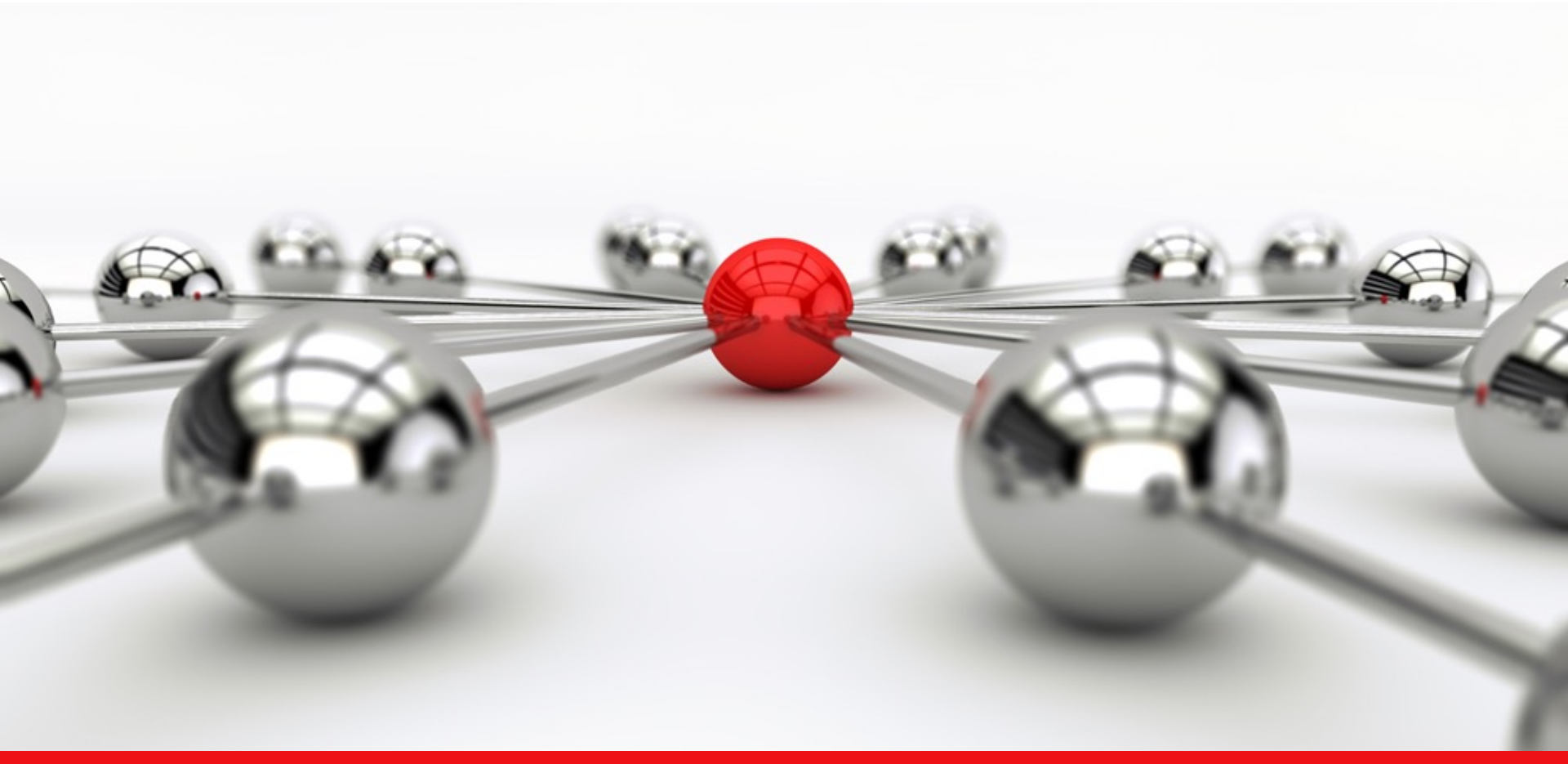
# Verteilte Systeme Master

christian.zirpins@hs-karlsruhe.de
Web Service Lab

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES



# Agenda

## Java Web Services with JAX-WS

- Architectural Overview
- Programming Model

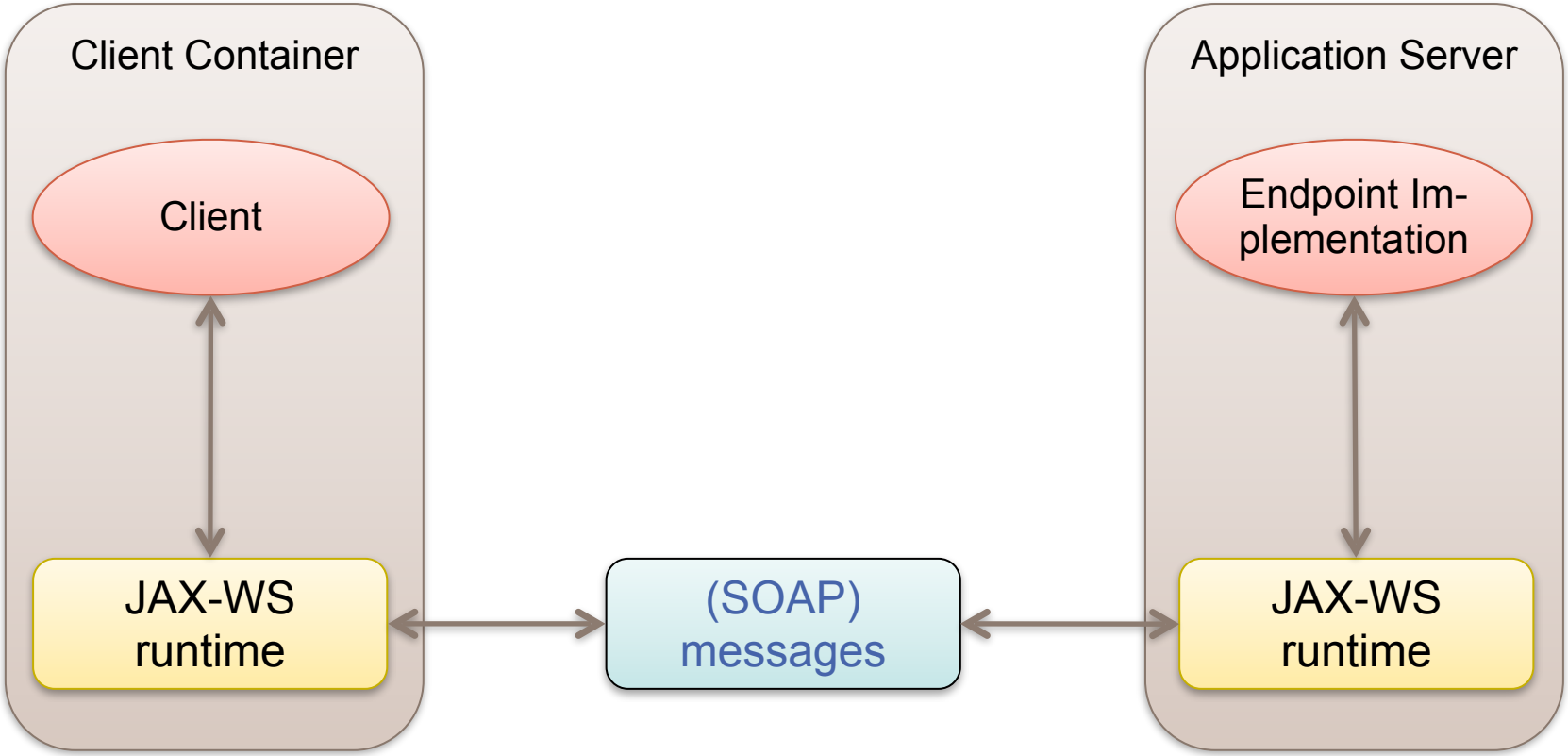
## Web Service Lab

- Code a simple distributed Web service system
- Examine WSDL Web service descriptions
- Links to related/supportive resources

# Java Web Services with JAX-WS

- Sun (Oracle) is developing a java.net project called Metro, providing a complete web services stack
- The Java API for XML Web Services (JAX-WS) is the current model for SOAP-based web services in Metro
  - JAX-WS replaces the older *JAX-RPC* model
  - It uses specific *Java EE 5* features, such as *annotations*, to simplify the task of developing web services
- Because it uses SOAP for messaging, …
  - …JAX-WS is *transport neutral*
  - …JAX-WS supports a wide range of modular WS-* specifications, such as *WS-Security* and *WS-Reliable Messaging*.

