Java Technologies

Web Services

The Context

- We are in the context of developing **distributed business components** that must be shared by various applications.
- EJB components are dedicated to the Java EE environment → inaccessible to a PHP app.
- How to expose a functionality in an **heterogeneous (mixed)** environment?
  - independent of platform, vendor, technology
- How to locate and invoke this functionality?
  - similar to using EE resources via JNDI

What is a Service?

An act of helpful activity

- The supplying of utilities or commodities, as water, electricity, or gas, required or demanded by the public.
- The providing of accommodation and activities required by the public, as maintenance, repair, etc.
- The supplying or a supplier of public communication and transportation: telephone service, bus service.

- **Someone is offering: how can you offer a service?**
- **Someone is using it: how can you use a service?**

Protocols are needed

Service Oriented Architecture

OASIS (the Organization for the Advancement of Structured Information Standards):

“A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.”

Distributed Systems Implementations

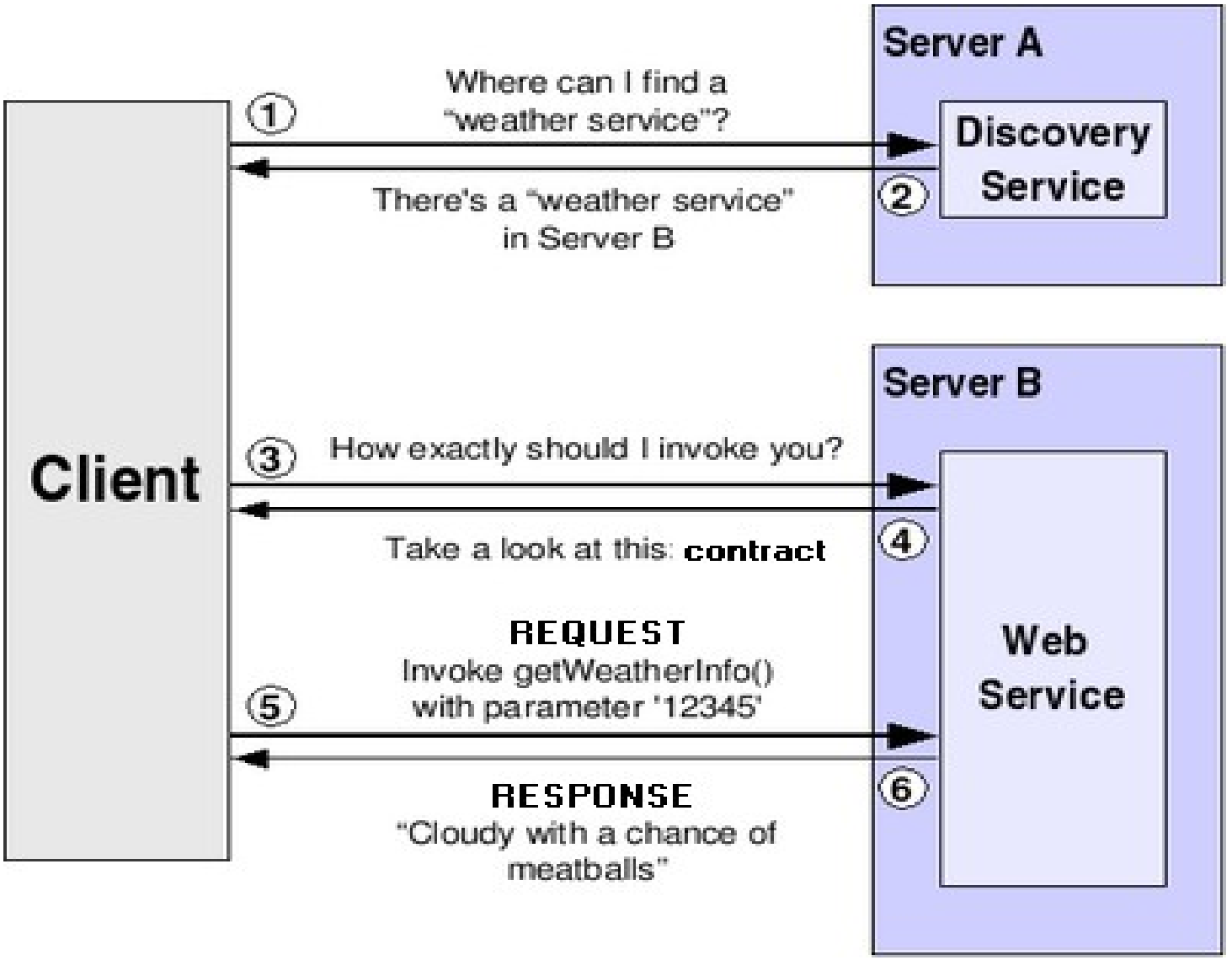
- **Proprietary or technology-specific**
  - **CORBA**: Common Object Request Broker Architecture (Object Management Group)
  - **Java RMI**: Java Remote Method Invocation (Sun)
  - **DCE**: Distributed Computing Environment (Open Group)
  - **DCOM**: Distributed Component Object Model (Microsoft)
  - **WCF**: Windows Communication Foundation (Microsoft)
- **Web Services**

