# XML, JSON & NoSQL Databases

Gilles Degols
based on the initial work of Ken Hasselmann

# Course organization

- XML, JSON
  - Theory and exercises
- NoSQL Databases
  - Theory
  - Developing a small python application iteratively through exercises
- Not everything will be written in the slides
  - If you do not come to the class, you will miss some information needed for the evaluations
  - Slides can be updated at any time, as well as the project/exercises deliverables (communicated orally)
  - https://github.com/gilles-degols/ecam-nosql
- Deliverables: must be in English

# Evaluation - Exercises

- XML, JSON (10%)
  - Submit exercises the next day of each related course (23:59)
- NoSQL Databases (20%)
  - Submit exercises the next day of each related course (23:59)
- Submit: g3d@ecam.be with "Exercises: {XML/JSON/NoSQL}" as title
  - "{lastname} {firstname}.zip"
    - Only .zip
    - No sharepoint
  - Late submission: 0/20 to the related evaluation

# Evaluation - Project

- Project (70%)
  - 3-people teams unless exception
  - Design, implementation & setup of the database in a docker-compose.yml + application
  - Presentation (.pdf) and code (app + database setup) must be sent ***the day before*** the evaluation at 23:59 the latest
  - Last course: 20 minutes presentation + 20 minutes Q/A
    - Time allocation is free to change if deemed necessary by the lecturer
  - Everyone will listen to every presentation
  - Different notes can be given depending on the contribution & comprehension of each student
- Submit: g3d@ecam.be with "Project - Team {i}" + .zip
  - Late submission: 0/20 to the related evaluation

# Evaluation - Project

| Database & Implementation justification | Feature implementation | Rating mark |
|---|---|---|
| Yes | Yes (full scope) | [14; 20] |
| Yes | No (full scope not done) | [0; 14[ |
| No | Yes | 0 |
| No | No | [0; 14[ |

Sending the code is part of "feature implementation"
(no code, no feature)

# About the lecturer

- Software Engineer / Big Data → Data Engineer
- Teaching Assistant at Université Libre de Bruxelles
- Companies I worked for
  - Université Libre de Bruxelles
  - Macq
  - ADB Safegate
  - Evonik
  - Proximus
  - Engie GEMS
- Course content
  - Directly related to day-to-day work

# Intro to XML

Why?
How to use it ?
How to validate it ?

# XML: Why ?

- Extensible Markup Language
- Markup language
- File format

- Goals
  - Simplicity
  - Generality
  - Usability
- To communicate data in a structured format (!= HTML)

# XML: Why ?

- SGML: Standard Generalized Markup Language
  - Released in 1986
  - Enable sharing of machine-readable documents, for several decades
- HTML is a variant of SGML
  - Pre-defined tags
  - Presentation layer
- XML is a variant of SGML
  - Data layer

# XML: Why ?

- Define your logging (log4j)
-
```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration debug="true" xmlns:log4j='http://jakarta.apache.org/log4j/'>
   <appender name="console" class="org.apache.log4j.ConsoleAppender">
      <layout class="org.apache.log4j.PatternLayout">
           <param name="ConversionPattern" value="%d{yyyy-MM-dd HH:mm:ss}" />
      </layout>
   </appender>
   <root>
      <level value="DEBUG" />
      <appender-ref ref="console" />
   </root>
</log4j:configuration>
```

# XML: Why ?

- Define your build settings (maven)

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>1.2.17</version>
  </dependency>
</project>
```

- And a multitude of other use cases across a lot of applications, languages, build tools, ...,
- Also extensively used to transfer data

# XML: Why ?

- Send & Receive data from an API (SOAP)
  - Envelope: identifies the XML document as SOAP message
  - Header: header information (authentication, ...)
  - Body: call & response
  - Fault: errors & status

# XML: Why ?

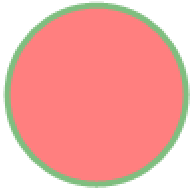
- SOAP

  -
    ```
    <?xml version="1.0"?>

    <soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

      <soap:Header>
      ...
      </soap:Header>

      <soap:Body>
      ...
        <soap:Fault>
        ...
        </soap:Fault>
      </soap:Body>

    </soap:Envelope>
    ```

# XML: Why ?

- SVG – Scalable Vector Graphics

```
<svg height="100" width="100" xmlns="http://www.w3.org/2000/svg">
   <circle r="45" cx="50" cy="50" stroke="green" stroke-width="3"
     fill="red" opacity="0.5" />
</svg>
```



# XML: How to use it ?

- Standard XML syntax, v1.0 (5th edition)
  - Released in 1998
  - Public format: https://w3.org/TR/xml
- Most languages have an XML library
- Structure definition (and validation)
  - DTD
  - XML Schema (XSD)

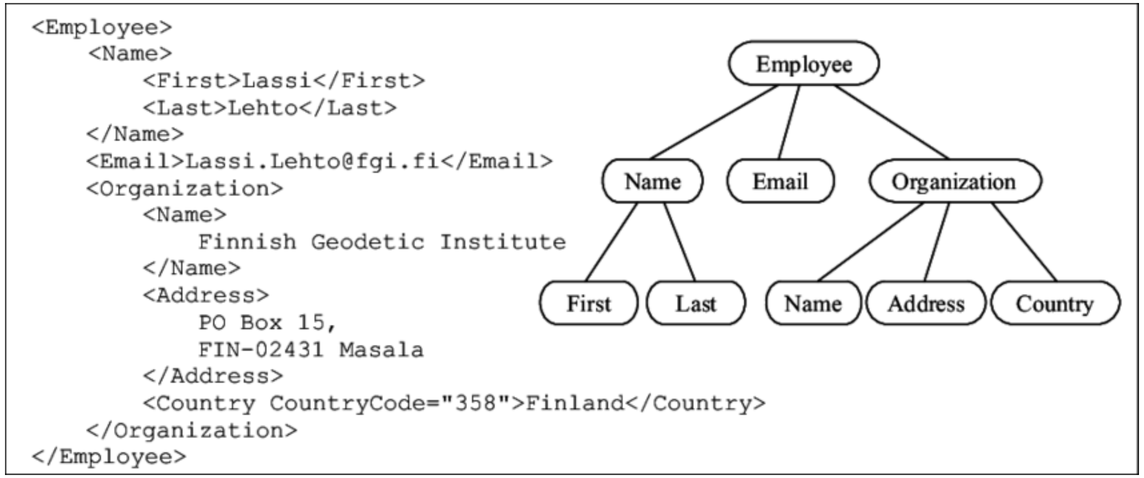
# XML: Some properties

- An XML document is *well-formed* if it follows the syntax rules of XML
- An XML document is *valid* if it is *well-formed* and follows the structured defined in a Document Type Definition (DTD) or XML Schema (XSD)
- An XML document does not contain any information on how it should be rendered

# XML: Declaration

- `<?xml version="1.0" encoding="UTF-8" ?>`
- Basic information for the XML parser:
  - XML version
  - Character encoding (optional) - most of the time, UTF-8
- But, how would you read the encoding of the first line without knowing the encoding ?

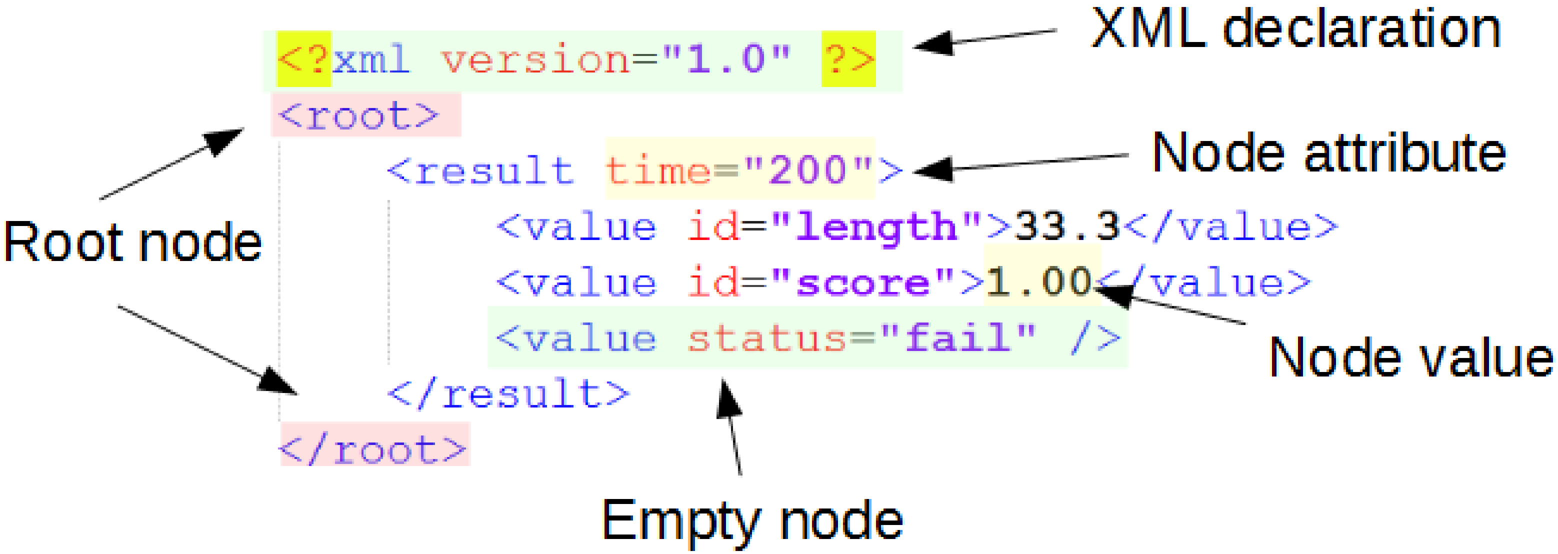
# XML: Structure



```
<Employee>
    <Name>
        <First>Lassi</First>
        <Last>Lehto</Last>
    </Name>
    <Email>Lassi.Lehto@fgi.fi</Email>
    <Organization>
        <Name>
            Finnish Geodetic Institute
        </Name>
        <Address>
            PO Box 15,
            FIN-02431 Masala
        </Address>
        <Country CountryCode="358">Finland</Country>
    </Organization>
</Employee>
```

- Root: `Employee`
- Nodes: `Name, Email, …`
- Attributes: `CountryCode=358`

# XML: Structure



```
<?xml version="1.0" ?>
<root>
    <result time="200">
        <value id="length">33.3</value>
        <value id="score">1.00</value>
        <value status="fail" />
    </result>
</root>
```

