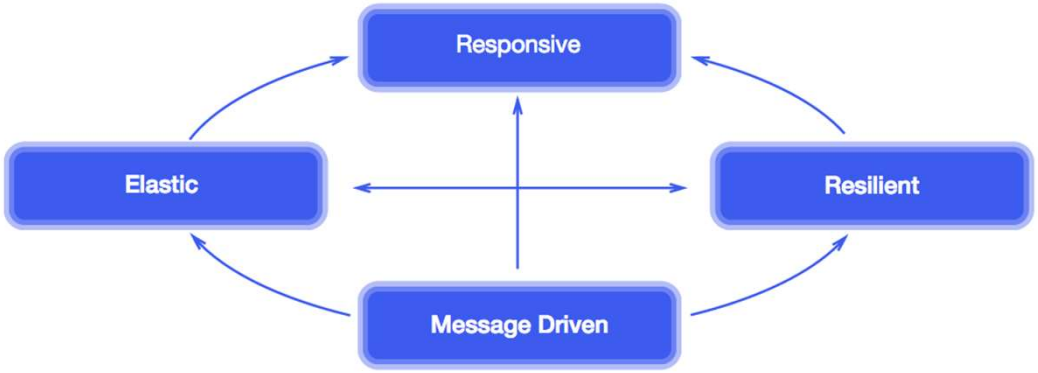
Lesson 6

WEBFLUX WEBSOCKETS

SPRING WEBFLUX

Reactive applications

- Non-blocking
- Event driven



- Implementations
  - Reactive Streams
  - JavaRx (Netflix)
  - Reactor (Pivotal)
    - Used by Spring: Spring webflux

Spring webflux library

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

This will add the embedded Netty container which support reactive web

Reactor

- Mono<T> : for handling 0 or 1 element
- Flux<T> : for handling N elements
- You can subscribe to a Mono or a Flux
  - Run some code when an object arrives in the Mono or Flux

Mono

```
public class SpringReactiveClientApplication {

  public static void main(String[] args) throws InterruptedException {
    System.out.println(LocalDateTime.now());
    Mono<String> mono = Mono.just("Frank")
                            .delayElement(Duration.ofSeconds(5));

    mono.subscribe(s->printName(s));

    Thread.sleep(10000);
  }

  public static void printName(String name) {
    System.out.print(LocalDateTime.now()+" : ");
    System.out.println(name);
  }
}
```

Add the name to the mono after 5 seconds

Whenever the name arrives in the mono, print it out (Callback method)

Wait until the name has arrived in the mono

Callback method

```
2018-03-25T18:46:25.942
2018-03-25T18:46:31.155 : Frank
```

Flux

```
public class SReactiveApplication {

  public static void main(String[] args) throws InterruptedException {
    Flux<String> flux = Flux.just("Walter", "Skyler",  "Saul",  "Jesse")
                            .delayElements(Duration.ofSeconds(3));

    flux.subscribe(s->printName(s));
    Thread.sleep(15000);
  }


  public static void printName(String name) {
    System.out.print(LocalDateTime.now()+" : ");
    System.out.println(name);
  }
}
```

Add every 3 seconds a name to the flux

Whenever a name arrives in the flux, print it out (Callback method)

