Software Engineering

SE for Web Applications I: Backend

Software & Systems Engineering | Prof. Dr. Andreas Vogelsang | 27.11.2023

@andivogelsang
vogelsang@cs.uni-koeln.de

Learning Goals for Today

- Know how the Internet works (on a very basic level)
- Know what HTTP is and how to request resources from servers
- Know architectures for web applications
- Know what a web server is and what backend abstractions it offers

SE for Web Applications – Why?

Web Applications

A **web application** (or web app) is application software that runs on a web server, unlike computer-based software programs that are run locally on the operating system (OS) of the device.

Web applications are accessed by the user through a **web browser** with an active network connection. These applications are programmed using a **client–server** modeled structure—the user ("client") is provided services through a server.

Where is the server?

- The server may be reached via the Internet, the intranet, or even on the same machine
- Web applications represent a general concept of system design – not just for the Internet.

Advantages of Web Applications

Runs in a browser

- Portability
- No local installation needed

Hosted on a server

- SW can be managed centrally (updates, maintenance, scalability)

Use of "web technology"

- Highly interoperable with other systems
- Based on well-defined standards

Disadvantages of Web Applications

Web technology originally made for static content → limited support for interactive applications

The Internet and HTTP

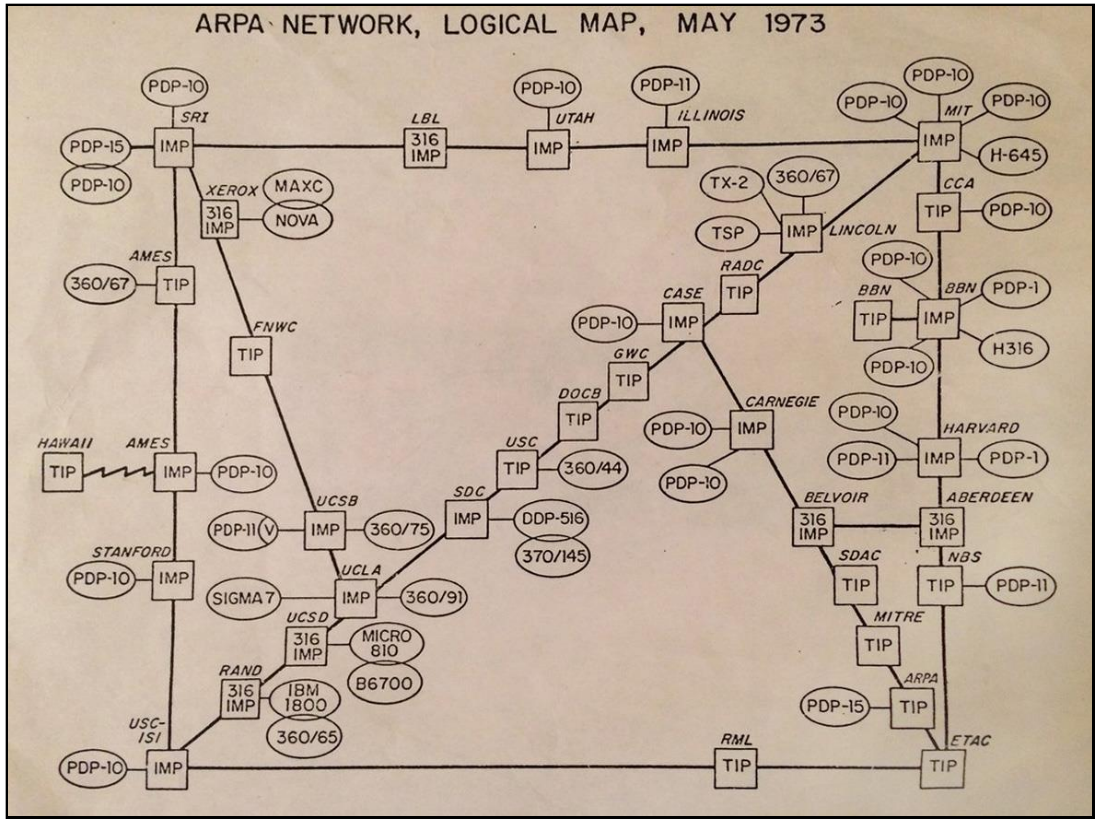
What is the Internet?

The Internet

The **Internet** is the global system of interconnected computer networks that uses the Internet protocol suite (TCP/IP) to link devices worldwide.

Important Concepts

- **TCP** (Transmission Control Protocol): connection-oriented protocol; Establishes a point-to-point connection between two entities in the network
- **IP** (Internet Protocol): principal communications protocol on the internet; Delivers packets of data across network boundaries
- **IP Address**: numerical label assigned to devices in a network that use the internet protocol to communicate with other devices



