Accessing JPA Data with REST

This guide walks you through the process of creating an application that accesses relational JPA data through a hypermedia-based RESTful front end.

What you'll build

You'll build a Spring application that let's you create and retrieve `Person` objects stored in a database using Spring Data REST. Spring Data REST takes the features of Spring HATEOAS and Spring Data JPA and combines them together automatically.

> Spring Data REST also supports Spring Data Neo4j, Spring Data Gemfire and Spring Data MongoDB as backend data stores, but those are not part of this guide.

What you'll need

- About 15 minutes
- A favorite text editor or IDE
- JDK 1.8 or later
- Gradle 2.3+ or Maven 3.0+
- You can also import the code straight into your IDE:
  - Spring Tool Suite (STS)
  - IntelliJ IDEA

How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step, or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Build with Gradle.

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-accessing-data-rest.git`
- cd into `gs-accessing-data-rest/initial`

- Jump ahead to Create a domain object.

When you're finished, you can check your results against the code in `gs-accessing-data-rest/complete`.

Build with Gradle

Build with Maven

First you set up a basic build script. You can use any build system you like when building apps with Spring, but the code you need to work with Maven is included here. If you're not familiar with Maven, refer to Building Java Projects with Maven.

Create the directory structure

In a project directory of your choosing, create the following subdirectory structure; for example, with `mkdir -p src/main/java/hello` on *nix systems:

```
└── src
    └── main
        └── java
            └── hello
```

`pom.xml`

```
<?xml version="1.0" encoding="UTF-8"?>

<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.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>



    <groupId>org.springframework</groupId>

    <artifactId>gs-accessing-data-rest</artifactId>
```

```
    <version>0.1.0</version>


    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.2.RELEASE</version>
    </parent>


    <properties>
        <java.version>1.8</java.version>
    </properties>


    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-rest</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
```

```
            <artifactId>h2</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

    <repositories>
        <repository>
            <id>spring-releases</id>
            <url>https://repo.spring.io/libs-release</url>
        </repository>
```

```
    </repositories>

    <pluginRepositories>

        <pluginRepository>

            <id>spring-releases</id>

            <url>https://repo.spring.io/libs-release</url>

        </pluginRepository>

    </pluginRepositories>

</project>
```

The Spring Boot Maven plugin provides many convenient features:

- It collects all the jars on the classpath and builds a single, runnable "über-jar", which makes it more convenient to execute and transport your service.
- It searches for the `public static void main()` method to flag as a runnable class.
- It provides a built-in dependency resolver that sets the version number to match Spring Boot dependencies. You can override any version you wish, but it will default to Boot's chosen set of versions.

Build with your IDE

Create a domain object

Create a new domain object to present a person.

`src/main/java/hello/Person.java`

```
package hello;



import javax.persistence.Entity;

import javax.persistence.GeneratedValue;
```

```java
import javax.persistence.GenerationType;

import javax.persistence.Id;



@Entity

public class Person {



	@Id

	@GeneratedValue(strategy = GenerationType.AUTO)

	private long id;



	private String firstName;

	private String lastName;



	public String getFirstName() {

		return firstName;

	}



	public void setFirstName(String firstName) {

		this.firstName = firstName;

	}



	public String getLastName() {

		return lastName;
```

```
        }


        public void setLastName(String lastName) {

                this.lastName = lastName;

        }

}
```

The `Person` has a first name and a last name. There is also an id object that is configured to be automatically generated so you don't have to deal with that.

Create a Person repository

Next you need to create a simple repository.

`src/main/java/hello/PersonRepository.java`

```
package hello;


import java.util.List;


import org.springframework.data.repository.PagingAndSortingRepository;

import org.springframework.data.repository.query.Param;

import
org.springframework.data.rest.core.annotation.RepositoryRestResource;


@RepositoryRestResource(collectionResourceRel = "people", path =
"people")

public interface PersonRepository extends
PagingAndSortingRepository<Person, Long> {
```

```
        List<Person> findByLastName(@Param("name") String name);



}
```

This repository is an interface and will allow you to perform various operations involving `Person` objects. It gets these operations by extending the PagingAndSortingRepositoryinterface defined in Spring Data Commons.

At runtime, Spring Data REST will create an implementation of this interface automatically. Then it will use the @RepositoryRestResource annotation to direct Spring MVC to create RESTful endpoints at `/people`.

> `@RepositoryRestResource` is not required for a repository to be exported. It is only used to change the export details, such as using `/people` instead of the default value of `/persons`.

