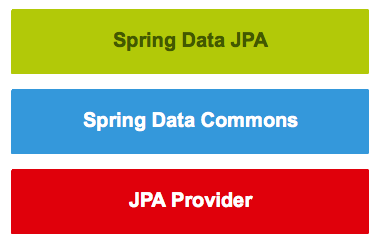
**Spring Data JPA is not a JPA provider**.

It is a library / framework that adds an extra layer of abstraction on the top of our JPA provider(It simply “hides” the Java Persistence API (and the JPA provider) behind its repository abstraction.)

If we decide to use Spring Data JPA, the repository layer of our application contains three layers that are described in the following:

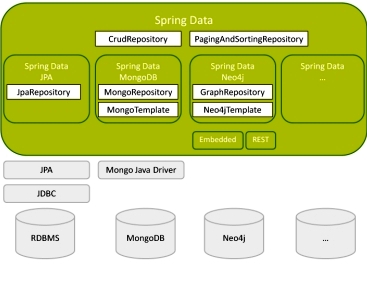
* [Spring Data JPA](http://projects.spring.io/spring-data-jpa/) provides support for creating JPA repositories by extending the Spring Data repository interfaces.
* [Spring Data Commons](https://github.com/spring-projects/spring-data-commons) provides the infrastructure that is shared by the datastore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

[](http://www.javacodegeeks.com/wp-content/uploads/2014/12/springdatajpalayers.png)

Spring Data JPA is part of the umbrella Spring Data project that makes it easier to implement JPA based repositories.

**Features:**

* Sophisticated support to build repositories based on Spring and JPA
* Support for QueryDSL predicates and thus type-safe JPA queries
* Transparent auditing of domain class
* Pagination support, dynamic query execution, ability to integrate custom data access code
* Validation of @Query annotated queries at bootstrap time
* Support for XML based entity mapping
* JavaConfig based repository configuration by introducing @EnableJpaRepositories  
    
  [](https://i.stack.imgur.com/lQkpL.jpg)

# Spring Data repository interfaces

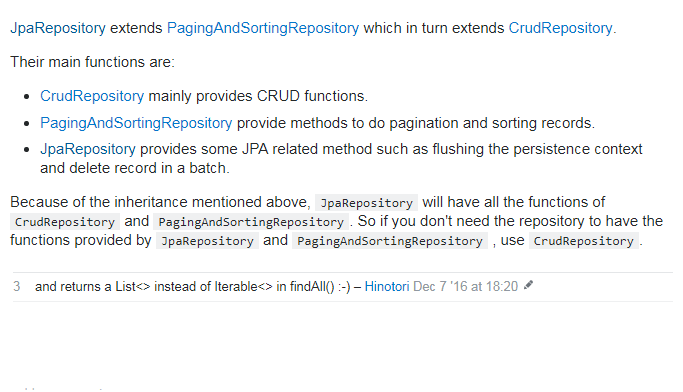
# 

* 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.

# [What is difference between CrudRepository and JpaRepository interfaces in Spring Data](https://stackoverflow.com/questions/14014086/what-is-difference-between-crudrepository-and-jparepository-interfaces-in-spring)



How to configure your Spring application with Simple CRUD configuration with Spring Data-JPA.

### 1. Add Spring Data-JPA to project configuration. In your Maven pom.xml file

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-jpa</artifactId>

<version>1.0.0.RELEASE</version>

</dependency>

### 2. Configure JPA Entity

...

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = "id")

private Long id;

@NotNull

@Column(unique = true)

private String productId;

@NotNull

private Integer quantity;

...

}

### 3. Configure Typed Repository Interface

Spring Data JPA will create the beans for us.

package com.gordondickens.myapp.repository;

import org.springframework.data.repository.CrudRepository;

import com.gordondickens.myapp.entity.Product;

public interface ProductRepository extends CrudRepository<Product, Long> {}

### 4. Configure the application context

Note: this example uses Hibernate and HSQL

<!-- Directory to scan for repository classes -->

<jpa:repositories

base-package="com.gordondickens.myapp.repository" />

<bean class="org.springframework.orm.jpa.JpaTransactionManager"

id="transactionManager">

<property name="entityManagerFactory"

ref="entityManagerFactory" />

<property name="jpaDialect">

<bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect" />

</property>

</bean>

<bean id="entityManagerFactory"

class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

<property name="dataSource" ref="dataSource" />

<property name="jpaVendorAdapter">

<bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

<property name="generateDdl" value="true" />

<property name="database" value="HSQL" />

</bean>

</property>

</bean>

### 5. Inject the Repository

In our service class & tests we use the repository to execute our crud methods. Repository Methods: count(), exists(), delete(), deleteAll(), findOne(), findAll(), save().

public class ProductServiceImpl extends ProductService {

@Autowired

ProductRepository productRepository;

...

