Sunbeam Infotech

Trainer: Nilesh Ghule

Default methods

```
class A {
    void fun() {
    }
};

class B {
    void fun() {
    }
};

class C : public A, public B {
};

C obj;
obj.fun(); //error
```

OR

```
interface A {
    def void fun() {
    }
}

interface B {
    def void fun() {
    }
}

class C implements A, B {
};
```

error.

C++

```
class A{
    void fun(){
    }
};

class B{
    void fun(){
    }
};

class C: public A, B{
};

C obj;
obj.fun(); // ambiguity error.
           // A::fun or B::fun
```

Java 7 or C#

```
interface A{
    void fun();
}

interface B{
    void fun();
}

class C implements A, B{
    @Override public void fun(){ ... }
}
```

Java 8 - default method in interfaces

```
interface A{
    default void fun(){
    }
}

interface B{
    default void fun(){
    }
}

class C implements A, B { // error
    must implement fun()
}
```

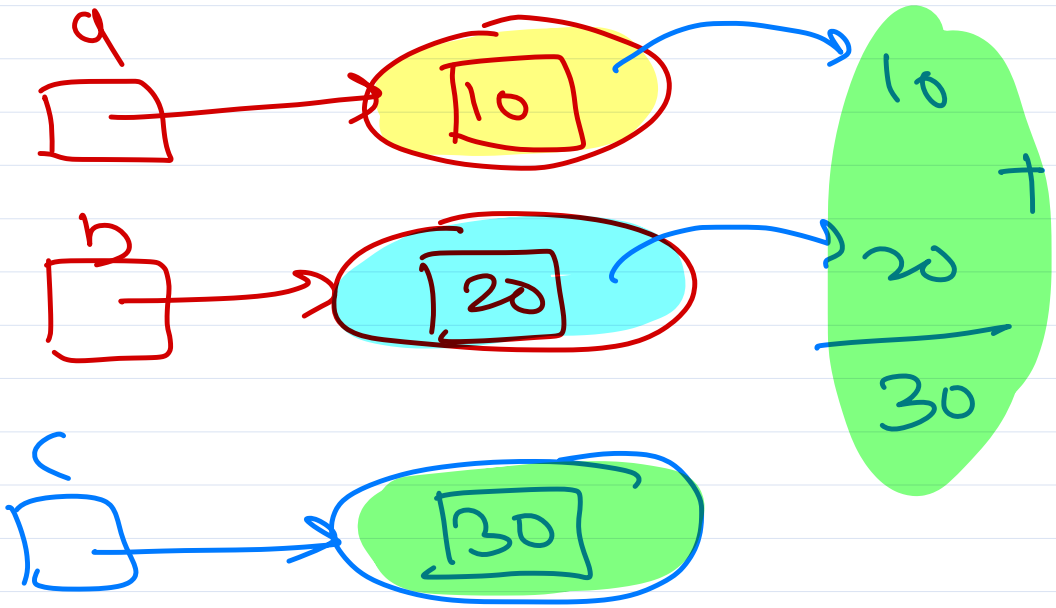
Wrapper classes

list.add(10);

↳ int → Integer

boxing

```
Integer a=10, b=20, c;
c= a+b;
sysout(c); //30
```



Wrapper classes

Wrapper

primitive	wrapper objects
int	Integer
float	Float
double	Double
byte	Byte
short	Short
long	Long
char	Character
boolean	Boolean

Wrapper uses

1. primitive → object
   - list.add(123);
   - int → Integer
2. Conversion methods
   - intValue()
   - shortValue()
   - longValue()
   - byteValue()

   (Number class)
   - valueOf()
   - parseXYZ()
3. static helper methods
   - max(), min()
   - sum()

   → Integer

Auto boxing

primitive type → wrapper type

```
int i = 10;
Integer j = i;
```

Auto unboxing

wrapper type → primitive type

```
int k = j;
```

Packages

1. code organization
2. maintainability / modularity
3. avoid name clashing

Pass by value and Pass by reference

① In java, objects are always pass by reference.

```
main() {
    Date d = new Date();
    method(d);
}
static void method(Date d) {
}
```

- no copy created / only addr passed
- changes reflected in calling

② In java, prim values are always pass by value.

- copy is created.
- changes not seen in calling.

```
void swap(int x, int y) {
    int t = x;
    x = y;
    y = t;
}
main() {
    a = 10, b = 20;
    swap(a, b);
}
```

```
void swap(int[] a) {
    int t = a[0];
    a[0] = a[1];
    a[1] = t;
}
main() {
    int[] ar = {10, 20};
    swap(ar);
}
```

```
ArrayList list = new AL();
list.add(11);
list.add(22);
list.add(33);
list.add("44"); ← ClassCastException
for(Object ele: list) {
    Integer i = (Integer) ele;
    sysout(i);
}
```

```
ArrayList<Integer> list = new AL<>();
list.add(11);
list.add(22);
list.add(33);
list.add("44"); //Compiler error.
for(Integer ele: list) {
    sysout(ele)
}
```

```
void print(List<Integer> list) {

}
void print(List<String> list) {

}
```

error ~~overloading~~

Class

- ✓ fields, methods, constructors
- ✓ use: objects, inherit, composition.
- ✓ override method in sub-class & invoke using super-class ref (polymorphism)

Abstract class

called when subcls object is created.

- ✓ fields, methods, constructors
- ✓ use: inherit in sub classes
- ✓ Force sub classes to override abstract methods.
- ✓ invoke overridden methods using super-class ref (polymorphism)
- ✓ conceptual / logical entity - object not in real-world.

interface

- method declarations
- inherited in sub-classes
  - force sub-cls to override methods
    - Contract.
    - specification / standards.
- ✓ invoke overridden methods using interface ref (polymorphism)
- ✓ multiple inheritance supported.

marker interface

- no method declarations too. (empty intf)
- associate info / meta data with class - to be validated by other classes / JVM.
- mark / sign that class has some feature.

import = import public types from other packages.

```
import java.util.Date;
    Date d=new Date();
    java.util.Scanner sc=new ... ();
```

import static = import static members of a type.

```
import static java.lang.Math.PI;
    area = PI * Math.pow(rad, 2);
```

```
ArrayList al = new ArrayList();
al.add(11);
al.add(22);
al.add("33");
for(Object ele : al)
    sysout( (Integer) ele);
```

Java 1.4

→ ClassCastException

generics
- classes
- methods
- interfaces

```
ArrayList<Integer> al = new ArrayList<>();
al.add(11);
al.add(22);
al.add("33"); // Compiler error
for(Object ele : al)
    sysout( (Integer) ele);
```

Java 5.0

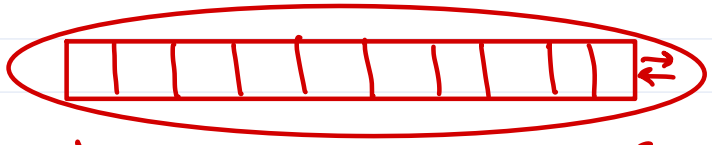
Prim types are always pass by value;
- copy created. Changes in called are not seen in calling method.

Ref types/objects are always pass by ref.
- addr is copied. Changes in called method are seen in calling method.

```
void swap(int [] arr){
    t = arr[0];
    arr[0] = arr[1];
    arr[1] = t;
}
swap(new int [] {11, 22});
```

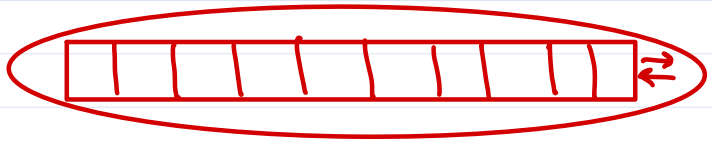
```
ArrayList<int> X
new T() X
new T[] X
throw T(); X
print(Box<Integer> b) { }
print(Box<Double> b) { } X
```

Vector - Legacy (1.0) - Synchronized.

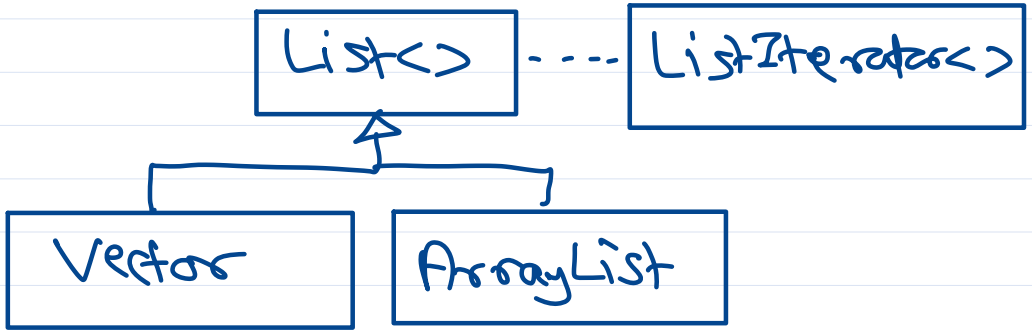


- traversal: Enumeration, Iterator (1.2)

ArrayList - 1.2 - Non-synchronized - Fast.

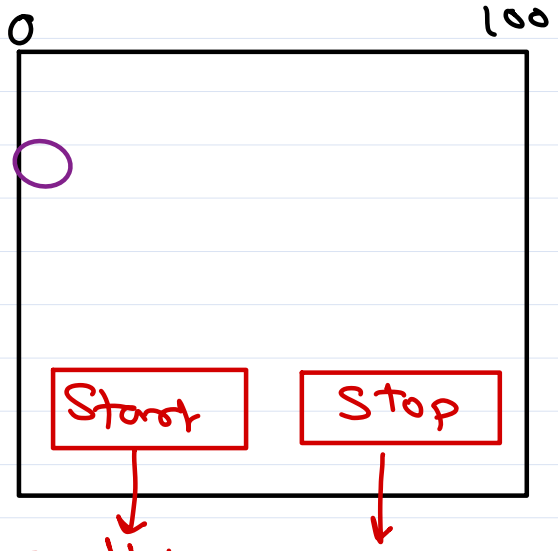


- traversal : Iterator (1.2)




```
ListIterator<String> itr;
itr = list.listIterator();
while(itr.hasNext()){
    String ele = itr.next();
    //...
}
itr = list.listIterator(list.size());
while( itr.hasPrevious()){
    String ele = itr.previous();
    //...
}
```

```
class Ball extends Thread{
    int x, y;
    static volatile flag=true;
    public void run() {
th1     while(flag) {
            for(i=0;i<100;i++){
                x=i; draw();
            }
            for(i=100;i>0;i--){
                x=i; draw();
            }
        }
    }
}
```




```
on_click()
Ball b=new Ball();
b.start();

th2
on click() {
    Ball.flag=false;
}
```

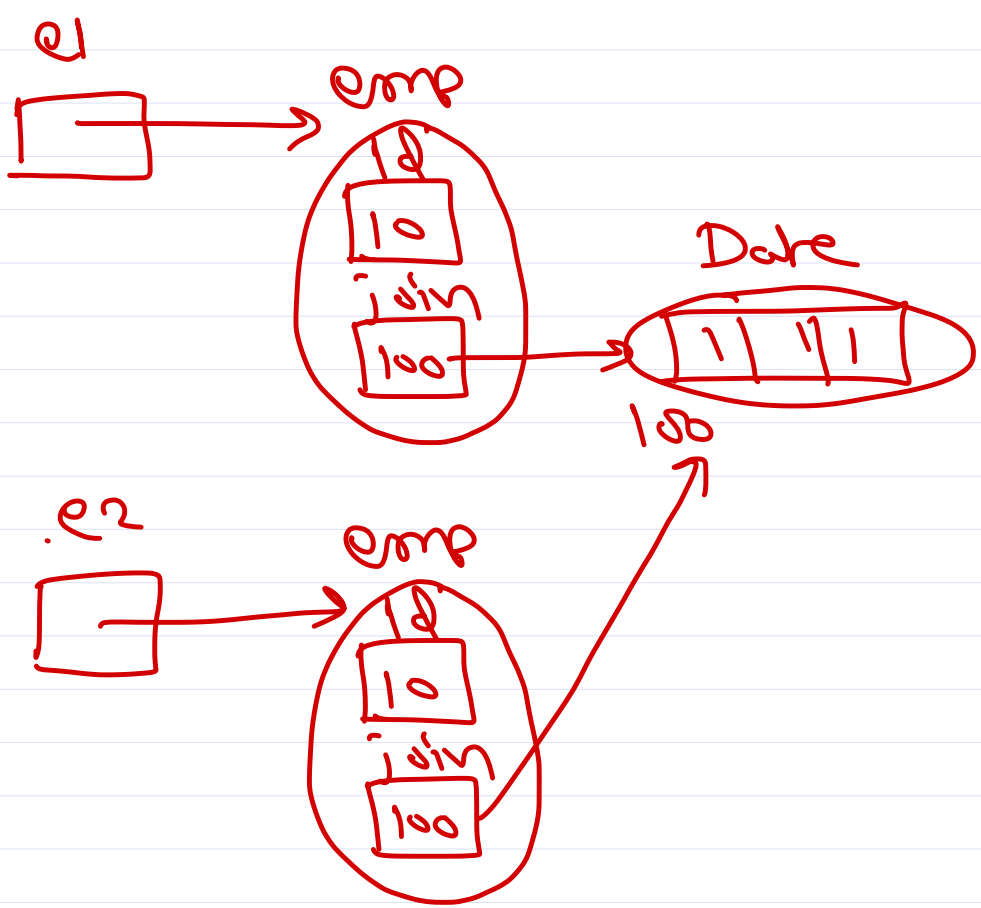
{1, 2, 1, 2, 1, 3, 4, 3, 2} ← s

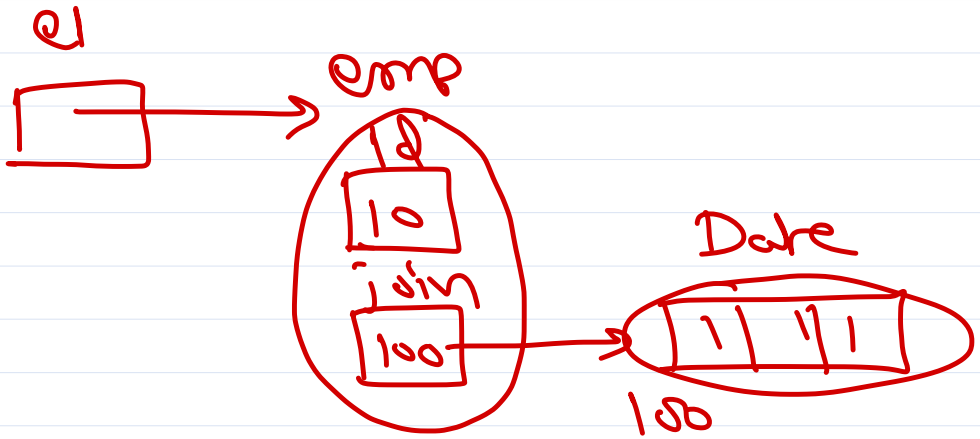
```
num = 3;
r = s.filter(i -> i==num)
     .count();
```

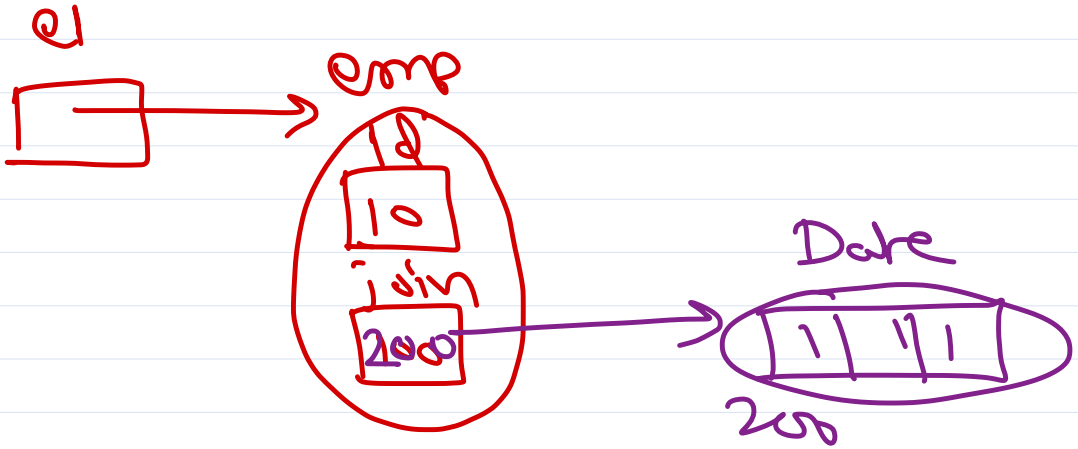
Collectors.toMap()
.groupingBy()

1 → 1, 1, 1
2 → 2, 2, 2
3 → 3, 3
4 → 4

Shallow and Deep copy







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

(id = 2 : true)

// all records will be fetched.

