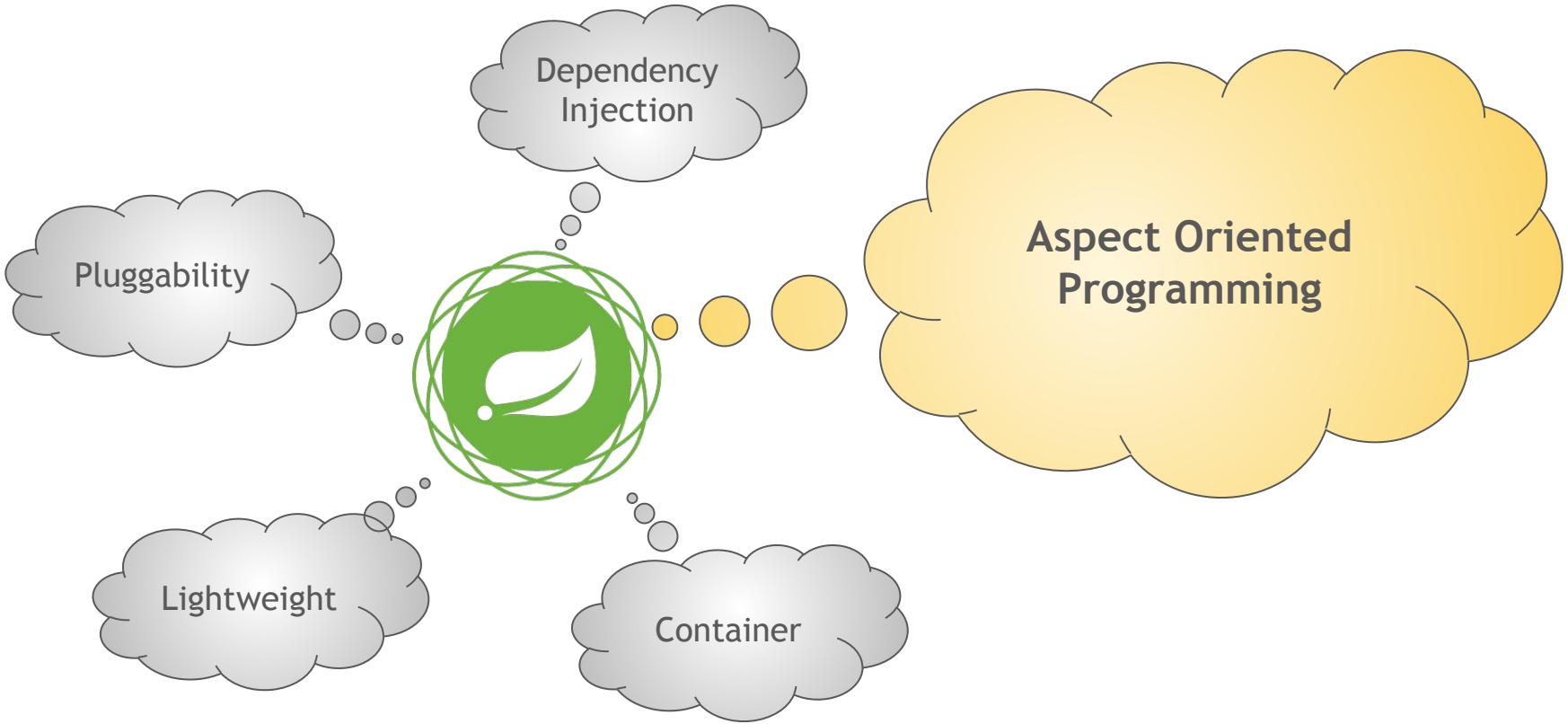
AN INTRODUCTION TO BACKEND FOR BEGINNERS

MODULO 5 – SPRING

Objectives:

- AOP basics
- Low level Spring layers
- Create a simple RESTful service

Spring Framework

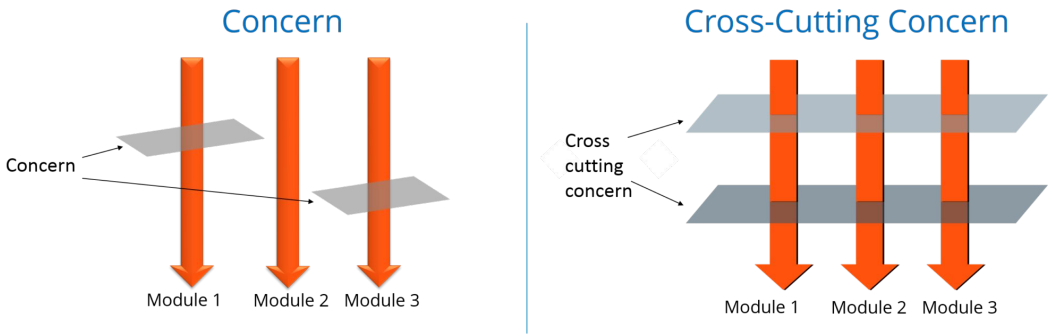


Concern vs Cross-Cutting Concern

*The **concern** is the **behavior** we want to have in a particular module of an application. It can be defined as a functionality we want to implement.*

