Spring Data JPA

Agenda

- What is Spring Data
- Configurations
- Domain/Entities
- Persistent Identity
- Identifier Generation
- Customizing the Entity Object
- Entity Relationships
- Entity Inheritance
- EntityManager & the Persistent Context
- Repository and Repository Hierarchy
- User Defined Repository
- Defining Query methods
- Pageable
- Query creation : Custom Queries and Named Queries
- Custom Interfaces
- Accessing Spring Data with rest
- Transactional
- Disadvantages

What and Why ?

Spring Data is a high level SpringSource project whose purpose is to unify and ease the access to different kinds of persistence stores, both relational database systems and NoSQL data stores.

Features

- Powerful repository and custom object-mapping abstractions
- Dynamic query derivation from repository method names
- Implementation domain base classes providing basic properties
- Support for transparent auditing
- Possibility to integrate custom repository code
- Advanced integration with Spring MVC controllers
- Several modules such as : Spring Data JPA, Spring Data MongoDB, Spring Data REST, Spring Data Cassandra etc.

Configuration

Dependencies :

```
compile('mysql:mysql-connector-java:5.1.6')
compile('org.springframework.boot:spring-boot-starter-data-jpa')
```

Setting :

```
spring:
 datasource:
   url: jdbc:mysql://localhost:3306/spring_jpa?autoReconnect=true&useUnicode=true&CharSet=UTF-
8&characterEncoding=UTF-8
   username: root
   password: igdefault
   driverClassName: com.mysql.jdbc.Driver
 jpa:
   hibernate.ddl-auto: create-drop
   show-sql: true
```


Domain/Entities

- An entity is a plain old java object (POJO)
- Requirements:
  - annotated with the javax.persistence.Entity annotation
  - public or protected, no-argument constructor
  - the class must not be declared final
  - no methods or persistent instance variables must be declared final
- Entities may extend both entity and non-entity classes
- Persistent instance variables must be declared private, protected

```
import javax.persistence.*;
@Entity
public class User {
   private String email;
   private String name;
}
```


Persistent Identity

- Each entity must have a unique object identifier (persistent identifier)
- Identifier (id) in entity = primary key in database
- Example :

```
import javax.persistence.*;
public class User {
 @Id
   private Long id;
}
```


Identity Generation

- Identifiers can be generated in the database by specifying @GeneratedValue on the identifier
- Four pre-defined generation strategies:
  - AUTO,
  - IDENTITY,
  - SEQUENCE,
  - TABLE
- Specifying strategy of AUTO indicates that the provider will choose a strategy
- Example:

```
import javax.persistence.*;
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
}
```


Customizing the Entity Object

- In most of the cases, the defaults are sufficient
- By default the table name corresponds to the unqualified name of the class
- Customization:

```
@Entity
@Table(name = "user")
public class User {}
```

