# API – Key Code Structure:

Security Configuration class

@Configuration  
@EnableWebSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
@Autowired  
 private ApiKeyAuthFilter apiKeyAuthFilter;  
 @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 http  
 .csrf().disable() // Disabling CSRF protection  
 .addFilterBefore(apiKeyAuthFilter, UsernamePasswordAuthenticationFilter.class) // Add our custom filter  
 .authorizeRequests()  
 .anyRequest().authenticated(); // All requests must be authenticated  
 }  
}

Filter class to check in header if the api key is present or not

@Component  
public class ApiKeyAuthFilter extends OncePerRequestFilter {  
@Value("${api.key}")  
 private String apiKey;  
 @Value("${api.secret}")  
 private String apiSecret;  
 @Override  
 protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)  
 throws ServletException, IOException {  
   
 // Get the API key and secret from request headers  
 String requestApiKey = request.getHeader("X-API-KEY");  
 String requestApiSecret = request.getHeader("X-API-SECRET");  
 // Validate the key and secret  
 if (apiKey.equals(requestApiKey) && apiSecret.equals(requestApiSecret)) {  
 // Continue processing the request  
 filterChain.doFilter(request, response);  
 } else {  
 // Reject the request and send an unauthorized error  
 response.setStatus(HttpStatus.UNAUTHORIZED.value());  
 response.getWriter().write("Unauthorized");  
 }  
 }  
}

# Basic Authentication using Username and Password:

userDetailsService class to load the user from the DB

@Service  
public class UserDetail implements UserDetailsService {

@Autowired   
 UserRepository userRepo;

@Override  
 public UserDetails loadUserByUsername(String username)throws

UsernameNotFoundException

{  
 `User user = userRepo.findByUserNameOrEmail(username, username);  
 if(user==null){  
 throw new UsernameNotFoundException("User not exists by Username");  
 }  
   
 Set<GrantedAuthority> authorities = user.getRoles().stream()  
 .map((role) -> new

SimpleGrantedAuthority(role.getName()))  
 .collect(Collectors.toSet());

return new org.springframework.security.core.userdetails.User(username,user.getPassword(),authorities);  
 }  
}

Login API Controller :

@PostMapping("/login")  
 public ResponseEntity<String> authenticateUser(@RequestBody LoginDto loginDto) {  
 Authentication authentication = authenticationManager  
 .authenticate(new UsernamePasswordAuthenticationToken(loginDto.getUsername(), loginDto.getPassword())); SecurityContextHolder.getContext().setAuthentication(authentication);  
 return new ResponseEntity<>("User login successfully!...", HttpStatus.OK);  
 }

Config Class for this security mechanism

@Configuration  
public class SecurityConfig {

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

@Bean  
 public AuthenticationManager authenticationManager

(AuthenticationConfiguration configuration) throws Exception {  
 return configuration.getAuthenticationManager();  
 }

@Bean  
 SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http.csrf().disable()  
 .authorizeRequests()

.antMatchers("/api/\*\*", "/h2-console/\*\*")

.permitAll()  
 .anyRequest().authenticated();  
 http.headers().frameOptions().disable();

return http.build();  
 }  
}

# What is stateless and Stateful beans

* Stateless beans are those beans which have only one instances throughout the project
* Example of stateless beans are Singelton class.
* State full beans are those beans where every time we create a new instances.

Stateless beans: beans that are singleton and are initialized only once. The only state they have is a shared state. These beans are created while the ApplicationContext is being initialized. The SAME bean instance will be returned/injected during the lifetime of this

# Why we need @SpringBootApplications annotations :

Ans: this Annotation is a combination of 3 annotation

1. @EnableAutoConfiguration
2. @Configuration
3. @ComponentScan

Either we can use these 3 annotation or the main one directly @SpringBootApplication

The @SpringBootApplication annotation is equivalent to using @Configuration, @EnableAutoConfiguration, and @ComponentScan with their default attributes, as shown in the following example:

# How to remove circular dependency in Spring

Instead of using the field injection in the spring we can use constructor injection

So that the auto configuration of the bean will be resolved at the runtime only

When using constructor injection, the dependencies are resolved at the time of object creation, and circular dependencies are less likely to occur.

When the spring boot runs our project it will follow the step

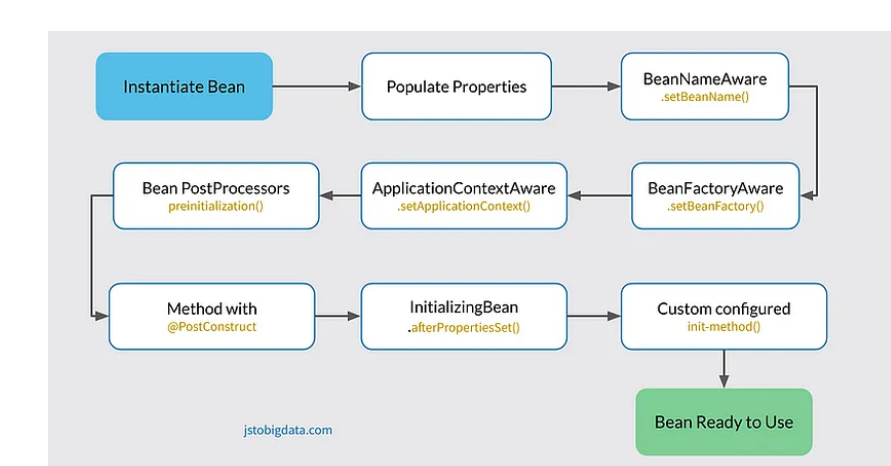
1. It will load all the classes and give the Spring Container which Beans we need
2. The constructor of the bean class is called
3. Dependency are injected into the bean including those annotated with @Auto wired
4. The @PostConstruct method (if present) is Invoked!!
5. WE can use constructor inject
6. Setter injection
7. Interface proxy

Interface proxy for the given code

introduce an interface for one of the classes involved in the circular dependency. Then, use Spring's proxy mechanism to create a proxy implementing the interface. This can help break the circular reference.

# Life Cycle of Spring Bean:

In the Spring Framework, the lifecycle of a bean refers to the series of steps that a bean goes through from its instantiation to its destruction. Spring provides a flexible and configurable way to manage the lifecycle of beans



First when we run the project then ioc container creates the bean of all the class then it will inject all the dependency

With the help of Init & destroy method we can start and destroy the life cycle of the beans

Locks & Reterant Locks in Java

Here we have locks to read and write the synchronized value when multithreading are working

So we have two interfaces for that ReadWriteLock and WriteLock() method in lock class

**public** **class** **SynchronizedHashMapWithReadWriteLock** {

Map<String,String> syncHashMap = **new** **HashMap**<>();

**ReadWriteLock** lock = **new** **ReentrantReadWriteLock**(); // ...

**Lock** writeLock = lock.writeLock();

**public** **void** **put**(String key, String value) {

**try** {

writeLock.lock();

syncHashMap.put(key, value);

}

**finally** {

writeLock.unlock();

}

} ...

