cognizant

Spring Fundamentals

Alexandru Babei - Software Engineer
Sorin Miron - Software Engineer

March 2024

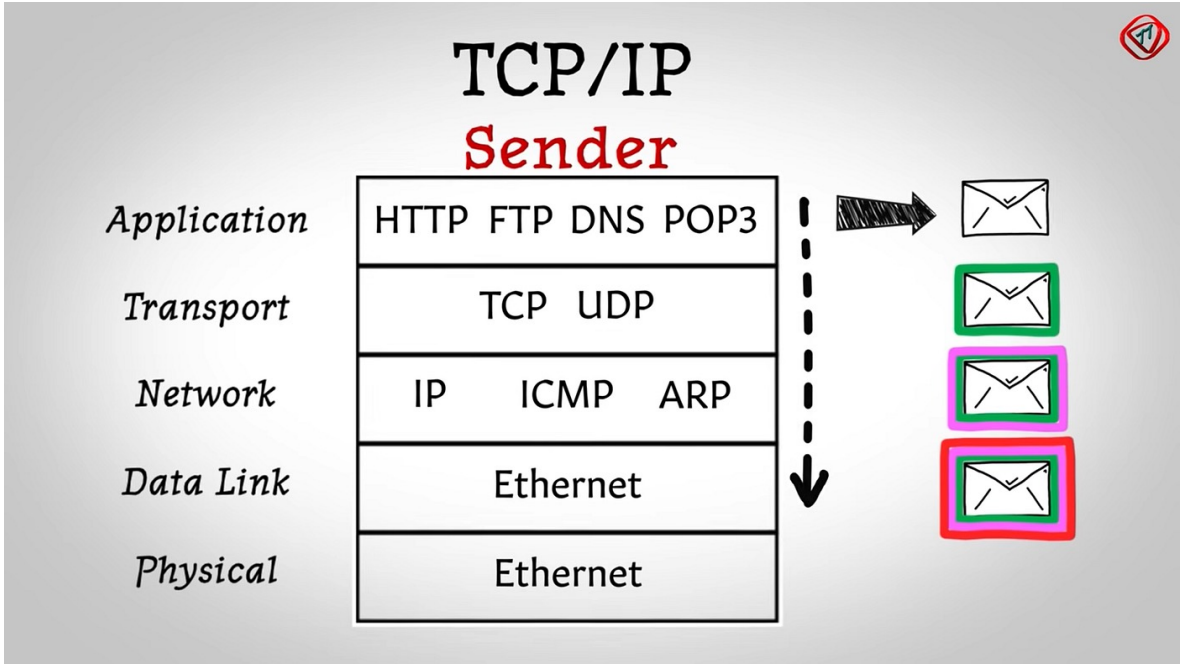
DIVE INTO THE WEB

Agenda

- HTTP Protocol Highlights
- HTTP Protocol Status Codes
- HTTP Verbs
- Spring MVC
- JSON vs XML
- REST API

HTTP highlights

TCP/IP Protocol Stack



HTTP - Details

The **Hypertext Transfer Protocol** is an application-level protocol for distributed, collaborative, hypermedia information systems.

This is the foundation for data communication for the **World Wide Web**.

HTTP is a **communication protocol**, which is used to deliver data HTML files, image files, queries on the World Wide Web.

The default port is **TCP 80**, but other ports can be used.

It provides a **standardized way for computers to communicate** with each other.

HTTP specification defines how clients request data will be constructed and sent to the server, and how servers respond to these requests.

HTTP - Features

HTTP is stateless: The HTTP client (ie. browser) opens a TCP connection, sends an HTTP request, and waits for an HTTP response. The client and server can close the underlying TCP connection and forget about each other after any HTTP exchange. Due to this nature of the protocol, neither the client nor the browser can retain information between different request across the web pages.

HTTP/1.0 uses a new connection for each request/response exchange

HTTP/1.1 connection may be used for one or more request/response exchanges.

HTTP is media independent: Any type of data can be sent by HTTP as long as both the client and the server know how to handle the data content. It is required for clients and servers as well to specify the content type using the appropriate MIME-type.

What can be controlled by HTTP?

Caching

How documents are cached can be controlled by HTTP. The server can instruct proxies and clients, about what to cache and for how long. The performance of websites and applications can be significantly improved by reusing previously fetched resources.

Authentication

Some pages may be protected so that only specific users can access them. Basic authentication may be provided by HTTP, either using the WWW-Authenticate and similar headers or by setting a specific session using HTTP cookies.

What can be controlled by HTTP?

