Spring Start Here

Chapter 1

Spring in the real world

1.0 what is framework ?

frame work is set of software functionalities - capabilities that helps us to build an application

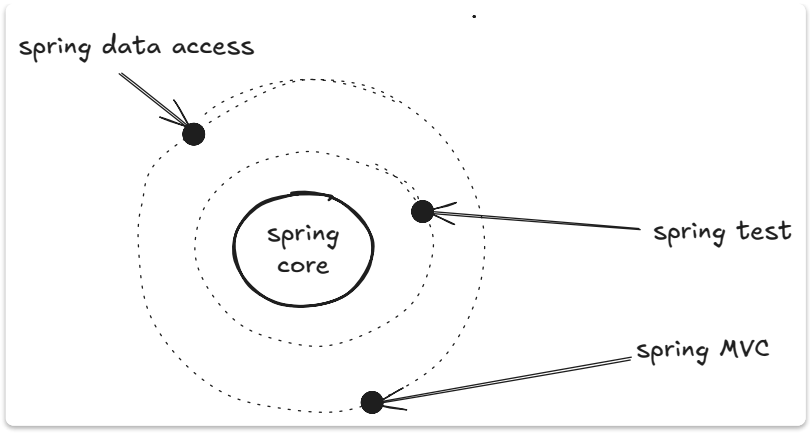
1.1 Why should we use frameworks :

- all Spring capabilities rely on two components (Spring context - Spring aspects)
- it us provide the needed components to build our app instead of typing the code from scratch it is provide convention over configuration .
- it also provide different tools that we could choose from them what we want to build our app and assemble these tools together .
- depending on your requirements you'll choose the right parts from the framework . '4/1'
- it is save a lot of time and effort by providing a well tested components .
- for reusing of code cause a lot of apps similar in some common requirements and only differs in business logic : '5/2'
  - logging error , warnings.
  - connecting to database .
  - protection.
  - communicate with other applications.
- business logic : '6/3'
  it is the implemented code that give the business requirements and user expectations .

1.2 The Spring ecosystem :

it is not just a framework it is a set of components that help you creating till deploying your application to web in spring ecosystem the main components are '8/1'

- spring core : '8/1.1'
  it is rely on two main components ,
  1. Spring context : enable spring to manage instances of the app .
  2. Spring aspects : helps spring to intercept and manipulate methods defined in app .
- spring MVC (model - view - controller) : '8/1.2'
  - enables you to develop web apps that serve http requests .
- spring data access : '8/1.3'
  helps us for data persistence cause some apps needs to connect to the data base and persist the data on the hard disk .
- spring test : '8/1.4'
  the part where we write test to validate app behaviour .
- Discovering spring core :
  spring works with the principle IOC 'inversion of control'
  - IOC 'spring context' : '8/2'
    - instead of letting the app controls the execution(dependencies), we let spring(dependency) to manage the app .
    - we instruct the framework to manage the code we write .
    - glue spring components and app components together to framework .
  - AOP 'aspect oriented programming ' : '9/2'
    - intercept methods that represent the behaviour of the instances 'Aspecting'.
    - let framework interacts with what the app does .
- 1.2.2 Spring Data Access :
  it is the way we use to persist the data .
- 1.2.3 Spring MVC
  - Model View Controller
  - we use it to develop web apps that serve http requests using standard servlet fashion .
- 1.2.4 Spring Testing
  it is where we type test cases 'unit, integration' to see how our app behave and see if it validate the expected requirements
- Spring Ecosystem has independent projects that has team working on it like :
  - Spring Data : '11/1'
  it helps us to easily ,
  1. easily connect to database .
  2. use persistence layer with minimum lines of code .
  3. deals with sql and NoSql .
  - Spring Core : '11/2,3'
  spring project that help us to create backend web apps in the form of :
  - conversion over configuration : it provides a foundation with default configuration and u just change what you want, this help us to write less code instead of write from scratch .



1.3 Spring in real world :

we can use spring in different technologies like :

1. Backend App :
   spring offers us a large number of tools that we can use to build our backend app and make it integrate with other apps or to persist the data
2. automation testing framework :
3. desktop app
4. mobile app

1.4 When not to use frameworks :

- when we do not feel a big impact from changing to framework cause if we do such a thing we could force a only errors without any benefits .
- you may be implementing a small app that could be easier to implement from scratch rather than using a framework .

1. need to have small foot print :
   - imagine that you are building a service that doesn't need to be much big and it has to be very small, in this scenario using frame work is a loss cause it requires a bigger space on the server and cause of containers that closed and opened more often you need your app to be small that could save a lot of seconds each time the container opened .
2. security :
   in some organizations or companies they do not prefer using an open source frameworks because hackers may find a vulnerable and find a way to access their data .
3. customizing components too much :
   in this case you may find that customizing the components and the configuration of the app is much bigger than implementing the app your self .
4. you don't find any benefit from switching to framework

1.5 What you will learn in this book :

Chapter 2

Spring Context :

- what is Spring context (application context) ?
  - it is a place in the memory of the application where we put the beans so that spring can see them and integrate with. '22/1'
- why context ?
  - to put the instances of our application in it because spring see only the instances that are inside the context .
  - spring use beans to connect them with different capabilities .
- how to make context instance ?
  - var context = new AnnotaionConfigApplicationContext(Configuration.class);

adding beans to context :

- there are different ways to put beans inside the context and we use the way that meets our requirements, at first we had to add spring context dependency in the pom.xml .
  1. @Bean Annotation
     1. @Configuration
        - we make class for the configurations that spring loads it while initializing the context and we put the configuration of the beans inside it
     2. @Bean
        - we declare method that return an instance of the object we want to put inside the context and the name of the method is a noun not verb .
     3. we add (pass) the configuration class into the context while initializing it
  - we can add different names for methods so that when we getting bean from context we can refer to them
    - Parrot p1 = context.getBean(Parrot.class) -> will get the instance if there is only one if there are more than one we must specify the bean is name -> p = context.getBean("chicken" , parrot.class)
    - we can name the beans with these attributes (name = " " , value = " " , " ")
    - @Primary : we put this annotation to point this bean as the default one when creating more than one bean

```java
@Configuration
public class Config {
    no usages
    @Bean
    Parrot parrot(){
        Parrot p = new Parrot();
        p.setName("koko");
        return p;
    }

}
```

```java
no usages
public static void main(String[] args) {
    var context = new AnnotationConfigApplicationContext(Config.class);
    Parrot PFromContext = context.getBean(Parrot.class);
    System.out.println(PFromContext.getName());
}
```

```java
@Bean(value = "first")
Parrot parrot(){
    Parrot p = new Parrot();
    p.setName("koko");
    return p;
}
no usages
@Bean(name = "second")
Parrot parrot2(){
    Parrot p = new Parrot();
    p.setName("lolo");
    return p;
}
no usages
@Bean("third")
@Primary
Parrot parrot3(){
    Parrot p = new Parrot();
    p.setName("kike");
    return p;
}
```

2. Stereo type annotations :
    1. @Component :
        - put this annotation in top of the class you want to make bean from
    2. @ComponentScan (basePackages = "package")
        - add this annotation in top of the configuration class to tell spring that there are components to search for .
    - Spring will create an instance and put it inside the context .
    - Notes :
        1. it provide more less code than @Bean annotation
        2. spring manage the instance and we cant modify it like we did with @Bean
        3. we can use this with the classes inside our application only and can't use it with external libraries .
        4. if we want to make spring do something after creating the bean or before destroying it we have two annotations from jakarta dependency they are :
            1. @PostConstruct .
            2. @PreDestroy .
3. programmability
    - sometimes we could have some conditions or business logic depending on this we wan to choose which of the instance we are going to put inside the context and we can't do this with the previous ways but with spring 5 we now can do it using :
        - .registerBean() method :
          it consist of 4 parameters
            1. the name that will be assigned to the bean
            2. the class of this instance
            3. supplier that supply the instance to put it into context

StereoType	@Bean
less code	had to create method that returns the instance
only with project classes	with any type of classes
had to provide basepackage	in configuration file
create only one instance	can create more than one instance
have no access to instance untill creation	can modify instance before putting it in context

---

Chapter 3

- in this chapter we talk about how to make relations between beans and connect them with each other cause in sometimes we need use some capabilities from one class inside another, we had to tell spring to get a ref of a bean from its context to use it with another bean and there are two ways to do such thing :
  1. wiring : by calling the method that create the instance of the bean .
  2. auto wire : by defining the method parameter with the same name as the bean is class
     - @autowired annotation also
- to make relation between beans we done it in two steps .
  1. creating bean in context .
  2. make relation with another bean (has-A) .

1. Wiring beans by calling method that create that instance :
   - in configuration file we call the method from another and the method will get a reference for the bean that is inside it is context .
     - spring checks if the bean exists spring will return a ref to it
     - else spring will create the bean , put it into context and return a ref after creation .
2. using auto wire with @bean method parameters :
   - in this we we name the identifier of the method parameter with the same name of the bean (method that creates the bean) .
   - so in this way we instruct spring to search inside its context for that bean and inject it inside parameter '59/1' .
   - instruct spring to provide instance from its context has the same type as method parameter '58/1' .
   - this method called Dependency Injection DI -> application of IOC -> spring injects the dependency ?
     - loose coupling .
     - less code .
3. @AutoWired Annotation :
   1. with class field :
      - we add the annotation above the class field and in this way we instruct spring to inject a value to this field from its context
      - used with examples , profs
      - not good because it doesn't allow is to make fields final .
   2. with class constructor :
      - the most used one with real world apps .
      - it allow us to make fields final so no one can change them .
      - in this way we instruct spring to inject value from its context to the desired bean is constructor parameter so when initializing that bean spring will provide the instance from its context '64/1'
   3. with setters :
      - we define setter with @autowired annotation .
      - more code, can't define fields as final .

Circular dependency :

- this happen when we define two beans that depends on each other while creation, so spring will be in a deadlock .
- we just need to avoid this method with our own cause no such a specific solution '66/1' .

choosing from multiple beans :

1. making the identifier of the parameter has the same name of the bean (method that returns that bean) .
2. what if the identifier doesn't have the same name of any bean
   1. @primary annotation , by making one of the beans inside the context as primary so if we didn't specify the correct name. spring will inject the primary one .
   2. @Qualifier and providing the bean name inside the value attribute @Qualifier("Parrot2") -> same name as bean (method that returns that instance).

Chapter 4

- using interfaces to define contracts
  - interfaces used to define contracts between implementations.
  - interfaces used to define responsibilities, object that implement this interface had to define this responsibility .
  - interfaces is what we need to happen .
  - objects is how it should happens . '76/1'

  1. using interfaces for decoupling implementations :
     - decoupling implementations is helpful cause with this we can type less code, can extend and maintain the app easily .
     - instead of defining what we want and how it happen, we should only define what we want and don't put in mind how it happens .
     - if we depend on how (implementation), further if we wanted to change something we will have to rewrite every thing .
  2. the requirement of the scenario :

3. implementing requirement without using framework :
    - object that deal with database -> repository, DAO (Data access object) . '80/2'
    - object that deal with things outside project -> proxy . '80/3'
    - object that implements use case, business logic -> service . '80/1'
- using dependency injection with abstraction
    1. deciding which objects should be part of spring context :
        - objects that spring had to see them to use them with its functionalities .
        - at first, we ask should this object be managed by spring ? '86/1'
        - we don't put every object cause if it is in the context and spring don't use it , it would lead to less performance , over engineering '86/2,3' .
        - it doesn't make sense to put annotation on top of I or abstract class that can't be instantiated '86/4' .
        - if we declared an attribute of an interface, spring is smart enough to find the instance, bean that provide the implementation of this interface and inject it there '88/0' .
        - @AutoWired optional with class that have one constructor .
        - with DI we don't need to implement, declare new instance of a class to use it, we only get it from context '89/1' .
        - DI, focus on what you want, not how it done, less code '89/2' .
    2. choosing what to wire from multiple implementations of an abstraction :
        - we need to tell spring which bean to inject '91/1'
        - @Priamary Annotation : with this we we instruct spring to inject default value from it is context if it didn't specified before .
        - using @Qaulifier annotation :
            - to define bean name , first
            - to instruct spring to inject this specified bean from it is context .
        - why to have more than one implementation ?
            - cause some times the provided implementation doesn't match our needs so had to provide our custom implementation .
            - more objects each object may wants another type of fruit .
- focusing on object responsibilities with stereotype annotations
    - @service -> define component , declare it is responsibility -> implement use case .
    - @repository -> define component, mark its responsibility -> manage data persisting '96/3' .
    - @component -> generic annotation doesn't specify object is responsibility .

Chapter 5

"In Singleton scope, immutability ensures thread safety, consistency, and avoids shared state bugs."

- in this chapter we talk about how spring manage the creation of the bean and maintain it is lifecycle inside the context . '100/1'

1. Singleton bean scope

- it is bean is default scope and the most used one '100/3'
- most used in production apps .

1. how singleton beans work :
    - the bean is created while loading the the context, start up of app and given a name for a bean (bean id) '100/4' .
    - you always get the same instance when you refer to specific bean by it is name '100/5' .
    - you can have multiple beans of same type each with a specific name, but each bean have only one instance '100/6' .
    - in spring -> singleton means -> same instance for unique name .
2. singleton beans in real world scenarios :
    - where you want to create instance shared between multiple objects .
    - must be immutable -> final -> DI constructor '107/1' .
    - mutable singleton will lead to race condition .
    - using beans gives three points :
        1. when you want spring to manage it and augment it with it is features and capabilities .
        2. when you want immutable instance from the context -> singleton
        3. if you want mutable bean, you can mace it prototyped scoped .
3. using eager and lazy instantiation :
    1. Eager :
        - spring is default approach when creating a singleton bean .
        - the bean is created while instantiating the context .
        - most used in real world apps .
        - best performance than lazy approach because it load all beans at first no need to search for beans further, but more memory .
        - the errors appears while starting up the app so it help us in test cases .
        - if object delegates to another it is better that the other bean is already exists in the context instead of waiting for it '100/1' .
    2. Lazy :
        - had to specify with @Lazy annotation
        - the instance is created when you refer to it for first time '108/3' .
        - if you refer to the bean for first time spring looks for it in context, if doesn't exist spring creates it and returns instance '110/2,3' .
        - if there is an error it will appear while runtime .
        - less performance because of the search of bean, saves memory cause you create bean only when needed .

