Java Web Services

CIXUG

CENTROS DE FORMACIÓN
GRUPO
ACADEMIA POSTAL

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 3rd to 7th (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

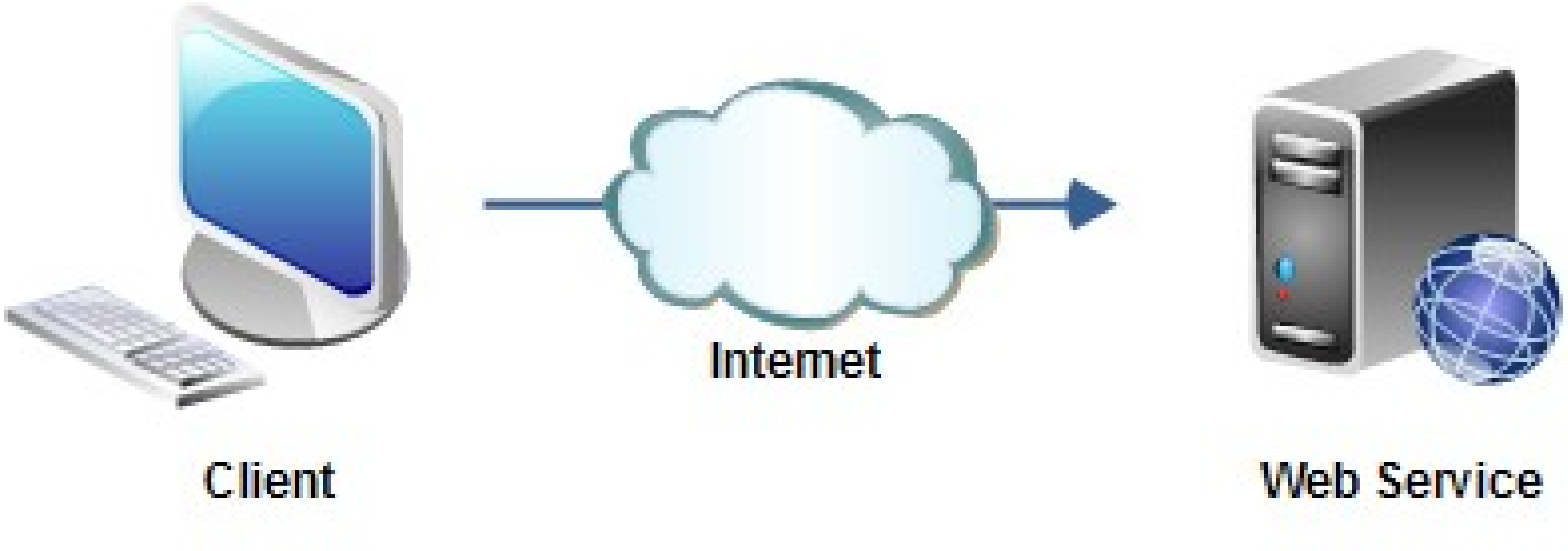
Web services

- 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.

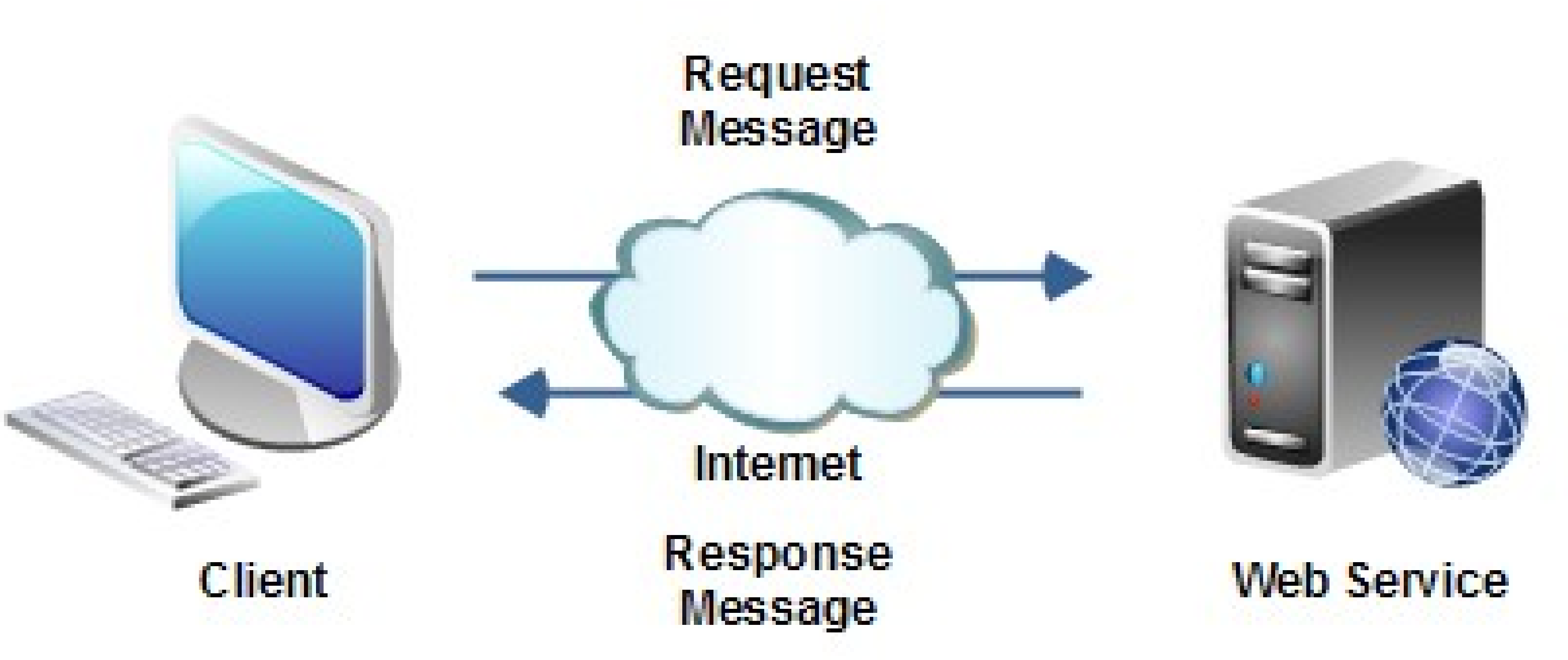
What is a WS?



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

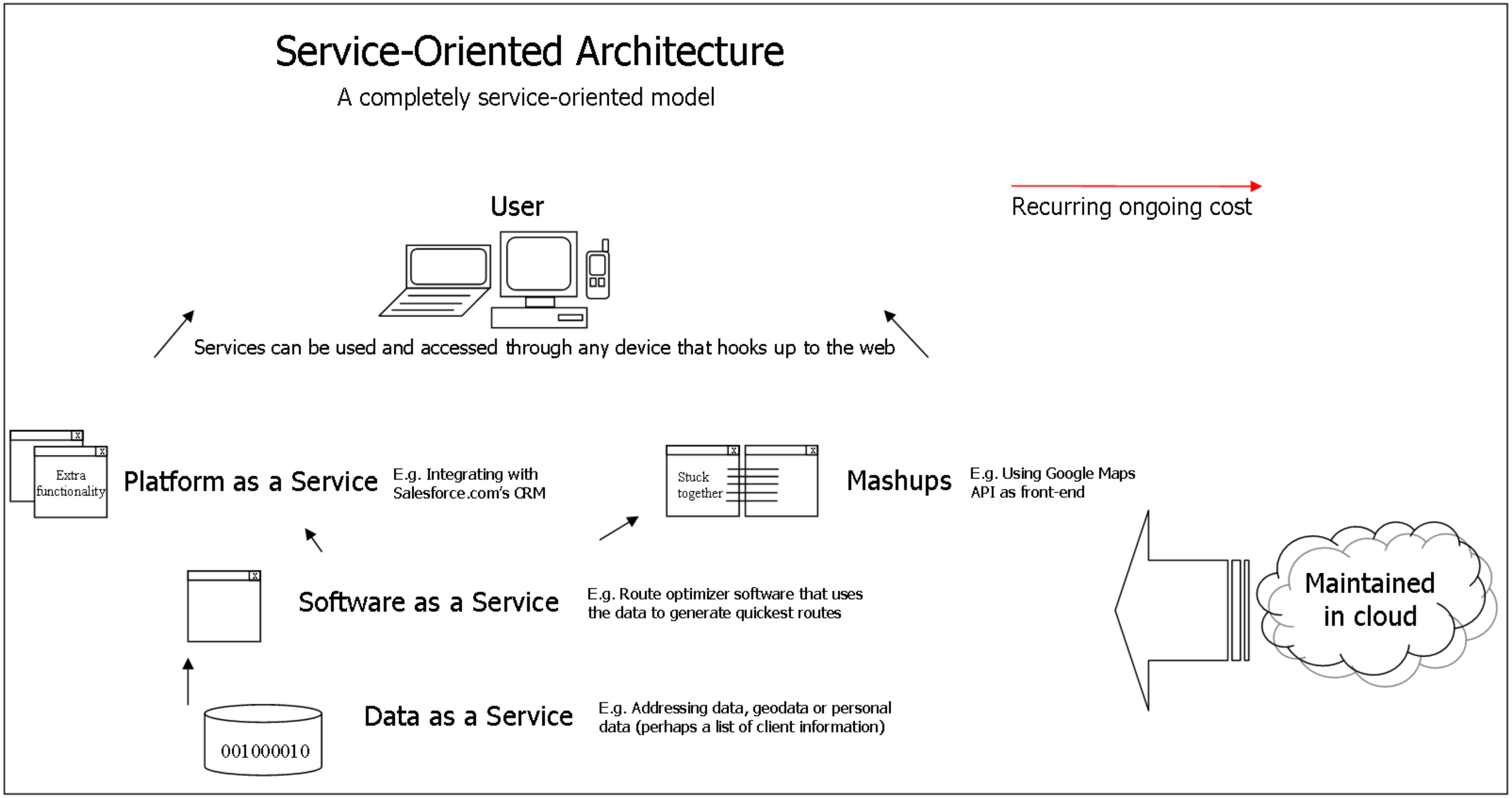
Architectures

- 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
User
Recurring ongoing cost
Services can be used and accessed through any device that hooks up to the web
Extra functionality
Platform as a Service
E.g. Integrating with Salesforce.com's CRM
Stuck together
Mashups
E.g. Using Google Maps API as front-end
Software as a Service
E.g. Route optimizer software that uses the data to generate quickest routes
Maintained in cloud
001000010
Data as a Service
E.g. Addressing data, geodata or personal data (perhaps a list of client information)

