Intro to Web 2.0 and RESTful WebAPIs

Software Engineering - Lab

Marco Robol - marco.robol@unitn.it

Contents

- Web2.0 - Overview
- RESTful WebAPIs - Concepts
- Exploring APIs with Postman

In the next labs... designing and documenting your APIs with OpenAPI specification language and implemention with Node.js Express web framework.

From Static Website to Web Applications and *Web Services*

- **Web 1.0** - Static content with HTML and hyperlinks. Interaction was only possible through HTML forms, where response pages were then generated on the server.
- **Web 2.0** - Javascript-based **web applications** powered by asynchronous HTTP requests APIs and DOM manipulation supported by most browser.
  Availability - Accessibility - Sharing - Compatibility

Asyncronous data retrieval and page modification

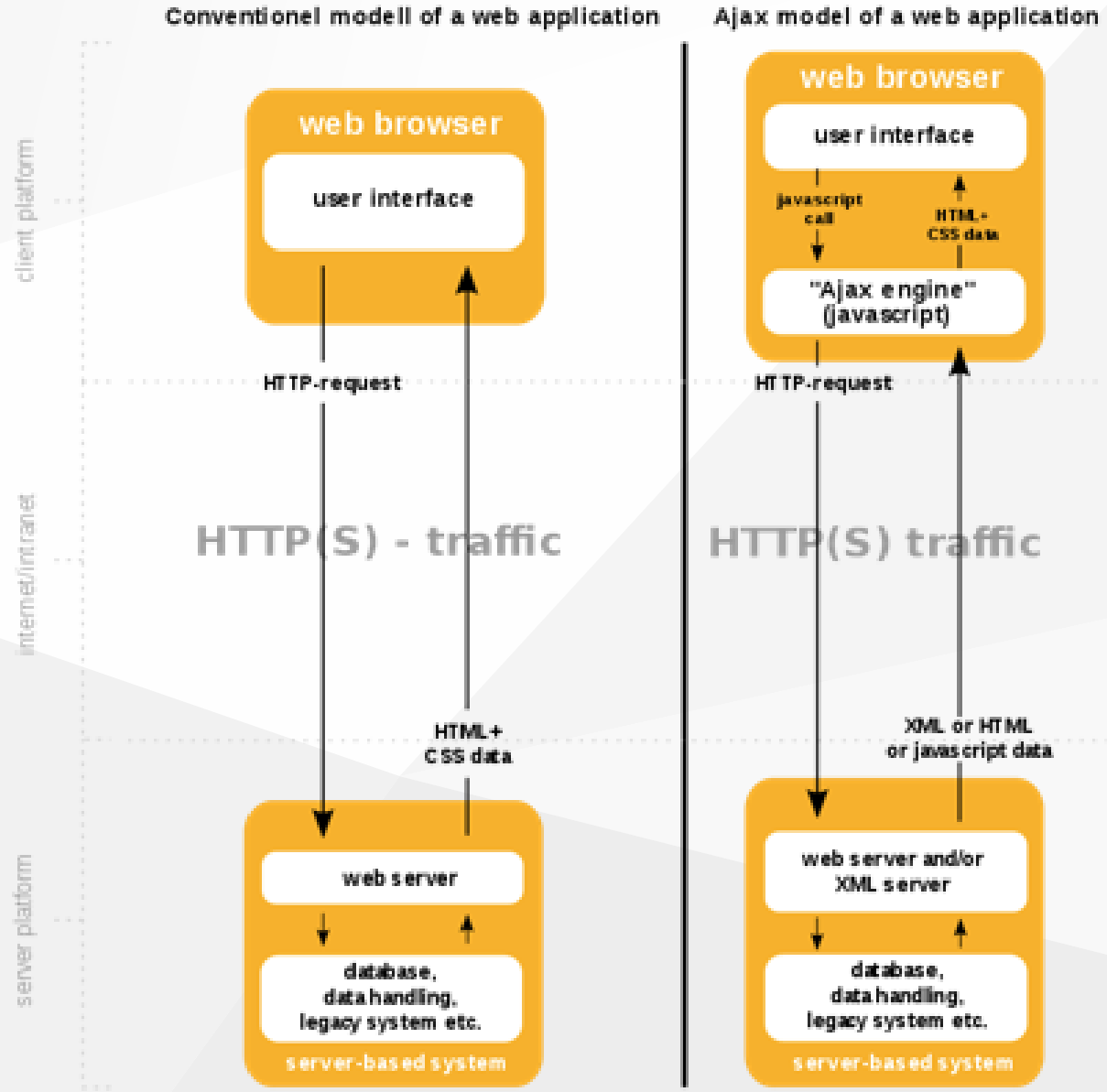
By decoupling the data interchange layer from the presentation layer Web applications:

- **send and retrieve data from a server asynchronously** (in the background) without interfering with the display and behaviour of the existing page.
- **change content dynamically** without the need to reload the entire page.

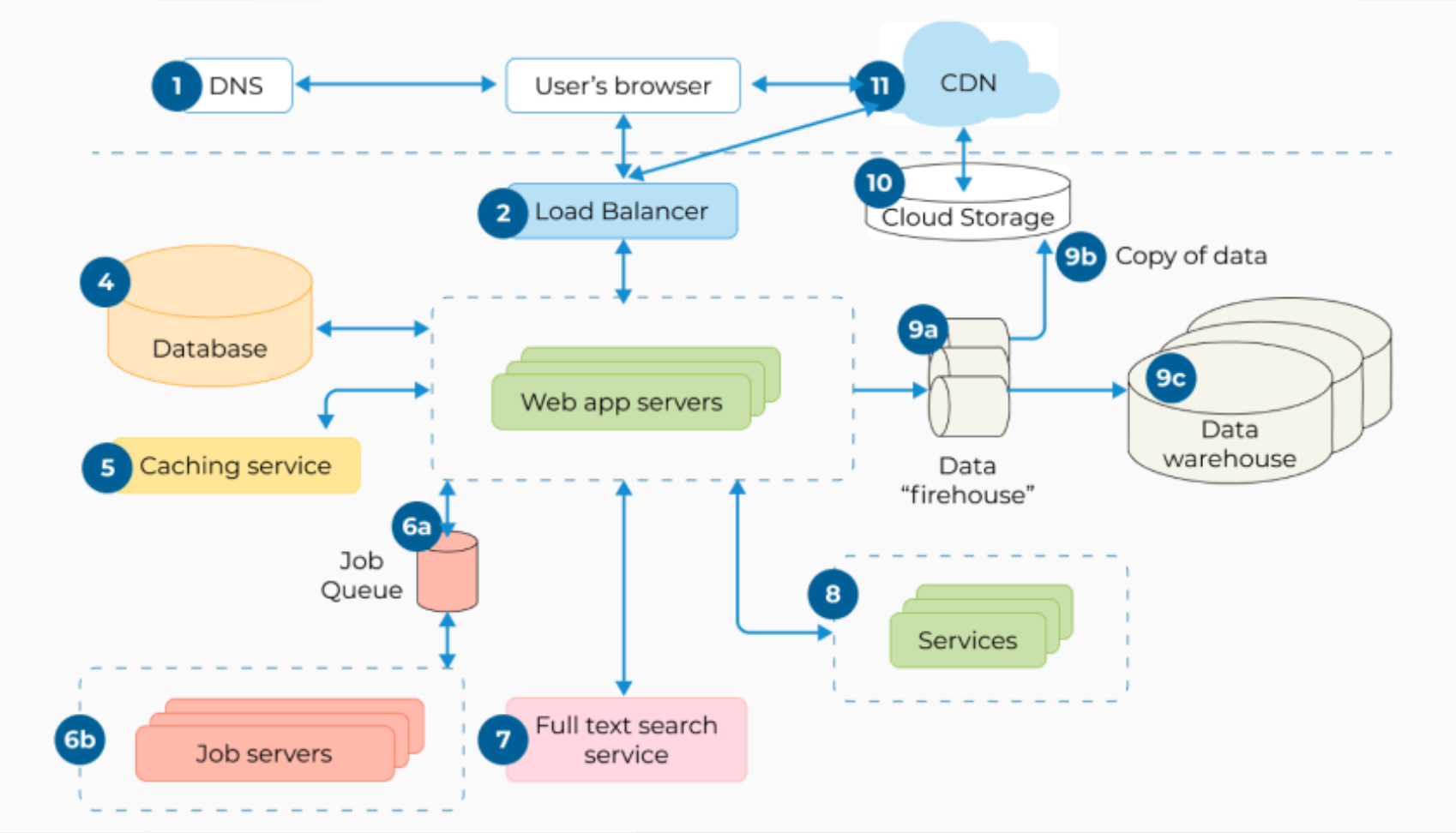
Initially presented as *AJAX* (Asynchronous JavaScript and XML), is the idea of using a set of web technologies on the client-side (including the built-in XMLHttpRequest) to create asynchronous web applications.