2. prototype bean scope

1. How prototype bean works ? :
    - every time you refer to the bean, spring creates new instance from that bean '111/1' .
    - spring manage object's type, every time you refer to it, spring creates new instance 111/2 .

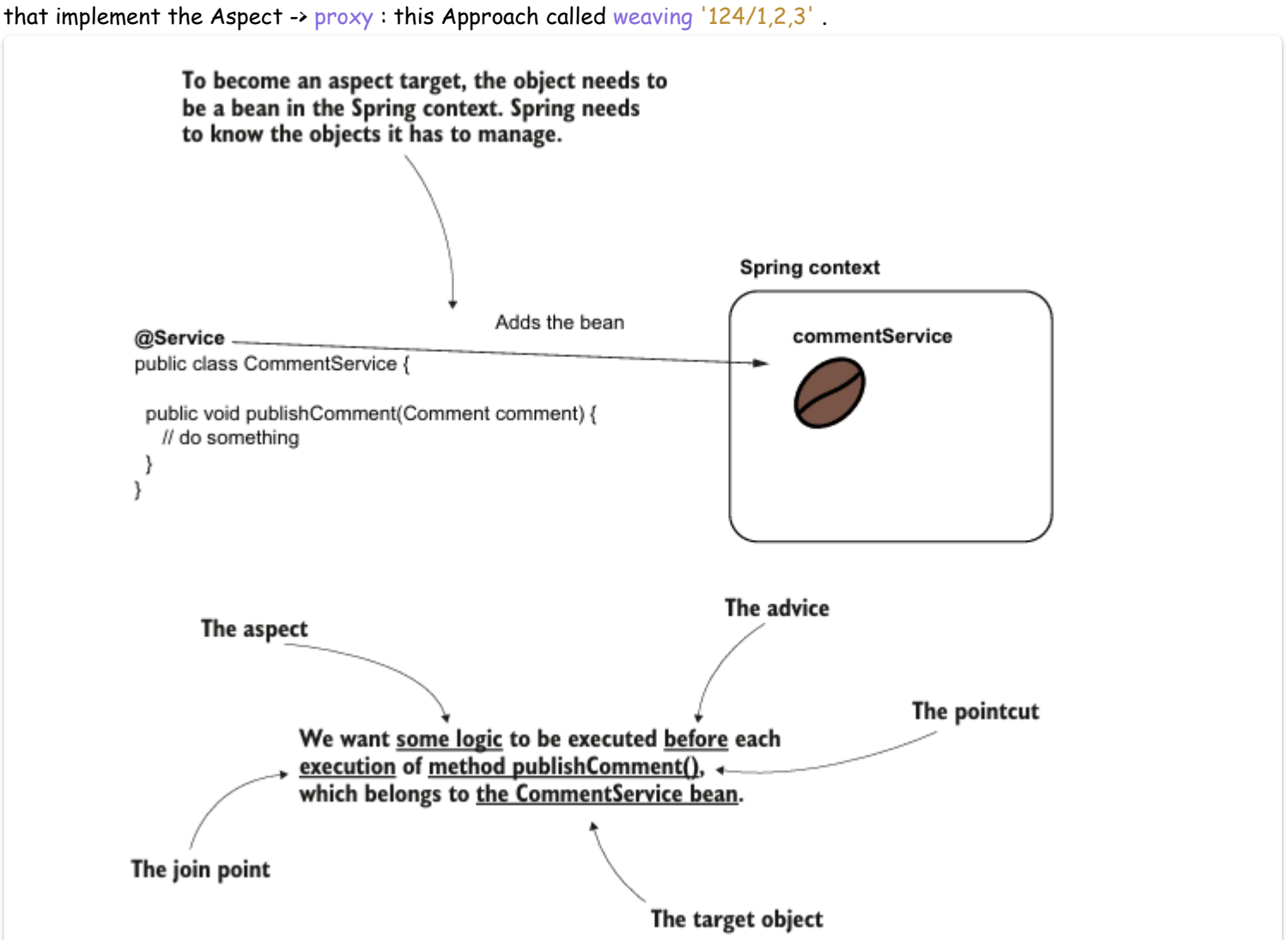
- @scope(BeanDefination.SCOPE_PROTOTYPE) .
- each thread requests the bean gets new instance and this solved race condition .
  - race condition :
    when two threads access one shared instance at the same time, and the threads tries to change it value, the last edit is the preserved one .
- prototype is mutable '111/3' .

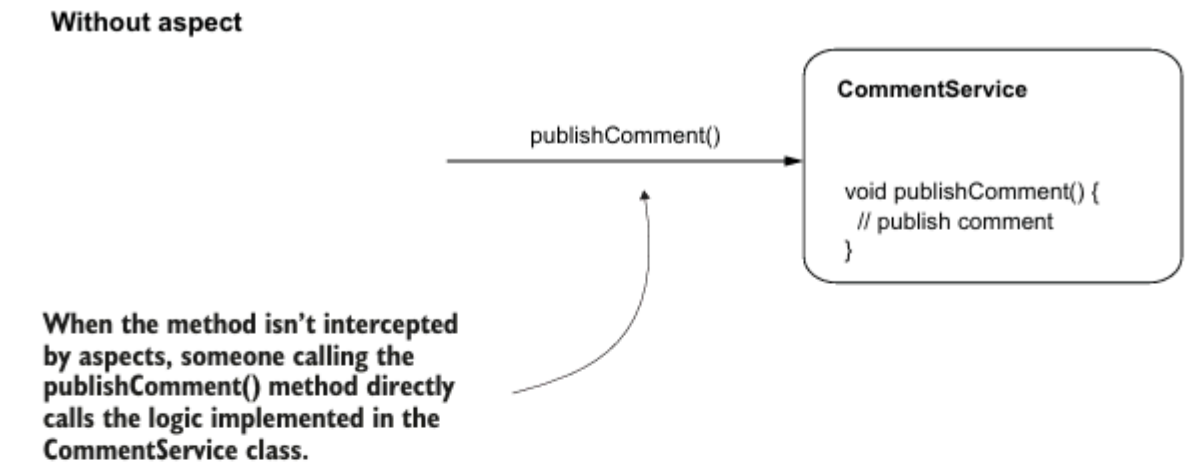
2. proto type bean in real world scenarios :
   - singleton is the most used one, prototype is rarely used .
   - if you want mutable instance, prototype is the solution for this case .
   - we want each call of method gets new instance '117/2' .
   - we would use prototype when refactoring or working with old applications .

Chapter 6

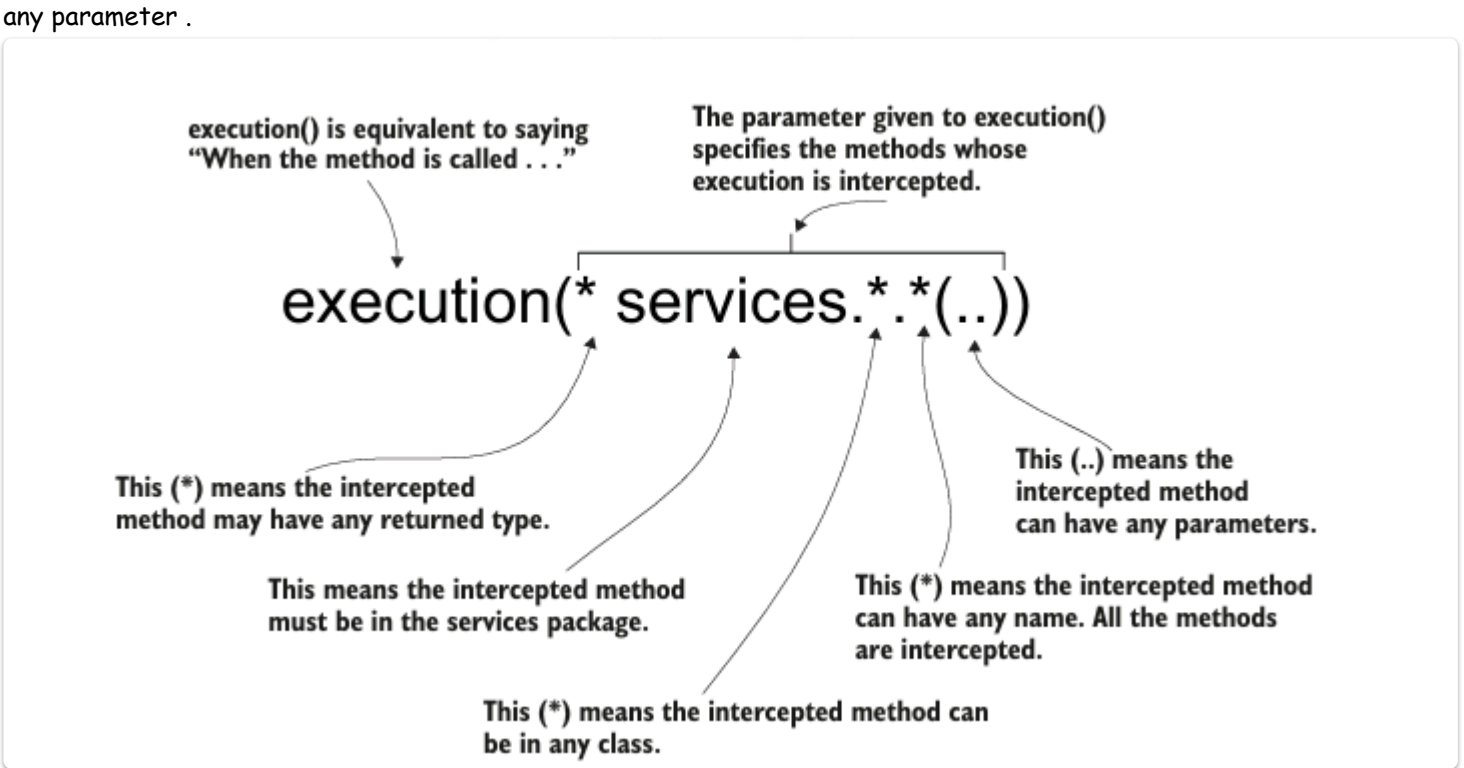
- in this Aspect Oriented Programming in spring (Spring Aspects)

1. What is Aspect in spring :
   - another technique of IOC .
   - the way spring intercept method call and can alter it is execution '122/1' .
   - a part of code you want spring to execute when calling specific method 123/5 .
2. Why to use Aspects :
   1. allow decoupling between method parts, this makes method easy to understand .
   2. let us focus on method logic .
   3. makes code more maintainable .
   4. spring uses aspects to implement its capabilities .
3. Put in mind
   - what part of code you want spring to execute when calling specific method -> Aspect .
   - when spring needs to execute logic of aspect (during method execution, after, or instead of method) -> Advice .
   - which methods you want spring to intercept and use Aspect for them -> Point cut .
   - event trigger -> method call -> join point '123/6' .
   - Bean declares methods that Spring will intercept -> target object '123/8' .
   - if you requested a reference from target bean spring doesn't give you reference for this bean, instead it gives you reference of the object that implement the Aspect -> proxy : this Approach called weaving '124/1,2,3' .





4. implementing Aspects with Spring AOP
   - to create Aspect we have to follow these 4 steps :
     1. Enabling Aspect by adding @EnableAspectJAutoProxy to the configuration class 129/1 .
     2. create new class Annotated with @Aspect and create a bean from it inside spring context 129/2 .
     3. define method that will implement aspect logic with advice Annotation and when to execute, which method to intercept '130'.
     4. implement Aspect logic code .
   - using @Aspect annotation doesn't create a bean we just telling spring that this class implementing logic of aspect, you had to explicitly create the bean using @Bean or stereo type annotation '131/1' .
   - we use advice annotation to tell spring when and which method to intercept by @Around("execute(*.services.*.*(..))") '132/1' .
     - @Around -> tells spring which method call to intercept .
     - first * -> any return type .
     - package name .
     - any class .
     - any method .
     - any parameter .




```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-context</artifactId>
   <version>5.2.8.RELEASE</version>
</dependency>
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-aspects</artifactId>
   <version>5.2.8.RELEASE</version>
</dependency>
```

We need this dependency to implement the aspects.