Architectures

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

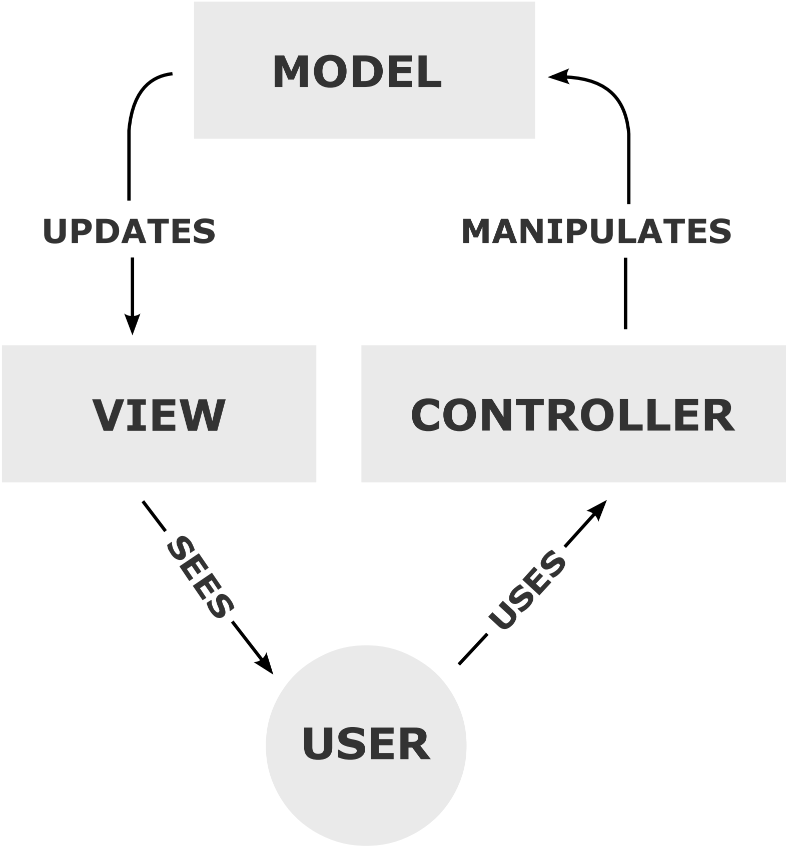
Architectures

- 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



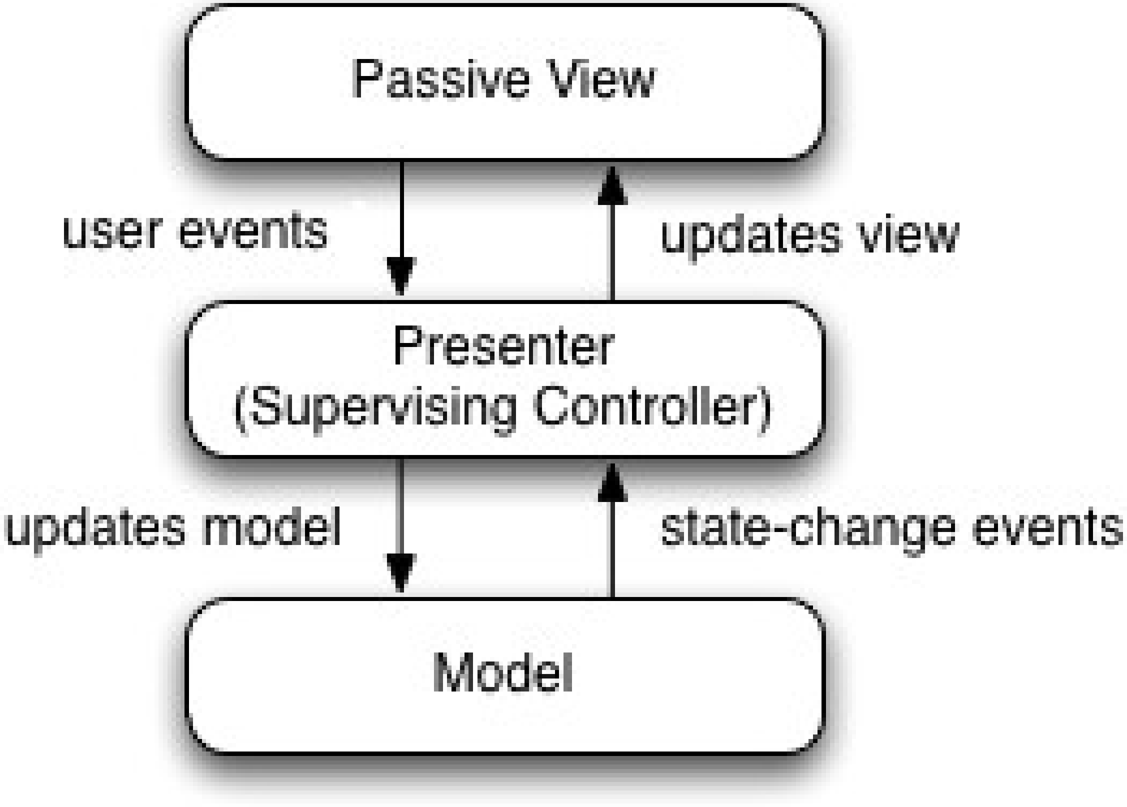
Architectures

- 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



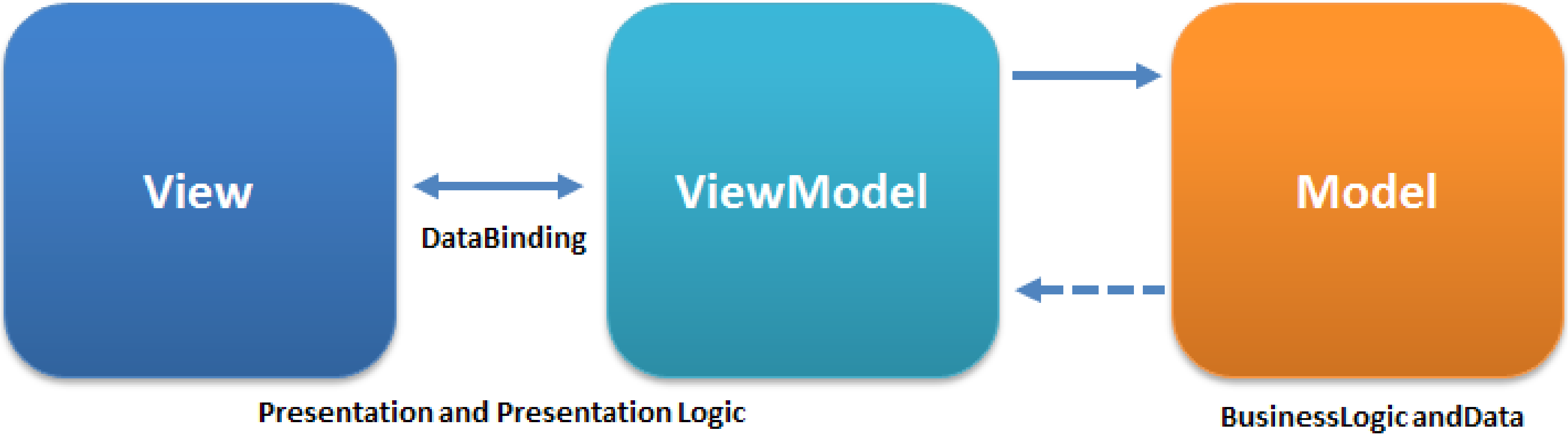
Architectures

- 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



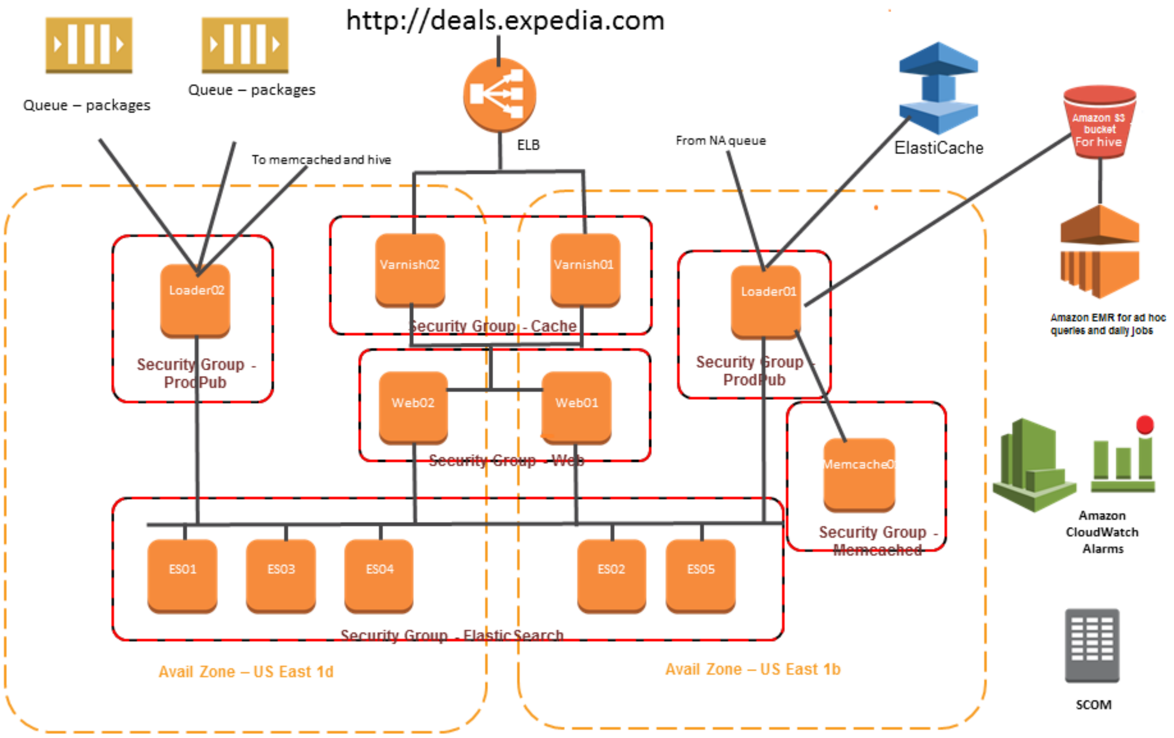
Architectures

- 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.

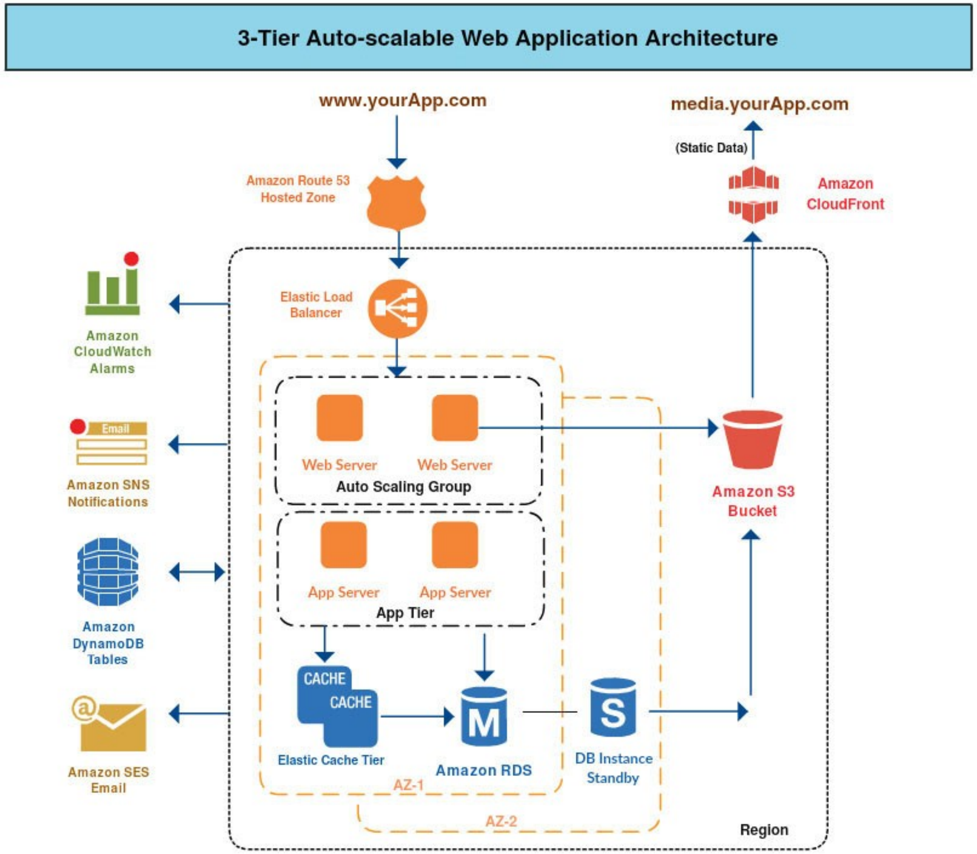
Architectures

- 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.