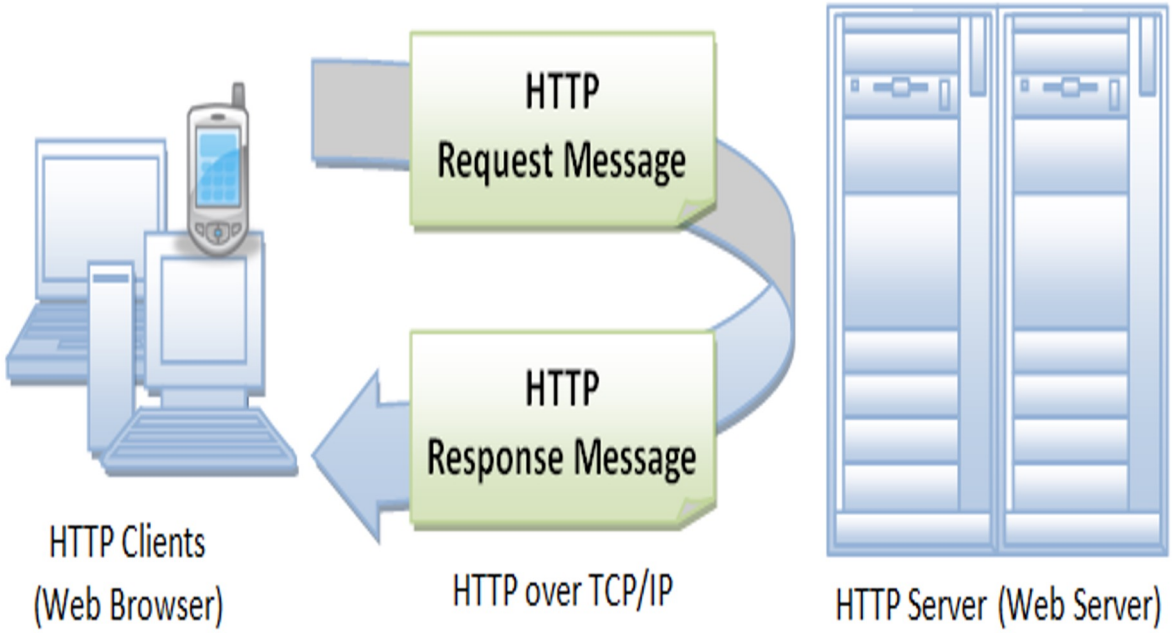
Proxy and tunneling

Servers or clients are often located on intranets and hide their true IP address from other computers. HTTP requests then go through proxies to cross this network barrier. Not all proxies are HTTP proxies. The SOCKS protocol, for example, operates at a lower level. Other protocols, like ftp, can be handled by these proxies.

Sessions

Using HTTP cookies allows you to link requests with the state of the server. This creates sessions, despite basic HTTP being a state-less protocol. This is useful not only for e-commerce shopping baskets, but also for any site allowing user configuration of the output.

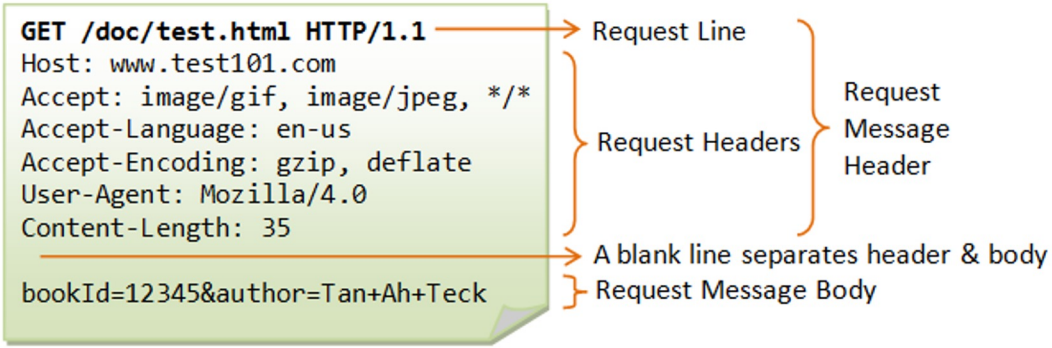
How does it work?



HTTP - Messages

There are two types of HTTP messages, **requests** and **responses**, each with its own format.

