

# Verteilte Systeme 2 Lab

Konstruktion Web-basierter Systeme

[christian.zirpins@hs-karlsruhe.de](mailto:christian.zirpins@hs-karlsruhe.de)

Messaging mit WebSockets / Redis



Hochschule Karlsruhe  
Technik und Wirtschaft

UNIVERSITY OF APPLIED SCIENCES



# Projektplan Winter 2017

<i>Termin</i>	<i>Laboraufgabe</i>	<i>Meetup</i>	<i>Seminarteil</i>	<i>Abgabe</i>	<i>Aufwand (Stunden)</i>	
					<i>Präsenz</i>	<i>Eigenst.</i>
12.10	<b>A1</b> Konzeption und Gestaltung		Web Frameworks/Spring		2	1
19.10			Persistenz/Redis		2	1
26.10					1	1
2.11					1	1
9.11		Diskussion A1		Entwürfe	1	
16.11	<b>A2</b> Seitenbasierte Implementierung				1	1
23.11					1	1
30.11					1	1
7.12					1	1
14.12		Diskussion A2		System V1	1	
21.12	<b>A3</b> Asynchrone Erweiterungen		Messaging/WebSockets		1	1
11.1					1	1
18.1					1	1
25.1		Demo Day		System V2	1	
					16	11

# Aufgabe

## Social Media Anwendung

- Microblogging ('Twitter-Klon')

## Funktionale Anforderungen

- Verwaltung von Nutzerkonten
- Nutzer suchen, folgen
- Posts schreiben, Timelines lesen
- **Push Benachrichtigungen**

## Nicht-funktionale Anforderungen

- Sicherheit durch Login/ Sessions
- **Benachrichtigungen in Echtzeit**
- **Replikationstransparenz (Webserver)**
- Mobilgerätetauglichkeit

## Welcome to the LKIT Microblog

Please login or register

### Login with existing account.

Username:

Password:

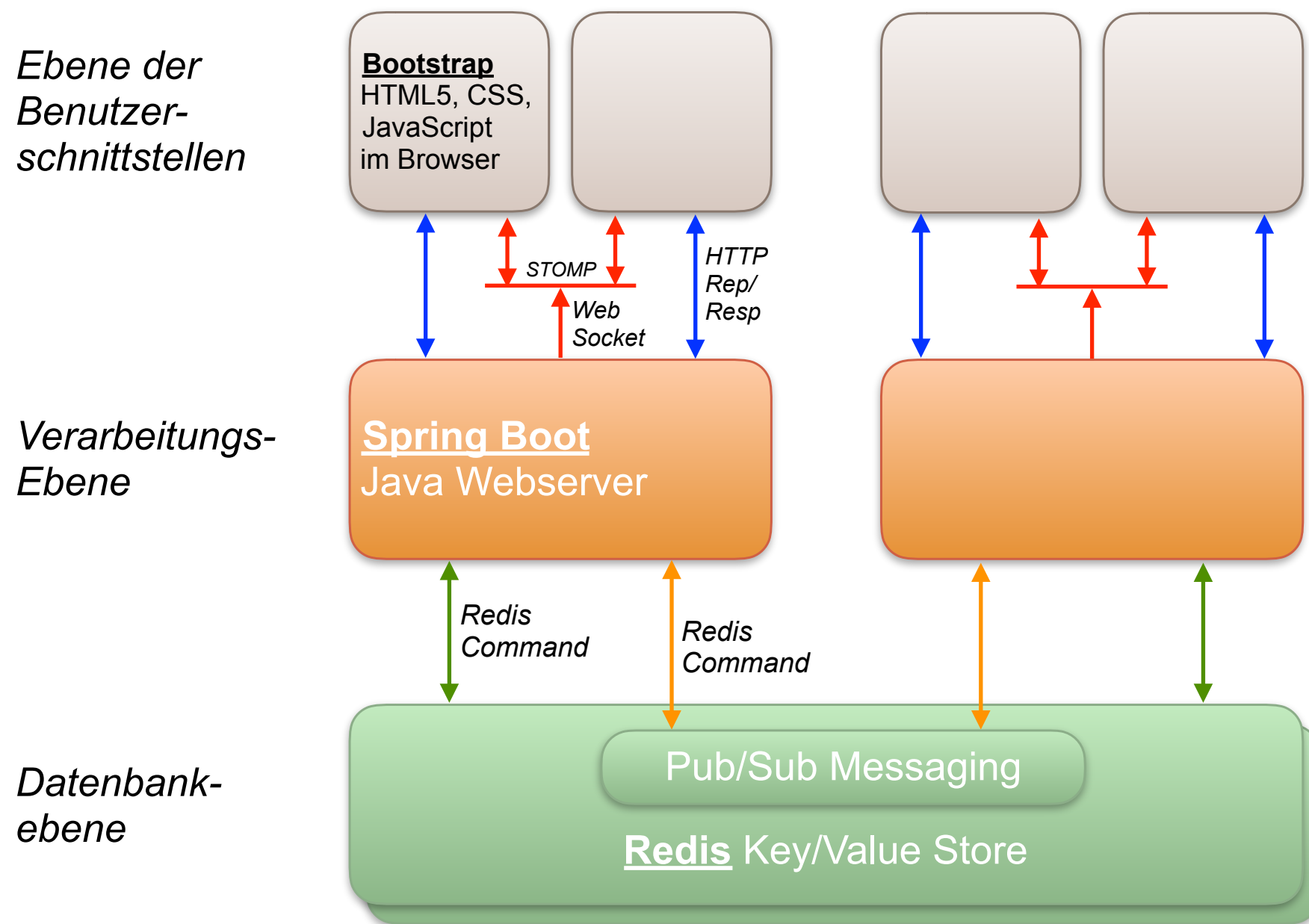
### Create new account.

