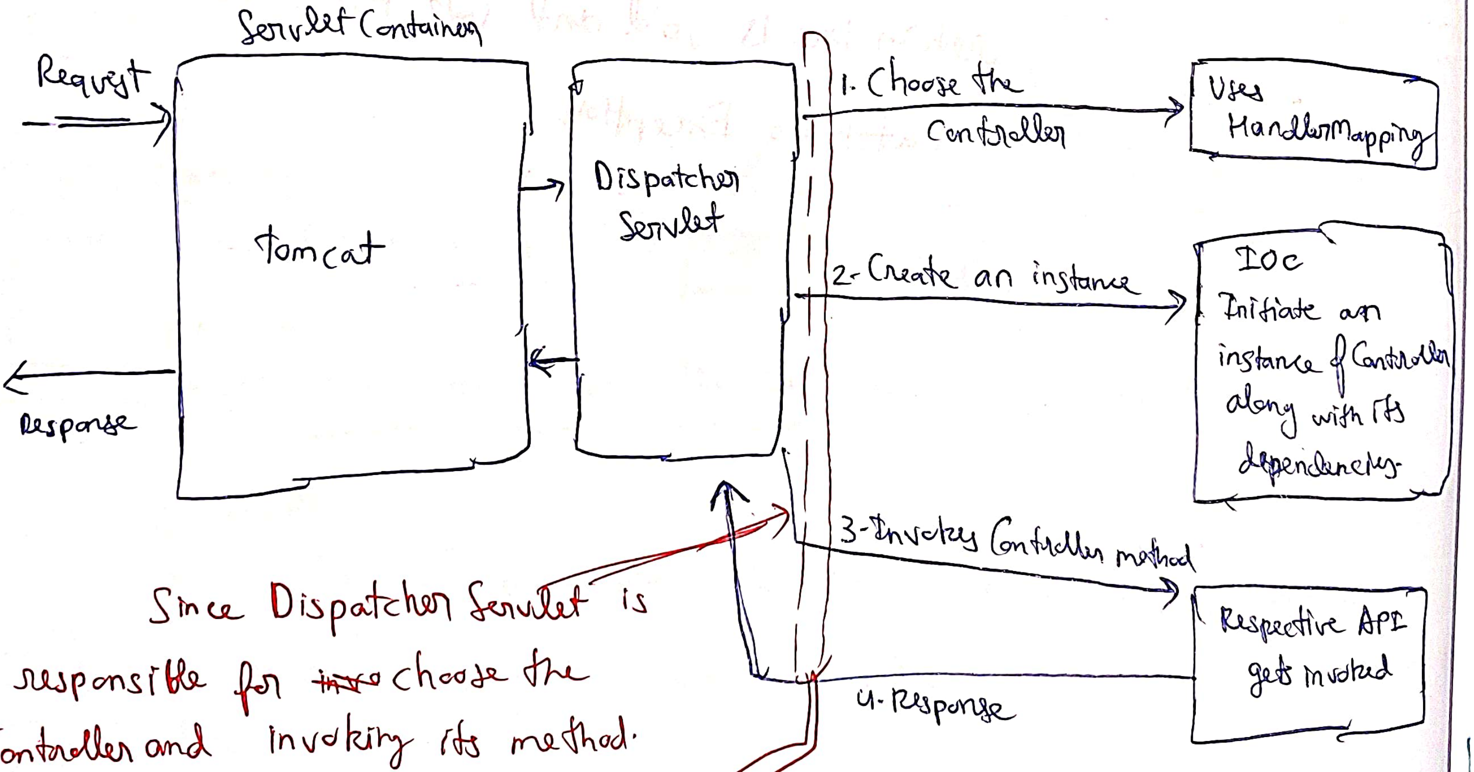
Video 17. Spring Boot : Custom Interceptors | How to Intercept Incoming HTTP Request and Custom Annotations.

Interceptor: Its a mediator, which get invoked before or after actual code.

In future topics we would need to write our custom interceptors.

- Springboot Caching
- SpringBoot logging
- SpringBoot Authentication etc.

Lets Recall the Flow diagram of SpringBoot



Since Dispatcher Servlet is responsible for ~~into~~ choose the Controller and invoking its method.

So somewhere in b/w we have to introduce our Custom Interceptor.

Example)

```
@RestController
@RequestMapping("/api")
public class UserController{

    @Autowired
    User user;

    @GetMapping("/getUser")
    public String getUser(){

        user.getUser();
        return "sucess";
    }

}
```

These 3 methods have ~~to~~ provide implementation as present in HandlerInterceptor

```
@Component
public class MyCustomInterceptor implements
                HandlerInterceptor{
                        → Parent
    @Override
    public boolean prehandle(------){
        sout("inside prehandle")
        return true;
    }

    @override
    public boolean posthandle(------){
        sout("inside posthandle")
    }

    @override
    public void afterCompletion(------){
        sout("inside aftercompletion")
    }
}
```

@Component hence directly injected by Spring

```
@Configuration
public class AppConfig implements WebMvcConfigurer{
                                    → (interface)
    @Autowired
    MyCustomInterceptor myCustomInterceptor;
                                    got implemented here
    @Override
    public void addInterceptors(InterceptorRegistry registry){

        registry.addInterceptor(myCustomInterceptor)
            .addPathPatterns("/api/*") => All api having /api/---
            .excludePathPatterns("/api/updateUser", "/api/deleteUser");
    }
};
```

Specify paths to avoid interception

If you explore the code more.