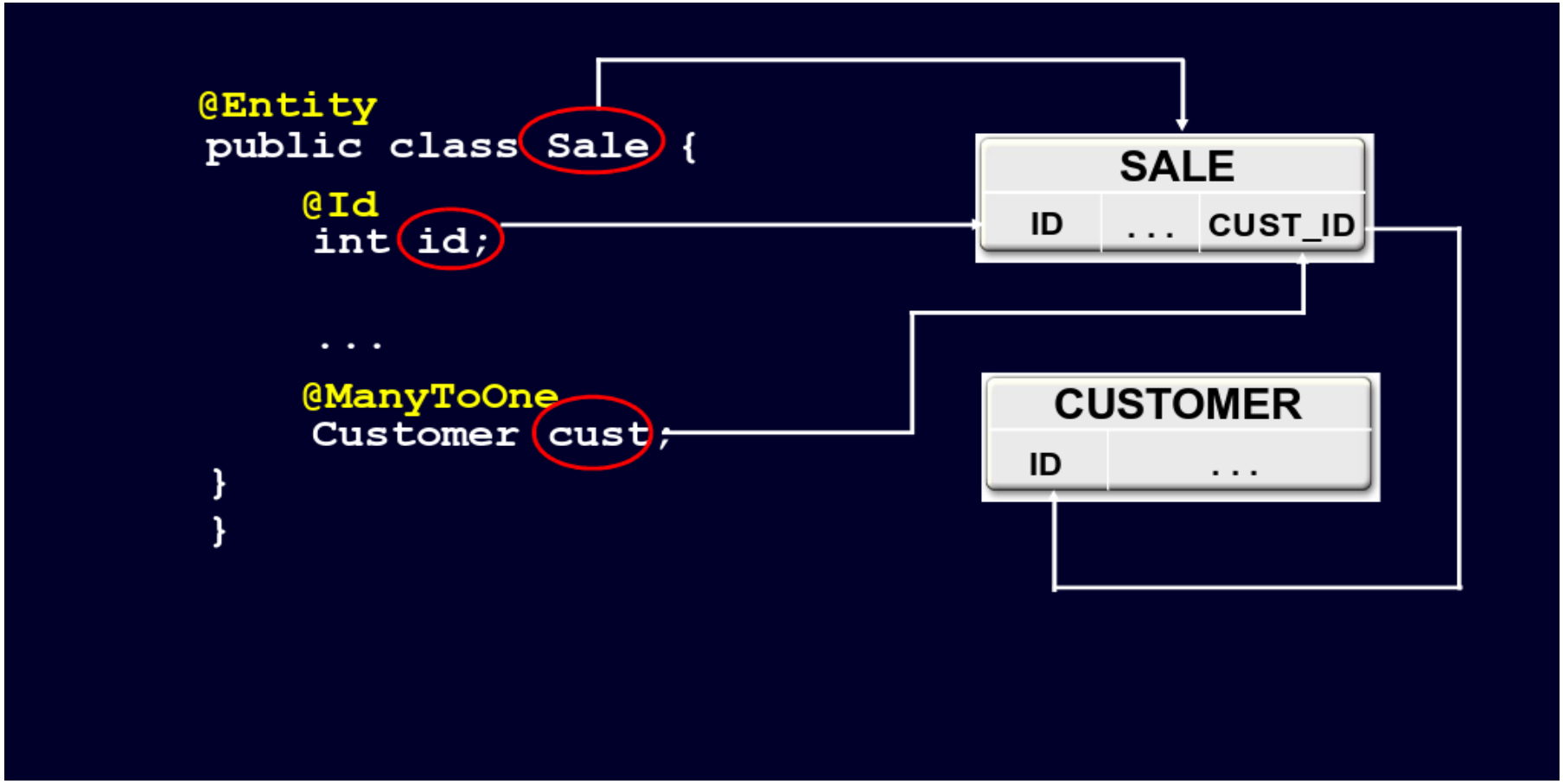
- The defaults of columns can be customized using the @Column annotation

```
@Column(nullable = true, unique = true)
private String email;
@Column(name = "full_name", nullable = false, length = 25)
private String name;
```