}

**Modify the Repository Interface**

Simply define the method interface and use the @Query annotation to define the OQL.

package com.gordondickens.myapp.repository;

import org.springframework.data.repository.CrudRepository;

import com.gordondickens.myapp.entity.Product;

public interface ProductRepository extends CrudRepository<Product, Long> {

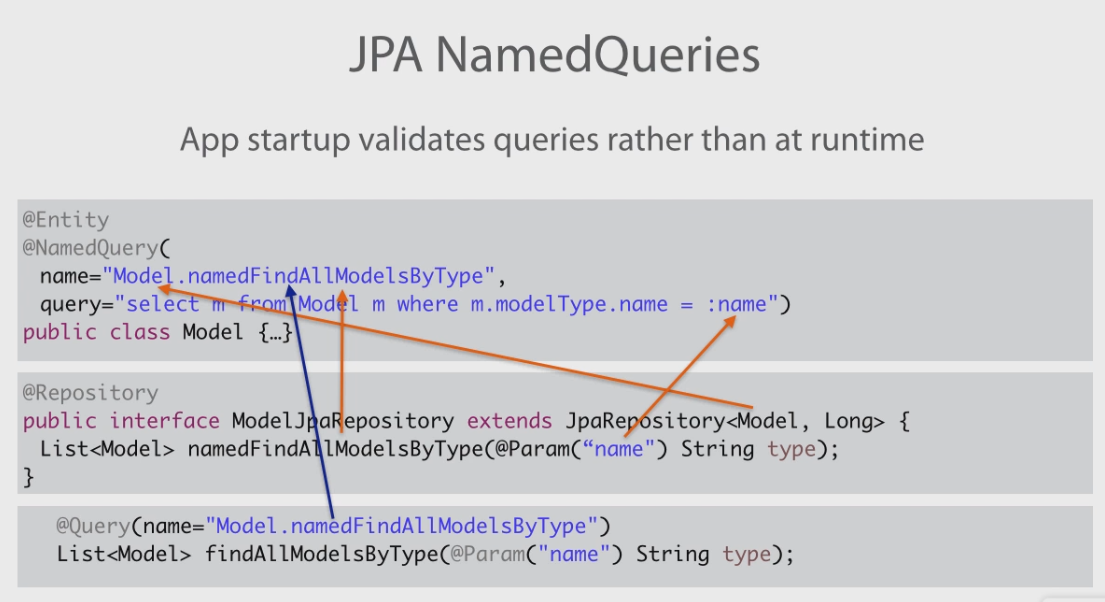
@Query("FROM Product")

List<Product> findAllProducts();

}

**Modifying Queries**

* Parameters can be marked by position with ?1, ?2, etc.
* Parameters can be marked by name with :paramName and annotation @Param("paramName")
* Modifying method signature can only return void, Integer or int
* Updating queries MUST be transactional, mark with @Transactional
* Spring Data will drop all non-flushed changes pending in the EntityManager, change with @Modifying(clearAutomatically=false)



package com.gordondickens.myapp.repository;

import org.springframework.data.jpa.repository.Modifying;

import org.springframework.data.jpa.repository.Query;

import org.springframework.data.repository.CrudRepository;

import org.springframework.data.repository.query.Param;

import org.springframework.transaction.annotation.Transactional;

@Transactional(readOnly=true)

public interface ProductRepository extends CrudRepository<Product, Long> {

@Query("FROM Product")

List<Product> findAllProducts();

// Example with positional params

@Modifying

@Transactional(readOnly=false)

@Query("update Product p set p.description = ?2 where p.productId = ?1")

Integer setNewDescriptionForProduct(String productId, String description);

// Example with named params

@Modifying

@Query("update Product p set p.description = :description where p.productId = :productId")

Integer setNewDescriptionForProduct(@Param("productId") String productId,

@Param("description") String description);

}

**Automatic Query Generation**

The <jpa:repositories/> has an option query-lookup-strategy which defaults to “create-if-not-found” which will generate queries for us.

The default is “create-if-not-found“. Other options are “create” or “use-declared-query“.