Inside Dispatcher Servlet.class.

there is a method

```
Dispatch Servlet # doDispatch(. --- ){

try {
    if (! mappedHandler.applyPreHandle( ---- )){      ⇒ preHandle( )
        return.
    }
                                [actual invocation of the ~~Handler~~
                                 controller and its method]
    mv = ha.handle( ---- );

    mappedHandler.applyPostHandle(      );
}                                          ⇒ PostHandle
catch (Exception ex){
    ----

    triggerAfterCompletion(---);       Gets Execution even if
}                                      Exception occurs.
                                       Similar to finally block
catch (Throwable th){                  in Java.
    ---

    triggerAfterCompletion(---); ⇒ AfterCompletion.
}

}.
```

Creating Custom Interceptor for Request after Reaching to a Specific class.

For Example

```
@GetMapping(      )                          @
public String getResult ( ){                 public class User{

    user.getAnswer ( );  - - - - - - - ->      public String
                                                 getAnswer ( ){
    return null;
}                                              }
                                             }
```

We want to invoke interceptor when the getAnswer ( ) method in User class gets invoked.

Step1 : Creation of custom annotation

We can create Custom Annotation using keyword @interface java annotation.

```
public @interface MyCustomAnnotation {
}


  public class User {

      @MyCustomAnnotation
      public void updateUser ( ){

      }
  }
```

2 Important Meta Annotation Properties are required :-

Meta Annotations :- Annotations which are applied over an annotation.

1) @Target :- this meta annotation tells, where we can apply the particular annotation on method or class or constructor

```
@Target(ElementType.METHOD)
public @interface MyCustomAnnotation{
}
```

if I want to use it over more than 1 place

```
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD,
         ElementType.PARAMETER, ElementType.FIELD})
public @interface MyCustomAnnotation {

}
```

2) @Retention: This meta annotation tells, how the particular annotation will be stored in Java

(P.T.O)

Rentention Policy.SOURCE

Annotation will be discarded by compiler itself and is not even recorded in .class file [After Javac compilation].

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.SOURCE)
public @interface MyCustomAnnotation{

}
```

```
public class User{

    @MyCustomAnnotation
    public void updateUser(){
        //.— - -
    }
}
```

After Javac Compilation.

User.class

```
package - - -

public class User {

    public User(){
    }

    public void updateUser(){
    }
}
```

→ Annotation not applied.

RetentionPolicy. CLASS

Annotation will be recorded in .class file but during runtime will be ignored by JVM.

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.Class)
public @interface MyCustomAnnotation{

}
```

```
public class User {

    @MyCustomAnnotation
    public void doSomething(){
    }

}
```

After Javac compilation.

User.class

```
package ---

@MyCustomAnnotation
public void doSomething(){

}
```

But it will be ignored during Runtime.

~~Rea~~ [RetentionPolicy. RUNTIME]

Annotation will be recorded in .class file and also available during runtime.

How to create Custom Annotation with methods (which are more like fields)

* No parameter, no body
* Return type is restricted to [Annotations are very light weight, because we don't want to extend its use cases.]
    * Primitive type (int, boolean, double etc)
    * String
    * Enum
    * Class <?> ⇒ should be .class, Like String.Class
    * Annotations
    * Array of Above types

```
@Target (ElementType.METHOD)
@Retention (RetentionPolicy.RUNTIME)
public  @interface  MyCustomAnnotation {

    String key() default "defaultKeyName";

}
```

```
public class User{
    @MyCustomAnnotation(key = "userKey")
    public void updateUser(){
        // some logic.
    }
```

Now Value key in Annotation is "userKey"

Lets say if @MyCustomAnnotation had multiple fields, then ~~[illegible]~~ we can define in the following way

```
public class User{

    @MyCustomAnnotation(int Key = 10, String Key = "user", classTypeKey = User.class,
    public void updateUser(){                     enumKey = MyCustomEnum.ENUM_VAL)
    }
}
```

```
@RestController
@RequestMapping("/api")
public class UserController{

    @Autowired
    User user;

    @GetMapping("/getUser")
    public String getUser(){
        user.getUser();
        return "sucess";
    }
}
```

```
@Target(ElementType.METHOD)
@Retention(RententionPolicy.RUNTIME)
public @interface MyCustomAnnotation{

    String name() default "";

}
```

```
@Component
public class User{

    @MyCustomAnnotation(name = "User")
    public void getUser(){
        sout( );
    }
}
```

```
@Component
@Aspect  ------> Aspect Oriented Programming
public class MyCustomInterceptor{
                    Location of the Annotation
    @Around("@annotation(com.conceptandCoding.CustomInterceptor.MyCustomAnnotation)")
    public void invoke(ProceedingJoinPoint joinPoint){

        Using this we got the method getUser() itself
        [Method method = ((MethodSignature) joinPoint.getSignature()).getMethod();]

        if(method.isAnnotationPresent(MyCustomAnnotation.class)){

(Fetching          MyCustomAnnotation annotation = method.getAnnotation(MyCustomAnnotation.
the annotation)                                                          class);

            sout(annotation.name());
                 ---> [Using its parameter]
        }
        joinPoint.proceed();
        sout("@around after method call");
}
```