Spring framework

Requirements: Java 17, Spring Framework 6/Spring boot 3 requires Java 17+, JDK

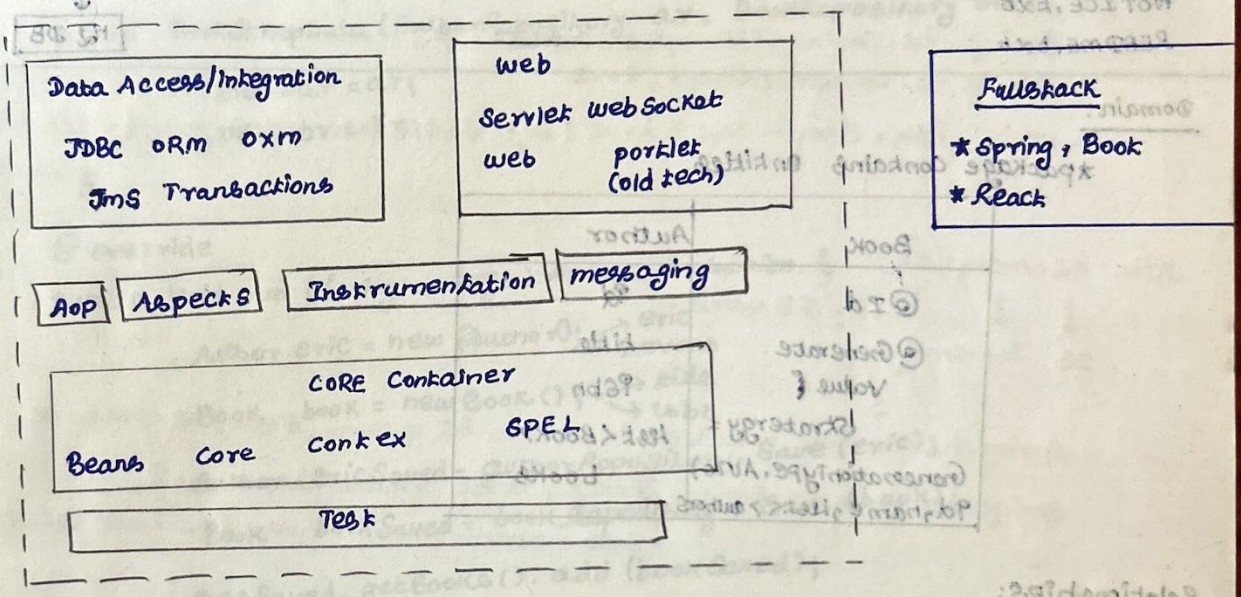
Trouble shoot problem:

* gist (Stacktrace), Source code for help (To the point: not vague)
* Change 1 time @ a time! (Thing): Retest!

Spring:

* Popular (Banking, Finance, Retail): monolithic, Backend, microservice [70%] Java using comp. (uses Spring)
* Simple alternative to J2EE (without Enterprise Java Bean): XML hell
* Focus on POJO, annotation based

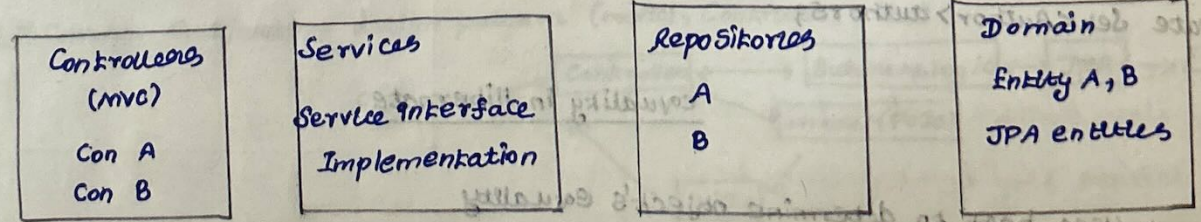
Spring Framework: Dependency Injection, Transaction mana, web
Spring Boot: Wrapper (build faster)



Features: H2 DB, Log auto config, Config using files, performance metrics, Health endpoints, enhanced failure info

Projects: Data (SQL, NoSQL persist), Cloud, disk, Eys, Security, Authori, Authent, Session, Distri. web app sessions, Integration, Batch, State machine, Integ patterns

* Browser (uses) HTTP: Spring (8080 port): Tomcat listens to this port
* Routes to Spring Boot app (can respond with .html, JSON)...



Directory structure (maven)

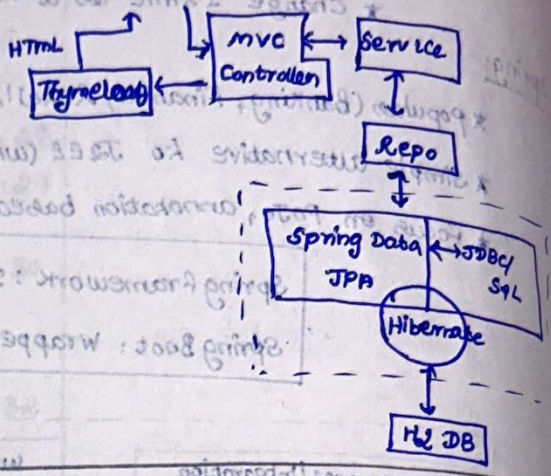
```
Src/main/java
Src/main/resources
    /filters
    /webapp
Src/test/java
    /resources
    /filters
Src/it
Src/assembly
Src/site
LICENSE.txt
NOTICE.txt
README.txt
```

Hostos

Jvm

TomCat

Springboot



Domain:

*package contains entities

Book	Author
@Id @GeneratedValue(strategy = GenerationType.AUTO) id, name, List<> authors	id title isbn List<Book> books

Relationships:

* 1 author many books, 1 book many authors

Author.java

```
@ManyToMany(mappedBy = "authors")
private Set<Book> books;
```

Book.java

```
@ManyToMany
@JoinTable(name = "author_book", joinColumns = @JoinColumn(name = "book_id"),
        inverseJoinColumn(name = "author_id"))
private Set<Author> authors;
```

equality in Hibernate

* uses hash to determine object's equality
* Some use id alone, Some uses all property (depends on case)

Data Repositories:

```
Public Interface AuthorRepository extends CrudRepository<Author, Long>   (Long → PK)
{
  // Already Implemented: (defined): Save, findAll, findById, Count, deleteById...
}

// Same for BookRepository <Book, Long>
```

H2 database (init data):

```
@Component
public class BootStrapData implements CommandLineRunner {

    private final AuthorRepository ar;
    private final BookRepository br;          → Autowires!

    public BootStrapData(AuthorRepository ar, BookRepository br) {
        this.ar = ar;
        this.br = br;
    }

    @Override
    public void run(String... args) throws Exception {
        Author eric = new Author();        → eric, evans
        Book book = new Book();            → title, isbn
        Author ericSaved = authorRepository.save(eric);
        Book bookSaved = bookRepository.save(book);
        ericSaved.getBooks().add(bookSaved);
        bookSaved.getAuthors().add(ericSaved);
    }
```

Good practice: Save the new values back!

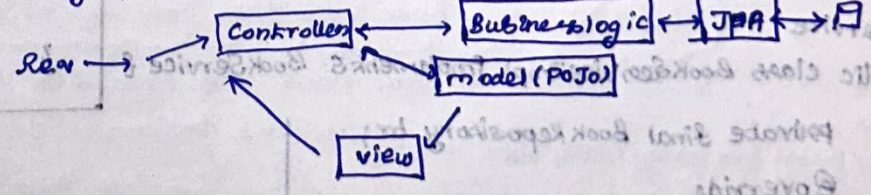
H2 Console: (Properties)

spring.h2.console.enabled = true (via browser).

localhost:8080/h2-console/login.do?

Spring MVC

* Common GUI/webapp design pattern (model, Controller, View)



* populate model to render in view

model: POJO (properties): may/maynot be used by view

view: Thymeleaf / JSP / Jackson (html, xml, JSON, text)

Controller: Request mapping (Cop): with business logic/service to give model & view!

H2-database:

*In Pom: make Scope (Remove: Runtime): make always available.

H2-db properties:

Spring.datasource.url = jdbc:h2:mem:db
- driverclassname = org.h2.Driver
- username
- password

Spring.jpa.database-platform = org.hibernate.dialect.H2Dialect
- hibernate.ddl-auto = none/

Spring.jpa.open-in-view = false (default: true)

Spring.h2.console.path = /h2 (default: /h2-console)
Spring.h2.console.enabled = true
Spring.jpa.show-sql = true (log sql statements gen. by Hibernate to Console)

jdbc:h2:mem:db
- jdbc → DB
- h2 → Inmemory
- db → my DB name

Spring.h2.console.enabled: to access h2 from localhost:8080/h2-console
Spring.h2.console.settings.trace = false (Allow trace o/p)
- web-allow-others = false (remote access)

* doesn't h2 DB persist by default (not retained data)

/resources
- schema.sql (Table Schema) → Not inside folders (Just /resources)
- data.sql (Data sql)

persist (use file):

Spring.datasource.url = jdbc:h2:file:c:/h2db/sample.

Service layer

```
@Service
public class BookServiceImpl implements BookService {
    private final BookRepository br;
    @Override
    public Iterable<Book> findAll() {
        return br.findAll();
    }
}
```

```
public BookServiceImpl(BookRepo br){
    this.br = br;
}
```

```
public interface BookService {
    public Iterable<Book> findAll();
}
```

Author.java
Book.java
] Entity

BookController.java
AuthorController.java

AuthorRepository.java
BookRepository.java

BookService.java
BookServiceImpl.java
AuthorService.java
AuthorServiceImpl.java
] Service

Spring Controllers

```
@Controller
public class BookController {

    private final BookService br;
    public BookController(BookService br) {
        this.br = br;
    }
    @RequestMapping("/books")
    public String getBooks(Model model){
        model.addAttribute("books", bookService.findAll());
        return "books";
    }
}
```

Can't return objects (For HTML response)

Thymeleaf: (Pom)

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>Spring Demo</title>
</head>
<body>
    <h1>Book list</h1>
    <table>
        <th>Id
        Title
        Publisher
        <br th:each="book:${books}">
    =
</html>
```

Spring: web, Restapi, microservices, Fullstack!

Loosely coupled: can be (easy to add/edit/remove)!

eg: Interface: Mario, Super Contra, Pacman (GamingConsole)

Tightly coupling:

* Coupling: how much work is involved in something?

eg: Flat tire: change (Loosely coupled)

1. Business new changes
2. Framework/Code changes

* make functional changes with as less code change as possible!

Loose coupling: Interface:

GamingRunner → GamingConsole (Interface) → Game 1 ... n

"No matter what game it is: no need to change GameRunner"

Wiring dependency

* GamingConsole is a dependency to GameRunner (wire to it)
* Enterprise: Can't create & manage/wire: Thousands of objects.

To make Spring manage a String:

```
main (String[] args) {
    //Launch Spring Context, Configure Spring (what it need to manage)
    AnnotationConfigApplicationContext context = new ——— (HelloWorld(on class))
    context.getBean("name");

@Configuration [1/more bean method implemented]
public class HelloWorldConfig {
    @Bean ——————→ (method produces a bean to be managed by Spring Container)
    public String name() {
        ——
    }
    @Bean
    public int age() {  ~~~~~→ access: Context.getBean("age");
    }
}
```

"Refer: Configure bean with Java Code: pg: no: 19" : Refer UseCase

↳ bean name different than actual

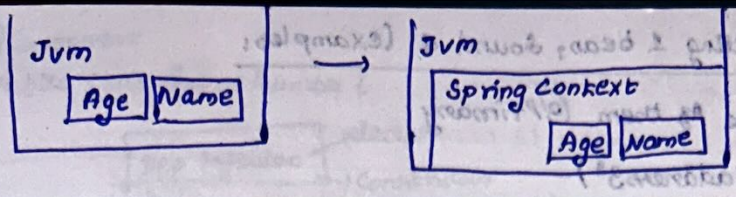
Use:

1) existing 3rd party class avail to Spring (multiple implementation: need to define specific)
2) No access to JAR

"Same as bean method name"

↓

Helpful in libraries (not editable)



JDK 16

record Person (String name, int age) { }; ⟶ Automatically creates getter, setter, Constructors.

own name to bean instead of method name:

@Bean (name = "addressNew")
public Address address(){ ⟶ Context.getBean ("addressNew");
}

1) Spring Container, Spring Context, Ioc Container, ApplicationContext:

2) Java bean (vs) Spring Bean

3) List all beans managed by S.F.

4) multiple matching beans?

5) Spring managing & performing autowiring ⟶ why we are not creating objects
↳ How Spring does?

1) Spring Container (Ioc Controller / Spring Context / Spring Ioc Container / AppContext)

* manages Spring beans, lifecycle.
* Ioc: Inversion of Controller

PoJo + Config ⟶ Ioc Container ⟶ Ready System (Objects)

2 popular types: BeanFactory (Basic)
ApplicationContext (Advanced JEE) → IOC
→ AoP support
→ Easy to use webapp

IoT (Severe memory Constraint) → Spring!

2) Java Bean (vs) Spring bean / PoJo:

* PoJo: plain old Java object (members, methods): NoConstraint (any class)
* Java bean: EJB (Enterprise Java Bean) 3 rules → 1) public no-arg Construct
→ 2) getter, setter
→ 3) Serializable

* (Not important (Java Bean) as EJB anymore
* Just PoJo (any class), Spring Bean!

Spring Bean: Anything managed by Spring Context is a spring bean!
name, age, address

3) List all beans managed by S.F:

Context.getBeanDefinitionNames() : String[]
.getBeanDefinitionCount() : int.

1) No unique bean exception : expecting 1 bean, found 2 (examples:

1) Fix : make one of them @Primary

2) @Qualifier ("address3")

* Autowiring/fetch object: multiple candidates (error)!

```
@Bean
@Qualifier("address3Qualifier")
public Address address3(){
}
```

```
@Bean
@Primary
public Address address(){
}
```

Apply it to Gaming Console:

```
@Configuration
public class GamingConfig {

    @Bean
    public GamingConsole game(){
        var game = new PacManGame();
        return game;
    }

    @Bean
    public GameRunner gameRunner(GamingConsole game){
        var gameRunner = new GameRunner(game);
        return gameRunner;
    }
}
```

make Spring create & manage objects: (without @Bean : manually)!

```
@Component
public class PacManGame implements GamingConsole{
    up()
    down()
    left()
    right()
}
```

@ComponentScan ("path"): where to look for dependencies

```
@Bean
public GameRunner gameRunner(GamingConsole game){
}
```

```
private GamingConsole game;
public GameRunner(GamingConsole game){
    this.game = game;
}
```

→ only 1 implementation (no worries)!

→ dependency injection!

→ Dependency injection!