Universität zu Köln

High Level Web Overview

Servers

- Wait for requests
- They serve web **resources**

Client

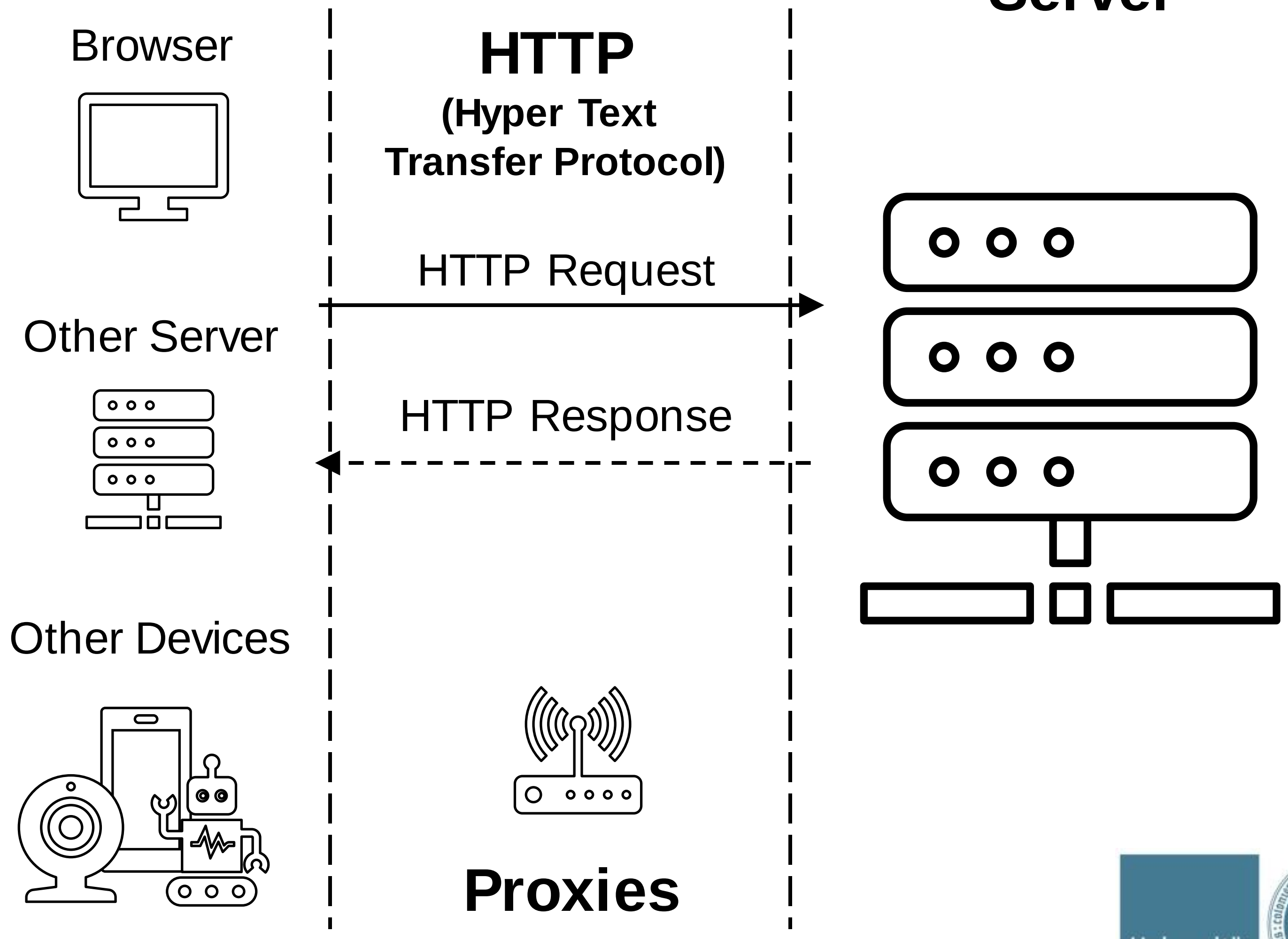
www.google.de → 142.250.179.131

Domain Name System (DNS)

Translating hostname to IP address

Multiple Proxies on the way

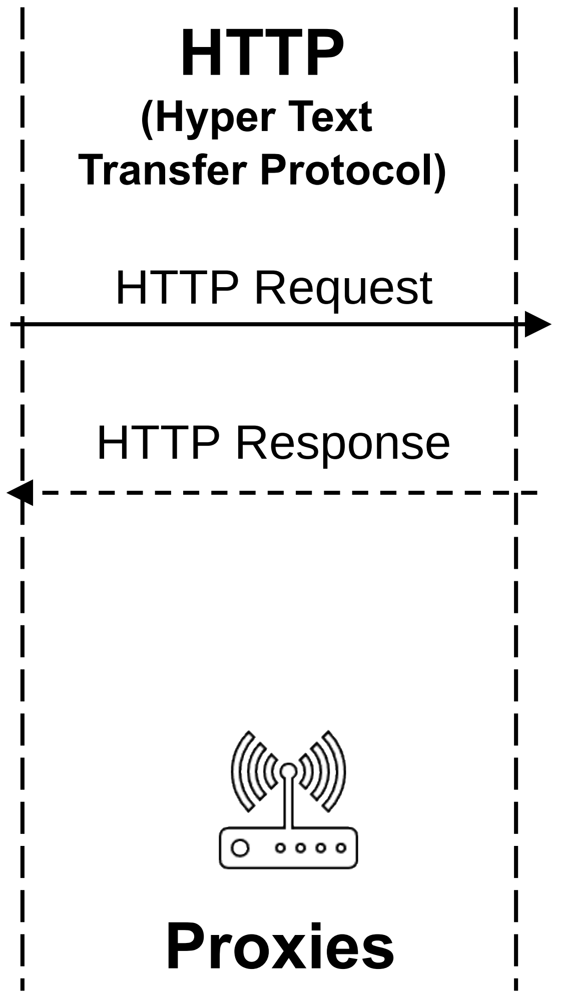
Try `traceroute www.google.de`



HTTP Overview

Hyper Text Transfer Protocol (HTTP)

- Builds upon TCP/IP
- Synchronous request-response protocol
  - Client (web browser) sends request
  - Web server replies with appropriate answer (could also be an error)
- “Stateless” protocol
  - Each request-response pair is independent
  - No permanent connection between server and browser (allows for a high number of users per server)
- Proxies mediate between browser and server (caching, filtering, etc.)
- In HTTP everything is sent and received as **clear text**
  - **Use HTTPS**: HTTP over a secured (TLS) connection



HTTP Resources and URL

HTTP resource

- Abstract concept for nodes in hypertext (e.g., HTML files, documents, images)
- Data types defined by MIME (RFC 2045): "text/html", "image/png", “application/xml“, etc.

Uniform Resource Locator (URL)

- Standardized way of identification and addressing of any resource on the internet
- Subtype of Uniform Resource Identifier (URI)

URL Syntax

<scheme>://[<user>[:<password>]@]<server>[:<port>]/[<path>][?<query>][#<fragment>]

https://usr:pwd@bookStore.de:3000/articles/books?year=1984#novels

URL Syntax

Scheme: Protocol to be used when connecting to a server: http(s), ftp, mongodb, etc.