Username:

Password:

© 2016 HsKA, LKIT

# Zielarchitektur



↕	Anforderung von Seiten über HTTP
↕	Datenbank-Queries mit Redis Key/Value Kommandos
↕	Push Updates per STOMP Nachrichten über WebSockets
↕	Update Events über Redis Pub/Sub Kommandos

# Teilaufgaben Winter 2017

## A1 Konzeption und Gestaltung (12.10 bis 9.11)

- Analyse der Anforderungen (Use Case Diagramm oder textuell)
- Entwurf der logische Seitenstruktur/Navigation (Zustandsdiagramm)
- Erstellen von Mockups (Spring Boot Projekt)
- Entwurf des Datenmodells (Redis Datenstrukturen und Key-Muster)

## A2 Seitenbasierte Implementierung (16.11 bis 14.12)

- Erstellung von Repositories und Services
- Realisierung von Authentifizierung und Sessions
- Konstruktion von Controllern und Templates
- Clientseitige Programmierung von Layouts und Interaktionen

## A3 Asynchrone Erweiterungen (21.12 bis 25.1)

- Realisierung von Client Push-Messages mit WebSockets
- Implementierung von Server Pub/Sub Messaging mit Redis

# Struktur — Messaging mit WebSockets / Redis

## WebSockets

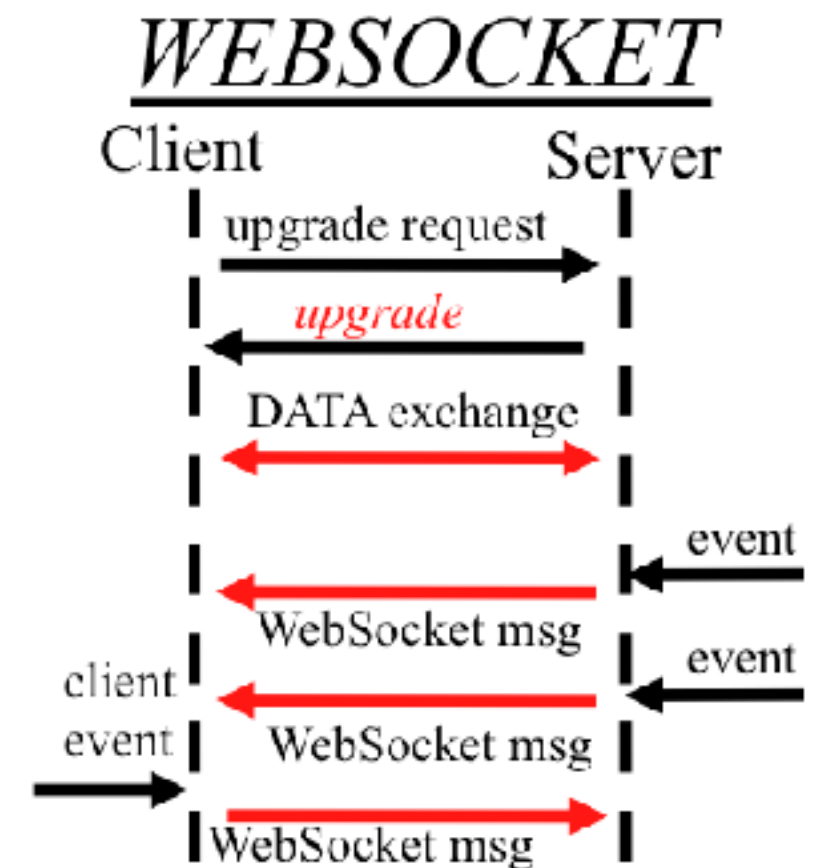
- WebSockets Überblick
- WebSockets/STOMP mit Spring

## Redis Pub/Sub

- Redis Pub/Sub Mechanismus
- Spring Beispiel

# WebSockets Überblick

- Das **WebSocket-Protokoll** (RFC 6455) erlaubt **bidirektionale Verbindungen** zwischen einem Browser und einem Web Server.
- WebSockets beinhalten ein **Eröffnungsprotokoll**, das, abwärtskompatibel zu HTTP, ein **Protocol Upgrade** durchführt.
- Die Kommunikation setzt auf einer TCP-Verbindung auf.
- Nach Öffnung der Verbindung durch den Client, kann der **Server dem Browser auch ohne vorherige Anforderung Daten schicken**.
- Dies ersetzt die sonst üblichen Verfahren für Zwei-Wege-Kommunikation, z.B. XMLHttpRequest und Long Polling in HTTP.



# STOMP over WebSockets

- WebSockets bilden nur eine sehr dünne Schicht über TCP.
  - Es bietet sich an, Nachrichten per **Sub-Protokoll** einzubetten.
- Eine mögliche Variante ist STOMP.