XML declaration

Node attribute

Root node

Node value

Empty node

# XML: Structure

- All elements start with a start tag and end with an end tag
- The name of the element is formed using
  - Alphanumeric characters a-zA-Z0-9
  - Underscore, dash, dot
  - Colons (:) are possible but they define a namespace
  - No space
  - Does not start with a number
  - Does not start with "xml"

# XML: Namespace

- Within an XML Schema, you might want to re-use some tags
- Namespaces
  - `log4j:configuration`
  - `soap:body`
- You must define them
  - For html code: `xmlns="http://www.w3.org/TR/html4/"`
  - `xmlns:log4j="http://jakarta.apache.org/log4j/ "`
  - `xmlns:soap="http://www.w3.org/2003/05/soap-envelope"`
- Default namespace
  - Avoid always putting the namespace as prefix

# XML: Elements

- Start and end tag must correspond
- No crossings: `<intro>…<title>…</intro>…</title>`
- Case sensitive: `<Title>` and `<title>` are different tags
- Only one root element
  - At the top of the document
  - Cannot appear again elsewhere in the tree
- XML comments: `<!-- comment -->`

# XML: Elements

- Elements can be:
  - Non empty: start with opening tag and end with closing tag, can contain text and other elements
    - `<title>The lord of the rings<title>`
  - Empty: do not contain text nor other elements
    - `<title></title>` or `<title />`
- Elements can have attributes:
  - `<title type="fantasy">The lord of the rings<title>`
  - Attributes should be defined between quotes (') or double quotes (")

# XML: Elements

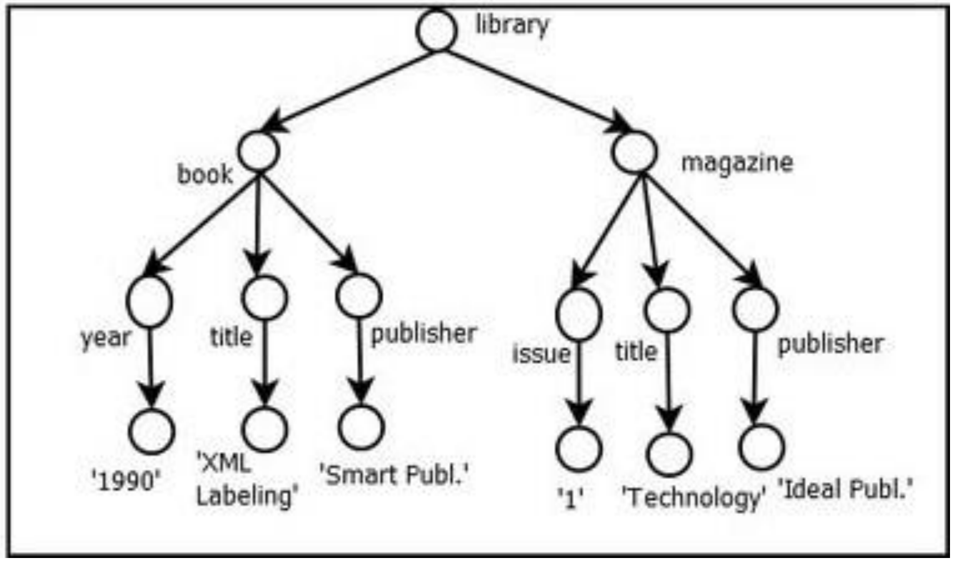
```
<parent>
  <sibling1> ... </sibling1>
  <sibling2> ... </sibling2>
  <self>
    <child1>
      ... <desc1></desc1> ... <desc2></desc2> ...
    </child1>
    <child2> ... </child2>
    <child3>
      ... <desc3><desc4> ... </desc4></desc3> ...
    </child3>
  </self>
  <sibling3> ... </sibling3>
</parent>
```

# XML: Related features

- XLink
  - Link to other xml documents, like "href" in html
- XPath
  - `/bookstore/book[1]/title`
  - `/bookstore/book[price>35]/price`
- XQuery
  - The SQL for your XMLs
  - 
    ```
    for $x in doc("books.xml")/bookstore/book
    where $x/price>30
    return $x/title
    ```
- …

# XML - Exercise 1

- Transform the following XML tree into a valid XML file (by hand)



# XML - Exercise 2

- Transform the following recipe you received from a friend into an XML file (by hand)
  - The XML is going to be used by a website to show all ingredients (wherever they appear), total execution time, necessary tooling …
  - Make sure the generated xml is consistent and easy to process by a software
- Recipe for Japanese Curry
- Ingredients
  - Beef, chopped: 450g
  - Onions, minced: 350g
  - Carrot, chopped: 100g
  - Potato, chopped: 150g
  - Water: 500ml
  - Golden Curry Sauce Mix: 92g

- Directions
  - Stir-fry meat and vegetables with oil in a large skillet on medium heat for approx. 5 min.
  - Add water and bring to boil. Reduce heat, cover and simmer until ingredients are tender, approx. 15min.
  - Turn the heat off, break S&B Golden Curry Sauce Mix into pieces and add them to the skillet. Stir until sauce mixes are completely melted. Simmer approx. 5 min., stirring constantly.
  - Serve hot over rice or noodles.

# DTD: What is it?

- Defines structural constraints in XML
- The Document Type Definition (DTD) defines the elements and their rules
- A document - with a related DTD - is valid if:
  - It is well-formed
  - It references a DTD
  - It complies with the DTD

# External DTD

- The DTD can be included directly in the document, or in an external file
- External DTD

  -
    ```
    <!DOCTYPE root_element SYSTEM|PUBLIC [name] DTD_uri>
    ```

  -
    ```
    <!DOCTYPE people_list SYSTEM "example.dtd">
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
    ```
- The DOCTYPE allows to declare the type of the document, the identifier for the root element is needed
  - SYSTEM : is local to computer, PUBLIC: can be retrieved from a catalog

# Internal DTD

- The DTD can be directly included in the document file

  -
    ```
    <!DOCTYPE people_list [
      ...
    ]>
    ```

# DTD: Example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE people_list [
  <!ELEMENT people_list (person*)>
  <!ELEMENT person (name, birthdate?, gender?,
socialsecuritynumber?)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT birthdate (#PCDATA)>
  <!ELEMENT gender (#PCDATA)>
  <!ELEMENT socialsecuritynumber (#PCDATA)>
]>
<people_list>
  <person>
    <name>Fred Bloggs</name>
    <birthdate>2008-11-27</birthdate>
    <gender>Male</gender>
  </person>
</people_list>
```

# DTD: Issues

- A DTD can be used to create a denial-of-service attack by defining nested entities expanding exponentially, or by sending the XML parser to an external resource that never returns
- Many frameworks & software (Microsoft Office) will not open files containing DTD declarations
- Other issues
  - It does not use an XML syntax
  - No typing of content
  - No regex matching
- → Replaced by XML Schema

# XML Schema: Overview

- Describe the structure of an XML document
- XML Document

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

# XML Schema: Overview

- DTD Rules

```
<!ELEMENT note (to, from, heading, body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

# XML Schema: Overview

- XSD

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="https://www.w3schools.com"
xmlns="https://www.w3schools.com"
elementFormDefault="qualified">

<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

# XML Schema - Benefits

- Introduce data types
- Use XML
  - Same language
  - Same parser
  - Same editor
- Extensible
  - Re-use a Schema in other Schemas
  - Create your own data type
  - Use multiple schemas in the same document

# XML Schema: Another example

- XML

```
<?xml version="1.0"?>
<Book>
    <Title>XML Schema Essentials</Title>
    <Authors>
        <Author>R. Allen Wyke</Author>
        <Author>Andrew Watt</Author>
    </Authors>
    <Publisher>John Wiley</Publisher>
</Book>
```

# XML Schema: Another example

- XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:annotation>
        <xsd:documentation>
            This is a sample XML Schema for Chapter 1 of XML Schema
            Essentials.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:element name="Book">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="Title"/>
                <xsd:element ref="Authors"/>
                <xsd:element ref="Publisher"/>
            </xsd:sequence>
            <xsd:attribute name="pubCountry" type="xsd:string"/>
        </xsd:complexType>
    </xsd:element>
```

…

# XML Schema: Another example

```
…
    <xsd:element name="Title" type="xsd:string"/>
    <xsd:element name="Authors">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="Author" minOccurs="1"
maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="Author" type="xsd:string"/>
    <xsd:element name="Publisher" type="xsd:string"/>
</xsd:schema>
```

# XML Schema - A few keywords

- Tags
  - `element`
  - `complexType`
  - `sequence`
  - `attribute`
- Attributes
  - `type`
  - `name`
  - `maxOccurs`
  - `minOccurs`
  - `ref`

# XML - Exercise 3

- Create a XSD to validate the following XML