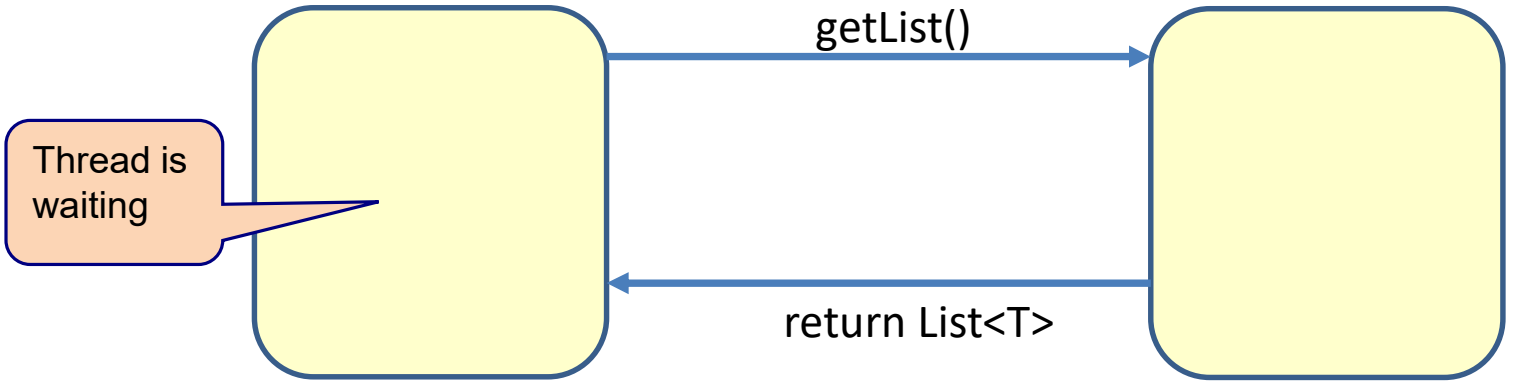
Wait until all names have arrived in the flux

Callback method

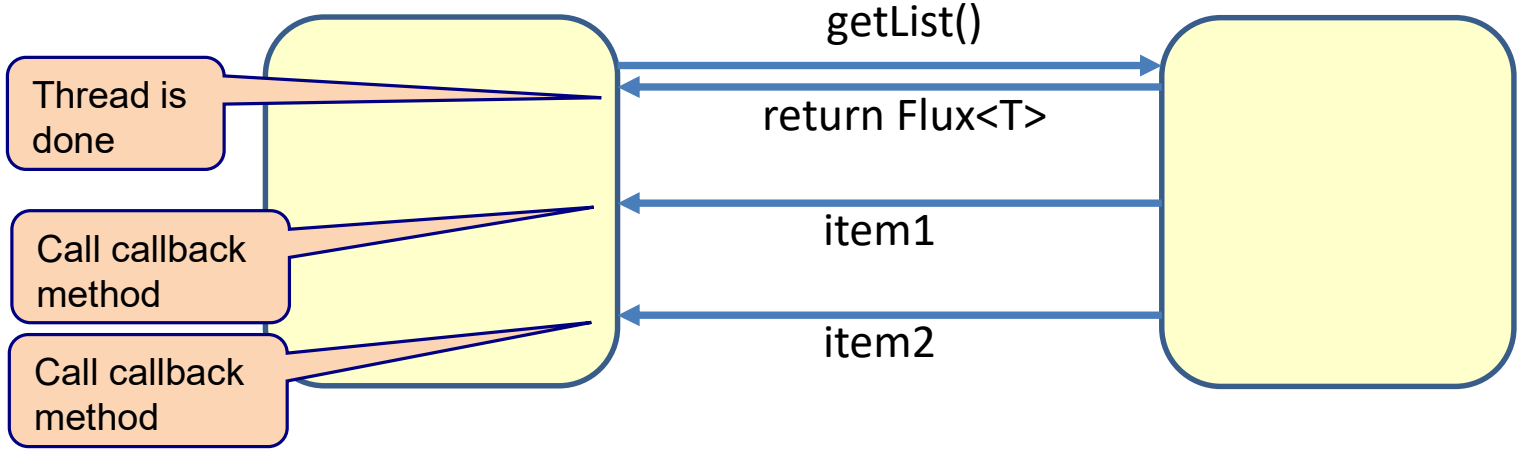
```
2018-03-25T18:37:38.481 : Walter
2018-03-25T18:37:41.484 : Skyler
2018-03-25T18:37:44.485 : Saul
2018-03-25T18:37:47.486 : Jesse
```