  → **Interoperability over custom integration**

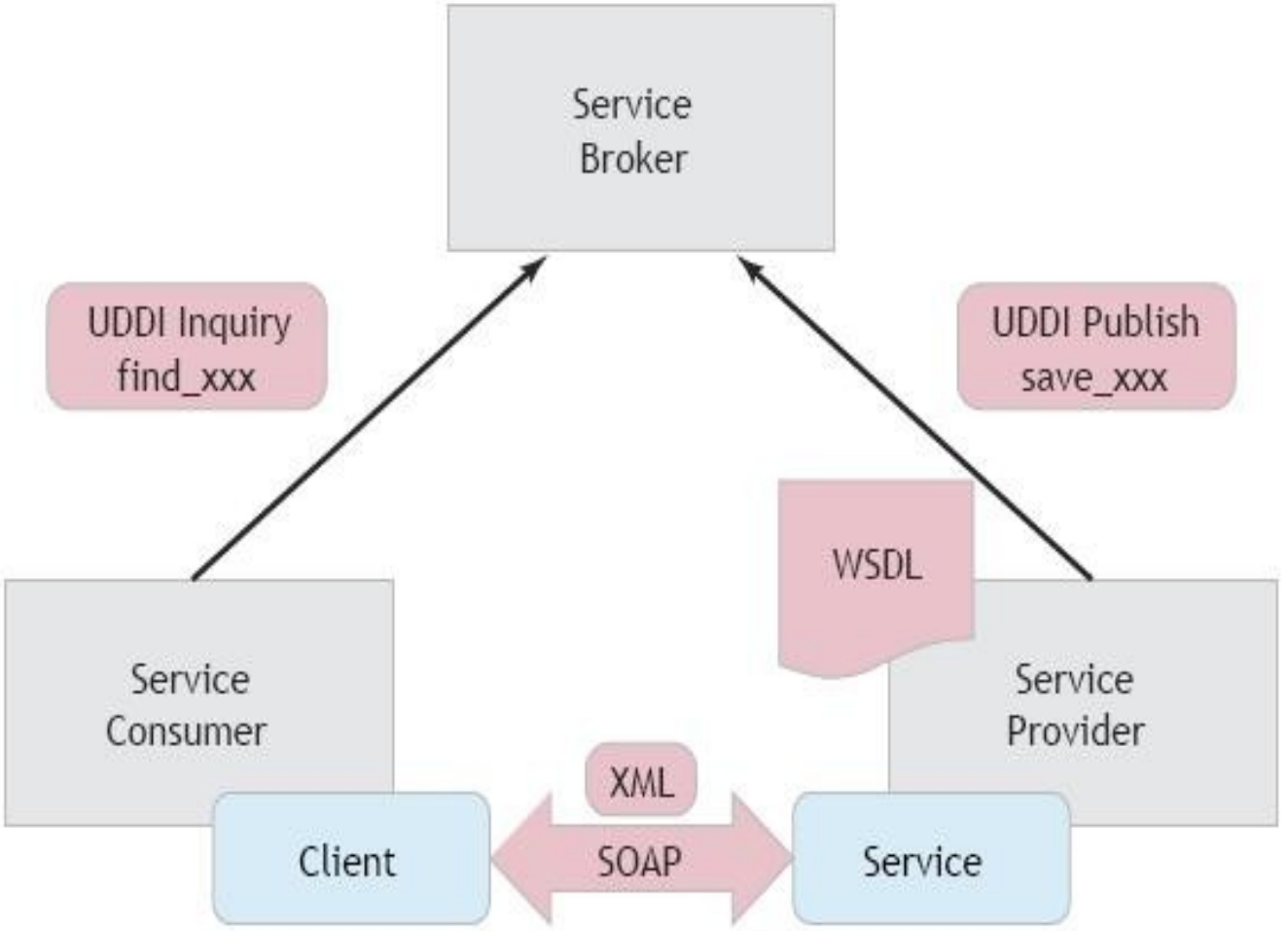
Web Services

- Web services are client and server applications that **communicate over HTTP in a standard manner**.
- Provide a **standard means of interoperating** between software applications running on a variety of platforms and frameworks → **Interoperability**.
  - Application-to-Application (A2A) → EAI
  - Business-to-Business (B2B) → JBI
- **Can be combined in** a loosely coupled way to achieve complex operations → **Extensibility**
  - Web Service [Automated] Composition, Mash-up
  - Composition languages, for example: BPEL

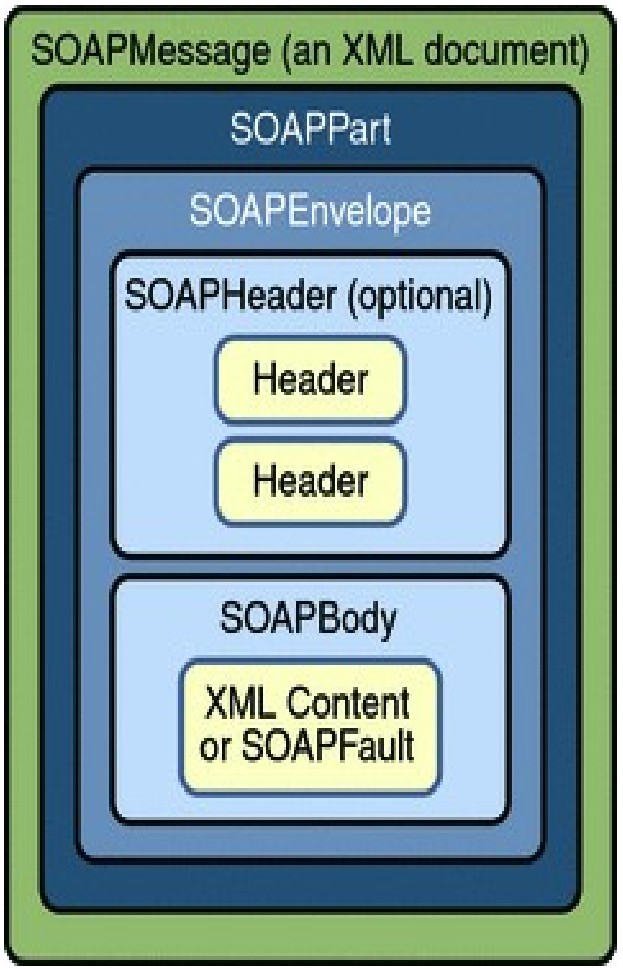
How Should a Web Service Work?



