Building RESTful APIs with Spring MVC

Student Workbook 9

Version 2.2

Table of Contents

Module 1

Spring Framework

Section 1–1

Spring Framework

Spring Framework

- **In the early 2000s, a group of people led by Rod Johnson decided to build a framework that would make programming enterprise Java applications easier**
  - It makes web programming easier, but it can also be used in Java applications
- **Spring is a framework that is used to manage dependencies between objects in your application**
- **The core of the Spring Framework is based on two principles:**
  - dependency injection
  - inversion of control

Dependency Injection (DI) and Inversion of Control (IoC)

- **The Dependency Injection design pattern is based on the idea that you pass to an object the things it needs (dependencies) rather than having it construct them itself**

  – Example: Your data manager object might be passed in a `DataSource` identifying the database it needs to access

- **The Inversion of Control design pattern inverts the flow of control as compared to traditional control flow**

  – Rather than the programmer explicitly creating an instance (e.g. `new StudentService()`) of a dependency and passing this instance to an object, the dependency is "received"

  – A framework is used to manage the process of creating instances of the dependencies and passing them in

Spring and DI / IoC

- **Inversion of Control externalizes the creation and management of component dependencies inside of a Java Application**

- **Dependencies are injected by the Spring Inversion of Control (Ioc) Container at runtime**

  – This IoC container works with Dependency Injection to have the dependencies available at the locations necessary

- **This replaces a programmer manually passing an object into a constructor or a method**

  – Example: if you create a Service that manages Student Registration, and you require an object that handles Database Access (a Database Access Object), you can *inject* the database access object into the service using Spring

Example

```
// example without DI
public class StudentRegistration {
  private StudentDAO sDao = new StudentDAOImpl();

  // rest of the class
}
```

Example

```
// example with DI & Spring (and the correct configs in place)
public class StudentRegistration {
  private StudentDAO sDao;

  public StudentRegistration(StudentDAO sDao) {
    this.sDao = sDao
  }

  // rest of the class
}
```

- **Spring provides classes, configuration files, annotations, etc. to make this magic happen**

Styles of Dependency Injection

- **Dependencies can be injected into an object in three ways:**
  - through a constructor
  - through a setter method
  - through a field
- **Constructor dependency injection occurs when a component's dependencies are provided to it in its constructor(s)**
  - The IoC container passes (injects) the dependencies to the component when instantiation occurs and is managed by the Spring Framework

Example

```
public class StudentRegistration {
  private StudentDAO sDao;

  @Autowired
  public StudentRegistration(StudentDAO sDao) { … }

  // rest of the class
}
```

- **Setter dependency injection occurs when a component's dependencies are provided to it via a setter method**
  - The IoC container passes (injects) the dependencies via style setter methods (e.g. `setDataSource()`)

Example

```
public class StudentRegistration {
  private StudentDAO sDao;

  public StudentRegistration() { }

  // code omitted

  @Autowired     // <-- constructor injection
  public void setStudentDAO(StudentDao sDao) {
    this.sDAO = sDAO;
  }

  // rest of the class
}
```

- **NOTE: In the future, setters are not going to be used**

- **Field dependency injection occurs when a component's dependencies are provided to it via the field**

  – The IoC container passes (injects) the dependencies via the declaration of the field

  – Not the preferred DI option due to performance issues

Example

```
public class StudentRegistration {

  @Autowired   // <-- property injection
  private StudentDAO sDao;

  // rest of the class
}
```

Section 1–2

Example: Simple Spring App

Getting Started with a Basic Spring Application

- **Create a Java Application with the Maven structure**

- **Then, add Spring as a dependency in pom.xml because we want to use it in our application**

  – Add a Spring version into Maven using a property

Example

```
<properties>
  <org.springframework.version>
    5.2.8.RELEASE
  </org.springframework.version>
</properties>
```

- **Now, add a dependency that references the property**

Example

```
<dependency>
     <groupId>org.springframework</groupId>
     <artifactId>spring-context</artifactId>
     <version>${org.springframework.version}</version>
</dependency>
```

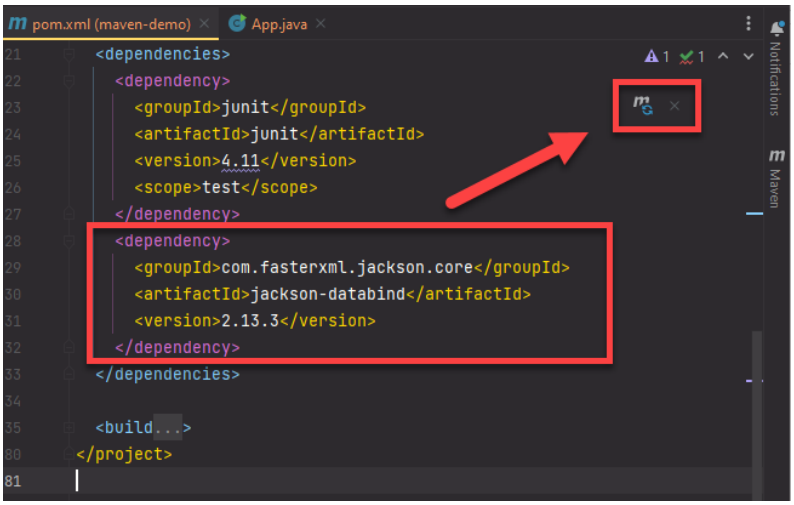
- **The completed maven pom.xml would be:**

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.o
g/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.xyzcorp</groupId>
    <artifactId>basic-spring</artifactId>
    <version>1.0-SNAPSHOT</version>

    <name>basic-spring</name>
    <!-- FIXME change it to the project's website -->
    <url>http://www.example.com</url>

    <properties>
           <project.build.sourceEncoding>
            UTF-8
         </project.build.sourceEncoding>
           <maven.compiler.source>1.8</maven.compiler.source>
           <maven.compiler.target>1.8</maven.compiler.target>

           <org.springframework.version>
                  5.2.8.RELEASE
           </org.springframework.version>
    </properties>

    <dependencies>
           <dependency>
                  <groupId>junit</groupId>
                  <artifactId>junit</artifactId>
                  <version>4.11</version>
                  <scope>test</scope>
           </dependency>
           <dependency>
                  <groupId>org.springframework</groupId>
                  <artifactId>spring-context</artifactId>
                  <version>${org.springframework.version}</version>
           </dependency>
    </dependencies>
</project>
```

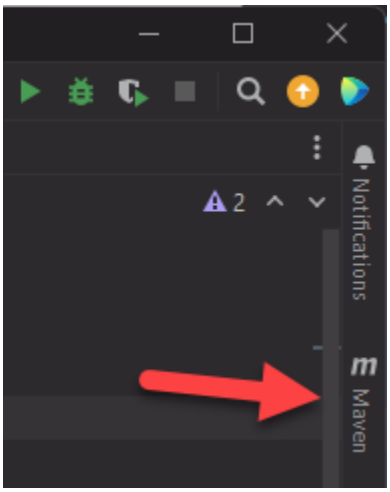
NOTE: Plugins not shown

Loading Dependencies into IntelliJ

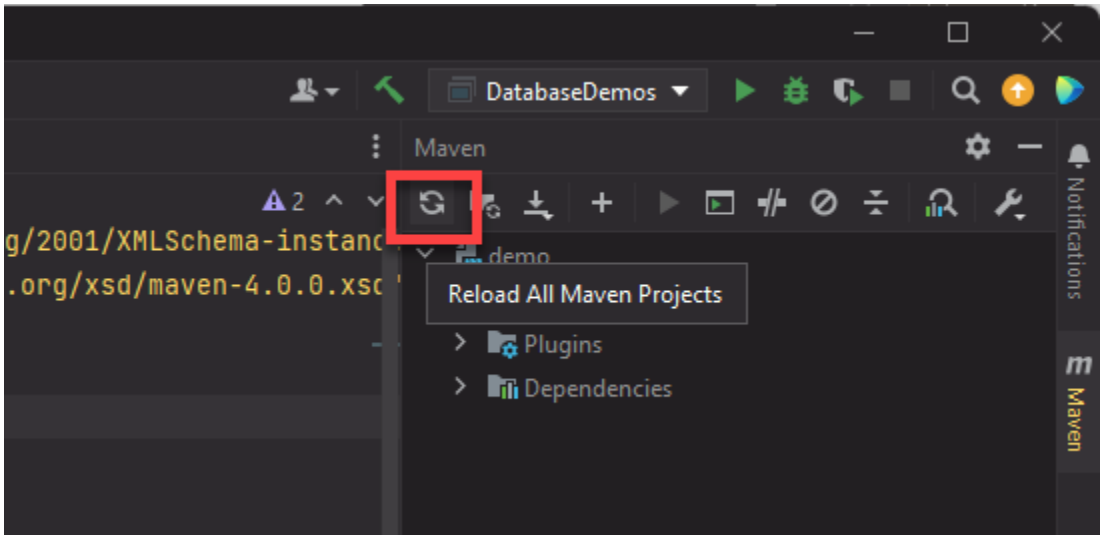
- **This is very important! After you add a Dependency, like Spring you must tell IntelliJ / Maven to update**



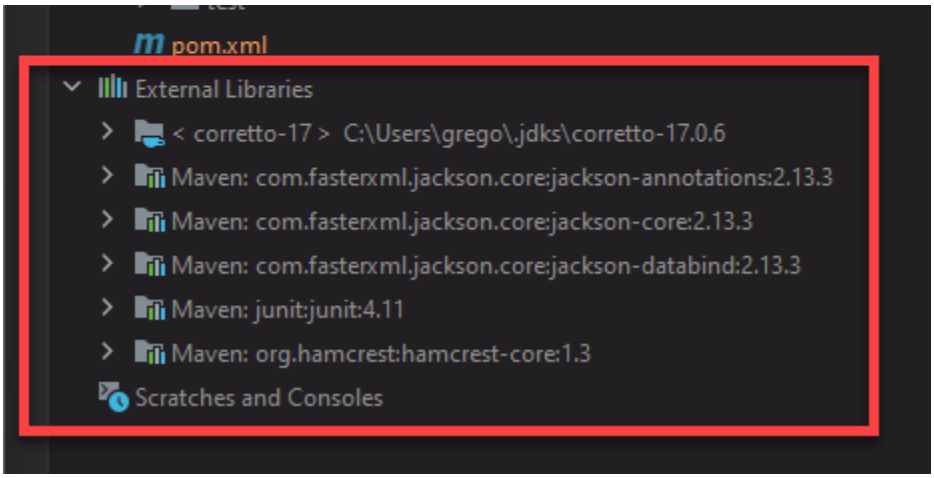
- **Dependencies can also be reloaded from the Maven toolbox**
  - Open the Maven toolbox



- Click the **Reload All Maven Projects** button



- You should be able to expand the **External Libraries** folder and view all the Spring jar files for your project

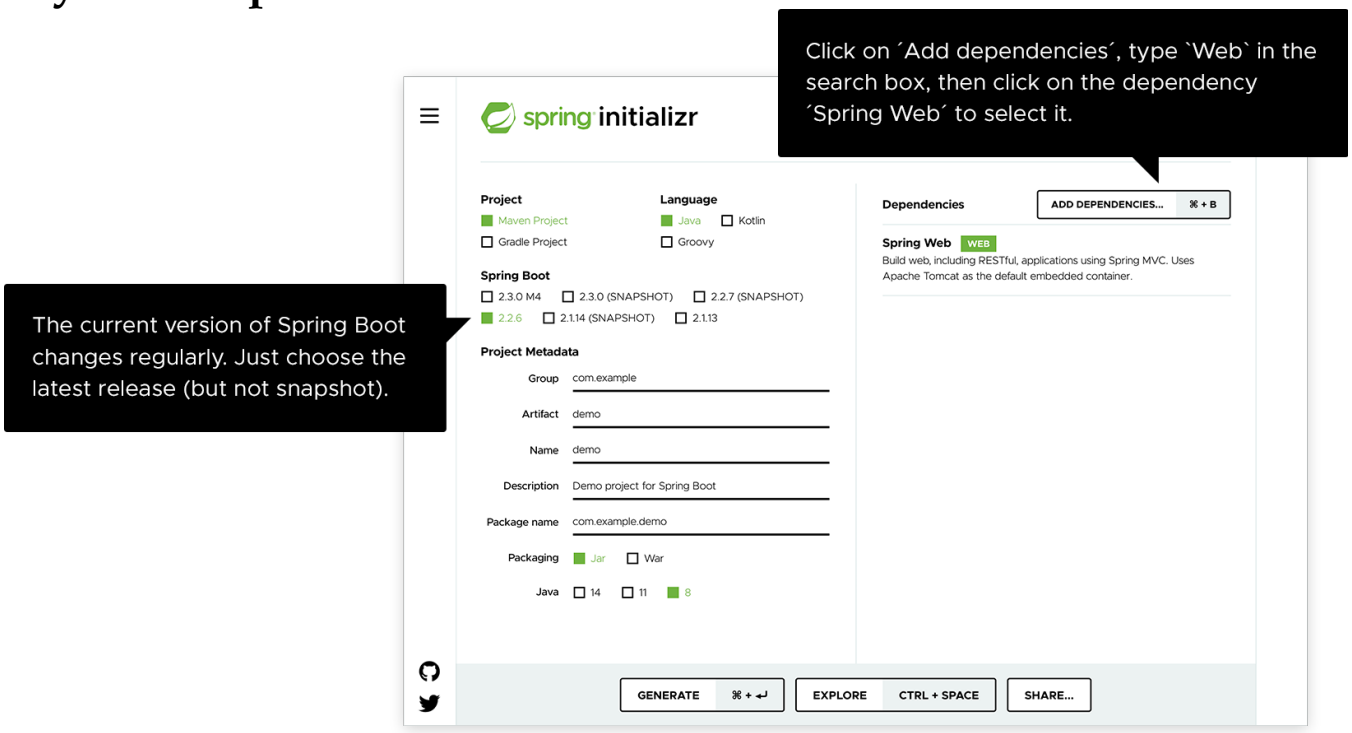


Spring Boot

- **Preconfigured "Out-of-the-box" Spring Application.**
- **Opinionated view of Spring with a set of libraries and extensions that are "Spring approved" and ready to go.**
  - The goal of Spring Boot is to take the best parts of Spring and make them even easier to use
- **Provides production-ready features, like metrics and health checks which are required for microservices.**
- **Spring Boot is completely annotation based**
  - There is little to no required XML configuration.
- **Building small, self-contained, ready to run applications can bring great flexibility and added resilience to your code.**

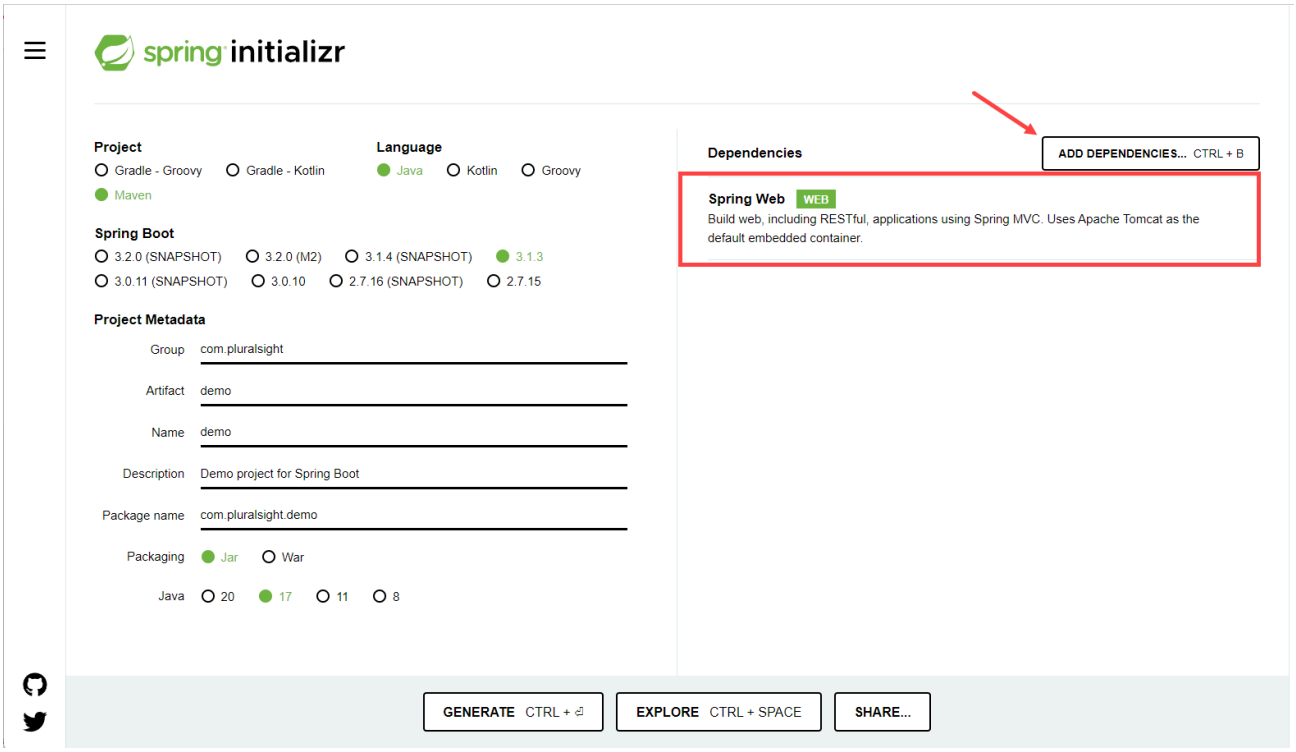
Getting Started with Spring Boot

- Use start.spring.io to create a “web” project.
- In the “Dependencies” dialog search for and add the “web” dependency as shown in the screenshot.
- Hit the “Generate” button, download the zip, and unpack it into a folder on your computer.
- The download, if you selected Maven as your build tool, will be ready to go and will just require an import into your Eclipse.



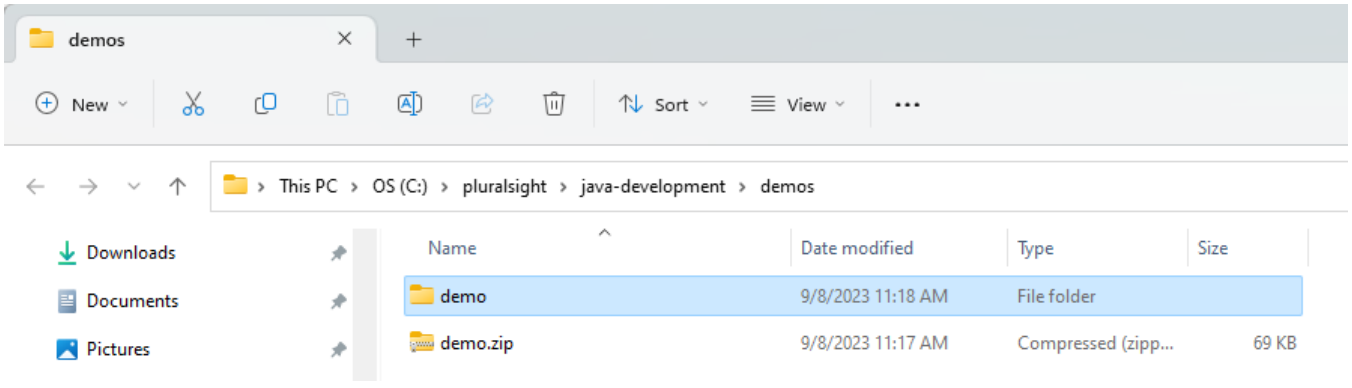
Creating a Spring Boot Demo

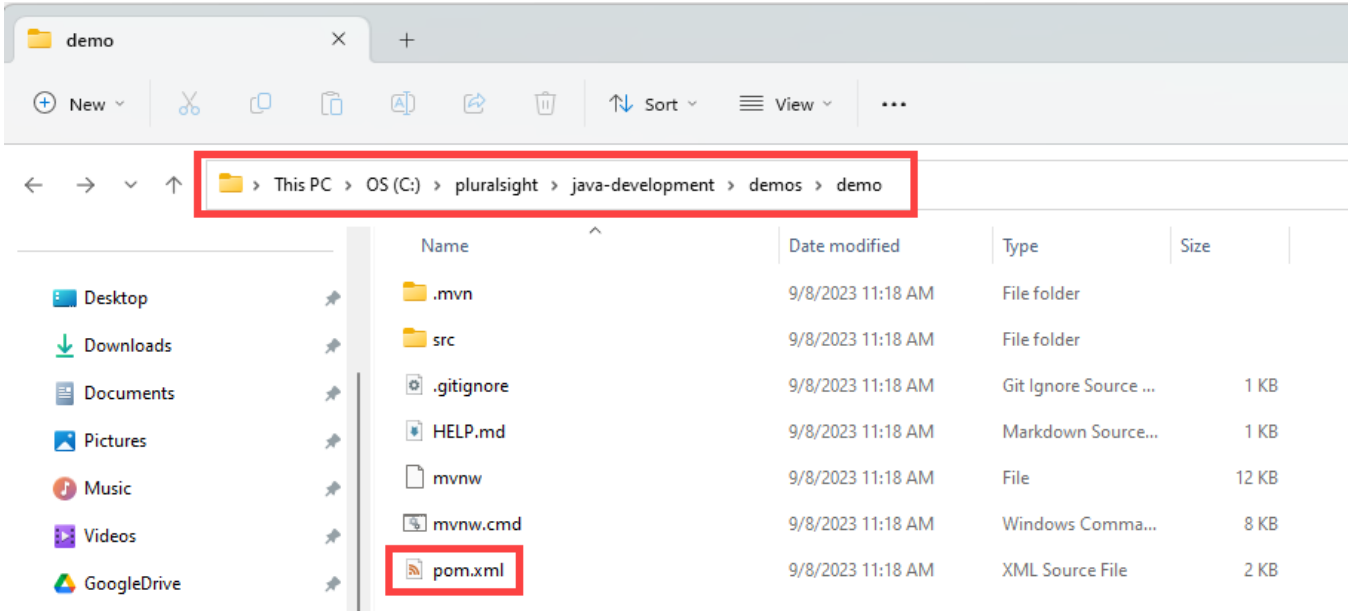
- Visit `https://start.spring.io/`
- In Dependencies add Spring Web
- Download into your workspace, and open in IntelliJ



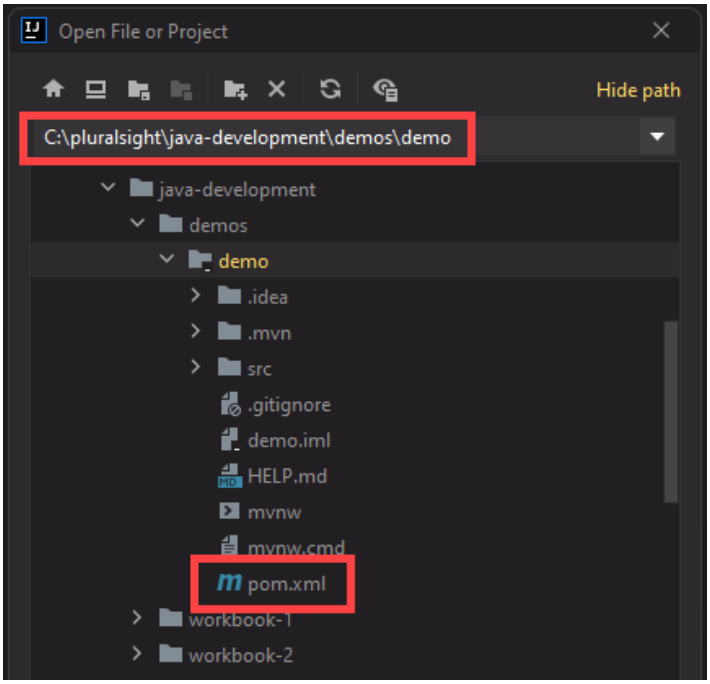
Opening the Spring Boot App in IntelliJ

- Once downloaded, copy and unzip the file to your workspace



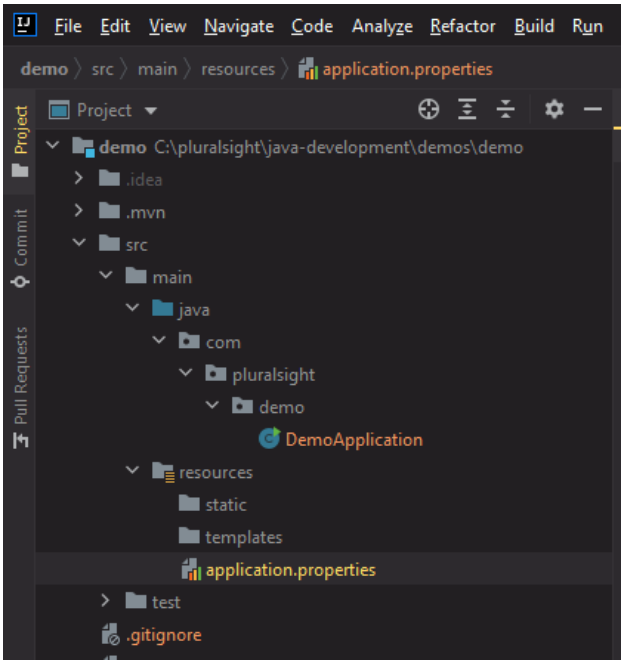


- **In IntelliJ choose File -> Open and navigate to the project folder to open it**
  - This is the folder that contains the `pom.xml` file



What do you get?

- A fully configured project with
  - All initial settings made
  - All configurations made
  - All libraries are included
  - Even an included Maven binary so you don't even need to download and install Maven.



`static void main`

- **In a Spring Boot application the `static void main` method changes**

- **The only purpose of the `main` method is to launch the application**

Example

