**Instructor Notes:**

Add instructor notes here.

**Instructor Notes:**

Add instructor notes here.

## Lesson Objectives

- After completing this lesson, participants will be able to understand:
- Spring support for JPA
- Implementing Spring JPA integration
- Spring Data JPA

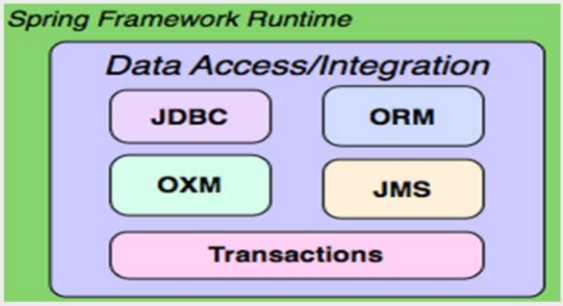
Capgemini

**Instructor Notes:**

Add instructor notes here.

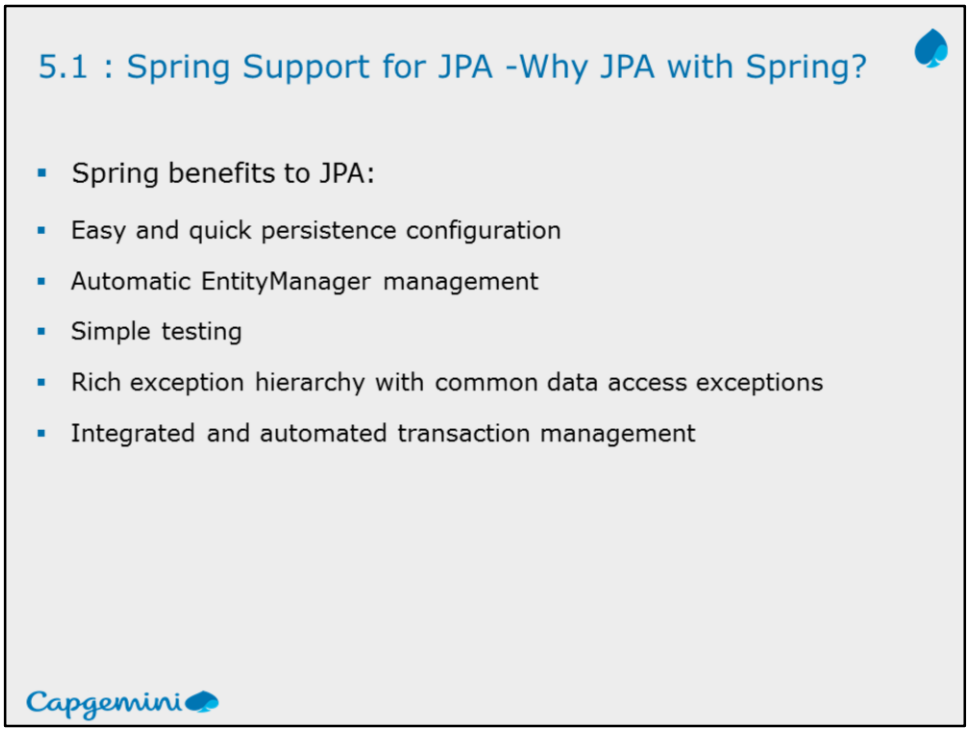
## 5.1 : Spring Support for JPA -Spring JPA Integration : Overview

- The Spring Framework supports integration with Hibernate, Java Persistence API (JPA) for
  - resource management,
  - data access object (DAO) implementations, and
  - transaction strategies




Capgemini

**Instructor Notes:**




**Why Spring JPA Integration?**

The client application that accesses the database using JPA has to depend on the JPA APIs like PersistenceContext, EntityManagerFactory, EntityManager and Transaction. These objects will continue to get scattered across the code throughout the Application.