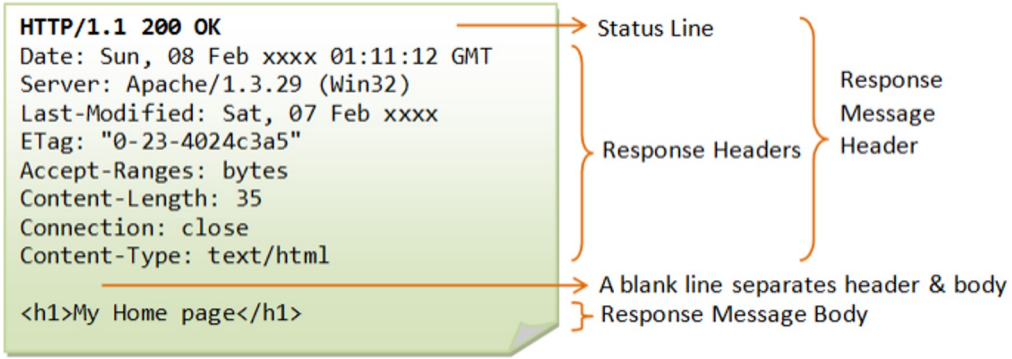
HTTP - Requests



Requests consist of the following elements:

- ➔ An HTTP method, usually a verb like GET, POST, or a noun like OPTIONS or HEAD defines the operation the client wants to perform. Typically, a client wants to fetch a resource (using GET) or post the value of an HTML form (using POST), though more operations may be needed in other cases.
- ➔ The path of the resource to fetch; the URL of the resource stripped from elements that are obvious from the context, for example without the protocol (e.g. http://), the domain (e.g. developer.mozilla.org), and the TCP port (e.g. 80).
- ➔ The version of the HTTP protocol.
- ➔ Optional headers that convey additional information for the servers.
- ➔ Or a body, for some methods like POST, similar to those in responses, which contain the resource sent.

HTTP - Responses



Responses consist of the following elements:

- ➔ The version of the HTTP protocol they follow.
- ➔ A status code, indicating if the request was successful, or not, and why.
- ➔ A status message, a non-authoritative short description of the status code.
- ➔ HTTP headers, like those for requests.
- ➔ Optionally, a body containing the fetched resource.

Response Status Codes

cognizant softvision

Response status codes

The first line of the response message contains the response **status code**, which is generated by the server to indicate the outcome of the request.

The status code is a 3-digit number:

- **1xx (Informational):** Request received, server is continuing the process.
- **2xx (Success):** The request was successfully received, understood, accepted, and serviced.
- **3xx (Redirection):** Further action must be taken in order to complete the request.
- **4xx (Client Error):** The request contains bad syntax or cannot be understood.
- **5xx (Server Error):** The server failed to fulfill an apparently valid request.

Response status codes - examples

200 OK

The request is fulfilled

301 Move Permanently

The resource requested for has been permanently moved to a new location.

302 Found & Redirect (or Move Temporarily)

Same as 301, but the new location is temporarily moved to a new location. The client should issue a new request, but applications need not update the references.

400 Bad Request

Server could not interpret or understand the request, probably syntax error in the request message.

401 Authentication Required

The requested resource is protected, and require client's credential (username/password). The client should re-submit the request with his credential.

403 Forbidden

Server refuses to supply the resource, regardless of identity of client.

Response status codes - examples

404 Not Found

The requested resource cannot be found in the server.

405 Method Not Allowed

The request method used, e.g., POST, PUT, DELETE, is a valid method. However, the server does not allow that method for the resource requested.

408 Request Timeout

500 Internal Server Error

Server is confused, often caused by an error in the server-side program responding to the request.

502 Bad Gateway

Proxy or Gateway indicates that it receives a bad response from the upstream server..

503 Service Unavailable

Server cannot response due to overloading or maintenance. The client can try again later.

HTTP - Verbs

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource. Although they can also be nouns, these request methods are sometimes referred to as HTTP verbs. Each of them implements a different semantic, but some common features are shared by a group of them: e.g. a request method can be **safe**, **idempotent**, or **cacheable**.

HTTP - Properties

Safe

An HTTP method is safe if it **doesn't alter the state of the server**. In other words, a http method is safe if it leads to a **read-only operation.**

All safe methods are also idempotent, but not all idempotent methods are safe. For example, PUT and DELETE are both idempotent but unsafe.

Browsers can call safe methods without fearing to cause any harm to the server;

