



UNIVERSITÀ
DELLA CALABRIA

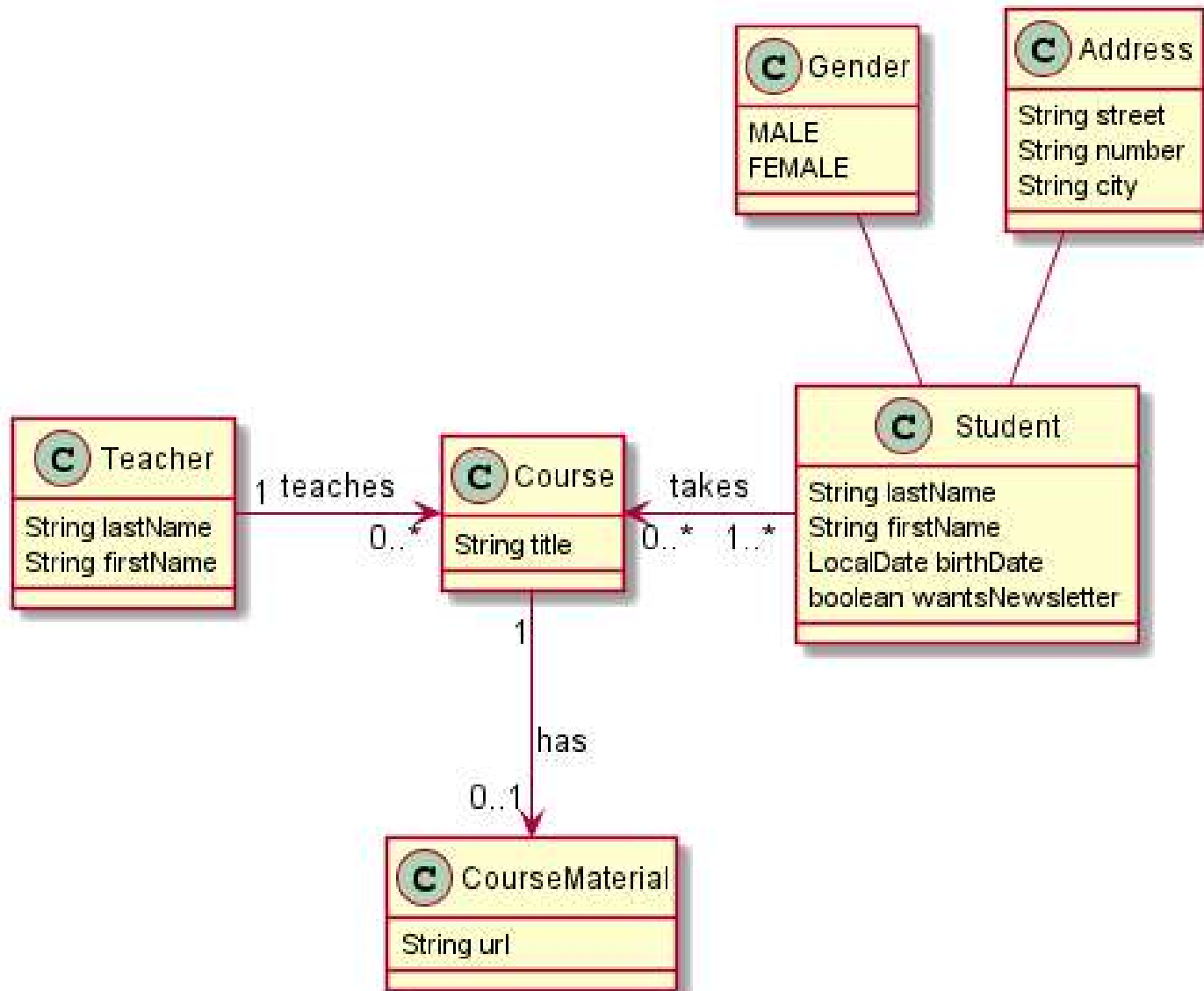
DIPARTIMENTO DI **MATEMATICA
E INFORMATICA**

AGILE SOFTWARE DEVELOPMENT