Here you have also defined a custom query to retrieve a list of `Person` objects based on the lastName. You'll see how to invoke it further down in this guide.

Make the application executable

Although it is possible to package this service as a traditional WAR file for deployment to an external application server, the simpler approach demonstrated below creates a standalone application. You package everything in a single, executable JAR file, driven by a good old Java `main()` method. Along the way, you use Spring's support for embedding the Tomcat servlet container as the HTTP runtime, instead of deploying to an external instance.

`src/main/java/hello/Application.java`

```
package hello;



import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;



@SpringBootApplication
```

```
public class Application {



        public static void main(String[] args) {

                SpringApplication.run(Application.class, args);

        }

}
```

`@SpringBootApplication` is a convenience annotation that adds all of the following:

- `@Configuration` tags the class as a source of bean definitions for the application context.
- `@EnableAutoConfiguration` tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
- Normally you would add `@EnableWebMvc` for a Spring MVC app, but Spring Boot adds it automatically when it sees **spring-webmvc** on the classpath. This flags the application as a web application and activates key behaviors such as setting up a `DispatcherServlet`.
- `@ComponentScan` tells Spring to look for other components, configurations, and services in the `hello` package, allowing it to find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there wasn't a single line of XML? No **web.xml** file either. This web application is 100% pure Java and you didn't have to deal with configuring any plumbing or infrastructure.

Spring Boot automatically spins up Spring Data JPA to create a concrete implementation of the `PersonRepository` and configure it to talk to a back end in-memory database using JPA.

Spring Data REST builds on top of Spring MVC. It creates a collection of Spring MVC controllers, JSON converters, and other beans needed to provide a RESTful front end. These components link up to the Spring Data JPA backend. Using Spring Boot this is all autoconfigured; if you want to investigate how that works, you could start by looking at the `RepositoryRestMvcConfiguration` in Spring Data REST.

Build an executable JAR

You can run the application from the command line with Gradle or Maven. Or you can build a single executable JAR file that contains all the necessary dependencies, classes, and resources, and run that. This makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you are using Gradle, you can run the application using `./gradlew bootRun`. Or you can build the JAR file using `./gradlew build`. Then you can run the JAR file:

```
java -jar build/libs/gs-accessing-data-rest-0.1.0.jar
```

If you are using Maven, you can run the application using `./mvnw spring-boot:run`. Or you can build the JAR file with `./mvnw clean package`. Then you can run the JAR file:

```
java -jar target/gs-accessing-data-rest-0.1.0.jar
```

> The procedure above will create a runnable JAR. You can also opt to build a classic WAR file instead.

Logging output is displayed. The service should be up and running within a few seconds.

Test the application

Now that the application is running, you can test it. You can use any REST client you wish. The following examples use the *nix tool `curl`.

First you want to see the top level service.

```
$ curl http://localhost:8080

{
  "_links" : {
    "people" : {
      "href" : "http://localhost:8080/people{?page,size,sort}",
      "templated" : true
```

```
    }

  }

}
```

Here you get a first glimpse of what this server has to offer. There is a **people** link located at http://localhost:8080/people. It has some options such as `?page`, `?size`, and `?sort`.

Spring Data REST uses the HAL format for JSON output. It is flexible and offers a convenient way to supply links adjacent to the data that is served.

```
$ curl http://localhost:8080/people

{

  "_links" : {

    "self" : {

      "href" : "http://localhost:8080/people{?page,size,sort}",

      "templated" : true

    },

    "search" : {

      "href" : "http://localhost:8080/people/search"

    }

  },

  "page" : {

    "size" : 20,

    "totalElements" : 0,

    "totalPages" : 0,

    "number" : 0

  }
```

```
}
```

There are currently no elements and hence no pages. Time to create a new `Person`!

```
$ curl -i -X POST -H "Content-Type:application/json" -d "{  \"firstName\" : \"Frodo\",  \"lastName\" : \"Baggins\" }" http://localhost:8080/people

HTTP/1.1 201 Created

Server: Apache-Coyote/1.1

Location: http://localhost:8080/people/1

Content-Length: 0

Date: Wed, 26 Feb 2014 20:26:55 GMT
```

- `-i` ensures you can see the response message including the headers. The URI of the newly created `Person` is shown
- `-X POST` signals this a POST used to create a new entry
- `-H "Content-Type:application/json"` sets the content type so the application knows the payload contains a JSON object
- `-d "{ \"firstName\" : \"Frodo\", \"lastName\" : \"Baggins\" }"` is the data being sent. Double quotes inside the data need to be escaped as `\"`.