HTML and CSS are used to mark up and style information. **JavaScript** is used to dynamically modify and display the new information.

> https://en.wikipedia.org/wiki/Ajax_(programming)


Conventionel modell of a web application
Ajax model of a web application
client platform
internet/intranet
server platform
web browser
user interface
HTTP-request
HTML+
CSS data
web server
database,
data handling,
legacy system etc.
server-based system
HTTP(S) - traffic
web browser
user interface
javascript
call
HTML+
CSS data
"Ajax engine"
(javascript)
HTTP-request
XML or HTML
or javascript data
web server and/or
XML server
database,
data handling,
legacy system etc.
server-based system
HTTP(S) traffic

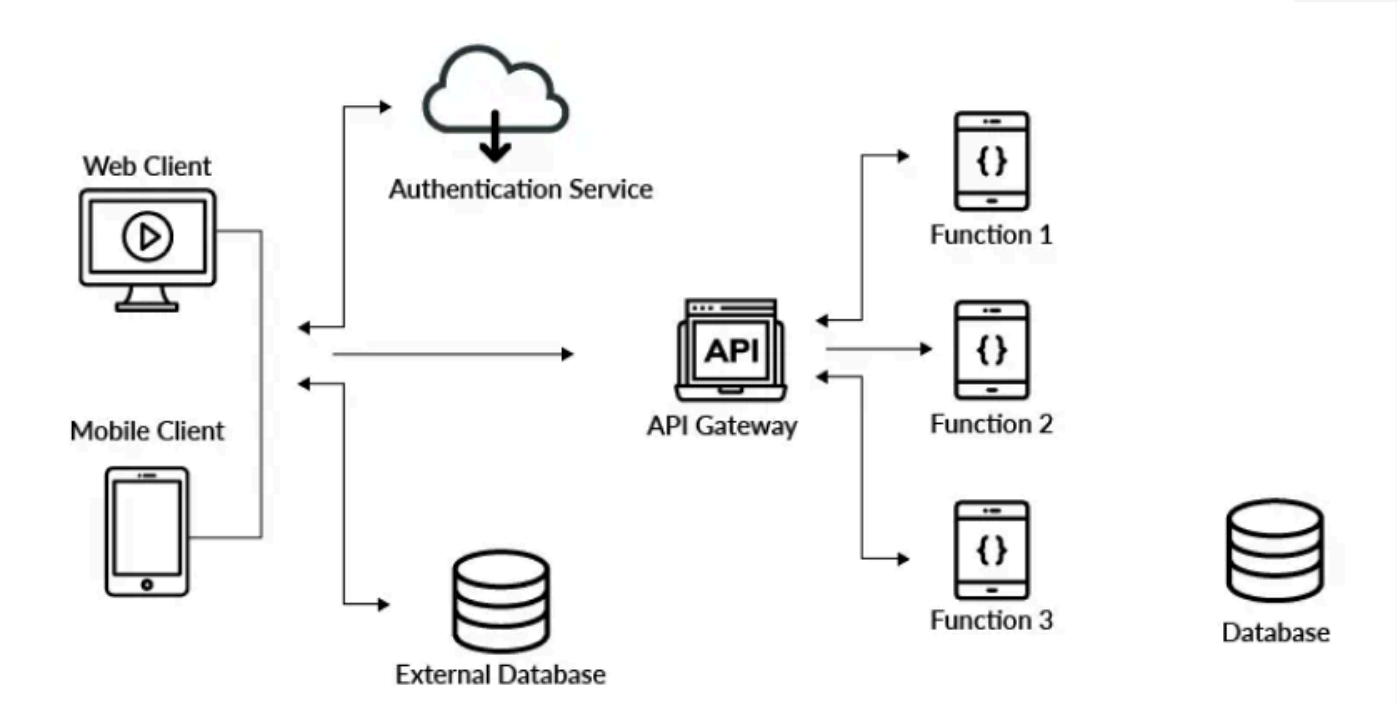
Modern Web Application Architecture



https://integrio.net/blog/modern-web-application-architecture