SQL injection attack

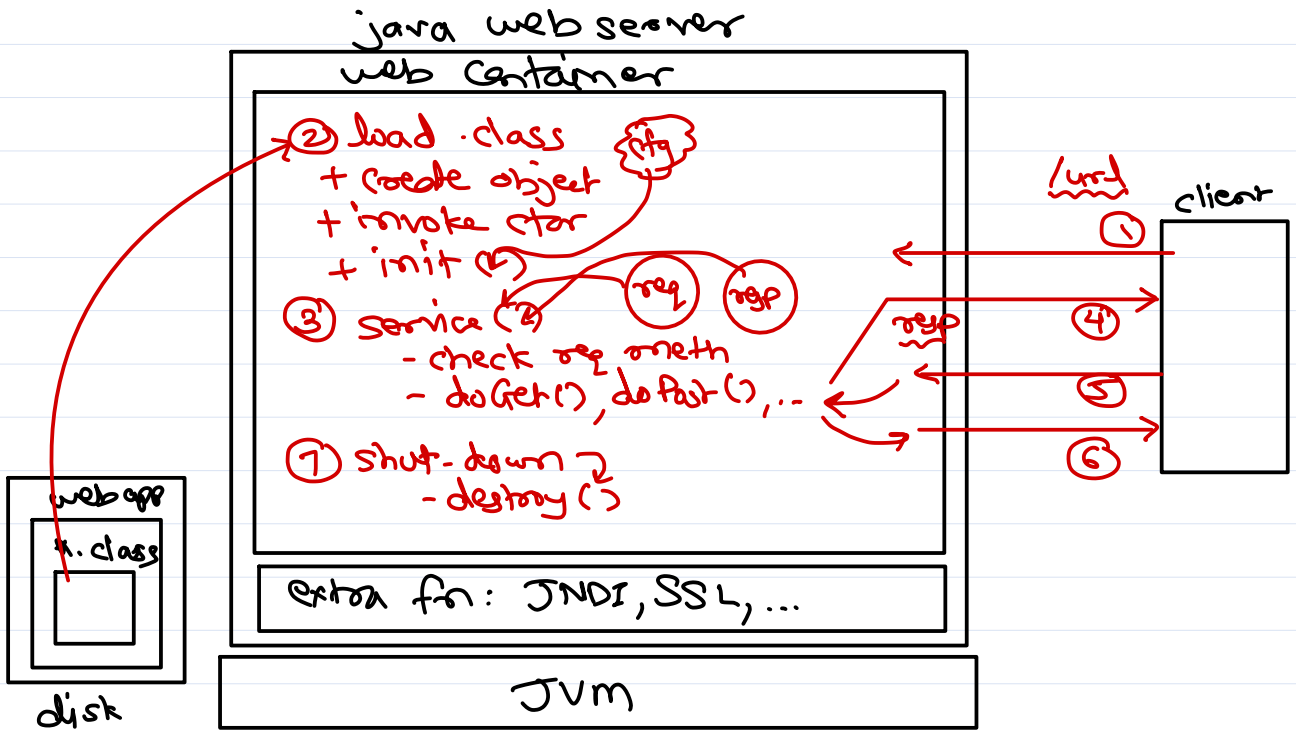
Solution: no string concat X

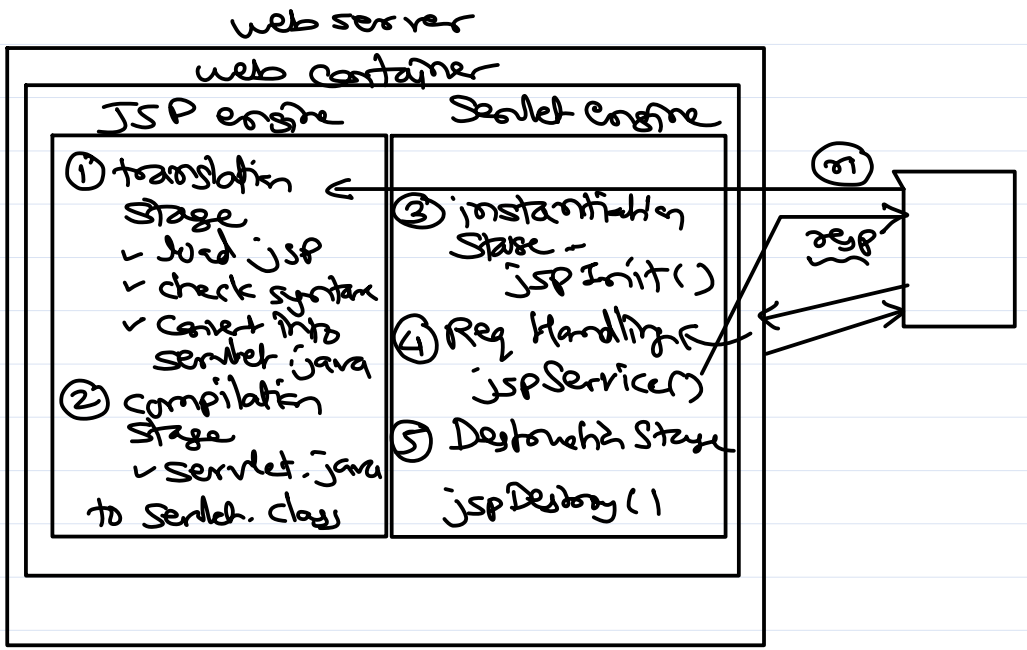
String sql = "Select * from tbl where id = ? "

use preparedStatement or CallableStatement ie. stored proc.

JPQL/HQL param queries internally use PreparedStatement.

Servlet and JSP life cycle





web.xml

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

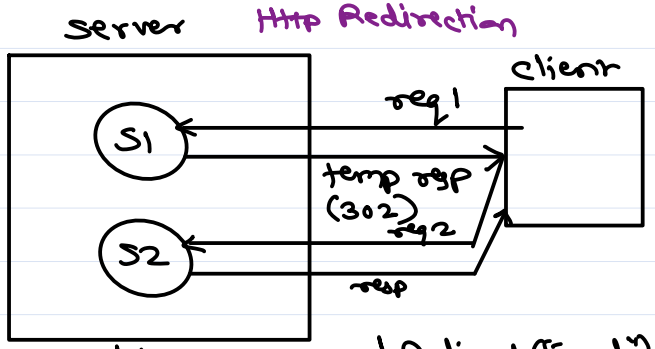
eager loading

when appln is deployed servlet class loaded, obj created, ctor called & init() called.

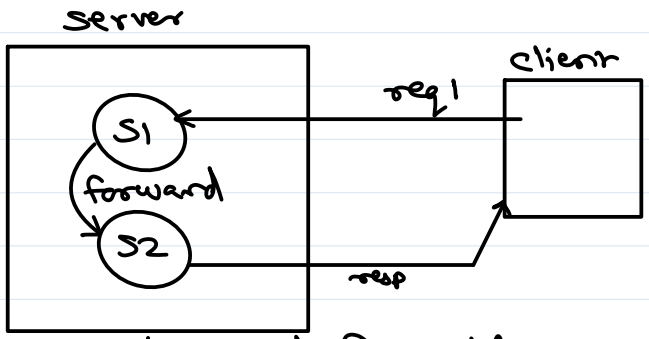
servlet obj is ready for next requests.

e.g. Spring mvc - DispatcherServlet.

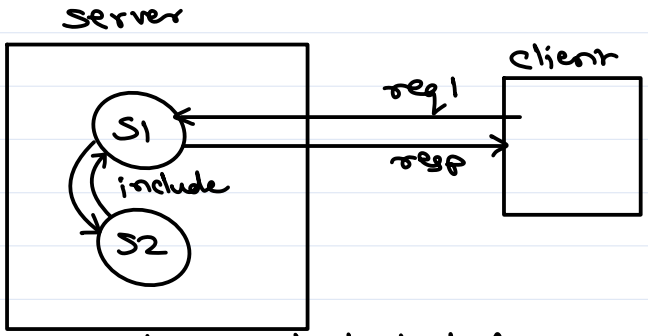
Redirection vs RequestDispatcher



servlet : resp.sendRedirect("url");
spring : view name - "redirect : url";

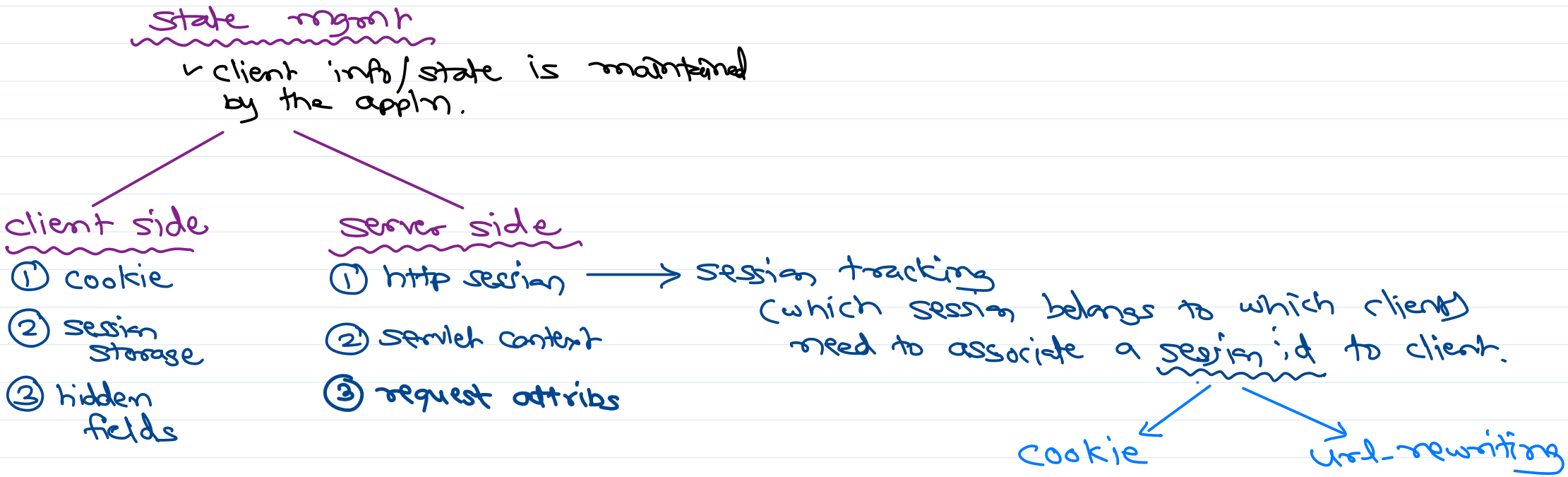


servlet : rd.forward(req, resp);
spring : view name "forward: url".

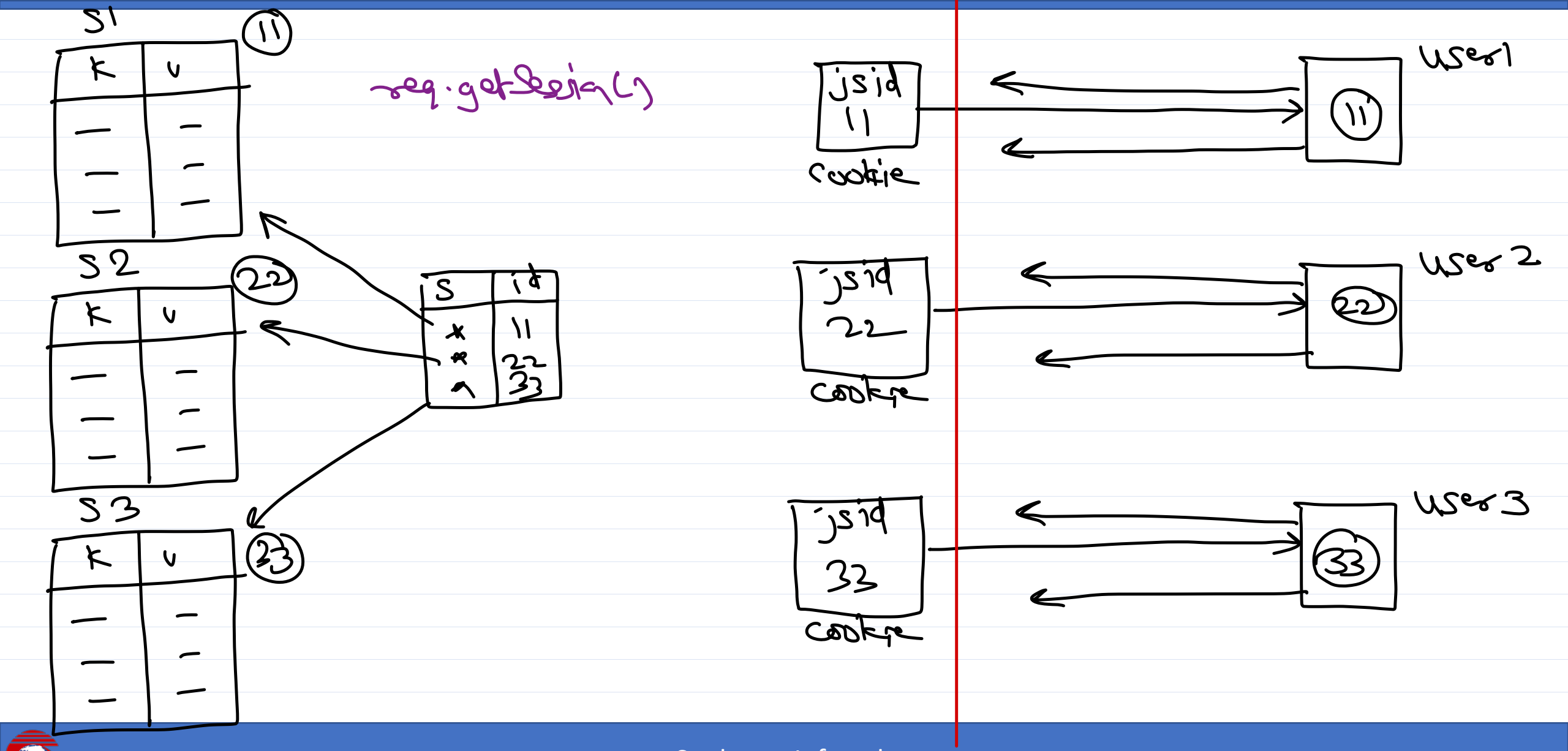


servlet : rd.include(req, resp);

