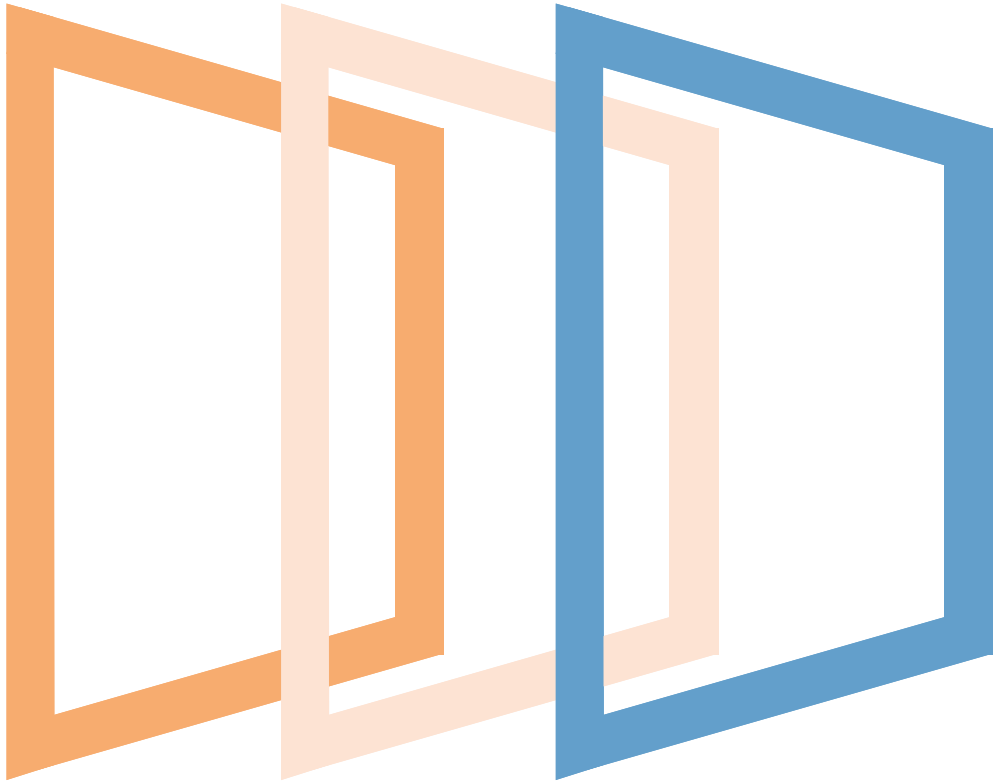
Microservices with Spring



June 2019

minsait

An Indra company

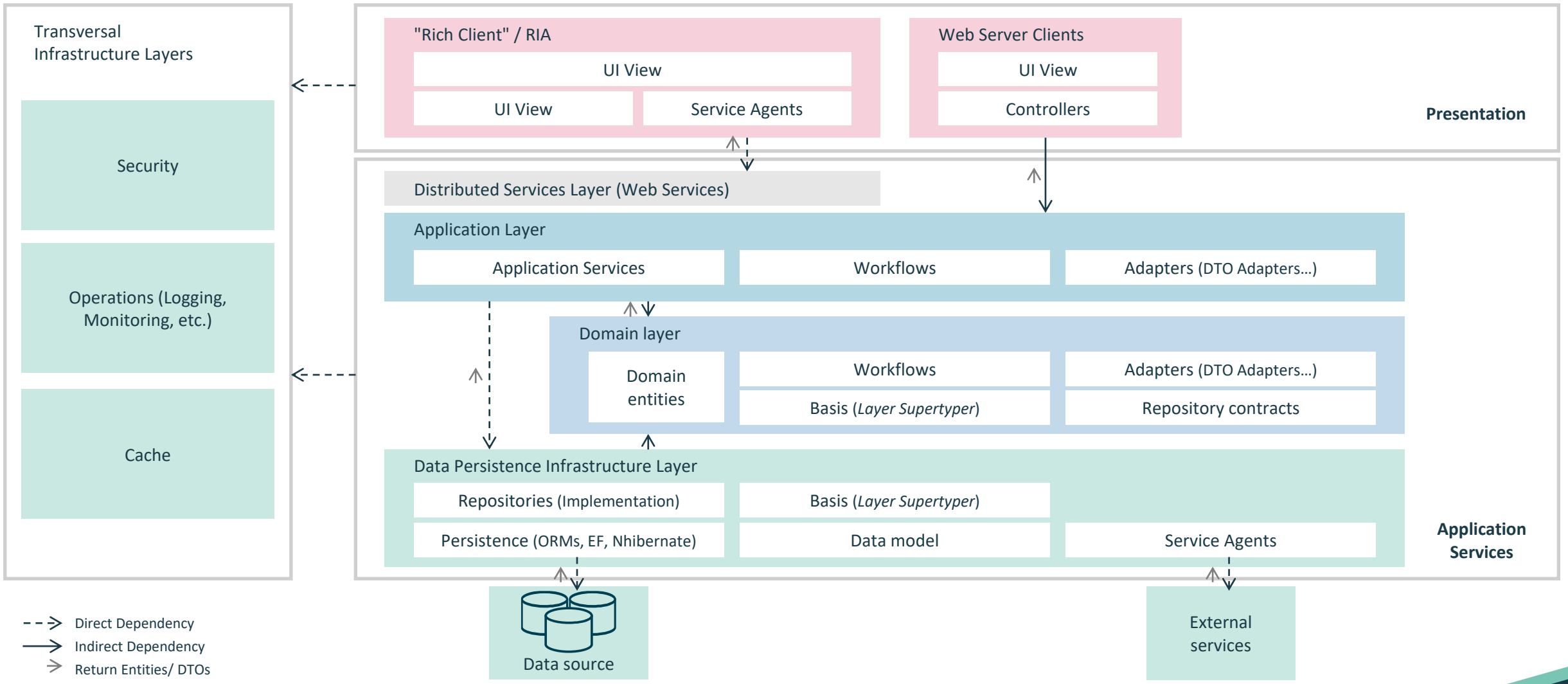
Índex

Overview and links

01

http://spring.io

N-Layered Architecture with Domain Orientation



Links

Microservices

https://martinfowler.com/articles/microservices.html

https://microservices.io/

Spring

https://spring.io/projects

Spring Core

https://docs.spring.io/spring-framework/docs/current/reference/html/

Spring Data

https://docs.spring.io/spring-data/jpa/docs/2.1.5.RELEASE/reference/html/

Spring MVC

https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html

Spring HATEOAS

https://docs.spring.io/spring-hateoas/docs/0.25.1.RELEASE/reference/html

Spring Data REST

https://docs.spring.io/spring-data/rest/docs/3.1.5.RELEASE/reference/html/

Example:

https://github.com/spring-projects/spring-data-examples

https://github.com/spring-projects/spring-data-rest-webmvc

https://github.com/spring-projects/spring-hateoas-examples

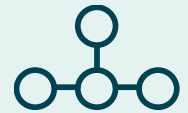
https://github.com/spring-projects/spring-integration-samples