HTTP - Properties

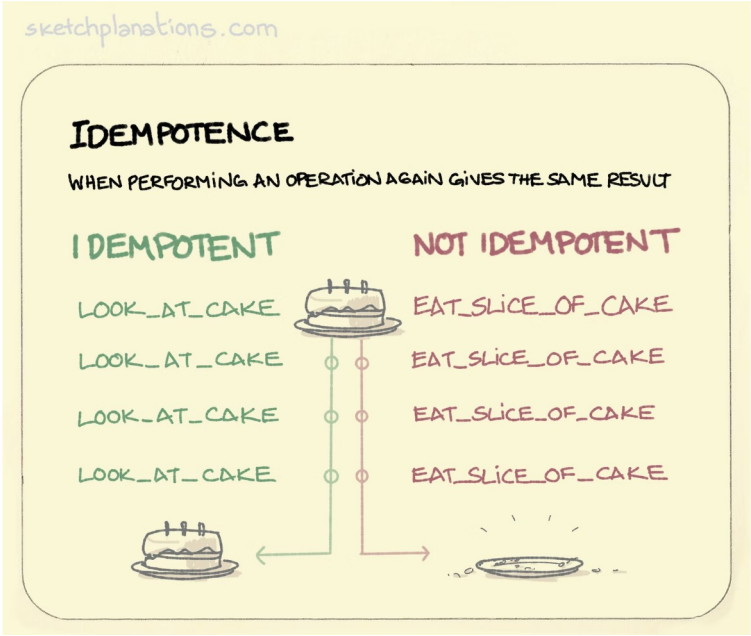
Idempotent

An HTTP method is idempotent if an identical request can be made once or several times in a row with the same effect while leaving the server in the same state.

In other words, an idempotent method should not have any side-effects (except for keeping statistics).

Implemented correctly, the GET, HEAD, PUT, and DELETE methods are idempotent, but not the POST method. All safe methods are also idempotent.

*** The idempotence of a method is not guaranteed by the server and some applications may incorrectly break the idempotence constraint.



HTTP – Verbs Properties

Cacheable

A cacheable response is an HTTP response that can be cached, that is stored to be retrieved and used later, saving a new request to the server. Not all HTTP responses can be cached

The method used in the request is itself cacheable, that is either a GET or a HEAD method. A response to a POST or PATCH request can also be cached if freshness is indicated and the Content-Location header is set, but this is rarely implemented.

Other methods, like PUT or DELETE are not cacheable and their result cannot be cached.

The status code of the response is known by the application caching, and it is considered cacheable. The following status code are cacheable: 200, 203, 204, 206, 300, 301, 404, 405, 410, 414, and 501.

There are specific headers in the response, like Cache-Control, that prevents caching.

HTTP

Verbs

Properties of request methods

Request method	RFC	Request has payload body	Response has payload body	Safe	Idempotent	Cacheable
GET	RFC 9110	Optional	Yes	Yes	Yes	Yes
HEAD	RFC 9110	Optional	No	Yes	Yes	Yes
POST	RFC 9110	Yes	Yes	No	No	Yes
PUT	RFC 9110	Yes	Yes	No	Yes	No
DELETE	RFC 9110	Optional	Yes	No	Yes	No
CONNECT	RFC 9110	Optional	Yes	No	No	No
OPTIONS	RFC 9110	Optional	Yes	Yes	Yes	No
TRACE	RFC 9110	No	Yes	Yes	Yes	No
PATCH	RFC 5789	Yes	Yes	No	No	No

GET (Safe=Yes, Idempotent=Yes, Cacheable=Yes)

The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.

HEAD* (Safe=Yes, Idempotent=Yes, Cacheable=Yes)

Retrieving metadata about the resource, e.g. its media type or its size, before making a possibly costly retrieval

Checking whether a resource has changed. This is useful when maintaining a cached version of a resource

* is actually a noun, not a verb

HTTP - Verbs

POST (Safe=No, Idempotent=No, Cacheable=Yes)

Creates resources on the server, several identical resources if the same payload is sent multiple times.

PUT (Safe=No, Idempotent=Yes, Cacheable=No)

Creates new resource on the server, if it doesn’t exist yet or updates the already existing one.

PATCH (Safe=No, Idempotent=No, Cacheable=No)

Similar to PUT, only that it applies partial modifications to a resource.

DELETE (Safe=No, Idempotent=Yes, Cacheable=No)

Deletes the specified resource from the server.

HTTP - Verbs

CONNECT (Safe=No, Idempotent=No, Cacheable=No)

Establishes a tunnel to the server identified by the target resource.

TRACE (Safe=Yes, Idempotent=Yes, Cacheable=No)

Determines what changes, if any, are made between the client and server in the HTTP request chain.

OPTIONS* (Safe=Yes, Idempotent=Yes, Cacheable=No)

Identifies which HTTP methods a resource supports, e.g. can we DELETE it or update it via a PUT?

