UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Software Entwicklung & Programmierung

Client-Server-Architektur

PALUNO
The Ruhr Institute for Software Technology

Disclaimer

Bilder und Texte der Veranstaltungsfolien und -unterlagen sowie das gesprochene Wort innerhalb der Veranstaltung und Lehr-Lern-Videos dienen allein dem Selbst- bzw. Gruppenstudium. Jede weiterführende Nutzung ist den Teilnehmenden der Moodle-Kurse untersagt, z.B. Verbreitung an andere Studierende, in sozialen Netzwerken, dem Internet!

Darüber hinaus ist ein studentischer Mitschnitt von Webkonferenzen im Rahmen der Lehre nicht erlaubt.

Zielsetzung

Am Ende dieser Präsentation könnt Ihr:

- Die **Kernkonzepte** einer **Client-Server-Architektur** erläutern
- Die **Grundlagen** von **Netzwerkkommunikation** erläutern
- Eine erste eigene **Implementierung** einer simplen **Client-Server-Struktur** in Java vornehmen

Agenda

1. **Motivation**
2. Grundlagen Client-Server Architektur
3. Kommunikation über Sockets
4. Request/Response am Beispiel von HTTP
5. Code-Beispiele

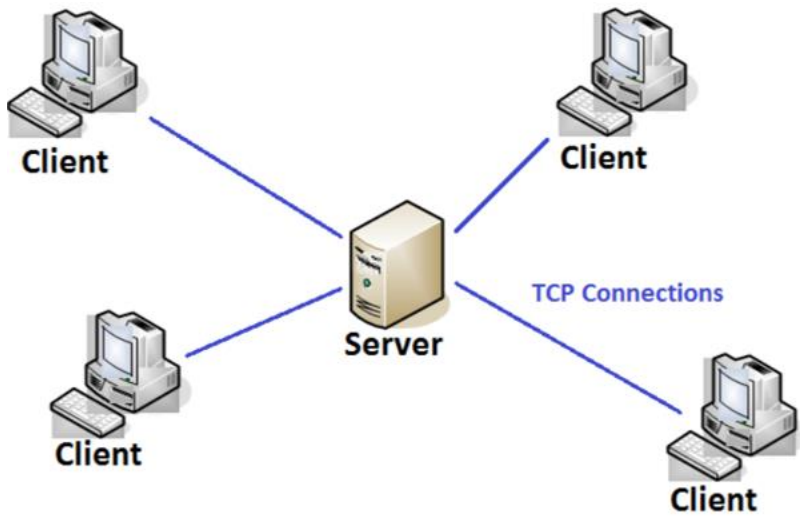
Motivation

- In der heutigen Zeit der Informationstechnologie spielt die Beherrschung von Komplexität eine kritische Rolle
  - Client-Server-Architektur hilft dabei durch die Zentralisierung von Funktionalitäten
- Dabei haben Client-Server-Architekturen einige Vorzüge
  - Durch zentralisierte Zugriffskontrolle kann effektiv beeinflusst werden, wer Zugriff auf was hat (**Sicherheit**)
  - Dadurch, dass Client-Server-Architekturen verteilte Verantwortlichkeiten zwischen unabhängigen Computern (Clienten) darstellen, können diese einfacher instand gehalten werden (Clients sind unbetroffen von Änderungen (**Verkapselung**)

Agenda

1. Motivation
2. **Grundlagen Client-Server Architektur**
3. Kommunikation über Sockets
4. Request/Response am Beispiel von HTTP
5. Code-Beispiele

Grundlagen Client-Server-Architektur



- Die Client-Server-Architektur zeichnet sich unter anderem durch horizontale und vertikale Skalierbarkeit aus
  - Erhöhung der Anzahl der Clients
  - Integration eines leistungsstärkeren Servers

- Die Client-Server-Architektur beschreibt ein Modell, in dem der Server Ressourcen verwaltet, verteilt und hosted, um diese den Clients bereitstellen zu können
  - Dabei können Server und Client auf einem Gerät laufen, aber auch über ein Netzwerk kommunizieren
  - Server können beispielsweise prüfen, ob ein Client überhaupt die nötigen Berechtigungen hat, um Daten abzufragen

Grundlagen Client-Server-Architektur: Client

- Definition: Ein Client ist ein Programm, welches Dienste eines Servers in Anspruch nimmt.
  - Der Client ist das Programm, mit dem die Nutzer direkt interagieren
  - Der Client muss Daten visualisieren und den Nutzern die Möglichkeit bieten, Daten zu bearbeiten
  - Der Client ist somit die Schnittstelle für den Nutzer, um mit dem Server zu interagieren.