```
@SpringBootApplication
public class Application {

  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

- **`SpringApplication.run` launches the application at its starting point**

  – If no custom application is defined, Spring Boot launches a default application

- **SpringApplication.run loads the ApplicationContext**

  – There is no `beans.xml` file

  – Spring Boot scans for any beans in the application

`static void main` *cont'd*

Example

NOTE: to get access to the ApplicationContext you just need to create a variable when you call the `run()` method

```
@SpringBootApplication
public class Application{

  public static void main(String[] args) {
    ApplicationContext context;
    context = SpringApplication.run(Application.class, args);
  }
}
```

Exercises

EXERCISE 1

You will create a new Java Maven project using Spring Boot. Open a browser and navigate to `https://start.spring.io`.

Configure your project with the following settings:

```
Selections:  Maven Project
             Java
             Latest stable Spring Boot version (no snapshot)
             Group: com.pluralsight
             Artifact: NorthwindTradersSpringBoot
             Jar
             Java Version 17
             No dependencies
```

Generate the project. Download and unzip it to your `workbook-9` folder.

Run the application to verify that your application runs.

Commit and push your code!

Java Beans

- JavaBeans are classes that encapsulate many objects and properties into a single object (the bean)

- It's a Java class that must adhere to the following conventions:

  – Must implement `Serializable` Interface

  – It must have a public **parameterless** constructor

  – All properties in Java bean must have public getter and setter methods and stick to naming conventions for these

Example

```
package com.pluralsight.demo;

import java.io.Serializable;

public class Person implements Serializable {

     // Private properties
     private String name;

     // Default Constructor - no parameters
     public Person() {}

     // Getters / Setters
     public void setName(String name) {
          this.name = name;
     }
     public String getName()  {
          return name;
     }
}
```

Example: Model for Student

Example

```
package com.pluralsight.model;

import java.util.Objects;

public class Student {
     private Long id;
     private final String firstName;
     private final String lastName;
     private final String major;

     public Student(Long id, String firstName, String lastName,
      String major) {
             this.id = id;
             this.firstName = firstName;
             this.lastName = lastName;
             this.major = major;
     }


     public Student(String firstName, String lastName,
      String major) {
             super();
             this.firstName = firstName;
             this.lastName = lastName;
             this.major = major;
     }

     public Long getId() {
             return id;
     }

     public String getFirstName() {
             return firstName;
     }

     public String getLastName() {
             return lastName;
     }

     public String getMajor() {
             return major;
     }
```

```
    @Override
    public String toString() {
            return "Student [id=" + id + ", firstName=" + firstName
                 + ", lastName=" + lastName + ", major=" + major
                 + "]";
    }

    @Override
    public int hashCode() {
            return Objects.hash(firstName, id, lastName);
    }

    @Override
    public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null) return false;
            if (getClass() != obj.getClass())     return false;

            Student other = (Student) obj;
            return Objects.equals(firstName, other.firstName)
                            && Objects.equals(id, other.id)
                            && Objects.equals(lastName, other.lastName);
    }

}
```

Example: Creating a RegistrationDAO

- **This is an interface that represents a Data Access Object.**
- **It would be responsible for handling storing a retrieving information in a database**

```
package com.pluralsight.dao;

import com.pluralsight.model.Student;

public interface RegistrationDAO {
     public Long persistStudent(Student student);
     public Student findById(Long id);
}
```

Example: Implementing RegistrationDAO

- **This is an implementation of the RegistrationDAO object**
  - Instead of running with an actual database we will make one out of an internal in-memory `HashMap`
- **This class will be marked with a `@Component` annotation to turn it into a bean**
  - This is an annotation that tells Spring to inject an object of this class whenever a (Simple)RegistrationDAO is required.

Example

```
package com.pluralsight.dao;

import java.util.HashMap;

import org.springframework.stereotype.Component;

import com.pluralsight.model.Student;

// Component tells Spring to create an object from the class and
// identify it as RegistrationDAO type. If any other component
// requires a RegistrationDAO type it will use this object created
// by Spring.

// The default scope for the object is singleton. Which means that
// it will be available for the duration of the application

@Component
public class SimpleRegistrationDAO implements RegistrationDAO {

    private HashMap<Long, Student> hashMap;
    private Long counter = 60L;
```

```
    public SimpleRegistrationDAO() {
      this.hashMap = new HashMap<Long, Student>();

      this.hashMap.put(10L, new Student(10L, "Marie", "Curie", "Science"));
      this.hashMap.put(22L, new Student(22L, "Albert", "Einstein", "Science"));
      this.hashMap.put(44L, Student(44L, "Niels", "Bohr", "Science"));
      this.hashMap.put(54L, new Student(54L, "Carl", "Jung", "Psychology"));
    }

    @Override
    public Long persistStudent(Student student) {
            long freshId = counter++;
            this.hashMap.put(freshId,
            new Student(freshId, student.getFirstName(),
                        student.getLastName(),
                        student.getMajor()));
            return freshId;
    }

    @Override
    public Student findById(Long id) {
            return hashMap.get(id);
    }

}
```

Example: Creating a Registration Service

- **The RegistrationService is a class that registers a student**
  - A Service usually is domain specific and not database specific in its method names
- **The Service requires a RegistrationDAO**
  - The RegistrationDAO is an interface -- not a specific implementation
- **Notice the constructor -- it contains an @Autowired annotation which tells Spring, look for a @Component that is a RegistrationDAO and inject it here**
- **The RegistrationService is also a bean because of the @Component and can be injected when required**

Example

```
package com.pluralsight.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.pluralsight.dao.RegistrationDAO;
import com.pluralsight.model.Student;


@Component
public class RegistrationService {

    private final RegistrationDAO registrationDAO;

    // Autowired tells Spring to automatically inject a
    // object into this constructor when a RegistrationService
    // is created, more on this later

    // When there's only one constructor with args, @autowired
    // is optional

    @Autowired
    public RegistrationService(RegistrationDAO registrationDAO) {
            this.registrationDAO = registrationDAO;
    }

    public Long registerStudent(Student student) throws
            StudentAlreadyRegisteredException {

            if (this.registrationDAO.findById(student.getId())!= null) {
                    throw new StudentAlreadyRegisteredException();
            }

            return registrationDAO.persistStudent(student);
    }
}
```

Exercises

Create a new folder in your `pluralsight` directory called `workbook-9`. Add code for the current week's exercises to this folder.

EXERCISE 1

You will be using the `NorthwindTradersSpringBoot` from the previous exercise.

Using the examples as a guide you will create admin capabilities for a store application to add products to your store. Create the following models.

Note: in this exercise you will not yet connect your DAO to a database, instead, the SimpleProductDAO will store a list of products in memory.

`Product` - class - add the following members

`productId` - int
`name` - String
`category` - String (for now)
`price` - double (or BigDecimal)

`ProductDao` - interface - add the following method definitions

```
add(Product product)
getAll()
```

`SimpleProductDao` - class - add the following members
---
implement the ProductDao interface (methods will interact with the list)
```
private List<Product> products
```

Configure the `SimpleProductDao` class as a Bean by adding the `@Component` annotation.

Add a constructor to the `SimpleProductDao` and add a few products to the list.

In your `main` method create a user interface that gives users the following options:

```
1.List Products
2.Add Product
```

Using Dependency Injection, create a variable for your `SimpleProductDao`. Use the DAO to store and retrieve products. **DO NOT** directly create a SimpleProductDao with a call to the constructor.

BONUS (optional):

Add the ability to Delete, Update and Search for products.

Commit and push your code!

Section 1–3

CodeWars

CodeWars Kata

- **Find your caterer**
  - You need to determine which caterer will give you the best value for your group size
- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/6402205dca1e64004b22b8de/java

Module 2

Spring Boot

Section 2–1

Getting started with Spring Boot

Annotations

- **Spring Boot relies heavily on the use of annotations for configuration**
  - This is a change from classic Spring which uses xml files extensively for project configuration
- **Annotations are a type of metadata that can be added to Java source code**
- **We have seen one already when we were working with inheritance and overriding methods in the child class:**
  `@Override`
- **Annotations can be added to classes, methods, fields and more**

Section 2–2

Spring Boot: beans and dependency injection

Spring Beans and Spring Container

- **Beans in Spring Boot are the same as in the Spring framework, but they're created differently. No more beans.xml file needed!**

- **A Spring Bean is an instance of a class, managed by the IoC container.**

- **The Spring IoC container is the core of the Spring Framework that configures the project and takes care of the dependency injection**

- **We have seen a bean already in the sample project. The class that holds the main method is a bean!**

- **Not all the classes are instantiated as beans, only the ones that are created using either the @Component or the @Bean annotation**

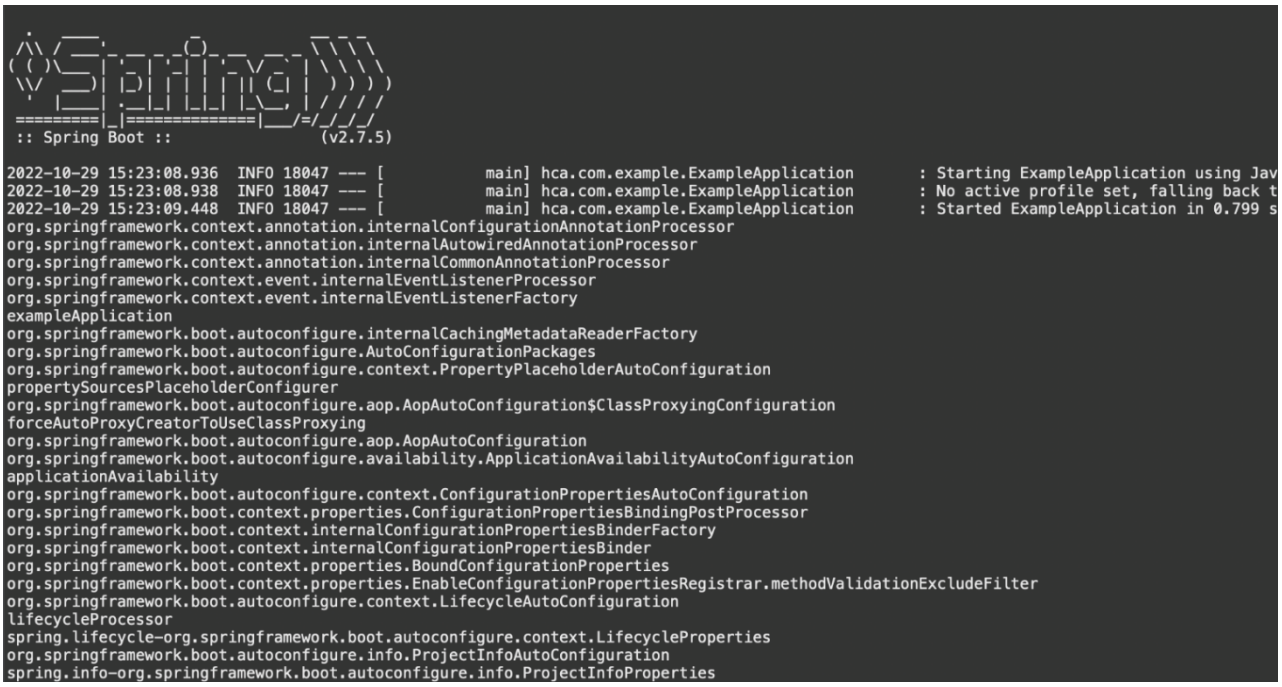
- **There are a lot of beans present that are part of Spring, let's inspect them by fetching the beans from the application context.**