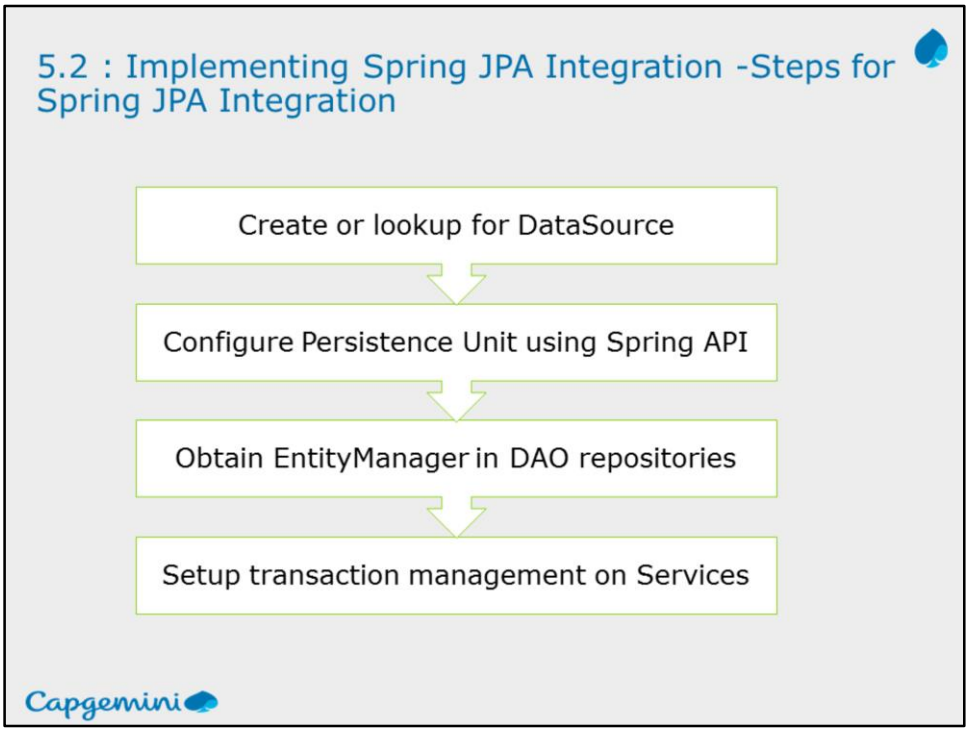
Moreover, the Application code has to **manually** maintain and manage these objects. In the case of Spring, the business objects can be highly configurable with the help of IOC Container. Which means that now it is possible to use the **JPA objects as Spring Beans** and they can enjoy all the facilities that Spring provides.

The EntityManager instance is managed automatically, and can be injected into data access object (DAO) beans; so there is no need to manage it manually in application code.

Another advantage of integrating JPA in Spring is that Spring manages starting, committing, and rolling back transactions automatically on your behalf. You will never use it directly and you'll only configure an implementation.

The JPA defines a modest exception hierarchy starting at PersistenceException, but even it is missing some key features like an exception to indicate that a unique key violation occurred. Spring Framework solve this problem by defining a thorough hierarchy of persistence exceptions.

**Instructor Notes:**




**Integration Steps:**

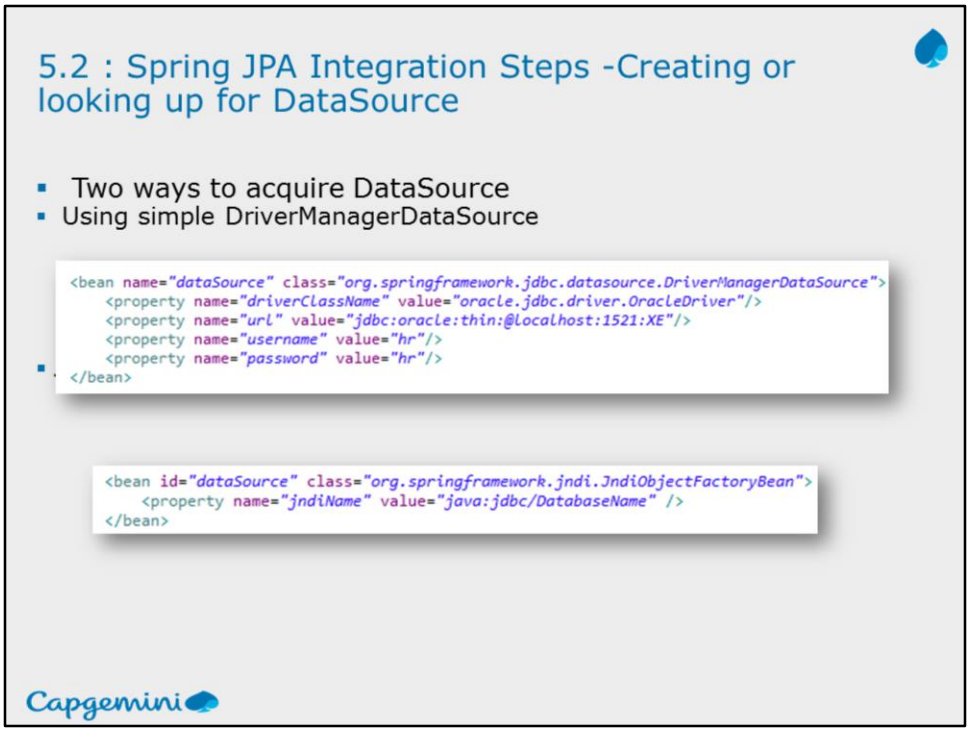
Integration work starts by defining or obtaining data source, which contains information about database url, username and password etc.

Once we obtain datasource, we need configure JPA related beans using spring configuration file. It is used in place of persistence.xml.
Though, we are using Spring configuration, optionally we may keep persistence.xml just to hold the persistence unit reference. This file may be required in future for advance configuration like caching etc.

After configuration, we can work on entities and DAO repositories to define our database operations.

