Organisatorisches

- 30.06.2019 (So) - Übungsabgabe
- 21.07.2019 (So) - Abgabe Projekte
- 23.07.2019 (Di) 10:00 - 13:00 Uhr – Abschlusspräsentationen

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Praktikum Mobile und Verteilte Systeme

Client Server Communication

Prof. Dr. Claudia Linnhoff-Popien et al.

Sommersemester 2019

Today

Basics

1. Communication models
2. TCP and HTTP basics
3. RESTful API architectures
4. WebSockets
5. REST VS Websockets

6. Implementation for your Apps

Communication models – Client Server

Examples

- **Smartphones**
  Client: (Android) App
  Server: Webserver (e.g. Apache)
  Protocol: HTTP(S)

- **WWW**
  Client: Webbrowser (e.g Firefox)
  Server: Webserver (e.g. Apache)
  Protocol: HTTP(S)

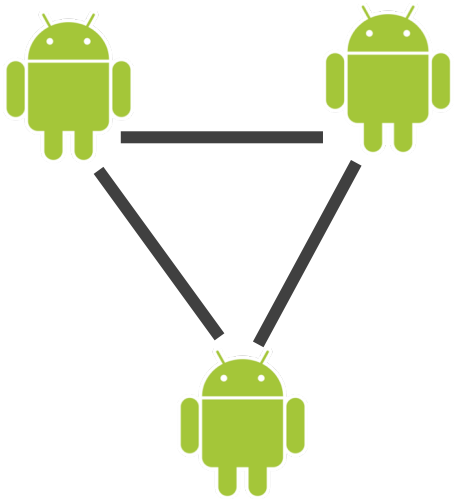
- **E-Mail**
  Client: Mailclient (e.g. Outlook)
  Server: Mailserver (e.g. MS Exchange)
  Protocol: POP, IMAP, SMTP, …

Communication models – P2P

Examples

- **Filesharing**
  Peer: BitTorrent-Client (e.g µTorrent)
  Protocol: BitTorrent

- **Cryptocurrency**
  Peer: Bitcoin-Wallet (e.g. Bitcoin Core)
  Protocol: Bitcoin-Protocol



ISO/OSI Model

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		

ISO/OSI - Firefox Example

What does an application like Firefox do?

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

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

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

Low-level example

1: Open a TCP connection in Netcat („nc“ on Linux) or telnet (on Windows)

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

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.

Bad Low-level example

2: Send useless bytes, e.g. „hello!“

```
nc google.de 80
hello!
HTTP/1.0 400 Bad Request
Content-Type: text/html; charset=UTF-8
Referrer-Policy: no-referrer
Content-Length: 1555
Date: Mon, 29 Apr 2019 12:30:17 GMT
```

We did not conform to the protocol, so the server closed the connection.

Good Low-level example

2: Conform to the protocol

```
$ nc google.de 80
GET / HTTP/1.1

HTTP/1.1 200 OK
Date: Mon, 29 Apr 2019 12:22:03 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
```

We receive the response code 200, OK!

Why do we need protocols like HTTP?

Layer 4 communication using sockets is protocol agnostic.

→ It has no concept of URLs, Methods, …

→ It only knows bytes

If we want „higher-logic“ (access documents, execute actions),

➔ we need a protocol on top

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

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

```
GET / HTTP/1.1
```

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

```
HTTP/1.1 200 OK
Date: Mon, 29 Apr 2019 12:22:03 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
```

Structuring application access with REST

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!**
- REST is simply a way to structure things, it is not limited to HTTP

Important architectural principles of REST

1. Everything is a **resource**
2. Communicate **statelessly**
3. Use a **common interface** for all resources
4. Resources can have **multiple representations**

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
```

REST principles II: Communicate statelessly

- 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 a common interface

- 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: Multiple representations

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

- The selection of data formats is done **using HTTP content negotiation**

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

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

- Advantages:
  - Multiple representations can be consumed 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

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 2606:4700:30::6812:24a3...
* Connected to jsonplaceholder.typicode.com (2606:4700:30::6812:24a3) port 443 (#0)
```