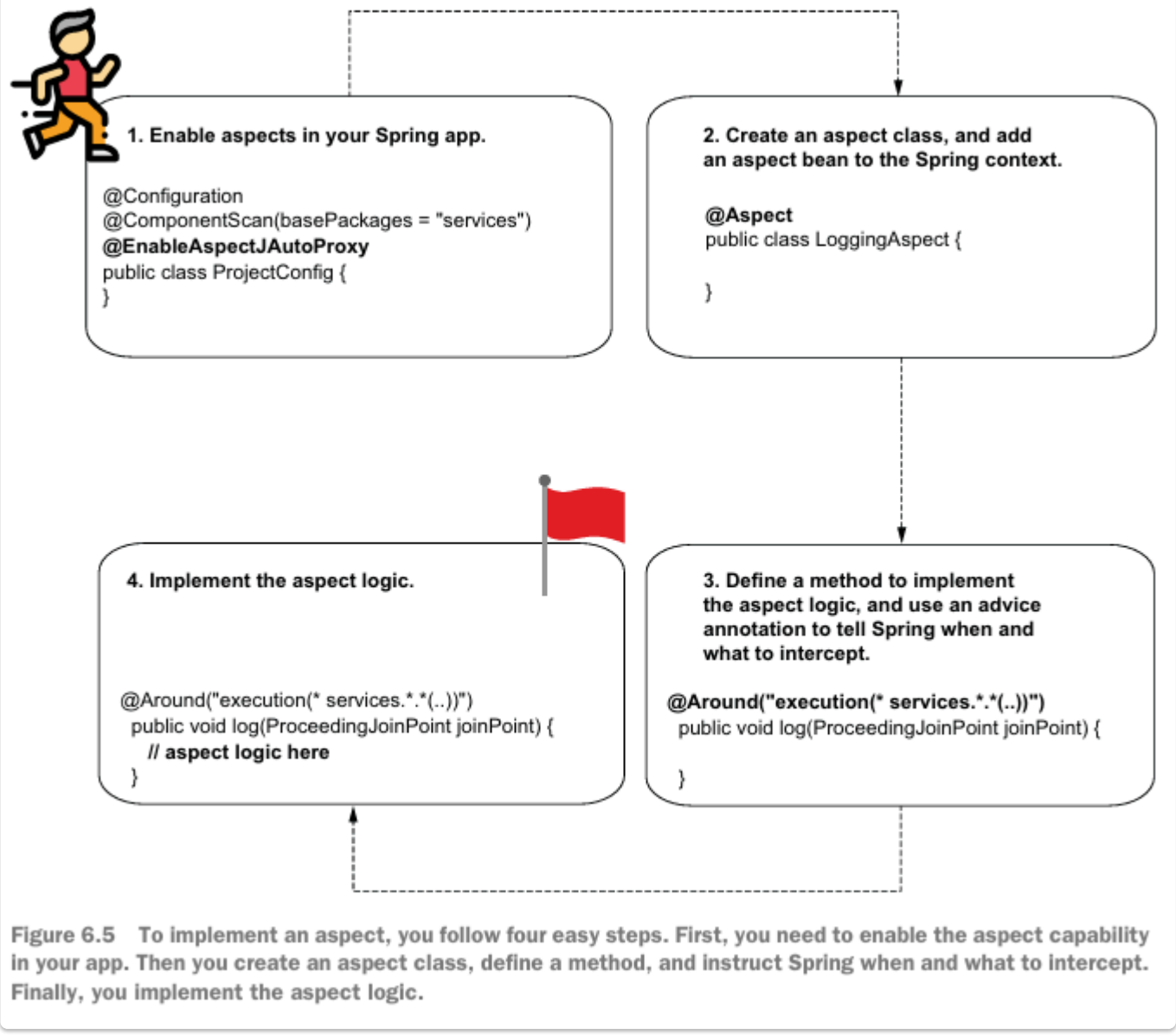


Figure 6.5 To implement an aspect, you follow four easy steps. First, you need to enable the aspect capability in your app. Then you create an aspect class, define a method, and instruct Spring when and what to intercept. Finally, you implement the aspect logic.

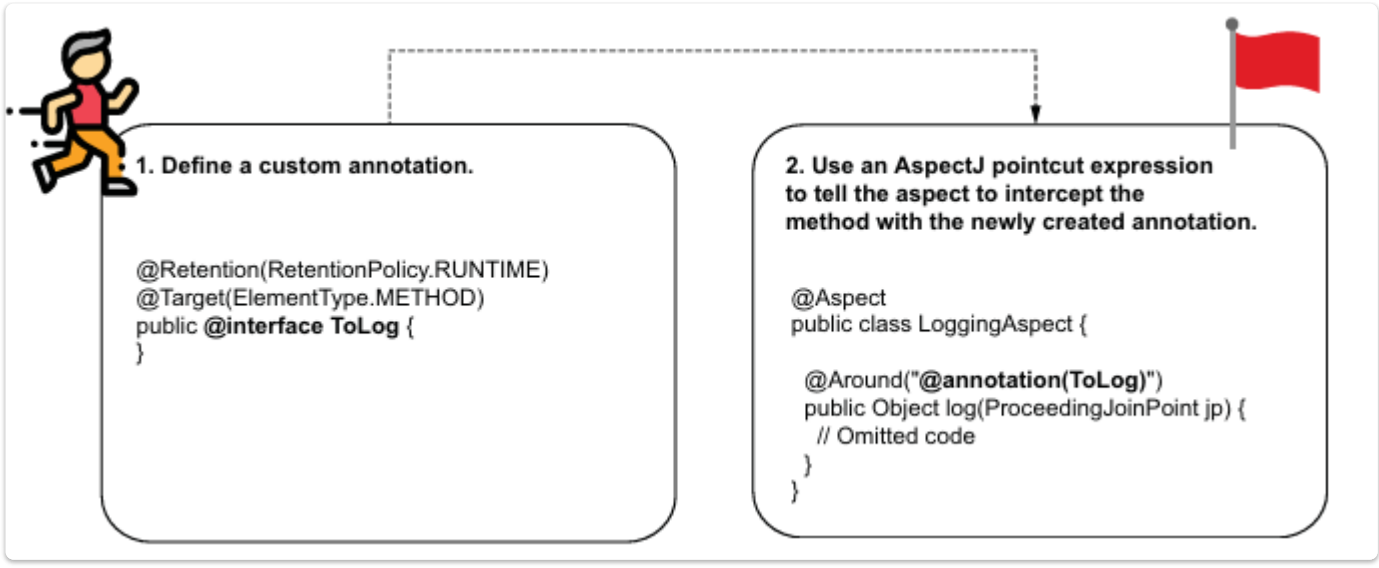
5. Altering intercepted method parameters
   - in aspects we can get the parameters used to call intercepted method and change their values
   - we can also get values returned from intercepted method and alter them '135/2' .
   - aspect can throw exception to caller and handle exception of intercepted method '138' .
   - ProceedJoinPoint :
     - it is a parameter used to represent intercepted method, we can use it to get (method name, parameters, target object) '136/1' .
     - to get parameters: Object [ ] values = joinPoint.getArgs(); .
     - method name : String name = joinPoint.getSignature().getName(); .
     - returned value : Object returned = joinPoint.proceed(); .
     - we can provide parameters to intercepted method using proceed(Object[ ] obj) method of pointJoint '139/2'.
   - we can change intercepted method behaviour, but the idea of decoupling part of code is to avoid duplicating the irrelevant code and focus on business logic '139/1' .

```
@Aspect
public class LoggingAspect {

  private Logger logger =
  Logger.getLogger(LoggingAspect.class.getName());

  @Around("execution(* services.*.*(..))")
  public Object log(ProceedingJoinPoint joinPoint) throws Throwable {
    String methodName = joinPoint.getSignature().getName();
    Object [] arguments = joinPoint.getArgs();


    logger.info("Method " + methodName +
        " with parameters " + Arrays.asList(arguments) +
        " will execute");

    Comment comment = new Comment();
    comment.setText("Some other text!");
    Object [] newArguments = {comment};

    Object returnedByMethod = joinPoint.proceed(newArguments);

    logger.info("Method executed and returned " + returnedByMethod);

    return "FAILED";
  }
}
```

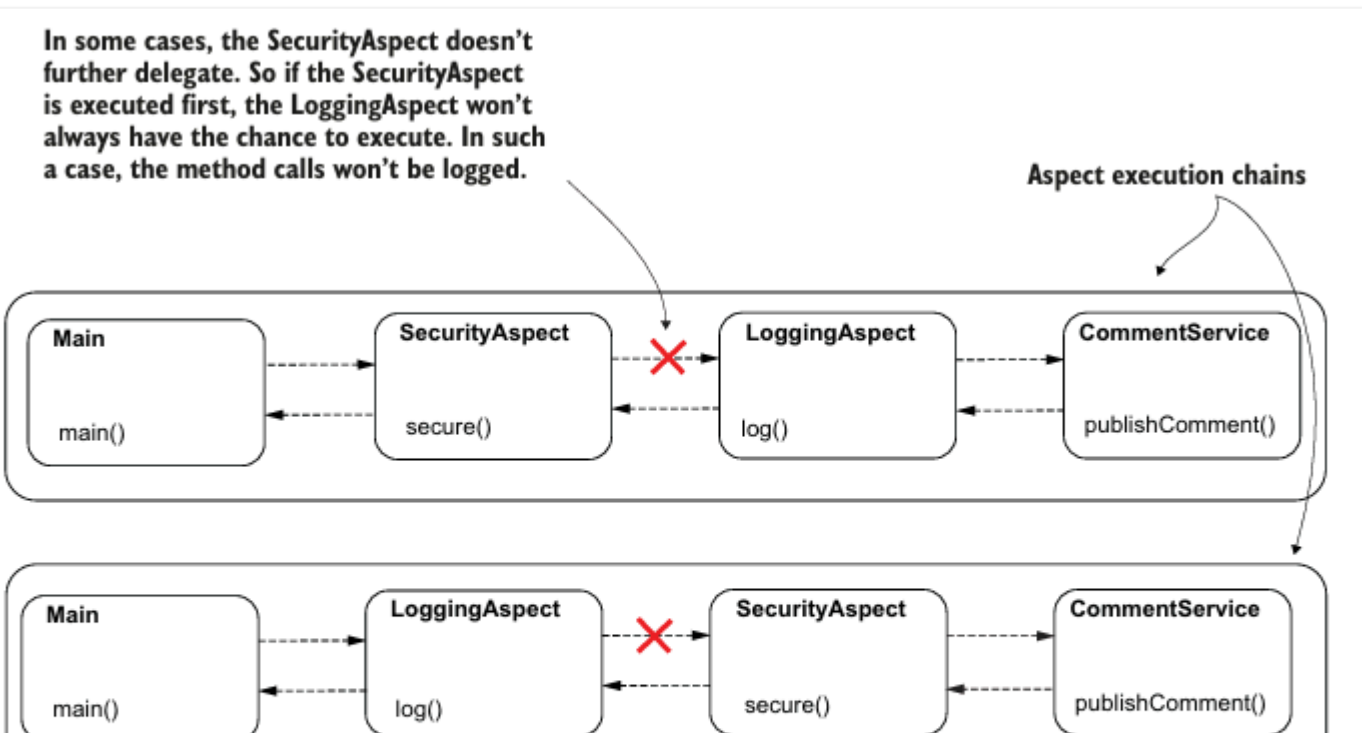
We send a different comment instance as a value to the method's parameter.

We log the value returned by the intercepted method, but we return a different value to the caller.

6. intercepting annotated method
   - we can use custom annotation to specify (mark) some methods and intercept them only and this happens as follow '140/1' :
     1. create custom annotation, accessed at run time .
        1. at runtime because all aspect happens at runtime
     2. tell Aspect to intercept these annotated methods .



7. Other advice annotations :
   - instead of typing complex code like @Around which is very powerful and can intercept and modify intercepted method in every situation but its better to use specific methods that are easy to understand and simple like :
     1. @Before : we tell spring to execute Aspect logic code before executing the intercepted method .
     2. @AfterReturn : execute Aspect logic After intercepted method return, no exception.
     3. @AfterThrowing : execute Aspect logic after intercepted method throw an exception .
     4. @After : execute Aspect after executing intercepted method in general .



   - we do not to receive parameter ProceedingJoinPoint cause the annotation telling Aspect when to executer '144/1'

8. Aspect execution chain :
   - in real world apps we would have more than one aspect to the same method so we had to know which aspect must run before the other so we had to ask our self :
     1. in which order spring executes these aspects ?
     2. does the execution order matter ? '144'
   - in some cases execution order is important cause changing the order may lead to different results:
     - to order methods we do this via :
       - @Order(1) .

```
The LoggingAspect is called first and
delegates to the SecurityAspect.
Sep 29, 2020 6:04:22 PM aspects.LoggingAspect log
INFO: Logging Aspect: Calling the intercepted method
Sep 29, 2020 6:04:22 PM aspects.SecurityAspect secure
INFO: Security Aspect: Calling the intercepted method
Sep 29, 2020 6:04:22 PM services.CommentService publishComment
INFO: Publishing comment:Demo comment
Sep 29, 2020 6:04:22 PM aspects.SecurityAspect secure
INFO: Security Aspect: Method executed and returned SUCCESS
Sep 29, 2020 6:04:22 PM aspects.LoggingAspect log
INFO: Logging Aspect: Method executed and returned SUCCESS
The SecurityAspect returns
to the LoggingAspect.
```

The SecurityAspect is called second and delegates to the intercepted method.

The intercepted method executes.

The intercepted method returns to the SecurityAspect.

Figure 6.15 helps you visualize the execution chain and understand the logs in the console.