**public** String **remove**(String key){

**try** {

writeLock.lock();

**return** syncHashMap.remove(key);

}

**finally** {

writeLock.unlock();

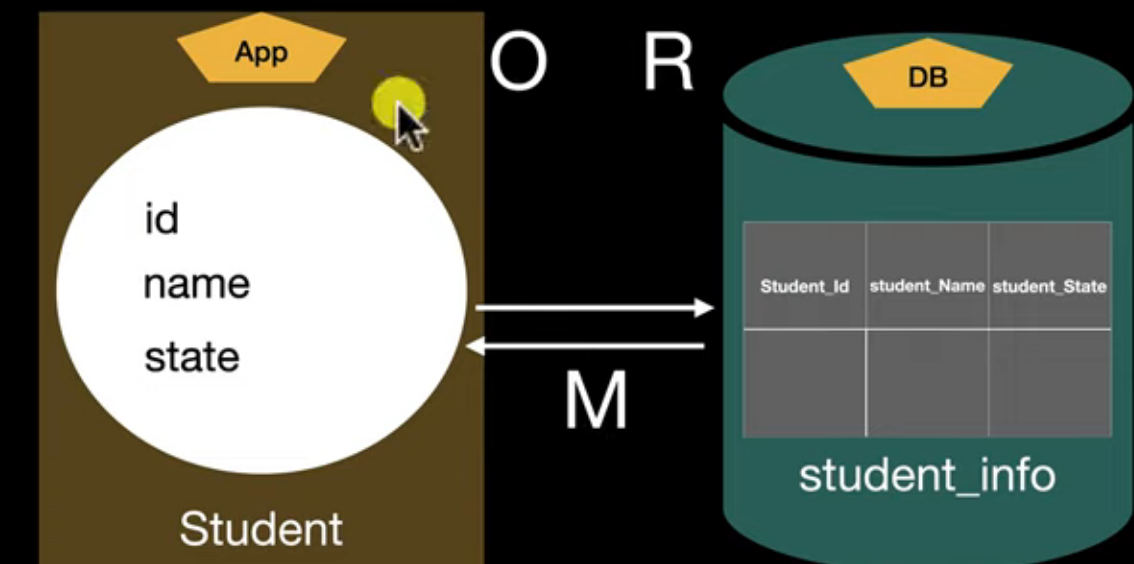
}

} //...

}

# Hibernate:

Copy from the object properties to data into the database table Is done by ORM framework and that one Is Hibernate



ORM Creates as bridge from our backend project to Databases

We have Different ORM framework :

1. Hibernate
2. TopLink
3. Mybatis
4. EclipseLink

# JPA – Java persistence API

Jpa is implementing by the ORM framework so if we are changing the implementing nature of

ORM framework then we need to implement the JPA persistence framework and JPA directly contact the ORM framework we just need to bother about writing the business logic !!

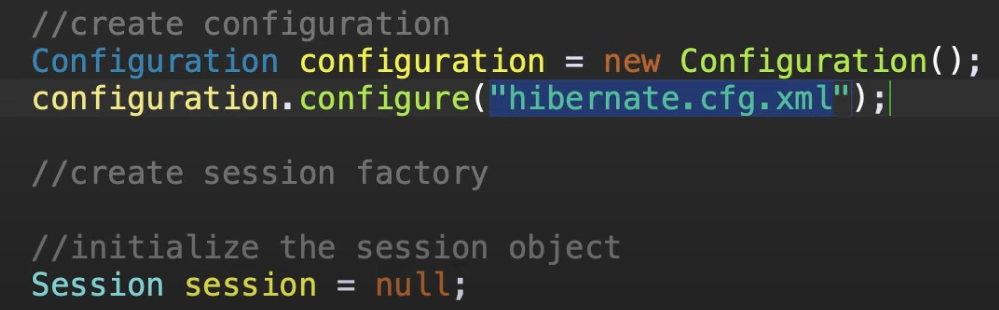
Steps to connect With the hibernate sessions :

1. Create Session Factory :

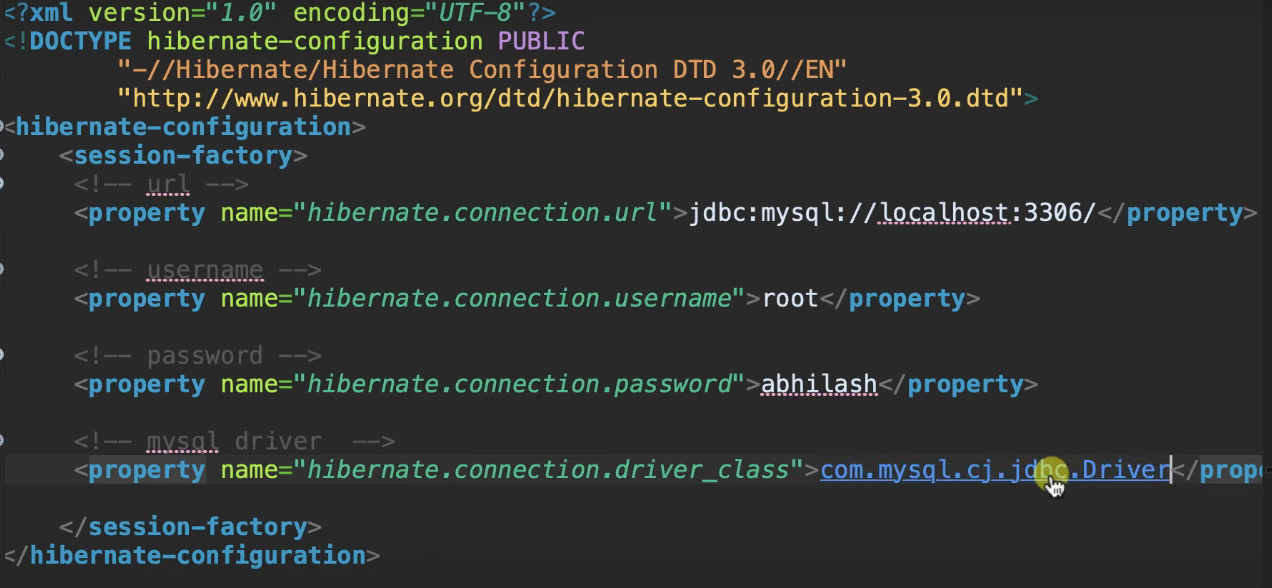
This will give us the datasource which will give the connectivity to the db

