Praktikum Mobile und Verteilte Systeme

Client Server Communication

Prof. Dr. Claudia Linnhoff-Popien et al.

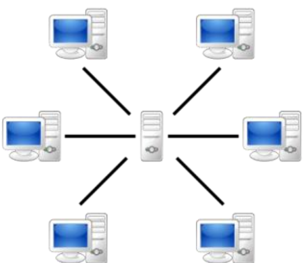
Sommersemester 2018

Today

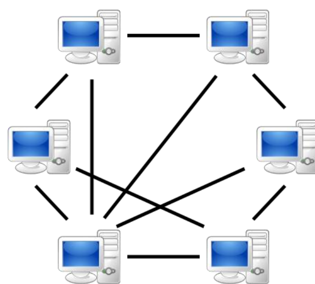
1. Communication models
2. TCP and HTTP basics
3. RESTful API architectures
4. WebSockets
5. REST VS Websockets
6. Client-Server Communication Implementation (Frameworks etc.)
7. Create REST and WebSocket servers with Spring

Communication models and examples

- **Client-server model**
  - **Application**: WWW
    Client: Webbrowser (e.g Firefox)
    Server: Webserver (e.g. Apache)
    Protocol: HTTP(S)
  - **Application**: E-Mail
    Client: Mailclient (e.g. Outlook)
    Server: Mailserver (e.g. MS Exchange)
    Protocol: POP, IMAP, SMTP, …



- **P2P model**
  - **Application**: Filesharing
    Peer: BitTorrent-Client (e.g µTorrent)
    Protocol: BitTorrent
  - **Application**: Cryptocurrency
    Peer: Bitcoin-Wallet (e.g. Bitcoin Core)
    Protocol: Bitcoin-Protocol



ISO/OSI Modell overview

ISO/OSI Layer		Protocol	Comment
7	Application-Layer	HTTP, IMAP, BitTorrent	Defined message format, e.g plain-text ASCII for HTTP
5-6	Session & Presentation		
4	Transport-Layer	TCP, UDP	Using a socket, we send&receive streams of bytes
1-3	Physical & Data Link & Network		

So what then does an Application like Firefox do?

1: It opens a connection to a server using a **socket** (IP and Port)
→ **TCP** Layer 4

2: It talks to the server in a certain protocol
→ **HTTP** Layer 7

3: It receives the response, parses it and displays it

Low-level example

We can do this step by step in order to understand what is really happening:

1: Open a TCP connection:

Use Netcat („nc“ on Linux) or telnet (on Windows)
It simply opens a TCP socket and lets you send bytes to the recipient (Layer 4)
A socket can be opened using an IP (or hostname) and a port.

```
$ nc google.de 80
```

← Press Enter
← Connection/Socket is opened and waiting for input
You can input any data which is sent as bytes to the server

Low-level example

We are talking to a Webserver which expects the HTTP Protocol

1. Lets do it **wrong** and say „hello!“ ;)

```
$ nc google.de 80
hello!                          <-- Our input
HTTP/1.0 400 Bad Request        <-- Response from Google's Webserver
```

We did not conform to the protocol, so the server responded how it saw fit and closed the connection.

2. Lets do it right and conform to the HTTP protocol:

```
$ nc google.de 80

GET / HTTP/1.1                  <-- Our input

HTTP/1.1 302 Found              <-- Response from Google's Webserver
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Referrer-Policy: no-referrer
Location: http://www.google.de/?gfe_rd=cr&dcr=0&ei=hYbgWdqbE6_PXvmVsqAE
Content-Length: 266
Date: Fri, 13 Oct 2017 09:25:25 GMT

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
```

Why do we need protocols like HTTP?

What do we see in these examples?

- Layer 4 communication using sockets is protocol agnostic.
- That means it has no concept of URLs, Methods, or anything like it.
- It only knows bytes.