```
main(—) {
    context.getBean(GamingConsole.class).up();
    context.getBean(GameRunner.class).run();
}
```

→ without o/p : it can automatically injects!

@Component
public class GameRunner {

Dep injection → declare
→ Constructor

run() {
game.up();
game.down();
}
}

@Primary
→ To class
→ To bean method

Constructor (@Qualifier("className/qualifiername") Gas gas) {
}

@Primary / @Qualifier: If more than 1 implementations

* Choose Specific: @Qualifier (preferred!)

@Component
@Qualifier("name")
public class —— {
}

call → public GameRunner (@Qualifier("name") GameRunner game) {
this.game = game;
}

"Don't use both" → "Think from user class's perspective"

* @Qualifier: higher priority

Types of dependency injection: Constructor, Setter, field (using reflections)

"Continued in upcoming pages"

Dependency Injection:

* OOP's SOLID (Robert Martin): principles [Agile Software development]
* Michael ← SOLID acronym was created.

S: Single Responsibility
O: open closed
L: Liskov Substitution principle
I: Interface segregation
D: Dependency Injection principle

→ Just because you can doesn't mean you should.

Single Responsibility:

* Single responsibility: Never more than 1 reason to change
* Small (screen full of code), No god classes (everything)
* Big → Small classes
* 2000 line method tested "fine".

open closed:

* open for extension, closed for modifying. (modify: should extend)
* private variables + getter, setters (only when needed)
* Abstract base class. [not initiated: need to subclass to use its property]

Liskov Substitution:

* Liskov (Barbara): 1998
* Objects: Replaceable with instances of their subtypes without altering.

the correctness of the program.

* Violation: often fall in the 'is a' test
* Square is a Rectangle, Rectangle is not a Square.

> Usable: Square, anywhere you use a rectangle! Strange!
> Square fails Liskov Substitution principle! Bad thing!

About interfaces & how to decide when to extend!

Interface Segregation principle:

* Fine grained (single purpose) interface; client specific.
* Focused, minimize dependencies b/w them.
* Relation with Single Responsibility principle: 'God' Classes/interfaces.

Dependency Inversion principle:

* Abstraction should not depend upon details
* Details should depend on abstraction
* Important that: higher level & lower level objects dep on same abstraction/interface.
* Different from Dependency Injection (obtaining dependent objects)

* higher class not dependent upon lower classes instead depend upon abstraction of lower classes (rather than detail: lower classes)

with DI

```
class Employee(object):
    work
class Manager():
    addEmployee();
class Developer():
    work();
class Designer():
    work():
class Tester();
    work()
```

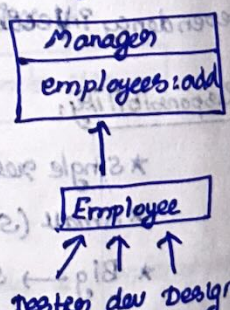
* Higher classes need not to know about lower classes!
* Loosely Coupled!

> Socket: Anything! Plug in!

Without DI

Manager → dev, designer, tester → letters (add)

* 'exposed everything about lower layer to the upper layer'
* manager must know about all types of workers.
* new role: Change manager class!



Summary:

* Testable, maintainable (not tightly coupled): Every request: own Controller. No need → "No need" class.

Spring Context

* Even no web dependency/tomcat: @Controller still creates the class as Spring bean
* SpringApplication.run(SpringDemoApp.class, args) → still returns ConfigurableApplicationContext

ConfigurableApplicationContext:

```
* ApplicationContext cxt = SpringApplication.run(_____, args);
* MyController controller = cxt.getBean(MyController.class);
* controller.SayHello();
```

"Spring create the bean & put it in the context" → "we can use for dependency injection"

Test Context:

>test>Java>....> SpringDemoAppTest

```
@SpringBootTest
class SpringDemoAppTest {
    @Autowired
    ApplicationContext appContext;

    @Test
    void contextLoads() {
    }

    @Test void testContext() {
        MyController myCon = appContext.getBean(MyController.class);
        System.out.println(myCon.sayHello());
    }

    @Autowired
    MyController cont;

    @Test
    void testCon() {
        con.sayHello();
    }
}
```

→ Error! why? we need webContext which is not in testContext (unusual! we rarely need this scenario to directly use)!

"But work"

Basics of dependency Injection

* 5 yrs old: open refrigerator: doors open, have not permitted / expired / not have.
  State need: we will make sure you will have something to eat.

* where a needed dependency is injected by other object (using setter / constructor)
* Class being injected: has no responsibility in injecting / instantiating the object.
* Doesn't mean
  1) You avoid using 100% new
  2) Be pragmatic in what is & not being managed in Spring Context.

Types:

* By class properties: Not / least preferred (public / private)
* using private property: evil (using reflection: slow, testing difficult)
* Setters: Area of much debate
* Constructor: Most preferred. [Instantiate: create object : pass dependency
  object can't exist without dependency)

Concrete classes (vs) Interfaces:

* DI can be done with concrete classes [has implementation for all of its meth
* Can / ~~easy~~ not: extend abstract class [Complete class]
  1. Implementation for all methods
* But generally avoid using DI with Concrete classes

DI via Interfaces highly preferred:

* Allow runtime to decide implementation to inject
* Follow Interface Segregation principle of SOLID.
* makes code more testable - mocking becomes trivial! (easy)

Inversion of control (IoC):

* Technique to allow dependencies to be injected at runtime.
* Dependencies are not predetermined.
* Allows the framework to compose the app by controlling which implementation is injected (eg: H2 / mySQL dB).

Important: chain: methods defined by the user to tailor the framework will often be called within the framework rather than from user app code.
framework: plays the role of the main prog in coordinating & sequencing app activity. IoC gives the power to framework to serve as 'extensible skeletons'

* user tailor the generic algo in framework!
* But framework is the one that calls it

* work with framework (not very directly): Just tailor!

Ioc (vs) dependency Injecting:

* DI: Compose code with dependencies in mind / write code to inject one
* Ioc: Runtime environment for your code.
  - ↳ Control DI: Inverted to framework
  - ↳ Spring controls the injection of dependencies.

Good practices:

1. Constructor Injection
2. Declare final private & initialize in the constructor (dependencies)
3. whenever practical: use interfaces (Code to an interface)

without Injection (dependencies)

* Not preferred: Internal dependency (create & use)

```
private final MyService myService;
public MyController() {
    this.myService = new MyServiceImpl();
}
```

→ not preferred!

@Component: Instance of class managed by Spring framework
@ComponentScan("path"): Scan the path for dependencies
Spring Bean: An objects managed by Spring framework
Ioc container: manage lifecycle & dependencies
Autowiring: wires dependencies for a Spring bean.

@Component	@Bean
used: Any Java Class	@Configuration classes
ease: very easy	we need to write code
Autowiring: Field, Setter, Constructor	method call / method param
who creates bean: Spring	Manual
Recommen: Instant bean for own app	1. Custom business logic 2. 3rd party bean initialization

* Each layer dependant to the next layer (As talking to it): eg: web → Business ↳ Data
* we focus on business logic not on create instances!

without dependency Injection

Property Injection:

```
class PropertyInjectedControllerTest {
    PropertyInjectedController propC;
    @BeforeEach
    void setUp() {
```

```
PropC = new PropertyInjectedController();
PropC.greetingService = new GreetingServiceImpl(); → without this : null pointer
                                                    (As we are calling empty constructor)
}
@Test
void SayHello() {
    propc.SayHello();
}
}
```

'least preferred'

```
public class PropInjCon {
    GreetingService gs;
    public String SayHello() {
    }
}
```

Setter injection (2nd least preferred):

```
public Class SetterInjectionController {
    private GreetingService gs;
    SayHello() { ——— }
    public void SetGreetingService (GreetingService gs) {
        this.gs = gs;
    }
}
```

```
Class ——— {
@BeforeEach
void setUp() {
    SetterInj = new SetterInjCon();
    SI.setGreetSer(new GS());
}
@Test
    :
}
```

Constructor Injection:

```
public class Cons ——— {
    private GS gs;
    public ConsInjSer(GS gs) {
        this.gs = gs;
    }
    SayHello() { }
}
```

Note

GS: GreetingService

↑

GreetingServiceImpl

```
@BeforeEach
setup() {
    new Cons ——— (new GS());
    :
}
:
```

Using Spring framework

```
@Controller
public class PropInjController {
    @Autowired private GS gs;
```

↓

Still works on private optional (Reflection):

```
@Controller
public class ConsInjCon {
@AW private final GS gs;
public ConsInjCon(GS gs) {
}}    this.gs = gs;
```

```
@Controller
@Autowire → property
            Setter!
```

Proper way: Constructor

```
@ActiveProfiles("EN")
@SpringBootTest
class myI18NControllerTestES {
    @Autowired
    MyI18NController myI18NController;

    Constructor
    SayHello() { myI18NController.SayHello(); }
}
```

Spring profiles

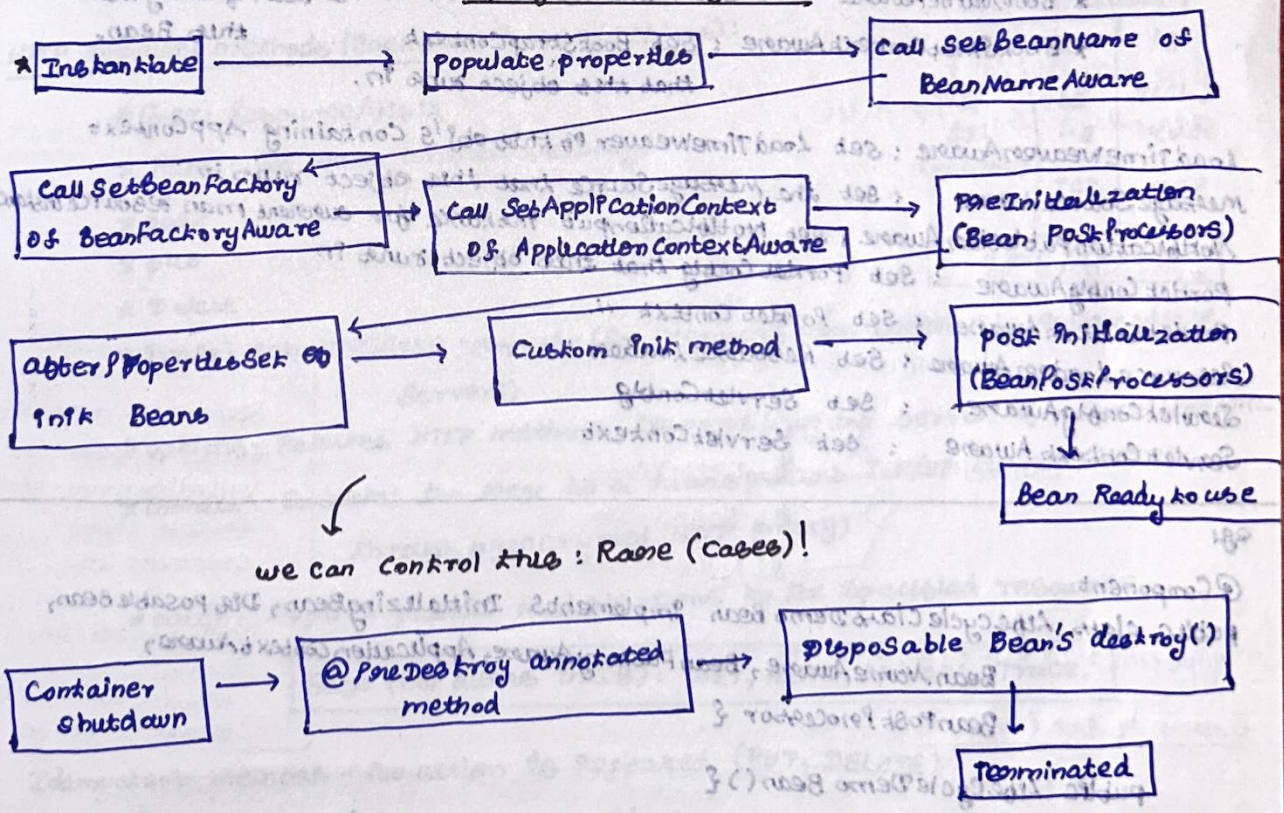
```
GreetingService                    [@Profile("EN")]
EnglishGreetingService             [@Service("I18NService")
SpanishGreetingService.            @Profile("ES")
                                   @Service("I18NService")

myI18NController {
    GS gs (private final)
    private MyI18NController(@Qualifier("English
```

@ActiveProfiles("EN") ⟶ Testing annotation!

Default profile:

```
@Profile("EN", "default")
@Service("I18NService")
public class EnglishGreetService implements GS {
    :
}
```

Spring Bean lifecycle



Call back Interfaces:

* InitializingBean.afterPropertiesSet() : called after properties set
* DisposableBean.destroy() ⟶ during bean destruction in shutdown

Lifecycle annotations:

* @PostConstruct : after bean construction (before returned to req obj)
* @PreDestroy : Before bean destruction by the Container.

BeanPostProcessors: (Interact with bean as they are processed)

* Implement BeanPostProcessor interface

PostProcessBeforeInitialization: Before bean init

PostProcessAfterInitialization: called after init

Globally

* called for all beans in context!

Aware Interfaces (over 14 aware interfaces):

* Access Spring framework infrastructure
* Largely used within the framework
* Rare: Spring dev.
* Review extensions of the aware interface for Current interfaces.

CurrentAware Interfaces:

* ApplicationContextAware: Implemented by any object that wishes to be notified of the ApplicationContext that it runs in
* ApplicationEventPublisherAware: Set App. EventPub that this object runs in.
* BeanClassLoaderAware: Callback that supplies the bean loader to a bean instance.
* BeanFactoryAware: Callback that supplies the owning factory to a bean instance
* BootNameAware: Set name of the bean in the bean factory that created this Bean.
* BootStrapContextAware: Set BootStrapContext that this object runs in.

LoadTimeWeaverAware: Set LoadTimeWeaver of this obj's Containing AppContext

MessageSourceAware: Set the MessageSource that this object runs in

NotificationPublisherAware: Set NotificationPub instance for current man resource

PortletConfigAware: Set PortletConfig that this object runs in

PortletContextAware: Set PortletContext "

ResourceLoaderAware: Set ResourceLoader "

ServletConfigAware: Set ServletConfig

ServletContextAware: Set ServletContext

---

eg:

```
@Component
public class LifecycleClassDemoBean implements InitializingBean, DisposableBean,
        BeanNameAware, BeanFactoryAware, ApplicationContextAware,
        BeanPostProcessor {

    public LifeCycleDemoBean(){
        Println("I'm in Lifecycle Bean Constructor");
    }
```

"Refer Repo for Code"

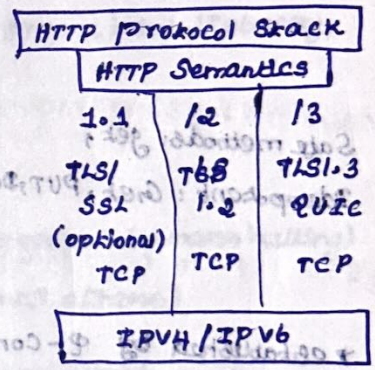
Restful web services

* way of providing web services (REST: protocol): No standard for this (opiniated)
* HTTP Protocol: HyperText Transfer protocol → Tim Berners Lee (1989): CERN
  HTTP/0.9: original version
  Telnet friendly protocol.

> telnet google.com 80
↳ "Connect"; Issue Commands
GET /about/

HTTP history:

* 1.0: 1991-95 (New Software 'web browser' emerged)
* HTTP Standards were developed by (IETF, W3C)
* 1.1: Solved ambiguities (support alive, encoding chunks, byte range transfer)
* Updated 1999, 2014 (still widely used)
* HTTP/2: Standardized (2015): 2021 (10 million websites 47% supported)
  ↳ Performance, low latency, Higher throughput!
* HTTP/3: 2018 (73% browsers, 25% of top 10 million websites)
* Uses QUIC network protocol not TCP.
* Adoption in Java Community!



HTTP request methods (Backbone)/verbs (actions):

* GET: Resource/visit
* HEAD: get meta without the body.
* post
* put
* Delete
* Trace: echo received request (See request was altered by intermediate servers)
* options: Returns HTTP methods supported by the server for specified URL
* Connect: Converts the req to a transparent TCP/IP tunnel (HTTPS through unencrypted HTTP proxy)
* patch: Applies partial modifications to the specified resource.

Safe (not alters in DB): GET, HEAD, options, Trace.

Idempotent methods: no action to repeated (PUT, DELETE)

Safe method: GET, HEAD, Trace, options

Being idempotent: not enforced by the protocol!

GET, patch, head, post: Cacheable.

100: Informational in nature

200: successful request (200 okay, 201: Created, 204: Accepted

300: Redirections (301: moved permanently)

400: Client error (400: Bad request, 401: Not authorized, 404 not found)

500: Server side errors (500: Internal Server error, 503: service unavailable)

Restful web services:

* Standard (Representational State transfer) JSON/XML via HTTP
* Roy Fielding (2000) established.
* Verb create, manage, edit: Server side resources → GET, DELETE
* Status Code: success, failures, errors
* Not formal standard (unlike SOAP): But agreed uses!

verbs: HTTP methods

messages: JSON/XML

URL: Uniform Resource Locator (network info)

* Stateless (no client side state info)
* HATEOS (Hypermedia As The Engine of Application State)

↳ Rest client should use server provided links dynamically to discover all available actions & resources it needs (Server responds with hyperlinks to other actions)

Safe methods: get,

Idempotent: Get, PUT, Delete.

Richardson maturity model (Rmm)

* established by P-Con presentation: Leonard Richardson (2008)
* Describe the maturity of Restful services.
* Unlike SOAP: no formal specification, Rmm's: describe quality of Restful service.

Rmm levels:
0) The Swamp of POX
1) HTTP verbs
2) Resources
3) Hypermedia Controls

Hypermedia, HTTP, URI: Core technologies

Resource not found: 404
Server exception: 500
Validation error: 400
Success: 200
Created: 201
No Content: 204
Unauthorized: 401
Bad request: 400
Resource not found: 404
Server error: 500

Swamp of POX (Level 0):

* plain old xml: Implementing protocol as transport protocol.
* URI (only one), 1 method (RPC, SOAP, XML-RPC)

Level 1: Resources)

* multiple URIS

http://www.google.com/1234

* Still uses single method (GET)

Level 2: * verbs, common in practical use

Level 3: * may have useful URIs, devs to explore, no clear std (HATEOS)

Level 1: Usage URI → distinct
2: Verbs
3: Discoverability (Self documenting)

Spring framework with Restful Web Services.

* 3 web frameworks: Create Restful services
* 2 web clients
* Spring mvc → compatible with JavaEE/Jakarta mvc model
  - → Robust support for traditional web app
  - → Based on traditional Java Servlet API
  - → Nature: Blocking, non-reactive. Components
* Spring webflux: Project reactor to provide web services (reactive), No use of Java Servlet API closely follow mvc, easy transition.
* webflux.fn: functional prog (annotation based), Simply define microservice endpoints.

* web: RestTemplate (primary, mature, configurable: no new features, Recommends webclient for new development)
* Spring webclient: Reactive web client, non-blocking HTTP client library, uses Reactor Netty.

Marshalling / UnMarshalling:

* Pojo to xml / JSON: marshalling (vice versa: Unmarshalling)
* Spring boot uses Jackson (Also supports others)

Single page App (SPA) → often Restful web service APIs are used
client: Vue, ReactJS, AngularJS, EmberJS (no choice)
↳ why? → decoupled with HTTP/JSON/xml layer
Server: Boot, Ruby on Rails, .NET!

Project Lombok

>Intellij >process annotation (enable)!

* Java Cursed: Too much ceremonial code (getters, setters): Lombok generates!
* @Getter, @Setters: save time, clean code.
  - * Reinier Zwitserloot: @Surial on Twitter & Reel Spirkes (Before 2009)
  - * Lombok: Island in Indonesia (Second Island East of Java
  - * Tag: Spice (Indonesia)
* Hooks in via annotation processor api
* AST (raw Source code) passed to Lombok for code generation before Java Continues
* produces properly compiled code with Java Compiler.
* Generated & compiled (No runtime performance penalty)

your code is used
  ↳ Easy to override
  ↳ Custom Setters

* Compiled Code Change not Source Code: IDEs may get confused (Latest: no!)
* plugin: There!

Val: declare final local variable
Var: declares mutable local variable

@Getter: create getter for all properties
@Setter: create setter for all non-final properties.
@ToString: String of class name, each field seperated by commas
  optional param to include field names, include call to the super toString meth
@EqualsAndHashCode: Implementations of equals, hashcode
  Default: use all non-static, non-transient properties
  optionally exclude certain properties.
@NoArgsConstructor: If final fields: cause Compiler Error
  can optionally force, which will init fields with 0/false/null
  no args constructor
@RequiredArgsConstructor: Constructor for all fields (final / @NotNull)
  @NotNull fields are null → Throw NullPointerException

@Data: POJO boilerplate (getter, setter, ToString, EqualsAndHashCode,
  RequiredArgsConstructor)
  No Constructor is generated if Constructors: explicitly declared.
@Value: Immutable variant of data
  All fields: private
@NotNull: Set on param of method/constructor & a nullpointerexception:
  will be thrown if parameter is null.

@Builder: Builder pattern
  person.builder().name("Adam").city("San").job("Myth").
  build();

@SneakyThrows: throw Checked exceptions without declaring in calling
  method's throws class
@Synchronized: A safer implementation of Java's synchronized.
@Log: create Java util logger (awful)
@Slf4j: SLF4J logger (Recommed: generic logging facade)
  Springboot's default logger: logBack.

logging.level.guru.springframework=debug

> log.debug("_____");

>builder > lombok > Choose (AutoCompletes)!

Spring Bean Initialization

* Default: Eager [Initialize beans at Start]

```
@Component
@Lazy
public class ClassA {
    ;
}
```

→ 'only when required' → 'use @Component / @Bean'

Every class load lazy

Spring.main.lazy-initialization=true.

* Eager: Recommended (Any errors discovered at Startup)
* Lazy: not frequently used / recommended
* lazy: @Component/@Bean: Lazy-resolution proxy is injected instead of actual dependency.
  @Configuration
  ↳ All @Bean methods lazily loaded.

Lazy: Load first usage, not default, @Lazy/@Lazy(value=true)
Errors: Runtime, Raise & not recommended, memory (until used no), Rare!

Eager: Startup init, default, @Lazy(value=false)/absence, Error: Startup,
Recommended.

Spring Bean scopes (Normal, Prototype, Singleton)

* How long live, how many instances, how shared.
* default: Singleton (only 1 instance, Cached: all dependency injection refers that)

```
@Component
public class ClassA {
    ;
}
```

(or)

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_SINGLETON)
public class CricketClass {
    ;
}
```

Prototype:

* SCOPE_PROTOTYPE (New object every time)

REQUEST: 1 object per single HTTP request
Session: 1 object per user HTTP Session
App: 1 object for the entire app
WebSocket: 1 object per WebSocket instance.

Java Singleton (GoF) vs Spring Singleton:

- * Spring Singleton: one object instance per Spring Ioc container
- * Java Singleton: one object instance per JVM
- → many Spring Ioc (Rare)!

Prototype: many instances, new one each time, @Scope (value = ConfigurableBeanFactory.SCOPE_PROTOTYPE), Rarely used, Stateful beans (Recommended)

Singleton: only one instance, • SCOPE_SINGLETON, Default, Stateless!

---

* @PostConstruct: init dependancy/configuration
* @PreDestroy: close connections

Evolution of Jakarta EE (vs) J2EE (vs) Java EE:

* Initially, Enterprise capabilities built in to JDK.
* Later: J2EE (Java 2 Enterprise Edition)
* Rebranded to Java EE → Oracle → eclipse foundation
* over time (Jakarta EE): eclipse renamed!
  - → JSP (Jakarta Server pages)
  - → JSTL (Jakarta Std Tag Lib)
  - → EJB (Jakarta Enterprise Beans)
  - → Jax-Rs (Jakarta Restful web services)
  - → Jakarta Bean validation
  - → Jakarta Context & Dependency injection,
  - → JPA (Jakarta persistence)

Spring 6+, Spring Boot 3+ → Jakarta EE

Jakarta Context & Dependency Injection:

* Released 2009, Introduced to Java EE 6 in 2009
* CDI: Specification (Interface)
* Spring framework implements CDI
* → Inject (Autowired)
  - → Named (~Component)
  - → Qualifier
  - → Scope
  - → Singleton,

To use CDI: need Pom Config

---

* Spring Boot Xml Config: Rarely web!
* new ClassPathXmlApplicationContext ("____.xml");

Annotation: easy, short, precise, No, PoJo's polluted, easy to maintain,
Usage: Almost all recent, Hard to debug

Xml Config: cumbersome, Not short, No Change in PoJo, hard to maintain/debug.

@Component : Generic (any class) : Base for all Stereotype annotation

- @Service : business logic
- @Controller : web controller
- @Repository : Implementation of DAO

Tips: use specific annotation! So easy to use AOP! (@Repo: Translates mysql errors)

@Configuration : Declare one/more @Bean method : Processed by IoC

@ComponentScan : where to scan

@Bean : method Produces bean, managed by IoC

@Component : Any class : maintained by IoC

@Service : Business logic

@Controller : web Controller

@Repository : DAO implementation

@Primary : when multiple implementations of a bean : choose the one

@Qualifier : choose particular bean

@Lazy : lazy loading

@Scope(...) : Define Scope

@PostConstruct : execute after dependency injection (any init)

@PreDestroy : Call back : Removed from IoC!

@Named] Jakarta CDI Annotation similar to @Component, @Autowired

@Inject]

DI (getter, setter, constructor)

IoC / Spring Bean factory / App Context

Spring Bean : obj maintained by IoC

Autowire : Process/wire dependancies

Spring Core : IoC, Dependancy Injection, Autowiring (webapp, Restful API, Implement authen, authorization, Talk to DB / other system, Unit test)

Spring frameworks : Spring Core, Test, MVC, JDBC

- web : mvc
- Reactive : webflux
- Dataaccess : JDBC, JPA
- Integration : Jms
- Testing : mock objects, Spring mvc test

Popular : Loose Coupling, Boilerplate Code, Flexible architecture, Evolve!

Why Spring boot : (Import many + many), Lot of Config! [Log, monitoring, Error handling]

* Reduce Config Code, no need to create project
* Spring init, Starter project, Auto Config, Dev tools, Actuator!

Production ready

Bundle : Spring Initializer

Autoconfig for Spring boot: (Based on Jars in Classpath):

* which classes / frameworks in classpath
* what is the existing config

] Springboot-autoconfig.Jar

* Set logging level to debug, Stack server
* Conditions Evaluate Report (five matches: AutoConfigured) → Based on this (based on existing config classes / framework)

webapp → eg: Tomcat (Embedded Servlet Container)
DispatcherServlet
Default Error page
Bean ↔ JSON (Jackson)
→ Refer Springboot-autoConfig.Jar

DevTools:

* Productivity, Auto refresh (restart) [Pom.xml → Need to restart manually]

Production ready:

* diff Environment: Dev, QA, Test, Stage, Prod
* profiles: diff environment, different config.

application-dev.properties:

logging.level.org.springframework=debug

application-Prod.properties:

logging.level.org.springframework=info

use profiles to take from production properties

application.properties:

spring.profiles.active=prod

trace
debug
info
warning
error
off

trace: trace + all below levels!

Configurations (Recommended: Use class)

```
@Component
@ConfigurationProperties(Prefix='Currency-service')
Public class CurrencyServiceConfiguration {
    Private String url;
            username;
            Key;
    //getters, setters, Constructor
}
```

application.properties:

Currency-Service.url = http://default.com
Currency-Service.username = default
Currency-Service.Key = default

use: Any class

```
@Autowired
CurrencyServiceConfiguration CSC;
//Constructor
```

Use (method): return Configuration / CSC; (POJO ↔ JSON)

Recommended: Reqest: declare in application.properties
Create a class
Use that class: get!

Embedded Servers

	old	new!
	WAR	JAR
	webServer	(Embedded Server)
	JAVA	Java

> Java -jar learn-spring-boot-0.0.1.jar

Actuator (monitoring): pom!

*provides endpoints: /beans: all beans
/health: up & running
/metrics:
/mappings
/actuator: all endpoints available!

management.endpoints.web.exposure.include=*

Spring (vs) SpringBoot (vs) Springmvc:

*Spring: DI, Autowire, Component
*Spring modules & projects: extend ecosystem (integrate JPA, hibernate)
*Spring mvc: module (simplify webapp, Rest API)
@Controller, @RestController, @RequestMapping("/courses")
*Spring Boot: project (production ready app), no many config, Deploy easily,
Log, error handling, profiles.
Starter project, AutoConfig, actuator, Devtools

Spring mvc Rest Services

Spring mvc
↓
mockmvc (test) mockito
↓
Exception handling w/mvc

Bean.java
BeanService
BeanServiceImpl: Add beans in Constructor
getById, getAll, create, edit, delete

@RestController
@Slf4j
@AllArgsConstructor
public class BeanController { → BeanService bs;
@RequestMapping("/api/v3/bean")
public List<Bean> listBeans(){
return bs.getAll();
}
}

webapp: Browser, HTTP, CSS, Rest, Resp, form, Session, Authentication
Springmvc: Dispatcher Servlet, view resolver, mvc validations
Spring boot: Starters, AutoConfig.
Framework/tools: JSP, JSTL, JPA, bootstrap, Spring security, H2, MySQL

http Client (Tools > Create request): Similar to postman (http-api.http)

GET http://localhost:8080/api/item/bean
Accept: application/json

<> 2022-file.json ——→ o/p stored in this file!

Get Path params: /api/v1/beer/{id}

```
@RequestMapping("/api/v3/beer/{beerId}")
public Beer getBeerById(UUID beerId){
}
```

→ works

↓ different name

```
@RequestMapping("url/{beerId}")
Public Beer getBeerId(@PathVariable("beerId") UUID beerId){
}
@RequestMapping(value = "{beerId}", method = RequestMethod.GET)
```

```
@PostMapping
@RequestMapping(method = RequestMethod.POST)
public ResponseEntity handlePost(@RequestBody Beer beer) → Bind JSon!
{
    Beer SavedBeer = beerService.saveNewBeer(beer);
    return new ResponseEntity(HttpStatus.Created);
}
```

@PostMapping

Respond with url (new one created):

```
HttpHeaders headers = new HttpHeaders();
headers.put("Location", "/api/v1/beer/" + SavedBeer.getUUID());
return new ResponseEntity(headers, HttpStatus.CREATED);
```

* Headers → Location

PUT

```
@PutMapping("{beerID}")
public ResponseEntity updateById(@PathVariable("beerId") UUID beerId,
@RequestBody Beer beer) {
}
@DeleteMapping("{beerId}")
public ResponseEntity deleteById(@PathVariable("beerId") UUID beerId){
}
@PatchMapping("{beerId}")
public ResponseEntity updateBeerByPatchId(@PathVariable..., @RequestBody...){
}
```

↳ update only if the param is found in JSon

JSP:

application.properties:

prefix (spring.mvc.view.prefix = /src/main/resources/META-INF/resources/WEB-INF/jsp

spring.mvc.view.suffix = .jsp

Redirect to view:

```
@RequestMapping("say-hello-jsp")
public String sayHelloJSP(){
    return "sayHello";
}
```

Logger

```
private Logger logger = LoggerFactory.getLogger(this.getClass());
method(){
    logger.debug("what to log : " + skill);
}
```

spring-boot-starter-logging

JSTL tags:

```
<c:forEach items="${todos}" var="todo">
    <tr>
        <td>${todo.id}</td>
        <td>${todo.desc}</td>
    </tr>
</c:forEach>
```

History:

1) All code in view (view, flow, DB) : complex, no seperation, hard to maintain

model 1: 2) Seperate concerns : MVC : simple to maintain, (concern: common feature to all con)

model 2 : 3) Front Controller: All request flow in to a central Controller (Front Controller)
Then Control flow to controllers, views.