Inspecting the beans

Example

```
@SpringBootApplication
public class ExampleApplication {

  public static void main(String[] args) {
    ApplicationContext ac =
      SpringApplication.run(ExampleApplication.class, args);
    for(String bean : ac.getBeanDefinitionNames()) {
      System.out.println(bean);
    }
  }
}
```

- **Output:**



Creating Spring Bean with @Component

- **We can add beans to the application context, by adding @Component on top of a class**

Example

```
package com.pluralsight.example;

import org.springframework.stereotype.Component;

@Component
public class ExampleBean {

}
```

- **If you run the code that outputs all the bean names again, you can see that it now contains a bean called `exampleBean`**

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.7.5)

2022-10-29 15:30:36.123  INFO 18704 --- [           main] hca.com.example.ExampleApplication       : Startin
2022-10-29 15:30:36.125  INFO 18704 --- [           main] hca.com.example.ExampleApplication       : No acti
2022-10-29 15:30:36.583  INFO 18704 --- [           main] hca.com.example.ExampleApplication       : Started
org.springframework.context.annotation.internalConfigurationAnnotationProcessor
org.springframework.context.annotation.internalAutowiredAnnotationProcessor
org.springframework.context.annotation.internalCommonAnnotationProcessor
org.springframework.context.event.internalEventListenerProcessor
org.springframework.context.event.internalEventListenerFactory
exampleApplication
org.springframework.boot.autoconfigure.internalCachingMetadataReaderFactory
exampleBean
org.springframework.boot.autoconfigure.AutoConfigurationPackages
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration
propertySourcesPlaceholderConfigurer
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration$ClassProxyingConfiguration
```

Creating Spring Bean with @Bean

- **In special files that are annotated with @Configuration, we can create beans by using the @Bean annotation on top of a method**

Example

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ExampleConfiguration {

  @Bean
  public String nameBean() {
    return "Maaike";
  }

}
```

- **What this method returns is going to be the bean in the application context**
- **This way, you have more control over the exact content of the bean**
- **The name of this bean will be the name of the method**
- **The type of the bean will be String**

Accessing a Bean

- **Creating a Bean tells Spring Boot to store an instance of that Bean in the IoC container**
  - The IoC container is the `ApplicationContext`

Example

`Product.java`

```
public class Product {
  private int id;
  private String name;
  private String category;
  private double price;

  public Product(){}

  public Product(int id, String name,
                 String category, double price) {
    this.id = id;
    this.name = name;
    this.category = category;
    this.price = price;
  }

  public int getId(){ return id; }
  public void setId(int id) { this.id = id; }

  public String getName() { return name; }
  public void setName(String name) { this.name = name; }

  public String getCategory() { return category; }
  public void setCategory (String category)
  { this.category = category; }

  public double getPrice() { return price; }
  public void setPrice(double price) { this.price = price; }
}
```

AppConfig.java

```
@Configuration
public class AppConfig {

  @Bean
  public Product product(){
    return new Product(55, "Hot Chocolate", "Beverages", 2.50);
  }
}
```

Application.java

```
@SpringBootApplication
public class Application {

  public static void main(String[] args) {
    ApplicationContext context;
    context = SpringApplication.run(Application.class, args);

    // this locates a Bean that is of type Product
    Product product = context.getBean(Product.class);
  }

}
```

Other ApplicationContext methods

- **Interesting ApplicationContext methods include**
  - `getBean()` this has several overloads
    * `getBean(Class<T>)` - looks for a bean that is of the same type
    * `getBean(String)` - looks for the bean by name (method name in the config file or class name if using `@Component`) - return type is Object
    * `getBean(String, Class<T>)` - search by name and convert to correct type (this allows you to have several beans of the same type, but with different names)
  - `containsBean(String)` - verifies that a bean with the specified name has been loaded

Example

AppConfig.java

```
@Configuration
public class AppConfig {

  @Bean
  public Product hotChocolate(){
    return new Product(55, "Hot Chocolate", "Beverages", 2.50);
  }

  @Bean
  public Product coke(){
    return new Product(56, "Mountain Dew", "Beverages", 1.50);
  }

}
```

Application.java

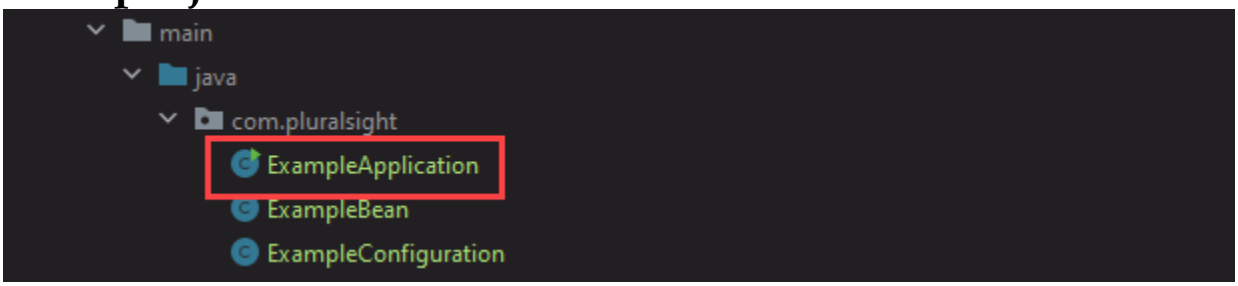
```
@SpringBootApplication
public class Application {

  public static void main(String[] args) {
    ApplicationContext context;
    context = SpringApplication.run(Application.class, args);


    Product product = null;

    // check if the bean exists before getting it
    if(context.containsBean("pepsi") {
      product = context.getBean("pepsi", Product.class);
    }
    else if (context.contains("coke"){
      product = context.getBean("coke", Product.class);
    }
    else {
      product = context.getBean("hotChocolate", Product.class);
    }
  }
}
```

Section 2–3

Injecting Beans

Component scanning

- Spring Boot automatically scans for beans
- It is important that the classes that need to become beans or contain methods annotated with @Bean are at the same level or a sublevel of the base package for component scanning, otherwise, the files are not found.
- The default base package for component scanning is the class that has the @SpringBootApplication annotation
- This project structure will find the beans:



- This project structure won't find the beans, because the method containing the main method is at a lower directory level:

Getting rid of ApplicationContext

- **Up until now we have been creating beans by manually calling the `ApplicationContext getBean` method**

- **When we launch a Spring Boot application, it can give us access to beans automatically**
  - We don't need to call `context.getBean()` manually

- **This means we need Spring Boot to manage our entire application**
  - We cannot just write application code inside `static void main`
  - The purpose of the `main` method is just to launch the application

- **To create a console application, create a new class that implements the `CommandLineRunner` interface**
  - This class is the new Application entry point

CommandLineRunner

Example

In this example the MainProgram.java class is the class that has the static void main method in it. It is the default entry point for a Java application

`Application.java`

In this example the Application class implements the CommandLineRunner interface. This allows us to hook into the SpringBoot application run event.

It must also include the @Component annotation so that this class is managed as a Spring Bean

```
@Component
public class Application implements CommandLineRunner {

  @Override
  public void run(String... args) {
    // your application logic starts here
  }
}
```

`MainProgram.java`

The `SpringApplication.run` method must launch the class that is the Spring IOC container.

```
@SpringBootApplication  // <-- the Spring IOC container
public class MainProgram {

  public static void main(String[] args) {

     // this line is where SpringBoot takes over
     // SpringBoot will check if there are any beans that
     //   implements the CommandLineRunner interface and launch it
     SpringApplication.run(MainProgram.class, args);
  }
}
```

Dependency injection

- **Spring instantiates the beans**
- **Sometimes beans have a property that is another bean**
- **This property is being set to the instance of that type (bean) that exists in the application context**
- **This phenomenon is called dependency injection**

Example

Instead of doing something like this:

```
@Component
public class ExampleBean {
    private String name = "Maaike";
}
```

Spring does the instantiation, so you can write your Bean code like this:

```
@Component
public class ExampleBean {
    @Autowired
    private String name;
}
```

Different dependency injection types

- **There are three types of dependency injection**
  - Field injection
  - Constructor injection
  - Setter injection
- **Field injection is typically not recommended due to compromising the performance**

Example

The example we've already seen uses field injection.

```
@Component
public class ExampleBean {
     @Autowired
     private String name;
}
```

Constructor injection is achieved by adding a constructor that takes the dependencies that are available as beans:

```
@Component
public class ExampleBean {
     private String name;

     @Autowired
     public ExampleBean(String name) {
            super();
            this.name = name;
     }
}
```

Setter injection is achieved by adding @Autowired to the setter:

```
@Component
public class ExampleBean {
     private String name;

     public String getName() {
            return name;
     }

     @Autowired
     public void setName(String name) {
            this.name = name;
     }
}
```

Which bean will be injected

- **The Spring IoC container needs to know which bean to inject**
- **It primarily does this by using the datatype of the bean**
- **When there are multiple beans of the same type the application fails to run**

Example

AppConfig.java

```
@Configuration
public class AppConfig {

  @Bean
  public Product candy(){
    return new Product(55, "Milka Alpen Milch", "Candy", 2.50);
  }

  @Bean
  public Product drink(){
    return new Product(56, "Coke Zero", "Beverages", 1.50);
  }

}
```

Example

SpringBoot will fail to autowire a bean because there are 2 Spring Beans that are a Product datatype

```
@Autowired
private Product drink;
```

Resolving Spring Bean Ambiguity

- **Bean datatype ambiguity can be resolved by specifying the name of the bean to load**
  - The `@Qualifier` annotation allows you to specify the name of the bean to load
  - The method name in the config class is used as the name

Example

```
@Autowired
@Qualifier("drink")
private Product drink;

@Autowired
@Qualifier("candy")
private Product candy;
```

- **Beans can also be named using the `@Bean` annotation**

Example

```
@Configuration
public class AppConfig {

  @Bean("candy")
  public Product defaultCandy(){
    return new Product(55, "Milka Alpen Milch", "Candy", 2.50);
  }

  @Bean("drink")
  public Product defaultBeverage(){
    return new Product(56, "Coke Zero", "Beverages", 1.50);
  }

}
```

Exercises

Continue working in your NorthwindTradersSpringBoot project.

EXERCISE 3

Create a new class for your command line application. Call it `NorthwindApplication`. It must implement the `CommandLineRunner` interface.

Create a class level variable for your `ProductDao` interface. Use the `@Autowired` annotation to allow Spring Boot to inject the bean for you.

```
@Autowired
private ProductDao productDao;
```

Move your application logic from the `static void main` method into the `run` method of your new `NorthwindApplication` class.

Update the `main` method to launch the SpringBootApplication. All application logic should have been moved to the `NorthwindApplication` class.

Commit and push your code!

Section 2–4

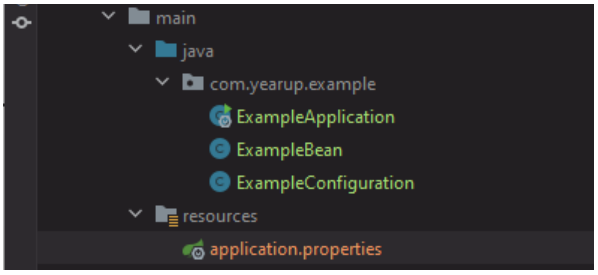
Spring Boot configuration

Configuring a Spring Boot application

- **There are different ways to configure a Spring Boot application**
- **We can use the @Configuration class to add all sorts of configuration, amongst which @Bean methods that define beans**
- **We are going to explore:**
  - Using properties
  - Using profiles

Using properties

- **We can use properties from a property file**
- **Property files are located in src/main/resources**
- **Spring Boot includes a default file named `application.properties`**



- **The properties file is a configuration file that stores values you will need in your application**
  - They are stored in `key-value` pair format
    * key=value
    * Each line holds a single key-value pair
  - It is common to store configuration variables that you do not want to have hard coded into your application - such as database connection

Example

```
datasource.url=jdbc:mysql://localhost:3306/devdatabase
datasource.username=localuser
datasource.password =localpasswordUsing properties    cont'd
```

- **The properties can be used in classes by adding the source of the properties on top, using an annotation**

- **The property can be used with the `@Value` annotation, `@Value(${name.prop})`**

Example

```
@Configuration
public class DatabaseConfig {

  private BasicDataSource basicDataSource;

  @Bean
  public DataSource dataSource(){
    return basicDataSource;
  }

  @Autowire
  public DatabaseConfig (@Value("${datasource.url}") String url,
              @Value("${datasource.username}") String username,
              @Value("${datasource.password}") String password)
  {
    basicDataSource = new BasicDataSource();
    basicDataSource.setUrl(url);
    basicDataSource.setUsername(username);
    basicDataSource.setPassword(password);

    // print the values to the screen just to verify it worked
    System.out.println(url + " : " + username + " : " + password);
  }
}
```

- **In this example we configured a DataSource bean.**

  – The connection string and user information was pulled from the application.properties file.

Example

This example goes back to our student registration scenario. Instead of storing student information in memory, we want to use a database. We create a new class that implements the RegistrationDAO interface.

```
@Component
public class JdbcRegistrationDAO implements RegistrationDAO {
  private DataSource dataSource;

  @Autowired
  public JdbcRegistrationDAO (DataSource dataSource) {

    // the dataSource is injected from the Configuration class
    this.dataSource = dataSource;
  }

  // implement the interface methods here
  public Long persistStudent(Student student){

    // code to insert the student into the database here
    // use the injected dataSource to connect to the database
  }

  public Student findById(Long id){

    // code to search for a student in the database
  }
}
```

Exercises

Continue working in your NorthwindTradersSpringBoot project.

EXERCISE 4

In this exercise you will add values to your `application.properties` file and use a configuration class to read the values into your application.

Add key value pairs with the information needed to connect to your Northwind database. You will need the following values:

```
connectionUrl
username
password
```

Create a `DatabaseConfiguration` class that has logic to create a new `DataSource` bean.

HINT: You will need to update your `pom.xml` file to import the MySQL driver and the `BasicDataSource` dependencies. (see Workbook 8 - Module 4)

Bonus (optional)

Create a NEW `JdbcProductDao` class that implements the `ProductDao` interface.

Autowire the dataSource dependency, and implement the interface members. Add the code to perform the CRUD operations in the Northwind database.

Add the `@Component` annotation to the new JdbcProductDao class and verify that your application now read from and writes to the database.

HINT: you will need to remove the `@Component` annotation from the `SimpleProductDao` class

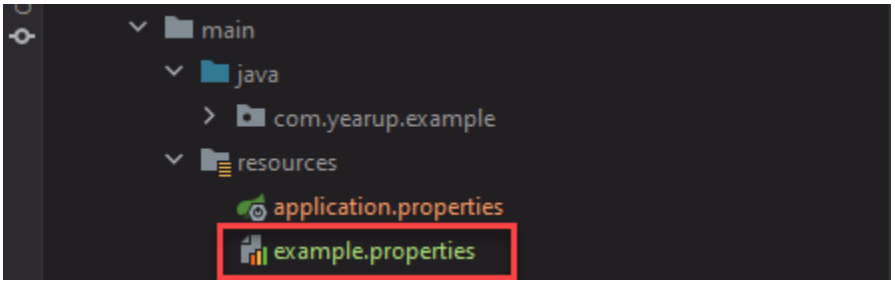
Commit and push your code!