State Management

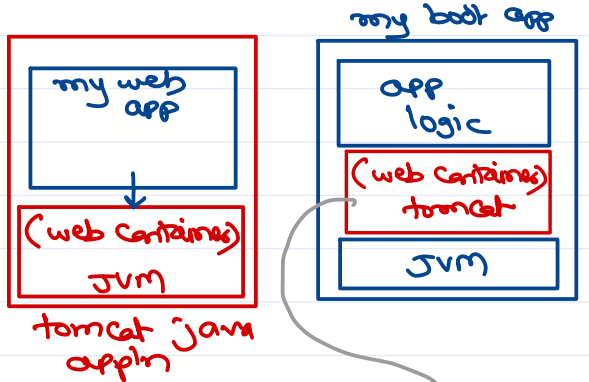


Session tracking

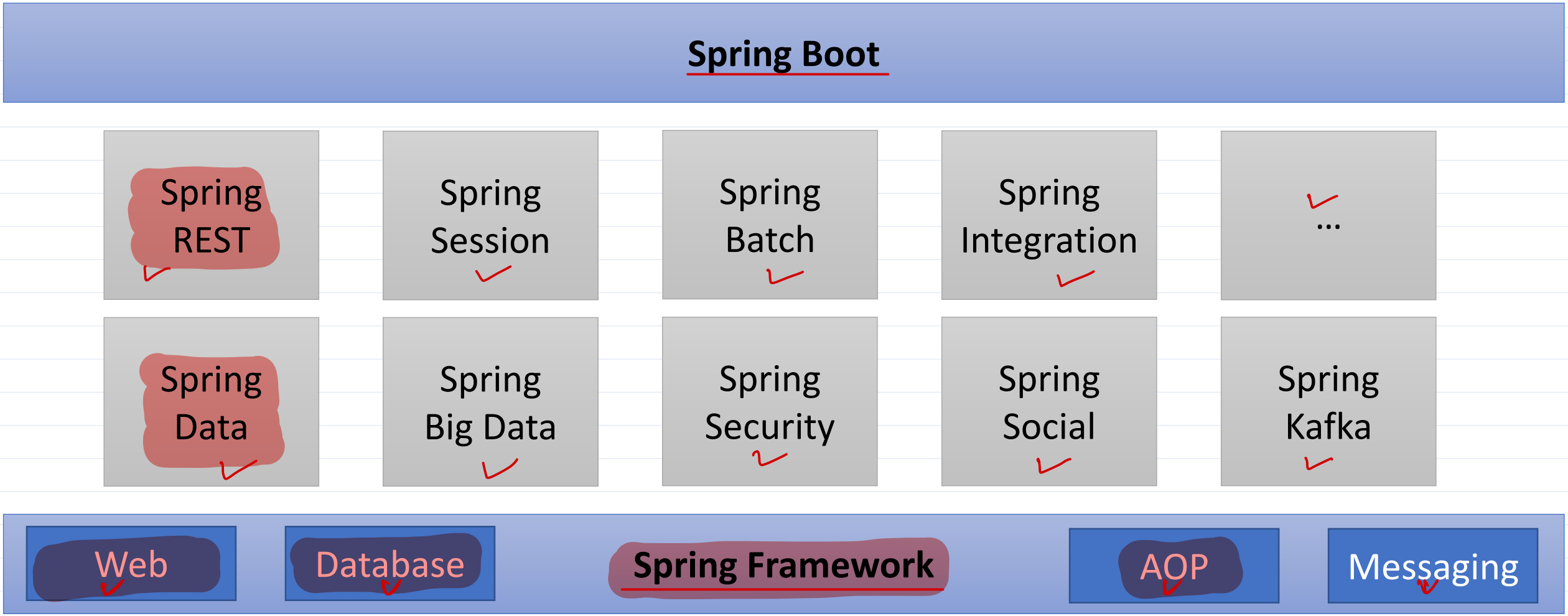


Spring Boot

- 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”.
- 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 framework + Embedded web-server + Auto-configuration - XML config - Jar conflicts

Dependency Injection


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

Spring Config
xml or anno configs

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

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

@Autowired
- field level
- setter method level
- constructor level → preferred by spring boot.

Dependency Injection
1. setter based di
2. constructor based di
3. field based di

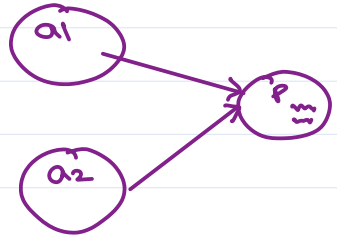
Autowiring
specify DI, auto selecting bean object to be injected.

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

app.properties

under @Configuration class

```
@Bean
Person p1() {
    return new Person(...);
}
```



Spring bean life cycle

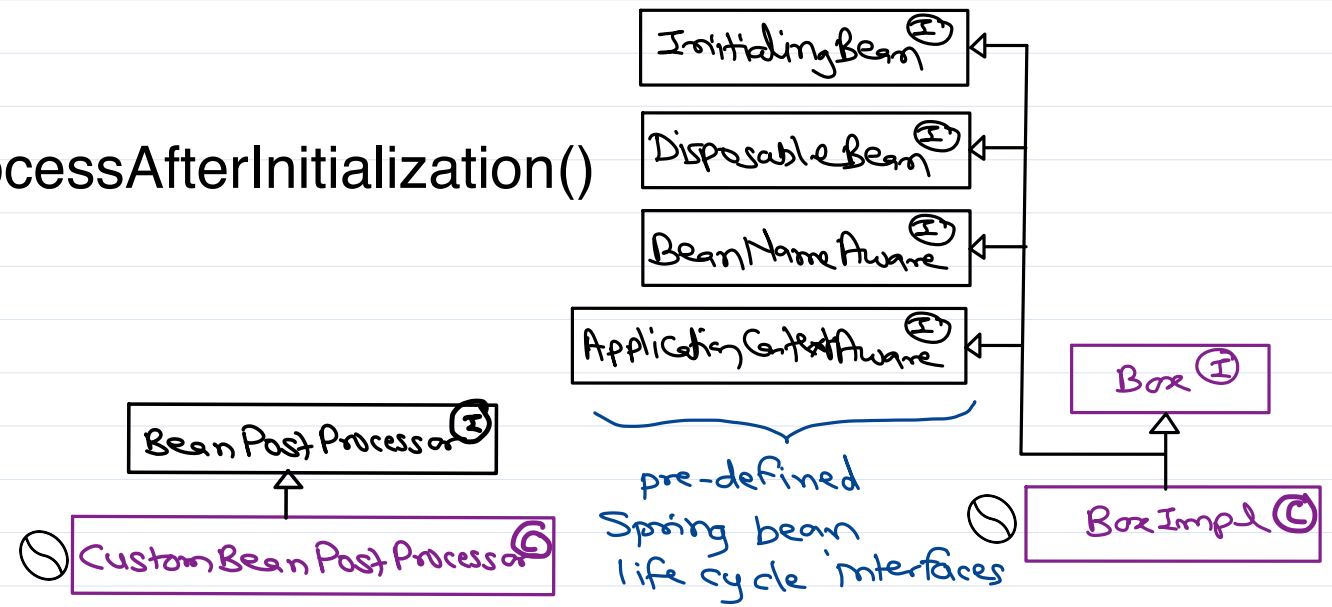
- Spring container creates (singleton) spring bean objects when container is started.
- Spring container controls creation and destruction of bean objects.
- There are four options to control bean life cycle.
  - InitializingBean and DisposableBean callback interfaces
  - Aware interfaces for specific 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() (@PostConstruct)
  7. InitializingBean.afterPropertiesSet()
  8. Custom BeanPostProcessor.postProcessAfterInitialization()
- Bean destruction
  1. Custom destroy() (@PreDestroy)
  2. DisposableBean.destroy()
  3. Object.finalize() & G.C.

Custom bean post processor allows you to write some logic before & after bean initialization outside the bean class.

boot 04

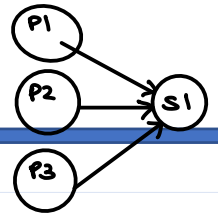


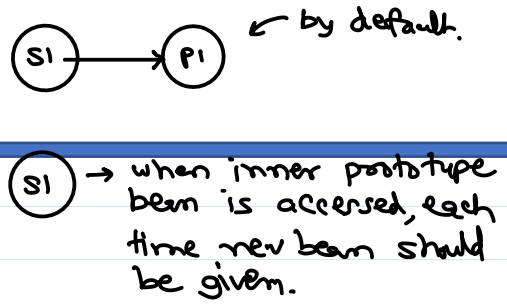
Spring Bean Life Cycle

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

1. Bean obj create (new) - constructor
2. init → setters are called
3. setBeanName()
4. setApplicationContext()
5. beforePropertiesSet()
6. @PostConstruct
7. afterPropertiesSet()
8. Bean ready.

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).

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

- 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.

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

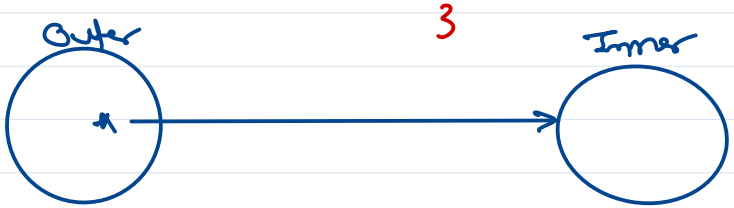
Bean scopes

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

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

3

3

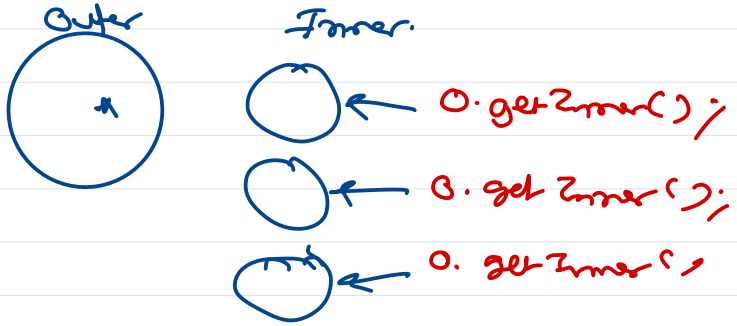



```
@Scope("singleton")
@Component
class Outer {
    @Lookup
    Inner getInner(){
        return null;
    }
}
```

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

3

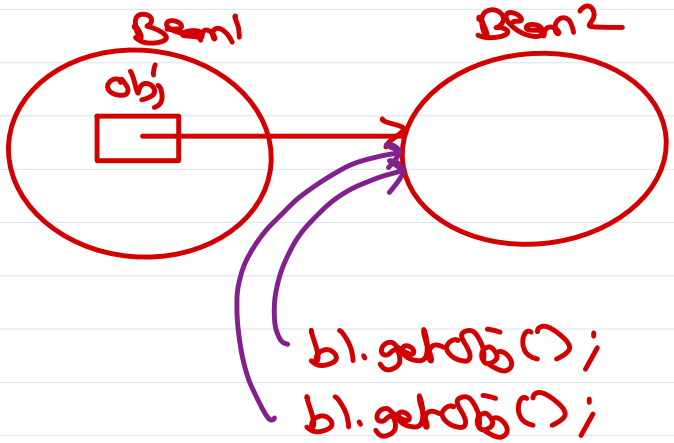
3



Prototype bean in Singleton bean

```
@Scope("singleton")
@Component
class Bean1 {
    @Autowired
    Bean2 obj;

    getObj() {
        return obj;
    }
}
```

```
@Scope("prototype")
@Component
class Bean2 {
}
```




```
@Scope("singleton")
@Component
class Bean1 {
    @Lookup
    Bean2 getObj() {
        return null;
    }
}
```

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

```
@Scope("singleton")
@Component
class Bean1 {
    @Autowired
    AppCtx ctx;
    Bean2 getObj() {
        return ctx.getBean(Bean2.class);
    }
}
```

```
@Scope("prototype")
@Component
class Bean2 {
}
```

b1.getObj() → ctx.getBean() → Bean2

b1.getObj() → ctx.getBean() → Bean2

Hibernate Architecture

CRUD ops
Hibernate Associations
HQL & Auto generated PK.

Java Application

obj
new Book

HIBERNATE

Persistent obj

Session Cache
Key | Value
book:11 | 11 obj Book
book:25 | 25 obj Book

Session

CRUD
- persist()
- delete()
- update()
- find()

open

SessionFactory Cache

SessionFactory

build

ServiceRegistry & Metadata

Configuration

NativeQuery ← SQL

(JPA) Criteria Query — where clause in select & DML sql.

ProcedureCall ← Call stored proc.

Criteria (deprecated) → where clause in select sql.

beginTransaction() → start tx.

mandatory for DML ops.

Transaction → commit() / rollback()

Query ← Hiberate Query Language

handle RDBMS differences
Actual SQL query generation as per RDBMS

Dialect

MySQL5Dialect, MySQL8Dialect, Oracle11gDialect, H2Dialect, ...

JTA

Java Transaction API

JDBC

JNDI

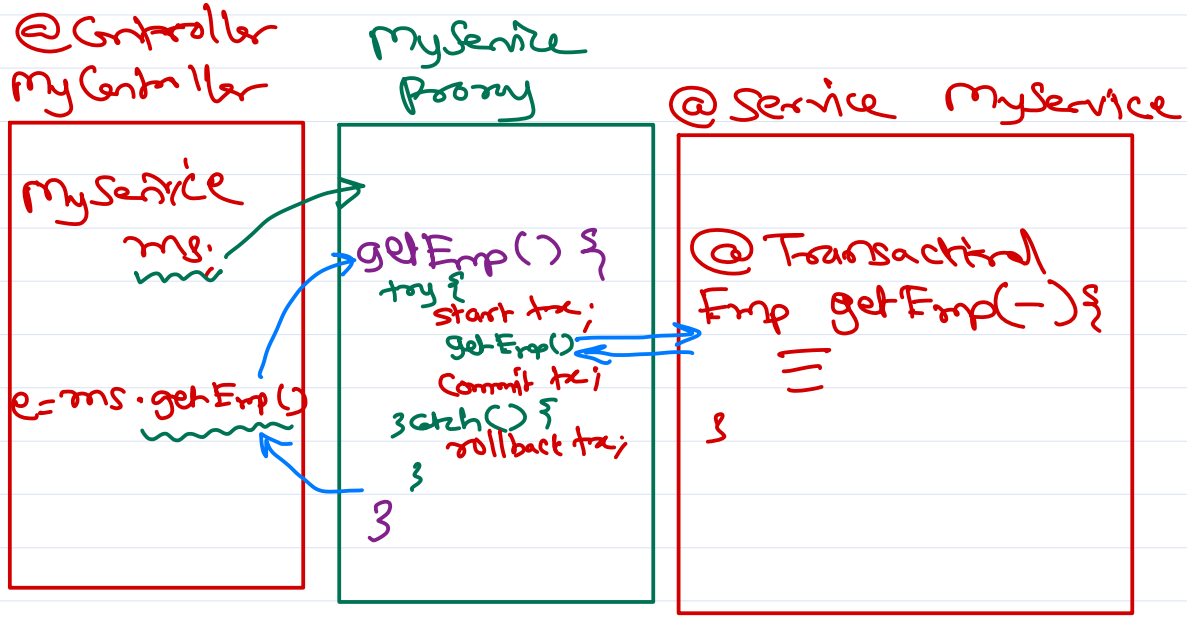
Java Naming & Directory Interface

RDBMS

hibernate.cfg.xml — url, password, username, dialect

@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.

