



# Sunbeam Infotech

*Trainer: Nilesh Ghule*



# SQL Injection

// id is input from user (text box) → 2  
2 OR 1=1

String sql = "Select \* from tbl where id = " + id;

Select \* from tbl where id = 2

Select \* from tbl where id = 2 OR 1=1

// all records will be fetched.

## SQL injection attack

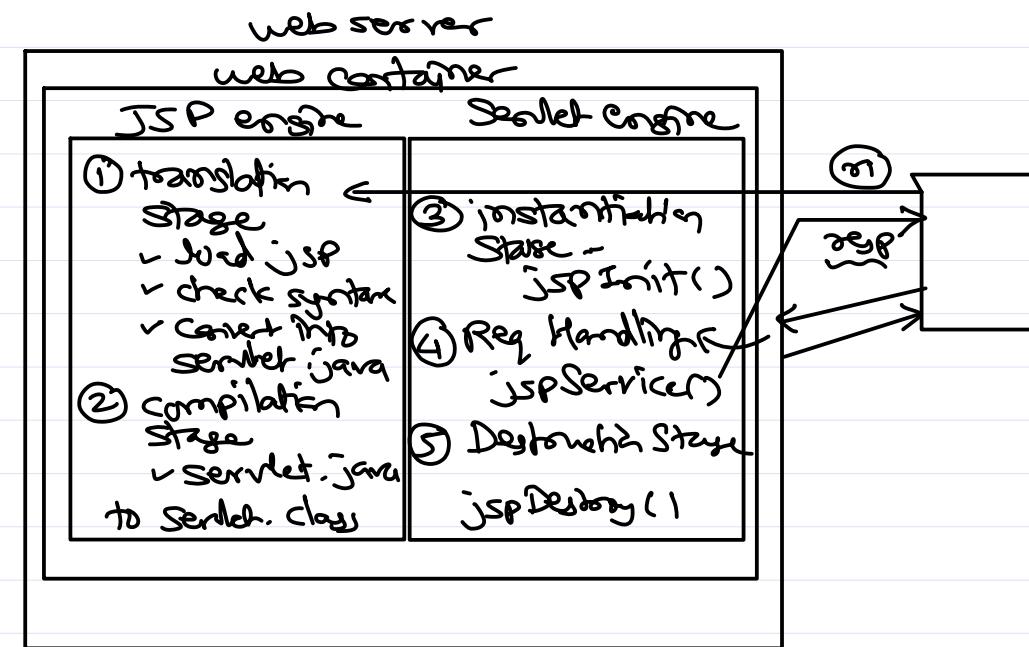
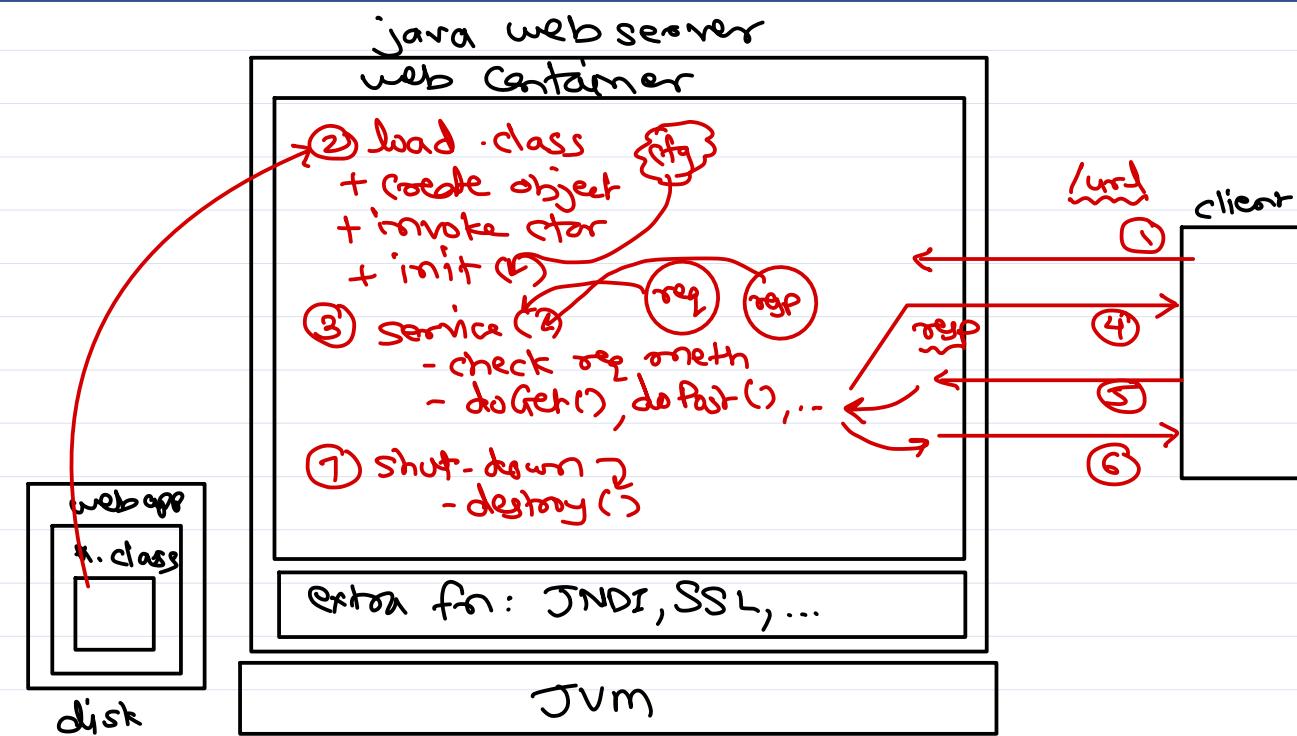
Solution: no String Concat X

String sql = "Select \* from tbl where id = ? "

use PreparedStatement or CallableStatement i.e. stored proc.

JDBC/HQL param queries internally use PreparedStatement.





**web.xml**

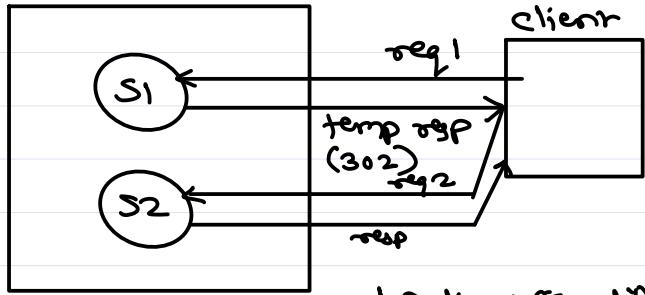
```

① < servlet >
    < servlet-name >
    < servlet-class >
    < load-on-startup > in
< /servlet >
< servlet-mapping >
    < servlet-name >
    < url-pattern >
< /servlet-mapping >

```

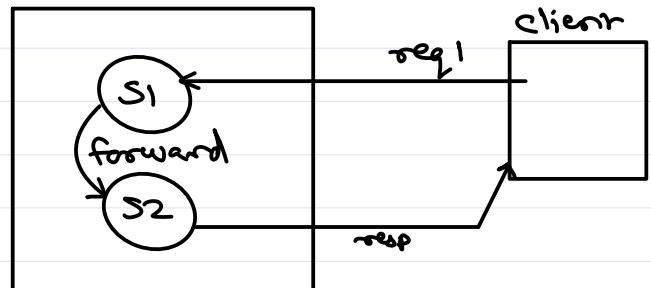
eager loading  
 when appm is deployed  
 servlet class loaded, obj created,  
 ctor called & init () called,  
 servlet obj is ready for next  
 requests.  
 e.g. Spring mvc - DispatcherServlet.

### Server      Http Redirection



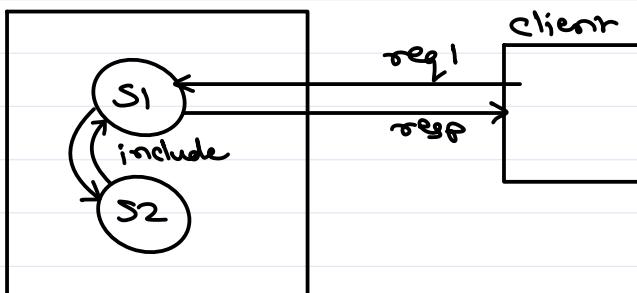
Servlet : `resp.sendRedirect("url");`  
Spring : `view name = "redirect: url";`

### Server



Servlet : `rd.forward(req, resp);`  
Spring : `view name = "forward: url";`

### Server



Servlet : `rd.include(req, resp);`

## State mgmt

↳ client info/state is maintained by the appn.

### client side

- ① cookie
- ② session storage
- ③ hidden fields

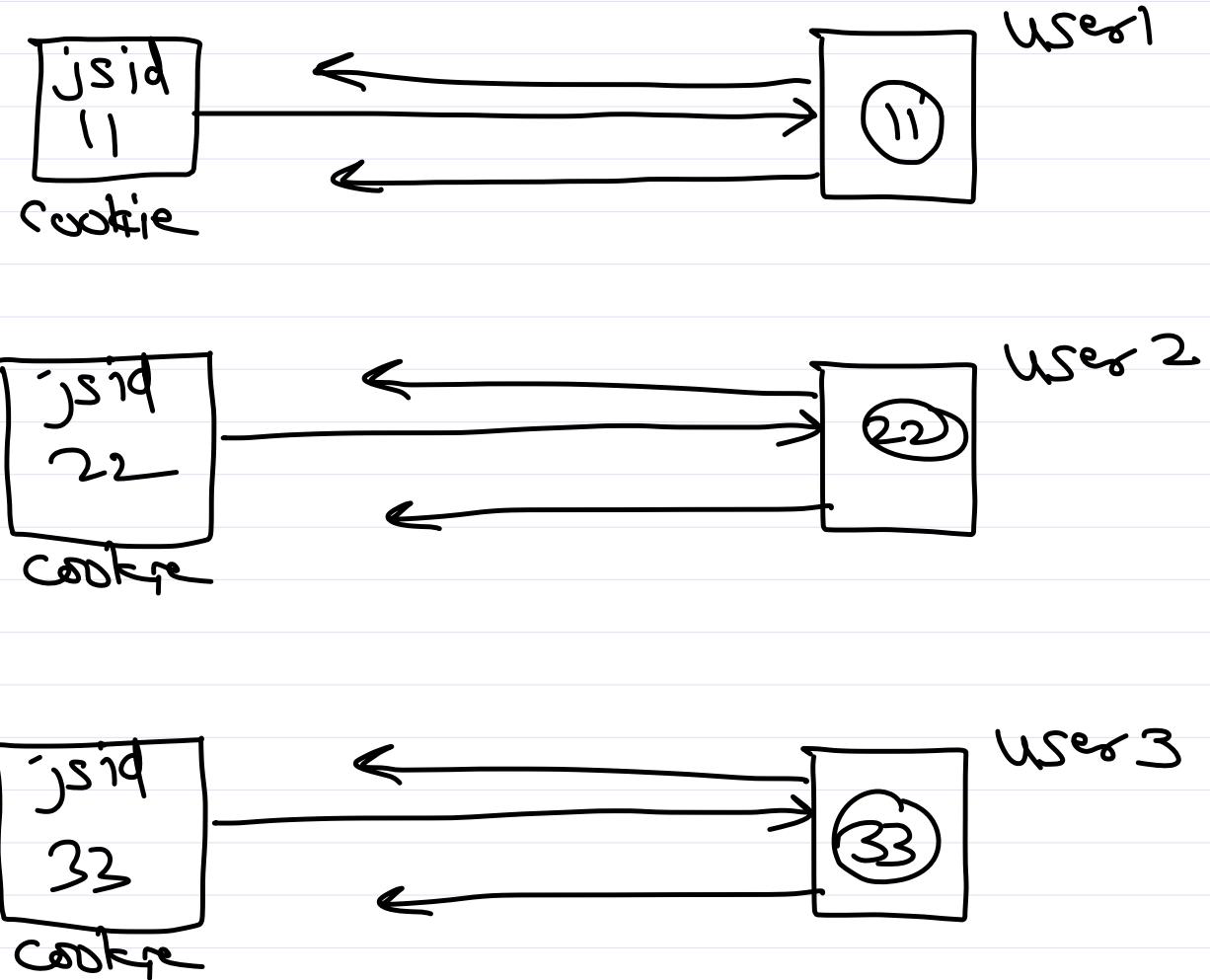
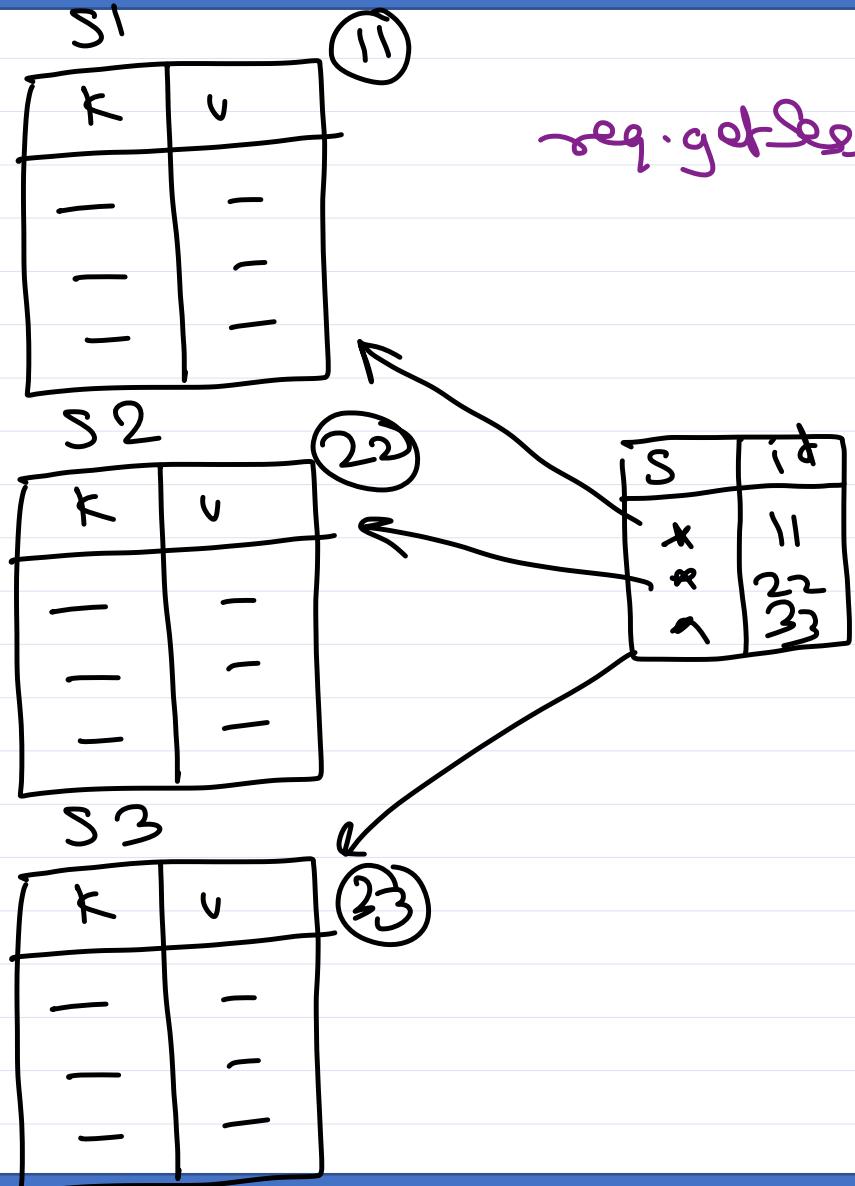
### server side