Finally, we need to instruct Spring to handle all JPA transactions, this is done by annotating and marking DAO operations with various transaction demarcation policies.

**Instructor Notes:**



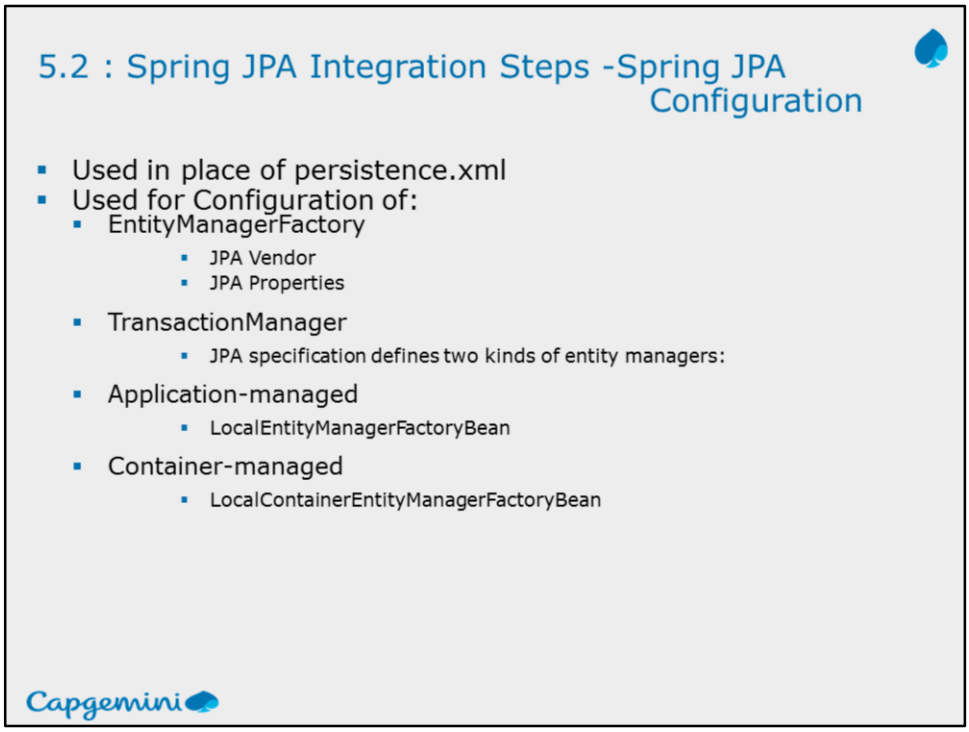

The first step towards integration is making DataSource available to application. There are different ways to do this. For example, if you need to simply test something quickly, use Spring's DriverManagerDataSource to create a DataSource on demand.

However, this creates a simple DataSource that returns single use Connections. Because it does not provide connection pooling, it really should never be used in a production environment. For production environment, we can define DataSource on server and look that DataSource up application.

Slide demonstrates both ways to acquire DataSource for our application.

