Spring Course - Jack Derby

* JDK: 17/higher (Spring 3)
* Intellij

Spring:

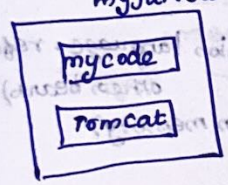
* Java app: helper classes, Annotation
* Hard: Jar dependencies (Config: xml/Java), Server: getting Started.
* Springboot: Rescue (easy to get Started): autoconfig, JAR Class path, Server

TomCAT, JETTY, UNDERTOW

* Springboot uses spring (makes easy!)

Spring.io

Spring Boot:

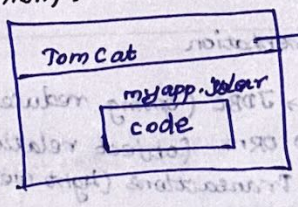


: App Server part of code: can be Independant too!

Run app

> Java -Jar mycoolapp.Jar

Deploy war: (traditional fashion):



→ my Company server

Tomcat

myapp.war | myapp1.war | myapp2.war

WAR: Web applicatione archieve.

Spring Book:

* uses MVC, REST, AOP (In Background): mainly about Config!
* Spring Boot uses Spring under the hood.

MAVEN:

* add dependency from its repo (downloads, available)! : Shopper

Rest controller:

* class level annotation: handle request from client

```
* @RestController
public class FunRestController {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello";
    }
}
```

Why Spring:

* Light weight: PoJo (Plain old Java object) : Compared to heavy EJB(JEE)
* Loose coupling: dependency injency
* Boilen plate: minimize by codes (readymade)

Core Container: (Create, manage : Beans) from properties, Config files
- Bean
- Core
- SpEL (Spring expression language: refer other beans)
- Context (holds Spring containers in memory)

AOP Section: Aspect oriented Program (app wide services)
- Security
- Transaction
- logging
- Instrumentation

Data access layer: Comm. with Database
- JDBC (Spring reduce code by 50%)
- ORM (object relational mapping: hibernate & JPA Support)
- Transactions (light weight): Flexible
- oxm, JmS (Java message Services: asynchronously)

pack of JEE.

Web layer: (Spring MVC)
- Socket
- web socket
- web

Instrumentation: Remotely monitor app (JMX: Java management extension)
- Instrumentation
- Logging
- Service: monitor! & Instrument
- Agent)
- messaging!

web Remoting (external clients making calls)

Distributed systems RPC: Remote procedure Calls

Test layer:

Test driven development (mock objects, out of container testing)
- Unit
- Integration
- Mock!

Spring projects:

* Spring cloud, Data
* Batch, Security
* web Services, LDAP

Spring.io : For official doc

MAVEN:

* Pom.xml : dependencies (project management, dependency tool)
* Helper / shopper.
* local repo, MAVEN Central (Save in local repo: cache) → Build & Run (handle class paths).

Directory Structure:

>myapp
  pom.xml
  >src
    >main
      >java → my source code
      >resources → property / config of app
      >webapp → JSP/web config files + assets (img, css, js)
    >test → Test & properties
      >java
      >resources
  >target → Destination of Compiled Code (By maven: created)

* Easy to handle: find where located, built in maven support: IDE (share project b/w people), IDES

Key maven:

* pom:
  * metadata, dependencies, plugins
    - metadata: name, version no, o/p file type
    - dependencies: Jars needed
    - plugins: (Add. Tasks) eg: Junit test report

eg.

```
<project ..>
  <groupId> Com.luv2code </groupId>
  <version>
  <artifactId>
  <packaging> Jar
  <name> myapp
  <dependencies>
  <dependency>
      groupId
      artifact
      version
      scope
      :
  </dependencies>

  <plugins>  → Custom tasks.
```

Project coordinates:

* groupId, artifactId, version
  - groupId → name of Company/group/org
    - reverse domain name: Com.luv2code
  - artifactId → name of project (myCoolAPP)
  - version → 1.0-Snapshot (active development), 1.2, ...

eg:
org.springframework
spring-context
6.0.0 → optional (But good practice), especially devops

GAV: groupId, artifactId, version

maven.org

Presentation layer (Authentication, JSon translation)
↕
Business layer (Logic, Validation, Authorization)
↕
Persistence (Storage logic)
↕
DB (actual DB)

Spring Boot: Takes care of DAO & Implementation (Repository class)! extending CRUD

Project structure: (std MAVEN directory)

maven wrapper classes:

If already installed: no need

1) mvnw
2) mvnw.cmd → windows!
   mvnw.sh → Linux/mac!

] Run mavel projects (even without maven! : downloads & run by itself)

```
>mvn clean compile test
```

pom.xml:

* Spring Initializer info, maven dependency info, plugin info!

```
<build>
    <plugins>
        <plugin>
            gId: org.springframework.boot
            aId: spring-boot-maven-plugin
        </plugin>
    </plugins>
</build>
```

- `./mvnw spring-boot:run` → To run
- `./mvnw package` → To package

Application properties:

- ★ application.properties
- ★ eg: port,

*) server.port = 8585

*) coach.name = Mickey Mouse — (Inject) Read By → `@Value("${coach.name}")` `private String coachName;`

Static:

- ★ /static: html, js, css, img, pdf (Read after loading)
- ★ src/main/webapp: don't use for JAR only works with WAR! (Silently ignored by build tools)

Templates:

- ★ Auto config for Free marker, Thymeleaf, mustache.

`src/resources/templates`
`src/resources/static`

Spring boot starters:

- ★ Less: Collections grouped, Tested & verified (easy dev), Reduces maven config!

Spring mvc: spring-webmvc, hibernate-validator, thymeleaf — Spring starter → spring-boot-starter-web (each version: have versions mentioned eg each JAR)

30+ Spring boot starters: : AutoConfig in pom.xml by Spring starters!

View > Tool Window > Maven > Dependencies

Starter parent: (default by Spring boot)

`spring-boot-starter-parent` → compiler level, UTF-8 source encoding

Set as property:

```
<properties>
    <java.version>17</java.version>
</properties>
```

override! (default property)

* spring-boot-starter-* (no need to version: Individually) : uses parent version.

* spring-boot-maven-plugin!

Benefit of starter parent:

*Use version of parent

*UTF encoding, Java version: default config

*Dependency management.

Dev tools

*problem: manually restart each time

* spring-boot-devtools (restarts app each time: code change)

Preference > Build > Execution, Deployment > compiler

*Buid project automatically: checkbox!

+

Pref > Advanced Setting

*Allow auto-make to start...

Spring Boot actuator

*monitor & manage app, check app health, access app metrics

*exposes endpoints: monitor, manage app, devops (out of box)

dependendency: adds rest endpoints for free

org.springbookframework.boot

spring-boot-starter-actuator

*prefix: /actuator

/help : health info (status) ; up/down

/info : info about app

Note:

* Default: /health alone exposed!

src/main/resources/application.properties

management.endpoints.web.exposure.include = health, info

management.info.env.enabled = true

/actuator/info:

* Default = { }
* info.app.name = my app
  info.app.description = my app desc
  info.app.version = 1.0.0

{ "app" : { name : myapp, desc : my app, ver : 1.0.0 } }

/auditevents : event audit
/beans : list of all beans registered in spring app context
/mappings : list of all @RequestMapping paths

actuator endpoints

management.endpoints.web.exposure.include = * (all enpoints)

Security: Can be done!

/actuator/beans : all spring beans (internal, @Component : classes)
/actuator/threaddump : list of all threads (Analyze, Profiling : check bottleneck)
/mappings : list of request mappings /endpoints.

Actuator Security

* Secure : Spring boot starter security (username, password)
  (user : default) → Console : generated.

override : (application.properties)

spring.security.user.name = scott
spring.security.user.password = password

customize : DB, Roles, Encrypted passwords!

exclude endpoints:

→ management.endpoints.web.exposure.exclude = health, info (These can't be accessed)

Process:

1. pom.xml : add spring-boot-starter-security
2. verify /beans.
3. Disable /health, /info : no one can access!

---

my app.jar
mycode
Tomcat

From Command line : Run app

1) Java -Jar
2) mvn spring-boot : run

option 1) Java -Jar myapp.jar
2) mvn spring-boot : run

Already maven installed : mvn clean Compile test

default: src/main/resources/app...prop — Read from config values of application.properties

```
@Value("${coach.name3}")
private String coachName;
```
→ Injection

Spring boot properties (more than 1000)

* port, context path, actuator, security.
* groups: core, web, security, data, actuator, integration, devtools, testing

log:

```
Logging.level.org.springframework = DEBUG
Logging.level.org.hibernate = TRACE
Logging.level.com.luv2code = INFO
```
For each projects! (Trace, debug, info, warn, error, fatal, off) : o/p : file

```
Logging.file = myfile.log
```

web

```
server.port = 7070
server.servlet.context-path = /stark (or) /books
server.servlet.session.timeout = 15m
```

Actuator:

```
management.endpoints.web.exposure.include = *
                                          • exclude = beans, mapping
web.base-path = /actuator
```

Security:

```
Spring.security.user.name = admin
                          • password = secret
```

Customize (use roles, DB):

```
public class DemoSecurity extends WebSecurity

@Autowired
private DataSource securityDataSource;

@override
protected void configure(AuthenticationManagerBea..)
{
    auth.jdbcAuthentication().dataSource(securityDataSource);
}
}
```
→ Read from Database!

```
@override
Protected void Configure (HttpSecurity http) throws Exception
{
    http.authorizeRequest()
        .antMatchers ("/actuator/**").hasRole ("ADMIN");
}
```

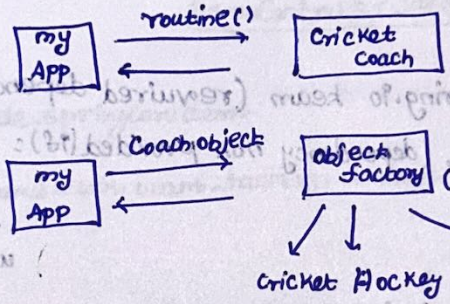
→ Only authorize admin users!

Data properties:

```
Spring.dataSource.url = jdbc:mysql://localhost:3306/ecommerce
Spring.datasource.username = Scott
Spring.datasource.password = tiger
```

Inversion of Control (Constructor, method)

* OutSource Construction & management of objects



* Configure app: many Coaches

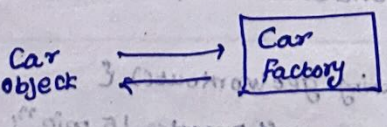
Spring Container: object factory → Inversion of Control (create, manage objects)
→ Inject dependencies (dependency Injection)

Configure Spring Container:

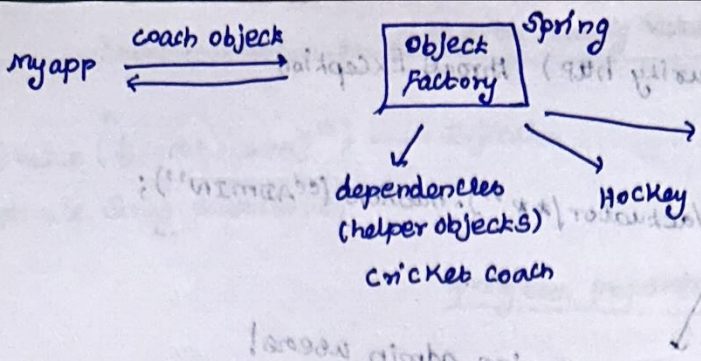
1) XML (legacy)
2) Java annotation (modern)
3) Java Source code (modern)

Dependency injection

* Dependency Inversion principle: Client delegates to another object the respons.. of providing its dependencies.



1) Car factory: assembles all dependencies & gives final one!



Spring Container (object factory) roles:

1) IoC : Create, manage objects
2) DI : Dependency Injects

eg:

* DemoController uses Coach (dependency)

Types of injection:

1) Construction
2) Setter

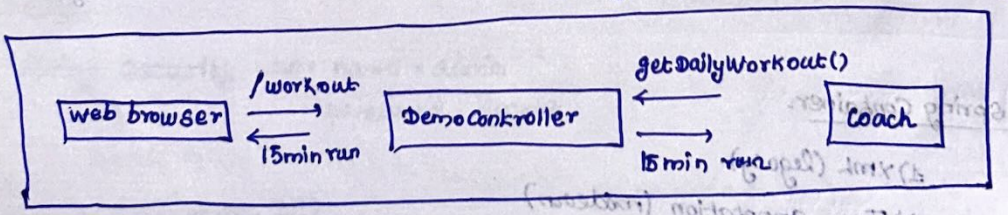
] preferred

* Constructor: preferred by spring.io team (required dependencies)
* setter: optional dependencies. (dependency not provided (if): app can do reasonable logic)

won't break!

Spring Autowiring (dependency injection):

* look for classes: match by type (inject it automatically)
* scan for @Components (any implementations: inject!)



Constructor injection

* Dependency interface & class: create Constructor, mapping!

Coach.java

```
package com.luv2code.demo

public interface Coach {

    String getWorkout();

}
```

CricketCoach.java

```
@Component
public class CricketCoach implements Coach {

    @Override
    public String getWorkout() {

        return "practice 15 min";
    }
}
```

DemoController:

@Component:

* Candidate for dependancy injection (Searched & injected) : makes available for Dependency Injection
* Regular Spring Bean : Regular Java class - managed by Spring.

```
@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(Coach theCoach) {
        myCoach = theCoach;
    }
}
```

```
import org. ... .factory.annotation.
    Autowired;

• RestController;
```

Coach: not added/imported (done by Spring)!

