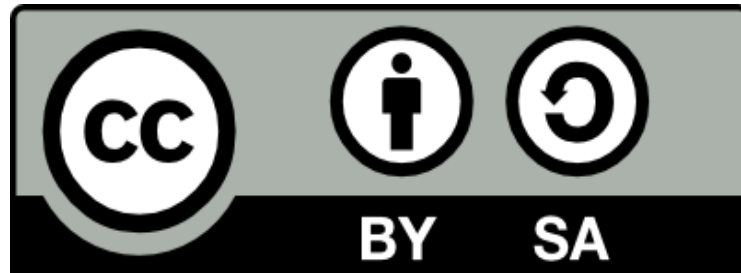


# Java Web Services



# License

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit:
- <http://creativecommons.org/licenses/by-sa/4.0/>



# About me

- Alejo Pacín Jul
- Software Analyst at Vippter
- Freelance IT trainer
- GPUL member
- <https://es.linkedin.com/in/alejopj>



# Java Web Services

Basic info

# Description

- Java Web Services development using SOAP and REST

# Profile & Attendance

- Developers with basic knowledge of Java programming
- Up to 15 attendees

# Schedule

- 2016, October 3<sup>rd</sup> to 7<sup>th</sup> (Mon-Fri)
- 09:00 to 13:00
- Total: 20h

# About you

- Name?
- Position?
- Java/WS skills?
- Why this course?
- Applications?



# Index

- Web services
- Protocol Stack
- Types
- JAX-WS
- RESTful

# Index

- Web services
  - What is a WS?
  - How do they work?
  - Architectures
  - Patterns & anti-patterns

# Index

- Protocol Stack
  - XML
  - SOAP
  - WSDL
  - UDDI
  - WS-Security

# Index

- Types
  - SOAP
  - REST

# Index

- JAX-WS
  - Wsimport
  - Wsgen

# Index

- RESTful
  - Jax-RS
  - Annotations
  - JAXB
  - JSON

<theory>

# Web Services



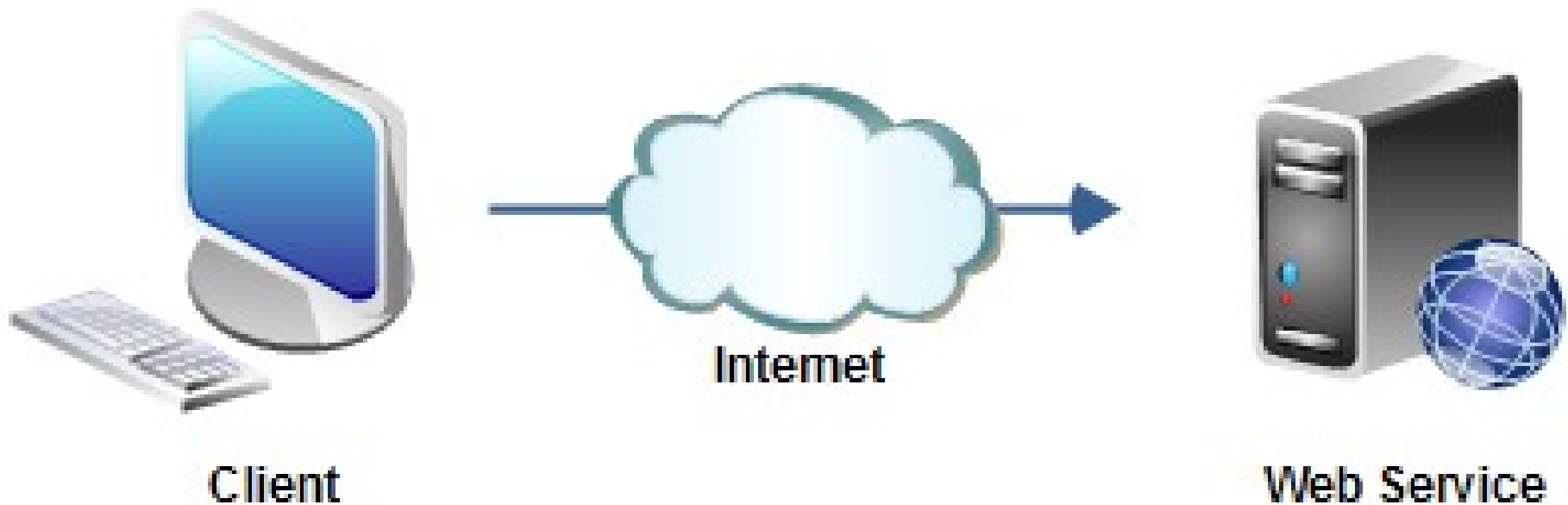
# Web services

- What is a WS?
- How do they work?
- Architectures
- Patterns & anti-patterns

# What is a WS?

- Web services are services that can be accessed over a network, for instance via the global internet.
- Often these web services and their clients communicate via web protocols like HTTP.

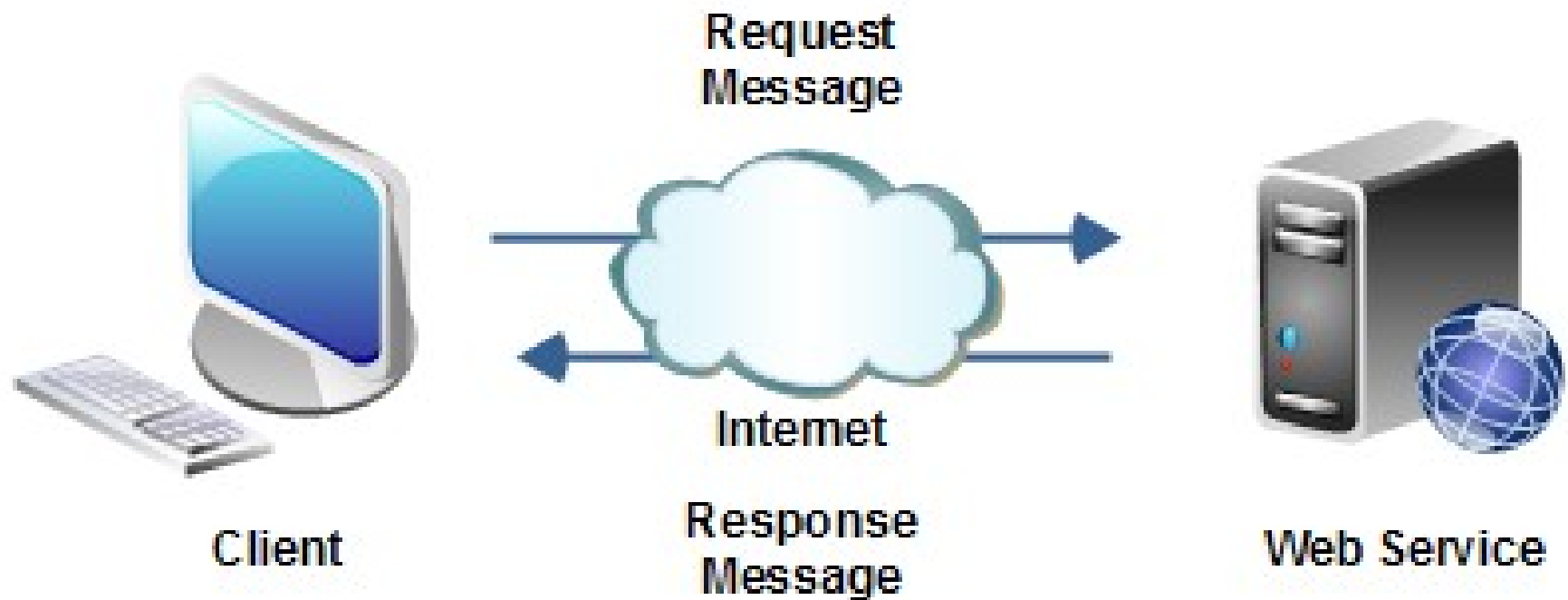
# What is a WS?



# How do they work?

- The term "web service" is often used to describe a service that a client (a computer) can call remotely over the internet, via web protocols like HTTP.
- Like calling a method, procedure or function which is running on a different machine than the client.

# How do they work?



# How do they work?

- If a web service is to be "callable" for clients from the outside world, the client need a description of the **service interface**.
- Without a description of the interface, how would the client know what data to send to the service?

# How do they work?

- You can think of a service interface like an interface in Java or C#.
- The only extra information needed is where the service is located (IP address), and the message format used by the service.

