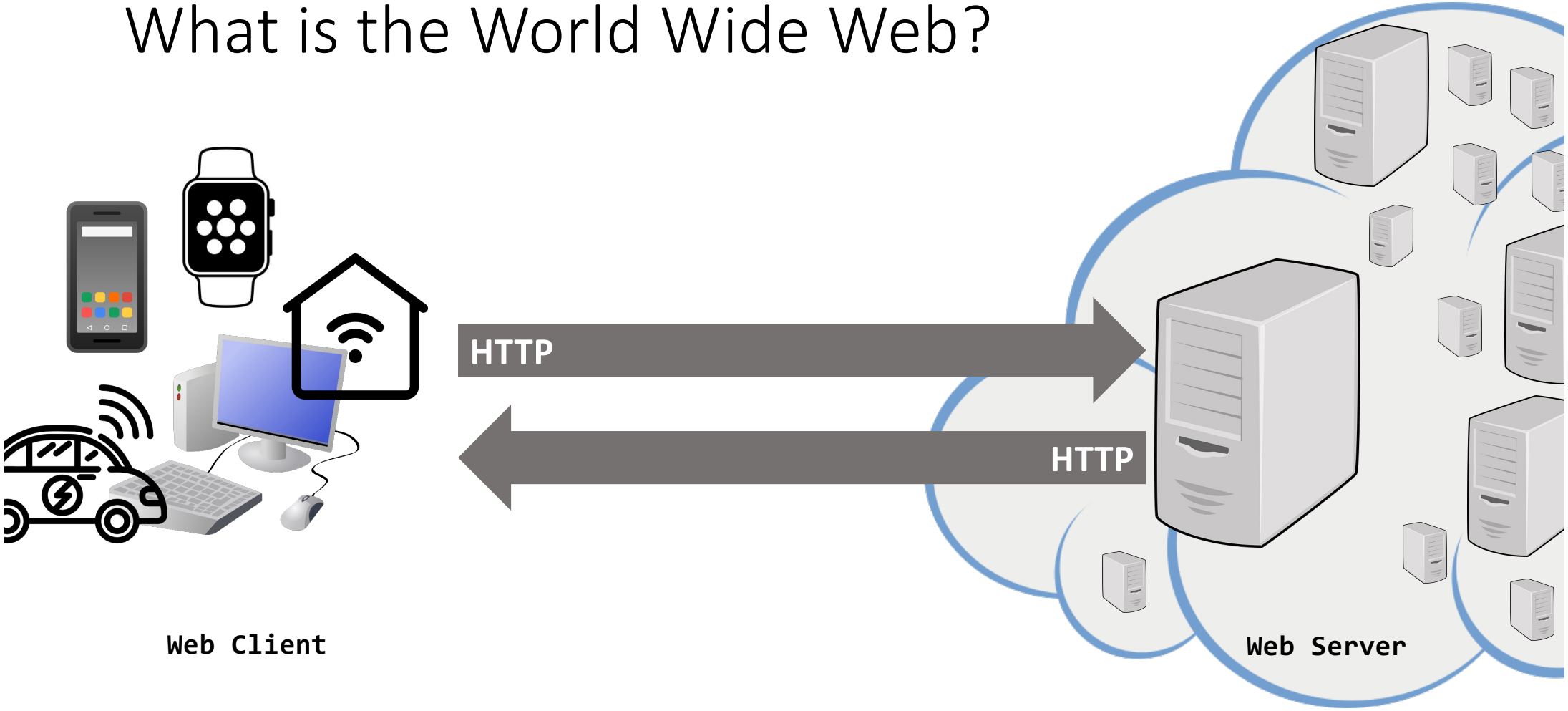
Introduction to Web Development

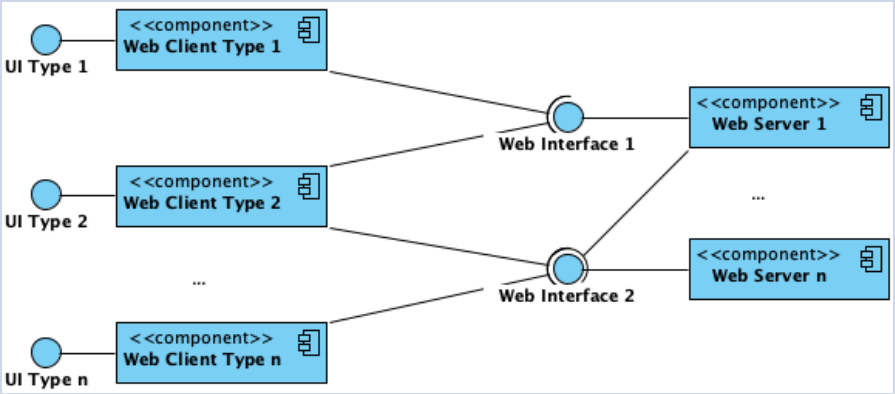
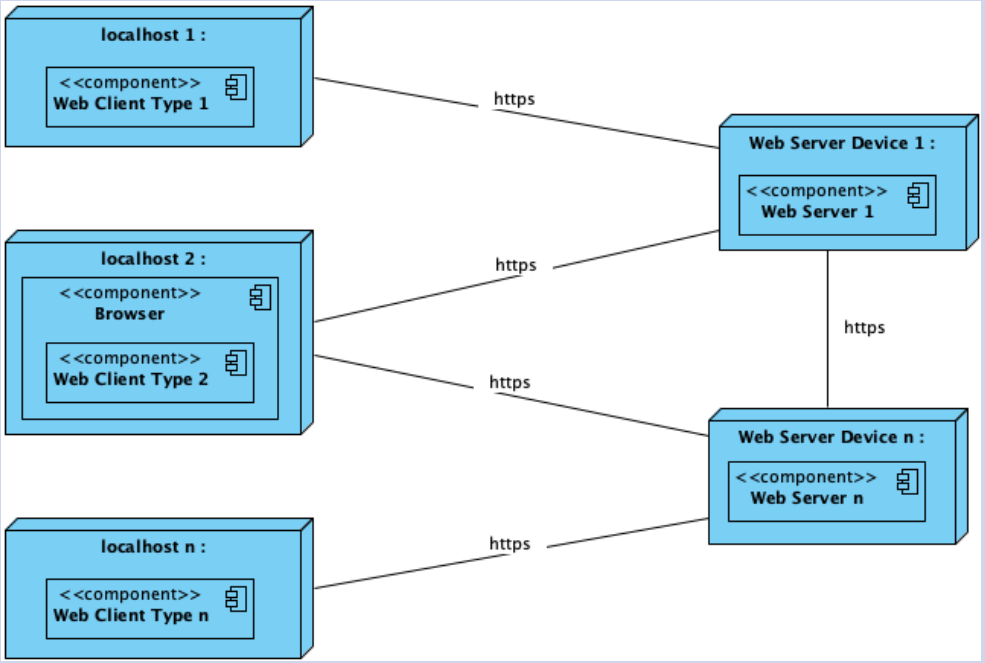
ARQSI 2022-2023

Instituto Superior de Engenharia do Porto

What is the World Wide Web?

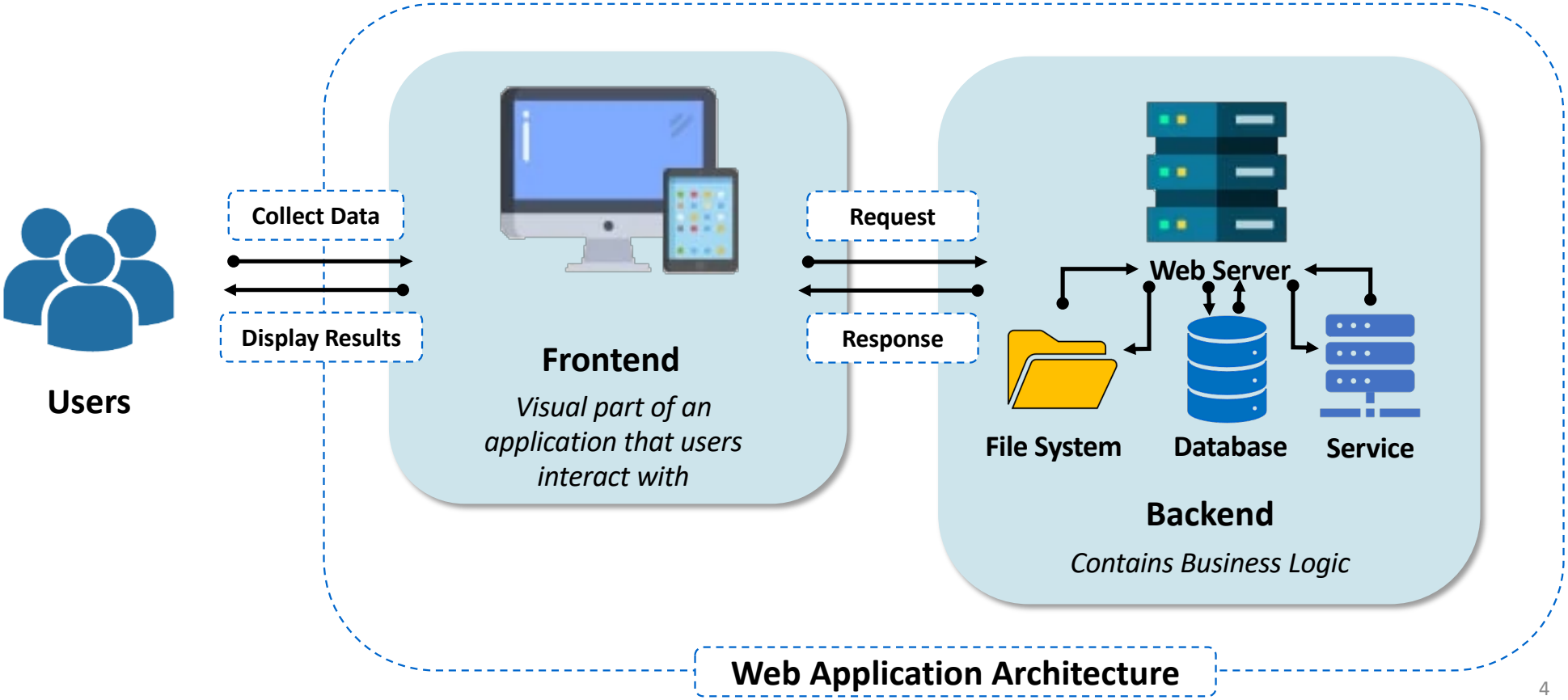


in UML

Logical View	Physical View
	

Web Application (details)

How would it be represented in UML?



Client-Server responsibilities

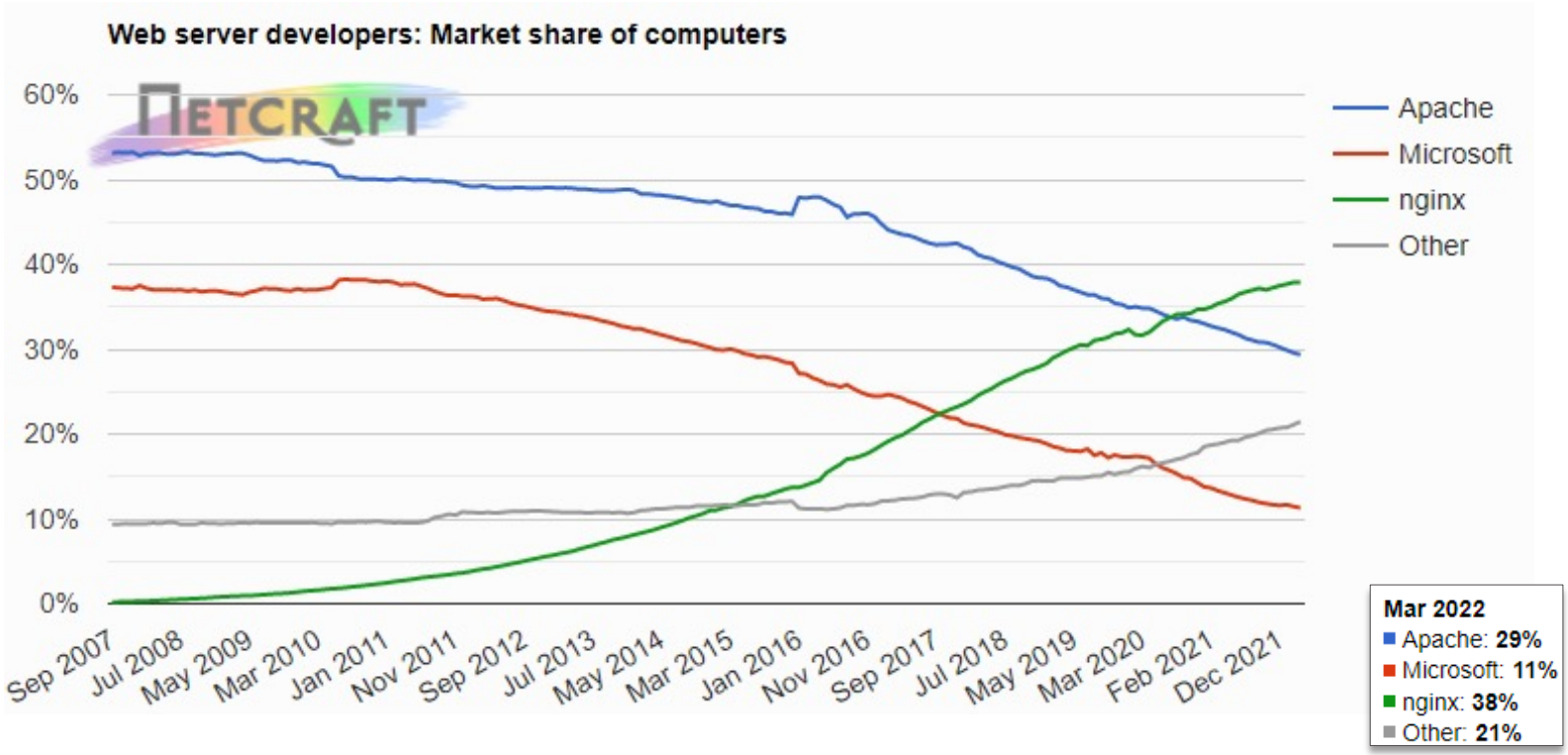
- Client
  - Sends request to the server
    - Protocol: HTTP/S, but other are possible, e.g. FTP
  - Renders the received content (e.g. JSON, XML), using technologies such as:
    - HTML (Hyper-Text Markup Language): language for characterization of content
    - CSS (Cascade Style Sheets): language for definition of markup (HTML) formatting styles
    - Javascript: scripting language for the runtime manipulation of data content, format and (internal and external) events
- Server
  - Receives the client's request
  - Prepares the response content using technologies such as:
    - PHP
    - .Net (C#, VB, F#)
    - Node.js
    - Java
  - Sends the response to the client, serialized in e.g.:
    - HTML
    - XML
    - JSON

Web Server and Application Server

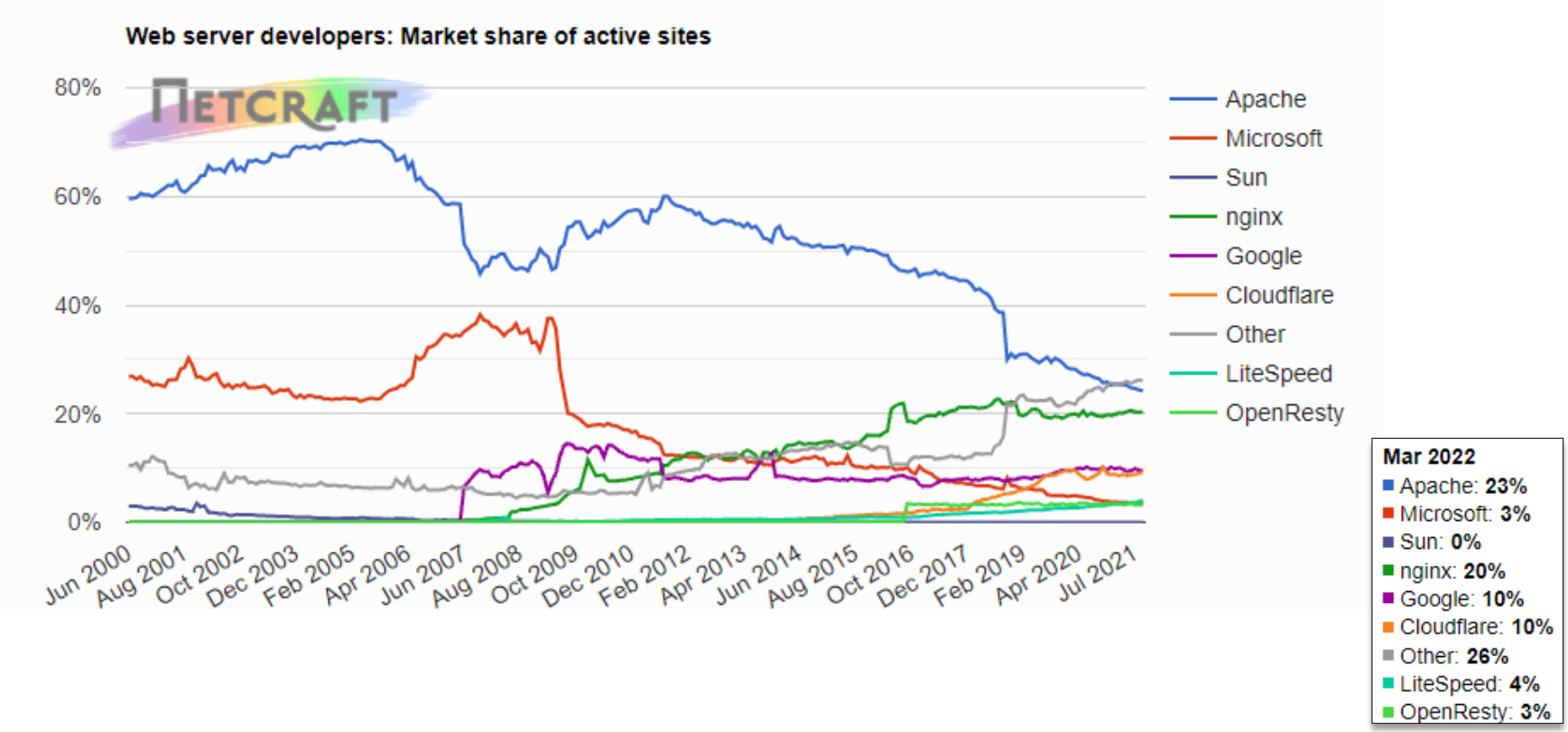
- Web Server
  - Receives client's HTTP/S requests
  - Either serves static content
  - Or forwards request to the application server
- Application Server
  - Interprets the request
  - Prepare the response
  - With data from database or other data sources (other web and application servers)