Web Service Protocols



Simple Object Access Protocol (SOAP)



Request Message

```
<soap:Envelope
    xmlns:m="http://demo/ ...">
  <soap:Body>
    <m:sayHelloRequest>
      <name>duke</name>
    </m:sayHelloRequest>
  </soap:Body>
</soap:Envelope>
```

Response Message

```
<soap:Envelope
    xmlns:m="http://demo/ ...">
  <soap:Body>
    <m:sayHelloResponse>
      <return>Hello duke !</return>
    </m:sayHelloResponse>
  </soap:Body>
</soap:Envelope>
```

Web Service Description Language (WSDL)

```
<message name="sayHelloRequest">
  <part name="parameters"
        type="xs:string"/>
</message>

<message name="sayHelloResponse">
  <part name="parameters"
        type="xs:string"/>
</message>

<portType name="Hello">
  <operation name="sayHello">
    <input message="sayHelloRequest"/>
    <output message="sayHelloResponse"/>
  </operation>
</portType>

<service name="HelloWorldService">
  <port name="HelloWorldServicePort">
    <soap:address
      location="http://localhost:8080/WebApp/HelloWorldService"/>
  </port>
</service>
```

A message corresponds to an operation, containing the information needed to perform the operation.

A portType defines a Web service, the operations that can be performed, and the messages that are used to perform the operation.

The WSDL describes services as collections of network *endpoints*, or *ports*.

Universal Description, Discovery and Integration (UDDI)

- **Service Broker:** directory service where businesses can register and search for Web services.
  - White Pages: address, contact, known identifiers;
  - Yellow Pages: industrial categorizations based on standard taxonomies;
  - Green Pages: technical information about services exposed by the business.
- The *provider* publishes the WSDL to UDDI and the *requester* can join to it using SOAP.
- Java API for XML Registries (JAXR)

Types of Web Services

- **"Big" Web Services**
  - Based on XML protocols: SOAP, WSDL, UDDI
  - High level of standardization → QoS
  - **JAX-WS**: The Java API for XML Web Services
- **RESTful Web Services**
  - Representational State Transfer (REST)
  - Simple, suited for basic, ad hoc integration scenarios, using HTTP protocol
  - **JAX-RS**: based on the Jersey Project.

Creating a Web Service with JAX-WS

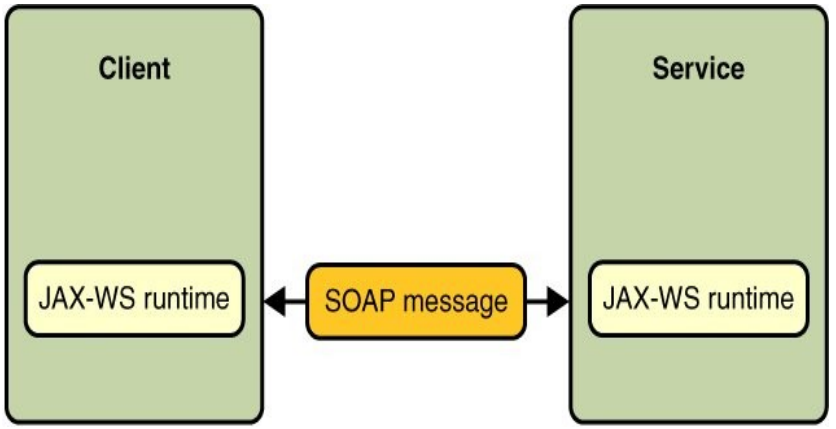
```
import javax.jws.*;

@WebService(serviceName="Greeting")
public class Hello {

  @WebMethod
  public String sayHello(String name) {
    return "Hello " + name + "!";
  }

  @WebMethod(operationName="sayHi")
  public String operation(@WebParam(name = "name") String param) {
    return "Hi " + param + "!";
  }
}
```

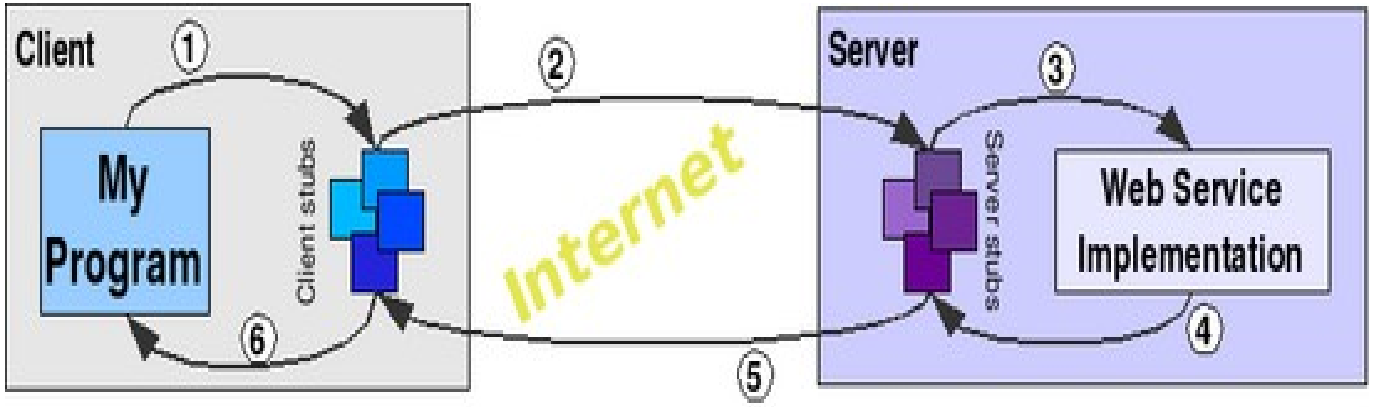
Marks a Java class as implementing a Web Service, or a Java interface as defining a Web Service interface.

