< ALL GUIDES

Accessing Data with JPA

This guide walks you through the process of building an application that uses Spring Data JPA to store and retrieve data in a relational database.

What You Will build

You will build an application that stores `Customer` POJOs (Plain Old Java Objects) in a memory-based database.

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.

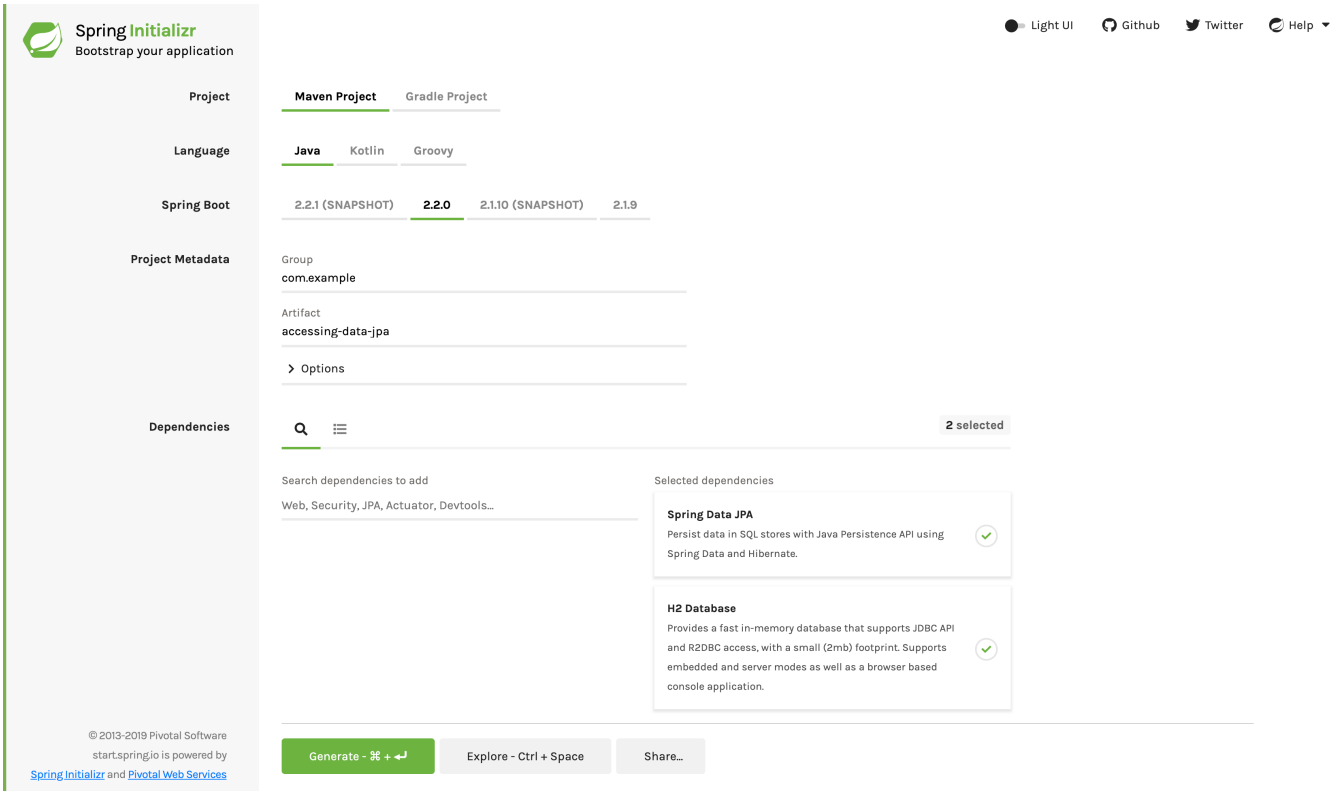
To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git:
`git clone https://github.com/spring-guides/gs-accessing-data-jpa.git`
- cd into `gs-accessing-data-jpa/initial`
- Jump ahead to Define a Simple Entity.