Spring with Spring Boot

02

http://spring.io

Spring



- Initially it was an example made for the book “J2EE design and development” by Rod Johnson in 2003, which defended alternatives to the “official vision” of JavaEE application based on EJBs.
- It is currently an open-source framework that facilitates the development of Java JEE & JSE applications (it is not limited to Web applications, nor to Java, they can also be .NET, Silverlight, Windows Phone, etc.)
- It provides a container in charge of managing the life cycle of objects (beans) so that developers can focus on business logic. Allows integration with different frameworks.
- It emerges as an alternative to EJB’s
- It is currently a complete framework composed of multiple modules/projects that covers all layers of the application, with dozens of developers and thousands of downloads per day.
  - MVC.
  - Business (where it originally started).
  - Data access-

Features

- Lightweight
  - Not about the number of classes, but about the minimum impact of integrating Spring.
- Non-intrusive
  - In general, the objects that are programmed do not have dependencies on Spring-specific classes.
- Resilient
  - Although Spring provides functionality to handle the different layers of the application (view, business logic, data access), it is not necessary to use it for everything. It offers the possibility of using it in the layer or layers that we want.
- Cross-platform
  - Written in Java, runs on JVM.

Proyectos

Spring Framework

Provides core support for dependency injection, transaction management, web apps, data access, messaging and more.

Spring Data

Provides a consistent approach to data access – relational, non-relational, map-reduce, and beyond.

Spring Boot

Takes an opinionated view of building Spring application and gets you up and running as quickly as possible.

Spring Security

Protects your application with comprehensive and extensible authentication and authorization support.

Spring Social

Easily connects your applications with third-party APIs such as Facebook, Twitter, Linkedin, and more.

Spring Cloud Data Flow

A cloud native programming and operating model for composable data microservices on structured platform.

Spring Cloud

Provides a set of tools for common patterns in distributed systems. Useful for building and deploying microservices.

Spring for Android

Provides key Spring components for use in developing Android applications.

Required modules

Spring Framework

- Spring Core
  - IoC (inversion of control) container - dependency injector.
- Spring MVC
  - MVC-based framework for web applications and REST services.

Spring Data

- Simplifies data access: JPA, relational/ NoSQL databases, cloud.

Spring Boot

- Simplifies Spring development: get started quickly with less coding.

Spring Boot

- Spring Boot is a tool that was born with the purpose of simplifying the development of applications based on the Spring Core framework: the developer only focuses on the development of the solution, completely forgetting about the complex configuration that Spring Core currently has for be able to function
  - Configuration: Spring Boot has a complex module that autoconfigures all the aspects of our application to be able to simply run the application, without having to define anything at all.
  - Dependency solver: With Spring Boot, we only have to determine what type of project we will be using, and Spring Boot will be in charge of resolving all the libraries/dependencies for the application to work.
  - Deployment: Spring Boot can be run as a Stand-alone application, but it is also possible to run web applications, since applications can be deployed using an integrated web server, such as Tomcat, Jetty or Undertow.
  - Metrics: By default, Spring Boot has services that allow us to check the health status of the application, letting is know whether the application is on or off, how much memory is used and available, the number and details of the Bean's created by the application, controls for turning on and off, etc.
  - Extensible: Spring Boot allows the creation of plugins, which help the Free Software community create new modules that make development even easier.
  - Productivity: Developer productivity tools like LiveReload and Auto Restart work in our favorite IDE: Spring Tool Suite, IntelliJ IDEA, and NetBeans.

Utilities to start from scratch. Environment

Download Hibernate:

- http://hibernate.org/orm/downloads/

Download and install JDK:

- http://www.oracle.com/technetwork/java/javase/downloads/jdk8-_downloads-2133151.html

Download and uncompress Eclipse:

- https://www.eclipse.org/downloads/

Add HibernateTools to Eclipse:

- Help > Eclipse Marketplace: JBoss Tools

Create a User Library for Hibernate:

- Window > Preferences > Java > Build Path > User Libraries > New
- Add External JARs: \lib\required

Download and register the JDBC driver definition:

- Window > Preferences > Data Management > Connectivity > Driver Definition > Add

Utilities to start from scratch. Installation

https://spring.io/tools

Spring Tool Suite

- Free IDE, Eclipse customization.

Plug-in for Eclipse (VSCode, Atom)

- Help .Eclipse Marketplace …
  - Spring Tools 4 for Spring Boot.

Utilities to start from scratch. Create project

From web:

- https://start.spring.io/
- Unzip in the workspace.
- Import--Maven --Existing Maven Project

From Eclipse:

- New Project --Sprint Boot --Spring Started Project

Dependencies:

- Web.
- JPA.
- JDBC (or customized Project).