Grundlagen Client-Server-Architektur: Server

- Definition: Ein Server ist ein Programm, welches Funktionalitäten für andere Programme (Clients) bereitstellt
  - Dabei gibt es verschiedene Arten von Servern, die unterschiedliche Funktionalitäten zur Verfügung stellen (Web-Server, File-Server, Print-Server etc.)
- Der Server hat die Aufgabe, Daten zentral zu verwalten
- Die meisten Server arbeiten nach dem Prinzip der Request-Response-Kommunikation
  - Das bedeutet, dass der Server Anfragen von Clients erhält, diese prüft und dann entsprechend antwortet
  - Diese Anfragen können beispielsweise das Abfragen oder Ändern von Daten auf dem Server sein

Agenda

1. Motivation
2. Grundlagen Client-Server Architektur
3. **Kommunikation über Sockets**
4. Request/Response am Beispiel von HTTP
5. Code-Beispiele

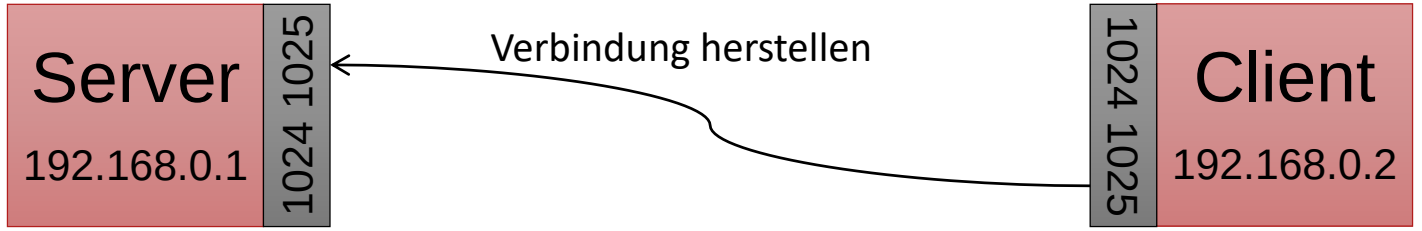
TCP-IP Referenzmodell

- Gliederung der Internetprotokollfamilie
- Top-Down: Ein Anwendungsprotokoll greift auf Protokolle der darunterliegenden Schichten zu, usw.

Schicht 5-7	Anwendungen	Anwendungsspezifische Protokolle wie z.B. HTTP für Web Requests oder FTP für Datenübertragung auf Dateiebene
Schicht 4	Transport	Protokolle für grundlegende Verbindungen (Ende-zu-Ende-Kommunikation)
Schicht 3	Internet	Protokolle für Identifizierung von Netzknoten (IP-Adresse) und Wegefindung (Routing)
Schicht 1-2	Netzzugang	Netzzugang, d.h. physikalische Verbindung und Sicherstellung der fehlerfreien Datenübertragung auf Byte-Ebene

Was ist ein Socket?

- Bei Verbindungen mittels Transport-Control-Protocol (TCP) werden die Endpunkte als sogenannte Sockets bezeichnet
- Jeder Socket besteht aus einer IP-Adresse und einem Port
- In Bezug auf das TCP/IP-Referenzmodell bedeutet dies, dass TCP auf IP basiert, welches für die Kennzeichnung der Endgeräte im Netz zuständig ist (IP-Adresse)
- Der Port kann frei gewählt werden, wird jedoch meist durch das Anwendungsprotokoll vorgegeben (vgl. well-known ports wie z.B. 80 für HTTP)
- Jede **TCP Verbindung** wird durch zwei Sockets gekennzeichnet