* is a noun, not a verb

HTTP – Verbs Properties

Idempotent

GET /pageX HTTP/1.1 is idempotent. Called several times in a row, the client gets the same results:

GET /pageX HTTP/1.1

GET /pageX HTTP/1.1

GET /pageX HTTP/1.1

HTTP – Verbs Properties

POST /user HTTP/1.1 is not idempotent; if it is called several times, it adds several rows:

POST /user HTTP/1.1

POST /user HTTP/1.1 -> Adds a 2nd user

POST /user HTTP/1.1 -> Adds a 3rd user

DELETE /idX/delete HTTP/1.1 is idempotent, even if the returned status code may change between requests:

DELETE /user/123 HTTP/1.1 -> Returns 200 if user 123 exists

DELETE /user/123 HTTP/1.1 -> Returns 404 as it just got deleted

DELETE /user/123 HTTP/1.1 -> Returns 404

Identify resources on web

cognizant softvision

HTTP – Resources on web

The target of an HTTP request is called a **"resource"**: it can be a document, a photo, an object, or anything else. Each resource is identified by a **Uniform Resource Identifier (URI)** used throughout HTTP for identifying resources.

The identity and the location of resources on the Web are mostly given by a single URL

URL (Uniform Resource Locator)

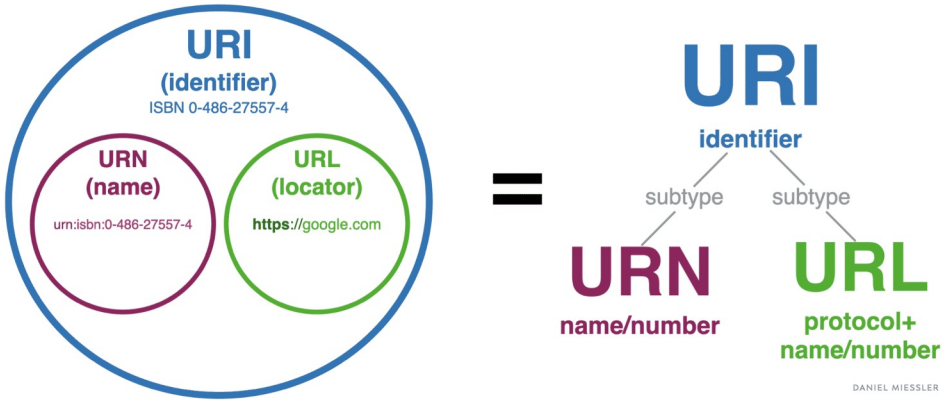
The most common form of URI is the Uniform Resource Locator (URL), which is known as the web address

e.g. https://www.cognizantsoftvision.com/

URN (Uniform Resource Name)

It is a URI that identifies a resource by name in a particular namespace.

