Chase Questions:

Write code to check if a given string array is alphabetically sorted or not but alphabetic order is not regular abcd....
Boolean isSorted(String[] testStrings, String pattern){}
For ex testString array = aaa, aaab, caaaz
And pattern is cbad....

Kubernetes based questions
1- what is a pod
2- how services forward requests to pod when pod keeps on getting created and deleted.
3- how does kubernetes get to know if one pod is ready to take requests
4- how to store computation if pods dies in between
5- different type of services
6- what is liveness Vs readiness of the pod
7- difference between helm2- helm3
8- what are the different part of Deployment

Aws based questions
1- why to use AWS
2- eks Vs ECS
3- terraform Vs cloud formation

Kafka based questions
1- how is Kafka different from rabbit mq
2- how to handle scenarios when producer is fast but consumer is slow
3- how to handle large messages in kafka
Ans:
https://docs.cloudera.com/documentation/enterprise/6/6.3/topics/kafka_performance_large_messages.html

JP Morgan Interview 1st round
1, Explain how fix messaging mechanism works.
2. How do we know that client connection is fine in fix mechanism

3. Can concurrent modification exception occur in single threaded application
4. How do you make UI updates to data asynchronous

   Follow up how the UI would be updated with latest results once the asynchronous computation is completed
5. Asked about how you would cache something in your application locally?

Ans : Caching is a common practice in software development to improve performance by storing frequently accessed data in a local storage (cache) to reduce the need for expensive data retrieval operations. Here's how you can cache something in your application locally:

Choose a Caching Library/Framework: Before implementing caching, consider using a caching library or framework. Java applications often use libraries like Caffeine, Ehcache, or Guava for caching. These libraries provide efficient caching mechanisms and handle cache eviction, expiration, and other complexities.

Identify What to Cache: Determine which data or results are suitable for caching. Consider caching data that is relatively static or expensive to compute/retrieve and is frequently accessed.

Select a Cache Storage: Choose the appropriate storage for your cache. In-memory caching is suitable for small to moderately sized datasets that can fit in memory. For larger datasets, consider using disk-based caching.

Cache Configuration: Configure the caching library with the desired cache parameters, such as maximum size, expiration policies, eviction strategies, etc.

Cache Data: When you want to cache something, retrieve the data, and store it in the cache using a cache-specific method (e.g., put in Caffeine, put in Guava, etc.). The key used for caching should be unique and meaningful to the data you're caching.

Retrieve Data from Cache: Before fetching data from the original data source (e.g., database, API), check if the data exists in the cache. If it does, retrieve it from the cache instead of the original source.

Update Cache: If the underlying data changes (e.g., database update), update the cache accordingly to ensure cached data remains consistent.

Cache Expiry and Eviction: Set cache expiry times or use eviction policies to automatically remove stale or least-used data from the cache.

Exception Handling: Handle cache misses gracefully. If data isn't found in the cache, retrieve it from the original source, update the cache, and then return the data.

Monitor and Tune: Monitor cache performance and usage to ensure it's delivering the expected benefits. Adjust cache settings as needed based on usage patterns.

Here's a simplified example using the Caffeine caching library:

java
Copy code
```
import com.github.benmanes.caffeine.cache.Cache;
import com.github.benmanes.caffeine.cache.Caffeine;

public class CachingExample {

    public static void main(String[] args) {
        // Create a cache
        Cache<String, String> cache = Caffeine.newBuilder()
```

```
            .maximumSize(100)
            .build();

        String key = "exampleKey";
        String value = "exampleValue";

        // Store data in cache
        cache.put(key, value);

        // Retrieve data from cache
        String cachedValue = cache.getIfPresent(key);

        if (cachedValue != null) {
            System.out.println("Cached Value: " + cachedValue);
        } else {
            // Retrieve data from original source and update cache
            String retrievedValue = retrieveFromOriginalSource(key);
            if (retrievedValue != null) {
                cache.put(key, retrievedValue);
                System.out.println("Retrieved Value: " + retrievedValue);
            }
        }
    }

    private static String retrieveFromOriginalSource(String key) {
        // Simulate fetching data from the original source (e.g., database)
        return "dataFromOriginalSource";
    }
}
```

Remember that caching introduces complexities such as cache consistency, cache invalidation, and proper management of cached data. Consider the specific needs of your application and choose a caching strategy that aligns with your performance and data consistency requirements.

User

explain caffeine methods

ChatGPT

Caffeine is a popular Java caching library that provides a simple and flexible API for in-memory caching. It offers various features, such as automatic eviction, expiration, and support for custom policies. Let's explore some of the key methods and concepts provided by Caffeine:

Builder Configuration: Caffeine provides a Caffeine.newBuilder() method to create a Caffeine builder. The builder allows you to configure various caching parameters, such as maximum size, eviction policies, expiration, and more.

java

Copy code

```
Cache<String, String> cache = Caffeine.newBuilder()
        .maximumSize(100)
        .expireAfterWrite(10, TimeUnit.MINUTES)
        .build();
```

Cache Put: You can add or update entries in the cache using the put(key, value) method.

java

Copy code
```
cache.put("key1", "value1");
```
Cache Get: To retrieve values from the cache, you can use the get(key) method. If the value is not in the cache, Caffeine allows you to provide a loader function to fetch the value from an external source.

java
Copy code
```
String value = cache.get("key1", key -> fetchValueFromDatabase(key));
```
Automatic Loading: Caffeine provides the get(key, loader) method, which automatically loads a value from the loader function if it's not present in the cache.

java
Copy code
```
String value = cache.get("key1", key -> fetchValueFromDatabase(key));
```
Cache Removal: You can explicitly remove entries from the cache using the invalidate(key) method.

java
Copy code
```
cache.invalidate("key1");
```
Eviction Policies: Caffeine supports various eviction policies such as maximumSize, maximumWeight, and weakKeys, which control how many entries are retained in the cache.

java
Copy code
```
Cache<String, String> cache = Caffeine.newBuilder()
        .maximumSize(100)
        .build();
```
Expiration: You can set an expiration time for cache entries using the expireAfterWrite or expireAfterAccess methods.

java
Copy code
```
Cache<String, String> cache = Caffeine.newBuilder()
        .expireAfterWrite(10, TimeUnit.MINUTES)
        .build();
```
Async Loading: Caffeine supports asynchronous value loading using the getAsync(key, loader) method.

java
Copy code
```
CompletableFuture<String> future = cache.getAsync("key1", key -> fetchValueAsync(key));
```
Listeners: Caffeine allows you to register listeners to be notified when entries are created, updated, or removed from the cache.

java
Copy code
```
cache.policy().eviction().ifPresent(eviction -> {
    eviction().addListener((key, value) ->
        System.out.println("Entry evicted: " + key + " -> " + value));
});
```
Stats and Monitoring: You can access cache statistics using the Cache.stats() method, which provides information about hit rate, eviction count, etc.

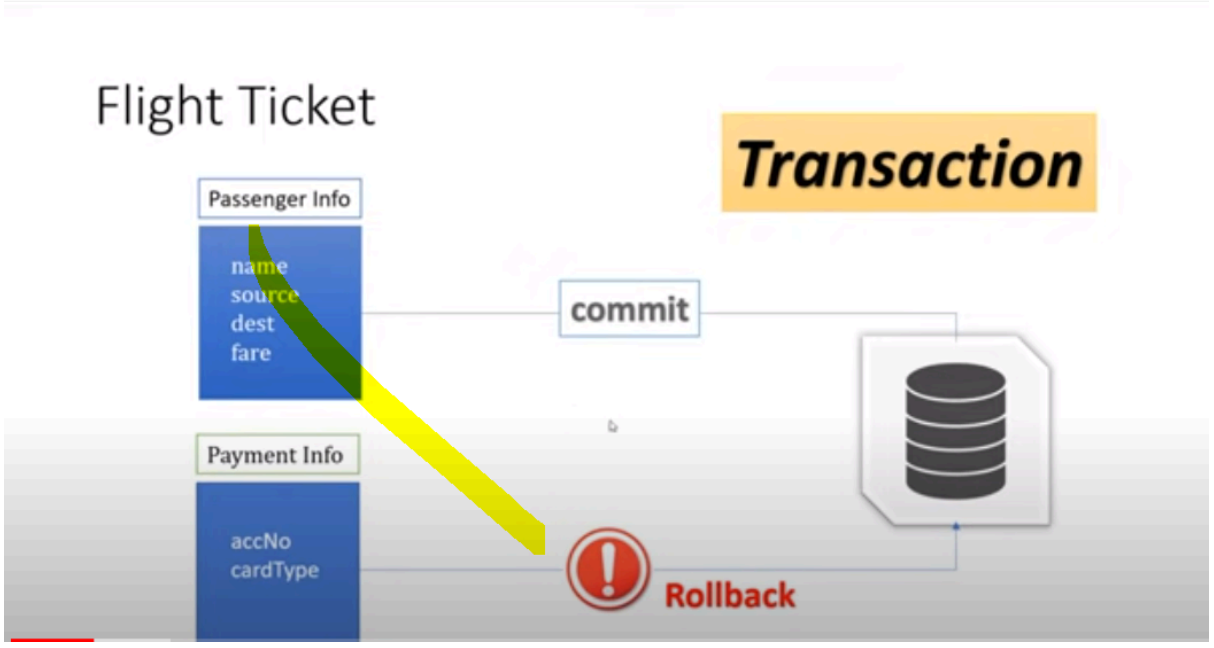
java
Copy code
```
CacheStats stats = cache.stats();
```

6. If I have a microservice and other microservice needs to call my microservice how would it call it in containerized env.
Rest project discussion at UBS/JPM Morgan and GS

Standard Chartered
1st round rapid fire round with core java and few linux shell commands question.
Then passed a code snippet to check pass by value or pass by reference concept

2nd Round
Design a multilevel parking which has three level 1, level 2 and level 3 each level has 100 spaces and a car takes 2 space, van takes 3 spaces, bike takes 1 space.
Design the system & then implement the park and unpark method.

CITI interview Questions
Benefits of spring boot
How you handled transactions spring boot based rest API



workspace-spring-tool-suite-4-4.16.1.RELEASE - SpringDemoProject/src/main/java/com/example/demo/service/EmployeeService.java - Spring Tool Suite 4

```java
@Autowired
private AddressService addressService;

@Transactional
public Employee addEmployee(Employee employee) throws Exception {

    Employee savedEmployee = crudRepo.save(employee);

    if(savedEmployee.getName().contains("code"))
        throw new Exception();

    Address address = new Address();
    address.setId(1L);
    address.setName("Banglore Whitefield");
    address.setEmp(employee);
    addressService.addAdressToo(address);

    return savedEmployee;
}
```

workspace-spring-tool-suite-4-4.16.1.RELEASE - SpringDemoProject/src/main/java/com/example/demo/service/AddressService.java - Spring Tool Suite 4

```java
package com.example.demo.service;

import org.springframework.beans.factory.annotation.Autowired;

@Service
public class AddressService {

    @Autowired
    private AddressRepo addressRepo;

    @Transactional
    public void addAdressToo(Address address) {
        addressRepo.save(address);

    }

}
```

How @Transactional works internally ?

Spring creates a proxy, or manipulates the class byte-code, to manage the creation, commit, and rollback of the transaction.

If we have a method like callMethod and we mark it as @Transactional, Spring will wrap some transaction management code around the invocation@Transactional method called:

```
createTransactionIfNecessary();
try {
    addEmployee();
    commitTransactionAfterReturning();
} catch (exception) {
    rollbackTransactionAfterThrowing();
    throw exception;
}
```

How to use @Transational

- You can use this annotation on following in the lowest to highest priority order :
  - interface,
  - superclass,
  - class,
  - interface method,
  - superclass method, and
  - method.

What is ACID transaction?

ACID is an acronym that stands for atomicity, consistency, isolation, and durability:

- **Atomicity** describes an **all or nothing** principle. Either all operations performed within the transaction get executed or none of them. That means if you commit the transaction successfully, you can be sure that all operations got performed. It also enables you to abort a transaction and roll back all operations if an error occurs.
- The **consistency** characteristic ensures that your **transaction takes a system from one consistent state to another consistent state**. That means that either all operations were rolled back and the data was set back to the state you started with or the changed data passed all consistency checks. In a **relational database**, that means that **the modified data needs to pass all constraint checks, like foreign key or unique constraints, defined in your database.**
- **Isolation** means that changes that you perform within **a transaction are not visible to any other transactions** until you commit them successfully
- **Durability** ensures that **your committed changes get persisted**

Employee.java | Address.java | EmployeeCon... | addr | emp | AddressRepo... | EmployeeSer... | hibernate_se... | Empl

```java
package com.example.demo.service;

import org.springframework.beans.factory.annotation.Autowired;

@Service
public class AddressService {

    @Autowired
    private AddressRepo addressRepo;

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void addAdressToo(Address address) {
        addressRepo.save(address);

    }

}
```

Default - transaction -> REQUIRED , if one physical is active (employee)- another transaction (nested for address) will not get created ,once method is executed then the code wrapped under transaction management , will save data for employee and address
Required_New-> seperate physical transaction for address, independent of parent

Types of Transaction Propagation - Require_new

Require_new -

- I want my completely own transaction.
- When the propagation is REQUIRES_NEW, Spring suspends the current transaction if it exists, and then creates a new one. It instructs the Spring container to always create a new inner physical transaction and not inherit an outer physical transaction's characteristics.
- An outer physical transaction will have its own database connection, while REQUIRES_NEW will create the inner physical transaction and will bound a new database connection to it.
- In a synchronous execution, while the inner physical transaction is running, the outer physical transaction is **suspended** and its database **connection remains open**. After the inner physical transaction commits, the outer physical transaction is resumed, continuing to run and commit/rollback.

NESTED - mark address one nested,it will create a save point at address save line , line above address save will not be rolled back, if something goes wrong

Types of Transaction Propagation - NESTED

NESTED Propogation -

- NESTED acts like REQUIRED, only it uses **savepoints** between nested invocations.
- For NESTED propagation, **Spring checks if a transaction exists**, and if so, **it marks a save point**. This means that if our business logic execution throws an exception, then the **transaction rollbacks to this save point**. If there's no active transaction, it works like REQUIRED.

Mandatory - need a transaction, but will not create my own ad throw an expedition instead

Types of Transaction Propagation - MANDATORY

MANDATORY Propagation -

- I'm not going to open up a transaction myself, but I'm going to throw exception if no one else opened one up. No New connection will be created by me.
- If this inner logical transaction is rolled back, then the outer logical transaction is rolled back as well, exactly as with the case of Propagation.REQUIRED.

```
Employee.java  Address.java  EmployeeCon...  addr  emp  AddressRepo...  EmployeeSer...  hibernate_se...  EmployeeRepo...  Adc
18      @Autowired
19      private EmployeeRepo crudRepo;
20
21      @Autowired
22      private AddressService addressService;
23
24      //@Transactional
25      public Employee addEmployee(Employee employee) throws Exception  {
26
27          Employee savedEmployee = crudRepo.save(employee);
28
29          Address address = new Address();
30          address.setId(1L);
31          address.setName("Banglore Whitefield");
32          //address.setEmp(employee);
33          addressService.addAdressToo(address);
34
35          return savedEmployee;
36      }
37
```

```
Employee.java  EmployeeCon...  addr  emp  AddressRepo...  EmployeeSer...  hibernate_se...  EmployeeRepo...  AddressServi...  Em
1  package com.example.demo.service;
2
3  import org.springframework.beans.factory.annotation.Autowired;
10
11 @Service
12 public class AddressService {
13
14     @Autowired
15     private AddressRepo addressRepo;
16
17     @Transactional(propagation = Propagation.MANDATORY)
18     public void addAdressToo(Address address) {
19         addressRepo.save(address);
20
21     }
22
23 }
24
```

NEVER- opposite of mandatory -> if transaction found , will throw an expectation, mark employee as trsactional, address with never propagation, it will throw an expedition

Types of Transaction Propagation - NEVER

NEVER Propogation -

- I'm going to throw an exception if someone else started up a transaction.
- The Propagation.NEVER states that no physical transaction should exist. If a physical transaction is found, then NEVER will cause an exception as follows - org.springframework.transaction.IllegalTransactionStateException: Existing transaction found for transaction marked with propagation 'never'.
- When you call a method annotated with NEVER, you must ensure that no physical transaction is open. The code inside this method can open physical transactions with no problem. . eg: : When the code reaches the save() method, Spring will open a physical transaction especially for running this call. This happens because save() takes advantage of the default Propagation.REQUIRED. JPA Repository methods are by default enclosed in transaction boundaries.

NOT Supported

Types of Transaction Propagation - NOT_SUPPORTED

Propagation.NOT_SUPPORTED -

- If a current transaction exists, first Spring suspends it, and then the business logic is executed without a transaction:

  This physical transaction will be automatically resumed at the end. After this transaction is resumed, it can be rolled back (in case of a failure) or committed.

  If the RuntimeException was thrown in child(), then this exception was propagated in parent(), and this logical transaction is rolled back.

Support

Types of Transaction Propagation - SUPPORTS

SUPPORTS Propogation -

- I don't really care if a transaction is open or not, i can work either way
- For SUPPORTS, Spring first checks if an active transaction exists. If a transaction exists, then the existing transaction will be used. If there isn't a transaction, it is executed non-transactional:

What Is @Transactional?

We can use **@Transactional** to wrap a method in a database transaction.

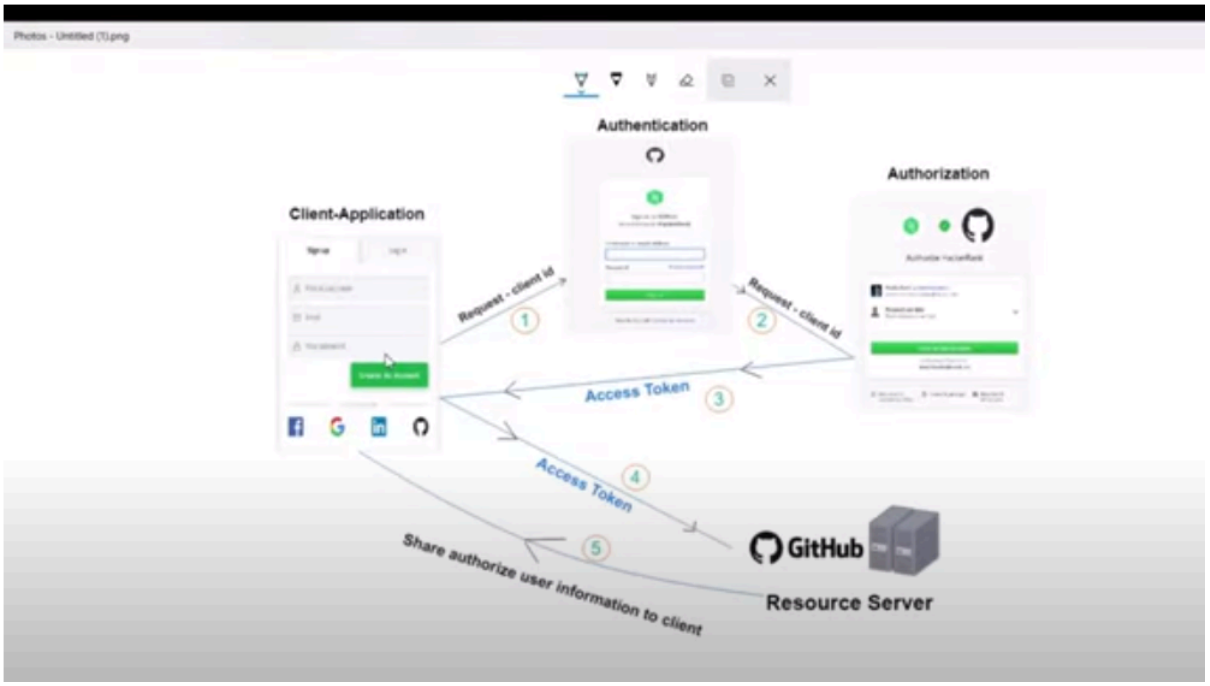
It allows us to set

- propagation,
- isolation,
- timeout,
- read-only, and
- rollback

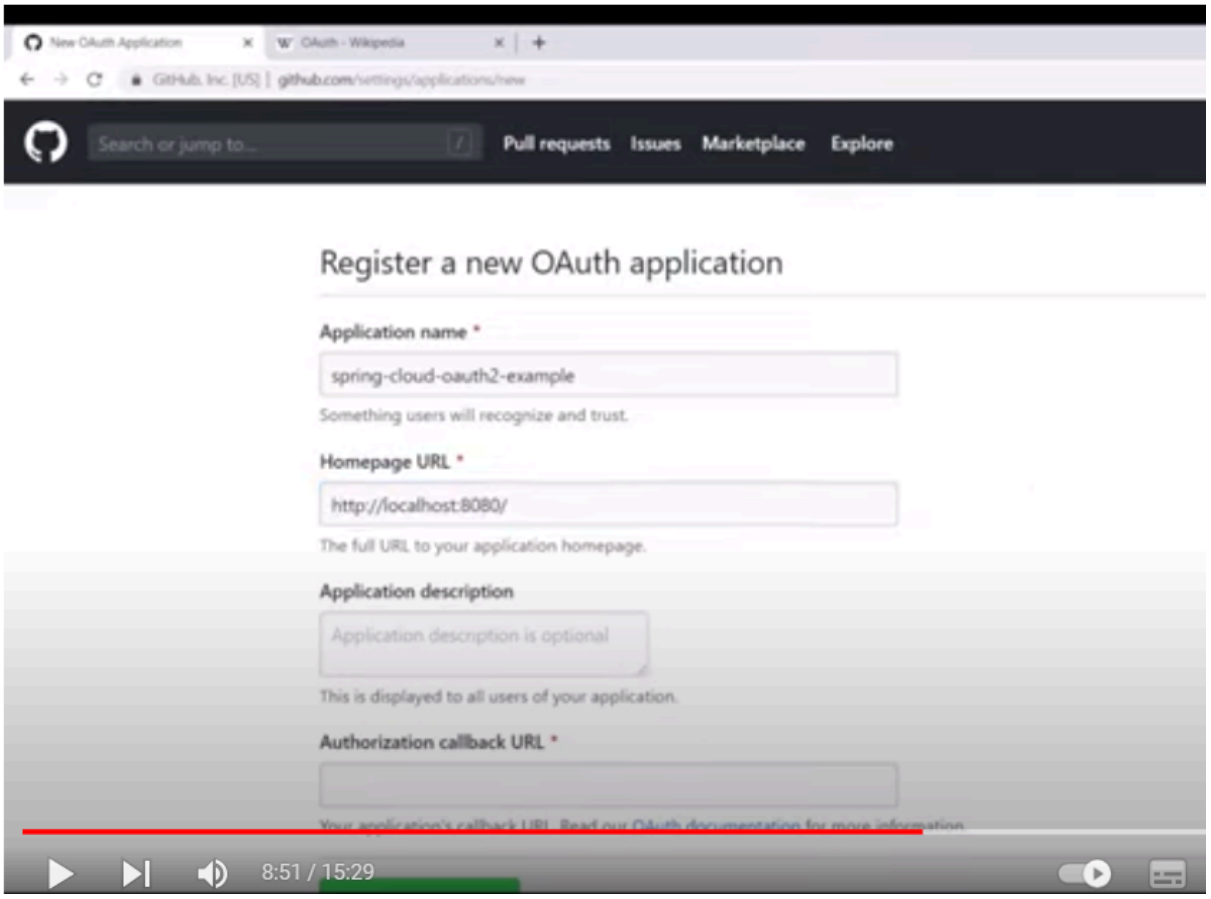
nditions for our transaction.

How you handled authentication/authorization in spring boot based rest application

Oauth2 flow : OAuth 2.0 (Open Authorization 2.0) is an authorization framework that allows applications to obtain limited access to user accounts on an HTTP service, such as Facebook, Google, or a custom API, without exposing the user's credentials. Means delegate authorization from third party applications, without exposing user credentials



Application registration in github - oauth section



This will give client id and secret for authentication and authorization

Client code change:

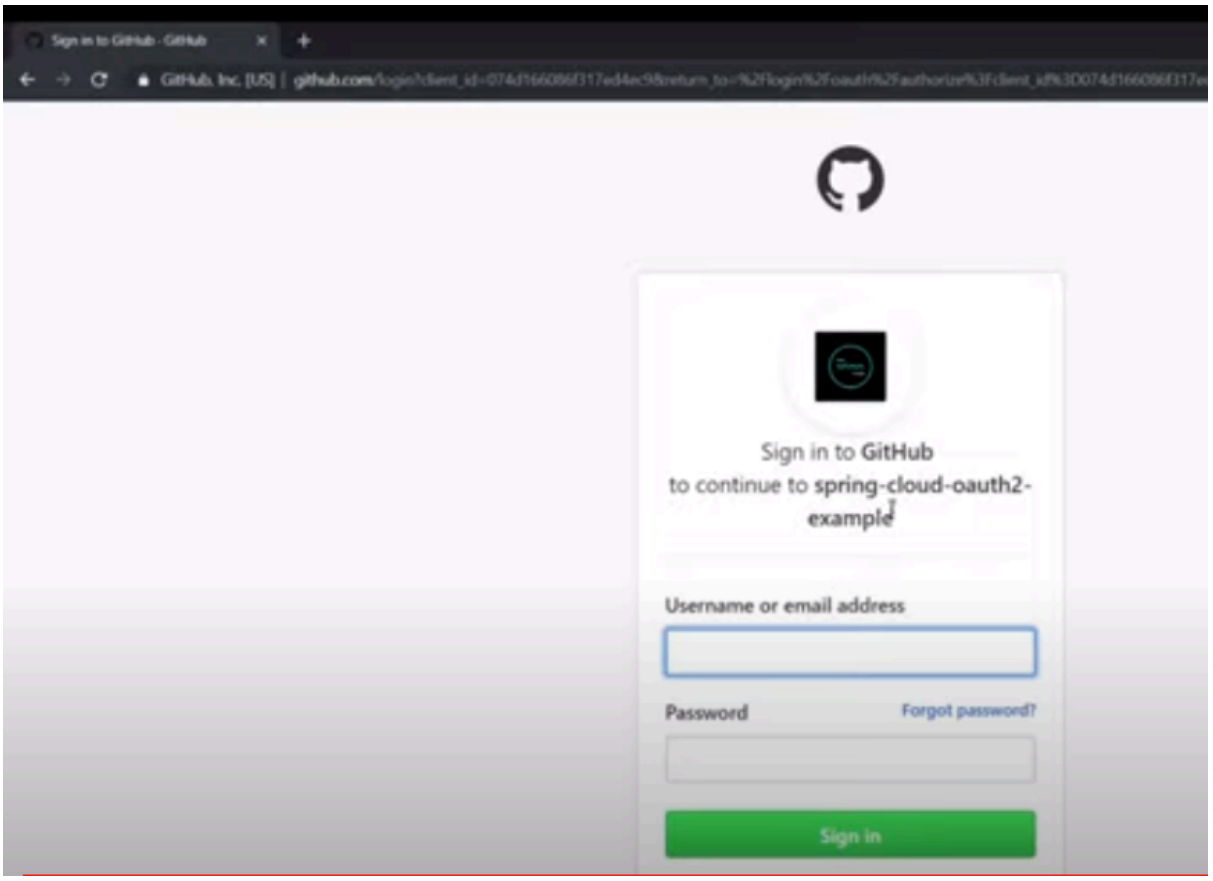
spring-cloud-oauth2-example/pom.xml | SpringCloudOauth2ExampleApplication.java | *application.yml

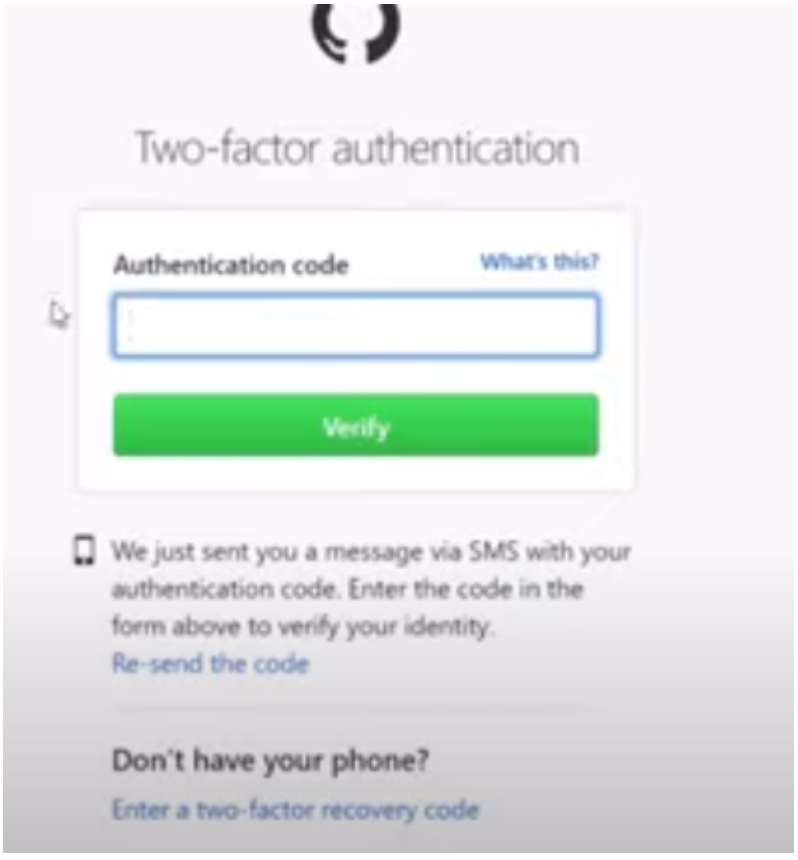
```yaml
security:
  oauth2:
    client:
      clientId:
      clientSecret:
      accessTokenUri: https://github.com/login/oauth/access_token
      userAuthorizationUri: https://github.com/login/oauth/authorize
      clientAuthenticationScheme: form
    resource:
      user-info-uri: https://api.github.com/user
      prefer-token-info: false
```

spring-cloud-oauth2-example/pom.xml | SpringCloudOauth2ExampleApplication.java | application.yml

```java
package com.javatechie.spring.cloud.security.api;

import java.security.Principal;

@SpringBootApplication
@EnableOAuth2Sso
@RestController
public class SpringCloudOauth2ExampleApplication {

    @GetMapping("/")
    public String message(Principal principal) {
        return "Hi "+principal.getName()+" welcome to SpringCloudOauth2ExampleApplication";
    }

    public static void main(String[] args) {
        SpringApplication.run(SpringCloudOauth2ExampleApplication.class, args);
    }

}
```

localhost:8080..


Sign in to GitHub
to continue to spring-cloud-oauth2-example
Username or email address
Password
Forgot password?
Sign in


Two-factor authentication
Authentication code
What's this?
Verify
We just sent you a message via SMS with your authentication code. Enter the code in the form above to verify your identity.
Re-send the code
Don't have your phone?
Enter a two-factor recovery code

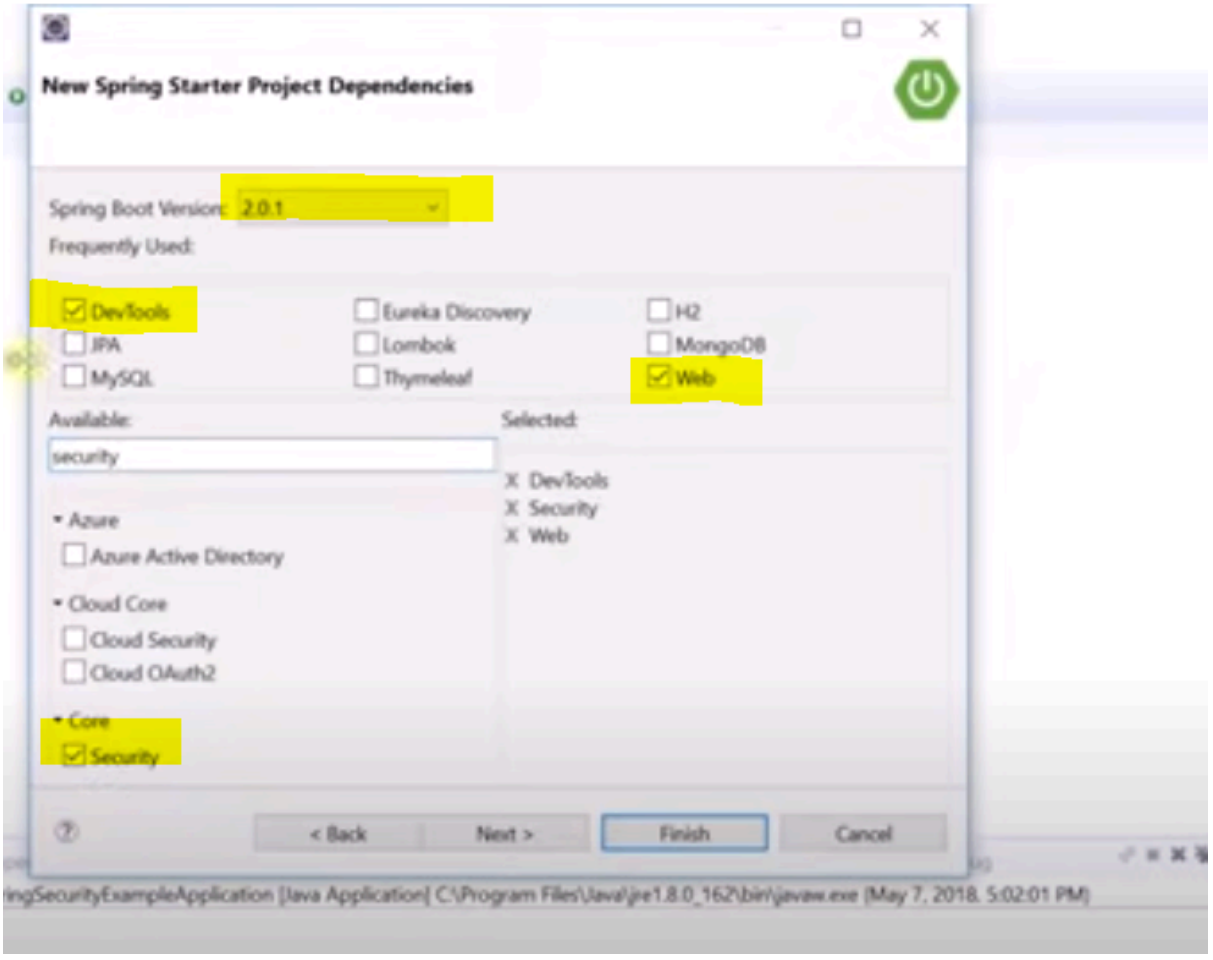
Access token git hub will provide dynamically- it can be seen under cookies(inspect in browser)

Annotations used : @enableOAuth2Sso

Dependency: spring-boot-starter-parent-2.1.4

Config- application.yaml

// Spring boot 3.0 security



```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService(){
        UserDetails admin= User.withUsername("Basant")
                .password("Pwd1")
                .roles("ADMIN")
                .build();
        UserDetails user= User.withUsername("Basant")
                .password("Pwd1")
                .roles("ADMIN")
                .build();
    }
```

Password encoder

```java
    public UserDetailsService userDetailsService(){
        UserDetails admin= User.withUsername("Basant")
                .password("Pwd1")
                .roles("ADMIN")
                .build();
        UserDetails user= User.withUsername("John")
                .password("Pwd2")
                .roles("USER")
                .build();
    }

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

Authorisation part

```
}

@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http.csrf().disable()
            .authorizeHttpRequests()
            .requestMatchers("/products/welcome").permitAll()
            .and()
            .authorizeHttpRequests().requestMatchers("/products/**").authenticated()
            .and().formLogin()
            .and().build();
}

@Bean
```

Now specify roles in controller - @preAuthorized(hasAuthorty='Admin') & put @enableMethodSecurity in securityConfig file

```
@GetMapping("/welcome")
public String welcome() {
    return "Welcome this endpoint is not secure";
}

@GetMapping("/all")
@PreAuthorize("hasAuthority('ADMIN')")
public List<Product> getAllTheProducts() {
    return service.getProducts();
}

@GetMapping("/{id}")
public Product getProductById(@PathVariable int id) {
    return service.getProduct(id);
}
```

OR

src / main / java / com / javatechie / controller / ProductController / getAllTheProducts

ProductService.java | application.properties | SecurityConfig.java | EnableGlobalMethodSecurity.java

```java
        return "Welcome this endpoint is not secure";
    }

    @GetMapping("/all")
    @PreAuthorize("hasAuthority('ROLE_ADMIN')")
    public List<Product> getAllTheProducts() {
        return service.getProducts();
    }

    @GetMapping("/{id}")
    @PreAuthorize("hasAuthority('ROLE_USER')")
    public Product getProductById(@PathVariable int id) {
        return service.getProduct(id);
    }
}
```

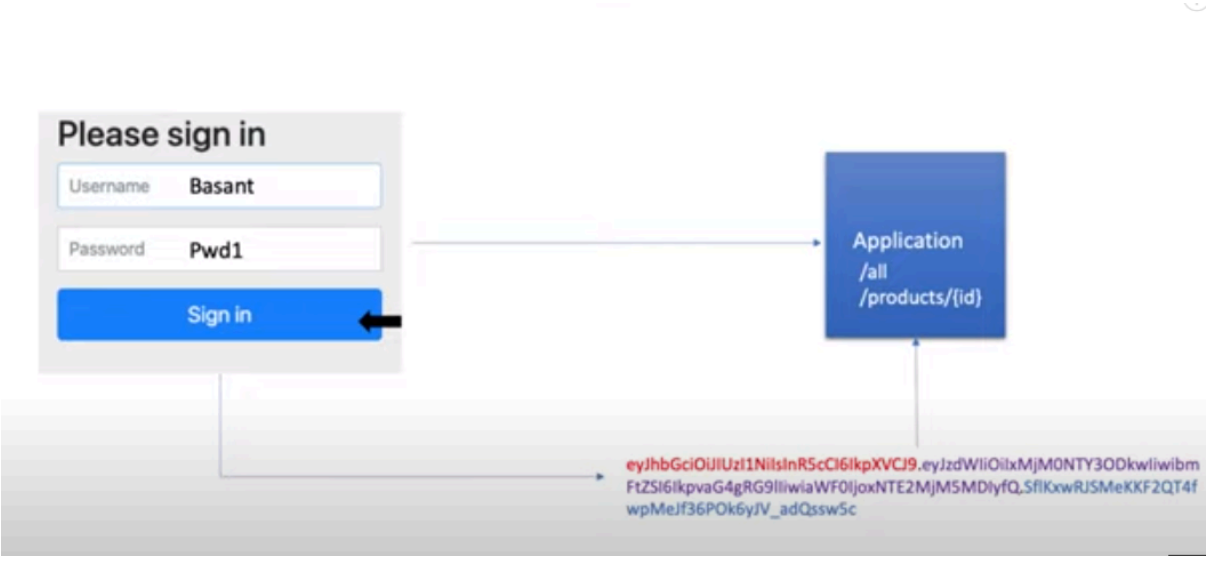
Run | Problems | Jenkins Linter | Terminal | DB Execution Console | TODO | Sequenc