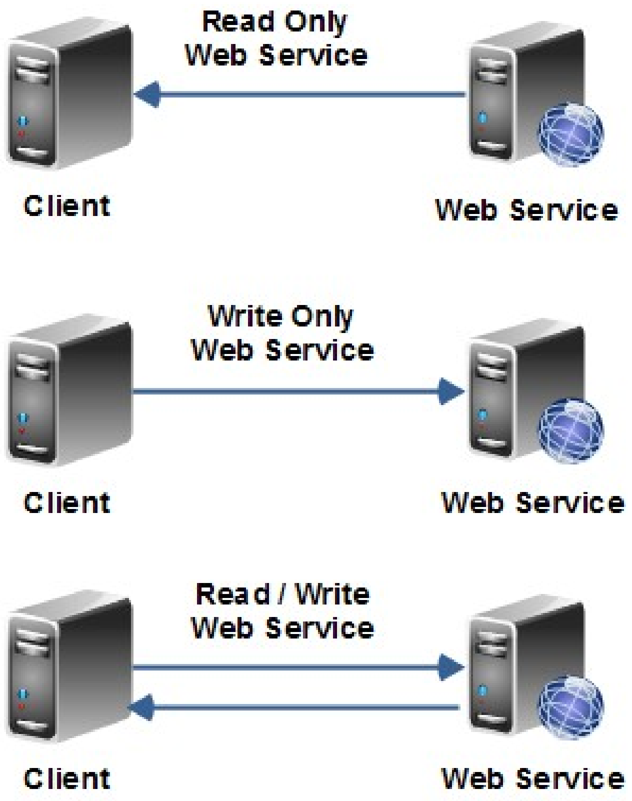
Patterns & Anti-patterns

- 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

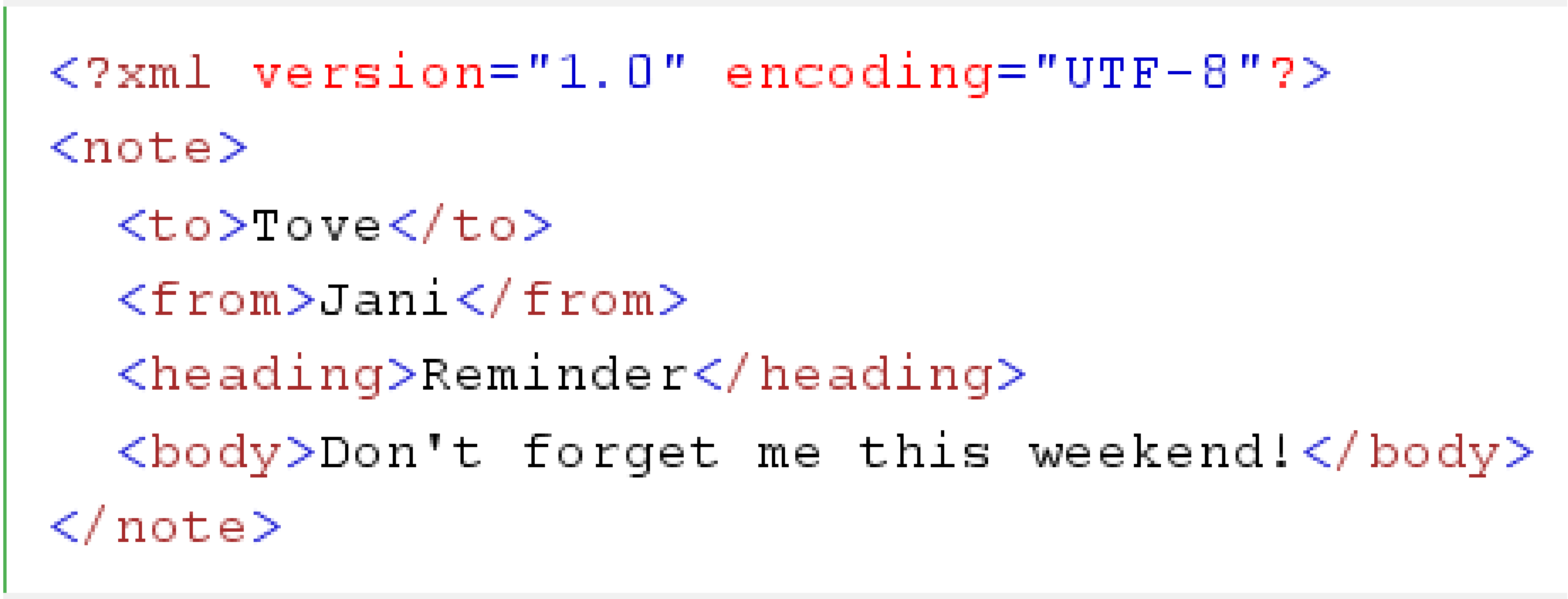
Protocol Stack

- 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>*

Exercises

- **Exercise 1**:
  - Create your own XML document containing several elements and tree levels
  - Validate it using the link below:
    - http://www.xmlvalidation.com/

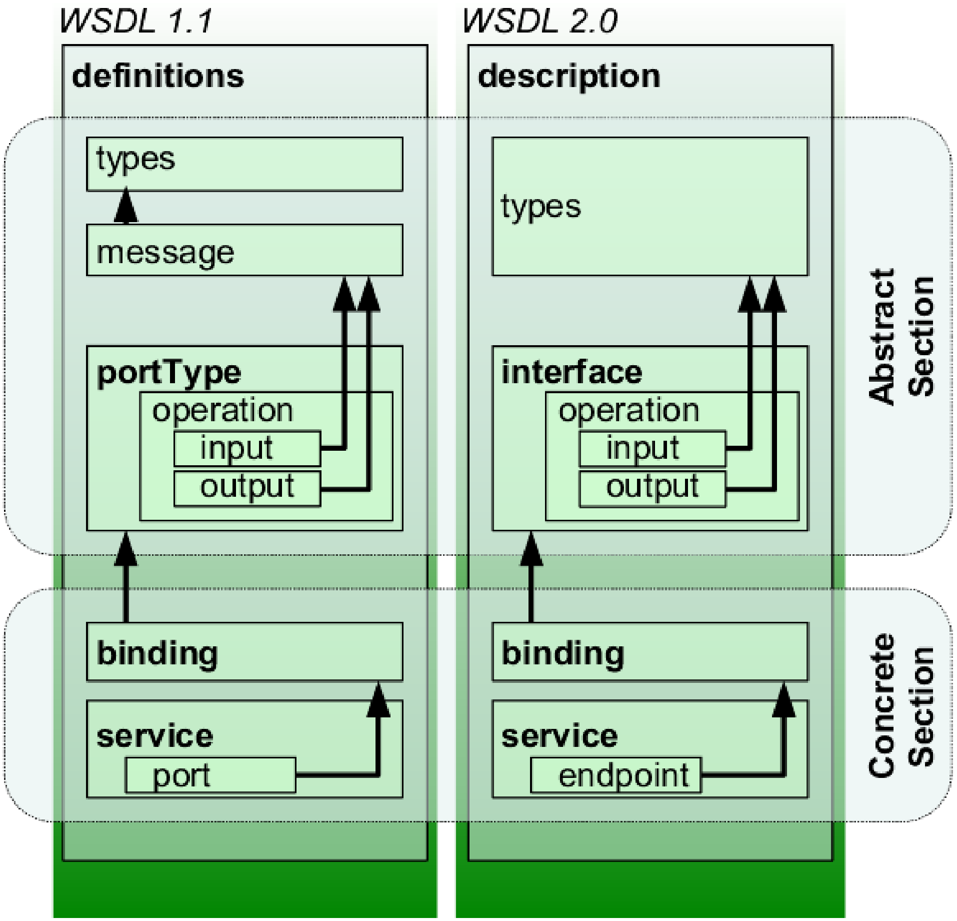
SOAP

- 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 1.1
definitions
types
message
portType
operation
input
output
binding
service
port
WSDL 2.0
description
types
interface
operation
input
output
binding
service
endpoint
Abstract Section
Concrete Section