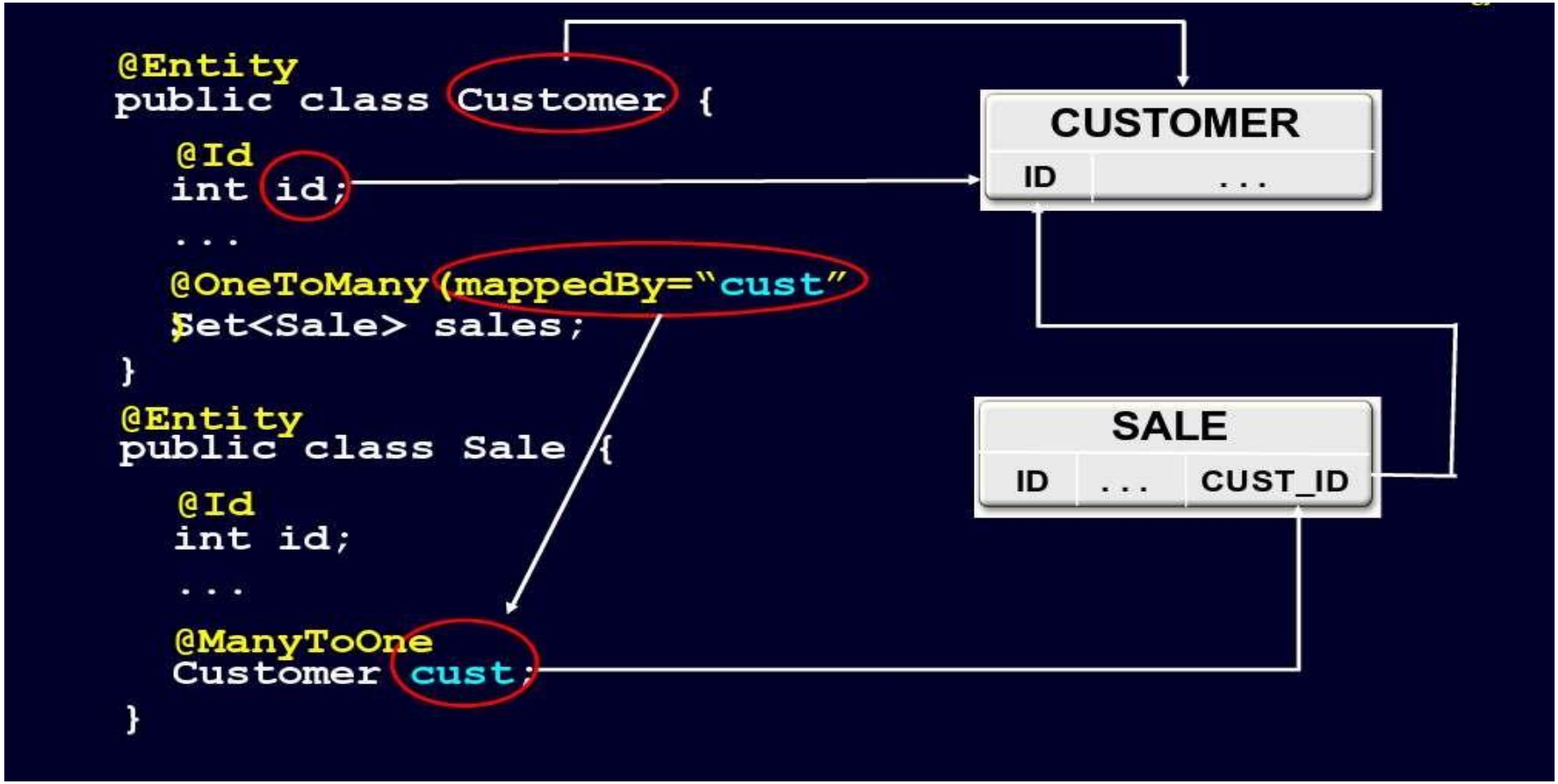

Entity Relationships

- There are four types of relationship multiplicities:
  - @OneToOne
  - @OneToMany
  - @ManyToOne
  - @ManyToMany
- The direction of a relationship can be:
  - **bidirectional** – owning side and inverse side
  - **unidirectional** – owning side only
- Supports cascading updates/deletes
- You can declare performance strategy to use with fetching related rows FetchType :
  LAZY, EAGER

ManyToOne Mapping



OneToMany Mapping



ManyToMany Mapping

```
@Entity
public class Customer {
   ...
   @JoinTable(
        name="CUSTOMER_SALE",
        joinColumns=@JoinColumn(
name="CUSTOMER_ID",referencedColumnName="customer_id"),
        inverseJoinColumns=@JoinColumn(
          name="SALE_ID", referencesColumnName="sale_id")
   Collection<Sale> sales;
}
```

```
@Entity
public class Sale {
    ...
   @ManyToMany(mappedBy="sales")
   Collection<Customer> customers;
}
```


Entity Inheritance

- Entities can inherit from other entities and from non-entities
- The @Inheritance annotation identifies a mapping strategy:
  - SINGLE_TABLE
  - JOINED
  - TABLE_PER_CLASS
- SINGLE_TABLE strategy - all classes in the hierarchy are mapped to a single table in the database
- Discriminator column - contains a value that identifies the subclass
- Discriminator type - {STRING, CHAR, INTEGER}
- Discriminator value - value entered into the discriminator column for each entity in a class hierarchy