```
<?xml version="1.0" encoding="UTF-8"?>
<Race date="2020-12-15" name="Holiday Meet">
    <Course>
        <CourseName>The track</CourseName>
        <Address>Track road 123</Address>
    </Course>
    <Horses>
        <Horse name="Bonfire">
            <Value>5000</Value>
            <Birthdate>1998-05-01</Birthdate>
            <Gender>M</Gender>
        </Horse>
        <Horse name="Dobby">
            <Value>1000</Value>
            <Birthdate>2001-04-05</Birthdate>
            <Gender>F</Gender>
        </Horse>
    </Horses>
</Race>
```

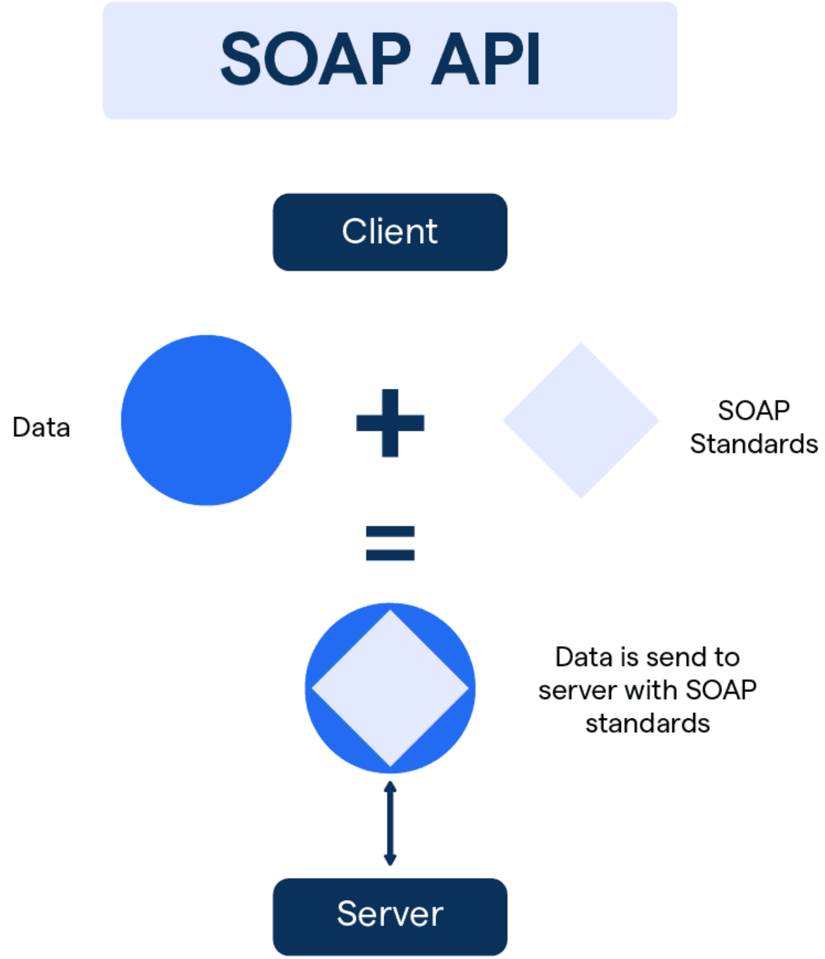
# XML - Exercise 4

- Use Python and the library lxml to load the xml of any given file, and list the content of any given xpath
- Example:
  - python ex4.py myfile.xml /Race/Horses/Horse

  -
    ```
    b'<Horse name="Bonfire">\n\t\t\t<Value>5000</Value>\n\t\t\t<Birthdate>1998-05-01</Birthdate>\n\t\t\t<Gender>M</Gender>\n\t\t</Horse>\n\t\t\n'
    ```

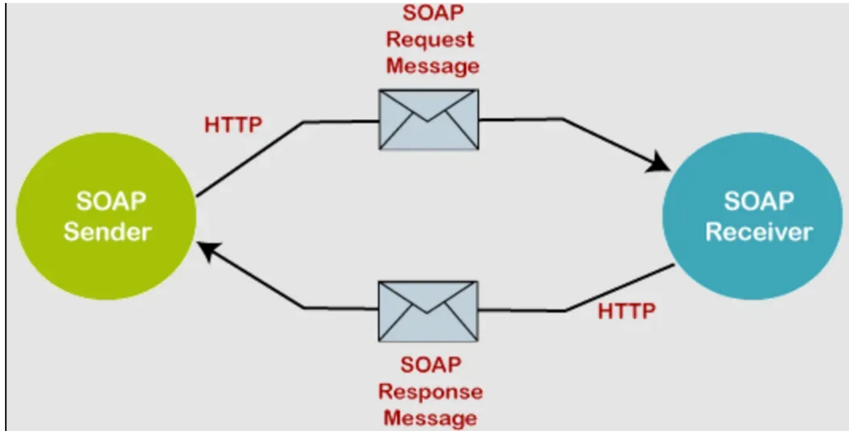
  -
    ```
    b'<Horse name="Dobby">\n\t\t\t<Value>1000</Value>\n\t\t\t<Birthdate>2001-04-05</Birthdate>\n\t\t\t<Gender>F</Gender>\n\t\t</Horse>\n\t\n'
    ```

# XML & HTTP: SOAP



# SOAP

- Simple Object Access Protocol
- Enveloppe
  - Root element with XML namespaces
- Header
  - Optional
  - Authentication tokens, encryption details, custom headers, …
- Body
  - Payload itself
- Fault
  - Error codes, error messages



# SOAP - Request

```xml
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="https://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetUser>
      <m:UserId>123</m:UserId>
    </m:GetUser>
  </soap:Body>
</soap:Envelope>
```

# SOAP - Response

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="https://www.w3.org/2003/05/soap-envelope/"
               soap:encodingStyle="https://www.w3.org/2003/05/soap-encoding">
    <soap:Body>
        <m:GetUserResponse>
            <m:Username>Tony Stark</m:Username>
        </m:GetUserResponse>
    </soap:Body>
</soap:Envelope>
```

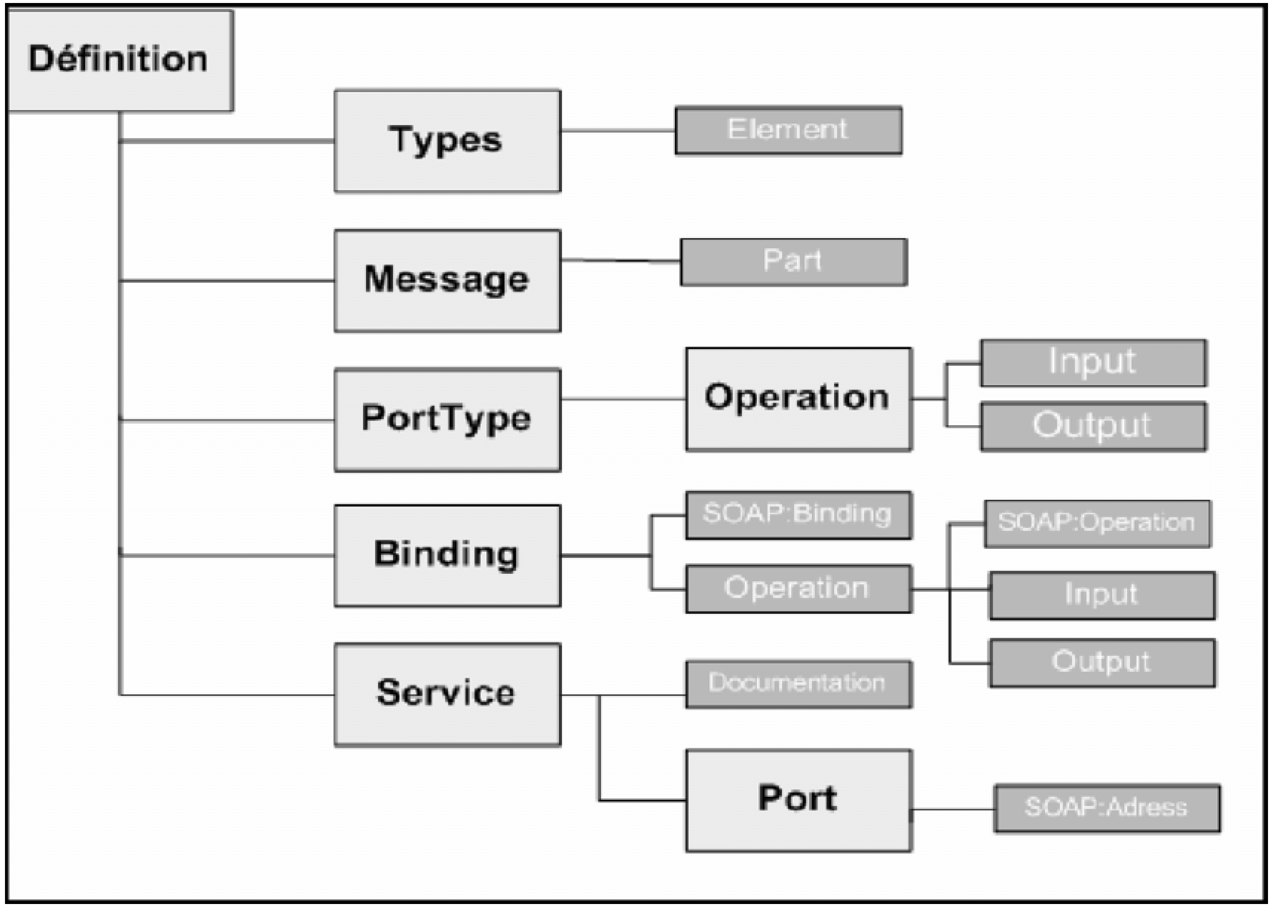
# SOAP API

- XSD
  - Describe the structure of the data types being exchanged
  - Describe the fields and restrictions on fields (max length, …)
- WSDL - Web Services Description Language
  - Describe the API and its operations
  - List of methods, parameters and returned values
  - Abstract definitions of endpoints and messages separated from the network deployment or data format bindings

# SOAP API - WSDL

- Definitions
  - `targetNamespace`
  - `xmlns`: default namespace of the WSDL document
  - `xmlns:tns` current namespace
- Types
  - Contains various xsd
- Message
- Operation
- portType
- Binding
- Port
- service

# SOAP API - WSDL



# SOAP API

- Demo

# XML - Exercise 5

- Use Python to create a SOAP API providing the various features
  - In memory "database"
  - (Shop) objects
    - Attributes: name, remaining quantity, price
    - List, create, update & delete
  - Orders
    - Attributes: object_id, customer_id, quantity
    - List, create, update & delete
  - Apply some basic validations
    - quantity >= 0, name must be of length [4;100], birth date is a real date …
- Provide a python script to test each endpoint

# XML - Credits & references

- Ken Hasselmann
  - Introduction au XML: https://brunomartin.be/cours/xml.pdf
  - Working with XML trees: https://docs.fab-image.com/studio/programming_tips/UsingXml.html
  - XML documentation: https://www.w3.org/XML/
- Official XML Schema tutorial from w3schools
  - https://www.w3schools.com/xml/schema_intro.asp
- XML Schema Essentials
  - https://nuleren.be/ebooks/xml-schema-essentials.pdf
- Japanese curry recipe
  - https://www.sbfoods-worldwide.com/recipes/010.html

# Evaluation - Exercises - Update

- ECTS description was updated by ECAM with 30% allocated to continuous evaluation
- XML, JSON (10%)
  - Submit exercises the next day of each related course (23:59)
- NoSQL Databases (20%)
  - Submit exercises the next day of each related course (23:59)
- Submit: g3d@ecam.be with "Exercises: Course {i}" as title
  - "{lastname} {firstname}.zip"
    - Only .zip
    - No sharepoint
  - Late submission: 0/20 to the related evaluation

# Evaluation - Project

- Project (70%)
  - 3-people teams unless exception
  - Design, implementation & setup of the database in a docker-compose.yml + application
  - Presentation (.pdf) and code (app + database setup) must be sent ***the day before*** the evaluation at 23:59 the latest
  - Last course: 20 minutes presentation + 20 minutes Q/A
    - Time allocation is free to change if deemed necessary by the lecturer
  - Everyone will listen to every presentation
  - Different notes can be given depending on the contribution & comprehension of each student
- Submit: g3d@ecam.be with "Project - Team {i}" + .zip
  - Late submission: 0/20 to the related evaluation

# Intro to JSON

Why?
How to use it ?
How to validate it ?