* Only 1 Constructor: @Autowired is optional (But preferred)

DemoController.java

```
package com.luv2code.springcoredemo

import org.springframework.beans.factory.annotation.Autowired;
                                    • GetMapping;
                                    • RestController;

@RestController
public class DemoController {

    private Coach myCoach;

    @Autowired
    public DemoController(Coach theCoach){
        myCoach = theCoach;
    }

    @GetMapping("/workout")
    public String getDailyWorkout(){
        return myCoach.getDailyWorkout();
    }
}
```

* No usage (IDE): Dynamic nature of Spring (unable to detect how beans are injected)! : Not explicitly referring implementation! (Referring Interface!)

Behind the Scenes: Constructor injection

→ Coach theCoach = new CricketCoach();
DemoController demoController = new DemoController(theCoach);

Note: Until now : Only one implementation!

I can do this new myself!

* Spring (more than IoC & DI)
* Small app : hard to see the benefits!
* Spring: Targeted for Enterprise applications!

DB access, transactions
REST API, Spring web mvc
security...

Component Scanning:

* @Component : Scan for Java classes (Automatically registers beans, DI)

@SpringBootApplication:

* Enables autoConfig, Component Scanning,....

@Configuration
@EnableAutoConfiguration
@ComponentScan

(Combination of these annotations)

@SpringBootAnnotation:

* Composed of following annotations
  - @EnableAutoConfiguration : enables auto config support
  - @ComponentScan : Scan packages, subpackages
  - @Configuration : Registers Beans, import other config (@Bean) extra

SpringApplication:

* BootStrap application : give reference to our app class (eg: myApp)
* Behind:
  1) Registers Bean, Starts Embedded Server.

Component Scanning:

* Starts : From folder : where Spring Boot App main
* Subpackages recursively!

Common pitfalls:

* Different package outside of that!
* Default : Ignores these packages

eg: com.myPackage

my main + its subfolders

> src
> main
> java
> com
> myApp
> Common
> Controller
> rest
myApp.java

Manual Scanning:

```
@SpringBootApplication(scanBasePackages = {"com.luv2code.myDemo",
                                           "com.luv2code.util",
                                           "edu.cmu.srs"})

public class SpringCoreDemoApp {
    ⋮
}
```

Setter injection

* Inject while calling setter methods!
* Inject Coach implementation (@Autowired)
  ↓
  Scan @Components ⟶ Create CricketCoach by dependency injection)!

eg:

```
@Autowired
public void setCoach(Coach theCoach){
    myCoach = theCoach;
}
```

Behind the scenes:

```
CricketCoach theCoach = new CricketCoach();
DemoController demoController =
        new DemoController();
demoController.setCoach(theCoach);
```

Note: we can @Autowired: any methods, variable,
↳ any method name!

which one to use?

Constructor: when (development team: Spring.io preferred 1st choice)!
required dependency!

Setter : optional (Even not provided: app can provide reasonable default logic)!

Field injection : No longer recommended

* early days: popular. Harder for unit test: not recommended
* Even private fields: can set by using Java reflections!

Java reflections → private variable (field injection!

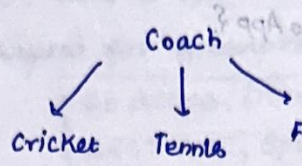
```
@Autowired
Private Coach myCoach;
@GetMapping("/")
public String getWorkout(){
    return myCoach.getWorkout();
}
```

→ Harder for unit test!

Qualifiers

Autowiring : inject, spring scans for @Components, search for Interface Implementation. Inject them! (Single Implementation: no problem)!

Multiple Implementation:




```
package com.luv2code.springdemo.common;
import org.springframework.stereotype.Component;

@Component
public class CricketCoach implements Coach{

    @override
    public String getWorkOut(){
        return "Net practice 15min";
    }
}
```

Similarly,
- TennisCoach
- TrackCoach
- BaseballCoach,

Error: Required single bean but 4 were found!

Be Specific : Qualifier!

Qualifier: [when multiple implementations there!]

Constructor Injection with qualifier

```
@Autowired
Public DemoController(@Qualifier("CricketCoach") Coach theCoach){
        myCoach = theCoach;
}
```

name : class (first character lower case)

Setter Injection with qualifier

```
@Autowired
public void setCoach(@Qualifier("tennisCoach") Coach theCoach){
    myCoach = theCoach;
}
```

Note: only classes with @Component are considered while injection!

Primary Annotation

* Alternative to Qualifier annotation.
* Kind of default annotation!

```
@Primary
@Component
public class TrackCoach implements Coach {
    ;
}
```

→ multiple implementations, takes & injects TrackCoach

Note: only one primary annotation allowed for multiple implementation! (more than 1 primary bean found among candidates)

mixing @Primary and @Qualifier

* Allowed: Qualifier has the highest priority! (overridden by Qualifier)

@Primary: TrackCoach

@Qualifier("CricketCoach") ⟶ Injects CricketCoach not TennisCoach

↳ "very specific": Recommended (as having high priority!)

Lazy loading

* Default: all beans initialized, scan for all components (new instance created, available)
* Test: add println() inside constructors!

Lazy Initialization:

* only int when needed for dependency injection / explicitly requested
* @Lazy annotation

```
@Component
@Lazy
public class TrackCoach implements Coach {
    ;
}
```

→ @Lazy to each class

Tedious to add @Lazy in every class:

* Global Config: application.properties

```
spring.main.lazy-initialization=true
```

* All beans lazy! eg: DemoController (using Cricket)
  1) Init CricketCoach
  2) Create DemoController
  3) Inject CricketCoach!

Advantage:

* only objects are created while using
* Helps faster startup (large no. of components)

Disadvantage:

* web related components like @RestController: not created until requested
* May not discover config issues: until too late!
* Ensure memory for all beans: once created

> Disabled by default
> Before enabling: Do profiling. Is it giving better results
> Trying to optimize which is not even worth: premature optimization pitfall

---

Java Beans

Scope:

* Lifecycle (live?, No. of instances created, how shared)
* Default Scope: singleton (only 1 instance, cached: all dependency injection refers that bean): same bean!

Singleton:

```
@Autowired
public DemoController (@Qualifier ("cricketCoach") Coach coach1, @Qualifier ("cricketCoach")
                                                   Coach coach2) {
    :
}
```

> Note: coach1, coach2 : same! reference : singleton

explicitly define Scope:

```
@Component
@Scope (ConfigurableBeanFactory.SCOPE_SINGLETON)
public class CricketCoach {
    :
}
```

1) ConfigurableBeanFactory.SCOPE_SINGLETON
2) . PROTOTYPE : new bean for each container request
3) . REQUEST: only used for web request.
4) . SESSION: Scoped to an HTTP web session
5) . GLOBAL_SESSION: global http web session

(3, 4, 5: web apps only)

• SCOPE_PROTOTYPE :

* New object instance for each injection.

Bean lifecycle

Container Started → Bean init → Dependency injection → Internal Spring processing.

↓

own custom init method

↓

Ready to use

↓

Bean container Shutdown

↓

Custom destroy method!

1) Add custom bean init
2) Business logic methods
3) Setup handles to resources (dB, sockets, file etc..)
4) Add custom code during bean destruction!

Init method Config:

```
@Component
public class CricketCoach implements Coach {

    @PostConstruct
    public void doMyStartupStuff() {
        ____
    }
    :
}
```

@PostConstruct → once bean is constructed, runs this startup init method!

```
@PreDestroy
Public void doMyCleanStuff() {
    ____
}
```

@PreDestroy → Before destroying!

Note

1) For 'prototype' scope: Spring won't call destroy method!

Doesn't manage lifecycle of a prototype bean

* Instantiate, Configure, handover (no further record of that object)

* No destroy method called
* Init method called (regardless of scope)
* prototype: Configured destruction lifecycle methods: not called.

client code must cleanup prototype scoped objects & cleanup. prototype scoped objects & release expensive resources held by prototype scope!

* All prototype beans are lazy by default (no need for @Lazy annotation)!

Configure beans with Java Code (no annotation)

1) Configuration class, Bean method to Configure the bean
2) Inject the bean into our Controller.

Step 1:

Package ——

Import org.springframework.context.annotation.Configuration;

```
@Configuration
public class SportsConfig {

}
```

Step 2: Define bean method to config bean

```
@Bean
public Coach swimCoach {
    return new SwimCoach();
}
```

→ method inside SportsConfig

"Return instance!"

Step 3 Inject!

```
@RestController {
public class DemoController {
    private class myCoach;
    @AutoWired
    public DemoController (@Qualifier("swimCoach") Coach theCoach ) {
        myCoach = theCoach;
    }
}
```

BeanID: Same as method name (swimCoach)

Use case eg @Bean: (why not @Component)

1) Make existing 3rd party class available to Spring Framework.
2) No access to (JAR) source code (not writable)

eg: AWS to store documents (S3: cloud based storage service) → Binary, text, pdf, img, ...

↳ Use S3 client as a spring bean (Part of AWS SDK)!

* Can't Component annotation added to AWS JAR!

★ Use @Bean : Configure as Bean!

```
@Configuration
public class DocumentsConfig {

    @Bean
    public S3Client remoteClient() {
        ____________
    }
}
```

Example,

```
@Autowired
public DocumentService(S3Client myS3){
    S3Client = myS3;
}
```

'Search for Beans' → @Components, @Beans

eg:

1) `public class SwimCoach implements Coach {` ⟶ No @Component annotation
   ⋮ Fails (autowiring)!
   }

2)

```
@Configuration
public class SportsConfig {

    @Bean ⟶ Name Same as class! (So, Spring can find)!
    public Coach swimCoach() {
        return new SwimCoach();
    }
}
```

Custom name : (Custom Bean ID)

```
@Bean("aquatic")
public Coach SwimCoach() {
    return new SwimCoach();
}
```

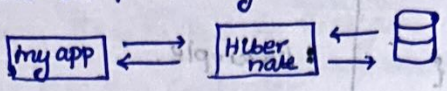
→ `@Qualifier("aquatic")`

Note : customName desired ⟶ Need to use that (else error!)

Hibernate JPA

Hibernate:

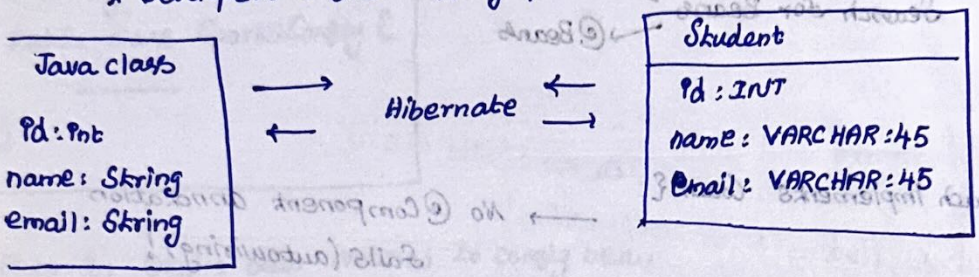
* Framework for persisting Java object in DB (www.hibernate.org/com)



* Benefits: minimize JDBC code, low level SQL code
  ORm (object Relational mapping)

ORm: (Object Relational mapping):

* Hibernate provides ORm
* Developer: Define mapping b/w Java class & DB table.

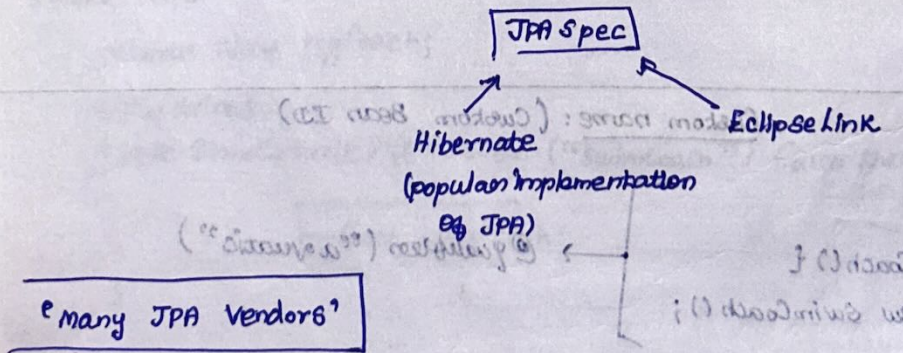


1) XML
2) Config files
3) Java annotation

} To map relationships

JPA: (Jakarta/Java Persistence API):

* Standard API for ORm
* Defines set of interfaces, implementation! (need for using)



Benefits of JPA:

* Interfaces: Not locked to a Vendor's implementation
* portable, flexible by coding to JPA specs: interfaces
* Easy to switch vendors of JPA! (As coding using Standard JPA!)

↓

1) Swap implementation (Advantage of using JPA!) for ORm!

Save Java object:

```
Student theStudent = new Student ('Paul', 'Doe', 'paul@gmail.com');
entityManager.persist(theStudent);
```

↳ JPA helper object! (Take & persist to DB)! : Hibernate will do!