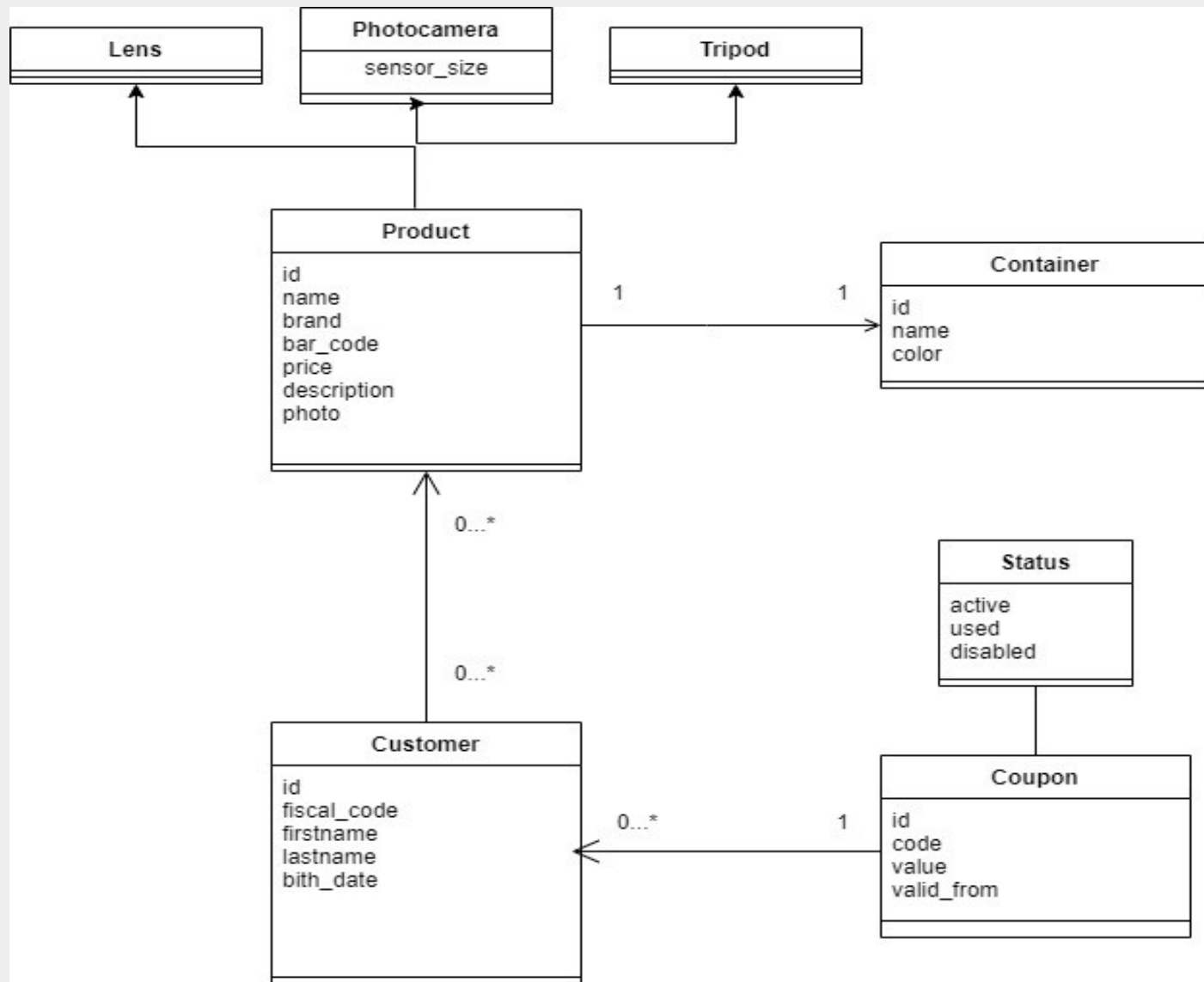
Dr. Francesco Scalzo - francesco.scalzo@unical.it
Enterprise Solution Engineer at **Revelis s.r.l.**