- Serverless Architecture

> In this architecture, the backend is built using cloud-based solutions, such as AWS or Azure. Each function is responsible for tasks like registering users or sending email.
> https://integrio.net/blog/modern-web-application-architecture

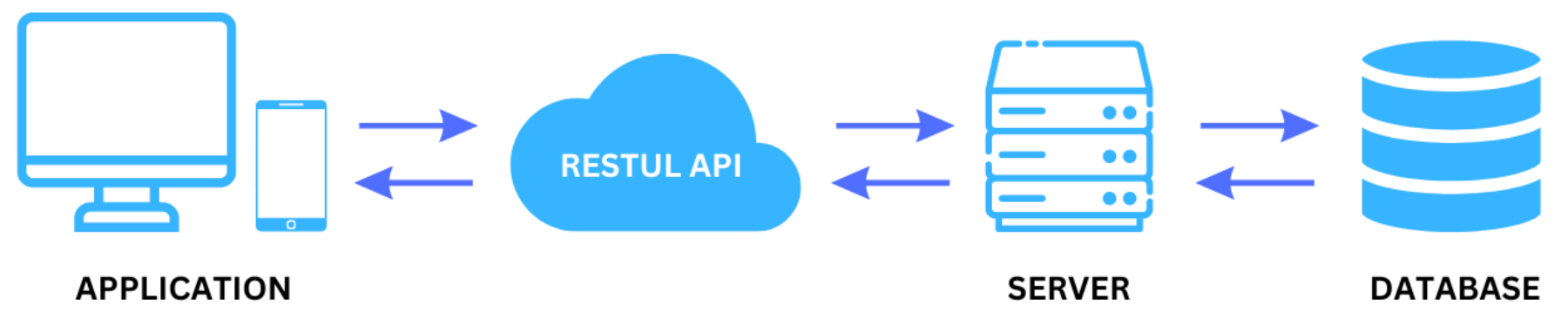


Design choices for this lab

- 1 RESTful **Web Service** implemented in Node.js. This will be deployed on the cloud on render.com or a similar **PaaS**.
- 1 frontend developed in Vue.js as a **Single-Page Applications** (SPAs) e.g. Gmail, Trello, Spotify, and Twitter. Still, additional pages are allowed.
    - No Server-Side Rendered Application (SSR) e.g. WordPress, Airbnb.
    - Served as static content through CDN (Content Delivery Networks delivers content from the closest server). e.g. Cloudflare, Akamai, and Amazon CloudFront.