Section 2–5

Custom Properties and Profiles (Self-Study)

Custom Properties Files

- **You can also create custom properties files**
  - It is just a text file with the extension `.properties`



- **To read values from a customer properties files you must specify the file name with the PropertySource annotation**
  - If no `PropertySource` is specified Spring Boot will search the `application.properties` file for keys

Example

```
@Configuration
@PropertySource("classpath:example.properties")
public class ExampleConfiguration {

    @Bean
    public String valueProp(@Value("${greeting}") String hi) {
            return hi;
    }
}
```

Using profiles

- **With properties we can configure our applications, but what if we wanted to have different configuration options available…**
- **This is a common need, for example, different configurations for the development, test and production stage**
- **This can be done with profiles**
- **We can map a bean to a certain profile, this is done with the @Profile annotation**

Example

DevConfig.java

```
@Configuration
@Profile("dev")
public class DevConfig {
     public String setting = "dev";

     public DevConfig() {
            System.out.println("In DevConfig constructor: " + setting);
     }
}
```

TestConfig.java

```
@Configuration
@Profile("test")
public class TestConfig {
     public String setting = "test";

     public TestConfig() {
            System.out.println("In TestConfig constructor: " + setting);
     }
}
```

- **There are many ways we can enable a certain profile, the one that we're going to be using is by adding a property to the application.properties file**

- **This property file is used for many configuration of a Spring Boot project**

Example

```
spring.profiles.active=dev
```

- **When we run the application we see the text from the DevConfig constructor in our output and not the text from the TestConfig**

Example: Using Profiles

- **Given 3 different configuration files, you can quickly switch which database your application connects to**
  - Just modify the `spring.profiles.active` value in the `application.properties` file

Example

database.dev.properties

```
connectionUrl=jdbc:mysql://localhost:3306/devdatabase
username=localuser
password =localpassword
```

database.test.properties

```
connectionUrl=jdbc:mysql://testserver:3306/testdatabase
username =testuser
password =testpassword
```

database.prod.properties

```
connectionUrl=jdbc:mysql://productionserver:3306/livedatabase
username =productionuser
password=$3cur3P@$$w0rd
```

application.properties

```
spring.profiles.active=dev
```

- **You would need a different configuration class for each profile.**

Example

DatabaseConfigDev.java

```
@Configuration
@Profile("dev")
@PropertySource("classpath:database.dev.properties")
public class DatabaseConfigDev {

  private BasicDataSource basicDataSource;

  @Bean
  public DataSource dataSource(){
    return basicDataSource;
  }

  @Autowire
  public DatabaseConfigDev (@Value("${connectionUrl}") String url,
                            @Value("${username}") String username,
                            @Value("${password}") String password)
  {
    basicDataSource = new BasicDataSource();
    basicDataSource.setUrl(url);
    basicDataSource.setUsername(username);
    basicDataSource.setPassword(password);
  }
}
```

DatabaseConfigTest.java

```
@Configuration
@Profile("test")
@PropertySource("classpath:database.test.properties")
public class DatabaseConfigTest {

  private BasicDataSource basicDataSource;

  @Bean
  public DataSource dataSource(){
    return basicDataSource;
  }

  @Autowire
  public DatabaseConfigTest(@Value("${connectionUrl}") String url,
                            @Value("${username}") String username,
                            @Value("${password}") String password)
  {
    basicDataSource = new BasicDataSource();
    basicDataSource.setUrl(url);
    basicDataSource.setUsername(username);
    basicDataSource.setPassword(password);
  }
}
```

Section 2–6

CodeWars

CodeWars Kata

- **Decimal Time Conversion**
  - Write 2 functions
    * One converts a decimal into hours and minutes
    * The other converts hours and minutes to a decimal

- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/6397b0d461067e0030d1315e/java

Module 3

Network Communication

Section 3–1

Network Communication

Single Computer

- **Even applications that run on a single computer often need to communicate with each other**
  - Your Java Application with your MySQL Database
- **Think of your computer like an industrial complex which leases office space**
  - You can install many applications on your computer
  - Each application is like a business leasing office space
  - All applications have the same physical address (your computer)
  - But each "office" can have an internal address (like an apartment or unit number)
    * The unit number is comparable to a port number on a computer
- **MySQL is a standalone application that allows other applications to interact with it**
  - The server address of MySQL installed on the local machine is `localhost`
  - MySQL listens on port `3306`

What is DNS

- **Each computer, like a business complex, has an ip address**
  - Ip address `127.0.0.1` refers to itself
  - Often referred to as Home
  - "There's no place like 127.0.0.1"
- **Your computer has a local DNS (Domain Name Server)**
  - This allows you to store aliases (or nicknames) for known ip addresses
  - `localhost` is an alias for `127.0.0.1`

Example

The following strings represent a JDBC connection string to a MySQL server that is installed on the same computer as the running computer.

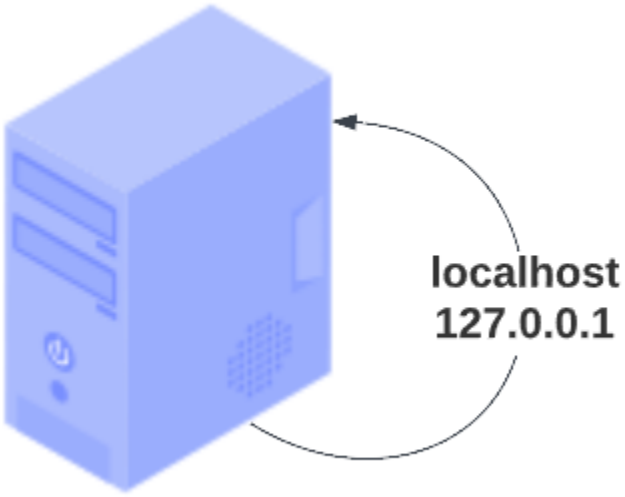
```
jdbc:mysql://localhost:3306/mydatabase

equals

jdbc:mysql://127.0.0.1:3306/mydatabase
```

The following urls represent a link to a website that is running on the same computer as the browser.

```
http://localhost/homepage.html

equals

http://127.0.0.1/homepage.html
```

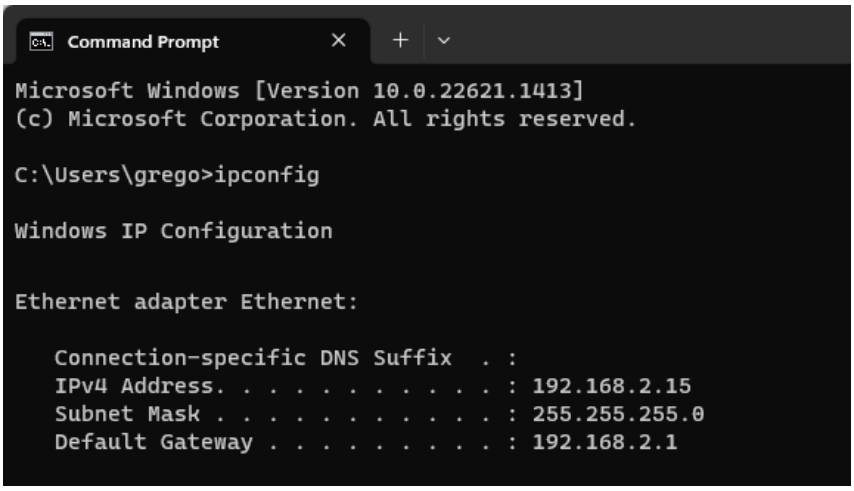
localhost

- **Localhost refers to the running computer**

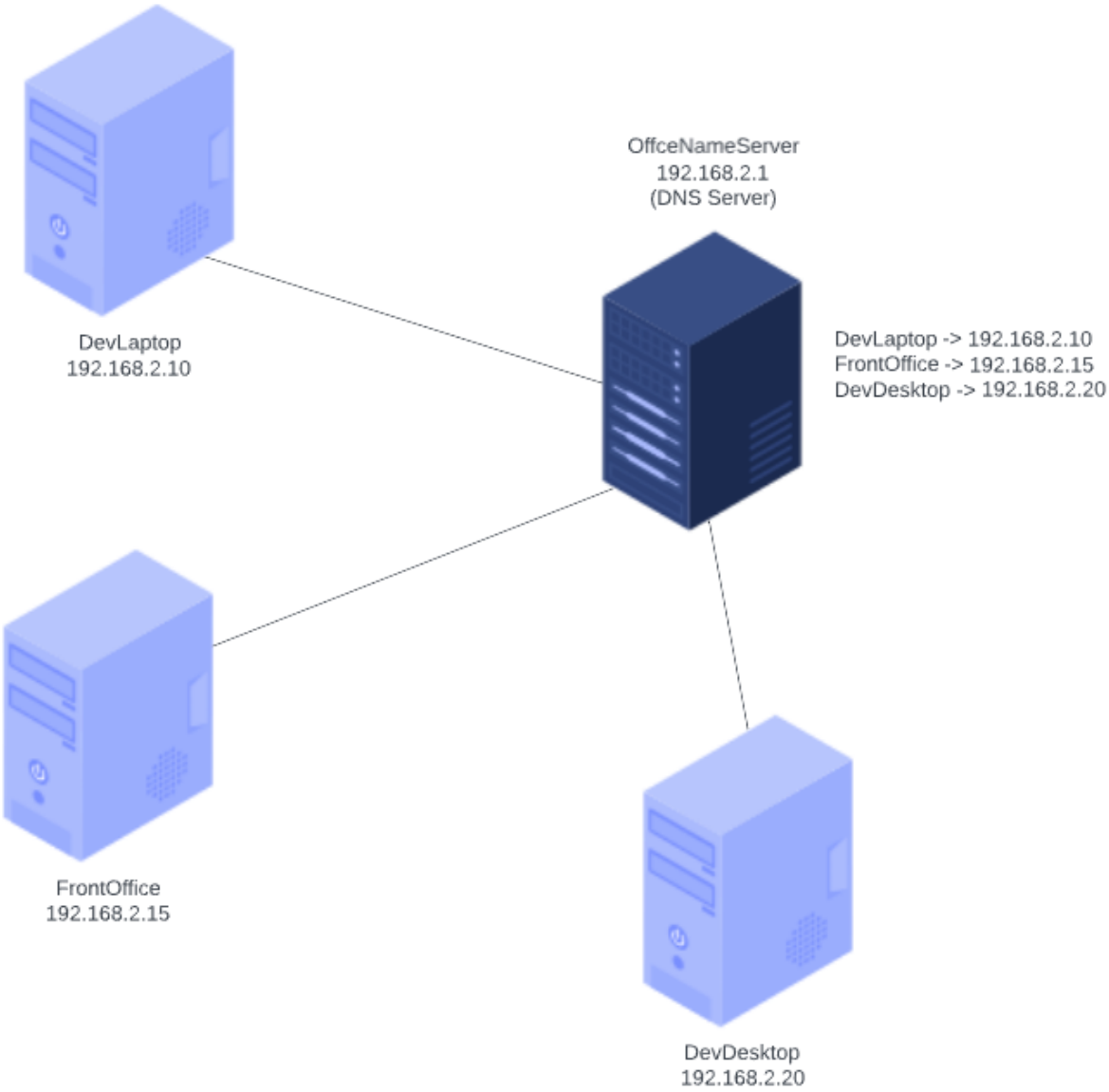


DNS on a small network

- **When your computer is part of a small home or corporate network your computer is assigned an IP address**
  - This is so that computers on the same network can communicate with each other
- **The network has a Domain Name Server (DNS)**
  - The DNS assigns ip addresses to each computer on the network
  - Common addresses are `192.168.2.1` to `192.168.2.255`
  - Run `ipconfig` to see what your network ip address is



- **The DNS can also assign aliases to each ip address on the network**
  - DevDbServer -> 192.168.2.15
  - TestDbServer -> 192.168.2.20


OffceNameServer
192.168.2.1
(DNS Server)
DevLaptop -> 192.168.2.10
FrontOffice -> 192.168.2.15
DevDesktop -> 192.168.2.20
DevLaptop
192.168.2.10
FrontOffice
192.168.2.15
DevDesktop
192.168.2.20

What about the Internet?

- **Each network that is connected to the internet also has an ip address that is unique**
  - Go to `https://whatsmyip.com` to discover your global IP address
- **If you know the IP address you can communicate with ANY computer that is connected to the internet**
- **Anyone can register a domain name to create a friendly, easy to remember name**
  - Use a registrar such as `https://godaddy.com` to purchase a domain name
  - Domain names are useful for websites, custom email addresses and more
- **When you register a domain name you can have it point to any ip address**
  - `my-custom-domain.com` -> `75.286.xx.xxx`
  - There are several global domain name servers that keep a list of all registered domains and where they point
- **Internet Service Providers maintain their own DNS servers**
  - They keep a local copy (cache) of domains and ip addresses

Communicating over the Internet

- **One computer can communicate with another computer on the internet**

  – The dev laptop can navigate to `www.my-custom-domain.com` if the domain is registered and pointing to the correct public ip address

Section 3–2

The Request Response Model

What is a Web Server

- **MySQL is a database server application that manages database access**
  - MySQL listens on port `3306` by default
  - The purpose of the database server is to give users access to data
- **A web server is an application that accepts incoming requests and responds to them**
  - The concept is similar to a database server
  - Web Servers generally listen either on port `80` or port `443`
  - The purpose of web servers is to give users access to web applications
- **Web Servers can host web sites**
  - www.pluralsight.com
  - www.espn.com
  - etc.

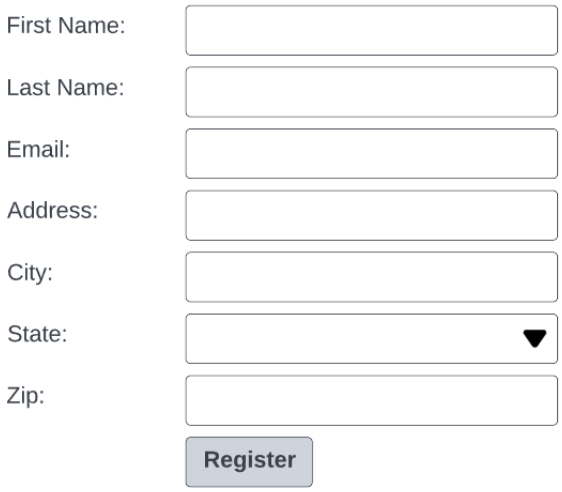
What's in a URL

- **The URL (Uniform Resource Locator) is made up of several parts that define the request**

- **Protocol - specifies how the data will be transmitted**
- **Domain - the domain name or ip address of the web server**
- **Port - the port number on which the server is listening**
  - Port 80 is the default port for http
  - Port 443 is the default port for https
  - If a default port is in use, you do not need to add it to the url
    * `http://www.my-custom-domain.com/path`
- **Path - the defined path for a resource on the server**
  - This often looks like a folder path to an html page
- **Query String - additional query information**
  - Key / value pairs of data passed to the server
  - These are often used like a filter in a WHERE clause of SQL

The Request

- **A web browser is an application that can request and view web pages**
  - The browser uses a URL to request a page
  - When the page is returned from the server the browser displays the result
- **The browser can make 2 types of requests**
  - A simple request to request information from the server
    * https://www.espn.com - simple request
  - Send information to save or update on the server
    * Registration information from a form



- **Form data is sent with the request**
- **The full request is a simple text stream that is made up of 2 parts**
  - header - contains the address of the resource you are requesting

* It also contains other bits of helper information that describes the request

– The body - this is optional information that is added at the end of the request

* It is a way to pass additional data or even attachments to the request

* If the request contains a body, it is added after a single empty line

Example

This is a simple request sent to the server. There is no body

```
GET https://www.my-custom-domain.com
```

Example

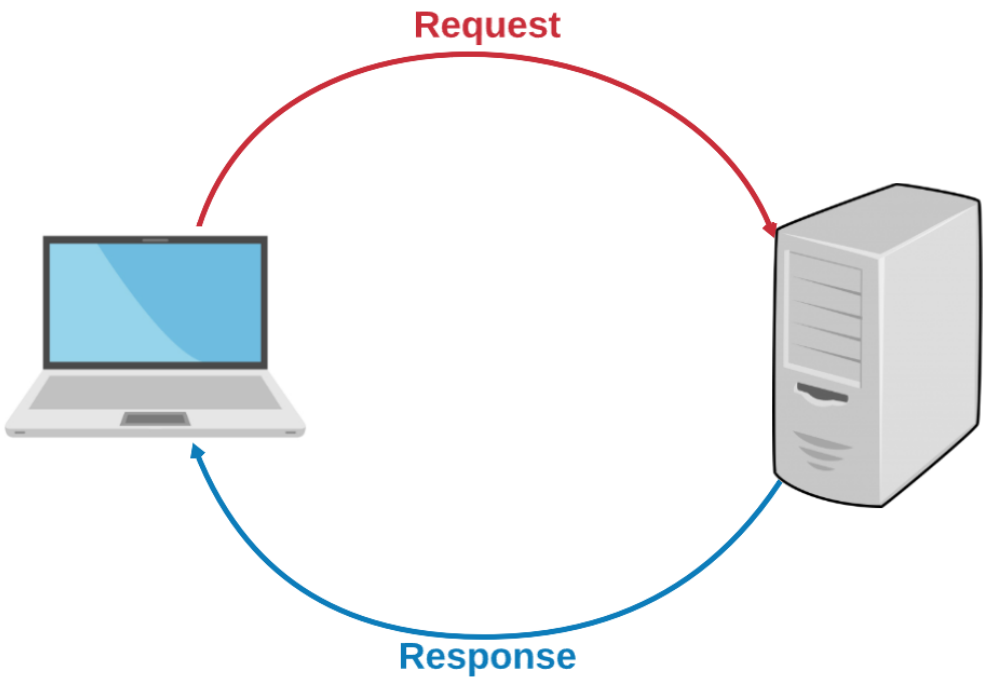
A request that sends form data in the body might look like this.

```
POST https://www.my-custom-domain.com/register
Content-Type: application/x-www-form-urlencoded

firstname=Carrie&lastname=Johnson&email=c.johnson@email.com
&address=123 Main Street&city=Salt Lake City&state=UT&zip=84111
```

Request and Response

- **The request / response programming model requires that the client initiates a request from the server**
  - In order to complete the request the client awaits a response

- **Each request is a new cycle**



Section 3–3

What is REST

Creating a RESTful Application

- **Web applications generally follow the request / response model**

- **REST is an architecture style of programming web applications**

  - REST - Representational State Transfer

  - Set of guidelines that is widely adopted (though not an official web standard)

- **REST is used to expose CRUD operations for resources on a server**

  - The verbs are the actions that resemble CRUD operations in SQL statements

    * GET = SELECT

    * POST = INSERT

    * PUT = UPDATE

    * DELETE = DELETE

- **Using rest to access resources**

  - The URL defines the resource you want to access

    * `https://www.my-custom-domain.com/`**`products`**

    * specifies access to the **products** resource on the web server

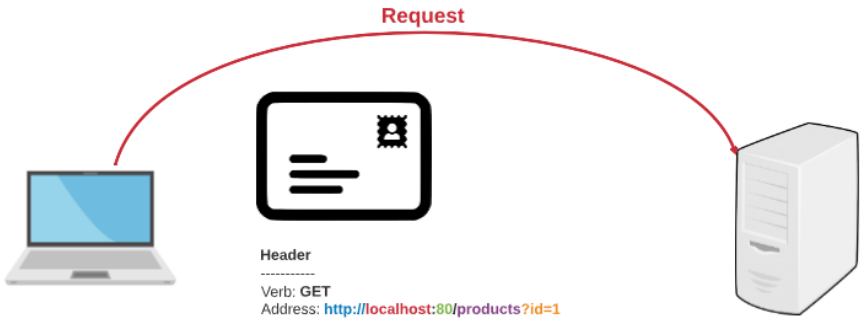
    * A resource often refers to a table in a database

