JAVA 2:

LESSON 14

Spring Data JPA

Spring Data JPA provides repository support for the Java Persistence API (JPA). It eases development of applications that need to access JPA data sources.

SPRING DATA REPOSITORIES

- The goal of the Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.

CORE CONCEPTS

- The central interface in the Spring Data repository abstraction is **Repository**.
- It takes the domain class to manage as well as the ID type of the domain class as type arguments. This interface acts primarily as a marker interface to capture the types to work with and to help you to discover interfaces that extend this one.
- The **CrudRepository** provides sophisticated CRUD functionality for the entity class that is being managed.

```
public interface CrudRepository<T, ID extends Serializable>
  extends Repository<T, ID> {

  <S extends T> S save(S entity);      (1)

  Optional<T> findById(ID primaryKey); (2)

  Iterable<T> findAll();               (3)

  long count();                        (4)

  void delete(T entity);               (5)

  boolean existsById(ID primaryKey);   (6)

  // … more functionality omitted.
}
```

1. Saves the given entity.
2. Returns the entity identified by the given ID.
3. Returns all entities.
4. Returns the number of entities.
5. Deletes the given entity.
6. Indicates whether an entity with the given ID exists.

TEXT

- On top of the **CrudRepository**, there is a **PagingAndSortingRepository** abstraction that adds additional methods to ease paginated access to entities:

JAVA

```java
public interface PagingAndSortingRepository<T, ID extends Serializable>
  extends CrudRepository<T, ID> {

  Iterable<T> findAll(Sort sort);

  Page<T> findAll(Pageable pageable);
}
```

To access the second page of User by a page size of 20, you could do something like the following:

```java
PagingAndSortingRepository<User, Long> repository = // … get access to a bean
Page<User> users = repository.findAll(new PageRequest(1, 20));
```

In addition to query methods, query derivation for both count and delete queries is available. The following list shows the interface definition for a derived count query:

```java
interface UserRepository extends CrudRepository<User, Long> {

  long countByLastname(String lastname);
}
```

The following list shows the interface definition for a derived delete query:

```java
interface UserRepository extends CrudRepository<User, Long> {

  long deleteByLastname(String lastname);

  List<User> removeByLastname(String lastname);
}
```

QUERY METHODS

- Standard CRUD functionality repositories usually have queries on the underlying datastore. With Spring Data, declaring those queries becomes a four-step process:
- 1. Declare an interface extending Repository or one of its subinterfaces and type it to the domain class and ID type that it should handle, as shown in the following example:

```
interface PersonRepository extends Repository<Person, Long> { … }
```

2. Declare query methods on the interface.

```
interface PersonRepository extends Repository<Person, Long> {
  List<Person> findByLastname(String lastname);
}
```

3. Set up Spring to create proxy instances for those interfaces, either with JavaConfig or with XML configuration.

JAVA

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories
class Config {}
```

Or XML…

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/data/jpa
     http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

   <jpa:repositories base-package="com.acme.repositories"/>

</beans>
```

DEFINING REPOSITORY INTERFACES

- First, define a domain class-specific repository interface. The interface must extend **Repository** and be typed to the domain class and an ID type. If you want to expose CRUD methods for that domain type, extend **CrudRepository** instead of **Repository**.

DEFINING QUERY METHODS

- The repository proxy has two ways to derive a store-specific query from the method name:
  - By deriving the query from the method name directly.
  - By using a manually defined query.

QUERY LOOKUP STRATEGIES

- For Java configuration, you can use the **queryLookupStrategy** attribute of the **EnableJpaRepositories** annotation. Some strategies may not be supported for particular datastores.
    - **CREATE** attempts to construct a store-specific query from the query method name. The general approach is to remove a given set of well known prefixes from the method name and parse the rest of the method.
    - **USE_DECLARED_QUERY** tries to find a declared query and throws an exception if cannot find one. The query can be defined by an annotation somewhere or declared by other means. Consult the documentation of the specific store to find available options for that store. If the repository infrastructure does not find a declared query for the method at bootstrap time, it fails.
    - **CREATE_IF_NOT_FOUND** (default) combines **CREATE** and **USE_DECLARED_QUERY**. It looks up a declared query first, and, if no declared query is found, it creates a custom method name-based query. This is the default lookup strategy and, thus, is used if you do not configure anything explicitly. It allows quick query definition by method names but also custom-tuning of these queries by introducing declared queries as needed.

QUERY CREATION

- The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository. The mechanism strips the prefixes **find…By**, **read…By**, **query…By**, **count…By**, and **get…By** from the method and starts parsing the rest of it.
- At a very basic level, you can define conditions on entity properties and concatenate them with **And** and **Or**.