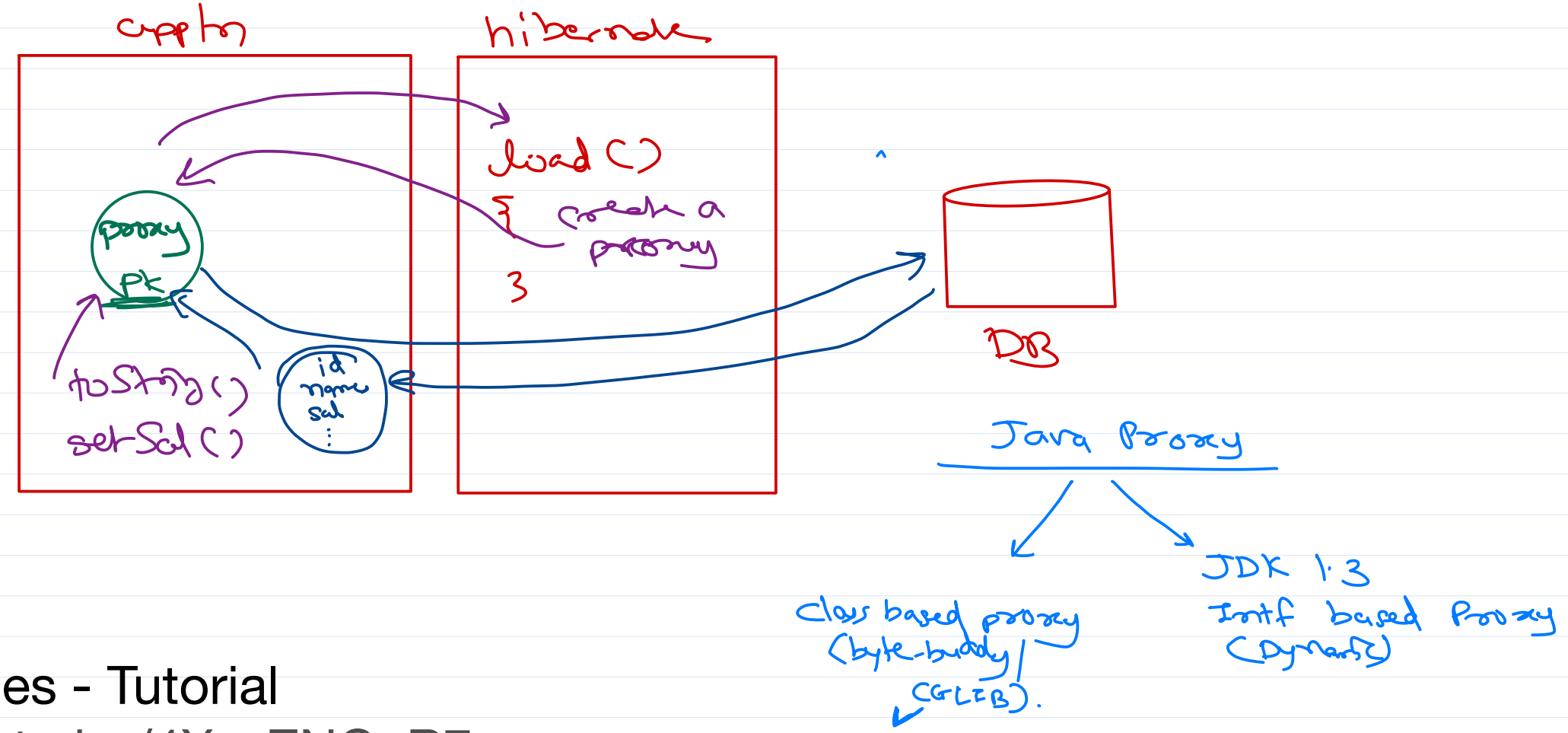


```
OrderService {

    @Transactional
    placeOrder() {
        orderDao.save(o);
        paymentDao.save(p);
        inventoryDao.update(s);
    }

}
```

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.

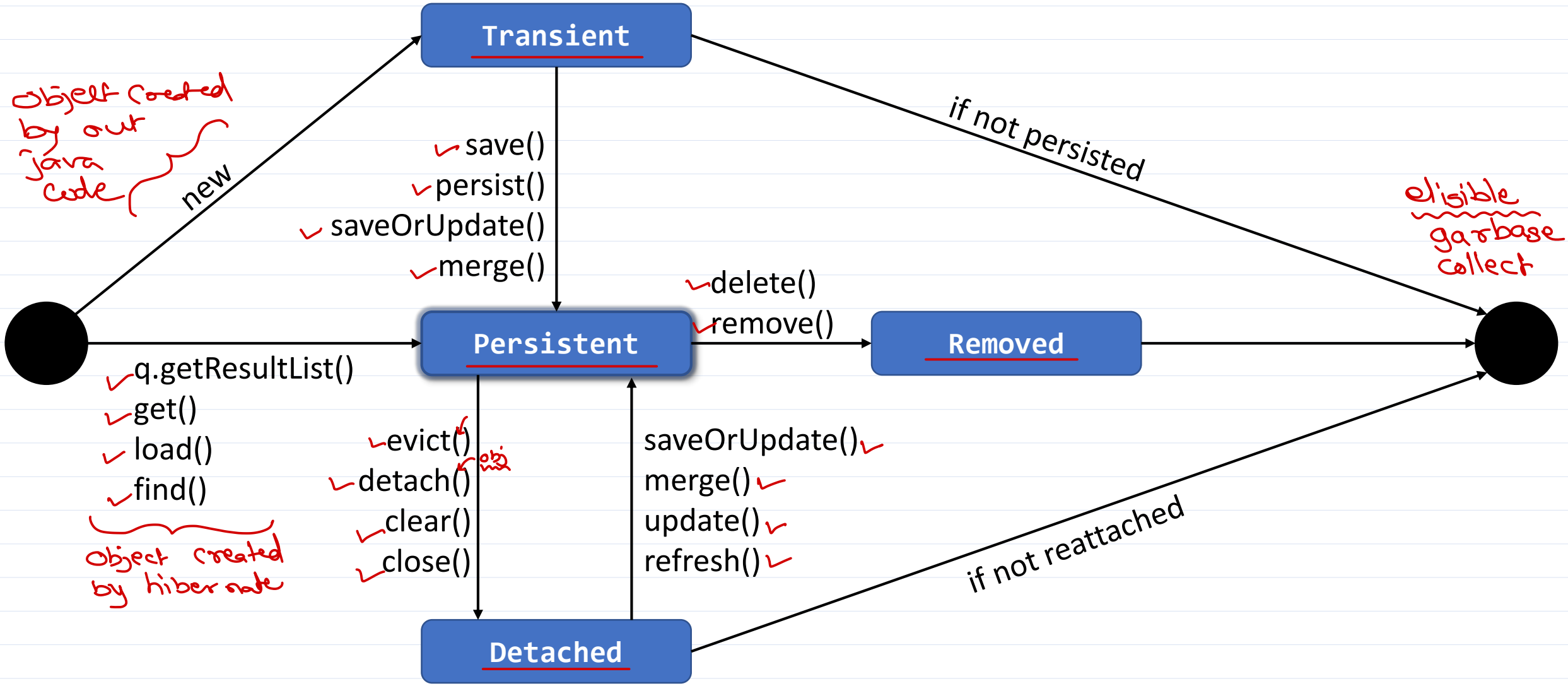
Hibernate load()



Java Proxies - Tutorial

https://youtu.be/4X_sZNOeR7g

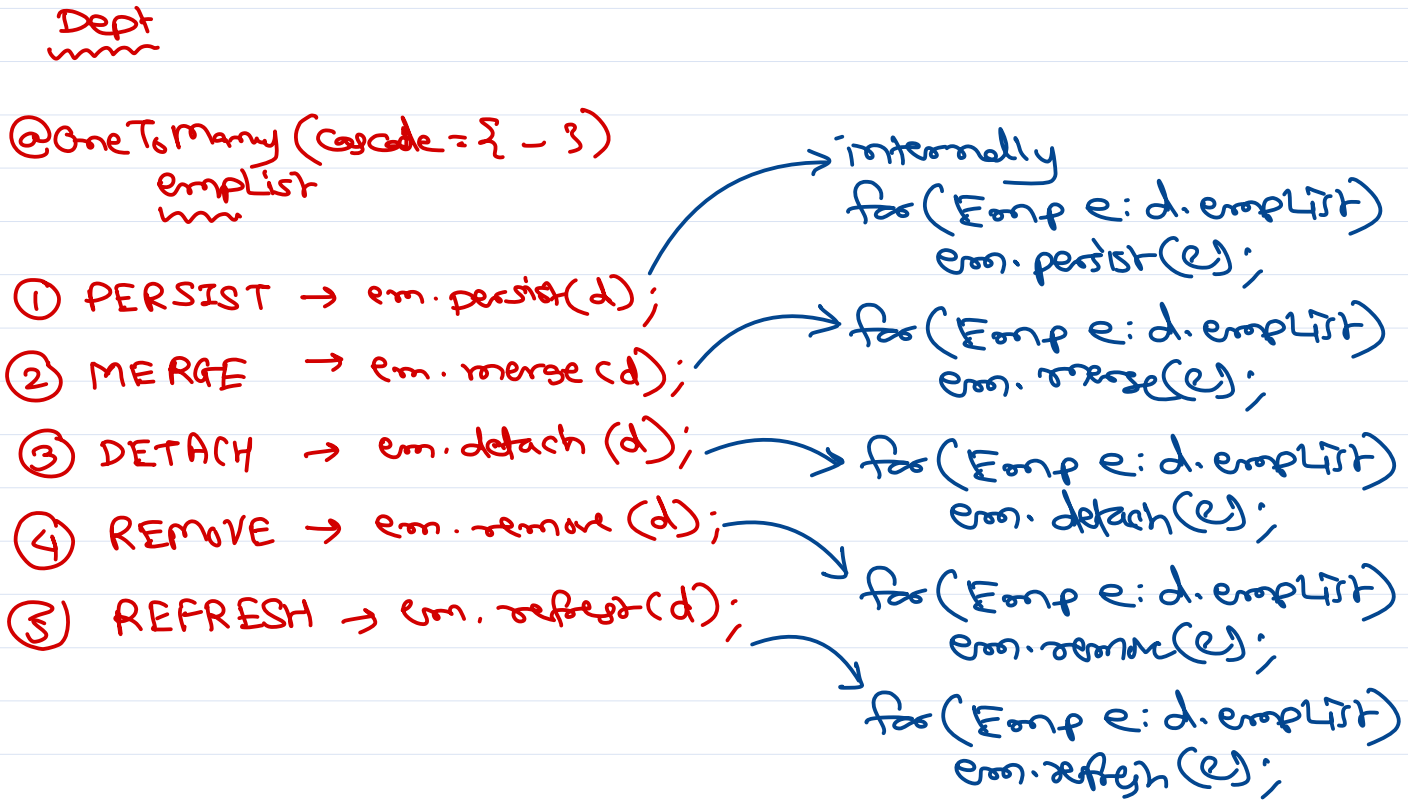
Hibernate Entity Lifecycle



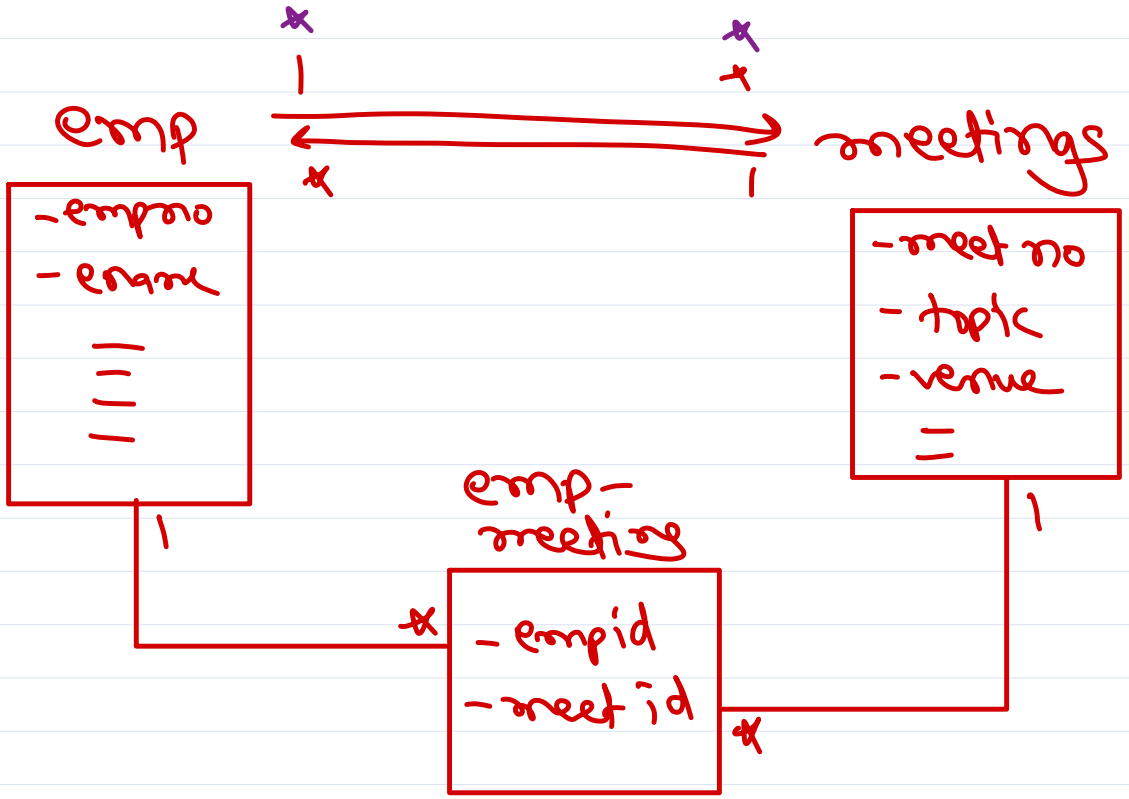
Hibernate – Entity life cycle

- 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.

Hibernate cascadeType.