- ① http session
- ② servlet context
- ③ request attrs

### session tracking

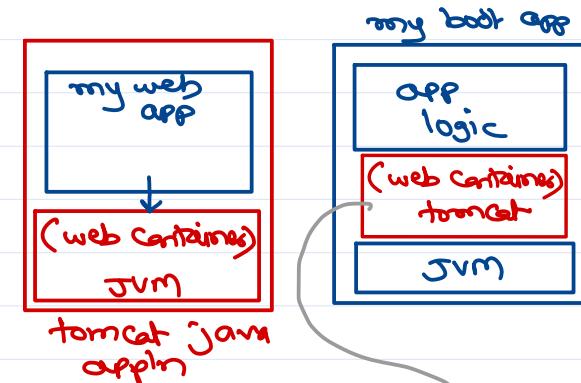
(which session belongs to which client)  
need to associate a session id to client.





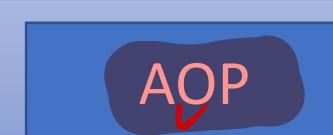
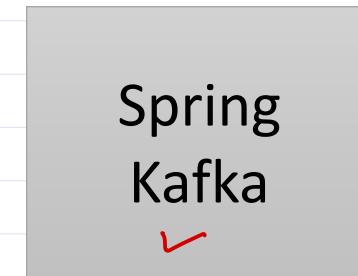
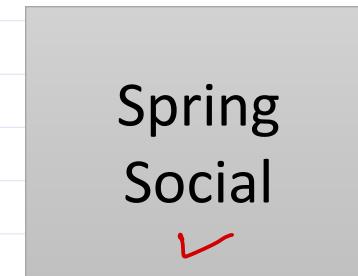
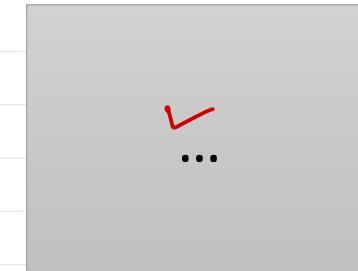
# Spring Boot

- Spring Boot is NOT a new standalone framework. It is combination of various frameworks on spring platform i.e. spring-core, spring-webmvc, spring-data, ...
- Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".  
*most commonly used JARs are auto added into classpath.*
- Spring Boot take an "opinionated view" of the Spring platform and third-party libraries. So most of applications need minimum configuration.
- Primary Goals/Features
  - ✓ Provide a radically faster and widely accessible "Quick Start".
  - ✓ Opinionated config, yet quickly modifiable for different requirements.
  - ✓ Managing versions of dependencies with starter projects.
  - ✓ Provide lot of non-functional common features (e.g. security, servers, health checks, ...).
  - ✓ No extra code generation (provide lot of boilerplate code) and XML config.
  - ✓ Easy deployment and containerisation with embedded Web Server.
- Recommended to use for new apps and not to port legacy Spring apps.



# Spring Boot

## Spring Boot



**Spring Boot = Spring framework + Embedded web-server + Auto-configuration - XML config - Jar conflicts**

# Spring bean life cycle

- Spring container creates (singleton) spring bean objects when container is started.
- Spring container controls creation and destruction of bean objects.
- There are four options to control bean life cycle.
  - InitializingBean and DisposableBean callback interfaces
  - Aware interfaces for specific behaviours
    - BeanNameAware, BeanFactoryAware, ApplicationContextAware , etc.
  - BeanPostProcessor callback interface
  - Custom init() / @PostConstruct and destroy() / @PreDestroy methods



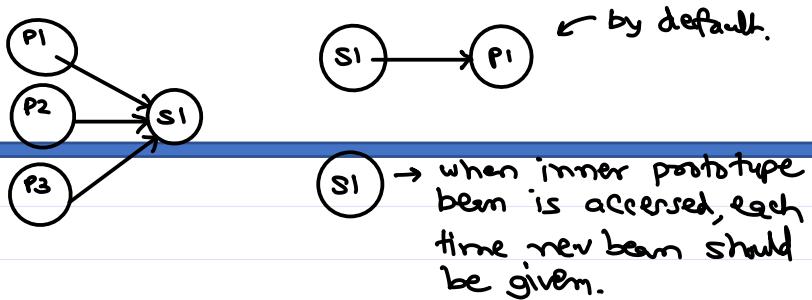
# Spring bean life cycle

- Bean creation
  - ✓ 1. Instantiation (Constructor)
  - ✓ 2. Set properties (Field/setter based DI)
  - ✓ 3. BeanNameAware.setBeanName()
  - ✓ 4. ApplicationContextAware.setApplicationContext()
  - ✓ 5. Custom BeanPostProcessor.postProcessBeforeInitialization()
  - ✓ 6. Custom init() (@PostConstructor)
  - ✓ 7. InitializingBean.afterPropertiesSet() *→ @Bean (init = "method")*
  - ✓ 8. Custom BeanPostProcessor.postProcessAfterInitialization()
- Bean destruction
  - ✓ 1. Custom destroy() (@PreDestroy)
  - ✓ 2. DisposableBean.destroy()
  - ✓ 3. Object.finalize()



# Bean scopes

- Singleton bean inside prototype bean
  - Single singleton bean object is created.
  - Each call to getBean() create new prototype bean. But same singleton bean is autowired with them.
- Prototype bean inside singleton bean
  - Single singleton bean object is created.
  - While auto-wiring singleton bean, prototype bean is created and is injected in singleton bean.
  - Since there is single singleton bean, there is a single prototype bean.
- Need multiple prototype beans from singleton bean (solution 1)
  - Using ApplicationContextAware
  - The singleton bean class can be inherited from ApplicationContextAware interface.
  - When its object is created, container call its setApplicationContext() method and give current ApplicationContext object. This object can be used to create new prototype bean each time (as per requirement).
- Need multiple prototype beans from singleton bean (solution 2)
  - using @Lookup method
  - The singleton bean class contains method returning prototype bean.
  - If method is annotated with @Lookup, each call to the method will internally call ctx.getBean(). Hence for prototype beans, it returns new bean each time.



S1 → when inner prototype bean is accessed, each time new bean should be given.

```
Inner getInner() {  
    return ctx.getBean(Inner.class);  
}  
@Lookup  
Inner getInner() {  
    return null;  
}
```

```
@Scope("singleton")
@Component
class Outer {
    @Autowired
    Inner inner;
```



```
@Scope("prototype")
@Component
class Inner {
```

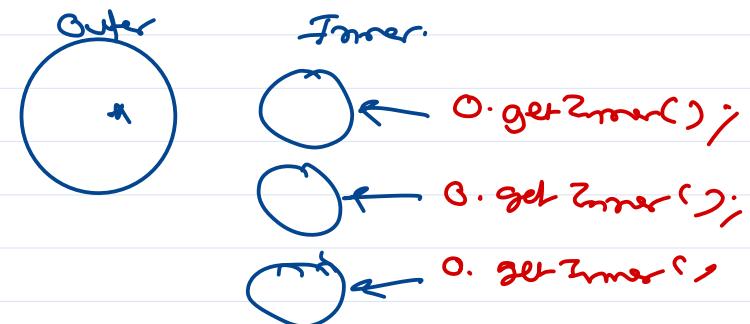
```
@Scope("singleton")
@Component
class Outer {
```

```
    @Lookup
    Inner getInner() {
        return null;
    }
}
```

3

```
@Scope("prototype")
@Component
class Inner {
```

3





```
Car c=new Car();
c.setEngine(e);
c.setChassis(ch);
c.setWheels(w);
;
c.drive();
```

## Dependency Injection

- ① setter based di
- ② constructor based di
- ③ field based di

## Autowiring

Special DI, auto selecting  
bean object to be injected.

Spring Configs  
XML or annotations config

```
Car c = ctr.getBean("c");
c.drive();
```

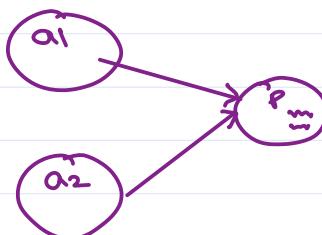
Autowiring types → XML config  
= byType, byName, constructor, none

## @Autowired

→ field level  
→ setter method level  
→ constructor level → preferred by Spring boot.

class Account {
 @Value("101")
 int id;
 @Value("\${bal3}")
 double balance;
 @Autowired
 Person accHolder;
}

under @Configuration  
↓ class  
@Bean
Person p1() {
 return new
 Person(..);
}



# Prototype bean in Singleton bean

```
@Scope("singleton")
```

```
@Component
```

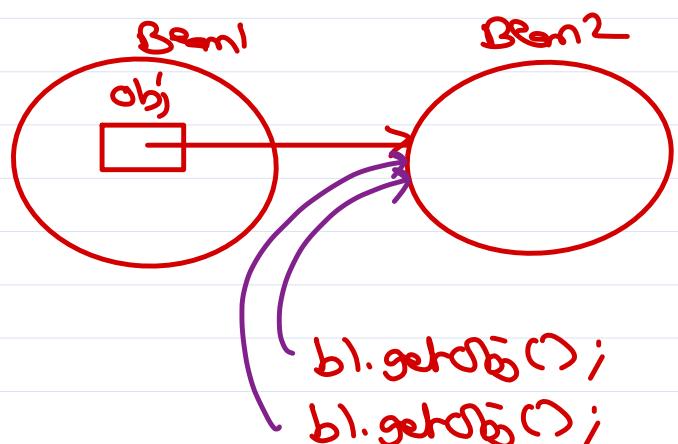
```
class Bean1 {
```

```
    @Autowired  
    Bean2 obj;
```

```
    getObj();
```

```
        sehr obj;
```

```
    }  
}
```



```
@Scope("prototype")
```

```
@Component
```

```
class Bean2 {
```

```
@Scope("singleton")
```

```
@Component
```

```
class Bean1 {
```

```
@Lookup
```

```
Bean2 getObj();
```

```
        sehr null;
```

```
    }  
}
```

```
@Scope("prototype")
```

```
@Component
```

```
class Bean2 {
```

```
@Scope("singleton")
```

```
@Component
```

```
class Bean1 {
```

```
    @Autowired
```

```
    AppCtx ctx;
```