# JSON: What ?

- **J**ava**S**cript **O**bject **N**otation
- Text format to store and transport data
- Self-describing and easy to read

-
  ```
  {
    "property": "value",
    "hello \" $'": null,
    "some-key": [
      {
        "id": 42,
        "is_valid": true,
        "precision": 42.23
      }
    ]
  }
  ```

# JSON: Syntax

- Syntactically similar to creating JavaScript objects
  - JSON.parse(), JSON.stringify()
- Syntactically similar to creating Python objects
  - json.loads(), json.dumps()
- Syntax rules
  - Data is in key/value pairs
  - Data is separated by commas (careful about an extra ",")
  - Curly braces hold objects
  - Square brackets hold arrays

# JSON: Syntax

- One single root object
- Supported data types
  - String
  - Number
  - Object
  - Array
  - Boolean
  - Null
- `null`: a valid json (case sensitive!)
  - But also: `"some-string", 40.0, {}, [], true`

# JSON: Accessing data

- Python

  -
    ```
    import json
    data = json.loads("""{"property": "value","some-key": [{"id":
    42,"is_valid": true}]}""")
    print(data["property"])
    print(data["some-key"][0]["id"])
    ```

  -
    ```
    data = {"property": "value", "some-key": [{"id": 42, "is_valid":
    True}]}
    print(json.dumps(data))
    ```

# XML vs JSON

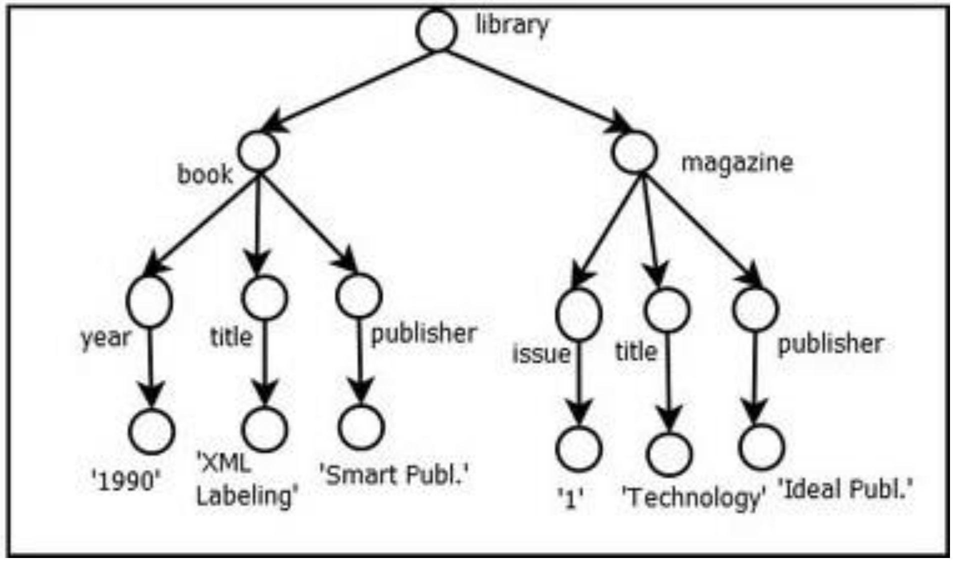
<table>
<tr><th>XML</th><th>JSON</th></tr>
<tr><td colspan="2">Human readable</td></tr>
<tr><td colspan="2">Hierarchical</td></tr>
<tr><td colspan="2">Supported by most languages</td></tr>
<tr><td>Legacy solution, wildly supported</td><td>Wildly supported</td></tr>
<tr><td>Specific parser needs to be implemented</td><td>Quick to read & write</td></tr>
<tr><td>Does not translate directly into basic python/javascript structures</td><td>Fast to parse</td></tr>
<tr><td></td><td>Smaller data storage</td></tr>
<tr><td></td><td>No start/end tags</td></tr>
</table>

# JSON: Applications

- SVG: still XML
- Config files (npm packaging, vscode configuration …)
- HTTP: SOAP API → (JSON) API
- Websocket
  - `{"id":42, "value": 56.0, "symbol": "BTCUSD"}`
  - …
- Databases
  - Structured format, so why not store it directly this way?
  - NoSQL Databases (and some SQL databases)

# JSON - Exercise 1

- Transform the following tree into a valid JSON file (by hand)



# JSON - Exercise 2

- Transform the following text into a JSON file (by hand)
- Ingredients
  - Beef, chopped: 450g
  - Onions, minced: 350g
  - Carrot, chopped: 100g
  - Potato, chopped: 150g
  - Water: 500ml
  - Golden Curry Sauce Mix: 92g
- Directions
  - Stir-fry meat and vegetables with oil in a large skillet on medium heat for approx. 5 min.
  - Add water and bring to boil. Reduce heat, cover and simmer until ingredients are tender, approx. 15min.
  - Turn the heat off, break S&B Golden Curry Sauce Mix into pieces and add them to the skillet. Stir until sauce mixes are completely melted. Simmer approx. 5 min., stirring constantly.
  - Serve hot over rice or noodles.

# JSON - Exercise 3

- Transform the following XML into a JSON
- 
```
<?xml version="1.0" encoding="UTF-8"?>
<Race date="2020-12-15" name="Holiday Meet">
    <Course>
        <CourseName>The track</CourseName>
        <Address>Track road 123</Address>
    </Course>
    <Horses>
        <Horse name="Bonfire">
            <Value>5000</Value>
            <Birthdate>1998-05-01</Birthdate>
            <Gender>M</Gender>
        </Horse>
        <Horse name="Dobby">
            <Value>1000</Value>
            <Birthdate>2001-04-05</Birthdate>
            <Gender>F</Gender>
        </Horse>
    </Horses>
</Race>
```

# JSON: How to validate ?

- JSON Schema
  - Specification (2020): https://json-schema.org/specification

```
{
  "firstName": "John",
  "lastName": "Doe",
  "age": 21
}
```

# JSON: How to validate ?

```
{
  "$id": "https://example.com/person.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "title": "Person",
  "type": "object",
  "properties": {
    "firstName": {
      "type": "string",
      "description": "The person's first name."
    },
    "lastName": {
      "type": "string",
      "description": "The person's last name."
    },
    "age": {
      "description": "Age in years which must be equal to or greater than zero.",
      "type": "integer",
      "minimum": 0
    }
  }
}
```

# JSON: How to validate ?

-
  ```
  {
    "fruits": [ "apple", "orange", "pear" ],
    "vegetables": [
      {
        "veggieName": "potato",
        "veggieLike": true
      },
      {
        "veggieName": "broccoli",
        "veggieLike": false
      }
    ]
  }
  ```

# JSON: How to validate ?

```
{
  "$id": "https://example.com/arrays.schema.json",
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "description": "Arrays of strings and objects",
  "title": "Arrays",
  "type": "object",
  "properties": {
    "fruits": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "vegetables": {
      "type": "array",
      "items": { "$ref": "#/$defs/veggie" }
    }
  }, […]
```

# JSON: How to validate ?

```
[…]
"$defs": {
  "veggie": {
   "type": "object",
   "required": [ "veggieName", "veggieLike" ],
   "properties": {
    "veggieName": {
     "type": "string",
     "description": "The name of the vegetable."
    },
    "veggieLike": {
     "type": "boolean",
     "description": "Do I like this vegetable?"
    }
   }
  }
 }
}
```

# JSON: How to validate ?

- Some other features
  - Regular expression
  - If-else
  - One of
  - All of
  - Any of
  - propertiesCount
  - Enumerations
  - …

# JSON - Exercise 4

- Re-use the JSON created for *Exercise 3*, and create the associated JSON Schema to validate it
- 
```xml
<?xml version="1.0" encoding="UTF-8"?>
<Race date="2020-12-15" name="Holiday Meet">
    <Course>
        <CourseName>The track</CourseName>
        <Address>Track road 123</Address>
    </Course>
    <Horses>
        <Horse name="Bonfire">
            <Value>5000</Value>
            <Birthdate>1998-05-01</Birthdate>
            <Gender>M</Gender>
        </Horse>
        <Horse name="Dobby">
            <Value>1000</Value>
            <Birthdate>2001-04-05</Birthdate>
            <Gender>F</Gender>
        </Horse>
    </Horses>
</Race>
```