Sockets in Java

- Java stellt Klassen zur Socket-Verbindung im Package java.net zur Verfügung
- Die Klasse Socket implementiert eine Seite der bidirektionalen Verbindung
- Die Klasse ServerSocket implementiert einen Socket der auf Verbindungsanfragen wartet („listen“) und Verbindungen zu Clients akzeptiert
- **Vorgehensweise**:
  - Socket öffnen
  - Input-/Outputstream zum Socket öffnen
  - Lesen von dem bzw. Schreiben auf den Stream
  - Streams schließen
  - Socket schließen

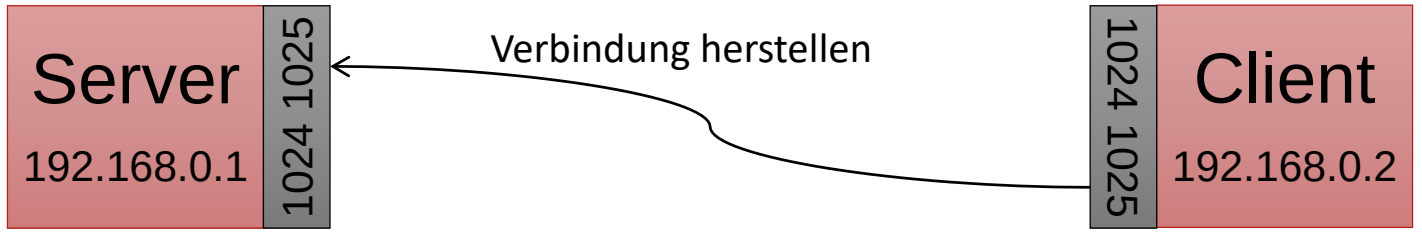
Arbeiten mit Sockets

- Sind die Sockets miteinander verbunden, können sie via Streams kommunizieren

- Jeder Socket besitzt zwei Streams
  - `InputStream getInputStream() throws IOException`
  - `OutputStream getOutputStream() throws IOException`
- Streams können je nach Verwendung aufgewertet werden
  - `InputStream z.B. zu BufferedReader oder Scanner`
  - `OutputStream z.B. zu DataOutputStream oder PrintWriter`

- Nach Abschluss der Arbeiten sollten Streams und Sockets wieder geschlossen werden
  - `InputStream.close()`
  - `OutputStream.close()`
  - `Socket.close()`

Beispiel: Verbindungsaufbau



Server Code:

```
ServerSocket serverSocket = new ServerSocket(1025);
Socket socket = serverSocket.accept();
serverSocket.close();
//-Hier kann die Verbindung genutzt werden-
socket.close();
```

Client Code:

```
Socket socket = new Socket("192.168.0.1", 1025);
//-Hier kann die Verbindung genutzt werden-
socket.close();
```

Beispiel: Datenübertragung

- Daten über einen Socket-Verbindung senden:

```
//-Eine Socket-Verbindung wurde aufgebaut-
OutputStream os = socket.getOutputStream();
os.write(42);
PrintWriter pw = new PrintWriter(os);
pw.println("Paluno – The Ruhr Institute for Software Technology");
```

- Daten aus einer Socket-Verbindung lesen:

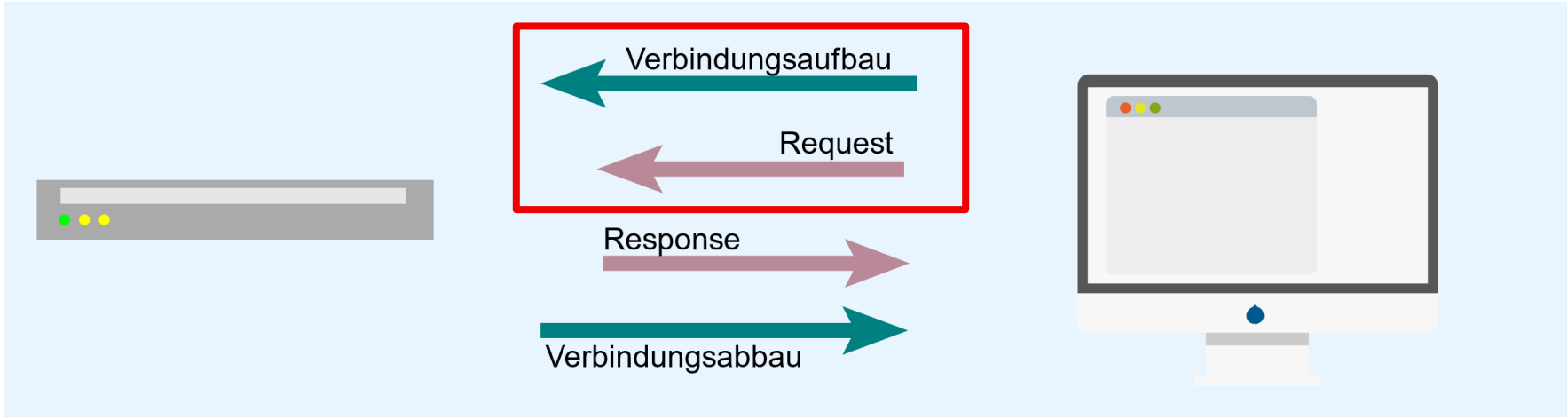
```
//-Eine Socket-Verbindung wurde aufgebaut-
InputStream is = socket.getInputStream();
System.out.println( is.read() );
BufferedReader br = new BufferedReader( new InputStreamReader(is) );
System.out.println( br.readLine() );
```

Agenda

1. Motivation
2. Grundlagen Client-Server Architektur
3. Kommunikation über Sockets
4. **Request/Response am Beispiel von HTTP**
5. Code-Beispiele

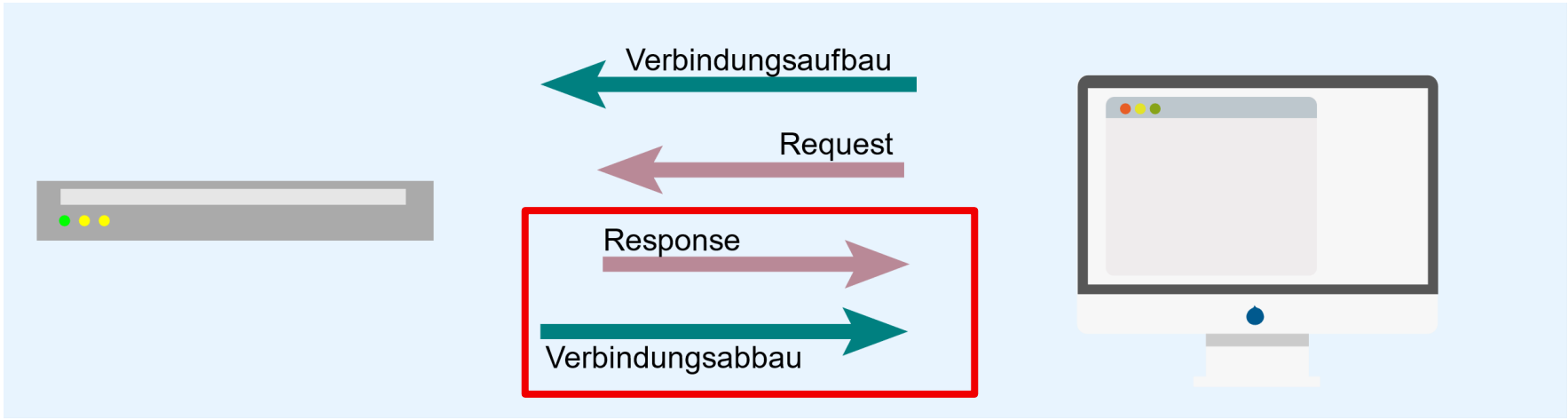
Request/Response am Beispiel von HTTP

- HTTP: Hyper Text Transfer Protocol
  - Protokoll für Datenkommunikation im Internet
  - Basiert auf TCP
- Im Fall des Aufrufs einer Internetseite stellt der Browser den HTTP-Client dar, welcher mittels URL-Aufrufs einen HTTP-Request an den Server sendet (s. Abbildung)
  - Dabei gibt es verschiedene Arten von Requests, die der Client stellen kann:
    - GET: Wird zum abrufen von Daten des Servers verwendet
    - POST: Sendet Daten zum Server (z.B.: ein Dateiupload)