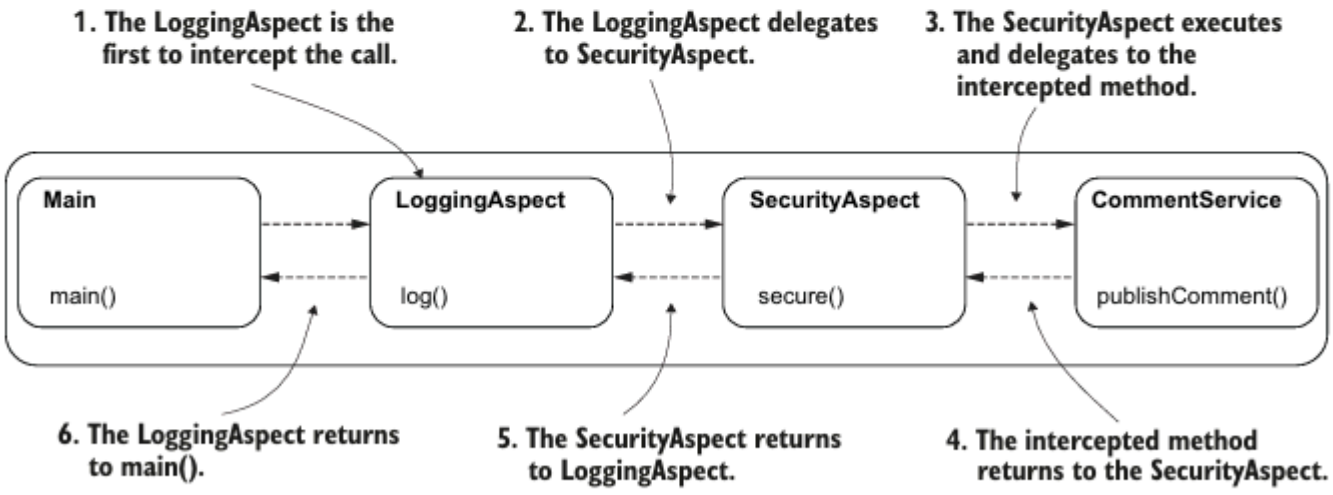
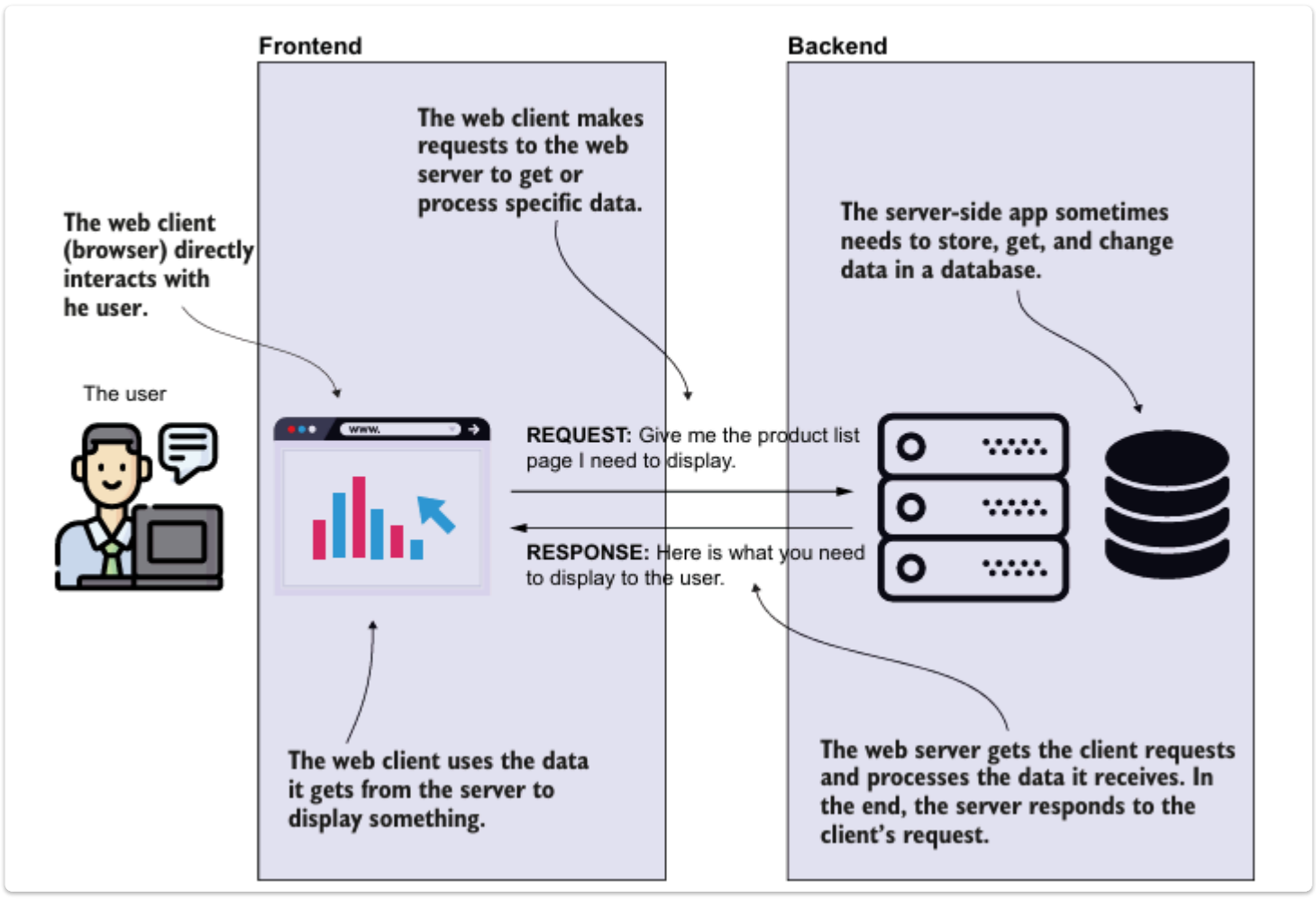


Figure 6.15 The execution flow. The `LoggingAspect` was first to intercept the method call. The `LoggingAspect` delegates further in the execution chain to the `SecurityAspect`, which further delegates the call to the intercepted method. The intercepted method returns to the `SecurityAspect`, which returns further to the `LoggingAspect`.

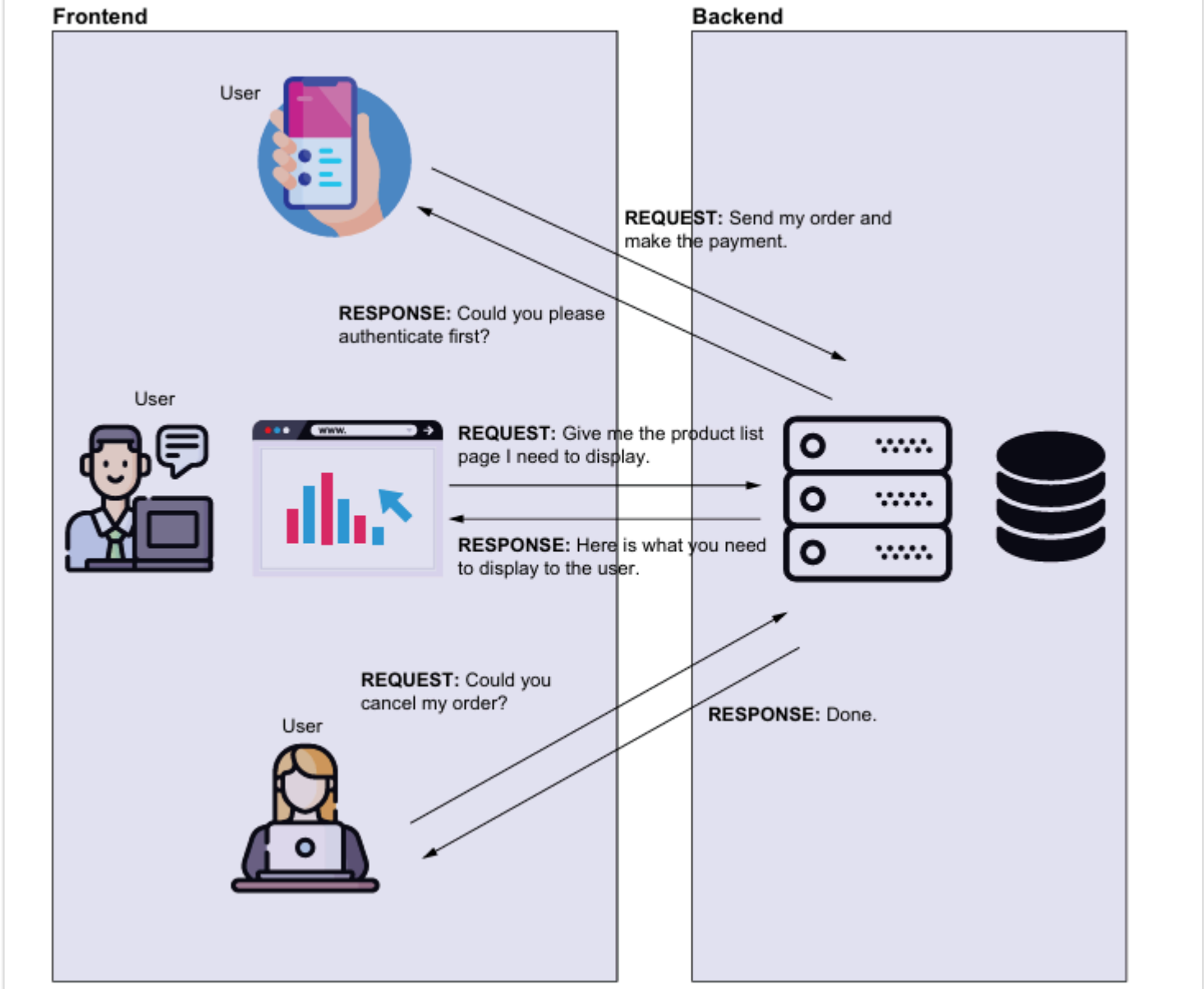
Chapter 7

- this chapter give us a foundation about the web apps, what web apps are, how it works, what are components of web apps, approaches of building web app, how the requests are handled.

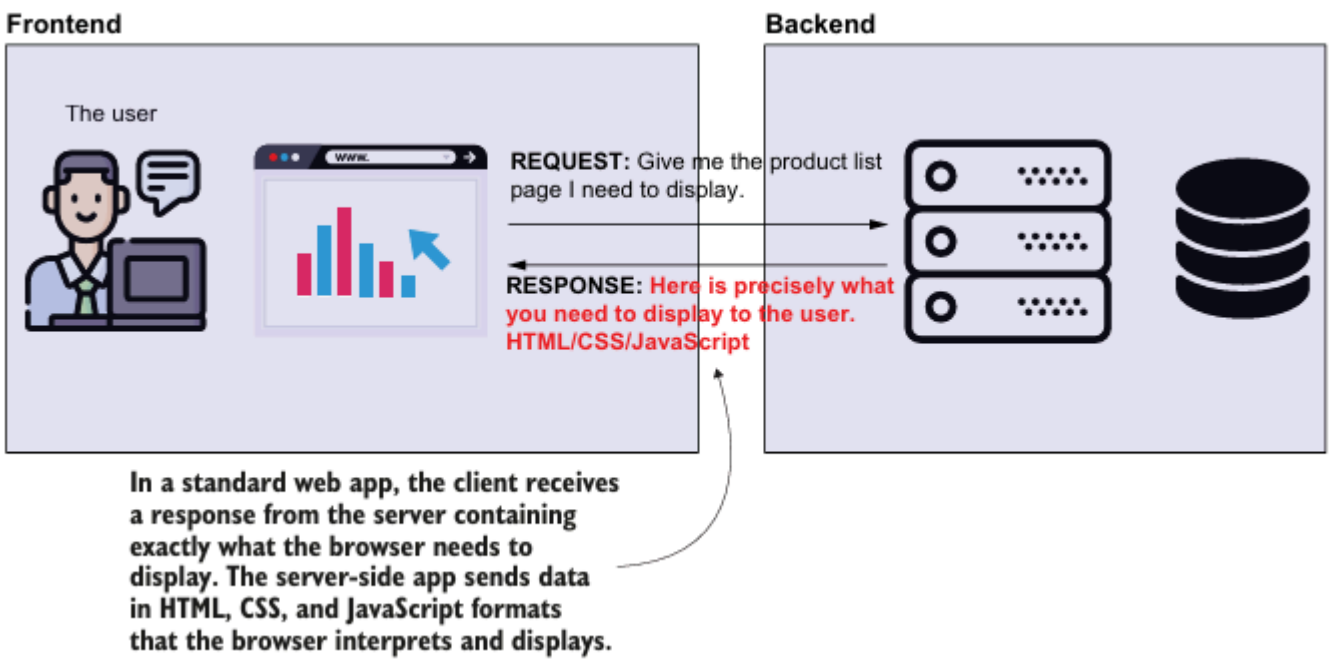
1. what is web app ?
   - web app is any application you can access via web browser '154/1' .
2. general view of web app :
   - what are parts of web app ?
     1. web client : it is the interface that the user interacts with directly, we sometimes refer to it as a browser side . provide a way to user to interact with the app, send request to web server and get responses from it -> frontend '154/1' .
     2. web server : it is the side in which get the client request and process it or store data, response to the request -> response back to web client -> beck end '154/2' .