<jpa:repositories base-package="com.gordondickens.myapp.repository"

query-lookup-strategy="create-if-not-found"/>

To create a find method that effectively does @Query("FROM Product p where p.productId = :productId")

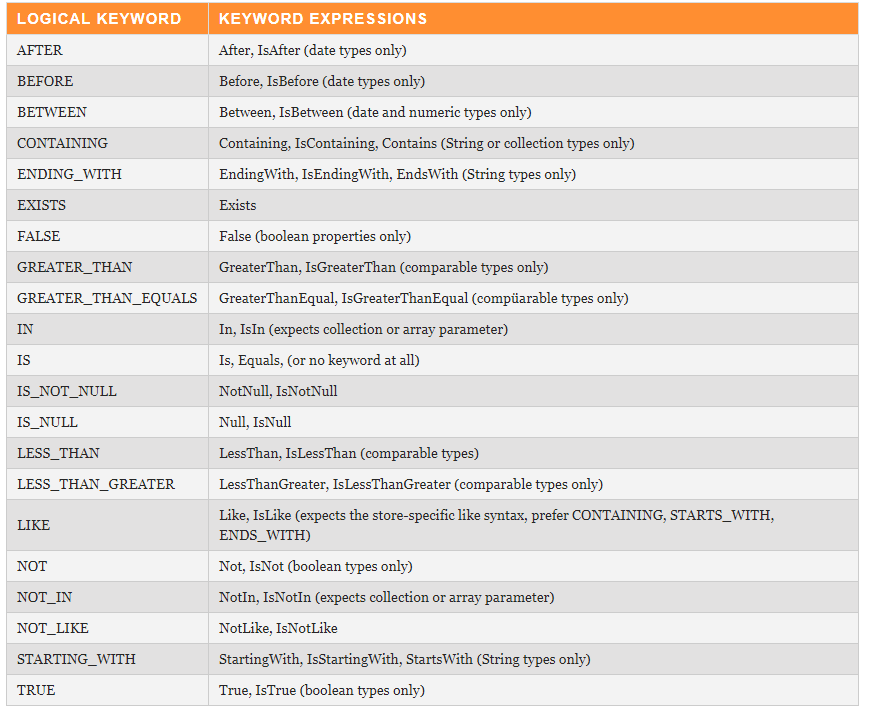
public interface ProductRepository extends CrudRepository<Product, Long> {

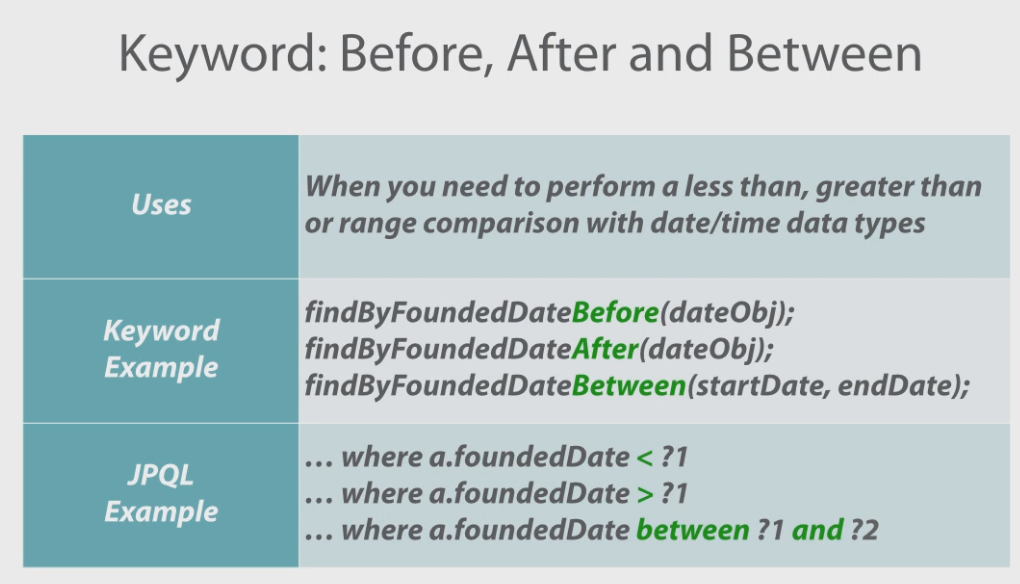
...