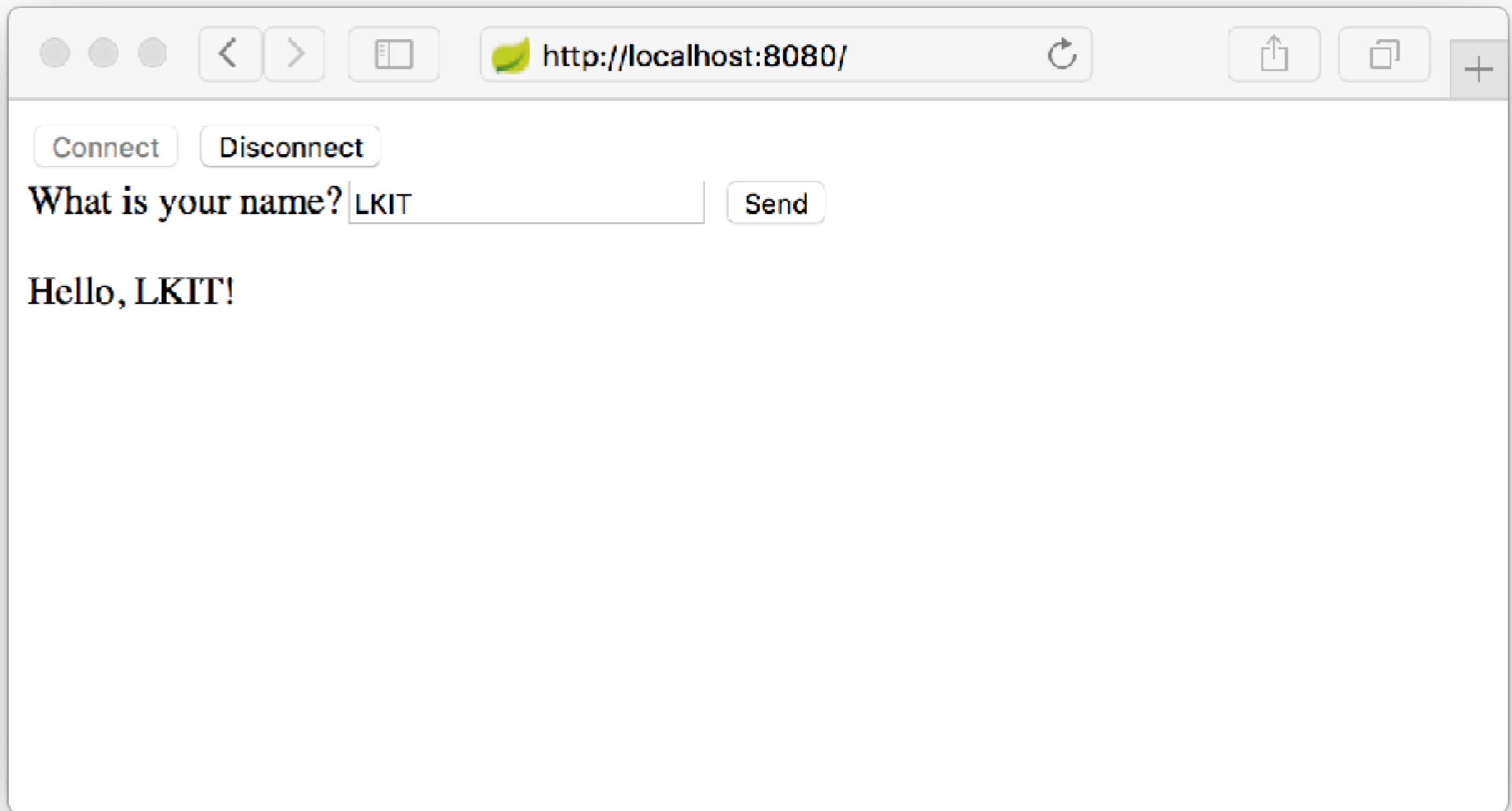
## STOMP (Simple/Streaming Text Orientated Messaging Protocol)<sup>1</sup>

- STOMP ist ein einfaches textorientiertes Messaging Protokoll<sup>2</sup>.
- Es definiert ein interoperables Übertragungsformat, mit dem beliebige Clients und Message Broker kommunizieren können.
- Ziel ist Messaging Interoperabilität über Sprachen und Plattformen.

<sup>1</sup>[Stomp 2016a] <sup>2</sup>[Mesnil 2016a]



# Spring Beispiel: Hello-Service



# Hello-Service — Messages

- Der Service empfängt STOMP Nachrichten per WebSocket Verbindung.
  - Das enthaltene JSON Objekt wird als **HelloMessage** POJO repräsentiert.
- Dann schickt der Service eine STOMP Nachricht per WebSocket zurück.
  - Die Antwort wird als **Greeting** POJO repräsentiert.
  - Sie geht an eine zweite STOMP Queue, die der Client abonniert hat.

```
package hello;

public class Greeting {

    private String content;

    public Greeting(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }

}
```

```
package hello;

public class HelloMessage {

    private String name;

    public String getName() {
        return name;
    }

}
```