Request/Response am Beispiel von HTTP

- Anschließend sendet der Server einen Response basierend auf dem Request des Clients
  - Dieser Response enthält unter anderem einen Status Code, welcher aus 3 Zahlen besteht
  - Die erste Zahl beschreibt dabei die Klassifizierung des Response
    - 2xx: Aktion war erfolgreich
    - 4xx: Client Error: kann z.B. durch falsche Syntax ausgelöst werden
- War der Request erfolgreich, werden neben dem Status Code auch Response Header, beispielsweise in Form von html Code mitgeschickt
  - Diese kann der Client-Browser nun visuell für den Benutzer darstellen



Agenda

1. Motivation
2. Grundlagen Client-Server Architektur
3. Kommunikation über Sockets
4. Request/Response am Beispiel von HTTP
5. **Code-Beispiele**

Code-Beispiel mit Jetty

- Jetty stellt einen Webserver und einen Servlet Container zu Verfügung
- Wie im folgenden Beispiel zu sehen, ist dieser sehr einfach zum Testen zu implementieren und später auch zu integrieren
- Der angegebene Link, dient als Hilfestellung zur Erstellung eines ersten Testprogramms
- Zusätzlich kann der hier angegebene Code zur Hilfestellung benutzt werden (Hier benutzte HTTP-Instanziierungen stehen erst ab Java 11 zur Verfügung!)

```
public class Client {

    public static void main(String[] args) throws IOException, InterruptedException {
        HttpClient client = HttpClient.newHttpClient();
        HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create("http://localhost:3001/tractor"))
                .POST(BodyPublishers.ofString("It's not much, but it's honest work."))
                .build();

        HttpResponse<String> response = client.send(request,
                HttpResponse.BodyHandlers.ofString());

        System.out.println(response.body());
    }

}
```

