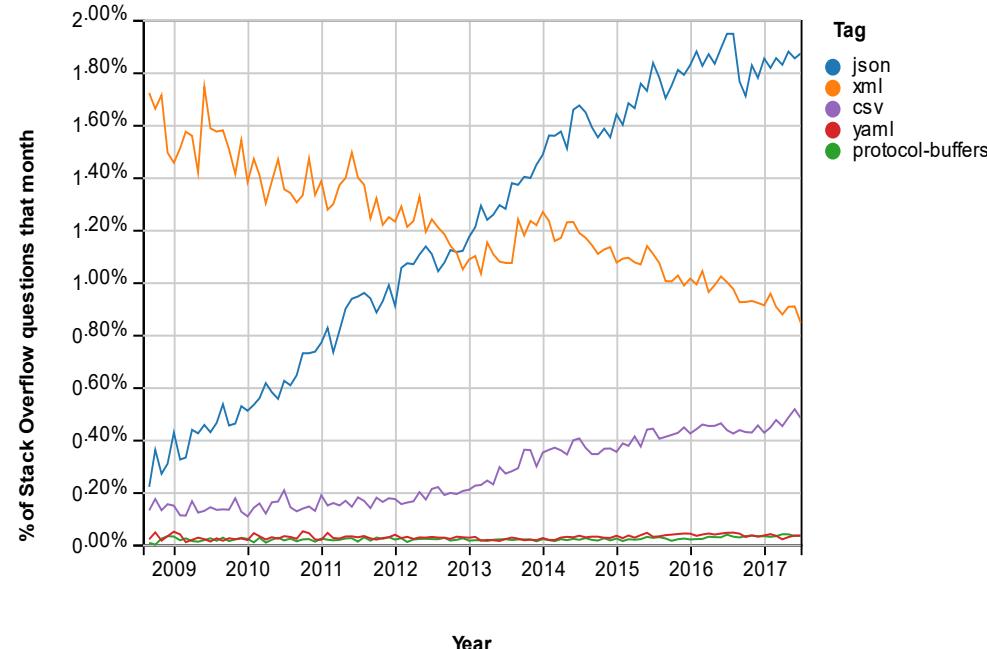
■ Problems with XML

- Verbose
- Difficult to parse
- Difficult to use in a JavaScript client

■ JSON is easy to use in JavaScript

- It stands for JavaScript Object Notation, right?

■ The chart on the right indicates the trends of XML and JSON



<https://twobithistory.org/2017/09/21/the-rise-and-rise-of-json.html>

Support Versioning

- Change is inevitable
 - So, we have to deal with it
- Plan ahead for versioning support
 - Will save much time and effort later
- Where to include the version information
 - HTTP header
 - Request URL
- Can provide support for old versions
 - While offering new ones

Token-Based Authentication

- Request authentication should be stateless
 - No dependence on sessions or cookies
- Each request should contain its authentication credentials
- More on this in the Security section

Support Caching

- HTTP provides built-in caching
- Use additional outbound response headers
 - ETag
 - Last-Modified

ETag

- The ETag HTTP header
 - Contains a hash or checksum of the requested resource
 - This value should change whenever the resource changes
- If a request for the resource contains an If-None-Match header with a matching ETag value:
 - Then return a 304 Not Modified status code
 - Instead of the requested resource

Last-Modified

- Last-Modified works similarly to ETag
- Returns a timestamp as the value of the Last-Modified response header
- This is validated against the If-Modified-Since request header

Use HTTP Status Codes

- Use those status codes
 - As discussed in the Status Codes section

Filtering Resources

- Good idea to keep the RESTful URLs as lean as possible
- Filtering, sorting, and searching can be implemented with query parameters
 - Instead of many different URLs
- Consider using a unique query parameter
 - For each field that implements filtering
- This could also be done for sorting and searching

GET /tickets?state=open

– Retrieves only the tickets in the open state

GET /tickets?sort=-priority

– Retrieves a list of tickets in descending order of priority



40 min

Exercise 2.2: Designing a RESTful Service API

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

How Fidelity Uses Spring Boot

Designing an Effective RESTful API

Deploying a Spring Boot-Based RESTful Web Service

The Twelve-Factor App

Debugging a RESTful Web Service

Chapter Summary

Deploying to CloudFoundry

- The Spring Boot “uber” jar file is easily deployed to the cloud
- CloudFoundry employs a “buildpack” approach
 - The buildpack wraps deployed code in what is needed to start the application
 - The Java buildpack provides support for Spring Boot applications
- Use the CloudFoundry `cf` command line tool
 - Deploy the Spring Boot application by using the `cf push` command
 - This will upload the uber jar file to CloudFoundry
- CloudFoundry uploads and deploys the application
 - Once that process completes successfully, the application is live!

Deploying to Amazon Web Services (AWS)

- Amazon Web Services provides multiple ways to install a Spring Boot application
 - Either as a war file or as an uber jar file
- AWS Elastic Beanstalk is the simplest option
 - Load balanced by default
 - Supports two options for a Java application
 - Tomcat platform
 - Java SE platform
- Tomcat platform
 - Supports web applications deployed as a war file
- Java SE platform
 - Supports Spring Boot uber jar file applications
 - Runs an nginx instance on port 80 to proxy the application
 - The application runs on port 5000

Deploying with Docker

- Docker has a simple Dockerfile file format
 - Used to specify the “layers” of a Docker image
- A Docker image consists of read-only layers
 - Each layer represents an instruction
 - Each layer is a delta of the changes from the previous layer
- Best practices for creating a Docker file are available online
- https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

A Simple Docker File

- The VOLUME “/tmp” is where a Spring Boot application creates working directories for Tomcat by default
- The DEPENDENCY parameter points to a directory where the Spring Boot uber jar has been unpacked
- The application’s main class is used as the entry point

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG DEPENDENCY=target/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java","-
cp","app:app/lib/*","hello.Application"]
```

Building a Docker Image with Maven

- The Maven pom.xml file should have the following content:
 - The dockerfile-maven-plugin added to the <plugins> section
 - The maven-dependency-plugin is configured to unpack the uber jar
- The Docker image can then be built by running the following command:

```
mvn install dockerfile:build
```

Running a Docker Image

- The Docker image can then be run by running the following command:
 - Where {imageName} is the name of the Docker image

```
docker run -p 8080:8080 {imageName}
```

- The application is then available on `http://localhost:8080`

Chapter Concepts

Designing an Effective RESTful API

Deploying a Spring Boot-Based RESTful Web Service

The Twelve-Factor App

How Fidelity Uses Spring Boot

Debugging a RESTful Web Service

Chapter Summary

The Twelve-Factor App¹

- A methodology for building software-as-a-service applications
- Uses declarative formats for setup automation
 - Makes it easier for new developers joining the project
- Has a clean contract with the underlying operating system
 - Supports ease of portability between environments
- Suitable for deployment on modern cloud platforms
 - Minimizes the need for servers and administrators
- Minimizes divergence between development and production systems
 - Enables continuous deployment
- Can scale up easily
 - Without significant changes to tooling, architecture, or development

1. <https://12factor.net/>

The Twelve Factors

- I. Codebase
 - One codebase tracked in version control
 - Many deploys
- II. Dependencies
 - Explicitly declared and isolated
- III. Configuration
 - Store configuration information in the environment
- IV. Backing services
 - Treat backing services as attached resources
- V. Build, release, run
 - Strictly separate the build and run stages



The Twelve Factors (continued)

VI. Processes

- Execute the application as a stateless process

VII. Port binding

- Export services via port binding

VIII. Concurrency

- Scale out the application via the process model

IX. Disposability

- Fast startup
- Graceful shutdown

X. Development/Production Parity

- Keep development, staging, and production as similar as possible

The Twelve Factors (continued)

XI. Logs

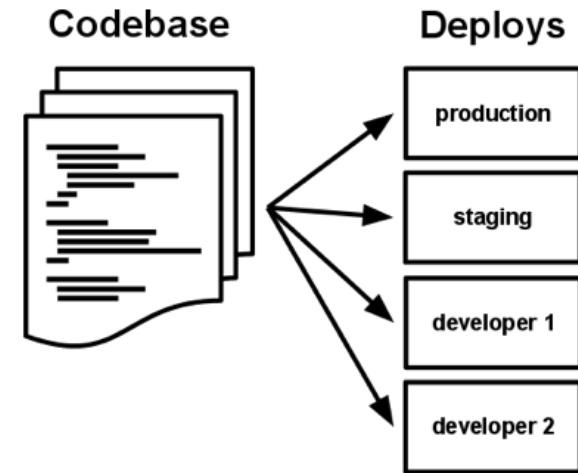
- Treat logs as event streams

XII. Administrative processes

- Run administrative/management tasks as one-off processes

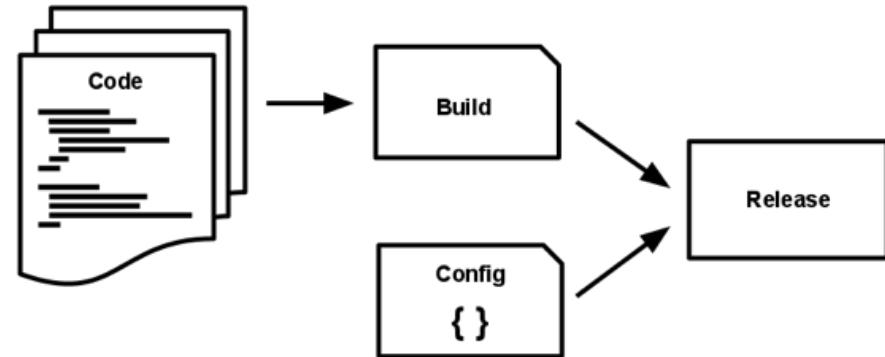
One Codebase

- One codebase, many deploys
- Only one codebase per application
- A deploy of the application is a running instance of the application
 - Production
 - Staging
 - Developers
- Codebase is tracked in a version control system



Build, Release, Run

- A codebase is transformed into a deploy through three stages
 - Build stage
 - Converts codebase into an executable
 - Release stage
 - Combines the build with the deploy's current configuration
 - Ready for immediate execution
 - Run stage
 - Runs the application in an execution environment



Processes

- The application is executed as one or more processes
- Twelve-factor processes:
 - Stateless
 - Share nothing
 - Data to be persisted are stored in a backing service (i.e., database)
- Session state data
 - No “sticky sessions”
 - Should be stored in a datastore with time expiration



HANDS-ON
EXERCISE

Exercise 2.3: Research the Twelve-Factor App

30 min

- Follow the directions in your Exercise Manual for this exercise

Chapter Concepts

How Fidelity Uses Spring Boot

Designing an Effective RESTful API

Deploying a Spring Boot-Based RESTful Web Service

The Twelve-Factor App

Debugging a RESTful Web Service

Chapter Summary

Debugging RESTful Services

- Debugging is one of the most important tools/skills for writing successful software
- Debugging a Spring Boot-based RESTful service is a bit more difficult than debugging a plain Java application
- The steps are:
 - Run the Spring Boot application with debugging enabled
 - Launch a remote debugger
 - Attach the remote debugger to our running Spring Boot application
 - Set breakpoints in the RESTful service method
 - Make an HTTP request that is handled by the RESTful service method
 - Step through the service method in Eclipse



HANDS-ON
EXERCISE

30 min

Exercise 2.4: Debugging a RESTful Service

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

How Fidelity Uses Spring Boot

Designing an Effective RESTful API

Deploying a Spring Boot-Based RESTful Web Service

The Twelve-Factor App

Debugging a RESTful Web Service

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Designing an effective RESTful API
 - RESTful API guidelines
- Deploying a Spring Boot-based RESTful web service
- The Twelve-Factor App
- How Spring Boot is used at Fidelity
- How to debug a RESTful web service

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 3: Testing RESTful Services

Chapter Overview

In this chapter, we will explore:

- The responsibilities of a RESTful Service Controller
- Performing unit testing of POJOs in the back end
- Testing the RESTful Service controller in the web (HTTP) environment
- Performing end-to-end testing of a RESTful Service using `TestRestTemplate`

Chapter Concepts

RESTful Service Controller Responsibilities

Testing RESTful Services

Testing the Back End

Testing the Web Layer

End-to-End Testing

Chapter Summary

A RESTful Web Service Front End

- A RESTful web service is a front end to some existing resource
 - Provides a simple means of accessing and managing the existing resource
 - Typically this is an HTTP interface
 - Text based, accessible to any type of client
- The RESTful service code is generally fairly simple
 - Management of the existing resource is provided by other classes
 - Often a Data Access Object
 - Or a Business Service

RESTful Service

```
@RestController
@RequestMapping("/warehouse")
public class WarehouseService {
    @Autowired
    private WarehouseBusinessService service;

    @PostMapping(value="/widgets", produces=MediaType.APPLICATION_JSON_VALUE,
                 consumes=MediaType.APPLICATION_JSON_VALUE)
    public DatabaseRequestResult insertWidget(@RequestBody Widget w) {
        int count = 0;
        try {
            count = dao.insertWidget(w);
        }
        catch (Exception e) {
            throw new ServerErrorException(DB_ERROR_MSG, e);
        }
        if (count == 0) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST);
        }
        return new DatabaseRequestResult(count);
    }
    ...
}
```

Processing a Request to a RESTful Service

#	Responsibility	Description
1	Listen for HTTP requests	The service should respond to the URLs specified in the web methods
2	Deserialize the inputs	Parse the incoming request and create Java objects from path and request parameters and the request body
3	Validate the input	Validate the incoming input in the request
4	Call the back end	Call on the back end to process the input
5	Serialize the output	Serialize the output from the back end into an HTTP response
6	Translate exceptions	Translate any exceptions into a meaningful error message and HTTP status

Unit or Integration Test of a RESTful Service

- The only step that would be possible in a unit test of a RESTful service would be Step 4, “Call the back end”
 - The other steps are provided by Spring Boot
- A simple unit test will not cover the HTTP layer at all
 - That is the context in which a RESTful service operates
- An integration test with Spring can provide the HTTP layer
 - Spring provides all the managed beans that the web service needs
 - Spring provides all the framework beans for all the other steps
- Spring Boot provides the @WebMvcTest annotation
 - This creates and uses a Spring application context that contains only the managed beans that are necessary for testing the web service controller

Chapter Concepts

RESTful Service Controller Responsibilities

Testing RESTful Services

Testing the Back End

Testing the Web Layer

End-to-End Testing

Chapter Summary

Testing a RESTful Service Project

- There are several techniques used to test the components of a RESTful service project
 1. Testing the back-end functionality with unit and integration tests
 - Data access objects
 - Business services
 2. Testing the RESTful service controllers running in the web layer
 - Provide complete testing coverage of the HTTP request handling
 3. End-to-end testing
 - Perform end-to-end testing of the RESTful service

Testing the Back End

1. True unit testing of the POJOs in the project
 - Spring is not used for configuration
 - This allows complete testing coverage of the business functionality
 - Dependencies can be mocked
2. Integration testing of POJOs
 - Verify the Spring configuration is correct
 - Verify the components (POJOs) work together correctly

Testing the RESTful Service Controllers

- Spring provides support for running tests on web services with `@MockMvcTest`
 - Spring Boot instantiates only the beans necessary for the web layer
 - An individual controller can be specified as shown below

```
@WebMvcTest(WarehouseService.class)
```

End-to-End Testing of the RESTful Service

- Spring provides `@TestRestTemplate` for testing actual HTTP calls to the web service
 - Tomcat runs in a separate thread
 - The full Spring configuration is utilized

Chapter Concepts

RESTful Service Controller Responsibilities

Testing RESTful Services

Testing the Back End

Testing the Web Layer

End-to-End Testing

Chapter Summary

Dao Test

- We often start by testing the integration tier first
- To verify that the DAO is actually working, we'll need to query the database directly
- We can use Spring's JdbcTestUtils class
 - We will use methods like `countRowsInTable()` and `deleteFromTables()`

```
@SpringBootTest
@Transactional
public class DaoTest {
    @Autowired
    private WarehouseDaoMyBatisImpl dao;
    private JdbcTemplate jdbcTemplate; // for executing SQL queries
    // We can't autowire a JdbcTemplate directly, but we can create an instance
    // of JdbcTemplate when a DataSource is available
    @Autowired
    void setDataSource(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
}
```

Dao Test (continued)

```
@Test
void testGetAllWidgets() {
    List<Widget> widgets = dao.getAllWidgets();
    assertThat(widgets, is(equalTo(allWidgets)));
}

@Test
void testDeleteWidget() {
    int id = 1;
    // verify that Widget 1 is in the database
    assertThat(1, is(equalTo(
        Jdbc TestUtils.countRowsInTableWhere(jdbcTemplate, "widgets", "id = " + id))));

    int rows = dao.deleteWidget(id);

    assertThat(rows, is(equalTo(1)));
    // verify that Widget 1 is no longer in the database
    assertThat(0, is(equalTo(
        Jdbc TestUtils.countRowsInTableWhere(jdbcTemplate, "widgets", "id = " + id))));

}
```

allWidgets is defined earlier to be the known collection of widgets

Mock Testing

- Mocks are often used to construct true unit tests of POJOs
 - No Spring configuration will be used (that would produce an integration test)
- The first step in testing a web service project:
 - Complete unit testing coverage of the functionality provided by POJOs
 - Business services
 - Data access objects
 - Mock the dependencies of a POJO
- The implementation of a web service is primarily calling on a POJO
 - It is not necessary to mock the dependencies of the web service
 - We will gain test coverage of the web service in our later tests

POJO Test

- POJO tests make sense for business objects with dependencies on DAOs, other services, etc.
- Quite often a business service will have a dependency on a DAO

```
@Service
public class WarehouseBusinessService {
    @Autowired
    private WarehouseDao dao;

    public List<Widget> findAllWidgets() {
        List<Widget> widgets;
        try {
            widgets = dao.getAllWidgets();
        } catch (Exception e) {
            String msg = "Error querying the Warehouse database.";
            throw new WarehouseDatabaseException(msg, e);
        }
        return widgets;
    }
}
```

Unit Test Mocking the Dependency

```
class WarehouseBusinessServiceTest {
    @Mock WarehouseDaoMyBatisImpl mockDao;
    @InjectMocks WarehouseBusinessService service;
    @BeforeEach
    void setUp() throws Exception {
        service = new WarehouseBusinessService();
        MockitoAnnotations.initMocks(this);
    }
    @Test
    void testFindAllWidgets() {
        // Initialize the list of Widgets that will be returned by the mock DAO
        List<Widget> widgets = Arrays.asList(
            new Widget(1, "Low Impact Widget", 12.99, 2, 3),
            new Widget(2, "High Impact Widget", 15.99, 4, 5));
        when(mockDao.getAllWidgets())
            .thenReturn(widgets);
        List<Widget> allWidgets = service.findAllWidgets();
        assertThat(allWidgets, equalTo(widgets));
    }
}
```

Integration Testing

- After unit tests are defined for the back end POJOs, integration tests can be written
 - Use Spring to manage and inject the dependencies
- For example, a business service has a dependency on a Dao
 - An integration test can verify the business service and dao work together successfully

Integration Test

```
@SpringBootTest
@Transactional
class WarehouseBusinessServiceIntegrationTest {
    @Autowired
    WarehouseBusinessService service;

    // Because the test database is tiny, we can check all products.
    // If the database was larger, we could just spot-check a few products.
    private static List<Widget> allWidgets = Arrays.asList(
        new Widget(1, "Low Impact Widget", 12.99, 2, 3),
        new Widget(2, "Medium Impact Widget", 42.99, 5, 5),
        new Widget(3, "High Impact Widget", 89.99, 10, 8)
    );

    @Test
    void testGetAllWidgets() {
        List<Widget> widgets = service.findAllWidgets();

        assertThat(widgets, is(equalTo(allWidgets)));
    }
}
```



Exercise 3.1: Testing Back-End POJOs

30 min

- Follow the directions in your Exercise Manual for this exercise

Chapter Concepts

RESTful Service Controller Responsibilities

Testing RESTful Services

Testing the Back End

Testing the Web Layer

End-to-End Testing

Chapter Summary

Testing the Web Layer

- It is not enough to verify the web service returns the correct HTTP status
- Using `@WebMvcTest` will narrow testing down to just the web layer
 - This, along with Spring Boot, provides everything needed to build web service controller tests
 - Be sure to define tests for all cases
 - We want to avoid ugly surprises at run time!
- The test assertions will be the same as in a complete application test
 - But Spring is only instantiating the web layer, not the entire context
- It is possible to only instantiate one controller in the application
 - `@WebMvcTest(controllers = WarehouseService.class)`
- The `MockMVC` object will perform the web requests

Testing with @WebMvcTest

```
@WebMvcTest/controllers=WarehouseService.class
public class WarehouseServiceWebMockTest {
    @Autowired
    private MockMvc mockMvc;
    @MockBean
    WarehouseBusinessService service;

    @Test
    public void testAddWidgetToWarehouse() throws Exception {
        Widget w = new Widget(42, "Test widget", 4.52, 20, 10);
        when(service.insertWidget(w)).thenReturn(1);
        ObjectMapper mapper = new ObjectMapper();
        String jsonString = mapper.writeValueAsString(w);

        mockMvc.perform(post("/warehouse/widgets")
                .contentType(MediaType.APPLICATION_JSON)
                .content(jsonString))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.rowCount").value(1));
    }
}
```

Testing Exception Handling

```
@Test  
public void testQueryForAllWidgets_ServiceThrowsException() throws Exception {  
    when(service.findAllWidgets()).thenThrow(new RuntimeException());  
  
    mockMvc.perform(get("/warehouse/widgets"))  
        .andDo(print())  
        .andExpect(status().is5xxServerError())  
        .andExpect(content().string(is(emptyOrNullString())));  
}
```



Exercise 3.2: Testing the Web Layer

30 min

- Follow the directions in your Exercise Manual for this exercise

Chapter Concepts

RESTful Service Controller Responsibilities

Testing RESTful Services

Testing the Back End

Testing the Web Layer

End-to-End Testing

Chapter Summary

End-to-End Testing with @TestRestTemplate

- We should write some tests that assert the actual behavior of the web service
 - Start the application and listen for a connection
 - Send a request to the web service
 - Assert the response
- TestRestTemplate
 - Supports standard HTTP methods, request headers, ...
 - Spring provides a `TestRestTemplate` automatically
 - Provided by `@SpringBootTest`
 - Just autowire it into your test class
- Use `WebEnvironment.RANDOM_PORT` to avoid port conflicts
 - The randomly selected port will automatically be used in the web request URL

Testing the Service with TestRestTemplate

```
@SpringBootTest(classes=WarehouseServiceApplication.class,  
                 webEnvironment=WebEnvironment.RANDOM_PORT)  
@Sql/scripts={"schema-dev.sql", "data-dev.sql"},  
             executionPhase=Sql.ExecutionPhase.BEFORE_TEST_METHOD)  
public class WarehouseServiceTestRestTemplateTest {  
    @Autowired  
    private TestRestTemplate restTemplate;  
  
    private JdbcTemplate jdbcTemplate; // for executing SQL queries  
  
    // We can't autowire a JdbcTemplate directly, but we can create an  
    // instance of JdbcTemplate when a DataSource is available  
    @Autowired  
    void setDataSource(DataSource dataSource) {  
        jdbcTemplate = new JdbcTemplate(dataSource);  
    }  
}
```

Managing the Database

- Note the use of `@Sql` on the class to execute the database setup scripts before each test case
 - Because `@SpringBootTest` runs Tomcat in a different thread than the test cases themselves, `@Transactional` has no effect here
 - So, we need to re-initialize the database after each test case
- For some test cases, we'll need to query or modify the database directly, so we'll use Spring's `JdbcTestUtils` class
 - We will call methods like `countRowsInTable()` and `deleteFromTables()`

Managing the Database in a Test