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

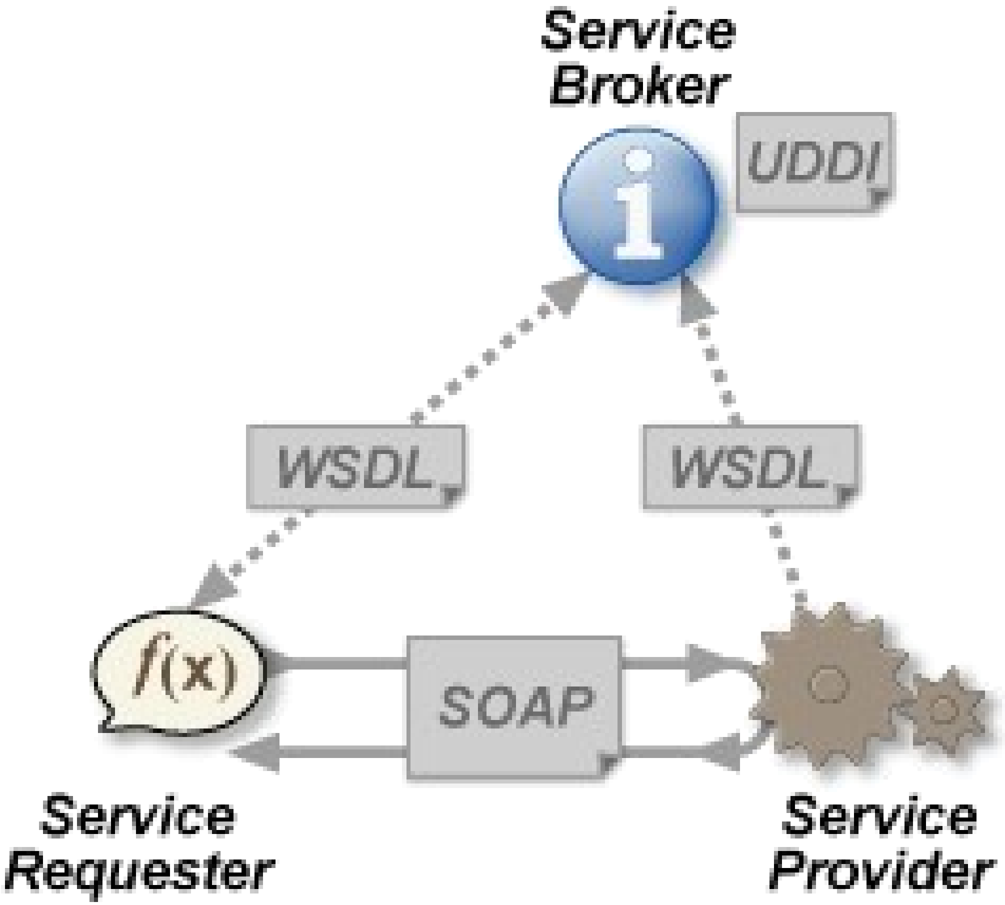
UDDI

- 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


Service Broker
UDDI
WSDL
WSDL
f(x)
SOAP
Service Requester
Service Provider

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

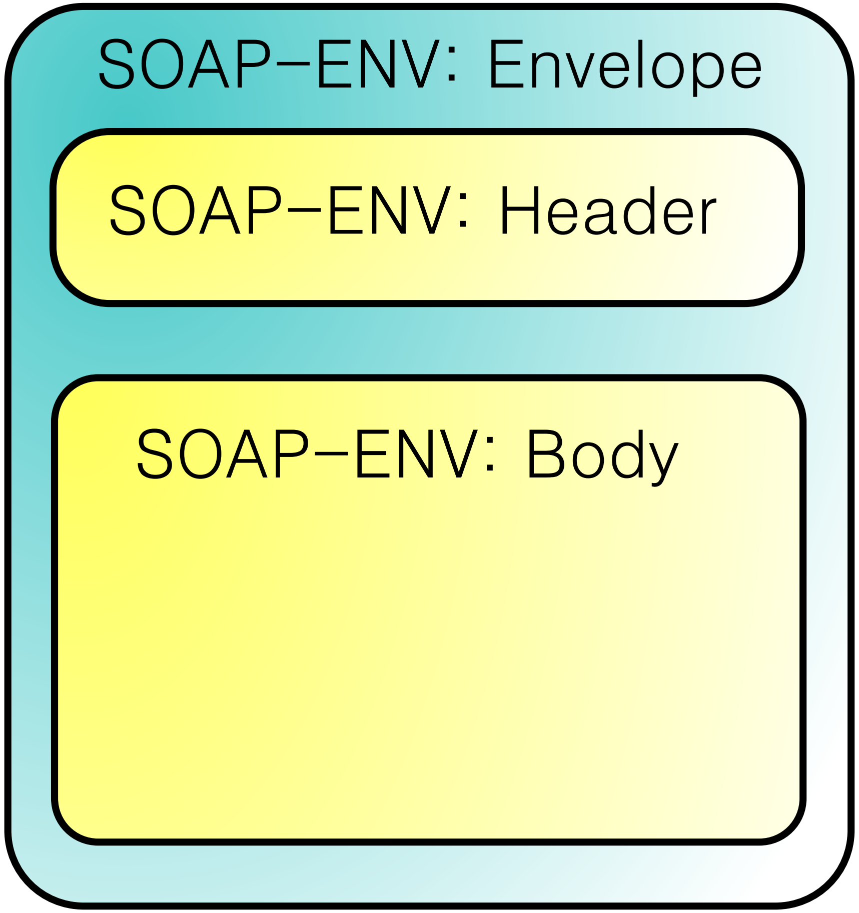
SOAP

- 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–ENV: Envelope

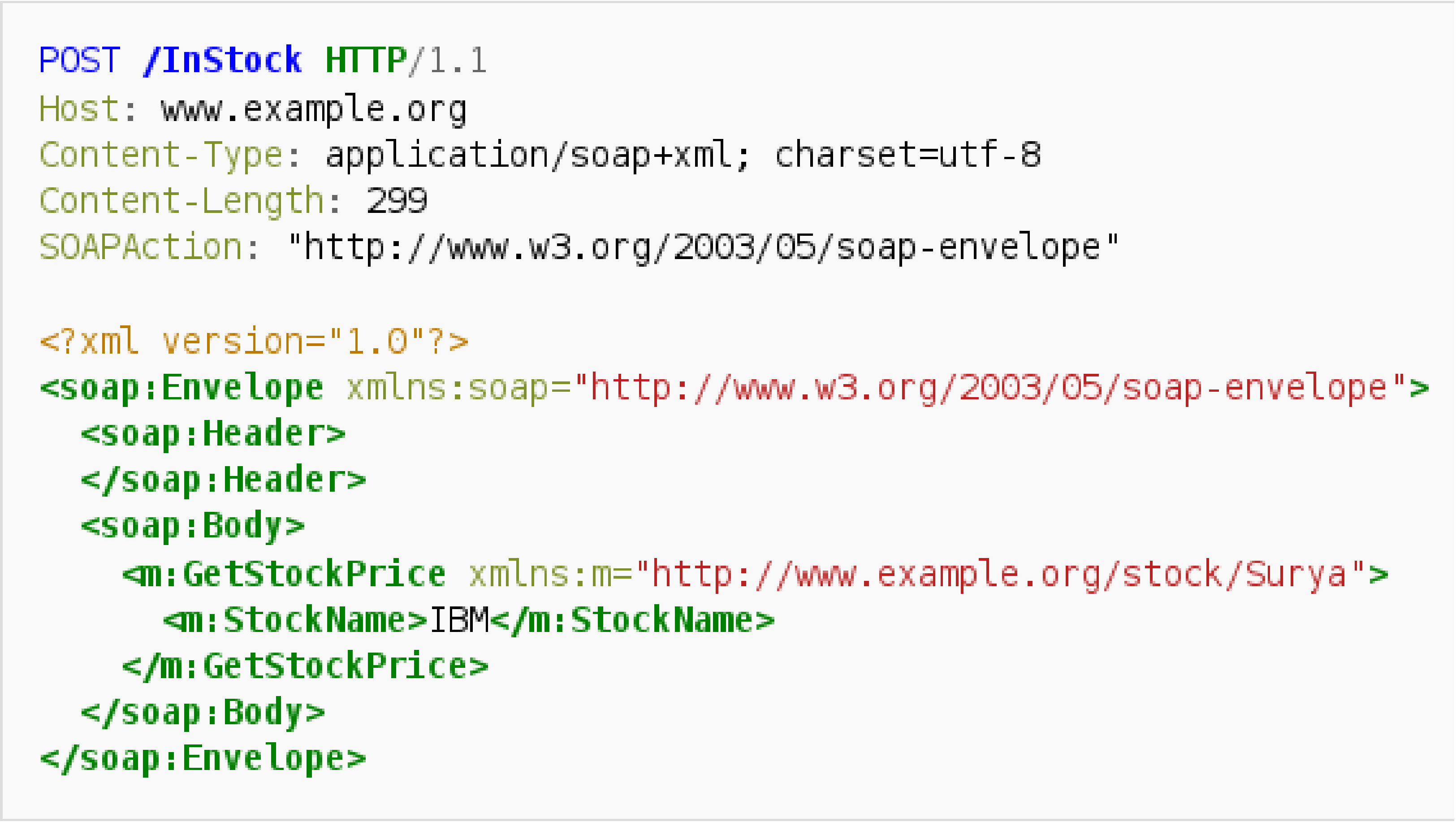
SOAP–ENV: Header

SOAP–ENV: Body

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

- API
- REST
- RESTful APIs

API

API

- Application Programming Interface
- Set of routines, protocols, and tools for building software and applications.
- An API expresses a software component in terms of its operations, inputs, outputs, and underlying types, defining functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising the interface.

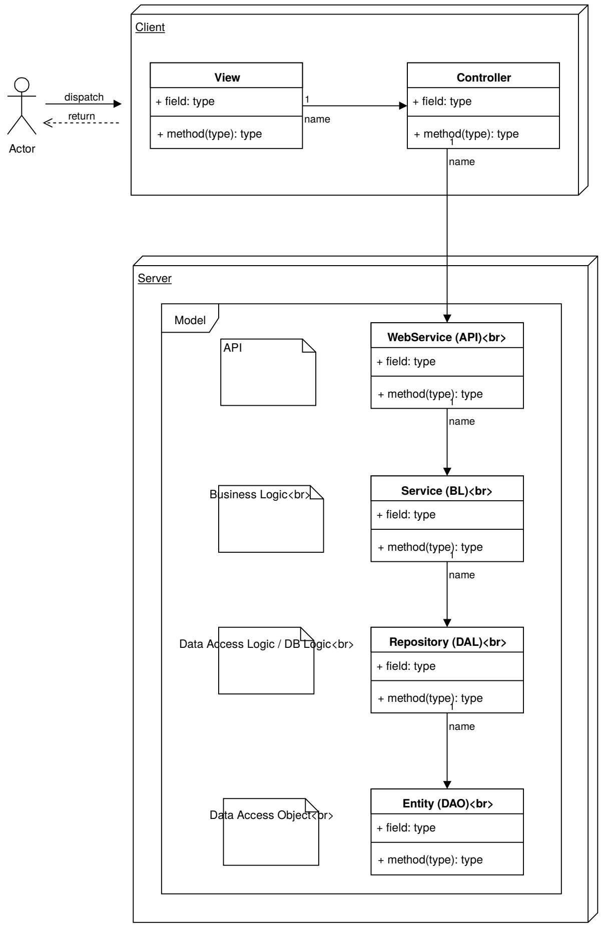
API

- In other words:
  - Contract
  - Providing functionalities grouped in services
  - By specifying a set of available operations by their signatures (names, inputs and outputs)
  - To allow communication between systems

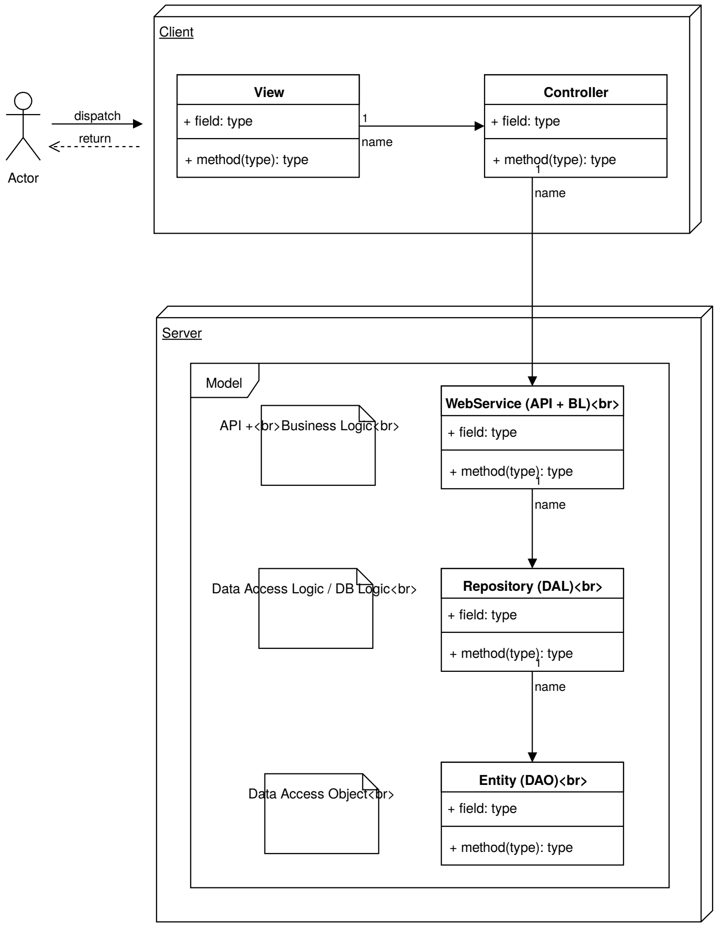
API

- Where to place an API?
  - Assuming we use a software architecture based on the following design patterns:
    - Client-Server
    - MVC
      - Note: it can also be applied to other patterns like MVP or MVVM, but as MVC is the most used one nowadays, we are going to use it as example.
  - Where the View and the Controller are placed on the Client's side.
  - We have 2 very-well-known options for our API architecture:
    - Standard
    - Simplified
      - Note: the designs below can be even more simple but we are focusing only on the API.