*In software development, **functions that span multiple points** of an application are called **cross-cutting concerns**. They are conceptually separate from the application's business logic but often embedded directly within, for example:*

- ***Logging***
- ***Security***
- ***Transaction***
- ***Caching***



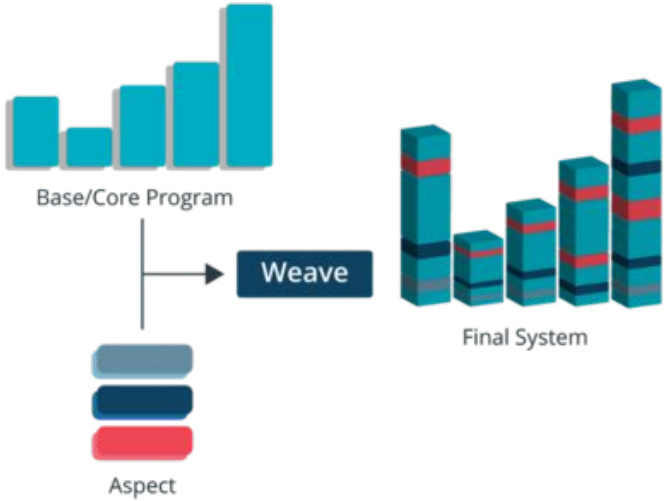
***Separating** these **cross-cutting concerns from the business logic** is where AOP goes to work.*

Aspect Oriented Programming

*AOP is a **programming technique** which allows programmers to **modularize crosscutting concerns** or behavior that cuts across the typical divisions of responsibility.*

*The core of AOP is an **ASPECT**. It **encapsulates behaviors** that can affect multiple classes into reusable modules.*

*With AOP can **define aspects** in one place and **declare how and where to apply** without the necessity to directly modify the class.*



Joinpoint - Advice - Pointcut

***Joinpoint** is a candidate **point** in the Program Execution of the application **where an aspect can be plugged in**:*

- *method being called*
- *exception being thrown*
- *field changes*

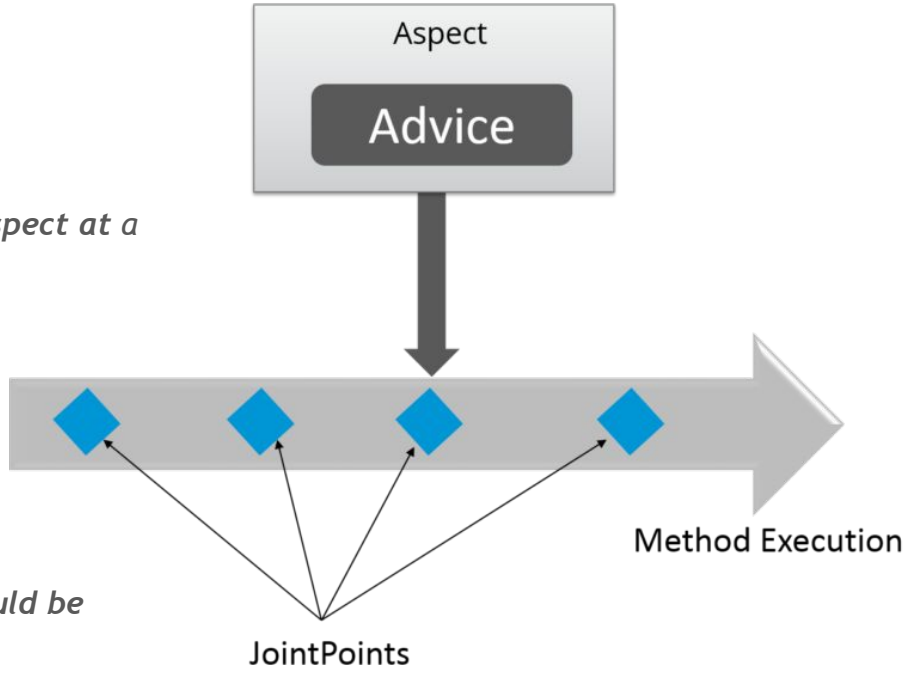
Advice** is an object **where** is contained the **action taken by an aspect at** a particular **joinpoint

It defines what an aspect will do and when it will be doing it:

- ***Before***
- ***After returning (success)***
- ***After throwing (fail)***
- ***After***
- ***Around***

Pointcut defines at which joinpoint, the associated Advice should be applied.

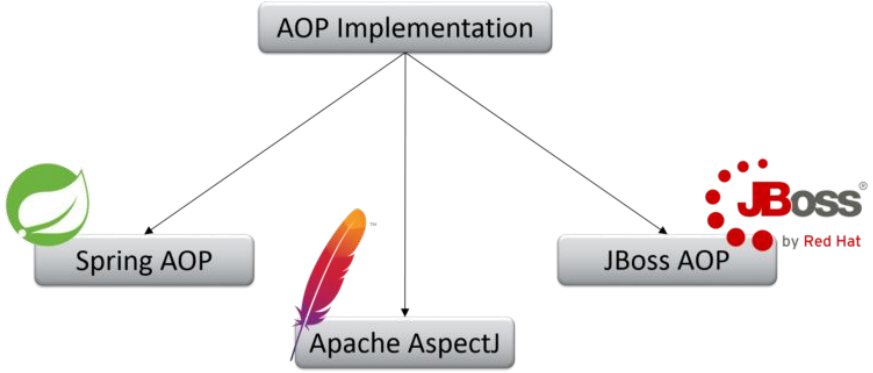
In other word it is an expression or pattern that groups join points that are to be associated to the advice.