Technology	Web Server	Application Server
IIS	Yes	Yes (.Net)
Apache	Yes	No
Tomcat	No	Yes (Java)
Nginx	Yes	No
Node.js	No	Yes (Node.js)
…	…	…

Web Server – Market share of computers



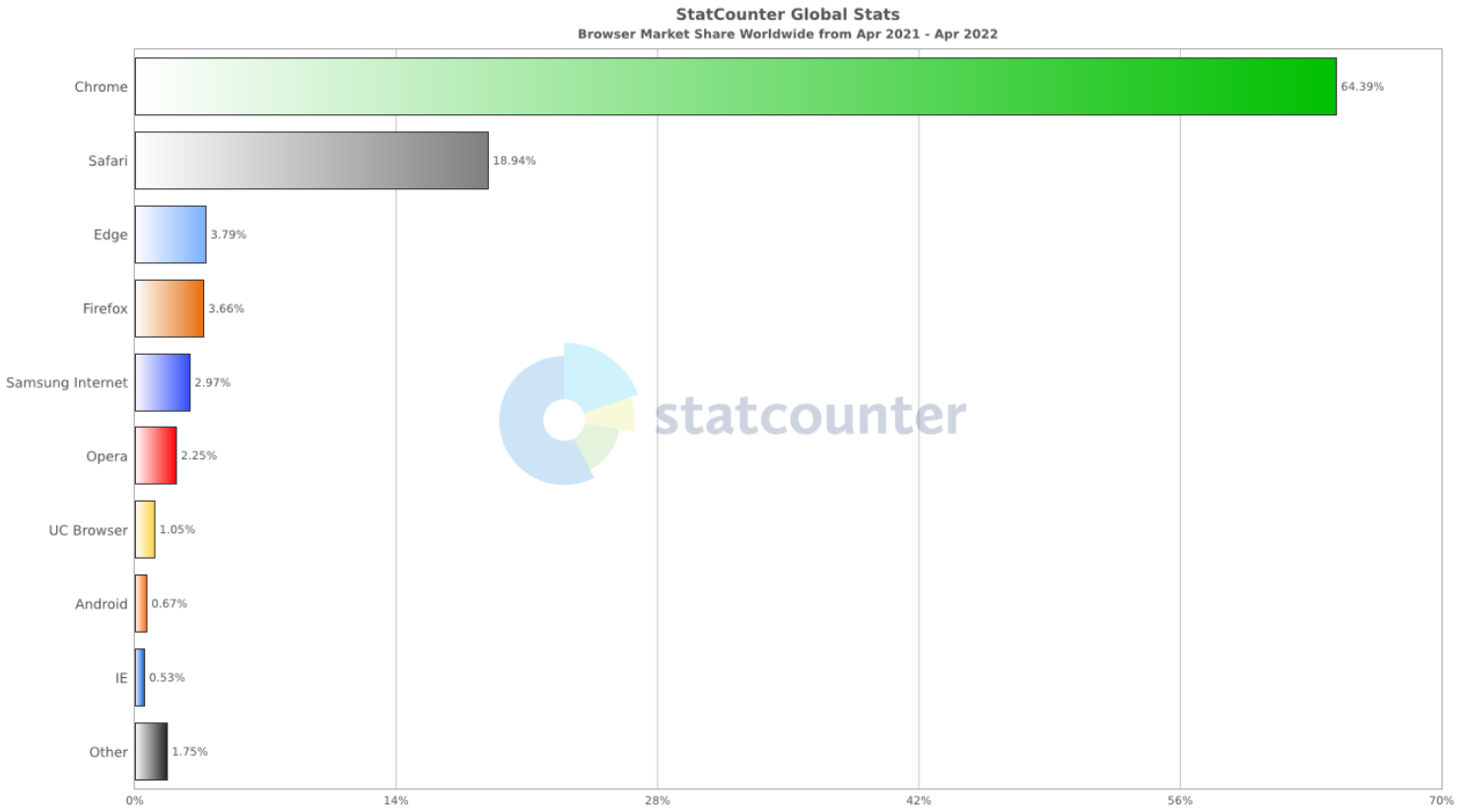
Web Server – Market share of active sites



Web client

- Aplicação informática Web Browser
- Acede e apresenta recursos Web
  - URL
    - Páginas Web
    - Imagens
    - Vídeos
- Processa
  - HTML, CSS, ...
  - XML, JSON, ...
  - SVG ,
  - Linguagens de scripting (e.g. JavaScript)

Web Client – Browser market share



Cf

HTTP/S Protocol

- HTTP – *Hypertext Transfer Protocol*
- W3C specification (*http://www.w3.org/Protocols/)*
- Defines the protocol for transferring data between the web client and the web server
- Most common methods
  - GET
  - POST
  - PUT
  - DELETE
  - INFO
  - OPTIONS

Hypertext Transfer Protocol

- The Hypertext Transfer Protocol (HTTP) is an application layer protocol in the Internet protocol suite model for distributed, collaborative, hypermedia information systems
- HTTP is the foundation of data communication for the World Wide Web
- HTTP development is a coordinated effort by the Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C) W3C (*http://www.w3.org/Protocols/)*
  - 1997 – HTTP/1 (version 1.1)
  - 2015 – HTTP/2
  - 2021 – HTTP/3 (internet draft as of July.2021)

Hypertext Transfer Protocol

- Request methods
  - **GET** - Is used to retrieve information from the given server using a given URI.
  - **HEAD** - Same as GET but transfers the status line and header section only.
  - **POST** - Is used to send data to the server.
  - **PUT** - Replaces all current representations of the target resource with the uploaded content.
  - **DELETE** - Removes all current representations of the target resource given by a URI.
  - **CONNECT** - Establishes a tunnel to the server identified by a given URI.
  - **OPTIONS** - Describes the communication options for the target resource.
  - **TRACE** - Performs a message loop-back test along the path to the target resource.

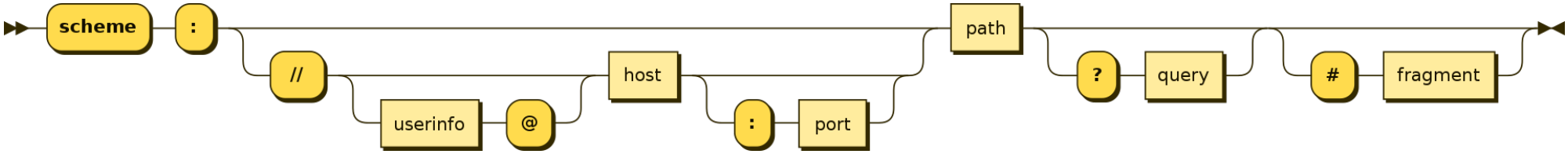
Uniform Resource Location

- A Uniform Resource Locator (URL), colloquially termed a web address, is a reference to a web resource that specifies its location on a network and a mechanism for retrieving it.
- Is a specific type of Uniform Resource Identifier (URI).
- URL's occur most commonly to reference web pages (*http*) but are also used for file transfer (*ftp*), email (*mailto*), database access (*JDBC*), and many other applications.

URL

- URL – *Uniform Resource Location*
- Syntax

protocol://address/path/fileOrResource



- *Examples:*
  - *http://www.dei.isep.ipp.pt/*
  - *ftp://ftp.dei.isep.ipp.pt*

Data serialization

What is JSON?

- Acronym for **J**ava**S**cript **O**bject **N**otation
- Is a lightweight data-interchange format
- Is self-describing and easy to understand
- Is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, (C, C++, C#, Java, JavaScript, Python, …)

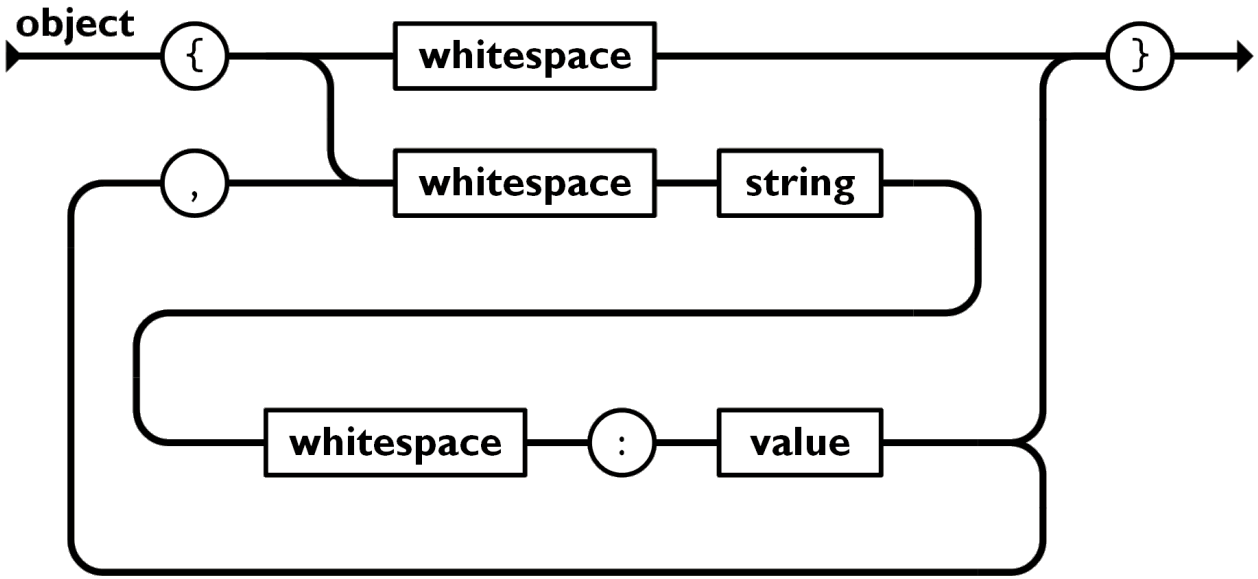
These properties make JSON an ideal data-interchange language

What is JSON?

- JSON is built on two structures:
  - A collection of name/value pairs.
  - An ordered list of values.

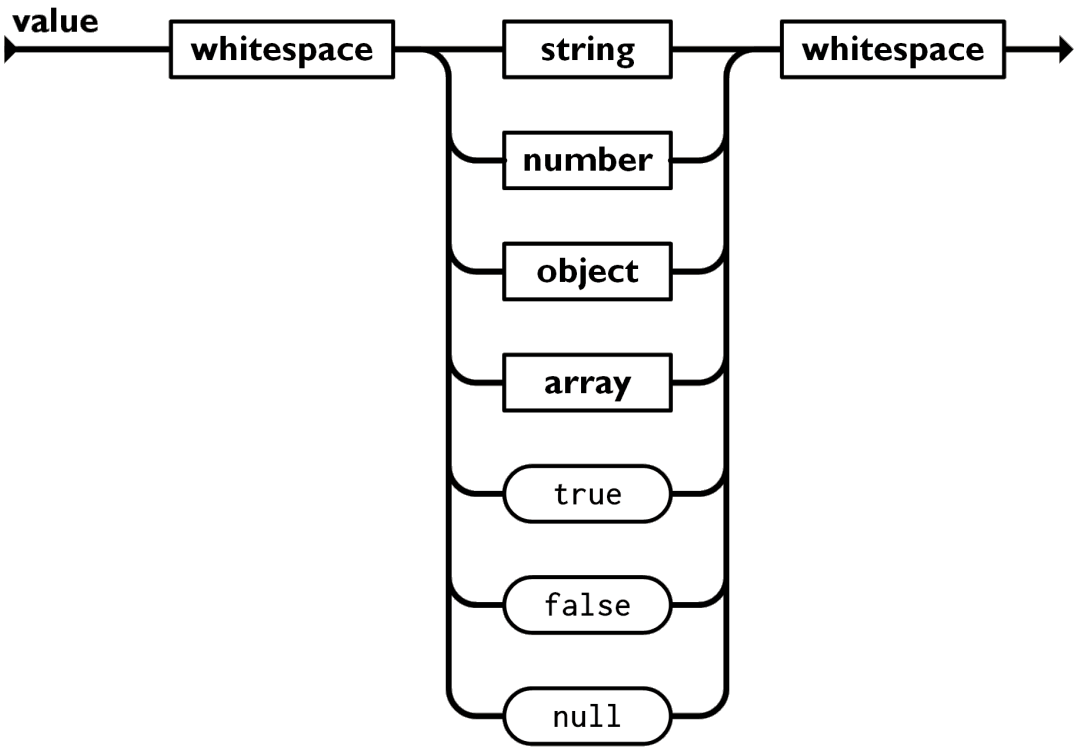
- These are universal data structures.
  - Virtually all modern programming languages support them in one form or another.
  - It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

JSON - object



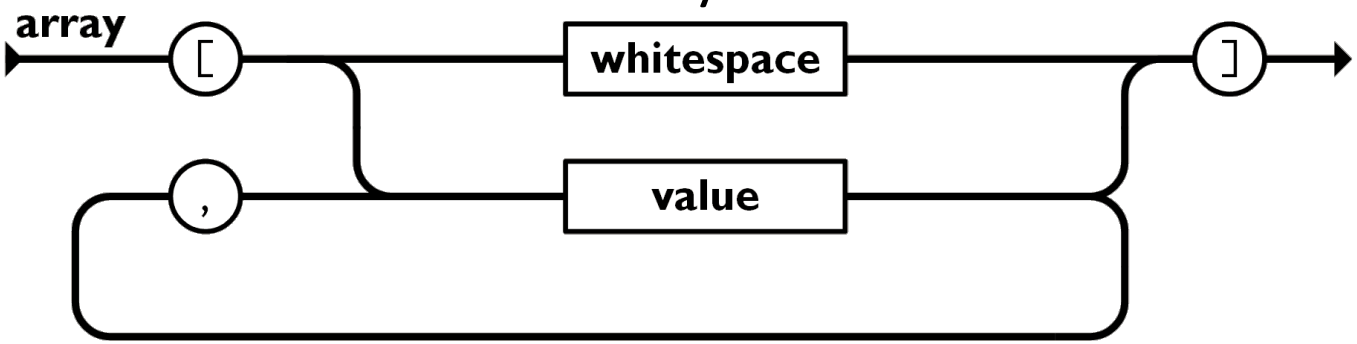

```
{
    "name": "John",
    "age": 31,
    "city": "New York"
}
```

JSON - value



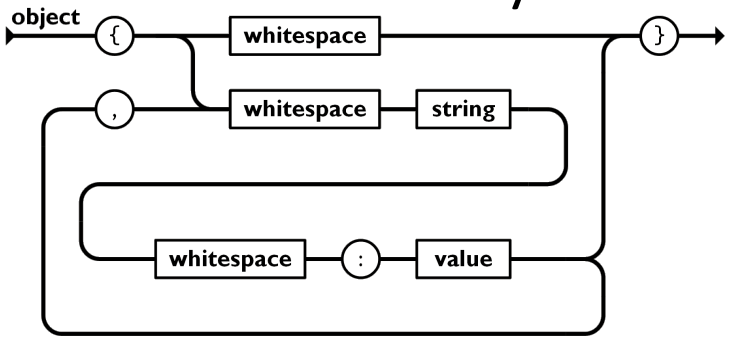

```
{
   "name": "John",
   "age": 31,
   "city": "New York",
   "canDrive":true
}
```

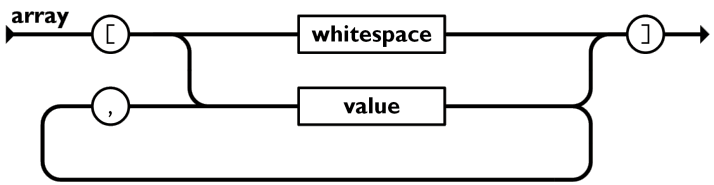
JSON - array




```
{
   "name": "John",
   "age": 31,
   "city": "New York",
   "canDrive":true,
   "cars":[
           "Ford",
           "BMW",
           "Fiat"
          ]
}
```

JSON - array






```
{
  "name":"John",
  "age":31,
  "city": "New York",
  "canDrive":true,
  "cars": [
    { "name":"Ford", "models":[ "Fiesta", "Focus", "Mustang" ] },
    { "name":"BMW", "models":[ "320", "X3", "X5" ] },
    { "name":"Fiat", "models":[ "500", "Panda" ] }
  ]
}
```

Historic evolution

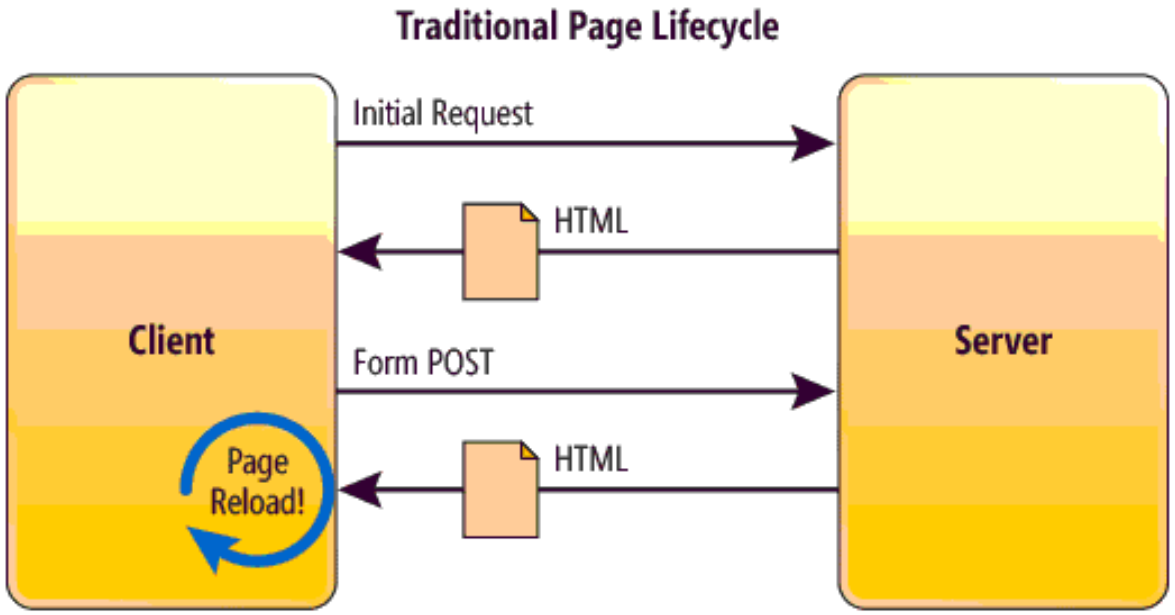
Static Web

- The web server serves the web client with the static content stored in the device
- The information and the format of the presented web page are static unless the developer explicitly changes the files