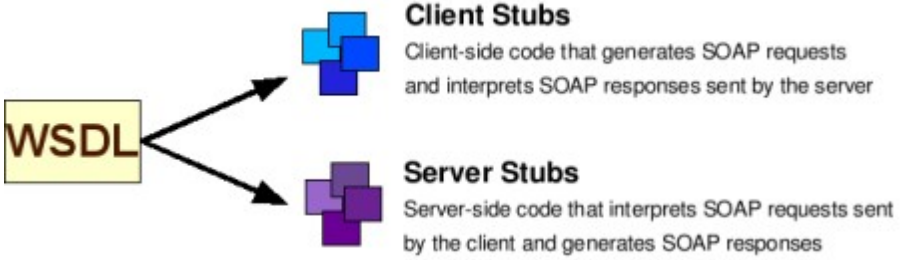
Testing the service in GlassFish

`http://localhost:8080/HelloApp/Greeting?Tester`

Understanding How Things Work







Creating a Client for the WS

- Generating the artefacts (→ Web Service Client)

```
@WebServiceClient(name = "Greeting",
        wsdlLocation = "http://localhost:8080/HelloApp/Greeting?WSDL")
public class Greeting extends Service {
  // JAX-WS generated code
  @WebEndpoint(name = "HelloPort")
  public Hello getHelloPort() {
    return ...;
  }
}
```

- Using the service in an application client

```
public class HelloClient {
  public static void main(String[] args) throws Exception {
    Greeting service = new Greeting();
    Hello hello = helloService.getHelloPort();
    System.out.println(hello.sayHello("duke"));
}
```

WebServices and EJBs

- A WebService might *use* an EJB

```
@WebService(serviceName = "MyWebService")
public class MyWebService {

  @EJB
  private MySessionBean ejbRef;

  @WebMethod(operationName = "businessMethod")
  public void businessMethod() {
      ejbRef.businessMethod();
  }
}
```

- A WebService might *be* a stateless bean

```
@WebService(serviceName = "MyWebService")
@Stateless
public class MyWebService { … }
```

Types Supported by JAX-WS

- JAX-WS delegates the mapping of Java programming language types to and from XML definitions to **JAXB** (Java API for XML Binding)
- Not every data type in the Java language can be used as a method parameter or return type in JAX-WS.
- Examples of *XSD schema-to-Java* bindings:
  - xsd:string ↔ java.lang.String
  - xsd:integer ↔ java.math.BigInteger
  - xsd:base64Binary ↔ byte[], etc.
  - xs:anyType ↔ java.lang.Object, etc.

Message Handlers

- A message handler provides a mechanism for **intercepting the SOAP message** in both the request and response of the Web Service.
  - pre-processing or post-processing of the message,
  - improve the performance using a cache, etc
- JAX-WS supports two types of handlers:
  - **SOAP handlers**: can access the entire SOAP message, including the message headers and body.
  - **Logical handlers**: can access the payload of the message only, and cannot change any protocol-specific information (like headers) in a message.

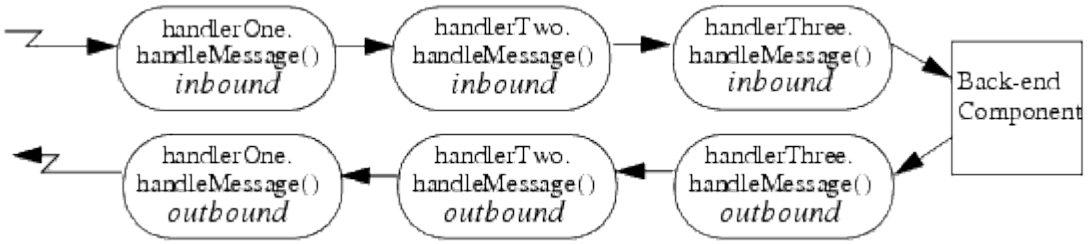
Creating a SOAP Handler

```
public class Handler1 implements SOAPHandler<SOAPMessageContext> {

  public boolean handleMessage(SOAPMessageContext context){
    Boolean outboundProperty = (Boolean) context.get(
       MessageContext.MESSAGE_OUTBOUND_PROPERTY);
    if (outboundProperty.booleanValue()) {
        System.out.println("\nOutbound message:");
    } else {
        System.out.println("\nInbound message:");
    }
    SOAPMessage message = context.getMessage();
    System.out.println(message);
    return true;
  }
  public Set<QName> getHeaders() {
    return Collections.emptySet();
  }
  public boolean handleFault(SOAPMessageContext messageContext){
    return true;
  }
  public void close(MessageContext messageContext){ }
}
```