```
@Test
public void testAddWidgetToWarehouse() throws Exception {
    int widgetCount = Jdbc TestUtils.countRowsInTable(jdbcTemplate, "widgets");
    int id = 42;
    Widget w = new Widget(id, "Test widget", 4.52, 20, 10);

    ResponseEntity<DatabaseRequestResult> response =
        restTemplate.postForEntity("/warehouse/widgets", w, DatabaseRequestResult.class);

    // verify the response HTTP status and response body
    assertThat(response.getStatusCode(), is(equalTo(HttpStatus.OK)));
    assertThat(response.getBody().getRowCount(), is(equalTo(1))); // {"rowCount": 1}

    // verify that one row was added to the Widgets table
    int newWidgetCount = Jdbc TestUtils.countRowsInTable(jdbcTemplate, "widgets");
    assertThat(newWidgetCount, is(equalTo(widgetCount + 1)));

    // verify that the new widget is in the Widgets table
    assertThat(1, equalTo(
        Jdbc TestUtils.countRowsInTableWhere(jdbcTemplate, "widgets", "id = " + id)));
}
```

Testing for Exceptions

```
@Test
public void testQueryForWidgetById_DbException() {
    // drop the Widgets table to force a database exception
    JdbcTestUtils.dropTables(jdbcTemplate, "widgets");

    String request = "/warehouse/widgets/99";

    ResponseEntity<Widget> response = restTemplate.getForEntity(request, Widget.class);

    // verify the response HTTP status
    assertThat(response.getStatusCode(), is(equalTo(HttpStatus.INTERNAL_SERVER_ERROR)));
}
```

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



30 min

- Follow the directions in your Exercise Manual for this exercise

Chapter Concepts

RESTful Service Controller Responsibilities

Testing RESTful Services

Testing the Back End

Testing the Web Layer

End-to-End Testing

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The responsibilities of a RESTful Service Controller
- Performing unit testing of POJOs in the back end
- Testing the RESTful Service controller in the web (HTTP) environment
- Performing end-to-end testing of a RESTful Service using `TestRestTemplate`

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 4: Securing RESTful Web Services

Chapter Overview

In this chapter, we will explore:

- Techniques for securing RESTful web services
- Session management practices
- Authentication and authorization
- The details of OAuth for authorization

Chapter Concepts

Session Management

Authentication and Authorization

Securing RESTful Web Services

Chapter Summary

Session Management

- RESTful services should use session-based authentication
 - Establish a session token via POST
 - Or use an API key as a POST body argument
- Critical information should not appear in the URL
 - Username
 - Password
 - Session token
 - API key

OWASP has a series of “cheat sheets” for developers:

<https://cheatsheetseries.owasp.org/>

For more information about RESTful security, see the following:

https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html

Protect Session State

- RESTful services are designed to be stateless
- Use only a session token or API key
 - To maintain client state in a server-side cache
- To prevent replay exploits
 - Use a time limited key or token

Chapter Concepts

Session Management

Authentication and Authorization

Securing RESTful Web Services

Chapter Summary

RESTful Authentication

- Several techniques to accomplish this
 - Basic authentication
 - HMAC
 - OAuth

Basic Authentication

- The simplest way to manage authentication
 - Use HTTP basic authentication
- Username and password are passed in the HTTP header
 - But **not** encrypted
- Use only with SSL/TLS

HMAC Authentication

- Hash-based Message Authentication (HMAC)
 - Sends a hashed version of the password
 - With some other information
 - Such as a timestamp, a nonce, or the md5 hash of the message body
 - To help prevent (or at least detect) any tampering of the message body
- Nonce
 - A number that is used only once
 - To prevent replay attacks

OAuth

- “An open protocol to allow secure authorization in a simple and standard method from web, mobile, and desktop applications.”¹
- Enables third-party applications to obtain limited access to a web service
 - Known as “secure designated access”
- Example scenario:
 - You can grant ESPN.com the right to access your Facebook profile
 - Without knowing your Facebook password
- OAuth does not share passwords
 - Instead, it uses authorization tokens
 - Tokens prove an identity between consumers and service providers

1. <https://oauth.net/>

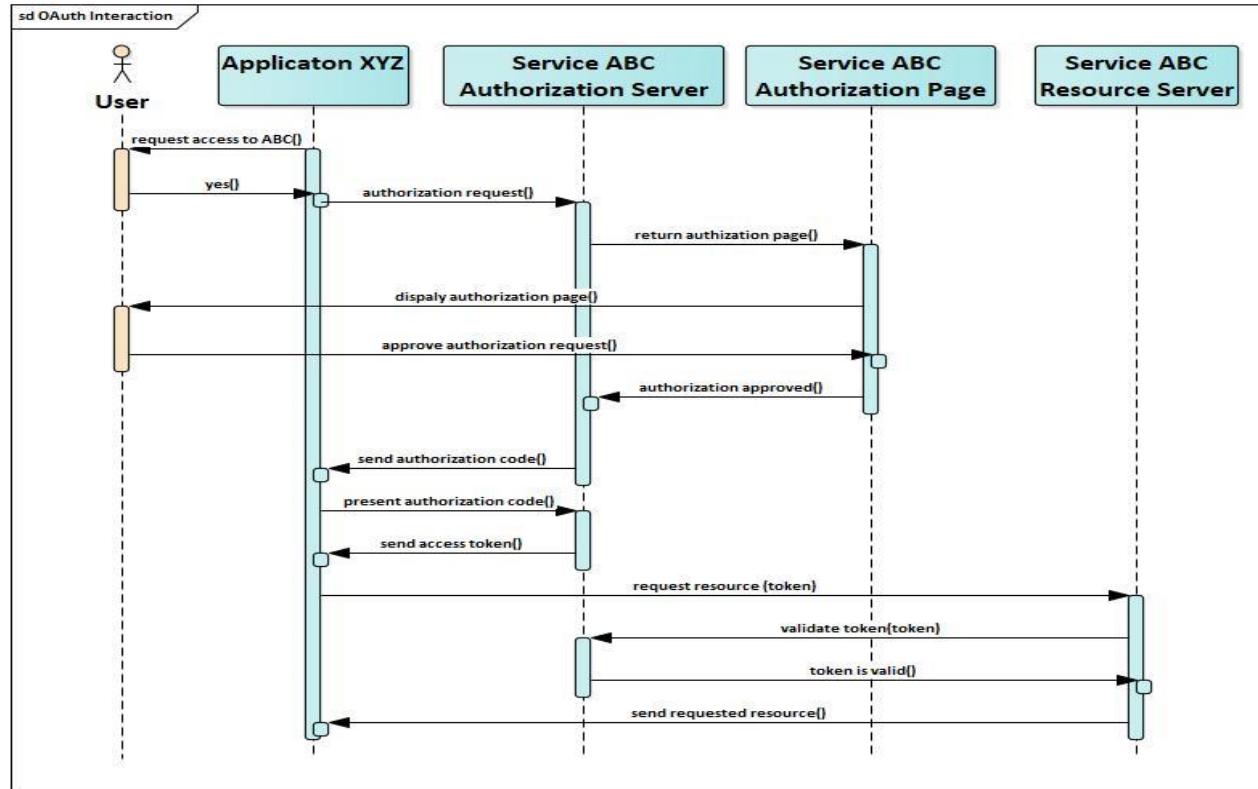
Application Registration

- Before using OAuth, the application must register with the service
 - Using the “developer (API)” portion of the service website
- The following information must be provided:
 - Application name
 - Application website
 - Redirect URI
 - The service will redirect the user to this location after authorization
 - This is the part of your application that will handle authorization codes
- Once the application is registered, the service issues client credentials
 - Client identifier
 - Publicly known string that identifies the application
 - Client secret
 - Used to authenticate the identity of the application to the service API

Authorization Grant

- There are four types of grant types supported by OAuth2
- Authorization Code
 - Used with server-side application
 - Most commonly used
- Implicit
 - Used with mobile applications or web applications that run on the user's device
- Resource Owner Password Credentials
 - Used with trusted applications
 - Like applications owned by the service
- Client Credentials
 - Used with applications API access

OAuth Authorization Code Flow



This diagram is based on RFC 6749, 4.1 <https://tools.ietf.org/html/rfc6749#section-4.1>

Stepping Through the Authorization Code Flow

1. Application XYZ asks the user for access to resources provided by Service ABC.
 - a. The user agrees.
2. Application XYZ sends an authorization request to Service ABC Authorization Server.
 - a. Using its authorization endpoint.
3. Service ABC returns an authorization page.
4. The authorization page is displayed to the user.
5. The user approves the requested permissions.
 - a. Authorization page form is submitted to the Service ABC Authorization Server.
6. Service ABC Authorization Server issues a short-lived authorization code.
 - a. Sent to application XYZ.

Stepping Through the Authorization Code Flow (continued)

7. Application XYZ presents the authorization code to Service ABC Authorization Server.
 - a. Sent to Service ABC Authorization Server token endpoint.
8. Service ABC Authorization Server issues an access token for application XYZ.
9. Application XYZ requests the desired resource from Service ABC Resource Server.
 - a. Presents the access token.
10. Service ABC Resource Server requests verification of access token.
 - a. Sends access token to Service ABC Authorization Server.
11. Service ABC Authorization Server returns confirmation of the access token.
12. Service ABC Resource Server returns the requested resource to application XYZ.



Exercise 4.1: Using OAuth2 for Authorization

20 min

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

Session Management

Authentication and Authorization

Securing RESTful Web Services

Chapter Summary

RESTful Web Security

- The following recommendations are based on the OWASP REST Security Cheat Sheet:
 - HTTPS
 - Security tokens
 - API keys
 - Restrict HTTP methods
 - Input validation
 - Security headers
 - HTTP return codes

For more information about RESTful security, see the following:
https://www.owasp.org/index.php/REST_Security_Cheat_Sheet

HTTPS

- Secure RESTful services should provide only HTTPS endpoints
- This protects all data transferred between the client and the service
- For highly sensitive or privileged services, consider using client-side certificates

Security Tokens

- RESTful services can require a security token from the client
 - The token can be used for access control decisions
- JSON Web Tokens are often the preferred format for security tokens
 - <https://tools.ietf.org/pdf/rfc7519.pdf>
 - A cryptographic signature is the recommended way to protect the integrity of the security token
- Contents of a JSON Web Token:
 - Issuer – is this a trusted issuer?
 - Audience – is the relying party in the target audience for this token?
 - Expiration time – is the token still valid?
 - Not before time – is the token valid yet?

JSON Web Tokens (JWT)

When to use JSON Web Tokens?

- Authorization
 - When the user logs in, the JWT will be generated and returned
 - Each subsequent request includes the JWT
- Information exchange
 - Transmits information securely
 - Signing the JWT with a private key assures the receiver who the sender is
 - Also verifies the content has not been modified

The contents (header, payload) of a JWT can be decoded and read by anyone

Encrypt sensitive data in the payload

JSON Web Token Structure

- JSON Web Tokens contain three parts separated by dots (.)
 - Header
 - The type of the token (JWT)
 - The signing algorithm (HMAC, SHA256, RSA, ...)
 - Payload
 - Contains claims about an entity (usually the user)
 - Signature
 - Calculated from the encoded header, encoded payload, and a secret
 - Using a specified algorithm

Example JWT

■ Header

- { "alg": "HS256", "typ": "JWT" }

■ Payload

- { "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }

■ Signature

- HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)

■ JWT

- eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o



Exercise 4.2: JSON Web Tokens (JWT)

20 min

- Follow the instructions in your Exercise Manual for this exercise

API Keys

- API keys can reduce the impact of a denial-of-service attack by doing the following:
 - Require an API key for every request to the service
 - Return a 429 HTTP response code if requests are arriving too quickly
 - Revoke the API key if the client violates the usage agreement
- Caution
 - Do not rely exclusively on API keys to protect sensitive, critical, or high-value resource

Restrict HTTP Methods

- Use a whitelist of permitted HTTP methods
 - Only support those on the whitelist
- Reject all requests not on the whitelist with a 405 HTTP response code of “Method not allowed”
- Verify the client is authorized to request the incoming HTTP method on the service

Protect HTTP Methods

- Make sure the incoming request has rights to execute the requested method
- Not everyone should be able to `DELETE` a resource
- Validate the incoming HTTP method
 - Against the session token or API key

Input Validation

- The usual “don’t trust the client” recommendations apply here
- Validate input parameters
- Validate input length, range, and data type
- Reject unexpected or illegal content
- Log validation failures
 - Assume that anyone performing many requests that fail validation rules is not to be trusted

Security Headers

- Send security headers
 - Always send the Content-Type header with the correct type specified
 - This insures the browser interprets the response correctly
- Send an X-Content-Type-Options: nosniff
 - Make sure the browser does not try to detect a different Content-Type
 - This helps to prevent XSS exploits
- Clients should send an X-Frame-Options: deny
 - Protect against drag and drop clickjacking attacks in older browsers

Protect Against Cross-Site Forgery

- Ensure that all PUT, POST, PATCH, and DELETE requests are protected
 - From cross-site request forgery
- Use a token-based approach
- See this cheat sheet for more detailed information:
 - [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet)

Cross-Site Request Forgery

```
<!-- This is embedded in another domain's site -->  

```

- Common approach is to use the tag
- This causes the user browser to send a request to a malicious server
- Mitigation techniques include:
 - Short-lived JWTs
 - Special headers added only when they originate from the correct origin
 - Per session cookies
 - Per request tokens
- If JWTs (and session data) are not stored as cookies, CSRF attacks are not possible

HTTP Return Codes

Status code	Message	Description
200	OK	Response to a successful REST API action
201	Created	The request has been fulfilled and resource created. A URI for the created resource is returned in the Location header.
202	Accepted	The request has been accepted for processing, but processing is not yet complete
204	No Content	The request succeeded, but no data is returned in the response
400	Bad Request	The request is malformed, such as message body format error
401	Unauthorized	Wrong or no authentication ID/password provided
403	Forbidden	It's used when the authentication succeeded but authenticated user doesn't have permission to the request resource
404	Not Found	When a non-existent resource is requested
406	Unacceptable	The client presented an unsupported content type in the Accept header
405	Method Not Allowed	The error for an unexpected HTTP method.
413	Payload too large	Use it to signal that the request size exceeded the given size
415	Unsupported Media Type	The requested content type is not supported by the REST service
429	Too Many Requests	The error is used when there may be DOS attack detected or the request is rejected due to rate limiting

Chapter Concepts

Session Management

Authentication and Authorization

Securing RESTful Web Services

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Techniques for securing RESTful web services
- Session management practices
- Authentication and authorization
- The details of OAuth for authorization

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 5: Cloud Design Patterns

Chapter Overview

In this chapter, we will explore:

- Some challenges in developing applications for cloud deployment
- Some cloud related design patterns
 - Circuit breaker
 - Transparency
 - Health check
 - Service discovery
 - Fail fast
- Creating and deploying an AWS Lambda function

Chapter Concepts

Challenges in Cloud Development

Cloud Patterns

AWS Lambda Functions

Chapter Summary

Challenges in Cloud Development

- Availability
- Data Management
- Messaging
- Management and Monitoring
- Performance and Scalability
- Resiliency
- Security

There are design patterns that address all of these challenges, and many more as well

This is based on information at:
<https://docs.microsoft.com/en-us/azure/architecture/patterns/>

Availability

■ Availability

- The proportion of the time that the system is functional and working properly
- Usually measured as a percentage of uptime

■ Factors that can affect availability include:

- System errors
- Infrastructure problems
- Malicious attacks
- System load

■ Availability is often part of a Service Level Agreement (SLA)

Data Management

- Influences most of the quality attributes of a cloud application
- Data is typically located in multiple locations
 - And across multiple servers
- Data consistency must be maintained
 - Often requiring synchronization across several locations

Messaging

- Cloud applications execute in a distributed environment
- Components and services are loosely coupled for scalability
- Asynchronous communication is widely popular
 - Provides benefits such as scalability
- Asynchronous messaging presents challenges
 - Message ordering
 - Poison message management
 - Idempotency

Management and Monitoring

- A cloud application executes in a remote data center
- Management and monitoring is more difficult than a local installation
- Services must expose runtime information for administrators and operators

Performance and Scalability

- Performance is the responsiveness of an application to execute a request
- Scalability is the ability to handle increases in load without impacting performance
- Cloud applications experience peaks and valleys in activity
- Cloud applications should be able to respond quickly to activity changes

Resiliency

- Cloud applications should be able to gracefully handle and recover from failures
- There are many sources of potential failures
 - Shared services
 - Competition for resources and bandwidth
 - Remote communications
- Detecting failures and recovering quickly is essential

Security

- Preventing malicious or accidental actions outside of expected usages is important
- Preventing loss or disclosure of sensitive information is essential
- Cloud applications must be designed and deployed to protect them from malicious attacks
 - Restrict access to known, approved users
 - Protect sensitive data



HANDS-ON
EXERCISE

Exercise 5.1: Researching Cloud Design Patterns

30 min

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Challenges in Cloud Development

Cloud Patterns

AWS Lambda Functions

Chapter Summary

Bad Things Can and Will Happen

- Many incidents have occurred that verify this maxim
- An airline flight search service is brought to a halt
 - By one method that did not catch an exception
- An online retailer loses massive amounts of revenue
 - Due to some especially costly downtime
- A web application becomes unexpectedly popular
 - And cannot respond to its sudden popularity
 - And a flood of web requests
- Operators for the Three Mile Island reactor:
 - Misinterpreted the meaning of coolant pressure and temperature values
 - Leading them to take exactly the wrong action at every turn

Stability Antipatterns

■ Integration points

- Communication between components
- Subject to network communication problems
- The number one source of problems

■ Blocked threads

- Can happen anytime you check out a resource from a pool
- Or make calls to an external system
- Can cause the system to “hang”

■ Cascading failures

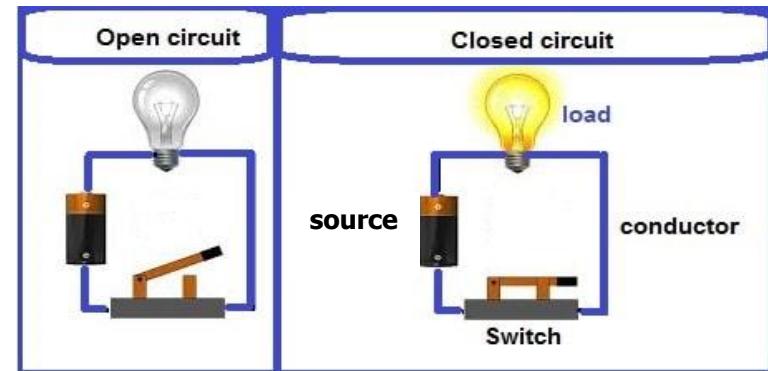
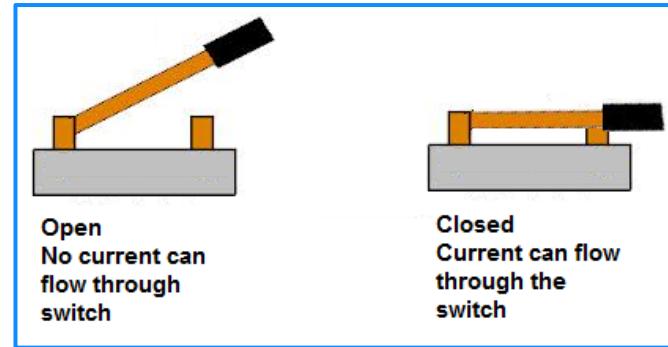
- Common example is a database failure
- A problem in one component may spread to other components
- The problem “jumps the gap”
- The number one accelerator of problems

Stability Solutions

- Circuit breaker
 - Detects when a service is not available
 - Prevents an application from repeatedly trying to call on a failed service
- Transparency
- Health checks
- Service discovery
- Fail fast

Electrical Circuit State Machine

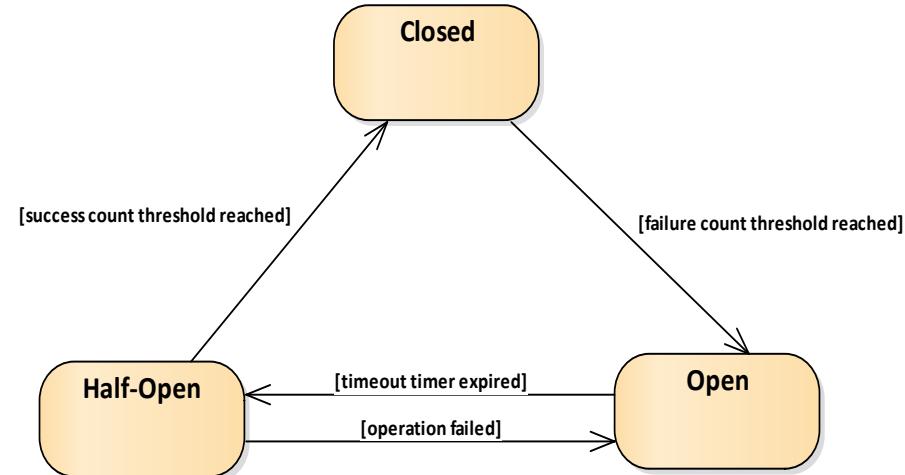
- A circuit breaker is similar to an electrical circuit with a switch
 - The switch determines if the electrical circuit is completed to allow electricity to flow through the circuit and light the bulb
 - The electrical circuit is one of two states
- Closed
 - Electricity flows through the circuit
 - The bulb lights up
- Open
 - The flow of electricity is broken
 - The bulb does not light up



Images from <https://environmentalb.com/electrical-circuits/> and <http://www.learningaboutelectronics.com/Articles/Door-alarm-circuit.php>

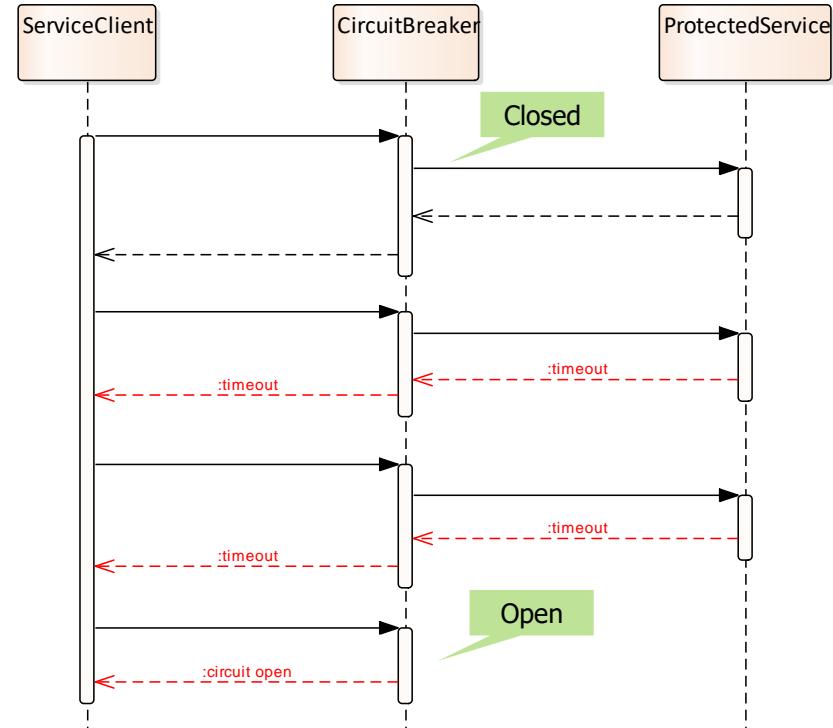
Circuit Breaker State Machine

- A circuit breaker acts as a proxy for the service that may fail
 - It can be modeled with a state machine with the following three states:
- Closed
 - Requests are routed to the service
- Open
 - Requests fail immediately
 - An exception is returned
- Half-Open
 - A limited number of requests are passed to the service to test its availability
 - Prevents the service from being flooded with requests



Circuit Breaker

- One of the most effective patterns to combat cascading failures
- Circuit breaker wraps the component to be protected
 - Monitors it for failures
 - Has a threshold for failed requests
- The circuit breaker will trap calls to a failed service
 - After the threshold is exceeded
 - May utilize a fallback strategy
 - Perhaps route call to a backup service
 - Or return a cached response
- The problem should be reported to ops





Exercise 5.2: Using the Circuit Breaker Pattern

30 min

- Please refer to your Exercise Manual to complete this exercise

Transparency

- Transparency provides information about environmental awareness
 - Historical trends
 - Current system state
 - Such systems will be much easier to debug

- Transparent systems must communicate with the outside world
 - Logging integrated into component source code
 - Make log destinations configurable
 - Use log levels
 - Instance metrics
 - Components should send its metrics to a destination such as a log file
 - Metrics may be difficult to interpret
 - It may take time to learn what is “normal”
 - And what is not

Health Checks

- A health check:
 - Is an application's view of its own health
 - Can help determine what the metrics reported by components actually mean
- What should a health check report?
 - Host ip address
 - Application version
 - Is the component accepting requests?
 - The status of resources
 - Connection pools
 - Caches
 - Circuit breakers

Health Checks (continued)

- The application provides an endpoint (service method) for health monitoring
 - Performs necessary checks
 - Returns an indication of its status
 - The endpoint should require authentication

- A health monitoring system is composed of two factors
 - The checks performed by the application when requested
 - Analysis of the response from the application

Service Discovery

- Service discovery has two parts
 - First, services must be registered somewhere
 - For example, a dynamic pool with load balancer
 - Second, services must be discoverable
 - Clients must know where to send a service discovery request
- Service discovery is a service
 - It can fail
 - Or become overloaded
 - Clients should cache services after discovery
- Don't roll your own discovery service
- There are available implementations
 - Apache Zookeeper
 - HashiCorp's Consul
 - Docker Swarm

Fail Fast

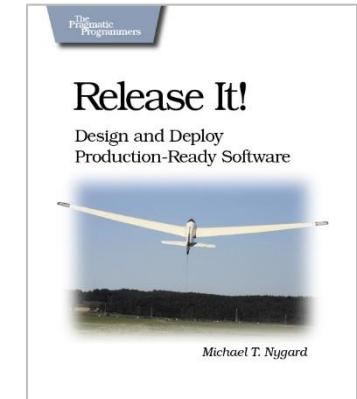
- Slow responses are frustrating
 - Slow failures are even worse!
- If it is possible to determine that an operation will fail:
 - It is better to fail fast
- How to make that determination?
 - Verify any user input before checking on resources
 - Verify all the necessary resources are available before starting to process the request
 - Database connections, circuit breakers, etc.

Fail Fast Guidelines

- Avoid slow responses
 - Don't make users wait for an error message
- Fail fast
 - Don't rely on a timeout to indicate an error condition
- Do basic input validation first
 - If the input is not valid, immediately return an error
- Reserve resources and verify integration points
 - Before starting the response process

Cloud Pattern Resources

- DZone Cloud Design Patterns and Practices
 - <https://dzone.com/articles/cloud-design-patterns-and-practices>
- Microsoft Cloud Design Patterns
 - <https://docs.microsoft.com/en-us/azure/architecture/patterns/>
- CloudAcademy AWS Cloud Design Patterns
 - <https://cloudacademy.com/blog/aws-cloud-design-patterns/>
- Arcitura Education Cloud Computing Design Patterns Catalog
 - <https://patterns.arcitura.com/cloud-computing-patterns>
- Amazon Web Services Cloud Design Patterns
 - http://en.clouddesignpattern.org/index.php/Main_Page
- *Release It!*, Michael Nygard



Chapter Concepts

Challenges in Cloud Development

Cloud Patterns

AWS Lambda Functions

Chapter Summary

AWS Lambda Functions

- AWS Lambda is a compute service
 - Lets you run code without managing any servers or resources
 - Executes code on demand
 - Scales automatically based on the number of requests
 - Can be written in several different programming languages
- There are several ways in which your code will run
 - Responding to an event
 - Such as modification of data in an Amazon S3 bucket or a DynamoDB table
 - Responding to an HTTP request
 - For example, a RESTful service call
 - Invoked by using API calls with AWS SDKs
- Serverless applications can be composed of functions
 - Triggered by events

Benefits of AWS Lambda Functions

- Serverless functions or applications
 - There are no servers for you to manage or provision
- Continuous scaling
 - Your application is automatically scaled in response to demand
 - The code runs in parallel
 - Each trigger is processed individually
- Cost efficient
 - You are charged only for the time your code is executing
 - Metered in 100ms increments
 - There is no charge when the code is not running

Lambda Function Reference Architectures

- Web application
 - AWS Lambda functions provide the backend business logic
- Mobile backend
 - AWS Lambda functions run in response to requests from mobile applications
 - Possible functionality:
 - Access to data stored in a DynamoDB database
 - Support users communicating with each other asynchronously
- Real-time stream processing
 - Real-time event data is processed by AWS Lambda functions
 - Possible functionality:
 - Storage of data in a backend datastore
 - Monitoring of aggregate metrics

Core Concepts of Lambda Programming

- Stateless
- Handler
 - The function that Lambda calls to start execution of your Lambda function
- Context
 - A context object is passed to the handler function
- Logging
 - Log entries are written to CloudWatch logs
- Exceptions
 - Used to indicate error conditions
- Concurrency
 - Each instance of your function handles only one request

Creating a Lambda Function

- The following is a simple Lambda function that returns a warm greeting

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class Greeter {
    public Greetings myHandler(String name, Context context) {
        LambdaLogger logger = context.getLogger();
        logger.log("Received : " + name);
        String greeting = name + " Welcome to the Wonderful World of Lambda";
        Greetings greetings = new Greetings(greeting);

        return greetings;
    }
}
```