SessionFactory factory = new Configure(“hibernate.cfg.xml”).configure().getSessionFactory();

Session session = factory.openSession() || factory.getSession();



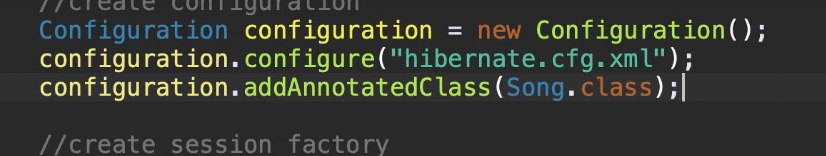
Environment from –> Org.hibernate.cfg package



This is the config file for hibernate

The above defined environment var will load all the property of the hibernate from this config file via matching the name of the properties !!

We need to tell the configuration about the annotated entity class to push the object into the DB

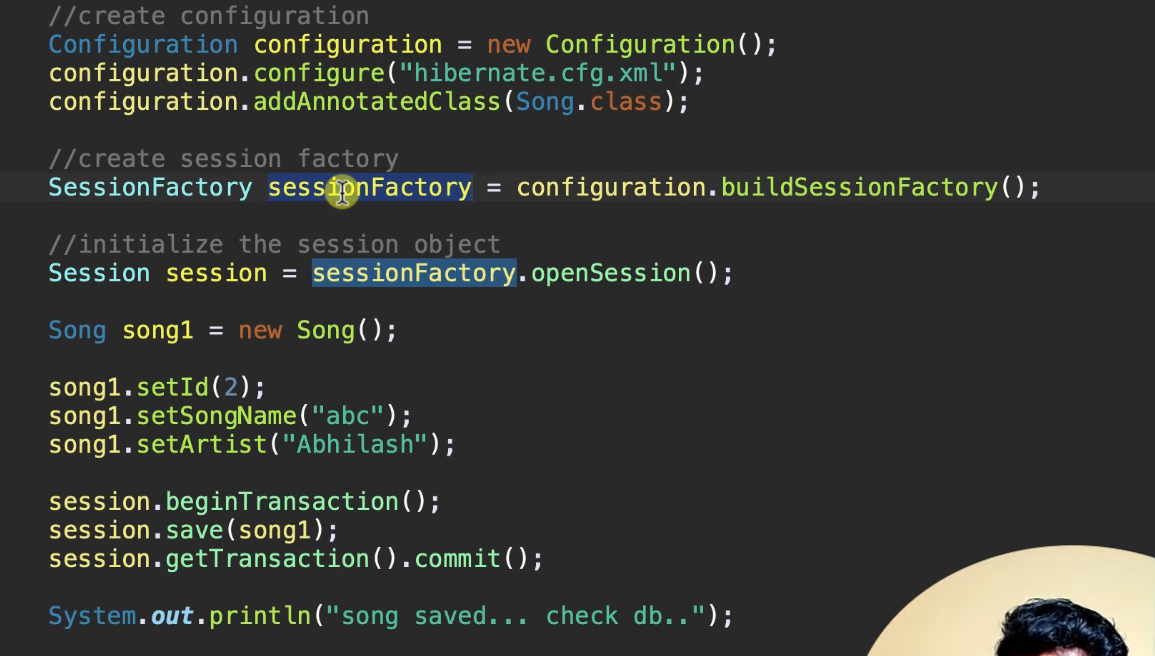


If we need to insert the object into the db

We need transaction to beginTransaction and

Commit the value into that txn

Internally SessionFactory using ConcurrentHashMap & Setting all the properties for acquire the valid property to save and create the session



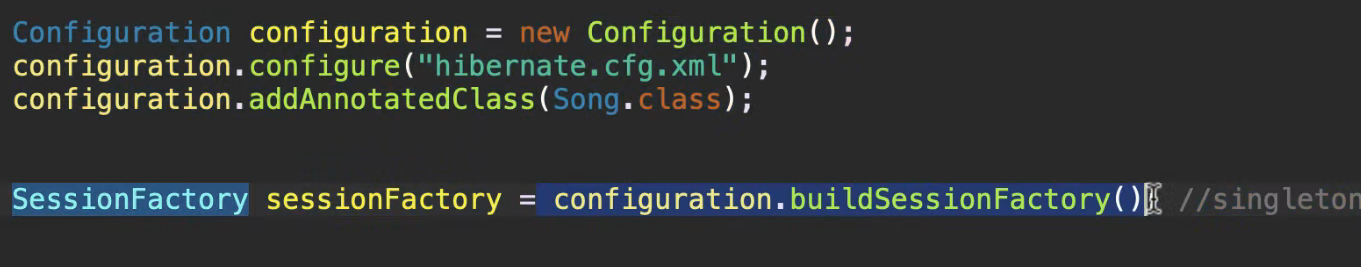
# Session Object:

We can have as many session as we want

# 4 Must Know Hibernate operations [#CRUD](https://www.youtube.com/hashtag/crud) | Design A singleton SessionFactory| Java Hibernate Framework

Sessionfactory is heavy object and its immutable in nature

Session factory

One way to configure SessionFactory is   


This is not recommended by default!!

Internally SessionFactory is Immutable and its heavy weight object so we need to make this session factory must be singleton pattern

Create a class of Singelton design pattern to get the object from the session factory and

Each time you will get the one instance each time !!

# Primary Key Generation

# Covariant Return type in java:

class Animal {

Animal get() {

System.out.println("Animal");

return this;

}

}

class Dog extends Animal {

@Override

Dog get() {

System.out.println("Dog");

return this;

}

void bark() {

System.out.println("Barking");

}

}

Before Java 5, if you wanted to override a method in a subclass, the return type in the subclass had to be the same as the return type in the superclass. With covariant return types, this restriction is relaxed, allowing you to return a more specific type in the subclass.

Give code explain the funcationality of covariant type in this

Animal class is the super class and have a method called get()

Which is returning the instance of that class

It's important to note that covariant return types only work for non-primitive types and are applicable only for the return type of the overridden method, not for its parameters.

Covariant return types can be useful when dealing with method chaining and providing more specific information about the type of object returned by a method. However, it's essential to use this feature judiciously, as it might lead to confusion if not used carefully.