Office hours & info: send a mail
Code: https://github.com/mascalzonef/asd_21-22.git

Exercise

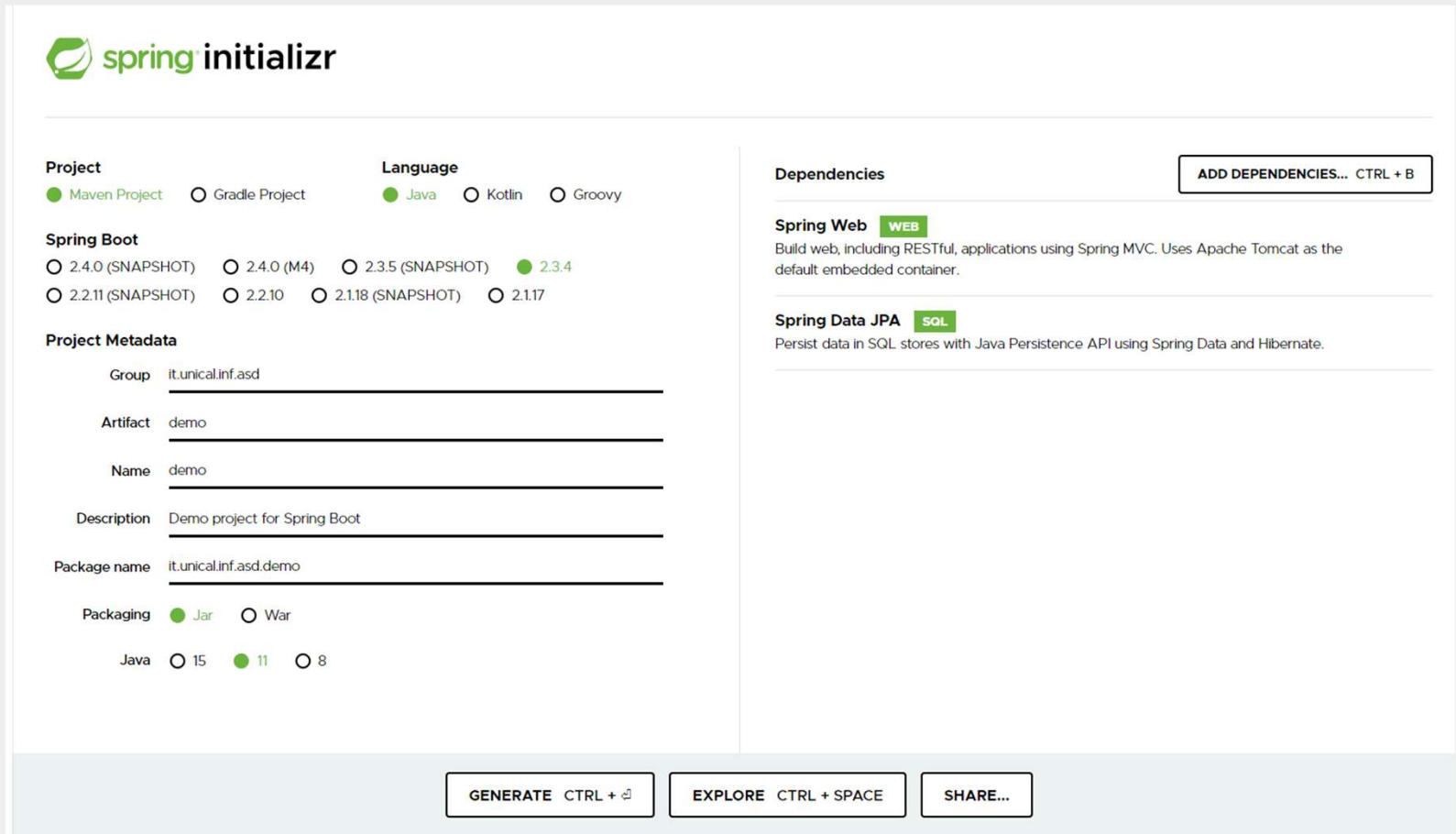


Exercise



Spring Project & IntelliJ

Spring initializr



The image shows the Spring Initializr web form. It is divided into several sections: Project, Language, Spring Boot, Project Metadata, Dependencies, and a bottom bar with action buttons.