API


Client
Actor
dispatch
return
View
+ field: type
+ method(type): type
1
name
Controller
+ field: type
+ method(type): type
1
name
Server
Model
API
WebService (API)<br>
+ field: type
+ method(type): type
1
name
Business Logic<br>
Service (BL)<br>
+ field: type
+ method(type): type
1
name
Data Access Logic / DB Logic<br>
Repository (DAL)<br>
+ field: type
+ method(type): type
1
name
Data Access Object<br>
Entity (DAO)<br>
+ field: type
+ method(type): type

API


Client
View
+ field: type
+ method(type): type
Controller
+ field: type
+ method(type): type
1
name
1
name
Actor
dispatch
return
Server
Model
API +<br>Business Logic<br>
WebService (API + BL)<br>
+ field: type
+ method(type): type
1
name
Data Access Logic / DB Logic<br>
Repository (DAL)<br>
+ field: type
+ method(type): type
1
name
Data Access Object<br>
Entity (DAO)<br>
+ field: type
+ method(type): type

REST

REST

- Representational State Transfer
- Software architectural style of the World Wide Web
- Architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia system
- Ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements

REST

- Properties
  - Performance
  - Scalability
  - Simplicity
  - Modifiability
  - Visibility
  - Portability
  - Reliability

REST

- Constraints
  - Client-Server
  - Stateless
  - Cacheable
  - Layered system
  - Code on demand (optional)
  - Uniform interface

REST

- Uniform interface
  - Identification of resources
  - Manipulation of resources through these representations
  - Self-descriptive messages
  - Hypermedia as the engine of application state (HATEOAS)

REST

- Richardson Maturity Model
  - Way to grade your API according to the constraints of REST
  - The better your API adheres to these constraints, the higher its score is
  - 4 levels, 0-3, where level 3 designates a truly RESTful API

REST

- Level 0: Swamp of POX
  - Use its implementing protocol (normally HTTP, but it doesn't have to be) like a transport protocol.
  - Tunnel requests and responses through its protocol without using the protocol to indicate application state.
  - It will use only one entry point (URI) and one kind of method (in HTTP, this normally is the POST method).
  - Examples of these are SOAP and XML-RPC.

REST

- Level 1: Resources
  - Can distinguish between different resources.
  - This level uses multiple URIs, where every URI is the entry point to a specific resource.
  - Instead of going through http://example.org/articles, you actually distinguish between http://example.org/article/1 and http://example.org/article/2
  - Still, this level uses only one single method like POST.

REST

- Level 2: HTTP verbs
  - Your API should use the HTTP***** protocol properties in order to deal with scalability and failures
  - Don't use a single POST method for all, but make use of GET when you are requesting resources, and use the DELETE method when you want to delete a resources
  - Also, use the response codes of your application protocol. Don't use 200 (OK) code when something went wrong for instance
  - *****REST is completely protocol agnostic, so if you want to use a different protocol, your API can still be RESTful

REST

- Level 3: Hypermedia controls
  - Uses HATEOAS to deal with discovering the possibilities of your API towards the clients

REST

- REST vs RESTful
  - REST: Level 2
  - RESTful: Level 3

REST

- REST/ful API Description Languages
  - WSDL: Level 0-2
  - WADL: Level 2
  - HATEOAS: Level 3

REST

- HATEOAS
  - Hypermedia as the Engine of Application State
  - Hypertext-driven APIs
  - The client software is not written to a static interface description shared through documentation
  - Instead, the client is given a set of entry points and the API is discovered dynamically through interaction with these endpoints

RESTful APIs

RESTful APIs

- Web service APIs that adhere to the REST architectural constraints are called RESTful APIs
- HTTP-based RESTful APIs are defined with the following aspects:
  - Base URI
    - Such as http://example.com/resources/
  - An Internet media type for the data.
    - This is often JSON but can be any other valid Internet media type (e.g., XML, Atom, microformats, application/vnd.collection+json, etc.)
  - Standard HTTP methods
    - E.g., OPTIONS, GET, PUT, POST, or DELETE
  - Hypertext links to reference state
  - Hypertext links to reference-related resources

RESTful APIs

- How to start building your own RESTful API?
  - It should start right after you domain name
    - Domain: http://www.example.org
    - Following the pattern below:
      - /api/{version}/{access-level}*
        - {version}: API version
        - {access-level}: API access level
        - *Both version and access level are optional
        - **Versioning your API is highly recommended**
        - Access level is not so important and most people don't use it, but it is useful for quick access filtering and pre-securitization

RESTful APIs

- What is the correct way to version my API?
  - The URL way:
    - A commonly used way to version your API is to add a version number in the URL. For instance:
      - /api/**v1**/article/1234
    - To "move" to another API, one could increase the version number:
      - /api/**v2**/article/1234
    - The Hypermedia way:
      - GET /api/article/1234 HTTP/1.1
      - Accept: application/vnd.api.article+xml; **version=1.0**

RESTful APIs

- How to specify the access level?
  - The most common way is to specify two different values:
    - /priv vs /pub
    - /private vs /public
  - And then match those values to you application user roles
  - At this level API resource entry level, it is not worth it to be more granular by adding more values

RESTful APIs

- How do I let users log into my RESTful API?
  - One of the main differences between RESTful and other server-client communications services is that **any session state** in a RESTful setup **is held in the client**, **the server is stateless**
  - This requires the client to provide all information necessary to make the request
  - Different ways:
    - HTTP Basic Authentication
    - HMAC
    - OAuth/OAuth2
    - JWT
  - Notes:
    - Most of them **MUST** be protected through **SSL/TLS**. They should not be used over plain HTTP
    - Using nonces can improve your security, but you MUST store and compare nonces server-side

RESTful APIs

- Tips on how to build a RESTful API
  - Use HTTP Verbs to Make Your Requests Mean Something
  - Provide Sensible Resource Names
  - Use HTTP Response Codes to Indicate Status
  - Offer Both JSON and XML*
    - *Mobile Apps -> JSON, please
  - Create Fine-Grained Resources
  - Consider Connectedness (HATEOAS)

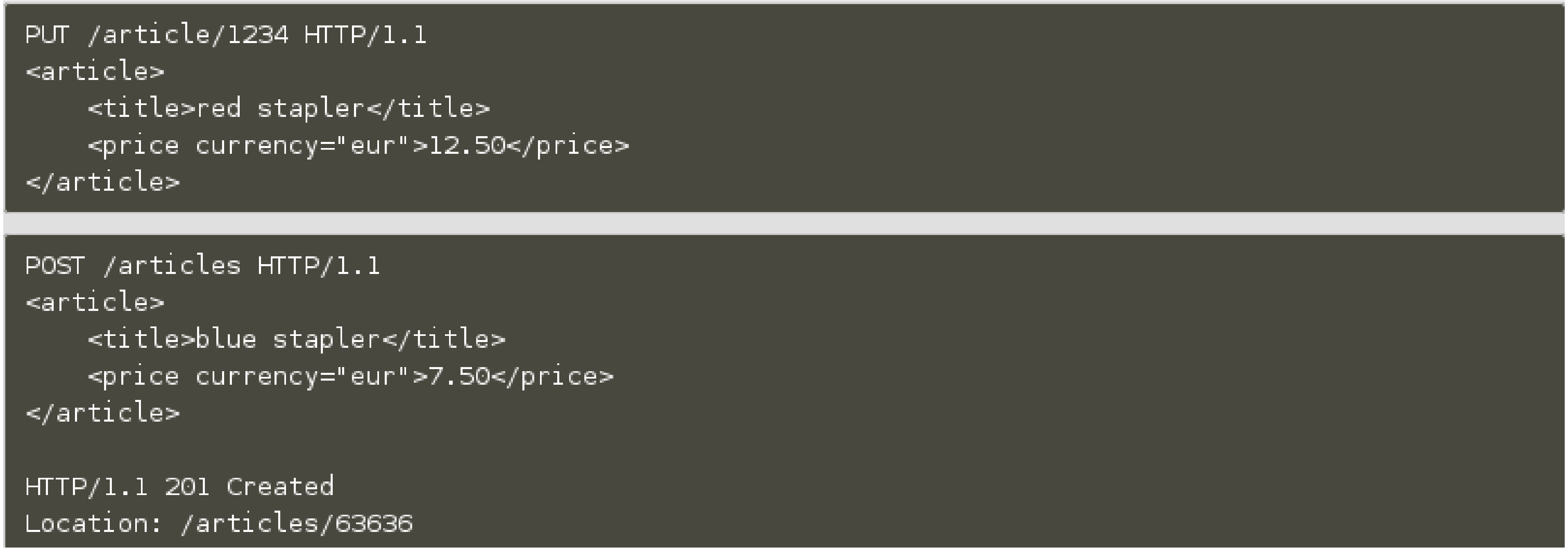
RESTful APIs

- Use HTTP Verbs to Make Your Requests Mean Something
  - The **HTTP verbs** comprise a major portion of our “uniform interface” constraint and **provide** us **the action** counterpart **to the noun-based resource**
  - The primary or most-commonly-used HTTP verbs:
    - GET (Read)
    - POST (Create)
    - PUT (Update)
    - DELETE (Delete)
  - There are a number of other verbs too, but are utilized less frequently:
    - PATCH (Partial update)
    - OPTIONS (Supported methods)
    - HEAD

RESTful APIs

- When to use PUT or POST?
  - Use PUT when you can update a resource completely through a specific resource
  - For instance, if you know that an article resides at http://example.org/article/1234, you can PUT a new resource representation of this article directly through a PUT on this URL
  - PUT and POST are both unsafe methods. However, PUT is idempotent, while POST is not
  - PUTting the same data multiple times to the same resource, should not result in different resources, while POSTing to the same resource can result in the creation of multiple resources