# Dependency Ambiguity in Spring Beans:

If one interface implemented by 2 classes and if inject the interface directly to a class then

Then we get the error required a single bean, but 2 beans were found

Then we can use the @Qualifer() annotations to define which implementing class bean dependency we need !!

@Component(“impl1”)

Impl1 implements SingleBeans {}

@Component(“impl2”)

Impl2 implements SingleBean{}

Class TransactionService{  
  
@Autowired

@Qualifier(“Impl1”)  
private SingleBean bean;

# Without Using @Qualifier Resolve the bean depency:

Yes we can remove the bean dependency ambiguity using @Resource annotation

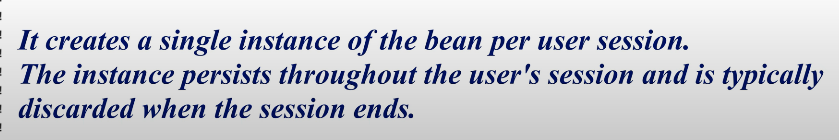
From java.jakarta!!

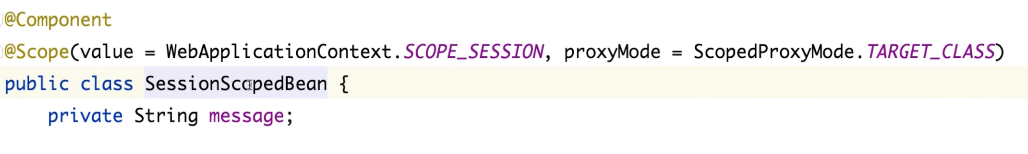
@Qualifier specific to spring library

& @Resource is given for any javaProject

# Explain Each Bean Scope:

Bean scope refer to life cycle and how long your bean will be live throughout the project

1. Singelton
2. Prototype -> Every time if you hit the request to that class then new bean will be created for that if we define the class with annotation @Scope(“prototype”)
3. Sesssion 



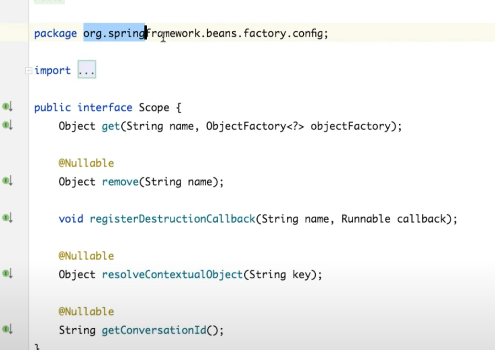
1. Request -> For each request from the web we will create one new bean every time for that particular request
2. Application
3. Websocket

3-5 bean scope is applicable for (Web based Applications !!)

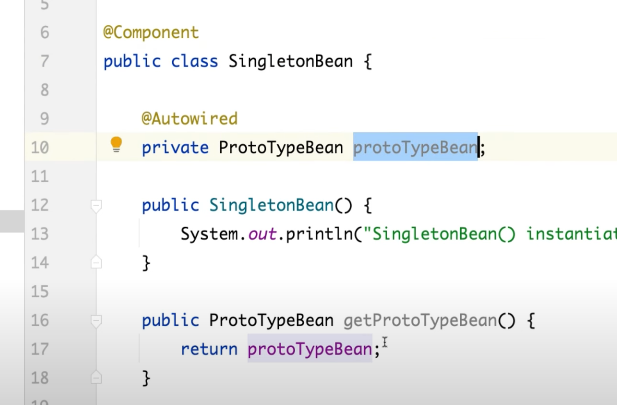
# Singelton:

By Default Every bean scope is singleton

# How To Define Custom Bean Scope:



We will create one custom scope class and implement the Scope interface and implement each of these methods as per requirement



In singleton bean if we define the prototype bean then it will loose it functionality and act like as Singelton Only

Solution for the above problem is we will Autowired the

@Autowired

ApplicationContext context;

Public ProtoTypeBean getProtoTypeBean(){

Return context.getBean(ProtoTypeBean.class);

}

But here we are getting the bean of prototype class manually !!

# @pathvariable VS @RequestParam

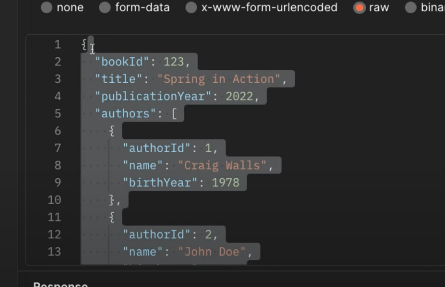
Pathvar is mandatory where requestParam we can make it optional

Public ResponseEntity<?> filterProducts(@RequestParam(required = false, value=”productType”, defaultValue=”Electronics”)){

}

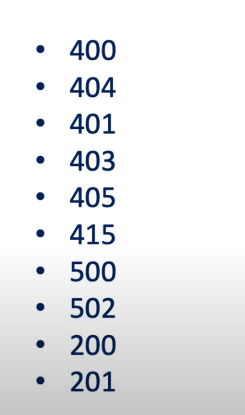


Json Payload :



