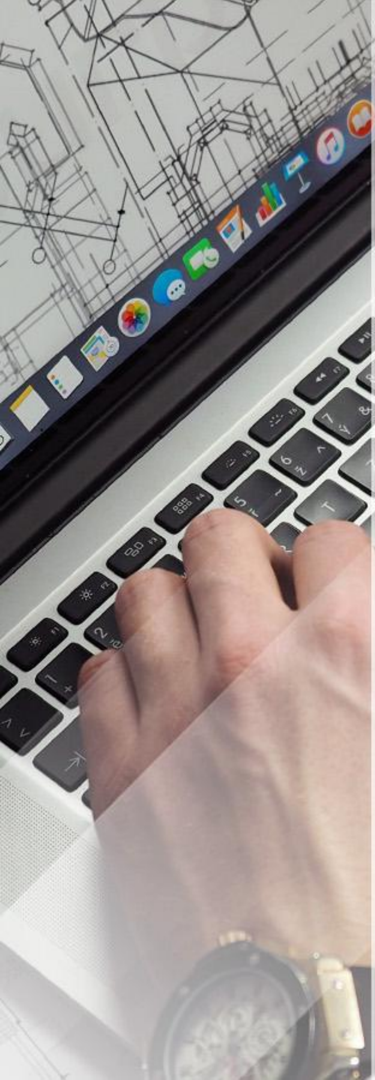
HHZ Sommersemester 2025

HHZ Bootcamp Java



Setup Jupyter Notebook

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

Verification:

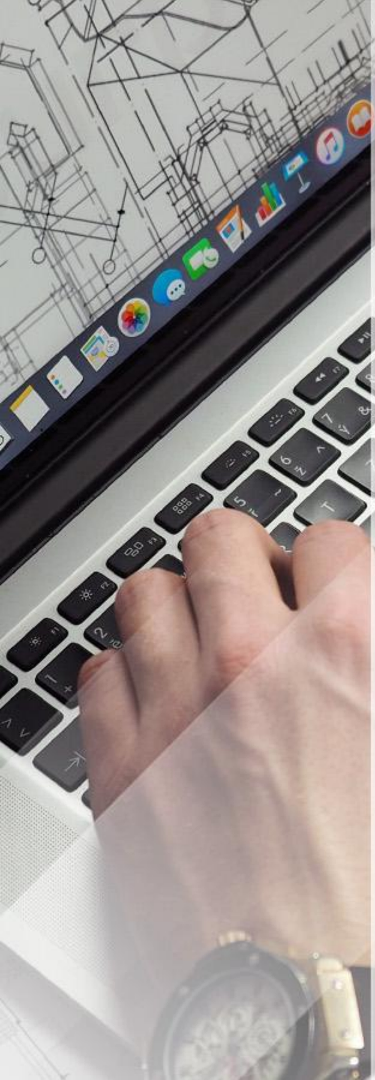
```
C:\\HHZ\Java - Code Camp\temp>jupyter kernelspec list
                              Available kernels:
 java              C:\Python313\share\jupyter\kernels\java
 python3           C:\Python313\share\jupyter\kernels\python3
```

Aufteilung der 3 Boot-Camp Tage

Tag #	Inhalt
1	• Git Rehash • Java Besonderheiten und Code testen • SQL (Fokus SELECT Befehl) • JDBC • ORM • Projektarbeit
2	• Rest Web Services in Java erstellen • Projektarbeit
3	• HTML5 • Projektarbeit einschließlich Integration der erarbeiteten Komponenten