Creating a Logical Handler

```
public class Handler2 implements LogicalHandler<LogicalMessageContext> {

  public boolean handleMessage(LogicalMessageContext context) {
    Boolean outboundProperty = (Boolean)
        context.get (MessageContext.MESSAGE_OUTBOUND_PROPERTY);
    if (outboundProperty.booleanValue()) {
           System.out.println("\nOutbound message:");
    } else {
           System.out.println("\nInbound message:");
    }
    LogicalMessage message = context.getMessage();
    Source payload = message.getPayload();
    System.out.println(payload);
    return true;
  }

  public boolean handleFault(LogicalMessageContext messageContext) {
    return true;
  }
  public void close(MessageContext messageContext)
  {
  }
}
```

Specifying a Handler Chain



Defining a Handler Chain: MyMessageHandler.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<handler-chains>
  <handler-chain>
    <handler>
      <handler-class>Handler1</handler-class>
    </handler>
    <handler>
      <handler-class>Handler2</handler-class>
    </handler>
    <handler>
      <handler-class>Handler3</handler-class>
    </handler>
  </handler-chain>
</handler-chains>
```

```
@WebService
@HandlerChain(file = "MyMessageHandler.xml")
@WebService(serviceName="Greeting")
public class Hello { … }
```

Message Transmission Optimization Mechanism (MTOM)

- W3C Recommendation designed for optimizing the electronic transmission of attachments.
- MTOM provides an elegant mechanism of efficiently transmitting binary data, such as images, PDF files, MS Word documents, between systems.
- An MTOM-aware web services engine detects the presence of Base64Binary encoded data types and makes a decision – typically based on data size – to convert the Base64Binary data to MIME data with an XML-binary Optimization Package (xop) content type.

RESTful Web Services

- Service implementation over HTTP that conforms to the REST *architectural style*.
- Data and functionality are considered **resources** and are accessed using an **uniform interface**:
  - **Identification** → **URI**, for example: `http://example.com/resources`
  - **Representations** → **Standard MIME types**: `Text, JSON, XML, YAML,` etc.
  - **Operations** → **Standard HTTP methods**: `HTTP POST, GET, PUT, DELETE`

JAX-RS Specification

- A RESTful API can be implemented in various ways: Spring, JAX-RS, or … write your own web components (servlets) to handle requests.
- **JAX-RS is a specification** that provides support in creating web services according to the REST architectural pattern. It uses annotations, to simplify the development and deployment of web service clients and endpoints.
  - Implementations:
    - **Jersey** (GlassFish, Payara),
    - **RESTEasy** (WildFly)
    - **Apache CXF** (TomEE)
    - Restlet, etc.

Creating a RESTful Web Service with JAX-RS

```
import javax.ws.rs.*;

@Path("/helloworld")        // --> Resource Identifier
public class HelloWorldResource {

  @GET                      // --> Process HTTP GET requests
  @Produces("text/plain") // --> MIME Media type
  public String getMessage() {
    return "Hello World";
  }
  @PUT
  @Consumes("text/plain")
  public void setMessage(String msg) {
  ...
  }
}
```

Testing the service:
`http://localhost:8080/HelloApp/`**`resources`**`/helloworld` → Hello World

Configuring JAX-RS

Identifies the application path that serves as the base URI for all resource URIs provided by Path.

```
@ApplicationPath("resources")
@ApplicationScoped
public class ApplicationConfig extends Application {
    /**
     * Do not modify addRestResourceClasses() method.
     * It is automatically populated with
     * all resources defined in the project.
     * If required, comment out calling this method in getClasses().
     */
    private void addRestResourceClasses(Set<Class<?>> resources) {
        resources.add(HelloWorldResource.class);
    }

    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        addRestResourceClasses(resources);
        return resources;
    }
}
```

Note: Earlier implementations needed the mapping of the Jersey servlet in web.xml

`com.sun.jersey.spi.container.servlet.ServletContainer`

@Path

The @Path annotation's value is a relative URI path indicating where the Java class will be hosted. You can also embed variables in the URIs to make a URI path template.

```
@Path("/hello/{name}") //--> URI Path Template
@RequestScoped
public class HelloResource {

  @GET @Produces("text/plain")
  public String getMessage(@PathParam("name") String name) {
    return "Hello " + name;
  }

  @GET @Path("{message}")
  public String get(@PathParam("name") String name,
                    @PathParam("message") String message) {
    return message + " " + name;
  }
}
```

http://localhost:8080/HelloApp/resources/**hello/world**
  --> Hello world
http://localhost:8080/HelloApp/resources/**hello/duke/Salut**
  --> Salut duke

@Produces

Specifies the MIME media types or representations a resource can produce and send back to the client.