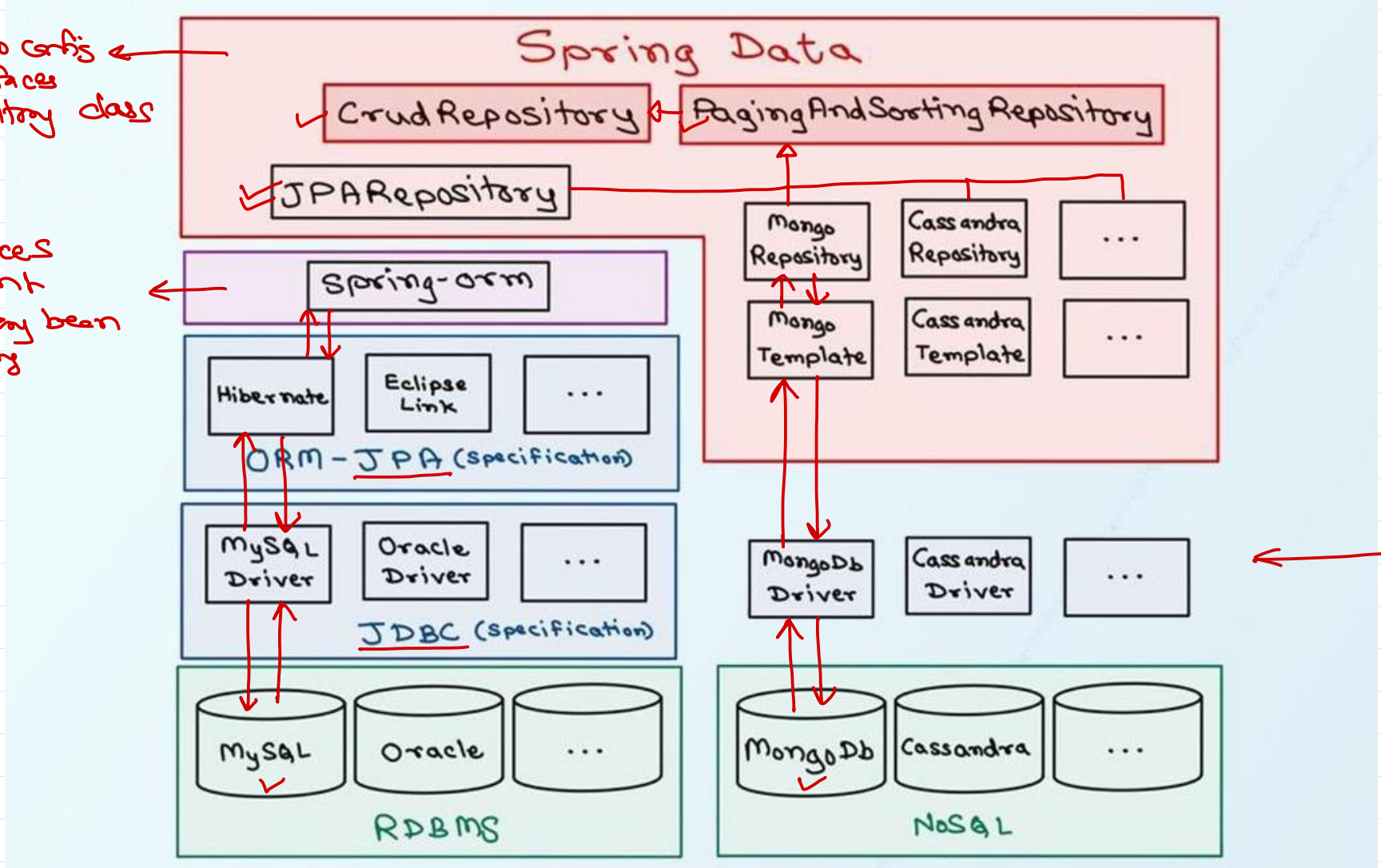


ManyToMany

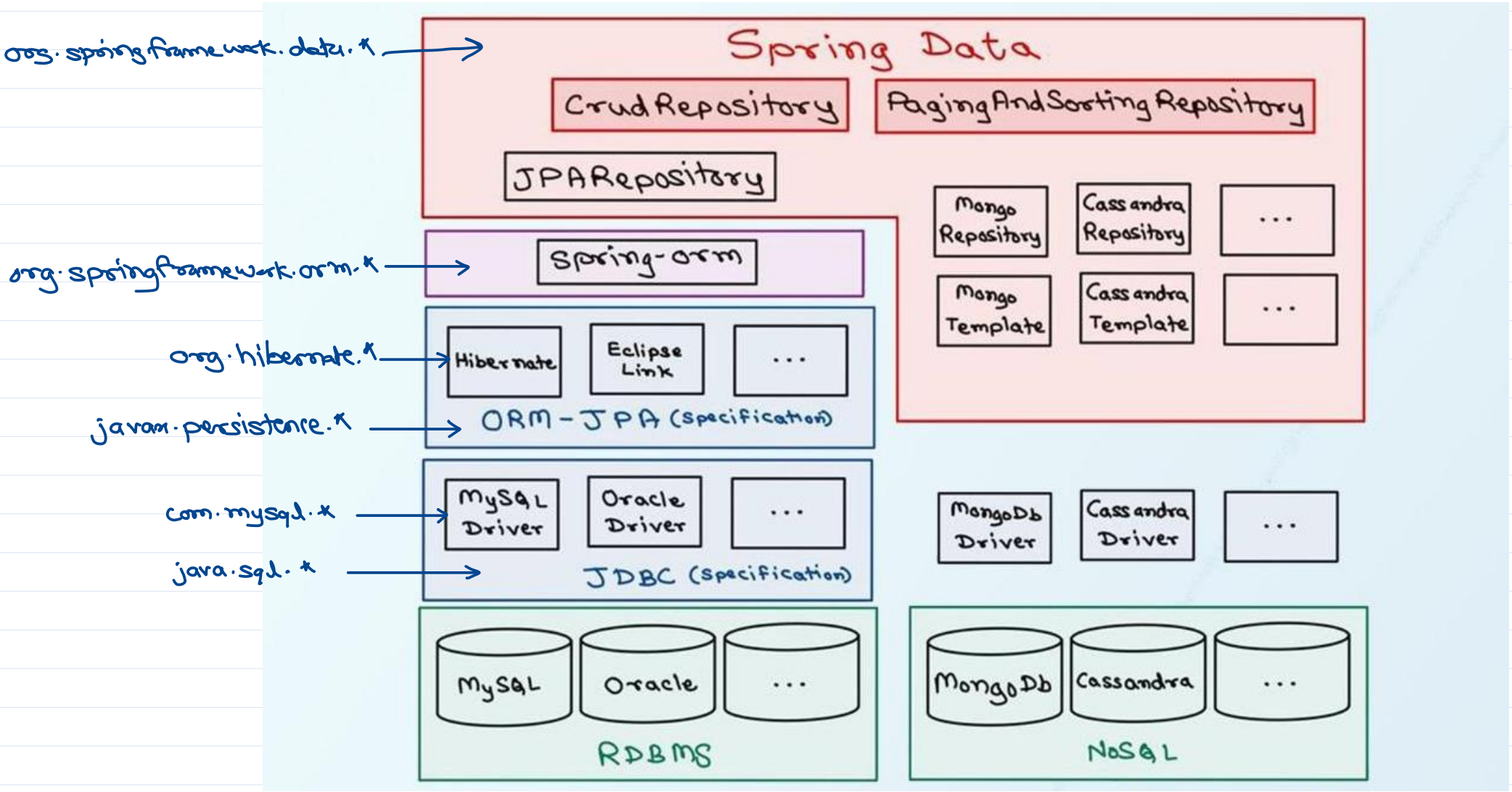



```
@Entity
class Emp {
  ...
  @ManyToMany
  @JoinTable(name="emp_meeting",
    joinColumns={@JoinCol("empid")},
    inverseJoinColumns={@JoinCol("meetid")})
  List<Meeting> meetList;
}
@Entity
class Meeting {
  ...
  @ManyToMany(mappedBy="meetList")
  List<Emp> empList;
}
```

Spring Data


Spring boot auto configs repository interfaces simple jpa repository class
data sources tx mgmt sessionfactory bean auto wiring
Java drivers for NoSQL db
Spring Data
CrudRepository
PagingAndSortingRepository
JPARepository
Mongo Repository
Cassandra Repository
...
Mongo Template
Cassandra Template
...
Spring-orm
Hibernate
Eclipse Link
...
ORM - JPA (specification)
MySQL Driver
Oracle Driver
...
JDBC (specification)
MongoDb Driver
Cassandra Driver
...
MySQL
Oracle
...
RDBMS
MongoDb
Cassandra
...
NoSQL

Spring Data


org.springframework.data.*
Spring Data
CrudRepository
PagingAndSortingRepository
JPARepository
Mongo Repository
Cassandra Repository
...
org.springframework.orm.*
Spring-orm
Mongo Template
Cassandra Template
...
org.hibernate.*
Hibernate
Eclipse Link
...
javax.persistence.*
ORM - JPA (specification)
com.mysql.*
MySQL Driver
Oracle Driver
...
MongoDb Driver
Cassandra Driver
...
java.sql.*
JDBC (specification)
MySQL
Oracle
...
MongoDb
Cassandra
...
RDBMS
NoSQL

Deploy Spring Boot application in AWS cloud

Ubuntu - EC2 instance

1. create instance
2. Configure network: inbound/outbound port 8080
3. terminal> sudo apt install openjdk-11-jdk
4. install mysql
   terminal> sudo apt install mysql-server
5. modify appln properties file to give path/creds of thisit db.
6. create/copy appln jar to the machine.
   terminal> nohup java -jar app.jar &
   - linux cmd: no hangup (signal)
     - do not terminate child apps even if terminal is closed.

Spring Boot – Application Bootstraping

@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

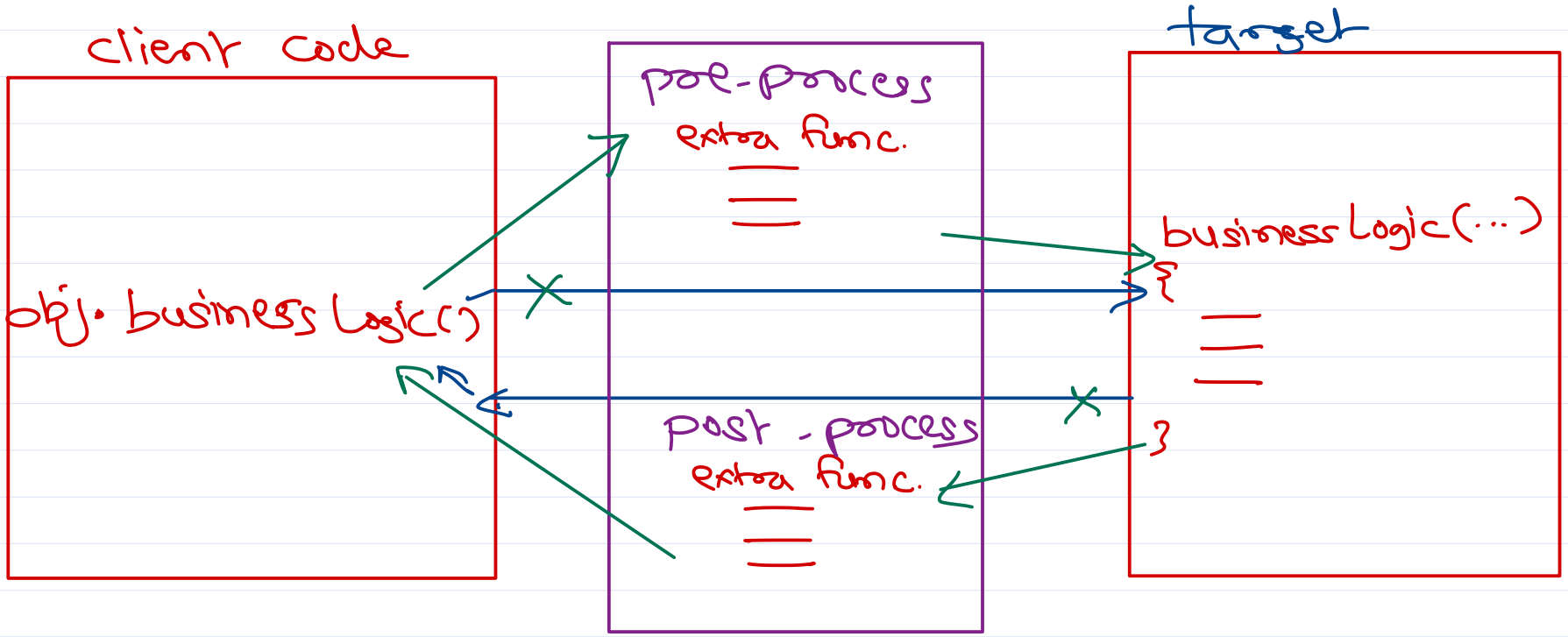
```
@SpringBootApplication
public class MyApp {
    public static void main(String[] args) {
        SpringApplication.run(MyApp.class, args);
    }
}
```