User/Password: *Optional*: Credentials to access a protected resource

Server: Domain name or IP address of the server

URL Syntax

Port: Port at which the server is listening for requests

Path: Local path to resource on a server

Query: Parameters that can be passed to the sever

Fragment: Name of an entity within the resource. **This is only used by clients**

HTTP Request

HTTP Request

- Refers to a certain **resource** (identified by its URL)
- Contains a certain **type** (“method”): Most common methods for access: GET, POST, PUT
- Can contain request metadata (headers)
  - Target host, User authentication, Cookies, etc.
- Can contain application metadata, e.g.,
  - Preferred data type and language (for GET, POST)
  - Content Negotiation (e.g., HTML, JSON)
  - Data type of the body (for POST, PUT)
- Can contain application data (“body”), e.g., the data of a form (POST, PUT)

```
http://www.test101.com/doc/test.html?
bookID=1234&author=Roy+Fielding
```

```
GET /doc/test.html HTTP/1.1
Host: www.test101.com
Accept: image/gif, image/jpg, */*
Accept-Language: en-us
Accept-encoding: gzip, deflate
User-Agent: Mozilla/4.0
Content-Length: 35

bookID=1234&author=Roy+Fielding
```

HTTP Request Method

HTTP Request Method

- Each access to a resource has a certain request type ("method")
- **GET**: request a resource, only retrieves data
- **POST**: submit data to a resource
  - Data is included in body of the request
  - May result in creation of new resource or update of existing resource
- **PUT**: replaces target resource with sent payload
- **DELETE**: delete a resource
- **PATCH**: provides a set of instructions to modify the target resource
- **OPTIONS, TRACE, HEAD, CONNECT**: access to the metadata of the servers, the Internet connection, the resource, etc.

Request Method Properties

- GET: Safe and Repeatable (expect no side effects)
- Expect side effects for POST, PUT, DELETE, PATCH
- PUT, DELETE: Idempotent (expect same effect even with multiple executions)

HTTP Response

HTTP Response

- Always follows a request message
- Contains a status code
- Can contain response metadata, e.g.:
  - Server, TCP connection state, date
- Can contain application metadata, e.g.:
  - Data type and encoding of the application data
  - Caching possibilities and expiring date
  - Current URL of a transferred resource (for GET)
- Can contain application data (“body”)

```
HTTP/1.1 200 OK
Date: Sun, 08 Feb xxxx 01:12:12 GMT
Server: Apache/1.3.29 (Win32)
Last-Modified: Sat, 07 Feb xxxx
Content-Type: text/html
Content-Length: 35

<h1> My Home page</h1>
```

Response Status Codes

- **1xx**: Informational (e.g., 101 Switching)
- **2xx: Success (e.g., 200 OK)**
- **3xx**: Redirect (e.g., 301 Permanent)
- **4xx: Client Error (e.g., 404 Not Found)**
- **5xx: Server Error (e.g., 500 Internal Server Error)**

RESTful APIs

From HTTP to REST

- HTTP-like communication became so popular that a new API style emerged from it: **RESTful APIs**
- API (application programming interface): An API is an interface for Machine-to-Machine communication
- RESTful APIs are frequently used when independent applications need to communicate with each other (not only web clients with web servers)

