**Intro to Spring and Spring Boot**

**Lesson Duration: 30 - 45 minutes**

**Learning Objectives**

* Understand what Spring and Spring Boot are used for
* Know some of the things you can do with Spring
* Be able to select some dependencies and generate a new Spring Boot project

**Introduction**

Spring is a framework for **back-end** web application development. It can be viewed as an “umbrella” framework, as it allows you to easily make use of many different technologies across a standard of Java technologies.

**Why are we learning this?** Spring is widely used, it is the go to framework in Java for creating web applications.

**What problem are we trying to solve?** When we are building back end applications in Java, we often want to do similar tasks that require a lot of code, such as building web servers, writing controllers, defining routes and more. Because developers often want to do these things in the same way, spring has been written in such a way that helps programmers achieve these tasks while writing as little code as possible.

By the end of this week you will be able to use Java with Spring to:

* Easily create back-end web applications that persist objects to databases
* Use RESTful routes to GET POST PUT and DELETE those objects from the database
* Create a RESTful API with additional custom routes

**What is Spring?**

Spring is a framework for back-end web application development. It can be viewed as an “umbrella” framework, as it allows you to easily make use of many different technologies across a standard of Java technologies.

**Spring is not just one library that we bring into our code to add features** Spring gives us one or more libraries to choose from

A close up of an umbrella

Description automatically generated

Visit [Spring’s project list](https://spring.io/projects) to view a list of some of the things which have been built using Spring. Spring projects can be installed as dependencies and are used to add additional functionality to our own projects.

Objective: Get students excited about the many different things you can do with Spring!

Previously when building application we have been building a lot of this functionality manually:

1. We have been mapping our objects to databases using SQL statements with the objects properties defined as columns.
2. We have been creating controllers that define a set of RESTful routes
3. We have been writing methods that read/write to databases to create CRUD functionality.

These things are all going to be done for us by using these spring projects. We are going to be using the following the Spring projects:

* **Java Persistence API** & **H2 SQL** for reading/writing to a database to persist objects. We will no longer have to manually map our objects properties to database tables, or write the methods that implement the CRUD functionality.
* **Spring Framework** to help us create controllers
* **Spring Data REST** for Building RESTful services, so that we do not need to define the restful routes, or the conventional behaviour for those restful routes

We will get all of the above functionality out of the box.

**Spring Boot**

As Spring is such a large framework it can be difficult navigating its features. It can also be time consuming handling the Spring configuration when starting a new project. This is where [Spring Boot](https://spring.io/projects/spring-boot) comes in. Spring Boot is a project that’s part of Spring that allows us to create Spring Projects really easily.

Spring Boot offers:

* Some API’s and libraries that make building some services easier
* A generator tool to “bootstrap” our project when creating a new Spring Project

**Using Spring Boot “initializr” to Initialise a Project**

Spring Boot Initializr is a starter tool for beginning Spring based projects. As Spring is huge, Spring Boot Initializr allows us to generate the boilerplate code for a project, with any dependencies we choose already installed. In industry, the Spring Boot Initializr (some times referred to as spring generator), is widely used to bootstrap projects.

* Go to <https://start.spring.io/>
* Search and add: **Web**, **Dev Tools**, **JPA**, **H2**
* Provide the group - ‘com.example.techconnect’
* Provide the artefact name - ‘pirateservice’
* Click “Generate Project” and the zip will be downloaded

The dependencies we have just installed for this project are:

* **Web** - allows us to do RESTful routes and create controllers
* **Dev Tools** - makes building spring a little easier with things like automatic reloading
* **H2 Database** - gives us the correct “bindings” to work with H2 Database. H2 is a type of in-memory SQL database.
* JPA - **Java Persistence API** Gives us the power to use Java to “persist” our objects to the database using Hibernate.

The .zip file given to us by Spring Boot Initializr contains a pom.xml file, and some very simple boiler plate Java code.

**pom.xml + Maven**

Each dependency we chose on the “initilizr” is put into the pom.xml for us. This is why Spring Boot “initilizr” is so useful. pom.xml describes the dependencies by listing libraries we want to download and install to build and run that project.

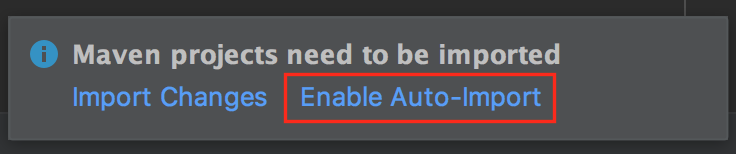
Maven is a build system for Java, just like Gradle. Build systems like Maven and Gradle are responsible for compiling or “building” multiple source files of code into a single application. Keep in mind that Java is a compiled language, meaning that the .java files are compiled into a machine language. Build systems, like Maven make the act of building multiple files and bringing in dependencies significantly easier.

Our pom.xml is describing to Maven what dependencies we want.

**IntelliJ + Maven**

IntelliJ can open up Maven based projects. Luckily this is what was generated and given to us by the Spring Boot Initializr online tool.

Unzip the file to somewhere in coursework and open the folder containing the pom.xml with IntelliJ. IntelliJ will ask if you want to enable the automatic import of Maven projects and you can select ‘Enable Auto-Import’.



*Screenshot of Enable Auto-import Window*

**Point to the Database**

As part of this application, we will want to read and write objects we create to the database. To do this, we need to tell Spring where the Database is.

**application.properties**

When using spring, a lot of the applications “static configuration” - this means config settings we will not want to change while our app’ runs - will need to be set in one common place where Spring can find it. This is what the application.properties file is for.

Find the initially empty application.properties file in the resources directory - src / main / resources - and paste in the following:

# Datasource and H2 stuff

spring.h2.console.enabled=true

spring.h2.console.path=/h2

hibernate.dialect=org.hibernate.dialect.H2Dialect

spring.datasource.url=jdbc:h2:file:./db/pirateservice\_db

spring.datasource.username=sa

spring.datasource.password=

spring.datasource.driver-class-name=org.h2.Driver

spring.jpa.hibernate.ddl-auto=update

spring.jpa.properties.hibernate.enable\_lazy\_load\_no\_trans=true

**DDL-auto setting**

spring.jpa.hibernate.ddl-auto = update is a setting that tells Hibernate + Spring that when we make changes to the database schema (the structure), the changes should be applied to the database.

There are some other value options for various situations. They are:

* **validate**: validate the schema, makes no changes to the database.
* **update**: update the schema.
* **create**: creates the schema, destroying previous data.
* **create-drop**: drop the schema when the application is stopped, then re-create the schema when it’s started again.

**Enable Lazy Load**

spring.jpa.properties.hibernate.enable\_lazy\_load\_no\_trans=true allows the option to do “lazy loading”. This simply means that when querying data on models with relationships, Hibernate will not bring back every single element for both sides of the relationship. This is good for not bringing back unnecessary data.

**Running Spring Applications**

Even though our application does nothing at the moment, we can run this bare-bones application. Find PirateserviceApplication.java in src / main / java / com / example / pirateservice.

The class PirateserviceApplication is special as it contains the main() method. Note that this was already written for us, and that it’s already been populated with some code:

**SpringApplication.**run**(PirateserviceApplication.**class**,** args**);**

This tells Spring to start, but does nothing else. For the most part that’s all we really need in here. This is because of the way Spring works, we launch it, and then ask it to find our code in other ways.

Run this. Everything has worked if the application runs and doesn’t exit then displays the following at the end of a lot of logging output:

Tomcat started on port(s): 8080 (http) with context path ...

Started PirateserviceApplication in 5.663 seconds (JVM running for 6.615)

We are now ready to start building a Spring Application. This will be done over the next few lessons.

**Recap**

* Spring is a framework for building large and small web application allowing you to make use of potentially many different applications
* Spring Boot allows us to start Spring applications easily. We choose what we want to build, and this start generator makes a project with a pom.xml (maven dependencies file) which is pre-filled with the list of dependencies. It is used much like generating a new project in IntelliJ.

**Useful links:**

* <https://spring.io/learn>
* <https://start.spring.io/>
* <https://www.baeldung.com/spring-boot>

**Next Lesson**

[Annotating Classes](https://codeclan.github.io/canvas_notes/course_java/week_3/day_1/2_annotating_class/annotating_class.html)

**Annotating Classes for Persistence**

**Lesson Duration: 60 - 90 minutes**

**Learning Outcomes**

* Know what an ORM (Object-relational mapping) is and how it is useful
* Be able to map classes to a database with Hibernate (via Spring and Spring Boot)
* Be able to use some basic JPA Annotations

**Introduction**

**Why are we learning this?** When building a Java back end web applications, being able to use interact with databases is crucial. Using an ORM (Object-relational mapping), such as Hibernate, helps us by handling the common mapping functionality.

Previously we have had to map our objects to database tables manually, creating the SQL statements that insert the objects, by mapping an object’s properties as column names. This is something developers often want to do in the same way, so an ORM implements this default functionality so we can use it out of the box.

**What are we building?** A web app for managing Pirate objects with Spring and Hibernate, with an API index end point, that gets all the pirates.

**You will be able to:** perform CRUD operations and persist the data into a H2 SQL database, as well as having creating API end points.

* Persist objects to H2 SQL DB via Spring + Hibernate
* Enable RestController on routes to GET JSON data back

Note: We are using the Spring Boot code from the last lesson. There is also a starting point project available.