Exkurs: Git Rehash

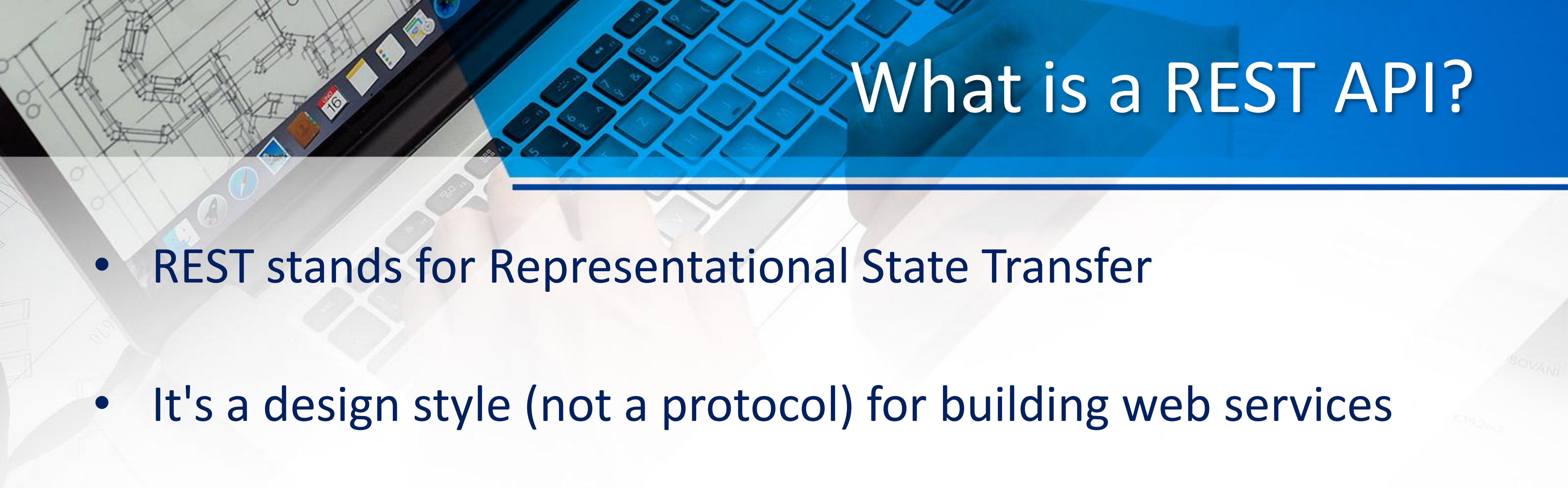
Siehe GIT-Foliensatz



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

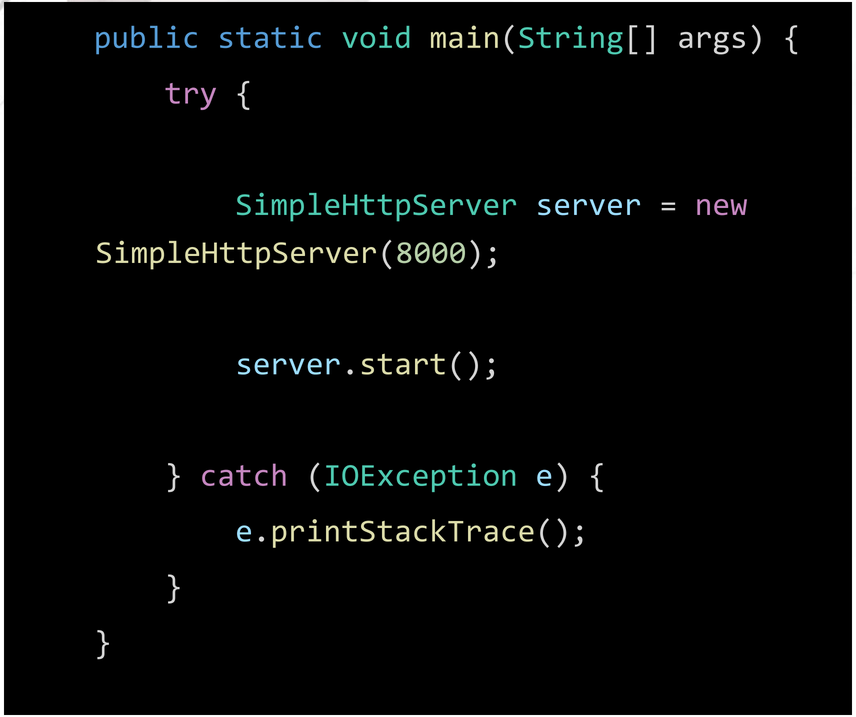
Benefits

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

Example: SimpleHttpServer

- Starts the server on port 8000

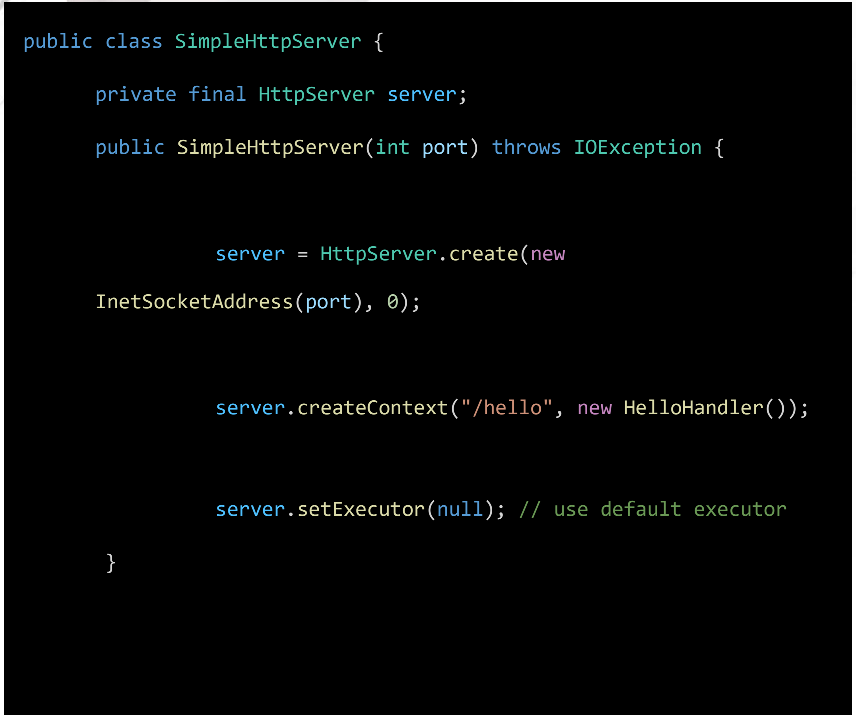
- Catches and prints any IOException that might occur (e.g. port already in use)



```
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

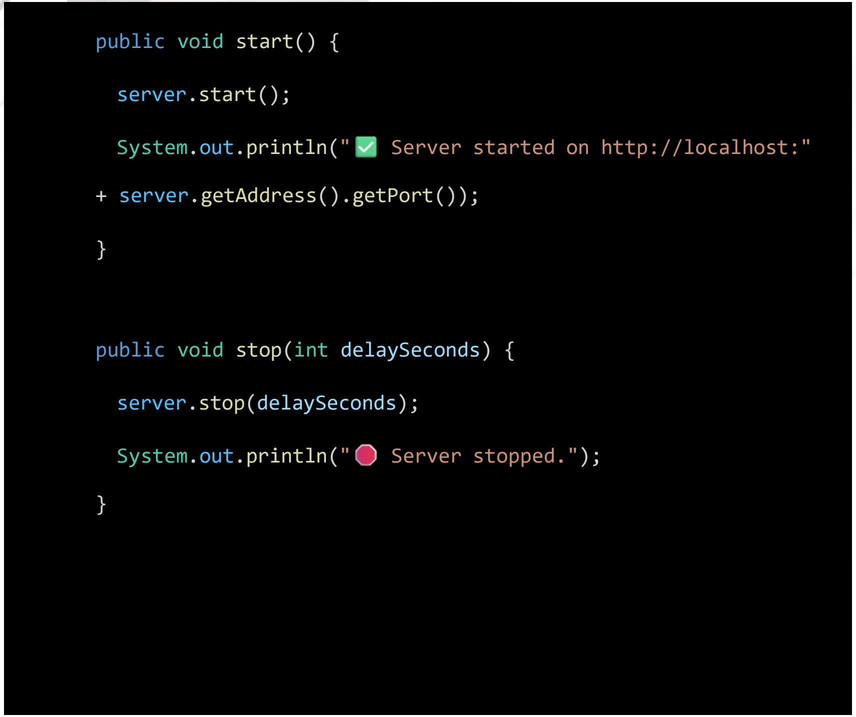
- What is an Executor?
  - An Executor is a Java interface used to manage threads.