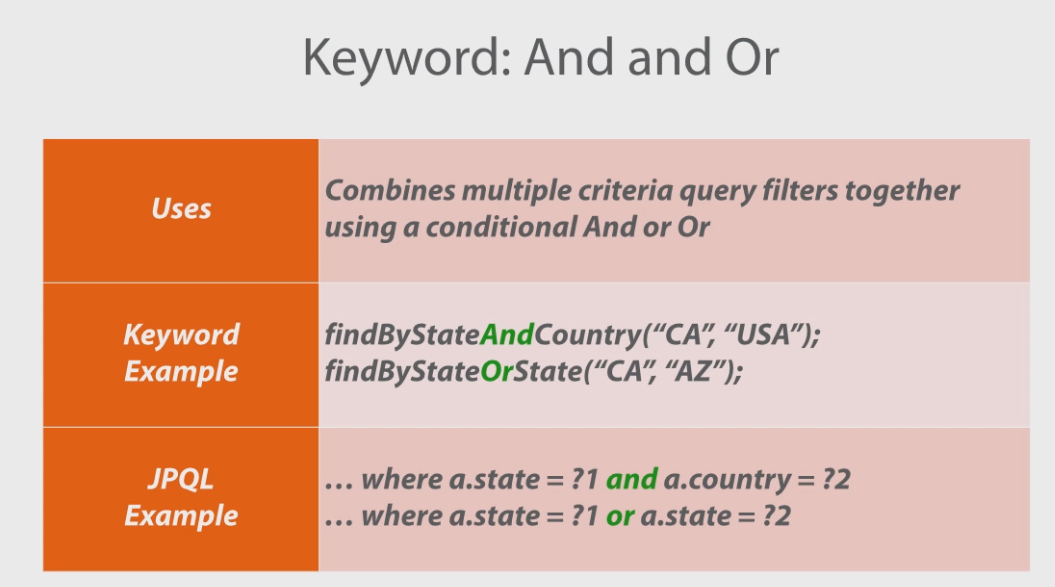
@Query

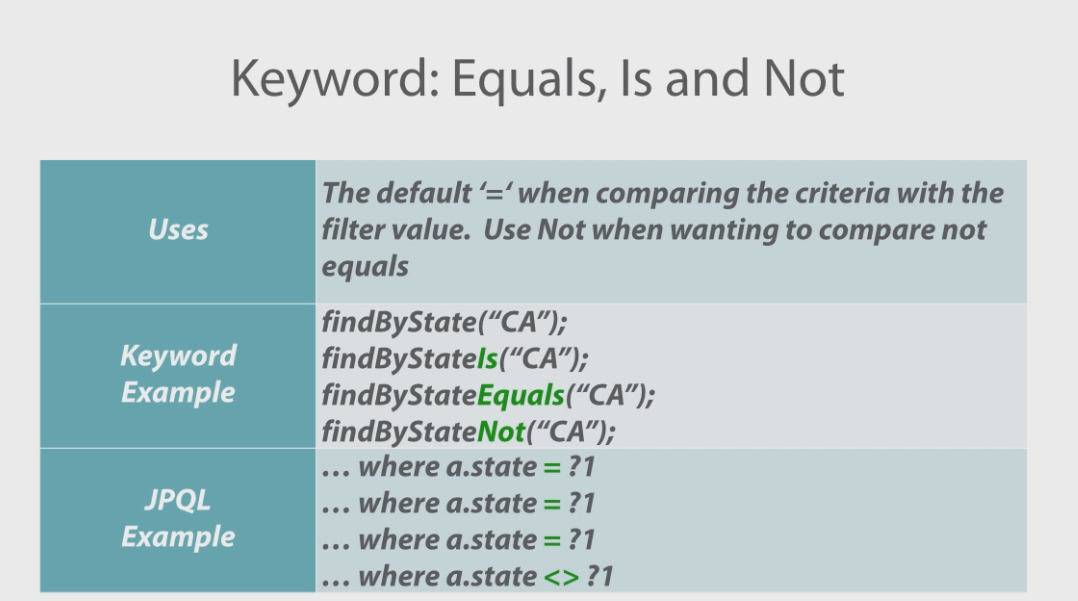
Product findByProductId(String productId);