- How then do we access different documents, execute different actions, etc. ? If we want „higher-logic“ like Methods, URLs, etc, we need a protocol on top.

```
$ nc google.de 80

GET / HTTP/1.1

HTTP/1.1 302 Found
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Referrer-Policy: no-referrer
Location: http://www.google.de/?gfe_rd=cr&dcr=0&ei=hYbgWdqbE6_PXvmVsqAE
Content-Length: 266
Date: Fri, 13 Oct 2017 09:25:25 GMT

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
```

Socket is open. To netcat everything that follows is just bytes

HTTP Protocol: Method (e.g. GET) followed by URI (e.g. /) followed by Protocol-Version (e.g. HTTP/1.1)

I want to GET the page at URI /

Note: Normally one would use „curl“ for that (e.g.: `curl http://google.de/`)

How to structure application access (i.e. the API)?

One possible way: REST (Representational State Transfer)

- Defined by Roy T. Fielding
  - Dissertation: “Architectural Styles and the Design of Network-based Software Architectures“ (2000)
  - main author of HTTP/1.0 and HTTP/1.1
  - co-founder of the Apache HTTP server project (**httpd**)

Important:

- REST is an architectural style, not a protocol!
- You can use HTTP and it’s URIs, Methods (GET, POST, etc.) to do this, but as REST is simply a way to structure things, it is not limited to HTTP

REST principles I: Everything is a resource

- **Every data element** of an application that should be accessible **is a resource** with it's own **unique URI**
- The resource is not an actual object or service itself, but rather **an abstract interface** for using it
- Using **human-readable URIs** is common (yet not obligatory)

```
http://example.com/customers/1234
http://example.com/orders/2013/1/12345
http://example.com/orders/2013/1
http://example.com/products/4554
http://example.com/products?color=green
http://example.com/processes/salary-increase
```

- Important architectural principles of REST
  - Everything is a **resource**
  - Communicate **statelessly**
  - Use a **common interface** for all resources
  - Resources can have **multiple representations**

REST principles II: Communicate statelessly

- REST includes the concept of **statelessness** on behalf of the server
  - but, of course, there is some state…

- All application state should either
  - be **turned into resource state**
  - or be **managed at the client**

- All **requests should be independent** from earlier requests
  - messages are **self-contained**, including all necessary information

- Advantages:
  - **scalability**
  - **isolation of the client** against changes on the server

REST principles III: Use standard methods

- REST demands the usage of **simple, uniform interfaces** for all resources
- When implementing REST using HTTP, we make use of the **HTTP verbs** (GET, POST, etc.)
- With REST, these verbs are mapped to resource-specific semantics

```
class Resource {                              // analogy to oo-programming
  Resource(URI u);                            // URI
  Response get();                             // HTTP GET
  Response post(Request r);                   // HTTP POST
  Response put(Request r);                    // HTTP PUT
  Response delete();                          // HTTP DELETE
}
```

REST principles IV: Different representations

- Resources can (and actually should) have **multiple representations**
  - provide multiple representations of resources for different needs
  - ideally, at least one standard format should be provided

- When using HTTP, the selection of data formats is done **using HTTP content negotiation**
  - clients can ask for a representation in a particular format