# Hello-Service — Message Controller

- Spring leitet STOMP Messages an @Controller Klassen weiter.
  - GreetingController empfängt Nachrichten an "/hello".
- Gleichzeitig sendet die Klasse Nachrichten an "/topic/greetings".

```
package hello;

import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.stereotype.Controller;

@Controller
public class GreetingController {

    @MessageMapping("/hello")
    @SendTo("/topic/greetings")
    public Greeting greeting>HelloMessage message) throws Exception {
        Thread.sleep(3000); // simulated delay
        return new Greeting("Hello, " + message.getName() + "!");
    }

}
```

# Hello-Service — Messaging Template

- Jede Komponente der Anwendung kann Nachrichten über den Message Broker verschicken.
  - `SimpMessagingTemplate` implementiert dazu Methoden zum Versenden von Nachrichten über einfache Messaging Protokolle.
- Eine Variante des `GreetingController` könnte damit so aussehen:

```
@Controller
public class GreetingController2 {

    @Autowired
    private SimpMessagingTemplate msgTemplate;

    @RequestMapping("/hello")
    public void greeting>HelloMessage message) throws Exception {
        Thread.sleep(2000); // simulated delay
        msgTemplate.convertAndSend("/topic/greetings",
                                   new Greeting("Another hello, " + message.getName() + "!!"));
    }
}
```

# Hello-Service — WebSockets Configuration

- Eine `@Configuration` mit `@EnableWebSocketMessageBroker` Annotation aktiviert WebSocket Messaging über einen Broker.
- `configureMessageBroker()` erstellt per `enableSimpleBroker()` einen In-Memory-Broker mit `"/topic"`-Prefix, der Greetings ausliefert.
  - Nachrichten an den `"/app"`-Prefix werden durch Controller verarbeitet.
- `registerStompEndpoints()` registriert den `"/mysocket"` Endpunkt zum initialen Aufbau der Verbindung.

```
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic");
        config.setApplicationDestinationPrefixes("/app");
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/mysocket").withSockJS();
    }
}
```

# Hello-Service — Webseite

- Ein JavaScript client implementiert die STOMP-over-WebSocket Kommunikation über die Bibliotheken **sockJS**<sup>1</sup> und **stomp.js**<sup>2</sup>

```
<!DOCTYPE html>
<html>
<head>
<title>Hello WebSocket</title>
<script src="sockjs-0.3.4.js"></script>
<script src="stomp.js"></script>
<script src="hello-client.js"></script>
<body onload="disconnect()">
  <div>
    <div>
      <button id="connect" onclick="connect();">Connect</button>
      <button id="disconnect" disabled="disabled" onclick="disconnect();">Disconnect</button>
    </div>
    <div id="conversationDiv">
      <label>What is your name?</label><input type="text" id="name" />
      <button id="sendName" onclick="sendName();">Send</button>
      <p id="response"></p>
    </div>
  </div>
</body>
</html>
```