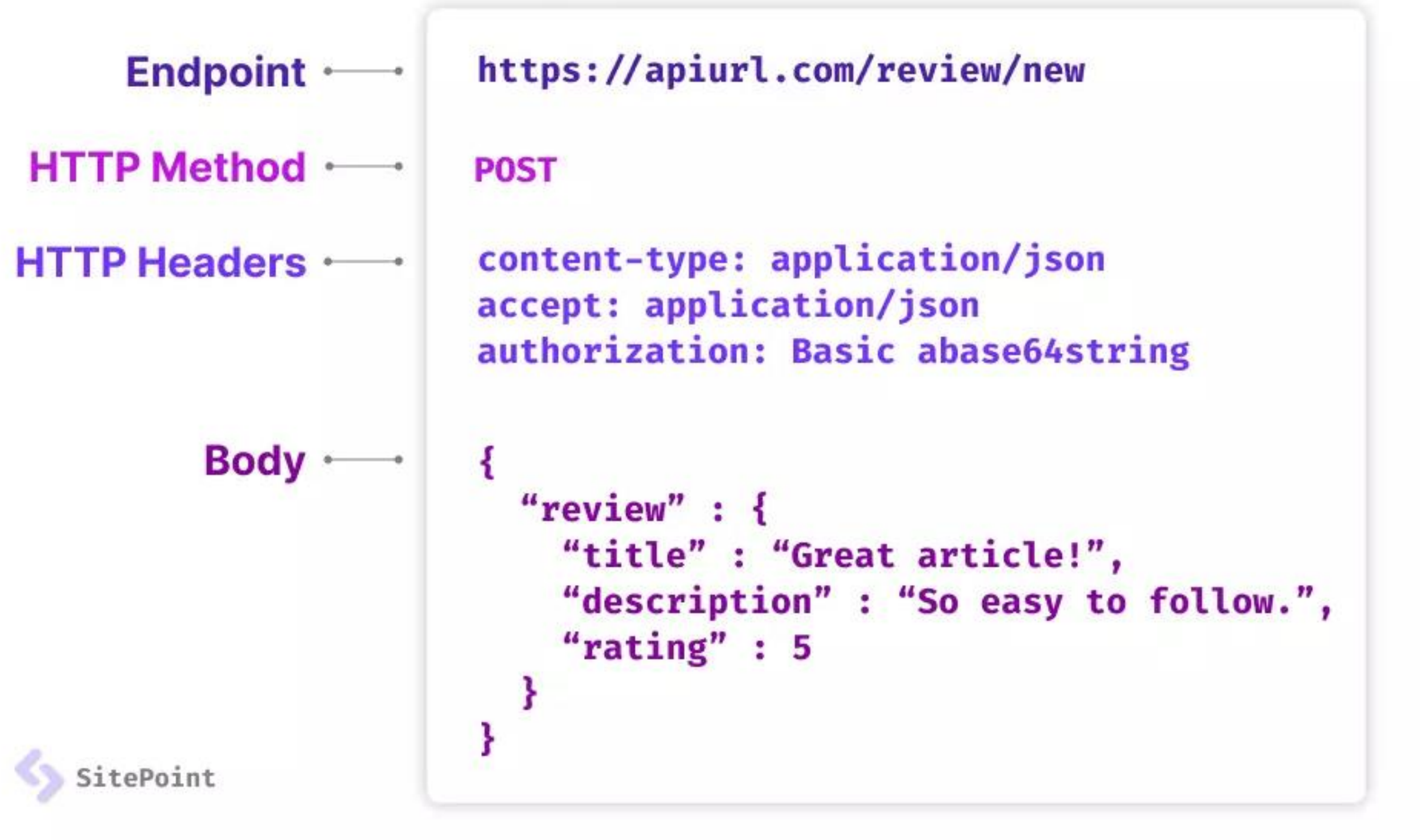
# (JSON) API

- Often used for RESTful API
- REST / Representational State Transfer
  - Set of architectural constraints, not a protocol nor a standard
  - Does not need to use JSON
- RESTful
  - API following the constraints of REST

# RESTful API

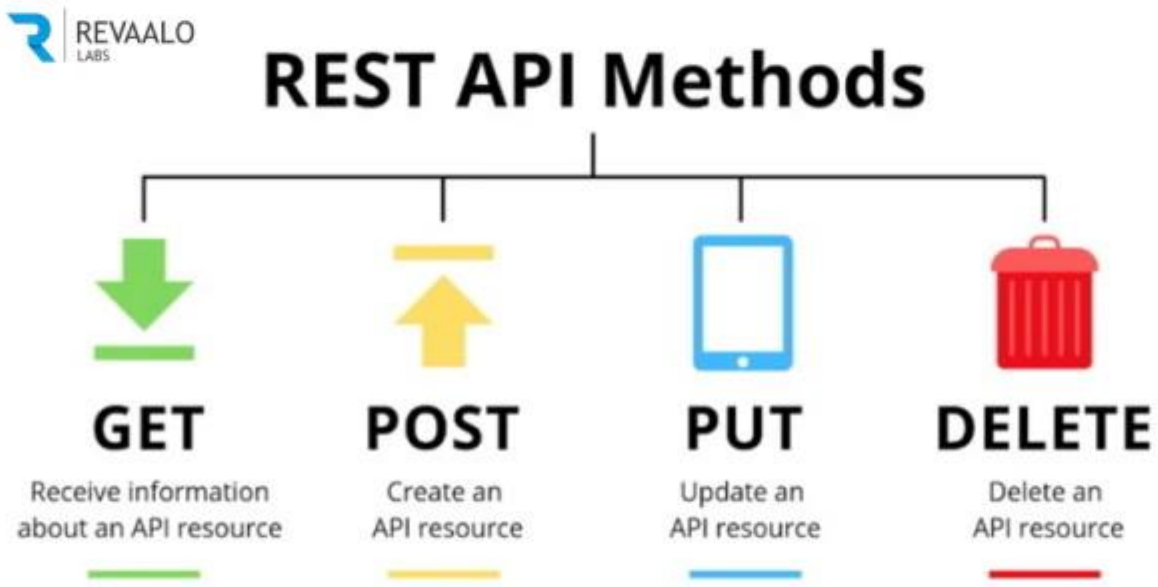
- Constraints
  - Client-server architecture, with requests managed through HTTP
  - Stateless client-server communication
  - Cacheable data
  - Uniform interface
    - A resource must have only one logical URI
    - Self-descriptive messages with enough information for the client to process it
    - Hypertext/hypermedia to find all related information
    - Your endpoints should behave the same, allowing a developer to integrate it easily
  - Layered system
    - Client contacts API on server A, but if data or authorization is done on other servers it should be transparent

# RESTful API



```
https://apiurl.com/review/new

POST

content-type: application/json
accept: application/json
authorization: Basic abase64string

{
  “review” : {
    “title” : “Great article!”,
    “description” : “So easy to follow.”,
    “rating” : 5
  }
}
```

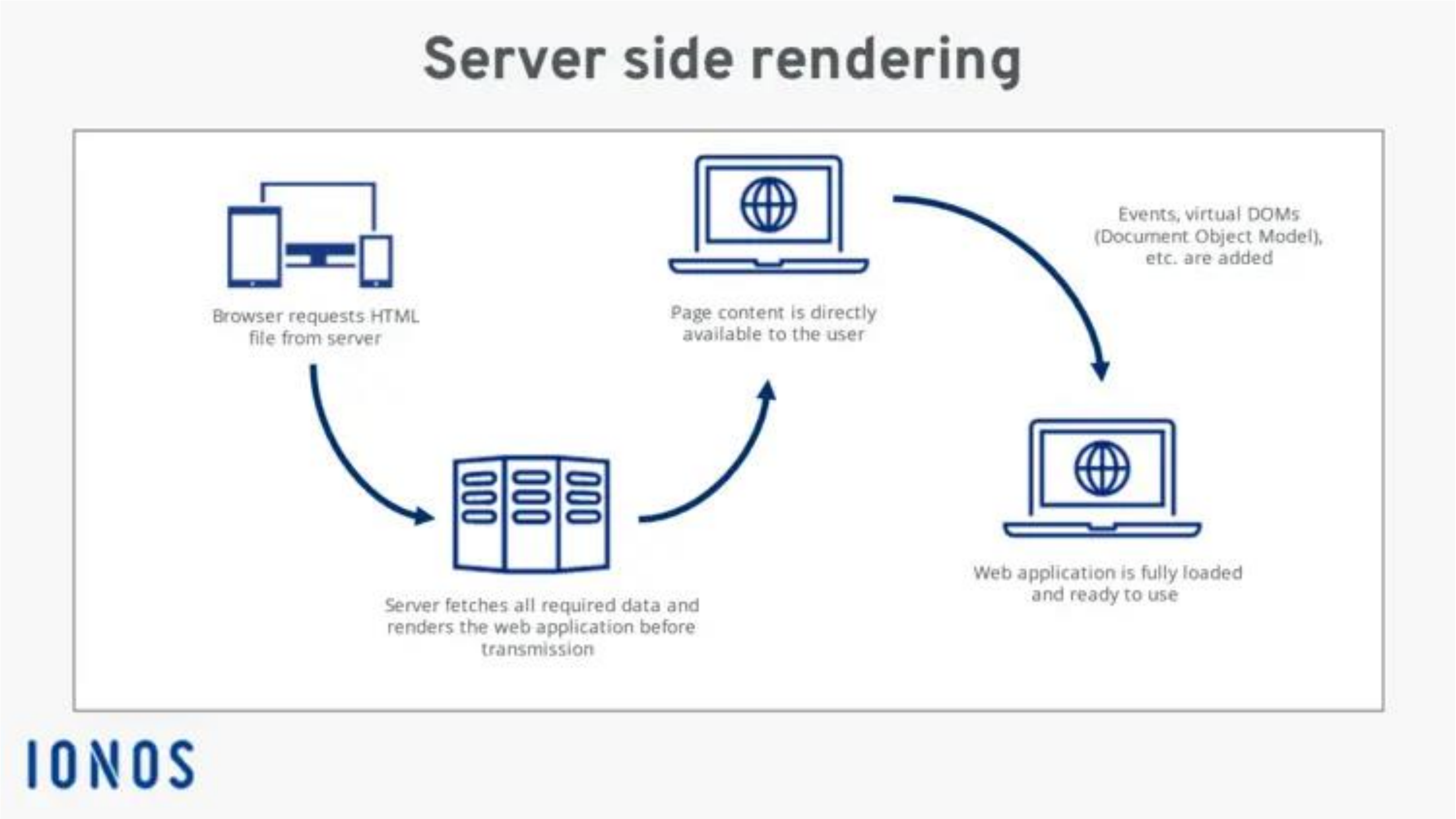
# RESTful API


REVAALO LABS
REST API Methods
GET
Receive information about an API resource
POST
Create an API resource
PUT
Update an API resource
DELETE
Delete an API resource

# RESTful API

- `/users`
  - GET + url parameters
  - POST + body
- `/user/{id}`
  - GET
  - POST
  - PUT
  - PATCH
- `/orders`
- `/order/{id}`
- …

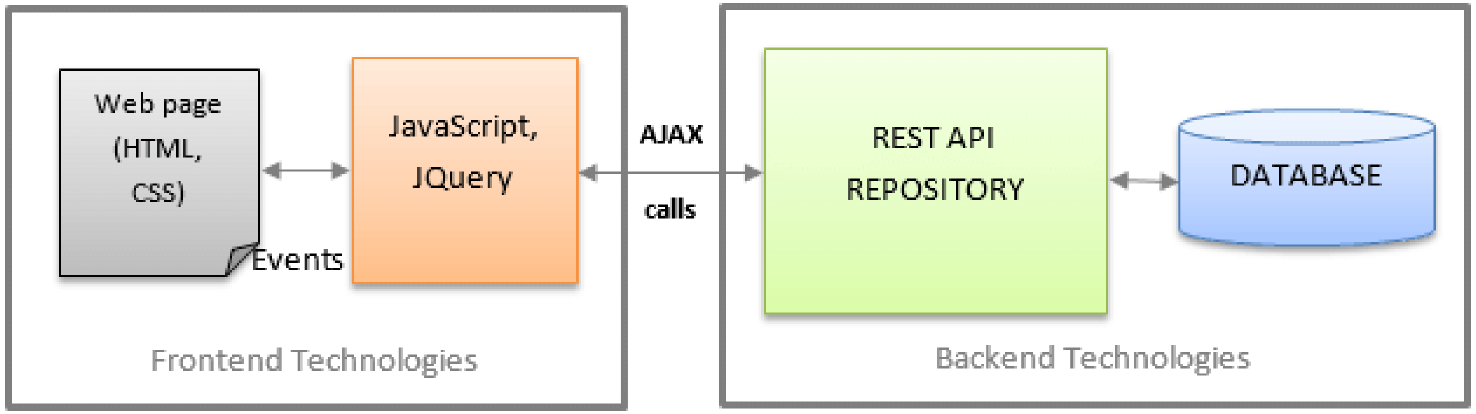
# RESTful API - Why so common?