Retrieve: (using PK)

```
int theID = 1;
Student myStudent = entityManager.find(Student.class, theID);
```

All Students:

```
TypedQuery<Student> theQuery = entityManager.createQuery("from Student",
                                                          Student.class);
List<Student> Students = theQuery.getResultList();
```

CRUD

Relation b/w Hibernate/JPA & JDBC

* Hibernate/JPA: uses JDBC for all DB connections/communication
  Hibernate: another layer of abstraction on top of JDBC
  (All goes through JDBC API)

* MySQL Database Server (main engine: stores data, supports CRUD)
  MySQL workbench: client GUI! (create, delete, edit DB, CRUD, admin roles.)

Database:

Set up table

```
CREATE USER 'springstudent'@'localhost' IDENTIFIED BY 'springstudent';
GRANT ALL PRIVILEGES ON *.* TO 'springstudent'@'localhost';

CREATE DATABASE IF NOT EXISTS 'student_tracker';
USE 'student_tracker';

CREATE TABLE 'student';
CREATE TABLE 'student' (
        'id' int NOT NULL AUTO_INCREMENT,
        'first_name' varchar(45) DEFAULT NULL
        'last_name' varchar(45) DEFAULT NULL
        'email' varchar(45) DEFAULT NULL

);
```

Load/add data

* Spring Boot: Hibernate is the main JPA by default
* EntityManager: Hibernate component: creating queries;...
  → From JPA!

Automatic data config:

* Based on config: Spring boot automatically creates the beans
* DataSource, EntityManger, ... (we can inject in our app (eg: DAO))

JDBC driver: mysql Connector-J

Spring data (ORM): Spring boot starter data jpa

Config (application.properties)

```
spring.datasource.url = "jdbc:mysql://localhost:3306/student-tracker
spring.datasource.username = springstudent
spring.datasource.password = springstudent.
```

"No need for jdbc driver class name": Spring will do!

Spring boot - command line app

```
@SpringBootApplication
public class CrudApplicationDemo {

    public static void main(String[] args) {

        SpringApplication.run(CrudApplicationDemo.class);
    }
    public CommandLineRunner commandLineRunner(String[] args) {

        return runner -> {

            System.out.println("Hello world");
        };
    }
}
```

Hook: allows to execute the code after spring beans are loaded in the app. context!

"eg: Code to interact with Database"

Dependency: Mysql driver, Spring data JPA

Turn off banner: Spring Skark

```
spring.main.banner-mode = off
```

spring.main.banner-mode = off

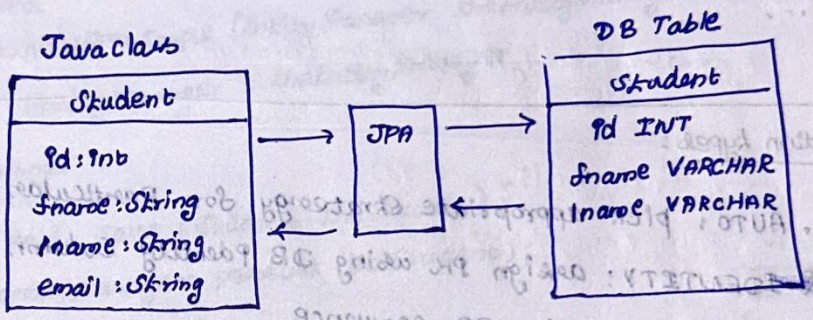
Logs: only warn & errors

```
logging.level.root=warn
```

JPA annotations

★ Entity class: Java class, mapped to a database

ORM (mapping)



@Entity: To mention pk as an Entity (public/protected class)

Note: Constructors:

1. No Constructors: Java will provide
2. If with arg constructor: Java won't give constructor with no arg!

Java annotation:

1. Map class to table → @Entity, @Table(name = "student")
2. Map Column to fields! → @Id, @Column(name = "id")

Note: @Column: optional → (column name: same as Java field)

↳ Not recommended: (Refactored: Broke!)

@Table: Optional (Same name as class): not recommended!

Primary Key:

* must not be null, unique, identifies each row

Auto Increment:

* each time, auto increment!

Primary Key

```
@Entity
@Table(name="Student")
public class Student {
    @Id                                   → Primary Key
    @GeneratedValue(strategy=GenerationType.IDENTITY)   → How it is generated.
    @Column(name="id")
    private int id;                       → By DB managed & maintained
    ...
}
```

ID generation types:

1. AUTO: pick appropriate strategy for particular DB
2. IDENTITY: assign PK using DB identity column (Based on PK Column of DB) — commonly used!
3. SEQUENCE: using DB sequence
4. TABLE: use underlying DB table to ensure uniqueness.

Custom generation strategy:

1. Implement org.hibernate.id.IdentifierGenerator
2. override method: public Serializable generate(...)

Saving Java object with JPA (CRUD)

Student DAO:

* Data Access Object (DAO): responsible for interfacing with DB.
* common design pattern: like helper class to talk with Databases

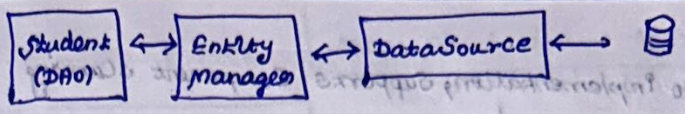
APP ↔ DAO ↔ DB

```
save()
findByID()
findAll()
findByLastName()
update()
delete()
deleteAll()
```

* JPA entity manager: Responsible for save/retrieve entities
* JPA entity manager: needs DataSource (defines DB Connection)

Note: provided by Spring : Based on application.properties

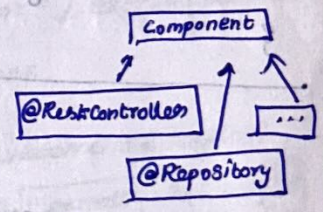
* we can autowire / inject entity manager into our DAO (Student



1. Define DAO interface, Implement it : Inject Entity manager
2. Update main app!

1)
```
import com.luv2code.crudapp.entity.Student;
public interface StudentDAO {
    void save(Student theStudent);
}
```

2)
```
@Repository
public class StudentDAOImpl implements StudentDAO {
    private EntityManager entityManager;
    @Autowired
    public StudentDAOImpl(EntityManager theEntityManager) {
        entityManager = theEntityManager;
    }
    @Transactional
    @Override
    public void save(Student theStudent) {
        entityManager.persist(theStudent);
    }
}
```



@Transactional

* Automagically begin & end a transaction for JPA code.
* Spring handles transaction management.

@Repository (DAO Implementation)

* Sub annotation of Component annotation
* Annotating DAO : Register DAO implementation.
* Spring also provides translation of any JDBC related exceptions.
* While Component scanning: automatically registers DAO implemen..

main app:

```
@Bean
public CommandLineRunner commandLineRunner(StudentDAO studentDAO) {
    return runner -> {
        print("Creating new student");
        Student st1 = new Student("Paul", "Doe", "paul");
        print("Save");
        studentDAO.save(tempStudent);
        print("Generated ID!" tempStudent.getId());
    };
}
```

@Repository:

* Annotation for DAO implementation, supports Component Scanning
* Translates JDBC exceptions!

Primary Keys

* Auto increment: db ↑ id : primary key!
* NOT NULL
* PRIMARY KEY

Change AUTO_INCREMENT values

```
ALTER TABLE Student AUTO_INCREMENT = 3000;
```

Read objects using JPA

```
Student myStudent = entityManager.find(Student.class, 1);
```

```
Public Student findById(Integer id){
    return entityManager.find(Student.class, id);
}
```

Query objects (multiple objects)

* JPQL: Java persistence query language (where, like, order, by, join, in)
* JPQL: Based on entity name & fields (not table names).

→ entityManager.createQuery
(From Student where lastName = 'Doe' OR firstName = 'Daffy', Student.class);

→ List<Student> Students = theQuery.getResultList();

---

email LIKE '%.luv2code.com'

Named parameters

```
public List<Student> findByLastName(String lname) {
    TypedQuery<Student> theQuery = entityManager.createQuery(
        "from Student WERE lastName = :theLname", Student.class);
    theQuery.setParameter("theLname", theLastName);
    return theQuery.getResultList();
}
```

Note: placeholders → with prefix (:) colon

1. New method: findAll()
2. Implement new method

Note: JPQL uses entity name, not table name

JPA

Create: 1) entityManager.persist(Student);

Read: 2) entityManager.find(Student.class, 10001); [Single] → By PK!
3) EntityManager.createQuery(Query, Student.class); [multiple]
theQuery.getResultList();

Update: 4) entityManager.merge(theStudent); [Single object]
5) entityManager.createQuery("UPDATE Student SET lastName='Tester'")
theQuery.executeUpdate();

Delete: 6) entityManager.remove(theStudent);
7) entityManager.createQuery("DELETE from Student WHERE lastName='Smith'").executeUpdate();

Update: generic term (means: modifying database!
→ Returns number of Rows deleted.

8) EntityManager.createQuery("DELETE FROM Student").executeUpdate();
→ delete all rows.

Create database: (running SQL script, Hibernate)

* Java code: JPA, Hibernate annotations (useful for development & testing)

Java code → Hibernate → SQL code → DB

spring.jpa.hibernate.ddl-auto=create
→ while running, drops & creates from scratch

Methods

1. FindById (.find)
2. findAll (CreateQuery)
3. findByLastName (CreateQuery)
4. findByFirstName (CreateQuery)
5. queryForStudentsByLastName(CQ)
6. updateStudent [Retrieve, update, persist]
7. deleteStudent [.remove], [.executeUpdate()]
8. deleteAll()

Database Creation:

* spring.jpa.hibernate.ddl-auto = create (while running drop & create new)
* PROPERTY_VALUES
  * none: no action
  * create-only: only created (DB tables)
  * drop: Database tables are dropped [All data is lost]
  * create: DB tables dropped & created
  * create-drop: drop, create → on shutdown: drop!
  * validate: validate DB tables schema
  * update: update the DB tables schema. (new fields, give updates)!

development: (ease)! : All data lost (every rerun)

spring.jpa.hibernate.ddl-auto = create (Not for production DB)

Production:

spring.jpa.hibernate.ddl-auto = update

(alters schema based on latest code updates! : very careful: Not for big projects)!

For production: DBAs need to run scripts

Use case for create:

* Automatic table generation in memory DBs
* Small/hobby projects.

don't use ddl-auto for production

---

* Corporate DBAs prefer sql scripts → Data governance, code review
* Good for complex designs, version controlled
* Can use migration tools such as Liquibase & Flyway!

Liquibase, Flyway

1. Log Config

* logging.level.root = warn
* logging.level.org.hibernate.SQL = debug (debug logs)
* logging.level.org.hibernate.orm.jdbc.bind = trace (prints actual value!)

↳ Now for all statements (traces are logged)!

spring.jpa.hibernate.ddl-auto = update : Keep the previous data!

Note: only Student is allowed : not ArrayList<Student> : entityManager!

Rest CRUD APIs

Rest APIs: (Spring Rest development)

* API: get weather application for client data! (over HTTP)
* Representational State Transfer (light weigh approach for comm b/w servers)
* Language independent : flexibility.
* Xml, Json (openweathermap.org)

In general : Same!

- Rest API
- Restful API
- Rest web services
- Restful web services
- Rest Services
- Restful Services

Currency Converter app

* Currency converter
* movie tickets app
* CRM app (Customers ...)

Json basics:

* Javascript object notation (plain text), Prog independant
* { "id" : "my" } → Always double quotes (name)
* Name : value pairs
* values : 14, "mario", true, null ... (anything)! , Nested Json object, array.

Spring HTTP

Methods
- POST
- GET
- PUT
- DELETE

Request message
- Request line → HTTP Command, method
- Headers → metadata
- Body → Content

Response line
- headers
- Body

→ Protocol & Server + Status

100-199 : Info
200-299 : Success
300-399 : Redirection
400-499 : Client error
500-599 : Server error

MIME Content type:

* multipurpose internet mail extension
* type / sub-type : text/html, text/plain, application/json...

Need: Client tool, Send HTTP to Rest web services/API,
Tools: postman, curl...

Rest Controller

```
@RestController
@RequestMapping("/test")
public class DemoRestController {
    @GetMapping("/hello")
    public String sayHello() {
        return "Hello world";
    }
}
```

"/test/hello"

Jackson databinding

* Java Json data binding (Json ↔ Java pojo) : Mapping/Serialization - Deserialization
  Marshalling - unmarshalling.
* Spring uses Jackson in behind! (Seperate Project)
* Jackson Data binding:
  * Call appropriate getter, setter

```
{"id": 14,
 "firstName": "mario",
 "lastName": "Ros",
 "active": true
}
```

call setter methods → Java pojo!

"Can't access internal private methods directly"

```
{"id": 14,
 ;
}
```

getter methods ← Java pojo

* Spring automatically handles Jackson integration!
* Pojo returned from REST Controller: Converted to JSON! (Behind the Scene)

Spring Rest Service

→ GET /api/students

* List<Student> from Rest! (need JSON!)
* Spring web starter: has Jackson dependency! : JSON Array!