Utilities to start from scratch. Optional dependencies

XML serialization to client

```
<dependency>
<groupId>com.fasterxml.jackson.dataformat</group  Id>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

Utilities to start from scratch. Application

```
import org.springframework.boot.CommandLineRunner;  import
org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

	public static void main(String[] args) {
		SpringApplication.run(ApiHrApplication.class, args);
	}

	@Override
	public void run(String... args) throws Exception {
		// Opcional: Process arguments once SpringBoot has booted

	}

}
```

Utilities to start from scratch. Configuración

@Configuration: Indicates that this is a class used to configure the Spring container.

@ComponentScan: Scans the packages of our project in search of the components that we have created – these are the classes that use the following annotations:

@EnableAutoConfiguration: Enables automatic configuration. This tool analyzes the classpath and the application.properties file to configure our application based on the libraries and configuration values found. For example: when the H2 database engine is found, the application is configured to use this data engine. When it finds Thymeleaf it will create the necessary beans to use this template engine to generate the views of our web application.

@SpringBootApplication: It is the equivalent of using the annotations:

@Configuration, @EnableAutoConfiguration y @ComponentScan

Utilities to start from scratch. Configuration

- Edit src/main/resources/application.properties:
  # Oracle settings
  spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
  spring.datasource.username=hr
  spring.datasource.password=hr

  spring.datasource.driver-class=oracle.jdbc.driver.OracleDriver

  # MySQL settings
  spring.datasource.url=jdbc:mysql://localhost:3306/sakila spring.datasource.username=root
  spring.datasource.password=root
  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

  logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n logging.level.org.hibernate.SQL=debug

  server.port=8080

- Repeat with src/test/resources/application.properties

Utilities to start from scratch. Oracle Driver with Maven

- http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html
- Maven installation:
  - Download and unzip (https://maven.apache.org)
  - Add this to the PATH: C:\Program Files\apache-maven\bin
  - Check in the command console: mvn -v
- Download the Oracle JDBC Driver (ojdbc6.jar):
  - https://www.oracle.com/technetwork/apps-tech/jdbc-112010-090769.html
- Install the ojdbc artifact in the local Maven repository:
  - mvn install:install-file -Dfile=Path/to/your/ojdbc6.jar -DgroupId=com.oracle - DartifactId=ojdbc6 -Dversion=11.2.0 -Dpackaging=jar
- In the pom.xml file:

```
<dependency>
        <groupId>com.oracle</groupId>
        <artifactId>ojdbc6</artifactId>
        <version>11.2.0</version>
  </dependency>
```

Utilities to start from scratch. Configuración del proxy: Maven

- Create setting.xml file, or edit %MAVEN_ROOT%/conf/setting.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository>C:\local\directory\.m2\repository</localRepository>
  <proxies>
    <proxy>
      <id>optional</id>
      <active>true</active>
      <protocol>http</protocol>
      <username>usuario</username>
      <password>contraseña</password>
      <host>proxy.dominion.com</host>
      <port>8080</port>
      <nonProxyHosts>local.net|some.host.com</nonProxyHosts>
    </proxy>
  </proxies>
</settings>
```

- Reference it in Window → Preferences → Maven → User setting → User setting , browse…, select newly created file, accept, update setting, apply and close.

Utilities to start from scratch. MySQL Installation

- Download and install:
  - https://mariadb.org/download/
- Include in the [mysqld] section of

  %MYSQL_ROOT%/data/my.ini
  - default_time_zone='+01:00'
- Download sample databases:
  - https://dev.mysql.com/doc/index-other.html
- Install sample databases:
  - mysql -u root -p < employees.sql
  - mysql -u root -p < sakila-schema.sql
  - mysql -u root -p < sakila-data.sql

IoC with Spring Core

03

https://docs.spring.io/spring-framework/docs/current/reference/html/

Inversion of Control

- Inversion of control (IoC) is a concept along with some programming techniques:
  - In which the execution flow of a program is reversed compared to traditional programming methods,
  - In which the interaction is expressed imperatively by calling procedures or functions.
- Traditionally, the programmer specifies the sequence of decisions and procedures that can occur during the life cycle of a program through function calls.
- Instead, inversion of control specifies desired responses to specific events or data requests, leaving some kind of external entity or architecture to perform the required control actions that have to happen, in the necessary order and for the required set of events

Dependency Injection

- Dependency Injection (DI) is an object-oriented architecture pattern, in which objects are injected into a class instead of the class itself creating the object. It basically recommends that the dependencies of a class are not created from the object itself, but instead are configured from outside the class.
- Dependency Injection (DI) comes from the more general design pattern that is Inversion of Control (IoC).
- Applying this pattern makes the classes independent of each other and increases the reusability and extensibility of the application, as well as facilitating their unit tests.
- From the Java point of view, a DI-based layout can be implemented using the standard language, since a class can read the dependencies of another class via the Java Reflection API and create an instance of that class by injecting its dependencies into it.

Injection

- Dependency Injection provides the following:
  - The code is cleaner.
  - Decoupling is more efficient, since the objects do not need to know where their dependencies are or what they are.
  - Ease in unit and integration testing.

Introduction

- Spring provides a container in charge of dependency injection (Spring Core Container).
- This container allows us inject some objects over others.
- To do this, the objects must simply be JavaBeans.
- Dependency injection will be either by constructor or by setter methods.
- The configuration can be done either by Java annotations or by means of an XML file (XMLBeanFactory).
- For the management of the objects, the class (BeanFactory) must be used.
- All objects will be created as singletons unless otherwise specified.

Dependencies module

- Create the applicationContext.xml configuration file and save it in the src/META-INF directory:

```
<beans xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemalocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-
       aop-2.5.xsd  http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context-2.5.xsd">

      <context:component-scan base-package="es.miEspacio.ioc.services">

      </context:component-scan>

</beans>
```