When you finish, you can check your results against the code in `gs-accessing-data-jpa/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 JPA and H2 dependencies. The following image shows the Initializr set up for this sample project:



The preceding image shows the Initializr with Maven chosen as the build tool. You can also use Gradle. It also shows values of `com.example` and `accessing-data-jpa` as the Group and Artifact, respectively. You will use those values throughout the rest of this sample.

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.apache.or
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.2.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>accessing-data-jpa</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>accessing-data-jpa</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>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</artifactId>
                </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 '2.2.2.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'
    runtimeOnly 'com.h2database:h2'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
    }
}

test {
    useJUnitPlatform()
}
```

Define a Simple Entity

In this example, you store `Customer` objects, each annotated as a JPA entity. The following listing shows the Customer class (in `src/main/java/com/example/accessingdatajpa/Customer.java`):

COPY

```
package com.example.accessingdatajpa;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Customer {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;
  private String firstName;
  private String lastName;

  protected Customer() {}
```

```
  public Customer(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  @Override
  public String toString() {
    return String.format(
        "Customer[id=%d, firstName='%s', lastName='%s']",
        id, firstName, lastName);
  }

  public Long getId() {
    return id;
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }
}
```

Here you have a `Customer` class with three attributes: `id`, `firstName`, and `lastName`. You also have two constructors. The default constructor exists only for the sake of JPA. You do not use it directly, so it is designated as `protected`. The other constructor is the one you use to create instances of `Customer` to be saved to the database.

The `Customer` class is annotated with `@Entity`, indicating that it is a JPA entity. (Because no `@Table` annotation exists, it is assumed that this entity is mapped to a table named `Customer`.)

The `Customer` object's `id` property is annotated with `@Id` so that JPA recognizes it as the object's ID. The `id` property is also annotated with `@GeneratedValue` to indicate that the ID should be generated automatically.

The other two properties, `firstName` and `lastName`, are left unannotated. It is assumed that they are mapped to columns that share the same names as the properties themselves.

The convenient `toString()` method print outs the customer's properties.

Create Simple Queries

Spring Data JPA focuses on using JPA to store data in a relational database. Its most compelling feature is the ability to create repository implementations automatically, at runtime, from a repository interface.

To see how this works, create a repository interface that works with `Customer` entities as the following listing (in `src/main/java/com/example/accessingdatajpa/CustomerRepository.java`) shows:

```
package com.example.accessingdatajpa;

import java.util.List;

import org.springframework.data.repository.CrudRepository;

public interface CustomerRepository extends CrudRepository<Customer, Long> {

  List<Customer> findByLastName(String lastName);

  Customer findById(long id);
}
```

`CustomerRepository` extends the `CrudRepository` interface. The type of entity and ID that it works with, `Customer` and `Long`, are specified in the generic parameters on `CrudRepository`. By extending `CrudRepository`, `CustomerRepository` inherits several methods for working with `Customer` persistence, including methods for saving, deleting, and finding `Customer` entities.

Spring Data JPA also lets you define other query methods by declaring their method signature. For example, `CustomerRepository` includes the `findByLastName()` method.

In a typical Java application, you might expect to write a class that implements `CustomerRepository`. However, that is what makes Spring Data JPA so powerful: You need not write an implementation of the repository interface. Spring Data JPA creates an implementation when you run the application.

Now you can wire up this example and see what it looks like!

Create an Application Class

Spring Initializr creates a simple class for the application. The following listing shows the class that Initializr created for this example (in `src/main/java/com/example/accessingdatajpa/AccessingDataJpaApplication.java`):

```
package com.example.accessingdatajpa;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class AccessingDataJpaApplication {

  public static void main(String[] args) {
    SpringApplication.run(AccessingDataJpaApplication.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. 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.

