**Spring Data JPA Tutorial: Introduction**

Petri Kainulainen November 30, 2014[44 comments](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-introduction/#comments)

[spring data](https://www.petrikainulainen.net/tag/spring-data/), [spring data jpa](https://www.petrikainulainen.net/tag/spring-data-jpa/), [Spring Framework](https://www.petrikainulainen.net/tag/spring-framework/)

Creating repositories that use the Java Persistence API is a cumbersome process that takes a lot of time and requires a lot of boilerplate code. We can eliminate some boilerplate code by following these steps:

1. Create an abstract base repository class that provides CRUD operations for entities.
2. Create the concrete repository class that extends the abstract base repository class.

The problem of this approach is that we still have to write the code that creates our database queries and invokes them. To make matters worse, we have to do this every time when we want to create a new database query. **This is a waste of time**.

What would you say if I would tell you that we can create JPA repositories without writing any boilerplate code?

The odds are that you might not believe me, but Spring Data JPA helps us to do just that. [The website of the Spring Data JPA project states that](http://projects.spring.io/spring-data-jpa/):

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that’s actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically

This blog post provides an introduction to Spring Data JPA. We will learn what Spring Data JPA really is and take a quick look at the Spring Data repository interfaces.

Let’s get started.

**My "Test With Spring" course helps you to write unit, integration, and end-to-end tests for Spring and Spring Boot Web Apps:**

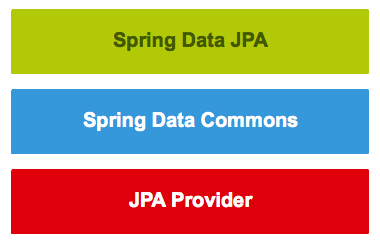
[**CHECK IT OUT >>**](https://www.testwithspring.com/save-time-by-writing-less-test-code/?utm_source=petri-kainulainen&utm_medium=web&utm_content=spring-data-jpa-start&utm_campaign=test-with-spring-course-presales)

**What Spring Data JPA Is?**

**Spring Data JPA is not a JPA provider**. It is a library / framework that adds an extra layer of abstraction on the top of our JPA provider. If we decide to use Spring Data JPA, the repository layer of our application contains three layers that are described in the following:

* [Spring Data JPA](http://projects.spring.io/spring-data-jpa/) provides support for creating JPA repositories by extending the Spring Data repository interfaces.
* [Spring Data Commons](https://github.com/spring-projects/spring-data-commons) provides the infrastructure that is shared by the datastore specific [Spring Data projects](http://projects.spring.io/spring-data/).
* The JPA Provider implements the Java Persistence API.

The following figure illustrates the structure of our repository layer:

[](https://www.petrikainulainen.net/wp-content/uploads/springdatajpalayers.png)

**Additional Reading:**

* [Spring Data JPA versus JPA: What’s the difference?](http://stackoverflow.com/questions/16148188/spring-data-jpa-versus-jpa-whats-the-difference)

At first it seems that Spring Data JPA makes our application more complicated, and in a way that is true. It does add an additional layer to our repository layer, but at the same time it frees us from writing any boilerplate code.

That sounds like a good tradeoff. Right?

**Introduction to Spring Data Repositories**

The power of Spring Data JPA lies in the repository abstraction that is provided by the Spring Data Commons project and extended by the datastore specific sub projects.

We can use Spring Data JPA without paying any attention to the actual implementation of the repository abstraction, but we have to be familiar with the Spring Data repository interfaces. These interfaces are described in the following:

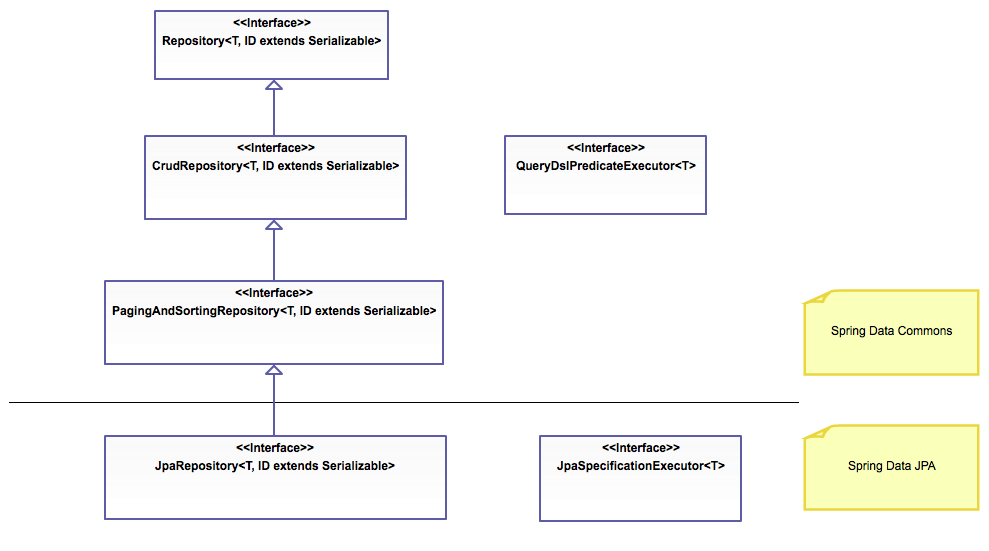
**First**, the Spring Data Commons project provides the following interfaces:

* The [*Repository<T, ID extends Serializable>*](http://docs.spring.io/spring-data/commons/docs/current/api/index.html?org/springframework/data/repository/Repository.html) interface is a marker interface that has two purposes:
  1. It captures the type of the managed entity and the type of the entity’s id.
  2. It helps the Spring container to discover the “concrete” repository interfaces during classpath scanning.
* The *[CrudRepository<T, ID extends Serializable>](http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html" \t "_blank)* interface provides CRUD operations for the managed entity.
* The *[PagingAndSortingRepository<T, ID extends Serializable>](http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/PagingAndSortingRepository.html" \t "_blank)* interface declares the methods that are used to sort and paginate entities that are retrieved from the database.
* The *[QueryDslPredicateExecutor<T>](http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/querydsl/QueryDslPredicateExecutor.html" \t "_blank)* interface is not a “repository interface”. It declares the methods that are used to retrieve entities from the database by using [QueryDsl](http://www.querydsl.com/" \t "_blank) *Predicate* objects.

**Second**, the Spring Data JPA project provides the following interfaces:

* The *[JpaRepository<T, ID extends Serializable>](http://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html" \t "_blank)* interface is a JPA specific repository interface that combines the methods declared by the common repository interfaces behind a single interface.
* The *[JpaSpecificationExecutor<T>](http://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaSpecificationExecutor.html" \t "_blank)* interface is not a “repository interface”. It declares the methods that are used to retrieve entities from the database by using [*Specification<T>*](http://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/domain/Specification.html) objects that use the JPA criteria API.

The repository hierarchy looks as follows:

[](https://www.petrikainulainen.net/wp-content/uploads/springdatajrepositories.png)

That is nice, but how can we use them?

That is a fair question. The next parts of this tutorial will answer to that question, but essentially we have to follow these steps:

1. Create a repository interface and extend one of the repository interfaces provided by Spring Data.
2. Add custom query methods to the created repository interface (if we need them that is).
3. Inject the repository interface to another component and use the implementation that is provided automatically by Spring.

Let’s move on and summarize what we learned from this blog post.

**My "Test With Spring" course helps you to write unit, integration, and end-to-end tests for Spring and Spring Boot Web Apps:**

[**CHECK IT OUT >>**](https://www.testwithspring.com/save-time-by-writing-less-test-code/?utm_source=petri-kainulainen&utm_medium=web&utm_content=spring-data-jpa-end&utm_campaign=test-with-spring-course-presales)

**Summary**

This blog post has taught us two things:

* Spring Data JPA is not a JPA provider. It simply “hides” the Java Persistence API (and the JPA provider) behind its repository abstraction.
* Spring Data provides multiple repository interfaces that are used for different purposes.

The next part of this tutorial describes how we can [get the required dependencies](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-getting-the-required-dependencies/).

**Spring Data JPA Tutorial**

Creating repositories that use the Java Persistence API is a cumbersome process that takes a lot of time and requires a lot of boilerplate code.

We can remove some boilerplate code by creating an abstract base repository class that provides CRUD operations for our entities. This means that we can create concrete repositories by extending that base repository class. In other words, we have to write the code that provides CRUD operations only once.

The problem is that this does not eliminate all boilerplate code. We still have write the code that creates our database queries and invokes them. To make matters worse, we have to do this every time when we create a new database query.

**This is a waste of time!**

[Spring Data JPA](http://www.springsource.org/spring-data/jpa) was created to solve this problem.

**My "Test With Spring" course helps you to write unit, integration, and end-to-end tests for Spring and Spring Boot Web Apps:**

[**CHECK IT OUT >>**](https://www.testwithspring.com/save-time-by-writing-less-test-code/?utm_source=petri-kainulainen&utm_medium=web&utm_content=spring-data-jpa-start&utm_campaign=test-with-spring-course-presales)

**Introducing: Spring Data JPA Tutorial**

This tutorial describes how you can create JPA repositories without writing any boilerplate code, and it consists of the following blog posts

* [Spring Data JPA Tutorial: Introduction](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-introduction/) provides a quick introduction to Spring Data JPA. It describes what Spring Data JPA really is and provides an overview of the Spring Data repository interfaces.
* [Spring Data JPA Tutorial: Getting the Required Dependencies](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-getting-the-required-dependencies/) describes how you can get the required dependencies.
* [Spring Data JPA Tutorial: Configuration](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-one-configuration/) helps you to configure the persistence layer of a Spring application that uses Spring Data JPA and Hibernate.
* [Spring Data JPA Tutorial: CRUD](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-two-crud/) describes how you can create a Spring Data JPA repository that provides CRUD operations for an entity.
* [Spring Data JPA Tutorial: Introduction to Query Methods](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-introduction-to-query-methods/) gives a very short introduction to query methods. It also describes what kind of values you can return from your query methods and how you can pass method parameters to your query methods.
* [Spring Data JPA Tutorial: Creating Database Queries From Method Names](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-creating-database-queries-from-method-names/) describes how you can create database queries from the method names of your query methods.
* [Spring Data JPA Tutorial: Creating Database Queries With the *@Query* Annotation](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-creating-database-queries-with-the-query-annotation/) describes how you can create database queries by annotating your query methods with the *@Query* annotation.
* [Spring Data JPA Tutorial: Creating Database Queries With Named Queries](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-creating-database-queries-with-named-queries/) describes how you can create database queries by using named queries.
* [Spring Data JPA Tutorial: Creating Database Queries With the JPA Criteria API](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-four-jpa-criteria-queries/) describes how you can create dynamic queries by using the JPA Criteria API.
* [Spring Data JPA Tutorial: Creating Database Queries With Querydsl](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-five-querydsl/) describes how you can create dynamic database queries by using Querydsl.
* [Spring Data JPA Tutorial: Sorting](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-six-sorting/) describes how you can sort your query results.
* [Spring Data JPA Tutorial: Pagination](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-seven-pagination/) helps you to paginate your query results.
* [Spring Data JPA Tutorial: Auditing, Part One](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-auditing-part-one/) describes how you can add the creation and modification time fields into your entities by using the auditing infrastructure of Spring Data JPA.
* [Spring Data JPA Tutorial: Auditing, Part Two](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-auditing-part-two/) describes how you can add the information of the authenticated user, who created and/or updated an entity, into your entities by using the auditing infrastructure of Spring Data JPA.
* [Spring Data JPA Tutorial: Adding Custom Methods to a Single Repository](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-eight-adding-functionality-to-a-repository/) describes how you can add custom methods to a single repository.
* [Spring Data JPA Tutorial: Adding Custom Methods to All Repositories](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-adding-custom-methods-into-all-repositories/) describes how you can add custom methods to all repositories.
* [Spring Data JPA Tutorial: Integration Testing](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-integration-testing/) describes how you can write integration tests for your Spring Data JPA repositories.

