



HHZ Sommersemester 2025

HHZ Bootcamp  
Java

# Setup Jupyter Notebook

Component	Command
Jupyter Notebook	<code>pip install notebook</code>
Ipython-sql	<code>pip install ipython-sql</code>
iJava	<p>Goto: <a href="https://github.com/SpencerPark/IJava/releases">https://github.com/SpencerPark/IJava/releases</a></p> <p>Download : iJava-1.x.0.zip (Pre-built binary) Unpack Run <code>python install.py --sys-prefix</code></p>

Verification:

```
C:\HHZ\Java - Code Camp\temp>jupyter kernelspec list
```

Available kernels:

```
java
```

```
C:\Python313\share\jupyter\kernels\java
```

```
python3
```

```
C:\Python313\share\jupyter\kernels\python3
```



# Aufteilung der 3 Boot-Camp Tage

Tag #	Inhalt
1	<ul style="list-style-type: none"><li>• Git Rehash</li><li>• Java Besonderheiten und Code testen</li><li>• SQL (Fokus SELECT Befehl)</li><li>• JDBC</li><li>• ORM</li><li>• Projektarbeit</li></ul>
2	<ul style="list-style-type: none"><li>• Rest Web Services in Java erstellen</li><li>• Projektarbeit</li></ul>
3	<ul style="list-style-type: none"><li>• HTML5</li><li>• Projektarbeit einschließlich Integration der erarbeiteten Komponenten</li></ul>



The background of the slide is a composite image. On the left, a portion of a laptop screen is visible, displaying a complex architectural drawing with scaffolding. Below the screen, a hand is shown typing on a silver laptop keyboard. The hand is wearing a black and gold watch. To the right of the laptop, there is a faint, semi-transparent image of a hand holding a pen, poised to write on a document. The overall image has a soft, white overlay that makes the text stand out.

# Exkurs: Git Rehash

Siehe GIT-Foliensatz



# Auswahl an Aufgaben

Nr.	Aufgabe	Kurzbeschreibung
1	Aufgaben-Manager (To-Do-Liste)	Nutzer können Aufgaben erstellen, bearbeiten, löschen und filtern
2	Online-Umfrage	Nutzer können an einer Umfrage teilnehmen, Ergebnisse werden angezeigt
3	Produktverwaltung (Bestandsanzeige)	Produkte anlegen und Lagerbestand anzeigen/bearbeiten
4	Bücherregal ("Digital Library")	Bücher speichern, bewerten und anzeigen, evtl. mit Sterne-System
5	GitHub als Datenquelle (z. B. Profil-Analyzer)	GitHub-Daten abrufen und anzeigen, z. B. Repos, Commits, Sprachen



# REST WEBSERVICE API



# What is a REST API?

---

- REST stands for Representational State Transfer
- It's a design style (not a protocol) for building web services
- REST APIs use HTTP methods to perform actions on resources
- Each resource is identified by a URL (endpoint)

# Key Concepts

- Resources = Data Objects (e.g., users, products, orders)
- HTTP Methods map to CRUD operations:
  - GET → Read data
  - POST → Create new data
  - PUT → Update existing data
  - DELETE → Remove data
- Stateless: Each request contains all information (no session state on server)
- JSON is the most common format for request and response data
- Follows client-server architecture (browser/mobile app = client, server = API)



# Example: REST API for a Bookstore

HTTP Method	Endpoint	Description
GET	/books	Get list of books
GET	/books/42	Get details of book #42
POST	/books	Add a new book
PUT	/books/42	Update book #42
DELETE	/books/42	Delete book #42



# Benefits

---

- Simple and readable
- Uses standard web protocols (HTTP)
- Scales well
- Widely supported by tools and libraries

# Example: SimpleHttpServer

- Starts the server on port 8000
- Catches and prints any IOException that might occur (e.g. port already in use)

```
public static void main(String[] args) {  
    try {  
  
        SimpleHttpServer server = new  
SimpleHttpServer(8000);  
  
        server.start();  
  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

# Example: SimpleHttpServer

- Creates a new HTTP server that listens on a specified port (e.g. 8000)
- Registers two paths (called contexts):
  - /hello → handled by HelloHandler
- Uses Java's built-in thread pool (null means default executor)