Server side rendering
Browser requests HTML file from server
Server fetches all required data and renders the web application before transmission
Page content is directly available to the user
Events, virtual DOMs (Document Object Model), etc. are added
Web application is fully loaded and ready to use
IONOS

# RESTful API - Why so common?

- Front-end for a desktop application
  - Then, for a mobile app
  - Then, custom front-end to deal with a legacy client
  - …
- You want your website to be more "dynamic" and smoother while loading new elements
  - Infinite scrolling on most websites
  - You want the client to automatically fetch the next data!
    - In HTML? XML? JSON? …?

# RESTful API - Why so common?



# Python web api

- demo

# JSON - Exercise 5

- Use Python to create a RESTful API providing the various features
  - In memory "database"
  - (Shop) objects
    - Attributes: name, remaining quantity, price
    - List, create, update & delete
  - Orders
    - Attributes: object_id, quantity
    - List, create, update & delete
  - Apply some basic validations
    - quantity >= 0, name must be of length [4;100], birth date is a real date …
- Provide a python script to test each endpoint

# JSON - Credits & references

- Json
  - https://www.json.org/json-en.html
  - https://www.w3schools.com/js/js_json_intro.asp
- Json Schema
  - https://json-schema.org/specification
  - https://json-schema.org/learn/miscellaneous-examples

# NoSQL Databases

Why?

How to use it ?

How to validate it ?

# Why use a database?

- Efficient and persistent
- More flexible than using files
- Handle concurrent access
- Libraries to easily integrate with any programming language
- SQL / Relational databases share a lot of similarities
  - Many libraries handle PostgreSQL, MySQL, SQLite without any change
  - But each of them has custom features

# Software and data

- Data usually lives longer than software
- Data is extremely valuable
  - Must be easy to interact with and stable
- Data should be at the center of the architecture

# Relational databases

- Schema
- Tables
- Relationship between tables
- Easy querying using SQL
- Most common relational databases
  - MySQL / MariaDB
  - PostgreSQL
  - Microsoft SQL Server
  - SQLite (for local development)

# Relational databases - Limitations

- Relationships
  - Indexes: RAM consumption, update overhead
  - High correlation between tables
- How to scale?
  - Vertically (Single server): Hardware limitations
  - Horizontally (multi-servers): How do deal with relationships efficiently?
  - Complex schema changes for large databases (1 TB+)
- We always manipulate (json) objects, so why use SQL at all?
  - NoSQL Databases

# Scaling

- Vertical Scaling / Scale up
  - More powerful server
  - Architecture stays the same
- Horizontal Scaling / Scale out
  - Add more servers
  - Architecture needs to be designed for it
    - ! SQL Databases are still possible
    - All processes will not necessarily see the same state

# Scaling - Database storage

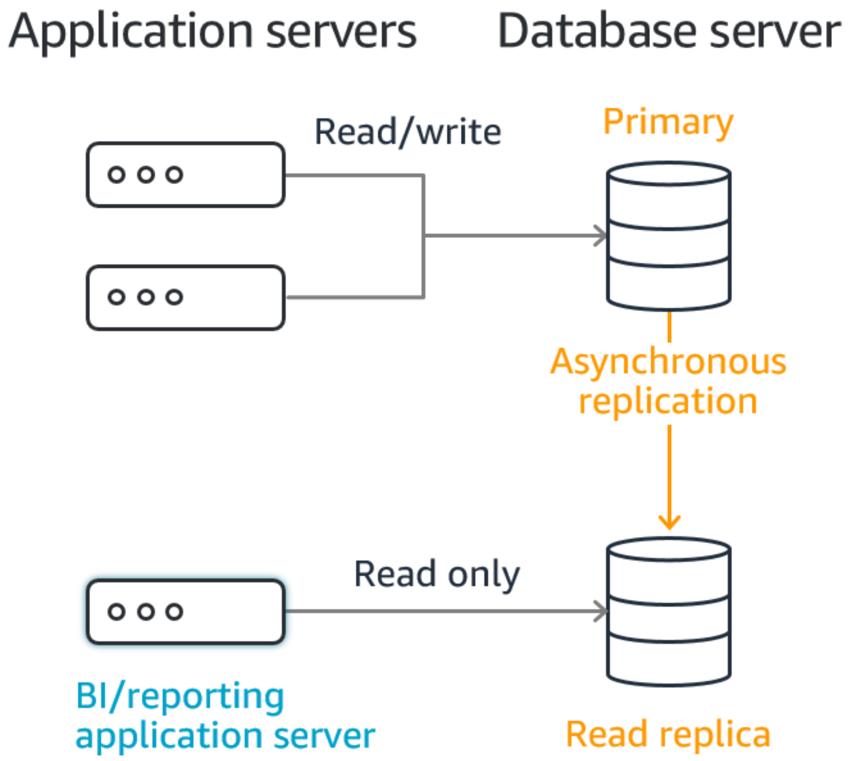
- Database storage
    - Often one table (or database) = one file
    - Re-use deleted rows for new rows
    - Colocation of data is important
- Issues with "one file"
    - Backup
    - Schema changes
    - File system limitations
    - Handling of many deletions / updates: iops, lost disk space

# Scaling - Database storage

- Issues with "many files"
  - Backup
  - Schema changes
  - File system limitations
  - Handling of many deletions / updates: iops
- Middle ground: partitions
  - User id [0…1000] → file 1
  - User id [1001…2000] → file 2

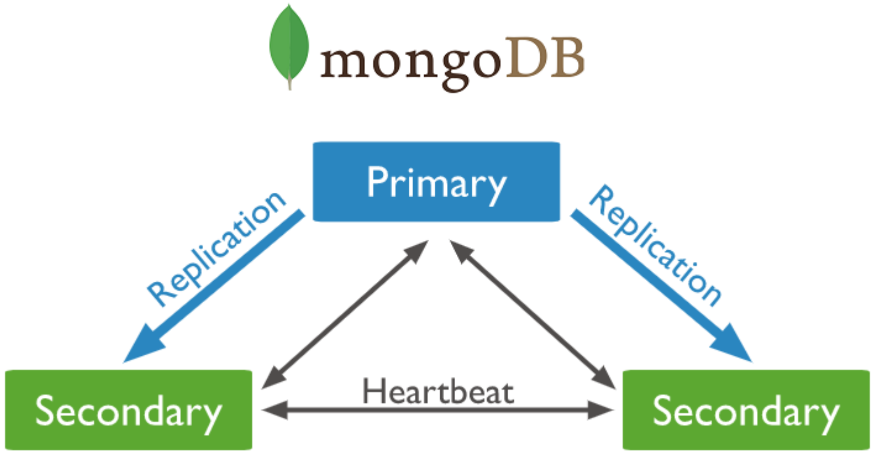
# Scaling - Horizontal scaling

- Why do we want to scale?
  - Too much to write?
  - Too much to read?
  - Both?
- Lots of read operations
  - Read-replicas (1-3 are common)
  - Async replication with configurable delay
  - Software should be aware of it
  - Each server must still store everything



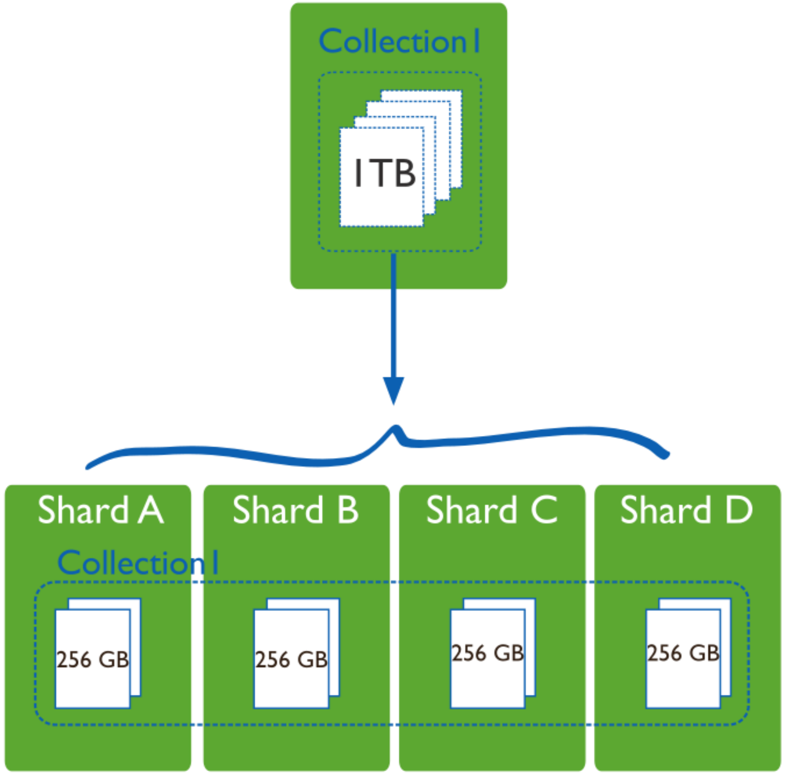
# Scaling - Horizontal scaling

- Primary
  - Accepts Read & Write
- Secondary
  - Accepts Read
- Handling failure of the primary
  - Primary election
  - Software needs to know all nodes
  - Odd number of nodes is required