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: application/xml
```

```
GET /customers/1234 HTTP/1.1
Host: example.com
Accept: text/x-vcard
```

- Advantages:
  - The different representations of a resource (e.g., text, XML, HTML, JSON…) can be consumed by different clients all via the same interface

REST-conformant usage of HTTP methods

- **HTTP GET**
  - Used for accessing the requested resource without any side-effects. A resource must never be changed via a GET request (read-only)!
- **HTTP PUT**
  - Used for creating or updating a resource at a known URI.
- **HTTP DELETE**
  - Used for removing a resource.
- **GET**, **PUT** and **DELETE** must be implemented as idempotent methods
  - can be called repeatedly without leading to different results
- **HTTP POST**
  - Update an existing resource or create a new one (not idempotent)

A simple example of a RESTful web service

- Mapping of “normal” method names to RESTful resource interfaces
  - combination of resource URIs and the standard HTTP methods

Normal method name	URI (RESTful resource)	HTTP method
listOrders	/orders	GET
addNewOrder	/orders	POST
addNewOrder	/orders/12344	PUT
getOrder	/orders/12344	GET
deleteOrder	/orders/12344	DELETE
listCustomers	/customers	GET
getCustomer	/customers/beck	GET
addCustomer	/customers	POST
addCustomer	/customers/beck	PUT
updateCustomer	/customers/ebert	PUT
…		

Advantages of the RESTful approach

- **Simplicity**
  - well known interfaces (URIs, HTTP methods), no new XML specification
- **Lightweightness**
  - short messages, little overhead
- **Multiple representations**
- **Security**
  - authentication and authorization can be done by the web server
- **Scalability (e.g., multi-device usage / multiple servers)**
- **Reliability (e.g., on restoring state / recovering)**
- **Caching**
- **Easy service orchestration** (via hyperlinks)
  - URIs define global namespace, no application boundaries

Example REST API Request

```
$ curl -v https://jsonplaceholder.typicode.com/posts/1
*   Trying 104.31.87.157...
* Connected to jsonplaceholder.typicode.com (104.31.87.157) port 443 (#0)
```

* are info messages printed by curl (not something sent or received to/from the server)
Curl is opening the Layer 4 TCP connection
(same as `nc jsonplaceholder.typicode.com 443` would do)