Imperative versus reactive

- Synchronous, blocking



- Asynchronous, non-blocking



Reactive systems

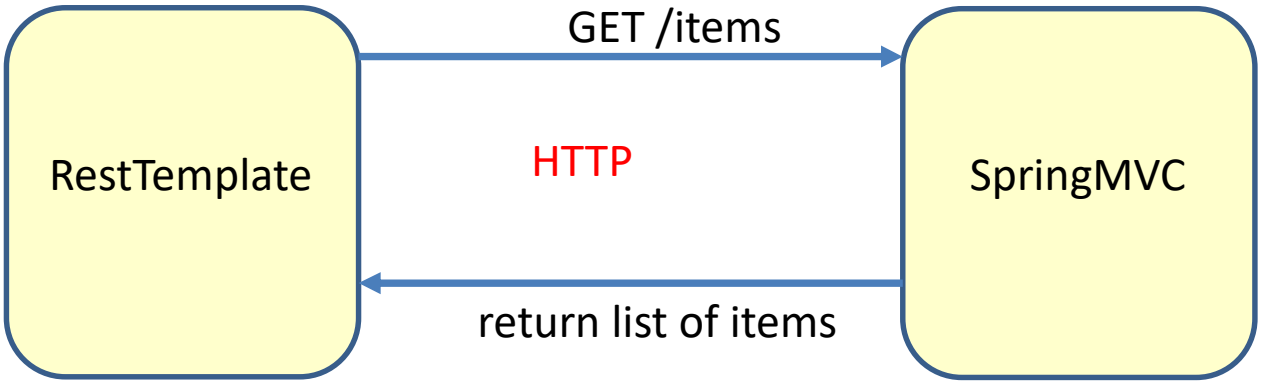
- Advantage
  - Performance
    - No need to wait till all results are available
  - Scaling
    - Less threads needed
- Disadvantage
  - The whole calling stack needs to be reactive
    - Client <->controller<->data access
  - Harder to debug

Spring WebFlux

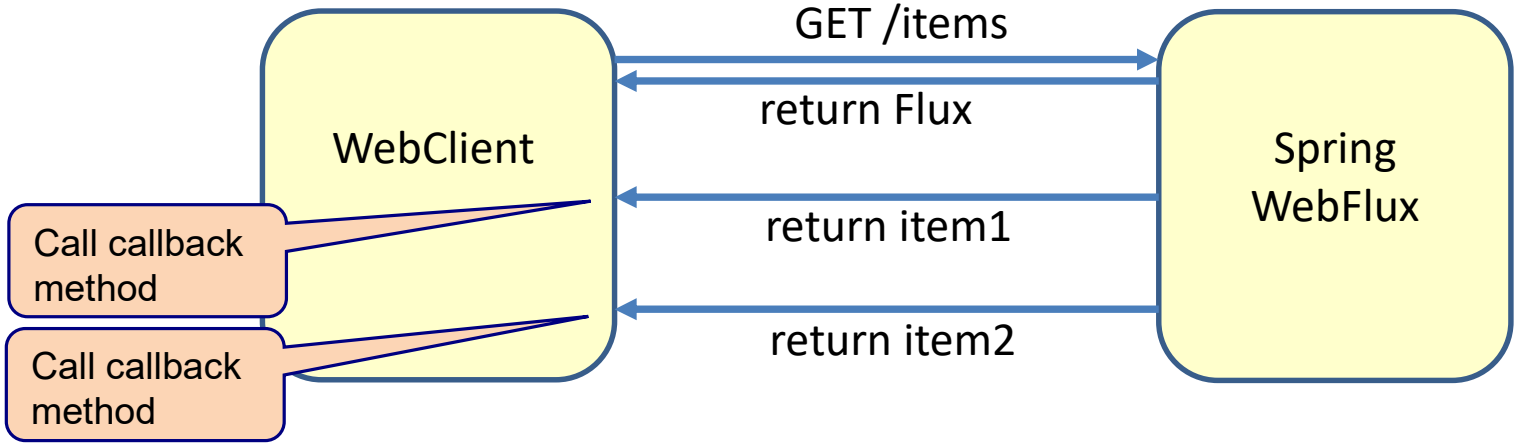
- Allows to build reactive web(REST) applications
- Uses Netty as embedded webserver