Building a Lambda Function in Java

- The deployment package can be a zip or executable jar file
- To build and deploy with Eclipse and Maven:
 1. Create a new Maven project in Eclipse.
 2. Add the aws-lambda-java-core dependency to the `pom.xml` file.
 3. Create the Java class and write the code for the Lambda function.
 4. Build the project using Maven build.
 5. Add the maven-shade-plugin plugin to the `pom.xml` file.
 6. Rebuild the project.
 - a. The resulting jar file can be deployed to AWS Lambda.
 - b. Can use the Lambda console to upload and test the Lambda function.

Deploying a Lambda Function in Java

- Once the Lambda function project has been built, it must be deployed to AWS Lambda
- This can be done by using the Lambda console to upload the file (zip or jar)
- It is also possible to use the AWS Command Line Interface (CLI)
 - The AWS CLI is a separate tool that must be installed
 - The CLI provides functionality to manage and use Lambda functions from the command line



30 min

Exercise 5.3: Creating and Deploying a Lambda Function

- Please refer to your Exercise Manual to complete this exercise

Chapter Concepts

Challenges in Cloud Development

Cloud Patterns

AWS Lambda Functions

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- Some challenges in developing applications for cloud deployment
- Some cloud related design patterns
 - Circuit breaker
 - Transparency
 - Health check
 - Service discovery
 - Fail fast
- Creating and deploying an AWS Lambda function

Fidelity LEAP

Technology Immersion Program

Dynamic Web Application Development

Chapter 6: Node.js

Chapter Overview

In this chapter, we will explore:

- How to use node.js to become your web server
- What modules are
- How to access databases from JavaScript

Chapter Concepts

Features and First Steps

The package.json File

Basic Database Access

JSON data-mock File

Chapter Summary

What Is Node.js?

- Node.js is a platform built on Google's V8 JavaScript runtime for easily building fast, scalable network applications
- Available for download from <https://nodejs.org/en/>

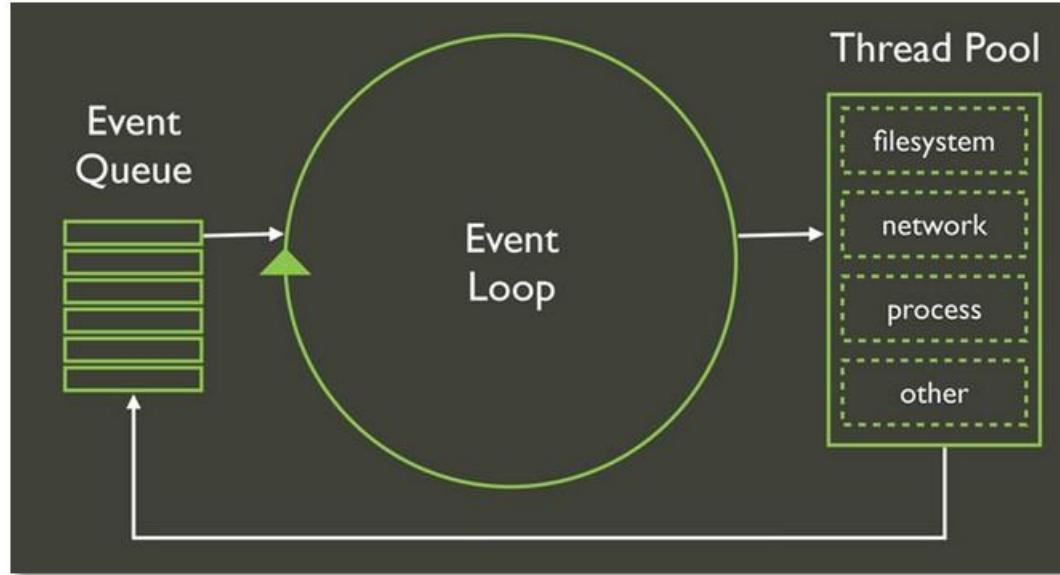


Node.js Features

- Uses JavaScript so allowing client- and server-side development with single language
- Event-driven, lightweight, and efficient with high scalability
- Non-blocking I/O model
 - Asynchronous execution
- High performance—very fast request processing cycle
 - V8 introduced compiled JavaScript
- Supports re-useable modules
- Provides a package manager called “Node Package Manager” (npm)

The Event Loop

- Node.js is an event-driven, single-threaded, non-blocking I/O framework



Blocking Code

- Blocking code forces other JavaScript code in the Node.js process to wait
 - Until the blocking code completes
 - The event loop must wait while the blocking code runs
- The i/o methods in Node.js that are blocking have names that end with Sync

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

// blocks on this line until file is read
const data = fs.readFileSync('/file.md');
console.log(data);
moreWork(); //will run after console.log
```

Non-Blocking Code

- Non-blocking code executes asynchronously
 - All the i/o methods in Node.js provide asynchronous versions
- The asynchronous calls allow higher throughput
 - The callback function executes after the non-blocking code completes

```
const fs = require('fs');
function read_file_cb(err, data) {
  if (err) throw err;
  console.log(data);
}

fs.readFile('/file.md', read_file_cb);

moreWork(); //will run before console.log
```

Getting Started with Node.js

■ Steps to getting started with Node.js:

1. Create a .js file such as `index.js` in a new directory
2. Identify which Node.js “modules” are needed in the code at the top of the file
3. Add your code and save the file
4. Run the following command to run your code:

```
node index.js
```



Exercise 6.1: Hello World

15 min

- Follow the instructions in your Exercise Manual for this exercise

What Are Node.js Modules?

- Node.js provides a minimalist core library composed of modules
- Examples of a few built-in “core” modules:

Module	Description	Usage
fs	Provides file I/O	<code>var fs = require('fs');</code>
http	HTTP server and client functionality	<code>var http = require('http');</code>
net	Asynchronous network wrapper	<code>var net = require('net');</code>
path	Utilities for handling and transforming file paths	<code>var path = require('path');</code>
util	Various utility functions used by Node.js and custom applications	<code>var utils = require('util');</code>

- Custom modules can be installed using a tool called “npm” (more on this later)

Node.js Documentation

- Get documentation on Node.js and the core modules at: <https://nodejs.org/api>

The screenshot shows the official Node.js documentation website. At the top is a dark header bar with the Node.js logo and navigation links for HOME, ABOUT, DOWNLOADS, DOCS, FOUNDATION, GET INVOLVED, SECURITY, and NEWS. A green arrow points down from the DOCS link to the main content area. The content area has a light gray background. On the left, there's a sidebar with a green header labeled 'Docs' containing links to 'ES6 in Node.js', 'FAQ', and 'API'. The main content is titled 'About Docs' and contains two paragraphs of text. The first paragraph discusses the different types of documentation available on the site. The second paragraph provides a detailed explanation of what API reference documentation entails.

node.js

HOME | ABOUT | DOWNLOADS | DOCS | FOUNDATION | GET INVOLVED | SECURITY | NEWS

Docs

ES6 in Node.js

FAQ

API

About Docs

It's important for Node.js to provide documentation to its users, but documentation means different things to different people. Here, on nodejs.org, you will find three types of documentation, reference documentation, ES6 features, and frequently asked questions.

Our API reference documentation is meant to provide detailed version information about a given method or pattern in Node.js. From this documentation you should be able to identify what input a method has, the return value of that method, and what, if any, errors may be related to that method. You should also be able to identify which methods are available for different versions of Node.js.

Creating a Node Server

1. The following code creates and runs a server listening on port 8080

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

http.createServer( (req, res) => {
    res.end('Hello World from the Server!');
}).listen(8080);
```

2. To launch the server, run `node myFile.js` at the command line
3. Navigate to <http://localhost:8080> in your browser to communicate with the server



Exercise 6.2: Creating a Server

20 min

- Follow the instructions in your Exercise Manual for this exercise

Creating an HTTP Server to Return HTML

- Node.js can be used to create an HTTP server that returns HTML (or other) content types

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

http.createServer( (req, res) => {

    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.write('<h1>Hello World</h1>');
    res.end();

}).listen(8080);
```



Exercise 6.3: Returning HTML

20 min

- Follow the instructions in your Exercise Manual for this exercise

Node.js Modules

- Modules are re-useable/self-contained pieces of software
- Node.js provides a module system for loading application resources:
 - Core Modules – Native modules built-in to Node.js such as http, networking, file system, and more
 - File Modules – Used to load custom modules from .js files
- Packages/Modules can be loaded using **npm** – Node Package Manager

Loading Modules

- Modules are loaded by using `require()`
- Core modules are defined using a shortcut name:

```
const http = require('http');
const net = require('fs');
```

- File modules can be loaded by defining a path:

```
const parser = require('./stringParser');
```

Using a Core Node.js Module

- File System module (fs) can be used to load files

```
const fs = require('fs');
const http = require('http');

function create_server_callback(req, res) {
  fs.readFile('myfile.html', (err, fileData) => {
    if (err) {
      console.log(err);
    } else {
      res.write(fileData);
      res.end();
    }
  });
}

const server = http.createServer(create_server_callback);
server.listen(8081);
```

Async callback function

<https://nodejs.org/api/fs.html>



Exercise 6.4: Using a Core Module

20 min

- Follow the directions in your Exercise Manual for this exercise

Creating and Loading a Custom Module

- A custom module “exports” functionality using *module.exports* or the *exports* alias

hello.js

```
module.exports = function() {  
    return 'Hello!';  
}
```

index.js

```
const hello = require('./hello');  
  
let text = hello();  
  
console.log(text);
```



Installing Modules with npm

- npm can be used to access packages from <http://npmjs.org>
 - Access thousands of packaged modules
 - Store modules globally or locally in a project
 - Handles dependencies automatically

```
npm help install  
npm install socket.io -g  
npm install underscore  
npm ls
```

Store module in global location

Store module in local node_modules folder

List local modules

Chapter Concepts

Features and First Steps

The package.json File

Basic Database Access

JSON data-mock File

Chapter Summary

The package.json File

- Node.js projects normally have a **package.json** file at their root that defines information such as:
 - Project author
 - Source control
 - Startup scripts
 - License information
 - Project dependencies
 - Project dev dependencies
 - More
- Create a **package.json** file by running **npm init**

package.json File Example

- Version Syntax: majorVersion.minorVersion.patchLevel

- With no ~ or ^:
all numbers must match exactly
- ~ patch level may be higher
- ^ minor version and/or
patch level may be higher
- To update to latest versions:
 - Run “npm update”
 - npm will modify package.json
with new version numbers

```
{  
  "name": "myApp",  
  "version": "0.1.0",  
  "description": "My super cool node app!",  
  "main": "server.js",  
  "author": "John \"Guru\" Doe",  
  "license": "ISC"  
  "dependencies": {  
    "rxjs": "~6.5.3",  
    "socket.io": "^1.3.5"  
  },  
  "devDependencies": {  
    "gulp": "^3.8.11"  
  }  
}
```

Installing Modules into package.json

- The `npm install` command can be used to install packages and update `package.json`:

`--save`

`--save-dev`

```
npm install socket.io --save  
npm install gulp --save-dev
```

Install module and update
dependencies property

Install module and update
devDependencies property

Note:

As of **npm** version 5,
`--save/-/--save-prod`"
are the defaults, so
`"npm install <module>"`
saves dependency to
`package.json` by default!

Note:

- `devDependencies` are for the development-related scripts, e.g., unit testing, packaging scripts, documentation generation, etc.
- dependencies are required for production use, and assumed required for dev as well



Exercise 6.5: Using npm

20 min

- Follow the directions in your Exercise Manual for this exercise

Chapter Concepts

Features and First Steps

The package.json File

Basic Database Access

JSON data-mock File

Chapter Summary

Database and node.js

- Most database have easy to install processes (e.g., npm install, etc.), installing the necessary drivers and modules
 - E.g., for MS SQL Server its: `npm install mssql`
- Oracle and node.js
 - Oracle is an exception in the way the driver is installed
 - Separate installation of thin client is required
 - Knowledge exists within Fidelity and can be obtained if required
 - See the installation instructions here:
<https://oracle.github.io/node-oracledb/INSTALL.html>

Simple Node.js to Database Example

- MongoDB is a commonly used No-SQL database, particularly for Big Data scenarios
- Example: Create database called "mydb"
- Query for all customers with address: Park Lane 38

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/mydb";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var query = { address: "Park Lane 38" };
  var dbo = db.db("mydb");
  dbo.collection("customers").find(query).toArray(
    function(err, result) {
      if (err) throw err;
      console.log(result);
      db.close();
    });
});
```

Custom Approach

- Each database (MySQL, MSSQL, Oracle, etc.) has its own approach
- Research the Internet and sufficient code examples will be available
- *Note:* Don't forget to run the `npm install` from within your express project or the modules won't be found!

Chapter Concepts

Features and First Steps

The package.json File

Basic Database Access

JSON data-mock File

Chapter Summary

JSON as Data File Format

- JSON, like XML, can be used to store data
- When databases are not available at the time of development, mock-data source can be used to emulate a data depository
- Check out: ./Day6/Resources/contact.json
- The code shown will load the entire file and exports.list will parse it to a JSON object, making it easy to access in JavaScript

```
var fs = require('fs');

function read_json_file() {
  var file = './data/contact.json';
  return fs.readFileSync(file);
}

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

module.Exports vs. exports.function

- ▀ Simplistically, there two ways to export functionality of a module

1. The module is handled like a function

hello.js

```
module.exports = () => 'Hello!';
```

index.js

```
const hello = require('./hello');  
console.log(hello());
```

- No further functions can be added, without reveal pattern

2. The module is handled like an object

- Every accessible function must lead with `exports.functionname`

multi.js

```
exports.list = () => 'List';  
exports.other = () => 'Other';
```

index.js

```
const multi = require('./multi');  
console.log(multi.list()); // List  
console.log(multi.other()); // Other
```

Chapter Concepts

Features and First Steps

The package.json File

Basic Database Access

JSON data-mock File

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- How to use node.js to become your web server
- What modules are
- How to access databases from JavaScript

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 7: Node.js and Express

Chapter Overview

In this chapter, we will explore:

- How to debug a node.js application
- How to utilize Express to program faster RESTful services with node.js
- How to test your RESTful services with Jasmine

Chapter Concepts

Debugging a Node Application

Working with Node and Express

CORS (Cross-Origin Resource and Sharing)

Building RESTful Services Faster with Express

Basic Use of Jasmine with node.js and Express

Chapter Summary

Debugging Node.js

- Several techniques can be used to debug Node.js
 - **Node Debug:** <http://nodejs.org/api/debugger.html>
 - **Node-Inspector:** <https://github.com/dannycoates/node-inspector>
 - **Visual Studio Code**
- Since it provides more and superior features, we will only focus on Visual Studio Code-based debugging

Debugging with Visual Studio Code

The screenshot shows the Visual Studio Code interface during a debugging session. The main editor pane displays a JavaScript file named `pi.js` with the following code:

```
function calc_pi() {
    let n = 200000;
    let pi = 0;
    for (var i = 0; i < n; i++) {
        let temp = 4 / (i * 2 + 1);
        if (i % 2 == 0) {
            pi += temp;
        } else {
            pi -= temp;
        }
    }
    return pi;
}
let pi = calc_pi();
console.log(pi);
```

The line `pi += temp;` is highlighted with a yellow background, indicating it is the current line being executed or has just been executed. The status bar at the bottom right shows the line number as `Ln 7, Col 9`.

The left sidebar contains several open tabs and sections:

- VARIABLES**: Shows a `Block` section with `this: global` and `temp: 4`, and Local and Global sections.
- WATCH**: An empty section.
- CALL STACK**: Shows a single entry: `calc_pi` from `pi.js` at line 7:9, with a note "(anonymous function)" and "PAUSED ON BREAKPOINT". Below it, "Show 6 More: read-only core mod..." is visible.
- LOADED SCRIPTS**: An empty section.
- BREAKPOINTS**: Shows checkboxes for "All Exceptions" and "Uncaught Exceptions", with a count of 1 indicated at the bottom.

The bottom status bar also includes icons for Go Live and a bell notification.



Exercise 7.1: Debugging Node.js

30 min

- Follow the directions in your Exercise Manual for this exercise

Chapter Concepts

Debugging a Node Application

Working with Node and Express

CORS (Cross-Origin Resource and Sharing)

Building RESTful Services Faster with Express

Basic Use of Jasmine with node.js and Express

Chapter Summary

Getting Started with Express

- The Express module provides a web framework you can build on
- Build single or multi-page applications
- Get started with Express by using **npm**:

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

Scaffolding with Express

- Express can generate scaffolding for a site