  - The VERB defines what you want to do with the resource

GET

- When you GET a resource you are requesting one or more items

- A GET request never includes a body



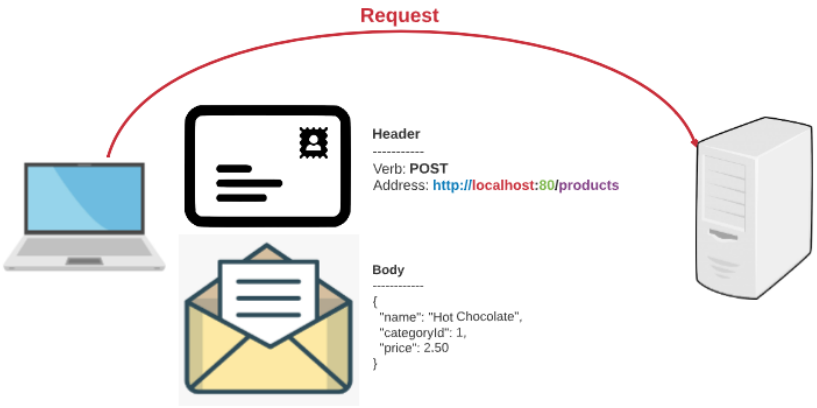
- The response includes the requested data in a body

- **The URL defines which resource you want to access**
  - `http://www.my-domain.com/`**`products`**
    * All products
  - `http://www.my-domain.com/`**`products?id=1`**
    * A single product with id 1
  - `http://www.my-domain.com/`**`products/1`**
    * A single product with id 1
    * Since finding a resource by id is extremely common this is standard alternative to using the query string
  - `http://www.my-domain.com/`**`products?category=beverages`**
    * All products in the beverage category
  - `http:/www.my-domain.com/products`**`?`**`category=beverages`**`&`**`price_gte=10`
    * You can specify multiple query string criteria by separating the key=value pairs with **&**
    * All products in the beverage category
    * AND price is greater than or equal to $10

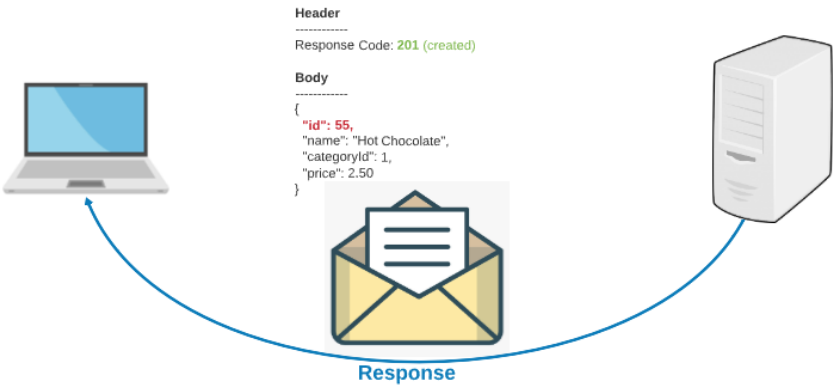
POST

- When you POST a resource you are inserting a new item to that resource

- A POST includes a body of the data that you want to insert



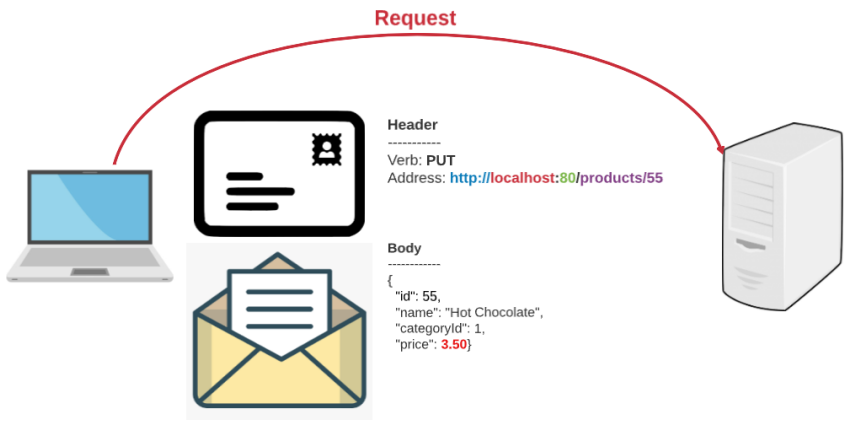
- The response includes the posted data in a body



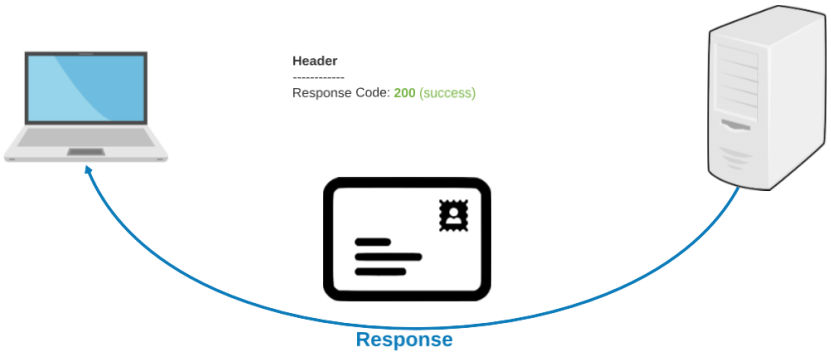
PUT

- When you PUT a resource you update an existing item in that resource collection

- A PUT includes the id of the resource you want to update, and a body of the data that you want to update



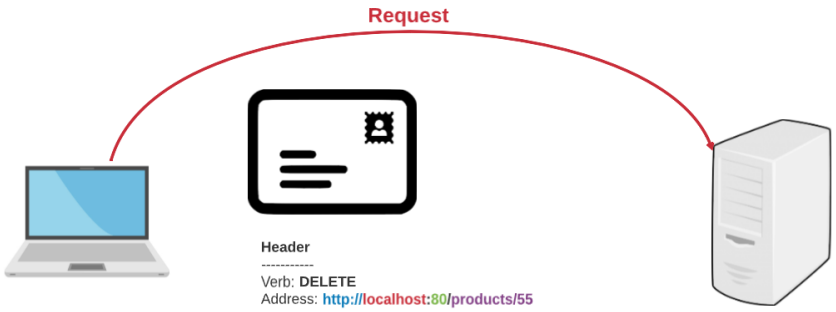
- The response DOES NOT include a body



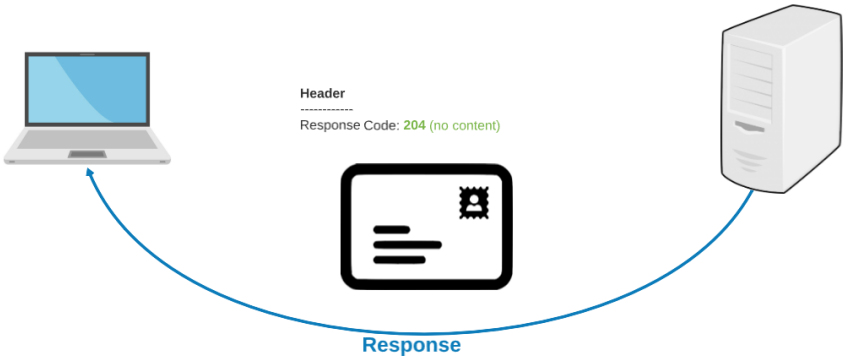
DELETE

- When you DELETE a resource you specify the id of the resource you wish to delete

- A DELETE does not include a body with the request



- The response DOES NOT include a body



Understanding the Request

- **The Verb and the URL in the request provide the critical information about what you want to accomplish**

- **Both the client and the server must agree on what the client expects and what the server provides**

- **Using REST provides standard practices so that developers have all of the information they need to program the server**

Section 3–4

CodeWars

CodeWars Kata

- **Walking in the Hallway**
  - As a security guard watching cameras, you need to figure out the minimum number of steps will be taken before two of the people meet

- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/6368426ec94f16a1e7e137fc/java

Module 4

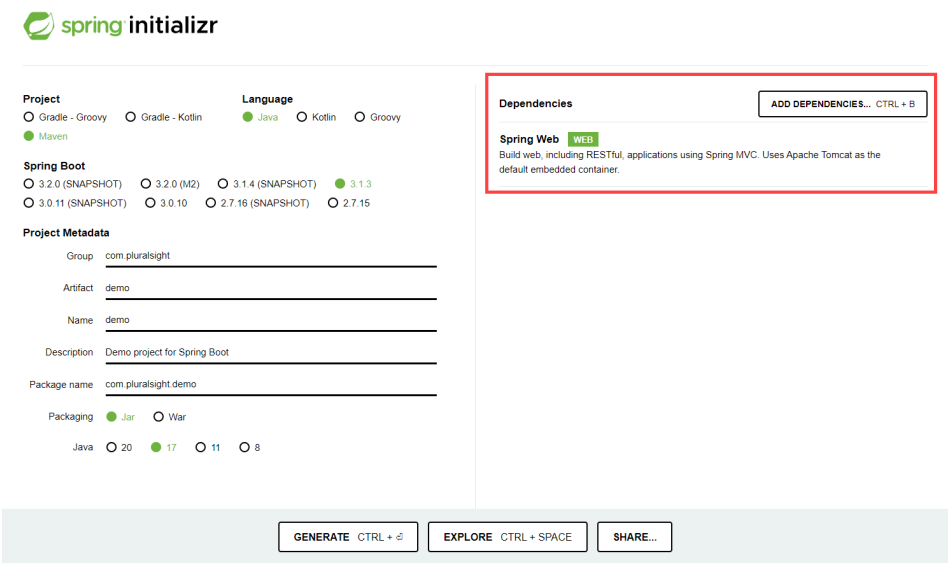
Building a Spring Boot Web API

Section 4–1

Creating a Web Server

Create a Spring Boot Web Server

- Go to `https://start.spring.io` and configure a new project by adding a Spring Web dependency

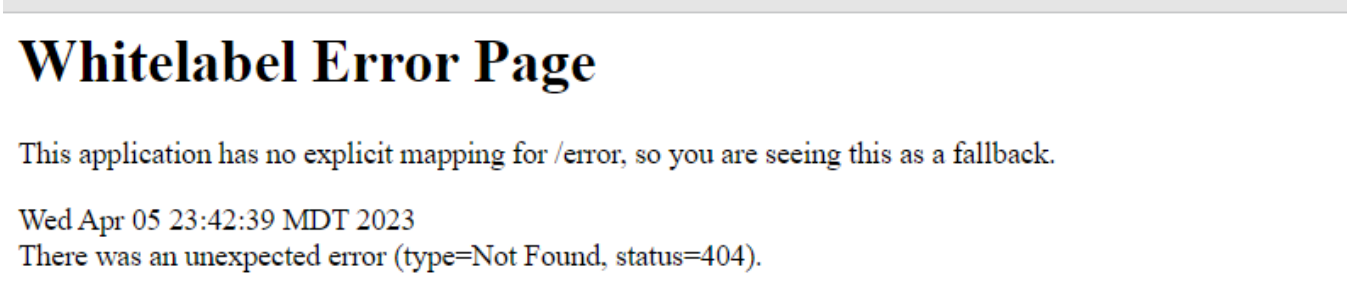


- Adding the web dependency converts your application into a web server

- When you run the application notice that it launches a web application and listens on port 8080

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.0.5)

2023-04-05T21:40:15.520-06:00  INFO 54232 --- [           main] com.yearup.apidemo.ApiDemoApplication    : Starting ApiDemoApplication using Java 17.0.6 with PID 54232 (C:\LearnToCo
2023-04-05T21:40:15.525-06:00  INFO 54232 --- [           main] com.yearup.apidemo.ApiDemoApplication    : No active profile set, falling back to 1 default profile: "default"
2023-04-05T21:40:16.250-06:00  INFO 54232 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2023-04-05T21:40:16.258-06:00  INFO 54232 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-04-05T21:40:16.258-06:00  INFO 54232 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.7]
2023-04-05T21:40:16.322-06:00  INFO 54232 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2023-04-05T21:40:16.323-06:00  INFO 54232 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 766 ms
```

- **Open a browser and navigate to `http://localhost:8080`**
  - The server cannot find the page



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Wed Apr 05 23:42:39 MDT 2023
There was an unexpected error (type=Not Found, status=404).

Responding to GET

- **In order to respond to a request we need to build a controller**
  - A controller is a class with pre-defined functionality that can respond http requests
  - Spring uses annotations to locate classes and functions to respond to web requests
- **@RestController is used to mark a class to be used for http requests**
- **@RequestMapping registers a path that a function will respond to**
  - path - specifies the resource path
  - method - specifies the verb that this method will respond to

Example

HomeController.java

```
@RestController
public class HomeController {

  // this method will respond to http://localhost:8080/
  @RequestMapping(path="/", method=RequestMethod.GET)
  public String index() {
    return "Hello World!";
  }
}
```

`http://localhost:8080 -> Hello World!`

- **Navigate to `http://localhost:8080`**

Hello World

Passing Query String Parameters

- **Often you need to pass additional information to the server**
  - You can add input parameters to your functions
  - Spring will map the query string variable to the function
    * The name of the query string parameter must match the input parameter name
    * input parameters are required unless you give them a `defaultValue` or mark them as `required=false`

Example

HomeController.java

```
@RestController
public class HomeController {

  // this method will respond to http://localhost:8080/
  @RequestMapping(path="/", method=RequestMethod.GET)
  public String index(
             @RequestParam(defaultValue="World") String name
             ) {
    return "Hello " + name + "!";
  }
}
```

```
http://localhost:8080               -> Hello World!
http://localhost:8080?name=Bob      -> Hello Bob!
http://localhost:8080?name=Cindy    -> Hello Cindy!
```

Exercises

EXERCISE 1

Create a new Java Maven project using Spring Boot. Open a browser and navigate to `https://start.spring.io`.

Configure your project with the following settings:

```
Selections:  Maven Project
             Java
             Latest stable Spring Boot version (no snapshot)
             Group: com.pluralsight
             Artifact: NorthwindTradersAPI
             Jar
             Java Version 17
             Dependencies: Spring Web
```

Create a `HomeController`. Add a method to handle the default "home page" request (`http://localhost:8080`). Return "Hello World" and test the application.

Modify your default method to accept an input parameter named country. The user should be able to enter a url with a query string that includes country (`http://localhost:8080?country=USA`).

Update the logic to display "Hello <country>" if the user included the country in the URL or "Hello World" if they did not. Test the application again, both with and without the query string.

Commit and push your code!

Section 4–2

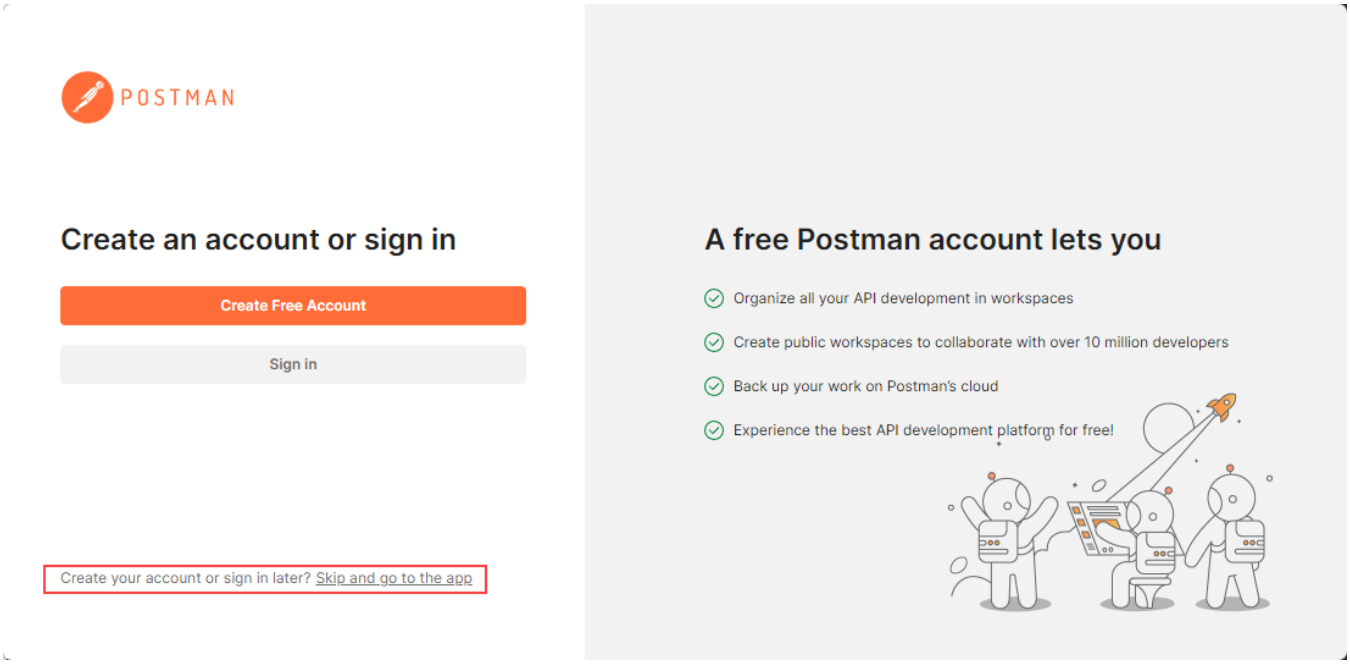
Using Postman

What is Postman

- **RESTful services support GET, POST, PUT and DELETE verbs**

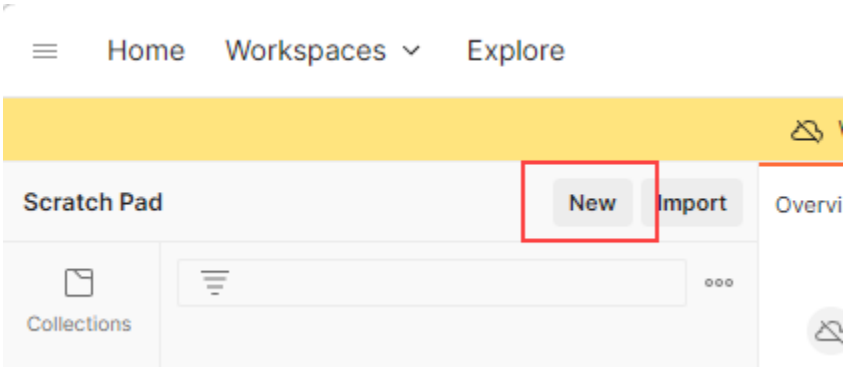
- **Browsers only directly support GET and POST methods**
  - This makes it difficult to test your RESTful APIs with a browser
    * You could create a web application with JavaScript functions to test the API
    * You could create a desktop application using Java, C# or another programming language

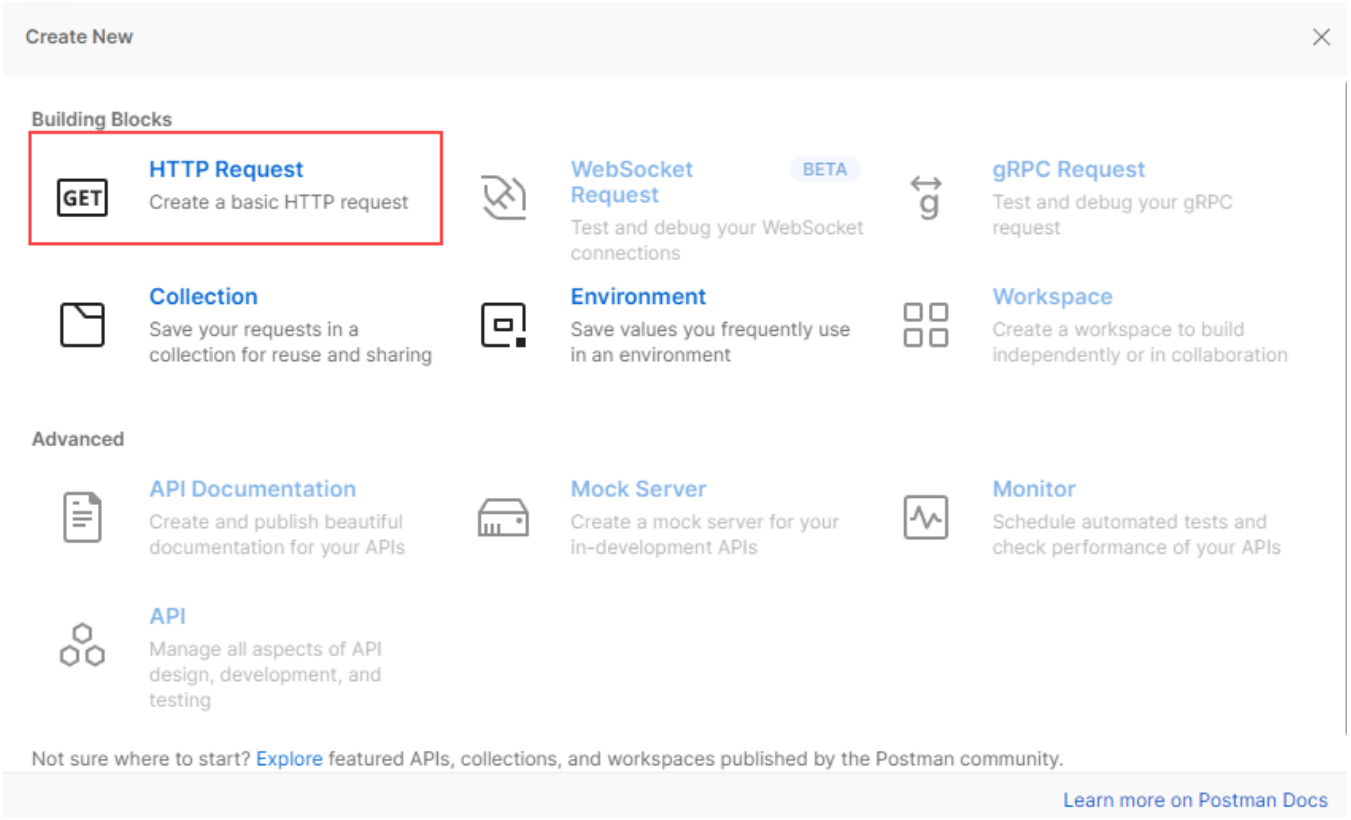
- **Google developed Postman as a tool to make it easy to test your APIs**
  - Postman will prompt you to create an account and sign in, but you do not need an account to use it



Using Postman

- **To test an API you need to create a new HttpRequest**

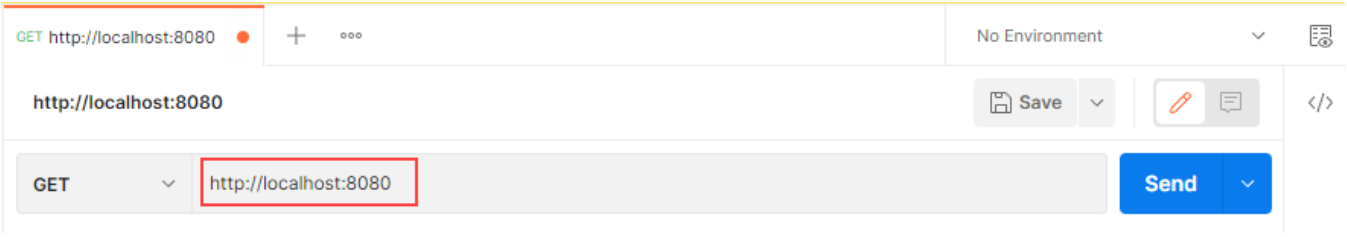




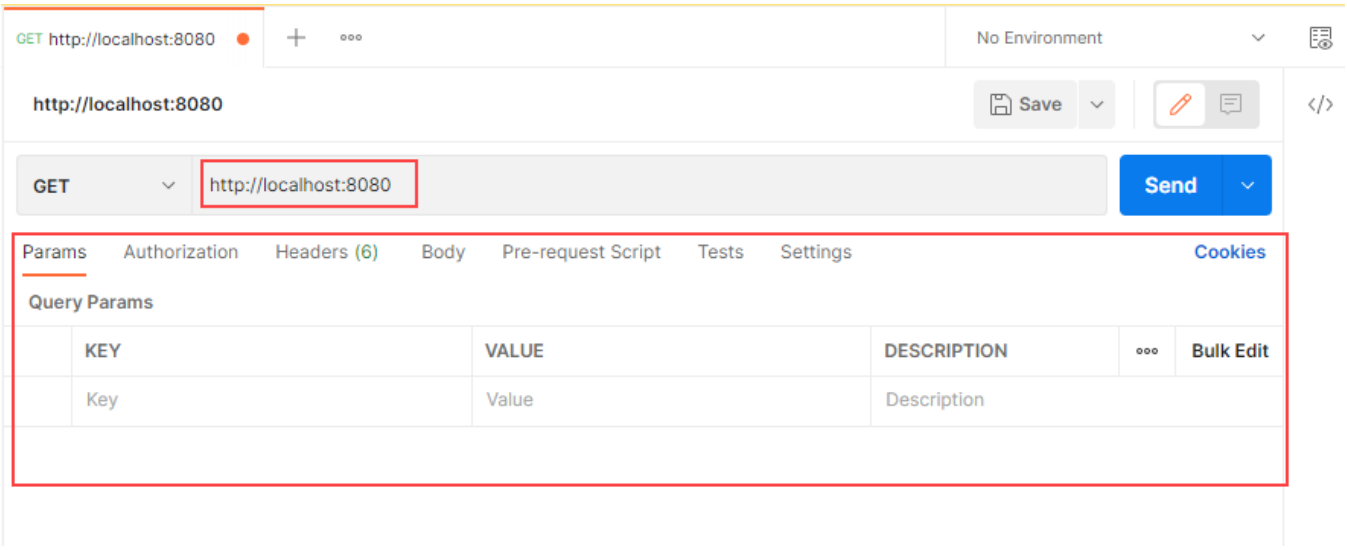
Making the Request

- **An HttpRequest in Postman is similar to navigating to a website in your browser**

- **For a basic request just add the address (url) of your API**
  - Ensure that the API is running before you click Send

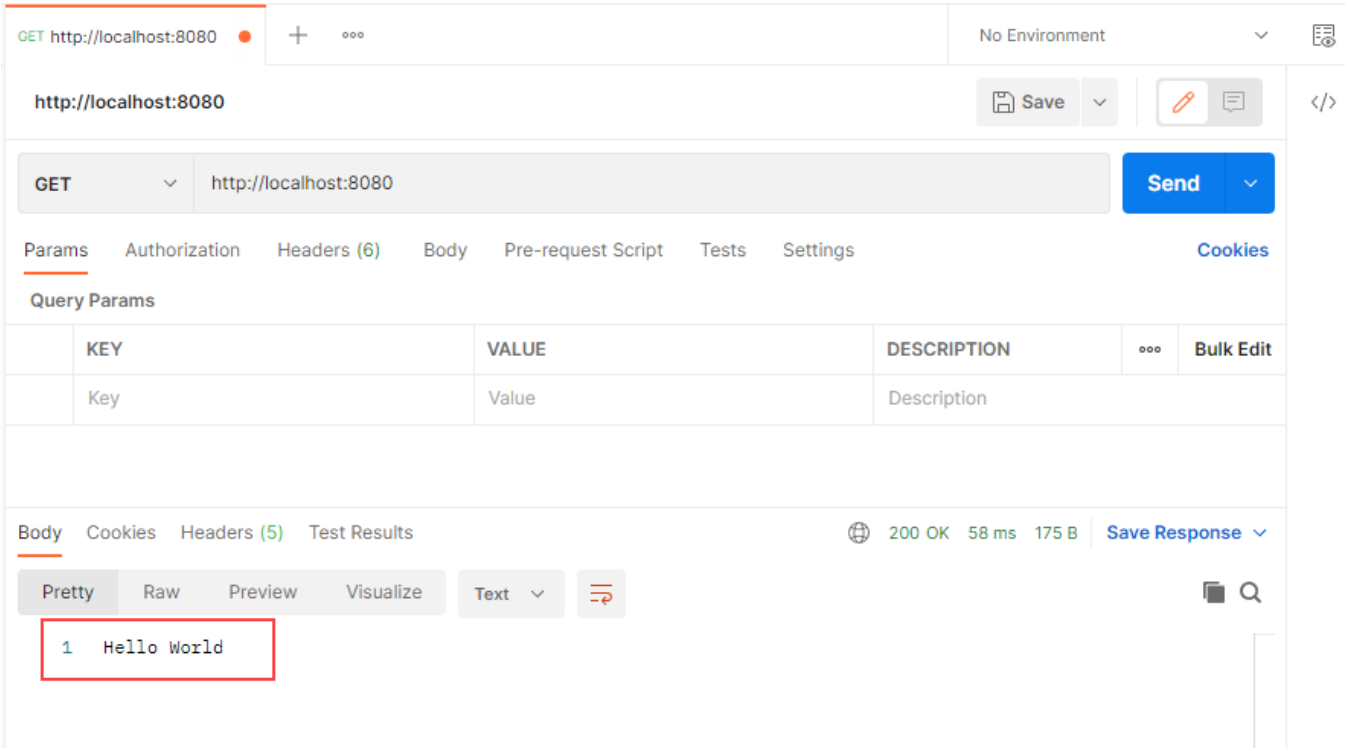


- **Below the URL you can configure the information that you want to SEND TO the server with your request**



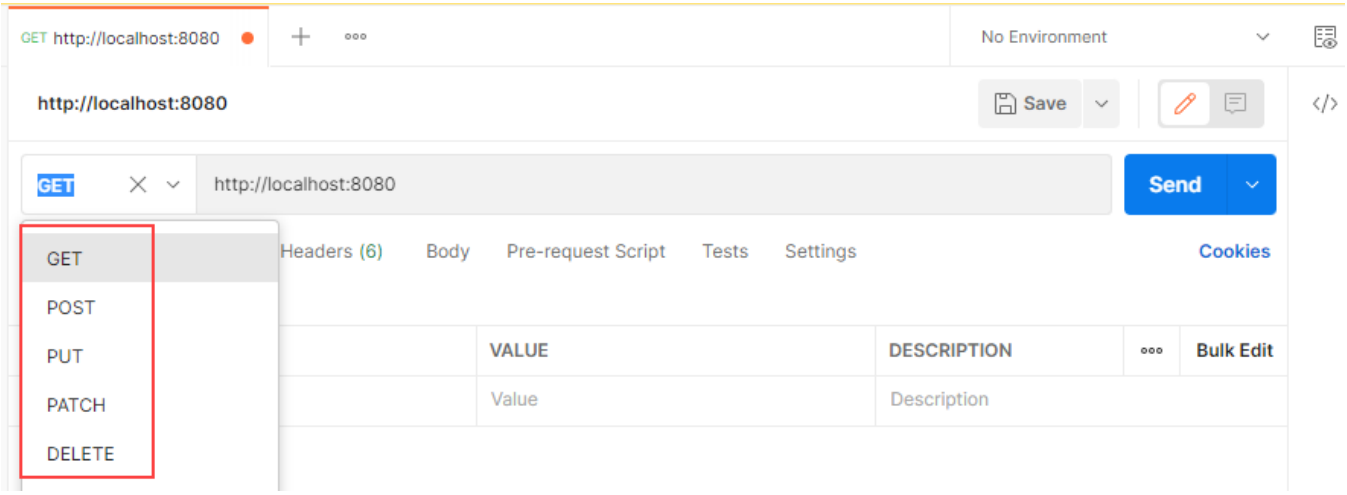
Receiving the Response

- When the server responds Postman will display the result of your request



Specifying the Verb

- **Postman allows you to specify the verb of the request**
  - GET, POST, PUT, DELETE etc.



- **As we develop APIs throughout this workbook, you will use various verbs with Postman**

Adding a Body to the Request

- **`POST` and `PUT` requests are accompanied by a request body**
- **To add a body select the Body tab and enter the data**
  - To send JSON you must set the body type to raw
  - Select JSON from the text type drop down
  - Enter the JSON data in the text

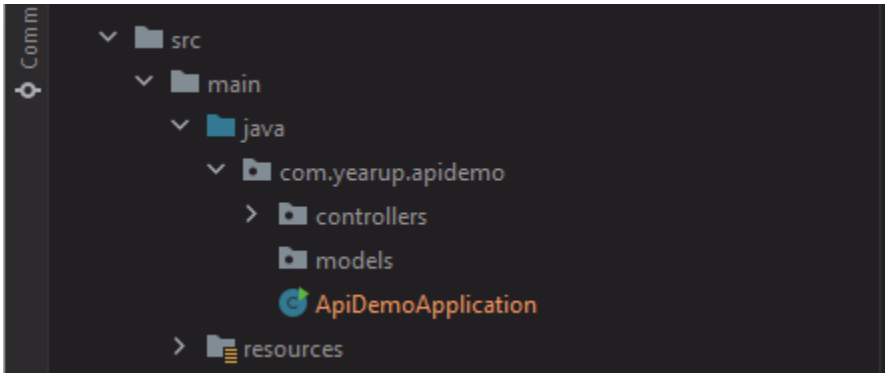
Section 4–3

The MVC Pattern

The MVC Pattern

- **MVC stands for Model View Controller**
- **It is a popular design pattern for web application development**
- **Spring Boot receives the incoming request**
  - Parses the URL and determines where to route the request
  - `http://localhost:8080/products` -> routes to **`/products`**
- **A Controller is a class that receives requests from Spring Boot**
  - Controllers are the entry point into the application
- **A Model is a class that represents a resource or data on the server**
  - Customer, Product, Order, Category
- **A View is the logic that creates a beautiful User Interface with the data**
  - In an API we do not put any effort into the view
  - We are only concerned about the data

- **It is common to create packages for any models, controllers and views in your project**



The Model

- **A model represents a resource or object that your application manages**

Example

`Customer.java`

```
public class Customer {

  private int id;
  private String name;
  private String address;

  public Customer(){}

  public int getId(){
    return id;
  }

  public String getName(){
    return name;
  }

  public String getAddress(){
    return address;
  }

  public void setId(int id){
    this.id = id;
  }

  public void setName(String name){
    this.name = name;
  }

  public void setAddress(String address){
    this.address = address;
  }
}
```

The Controller

- **A controller is a container for functions that are publicly exposed to the web**

- **A well designed Controller should encapsulate only highly cohesive methods**

  – A good practice is to create 1 controller per resource

  – All CRUD operations for that resource are then contained in a single class

Example

`CustomerController.java`

```
@RestController
public class CustomerController {

  // respond to http://localhost:8080/customers
  @RequestMapping(path="/customers", method=RequestMethod.GET)
  public List<Customer> getCustomers() {

    List<Customer> customers = new ArrayList<>();

    customers.add(new Customer(...));
    customers.add(new Customer(...));
    customers.add(new Customer(...));

    return customers;
  }
}
```

- **Navigate to** `http://localhost:8080/`**`customers`**

  – The list of customers is returned as an array of JSON data


localhost:8080/customers
localhost:8080/customers
Raw
Parsed
[
{
"id": 5,
"name": "John",
"address": "123 Main Street"
},
{
"id": 6,
"name": "Grace",
"address": "521 State"
},
{
"id": 7,
"name": "Alfred",
"address": "19 Hwy 82"
}
]

Requesting a Resource by Id

- **You saw earlier that we can pass parameters by using the query string**
  - `http://localhost:8080/customers?name=Alfred`

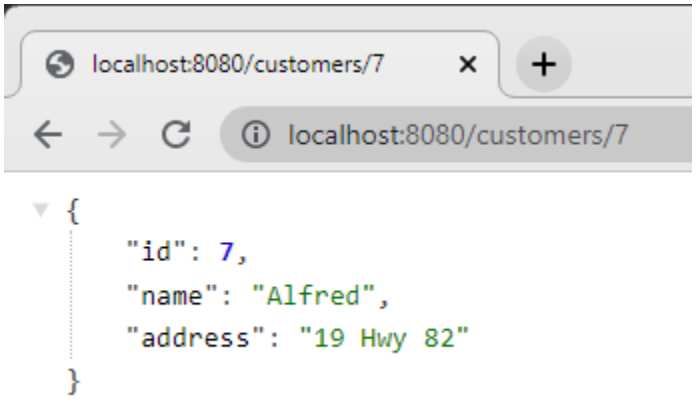
Example

```
@RequestMapping(path="/customers",method=RequestMethod.GET)
public List<Customer> getCustomers(
                    @RequestParam(required=false) String name
                   )
{
  // your logic goes here
}
```

- **But it is most common to find a single resource by its id**
  - `http://www.localhost:8080/customers/7`
  - Spring can pull a value from the path as well
  - To read a value from the path instead of the query string you must use the **`@PathVariable`** annotation
  - The path variable definition must match the name of the input parameter

Example

```
@RequestMapping(path="/customers/{id}",method=RequestMethod.GET)
public List<Customer> getCustomer(@PathVariable int id)
{
  // your logic goes here
}
```



Exercises

EXERCISE 2

Continue working with the NorthwindTradersAPI project.

Create a models package and add 2 models to the project - Product and Category.