4) Spring mvc Front Controller – DispatcherServlet (Receive, process HTTP request)

A: Receives request
B: Process Request
B1: Current Controller
B2: Execute
B3: Identify view
B4: Execute view.
C: Return Response

model: only available for that request
↳ Live longer: put it in Session!
(Retain values)

```
@Controller
@SessionAttributes("name")
public class LoginController {
    private AuthenticationService as;
    public LoginController(AS as){ ——— }
    @RequestMapping("login", method = RequestMethod.GET)
}
```

Note: Need to put @SessionAttributes("name") → In all Controllers whichever you want to use that attribute!

---

* Request Scope: Active for a single request (once response sent: request attributes removed from memory (can't use in future): Recommended.

* Session Scope: Details stored across multiple requests.
(Be careful: what you store: takes memory)

Forget it: we are going to set in model & use! (Don't worry about JSP)

@RequestMapping(value = "add-todo", method = RequestMethod.POST)
public String addNewJob (@RequestParam String description, ModelMap model) {
——— model.get("name");
}

↳ For using like this (SessionAttribute) in Controllers.

Validations:

1. Spring Boot Starters Validation
2. Command Bean (Form backing object): 2 way binding (todo.JSP, TodoController)
3. Add bean validation (Todo.java)
4. Display errors in view.

<%@ taglib prefix = "form" uri = "http://www.springframework.org/tags/form" %>

<form:form method = "post" modelAttribute = "todo">
Description: <form:input type = "text" path = "description" required = "requ
</form>

<form:input type = "text" path = "done"

Validation to bean:

@Size(min = 10, message = "Enter atleast 10 characters")

@RequestMapping(value = "addToDo", method = RequestMethod.POST)
public String addNewToDo (ModelMap model, @Valid Todo todo) {
String username = (String) model.get("name")
———
}

To bind!
(error page!)

——— (ModelMap model, @Valid Todo todo, @BindingResult result)
{
if (result.hasErrors()){
return "to-do"; → Go to same page / show error in page.
} }

JSP (Show error):

Description: <form:input type="text" path="description" required="required" />

<form:errors path="description" />

↳ Shown near that field!

Predicate:

```
Predicate<ToDo> predicate = todo -> todo.getId() == id;
```

Arrow function: `() -> { }`

Password:

1) InMemoryUserDetailsManager → createUserDetailsManager() { ... }

2) with encryption:

```
@Bean
Public PasswordEncoder passwordEncoder(){
    return new BCryptEncoder();
}

@Bean
public InMemoryUserDetailsManager createUserDetailsManager() {
    Function<String,String> pe = input -> passwordencoder().encode(input);

    UserDetails ud = User.builder()
        .passwordEncoder(pe);
        .username("")
        .password("")
        .roles("")
        .build();
    return new InMemoryUserDetailsManager(ud);
}
```

Other way: use DB (H2/mySQL) with default Configuration!

Get username:

```
Authentication auth = SecurityContextHolder.getContext().getAuthentication();
return auth.getName();
```

```
findByUserName(String username){
    Predicate<Todo> predicate = todo -> todo.getUserName().
                                equalsIgnoreCase(username);
    return todos.stream().filter(predicate).toList();
}
```

predicate → matching condition

return → get from stream.

H2 DB

spring.datasource.url = jdbc:h2:mem:dfa4c38b-e53a-4b92-9b0f-cb3e078ab9a1

Console: Put db name, (JDBC URL) : for each request, filter chain starts!

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) {
    http.authorizeHttpRequests(auth -> auth.anyRequest().authenticated());
    return http.build();
}
```

`http.formLogin(withDefaults());` → Show form login!

options:
- `http.csrf().disable();`
- `http.headers().frameOptions().disable();`

@Entity
@Id
@GeneratedValue
→ Automatically creates DB & Schema

- *data.sql
- *schema.sql

spring.jpa.defer-datasource-initialization=true → Tables created when entity present
data.sql : executed before table creation

Change! (After)

Spring web, JDBC, JPA, H2 database

JDBC: lot of sql queries + Java code

Spring JDBC: Lot of sql queries + less Java code.

JDBC

```
public void deleteToDo(int id) {
    PreparedStatement st = null;
    try {
        st = db.conn.prepareStatement
            ("delete from todo where
                id = ?");
        st.setInt(1, id);
        st.execute();
    } catch (SQLException e) {
        logger.fatal("query failed", e);
    } finally {
        if (st != null) {
            try {
                st.close();
            }
            catch (SQLException e) {}
        }
    }
}
```

Spring JDBC

```
jdbcTemplate.update("delete from
    todo where id=?", id);
```

SpringJdbcTemplate.update('') → Insert
→ Update
→ Delete queries.

Execute at Startup: (CommandLineRunner)

```
public class CourseJdbcCommandLineRunner implements CommandLineRunner
{
    @Override
    public void run(String... args) {        → Run at startup!
        //TO DO [Load DB]
    }
}
```

```
jdbcTemplate.queryForObject(SELECT_QUERY, new BeanPropertyRowMapper<Course>(Course.class),
                            (null), id);
```
↳ 'mapping' Table : Bean/entity.

Spring JPA

```
@Repository
public class CourseRepo {
    private EntityManager em;
    //Constructor;
    public void insert(Course course) {
        em.merge(course);  → uses entity mapping (Course.java) binding....
    }
}
```

```
em.find(Course.class, id);
em.remove(course);
```

`spring.jpa.show-sql=true` : Show sql queries in console log!

Don't worry: Hibernate! (main)

JDBC : Lot of sql + java

Spring JDBC : Lot of sql + less java

JPA : Don't worry about queries, Just map table

Spring Data JPA : Let's make JPA more simple!

Hibernate (vs) JPA:

* JPA: API (Interface / Specification): how to define entities, manage entities
* Hibernate: Implementation of JPA: use hibernate annotations.

We are using JPA annotations: why? → Not wanting to lock in with hibernate

[Use JPA, only implementation: hibernate → hibernate
Future Other : Use that one!]
(TOPLink)

> Note: Always use JPA annotations

Spring Autoconfiguration magic: Initializes Spring Data JPA frameworks
Launch & connect to DB Inmemory
Launch data.sql / schema.sql

Docker

* Launch mysql using Docker

→ Root Password → User

```
docker run --env MYSQL_ROOT_PASSWORD=dummy --env MYSQL_USER=todos
--env MYSQL_PASSWORD=dummy
--env MYSQL_DATABASE=todos → Database
--name mysql --publish 3306:3306 → publish in a port.
mysql:8-oracle
```

↳ Supported for all OS

```
jdbc:mysql://localhost:3306/todos
spring.datasource.url=http://localhost:3306
spring.datasource.username=todos
spring.datasource.password=dummy
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
```

↳ Class acts as a bridge b/w JDBC & SQL!
Allow hibernate to generate SQL optimized for a particular DB
In so to framework: hibernate query → mysql

```
spring.jpa.hibernate.ddl-auto=update
```

↳ 'maintain tables': No need to create each time dropping.

```
spring.jpa.defer-datasource-initialization=true
```

↳ execute 'data.sql' after DB creation!

Rest best practices: (Customer POV)

* validation, I18N
* Exception handling
* HATEOS
* versioning
* Documentation
* Content negotiation

* @RestController : expose Rest API (JSON, XML)
* @RequestMapping(method = RequestMethod.GET, path = "/urL")
* @PathVariable (/api/v3/invoice/{id})
*

---

* Logging: Logs Keep changing frequently.

1. How requests handled: Front Controller Pattern.
   * DispatcherServlet routes to respective Controller. Based on mapping!
   * AutoConfiguration : DispatcherServlet AutoConfiguration

2. How object ⟶ Json
   * @ResponseBody + Jackson Http Message Converters
     - @RestController has @ResponseBody annotated class level (return bean as is)
     - Message Conversion : using Jackson.
   - JacksonHttpMessageConvertersConfiguration

3. Who is Configuring Error mapping?
   - errorPage : ErrorMvcAutoConfiguration

4. How all jars available?
   - Spring boot Starter web (mvc, web, tomcat, json)

```
URI location = ServletUriComponentBuilder.fromCurrentRequest()
                .path("/{id}")
                .buildAndExpand(savedUser.getId())
                .toUri();
return new ResponseEntity.created(location).build();
```

---

whitelabel error page with stacktrace: (Any errors : Thrown / unexpected)

```
@ResponseStatus(code = HttpStatus.NOT_FOUND)
public class UserNotFoundException {
    public UserNotFoundException(String message){
        super(message);
    }
}
```

Exception trace: Due to dev tools

Predefined error JSon Structure: (enterprise app)

ErrorDetails
- timestamp
- message
- details

Constructor
getters, setters

Use the manual structure for error messages:

@ControllerAdvice → Applicable to all Controllers & test Controllers

```
public class CustomizedResponseEntityExceptionHandler extends ResponseEntityExceptionHandler
{
    @ExceptionHandler(Exception.class) → All exceptions
    public final ResponseEntity<Object> handleAllException(Exception ex, WebRequest req)
    {
        ErrorDetails ed = new ErrorDetails(LocalDate.now());
        new ResponseEntity(errorDetails, HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

@ControllerAdvice : Special type of @Component for classes with

- @ExceptionHandler
- @InitBinder
- @ModelAttribute methods to be shared across controllers

Validations (Spring validation)

```
createUser(@Valid @RequestBody User user)
{
    ____
}
```

↳ To Consider @Size, ... validations while binding

```
ex.getErrorCount();
ex.getFieldError().getDefaultMessage();
```

Rest API documentation

* Resource, Action, Req/Response format
* Challenge: Accuracy, Consistency
* options: manual, From code [Swagger & openAPI] : Swagger UI + openAPI Std.

(Spring config, annotation, class structure)

Spring 3 - openAPI:

* Use Github: 2.x : Pom.xml
* User - resource, hello-world - Controller, Schemas!

Resources!

/v3/api-docs : OpenAPI Specification: Json

```
{
    openapi - 3.0.1
    info : title, version
    server: path, desc
    Paths : endpoints
    Components : Schema
}
```

Content negotiation (JSon, Xml):

* content type, language.
* Using Rest : Content negotiation

mime type : application/xml, application/json

accept language header: en, nl, fr, ...

xml: Jackson-dataformat-xml

I18N:

* HTTP request header : Accept-Language : natural, locale : en, nl, fr, ..

messages.properties (resources)

good.morning.message = Good Morning

messages_nl.properties:

good.morning.message = Goedemorgen

```
@GetMapping("/hello")
public String sayHello() {
    Locale locale = LocaleContextHolder.getLocale();
    return messageSource.getMessage("good.morning.message", null,
                                    "Default Message", locale);
}
```

Versioning: (100s of users : can't implement directly (Broken Va) : new version)!

* Breaking changes: Versioning Rest API
* Options: URL (different URL), request param, request headers, mime type

1. URL: /v1/person, /v2/person ⟶ Twitter
2. Request param: www.localhost.com/person?version=1] Amazon

```
@GetMapping(path="/person", params="version=1")
@GetMapping(path="/person", params="version=2")
```

3. custom header versioning: (microsoft): same URL & headers = [X-API-Version=1]

```
@GetMapping(path="/person", headers="version=1")
@GetMapping(path="/person", headers="X-API-Version=1")
```

4. mime type versioning (Github): content negotiation/accept headers.

application/vnd.company.app-v1+json

application/vnd.company.app-v2+json

```
@GetMapping("/person", produces="application/vnd.company.app-v1
                                 +json")
```

What to choose ⟶ (factors)

* URI pollution (lot of pollution) URI's : method 1
* misuse of HTTP headers (never meant for versioning) : 3, 4
* Caching: (based on versioning): 3, 4 (Same URL): can't cache! ↳ Look at headers.
* Execute browser (on): 1, 2 (yes), 3, 4 ↳ need Cmd line/Rest Client!

1,2: Just change URL / add parameter
3,4: Need command line / Rest client

*API documentation: 1,2 (easy) tools, 3,4: many not support!

"No perfect solution"

---

1. Think while building
2. 1 enterprise : 1-version!

HATEOS:

* Hypermedia as the Engine of Application State
* See data, perform actions (Links)
* Tell consumers : how to perform subsequent actions on those resources

Implementation:

1. Custom format & implementation: Difficult to maintain (bean)!
2. Standard implementation (HAL: Json hypertext App language):
   Simple format for consistent + easy to hyperlink b/w resources.
   * HAL
   * Spring HATEOS

Pom.xml + Changes in Bean:

User Resource : wrap using EntityModel (wrap domain objects : To add links)

```
@PostMapping("/users")
public EntityModel<User> retrieveUser(@PathVariable int id) {
    EntityModel<User> em = EntityModel.of(user);
    return em;
}
```

```
WebMvcBuilder link = linkTo(methodOn(this.
        getClass()).retrieveAllUsers());
```

1. wrap using EntityModel
2. link methods using WebMvcBuilder
3. em.add(link.withRel("all-users"));
4. Return EntityModel

```
@PostMapping("/users")
public EntityModel<User> retrieveUser(@PathVariable int id) {
    EntityModel<User> em = EntityModel.of(user);
    WebMvcBuilder link = linkTo(methodOn(this.getClass()).
            retrieveAllUsers());
    em.add(link.withRel("all-users"));
    return em;
}
```

```
{ id
  name
  links: { "all-users": { "href": "_____" } }
}
```

Serialization: object → Stream (JSON/XML)] Jackson!
Deserialization: JSON/XML → object

Customize Rest Response:

* Field name: @JsonProperty("name")
  private String userName;

* Filter: only selected fields (password) → class level / field level
  Same: → Static @JsonIgnoreProperties, @JsonIgnore
  Based on decision: → Dynamic @JsonFilter with FilterProvider

Dynamic filtering:

* Can't apply to bean (As based on API: need to filter) → Do it in RestAPI

MappingJacksonValue class: For passing instruction to Jackson Converter

```
@GetMapping("hello")
public SomeBean sayHello(){
    SomeBean sb = new SomeBean("value1", "value2", "value3");
    MappingJacksonValue mjv = new MappingJacksonValue(SomeBean);
    return someBean;
}
```

```
SimpleBeanPropertyFilter filter = ...
mjv.setFilters(new SimpleFilterProvider()
    .addFilter("SomeBeanFilter", filter));
```

SomeBean → field1, field2, field3

```
@GetMapping("/filtering")
public MappingJacksonValue filtering(){
    User user = new User("field1", "2", "3");
    MappingJacksonValue mpj = new MPJV(this);
    SimpleBeanPropertyFilter filter =
        SBPF.filterOutAllExcept("f1", "f2");
    FilterProvider filters = new SimpleFilterProvider().addFilter
        ("SomeBeanFilter", filter);
    mpjv.setFilters(filters);
    return mpjv;
}
```

```
@JsonFilter("SomeBeanFilter")
SomeBean{
    field1;
    2;
    3;
}
```

Actuator

* monitor & manage app in production
* Spring-boot-starter-actuator

1. Beans
2. Health
3. Metrics
4. Mappings

management.endpoints.web.exposure.include=*

https://localhost:8080/actuator

* /actuator/beans : All beans, dependancies
* /actuator/env : environment details (JVM, classpath, System env)
* /actuator/metrics

HAL explorer

```
{
  "id" : 1,
  :
  "_links" : {
    "all-users" : {
      "href" : "http://localhost:8080/users"
    }
  }
}
```

* HAL explorer : An API explores for Restful Hymedia APIS using HAL
* Enable : non tech team play with APIS
* Spring boot HAL explorer : Autoconfig HAL explorer for Spring boot projects.

http://localhost:8080/explorer/index.html#/url=
http://localhost:8080/actuator

↳ links
Response
Headers...
→ make requests!
(explore!)