```
Command Prompt

D:\NextGenLeap\RESTful Services Solutions\Day6>express myApp

warning: the default view engine will not be jade in future releases
warning: use `--view=jade` or `--help` for additional options

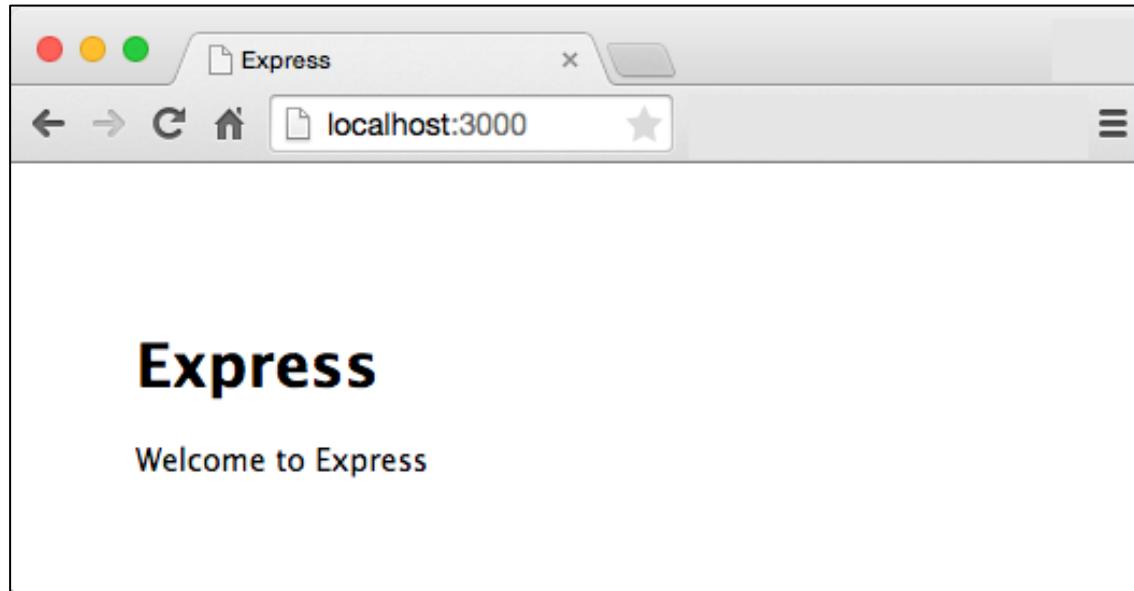
create : myApp\
create : myApp\public\
create : myApp\public\javascripts\
create : myApp\public\images\
create : myApp\public\stylesheets\
create : myApp\public\stylesheets\style.css
create : myApp\routes\
create : myApp\routes\index.js
create : myApp\routes\users.js
create : myApp\views\
create : myApp\views\error.jade
create : myApp\views\index.jade
create : myApp\views\layout.jade
create : myApp\app.js
create : myApp\package.json
create : myApp\bin\
create : myApp\bin\www

change directory:
> cd myApp

install dependencies:
> npm install
```

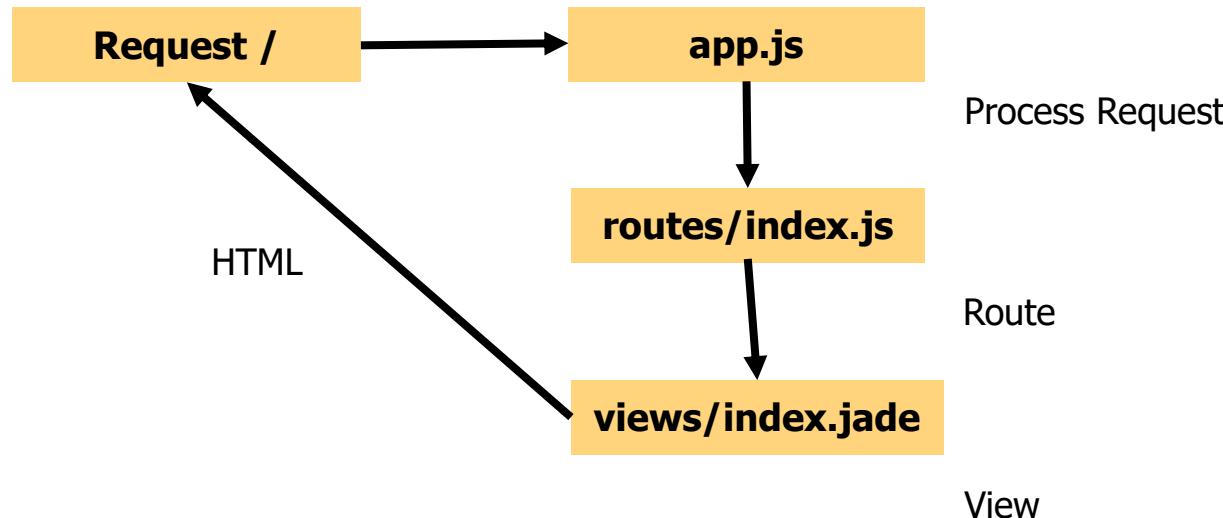
Scaffolding with Express (continued)

- Run **npm install** and **npm start** to start the website



Express Components

- Express provides processing, routing, and view files that control what is rendered to the client



The app.js File

■ Express scaffolding creates an app.js file

- Configures the views folder
- Configures a view engine
- Configures routes
- Configures “middleware”

```
// define routes
const indexRouter = require('./routes/index');
const usersRouter = require('./routes/users');
app.use('/', indexRouter);
app.use('/users', usersRouter);

// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
```

Express Route File (routes/index.js)

- Express provides route files that control the view that is rendered for a given request to the server

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

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

module.exports = router;
```

Exercise 7.2: Using the Express Module to Create a Website



30 min

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

Debugging a Node Application

Working with Node and Express

CORS (Cross-Origin Resource and Sharing)

Building RESTful Services Faster with Express

Basic Use of Jasmine with node.js and Express

Chapter Summary

What Is CORS?

- Cross-site HTTP requests are HTTP requests for resources loaded from a different domain than what was initially requested
- The HTTP request headers are as follows:
 - Origin: This defines where the actual request originates from
 - Access-Control-Request-Method: This defines the method that should be used for the actual request
 - Access-Control-Request Header: This defines the headers that should be used in the actual request

Implementing CORS

- Handling these headers according to the CORS recommendation is a rather difficult task
- There is a `cors` npm package that we can use to enable `cors` within our Express application
- It provides a piece of middleware that enables CORS within the Express application
 - Executing `npm install cors` will install the package

CORS: Whole Service or One Route Only?

- After npm installs the package, don't forget to add it to your declaration list:

```
var cors = require('cors');
```

- For application wide use just add this:

```
app.use(cors());
```

- For one Route only—add it to your route declaration

```
app.get('/', cors(), function(request, response){...
```

Chapter Concepts

Debugging a Node Application

Working with Node and Express

CORS (Cross-Origin Resource and Sharing)

Building RESTful Services Faster with Express

Basic Use of Jasmine with node.js and Express

Chapter Summary

Specifying the API

- The very first thing that has to be defined for any web API is the operations it will expose
- According to the REST principles, an operation is exposed by an HTTP method and an URI
- The action performed by each operation should not contradict the natural meaning of its HTTP method

Method	URI	Description
GET	/contacts	Retrieves all available contacts
GET	/contacts?lastname	Retrieves single contact by its last name

Choosing Data Format/Source

- The second step in our definition phase would be to choose data format for our contact application
- JSON objects are natively supported by JavaScript
- They are easy to extend during the evolution of our application and are consumable by almost any platform available
- Thus, the JSON format seems to be our logical choice here
- A sample file will be provided for the exercise

Planning the Changes

- Express 4.x has some changes in architecture; certain structure should be maintained
- Changing the PORT address
 - The port address used to be defined in `app.js`
 - However, since version 4.x, all port and address settings are made in `/bin/www` file
- Data files location
 - It's prudent to have a dedicated location for your data files
 - Creating a `data` folder is a good start
 - Place all data (`json`, `xml`, etc.) files in that location
- For routing, use the predefined `index.js` file and add your additional routes in there
- For a dedicated location for all user-defined modules, create a folder `modules`

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



45 min

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

Debugging a Node Application

Working with Node and Express

CORS (Cross-Origin Resource and Sharing)

Building RESTful Services Faster with Express

Basic Use of Jasmine with node.js and Express

Chapter Summary

Setup for Testing with Jasmine

- After creating a new node.js project, create following folders in the project directory:
 - app
 - spec
- Install dependencies
 - `npm install jasmine-node --save-dev`
- Later, we will execute several requests facing our application. We will use the request npm package for that purpose.
 - `npm install request --save`
- Let's also add Express to our project
 - `npm install express --save`

Setup (continued)

■ Add test instructions to your package.json file

- Add `./node_modules/.bin/jasmine-node --captureExceptions spec` as shown

```
"main": "index.js",
"scripts": {
  "test": "./node_modules/.bin/jasmine-node --captureExceptions spec"
},
"author": "me",
"license": "ISC",
```

The First Jasmine Test File

Create a Jasmine test file

firstnodetest_spec.js inside the spec folder

- Note: all test files need to end with spec.js !!!!!!!!

Add following code to the test file:

Add another spec (it) to test for the status code 200

- Hint: you may need response.statusCode variable to write your expect statement

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

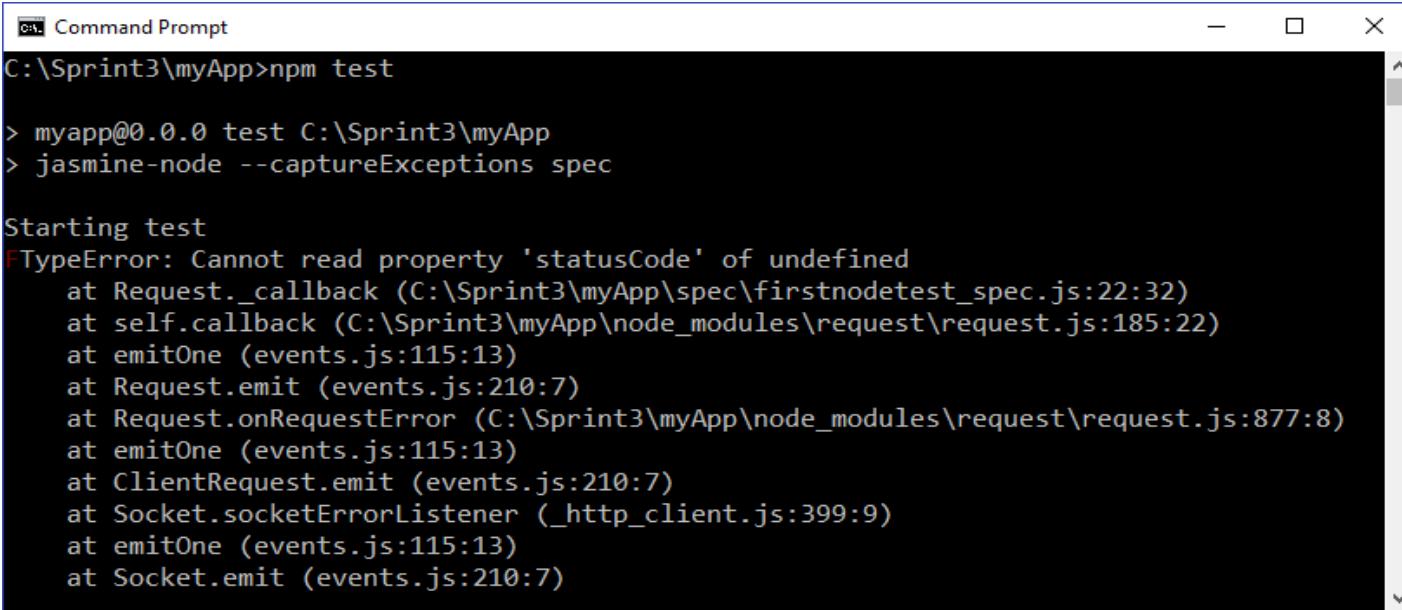
const base_url = "http://localhost:3000/";

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();
      });
    });
  });
});
```

See Your Test Fail

- Run your test to verify that it will fail when no server is started/developed yet
- Make sure you are in your project folder and enter: `npm test`
 - Your output should look like this:



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The command entered is `C:\Sprint3\myApp>npm test`. The output shows the test command being run and then a stack trace for a `TypeError` occurring in the `firstnodetest_spec.js` file at line 22:32. The error message is: "Cannot read property 'statusCode' of undefined". The stack trace details the call stack from the error point back up to the top level.

```
C:\Sprint3\myApp>npm test
> myapp@0.0.0 test C:\Sprint3\myApp
> jasmine-node --captureExceptions spec

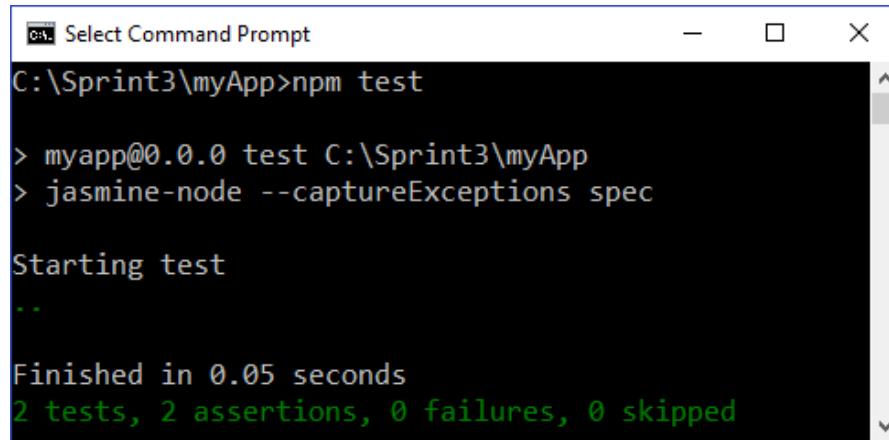
Starting test
FTypeError: Cannot read property 'statusCode' of undefined
  at Request._callback (C:\Sprint3\myApp\spec\firstnodetest_spec.js:22:32)
  at self.callback (C:\Sprint3\myApp\node_modules\request\request.js:185:22)
  at emitOne (events.js:115:13)
  at Request.emit (events.js:210:7)
  at Request.onRequestError (C:\Sprint3\myApp\node_modules\request\request.js:877:8)
  at emitOne (events.js:115:13)
  at ClientRequest.emit (events.js:210:7)
  at Socket.socketErrorListener (_http_client.js:399:9)
  at emitOne (events.js:115:13)
  at Socket.emit (events.js:210:7)
```

Now Add a Server for Testing

- Create a file called `firstnodetest.js` in the app folder
- Add the code shown at right to that file and save
- Start server from console
 - `node firstnodetest.js`
- Open a second console, navigate to the project folder, and start test
 - `npm test`
- Your output should be like this:
- **Note:** for each expect, you will get an assertions count

```
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");
});
```



The screenshot shows a terminal window titled "Select Command Prompt". The command `C:\Sprint3\myApp>npm test` was run. The output shows the test runner (jasmine-node) starting, running two tests, and completing successfully with 2 assertions, 0 failures, and 0 skipped.

```
C:\Sprint3\myApp>npm test

> myapp@0.0.0 test C:\Sprint3\myApp
> jasmine-node --captureExceptions spec

Starting test
..

Finished in 0.05 seconds
2 tests, 2 assertions, 0 failures, 0 skipped
```



Exercise 7.4: Testing Express with Jasmine

30 min

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

Debugging a Node Application

Working with Node and Express

CORS (Cross-Origin Resource and Sharing)

Building RESTful Services Faster with Express

Basic Use of Jasmine with node.js and Express

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- How to debug a node.js application
- How to utilize Express to program faster RESTful services with node.js
- How to test your RESTful services with Jasmine

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 8: Testing Node with Jasmine

Chapter Overview

In this chapter, we will explore:

- How to apply Behavior-Driven Development (BDD) using the Jasmine framework
- The basics of the Jasmine framework
 - What are:
 - Suites
 - Specs
 - Matchers
 - And others

Chapter Concepts

Automation Frameworks

End-to-End Testing

Testing JavaScript with Jasmine

Chapter Summary

JavaScript Automation Testing Frameworks

- This is an area that has exploded with interest over the past several years
- Several of the many JavaScript automation testing frameworks include:
 - PhantomJS
 - A webkit that is scriptable with JavaScript for unit-level, front-end testing
 - Jasmine
 - A behavior-driven framework for testing JavaScript code
 - Protractor
 - Automated testing framework for testing AngularJS in the browser
 - Nightwatch.js
 - Node.js-based end-to-end testing solution for browser-based apps
 - Nemo.js
 - Open-source, Node.js framework developed by PayPal

Why JavaScript for Testing?

- Open-source
- Modular
- Very active community
- Client and server can be written in JavaScript
 - So why not tests too?
- Free

Chapter Concepts

Automation Frameworks

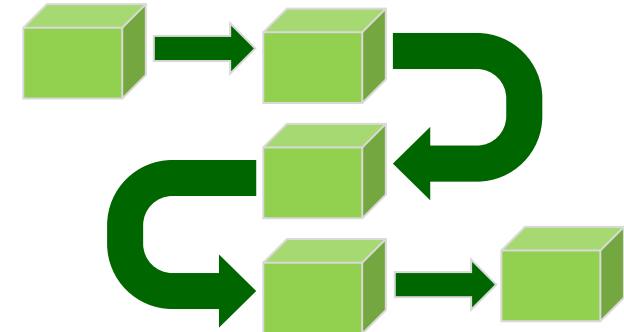
End-to-End Testing

Testing JavaScript with Jasmine

Chapter Summary

End-to-End Testing

- End-to-end testing (E2E)
 - A methodology used to test the flow of an application
 - From start to finish
- Testing the application from the user point of view
 - Treats the application as a blackbox
 - Only the user interface is exposed to the user (tester)
- Usually performed after functional and system testing



End-to-End Testing

Types of End-to-End Testing

Horizontal

- Occurs horizontally across the context of multiple applications
 - This method can easily occur in single ERP (Enterprise Resource Planning) application
 - Take an example of a web-based application of an online ordering system
 - The whole process will include accounts, inventory status of the products, as well as shipping details

Vertical

- All interactions with an application are verified
 - From start to finish
- Each layer of the application is tested from top to bottom
 - Much more thorough and much harder to implement

When to Perform End-to-End Testing

- In the delivery lifecycle, or in the hierarchy of software testing methodologies, end-to-end testing is usually conducted after functional and system testing
- While system testing and end-to-end testing seem very similar, there are a few key differences
 - System testing is used to check that each feature of your application functions as intended
 - End-to-end testing focuses on the flow through a system from a realistic end-user point of view
 - You're not only looking to see if a user can navigate your website or app, but also that all the necessary data and information is passed on correctly

Simple End-To-End Testing Scenario: ATM

- A simple test flow for an everyday task, getting money from an ATM
 - Inserting card → Reading chip, recognizing card
 - Entering PIN → Validating PIN
 - Choosing operation → Withdraw
 - Choosing amount → \$100
 - Choosing to perform no further tasks
 - Take dispensed card
 - Take dispensed money

Types of Test Cases

- Tests should be designed from the user's point of view
- Tests should focus on testing some existing features of the application
- Multiple scenarios should be considered
- Different sets of test cases should focus on different scenarios of the application

Why Do End-to-End Testing?

- Modern enterprise applications are complex
 - With multiple subsystems
- If there is a problem in any subsystem:
 - It can affect many other parts of the enterprise
- This presents a very important risk to the organization
 - Controlling this is of major importance

Designing End-to-End Testing

■ User functions

- Listing features of the software systems
- Document actions performed, input and output
- Document relationships between different actions

■ Conditions

- For each user function, document its conditions
- Timing, data conditions, etc.

■ Test cases

- For each scenario, create one or more tests cases
- Each condition should be covered in a separate test case

Chapter Concepts

Automation Frameworks

End-to-End Testing

Testing JavaScript with Jasmine

Chapter Summary

Jasmine

- Jasmine is a behavior-driven framework
 - For testing JavaScript code
- Has no dependencies on other JavaScript frameworks
- Does not require a DOM
- Clean syntax makes it easy to write tests

Test Suite – What Do We Want to Test?

- Explore the sample code from a previous exercise
- Two functions are exported, and thus, suitable for testing
 - `list()`
 - `query_by_arg()`

contacts.js

```
const fs = require('fs');
let read_json_file = () => {
    let file = './data/contact.json';
    return fs.readFileSync(file);
}
exports.list = () => {
    return JSON.parse(read_json_file());
};
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++) {
        // some code
    }
    return null;
};
```

Test Suite

- A test suite starts up with a call to describe
 - A global Jasmine function
 - Takes two parameters
 - A string and a function
- The string is a name or title for a spec suite
 - What is being tested
- The function is a block of JavaScript code
 - Implements the suite

```
let request = require("request");
let contacts = require("../modules/contacts");

describe("Unit tests on contacts module", () => {
  describe("load all contacts, test list()", () => {
    //positive test to load all contacts
    it("have two elements", () => {
      let results = contacts.list();
      expect(results.result.length).toBe(2);
    });
  });
  describe("load specific contacts, test query_by_arg()", () => {
    //positive test to load contact by last name
    it("with last name Smith", () => {
      let results = contacts.query_by_arg("lastname", "Smith");
      expect(results.firstname).toBe("Joe");
    });
  });
  .....
});
```

Specs

- Specs declare test cases belonging to a suite
 - The test is initiated by calling the Jasmine function `it()`
- The string is the title of the spec
- The function is the spec
 - The test
- A spec contains one or more **expectations**
 - Test the state of the code
- An **expectation** is an **assertion**
 - Is either true or false
- A spec with all true expectations passes
 - Otherwise, it fails

```
// the first Jasmine test suite
describe("A suite", function() {
  it("contains spec with an expectation",
    function() {
      expect(true).toBe(true);
    });
});
```

Expectations

- Expectations are built with the `expect` function
 - Which takes a value
 - Called the actual
- Chained with a `matcher` function
 - Which contains the expected value
- Each matcher compares the actual and expected values
 - To negate a matcher, chain `expect` to `not` before the matcher
- Jasmine has many matchers included
 - Custom matchers can also be defined

```
it("The 'toBeLessThan' matcher", function() {  
  var pi = 3.1415926,  
      e = 2.78;  
  
  expect(e).toBeLessThan(pi);  
  expect(pi).not.toBeLessThan(e);  
  expect(pi).toBeGreaterThan(e);  
});
```

```
it("'toThrow' matcher is for testing exceptions",  
  function() {  
    var foo = function() { return 1 + 2; };  
    var bar = function() { return a + 1; };  
    var baz = function() { throw 'what'; };  
  
    expect(foo).not.toThrow();  
    expect(bar).toThrow();  
    expect(baz).toThrow('what');  
  });
```

Expectations – Example for Throws

- Simple code example of a divide function throwing an Error object when division by zero occurs
- This time, we not only check whether a throws occurred, but whether the correct object with the correct message had been thrown

calc.js

```
exports.divide = (x, y) => {
  if (y === 0) {
    throw new Error("Can't divide by 0");
  }
  return x / y;
}
```

Run_calc.js

```
const calc = require('./calc');
console.log(calc.divide(10,2)); // 5
```

...