Dynamic content

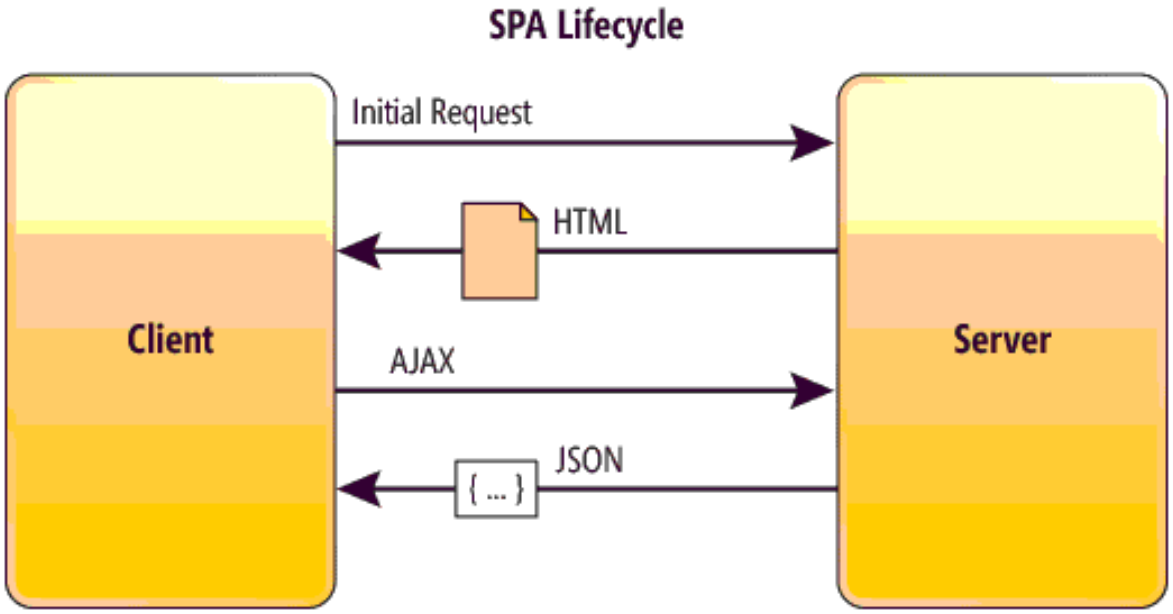
- Server-side rendering/Traditional scenario
- Client-side rendering/ SPA scenario

Dynamic web pages – Traditional scenario



cf: https://moz.com/blog/optimizing-angularjs-single-page-applications-googlebot-crawlers

Dynamic web pages – SPA scenario



cf: https://moz.com/blog/optimizing-angularjs-single-page-applications-googlebot-crawlers

Server-side rendering

- The application server renders the page(s) in runtime and sends it (as if it were static pages) to the requesting client
- The page content is often based on data from databases and other data sources (e.g. other web servers)
- The format and content depends on parameters sent on client's request

CGI-generated dynamic pages

- CGI – *Common Gateway Interface*
- The CGI application is triggered by the client's request and renders the page
- The CGI application is implemented in any programming language that access STDIN and STDOUT, namely:
    - Perl
    - TCL
    - C

HTML embedded code

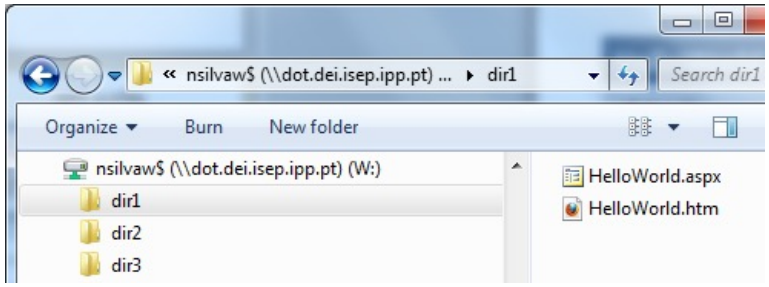
- The content generating code is embedded in the static content, usually HTML+CSS

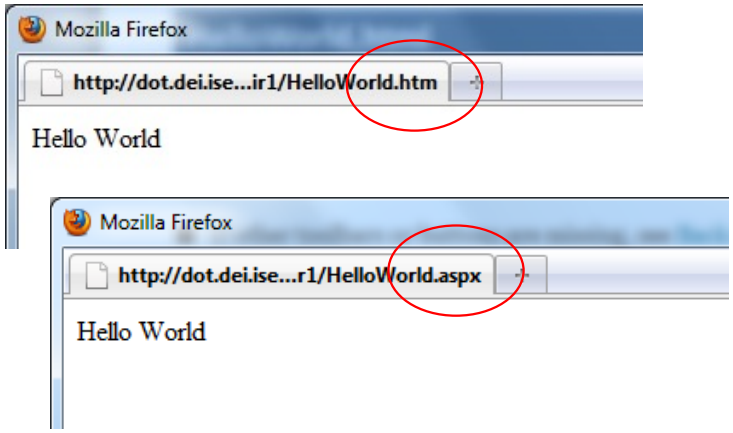
- Common languages/technology:
  - ASPX
  - PHP
  - Python
  - JSP

ASPX is HTML

HelloWorld.htm	HelloWorld.aspx

<html>
  <head>
  <title></title>
  </head>
  <body>
  Hello World
  </body>
</html>
	<html>
	  <head>
	  <title></title>
	  </head>
	  <body>
	  Hello World
	  </body>
	</html>





ASPX embebed code

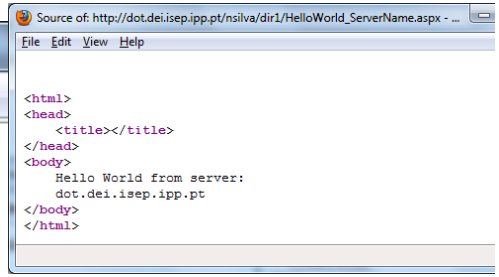
HelloWorld_DateTime.aspx

```
<%@ Page Language="C#" %>

<html>
<head>
   <title></title>
</head>
<body>
   Hello World from server:
   <% Response.Write(Request.ServerVariables.Get("server_name") ); %>
</body>
</html>
```

Directive for the server: programming language (e.g. VB, C#)

ASPX code in the specified language (e.g. VB, C#)

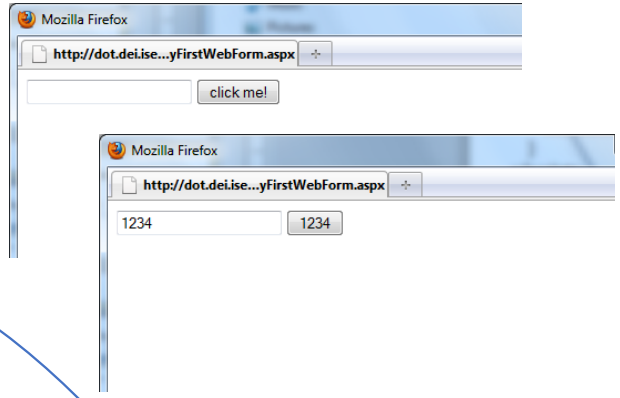
Start/end tag of ASPX code



ASPX.Net events

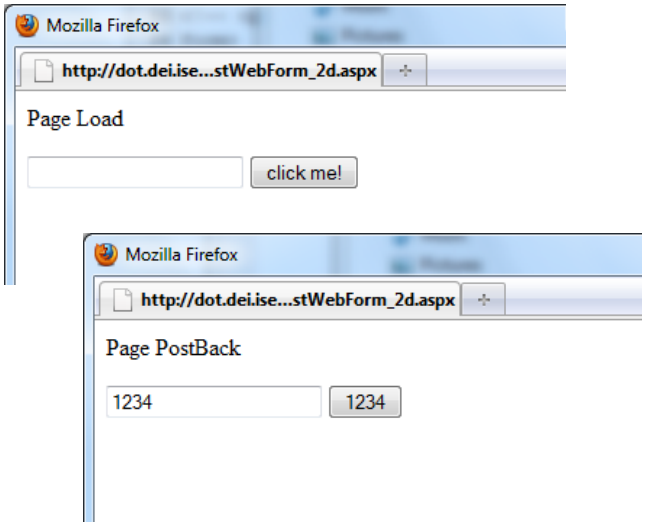
```
<%@ Page Language="C#" %>
<script runat="server">
   void Button1_Click(Object s, EventArgs e)
   {
      Button1.Text = TextBox1.Text;
   }
</script>
<html>
   <head>
      <title></title>
   </head>
   <body>
      <form runat="server">
         <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
         <p><asp:Label ID="Label1" runat="server"></asp:Label>
         <asp:Button id="Button1" Text="click me!" runat="server" OnClick="Button1_Click"/>
      </form>
   </body>
</html>
```

One per .aspx file



Load vs. PostBack

```
<%@ Page Language="C#" %>
<script runat="server">
   void Page_Load(object sender, EventArgs e)
   {
   if( Page.IsPostBack )
         Label1.Text = "Page PostBack";
      else
         Label1.Text = "Page Load";
   }
   void Button1_Click(Object s, EventArgs e)
   {
      Button1.Text = TextBox1.Text;
   }
</script>
<html>
<head>  <title></title> </head>
<body>
   <form runat="server">
        <asp:Label ID="Label1" runat="server"></asp:Label><p>
        <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
        <asp:Button id="Button1" Text="click me!" runat="server" OnClick="Button1_Click"/>
   </form>
</body>
</html>
```



Separação de HTML de Código

MyFirstWebForm_WithCodeFile.aspx

```
<%@ Language="C#" CodeFile="MyFirstWebForm_WithCodeFile.aspx.cs" Inherits="_Default" %>
<html>
<head>  <title></title> </head>
<body>
   <form runat="server">
     <asp:Label ID="Label1" runat="server"></asp:Label><p>
     <asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
     <asp:Button id="Button1" Text="click me!" runat="server" OnClick="Button1_Click"/>
   </form>
</body>
</html>
```

MyFirstWebForm_WithCodeFile.aspx.cs

```
using System;
public partial class _Default : System.Web.UI.Page {
   protected void Page_Load(object sender, EventArgs e) {
     if( Page.IsPostBack )
       Label1.Text = "Page PostBack";
     else
       Label1.Text = "Page Load";
   }
protected void Button1_Click(object sender, EventArgs e)  {
       Button1.Text = TextBox1.Text;
   }
}
```

Client-side rendering

- Page (contente) is rendered in the client (browser) based on:
  - Static pages
  - Embedded scripting code
  - Data requested to the web server/application

Credits

- Documents prepared in previous years by:
  - Miguel Losa
  - Paulo Proença
  - Constantino Martins
  - Paulo Sousa
  - Paulo Maio
  - Nuno Silva
- This document has been designed using resources from Flaticon.com

Something about REST

REST: REpresentational State Transfer

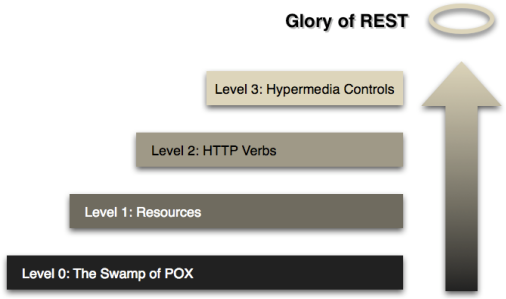
Intro

- Proposed in Roy Fielding's PhD thesis (2000)

- REST is an architectural style/pattern
  - not a specification
  - REST/HTTP is the most common instantiation

- 2 main ideas/assumptions
  - Everything is a resource
  - Each resource has a uniform interface

Key principles



1. Provide multiple representations (e.g. XML, JSON, XHTML, …)

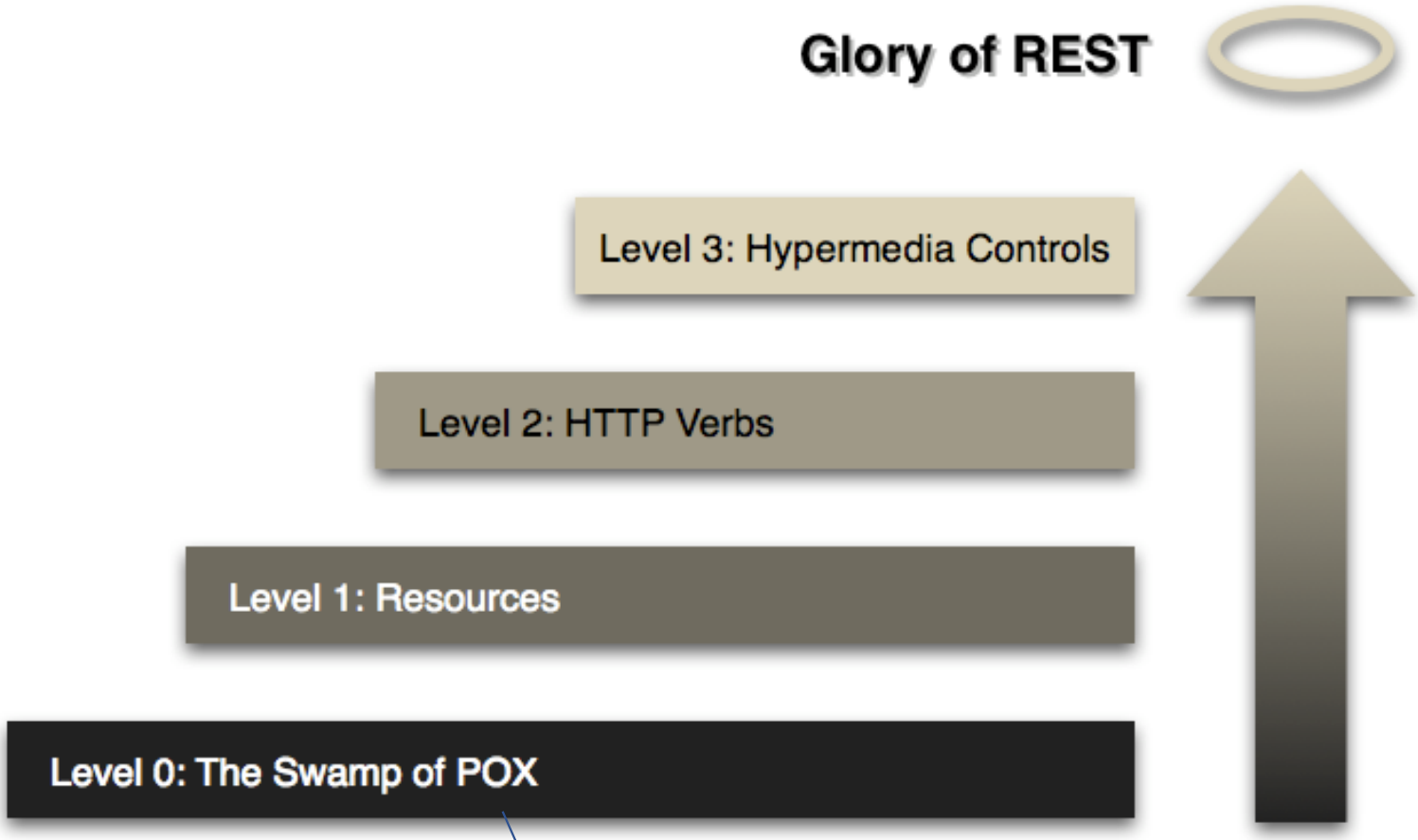
2. Give every “thing” and ID (Resources ID)

3. Use standard methods (e.g. HTTP methods: GET, POST, PUT, …)

   + Communicate statelessly (No server session)

4. Link things together (HATEOAS)

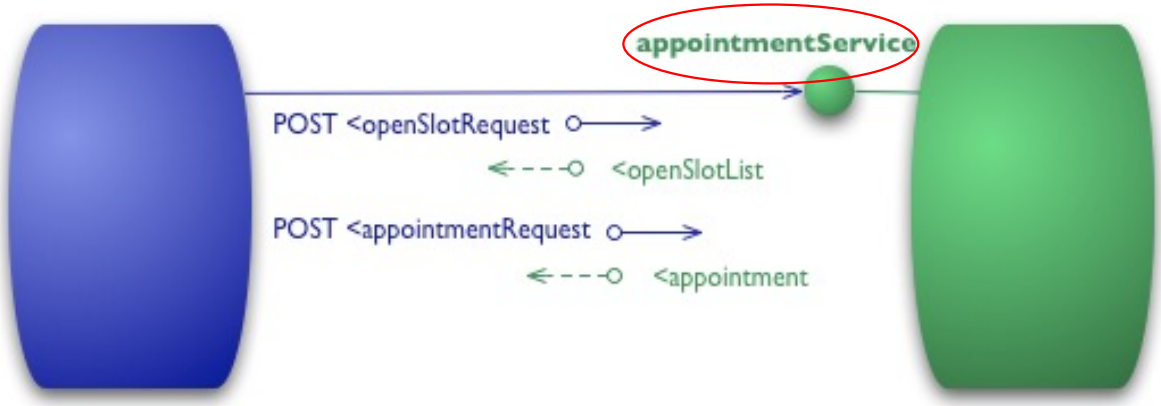
Richardson Maturity Model



Only when the three level are correctly and fully adopted in an API, one can call it REST API (cf. https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven).