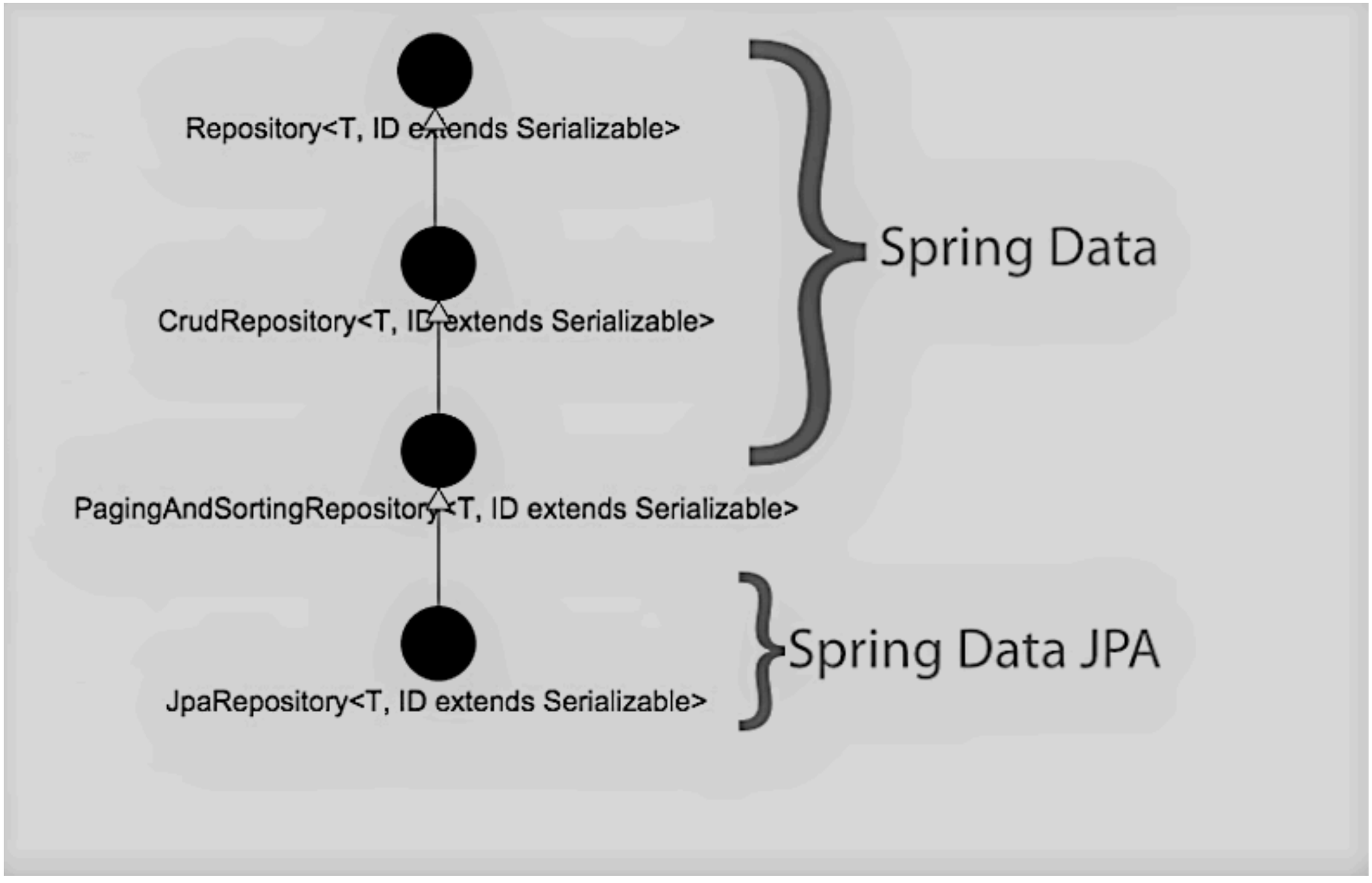
```
interface PersonRepository extends Repository<User, Long> {

  List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

  // Enables the distinct flag for the query
  List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
  List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

  // Enabling ignoring case for an individual property
  List<Person> findByLastnameIgnoreCase(String lastname);
  // Enabling ignoring case for all suitable properties
  List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

  // Enabling static ORDER BY for a query
  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```


Repository<T, ID extends Serializable>
CrudRepository<T, ID extends Serializable>
PagingAndSortingRepository<T, ID extends Serializable>
JpaRepository<T, ID extends Serializable>
Spring Data
Spring Data JPA

```
@Configuration
@EnableJpaRepositories
@EnableTransactionManagement
class ApplicationConfig {

  @Bean
  public DataSource dataSource() {

    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    return builder.setType(EmbeddedDatabaseType.HSQL).build();
  }

  @Bean
  public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

    HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    vendorAdapter.setGenerateDdl(true);

    LocalContainerEntityManagerFactoryBean factory = new
LocalContainerEntityManagerFactoryBean();
    factory.setJpaVendorAdapter(vendorAdapter);
    factory.setPackagesToScan("com.acme.domain");
    factory.setDataSource(dataSource());
    return factory;
  }

