< ALL GUIDES

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 Will Build

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

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 Need

- About 15 minutes
- A favorite text editor or IDE
- JDK 1.8 or later
- Gradle 4+ or Maven 3.2+
- 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 Starting with Spring Initializr.

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 finish, you can check your results against the code in `gs-accessing-data-rest/complete`.

Starting with Spring Initializr

For all Spring applications, you should start with the Spring Initializr. The Initializr offers a fast way to pull in all the dependencies you need for an application and does a lot of the set up for you. This example needs the Rest Repositories, Spring Data JPA, and H2 dependencies.

The following listing shows the `pom.xml` file created when you choose Maven:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apach
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.3.2.RELEASE</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>accessing-data-rest</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>accessing-data-rest</name>
        <description>Demo project for Spring Boot</description>

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

        <dependencies>
```

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

		<dependency>
			<groupId>com.h2database</groupId>
			<artifactId>h2</artifactId>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-test</artifactId>
			<scope>test</scope>
			<exclusions>
				<exclusion>
					<groupId>org.junit.vintage</groupId>
					<artifactId>junit-vintage-engine</artifac
				</exclusion>
			</exclusions>
		</dependency>
	</dependencies>

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

</project>
```

The following listing shows the `build.gradle` file created when you choose Gradle:

```
plugins {
	id 'org.springframework.boot' version '2.3.2.RELEASE'
	id 'io.spring.dependency-management' version '1.0.8.RELEASE'
	id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
	mavenCentral()
}

dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
        implementation 'org.springframework.boot:spring-boot-starter-data-rest'
        runtimeOnly 'com.h2database:h2'
        testImplementation('org.springframework.boot:spring-boot-starter-test') {
                exclude group: 'org.junit.vintage', module: 'junit-vintage-engine
        }
}

test {
        useJUnitPlatform()
}
```

Create a Domain Object

Create a new domain object to present a person, as the following listing (in `src/main/java/com/example/accessingdatarest/Person.java`) shows:

COPY

```
package com.example.accessingdatarest;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
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` object has a first name and a last name. (There is also an ID object that is configured to be automatically generated, so you need not deal with that.)

Create a Person Repository

Next, you need to create a simple repository, as the following listing (in `src/main/java/com/example/accessingdatarest/PersonRepository.java`) shows:

COPY

```
package com.example.accessingdatarest;

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 that lets you perform various operations involving `Person` objects. It gets these operations by extending the `PagingAndSortingRepository` interface that is defined in Spring Data Commons.

At runtime, Spring Data REST automatically creates an implementation of this interface. Then it uses 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 used only 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 can see how to invoke it later in this guide.

`@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. For example, if `spring-webmvc` is on the classpath, this annotation 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 `com/example` package, letting it find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there was not a single line of XML? There is no `web.xml` file, either. This web application is 100% pure Java and you did not 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 by using JPA.

Spring Data REST builds on top of Spring MVC. It creates a collection of Spring MVC controllers, JSON converters, and other beans to provide a RESTful front end. These components link up to the Spring Data JPA backend. When you use Spring Boot, this is all autoconfigured. If you want to investigate how that works, 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. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar 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 use Gradle, you can run the application by using `./gradlew bootRun`. Alternatively, you can build the JAR file by using `./gradlew build` and then run the JAR file, as follows:

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

If you use Maven, you can run the application by using `./mvnw spring-boot:run`. Alternatively, you can build the JAR file with `./mvnw clean package` and then run the JAR file, as follows:

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

The steps described here create a runnable JAR. You can also build a classic WAR file.

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. The following example shows how to do so:

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

The preceding example provides 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.

The following example shows how to see the people records (none at present):

```
$ curl http://localhost:8080/people
{
  "_embedded" : {
    "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`! The following listing shows how to do so:

```
$ curl -i -H "Content-Type:application/json" -d '{"firstName": "Frodo", "lastName
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.
- `-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.
- If you are on Windows, the command above will work on WSL. If you can't install WSL, you might need to replace the single quotes with double quotes and escape the existing double quotes, i.e. `-d "{\"firstName\": \"Frodo\", \"lastName\": \"Baggins\"}"`.

Notice how the response to the `POST` operation includes a `Location` header. This contains the URI of the newly created resource. Spring Data REST also has two methods (`RepositoryRestConfiguration.setReturnBodyOnCreate(…)` and `setReturnBodyOnUpdate(…)`) that you can use to configure the framework to immediately return the representation of the resource just created.

`RepositoryRestConfiguration.setReturnBodyForPutAndPost(…)` is a shortcut method to enable representation responses for create and update operations.

You can query for all people, as the following example shows:

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

The `people` object contains a list that includes `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, as follows:

```
$ 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. However, behind the scenes, there is an H2 relational database. In production, you would probably use a real one, such as 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 renders additional links to help navigate to connected records.

You can find all the custom queries, as shown in the following example:

```
$ 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`. Note that this matches the `@Param("name")` annotation embedded in the interface.

The following example shows how to use the `findByLastName` query:

```
$ 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 returns all of the results. If you had defined it to return only `Person`, it picks one of the `Person` objects to return.

Since this can be unpredictable, you probably do not want to do that for queries that can return multiple entries.

You can also issue `PUT`, `PATCH`, and `DELETE` REST calls to replace, update, or delete existing records (respectively). The following example uses a `PUT` call:

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

The following example uses a `PATCH` call:

```
$ curl -X PATCH -H "Content-Type:application/json" -d '{"firstName": "Bilbo Jr."}
$ 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 are replaced with `null`. You can use `PATCH` to update a subset of items.

You can also delete records, as the following example shows:

```
$ 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 convenient aspect of this hypermedia-driven interface is that you can discover all the RESTful endpoints by using curl (or whatever REST client you like). You need not exchange a formal contract or interface document with your customers.

Summary

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

See Also

The following guides may also be helpful:

- Building a Hypermedia-Driven RESTful Web Service
- Accessing GemFire Data with REST
- Accessing MongoDB Data with REST
- Accessing data with MySQL
- Accessing Neo4j Data with REST
- Consuming a RESTful Web Service
- Consuming a RESTful Web Service with AngularJS
- Consuming a RESTful Web Service with jQuery
- Consuming a RESTful Web Service with rest.js
- Securing a Web Application
- Building REST services with Spring

- Building an Application with Spring Boot
- Creating API Documentation with Restdocs
- Enabling Cross Origin Requests for a RESTful Web Service

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

All guides are released with an ASLv2 license for the code, and an Attribution, NoDerivatives creative commons license for the writing.