# JAX-WS Client/Server-Architecture



# JAX-WS RI Components

- **APIs for implementing Web services**
  - “high level” (POJOs), “low level” (Provider Interface)
  - “plug-ins” (Message-Handler Framework)
  - …
- **Annotations**
  - Mapping from Java to WSDL and XML-Schema
  - Control of Web service calls and processing at runtime
- **Tools**
  - wsimport: generates *portable artifacts* like *Port* (Proxy-Objekt), *SEI* (Service Endpoint Interface ~ Stub), JAXB mapper classes, ...)
  - wsgen: generates XML artifacts like WSDL, XML-Schemata
- **Runtime platform**
  - Marshaling/unmarshaling
  - XML processing
  - Protocol handling (SOAP)

# JAX-WS Programming Model

- **Server-site implementing *(bottom-up)***
  - Implementation of *SEI (Service Endpoint Interface)*
  - Generation of *portable artifacts* (WSDL, XML-Schemata) from SEI → *wsgen*
  - Package into WAR archive, deployment
  - Common: tool-support by IDE for application server
- **Server-site implementing *(top-down)***
  - Generation of *portable artifacts* (SEI, JAXB mapper classes) from WSDL (+ embedded XML-Schema) → *wsimport*
  - Implementation of SEI
  - Package into WAR archive, deployment

# Example: JAX-WS Endpoint

```
package helloservice.endpoint;
import javax.jws.*;

@WebService
public class Hello {
  private String message = new String("Hello,");
  public void Hello() {} //constructor

  @WebMethod
  public String sayHello(String name){
     return message + name + ".";
  }
}
```

# JAX-WS Programming Model (2)

- **JAX-WS endpoint requirements** (some)
  - Implementing class has to be annotated either with `javax.jws.WebService` or `javax.jws.WebServiceProvider`
  - Business methods that are to be provided by the service on the Web have to be annotated with `javax.jws.WebMethod`
  - All input and return parameters need to be JAXB-conformant
- **Client implementation**
  - Generation of *portable artifacts* from WSDL → *wsimport*
  - Access of Web service methods
    - By means of dynamic, generated (from client-side artifacts) proxies *(Ports)*, transparent access like with endpoint implementation
    - By means of *Dispatch API*, access on (SOAP) message-level, allows e.g. for asynchronous calls, REST calls

# Agenda

## Java Web Services with JAX-WS

- Architectural Overview
- Programming Model

## Web Service Lab

- Code a simple distributed Web service system
- Examine WSDL Web service descriptions
- Links to related/supportive resources

# Web Service Lab (1/2)

## 1a: A simple distributed Web service system

- The exercise builds on a Maven project showcasing a simple Web service (lab1-example)
  - Study and run the example (see tutorial link on slide 13)
- Extend the Maven project to implement a new Web Service
  - Create a simple *Fibonacci Web Service* that takes an integer *n* as its parameter and returns the $n^{th}$ number of the Fibonacci row as integer result.
  - Write an interface (`FibonacciService`) and an implementation class (`FibonacciServiceImpl`) in the package `de.hska.iwi.vislab.lab1.example.ws`
  - There is already a test for the service: `FibonacciTest.java`
    - To include this test, copy the source from `lab1-example/` to `lab1-example/src/test/java/de/hska/iwi/vislab/lab1/`
    - Your goal is to run `mvn test` without error.
- Describe the characteristics of this distributed system:
  - Which interaction style (synchrony, invocation semantics) is used in the communication between the test client and your Web service?
  - Describe the activities of the system components during the communication process.

# Web Service Lab (2/2)

## 1b: WSDL Web service descriptions

- Get the WSDL description of the Fibonacci Web Service using a Web browser (run TestWsServer.java, GET the Web service endpoint with query param 'wsdl') and describe its characteristics.
  - What kind of **binding** was used?
  - Which **communication style** was used for the specific binding?
  - What kind of **message encoding** was used?