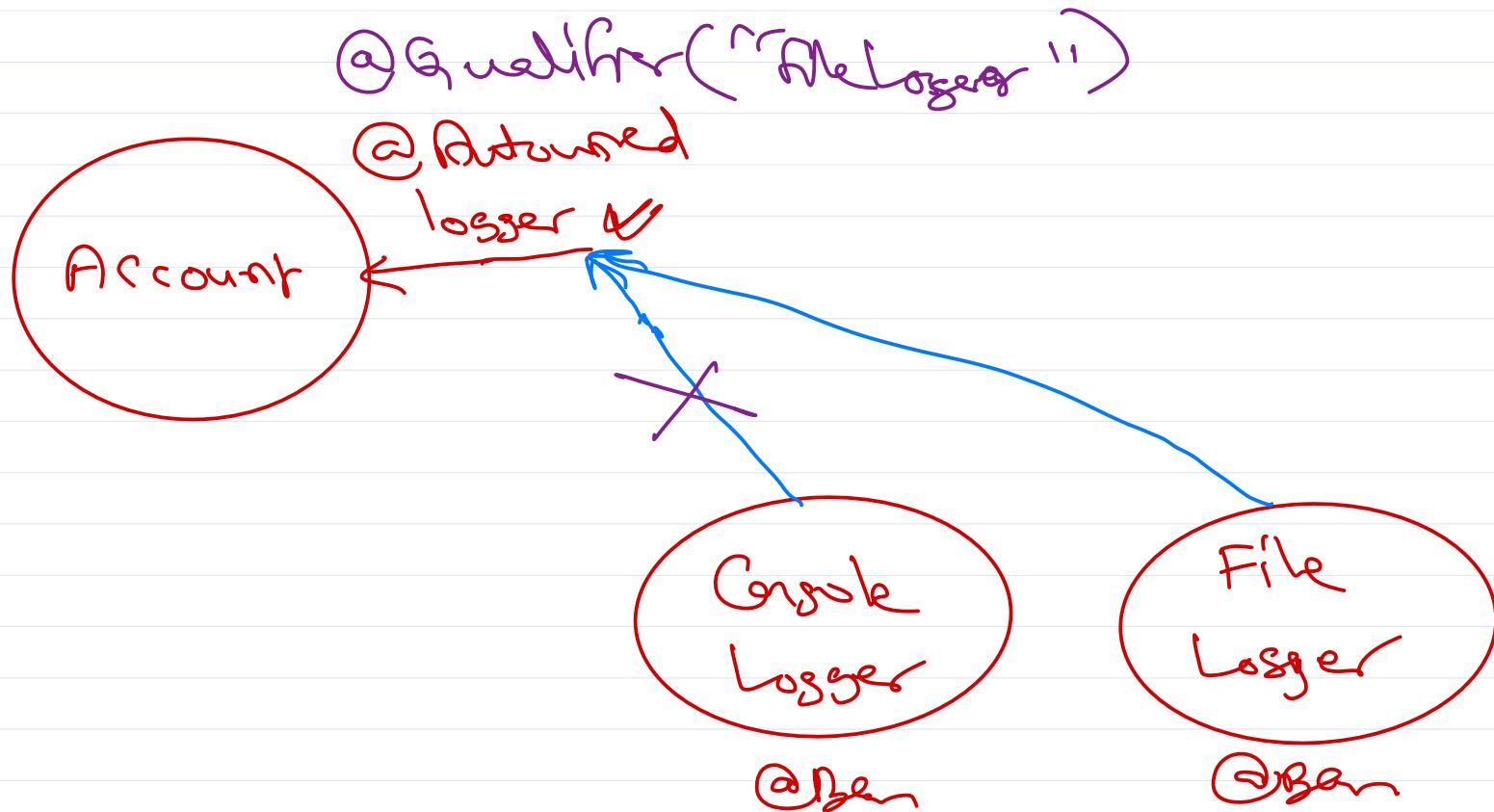
```
    Bean2 getObj();
```

```
        sehr ctx.getBean  
        (Bean2.class);
```

```
    }  
}
```

bl.getObj() → ctx.getBean()  
bl.getObj() → ctx.getBean()

will call ctx.getBean()  
internally for each  
call to getObj();



## Spring Bean Life Cycle

```
class MyBean  
    implements BeanNameAware,  
    ApplicationContextAware,  
    InitializingBean,  
    DisposableBean  
{  
    @PostConstruct  
    void init() {  
        ...  
    }  
    @PreDestroy  
    void destroy() {  
        ...  
        setApplicationContext(ctx); ...  
        setBeanName(name); ...  
        beforePropertiesSet(); ...  
        afterPropertiesSet(); ...  
        ...  
    }  
}
```

- ① bean obj (create from) - constructor
- ② init → setters are called
- ③ setBeanName()
- ④ set Application Context()
- ⑤ before Properties Set()
- ⑥ @PostConstruct
- ⑦ after Properties Set()
- ⑧ Bean ready.



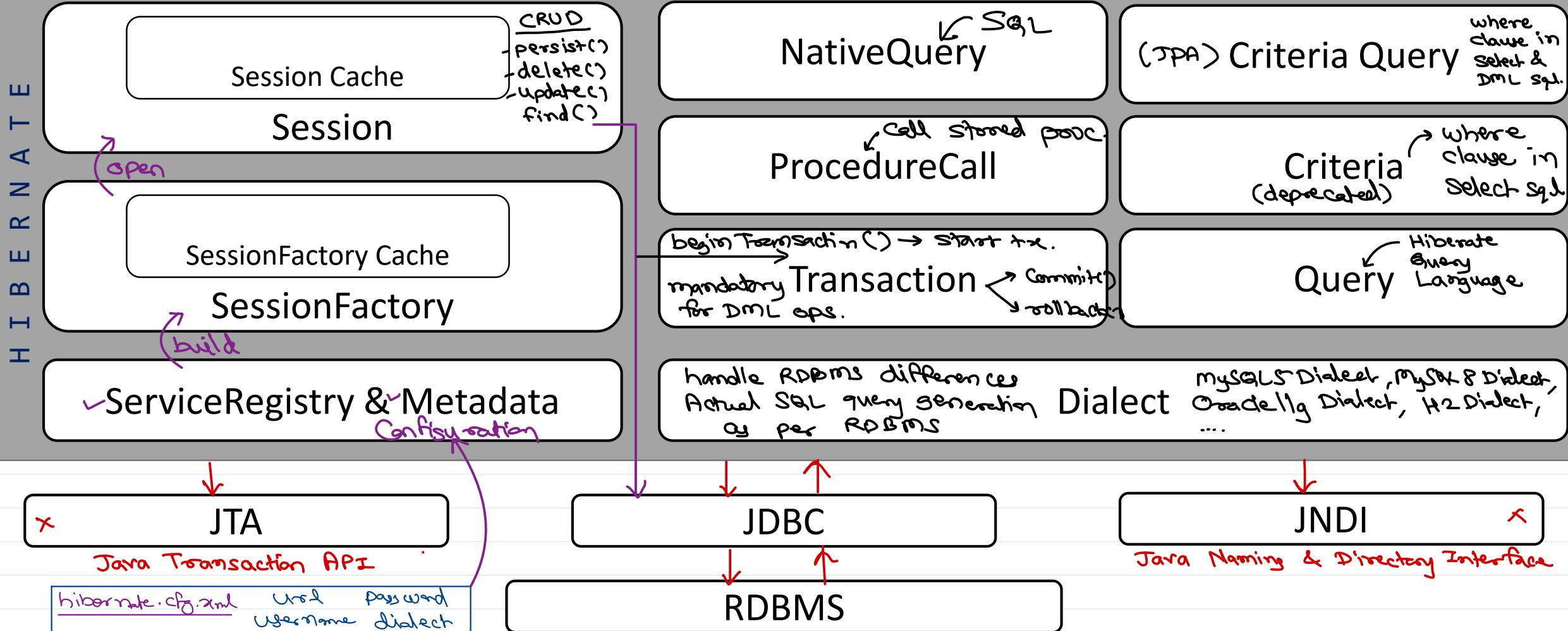
# Hibernate Architecture

CRUD ops  
Hibernate Associations  
HQL & Auto-generated PK.



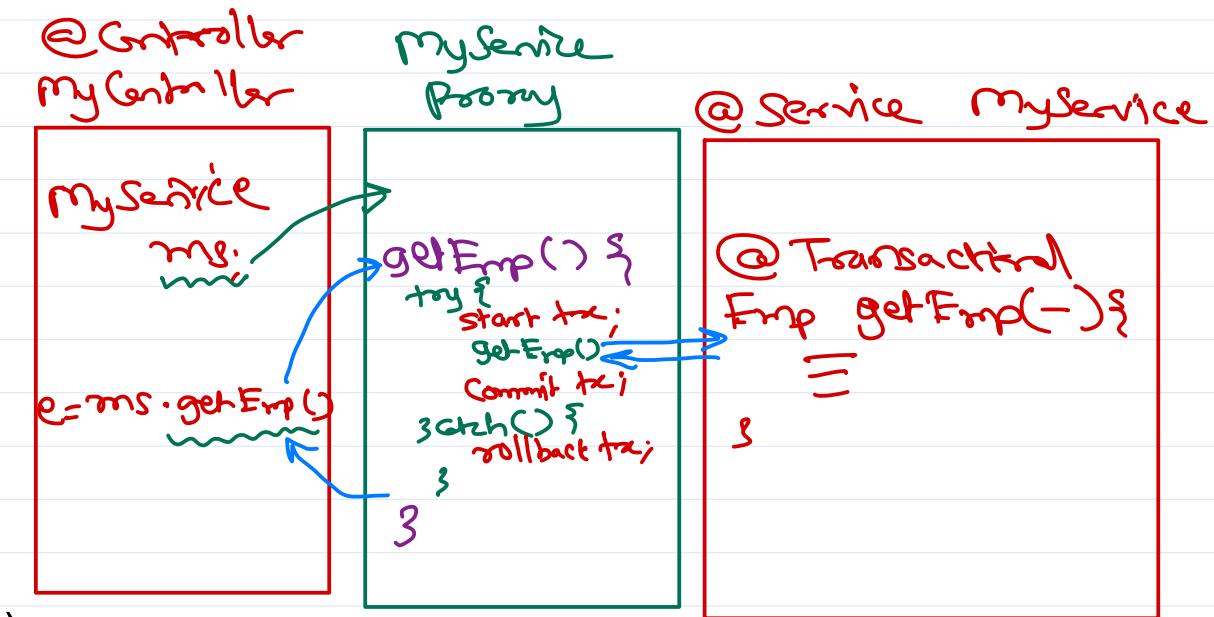
## Java Application

H I B E R N A T E



# @Transactional

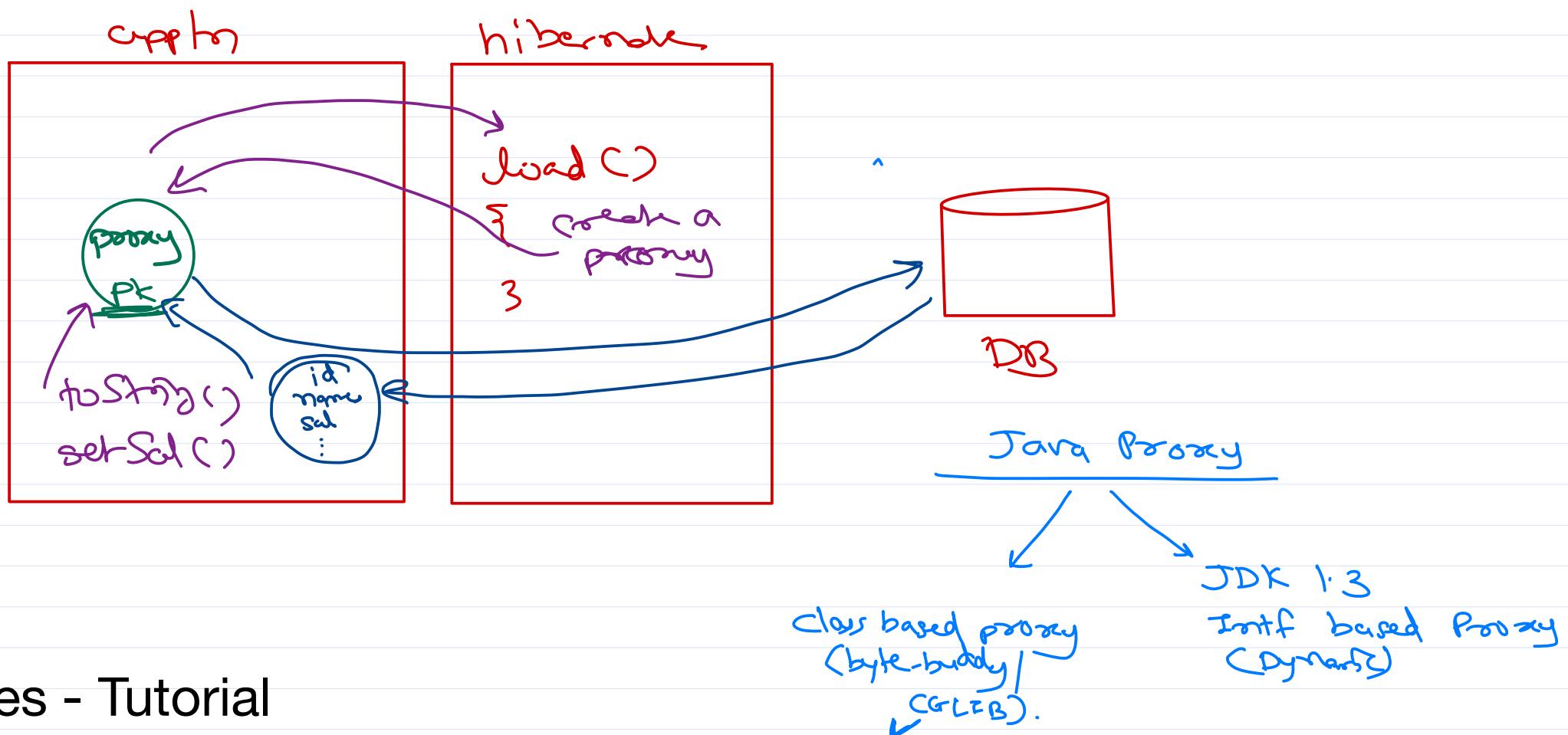
- @Transactional is declarative transaction management of Spring.
- It can be used method level or class level. If used on class level, it applies to all methods in the class.
- Spring internally use JDBC transaction in AOP fashion.
  - start transaction (before method).
  - commit transaction (if method is successful).
  - rollback transaction (if method throw exception).
- Transaction management is done by platform transaction manager e.g. datasource, hibernate or jpa transaction manager.