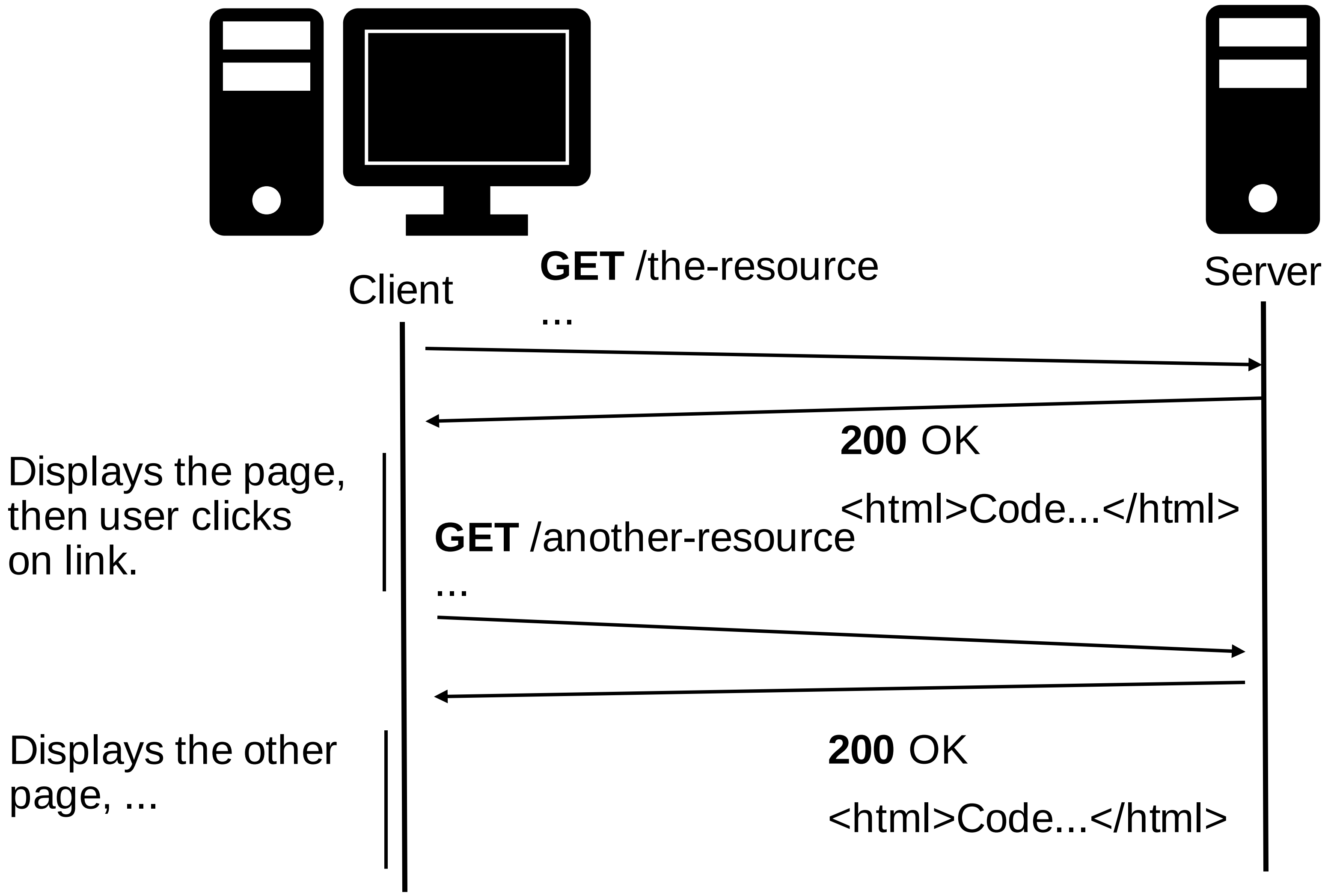
RESTful APIs

- Everything a server has to offer is a **resource**
- Resources are identified by global IDs (URIs)
- A client can request or change a resource
- The HTTP standard methods (GET, POST,…) are used as the uniform interface for all API calls.

6 Constraints of RESTful APIs

- **Client-server architecture:** RESTful systems separate processing and storing data (server) from collecting, requesting, and presenting the data (client)
- **Statelessness**: Responsibility for managing state (e.g., logged in or not) is on the client
- **Cacheability**: Data may be temporarily stored by client and server to avoid redundant requests.
- **Layered System**: A client cannot tell whether it is connected directly to an end server or to an intermediary.
- **Code on demand (optional):** A sever can temporarily extend the functionality of a client by transferring executable code
- **Uniform interface:** resource identification and manipulation, self-descriptive messages

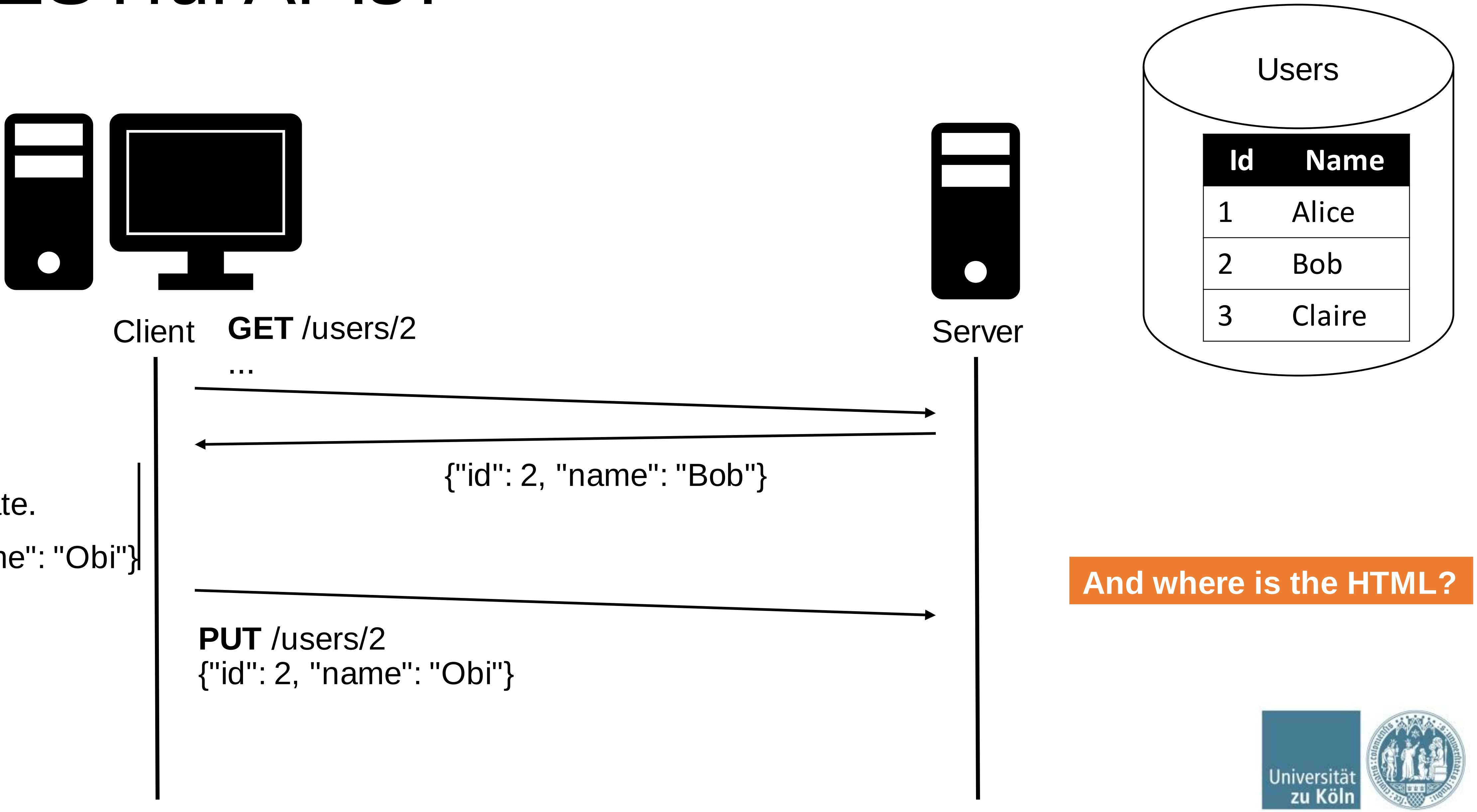
Traditional Web Applications



Drawbacks

- The client must understand both HTTP and HTML.
  - E.g., smartphone apps do not use HTML
- The entire webpage is replaced with another one.
  - No way to animate transitions between webpages.
- Same data is usually sent in multiple responses.
  - E.g., HTML code for the layout

RESTful APIs?