# Links

## Lab Exercise Download

- https://github.com/zirpins/vislab

## Metro Web Services Home

- https://javaee.github.io/metro/

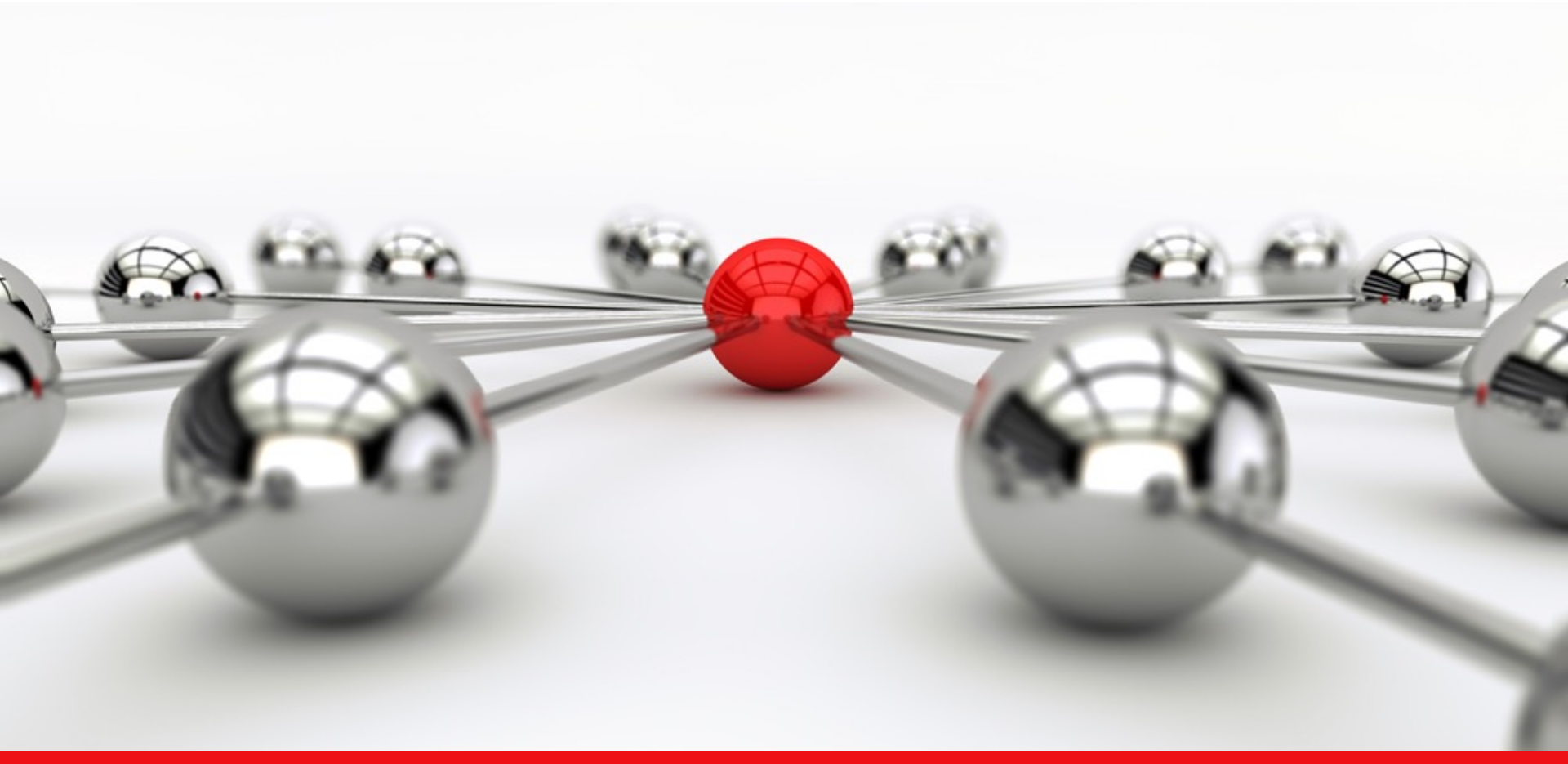
## Maven JAX-WS Tutorial

- http://www.torsten-horn.de/techdocs/jee-jax-ws.htm

# Verteilte Systeme Master

christian.zirpins@hs-karlsruhe.de
Web Service Lab

Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES



# Agenda

## Java Web Services with JAX-WS

- Architectural Overview
- Programming Model

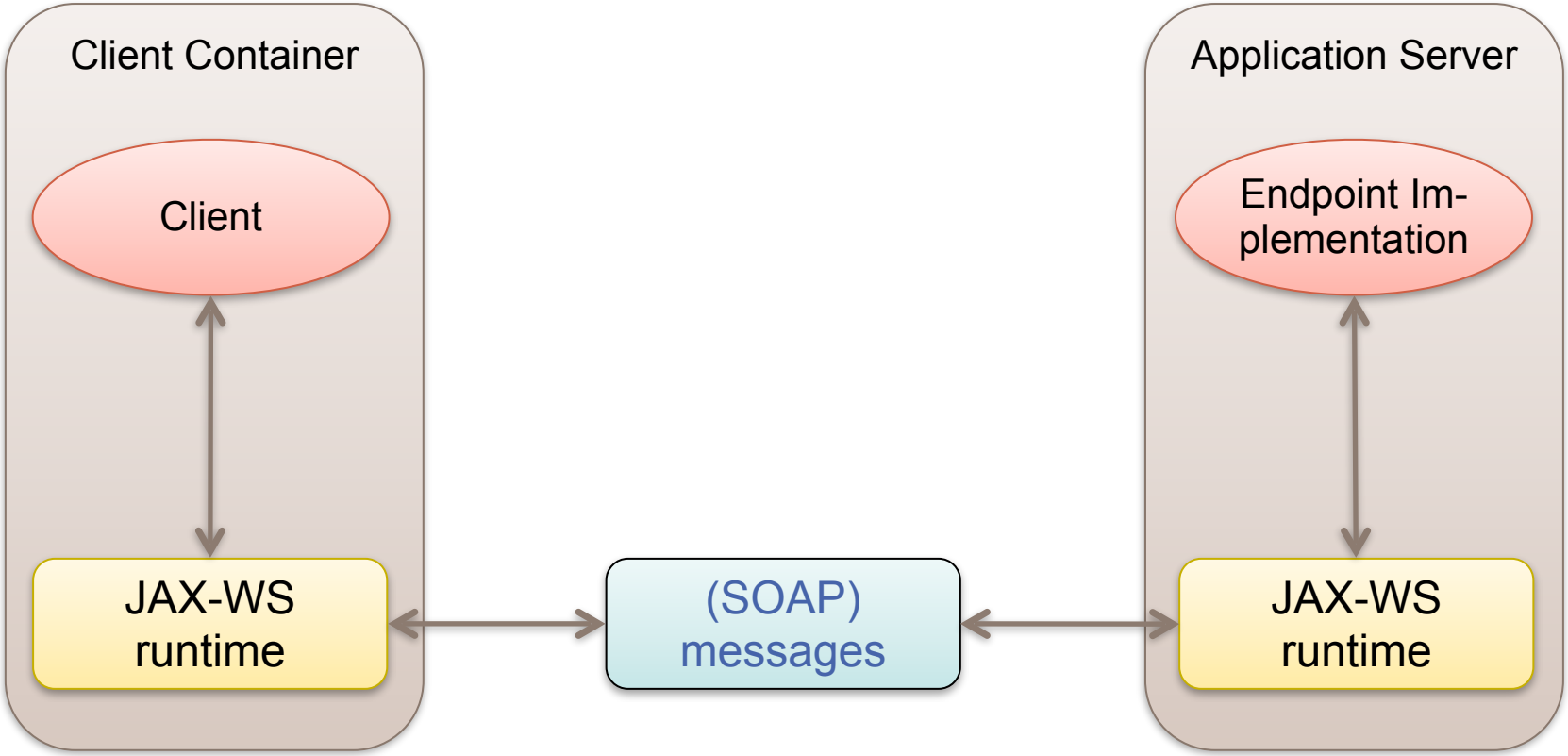
## Web Service Lab

- Code a simple distributed Web service system
- Examine WSDL Web service descriptions
- Links to related/supportive resources

# Java Web Services with JAX-WS

- Sun (Oracle) is developing a java.net project called Metro, providing a complete web services stack
- The Java API for XML Web Services (JAX-WS) is the current model for SOAP-based web services in Metro
  - JAX-WS replaces the older *JAX-RPC* model
  - It uses specific *Java EE 5* features, such as *annotations*, to simplify the task of developing web services
- Because it uses SOAP for messaging, …
  - …JAX-WS is *transport neutral*
  - …JAX-WS supports a wide range of modular WS-* specifications, such as *WS-Security* and *WS-Reliable Messaging*.