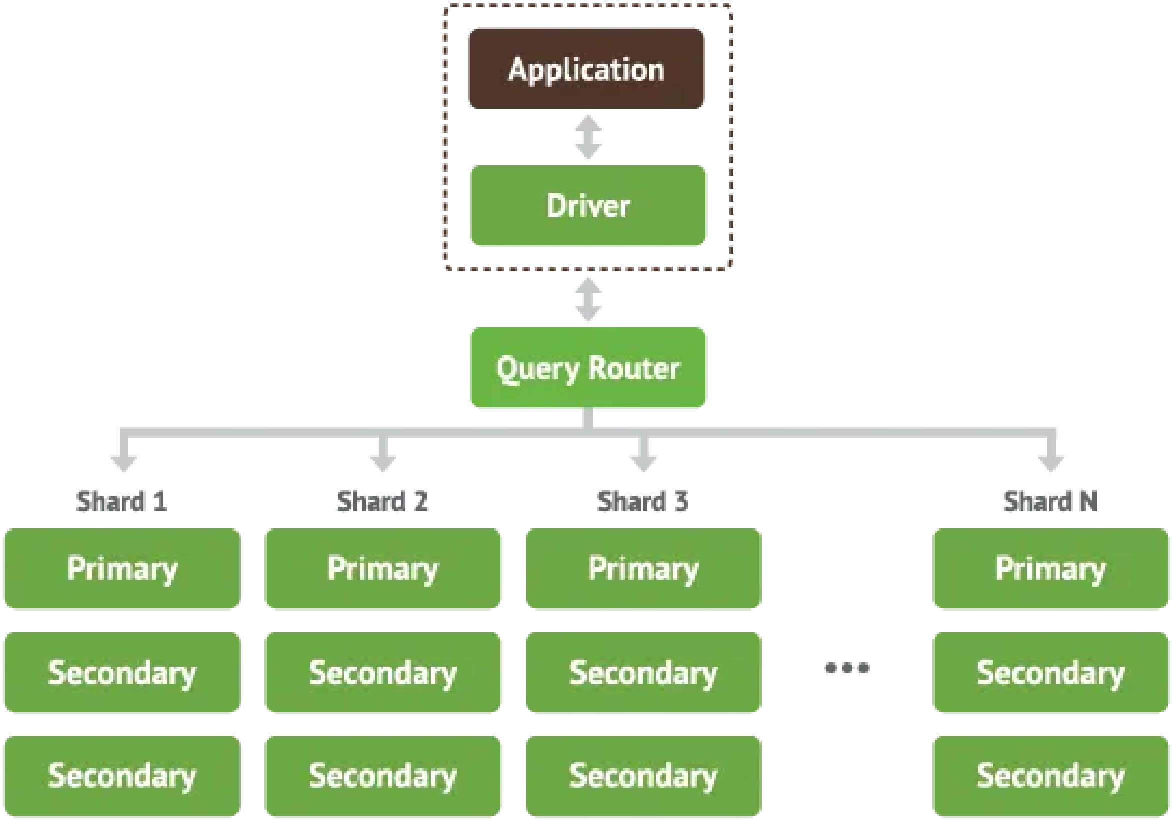
# Scaling - Horizontal scaling

- Read replica
  - All the data still in each server
  - How to handle TBs?
  - Partitions at the cluster level: sharding



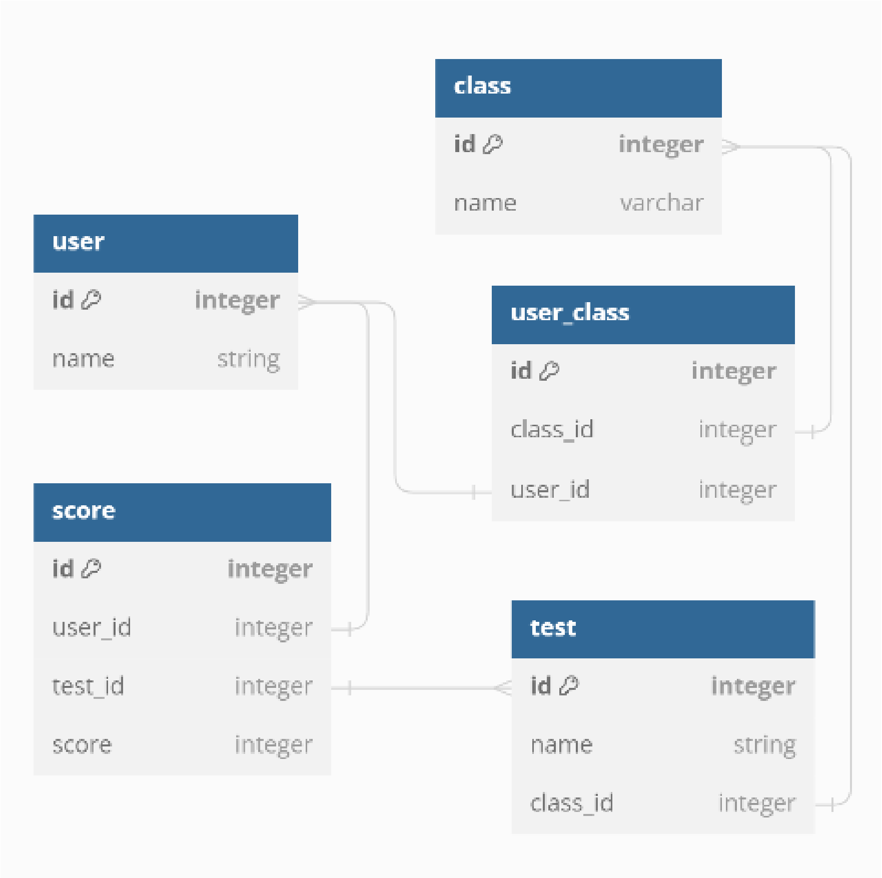
# Scaling - Horizontal scaling

- Read replica + Sharding



# Scaling - Horizontal scaling

- But JOINs ?
  - Highly normalized
- Example
  - Students registered to classes
  - Students have scores for tests
- How to scale?
  - Partition every table?
  - Latency ?
  - Bandwidth ?
  - RAM ?



| class | |
|---|---|
| id | integer |
| name | varchar |

| user | |
|---|---|
| id | integer |
| name | string |

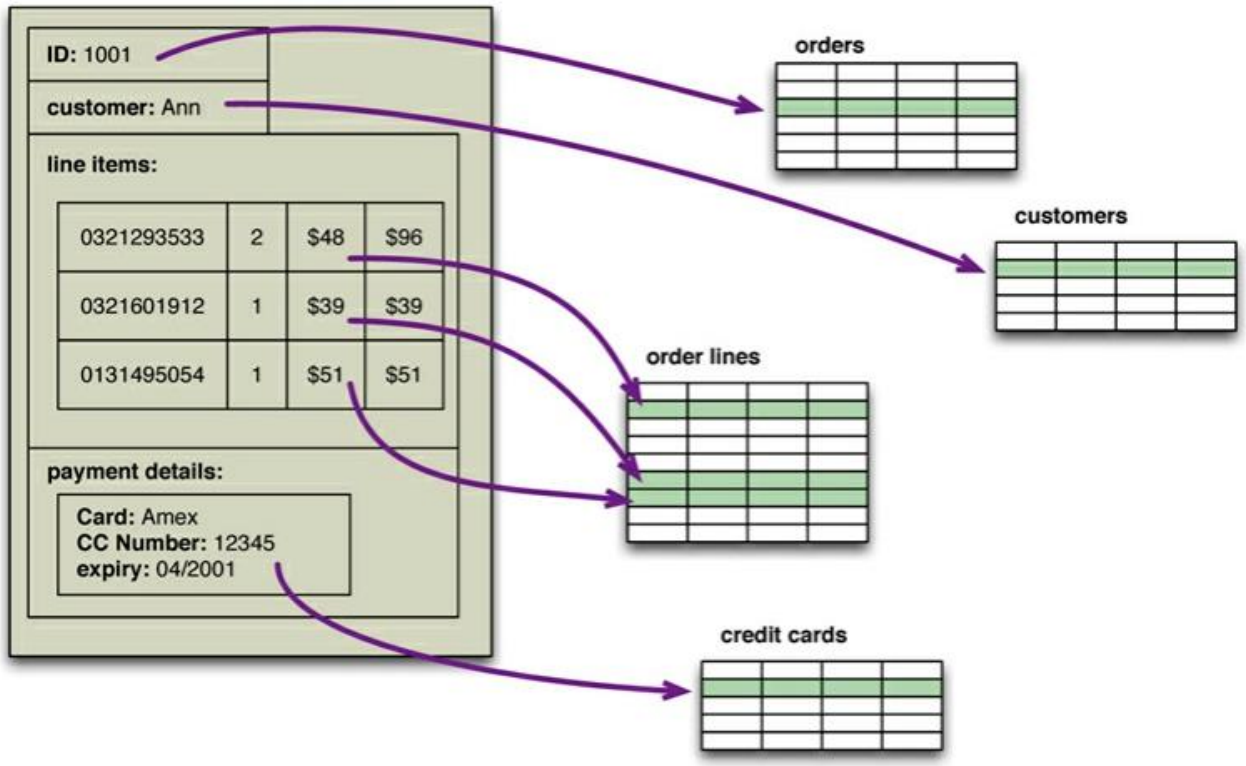
| user_class | |
|---|---|
| id | integer |
| class_id | integer |
| user_id | integer |

| score | |
|---|---|
| id | integer |
| user_id | integer |
| test_id | integer |
| score | integer |

| test | |
|---|---|
| id | integer |
| name | string |
| class_id | integer |

# Impedance mismatch

- User point of view
  - A single document
- Developer point of view
  - Multiple tables to manage
- Impedance mismatch
  - Difference between the relational model and the in-memory data structures



# NoSQL Database - MongoDB

- Optimize storage for read
  - "Similar" to a Materialized View managed by yourself
- High freedom
  - Add/remove any field
  - Set any type
- Every user object is handled separately (no constraint)

```json
{
  "id": 0,
  "name": "Alex",
  "classes": [
    {
      "id": 42,
      "name": "Programming"
    }
  ],
  "scores": [
    {
      "test": {
        "id": 25,
        "name": "1st test"
      },
      "score": 20
    }
  ]
}
```

# Relational Databases - Issues

- Conversion of data between end-user and data storage
- Reconstruction of data from tables
- Fixed data model
- Relational databases forces columns of a specific type (generally)
- Scaling issues
- Complicated searching in relational database
- But
  - SQL
  - Many features to do "anything" (streaming, ...)
  - Schema enforcement