**Note**: To keep configuration simple, we are using DriverManagerDataSource throughout integration.

**Instructor Notes:**

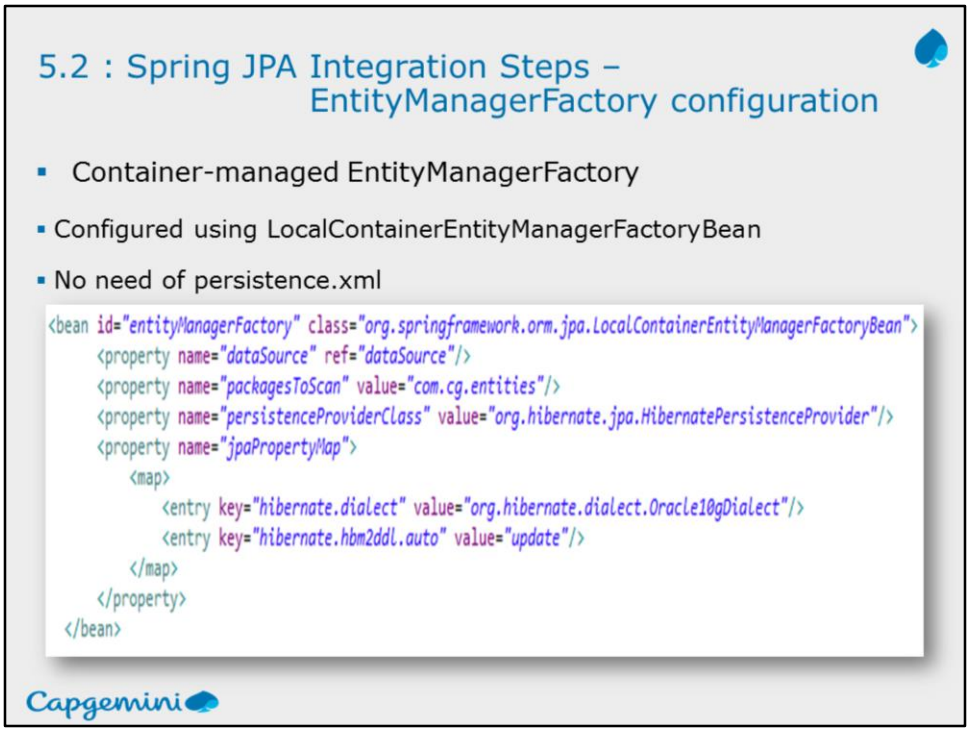



The EntityManagerFactory is what we use to start up JPA inside our application. Without Spring, we use persistence.xml file to configure this object. While working with Spring, we don't need configuration via persistence.xml, instead we can define all JPA related objects as spring beans.
Spring allows two ways to define this object, either application managed or container managed (used throughout this integration).

In addition to EntityManagerFactory, Spring provides TransactionManager to handle all JPA related transactions, which is also configured inside spring configuration file.

Let us see in subsequent slides, how to bootstrap this configuration file.

**Instructor Notes:**




Spring offers three different options to configure EntityManagerFactory in a project:

1. LocalEntityManagerFactoryBean
2. EntityManagerFactory lookup over JNDI
3. LocalContainerEntityManagerFactoryBean

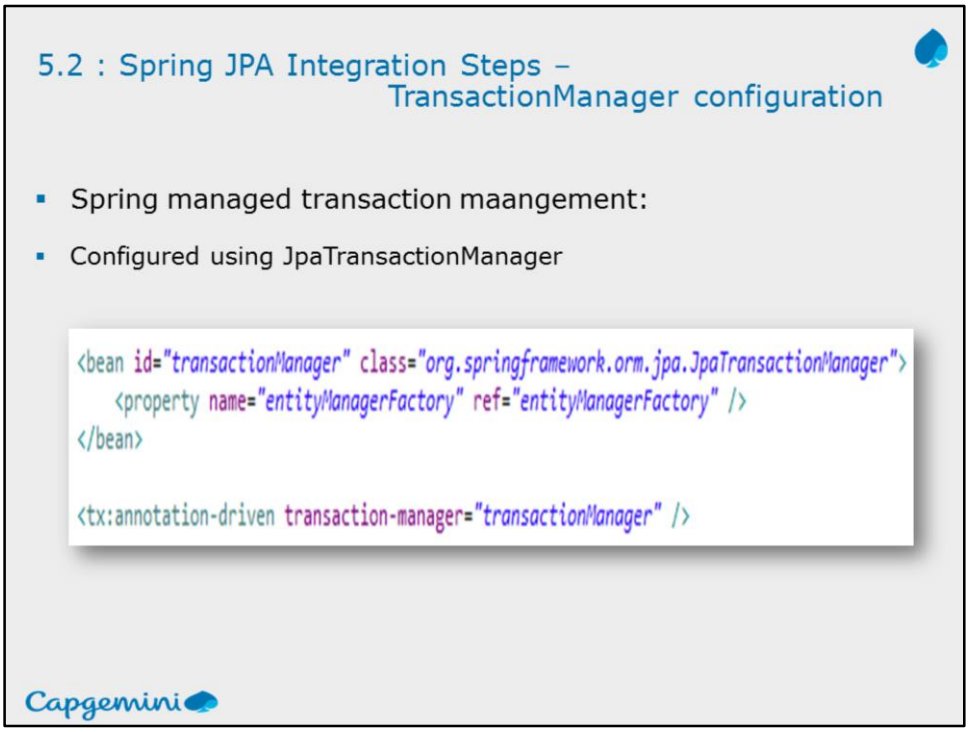
**LocalEntityManagerFactoryBean** is the most basic and limited one. It is mainly used for testing purposes and standalone environments. It reads JPA configuration from /META-INF/persistence .xml, doesn't allow you to use a Spring-managed DataSource instance, and doesn't support distributed transaction management.

Use **EntityManagerFactory** lookup over JNDI if the run time is Java EE 5 Server.

**LocalContainerEntityManagerFactoryBean** is the most powerful and flexible JPA configuration approach Spring offers. It gives full control overEntityManagerFactory configuration, and it's suitable for environments where fine-grained control is required. It enables you to work with a Spring-managedDataSource, lets you selectively load entity classes in your project's classpath, and so on. It works both in application servers and standalone environments.

Above slide demonstrates on how to configure LocalContainerEntityManagerFactoryBean in spring XML configuration file.