e.g. urn:isbn:9780141036144



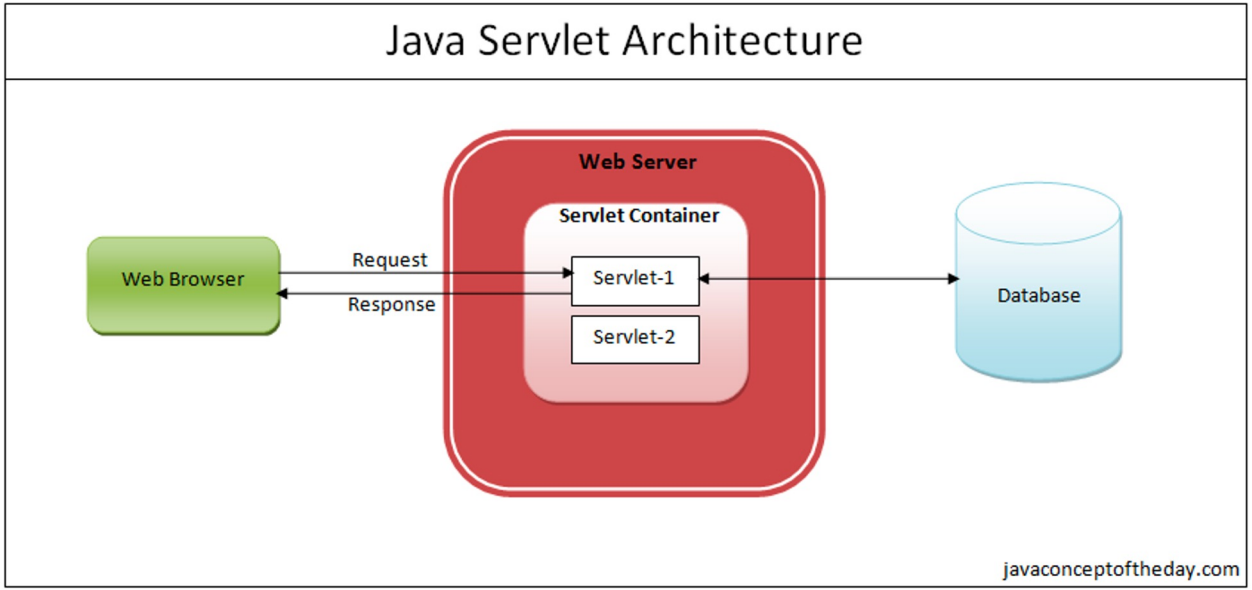
Demo

SPRING MVC

Java Servlet Specification

SERVLET

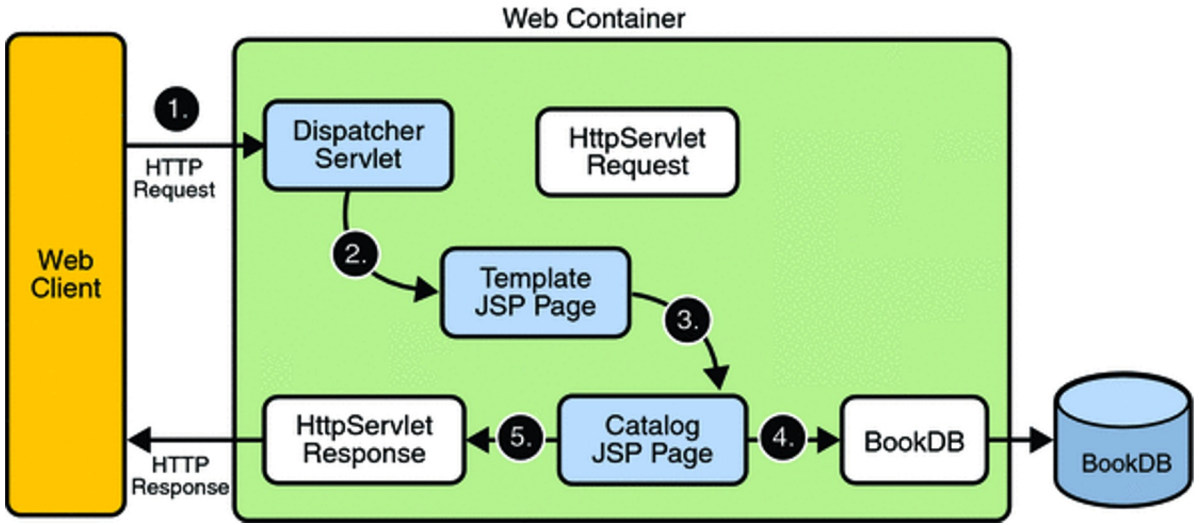
- a Java class commonly used to extend the capabilities of **web servers** that host java applications.
  - implements a request-response programming model.



JSP

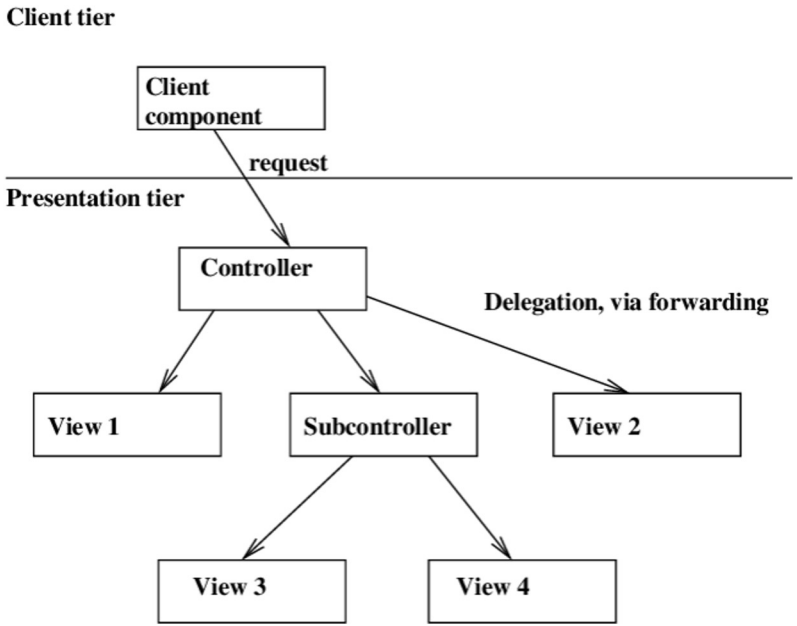
Java Server Pages (JSP) is a Java standard technology that enables you to write **dynamic, data-driven pages** for your Java web applications.

JSP is built on top of the **Java Servlet specification**.



Front Controller Java Enterprise Pattern

- Java EE design pattern, implemented by Spring MVC
- The initial point of contact for handling a request.
- Manages the handling of the request:
  - invoking security services
    - authentication
    - authorization
  - delegating business processing
  - managing selection of an appropriate view
  - handling errors
  - managing selection of content creation strategies.



Spring MVC

Spring MVC helps in building flexible web applications.