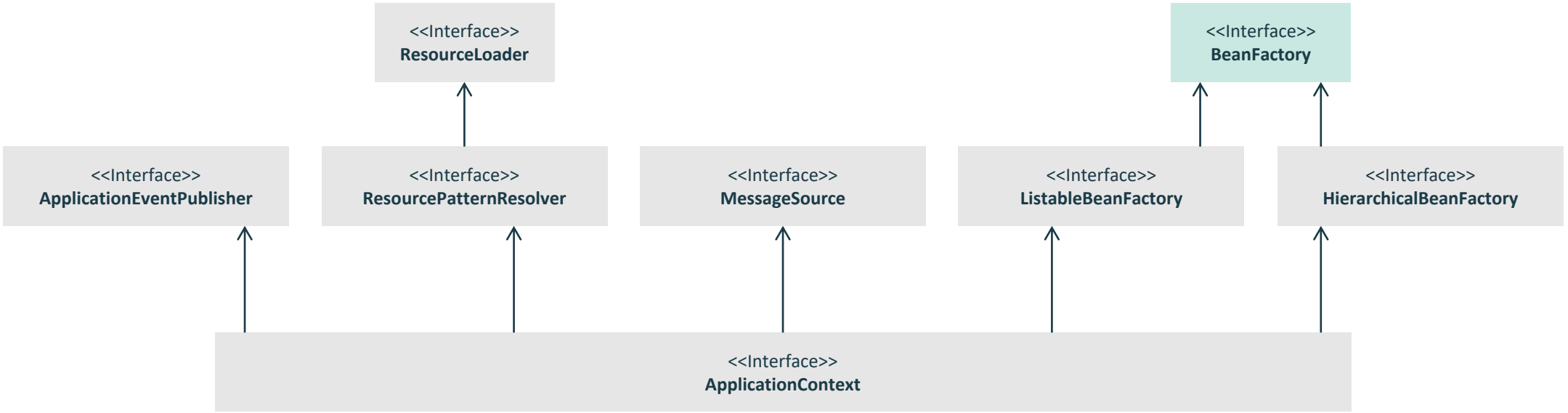
Beans

- Beans correspond to the real objects that make up the application and that need to be injectable: service layer objects, data access objects (DAO), presentation objects (such as Action instances of Struts) , infrastructure objects (such as Hibernate SessionFactories, JMS Queues), etc.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- services -->
    <bean id="petStore" class="com.samples.PetStoreServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>
    <!-- more bean definitions for services go here -->
</beans>
```

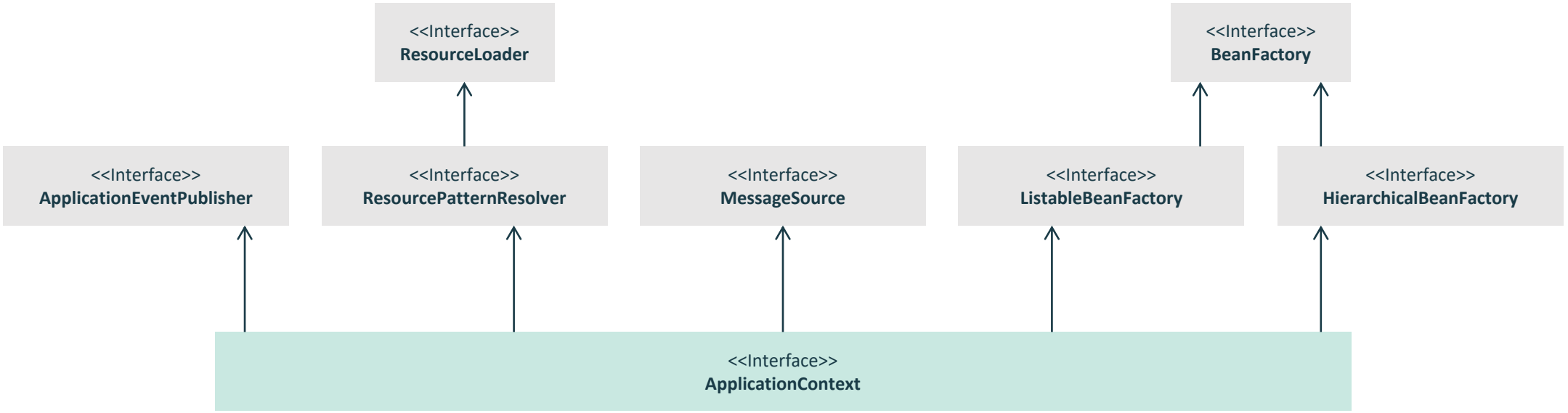
Bean factory

- The Spring container will be called Bean Factory.
- Any Bean Factory allows the configuration and binding of objects through dependency injection.
- This Bean Factory also allows a life cycle management of the beans instantiated in it.
- All Spring containers (Bean Factory) implement the BeanFactory interface and some sub-interfaces to extend functionalities



Application Context

- Spring also supports something more advanced, a “bean factory”, called the application context.
- Application Context is a Bean Factory specification that implements the ApplicationContext interface.
- In general, anything a Bean Factory can do,
  - an application context can also do.



Using Dependency Injection

- Create an injector starting from a dependency module.
- Request the injector for the instances to resolve the dependencies.

```
import org.springframework.beans.factory.BeanFactory;
import  org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public static void main(String[] args)  {
        ApplicationContext context = new ClassPathXmlApplicationContext("META-
        INF/applicationContext.xml");
        BeanFactory factory = context;
        Client client = (Client )factory.getBean("ID_Cliente");
        client.go();
}
```

- Sample:
  - This is a service...

IoC annotations

Self-discovery
- @Scope
- @Component
- @Repository
- @Service
- @Controller

Customization
- @Configuration
- @Bean

Injection
- @Autowire (@Inject)
- @Qualifier (@Named)
- @Value
- @PropertySource
- @Required
- @Resource

Other
- @PostConstruct
- @PreDestroy

Stereotypes

- Spring defines a set of core annotations that categorize each one of the components, associating them with a specific responsibility.
  - @Component: It is the general stereotype and allows to annotate a bean so that Spring considers it one of its objects.
  - @Repository: It is the stereotype in charge of registering a bean so that it implements the repository pattern that is responsible for storing data in a database or repository of information as needed. By marking the bean with this annotation, Spring provides traversal services such as exception type conversion.
  - @Service: This stereotype is responsible for managing the most important business operations at the application level and brings together calls to several repositories simultaneously. Its fundamental task is that of aggregator.
  - @Controller The last of the stereotypes, which is the one that performs the tasks of controller and communication management between the user and the application. To do this, it is usually supported by some template engine or tag library that facilitates the creation of pages.
  - @RestController: It is a controller specialization containing @Controller and @ResponseBody annotations (writes directly to the response body instead of the view).

Scope

- An important aspect of the Beans life cycle is whether the container will create a single instance or as many as scopes are needed.
  - prototype: Does not reuse instances, always generates a new one.
- instance. @Scope("prototype")
  - singleton: (Default) Single instance for the entire Spring IoC container. @Scope("singleton") @Singleton
  - Additionally, in the context of a Spring Web ApplicationContext: @RequestScope @SessionScope @ApplicationScope

    request: Unique instance for the lifecycle of a single HTTP request. Each HTTP request has its own unique instance.

    session: Unique instance for the lifecycle of each HTTP Session.

    application: Unique instance for the lifecycle of a ServletContext.

    websocket: Unique instance for the lifecycle of a WebSocket.

Injection

- The injection is done with the @Autowire annotation:
  - Under attributes:

    ```
    @Autowire
    private MyBeans myBeans;
    ```

  - Under properties (setter):

    ```
    @Autowire
    public void setMyBeans(MyBeans value) { … }
    ```

  - Under constructors.
- By default the injection is mandatory. It can be marked as optional in which case, should the Bean be not found, a null will be injected:

  ```
  @Autowire(required=false) private MyBeans myBeans;
  ```

- @Autowire can be completed with the @Lazy annotation to inject a lazy resolution proxy.

Access to property files

- Localization (.properties, .yml, .xml file):
  - Default: src/main/resources/application.properties
  - In the resources folder src/main/resources:

    @PropertySource("classpath:my.properties")
  - In a local file:

    @PropertySource("file://c:/cng/my.properties")
  - In a URL:

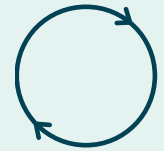
    @PropertySource("http://myserver/application.properties")
- Direct access:

  @Value("${spring.datasource.username}") private String name;
- Access through environment:

  @Autowired private Environment env;

  env.getProperty("spring.datasource.username")

Lifecycle



- With injection, the process of creating and destroying bean instances is managed by the container. ▪To be able to intervene in the cycle to control the
- In order to intervene in the cycle to control the creation and destruction of instances, you can:
  - Implement the InitializingBean and DisposableBean callback interfaces.
  - Override the init() and destroy() methods.
  - Annotate the methods with @PostConstruct and @PreDestroy.
- These mechanisms can be combined to control a given bean.

Configuration by code

- You have to create one (or several) class annotated with @Configuration , that will contain a method for each class/interface (without stereotype) that you want to treat as an injectable Bean.
- The method will be annotated with @Bean . It should be named as the class, in CamelCase notation, and return the already created instance from the class type. Additionally, it can be annotated with @Scope and @Qualifier.
  - public class MyBean { … }

    @Configuration
  - public class MyConfig {

    @Bean

    @Scope("prototype")
  - public MyBean myBean() { …]

Double inheritance

- The interface with the desired functionality is created:

```
public interface Service {
        public void go();
}
```

- The interface is implemented in a class (by convention, the Impl suffix is used):

```
import org.springframework.stereotype.Service;
@Service
@Singleton
public class ServiceImpl implements Service {
        public void go() {
                System.out.println("Este es un servicio...");
        }
}
```

Client

```
import org.springframework.beans.factory.annotation.Autowired;  import
org.springframework.stereotype.Service;


@Service("ID_Cliente")
public class Client {
    private final Service service;

    @Autowired
    public void setService(Service service){
            this.service = service;
    }