```java
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {

    @Bean
    //authentication
    public UserDetailsService userDetailsService(PasswordEncoder encoder) {
        UserDetails admin = User.withUsername("Basant")
                .password(encoder.encode( rawPassword: "Pwd1"))
                .roles("ADMIN")
                .build();
        UserDetails user = User.withUsername("John")
                .password(encoder.encode( rawPassword: "Pwd2"))
                .roles("USER")
                .build();
        return new InMemoryUserDetailsManager(admin, user);
    }
```

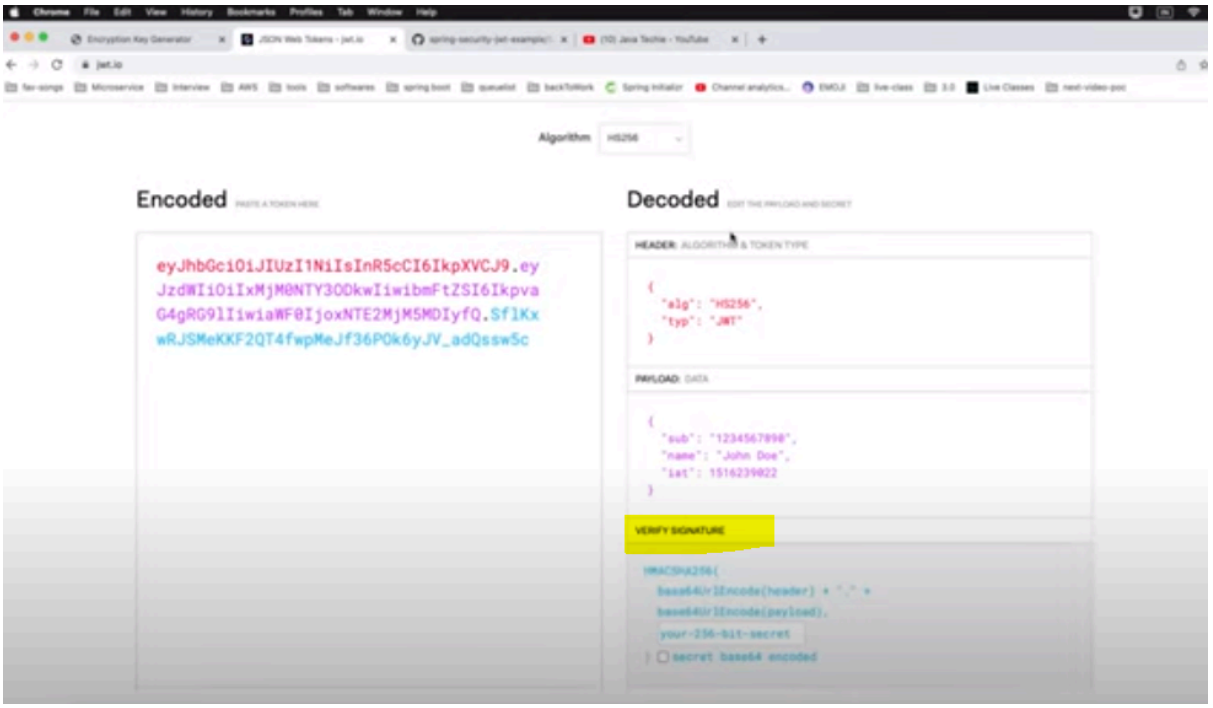
In the above approach , we provide username/pwd everytime , which we dont want

JWT

In this approach only 1st time we provide credentials then for next time access- a token is getting provided



This token has 3 parts-



Header ,payload and signature (this comes from server,need to keep it safe)

Add jwt dependencies - these will help in creating token

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
```

spring-boots-jwt / src / main / java / com / javatechie / controller / ProductController.java

Code Blame 64 lines (51 loc) · 2.06 KB

```java
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/products")
public class ProductController {

    @Autowired
    private ProductService service;
    @Autowired
    private JwtService jwtService;

    @Autowired
    private AuthenticationManager authenticationManager;

    @GetMapping("/welcome")
    public String welcome() {
        return "Welcome this endpoint is not secure";
    }

    @PostMapping("/new")
    public String addNewUser(@RequestBody UserInfo userInfo) {
        return service.addUser(userInfo);
    }

    @GetMapping("/all")
    @PreAuthorize("hasAuthority('ROLE_ADMIN')")
    public List<Product> getAllTheProducts() {
        return service.getProducts();
    }

    @GetMapping("/{id}")
    @PreAuthorize("hasAuthority('ROLE_USER')")
    public Product getProductById(@PathVariable int id) {
        return service.getProduct(id);
    }


    @PostMapping("/authenticate")
    public String authenticateAndGetToken(@RequestBody AuthRequest authRequest) {
        Authentication authentication = authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(authRequest.getUsername(), authRequest.getPassword()));
        if (authentication.isAuthenticated()) {
            return jwtService.generateToken(authRequest.getUsername());
        } else {
            throw new UsernameNotFoundException("invalid user request !");
        }


```

JWT Service

```java
package com.javatechie.service;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.io.Decoder;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;

import java.security.Key;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

@Component
public class JwtService {
```

```
    public static final String SECRET =
"5367566B59703373367639792F423F4528482B4D6251655468576D5A71347437";


    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    public Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }

    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }

    private Claims extractAllClaims(String token) {
        return Jwts
                .parserBuilder()
                .setSigningKey(getSignKey())
                .build()
                .parseClaimsJws(token)
                .getBody();
    }

    private Boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }

    public Boolean validateToken(String token, UserDetails userDetails) {
        final String username = extractUsername(token);
        return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
    }


    public String generateToken(String userName){
        Map<String,Object> claims=new HashMap<>();
        return createToken(claims,userName);
```

```
    }

    private String createToken(Map<String, Object> claims, String userName) {
        return Jwts.builder()
                .setClaims(claims)
                .setSubject(userName)
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis()+1000*60*30))
                .signWith(getSignKey(), SignatureAlgorithm.HS256).compact();
    }

    private Key getSignKey() {
        byte[] keyBytes= Decoders.BASE64.decode(SECRET);
        return Keys.hmacShaKeyFor(keyBytes);
    }
}
```

AuthRequest

```
package com.javatechie.dto;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class AuthRequest {

    private String username ;
    private String password;
}
```

SecurityConfig

```
package com.javatechie.config;

import com.javatechie.filter.JwtAuthFilter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
```

```
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import
org.springframework.security.config.annotation.authentication.configuration.AuthenticationC
onfiguration;
import
org.springframework.security.config.annotation.method.configuration.EnableGlobalMethodS
ecurity;
import
org.springframework.security.config.annotation.method.configuration.EnableMethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {
    @Autowired
    private JwtAuthFilter authFilter;

    @Bean
    //authentication
    public UserDetailsService userDetailsService() {
//      UserDetails admin = User.withUsername("Basant")
//              .password(encoder.encode("Pwd1"))
//              .roles("ADMIN")
//              .build();
//      UserDetails user = User.withUsername("John")
//              .password(encoder.encode("Pwd2"))
```

```
//                .roles("USER","ADMIN","HR")
//                .build();
//        return new InMemoryUserDetailsManager(admin, user);
        return new UserInfoUserDetailsService();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http.csrf().disable()
                .authorizeHttpRequests()
                .requestMatchers("/products/new","/products/authenticate").permitAll()
                .and()
                .authorizeHttpRequests().requestMatchers("/products/**")
                .authenticated().and()
                .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and()
                .authenticationProvider(authenticationProvider())
                .addFilterBefore(authFilter, UsernamePasswordAuthenticationFilter.class)
                .build();
    }

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

    @Bean
    public AuthenticationProvider authenticationProvider(){
        DaoAuthenticationProvider authenticationProvider=new DaoAuthenticationProvider();
        authenticationProvider.setUserDetailsService(userDetailsService());
        authenticationProvider.setPasswordEncoder(passwordEncoder());
        return authenticationProvider;
    }
    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration config)
throws Exception {
        return config.getAuthenticationManager();
    }
```

}

JWT Auth filter - to validate

```
package com.javatechie.filter;

import com.javatechie.config.UserInfoUserDetailsService;
import com.javatechie.service.JwtService;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.aspectj.weaver.patterns.IToken;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

@Component
public class JwtAuthFilter extends OncePerRequestFilter {

    @Autowired
    private JwtService jwtService;

    @Autowired
    private UserInfoUserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse
response, FilterChain filterChain) throws ServletException, IOException {
        String authHeader = request.getHeader("Authorization");
        String token = null;
        String username = null;
        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            token = authHeader.substring(7);
            username = jwtService.extractUsername(token);
        }
```

```
        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null)
{
            UserDetails userDetails = userDetailsService.loadUserByUsername(username);
            if (jwtService.validateToken(token, userDetails)) {
                UsernamePasswordAuthenticationToken authToken = new
UsernamePasswordAuthenticationToken(userDetails, null, userDetails.getAuthorities());
                authToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(authToken);
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

UserInfoRepository

```
package com.javatechie.repository;

import com.javatechie.entity.UserInfo;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserInfoRepository extends JpaRepository<UserInfo, Integer> {
    Optional<UserInfo> findByName(String username);

}
```

UserInfo

```
package com.javatechie.entity;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Data;
```

```
import lombok.NoArgsConstructor;

@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class UserInfo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String name;
    private String email;
    private String password;
    private String roles;
}
```

Microservices:

, it's very easy to build your complex applications because understanding and deploying a whole complex application in one single unit is very difficult. But when you divide your whole complex application into small, small applications and modules, it is very easy. And you can actually delegate those small, small applications to one person in a team and you can easily deploy and develop them. It also gives flexibility to choose the technology.
It's just that they should interact with each other through any medium, RESTful web services or messaging. identify all possible standalone functionalities and modules. Spring Boot because these are the rapid application development technology
So, to make this a proper RESTful web services implemented into your microservices, you need proper load balancers, you need Eureka for service discovery, which is actually useful when you have the load balancing in place, you have cloud deployments. Then these applications are not just interacting with each other, they are actually interacting with Eureka for the service discovery in many more ways. You have API gateways and many more stuff to make sure that your web services are capable enough to interact even at the cloud environment. So, this is not just enough to create a Spring Boot application and interact with the REST APIs. Eureka is for sure, say that no, this is not a microservices architecture. These are RESTful web services that you have created. You need different tools and technologies to make them a successful microservice architecture, which we are going to see in a few seconds. So, let's get started.

Different design patterns used for microservices based architecture

What are Microservices?

Microservices is an architecture where the application is exposed as loosely coupled services that can be independently developed, deployed, and maintained. Each service exposed is referred to as Microservice. Each service performs a unique function.

Speciality of this architecture is that polyglot architecture is supported. For example, if a team is working on one of the microservice using Java, Spring Boot, and MySQL, another team can work on another microservice using Python, Node JS, and NoSQL.

Different microservices can use a different version of the same programming language.
Different microservices can use different programming languages.
Different microservices can use different architectures as well.

docs.google.com/presentation/d/1UawAii2_ATPNmLZo8mR2vrE1U4UVq_xD/edit#slide=id.g10b802bbb0a_0_0

Why Microservices?

In the case of monolith applications, there are several problems like

- Same code base for presentation, business layer, and data access layer. Application is deployed as a single unit.
- Complex to maintain and scalability is an issue.

Microservice solves the above problems.
Microservices are ideal when a monolith or a legacy application needs to be modernized.

For new software development, if the key business drivers are to reduce time to market, scalable better software, lower costs, faster development, or cloud-native development, microservices are ideal.

Each service is independent and gives the flexibility to choose the programming language, database, and/or architecture.

Distinct services can be developed, deployed, and maintained independently.

docs.google.com/presentation/d/1UawAii2_ATPNmLZo8mR2vrE1U4UVq_xD/edit#slide=id.g10b802bbb0a_0_0

What are the pros and cons of Microservice Architecture?

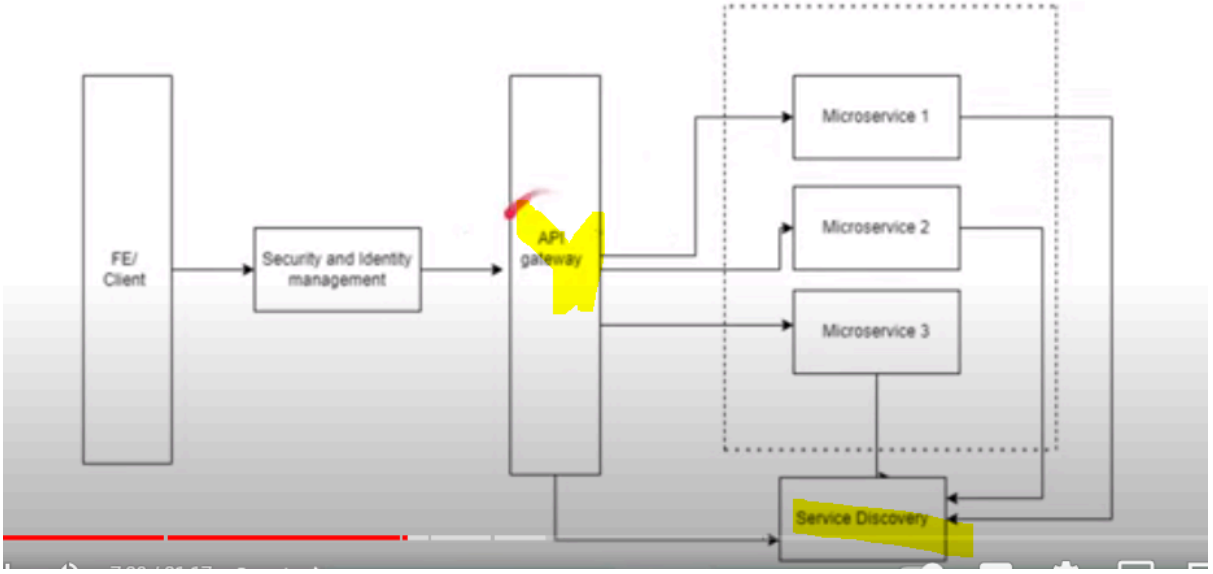
Pros of Microservice Architecture	Cons of Microservice Architecture
Freedom to use different technologies	Management of a large number of services is difficult.
Each microservices focuses on single capability	Communication between microservices is complex.
Supports individual deployable units	Increased efforts for configuration and other operations
Allow frequent software releases	Difficult to maintain transaction safety and data boundaries
Ensures security of each service	Due to the decentralized nature of microservices, more microservices will mean more resources hence high Investment
Multiple services are parallelly developed and deployed	Debugging of problems is harder unless the right instrumentation is followed during design and development.
	Microservices will need a large team size with the right mix of experience in design, development, automation, deployments, tools, and testing.

11:53 / 23:59

When to use microservices?

- Reduce time to market,
- Scalable better software,
- Lower costs,
- Faster development,
- Cloud-native development
- It makes sense to adopt a microservices architecture, if the team size is big enough as each service will require its team to develop, deploy and manage.
- Timeframe and skills of team members are a constraint.
- If fast results are required,
- choose microservices architecture only if the team also has experience in microservices.
- Do not use this architecture for simple application which can be managed by monolithic application .
- So you use ask yourself first do we really need this microservice architecture to decouple the services as it comes with a cost

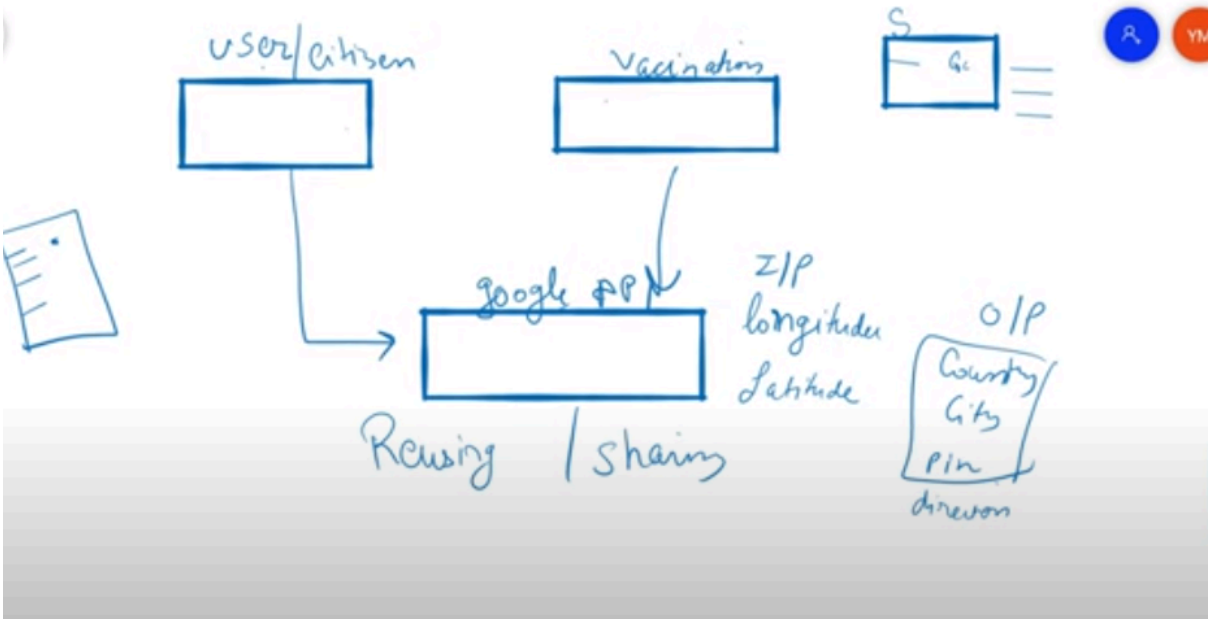
How does Microservice Architecture work?



What is the difference between Monolithic, SOA and Microservices Architecture?

- Monolithic Architecture is similar to a big container wherein all the software components of an application are assembled together and tightly packaged.
- A Service-Oriented Architecture is a collection of services which communicate with each other. The communication can involve either simple data passing or it could involve two or more services coordinating some activity.
- Microservice Architecture is an architectural style that structures an application as a collection of small autonomous services, modeled around a business domain.
- Main diff b/w SOA and MS architecture is Based on sharing of data and info. SOA shares and reuses as much as possible while MS focuses on sharing as little as possible

SOA



Ways to communicate between Microservices

- We have seen Synchronous communications through -
  - Rest APIs
  - GraphQl
  - Feign using Eureka discoveries
  - GRPC (10 times faster than REST APIs) - developed by Google as substitute of REST with many more features.
- A synchronous call means that a service waits for the response after performing a request.
- Today we will look at ways to do asynchronous communication in java. This communication usually involves some kind of messaging system like
  - Active Mqs
  - Rabbit MQs
  - Kafka

Point TO Point & Producer -Consumer

Async Communication

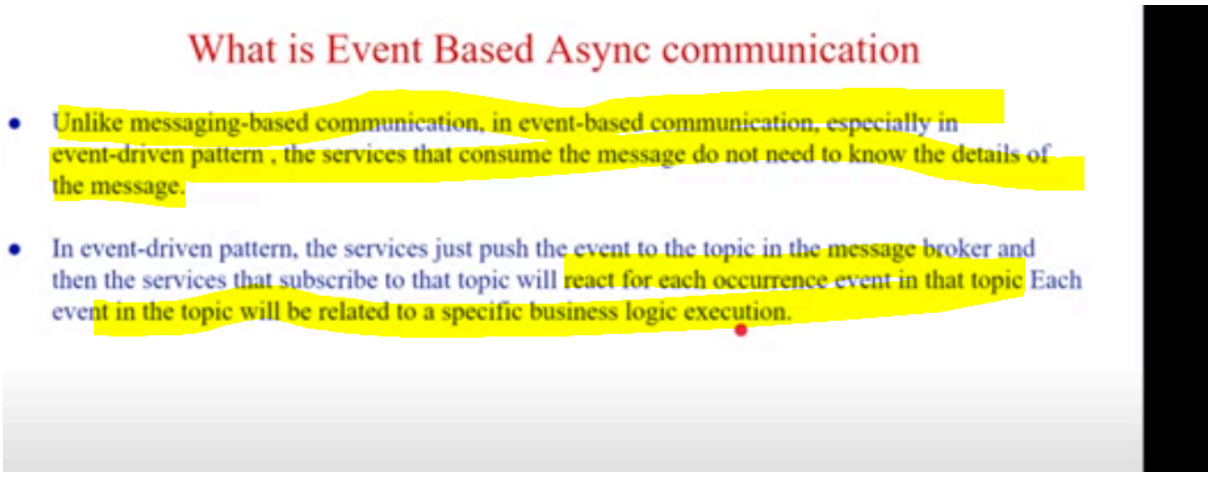
What is PTP Async communication

- PTP - A queue will be used for this type of messaging-based communication.
- The service that produces the message, which is called as producer (sender), will send the message to a queue in one message broker and the service that has an interest in that message, which is called a consumer (receiver), will consume the message from that queue and carry out further processes for that message
- One message sent by a producer can be consumed by only one receiver and the message will be deleted after consumed.
- If the receiver or an interested service is down, the message will remain persistent in that queue until the receiver is up and consumes the message.
- For this reason, messaging-based communication is one of the best choices to make our microservices resilient.
- A popular choice for the queueing system is RabbitMQ, ActiveMQ

10:47 / 15:45 • Types of asynchronous communication

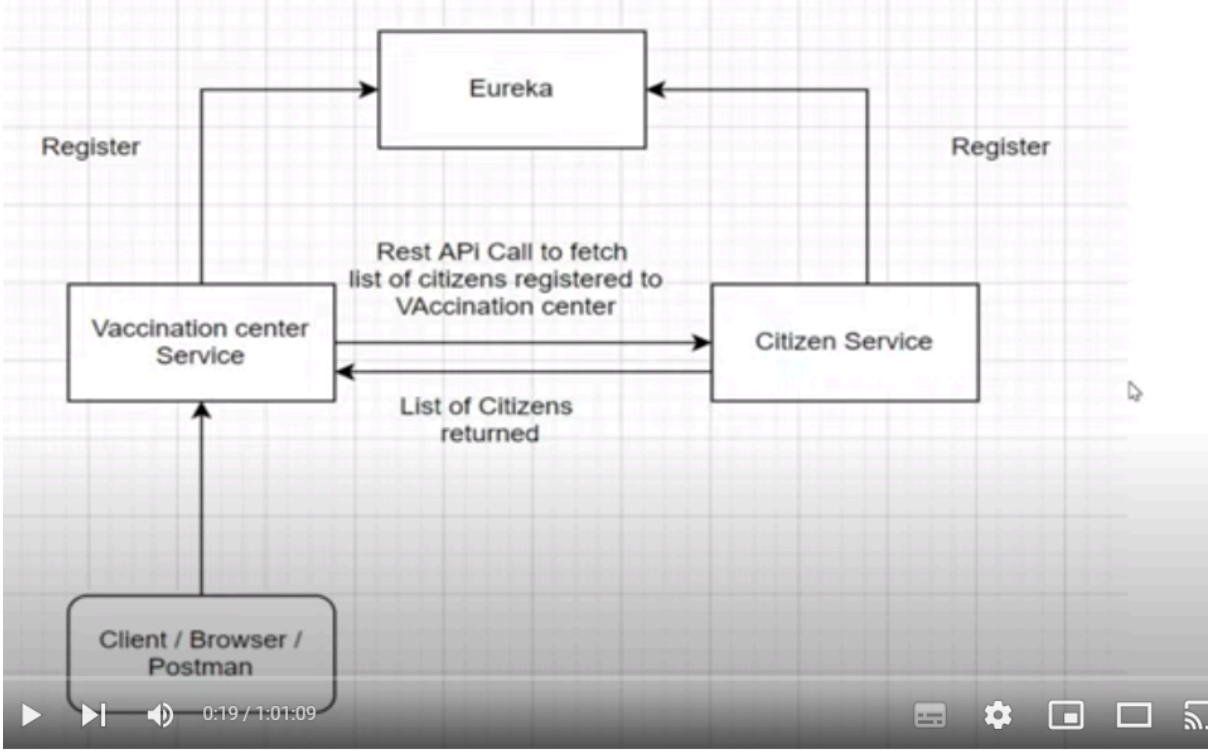
What is Publisher-Subscriber Async communication

- In publisher-subscriber messaging-based communication, the topic in the message broker will be used to store the message sent by the publisher and then subscribers that subscribe to that topic will consume that message
- Unlike point to point pattern, the message will be ready to consume for all subscribers and the topic can have one or more subscribers. The message remains persistent in a topic until we delete it.
- In messaging-based communication, the services that consume messages, either from queue or topic, must know the common message structure that is produced or published by producer or publisher.
- examples are Kafka, Amazon SNS etc

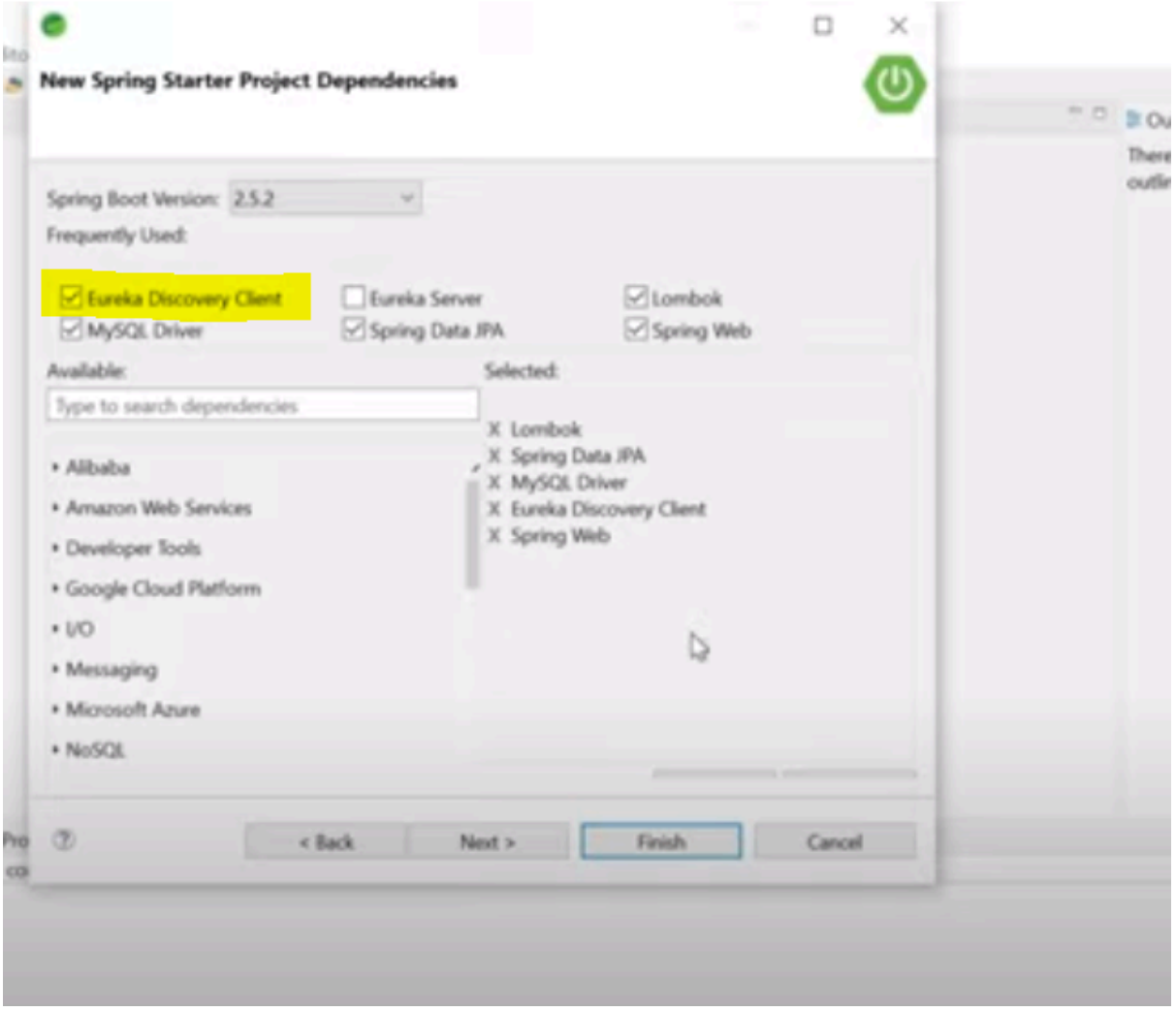


Eureka

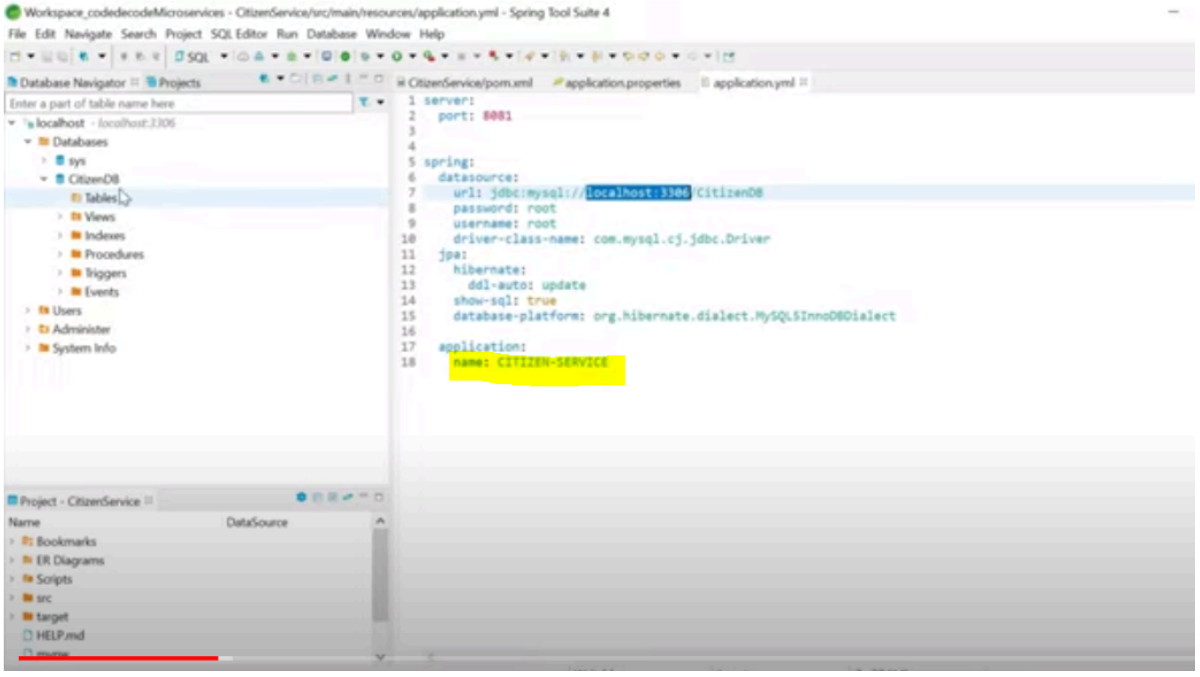
It handles load balancing situation and dynamic urls



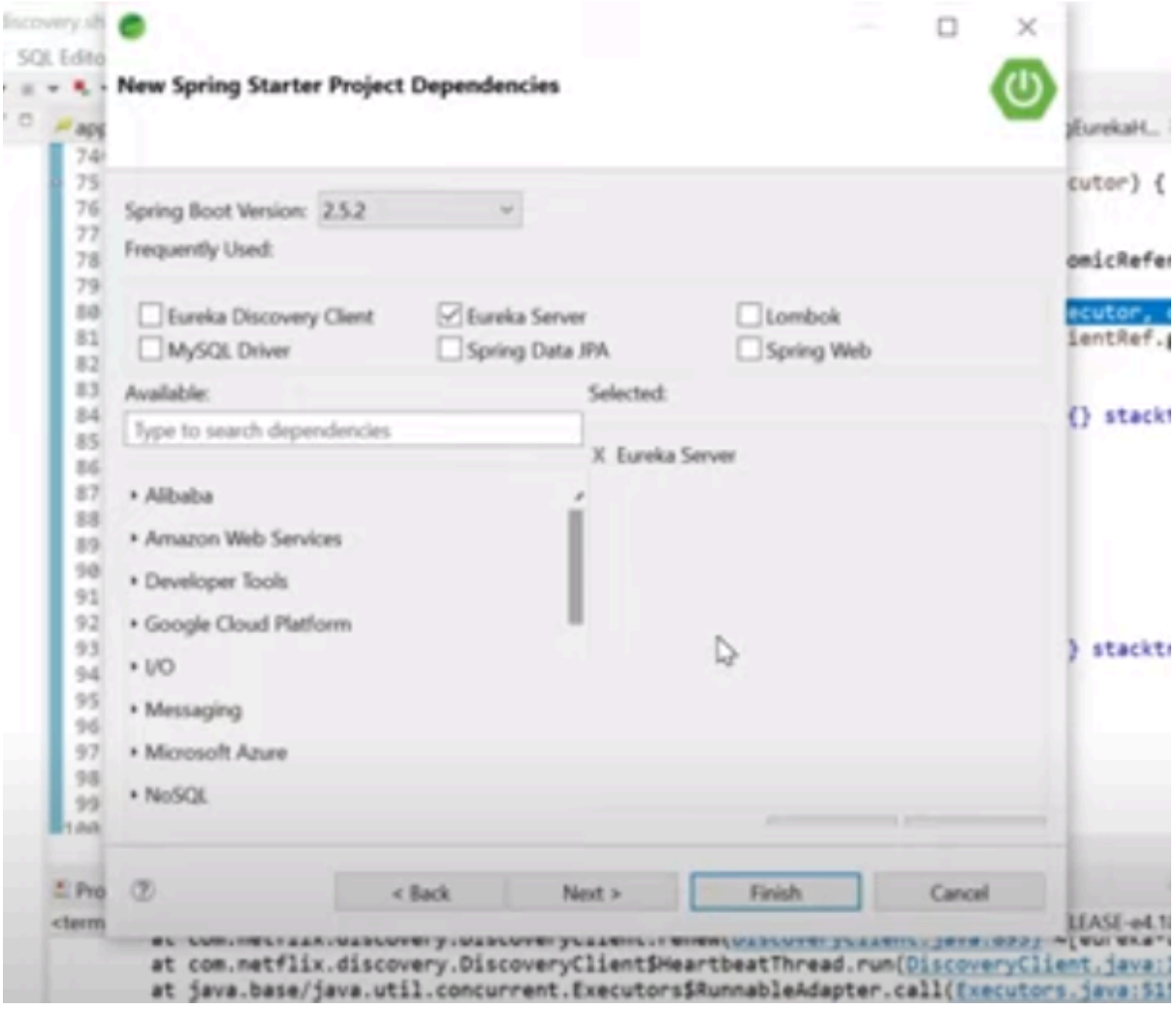
Vaccination and citizen service:



Application yaml

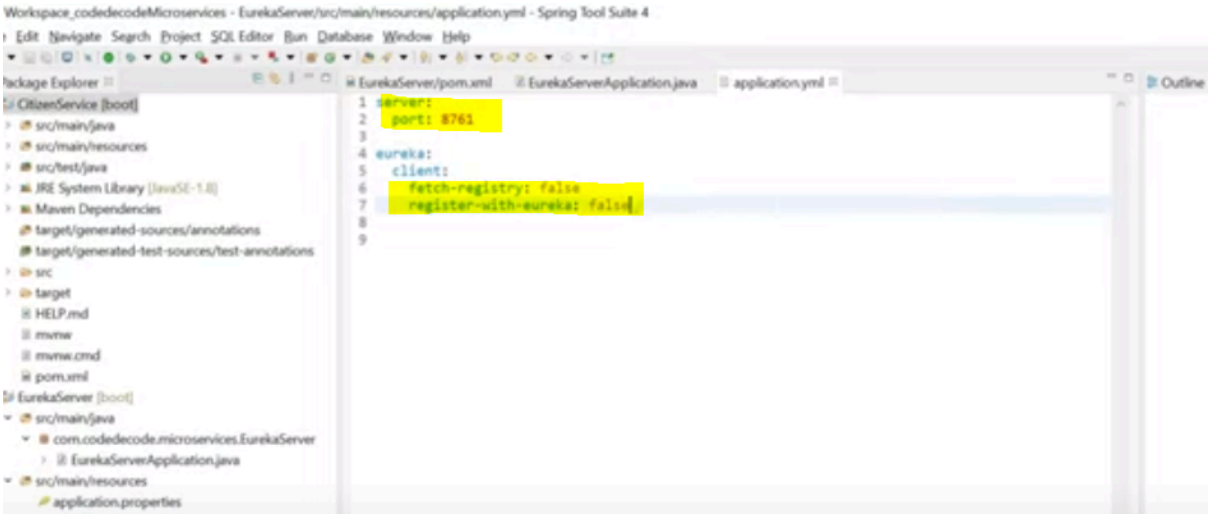


Eureca Server Service

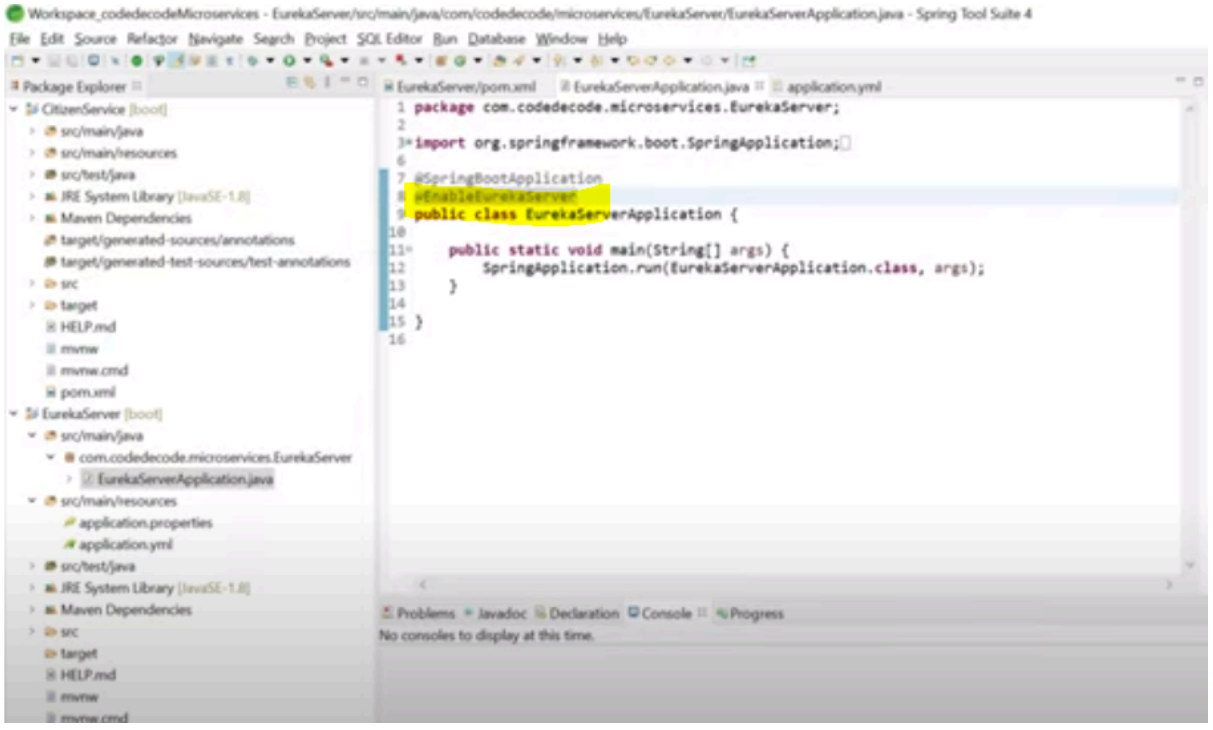


Application yaml -> default server port

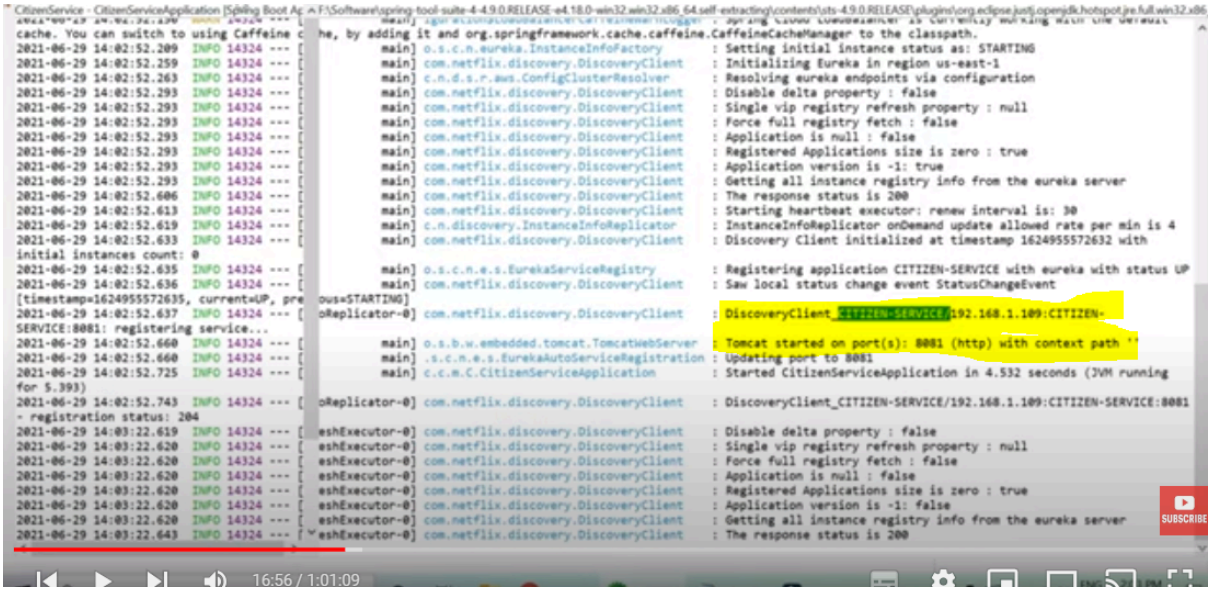
And do not try to connect with other eureka server properties



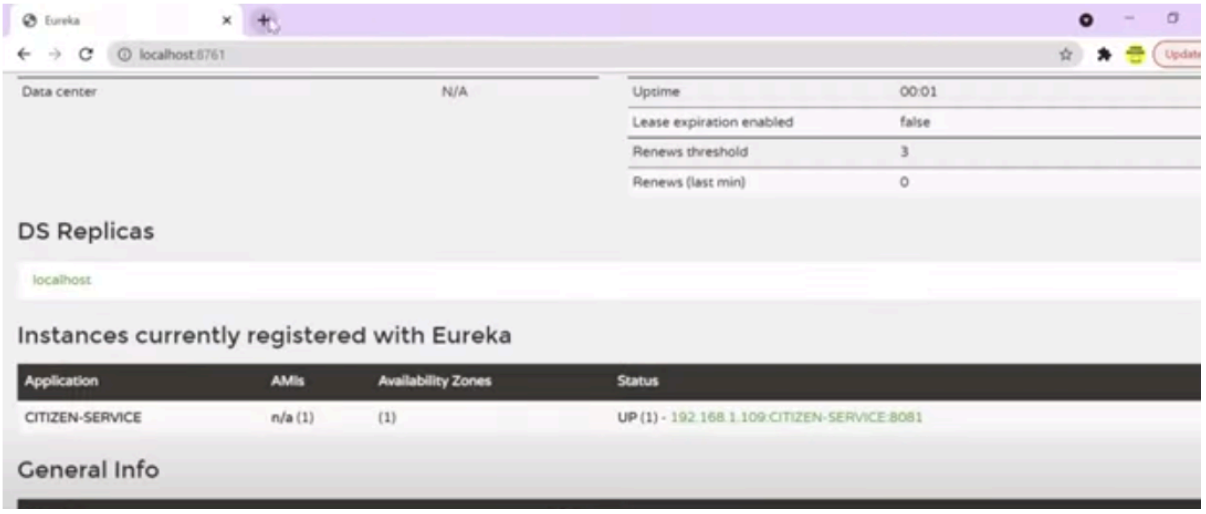
@EnableEurekaServer



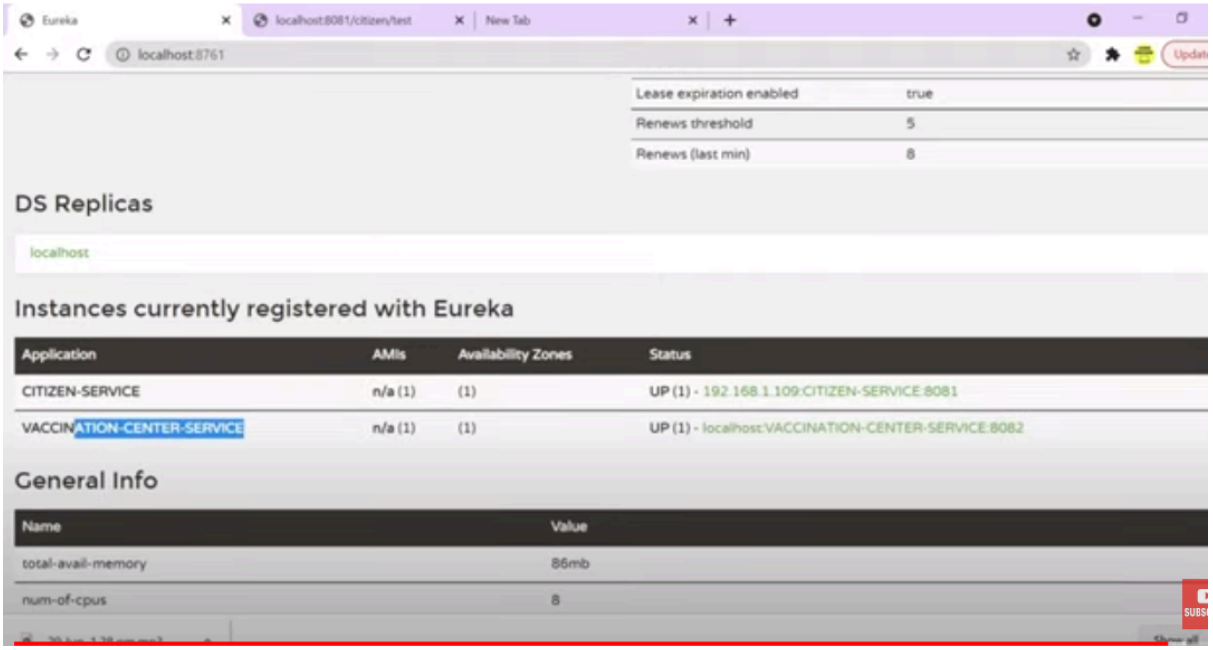
Once you start both the application , eureka and citizen



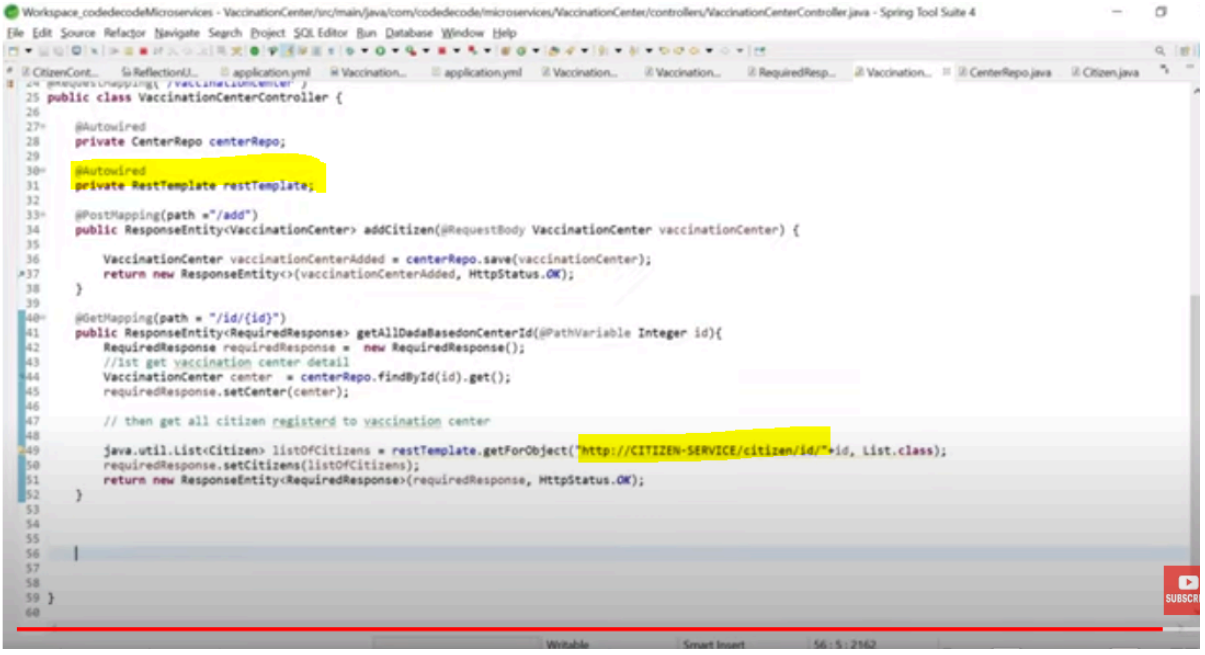
Eureka server discover client -> citizen(application name)

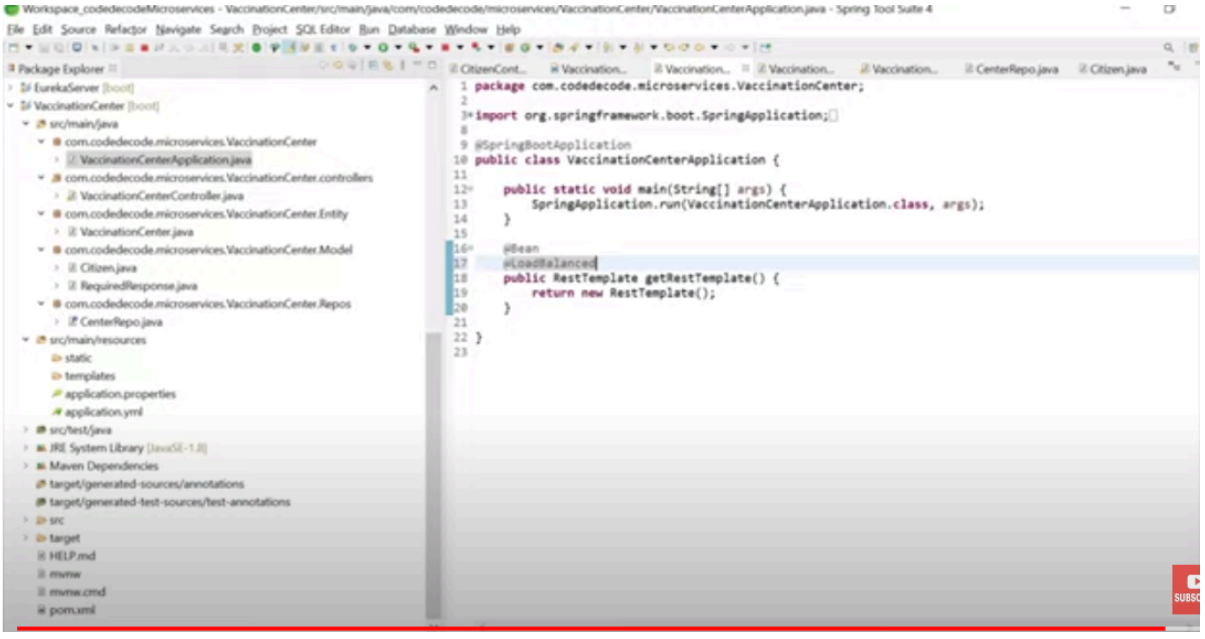


After vaccination microservice- you will see 2 entries in eureka server

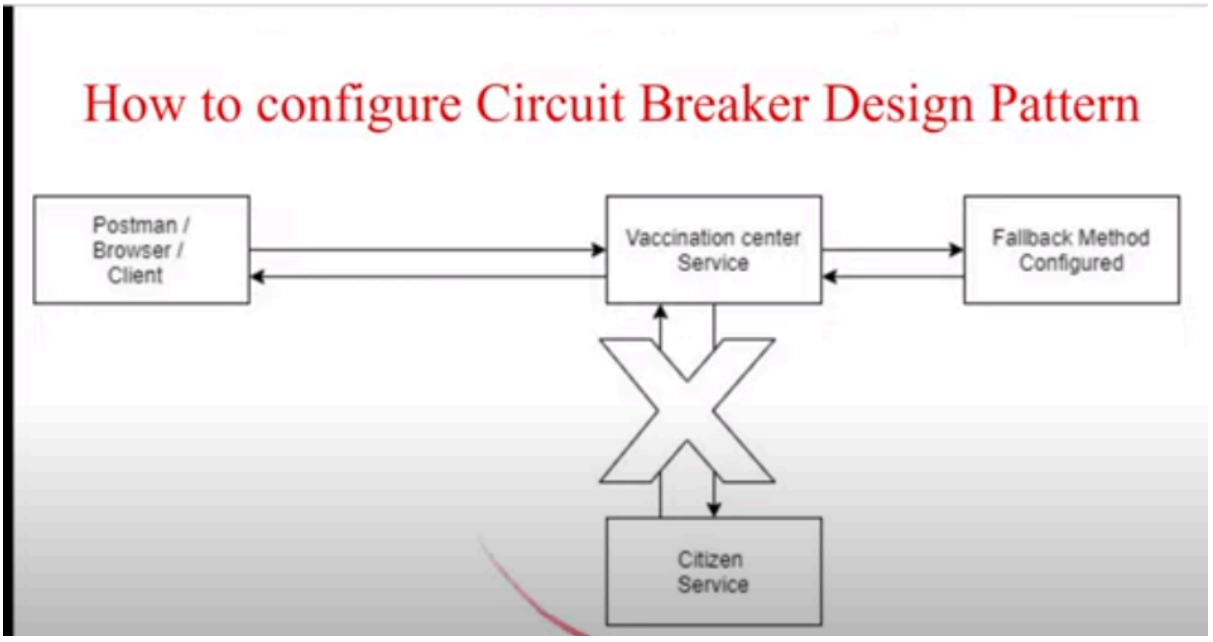


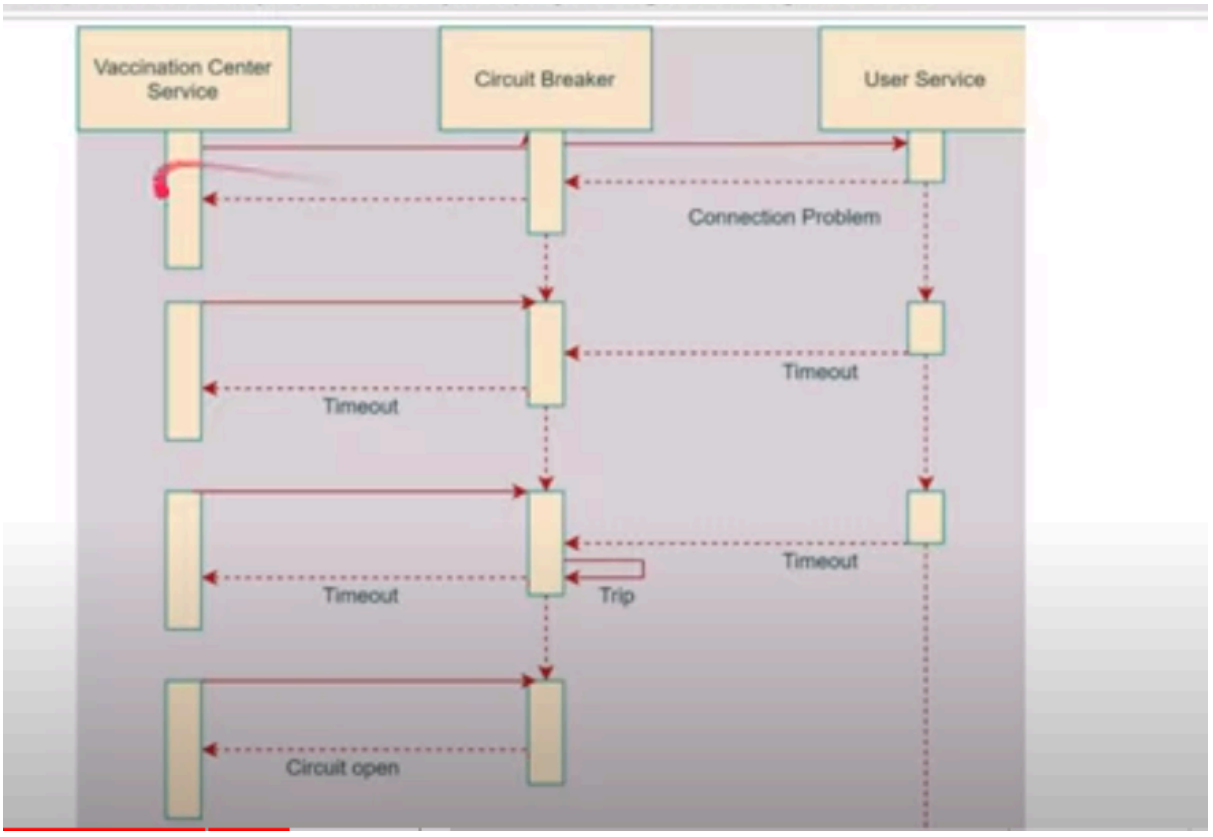
Communication:to use eureka server urls- vaccination connects wth citizen using load balance url(it can use localhost:8080/citizen/id as well)and Resttemplete should be annotated with loadbalanced



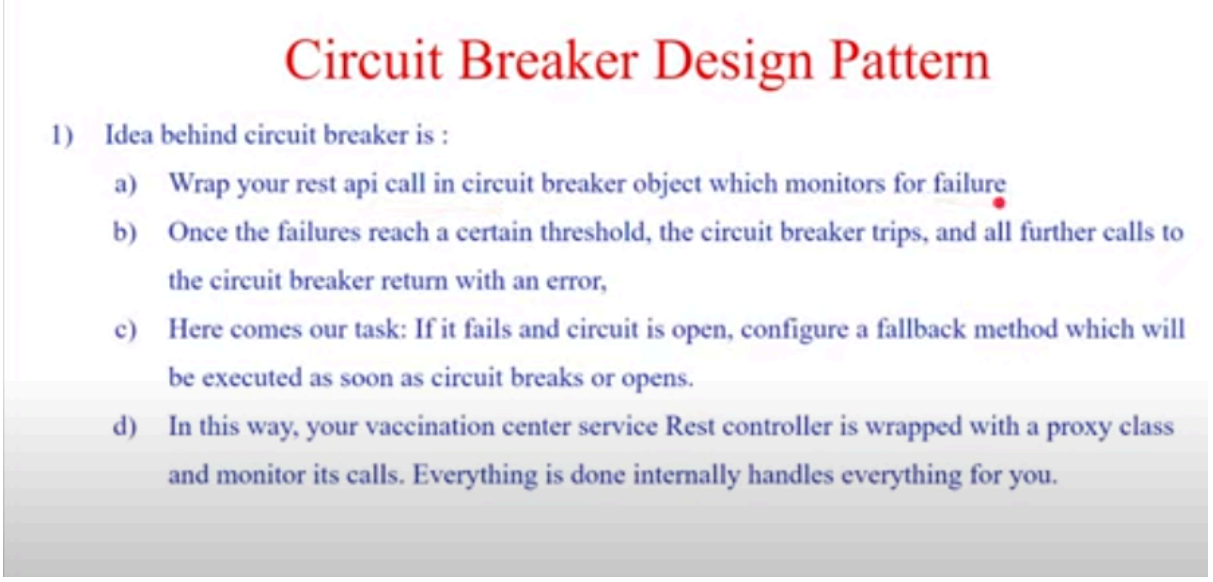


Fault tolerance





Vac service hits userservice -> conection is down
It hits- 2 times(which is config), and then it trips , circuit open and it look for fallback method - to run it without user service



Add pom dependency , enalecircuitbreaker annotation , critical code with hystriccommand
Fallback method should have same parameter as previous code

Workspace_codedecodeMicroservices - VaccinationCenter/pom.xml - Spring Tool Suite 4

File Edit Source Navigate Search Project SQL Editor Run Database Window Help

Package Explorer

- CitizenService [boot]
- EurekaServer [boot]
- VaccinationCenter [boot]
  - src/main/java
    - com.codedecode.microservices.VaccinationCenter
    - com.codedecode.microservices.VaccinationCenter.controllers
      - VaccinationCenterController.java
    - com.codedecode.microservices.VaccinationCenter.Entity
    - com.codedecode.microservices.VaccinationCenter.Model
    - com.codedecode.microservices.VaccinationCenter.Repos
  - src/main/resources
  - src/test/java
  - JRE System Library [JavaSE-1.8]
  - Maven Dependencies
  - target/generated-sources/annotations
  - target/generated-test-sources/test-annotations
  - src
  - target
  - HELP.md
  - mvnw
  - mvnw.cmd
  - pom.xml

VaccinationCenterController.java | InvocableHandlerMethod.java | *VaccinationCenter/pom.xml