The **model-view-controller** design pattern helps in separating the data logic, presentation logic, and business logic.

- **MODELS** - responsible for encapsulating the application data.
- **VIEWS** - render a response to the user with data from the model objects.
- **CONTROLLERS** - responsible for receiving the request from the user and calling the back-end services.

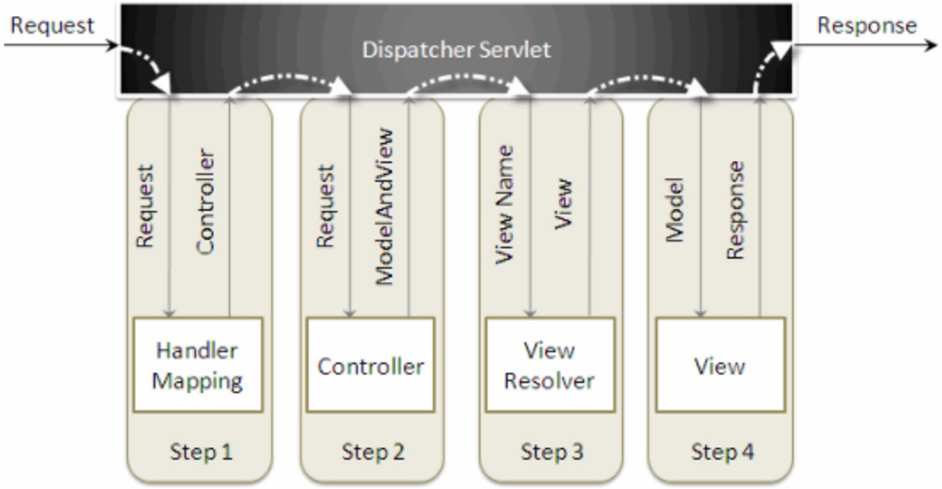
Spring MVC - How does it work?

When a request is sent to the Spring MVC framework the following sequence of events happen:

The **DispatcherServlet** receives the request.

1. The DispatcherServlet consults the **HandlerMapping** and invokes the proper **Controller**.
2. The Controller processes the request by calling the appropriate **service methods** and returns a **ModelAndView** object to the DispatcherServlet. The ModelAndView object contains the model data and the view name.
3. The DispatcherServlet sends the **view name** to a **ViewResolver** to find the actual **View** to invoke.
4. The DispatcherServlet will pass the **model object** to the **View** to render the result.

The View, with the help of the model data, will render the result back to the user.



Spring MVC Demo

REST

REST

Representational State Transfer is often referred to as a protocol which defines how applications communicate over the Hypertext Transfer Protocol (HTTP).

Applications that use REST are loosely-coupled and transfer information quickly and efficiently.

While REST doesn’t define data formats, it’s usually associated with exchanging JSON or XML documents, as a resource representation between a client and a server.

JSON vs XML

cognizant softvision

JSON vs XML

JSON	XML
JSON object has a type	XML data is typeless
JSON types: string, number, array, Boolean	All XML data should be string
Data is readily accessible as JSON objects	XML data needs to be parsed.
JSON is supported by most browsers	Cross-browser XML parsing can be tricky
JSON has no display capabilities	XML offers the capability to display data because it is a markup language.
It doesn't support comments	Retrieving value is difficult
	It supports comments.
	XML documents are relatively more difficult to read and interpret.
	JXML support various data types such as number, text, images, charts, graphs, etc.

JSON vs XML

```
{ "students": [
    {  "id":"01",
       "name": "Tom",
       "lastname": "Price"
    },
    {  "id":"02",
       "name": "Nick",
       "lastname": "Thameson"
    }
  ]
}
```

```
<?xml version="1.0" encoding="UTF-8"
?>
<students>
     <student>
               <id>01</id>
               <name>Tom</name>

     <lastname>Price</lastname>
     </student>
     <student>
               <id>02</id>
               <name>Nick</name>

     <lastname>Thameson</lastname>
     </student>
</root>
```

REST API

Spring MVC Annotations

cognizant softvision

Annotations

Component

@Component tells to Spring to automatically detect our custom beans.

During the first Spring release, all beans used to be declared in an XML file. For a large project, this quickly becomes a massive task, and Spring guys recognized the problem quickly.

In later versions (Spring 2.5), they provide **annotation-based dependency injection** and **Java-based configuration**.

Annotation-based dependency injection automatically scans and registers classes which is annotated using @Component annotation as Spring beans.

This means you don't declare that bean using the **<bean> tag** and inject the dependency, it will be done automatically by Spring. This functionality was enabled and disabled using **<context:component-scan> tag**.

Annotations

@Service, @Controller, and @Repository

Specialized annotations of @Component for certain situations.