**Part 1: Mapping our Pirate class**

**Add a new model class Pirate**

We will start by creating a Pirate class which will be our “model” for pirates in the database. In general terms ‘Model’ is a term used for an entity (in this case a Java class) that acts as the template or schema for a table in a database.

Since we want to keep all models together, we start by creating a package for all models:

1. Create a new package in com.example.codeclan.pirateservice by right clicking on com.example.codeclan.pirateservice - new > package. Give it the name ‘models’.
2. In ‘models’, create a new Java class called Pirate.

**Task: (5 minutes)**

1. Create the Pirate class properties:
   * firstName (String)
   * lastName (String)
   * age (int)
2. The Pirate constructor should take the above arguments
3. Add getters and setters for each of the properties

**Solution**

**POJO**

In order to save instances of a class to a database with Java when using an ORM, the class must follow the POJO (Plain Old Java Object) rules:

* All attributes that you want to save to the database should be private and have getter and setter methods.
* All Java classes that will be mapped need a default constructor (an empty constructor that takes no arguments), in additional to any required constructors that take arguments.
* All classes need a id of type Long (a very big Integer) in order to allow Hibernate to map the property to the primary key column of a database table.

So we will first of all change our Pirate class to match these rules.

*// Pirate.java*

**public** **class** **Pirate** **{**

**private** **Long** id**;** *// NEW*

**private** **String** firstName**;**

**private** **String** lastName**;**

**private** **int** age**;**

**public** **Pirate(String** firstName**,** **String** lastName**,** **int** age**)** **{**

**this.**firstName **=** firstName**;**

**this.**lastName **=** lastName**;**

**this.**age **=** age**;**

**}**

**public** **Pirate()** **{** *// NEW*

**}**

**public** **Long** **getId()** **{** *// NEW*

**return** id**;**

**}**

**public** **void** **setId(Long** id**)** **{** *// NEW*

**this.**id **=** id**;**

**}**

*// ...*

**}**

Ok, so now our class is ready to be mapped to a database table using Hibernate.

**Hibernate + JPA Annotations**

JPA stands for Javax Persistance API. Hibernate + JPA annotations are a way to define how classes are mapped to tables in a database.

We have already seen the use of annotations in our test files. @Before, @Test, etc that allows JUnit to identify specific actions we want to perform. Annotations for Hibernate + JPA are added to the Java classes in the same way. This way we can quickly understand the database table structure relating to the class, as it will be visible in the class itself.

Hibernate + JPA annotations come from a set of standards known as EJB 3 (Enterprise Java Beans). Following these standards allows you to transfer your application from one database type to another (for example, PSQL to mySQL).

**Mapping a Class to a Table**

We are going to add come annotations that let Hibernate know how to map our class to a database table. The EJB 3 standard annotations are located in a package called javax.persistence, so we need to import what we need from this package.

* The @Entity annotation lets Hibernate know that we want to map this class to a database table.
* The @Table annotation tells Hibernate the name of the table we want to map to.

*// Pirate.java*

**import** javax.persistence.Entity**;** *// NEW*

**import** javax.persistence.Table**;** *// NEW*

**@Entity** *// NEW*

**@Table(**name**=**"pirates"**)** *// NEW*

**public** **class** **Pirate** **{**

*// ...*

**}**

Note: If we left the name blank Hibernate would create a table with the same name as the class (pirate), but convention says that our table name should be plural.

**Mapping the ID to a Primary Key**

Each database table will have a primary key, and we tell hibernate which property we want it to use for the primary key by using the @Id annotation. We are going to use the idproperty.

We are also going to tell it to generate the value for the id primary key using the IDENTITY strategy. This uses an auto-incremented value generated by the database.

**@Id** *// NEW*

**@GeneratedValue(**strategy **=** **GenerationType.**IDENTITY**)** *// NEW*

**private** **Long** id**;**

So at the point the object is instantiated, its id property is null. When it is saved into the database, the value for the id is generated and used as the primary key.

**Mapping Properties to Columns**

The @Column annotation is used to map the object’s property to a column. There are a set of values we can give @Column to specify the details of the column.

* name specifies the name of the column. (Left blank Hibernate will use the property name)
* length specifies the size of the column.
* nullable when set to true, allows null values in the column. When set to false Hibernate marks the column as NOT NULL and disallows ‘null’ values.
* unique specifies that the column should contain only unique values.

For now, we will set the name of each column, using the property name with snake case syntax.

**@Id**

**@GeneratedValue(**strategy **=** **GenerationType.**IDENTITY**)**

**@Column(**name **=** "id"**)** *// NEW*

**private** **Long** id**;**

**@Column(**name **=** "first\_name"**)** *// NEW*

**private** **String** firstName**;**

**@Column(**name **=** "last\_name"**)** *// NEW*

**private** **String** lastName**;**

**@Column(**name **=** "age"**)** *// NEW*

**private** **int** age**;**

**JPA Repository**

The JPA (Javax Persistence API) Repository sits between the models and the database to allow you do operations (like CRUD) on the database from the program logic. We use Spring’s JPA Repository interface to do this.

We will create our own repository for any model we wish to do database operations with. In this case we have one, our Pirate model, so the repository will be called PirateRepository. It will borrow functionality from the JpaRepository by inheriting from it. As we don’t want to override all the functionality of JpaRepository, our PirateRepository should be created as an interface.

Note: An interface can inherit one or more interfaces.

**Creating a PirateRepository**

* Create a new package inside com.example.codeclan.pirateservice called repositories Note: all repositories we create will live in here
* Inside the ‘repositories’ directory create an interface called PirateRepository

We now need to extend the JpaRepository interface to borrow its behaviours. We also need to tell the JpaRepository the type of object we are creating the repository for (Pirate). Lastly we need to specify the indexing system (Long). The indexing system limits the maximum number of entries we can do queries on.

**package** com**.**example**.**pirateservice**.**repositories**;**

**import** com.example.pirateservice.models.Pirate**;** *// NEW*

**import** org.springframework.data.jpa.repository.JpaRepository**;** *// NEW*

**public** **interface** **PirateRepository** **extends** **JpaRepository<Pirate,** **Long>** **{** *// MODIFIED*

**}**

This is a perfect example of Interfaces and Inheritance in real applications. This pattern allows us to use some of the parent interface’s functionality, whilst still giving the opportunity to add our own. For now, we only want what’s provided by JpaRepository.

**Annotating the Repository**

In Spring, a Repository is a general way of accessing some external resource (often a database) and performing actions on it. We need to tell Spring that this interface is a Repository. We will use the @Repository annotation and place it above the interface declaration to do this.

**@Repository** *//NEW*

**public** **interface** **PirateRepository** **extends** **JpaRepository<Pirate,** **Long>** **{**

**}**

Note: Make sure Repository is brought in via the org.springframework.stereotype.Repository dependency.

Nothing will be put into this interface yet. For now, we will use the methods provided to it by JpaRepository.

**Saving a Pirate into the Database**

We are going to use a test to instantiate a Pirate and save it into the database using the repository we have just created.

**Task: (5 minutes)**

Inside test / Java / com.codeclan.pirateservice / PirateserviceApplicationTests create a test called createPirate that instantiates a Pirate object with the following properties:

* first name: “Jack”
* last name: “Sparrow”
* age: 32

Note: The text doesn’t need to assert anything at the moment.

**Example solution**

We know that we want to use our PirateRepository to save our Pirate object into the database, so we need it available to use in the test. To do this we are going to use Dependency Injection.

**Dependency Injection**

Dependency Injection is where an object is instantiated somewhere else and then given to you; the instance is ‘injected’ as a dependency. In Spring we can use the annotation @Autowired on a class property to ask the framework for an instance of a class to be provided as the dependency.

Looking inside PirateserviceApplicationTests.java, we wish to **Dependency Inject** the PirateRepository. We can give the test class and instance of PirateRepository using @Autowired annotation.

**@RunWith(SpringRunner.**class**)**

**@SpringBootTest**

**public** **class** **PirateserviceApplicationTests** **{**

**@Autowired** *// NEW*

**PirateRepository** pirateRepository**;** *// NEW*

*// ...*

**}**

**Save the Object to the Database**

Now we’ve dependency injected PirateRepository, it can be used to save the newly created pirate to the database.

**Pirate** jack **=** **new** **Employee(**"Jack"**,** "Sparrow"**,** 30**);**

pirateRepository**.**save**(**jack**);**

Let’s run the test to save the object to the database.

**Then**, lets find the main for the application and run that. Once it’s running we should be able to go to

<http://localhost:8080/h2>. You can only go here if the PirateserviceApplication main is running correctly.

This should allow us to login and play with the H2 SQL database.

1. Run the test
2. Check the table by running this command in the h2 console:

SELECT **\*** FROM pirates;

It should show something like:

id | age | first\_name | last\_name

----+-----+------------+-----------

1 | 32 | Jack | Sparrow

**Conclusion**

We’ve seen how to map classes to a database using the ORM, Hibernate, (via Spring and Spring Boot) with JPA Annotations.

We also learned how to use dependency injection to inject the PirateRepository interface we created into where we need instances of it.

**Next Lesson**

[Controller and JPA Repository Queries](https://codeclan.github.io/canvas_notes/course_java/week_3/day_1/3_controller_and_jpa_repository_queries/controller_and_jpa_repository_queries.html)

 RestController and JpaRepository Queries

**Lesson Duration: 30 - 40 minutes**

## Learning Outcomes

* Know how to create controllers with Spring and dependency injection
* Be able to create a RESTful endpoint using Spring

## Introduction

**Why are we learning this?** We are creating a RESTFUL api, so we want to be be able to make requests (GET, POST, PUT, DELETE) to a set of API endpoints, that allows us to perform CRUD operations on our database.