    public void go(){
            service.go();
    }
}
```

@Autowired states that parameters must be resolved by DI.

JSR-330 standard annotations

- As of Spring 3.0, Spring offers support for the JSR-330 standard annotations (dependency injection). Those annotations are scanned in the same way as Spring annotations.
- When working with standard annotations, bear in mind that some important features are not available.

Spring annotation	JSR-330 standard annotation (javax.inject.*) JSR-330
@Autowired	@Inject
@Component	@Named / @ManagedBean
@Scope("singleton")	@Singleton
@Qualifier	@Qualifier / @Named
@Value	-
@Required	-
@Lazy	-

Access to data with Spring Data

04

Spring Data

- Spring Framework already provided support for JDBC, Hibernate, JPA or JDO, simplifying the implementation of the data access layer, unifying the configuration and creating a common exception hierarchy for all of them.

- Spring Data is a SpringSource project (subprojects) whose purpose is to unify and facilitate access to different types of persistence technologies, both relational and NoSQL-type databases.

- Spring Data comes to cover the necessary support for different NoSQL database technologies, integrating them with traditional data access technologies, thus simplifying the work when creating specific implementations.

- With each type of persistence technology, DAOs (Data Access Objects) offer the typical CRUD functionalities for own domain objects, search methods, sorting and pagination. Spring Data provides generic interfaces for these aspects (CrudRepository, PagingAndSortingRepository) and specific implementations for each type of persistence technology.

Models: Entities

- An entity is a type of class dedicated to representing a persistent domain model that:
  - Must be public (cannot be nested or final or have final members).
  - Must have a public constructor without any arguments.
  - For each property that we want to persist, must have a get/setassociated method.
  - Must have a primary key.
  - Should override the equals and hashCode methods.
  - Should implement Serializable interface to use remotely.

JPA annotations

Annotation	Description
@Entity	▪ Applies to the class. ▪ Indicates that this Java class is an entity to be persisted.
@Table(name="Tabla")	▪ Applies to the class and indicates the name of the database table where the class will be persisted. ▪ Is optional if the class name matches the table name.
@Id	▪ Applies to a Java property and indicates that this attribute is the primary key.
@Column(name="Id")	▪ Applies to a Java property and indicates the name of the database column in which the property will be persisted. ▪ Is optional if the Java property name matches the database column name.
@Column(…)	▪ name: its name. ▪ length: its length. ▪ precision: total number of digits. ▪ scale: number of decimal digits. ▪ unique: unique value constraint. ▪ nullable: mandatory value constraint. ▪ insertable: whether it is insertable. ▪ updatable: whether it is modifiable.
@Transient	▪ Applies to a Java property and indicates that this attribute is not persistent.

Associations

- One to One (Unidirectional):
  - In the strong entity, the property is recorded with the reference of the entity.
  - @OneToOne(cascade=CascadeType.ALL):
    - This annotation indicates the one-to-one relationship of the 2 tables.
  - @PrimaryKeyJoinColumn:
    - We indicate that the relationship between the two tables is made through the primary key.
- One to one (Bidirectional):
  - The two entities have a property with the reference to the other entity.

Associations

- One to Many:
  - In One:
    - Has a collection-type property that contains the references of the Many entities:
      - List: Sorted with repeats.
      - Set: Unsorted without repeats.
    - @OneToMany(mappedBy="propEnMuchos",cascade= CascadeType.ALL)
      - mappedBy: will contain the name of the property in the Many entity with the reference to the One entity.
    - @IndexColumn (name="idx")
      - Optional. Name of the column in the Many table for the sorting within the List.
  - In Many:
    - Has a property with the reference of entity One.
    - @ManyToOne
      - This annotation indicates the relationship of Many to One.
    - @JoinColumn (name="idFK")
      - We will indicate the name of the column that in the Many table contains the foreign key to the One table.

Associations

- Many to Many (Unidirectional):
  - Has a collection-type property that contains the references of the Many entities.
  - @ManyToMany(cascade=CascadeType.ALL):
    - This annotation indicates the many-to-many relationship of the 2 tables.
- Many to Many (Bidirectional):
  - The second entity also has a collection-type property that contains the references of the Many entities.
  - @ManyToMany(mappedBy="propEnOtroMuchos"):
    - mappedBy: Property with the collection on the other entity to preserve synchronicity between both sides.

Cascade

- The cascade attribute is used in association mappings to indicate when the action in one instance should be propagated to related instances through the association.
- CascadeType enumeration:
  - ALL = {PERSIST, MERGE, REMOVE, REFRESH, DETACH}
  - DETACH (Separar)
  - MERGE (Modificar)
  - PERSIST (Crear)
  - REFRESH (Releer)
  - REMOVE (Borrar)
  - NONE
- Accepts multiple values:
  - @OneToMany(mappedBy="profesor", cascade={CascadeType.PERSIST, CascadeType.MERGE})

Inheritance mapping

- Table by class hierarchy:
  - Parent:
    - @Table("Account")
    - @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
    - @DiscriminatorColumn(name="PAYMENT_TYPE")
  - Child:
    - @DiscriminatorValue(value = "Debit")
- Table by subclasses:
  - Parent :
    - @Table("Account")
    - @Inheritance(strategy = InheritanceType.JOINED)
  - Child:
    - @Table("DebitAccount")
    - @PrimaryKeyJoinColumn(name = "account_id")
- Table by specific class:
  - Parent :
    - @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
  - Child:
    - @Table("DebitAccount")

Repository

- A repository is a class that acts as a mediator between the application's domain and the data that gives it persistence.
- Its goal is to abstract and encapsulate all accesses to the data source.
- It completely hides the implementation details of the data source from its clients.
- The interface exposed by the repository does not change, even if the underlying data source implementation changes (different storage schemes).
- A repository is created for each domain entity that offers CRUD (Create-Read-Update-Delete), search, sort, and paging methods.

Repository

- With the support of Spring Data, the repetitive task of creating the concrete DAO implementations for entities, is simplified, because we only need one interface that extends one of the following interfaces:
  - CrudRepository<T,ID>
    - count(), delete(T entity), deleteAll(), deleteAll(Iterable<? extends T> entities), deleteById(ID id), existsById(ID id), findAll(), findAllById(Iterable<ID> ids), findById(ID id), save(S entity), saveAll(Iterable<S> entities)
  - PagingAndSortingRepository<T,ID>
    - findAll(Pageable pageable), findAll(Sort sort)
  - JpaRepository<T,ID>
    - deleteAllInBatch, deleteInBatch, flush, getOne, saveAll, saveAndFlush
- In the injection process, Spring implements the interface before injecting it:
  - public interface ProfesorRepository extends JpaRepository<Profesor, Long> {}
  - @Autowired
  - private ProfesorRepository repository;

Repository

- The interface can be extended with new methods that will be implemented by Spring:
  - By deriving the query from the method name directly.
  - By using a manually-defined query.
- The implementation will be done by decoding the name of the method. It has a specific syntax to create such names:
  - List<Profesor> findByNombreStartingWiths(String nombre);
  - List<Profesor> findByApellido1AndApellido2OrderByEdadDesc(String apellido1, String apellido2);
  - List<Profesor> findByTipoIn(Collection<Integer> tipos);
  - int deleteByEdadGreaterThan(int valor);

Repository

- Derived query prefix:
  - find (read, query, get), count, delete
- Optionally, limit the query results:
  - Distinct, TopNumFilas y FirstNumFilas
- Property expression: ByProperty
  - Operator (Between, LessThan, GreaterThan, Like, …); default: equal.
  - Several can be concatenated with And and Or.
  - Optionally supports the IgnoreCase and AllIgnoreCase flags.
- Optionally, OrderByPropertyAsc to order,
  - Asc can be replaced by Desc. Supports several sorting expressions.
- Parameters:
  - One parameter for each operator that requires a value, and it must be of the appropriate type.
- Optional parameters:
  - Pageable, Sort

Repository

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, findByFirstna meIs, 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

Repository

Keyword	Sample	JPQL snippet
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 (linked parameter, with % appended)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (linked parameter, with % prepended)

Repository

Keyword	Sample	JPQL snippet
Containing	findByFirstnameContaining	… where x.firstname like ?1 (linked parameter between two %)
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)

Repository

- Return value of synchronous queries:
  - Find, read, query, get:
    - List<Entity>
    - Stream< Entity >
    - Optional<T>
  - Count, delete:
    - Long
- Return value of asynchronous queries (must be annotated with @Async):
  - Future<Entity>
  - CompletableFuture<Entity>
  - ListenableFuture<Entity>

Repository

- Using JPQL queries:

```
@Query("from Professor p where p.age > 67")
List<Professor> findRetired();

@Modifying
@Query("delete from Professor p where p.age > 67")
List<Professor> deleteRetired();
```

- Using native SQL queries
  - @Query("select * from Professors p where p.age between ?1 and ?2", nativeQuery=true)
    List<Professor> findActive(int inicial, int final);

DTO

- A Data Transfer Object (DTO) is an object that defines how data will be sent over the network.
- Its purpose is to:
  - Decouple the service tier from the database tier.
  - Remove circular references.
  - Hide certain properties that clients should not see.
  - Override some of the properties in order to reduce the size of the payload.
  - Remove the formatting of object graphs that contain nested objects, to make them more convenient for clients.
  - Avoid “excess” and vulnerabilities by publication.

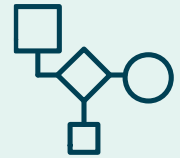
Lombok

- https://projectlombok.org/
- Java classes have a lot of code that repeats itself over and over again: constructors, equals, getters and setters. Methods that are defined once said class has specified its properties, and except for minor adjustments, they will always be same old, same old.
- Project Lombok is a Java library that automatically connects to the editor and creates tools that automate the Java writing.
- Using simple annotations, you'll never have to write another get or equals method again.