# JAX-WS Client/Server-Architecture



# JAX-WS RI Components

- **APIs for implementing Web services**
  - “high level” (POJOs), “low level” (Provider Interface)
  - “plug-ins” (Message-Handler Framework)
  - …
- **Annotations**
  - Mapping from Java to WSDL and XML-Schema
  - Control of Web service calls and processing at runtime
- **Tools**
  - wsimport: generates *portable artifacts* like *Port* (Proxy-Objekt), *SEI* (Service Endpoint Interface ~ Stub), JAXB mapper classes, ...)
  - wsgen: generates XML artifacts like WSDL, XML-Schemata
- **Runtime platform**
  - Marshaling/unmarshaling
  - XML processing
  - Protocol handling (SOAP)

# JAX-WS Programming Model

- **Server-site implementing *(bottom-up)***
  - Implementation of *SEI (Service Endpoint Interface)*
  - Generation of *portable artifacts* (WSDL, XML-Schemata) from SEI → *wsgen*
  - Package into WAR archive, deployment
  - Common: tool-support by IDE for application server
- **Server-site implementing *(top-down)***
  - Generation of *portable artifacts* (SEI, JAXB mapper classes) from WSDL (+ embedded XML-Schema) → *wsimport*
  - Implementation of SEI
  - Package into WAR archive, deployment

# Example: JAX-WS Endpoint

```
package helloservice.endpoint;
import javax.jws.*;

@WebService
public class Hello {
  private String message = new String("Hello,");
  public void Hello() {} //constructor

  @WebMethod
  public String sayHello(String name){
     return message + name + ".";
  }
}
```

# JAX-WS Programming Model (2)

- **JAX-WS endpoint requirements** (some)
  - Implementing class has to be annotated either with `javax.jws.WebService` or `javax.jws.WebServiceProvider`
  - Business methods that are to be provided by the service on the Web have to be annotated with `javax.jws.WebMethod`
  - All input and return parameters need to be JAXB-conformant
- **Client implementation**
  - Generation of *portable artifacts* from WSDL → *wsimport*
  - Access of Web service methods
    - By means of dynamic, generated (from client-side artifacts) proxies *(Ports)*, transparent access like with endpoint implementation
    - By means of *Dispatch API*, access on (SOAP) message-level, allows e.g. for asynchronous calls, REST calls

# Agenda

## Java Web Services with JAX-WS

- Architectural Overview
- Programming Model

## Web Service Lab

- Code a simple distributed Web service system
- Examine WSDL Web service descriptions
- Links to related/supportive resources

# Web Service Lab (1/2)

## 1a: A simple distributed Web service system

- The exercise builds on a Maven project showcasing a simple Web service (lab1-example)
  - Study and run the example (see tutorial link on slide 13)
- Extend the Maven project to implement a new Web Service
  - Create a simple *Fibonacci Web Service* that takes an integer *n* as its parameter and returns the $n^{th}$ number of the Fibonacci row as integer result.
  - Write an interface (`FibonacciService`) and an implementation class (`FibonacciServiceImpl`) in the package `de.hska.iwi.vislab.lab1.example.ws`
  - There is already a test for the service: `FibonacciTest.java`
    - To include this test, copy the source from `lab1-example/` to `lab1-example/src/test/java/de/hska/iwi/vislab/lab1/`
    - Your goal is to run `mvn test` without error.
- Describe the characteristics of this distributed system:
  - Which interaction style (synchrony, invocation semantics) is used in the communication between the test client and your Web service?
  - Describe the activities of the system components during the communication process.

# Web Service Lab (2/2)

## 1b: WSDL Web service descriptions

- Get the WSDL description of the Fibonacci Web Service using a Web browser (run TestWsServer.java, GET the Web service endpoint with query param 'wsdl') and describe its characteristics.
  - What kind of **binding** was used?
  - Which **communication style** was used for the specific binding?
  - What kind of **message encoding** was used?

# Links

## Lab Exercise Download

- https://github.com/zirpins/vislab

## Metro Web Services Home

- https://javaee.github.io/metro/

## Maven JAX-WS Tutorial

- http://www.torsten-horn.de/techdocs/jee-jax-ws.htm