Coding

Student → firstName
Student → lastName

```
@RestController
@RequestMapping("/api")
Public class StudentRestController {
    @GetMapping("/students")
    public List<Student> getStudents() {
    }
}
```

Path variables

→ GET /api/students/{studentID}

Jackson:
Student ⟷ JSON

@PathVariable (Bind)

```
@RestController
@RequestMapping("/api")
public class StudentRestController {

    @GetMapping("/student/{studentId}")
    public Student getStudent(@PathVariable int studentId) {
        return theStudents.get(studentId);
    }
}
```

(Same name)

Error handling

* Say Customer gives: 200 (Index out of bound)

1. Custom error response class, exception class (update Rest service to throw exception)! (@ExceptionHandler)

StudentErrorResponse
status: int
message: String
timeStamp: Long
getStatus(): int
SetStatus(): void
⋮

```
StudentNotFoundException extends RuntimeException {
    public StudentNotFoundException(String message) {
        super(message);
    }
}
```

↓

Create Custom exception!

Update Rest Service

```
if((StudentId >= theStudents.size()) || (StudentId < 0)) {
        throw new StudentNotFoundException("StudentId NotFound - "+ex);
}
return theStudentList.get(studentId);
```

Handleon (Json!)

@ExceptionHandleon : return ResponseEntity (wrapper obj HTTP response objects)

* It provides fine grained control to specify
  - Http status code
  - Http headers
  - Response body

StudentRestControlleon:

```
@ExceptionHandleon
public ResponseEntity<StudentErrorResponse> handleException(StudentNotFound
                                    Exception ex) {
    StudentErrorResponse ex=new StudentErrorResponse();
    error.setStatus(HttpStatus.NOT_FOUND.value());
    error.setMessage(ex.getMessage());
    error.setTimeStamp(System.currentTimeMillis());
    return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
}
```

↳ Jackson will convert to Json!

1. Throw error! (Service)
2. Handleon handles & parse as Json!

Edge case: Other datatypes

Generic exception

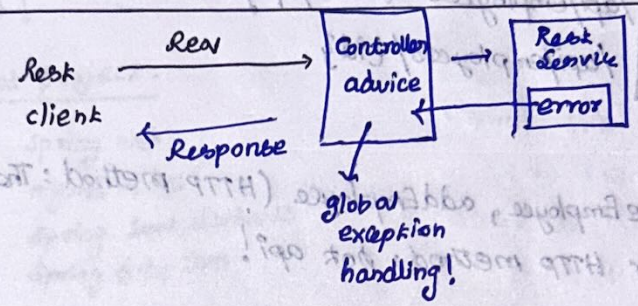
```
@ExceptionHandleon
public ResponseEntity<StudentErrorResponse> handleException(Exception ex)
{
    HttpStatus.BAD_REQUEST;
}
```

Global exception handling

* Specific Rest Controller's : Exception handlers (can't be reused) by other Contro..
* Global handlers : minimum duplications! , Centralize exception handling.

@ ControllerAdvice :

* Similar to InterCeptor / filter : pre-process requests to Controllers (Controllers)
* preprocess / post-process [Handle exceptions]
  Req Responses
* perfect for Global exception handling (Real time app of Aoc]
  Aspect oriented programming [Before, After]



1. Create Rest : @ ControllerAdvice
2. Refactor REST Service : Remove exception handling code.
3. Add exception handling code to @ Controller advice.

1) @ ControllerAdvice
public class StudentRestExceptionHandler {

}

2) Remove @ ExceptionHandler & methods from individual Rest Controllers.

place here!

Best practice

Best practices

1. Who uses? How? Requirement! → Identity
   → Main resource/entity
   → Use Http to assign action on resource.

Requirement:

1. Employee directory: get list, single emp by id, add, update, delete : Full CRUD
2. (prominent noun): Employee (use plural for endpoints)

/api/employees

POST	Create	/api/employees
PUT	Update	/api/employees/{id}
GET	list/single entity	/api/employees (or) /api/employees/{id}
DELETE	delete entity	/api/employees/{id}

Bad habit:

* No actions: deleteEmployee, addEmployee (HTTP method: That's why)
* Pass behaviours over HTTP method: not api!

More examples

paypal (Invoicing):

POST	/v1/invoicing/invoices
GET	/v1/invoicing/invoices
GET	/v1/invoicing/invoices/{invoice_id}
PUT	/v1/invoicing/invoices/{id}
DELETE	/v1/invoicing/invoices/{invoices_id}

Github

POST	/user/repos
DELETE	/repos/:owner/:repo
GET	/repos/:owner/:repo
GET	/user/repos

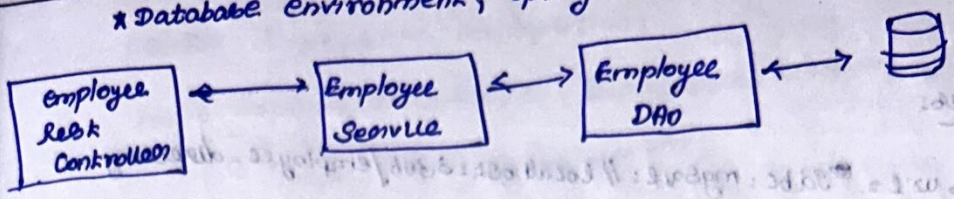
Salesforce:

GET	/services/apexrest/v1/individual
GET	/{id}
POST	/individual
PUT	/individual/{id}

Real time project

Requirement:

* Database environment, Spring boot



Setup table:

Employee:
- id (INT 11)
- f_name (45)
- last name (45)
- email (45)

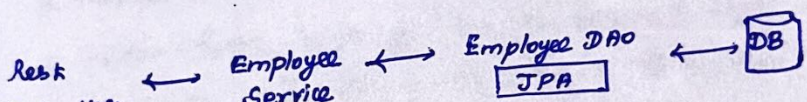
] VARCHAR

1. create DB !

2. Spring Boot project:

- spring web
- mysql driver
- spring boot devtools
- spring data JPA

JPA DAO: Spring boot

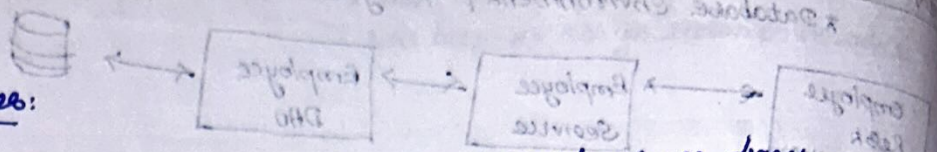



```
@Repository
public class EmployeeDAOJPAImpl implements EmployeeDAO {

    private EntityManager entityManager;

    @Autowired
    public EmployeeDAOJPAImpl(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    ...

}
```

Employee:
- get list, get employee
- add
- delete
- update

Settings > Build, Exe, Dep > Compiler > Build project automatically : Devtools
2) Adv Settings > Allow auto-make to start...

JDBC:



application.properties:

```
Spring.datasource.url = jdbc:mysql://localhost:3306/employee_directory
Spring.datasource.username = root
Spring.datasource.password =
```

entity.Employee.java

```
@Table (name="employee")
@Entity
public class Employee {
    @Id @GeneratedValue (Strategy = GenerationType.IDENTITY) @Column(name="id")
    private int id;
    @ Column (name="first_name")
    private String firstName;
    @Column (name = "last_name")
    private String lastName;
    @Column (name = "email")
    private String email;
}
```

dao.EmployeeDAO:

```
public Interface EmployeeDAO {
    List <Employee> findAll();
    :
}
```

dao.EmployeeDAOJPAImpl:

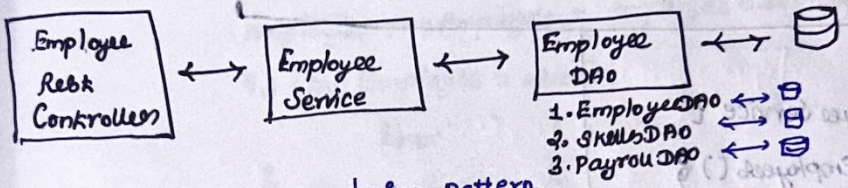
```
@Repository
public class EmployeeDAOJPAImpl implements EmployeeDAO {
    @Override
    public List <Employee> findAll () {
        ____
        ____
    }
    private EntityManager entityManager;
    @Autowired
    Public EmployeeDAOJPAImpl (EntityManager entityManager) {
        this.entityManager = entityManager;
    }
```

```
TypedQuery<Employee> query
= entityManager.createQuery
("from Employee",
Employee.class);
return query.getResultList();
```

Rest Controller: (Refactor: Inject from service layer : after)

```
@RestController @RequestMapping("/api/student")
public class EmployeeRestController {

    private EmployeeDAO employeeDAO;

    public EmployeeRestController ( EmployeeDAO theEmployeeDAO) {

        employeeDAO = theEmployeeDAO;

    }

    @GetMapping ("/employees")
    public List <Employee> findAll () {

        return employeeDAO.findAll();

    }

}
```

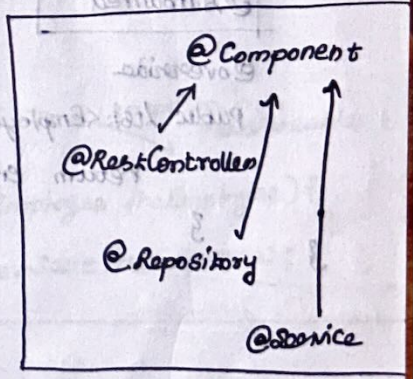
Spring Boot Service Layer



* Service Facade design pattern
* Intermediate layer for custom business logic
* Integrate data from multiple sources (DAO/repositories)

Controller (single view) ← data : Integrated from multiple backend datasource!

@Service



@Service

* Service Implementations (automatically registered by spring) [Component Scanning]

Steps:

1. Define Service Interface
2. Define Service Implementation (Inject EmployeeDAO)

```
@Service
public class EmployeeServiceImpl implements EmployeeService {

    Inject EmployeeDAO

    @Override
    public List<Employee> findAll () {
        return employeeDAO.findAll();
    }

}
```

Service layer

> dao
> rest
> service → EmployeeService
> entity → EmployeeServiceImpl

RestController: (EmployeeRestController) [Refactoring to Service layer]

```
Public Class EmployeeRestController {

    private EmployeeService employeeService;

    @Autowired : Constructor!

    @GetMapping("")
    public List<Employee> getEmployees() {
        return employeeService.getEmployees();
    }
}
```

Service:

```
public Interface EmployeeService {

    List<Employee> getEmployees();
}
```

```
@Service
public Class EmployeeServiceImpl implements EmployeeService {

    private EmployeeDAO employeeDAO;

    @Autowired

    @Override
    Public List<Employee> getAll() {
        return employeeDAO.findAll();
    }
}
```

Service layer: best practice

* Apply transactional boundaries at service layer
* It is the service layer's responsibility to manage transaction boundaries
* For implementation code
  1. Apply @Transactional on service methods
  2. Remove @Transactional from DAO methods (if exists)

Shift @Transactional annotations from DAO to Service

Get emp by id ⟶ entityManager.find(Employee.class, id);
Add emp ⟶ entityManager.merge(theEmployee) [return Employee: updated id]
Update ⟶ ⟶ Save/update!
Delete ⟶ find, entityManager.remove(employee);

Save, update, delete : @Transactional : modify DB.
↓
In service layer!

Send Json:

* Set header (HTTP) : Content-Type : application/json

Rest:

```
@GetMapping("/employees/{employeeId}")
public Employee getEmployee(@PathVariable int employeeId) {
    Employee theEmployee = employeeService.findById(employeeId);
    if (theEmployee == null) {
        Error
    }
    return theEmployee;
}

@PostMapping("/employees/{employeeId}")   → optional
public Employee addEmployee(@RequestBody Employee theEmployee) {
    Employee dBEmployee = employeeService.save(theEmployee);
    return dBEmployee;
}

@PutMapping("/employees/{employeeId}")   → mention in PathVariable
public Employee updateEmployee(@RequestBody Employee theEmployee) {
    Employee dBEmployee = employeeService.save(theEmployee);
    return dBEmployee;
}

@DeleteMapping("/employees/{employeeId}")
public String deleteEmployee(@PathVariable int employeeId) {
    Employee tempEmployee = employeeService.findById(employeeId);
    if (tempEmployee == null) Error;
    employeeService.deleteById(employeeId);
    return "deleted Id: " + tempEmployee.
}
```

Spring Data JPA

Previous:

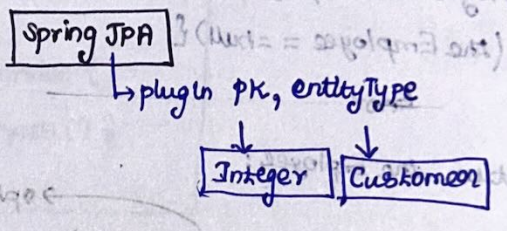
* Employee ⟶ created DAO.
* Problem: many entities (Student, Product, Book...)
* Code replikatation.

```
@Override
Public Employee findById(int id) {
    Employee data = entityManager.find(Employee.class, id);
    return data;
}
```

→ Repeat! (why? → Solve using Spring JPA)

why?

* Spring creates DAO : if will give PK, entityType : Spring gives CRUD methods
  eg: findbyAll(), findById(), save(), deleteById(), others...



JPA Repository:

* From Spring (Interface): Provides basic methods.

```
public Interface EmployeeRepository extends JPARepository<Employee, Integer>
{
    No need: parent methods: These
}
```

Available methods:

* use in app

```
private EmployeeRepository er;
@Autowired
public EmployeeServiceImpl(EmployeeRepository er) {
    this.er = er;
}
```

→ er.findAll()

Advantage:

* Reduce code, Support Custom Queries with JPQL
* Query Domain Specific Language, Custom methods: Allowed.