Spring AOP

Spring has its own AOP framework implementation. IoC container does not depend directly from it, AOP complements Spring IoC to provide a very capable middleware solution:

- *weaving done at **runtime by proxy***
- *pointcut supported ONLY at **method level***
- *aspects can be applied to **spring beans***




```
@Configuration
@EnableAspectJAutoProxy
public class
}
```

```
<aop:aspectj-autoproxy/>
```

config

```
<dependencies>
   <dependency>
       <groupId>org.springframework.boot</groupId>
       <artifactId>spring-boot-starter-aop</artifactId>
   </dependency>
</dependencies>
```

pom.xml

Spring AOP - Example

ASPECT

```java
@Component
@Aspect
public class AdderAfterReturnAspect {

    @Pointcut("@annotation(example.package.Loggable)")
    public void loggableMethods() throws Throwable {}

    @Around("loggableMethods()")
    public Object measureMethodExecutionTime(ProceedingJoinPoint pjp) throws Throwable {
        // before method invocation

        Object retval = pjp.proceed(); // method invocation

        // after method creation

        return retval;
    }
}
```

DI by Interface

```java
@Component
public class ExampleBeanImpl implements ExampleBean {

    @Autowired
    private Adder adder;

    @Override
    public void example() {
        //...
        Integer result = adder.add(1,2);
        //...
    }
}
```

BEAN METHOD

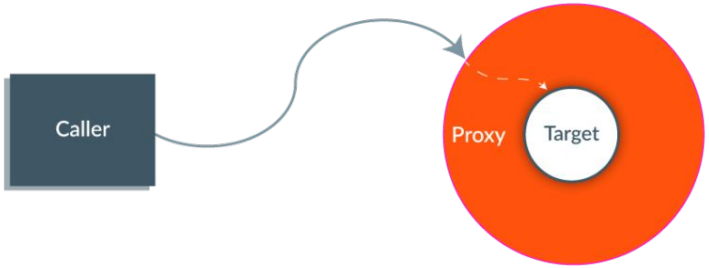
```java
@Component
public class SampleAdder implements Adder {

    @Loggable
    public Integer add(Integer a, Integer b) {
        return a + b;
    }
}
```

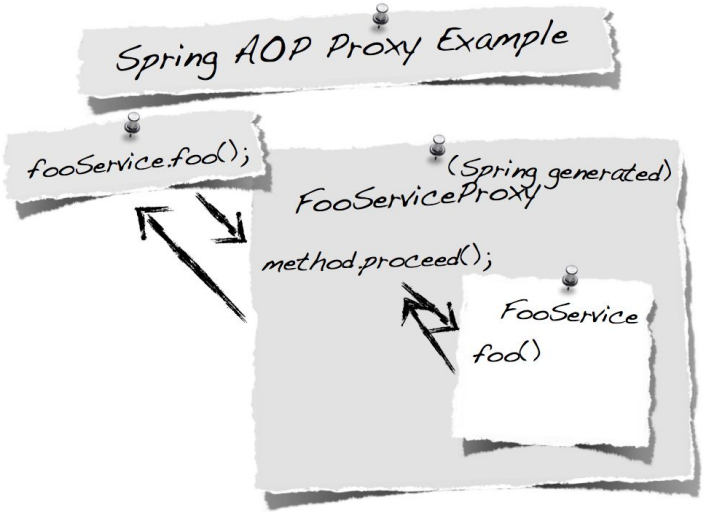
How AOP works - Weaving by Proxy

*The process of **linking an aspect** with other application types or objects to create an advised object is called **Weaving**.*

*In Spring AOP, weaving is performed at **runtime**. An **object which is created after applying advice** to a target object is known as a **Proxy**.*

*In Spring as **default the JDK Proxy Interface based** is used. So make **DI always by Interface**.*