```
25        <dependency>
26            <groupId>org.springframework.boot</groupId>
27            <artifactId>spring-boot-starter-web</artifactId>
28        </dependency>
29        <dependency>
30            <groupId>org.springframework.cloud</groupId>
31            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
32        </dependency>
33
34        <dependency>
35            <groupId>org.springframework.cloud</groupId>
36            <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
37            <version>2.2.8.RELEASE</version>
38        </dependency>
39
40        <dependency>
41            <groupId>mysql</groupId>
42            <artifactId>mysql-connector-java</artifactId>
43            <scope>runtime</scope>
44        </dependency>
45        <dependency>
46            <groupId>org.projectlombok</groupId>
```

Overview | Dependencies | Dependency Hierarchy | Effective POM | pom.xml

Problems | Javadoc | Declaration | Console | Progress | Debug

Workspace_codedecodeMicroservices - VaccinationCenter/src/main/java/com/codedecode/microservices/VaccinationCenter/VaccinationCenterApplication.java - Spring Tool Suite 4

File Edit Source Refactor Navigate Search Project SQL Editor Run Database Window Help

Package Explorer

- CitizenService [boot]
- EurekaServer [boot]
- VaccinationCenter [boot]
  - src/main/java
    - com.codedecode.microservices.VaccinationCenter
      - VaccinationCenterApplication.java
    - com.codedecode.microservices.VaccinationCenter.controllers
      - VaccinationCenterController.java
    - com.codedecode.microservices.VaccinationCenter.Entity
    - com.codedecode.microservices.VaccinationCenter.Model
    - com.codedecode.microservices.VaccinationCenter.Repos
  - src/main/resources
  - src/test/java
  - JRE System Library [JavaSE-1.8]
  - Maven Dependencies
  - target/generated-sources/annotations
  - target/generated-test-sources/test-annotations
  - src
  - target
  - HELP.md
  - mvnw
  - mvnw.cmd
  - pom.xml

VaccinationCenterContro... | InvocableHandlerMethod... | VaccinationCenter/pom... | *VaccinationCenterAppli...

```
1  package com.codedecode.microservices.VaccinationCenter;
2
3  import org.springframework.boot.SpringApplication;
9
10 @SpringBootApplication
11 @EnableCircuitBreaker
12 public class VaccinationCenterApplication {
13
14     public static void main(String[] args) {
15         SpringApplication.run(VaccinationCenterApplication.class, args);
16     }
17
18     @Bean
19     @LoadBalanced
20     public RestTemplate getRestTemplate() {
21         return new RestTemplate();
22     }
23
24 }
25
```

Problems | Javadoc | Declaration | Console | Progress | Debug

<terminated> VaccinationCenter - VaccinationCenterApplication [Spring Boot App] F:\Software\spring-tool-suite-4-4.9.0.RELEASE-e4.18.0-win32.win32.x86

```
at org.springframework.cloud.client.loadbalancer.LoadBalancerInterceptor.intercept(LoadBalancerInterc
at org.springframework.http.client.InterceptingClientHttpRequest$InterceptingRequestExecution.execute
at org.springframework.http.client.InterceptingClientHttpRequest.executeInternal(InterceptingClientHt
at org.springframework.http.client.AbstractBufferingClientHttpRequest.executeInternal(AbstractBuffer
at org.springframework.http.client.AbstractClientHttpRequest.execute(AbstractClientHttpRequest.java:
at org.springframework.web.client.RestTemplate.doExecute(RestTemplate.java:776) ~[spring-web-5
```

Workspace_codedecodeMicroservices - VaccinationCenter/src/main/java/com/codedecode/microservices/VaccinationCenter/controllers/VaccinationCenterController.java - Spring Tool Suite 4

File Edit Source Refactor Navigate Search Project SQL Editor Run Database Window Help

Package Explorer

- CitizenService [boot]
- EurekaServer [boot]
- VaccinationCenter [boot]
  - src/main/java
    - com.codedecode.microservices.VaccinationCenter
      - VaccinationCenterApplication.java
    - com.codedecode.microservices.VaccinationCenter.controllers
      - VaccinationCenterController.java
    - com.codedecode.microservices.VaccinationCenter.Entity
    - com.codedecode.microservices.VaccinationCenter.Model
    - com.codedecode.microservices.VaccinationCenter.Repos
  - src/main/resources
  - src/test/java
  - JRE System Library [JavaSE-1.8]
  - Maven Dependencies
  - target/generated-sources/annotations
  - target/generated-test-sources/test-annotat
  - src
  - target
  - HELP.md
  - mvnw
  - mvnw.cmd
  - pom.xml

*VaccinationCenterContr... | InvocableHandlerMethod... | VaccinationCenter/pom... | VaccinationCenterApplic...

```
    public ResponseEntity<vaccinationCenter> addCitizen(@RequestBody vaccinationCenter vacci
36
37          VaccinationCenter vaccinationCenterAdded = centerRepo.save(vaccinationCenter);
38          return new ResponseEntity<>(vaccinationCenterAdded, HttpStatus.OK);
39      }
40
41      @GetMapping(path = "/id/{id}")
42      @HystrixCommand()
```

Hystrix command key.

default => the name of annotated method. for example: ...

Returns:
command key

@HystrixCommand
public User getUserById(...) ... the command name will be: 'getUserById'

Default:
--

commandKey : String - HystrixCommand
commandProperties : HystrixProperty[] - HystrixCommand
defaultFallback : String - HystrixCommand
fallbackMethod : String - HystrixCommand
groupKey : String - HystrixCommand
ignoreExceptions : Class<? extends java.lang.Throwable>[] -
observableExecutionMode : ObservableExecutionMode - Hys
raiseHystrixExceptions : HystrixException[] - HystrixCommand
threadPoolKey : String - HystrixCommand
threadPoolProperties : HystrixProperty[] - HystrixCommand

Press 'Ctrl+Space' to show Template Proposals

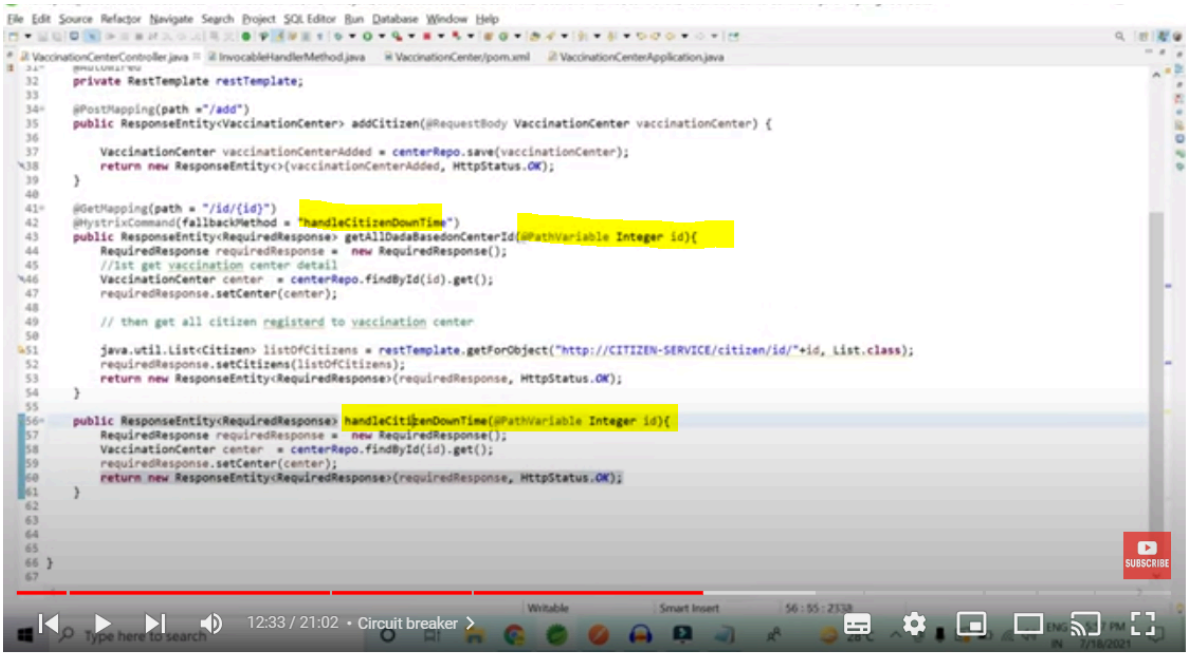
Problems | Javadoc | Declaration | Console | Progress | Debug

<terminated> VaccinationCenter - VaccinationCenterApplication [Spring Boot App] F:\Software\spring-tool-suite-4-4.9.0.RELEASE-e4.18.0-win32.win

```
at org.springframework.cloud.client.loadbalancer.LoadBalancerInterceptor.intercept(LoadBalancerInt
at org.springframework.http.client.InterceptingClientHttpRequest$InterceptingRequestExecution.exec
at org.springframework.http.client.InterceptingClientHttpRequest.executeInternal(InterceptingClien
at org.springframework.http.client.AbstractBufferingClientHttpRequest.executeInternal(AbstractBuff
at org.springframework.http.client.AbstractClientHttpRequest.execute(AbstractClientHttpRequest.ja
at org.springframework.web.client.RestTemplate.doExecute(RestTemplate.java:776) ~[spring-web-5
at org.springframework.web.client.RestTemplate.execute(RestTemplate.java:711) ~[spring-web-5.3
at org.springframework.web.client.RestTemplate.getForObject(RestTemplate.java:334) ~[spring-we
```

Writable | Smart Insert | 42 : 21 : 1617

11:16 / 21:02 • Circuit breaker



Hystrix circuit breaker And Fault Tolerance in Microservices Spring boot |Microservices in Java

Code Decode

@HystrixCommand Elements

1) fallbackMethod : Specifies a method to process fallback logic
2) threadPoolKey : The thread-pool key is used to represent a HystrixThreadPool for monitoring, metrics publishing, caching and other such uses.
3) threadPoolProperties: Specifies thread pool properties.
4) groupKey: The command group key is used for grouping together commands such as for reporting, alerting, dashboards or team/library ownership

Mazor props

@HystrixCommand Elements

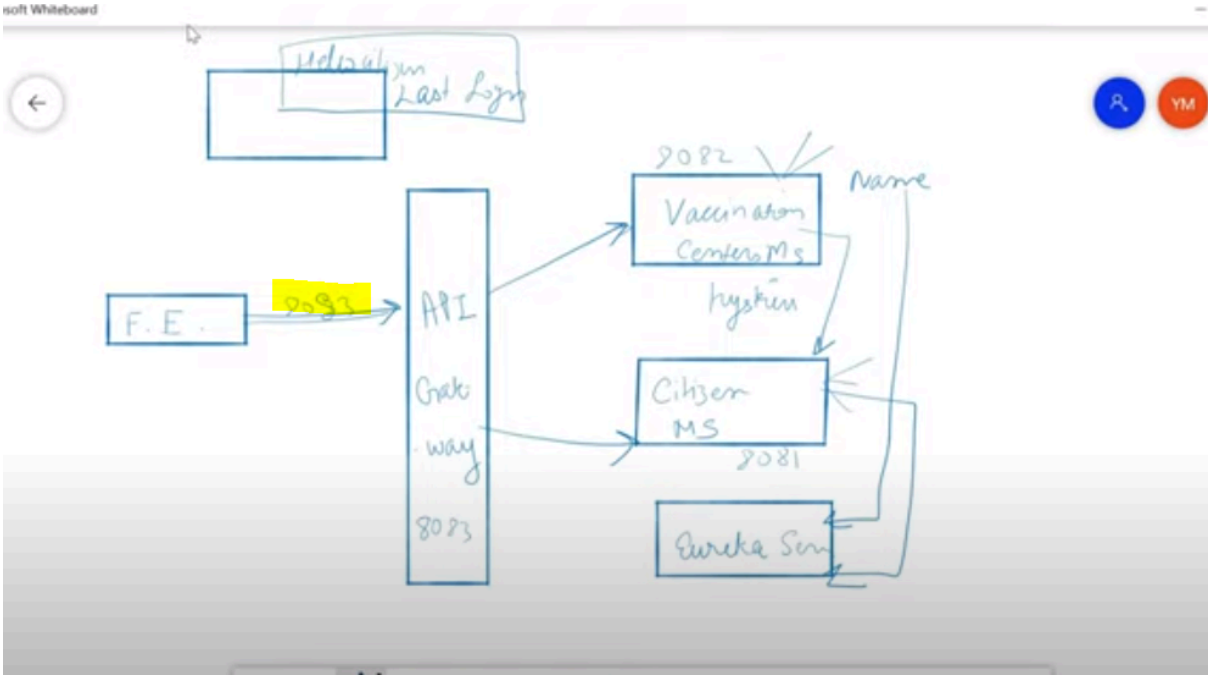
5) commandProperties: Specifies command properties like

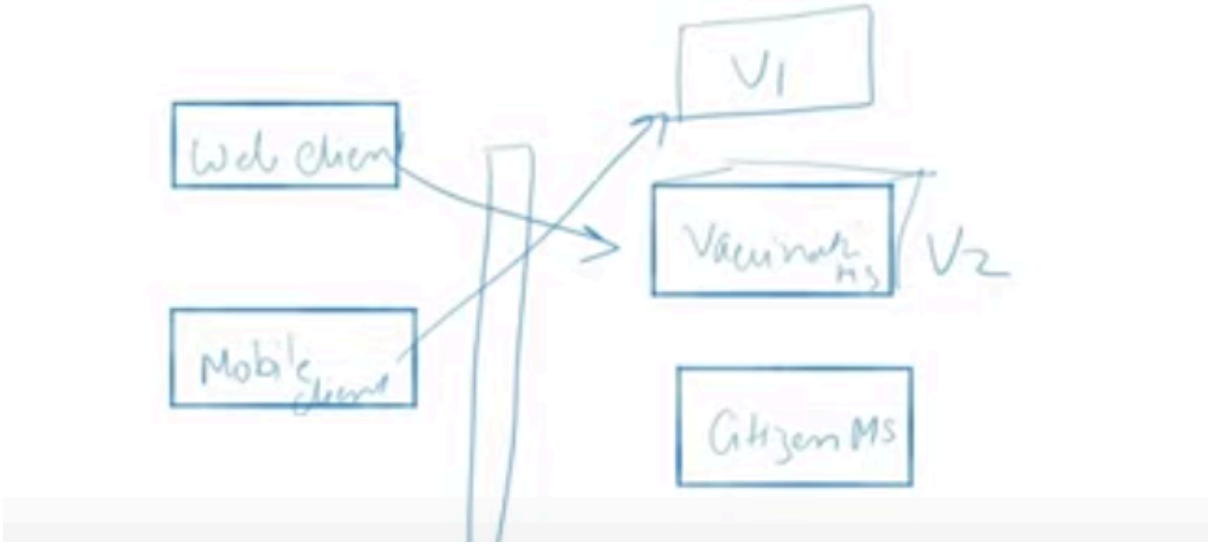
a) circuitBreaker.requestVolumeThreshold : number of requests reties before which circuit breaks

b) circuitBreaker.errorThresholdPrecentage eg 60%: If 60 % request fails out of total requests made then open the circuit

c) timeout s: Its like wait for milliseconds and if no response, break circuit

API Gateway- hide your microservcies (an abstraction from client to backend micro services) client will call api gateway- it will navigate to right service, authorization, authentication, logging and monitoring can be applied to api gateway, which will help in solving code redundancy, api gateway, supports multiple versions of a service commatible with diff clients





Different Implementations of API Gateways Available?

1) There are a number of API Gateways available and one can use any of these based on the needs.
   a) Netflix API Gateway (Zuul)
   b) Amazon API Gateway
   c) Mulesoft
   d) Kong API Gateway
   e) Azure API Gateway

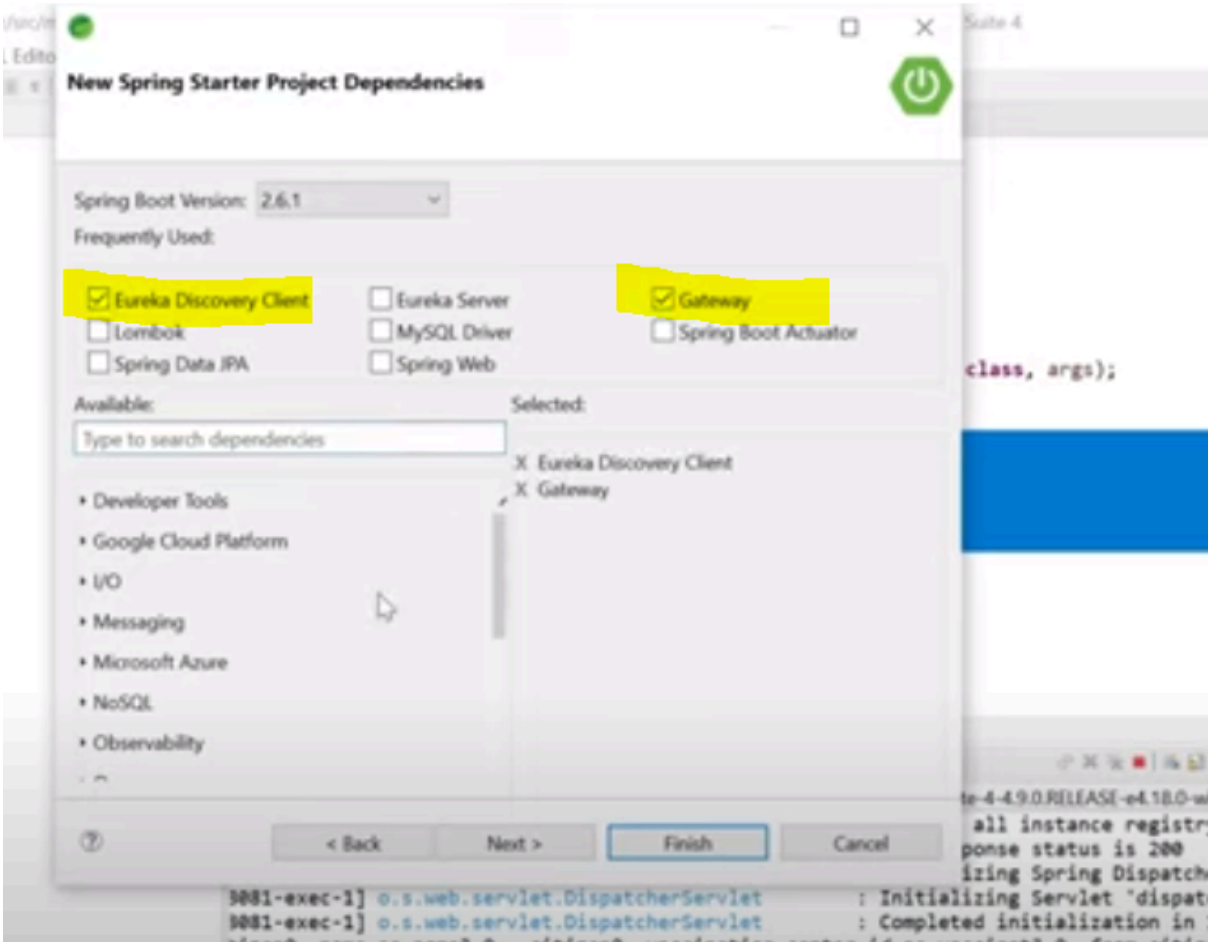
In Real world you will see most of the projects using Netflix API Gateway (Zuul), But we will be implementing API Gateway using Spring Cloud Gateway. Why??

Why Not Zuul? Why Spring Cloud Gateway.

Zuul is a blocking API. A blocking gateway api makes use of as many threads as the number of incoming requests. So this approach is more resource intensive. If no threads are available to process incoming request then the request has to wait in queue.

Spring Cloud Gateway is a non blocking API. When using non blocking API, a thread is always available to process the incoming request. These request are then processed asynchronously in the background and once completed the response is returned. So no incoming request never gets blocked when using Spring Cloud Gateway.

Spring cloud gateway implementation -



Now implement routing

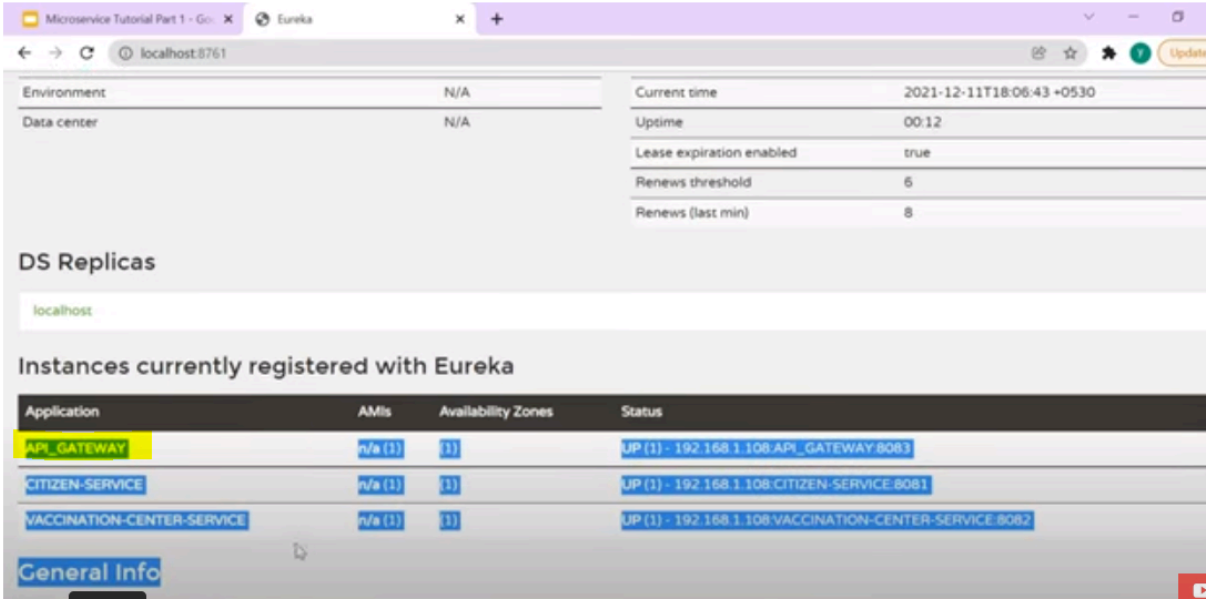
Pom- api gateway depedency

Application yaml - id (application name), uri- load balancer one or the normalone localhost:8080/.. , and predicates- path of any url having match in it

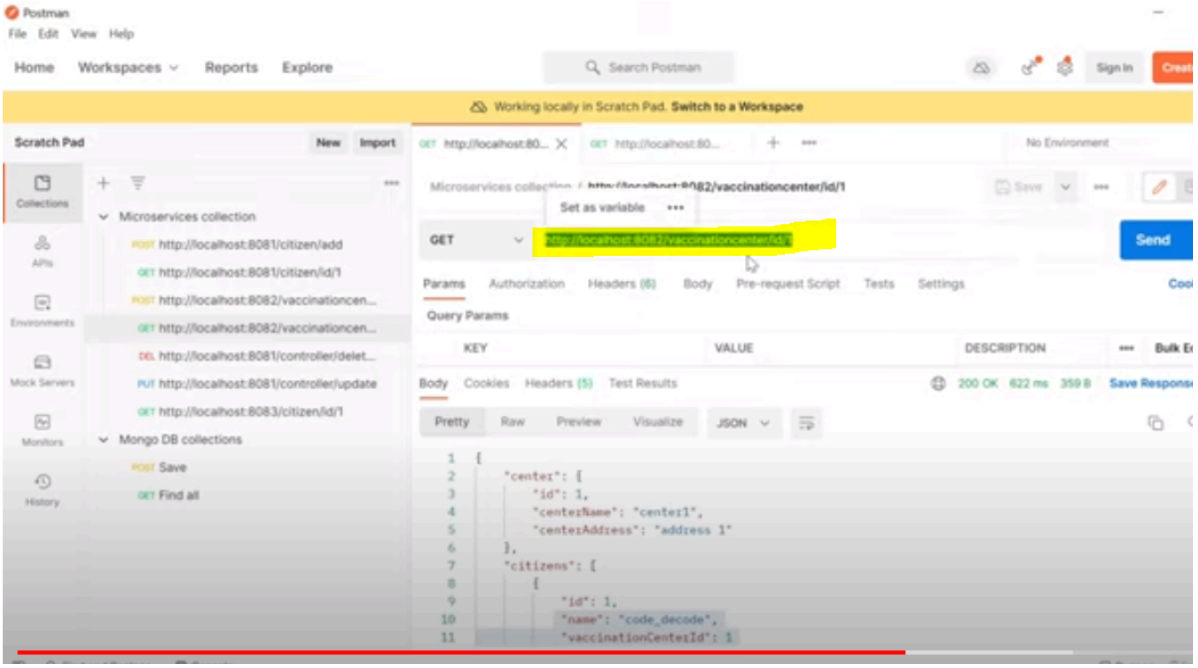
```yaml
server:
  port: 8083

spring:
  application:
    name: API_GATEWAY


  cloud:
    gateway:
      routes:
      - id: CITIZEN-SERVICE
        uri:
          lb://CITIZEN-SERVICE
        predicates:
        - Path=/citizen/**