```
Product
------------
productId
productName
categoryId
unitPrice

Category
------------
categoryId
categoryName
```

Add a controllers package and add 2 Controllers
`ProductsController`
`CategoriesController`

Add public methods to the controllers to return all products and all categories. Do not connect to a database yet, just create a temporary list of products and categories for now (we will connect to the database shortly)

`http://localhost:8080/products` should return a list of all products
`http://localhost:8080/products/5` should return a specific product
`http://localhost:8080/categories` should return a list of all categories
`http://localhost:8080/categories/17` should return a specific category

Bonus (optional):

Modify your `ProductsController` to allow a user to filter for products by name, categoryid or price

Modify your `CategoriesController` to allow a user to filter for categories by name

Commit and push your code!

Section 4–4

Dependency Injection with Spring MVC

Spring Boot MVC Applications

- **Spring Boot Web Applications do not include a CommandLineRunner class**
  - Web applications do not interact with the console
  - But Spring still manages all Beans and give you the same access to the IoC container
- **Controllers are managed by the ApplicationContext**
  - When an HTTP Request is received the ApplicationContext determines which controller should handle it
  - Spring creates the controller and calls the appropriate method
  - Spring **injects** any dependencies that the controller requires
    * All `@Autowired` variables / constructors are resolved with a Spring Bean

Configure the Dependencies

- **Add necessary dependencies to the pom.xml**

Example

pom.xml

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.32</version>
</dependency>
<dependency>
     <groupId>org.apache.commons</groupId>
     <artifactId>commons-dbcp2</artifactId>
     <version>2.9.0</version>
</dependency>
```

- **Configure your application properties**

Example

application.properties

```
datasource.url=jdbc:mysql://localhost:3306/northwind
datasource.username=root
datasource.password =P@ssw0rd
```

Create your DatabaseConfiguration

`DbConfiguration.java`

This creates a **dataSource Bean** that is injectable

```
@Configuration
public class DbConfiguration
{
  private BasicDataSource basicDataSource;

  @Bean
  public DataSource dataSource(){
    return basicDataSource;
  }

  public DbConfiguration(@Value("${datasource.url}") String url,
              @Value("${datasource.username}") String username,
              @Value("${datasource.password}") String password
  ) {
    basicDataSource = new BasicDataSource();
    basicDataSource.setUrl(url);
    basicDataSource.setUsername(username);
    basicDataSource.setPassword(password);
  }
}
```

Create the JDBC implementation of your DAO

Example

JdbcCustomerDao.java

This class has a dependency on a DataSource that is Autowired. It is also a @Component - which means it is a bean that can be injected.

```
@Component
public class JdbcCustomerDao implements CustomerDao{
  private DataSource dataSource;

  @Autowired
  public JdbcCustomerDao(DataSource dataSource){
    this.dataSource = dataSource;
  }

  @Override
  public List<Customer> getAll(){
    List<Customer> customers = new ArrayList<>();

    String sql = "SELECT * FROM Customers;";

    try(Connection connection = dataSource.getConnection()){
      Statement statement = connection.createStatement();
      ResultSet rows = statement.executeQuery(sql);
      while(rows.next()){
        Customer customer = new Customer();
        customer.setId(rows.getString("CustomerID"));
        customer.setName(rows.getString("CompanyName"));
        customer.setAddress(rows.getString("Address"));
        customers.add(customer);
      }
    }
    catch (SQLException e){ System.out.println(e); }
    return customers;
  }

  @Override
  public Customer findByCustomerId(String id){
    // code to get one customer here
  }
}
```

Update the Controller to use the JDBC DAO

- **Your CustomerDao can now be injected into your controller**
  - Add a class level variable for the CustomerDao
  - Use the dao throughout the class

Example

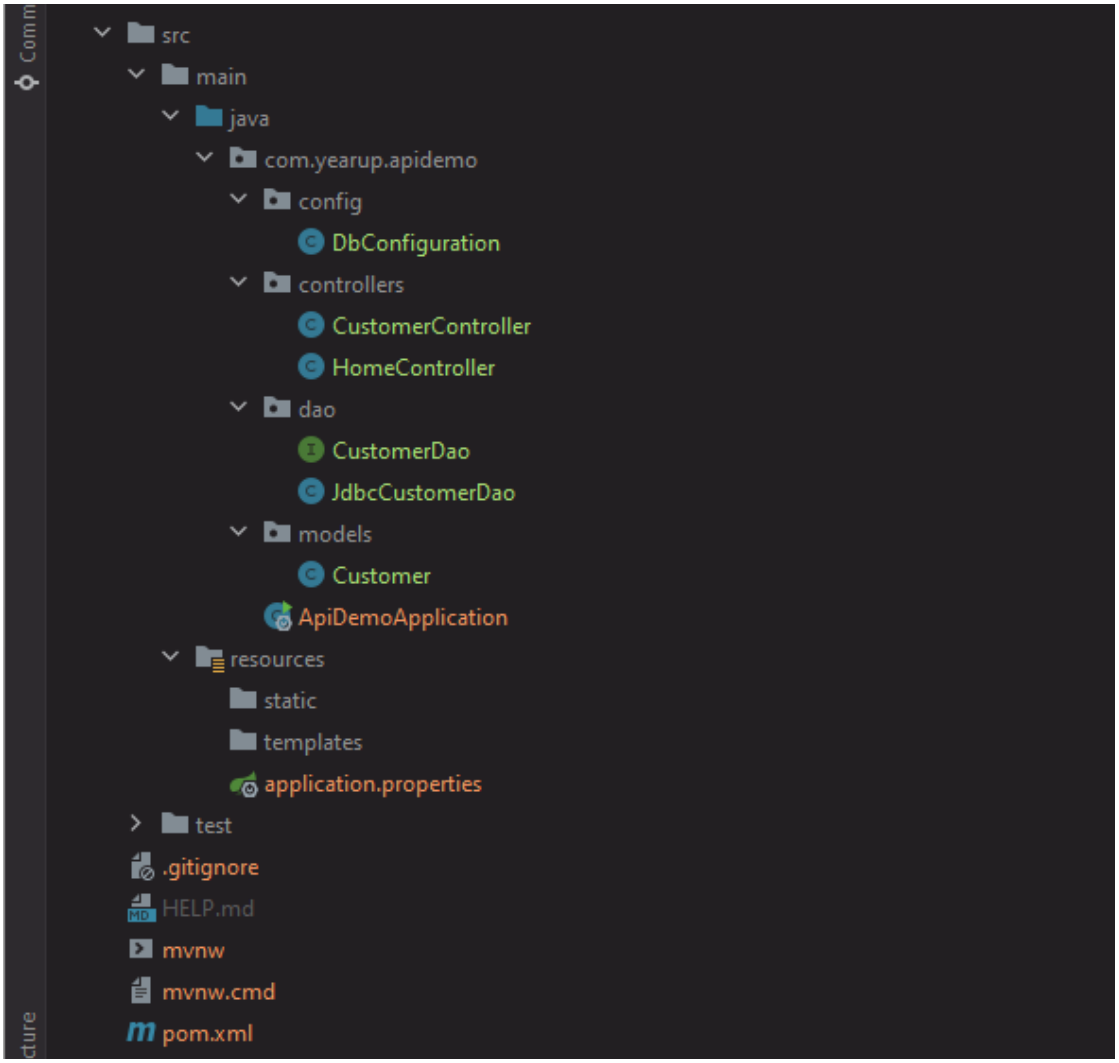
JdbcCustomerDao.java

This class has a dependency on a DataSource that is Autowired. It is also a @Component - which means it is a bean that can be injected.

```
@RestController
public class CustomerController
{
  private CustomerDao dao;

  @Autowired
  public CustomerController(CustomerDao dao)
  {
    this.dao = dao;
  }

  @RequestMapping(path="/customers", method = RequestMethod.GET)
  public List<Customer> getCustomers()
  {
    var customers = dao.getAll();
    return customers;
  }

  @RequestMapping(path="/customers/{id}",
                  method = RequestMethod.GET)
  public Customer getCustomer(@PathVariable String id)
  {
    Customer customer = dao.findByCustomerId(id);
    return customer;
  }
}
```

The Application File Structure

- **The final application structure looks like this**



Exercises

EXERCISE 3

Continue working with the NorthwindTradersAPI project. You will modify your application to interact with the Northwind Database.

Step 1

Add the MySQL and BasicDataSource dependencies to your `pom.xml`. Then add your database connection information to your `application.properties` file.

Create a DatabaseConfiguration class and register a bean for your data source.

Step 2

Create DAO interfaces for both Product and Category.

```
ProductDao
------------
List<Product> getAll()
Product getById(int id)

CategoryDao
------------
List<Category> getAll()
Category getById(int id)
```

Create JDBC implementations of your interfaces. Both should be annotated with `@Component`, and both should require a `DataSource` as an input parameter.

```
JdbcProductDao
JdbcCategoryDao
```

Step 3

Update your ProductsController to inject the ProductDao bean through constructor injection. Update your controller methods to use your ProductDao to pull data from the database.

Make the same changes with your Categories controller.

Test your changes in Postman.

Commit and push your code!

Section 4–5

CodeWars

CodeWars Kata

- **Speed Limit**
  - Calculate the total amount of the fine for a driver as they drive through multiple speed limit zones

- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/635a7827bafe03708e3e1db6/java

Module 5

POST, PUT, DELETE

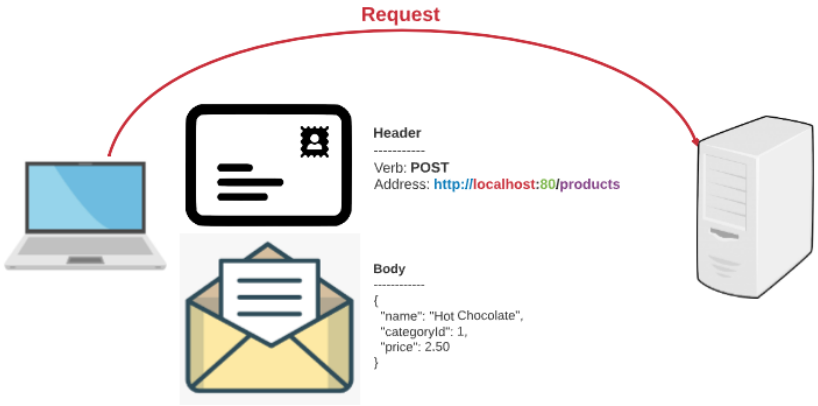
Section 5–1

Adding a new Resource

POST

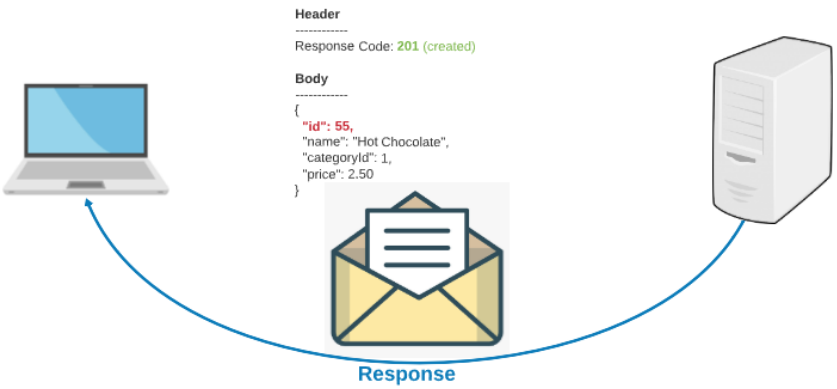
- **Adding or inserting a new record should be done through a POST request**

- **When a client POSTs a new record - they must include the information that will be inserted in the body**



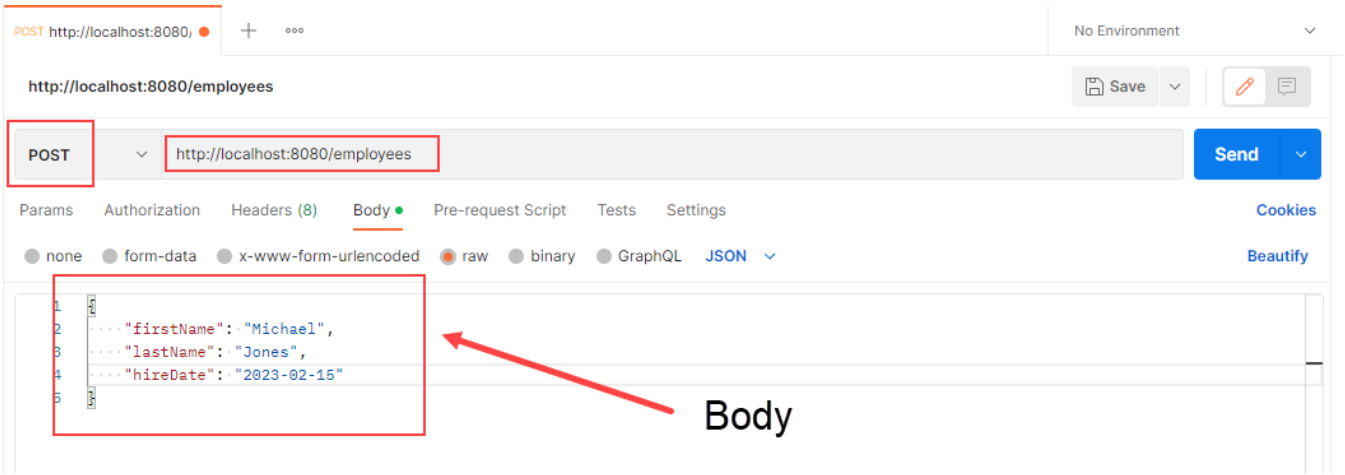
- **The server is responsible for saving that data**

  – The server should respond with the full resource, including the new ID that was generated



POST Example Client

- **The Client requests to POST (insert) a new record**
  - Using Postman as a client



Responding to a POST

- **When a client makes a request, just like a GET the server directs the request to a `Controller`**

- **The method responsible for handling the request must be marked with a `RequestMapping` annotation to handle the `POST`**

  - The body of the request is passed as a method parameter

    * You must use the `@RequestBody` annotation to access the body

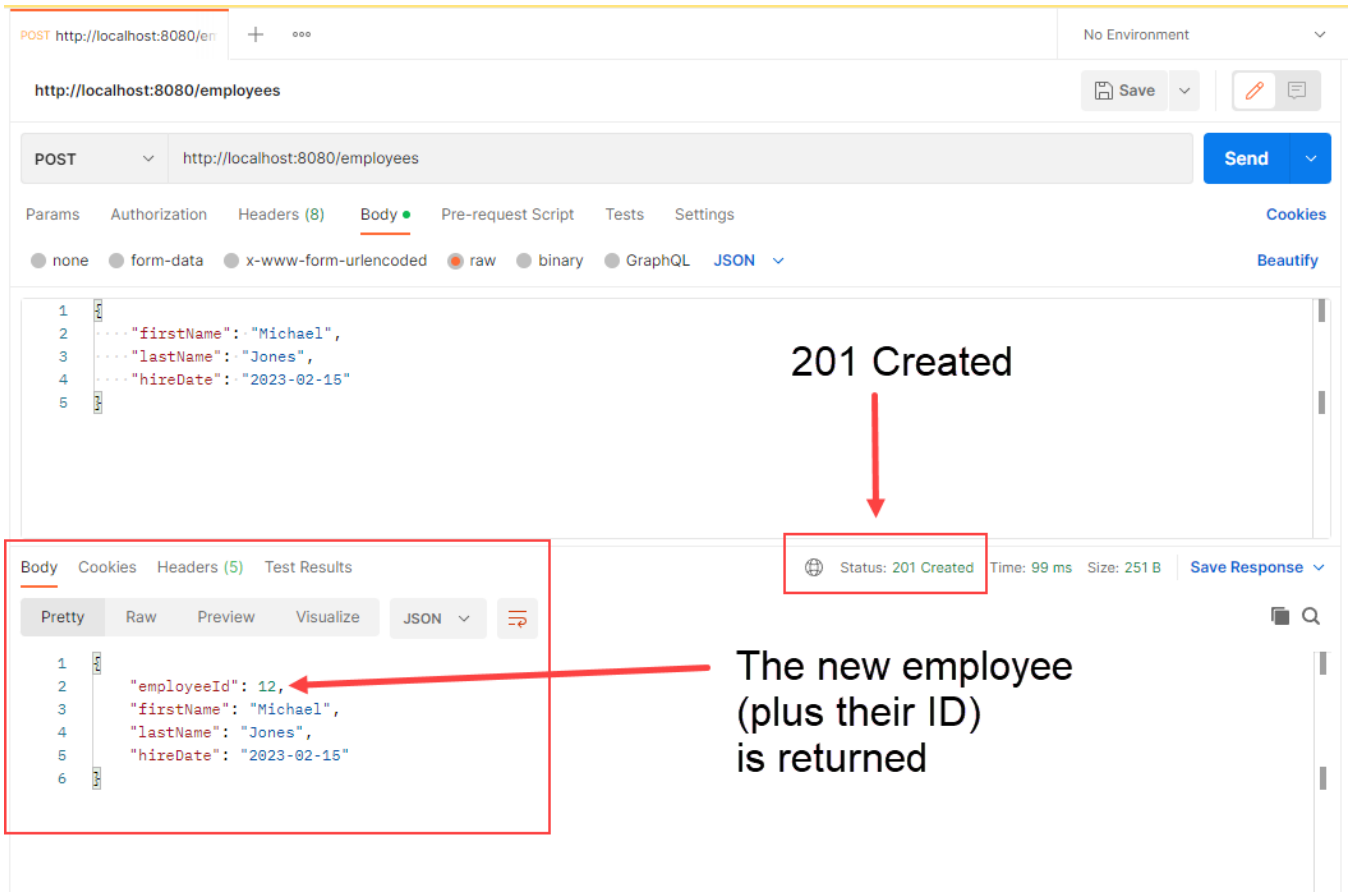
  - The `@ResponseStatus` annotation specifies the status code that will be returned

    * `HttpStatus.CREATED` returns a status of `201`

Example

```
@RequestMapping(path="/employees",method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.CREATED)
public Employee addEmployee (
                    @RequestBody Employee employee
                   )
{
  // the insert method of the DAO should return
  // a Employee object with the new id that was generated
  Employee newEmployee = employeeDao.insert(employee);

  // return the new supplier object
  return newEmployee;
}
```

- **The client then receives the following response**



Exercises

EXERCISE 1

Continue to work with the NorthwindTradersAPI project. You will modify the API to allow users to add products and categories.