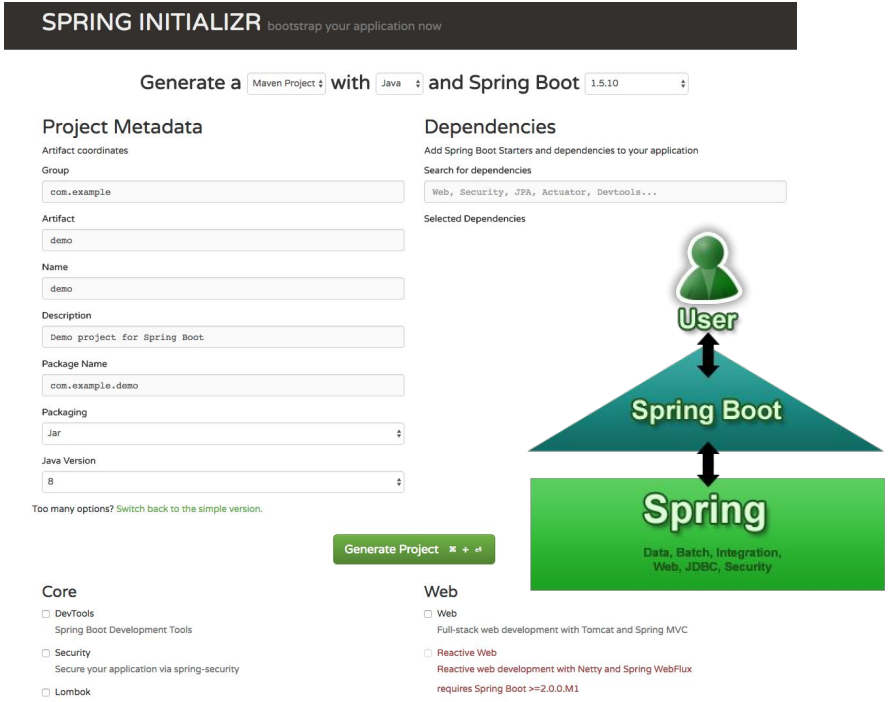
Spring Boot

"Spring Boot makes it easy to create standalone, production-grade Spring based applications that you can just run"

Spring IO

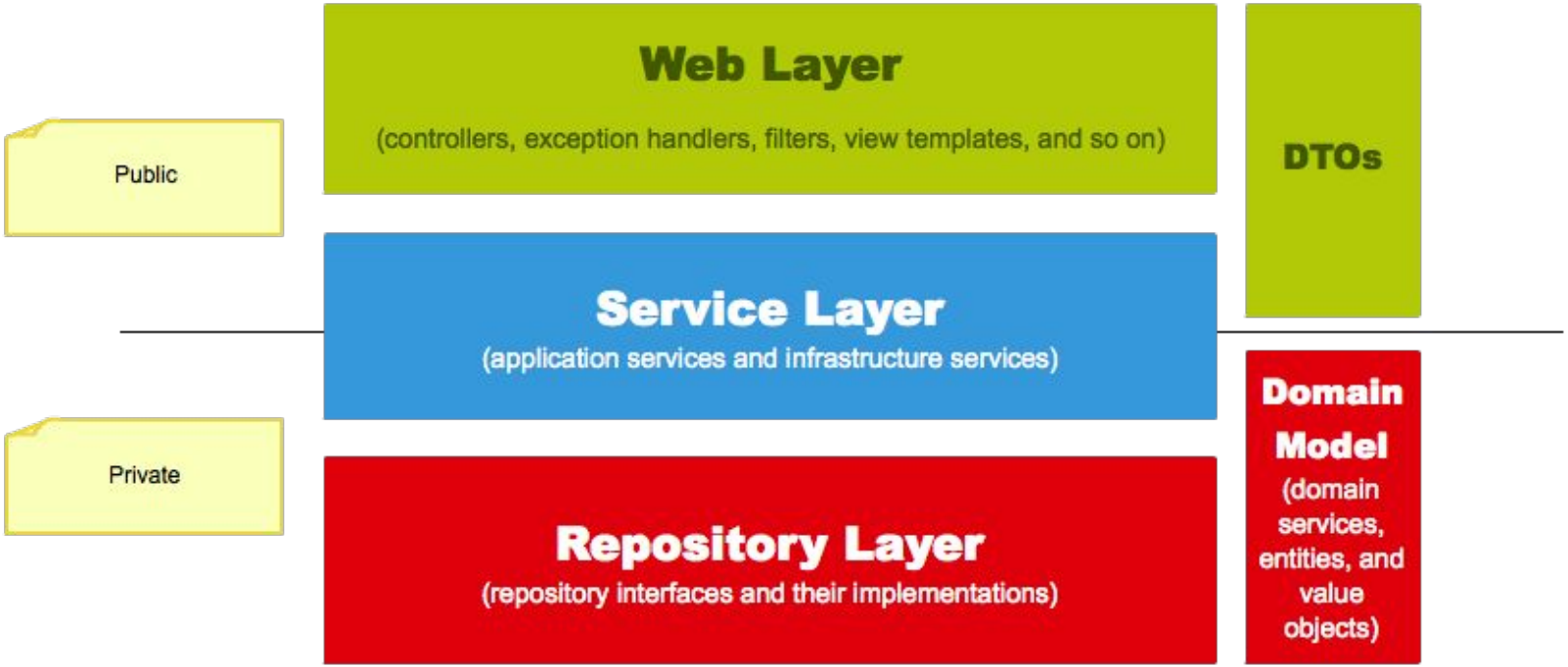
KEY FEATURES:

- Create stand-alone Spring applications
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' POMs to simplify your Maven configuration
- Automatically configure Spring whenever possible
- Provide production-ready features such as metrics, health checks and externalized configuration
- Absolutely no code generation and no requirement for XML configuration.



Spring - SoC

Separation of concerns (SoC) **is a design principle** for separating a computer program into distinct sections, such that each section addresses a separate concern.