Plain Old XML

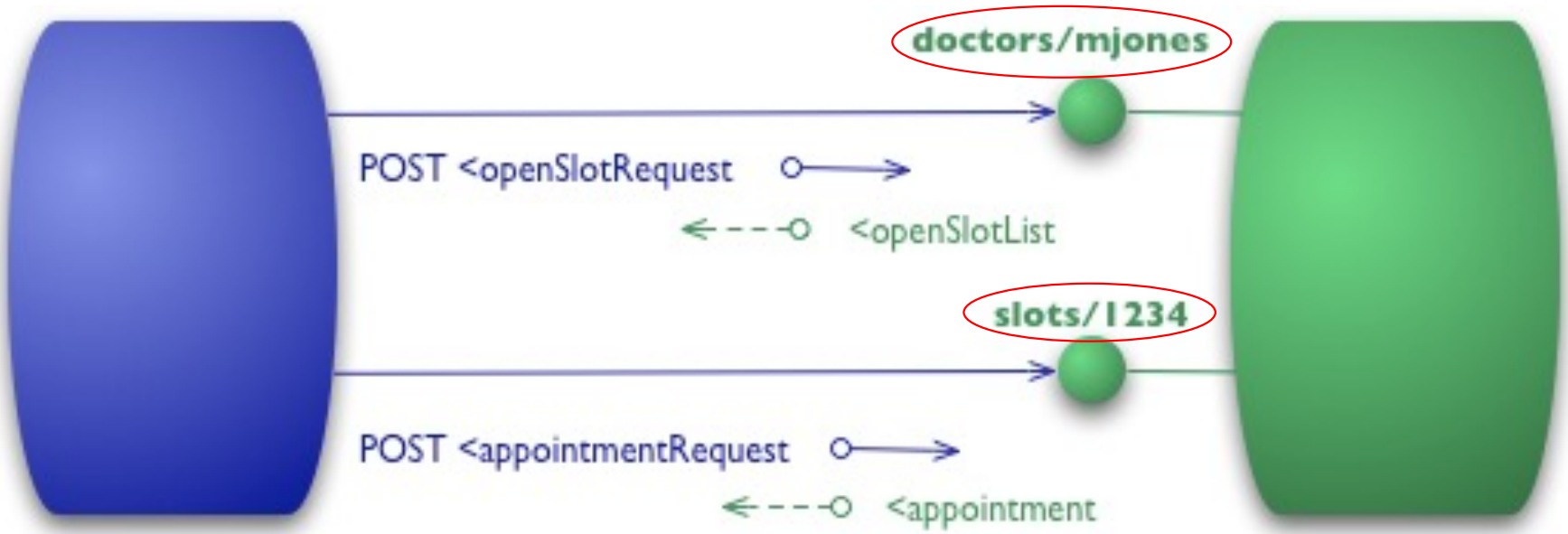
Level 0: Swamp of POX
or RPC - Remote Procedure Call



- HTTP as transport system for remote interactions
- Without using any of the mechanisms of the web
- Usually based on Remote Procedure Invocation/Call

Level 1: Resources

Talk to individual resources, as identified.



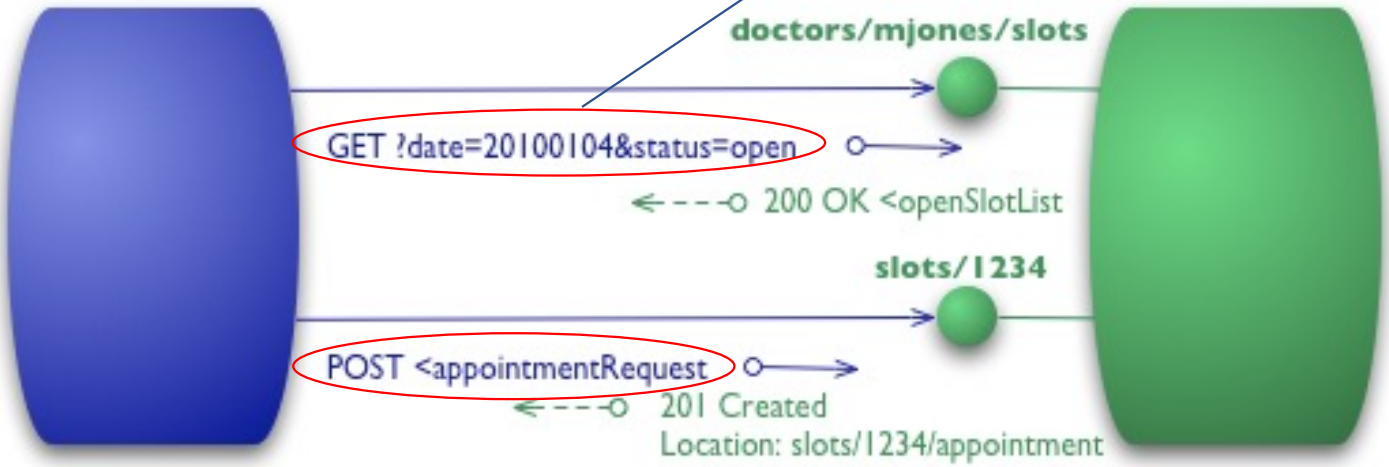
Resource Archetypes

- Document/Element
  - Akin to the object instance or database record
  - Typically includes fields with respective values and links to other resources
  - Base archetype of other resource archetypes
- Collection
  - Server-managed directory of resources
  - Is up to the collection to create+add the client-proposed new resources
- Store
  - Client-managed resource repository (not to confuse with the DDD concept)
  - The client put resources in and get them back out, and decide when to delete them
  - Resources ID are created by the client and not by the store
- Controller
  - Represents/models a procedural concept/execution function
  - With with parameters and return values; inputs and outputs (e.g. POST /alerts/245743/resend)

from: REST API Design Rulebook; Mark Massé; O'Reily; 2011

Level 2: Resource + Methods

Adopted according to uniform interface and semantics.



Uniform interface

Not safe if changes occur in the state of the resource.

N>0 identical requests have the same side-effects as N=1

Method	Description/Semantics	Safe	Idempotent
GET	Request a specific representation of a resource.	Yes	Yes
POST	Submit data to be processed by the identified resource.	No	No (*)
PUT	Create or update (substitute) a resource with the supplied representation.	No	Yes
PATCH	Partially update a resource with the supplied specification/instructions.	No	No (+)
DELETE	Deletes the specified resource.	No	Yes
HEAD	Similar to GET but only retrieves headers and not the body.	Yes	Yes
OPTIONS	Returns the methods supported by the identified resource	Yes	Yes

(*) POSTing to the same resource can result in the creation of multiple and different resources.
(+) PATCHing specifications/instructions are very flexible, so identical request may lead to different outcomes.

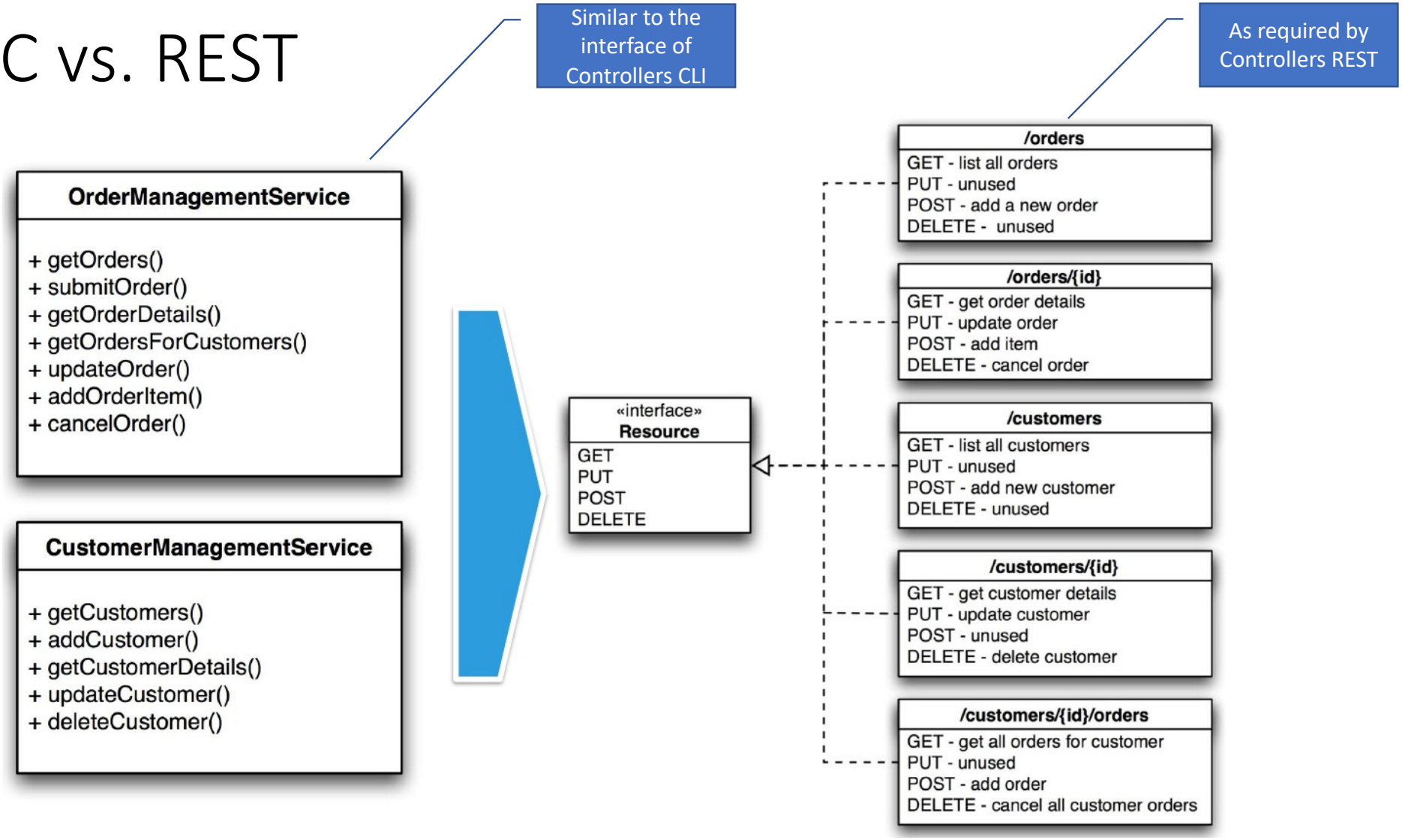
https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html + https://tools.ietf.org/html/rfc5789

Semantics of methods on collection and element

This is an external representation of information (for the clients), not a reflex of the server internal representation/storage.

Method	Collection URI (e.g. http://ex.com/customers/)	Element URI (e.g. http://ex.com/customers/12)
GET	List the URI and perhaps other details of the collection's members.	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type (e.g. JSON)
PUT	Replace the entire collection with another collection.	Update the addressed member of the collection, or if it doesn't exist, create it.
POST	Create a new entry in the collection. The new entry's URL is assigned automatically and is usually returned by the operation.	Treat the addressed member as a collection in its own right and create a new entry in it.
DELETE	Delete the entire collection.	Delete the addressed member of the collection.
PATCH	Update parts of the collection according to the supplied specification.	Partially update the addressed member of the collection with the supplied specification.

RPC vs. REST


Similar to the interface of Controllers CLI
As required by Controllers REST
OrderManagementService
+ getOrders()
+ submitOrder()
+ getOrderDetails()
+ getOrdersForCustomers()
+ updateOrder()
+ addOrderItem()
+ cancelOrder()
CustomerManagementService
+ getCustomers()
+ addCustomer()
+ getCustomerDetails()
+ updateCustomer()
+ deleteCustomer()
«interface»
Resource
GET
PUT
POST
DELETE
/orders
GET - list all orders
PUT - unused
POST - add a new order
DELETE - unused
/orders/{id}
GET - get order details
PUT - update order
POST - add item
DELETE - cancel order
/customers
GET - list all customers
PUT - unused
POST - add new customer
DELETE - unused
/customers/{id}
GET - get customer details
PUT - update customer
POST - unused
DELETE - delete customer
/customers/{id}/orders
GET - get all orders for customer
PUT - unused
POST - add order
DELETE - cancel all customer orders

URI path design rules

- CRUD function names should not be used in URIs
  - DELETE /users/12 (is preferred over) GET /deleteUser/12 (or) DELETE /deleteUser/12
- A plural noun should be used for collection names
  - http://api.soccer.restapi.org/leagues/seattle/teams/trebuchet/players
- A singular noun should be used for document names
  - http://api.soccer.restapi.org/leagues/seattle/teams/trebuchet/players/claudio
- A plural noun should be used for store names
  - http://api.music.restapi.org/artists/mikemassedotcom/playlists
- A verb or verb phrase should be used for controller names
  - http://api.college.restapi.org/students/morgan/register
- Variable path segments may be substituted with identity-based values
  - http://api.soccer.restapi.org/leagues/{leagueId}/teams/{teamId}/players/{playerId}

from: REST API Design Rulebook; Mark Massé; O'Reily; 2011

URI query design rules

- The query component of a URI may be used to filter collections or stores
  - GET /users can be complemented with GET /users/?role=admin
- The query component of a URI should be used to paginate collection or store results
  - GET /users?pageSize=25&pageStartIndex=50
  - POST /users/search, where the body includes more complex pagination info

from: REST API Design Rulebook; Mark Massé; O'Reily; 2011

REST is not CRUD

- HTTP methods/verbs do not map 1:1 to CRUD
  - PUT/POST ←→ Create
  - GET ←→ Read
  - PUT/POST/PATCH ←→ Update
  - DELETE ←→ Delete

Example/Exercises

US	Description	Method	Resource/URL	Input body	Output body	Status
US001	Como gestor de Sistema quero saber se determinada pessoa é irmão/irmã de outra.	GET	/pessoas/{id}/irmaos/{id2}	n/a	{“isIrmao”:true/false}	200/422
US002.1	Como utilizador, quero criar grupo, tornando-me administrador de grupo.	POST/PUT	/grupos/	{“desc”:”adsf”, “idPessoaAdm”:”??”, ...}	{“_self”:”URI”} e.g. URI: grupos/adsf	201/422
			/pessoas/{id}/grupos/	{“desc”:”adsf”, ~~“idPessoaAdm”:”??”~~, ...}		201/422
US003	Como gestor de sistema, quero acrescentar pessoas ao grupo.	POST (!) PATCH (?) PUT (?)	/grupos/{id}/membros/{id2}		{“_self”: “/grupos/{id}/membros”, {“sucess”:[id2, id4]}, {“insucess”:[id3]}}	201/422
			/grupos/{id}/membros/	[id2, id3, id4, ...]		
US004	Como gestor quero saber quais os grupos que são família, [...]	GET				

Level 3: Hypermedia/Hypertext

HATEOAS: Hypermedia As The Engine Of Application State

Introduction

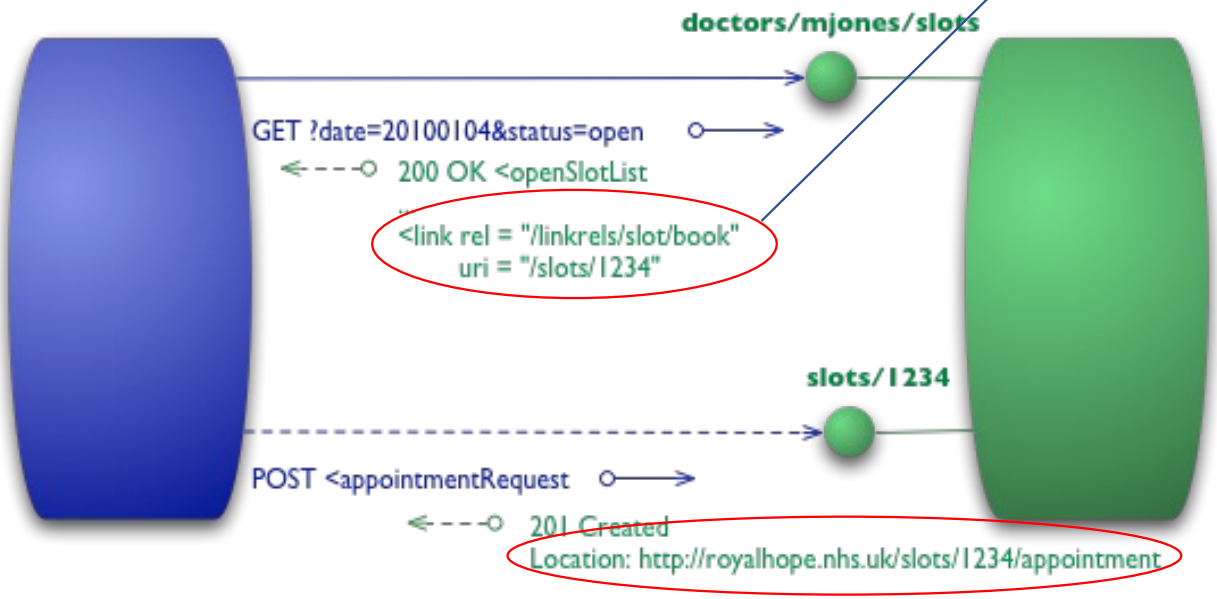
- The fundamental idea of hypermedia is to enrich the representation of a resource with hypermedia elements.
- The simplest form of that are links (hypertext).
- They indicate a client that it can navigate to a certain resource.
- The semantics of a related resource are defined in a so-called link relation.

```html
<link href="theme.css" rel="stylesheet" type="text/css" />
```

```json
{
    "_links": {
        "self": {
            "href": "http://ex.com/customers/1234"
        }
    }
}
```

Resources + Methods + Hypermedia

The principle implies that the API should guide the client through the application, by returning relevant information about the next potential steps, along with each response.



Discoverability

- Discoverability of an API is a topic that doesn't get enough well-deserved attention. As a consequence, very few APIs get it right. It's also something that, if done correctly, can make the API not only RESTful and usable but also elegant.
- HATEOAS **is about full discoverability of actions/transitions on a Resource from Hypermedia (Hypertext, really), as the only driver of application state.**
- If the interaction is to be driven by the API through the conversation itself, concretely via Hypertext, then there can be no documentation. That would coerce the client to make assumptions that are in fact outside of the context of the API.
- In conclusion, **the server should be descriptive enough to instruct the client how to use the API** via Hypertext only. In the case of an HTTP conversation, we could achieve this through the *Link* header.