Project
☒ Maven Project ☐ Gradle Project

Language
☒ Java ☐ Kotlin ☐ Groovy

Spring Boot
☐ 2.4.0 (SNAPSHOT) ☐ 2.4.0 (M4) ☐ 2.3.5 (SNAPSHOT) ☒ 2.3.4
☐ 2.2.11 (SNAPSHOT) ☐ 2.2.10 ☐ 2.1.18 (SNAPSHOT) ☐ 2.1.17

Project Metadata
Group:
Artifact:
Name:
Description:
Package name:
Packaging: ☒ Jar ☐ War
Java: ☐ 15 ☒ 11 ☐ 8

Dependencies ADD DEPENDENCIES... CTRL + B
Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Bottom Bar:
GENERATE CTRL + G EXPLORE CTRL + SPACE SHARE...

Generate the file and unzip it into project folder



1. File → New → Module from Existing sources...
2. Select the unzipped folder
3. Import module from external model → Maven
4. Finish

Add logback.xml for the log

And add db-driver and junit dependencies

```
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <version>2.4.0</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <scope>test</scope>
</dependency>
```



Every repository in Spring Data extends the generic Repository interface

1. **CrudRepository** provides CRUD functions
2. **PagingAndSortingRepository** provides methods to do pagination and sort records
3. **JpaRepository** provides JPA related methods such as flushing the persistence context and delete records in a batch

- `save(...)` – save an Iterable of entities. Here, we can pass multiple objects to save them in a batch
- `findOne(...)` – get a single entity based on passed primary key value
- `findAll()` – get an Iterable of all available entities in database
- `count()` – return the count of total entities in a table
- `delete(...)` – delete an entity based on the passed object
- `exists(...)` – verify if an entity exists based on the passed primary key value

- `findAll(Sort sort)` - get a **sorted** Iterable of all available entities
- `findAll(Pageable pageable)` - get a **sorted and paginated** Iterable of all available entities

- *`findAll()`* – get a *List* of all available entities in database
- *`findAll(...)`* – get a *List* of all available entities and sort them using the provided condition
- *`save(...)`* – save an *Iterable* of entities. Here, we can pass multiple objects to save them in a batch
- *`flush()`* – flush all pending task to the database
- *`saveAndFlush(...)`* – save the entity and flush changes immediately
- *`deleteInBatch(...)`* – we can pass multiple objects to delete them in a batch



Query methods are methods that find information from the database and are declared on the repository interface.

```
1 import org.springframework.data.repository.Repository;
2
3 interface TodoRepository extends Repository<Todo, Long> {
4
5     //This is a query method.
6     Todo findById(Long id);
7 }
```

A query method can return only one result or more than one result:

- Basic type. Our query method will return the found basic type or null.
- Entity. Our query method will return an entity object or null.
- Optional<T>. Our query method will return an Optional that contains the found object or an empty Optional.

```
1 import java.util.Optional;
2 import org.springframework.data.jpa.repository.Query;
3 import org.springframework.data.repository.Repository;
4 import org.springframework.data.repository.query.Param;
5
6 interface TodoRepository extends Repository<Todo, Long> {
7
8     @Query("SELECT t.title FROM Todo t where t.id = :id")
9     String findTitleById(@Param("id") Long id);
10
11     @Query("SELECT t.title FROM Todo t where t.id = :id")
12     Optional<String> findTitleById(@Param("id") Long id);
13
14     Todo findById(Long id);
15
16     Optional<Todo> findById(Long id);
17 }
```



if we are writing a query method that should return more than one result, we can return the following types:

- *List<T>*. Our query method will return a list that contains the query results or an empty list.
- *Stream<T>*. Our query method will return a *Stream* that can be used to access the query results or an empty *Stream*.

```
1 import java.util.stream.Stream;
2 import org.springframework.data.repository.Repository;
3
4 interface TodoRepository extends Repository<Todo, Long> {
5
6     List<Todo> findByTitle(String title);
7
8     Stream<Todo> findByTitle(String title);
9 }
```

if we want that our query method is executed asynchronously, we have to annotate it with the `@Async` annotation and return a `Future<T>` object. Here are some examples of query methods that are executed asynchronously:

```
interface TodoRepository extends Repository<Todo, Long> {

    @Async
    @Query("SELECT t.title FROM Todo t where t.id = :id")
    Future<String> findTitleById(@Param("id") Long id);

    @Async
    @Query("SELECT t.title FROM Todo t where t.id = :id")
    Future<Optional<String>> findTitleById(@Param("id") Long id);

    @Async
    Future<Todo> findById(Long id);

    @Async
    Future<Optional<Todo>> findById(Long id);

    @Async
    Future<List<Todo>> findByTitle(String title);

    @Async
    Future<Stream<Todo>> findByTitle(String title);
}
```



Table 3. Supported keywords inside method names

Keyword	Sample	JPQL snippet
And	<code>findByLastnameAndFirstname</code>	<code>... where x.lastname = ?1 and x.firstname = ?2</code>
Or	<code>findByLastnameOrFirstname</code>	<code>... where x.lastname = ?1 or x.firstname = ?2</code>
Is, Equals	<code>findByFirstname</code> , <code>findByFirstnameIs</code> , <code>findByFirstnameEquals</code>	<code>... where x.firstname = ?1</code>
Between	<code>findByStartDateBetween</code>	<code>... where x.startDate between ?1 and ?2</code>
LessThan	<code>findByAgeLessThan</code>	<code>... where x.age < ?1</code>
LessThanEqual	<code>findByAgeLessThanEqual</code>	<code>... where x.age <= ?1</code>
GreaterThan	<code>findByAgeGreaterThan</code>	<code>... where x.age > ?1</code>
GreaterThanEqual	<code>findByAgeGreaterThanEqual</code>	<code>... where x.age >= ?1</code>
After	<code>findByStartDateAfter</code>	<code>... where x.startDate > ?1</code>
Before	<code>findByStartDateBefore</code>	<code>... where x.startDate < ?1</code>
IsNull, Null	<code>findByAge(Is)Null</code>	<code>... where x.age is null</code>

Spring Data

Query Methods

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.firstname) = UPPER(?1)

- It supports both JPQL and SQL queries, and the query that is specified by using the *@Query* annotation
- Precedes all other query generation strategies.

```
1 import org.springframework.data.repository.Repository;
2
3 import java.util.Optional;
4
5 interface TodoRepository extends Repository<Todo, Long> {
6
7     @Query(value = "SELECT * FROM todos t WHERE t.title = 'title'",
8           nativeQuery=true
9           )
10    public List<Todo> findByTitle();
11 }
```

```
1 import org.springframework.data.repository.Repository;
2
3 import java.util.Optional;
4
5 interface TodoRepository extends Repository<Todo, Long> {
6
7     @Query("SELECT t FROM Todo t WHERE t.title = 'title'")
8    public List<Todo> findByTitle();
9 }
```

```
1 import org.springframework.data.jpa.repository.Query;
2 import org.springframework.data.repository.Repository;
3 import org.springframework.data.repository.query.Param;
4
5 import java.util.List;
6
7 interface TodoRepository extends Repository<Todo, Long> {
8
9     @Query(value = "SELECT * FROM todos t WHERE " +
10           "LOWER(t.title) LIKE LOWER(CONCAT('%',:searchTerm, '%')) OR " +
11           "LOWER(t.description) LIKE LOWER(CONCAT('%',:searchTerm, '%'))",
12           nativeQuery = true
13           )
14    List<Todo> findBySearchTermNative(@Param("searchTerm") String searchTerm);
15 }
```



Query methods benefits:

- Creating simple queries is fast.
- The method name of our query method describes the selected value(s) and the used search condition(s).