JPA

User:

```
@OneToMany (mappedBy="user")
@JsonIgnore
Private List<Post> posts;
```

Post:

```
@ManyToOne @JsonIgnore
(fetch = FetchType.LAZY)
private User user;
```

spring.jpa.show-sql=true

```
@PostMapping ("/jpa/users/{id}/posts")
public ResponseEntity<Object> CreatePostForUser (@PathVariable int id, @Valid
                                                 @RequestBody Post post) {

    user ≠ null → exception
    post.setUser(user);
    post savedPost = postRepo.save(post);
    URI location = ServletUriComponentsBuilder.fromCurrentRequest()
                   .path("/{id}")
                   .buildAndExpand(savedPost.getId());
                   .toUri();
    return ResponseEntity.created(location).build();
}
```

* Default (Spring Security) : All URLs are password protected

```
spring.jpa.hibernate.ddl-auto = update
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.
                                          MySQLDialect
spring.security.username = root
spring.security.user.password = password
```

* Filter (interceptr) : Spring Security

  * execute series of filters (filterchain) : Authenticate, CSRF(post, Put)
  * override: Returnit filter chain again!

```
@Configuration (class having 1/more bean methods)
public class SecurityFilterConfiguration {

    @Bean
    Public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

        return http.build();

    }
```

authenticate:

```
http.authorizeHttpRequests(auth -> auth.anyRequest().authenticated());
http.httpBasic(withDefaults());
                  ↳ customizer
   ↳ popup (asking password)! [no need to show webpage for signin]

http.csrf().disable(); →
```

```
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

Non client browser: disable CSRF
browser clients: need CSRF!

- Modern JS
- React fundamentals
- Components
- State
- Routing
- Rest API
- Authentication
- Counter example
- Todo management app

Frontend: Modern JS (React)
Backend: Spring boot
Database: mySql
Authentication: JWT (Spring Security)

Full Stack:

* Diff languages, frameworks, build/other tools

Javascript:

* Earlier versions: hard to maintain code
* ES5: 2009, ES6 2015, ES7 2016, ES13 : 2022, ES14 (2023)
* ES (ecma script) : Standard JS Ecmascript, JS : Implementation.

node JS, npm:

```
node --version
npm --version
```

npm package manager:

* npm init : Create new project
* npm install jquery [package.json] → npm install versions! packages!
* node_modules: Folder of temporary dependencies.

React

* Popular JS library to build SPA (Single page application) : load part of page.
* Angular, vueJS
* React: Facebook (Component based)
* React native: Ios, Android
* → Create react app (mac, windows, Linux) : JS, npm
* → npx : package executor (execute JS packages directly without installing)

Create-react-app

```
>npx create-react-app todo-app
```

(todo-app: name)

```
>run: npm start (development mode) : use chrome [Recommended]
> npm test (Run unit tests)
> npm run build (Production: minify, optimize (performance))
```

↳ React: Lot of dependencies! files: 3 files!

```
> npm install --save react-router-dom [add dependencies]
```

VScode

* Toggle view: Ctrl+B (explorer)
* Ctrl+P : Launch a file
* Search: Search in code, find & replace!

How Immediately changing: npm automatically builds, renders!

React folder Structure:

* README.me : documentation
* package.JSon : dependencies ['ll" to Pom.xml]
* node_module : folder (all dependencies)
* React Initialization (3 files):
  * public/Index.html : root div (loaded first) /page
  * Src/Index.css : styling for entire application
  * Src/Index.JS : React app. Load app Component.
  * Src/APP.JS → code for app Component
  * Src/App.test.JS : Unit test for JS Component.
  * Src/App.CSS : Styling for app Component

Alongside production code (Diff from Java) ←

Components

* modularize : Menu, Footer, Header, welcome page, Login, Logout, Todo.
* Reuse, easy to understand.
* App Component : loaded first

Component → View (JSX/JS) : → HTML
Component → Logic (JS)
Component → Styling (CSS)
Component → State (Internal data store)
Component → Props (pass data)

Index.JS

```
root.render(
    <React.StrictMode>
        <App />
    <React.StrictMode>
);
```

Name of a Component : Always starts with capital letter

First Component:

* All other Components : child Component of React's App Component.

```
function App() {
    return (
        <div className="App">
            <FirstComponent />
            <ThirdComponent />
        </div>
    );
}

function FirstComponent() {
    return (
        <div> Helloworld </div>
    );
}
export default APP;
```

→ Add Components in App()

→ with in parenthesis return from Component.

Function Components : As they use functions

Class Components:

→ No need for parenthesis! (Just a name)

→ extends Component

→ render method!

```
class ThirdComponent extends Component {
  SecondComponent() {
    return (
      <div className="ThirdComponent">Third Component</div>
    );
  }
}
```

```
import {Component} from 'react'
```

---

* module: Javascript file
* when function Component? class Component?

State: Built in React object to contain data / info about Component

Earlier version: only class Component can have state!

16.8 : Hooks were introduced (Allowed State to function Component too)!

After 16.8 : mostly use functional components

(no real need for going for class components)

Parenthesis : what is we returning

* JSX : Javascript xml (Strict : close tags mandatory)
  * only 1 top level JSX tag
  * multiple : wrap them in a shared tag </> ↳ empty wrapper

* each function / class Component returns JSX (view info)
*

```
{ return (
    <div>
    ==========
    ==========
    </div>
  );
}
```

Enable JSX enabled in React project:

* Different browser : diff levels of modern JS feature
* Ensure backward Compatibility : Babel JS.
* Babel understands & Converts JSX!

All Older browsers : understands JSX!

Babel: JS Compiler

Babel JS:

* Converts JSX ⟶ JS
* `<h1 className='Something' attr='10'>Heading</h1>`

`<parent attr1='1'><child><Another child></Another child></child></parent>`

`React.CreateElement('h1', {className: 'Something', 'attr': '10'}, 'heading');`

Attributes — Value

Why parenthesis:

* makes returning Complex JSX Components easier
*

work	Recommended
return <Com1/> <Com2/> <Com3/> 	return (==);

'Same line as return: work'

```
return
  <div>
  </div>
```

→ won't work as not starting on same line.

Custom Components: Should start with Upper case letter

`<div>` works
`<DIv>` won't → Even it works in HTML, not in JSX.

* JSX: className instead of class!
*

Rules

1. only 1 upper tag (wrapper)
2. use parenthesis (even other way works)
3. Small case: default Components/tags,
   Upper case: Custom Components
4. className instead of 'class' attribute!

JS/ React practices

1. Seperate file (JS) / module: for react modules → import FirstComponent from './myComp/FirstComponent'
2. Can't have hyphens in React Javascript!

Separate module: FirstComponent.JSX

```
export default function FirstComponent() {
    return (
        <div className='FirstComponent'> First </div>
    );
}
```

Why braces (when to use):

* a module can't have more than 1 default export of Component.
* import {Comp5} from './Comp/FirstComp'

Default import: import Comp1 from './Comp/Comp1'
Named import: import {Comp2} from './Comp/Comp1'

Javascript:

* Optional: Semicolon
* Dynamic objects + we can store data in an object + function.

```
const Person = {
    name: 'Ram',
    address: {
        line1:
        city:
    },
    profiles: [twitter, fb, ins]
    printProfile: () => {
        person.profiles.map(
            profile => console.log(profile)
        )
    }
}
```

CSS in JSX

```
<button className='CounterButton'
    onClick={incrementFunction()}
    style={{fontSize: '30px'}}
```

↳ Double braces

Other way:

```
const stylingButton = {
    fontSize: '30px',
    backgroundColor: '#0045b',
    width: '100px',
    margin: '10px',
    color: 'white',
    padding: '15px'
};

<button className = 'counterButton' onClick = {incButton()}
style = {stylingButton} > +1 </Button>
```

css file

Counter.css

```
.counterButton {
    font-size: '16px';
    background-color: '#0045ab';
    margin: '10px'
    width: '100px';
    color: 'white';
    padding: '15px';
    border-radius: '30px';
}
```

→ css: no need for ' ' in css!

import './Counter.css' → 'use'

Functionality: (State of the Component)

* Built in React object: to contain data / info about the component
* Earlier: only class components have state (complex)!
* 16.8: Hooks (easy to use, can be used with functional comp)

useState: hook → current state
→ function to update state

* each instance of a component has its own state.
* can share states b/w component.

const state = useState(0) → returns [0, f]
→ value
→ function to update state.

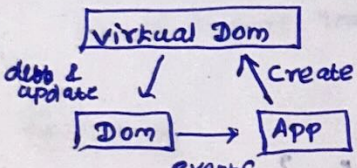
```
export default function Counter(){
    const [count, setCount] = useState(0);
    function incrementCounter(){
        setCount(count+1)
    }
}
```

```
return (
    <div className='Counter'>
        <span className='Count'>{count}</span>
        <div>
            <button className='CounterButton'
                onClick={incrementCounterFunction}>+1</button>
        </div>
    </div>
)
}
```

React background

* Dom (Document Object Model): updated by React! [Manual: huge hierarchy]
  ↳ slow, complex

* React uses Virtual Dom: Keeps it in memory [Identify changes, Sync with HTML] (Render)
  ↳ (Rerender): Do diff!
  ↳ Entire Dom is not rendered: Just update!



: Efficient!

Props (Properties) ⟶ Constant!

```
<button prop1='value1' prop2='value2'/>
```

function

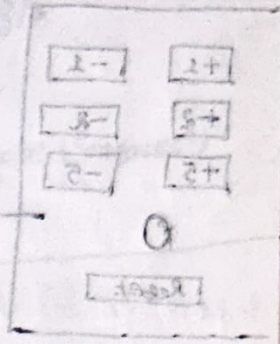
```
export default function Counter({by}) {
    const [count, setCount] = useState(0)

    function incrementCounter() {
        setCount(count + by);
    }
    function decrementCounter() {
        setCount(count - by);
    }
    return (
        <div className='Counter'>
            <span className='Count'>{count}</span>
            <div>
                <button className='CounterButton' onClick={incrementButton}>
                    +1{by}</button>
                <button className=
                    >-1{by}</button>
            </div>
        </div>
    )
```

JS

1+1 = 11

Need type to do it correct

```
Counter.PropTypes = {
    by: PropTypes.Number
}
Counter.defaultProps = {
    by: 1
}
}

function APP() {
    return (
        <div className='APP'>
        <Counter by={2}/>
        <Counter by={3}/>
    )
}
```



Common State for buttons

* One state for all counters : move <Counter ...> to a common class!

```
function Counter() {
    const [Count, SetCount] = useState(0)
    function updateCounter(by) {
        SetCount(Count + by)
    }
    return (
        <>
        <span>{Count}</span>
        <CounterButton by={1} updateMethod={updateCounter}/>
        <>
    )
}
```

Passing 2 properties

Child can't access parent's method: pass it as property!

```
Counter({by, updateMethod})
```

Properties!

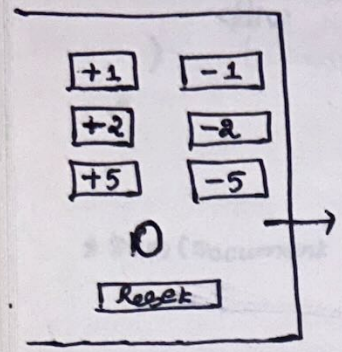
```
function incrementCounter() {
    SetCount(Count + by);
    updateParentMethod(Count + by);
}
decrement() {...
    updateParentMethod(Count - by);
}
```

React extension

* Inspect Component hierarchies (root, sub Component)
* See & edit properties, state, debug!

Reset button:

* Reset all child (our case: only 1 Counter: root)
* Reset the root Counter [Directly in root/parent]; No need to set in a button!



Counter (parent)
each button: A module!

'No need for button's to have state' : Only root (Counter)

onClick = { () ⇒ IncrementMethod(by) }

'Need to use arrow function to pass'
why? [Not reference: Just pass by value]

TodoApp.jsx

To Do App

```
export default function TodoApp () {
    return (
        <div className = 'TodoApp'>
            ToDo Management APP
            <LoginComponent />
        </div>
    )
}
function LoginComponent() {
    return (
        <div ClassName = 'Login'>
            div
                div
                    Label: User name
                    Input : text
                </div>
            </div>
                password
                submit
        </div>
    )
}
```

'import './TodoApp.css'

Input changes: onChange

* when changed: change state! (else Read only) : As value changes (undefined → defined)

* Controlled Component:
  * Hold state in React (Sync Dom with React virtual Dom)

```
<input type='text' name='username' value={username}
       onChange={handleUserNameChange} />

function handleUserNameChange(event) {
    SetUsername(event.target.value);
}
```

→ from useState('default')

'So Sync with React's virtual Dom'
Why? we are using {username}: React's state

Show Component: Based on Scenarios (Login)

* Can't use if directly in a react jsx : helper function, Functional expression
  Ternary & logical conditioning (4ways)

```
Const [canShowSuccessMessage, setCanShowSuccessMessage] = useState(false)

function handleSuccessMessage(event) {
    SetUsername(event.target.value)
}
function handleSubmit() {
    if (username === 'rno28' && password === 'dummy') {
        SetShowErrorMessage(false)
        SetShowSuccessMessage(true)
    }
}

<button type='button' onClick={handleSubmit}>Login</button>
<input type='text' name='username' value={username}
       onChange
```

```
function WelcomeComponent() {
    if (ShowSuccessMessage) {
        <div> . . . </div>
    }
}
```

→ return null (else)

<WelcomeComponent />

Logical (using AND operation)

true && 'Ranga' ⟶ 'Ranga'

false && 'Ranga' ⟶ false

{ShowSuccessMessage && ——— }

React router dom

> npm install react-router-dom

```
import {BrowserRouter, Routes, Route} from 'react-router-dom'

export default function TodoAPP() {
    return (
        <div className='TodoAPP'>
            <BrowserRouter>
                <Routes>
                    <Route path='/' element={<LoginComponent />} ></Route>
                    <Route path='/login' element={<Login />} />
                </Routes>
            </BrowserRouter>
        </div>
    )
}
```

```
import {useNavigate} from 'react-router-dom'
```

↳ Navigate to different page!