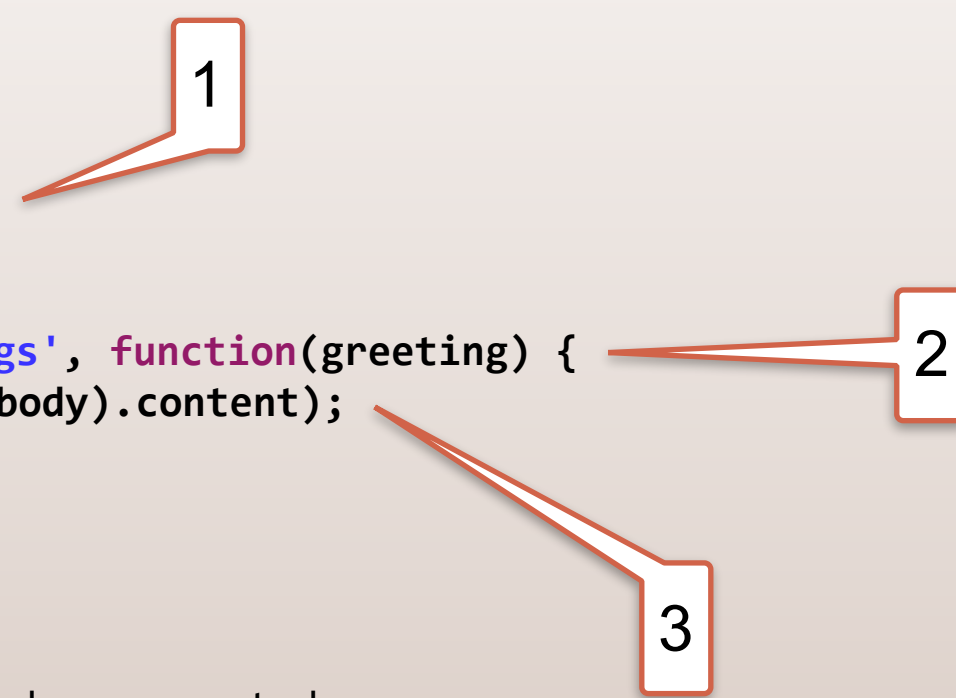
# Hello-Service — hello-client.js 1/2

1. `connect()` öffnet eine WebSocket Verbindung zum `"/mysocket"` Endpunkt (wo `GreetingController` auf Serverseite wartet).
2. Dann wird `"/topic/greetings"` abonniert, wo Greetings ankommen.
3. Bei Empfang wird ein Paragraph mit dem Greeting an das DOM gehängt.

```
var stompClient = null;

function connect() {
  var socket = new SockJS('/mysocket');
  stompClient = Stomp.over(socket);
  stompClient.connect({}, function(frame) {
    setConnected(true);
    console.log('Connected: ' + frame);
    stompClient.subscribe('/topic/greetings', function(greeting) {
      showGreeting(JSON.parse(greeting.body).content);
    });
  });
}

function setConnected(connected) {
  document.getElementById('connect').disabled = connected;
  document.getElementById('disconnect').disabled = !connected;
  document.getElementById('conversationDiv').style.visibility = connected ? 'visible' : 'hidden';
  document.getElementById('response').innerHTML = '';
}
```



## Hello-Service — hello-client.js 2/2

- **sendName()** entnimmt einen Namen aus dem Input-Element und sendet ihn per STOMP Client an **"/app/hello"** (wo ihn schließlich **GreetingController.greeting()** empfängt).

```
function sendName() {
    var name = document.getElementById('name').value;
    stompClient.send("/app/hello", {}, JSON.stringify({
        'name' : name
    }));
}

function showGreeting(message) {
    var response = document.getElementById('response');
    var p = document.createElement('p');
    p.style.wordWrap = 'break-word';
    p.appendChild(document.createTextNode(message));
    response.appendChild(p);
}

function disconnect() {
    if (stompClient != null)
        stompClient.disconnect();
    setConnected(false);
}
```



# Struktur — Messaging mit WebSockets / Redis

## WebSockets

- WebSockets Überblick
- WebSockets/STOMP mit Spring

## Redis Pub/Sub

- Redis Pub/Sub Mechanismus
- Spring Beispiel

# Redis Pub/Sub

- **Redis** ist nicht nur NoSQL Store, sondern auch **Messaging System**.
- Mit **Spring Data Redis** kann man **Nachrichten publizieren oder abonnieren**, die per Redis gesendet werden.
- Es muss ein **Redis Server** installiert und gestartet werden, der Empfang und Versenden von Nachrichten übernimmt.

## Simple Beispiel

- Die Beispielanwendung verwendet das `StringRedisTemplate` um eine String Nachricht zu publizieren.
- Diese Nachrichten werden gleichsam wieder per POJO mit dem `MessageListenerAdapter` abonniert.