Id	Name
1	Alice
2	Bob
3	Claire

GET /users/2
...

{"id": 2, "name": "Bob"}

Changes state.

{"id": 2, "name": "Obi"}

PUT /users/2
{"id": 2, "name": "Obi"}

And where is the HTML?

Web Server

Web Server

Web Server

Ambiguous term:

1. **Hardware**: A computer (“server”) connected to the internet (or any network)
2. **Software**: A program running on a computer/server that accepts HTTP requests over a specific port and answers with HTTP responses

Web Server – Hardware

- Properties of contemporary web servers
  - Part of large data centers
  - Latency is geographically dependent, so web servers are often geographically distributed (works through, e.g., DNS)
  - Virtual servers: Physical servers can host many virtualized (web) servers
- Can also be your own computer (**localhost**)

Web Server – Software (HTTP Server)

- Makes resources accessible over a URL and HTTP/S (standard ports 80 and 433)
- Starting a web server on your local computer makes it accessible over http://localhost or http://127.0.0.1
- Maps **path component** of URL to
  - static asset on the file server
  - dynamically rendered resources
- Often incorporates some functionality for caching and session handling

niversität zu Köln

Web Server – Static Assets

Serving Static Assets from the File System

- Web server automatically wraps static files with HTTP Response Headers
- Static assets directly map URL path to relative part of the file system
  - They cannot react to other part of the request (e.g., query parameter)
- MIME-Type is inferred through heuristics (e.g., file endings)
- Example of common static files in web servers
  - HTML, CSS
  - JavaScript (for use in browser)
  - Media (Images, Video, Audio, etc.)

Example

- Static assets made available at path
  `/var/www/public_html`
  on the server

- If we determine [this is configurable]
  `http://localhost/static/js/search.js`
  to be a request for static assets, we could return
  `/var/www/public_html/js/search.js`

Web Server – Dynamic Resources

Dynamic Resources

- Executing programs in a server-side programming language on the server
- Dynamic resources can react to complete HTTP request (including header information)
  - Path and Query Parameters
  - HTTP Method (GET, POST, PUT, …)
  - Content Negotiation (Accept: application/json)
  - …
- System output is treated as the complete HTTP response (including headers)
- However, many programming languages offer library support for basic HTTP related functions and provide abstractions (e.g., for dealing with response headers)

Web Server Examples

- Apache (httpd): one of the first and most popular web servers
- NginX: Similarly popular as Apache; better performance but less flexible
- Node.js: WebServer based on JavaScript
- Apache Tomcat: Specifically for Java backends

Web Application Architectures

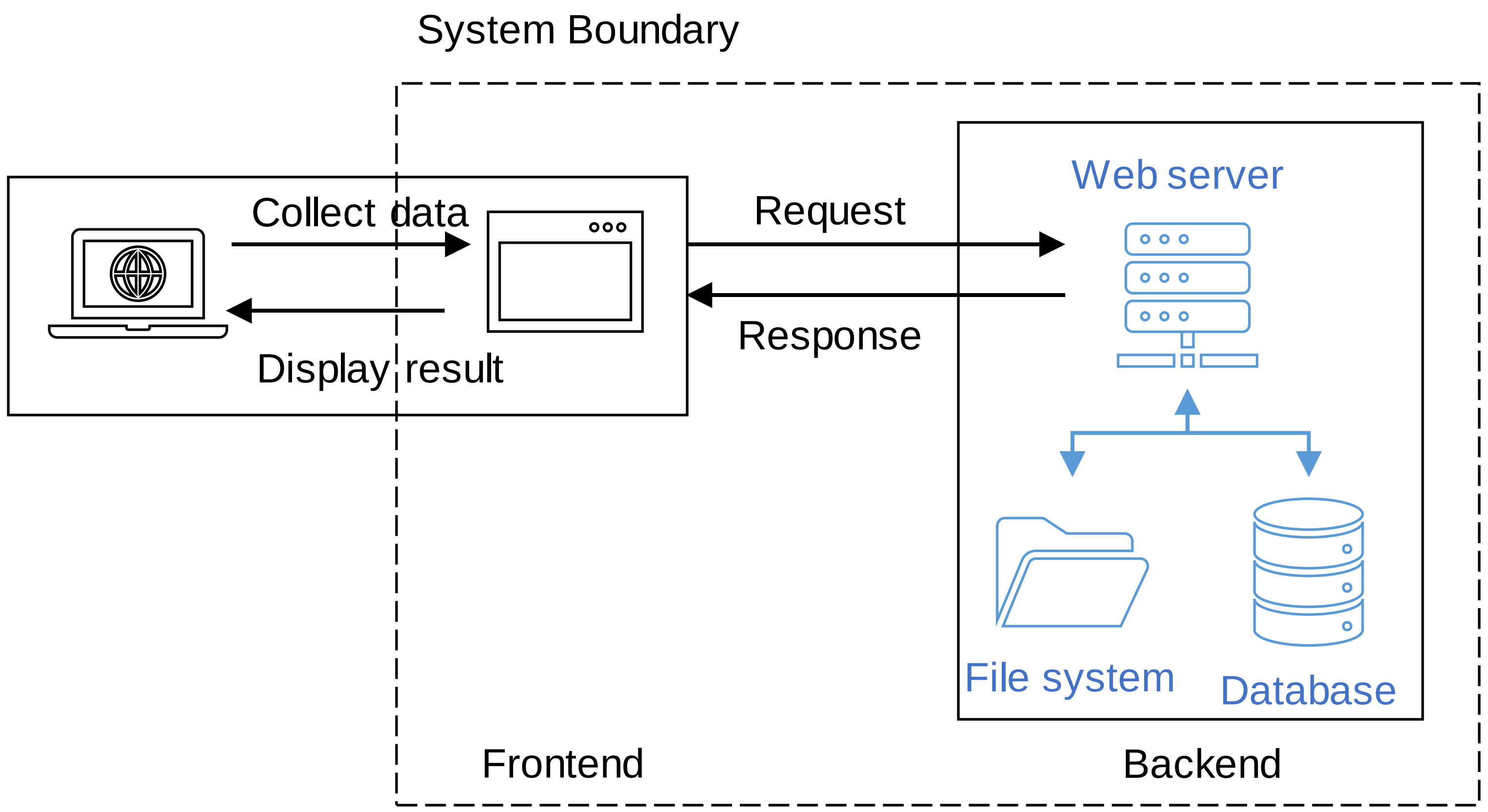
Web App Architectures – Physical View

Frontend

Client-side aka. the frontend: Code is typically written in HTML, CSS, JavaScript and stored within the browser. It's where user interaction takes place.