By using that annotation we do two things: first, **we declare that this class is a Spring bean** and should be created and maintained by Spring ApplicationContext, but also we indicate **its role**.

This 2nd property is used by web-specific tools and functionalities. For example, DispatcherServlet will look for @RequestMapping on classes that are annotated using **@Controller but not with @Component**.

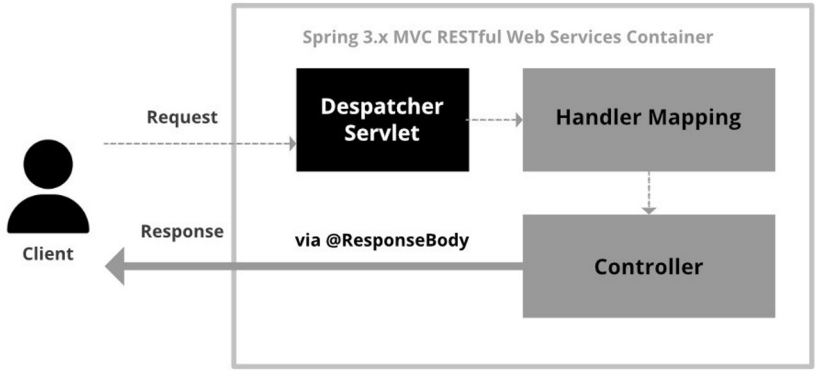
This means @Component and @Controller are the same **with respect to bean creation and dependency injection** but later is a specialized form of former. Even if you replace @Controller annotation with @Component, Spring can automatically detect and register the controller class but it may not work as you expect with respect to request mapping.

Annotations

@ResponseBody

Tells to the controller that the object returned is automatically serialized into JSON/XML and passed back into the HTTP Response object.

```
@ResponseBody
public ResponseEntity<User> getUsers() {
}
```



@RestController = @Controller + @ResponseBody

It is used to define the RESTful web services. It serves JSON, XML and custom response.

```
@RestController
public class UserController{}
```

Annotations

Request Mapping

@RequestMapping is used to define the Request URI to access the REST endpoints.

We can define Request method to consume and produce object. (@GetMapping, @PostMapping)

The default request method is GET.

```
@RequestMapping(value = "/user")
public ResponseEntity<User> getUsers() { }
```

Request Body

@RequestBody annotation is used to define the request body content type.

```
public ResponseEntity<User> createUser(@RequestBody User user) {}
```

Annotations

Path Variable

@PathVariable annotation is used to define the custom url: ex http://localhost:8080/api/users/111

```
@GetMapping("/api/users/{id}")
@ResponseBody
public String getUserById(@PathVariable String id) {
    return "ID: " + id;
}
```

Request Parameter

@RequestParam annotation is used to read the request parameters from the Request URL. ex http://localhost:8080/users?id=111

```
@GetMapping("/users")
@ResponseBody
public String getUserByIdUsingQueryParam(@RequestParam String id) {
    return "ID: " + id;
}
```

Annotations – Best Practices

@PathVariable or **@RequestParam?**

Exception Handling

EmployeeNotFoundException **is an exception used to indicate when an employee is looked up but not found.**

```
class EmployeeNotFoundException extends RuntimeException {
  EmployeeNotFoundException(Long id) {
    super("Could not find employee " + id);
  }
}
```

When an *EmployeeNotFoundException* is thrown, we need an extra bit of Spring MVC configuration to render a HTTP 404 response

Exception Handling

```
@ControllerAdvice
class EmployeeNotFoundAdvice {

  @ResponseBody
  @ExceptionHandler(EmployeeNotFoundException.class)
  @ResponseStatus(HttpStatus.NOT_FOUND)
  String employeeNotFoundHandler(EmployeeNotFoundException ex) {
    return ex.getMessage();
  }
}
```

- @ResponseBody signals that this advice is rendered straight into the response body.
- @ExceptionHandler configures the advice to only respond if an EmployeeNotFoundException is thrown.
- @ResponseStatus says to issue an HttpStatus.NOT_FOUND, i.e. an HTTP 404.

  The body of the advice generates the content. In this case, it gives the message of the exception.

Postman tool

Spring Rest API Demo

Time for practice

Exercises

1. Using the browser console find 5 different HTTP Status Codes in some responded on a website (or more websites)
2. Implement an Update and a Partial Update for a resource in your Spring application
3. Return all posts added by the current user.
4. Create a custom exception. Based on that exception return an appropriate HTTP response

Thank you

Sorin Miron

https://www.linkedin.com/in/sorinmiron

sorin.miron@cognizant.com

Alexandru Babei

alexandru.babei@cognizant.com