  - It decides how tasks (like HTTP requests) are executed in the background.
  - Part of the java.util.concurrent package.

- 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

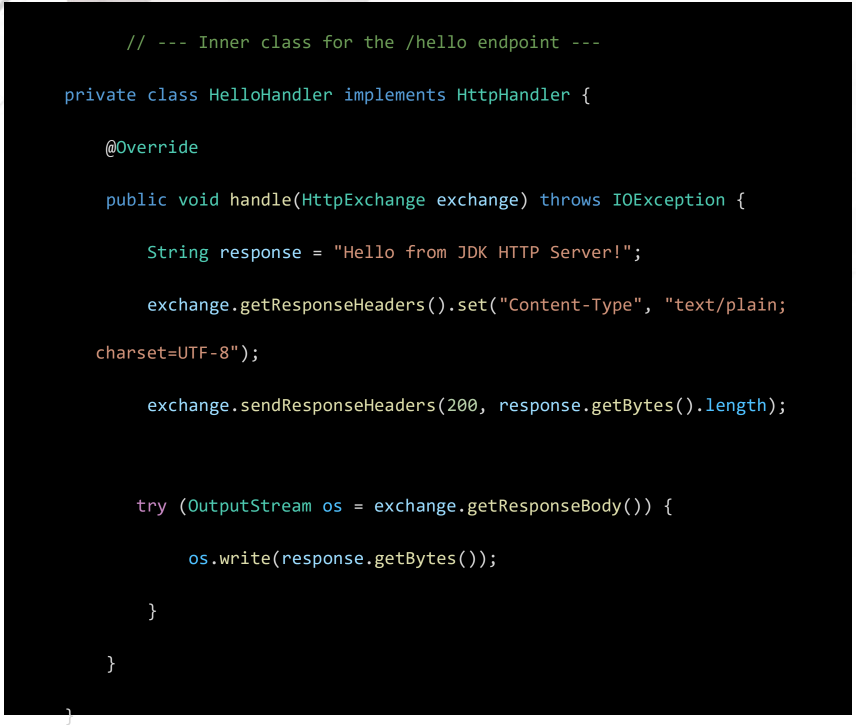
- stop(delaySeconds) → gracefully shuts it down after a delay (in seconds)



```
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 **HTTP method**: GET (optional in this case)
http://localhost:8000/hello	The **URL** 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 webservices
- Framework used: Bootstrap
- Web technologies: HTML5 & Javascript

010.html

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

020.html

- 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 ist return text is presented.
- There is extra code to work with CORS

020.html

- 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.

020.html

- 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."

020.html

- 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

020.html

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

020.html

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:

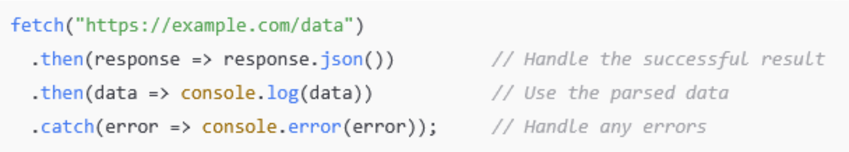
1. Pending – still working on it.
2. Fulfilled – completed successfully.
3. Rejected – failed with an error.

020.html

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
```