Backend

Server-side, also known as the backend: Controls the business logic and responds to HTTP requests. The server-side code is written in Java, PHP, Ruby, Python, etc.



Web App Architectures – Variants

Monolithic Architectures

One Web Server, One Database

+ Easy
+ Cheap

— Single point of failure
— Not easily scalable

Multiple Web Servers, One Database

+ Reliable
+ Stateless Server

— DB is single point of failure

Multiple Web Servers, Multiple Databases

+ Reliable
+ Scalable

— Complex
— Data distributed

Service-based Architectures

Microservices

+ Scalable
+ Easy to distribute

— Dependency hell

Serverless

+ No need to maintain servers
+ Scalable

— Hard to understand, maintain, and test

Web App Architectures – Logical View

Presentation

Code that transforms data into views.

Logic

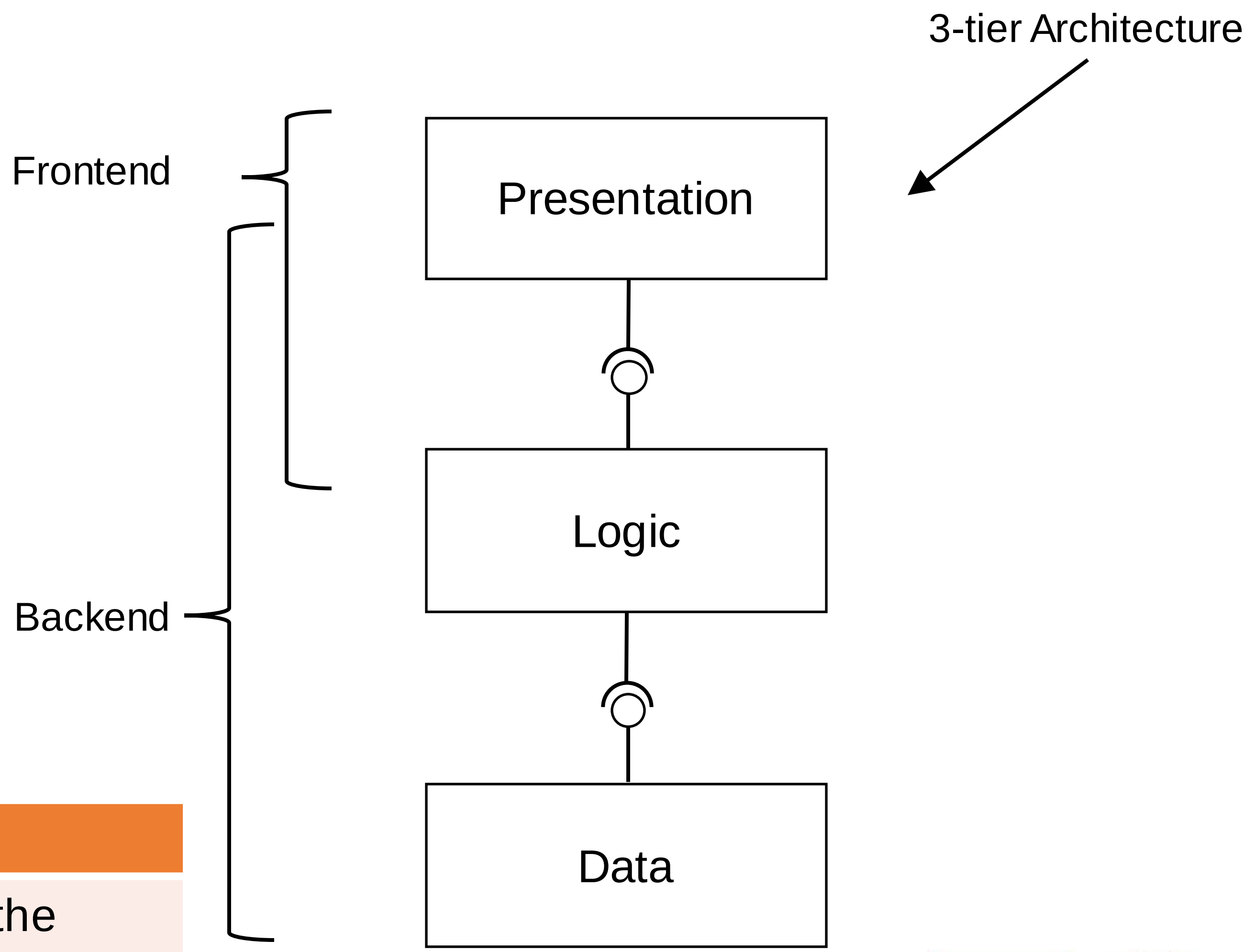
Code that handles requests by potentially invoking additional *services.*

Data

Code that specifies and manages application data.