Note: @Transactional (JPA repository gives this by default)

Difference:

* methods like findById returns Optional <...>
* Use: No need to check for null!

```
if (result.isPresent()) {
    return result.get();
}
else
    return null;
}
```

Format:

> DAO
  EmployeeRepository (interface) : nothing!
> Rest
> Service (change entityDO ⟶ employeeRepository)
> entity.

Spring data Rest

> Rest Controllers.
> Service → interface Impl
> EmployeeRepository

* Same thing like Spring JPA applies to Rest?
* Give all basic CRUD features! (Spring gives Rest CRUD implementations)!

Working:

1. Scans for JPA Repository, exposes entityType (pluralize, lowerCase, add 's')

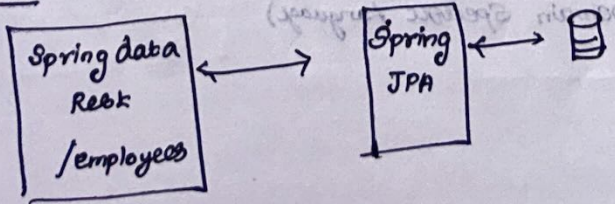
public interface EmployeeRepository extends JPARepository <Employee, Integer> {

}

/employees (Can Customize)

Pom: spring-boot-starter-data-rest (artifactId)

Now:



HATEOS

★ HATEOS Compliant (Hypermedia as the Engine of App. State)

★ Provides Info to access REST Interface (meta data for Rest data)

eg: `/api/employees/3 : GET`

```
{
  'fname': ———— ,
  'lname': ———— ,
  'email': ———— ,
  '_links': {
      'self': { 'href': 'https://localhost:8080/employees/3' },
      'employee': { 'href': 'https://localhost:8080/employees/3" }
  }
}
```

→ response metadata (links to data!)

GET: /api/employees

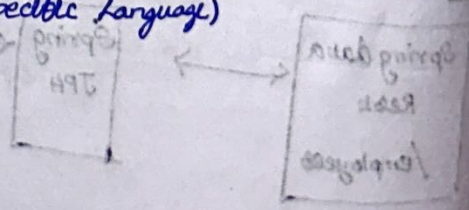
```
{
  '_embedded': {
      'employees': [ {
          'fname': —— ,
          'lname': —— ,
          'email': —— ,
      } ]
  },
  'page': {
      'size': 20,
      'totalElements': 5,
      'totalPages': 1,
      'number': 0
  }
}
```

About the page!

★ HATEOS uses HAL (Hypertext App Language) for data formats

Advanced features:

1. pagination, sorting, searching
2. Extending & adding custom queries with JPQL
3. query DSL (query Domain Specific Language)



spring.data.rest.base-path = "/api"

Rest: Config: Pagination, Sorting

* Basic endpoind: From EntityType : simple! (put 's')
  eg: Employee ⟶ /employees

* manually set : different resource name
  plural name
  eg: Employees ⟶ /members

can't handle.
* Goose ⟶ Geese
:

② RepositoryRestResource(path = 'members')
public interface EmployeeRepository extends JpaRepository<Employee, Integer> {
}

Pagination:

* Default: Returns 20 elements (Spring Data Rest)
* Navigate to different pages: using query param

https://localhost:8080/employees?page=0

spring.data.rest.base-path : base path used to expose repo resources
spring.data.rest.default-page-size : default page size
spring.data.rest.max-page-size : max size of pages.

Sorting:

our case: firstName, lastName, email

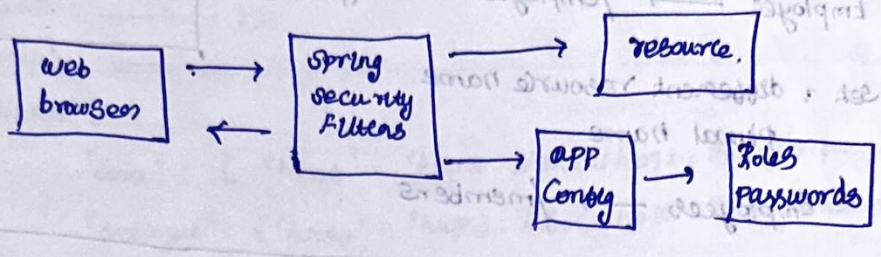
https://localhost:8080/employees?sort=lastName (ASC: default)

?sort=lastName,desc
,asc

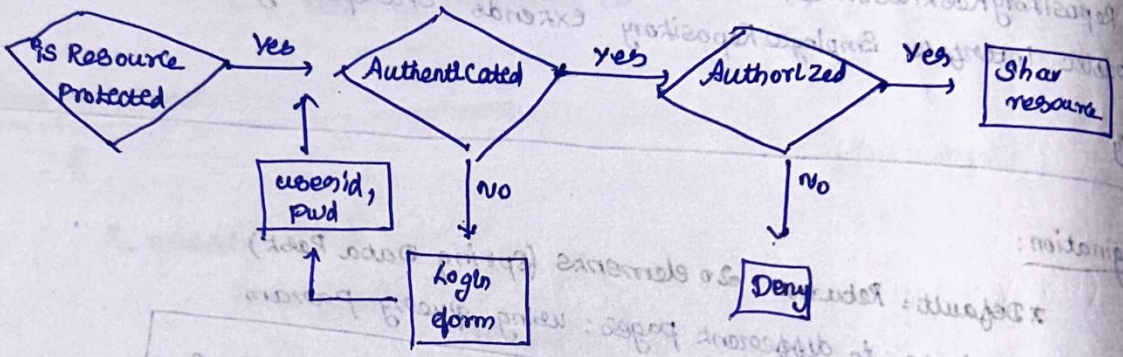
?sort=lastName,firstName,asc.

Spring Security

* Secure API, Users & Roles, password (plain/encrypted)
* Implemented using Servlet filters in background. (pre-process / post-process of web requests)
* 2 methods of securing an app → declarative
  → programmatic
* Servlet filters: Can route web requests based on security logic.



Flow:



* Authenticated: user id, pwd
* Authorized: Role,

`@Configuration` : define app's security constraints

Spring:

* Customize, custom coding.

`Spring-boot-starter-security` : Secure all endpoints by default.

Override:

```
spring.security.user.name = Scott
spring.security.user.password = pwd
```

In memory
DB (JDBC)
LDAP
Custom plug in
others..

Users:

User id	Password	Roles

1) @Configuration, Add user, pwd, roles

```
@Configuration
public class DemoSecurityConfig {
    // add Security config
}
```

Spring password storage

{id} encodedPassword

1) noop (plain text)

2) bcrypt (Bcrypt algo: one way algo): hashing algo

{noop}test123

ToDO

- Logout
- Access denied page

```
@Configuration
public class DemoSecurityConfig {

    @Bean
    public InMemoryUserDetailsManager userDetailsManager(){

        UserDetails john = User.builder()
            .username("john")
            .password("{noop}test123").roles("Employee")
            .build();
        return new InMemoryUserDetailsManager(john, Mary, Susan);
    }
}
```

Roles : Restrict

GET	/api/employees	Read all	EMPLOYEE
GET	/api/employees/{employeeID}	Read Single	EMPLOYEE
POST	/api/employees	Create	MANAGER
PUT	/api/employees	update	MANAGER
DELETE	/api/employees/{empID}	DELETE emp	ADMIN

```
requestMatchers (<< add Path to match on >>)
    .hasRole (<< authorized Role >>)
```

→ POST/ GET/ PUT/ DELETE , /api/employees

```
requestMatchers (method, path)
    .hasAnyRole (list of roles);
```

METHOD, Endpoint

requestMatchers (HttpMethod.GET, "/api/employees").hasRole ('EMPLOYEE')

"/api/employees/**"

↳ All (wild card) : All Subpaths

```
@Bean
Public SecurityFilterChain filterChain (HttpSecurity http) throws Exception
{
    http.authorizeRequests (Configuer →
        Configurer
            • requestMatchers (HttpMethod.GET, "/employees").hasRole ('EMPLOYEE')
              GET, "/employees/**").  'EMPLOYEE')
              POST, "/employees").  'MANAGER
              PUT,  "               MANAGER
              DELETE, "/**")        ADMIN

    http.basic (Customizer.withDefaults())

    http.httpBasic(); http.csrf().disable();
    return http.build();
}
```

↳ stateless REST API (not required)

CSRF

* Spring provides protection against CSRF.
* Embed data (additional authentication) /token into all HTML forms.
* Subsequent requests: web app verifies token before processing

Use case : Traditional web app (HTML forms)

Browser based web requests :

* Recommended : CSRF protection (Traditional web app with HTML forms to add/modify data)
* Non browser : Need to disable CSRF protection.

CSRF – Not required for Stateless Rest API's : POST/PUT/PATCH/DELETE

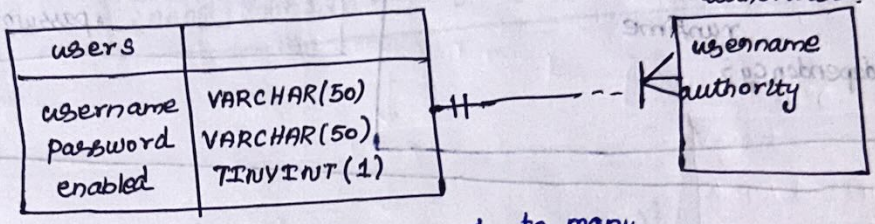
```
http.csrf().disable();
```

* Use predefined schema tables. (spring takes care all the heavy lifting)
* Can customize table schemas (use: custom projects)
* Responsible for accessing data: write JDBC code / hibernate code.

eg: Account, user role

1. Sql Script (DB)
2. Pom: DB support
3. Create JDBC file
4. Make Spring Security to use JDBC

Default Schema



* authorities means roles

```
1. CREATE TABLE `users` (
        `username` VARCHAR(50) NOT NULL
        `password` VARCHAR(50) NOT NULL
        `enabled` tinyint NOT NULL,
        PRIMARY KEY (`username`)
    ) ENGINE = InnoDB DEFAULT CHARSET = latin1;

INSERT INTO `users` VALUES
    (`john`, `{noop}test123`, 1),
    (`mary`, `{noop}test123`, 1),
    (`susan`, `{noop}test123`, 1);

CREATE TABLE `authorities` (
        `username` varchar(50) NOT NULL
        `authority` VARCHAR(50) NOT NULL,
        UNIQUE KEY `authorities_idx_1` (`username`, `authority`),
        CONSTRAINT `authorities_fk_1`
        FOREIGN KEY (`username`)
        REFERENCES `users` (`username`)
    ) ENGINE = InnoDB DEFAULT CHARSET = latin1;
```

```
INSERT INTO 'authorities'
VALUES
    ('John', 'ROLE_EMPLOYEE'),
    ('mary', 'ROLE_EMPLOYEE'),
    ('mary', 'ROLE_MANAGER'),
```

→ For Spring internal purposes

ROLE_

Susan → emp
→ man
→ ROLE_ADMIN

```
<dependency>
    com.mysql
    mysql-connector-j
    runtime
</dependency>
```

```
spring.datasource.url = jdbc:mysql://
    localhost:3306/emp_db
    .username =
    .password =
```

```
@Configuration
public class DemoSecurityConfig {

    @Bean
    public UserDetailsManager userDetailsManager(DataSource dataSource) {

        return new JdbcUserDetailsManager(dataSource);

    }

}
```

↳ No hard coding!

Automatically injected by Springboot

Spring uses JDBC Authentication!

(Spring security: assumes using predefined table, row format)

Note: Each time, retrieved from DB! validated

(Any update in DB: No need to restart app as intended)

Encrypt Password

* Spring team: Recommends bcrypt algo (1-way encrypted hashing)
* Applies random salt, defeats brute force attacks

password-hashing-best-practices

* website utility for encrypt/decrypt (or) Java code.

Salting: Random bits (To make hash unique)

1. modify DDL: Length 68 (password)

{noop}john123 → {bcrypt} hash

Steps:

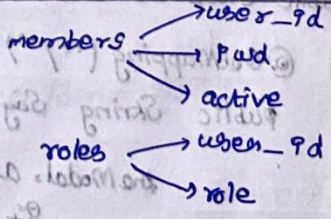
1. Retrive
2. Decrypt → No! (Instead encrypt pwd received)!
3. match

bcrypt: 1-way algorithm (unable to decrypt)!

§char: {bcrypt}
&char: encrypted text

Update DDL, Spring takes care! (Remaining): no need to do!

Custom tables



* own table, own column names
  1. Tell how to query
  2. find user by user_name
  3. find authority/roles by user name

```
@Bean
Public UserDetailsManager userDetailsManager(DataSource ds) {
    JdbcUserDetailsManager um = new JdbcUserDetailsManager(ds);
    um.setUsersByUsernameQuery("Select user_id, pw, active from
        members where user_id = ?");
    um.setAuthoritiesByUsernameQuery("Select user_id, role from roles
        where user_id = ?");
    return um;
}
```

Change logging to DEBUG : To debug!