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

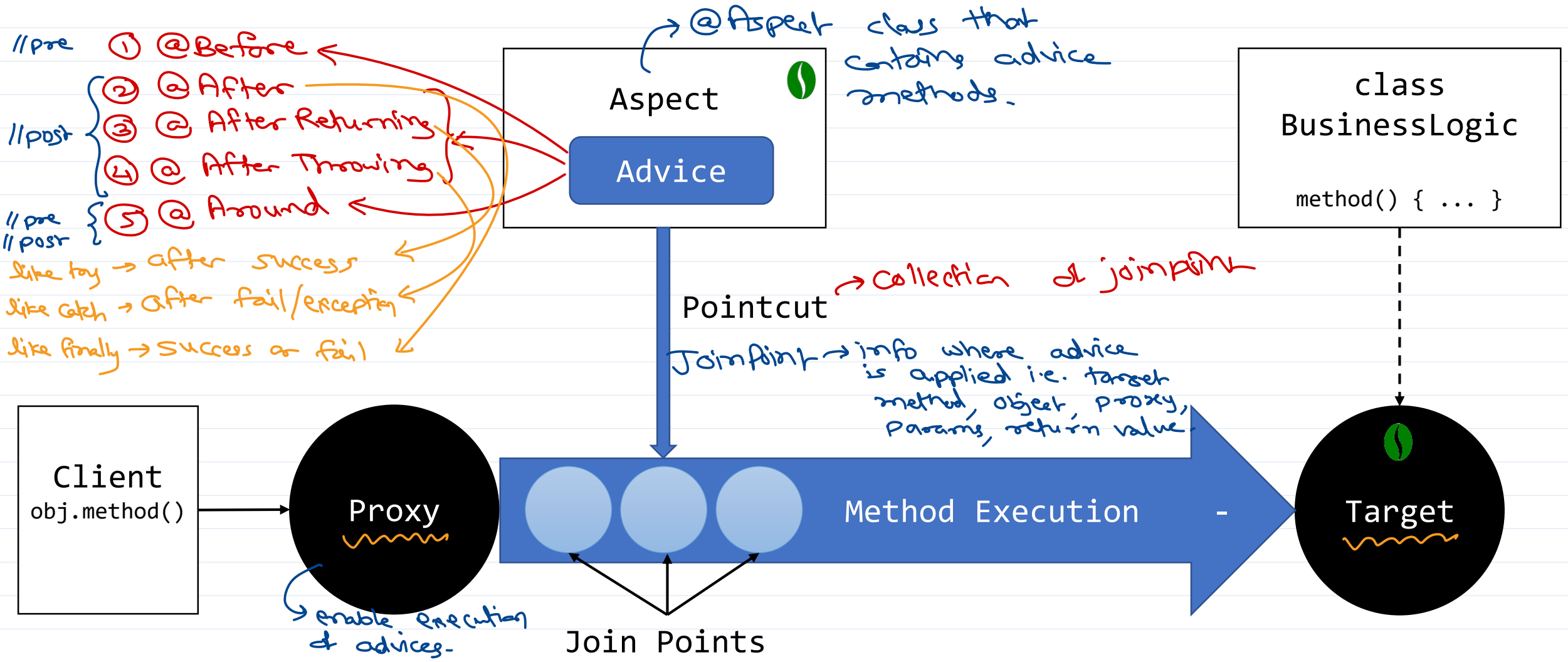
AOP in Java EE

1. Java EE filters

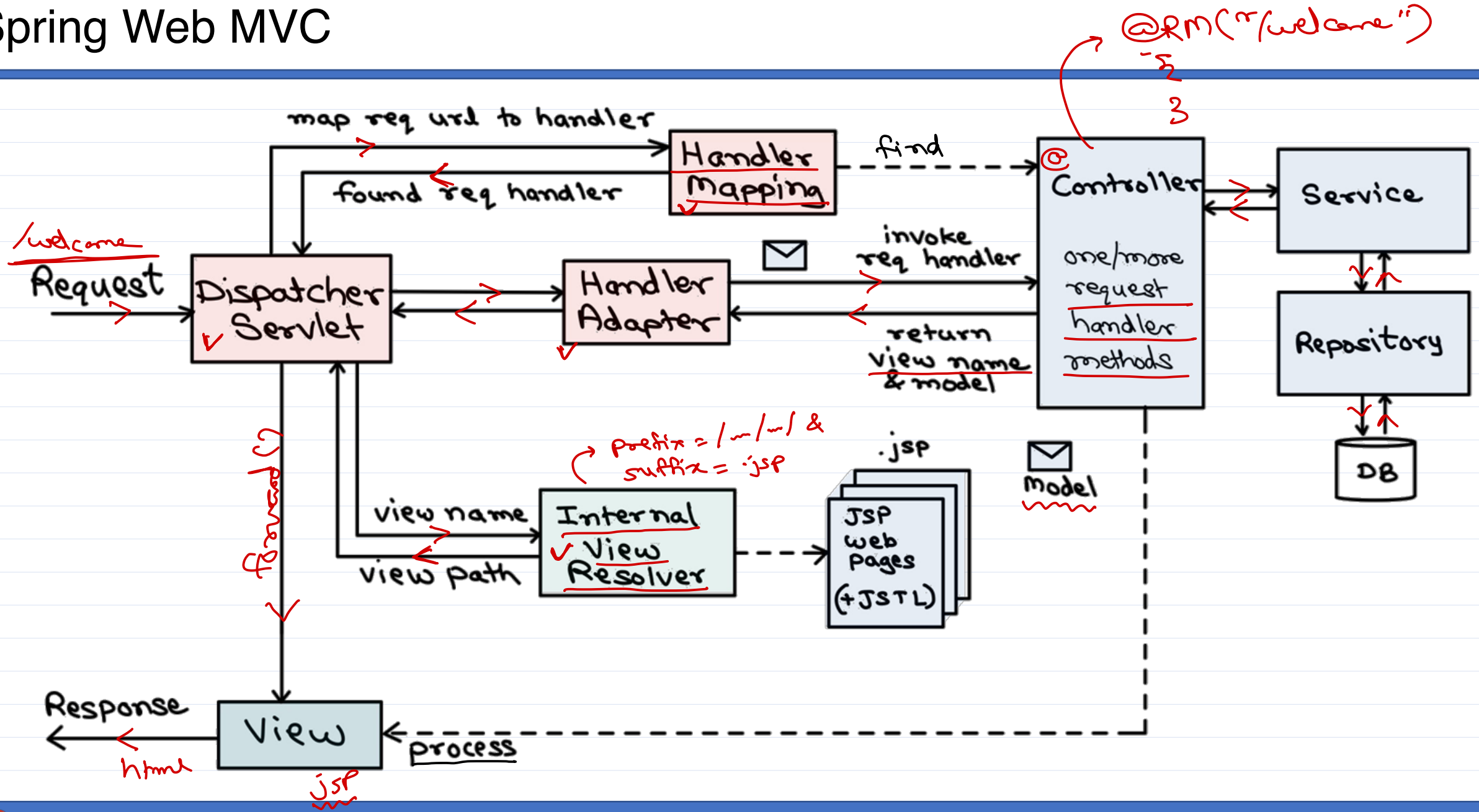
AOP in Spring

1. Spring-AOP (AspectJ)
2. Interceptors

Spring AOP


@Aspect class that contains advice methods.
Aspect
Advice
class BusinessLogic
method() { ... }
//pre
① @Before
//post
② @After
③ @ After Returning
④ @ After Throwing
//pre //post
⑤ @ Around
like try → after success
like catch → after fail/exception
like finally → success or fail
Pointcut → collection of joinpoints
Joinpoint → info where advice is applied i.e. target method, object, proxy, params, return value.
Client
obj.method()
Proxy
enable execution of advices.
Method Execution
Target
Join Points

Spring Web MVC

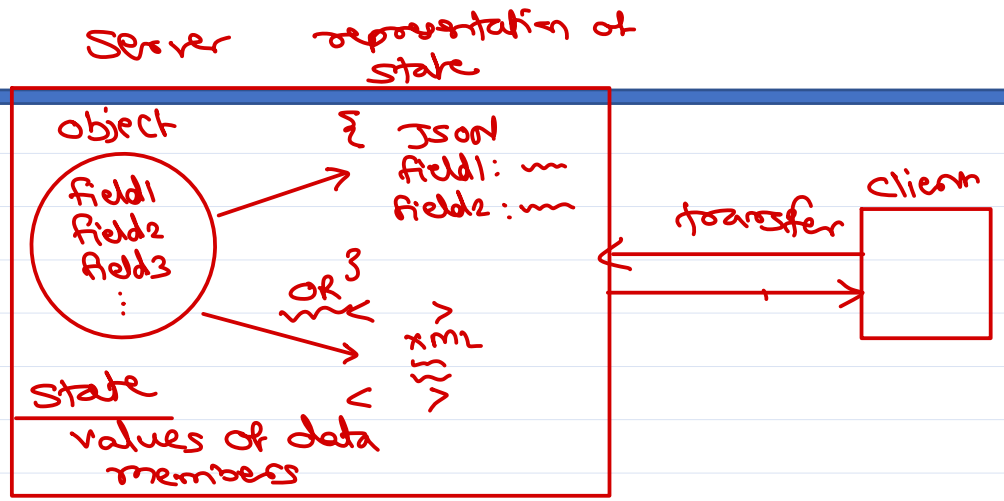


Spring Web MVC

- 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.

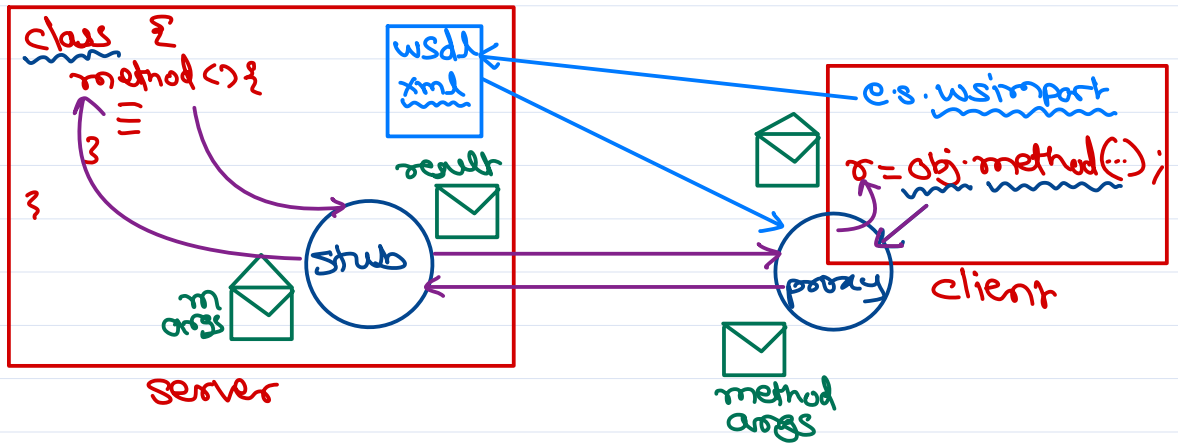
- https://terasolunaorg.github.io/guideline/1.0.x/en/Overview/SpringMVCOverview.html

REST services



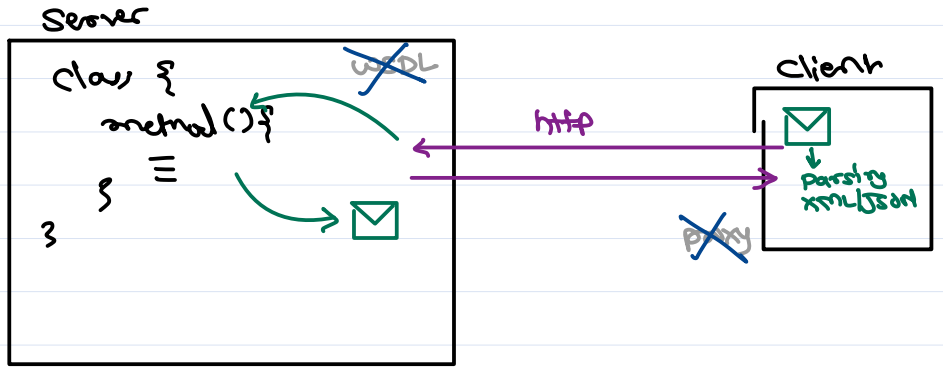
- REpresentation State Transfer (without UI)
- Protocol to invoke web services from any client.
  - Client can use any platform/language.
- REST works on top of HTTP protocol.
  - Can be accessed from any device which has internet connection.
  - REST is lightweight (than SOAP) – XML or JSON.
  - Uses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

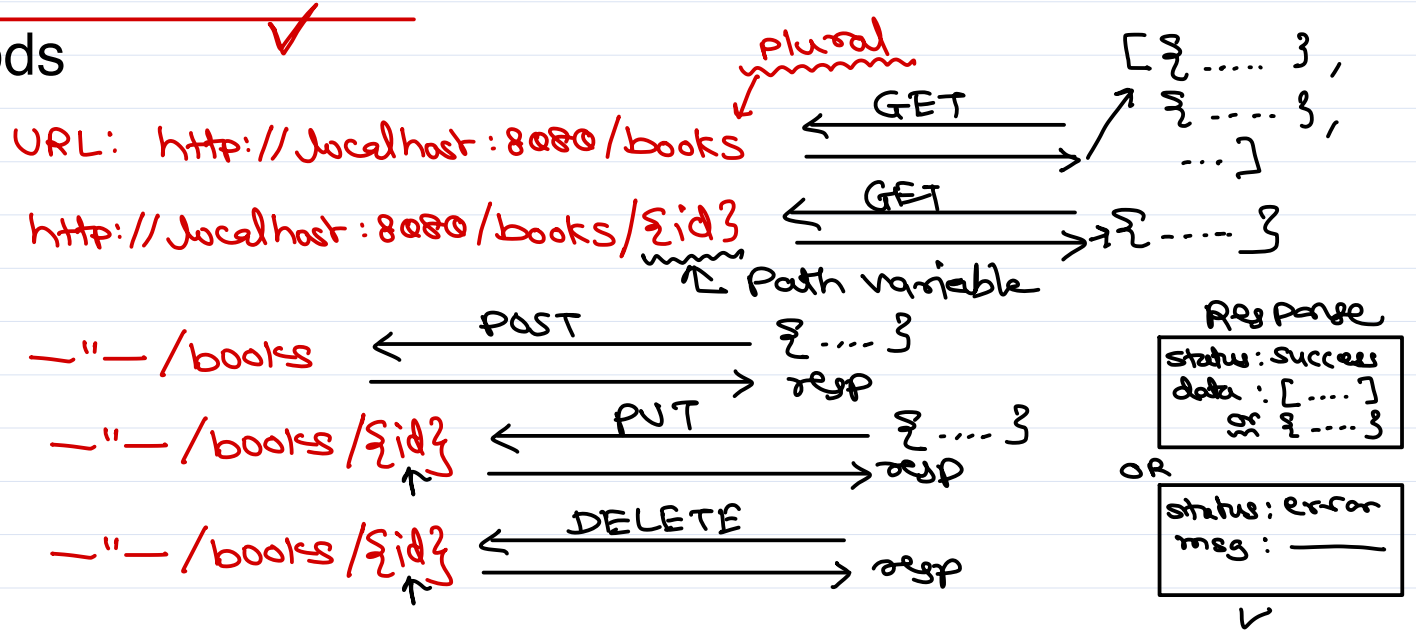
SOAP = HTTP + XML with format

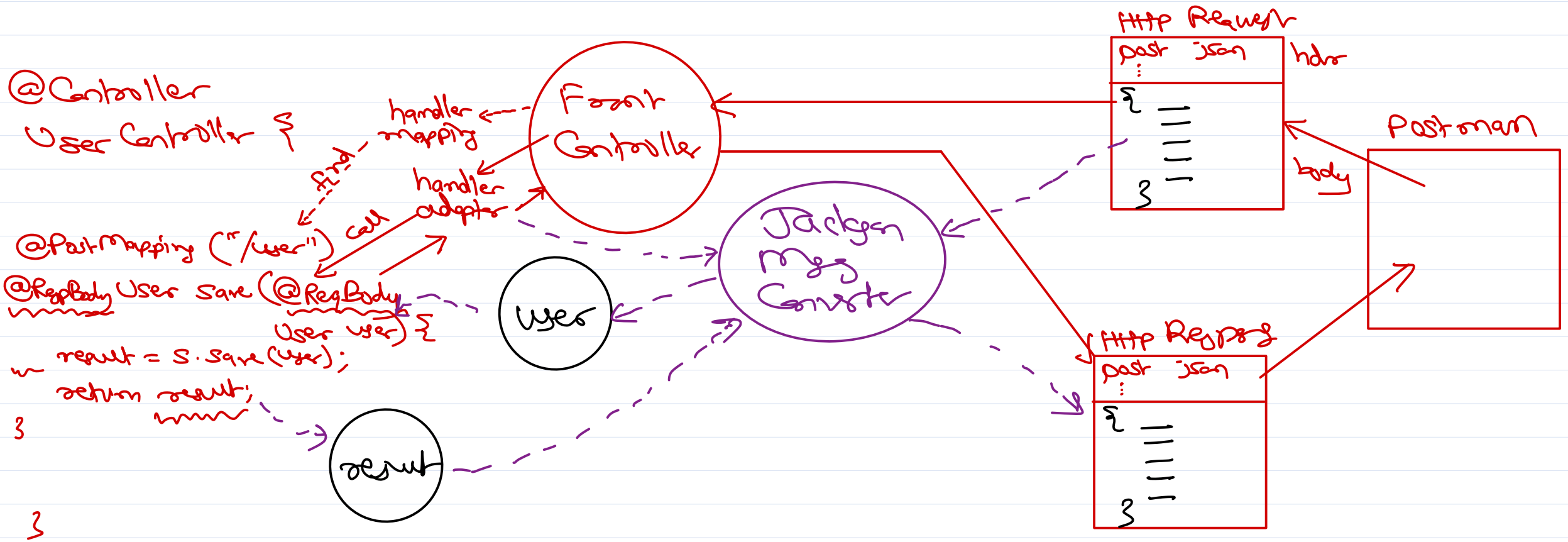


REST services

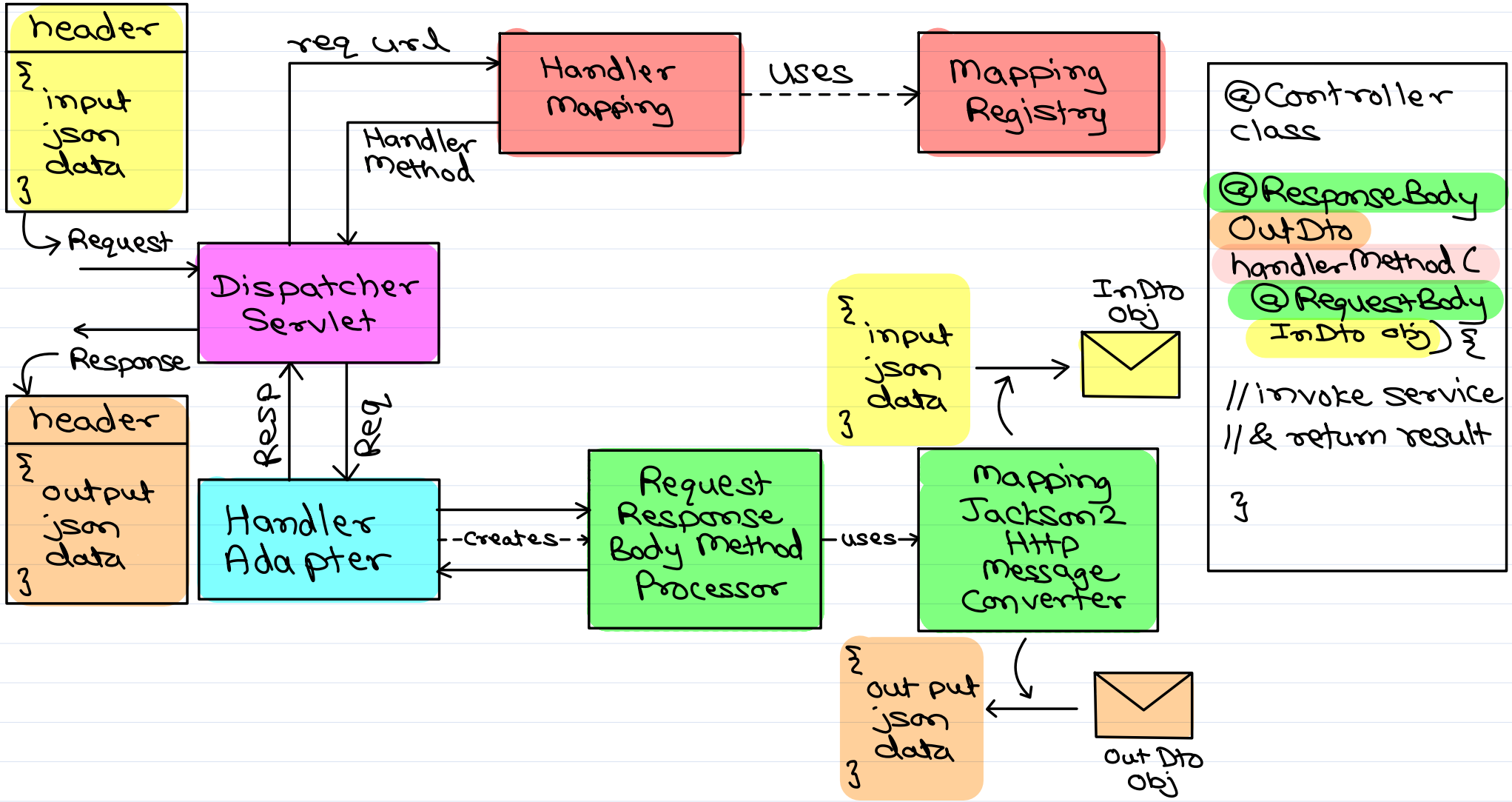
- REpresentation State Transfer
- Protocol to invoke web services from any client.
  - Client can use any platform/language.
- REST works on top of HTTP protocol.
  - Can be accessed from any device which has internet connection.
  - REST is lightweight (than SOAP) – XML or JSON.
  - Uses HTTP protocol request methods
    - GET: to get records
    - POST: to create new record
    - PUT: to update existing record
    - DELETE: to delete record
    - PATCH : to update data partially