Entity Inheritance : Example

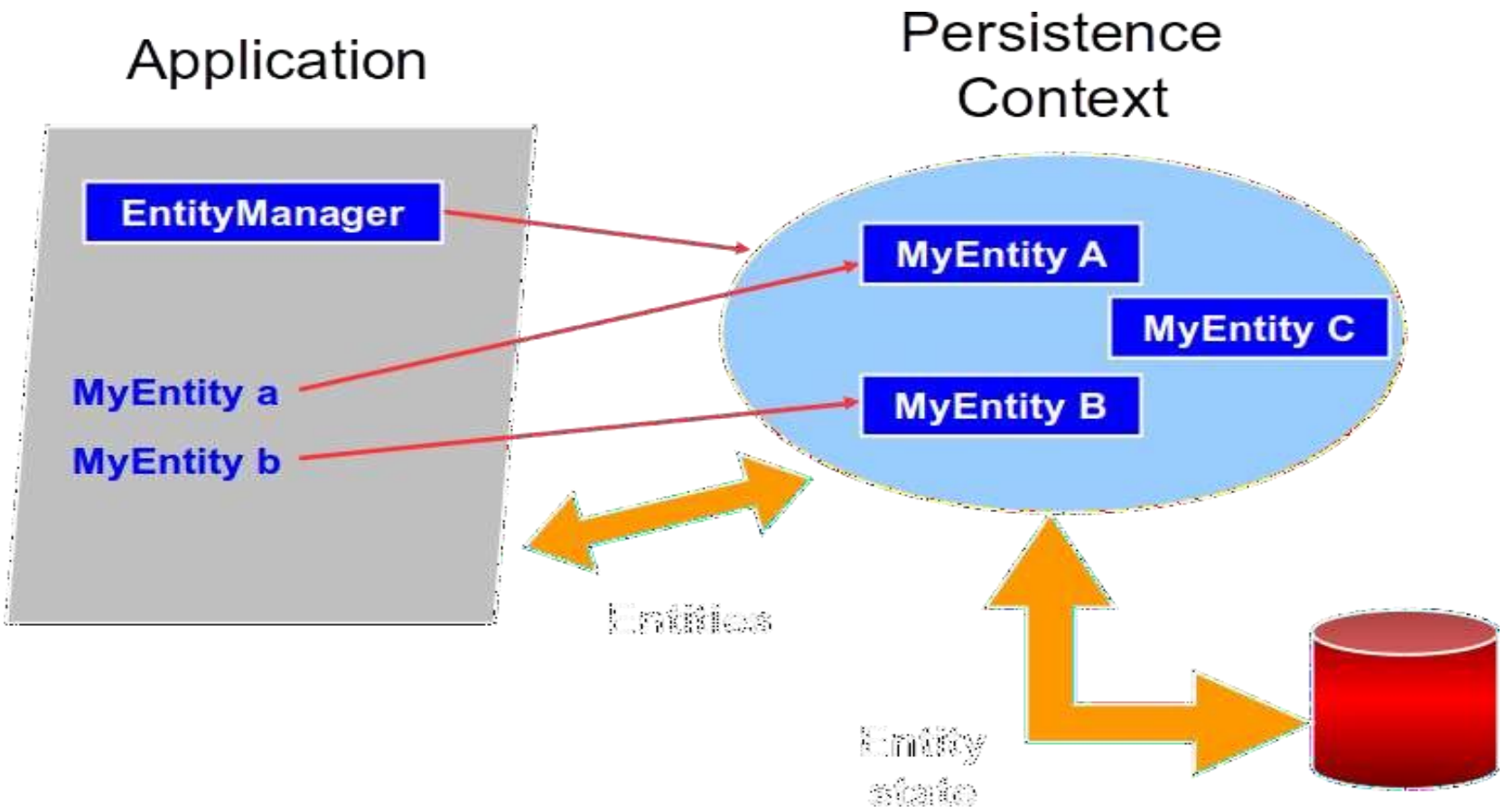
```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_T
ABLE)
@DiscriminatorColumn(name="DISC",
discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue(value="USER")
public class User { . . . }


@Entity
@DiscriminatorValue(value="PUS
ER")
public class PremiumUser extends User { . . . }
```


Managing Entities - JPA

- Entities are managed by the **entity manager**
- The entity manager is represented by javax.persistence.EntityManager instances
- Each EntityManager instance is associated with a **persistence context**
- A persistence context defines the scope under which particular entity instances are created, persisted, and removed
- A **persistence context** is a set of managed entity instances that exist in a particular
  data store
  - Entities keyed by their persistent identity
  - Only one entity with a given persistent identity may exist in the persistence context
  - Entities are added to the persistence context, but are not individually removable
    ("detached")
- Controlled and managed by **EntityManager**
  - Contents of persistence context change as a result of operations on EntityManager API

Persistence Context



Pains

- A lot of code in the persistent framework and the DAO.
- Duplicate code in concrete DAOs
- Pagination need to handle yourself, and integrated from MVC to persistent layer.
- If hybrid database (MySql + Mongo) are required for the system. It is not easy to have similar design concept in the Architecture.