```
const navigate = useNavigate();

function handleSubmit() {
    if (username === 'U' && password === 'dummy') {
        navigate('/welcome')
    }
}
```

```
<Route path='*' element={<ErrorComponent />} />
```

↳ Show Some error page!

navigate(`/welcome/${username}`)

↳ Shift + Tilde.

List ToDO:

```
todos.map(
    todo => (
        <tr key={todo.id}>
            <td>{todo.id}</td>
            <td>{todo.desc}</td>
        </tr>
    )
)
```

→ mentioning key (each child need to have key) for efficient Dom manipulation!

Manage todos Go here

↳ Refresh/load entire page!

Best way:

<Link to='/todos'> Go here </Link>

BookStrap

npm install bootstrap.

Note: <Link> : only used when the Component is a part of the Router

---

Share States across Component (Hooks):

* Create Context: Share it across multiple components.

AuthContext.js

```
const AuthContext = createContext()        → Create

function AuthProvider({children}) {
export const [num, setNum] = useState(0);
    return (
        <AuthContext.Provider value={{num}}>
            {children}
        </AuthContext.Provider>
    )
}

export default AuthProvider;
```

↳ Share it with children

wrap the children

```
export default function TodoApp() {
    return (
        <div>
            <AuthProvider>
                <HeaderComp />
                <Routes>
                </Routes>
            </AuthProvider>
        </div>
    )
}
```

```
export const AuthContext = CreateContext()
export default function AuthProvider ({children}){

    const [num, setNum] = useState(0);

    return (
        <AuthContext.Provider value={{number}}>
            {children}
        </AuthContext.Provider>
    )
}
```

Usage:

```
TodoAPP() {
    return (
        <div ——>
            <AuthProvider>
                :
            </AuthProvider>
        </div>
    )
}

function HeaderComponent() {
    const authContext = useContext(AuthContext)
    num = authContext.number
}
```

→ Can use in any children Component.

Directly use (Create hook in AuthProvider)

```
export const useAuth = () => useContext(AuthContext)
```

→ I want function! That's why () =>

useAuth is a function returning AuthContext

```
HeaderComponent() {
    const authContext = useAuth()
}
```

Enable & disable menu (Based on login)

Objects in JS:

{val1, val2, val3} ⟶ {val1: 10, val2: 20, val3: 30}

{val, str} ⟶ {val: 10, str: 'This is awesome'}.

```
return (
    <AuthContext.Provider value={{num, isAuthenticated, setAuthenticated}}>
        {children}
    </AuthContext>
)}
```

Show menu only if logged in:

```
const authContext = useAuth()
isAuthenticated = authContext.isAuthenticated

{isAuthenticated && <Link to="/welcome/${-3}">Home</Link>}
```

Don't allow any function to setAuthenticated! (Don't pass) in context

Protected URLS

```
function AuthenticateRoute ({children}) {
    const AuthContext = useAuth()
    if (authContext.isAuthenticated){
        return children
    }
    return <Navigate to="/" />
}

return (
    <div>
        <AuthProvider>
            <BrowserRouter>
                <HeaderComponent />
                <Routes>
                    <Route path="/" element={<LoginComponent />} />
                    <Route path="/welcome/:username" element={
                        <AuthenticateRoute>
                            <WelcomeComponent />
                        </AuthenticateRoute>
                    } />
                </Routes>
            </BrowserRouter>
        </AuthProvider>
    </div>
)
```

Cross origin requests

* Default fails : Allow only localhost : 3000 (To spring Boot)
* add CorsMapping : Blocks all CORS

MyAPP.java

Public Static void main (String[] args) {

}

→ custom Configuration!

@Bean
public WebMvcConfigurer corsConfigurer() {
    return new WebMvcConfigurer() {
        public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/**")
                .allowedMethods("*")
                .allowedOrigins("http://localhost:3000");
        }
    };
}

Axios

axios.get('http://localhost:8080/hello-world-bean')
.then((response) => successfulResponse(response))
.catch((error) => errorResponse(error))
.finally(() => console.log('cleanup'))

move to module: (HelloService.JS)

export const retreiveHello = () => axios.get('_____')

↳ 'Need function'

Base URL:

const apiClient = axios.create(
    { baseURL: 'http://localhost:8080' }
);

export const retrieveHelloWorldBean = () => apiClient.get('/hello/${user}')

Todos

a = axios.create({baseURL: 'http://localhost:8080'});

export const retrieveAllUserTodo = (username) => a.get('/todos/${username}');

ListTodoComponent() {
    const [todos, setTodos] = useState([])
    // when to load : Best practice : method useEffect

When should we call refreshToDo: useEffect

```
function refreshTodos() {
    retrieveAllTodosForUserName('in28Minutes')
        .then(response => console.log(response) setTodos(response.data))
        .catch(error => console.log(error))
}

useEffect(() => refreshTodos());  → Load data as soon as rendered the component.
```

Trigger: when it is rendered

Tell useEffect when to render: (effect callBack, dependent : args)

```
useEffect(() => refreshTodos(), [])
```

Delete a todo

* `<td><button className='btn btn-warning' onClick={deleteTodo(todo.id)}>Delete</button></td>`
  - ↳ pass param: need to be arrow function
  - `{() => deleteTodo(todo.id)}`

Show message: Deleted Successfully! (if deleted) + Refresh ToDo

* capture username: when logged in [using AuthProvider state]

update: (Redirect to a new page)

* Retrieve & Show: Details for that ToDo
* After submit/update: update!

Formik: (library)

Build Forms in React! Intuitively.

```
<div>
  <Formik initialValues={{description, targetDate}}
          enableReinitialize={true} onSubmit={updateForm}>
    {(props) => (
      <Form>
        <fieldset className='form-group'><label>Desc:</label>
          <Field type='text' className='form-control' name='desc' />
        </fieldset>
        <fieldset ...><label>Target Date</label>
          <Field name='date' />
        </fieldset>
        <div><button ...>Submit</button></div>
      </Form>
    )}
  </Formik>
</div>
```

Validations:

```
function validate (values) {
    let errors: { description: 'Enter a valid description',
                  targetDate: 'Enter a valid target date' }
    if (values.description.length < 5)
        errors.description = 'Enter...'
    return errors
}

<Formik initialValues = {{ desc, target date }}
    enableReinitialize = {true}
    onSubmit = {updateTodo}
    validate = {validate}
    validateOnChange = {false} ——→ validate type a letter, go to other field ] Not required!
    validateOnBlur = {false} >
```

Validate only when I click Submit

```
function onSubmit (values) {
    console.log (values)
    const todo = { id: id, username: username, description: description,
                   targetDate: targetDate, done: false }
    updateTodoApi (username, id, todo).then (response =>
        {setDesc (res.data.desc)
         setTargetDate ('') })
    .catch (error => console.log (error))
```

navigate to list page

Validation : moment (values.Description).isValid()

↳ 'Library'!

Authentication (Basic)

Pom.xml → spring-boot-starter-security

* Default: All URLs protected: printed password on console [sign in page: web]
* spring.security.user.name = in28
  spring.security.user.password = dummy

@Configuration
class ——— {
    CSRF (disable)
    @Bean
    public SecurityFilterChain filterChain (HttpSecurity http) {
        return http.build ()
    }
}

↳ Configure security!

disable CSRF:

```
http.authorizeHttpRequests(
        auth → auth.anyRequest().authenticated());
http.httpBasic(Customizer.withDefaults()); ⟶ Give me popup (no page)! sign in!
http.sessionManagement(
        session → session.sessionCreationPolicy(SessionCreationPolicy.
                STATELESS));
http.csrf().disable();
return http.build();
}
```

Disable CSRF (:X:) if Stateless!

React

```
export const retrieveHelloWorld = (username) ⇒ apiClient.get('/hello/${username}',
        { headers: {
                Authorization: 'Basic ____'
        }
});
```

All options request for every body:

```
http.authorizeHttpRequests(
        auth → auth.antMatchers(HttpMethod.OPTIONS, "/**")   → Any URL
                .permitAll()
                .anyRequest().authenticated())
http.httpBasic(Customizer.withDefaults())
http.sessionManagement(session → session.sessionCreationPolicy
                (SessionCreationPolicy.STATELESS))
                .csrf().disable()
http.build();
```

If loggedin: send token (authorization header):

```
Basic aW4gOG1PbnV0ZXx06ZHVbbxK=
```

```
const baToken = 'Basic ' + window.btoa(username + ':' + password)
                                  ↓
                              base64 encoding.
```

```
=(token) ⇒ apiClient.get(url) {
        headers: {
                Authorization : token
        }
})
```

async:

```
function login(uname, pwd) {
    const baToken = "Basic " + window.btoa(uname + ":" + pwd)
    executeBasicAuth(baToken)
        .then(console.log(2))
        .catch(error => console.log(error))
    console.log(1)
}
```

Note: JS → default asynchronous (don't wait)

Console:

1

2 → only logged after response.

> I want: Complete response before going further!

```
async function login(uname, pwd) {
    const res = await executeBasicAuth(uname, pwd)
    if(res.status === 200) {
    }
}
```

→ async function

→ await: wait for Completion!

```
async function handleSubmit() {
    if(await authContext.login(uname, pwd)) {
    }
    else {
    }
}
```

→ Calling function!

Store token in context: (AuthContext.js)

```
const [token, setToken] = useState(null)

if (response === 200) {
    setToken(token)
    setAuthenticated(true)
    setUserName(true)
}
```

```
catch {
    logout()
}
```

can't make changes to every api (To add token) : Generic way :

APIclient.js

```
import axios from 'axios'
import {apiClient} from "./APIclient"        → In other classes, to use APIclient.js

export const apiClient = axios.create(
{
    baseURL: 'http://localhost:8080'
}
)
```

when logged in : Intercept & add token in apiClient :

```
if (response.status == 200) {
    apiClient.interceptors.request.use(
        (config) => {
            console.log('adding token')
            config.headers.Authorization = token
            return config
        }
    )
}
```

JWT (JSON web token) :

* Basic : No expiry, No user details, Easily decoded!
* Custom token : Security flaws, (provider, consumer: understanding)
* JWT (Standard)
    * open, industry standard (B/w two parties : secure)
    * Has user details, Authorization
    * Header :
        * Type : JWT
        * Algo : HS512 (hashing algo)
        * payload : attributes (iss : issuer
          sub : subject
          aud : Audience
          exp : expiration time
          iat : when issued)
        * Custom attributes
    * Signature

JWT + Spring : Complex !

http://localhost:8080/authenticate → send token → use in future.

* Authorization: "Bearer _____"

AuthenticationAPIService.js

```
import {apiClient} from './ApiClient';

export const executeBasicAuthService = (token) =>
        apiClient.get('/basicauth', {
            headers: {Authorization: token}
        })

export const executeJWTExecutionService = (username, password) =>
        apiClient.get('/authenticate', {username, password});
```

```
if(resp === 200) {
    setToken('Bearer ' + response.data.token)
}
```

→ use!

Debugging:

* make sure: any other Authentication like Basic configured.
* Debugger!

---

* JPA, Hibernate: Can switch b/w vendors easily!
* extend JPARepository: Basic implementations there.

Junit

* 1000s of files, millions of code: Check expected behaviours
* options:
    1. System/Integration testing: Entire app
    2. Unit test: method/features (independent)
        - CI (easily identify, fix bug

Junit, mockito

```
@Test
void test() {
    int result = math.calculateSum(new myMath(1,2,3));
    assertEquals(5, result);
}
```

* assertTrue(flag);
* assertFalse(flag);
* assertArrayEquals(new int[]{1,2}, new int[]{1,2});

Junit: Doesn't guarantee test order!

* @BeforeEach
  void beforeEach(){ ⟶ Before every test!
  }
* @AfterEach
  void afterEach(){ ⟶ After every test!
  }
* @BeforeAll ⟶ Static method (class level) ⟶ Run before all methods
* @AfterAll ⟶ Static method (class level) ⟶ Run after all methods

---

Mockito

* great unit test: easy to maintain
* hard: lot of code, multiple dependencies (business layer): can't use data from Database:

1. Stubs
2. mocks

Stub for dataService:

```
class DataServiceStub implements DataService {
    @Override
    public int[] retrieveAllData(){
        return int[] {25, 15, 5};
    }
```

Test class:

```
@Test
void test(){
    DataServiceStub ds = new DataServiceStub();
    SomeBusinessImpl bi = new SomeBusinessImpl(ds);
    bi.retrieveAllData(); ⟶ assertEquals(25, result)
}
```

Stub: Data layer
mock: Alternative (Stub: lot of maintainence)

* Mockito provides mock()

```
@Test
void test(){
    mock(DataService.class); [DataService dsMock]
    SBImpl bi = new SBImpl(dsMock);
    int result = bi.findGreatest();
    assertEquals(5, result);
}
```

returns null: not told what to return

Return mocked data:

when(ds.retrieveAllData()).thenReturn(new int[] {25, 30, 40});

Advantage of mocks:

* easy to write, maintain.

Annotations by mockito:

@Mock → if used mock must! → @ExtendWith(MockitoExtension.class)

Private DataService dataServiceMock;

@InjectMocks

Private SomeBusinessImpl businessImpl;

public class TestClass {

@Test

void findData() {

}

More on mockito

{

List listMock = mock(List.class);

when(listMock.size()).thenReturn(3).thenReturn(2).thenReturn(1)

}

→ 1st time returns 3
2nd time returns 2
From 3rd time onwards returns 1.

Spring Security

→ Principles
→ Authorization, Authentication
→ Fundamentals, FilterChain
→ Form, JWT, Basic authentication
→ Best practices: CSRF, CORS, OAUTH.

Fundamentals:

* Resources: api, webapp, DB
* Identities: Users (access, perform actions against resources)
* How to identify users: Authentication(userid, pwd), Authorization (has access?)

Biometrics
multifactor → Know
→ Possess
→ Combination (Know, Possess)

Chain: Strong as the weak link

6 principles:

* Trust nothing: every request, data: Validate
* Assign Least privileges: design (keep in mind), user + roles, minimum privileges
* Complete mediation: filter! minimize access, mediation!
* Defense in depth: multiple levels (transport, network, OS, App level)
* Simple: architecture, simple system (easy to protect)
* openness of design: easy to identify & fix flaws

→

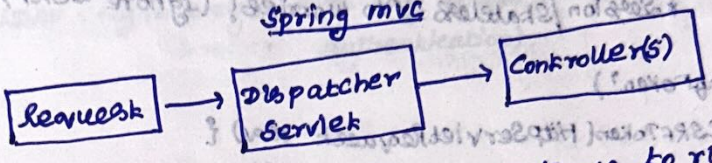
Airport:

* passport/Id
* can't go anywhere wanted.
* Levels/terminals.
* Defense in all levels (multiple times)
* Simple process
* open: standard across airports.

Spring security:

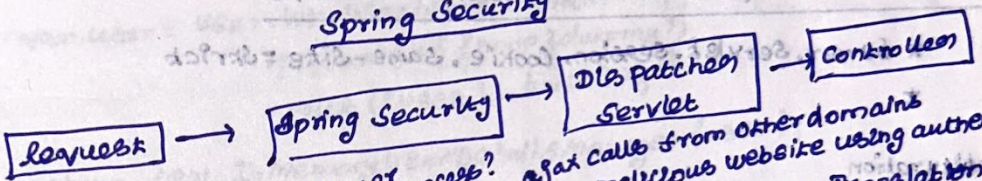
* No 1 priority (webapp/api, microservice)
* Filter chain, Authentication/Authorization manager, JWT, OAUTH...
* Highly customizable, flexible.
* Default: Everything protected.

Spring mvc



* Dispatcher Servlet: Front Controller : Redirects/Route to right Controller.

Spring Security



Spring Security Filter chain

(Authen, Author, CORS, CSRF, Login, Logout, Translation Errors)

→ valid user
→ access?
→ ajax calls from other domains
→ malicious website using authen (Browse)
→ Default: PUT, POST etc... (CSRF enabled)

* CSRF filter
* Logout filter
* Default Login page Generating Filter
* Basic Authentication Filter
* Exception Translation Filter
* Authorization Filter.

Default Configuration:

* Series of filters (reqeuests need to pass)
* order: CORS, CSRF (Basic check filters)
  Authentication filters
  Authorization filters

default Configuration:

* Everything authenticated.
* Form authentication enabled, Basic auth enabled → (Session Cookie JSessionId maintained)
* Test user (log user name), CSRF protection
* CORS: denied.
* X-Frame options = 0 (disabled)
* Default login, logout pages

Basic Auth:

* Not for production, many flaws (64Base encoding: username, pwd): In Auth header
* Basic aW4.....
* easily decoded (Base 64): don't have authorization info, expiry date

CSRF:

* Logged in Bank website: malicious site uses cookie + make request to bank.

1 → * Synchronizer token pattern: Token for each request
Need the previous req token: PUT, UPDATE req
Spring Security: By default enabled.
↳ hidden form element!

2 → * SameSite Cookie: Strict

1) How to use the CSRF token in Rest API:

* webapp: Spring Security automatically includes.
* Session/Stateless: No worries! (Ignore CSRF) → disable!