**My "Test With Spring" course helps you to write unit, integration, and end-to-end tests for Spring and Spring Boot Web Apps:**

[**CHECK IT OUT >>**](https://www.testwithspring.com/save-time-by-writing-less-test-code/?utm_source=petri-kainulainen&utm_medium=web&utm_content=spring-data-jpa-end&utm_campaign=test-with-spring-course-presales)

The following blog posts belong to my “old” Spring Data JPA tutorial and use Spring Data JPA 1.2.0:

* [Spring Data JPA Tutorial, Part Three: Custom Queries with Query Methods](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-three-custom-queries-with-query-methods/) describes how you can create custom queries by adding query methods to your repository interfaces.
* [Spring Data JPA Tutorial, Part Nine: Conclusions](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-nine-conclusions/) evaluates the usefulness of Spring Data JPA, gives recommendations about its usage and provides links to useful resources.
* [Spring Data JPA Tutorial: Using Querydsl in a Multi-module Maven Project](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-using-querydsl-in-a-multi-module-maven-project/) describes how you can use Querydsl and Spring Data JPA in a multi-module Maven project.

Congratulations. If you read every part of this tutorial, you know that implementing data access code does not have to be cumbersome. It can be fast, simple and fun.

It is time to put your new skills to use.

# Spring Data JPA Tutorial: Getting the Required Dependencies

Petri Kainulainen December 7, 2014[22 comments](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-getting-the-required-dependencies/#comments)

[spring data](https://www.petrikainulainen.net/tag/spring-data/), [spring data jpa](https://www.petrikainulainen.net/tag/spring-data-jpa/), [Spring Framework](https://www.petrikainulainen.net/tag/spring-framework/)

Before we can create an application that uses Spring Data JPA, we need to get the required dependencies.

This blog post identifies the required components and describes how we can get them by using Maven.

Let’s get started.

**Additional Reading:**

If you are not familiar with Spring Data JPA, **you should read** the following blog post before you continue reading this blog post:

* [Spring Data JPA Tutorial: Introduction](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-introduction/) provides a quick introduction to Spring Data JPA and gives an overview of the Spring Data repository interfaces.

## What Components Do We Need?

If we want to implement a persistence layer that uses Spring Data JPA, we need the following components:

* The **JDBC driver** provides a database specific implementation of the JDBC API. We use the [H2 in-memory database](http://www.h2database.com/html/main.html) because it makes our example application easier to run.
* The **datasource** provides database connections to our application. We use the [HikariCP datasource](http://brettwooldridge.github.io/HikariCP/" \t "_blank) because it is the fastest datasource on this planet.
* The **JPA Provider** implements the Java Persistence API. We use [Hibernate](http://hibernate.org/orm/) because it is the most common JPA provider.
* [Spring Data JPA](http://projects.spring.io/spring-data-jpa/) hides the used JPA provider behind its repository abstraction.

Let’s move on and find out how we can get the required dependencies with Maven.

**My "Test With Spring" course helps you to write unit, integration, and end-to-end tests for Spring and Spring Boot Web Apps:**

[**CHECK IT OUT >>**](https://www.testwithspring.com/save-time-by-writing-less-test-code/?utm_source=petri-kainulainen&utm_medium=web&utm_content=spring-data-jpa-start&utm_campaign=test-with-spring-course-presales)

## Getting the Required Dependencies with Maven

We can get the required dependencies with Maven by using one of these options:

1. We can manage our dependencies by using the [Spring IO Platform](http://spring.io/platform).
2. We can manage our dependencies “manually”.

Let’s take a look at both options.