**Instructor Notes:**



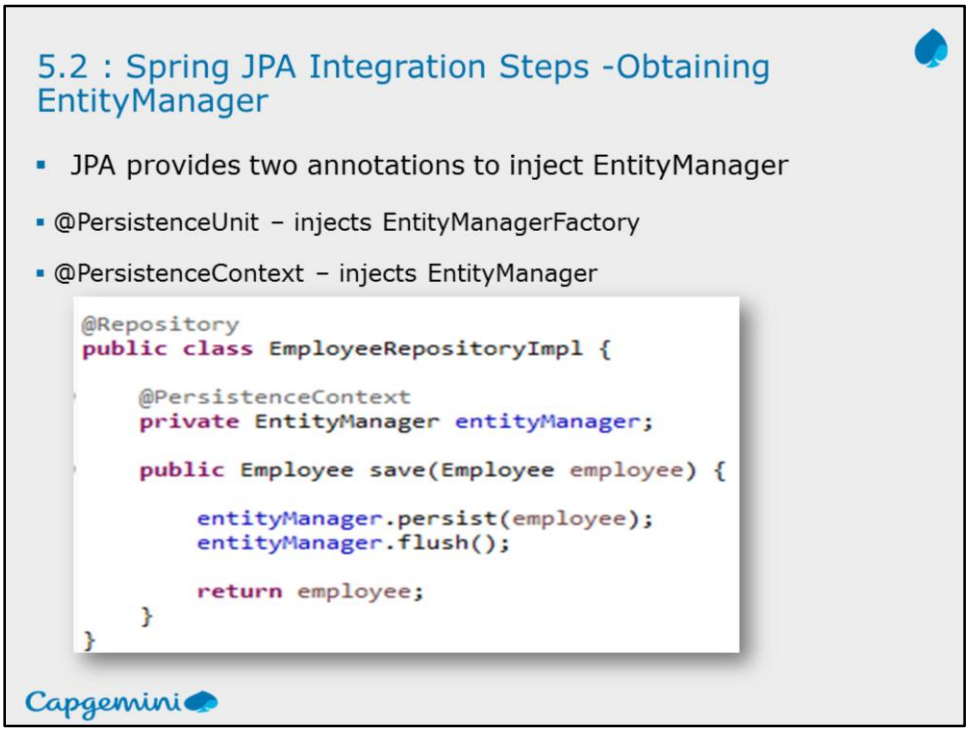

**Configuration of TransactionManager:**

Spring provides a class *org.springframework.orm.jpa.JpaTransactionManager* to work with JPA specific transactions. Slide demonstrate how to configure this class in spring configuration file. It is required to inject entitymanagerfactory created in earlier step in this class using setter injection.

Basically all of JPA interactions will be wrapped in transaction by this TransactionManager and we are instructing spring to create a bean of this class for application transaction management.

Once this transaction manager is made available, instead of opening and closing transactions manually, we can instruct Spring to handle transaction using annotations. To do so, we must use **<tx:annotation-driven/>** injecting the transaction manager instance.

**Instructor Notes:**

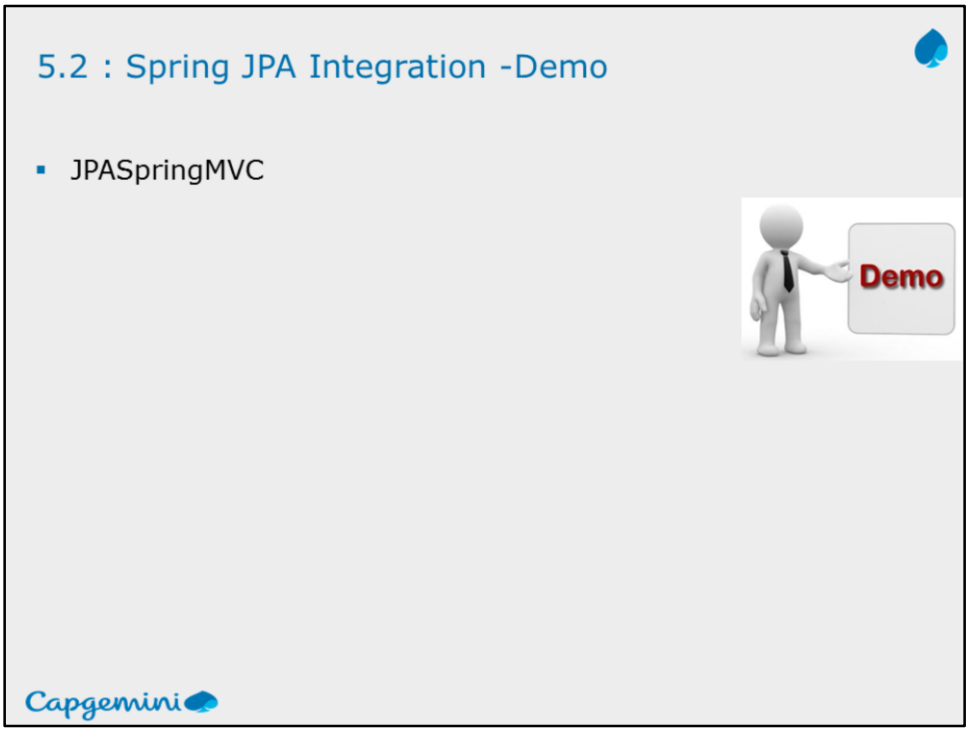



You can obtain EntityManagerFactory or EntityManager from a container via dependency injection, and you can automatically participate in the current transaction.