RESTful WebAPIs

- An **API** stands for an application programming interface. It defines how applications or devices can connect to and communicate with each other.
- **REST** (Representational State Transfer) is an architectural style for APIs that use HTTP method to access and manipulate web resources using a uniform and predefined set of stateless operations. (Source wikipedia and restful-api.dev)



RESTful concepts

> A complete guide to RESTful www.restapitutorial.com

- **Resources**: Web resources can be identied by an URI (universal resource identifier - urls are the most common type of identifiers).
- **Representation**: json, xml, ...
- **Operations**: CRUD operations are mapped to the standard HTTP verbs

For example, given a RESTful API for managing products, the http request `GET /api/products` would return:

```
{ "id" : 1,
  "name" : "iPhone XL",
  "description" : "Extra large"
}
```

Operations

CRUD operations are mapped to the standard HTTP verbs. In our example we will have that:

Operation	HTTP Verb	URI	Req body	Resp body	success
Search	GET	/products	Empty	[Product+]	200
Create	POST	/products	Product	Product	201
Read	GET	/products/:id	Empty	Product	200
Update	PUT / PATCH	/products/:id	Product*	Product	200
Delete	DELETE	/products/:id	Empty	Empty	204

- **GET** - **Read existing data** from a data source. The response can contain data such as a list of items, a single item, or even just a status message. A GET request is a **safe** and **idempotent** method, meaning that it can be repeated multiple times without having any side effects because it should only retrieve data, not modify it.
- **POST** - **Create a new resource**. The data which is sent as part of a POST request is encoded in the body of the request and is not visible in the URL.
- **PUT** - **Update an existing resource**. The request body should contain a *complete* representation of the resource, otherwise, the missing fields will be set to NULL, or the request just would fail. Look at PATCH for a partial update.
- **DELETE** - **Delete an existing resource**. ID of the resource is specified in the URL.
- **PATCH** - **Partially update an existing resource**. You specify only fields to be modifed.

Status codes

Using status code in RESTful APIs:
https://www.restapitutorial.com/lessons/httpmethods.html

A complete list of HTTP status code: restapitutorial.com/httpstatuscodes.html

Links

Complex APIs require special attention to the relationship between web resources, and ways of traversing the relationships. The same resource could be accessed under different URLs. For example, to get the list of products associated to a company (`/company/:id/products`).

To easily navigate through resources, we should use links in place of ids. For example:

```
// GET /products/123 JSON:
{ "self": "/products/123", "name": "iPhone", "producer": "/company/456" }

// GET /company/456 JSON:
{ "self": "/company/456", "name": "Apple", "products": "/products?producer=/company/456" }
```

> ***Read the following article:*** API design: Why you should use links, not keys, to represent relationships in APIs

Let's check out existing RESTful APIs on the web

Use Postman to test RESTful APIs at restful-api.dev

1. https://jsonplaceholder.typicode.com/ Fake REST APIs for testing and prototyping;
2. https://cloud.google.com/translate/docs?hl=it Google Translations APIs;
3. https://openweathermap.org/api Open Weather Map;
4. https://restcountries.com/ REST Countries;
5. https://ipapi.co/ IP API provides you with information about an IP address;
6. https://random-data-api.com/ Random Data API;
7. https://pokeapi.co/ The Pokemon API is a simple API for Pokemon;

> https://dev.to/ruppysuppy/7-free-public-apis-you-will-love-as-a-developer-166p

EasyLib

Web service for the management of book lendings to students.

> Repository: https://github.com/unitn-software-engineering/EasyLib

> APIs documentation: https://easylib.docs.apiary.io/#

> Deploy https://easy-lib.onrender.com

Designing your RESTful APIs

Start from your user stories and provide RESTful APIs for each of them:

- Identify resources and define their structure
- Organize into main and sub-resources
- Define supported methods and accepted parameters / constraints
- Define returned status codes

Questions?

marco.robol@unitn.it