```
> GET /posts/1 HTTP/1.1
> Host: jsonplaceholder.typicode.com
> User-Agent: curl/7.47.0
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Mon, 16 Oct 2017 11:52:53 GMT
< Content-Type: application/json; charset=utf-8
< Content-Length: 292
< Connection: keep-alive
< Set-Cookie: __cfduid=daf12330e5d91d7757d30164aade610031508154773; expires=Tue, 16-Oct-18 11:52:53 GMT; path=/; domain=.typicode.com; HttpOnly
< X-Powered-By: Express
< Vary: Origin, Accept-Encoding
< Access-Control-Allow-Credentials: true
< Cache-Control: public, max-age=14400
< Pragma: no-cache
< Expires: Mon, 16 Oct 2017 15:52:53 GMT
< X-Content-Type-Options: nosniff
< Etag: W/"124-yiKdLzqO5gfBrJFrcdJ8Yq0LGnU"
< Via: 1.1 vegur
< CF-Cache-Status: HIT
< Server: cloudflare-nginx
< CF-RAY: 3aead0872e4a642d-FRA
<
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem s
unt rem eveniet architecto"
* Connection #0 to host jsonplaceholder.typicode.com left intact
}$
```

> shows data sent by curl (only if curl is run in verbose mode `-v`)
We request to **GET** the resource at **/posts/1**
Note: we did not specify GET explicitly – this is the default for curl
You can change this by using e.g. **`curl -X POST http://...`**

< shows HTTP Header data received from the server

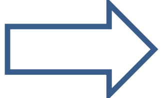
Here starts the http response body. In this case the resource
In json format

Great, so REST is the answer to everything, right?

- What if the server should notify the client?
  E.g.: Chat-Application, …
- What if we need to continously send and receive data?
  E.g.: Video-Conference
- What if we need to be as fast as possible (low overhead)?
  E.g.: Online-Games

OK, I‘ll just use plain old sockets for that

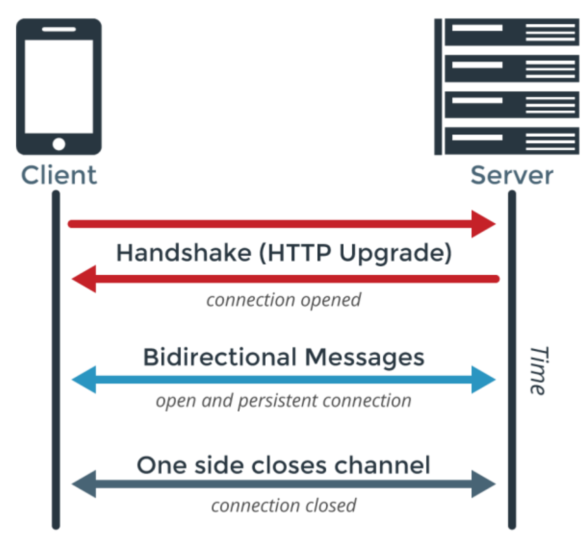
- Lots of Firewalls block non HTTP-Ports / Communication
- How do you initiate a socket connection? – You can‘t do that using a Browser



WebSockets!

WebSockets

- TCP-based network protocol, standardized in 2011
- enables **bidirectional communication** between web applications and a WebSocket server
- **Client-server** as well as **Peer-to-peer** (P2P) usage is possible
- Server stays **always open** for commu-nication and data may always be pushed to the client
- Good fit for high message frequencies, Ad hoc messaging and Fire-and-forget scenarios



Source: https://www.pubnub.com/blog/2015-01-05-websockets-vs-rest-api-understanding-the-difference/

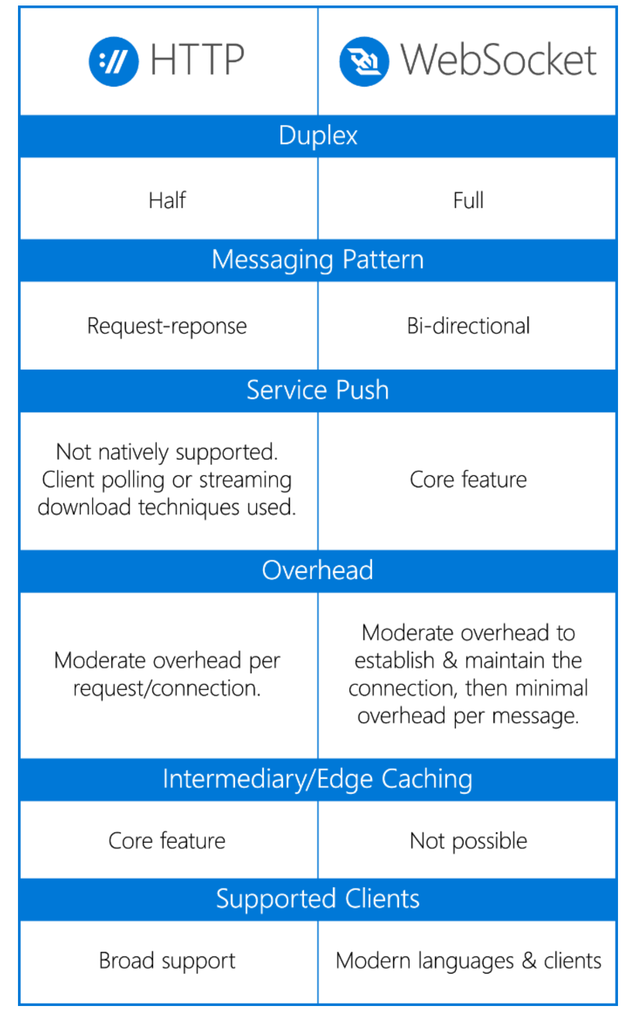
Advantages of WebSockets

- **Two-Way communication**
  - Frequent updates **without regular polling**
  - no redundant creation of new TCP connections for every exchange
- **Lower overhead per message**
  - **TCP connection** is created only **once**
  - No need to send something like the HTTP-Header everytime
- **Stateful connections**
  - Without sending cookies or session Ids with every request
  - Server side handling of **complex application states** possible

REST VS WebSockets

- **When to use REST APIs VS HTTP:**
  → For retrieving single resources
  → Stateless communication, **application state is** handled on client side
  → No frequent updates needed
  → Idempotency enables **cachable** resources
  → Handling of **synchronized communication**

- **When to use WebSockets**
  → **Fast reaction time**
  → **High frequent updates** needed with small payloads
  → Complex application state can be handled on server side
  → **Fire-and-forget** scenarios
  → Constantly **changing states** (e.g., game state)
  → Distribution of application state to **multiple clients**



	HTTP	WebSocket
Duplex	Half	Full
Messaging Pattern	Request-reponse	Bi-directional
Service Push	Not natively supported. Client polling or streaming download techniques used.	Core feature
Overhead	Moderate overhead per request/connection.	Moderate overhead to establish & maintain the connection, then minimal overhead per message.
Intermediary/Edge Caching	Core feature	Not possible
Supported Clients	Broad support	Modern languages & clients

Source: https://blogs.windows.com/buildingapps/2016/03/14/when-to-use-a-http-call-instead-of-a-websocket-or-http-2-0/#mEmEPjKq1bZXFGsE.97

Client-Server Communication Implementation

- Implement everything above Layer 4 yourself
  (in any programming language that supports sockets)

Probably better idea:

- Use one of the various frameworks or a programming language that provides out-of-the-box support:
  - Java Spring
  - Firebase
  - Python Flask
  - NodeJS
  - Jersey (JAX-RS)
  - **Others...**

Python Flask

- Open source REST API framework for Python
- Pretty simple implementation