```
it("throws an exception if the divisor is 0", () => {
  expect(
    () => calc.divide(99, 0)
  ).toThrow(new Error("Can't divide by 0"));
});
```

...

Setup and Teardown

- Jasmine provides functions for setting up and tearing down code

- beforeEach
- afterEach
- beforeAll
- afterAll

```
describe("A spec using beforeEach and afterEach", function() {  
  var theAnswer = 42;  
  var foo = 0;  
  
  beforeEach(function() {  
    foo = theAnswer;  
  });  
  
  afterEach(function() {  
    foo = 0;  
  });  
  
  it("does foo equal theAnswer", function() {  
    expect(foo).toEqual(theAnswer);  
  });  
});
```

The `this` Object Reference

- The `this` key word is set to an empty object
 - Each spec has this set
 - For the `beforeEach` / `it` / `afterEach` cycle
- The `this` reference is set to an empty object before `beforeEach()`, `afterEach()`, or `it()` is executed
 - `this.my_shared_data = "A value to be shared"`
- The same object is shared by all the specs
 - And is set to be empty at the end of each spec

Disabling a Test Suite

- A test suite can be disabled with the `xdescribe` function

- The suite is skipped
- Specs inside it are not run
- Results show as pending

```
xdescribe("Not ready for prime time", () => {
  let value;

  beforeEach( () => {
    value = 0;
    value += 1;
  });

  it("not completed yet", () => {
    expect(value).toEqual(1);
  });
});
```

Disabling a Spec

- A spec can be disabled by using the `xit` function
 - The spec is not run
 - The results will show it as pending
- Can also use the `pending` function
 - Anywhere in a spec body
 - It will be treated as pending

```
describe("Pending specs", function() {  
  xit("Skip this one", function() {  
    expect(true).toBe(false);  
  });  
  
  it("a spec w/o a body is skipped");  
  
  it("calling 'pending' ", function() {  
    expect(true).toBe(false);  
    pending('this is why it is pending');  
  });  
});
```

Jasmine Matchers

- So far we have seen `toEqual`,
`toThrow`, `toBeLessThan`
- See a few more in the list

<code>toBe()</code>	passed if the actual value is of the same type and value as that of the expected value. It compares with <code>==</code> operator
<code>toEqual()</code>	works for simple literals and variables; should work for objects too
<code>toMatch()</code>	to check whether a value matches a string or a regular expression
<code>toBeDefined()</code>	to ensure that a property or a value is defined
<code>toBeUndefined()</code>	to ensure that a property or a value is undefined
<code>toBeNull()</code>	to ensure that a property or a value is null.
<code>toBeTruthy()</code>	to ensure that a property or a value is <code>true</code>
<code>toBeFalsy()</code>	to ensure that a property or a value is <code>false</code>
<code>toContain()</code>	to check whether a string or array contains a substring or an item.
<code>toBeLessThan()</code>	for mathematical comparisons of less than
<code>toBeGreaterThan()</code>	for mathematical comparisons of greater than
<code>toBeCloseTo()</code>	for precision math comparison
<code>toThrow()</code>	for testing if a function throws an exception
<code>toThrowError()</code>	for testing a <i>specific</i> thrown exception

Testing for Errors

- It is **vital** to test for error conditions
 - To verify the service returns the proper HTTP status code

```
// when searching for unknow contact return 404
it("returns 404", (done) => {
  request.get(base_url + 'contacts?lastname=Washington',
    (error, response, body) => {
      expect(response.statusCode).toBe(404);
      done();
    });
});
```

```
// when using wrong search key word return 500
it("returns 500 when searching for cell phone", (done) => {
  request.get(base_url + 'contacts?cellphone=%2B000001',
    (error, response, body) => {
      expect(response.statusCode).toBe(500);
      done();
    });
});
```

Testing for Errors – What Is done()?

- Asynchronous testing requires additional functionality to manage a delayed response
 - Jasmine.Async add-on library provides that functionality in form of a `done()` function
- In simple terms, the `done()` function tells Jasmine that the test has completed, however, it will timeout after 5000ms

```
// when using wrong search key word return 500
it("returns 500 when searching for cell phone", (done) => {
  request.get(base_url + 'contacts?cellphone=%2B000001',
    (error, response, body) => {
      expect(response.statusCode).toBe(500);
      done();
    });
});
```

Evaluate test when
`done()` is called

Call `done()` as last
method in async
function and only
once!



HANDS-ON
EXERCISE

Exercise 8.1: Testing with Jasmine and Node

45 min

- Follow the directions in your Exercise Manual for this exercise

Spies – Why Do We Need Them?



- Easy to test: profileService has profile member that is changed by updateProfile()

```
describe("The profile service", () => {
  let profileService;

  beforeEach( () => {
    profileService = {
      profile: null,
      updateProfile: (newProfile) => {
        this.profile = newProfile;
      }
    };
  });

  it("sets its profile property when its updateProfile method is called", () => {
    var profileValue = "New profile";
    profileService.updateProfile(profileValue);
    expect(profileService.profile).toEqual(profileValue);
  });
});
```

Spies – Why Do We Need Them? (continued)



- Now an example that's hard to test because the method being tested doesn't change the object's state
 - So, we need a spy to verify correct behavior

```
describe("Given a connection to the profile service", () => {
    var profileDao;
    var profileService;

    beforeEach(() => {
        profileDao = {
            updateProfileInDb: (newProfile) => { /*... */ }
        };
        // Hard to test: profileService has no data members
        // that are changed by updateProfile()
        profileService = {
            dao: profileDao,
            updateProfile: (newProfile) => {
                profileDao.updateProfileInDb(newProfile);
            }
        };
    });
    ... more next slide
```

Spies – Why Do We Need Them? (continued)



- We can now check whether the appropriate dependent function was called, and how many times, to verify proper process flow

```
it("When updateProfile is called, Then it calls its DAO's updateProfileInDb method",  
    () => {  
        spyOn(profileDao, 'updateProfileInDb').and.returnValue(true);  
  
        var profileValue = "New profile";  
        var result = profileService.updateProfile(profileValue);  
  
        expect(result).toEqual(true);  
        expect(profileDao.updateProfileInDb).toHaveBeenCalledWith(profileValue);  
        expect(profileDao.updateProfileInDb).toHaveBeenCalledTimes(1);  
    });  
});
```

Asynchronous Tests

- Jasmine also supports specs for asynchronous operations (see also Slide 8-27)
- Functions that take an optional single argument that will be called when the async work has completed
 - Setup and teardown functions
 - The `it` function
- The spec will not start until the `done` function is called in `beforeEach`
- The spec will not complete until its `done` function is called

```
describe("Asynchronous specs", () => {
  var value;

  beforeEach((done) => {
    setTimeout(() => {
      value = 0;
      done();
    }, 1);
  });

  it("should support async execution of test preparation and expectations", (done) => {
    value++;
    expect(value).toBeGreaterThan(0);
    done();
  });
});
```

Jasmine Docs

- There is much more to Jasmine
- For more details, documentation, and examples, visit the following websites
 - <https://jasmine.github.io/>
 - <http://angular-tips.com/blog/2014/03/introduction-to-unit-test-spies/>
 - <https://howtodoinjava.com/javascript/jasmine-unit-testing-tutorial/>

Chapter Concepts

Automation Frameworks

End-to-End Testing

Testing JavaScript with Jasmine

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- How to apply Behavior-Driven Development (BDD) using the Jasmine framework
- The basics of the Jasmine framework
 - What are:
 - Suites
 - Specs
 - Matchers
 - And others

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 9: Service Virtualization

Chapter Overview

In this chapter, we will explore:

- What service virtualization is
- What types of problems service virtualization solves
- How to use virtual services in enterprise development

Chapter Concepts

Service Virtualization

Chapter Summary

Service Virtualization

- A major development bottleneck is waiting on dependent components
- Service virtualization allows the use of virtual services instead of production services
 - Enables development before a dependent component is available
 - Emulates the behavior of a dependent component
 - Allows integration testing much sooner

Mocking vs. Service Virtualization

- Mocks are fake software components used to imitate real software components
 - Tend to be very context specific
 - Simulate a specific response to a certain request
 - Usually simulates an individual class
 - Best suited for unit tests
- Virtual services can be deployed throughout the entire production cycle
 - Consistently delivering functionality that developers and testers can use
 - Eliminate the need for individual developers or testers to write and rewrite mocks
 - Can simulate an entire network of backend services
 - Best suited for integration and performance tests
- Both approaches have value and both will often be used in application development

Use Cases for Service Virtualization

- Consuming a third-party web service such as a credit check
 - Use service virtualization to avoid paying for web service calls
- Test web service response with certain data
 - Use service virtualization to create the desired response
- Automated tests expect valid and invalid data
 - Service virtualization can provide both types of data
- Testing front-end web clients
 - Virtual services can run many tests at one time on a single server
- Enterprise application with dozens of service dependencies
 - Simulate the back-end service dependencies with virtual services

How Does Service Virtualization Work?

- Service virtualization software typically uses one of two methods to generate virtual components
- Generate virtual service code from a standard service description
 - Requires some setup and refinement
 - Each virtual service only needs to be created once
 - Then can be used repeatedly and continuously in development and testing
- Create traffic recordings of actual system interactions between your application and the dependent components
 - Potentially simpler approach
 - Does require the dependent component to be available for recording

Service Virtualization Tools

There are many software tools that support service virtualization:

- Hoverfly
- ServiceV Pro
- CA LISA
- Micro Focus Service Virtualization
- IBM Green Hat
- Tricentis TOSCA Orchestrated Service Virtualization
- Soap UI
- Parasoft Virtualize
- Traffic Parrot for Microservices

Hoverfly

- Hoverfly is an open-source API (service) simulation tool
 - Lightweight
 - High performance
 - REST API
 - Extend and customize with any programming language
- Extensive documentation is available here:
<https://hoverfly.readthedocs.io/en/latest/index.html>



HANDS-ON
EXERCISE

20 min

Exercise 9.1: Using Service Virtualization

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

Service Virtualization

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- What service virtualization is
- What types of problems service virtualization solves
- How to use virtual services in enterprise development

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 10: Apigee Edge

Chapter Overview

In this chapter, we will explore:

- What Apigee Edge is
- What types of problems Apigee Edge solves

Chapter Concepts

Apigee Edge

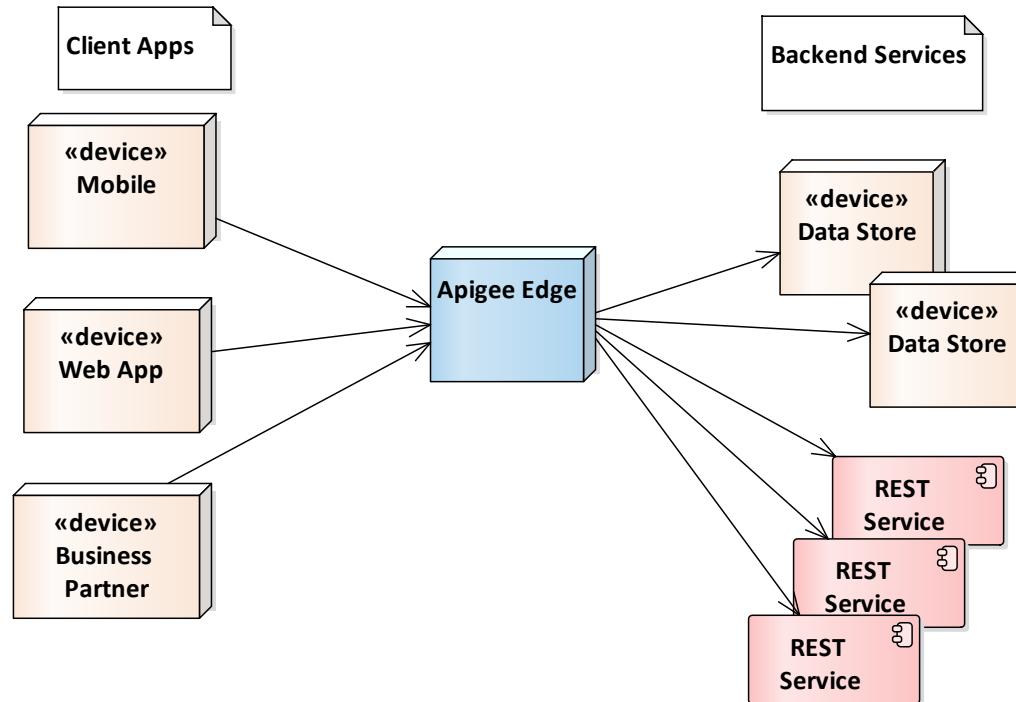
Chapter Summary

Understanding Apigee Edge

- Apigee Edge is a platform for developing and managing API proxies
- An API proxy is a proxy for a back-end service that provides other features
 - Security
 - Rate limiting
 - Quotas
 - Analytics
- Front-end developers will communicate with the API proxies in order to access the back-end services

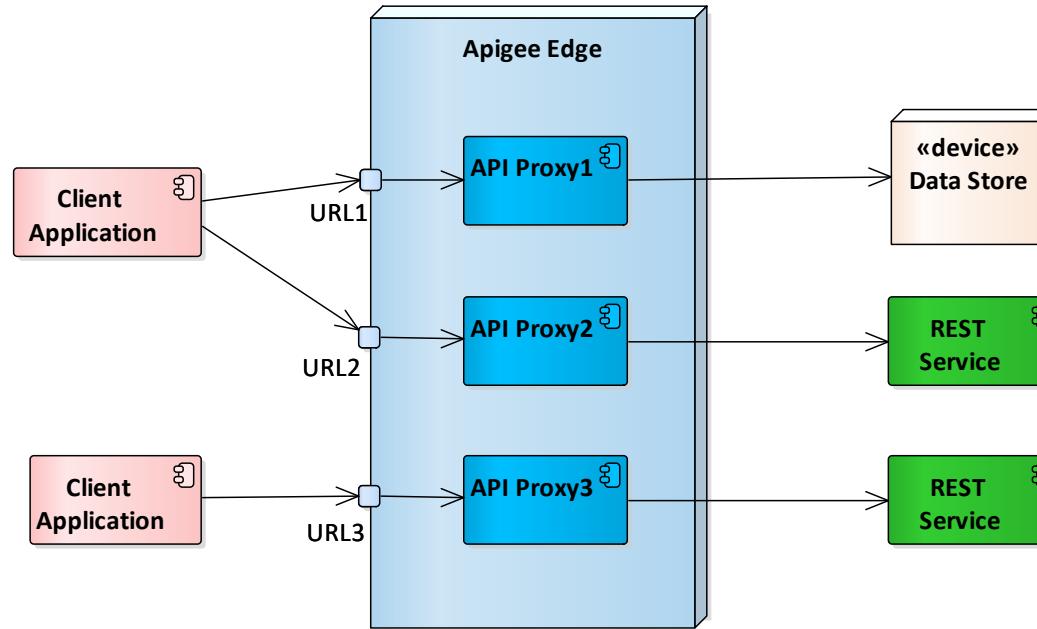
Enterprise Architecture with Apigee Edge

- This diagram illustrates how Apigee Edge can be used in an enterprise application



Client Communication

- Client applications communicate with an API proxy to access back-end resources



API Proxies

- The API proxy isolates the client developer from back-end resource details
 - Client developers do not need to know details of resource implementations
- Client developers need to know:
 - The URL of the API proxy endpoint
 - Parameters or headers passed in a request
 - Authentication and authorization credentials
 - Format of the response

API Key

- Client developers must register the application with Apigee
 - The client developer will receive an API key
- Every client request must include the API key in every request to an API proxy
- Keys can be revoked at any time
 - Clients using that key will no longer have access to the services
- Keys can also have a time limit placed on them
 - The key must be refreshed when the time expires

Controlling API Proxies

- Flows are used to control how an API proxy performs
 - Define business logic
 - Add conditional behavior
 - Determine error handling
- Flows are sequential steps that define a processing path
- Policies can be added to flows to implement various features
 - Security such as OAuth
 - Traffic management
 - Message manipulation such as returning data in XML or JSON
 - Caching data across requests
 - CORS support

API Proxy Design and Development

- Apigee provides guidelines, documentation, and examples for working with Apigee Edge
- Getting started with Apigee Edge
 - <https://docs.apigee.com/api-platform/get-started/get-started>
- Best practices for API proxy design
 - <https://docs.apigee.com/api-platform/fundamentals/best-practices-api-proxy-design-and-development>
- Apigee Edge antipatterns
 - <https://community.apigee.com/articles/44662/the-book-of-apigee-edge-antipatterns.html>

Chapter Concepts

Apigee Edge

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- What Apigee Edge is
- What types of problems Apigee Edge solves

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 11: The FidZulu Mini Project

Chapter Overview

In this chapter, we will explore:

- The FidZulu mini project

Chapter Concepts

The FidZulu Mini Project

Chapter Summary

Introduction to the FidZulu Mini Project

■ Background:

- Fidelity wants to develop a service to compete with Amazon
 - For this purpose, you will develop a Single Page Application using node and Angular only

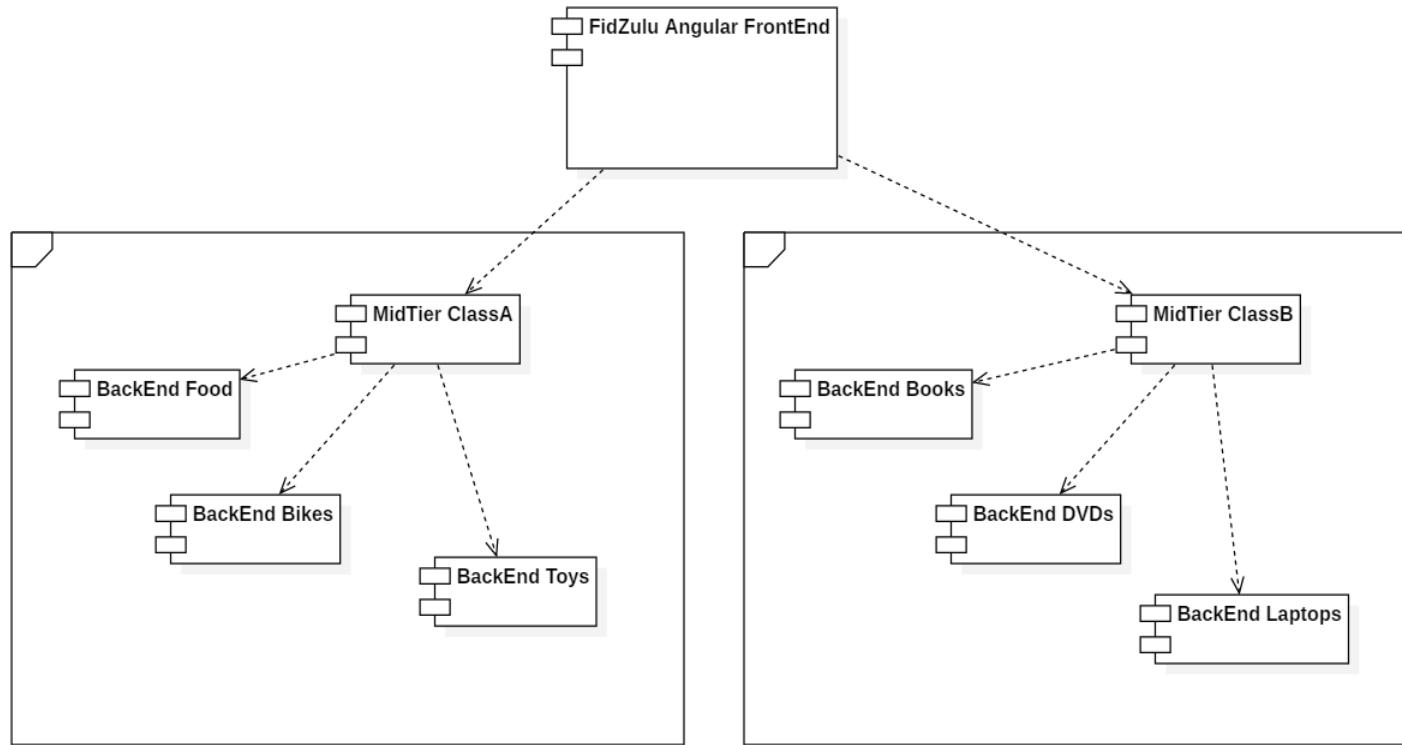
■ Scope:

- For the first phase of this project, you will develop six RESTful services using node and Express
 - Each service will be responsible for a certain data type
- There will also be a RESTful service that will manage two of the data services and provide a composite API for the underlying functionality
- For the front-end, an Angular application will be developed accessing the composite RESTful services to receive the necessary data

■ For complete details, refer to the FidZulu project document

- Your instructor will provide the location of this document

The FidZulu Architecture



Chapter Concepts

The FidZulu Mini Project

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The FidZulu mini project

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 12: Testing with Cucumber.js

Chapter Overview

In this chapter, we will explore:

- The basics of regular expressions
- Behavior-Driven Development (BDD) with Cucumber

Chapter Concepts

Acceptance Testing

Regular Expressions

Behavior-Driven Development with Cucumber

Testing RESTful Services

Chapter Summary

Acceptance Testing: Cucumber

- Cucumber is a tool for running automated **acceptance tests** written in a behavior-driven development (BDD) style
- Cucumber was originally written for the Ruby programming language
 - Written by Aslak Hellesoy
- It introduced **Gherkin** language
 - 'Given-When-Then'
- Ever since, Cucumber has been ported to Java, .NET, and other languages
 - Google Cucumber and the language you need—someone has probably implemented it!
- Gherkin parser implements feature file written in business-facing text

Example: Valid ATM Card

- Feature: As a bank customer, I want to withdraw cash from my account at an ATM, so I can make purchases
- Scenario:

Given My ATM card is valid
And The ATM dispenser contains cash
And My account has funds

When I withdraw cash

Then My account is debited
And My cash is dispensed
And My card is returned

Example: CEO of Fidelity

- We can also use parameters to make our test more flexible

Feature: As a Fidelity team member, I want to be able to search the internet, so I can get fast, accurate answers to questions to help me complete tasks quickly

Scenario: Google search for CEO of Fidelity

Given I have an internet connection available

When I search Google for "CEO of Fidelity Investments"

Then I should see "Abigail Johnson" in the result

- By placing parameters in quotes, we can make them available to the testing suite



HANDS-ON
EXERCISE

20 min

Exercise 12.1: Writing an Acceptance Test with Gherkin Syntax

- Write a feature file for making a purchase at Amazon
- Follow the directions in your Exercise Manual for this exercise
- Discuss with instructor your approach before you get started

Chapter Concepts

Acceptance Testing

Regular Expressions

Behavior-Driven Development with Cucumber

Testing RESTful Services

Chapter Summary

Regular Expression

- In order to access those parameters used in the feature file, we need to know about regular expressions
- It clearly goes beyond the scope of this course to cover to a full-extend regular expression, but a simple introduction should suffice
- What is a regular expression?
 - A regular expression (RE) is a sequence of characters, meta-characters, operators, and precedence rules which describe a set of strings
- Many webpages are available to be able to explore REs
 - One useful one is <https://regex101.com/> to help understand

Cucumber Expression

- There is another way to access Gherkin strings and parameters called Cucumber Expression (CE)
- CEs are simpler and easier to learn
- However, a lot of feature files in Fidelity are still written with REs, thus, we need to know about them too
- How do CEs work? (See the JavaScript example below)