### Using the Spring.IO Platform

If we use the Spring IO Platform, we need to follow these steps:

1. Enable the Spring IO Platform.
2. Configure the required dependencies in the pom.xml file.

**First**, we can enable the Spring IO Platform by adding the following XML to our POM file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | <dependencyManagement>      <dependencies>          <dependency>              <groupId>io.spring.platform</groupId>              <artifactId>platform-bom</artifactId>              <version>1.1.2.RELEASE</version>              <type>pom</type>              <scope>import</scope>          </dependency>      </dependencies>  </dependencyManagement> |

After we have enabled the Spring IO Platform, we don’t have to worry about dependency versions because the Spring IO Platform takes care of that. This means that we can get the required dependencies by adding the following XML to the dependencies section of our POM file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | <!-- Database (H2) -->  <dependency>      <groupId>com.h2database</groupId>      <artifactId>h2</artifactId>  </dependency>    <!-- DataSource (HikariCP) -->  <dependency>      <groupId>com.zaxxer</groupId>      <artifactId>HikariCP</artifactId>  </dependency>    <!-- JPA Provider (Hibernate) -->  <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-entitymanager</artifactId>  </dependency>    <!-- Spring Data JPA -->  <dependency>      <groupId>org.springframework.data</groupId>      <artifactId>spring-data-jpa</artifactId>  </dependency> |

Our example application has a few other dependencies as well. You can get the full list of dependencies by reading [its pom.xml file](https://github.com/pkainulainen/spring-data-jpa-examples/blob/master/query-methods/pom.xml).

**Additional Reading:**

* [The Spring IO Platform](http://platform.spring.io/platform/)
* [The Spring IO Platform Reference Guide](http://docs.spring.io/platform/docs/1.1.2.RELEASE/reference/htmlsingle/)

Let’s move on and find out how we can manage our dependencies manually.

### Managing Our Dependencies Manually

If we manage our dependencies “manually”, we need specify the version numbers of all dependencies. We can do this by adding the following dependency declarations to the dependencies section of our pom.xml file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | <!-- Database (H2) -->  <dependency>      <groupId>com.h2database</groupId>      <artifactId>h2</artifactId>      <version>1.4.185</version>  </dependency>    <!-- DataSource (HikariCP) -->  <dependency>      <groupId>com.zaxxer</groupId>      <artifactId>HikariCP</artifactId>      <version>2.2.5</version>  </dependency>    <!-- JPA Provider (Hibernate) -->  <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-entitymanager</artifactId>      <version>4.3.8.Final</version>  </dependency>    <!-- Spring Data JPA -->  <dependency>      <groupId>org.springframework.data</groupId>      <artifactId>spring-data-jpa</artifactId>      <version>1.7.2.RELEASE</version>  </dependency> |

I used the [dependency versions that are provided by the Spring IO platform](http://docs.spring.io/platform/docs/1.1.2.RELEASE/reference/htmlsingle/#appendix-dependency-versions). If you want to use newer versions, you can find the latest available versions from [mvnrepository.com](http://mvnrepository.com/).

I guess the obvious question is: which option should we use?

### What Is the Best Way to Manage Our Dependencies?

If we are starting a new project, we should use the Spring IO Platform because

* We don’t have to worry about the dependency versions. For example, we don’t have to worry about incompatibility issues because we know that our dependencies work together like a charm.
* We can always override the dependency versions provided the by the Spring.IO platform.

On the other hand, if we are adding Spring Data JPA to an existing project, it is often wiser to manage our dependencies manually because it requires less work.

Let’s move on and summarize what we learned from this blog post.

**My "Test With Spring" course helps you to write unit, integration, and end-to-end tests for Spring and Spring Boot Web Apps:**

[**CHECK IT OUT >>**](https://www.testwithspring.com/save-time-by-writing-less-test-code/?utm_source=petri-kainulainen&utm_medium=web&utm_content=spring-data-jpa-end&utm_campaign=test-with-spring-course-presales)

## Summary

This blog post has taught us four things:

* If we want to implement a persistence layer that uses Spring Data JPA, we need the following components: a JDBC driver, a datasource, a JPA provider, and the Spring Data JPA.
* We can get the required dependencies by using the Spring IO platform or managing our dependencies manually.
* If we are starting a new project, we should use the Spring IO platform because it ensures that our dependencies work together like a charm.
* If we are adding Spring Data JPA to an existing project, we should manage our dependencies manually because it requires less work.

The next part of this tutorial describes how we can [configure Spring Data JPA](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-one-configuration/).

After we have declared the required dependencies in our POM file, we have to configure the application context of our Spring application.

This blog post describes how we can configure the persistence layer of a Spring application that uses Spring Data JPA and Hibernate.

Let’s get started.

**Additional Reading:**

If you are not familiar with Spring Data JPA, **you should read** the following blog posts before you continue reading this blog post:

* [Spring Data JPA Tutorial: Introduction](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-introduction/) provides a quick introduction to Spring Data JPA and gives an overview of the Spring Data repository interfaces.
* [Spring Data JPA Tutorial: Getting the Required Dependencies](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-getting-the-required-dependencies/) describes how you can get the required dependencies.

## Configuring the Persistence Layer of a Spring Application

We can create the configuration class, which configures the persistence layer of a Spring application, by following these steps:

1. Create the properties file that contains the properties used by our application context configuration class.
2. Configure the datasource bean.
3. Configure the entity manager factory bean.
4. Configure the transaction manager bean.
5. Enable annotation-driven transaction management.
6. Configure Spring Data JPA.

**My "Test With Spring" course helps you to write unit, integration, and end-to-end tests for Spring and Spring Boot Web Apps:**

[**CHECK IT OUT >>**](https://www.testwithspring.com/save-time-by-writing-less-test-code/?utm_source=petri-kainulainen&utm_medium=web&utm_content=spring-data-jpa-start&utm_campaign=test-with-spring-course-presales)

But before we can get started, we have to create the configuration class that configures the persistence layer of our application. The source code of the PersistenceContext class looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Configuration  class PersistenceContext {        //Configure the required beans here  } |

Let’s start by creating the properties file.