# Receiver

- Der Receiver ist ein POJO mit einer Methode um Nachrichten zu empfangen. Die Methode kann beliebig benannt werden.

```
public class Receiver {  
    private static final Logger LOGGER = LoggerFactory.getLogger(Receiver.class);  
  
    private CountdownLatch latch;  
  
    @Autowired  
    public Receiver(CountDownLatch latch) {  
        this.latch = latch;  
    }  
  
    public void receiveMessage(String message) {  
        LOGGER.info("Received <" + message + ">");  
        latch.countDown();  
    }  
}
```

*"A synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes. ... The await methods block until the current count reaches zero due to invocations of the countDown() method"*

# Redis Application 1/2

1. Die Bean aus `listenerAdapter()` wird in der Bean aus `container()` als Nachrichtenempfänger für das "chat" Thema registriert.
2. `container()` nutzt eine standard `RedisConnectionFactory`.
3. `listenerAdapter()` kapselt das Receiver POJO als `MessageListener`.

```
@SpringBootApplication
public class Application {

    @Bean
    RedisMessageListenerContainer container(RedisConnectionFactory connectionFactory,
        MessageListenerAdapter listenerAdapter) {
        RedisMessageListenerContainer container = new RedisMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);
        container.addMessageListener(listenerAdapter, new PatternTopic("chat"));
        return container;
    }

    @Bean
    MessageListenerAdapter listenerAdapter(Receiver receiver) {
        return new MessageListenerAdapter(receiver, "receiveMessage");
    }

    @Bean
    Receiver receiver(CountDownLatch latch) {
        return new Receiver(latch);
    }
}
```

213

# Redis Application 2/2

- Um Nachrichten zu senden wird ein **RedisTemplate** benötigt.
- **main()** holt **StringRedisTemplate** Bean (1) aus **ApplicationContext** (2) und nutzt diese, um Nachricht an "chat" zu senden (3).

```
@Bean
CountDownLatch latch() {
    return new CountDownLatch(1);
}

@Bean
StringRedisTemplate template(RedisConnectionFactory connectionFactory) {
    return new StringRedisTemplate(connectionFactory);
}

public static void main(String[] args) throws InterruptedException {
    ApplicationContext ctx = SpringApplication.run(Application.class, args);

    StringRedisTemplate template = ctx.getBean(StringRedisTemplate.class);

    CountDownLatch latch = ctx.getBean(CountDownLatch.class);

    template.convertAndSend("chat", "Hello from Redis!");

    latch.await();
    System.exit(0);
}
```

1

2

3

# Nächste Aufgabe

## A3 Asynchrone Erweiterungen (21.12 bis 25.1)

- Realisierung von **Client Push-Messages** mit WebSockets
- Implementierung von **Server Pub/Sub Messaging** mit Redis

**Hinweis:** vollständige Spezifikation und Aufgabenstellung als PDF im ILIAS

# Zum Nach- und Weiterlesen

## Aus dem Web

### WebSockets

[Spring 2016a] I. Fette et al. *"The WebSocket Protocol"*, <https://tools.ietf.org/html/rfc6455>

[Spring 2016a] Spring Framework Reference: *WebSocket Support*, <http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/#websocket>

[Spring 2016b] Spring Getting Started Guides, *"Using WebSocket to build an interactive web application"*, <https://spring.io/guides/gs/messaging-stomp-websocket/>

[Stomp 2016a] *The Simple Text Oriented Messaging Protocol*, <https://stomp.github.io>

[Mesnil 2016a] J. Mesnil, *Stomp Over WebSocket* <http://www.jmesnil.net/stomp-websocket/doc/>

[sockJS 2016a] *WebSocket emulation - Javascript client* <http://sockjs.org>

### Redis Pub/Sub

[Redis 2016a] *Redis Pub/Sub*, <http://redis.io/topics/pubsub>

[Spring 2016c] C. Leau et al. *"Spring Data Redis: 5.8. Redis Messaging/PubSub"*, <http://docs.spring.io/spring-data/redis/docs/1.7.1.RELEASE/reference/html/#pubsub>

[Spring 2016c] Spring Getting Started Guides, *"Messaging with Redis"*, <https://spring.io/guides/gs/messaging-redis/>