# CRUD operations

- Hibernate Session methods
  - `get()` or `find()`: Find the database record by primary key and return it. If record is not found, returns null.
  - `load()`: Returns proxy for entity object (storing only primary key). When fields are accessed on proxy, SELECT query is fired on database and record data is fetched. If record not found, exception is thrown.
  - `save()`: Assign primary key to the entity and execute INSERT statement to insert it into database. Return primary key of new record.
  - `persist()`: Add entity object into hibernate session. Execute INSERT statement to insert it into database (for all insertable columns) while committing the transaction.
  - `update()`: Add entity object into hibernate session. Execute UPDATE statement to update it into database while committing the transaction. All (updateable) fields are updated into database (for primary key of entity).
  - `saveOrUpdate()` or `merge()`: Execute SELECT query to check if record is present in database. If found, execute UPDATE query to update record; otherwise execute INSERT query to insert new record.
  - `delete()` or `remove()`: Delete entity from database (for primary key of entity) while committing the transaction.
  - `evict()` or `detach()`: Removes entity from hibernate session. Any changes done into the session after this, will not be automatically updated into the database.
  - `clear()`: Remove all entity objects from hibernate session.
  - `refresh()`: Execute SELECT query to re-fetch latest record data from the database.



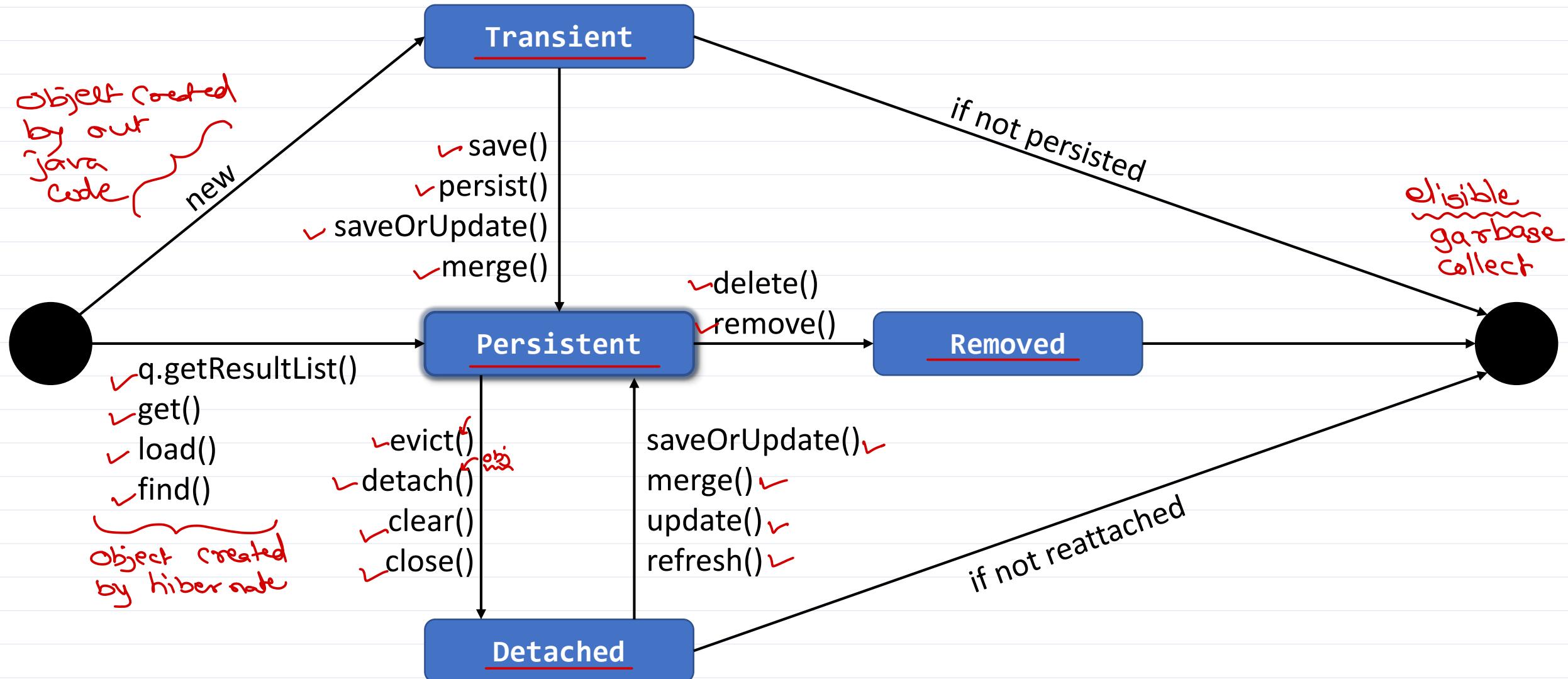


## Java Proxies - Tutorial

[https://youtu.be/4X\\_sZNOeR7g](https://youtu.be/4X_sZNOeR7g)



# Hibernate Entity Lifecycle



# Hibernate – Entity life cycle

- Transient **(New)**

- New Java object of entity class.
- This object is not yet associated with hibernate.

- Persistent **(Managed)**

- Object in session cache.
- For all objects created by hibernate or associated with hibernate.
- State is tracked by hibernate and updated in database during commit.

- Never garbage collected.

- Detached **(Detached)**

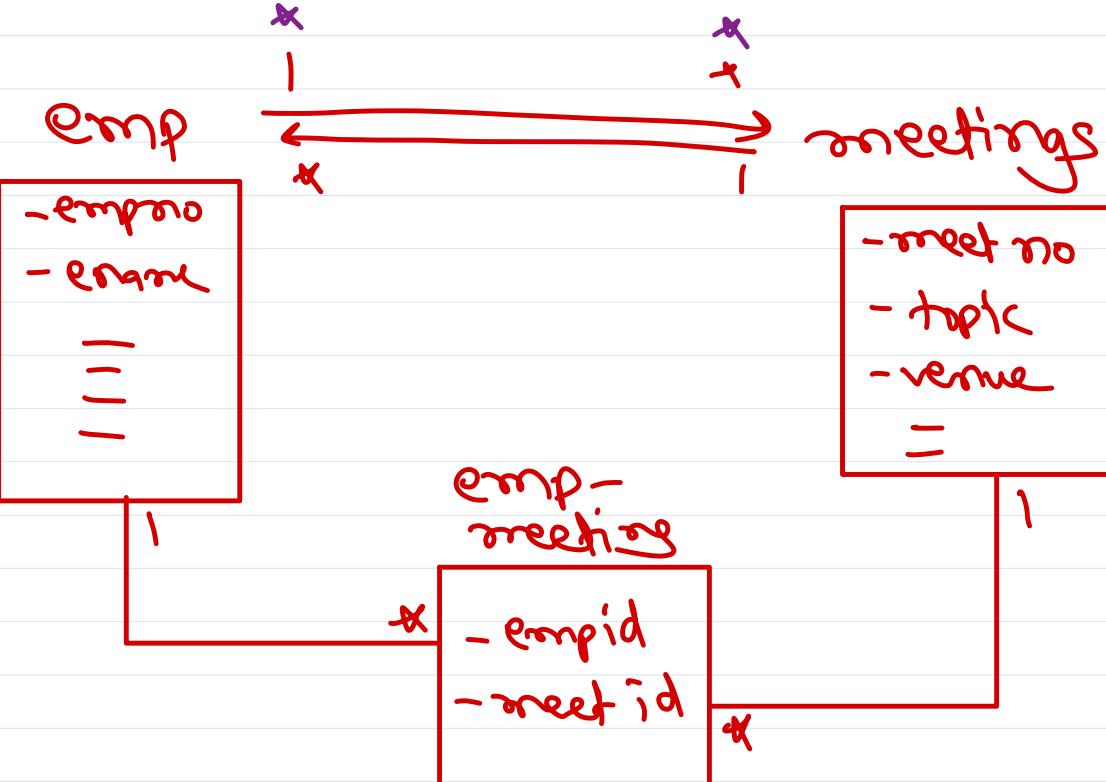
- Object removed from session cache.

- Removed **(Removed)**

- Object whose corresponding row is deleted from database.



# ManyToMany



```
@Entity  
class Emp {  
    =  
    @ManyToMany  
    @JoinTable(name = "emp_meeting",  
    joinColumns = {@JoinColumn("empid")},  
    inverseJoinColumns = {@JoinColumn("meet_id")})  
    List<Meeting> meetlist;  
}
```

```
@Entity  
class Meeting {  
    =  
    @ManyToMany (mappedBy = "meetlist")  
    List<Emp> empList;  
}
```

# Stored Procedure

JDBC → CallableStatement

Hibernate → ProcedureCall

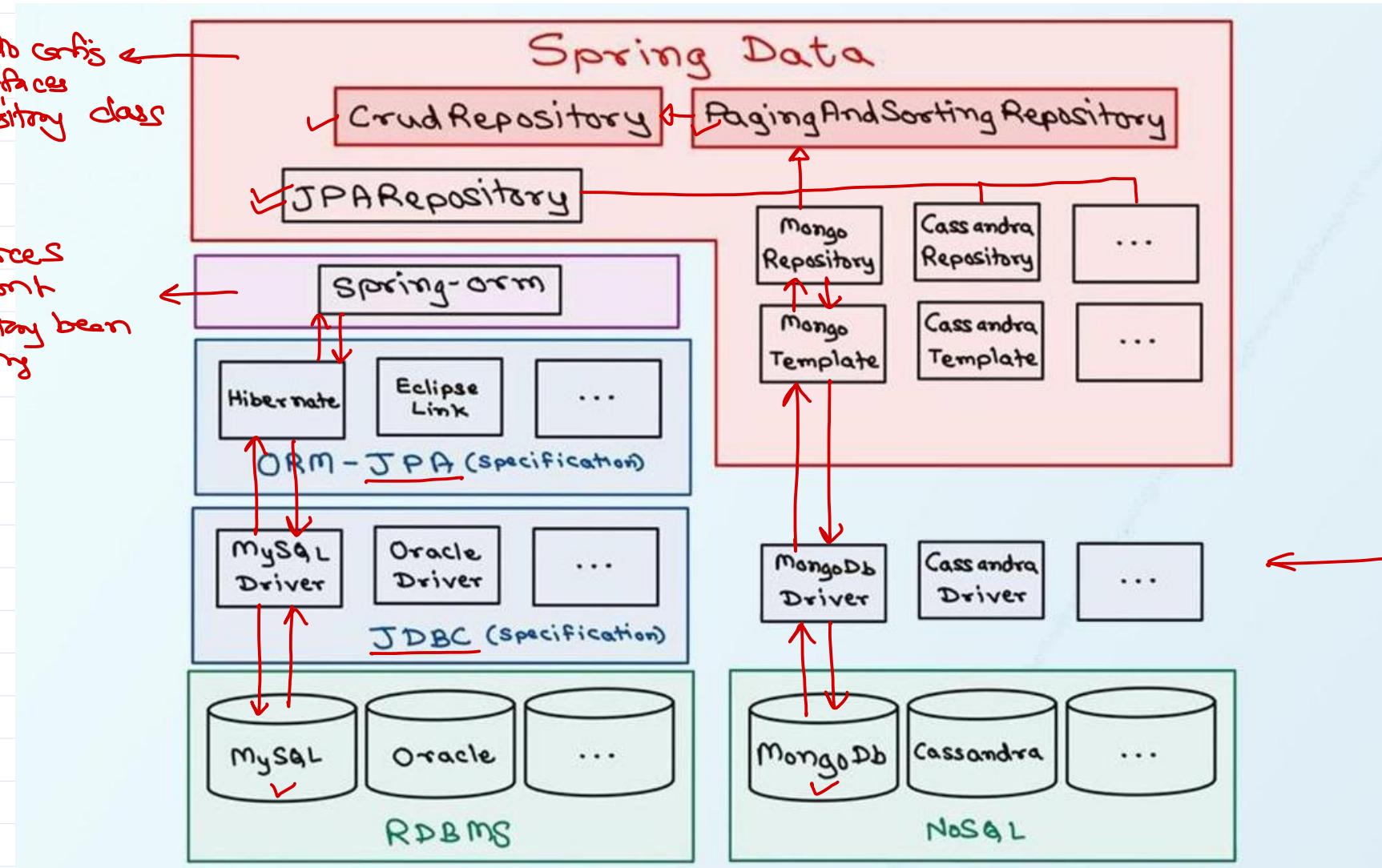
Spring Data → @Procedure



# Spring Data

Spring boot auto config  
repository interfaces  
simple JPA repository class

data sources  
tx manager  
sessionfactory bean  
auto wiring



Dept  
  |  
  |

@OneToMany (cascade = { - 5 })

emplist  
  |  
  |

① PERSIST → em.persist(d);

② MERGE → em.merge(d);

③ DETACH → em.detach(d);

④ REMOVE → em.remove(d);

⑤ REFRESH → em.refresh(d);

internally

for (Emp e: d.emplist)  
em.persist(e);

for (Emp e: d.emplist)  
em.merge(e);

for (Emp e: d.emplist)  
em.detach(e);

for (Emp e: d.emplist)  
em.remove(e);

for (Emp e: d.emplist)  
em.refresh(e);



## Ubuntu - EC2 instance

- ① Create instance
- ② Configure network: inbound/outbound port  
  8080
- ③ terminal> sudo apt install openjdk-11-jdk
- ④ create/copy appn jar to the machine.

terminal> nohup java -jar app.jar &

↳ Linux cmd: no hangup (signal)

do not terminate child appns  
even if terminal is closed.

## ④ install my sql

terminal> sudo apt install mysql-server

- ⑤ modify appn properties file to give path/cod of this db.



# Spring Boot – Application Bootstrapping

`@SpringBootApplication = @ComponentScan + @Configuration + @EnableAutoConfiguration`

## `@ComponentScan`

- Auto detect spring bean classes
- Default basePackage is current

## `@Configuration`

- `@Bean` methods to create beans
- Other config: property source, ...