Thymeleaf

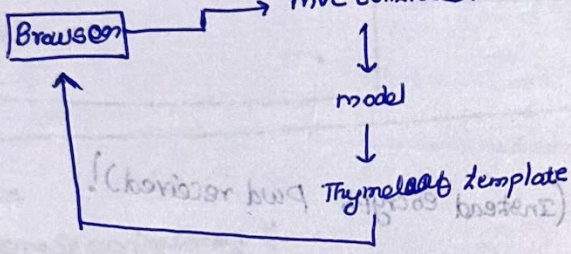
* Java template engine: HTML view generate for web apps
* General purpose templating engine. (unrelated to spring. so) → no need for Spring
* Sound: tymeleaf ('h' silent)

Thymeleaf template:

* HTML page with thymeleaf expressions
* Includes dynamic content from thymeleaf expressions
* Can access Java code, objects, Spring beans.

Where Thymeleaf processed:

* processed on server: Result HTML!



Process:

1. Add thymeleaf pom
2. Develop MVC
3. Create Thymeleaf template

```
<dependency>
    groupId: org.springframework.boot
    artifactId: spring-boot-starter-thymeleaf
```

MVC

```
@Controller
public class DemoController {
    @GetMapping("/")
    public String sayHello(Model theModel) {
        theModel.addAttribute("theDate", new java.util.Date());
        return "theModel";
    }
}
```

→ Pom: Spring auto configures thymeleaf & look for this template!

src/main/resources/templates/helloworld.html

Thymeleaf templates : /src/main/resources/templates

helloworld.html

```
<!DocType Html>
<html xmlns:th="http://www.thymeleaf.org">
<head> ... </head>
<body>
    <p th:text="'Time on the Server is ' + ${theDate}" />
<body>
</html>
```

From model !

"theDate"

Supports → Looping, Conditionals
→ CSS, JS integration
→ Template Layouts & Fragments

CSS & Thymeleaf

* Local CSS / Referencing CSS

src/main/resources/static → JS, CSS → demo.css, Images

(Any subdirectory under /static)

```
.funny {
    font-style: italic;
    color: green;
}
```

```
<head>
    <!-- reference css file -->
    <link rel="stylesheet" th:href="@{/css/demo.css}" />
</head>

<body>
    <p th:text="'Time on Server is ' + ${theDate}" class="funny" />
</body>
</html>
```

Commonly used static resources:

/META-INF/resources
/resources
/static
/public

Search order: top-down.

Bootstrap (Download / CDN)

```
<link rel='StyleSheet' th:href='@{/css/bootstrap.min.css}'/>
```

(or)

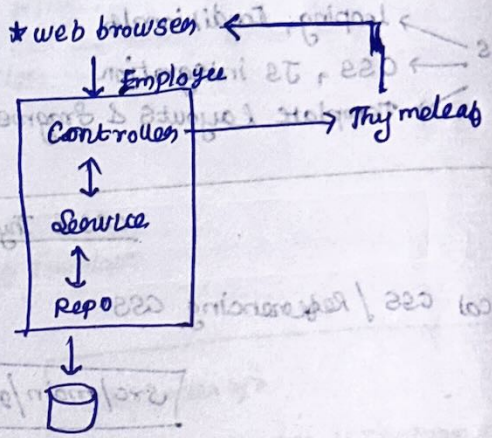
```
<link rel='StyleSheet' href='https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css'/>
```

View

EMPLOYEE DIRECTORY

Add Employee

First Name	Last Name	Email	Action
Leslie	Andrews	leslie@_	update delete
Emma	Gupta	/	update delete
Yuri	Petrov	/	u
Juan	vega	/	"



1. Employee Repository, Employee entity, Controllers, Thymeleaf!
2. List, add, update, Delete.
3. Redirect to list (/index.html)! (default)!

/index.html : (default)

```
<meta http-equiv='refresh' content='0; URL='/employees/list'/>
```

Add employee:

Employee Directory

Save Employee

First name

Last name

Email

Save

Back to Employees List

1. Html form
2. Button
3. Server!

Spring Form

* In Spring Controllers → Before showing form: must add a modal attribute
  → model : holds data for data binding!

```
@Controller
@RequestMapping("/employees")
public class EmployeeController {
    @GetMapping("/showFormForAdd")
    public String showFormForAdd(Model theModel) {
        Employee theEmployee = new Employee();
        theModel.addAttribute("employee", theEmployee);
        return "employees/employee-form";
    }
}
```

Thymeleaf & Spring MVC Data binding

* Thymeleaf has special expressions for binding Spring MVC form data.
* Automatically set/retrieve data from Java object.

* th:action ⟶ Location to send form data
* th:object ⟶ Reference to model attribute
* th:field ⟶ Bind i/p field to a property on model attribute

```
<form action="#" th:action="@{/employees/save}"
      th:object="${employee}" method="POST">
    :
</form>
```

```
<input type="text" th:field="*{firstName}" placeholder="First name">
```

Selects property on referenced th:object

Load:
1. getFirstName()
2. getLastName()

Submit:
1. setFirstName()
2. setLastName()

```
@Controller
public class EmployeeController {

    private EmployeeService employeeService;

    public EmployeeController(EmployeeService employeeService) {
        this.employeeService = employeeService;
    }

    @PostMapping("/save")
    public String saveEmployee(@ModelAttribute("employee") Employee employee) {
        employeeService.save(theEmployee);
        return "redirect:/employees/list";
    }
}
```

→ prevent duplicate submissions!

Sort:

JPA: `EmployeeRepository.findAllByOrderByLastNameASC();`

Update button:

Employee Directory

Update Employee

Trupti

Sampthi

email

update

Back to Employee List

Pre-populate data

update!

/employees/showFormForUpdate/id

ID: can't update!

'use loop!' : Thymeleaf!

Delete:

★ delete (Prompt: Delete?)

★ Redirect to employee list!

@RestController : Return JSON / XML

@Controller : For HTML templates

Update:

1) hidden: Up button: Id, Submit form → use that to update (POST)

2) get Mapping: URL param! EmployeeID

use ajax!

Do: PUT, POST...

Delete:

1) get mapping

2) POST (Id: hidden)

Validation

beanvalidation.org

* Required field
* Valid numbers: Range
* Format (ZIP Code)
* Business rule.

Java Standard Bean Validation API

* Define meta data model & API for entity validation
* Not tied to web tier / persistence tier.
* Server side / client side: JavaFx / Swing apps!

Construct once, use everywhere

* Spring 4+ : Supports Bean Validation API : preferred method! (For Spring apps)
* Add validation jar, annotate, use!

* required
* Validate length
* Numbers
* Regex
* Custom Validation

Validation annotation:

@NotNull
@Min → >=
@Max → <=
@Size → given size
@Pattern → Regex
@Future / @Past → date (future/past) of given date.
Others..

Java Standard Bean Validation API

* Only a Standard (JSR-303) : Vendor Independent, portable
* Implementation: Hibernate (Initiated as ORM project, recent years: expanded)
* Now fully compliant JSR-303 implementation.
* Hibernate Validator: Seperate project!

http://www.hibernate.org/validator

* Jakarta EE: Community version of Java EE (rebranded, relicensed)
* Collection of API (enterprise): Servelat, JDBC, Java Beans--
* Not controlled by Oracle: managed by Community
* Java EE: Seperate (oracle)!

Jakarta EE:

* package renamed: javax.* → jakarta.*
* Hibernate 7 based on Jakarta EE 9. ← not compatible with Hibernate validator 7!
* Spring 5 still uses Java EE!
* Hibernate 6.2: same features as 7 (compatible)
  ↳ uses javax.* packages!

JARS
> web-INF > lib

Spring MVC: Form Validation: Required

* Last Name: Required!
* Add validation, show error message on HTML, validation in Controller, Confirmation page

1. Add validation:

```
Customer {
    public String firstName;
    @NotNull (message="is required")        → To display for error message
    @Size (min=1, message="is required")      (default: may not be null)
    private String lastName;
    :
}
```

2. `<form:input path="lastName" />`
   `<form:errors path="lastName" cssClass="error" />` → display error message
   ↳ Name of CSS class

LastName (*): [ ] is required

3.
```
@RequestMapping ("/processForm")
public String processForm (
        @Valid @ModelAttribute ("customer") Customer customer,
        BindingResult bindingResult) {          → Result of validation rule!

    if (bindingResult.hasErrors()) {
        return "customer-form";
    }
    else {
        return "customer-confirmation";
    }
}
```

Spring calls validation rules (Hibernate Validator API)

result: bindingResult

4. Confirmation-page

<body> The Cx is confirmed ${customer.firstName} </body>

Note: bindingResult must be immediately after model attribute!
Else ignored.

error.css

```
<style>
.error {
    color: red;
}
</style>
```

Inside HTML

Page:

First name: [ ]

last name(s): [ ]

Submit

White Space:

* Trim whitespace! (@InitBinder: preprocess each web request to the Controller): method annotated with @InitBinder is executed!
* Leading & Trailing whitespace!

CustomerController.java

```
@InitBinder
public void initBinder(WebDataBinder dataBinder) {
    StringTrimmerEditor stringTrimmerEditor = new StringTr... (true);
    dataBinder.registerCustomEditor(String.class, stringTrimmerEditor);
}
```

→ Trims space (strings)! Leading/Trailing

: true means ⟶ null if all white space!

→ Every String class applies this trimmer!

Note: This uses Server side (Server returns page with error message! Instead → we can handle in client (no request!)

@min, @max

```
Customer {
;
@min (value = 0, message = "> 0")
@max (value = 10, message = "< 10")
Private Int freeParses;
}
```

2) `<form:errors path="freeParses" cssClass="error" />`

3) error → Customer_form
Success → Conformation page!

Custom validations

* Array of Strings

* `public String[] value() default {"LUV"};`

```
@Override
public boolean isValid(String code, ConstraintValidatorContext constraintValidatorContext)
{ boolean flag = false;
  if (code != null) {
      for(String code : String) {
          result = String.startsWith(code);
          if (result) {
              break;
          } } else { return true; }
      return false; }
```

eg: startsWith, endsWith!

Regex

* Search Pattern: regex! eg: `"^[a-zA-Z0-9]{5}"`
  ↳ only 5 char/digits!

```
@Pattern (regexp = "^[a-zA-Z0-9]{5}", message = "only 5 char/digits")
Private String PostalCode;
```

Integer field required

```
@NotNull (message = "is required")
@Min (value = 0, message = "Greater than 0")
@Max (value = 10, message = "Less than 10")
Private Int freeParses;
```

↓

Error (Failed to convert property value of type java.lang.String to [Int]

"NumberFormatException"

Solution: Int can't be null (Integer!)

Handle String Input for Integer

* Error: Can't convert to Integer!
* Custom error message!

message.properties → /resources/messages.properties

typeMismatch.Customer.freePasses = Invalid number

- typeMismatch → Error type
- Customer → attribute name
- freePasses → field name
- Invalid number → error message

Custom Validation

* Course Code starts with "LUV" ; Custom Validator (returns true/false)

@CourseCode

@CourseCode (value = "LUV", message = "must start with LUV")
private String courseCode;

@CourseCode:

1) @Constraint (validatedBy = CourseCodeConstraintValidator.class)
   @Target ({ElementType.METHOD, ElementType.FIELD}) → (Annotation) Applied to method, field.
   @Retention (RetentionPolicy.RUNTIME) → Java bytecode (use, introspect, instrument on run time).
   public @interface CourseCode
   {
   public String value() default "LUV";
   public String message() default "must start with LUV";
   public Class<?>[] groups() default {};
   public Class<? extends Payload>[] payload() default {};
   }

2) CourseCodeConstraintValidator:

public class CourseCodeConstraintValidator implements ConstraintValidator<CourseCode, String>
{
private String coursePrefix;
@Override
public void initialize (CourseCode cc) {
coursePrefix = cc.value();
}
@Override
public boolean isValid (CourseCode cc, ConstraintValidatorContext cvc) {

return cc.startsWith(courseCodePrefix);
}

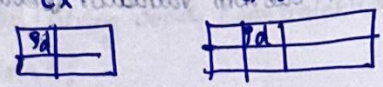
Advanced mappings

* One to one
* One to many
* Many to many

PK: Unique row in a table
FK: Link tables (PK of linked table)

Cascade:

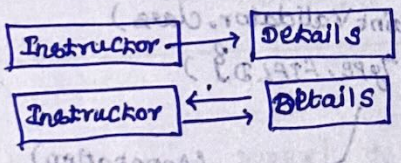
* Delete Customer ⟶ Delete his detail also



Cascade delete! (use case!)

Eager loading: Retrieve everything
Lazy: Only on demand.