```
public class SimpleHttpServer {  
    private final HttpServer server;  
    public SimpleHttpServer(int port) throws IOException {  
  
        server = HttpServer.create(new  
            InetSocketAddress(port), 0);  
  
        server.createContext("/hello", new HelloHandler());  
  
        server.setExecutor(null); // use default executor  
    }  
}
```



# Example: SimpleHttpServer

- What is an Executor?
  - An Executor is a Java interface used to manage threads.
  - It decides how tasks (like HTTP requests) are executed in the background.
  - Part of the `java.util.concurrent` package.
- Why is this important?
  - A web server may receive many requests at once.
  - Without an executor, requests would be processed one after another – very slow.
  - An executor enables parallel processing – faster and more scalable.

# Example: SimpleHttpServer

- `start()` → launches the server
- `stop(delaySeconds)` → gracefully shuts it down after a delay (in seconds)

```
public void start() {  
    server.start();  
  
    System.out.println("✅ Server started on http://localhost:"  
+ server.getAddress().getPort());  
}  
  
public void stop(int delaySeconds) {  
    server.stop(delaySeconds);  
  
    System.out.println("🛑 Server stopped.");  
}
```

# Example: SimpleHttpServer

- Sets the response text
- Sets the HTTP response header:
  - Key: Content-Type
  - Value: text/plain; charset=UTF-8
  - This tells the browser/client that the response is plain text
- Sends the HTTP status code 200 OK and the content length of the response
- Writes the response string to the output stream, which sends it to the client: Uses try-with-resources to automatically close the stream after sending the response

```
// --- Inner class for the /hello endpoint ---  
  
private class HelloHandler implements HttpHandler {  
  
    @Override  
  
    public void handle(HttpExchange exchange) throws IOException {  
  
        String response = "Hello from JDK HTTP Server!";  
  
        exchange.getResponseHeaders().set("Content-Type", "text/plain;  
        charset=UTF-8");  
  
        exchange.sendResponseHeaders(200, response.getBytes().length);  
  
        try (OutputStream os = exchange.getResponseBody()) {  
  
            os.write(response.getBytes());  
  
        }  
  
    }  
  
}
```

# Example: SimpleHttpServer

---

- How to test it after server has been started?
  1. <http://localhost:8000/hello>
  2. `curl -X GET http://localhost:8000/hello`



# Example: SimpleHttpServer

---

- How to test it after server has been started?
  1. <http://localhost:8000/hello>
  2. `curl -X GET http://localhost:8000/hello`

# Example: SimpleHttpServer