**What are we building?** A web app for tracking Pirates with Spring and Hibernate that will allow us to call API endpoints to perform CRUD operations and persist the pirate data into a H2 SQL database.

**You will be able to:** enable RestController on routes to GET JSON data back for all Pirates we create.

**RESTful** typically has the front-end being hosted from a different location on the web from the back-end. Data is passed between back-end and front-end by HTTP request/responses. The front-end application makes these HTTP requests and renders the data when it’s received.

We will be implementing a set of RESTful routes, using the RESTFUL Controller, exposing an index routes that returns all the pirate data.

## Creating the PirateController

1. Create a new package called ‘controllers’ inside com.example.codeclan.pirateservice
2. Inside ‘controllers’ create a new Java class called PirateController

To tell Spring this is a RestController, we use the annotation @RestController.

We use the annotation @RequestMapping to specify the base URL for the controllers endpoints.

**@RestController** *// NEW*

**public** **class** **PirateController** **{**

**}**

This means that all routes in this class will be accessible within /pirates.

### Inject PirateRepository

Like before, we ask Spring to simply inject an instance of our PirateRepository into our controller so we can use it:

**public** **class** **PirateController** **{**

**@Autowired** *// NEW*

**PirateRepository** pirateRepository**;** *// NEW*

**}**

### Find all Pirates

We will define the method that will return the data we want to serve up on the route. As we are creating the index route, we will return a list of all the Pirates from the database.

**public** **class** **PirateController** **{**

*// ...*

**public** **List<Pirate>** **getAllPirates()** **{** *// NEW*

**return** pirateRepository**.**findAll**();**

**}**

**}**

We use the @GetMapping annotation to map a route to a method. We are going map getAllPirates and because we are not going to pass the annotation a path, it will use the base route path defined in @RequestMapping.

**public** **class** **PirateController** **{**

*// ...*

**@GetMapping(**value **=** "/pirates"**)** *// NEW*

**public** **List<Pirate>** **getAllPirates()** **{** *// NEW*

**return** pirateRepository**.**findAll**();**

**}**

**}**

Note: @GetMapping is simply a version of the commonly used @RequestMapping with the method already set to GET

We can also give @GetMapping arguments for sub routes as is common practice.

### Find one Pirate

We can get a single pirate by id on the /pirates/id route by grabbing the id from the route using the @PathVariable annotation.

Since we are unsure if findById on the repository will return a Pirate or null we should return an Optional. Optional is a container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value.

*// PirateController.java*

**@GetMapping(**value **=** "/pirates/{id}"**)**

**public** **Optional<Pirate>** **getPirate(@PathVariable** **Long** id**){**

**return** pirateRepository**.**findById**(**id**);**

**}**

To make better use of Optional the checks should be done to check the contained value and handle appropriately. Fortunately Optional can be serialised to the value it contains.

## Run it

Run the application runner (PirateserviceApplication) and go to http://localhost:8080/pirates where some JSON output of the pirates should be shown.

# Summary

We’ve seen how to build a RESTful controller easily with spring and have it return a JSON version of our object

# Next Lessons

**Now: Lab** - [Lab - Bootstrap a new Spring Project](https://codeclan.github.io/canvas_notes/course_java/week_3/day_1/lab_bootstrap_new_spring_project/lab_bootstrap_new_spring_project.html)

**Later: Homework** - Learn about Many to One and Many to Many database relationships with Spring with these flipped lessons:

* [Learn Many to One](https://codeclan.github.io/canvas_notes/course_java/week_3/day_1/homework_relationships/hw_one_to_many.md)
* [Learn Many to Many](https://codeclan.github.io/canvas_notes/course_java/week_3/day_1/homework_relationships/hw_many_to_many.md)

### Thinking in Spring

With Spring, it’s a common pattern to create classes, annotate them (which means to add little code snippets to give them some functionality) and then never actually directly instantiate them ourselves. This is something that will seem strange at first, but after time will make sense and seem natural.

**Lab: Bootstrap an Employee tracking application**

**Lab Duration: 45-60 minutes**

**Learning Objectives**

* Consolidate how to bootstrap a new Spring Application
* Be able to annotate a class for persistence to the database
* Be able to create a repository for doing database operations
* Be able to inject the repository into the controller

**Brief**

In this lab please use what you’ve already learned about Spring to create an entirely new Spring application that allows you to get the details of an employee.

Start from scratch using the [Spring Boot Initializr](https://start.spring.io/).

**MVP**

Create a Spring Boot application for recording employee data that has:

* An Employee class that conforms to POJO and is annotated with fields name, age, employeeNumber and email.
* A repostory for doing database operations
* A RestController with one route for getting a JSON list of all Employees.

**Planning**

Make a list/diagram of the files that will make up your program and note down their responsibilities.

**Flipped Lessons - Learn about Relationships**

For this homework you will follow lessons on One To Many and Many to Many relationships with Spring and Hibernate.

We’ve provided the end code for you to follow along.

* Follow along with the [Many to One Lesson](https://codeclan.github.io/canvas_notes/course_java/week_3/day_1/flipped_lessons_relationships/flipped_lesson_one_to_many.html) and read the code along with it to help your understanding
* Follow along with the [Many to Many Lesson](https://codeclan.github.io/canvas_notes/course_java/week_3/day_1/flipped_lessons_relationships/flipped_lesson_many_to_many.html) and read the code provided to understand how it’s implemented.

**One-To-Many Relationships using annotations**

**Learning Objectives**

* Understand the one-to-many relationship.
* Know how to map one-to-many relationships using hibernate annotations.

**Introduction**

**Why are we learning this?** In Object Orientated programming it’s extremely common to have objects that contain lists of other objects, or are related in some other way.

When saving these objects to the database, we want to save them in such a way that when the data is queried, the relationships are preserved, or that the relationships between objects actually help us query the data. For example:

* Give me all of the students for instructor Barbara
* Give me the students Barbara has who’ve done the weekend homework

Forgetting about databases for a moment, the above examples couldn’t work if there was no relationship between the classes that represent the student and instructor objects.

We need to allow relationships to be saved to the database. There are three types of relationship:

1. One-to-Many
2. Many-to-Many
3. One-to-One

This lesson will only focus on One-to-Many.

**Understanding the relationships**

Extending on our pirate system, we now want to create a Ship class that will be persisted with a ships table that has the following properties:

1. name - the name of the ship
2. pirates - a List of Pirate

A ship will have a name and List of Pirate. To do this without databases entirely, that’s all we would need. However to save that relationship in the database we need to do a little more. We need to know:

* the relationship between Ship and Pirate is **one-to-many**, that one Ship has many Pirate’s.
* the backwards relationship is **many-to-one**, that many Pirates has one Ship.

**One ship has many pirates**

**Why map the opposite relationship?**

With JPA + Hibernate, which we are going to use, this opposite relationship must also be realised and then mapped because queries from both sides need to work. For example, if we have a Pirate object and want to know its associated Ship, that ship (or at least the ships ID) needs to be stored in Pirate.

In simple terms, a ship will have a list of Pirates and a Pirate will have a single Ship. This single Ship in Pirate simply “points back” to the Ship.

The following diagram shows very simple UML with the Ship containing a List of Pirate and the Pirate containing one Ship with the relationships annotated in orange.

**Database Tables**

Let’s talk “database”, putting the Java to one side for a moment. We know two tables are required: One called ships and one pirates. Remember it’s a convention that table names use the plural.

**Foreign Keys**

Relationships between tables in databases are made with **foreign keys**. A foreign key is a column in a table that contains the primary key of another table. Thus, the foreign key points to the primary key in another table, pointing to that table.

*Database Tables*

**Setting up the Relationships for JPA + Hibernate**

We now completely understand the relationship we want to create in Java code, and have some understanding of what the resulting table structure needs to be. A quick summary so far will help with the next steps.

* One ship should have many pirates
  1. Ship should contain an ArrayList of Pirates
  2. This is the **one-to-many** relationship
* Many pirates should have one ship
  1. Pirate should have a single associated Ship to “point back”
  2. This is the **many-to-one** relationship
  3. The pirates table has a foreign key ship\_id pointing to the associated ship.

**Annotations**

By using the above list as a guide, we now have enough information to correctly annotate the Pirate and Ship classes.

**One Ship should have many Pirates**

An annotation for this is provided by JPA @OneToMany. It should be used to annotate a Collection, List or ArrayList property. In our case, we use it to annotate the List of Pirate in the Ship class. Like all Hibernate + JPA annotations, it can be either put directly on that property or the getter.

@OneToMany takes an argument mappedBy which needs to have the name of the property in Pirate class that we wish to use as the foreign key.

We will also use the fetch type LAZY. This means that when a ship is loaded from the database then the associated pirates should only be loaded on demand, or when it’s asked for. The alternative here is that when a ship is loaded, it loads all the pirates too, this is called EAGER FetchType. We may not want **all** the pirates when loading a ship, so for now we use FetchType.LAZY. Looking into Ship.java we can see the list of Pirate annotated with @OneToMany.

**@OneToMany(**mappedBy**=**"ship"**)**

**private** **List<Pirate>** pirates**;**

#### ’new’ the List in constructor

When an instance of Ship is being created, the List of Pirate must also be instansiated, otherwise we will get a null pointer exception. This is when an object is being used that hasn’t been instansiated or *“new’d”*.