Curl is opening the Layer 4 TCP connection
(same as `nc jsonplaceholder.typicode.com 443` would do)

```
> GET /posts/1 HTTP/2
> Host: jsonplaceholder.typicode.com
> User-Agent: curl/7.54.0
> Accept: */*

…

< HTTP/2 200
< date: Thu, 02 May 2019 12:22:43 GMT
< content-type: application/json; charset=utf-8
…
{
  "userId": 1,
  "id": 1,
  …
}
```

shows data sent by curl (only if curl is run in verbose mode `-v`)
GET (default) the resource at **/posts/1**

shows HTTP Header data received from the server

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

REST ist not the solution to everything

- 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

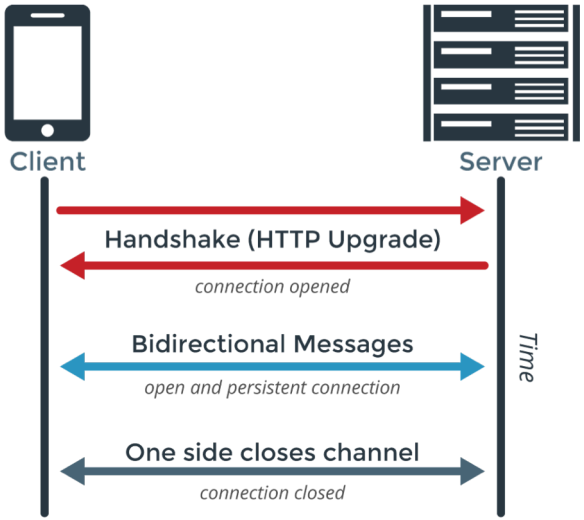
WebSockets for real-time communication

Open up a plain TCP socket connection as the solution?

→ Non-HTTP Communication is often blocked by Firewalls

→ You can‘t open a plain socket in your browser

➔ **WebSockets** for fast real-time communication between Server and Client



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

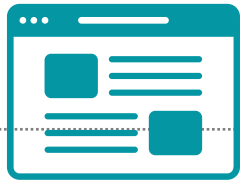
WebSockets

- 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 communication and data may always be pushed to the client
- Good fit for high message frequencies, Ad hoc messaging and Fire-and-forget scenarios

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

When to use REST APIs



→ For retrieving single resources

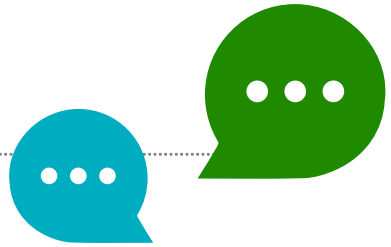
→ Stateless communication,
application state is handled on client side

→ No frequent updates needed

→ Idempotency enables **cachable** resources

→ Handling of **synchronized communication**

When to use WebSockets



→ **Fast, 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**

Client-Server Implementation

Option 1

- Implement everything above Layer 4 yourself!

Option 2 

- Use one of the various frameworks or a programming language that provides out-of-the-box support.