RESTful APIs



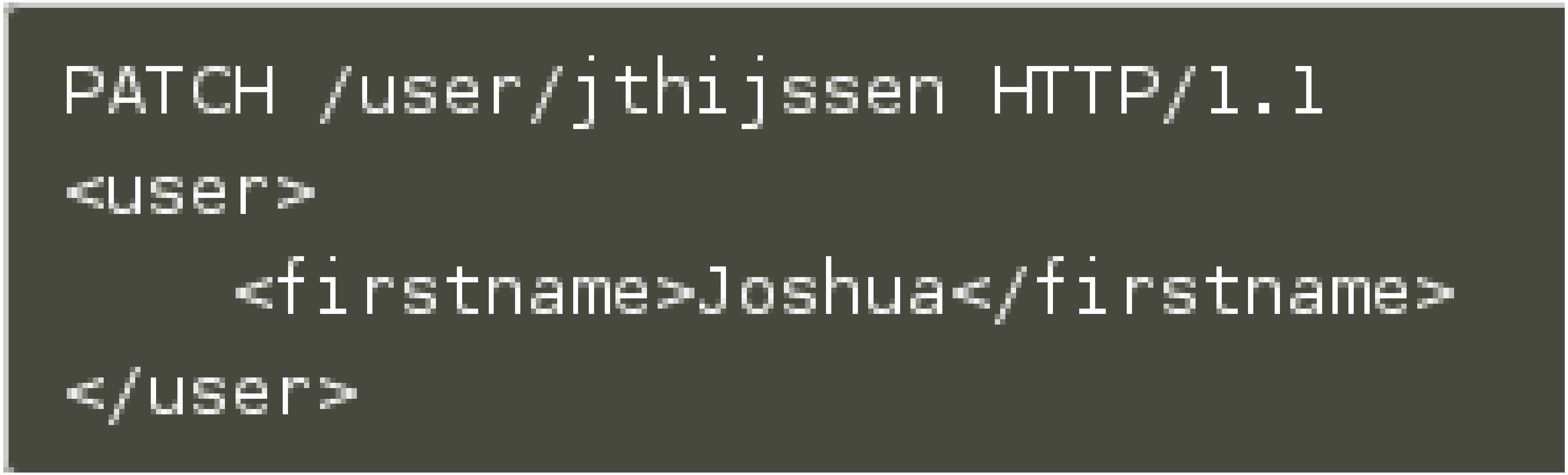
```
PUT /article/1234 HTTP/1.1
<article>
    <title>red stapler</title>
    <price currency="eur">12.50</price>
</article>
```

```
POST /articles HTTP/1.1
<article>
    <title>blue stapler</title>
    <price currency="eur">7.50</price>
</article>

HTTP/1.1 201 Created
Location: /articles/63636
```

RESTful APIs

- When to use PATCH?
  - The HTTP methods PATCH can be used to update partial resources
  - For instance, when you only need to update one field of the resource, PUTting a complete resource representation might be cumbersome and utilizes more bandwidth



```
PATCH /user/jthijssen HTTP/1.1
<user>
    <firstname>Joshua</firstname>
</user>
```

RESTful APIs

- What are idempotent and/or safe methods?
  - Safe methods are HTTP methods that do not modify resources
    - For instance, using GET or HEAD on a resource URL, should NEVER change the resource
    - However, this is not completely true
    - It means: it won't change the resource representation
    - It is still possible, that safe methods do change things on a server or resource, but this should not reflect in a different representation

RESTful APIs

- What are idempotent and/or safe methods?
  - Be careful when dealing with safe methods:
    - If a seemingly safe method like GET will change a resource, it might be possible that any middleware client proxy systems between you and the server, will cache this response
    - Another client who wants to change this resource through the same URL (like: http://example.org/api/article/1234/delete), will not call the server, but return the information directly from the cache
    - Non-safe (and non-idempotent) methods will never be cached by any middleware proxies

RESTful APIs

- What are idempotent and/or safe methods?
  - An idempotent HTTP method is a HTTP method that can be called many times without different outcomes
  - It would not matter if the method is called only once, or ten times over. The result should be the same
  - Again, this only applies to the result, not the resource itself. This still can be manipulated

RESTful APIs

- Idempotency is important in building a fault-tolerant API
  - Suppose a client wants to update a resource through POST
  - Since POST is not a idempotent method, calling it multiple times can result in wrong updates
  - What would happen if you sent out the POST request to the server, but you get a timeout. Is the resource actually updated?
  - Does the timeout happened during sending the request to the server, or the response to the client?
  - Can we safely retry again, or do we need to figure out first what has happened with the resource?
  - By using idempotent methods, we do not have to answer this question, but we can safely resend the request until we actually get a response back from the server.

RESTful APIs

HTTP Method	Idempotent	Safe
OPTIONS	Yes	Yes
GET	Yes	Yes
HEAD	Yes	Yes
PUT	Yes	No
POST	No	No
DELETE	Yes	No
PATCH	No	No

RESTful APIs

- Caching your REST API
  - The goal of caching is never having to generate the same response twice
  - The benefit of doing this is that we gain speed and reduce server load
  - The best way to cache your API is to put a gateway cache (or reverse proxy) in front of it
  - Some frameworks provide their own reverse proxies, but a very powerful, open-source one is **Varnish**

RESTful APIs

- Caching your REST API
  - When a safe method is used on a resource URL, the reverse proxy should cache the response that is returned from your API
  - It will then use this cached response to answers all subsequent requests for the same resource before they hit your API
  - When an unsafe method is used on a resource URL, the cache ignores it and passes it to the API
  - The API is responsible for making sure that the cached resource is invalidated

RESTful APIs

- Caching your REST API
  - HTTP has an unofficial PURGE method that is used for purging caches
  - When an API receives a call with an unsafe method on a resource, it should fire a PURGE request on that resource so that the reverse proxy knows that the cached resource should be expired
  - Note that you will still have to configure your reverse proxy to actually remove a resource when it receives a request with the PURGE method

RESTful APIs

- Provide Sensible Resource Names
  - Producing a great API is 80% art and 20% science
  - Creating a URL hierarchy representing sensible resources is the art part
  - Having sensible resource names (which are just URL paths, such as /customers/12345/orders) improves the clarity of what a given request does
  - Appropriate resource names provide context for a service request, increasing understandability of the API
  - Resources are viewed hierarchically via their URI names, offering consumers a friendly, easily-understood hierarchy of resources to leverage in their applications

RESTful APIs

- Provide Sensible Resource Names
  - Here are some quick-hit rules for URL path (resource name) design:
    - Use identifiers in your URLs instead of in the query-string. Using URL query-string parameters is fantastic for filtering, but not for resource names
      - Good: /users/12345
      - Poor: /api?type=user&id=23
    - Leverage the hierarchical nature of the URL to imply structure.
    - **Design for your clients, not for your data**
    - **Resource names should be nouns. Avoid verbs as resource names**, to improve clarity. **Use the HTTP methods to specify the verb** portion of the request
    - **Use plurals in URL segments to keep your API URIs consistent across all HTTP methods**, using the collection metaphor.
      - Recommended: /customers/33245/orders/8769/lineitems/1
      - Not: /customer/33245/order/8769/lineitem/1
    - Avoid using collection verbiage in URLs. For example 'customer_list' as a resource. Use pluralization to indicate the collection metaphor (e.g. customers vs. customer_list)
    - Use lower-case in URL segments, separating words with underscores ('_') or hyphens ('-'). Some servers ignore case so it's best to be clear
    - Keep URLs as short as possible, with as few segments as makes sense

RESTful APIs

- Use HTTP Response Codes to Indicate Status
  - Response status codes are part of the HTTP specification
  - There are quite a number of them to address the most common situations
  - In the spirit of having our RESTful services embrace the HTTP specification, our Web APIs should return relevant HTTP status codes
  - For example, when a resource is successfully created (e.g. from a POST request), the API should return HTTP status code 201.

RESTful APIs

- Use HTTP Response Codes to Indicate Status
  - Suggested usages for the "Top 10" HTTP Response Status Codes are as follows:
    - **200 OK**
      - General success status code. This is the most common code. Used to indicate success
    - **201 CREATED**
      - Successful creation occurred (via either POST or PUT). Set the Location header to contain a link to the newly-created resource (on POST). Response body content may or may not be present
    - **204 NO CONTENT**
      - Indicates success but nothing is in the response body, often used for DELETE and PUT operations
    - **400 BAD REQUEST**
      - General error for when fulfilling the request would cause an invalid state. Domain validation errors, missing data, etc. are some examples
    - **401 UNAUTHORIZED**
      - Error code response for missing or invalid authentication token
    - **403 FORBIDDEN**
      - Error code for when the user is not authorized to perform the operation or the resource is unavailable for some reason (e.g. time constraints, etc.)
    - **404 NOT FOUND**
      - Used when the requested resource is not found, whether it doesn't exist or if there was a 401 or 403 that, for security reasons, the service wants to mask
    - **405 METHOD NOT ALLOWED**
      - Used to indicate that the requested URL exists, but the requested HTTP method is not applicable
      - For example, POST /users/12345 where the API doesn't support creation of resources this way (with a provided ID). The Allow HTTP header must be set when returning a 405 to indicate the HTTP methods that are supported. In the previous case, the header would look like "Allow: GET, PUT, DELETE"
    - **409 CONFLICT**
      - Whenever a resource conflict would be caused by fulfilling the request. Duplicate entries, such as trying to create two customers with the same information, and deleting root objects when cascade-delete is not supported are a couple of examples
    - **500 INTERNAL SERVER ERROR**
      - Never return this intentionally. The general catch-all error when the server-side throws an exception. Use this only for errors that the consumer cannot address from their end

RESTful APIs

- When to return 4xx or 5xx codes to the client?
  - 4xx codes are used to tell the client that a fault has taken place on THEIR side
    - They should not retransmit the same request again, but fix the error first
  - 5xx codes tell the client something happened on the server and their request by itself was perfectly valid
    - The client can continue and try again with the request without modification

RESTful APIs

HTTP Verb	**CRUD**	**Entire Collection** (e.g. /customers)	**Specific Item** (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists..
GET	Read	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single customer. 404 (Not Found), if ID not found or invalid.
PUT	Update / Replace	404 (Not Found), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update / Modify	404 (Not Found), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	404 (Not Found), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

RESTful APIs

- Are REST and HTTP the same thing?
  - Nope!
  - HTTP stands for HyperText Transfer Protocol and is a way to transfer files. This protocol is used to link pages of hypertext in what we call the World Wide Web. However, there are other transfer protocols available, like FTP
  - REST, or REpresentational State Transfer, is a set of constraints that ensure a scalable, fault-tolerant and easily extendible system. The world-wide-web is an example of such system

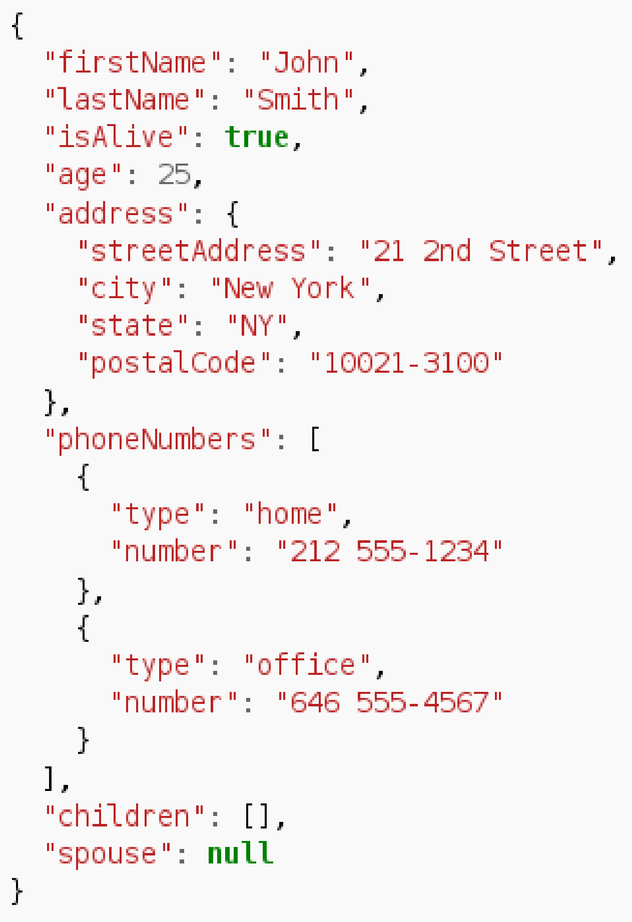
RESTful APIs

- Offer Both JSON and XML
  - Favor JSON support unless you're in a highly-standardized and regulated industry that requires XML, schema validation and namespaces, and offer both JSON and XML unless the costs are staggering
  - Ideally, let consumers switch between formats using the HTTP Accept header, or by just changing an extension from .xml to .json on the URL