      - id: VACCINATION-CENTER-SERVICE
        uri:
          lb://VACCINATION-CENTER-SERVICE
        predicates:
        - Path=/vaccinationcenter/**
```

```
2021-12-11 18:05:53.893  INFO 13308 --- [          main] o.s.c.g.r.RouteDefinitionRouteLocator    : Loaded F
```

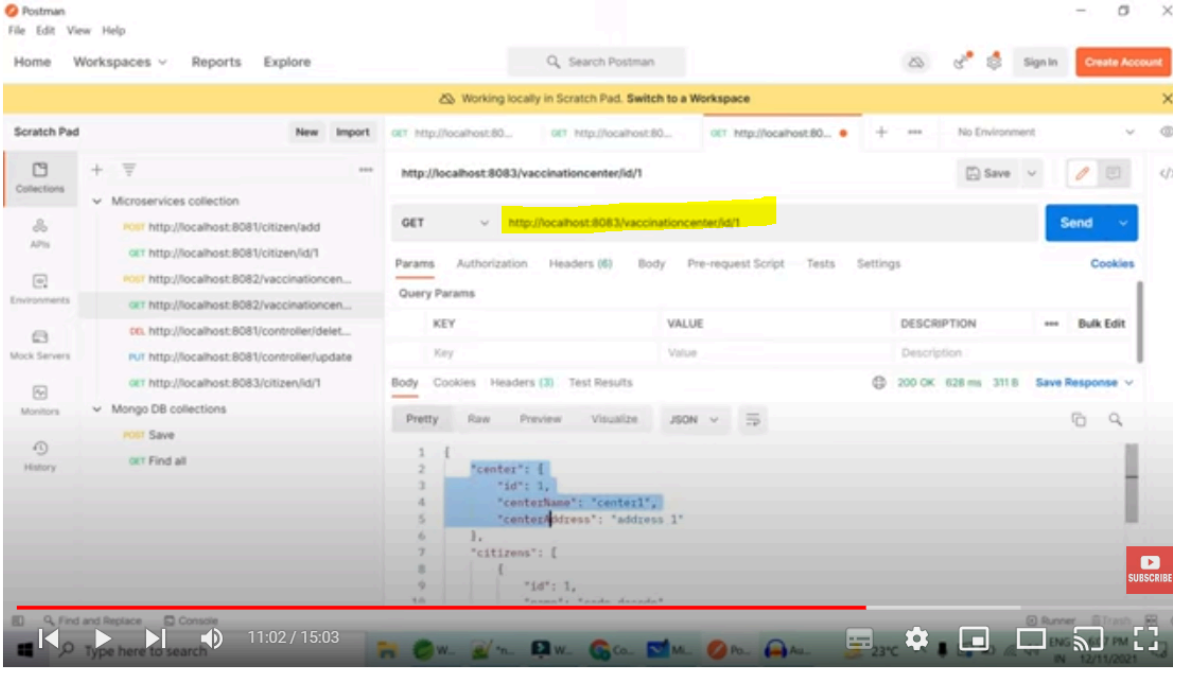
Eureka will discouve this new service



Before api gateway implementation- see the url- which is direct hit to vaccination service port: 8081



After api gateway implementation - postman url, port : 8083

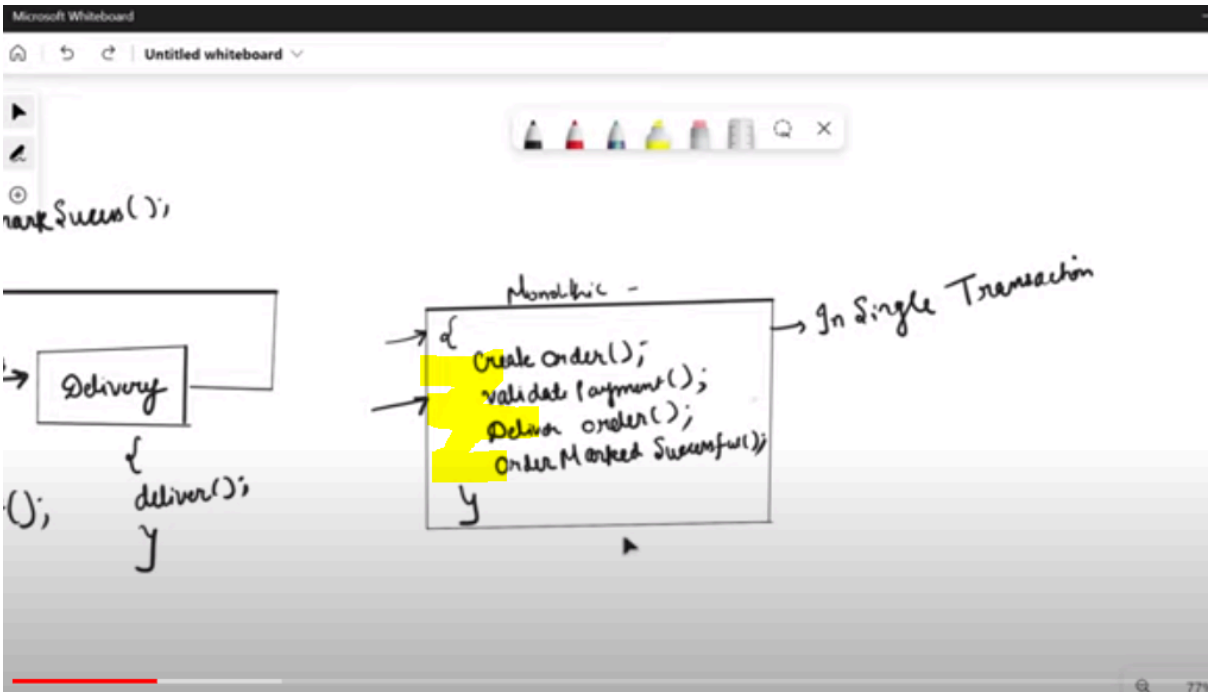


Microservices API Gateway Implementation in Spring Boot | Code Decode | Citizen Vaccination Example

SAGA - degin pattern

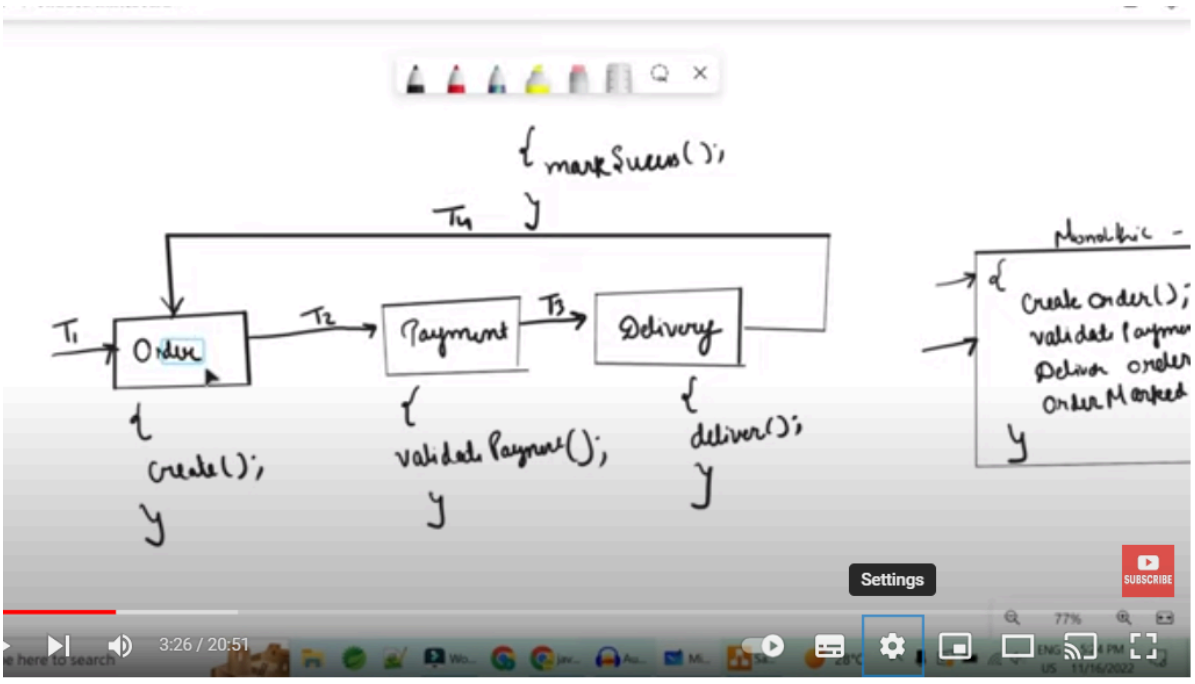
In distributed application-> when moved from monolithic to microservice

Problem



Everything happens in single transaction no problem

But in microservice- every transaction is separate - rollback will not roll back every layer

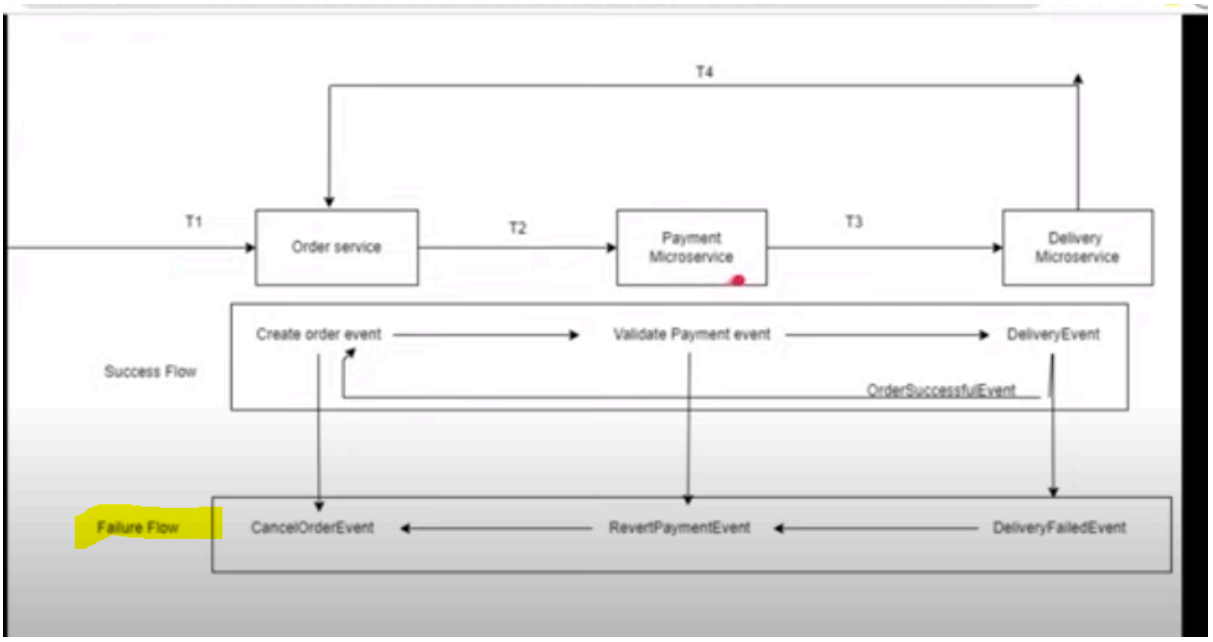


If delivery fails can not roll back the payment and other layer, distributed transaction problem.

There are 4 segments or saga(t1, t2, t3, t4)- each saga , does its task, and trigger the event to next saga

How saga design pattern handles

Check failure path:



How SAGA DP handles failure of any individual SAGA?

- The saga pattern provides **transaction management** with using a sequence of local transactions of microservices. **Every microservices** has its own database and it can able to **manage local transaction in atomic way** with strict consistency.
- So saga pattern **grouping these local transactions and sequentially** invoking one by one. Each local transaction updates the database and **publishes an event to trigger the next local transaction.**
- If one of the step is failed, than saga patterns trigger to **rollback transactions** that are a set of **compensating transactions that rollback the changes on previous microservices** and restore data consistency.

Ways to Implement SAGA?

- There are two type of saga implementation ways
  - choreography
  - orchestration

Choreograppy: mB- will send event backwork and forward, asynchronous pattern

Advantages of Choreography Saga Pattern?

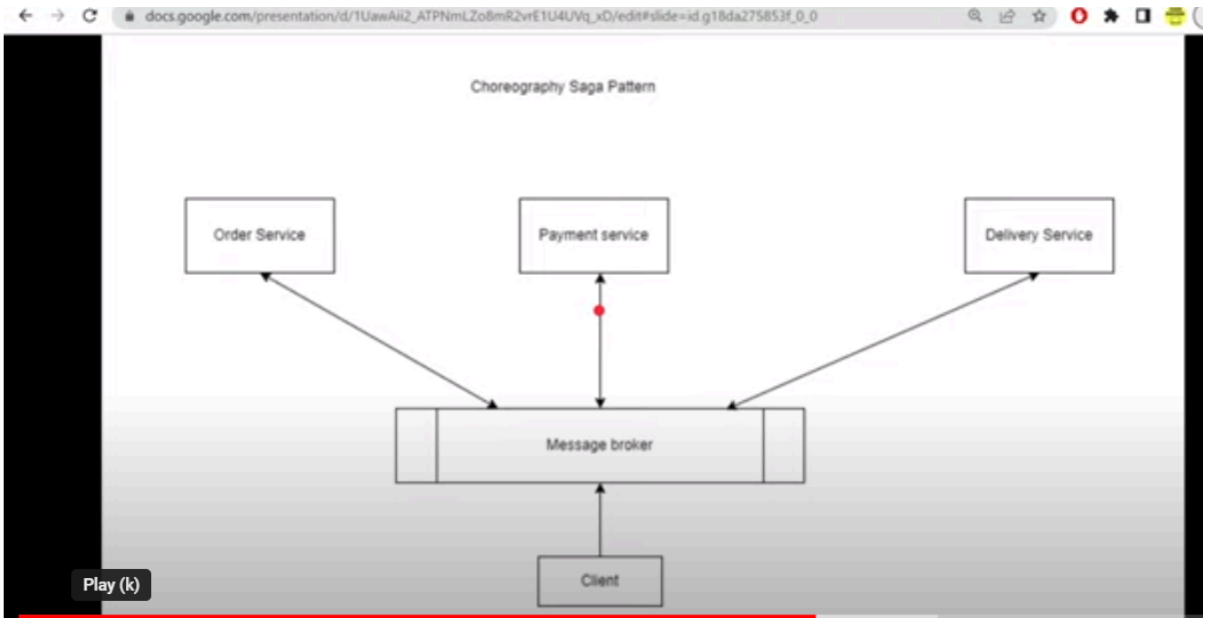
Good for simple workflows that require few participants and don't need a coordination logic

Doesn't require additional service implementation and maintenance.

Doesn't introduce a single point of failure, since the responsibilities are distributed across the saga participants.

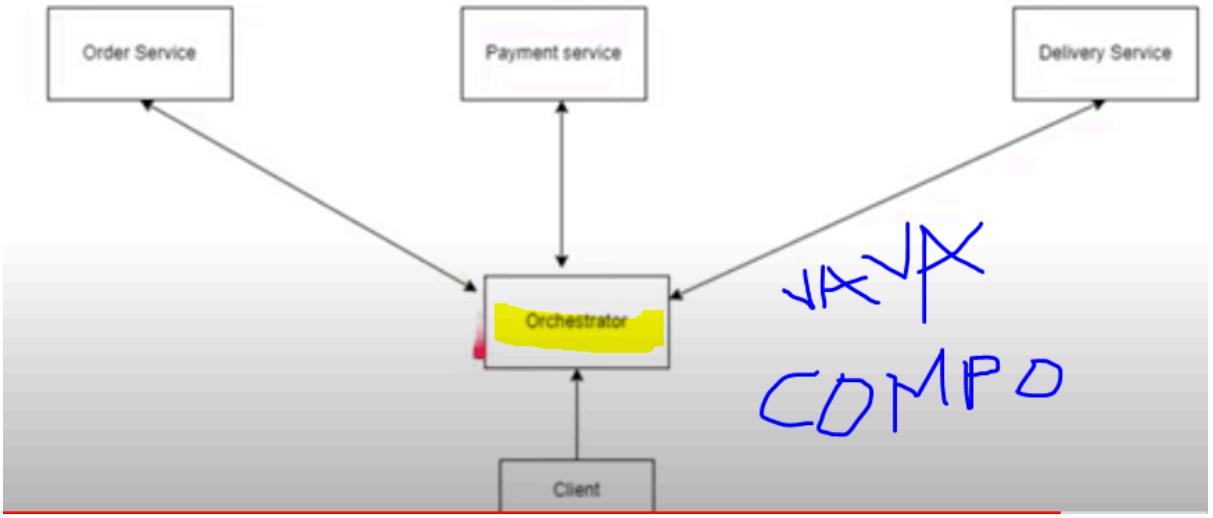
Disadvantages of Choreography Saga Pattern?

- Workflow can become confusing when adding new steps, as it's difficult to track which saga participants listen to which commands.
- There's a risk of cyclic dependency between saga participants because they have to consume each other's commands
- Integration testing is difficult because all services must be running to simulate a transaction.



Orchestration

Orchestration Saga Pattern



What is Orchestration Saga Pattern?

- Orchestration is a way to **coordinate sagas where a centralized controller** tells the saga participants what local transactions to execute.
- The saga **orchestrator handles all the transactions** and **tells the participants which operation to perform based on events**.
- The orchestrator
  - executes saga requests,
  - stores and interprets the states of each task,
  - handles failure recovery with compensating transactions.

Advantages of Orchestration Saga Pattern?

- Good for complex workflows involving many participants or new participants added over time.
- Suitable when there is control over every participant in the process, and control over the flow of activities.
- Doesn't introduce cyclic dependencies, because the orchestrator unilaterally depends on the saga participants.
- Saga participants don't need to know about commands for other participants. Clear separation of concerns simplifies business logic.

Disadvantages of Orchestration Saga Pattern?

- Additional design complexity requires an implementation of a coordination logic.
- There's an additional point of failure, because the orchestrator manages the complete workflow.

How you handle denial of service between microservices
What do you mean by idempotent verbs in http
How to handle versions in rest Api
URI Versioning

```
http://localhost:8080/api/v1/resource
http://localhost:8080/api/v2/resource
```

x-cutom-header = v1 or v2

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class CustomHeaderController {

    @GetMapping("/resource")
    public ResponseEntity<String> getResource(@RequestHeader("X-Custom-Header")
String customHeader) {
        return ResponseEntity.ok("Custom Header Value: " + customHeader);
    }
}
```

Media type - Accept Header versioning

```
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class VersionController {

    @GetMapping(value = "/resource", produces = { "application/vnd.myapp-v1+json",
"application/vnd.myapp-v2+json" })
    public ResponseEntity<String> getResource() {
        return ResponseEntity.ok("API Resource");
    }
}
```

```
"Accept: application/vnd.myapp-v1+json"
http://localhost:8080/api/resource
```

OR

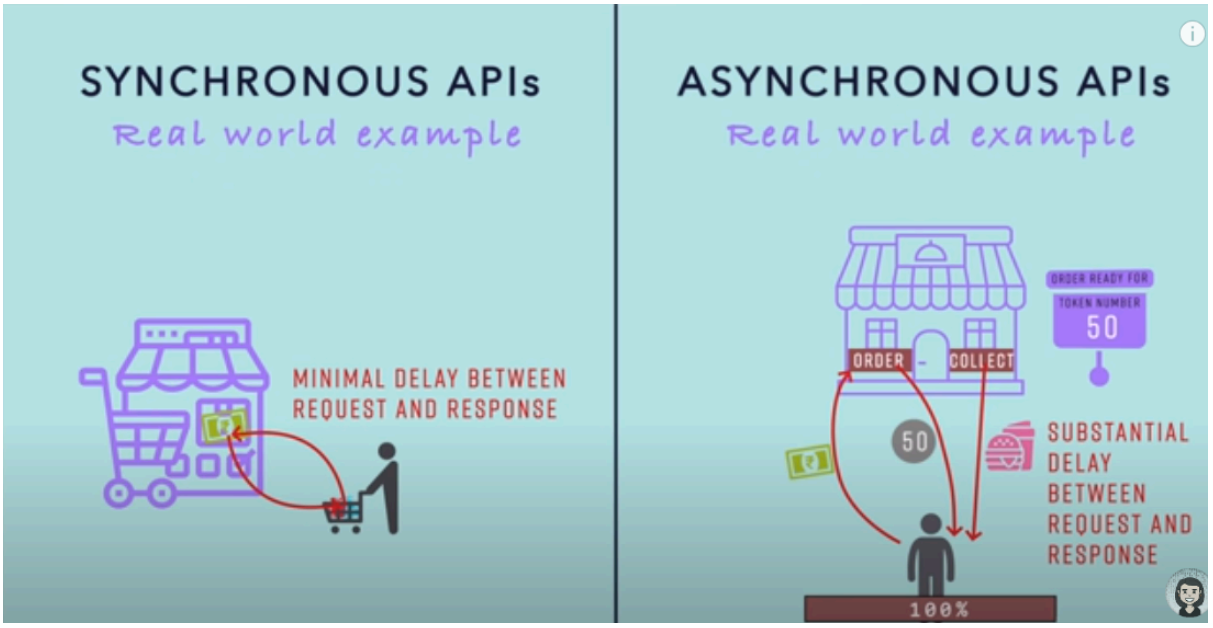
```
"Accept: application/vnd.myapp-v2+json"
http://localhost:8080/api/resource
```

Requestparam - > version

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class VersionController {

    @GetMapping("/resource")
    public ResponseEntity<String> getResource(@RequestParam("version") String version) {
        return ResponseEntity.ok("API Version: " + version);
    }
}
```

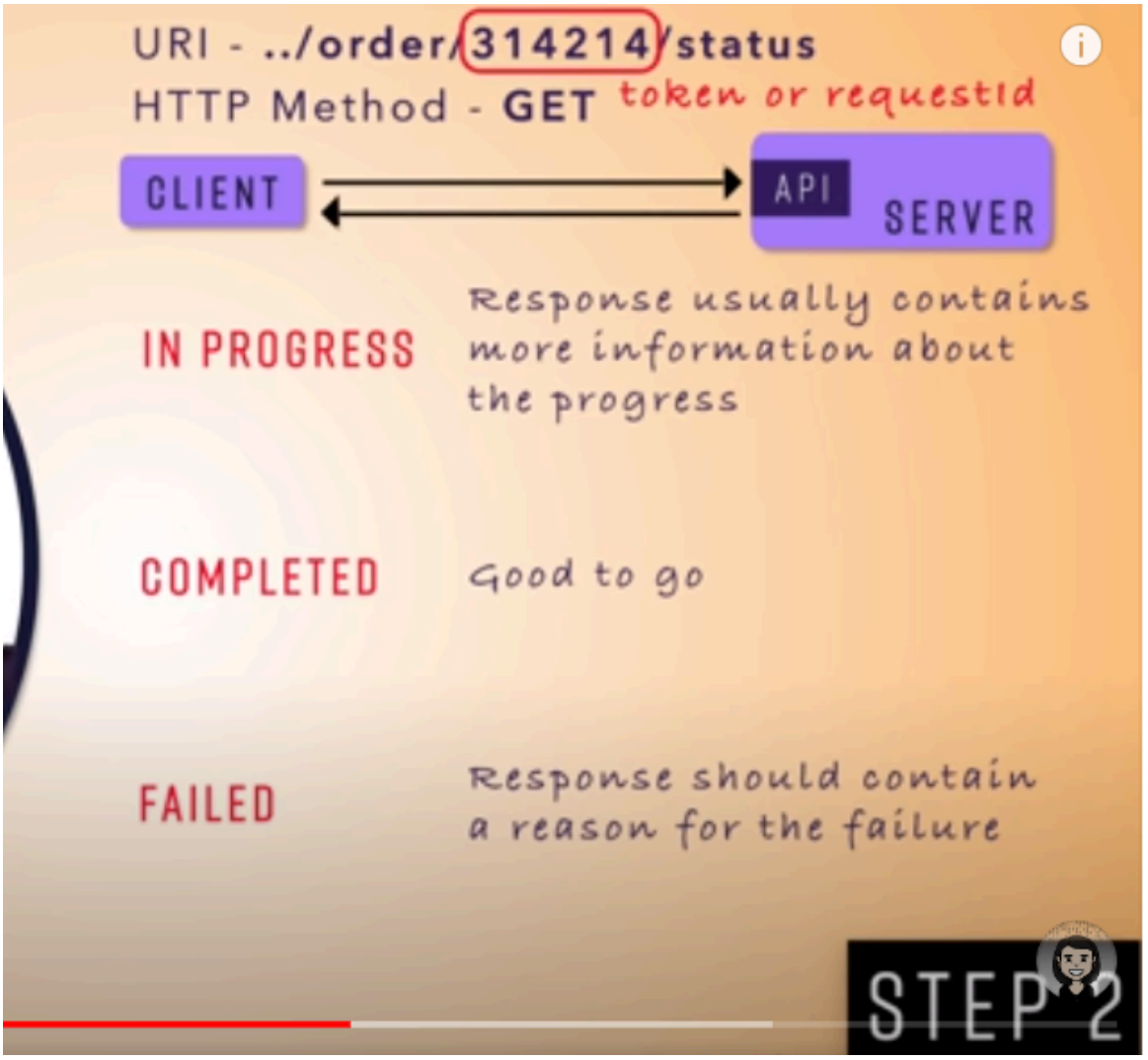
```
http://localhost:8080/api/resource?version=v1
```

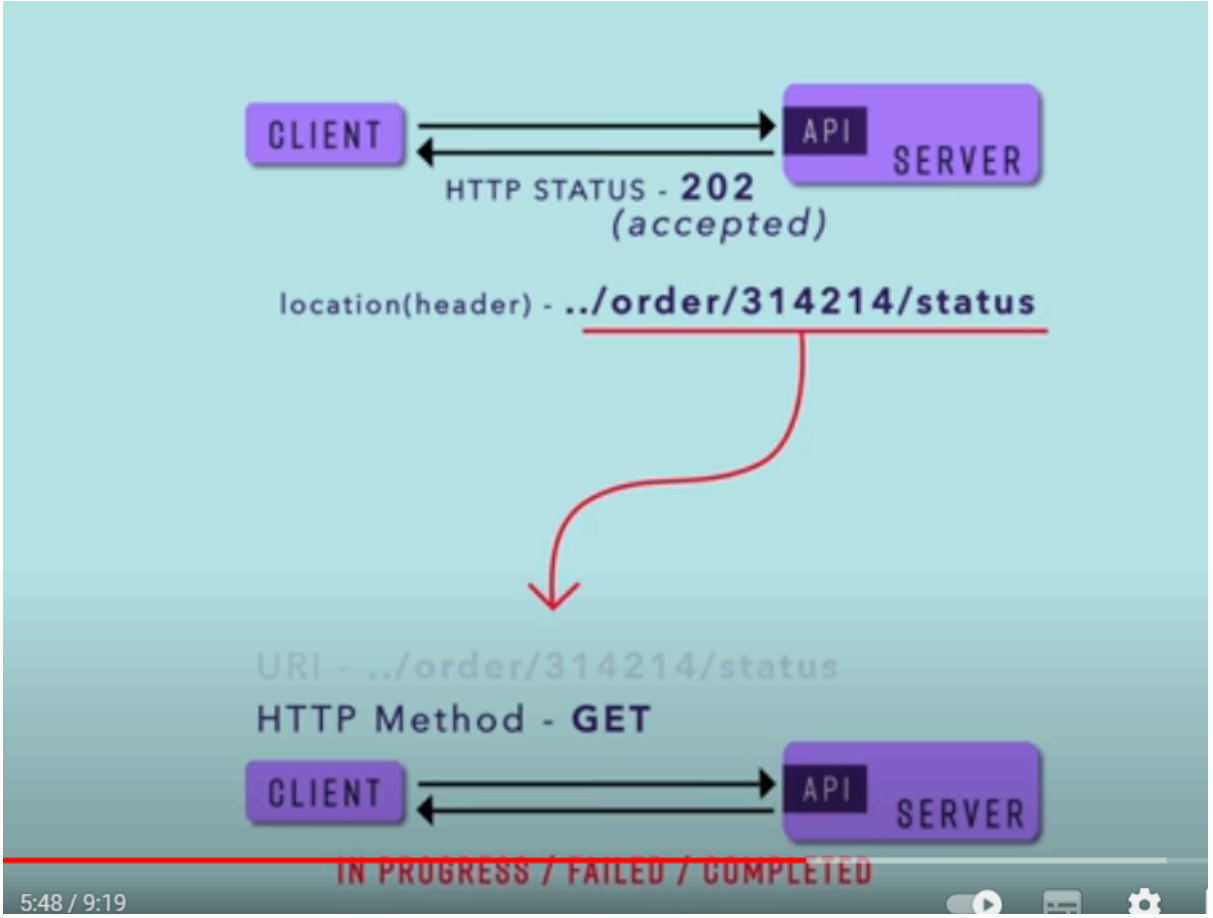
How to handle big responses in rest api

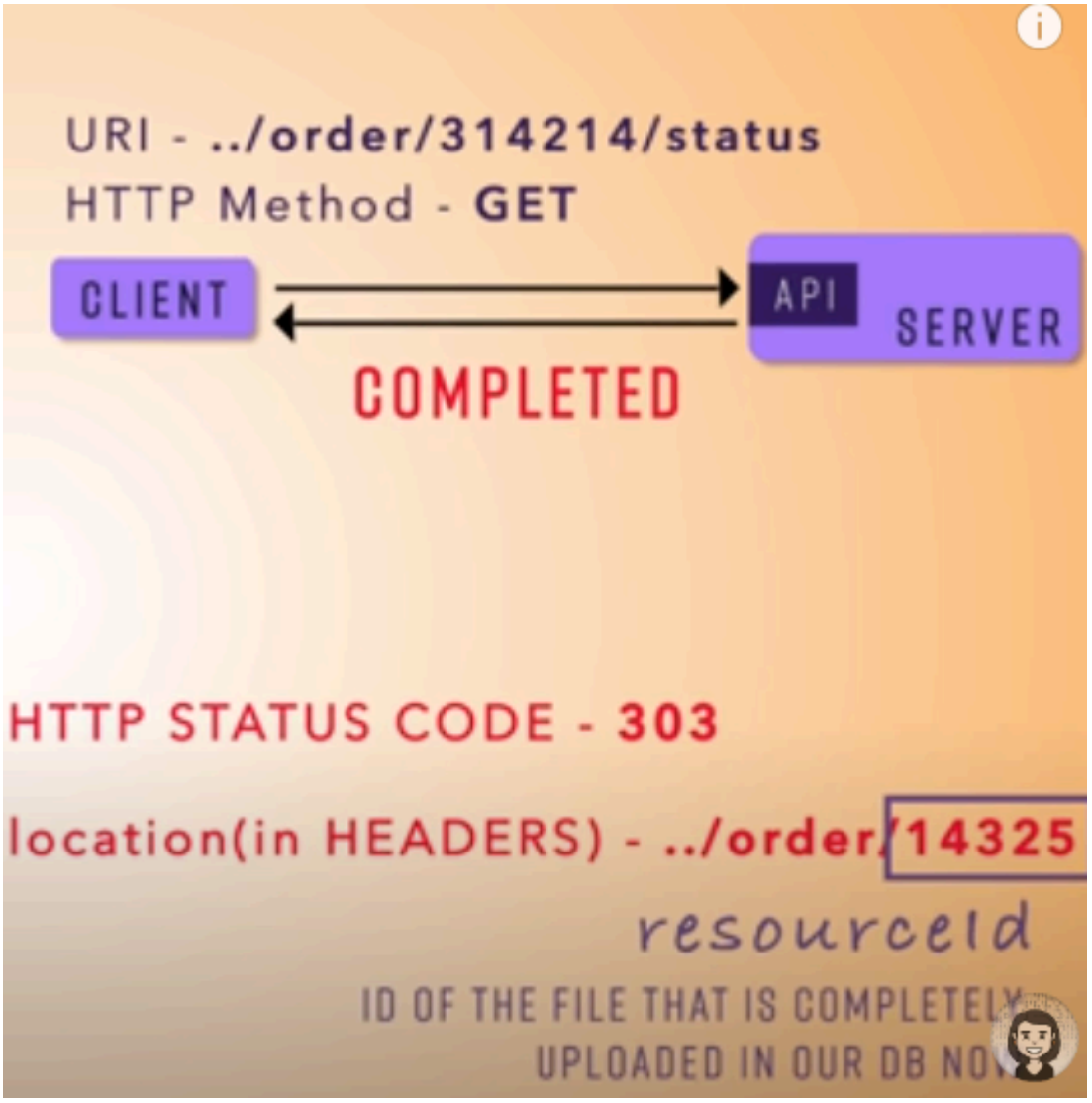


Or huge data which can not be completed in milliseconds

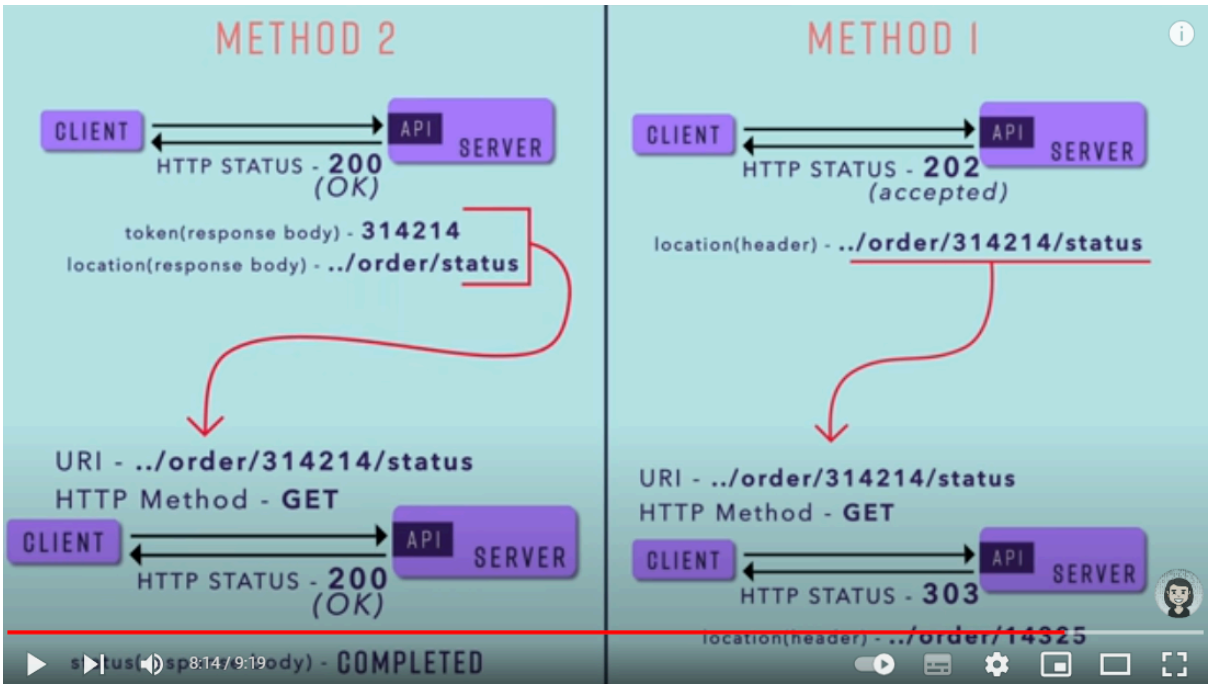
Step 1 -> post/put api will respond with 202 (accepted) status code + a new location url(get api -> …/order/50/status) to check the the status of completion
Step 2 -> run this location url to fetch the status of request API


URI - ../order/314214/status
HTTP Method - GET
token or requestId
CLIENT
API
SERVER
IN PROGRESS
Response usually contains more information about the progress
COMPLETED
Good to go
FAILED
Response should contain a reason for the failure
STEP 2


CLIENT
API
SERVER
HTTP STATUS - 202
(accepted)
location(header) - ../order/314214/status
URI - ../order/314214/status
HTTP Method - GET
CLIENT
API
SERVER
IN PROGRESS / FAILED / COMPLETED
5:48 / 9:19



2 methods to implement



How to handle big payload in rest api

Java 8 Vs java 11

How is G1 better collector

Java 8 features

How to stop task already submitted in threadpool

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ThreadPoolDemo {

    public static void main(String[] args) {

        ExecutorService executor = Executors.newFixedThreadPool(2);

        Runnable processor = new MessageProcessor(2);
        executor.execute(processor);

        Runnable processor2 = new MessageProcessor(3);
        executor.execute(processor2);

        Runnable processor3 = new MessageProcessor(4);
        executor.execute(processor3);


    }

}
```

```
pool-1-thread-1 [RECEIVED] Message = 2
pool-1-thread-2 [RECEIVED] Message = 3
pool-1-thread-1 (DONE) Processing Message = 2
pool-1-thread-2 (DONE) Processing Message = 3
pool-1-thread-1 [RECEIVED] Message = 4
pool-1-thread-1 (DONE) Processing Message = 4
```

```java
        executor.execute(processor2);

        Runnable processor3 = new MessageProcessor(4);
        executor.execute(processor3);

        Runnable processor4 = new MessageProcessor(4);
        executor.execute(processor4);

        executor.shutdown(); // Rejects any new tasks from being submitted. Gracefully shuts down the service

        try {
            executor.awaitTermination(20, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        System.out.println("submitted all tasks...");


    }
```

```
pool-1-thread-1 (DONE) Processing Message = 2
pool-1-thread-2 (DONE) Processing Message = 3
pool-1-thread-1 [RECEIVED] Message = 4
pool-1-thread-2 [RECEIVED] Message = 4
pool-1-thread-2 (DONE) Processing Message = 4
pool-1-thread-1 (DONE) Processing Message = 4
submitted all tasks...
```

Cancellation via Future: If you submit tasks to the thread pool using `ExecutorService.submit()`, you can use the returned `Future` object to attempt to cancel the task. Keep in mind that this will only work if the task cooperates and periodically checks the `isCancelled()` method of the `Future`. Example:

java

Copy code

```
ExecutorService executor = Executors.newFixedThreadPool(5);
Future<?> future = executor.submit(new MyTask());

// To attempt to cancel the task:
future.cancel(true); // true
```

Note that calling `cancel(true)` may interrupt the running thread, which could lead to exceptions if the task isn't designed to handle interruptions.
Thread Interruption: You can use the `Thread.interrupt()` method to interrupt a running thread. However, this should be done carefully, as it can lead to improper cleanup if not handled correctly. Your task code needs to explicitly handle interruptions by checking `Thread.currentThread().isInterrupted()` periodically and performing appropriate cleanup and stopping.

java

Copy code

```
public void run() {
    while (!Thread.currentThread().isInterrupted()) {
        // Your task's work
    }
}
```

To stop the task, you would call `thread.interrupt()`, where `thread` is the thread running the task.