## `@EnableAutoConfiguration`

- Intelligent & automatic config depending on classes in classpath and beans created in application
- Locates `@Configuration` classes
- Auto-config beans internally use *Spring* `@Conditional` config
  - `@ConditionalOnClass`
  - `@ConditionalOnMissingBean`

```
@SpringBootApplication
public class MyApp {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}
```

## `main()`

- Entry-point of Spring boot app
- Called by JVM.

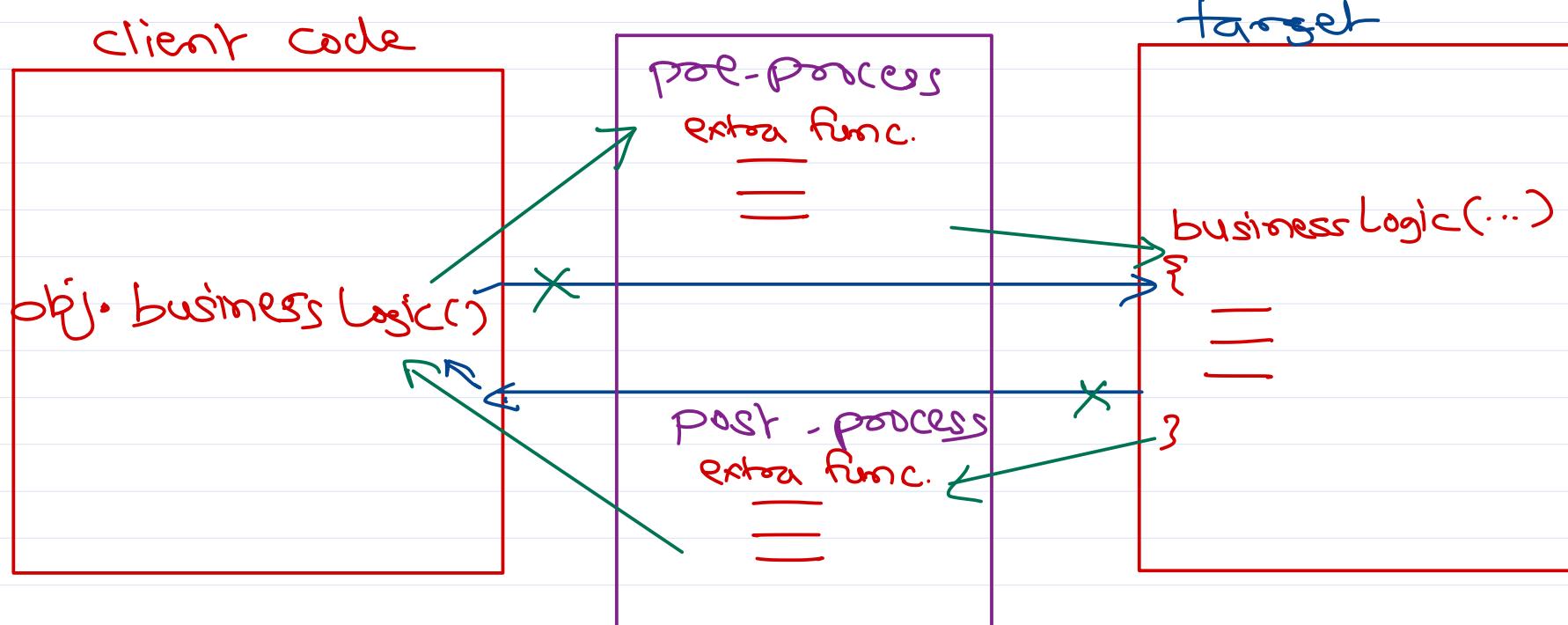
## `SpringApplication.run()`

- Creates *Spring* ApplicationContext (as per auto-configuration)
- Expose command-line args as *Spring* properties
- Prepare spring container for bean life-cycle management
- Load all *Spring* singleton beans
- Runs CommandLineRunner (if any)

***SpringApplication* class used to bootstrap and launch *Spring* application from `main()`**



# AOP



AOP terminologies:

- ① Aspect
- ② Advice
- ③ Target
- ④ Proxy
- ⑤ Point-cut
- ⑥ Join Point

## AOP in Core Java

- ① Dynamic proxies (interface based)
- ② Class based proxies (byte code weaving)  
e.g. CGLIB

## AOP in Java EE

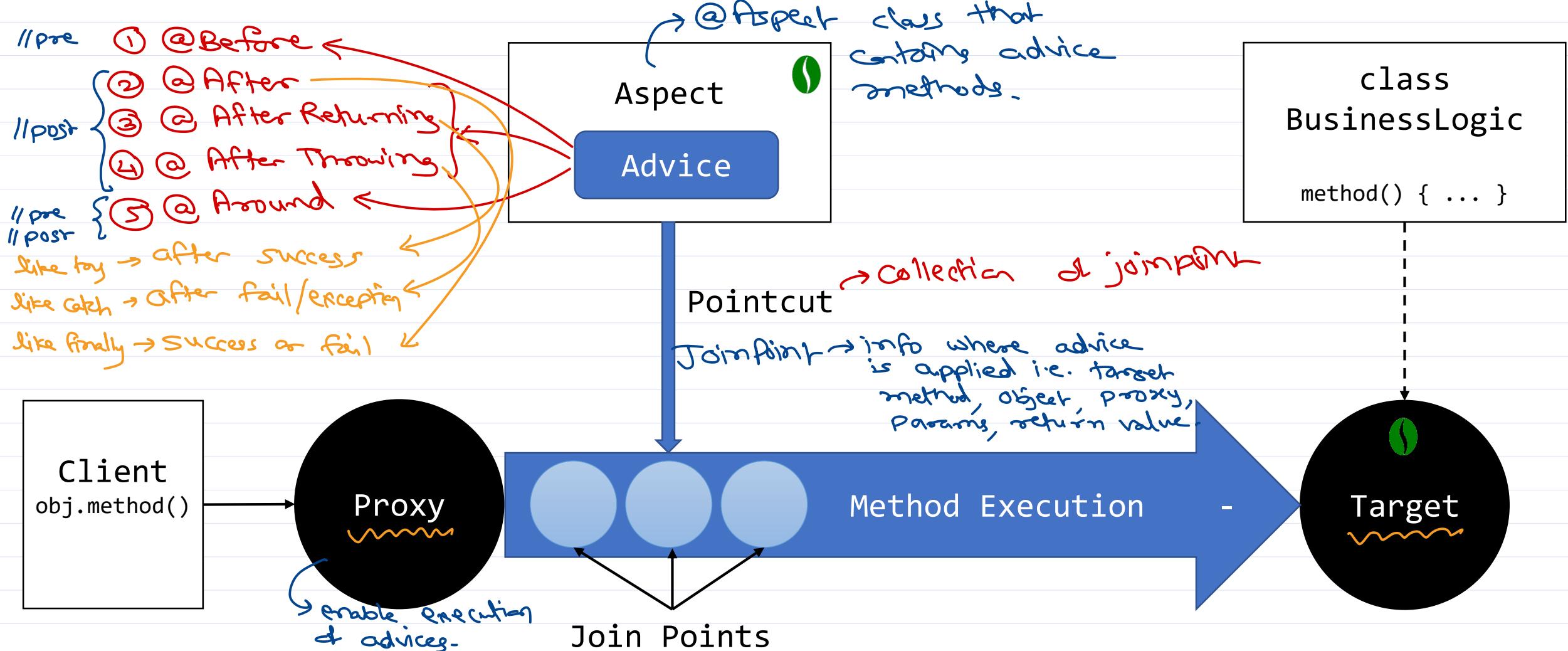
- ① Java EE filters

## AOP in Spring

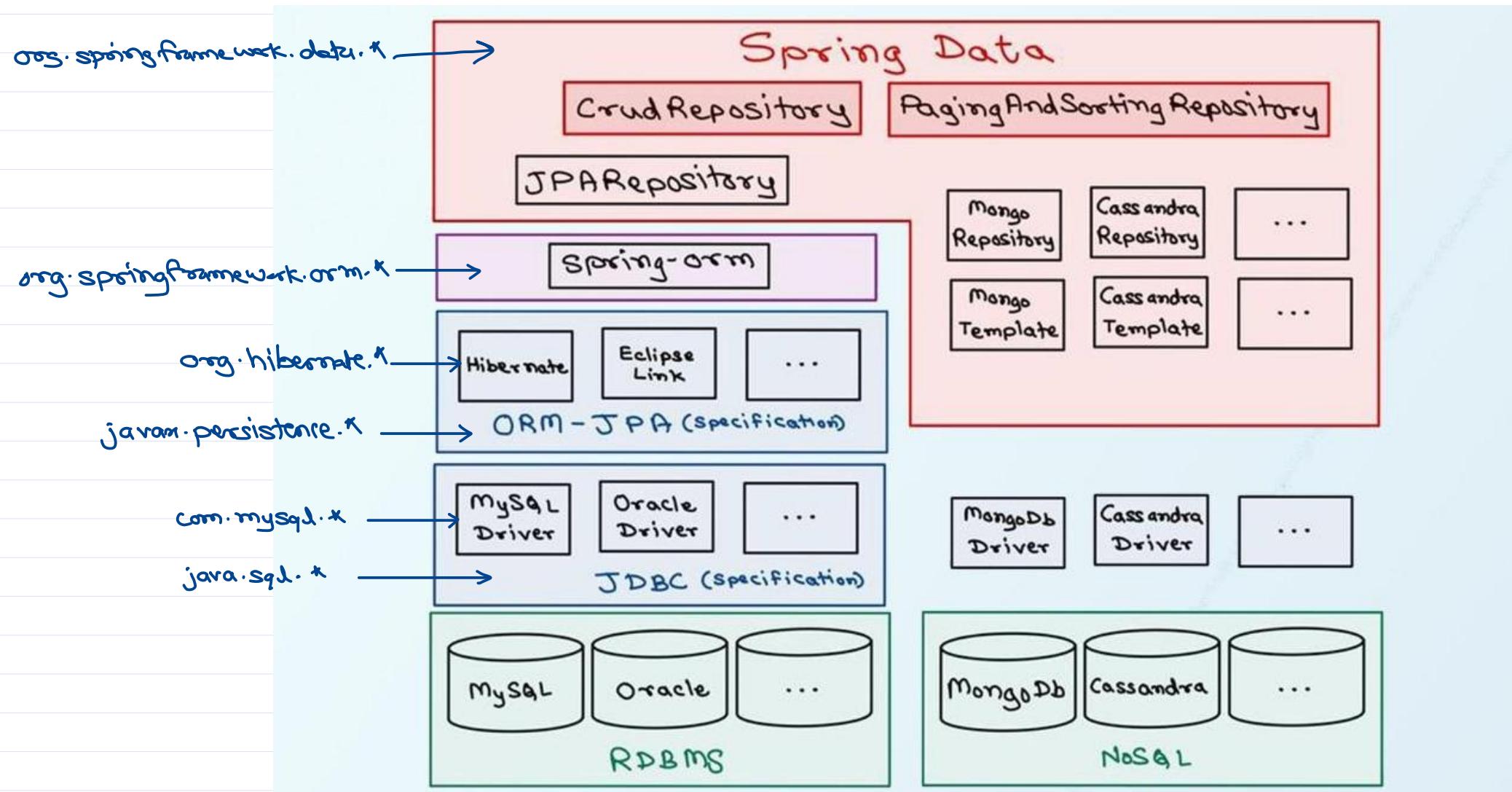
- ① Spring-AOP (AspectJ)
- ② Interceptors