```
Given('I can connect to the service', () => { ... }); // No RE required
```

```
When('I make a request for all {string}', (search_str) => { ... }); // captures a string arg
```

```
Then('I should receive {int} contacts', (contact_count) => { ... }); // No conversion to number required
```

Basic Regular Expressions

Regular Expression	Set of Strings			
a	a			
ab	ab			
a [abc]	aa ab ac			
a [a-c]	aa ab ac			
a [] a-c]	a] aa ab ac			
a [] a-c-]	a] aa ab ac a-			
a [^a-z]	a followed by not a through z			
a [a^c]	aa a^ ac			

Basic Regular Expressions (continued)

Regular Expression	Set of Strings
a[[:alnum:]]q	a followed by any alphabetic or numeric followed by q
a.	a followed by any character
a\.	a.
a\\	a\

Basic Regular Expressions (continued)

Regular Expression	Set of Strings
a*	zero or more of the letter a
aa*	a followed by zero or more of letter a
. *	The * repeats any character zero or more times
[a-c] *	This matches any string of characters zero or more a, b, or c in any order

Do Now



- For each set of strings, write an RE which matches the set of strings

1. sherlock

2. Any string of at least three characters

3. 1.00 2.00 3.00 4.00 5.00 6.00

Do Now (continued)



4. The string "bob" followed by any number of lowercase characters

5. The string .\-[]

Anchors: Definition

- In UNIX, regular expressions are always compared against lines
- A line is a sequence of zero or more characters terminated, but not including, a new line
- Anchors position regular expressions within the line

Anchors: Definition (continued)

Anchor	Explanation
<code>^a</code>	a first on line
<code>a\$</code>	a last on line
<code>^a\$</code>	Beginning of line, a end of line (<code>a</code> is the only thing on the line)
<code>\<a</code>	a at beginning of word
<code>a\></code>	a at end of word
<code>\<a\></code>	The word a

 **Note:** that `\<` and `\>` are not supported by JavaScript

Anchors: Definition (continued)

Anchor	Explanation
\ba	The \b matches the empty string on the edge of a word followed by a
\Ba\B	The \B matches the empty string NOT at the edge of a word followed by a (the a is inside a word)

Reading Regular Expressions

- First, look at the first symbol in the RE
 - If it is an anchor, move on to the next character
 - If there is no next character, you are done
 - Expand the list of characters with these characters
 - If this is the first character, create a list of characters
- Second, look at the second character
 - If it modifies the first character, expand the list as directed by the second character
 - If it does not modify the first character, go to Step 1 with this symbol as the first symbol

Reading Regular Expressions (continued)

Regular Expression	Explanation
[ab] [xyz]	a ax ay az b bx by bz

Do Now



■ What will each of the following regular expressions check for?

1. ab^*c

2. The

3. testing

4. $\bb{[ou]}y\bb$

Creating Regular Expressions

1. Write down the list of the strings to be matched
2. If there are anchors, write them down and start with a character before or after the anchor
3. Look at the first character of each string to be matched
 - Write down an RE to describe this letter
4. Look at the next few letters
 - Are they repeats of the last character written down?
 - If so, add an RE to specify these characters
5. Go to Step 3 and use the next character as the first character



Do Now: Regular Expressions

- Write a regular expression that will match a floating-point number without a leading sign
-

- Write a regular expression which can handle a telephone number with format (999) 999 9999
-

- Check your regular expression at <https://regex101.com/>

Chapter Concepts

Acceptance Testing

Regular Expressions

Behavior-Driven Development with Cucumber

Testing RESTful Services

Chapter Summary

BDD: Putting It All Together

- Let's try to make this all work in the real world
- At first, we create a BDD project to test whether a search in Google lets us find the CEO of Fidelity
- We need following steps to run our test:
 - Create the necessary directory structure
 - Create and write your feature file
 - Install necessary node libraries (cucumber.js related files)
 - Write your xxxx-steps.js file and implement functionality for Gherkin language



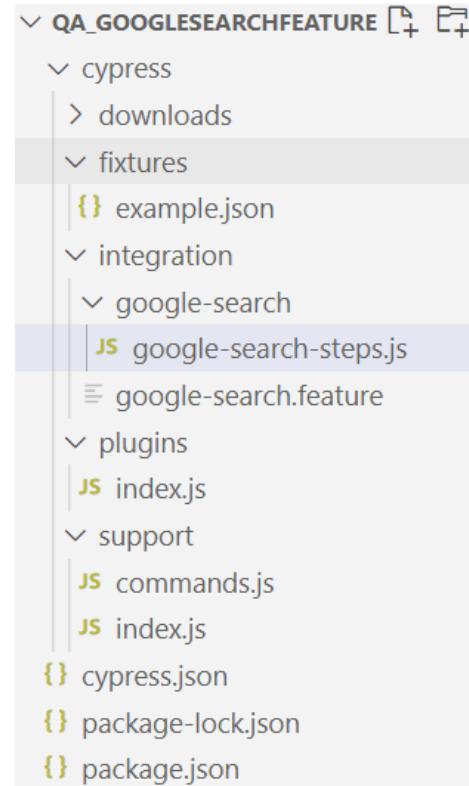
Exercise 12.2: Test Your Calculation

20 min

- Explore the steps file and feature file in this example
- Make the tests work for the calculator project
- Follow the directions in your Exercise Manual for this exercise

Typical Structure for Cucumber Project with Cypress

- The `integration` folder is used to store all feature files processed during tests
- For every `feature` file, a folder with the same name must be created to store all related `steps.js` files
 - See at right: `google-search`
- The `plugins` folder is the location where further configurations can be added
- The `<featureFileName>` folder is where the testing code goes (as mentioned above)
 - The file has to end with `steps.js`
- `Cypress.json` contains Cypress-specific configuration (e.g., recognizing feature files as test files)



The Feature File

- Create a file that ends with
.feature
- Visual Studio Code has an extension
which is called:
Cucumber (Gherkin) Full
Support
- Enter the code similar to the
displayed one
- Note: you don't have to repeat the
GIVEN if it already covers the
required preparations for another
Scenario

```
#Search.feature
Feature: Google Search Testing
As a browser user,I should be able to go to a website
and verify its search functionality,
so that I can trust its functionality

Scenario: Google search for CEO of Fidelity
Given I have internet available
When I search Google for "CEO of Fidelity Investments"
Then I should see "Abigail Johnson" in the result

Scenario: I evaluate search for product of 7*6
When I enter "7*6" in input
Then I get "42" in the result
```

The Cucumber Related Libraries

- Depending on what is tested, different sets of node modules need to be loaded
- The following package.json file will be provided for the exercise

```
"scripts": {  
    "test": "./node_modules/.bin/cypress open"  
},  
"author": "ROI training",  
"license": "ISC",  
"devDependencies": {  
    "chai": "^4.2.0",  
    "cucumber": "^5.1.0",  
    "cypress": "^9.2.1",  
    "cypress-cucumber-preprocessor": "^4.3.1"  
},  
"dependencies": {},  
"cypress-cucumber-preprocessor": {  
    "nonGlobalStepDefinitions": true  
}
```

Coding Your Steps File

- First, load the necessary libraries
 - InternetAvailable is a module to easily determine whether the internet can be accessed

```
import { Given, When, Then } from "cypress-cucumber-preprocessor/steps";
const url = 'https://www.google.com';
```

- GIVEN

```
Given('I have internet available', () => {
  cy.request(url).its('status').should('equal', 200);
  // if that succeeds access google page
  cy.visit(url);
})
```

Coding Your Steps File (continued)

■ WHEN

Using Cucumber Expressions rather than REs

```
When('I search Google for {string}', (text) => {
    cy.get('input[name="q"]').type(text + '{enter}', { log: false }).should("have.value", text);
})
```

Typing your search text into Google input field

■ THEN

```
Then('I should see {string} in the result', (text) => {
    cy.get('a').should('contain', text);
});
```



20 min

Demo 12.3: Who Is the CEO?

- This an instructor-led demonstration
- You can explore the code for this exercise, however, due to fast changing versions for chrome browser, your version might not execute properly
- Refer to your Exercise Manual for demo details

Chapter Concepts

Acceptance Testing

Regular Expressions

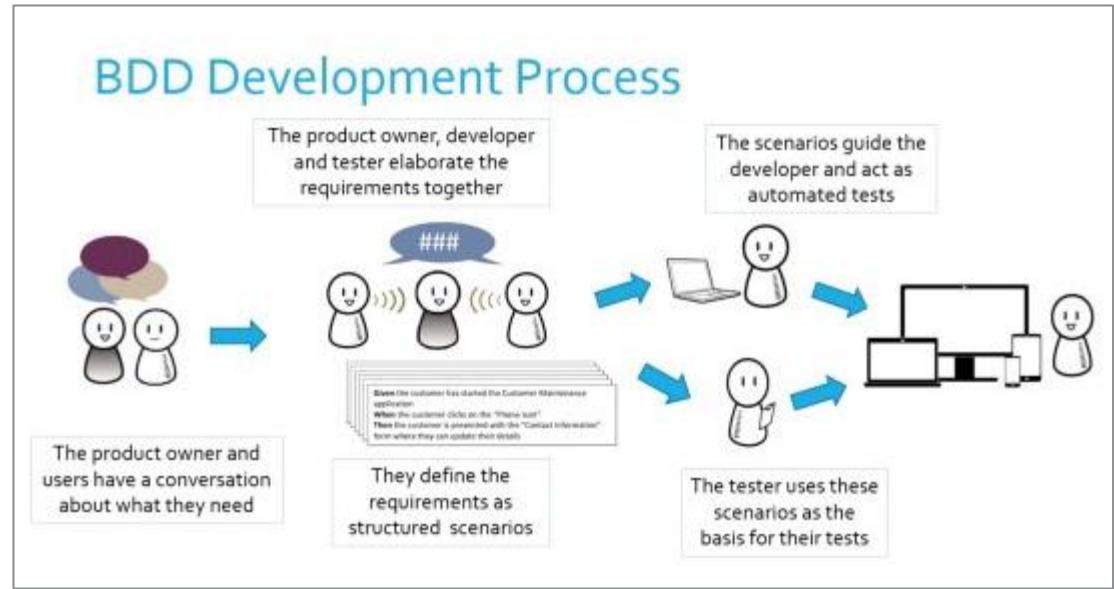
Behavior-Driven Development with Cucumber

Testing RESTful Services

Chapter Summary

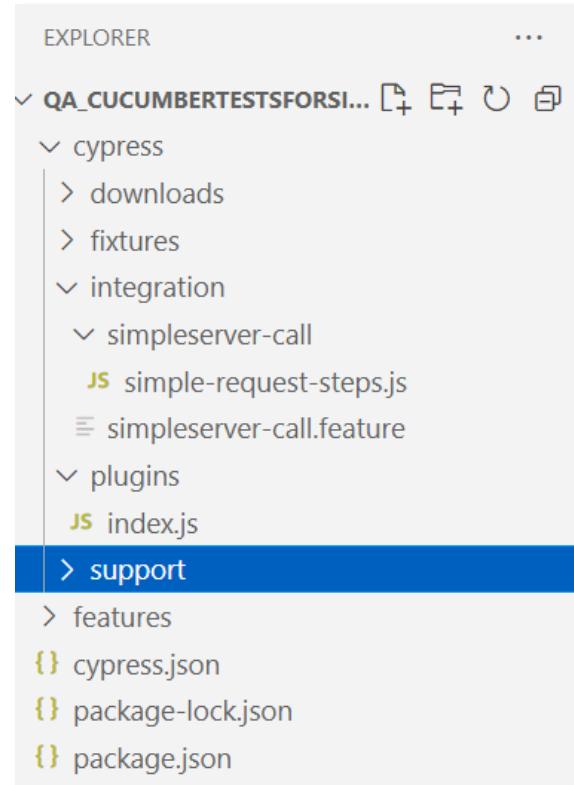
BDD: How to Test RESTful Services

- The previous example needed selenium support (or any similar web driver) to handle browser actions
- RESTful Services don't have a web interface, so how does our scenario differ?
- Thus, we need a slightly different set of node modules, but the principles remain the same



Simplified Structure for this Cucumber Project

- The integration folder is used to store all feature files processed in the project
- Inside the integration folder, create a <featurename> support folder for all steps.js files



The Feature File

- Create a file that ends with
.feature
- Visual Studio Code has an
extension which is called:
Cucumber (Gherkin) Full
Support
- Enter the code similar to the
displayed one
- Note: In this case, you do repeat
the same **GIVEN**; it will execute the
same implementation, matching the
text

```
#Search.feature
Feature: Accessing SimpleService Server
    As Software Engineer,I should be to access SimpleService
    and retrieve all contacts,
    so that I contact every person

Scenario: Getting contacts from SimpleService
    Given I can connect to the service
    When I make a request for all "contacts"
    Then I should receive "2" contacts

Scenario: Getting single contact from SimpleService
    Given I can connect to the service
```

The Cucumber Related Libraries

- Depending on what is tested, different sets of node modules need to be loaded
- The following package.json file will be provided for this exercise

```
  "scripts": {  
    "test": "./node_modules/.bin/cypress open"  
  },  
  "author": "ROI training",  
  "license": "ISC",  
  "devDependencies": {  
    "chai": "^4.2.0",  
    "cucumber": "^5.1.0",  
    "cypress": "^9.2.1",  
    "cypress-cucumber-preprocessor": "^4.3.1"  
  },  
  "dependencies": {},  
  "cypress-cucumber-preprocessor": {  
    "nonGlobalStepDefinitions": true  
  }  
}
```

Coding Your Steps File

- First, load the necessary libraries
 - We have a different set of modules here
 - This will impact the format of our implementation

```
import { expect } from "chai";
import { Given, When, Then } from "cypress-cucumber-preprocessor/steps";

let result;
let testName;
const url = 'http://localhost:8081/';
```

GIVEN

```
// -----
// Scenario 1
// -----
Given('I can connect to the service', () => {

    cy.request('GET', url).its('status').should('equal', 200);

    // Given is better used for authentication or other configuration
})
```

Coding Your Steps File (continued)

■ WHEN

```
When(`I make a request for all {string}`, (text) => {
    cy.request('http://localhost:8081/' + text).should((res) => {
        result = res.body;
        //extract body
    });
})
```

■ THEN

Using Regular Expressions

```
Then(/^I should receive "([\d+])" contacts$/, (value) => {
    expect(result.result.length).to.equal(Number(value));
    // have to use the "expect" assertion here, since "should" only works with "cy" promises
});
```

■ Note: / replaces quotes and \$ represents end of line



Exercise 12.4: Let's Test SimpleService BDD Style

30 min

- Program a Cucumber project to test SimpleService
- Follow the directions in your Exercise Manual for this exercise

Chapter Concepts

Acceptance Testing

Regular Expressions

Behavior-Driven Development with Cucumber

Testing RESTful Services

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- The basics of regular expressions
- Behavior-Driven Development (BDD) with Cucumber

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 13: Server-Side JavaScript Programming

Chapter Overview

In this chapter, we will explore JavaScript technologies:

- Function techniques
- Prototypes
- Factories
- Closures
- Iterators and Generators
- Using Grunt

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

Understanding 'this'

- In JavaScript, the thing called **this** is the object reference that “owns” the executing JavaScript code
- When used in a function:
 - **this** is the object that “owns” the function
- When used in an object:
 - **this** is the object itself
- When used in an object constructor:
 - **this** does not have a value until the time the object is created
 - At the time of creating the object, **this** is a reference to the object
- When an object constructor is used to create an object:
 - **this** is the newly created object

Using call(), apply(), and bind()

- JavaScript functions have methods
 - Including `call()`, `apply()`, and `bind()`
- The `call()` and `apply()` methods can be invoked immediately
- However, the `bind()` method returns a bound function that, when executed later, will have a chosen context for ("this") when calling the original function

Using call()

- Either `call()` or `Function.prototype.call()` can be used
- Allows us to explain what ("this") is in a function without context for ("this")
- The first parameter in the `call` method provides the context for ("this")
- Note:** `call()` cannot be used with arrow functions, since this is bound to scope at initialization

```
//Demo with javascript .call()
let obj = { name: "Fidelity" };
let greeting = function (a, b, c) {
    return "Welcome at " + this.name + " where " + a + " is " +
    b + " and " + c;
};
console.log(greeting.call(obj, "working", "fun", "rewarding"));
// returns output Welcome at Fidelity where working is fun and rewarding
```

Using apply()

- Either `apply()` or `Function.prototype.apply()` can be used
- Similar to `call()`, `apply()` allows to identify ("this") is in a function without context for ("this")
- However, in this case we have two parameters, first the bound object, second a parameter list, which must match the function called
- Note:** `apply()` cannot be used with arrow functions, since this is bound to scope at initialization

```
//Demo with javascript .apply()
let obj = { name: "Fidelity" };
let greeting = function (a, b, c) {
    return "Welcome at " + this.name + " where " + a + " is " +
    b + " and " + c;
};
let args = ["working", "fun", "rewarding"];
console.log(greeting.apply(obj, args));
// returns output Welcome at Fidelity where working is fun and rewarding
```

Using bind()

- In this case, we are binding the object without invoking the function, thus, we get an altered function back
 - Now we can call the new function to get the correct response
- Note:** `bind()` cannot be used with arrow functions, since this is bound to scope at initialization

```
//Demo with javascript .call()
let obj = { name: "Fidelity" };
let greeting = function (a, b, c) {
    return "Welcome at " + this.name + " where " + a + " is " +
    b + " and " + c;
};
let boundFunctionRef = greeting.bind(obj);
console.dir(boundFunctionRef); //returns [Function: bound greeting]
console.log(boundFunctionRef("working", "fun", "rewarding"));
// returns output Welcome at Fidelity where working is fun and rewarding
```

Immediately Invoked Functions

- JavaScript supports Immediately Invoked Function Expressions (IIFE)
 - Pronounced “iffy”

- A function definition is the “normal” way of creating a named function

```
function normalNamedFunction() { /*...*/ }
```

- You can assign a function expression to a variable or property

```
var varFunc = function () { /* ... */ };
```

- If we want to evaluate the function right away (like immediately):
 - Just add the parentheses at the end

```
(function () {/* ... */})();
```

- Can pass arguments too

```
var foo = "bar";
(function (innerFoo) {
  console.log(innerFoo);
})(foo)
```

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

Object Literal

- As you already know, an object literal is a fast way to create an object with defined values
 - *Note:* Object literals cannot be further instantiated, only use them for singletons!

```
var cell = {  
    name : "LG",  
    model : "Stylo",  
    weight : 144.6,  
    color : "silver",  
    fullname : function() {  
        return this.name + " " + this.model;  
    }  
};
```

- Values can still be changed later, but no other objects of that type can be created

```
cell.name = "Samsung";  
cell.model = "Galaxy S7";
```

Function Literal (or Function Expression)

- A function literal is very similar to a constructor function, but is unnamed
 - It allows parameters and multiple instances

```
var cell = function (name, model, weight, color) {  
    var name = name; // made name private  
    var model = model; // made model private  
    this.weight = weight; // these are public  
    this.color = color;  
    this.changeColor = function(color) {  
        this.color = color;  
    }  
}  
  
var myLG = new cell("LG", "Stylo", 144.6, "silver");  
myLG.changeColor("red");
```

- However, function expressions cannot be hoisted
 - They only exist when code is reached
- What is hoisting?

Hoisting

- Hoisting means that I can call a function before it is defined in the code sequence

```
// Output: "Hello!"  
functionTwo();  
  
function functionTwo() {  
    console.log("Hello!");  
};
```

```
// TypeError: undefined is not a function  
functionOne();  
  
var functionOne = function() {  
    console.log("Hello!");  
};
```

Class Definitions

- New to ECMAScript 6 are classes and inheritance
 - Similarities to TypeScript should make adaptation easier

Before ECMAScript 6

```
var Shape = function (id, x, y) {
    this.id = id;
    this.move(x, y);
};

Shape.prototype.move = function (x, y) {
    this.x = x;
    this.y = y;
};
```

```
class Shape {
    constructor(id, x, y) {
        this.id = id;
        this.move(x, y);
    }
    move(x, y) {
        this.x = x;
        this.y = y;
    }
}
```

Inheritance

Before ECMAScript 6

```
var Rectangle = function (id, x, y, width, height) {
    Shape.call(this, id, x, y);
    this.width = width;
    this.height = height;
};
Rectangle.prototype = Object.create(Shape.prototype);
Rectangle.prototype.constructor = Rectangle;
var Circle = function (id, x, y, radius) {
    Shape.call(this, id, x, y);
    this.radius = radius;
};
Circle.prototype = Object.create(Shape.prototype);
Circle.prototype.constructor = Circle;
```

```
class Rectangle extends Shape {
    constructor(id, x, y, width, height) {
        super(id, x, y);
        this.width = width;
        this.height = height;
    }
}
class Circle extends Shape {
    constructor(id, x, y, radius) {
        super(id, x, y);
        this.radius = radius;
    }
}
```

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

What Is a Factory Function?

- The purpose of the object factory is to create objects
- Usually implemented in a class or a static method of a class
 - Can produce repeatedly similar objects
 - Provides a way to users of the factory to create objects without knowing the specific type (class) at compile time
- Objects created by the factory method are by design inheriting from the same parent object
 - However, there are specific subclasses implementing specialized functionality
 - Sometimes the common parent is the same class that contains the factory method
- Though it is possible to use some ECMAScript 6 features, most of the technology used here was available before

What Should the Factory Do for Us?

- We want to have a method that accepts a type given as a string at runtime and then creates and returns specific objects of that type
- We don't want to use a constructor ("new" statement) to create these objects
 - Just a function that creates objects

```
var corolla = CarMaker.factory('Compact');
var solstice = CarMaker.factory('Convertible');
var cherokee = CarMaker.factory('SUV');

console.log(corolla.drive() ); // Vroom, I have 4 doors
console.log(solstice.drive() ); // Vroom, I have 2 doors
console.log(cherokee.drive() ); // Vroom, I have 17 doors
```

Let's Build Cars

- We create a common parent *CarMaker* constructor
- Then we add a static method of the *CarMaker* called *factory()*, which creates car objects
- And also add a prototype.drive function to provide that feature to all cars

```
// parent constructor
function CarMaker() {}

// a method of the parent
CarMaker.prototype.drive = function () {
    return "Vroom, I have " + this.doors + " doors";
};
```

Courtesy: JavaScript Patterns by Stoyan Stefanov

Let's Build Cars (continued)

- Now we need the specific car models

```
CarMaker.Compact = function() {  
    this.doors = 2;  
};  
  
CarMaker.Convertible = function() {  
    this.doors = 4;  
};  
  
CarMaker.SUV = function() {  
    this.doors = 17;  
};
```

Let's Build Cars (continued)

```
// the static factory method
CarMaker.factory = function (type) {
    var newcar;

    // error if the constructor does not exist
    if (typeof CarMaker[type] !== "function") {
        throw {
            name: "Error",
            message: constr + " doesn't exist"
        };
    }
    // at this point the constructor is known to exist
    // let's have it inherit the parent but only once
    if (typeof CarMaker[type].prototype.drive !== "function") {
        CarMaker[type].prototype = new CarMaker();
    }
    // create a new instance
    newcar = new CarMaker[type]();
    // optionally call some methods and then return.....
    return newcar;
};
```

Let's Build Cars (continued)

- We now have a method that accepts a type given as a string at runtime and then creates and returns objects of that type
- There is no constructor used with new
 - Just a function that creates objects

```
var corolla = CarMaker.factory('Compact');
var solstice = CarMaker.factory('Convertible');
var cherokee = CarMaker.factory('SUV');

console.log(corolla.drive() ); // Vroom, I have 4 doors
console.log(solstice.drive() ); // Vroom, I have 2 doors
console.log(cherokee.drive() ); // Vroom, I have 17 doors
```



HANDS-ON
EXERCISE

30 min

Exercise 13.1: Factories

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

Understanding Scope

- In JavaScript, scope is the set of variables that the code currently has access to
 - The set of variables, objects, and functions that can be accessed
- JavaScript has lexical scoping
 - With function scope
 - Even though it looks like it should have block scope ({ ... })
- A new scope is created only when a new function is created
- Nested functions
 - The inner function has access to the outer function scope
 - Known as “lexical scope”
 - Aka “closure”

Nested Functions

- Nested functions have access to outer function scope

```
var GangOfFour = function () {  
    var amigo1 = "Grady";  
    this.scope1 = function () {  
        console.log("The first amigo:" + amigo1);  
        this.scope2 = function () {  
            var amigo2 = "Ivar";  
            console.log("Two amigos: " + amigo1 + " " + amigo2);  
            this.scope3 = function () {  
                var amigo3 = "James";  
                console.log("Three egos: " + amigo1 + " " + amigo2 + " " + amigo3);  
            };  
            return this;  
        };  
        return this;  
    };  
};
```

```
var amigo = new GangOfFour();  
amigo.scope1().scope2().scope3();
```

Block Scope with ECMAScript 6

- Block Scoping was introduced 2015 with ECMAScript 6

```
let callbacks = [];
for (let i = 0; i <= 2; i++) {
    callbacks[i] = () => i * 2;
}
console.log(callbacks[0]() === 0); //true
console.log(callbacks[1]() === 2); //true
console.log(callbacks[2]() === 4); //true
```

callbacks
array is global

"i" only exists
within block

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

Closures Step by Step

- A closure is a function having access to the parent scope
 - Even after the parent function has closed
- Let's go step by step to understand why we need closures

The Scope Problem

- We already know private variables and global variables

```
function myFunction() {  
    var a = 4;  
    return a * a;  
}
```

```
var a = 4;  
function myFunction() {  
    return a * a;  
}
```

- However, the lifetime of these variables is very different due to their nature
 - Private variables live within the function they are declared in; whenever the function is called, the variable is created
 - Global variables live as long as your window/web page does
 - A variable that is created **without** the **var** key word is ALWAYS GLOBAL

The Counter Problem

No privacy!