```
@Data @AllArgsConstructor @NoArgsConstructor public class MyDTO {

        private long id;

        private String name;

}
```

- The @Value annotation (not to be confused with Spring's) creates the read-only version.
- You need to add the libraries to the project and configure the environment.

MapStruct



- http://mapstruct.org
- Applications often contain similar, but different, object models, where the data in two models may be similar but the structure and responsibilities of the models are different. Object mapping makes it easy to convert from one model to another, allowing separate models to remain segregated.
- MapStruct makes object mapping easy by automatically determining how one object model maps to another, according to conventions, just as a human would, while at the same time providing a simple and secure refactoring API to handle specific use cases.

```
@Mappings({
  @Mapping(source = "itemId", target = "persistenceItemId"),
  @Mapping(target = "supplyType", ignore = true),
  @Mapping(target = "size", defaultValue = "10")
})
PersistenceMongoFindInventoryIDTO toPersistenceMongoFindInventoryIDTO(InventoryGetServiceIDTO
idto);
```

Projections

- Spring Data query methods typically return one or more instances of the repository-managed aggregate root. However, sometimes it may be convenient to create projections based on certain attributes of those types. Spring Data allows you to model dedicated return types, to more selectively retrieve partial views of managed aggregates.
- The easiest way to limit the result of queries to only the desired attributes, is to declare an interface or DTO that exposes access methods for the properties to be read. These must exactly match the entity's properties :

```
public interface NamesOnly {
    String getNombre();
    String getApellidos();
}
```

- The query execution engine creates proxy instances of that interface, at run time, for each returned item; and forwards the calls to exposed methods to the target object.

```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {
    List<NamesOnly> findByNombreStartingWith(String nombre);
}
```

Projections

- Projections can be used recursively.

```
interface PersonSummary {
    String getName();
    String getApellidos();
    DireccionSummary getDireccion();
interface
    DireccionSummary {  String
getCiudad();
    }
}
```

- In open projections, access methods in projection interfaces can also be used to compute new values:

```
public interface NamesOnly {
    @Value("#{args[0] + ' ' + target.nombre + ' ' + target.apellidos}")  String
    getNombreCompleto(String tratamiento);
    default String getFullName() {
    return getNombre.concat(" ").concat(getApellidos());
    }
}
```

Projections

- You can implement a more complex custom logic in a Spring bean, and then invoke it from the SpEL expression:

```
@Component  class MyBean{
String getFullName(Person person) { … }
}
interface NamesOnly {  @Value("#{@myBean.getFullName(target)}")  String getFullName();
…
}
```

- Dynamic projections allow you to use generics in the repository definition, to resolve the return type at invocation time:

```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {
<T> List<T> findByNombreStartingWith(String prefijo, Class<T> type);
}
dao.findByNombreStartingWith("J", ProfesorShortDTO.class)
```

Jackson serialization

- Jackson is a Java utility library that makes it easy for us to serialize (convert a Java object to a string with its JSON representation), and deserialize (convert a string with a JSON representation of an object, to a real Java object) JSON objects.
- Jackson is fairly “intelligent” and, without telling it anything, it is able to serialize and deserialize objects quite well. To do this, it basically uses reflection so that if we have a “name” attribute in the JSON object, it will look for a method “getName()” for serialization, and a method “setName(String s)” for deserialization.

```
ObjectMapper objectMapper = new ObjectMapper();

String jsonText = objectMapper.writeValueAsString(person);

Person person = new ObjectMapper().readValue(jsonText, Person.class);
```

- The serialization and deserialization process can be controlled declaratively using annotations:

  https://github.com/FasterXML/jackson-annotations

Jackson serialization

- @JsonProperty: indicates the alternative name of the Property in JSON.
  - @JsonProperty("name") public String getTheName() { ... } @JsonProperty("name") public void setTheName(String name) { ...
  - }
- @JsonFormat: specifies a format to serialize date/time values.
  - @JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd- MM-yyyy hh:mm:ss")
  - public Date eventDate;
- @JsonIgnore: flags a property (member level) to be ignored.
  - @JsonIgnore public int id;

Jackson serialization

- @JsonIgnoreProperties: flags one or more properties (class level) to be ignored.
  - @JsonIgnoreProperties({ "id", "ownerName" })
  - @JsonIgnoreProperties(ignoreUnknown=true) public class Item {
- @JsonInclude: used to include properties with empty, null, or default values.
  - @JsonInclude(Include.NON_NULL) public class Item {
- @JsonAutoDetect: used to override the default semantics of which properties are visible and which are not.
  - @JsonAutoDetect(fieldVisibility = Visibility.ANY) public class Item {

Jackson serialization

- @JsonView: allows to indicate the View in which the property for serialization / deserialization will be included .

```
public class Views {
    public static class Partial {}
    public static class Complete extends Partial {}
}
public class Item {  @JsonView(Views.Partial.class)  public int id;
    @JsonView(Views.Partial.class) public String itemName;
    @JsonView(Views.Complete.class)  public String ownerName;
}
String result = new ObjectMapper().writerWithView(Views.Partial.class)
    .writeValueAsString(item);
```

Jackson serialization

- @JsonFilter: indicates the filter to be used during serialization (it is mandatory to provide this).