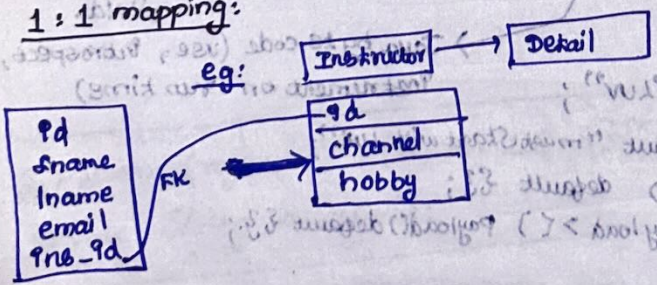
Unidirectional relationship:



Why FK
* Referential Integrity (saves illegal ops)
* Valid data inserted! (valid reference)

1:1 mapping:

eg:



Java Spring Boot

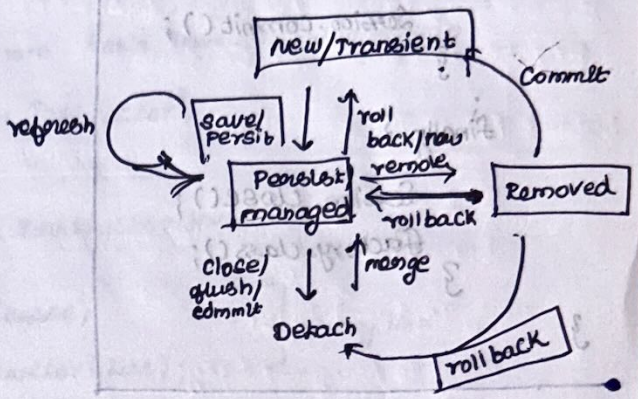
```
@OneToOne
@JoinColumn(name = "instructor_detail_id")
private InstructorDetail instructorDetail;
```

Note: Hibernate can go both ways, use FK: Load data accordingly

Instructor + details

Entity lifecycle:

* Detatch : Not associated with hibernate session
* Merge : attach an entity to a session which is detatched.
* persion : Transitions new instances to managed state. (Next flush/commit will save in dB)
* Remove : Take managed entity to be removed. Next Flush/Commit removes from DB
* Refresh : Reload/Synch object with data from DB. prevents stale data



Cascade types:

* Save A → also B
* Refresh A → also B
* Remove A → also B!

```
@OneToOne (cascade = CascadeType.ALL)
private InstructorDetail instructorDetail;
```

Cascade = { CascadeType. DETACH
MERGE
PERSIST
REMOVE
REFRESH }.

```
SessionFactory factory = new Configuration()
    .configure ("hibernate.cfg.xml")
    .addAnnotatedClass (Instructor.class)
    .buildSessionFactory();
try {
    Session.beginTransaction();
    Session.save (tempInstructor);
    Session.getTransaction().commit();
}
Session session = factory.getCurrentSession();
```

```
session.delete (tempInstructor);
```

→ Also delete details object!

```
@Entity
@Table(____)
public class name ...
{ . . .
}
```

Bidirectional

* @OneToOne(mappedBy = "instructorDetail", cascade = CascadeType.ALL)
  private Instructor instructor; ⟶ InstructorDetail.class

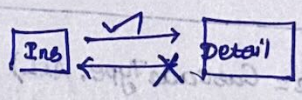
* Note: delete detail ⟶ delete Instructor!

Prevent leaking:

```
try {
    Session.commit();
}
finally {
    Session.close();
    Factory.class();
}
```

Delete detail, Keep Instructor!

* Set Instructor of details class to null



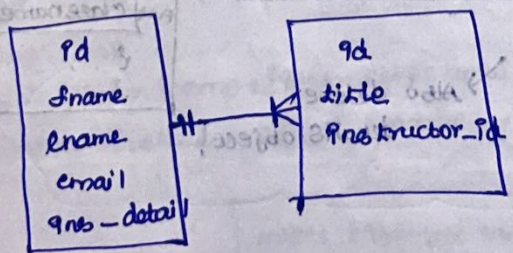
* Now Session.delete(detailsObject); ⟶ Only deletes details
* Why: Link is broken!

---

one to many mapping

* Instructor has multiple Courses (Bi-directional)
* many Courses : 1 Instructor (atleast)

Note: Delete Instructor: No Cascading delete of Courses



assume 1 ins, many Course
1 Course, 1 instructor (usually not: assume)

@ Many To One

@ Join Column (name = "instructor_id") → class → Many Course 1 Instructor!
private Instructor instructor; (Courses)

Instructor. class

@ one To Many (mapped By = "instructor") Cascade = { CascadeType. persist . . . })
private List <Course> courses;

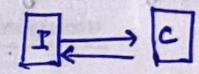
Mapped By:

* Look at instructor property from table info. of Join Column
* find associated courses for an instructor!

Save : Instructor.java

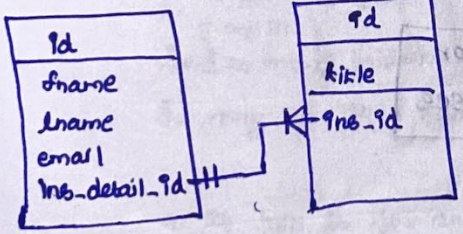
add (Course tempCourse) {
Courses. add (temp Course)
temp Course. set Instructor (this);
}

2 way link

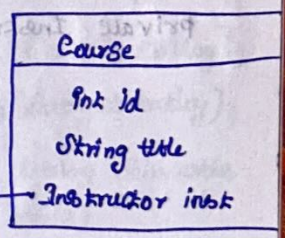


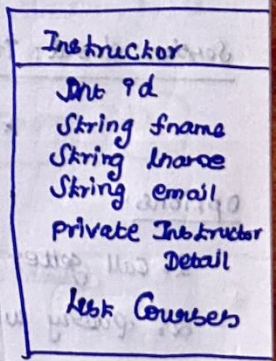
Use Join Column in the table that owns the foreign key column

Our case:



Course!
Owns
FK





Eager loading

* Load all dependent entities all at once
* Large data: Impact performance
(get all Courses of a Student)!
(Load Course, Load all Students)!
↳ Not needed! (only list of Courses)

Best practice

only load data when absolutely needed

Lazy loading:

* on demand!

usecases

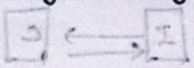
* List of Instructors: Lazy loading
* Detail View of an Instructor: Eager loading! (Instructor + Courses)!

Instructor

```
@OneToMany(fetch=FetchType.LAZY, mappedBy="instructor")
Private List<Course> courses
```

Default Fetch Types:

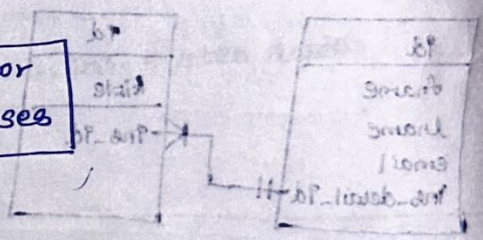
1:1 → Eager
1: many → Lazy
many To 1 → Eager
many To many → Lazy



Course

```
@ManyToOne(fetch=FetchType.LAZY)
@JoinColumn(name="instructor_id")
private Instructor instructor;
```

Many Courses: 1 Instructor
1 Instr : many Courses



Session closed: Trying lazy loading throws error!

* still access data! → load when session is open (next time use (not fetched from DB)!

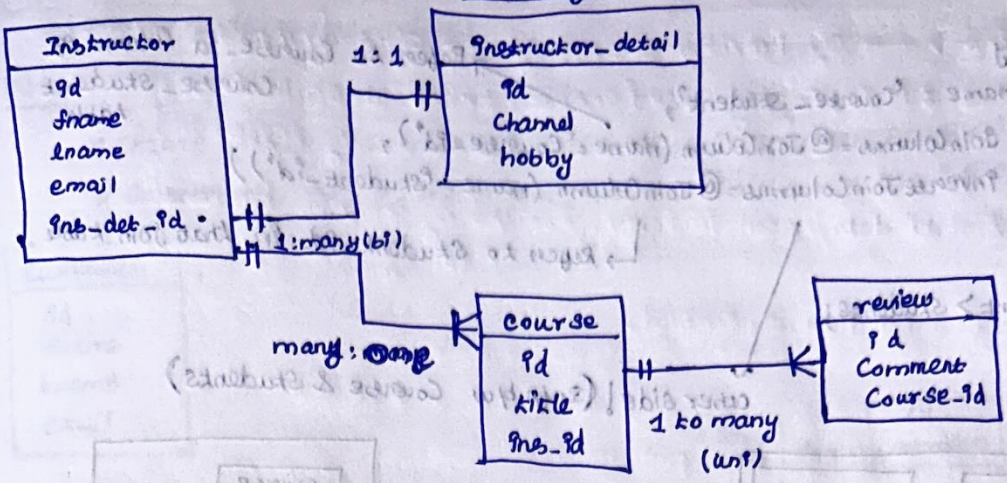
Options:

1. Call getter method (tempInstructor.getCourses()) while session is open!
2. Query with HQL!

```
Query<Instructor> query = session.createQuery
                ("SQL query", Instructor.class);
query.setParameter("id", id);
Instructor ins = query.getSingleResult();
```

Load whenever needed using HQL!

OneToMany: Unidirectional



Courses.java (1 Course many reviews) → Lazy, ALL

```
@OneToMany @JoinColumn(name = "Course_id")
List<Review> reviews;
```
(review table)

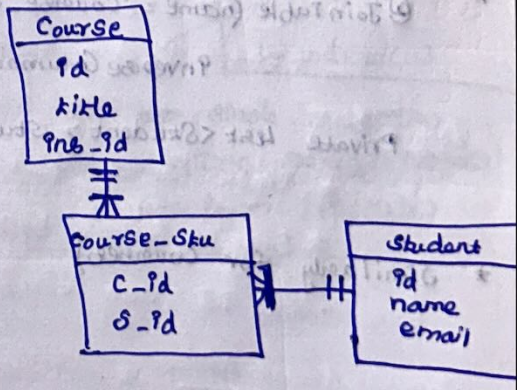
How hibernate finds Column @JoinColumn

* @JoinColumn (name = "course_id") ⟶ Course.java (don't have this column)
* How does @JoinColumn knows where to find? ⟶ Complex + Adv. steps
  1. 1 To 1 / many To 1 ⟶ uses FK strategy [FK is in table of source entity]
  2. 1 to many (unidirectional): FK (FK is in table of target entity)
  3. many to many / Bidirectional many to one / one to one using Join table (uses FK in join table)
  4. If Join is for an element Collection, the FK is collection table.

1 To many (unidirectional) with Join Column ⟶ uses Join table (Course_id)

@JoinTable (maintain relationship b/w 2 tables)

* FK for each table!



John's Courses

1. Get John's id
2. Look John's Courses

Course.java

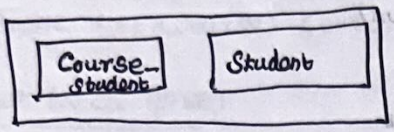
@ManyToMany
@JoinTable (name = 'Course_Student',
joinColumns = @JoinColumn (name = 'Course_Id'),
inverseJoinColumns = @JoinColumn (name = 'Student_Id'))

Refer to Course_Id column of Course_Student table

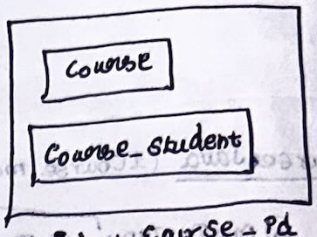
↳ Refer to Student_Id in that join table.

List<Student> students;

other side! (Info b/w Course & Students)



Join: Student_Id.



Join: Course_Id

Inverse: Other side of the relationship.

Student.java

@Entity
@Table (name = 'Student')
public class Student {

@ManyToMany
@JoinTable (name = 'Course_Student',
joinColumns = @JoinColumn (name = 'Student_Id'),
inverseJoinColumns = @JoinColumn (name = 'Course_Id'))
List<Course> courses;

1. Look at Student_Id (Course_Student)
2. Other side: Look at Course_Id in (Course_Student)
3. Find relationship b/w Student & Courses

---

with Lazy loading

@ManyToMany (fetch = FetchType.LAZY, cascade = {CascadeType.ALL})
* @JoinTable (name = 'Course_Student', joinColumns = @JoinColumn (name = 'Course_Id'),
inverseColumn = @JoinColumns = @JoinColumn (name = 'Student_Id'))

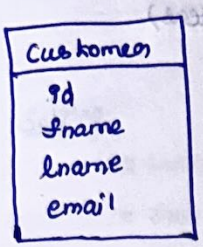
Private List<Student> students;

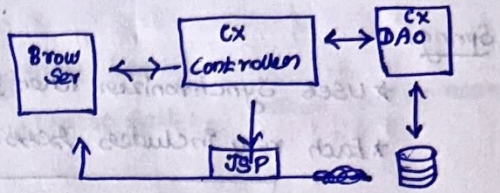
* Similarly for Courses!

* Customer: add, update, delete!

Tables:

* CREATE USER 'sampleUser'@'localhost' IDENTIFIED BY 'sampleUser';
* GRANT ALL PRIVILEGES ON *.* TO 'sampleUser'@'localhost';

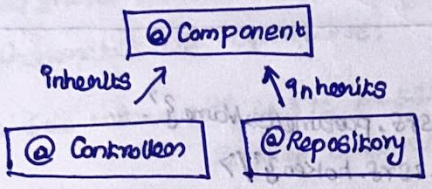




DAO:

* entityManager! : Autowire!
* getCustomers (@Transactional)

@Repository: For DAO implementation



"Auto component scanning", (Translates any JDBC exceptions)

Controller:

* get Customers : add as attribute to model!
* JSTL tags: JSP

Service layer:

↳ Put @Transactional in Service layer!

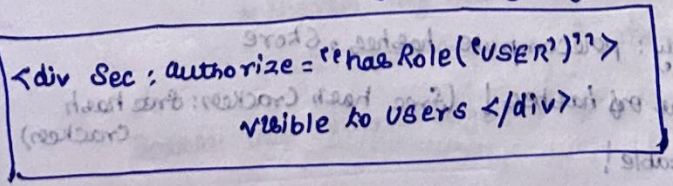
Service layer design pattern:

* multiple DAOs will be integrated using Service layer
* perform transaction management!

deposit(..)
withdraw(..)