We need to design the Class of BooK with all the field

# StatusCode:



400 bad request

404 resource is not available

401 Unauthorized

403 forbidden you are not authorize to access the resource

405 Method not allowed

415 unsupported media type

500 internal server error

502 Bad gateway

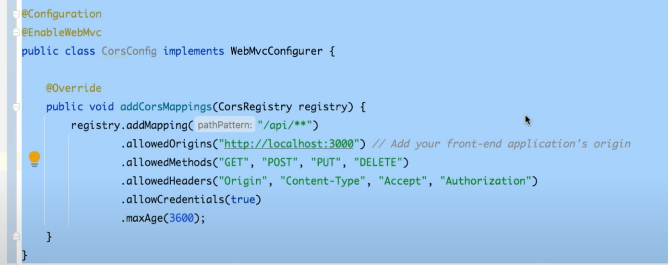
200 ok

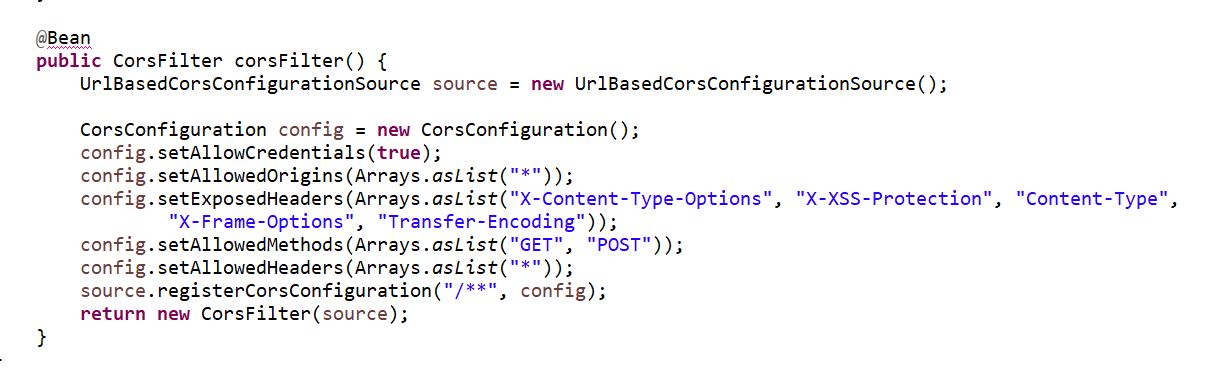
201 created resource

# Customize the status Code:

@ResponseStatus(HttpStatus.Created)

# CrossOrigin Config:





For Documentation we use the Swagger Open Api

And it will exposed all the API’s to the Outer world &

If we use the @Hidden Annotation then we will be able to hide that particular api from the documentations

# How to consume restful services In spring

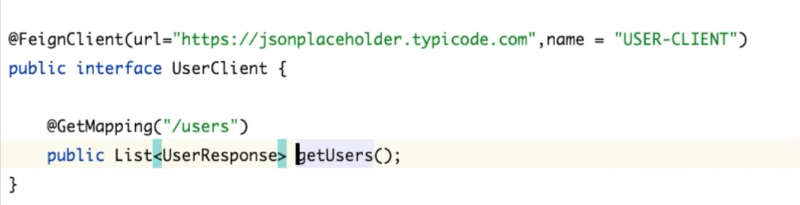
There are multiple ways to consume the rest Services:

1. Feign Client
2. Rest Template
3. Web Client
4. Advance Rest client

Rest template Example: 

Feign Client : Developed by Netflix we need to create the proxy of the service using Interface and we need to call the interface into our project to communication with other services

Example



Now we will autowired the UserClient userClient

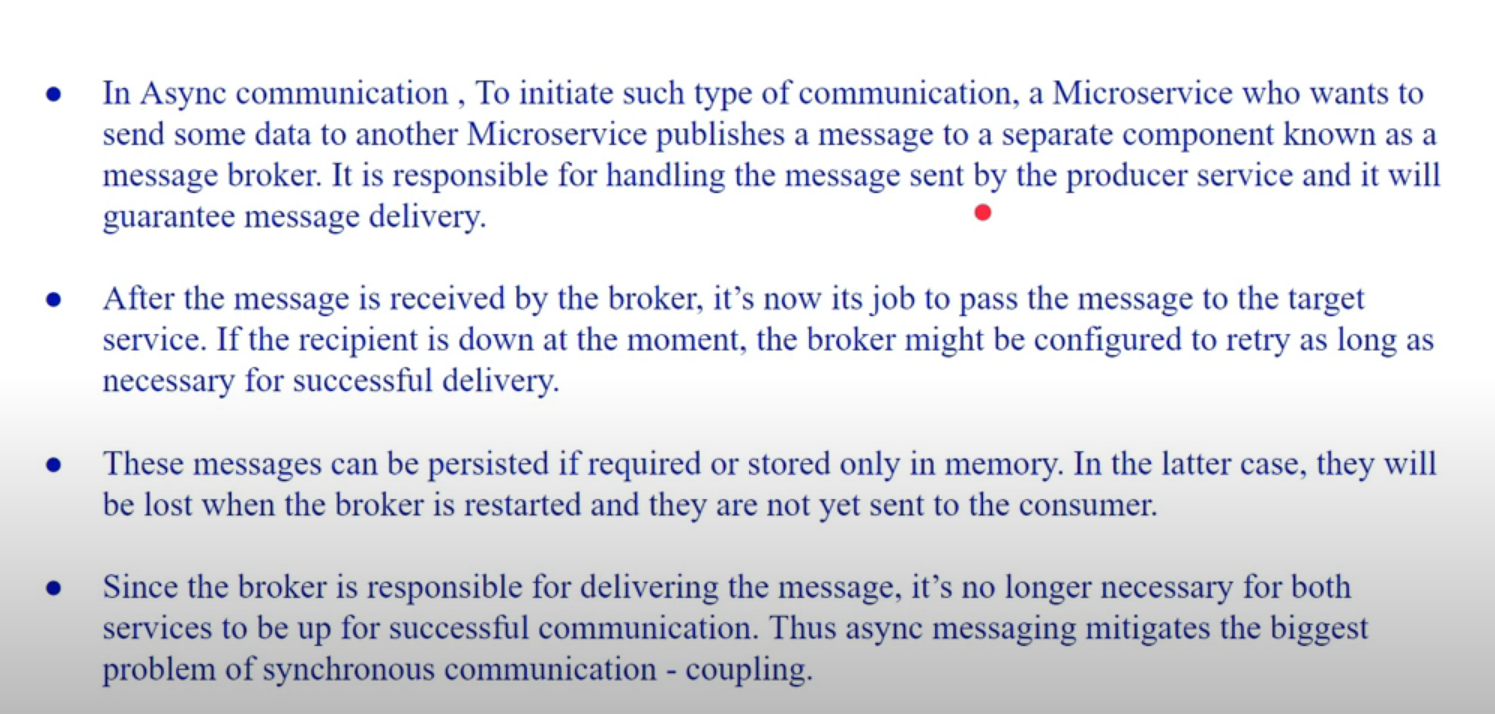
userClient.getUser();

# Ways to communicate In Microservices:

1. Synchronous communication

If we send the request then we will wait for the response (GRPC 10X faster than Rest Api)