```
@GetMapping("/csrfToken")
public String getCSRFToken(HttpServletRequest req) {
    return req.getAttribute("_csrf").toString();
```

2) SameSite Cookie (Set-Cookie: SameSite=Strict):

* Server.Servlet.Session.Cookie.same-site=Strict
*

```
@Configuration
public class Sample{
@Bean
SecurityFilterChain defaultOverrideConfig(HttpSecurity http) throws Exception{
    http.authorizeHttpRequests(auth → {
            auth.anyRequest().authenticated();
        });
    http.sessionManagement(
            session → session.sessionCreationPolicy(SessionCreationPolicy.
                                                    STATELESS));
    http.csrf().disable();
    http.httpBasic();
    return http.build();
}
```

CORS

* Browsers don't allow AJAX calls to resources outside current origin.

```
@Bean
public WebMvcConfigurer CorsConfigurer() {
    return new WebMvcConfigurer() {
        public void addCorsMappings(CorsRegistry reg) {
            reg.addMapping("/**")
               .allowedMethods("*")
               .allowedOrigins("http://localhost:3000");
        }
    };
}
```

eg: localhost: 3000 making request to 8080.

* CORS (Cross Origin Resource Sharing): allow CORS(cross origin domain request)
  1. Global Config: Configure addCorsMappings Callback method in WebMvcConfigurer.
  2. Local Config: @CrossOrigin: Allow from all origin.
     @CrossOrigin(origins = "https://www.in.com")
     ↳ Allow from specific origin.

Storing user credentials

* In memory: Testing
* DB: JDBC / JPA
* LDAP: Light weight directory access protocol (directory services & authentication).

In memory:

```
@Bean
public UserDetailsService userDetailService() {
    var user = User.withUserName("in28min")
                .password("{noop}dummy")
                .roles("USER").build();
    return new InMemoryUserDetailsManager(user);
}
```

JDBC:

Spring Security disables iframe by default

```
http.headers().frameOptions().sameOrigin();
```

default ddl schema for users: (Spring Security)

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.H2)
            .addScript(org.springframework.security.core.
userdetails.jdbc.JdbcDaoImpl.DEFAULT_USER_SCHEMA_DDL_LOC)
            .build();
}

@Bean
public UserDetailService userDetailService(DataSource dataSource) {
    var user = ...;
    var admin = ... .roles("ADMIN", "USER").build();
    var jdbcUserDetailsManager = new JdbcUserDetMan(dataSource);
    jdbcUserDetailsManager.createUser(user);...
    return jdbcUserDetailsManager;
```

Encoding (vs) Hashing (vs) Encryption:

* Encode: one form to another (No key/password): Reversible → Not to secure data but to compress / stream (mp3, wav)
* Hashing: Data → hash (not reversible: one way process) → validate integrity of data. (bcrypt, scrypt), Store hashed password.
* encryption: encode using a key (RSA)

Spring Security: store passwords:

* Hashes like SHA-256: no longer safe (modern systems: can perform billions of hash calculations a second)
* Recommended: 1 way functions with work factor of 1 second. (atleast 1s to verify): Increased Complexity.

Why? Fast: easy to bruteforce

(bcrypt, scrypt, argon 2)

* passwordEncoder Interface!

```
@Bean
public BCryptPasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}

var user = User.withUsername("in28");
    .password("dummy")
    .passwordEncoder(str -> passwordEncoder().encode(str));
```

JWT:

* Basic: no expiry, user/authorization details, easily decodable.
* JWT: open, industry std, secure!

Header: Type: JWT HS512 hash algo

payload: iss, sub, aud, exp, iat, custom attributes

Signature: Includes secret → Combine header, payload..

Symmetric and Assymmetric Key encryption:

- Same Key
- Different Keys

1. Right encryption?
2. Secure (how) encryption key?
3. share encryption key?

* Assymmetric: public key cryptography
  Powerful algo: even have public key Not: privateKey!

JWT flow:

Create JWT → Send server → verify JWT

1. Credentials (uname)	1. Auth header	1. Decode
2. User data (Payload)	2. Bearer token	2. Verify key (public)
3. RSA Keypair (asymmetric)	3. Authorization: Bearer {JWT}	

* Spring OAUTH2 Resource Server (enable JWT Config)

Dependency:
Spring-boot-starter-oauth2-resource-server

1. Create Key pair (KeyPair Generator)/openssl
2. Create RSA Key object using Key Pair (RSAKey)
3. Create JWK Source (JWKSet: JSonWeb Key Set with RSA Key
   JWK Source using JWKSet)
4. RSA public key for decoding (Nimbus JwtDecoder.
   withPublicKey(rsaKey().toRSAPublicKey()).
   build()
5. JWK Source (return new NimbusJWKEncoder(
   JwkSource());

JWT Security Configuration.java

* http.oauthResourceServer(OAuthResourceServerConfigurer::jwt);
* @Create JWT decoder

1. KeyPair:

```
@Bean
public KeyPair KeyPair() {
    try{
        var KeyPairGenerator = KeyPairGenerator.getInstance("RSA");
        KeyPairGenerator.initialize(2048); ——→ KeySize (bit)
        return KeyPairGenerator.generateKeyPair();
    }catch(Exception ex){
    }
}
```

2. RSA Key object (nimbus library: For RSA encoding & decoding):

```
@Bean
Public RSAKey rsaKey(KeyPair KeyPair)
return new RSAKey.Builder(KeyPair.getPublic())
        .privateKey(KeyPair.getPrivate())
        .KeyID(UUID.randomUUID().toString())
        .build();
}
```

3. JWK Source:

```
@Bean
Public JWKSource<SecurityContext> JwkSource(RSAKey rsaKey) {
    var JwkSet = new JWKSet(rsaKey);
    var JwkSource = new JWKSource() {
        @Override
        public List get(JWKSelector JS, SecurityContext c)
                throws Exception {
            return JwkSelector.select(JwkSet);
        }
    };
```

(or)

```
return (JwkSelector, context) ——→ JwkSelector.select(JwkSet);
}
```

4. RSA public key:

```
@Bean
Public JwtDecoder jwtDecoder(RSAKey rsaKey) throws JOSEException {
    return NimbusJwtDecoder.withPublicKey(rsaKey.toRSAPublicKey())
            .build();
}
```

5. Encoder:

```
@Bean
public JwtEncoder jwtEncoder(JWKSource<SecurityContext> jwkSource) {
    return new NimbusJwtEncoder(jwkSource);
}
```

Generate JWT token:

```
@PostMapping("/authenticate")
public void authenticate(Authentication authentication) {
        return authentication;
}
```

→ Just return Authentication object (user, role)

```
private JwtEncoder jwtEncoder;
```

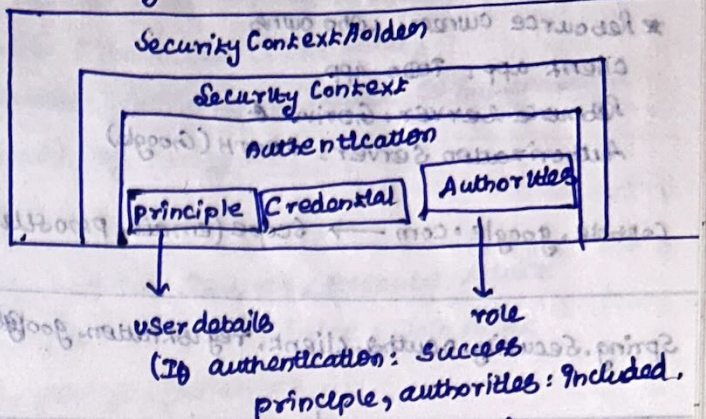
JWT:

```
@Autowired
public JwtAuthenticationResource(JwtEncoder jwtEncoder) {
    this.jwtEncoder = jwtEncoder;
}
@PostMapping("/authenticate")
public void authenticate(Authentication authentication) {
    return new JwtResponse(createToken(authentication));
}
private String createToken(Authentication auth) {
var claims = JwtClaimSet.builder().issuer("self").
                .issuedAt(Instant.now())
                .expiresAt(Instant.now().plusSeconds(60*16
                .subject(authentication.getName())
                .claim("scope", createScope(authentication))
                .build();
    return jwtEncoder.encode(JwtEncoderParameters.from(claims)).
                getTokenValue();
}
private String createScope(Authentication authentication) {
    authentication.getAuthorities().stream()
            .map(a -> a.getAuthority())
            .collect(Collectors.join(" "));
}
```

Spring Security Authentication:

* Spring Security Filter

1. AuthenticationManager Interface (Responsible for Authentication)



(If authentication: success
principle, authorities: included.

2. AuthenticationProvider (JwtAuthentication):
(can have multiple provider type, multiple implementations active at the same time (check all datasources))

3. UserDetailsService: Interface load data

4. How Authentication result stored

SecurityContextHolder > SecurityContext > Authentication > GrantedAuthority

- Authentication: Principle
- GrantedAuthority (roles, scopes) of user

Authorization (global, method level Security):

* Global : requestMatchers ('/users').hasRole('USER')
  - hasRole, hasAuthority, hasAnyAuthority, isAuthenticated.
* method : @EnableMethodSecurity [class level] : Enable

```
@PreAuthorize ("hasRole ('USER') and #username == authentication.name")
@GetMapping ("/users/{username}/todos")
@PreAuthorize ("hasRole ('USER') and #username == authentication.name")
public Todo retrieveToDo (@PathVariable String username){
        return ToDo_List.get (0);
}
@PostAuthorize ("returnObject.username == 'in28'")
```

(Should match!)

method level → pre, post (Recommended)
→ JSR 250 (@EnableMethodSecurity (Jsr250Enabled = true))
→ class level
@RolesAllowed ({"USER"})
→ Secured annotation

```
@EnableMethodSecurity (SecuredEnabled = true,
Jsr250Enabled = true)
class ____ {
}
```

```
@EnableMethodSecurity (Secured
Enabled = true)
@Secured ({"ROLE_ADMIN",
"ROLE_USER"})
```

→ check Authority!

OAUTH

* Give permission to a file in google drive (OAUTH: Authorization standard, also supports authentication now)

* Resource owner: who owns
  Client app: Todo app
  Resource server: GDrive &
  Authorization server: OAUTH (Google)

console.google.com → Scope (email, profile) → Create OAUTH client Id

```
spring.security.oauth2.client.registration.google.client-id = ——
                                              .client-secret = ——
```

↳ Spring does this automatically! →

Spring AOP

* Layered approach to build applications
  * web layer: Web logic, JSon Conversion
  * Business layer
  * Data layer

Cross Cutting Concern ↳ Few common aspects (Security, Logging, performance)

* Cross Cutting Concern: Common aspects (Repeat : Code Complexity)
  1. Implement as seperate aspect (code)
  2. point cut: define when/where should be applied

popular: Spring AOP (not complete): only works with spring beans
AspectJ: Complete (very rarely used): lot of functionality
Intercept any Java class, values / fields.

Logging Aspect:

```
@Configuration
@Aspect
public class LoggingAspect {
    private Logger logger = LoggerFactory.getLogger(getClass());
    //when to Intercept (all calls to a class in specific package)
    @Before("execution(* com.in28.business.*.*(..))") → All class calls!
    public void logMethodCall(JoinPoint joinPoint) {
        logger.info("Method Called!", joinPoint);
    }
}
```

↳ Before every method in this package, run this!

Compile time:

* Advice: what code to execute (Logging, Authentication) : eg: Logger.log("__");
* Pointcut: expression: what method calls to be intercepted
* Aspect: combination of advice & pointcut. [class level]
* Weaver: Framework implements AOP (ensure Advice is implemented for the pointcut at right point in time): eg: Spring AOP, AspectJ.

Runtime:

* JoinPoint: each time pointcut condition is true, execute Advice
  Specific execution instance of an advice: Join Point
  * getClass, getArgs, getTarget ....

@Around: Before & after method execution
@Before
@After [irrespective of success / throws an exception]
@AfterReturning: Success only
@AfterThrowing: exception thrown

```
@Before ("execution (* com.pack.data.*.*(..))")
@After ("execution(* com.in28.**.*(..))")
@AfterThrowing (pointcut = "execution (* ____)", throwing = "exception")
@AfterReturning (pointcut = "execution (* com.in28.*.*.* (..))",
                 returning = "resultValue")

@Around ("execution (* ____)").
public Void findExeTime (ProceedingJoinPoint pj) {
    StartTime;
    pj.proceed(); ---> Return value of method
    StopTime;
    return returnValue; ---> Good practice return the return value.
}
```

When package name changed: need to change all aspects:

* Avoid using pointcuts

```
Public class CommonPointCutConfig {
    @Pointcut ("execution (* ____)") ---> Only change this!
    public void businessPackageConfig() {}
}
```

Use:

```
@After ("com.in28.aspects.CommonPointCutConfig.businessPackageConfig()")
____ method()
{
```

Only beans: (Instead of execution, use beans)

```
@Pointcut ("bean(*Service*)") ——→ All Service beans
public void dataPackageConfigUsingBean() {}
```
DataService
BusinessService

Specific methods: use annotation: (Custom annotation)

```
@Target (ElementType.METHOD)
@Retention (RetentionPolicy.RUNTIME) ——→ Create annotation
public @interface TrackTime {
}
```

Pointcut:

```
@Pointcut ("annotation(Com. . . . . TrackTime)")
public void TrackTimeAnnotation() {}
```

Track specific methods

```
@Around ("Com. ——— .CommonPointCutConfig.TrackTimeAnnotation()")
public object findTime (ProceedingJoinPoint pj) {
    StartTime;
    pj.proceed();
    endTime; ——→ log!
    return resultData;
}
```

→ methods with only TrackTime annotation!

usage:

```
@TrackTime
public class BS1 (){
    ———
}
```

Maven

Dev → Create project
→ maintain dependencies + versions (framework, libraries)
→ Build JAR, deploy
→ Run locally (Tomcat, Jetty)
→ Unit tests
→ Deploy to test environment

Pom.xml : Project object model

Parent Pom:

```
*Parent>
    groupId
    artifact
    version
</Parent>
```
↳ It has dependencies, version config!