```
@JsonFilter("ItemFilter")  public class Item {

public int id;

public String itemName;  public String ownerName;

}

FilterProvider filters = new SimpleFilterProvider().addFilter("ItemFilter",  SimpleBeanPropertyFilter.filterOutAllExcept("id",  "itemName"));

MappingJacksonValue mapping = new  MappingJacksonValue(dao.findAll());

mapping.setFilters(filters);  return mapping;
```

Jackson serialization

- @JsonManagedReference and @JsonBackReference: used to manage master/detail relationships by marking the collection on the master, and the reverse property on the detail (multiple relationships require assigning unique names).
  - @JsonManagedReference public User owner;
    @JsonBackReference
  - public List<Item> userItems;

- @JsonIdentityInfo: indicates the identity of the object to avoid infinite recursion problems.
  - @JsonIdentityInfo(
  - generator = ObjectIdGenerators.PropertyGenerator.class,
  - property = "id")
  - public class Item { public int id;

XML serialization (JAXB)

- JAXB (Java XML API Binding) provides a fast, convenient way to create bidirectional bindings between XML documents and Java objects. Given a schema which specifies the structure of XML data, the JAXB compiler generates a set of Java classes that contain all the code to parse XML documents based on the schema. An application using the generated classes can build a tree of Java objects that represents an XML document, manipulate the contents of the tree, and regenerate the documents in the tree, all in XML without requiring the developer to write complex parsing and processing code.
- The main benefits of using JAXB are:
  - Uses Java and XML technology.
  - Guarantees valid data.
  - Is fast and easy to use.
  - Can restrict data.
  - Is customizable.
  - Is extensible.

Main Annotations (JAXB)

- To tell JAXB formatters how to transform a Java object to XML and vice versa, you can annotate (javax.xml.bind.annotation) the JavaBean classes so that JAXP infers the binding scheme.
- The main annotations are:
  - @XmlRootElement(namespace = "namespace"): Defines the root of the XML.
  - @XmlElement(name = "newName"): Defines the XML element to use.
  - @XmlAttribute(required=true): Serializes the property as an attribute of the element.
  - @XmlID: Maps a JavaBean property to an XML ID.
  - @XmlType(propOrder = { "field2", "field1",.. }): Allows you to define the order in which the elements within the XML will be written.
  - @XmlElementWrapper: Wraps the elements of a collection in an element.
  - @XmlTransient: The property is not serialized.

Validations

- Since version 3, Spring has greatly simplified and enhanced data validation, thanks to the adoption of the JSR 303 specification. This API allows data to be validated declaratively, with the use of annotations. This makes it easy for us to validate the data sent before it reaches the REST controller.
- Annotations can be set at the class, attribute, and method parameter level.
- Validity can be enforced using the @Valid annotation on the element to be validated.
  - Public ResponseEntity<Object> create(@Valid @RequestBody Persona item)
- To perform validation manually:

```
@Autowired
private Validator validator;
Set<ConstraintViolation<@Valid Persona>> constraintViolations =  validator.validate( persona );
Set<ConstraintViolation<@Valid Persona>> constraintViolations =  validator.validateProperty (persona, "nombre" );
```

Validations

- @Null: Checks whether the annotated value is null.
- @NotNull: Checks whether the annotated value is not null.
- @NotEmpty: Checks if the annotated element is not null or empty.
- @NotBlank: Checks that the annotated character sequence is not null and that the trimmed length is greater than 0. The difference with @NotEmpty is that this restriction can only be applied on character sequences, and that trailing whitespace is ignored.
- @AssertFalse: Checks whether the annotated element is false.
- @AssertTrue: Checks whether the annotated element is true.

Validations

@Max(value=): Checks if the noted value is less than or equal to the specified maximum.

@Min(value=): Checks if the noted value is greater than or equal to the specified minimum.

@Negative: Checks if the element is strictly negative. Zero values are considered invalid.

@NegativeOrZero: Checks if the element is negative or zero.

@Positive: Checks if the element is strictly positive. Zero values are considered invalid.

@PositiveOrZero: Checks if the element is positive or zero.

@DecimalMax(value=, inclusive=): Checks if the noted numeric value is less than the specified maximum, where inclusive= false. Otherwise, checks if the value is less than or equal to the specified maximum.

@DecimalMin(value=, inclusive=): Checks if the noted value is greater than the specified minimum, when inclusive=false. Otherwise, checks if the value is greater than or equal to the specified minimum.

Validations

@Past: Checks if the noted date is in the past.

@PastOrPresent: Check if the noted date is in the past or present .

@Future: Checks if the noted date is in the future.

@FutureOrPresent : Checks if the annotated date is in the present or in the future.

@Email: Checks if the specified character sequence is a valid email address.

@Pattern(regex=, flags=): Checks if the annotated string matches the regex regular expression considering the given flag.

@Size(min=, max=): Checks if the size of the annotated element is between min and max (included).

Transactions

- By default, CRUD methods on repository instances are transactional. For read operations, the transaction configuration readOnly flag is set to true to optimize the process. All others are configured with a @Transactional plane so that the default transaction settings apply.
- When several calls to the repository or to several repositories are going to be made, the method can be annotated with @Transactional so that all the operations are within the same transaction.
  - @Transactional
  - public void create(Pago pago) { … }
- To make query methods transactional:
  - @Override @Transactional(readOnly = false) public List<User> findAll();

Service

- Services represent operations, actions, or activities that do not conceptually belong to any particular domain object. Services have neither their own state nor a meaning beyond the action that defines them. They are annotated with @Service.
- We can divide the services into three different types:
  - Domain services
    They are responsible for the more specific behavior of the domain, that is to say, they carry out actions that do not depend on the specific application that we are developing, but rather belong to the most internal part of the domain, and thus could make sense in other applications belonging to the same domain.
  - Application services
    They are responsible for the main flow of the application, that I to says, they are the use cases of our application. They are the external visible part of the domain of our system, so they are the input-output point to interact with the internal functionality of the domain. Its function is to coordinate entities, value objects, domain services and infrastructure services to carry out an action.
  - Infrastructure services
    They declare behavior that does not really belong to the domain of the application but that we must be able to perform as part of it.

Thank you!

Presentation:
Arquitectura DSP
arquitecturaDSP@minsait.com

Avda. de Bruselas 35
28108 Alcobendas,
Madrid España

T +34 91 480 50 00
F +34 91 480 50 80
www.minsait.com

minsait

An Indra company

Mark Making the way forward

An Indra company