### Creating the Properties File

Often we want to use a slightly different configuration in different environments. A good way to do this is move the configuration to a properties file and use a different properties file in different environments.

The application.properties file contains the configuration that is used to configure our example application. We can create this properties file by following these steps:

1. Configure the database connection of our application. We need to configure the name of the JDBC driver class, the JDBC url, the username of the database user, and the password of the database user.
2. Configure Hibernate by following these steps:
   1. Configure the used database dialect.
   2. Ensure that Hibernate creates the database when our application is started and drops it when our application is closed.
   3. Configure the naming strategy that is used when Hibernate creates new database objects and schema elements.
   4. Configure the Hibernate to NOT write the invoked SQL statements to the console.
   5. Ensure that if Hibernate writes the SQL statements to the console, it will use prettyprint.

The application.properties file looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | #Database Configuration  db.driver=org.h2.Driver  db.url=jdbc:h2:mem:datajpa  db.username=sa  db.password=    #Hibernate Configuration  hibernate.dialect=org.hibernate.dialect.H2Dialect  hibernate.hbm2ddl.auto=create-drop  hibernate.ejb.naming\_strategy=org.hibernate.cfg.ImprovedNamingStrategy  hibernate.show\_sql=false  hibernate.format\_sql=true |

The properties found from the application.properties file are loaded by the ExampleApplicationContext class. If you want to take a closer look at it, you can [get it from Github](https://github.com/pkainulainen/spring-data-jpa-examples/blob/master/query-methods/src/main/java/net/petrikainulainen/springdata/jpa/config/ExampleApplicationContext.java).

Let’s move on and configure the datasource bean.

### Configuring the Datasource Bean

We can configure the datasource bean by following these steps:

1. Ensure that the close() method of the created DataSource object is invoked when the application context is closed.
2. Configure the database connection. We need to set the name of the JDBC driver class, the JDBC url, the username of database user, and the password of the database user.
3. Create a new HikariDataSource object and return the created object.

The method that configures the datasource bean looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @Configuration  class PersistenceContext {        @Bean(destroyMethod = "close")      DataSource dataSource(Environment env) {          HikariConfig dataSourceConfig = new HikariConfig();          dataSourceConfig.setDriverClassName(env.getRequiredProperty("db.driver"));          dataSourceConfig.setJdbcUrl(env.getRequiredProperty("db.url"));          dataSourceConfig.setUsername(env.getRequiredProperty("db.username"));          dataSourceConfig.setPassword(env.getRequiredProperty("db.password"));            return new HikariDataSource(dataSourceConfig);      }        //Add the other beans here  } |

**Additional Reading:**

* [The Javadoc of the DataSource interface](https://docs.oracle.com/javase/7/docs/api/javax/sql/DataSource.html)
* [Spring Framework Reference Manual: 14.3.1 DataSource](http://docs.spring.io/spring/docs/4.1.x/spring-framework-reference/htmlsingle/#jdbc-datasource)
* [Hikari CP Initialization](https://github.com/brettwooldridge/HikariCP#initialization)
* [HikariCP configuration properties](https://github.com/brettwooldridge/HikariCP#configuration-knobs-baby)

Let’s move on and configure the entity manager factory bean.

### Configuring the Entity Manager Factory Bean

We can configure the entity manager factory bean by following these steps:

1. Create a new LocalContainerEntityManagerFactoryBean object. We need to create this object because it creates the JPA EntityManagerFactory.
2. Configure the used datasource.
3. Configure the Hibernate specific implementation of the JpaVendorAdapter interface. It will initialize our configuration with the default settings that are compatible with Hibernate.
4. Configure the packages that are scanned for entity classes.
5. Configure the JPA properties that are used to provide additional configuration to the used JPA provider.

The method that configures the entity manager factory bean looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | @Configuration  class PersistenceContext {        @Bean      LocalContainerEntityManagerFactoryBean entityManagerFactory(DataSource dataSource,                                                                  Environment env) {          LocalContainerEntityManagerFactoryBean entityManagerFactoryBean = new LocalContainerEntityManagerFactoryBean();          entityManagerFactoryBean.setDataSource(dataSource);          entityManagerFactoryBean.setJpaVendorAdapter(new HibernateJpaVendorAdapter());          entityManagerFactoryBean.setPackagesToScan("net.petrikainulainen.springdata.jpa.todo");            Properties jpaProperties = new Properties();            //Configures the used database dialect. This allows Hibernate to create SQL          //that is optimized for the used database.          jpaProperties.put("hibernate.dialect", env.getRequiredProperty("hibernate.dialect"));            //Specifies the action that is invoked to the database when the Hibernate          //SessionFactory is created or closed.          jpaProperties.put("hibernate.hbm2ddl.auto",                  env.getRequiredProperty("hibernate.hbm2ddl.auto")          );            //Configures the naming strategy that is used when Hibernate creates          //new database objects and schema elements          jpaProperties.put("hibernate.ejb.naming\_strategy",                  env.getRequiredProperty("hibernate.ejb.naming\_strategy")          );            //If the value of this property is true, Hibernate writes all SQL          //statements to the console.          jpaProperties.put("hibernate.show\_sql",                  env.getRequiredProperty("hibernate.show\_sql")          );            //If the value of this property is true, Hibernate will format the SQL          //that is written to the console.          jpaProperties.put("hibernate.format\_sql",                  env.getRequiredProperty("hibernate.format\_sql")          );            entityManagerFactoryBean.setJpaProperties(jpaProperties);            return entityManagerFactoryBean;      }        //Add the other beans here  } |