RESTful APIs

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

RESTful APIs



```
{
  "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
}
```

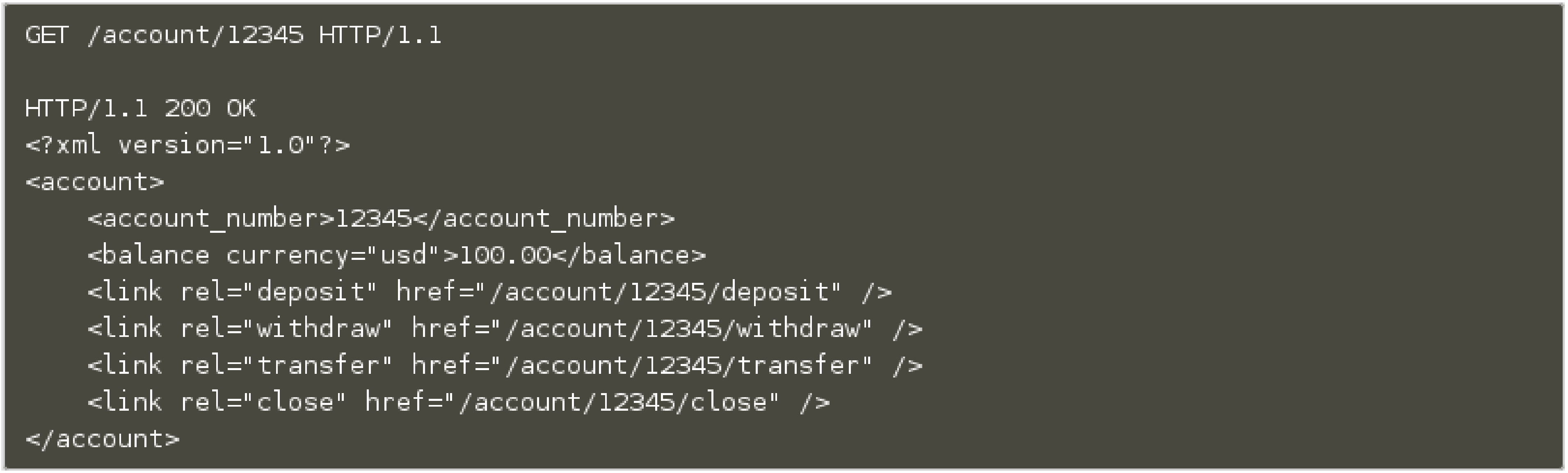
RESTful APIs

- Consider Connectedness
  - One of the principles of REST is connectedness—via hypermedia links (HATEOAS)
  - While services are still useful without them, APIs become more self-descriptive and discoverable when links are returned in the response
  - At the very least, a 'self' link reference informs clients how the data was or can be retrieved
  - Additionally, utilize the HTTP Location header to contain a link on resource creation via POST (or PUT)
  - For collections returned in a response that support pagination, 'first', 'last', 'next' and 'prev' links at a minimum are very helpful

RESTful APIs

- What is HATEOAS and why is it important for my REST API?
  - HATEOAS stands for Hypertext As The Engine Of Application State
  - It means that hypertext should be used to find your way through the API
  - Let's see an example

RESTful APIs



```
GET /account/12345 HTTP/1.1

HTTP/1.1 200 OK
<?xml version="1.0"?>
<account>
    <account_number>12345</account_number>
    <balance currency="usd">100.00</balance>
    <link rel="deposit" href="/account/12345/deposit" />
    <link rel="withdraw" href="/account/12345/withdraw" />
    <link rel="transfer" href="/account/12345/transfer" />
    <link rel="close" href="/account/12345/close" />
</account>
```

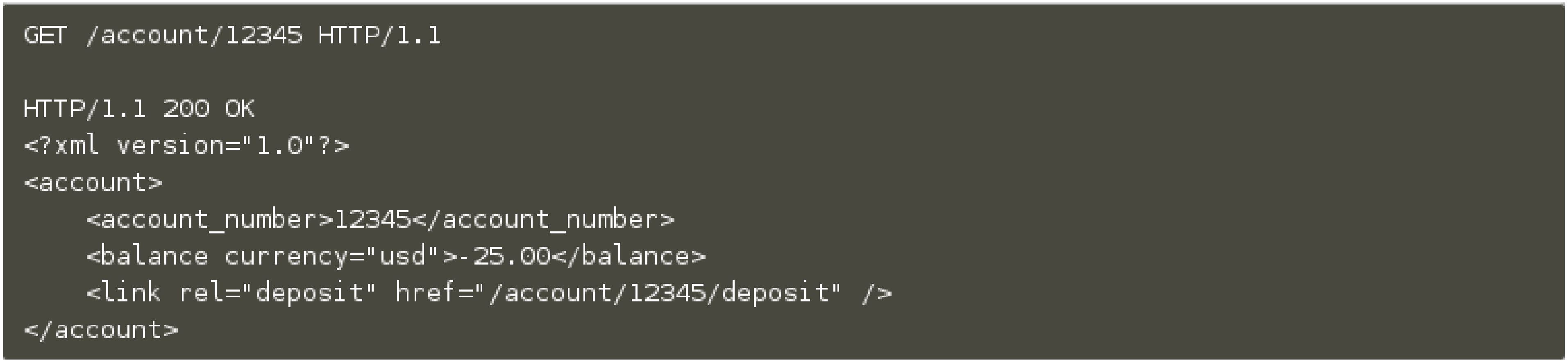
RESTful APIs

- Apart from the fact that we have 100$ in our account, we can see 4 options: deposit more money, withdraw money, transfer money to another account, or close our account.
- The "link"-tags allows us to find out the URLs that are needed for the specified actions

RESTful APIs

- Now, let's suppose we didn't have 100$ in the bank, but we actually are in the red

RESTful APIs



```
GET /account/12345 HTTP/1.1

HTTP/1.1 200 OK
<?xml version="1.0"?>
<account>
    <account_number>12345</account_number>
    <balance currency="usd">-25.00</balance>
    <link rel="deposit" href="/account/12345/deposit" />
</account>
```

RESTful APIs

- Now we are 25$ in the red
- Do you see that right now we have lost many of our options, and only depositing money is valid?
- As long as we are in the red, we cannot close our account, nor transfer or withdraw any money from the account
- The hypertext is actually telling us what is allowed and what not: HATEOAS

RESTful APIs

- Is my API RESTful when I use (only) JSON?
  - Nope!
  - One of the key constraints on REST is that a RESTful API must use hypermedia formats (HATEOAS)
  - JSON is not a hypermedia format
  - Although JSON does't have inherent hypermedia support, some standardisation is on its way to change that:
    - **JSON-LD**, which is already an official W3C standard
    - And **HAL**, which is a personal project
  - Both formalize the expression of links in JSON so that clients can instantly follow and discover these links without having to rely on out-of-band additional knowledge

Exercises

- **Exercise 4**: provide an API that fits the requirements for the following user story
  - **Story/Feature**: user resets his/her password
    - **In order to** log in
    - **As a** user
    - **I want to** reset my password
- Note: BDD, Gherkin language

Exercises

- **Scenario 1**: providing a valid user's email while resetting the password should send a unique one-shot time-limited link by email to the user
  - **Given** the user is not logged in
  - **And** is on the login page
  - **And** clicks on the reset password link
  - **When** filling the email field with a valid email
  - **And** clicking on the reset password button
  - **Then** a reset password email should be sent to the provided email address

Exercises

- **Scenario 2**: clicking on the email's reset password link should display the reset password page
  - **Given** the user is not logged in
  - **When** clicking on the email's reset password link
  - **Then** the reset password page should be displayed

Exercises

- **Scenario 3**: resetting the password should get the user logged in
  - **Given** the user is not logged in
  - **And** is on the reset password page
  - **When** introducing a valid password twice
  - **And** both passwords match
  - **Then** the password is reset
  - **And** a confirmation email is sent to the user
  - **And** the user is automatically logged in

JAX-WS

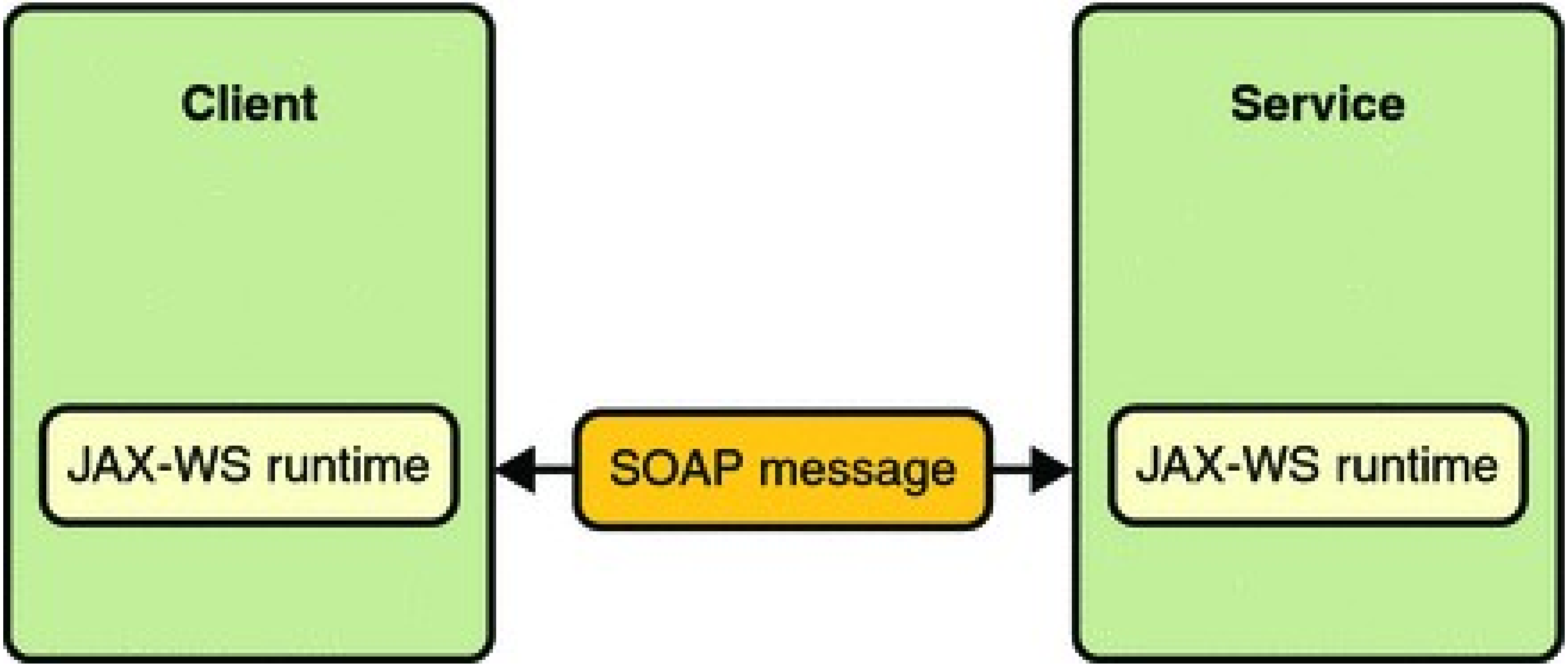
JAX-WS

- 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

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

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)