DTO vs DM

A **data transfer object** is an object that is just a **simple data container**, and these objects are used to carry data between different processes and between the layers of our application.

In generals **DTOs** are the **representation of the data exposed to the clients**

A **domain model** consists of three different objects:

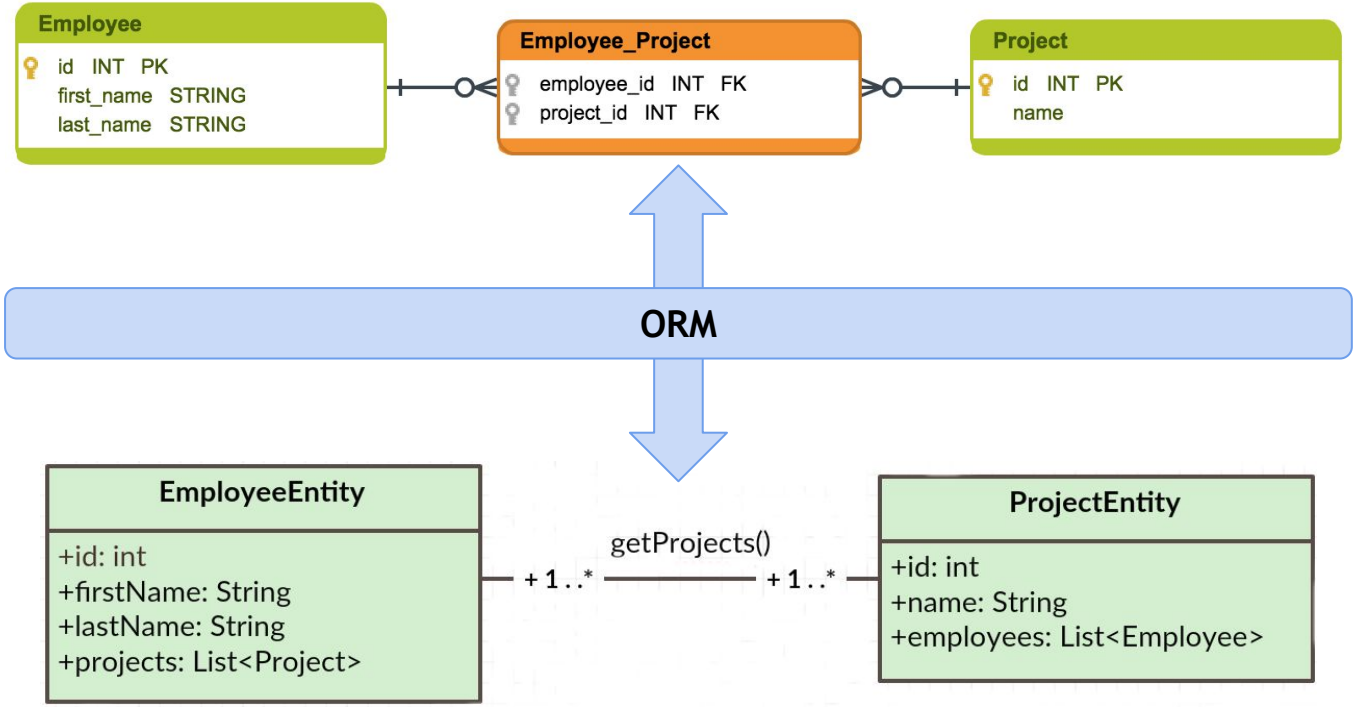
- A **domain service** is a stateless class that provides operations which are related to a domain concept but aren't a "natural" part of an entity or a value object.
- An **entity** is an object that is defined by its identity which stays unchanged through its entire lifecycle.
- A **value object** describes a property or a thing, and these objects don't have their own identity or lifecycle. The lifecycle of a value object is bound to the lifecycle of an entity.

ORM

ORM (Object-relational mapping) is a programming technique for converting data between relational and object-oriented data models. This creates a "virtual object database" that can be used from within the programming language.

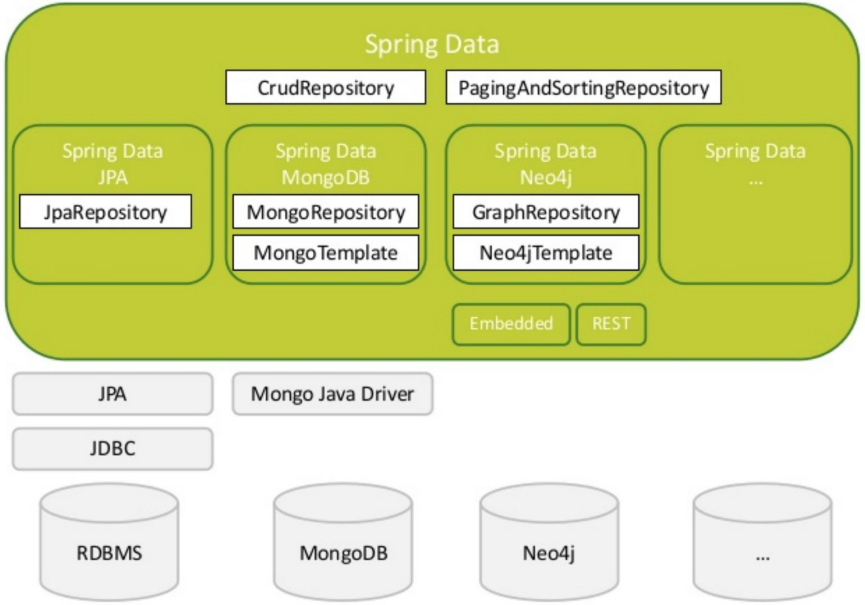
The most used techniques are:

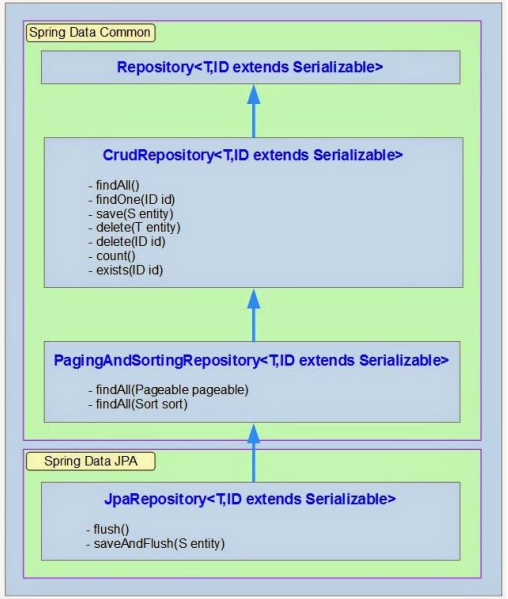
- Data Mapper
- Repository DAO
- Active Record



Spring Repository Layer

The **repository layer** is the **lowest** layer of a web application. It is responsible of communicating with the used data storage.
The repository layer takes **entities** (and basic types) as method parameters and returns entities (and basic types).
The **Repository** interface is central abstraction provided by **Spring Data** module.





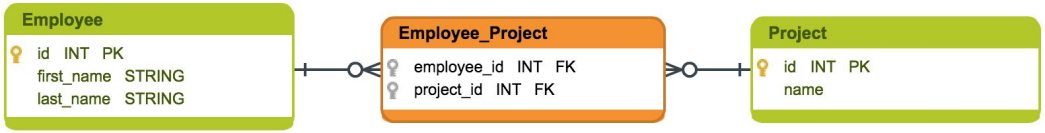
JPA Repository

The **Java Persistence API** (JPA) is a Java specification for a**ccessing, persisting, and managing** data between Java **objects** / classes **and** a **relational database.**

JPA itself is **just a specification, not a product**; it cannot perform persistence or anything else by itself. JPA is just a set of interfaces, and requires an implementation.

```
@Repository
public interface ProjectDao extends JpaRepository<Project, Integer> {

    List<Project> findProjectByNameEquals(String name);

    @Query(value = "select p from Project p where p.name = :name ")
    List<Article> findbyQuery(@Param("name") String name);

}
```

From DM to DTO




```java
@Entity
public class Employee {
 @Id
 @Column(name="id")
 private int id;
 @Column(name="first_name")
 private String firstName;
 @Column(name="last_name")
 private String lastName;

 @ManyToMany
 @JoinTable(
     name="Employee_Project",
     joinColumns=@JoinColumn(
                 name="employee_id",
                 referencedColumnName="id"),
     inverseJoinColumns=@JoinColumn(
                 name="project_id",
                 referencedColumnName="id"))
 private List<Project> projects;
 .....
}
```

```java
@Entity
public class Project {
 @Id
 @Column(name="id")
 private int id;
 @Column(name="name")
 private String name;

 @ManyToMany(mappedBy="projects")
 private List<Employee> employees;
 ...
}
```

MAPPER
CONVERTERS
POPULATORS

Employee assignments

Employee	Projects
John Doe	Project 1
Jane Doe	Project 2
Richard Doe	Project 1
Janie Doe	Project 1
Jonnie Doe	Project 2

```java
public class Assignment {
  private String employee;
  private String project;
}
```

Why DTO?

Do we really need data transfer objects?

Why cannot we just return entities and value objects back to the web layer?

The domain model specifies the **internal model** of our application. If we expose this model to the outside world, the **clients would have to take care of things that don't belong to them**.

If we use DTOs, we can **provide an easier and cleaner API**.

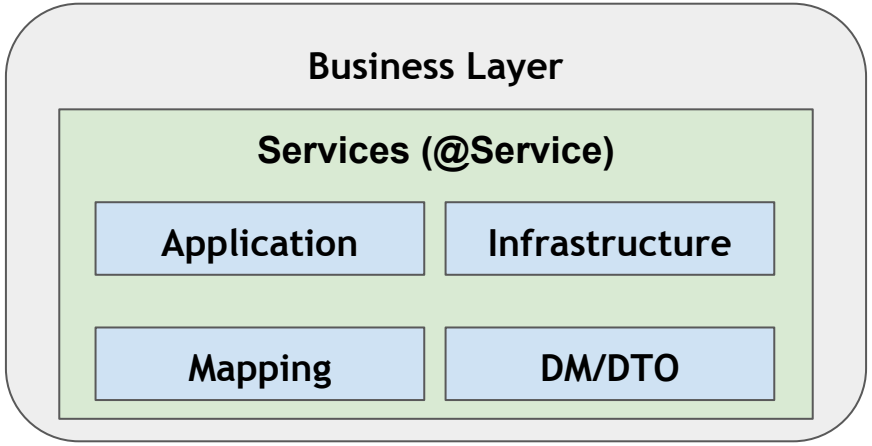
If we use DTOs, we can **change our domain model** as long as we don't make any changes to the DTOs.