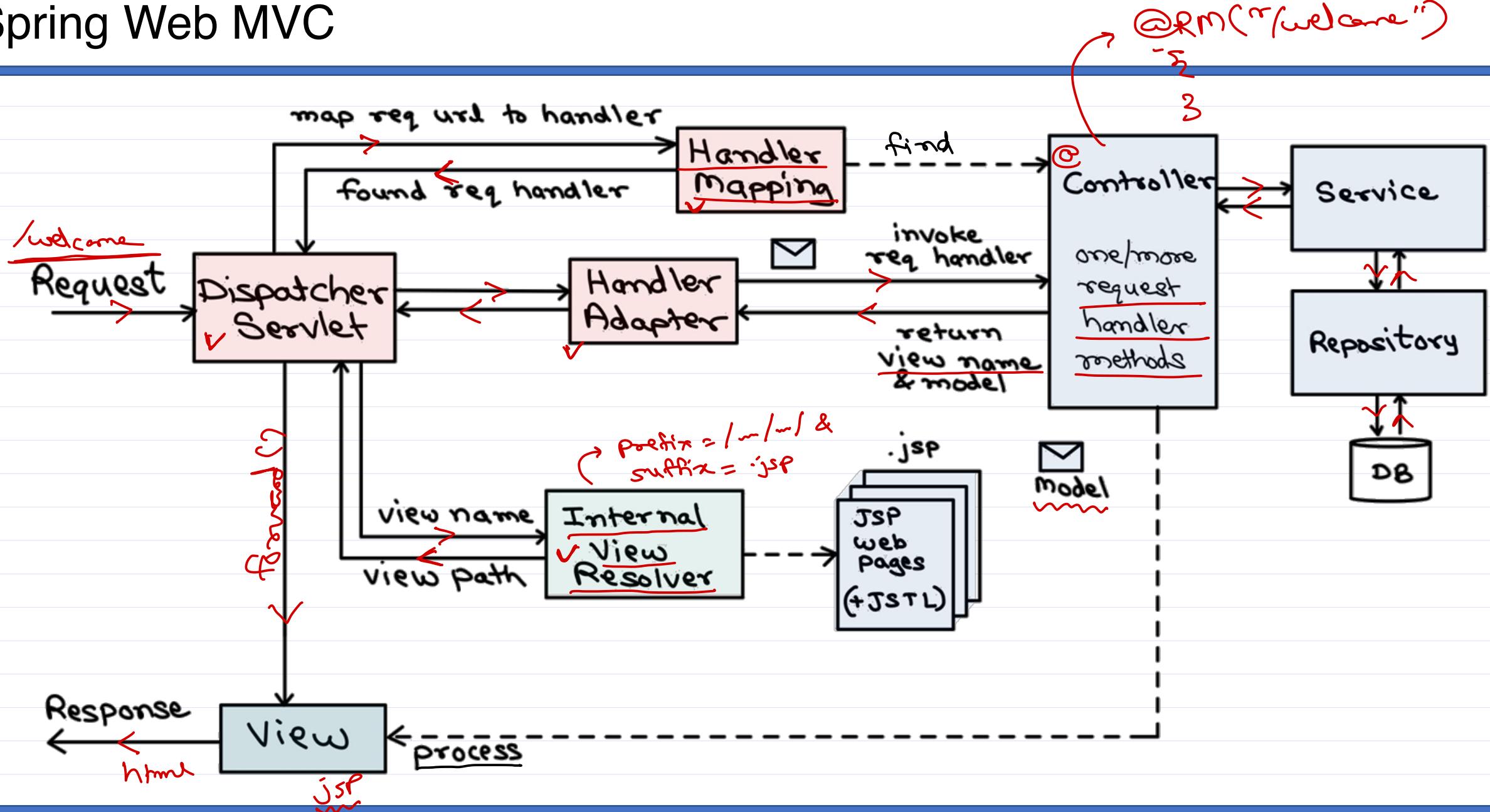
# Spring AOP



# Spring Data



# Spring Web MVC



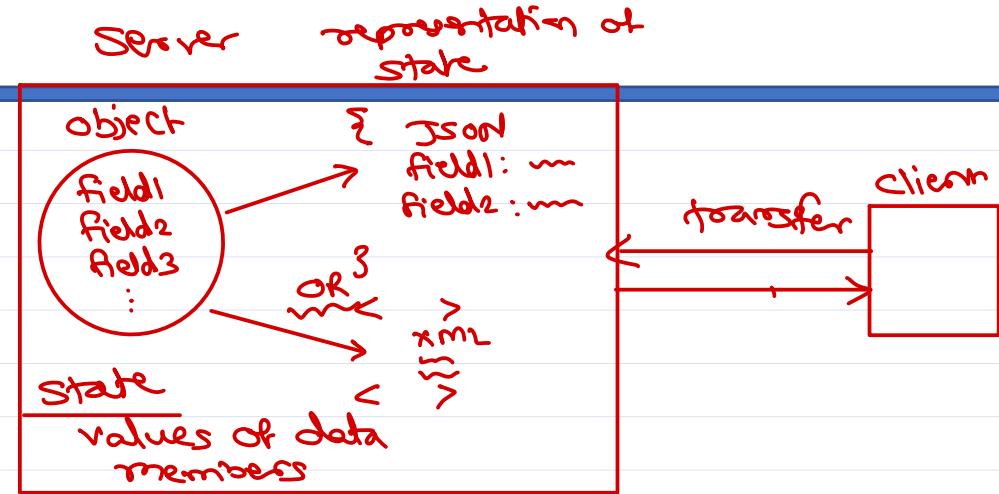
# Spring Web MVC

- DispatcherServlet receives the request.
  - DispatcherServlet dispatches the task of selecting an appropriate controller to HandlerMapping. HandlerMapping selects the controller which is mapped to the incoming request URL and returns the (selected Handler) and Controller to DispatcherServlet.
  - DispatcherServlet dispatches the task of executing of business logic of Controller to HandlerAdapter.
  - HandlerAdapter calls the business logic process of Controller.
  - Controller executes the business logic, sets the processing result in Model and returns the logical name of view to HandlerAdapter.
  - DispatcherServlet dispatches the task of resolving the View corresponding to the View name to ViewResolver. ViewResolver returns the View mapped to View name.
  - DispatcherServlet dispatches the rendering process to returned View.
  - View renders Model data and returns the response.
- 
- <https://terasolunaorg.github.io/guideline/1.0.x/en/Overview/SpringMVCOversview.html>

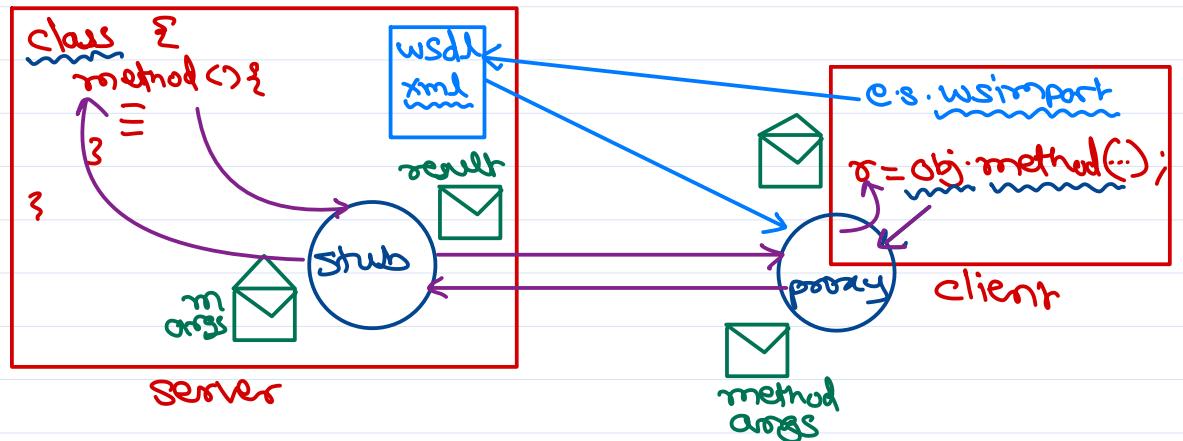


# REST services

- REpresentation State Transfer *without ui*
- Protocol to invoke web services from any client.
  - Client can use any platform/language.
- REST works on top of HTTP protocol.
  - Can be accessed from any device which has internet connection.
  - REST is lightweight (than SOAP) – XML or JSON.
  - Uses HTTP protocol request methods
    - GET: to get records
    - POST: to create new record
    - PUT: to update existing record
    - DELETE: to delete record



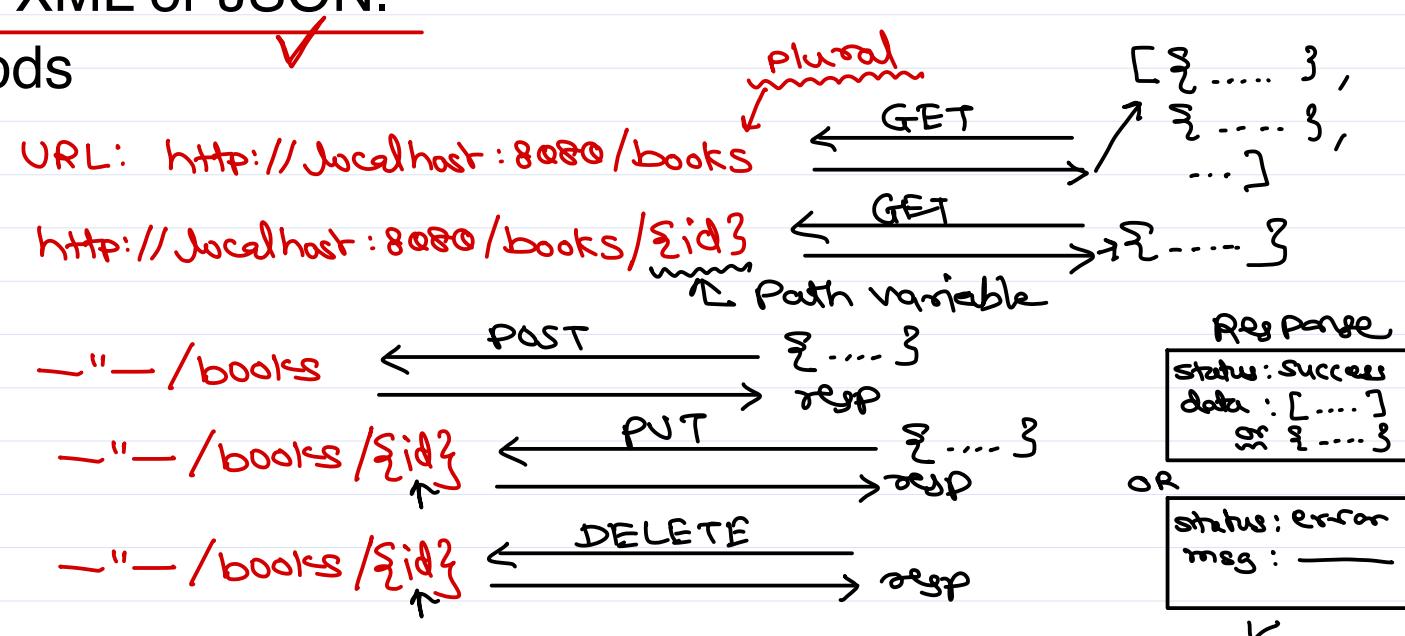
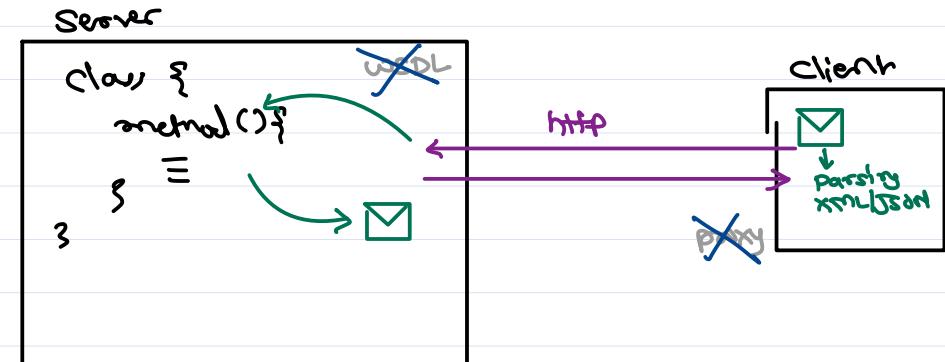
SOAP = HTTP + XML with format