Reactive Web

- Synchronous, blocking



- Asynchronous, non-blocking



Reactive REST service

```
@RestController
public class CustomerController {

  @GetMapping(value="/customers", produces=MediaType.TEXT_EVENT_STREAM_VALUE)
  public Flux<Customer> getAllCustomers() {
    Flux<Customer> customerFlux = Flux.just(
      new Customer(new Long(1), "Walter", "White", 29),
      new Customer(new Long(2), "Skyler", "White", 24),
      new Customer(new Long(3), "Saul", "Goodman", 27),
      new Customer(new Long(4), "Jesse", "Pinkman", 24)
    ).delayElements(Duration.ofSeconds(3));
    return customerFlux;
  }
}
```

Generate a new Customer every 3 seconds

```
public class Customer {

  private long custId;
  private String firstname;
  private String lastname;
  private int age;
  ...
}
```

```
@SpringBootApplication
public class SpringReactiveApplication{

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

Reactive REST Client

```
@SpringBootApplication
public class ClientApplication {

  public static void main(String[] args) throws InterruptedException{
    Flux<Customer> result = WebClient.create("http://localhost:8080/customers")
      .get()
      .retrieve()
      .bodyToFlux(Customer.class);
    result.subscribe(s->{
      System.out.print(LocalDateTime.now()+" : ");
      System.out.println(s);
    });

    Thread.sleep(15000);
  }
}
```

Print the customer when it arrives

Wait for all Customers to arrive

```
2018-03-25T18:26:27.107 : custId = 1, firstname = Walter, lastname = White, age = 29
2018-03-25T18:26:27.109 : custId = 2, firstname = Skyler, lastname = White, age = 24
2018-03-25T18:26:29.986 : custId = 3, firstname = Saul, lastname = Goodman, age = 27
2018-03-25T18:26:32.991 : custId = 4, firstname = Jesse, lastname = Pinkman, age = 24
```

Data access

- Spring Data JPA is not reactive yet
- Spring Data Mongo is reactive

```
interface ReactiveCrudRepository<T, ID> extends Repository<T, ID> {
    <S extends T> Mono<S> save(S entity);
    <S extends T> Flux<S> saveAll(Iterable<S> entities);
    <S extends T> Flux<S> saveAll(Publisher<S> entityStream);
    Mono<T> findById(ID id);
    Mono<T> findById(Mono<ID> id);
    Mono<Boolean> existsById(ID id);
    Mono<Boolean> existsById(Mono<ID> id);
    Flux<T> findAll();
    Flux<T> findAllById(Iterable<ID> ids);
    Flux<T> findAllById(Publisher<ID> idStream);
    Mono<Long> count();
    Mono<Void> deleteById(ID id);
    Mono<Void> delete(T entity);
    Mono<Void> deleteAll(Iterable<? extends T> entities);
    Mono<Void> deleteAll(Publisher<? extends T> entityStream);
    Mono<Void> deleteAll();
```

Reactive REST +Mongo

```
@Repository
public interface PersonRepository extends ReactiveCrudRepository<Person, String>{
  @Tailable
  Flux<Person> findByJob(String job);
}
```

Tailable

```
@Document
public class Person {
    @Id
    private String id;
    private String name;
    private String job;

  ...
}
```

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>
```

Capped collections in MongoDB

- By default, MongoDB automatically closes a cursor when the client exhausts all results supplied by the cursor.
- For capped collections, you can use a Tailable Cursor that remains open after the client consumed all initially returned data.
- Capped collections are fixed-size collections that work in a way similar to circular buffers
  - Once a collection fills its allocated space, it makes room for new documents by overwriting the oldest documents in the collection.