```
@Path("/myResource")
@Produces("text/plain")  // --> applied at class level
public class SomeResource {
    @GET
    public String doGetAsPlainText() {
        ...
    }

    @GET
    @Produces("text/html") // --> applied at method level
    public String doGetAsHtml() {
        ...
    }

    @GET
    @Produces({"image/jpeg, image/png"})
    public byte[] doGetAsImage() { ... }
}
```

If a resource class is capable of producing more than one MIME media type, the resource method chosen will correspond to the most acceptable media type as declared by the client.

The ResponseBuilder

```
import javax.ws.rs.core.Response;

@Path("/example")
public RootResource {
  @GET
  @Produces("text/xml")
  public Response getSomeXML() {
    return Response.ok("some xml").type("text/xml").build();
  }

  @POST
  @Produces(MediaType.APPLICATION_JSON)
  public Response post(...) {
    ....
    return Response.created(uriBuilder.build(index)).
       tag(etag).
       cookie(cookie).
       entity(widget).
       type(MediaType.APPLICATION_JSON).
       build();
  }
```

Build Response instances that contain metadata instead of or in addition to an entity

Methods decorated with request method designators must return void, a Java programming language type, or a javax.ws.rs.core.Response object.

HTTP Methods

- GET: Used to request data from a specified resource. Should not produce any side effect.
- POST: Used to **create** a new child resource, at a *server* defined URL. POST is **non-idempotent** which means multiple requests will have different effects
- PUT: Used to **create or replace if exists** a resource, at a URL known by the *client*. **Idempotent**, which means multiple requests will have the same effect.
- PATCH: Used to **update** part of the resource at the *client* defined URL.
- DELETE: Used to **remove** a specified resource (from the database, for example)

Read (GET)

```
@Path("/products")
@ApplicationScoped
public class ProductController {
    private final List<Product> products = new ArrayList<>();
    public ProductController() {
        products.add(new Product(1, "Mask"));
        products.add(new Product(2, "Gloves"));
    }
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Product> getProducts() {
        return products;
    }

    @GET
    @Path("/count")
    @Produces(MediaType.TEXT_PLAIN)
    public int countProducts() {
        return products.size();
    }

    @GET
    @Path("/{id}")
    public Product getProduct(@PathParam("id") int id) {
        return products.stream()
           .filter(p -> p.getId() == id).findFirst().orElse(null);
    }
}
```

localhost:8080/resources/products

```
[{"id":1,"name":"Mask"},
{"id":2,"name":"Gloves"}]
```

localhost:8080/resources/products/count

```
2
```

localhost:8080/resources/products/1

```
{"id":1,"name":"Mask"}
```

Create (POST) using FORMs

```
@Path("/products")
@ApplicationScoped
public class ProductController {
    ...
    private int add(Product product) {
        product.setId(1 + products.size());
        products.add(product);
        return product.getId();
    }

    @POST
    public int createProduct(@QueryParam("name") String name) {
        return add(new Product(name));
    }

}
```

```
<html> <body>
  <form action="http://localhost:8080/resources/products" method="POST">
    Product: <input type="text" name="name"/> <br/>
    <input type="submit" value="Submit"/>
  </form>
</bodY> </html>
```

Create (POST) using Raw Data

```
@Path("/products") @ApplicationScoped
public class ProductController {
    ...
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Product createProduct(Product product) {
        add(product);
        return product;
    }
    @POST
    public Response createProductAndResponse(Product product) {
        int productId = add(product);
        URI uri = UriBuilder.fromResource(this.getClass())
                            .path("" + productId).build();
        return Response.created(uri).entity(product).build();
    }
}
```

Postman:

POST
http://localhost:8080/resources/products
Header:
key="Content-Type" value="application/json"
Body: raw:
{ "name": "Vaccine" }

Update (PUT)

```
@Path("/products") @ApplicationScoped
public class ProductController {
    ...
    @PUT
    @Path("/{id}")
    public Response changeName(@PathParam("id") int id,
                               @QueryParam("name") String newName) {

        Product product = findById(id);
        if (product != null) {
            product.setName(newName);
            return Response.ok().build();
        }
        return Response.noContent().build();
    }
}
```

Postman:

PUT
http://localhost:8080/resources/products/1
Query params:
key="name" value="The Mask of Zorro"

DELETE

```
@Path("/products") @ApplicationScoped
public class ProductController {
    ...
    @DELETE
    @Path("/{id}")
    public Response deleteProduct(@PathParam("id") int id) {
        Product product = findById(id);
        if (product == null) {
            return Response.noContent().build();
        }
        products.remove(product);
        return Response.ok().entity("Product deleted!").build();
    }
}
```

Postman:

DELETE

http://localhost:8080/resources/products/1

Naming Conventions

- The URL is a sentence, where resources are nouns and HTTP methods are verbs.
- The resource should always be plural.
- Specify an *id* to access one instance of the resource.
  - `GET /products`
  - `GET /products/123`
  - `POST /products`
  - `PUT /products/123`
  - `DELETE /products/123`
- Searching, sorting, filtering and pagination
  - `GET /products`**`?sort=name_asc`**
  - `GET /products?category=food&country=Romania`
  - `GET /products?search=Pizza`
  - `GET /products?page=2`
- Versioning `/products/v1`

HTTP Response Status Codes

- 2xx (Success category)
  - 200 **Ok**, 201 **Created**, 202 **Accepted**, 204 **No Content**
- 3xx (Redirection Category)
  - 301 **Moved Permanently**, 304 **Not Modified**
- 4xx (Client Error Category)
  - 400 **Bad Request**, 401 **Unauthorized**,
  - 403 **Forbidden**, 404 **Not Found**, 410 **Gone**
- 5xx (Server Error Category)
  - 500 **Internal Server Error**, 503 **Service Unavailable**

Calling a Service: ClientBuilder

Distributed object communication

Programatic

```
String uriOne = "http://localhost:8080/resources/products/1";
Product product =
   ClientBuilder.newClient()
    .target(uriOne)
    .request(MediaType.APPLICATION_JSON)
    .get(Product.class);
```

```
String uriAll = "http://localhost:8080/resources/products";
List<Product> list =
   ClientBuilder.newClient()
    .target(uriAll)
    .request(MediaType.APPLICATION_JSON)
    .get(new GenericType<ArrayList<Product>>() {});
```

Asynchronous Invocation

in the Client API

- Network issues can affect the perceived performance of the application, particularly in long-running or complicated network calls. Asynchronous processing helps prevent blocking and makes better use of an application's resources.
- In the JAX-RS Client API, the Invocation.Builder.async method is used when constructing a client request to indicate that the call to the service should be performed asynchronously.
- An asynchronous invocation returns control to the caller immediately, with a return type of java.util.concurrent.Future<T> and with the type set to the return type of the service call.

Polling

- *Future<T>* objects have methods to check if the asynchronous call has been completed, to retrieve the final result, to cancel the invocation, and to check if the invocation has been cancelled.