```
var counter = 0;

function add() {
    counter += 1;
}

add();
add();
add();

// the counter is now equal to 3
```

Too private

```
function add() {
    var counter = 0;
    counter += 1;
}

add();
add();
add();

// want the counter to be 3
// but it does not work
```

Correct result, but everyone can overwrite counter, without add()

Every time we call add(), the counter is newly created and set to 1

Nested Functions

```
function add() {  
    var counter = 0;  
    function plus() {counter += 1;}  
    plus();  
    return counter;  
}
```

- This could work, if we could reach the plus function
 - So far, we can only reach that function if we create an object from add
 - But that is NOT what we are looking for
- If only we could find a way to execute **var counter = 0** only once

Closures

- Remember IIFE functions and what they do?
- What is happening here?
- The first `add` (declared with `var`) invokes the entire function
 - `add` receives the inner function as return value
- When you call `add()` you actually only invoke the inner function without recreating the counter variable
- This is a **closure** which allows functions to have private variables

```
var add = (function () {  
  var counter = 0;  
  return function () {  
    return counter += 1;  
  }  
})();  
  
console.log("counter = " + add());  
console.log("counter = " + add());  
console.log("counter = " + add());
```

Objects vs. Closures

Objects	Closures
Can add functions later; flexibility	Cannot add function; safety
Can use function from other source; Reuse of code!	Has to create function always from scratch; More memory used; Possible redundancy
Need to be careful when a method gets detached from an object, this will get a different meaning; Complexity!	You don't need to keep track of this
WHEN TO USE	
Less concern with privacy, many instances of an object required	High privacy requirements and few "objects" of that type are needed



HANDS-ON
EXERCISE

20 min

Exercise 13.2: Exploring Closures

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

Processing Collections

- Processing a collection of items is very common
- JavaScript supports several ways of iterating over a collection
 - Simple for loops
 - Iterators and generators
 - Provide a mechanism for customizing the behavior of `for...of` loops

Iterators

- An iterator knows how to access the items in a collection
 - One at a time
 - Keeps track of its current position in the collection

- JavaScript iterators

- Provide a `next()` function
 - Returns the next item in the collection
 - The object returned has two properties
 - `done`
 - `value`

```
function makeAnIterator(array) {  
    var nextIndex = 0;  
  
    return {  
        next: function() {  
            return nextIndex < array.length ?  
                {value: array[nextIndex++], done: false}  
                : {done: true};  
        }  
    };  
}
```

```
var it = makeAnIterator(['Grady', 'Ivar', 'James']);  
console.log(it.next().value); // 'Grady'  
console.log(it.next().value); // 'Ivar'  
console.log(it.next().done); // false
```

Iterables

- An iterable in ES6 is an object that defines its iterator
- The `for...of` loop can loop over any iterable
- You can create your own iterables
 - Define a function on the object names
`@@iterator`
 - Or use `Symbol.iterator` as the function name
- Since JavaScript does not have interfaces:
 - Iterable is a convention
- JavaScript does provide some built-in iterables
 - String
 - Array
 - Map
 - Set

```
let iterableUser = {  
  name: 'Grady',  
  lastName: 'Booch',  
  [Symbol.iterator]: function* () {  
    yield this.name;  
    yield this.lastName;  
  }  
}  
  
// logs 'Grady' and 'Booch'  
for(let item of iterableUser) {  
  console.log(item);  
}
```

Generators

- A generator is a special function
 - Allows you to write an algorithm that maintains its own state
 - And can be paused and resumed
- A generator is a factory for iterators
- A generator function is marked with an *
 - And contains at least one yield statement

```
function* generateRandomNumbers(){  
  let start = 1;  
  let end = 42;  
  while(true)  
    yield Math.floor((Math.random() * end) + start);  
}
```

```
//no execution here  
//just getting a generator  
let sequence = generateRandomNumbers();  
  
for(let i=0;i<5;i++){  
  console.log(sequence.next());  
}
```

Advanced Generators

- Generators compute their yielded values on demand
 - Efficiently represent a sequence of values that are expensive to compute
 - Or even an infinite sequence!
- The `next()` function accepts an input argument
 - Can be used to modify the internal state of the generator
 - Will be used as the result of the last `yield` expression of the generator
- Note: generators do NOT like recursion!

```
function* fibonacci() {  
  var fn1 = 1;  
  var fn2 = 1;  
  while (true) {  
    var current = fn1;  
    fn1 = fn2;  
    fn2 = current + fn1;  
    var reset = yield current;  
    if (reset) {  
      fn1 = 1;  
      fn2 = 1;  
    }  
  }}}
```

```
var sequence = fibonacci();  
console.log(sequence.next().value); // 1  
console.log(sequence.next().value); // 1  
console.log(sequence.next().value); // 2  
console.log(sequence.next().value); // 3  
console.log(sequence.next(true).value); // 1  
console.log(sequence.next().value); // 1  
console.log(sequence.next().value); // 2  
console.log(sequence.next().value); // 3
```



HANDS-ON
EXERCISE

20 min

Optional Exercise 13.3: Iterators and Generators

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

Function Techniques

JavaScript Classes and Inheritance

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

What Is Grunt?

- Grunt is a task-based command line build tool for managing JavaScript projects
- What kind of tasks?
 - Running tests
 - Minifying code
 - Running JSHint on your code
- Grunt is built on Node.js
 - Available via the Node package manager (npm)
 - Installs a bunch of dependencies

```
npm install -g grunt
```

Using Grunt

- Grunt needs a grunt.js project file to run
- The 'grunt init' command will create the grunt.js file
 - Specify what type of project type you want
 - jquery: A jQuery plugin
 - node: A Node module
 - commonjs: A CommonJS module
 - gruntplugin: A Grunt plug-in
 - gruntfile: A Gruntfile (grunt.js)
- You will need to provide some values
 - Project name
 - Project title
 - Etc.

Some Grunt Commands

■ Some of the things Grunt will do for you

- grunt lint – checks your JavaScript using JSHint
- grunt qunit – runs your Qunit tests
- grunt concat – concatenates project files together and places the new file in the dist folder
- grunt min – minifies the file produced by the concat command

Customizing Grunt

- You can customize Grunt
 - Edit the grunt.js file
- Properties of the config object passed to Grunt
 - pkg – points to the package.json file which stores project metadata
 - meta – object with only a banner property which is placed at the top of the concatenated (or minified) file
 - concat / min / qunit / watch – set options for each task (such as the files to operate on)
 - JSHint – examines your JavaScript code for possible problems
- The end of the file defines the default task
 - `grunt.register('default', 'lint qunit concat min')`

Chapter Concepts

Function Techniques

JavaScript Prototype

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Using Grunt to Manage JavaScript Projects

Chapter Summary

Chapter Summary

In this chapter, we have explored JavaScript technologies:

- Function techniques
- Prototypes
- Factories
- Closures
- Iterators and Generators
- Using Grunt

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Chapter 14: Functional and Reactive Programming in JavaScript

Chapter Overview

In this chapter, we will explore:

- JavaScript support of some functional programming features
- Several functions for working with arrays and lists
- How higher order functions accept functions as arguments and/or return a function
- Several frameworks that provide support for reactive programming in JavaScript

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Replacing Callbacks with Promises

Composition

Currying

Reactive Programming

Chapter Summary

The `forEach()` Function

- To iterate over a collection of objects:
 - Can use a `for` loop
 - Or use the `forEach` function
- The `forEach` function is defined in `Array.prototype`
 - Requires a function that will execute for each object in the array

```
var products = [  
  { name: 'Golf clubs', price: 175 },  
  { name: 'Basketball', price: 25 },  
  { name: 'Tennis racket', price: 95 },  
  { name: 'Baseball glove', price: 65 },  
  { name: 'Catnip mouse', price: 5.5 }  
];  
  
// traditional for loop  
for (var i = 0; i < products.length; i++) {  
  console.log(products[i]);  
}  
  
// the forEach function  
products.forEach(function(product, index) {  
  console.log(product);  
});
```

Which to Use?

- The `forEach` function improves readability
 - The next object is automatically passed to the function argument
 - Don't have to use loop counter variables
- Fewer off-by-one errors with `forEach`
 - No loop counter variables
 - No off-by-one bug
- The `for` loop allows breaking out early
 - If you need to break out of the loop before iterating completely through it:
 - Use the `break` key word
 - Not available in `forEach`

Usage of "break"

```
let text = "";
let i;
for (i = 0; i < 5; i++) {
  if (i === 3) {
    break;
  }
  text += "The number is " + i + "<br>";
}
```

The map () Function

- The `map()` function is defined in `Array.prototype`
- Creates a new array
 - Containing the results of calling the provided function on each array element
- The function passed to `map` can take three arguments
 - The current object
 - The index of the current object
 - The array being processed
- The `map` function can take an optional second argument
 - The object to be used as `this`

Global functions and array

```
function isOdd(num) { return num % 2 !== 0 }
function timesTwo(num) { return num * 2 }
let numbers = [1, 4, 9];
```

```
let doubles = numbers.map(timesTwo);
// or
let doubles = numbers.map(num => num * 2);
// doubles is now [2, 8, 18]
// numbers is still [1, 4, 9]
```

The filter() Function

- What if you only want to transform some of the values?
 - Map works on all the values
- The filter function solves this problem
 - It takes a function that returns a Boolean
 - True, keep the value
 - False, discard the value

```
let numbers = [1, 2, 3, 4];
let newNumbers = numbers.filter(isOdd) // [ 1, 3 ]
    .map(timesTwo) // [ 2, 6 ]
// or using arrow functions
let newNumbers = numbers.filter(n => number % 2 !== 0)
    .map(n => n * 2);
```

The reduce () Function

- What if I want to sum the values in an array?
 - Using the reduce function does the trick
- The second argument to reduce is the starting value
- The first argument to reduce is the function called for each element in the array

```
var numbers = [1, 2, 3, 4];

var totalNumber = numbers.map(function(number){
    return number * 2;
}).reduce(function(total, number){
    return total + number;
}, 0);

console.log("The total number is", totalNumber); // 20
```



HANDS-ON
EXERCISE

20 min

Exercise 14.1: Working with Arrays

- Follow the direction in your Exercise Manual for this exercise

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Replacing Callbacks with Promises

Composition

Currying

Reactive Programming

Chapter Summary

Higher Order Functions

- A higher order function is a function that takes another function as an argument
 - Like map, filter, reduce, etc.
 - The argument function is often referred to as a callback
- A function that returns a function:
 - Is also known as a higher order function

No More Loops

- Using higher order functions
 - There is no need for most imperative loops
 - No need to use the for loop
- The map, filter, reduce functions
 - Apply an argument function to every element in a collection
- Side-effect-free
 - Array higher order functions do not mutate the array they are called on



20 min

Optional Exercise 14.2: Using Higher Order Functions

- Follow the directions in your Exercise Manual for this exercise

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Replacing Callbacks with Promises

Composition

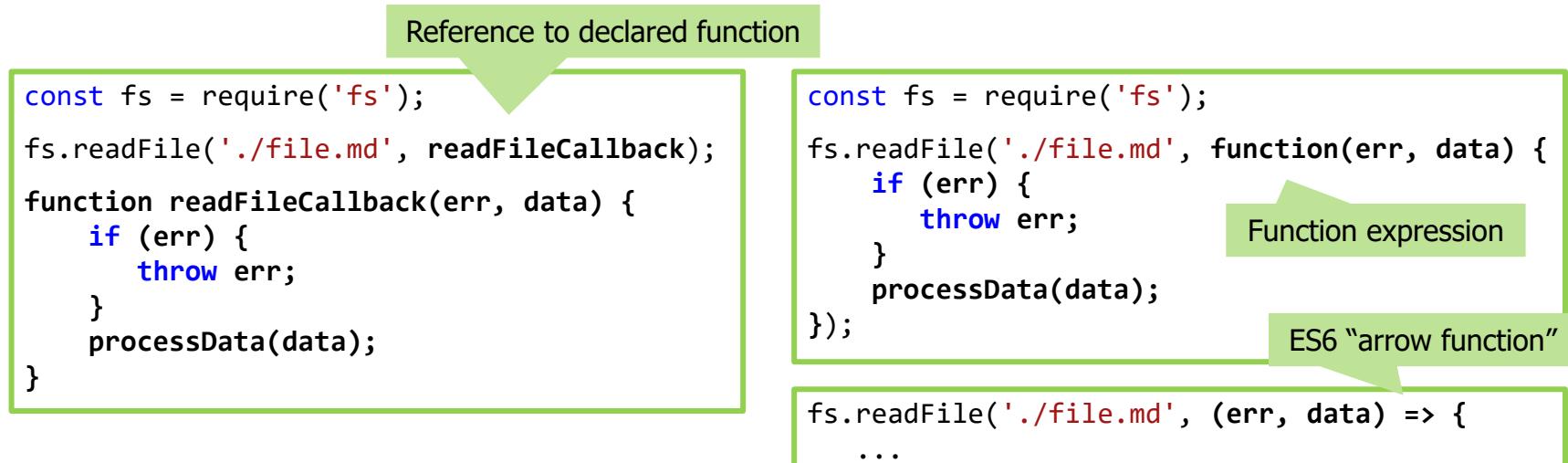
Currying

Reactive Programming

Chapter Summary

Callbacks

- JavaScript library functions often take a *callback* argument
 - Callback: a function passed as an argument to another function
 - Example: in Node standard library, `fs.readFile()` takes a callback argument
 - When the read operation is complete, `readFile()` calls the callback
 - Callback can be a reference to a declared function, or a function expression:



Callbacks with Asynchronous Operations

- Callbacks are good for asynchronous operations
 - When the order of processing is unpredictable
- Example: handling button clicks in a jQuery UI

```
$('#previous_button').click(function () {  
    ... // handle "Previous" button click  
});  
$('#play_button').click(function () {  
    ... // handle "Play" button click  
});  
$('#next_button').click(function () {  
    ... // handle "Next" button click  
});
```

Argument to `click()` method
is a callback function

User may click buttons in
any order, so functions may
be called in any order

Callbacks with Synchronous Operations

- **But:** callbacks aren't good for operations that must execute in a specific order
- Example: Reading transactions from a file to update records in a database
- Scenario:
 1. Send web service request to get current record from database
 2. Update record with new transaction data from input file
 3. Send web service request with updated record
- All steps require calls to asynchronous functions
 - But each step must complete before the next step begins
 - The asynchronous results must be processed in the correct order
- Implementing this use case with traditional callbacks results in "Callback Hell"
 - Deeply nested callbacks make code difficult to understand and maintain
 - See example on next slide

Callback Hell

```
function updateRecordFromTxnFile(filepath, txnId) {
    getRecordFromWebService(txnId, function(err, currentRecord) {
        if (err) {
            failureCallback(err);
        }
        else {
            updateRecordFromTxnFile(filepath, currentRecord, function(err, updatedRecord) {
                if (err) {
                    failureCallback(err);
                }
                else {
                    sendUpdatedRecord(updatedRecord, function(err, confNum) {
                        if (err) {
                            failureCallback(err);
                        }
                        else {
                            console.log(`Confirmation number ${confNum}`);
                        }
                    });
                }
            });
        }
    });
}
```

What's Really Going On?

```
function updateRecordFromTxnFile(filepath, txnid) {  
    getRecordFromWebService(txnid, function(err, currentRecord) {  
        if (err) {  
            failureCallback(err);  
        }  
        else {  
            updateRecordFromTxnFile(filepath, currentRecord, function(err, updatedRecord) {  
                if (err) {  
                    failureCallback(err);  
                }  
                else {  
                    sendUpdatedRecord(updatedRecord, function(err, confNum) {  
                        if (err) {  
                            failureCallback(err);  
                        }  
                        else {  
                            console.log(`Confirmation number ${confNum}`);  
                        }  
                    });  
                }  
            });  
        }  
    });  
}
```

1. Call `getRecordFromWebService()` ...

2. ...then call `updateRecordFromTxnFile()` ...

3. ...then call `sendUpdatedRecord()` ...

4. ...then log the final result

...but if there's an error anywhere,
call `failureCallback()`

Promises

- *Promise*: object that represents the eventual result of an asynchronous operation
 - Lets asynchronous methods return values like synchronous methods
 - Asynchronous method returns a *promise* to supply the value at some point in the future
- A Promise has two methods:
 - `then(successAction)` – *successAction* is a function to be executed on success
 - `catch(errorAction)` – *errorAction* is a function to be executed on error
- `then()` and `catch()` both return Promises
 - Allows chaining of method calls

```
function updateRecordFromTxnFile(filepath, txnId) {  
    getRecordFromWebService(txnId)  
        .then(currentRecord => updateRecordFromTxnFile(filepath, currentRecord))  
        .then(updatedRecord => sendUpdatedRecord(updatedRecord))  
        .then(confNum => console.log(`Confirmation number ${confNum}`))  
        .catch(err => failureCallback(err));  
}
```

If functions from previous example return Promises, code is much simpler

Standard Library Usage Without Promises

- Many standard Node modules now have a Promise-based API
 - Example: global replacement of a string in a file using fs module
 - Code below uses “classic” nested callback technique, code on next slide uses Promises

```
const fs = require('fs'); // require "classic" fs module                                replace_in_file_callbacks.js

function replaceInFileCallback(filename, str, repl) {
  fs.readFile(filename, 'utf8', (err, contents) => { // pass callback to fs.readFile()
    if (err) {
      console.error(err);
    } else {
      let result = contents.toString().replace(new RegExp(str, 'g'), repl); // do the replacement
      fs.writeFile(filename, result, 'utf8', (err) => { // pass callback to fs.writeFile()
        if (err) {
          console.error(err);
        } else {
          console.log(`done replacing ${str} with ${repl} in ${filename}`);
        }
      });
    });
}
```

Standard Library Usage with Promises

- Version 2: using `fs.promises` module for file operations
 - We still need callbacks as arguments to `then()` and `catch()`
 - But callbacks won't be nested

```
const fs = require('fs').promises; // require fs Promise API          replace_in_file_promises.js

function replaceInFile(filepath, str, repl) {
    return fs.readFile(filepath) // Read file asynchronously and return a Promise
        .then(contents => { // Result of read operation is input to next callback
            let result = contents.toString().replace(new RegExp(str, 'g'), repl);
            return fs.writeFile(filepath, result); // Write asynchronously, return a Promise
        })
        .then(() => console.log(`done replacing ${str} with ${repl} in ${filepath}`))
        .catch(err => console.log(err));
}
```

Format of Callbacks for `then()`

- The argument to `then()` is a callback function
 - Return value of the callback function becomes the argument to the next `then`'s callback
- You can specify a callback function's return value in two ways:
 - With an explicit `return` statement (required for multi-line callbacks)

```
doFirstThing()  
  .then(firstInput => {  
    let result = doSecondThing(firstInput);  
    console.log(`result = ${result}`);  
    return result;  
  })  
  .then(secondInput => ...)
```

This return value becomes the input to the callback of the next `then()`

- With the implied return of a single-line callback

```
doFirstThing()  
  .then(firstInput => doSecondThing(firstInput))  
  .then(secondInput => ...)
```

The value of this expression is implicitly the return value of the callback

Asynchronous Operations with Promises

- A function that starts an asynchronous operation can return a Promise
- The returned Promise represents the future result of the asynchronous operation
 - Initially the returned Promise is *pending*
 - If the operation is successful, then the Promise gets *fulfilled*
 - If the operation fails, then the Promise gets *rejected*
- So a Promise has three possible states:
 - **Pending** – the asynchronous operation is not yet complete (initial state)
 - **Fulfilled** – the asynchronous operation succeeded (the Promise was “kept”)
 - **Rejected** – the asynchronous operation failed (the Promise was “broken”)
- Promise's `then()` and `catch()` register actions (callbacks) for fulfilled and rejected states
 - `then(fulfilledAction)` : “If I am fulfilled, I'll call *fulfilledAction*”
 - `catch(rejectedAction)` : “If I am rejected, I'll call *rejectedAction*”

Functions that Return Promises

- Common idiom: asynchronous function creates and returns a new Promise object

```
function doAsyncOperation(...) { return new Promise((fulfill, reject) => { ... }); }
```

- The Promise constructor's argument is your *executor* function
 - Your executor function arguments are a fulfill function and a reject function

1. You call Promise constructor

2. Promise constructor calls your executor function

```
function doAsyncOperation(input) {
  return new Promise((fulfill, reject) => {
    ... // start an async operation
    if (...) // async operation succeeded
      fulfill(successValue);
    else
      reject(failureValue);
  });
}
```

3. In executor function, you start async operation

4. When async operation completes, you call
fulfill() or reject()