Reactive REST+Mongo

```
@RestController
public class PersonController {
  @Autowired
  private PersonRepository personRepository;
  private int x = 10;

  @GetMapping(value="/persons", produces= MediaType.TEXT_EVENT_STREAM_VALUE)
  public Flux<Person> getAllPersons() {
    return personRepository.findByJob("Developer");
  }

  @Scheduled(fixedRate = 3000)
  private void savePerson() {
    personRepository.save(new Person(x + "", "Frank Brown" + x, "Developer")).block();
    x++;
  }

}
```

Save a new Person every 3 seconds

save() is returning a Mono, which is a Publisher, and it won't start working until you subscribe to it, or call block() on it

Reactive REST+Mongo

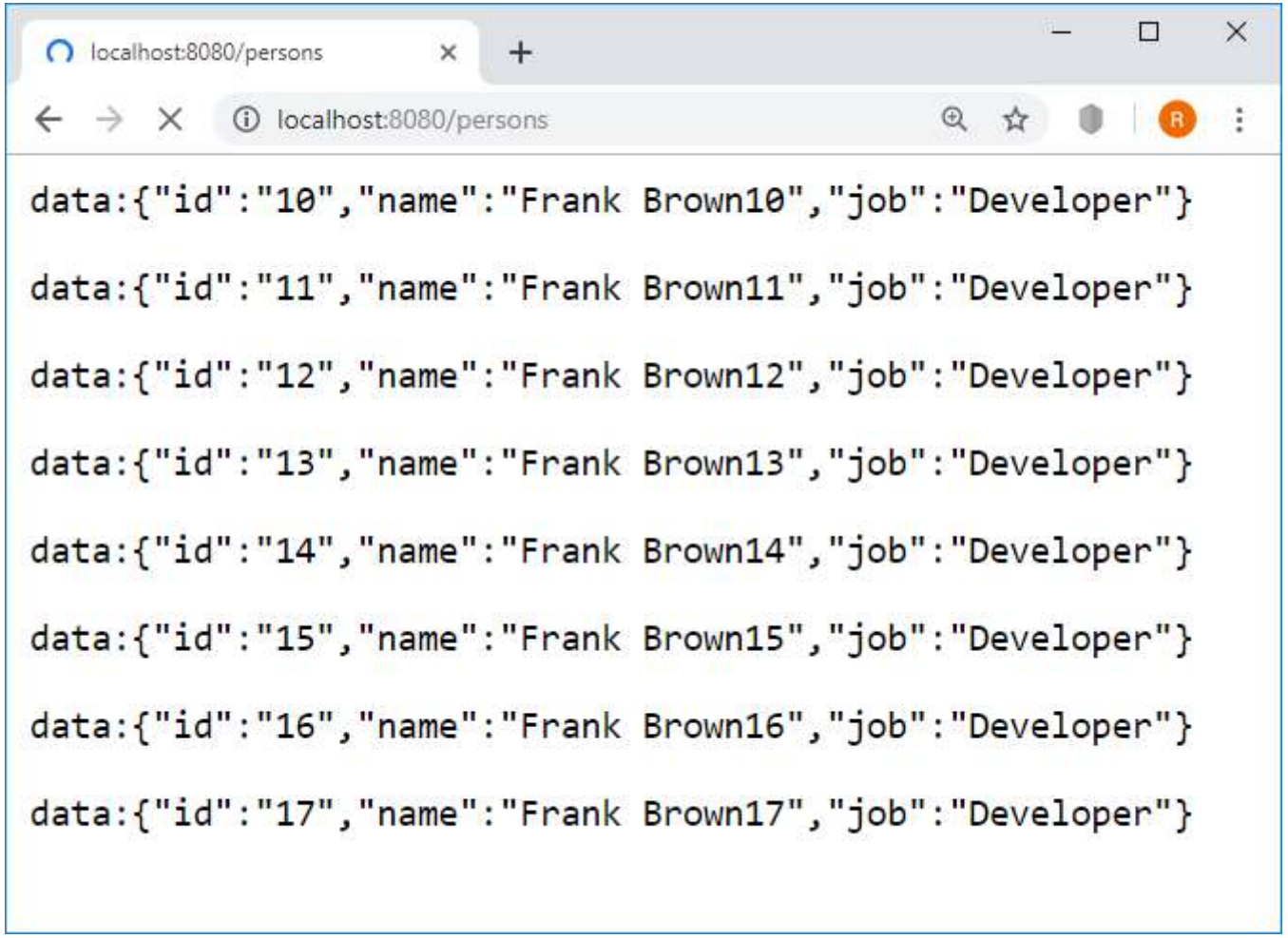
```
@SpringBootApplication
@EnableReactiveMongoRepositories
@EnableScheduling
public class SpringBootReactiveApplication {
  @Autowired
  private PersonRepository personRepository;
  @Autowired
  ReactiveMongoTemplate reactiveMongoTemplate;

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

  @PostConstruct
  public void init() {
    reactiveMongoTemplate.dropCollection("person").
      then(reactiveMongoTemplate.createCollection("person",
      CollectionOptions.empty().capped().size(2048).maxDocuments(10000))).subscribe();
  }
}
```