Repository and Repository Hierarchy

- The goal of the repository abstraction of Spring Data is to reduce the effort to implement data access layers for various persistence stores significantly.
- The central marker interface
  - Repository<T, ID extends Serializable>

- **Hierarchy :**

```
Interface JpaRepository<T, ID>
        interface PagingAndSortingRepository<T, ID>
                interface CrudRepository<T, ID>
                        interface Repository<T, ID>
```


User Defined Repository

Spring Data Repository, you'll have three options:

- Using of a CRUD operations that implemented by the Spring Data infrastructure
- Defining of a query methods and
- Manually implementing your own custom repositories
- Example :

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


Defining Query methods

Query methods implemented in spring data repositories will be used for creating the dynamic queries.

```
public interface UserRepository extends JpaRepository<User, Long> {
   User findByEmail(String email);
    List<User> findAllByName(String name);
  }
```

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2

Pageable

```
Pageable pageable = new PageRequest(0, 10);
Sort.Order order1 = new Sort.Order(Sort.Direction.ASC, "id");
Sort.Order order2 = new Sort.Order(Sort.Direction.DESC, "name");
Sort sort = new Sort(order1, order2);
```

```
pageable = new PageRequest(0, 10, sort);
```

```
pageable = new PageRequest(0, 10, new Sort(Sort.Direction.DESC, "name"));
```


Query creation : Custom Queries

- @Query annotation is used to defining the custom queries in spring data.
- Supports JPQL and native SQL.
- @Param method arguments to bind query parameters.
- Supports SpEL expression.
- Like expression supported inside @Query annotation.
- @Query annotation, this will take the precedence over @NamedQuery
- Examples :

```
@Query("select u from User u where u.name=?1")
 User findByUserName(String name);
 @Query("select u from User u where u.name like%:name%")
User findByUserName(@Param("name") String name);
@Query(value = "select * from user where name=?1", nativeQuery = true)
User findByUserName(String name);
```


Query creation : Named Queries

- Named query are the static queries.
- The named queries are defined in the single place at entity class itself with each query has its unique name.
- @NamedQuery annotation can be applied only at the class level.
- Named queries have the global scope.
- If you have to define more than one named queries the use @NamedQueries
- All the named queries are validated at application start-up time and there is no failure at run time.
- Example :

```
@NamedQuery(name = "User.findByNameNamed", query = "SELECT u
FROM User u WHERE LOWER(u.name) = LOWER(?1)")
@Table(name = "user")
public class User {
…..
}
```


Custom Interfaces

- **Adding custom behavior to single repositories**

Create an interface which declares the custom methods :

```
public interface UserCustomRepository{
  public User customMethod();
 }
```

Implement custom repository :

```
public class UserRepositoryImpl implements UserCustomRepository{
  @Override
  public User customMethod(){
   }
}
```

Extend interface:

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


Custom Interfaces

- **Adding custom behavior to all repositories**

Creating a Base Repository Interface :

```
@NoRepositoryBean
public interface MyRepository<T, ID extends Serializable> extends JpaRepository<T, ID>{
   T sharedMethod(ID id);
}
```

Implementing the Base Repository Interface :

```
public class MyRepositoryImpl<T, ID extends Serializable> extends SimpleJpaRepository<T, ID>
implements MyRepository<T, ID>{
}
```


Custom Interfaces

- **Adding custom behavior to all repositories**

Creating a Custom RepositoryFactoryBean :

```
public class MyRepositoryFactoryBean extends JpaRepositoryFactoryBean{
}
```

Configuring Spring Data JPA :

```
@EnableJpaRepositories(repositoryFactoryBeanClass = MyRepositoryFactoryBean.class)
```


Accessing Spring Data with REST

- Add dependency :
  compile(**"org.springframework.boot:spring-boot-starter-data-rest"**)

- Annotate repository :
  @RepositoryRestResource()

Transactions

- CRUD methods on repository instances are transactional by default.
- Use @Transactional annotation in repository.
  @Transactional(timeout = 10)
  @Transactional(readOnly = true)

Disadvantages

Methods name are very long in the complicated structure.
No support for aggregation queries.

References

- http://projects.spring.io/spring-data/
- https://dzone.com/articles/easier-jpa-spring-data-jpa
- http://docs.spring.io/spring-data/jpa/docs/current/reference/html/