# How do they work?

- Here is what a **service description** should contain:
  - Interface Name
  - Operation Name(s) (if the service has more than one operation)
  - Operation Input Parameters
  - Operation Return Values
  - Service Message Format
  - Service Location (IP Address / URL)



# How do they work?

- How a web service interface description looks depends on the **message format** used by the web service.
- Currently, only SOAP web services has a **standardized interface description** - the Web Service Description Language (WSDL).

# Architectures

- Software architectures
  - Software design architectures
- Hardware architectures

# Architectures

- Software architectures
  - An **architectural pattern** is a general, reusable solution to a commonly occurring problem in software architecture within a given context.
  - Architectural patterns are **similar to software design pattern** but have a broader scope.
  - The architectural patterns **address various issues in software engineering**, such as computer hardware performance limitations, high availability and minimization of a business risk.
  - Some architectural patterns have been implemented within software frameworks.

# Architectures

- Service-oriented architecture (SOA)
  - A **service-oriented architecture (SOA)** is an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network.
  - The principles of service-orientation are independent of any vendor, product or technology.

# Architectures

- Service-oriented architecture (SOA)
  - Horizontal layers:
    - Consumer Interface Layer
    - Business Process Layer
    - Services
    - Service Components
    - Operational Systems

# Architectures

- Service-oriented architecture (SOA)
  - SOA depends on data and services that are described by **metadata** that should meet the following two criteria:
    - The metadata should be provided in a form that software systems can use to configure dynamically by discovery and incorporation of defined services, and also to maintain coherence and integrity.
    - The metadata should be provided in a form that system designers can understand and manage with a reasonable expenditure of cost and effort.

# Architectures

- Service-oriented architecture (SOA)

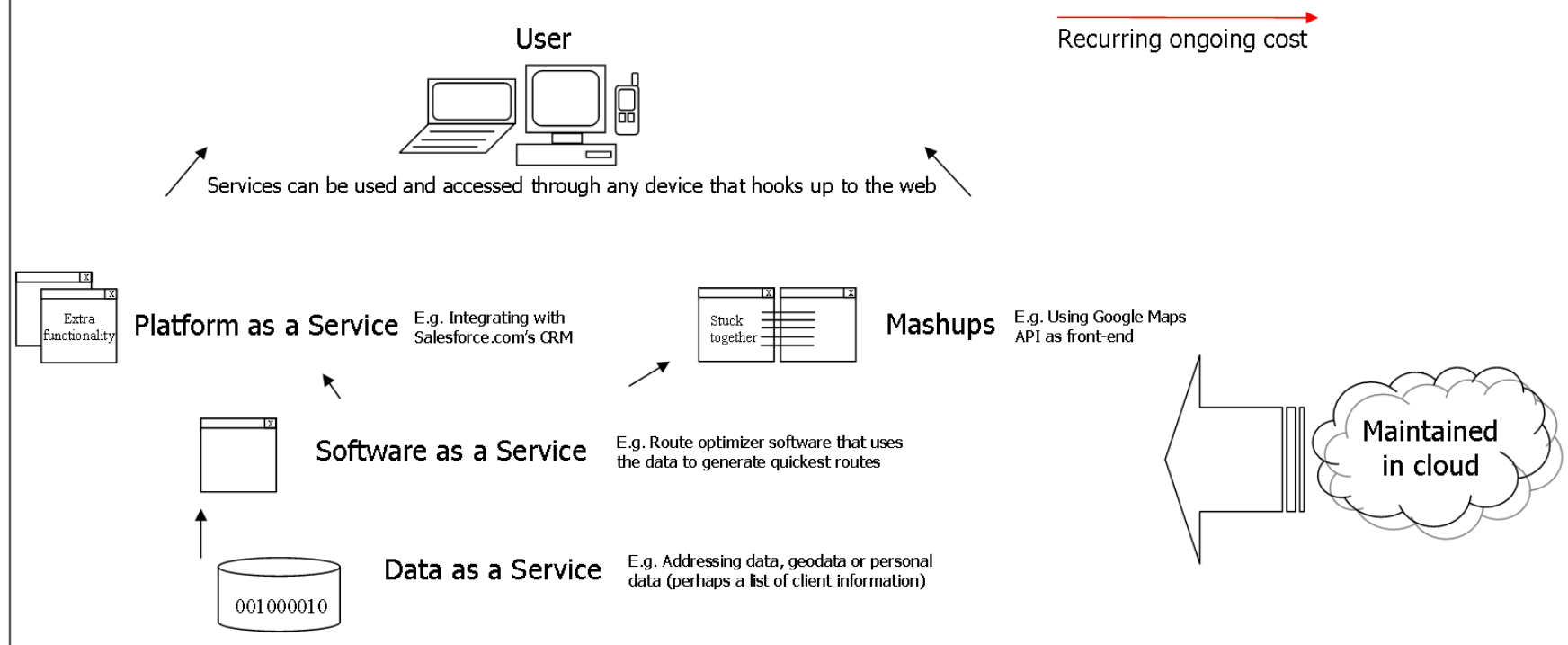
- Principles:

- Standardized service contract
    - Service loose coupling
    - Service abstraction
    - Service reusability
    - Service autonomy
    - Service statelessness
    - Service discoverability
    - Service composability
    - Service granularity
    - Service normalization
    - Service optimization
    - Service relevance
    - Service encapsulation
    - Service location transparency

# Architectures

## Service-Oriented Architecture

A completely service-oriented model





# Architectures

- Service-oriented architecture (SOA)
  - Software as a Service (SaaS)
  - Platform as a Service (PaaS)
  - Data as a Service(DaaS)

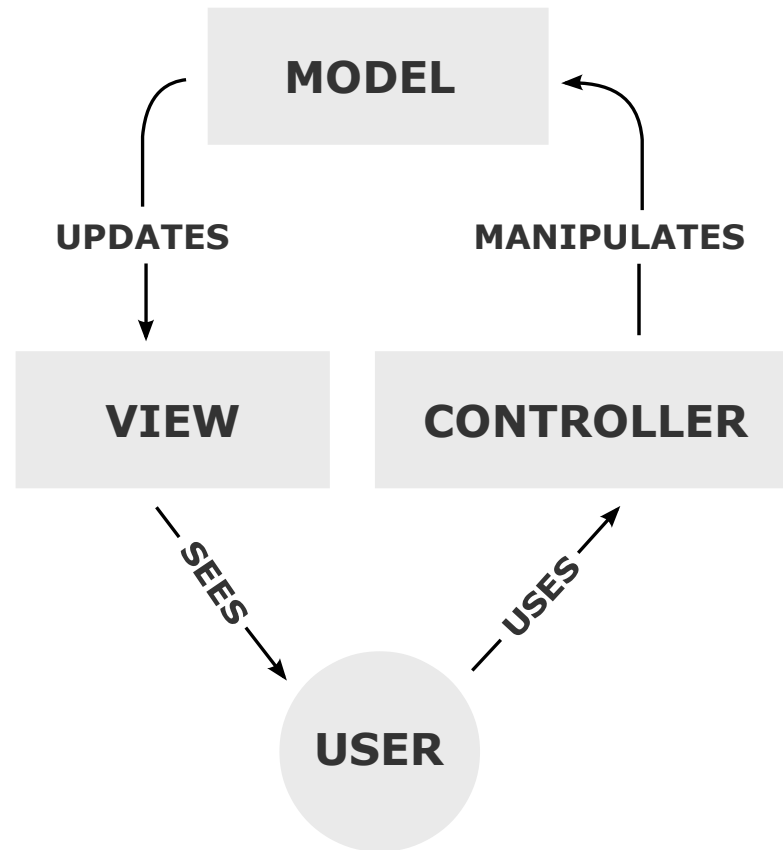
# Architectures

- Software design patterns
  - Model-View-Controller (MVC)
  - Model-View-Presenter (MVP)
  - Model-View-ViewModel (MVVM)
  - Etc.

# Architectures

- Model-View-Controller (MVC)
  - It is a software architectural pattern for implementing user interfaces on computers.
  - It divides a given software application into three interconnected parts, so as to separate internal representations of information from the ways that information is presented to or accepted from the user.