Different ways to avoid deadlock

Design concurrent HashMap with using current concurrent hashmap

Problem Solving questions

String compress(string){}

Input aaababbrxz output a3bab2rxz

Spring boot

Hey guys, welcome to Code Decode! In this video, we will be looking into mostly asked Springboard interview questions for three-plus years of experienced people. So, let's get started and look at these interview questions. Please like, share, and subscribe to support us so that this video reaches more of the people who are preparing for interviews like you.

What is the difference between Spring Boot and Spring Framework? This is a very basic question to check your knowledge of the basics of the Spring Framework.

Spring Boot is a framework that simplifies the development of Spring-based applications. Remember, Spring Boot is built on top of Spring. It was difficult to create and develop an end-to-end application with the Spring Framework. Hence, Spring Boot came into the picture to reduce the time to market. Spring Boot provides a number of features that make it easier to create standalone, production-ready applications.

What are those features that enable Spring Boot to be able to do that? The features are:
1. Auto-configuration: In Spring Framework, either through Java-based configuration or XML-based configuration, you have to explicitly define which bean to create, how to create, and where to create. In Spring Boot, it automatically does it with one single annotation. Spring Boot automatically configures many of these Spring features that you would typically need to configure manually.

2. Starter dependencies: Spring Boot provides starter dependencies like, for example, for Spring Security. Just add one single Spring Security starter dependency, and you have implemented security successfully. You will see a login page that will not allow you to log in without a proper ID and password. That's the beauty of starter dependencies that Spring Boot has.

3. Embedded servers: Spring Boot also provides embedded servers such as Tomcat and Jetty, so you don't have to explicitly configure a separate web server. It comes as a package.

4. Metrics and health checks: These help you check the health and winterizes of your Spring Boot application in production and monitor your application in production, which is not available and possible in the Spring Framework standalone.

So, Spring Boot is built on top of the Spring Framework, but it takes a different approach to development. Development in Spring is difficult, while in the case of Spring Boot, it is very easy because it is opinionated. That means Spring Boot puts some assumptions as to how you want to develop your applications. For basic default configuration, everything will be done by Spring Boot itself. So, it is easier to get started with Spring Boot, but the problem is, since it assumes a lot, it creates a lot of default configurations. Your flexibility is very much limited. If you want to have your application completely in your hands and in control, go for Spring Framework. If you are okay with Spring Boot taking some control of your default configuration, then go for Spring Boot.

So, this is a very short summary of the difference between Spring Boot and Spring Framework. Opinionated means to give you custom default configuration. Spring Boot does it; Spring Framework doesn't. Spring Boot gives you maximum things automatically configured as a default configuration in your application. Auto-configuration is possible in Spring Boot, not in Spring Framework. Starter dependencies are not in Spring Framework, but you have them in Spring Boot. Embedded servers like Tomcat and Jetty are available in Spring Boot, not in Spring Framework. You have to explicitly do it. Metrics and health checks are automatically given as actuator endpoints with Spring Boot, which is not there in the Spring Framework.

But the only disadvantage I can find in Spring Boot is the lack of flexibility. Because most of the things are done by Spring Boot, you have less to do. Though you can customize it later on with properties files and all, much of the things are assumed and opinionated by Spring Boot, and you don't have much left on those default configurations. That is why Spring Boot is less flexible. Also, Spring Boot focuses on convention over configuration. With Spring Framework, you will keep on doing some configuration with XML files or Java configuration files. What Spring Boot says is, "I'm much over this configuration. I'll give you the maximum configuration by doing it myself. You don't have to explicitly do anything." So, it minimizes the need for explicit configuration and reduces the overall configuration efforts.

While, on the other hand, Spring Framework requires a lot more explicit configuration, either through XML or Java-based configuration files. That's the biggest advantage I find with Spring Boot.

Now, in general, Spring Boot is a good choice for developers who want to get started with Spring quickly and easily. But if you need more flexibility, then you should use Spring Framework directly. Remember, Spring Boot is over and above Spring Framework only. So, rather than going one level up, if you need more flexibility, go one level down and do every configuration, every heading yourself. Or else, give that task to Spring Boot and have your time to market various. So, it reduces configuration overhead and improves your productivity. You don't have to put your head into how to manage version dependency management. You don't have to do anything; Spring Boot handles it for you. So, you can focus on your development work and increase your productivity in the project, rather than thinking about XML-based configurations in Spring Framework.

Now, how would you configure a database connection in Spring Boot? So, you're developing an application in Spring Boot. Great, wow! But now, how will you connect this application to a database? Because at the end, you have to get some data from the database, add some

data, do CRUD operations on the database, right? So, what are the ways to configure a database connection in Spring Boot?

There are typically two ways to do it:
1. Using `application.properties` file: Configure your data source using the `application.properties` file.
2. Using a configuration class: Create a Java class annotated with `@Configuration`, which says, "I am going to be a source of beans for you for the entire lifecycle of your application."

All I have is the bean created inside me, so you can use that and create the bean in the Spring container. So, these are the two ways, and to create a database connection, you need to configure the data source.

The only thing that these two methods do is configure your database. I'll give you one quick example if you can see here. This is an example of doing it through the properties file. So, what we have done is we have given properties for the Spring data source URL, ID, password, username, driver class name. All these things. With the `application.properties` file, the most common way is this. This is the most common way. With this, what happens is you don't have to change anything in the Java class. You just change your properties, and you're good to go. You don't have to modify anything in the source code. It's just the properties that you need to change in case later on the URL, username, or password changes for any of the databases.

So, this is the most common way to configure a database connection. You specify the connection properties in the `application.properties` or `application.yaml` file, and Spring Boot will automatically configure the database connection. So, using this, that's why I say Spring Boot is smart enough. What it will do is it will automatically configure the data source because it sees that you've given a URL. It understands that you want to connect to localhost, port 330

In a .XML file, add the starter cache dependency of Spring Boot itself. As soon as you do that, you will have one of The annotations: `@EnableCaching` available to you. You can use this annotation on the main class (Spring Boot application root class). So, when you annotate this with `@EnableCaching`, which I won't get because I don't have the pom.xml upgraded with the starter cache dependency. If you need to have a live demo on how to implement Spring Boot Starter Cache, just let me know in the comment section. I have many questions to cover, so I'll quickly go through this.

Now, the next thing that you need to do is to enable caching on your main or the root application. And as soon as you do that, you can also configure the cache manager. By default, you don't need to do it because Spring Boot provides you a default cache manager. But even if you are not happy with the default configurations of the cache manager, Spring Boot gives you the capability to configure it or customize it. So, you can define a custom cache manager. How would you do that? You can just annotate a class with `@Configuration` and provide a bean of cache manager. I told you Spring Boot gives maximum things as the default one, but if you don't want to use it, you can always customize it. So, what you need to do is just give a cache manager bean after customizing here after giving a code here.

To use the default caching manager, you don't have to do anything. However, if you want to define your custom cache manager, create a bean of type `CacheManager` in your configuration class. Spring Boot will ignore its default cache manager and will use the cache manager that you have defined in the configuration class with the custom properties. So, what you have done until now is implemented a data dependency of cache and enabled caching in your whole application. You have given your own cache manager if needed.

The last thing is to use the cache, that is, to put the things into your cache and to retrieve things from the cache. So, how will you get and put things into the cache? You have multiple methods like `@Cacheable`, `@CachePut`, `@CacheEvict`, and more. These will enable caching for a specific method. So, suppose this is your service, you are getting some data from the database. Now you want to cache it. So, what you will do first, you will annotate that method with `@Cacheable` and give the cache name, say `employeeCache`. So, whenever you get the data of employees from the database and you are returning it, before that Spring Boot will automatically create that entry in Cache with that key.

For example, you're getting the data for employee ID 1. So, automatically you are getting the data out of the database for the first time. But the second time when you try to get the data again for employee ID 1, the database will not be hit. It will be fetched from the cache, specifically the `employeeCache`. So, that's how you make use of caching to get the data in a quick and effective manner, and you don't have to hit the database again and again. Also, you have multiple things like time to live, eviction policy, and many such things you can configure here in the Spring Boot itself, either in the properties file or with configurations.

So, it makes sure that subsequent invocations to get employee with ID 1 will not be fetched from the database but from your cache, `employeeCache`. So, additionally, you can also configure time to live, eviction policy, cache name, and all those things in the properties itself. So, that was all about how to implement caching in Spring Boot.

Now, can you explain the different types of caching mechanisms? So, we have several different types of caching mechanisms available in Spring Boot. The common ones are in-memory cache. Remember, this is the default cache implementation given by Spring Boot to you that is in-memory cache using concurrent hashmap. This is a very basic caching mechanism you can use in your application, where the cached data is stored in memory itself. This allows for faster access and retrieval. This type of caching is suitable for applications where the cached data is not expected to change frequently. Otherwise, you have to go and change the concurrent hashmap implementation again and again in-memory.

Also, if you have a very small set of data, go for in-memory caching because all the data is stored in the memory and memory is limited, you know that. So always you should go for such where your data can fit in the given memory and your memory doesn't go out of space. This is not a very practical solution. I will never suggest in-memory caching for any production-based use. If you want to test something, go for it, but not for the production-based applications.

The second is distributed caching. This is something you always work with in the real-time projects because you have multiple environments and multiple regions. Basically, your

application when deployed through the cloud also involves a distributed environment where your data is present. Distributed caching allows you to store the data across multiple nodes in a distributed environment. It allows you for scaling the caching capacity and provides high availability. Now, a very good example of such caching frameworks are Redis and Hazelcast.

The third is external caching. External caching involves external cache services. For example, I give you, if you're using Hibernate, we have implemented Ehcache using Hibernate. Ehcache is actually an external caching mechanism where you use Ehcache, which seamlessly gets integrated with your Spring Boot application. So, it's the beauty of the Spring Boot application which provides you end points to be able to seamlessly integrate with such external caching mechanisms if you do not want to use the default hashmap-based Spring Boot cache mechanism.

Mean cache and Ehcache are those such external examples of cache, and these services provide dedicated caching capabilities that can be integrated with Spring Boot applications. The fourth is page caching. Page caching involves the caching of the entire HTML page of a website or a web application. Remember when you hit the Wikipedia when you hit that page again and again, remember that page is a static content. Now, when you every time go and hit that website, you don't always get the data from their database or data source. You might get the data from cache also because Wikipedia knows that the content they provide are static. That means they are rarely there in any chance that they are going to change.

They are going to be mostly static over the whole lifespan of that particular web page. Thus, for static content which doesn't change frequently, you can always cache the whole entire page and make it available to the people who are trying to access it again and again to increase the performance and to reduce the database or data sources. So, Spring Boot application also supports such page caching through interaction with web servers like HTTP servers or Nginx. Page caching is only feasible when you have a lot of static data, like Wikipedia.

The last and the one which you will always come across in your Spring Boot application lifecycle is query caching. Whenever you hit a database with a query, the query itself is cached. Query caching is used to cache the result of database queries, improving performance for database-heavy applications. Spring Boot integrates with ORM frameworks like Hibernate and JPA to provide query caching capability out of the box. So, query caching is used to cache the result of a particular database query. If you have queried something and that query is cachable, the result is cached. The next time you hit the same query, you get the data from caching and you don't have to hit the database, thereby increasing the performance and lead time of the project of the database.

And what is the significance of `@Autowired` annotation in Spring Boot? The `@Autowired` annotation is used to automatically inject the dependency, and it is a key component of the Inversion of Control principle on which Spring is built upon. Suppose there is a `Car` class and there is an `Engine` class. Now, a car needs an engine, so you inject the dependency of engine into car. That's dependency injection. You don't create it with the `new` keyword; you rather inject its dependency. Both are loosely coupled; car can exist on its own, engine can exist on its own. Whenever a car needs an engine, you can inject its dependency, and car can use the engine to work appropriately. So, that's automatic resolution and injection of

engine dependency into the `Car` class. It simplifies the wiring of dependencies between car and engine, and instead of manually injecting or instantiating engine in car with the `new` keyword and looking up for dependencies, Spring Boot automatically takes care of resolving and injecting the engine dependency at runtime for the car class.

So, your code becomes much cleaner and maintainable. You achieve loose coupling; they both can exist on their own, and the dependencies for both car and engine are managed by the Spring container rather than hardcoding everything in the codebase. So, it makes your code more modular, easily testable, and promotes better separation of concerns, making them stand-alone. So, `@Autowired` is the one which supports the concept of dependency injection, that is, injecting the dependency of one component into another or injecting the dependency, in our case of engine, into car. It allows for easy swapping of dependencies and promotes code reusability. It enables you to easily switch implementations or mock the dependencies using unit testing. This is very important when you are unit testing. If you have created any class with the `new` keyword, mocking that `new` keyword instance becomes next to impossible with simple Mockito. But if you have autowired it, you can easily mock it using `@Mock` and Mockito for JUnit test cases. So, whenever you go for a Spring Boot application having multiple classes having dependencies on each other, go for `@Autowired` annotation always, rather than creating anything with the `new` keyword. Otherwise, your unit testing will make your life hell. If you want an easy unit test case, always go for `@Autowired`, making your code loosely coupled, so that their dependencies can be easily removed and mocked when unit testing has to be done for that particular class. So, whenever an interviewer asks you about the real advantage of `@Autowired`, always tell them that unit test cases are possible in Spring Boot only because we had `@Autowired` annotation. If we would not have that, mocking of this class without `@Autowired` and doing it with the `new` keyword would have been a difficult task. It would have been a nightmare for developers like us. So, `@Autowired` has actually helped us, the developers, in making the code unit-testable and maintainable for the whole life of the application. So, that's the significance of `@Autowired` in Spring Boot applications.

Now, again, they might ask you how did you handle exceptions in your Spring Boot application? So, there are multiple approaches to handling exceptions. Global exception handling can also be done for common exceptions. For example, if you are facing a null pointer exception, you can always use global exception handling. You can have a `ControllerAdvice` which is a class-based annotation and an `ExceptionHandler` which is a method-based annotation. You can create a global exception handler using `@ControllerAdvice` on a class. Inside the class, you can define methods which are annotated with `@ExceptionHandler` to provide specific exception handling. For example, null pointer exception can be handled using an exception handler. This method provides you the custom response for handling the exception. So, rather than giving a generic 500 Internal Server Error for a null pointer exception, which your client will not be able to understand, you could have said that the client forgot to pass some mandatory fields because of which the null pointer exception has occurred. Please try sending a proper request so that we don't face null pointer exceptions in the future. Send something meaningful that you can send from here to your client. So, rather than giving any 500 Internal Server Error and making the client think what went wrong, you can actually give some very specific reason why the null pointer has occurred and return a proper, specific HTTP response. Redirect them to a proper

error page and log the exception details so that it is better for you to understand what went wrong when a customer comes to you and says that they are getting a white label page.

The second way of handling exceptions is custom exception handling. You can have your own custom exceptions. For example, if a person has given an age greater than 100 or less than zero, then you can create an exception class called `AgeException`, where that class will extend the `RuntimeException` exception class. And then you can handle it using try and catch. So, whenever the person passes an age less than 0 or greater than 100, you can always throw the `AgeException` and catch it and give a proper, appropriate exception to the customer, saying that the age you have provided is either greater than 100 or less than zero. Also, you can handle these exceptions and handle them in appropriate exception handlers using try and catch blocks. So, these are the three ways of handling exceptions. Global exception handling is something which is global to the whole application. So, whenever in any class a null pointer exception comes, the global exception handler will catch it. Custom exception handling is somewhat more related to class-based exception handling, where you want to send some class-specific exceptions. For example, your service layer will always throw a service exception, your repository layer will always throw a repository or SQL exception. The class-based or scenario-based exception handling is custom exception handling, and the global exception handling is done by `ControllerAdvice` and `ExceptionHandler`. So, these are two ways in which you can manage your exceptions in a Spring Boot application. And always, try-catch will always be there to prevent you from any such 500 Internal Server Error types, white labels for your clients.

Now, if you are giving a Spring Boot interview, I'm 110% sure they'll ask you about profiling in Spring Boot. This is one of the most common questions we have always faced in each and every Spring Boot interview. So, explain the concept of profiles and how do you use them? Now, I'll give you an example first. Suppose I am using MySQL database, a relational database, for my current operations on my data. Now, since the production environment is used by my real clients, I am using AWS relational databases, which are very costly. Because for each and every request you give to AWS DB, it will cost you something like for RDS. Suppose I'm using relational DB RDS, each and every hit to that database is very costly to me. Now, suppose you have a team of 10 people who are developing this application. Now, for testing purposes, what they will do is they will keep on adding the data, keep on retrieving the data, we'll do all the CRUD operations, we'll put hundreds of queries to your database. Your billing amount for that RDS in AWS will reach a lot, in dollars, which is not required, right? In local, for testing, you could have used your in-memory DB or you could have used your local SQL. Why connect with AWS RDS and incur the data production cost? Also, suppose while working, they have connected to the production DB, added data, some deleted data, updated some wrong data. Your real customers are going to face a problem.

So, you understand that we need different properties for different environments. If you are testing and developing and you are in local, all three can have a local DB. But if you are in production or QA has to test it, they might have a different database. So, this clearly makes sense to have different configurations for different environments so that you can customize the behavior of your application based on the specific profile that's active at runtime. So, at runtime, if you are working locally or on development or testing, you can do the database operations in your local DB. It's not necessary to always go for AWS RDS production DB.

So, different sets of configurations and beans are required for different various environments like development environment, test environment, production environment. Profiles are very useful.

So, how does profiling work? I'll give you an example. Here, you can see I have `application.yml` file, I have `application-development.yml` file, I have `application-local.yml` file, and I also have `application-prod.yml` file. So, you have four environments: development, local, test, and prod. N number of profiles you can create here. And if you can see, in the development environment, I have given the Eureka service URL as `http://eureka-cluster.aws.com`. But for local, my Eureka URL is `http://localhost:8761`. So, this is how you can differentiate the URLs for the same Eureka in different environments. Because when I'm going to connect with AWS, AWS is going to have a cluster, an EKS cluster, where Eureka will be deployed. In that case, a different Eureka URL will be required to connect all your applications to Eureka successfully. So, this is how you give the local connection to local YAML and the cluster connection to Eureka, which is deployed for production or development. So, the first thing you need to do is you can define profile-specific configurations. Profile-specific configurations like, for example, `application-dev.yml`, `application-local.yml`, `application-prod.yml`. So, what Spring Boot does is it automatically loads the properties from the active profile. Now, how do you make sure which profile is active for you? So, here in the `application.yml`, this is how you tell which profile is active. So, currently, I say `dev` is active. That means `application-dev.yml` will automatically be picked up for properties to be configured by Spring Boot for your Spring Boot application.

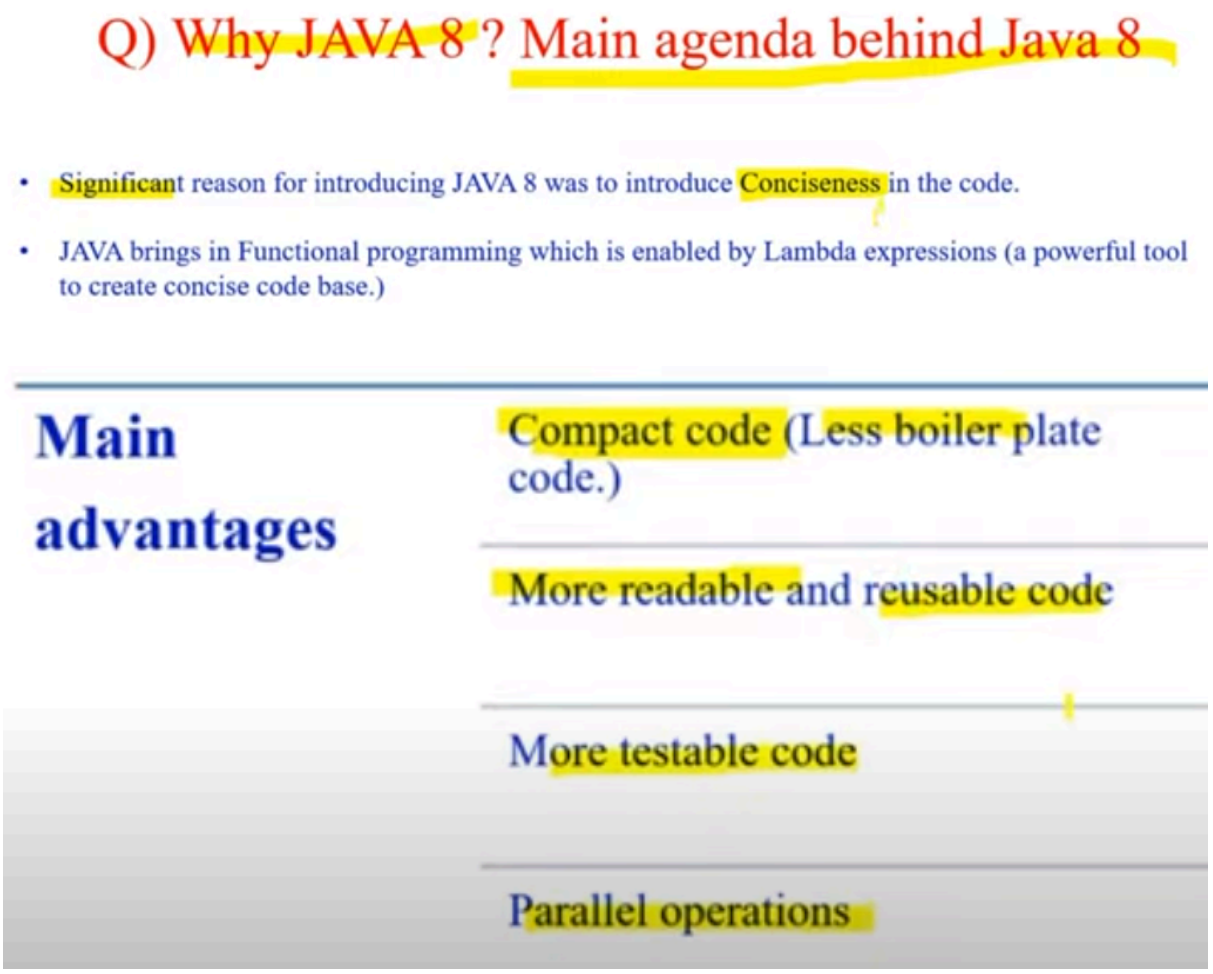
So, profiles can be defined in properties files or YAML files using `spring.profiles.active`. So, suppose `spring.profiles.active` is set to `dev`, that means you are going to work with the development environment. You can also have multiple profiles here, like `dev,local`. So, both can be picked up, and the one which is the last one will have precedence if both have common properties. So, here, the local property can be picked up next. How will you pick which profile has to be picked up? That is conditional-based bean registration. You can conditionally register beans based on the active profile. So, if you annotate a bean with `@Profile("test")`, that bean will only be available during unit testing or testing that you are doing. So, for example, if I have an entity `Restaurant`, but I'm just using it for testing purposes, so I could have used `@Profile("test")` like this and give it as a profile. So, until this profile of test is not active, the `Restaurant` entity will not be created and registered in the Spring container and will be used. For all the other profiles, you will get a bean creation exception. The bean for `Restaurant` is not found because you have given a property that it should be available only in the test environment. And in the application.properties or application.yml, you have given the current environment. So, remember, judiciously use this `@Profile` annotation on any bean. If you give the wrong profile on that bean, that bean will give a bean creation or bean instantiation exception when you are trying it in another profile.

Also, you can use `@ConditionalOnProperty` and other conditions when you want to configure a bean based on specific property values. This allows you to fine-tune the bean configuration based on different profiles. Also, our life becomes hell when unit testing becomes difficult. Now, suppose I want some properties to be picked up only during the testing and should be discarded after that. That can also be easily done using the `test`

profile. So, `@Profile("test")` and that bean will only be created in the test scenario. I've already told you how to do that.

So, profiles enable you to achieve clear separation of concerns, simple configuration management, one DB configuration for local and development, or a completely different one for production. It makes you more adaptable to different environments. Profiles enable you to easily switch between configurations without modifying the code. They promote rate of flexibility and maintainability of a Spring Boot application. Now, I have many more questions for five-plus and seven-plus years of experience. Questions like how to implement asynchronous processing, how to manage transactions in Spring Boot, how to handle cross-origin, the role of Actuator, how to schedule a cron job or a task in Spring Boot, and many more things. So, if you want me to cover the five-plus and seven-plus years of experience questions, you have to let me know in the comment section. Then only I'll create answers to these questions that are commonly asked. I'll be waiting for you in the comment section to help me decide whether I should go next to the five-plus or the seven-plus years of experience questions. Or even if you have a different topic in your mind, like Kafka or something, just let me know in the comment section. The next video I'll create based on your comments and requests. Thank you.

Java 8



Q) What is lambda expression?

- Lambda expression is an anonymous function (without name, return type and access modifier and having one lambda (→) symbol)
- Eg :

Normal programming technique

```
public void add(int a, int b) {
        System.out.println(a+b);
}
```

Equivalent Lambda Expression

```
(a, b) -> System.out.println(a + b);
```

Q) What are functional interfaces?

- Functional interfaces are those interfaces which can have only one abstract method
- It can have any number of static method, default methods. No restriction on that
- There are many functional interfaces already present in java such as eg : Comparable, Runnable

LambdaDemo.java | Runnable.class

```java
 * be used if you are only planning to override the <code>run()</code>
 * method and no other <code>Thread</code> methods.
 * This is important because classes should not be subclassed
 * unless the programmer intends on modifying or enhancing the fundamental
 * behavior of the class.
 *
 * @author  Arthur van Hoff
 * @see     java.lang.Thread
 * @see     java.util.concurrent.Callable
 * @since   JDK1.0
 */
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

Read-Only | Smart Insert | 68 : 32

SUBSCRIBE

Q) How lambda expression and functional interfaces are related?

- **Functional Interface is used to provide reference to lambda expressions.** --→ This is the relation.
- Comparator<String> c = (s1,s2) → s1.compareTo(s2);
- (s1,s2) → s1.compareTo(s2) : This is lambda Expression
- Comparator<String> c : This is Functional Interface
- Thus you can see , To call lambda expressions we need Functional Interfaces.
-

Q) Can you create your own functional interface?

- As we know Functional interface is an interface with Exactly One Single Abstract method and can have multiple Static or default methods.
- To create our own Functional interface, You can do following steps:
  - Create An interface
  - Annotate that with @FunctionalInterface.
  - Define exactly one Abstract method.
  - There is no restriction on number of static and default methods defined in such and interface.
- Java can implicitly identify functional interface but still you can also annotate it with @FunctionalInterface . It just give you the security that in case if u by mistake add 2 abstract methods then Compiler will throw **compile time error.**

LambdaDemo.java | Runnable.class | BiConsumer.class | FunctionalInterfaceDemo.java

```java

@FunctionalInterface
public interface FunctionalInterfaceDemo {

    void singleAbstMethod();

//  void singleAbstMethod2();

    default void printName()
    {
        System.out.println("Welcome to Code Decode");
    }


}
```

Q) What is method reference in java 8?

- EG : MethodReferenceDemo and FunctionalInterfaceDemo
- Method reference is replacement of lambda expressions. It is used to refer method of Functional interface to an existing method. Mainly it is used for code reusability.
- Functional Interface's Abstract method can be mapped to specific existing method using double colon operator (::) . This is Method reference.
- Hence Method reference is an alternative to Lambda expressions.
- Whenever we have existing Implementation of Abstract method of our Functional interface then we can go for method reference. If no such method like testImplementation() is available then go for lambda expressions.

```java
public class MethodReferenceDemo {
    public static void main(String[] args) {
        FunctionalInterfaceDemo functionalInterfaceDemo = Test :: testImplementation;
        functionalInterfaceDemo.singleAbstMethod();

        //If no testImplementation is available in existing code base then use following
//      code for lambda expressions:
//      FunctionalInterfaceDemo f = () -> System.out.println("IMPLEMENTATION of SAM");
//      f.singleAbstMethod();
    }


}

class Test {
    public static void  testImplementation() {
        System.out.println("This is test implementation of your abstract method");
    }
}
```

Here , for the implementation of single abstract method of function interface, Test:: testImplementaion , method can be refereed.
If this ref solution was not there , you would have done this using normal lambda expression

```java
public class MethodReferenceDemo {
    public static void main(String[] args) {
//      FunctionalInterfaceDemo functionalInterfaceDemo = Test :: testImplementation;
//      functionalInterfaceDemo.singleAbstMethod();

        //If no testImplementation is available in existing code base then use following
//      code for lambda expressions:
        FunctionalInterfaceDemo f = () -> System.out.println("IMPLEMENTATION of SAM");
        f.singleAbstMethod();
    }

}

//class Test {
//  public static void  testImplementation() {
//      System.out.println("This is test implementation of your abstract method");
//  }
//}
```

Q) What are Predicates

- Predicate is a predefined Functional Interface (Having only 1 abstract method).
- The only abstract method of predicate is test(T t):
  - public boolean test(T t);
- Whenever we want to check some boolean condition then you can go for Predicates.

Q) What are streams?

If we want to process bulk objects of collection then go for streams concept.
Way to operate on collection in java 8 is Stream.
Its a special iterator class that allows processing collections of objects in a functional manner.

Eg : fetch all objects from collection of list whose value is greater than 15

```
treamDemo.java  CreateStream.java  MapDemo.java  FilterDemo.java  FilterVsMap.java

public class StreamDemo {
    public static void main(String[] args) {

        List<Integer> arList =new ArrayList<Integer>();
        arList.add(15);
        arList.add(25);
        arList.add(5);

        List<Integer> newAl =new ArrayList<Integer>();

        newAl = arList.stream().filter(x -> x>=15).collect(Collectors.toList());

        newAl.stream().forEach(x -> System.out.println(x));

        //without streams:
        /*List<Integer> arListFromMEthod = findElements(arList);
        for (Integer i : arListFromMEthod) {
            System.out.println(i);
        }
    }
```

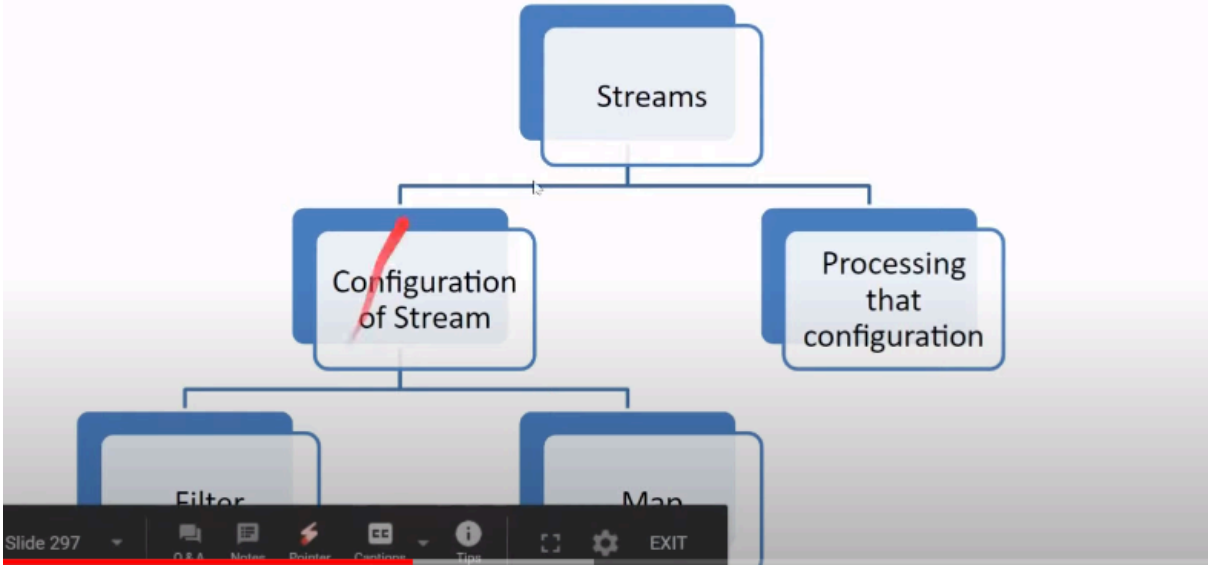
```
InterfaceDemo.java  InterfaceWithStaticInterfaceImpl.java  PredicateFunctionDemo.java

1  package predicate_demo;
2  import java.util.function.Predicate;
3
4  public class PredicateFunctionDemo {
5
6      public static void main(String[] args) {
7              PredicateFunctionDemo predicateFunctionDemo = new  PredicateFunctionDemo();
8              System.out.println(predicateFunctionDemo.testStringLength("code decode"));
9
10         Predicate<String> checkLength = s -> s.length() >= 5;
11         System.out.println("The length of string is greater than 5 : " + checkLength.test("code decod
12
13         //      Function<Integer , Integer> squareMe = i -> i*i;
14         //      System.out.println("Square of 5 is " + squareMe.apply(5));
15     }
16
17         public boolean testStringLength(String s) {
18             if (s.length() >= 5) {
19                 return true;
20             } else {
21                 return false;
22             }
23         }
24
25
26 }
```

Q) Difference between streams (java 1.8) and java.io.Stream?

- Why streams were introduced in java 8 if we already had java.io.stream?

- Java io streams is a sequence of characters or binary data which is used to be written to a file or to read data from a file.
- While streams java 1.8 is no where related to files, its related to collection object.
- Java io streams related to file whereas java 8 streams are related to collection object.
- Hence if we need to perform some operations on collection there we should go for streams.

Q) Steps to create and process stream



Q) How to Filter the stream objects

- Stream s = collectionObject.stream().filter(i → i % 2 ==0);
- See Demo

StreamDemo.java | CreateStream.java | MapDemo.java | FilterDemo.java | FilterVsMap.java

```java
package streams_demo;

import java.util.ArrayList;

public class FilterDemo {

    public static void main(String[] args) {

        List<Integer> arList =new ArrayList<Integer>();
        arList.add(15);
        arList.add(25);
        arList.add(52);

//      Stream s = arList.stream().filter(i -> i%2 == 0);
//      s.forEach(x -> System.out.println(x));

        arList.stream().filter(i -> i%2 == 0).forEach(x -> System.out.println(x));

    }


}
```

Q) How to Map the stream objects

- What if we don't want to filter out.
- We rather want to create new object against each existing stream object based on some function.

- EG in given stream create new object by squaring its value
- Demo

Search Project Run Window Help

StreamDemo.java | CreateStream.java | MapDemo.java | FilterDemo.java | FilterVsMap.java

```java
package streams_demo;

import java.util.ArrayList;

public class MapDemo {

    public static void main(String[] args) {

        List<Integer> arList =new ArrayList<Integer>();
        arList.add(15);
        arList.add(25);
        arList.add(52);

        Stream s = arList.stream().map(i -> i*i);
        s.forEach(x -> System.out.println(x));
    }


}
```

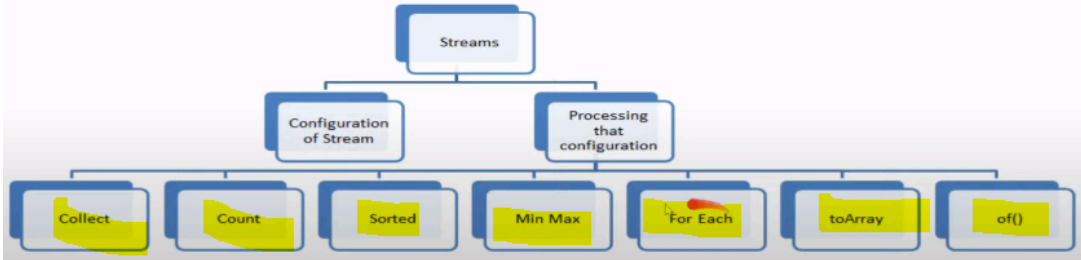
Q) Difference between Filter and Map

- If we want to fetch / filter objects from collection like eg : filter only even numbers from array list collection the use Filter for configuration of stream.

- If we want to perform some operation on each objects of the collection then create another mapped object with different value(after operation is performed) for each object of that collection, then use map.

- In filter, because of filtering, number of objects in filtered list is less than original list While in Map same number of objects are there in both new and original list created.

Q) Steps to create and process stream



Why Optional Objects

- Java Optional class provides a way to deal with null values. It is used to represent a value is present or not. Java 8 added a new class Optional available in java.util package.
- A NullpointerException is a common issue in java applications. To prevent this, we normally add frequent NULL checks in our code to check if a variable is not empty before we use it in our program. Optional provides a better approach to handle such situations.
- You can view Optional as a single-value container that either contains a value or doesn't (it is then said to be "empty"),