**Note:** You need to configure the PersistenceAnnotationBeanPostProcessor of Spring in order for annotations given in the slide to be processed. You can either define it as bean, or enable a context namespace and add <context:annotation-config/> or <context:component-scan/> elements into your XML bean configuration file.

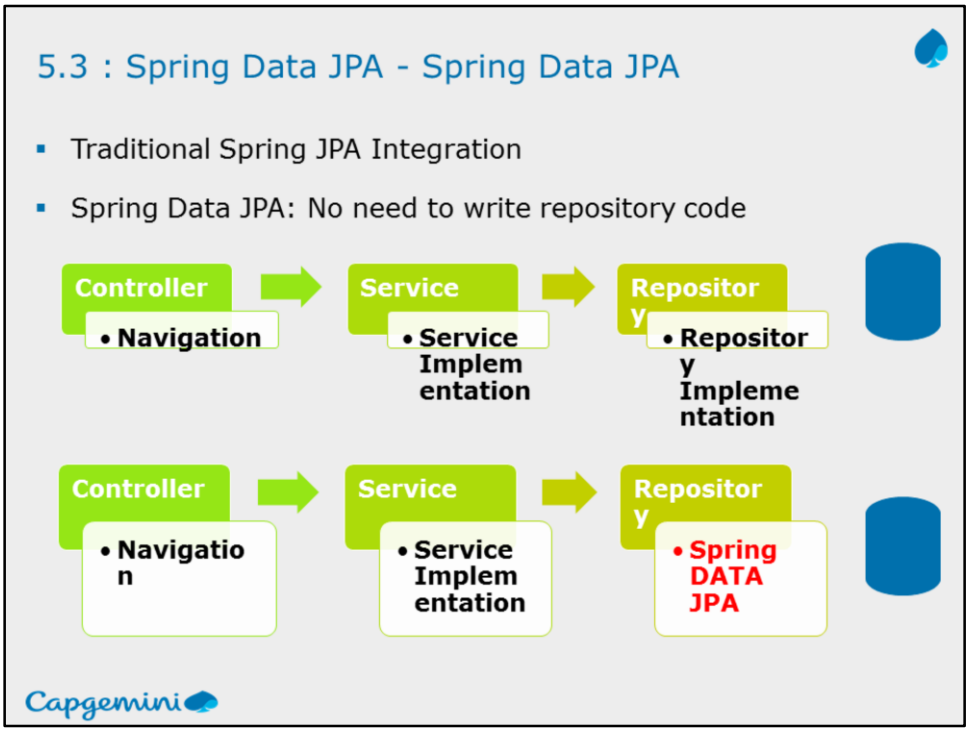
**Instructor Notes:**

Add instructor notes here.




Add the notes here.

**Instructor Notes:**




**What is Spring Data JPA?**

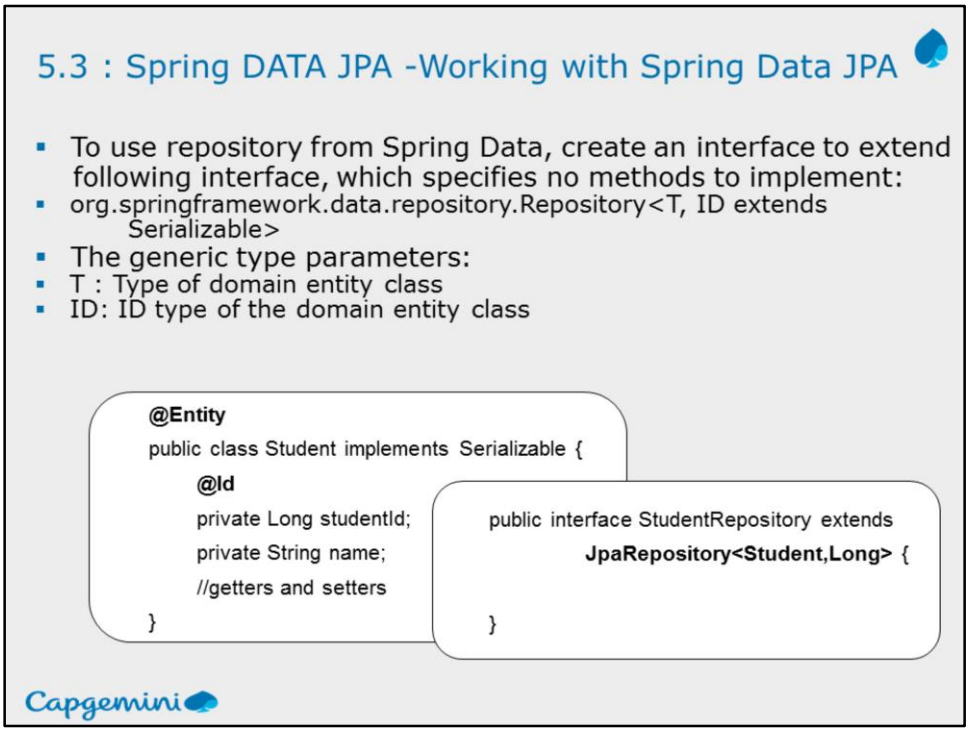
**Spring Data** supports a variety of data access methodologies, including JPA, JdbcTemplate, NoSQL, and more. Its primary subproject, Spring Data Commons, provides a core toolset that all other subprojects use to create repositories. The **Spring Data JPA subproject** provides support for repositories implemented against the Java Persistence API.