Frameworks and BaaS

Frameworks

Backend As A Service

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

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'
```

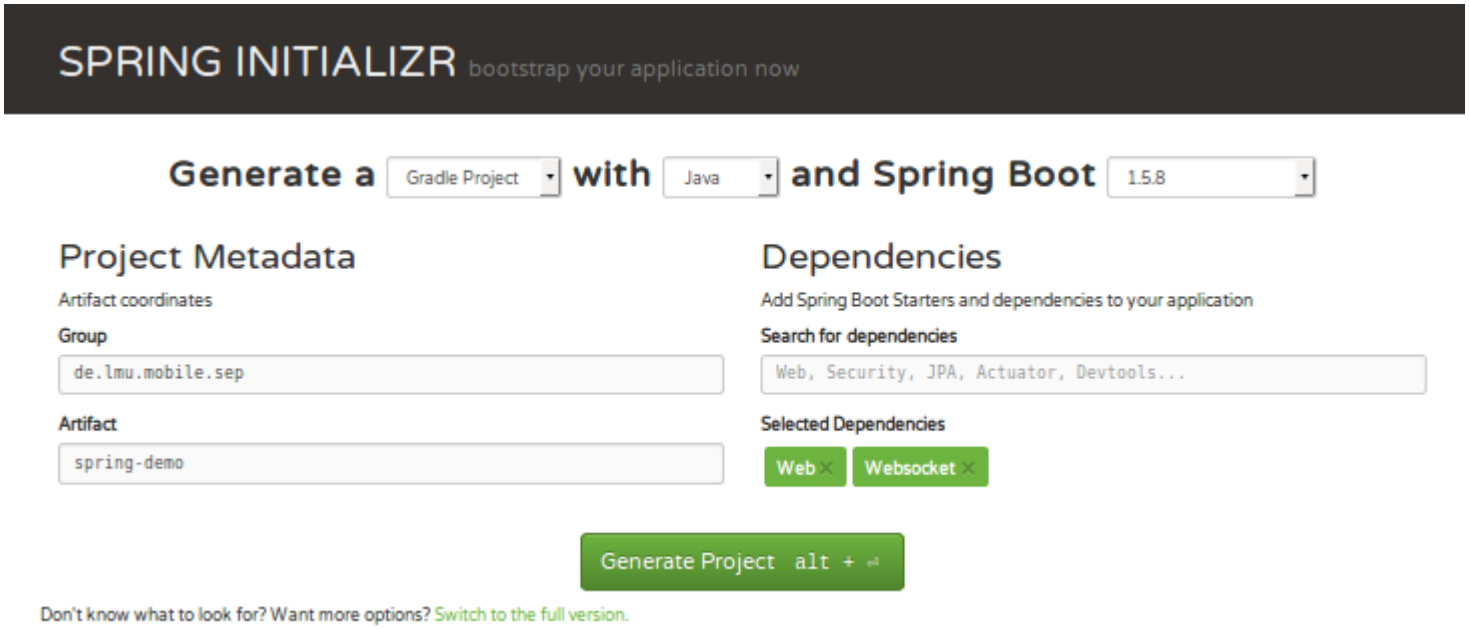
Running **gradle build** has the effect of:

```
> 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

- 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

```
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
```

- 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

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

public class MyHelloObject {

    private final String message;

    public MyHelloObject(String message) { this.message = message; }

    public String getMessage() {
        return message;
    }
}
```

DemoController.java

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

import ...

@RestController
public class DemoController{

    private static final Logger logger = LoggerFactory.getLogger(DemoController.class);

    @RequestMapping("/hello-object")
    public MyHelloObject myHelloObjectMethod(){
        return new MyHelloObject("I am a java object");
    }

    @RequestMapping("/hello-receive-object")
    public MyHelloObject myHelloReceiveObjectMethod(@RequestParam("hello") MyHelloObject myObject){
        // write log
        logger.info("We received a "+myObject.getClass()+ " object with message "+myObject.getMessage());
        // echo it back to the client
        return myObject;
    }
}
```

- 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
    }
}
```

Frameworks and BaaS

Frameworks

Backend As A Service

Node.js – Hello World

Once you have installed Node, let's try building our first web server. Create a file named "app.js", and paste the following code:

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

After that, run your web server using `node app.js`, visit http://localhost:3000, and you will see a message 'Hello World'

Node.js – Express Framework

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

Express Generator

- With the Express Generator, you can quickly bootstrap websites/-services.