```
@GetMapping(path ="/id/{id}")
public ResponseEntity<?> getById(@PathVariable Integer id) {
    Optional<Employee> e = repo.findById(id);
    if(e.isPresent()){
        return new ResponseEntity<>(e.get(), HttpStatus.OK);
    }else
        return new ResponseEntity<>(" Sorry Employee with given ID is not present", HttpStatus.NOT_FOUND);

}

@GetMapping(path ="/findAll")
```

Creating Optional Objects

- We can create optional object with Optional's static method - Of()
- the argument passed to the of() method can't be null. Otherwise, we'll get a NullPointerException.

- But in case we expect some null values, we can use the ofNullable() method:
- By doing this, if we pass in a null reference, it doesn't throw an exception but rather returns an empty Optional object:

Optional - Of/ isEmpty/isPresent

```
@GetMapping(path ="/id/{id}")
public ResponseEntity<?> getById(@PathVariable Integer id) {
    Optional<Employee> e = repo.findById(id);
    if (!e.isEmpty()) {
        Optional<String> name = Optional.of(e.get().getName());
        if (name.isPresent())
            return new ResponseEntity<>(name.get().toUpperCase(), HttpStatus.OK);
        else
            return new ResponseEntity<>(" Name is null ", HttpStatus.NOT_FOUND);
    }else
        return new ResponseEntity<>(" Sorry Employee with given ID is not present", HttpStatus.NOT_FOUND);

}

@GetMapping(path ="/findAll")
public ResponseEntity<?> getAll() {
```

Or if not sure about "e.get().getName" might throw null

```
@GetMapping(path ="/id/{id}")
public ResponseEntity<?> getById(@PathVariable Integer id) {
    Optional<Employee> e = repo.findById(id);
    if (!e.isEmpty()) {
        Optional<String> name = Optional.ofNullable(e.get().getName());
        if (name.isPresent())
            return new ResponseEntity<>(name.get().toUpperCase(), HttpStatus.OK);
        else
            return new ResponseEntity<>(" Name is null ", HttpStatus.NOT_FOUND);
    }else
        return new ResponseEntity<>(" Sorry Employee with given ID is not present", HttpStatus.NOT_FOUND);

}
```

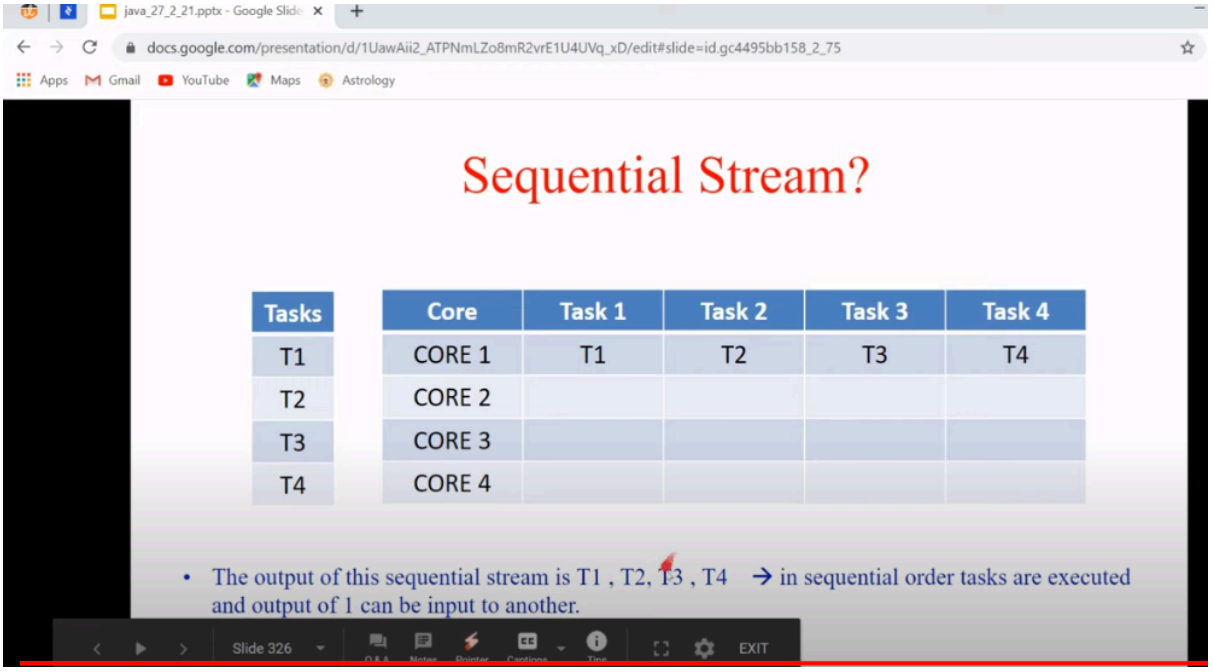
IfPrsent

```
@GetMapping(path ="/id/{id}")
public ResponseEntity<?> getById(@PathVariable Integer id) {
    Optional<Employee> e = repo.findById(id);
    if (!e.isEmpty()) {
        Optional<String> name = Optional.ofNullable(e.get().getName());
        /*
         * if (name.isPresent()) return new ResponseEntity<>(name.get().toUpperCase(),
         * HttpStatus.OK); else return new ResponseEntity<>(" Name is null ",
         * HttpStatus.NOT_FOUND);
         */
        name.ifPresent(n -> System.out.println("name is present " + n));
    }else
        return new ResponseEntity<>(" Sorry Employee with given ID is not present", HttpStatus.NOT_FOUND);
    return null;
}
```

IfPresentOrElse

```
@GetMapping(path ="/id/{id}")
public ResponseEntity<?> getById(@PathVariable Integer id) {
    Optional<Employee> e = repo.findById(id);
    if (!e.isEmpty()) {
        Optional<String> name = Optional.ofNullable(e.get().getName());
        /*
         * if (name.isPresent()) return new ResponseEntity<>(name.get().toUpperCase(),
         * HttpStatus.OK); else return new ResponseEntity<>(" Name is null ",
         * HttpStatus.NOT_FOUND);
         */
        //name.ifPresent(n -> System.out.println("name is present " + n));
        name.ifPresentOrElse(n -> System.out.println("name is present " + n), () -> System.out.println("Sorry name
    }else
        return new ResponseEntity<>(" Sorry Employee with given ID is not present", HttpStatus.NOT_FOUND);
    return null;
}

@GetMapping(path ="/findAll")
```

Sequential Stream & core



To introduce multitasking

Gmail YouTube Maps Astrology

Parallel Stream?

Tasks
T1
T2
T3
T4

Core	Task 1	Task 2	Task 3	Task 4
CORE 1		T2		
CORE 2				T4
CORE 3	T1			
CORE 4			T3	

When order does not matter, things can be executed indepedently

StreamVsParallelStream.java

```java
package com.bse.parallelstream;

import java.util.Arrays;
import java.util.List;

public class StreamVsParallelStream {

    public static void main(String[] args) {

    List<String> listStr =  Arrays.asList("a","b","c","d","e","f");

    listStr.stream().forEach(System.out::println);

    System.out.println("----Parallel Stream----");

    listStr.parallelStream().forEach(System.out::println);
```

Problems Javadoc Declaration Console Progress Git Staging Debug

No consoles to display at this time.

```
<terminated> StreamVsParallelStream [Java Application] C:\P
a
b
c
d
e
f
----Parallel Stream----
d
f
e
b
c
a
```

Runnable vx Callable

- f you simply need to perform a unit of work without returning a result, use `Runnable`. For example, updating a shared data structure, writing to a log, or performing some asynchronous action.
- If you need to perform a computation and return a result or handle exceptions, use `Callable`. This is suitable for scenarios like computing a value, making a network request, or reading from a database.

```
import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;


public class ConcurrencyExample {

    public static void main(String[] args) throws Exception {

        ExecutorService executorService = Executors.newFixedThreadPool(2);


        Runnable runnable = new MyRunnable();

        Callable<Integer> callable = new MyCallable();


        executorService.submit(runnable);


        Future<Integer> future = executorService.submit(callable);

        Integer result = future.get();

        System.out.println("Callable result: " + result);


        executorService.shutdown();

    }

}


class MyRunnable implements Runnable {
```

```
    @Override
    public void run() {
        System.out.println("Runnable task executed");
    }
}
import java.util.concurrent.Callable;


class MyCallable implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        return 42;
    }
}
```

`Future` is particularly useful in scenarios where you need to perform tasks concurrently and then collect the results or perform further actions once the tasks are complete. It's commonly used with the `ExecutorService (isDOne,cancel.get())`

Caching in spring boot

Spring Boot provides several caching annotations that you can use:

`@EnableCaching`: This annotation is typically added to your main application class to enable caching at the application level.
`@Cacheable`: This annotation is used to indicate that the result of a method should be cached. When a method annotated with `@Cacheable` is called, Spring will first check the cache for the result before actually executing the method. If the result is found in the cache, it is returned directly; otherwise, the method is executed, and its result is stored in the cache.
`@CachePut`: This annotation is used to update the cache with the result of a method, regardless of whether the result was found in the cache. It is typically used when you want to update the cached value after a method call.
`@CacheEvict`: This annotation is used to remove an entry from the cache. It is commonly used when you want to explicitly invalidate cached data, such as when updating or deleting data.
`@Caching`: This annotation allows you to apply multiple caching annotations to a single method. It can be useful when you want to configure different caching behaviors for different scenarios.

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter'
    implementation 'org.springframework.boot:spring-boot-starter-cache'
}
Application.yaml ==
# Cache configuration
spring.cache.type=caffeine


================
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;

@SpringBootApplication
@EnableCaching
public class CachingExampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(CachingExampleApplication.class, args);
    }
}


import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;
```

```
@Service
public class BookService {

    @Cacheable(value = "books", key = "#isbn")
    public Book getBookByIsbn(String isbn) {
        // Simulate fetching book details from a data source
        return new Book(isbn, "Sample Book Title");
    }

    @CacheEvict(value = "books", key = "#isbn")
    public void updateBookTitle(String isbn, String newTitle) {
        // Simulate updating book title in a data source
    }
}
=====

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class BookController {

    private final BookService bookService;

    public BookController(BookService bookService) {
        this.bookService = bookService;
    }

    @GetMapping("/books/{isbn}")
    public Book getBookByIsbn(@PathVariable String isbn) {
        return bookService.getBookByIsbn(isbn);
    }

    @PutMapping("/books/{isbn}")
    public void updateBookTitle(@PathVariable String isbn, @RequestBody String newTitle) {
        bookService.updateBookTitle(isbn, newTitle);
    }
}
===
public class Book {
    private String isbn;
    private String title;

    // Constructors, getters, and setters
```

}

Serilization./Deserilization

In this example, attempting to serialize the `Person` object containing a non-serializable `Address` object will result in a `java.io.NotSerializableException`.

To resolve this issue, you have a few options:

- Make `Address` Serializable: Implement the `Serializable` interface in the `Address` class.
- Mark `Address` as Transient: If the `Address` field is not crucial to the serialization/deserialization process, you can mark it as `transient` to exclude it from serialization.
- Custom Serialization Logic: Implement custom serialization and deserialization methods in the `Person` class to handle the `Address` object's serialization and deserialization explicitly, as shown in the previous example.

Remember that when working with serialization, all non-transient fields and their references must be serializable, otherwise, you'll encounter serialization issues.

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.io.IOException;

public class SerializationExample {
    public static void main(String[] args) {
        Person person = new Person("John", 30);

        try (FileOutputStream fileOutputStream = new FileOutputStream("person.ser");
             ObjectOutputStream objectOutputStream = new
ObjectOutputStream(fileOutputStream)) {

            objectOutputStream.writeObject(person);
            System.out.println("Object serialized successfully.");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class Person implements Serializable {
```

```
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getters, setters, other methods
}
```

====

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.Serializable;
import java.io.IOException;

public class DeserializationExample {
    public static void main(String[] args) {
        try (FileInputStream fileInputStream = new FileInputStream("person.ser");
             ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream)) {

            Person deserializedPerson = (Person) objectInputStream.readObject();
            System.out.println("Object deserialized successfully.");
            System.out.println("Name: " + deserializedPerson.getName());
            System.out.println("Age: " + deserializedPerson.getAge());

        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Java garbage collection is the process by which Java programs perform automatic memory management.
You see Java programs compile into bytecode that can run on a Java Virtual Machine, when Java programs run on the JVM, objects are created in the Heap Space which is a portion of memory dedicated to the program. Inside this space and over the lifetime of a Java
application new objects are created and released and at any point in time the Heap memory consists
of two types of objects: Alive or Dead. Alive objects are being used and

referenced from somewhere else. Dead objects are no longer used or
referenced from anywhere in the application and it's these objects that are detected
by the GC and deleted to free up memory. Now we know that the garbage collector detects
and deletes unused objects from the Heap space but how exactly does it work in Java?
Well there are are various ways in which the reference to an object can be released
Generational Garbage Collection: How does it Work?
to make it a candidate for garbage collection some of them are by making a reference null,
or by assigning a reference to another, or by using an anonymous object.
Keep in mind that everything we mentioned is automated, the programmer does not
need to explicitly mark objects to be deleted. Okay you should know that the garbage collection
implementation lives in the JVM and each JVM can Implement its own version of garbage collection.

Now, to perform garbage collection or write your garbage collector you should include
in your implementation three main phases:
First, marking objects as alive. In this
step the GC identifies all the alive objects in memory by traversing the object graph.
When GC visits an object it marks it as accessible and thus alive,
every object the garbage collector visits is marked as alive
all the objects which are not reachable are considered candidates for garbage collection.

The second phase is sweeping dead objects. After marking each object as alive or dead,
the sweep phase releases the memory fragments which contain these dead objects.

Finally in the third and last phase the remaining objects in memory are compacted,
the dead objects that were removed during the sweep phase may not
necessarily be next to each other thus we can end up with fragmented
memory space this phase ensures that the memory is compact after the garbage
collector deletes the dead objects so that the remaining objects are in
a contiguous block at the start of the heap.
But wait! Having to mark and compact all the
objects in a JVM every single time is inefficient, it must be right?
Well yes it is, and to solve this Java garbage collectors implement a

generational garbage collection strategy that categorizes objects by age

called: Generational Garbage Collection. You see, as more and more objects are allocated,
the list of objects grows leading to longer garbage collection times
and in fact several analysis of applications have shown that most objects have a very short lifespan
to benefit from these findings the heap memory area and the JVM was divided into three sections:
the Young Generation which consists of newly created objects and is further subdivided
into the Eden space and the Survivor spaces, all the new objects that start in the Eden and
initial memory is allocated to them if an object survives one GC cycle
then they are moved to Survivor spaces when objects are garbage collected from the Young

Generation it is a minor garbage collection event a minor GC is performed when the Eden space
is filled with objects either dead or alive although that objects are deleted and all the live objects are moved to one of the Survivor spaces. Take this sequence as an example, suppose Eden is filled with objects alive and dead so a minor GC should occur, all dead objects are
removed from Eden and all alive objects are moved to the From Space, Eden and To Spaces are now empty
new objects are created and added to Eden some objects in the From Space become dead as well
Minor GC occurs all dead objects are removed from Eden and From Space,
all the alive objects are moved to: To Space, Eden and From Space are now empty.
So at any time one of the Survivor spaces is always empty
when the surviving objects reach a certain threshold of moving around the Survivor spaces they are moved to the old generation. The Old Generation is the second
out of the three sections we mentioned objects that are alive long enough are
eventually moved from the Young Generation to the Old generation.
It contains objects that have remained in the Survivor spaces for a long time.
When objects are garbage collected from the older generation it is a major garbage collection event.
So the cycle we previously saw will be repeated for a specific number of times,
if the object is still used after this point, the next garbage collection cycle will move
it to the old generation space. The third and final generation
we have is the permanent generation. Metadata such as classes and methods
are stored in the permanent generation it is populated by the JVM at runtime
based on classes in use by the application. Starting with Java 8 the MetaSpace memory space replaced the PermGen space, the implementation differs from
the PermGen as the space of the heap is now automatically resized,
this avoids the problem of applications running out of memory due to the limited
size of the PermGen Space of the heap. The MetaSpace memory can be garbage
collected and the classes that are no longer used can be automatically cleaned
when the metaspace reaches its maximum size. Okay now that we know how the GC operates,
Types of Garbage Collectors
let's take a look at different types of garbage collectors available in the Java virtual machine.
First, the Serial GC. This is the simplest implementation of
GC and is designed for small applications running on single threaded environments
all garbage collection events are conducted serially in one thread
when it runs it leads to a stop-the-world event where the entire application is paused.
Now the pair Parallel GC. The parallel collector is intended for applications
with medium-sized to large-sized data sets that run on multi-threaded hardware,
multiple threads are used for minor garbage collection in the Young Generation
and the single thread is used for major garbage collection in the old generation.
Running the parallel GC also causes a 'stop the world' event and freezes the application,
so since it is more suitable in a multi-threaded environment it can be used when a lot of work
needs to be done and long pauses are acceptable. Next is the Concurrent Mark Sweep or CMS GC.

Multiple threads are used for minor and major garbage collection
CMS runs concurrently alongside application processes to minimize stop-the-world events
because of this the CMS collector uses more CPU than other GCs
so if you can allocate more CPU for better performance then the CMS garbage connector is
a better choice than the parallel collector. Now the Garbage First or G1 GC also
chosen as the default collector by Java. G1 GC was intended as a replacement for CMS and
was designed for multi-threaded applications that have a large Heap size available.
It is parallel and concurrent like CMS but it works quite differently under
the hood compared to the old garbage collectors although G1 is also generational it does not have
separate regions for young and old generations, instead each generation is a set of regions which
allows resizing of the Young Generation in a flexible way
it partitions the Heap into a set of equal size regions and uses multiple threads to scan them.
A region might be either an old region or a young region at any time during the program run
after the marked phase is completed, G1 knows which regions contains the most garbage objects.
If the user is interested in minimal pause times G1 can choose to evacuate only a few regions
if not G1 might choose to include more regions since G1 GC identifies
the regions with the most garbage and performs garbage collection on
that region first, it is called garbage first. Next is the Epsilon garbage collector. Epsilon
is a do-nothing garbage collector that was released as part of JDK 11
it handles memory allocations but does not Implement any actual memory reclamation mechanism
once the available Java Heap is exhausted the JVM shuts down.
It can be used for ultra latency sensitive applications where developers
know the application memory footprint exactly or even have completely garbage-free applications.
The use of the Epsilon GC and any other scenario is otherwise discouraged.
Now the Shenandoah GC. Shenandoah is a new GC that was released as part of JDK 12.
Shenandoah's key advantage over G1 is that it does more of its garbage collection cycle work
concurrently with the application threads. G1 can evacuate its Heap regions only
when the application is paused while Shenandoah can relocate
objects concurrently with the application. Shenandoah can compact live objects,
clean garbage, and release RAM back to the OS almost immediately after detaching free memory
since all of this happens concurrently while the application is running
Shenandoah is more CPU intensive. Finally the ZGC. ZGC is another
GC that was released as part of JDK 11 and has been improved in JDK 12.
It is intended for applications which require low latency and or use a very large heap.
The primary goals of ZGC are low latency, scalability, and ease of use.
To achieve this, ZGC allows a Java application to continue running while it performs all
garbage collection operations thus ZGC brings a significant
improvement over other traditional GCs by providing extremely low pause times.
Okay in the last section of this video I want to discuss a couple advantages

Advantages of Garbage Collection
of garbage collection in Java. First it makes your code simple,
you don't have to worry about proper memory assignment then release cycles,
you just stop using an object in your code and the memory it is using will
be automatically reclaimed at some point. Additionally it makes Java memory efficient
because the garbage collector removes the unreferenced objects from heap memory
this frees the heap memory to accommodate new objects.
While some programmers argue in favor of manual memory management over a garbage collection,
garbage collection is now a standard component of many popular programming languages.
So that's it for this video I hope it was helpful thank you

SQL queries:

Q18. Write a query to calculate the even and odd records from a table.

To retrieve the even records from a table, you have to use the MOD() function as follows:

```
SELECT EmpID FROM (SELECT rowno, EmpID from EmployeeInfo)
WHERE MOD(rowno,2)=0;
```

Similarly, to retrieve the odd records from a table, you can write a query as follows:

```
SELECT EmpID FROM (SELECT rowno, EmpID from EmployeeInfo)
WHERE MOD(rowno,2)=1;
```

Q19. Write a SQL query to retrieve employee details from EmployeeInfo table who have a date of joining in the EmployeePosition table.

```
SELECT * FROM EmployeeInfo E
WHERE EXISTS
(SELECT * FROM EmployeePosition P WHERE E.EmpId = P.EmpId);
```

Q24. Write a query to retrieve the last 3 records from the EmployeeInfo table.

```
SELECT * FROM EmployeeInfo WHERE
EmpID <=3 UNION SELECT * FROM
(SELECT * FROM EmployeeInfo E ORDER BY E.EmpID DESC)
```

```
AS E1 WHERE E1.EmpID <=3;
```

Q25. Write a query to find the third-highest salary from the EmpPosition table.

```
SELECT TOP 1 salary
FROM(
SELECT TOP 3 salary
FROM employee_table
ORDER BY salary DESC) AS emp
ORDER BY salary ASC;
```

Q16. Write a query to fetch all employees who also hold the managerial position.

```
SELECT E.EmpFname, E.EmpLname, P.EmpPosition
FROM EmployeeInfo E INNER JOIN EmployeePosition P ON
E.EmpID = P.EmpID AND P.EmpPosition IN ('Manager');
```

Q9. Write a query to fetch top N records.

By using the TOP command in SQL Server:

```
SELECT TOP N * FROM EmployeePosition ORDER BY Salary DESC;
```

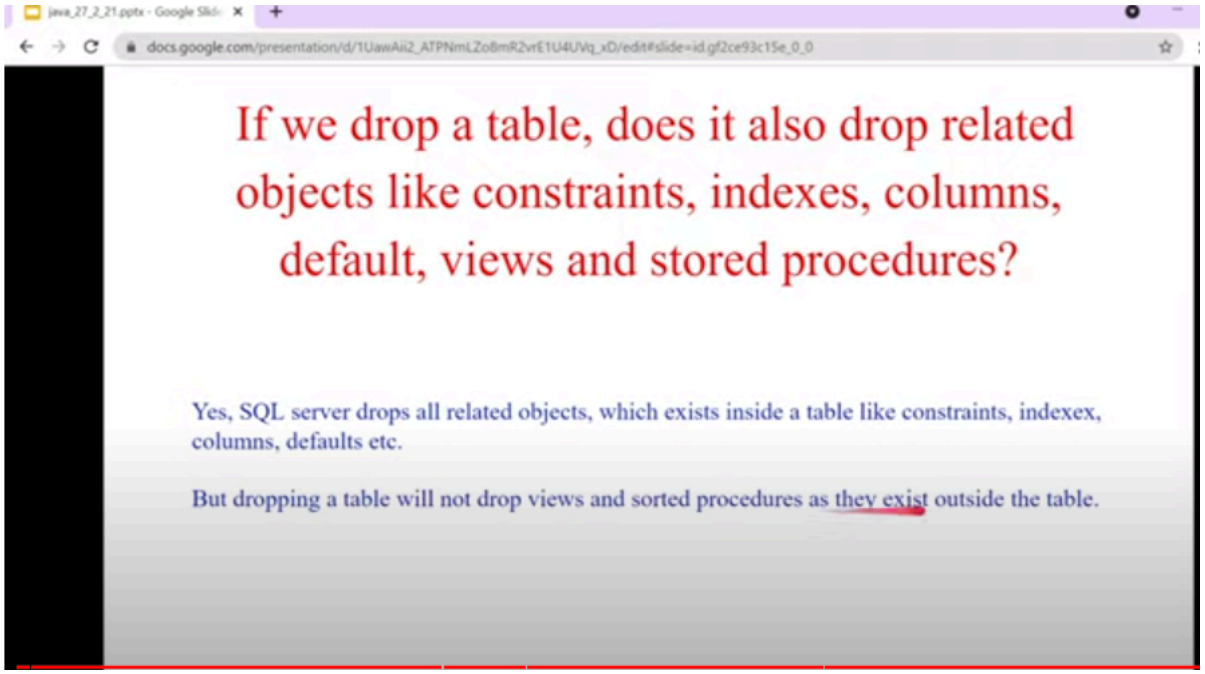
By using the LIMIT command in MySQL:

```
SELECT * FROM EmpPosition ORDER BY Salary DESC LIMIT N;
SELECT salary
FROM Table_Name
ORDER BY salary DESC
LIMIT 2, 1 ( 2-> start index, 1- how manyrows)
```

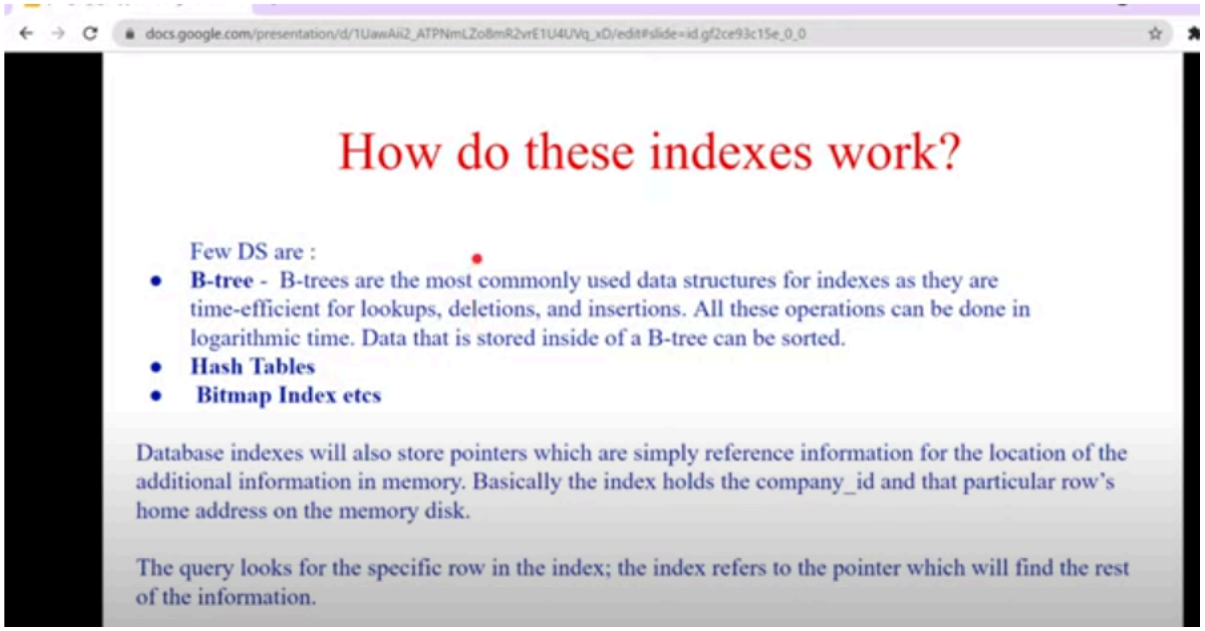
Write a query to find the Nth highest salary from the table without using TOP/limit keyword.

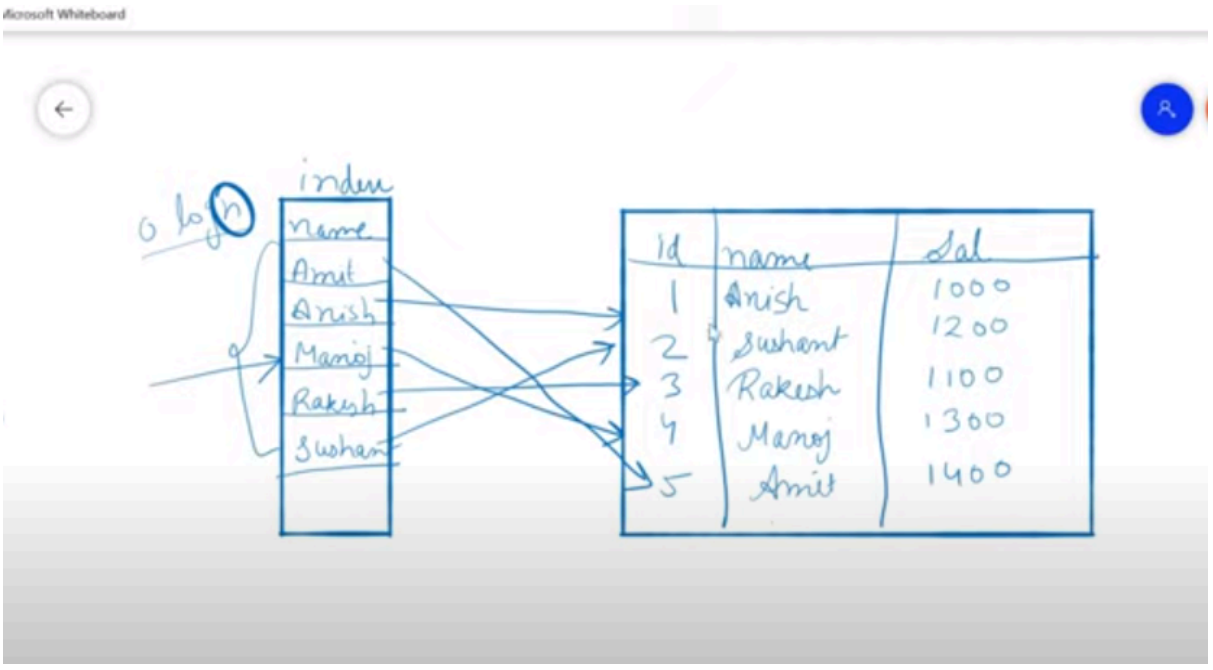
```
SELECT Salary
FROM EmployeePosition E1
WHERE N-1 = (
      SELECT COUNT( DISTINCT ( E2.Salary ) )
      FROM EmployeePosition E2
      WHERE E2.Salary >  E1.Salary );
```

we drop a table
does it also drop the objects like
constraint indexes columns
views stored procedures everything
answer is yes as you drop the sql table
all the related objects which exists
inside the table like constraint indexes
columns defaults are all dropped but no
what doesn't get dropped with the table
deletion or table dropping is the views
and the stored procedures why they'll
ask you for sure why does the view and
stored procedure doesn't get deleted or
dropped with the table dropping that is
because they exist outside the table



How index works





What are Indexes and How to create an index in SQL?

Indexes are database objects which help in retrieving records quickly and more efficiently. Column indexes can be created on both Tables and Views. By declaring a Column as an index within a table/ view, the user can access those records quickly by executing the index. Indexes with more than one column are called Clustered indexes.

Syntax:

CREATE INDEX INDEX_NAME

ON TABLE_NAME(COL1, COL2);

The syntax to drop an Index is DROP INDEX INDEX_NAME on TABLE_NAME;

Indexes are known to improve the efficiency of SQL Select queries

How do these indexes work?

Suppose we need to search by employee name = Manoj

What goes on behind the scenes is Every single row is checked to see if the employee_name matches with Manoj. This effectively means that the entire table will have to be scanned (know as the full table scan).

An index is a data structure that stores the values for a certain specific column of a table and hel us avoid a full table scan.

Database Indexing in reality, allows us to cut down the number of rows/records that need to be examined when a select query with a where clause is executed.

Disadvantages of indexes?

- Index takes up additional space, so the larger the table, the bigger the index.
- Every time you perform an add, delete, or update operation, the same operation will need to be performed on the index as well.

java_27_2_21.pptx - Google Slid
docs.google.com/presentation/d/1UawAii2_ATPNmLZo8mR2vrE1U4UVq_xD/edit#slide=id.g10490bc21fd_0_49

How to Tune your SQL Query

- ✓ SELECT fields instead of using SELECT *
- ✓ Avoid SELECT DISTINCT
- ✓ Create joins with INNER JOIN (not WHERE)
- ✓ Use WHERE instead of HAVING to define filters
- ✓ Avoid too much JOINs on your query: use only what is necessary!
- ✓ Avoid cursors at all costs!

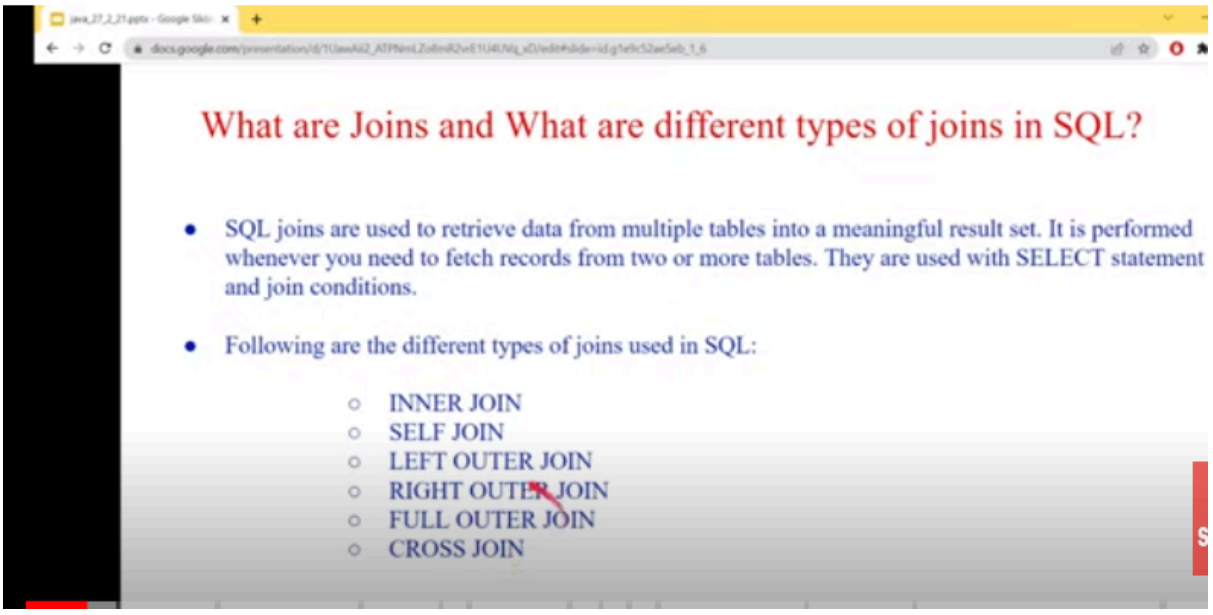


Table 1- 3 rows
Table2 - 4 rows

Say 2 rows are common

Cross join- > 12 rows
Full join -> 7 rows (union)
Left outer join- > all the rows from left table which could be common for right + null for right) -> 3 rows

right outer join- > all the rows from right table which could be common for left+ null for left) -> 4 rows

Rest Interview Questions:

In this video, we're gonna dive deep into REST APIs and talk about eight advanced interview questions. My name is Kevin Way. I'm a product manager, and I'm here to help you land your dream job in tech.
[Music]

Understanding REST APIs is crucial to getting your dream software engineering job. In our last video, we talked about nine common REST API interview questions and answers. Today, we're going to dive deep and cover eight more advanced questions and answers. So let's get started.

Question 1: What is the difference between PUT, POST, and PATCH?
Put and post are very common HTTP methods. The key difference is that PUT is used to update a specified existing resource, while POST is used to create a new resource. There's also the method PATCH, which is another way of updating a particular resource. The difference between PUT and PATCH is that PATCH alters only the data that is specified, whereas PUT replaces the entire resource with a new resource comprised of the given data.

Question 2: What is a payload in the context of a REST API?
Payload refers to any data that has been transferred via request or response through the REST API. For instance, if the client makes a GET request, the server will provide a response with a payload. Or the client may provide the payload, like in any POST request where the client is giving data to the server.

Question 3: What is a REST message?
Messaging refers to the back and forth communication via request and response by the client and server. There's typically a message within every response body associated with the provided status code. This message is one of the most important parts of debugging and testing a REST API. Examples of a message include confirmation of resource creation or exceptions being raised.

Question 4: What are the core components of an HTTP request?
There are five components in an HTTP request:
- Method
- URI
- HTTP version
- Request header
- Request body

The method refers to operations like GET, POST, PUT, DELETE, etc. URI is used to identify the resource typically using a URL. HTTP version indicates the version being used. Request header contains metadata, and the request body contains the content or data being sent.

Question 5: What are the core components of an HTTP response?
There are four components in an HTTP response:
- Status code
- HTTP version
- Response header
- Response body

The status code provides information about the success or failure of a request. 200 codes indicate success, 400 codes indicate a client-side error, and 500 indicates a server-side error. The response header contains response metadata, and the response body contains the return data.

Question 6: What is an idempotent method and why are they important?
An idempotent method is one that yields the same response regardless of how many times a request is sent. Rest APIs are designed on the principle that the client should not depend

upon previous API calls, hence why the rest architecture is made to be stateless. Therefore, rest APIs should be made such that their methods are idempotent when possible.

Question 7: What's the difference between idempotent and safe HTTP methods?
An idempotent method is one that can be called multiple times without changing responses. A safe method is one that does not change a resource, meaning it may read but not write. For example, a GET method is considered to be safe.

Question 8: Explain caching in a RESTful architecture?
A client may request the same data from a REST API many times. In this case, it's beneficial for the response to be cached. This way, less bandwidth is needed, and the client may retrieve data faster. Each REST API contains specific metadata related to the caching of responses. Headers like cache control and expires specify what responses may be cached, by whom, and for how long.

Before we wrap up this video, let's talk a bit about best practices in developing a RESTful web service.

Best practices in developing a RESTful web service:
- Supporting JSON data transfer
- Using proper status codes
- Using URI hierarchy to represent resource relationships
- Using idempotent HTTP methods
- Using caching
- Incorporating security measures

REST APIs can be easy to use and are powerful, especially when used properly. I hope this video helped paint a deeper picture about REST APIs. For more tech interview prep content, Exponent has the best resources to help you ace your interview, including in-depth interview courses, private coaching, and a community of experts ready to help you prep for even the toughest questions. Go to tryexponent.com to become a member today. Thanks for watching.

In this video, we're going to talk about nine common REST API interview questions. My name is Kevin Way, I'm a product manager, and I'm here to help you land your dream job in Tech.
[Music]

REST APIs are some of the most prominent software architectures among web services. Having a strong understanding of this architecture and its applications is critical to passing your software engineering interview. I'll be taking you through nine common REST API questions and answers to help you review some topics in ACER.
Question 1: How are REST APIs stateless?
How are REST APIs stateless? To answer this question, we need to first understand the meaning of statefulness and statelessness. An application, API, or web service that is considered stateful stores data from the client on its own servers. For example, if a username and password were passed from the client to the server as a form of authentication and the server stores that data, then the web server is stateful. This is

because the server is now storing data from the client. On the other hand, the REST architecture requires that the client's state is not stored on the server. Instead, each request made by the client must contain all necessary information for that particular HTTP method. That's how REST APIs are stateless.

Question 2: Explain the HTTP methods
Alright, onto question number two: explain the HTTP methods. RESTful web services will use HTTP methods in client requests. The most common methods are GET, POST, PUT, and DELETE. GET fetches a resource from the server. POST requests for a resource to be created on the server. PUT requests for a resource to be updated. DELETE requests for a resource to be deleted. These four methods correspond to CRUD operations or Create, Read, Update, and Delete.

Question 3: Explain the HTTP status codes
RESTful web services use HTTP status codes in server responses. The most common types of status codes are 200 codes, which represent a successful request and response. 400 codes represent a client-side error, and 500 codes represent a server-side error.

Question 4: What is a URI?
URI stands for Uniform Resource Identifier. It identifies every resource in the REST architecture. A URI can be one of two types: URN, which identifies a resource through a unique and persistent name, like a book's ISBN, and URL, which is your typical web address. On the web, URLs will typically be used when designing REST APIs.

Question 5: What are some best practices in making URIs for RESTful web services?
URIs should be mostly standardized when developing a RESTful web service. This way, clients can more easily work with the web service. Here are five best practices when making RESTful URIs:
- Develop them with the understanding that forward slashes indicate hierarchy.
- Use plural nouns for branches.
- Use hyphens for multiple words.
- Use lowercase.
- Refrain from using file extensions.

Question 6: Differences between REST and SOAP?
Question six: what are the differences between REST and SOAP? Well, REST is an architecture used to develop web services. SOAP, or Simple Object Access Protocol, serves as a protocol for exchanging structured information by way of APIs. While REST has flexible standards, SOAP standards are much more strict. Their implementation and statefulness often mean that the client and server are closer connected. Additionally, while REST allows for data transfer in JSON, XML, and others, SOAP only supports XML. In general, SOAP is a stricter and more niche alternative to REST. It's used in cases where more regulated and stable data needs to be transferred.

Question 7: Differences between REST and AJAX?
Differences between REST and AJAX. AJAX refers to Asynchronous JavaScript and XML. It is a collection of web technologies that allow for asynchronous web applications using the built-in XML HTTP request object. While REST API refers to an architecture for handling

HTTP requests, AJAX refers to a collection of web technologies for making asynchronous web requests. This means that a REST API may handle AJAX clients, and AJAX may be used to send RESTful requests. But a REST API could never be implemented nor replaced by AJAX.

Question 8: What are some tools used to develop and test REST APIs?
Tools used to develop and test REST APIs. The tools used to develop REST APIs depend on the language used. Different tools exist depending on whether the language is Java, Node.js, Python, etc. For example, in the case of Node.js, frameworks like Express are commonly used. For testing, there are a few commonly used tools like Postman, which is used to ensure resources are being delivered and each facet of a REST API is performing as designed.

Question 9: Real-world examples of REST APIs?
What are some real-world examples of REST APIs? REST APIs can be seen in nearly every facet of the web. By going to the Exponent website, there is a GET request sent to a server at the URL for the necessary HTML to display to the client. Most commonly, as a developer, REST APIs are used to manipulate data from a particular database. Using the four main HTTP methods, operations like retrieving file data, accessing images, and even hosting a website all require the use of REST APIs.

Now, to wrap up this video, let's discuss the pros and cons of RESTful web services.
Pros and cons of RESTful web services:
Advantages include REST APIs being easy to learn, a wide range of data transfers like JSON and XML, statelessness allowing for a simpler client experience, and scalability due to the independent nature of client and server. Disadvantages include sessions not being able to be maintained because of statelessness, a lack of built-in security measures, the need to be versioned for backward compatibility when being updated, and consistency in URIs being increasingly difficult to maintain for complex projects.

I hope this video helped refresh your memory of REST APIs and gave you a good indication of common questions you might be asked. For more tech interview prep content, Exponent has the best resources to help you ace your interview, including in-depth interview courses, private coaching, and a community of experts ready to help you prep for even the toughest questions. Go to tryexponent.com to become a member today. Thanks for watching.

HTTP Catch
Today's topic is Web Cache or the HTTP Caching Headers. As users, we get frustrated when the websites are slow. And to improve performance, we normally use some kind of caching, which can either be on the server-side in the form of application-level caching, for example using Redis or Memcached, or it can be on the client-side. So when the request comes for a resource, it will be checked in the cache. If it is not found in the cache, then the request will be sent to the server, the response will be fetched from there, and then it will be cached. And in the future, when the same resource is requested again, there will be no more calls to the server, the item will be found in the cache and it will be returned from there. This brings us a couple of benefits. So, first of all, we have faster response times. Because the requests are not going to the server, so our response times are going to be much lower. Secondly, because the items are returned from the cache and not the server, our server

bandwidth usage is going to decrease also. And the third one is less load on the server. Because not all the requests are going to the server, the load on the server is also going to decrease.

Alright, so now that we know the benefits of caching and what caching is, let's look at the caching locations. So the cache can be in one of the three locations. First of all, we have the browser cache, so the browser you use might cache the response from any previous visits. It depends on the caching headers which we will see in a minute. And the cache in the browser is private to the specific user using the website. Next, we have the proxy cache which is the cache at the proxy level. I will make a detailed video on what the proxies and the reverse proxies are. But for now, just know that the proxies are installed either by your ISP or by the organization you work at. And the content might also be cached there. And the last one we have is the reverse proxy, which is the proxy that is closer to our server, is set up by the server administrators, and the content might also be cached there.

Okay, now that we know what caching is, what the benefits of caching are, and the common places for caching, let's look at some terminology before we jump on to discuss the caching headers and how caching works. First of all, we have the client which is your browser or any application requesting the server for some resources. Then we have the origin server, which is the source of truth, the place where we find the content from. And then on any location of the caching, we might have stale content, which is the content that is cached but it is expired. And then we have fresh content, which is cached and it is still usable because it has not expired yet. And then we have some actions such as content validation, which is a process of contacting the server to check the validity of the cached content and to get it updated if it is expired. And lastly, we have cache invalidation, which is the process of removing any stale content which is available in the cache.

With that out of the way, let's understand how the content gets cached in any of the locations that we have discussed above. HTTP headers play the key role in caching the content at any of the places. Whenever the server sends the response to the client, it sends the HTTP headers in the response, which are then used by the client or any other intermediate places to cache the content. Let's look at the caching headers and see how they work. We have mainly Expires, Pragma, and Cache-Control.

Cache Control:
Expires and Pragma were there before HTTP 1.1, and they aren't used as much. But you might see them being used for backward compatibility. Cache-Control was introduced in HTTP 1.1, and it is the preferred way of caching, and everything that you can do with Expires and Pragma, it's also possible with Cache-Control. We also have some validator headers which are used by the client to make sure that the cached content is still usable. Let's look at each of these headers in detail.

Expires:
First of all, we have the Expires header. The value for the Expires header is the absolute expiry date till which the response can be cached. Some things that you should note about the Expires header are: the value can't be more than one year. And if you provide an invalid value or return the wrong date format, the response will be considered stale.

Pragma:
Next, we have the Pragma header. The only possible value for that is "no-cache". It is one of the pre-HTTP/1.1 headers and it was used to prevent caching. You should prefer to use Cache-Control over it, but you might still see it being used. The reason for that is the backward compatibility, so that the clients that don't support Cache-Control, they might be using this header to avoid caching.

Cache-Control:
Lastly, in the caching headers, we have the Cache-Control header, which is the preferred caching header and it was introduced in HTTP/1.1. It's a multi-value header, so it can have multiple values or directives, and they determine the caching behavior. Let's look at the possible values for the Cache-Control header.

Private:
First of all, we have the "private" directive. If the cache is set to private, it means that the content is private to the user, and it will only be cached in the client or the browser.

Public:
Next, we have the "public" directive, which means that the cache is available to multiple users and it can be cached publicly at any of the proxies.

No Store:
Then we have "no-store", which means that the content can't be cached anywhere. So every time the client needs the resource, it must make the call to the server for the fresh copy.

No Cache:
Next, we have the "no-cache" value, which means that the content can be cached, but for the client to reuse it, it must first validate it from the server. This is done using the ETag header, for example, which you will see in a second.

Max Age:
Then, we have the "max-age" which tells the client that the content can be cached for the given number of seconds. So if you get a value like this, it would mean that the content can be cached at any public places or the proxies for 3600 seconds, or 60 minutes. And having "private" would mean that the content can only be cached at the client and not any other place for 60 minutes.

S-Max Age:
Next, we have "s-maxage". The prefix "-s" stands for shared. It is the same as max-age but gives the caching duration for the shared places or the proxies. And in cases when you have both the max-age and the s-maxage values present, it will take the max-age value for the client-side caching and s-maxage value for the shared caching.

Must Revalidate:
Next, we have "must-revalidate". Sometimes, it might happen that the server is not reachable or there is a network issue. In that case, the client might serve the cached content, even if it is stale. If you have "must-revalidate" in the value for Cache-Control, it won't allow using the stale content and the content must be revalidated from the server

before serving. If you look at the example value that I have here, if you don't have the "must-revalidate" directive, the client might keep serving the cached content even after it has expired, i.e. after the max-age of 600 seconds has passed and the server is not reachable for the fresh content. But if you have "must-revalidate" present, the client can only serve the cached content for 600 seconds. And after that, it must fetch it from the server. And if the server is not reachable, it won't serve the cached content because it is stale.

Proxy Revalidate:
Next, we have "proxy-revalidate", which is the same as "must-revalidate" but it applies to the proxies or the shared caching. So if the given header is present, it would mean that the client may keep serving the stale content available at the client if, let's say, it is not connected to the internet or it has network issues. But if it is able to connect to the proxy and there is stale content at the proxy, the proxy can't use the stale content and it must get it from the server.

Mixing Cache Control Values:
A note about mixing the values for Cache-Control: you can mix the Cache-Control values however you want to achieve the caching behavior you require for your application. For example, the value that I'm showing here right now means that the cache is going to be public, so it can be stored at the proxies level as well as on the client with the max-age of 3600 seconds. We are setting the client-level caching to be one hour. And with the s-maxage of 600 seconds, we are setting the proxy-level caching to be 600 seconds. Also, since we have "must-revalidate" present, the client will never serve the stale content, and it must fetch the fresh content from the server whenever it is expired.

Up until now, we only discussed how the content is cached and how long the cached content is to be considered fresh, but we did not discuss how the client does the validation from the server. For that, the server might send one or more validation headers in the response, which are used by the client to make a conditional request to the server.

ETags:
First of all, we have ETags or the entity tags. When serving the content, the server normally sends an ETag header in the response, which is just a unique identifier associated with the resource. The client then uses this ETag to make a request to the server to check if the content has been changed. Let's consider a scenario where the client requested an image, and the server responded with the image and the following cache control and ETag headers.

Now the client will keep using the image from the cache for 3600 seconds or one hour. And after one hour, the client will make the call to the server with the "If-None-Match" header, and it will send the ETag of the resource in the call. The server will then match the ETag of the resource with the newly available resource. If it doesn't match, the server will respond with the new ETag and the new resource which will then be used to replace the current image. If it does match the existing image, the server will respond with the status code of 304 which means "Not Modified". And the client will renew the cache for another 3600 seconds.

Before we move on, there are two types of ETags: strong and weak. Strong ETag means that the two resources are exactly the same and there is no difference between them at all. Weak tags are prefixed with "W/" and they mean that the two resources are not strictly the same, but could be considered the same.

Last-Modified:
Next, we have "Last-Modified", which indicates the date and time when the content was last modified. When the content gets stale in the client, it will make the conditional request with the last-modified date using the "If-Modified-Since" header, which will then be used by the server to either return 304 — Not Modified, or to return the new response.

No Headers:
Okay, so what about the case when the server did not send any of these validation headers? Well, in that case, there will be no calls to validate and refresh the existing cache, and the fresh content will be requested as soon as the content gets stale.

Alright, now that we know the caching headers and the validation, you might be asking yourself what should be your caching strategy? There is no global answer to that. It varies and depends on the nature of your application, but normally it might fall into two categories: light caching for the items where you need the client to validate with the server. For example, your HTML files where you can make the client cache it but it must validate it with the server before using it. This way we are making sure that the client always gets the latest HTML whenever we have it available on the server, but if the HTML files have not been updated, it can avoid the download and serve the one that it has cached in the browser.

For the dynamic content, it depends on your needs. So, for example, if you have an RSS feed for your blog, you might cache it publicly for two hours. But if you have a product listing, let's say JSON for an E-Commerce website, you might have a different kind of caching headers. In the response for the assets, you can have aggressive caching. So, for example, here I am caching my CSS, JavaScript, and images for a year in the public caches and I might use the fingerprinted file names. So, for example, my build system will generate a different file name for each build and inject them into my HTML. So whenever the client will request the HTML, it will see that the links have been changed, so it will load the new assets from the original server instead of serving them from the cache.

And lastly, to debug the HTTP headers, you can either do it with the curl command with the option `-I`, which will give you the headers from the response, or you can do it from the dev tools in the network tab.

I hope you enjoyed the video. If you have any questions or comments, leave them below and I will see you in the next one.

Kafka

Microservices:

Java Questions:
what method overriding and overloading

https://www.javatpoint.com/method-overloading-in-java

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).
-

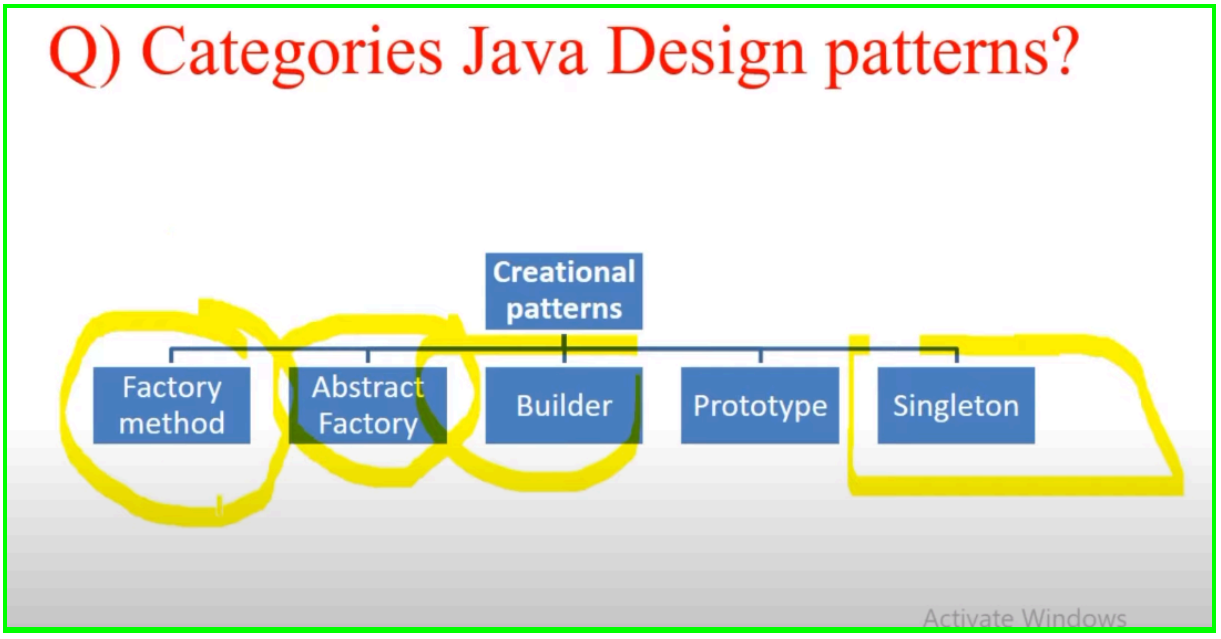
what is compile time polimorphism and run time polimorphism.
what is solid design principle and what is linskov principle : NA

https://www.baeldung.com/solid-principles
https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/

what is singleton Design principal , factory, chain of responsibilty, abstract design principal