# Architectures



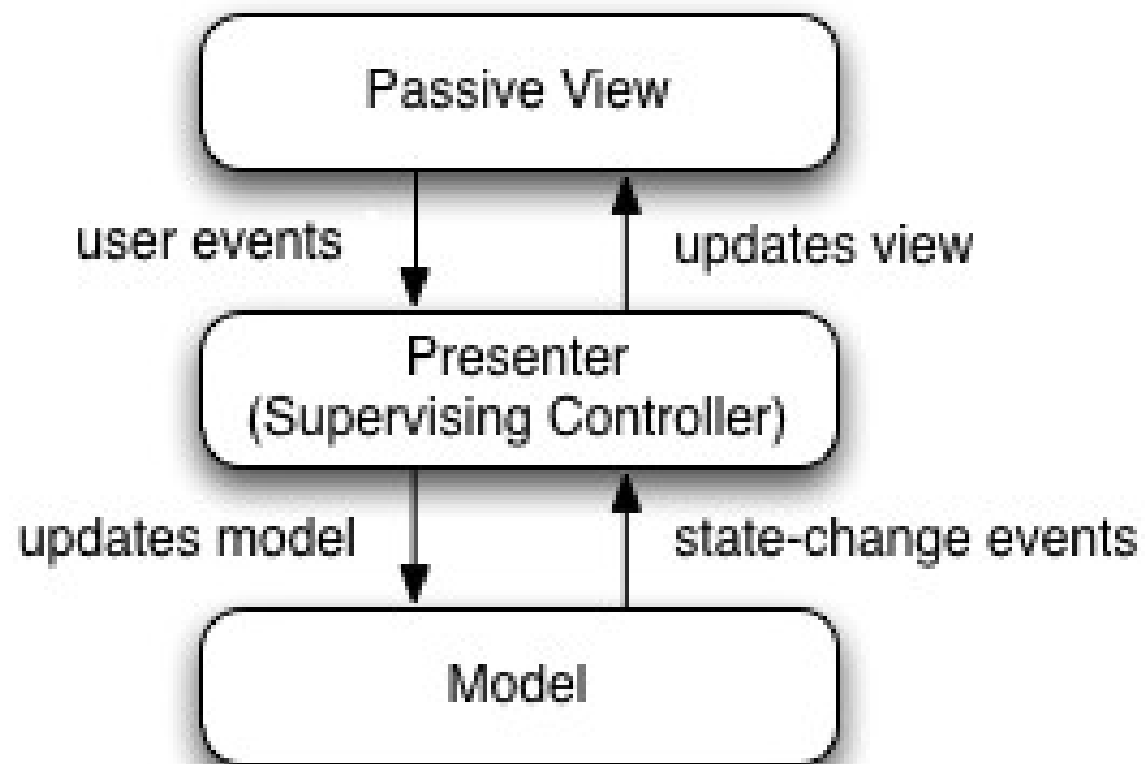
# Architectures

- Model-View-Controller (MVC)
  - The **model** directly manages the data, logic and rules of the application.
  - A **view** can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.
  - The third part, the **controller**, accepts input and converts it to commands for the model or view.

# Architectures

- Model-View-Presenter (MVP)
  - It is a derivation of the model–view–controller (MVC) architectural pattern, and is used mostly for building user interfaces.
  - In MVP the presenter assumes the functionality of the "middle-man". In MVP, all presentation logic is pushed to the presenter.

# Architectures



# Architectures

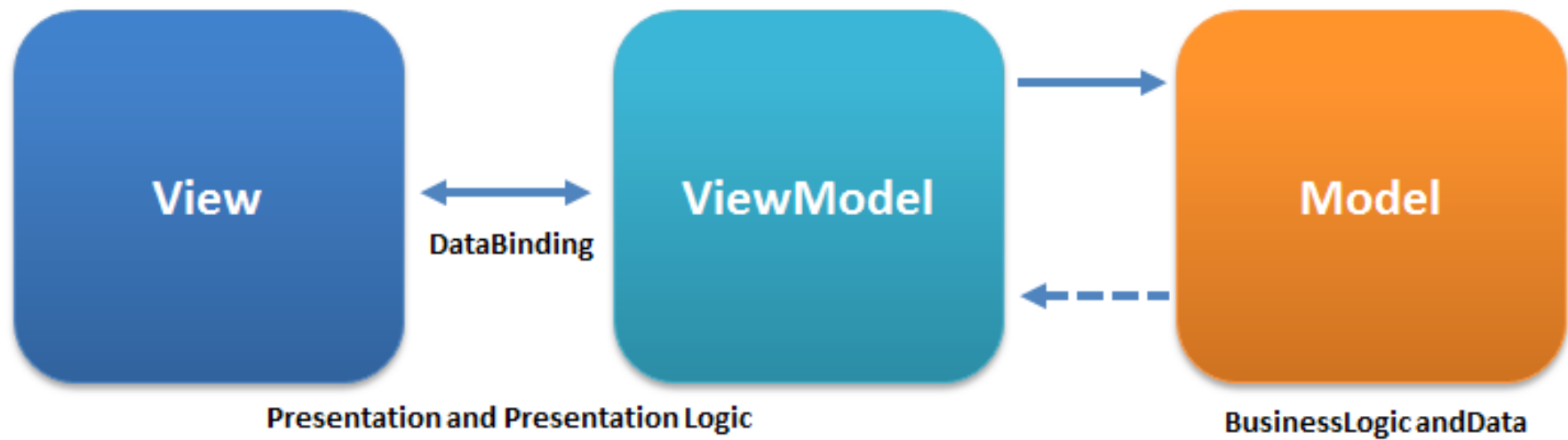
- Model-View-Presenter (MVP)
  - The **model** is an interface defining the data to be displayed or otherwise acted upon in the user interface.
  - The **presenter** acts upon the model and the view. It retrieves data from repositories (the model), and formats it for display in the view.
  - The **view** is a passive interface that displays data (the model) and routes user commands (events) to the presenter to act upon that data.



# Architectures

- Model-View-ViewModel (MVVM)
  - It facilitates a separation of development of the GUI from development of the business logic or back-end logic (the data model).
  - The view model of MVVM is a value converter; meaning the view model is responsible for exposing (converting) the data objects from the model in such a way that objects are easily managed and presented.
  - In this respect, the view model is more model than view, and handles most if not all of the view's display logic.
  - The view model may implement a mediator pattern, organizing access to the back-end logic around the set of use cases supported by the view.