Query methods weaknesses:

- The features of the method name parser determine what kind of queries we can create. If the method name parser doesn't support the required keyword, we cannot use this strategy.
- The method names of complex query methods are long and ugly.
- There is no support for dynamic queries.

Query methods @Query benefits:

- It supports both JPQL and SQL.
- The invoked query is found above the query method. In other words, it is easy to find out what the query method does.
- There is no naming convention for query method names.

Query methods @Query weaknesses:

- There is no support for dynamic queries.
- If we use SQL queries, we cannot change the used database without testing that our SQL queries work as expected.



```
1 import java.util.Optional
2 import org.springframework.data.jpa.repository.Query;
3 import org.springframework.data.repository.Repository;
4
5
6 interface TodoRepository extends Repository<Todo, Long> {
7
8     public Optional<Todo> findByTitleAndDescription(String title, String description);
9
10    @Query("SELECT t FROM Todo t where t.title = ?1 AND t.description = ?2")
11    public Optional<Todo> findByTitleAndDescription(String title, String description);
12
13    @Query(value = "SELECT * FROM todos t where t.title = ?0 AND t.description = ?1",
14           nativeQuery=true
15    )
16    public Optional<Todo> findByTitleAndDescription(String title, String description);
17 }
```

```
1 import java.util.Optional
2 import org.springframework.data.jpa.repository.Query;
3 import org.springframework.data.repository.Repository;
4 import org.springframework.data.repository.query.Param;
5
6
7 interface TodoRepository extends Repository<Todo, Long> {
8
9     @Query("SELECT t FROM Todo t where t.title = :title AND t.description = :description")
10    public Optional<Todo> findByTitleAndDescription(@Param("title") String title,
11                                                    @Param("description") String description);
12
13    @Query(
14        value = "SELECT * FROM todos t where t.title = :title AND t.description = :description",
15        nativeQuery=true
16    )
17    public Optional<Todo> findByTitleAndDescription(@Param("title") String title,
18                                                    @Param("description") String description);
19 }
```

The *@Param* annotation configures the name of the named parameter that is replaced with the value of the method parameter

Spring Data

Query Example

```
1 import org.springframework.data.repository.Repository;
2
3 interface TodoRepository extends Repository<Todo, Long> {
4
5     /**
6      * Returns the number of todo entry whose title is given
7      * as a method parameter.
8      */
9     public long countByTitle(String title);
10 }
```

```
1 import org.springframework.data.repository.Repository;
2
3 import java.util.List;
4
5 interface TodoRepository extends Repository<Todo, Long> {
6
7     /**
8      * Returns the distinct todo entries whose title is given
9      * as a method parameter. If no todo entries is found, this
10     * method returns an empty list.
11     */
12     public List<Todo> findDistinctByTitle(String title);
13 }
```

```
1 import org.springframework.data.repository.Repository;
2
3 import java.util.List;
4
5 interface TodoRepository extends Repository<Todo, Long> {
6
7     /**
8      * Returns the found todo entry whose title or description is given
9      * as a method parameter. If no todo entry is found, this method
10     * returns an empty list.
11     */
12     public List<Todo> findByTitleOrDescription(String title, String description);
13 }
```



Spring Data

Query Example

```
1 import org.springframework.data.repository.Repository;
2
3 import java.util.Optional;
4
5 interface TodoRepository extends Repository<Todo, Long> {
6
7     /**
8      * Returns the found todo entry by using its id as search
9      * criteria. If no todo entry is found, this method
10      * returns null.
11      */
12     public Todo findById(Long id);
13
14     /**
15      * Returns an Optional which contains the found todo
16      * entry by using its id as search criteria. If no to entry
17      * is found, this method returns an empty Optional.
18      */
19     public Optional<Todo> findById(Long id);
20 }
```

```
1 import org.springframework.data.repository.Repository;
2
3 import java.util.List;
4
5 interface TodoRepository extends Repository<Todo, Long> {
6
7     /**
8      * Returns the first three todo entries whose title is given
9      * as a method parameter. If no todo entries is found, this
10      * method returns an empty list.
11      */
12     public List<Todo> findFirst3ByTitleOrderByTitleAsc(String title);
13
14     /**
15      * Returns the first three todo entries whose title is given
16      * as a method parameter. If no todo entries is found, this
17      * method returns an empty list.
18      */
19     public List<Todo> findTop3ByTitleOrderByTitleAsc(String title);
20 }
```