```
doAsyncOperation('some data')
  .then(value => recordSuccess(value))
  .catch(value => recordFailure(value))
```

6. Argument to fulfill() becomes argument to then() callback.
Argument to reject() becomes argument to catch() callback.

5. If you call fulfill(), then() callback executes. If
you call reject(), catch() callback executes.

Functions that Return Promises: Example

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

function readData(file) {
  return new Promise((fulfill, reject) => {
    fs.readFile(file, 'utf8', (err, data) => {
      if (!err) {
        fulfill(data);
      } else {
        reject(` ${file}: ${err}`);
      }
    });
  });
}
```

See `functions_returning_promises.js`

```
function writeData(file, data) {
  return new Promise((fulfill, reject) => {
    fs.writeFile(file, data, 'utf8', (err) => {
      if (!err) {
        fulfill();
      } else {
        reject(` ${file}: ${err}`);
      }
    });
  });
}
```

```
// copy a file
readData('./data/txn.json')
  .then(contents => writeData('./data/txn-copy.json', contents))
  .then(() => console.log('file copied successfully'))
  .catch(err => console.log(`file copy failed: ${err}`));
```



HANDS-ON
EXERCISE

30 min

Optional Exercise 14.3: Using JavaScript Promises

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Replacing Callbacks with Promises

Composition

Currying

Reactive Programming

Chapter Summary

Function Composition

- Function composition is the process of combining two or more functions
 - To produce a new function:
 - $\text{compose}(f, g)(x)$ equals $g(f(x))$

```
var compose = function(f, g) {  
    return function(x) {  
        return g(f(x));  
    }  
};
```

- This can be done more concisely with a lambda expression

```
var lambdaCompose = (f,g) => (x) => g(f(x));
```

- Process order is similar to piping in UNIX
 - Commonly used in data science

Function Composition (continued)

■ What will this produce?

```
let trim = function(str) {return str.replace(/^\s*|\s*$/g, '')};  
  
let capitalize = function(str) {return str.toUpperCase();};  
  
let convert = compose(trim, capitalize); // see previous slide  
  
console.log('"' + convert(' abc def ghi ') + '"');
```

■ What will this produce?

```
let add1 = function(x) {return x + 1;};  
let mult2 = function(x) {return x * 2;};  
  
// what will this produce?  
let f = compose(add1, mult2);  
console.log(f(7));
```



HANDS-ON
EXERCISE

20 min

Exercise 14.4: Using Function Composition

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Replacing Callbacks with Promises

Composition

Currying

Reactive Programming

Chapter Summary

Currying

- Currying is a way to construct functions that allow partial application of the function's arguments
 - If you pass all the arguments in a call:
 - You get the result back
 - If you pass a subset of the arguments:
 - You get a function that expects the rest of the arguments

```
function multiply(x, y) { return x * y; }
```

```
function curriedMultiply(x) {
    return function(y) { return x * y; }
}
```

```
var multiplyBy5 =
curriedMultiply(5);
multiply(5,2) ===
multiplyBy5(2);
```

Currying and Callbacks

- Currying can be combined with callbacks
 - To create a higher order *factory* function
- Useful in event handling
- Can replace the callback pattern used in node.js
- This code is an example of combining currying with node.js
 - To process a file
 - Allows for the read data to be passed around
 - As the file is being processed
 - Defer invoking the read function's callback
 - Until the result is needed
 - Can allow for sequential and parallel i/o processing of multiple files

```
// a curried version of node's fs.readFile(path,
encoding, callback) function
var readfileC = curriedReadfile(path, encoding);

readfileC(function(err, data) {
  if (err) {
    throw err;
  }
  // do something clever with the data
});
```

The Power of Currying

- With currying, you can separate the initiation of an asynchronous operation
 - From the retrieval of the result
- So, it is possible to initiate several operations in close sequence
 - Let them do their i/o in parallel
 - Retrieve the results later

```
var reader1 = curriedReadFile(path1, "utf8");
var reader2 = curriedReadFile(path2, "utf8");
// I/O is parallelized and we can do other important
things while it runs

// further down the line:
reader1(function(err, data1) {
  reader2(function(err, data2) {
    // do something clever with data1 and data2
  });
});
```



HANDS-ON
EXERCISE

20 min

Optional Exercise 14.5: Currying

- Follow the instructions in your Exercise Manual for this exercise

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Replacing Callbacks with Promises

Composition

Currying

Reactive Programming

Chapter Summary

Reactive Programming

■ Reactive systems are:

- Responsive
 - Responds in a timely manner
 - Focus on providing rapid and consistent response times
- Resilient
 - Stays responsive in the face of failure
- Elastic
 - Stays responsive under varying workload
- Message-driven
 - Relies upon asynchronous message passing

From *The Reactive Manifesto*, <http://www.reactivemanifesto.org/>

From Promises to Observables

- Promises act on data
 - And then return a single value
- Observables are the observer pattern at work
 - Can produce multiple values asynchronously over time
- An observable is a function that takes an observer
 - And returns a cancellation function
- An observer is an object with next, error, and complete functions

The Rx Library

■ Reactive Extensions (RxJs)

- A library for composing asynchronous applications
- Using observable sequences and LINQ-style query operators
- Works with synchronous and asynchronous data streams

	Single Return Value	Multiple Return Values
Pull/Synchronous/Interactive	Object	Iterables(Array...)
Push/Asynchronous/Reactive	Promise	Observable

LINQ – Language Integrated Query

Reactive Programming in JavaScript

- Reactive programming revolves around asynchronous data
 - Called observables
 - Or streams
- There are quite a few choices for frameworks that extend JavaScript
 - And add support for reactive programming
 - Reactive Extension (RxJs)
 - ReactiveX
 - Omniscient
 - WebRx

Observable Example

- The basic building blocks of RxJs
 - Observables (producers)
 - Observers (consumers)
- Two types of observables
 - Hot observables
 - Pushing even if we are not subscribed to them
 - Cold observables
 - Pushing only when we subscribe to them

```
// Creates an observable sequence of 5
// integers, starting from 1
var source = Rx.Observable.range(1, 5);

// Prints out each item
var subscription = source.subscribe(
  function (x) { console.log('onNext: %s', x); },
  function (e) { console.log('onError: %s', e); },
  function () { console.log('onCompleted'); });

// => onNext: 1
// => onNext: 2
// => onNext: 3
// => onNext: 4
// => onNext: 5
// => onCompleted
```

Prime Number Generator Example

- Suppose we want to aggregate the results from a prime number generator over time
 - So, the user interface does not have to deal with too many updates
 - We are interested in only the number of generated prime numbers
- Probably want to use a buffer
 - RxJs provides a buffer function
- Also, may want to use a map
 - To transform the data
- The `fromEvent` function constructs an observable

```
var worker = new Worker('prime.js');
var observable = Rx.Observable.fromEvent(worker, 'message')
    .map(function (ev) { return ev.data * 1; })
    .buffer(Rx.Observable.interval(500))
    .where(function (x) { return x.length > 0; })
    .map(function (x) { return x.length; });
```



HANDS-ON
EXERCISE

20 min

Exercise 14.6: Using Observables

- Follow the directions in your Exercise Manual for this exercise

Chapter Concepts

Working with Arrays and Lists

Higher Order Functions

Replacing Callbacks with Promises

Composition

Currying

Reactive Programming

Chapter Summary

Chapter Summary

In this chapter, we have explored:

- JavaScript support of some functional programming features
- Several functions for working with arrays and lists
- How higher order functions accept functions as arguments and/or return a function
- Several frameworks that provide support for reactive programming in JavaScript

Resources

- There is much more to the testing JavaScript
- Here are some resources to aid in further explorations



- **JavaScript: The Good Parts**
 - Douglas Crockford
 - <https://www.youtube.com/watch?v=hQVTIJBZook>
 - Available on Safari Books Online



- *Test-Driven JavaScript Development*
 - Gupta, Prajapati & Singh, Packt Publishing
 - Available on Safari Books Online

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Course Summary

Course Summary

In this course, we have:

- Solved common programming problems by using design patterns
- Designed and built RESTful web services
- Used JAX-RS and Spring Boot to create RESTful web services written in Java
- Used Node.js to execute RESTful services written in JavaScript
- Used some advanced JavaScript programming techniques

Fidelity LEAP

Technology Immersion Program

Developing RESTful Services

Appendix A: Building RESTful Services with JAX-RS

Appendix Overview

In this appendix, we will explore:

- What a RESTful web service is
- Building RESTful services with JAX-RS
- Testing RESTful web services
- Spring Support for JAX-RS

Appendix Concepts

RESTful Web Services

Building RESTful Services with JAX-RS

Testing RESTful Web Services

Spring Support for RESTful Services

Appendix Summary

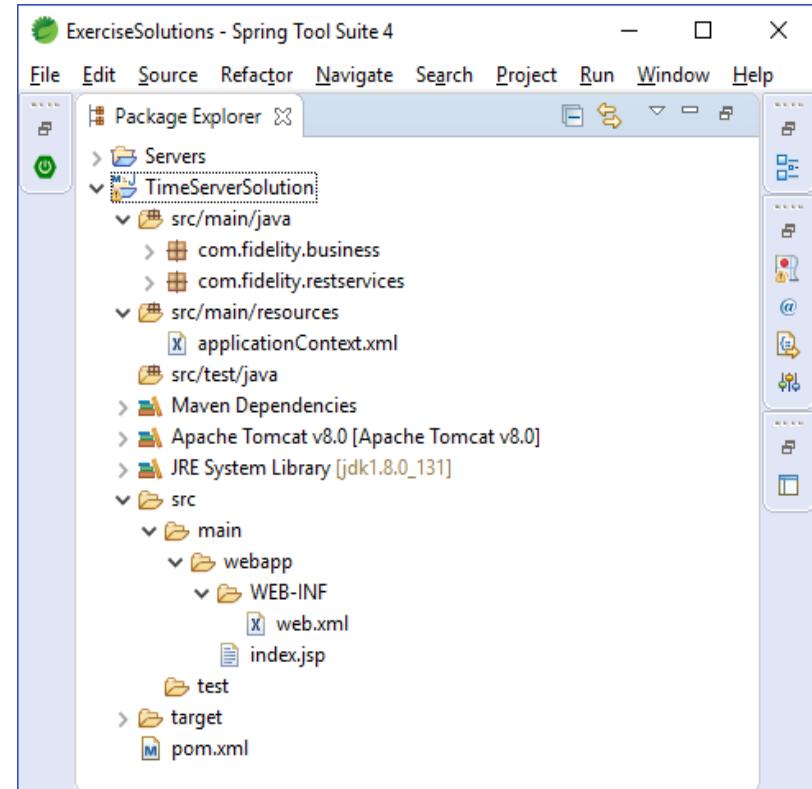
Structure of Maven Web Project

Java Resources

- src/main/java
 - Java classes
- src/main/resources
 - Resources like Spring configuration files

Deployed Resources

- src/main/webapp
 - The deployed web application
- WEB-INF/web.xml
 - A deployment descriptor used for servlet mappings and many other tasks
 - Any file under WEB-INF can be used by the server, but will not be available as a URL



Using web.xml to Map URLs

```
<?xml version="1.0" encoding="UTF-8" ?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>JaxrsServlet</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>com.fidelity.restservices</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>JaxrsServlet</servlet-name>
    <url-pattern>/jaxrs/*</url-pattern>
  </servlet-mapping>
</web-app>
```



HANDS-ON
EXERCISE

30 min

Exercise A.1: Exploring the Time Service

- Follow the instructions in your Exercise Manual for this exercise

Appendix Concepts

RESTful Web Services

Building RESTful Services with JAX-RS

Testing RESTful Web Services

Spring Support for RESTful Services

Appendix Summary

What Is JAX-RS?

- JAX-RS is a standard Java API
 - Specification is JSR311 (Java Service Request)
 - Reference implementation is called Jersey
 - Other implementations: Apache's CXF, JBoss' RESTEasy

<http://cxf.apache.org/>

<https://resteasy.github.io/>

Writing a Service

■ Steps to write a service:

- Write a plain Java class with methods
 - Use annotations to mark it as a service
 - Use annotations to mark methods as web service methods
- For each web service method in the class, specify:
 - URL of operation
 - HTTP operation
 - MIME types produced/consumed
 - Path, query, or form parameters used as input
- Deploy web archive

Step 1: ServletContainer Configuration

- In web.xml, configure ServletContainer servlet: specify packages and URL pattern

```
<web-app ... >
  <servlet>
    <servlet-name>JaxrsServlet</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>com.fidelity.greeter</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>JaxrsServlet</servlet-name>
    <url-pattern>/jaxrs/*</url-pattern>
  </servlet-mapping>
</web-app>
```

ServletContainer looks for RESTful services in this list of packages

ServletContainer intercepts URLs that start with jaxrs;
e.g., http://server/context/jaxrs/greet

Step 2: Write Service

- The service class is a plain Java class
 - Has to have a `@Path` annotation but its value could be an empty string
 - This is the URL pattern corresponding to this class

```
@Path("/greet") // @Path("") is also allowed
public class GreeterService {

}
```

- Because of the URL specified in `web.xml`, the full URL for this class is:

`http://localhost:8080/greeterService/jaxrs/greet`

- The machine URL comes from a Domain Name Service (DNS)
- The application context (`greeterService`) comes from name of WAR file
- `jaxrs` comes from `web.xml` URL pattern
- `greet` comes from the `@Path` of the service class

Step 3: Write Method

- For each method, specify the HTTP operation, URL, MIME types produced/consumed
 - Return values will be coerced into the MIME type specified in a common-sense manner

```
@GET  
@Path("/languages")  
@Produces(MediaType.TEXT_HTML) // or @Produces("text/html")  
public String getSupportedLanguages(){  
    return "<html><body><ol><li>English</li><li>French</li></ol></body></html>";  
}
```

- Note that the `@Path` adds to the path of the class, so that above method is invoked when user browses to the following URL:

```
http://localhost:8080/greeterService/jaxrs/greet/languages
```

Handling Form Inputs

- Can have form inputs injected as parameters to web service method

```
<form action="jaxrs/greet" method="POST">
    Your name:
    <input type="text" name="fullName"></input>
    <select name="language">
        <option value="fr">French</option>
        <option value="en">English</option>
    </select>
    <input type="submit" value="Submit" />
</form>
```

HTML form (client of web service)

- The web service method looks like this:

```
@POST
// no special path, so gets path of class
@Produces(MediaType.APPLICATION_JSON)
public String greetForm(@FormParam("fullName") String name,
                       @FormParam("language") String lang){
    // use name, lang normally
}
```

Raising Error Conditions

- Can raise HTTP error conditions by throwing a WebApplicationException

```
public String greetForm(@FormParam("fullName") String name,
                      @FormParam("language") String lang){
    String greeting;
    if ( "fr".equals(lang) ){
        greeting = "Bonjour, ";
    } else if ("en".equals(lang)){
        greeting = "Good morning, ";
    } else {
        throw new WebApplicationException(Status.BAD_REQUEST);
    }
    return "<greeting>" + greeting + name + "</greeting>";
}
```

- Can also have method return a Response object which contains only a HTTP Status (no document)

Handling Path Parameters

- Can inject variables in the URL to a method:

```
@GET  
@Path("/{language}/{name}")  
@Produces(MediaType.APPLICATION_JSON)  
public String greetPath(@PathParam("name") String name,  
                        @PathParam("language") String lang){  
    // use name, lang as normal  
}
```

- Example of Path handled by the above method:

```
http://localhost:8080/greeterService/jaxrs/greet/fr/Jane%20Doe
```

- Similarly, @CookieParam, @QueryParam, etc.

Query Parameters

- Query parameters can also be injected:

```
@GET  
@Produces(MediaType.APPLICATION_JSON)  
public String greetPath(  
    @DefaultValue("Guest") @QueryParam("name") String name,  
    @DefaultValue("en") @QueryParam("lang") String lang){  
        // use name, lang as normal  
}
```

- Examples of URLs handled by the above method:

http://localhost:8080/greeterService/jaxrs/greet?lang=fr&name=Jane%20Doe

http://localhost:8080/greeterService/jaxrs/greet?name=Jane%20Doe

http://localhost:8080/greeterService/jaxrs/greet

http://localhost:8080/greeterService/jaxrs/greet?lang=fr

Allowed Parameter Types

- Path and Query parameters can be used on:

- String
- All primitive types (and their wrappers) except `char`
- Any class with the static method `valueOf(String)`
 - So, any enum works
- Any class that has a constructor that takes only one `String`
- `List<T>`, `Set<T>`, or `SortedSet<T>` where `T` matches above criteria

```
@GET  
@Path("/details/{id}")  
@Produces(MediaType.APPLICATION_JSON)  
public String showDetails(@PathParam("id") Employee employee){  
    // etc.  
}
```

```
public class Employee {  
    public static Employee valueOf(String id){  
        // etc.  
    }  
}
```

XML and JSON

- JSON is the standard format used with REST services
- Clients are often Dynamic Web Applications using JavaScript
- Jax-RS handles both JSON and XML data
 - Will marshal data to/from Java classes based on type of data
- If a client requires data from a service in a particular format, they should set the HTTP header on the request
 - Accept: application/json or Accept:application/xml
 - Service will automatically send data in the correct format
 - With no change to code
- For a service to accept data, client should set HTTP header indicating type of data being sent
 - Content-type: application/json or Content-type:application/xml

Producing JSON or XML Example

- Consider the following service:

- Will produce XML or JSON based on HTTP header and data format

```
@Path("/exhibits")
@Singleton
public class ExhibitsService {
    private List<Exhibit> exhibits = new ArrayList<>();

    public ExhibitsService(){
        // create initial exhibits for service and add to list above
    }
    @Path("/all")
    GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public List<Exhibit> getExhibits(){
        return exhibits;
    } ...
```

Producing JSON or XML Example (continued)

- Following method will accept JSON or XML based on header set:

```
@Path("/add")
@POST
@Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
public void addExhibit(Exhibit exhibit){
    exhibits.add(exhibit);
}
```

```
@XmlRootElement
public class Exhibit {
    private String name;
    private String artist;
```

```
// get/set methods
```

```
}
```

Exhibit parameter will be created from
JSON or XML data



HANDS-ON
EXERCISE

30 min

Exercise A.2: Creating a RESTful Service

- Follow the instructions in your Exercise Manual for this exercise

Appendix Concepts

RESTful Web Services

Building RESTful Services with JAX-RS

Testing RESTful Web Services

Spring Support for RESTful Services

Appendix Summary

Testing RESTful Services as a POJO

- It is possible to test a RESTful service as a POJO using JUnit
- Simply create an instance of the service using a constructor
- In the test methods, call the service methods that you want to test
- This will verify that the service works locally

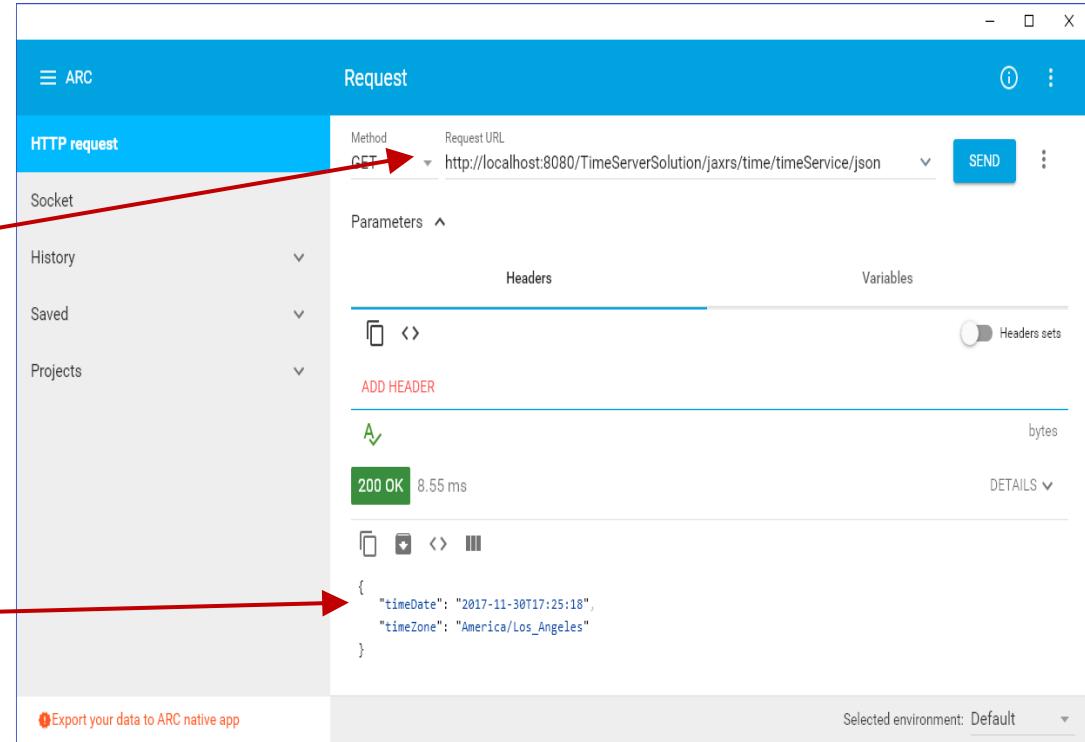
```
public class RestTimeServiceTest {  
    RestTimeService service;  
  
    @Before  
    public void setUp() throws Exception {  
        service = new RestTimeService();  
    }  
  
    @Test  
    public void testgetTime() {  
        TimeZoneInfo info = service.getTime();  
        assertNotNull(info);  
    }  
}
```

Testing RESTful Services as a Service

■ It is possible to test a RESTful service as a Service using tools like the Advanced Rest Client (ARC)

■ Using the tool, enter the URL for the service method that you want to call

- You can set parameters, pass data in the body of the request, and set HTTP headers
- The response data will be displayed once the request is sent





Exercise A.3: Testing a RESTful Service

40 min

- Follow the instructions in your Exercise Manual for this exercise

Appendix Concepts

RESTful Web Services

Building RESTful Services with JAX-RS

Testing RESTful Web Services

Spring Support for RESTful Services

Appendix Summary

Spring Support for RESTful Services

- The RESTful services we have created so far have a problem
- They are created by the ServletContainer servlet
 - Which means they are not Spring managed beans
 - Which means we cannot ask Spring to inject any dependencies into our RESTful services
- But wait! There is good news!
 - Spring provides an answer to this problem
 - Spring provides a servlet context listener
 - When the servlet is initialized, the listener creates a Spring context (object factory)
 - When the servlet is destroyed, the listener closes the Spring context

Configuring Spring Support for RESTful Services

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:beans.xml</param-value>
</context-param>

<servlet>
    <servlet-name>SpringJaxrsServlet</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
        <param-name>jersey.config.server.provider.packages</param-name>
        <param-value>com.fidelity.restservices</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>SpringJaxrsServlet</servlet-name>
    <url-pattern>/jaxrs/*</url-pattern>
</servlet-mapping>
```

Define Spring context listener and beans.xml configuration file

These sections are unchanged (normal servlet container)



30 min

Exercise A.4: Integrating Spring with JAX-RS

- Follow the instructions in your Exercise Manual for this exercise

Appendix Concepts

RESTful Web Services

Building RESTful Services with JAX-RS

Testing RESTful Web Services

Spring Support for RESTful Services

Appendix Summary

Appendix Summary

In this appendix, we have explored:

- What a RESTful web service is
- Building RESTful services with JAX-RS
- Testing RESTful web services
- Spring Support for JAX-RS