@Controller
User Controller {
@PostMapping ("/user")
@ReqBody User save (@ReqBody User user) {
result = s.save(user);
return result;
}
}
find
handler mapping
handler adapter
call
Front Controller
Jackson msg Converter
User
result
HTTP Request
post json
{
}
hdr
body
Postman
HTTP Response
post json
{
}

REST services internals



```
@Controller
class MyClass {
  @ResponseBody User ---() {

  }
}
```

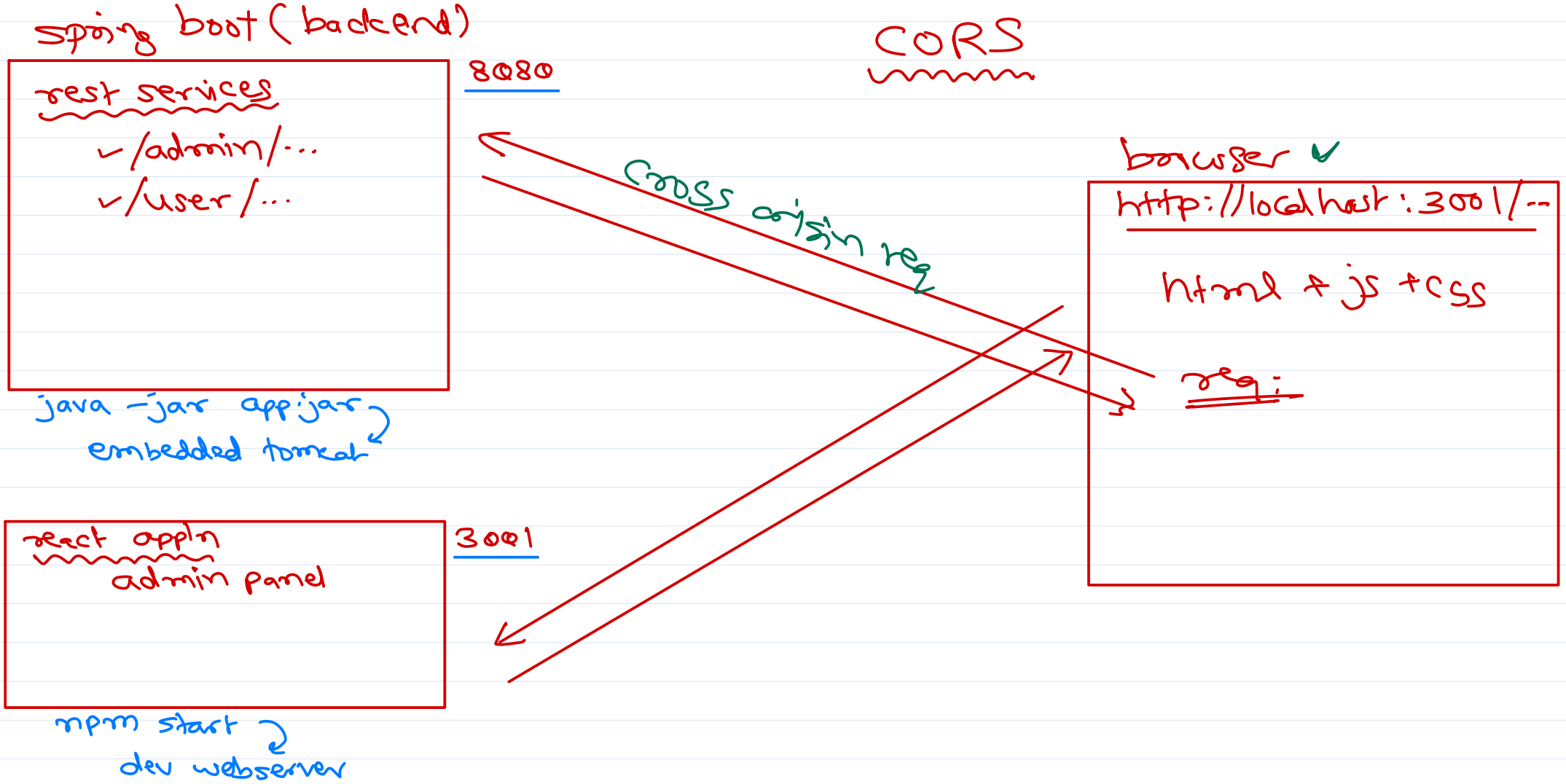
```
@RestController
class MyClass {
  @ResponseBody User ---() {

  }
}
```

```
@Controller/@RestController
class MyClass {
  ResponseEntity<User> ---() {

    return body(200) or
    StatusCode.

  }
}
```

URL

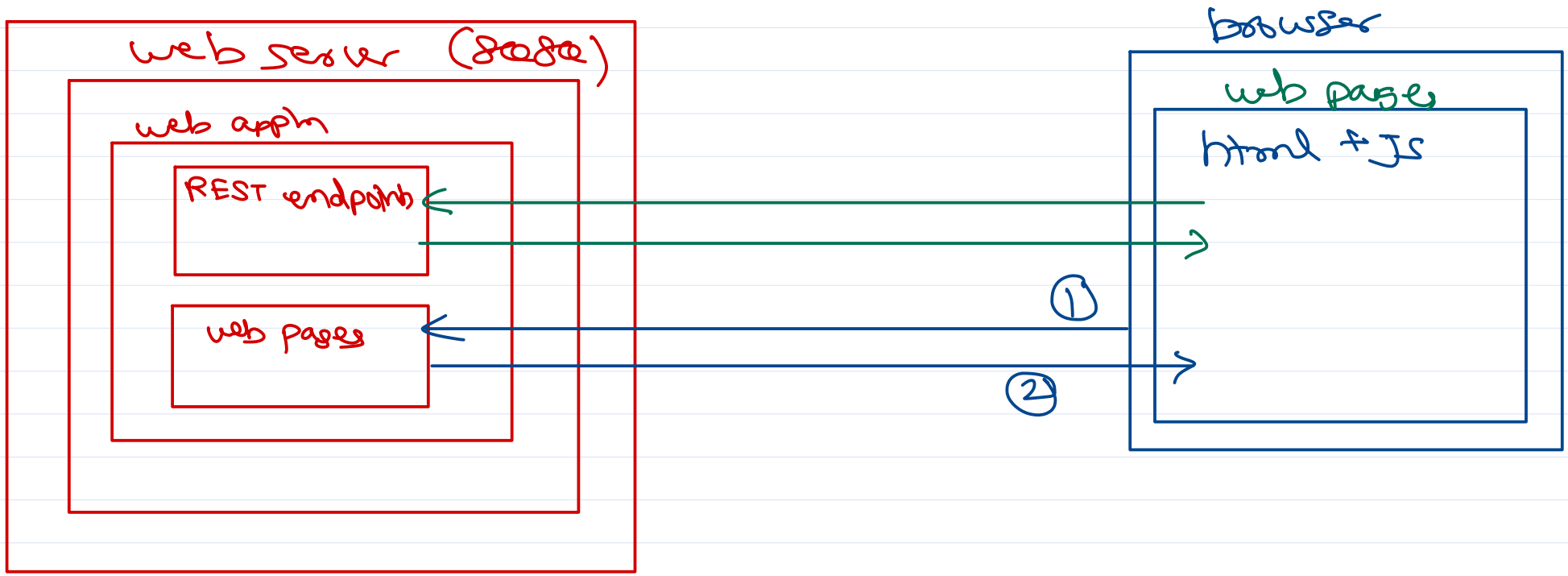
uri

http://localhost:8080/admin/books

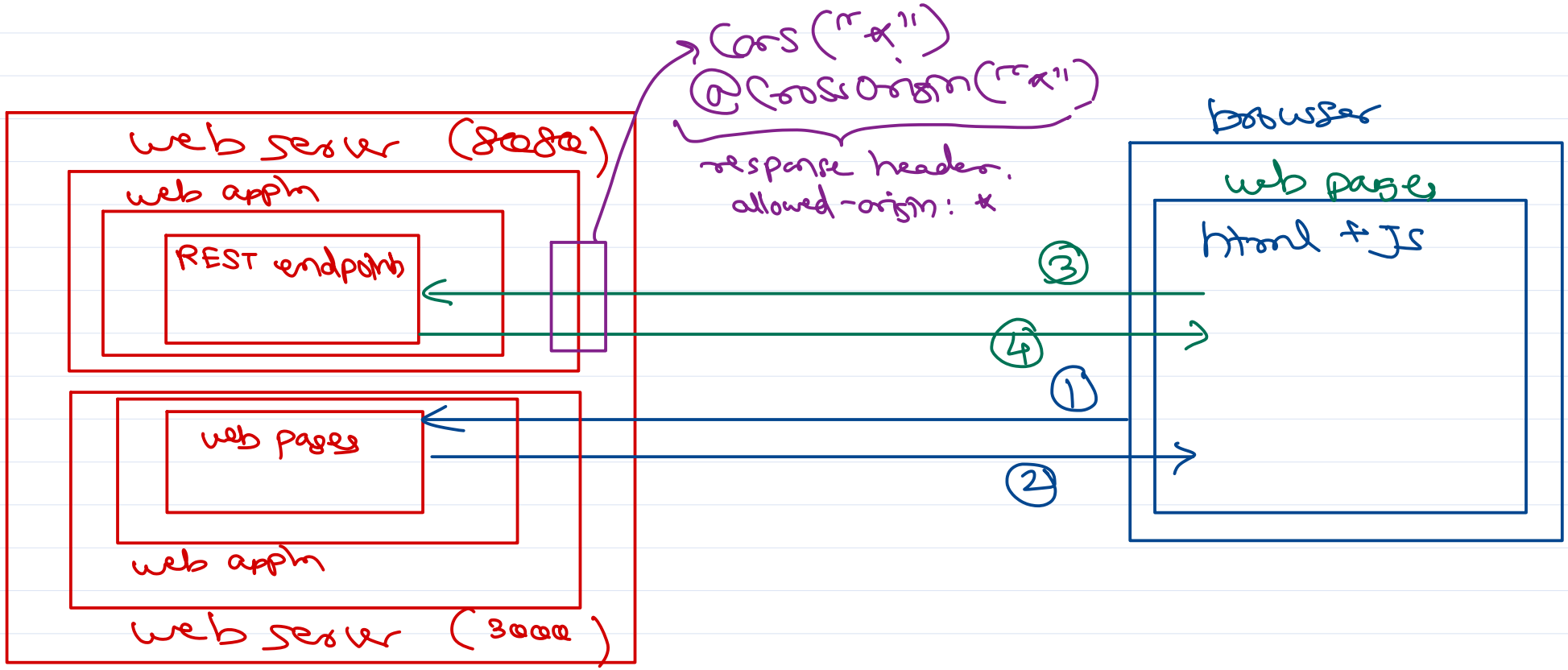

spring boot (backend)
rest services
/admin/...
/user/...
8080
java -jar app.jar
embedded tomcat
CORS
cross origin req
browser
http://localhost:3001/...
html + js + css
req:
react appln
admin panel
3001
npm start
dev webserver

CORS (not required)

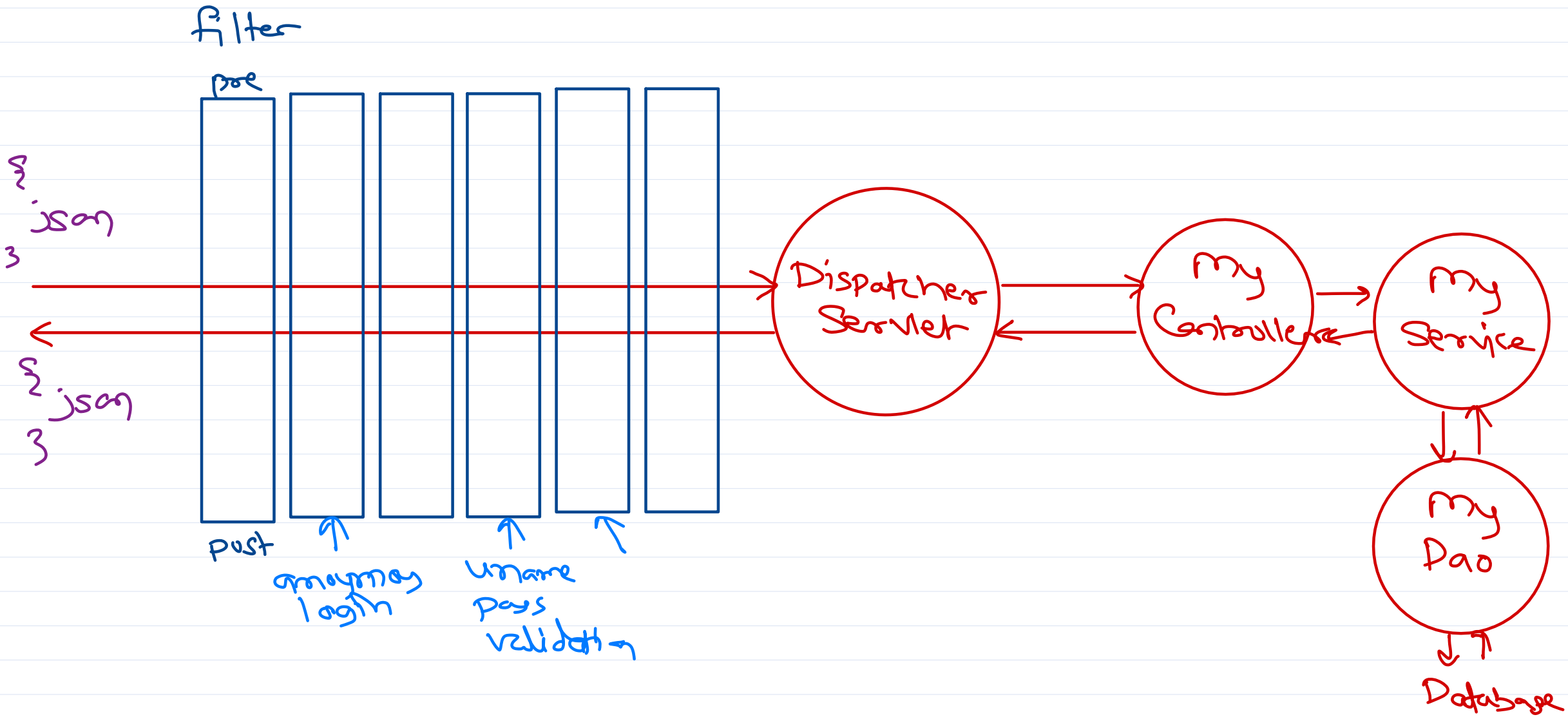
if front end & backend apps served from same web server.