Remember to new the list in the constructor of Ship

**this.**pirates **=** **new** **ArrayList<Pirate>();**

**Many Pirates should have one Ship**

An annotation for this is provided by JPA called @ManyToOne. It should be used to annotate the property in our class that **points back** to the class that stores many of it.

In our case we use it to annotate the ship property in Pirate. Since a Pirate needs to point back to its Ship. Also, because we need to tell JPA that we know pirates table needs to have a foreign key for ships.

*// ...*

**@ManyToOne** *// NEW*

**@JoinColumn(**name**=**"ship\_id"**,** nullable**=false)** *// NEW*

**private** **Ship** ship**;** *// NEW*

The @ManyToOne annotation is used to set the relationship between Pirate and Ship classes.

The @JoinColumn annotation will generate a column ship\_id (a foreign key) in the pirates table which will point to the id of the ships table (primary key). JPA + Hibernate will take care of this for us. We only need to worry about the Java side which is why we don’t give the Pirate class a ship\_id property.

To ensure that each pirate needs to have a ship we will set the foreign key as nullable = false. So there must be an entry in that column. If we tried to save a pirate to the database without assigning them a ship then we would get a null pointer exception, for exmaple, not-null property references a null or transient value : models.Pirate.ship. We would also get an error if we save a pirate with a ship before we save the ship to the database as there would be no matching ship id.

So to ensure Pirate must have a ship we will also change the constructor of Pirate to take in the ship when we create a new Pirate object.

*// ...*

**@ManyToOne**

**@JoinColumn(**name**=**"ship\_id"**,** nullable**=false)**

**private** **Ship** ship**;**

**public** **Pirate(String** firstName**,** **String** lastName**,** **int** age**,** **Ship** ship**)** **{** *// MODIFIED*

**this.**firstName **=** firstName**;**

**this.**lastName **=** lastName**;**

**this.**age **=** age**;**

**this.**ship **=** ship**;** *// NEW*

**}**

**Set ship getter + setter**

We’ve just added a new property ship to the Pirate class so it’s important to not forget to add a getter and setter for that property.

**public** **Ship** **getShip()** **{**

**return** ship**;**

**}**

**public** **void** **setShip(Ship** ship**)** **{**

**this.**ship **=** ship**;**

**}**

**Ship JpaRepository**

In exactly the same way we done with PirateRepository we should create an interface ShipRepository that implements JpaRepository

**public** **interface** **ShipRepository** **extends** **JpaRepository<Ship,** **Long>** **{**

**}**

**Creating Pirates and Ships**

One we’ve updated Pirate and Ship it’s now possible to setup both to save some data to the database.

We will need to inject the ship repository as well.

**public** **class** **PirateserviceD1StarterApplicationTests** **{**

**@Autowired**

**PirateRepository** pirateRepository**;**

**@Autowired**

**ShipRepository** shipRepository**;**

*// AS BEFORE*

**@Test**

**public** **void** **createPirateAndShip(){**

**Ship** ship **=** **new** **Ship(**"The Flying Dutchman"**);**

shipRepository**.**save**(**ship**);**

**Pirate** pirate1 **=** **new** **Pirate(**"Jack"**,** "Sparrow"**,** 32**,** ship**);**

pirateRepository**.**save**(**pirate1**);**

**}**

**}**

**Output from psql**

pirateservice=> select \* from pirates;

id | age | first\_name | last\_name | ship\_id

----+-----+------------+-----------+---------

1 | 32 | jack | sparrow | 1

**Checking the controller**

Looking back at the PirateController we still have endpoint setup to get all pirates on /pirates.

Trying this endpoint now will result in some odd and likely non-deterministic behavior. Non-deterministic means it won’t give a consistent results each time. In computing, non-determinisim is often bad, as inconsistency proves shows things are not working correctly.

To investigate this, look at the log output from Spring. We might see some errors like:

Cannot render error page for request [/pirates] and exception [Could not write JSON: Infinite recursion (StackOverflowError);

This means that for some reason, when the JSON is being created from our object, it’s infinitely recursing becuase of some property.

Thinking about the relationship between Pirate and Ship. A Ship has a list of Pirate, and Pirate has a single Ship it’s “in”.

The serializer, which converts our objects to JSON is attempting the following:

* Look at a ship
* Go through all its properties and try to serialize them
* If a relationship property is found, go to that relationship. So it finds pirates and goes into that object.
* Look at that pirate
* Go through all its properties and try to serialize them
* If a relationship property is found, to to that relationship. So it finds ship and goes to that object.
* Look at that ship
  1. ..
  2. .
  3. And so on repeating this pattern

This will happen without any issue for a long time in computing terms, until Java runs out of memory. This explains the “StackOverflowError”.

**Fixing the recursive JSON StackOverflow**

We now know that the problem is caused by the serializer which converts our object to JSON trying to serialize the properties of our model which have relationships.

This can be stopped simply by adding an annotation to these properties @JsonIgnoreProperties("ship"). This tells the serializer to not try and convert these properties to JSON.

In Ship we now have:

**@JsonIgnoreProperties(**"ship"**)** *// NEW*

**@OneToMany(**mappedBy **=** "ship"**)**

**private** **List<Pirate>** pirates**;**

The /api/pirates custom controller should now return correct data without trying to serialize the model relationships.

**Summary**

* Should have some understanding the one-to-many relationship.
* Should have some idea how to map one-to-many relationships using hibernate annotations.

**Many-To-Many Relationships using Annotations**

**Learning Objectives**

* Understand the many-to-many relationship.
* Know how to map many-to-many relationships using hibernate annotations.

**Introduction**

**Why are we learning this?** When saving objects to the database, we want to save them in such a way that when the data is queried, the relationships are preserved. Or that the relationships between objects actually help us query the data.

For example:

* Given a student “Jim”, give me the classroom he’s part of that has white walls
* Given classroom, give me all the students which use it that have brown hair

Without the many to many relationships, these queries of structured data wouldn’t be possible. We need to allow relationships to be saved to the database. As mentioned before, there are three types.

This lesson will focus on Many-to-Many.

**Understanding the relationships**

Extending our pirate system even further, we now want to add a Raid class that will be persisted with a raids table. This model should have properties:

1. location - the name of a location where the raid takes place. Eg: Glasgow, Edinburgh, Tortuga, Tresure Island
2. loot - the amount of plunder that was looted in that raid

A pirate can go on many different raids, and raids can have many different pirates go on them. A raid is simply a location that a group of pirates will visit together to “collect plunder” by whichever means they please. Without the database mappings, this is just a case of having a List of Pirate in Raid and a List of Raid in Pirate. To save that relationship in the database we need to do a little more.

This requires the **many-to-many** relationship. A pirate can go on many raids, and a raid can have many pirates:

*Many to Many Relationship*

We need to understand that with many-to-many, the relationship is relatively symmetrical. It’s also important to know that each side still has to be mapped for JPA + Hibernate to setup the relationship correctly.

**Why do both sides need to be mapped?**

With JPA + Hibernate which we are going to use, both sides of the relationship need to be mapped because we want to be able to make queries from both sides. For example if we have a Raid object and want to know which Pirates went on that raid, then the mapping must exist from the Raid side. The same applies for the opposite. For example if we have a Pirate and want to know which Raid’s that pirate went on, we need to have the mapping on that side.

**Database Tables**

Taking about the database side and putting the Java to one side for a moment. We know two tables are required. One called pirates and one raids.

### Foreign Keys in a Join Table

Many-to-many relationships between two tables in databases are made by creating a join table that sits between the two tables. The join table has two columns for foreign keys. One with the foreign key of one side (our pirates foreign key) and the other with foreign key of the other side of our relationship (the raids foreign key). The columns are simply named pirate\_id and raid\_id, respectively.

Each row of the join table maps one side of the relationship to the other and each mapping can exist many times for either side. An example of populated tables including join table for our pirates + raids might look like this:

**raids**

|  |  |  |
| --- | --- | --- |
| id | location | loot |
| **1** | **Tortuga** | **100** |
| **2** | **Tortuga** | **100** |
| **3** | **Tortuga** | **100** |
| **4** | **Tortuga** | **100** |

**pirates\_raids** (join table)

|  |  |
| --- | --- |
| Pirate ID | Raid ID |
| **1** | **3** |
| **1** | **4** |
| **1** | **2** |
| **2** | **1** |

**pirates**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id | age | first\_name | last\_name | ship\_id |
| **1** | **32** | **jack** | **sparrow** | **1** |
| **2** | **40** | **jimmy** | **nail** | **2** |

1. Pirate with id 1 has raids 3, 4 and 2
2. Pirate with id 2 has raid 1
3. Raid with id 3 has pirate 1

Our table diagram now looks like this.

*Database table diagram with relationships*

**Setting up the Relationships for JPA + Hibernate**

We now understand the relationship we want to create in Java code and have some understand of what the resulting table structure needs to be. A quick summary so far will help with the next steps:

* Pirate should have many raids
  1. Pirate should contain a list of Raid
  2. Pirate should have something that sets up one side of the join table pirates\_raids
* Raid should have many pirates
  1. Raid should contain a list of Pirate
  2. Raid should have something that sets up the other side of the join table pirates\_raids

**Remember** when adding lists they need to be instansiated in the constructor.

**Annotations**

By using the above steps as a guide, we now have enough information to correctly annotate Raid and Pirate classes.