# Architectures



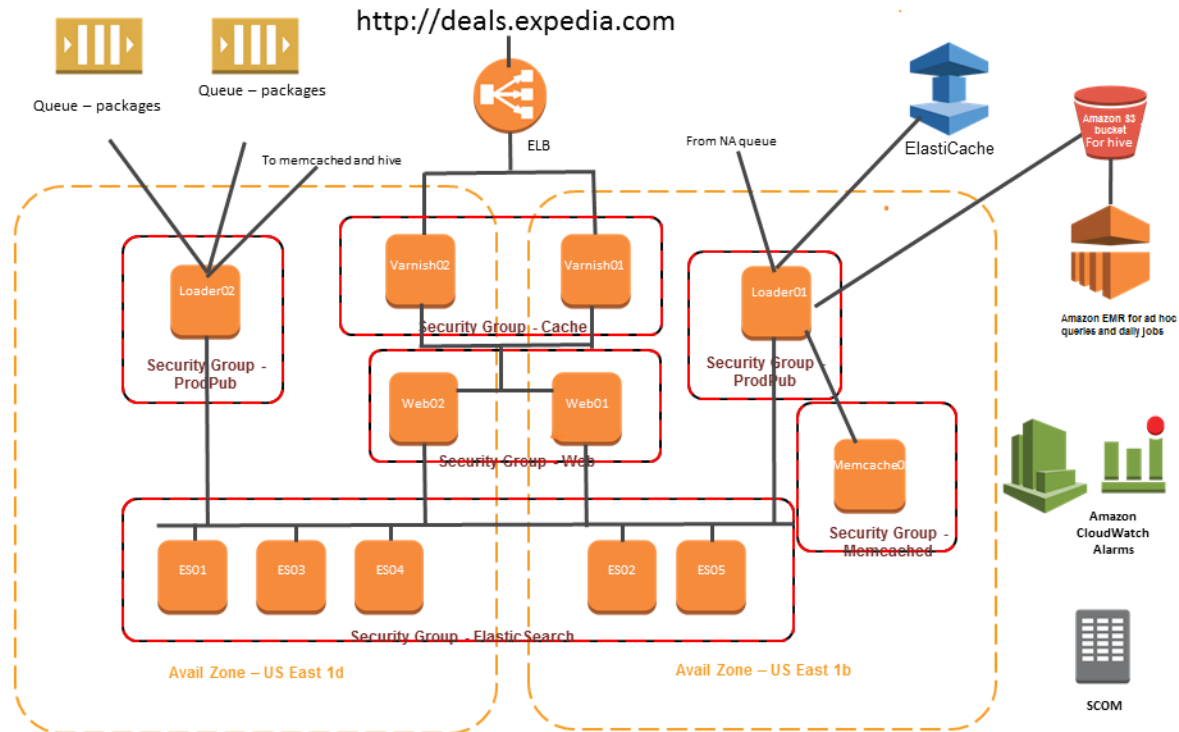
# Architectures

- Model-View-ViewModel (MVVM)
  - **Model** refers either to a domain model, which represents real state content (an object-oriented approach), or to the data access layer, which represents content (a data-centric approach).
  - As in the MVC and MVP patterns, the **view** is the structure, layout, and appearance of what a user sees on the screen.
  - The **view model** is an abstraction of the view exposing public properties and commands. Instead of the controller of the MVC pattern, or the presenter of the MVP pattern, MVVM has a binder. In the view model, the binder mediates communication between the view and the data binder.[clarification needed]  
The view model has been described as a state of the data in the model.
  - The **binder** frees the developer from being obliged to write boiler-plate logic to synchronize the view model and view.

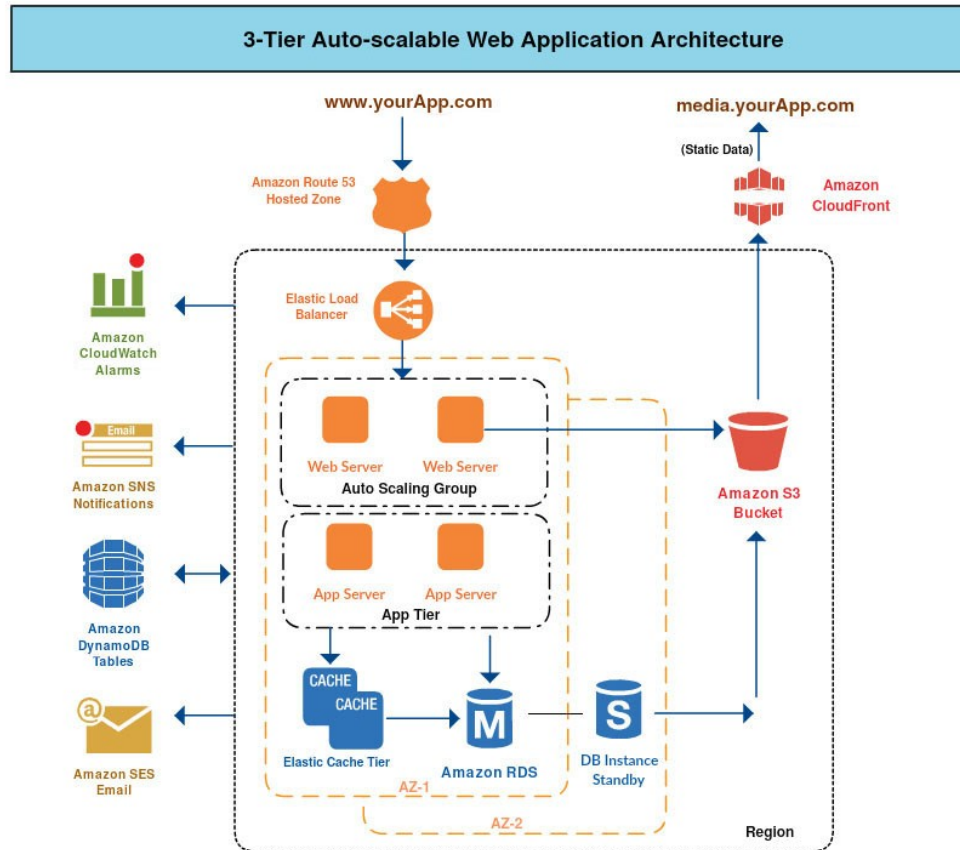
# Architectures

- Hardware architectures
  - Several layers
  - Large amount of heterogeneous components
  - Each component has a well defined functionality
  - Need to be orchestrated to reach the expected requirements, helping software architecture.
  - Similar principles to SOA's
  - Infrastructure as a Service (IaaS)

# Architectures



# Architectures



# Patterns & Anti-patterns

- Software design patterns
- Publish/subscribe pattern
- Request/reply pattern
- Message exchange patterns

# Patterns & Anti-patterns

- Software design patterns
  - In software engineering, a **software design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design.



# Patterns & Anti-patterns

- Software design patterns
  - Creational patterns
  - Structural patterns
  - Behavioral patterns
  - Concurrency patterns

# Patterns & Anti-patterns

- Software design patterns
  - Creational patterns
    - Abstract factory
      - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
    - Builder
      - Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.
    - **Factory method**
      - Define an interface for creating a single object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses (**dependency injection**).
    - **Lazy initialization**
      - Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.
    - **Object pool**
      - Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.
    - **Prototype**
      - Specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.
    - **Singleton**
      - Ensure a class has only one instance, and provide a global point of access to it.

# Patterns & Anti-patterns

- Software design patterns
  - Structural patterns
    - **Adapter**
      - Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces.
    - **Composite**
      - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
    - **Decorator**
      - Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.
    - **Facade**
      - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
    - **Front controller**
      - The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.
    - **Proxy**
      - Provide a surrogate or placeholder for another object to control access to it.

# Patterns & Anti-patterns

- Software design patterns
  - Behavioral patterns
    - **Chain of responsibility**
      - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
    - **Command**
      - Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations.
    - **Iterator**
      - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
    - **Observer (Publish/subscribe)**
      - Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
    - **State**
      - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
    - **Strategy**
      - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
    - **Template method**
      - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
    - **Visitor**
      - Represent an operation to be performed on the elements of an object structure. Visitor lets a new operation be defined without changing the classes of the elements on which it operates.

# Patterns & Anti-patterns

- Software design patterns
  - Concurrency patterns
    - **Active Object**
      - Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.
    - **Binding properties**
      - Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way.
    - **Double-checked locking (anti-pattern)**
    - **Event-based asynchronous**
      - Addresses problems with the asynchronous pattern that occur in multithreaded programs.
    - **Join**
      - Join-pattern provides a way to write concurrent, parallel and distributed programs by message passing.
    - **Lock**
      - One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.
    - **Messaging design pattern (MDP)**
      - Allows the interchange of information (i.e. messages) between components and applications.
    - **Reactor**
      - A reactor object provides an asynchronous interface to resources that must be handled synchronously.
    - **Scheduler**
      - Explicitly control when threads may execute single-threaded code.
    - **Thread pool**
      - A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads.

# Patterns & Anti-patterns

- Publish/subscribe pattern
  - In software architecture, **publish–subscribe** is a messaging pattern where senders of messages, called publishers, do not program the messages to be sent directly to specific receivers, called subscribers, but instead characterize published messages into classes without knowledge of which subscribers, if any, there may be.
  - Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.
  - Publish–subscribe is a sibling of the **message queue** paradigm.
  - Most messaging systems support both the pub/sub and message queue models.
  - This pattern **provides greater network scalability**.

# Patterns & Anti-patterns

- Request/reply pattern
  - **Request-response** is a message exchange pattern in which a requestor sends a request message to a replier system which receives and processes the request, ultimately returning a message in response.
  - This is a simple, but powerful messaging pattern which allows two applications to have a two-way conversation with one another over a channel.
  - This pattern is especially common in **client-server architectures**.
  - For simplicity, this pattern is typically implemented in a purely **synchronous** fashion, as in web service calls over HTTP.
  - However, request-response may also be implemented **asynchronously**.

# Patterns & Anti-patterns

- Message Exchange Patterns
  - There are multiple types of web services.
  - Some web services a client calls to obtain some information.
  - For instance, a client may call a weather web service to read weather information.
  - These are typical **read-only web services**.
  - A read-only web service may in practice send an empty request to the web service, which then sends the data back.
  - So, even if a web service is read-only, the client might actually have to send some data (a minimal request) to the web service to obtain the data it wants to read.