Now you need to modify the simple class that the Initializr created for you. To get output (to the console, in this example), you need to set up a logger. Then you need to set up some data and use it to generate output. The following listing shows the finished `AccessingDataJpaApplication` class (in `src/main/java/com/example/accessingdatajpa/AccessingDataJpaApplication.java`):

```
package com.example.accessingdatajpa;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
```

```
@SpringBootApplication
public class AccessingDataJpaApplication {

  private static final Logger log =
LoggerFactory.getLogger(AccessingDataJpaApplication.class);

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

  @Bean
  public CommandLineRunner demo(CustomerRepository repository) {
    return (args) -> {
      // save a few customers
      repository.save(new Customer("Jack", "Bauer"));
      repository.save(new Customer("Chloe", "O'Brian"));
      repository.save(new Customer("Kim", "Bauer"));
      repository.save(new Customer("David", "Palmer"));
      repository.save(new Customer("Michelle", "Dessler"));

      // fetch all customers
      log.info("Customers found with findAll():");
      log.info("-------------------------------");
      for (Customer customer : repository.findAll()) {
        log.info(customer.toString());
      }
      log.info("");

      // fetch an individual customer by ID
      Customer customer = repository.findById(1L);
      log.info("Customer found with findById(1L):");
      log.info("--------------------------------");
      log.info(customer.toString());
      log.info("");

      // fetch customers by last name
      log.info("Customer found with findByLastName('Bauer'):");
      log.info("--------------------------------------------");
      repository.findByLastName("Bauer").forEach(bauer -> {
        log.info(bauer.toString());
      });
      // for (Customer bauer : repository.findByLastName("Bauer")) {
      //  log.info(bauer.toString());
      // }
      log.info("");
    };
  }

}
```

`Application` includes a `demo()` method that puts the `CustomerRepository` through a few tests. First, it fetches the `CustomerRepository` from the Spring application context. Then it saves a handful of `Customer` objects, demonstrating the `save()` method and setting up some data to work with. Next, it calls `findAll()` to fetch all `Customer` objects

from the database. Then it calls `findOne()` to fetch a single `Customer` by its ID. Finally, it calls `findByLastName()` to find all customers whose last name is "Bauer". The `demo()` method returns a `CommandLineRunner` bean that automatically runs the code when the application launches.

The `AccessingDataJpaApplication` class includes a `main()` method that puts the `CustomerRepository` through a few tests. First, it fetches the `CustomerRepository` from the Spring application context. Then it saves a handful of `Customer` objects, demonstrating the `save()` method and setting up some data to use. Next, it calls `findAll()` to fetch all `Customer` objects from the database. Then it calls `findOne()` to fetch a single `Customer` by its ID. Finally, it calls `findByLastName()` to find all customers whose last name is "Bauer".

> By default, Spring Boot enables JPA repository support and looks in the package (and its subpackages) where `@SpringBootApplication` is located. If your configuration has JPA repository interface definitions located in a package that is not visible, you can point out alternate packages by using `@EnableJpaRepositories` and its type-safe `basePackageClasses=MyRepository.class` parameter.

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-jpa-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-jpa-0.1.0.jar
```

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

When you run your application, you should see output similar to the following:

```
== Customers found with findAll():
Customer[id=1, firstName='Jack', lastName='Bauer']
Customer[id=2, firstName='Chloe', lastName='O'Brian']
Customer[id=3, firstName='Kim', lastName='Bauer']
Customer[id=4, firstName='David', lastName='Palmer']
Customer[id=5, firstName='Michelle', lastName='Dessler']

== Customer found with findById(1L):
Customer[id=1, firstName='Jack', lastName='Bauer']

== Customer found with findByLastName('Bauer'):
Customer[id=1, firstName='Jack', lastName='Bauer']
Customer[id=3, firstName='Kim', lastName='Bauer']
```

Summary

Congratulations! You have written a simple application that uses Spring Data JPA to save objects to and fetch them from a database, all without writing a concrete repository implementation.

> If you want to expose JPA repositories with a hypermedia-based RESTful front end with little effort, you might want to read Accessing JPA Data with REST.

See Also

The following guides may also be helpful:

- Accessing JPA Data with REST
- Accessing Data with Gemfire
- Accessing Data with MongoDB
- Accessing data with MySQL

- Accessing Data with Neo4j

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