```
$ npm install express-generator -g
```

```
$ express --view=pug myapp

   create : myapp
   create : myapp/package.json
   create : myapp/app.js
   create : myapp/public
   create : myapp/public/javascripts
   create : myapp/public/images
   create : myapp/routes
   create : myapp/routes/index.js
   create : myapp/routes/users.js
   create : myapp/public/stylesheets
   create : myapp/public/stylesheets/style.css
   create : myapp/views
   create : myapp/views/index.pug
   create : myapp/views/layout.pug
   create : myapp/views/error.pug
   create : myapp/bin
   create : myapp/bin/www
```

Express Routing

- RESTful routes can easily be created with Express

```
app.get('/', function (req, res) {
  res.send('Hello World!');
});
```

```
app.put('/user', function (req, res) {
  res.send('Got a PUT request at /user');
});
```

```
app.post('/', function (req, res) {
  res.send('Got a POST request');
});
```

```
app.delete('/user', function (req, res) {
  res.send('Got a DELETE request at /user');
});
```

Integrating socket.io

- Socket.IO is composed of two parts:
  - A server that integrates with the Node.JS HTTP Server: socket.io
  - A client library that loads on the browser side: socket.io-client

Server

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
  res.sendFile(__dirname + '/index.html');
});

io.on('connection', function(socket){
  console.log('an user connected');
});

http.listen(3000, function(){
  console.log('listening on *:3000');
});
```

Integrating socket.io

Client – connect to socket

```
import com.github.nkzawa.socketio.client.IO;
import com.github.nkzawa.socketio.client.Socket;

private Socket mSocket;
{
    try {
        mSocket = IO.socket("http://chat.socket.io");
    } catch (URISyntaxException e) {}
}

@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    mSocket.connect();
}
```

Client send message

```
private EditText mInputMessageView;

private void attemptSend() {
    String message = mInputMessageView.getText().toString().trim();
    if (TextUtils.isEmpty(message)) {
        return;
    }

    mInputMessageView.setText("");
    mSocket.emit("new message", message);
}
```

Frameworks and BaaS

Frameworks

Backend As A Service

Google Firebase - BaaS

Authentication

Hosting

Cloud Storage

Cloud Firestore

Realtime Database

Cloud Functions

ML Kit BETA

Examples (1)

Authentication – Create a User

```
mAuth.createUserWithEmailAndPassword(email, password)
        .addOnCompleteListener(this, new OnCompleteListener<AuthResult>() {
            @Override
            public void onComplete(@NonNull Task<AuthResult> task) {
                if (task.isSuccessful()) {
                    // Sign in success, update UI with the signed-in user's information
                    Log.d(TAG, "createUserWithEmail:success");
                    FirebaseUser user = mAuth.getCurrentUser();
                    updateUI(user);
                } else {
                    // If sign in fails, display a message to the user.
                    Log.w(TAG, "createUserWithEmail:failure", task.getException());
                    Toast.makeText(EmailPasswordActivity.this, "Authentication failed.",
                            Toast.LENGTH_SHORT).show();
                    updateUI(null);
                }

                // ...
            }
        });
```

Examples (1)

Authentication – Sign in a User

```
mAuth.signInWithEmailAndPassword(email, password)
        .addOnCompleteListener(this, new OnCompleteListener<AuthResult>() {
            @Override
            public void onComplete(@NonNull Task<AuthResult> task) {
                if (task.isSuccessful()) {
                    // Sign in success, update UI with the signed-in user's information
                    Log.d(TAG, "signInWithEmail:success");
                    FirebaseUser user = mAuth.getCurrentUser();
                    updateUI(user);
                } else {
                    // If sign in fails, display a message to the user.
                    Log.w(TAG, "signInWithEmail:failure", task.getException());
                    Toast.makeText(EmailPasswordActivity.this, "Authentication failed.",
                            Toast.LENGTH_SHORT).show();
                    updateUI(null);
                }

                // ...
            }
        });
```

Documentation

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/

Useful links

- Node.js
  - https://nodejs.org/en/docs/
- ExpressJS
  - https://expressjs.com/de/4x/api.html
- Android Volley
  - https://developer.android.com/training/volley
- socket.io
  - https://socket.io/blog/native-socket-io-and-android/
- Firebase
  - https://firebase.google.com/docs/