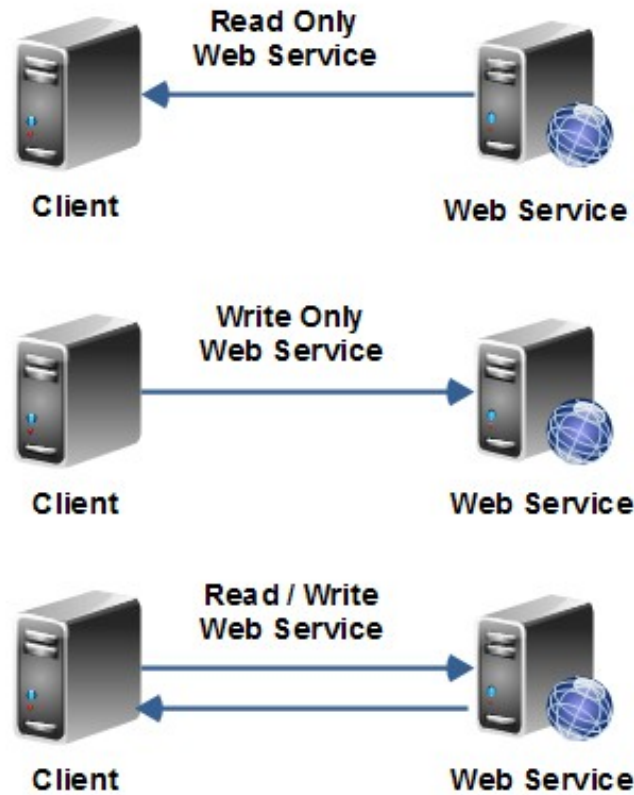
# Patterns & Anti-patterns

- Message Exchange Patterns
  - Other **web services** are more **write-only** kind of web services.
  - For instance, you may transfer data to a web service at regular intervals.

# Patterns & Anti-patterns

- Message Exchange Patterns
  - And then others are **read-write** services where it makes sense to both send data to the **web service**, and receive data back again.

# Patterns & Anti-patterns



# Patterns & Anti-patterns

- Anti-patterns
  - Better not to call for them. ;)
  - Please, check bibliography if interested.

# Protocol Stack

# Protocol Stack

- XML
- SOAP
- WSDL
- UDDI
- WS-Security

# XML

- XML stands for **EXtensible Markup Language**
- Designed to store and transport data
- Human- and machine-readable
- Self-descriptive

# XML

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```



# XML

- XML declaration
  - XML documents may begin with it.
  - `<?xml version="1.0" encoding="UTF-8"?>`
- Tag
  - A tag is a markup construct that begins with < and ends with >
  - `<tag>`, `</tag>`, `<tag/>`
- Element
  - An element is a component that either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag.
  - The characters between the start-tag and end-tag, if any, are the element's content.
  - `<greeting>Hello, world!</greeting>`
- Attribute
  - An attribute is a markup construct consisting of a name–value pair that exists within a start-tag or empty-element tag.
  - `<step number="3">Connect A to B.</step>`

# XML

- *Go to **Exercise 1***

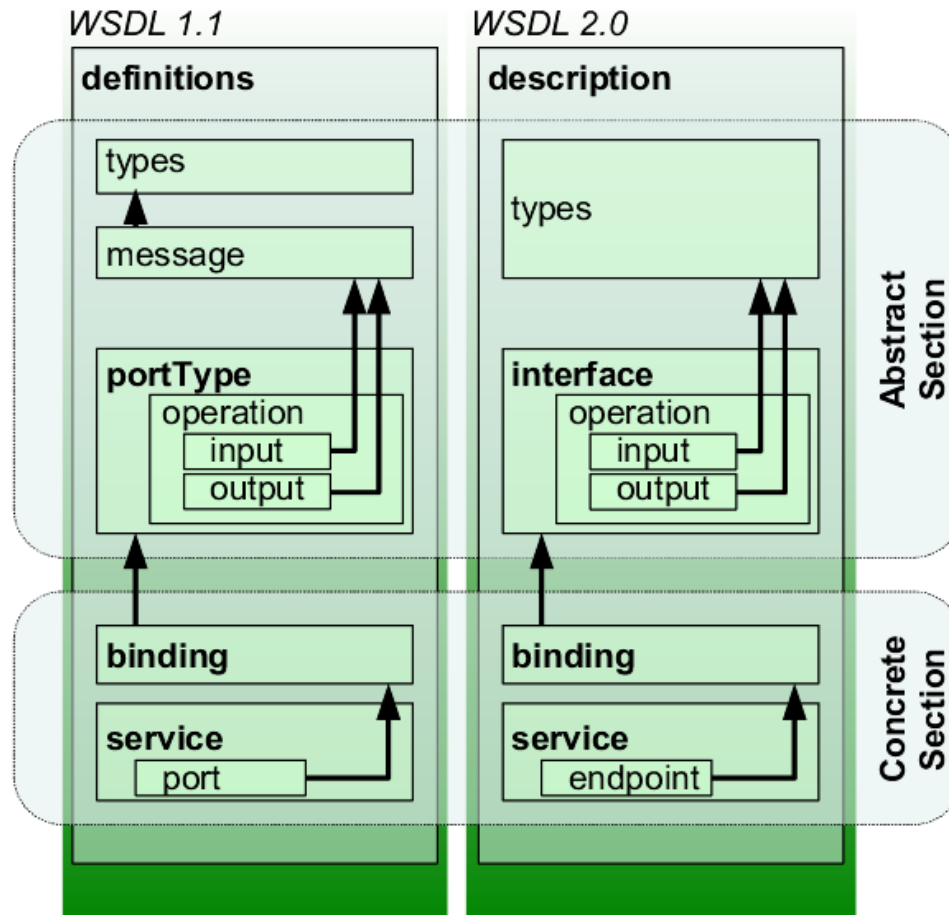
# SOAP

- SOAP stands for **Simple Object Access Protocol**
- Application communication protocol
- Format for sending and receiving messages
- Platform independent
- Based on XML

# WSDL

- WSDL stands for **Web Services Description Language**
- Used to describe web services
- Written in XML

# WSDL



# WSDL

- Service
  - Contains a set of system functions that have been exposed to the Web-based protocols.
- Endpoint
  - Defines the address or connection point to a Web service. It is typically represented by a simple HTTP URL string.
- Binding
  - Specifies the interface and defines the SOAP binding style (RPC/Document) and transport (SOAP Protocol). The binding section also defines the operations.
- Interface
  - Defines a Web service, the operations that can be performed, and the messages that are used to perform the operation.
- Operation
  - Defines the SOAP actions and the way the message is encoded, for example, "literal." An operation is like a method or function call in a traditional programming language.
- Types
  - Describes the data. The XML Schema language (also known as XSD) is used (inline or referenced) for this purpose.

# WSDL

- Example:
  - [https://en.wikipedia.org/wiki/Web\\_Services\\_Description\\_Language#Example\\_WSDL\\_file](https://en.wikipedia.org/wiki/Web_Services_Description_Language#Example_WSDL_file)