# Asynchronous call



# What are the steps to connect an external database like MySQL or Oracle?

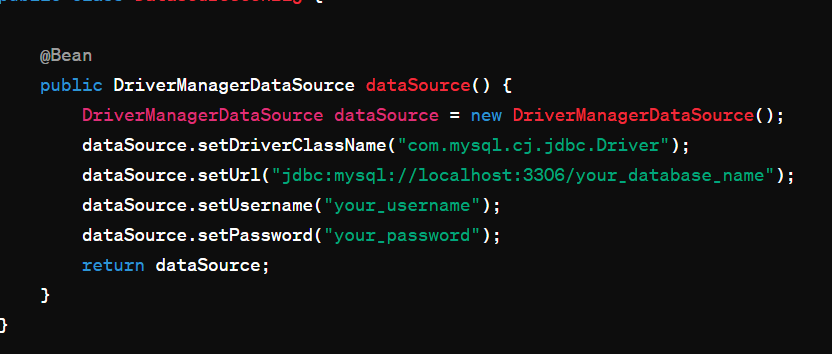
spring.jpa.hibernate.ddl-auto=none

spring.datasource.url=jdbc:mysql://localhost:3306/todo\_example

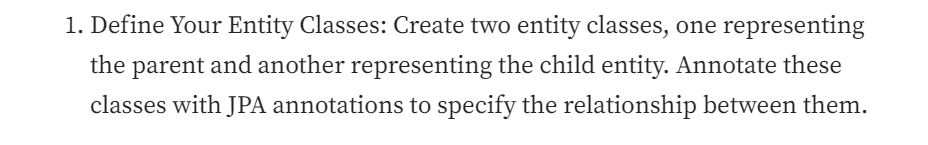
spring.datasource.username=todouser

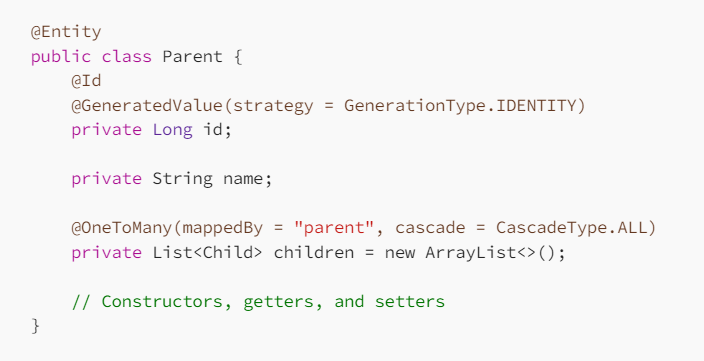
spring.datasource.password=YOUR\_PASSWORD

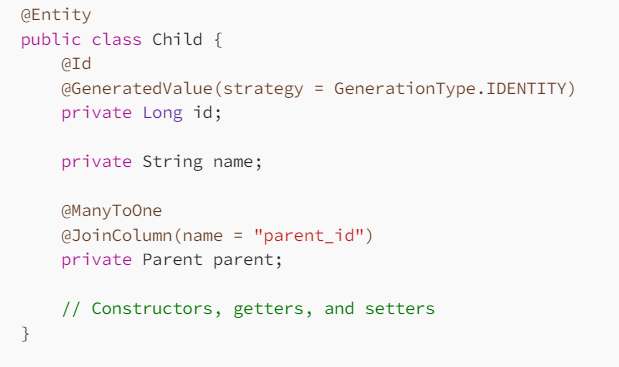
Bean Datasource Implementation:



# Mapping In Hibernate / JPA

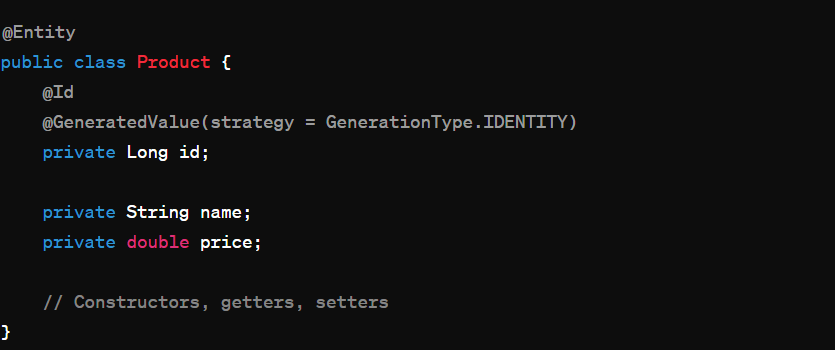


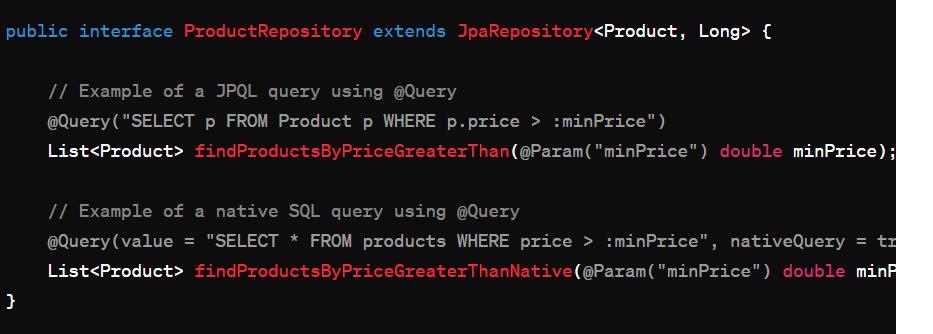




# **What is the purpose of @Query annotation in Spring Data JPA?**

It allows to specify the JPQL (Java Persistence query language) to query to the DB





1. First product findProductByPriceGreaterThan the give price it will give you that records in list<Product>
2. The second method, **findProductsByPriceGreaterThanNative**, uses a native SQL query. The **nativeQuery = true** attribute indicates that this is a native SQL query.

# **What is the purpose of the cascade attribute in Hibernate associations?**

1. All => This cascade type means that all operations (including persist, merge, remove, and refresh) should be cascaded from the parent entity to the associated entities.
2. Persist => Only the persist operation should be cascaded. When you persist the parent entity, the associated entities will be persisted as well.
3. Merge => Only the merge operation should be cascaded. When you merge the parent entity, the associated entities will be merged as well
4. Detach => Only the detach operation should be cascaded. When you detach the parent entity, the associated entities will be detached as well.
5. Remove => Only the remove operation should be cascaded. When you remove the parent entity, the associated entities will be removed as well.
6. Refresh => Only the refresh operation should be cascaded. When you refresh the parent entity, the associated entities will be refreshed as well.

# Shallow Copy VS Deep Copy

Shallow copy

# Abstract Vs Interface Java 8

Abstract is a keyword in java

We cannot create the instance of the abstract class

Sub class which extend the abstract class can give the implementation of the abstract method

# JWT Header and Architecture of JWT

JSON Web Token (JWT) is an open standard ([RFC 7519](https://datatracker.ietf.org/doc/html/rfc7519)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

A JWT is a string representing a *set of claims* as a JSON object. A claim is represented as a name/value pair consisting of a claim name and a claim value. A claim name is always a string, and a claim value can by any JSON value.