```
from flask import Flask
app = Flask(__name__)

@app.route("/hello")
def hello():
    return "Hallo Welt"

if __name__ == "__main__":
    app.run()
```

Flask
web development,
one drop at a time

- Resource `/hello` is now available via HTTP:
  - `http://<your-server-ip>:<port>/hello`
- Server responds with *„Hallo Welt“*

Google Firebase

- **Mobile and web application platform** for building „high quality apps“
- Founded in 2011, acquired by **Google** in 2014
- Originally: **realtime database** for **synchronizing and storing** data across multiple devices.
- Now: **full suite** for app development:
  - FB Analytics
  - FB Cloud Messaging (successor of Google Cloud Messaging (GCM))
  - FB Auth
  - FB Database
  - **Others…**
- Integration of Android, iOS, Javascript, Java, Objective-C, swift, etc.
- REST API: **Server-Sent Events** protocol (HTTP connections are created for receiving push notifications from a server)

The Spring Framework

- Open-Source framework for Java and JavaEE development
- Probably the most widely used enterprise web framework
- Capable of WebSocket as well as RESTful communication
- Easy and fast server setup for client-server communication
  - Dependency injection via **inversion of control (IoC)** container:
    - Manages object lifecycles automatically
    - Needed resources become assigned to objects automatically
  - **Aspect oriented programming (AOP)**
    - Technical aspects such as *transactions* or *security* are handled isolated from the actual code
- Possible to use only the parts you need
  - Spring Boot (Get up and running quickly by following conventions)
  - Spring Data (Consistent approach to data access – no matter what databse is really used)
  - Spring Security (Authentication / Authorization)
  - …

Creating a Spring Server

Install an IDE for java programming: **IntelliJ**

- Download of the community version at:
  - https://www.jetbrains.com/idea/
- If you choose the version without JDK, make sure that the latest Java Runtime Environment (JRE) and Java Software Development Kit (JDK) are installed
  - http://openjdk.java.net
  - http://www.oracle.com/technetwork/java/javase/downloads/index.html
- JDK and JRE must be known to your machine‘s runtime environment
  - i.e. **JAVA_HOME environment variable must be set**

    On Debian based systems: **update-alternatives --display javac**

    Add the path (without /bin) shown in the following line to your ~/.bashrc then logout/login again Example:

    export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
- Spring can use maven or gradle for build management
- Install gradle. On Debian based systems (Ubuntu etc.):
  - **sudo apt-get install gradle**

Gradle Build Manager

- Automated **dependency management** and **build automation**
- Based on *Apache Ant* and *Apache Maven*
- Uses *Groovy*-based domain-specific language (DSL)
- Dependency configuration within the `build.gradle` file e.g.:

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-plugin:1.5.5.RELEASE")
    }
}
apply plugin: 'java'
apply plugin: 'org.springframework.boot'

jar {
    baseName = 'yourExampleApplication'
    version =  '0.1.0'
}
sourceCompatibility = 1.8
targetCompatibility = 1.8

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

WATCH YOUR build.gradle **CODE e.g. buildscript** must be the first function

Gradle Build Manager

- By default assumes **Maven directory structure** for **Java sources** and **resources**
- These directories are:
  - src/main/java
  - src/main/resources
  - src/test/java
  - src/test/resources
- Simplest build.gradle for a basic java project