**Additional Reading:**

* [The Javadoc of the EntityManagerFactory interface](http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManagerFactory.html)
* [The Javadoc of the LocalContainerEntityManagerFactoryBean class](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/orm/jpa/LocalContainerEntityManagerFactoryBean.html)
* [The Javadoc of the JpaVendorAdapter interface](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/orm/jpa/JpaVendorAdapter.html)
* [The Javadoc of the HibernateJpaVendorAdapter class](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/orm/jpa/vendor/HibernateJpaVendorAdapter.html)
* [Spring Framework Reference Manual: 15.5.1 Three options for JPA setup in a Spring environment](http://docs.spring.io/spring/docs/4.1.x/spring-framework-reference/htmlsingle/#orm-jpa-setup)
* [Hibernate ORM Reference Manual: 3.4. Optional configuration properties](http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html_single/#configuration-optional)

Let’s move on and configure the transaction manager bean.

### Configuring the Transaction Manager Bean

Because we are using JPA, we have to create a transaction manager bean that integrates the JPA provider with the Spring transaction mechanism. We can do this by using the JpaTransactionManagerclass as the transaction manager of our application.

We can configure the transaction manager bean by following these steps:

1. Create a new JpaTransactionManager object.
2. Configure the entity manager factory whose transactions are managed by the created JpaTransactionManager object.

The method that configures the transaction manager bean looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | @Configuration  class PersistenceContext {        @Bean      JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {          JpaTransactionManager transactionManager = new JpaTransactionManager();          transactionManager.setEntityManagerFactory(entityManagerFactory);          return transactionManager;      }        //Add the other beans here  } |

**Additional Reading:**

* [The Javadoc of the JpaTransactionManager class](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/orm/jpa/JpaTransactionManager.html)
* [Spring Framework Reference Manual: 15.5.3 Transaction Management](http://docs.spring.io/spring/docs/4.1.x/spring-framework-reference/htmlsingle/#orm-jpa-tx)

Let’s move on and enable annotation-driven transaction management.

### Enabling Annotation-Driven Transaction Management

We can enable annotation-driven transaction management by annotating the PersistenceContext class with the @EnableTransactionManagement annotation. The relevant part of the PersistenceContextclass looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Configuration  @EnableTransactionManagement  class PersistenceContext {        //The beans are configured here  } |

**Additional Reading:**

* [The Javadoc of the @EnableTransactionManagement annotation](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html)
* [Spring Framework Reference Manual: 12.5.6 Using @Transactional](http://docs.spring.io/spring/docs/4.1.x/spring-framework-reference/htmlsingle/#transaction-declarative-annotations)

Let’s move on and configure Spring Data JPA.

### Configuring Spring Data JPA

We can configure Spring Data JPA by following these steps:

1. Enable Spring Data JPA by annotating the PersistenceContext class with the @EnableJpaRepositories annotation.
2. Configure the base packages that are scanned when Spring Data JPA creates implementations for our repository interfaces.

The relevant part of the PersistenceContext class looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Configuration  @EnableJpaRepositories(basePackages = {          "net.petrikainulainen.springdata.jpa.todo"  })  @EnableTransactionManagement  class PersistenceContext {        //The beans are configured here  } |

**Additional Reading:**

* [The Javadoc of the @EnableJpaRepositories annotation](http://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/config/EnableJpaRepositories.html)
* [Spring Data JPA Reference Manual: 3.5. Creating repository instances](http://docs.spring.io/spring-data/jpa/docs/1.8.x/reference/html/#repositories.create-instances)

That is all folks. We have now successfully configured the persistence layer of our example application. Let’s move on and summarize what we learned from this blog post.

# Spring Data JPA Tutorial: CRUD

Petri Kainulainen January 24, 2012[183 comments](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-two-crud/#comments)

[spring data](https://www.petrikainulainen.net/tag/spring-data/), [spring data jpa](https://www.petrikainulainen.net/tag/spring-data-jpa/), [Spring Framework](https://www.petrikainulainen.net/tag/spring-framework/)

We have now [configured the persistence layer of our Spring application](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-one-configuration/). We are finally ready to create our first Spring Data JPA repository.

This blog post describes how we can create a repository that provides CRUD operations for todo entries.

Let’s get started.

**Additional Reading:**

If you are not familiar with Spring Data JPA, **you should read** the following blog posts before you continue reading this blog post:

* [Spring Data JPA Tutorial: Introduction](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-introduction/) provides a quick introduction to Spring Data JPA and gives an overview of the Spring Data repository interfaces.
* [Spring Data JPA Tutorial: Getting the Required Dependencies](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-getting-the-required-dependencies/) describes how you can get the required dependencies.
* [Spring Data JPA Tutorial: Configuration](https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-part-one-configuration/) describes how you can configure the persistence layer of a Spring application that uses Spring Data JPA.