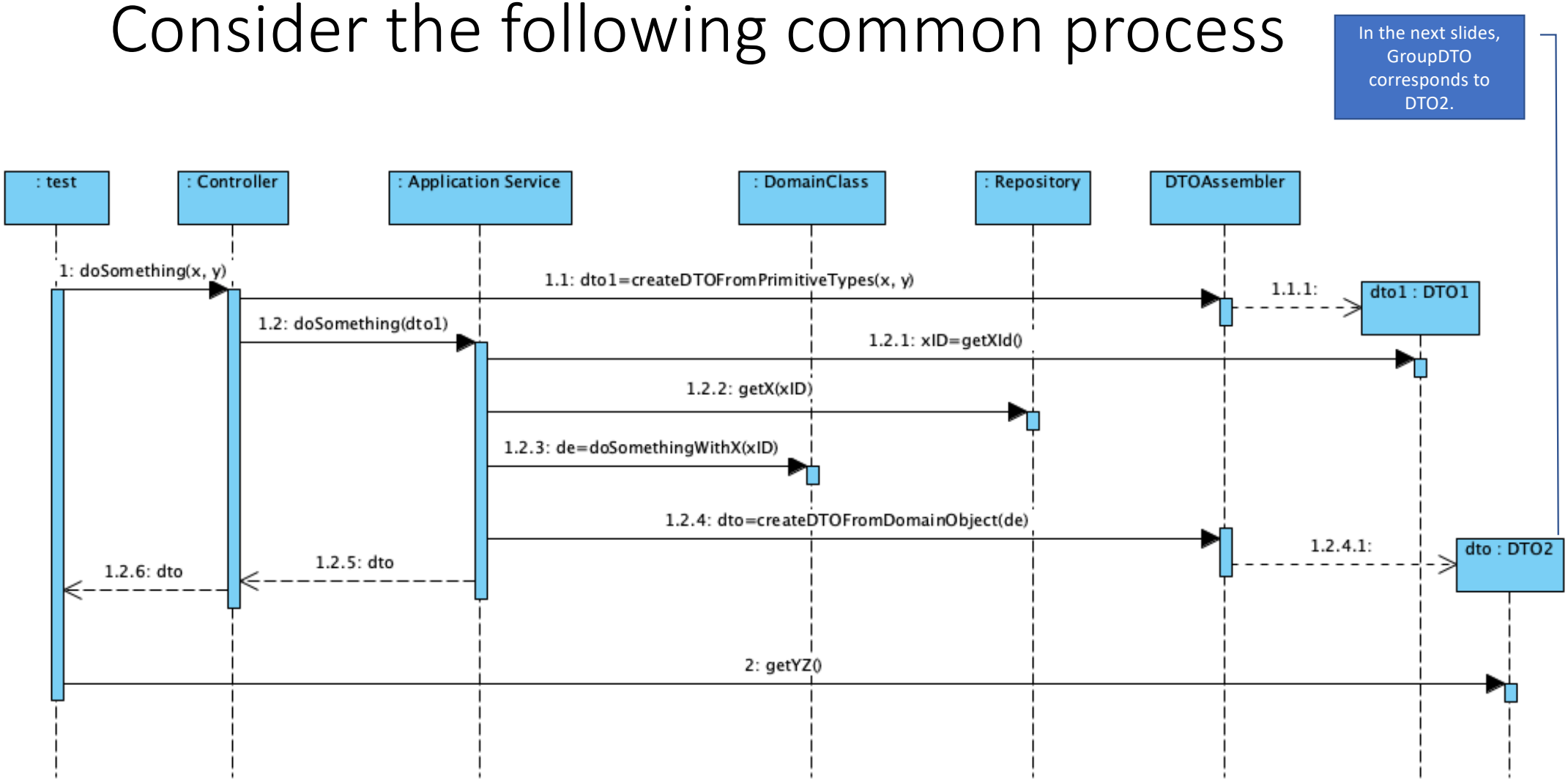
https://www.baeldung.com/restful-web-service-discoverability

HATEOAS in/with Spring Boot

- Spring HATEOAS is a library of APIs to support HATEOAS

```
//add to pom.xml
<dependencies>
//...
      <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-hateoas</artifactId>
            <!-- <version>2.1.4.RELEASE</version> -->
      </dependency>
//...
<dependencies>
```

Consider the following common process



Extend the DTO classes

Extends GroupDTO with methods and attributes to deal with hypermedia links (e.g. groupDTO.add(link1) so they are serialized back to the client.

```
//...
import org.springframework.hateoas.RepresentationModel;

public class GroupDTO extends RepresentationModel<GroupDTO> {
        private String groupDescription;

        public GroupDTO(String groupDescription) {
                this.groupDescription = groupDescription.toUpperCase();
        }
//...
}
```

Generated JSON (example). Cf. next slide.

```
{
    "groupDescription": "BASHTARDS6",
    "_links": {
        "self": {
            "href": "http://localhost:8080/groups/Bashtards6"
        }
    }
}
```

Controller (updated)

```
@RestController
public class US002ControllerREST {
    @Autowired    IUS002_1Service serviceUS002;

    //...

@PostMapping("/groups")
public ResponseEntity<Object> createProduct(@RequestBody NewGroupInfoDTO info) {
     try {
         AdminCreateGroupDTO adminCreateGroupDTO =
                 new AdminCreateGroupDTO(info.getGroupDescription(),info.getPersonName());

         GroupDTO result = serviceUS002.createGroupAndBecomeAdmin(adminCreateGroupDTO);

         Link link = linkTo(methodOn(US002ControllerREST.class)
                                 .getGroupByDesc(info.getGroupDescription()))
                                 .withSelfRel();

         result.add(link);

         return new ResponseEntity<>(result, HttpStatus.CREATED);
     } catch (Exception e) {
         // for example only
         return new ResponseEntity<>(null, HttpStatus.UNPROCESSABLE_ENTITY);
     }
}
```

Correct Method and resource (collection)

Create a link based on the interface of the controller method.
I.e. a URL is generated from a dummy invocation of the method (cf. next slide).

Add the link to the instance of the extended GroupDTO.

No changes in the return. The framework is able to serialize the result parameter if it is a POJO (class with gets and sets).

```java
@GetMapping(value="/groups/{groupDesc}")
public ResponseEntity<Object> getGroupByDesc(@PathVariable final String groupDesc) {
  try {
    GroupDTO result = serviceUS002.getGroupByDesc(groupDesc);
    Link link_to_members = linkTo(US002ControllerREST.class).
        slash("groups").slash(groupDesc).slash("members").withRel("members");
        result.add(link_to_members);
    Link link_to_admins = linkTo(US002ControllerREST.class).
        slash("groups").slash(groupDesc).slash("admins").withRel("admins");
    result.add(link_to_admins);
    Link link_to_ledger = linkTo(US002ControllerREST.class).
        slash("groups").slash(groupDesc).slash("ledger").withRel("ledger");
    result.add(link_to_ledger);
    return new ResponseEntity<>(result, HttpStatus.OK);
  } catch (Exception e) {
    BusinessErrorMessage msg = new BusinessErrorMessage(e.getMessage(),
                              BusinessErrorMessage.GROUP_DESCRIPTION_NOT_FOUND);
    return new ResponseEntity<>(msg, HttpStatus.UNPROCESSABLE_ENTITY);
  }
}
```

Generation of hypermedia links (examples).

Cf. relations IANA

These links are statically built. In comparison with the previous method, here the maintainability is affected.

```json
{
    "groupDescription": "BASHTARDS6",
    "_links": {
        "members": {
            "href": "http://localhost:8080/groups/Bashtards6/members"
        },
        "admins": {
            "href": "http://localhost:8080/groups/Bashtards6/admins"
        },
        "ledger": {
            "href": "http://localhost:8080/groups/Bashtards6/ledger"
        }
    }
}
```

Endpoints not yet implemented, thus the need for static resources links.

Generated JSON (example).

Example/Exercises

US	Description	Method	Resource/URL	Input body	Output body	Hypermedia/ Hypertext	Status
US001	Como gestor de Sistema quero saber se determinada pessoa é irmão/irmã de outra.	GET	/pessoas/{id}/irmaos/{id2}	n/a	{“isIrmao”:true/false}		200/422
US002.1	Como utilizador, quero criar grupo, tornando-me administrador de grupo.	POST/PUT	/grupos/	{“desc”:”adsf”, “idPessoaAdm”:”??”, ...}		{“_self”:”URI”} e.g. URI: grupos/adsf	201/422
			/pessoas/{id}/grupos/	{“desc”:”adsf”, ~~“idPessoaAdm”:”??”~~, ...}			
US003	Como gestor de sistema, quero acrescentar pessoas ao grupo.	POST (!) PATCH (?) PUT (?)	/grupos/{id}/membros/{id2}		{“_self”: “/grupos/{id}/membros”, {“sucess”:[id2, id4]}, {“insucess”:[id3]}}	Cf. previous slides	201/422
			/grupos/{id}/membros/	[id2, id3, id4, ...]			
US004	Como gestor quero saber quais os grupos que são família, [...]	GET					

HTTP Status Codes

HTTP Status Code

When a client makes a request to an HTTP server - and the server successfully receives the request - the server must notify the client if the request was successfully handled or not.

HTTP accomplishes this with five categories of status codes:

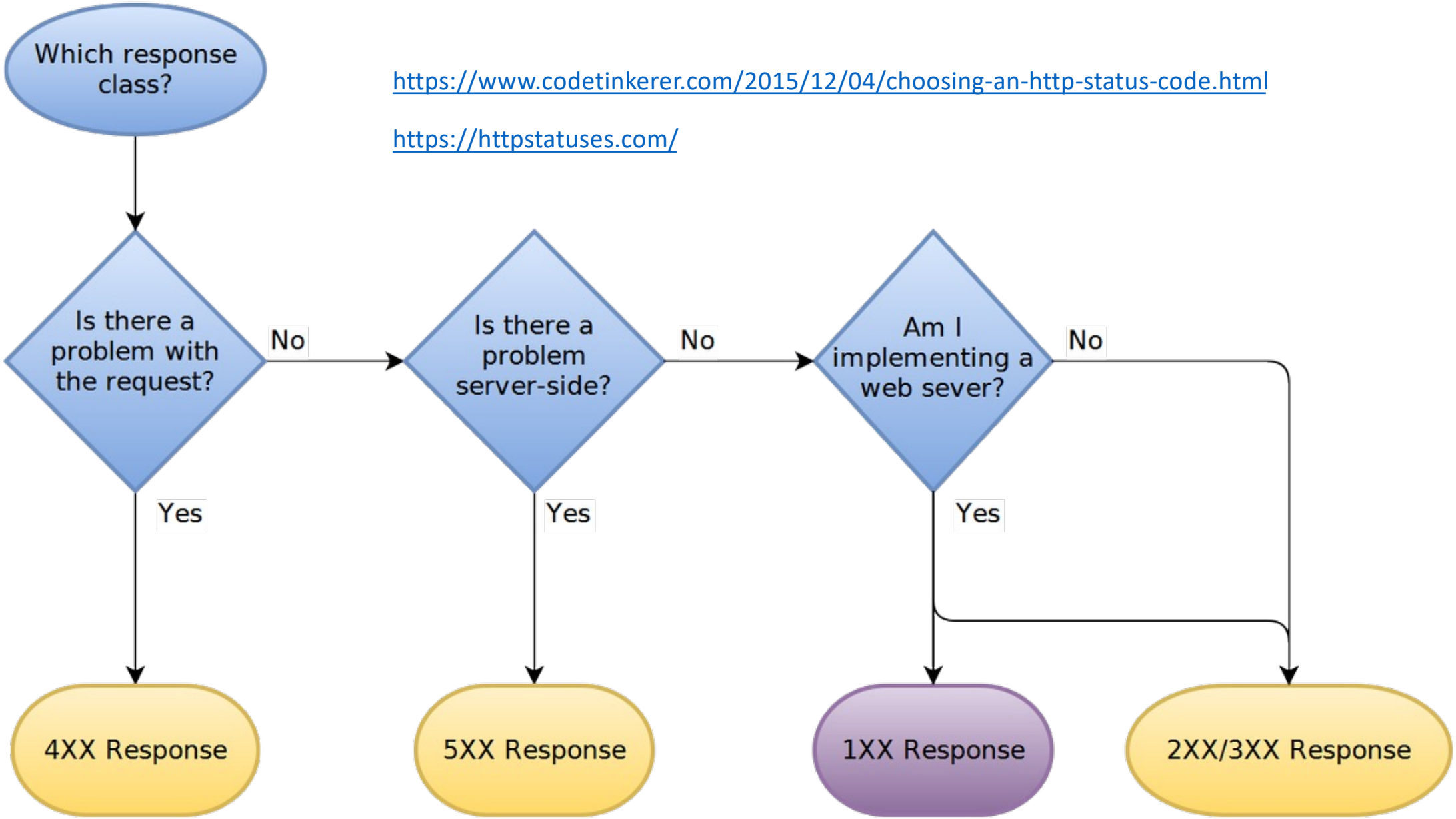
- 100-level (Informational) - Server acknowledges a request
- 200-level (Success) - Server completed the request as expected
- 300-level (Redirection) - Client needs to perform further actions to complete the request
- 400-level (Client error) - Client sent an invalid request
- 500-level (Server error) - Server failed to fulfill a valid request due to an error with server

Based on the response code, a client can surmise the result of a particular request.

from: https://www.baeldung.com/rest-api-error-handling-best-practices

https://www.codetinkerer.com/2015/12/04/choosing-an-http-status-code.html

https://httpstatuses.com/

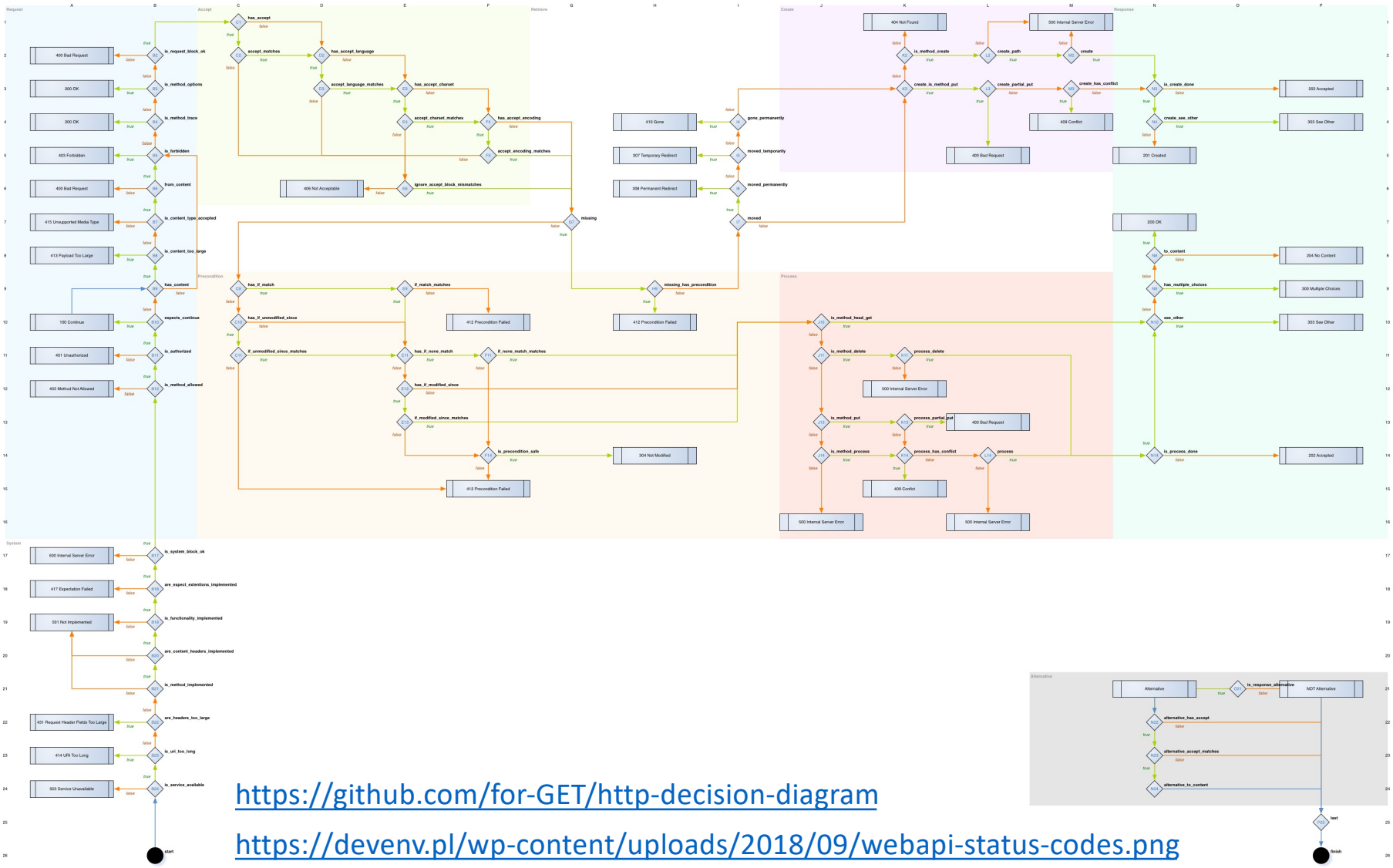


HTTP decision diagram (v4.0.201410)

An activity diagram to describe the resolution of HTTP response status codes, given various methods and headers, implemented via semantical callbacks.

source: https://github.com/for-GET/http-decision-diagram

v1-v3 source: https://code.google.com/p/http-headers-status



https://github.com/for-GET/http-decision-diagram

https://devenv.pl/wp-content/uploads/2018/09/webapi-status-codes.png

References & Bibliography

- https://martinfowler.com/articles/richardsonMaturityModel.html
- https://pt.slideshare.net/pagsousa/rest-web-services-5478036
- https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html
- https://tools.ietf.org/html/rfc5789
- https://docs.spring.io/spring-hateoas/docs/current/reference/html/
- https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
- https://www.baeldung.com/restful-web-service-discoverability
- https://www.baeldung.com/exception-handling-for-rest-with-spring
- https://www.baeldung.com/global-error-handler-in-a-spring-rest-api
- https://www.baeldung.com/rest-api-error-handling-best-practices
- REST API Design Rulebook; Mark Massé; O'Reily; 2011

Bulletproof node.js base project

Bootstrap (1/2)

src/app.ts

```
import config from './config';
import express from 'express';
import Logger from './loaders/logger’;

startServer();

async function startServer() {

   const app = express();

   await require('./loaders').default({ expressApp: app });

   app.listen(config.port, err => {
      if (err) {
          Logger.error(err);
          process.exit(1);
          return;
      }

      Logger.info(‘🛡️Server listening on port:
   ${config.port}🛡️’);
   });
}
```

Default file

Init function (bootstrap)

New Express application (routing engine).