Notice how the previous `POST` operation includes a `Location` header. This contains the URI of the newly created resource. Spring Data REST also has two methods on `RepositoryRestConfiguration.setReturnBodyOnCreate(…)` and `setReturnBodyOnUpdate(…)` which you can use to configure the framework to immediately return the representation of the resource just created. `RepositoryRestConfiguration.setReturnBodyForPutAndPost(…)` is a short cut method to enable representation responses for creates and updates.

From this you can query for all people:

```
$ curl http://localhost:8080/people

{

  "_links" : {

    "self" : {

      "href" : "http://localhost:8080/people{?page,size,sort}",
```

```
    "templated" : true
  },
  "search" : {
    "href" : "http://localhost:8080/people/search"
  }
},
"_embedded" : {
  "persons" : [ {
    "firstName" : "Frodo",
    "lastName" : "Baggins",
    "_links" : {
      "self" : {
        "href" : "http://localhost:8080/people/1"
      }
    }
  } ]
},
"page" : {
  "size" : 20,
  "totalElements" : 1,
  "totalPages" : 1,
  "number" : 0
}
```

```
}
```

The **persons** object contains a list with Frodo. Notice how it includes a **self** link. Spring Data REST also uses Evo Inflector to pluralize the name of the entity for groupings.

You can query directly for the individual record:

```
$ curl http://localhost:8080/people/1

{

  "firstName" : "Frodo",

  "lastName" : "Baggins",

  "_links" : {

    "self" : {

      "href" : "http://localhost:8080/people/1"

    }

  }

}
```

This might appear to be purely web based, but behind the scenes, there is an H2 relational database. In production, you would probably use a real one, like PostgreSQL.

In this guide, there is only one domain object. With a more complex system where domain objects are related to each other, Spring Data REST will render additional links to help navigate to connected records.

Find all the custom queries:

```
$ curl http://localhost:8080/people/search

{

  "_links" : {

    "findByLastName" : {
```

```
      "href" :
"http://localhost:8080/people/search/findByLastName{?name}",
      "templated" : true
    }
  }
}
```

You can see the URL for the query including the HTTP query parameter `name`. If you'll notice, this matches the `@Param("name")` annotation embedded in the interface.

To use the `findByLastName` query, do this:

```
$ curl http://localhost:8080/people/search/findByLastName?name=Baggins
{
  "_embedded" : {
    "persons" : [ {
      "firstName" : "Frodo",
      "lastName" : "Baggins",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/people/1"
        }
      }
    } ]
  }
}
```

Because you defined it to return `List<Person>` in the code, it will return all of the results. If you had defined it only return `Person`, it will pick one of the Person objects to return. Since this can be unpredictable, you probably don't want to do that for queries that can return multiple entries.

You can also issue `PUT`, `PATCH`, and `DELETE` REST calls to either replace, update, or delete existing records.

```
$ curl -X PUT -H "Content-Type:application/json" -d "{ \"firstName\":
\"Bilbo\", \"lastName\": \"Baggins\" }" http://localhost:8080/people/1

$ curl http://localhost:8080/people/1

{

  "firstName" : "Bilbo",

  "lastName" : "Baggins",

  "_links" : {

    "self" : {

      "href" : "http://localhost:8080/people/1"

    }

  }

}

$ curl -X PATCH -H "Content-Type:application/json" -d "{ \"firstName\":
\"Bilbo Jr.\" }" http://localhost:8080/people/1

$ curl http://localhost:8080/people/1

{

  "firstName" : "Bilbo Jr.",

  "lastName" : "Baggins",

  "_links" : {

    "self" : {
```

```
      "href" : "http://localhost:8080/people/1"
    }
  }
}
```

PUT replaces an entire record. Fields not supplied will be replaced with null. PATCH can be used to update a subset of items.

You can delete records:

```
$ curl -X DELETE http://localhost:8080/people/1

$ curl http://localhost:8080/people

{
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/people{?page,size,sort}",
      "templated" : true
    },
    "search" : {
      "href" : "http://localhost:8080/people/search"
    }
  },
  "page" : {
    "size" : 20,
    "totalElements" : 0,
    "totalPages" : 0,
    "number" : 0
```

```
    }

}
```

A very convenient aspect of this hypermedia-driven interface is how you can discover all the RESTful endpoints using curl (or whatever REST client you are using). There is no need to exchange a formal contract or interface document with your customers.

Summary

Congratulations! You've just developed an application with a hypermedia-based RESTful front end and a JPA-based back end.

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.