  @Bean
  public PlatformTransactionManager transactionManager(EntityManagerFactory
entityManagerFactory) {

    JpaTransactionManager txManager = new JpaTransactionManager();
    txManager.setEntityManagerFactory(entityManagerFactory);
    return txManager;
  }
}
```

TEXT

- As of Spring Data JPA 2.1 you can now configure a BootstrapMode (either via the @EnableJpaRepositories annotation or the XML namespace) that takes the following values:
  - DEFAULT (default) – Repositories are instantiated eagerly unless explicitly annotated with @Lazy. The lazification only has effect if no client bean needs an instance of the repository as that will require the initialization of the repository bean.
  - LAZY – Implicitly declares all repository beans lazy and also causes lazy initialization proxies to be created to be injected into client beans. That means, that repositories will not get instantiated if the client bean is simply storing the instance in a field and not making use of the repository during initialization. Repository instances will be initialized and verified upon first interaction with the repository.
  - DDEFERRED – Fundamentally the same mode of operation as LAZY, but triggering repository initialization in response to an ContextRefreshedEvent so that repositories are verified before the application has completely started.

SPRING DATA JPA

- Spring Data JPA repositories are interfaces that you can define to access data. JPA queries are created automatically from your method names. For example, a CityRepository interface might declare a findAllByState(String state) method to find all the cities in a given state.
- For more complex queries, you can annotate your method with Spring Data's Query annotation.
- Spring Data repositories usually extend from the Repository or CrudRepository interfaces. If you use auto-configuration, repositories are searched from the package containing your main configuration class (the one annotated with **@EnableAutoConfiguration** or **@SpringBootApplication**) down.
- Spring Data JPA repositories support three different modes of bootstrapping: default, deferred, and lazy. To enable deferred or lazy bootstrapping, set the **spring.data.jpa.repositories.bootstrap-mode** to deferred or lazy respectively. When using deferred or lazy bootstrapping, the auto-configured **EntityManagerFactoryBuilder** will use the context's **AsyncTaskExecutor**, if any, as the bootstrap executor. If more than one exists, the one named **applicationTaskExecutor** will be used.

Keyword	Sample	JPQL snippet
And	`findByLastnameAndFirstname`	`… where x.lastname = ?1 and x.firstname = ?2`
Or	`findByLastnameOrFirstname`	`… where x.lastname = ?1 or x.firstname = ?2`
Is,Equals	`findByFirstname`, `findByFirstnameIs`, `findByFirstnameEquals`	`… where x.firstname = ?1`
Between	`findByStartDateBetween`	`… where x.startDate between ?1 and ?2`
LessThan	`findByAgeLessThan`	`… where x.age < ?1`
LessThanEqual	`findByAgeLessThanEqual`	`… where x.age <= ?1`
GreaterThan	`findByAgeGreaterThan`	`… where x.age > ?1`
GreaterThanEqual	`findByAgeGreaterThanEqual`	`… where x.age >= ?1`
After	`findByStartDateAfter`	`… where x.startDate > ?1`
Before	`findByStartDateBefore`	`… where x.startDate < ?1`
IsNull	`findByAgeIsNull`	`… where x.age is null`
IsNotNull,NotNull	`findByAge(Is)NotNull`	`… where x.age not null`

Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?1)

USING @QUERY

- Using named queries to declare queries for entities is a valid approach and works fine for a small number of queries. As the queries themselves are tied to the Java method that executes them, you can actually bind them directly by using the Spring Data JPA @Query annotation rather than annotating them to the domain class. This frees the domain class from persistence specific information and co-locates the query to the repository interface.

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.emailAddress = ?1")
  User findByEmailAddress(String emailAddress);
}
```

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.firstname like %?1")
  List<User> findByFirstnameEndsWith(String firstname);
}
```

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.lastname like ?1%")
  List<User> findByAndSort(String lastname, Sort sort);

  @Query("select u.id, LENGTH(u.firstname) as fn_len from User u where u.lastname like ?1%")
  List<Object[]> findByAsArrayAndSort(String lastname, Sort sort);
}
```

```
repo.findByAndSort("lannister", new Sort("firstname"));                 (1)
repo.findByAndSort("stark", new Sort("LENGTH(firstname)"));             (2)
repo.findByAndSort("targaryen", JpaSort.unsafe("LENGTH(firstname)"));   (3)
repo.findByAsArrayAndSort("bolton", new Sort("fn_len"));                (4)
```

1. Valid `Sort` expression pointing to property in domain model.
2. Invalid `Sort` containing function call. Thows Exception.
3. Valid `Sort` containing explicitly *unsafe* `Order`.
4. Valid `Sort` expression pointing to aliased function.

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.firstname = :firstname or u.lastname = :lastname")
  User findByLastnameOrFirstname(@Param("lastname") String lastname,
                                 @Param("firstname") String firstname);
}
```