* Why so many dependencies: Transitive dependencies
*

Pom.xml ——→ dependencies, properties (Java version, plugin & Config)

* Name of project: GroupId + artifactId (Important: how others find & use)
  - GroupId → Similar to project name/package name
  - artifactId → class name.

Maven build cycle:

validate > Compile > Test > package > Integration test > verify > Install > Deploy

`mvn clean install`

Convention over Configuration:

* pre-defined folder structure (Almost all Java projects follow)
* Consistent (Standard)
* maven central repo: Indexed Jar (groupid, Artifactid)

Similar: repo.spring.io

Download to local repository! (Temp local folder on machine)

Commands:

> mvn --version
> mvn Compile (Just compile)
> mvn test-Compile (only test files) + Compile Source files
> mvn clean (Clean target folder)
> mvn test
> mvn package (Create Jar)
> mvn help:effective-pom (opens effective pom: detailed)
> mvn dependency:tree

Naming of Spring versions:

MAJOR.MINOR.PATCH [-Modifiers]

MAJOR (10 to 11)
minor (no/little work: properties)
patch (no work)
modifiers (optional: Release, milestone, release candidate, Snapshot)

Latest: no modifiers!

M1, M2
RC1, RC2
Snapshot → seperate repo! (versions undecided)
↳ Not for live!

Gradle

* Build, Automate, Deliver better Software, faster
* Java specific but Cross platform (C++, JS, Python, Java...)
* maven: Pom.xml (Plugin → xml config)
* Gradle: programmable (Domain specific language: Kotlin, Groovy) → efficient
* Advantage: fast build (Compile avoidance, advanced Caching)
  - → Speed up 90% (maven builds)
  - → Run only what is necessary
  - → Compile changed files
  - → Uses same project layout as maven
* IDE support evolving

build.gradle → deps
settings.gradle → project name, plugin management

build.gradle

```
repositories {
    mavenCentral()  → maven central
    maven { url 'https://repo.spring.io/milestone' } → another repo!
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-web'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

plugins {
    id 'org.springframework.boot' version '3.0.0-RC1'
    id 'java'
    id 'io.spring.dependency-management' version '1.1.0'
}

group = 'com.pn28'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '17'

tasks.named('test') {
    useJUnitPlatform()
}
```

Gradle plugins

1. Java plugin: (Java compile + test + build capabilities)
   * Default project layout
     - * src/main/java] production Java source, xml....
     - * src/main/resources
     - * src/test/java] test
     - * src/test/resources
   * Key task: build.

2. dependency management (maven like)

org.springframework.boot : spring-boot-starter-web : 1.0.3

groupId — artifactId — version

3. spring boot gradle plugin (execute jar, container images), enables dep management by spring boot dependencies.

↳ No need to mention version (Taken care)!

---

maven (vs) gradle

* maven: Familiar, simple, restrictive
* Gradle: Faster builds, less verbose, (groovy: Java code)

```
tasks.register('helloworld') {
    doLast {
        System.out.println('Hello world');
    }
}
```

gradle run

Docker

* Deployment process described in a document
  - → Hardware setup
  - → OS
  - → Install software (Python, ...)
  - → Setup App. dependencies
  - → Install application.

  manual, timeconsuming, mistake!

* docker --version
* docker container run -d -p 5000:5000 in28min/hello-world:0.0.1m1

  -p: port; image (release: version)

* Simplify deployment process: OS, software, hardware: doesn't matter

Create Containers, run image!

docker container stop <id>

docker container ls → list all containers

Run the same way (local machine, datacenter, cloud)

Why Popular:

* Standard app. packaging for all apps (JS, Python..)
* multi platform support (local, DC, cloud: Azure, Aws..)
* All containers are isolated!

* Docker image is downloaded from docker registry (default: Docker Hub)
* Image: set of bytes, container: running version of image.

-p 5001:5000 → port 5001

* Default: Docker uses own internal network (bridge network)
* map a host port: to access the app.

-d : detached mode

↓

Terminal is not returned if -d is not there!
(need to use new one / stop image)

Docker terminology:

* Image (version of package / software: OS, code, dep)
* Docker registry: Store image
* Docker hub: Registry to host docker images (default registry)
* Docker repo: Images for a specific app (Has tags)
* Container: Runtime instance of a docker image
* Docker file: Instructions (file) to create a docker image.

app.jar

Create image

```
FROM openjdk:18.0-slim        → base image
COPY target/*.jar app.jar     → copy jar to target
EXPOSE 5000                   → port to listen.
ENTRYPOINT ["java", "-jar", "/app.jar"]   → run at container launch.
```

Convention:

Dockerfile. (file) → /target

mvn install

docker build -t in28min/hello-world-docker:v1 .

> docker image list

Context (Current folder)

Correct approach

what's wrong: Jar file Creation > copy jar (Seperate)
Correct : Build everything in docker image! (Copy: not recommended)

```
FROM maven:3.8.6-openjdk-18-slim AS build
WORKDIR /home/app
COPY . /home/app
RUN mvn -f /home/app/pom.xml clean package
FROM openjdk:18.0-slim
VOLUME /tmp
```

→ Build Jar

→ Run

```
EXPOSE 8080
COPY --from=build /home/app/target/*.jar app.jar
ENTRYPOINT ["sh", "-c", "java -jar /app.jar"]
```

why? (If done locally: may be difference while running in other machine)

```
docker build -t in28min/hello:v2 .
```
Create image!

```
docker container run -d -p 5000:5000 in28/hello:v2
```

Note: build doesn't use any local dependencies (except source code)

problem: Long time to build even for smaller changes (caching)

* Copy files that often won't change (pom.xml, MyApp.java), trigger build.
* Caches every layer & reuses it.

```
FROM maven:3.8.6-openjdk-18-slim AS build
WORKDIR /home/app
COPY ./pom.xml /home/app/pom.xml
COPY ./src/main/java/com/example/demodocker/DemoApp.java
     /home/app/src/main/java/com/example/demodocker/DemoApp.java
RUN mvn -f /home/app/pom.xml clean package

COPY . /home/app
RUN mvn -f /home/app/pom.xml clean package

FROM openjdk:18.0-slim
EXPOSE 5000
COPY --from=build /home/app/target/*.jar app.jar
ENTRYPOINT ["sh", "-c", "java -jar /app.jar"]
```

Spring boot maven plugin: Create docker image

* Create an executable jar, Run app, Create Container image
* mvn spring-boot:repackage → war
* mvn spring-boot:repackage -Jar
* mvn spring-boot:run (Run app)
* mvn spring-boot:start (non-blocking, run integration tests)
* mvn spring-boot:stop (stop app started with start)
* mvn spring-boot:build-image (container image)

Cloud

* Varying load (peak: weekend, holidays) : Sudden load (Infrastructure)
* low load: wastage of Infrastructure (guess future, high buying cost, maintain)
* Cloud: Buy (no): Rent resources (Rent only on demand)

Advantages:

* Capital expense (Buy in advance) → Variable expense
* Scale most! (best deals), no need to guess
* Go global in minutes.
* Avoid undifferentiated heavy lifting (no worry: Backup/Standby)
* Focus on Solutions (not on datacenters)

AWS:

* Leader (Azure, Google cloud) : 200+ services (most)
* Reliable, Secure, cost-effective
* Netflix
* Learn AWS: Amazon S3, EC2, Amazon EBS, ELB, ECS (no single path)
  (not independent)

Setup:

* personal details, valid credit/debit card, mobile
* AWS: powerful Infrastructure. (verify who you're) : 1 time process
* AWS management console!

IAM: Identity And Access Management

* Best practices: Control access, Identify users (Authentication, Authorization)
  (Authentication → valid user; Authorization → Has right)
* Root user: can do anything, not recommended for day-day activities
  Create IAM user, use that!

IAM group: (developers: admin access)

* IAM > Create group (AdministratorAccess) > Add users
* Recommended: Add users to a group (Instead of
  Copy permissions from existing user
  Attach existing policies directly)

Regions & Zones:

* High latency, Crash (one instance): Availability low → Add more (different region)
* Solution: AWS: 25+ regions (expanding)
* Advantage: High availability, Low latency, Global footprint, Regulations (Reside with)
* Availability Zones: (Same region): Set of 1/ more discrete DCs
  (Independent, redundant power, network connectivity)

Low latency links

Fault tolerance!

EC2:

* Corporate DC: Deployed to physical servers
* Cloud: Renting virtual machines (EC2 : Elastic Compute Cloud) Instance
* EC2 Service.
* Create, manage lifecycle of EC2 Instance (Start, Stop, Terminate)
* Attach Storage to an EC2 Instance (Amazon EBS: Elastic Block Store)
* Load balance for multiple EC2 Instances (ELB: Elastic Load Balancer)

Create EC2 Instance:

* Launch Instance / Instances > Launch Instance
  1. App & OS Image (Software config + app installed)
  2. Instance type (hardware) : memory, VCPU, pricing
  3. Key pair (Securely Connect) : RSA | ED25519
  4. Network Settings (Security Group: Firewall)
  5. Configure Storage

Connect to an Instance: (Amazon Linux)

* yum update -y
* yum install httpd (webserver)
* start httpd ⟶ Systemctl start httpd (apache web server)

0.0.0.0/0 : Allow from everywhere

EC2 Concepts:

* AMI (Amazon machine Image): OS, Software want to Instance
* Instance Families : Hardware (CPU/Compute/Storage/memory optimized/GPU)
* Instance Size : right quantity of hardware (VCPUs, 4GB of memory)
* Elastic Block Store : Attach Disks (EBS) to EC2 Instance
* Security Group (firewall : Incoming, outgoing traffic)
* Key pair : public & private key (public : Stored in EC2 Instance)
  private : Stored by customer

Run app in cloud

* Approaches : Platform as Service, Infrastructure as Service (IAAS, PAAS)

IAAS:

* Only Infra (VM service : deploy dB, apps)
* Cloud provider : Networking, virtualize, Hardware (Takes care)
* I'm responsible : OS, Software, updates, app code, Runtime, Load balancing, Scaling (no. of Instances), Availability (crash : detect, fix)

PAAS:

* Use platform : Hardware, Networking, virtualization, OS upgrade, patches, Runtime, AutoScaling, Availability, Load balancing
* I'm : Config (app, services), code

Elastic Beanstalk, Azure app Service, Google app engine

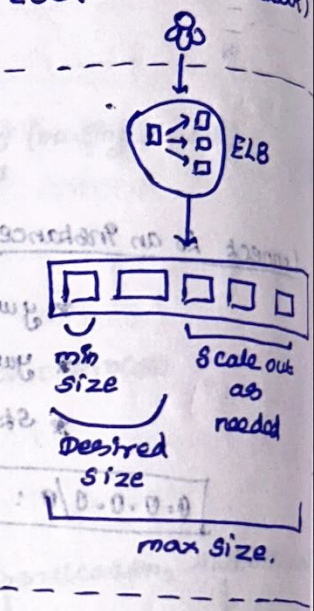
* DB: Relational & NoSql (Amazon RDS, Google Cloud ... DB)
* Queues API, ML, operations.

AWS elastic bean stalk:

* Simple: deploy, scale (AWS)
* End to end web app management (Java, Php, .Net, Node.JS, Go, Docker, Py)
* No usage charge: Pay for AWS resources provisioned.
* Automatic Load balancing, Auto Scaling, manage platform updates (no downtime)

Auto Scaling Group & Elastic load balancing:

* millions of users: same app have deployed on multiple vms
* create & manage: Autoscaling groups (ec2 instances)
* Distribute traffic: ELB (easy way: ELB, ASG) in EBS (elastic bean stalk)



Elastic Bean Stalk:

* Name, platform (Java), upload code
* others:
  - Amazon x-ray (tracing service)
  - Log Streaming, Rotate logs
  - Capacity: envtype (single instance)
    - Instance type (size, VCPU)
    - OS, on demand instance
  - Load balancing config
  - Security, monitoring, platform updates
  - event alerts
  - Database
  - Tags

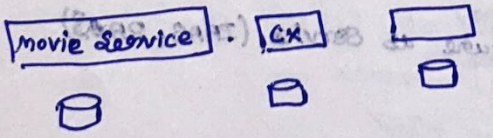
logs, health, monitoring, Alarms, updates, Events

Environments:

* dev, local, IDC/production

---

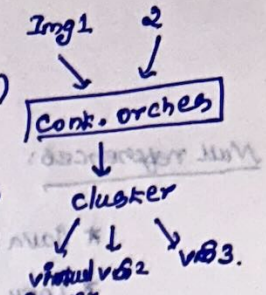
microservices



* Enterprises: heading towards microservices architectures
* Flexible, different languages, Deployment: Complex!
* 1 way monitoring: Containers! ⟶ All dependencies itself bundled.
  - Run on any infrastructure.
  - cloud neutral,
  - Standard: consistent deployment, monitoring, logging.
  - light weight (vms: Guest os impact performance)
  - Docker: good isolation!

Container Orchestration:

* 10 instances of A, 20 of B, 15 of C
* Autoscaling: Scale based on demand
* Service discovery: One ms find others (no hardcoding)
* Load balancer: distribute load among instances
* Self healing: health check, replace failing instances
* Zero downtime deployments: Release no downtime (versions)



Aws Orchestration (EKS, ECS, Fargate → Serverless ECS/EKS

* Kubernates: opensources container Orchestration (AWS, Azure, Google)
* EKS (Elastic Kubernetes Service): No free tier.
* Amazon specific: ECS (Elastic Container Service)
* In ECS/EKS: manage cluster, how many EC2 instance

Amazon ECS: (Fargate)

Cluster (hardware) → Service → Task definition → Container definition

(with multiple node) (multiple instance of microservice) (Run 2/more Containers together)

Serverless:

* Where to deploy? What kind of server? OS? Scaling, load balance, Availa?
* Don't worry about server part: only code (Not means no servers)
* Just servers are not your problem (Automated), Pay for use

eg: Aws Lambda, Lambda Fn

↓

Truly serverless.

No worry: servers, Scaling/availability

Worry: code

Pay: No. of requests (usage), memory, time

Support: Py, Java, Powershell, Ruby, C#, NodeJS....

Code Section > Upload project

Aws Services:

* Amazon EC2 + ELB: Traditional Approach (Virtual servers + Load balancing) Control over OS/Run custom software.
* Aws Elastic beanstalk: Simply manage web app & batch applications Automatically creates EC2, ELB (AutoScale, LB)
* Amazon Elastic Cont. Service (ECS): Simply run microservice with docker Containers (EC2 based ECS clusters)
* EKS: Amazon Elastic Kubernetes Service): Run & Scale Kubernetes clusters
* Aws Fargate: serverless ECS/EKS version
* Aws Lambda: Serverless: No worry about servers

numbers.stream()
- .filter(myclass::isEven) → .filter(number → number%2==0)
- .forEach(System.out::println);

Null references:

* Java.lang.NullPointerException
* Tony Hoare: Introduced Null references in ALGOL
  * popular for developing quick sort algo
  * He calls null reference: billion dollar mistake
* Optional: Java's way of dealing with nulls

```
List<String> fruits = Arrays.asList(_____);
Predicate<? super String> predicate = fruit → fruit.startsWith("b");
fruits.stream().filter(predicate).findFirst();
```

Returns Optional (container: may/may not contains null)

optional.isEmpty();
optional.isPresent(); → hasValue("banana")
optional.get();

Stream().map(number → number * number) → map one value to another