## Creating the Repository

Before we can create our first Spring Data JPA repository, we have to create an entity class that contains the information of a single todo entry. The relevant part of the Todo class looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | import org.hibernate.annotations.Type;    import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.PrePersist;  import javax.persistence.Table;  import javax.persistence.Version;  import java.time.ZonedDateTime;    @Entity  @Table(name = "todos")  final class Todo {        @Id      @GeneratedValue(strategy = GenerationType.AUTO)      private Long id;        @Column(name = "creation\_time", nullable = false)      @Type(type = "org.jadira.usertype.dateandtime.threeten.PersistentZonedDateTime")      private ZonedDateTime creationTime;        @Column(name = "description", length = 500)      private String description;        @Column(name = "modification\_time")      @Type(type = "org.jadira.usertype.dateandtime.threeten.PersistentZonedDateTime")      private ZonedDateTime modificationTime;        @Column(name = "title", nullable = false, length = 100)      private String title;        @Version      private long version;        //The constructor, builder, and other methods are omitted  } |

**Additional Reading:**

* [Hibernate Reference Documentation: 6.4 Custom Types](http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html_single/#types-custom)

We are now ready to create our first Spring Data JPA repository. We can create the repository that provides CRUD operations for Todo objects by using one of the following methods:

1. Create an interface that extends the [CrudRepository](http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/CrudRepository.html" \t "_blank) interface.
2. Create an interface that extends the [Repository](http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/Repository.html) interface and add the required methods to the created interface.

Let’s take a closer look at these methods.

**My "Test With Spring" course helps you to write unit, integration, and end-to-end tests for Spring and Spring Boot Web Apps:**

[**CHECK IT OUT >>**](https://www.testwithspring.com/save-time-by-writing-less-test-code/?utm_source=petri-kainulainen&utm_medium=web&utm_content=spring-data-jpa-start&utm_campaign=test-with-spring-course-presales)

### Extending the CrudRepository Interface

If we create our repository by extending the CrudRepository interface, we have to provide two type parameters:

1. The type of the entity that is managed by our repository.
2. The type of the entity’s id field.

In other words, when we create the repository that provides CRUD operations for Todo objects, we have to provide the following type parameters:

1. The type of the entity is Todo.
2. The type of the entity’s id field is Long.

The source code of the TodoRepository interface looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5 | import org.springframework.data.repository.CrudRepository;    interface TodoRepository extends CrudRepository<Todo, Long> {    } |

**Additional Reading:**

* [The Javadoc of the CrudRepository interface](http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/CrudRepository.html)
* [Spring Data JPA Reference Manual: 3.3 Defining Repository Interfaces](http://docs.spring.io/spring-data/jpa/docs/1.8.x/reference/html/#repositories.definition)

The CrudRepository interface declares many methods, but the methods that are relevant for this blog post are described in the following:

* [The void delete(T entity) method](http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#delete-ID-) deletes the entity whose id is given as a method parameter.
* [The Iterable<T> findAll() method](http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#findAll--) returns all entities that are saved to the database.
* [The T findOne(Long id) method](http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#findOne-ID-) returns the entity whose id is given as method parameter. If no entity is found, this method returns null.
* [The T save(T entity) method](http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/CrudRepository.html#save-S-) saves the entity given as a method parameter and returns the persisted entity.

The [service class](https://github.com/pkainulainen/spring-data-jpa-examples/blob/master/query-methods/src/main/java/net/petrikainulainen/springdata/jpa/todo/RepositoryTodoService.java) that provides CRUD operations for todo entries uses these methods for fulfilling its responsibilities.

Let’s find out how we can create a repository interface that extends the Repository interface.

### Extending the Repository Interface

If we create our repository by extending the Repository interface, we have to follow these steps:

1. Provide two type parameters:
   1. The type of the managed entity (Todo).
   2. The type of the entity’s id field (Long).
2. Add the required methods to the repository interface:
   1. The void delete(Todo deleted) method deletes the Todo object given as a method parameter.
   2. The List<Todo> findAll() method returns all Todo objects that are found from the database.
   3. The Optional<Todo> findOne(Long id) method finds the todo entry whose id is given as a method parameter. If no todo entry is found, this method returns an empty Optional.
   4. The Todo save(Todo persisted) method saves the Todo object given as a method parameter and returns the persisted object.

The source code of the TodoRepository interface looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import org.springframework.data.repository.Repository;    import java.util.List;  import java.util.Optional;    interface TodoRepository extends Repository<Todo, Long> {        void delete(Todo deleted);        List<Todo> findAll();        Optional<Todo> findOne(Long id);        Todo save(Todo persisted);  } |

If we don’t want to return Optional (Guava / Java 8) objects, we can also use the “traditional” Todo findOne(Long id) method.

**Additional Reading:**

* [The Javadoc of the Repository interface](http://docs.spring.io/spring-data/data-commons/docs/current/api/org/springframework/data/repository/Repository.html)
* [Spring Data JPA Reference Manual: 3.3.1 Fine-tuning repository definition](http://docs.spring.io/spring-data/jpa/docs/1.8.x/reference/html/#repositories.definition-tuning)
* [What’s New in Spring Data Dijkstra](https://spring.io/blog/2014/05/21/what-s-new-in-spring-data-dijkstra) (search for ‘support for wrapper types as return values’)

Let’s move on and find out which method we should use.

## Which Method Should We Use?

It depends.

I know that this is probably the most annoying answer one can give to a question. That is why I created two rules that we can follow when we are creating Spring Data JPA repositories. These rules are:

* If we want to expose all repository methods that are declared by the CrudRepository interface **AND**we don’t want to return Optional (Guava / Java 8) objects, our repository interfaces should extend the CrudRepository interface.
* If we don’t want to expose all repository methods that are declared by the CrudRepository interface **OR** we want to return Optional (Guava / Java 8) objects, our repository interfaces must extend the Repository interface.