Make the person collection a capped collection

Reactive REST + Mongo



WEBSOCKETS

Websocket application

WebSocketConfig

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

  public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
    registry.addHandler(new SocketTextHandler(), "/user");
  }

}
```

/user endpoint

SocketHandler

```
@Component
public class SocketTextHandler extends TextWebSocketHandler {

  @Override
  public void handleTextMessage(WebSocketSession session, TextMessage message) throws Exception {
    System.out.println("got meesage"+ message);
    session.sendMessage(new TextMessage("Hi " + message.getPayload() + " how may we help you?"));
  }

  @Override
  public void afterConnectionEstablished(WebSocketSession session) throws Exception {
    super.afterConnectionEstablished(session);
    System.out.println("Connected");
    //send message back to the client
    session.sendMessage(new TextMessage("Connected !"));

    MyThread myThread = new MyThread(session);
    Thread t = new Thread(myThread);
    t.start();
  }

  @Override
  public void afterConnectionClosed(WebSocketSession session, CloseStatus status) throws Exception {
    super.afterConnectionClosed(session, status);
    System.out.println("Closed");
  }
}
```

Called when we receive a message

Send back to client

Start another thread

WebSocketConfig

```
public class MyThread implements Runnable {

   private WebSocketSession session;

   public MyThread(WebSocketSession session) {
      this.session = session;
   }

   public void run() {
      try {
         for (int x=0; x< 100; x++) {
            if (session.isOpen())
               session.sendMessage(new TextMessage("Server message nr "+ x));
            Thread.sleep(5000);
         }
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Executed when the thread is started