Say both: single transaction
(Applied in DAO: makes it not pass!
Service layer: ! possible)

Spring Security tags



Property

connection.pool.initialPoolSize = 5
connection.pool.minPoolSize = 5
maxPoolSize = 20.
maxIdleTime = 3000.

Old (Refer doc for new versions)

CSRF

* <form: form> ⟶ Tag provides security against CSRF [no need now latest]
* CSRF: Cross Site Req Forgery ⟶ evil website, tricks execute unwanted to a website currently logged in!
* protect: Embed data/token in form: Then verify req in backend! (Spring Security Filters)

* CSRF protection by default, enabled.

Spring:

* Uses Synchronizer Token pattern
* Each req includes Session cookie + random token!

Non browser client: may disable CSRF protection (after careful review)

* Use POST: In form: Include CSRF token.
* <form: form> automagically adds CSRF tokens.

---

manually add:

<input type='hidden' name='${_csrf.parameterName}'.
value='${_csrf.token}'/>

↳ Else error as by default CSRF protection enabled.

Salted password hashing:

* Use password hashing frameworks! (libsodium, Password hashing)
* 1 way algorithm: match hash (not decrypted version)
* Never tell what's wrong (username/pwd)

Datastructure: Hash ⟶ Fast, Insecure.

Secure: SHA 256, 512, RipeMD, Whirlpool

How cracked:

* Dictionary, Brute force attack
  ↳ likely word, phrase, common pwd
* Brute: all combinations

Lookup table:

* Crack hashes effectively: precompute hashes, store
* Easy to lookup billions of hashes! (free hash Crackers: free hash cracker)

Compromised DB: Use lookup table!

Reverse lookup table:

* Apply dict/brute force to many hashes (not single): No need for lookup table
* Create Lookup table mapping each pwd hash from Compromised DB.

DB.	Lookup	Reverse
	Found	Found [user1, user5...]
	Not Found	Not Found

* Common: Same password

Rainbow table:

* Time-memory tradeoff technique (like lookup)
* Sacrifice hash cracking speed to make lookup table smaller.

md5: upto length 8: easy to crack!

Salting:

* Lookup/Rainbow: cracks as hash done the same way.
* Salt: hash twice: o/p different.
* Salt: may not be secret. (Stored along with hash)

Can't be precomputed: lookup (diff salt: reverse lookup now)

Don't:

* Short salt / common salt (hard code / generate randomly once)
* 2/more common password: Reverse lookup can be used.
* Generate salt each time.

3 digit salt means: 857,375 possibilities.

Easy to construct lookup tables! (Harddrive cheap!)

Don't use username as salt! (predictable)

256 bits SHA: Salt atleast 32 random bytes

Wrong way:

* Combine hashing functions: With no benefit
* Creates interoperability problems: less secure
* Never invent hash crypto functions.

Better than combining hash functions! Better way (make cracking slower)

1. md5, md5 with salt, sha1, sha1 salt.

Combinations like md5(sha1(pwd))

↓

Never!

Kerchkoff's principle:

* Hacker may have access: opensource / free
* With few samples: Reverse Engineering done.

Better: Use standard algo! proper salting. (HMAC)

Collision:

* Designed to make difficult. (Finding collision: hard even in weak md5)

Better use SHA256, SHA512, RipeMD, Whirlpool.

Java.security.SecureRandom

Generate salt: using Pseudo random number generators.

Web app: Always hash at server

Aop: Aspect oriented programming

Controller ↔ Service ↔ DAO

Logging:

* Security, DAO, controller: All layers → 100+ classes
* Code tangling: Logging, Security code inside all methods
* Code Scattering: Any change, change all classes.
* Solutions: Inheritance (no multiple inheritance, all classes need to extend)
  ↓
  Problem! (can't use inheritance!)

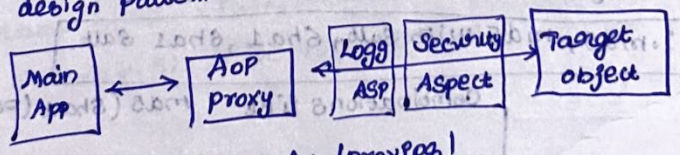
* Delegate:
  * classes delegate logging, security
  * Still need to update classes if → add/remove code
    → new feature auditing, API management, Instrumentation

Solution: Aspect oriented programming:

* Aspect encapsulates Cross cutting logic/concern.
* Basic info a all app will need!
* Take code → capsulate in class/module!: call accordingly!
* Aspect: class applied to diff parts of project.

Usecases:

* proxy design pattern



* main app: no idea about Aop/proxies!

Benefit:

* code reuse, change easily (not scattered)
* clean business code, less complex
* Configure wherever needed!
  * Logging, Security, transaction
  * Audit log: who, when, what, where
  * Exception handling: notify devops team
  * API management: analytics, load, performance.

Adv

* Reuse, no code tangling
* no code scattering
* Apply selectively based on config

Disadv

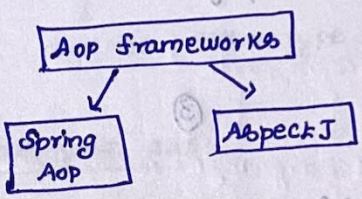
* Too many aspects: hard to follow
* minor performance cost (runtime weaving)

Spring AOP Support

* Aspect: module of code for cross-cutting concern (log, Security)
* Advice: what action is taken, when it should be applied.
* Join Point: when to apply code during program execution.
* pointcut: where advice should be applied.

Before advice: Before method!
After Advice: (Finally) after method.
After returning advice: Run after method (Success)
After throwing advice: Run after method (If exception thrown)
Around advice: run before & after method.

Weaving:
* Connect Aspect object to target object to create an advised object
* Types: Compile time, Run time, Load-time
* Runtime weaving: Slowest!



Spring Aop:
* Spring provides Aop support.
* In background uses Aop for Security, Caching, transactions etc..
* Uses run-time weaving of aspects.

AspectAOP: * original, Complete, Rich support for
Join points: method, Const, field level.
code weaving: compile, run, load time

Spring Aop adv:
* Easy to use, easy to migrate to AspectJ (@ASpect annotation)
* Only supports method level Join points!
only apply aspect to beans created by spring app context
Run time weaving: minor performance cost for aspect execution.

AspectJ:
* Supports all Joint points
* works with any PoJo not just beans from app context
* Complete Aop support
* Faster performance than Spring Aop

Disadv: Compile time weaving takes extra time
pointcut syntax can become complex.

* Spring Aop: light implementation: Common problem in Enterprise...

* Skoonk with Spring Aop then move to AspectJ.

Book: AspectJ in Action

Aspect oriented dev with use cases

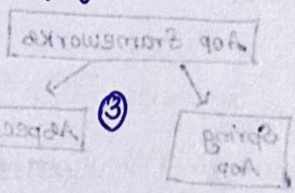
---

1. Target object: AccountDAO
2. Create Spring Java Config class
3. main app
4. Create Aspect with @Before advice.

① ②

```
@Component
Public Class AccountDAO {

    Public void addAccount() {

    }
}
```

```
@Configuration
@EnableAspectAutoProxy
@ComponentScan ("com.luv2code.aopdemo")
public class DemoConfig {
}
```



③

```
public class MainApp {

    main (String[] args) {

        AnnotationConfigApplicationContext con = new _____ (DemoConfig.class);

        AccountDAO ADAO = context.getBean ("accountDAO", AccountDAO.class);

        theAccountDAO.addAccount();

        Context close();
    }
```

④

```
@Aspect
@Component
public class MyAspect {

    @Before ("execution(public void addAccount())")
    public void beforeAccountAdvice() {

    }
}
```

Aop pointcut expression

* where advice should be applied
* Spring Aop uses AspectJ's pointcut expression language.
* many types : before, after (return, exception), after finally, around
* Execution pointcut : Applies to execution of method.

pointcut expression language :

execution (modifiers-pattern? return-type-pattern
declaring-type-pattern?
method-name-pattern(param-pattern) throws-pattern?)

? → optional (modifiers, declaring, throws)

@Before ("execution (public void addAccount())")

public → modifiers
void → return type
addAccount() → method

wild card : any method starting with add :

..., (public void add*())")

public * add())")

↳ any return type

Say:

AccountDAO.addAccount()
MembershipDAO.addAccount()

→ public * addAccount()
matches both!

match only AccountDAO class

* use Full Qualified name instead of method name

com.company.aopdemo.dao.AccountDAO.addAccount()

match methods with return type

public void add*() → any method with void, starts with add
void add*() → Any modifiers "
* add*() → Any return type "

Parameter pattern wildcards

() → No arg

(*) → one argument of any type

(..) → 0 or more argument of any type

@Before ("execution (* addAccount (com.appDemo.Account))")
↳ Argument Type

Match on package

*dao package

Com.luv2code.appdemo.dao.*.*(..))"
↳ Any class, any method
0/more arguments

(* add* (_____.Account, ..))")
↓ ↓
1st arg any no. of arg therefore!

method: any param

add(..)

Reuse pointcut expression

@Before ("execution (* com._____.*.*(..))")
public void addAccount() {
}

@Before ("addAccount()") {
...
}
↳ no need to @Before ("ex...(..))") Just mention reference!

@Before Advice

* Execute before target method! (Injection, Logging, Security)
* Audit log: when, where, what, who.
* API analytics.

Advices & Aspect: (Best practice)

* make Code Small, fast - No expensive / slow operations
* Get in & out quickly!

Problems

1. multiple pointcut expressions
2. Execute an advice only if certain condt met.
   (except getter & setter methods : package)
   ↳ Logic operators!

```
@Before ("exp1()" || "exp2")

@PointCut ("execution(* com.____.dao.*.get*(..))")
private void getter(){}

@Pointcut ("execution(* com.____.dao.*.set*(..))")
private void setter(){}

@Pointcut ("execution(* com.____.dao.*.*(..))") → every method in dao.
private void forDAOPackage(){}
```

All methods except getter & Setter:

```
@PointCut ("forDAOPackage() && !(getter() || setter())")
private void forDAOPackageExcludingGettersAndSetters(){}

@Before ("forDAOPackageExcludingGettersAndSetters()")
public void beforeAccountCreate(){
    ____
}
```

Control order of aspects being applied

* when matched on same condition
* when order not mentioned: Spring just picks one!

BeforeAddAdvice
PerformAdvice
LoggingAdvice

@Order

Lower no: precedence
1 > 2 > 3 . . .

```
@Before (____)
@Order(1)
public class LogAdvice{}{
    ____
}
```

* can (-)ve numbers : allowed!
* Range (Integer.MIN to MAX)
* No need to be Consecutive

Same order: Kind of uncertainity (Spring picks one!)

Read method arguments using JoinPoints

* Access & display method signature, arguments

```
@Before ("...")
public void Sample (JoinPoint jp)
                    └→ gives info about executing method
{
   Method signature ms = jp.getSignature();
   Object [] args = jp.getArgs();
   for (Object arg : args){
         Print (arg);
   }
}
```

@Before → public void sample () { } → @AfterReturning

@AfterReturning advice

Use: Log, Security, Transactions, Audit: who, when, what, where
Post process data (format): Be Careful!

Run after successful execution of a method:

```
* List <Account> findAccounts()
* @AfterReturning ("execution(* findAccounts(..))")
  public void meth() {
  }
```

Access Return value:

```
Pointcut =
@AfterReturning ("execution(* ____)", returning="result")
public void afterResult(JoinPoint jp, List<Account> result) {
   Account temp = result.get(0);
   temp.setName("Donald");
}
```

→ post process!

must be Same!

@AfterThrowing

* If exception thrown.
* Log exception, Notify devops; Encapsulate: reuse!

```
@AfterThrowing ("execution(* ____)")
____
{ ____
}
```

Access exception:

```
@AfterThrowing (PointCut = "_____", throwing = "ex")
public void test (JoinPoint jp, Throwable ex) {
        print (ex);
}
```

@After:

* After the method is completed finally!
* Regardless of success/failure!
* Log exception/audit, run regardless of outcome!, Cause.

@After: No access to exception. To Do So: Use @AfterThrowing!

* Shouldn't depend on outcome! (Log/Audit)

Recent Changes:

@Around > @Before > @After > @AfterReturning > @AfterThrowing.

But due to implementation in AspectJ AfterAdvice:

@After: only invoked after @AfterReturning/@AfterThrowing in the same aspect.

@Around:

* Log, audit, security *pre/post process
* Instrumentation, profiling, manage exceptions: Swallow, handle, stop

Proceeding JoinPoint:

* Handle to target method: Can use to execute target method!
  eg: Log timing.

```
@Around ("execution _____")
public void noteTime (ProceedingJoinPoint pjp) {
        long begin = System.currentTimeMillis();
        Object result = pjp.proceed();
        print (now - begin);
```

Resolve print order issue

* Data is printed using 2 o/p stream → Logger o/p stream (Spring)
  → println: std out o/p stream

* Solution: * Send to some o/p stream!
  (use o/p stream logger of spring) (Same as) : To synchro..

```
private Logger logger = Loggers.getLogger(getClass().getName());

@Around(_____)
public void test(ProceedingJP jp) {
    Loggers.info(_____);
}
```

Handling exception

* Rethrow/handle exception

```
@Around("execution(_____)")
public Object test throws Throwable {
    _____
    try {
        result = JP.proceed();
    } catch ____
        ____
        ____
}
```