```
curl -X GET http://localhost:8000/hello
```

Part	Meaning
curl	Command-line tool to send HTTP requests
-X GET	Specifies the <b>HTTP method</b> : GET (optional in this case)
http://localhost:8000/hello	The <b>URL</b> to your local server's /hello endpoint

# Example: SimpleHttpServer

---

- Source code:

`SimpleHttpServer.java`

# Sending back JSON

---

- Please compare the SimpleHttpServer.java with SimpleHttpServerJSON.java
- Test with the browser and with curl



# Passing data in with the GET method

---

- Have a look at SimpleGreetServer.java.
- Test it like this:
  - curl <http://localhost:8000/greet?name=Alice&age=30>
  - Browser: <http://localhost:8000/greet?name=Alice&age=3>

# Passing data in with the POST method

- Run SimplePostServer.java
- Test it with Curl:
  - `curl -X POST http://localhost:8000/receive \`  
`-H "Content-Type: application/json" \`  
`-d '{"name": "Alice", "age": "30"}'`
  - For Windows:  
`curl -X POST http://localhost:8000/receive -H "Content-Type: application/json"`  
`-d "{\"name\": \"Alice\", \"age\": \"30\"}"`
- It cannot be tested with the browser!



# WEB UI



# WEB UI

---

- Creation of a Web UI, which accesses REST webservice
- Framework used: [Bootstrap](#)
- Web technologies: HTML5 & Javascript



- Basic Web page based on Bootstrap
- There is not much more than the title

- There is a button as a new element
- There is JavaScript code, which implements an Eventlistener for Click-events on the button
- When the button gets clicked the /hello Endpoint is called and its return text is presented.
- There is extra code to work with CORS

- What is CORS?
  - **CORS** is a **security feature implemented by browsers** to restrict web pages from making requests to a different domain (or port/protocol) than the one that served the web page.

- Why does CORS exist?
  - Prevent malicious websites from reading sensitive data from another site the user is logged into (e.g., your bank or email).
  - It enforces the same-origin policy, which allows requests only to the same domain unless the target server explicitly says: "I allow this."

- What happens without CORS?
  - The browser blocks the request and shows an error in the console like:  
  
Access to fetch at 'http://localhost:8000' from origin 'http://localhost:5500' has been block



- How to fix it?
  - the server (not the browser) must include special HTTP headers, like:
    - Access-Control-Allow-Origin: \*
    - or more securely:  
Access-Control-Allow-Origin: http://localhost:5500

## What is a Promise?

- A Promise is an object in JavaScript that represents the future result (or failure) of an asynchronous operation.
- It's like a placeholder for a value that will be available later.

## Promise States

- A Promise can be in one of three states:
  1. Pending – still working on it.
  2. Fulfilled – completed successfully.
  3. Rejected – failed with an error.

## Why are Promises useful?

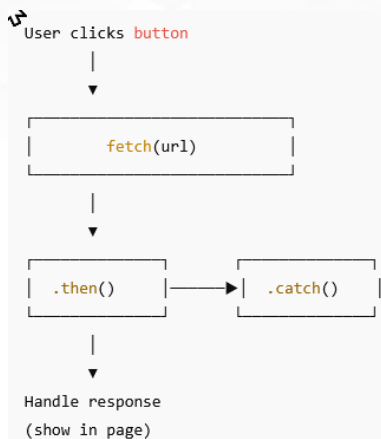
- JavaScript is asynchronous (non-blocking), so Promises let you:
  - Write cleaner code without deeply nested callbacks.
  - Wait for a result before continuing (using `.then()` or `await`).

## Example with `fetch()`

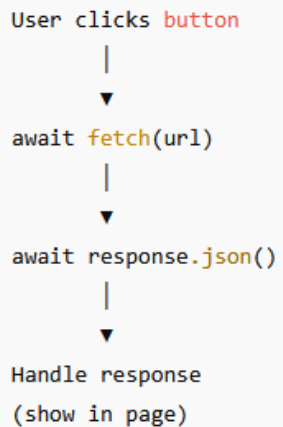
```
fetch("https://example.com/data")  
  .then(response => response.json())           // Handle the successful result  
  .then(data => console.log(data))             // Use the parsed data  
  .catch(error => console.error(error));       // Handle any errors
```

- This time the `async – await` pattern is replaced by using promises.

## Promises



## Async - Await





Feature	.then() / .catch()	async/await
Introduced in	ES6 (2015)	ES8 (2017)
Syntax style	Chained function calls	Linear, synchronous-looking code
Error handling	.catch() function	try/catch block
Readability (simple tasks)	Clear for small, short chains	More concise for longer sequences
Readability (complex)	Can get nested and messy	Cleaner for multiple awaits
Works in top-level scope	Yes	No (must be inside an async function)
Learning curve	Easy to start, but chaining can be tricky	Requires understanding of promises first
Common in	Early tutorials, older codebases	Modern JavaScript, frameworks like React/Vue

- This time the greet handler is called in our web service.
- Parameters are received by the content of text boxes in the UI.

- Here we receive a JSON string from the web api.

- Sending a post request
- In Java code we need to take care of the OPTIONS processing before the post request is sent.