```java
class Singleton
{
    private static volatile Singleton obj  = null;
    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj == null)
        {
            // To make thread safe
            synchronized (Singleton.class)
            {
                // check again as multiple threads
                // can reach above step
                if (obj==null)
                    obj = new Singleton();
            }
        }
        return obj;
    }
}
```

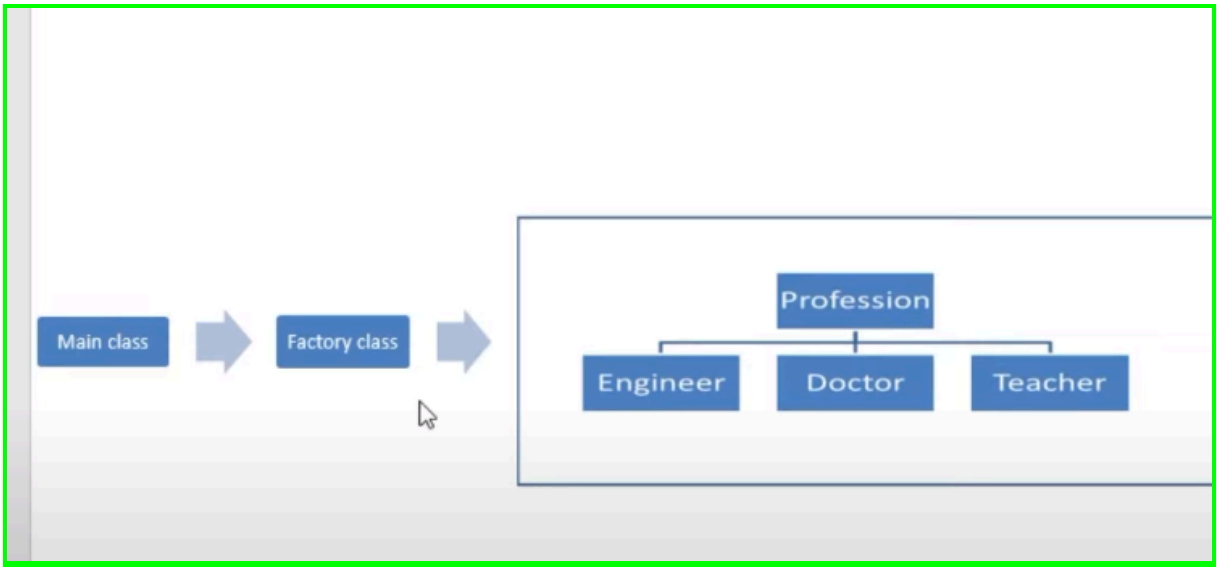
Design pattern



Q) What Is Factory Pattern?

- In the Factory pattern, we don't expose the creation logic to the client and refer the created object using a standard interface.
- The Factory Pattern is also known as Virtual Constructor.

- Steps:
- 1) create main class which call factory class.
- 2) Factory class returns required class instance



FactoryPatternMainClass.java

```java
package factory.pattern;

public class FactoryPatternMainClass {

    public static void main(String[] args) {
        ProfessionFactory professionFactory = new ProfessionFactory();

        Profession doc = professionFactory.getProfession("Doctor");
        doc.print();

    }

```

```java
package factory.pattern;

public class Teacher implements Profession{

    @Override
    public void print() {
        System.out.println("In Print of Teacher class");

    }

}

```

```java
package factory.pattern;

public class ProfessionFactory {

    public Profession getProfession(String typeOfProfession){
            if(typeOfProfession == null){
                return null;
            }
            if(typeOfProfession.equalsIgnoreCase("Doctor")){
                return new Doctor();

            } else if(typeOfProfession.equalsIgnoreCase("Engineer")){
                return new Engineer();

            } else if(typeOfProfession.equalsIgnoreCase("Teacher")){
                return new Teacher();
            }

            return null;
        }

}

```

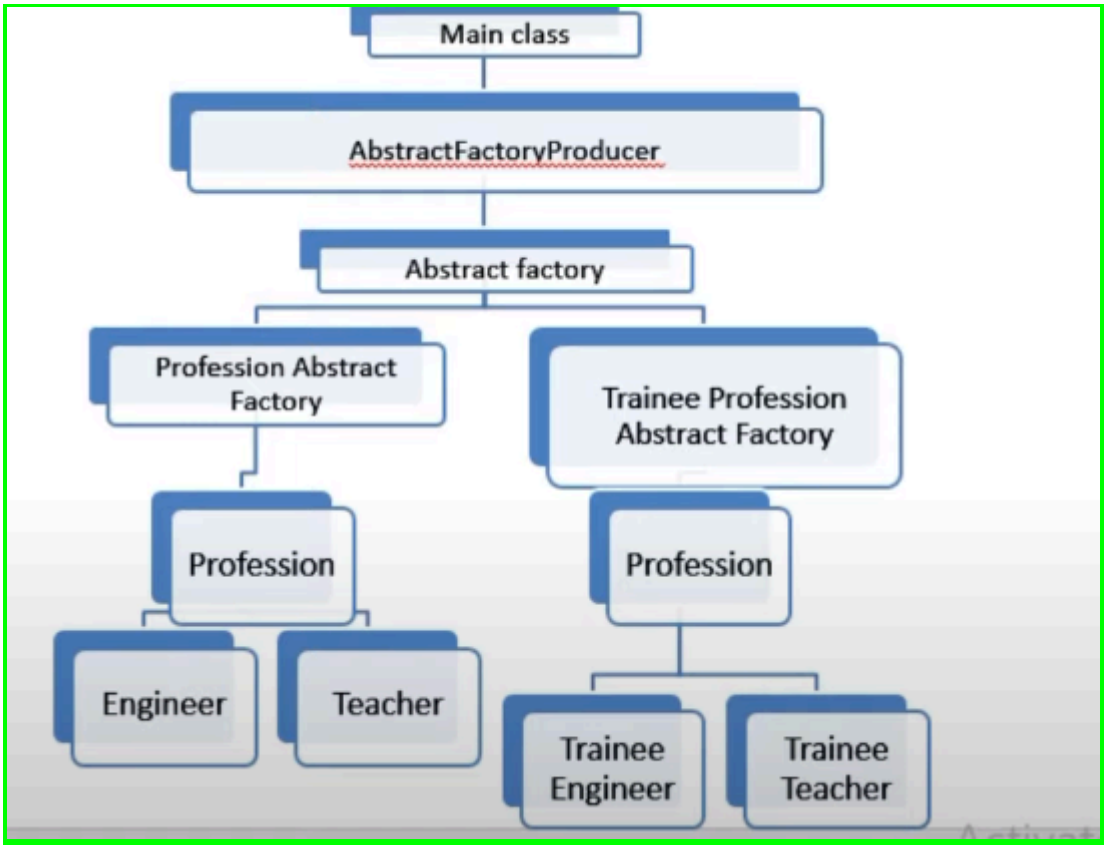
Abstract Factory

Q) What Is Abstract Factory Pattern?

- This factory is also called as factory of factories.
- Abstract Factory lets a class returns a factory of classes.
- So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

- An Abstract Factory Pattern is also known as Kit.

- Steps:
- 1) create main class which call factory of factory class.
- 2) Factory of factory / factory Producer creates instance of factory class.
- 3) Factory class returns required class instance



FactoryPatte... | Doctor.java | Engineer.java | Teacher.java

```java
package abstract_factory.pattern;

public interface Profession {

    void print();

}
```

FactoryPatte... | Profession.java | Teacher.java | TraineeTeac... | Engineer.java | Tr

```java
package abstract_factory.pattern;

public class TraineeEngineer implements Profession{

    @Override
    public void print() {
        System.out.println("In Print of Trainee Engineer class");

    }

}
```

Profession.java | Teacher.java | TraineeTeac... | Engineer.java | TraineeEngi... | Profession

```java
package abstract_factory.pattern;

public class ProfessionAbstractFactory extends AbstractFactory{

    @Override
    public Profession getProfession(String typeOfProfession){

        if(typeOfProfession == null){
            return null;
        }

        else if(typeOfProfession.equalsIgnoreCase("Engineer")){
            return new Engineer();

        } else if(typeOfProfession.equalsIgnoreCase("Teacher")){
            return new Teacher();
        }

        return null;
    }

}
```

Teacher.java | TraineeTeac... | Engineer.java | TraineeEngi... | ProfessionAb... | TraineeProfe...

```java
package abstract_factory.pattern;

public class TraineeProfessionAbstractFactory extends AbstractFactory{

    @Override
    public Profession getProfession(String typeOfProfession){

        if(typeOfProfession == null){
            return null;
        }

        else if(typeOfProfession.equalsIgnoreCase("Engineer")){
            return new TraineeEngineer();

        } else if(typeOfProfession.equalsIgnoreCase("Teacher")){
            return new TraineeTeacher();
        }

        return null;
    }

```

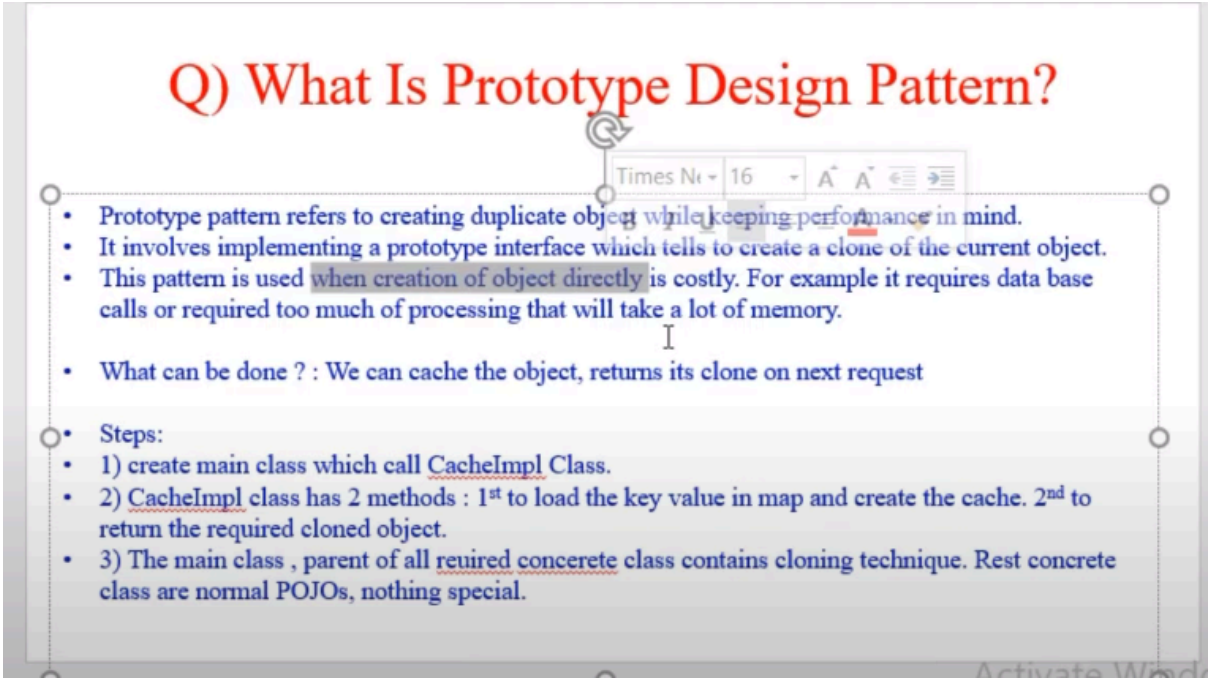
```java
package abstract_factory.pattern;

// this is factory of factory , gives u factory instance which wil in turn give u required class ob
public class AbstractFactoryProducer {

    public static AbstractFactory getProfession(boolean isTrainee) {
        if(isTrainee) {
            return new TraineeProfessionAbstractFactory();
        }
        else {
            return new ProfessionAbstractFactory();
        }
    }

}

```

TraineeTeac... | Engineer.java | TraineeEngi... | ProfessionAb... | TraineeProfe... | AbstractFac...

```java
package abstract_factory.pattern;

public abstract class AbstractFactory {

    abstract Profession getProfession(String typeOfProfession);

}

```

```java
package abstract_factory.pattern;

public class FactoryPatternMainClass {

    public static void main(String[] args) {

        //first get factory instance from factory of factory class, if u want trainees.
        AbstractFactory abstractFactory = AbstractFactoryProducer.getProfession(true);

        Profession engg = abstractFactory.getProfession("Engineer");

        engg.print();

    }

}
```

1)**Go to factory (toothpaste)**
2)Factory goes to intermediate mactory(colgate, patanjali)
3)Intermediate factory internally give you real object (Colgate derivative products, patanjali 4)derivative product)



code decode - code_decode/src/prototype/pattern/ProfessionCache.java - Spring Tool Suite 3

File Edit Source Refactor Navigate Search Project Run Window Help

Quick Access

Package Explorer

- code_decode
  - JRE System Library [jre1.8.0_181]
  - src
    - abstract_factory.pattern
    - builder.design.pattern
    - code_decode
    - factory.pattern
    - prototype.pattern
      - Doctor.java
      - Engineer.java
      - Profession.java
      - ProfessionCache.java
      - ProtoTypePatternMainClass.java
      - Teacher.java
    - singelton.design.pattern
  - demo
- Servers

Console | MainClass.java | AbstractFac... | SingeltonCla... | Profession.java | ProfessionCa...

```
1  package prototype.pattern;
2
3  import java.util.Hashtable;
4
5  public class ProfessionCache {
6
7      private static Hashtable<Integer, Profession> professionMap  = new Hashtable<Integer, Professic
8
9
10     public static Profession getCloneNewProfession(int id) {
11         Profession cachedProfessionInstance = professionMap.get(id);
12       return (Profession) cachedProfessionInstance.cloningMethod();
13     }
14
15     public static void loadProffesionCache() {
16             Doctor doc = new Doctor();
17             doc.id = 1;
18             professionMap.put(doc.id,doc);
19
20             Engineer engineer = new Engineer();
21             engineer.id = 2;
22             professionMap.put(engineer.id,engineer);
23
24             Teacher teacher = new Teacher();
25             teacher.id = 3;
26             professionMap.put(teacher.id,teacher);
27
```

Activate Windows
Go to Settings to activate Windows.

Writable | Smart Insert | 7 : 32

Console | MainClass.java | SingeltonCla... | Profession.java | ProfessionCa... | Doctor.java

```
1  package prototype.pattern;
2
3  public class Doctor extends Profession{
4
5      @Override
6      public void print() {
7          System.out.println("In Print of Doctor class");
8
9      }
10
11 }
12
```

Profession.java | ProfessionCa... | Doctor.java | Engineer.java | Teacher.java | ProtoTypePa...

```java
package prototype.pattern;

public class ProtoTypePatternMainClass {

    public static void main(String[] args) {
        ProfessionCache.loadProfessionCache();

        Profession docProfession = ProfessionCache.getCloneNewProfession(1);
        System.out.println(docProfession);

        Profession engProfession = ProfessionCache.getCloneNewProfession(2);
        System.out.println(engProfession);

        Profession teachProfession = ProfessionCache.getCloneNewProfession(3);
        System.out.println(teachProfession);

        Profession docProfession2 = ProfessionCache.getCloneNewProfession(1);
        System.out.println(docProfession2);



    }

}
```

Activate Windows

Q) What Is Builder Design Pattern?

- Builder Pattern refers to approach that focuses on constructing a complex object from simple objects using step-by-step approach.
- Major roles used in this design patterns are :
  - **Complex Object / Final Product** – e.g. house – complex object which we will generate with builder design pattern
  - **Builder** – abstract class / interface that defines all ways to create the product. It also has getFinalProduct method that will finally return complex object.
  - **ConcreteBuilder** – multiple Builder Impls that will give different final objects which are complex to design,
  - **Director**: Controls complex object creation. It has 2 main goals : 1st to call appropriate concrete builder class to create correct complex object. 2nd to return that complex object.

Activate Window

Q) What Is Builder Design Pattern?

- Steps:
- Create complex class POJO.
- Create Builder Interface/Abstract class which has definitions.
- Create concrete Builder class that has implementations.
- Create Director that will have responsibility to call correct concrete Builder and then return final object.
- Create main class that will initialize Director , and call method of director finally which will in turn return complex object instance required.

Package Explorer

- code_decode
  - JRE System Library [jre1.8.0_181]
  - src
    - abstract_factory.pattern
    - builder.design.pattern
      - Builder.java
      - Director.java
      - EarthQuakeResistantBuilder.java
      - FloodResistantBuilder.java
      - Home.java
      - MainClass.java
    - code_decode
    - factory.pattern
    - prototype.pattern
    - singelton.design.pattern
  - demo
  - Servers

Console | Profession.java | ProfessionCa... | ProtoTypePa... | Home.java | MainClass.java

```
1  package builder.design.pattern;
2
3  public class Home {
4
5      public String floor;
6      public String walls;
7      public String terrace;
8
9  }
10
```

Console | Profession.java | ProfessionCa... | ProtoTypePa... | Home.java | MainClass.java

```java
package builder.design.pattern;


public class MainClass {
    public static void main(String[] args) {

        //create object of required homebuilder
        EarthQuakeResistantBuilder earthQuakeResistantBuilder = new EarthQuakeResistantBuilder();

        //create object of director that will keep an eye on your builder
        Director director = new Director(earthQuakeResistantBuilder);

        director.manageRequiredHomeConstruction();
        Home homeConstructedAtTheEnd = director.getComplexObjectOfHome();

        System.out.println(homeConstructedAtTheEnd);
        System.out.println(homeConstructedAtTheEnd.floor);
    }

}
```

Activate Windows
Go to Settings to activate Windows

Writable | Smart Insert | 1 : 1

Package Explorer

- code_decode
  - JRE System Library [jre1.8.0_181]
  - src
    - abstract_factory.pattern
    - builder.design.pattern
      - Builder.java
      - Director.java
      - EarthQuakeResistantBuilder.java
      - FloodResistantBuilder.java
      - Home.java
      - MainClass.java
    - code_decode
    - factory.pattern
    - prototype.pattern
    - singelton.design.pattern
- demo
- Servers

Console | Profession.java | ProfessionCa... | Home.java | MainClass.java | Director.java

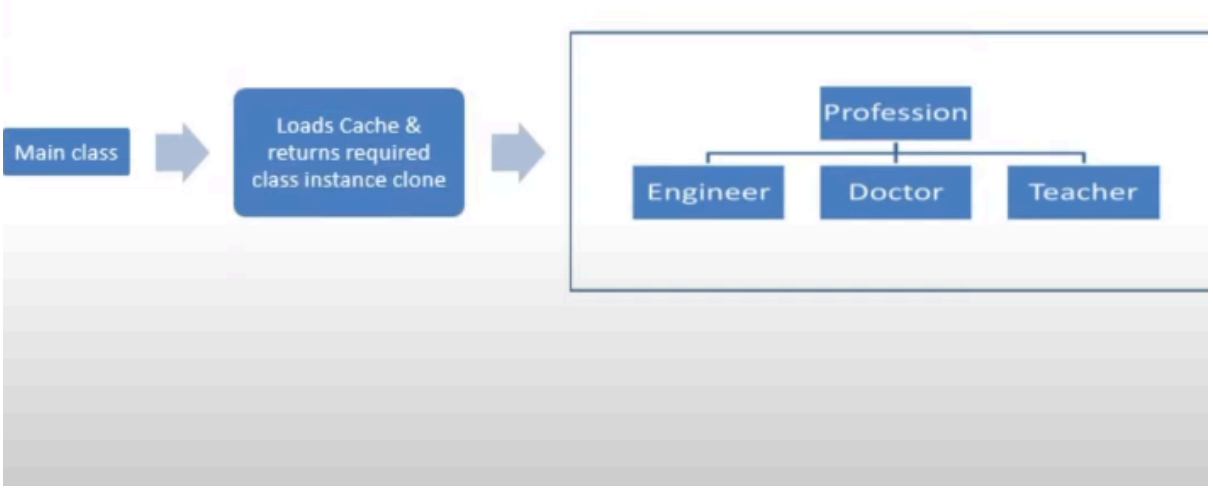
```java
package builder.design.pattern;

class Director {

    private Builder builder;

    public Director(Builder builderType) {
        this.builder = builderType;
    }

    public Home getComplexObjectOfHome()
    {
        return this.builder.getComplexHomeObject();
    }

    public void manageRequiredHomeConstruction()
    {
        this.builder.buildFloor();
        this.builder.buildWalls();
        this.builder.bulidTerrace();
    }

}
```

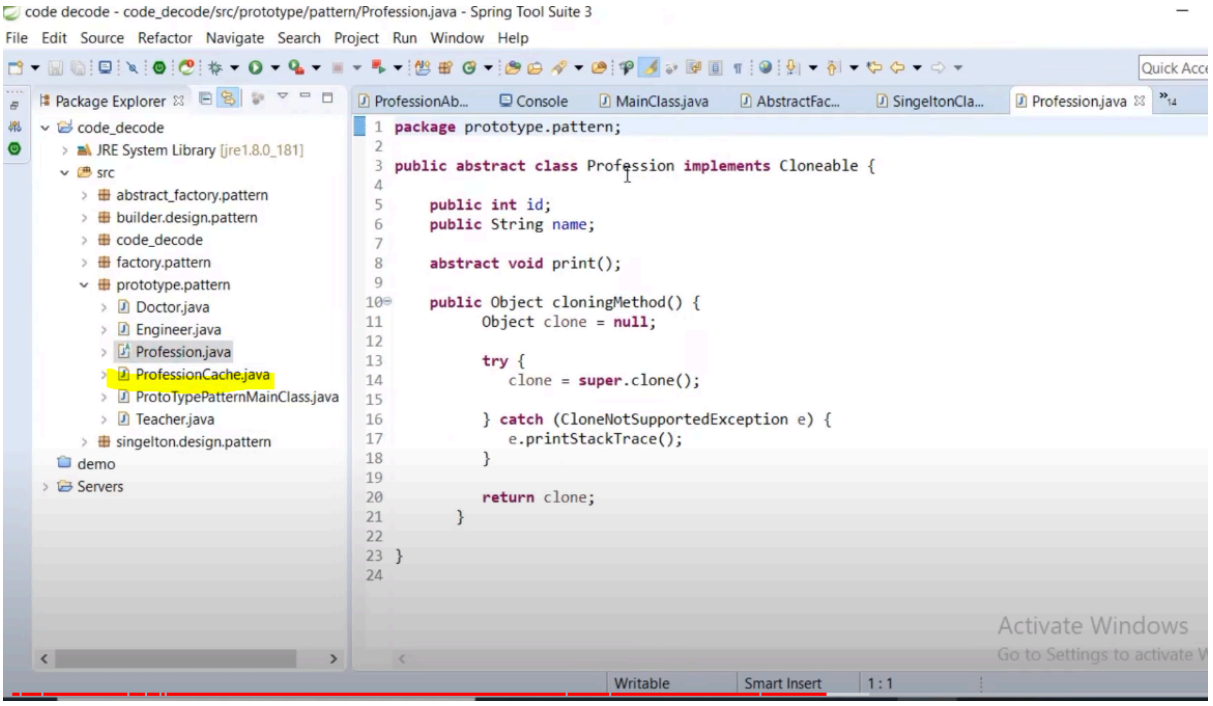
```java
package builder.design.pattern;

class EarthQuakeResistantBuilder implements Builder {

    private Home earthquakeResistantHome = new Home();


    @Override
    public void buildFloor() {
        this.earthquakeResistantHome.floor = "Wooden floor";

    }

    @Override
    public void buildWalls() {
        this.earthquakeResistantHome.walls = "Wooden walls";

    }

    @Override
    public void bulidTerrace() {
        this.earthquakeResistantHome.terrace = "Wooden terrace";

    }

    @Override
    public Home getComplexHomeObject() {
```

```java
    @Override
    public void buildFloor() {
        this.earthquakeResistantHome.floor = "Wooden floor";

    }

    @Override
    public void buildWalls() {
        this.earthquakeResistantHome.walls = "Wooden walls";

    }

    @Override
    public void bulidTerrace() {
        this.earthquakeResistantHome.terrace = "Wooden terrace";

    }

    @Override
    public Home getComplexHomeObject() {
        return this.earthquakeResistantHome;
    }


}
```

code decode - code_decode/src/builder/design/pattern/Builder.java - Spring Tool Suite 3

```java
package builder.design.pattern;

interface Builder {

    public void buildFloor();

    public void buildWalls();

    public void bulidTerrace();


    public Home getComplexHomeObject();

}
```





Structural Design Pattern

Q) What is Proxy Design Patterns?

- It's a Structural design pattern.
- It lets you provide a substitute or placeholder for another object.
- A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

RealSubjectClass.java | ProxyClass.java | ClientClass.java

```java
package structural.design.patterns.demo.proxy.dp;


public class ClientClass {

    public static void main(String[] args) {

        Subject  proxy = new ProxyClass();
        proxy.method();

    }
}
```

RealSubjectClass.java | ProxyClass.java | ClientClass.java | Subject.java

```java
package structural.design.patterns.demo.proxy.dp;

//extends
public class ProxyClass extends RealSubjectClass {

    @Override
    public void method() {

        System.out.println("Hi, I am Proxy, i will perform authentication and security checks");

        // Logic to check if user is authentic or not. If yes then call real object
        // method else dont call below method

        System.out.println("Calling real method of Real subject implementing class after the call is aut
        super.method();



    }

}
```

RealSubjectClass.java | ProxyClass.java | ClientClass.java | Subject.java

```java
package structural.design.patterns.demo.proxy.dp;

//implementation of Subject
public class RealSubjectClass implements Subject {

    @Override
    public void method() {
        System.out.println(" I Am an actual implementation of Subject, rest all are proxies :D");