```
Client client = ClientBuilder.newClient();
WebTarget myResource =
    client.target("http://localhost:8080/resources/products/1");

Future<Product> response
    = myResource.request(MediaType.APPLICATION_JSON)
      .async()
      .get(Product.class);
...
response.isDone()
        .isCancelled()
        .get()
```

Custom Callbacks

- The *InvocationCallback* interface defines two methods, *completed* and *failed*, that are called when an asynchronous invocation either completes successfully or fails, respectively.

```
Client client = ClientBuilder.newClient();
WebTarget myResource =
    client.target("http://localhost:8080/resources/products/1");
Future<Product> response
    = myResource.request(MediaType.APPLICATION_JSON)
      .async()
      .get(new InvocationCallback<Product>() {
         @Override
         public void completed(Product product) {
            // Do something with the product object
         }
         @Override
          public void failed(Throwable throwable) {
            // Handle the error
         }
});
```

Other Features of the Client API

https://docs.oracle.com/javaee/7/tutorial/jaxrs-client003.htm

- Setting Message Headers in the Client Request

```
WebTarget.header(String name, Object value)
WebTarget.headers(MultivaluedMap<String, Object> headers)
```

The *MultivaluedMap* interface allows you to specify multiple values for a given key.

- Setting Cookies in the Client Request

```
WebTarget.cookie(String name, String value)
WebTarget.cookie(Cookie cookie)
```

- Adding Filters to the Client

```
Client.register(MyLoggingFilter.class);
```

Invocation Filters

- You can register custom filters with the client request or the response received from the target resource

```
Client client = ClientBuilder.newClient()
    .register(MyLoggingFilter.class);
```

- Request and response filter classes implement the *ClientRequestFilter* / *ClientResponseFilter* interfaces.

```
@Provider
public class MyLoggingFilter implements ClientRequestFilter  {
    static final Logger logger = Logger.getLogger(...);
    @Override
    public void filter(ClientRequestContext requestContext)
            throws IOException {
        logger.log(...);
        ...
    }
}
```

@Provider marks an implementation of an extension interface that should be discoverable by JAX-RS runtime during a provider scanning phase.

Calling a Service: CDI

Declarative

```
@RegisterRestClient(baseUri="http://localhost:8081/resources")
@ApplicationScoped
public interface Service {
    @GET
    @Path("/products")
    List<Product> getProducts();
}
```

May be defined in microprofile-config.properties

```
@Path("/client")
@ApplicationScoped
public class ClientController {

    @Inject
    @RestClient
    private Service service;

    @GET
    @Path("/test")
    public List<Product> onClientSide() {
        return service.getProducts();
    }
}
```

Calling a Service: jQuery

jQuery is a lightweight, "write less, do more", JavaScript library.