**Many Pirates should have many Raids**

An annotation for this is provided by JPA @ManyToMany. It should be used to annotate properties of a class or the properties getters. Typically these properties are of type Collection, List or ArrayList.

**JoinTable and JoinColumn**

@JoinTable is required to setup the join table. We must instruct JPA about the join table we wish to have created for us, as well as which columns to setup within that join table. To indicate which columns we want in the join table, a @JoinColumn must be passed into the @JoinTable via the joinColumn argument. We must give that the name of the columns we wish to setup in the join table **from the point of view of the class or table we are “in”**.

So in Pirate we will give the JoinTable a JoinColumn with name pirate\_id.

#**Pirate** **Class**

**@ManyToMany**

**@JoinTable(**

name **=** "pirates\_raids"**,**

joinColumns **=** **{** **@JoinColumn(**

name **=** "pirate\_id"**,**

nullable **=** **false,**

updatable **=** **false)**

**}**

**)**

**private** **List<Raid>** raids**;**

We must also map the reverse with the inverseJoinColumns argument to @JoinTable. Remember this is from the point of view of the class we are “in”, so for Pirate this would be raid\_id.

#**PirateClass**

**@ManyToMany**

**@JoinTable(**

name **=** "pirates\_raids"**,**

joinColumns **=** **{** **@JoinColumn(**

name **=** "pirate\_id"**,**

nullable **=** **false,**

updatable **=** **false)**

**},**

inverseJoinColumns **=** **{** **@JoinColumn(**

name **=** "raid\_id"**,**

nullable **=** **false,**

updatable **=** **false)**

**}**

**)**

**private** **List<Raid>** raids**;**

We don’t want these columns in the join table to be allowed to be set to null. Also, it’s not desired that they should be able to be individually updated. This is because allowing the join tables rows to be set individually will break relationships that exist. This is not to say the relationships should never be removed, it just means they cannot be individually changed once created.

**Many Raids should have many Pirates**

For the other side of this relationship, we use exactly the same annotations with the same rules:

* @ManyToMany
* @JoinTable with
  1. name of join table
  2. joinColumns containing a @JoinColumn with the name of the column we wish to map the foreign key for from that class.
  3. inverseJoinColumns containing a @JoinColumn with the name of the column we wish to map to the foreign key for the **other** class.

*//Raid.java*

**@ManyToMany**

**@JoinTable(**

name **=** "pirates\_raids"**,**

joinColumns **=** **{** **@JoinColumn(**

name **=** "raid\_id"**,**

nullable **=** **false,**

updatable **=** **false)**

**},**

inverseJoinColumns **=** **{** **@JoinColumn(**

name **=** "pirate\_id"**,**

nullable **=** **false,**

updatable **=** **false)**

**})**

**private** **List<Pirate>** pirates**;**

**Cascade Types and Cascading**

When we make a change to one side of the many to many, we can choose what happens to the other side. Thus, if we delete one side, we can use the @Cascade setting to allow or disallow the delete action. These settings are filters for the cascade actions we wish to allow. If for example we set it to SAVE\_UPDATE then only save and update actions are cascaded to the other side of the relationship.

**Classes Methods for Adding to the Lists**

Before trying it, make sure there’s class methods for adding Pirate objects to the list in Raid and adding Raid objects to the list in Pirate, otherwise there’s no way to setup the relationships.

**JpaRepository**

A JpaRepository will need to be used for each model to persist that model to the database.

It should now be possible to add pirates to raids and raids to pirates, save these with the repository and check the H2 console. http://localhost:8080/h2for the relationship.

**Summary**

* Should understand the many-to-many relationship.
* Should know how to map many-to-many relationships using JPA + Hibernate annotations.

**Lab: One To Many and Many to Many**

**Lab Duration: 90 minutes**

**Learning Objectives**

* Understand the relationships, one to many and many to many.
* Be able to implement one-to-many and many-to-many relationships between models with Spring JPA + Hibernate

**Introduction**

In this lab, use what you’ve learned about **one-to-many** and **many-to-many** to create an application for a company to track their employees, departments and projects.

**Task**

**MVP**

A Department must have one-to-many with Employee: A Department has many Employees. An Employee must have a many-to-many with a Project: Many Employees can be part of a Project and a Project can have many different Employees in it.

Implement the following models to achieve the above relationships:

An Employee must:

* have a first name
* have last name
* have an employee number
* be able to be assigned to multiple projects

A Department must:

1. have a name
2. be able to add multiple employees

A Project must

1. have a name
2. have a duration in number of days
3. be able to add multiple employees

**Planning**

Draw a diagram modelling the relationships between the models. Study the previous two lessons on **one-to-many** and **many-to-many** to help you.

This lab requires two relationships to be setup, it’s a good idea to complete one relationship at a time.

Remember to create a JpaRepository for each model

# Connecting REST Controllers to Data

**Lesson Duration: 60 minutes**

### Learning Objectives

* Understand how to create controllers that conform to REST principles

## Introduction

**Why are we learning this?** When there’s a relationship between two or more models, it is valuable to be able to access that data via RESTful routes

**By the end of this lesson you will be able to** Setup routing to be able to access the collections of raids in pirates and pirates in ships

### Seeding the Database

When doing both integration and unit testing, we will need a way to seed the database. Doing this will allow us to check relationships are setup correctly and that the data we expect is coming back. This will become especially valueable when testing custom queries.

With Spring, we can tap into Spring Boot’s ApplicationRunner. By creating a class that inherits from ApplicationRunner we can override the run method which is called implicitly on Spring startup. It will be picked up by Spring and run if we use the @Component annotation.

Making use of the fact that this is run each time our Spring application starts. Create a package called components a new class DataLoader in it. Just slack the main for this out.

**@Component**

**public** **class** **DataLoader** **implements** **ApplicationRunner** **{**

**@Autowired**

**PirateRepository** pirateRepository**;**

**@Autowired**

**ShipRepository** shipRepository**;**

**@Autowired**

**RaidRepository** raidRepository**;**

**public** **DataLoader()** **{**

**}**

**public** **void** **run(ApplicationArguments** args**)** **{**

**Ship** dutchman **=** **new** **Ship(**"The Flying Dutchman"**);**

shipRepository**.**save**(**dutchman**);**

**Ship** pearl **=** **new** **Ship(**"The Black Pearl"**);**

shipRepository**.**save**(**pearl**);**

**Ship** blackPig **=** **new** **Ship(**"The Black Pig"**);**

shipRepository**.**save**(**blackPig**);**

**Ship** dustman **=** **new** **Ship(**"The Flying Dustman"**);**

shipRepository**.**save**(**dustman**);**

**Ship** galley **=** **new** **Ship(**"Adventure Galley"**);**

shipRepository**.**save**(**galley**);**

**Ship** revenge **=** **new** **Ship(**"Queen Anne's Revenge"**);**

shipRepository**.**save**(**revenge**);**

**Ship** fancy **=** **new** **Ship(**"Fancy"**);**

shipRepository**.**save**(**fancy**);**

**Ship** fortune **=** **new** **Ship(**"Royal Fortune"**);**

shipRepository**.**save**(**fortune**);**

**Pirate** jack **=** **new** **Pirate(**"Jack"**,** "Sparrow"**,** 32**,** pearl**);**

pirateRepository**.**save**(**jack**);**

**Pirate** john **=** **new** **Pirate(**"John"**,** "Silver"**,** 55**,** dutchman**);**

pirateRepository**.**save**(**john**);**

**Pirate** pugwash **=** **new** **Pirate(**"Horatio"**,** "Pugwash"**,** 55**,** blackPig**);**

pirateRepository**.**save**(**pugwash**);**

**Pirate** maggie **=** **new** **Pirate(**"Maggie"**,** "Lafayette"**,** 35**,** dustman**);**

pirateRepository**.**save**(**maggie**);**

**Pirate** william **=** **new** **Pirate(**"William"**,** "Kidd"**,** 40**,** galley**);**

pirateRepository**.**save**(**william**);**

**Pirate** blackbeard **=** **new** **Pirate(**"Edward"**,** "Teach"**,** 45**,** revenge**);**

pirateRepository**.**save**(**blackbeard**);**

**Pirate** henry **=** **new** **Pirate(**"Henry"**,** "Avery"**,** 25**,** fancy**);**

pirateRepository**.**save**(**henry**);**

**Pirate** bart **=** **new** **Pirate(**"Bartholomew"**,** "Roberts"**,** 47**,** fortune**);**

pirateRepository**.**save**(**bart**);**

**Raid** raid1 **=** **new** **Raid(**"Tortuga"**,** 100**);**

raidRepository**.**save**(**raid1**);**

**Raid** raid2 **=** **new** **Raid(**"Treasure Island"**,** 690**);**

raidRepository**.**save**(**raid2**);**

**Raid** raid3 **=** **new** **Raid(**"Barbados"**,** 500**);**

raidRepository**.**save**(**raid3**);**

**Raid** raid4 **=** **new** **Raid(**"St. Kitts"**,** 500**);**

raidRepository**.**save**(**raid4**);**

**Raid** raid5 **=** **new** **Raid(**"Havana"**,** 200**);**

raidRepository**.**save**(**raid5**);**

**Raid** raid6 **=** **new** **Raid(**"Port Royal"**,** 1000**);**

raidRepository**.**save**(**raid6**);**

jack**.**addRaid**(**raid1**);**

jack**.**addRaid**(**raid2**);**

pirateRepository**.**save**(**jack**);**

raid2**.**addPirate**(**john**);**

raidRepository**.**save**(**raid2**);**

raid3**.**addPirate**(**pugwash**);**

raid3**.**addPirate**(**maggie**);**

raidRepository**.**save**(**raid3**);**

raid4**.**addPirate**(**pugwash**);**

raid3**.**addPirate**(**jack**);**

raidRepository**.**save**(**raid4**);**

blackbeard**.**addRaid**(**raid5**);**

blackbeard**.**addRaid**(**raid6**);**

pirateRepository**.**save**(**blackbeard**);**

raid5**.**addPirate**(**william**);**

raidRepository**.**save**(**raid5**);**

raid6**.**addPirate**(**henry**);**

raidRepository**.**save**(**raid6**);**

**}**

**}**

### Now use create-drop setting

Since, we’re now seeding the database, the spring.jpa.hibernate.ddl-auto = update setting should be set to create-drop.

spring.jpa.hibernate.ddl-auto = create-drop

This drops the db including it’s schema every time the application is terminated. It’s then created every time the application starts.