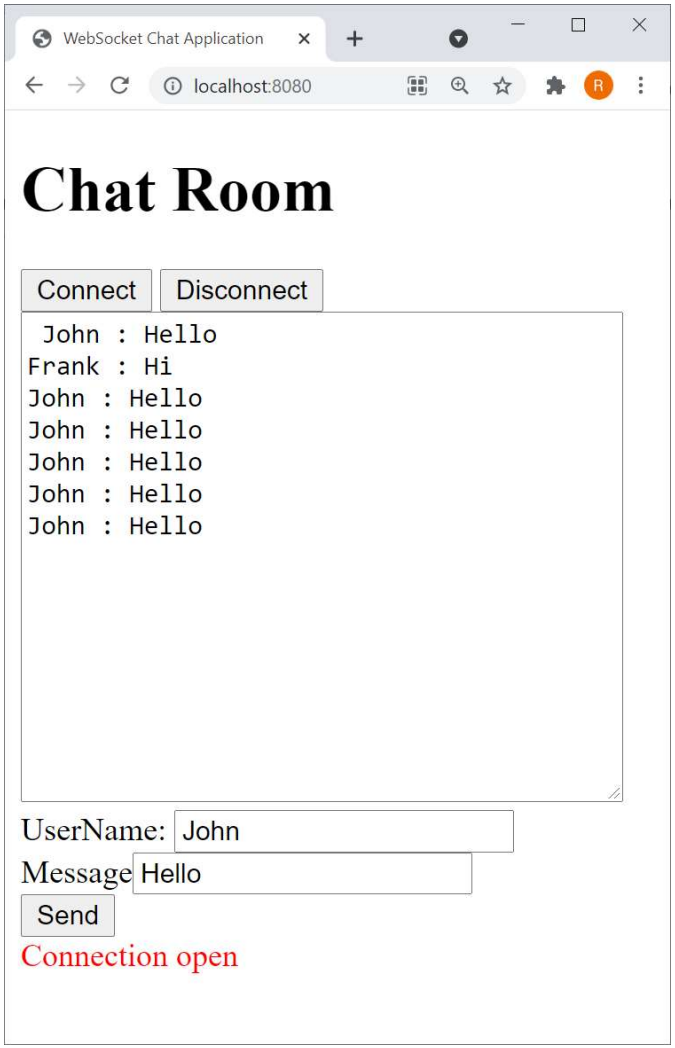
index.html

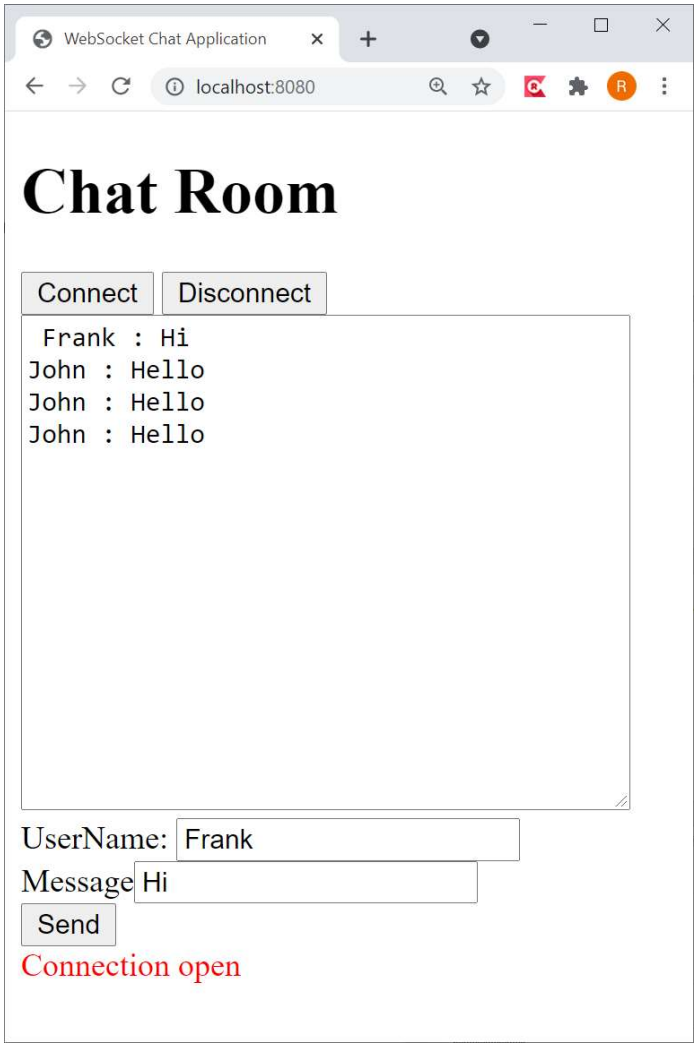
```
<!DOCTYPE html>
<html>
<head>
   <title>WebSocket Chat Application</title>
   <script type="text/javascript">
var ws;
function connect() {
  ws = new WebSocket("ws://localhost:8080/user");
  ws.onmessage = function(event) {
    showdata(event.data);
  }
   ws.onclose = function(event){
      showdata("Connection closed!");
   };
  helloworldmessage.innerHTML = "";
}
function disconnect() {
  if (ws != null) {
    ws.close();
  }
}
function sendData() {
  var text = document.getElementById("user").value;
  ws.send(text);
}
function showdata(message) {
  helloworldmessage.innerHTML += "<br/>" + message;
}
</script>
```