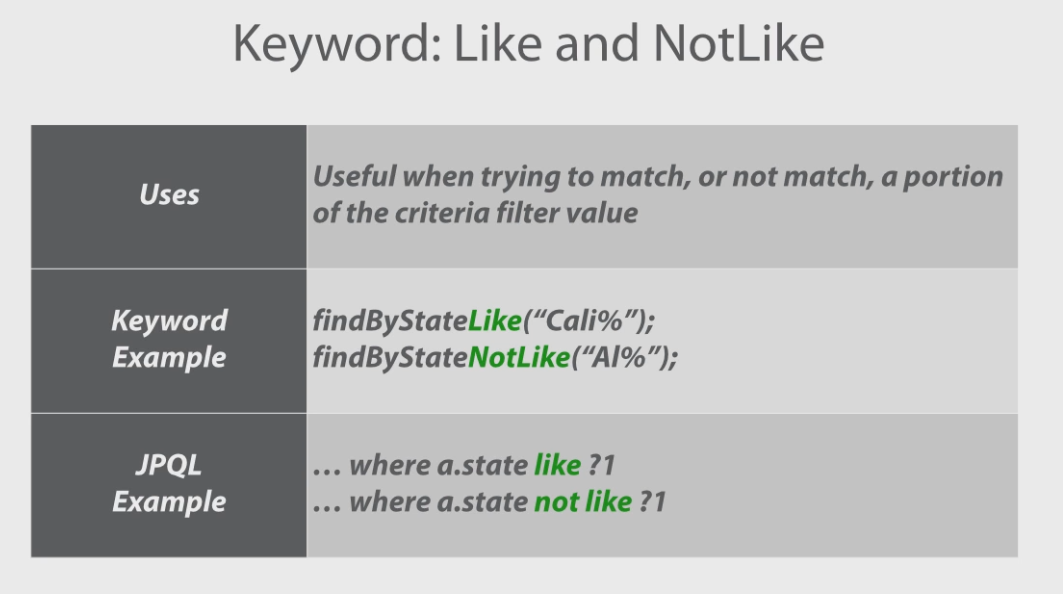
...