Service Layer

The **service layer resides below the web layer**. It acts as a transaction boundary and contains both application and infrastructure services:

- **The application services provides the public API of the service layer**. They also act as a transaction boundary and are responsible of authorization.
- **The infrastructure services** contain the “plumbing code” that **communicates with external resources** such as file systems, databases, or email servers. Often these methods are used by more than a one application service.

The service layer **takes DTO** (and basic types) as method parameters. It **can handle domain model objects** but it can **return only DTO** back to the web layer.



Converter - Populator - Mappers

How do you transform objects from one type to another, in particular from entity to DTO in our case?
Implement an translator that transforms entity data types to business types:

- Converters
- Converters + Populators
- DataMapper

Many library can help performing this work in a configurable way:

- OrikaMapper
- Dozer
- MapStruct
- Selma

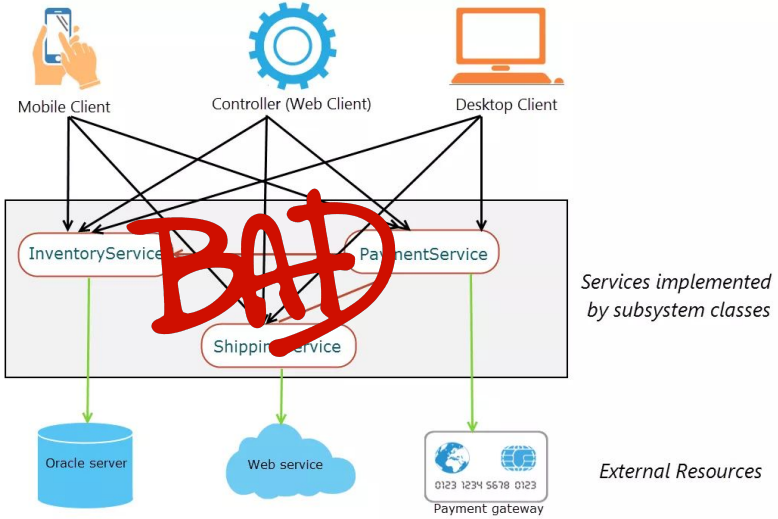
```
public interface GenericConverter<I, O>
  extends Function<I, O> {

  default O convert(final I input) {
    O output = null;
    if (input != null) {
      output = this.apply(input);
    }
    return output;
  }
}
```

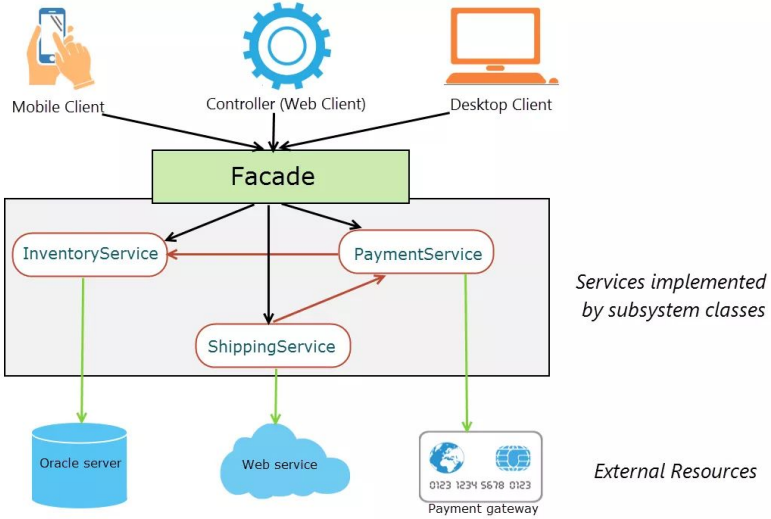
```
public class PersonConverter
  implements GenericConverter<PersonEntity, PersonDto> {

  @Override
  public PersonDto apply(PersonEntity input) {
    PersonDto output = new PersonDto();

     output.setFiscalCode(input.getFiscal());
     output.setFullName(input.getName() + " " input.getLastName());

     return output;
  }
}
```

Facade

*"Provide a **unified interface** to a set of interfaces in a subsystem. Facade defines a higher-level interface that **makes the subsystem easier to use**."*

Design Patterns: Elements of Reusable Object-Oriented Software





RESTful

It stands for Representational State Transfer and it is a way to provide interoperability between systems over HTTP

- *Resources expose their data and functionality through **resources identified by a unique URI***
- ***Uniform Interface Principle**: Clients interact with resources through a fix set of verbs.*
- ***Multiple representations for the same resource***
- ***Hypermedia model resource** relationships for dynamic navigation*