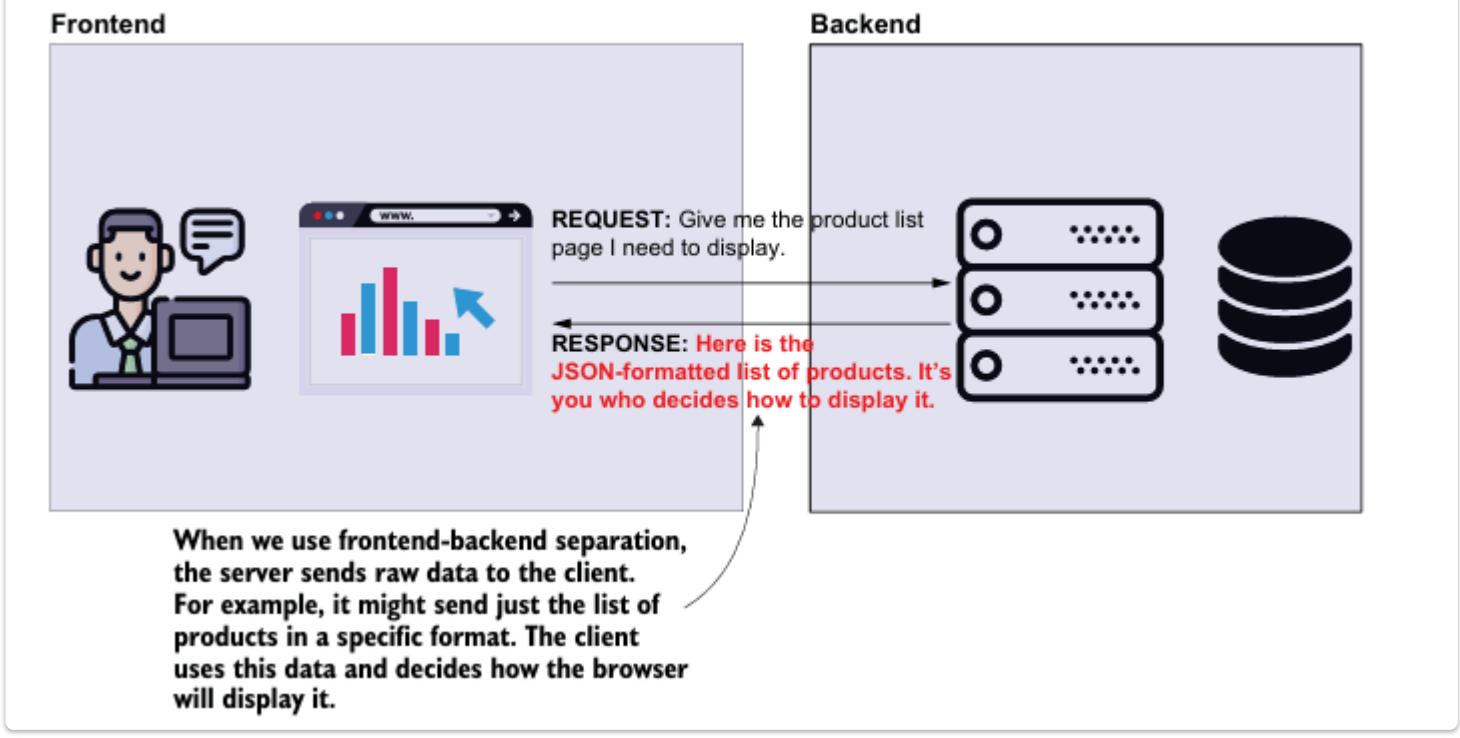
- note :
  - web apps (client-server), the server serves multiple request at the same time (concurrently) .
  - each client -> instance -> make request so avoid the race condition . '156/1,2' .



3. different approaches of implementing web apps :
    1. web server provides fully prepared view and send it to web client (browser) only to display it, server gets request, process it, response in fully prepared view to the browser that will only display the response, web server tells browser how to display it and response in (HTML, CSS, JavaScript) '157/1' .



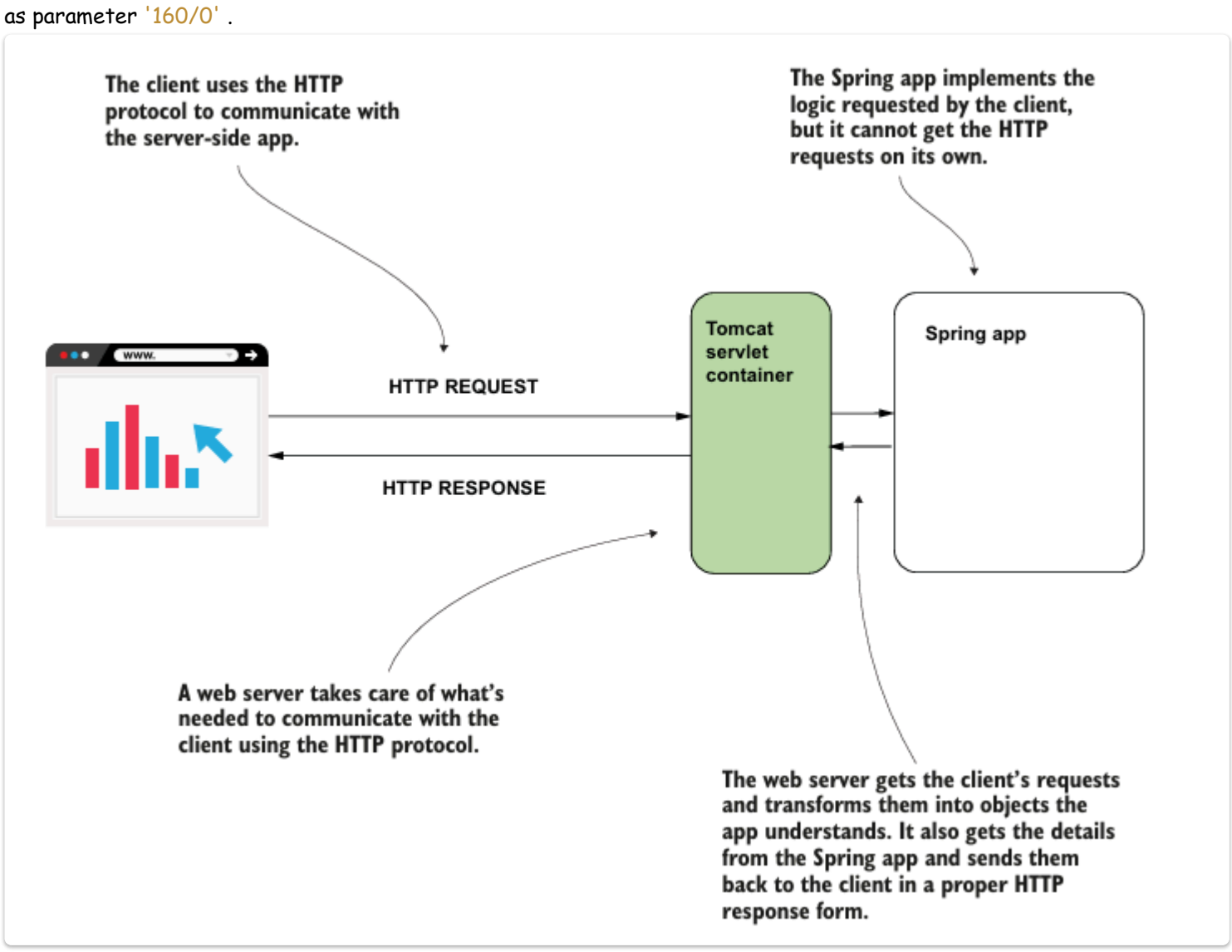
    2. frontend-backend separation : in this way server gets the request, process it and response in raw data, the browser doesn't display it directly, instead it loads frontend app that will get the response and manage how to display the data '157/2' .



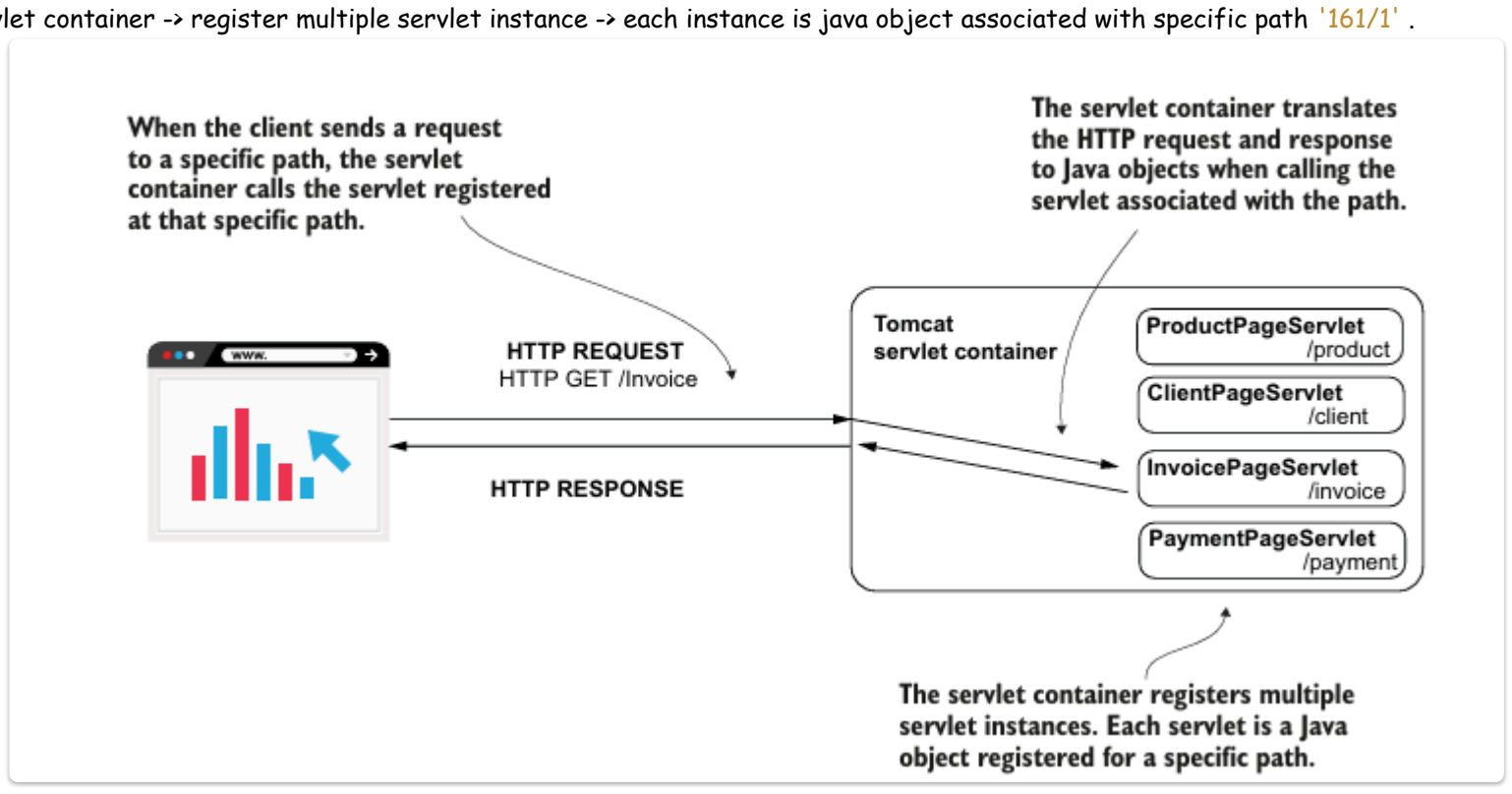
- frontend-backend separation :
    1. called modern approach .
    2. allow more developers to co-operate with each other .

3. make development more manageable .
4. make each team responsible for only one thing (front or back) .-> separate responsibilities .
5. deployment of app can done independently this give flexibility '158/1,2,3,4,5' .

4. Servlet container :
   - web (client) use a protocol called http to communicate with the app using network, in order to make the app understand the protocol there must be something that can understand HTTP and translate the requests and responses between them . this thing called servlet container '159/2,5' .
   - http describes how (client - browser) exchange the data '159/3' .
   - what happens when servlet container gets request ?
     - it calls the method of the servlet and pass request as an object (parameter) to it '159/8' .
     - servlet container gets the request, transfer it to object so that servlet can understand the call the method of servlet and pass request as parameter '160/0' .



   - suppose that you want to build new web page associated with specific path what to do ?
     - you had to create new servlet .
     - configure it into web container .
     - assign a path to it '160/1' .
   - for any path client could request, their must be a servlet instance inside servlet container associated with this path 160/2 .
   - servlet container -> register multiple servlet instance -> each instance is java object associated with specific path '161/1' .



   - what is servlet ?
     - it is a java object that interacts directly with servlet container .
     - entry point of web app .
     - how data gets inside the app and how the app response '161/2' .
   - spring define servlet object -> register it into servlet container -> both can manage it -> if there is any request -> container call method of this object -> in this way we allow servlet to manage requests and responses '161/3' .

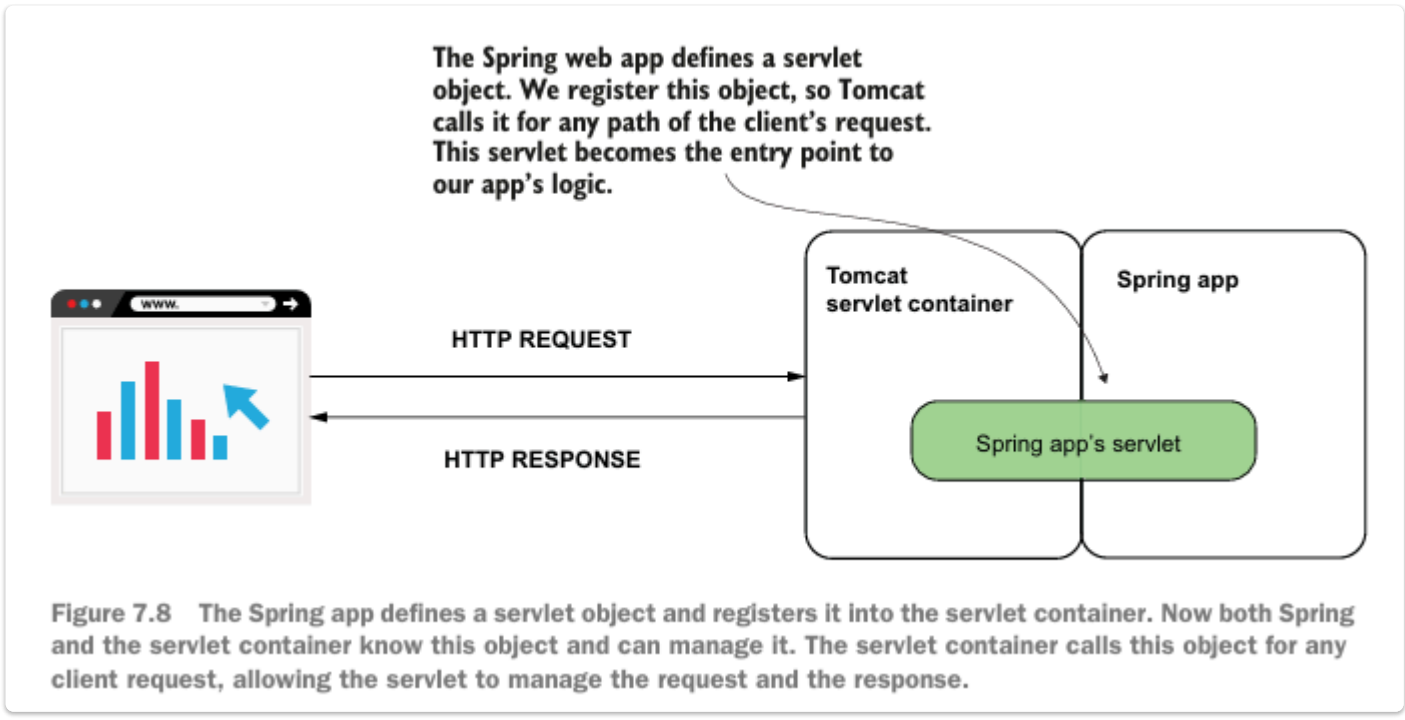


Figure 7.8 The Spring app defines a servlet object and registers it into the servlet container. Now both Spring and the servlet container know this object and can manage it. The servlet container calls this object for any client request, allowing the servlet to manage the request and the response.