Step 1

Modify the `ProductDao` and `CategoryDao` to add the following methods

```
ProductDao
------------
Product insert(Product product);

CategoryDao
------------
Category insert(Category category)
```

Implement the changes that you made to your interfaces in the JdbcDataSources.

```
JdbcProductDao
JdbcCategoryDao
```

Step 2

Modify the `ProductsController` to add a public web function to add a new product.

Add the `@RequestMapping` annotation for `POST`

Use the `ProductDao` to insert the record into the database. Test your code using Postman.

Step 3

Modify the `CategoriesController` to add a public web function to add a new category.

Add the `@RequestMapping` annotation for `POST`

Use the `CategoryDao` to insert the record into the database. Test your code using Postman.

Commit and push your code!

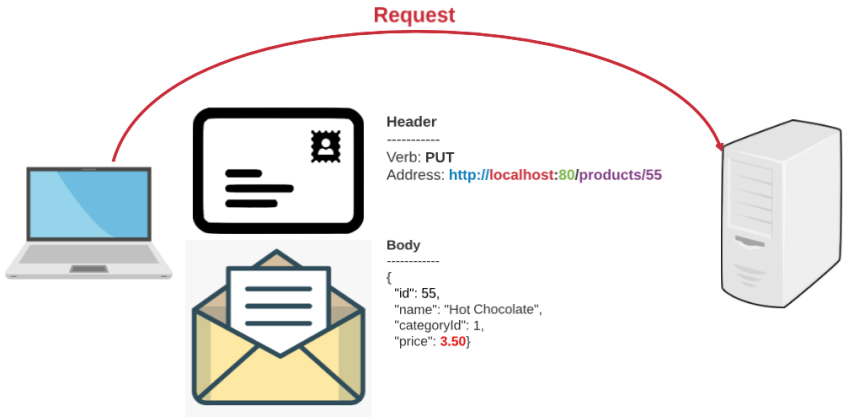
Section 5–2

Updating Resources

PUT

- **Updating a record should be done using a `PUT` request**

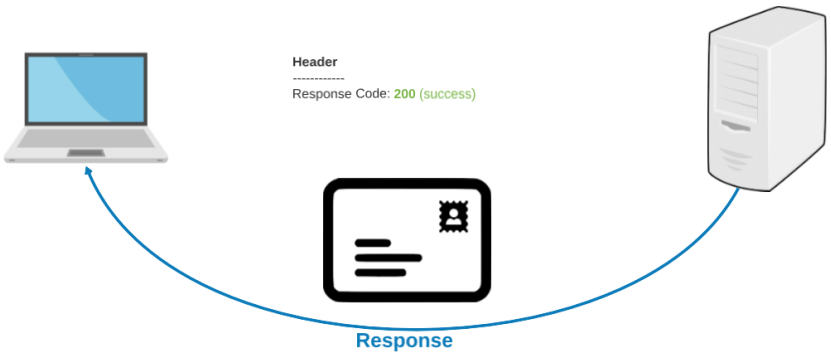
- **The url of a `PUT` includes the id of the resource you want to update, and a body of the data that you want to update**



- **The response usually DOES NOT include a body**

  – This is different from a POST because on insert the client needs to know the new id that was generated

  – When updating the client already knows the id, and just needs confirmation that the update succeeded



PUT Example Client

- **The Client requests to PUT (update) an existing record**
  - Using Postman as a client

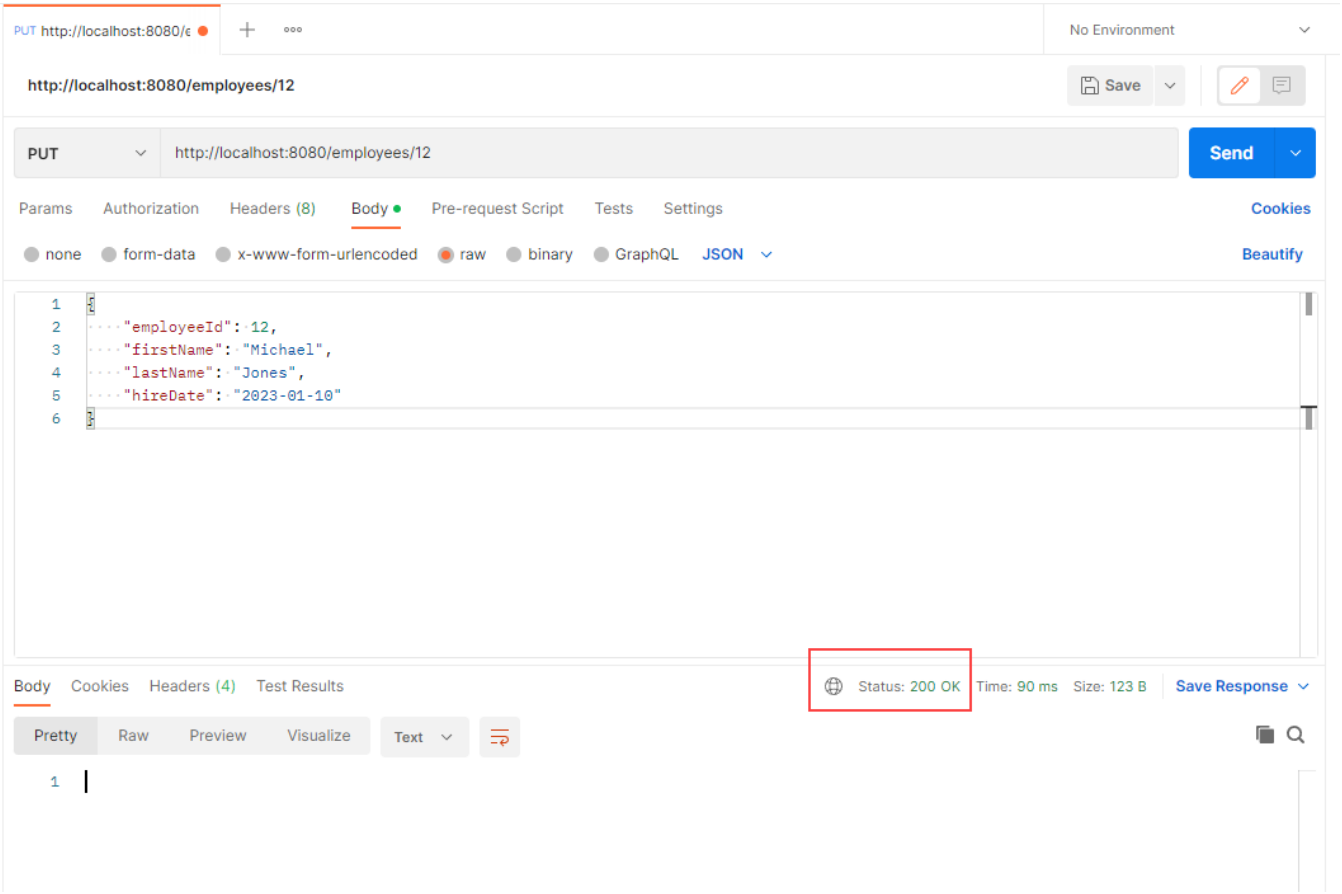
Responding to a PUT

- **The method responsible for handling the request must be marked with a `RequestMapping` annotation to handle the `PUT`**

Example

```
@RequestMapping(path="/employees/{id}",method=RequestMethod.PUT)
public void updateEmployee (
                    @PathVariable int id,
                    @RequestBody Employee employee
                   )
{
  // the update method of the DAO should accept the id
  // and the new employee information to update
  employeeDao.update(id, employee);
}
```

- **The client then receives the following response**



Exercises

EXERCISE 2

Continue working on the NorthwindTradersAPI project. You will modify the application to allow users to update products and categories.

Step 1

Modify the `ProductDao` and `CategoryDao` to add the following methods

```
ProductDao
------------
void update(int id, Product product);

CategoryDao
------------
void update(int id, Category category)
```

Implement the changes that you made to your interfaces in the JdbcDataSources.

```
JdbcProductDao
JdbcCategoryDao
```

Step 2

Modify the `ProductsController` to add a public web function to update a product.

Add the `@RequestMapping` annotation for `PUT`

Use the `ProductDao` to update the record in the database. Test your code using Postman.

Step 3

Modify the `CategoriesController` to add a public web function to update a category.

Add the `@RequestMapping` annotation for `PUT`

Use the `CategoryDao` to update the record in the database. Test your code using Postman.

Commit and push your code!

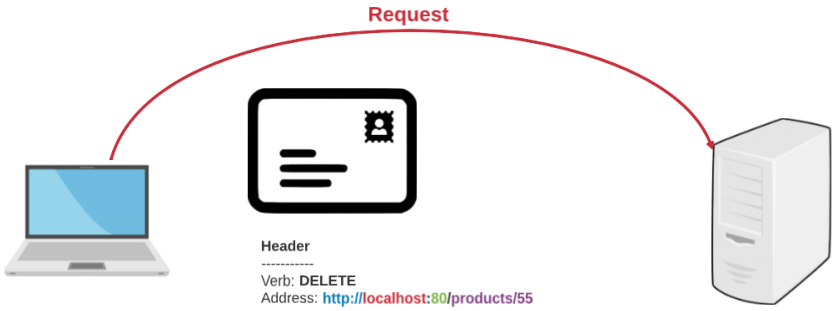
Section 5–3

Deleting Resources

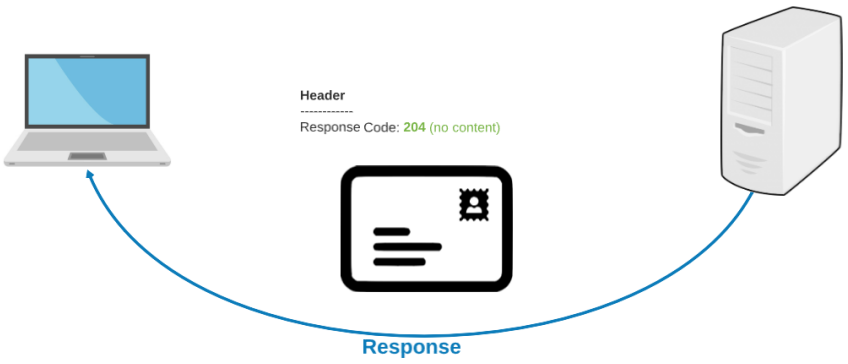
DELETE

- **Deleting records should be completed using a DELETE request**

- **A DELETE does not include a body with the request**
  - The client just sends the id of the record that is being deleted
    * The id should be included in the URL

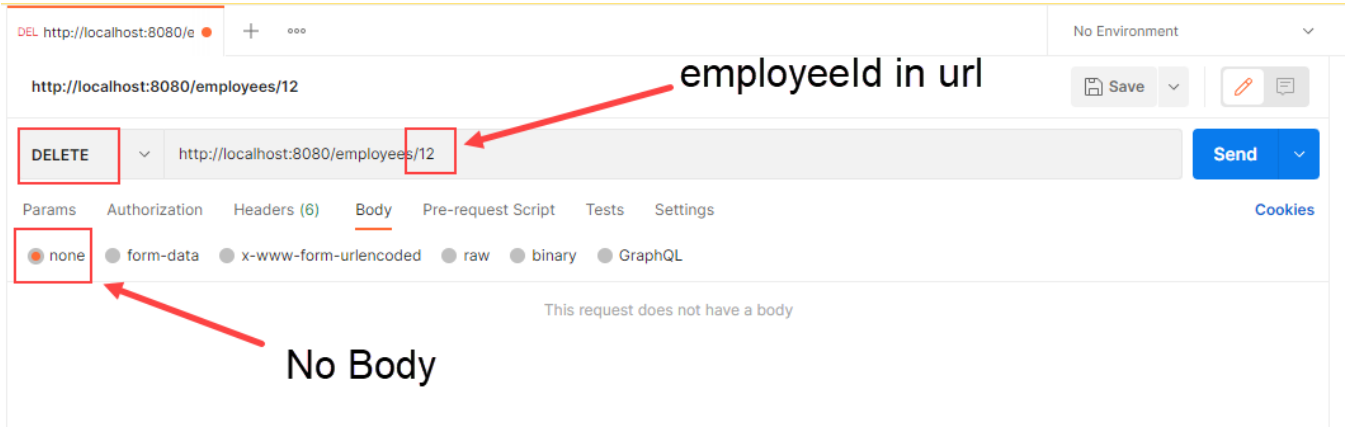


- **The response DOES NOT include a body**



DELETE Example Client

- **The Client requests to PUT (update) an existing record**
  - Using Postman as a client



Responding to a DELETE

- **The method responsible for handling the request must be marked with a `RequestMapping` annotation to handle the `DELETE`**

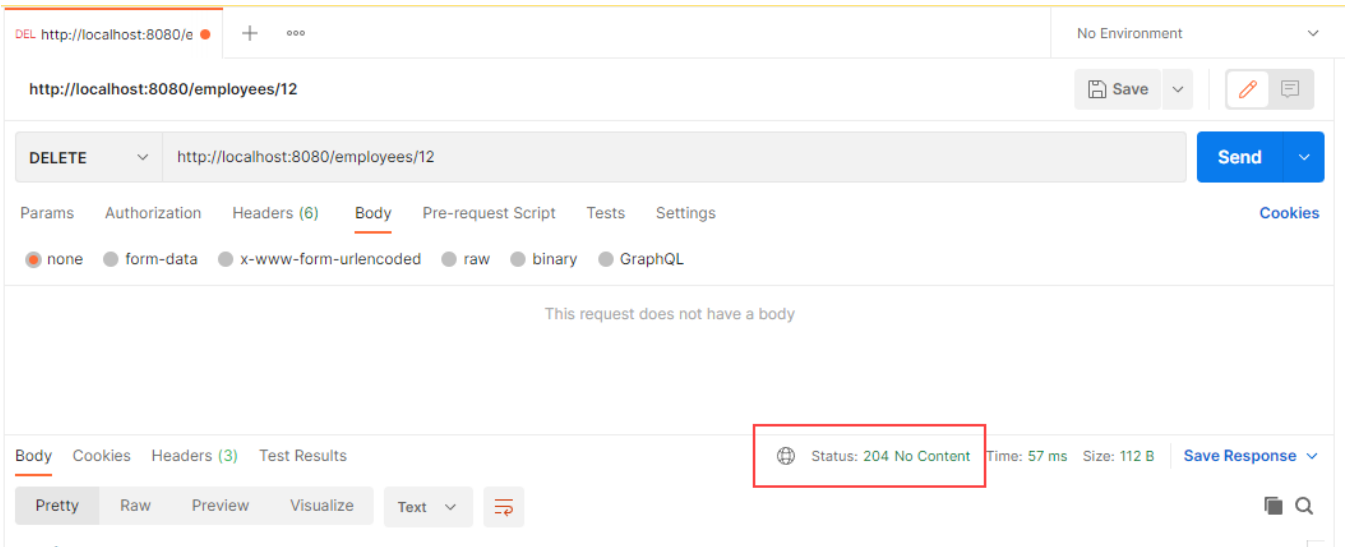
  – Add the `@ResponseStatus` to override the default status of `200`

    * `NO_CONTENT` returns a status of `204`

Example

```
@RequestMapping(path="/employees/{id}",method=RequestMethod.DELETE)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void deleteEmployee (@PathVariable int id)
{
  // the delete method of the DAO
  // should accept the id of the employee
  employeeDao.delete(id);
}
```

- **The client then receives the following response**



Exercises

EXERCISE 3

Continue working on the NorthwindTradersAPI project. You will modify the application to allow users to delete products and categories.

Step 1

Modify the `ProductDao` and `CategoryDao` to add the following methods

```
ProductDao
------------
void delete(int id);

CategoryDao
------------
void delete(int id);
```

Implement the changes that you made to your interfaces in the JdbcDataSources.

```
JdbcProductDao
JdbcCategoryDao
```

Step 2

Modify the `ProductsController` to add a public web function to delete a product.

Add the `@RequestMapping` annotation for `DELETE`

Use the `ProductDao` to delete the record from the database. Test your code using Postman.

Step 3

Modify the `CategoriesController` to add a public web function to delete a category.

Add the `@RequestMapping` annotation for `DELETE`

Use the `CategoryDao` to delete the record from the database. Test your code using Postman.

Commit and push your code!

Section 5–4

CodeWars

CodeWars Kata

- **Survive the Attack**
  - Given 2 arrays (the power levels of your soldiers, and the power levels of the attackers) determine if you win or lose the battle

- **Complete this Kata for additional Java practice**
  - https://www.codewars.com/kata/634d0f7c562caa0016debac5/java

1.

Section 5-5

Authentication

What are JWT tokens?

- **JSON Web Tokens (JWTs) are compact, self-contained tokens used for securely transmitting information between parties. They are commonly used for authentication and authorization in web applications.**

- **JWTs are stateless, scalable, and can be used across different domains.**

- **How JWTs Work?**

  – When a user authenticates, a JWT is generated.

  – This token is then sent to the client.

  – On subsequent requests, the client includes the token in the request header.

  – The server verifies the token's signature and extracts the data contained within.

- **JWT's in SpringBoot**

  – The `jsonwebtoken` library provides classes for creating, parsing, and verifying tokens.