# UDDI

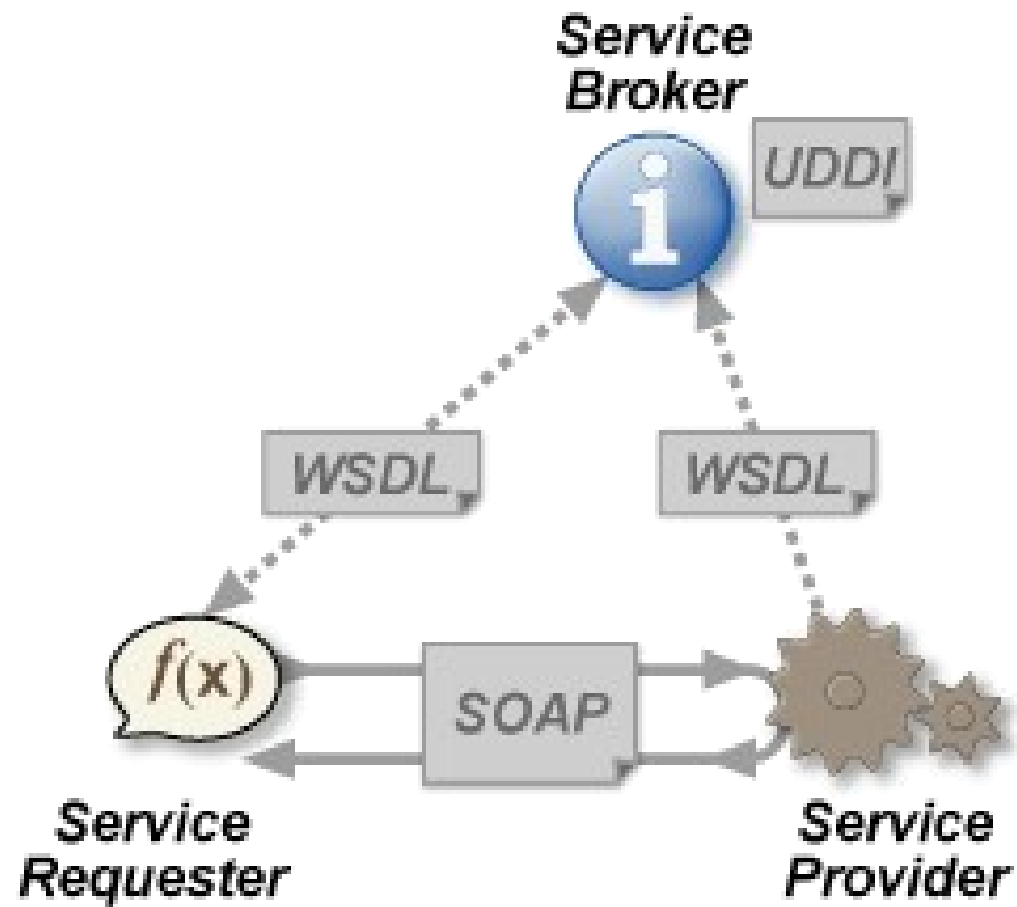
- UDDI is a platform-independent framework for describing services, discovering businesses, and integrating business services by using the Internet.
  - UDDI stands for Universal Description, Discovery and Integration
  - Directory for storing information about web services
  - Directory of web service interfaces described by WSDL
  - UDDI communicates via SOAP



# UDDI

- A UDDI business registration consists of three components:
  - White Pages
    - Address, contact, and known identifiers
  - Yellow Pages
    - Industrial categorizations based on standard taxonomies
  - Green Pages
    - Technical information about services exposed by the business

# UDDI



# WS-Security

- Extension to SOAP to apply security to Web services
- Specifies how integrity and confidentiality can be enforced on messages and allows the communication of various security token formats, such as Security Assertion Markup Language (**SAML**), **Kerberos**, and **X.509**
- Its main focus is the use of **XML Signature** and **XML Encryption** to provide end-to-end security

# WS-Security

- WS-Security describes three main mechanisms:
  - How to **sign** SOAP messages to assure integrity
    - Signed messages also provide non-repudiation
  - How to **encrypt** SOAP messages to assure confidentiality
  - How to **attach** security tokens to ascertain the sender's identity

# WS-Security

- The specification allows a variety of signature formats, encryption algorithms and multiple trust domains, and is open to various security token models, such as:
  - **X.509 certificates**
  - Kerberos tickets
  - **User ID/Password** credentials
  - SAML Assertions
  - Custom-defined tokens
- WS-Security incorporates **security** features in the **header** of a **SOAP message**, working in the **application layer**

# WS-Security

- These mechanisms by themselves do not provide a complete security solution for Web services
- In general, WSS by itself does not provide any guarantee of security
- When implementing and using the framework and syntax, it is up to the **implementor** to ensure that the result is not vulnerable
- Key management, trust bootstrapping, federation and agreement on the technical details (ciphers, formats, algorithms) is outside the scope of WS-Security

# WS-Security

- End-to-end security
  - If a SOAP intermediary is required, and the intermediary is not or is less trusted, messages need to be signed and optionally encrypted. This might be the case of an application-level proxy at a network perimeter that will terminate TCP connections
- Non-repudiation
  - The standard method for non-repudiation is to **write transactions to an audit trail** that is subject to specific security safeguards
  - However, if the audit trail is not sufficient, digital signatures may provide a better method to enforce non-repudiation
- Alternative transport bindings
- Reverse proxy/common security token

# WS-Security

- Issues
  - Performance
    - WS-Security adds significant overhead to SOAP processing due to the increased size of the message on the wire, XML and cryptographic processing, requiring faster CPUs and more memory and bandwidth
- Alternatives
  - In point-to-point situations confidentiality and data integrity can also be enforced on Web services through the use of **Transport Layer Security (TLS)**, for example, by sending messages over **HTTPS**
    - Problem: proxy-servers routing
    - Solution: copy of the client's key and certificate into the proxy



Types

# Types

- SOAP
- REST

# SOAP

- SOAP stands for **Simple Object Access Protocol**
- Application communication protocol
- Format for sending and receiving messages
- Platform independent
- Based on XML

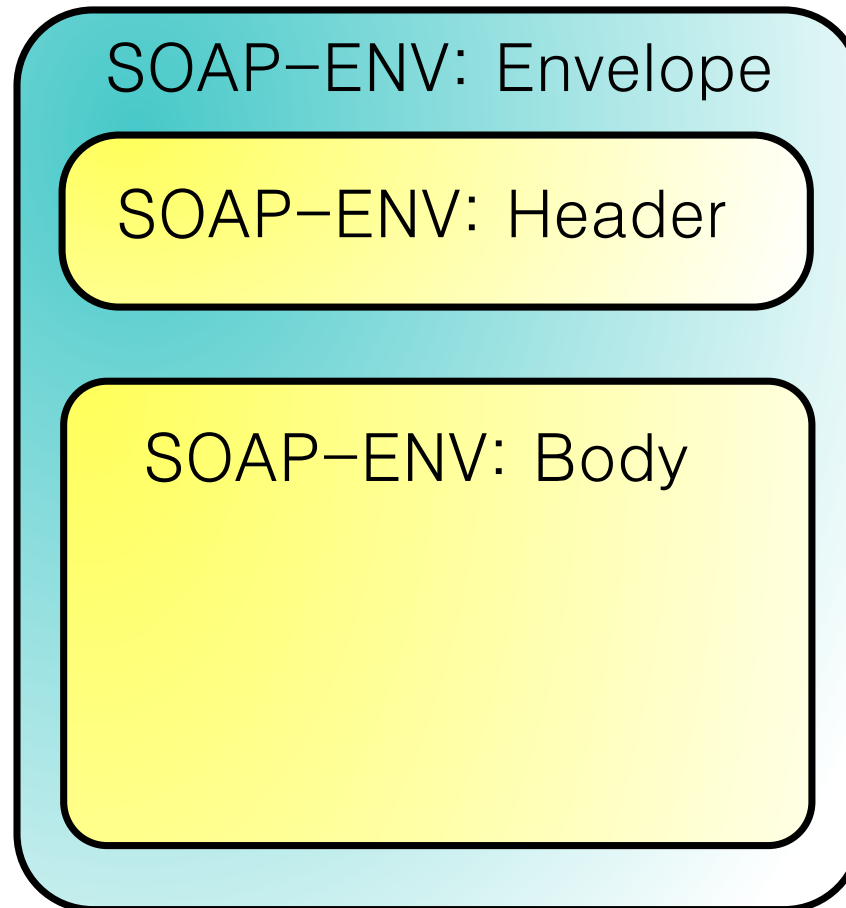
# SOAP

- Specification
  - The SOAP specification defines the messaging framework, which consists of:
    - The SOAP **processing model** defining the rules for processing a SOAP message
    - The SOAP **extensibility model** defining the concepts of SOAP features and SOAP modules
    - The SOAP **underlying protocol** binding framework describing the rules for defining a binding to an underlying protocol that can be used for exchanging SOAP messages between SOAP nodes
    - The SOAP **message** construct defining the structure of a SOAP message

# SOAP

- Processing model
  - The SOAP processing model describes a distributed processing model, its participants, the SOAP nodes, and how a SOAP receiver processes a SOAP message.
  - The following SOAP nodes are defined:
    - SOAP sender
    - SOAP receiver
    - SOAP message path
    - Initial SOAP sender (Originator)
    - SOAP intermediary
    - Ultimate SOAP receiver

# SOAP



# SOAP

- Message
  - A SOAP message is an ordinary XML document containing the following elements:
    - An **Envelope** (required) element that identifies the XML document as a SOAP message
    - A **Header** element that contains header information
    - A **Body** (required) element that contains call and response information
    - A **Fault** element containing errors and status information