Exercises

- **Exercise 5**:
  - 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

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

Exercises

- **Exercise 3**:
  - Create a new project
  - Add the xml file from the exercice 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

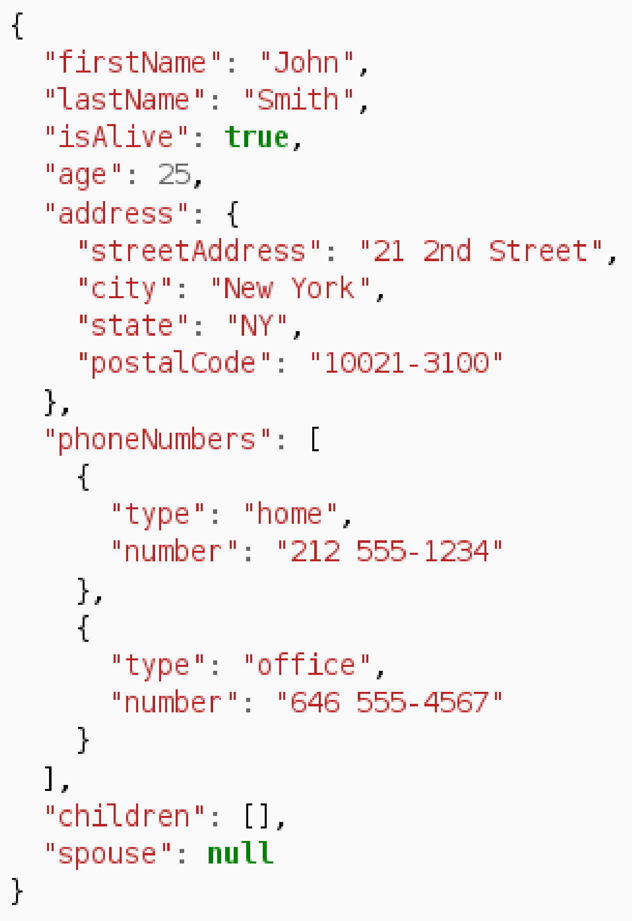
Exercises

- **Bonus exercise**:
  - Repeat Exercise 2 but using complex objects

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
}
```

Exercises

- **Exercise 6**:
  - Create your own JSON document containing several elements and tree levels
  - Validate it using the link below:
    - http://jsonlint.com/

JSON

- Example
  - Serialize & deserialize with Jackson

Exercises

- **Exercise 7**:
  - Create a new project
  - Add the json file from the exercice 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

</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 exercice 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

- **Bonus exercise**:
  - Repeat Exercise 2 but using complex objects

Exercises

- **Exercise 4**: provide an API that fits the requirements for the following user story
  - **Story/Feature**: user resets his/her password
    - **In order to** log in
    - **As a** user
    - **I want to** reset my password
- Note: BDD, Gherkin language

Exercises

- **Scenario 1**: providing a valid user's email while resetting the password should send a unique one-shot time-limited link by email to the user
  - **Given** the user is not logged in
  - **And** is on the login page
  - **And** clicks on the reset password link
  - **When** filling the email field with a valid email
  - **And** clicking on the reset password button
  - **Then** a reset password email should be sent to the provided email address

Exercises

- **Scenario 2**: clicking on the email's reset password link should display the reset password page
  - **Given** the user is not logged in
  - **When** clicking on the email's reset password link
  - **Then** the reset password page should be displayed

Exercises

- **Scenario 3**: resetting the password should get the user logged in
  - **Given** the user is not logged in
  - **And** is on the reset password page
  - **When** introducing a valid password twice
  - **And** both passwords match
  - **Then** the password is reset
  - **And** a confirmation email is sent to the user
  - **And** the user is automatically logged in

Exercises

- **Exercise 5**:
  - 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

- **Exercise 6**:
  - Create your own JSON document containing several elements and tree levels
  - Validate it using the link below:
    - http://jsonlint.com/

Exercises

- **Exercise 7**:
  - Create a new project
  - Add the json file from the exercice 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

REST

- Curl
  - Is a tool to transfer data from or to a server, using one of the supported protocols (DICT, FILE, FTP, FTPS, GOPHER, HTTP, HTTPS, IMAP, IMAPS, LDAP, LDAPS, POP3, POP3S, RTMP, RTSP, SCP, SFTP, SMTP, SMTPS, TELNET and TFTP)
  - The command is designed to work without user interaction

REST

- Curl
  - # man curl
  - # curl
    http://maps.googleapis.com/maps/api/geocode/
    json?address=%22a%20coru%C3%B1a%22&sensor=fa
    lse
  - # curl
    http://www.openstreetmap.org/api/0.6/relation/3468
    10

REST

- SoapUI
  - Is a free and open source cross-platform Functional Testing solution
  - With an easy-to-use graphical interface, and enterprise-class features, SoapUI allows you to easily and rapidly create and execute automated functional, regression, compliance, and load tests
  - In a single test environment, SoapUI provides complete test coverage and supports all the standard protocols and technologies

REST

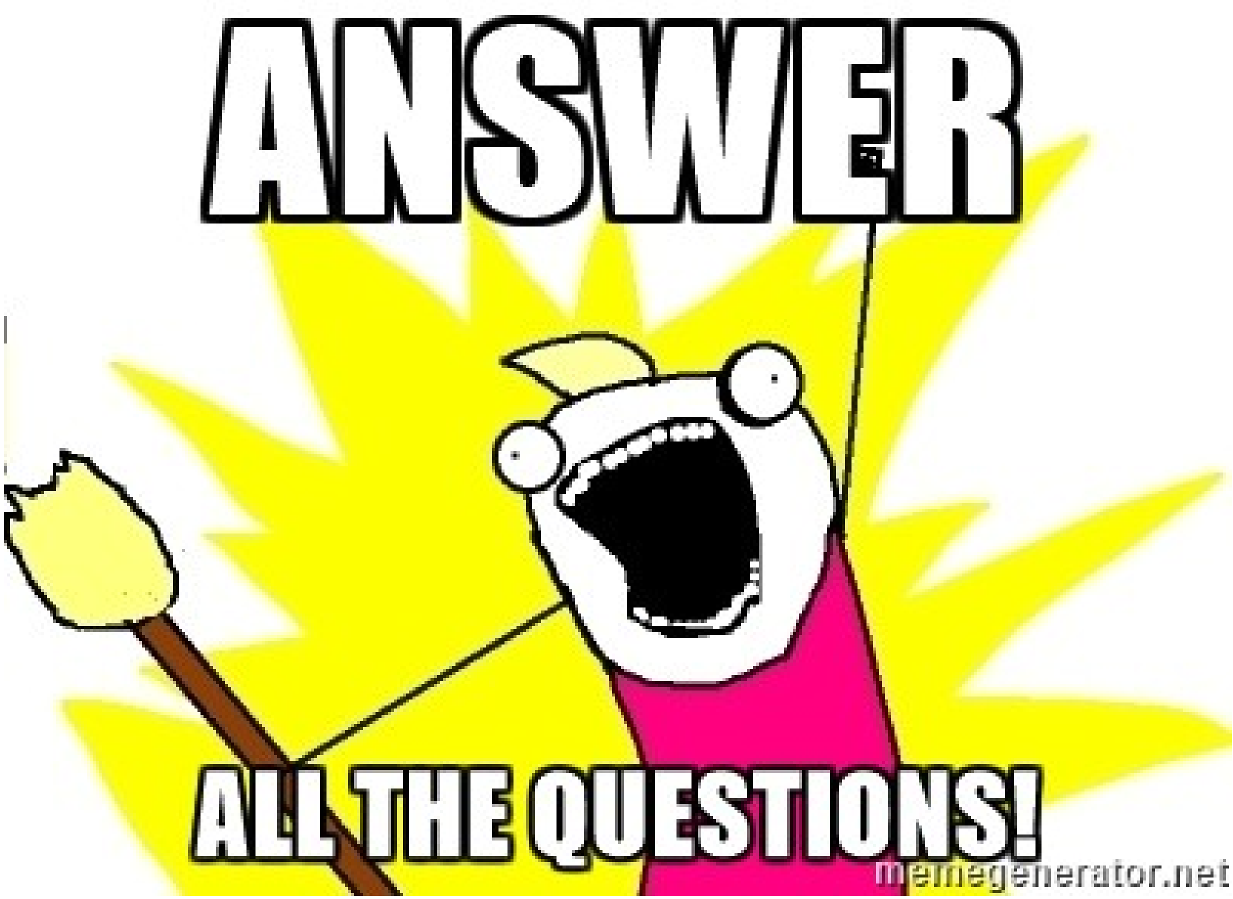
- SoapUI
  - New REST project

</practice>

'Quiz' Pro Quo

- Exam:
  - TBD
- Survey:
  - TBD

'Quiz' Pro Quo



</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
    - Service-Oriented Architecture (SOA)
      - https://en.wikipedia.org/wiki/Service-oriented_architecture
    - 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
    - Publish/subscribe 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
    - Software design anti-patterns
      - 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
    - https://en.wikipedia.org/wiki/SOAP
  - WSDL
    - http://www.w3schools.com/xml/xml_wsdl.asp
    - https://en.wikipedia.org/wiki/Web_Services_Description_Language
  - UDDI
    - 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
    - https://en.wikipedia.org/wiki/SOAP
  - REST
    - https://github.com/alejopj/restful-apis

Bibliography

- API
  - Wikipedia
    - https://en.wikipedia.org/wiki/Application_programming_interface

Bibliography

- REST
  - How I Explained REST to Non-Tech People
    - http://web.archive.org/web/20130116005443/http://tomayko.com/writings/rest-to-my-wife
  - Wikipedia - REST
    - https://en.wikipedia.org/wiki/Representational_state_transfer
  - REST Cook Book
    - http://restcookbook.com/

Bibliography

- RESTful APIs
  - Wikipedia - REST
    - https://en.wikipedia.org/wiki/Representational_state_transfer
  - REST Cook Book
    - http://restcookbook.com/
  - REST API Tutorial
    - http://www.restapitutorial.com/
  - Wikipedia - RESTful API Description Languages
    - https://en.wikipedia.org/wiki/Overview_of_RESTful_API_Description_Languages
  - Wikipedia – JSON
    - https://en.wikipedia.org/wiki/JSON

Bibliography

- Exercises
  - BDD
    - https://en.wikipedia.org/wiki/Behavior-driven_development
  - Gherkin language
    - https://en.wikipedia.org/wiki/Cucumber_%28software%29
  - GPUL IoT Hackaton Ideas:
    - https://twitter.com/gpul_/status/705140161907138569
  - GPUL </Lab> about Django Framework;
  - https://github.com/gpul-labs/roadmap#semana4

Bibliography

- JAX-WS
  - Wikipedia
    - 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_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/
    - http://www.mkyong.com/webservices/jax-rs/json-example-with-jersey-jackson/
    - http://www.mkyong.com/webservices/jax-rs/restful-java-client-with-jersey-client/
  - Baeldung
    - http://www.baeldung.com/jackson-annotations
  - StackOverflow
    - http://stackoverflow.com/questions/3280979/running-jetty-from-eclipse
  - How to do in Java
    - http://howtodoinjava.com/jersey/jersey-restful-client-examples/

</bibliography>