Case closed?

Not exactly. I argue that **we should always use the second method**. This opinion is based on two reasons:

* When we create an interface, we should not add unnecessary methods to it. We should keep the interface as small as possible because small interfaces are easier to use and they help us to create components that have only one job.
* Optional helps us to create better APIs because it reveals that there might not be a return value.

**Additional Reading:**

* [Minimal Interface](http://martinfowler.com/bliki/MinimalInterface.html)
* [Intention Revealing Code With Java 8’s New Type ‘Optional’](http://blog.codefx.org/techniques/intention-revealing-code-java-8-optional/)
* [Tired of Null Pointer Exceptions? Consider Using Java SE 8’s Optional](http://www.oracle.com/technetwork/articles/java/java8-optional-2175753.html)
* [The Design of Optional](http://blog.codefx.org/jdk/dev/design-optional/)
* [Java 8 Optional Objects](http://web.archive.org/web/20160323005315/http:/blog.informatech.cr:80/2013/04/10/java-optional-objects/)

If we create our repositories by extending the Repository interface and adding the required methods to the created repository interfaces, we need to add the “same” methods to every interface. Right?

**Wrong**.

We can avoid this by following these steps:

1. Create a base interface that extends the Repository interface and add the common methods to that interface.
2. Create the actual repository interface that extends our base interface.

Let’s move on and take a closer look at these steps.

**First**, we have to create a base interface that declares the methods shared by our repositories. We can do this by following these steps:

1. Create the BaseRepository interface that extends the Repository interface. This interface has two type parameters:
   1. T describes the type of the managed entity.
   2. ID describes the type of the entity’s id field.
2. Annotate the created interface with the @NoRepositoryBean annotation. This ensures that Spring Data JPA doesn’t try to create an implementation for our base repository interface.
3. Add the common methods to the created interface.

The source code of the BaseRepository interface looks as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | import org.springframework.data.repository.NoRepositoryBean;  import org.springframework.data.repository.Repository;    import java.util.List;  import java.util.Optional;    @NoRepositoryBean  interface BaseRepository<T, ID extends Serializable> extends Repository<T, ID> {        void delete(T deleted);        List<T> findAll();        Optional<T> findOne(ID id);        T save(T persisted);  } |

**Additional Reading:**

* [The Javadoc of the @NoRepositoryBean annotation](http://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/NoRepositoryBean.html)

**Second**, we have to create the actual repository interface that extends our base interface. We can do this by following these steps:

1. Create the TodoRepository interface.
2. Extend the BaseRepository interface and provide two type parameters:
   1. The type of the managed entity is Todo.
   2. The type of the entity’s id field is Long.

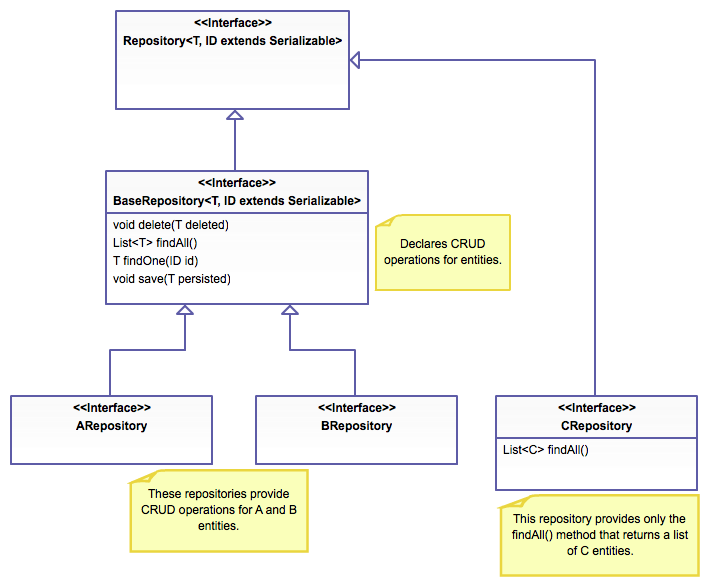
The source code of the TodoRepository interface looks as follows:

|  |  |
| --- | --- |
| 1  2  3 | interface TodoRepository extends BaseRepository<Todo, Long> {    } |

We have now created a repository hierarchy that allows us to:

* Create repositories that provides CRUD operations for entities without declaring the “same” methods in every repository interface.
* Create repositories that do not provide all CRUD operations. For example, we can create a repository that provides only the findAll() method.

The following figure illustrates the benefits of this solution:



Let’s move on and summarize what we learned from this blog post.

**My "Test With Spring" course helps you to write unit, integration, and end-to-end tests for Spring and Spring Boot Web Apps:**

[**CHECK IT OUT >>**](https://www.testwithspring.com/save-time-by-writing-less-test-code/?utm_source=petri-kainulainen&utm_medium=web&utm_content=spring-data-jpa-end&utm_campaign=test-with-spring-course-presales)

## Summary

This blog post has taught us three things:

* We can create repository interfaces by extending either the CrudRepository or the Repositoryinterface.
* We should create our repositories by extending the Repository interface and adding the required methods to the created repository interface.
* If our application has more than one repository, we should create a base repository interface that declares the methods that are shared by our “concrete” repositories.