Spring Data JPA, a separate Spring project but dependent on Spring Framework, can write your repositories on behalf of developers.

Spring Data JPA is another layer of abstraction for the support of persistence layer in spring context. The goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores. We just have to write repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

**Note:** To configure Spring Data using XML required a new Spring Data JPA namespace, http://www.springframework.org/schema/data/jpa and to enable auto scanning of application repositories, one need to specify <jpa:repositories> element in spring configuration.

**Instructor Notes:**




As a first step we define a domain class-specific repository interface. The interface must extend JpaRepository and be typed to the domain class and an ID type. JpaRepository is a child inteface of following:

1. CrudRepository<T,ID>,
2. PagingAndSortingRepository<T,ID>,
3. QueryByExampleExecutor<T>,
4. Repository<T,ID>

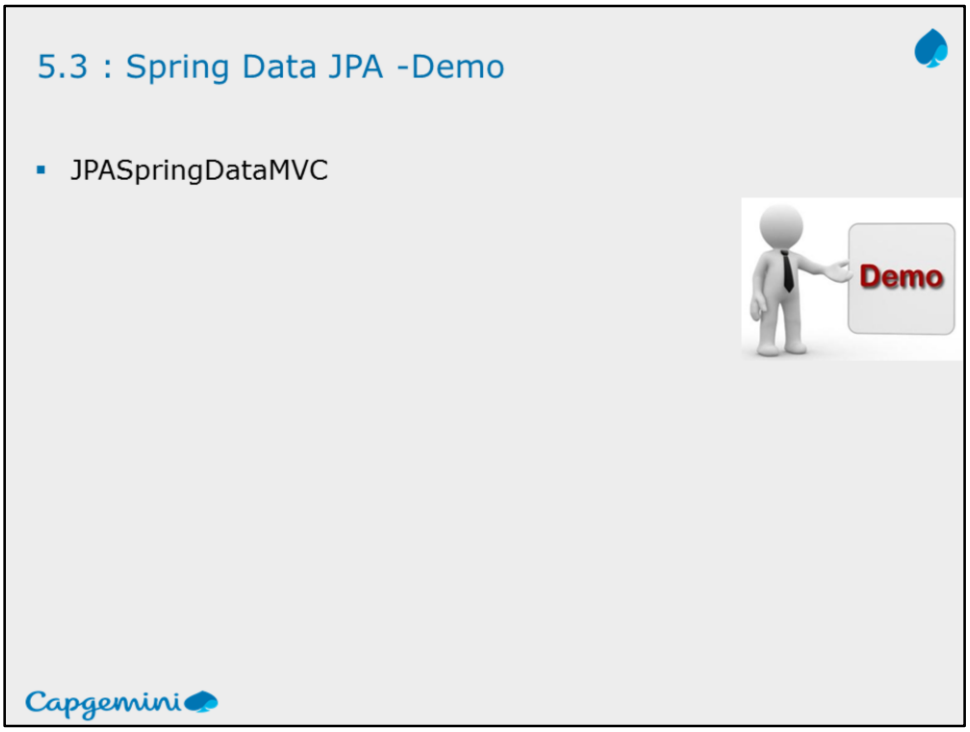
The CrudRepository provides sophisticated CRUD functionality for the entity class that is being managed. One can add additional query methods to this repository interface. Below listed are few methods of CrudRepository:

1. **count()** returns a long representing the total number of unfiltered entities extending T.
2. **delete(T)** and **delete(ID)** delete the single, specified entity and **deleteAll()** deletes every entity of that type.
3. **exists(ID)** returns a boolean indicating whether the entity of this type with the given key exists.
4. **findAll()** returns all entities of type T, whereas **findOne(ID)** retrieves a single entity of type T given its key.
5. **save(T)** saves the given entity (insert or update).