Rich-Client vs. Thin Client

Rich-Client: Parts of the logic are implemented in the frontend (e.g., data preprocessing or data validation)
Thin-Client: Parts of the presentation are implemented in the backend (e.g., generation of client-specific views)



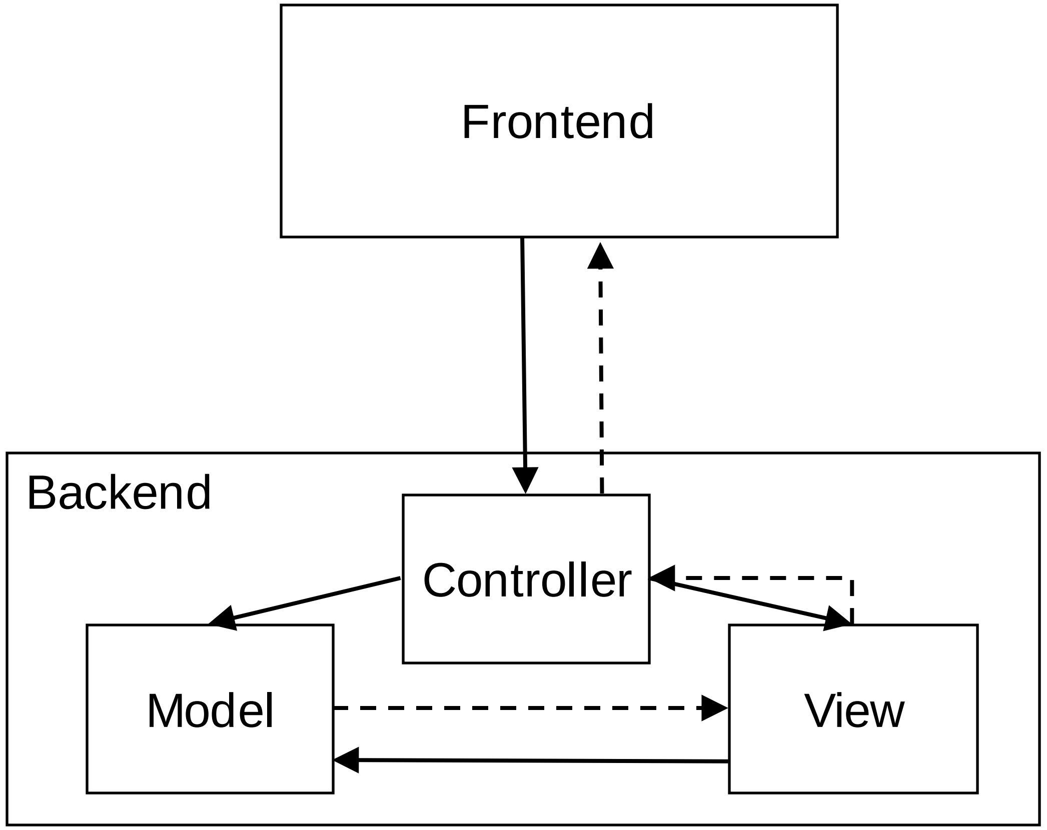
Backend Architectures – Logical View

MVC or what?

The MVC pattern in the backend is well suited for **thin client** applications where a lot of the presentation (view) is implemented in the backend.

For **rich client** applications, the view in the backend may be very simple (e.g., just transforming data into a JSON object)

Model-View Controller Architecture within the backend



Backend Frameworks

Backend Framework Abstractions

Web (HTTP Abstractions)

HTTP Request

Cookies and Sessions

HTTP Response

- Routes (URL mapping)
- Path Parameters
- Message Body
- Request Method
- Query Parameters
- Content-Negotiation

- Status Code
- Content Type (MIME Type)
- Encoding (Compression)
- Dynamic Content/ Response
- Caching Behavior
- Static Files

Architecture (Code Organization)

- Modularization
- Layered Architectures
- Model-View-Controller

Standard Utilities

- Networking
- File System
- Database Access

General Purpose Abstraction

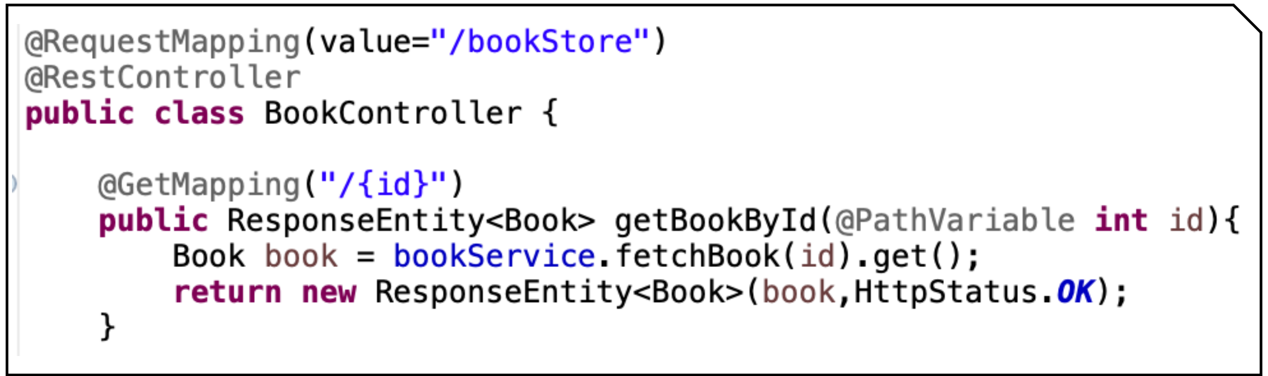
- Content Templating
- Environment Variables (Secret Management)

Backend FW Abstractions – HTTP Req/Res

http://www.test101.com/bookStore/1

```
GET /bookStore/1 HTTP/1.1
Host: www.test101.com
```

Java/Spring



```java
@RequestMapping(value="/bookStore")
@RestController
public class BookController {

    @GetMapping("/{id}")
    public ResponseEntity<Book> getBookById(@PathVariable int id){
        Book book = bookService.fetchBook(id).get();
        return new ResponseEntity<Book>(book,HttpStatus.OK);
    }
```