    }

}

```

Architecture of Proxy Design Pattern?

Q) Roles in Proxy Design Pattern?

Major roles used in this design patterns are :

- **Subject** – is an interface which expose the functionality available to be used by the clients.
- **Real Subject** – is a class implementing Subject and it is concrete implementation which needs to be hidden behind a proxy.
- **Proxy** – hides the real object by extending it and clients communicate to real object via this proxy object. Usually frameworks create this proxy object when client request for real object.
- Demo:

Advantages of Proxy Design Pattern?

- Access control / **protection** given by proxy object : This is when you want only specific clients to be able to use the service object.
- **Caching** request results (caching proxy). This is when you need to cache results of client requests and manage the life cycle of this cache, especially if results are quite large.
- **Logging** requests (logging proxy). This is when you want to keep a history of requests to the service object.

Where to use Flyweight DP?

- When your application needs large number of objects that shares most of the common attributes called Intrinsic Attributes or properties and only few unique attributes called as Extrinsic attributes / properties.
- We need to control the memory consumption by large number of objects – by creating fewer objects and sharing them across.

What is Flyweight Design Patterns?

- It's a Structural design pattern.
- A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context.

RealSubjectClass.java ProxyClass.java ClientClass.java Subject.java Con

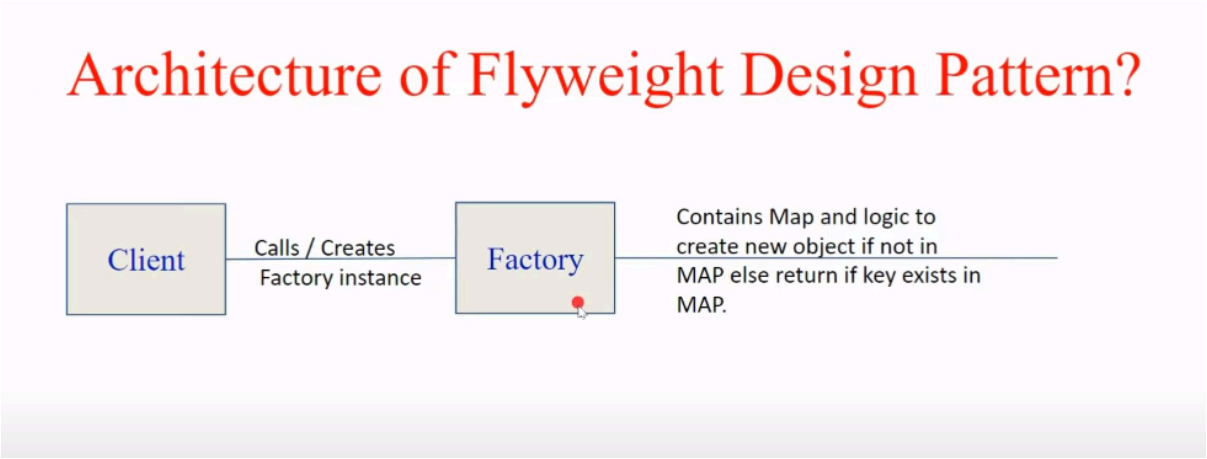
```java
package structural.design.patterns.demo.Flyweight.DP;

public class CommonShareableClass {

    public static int eyes = 2;
    public static int nose = 1;
    public static int legs = 4;
    public static int tail = 1;


}

```

RealSubjectClass.java ProxyClass.java ClientClass.java Subject.java CommonShareable

```java
package structural.design.patterns.demo.Flyweight.DP;

public interface Animal {

    void setName(String name);
    public void printAnimalAttributes();


}

```

RealSubjectClass.java ProxyClass.java ClientClass.java Subject.java CommonShareableClass.java Cat.java Animal.java

```java
package structural.design.patterns.demo.Flyweight.DP;

public class Cat implements Animal{

    private String name = null;


    @Override
    public void setName(String name) {
        this.name = name;
    }

    @Override
    public void printAnimalAttributes() {
        System.out.println("Name of animal is " + this.name);
        System.out.println("Number of eyes of animal is " + CommonShareableClass.eyes);
        System.out.println("Number of legs of animal is " + CommonShareableClass.legs);
        System.out.println("Number of nose of animal is " + CommonShareableClass.nose);
        System.out.println("Number of tail of animal is " + CommonShareableClass.tail);

    }

```


Architecture of Flyweight Design Pattern?
Client
Calls / Creates
Factory instance
Factory
Contains Map and logic to
create new object if not in
MAP else return if key exists in
MAP.

SOLID

3. Single Responsibility

Let's begin with the single responsibility principle. As we might expect, this principle states that **a class should only have one responsibility. Furthermore, it should only have one reason to change.**

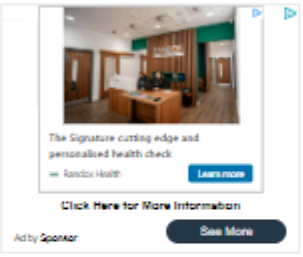
How does this principle help us to build better software? Let's see a few of its benefits:

1. **Testing** – A class with one responsibility will have far fewer test cases.
2. **Lower coupling** – Less functionality in a single class will have fewer dependencies.
3. **Organization** – Smaller, well-organized classes are easier to search than monolithic ones.

For example, let's look at a class to represent a simple book:

```
public class Book {

    private String name;
    private String author;
    private String text;

    //constructor, getters and setters
}
```

In this code, we store the name, author and text associated with an instance of a *Book*.



Let's now add a couple of methods to query the text:

```
public class Book {

    private String name;
    private String author;
    private String text;

    //constructor, getters and setters

    // methods that directly relate to the book properties
    public String replaceWordInText(String word, String replacementWord){
        return text.replaceAll(word, replacementWord);
    }

    public boolean isWordInText(String word){
        return text.contains(word);
    }
}
```

Now our *Book* class works well, and we can store as many books as we like in our application.

But what good is storing the information if we can't output the text to our console and read it?

Let's throw caution to the wind and add a print method:

```
public class BadBook {
    //...

    void printTextToConsole(){
        // our code for formatting and printing the text
    }
}
```

However, this code violates the single responsibility principle we outlined earlier.

To fix our mess, we should implement a separate class that deals only with printing our texts:

```
public class BookPrinter {

    // methods for outputting text
    void printTextToConsole(String text){
        //our code for formatting and printing the text
    }

    void printTextToAnotherMedium(String text){
        // code for writing to any other location..
    }
}
```

Awesome. Not only have we developed a class that relieves the *Book* of its printing duties, but we can also leverage our *BookPrinter* class to send our text to other media.

Whether it's email, logging, or anything else, we have a separate class dedicated to this one concern.

4. Open for Extension, Closed for Modification

It's now time for the O in SOLID, known as the **open-closed principle.** Simply put, **classes should be open for extension but closed for modification. In doing so, we stop ourselves from modifying existing code and causing potential new bugs** in an otherwise happy application.

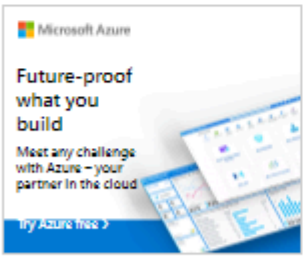
Of course, the **one exception to the rule is when fixing bugs in existing code.**

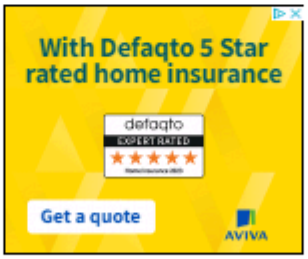
Let's explore the concept with a quick code example. As part of a new project, imagine we've implemented a *Guitar* class.

It's fully fledged and even has a volume knob:

```
public class Guitar {

    private String make;
    private String model;
    private int volume;

    //Constructors, getters & setters
}
```

We launch the application, and everyone loves it. But after a few months, we decide the *Guitar* is a little boring and could use a cool flame pattern to make it look more rock and roll.





At this point, it might be tempting to just open up the *Guitar* class and add a flame pattern — but who knows what errors that might throw up in our application.

Instead, let's **stick to the open-closed principle and simply extend our *Guitar* class**:

```
public class SuperCoolGuitarWithFlames extends Guitar {

    private String flameColor;

    //constructor, getters + setters
}
```

By extending the *Guitar* class, we can be sure that our existing application won't be affected.

5. Liskov Substitution

Next on our list is Liskov substitution, which is arguably the most complex of the five principles. Simply put, **if class *A* is a subtype of class *B*, we should be able to replace *B* with *A* without disrupting the behavior of our program.**

Let's jump straight to the code to help us understand this concept:

```
public interface Car {

    void turnOnEngine();
    void accelerate();
}
```

Above, we define a simple *Car* interface with a couple of methods that all cars should be able to fulfill: turning on the engine and accelerating forward.

Let's implement our interface and provide some code for the methods:

```
public class MotorCar implements Car {

    private Engine engine;

    //Constructors, getters + setters

    public void turnOnEngine() {
        //turn on the engine!
        engine.on();
    }

    public void accelerate() {
        //move forward!
        engine.powerOn(1000);
    }
}
```

As our code describes, we have an engine that we can turn on, and we can increase the power.

Business Security Simple.

Securitas UK

Open

But wait — we are now living in the era of electric cars:

```
public class ElectricCar implements Car {

    public void turnOnEngine() {
        throw new AssertionError("I don't have an engine!");
    }

    public void accelerate() {
        //this acceleration is crazy!
    }
}
```

By throwing a car without an engine into the mix, we are inherently changing the behavior of our program. This is **a blatant violation of Liskov substitution and is a bit harder to fix than our previous two principles.**

One possible solution would be to rework our model into interfaces that take into account the engine-less state of our *Car*.

6. Interface Segregation

6. Interface Segregation

The I in SOLID stands for interface segregation, and it simply means that **larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.**

For this example, we're going to try our hands as zookeepers. And more specifically, we'll be working in the bear enclosure.

Let's start with an interface that outlines our roles as a bear keeper:

```
public interface BearKeeper {
    void washTheBear();
    void feedTheBear();
    void petTheBear();
}
```

As avid zookeepers, we're more than happy to wash and feed our beloved bears. But we're all too aware of the dangers of petting them. Unfortunately, our interface is rather large, and we have no choice but to implement the code to pet the bear.

Let's **fix this by splitting our large interface into three separate ones**:

```
public interface BearCleaner {
    void washTheBear();
}

public interface BearFeeder {
    void feedTheBear();
}

public interface BearPetter {
    void petTheBear();
}
```

Now, thanks to interface segregation, we're free to implement only the methods that matter to us:

```
public class BearCarer implements BearCleaner, BearFeeder {

    public void washTheBear() {
        //I think we missed a spot...
    }

    public void feedTheBear() {
        //Tuna Tuesdays...
    }
}
```

And finally, we can leave the dangerous stuff to the reckless people:

```
public class CrazyPerson implements BearPetter {

    public void petTheBear() {
        //Good luck with that!
    }
}
```

Going further, we could even split our *BookPrinter* class from our example earlier to use interface segregation in the

7. Dependency Inversion

The principle of dependency inversion refers to the decoupling of software modules. This way, instead of high-level modules depending on low-level modules, both will depend on abstractions.

To demonstrate this, let's go old-school and bring to life a Windows 98 computer with code:

```
public class Windows98Machine {}
```

But what good is a computer without a monitor and keyboard? Let's add one of each to our constructor so that every *Windows98Computer* we instantiate comes prepacked with a *Monitor* and a *StandardKeyboard*:

```
public class Windows98Machine {

    private final StandardKeyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine() {
        monitor = new Monitor();
        keyboard = new StandardKeyboard();
    }

}
```

This code will work, and we'll be able to use the *StandardKeyboard* and *Monitor* freely within our *Windows98Computer* class.

Problem solved? Not quite. **By declaring the *StandardKeyboard* and *Monitor* with the *new* keyword, we've tightly coupled these three classes together.**

Not only does this make our *Windows98Computer* hard to test, but we've also lost the ability to switch out our *StandardKeyboard* class with a different one should the need arise. And we're stuck with our *Monitor* class too.

Let's decouple our machine from the *StandardKeyboard* by adding a more general *Keyboard* interface and using this in our class:

```
public interface Keyboard { }
```

```
public class Windows98Machine{

    private final Keyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine(Keyboard keyboard, Monitor monitor) {
        this.keyboard = keyboard;
        this.monitor = monitor;
    }
}
```

Here, we're using the dependency injection pattern to facilitate adding the *Keyboard* dependency into the *Windows98Machine* class.

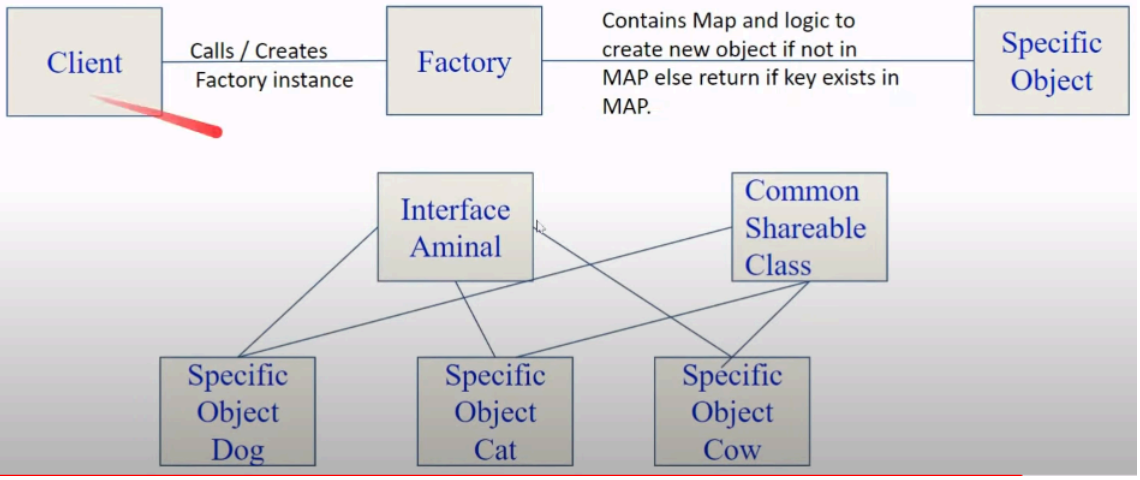
Let's also modify our *StandardKeyboard* class to implement the *Keyboard* interface so that it's suitable for injecting into the *Windows98Machine* class:

```
public class StandardKeyboard implements Keyboard { }
```

Now our classes are decoupled and communicate through the *Keyboard* abstraction. If we want, we can easily switch out the type of keyboard in our machine with a different implementation of the interface. We can follow the same principle for the *Monitor* class.

Excellent! We've decoupled the dependencies and are free to test our *Windows98Machine* with whichever testing framework we choose.

Architecture of Flyweight Design Pattern?

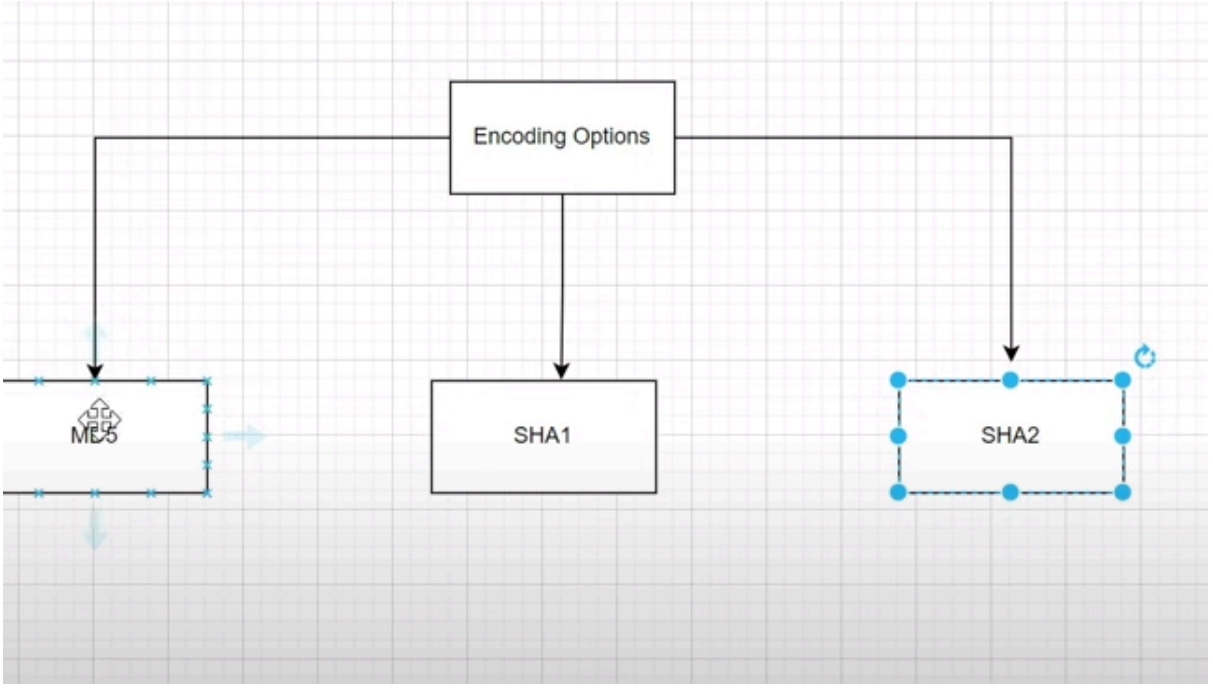



```java
package structural.design.patterns.demo.Flyweight.DP;

public class AnimalClient {

    public static void main(String[] args) {
        Animal catAnimal = AnimalFactory.getCat("CAT");
        catAnimal.printAnimalAttributes();

        Animal catAnimal2 = AnimalFactory.getCat("CAT");
        catAnimal2.printAnimalAttributes();

        Animal cowAnimal = AnimalFactory.getCow("COW");
        cowAnimal.printAnimalAttributes();

        Animal dogAnimal = AnimalFactory.getDog("DOG");
        dogAnimal.printAnimalAttributes();

        System.out.println(catAnimal.hashCode());
        System.out.println(catAnimal2.hashCode());

        System.out.println(cowAnimal.hashCode());

```

```java
public class AnimalFactory {

    private static final HashMap<String, Animal> animalMap = new HashMap<>();

    public static Animal getCat(String name)
    {
        String key = name + "-CAT";

        Animal animal = animalMap.get(key);

        if(animal != null) {
            return animal;
        } else {
            animal = new Cat();
            animal.setName(name);
            animalMap.put(key, animal);
        }
        return animal;
    }

    public static Animal getDog(String name)
```

What is Strategy Design Pattern?

- Strategy design pattern is a behavioral design pattern that enables selecting an algorithm at run-time. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.



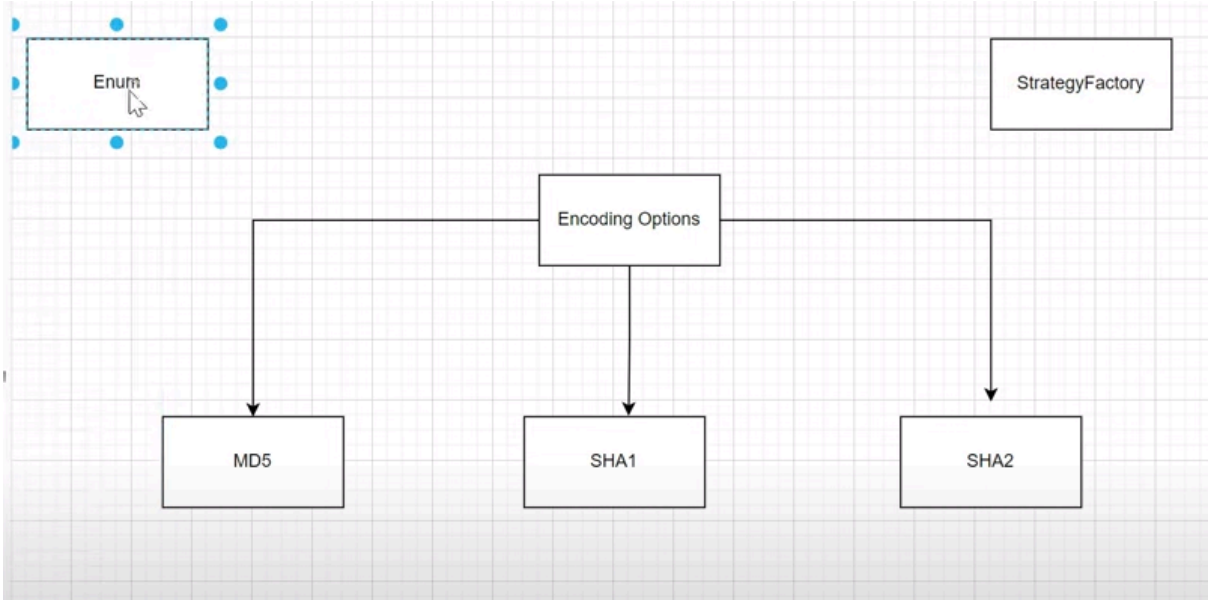
Why is Strategy Design Pattern Needed at all?

- Typically, wetend to bundle all the algorithm logic in the host class, resulting in a monolithic class with multiple switch case or conditional statements. The following example shows the structure of one such class that supports multiple algorithms to encrypt data. family of algorithms to use.

```
public class Encryptor {
    public void encrypt(){
    if (algorithmName.equals("Aes")){
        System.out.println("Encrypting data using AES algorithm");      }
    else if (algorithmName.equals("Blowfish")){
        System.out.println("Encrypting data using Blowfish algorithm");
            }
    /*More else if statements for other encryption algorithms*/
    }
}
```

Why is Strategy Design Pattern Needed at all?

- At runtime, the code loops through the statements to perform encryption based on the client specified algorithm. The result is a **tightly coupled and rigid software** that is **difficult-to-change.**
- You can imagine the consequences if you try to implement a new encryption algorithm, say TripleDES or a custom encryption algorithm.
- You'd have to open and modify the Encryptor class. Also, if an existing algorithm needs to be changed, the Encryptor class will again require modification.
- As you can see, our Encryptor class is a clear violation of the Open Closed principle – one of the SOLID design principles. As per the principle, new functionality should be added by writing new code, rather than modifying existing code.
- You can avoid them by using the Strategy pattern



Package Explorer | Type Hierarchy | JUnit

- DemoDesignPattern [boot]
- Example demo
- Flat
- GlobalException
- GroupingJava8 [boot]
- hack
- Java8Refactor
- JulyDemo [boot]
- Junitdemo [boot]
- KafkaDemo [boot]
- MApVsFlatMAp
- SBProject [boot]
- SerializationDemo
- SingeltonDemo
- SolidPrinciplesDemo
- SQLFinetune_CodeDecodeCRUD [boot] [SQLFinetune_Code
- StrategyDesignPatternDemo [boot]
  - src/main/java
    - com.codedecode.StrategyDesignPattern
      - StrategyDesignPatternDemoApplication.java
    - com.codedecode.StrategyDesignPattern.enums
      - EncodingPatternEnum.java
  - src/main/resources

StrategyDesignPatternDemoApplication.java | EncodingPatternEnum.java

```java
package com.codedecode.StrategyDesignPattern.enums;

public enum EncodingPatternEnum {
    MD5,
    SHA1,
    SHA2

}

```

StrategyDesignPatternDemoApplication.java | EncodingPatternEnum.java | Encryption.java

```java
package com.codedecode.StrategyDesignPattern.encrpt;

import com.codedecode.StrategyDesignPattern.enums.EncodingPatternEnum;

public interface Encryption {

    public void encrypt(String toBeEncrypted);

    public EncodingPatternEnum getEncryptionType();

}

```

StrategyDesignPatternDemoApplication.java | EncodingPatternEnum.java | Encryption.java | MD5Encryption.java

```java
package com.codedecode.StrategyDesignPattern.encrpt;

import com.codedecode.StrategyDesignPattern.enums.EncodingPatternEnum;

public class MD5Encryption implements Encryption {

    @Override
    public void encrypt(String toBeEncrypted) {
        System.out.println("I am encrypting using MD5 encryption Method");

    }

    @Override
    public EncodingPatternEnum getEncryptionType() {
        return EncodingPatternEnum.MD5;
    }

}
```

StrategyDesignPatternDemoApplication.java | EncodingPatternEnum.java | Encryption.java | MD5Encryption.java | SHA1Encryption.java

```java
package com.codedecode.StrategyDesignPattern.encrpt;

import com.codedecode.StrategyDesignPattern.enums.EncodingPatternEnum;

public class SHA1Encryption implements Encryption {

    @Override
    public void encrypt(String toBeEncrypted) {
        System.out.println("I am encrypting using SHA2 encryption Method");

    }

    @Override
    public EncodingPatternEnum getEncryptionType() {
        return EncodingPatternEnum.SHA2;
    }

}
```

Problems | Javadoc | Declaration | Search | Console | Progress | Debug | Call Hierarchy | Maven Repositories | Coverage

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.codedecode.StrategyDesignPattern.encrpt.Encryption;
import com.codedecode.StrategyDesignPattern.enums.EncodingPatternEnum;

@Component
public class EncryptionFactory {

    Map<EncodingPatternEnum, Encryption> map;

    @Autowired
    public EncryptionFactory(Set<Encryption> encryptionTypesSet){
        createStrategy(encryptionTypesSet);
    }

    private void createStrategy(Set<Encryption> encryptionTypesSet) {
        map = new HashMap<EncodingPatternEnum, Encryption>();
        encryptionTypesSet.stream()
                .forEach(encryptionTypes -> map.put(encryptionTypes.getEncryptionType(), encryptionTypes));
    }

    public Encryption findEncrytionType(EncodingPatternEnum encodingPatternEnum) {
        return map.get(encodingPatternEnum);
    }

```

```java
package com.codedecode.StrategyDesignPattern.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import com.codedecode.StrategyDesignPattern.enums.EncodingPatternEnum;
import com.codedecode.StrategyDesignPattern.factory.EncryptionFactory;

@RestController
public class DemoController {

    @Autowired
    EncryptionFactory factory;


    @GetMapping("/encrypt")
    public void encryptWithEncoder(@RequestParam EncodingPatternEnum encodingPatternEnum) {
        factory.findEncrytionType(encodingPatternEnum).encrypt("code decode");
    }

}

```

Real time use of Strategy Design Pattern?

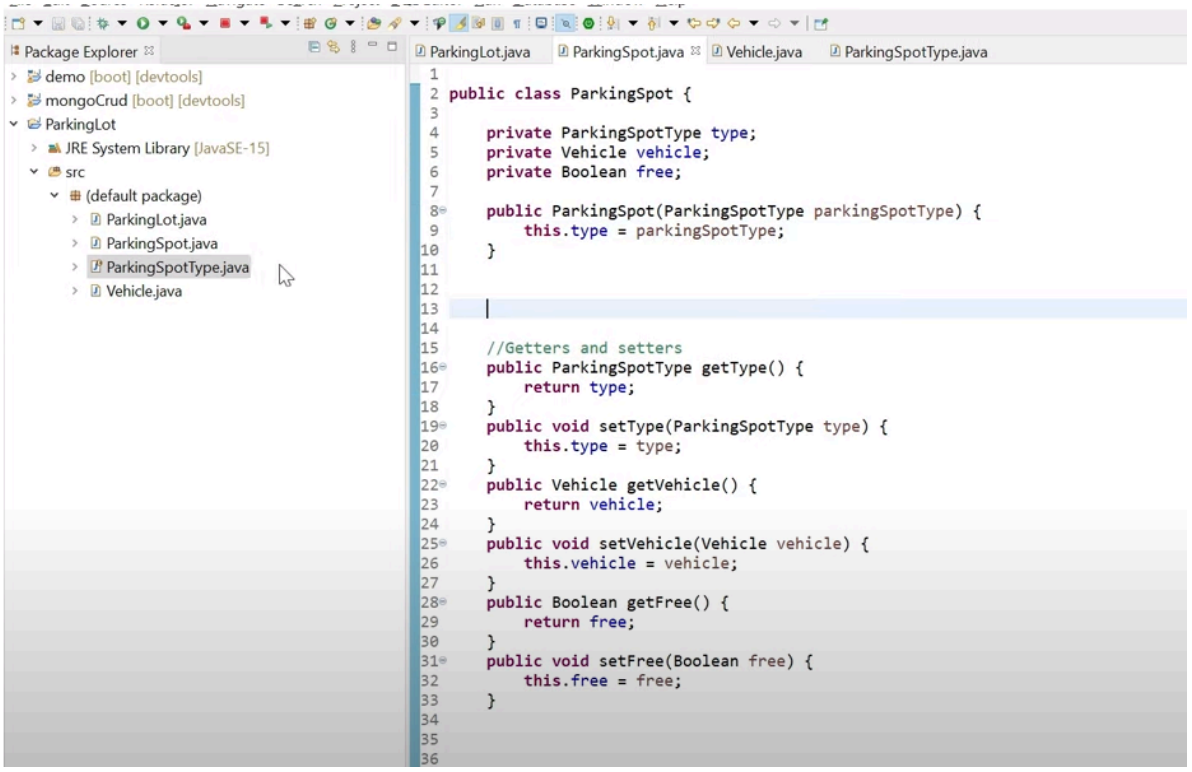
- Sorting class that supports multiple sorting algorithms, such as bubble sort, merge sort, and quick sort.
- a file compression class can support different compression algorithm, such as ZIP, GZIP, LZ4, or even a custom compression algorithm
- data encryption class that encrypts data using different encryption algorithms, such as AES, TripleDES, and Blowfish

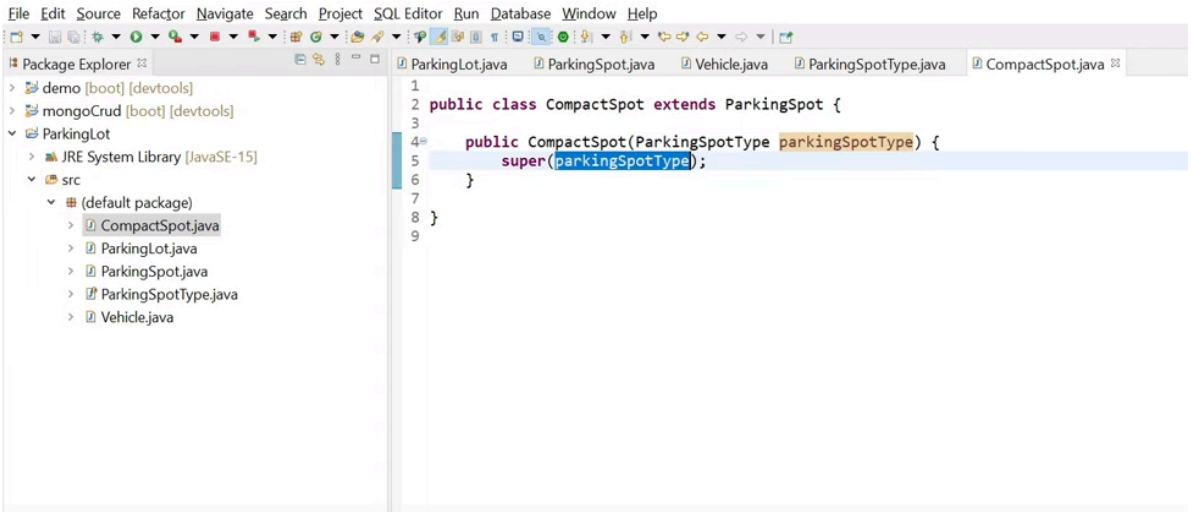
Parking lot

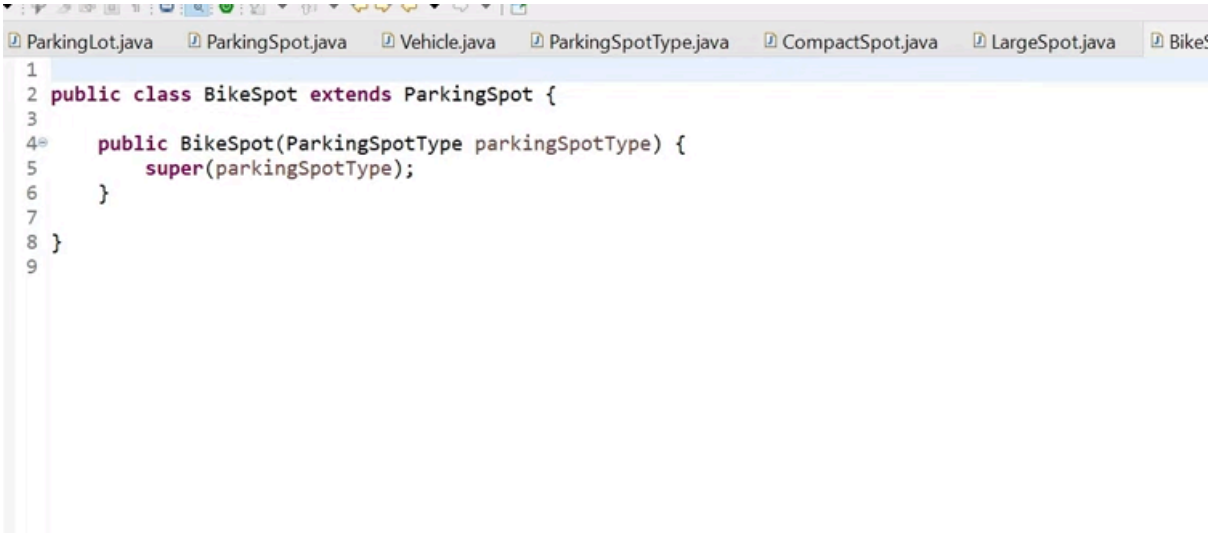
```
Parking Lot
            List <Parking Spot>
            freeCompactSlot
            Large spots
            Bike spots

parkVehicle
Remove/unpark vehicle




Parking Spot
    Compact Spots
    Large spots
    Bike spots



Vehicles
    Truck
    Car
    bike
```

System Requirements

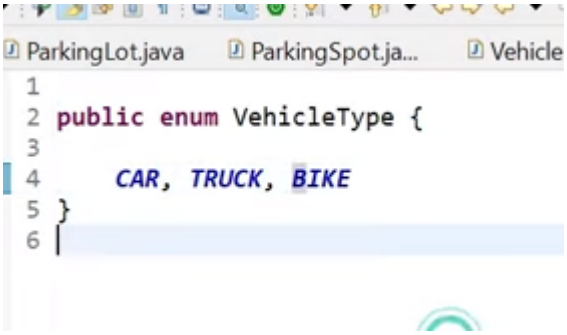
- The parking lot should have multiple slots where customers can park their vehicles.
- The system should not allow more vehicles than the maximum capacity. If the parking is full, the system should be able to show a message
- It will have many parking spots. The system should support multiple types of parking spots such as Compact, Large, Motorcycle, etc.
-







And largeSpot



ParkingLot.java | ParkingSpot.ja... | Vehicle.java | ParkingSpotTy... | CompactSpot.ja...

```java
public class Vehicle {

    private int vehicleNo;
    private VehicleType vehicleType;
    private ParkingSpot parkingSpot;

    public Vehicle(VehicleType vehicleType) {
        this.vehicleType = vehicleType;
    }

    public int getVehicleNo() {
        return vehicleNo;
    }

    public void setVehicleNo(int vehicleNo) {
        this.vehicleNo = vehicleNo;
    }

    public VehicleType getVehicleType() {
        return vehicleType;
    }

    public void setVehicleType(VehicleType vehicleType) {
        this.vehicleType = vehicleType;
    }

    public ParkingSpot getParkingSpot() {
        return parkingSpot;
    }

    public void setParkingSpot(ParkingSpot parkingSpot) {
        this.parkingSpot = parkingSpot;
    }

}
```

ParkingLot.java | ParkingSpot.... | Vehicle.java | CompactSpot.... | LargeSpot.java | BikeSpot.java | Vehic

```java
import java.util.ArrayList;
import java.util.List;

public class ParkingLot {
    List<CompactSpot> compactSpotList = new ArrayList<>();
    List<LargeSpot> largeSpotsList = new ArrayList<>();
    List<BikeSpot> bikSpotsList = new ArrayList<>();

    int freeCompactSpots;
    int freeLargeSpots;
    int freeBikeSpots;

    public ParkingLot(int freeCompactSpots, int freeLargeSpots, int freeBikeSpots){
        this.freeCompactSpots = freeCompactSpots;
        this.freeLargeSpots = freeLargeSpots;
        this.freeBikeSpots = freeBikeSpots;
    }

    //parkvehicle

    public void parkVehicle(Vehicle vehicle) {


    }


    //UnparkVehicle

}

```

```java
//parkvehicle

public void parkVehicle(Vehicle vehicle) {
    System.out.println("We are about to park your Vehicle of type " + vehicle.getVehicleType());


    if(vehicle.getVehicleType().equals(VehicleType.TRUCK)) {
        if(freeLargeSpots > 0)
            parkYourLargeVehicle();
        else
            System.out.println("Sorry all Large spots are Full");

    }else if(vehicle.getVehicleType().equals(VehicleType.CAR)) {
        if(freeCompactSpots > 0)
            parkYourCompactVehicle();
        else if(freeLargeSpots > 0)
            parkYourLargeVehicle();
        else
            System.out.println("Sorry neither Large Or Compact spots Available");

    }else if(vehicle.getVehicleType().equals(VehicleType.BIKE)) {
        if(freeBikeSpots > 0)
            parkYourBikeVehicle();
        else if(freeCompactSpots > 0)
            parkYourCompactVehicle();
        else if(freeLargeSpots > 0)
            parkYourLargeVehicle();
        else
            System.out.println("Sorry neither Large Or Compact or Bike spots Available");
    }


}
```

```java
void parkYourLargeVehicle(Vehicle vehicle){
    LargeSpot largeSpot = new  LargeSpot(ParkingSpotType.LARGE);
    largeSpot.setFree(false);
    largeSpot.setVehicle(vehicle);
    vehicle.setParkingSpot(largeSpot);

    largeSpotsList.add(largeSpot);
    System.out.println("We have successfully parked your " + vehicle.getVehicleType() + " Vehicle.");
    freeLargeSpots--;

}

void parkYourCompactVehicle(Vehicle vehicle){
    CompactSpot compactSpot = new  CompactSpot(ParkingSpotType.COMPACT);
    compactSpot.setFree(false);
    compactSpot.setVehicle(vehicle);
    vehicle.setParkingSpot(compactSpot);

    compactSpotList.add(compactSpot);
    System.out.println("We have successfully parked your " + vehicle.getVehicleType() + " Vehicle.");
    freeCompactSpots--;

}

void parkYourBikeVehicle(Vehicle vehicle){
    LargeSpot largeSpot = new  LargeSpot(ParkingSpotType.LARGE);
    largeSpot.setFree(false);
    largeSpot.setVehicle(vehicle);
    vehicle.setParkingSpot(largeSpot);

    largeSpotsList.add(largeSpot);
    System.out.println("We have successfully parked your " + vehicle.getVehicleType() + " Vehicle.");
    freeLargeSpots--;

}
```

```java
// UnparkVehicle

public void unParkVehicle(Vehicle vehicle) {

    System.out.println("We are about to  Unpark or remove your Vehicle of type " + vehicle.getVehicleType());

    ParkingSpot  parkingSpot = vehicle.getParkingSpot();
    parkingSpot.setFree(true);


    if (vehicle.getVehicleType().equals(VehicleType.TRUCK)) {
        if(largeSpotsList.remove(vehicle)) {
            System.out.println("We have successfully removed your Vehicle");
            freeLargeSpots++;
        }
        else
            System.out.println("You dont have your vehicle parked at this parking lot");

    }else if (vehicle.getVehicleType().equals(VehicleType.CAR)) {
        if(compactSpotList.remove(vehicle)) {
            System.out.println("We have successfully removed your Vehicle");
            freeCompactSpots;
        }
        else
            System.out.println("You dont have your vehicle parked at this parking lot");

    } else if (vehicle.getVehicleType().equals(VehicleType.TRUCK)) {
        if(largeSpotsList.remove(vehicle)) {
            System.out.println("We have successfully removed your Vehicle");
        }
        else
            System.out.println("You dont have your vehicle parked at this parking lot");
```

Writable | Smart Insert | 108 : 34 : 3438

ParkingLot.java | ParkingSpot.... | Vehicle.java | CompactSpot.... | LargeSpot.java | BikeSpot.java

```java

public class MainClient {
    public static void main(String[] args) {

        ParkingLot parkingLot = new ParkingLot(2, 2, 2);

        Vehicle truck = new Vehicle(VehicleType.TRUCK);

        parkingLot.parkVehicle(truck);

        parkingLot.unParkVehicle(truck);
    }

}
```