**Note**: One can define additional query methods using @Query. Please see the demo for more details.

**Instructor Notes:**

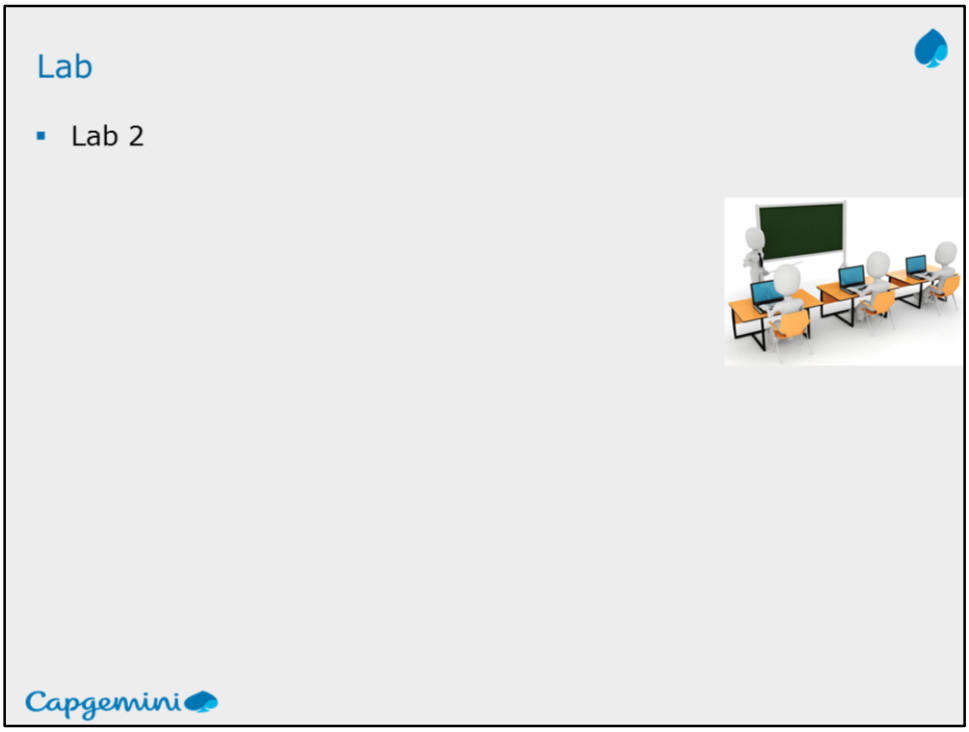
Add instructor notes here.




Add the notes here.

**Instructor Notes:**

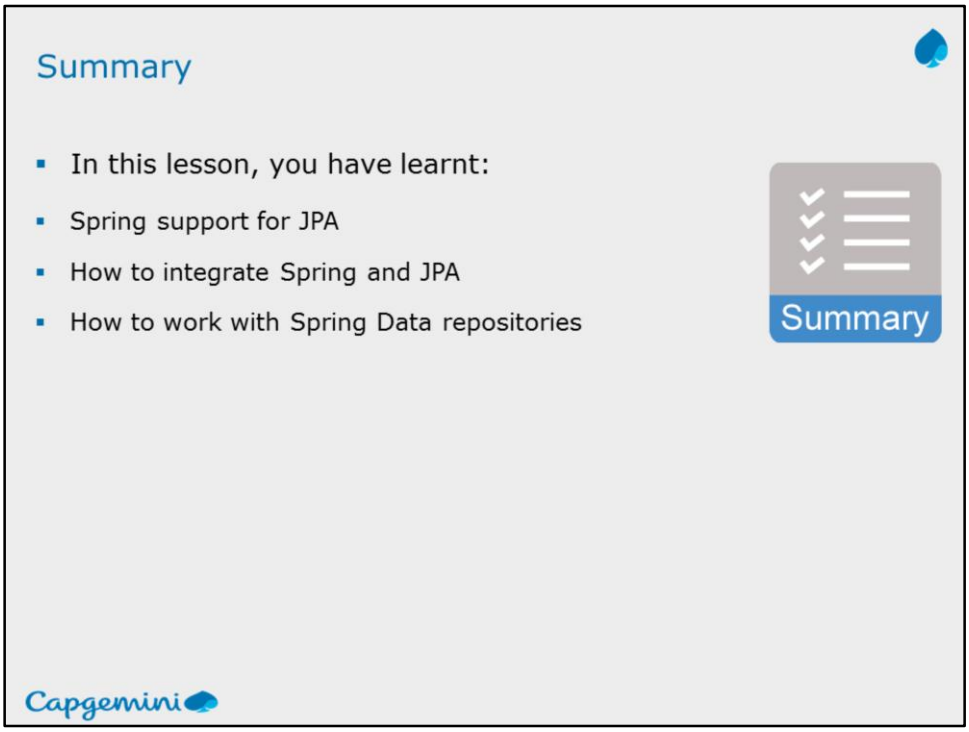
Add instructor notes here.




Add the notes here.

**Instructor Notes:**

Add instructor notes here.




Add the notes here.

**Instructor Notes:**

1. Option 1
2. True

## Review Question

- Question 1 Which one of the following is valid type of entity manager in JPA?
  - Option 1: Container managed
  - Option 2: JVM Managed
  - Option 3: Server Managed
- Question 2 LocalEntityManagerFactoryBean doesn't allow you to use a Spring managed DataSource instance.
  - True/False

Capgemini

Add the notes here.