# SOAP

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"
```

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock/Surya">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```



# REST

- Material:
  - <https://github.com/alejopj/restful-apis>

JAX-WS

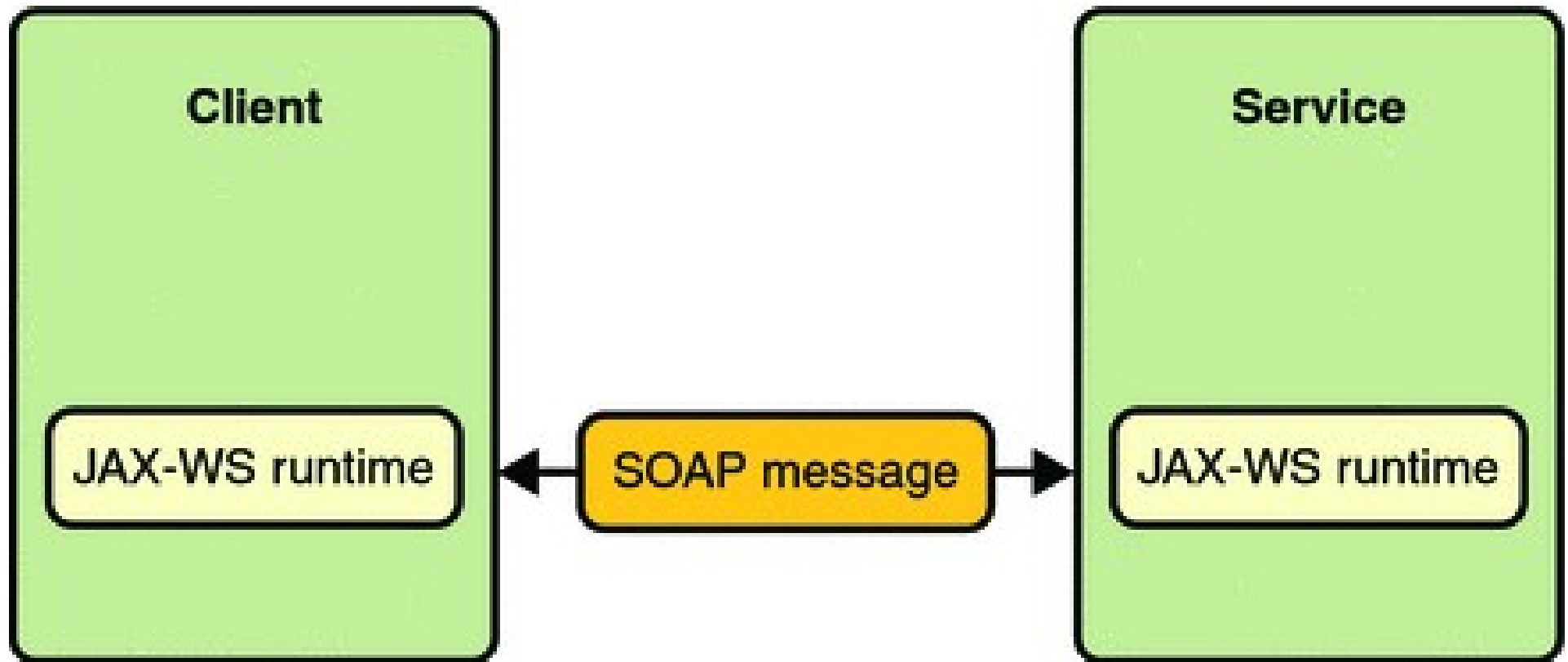
# JAX-WS

- Wsimport
- Wsgen

# JAX-WS

- The **Java API for XML Web Services** (JAX-WS) is a Java programming language API for creating web services, particularly SOAP services
- JAX-WS provides many **annotations** to simplify the development and deployment for both web service clients and web service providers (endpoints)

# JAX-WS



# JAX-WS

- `javax.xml.ws`
  - Has the Core JAX-WS APIs
- `javax.xml.ws.http`
  - Has APIs specific to XML/HTTP Binding
- `javax.xml.ws.soap`
  - Has APIs specific to SOAP/HTTP Binding
- `javax.xml.ws.handler`
  - Has APIs for message handlers
- `javax.xml.ws.spi`
  - defines SPIs for JAX-WS
- `javax.xml.ws.spi.http`
  - Provides HTTP SPI that is used for portable deployment of JAX-WS in containers
- `javax.xml.ws.wsaddressing`
  - Has APIs related to WS-Addressing
- `javax.jws`
  - Has APIs specific to Java to WSDL mapping annotations
- `javax.jws.soap`
  - Has APIs for mapping the Web Service onto the SOAP protocol

# JAX-WS

- Example
  - Hello world!

# Wsimport

- The **wsimport** tool is used to parse an existing Web Services Description Language (**WSDL**) file and generate required files (**JAX-WS** portable artifacts) for web service client to access the published web services
- This wsimport tool is available in the **\$JDK/bin** folder



# Wsimport

- Example
  - Create a new Java project
  - Open a terminal on the project /src path
  - `wsimport -verbose -keep -s .`  
<http://localhost:8080/hello-world-service?wsdl>

# Wsgen

- The **wsgen** tool is used to parse an existing web service implementation class and generates required files (**JAX-WS** portable artifacts) for web service deployment
- This wsgen tool is available in **\$JDK/bin** folder
- Use cases:
  - Generate JAX-WS portable artifacts (Java files) for web service deployment
  - Generate WSDL and xsd files, for testing or web service client development

# Wsgen

- Example
  - Open a terminal on the server project /bin path
  - `wsgen -verbose -keep -cp .  
com.github.alejopj.ws.soap.helloworld.server.services.impl.HelloWorldServiceImpl`
  - `wsgen -verbose -keep -wsdl -cp .  
com.github.alejopj.ws.soap.helloworld.server.services.impl.HelloWorldServiceImpl`

# JAX-WS

- *Go to Exercise 2*

RESTful

# RESTful

- Jax-RS
- Annotations
- JAXB
- JSON

# Jax-RS

- **Java API for RESTful Web Services (JAX-RS)** is a Java programming language API spec that provides support in creating web services according to the Representational State Transfer (**REST**) architectural pattern
- JAX-RS uses **annotations**, introduced in Java SE 5, to simplify the development and deployment of web service clients and endpoints

# Annotations

- JAX-RS provides some annotations to aid in mapping a resource class (a POJO) as a web resource. The annotations include:
  - **@Path** specifies the relative path for a resource class or method
  - **@GET**, **@PUT**, **@POST**, **@DELETE** and **@HEAD** specify the HTTP request type of a resource
  - **@Produces** specifies the response Internet media types (used for content negotiation)
  - **@Consumes** specifies the accepted request Internet media types



# Annotations

- In addition, it provides further annotations to method parameters to pull information out of the request
- All the **@\*Param** annotations take a key of some form which is used to look up the value required
  - **@PathParam** binds the method parameter to a path segment
  - **@QueryParam** binds the method parameter to the value of an HTTP query parameter
  - **@MatrixParam** binds the method parameter to the value of an HTTP matrix parameter
  - **@HeaderParam** binds the method parameter to an HTTP header value
  - **@CookieParam** binds the method parameter to a cookie value
  - **@FormParam** binds the method parameter to a form value
  - **@DefaultValue** specifies a default value for the above bindings when the key is not found
  - **@Context** returns the entire context of the object (for example `@Context HttpServletRequest request`)

# Jax-RS

- Go to **Exercise 4**

# JAXB

- **Java Architecture for XML Binding (JAXB)** is a software framework that allows Java developers to map Java classes to XML representations
- **Features:**
  - **Marshal** (serialize) Java objects into XML
  - **Unmarshal** (deserialize) XML back into Java objects

# JAXB

- Example
  - Marshal & unmarshal with JAXB

# JAXB

- *Go to **Exercise 3***

# JSON

- JavaScript Object Notation
- The official Internet media type for JSON is “application/json”
- Data types:
  - Number
  - String
  - Boolean
  - Array
  - Object
  - null

# JSON

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

# JSON

- Go to **Exercise 5**



# JSON

- Example
  - Marshal & unmarshal with Jackson

# JSON

- Go to **Exercise 6**

</theory>

</br>

<practice>

# Exercises

# Exercises

## – **Exercise 1:**

- Create your own XML document containing several elements and tree levels
- Validate it using the link below:
  - <http://www.xmlvalidation.com/>

# Exercises

## – **Exercise 2:**

- Create your own SOAP service having at least an example of each CRUD operation working over a list of elements
- Create a SOAP service client of a SOAP service developed by any other participant in the course
- Import that client into a new project to play with every operation available



# Exercises

## – **Exercise 3:**

- Create a new project