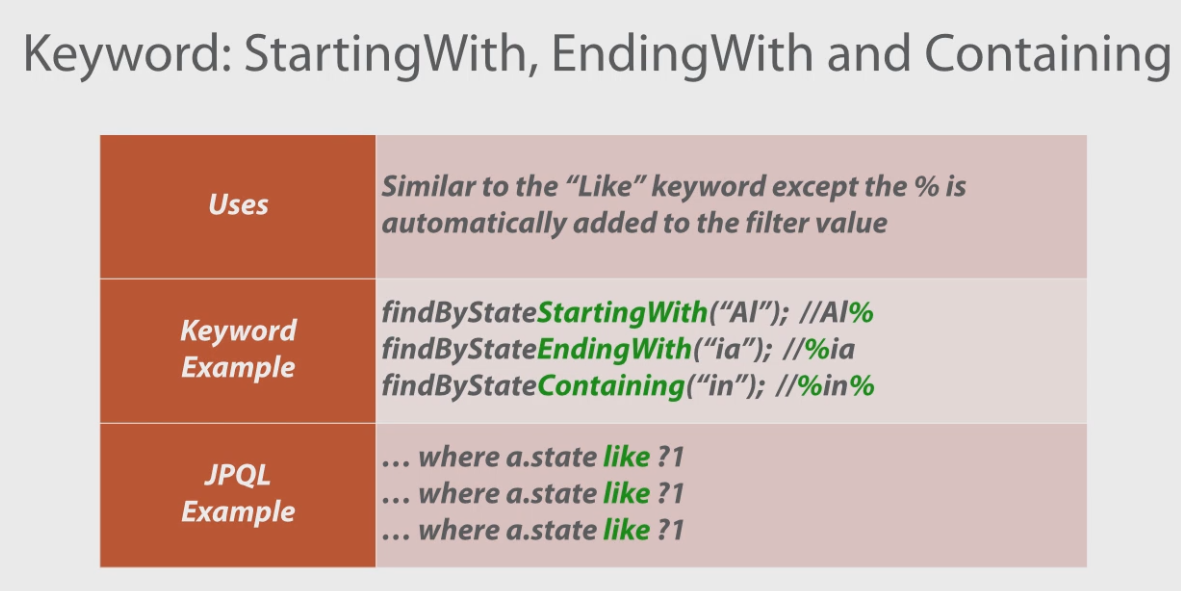
****

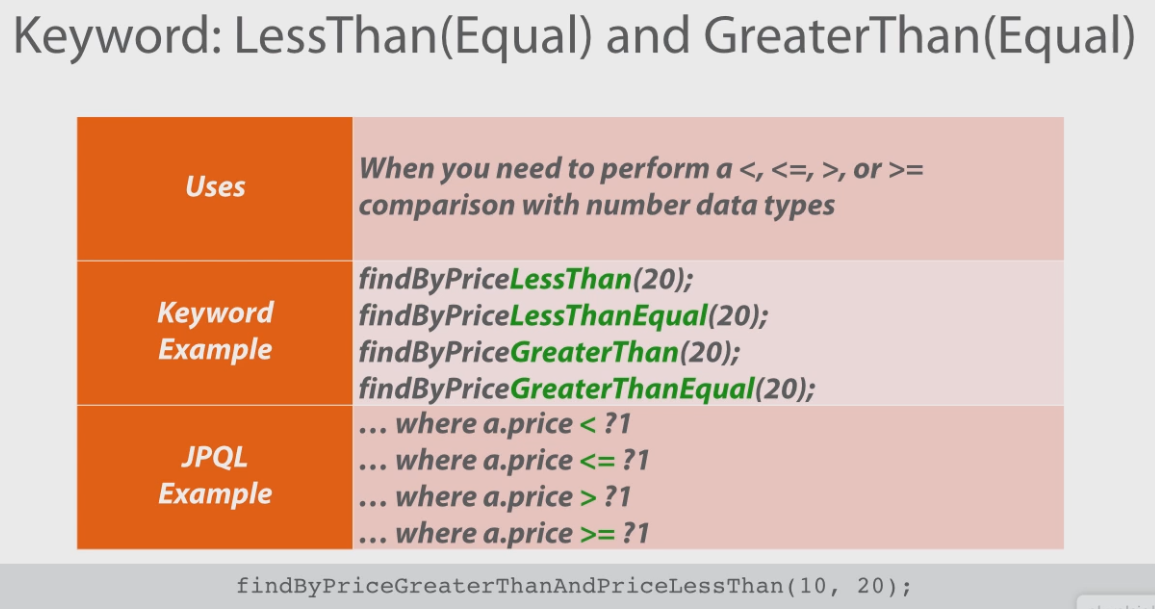
****

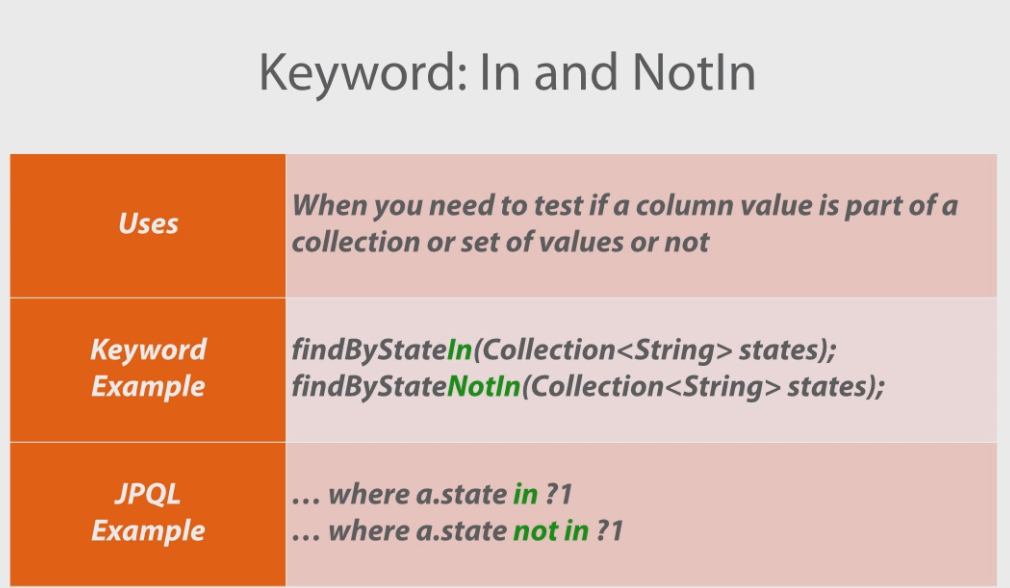
****

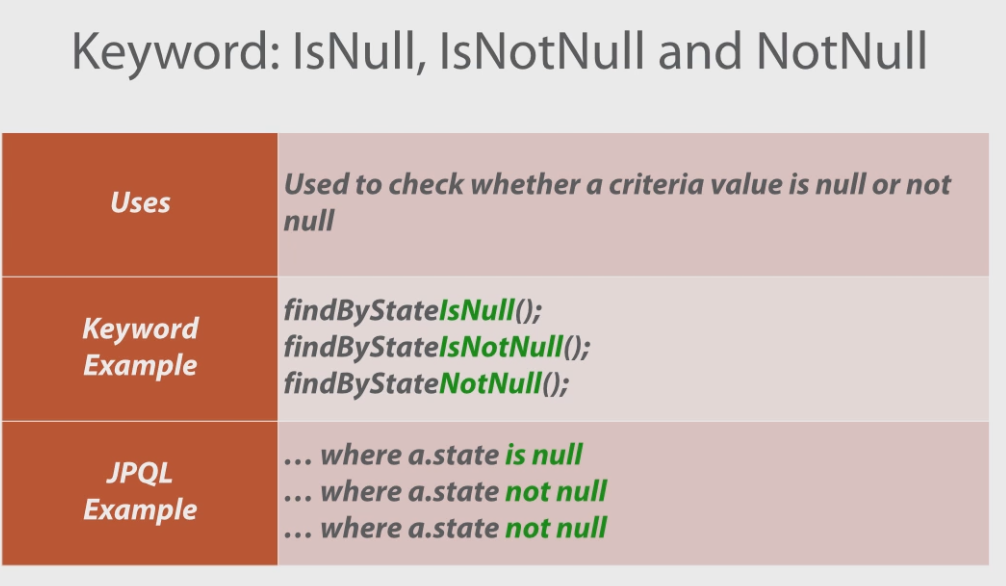