5. the magic of spring boot :
   - spring boot is a tool for implementing modern spring apps '162/1' .
   - spring boot help us create more efficient apps and minimize code effort by eliminating code configurations and focus on business code '162/2' .
   - spring boot features :
     1. simplified project creation :
        - by providing project initialization service we can create an empty project with skeleton configuration instead of creating it from scratch .
     2. Dependency starter :
        - in the past if we wanted to create a spring web app, we had to provide all dependencies we want and their versions that had to be compatible with each other . but with dependency starter, we only had to tell spring boot what is the capability we want and spring starter will provide us the dependency we want to this purpose and we don't have to tell spring about the versions we want for compatibility .
     3. Autoconfiguration :
        - based on the dependencies you specified the pom file, spring boot will provide all the default configurations needed to this dependency, like tomcat .

1. Project initialization service :
   1. Main class provided by start.spring.io :
      - main class of spring app .
      - annotated with @SpringBootApplication .
   2. POM parent :
      - parent node added in the pom.xml file
      - provide default configurations we will use inside the app .
      - define dependencies and it is versions to be compatible .
      - we do not need to specify versions inside our app we let this to spring boot .
   3. maven plugin :
      - used to help us in building the app .
   4. dependencies added by start.spring.io :
      - we do not have to specify versions because spring provides them from parent .

2. . dependency starter :
   - group of dependencies provided for specific purpose .
   - used to simplify dependency management .
   - we only say the capability we want and then spring boot provides the dependencies needed for this purpose, group of capability oriented with compatible versions .
   3. AutoConfiguration :
      - depending on the dependencies we add to pom file, spring boot provide the default configuration needed for this dependency, like web dependency needs tomcat. spring boot provide this to us
      - spring boot works with Convention over configuration principle and if there is a specific configuration we want to change we only configure it our self .
      - provide default configuration to the capability we mentioned .

6. implementing web app using spring mvc :
   - Controller :
     - component in web apps used to define methods (actions) that will be linked, executed for specific http request .
   - methods actions inside Controller :
     - used to link Http request with specific web page you want your app provide in response .
     - returns reference to web page the app returns in response .
   - to mark class as controller and add it inside context we use @Controller stereo type annotation .
   - to define actions we use @RequestMapping annotation and provide the path as its value (used to associate action to specific http request path) .
     - method return string of name of page we want to return in response .

7. How Spring handles the request :
   1. client makes http request .
   2. tomcat gets this request then call dispatcher servlet .
   3. dispatcher servlet is the entry point of the app which take http request and manage it further in the app then provide response, tomcat calls dispatcher servlet with any http request (front controller)

4. dispatcher had to find the action that associated with this request so dispatcher delegates this to "handler mapping", after handler mapping find the action, dispatcher servlet execute this method and controller return the name of the page that will be returned in response, we call this page "view".
5. dispatcher had to find view so it delegates this to "view resolver",
6. dispatcher return this view in the http response .

- in spring boot we only had to type the controller actions and map them with the requests, spring provides all the mentioned components above .

Chapter 8

in this chapter we will talk about how to create dynamic pages and use the data on the requests, process it .

- template engine :
  - it is a dependency used to allow as send data from controller to view and it provide a way to display it '177/1' .
- request path not enough to now what the client request, so that it had to be an HTTP method that express the user intention (get, post, delete) 177/2 .
- Implementing web app using dynamic view :
  - dynamic means that the data back end sends to the client might be different for each request '179/1' .
  - view can get dynamic values from controller '179/2' .

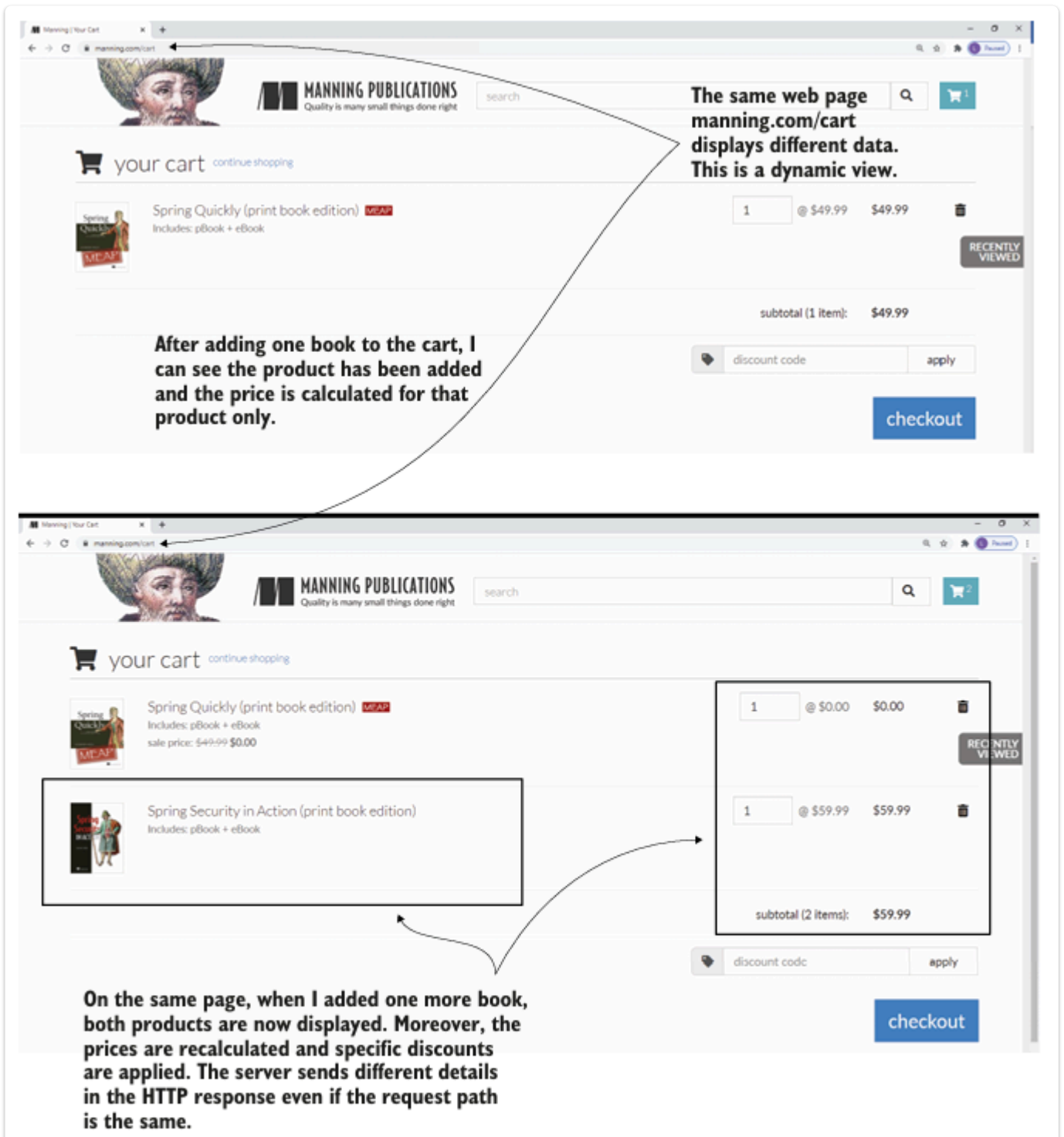


Figure 8.1 A dynamic view presented with the Manning shopping cart functionality. Even if the requested page is the same, the content of the page is different. The backend sent different data in the response before and after adding one more product to the cart.

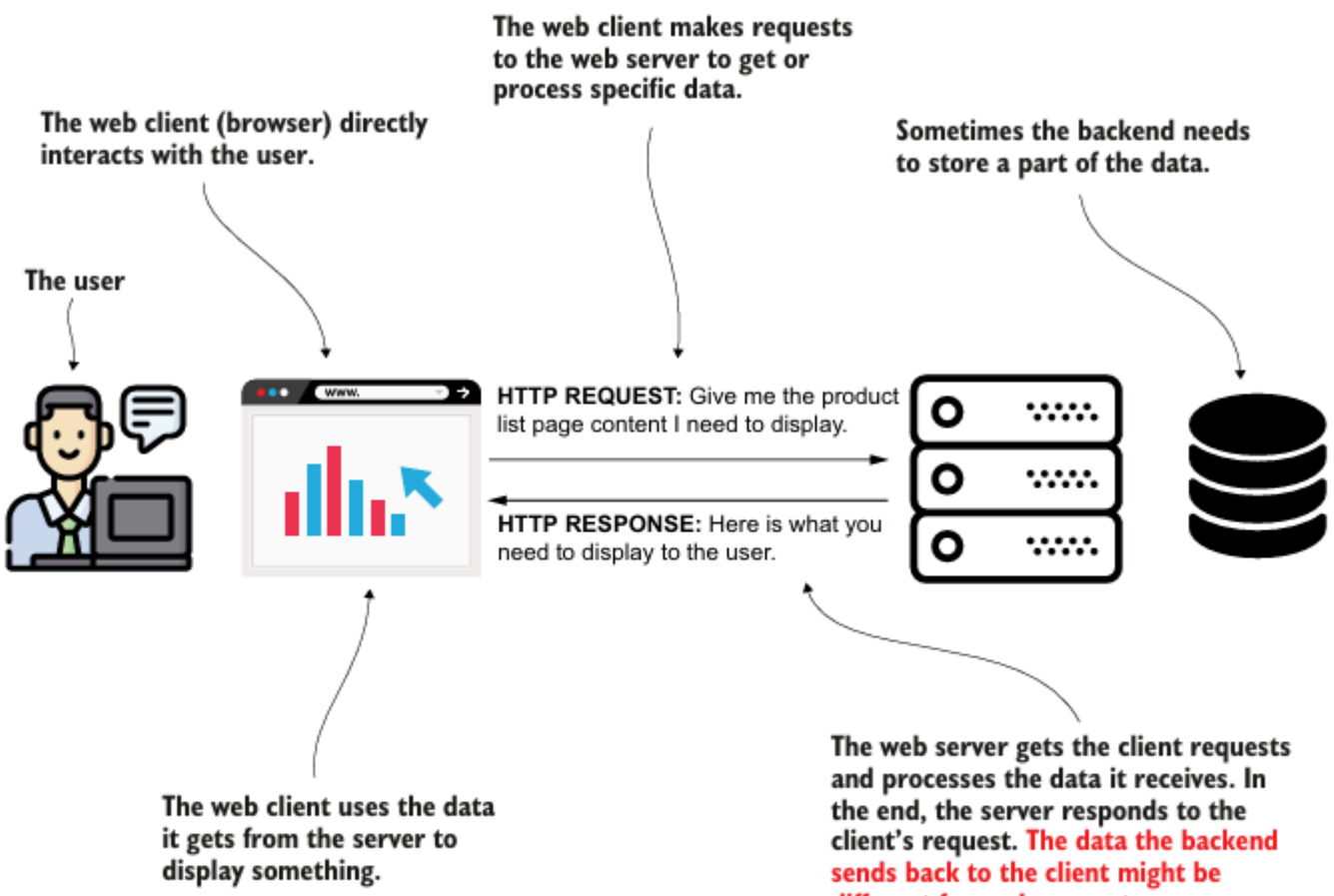


Figure 8.2 A client sends data through the HTTP request. The backend processes this data and builds a response to send back to the client. Depending on how the backend processed the data, different requests may result in other data displayed to the user.