The *JpaSpecificationExecutor*<T> interface declares the methods that can be used to invoke database queries that use the JPA Criteria API.

```
1 import org.springframework.data.jpa.repository.JpaSpecificationExecutor;
2 import org.springframework.data.repository.Repository;
3
4 interface TodoRepository extends Repository<Todo, Long>, JpaSpecificationExecutor<Todo> {
5 }
```

After we have extended the *JpaSpecificationExecutor* interface, the classes that use our repository interface get access to the following methods:

- The `long count(Specification<T> spec)` method returns the number of objects that fulfil the conditions specified by the `Specification<T>` object given as a method parameter.
- The `List<T> findAll(Specification<T> spec)` method returns objects that fulfil the conditions specified by the `Specification<T>` object given as a method parameter.
- The `T findOne(Specification<T> spec)` method returns an object that fulfils the conditions specified by the `Specification<T>` object given as a method parameter.



DEFINITION

```

1  import org.springframework.data.jpa.domain.Specification;
2
3  import javax.persistence.criteria.CriteriaBuilder;
4  import javax.persistence.criteria.CriteriaQuery;
5  import javax.persistence.criteria.Predicate;
6  import javax.persistence.criteria.Root;
7
8  final class TodoSpecifications {
9
10     private TodoSpecifications() {}
11
12     static Specification<Todo> hasTitle(String title) {
13         return new Specification<Todo>() {
14             @Override
15             public Predicate toPredicate(Root<Todo> root,
16                                         CriteriaQuery<?> query,
17                                         CriteriaBuilder cb) {
18
19                 //Create the query here.
20             }
21         }
22     }
23 }

```

```

1  import org.springframework.data.jpa.domain.Specification;
2
3  import javax.persistence.criteria.CriteriaBuilder;
4  import javax.persistence.criteria.CriteriaQuery;
5  import javax.persistence.criteria.Predicate;
6  import javax.persistence.criteria.Root;
7
8  import org.springframework.data.jpa.domain.Specification;
9
10 final class TodoSpecifications {
11
12     private TodoSpecifications() {}
13
14     static Specification<Todo> hasTitle(String title) {
15         return (root, query, cb) -> {
16             //Create query here
17         };
18     }
19 }

```

USE

```

1  Specification<Todo> spec = TodoSpecifications.hasTitle("foo");
2  long count = repository.count(spec);

```

```

1  Specification<Todo> spec = TodoSpecifications.hasTitle("foo");
2  List<Todo> todoEntries = repository.findAll(spec);

```

```

1  Specification<Todo> spec = TodoSpecifications.hasTitle("foo");
2  List<Todo> todoEntries = repository.findOne(spec);

```

```

1  Specification<Todo> specA = ...
2  Specification<Todo> specB = ...
3  List<Todo> searchResults = repository.findAll(
4      Specifications.where(specA).and(
5          Specifications.not(specB)
6      )
7  );

```

```

1  Specification<Todo> specA = ...
2  Specification<Todo> specB = ...
3  List<Todo> todoEntries = repository.findAll(
4      Specifications.where(specA).and(specB)
5  );