The structure of a JWT consists 3 parts separated by dots: header, payload and signature.

Header part of the JWT Token

{

"typ":"JWT",

"alg":"HS256"

}

Payload of the JWT Give us the information about each and everything

{

  "sub": "1234567890",

  "name": "Nam Ha Minh"

  "iss": "codejava.net",

}

The algorithm specified in the header is used to sign the header and payload. For example, if the algorithm is HS256 (HMAC SHA256), then the signature will be created in the following way:

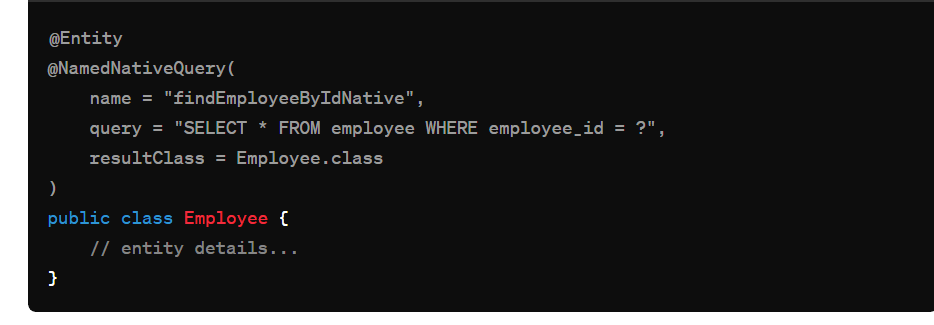
|  |  |
| --- | --- |
| 1  2  3  4  5 | HMAC\_SHA256(      base64urlEncode(header) + "." +      base64urlEncode(payload),      secret  ) |

base64urlEncode(header) . base64urlEncode(payload) . signature

# Which annotation/ configuration is required to enable the native SQL in JPA?

In JPA We can directly use the @NamedNativeQuery() annotation to enable the native query

From the given example we can understand the logic



Call of the query be like

List<Employees> employees = entityManager.createNativeQuery(“findEmployeeByIdNative”)

.setParameter(1,employeeId);

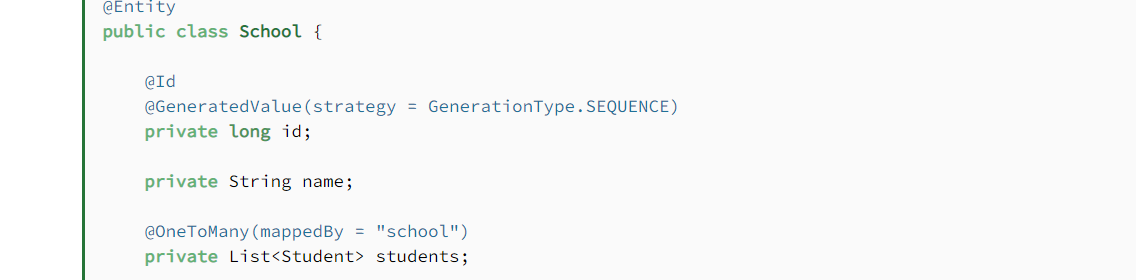
.getResultList();

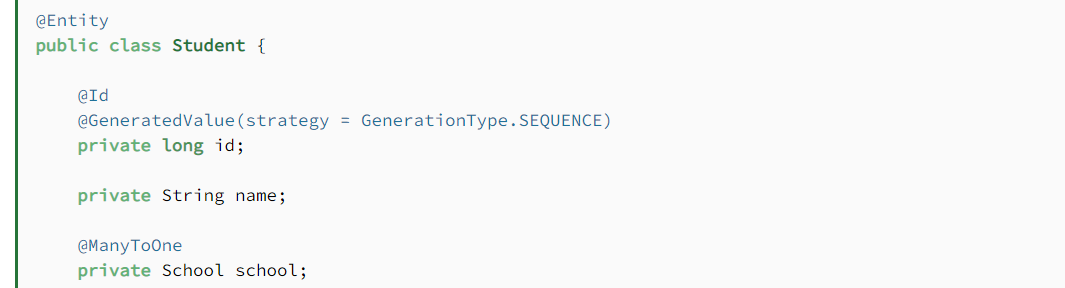
# Batch Update and Insert in JPA/Hibernate

When we run a insert or update program in JPA for inserting 100+ record then 100+ insert query

To insert into the db for running such query we need to create 100+ Java Object which is memory consumption task

And We will get One error “OutOfMemoryError”





@Transactional

Public void insertDataSeparately(){

For(int I = 0; I < 100; I++){

School school = createSchool(I);

entityManager.persist(school);

}

entityManager.flush();

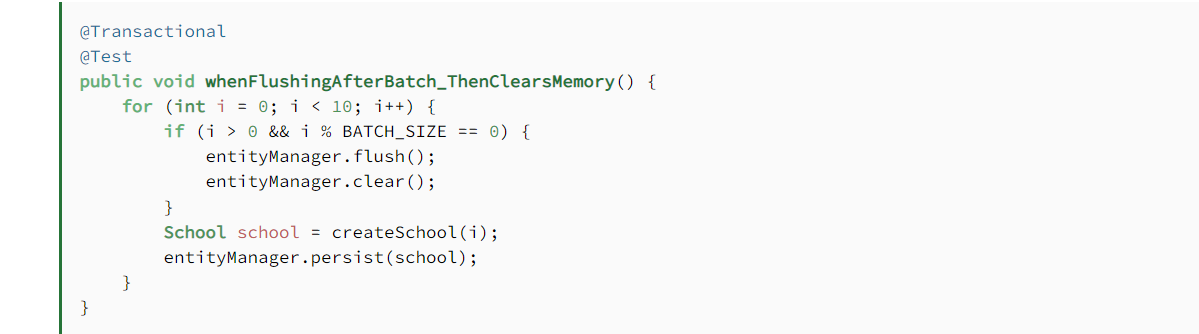
Here 100 time object creation happen and 100 query will run

NOW We will configure the batch in properties file

spring.jpa.properties.hibernate.jdbc.batch\_size=5

Now query will execute in the batch size of 5

So there will be 20 queries only ::



spring.jpa.properties.hibernate.order\_inserts=true

This will make sure that if we are inserting multiple Table then it will work run the query in the order!!

# Transactional In JPA/Hibernate

Here when we perform the crud operation then we wrap the whole Code into the transaction if we need to rollback or commit the data

1. Propogation Level

@Transactional(Propogation = propogation.Required)

**Timeout (timeout):**

1. Specifies the maximum time (in seconds) a transaction is allowed to execute.

If the transaction takes longer, it will be rolled back.