CORS (required)



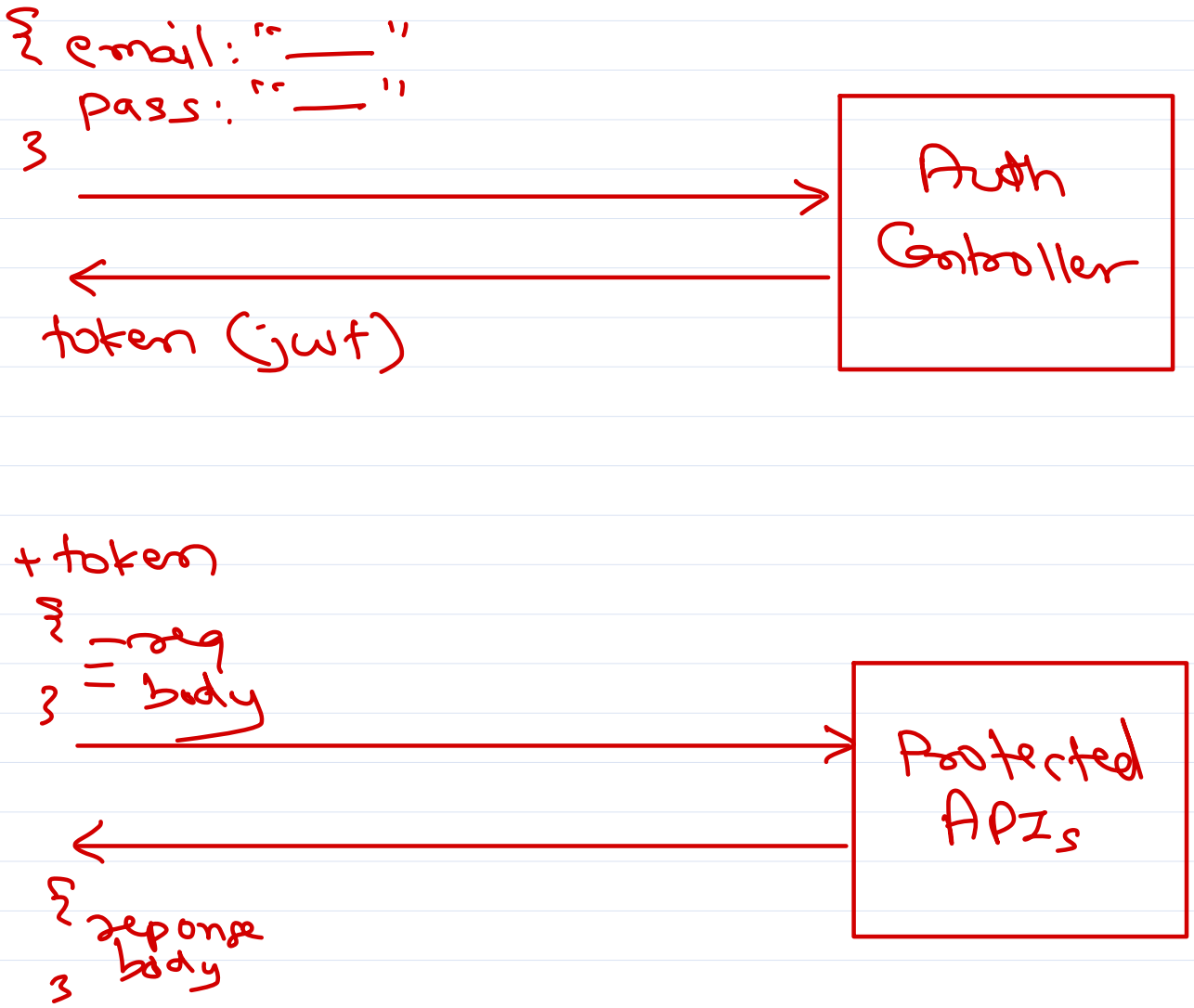
Spring security filter chain



Terminologies

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

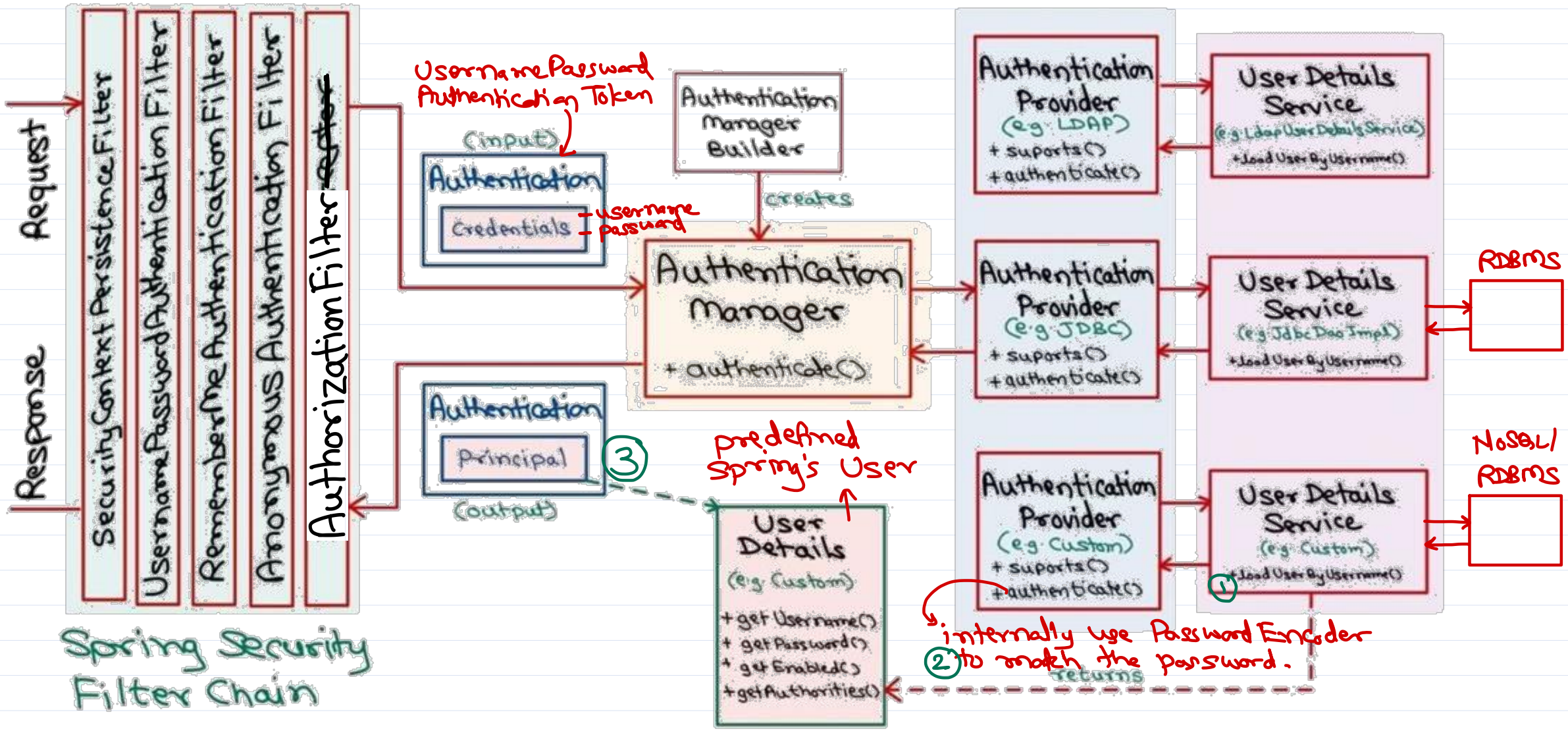


Why JWT ?

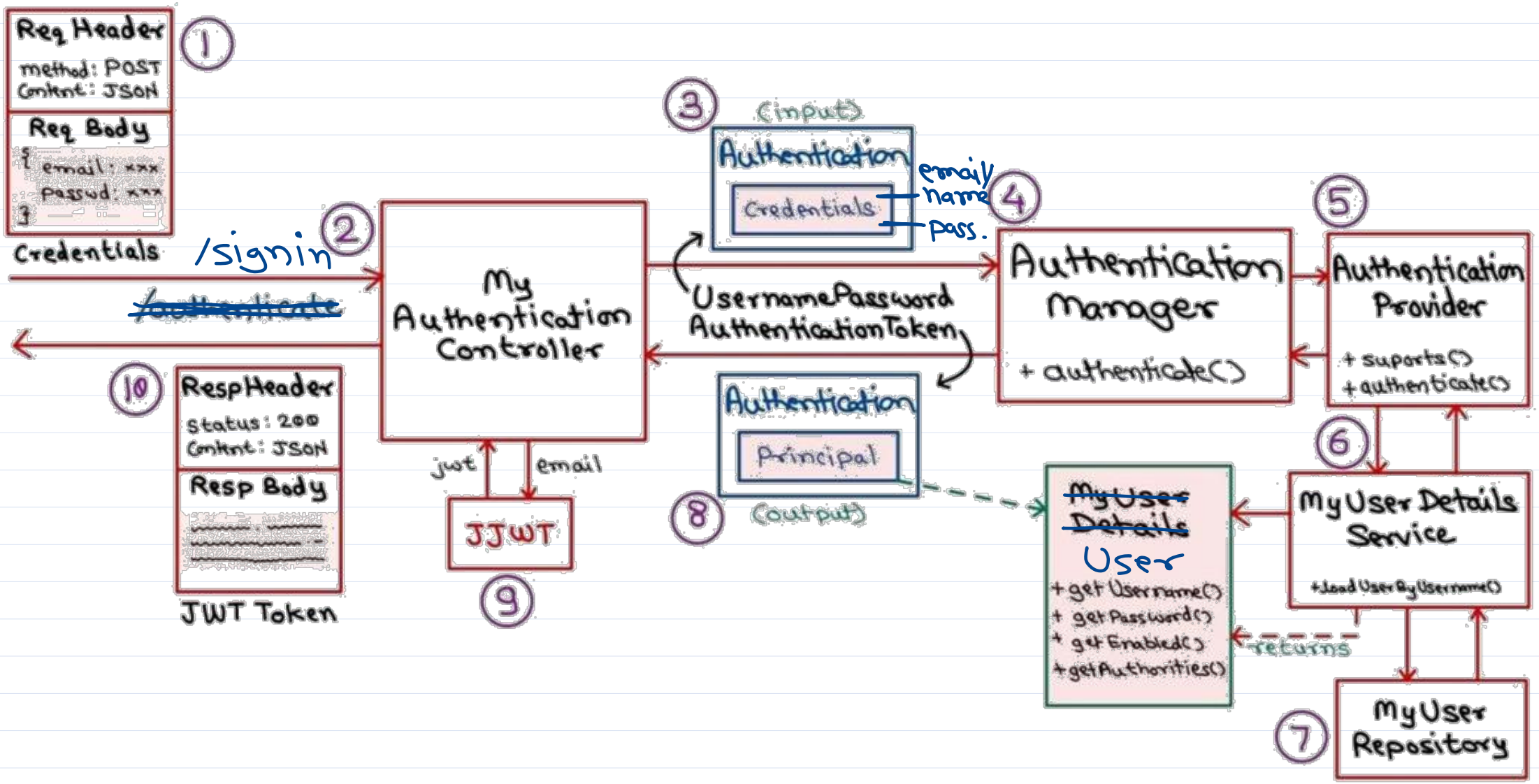
1. REST protocol is stateless. So each req must have something to indicate that user is valid.
2. JWT is tamper-proof. Because it is encrypted. It can be read by client, but cannot be modified by client.
3. Do not store any sensitive info in JWT token. Keep it in db.

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

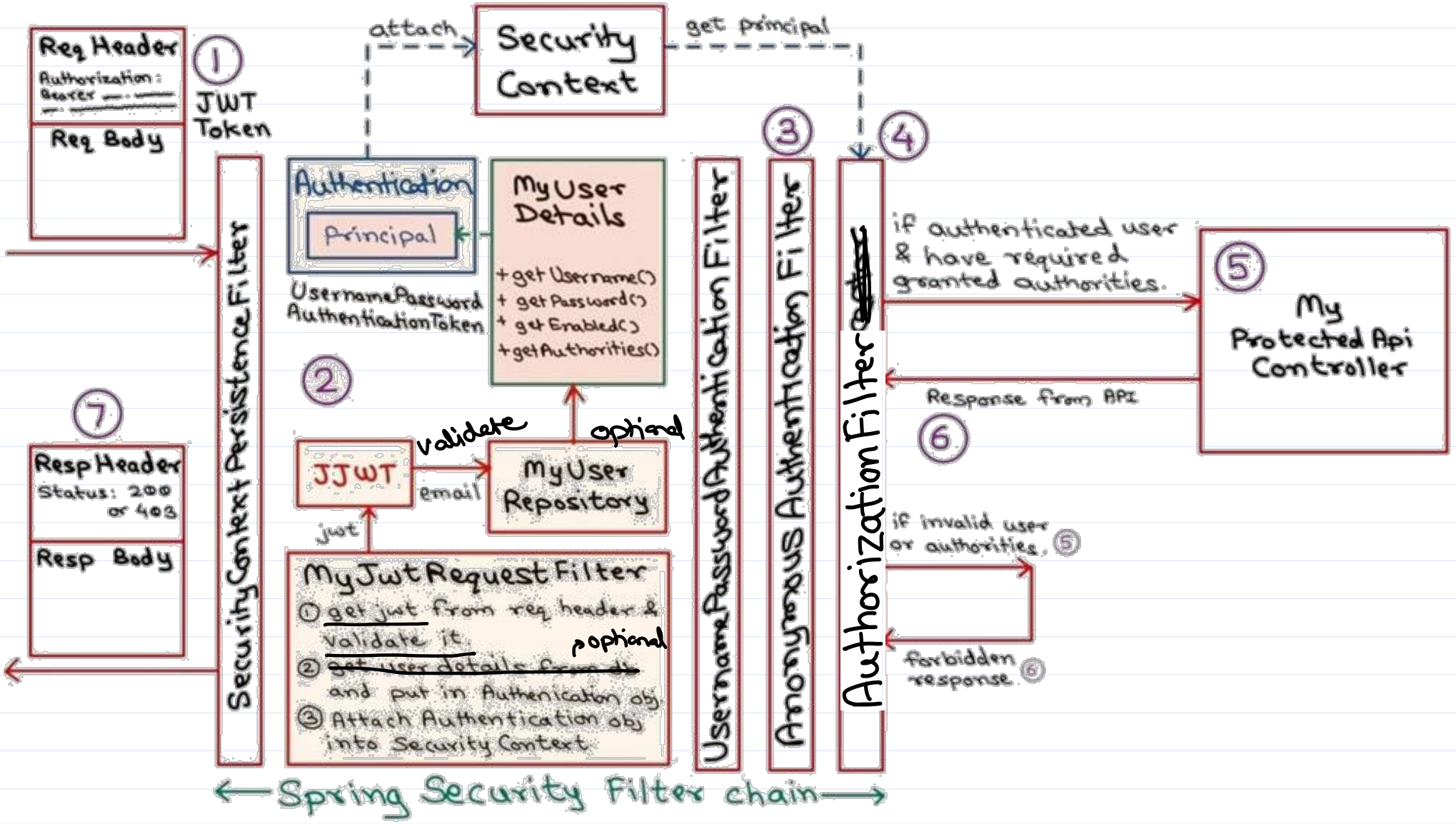
Spring Web Security



Spring REST Security (1)



Spring REST Security (2)



Thank you!

Nilesh Ghule <nilesh@sunbeaminfo.com>