### REST Routes we need

Get the students to help put together this list of routes we need to create on the board. Talk them through this and re-explain REST if you need to.

* Get all pirates/raids/ships
  + GET /pirates - returns JSON array of all pirates
  + GET /raids - returns JSON array of all raids
  + GET /ships - returns JSON array of all ships
* Get one pirate / raid /ship
  + GET /pirates/{id} - returns JSON for a single pirate
  + GET /raids/{id} - returns JSON for single raid
  + GET /ships/{id} - returns JSON for single ship
* Get a pirate’s raids
  + GET /pirates/{id}/raids - returns a list of raids for a given pirate id
* Get a specific raid from a pirate
  + GET /pirates/{id}/raids/{id} - returns a specific single raid for a given pirate.

## Pirates Controller

First let’s create all the routes that will be in the PirateController controller.

If we don’t have it already, we can add the route that gets all the pirates.

**@Autowired**

**PirateRepository** pirateRepository**;**

**@GetMapping(**value **=** "/pirates"**)**

**public** **List<Pirate>** **getAllPirates(){**

**return** pirateRepository**.**findAll**();**

**}**

The GET mapping getAllPirates will take the **ArrayList** of Pirate objects using the PirateRepository and **serailize** it into JSON.

**Reminder:** A *PirateRepository* instance need to be Autowired or injected into the controller. We ***don’t*** do *PirateRepository pirateRepository = new PirateRepository()* - instead we ask Spring for an instance of *PirateRepository* that we know it has.

It will then send that as an HTTP response back to the client that requested it.

By default, the HTTP status code on this response will be 200 OK.

### HTTP Status Codes

HTTP Status codes represent various statuses of the back-end of the application. An HTTP response has an optional payload (response data) and a required code.

An incomplete list of common HTTP status codes.

| Code | Type | Used for |
| --- | --- | --- |
| **200** | **2xx** | **Success OK** |
| **404** | **4xx** | **Client Error - data not found** |
| **500** | **5xx** | **Server Error** |

There are dozens of errors through various categories. For a full list look here: <https://www.restapitutorial.com/httpstatuscodes.html>

### Specifying a Status Code

We need to use a spring ResponseEntity. That gives us an object that allows us to specify the status code, allowing us to customise it to our needs.

Let’s convert that getAllPirates method to return a ResponseEntity that takes a payload of type List<Pirate>

*//PirateController.java*

**@GetMapping(**value **=** "/pirates"**)**

**public** **ResponseEntity<List<Pirate>>** **getAllPirates(){**

**return** **new** **ResponseEntity<>(**pirateRepository**.**findAll**(),** **HttpStatus.**OK**);**

**}**

When creating a ResponseEntity object we need to give it a payload (our List of Pirate objects) and a status Code. Status code for success is always 200 - OK, but we can get to that by using HttpStatus.OK

Now run the main and check the response in Insomnia or Chrome. http://localhost:8080/pirates - it should be a list of Pirates in JSON form, the same as before, but now we have control over the response status code.

## The infinite recursion problem

When hitting http://localhost:8080/ships/pirates/1 (or of course localhost:8080/ships/pirates) we see a problem in the resulting JSON output. This is caused by a recursive relationship between Raid and Pirate.

The serialisation becomes recursive as Raid serialisation tries to serialise a Pirate and it goes like this forever until there’s an exception.

We can handle this in a way that works well.

**Any List in a relationship that’s causing this should be annotated with @JsonBackReference**

Go to the Ship.java class, find the pirates property and add the @JsonBackReference annotation to it as follows:

*//Ship.java*

**@JsonBackReference** *// NEW - add this!*

**@OneToMany(**mappedBy **=** "ship"**,** fetch **=** **FetchType.**LAZY**)** *// As before*

**private** **List<Pirate>** pirates**;** *// As before*

Go to the Raid.java class, find the pirates property and add the @JsonBackReference annotation to it as follows:

*//Ship.java*

**@JsonBackReference** *// NEW - add this!*

**@ManyToMany** *// As before*

**@Cascade(**org**.**hibernate**.**annotations**.**CascadeType**.**SAVE\_UPDATE**)**

**@JoinTable(**

name **=** "pirates\_raids"**,**

joinColumns **=** **{@JoinColumn(**name **=** "raid\_id"**,** nullable **=** **false,** updatable **=** **false)},**

inverseJoinColumns **=** **{@JoinColumn(**name**=**"pirate\_id"**,** nullable **=** **false,** updatable **=** **false)}**

**)**

**private** **List<Pirate>** pirates**;** *// As before*

**Note:** The @JsonIgnoreProperties() annotation can be used to a similar effect.

## Other routes

### /pirates/{id}

We need to handle the route where we pass in an ID and get back an individual pirate JSON back. For example if we GET http://localhost:8080/pirates/3 we’re GETting the pirate with ID **3** back.

We want to be able to get pirates with any ID so we need to write a route that allows us to grab the number part for the ID from the URL. We’ll use the @PathVariable annotation.

This allows us to match the {id} part of our URL and get that value into a value (we’ll use a Long) but a String would also work. Let’s code it out..

**@GetMapping(**value **=** "/pirates/{id}"**)**

**public** **ResponseEntity<>** **getPirate(@PathVariable** **Long** id**){**

*// TODO*

**}**

We now need to look at the pirateRepository object and see what methods we have available. We’ll use .findById() which is a method already available to us through JPA.

**@GetMapping(**value **=** "/pirates/{id}"**)**

**public** **ResponseEntity** **getPirate(@PathVariable** **Long** id**){**

**return** **new** **ResponseEntity<>(**pirateRepository**.**findById**(**id**),** **HttpStatus.**OK**);**

**}**

Let’s go to http://localhost:8080/pirates/3 and you should see an individual pirate JSON.

## TASK: 20 mins Complete routes for Ship and Raid

Now complete the following routes and test them with insomnia or in your browser.

* Get all raids/ships
  + GET /raids - returns JSON array of all raids
  + GET /ships - returns JSON array of all ships
* Get one pirate / raid /ship
  + GET /raids/{id} - returns JSON for single raid
  + GET /ships/{id} - returns JSON for single ship

TIP: \* You’ll need to create RaidController and ShipController classes if you don’t already have them. Raid routes go in the RaidController ship routes go in the ShipController

TIP 2: Remember to use @RestController annotation on the controller class you create.

### Solution

*//ShipController.java - imports + package omitted for berevity*

**@RestController**

**public** **class** **ShipController** **{**

**@Autowired**

**ShipRepository** shipRepository**;**

**@GetMapping(**value **=** "/ships"**)**

**public** **ResponseEntity<List<Ship>>** **getAllShips(){**

**return** **new** **ResponseEntity<>(**shipRepository**.**findAll**(),** **HttpStatus.**OK**);**

**}**

**@GetMapping(**value **=** "/ships/{id}"**)**

**public** **ResponseEntity** **getShip(@PathVariable** **Long** id**){**

**return** **new** **ResponseEntity<>(**shipRepository**.**findById**(**id**),** **HttpStatus.**OK**);**

**}**

**}**

*//RaidController.java - imports + package omitted for berevity*

**@RestController**

**public** **class** **RaidController** **{**

**@Autowired**

**RaidRepository** raidRepository**;**

**@GetMapping(**value **=** "/raids"**)**

**public** **ResponseEntity<List<Raid>>** **getAllRaids(){**

**return** **new** **ResponseEntity<>(**raidRepository**.**findAll**(),** **HttpStatus.**OK**);**

**}**

**@GetMapping(**value **=** "/raids/{id}"**)**

**public** **ResponseEntity** **getRaid(@PathVariable** **Long** id**){**

**return** **new** **ResponseEntity<>(**raidRepository**.**findById**(**id**),** **HttpStatus.**OK**);**

**}**

**}**

# RESTful Controllers - create + Insomnia

## Learning Objectives

* Learn how to create REST Controllers that allow us to POST data that gets saved int the database)
* Learn how to use Insomnia to simulate using a front-end or client to make the HTTP POST requests.

## Introduction

We know how to use HTTP GET to get data from our RESTful API.

We need to be able to save data into our database through our JPA ORM. We do this by creating REST Endpoints that allow us to perfrom HTTP POST actions.

From our RESTful routes we will implement the create action.

## REST “Create” Controller Endpoints

Let’s use Insomnia to POST a new pirate to /pirates.

Do this. It will give an error, as we need to create build the “create” route.