1. @Transactional(readOnly = true)

Public void myReadOnlyMethod(){}

This will make the method read only and save the efforts

1. Transaction annotation doesn’t roll back for checked exception but it will for unchecked exception or we define in the annotation for which exception transaction have to roll back !!

@Trasnaction(rollbackFor=Exception.class)

# Which annotation is used to handle the joins between multiple tables at the Entity class level?

@JoinTable is used to handle the joins between multiple table

If we have a class of student so many student can enrol into many course or one course

@ManyToMany Annotation we used

Class Student{

@ManyToMany

@JoinTable(

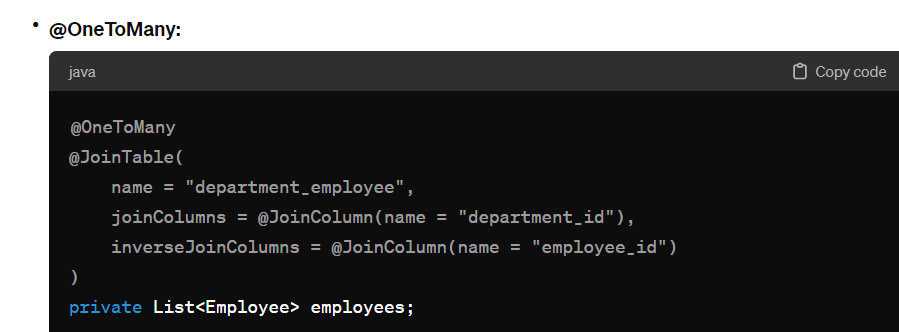
Name = “student\_course”

joinColumns = @JoinColumn(name = “student\_id”),

inversedJoinColumns= @JoinColumn(name =”course\_id”)

)

Set<Course> courses = new HashSet<>();



# How do handle unidirectional join and bidirectional join at the Entity level?

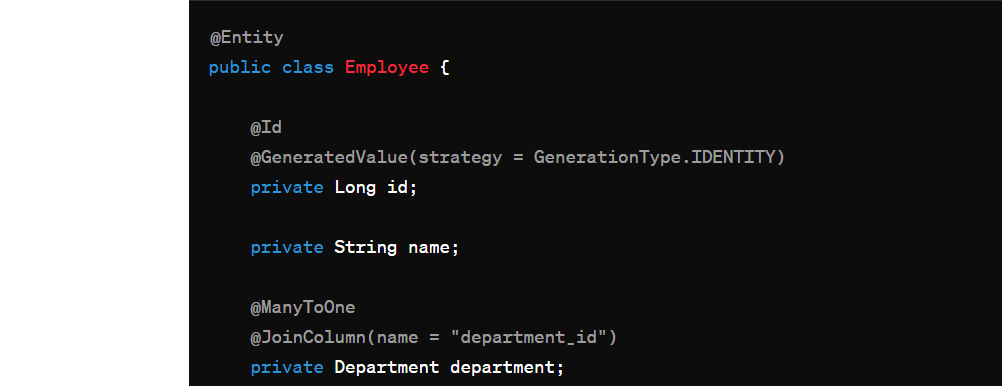
In Jpa Unidirectional & Bidirectional Join are handled using different annotations depending on the type of association between entities

### Unidirectional Join:

In Unidirectional association only one entity has reference to other.

For example

Employee and Department entity !!

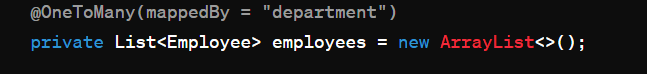


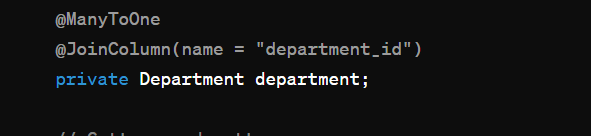
* **Employee** has a unidirectional **@ManyToOne** association with **Department**.
* The **@JoinColumn** annotation is used to specify the foreign key column in the **Employee** table (**department\_id**) that references the primary key of the **Department** table.

### Bidirectional Join:

In a bidirectional association, both entities have references to each other. Using the same example, let's add a bidirectional association to the **Department** entity:

#### **Bidirectional Join in** Department **Entity:**





# Design Patterns in JAVA

To solve a problems in a project we write a code for this and for writing such code we follow some conventions.

To solve a problem we follow Design patterns

There are 3 Design pattern

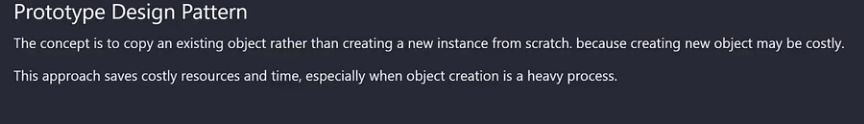
* Creational Design Pattern
  + Factory
  + Abstract Design
  + Singelton Design pattern
  + Builder Design Pattern
* Structural Design Patter
  + Adapter
  + Façade
  + Proxy
* Behavioural Design Pattern
  + Observer
  + Command
  + Strategy design pattern

## Creational Design Pattern:

*Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. A class creational pattern uses inheritance to vary the class that’s instantiated, whereas an object creational pattern will delegate instantiation to another object.*

# Prototype design pattern:

The concept to copy an exisiting



# @Primary Annotation

If one interface is implemented by more than one classes than we will define one of the classes as @Primary annotation If we inject the dependency of the class then by default @Primary annotation one class will be injected ☺

# Micro services Vs Monolithic Architecture

* Micro service architecture break down the whole project and make it loosely coupled independently deployable services each service responsible for specific business purpose.
* These Services are communicate with each other over the protocol of Http or messaging Queue.

**Benefits of Micro services:**

1. Scalability: It enables auto scaling and independent scaling services according to the demand allowing for better resource utilizations
2. **Flexibility and Agility**: Microservices promote agility by enabling teams to develop, deploy, and scale services independently.
3. **Resilience and Fault Isolation**: Failure in one service does not necessarily affect the entire system, as services are isolated from each other. This enhances fault tolerance and resilience.

**When to Use:**

* For large and complex projects with multiple teams working on different components.
* When scalability, flexibility, and fault tolerance are critical requirements.
* When the system needs to evolve rapidly to meet changing business needs.
* When a high degree of autonomy and independence among development teams is desired.

Monolithic:

When we have small requirements and we don’t have a big teams and we don’t need scalability and fault tolerance system we need one small project to handle each and everything and it’s deployed on single instance with low inputs

It will act as a single unit