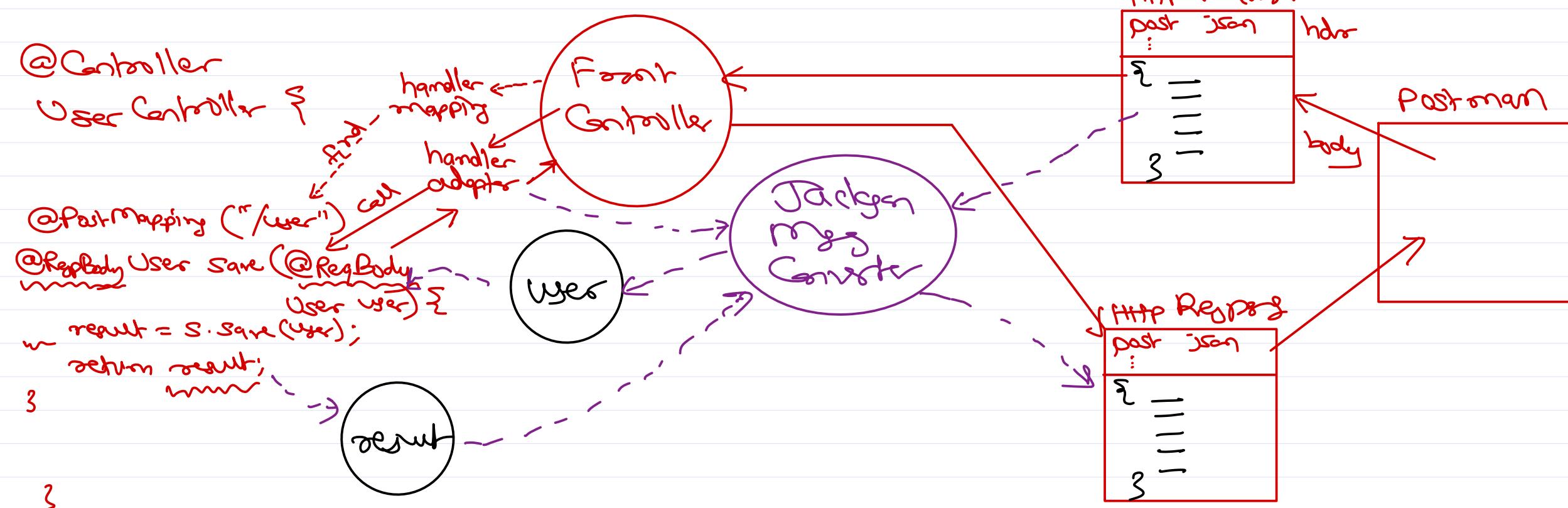


# REST services

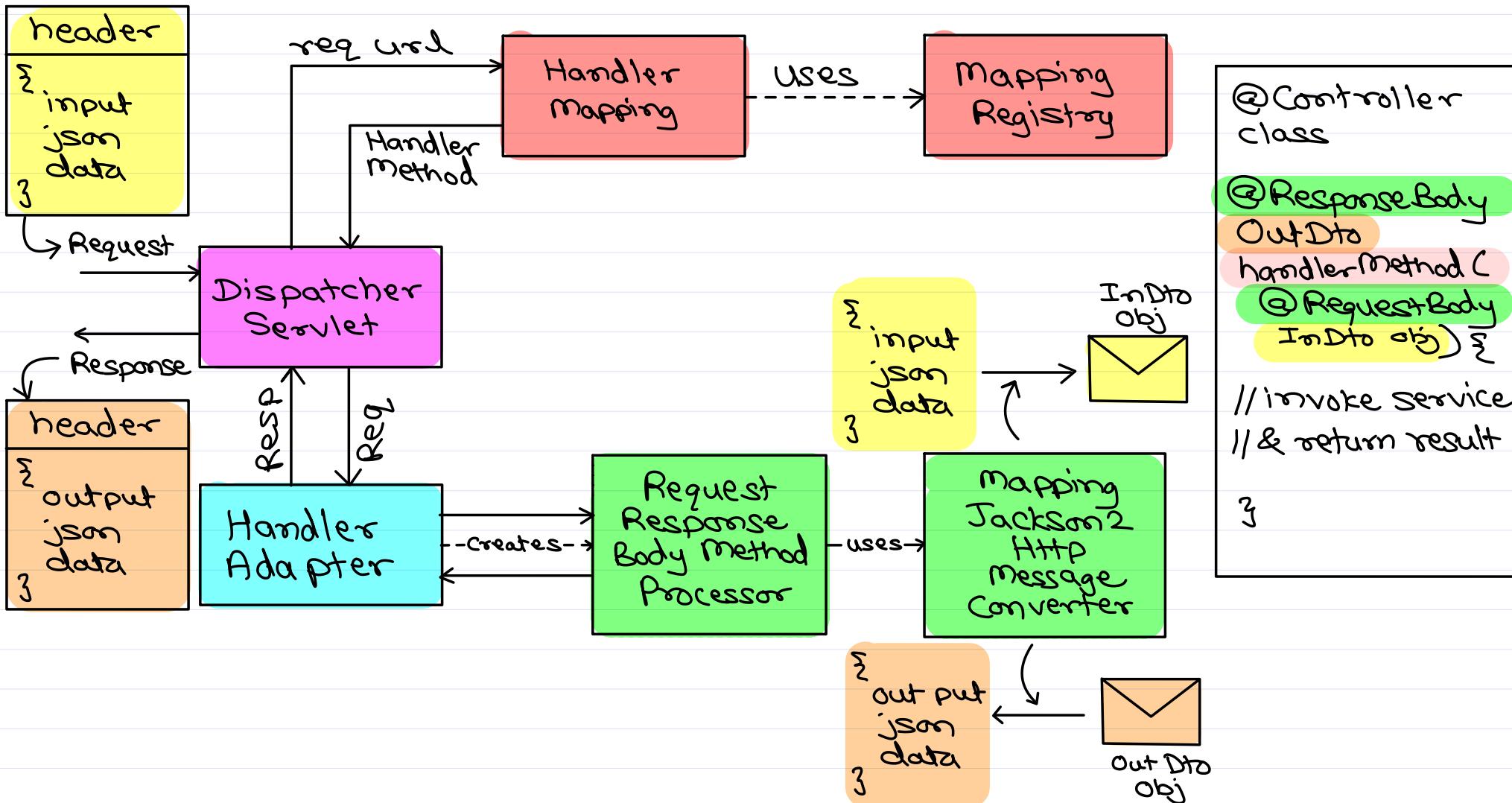
- REpresentation State Transfer
- Protocol to invoke web services from any client.
  - Client can use any platform/language.
- REST works on top of HTTP protocol.
  - Can be accessed from any device which has internet connection.
  - REST is lightweight (than SOAP) – XML or JSON.

- ✓ select • GET: to get records
- ✓ insert • POST: to create new record
- ✓ update • PUT: to update existing record
- ✓ delete • DELETE: to delete record
- ✓ update • PATCH: to update data partially





# REST services internals



@Controller

class MyClass {

@ResponseBody User — (—){

3

3

@Controller / @RestController

class MyClass {

ResponseEntity<User> — (—){

return body(200) or  
status code.

3

3

@RestController

class MyClass {

~~@ResponseBody~~ User — (—){

3

3

URL

http://localhost:8080/admin/books

uri



spring boot (backend)

rest services

- ✓ /admin/...
- ✓ /user/...

8080

java -jar app.jar  
embedded tomcat

Cross origin req

CORS

browser ✓

http://localhost:3001/...

html + js + css

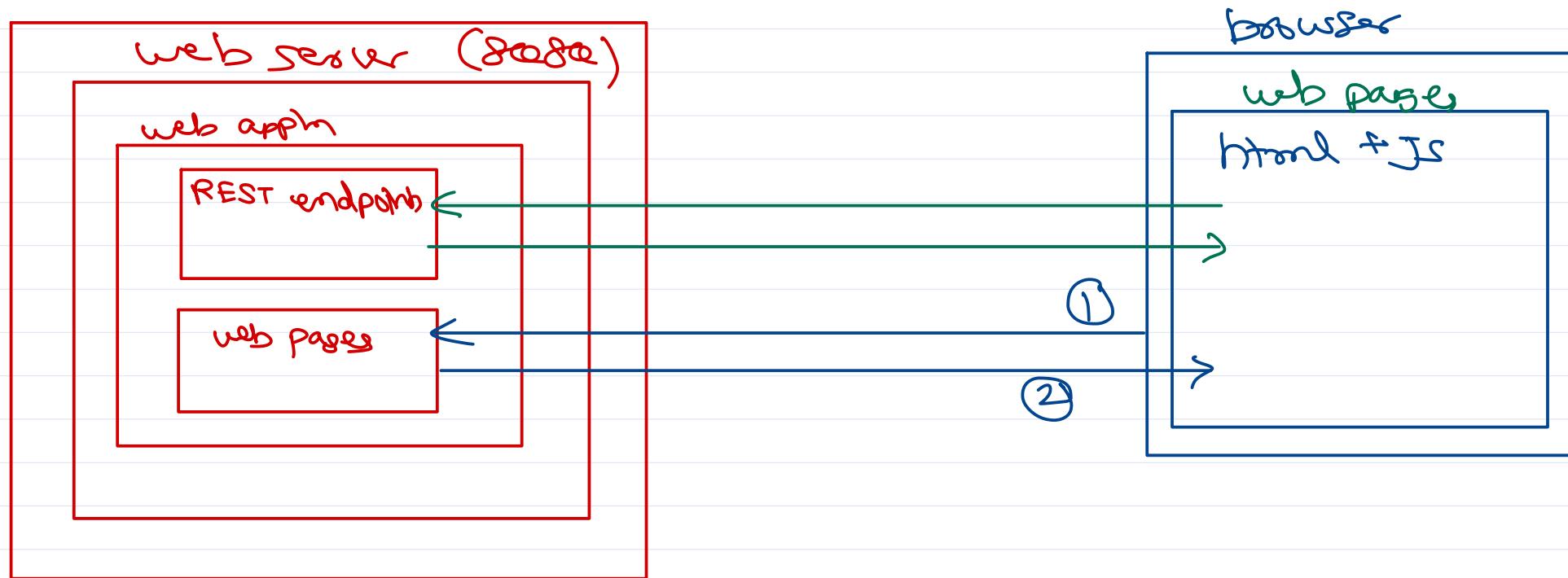
req:

react app  
admin panel

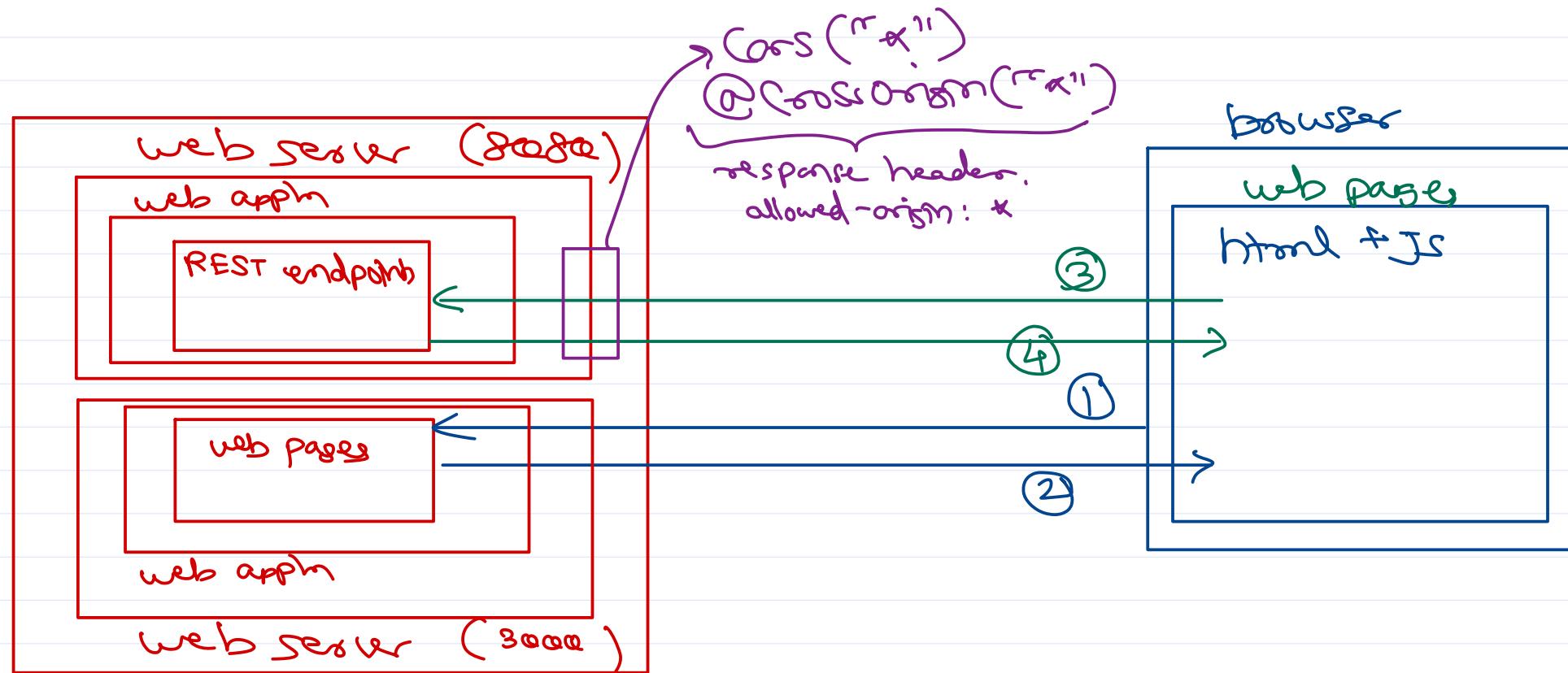
3001

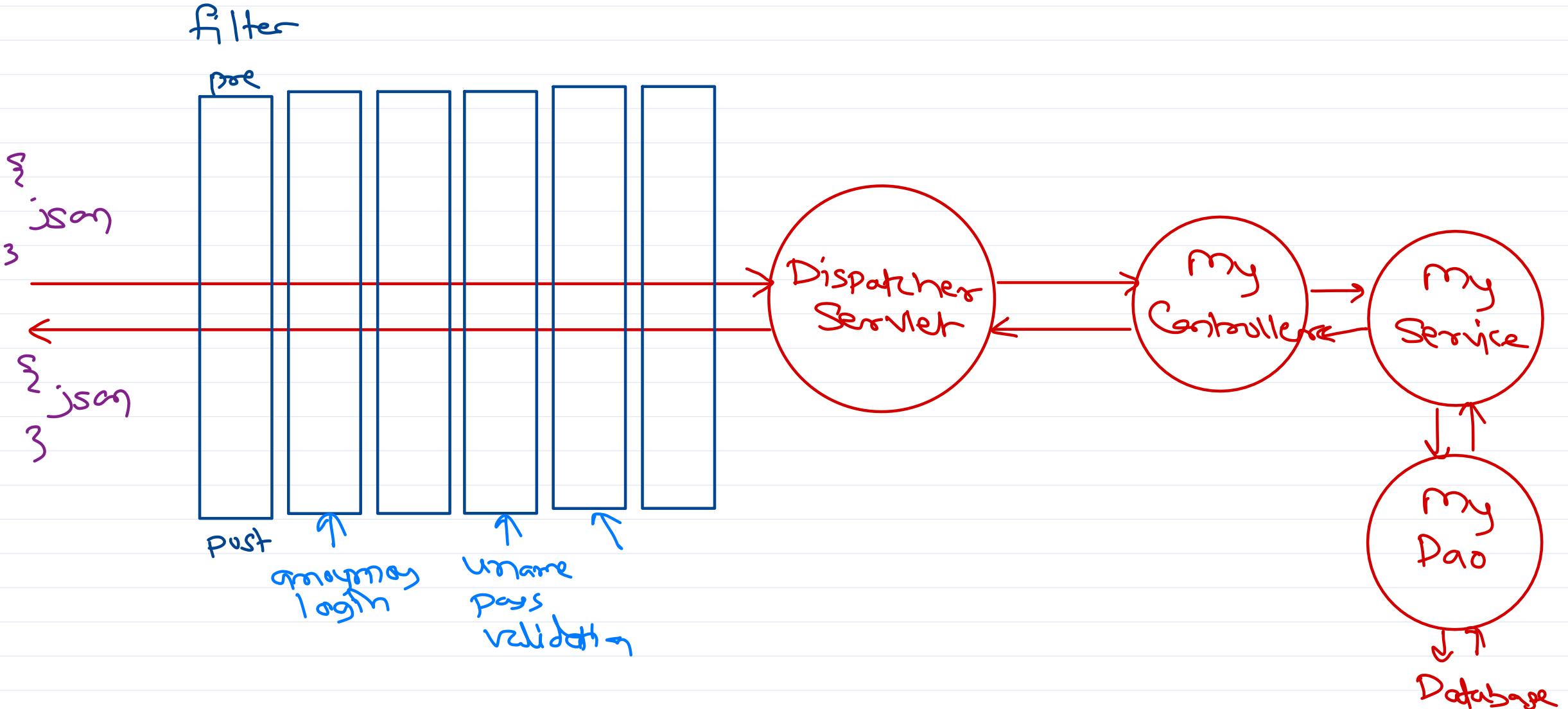
npm start  
dev webserver

# CORS (not required)



# CORS (required)





## ① Authentication

- who are you?
- knowledge based
  - e.g. user+pass, pin, ...
- possession based
  - e.g. I-Card, FP, ...
- two-factor
  - e.g. ATM card+pass, ..