Initialize the application: routes, service/singletons, dbconnection, etc.

Start the http/s server.

Bootstrap (2/2)

src/loaders/index.ts

```
import expressLoader from './express';
import dependencyInjectorLoader from './dependencyInjector';
import mongooseLoader from './mongoose’;
import config from ‘../../config';

export default async ({ expressApp }) => {
    const mongoConnection = await mongooseLoader();

    const roleModel = {
        name: 'roleModel',
        model: require('../persistence/role').default,
    };
    const roleController = {
        name: config.controller.role.name,
        path: config.controller.role.path
    }
    const roleRepo = {
        name: config.repos.role.name,
        path: config.repos.role.path
    }
    const roleService = {
        name: config.services.role.name,
        path: config.services.role.path
    }
    await dependencyInjectorLoader({
        mongoConnection,
        schemas: [roleSchema],
        controllers: [roleController],
        repos: [roleRepo],
        services: [roleService]
    });
    await expressLoader({ app: expressApp });
}
```

Initialize the Db connection (infrastructure layer).

Define the Role data model (framework layer) for insert in DI container.

Define the Role controller (interface-adapter layer) for insert in DI container.

Define the Role repository (interface-adapter layer) for insert in DI container.

Define the Role service (app service layer) for insert in DI container.

Initialize the DI container with the required objects. Cf. next slide.

config.js

```
export default {
    api: { prefix: ”/api },
    controller: {role: {name: "RoleController",path: "../controllers/roleController"}},
    repos: {role: {name: "RoleRepo",path: "../repos/roleRepo"}},
    services: {role: {name: "RoleService",path: "../services/roleService"}},
};
```

Dependency Loader/Injector

src/loaders/dependencyInjector.ts

```
import { Container } from 'typedi';
export default ({ mongoConnection, schemas, controllers, repos, services}: {
         mongoConnection;
         schemas: { name: string; schema: any }[],
         controllers: {name: string; path: string }[],
         repos: {name: string; path: string }[],
         services: {name: string; path: string }[] }) => {
   try {
      schemas.forEach(m => {
         let schema = require(m.schema).default;
         Container.set(m.name, schema);
      });
      repos.forEach(m => {
         let repoClass = require(m.path).default;
         let repoInstance = Container.get(repoClass);
         Container.set(m.name, repoInstance);
      });
      services.forEach(m => {
         let serviceClass = require(m.path).default;
         let serviceInstance = Container.get(serviceClass)
         Container.set(m.name, serviceInstance);
      });
      controllers.forEach(m => {
         let controllerClass = require(m.path).default;
         let controllerInstance = Container.get(controllerClass);
         Container.set(m.name, controllerInstance);
      });
      return;
   } catch (e) {
      LoggerInstance.error('🔥 Error on dependency injector loader: %o', e);
      throw e;
   }
};
```

Load the @Service() class by its path.

Create/get the instance of the @Service() class.

Rename the instance inside the container.

Configure App

src/loaders/express.ts

```
import express from './express';
import bodyParser from 'body-parser';
import cors from 'cors';
import routes from '../api';
import config from '../../config';


export default ({ app }: { app: express.Application }) => {
   app.enable('trust proxy');
   app.use(cors());

   app.use(bodyParser.json());
   app.use(config.api.prefix, routes());

   app.use((req, res, next) => {
      const err = new Error('Not Found');
      err['status'] = 404;
      next(err);
   });

   app.use((err, req, res, next) => {
      if (err.name === 'UnauthorizedError') {
         return res.status(err.status)
         .send({ message: err.message }).end();
      }
      return next(err);
   });

   app.use((err, req, res, next) => {
      res.status(err.status || 500);
      res.json({errors: { message: err.message} });
   });
};
```

Useful if you're behind a reverse proxy (Heroku, Bluemix, AWS ELB, Nginx, etc.)
It shows the real origin IP in the Heroku or Cloudwatch logs

The magic package that prevents frontend developers going nuts
Alternate description: Enable CORS to all origins by default

Middleware that transforms the raw string of req.body into json

Load (business) API routes (iNB: defined in folder ../api)

catch 404 and forward to error handler

error handlers
e.g. Handle 401 thrown by express-jwt library

Routing

src/api/index.ts

```
import { Router } from 'express';
import user from './routes/user';
import role from './routes/role';

export default () => {
    const app = Router();
    role(app);
    return app
}
```

Default file

Default function

src/api/routes/role.ts

```
export default (app: Router) => {
    app.use('/roles', route);

    const ctrl = Container.get(config.controller.role.name) as
              IRoleController;
    route.post(’’,
          celebrate({
              body: Joi.object({
                  name: Joi.string().required()
              })
          }),
          (req, res, next) => ctrl.createRole(req, res, next) );
    route.put(’’,
          celebrate({
              body: Joi.object({
                  id: Joi.string().required(),
                  name: Joi.string().required()
              }),
          }),
          (req, res, next) => ctrl.updateRole(req, res, next) );
};
```

Gets the instance of the controller that will handle the request. The controller is defined by configuration and created&stored by DI (cf. previsou slides)

Address: /api/roles/

Checks the content of the request body.

The callback function: when the router is activated, the createRole function of the controller will be called with the same parameters.

Controller

src/controllers/roleController.ts

```
import { Request, Response, NextFunction } from 'express';
import { Inject } from 'typedi';
import config from "../../config";
import IRoleController from "./IControllers/IRoleController";
import IRoleService from '../services/IServices/IRoleService';
import IRoleDTO from '../dto/IRoleDTO';
import { Result } from "../core/logic/Result";

export default class RoleController implements IRoleController {
   constructor(
      @Inject(config.services.role.name)
                  private roleServiceInstance : IRoleService
   ) {}

   public async createRole(req: Request, res: Response, next: NextFunction) {
     try {
        const roleOrError = await this.roleServiceInstance.createRole(
            req.body as IRoleDTO) as Result<IRoleDTO>;

        if (roleOrError.isFailure) {
            return res.status(402).send();
         }

        const roleDTO = roleOrError.getValue();

        return res.status(201).json( roleDTO );
     }
     catch (e) {
         return next(e);
     }
   };
//...
```

Injecting the service object, applying Dependency Injection, Dependency Inversion Principle, Inversion of Control…

Call the service createRole function with the DTO generated by the default mapper.

The result of the createRole function is itself, a DTO object.

If some error occurs inside the service, generate a response accordingly. Otherwise, return the received DTO.

If some exception occurs, forward the error to the next handler (cf. bootstrap).

```
import { Request, Response, NextFunction } from 'express';

export default interface IRoleController {
    createRole(req: Request, res: Response, next: NextFunction);
    updateRole(req: Request, res: Response, next: NextFunction);
}
```

Service

src/services/roleService.ts

```
import { Service, Inject } from 'typedi';
import config from "../../config";
import IRoleDTO from '../dto/IRoleDTO';
import { Role } from "../domain/role";
import IRoleRepo from '../services/IRepos/IRoleRepo';
import IRoleService from './IServices/IRoleService';
import { Result } from "../core/logic/Result";
import { RoleMap } from "../mappers/RoleMap";

@Service()
export default class RoleService implements IRoleService {
   constructor(
      @Inject(config.repos.role.name) private roleRepo : IRoleRepo
   ) {}
   public async createRole(roleDTO: IRoleDTO): Promise<Result<IRoleDTO>> {
      try {
         const roleOrError = await Role.create( roleDTO );
         if (roleOrError.isFailure) {
            return Result.fail<IRoleDTO>(roleOrError.errorValue());
         }
         const roleResult = roleOrError.getValue();
         await this.roleRepo.save(roleResult);
         const roleDTOResult = RoleMap.toDTO( roleResult ) as IRoleDTO;
         return Result.ok<IRoleDTO>( roleDTOResult )
      }
      catch (e) {
         throw e;
      }
   }
//...
```

src/services/IRoleRepo.ts

```
import { Repo } from "../core/infra/Repo";
import { Role } from "../domain/role";
import { RoleId } from "../domain/roleId";

export interface IRoleRepo extends Repo<Role>
{
   save(role: Role): Promise<Role>;
   findByDomainId (roleId: RoleId | string):
   Promise<Role>;
}
```

The Role Repository interface.

Aggregate Root static creation method

Save the role through the repository.

Return the result of the repository save function.

@Service defines this class as injectable through @inject and through Container.

Repository

src/repos/roleRepo.ts

```
import { Service, Inject } from 'typedi';
import IRoleRepo from "../services/IRepos/IRoleRepo";
import { Role } from "../domain/role";
import { RoleId } from "../domain/roleId";
import { RoleMap } from "../mappers/RoleMap";
import { Document, Model } from 'mongoose';
import { IRolePersistence } from '../dataschema/IRolePersistence';

@Service()
export default class RoleRepo implements IRoleRepo {
  private models: any;
  constructor(
    @Inject('roleSchema') private roleSchema :
                  Model<IRolePersistence & Document>
    ) {}
  private createBaseQuery (): any {
    return {
      where: {},
    }
  }

public async exists (roleId: RoleId | string): Promise<boolean> {
  const idX = roleId instanceof RoleId ?
                  (<RoleId>roleId).id.toValue() : roleId;

  const query = { domainId: idX};
  const roleDocument = await this.roleSchema.findOne( query );

  return !!roleDocument === true;
}
// ...
```

Declared in the App Service Layer

Persistence Layer

```
// ...
public async save (role: Role): Promise<Role> {
  const query = { domainId: role.id.toString()};
  const roleDocument = await this.roleSchema.findOne( query );

  try {
    if (roleDocument === null ) {
      const rawRole: any = RoleMap.toPersistence(role);

      const roleCreated = await this.roleSchema.create(rawRole);

    return RoleMap.toDomain(roleCreated);
    } else {
      roleDocument.name = role.name;
      await roleDocument.save();
      return role;
    }
  } catch (err) {
    throw err;
  }
}
// ...
```

Celebrate Joi validation

joi

- **joi** lets describe data using a simple, intuitive, and readable language

```
const Joi = require('joi');
const schema = Joi.object({
   username: Joi.string()
      .alphanum()
      .min(3)
      .max(30)
      .required(),
   password: Joi.string()
      .pattern(new RegExp('^[a-zA-Z0-9]{3,30}$')),
   repeat_password: Joi.ref('password’),
   access_token: [
      Joi.string(),
      Joi.number()
   ],
   birth_year: Joi.number()
      .integer()
      .min(1900)
      .max(2010),
   email: Joi.string()
      .email({ minDomainSegments: 2, tlds: { allow: ['com', 'net'] } })
})
  .with('username', 'birth_year’)
  .xor('password', 'access_token’)
  .with('password', 'repeat_password');
```

celebrate

- **`celebrate`** is an express middleware function that wraps the `joi` validation library
- This allows to use this middleware in any single route, or globally, and ensure that all inputs are correct before any handler function
- The middleware allows to validate `req.params`, `req.headers`, `req.body`, …

celebrate - example

```
const express = require('express');
const BodyParser = require('body-parser');
const { celebrate, Joi, errors, Segments } = require('celebrate');

const app = express();
app.use(BodyParser.json());

app.post('/signup', celebrate({
  [Segments.BODY]: Joi.object().keys({
     name: Joi.string().required(),
     age: Joi.number().integer(),
     role: Joi.string().default('admin')
  }),
  [Segments.QUERY]: {
     token: Joi.string().token().required()
  }
}), (req, res) => {
   // At this point, req.body has been validated and
   // req.body.role is equal to req.body.role if provided in the POST or set to 'admin' by joi
});
app.use(errors());
```

References and Bibliography

- https://blog.ploeh.dk/2013/12/03/layers-onions-ports-adapters-its-all-the-same/
- https://jeffreypalermo.com/tag/onion-architecture/
- https://stackoverflow.com/questions/23479879/clean-architecture-vs-onion-architecture
- https://herbertograca.com/2017/09/21/onion-architecture/

Something about software testing

Observations about Testing

- “Testing is the process of executing a program with the intention of finding errors.” – Myers
- “Testing can show the presence of bugs but never their absence.” – Dijkstra
- “A quality control activity aimed at evaluating a software item against the given system requirements. This includes, but is not limited to, the process of executing a program or application with the intent of finding software bugs.”

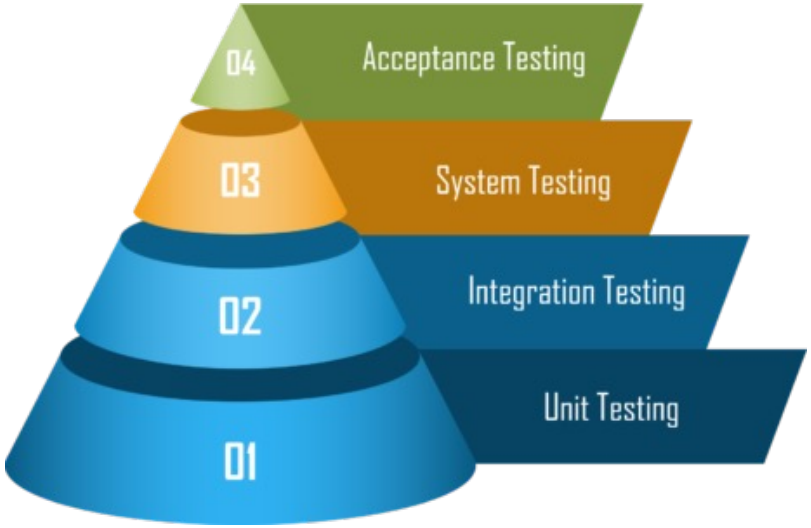
Verification vs. Validation

Criteria	Verification	Validation
Definition	The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase. "Software verification provides objective evidence that the design outputs of a particular phase of the software development life cycle meet all of the specified requirements for that phase" [1]	The process of evaluating software during or at the end of the development process to determine whether it satisfies specified business requirements. "Confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled"[1]
Objective	To ensure that the product is being built according to the requirements and design specifications. In other words, to ensure that work products meet their specified requirements.	To ensure that the product actually meets the user's needs and that the specifications were correct in the first place. In other words, to demonstrate that the product fulfils its intended use when placed in its intended environment.
Question	Are we building the product *right*?	Are we building the *right* product?
Evaluation Items	Plans, Requirement Specs, Design Specs, Code, Test Cases	The actual product/software.

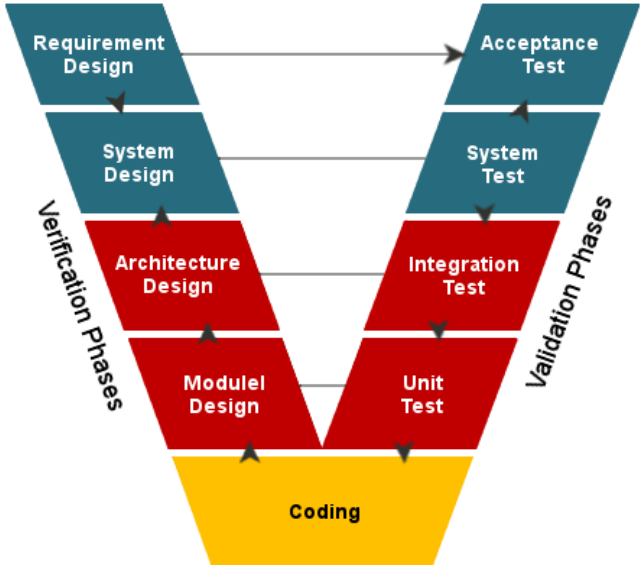
[1] U.S. Food and Drug Administration. General Principles of Software Validation; Final Guidance for Industry and FDA Staff; pp. 47, 2002.