index.html

```
</head>
<body>
<div>
  <div>
    <div>
      <div>
        <button id="connect" type="button" onclick="connect();" >Connect</button>
        <button id="disconnect" type="button" onclick="disconnect();" >Disconnect</button>
      </div>
    </div>
  </div>
  <div>
    <div>
      <table id="chat">
        <thead>
        <tr>
          <th>Welcome user. Please enter you name</th>
        </tr>
        </thead>
        <tbody id="helloworldmessage">
        </tbody>
      </table>
    </div>
    <div>
      <div>
        <div>
          <textarea id="user" placeholder="Write your name here..." required></textarea>
        </div>
        <button id="send" type="button" onclick="sendData();">Send</button>
      </div>
    </div>
  </div>
</div>
</body>
</html>
```

Chat application





WebSocketConfig

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

  public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
    registry.addHandler(new SocketTextHandler(), "/chat");
  }

}
```

/chat endpoint

SocketHandler

```
@Component
public class SocketTextHandler extends TextWebSocketHandler {

  List<WebSocketSession> sessions = new ArrayList<WebSocketSession>();

  @Override
  public void handleTextMessage(WebSocketSession session, TextMessage message)
      throws Exception {
    System.out.println("got message"+ message);
    for (WebSocketSession theSession : sessions){
      if (theSession.isOpen())
        theSession.sendMessage(new TextMessage(message.getPayload()));
    }
  }

  @Override
  public void afterConnectionEstablished(WebSocketSession session) throws Exception {
    super.afterConnectionEstablished(session);
    System.out.println("Connected");
    sessions.add(session);
  }

  @Override
  public void afterConnectionClosed(WebSocketSession session, CloseStatus status) throws Exception {
    super.afterConnectionClosed(session, status);
    sessions.remove(session);
    System.out.println("Closed");
  }
}
```

List of sessions

Send message to all connected sessions

Add sessions to the list

Remove sessions from the list

index.html

```
<html>
<head>
   <title>WebSocket Chat Application</title>
   <script type="text/javascript">
 var webSocket;
 var messagesArea = document.getElementById("messages");

 function connect() {
    // Ensures only one connection is open at a time
    if(webSocket !== undefined && webSocket.readyState !== WebSocket.CLOSED){
       writeStatus("Connection is already opened.");
        return;
    }
    webSocket = new WebSocket("ws://localhost:8080/chat");

    webSocket.onopen = function(event){
       writeStatus("Connection open")
    };

    webSocket.onclose = function(event){
      writeStatus("Connection closed");
    };

    webSocket.onmessage = function (event) {
      writeMessage(event.data);
    };
 }
```

index.html

```
function disconnect() {
   webSocket.close();
 }
 function sendMessage() {
    var user = document.getElementById("userName").value.trim();
    var message = document.getElementById("messageInput").value.trim();
    webSocket.send(user+" : " + message+"\r\n")
 }

 function writeMessage(text){
  document.getElementById("messages").value =
document.getElementById("messages").value + text;
 }

 function writeStatus(text){
  document.getElementById("status").innerHTML =  text;
 }


  </script>
```

index.html

```
</head>
<body>
<h1> Chat Room </h1>
<div>
   <button id="connect" type="button" onclick="connect();">Connect</button>
   <button id="disconnect" type="button" onclick="disconnect();">Disconnect</button>
</div>
<textarea id="messages" readonly cols="40" rows="15"> </textarea><br/>
UserName: <input id="userName" type="text"/><br/>
Message<input id="messageInput" type="text" width="300"/><br/>
<input type="button" value="Send" onclick="sendMessage();"/>
<br/>
<div style="color:red" id="status"></div>
</body>
</html>
```