# Non-Relational Databases - Issues

- Almost no data type enforcement
- Data Model is extremely free (few available constraints)
- Limited set of features
- Specific languages
- Transactions are generally not supported natively
- But
  - Designed for scalability
  - Data type freedom (media, text, json, …)
  - Do a few things efficiently
  - Schema enforcement

# Which one to choose?

- RDBMS are powerful and stable
- NoSQL DBs are specialized and easily scalable
- Many architectures use both

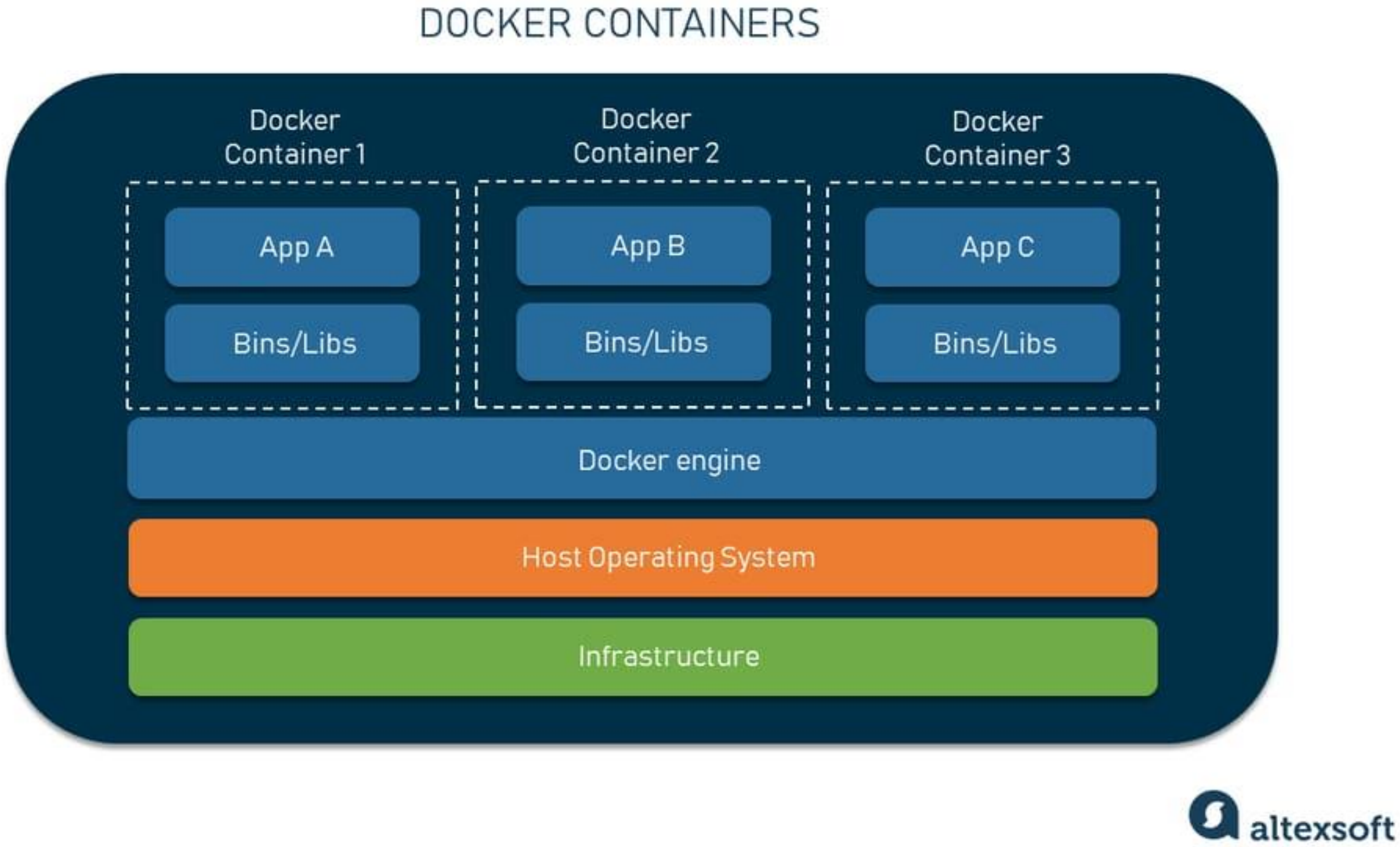
# Which one to choose?

- MongoDB
- Cassandra
- Couchdb
- Hbase
- Redis
- Neo4j
- Amazon AWS
  - RDS
  - DynamoDB
  - …
- …

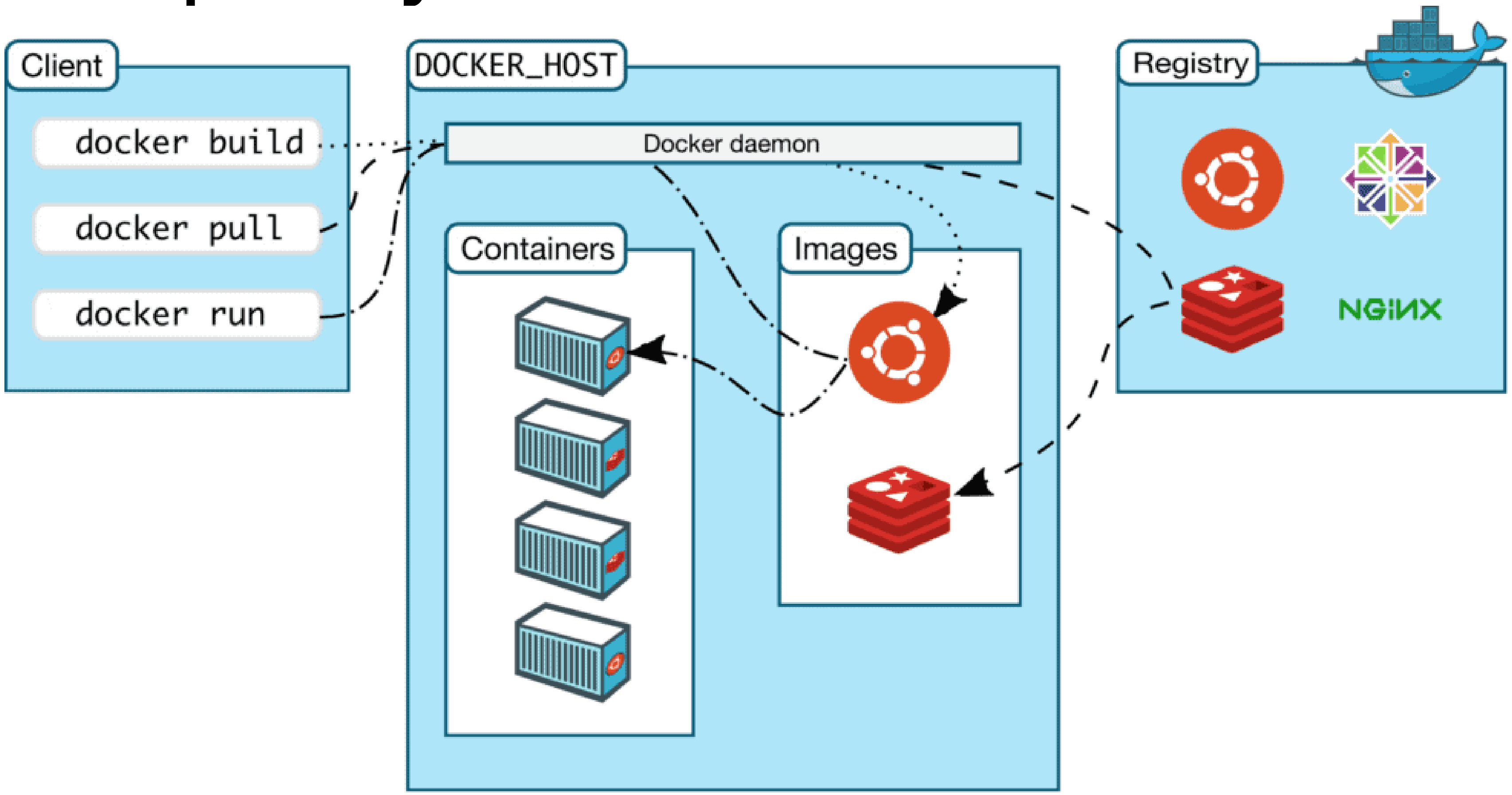
# How to quickly test a database?

- Manual installation of redis
  - https://redis.io/docs/latest/operate/oss_and_stack/install/install-redis/
  - Centos
    - `sudo yum install redis && sudo system start redis`
  - How to remove all files?
    - `sudo yum uninstall redis`
    - But… some files will remain
    - How to quickly test various versions and make sure all dependencies are properly removed / installed?
  - How to make sure all the applications are installed together for your software?
    - Bash script but how to handle updates? Deletion? …

# How to quickly test a database?



# How to quickly test a database?



# How to create a docker image?

- Dockerfile

  -
    ```
    FROM python:3.8-slim-buster
    WORKDIR /python-docker
    COPY requirements.txt requirements.txt
    RUN pip3 install -r requirements.txt
    COPY . .
    CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
    ```

  -
    ```
    docker build -t flask:0.1.0 .
    ```
- Redis
  - https://github.com/redis/docker-library-redis/blob/master/Dockerfile.template
- MongoDB
  - https://github.com/docker-library/mongo/blob/master/Dockerfile-windows.template

# How to quickly test a database?

- Install docker
  - https://docs.docker.com/engine/install/
  - `docker run --name some-redis -d redis`
- Install docker-compose
  - https://docs.docker.com/compose/install/
  - Create a `docker-compose.yml`

```yaml
version: '3'

services:
  redis:
    # https://hub.docker.com/_/redis
    image: redis:7.4.0
    container_name: redis
    restart: unless-stopped
```

# How to quickly test a database?

- Interact with your processes
  - `docker compose up redis`
  - `docker compose up redis -d`
  - `docker compose down`
  - `docker logs -f redis`
- How to enter a docker?
  - `docker exec redis -it /bin/bash`
- How to keep your data?
  - Volumes
- How to provide some configuration files?
  - Volumes
- How to expose a port?
  - Ports

# NoSQL - Exercise 1

- Create a docker-compose which contains the following services (single node)
  - Postgres
  - MongoDB
  - Cassandra
  - Couchdb
  - Redis