Node.js/Express

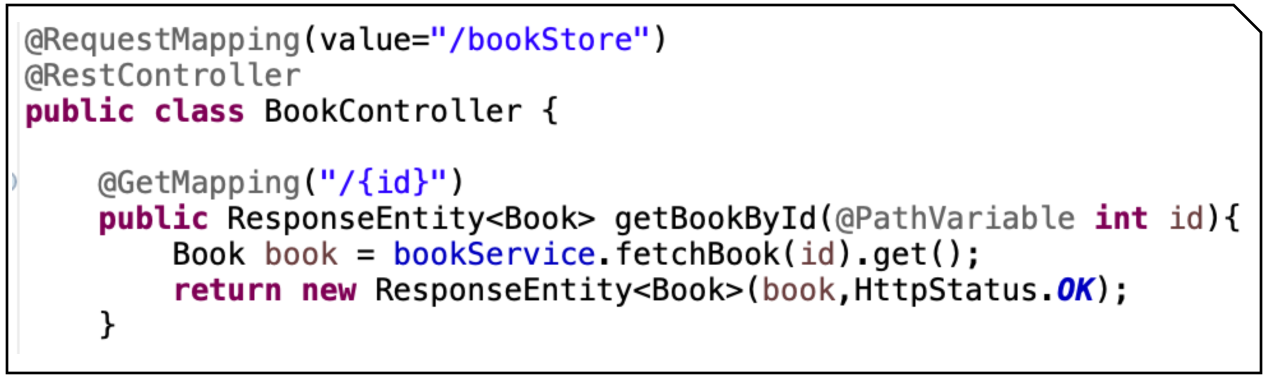
```
app.get('/:id', (req, res)=>{
  const bookId = req.params.id;
  const book = Books.find(bookID);
  res.status(200);
  res.send(book);
});
```

```
HTTP/1.1 200 OK
Date: Sun, 08 Feb xxxx 01:12:12 GMT
Content-Type: application/json

{"name": "Book1","author": "Albert Camus","year": 1900,"price": 20.95}
```

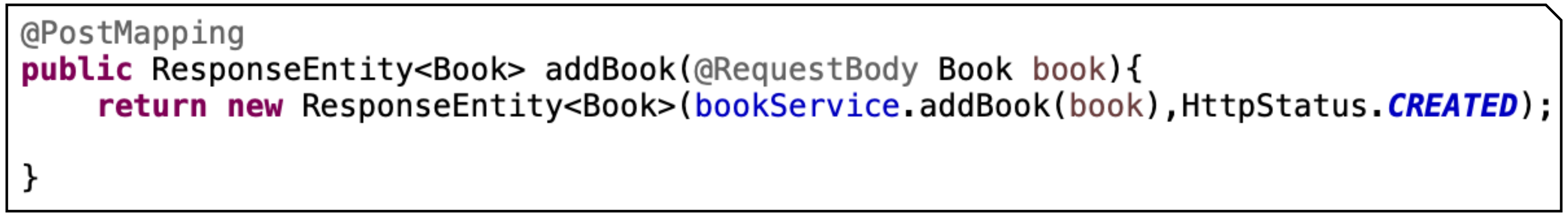
Backend FW Abstractions – HTTP Req/Res

Java/Spring



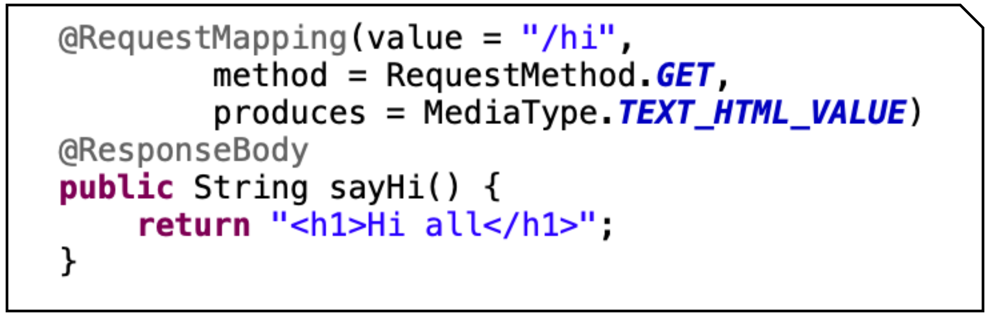
```
@RequestMapping(value="/bookStore")
@RestController
public class BookController {

    @GetMapping("/{id}")
    public ResponseEntity<Book> getBookById(@PathVariable int id){
        Book book = bookService.fetchBook(id).get();
        return new ResponseEntity<Book>(book,HttpStatus.OK);
    }
```



```
@PostMapping
public ResponseEntity<Book> addBook(@RequestBody Book book){
    return new ResponseEntity<Book>(bookService.addBook(book),HttpStatus.CREATED);

}
```



```
  @RequestMapping(value = "/hi",
          method = RequestMethod.GET,
          produces = MediaType.TEXT_HTML_VALUE)
  @ResponseBody
  public String sayHi() {
      return "<h1>Hi all</h1>";
  }
```

Backend FW Abstractions – Templating

Templating

Templates (sometimes also called views) provide separation between program logic and output.

Template engines replace variables in static template files and control structures (conditionals and loops) with values passed from the program.

Java/Spring/FreeMarker

`index.ftl`

```
<div>
  <table class="datatable">
    <tr>
      <th>Company</th>
      <th>Type</th>
    </tr>
    <#list model["carList"] as car>
      <tr>
        <td>${car.company}</td>
        <td>${car.type}</td>
      </tr>
    </#list>
  </table>
</div>
```

+

```
@RequestMapping(value = "/cars", method = RequestMethod.GET)
public String init(@ModelAttribute("model") ModelMap model) {
    model.addAttribute("carList", carList);
    return "index";
}
```

=

Company	Type
VW	Golf
Daimler	C-Class

Summary

SE for Web Applications I

- Web applications are an interoperable, easily manageable way to develop software applications
- Web applications rely on web technology (esp. HTTP)
- Client-Server Architecture
- Backend captures business logic, data handling, and sometimes also views.