```
apply plugin: 'java'
```

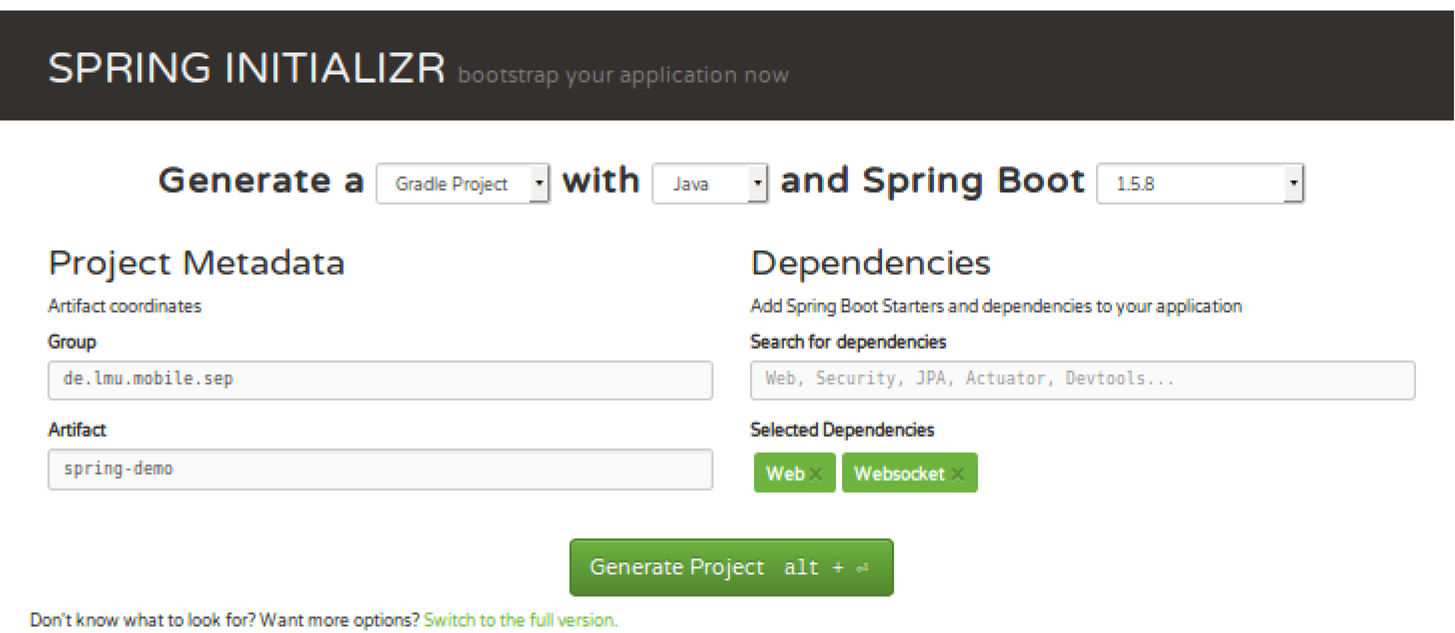
Running **gradle build** has the effect of:

```
> gradle build
:compileJava
:processResources
:classes
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build

BUILD SUCCESSFUL
```

Using the Spring framework

- Use the online tool https://start.spring.io/ to generate an empty spring boot project. You can use it to get started quickly instead of manually setting up your src/main/java folders, build.gradle etc.
- Choose the dependencies you want to use in your project.
  - **Required: Web** Optional: **WebSocket**
- Choose a Group name: Usually your company's domain & department Convention: reverse url style e.g. de.lmu.mobile.sep
- Choose an Artifact name (The name of your application)



Using the Spring framework

- Unpack the zip file
- Import the project into IntelliJ
  - Use „Import project from external model": Gradle

Project SDK is not defined Setup SDK

- Tell IntelliJ which java SDK to use. Add the JDK which is available on your system.
- Now no more errors should be shown in IntelliJ
- Open the SpringDemoApplication in src/main/java/de/lmu/mobile/sep/springdemo

SpringDemoApplication.java

```
1   package de.lmu.mobile.sep.springdemo;
2
3   import org.springframework.boot.SpringApplication;
4   import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6   @SpringBootApplication
7   public class SpringDemoApplication {
8
9       public static void main(String[] args) { SpringApplication.run(SpringDemoApplication.class, args); }
12  }
13
```

Start the application from IntelliJ – The server will listen on port 8080:
Run/Run ‚SpringDemoApplication'
Or from the terminal: **gradle bootrun**
Or to build an executable, standalone jar to deploy on a server: **gradle build**

Using the Spring framework

SpringDemoApplication.java

```java
package de.lmu.mobile.sep.springdemo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class SpringDemoApplication {

    public static void main(String[] args) { SpringApplication.run(SpringDemoApplication.class, args); }
}
```

- The class is marked for Spring as the launching class by using the annotation `@SpringBootApplication`
- Normally you don‘t need to add/modify anything else here
  - Other classes, controllers etc. are configured & started using annotations
- Spring uses Apache Tomcat by default as it‘s servlet/web server
- The used port is 8080 by default and optionally configurable within the *`application.properties`* file
  (create it if it doesn‘t exist in `/src/main/resources/`):

```
server.port = <port-number>
```

- You can now access the application at http://localhost:8080

REST with Spring

- Create a new class and annotate it with `@RestController`:

DemoController.java

```java
package de.lmu.mobile.sep.springdemo;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class DemoController{
    @RequestMapping("/hello")
    public String myHelloMethod(){
        return "Hello";
    }

    @RequestMapping("/hello2")
    public String myHelloMethod2(@RequestParam("a") String aParam, @RequestParam("b") int anotherParam){
        return "a: " + aParam + " b:" + (anotherParam+1);
    }
}
```

- `@RequestMapping` allows us to map a path to a specific REST resource
  - without specific request type definition, the annotated method is responsible for all incoming calls to the specified path (i.e., POST, GET, etc.)
  - HTTP methods may be routed to different java methods by adjusting the basic annotation (e.g., `@RequestMapping(method=GET, value=“/hello“)` for handling only GET requests)
- Due to the annotation, the REST controller runs automatically on application startup (no initialization within the main application is needed!)
- The REST interface is accessible via HTTP:
  - http://localhost:8080/hello
  - http://localhost:8080/hello2?a=something&b=5

REST with Spring

Automatic creation of JSON encoded resources from POJOs (plain-old-java-objects)

Spring can automatically serialize and deserialize java objects to json/xml etc.

- We can **return java objects** from the methods
- We can **receive serialized (json, etc.)** objects and spring automatically **deserializes** them into **java objects**

MyHelloObject.java

```
1   package de.lmu.mobile.sep.springdemo;
2
3   public class MyHelloObject {
4
5       private final String message;
6
7       public MyHelloObject(String message) { this.message = message; }
10
11      public String getMessage() {
12          return message;
13      }
14  }
15
```

DemoController.java