```
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"/>
<script>
$(document).ready(function() {
    $.ajax({
        url: "http://localhost:8080/resources/products/1"
    }).then(function(product) {
       $('.product-id').append(product.id);
       $('.product-name').append(product.name);
    });
});
</script>
</head>
<body>
  <div>
    <p class="product-id">The ID is </p>
    <p class="product-name">The name is </p>
  </div>
</bodY>
</html>
```

AJAX = **A**synchronous **J**avaScript **A**nd **X**ML

Asynchronous Server

in the Container API

https://dzone.com/articles/jax-rs-20-asynchronous-server-and-client

- In synchronous request/response processing model, client connection is accepted and processed in a single I/O thread by the server (taken from the thread pool).
- The thread blocks until the processing is finished and returned. When the processing is done and response is sent back to the client, the thread can be released and sent back to the pool.
- When there is a huge number of requests and processing is heavy and time consuming → all the threads are busy processing → thread pool becomes empty.
- Asynchronous processing model separates the two tasks:
  - connection accepting → one thread: **acceptor**
  - request processing → another thread: **worker**.
- Improves: THROUGHPUT and SCALABILITY

Documenting – OpenAPI

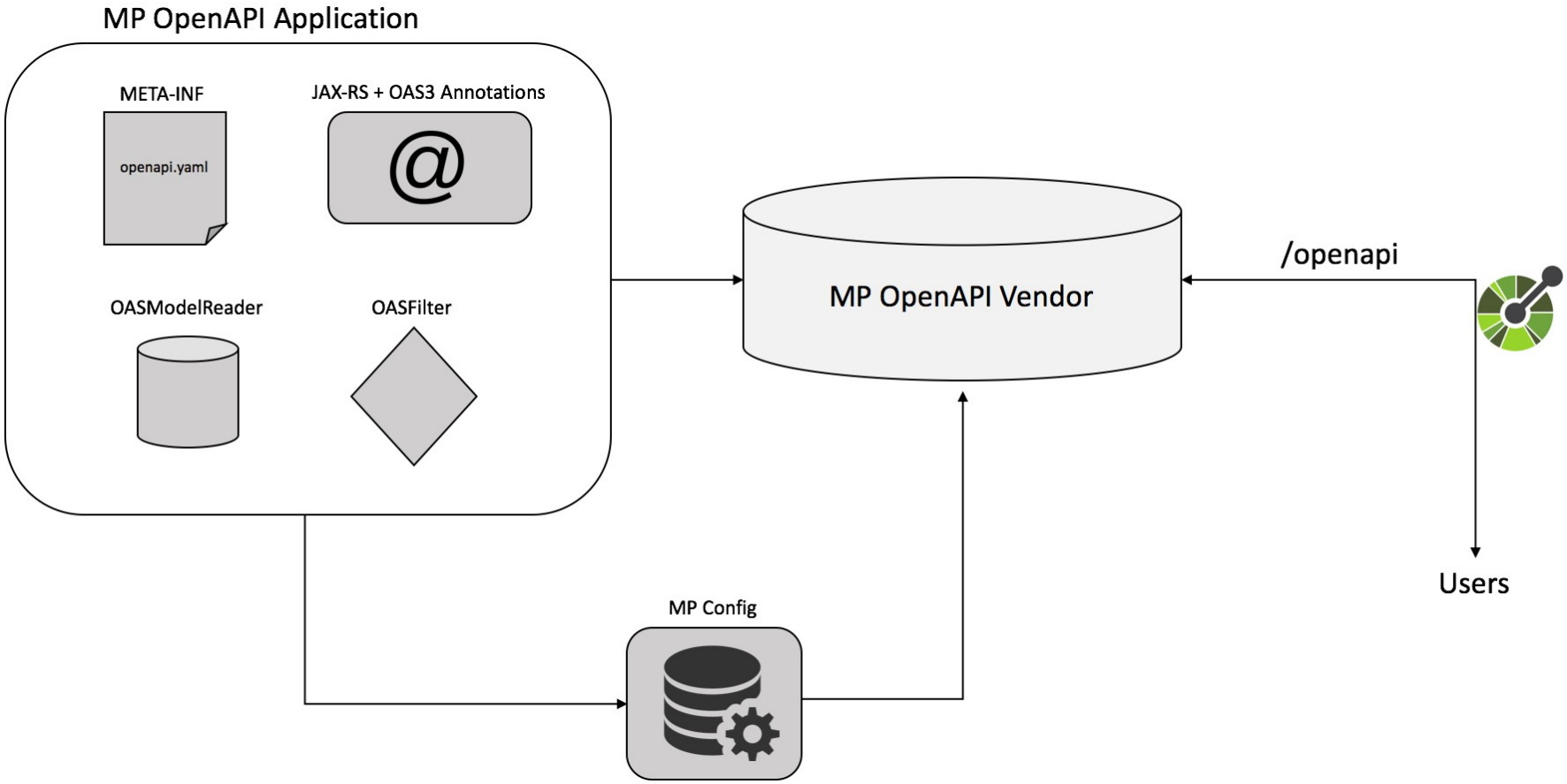
https://swagger.io/

- **OpenAPI**: description format for REST APIs.
  - Available endpoints (/users) and operations on each endpoint (GET /users, POST /users)
  - Operation parameters input and output for each operation
  - Authentication methods
  - Contact information, license, terms of use and other information.
- **Swagger is** a set of open-source tools built around the OpenAPI Specification
  - Browser-based **editor** where you can write OpenAPI specs.
  - **UI** – renders OpenAPI specs as interactive API doc.
  - **Codegen** – generates server stubs and client libraries

MicroProfile OpenAPI

https://download.eclipse.org/microprofile/microprofile-open-api-1.1.2/microprofile-openapi-spec.html

Aims to provide a set of interfaces and programming models which allow developers to natively produce OpenAPI documents from their JAX-RS applications



Example OpenAPI

https://openliberty.io/guides/microprofile-openapi.html

```
@Path("/hello")
@ApplicationScoped
public class HelloController {
  @GET
  @Produces(MediaType.TEXT_PLAIN)
  @Operation(
      summary = "Hello World",
      description = "Returns the <b>Hello World</b> message ")
  public String sayHello() {
      return "Hello World!";
  }
}
```

http://localhost:8080/openapi →

```
openapi: 3.0.3
info:
  title: Generated API
  version: "1.0"
servers:
- url: http://localhost:9080
- url: https://localhost:9443
paths:
  /data/hello:
    get:
      summary: Hello World
      description: 'Returns the ...'
      responses:
        "200":
          description: OK
          content:
            text/plain:
              schema:
                type: string
```

Web Service Composition

- **Programming in the small**: definition of atomic services (accessing resurces, etc.)
  - Define atomic Web Services
  - Direct collaboration → *Choreography*
- **Programming in the large**: composing the existing services using some sort of simple algorithm (usually, an economic process)
  - Create new Web Services
  - Coordinating the services → *Orchestration*
  - Example: choosing an insurance, planning a trip

Methods of WS Composition

- **User defined**: for example, Business Process Execution Language for Web Service (BPEL4WS) and DAML-S ServiceModel, are focused on representing service compositions where flow of a process and bindings between services are known a priori.
- **Automated**: "the ability to efficiently and effectively select and integrate inter organizational and heterogeneous services on the Web **at runtime** is an important step towards the development of the Web service applications. "*"A Survey of Automated Web Service Composition Methods", Jinghai Rao and Xiaomeng Su*