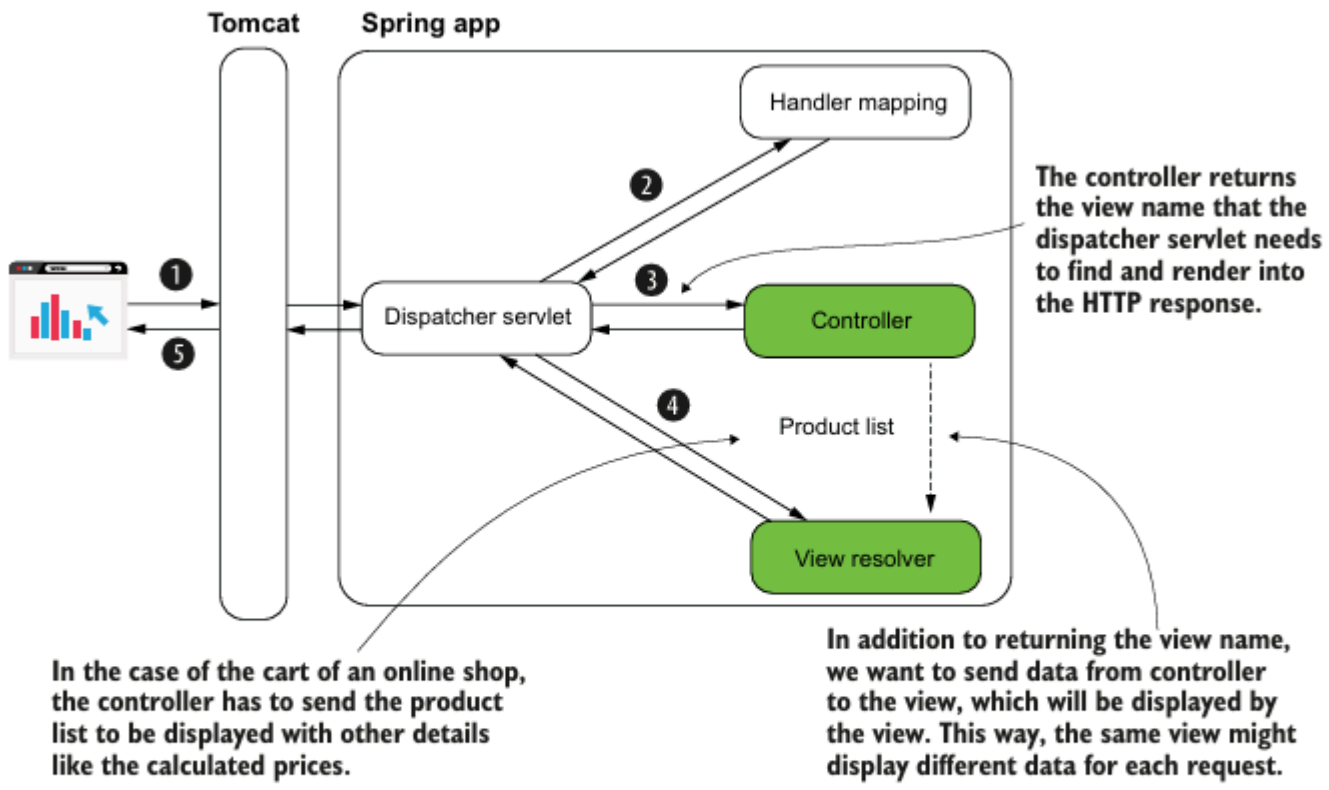


Figure 8.3 The Spring MVC flow. To define a dynamic view, the controller needs to send data to the view. The data the controller sends can be different for each request. For example, in an online shop's cart functionality, the controller initially sends a list of one product to the view. After the user adds more products, the list the controller sends contains all the products in the cart. The same view shows different information for these requests.

- Steps to create dynamic views :
  1. create spring project .
  2. add template engine dependency to pom.xml .
     1. model : used to store data from controller to view (will be sent to view) '181/1' .
        - how to add values to model ?
          - using .addAttribute("key", value);

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

The dependency starter that needs to be added to use Thymeleaf as a template engine

Though you're building a web app, you still need to add the dependency starter for web apps.

Listing 8.1 The controller class defines the page action

```
@Controller
public class MainController {

  @RequestMapping("/home")
  public String home(Model page) {
    page.addAttribute("username", "Katy");
    page.addAttribute("color", "red");
    return "home.html";
  }
}
```

The @Controller stereotype annotation marks this class as Spring MVC controller and adds a bean of this type to the Spring context.

We assign the controller's action to an HTTP request path.

The action method defines a parameter of type Model that stores the data the controller sends to the view.

We add the data we want the controller to send to the view.

The controller's action returns the view to be rendered into the HTTP response.

3.

- NOTE :
  - when creating dynamic view we sill put the view inside the template folder
    - resources/template .
- inside view we had to add prefix of thymeleaf to get the data stored in model

Listing 8.2 The home.html file representing the dynamic view of the app

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">

  <head>
    <meta charset="UTF-8">
    <title>Home Page</title>
  </head>

  <body>
    <h1>Welcome
    <span th:style="'color:' + ${color}"
          th:text="${username}"></span>!</h1>
  </body>

</html>
```

Defines the Thymeleaf "th" prefix

Uses the "th" prefix to use the values sent by the controller

- we will fetch the data using the prefix th and to get attribute value we will use ${key}

- getting data on HTTP request :
  - there are multiple ways to do it .

1. HTTP request parameter :
   - for small quantity of data .
   - used with optional data .
   - typed in URI with query expression .
   - Query parameter .
   - in key, value format .
2. HTTP request head :
   - like http request parameter but the data hidden .
   - small quantity of data .
3. HTTP request path :
   - used with mandatory data .
   - like request parameter .
4. HTTP request body :
   - used when sending big data .
   - data is hidden .
   - text formatted, binary .
   - if we want to (add, update, delete ...) .

1. HTTP request parameter :
   - when ?
     - the quantity of data is small .
       - in a k,v format .
       - max 2,000 char .
     - defining search, filter criteria .
     - the parameter is optional .
       - the server knows it might be empty .

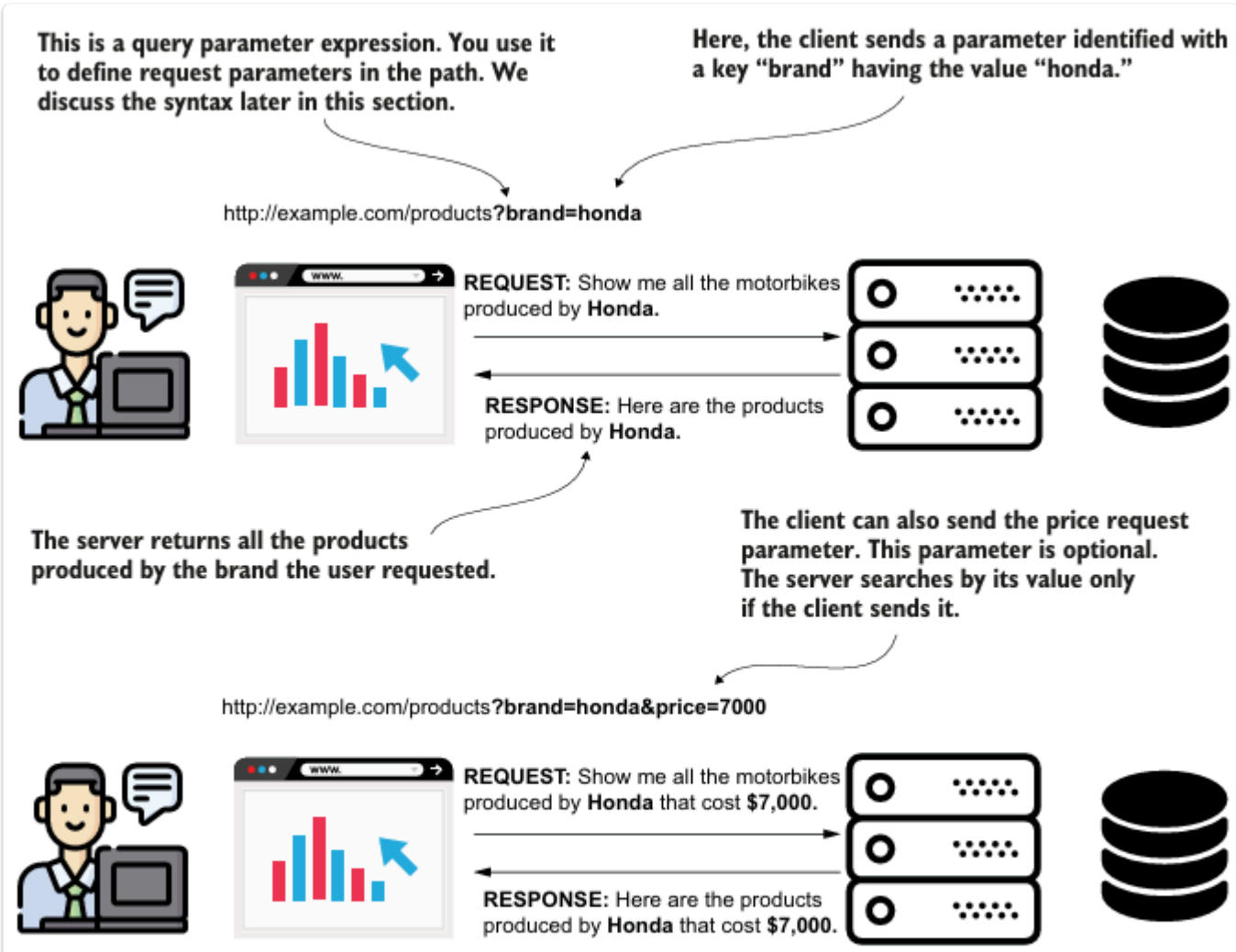


Figure 8.5 A request parameter can be optional. A common scenario for using request parameters is implementing a search functionality where the search criteria are optional. The client sends only some of the request parameters, and the server knows to use only the values it receives. You implement the server to consider it might not get values for some of the parameters.

- we send the data in the request parameter as query .
- it is a simple way to send data from client to backend .
- to receive the parameters we declare a parameter inside controller is action with the @RequestParam annotation named with the same name of the request parameter .
  - in this way we are telling spring to get a value from the request parameter with the name of action parameter .

Listing 8.3 Getting a value through a request parameter

```
@Controller
public class MainController {

  @RequestMapping("/home")
  public String home(
      @RequestParam String color,
      Model page) {
    page.addAttribute("username", "Katy");
    page.addAttribute("color", color);
    return "home.html";
  }
}
```

We define a new parameter for the controller's action method and annotate it with @RequestParam.

We also add the Model parameter that we use to send data from the controller to the view.

The controller passes the color sent by the client to the view.

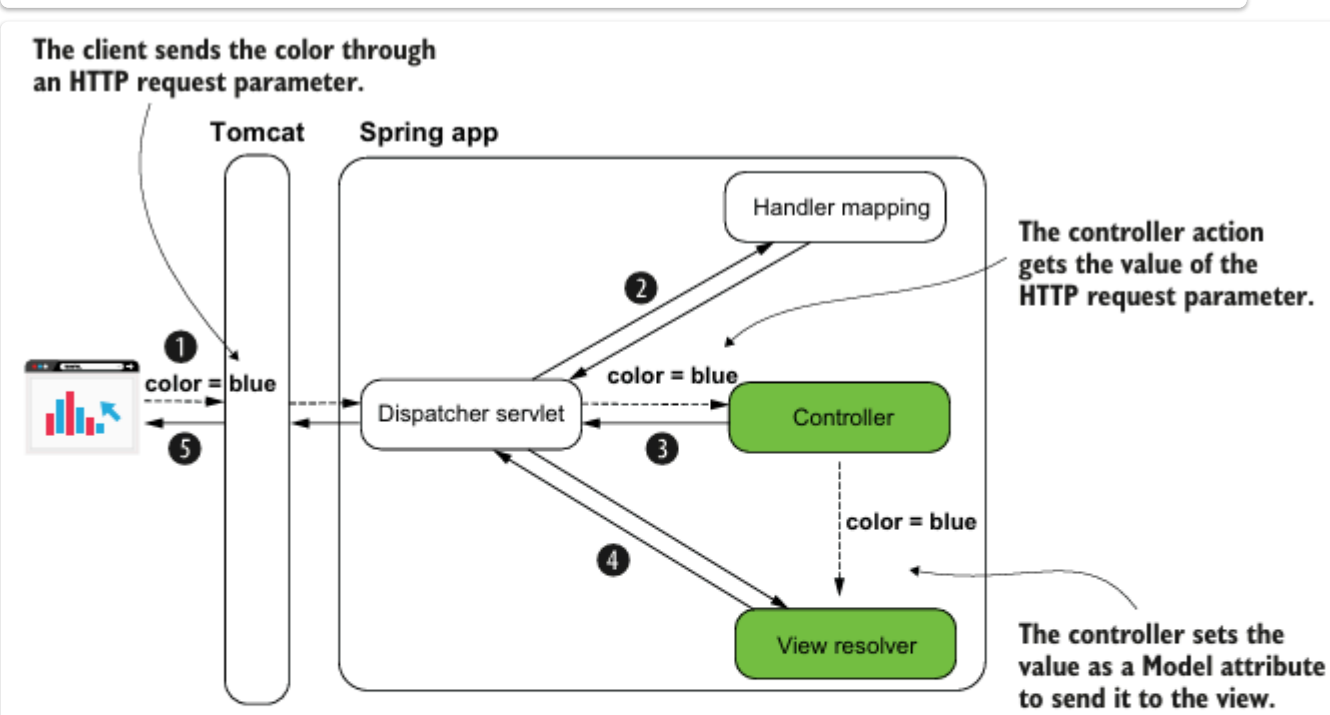


Figure 8.6 The value sent by the client from the Spring MVC perspective. The controller action gets the request parameters the client sends and can use them. In our example, the value is set on the `Model` and delivered to the view.

Run the application and access the /home path. To set the request parameter's value, you need to use the next snippet's syntax:

```
http://localhost:8080/home?color=blue
```

```
@Controller
public class MainController {

  @RequestMapping("/home")
  public String home(
      @RequestParam(required = false) String name,
      @RequestParam(required = false) String color,
      Model page) {
    page.addAttribute("username", name);
    page.addAttribute("color", color);
    return "home.html";
  }
}
```

Gets the new request parameter "name"

Sends the "name" parameter's value to the view

In the group `key=value` (for example, `color=blue`), "key" is the name of the request parameter, and its value is written right after the = symbol.

Figure 8.7 visually summarizes the syntax for request parameters.

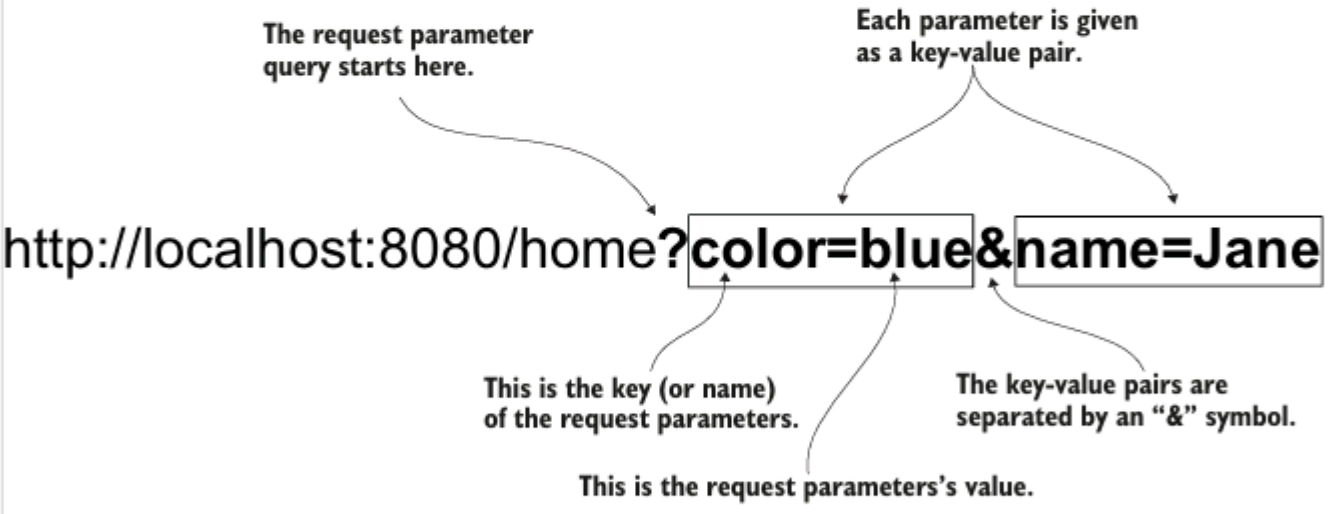


Figure 8.7 Sending data through request parameters. Each request parameter is a key-value pair. You provide the request parameters with the path in a query starting with the question mark symbol. If you set more than one request parameter, you separate each key-value pair with the "and" (&) symbol.

> **NOTE** A request parameter is mandatory by default. If the client doesn't provide a value for it, the server sends back a response with the status HTTP "400 Bad Request." If you wish the value to be optional, you need to explicitly specify this on the annotation using the optional attribute: `@RequestParam(optional=true)`.

2. Http request path :
    1. prefer to use with mandatory values not optional .
    2. easier to read .
    3. directly define variable values in the path .
    4. it is better to not use this with more than a couple parameters .

Listing 8.4 Using path variables to get values from the client

```
@Controller
public class MainController {

  @RequestMapping("/home/{color}")
  public String home(
      @PathVariable String color,
      Model page) {
    page.addAttribute("username", "Katy");
    page.addAttribute("color", color);
    return "home.html";
  }
}
```

To define a path variable, you assign it a name and put it in the path between curly braces.

You mark the parameter where you want to get the path variable value with the @PathVariable annotation. The name of the parameter must be the same as the name of the variable in the path.

Run the app and access the page in your browser with different values for the color.

```
http://localhost:8080/home/blue
http://localhost:8080/home/red
http://localhost:8080/home/green
```

Each request colors the name displayed by the page in the given color. Figure 8.8 visually represents the link between the code and the request path.

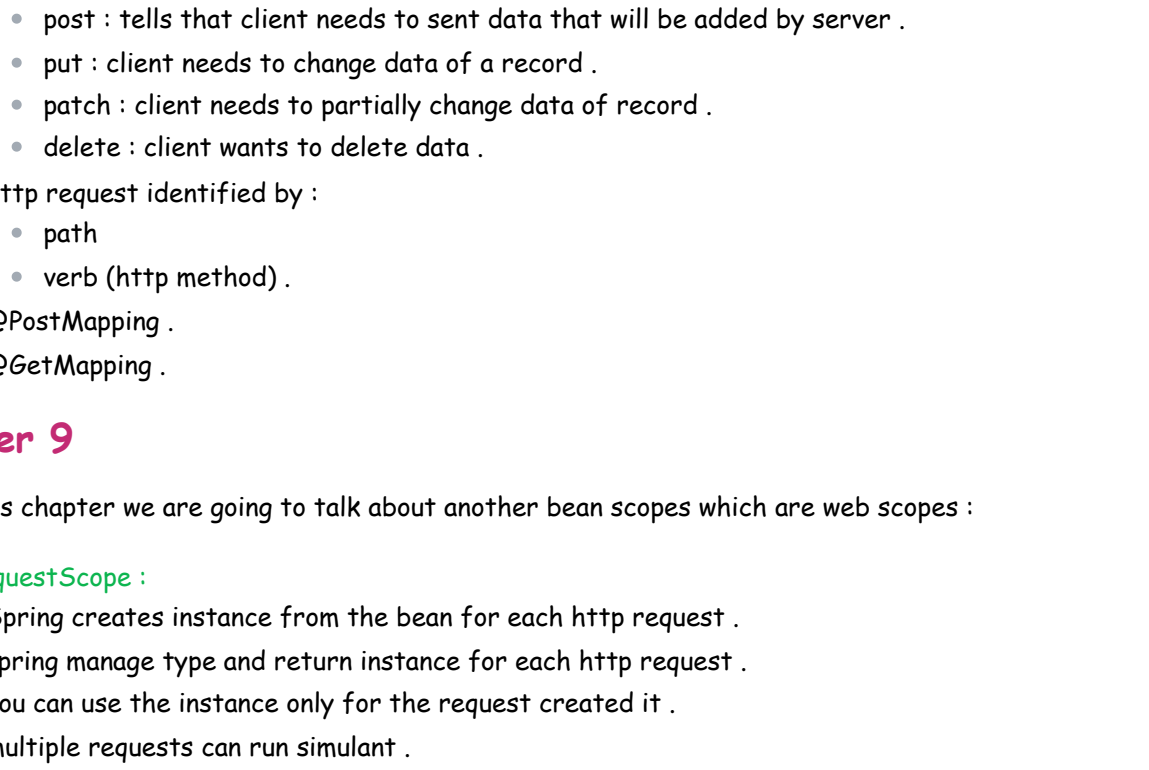


Figure 8.8 Using path variables. To get a value from a path variable, you give the variable a name between curly braces when defining the path on the controller action. You use a parameter annotated with `@PathVariable` to get the value of the path variable.

5.

3. using http methods
   - request path only doesn't describe what client want, so we had to use a verb that describes client intention -> http method .
   - http method :
     - client use it to describe what action it want to apply to requested resource .
     - (get, post, pot, delete)
     - get : tells that client needs to retrieve data .
     - post : tells that client needs to sent data that will be added by server .
     - put : client needs to change data of a record .
     - patch : client needs to partially change data of record .
     - delete : client wants to delete data .
   - http request identified by :
     - path
     - verb (http method) .
     - @PostMapping .
     - @GetMapping .

Chapter 9

- in this chapter we are going to talk about another bean scopes which are web scopes :

1. @RequestScope :
   - Spring creates instance from the bean for each http request .
   - spring manage type and return instance for each http request .
   - you can use the instance only for the request created it .
   - multiple requests can run simulant .
   - each instance is unique for each http request .
   - we can use it when we have sensitive data that we need it to be deleted with the request .
2. @SessionScope :
   - for http session created spring create instance from a session scoped bean in the memory for the whole http session .
   - once user creates request, server reserves a place in the memory for this request for the whole duration of their session .
   - only one instance per http session .

Chapter 10

في الشابتر ده اتكلم عن ال **restful** وعرفنا ايه معناها وبيتم إستخدامها في ايه .

- يعني ايه **rest** ؟
- معناها **representational state transfer** .

- it is a way to implement communication between two apps .

- communicate through end points (path + http method).
- endpoint : controller action mapped to http method and path .
- server sends back in http response what controller action returns, not view .
- Challenges :
  - time out due to:
    1. action is logic takes much time to execute so the request call will timeout .
    2. client send huge amount of data in the request body so it may take long time to be processed .
  - multiple requests on the same endpoint may lead to failure .
  - network unreliable so it may affect the communication .
- @ResponseBody :
  - used to till spring that this action will return the data directly not view .
  - instead of duplicating this annotation we could use @RestController to tell spring that this controller implements the actions of end points and the methods return the data directly .
- HTTP response :
  - header : small amount of data may be some words, used to add some description or mete data about the response .
  - body : used to send huge amount of data .
  - status : used to specify the state of the request http/1.0 200 ok .
- @ResponseEntity :
  - used to send data in the response header, body, status .

- handling exceptions :

1. straight way
2. using @RestControllerAdvice :
   - used to define the controller as aspect that will intercept the exceptions throwed by controller actions and applies the aspect logic defined for this specific exception .

- @ExceptionHandler :
  - used to define what exception triggers the aspect logic, what exception you want aspect logic executed for .

- @RequestBody:
  - used to get data that sent inside the request body by defining it in the parameter and spring will decode the format "JSON" to the type of the parameter you specified, if not; returns badRequest .

Chapter 11

in this chapter we talk about how to call rest endpoints and make communication between two apps and the ways are:

1. OpenFeign :
   - the first approach is using open feign which is :
     - from spring cloud family .
     - simple syntax .
     - we only define interface & open feign provides the implementation .
     - modern approach in spring for calling rest endpoints .
   - how to use it ?
     1. add openfeign dependency .
     2. @EnableFeignClients :
        - used to tell feign where to find the clients contracts .
        - used to enable open feign .
     3. @FeignClient :
        - used to tell the tool that it has to provide implementation to this contract .
        - used to define interface as feign client .
        - each method defined inside the contract represent rest endpoint call .
2. RestTemplate :
   - old approach
   - had been into maintenance mode .
   - not simple .
   - how to use ?
     1. create bean from RestTemplate inside config class .
     2. define proxy class with @component .
     3. inject RestTemplate using Constructor DI .
     4. define the action that will call the service
        1. create RequestHeader Instance to add header values "define type of content" :
           - header.add("key",value);
        2. create HTTPEntity<class> to add the data of the request inside the constructor :
           - new HttpEntity(model,header);
        3. create Response entity that will be returned from :
           - restTemplate.exchange(uri, HttpMethod, HttpEntity, result.class)
        4. return response.getBody();
3. WebClient :
   - alternative way to RestTemplate .

- with reactive programming .
- how :
    1. add flusk dependency .
    2. add bean of WebClient to spring context .
    3. create proxy :
        1. inject webClient instance from context .
        2. declare the method :
            - return type is Mono<Model>
            - return webClient.method.
              .uri("")
              .header("key",value)
              .body(Mono.just(model),Model.class)
              .retrieve()
              .bodyToMono(Model.class) .

Chapter 12

this chapter talks about data source and how they are useful.

1. what is data source ?
    - it is a component, object used to manage connection of DBMS for he application .
2. why to use data source ?
    - JDBC driver only not enough cause it creates a new connection each time and this may take time and lead to less performance but with data source :
        1. give a new connection only if needed .
        2. manage connection better .
        3. connection pool which make sure make new connection each time .
3. How spring boot creates data source bean ?
    - when the framework sees the settings inside the properties file & the existence of the dependency it is smart enough to provide bean based on these settings .
4. what is JDBCTemplate ?
    - it is a simple way or abstract way of the JDBC it self which makes it easier to us to deal with the operations on DB, eliminate boilerplate code and let us focus on business logic .
5. describe how JDBC template is created ?
    1. we add dependency of (JDBC driver - JDBC "libraries") .
    2. add configuration inside properties file .
    3. spring boot will use the driver to make a connection and make a data source bean based on the driver connection .
    4. spring will use the data source bean and inject it inside JDBC template bean and it becomes ready for use.
6. what is row mapper ?
    - it is a way to tell JDBC template how to map, transform each row of result set into a specific class type .

```
RowMapper<Model> rowMapper = (rs, rowCount) -> {
    Model a = new Nodel();
    a.setId(rs.getInt("id"));
    return a
};
```

7. example on JDBC template
    ```java
    String sql = "select * from table where id = ?";
    return JDBCTemplate.queryForObject(sql,rowMapper,id);
    ```

8. what is schema.sql ?
    - a file used to store DDL queries to be applied by spring and spring will by default apply them if we are using in memory database .
9. how to configure connection inside Application.properties file ?
    1. name.datasource.url=
    2. name.datasource.username=
    3. name.datasource.password=

Chapter 13

this chapter discussed what transaction is and how it help us to save our data during unsuccessful operations and introduced us to @transctional operation .

- what is transaction ?
    - it is a set of mutable operations (change data) that had to run all together or not at all (atomicity) .
- why to use transactions ?
    - we use transactions to avoid data inconsistency .
- what are the main options of a transaction ?
    - commit :
        - it is an operation that happens when transaction ends and all steps successfully execute so after that it persists the data .
    - rollback :
        - it happens when any of steps inside transaction throws an exception, the spring restore data to how it looks before transaction .
- what is @transactiol annotation ?
    - it is an annotation used to tell spring to wrap the execution of this method .

    - it can be used with method or on the class (to annotate all methods inside the class) .
- how transaction works in spring ?
    - spring aspect configure transaction logic and intercept the annotated transactions, if any step fails, spring rollback data. if it executed successfully, it commits data .

Chapter 14

in this chapter we discover what spring data is and how it works .

- what is spring data :
    - it is a project of spring ecosystem which allow us to implement persistence layer with less effort by providing set of contracts .
    - simplifies development of persistence layer by providing implementations according to technology we use .
- why we use spring data ?
    - we use spring data to persist our spring app data .
    - to implement persistence layer .
    - there are to many technologies that we can use to implement persistence layer so we need to unify them with common contracts provided by spring data .
- how spring data works ?
    - it consist of 3 contracts :
        - repository : marker interface , has no methods .
        - CRUDRepository : used for simple operations like, reading , updating, deleting .
        - Paging&SortingRepository : extends all and adds more operations like sorting and paging the records .
    - we only extend the contract and spring data creates a bean inside the context that implements the contract and we can use it via DI .
    - there is nothing calls spring data dependency only, they are set of modules provided by spring data and we choose the module which meets the technology we use .