- Add the xml file from the exercise 1 to the project
- Create a Java class representation of the xml object
- Read the xml from the file
- Print the xml
- Unmarshal it to the Java class
- Marshal it again
- Print the result

# Exercises

## – **Exercise 4:**

- Create your own REST service having at least an example of each CRUD operation working over a list of elements
- Create a REST service client of a REST service developed by any other participant in the course
- You can both start a new Java project nor use any other REST client tool

# Exercises

- **Bonus exercise:**
  - Repeat Exercise 4 but using complex objects

# Exercises

## – **Exercise 5:**

- Create your own JSON document containing several elements and tree levels
- Validate it using the link below:
  - <http://jsonlint.com/>

# Exercises

## – **Exercise 6:**

- Create a new project
- Add the json file from the exercise 5 to the project
- Create a Java class representation of the json object
- Read the json from the file
- Print the json
- Unmarshal it to the Java class
- Marshal it again
- Print the result

# Exercises

- **Bonus exercise:**
  - Repeat Exercise 5 but using complex objects

# Exercises

- **Final exercise:**
  - Group SOAP/REST project

Tools



# IDE

- Eclipse
  - <https://www.eclipse.org/>
  - .bashrc
    - alias eclipse=/home/user/installation-path/eclipse/eclipse

# Git

- GitEye
  - <http://www.collab.net/products/giteye>
  - .bashrc
    - alias giteye=/home/user/installation-path/GitEye-1.10.0-linux.x86\_64/GitEye
- SmartGit
  - <http://www.syntevo.com/smartgit/>

# REST

- Curl
- SoapUI
  - <https://www.soapui.org/>
- Firefox + plugins
  - REST
    - REST Easy
    - RESTED
    - RESTClient
  - JSON
    - JSON-handle
    - JSONView
- Chrome + plugins
  - Postman
  - <https://chrome.google.com/webstore/search/rest>

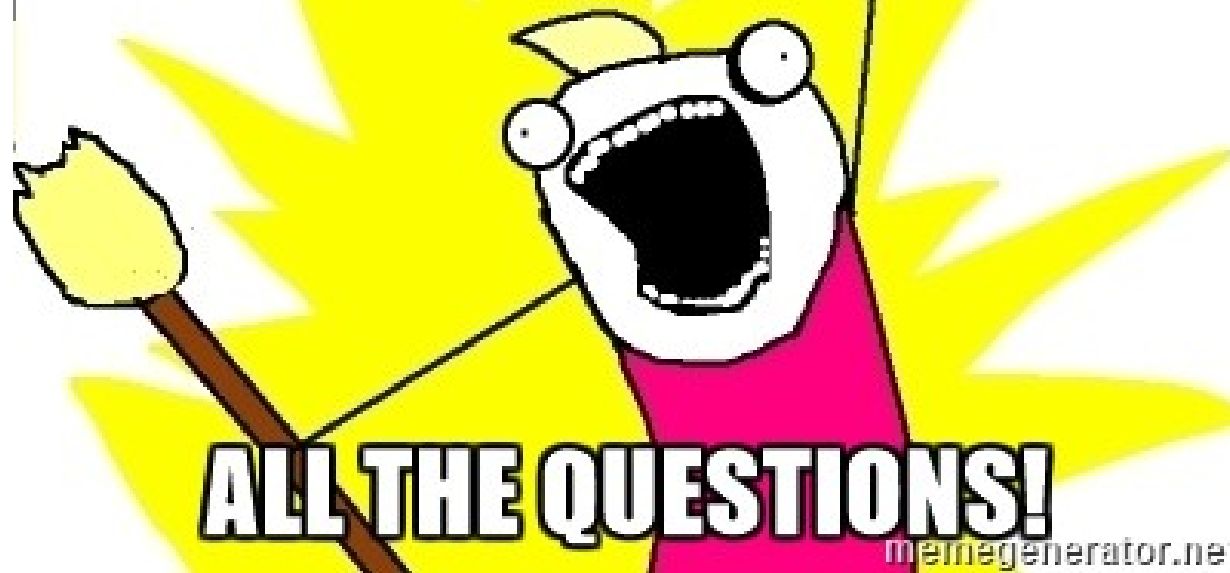
</practice>

# 'Quiz' Pro Quo

- Exam:
  - TBD
- Survey:
  - TBD

'Quiz' Pro Quo

**ANSWER**



</end>

Thank you!





<bibliography>

# Bibliography

- Web services
  - Jakob Jenkov's Tutorial
    - <http://tutorials.jenkov.com/web-services/index.html>
  - Architectural patterns
    - [https://en.wikipedia.org/wiki/Architectural\\_pattern](https://en.wikipedia.org/wiki/Architectural_pattern)
    - Service-Oriented Architecture (SOA)
      - [https://en.wikipedia.org/wiki/Service-oriented\\_architecture](https://en.wikipedia.org/wiki/Service-oriented_architecture)
    - Cloud computing
      - [https://en.wikipedia.org/wiki/Cloud\\_computing](https://en.wikipedia.org/wiki/Cloud_computing)
    - Software design architectures
      - <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>
      - <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter>
      - <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel>

# Bibliography

- Web services
  - Patterns & anti-patterns
    - Software design patterns
      - [https://en.wikipedia.org/wiki/Software\\_design\\_pattern](https://en.wikipedia.org/wiki/Software_design_pattern)
    - Publish/subscribe pattern
      - [https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe\\_pattern](https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern)
    - Request/reply pattern
      - <https://en.wikipedia.org/wiki/Request%E2%80%93response>
    - Message exchange pattern
      - [https://en.wikipedia.org/wiki/Messaging\\_pattern](https://en.wikipedia.org/wiki/Messaging_pattern)
    - Software design anti-patterns
      - [https://en.wikipedia.org/wiki/Anti-pattern#Software\\_engineering](https://en.wikipedia.org/wiki/Anti-pattern#Software_engineering)

# Bibliography

- Protocol stack
  - XML
    - <http://www.w3schools.com/xml/default.asp>
    - <https://en.wikipedia.org/wiki/XML>
  - SOAP
    - [http://www.w3schools.com/xml/xml\\_soap.asp](http://www.w3schools.com/xml/xml_soap.asp)
    - <https://en.wikipedia.org/wiki/SOAP>
  - WSDL
    - [http://www.w3schools.com/xml/xml\\_wsdl.asp](http://www.w3schools.com/xml/xml_wsdl.asp)
    - [https://en.wikipedia.org/wiki/Web\\_Services\\_Description\\_Language](https://en.wikipedia.org/wiki/Web_Services_Description_Language)
  - UDDI
    - [https://en.wikipedia.org/wiki/Web\\_Services\\_Discovery#Universal\\_Description\\_Discovery\\_and\\_Integration](https://en.wikipedia.org/wiki/Web_Services_Discovery#Universal_Description_Discovery_and_Integration)
  - WS-Security
    - <https://en.wikipedia.org/wiki/WS-Security>

# Bibliography

- Types
  - SOAP
    - [http://www.w3schools.com/xml/xml\\_soap.asp](http://www.w3schools.com/xml/xml_soap.asp)
    - <https://en.wikipedia.org/wiki/SOAP>
  - REST
    - <https://github.com/alejopj/restful-apis>

# Bibliography

- JAX-WS
  - Wikipedia
    - [https://en.wikipedia.org/wiki/Java\\_API\\_for\\_XML\\_Web\\_Services](https://en.wikipedia.org/wiki/Java_API_for_XML_Web_Services)
  - Java 2 Blog
    - <http://www.java2blog.com/2013/03/jaxws-web-service-eclipse-tutorial.html>
  - Mkyong
    - <http://www.mkyong.com/tutorials/jax-ws-tutorials/>
  - Memory not found
    - <http://memorynotfound.com/jax-ws-soap-web-service-example/>

# Bibliography

- RESTful
  - Github
    - <https://github.com/alejopj/restful-apis>
  - Wikipedia
    - [https://en.wikipedia.org/wiki/Java\\_API\\_for\\_RESTful\\_Web\\_Services](https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services)
    - [https://en.wikipedia.org/wiki/Java\\_Architecture\\_for\\_XML\\_Binding](https://en.wikipedia.org/wiki/Java_Architecture_for_XML_Binding)
  - Mkyong
    - <http://www.mkyong.com/tutorials/jax-rs-tutorials/>
    - <https://www.mkyong.com/java/jackson-2-convert-java-object-to-from-json/>

</bibliography>