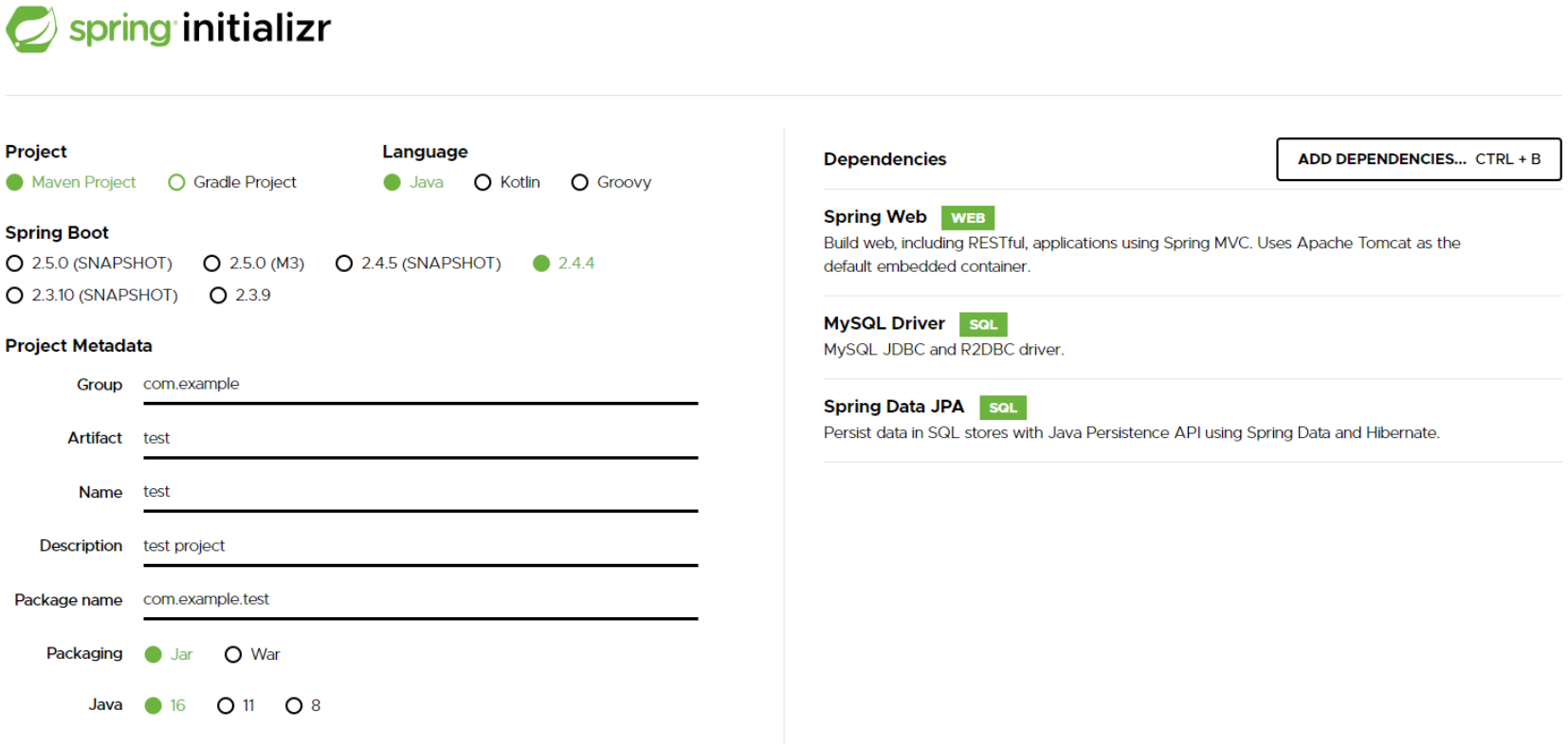
https://www.vogella.com/tutorials/Jetty/article.html

Code-Beispiel mit Jetty

```java
public static void main(String[] args) throws Exception {
    int port = 3001;
    Server server = new Server(port);
    server.setHandler(new AbstractHandler() {
        @Override
        public void handle(String target,
                           Request baseRequest,
                           HttpServletRequest request,
                           HttpServletResponse response) throws IOException, ServletException
        {
            if(baseRequest.getMethod().equals("POST")) {

                // Declare response encoding and types
                response.setContentType("text/text; charset=utf-8");

                // Declare response status code
                response.setStatus(HttpServletResponse.SC_OK);

                // Write back response
                response.getWriter().println("Request sent to: " + target + "\nHere is our meme wisdom:\n" + request.getReader().readLine());

                // Inform jetty that this request has now been handled
                baseRequest.setHandled(true);
            }
        }
    });
    server.start();
    server.join();
}
```

Beispiel mit Spring Boot

- Spring Boot erleichtert Entwicklern die Konfiguration von Spring-Anwendungen
- Entwickler können sich mithilfe des spring initializrs ein sofort lauffähiges Projekt mit den benötigten Dependencies erstellen lassen



https://spring.io/
https://start.spring.io/
https://spring.io/quickstart

Quellen

- https://de.wikipedia.org/wiki/Server
- https://www.exone.de/ratgeber/der-server-einfach-erklaert/
- https://cio-wiki.org/wiki/Client_Server_Architecture
- https://apachebooster.com/blog/what-is-client-server-architecture-and-what-are-its-types/
- https://www.vogella.com/tutorials/Jetty/article.html
- https://www.tutorialspoint.com/http/http_overview.htm
- https://www.mediaevent.de/tutorial/http-request.html

Verwendete Grafiken

- Grafiken von
  - https://www.draw.io/
  - https://cio-wiki.org/wiki/File:ClientServerArchitecture1.png
  - https://www.mediaevent.de/tutorial/svg/http-request-ablauf.svg

UNIVERSITÄT DUISBURG ESSEN

Offen im Denken

Vielen Dank für Eure Aufmerksamkeit

PALUNO
The Ruhr Institute for Software Technology