Levels of Testing

- Unit Testing
- Integration Testing
- System Testing
  - Alpha Testing
  - Beta Testing
- Acceptance Testing

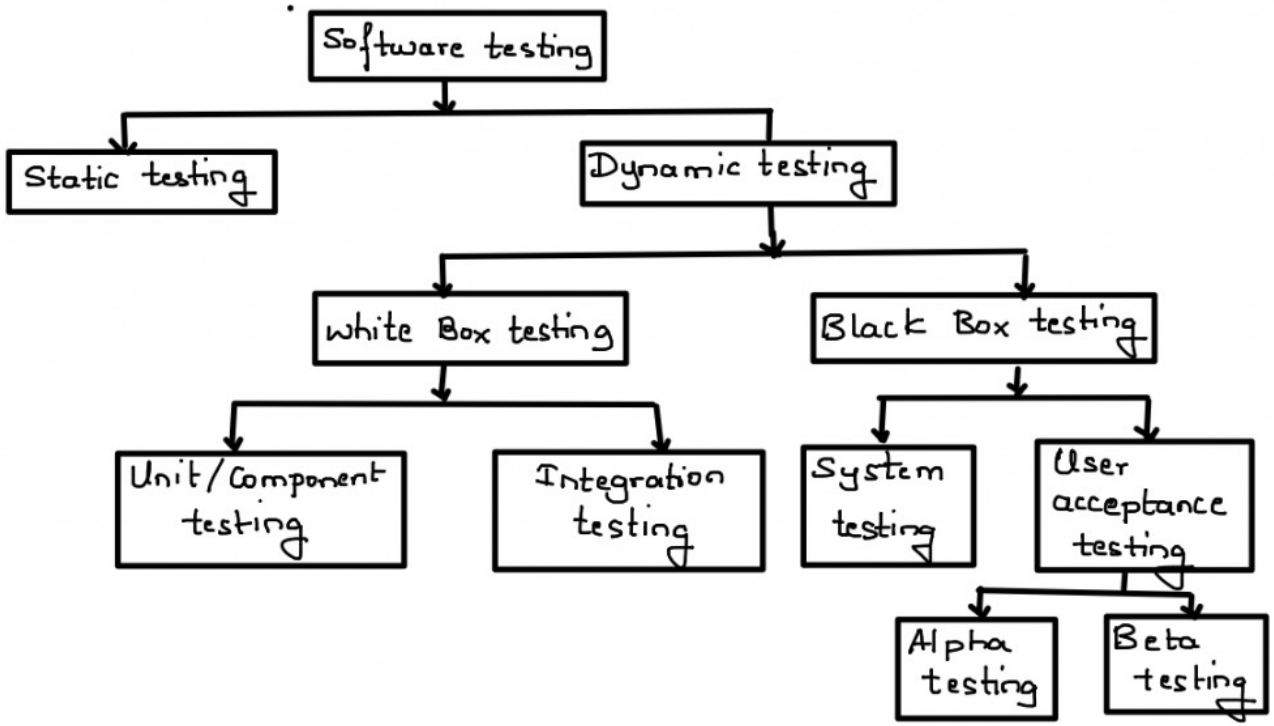


https://www.edureka.co/blog/software-testing-levels/

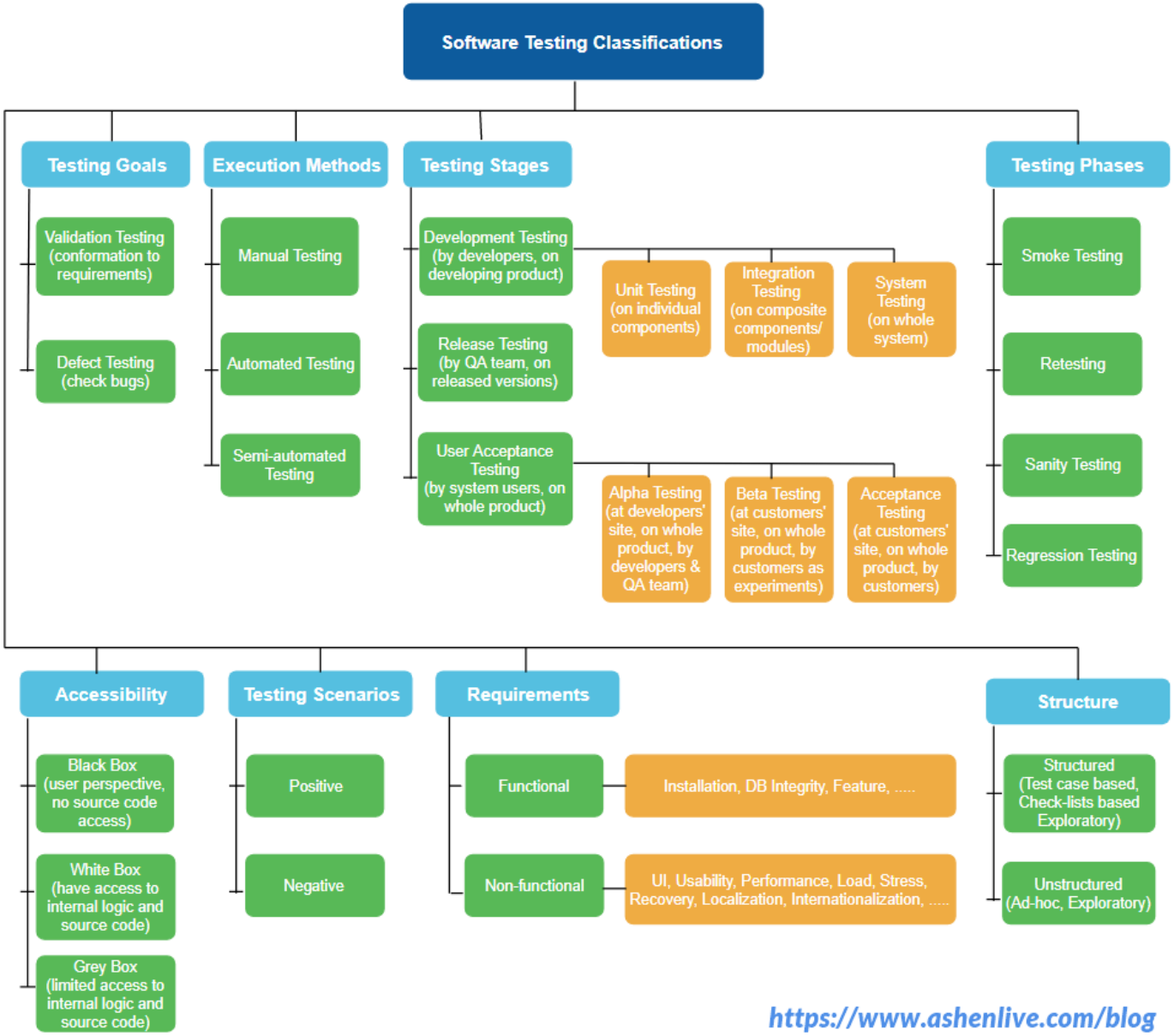


http://www.professionalqa.com/v-model

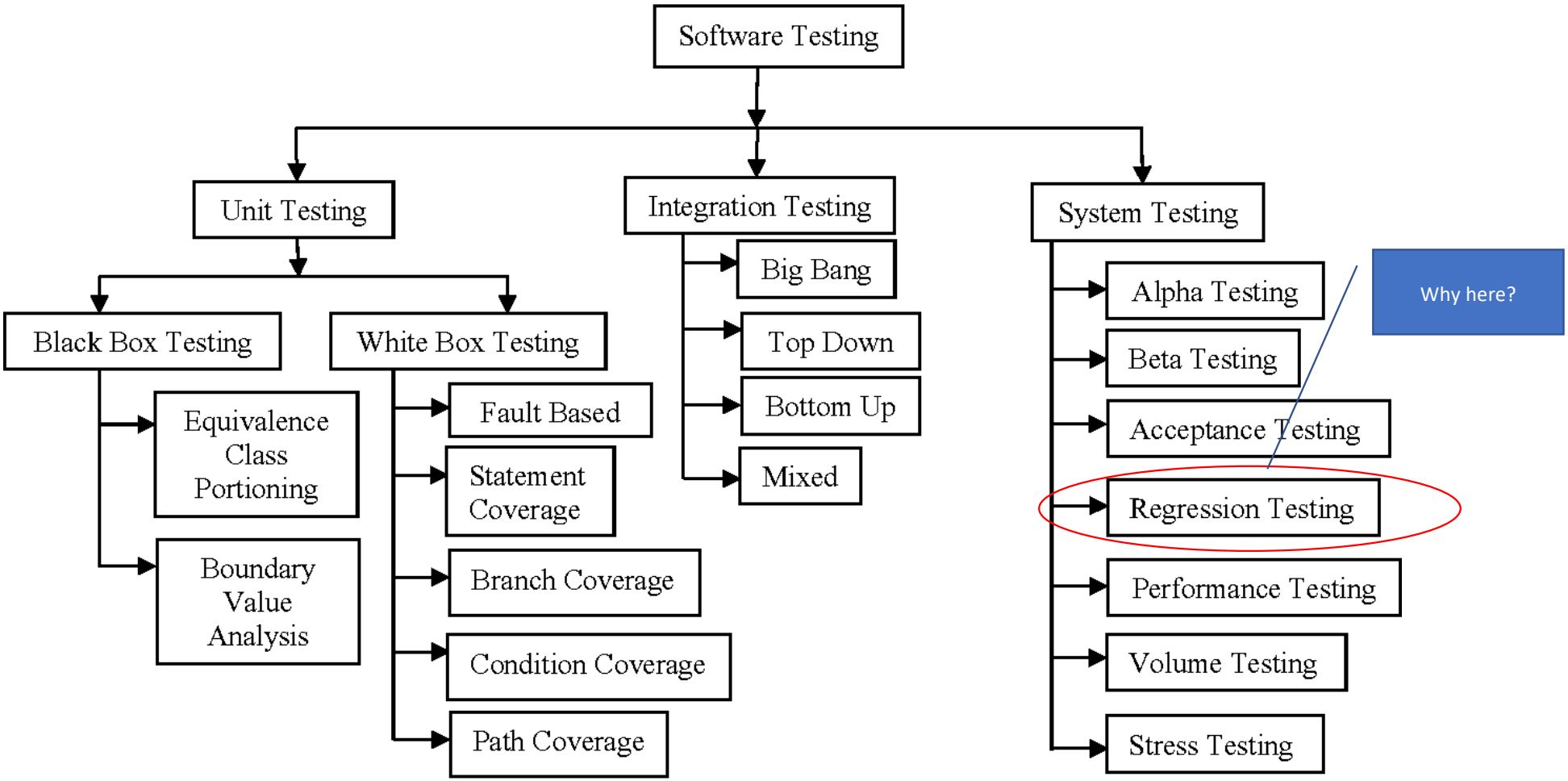
A taxonomy of tests



another…

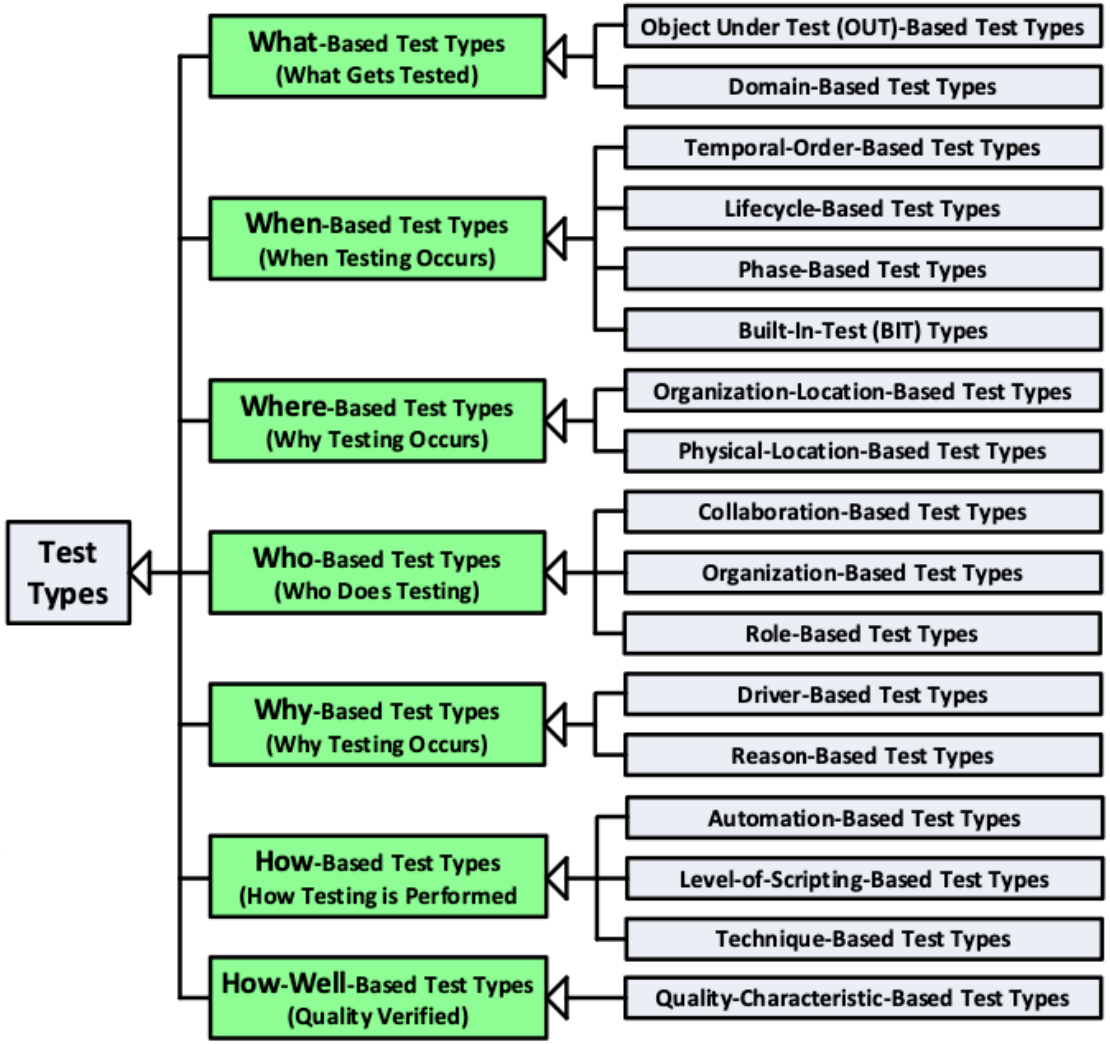

Software Testing Classifications
Testing Goals
Validation Testing (conformation to requirements)
Defect Testing (check bugs)
Execution Methods
Manual Testing
Automated Testing
Semi-automated Testing
Testing Stages
Development Testing (by developers, on developing product)
Unit Testing (on individual components)
Integration Testing (on composite components/ modules)
System Testing (on whole system)
Release Testing (by QA team, on released versions)
User Acceptance Testing (by system users, on whole product)
Alpha Testing (at developers' site, on whole product, by developers & QA team)
Beta Testing (at customers' site, on whole product, by customers as experiments)
Acceptance Testing (at customers' site, on whole product, by customers)
Testing Phases
Smoke Testing
Retesting
Sanity Testing
Regression Testing
Accessibility
Black Box (user perspective, no source code access)
White Box (have access to internal logic and source code)
Grey Box (limited access to internal logic and source code)
Testing Scenarios
Positive
Negative
Requirements
Functional
Installation, DB Integrity, Feature, ...
Non-functional
UI, Usability, Performance, Load, Stress, Recovery, Localization, Internationalization, ...
Structure
Structured (Test case based, Check-lists based Exploratory)
Unstructured (Ad-hoc, Exploratory)
https://www.ashenlive.com/blog

and another…

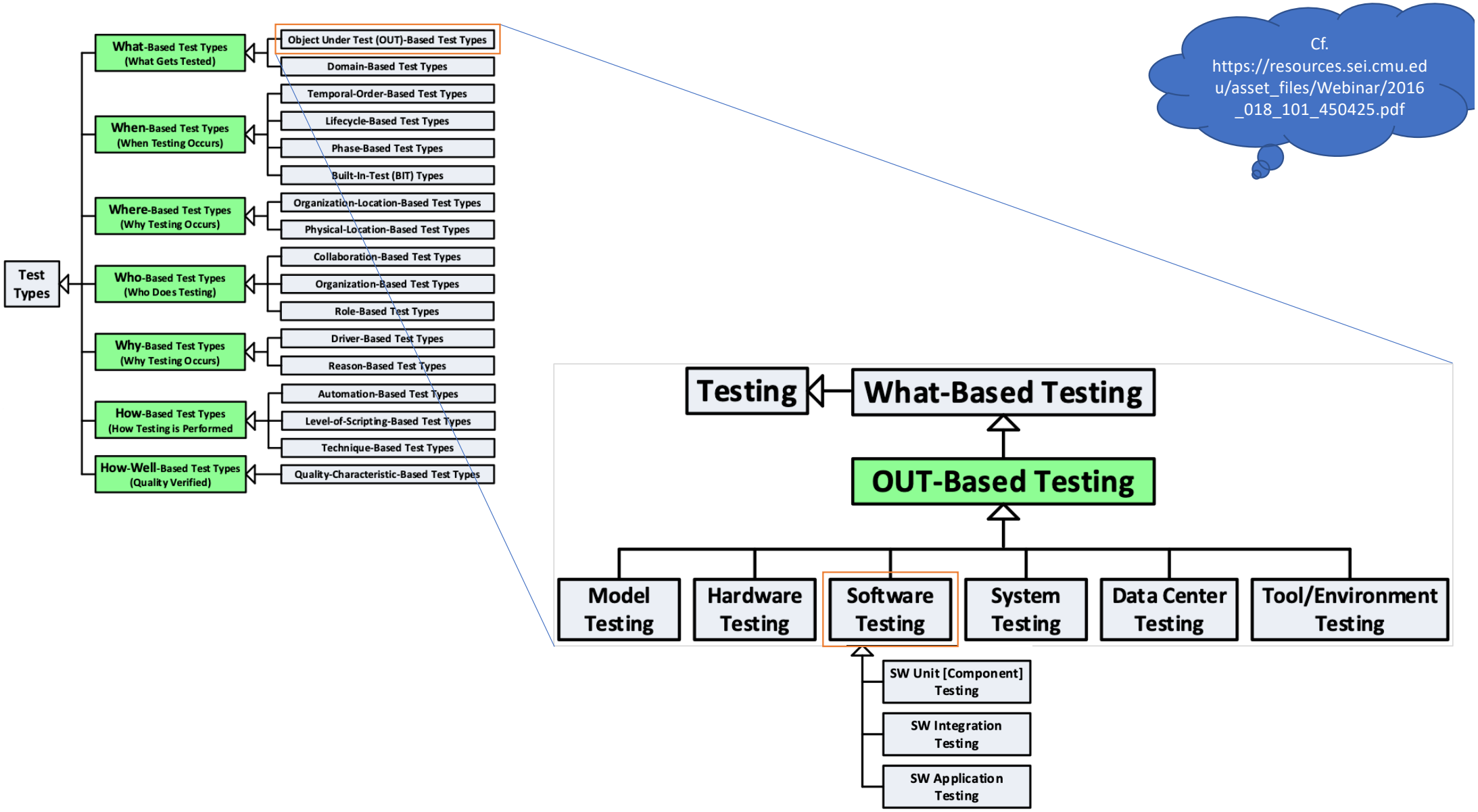

Software Testing
Unit Testing
Integration Testing
System Testing
Black Box Testing
White Box Testing
Equivalence Class Portioning
Boundary Value Analysis
Fault Based
Statement Coverage
Branch Coverage
Condition Coverage
Path Coverage
Big Bang
Top Down
Bottom Up
Mixed
Alpha Testing
Beta Testing
Acceptance Testing
Regression Testing
Performance Testing
Volume Testing
Stress Testing
Why here?