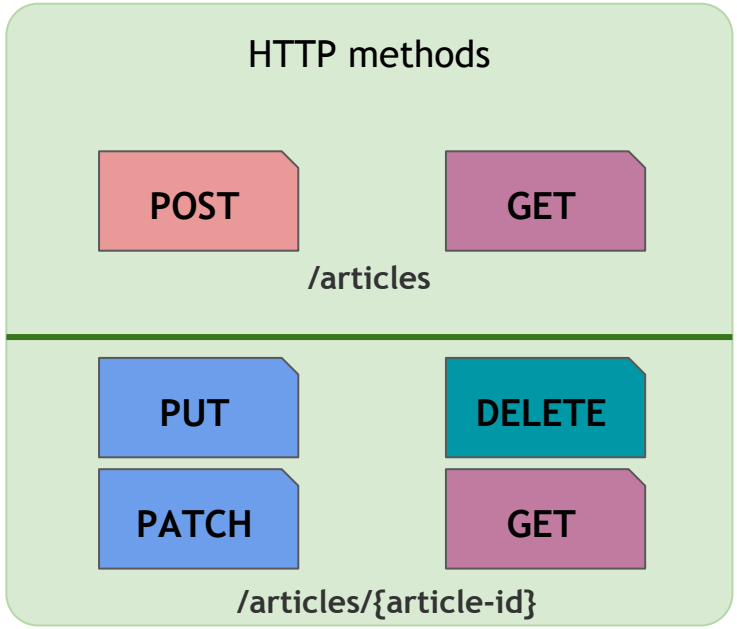
Client - Server | Stateless | Layered System | Cacheable | Uniform Interface

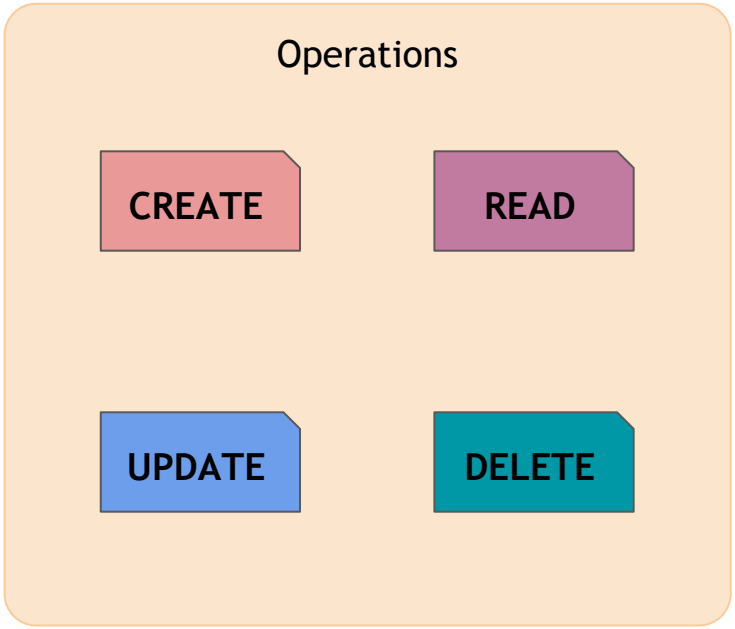
Resources and operations

*In REST, primary data representation is called **Resource**:*

- *A resource can be a **single** or a **collection***
- *A resource **may contain sub-collection** resources*

http://api.example/**articles**
http://api.example/**article/{article-id}**
http://api.example/**article/{article-id}/comments**
http://api.example/**article/{article-id}/comments/{comment-id}**





Spring MVC - REST

REST

- Resources
- URIs/URLs
- HTTP Methods
- Representations

- @RestController, @ResponseBody
- @PathParam, @RequestParam, @RequestBody
- @RequestMapping, @GetMapping, …
- JacksonConverter: XML, JSON.

REST Controller Example in Spring

```java
@RestController
@RequestMapping(value = "/assignments")
public class RestAssignmentController {

    @Autowired
    private AssignmentFacade assignmentFacade;


    @GetMapping
    public List<AssignmentDto> getAssignments() {
        return assignmentFacade.retrieveAllAssignments();
    }

    @GetMapping(value = "/{id}")
    public AssignmentDto getAssignment(@PathVariable("id") Long id) throws ElementNotFound {
        return assignmentFacade.retrieveAssignmentById(id);
    }

    @RequestMapping(method = {RequestMethod.POST, RequestMethod.PUT, RequestMethod.PATCH})
    public ArticleDto createOrUpdateAssignment(@RequestBody AssignmentDto assignmentDto) throws ElementNotFound {
        return assignmentFacade.saveOrUpdate(assignmentDto);
    }

}
```

```java
public class AssignmentDto implements Serializable {
  private String employee;
  private Integer employeeId
  private String project;
  private Integer projectId
}
```

Useful links

Rest

Martin Fowler - ORM Hate

Comparing Spring AOP and AspectJ

Exception Handling with ControllerAdvice example

REST hypermedia with Spring HATEOAS

Github project course repository and Lessons documentation:
https://github.com/mcolombosperoni/an-introduction-to-backend-for-beginners