****

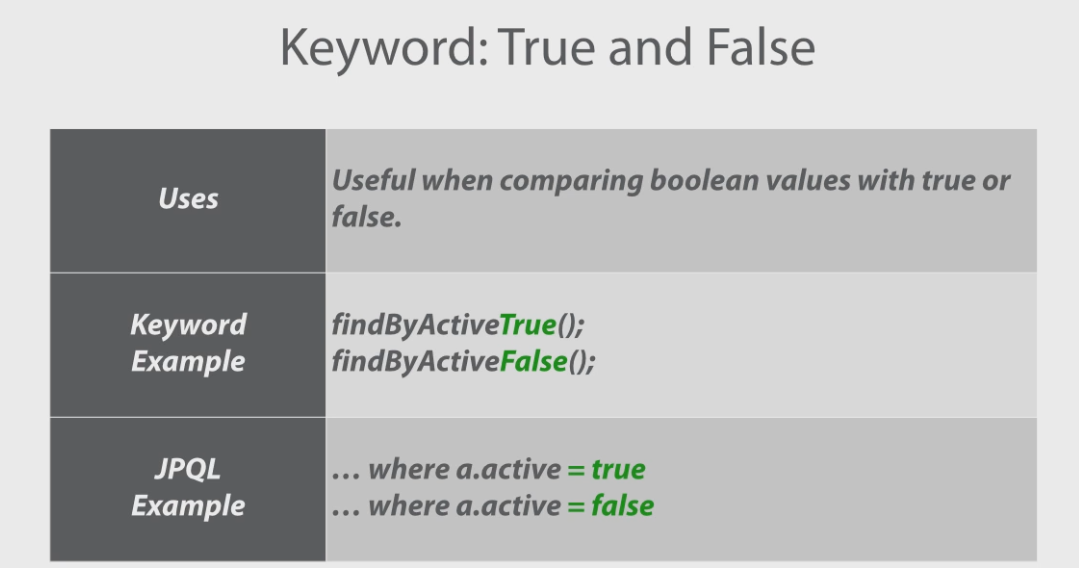
****

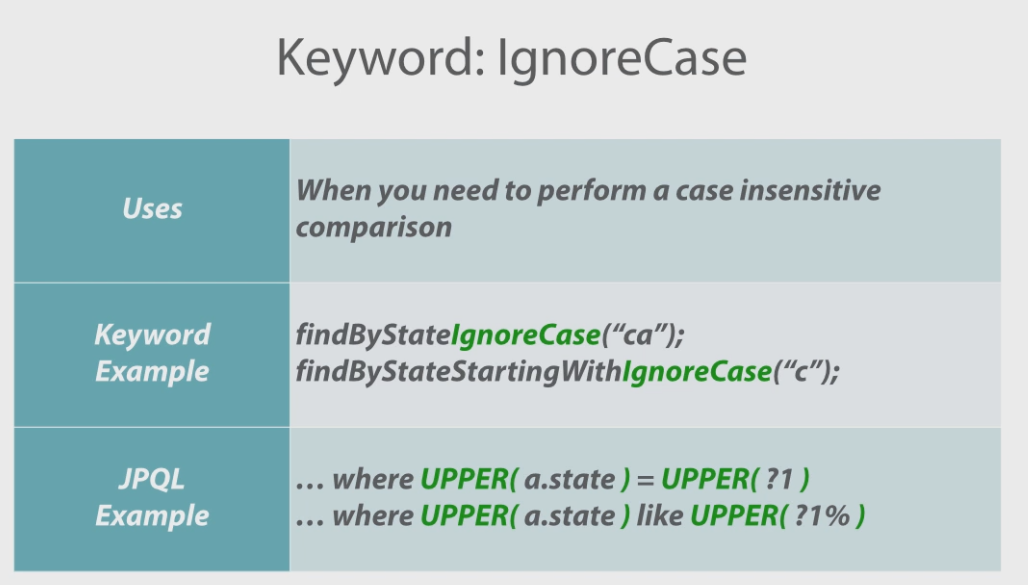
****

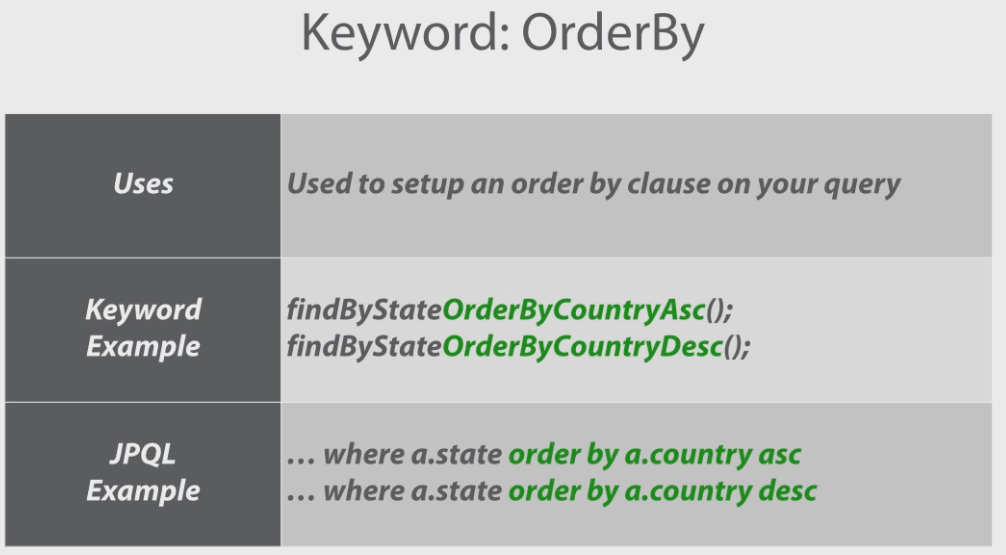
****

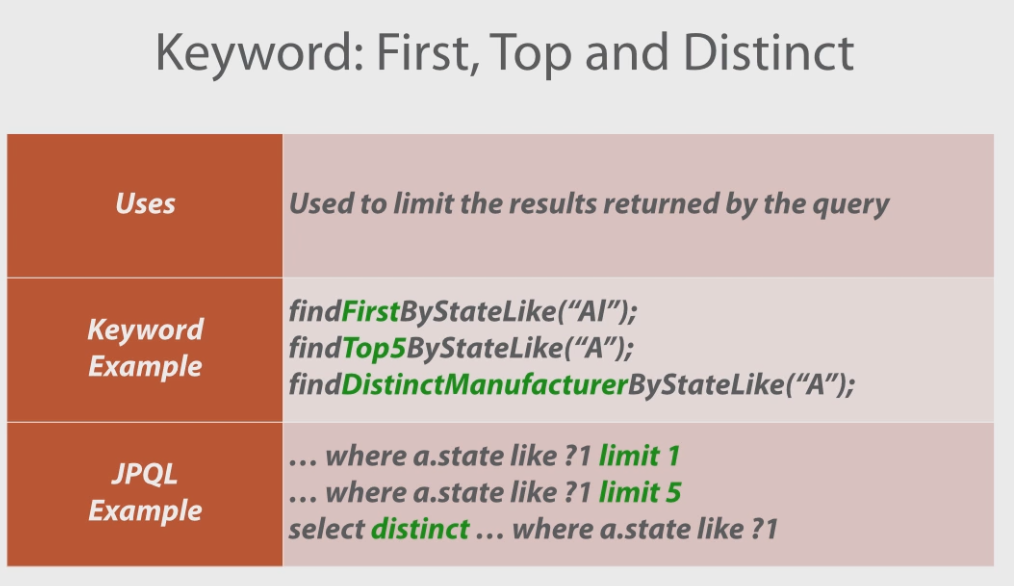
****

****

****

****

****

****

**Summary**

We see how simple interface additions provide custom methods based on query language. We can query either by positional or named parameters.