We’ve found that this won’t work and we get an error response from our back-end.

That’s because the mappings we have in our controllers are so-far only GetMappings. We need to add PostMapping controllers to allow us to POST data to that endpoint.

**Let’s start with PirateController**. Before we do, we need to think about what we want to give as a response to a POST.

Ask the class for ideas on what a POST response should be.

Ask them what status code specifically it should have. Remember to look at <https://www.restapitutorial.com/httpstatuscodes.html>

Should it be:

* An empty response with 201 (Created)?
* A response with a body containing a message saying “success” or something and have a 201 (Created) status code?
* A response with a body that is the JSON of the object we just posted and a 201 (Created) status.

There’s no actual correct way.. But it’s common using REST principles to do the latter: **A response with a body that is the JSON of the object we just posted and a 201 (Created) status.** This is what we’ll do.

Let’s be clear on what we want this postPirate method to do. It will.

* Accept POST requests on /pirates
* Grab a request body payload which we presume to be a JSON of a pirate object.
* Use the pirateRepository to save that object.
* Return that same object with a ResponseEntity and HTTP Code 201 Created. ( HttpStatus.CREATED )

We need to use annotation @RequestBody to grab the Pirate object.

*//PirateController.java*

**@PostMapping(**value **=** "/pirates"**)**

**public** **ResponseEntity<Pirate>** **postPirate(@RequestBody** **Pirate** pirate**){**

pirateRepository**.**save**(**pirate**);**

**return** **new** **ResponseEntity<>(**pirate**,** **HttpStatus.**CREATED**);**

**}**

## Manually Testing POST - Insomnia

Launch the Mac desktop app Insomnia. If it’s not installed install it with brew cask install insomnia.

Show students how to create a GET action on http://localhost:8080/pirates

Use the following JSON:

{

**"firstName"**: "Pete",

**"lastName"**: "Glasseye",

**"age"**: 33,

**"ship"**: {

**"id"**: 2,

**"name"**: "The Black Pearl"

}

}

**Note: IMPORTANT**: If we post a ship property that doesn’t exist in the DB already, we’ll get problems.

Notice that the **response** we see in Insomnia has the id, as it’s been passed to the DB and indexed.

{

**"firstName"**: "Pete",

**"lastName"**: "Glasseye",

**"age"**: 33,

**"id"**: 11,

**"ship"**: {

**"id"**: 2,

**"name"**: "The Black Pearl"

},

**"raids"**: **null**

}

### Task: 15 mins-ish

Write and (using Insomnia) test the **create** route for Raid and Ship in RaidController and ShipController respectively.

#### Solution

*//ShipController.java*

**@PostMapping(**value **=** "/ships"**)**

**public** **ResponseEntity<Ship>** **createShip(@RequestBody** **Ship** ship**){**

shipRepository**.**save**(**ship**);**

**return** **new** **ResponseEntity<>(**ship**,** **HttpStatus.**CREATED**);**

**}**

JSON Body to test with:

{

**"name"**: "Boaty Mc Boatface"

}

*// RaidController.java*

**@PostMapping(**value **=** "/raids"**)**

**public** **ResponseEntity<Raid>** **createRaid(@RequestBody** **Raid** raid**){**

raidRepository**.**save**(**raid**);**

**return** **new** **ResponseEntity<>(**raid**,** **HttpStatus.**CREATED**);**

**}**

JSON Body to test with:

{

**"location"**: "Port of Leith",

**"loot"**: 100

}

## Summary

We’ve created POST or “create” routes that conform the the RESTful design principles.

* We’ve seen how to use @RequestBody to accept a JSON of that object and allow Spring to automatically de-serialize into a Java object.
* Use the appropriate repository to save the object into the DB.
* We know how to return a response entity with the appropriate body and the 201 created HTTP status code.

**Homework: Annotating One-to-Many and Many-to-One Relationships**

**Learning Objectives**

* Understand one to many relationships
* Be able to use Spring Data REST to create the RESTful routes for a set of resources
* Use insomnia to GET, POST, PUT and DELETE records

**Brief**

Create a one-to-many Spring application using annotations as you’ve done before.

Your application should have the following models: Folders, Files and Users. Users should have many folders, and folders should have many files.

You’ll need to create repositories for each entity.

You’ll need to create controllers for each entity, too, and ensure they implement the **index**, **create** and **show** routes

**MVP**

* Create a system to track files and folders:
  1. A File should have:
     1. a name
     2. extension (e.g. txt, rb, java, ppt)
     3. size
     4. folder
  2. A User should have:
     1. name
     2. a list of folders
  3. A Folder should have:
     1. a title
     2. list of files
     3. a user
* Creata a seeding file DataLoader component to pre-seed the database.
* Test with insomnia to test the

**Planning**

Draw a diagram detail the relationships between models.

**Simple JPA Queries**

**Lesson Duration: 60 minutes**

**Learning Objectives**

* Understand and use the power and simplicity of Spring Data JPA Derived Query Methods

**Introduction**

**Why are we learning this?** Relational databases are great for storing data and preserving its structure. Quite often, when trying to get data we rely on that structure.

For example. We might say *“I’ve got this object, ‘Shop’. Give me all of the ‘Product’ items in the shop”*

This type of query is extremely *relational* in that we are relying on the relationship to get the wanted data.

**Some things to think about**

This is just to get the right mindset and can be skipped if desired.

### Know what data you already have

When doing any sort of query, whether it be with or without Spring or Hibernate - one thing is always true. **You always have some data to get started**.

Consider the following examples in SQL:

* SELECT \* from pirates;, data we have is that we know to query pirates table.
* SELECT \* from pirates WHERE name ='Long John'; we have the name and that we’re querying the pirates table
* SELECT \* from pirates WHERE name ='Jack' AND age = 30. We have name and age, as well as that we are querying the pirates table.

Knowing that there’s always a start point will help us construct queries easier.

Our start point is that we have a Ship meaning we can build a query method that takes a Ship. We can move forward knowing our query needs to take a Ship.

**Know what data you want back**

It may seem obvious, but knowing exactly what you want is extremely important. Some things to consider when figuring out what you want that will make things easier:

1. Do you want a List of a particular object?
2. Do you want just **one** of a partular object?
3. Do you want a primitive type back - such as a String, Integer or Boolean?

The question on whether we want a scalar (just one of an object) or a vector (a list) **depends on the uniqueness involved in the query**.

**Uniqueness**

Uniqueness of a queriy depends on whether a unique id like the id property of a model was used to construct the query or if some non-unique properites are being used to build it like “first name”, “last name”, “age”, “date of birth”. These kinds of things are properties which potentially many could share.

**Unique**

When building a query with known unique identifiers like the id, or some other known key we then also know that there will only be one value, rather than a list of values, returned.

**Non-unique**

When building a query with non-unique properites, then it’s best to save the results of the query as a list.

**Build the Derived Query**

Lets put together a query for getting all the Pirates over a given age.

Since we’re getting **all** the Pirates over an age, it’s a List of Pirate we want back. This is very much a non-unique query result. The method should therefore return a List<Pirate>.

We now know we:

1. Want to find a List<Pirate>
2. Want to give it a number, the age.

In English, we can say:

* "Find all Pirates over age X"

*Derived Queries* are where we write the method **prototype** for a query we want to run, and Spring tries to derive the query from how that method prototype is structured.

The prototype needs to match a certain pattern. That pattern needs to match one of find..By.., read..By.., count..By.., get..By...

So, the general structure of these methods needs to match:

**operation**<**object**>By<**property**><**operator**> where:

* **operation** is one of find, read, count, get.
* **object** is the type of object you expect back from the query.
* **property** is a property on the **object**
* **operator** is one of And, Or.

**Some unrelated examples** (can skip to **for our query** if desired)

If we wish to get all pirates by a specific age, then we would write:

**List<Pirate>** **findPiratesByAge(int** age**);**

Find a Pirate by name:

**List<Pirate>** **findPiratesByName(String** name**);**

Find a Pirate by name, but give only one result (notice use of Distinct:

**Pirate** **findDistinctPiratesByName(String** name**);**

Find people by first name **or** last name:

**List<Person>** **findDistinctPeopleByLastnameOrFirstname(String** lastname**,** **String** firstname**);**

**For our query** "Find all Pirates over a given age", we will add the following to PirateRepository.java:

*//PirateRepository.java*

**public** **interface** **PirateRepository** **extends** **JpaRepository<Pirate,** **Long>** **{** *// AS BEFORE*

**List<Pirate>** **findByAgeGreaterThan(int** age**);**

**}** *// AS BEFORE*

Wherever we have instances of PirateRepository we can now use that query. Spring should parse the method prototype and be able to execute the query we want.

It’s reccomended the queries you write like this are well tested:

*//PirateserviceApplicationTests.java*

**@Test**

**public** **void** **canFindPiratesOver30(){**

**List<Pirate>** found **=** pirateRepository**.**findByAgeGreaterThan**(**30**);**

**}**

**Task (10 mins): Find Raid by location**

* Write a query to find a raid by location
* Write a test to make sure the query works.
* TIP: Think what repository it should be in. We want Raids back so RaidRepository.java

**Solution**

Published with [GitHub Pages](https://pages.github.com/)

**Advanced Derived Queries with JPA**

**Lesson Duration: 30 minutes**

**Learning Objectives**

* Understand how to use derived queries to perform queries across multiple classes.

**Intro**

We’ve seen how to to basic derived queries - ones which perform queries on just one class. We’re now going to see building derived queries across multiple classes.