## ② Authorization

- what you are allowed to access?
- user based
- role based

## ③ Principal

- authenticated user

## ④ Granted authority

- a permission

## ⑤ Role

- set of granted authorities



# JWT Security

{ email: "\_\_\_\_"  
  pass: "\_\_\_\_"  
  3

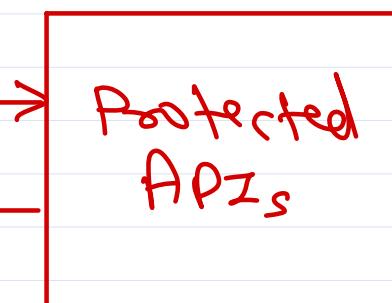


Auth  
Controller

←  
token (JWT)

+ token

{  
  req  
  3  
  body



←  
{  
  response  
  3  
  body

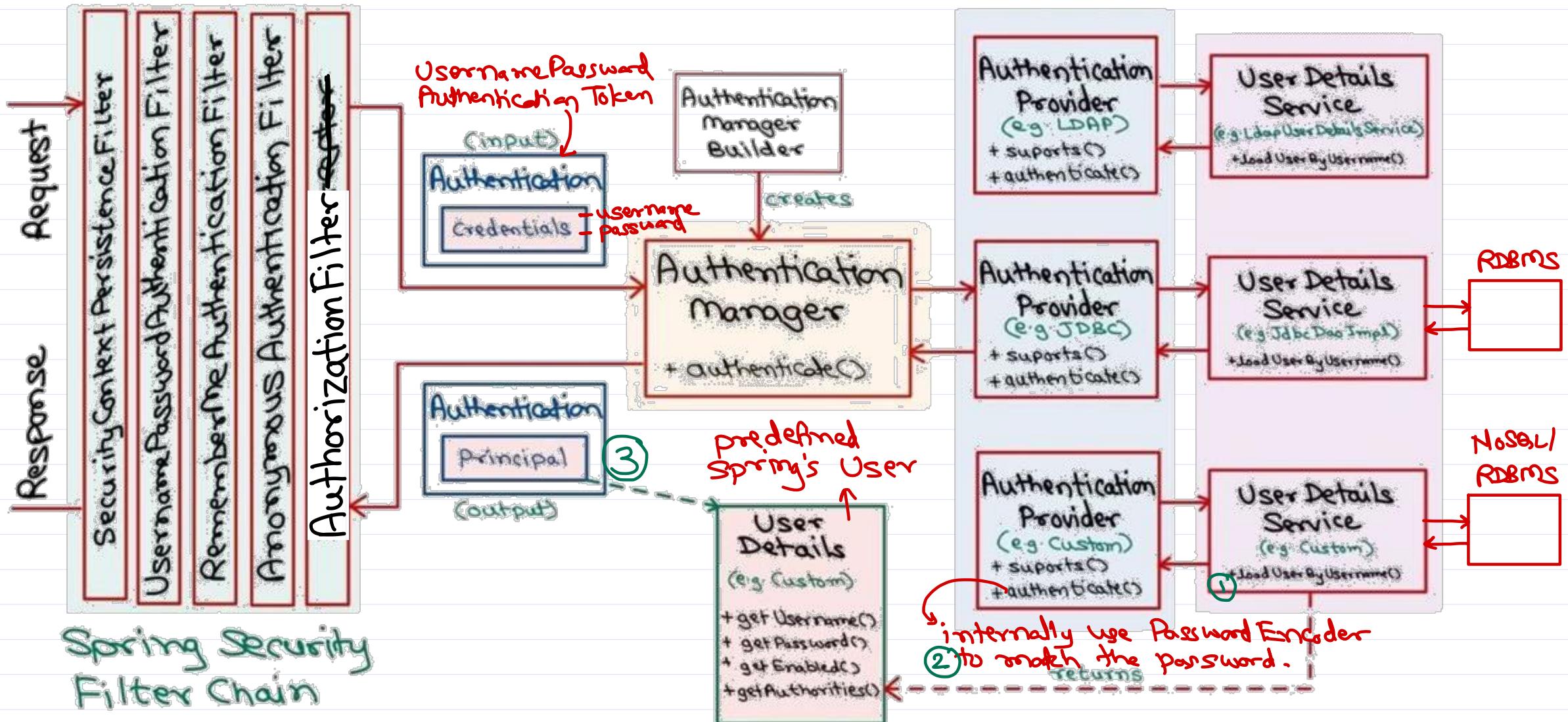
## Why JWT?

- ① REST protocol is stateless. So each req must have something to indicate that user is valid.
- ② JWT is tamper-proof. Because it is encrypted. It can be read by client, but cannot be modified by client.
- ③ Do not store any sensitive info in JWT token. Keep it in db.

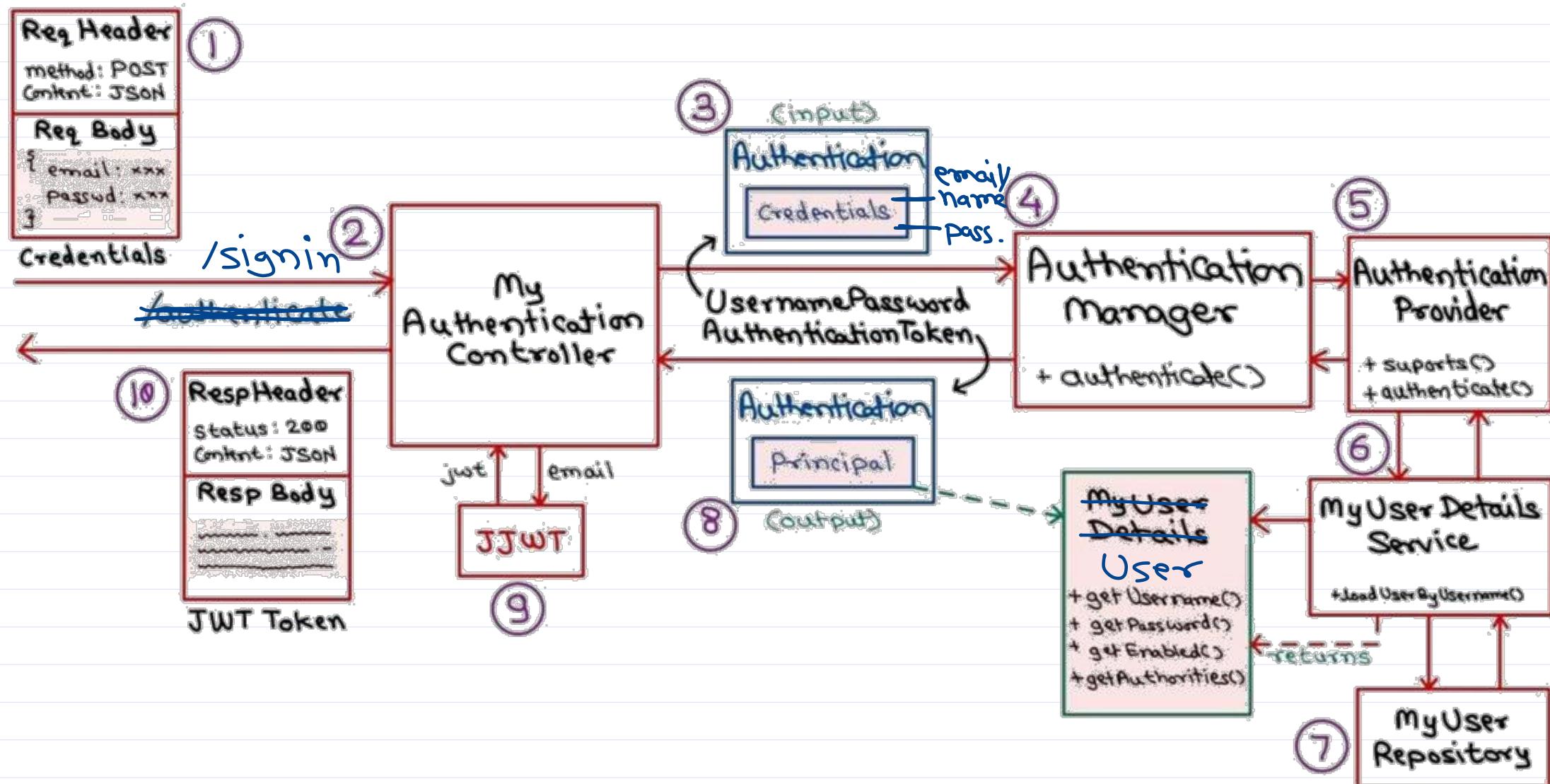
JWT token = Header (also enc)  
+ Payload (data)  
+ Signature (secret key).



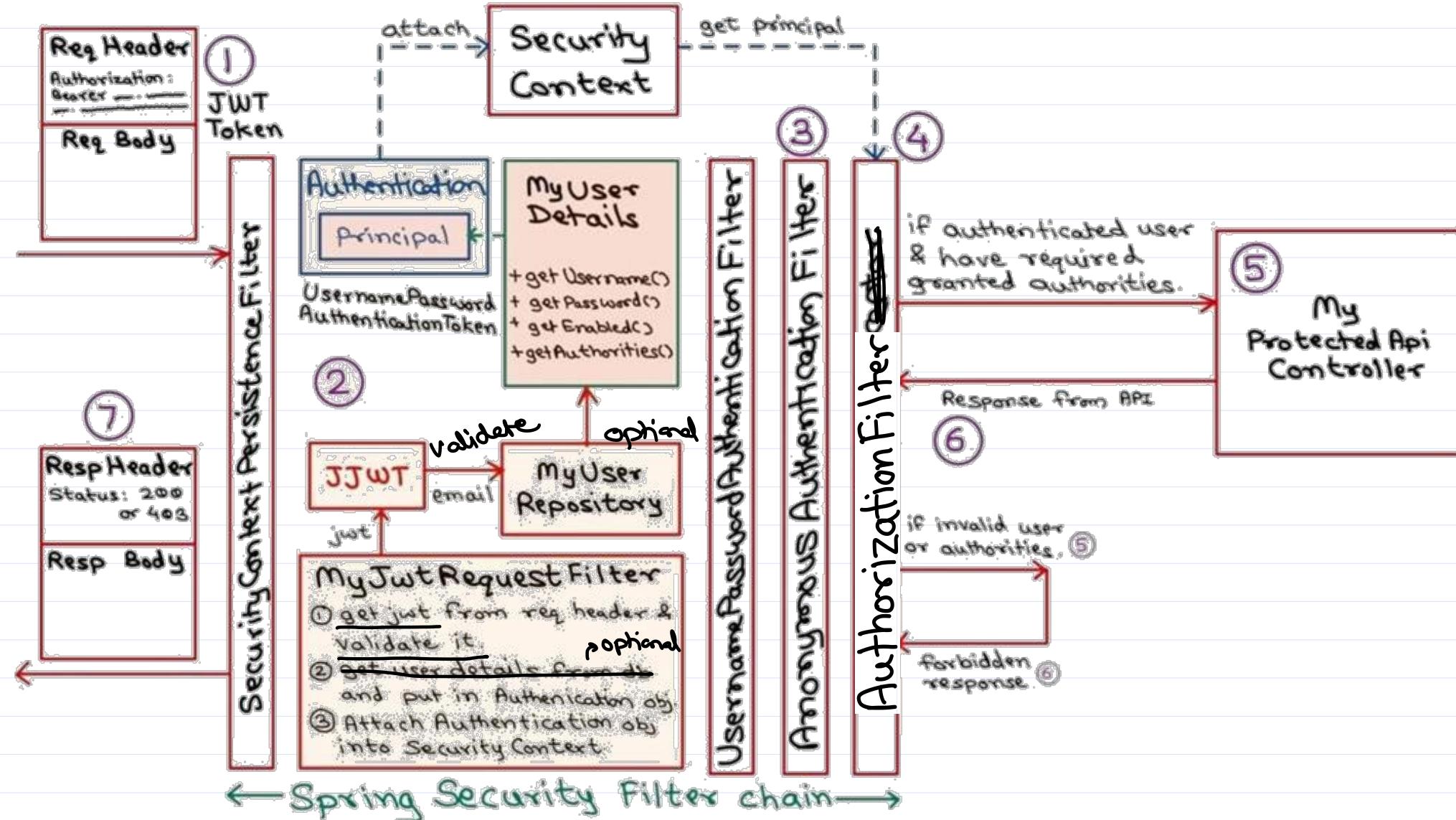
# Spring Web Security



# Spring REST Security (1)



# Spring REST Security (2)





*Thank you!*

Nilesh Ghule <[nilesh@sunbeaminfo.com](mailto:nilesh@sunbeaminfo.com)>