```
1   package de.lmu.mobile.sep.springdemo;
2
3   import ...
8
9   @RestController
10  public class DemoController{
11
12      private static final Logger logger = LoggerFactory.getLogger(DemoController.class);
13
14      @RequestMapping("/hello-object")
15      public MyHelloObject myHelloObjectMethod(){
16          return new MyHelloObject("I am a java object");
17      }
18
19      @RequestMapping("/hello-receive-object")
20      public MyHelloObject myHelloReceiveObjectMethod(@RequestParam("hello") MyHelloObject myObject){
21          // write log
22          logger.info("We received a "+myObject.getClass()+ " object with message "+myObject.getMessage());
23          // echo it back to the client
24          return myObject;
25      }
26  }
```

- The returned object looks as follows: `{"message":"I am a java object"}`

WebSockets with Spring

- Check that the Spring libraries for websockets is included in build.gradle:

```
dependencies {
    …
    compile("org.springframework.boot:spring-boot-starter-websocket")
```

- Create a class which implements the WebSocketConfigurer

```
@Configuration
@EnableWebSocket
public class MyWebSocketConfigurer implements WebSocketConfigurer {

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        //registry.addHandler(..) enables us to register a handler for a specific path
        WebSocketHandler handler = new MySocketHandler();
        registry.addHandler(handler, "/<pathname>" );
    }
}
```

- The annotations `@Configuration` and `@EnableWebSocket`:
  - indicate to Spring, that a web socket connection should be made available
  - ensure that our configurer class is launched automatically on server startup
- A `<pathname>` must be set at which the WebSocket connection is accessible
- Create and configure a Handler class for session and message handling

WebSockets with Spring

- Basically, we want our WebSocket handler to:
  - manage sessions (e.g. store incoming sessions, remove closed ones)
  - distribute messages among server and client sessions

```
public class MySocketHandler extends TextWebSocketHandler {

    private ArrayList<WebSocketSession> sessionQueue;

    @Override
    public void afterConnectionEstablished(WebSocketSession session) throws Exception {
        super.afterConnectionEstablished(session);
        // handle incoming sessions
    }

    @Override
    public void afterConnectionClosed(WebSocketSession session, CloseStatus status) throws Exception {
        super.afterConnectionClosed(session, status);
        // handle closed sessions
    }

    @Override
    public void handleMessage(WebSocketSession session, WebSocketMessage<?> message) throws Exception {
        super.handleMessage(session, message);
        // handle incoming messages, e.g., distribute them among target sessions
    }
}
```

Remember: With WebSockets you are basically back on „Layer 4“

- You need to define your own protocol
  (i.e. your websocket knows no URIs like /myresourceX
  or HTTP methods like GET/POST etc.)

- You could e.g. simply exchange JSON objects with a „method“ field instead which you use in your server code to route to the correct java method

- You can use a generic protocol like STOMP
  (https://en.wikipedia.org/wiki/Streaming_Text_Oriented_Messaging_Protocol)

- To test the basic function of your WebSocket Server you can e.g. use the python tool websocket-client. If you have python and pip installed on your system: `pip install websocket-client`
  Use the command wsdump <url> to test:
  E.g. if you set the <pathname> of your websocket endpoint in spring to „/mywebsocket“:
  `wsdump ws://localhost:8080/mywebsocket`

Useful Links

- Spring Documentation
  - https://docs.spring.io/spring-framework/docs/
  - https://spring.io/guides/gs/spring-boot/

- Spring REST Tutorial
  - https://spring.io/guides/gs/rest-service/

- Creating and configuring Gradle projects with IntelliJ
  - https://www.jetbrains.com/help/idea/gradle.html
- Spring Unit / Integration Testing
  - http://www.baeldung.com/integration-testing-in-spring
  - https://spring.io/guides/gs/testing-web/
  - https://joel-costigliola.github.io/assertj/