```



- **Predicate:** a conjunction or disjunction of restrictions.
- **Root:** A root type in the from clause. Query roots always reference entities.
- **CriteriaQuery:** defines functionality that is specific to top-level queries.
- **CriteriaBuilder:** construct criteria queries, compound selections, expressions, predicates, orderings.

```
1 import org.springframework.data.jpa.domain.Specification;
2
3 final class TodoSpecifications {
4
5     private TodoSpecifications() {}
6
7     static Specification<Todo> titleOrDescriptionContainsIgnoreCase(String searchTerm) {
8         return (root, query, cb) -> {
9             String containsLikePattern = getContainsLikePattern(searchTerm);
10             return cb.or(
11                 cb.like(cb.lower(root.<String>get(Todo_.title)), containsLikePattern),
12                 cb.like(cb.lower(root.<String>get(Todo_.description)), containsLikePattern)
13             );
14         };
15     }
16
17     private static String getContainsLikePattern(String searchTerm) {
18         if (searchTerm == null || searchTerm.isEmpty()) {
19             return "%";
20         }
21         else {
22             return "%" + searchTerm.toLowerCase() + "%";
23         }
24     }
25 }
```

```
@Transactional(readOnly = true)
@Override
public List<TodoDTO> findBySearchTerm(String searchTerm) {
    Specification<Todo> searchSpec = titleOrDescriptionContainsIgnoreCase(searchTerm);
    List<Todo> searchResults = repository.findAll(searchSpec);
    return TodoMapper.mapEntitiesIntoDTOs(searchResults);
}
```



If we need to create dynamic queries, we have to create these queries programmatically, and using the JPA Criteria API is one way to do it. The pros of using the JPA Criteria API are:

- It supports dynamic queries.
- If we have an existing application that uses the JPA Criteria API, it is easy to refactor it to use Spring Data JPA (if we want to).
- It is the standard way to create dynamic queries with the Java Persistence API (this doesn't necessarily matter, but sometimes it does matter).

JPA Criteria API has one big problem:

It is very hard to implement complex queries and even harder to read them.



By default, CRUD methods on repository instances are transactional. For read operations, the transaction configuration `readOnly` flag is set to `true`. All others are configured with a plain `@Transactional` so that default transaction configuration applies.

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

    @Override
    @Transactional(timeout = 10)
    public List<User> findAll();

    // Further query method declarations
}
```

```
@Transactional(readOnly = true)
public interface UserRepository extends JpaRepository<User, Long> {

    List<User> findByLastname(String lastname);

    @Modifying
    @Transactional
    @Query("delete from User u where u.active = false")
    void deleteInactiveUsers();
}
```



Service Components are the class file which contains `@Service` annotation. These class files are used to write business logic in a different layer, separated from `@RestController` class file.

```
1 @Repository
2 interface TodoRepository extends JpaRepository<Todo, Integer> {}
```

```
1 interface TodoService {
2
3     List<Todo> findAll()
4
5     Todo findById(Integer todoId)
6
7     Todo saveTodo(Todo todo)
8
9     Todo todo (Todo todo)
10
11     Todo deleteTodo(Integer todoId)
12 }
```

```
1 @Service
2 class TodoServiceImpl implements TodoService {
3
4     //...
5
6     @Override
7     List<Todo> findAll() {
8         todoRepository.findAll()
9     }
10
11     @Override
12     Todo findById(Integer todoId) {
13         todoRepository.findById todoId get()
14     }
15
16     @Override
17     Todo saveTodo(Todo todo){
18         todoRepository.save todo
19     }
20
21     @Override
22     Todo todo(Todo todo){
23         ... business logic
24     }
25
26     @Override
27     Todo deleteTodo(Integer todoId){
28         todoRepository.deleteById todoId
29     }
30 }
```



Another way to alter transactional behaviour is to use a facade or service implementation that (typically) covers more than one repository. Its purpose is to define transactional boundaries for non-CRUD operations.

```
@Service
class UserManagementImpl implements UserManagement {

    @Autowired
    private final UserRepository userRepository;

    @Autowired
    private final RoleRepository roleRepository;

    @Transactional
    public void addRoleToAllUsers(String roleName) {

        Role role = roleRepository.findByName(roleName);

        for (User user : userRepository.findAll()) {
            user.addRole(role);
            userRepository.save(user);
        }
    }
}
```

This example causes call to `addRoleToAllUsers(...)` to run inside a transaction (participating in an existing one or creating a new one if none are already running).

Note that you must use `@EnableTransactionManagement` explicitly to get annotation-based configuration of facades to work.

However, if we're using a Spring Boot project, and have a `spring-data-*` or `spring-tx` dependencies on the classpath, then transaction management will be enabled by default