**Building A Derived Query**

Lets put together a query for getting all the Pirates for a given Raid. [Jump to doing this](https://codeclan.github.io/canvas_notes/course_java/week_3/day_3/2_advanced_derived_queries/2_advanced_derived_queries.html#build_query)

**Build the Derived Query**

We can make things easier by saying **we want all the Pirates for a given Raid id**. This time, we’re traversing multiple models to perform the query it’s possible we just need to remember:

**To filter for an attribute of an associated entity, you can traverse managed relationships by referencing the attribute that maps the association followed by the attribute of the related entity.**

This is relative to where the query is taking place. We want to get back all the **Pirate** objects for a Raid, so we need to do the query in the PirateRepository.

In English, we can think about our query as:

*Find all Pirates for Raid id*

To traverse across to an associated model we need to use the raids property of the Pirate. We’ll use that to access the id on the Raid. So we will write findByRaidsId. We’re ‘in’ doing this from the perspective of the Pirate model, so we need to “jump” to the Raid to get the raid id, and to do that we’ll go via the raids property.

*//PirateRepository.java*

**List<Pirate>** **findByRaidsId(long** id**);** *// Added*

*// 'Raids' is the property in pirates which links to raid.*

Let’s add a test for it, and check we get the only pirate on Raid with id 1.

**@Test**

**public** **void** **findPiratesByRaidId(){**

**List<Pirate>** foundPirates **=** pirateRepository**.**findByRaidsId**(**1L**);**

assertEquals**(**1**,** foundPirates**.**size**());**

assertEquals**(**"Jack"**,** foundPirates**.**get**(**0**).**getFirstName**());**

**}**

Generally we can use this pattern to figure out how to write the query.

**operation**By<**connectedProperty-Property**><**operator**>[]

where:

* **operation** is one of find, read, count, get.
* **connectedProperty** is the related property we need to use to access the property. In our case above we used raids in Pirate to get to the raid id.
* **object** is the type of object you expect back from the query.
* **property** is a property on the **object**
* **operator** is one of And, Or.

**UML Diagram (for reference)**

**Task (10 mins): Find all Ships with a given pirate’s first name on them**

1. Write a query to find all the ships that have pirates with a given first name on them
2. Write a test to make sure the query works.
3. TIP: Think what repository it should be in. We want Ship(s) back so ShipRepository.java

**Solution**

Published with [GitHub Pages](https://pages.github.com/)

**Connect Queries To Controllers**

**Lesson Duration: 30 minutes**

**Learning Objectives**

* Be able to connect a query defined in a JPA Repository (derived or otherwise) to a controller.

**Create a custom route for finding ships that have pirates named**

Let’s create a new custom route in ShipController that allows anyone using our API to use the findByPiratesFirstName query we creted earlier.

*//ShipController.java*

*// These are our custom routes:*

**@GetMapping(**value **=** "/ships/pirates/named/{name}"**)**

**public** **List<Ship>** **findShipsThatHavePiratesNamed(** **@PathVariable** **String** name**){**

**return** shipRepository**.**findByPiratesFirstName**(**name**);**

**}**

@PathVariable makes the annotated variable’s value that of the one provided in {name}. We can then simply pass the name into our already created custom query.

The HTTP Request can now be made to /ships/pirates/named/<name> to give us the Ships we want.

Test by going to: <http://localhost:8080/ships/pirates/named/John>

**Note on Query Strings**

Query strings are arguments we can give to a URL.

For example in a URL you often see something like http://example.com/over/there?name=ferret - the name=ferret part is called a query string. It’s a way of passing extra arguments into a URL.

It’s up to us whether we do this for /ships/pirates/named/Maggie.

An example of this as a query string looks arguably nicer:

localhost:8080/ships/pirates?named=Maggie.

**Since the Query String** version makes for the *better API design*, we’ll go with that. To implement a query string we need to use a @RequestParam annotation on the method argument.

We give the @RequestParam annotation a value to look for which is the *key* for the query string param:

@RequestParam(name="named") String name Will look for a query string with **key** named. Like ?named=Maggie, for example.

The method will look like this:

*//ShipController.java*

*// This is the same as above, but with a Query String instead.*

**@GetMapping(**value **=** "/ships/pirates"**)**

**public** **ResponseEntity<List<Ship>>** **findShipsThatHavePiratesNamedQueryString(**

**@RequestParam(**name**=**"named"**)** **String** name**){**

**return** **new** **ResponseEntity<>(**shipRepository**.**findByPiratesFirstName**(**name**),** **HttpStatus.**OK**);**

**}**

At this point if the longer path version of the route is still in the code, just delete it - we want to go with the query string.

Note on the route for the query string we should leave off the trailing slash /, as it will cause problems if it’s left in.

**When to use a Query String**

We don’t want to over-use query strings. So it’s a good idea to only use them in certain situations. Filtering an existing query is a really good example of when to use it.. It’s worth keeping in mind we can have multiple query strings and and they’re just separated with an ampersand &.

**Task (10 mins): Create a controller for “find / filter raid by location name”**

This task can be quite challenging. Not many students will ‘get it’ allow them to get to the error and then talk them through the solution.

* Write a controller that connects into the “find raid by location” query we completed in the last lesson
* TIP: Think about what controller it should be in. We’re dealing with raids… so RaidController.java
* TIP: Consider using a query string to create a filter.
* TIP: Consider what might happen when our route clashes with the existing GET on /raids.

**Solution**

Published with [GitHub Pages](https://pages.github.com/)

# Lab + Homework

Given the basic code for classes and JpaRepositories for a whisky tracker, your task for this lab is to complete the custom queries and connected them to appropriate RESTful endpoints.

**Whisky**

* name - the name of that individual whisky often uses the name of the distillery. An example is “The Glendronach - Revival” which is from “Glendronach” distillery
* distillery - the distillery object that it’s related to
* year - the year edition that the whisky was released/ bottled. The year is not a way of determining the age.
* age - how long the whisky in the bottle was matured for before bottling

**Distillery**

* name - The name of the whisky distillery.
* region - The region of scotland where the whisky is from. These are one of either {**Lowlands**, **Speyside**, **Highlands**, **Islay**, **Campbelltown**, **Island** }
* whiskys - the list of related whiskies tracked against that distillery

## Task

**Custom Queries + REST** Write each of the following queries and connect them to an appropriate controller:

### MVP

* Get all the whiskies for a particular year
* Get all the distilleries for a particular region
* Get all the whisky from a particular distillery that’s a specific age

### Extensions

* Get all the whisky from a particular region
* Get distilleries that have whiskies that are 12 years old

## Tips

* “get all the whiskies for a particular year” query can be done as a Spring Data JPA Method Query or using a Criteria Query.
* “get all the distilleries for a particular region” query can be done as a Spring Data JPA Method Query or using a Criteria Query.
* It’s reccomended to use ids for queries that need other objects as part of the query.
* The other queries may need Criteria Queries to complete them, and might also need an alias.
* To stop your custom routes recursively referencing each other’s properties, you will need a @JsonIgnoreProperties("some\_property\_you\_need\_to\_ignore") annotation on related properties.

**All Day Lab - Build a Course Booking System**

**Intro**

For this lab, we will build a simple course booking back-end. At the end of the lab we should have a RESTful API that allows connected clients to create course bookings and find useful information about the bookings created.

The API should be built from scratch with Spring using Spring Boot Initialiser.

**MVP**

**Models**

The course reservations API needs to be built with three models with the following properties:

* Course
  1. name - the name of the course eg: Intro to Python
  2. town - the town/city/village where the course is located. We will not bother with full address yet.
  3. star rating - Out of 5, each course has a rating
* Booking
  1. date - a **string** in the form “dd-mm-yy” for the booking date. Dates can be in the future or in the past
* Customer
  1. name - **string** containing customer’s name
  2. town - a **string** containing the town where the customer lives. We will not record an address at this stage
  3. age - the customers age. Useful for marketing purposes

The relationships should be:

* A Course has many Bookings
* A Booking has a Course
* A Customer has many bookings
* A booking has a customer

**Queries + Custom Routes**

Write queries using the derived method we’ve shown. Connect these to suitable RESTful endpoints and decide whether you should use a filter or not for all of:

1. Get all courses with a given rating
2. Get all customers for a given course
3. Get all courses for a given customer
4. Get all bookings for a given date

**Extensions**

### Extension Queries + Routes

1. Get all customers in a given town for a given course
2. Get all customers over a certain age in a given town for a given course

You need to write and test these queries.

**Other Extensions**

* Correctly handle case-insensitive routes with Spring RestController or the IgnoreCase in derived queries.

**Tips / Reminders**

* Stick to the RESTful routes for each resource:
  1. GET /resources
  2. GET /resources/{id}
  3. GET /resources?property=value
  4. POST /resources
  5. PUT /resources/{id}
  6. DELETE /resources/{id}
* You’re allowd to create routes outside of this structure, but you need to be able to justify the design decision. We of course reccomend you stick with this RESTful way of doing things.
* Clearly understand the relationships before coding. Draw it out.
* Be careful with town property - consider how to handle caps.
* Write tests for your queries.
* Use a ddl-auto setting of create-drop when doing development. This means every time the app starts, the schema is created, and when the app stops the schema is destroyed.
* Provide a data loader to seed some initial seed data
* Reminder on which dependencies to use with spring Initialiser:
  1. Web
  2. JPA
  3. H2
  4. DevTools