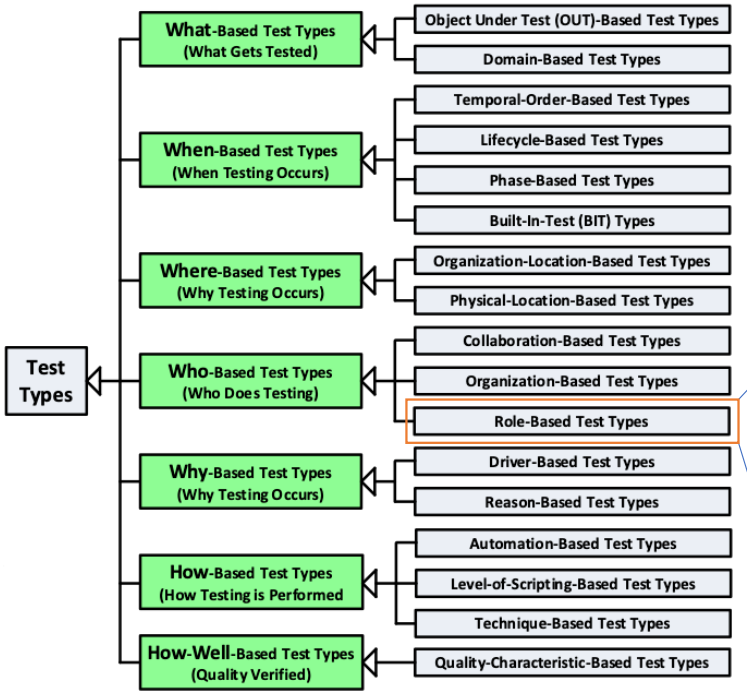
A thorough taxonomy

- 16 Categories of Testing Types Answering the 5W+2H Questions:
- • What?
  • When?
  • Where?
  • Who?
  • Why?
  • How?
  • How Well?
- These supertypes are not disjoint (think multiple inheritance)!

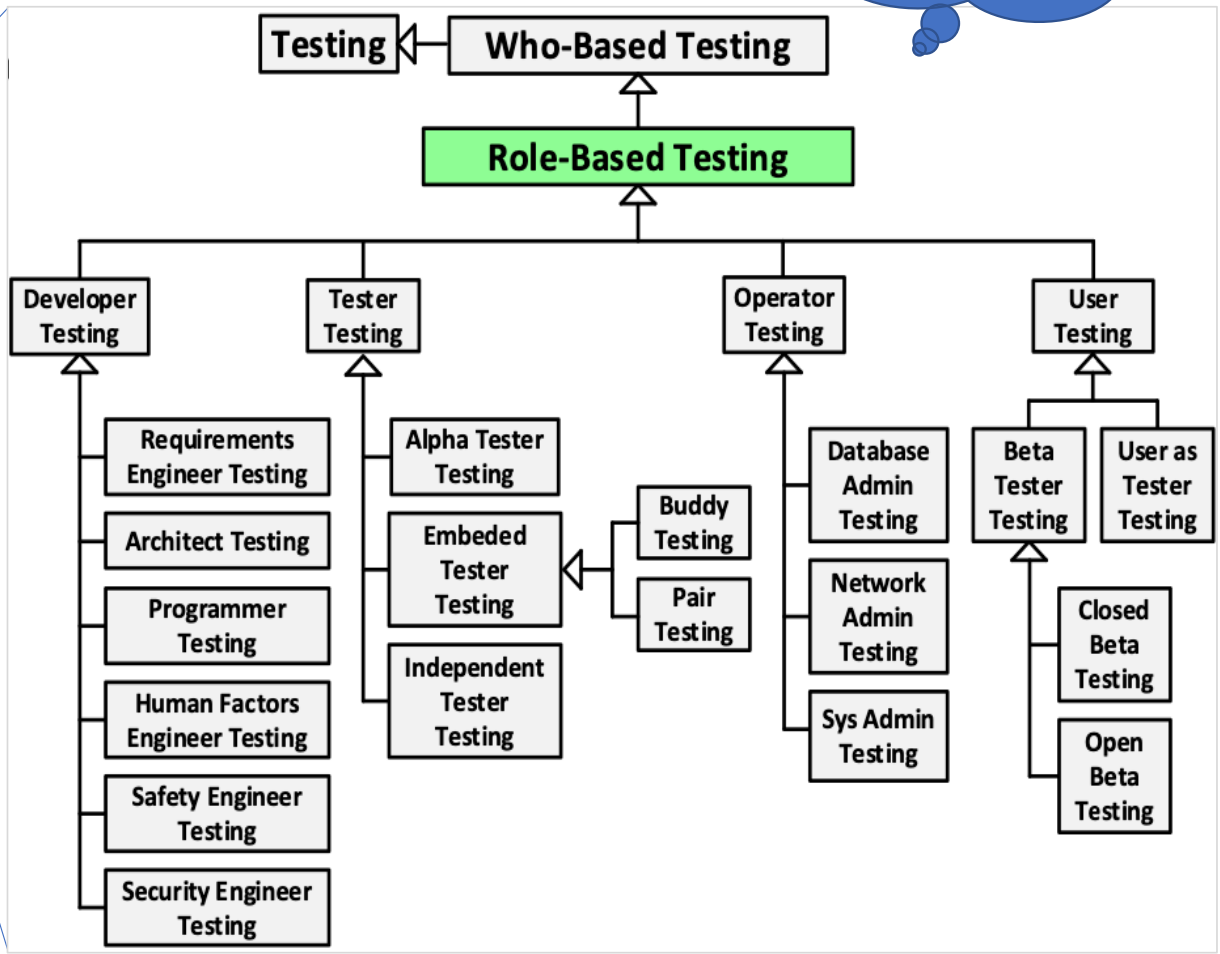


https://resources.sei.cmu.edu/asset_files/Webinar/2016_018_101_450425.pdf


Test Types
What-Based Test Types (What Gets Tested)
Object Under Test (OUT)-Based Test Types
Domain-Based Test Types
When-Based Test Types (When Testing Occurs)
Temporal-Order-Based Test Types
Lifecycle-Based Test Types
Phase-Based Test Types
Built-In-Test (BIT) Types
Where-Based Test Types (Why Testing Occurs)
Organization-Location-Based Test Types
Physical-Location-Based Test Types
Who-Based Test Types (Who Does Testing)
Collaboration-Based Test Types
Organization-Based Test Types
Role-Based Test Types
Why-Based Test Types (Why Testing Occurs)
Driver-Based Test Types
Reason-Based Test Types
How-Based Test Types (How Testing is Performed
Automation-Based Test Types
Level-of-Scripting-Based Test Types
Technique-Based Test Types
How-Well-Based Test Types (Quality Verified)
Quality-Characteristic-Based Test Types
Cf. https://resources.sei.cmu.edu/asset_files/Webinar/2016_018_101_450425.pdf
Testing
What-Based Testing
OUT-Based Testing
Model Testing
Hardware Testing
Software Testing
System Testing
Data Center Testing
Tool/Environment Testing
SW Unit [Component] Testing
SW Integration Testing
SW Application Testing

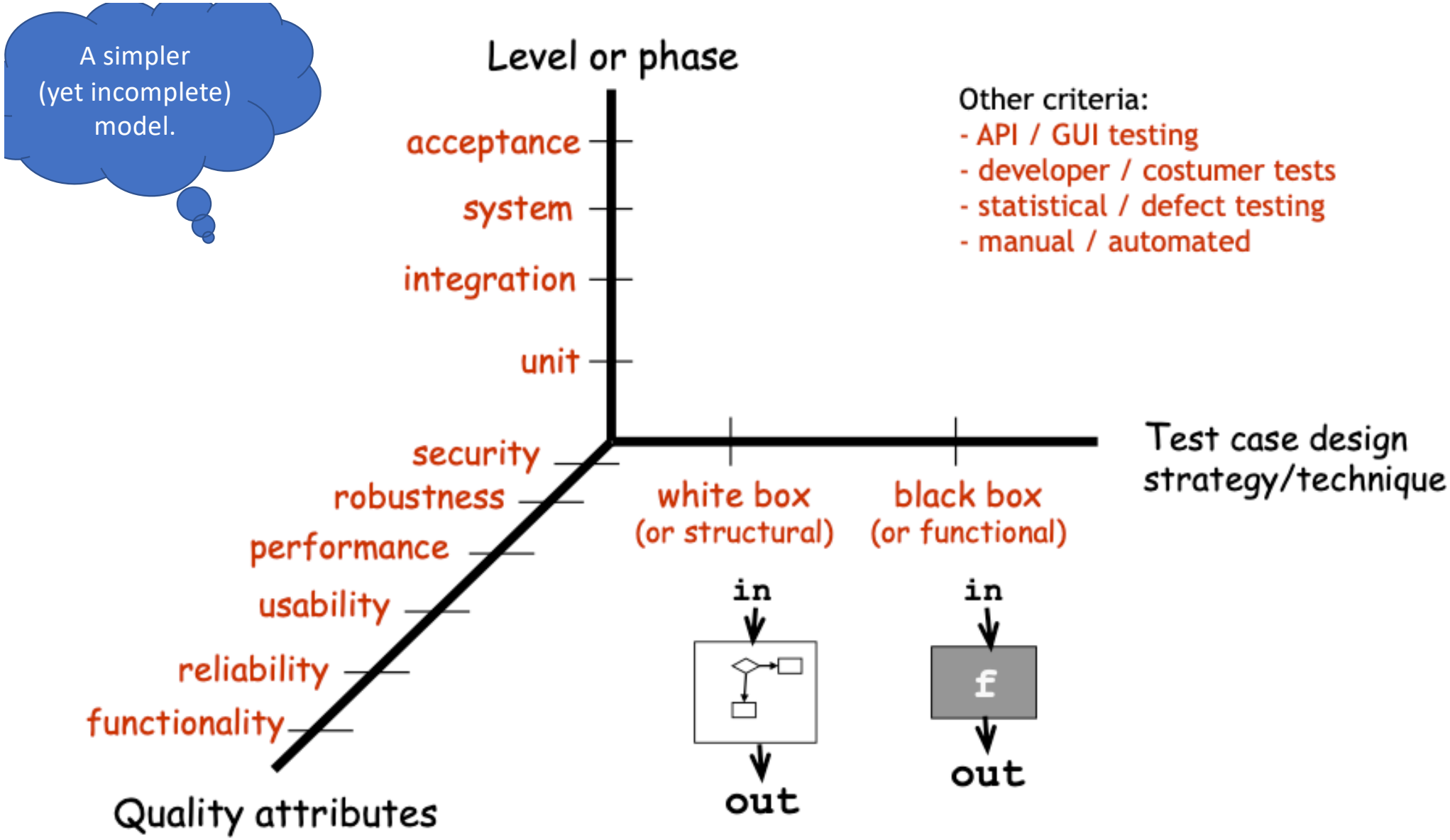


Cf. https://resources.sei.cmu.edu/asset_files/Webinar/2016_018_101_450425.pdf



Cf. details

- in: https://resources.sei.cmu.edu/asset_files/Webinar/2016_018_101_450425.pdf


A simpler
(yet incomplete)
model.
Level or phase
acceptance
system
integration
unit
Other criteria:
- API / GUI testing
- developer / costumer tests
- statistical / defect testing
- manual / automated
security
robustness
performance
usability
reliability
functionality
Quality attributes
white box
(or structural)
black box
(or functional)
Test case design
strategy/technique
in
out
in
f
out

(some) Definitions

Having written requirements is essential.

Some of them can be written as software programs and executed by the testing software, but other are to be executed by humans.

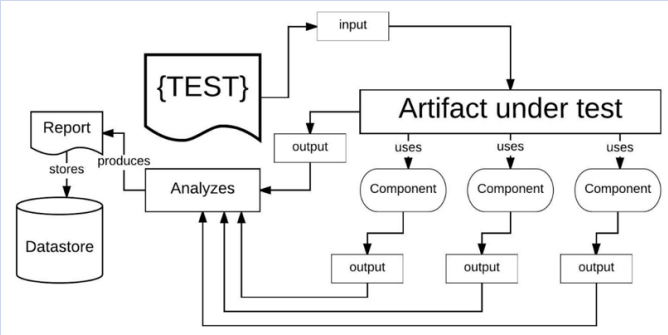
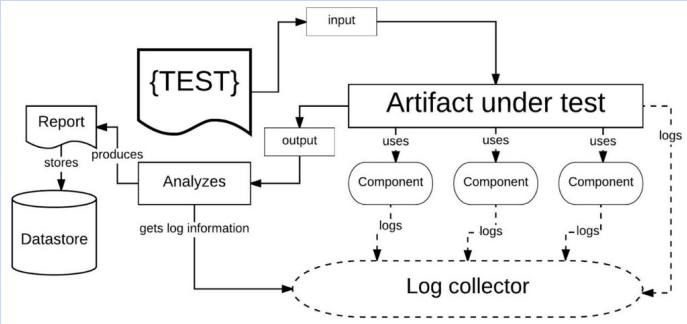
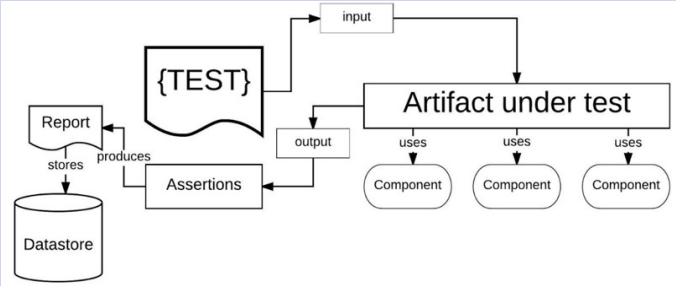
- Functional testing (verification)
  - checking the behavior of the software
  - checking the correctness of the outputs
- Acceptance testing (validation)
  - formal test determining whether a system satisfies its acceptance criteria, i.e., the software can or cannot be used by the end user to perform the functions for which the software was built
  - performed by customer
- Regression testing
  - performed to determine if the software still meets all its requirements considering the changes and modifications to the software
  - ensure that no degradation of baseline functionality has occurred with modifications
  - involves selectively repeating existing tests, not developing new tests

Better use automated (programmed) tests.

Methods

- White (Transparent) Box Testing: Structural Testing
  - the internal structure/design/implementation of the item being tested is known to the tester
- Black (Opaque) Box Testing: Behavioral Testing
  - the internal structure/design/implementation of the item being tested is not known to the tester
- Gray (Semi-transparent) Box Testing
  - the internal structure is partially known
  - E.g. isolating the SUT but not checking the internal structure/design/implementation

Comparison between test methods

White/Transparent Box	Gray/Semi-Transparent Box	Black/Opaque Box
		
White/Transparent Box Integration Tests have access to and verify all the parts relevant to the artifact under test.	In Gray/Semi-transparent Box Testing input and output data is apparent, but also data emitted and recorded by constituent components can be collected for analysis.	Everything but the input and output of an artifact under test is apparent and subject to verification in a Black/Opaque Box Test.

Cf. https://www.mabl.com/blog/the-fundamentals-of-integration-testing

Opaque-box vs. Transparent-box testing (i)

Parameter	Opaque-box testing	Transparent-box testing
Definition	Used to test the software without the knowledge of the internal structure of program or application.	Internal structure is known to the tester.
Alias	It is also known as data-driven, box testing, data-, and functional testing.	It is also called structural testing, clear box testing, code-based testing, or glass box testing.
Base of Testing	Testing is based on external expectations; internal behavior of the application is unknown.	Internal working is known, and the tester can test accordingly.
Usage	This type of testing is ideal for higher levels of testing like System Testing, Acceptance testing.	Best suited for a lower level of testing like Unit Testing, Integration testing.
Programming knowledge	Not needed.	Required.

https://www.guru99.com/back-box-vs-white-box-testing.html

Opaque-box vs. Transparent-box testing (ii)

Parameter	Opaque-box testing	Transparent-box testing
Implementation knowledge	Implementation knowledge is not required.	Complete understanding needs to implement WhiteBox testing.
Automation	Test and programmer are dependent on each other, so it is tough to automate.	Easy to automate.
Objective	The main objective of this testing is to check the functionality of the SUT.	The main objective is to check the quality of the code.
Basis for test cases	Testing can start after preparing requirement specification document.	Testing can start after preparing for Detail design document.
Tested by	Performed by the end user, developer, and tester.	Usually done by tester and developers.
Granularity	Granularity is low.	Granularity is high.

https://www.guru99.com/back-box-vs-white-box-testing.html

Opaque-box vs. Transparent-box testing (iii)

Parameter	Opaque-box testing	Transparent-box testing
Testing method	It is based on trial and error method.	Data domain and internal boundaries can be tested.
Time	It is less exhaustive and time-consuming.	Exhaustive and time-consuming method.
Algorithm test	Not the best method for algorithm testing.	Best suited for algorithm testing.
Code Access	Code access is not required.	Requires code access. Thereby, the code could be stolen if testing is outsourced.
Benefit	Well suited and efficient for large code segments.	It allows removing the extra lines of code, which can bring in hidden defects.

https://www.guru99.com/back-box-vs-white-box-testing.html

Opaque-box vs. Transparent-box testing (iv)

Parameter	Opaque-box testing	Transparent-box testing
Skill level	Low skilled testers can test the application with no knowledge of the implementation of programming language or operating system.	Need an expert tester with vast experience to perform transparent box testing.
Techniques	Equivalence partitioning divides input values into valid and invalid partitions and selecting corresponding values from each partition of the test data. Boundary value analysis checks boundaries for input values.	Statement Coverage validates whether every line of the code is executed at least once. Branch coverage validates whether each branch is executed at least once Path coverage method tests all the paths of the program.
Drawbacks	Update to automation test script is essential if you to modify application frequently.	Automated test cases can become useless if the code base is rapidly changing.

https://www.guru99.com/back-box-vs-white-box-testing.html

Levels of tests

- Unit
- Integration
- System
- Acceptance

Unit Testing

- Tests each module individually
- Performed by developers
- Follows a transparent box testing
  - Algorithms and logic
  - Data structures
    - Global
    - Local
  - Interfaces
  - Independent paths
  - Boundary conditions
  - Error handling

System or Module under test (SUT or MUT).
For example:
- Method
- Class
- Component
- Library
- Framework
- (System)

Integration Testing

- Systematic testing
- Individual units are combined and tested as a group
- The purpose is to expose faults in the interaction between integrated units

Why Integration Testing Is Necessary

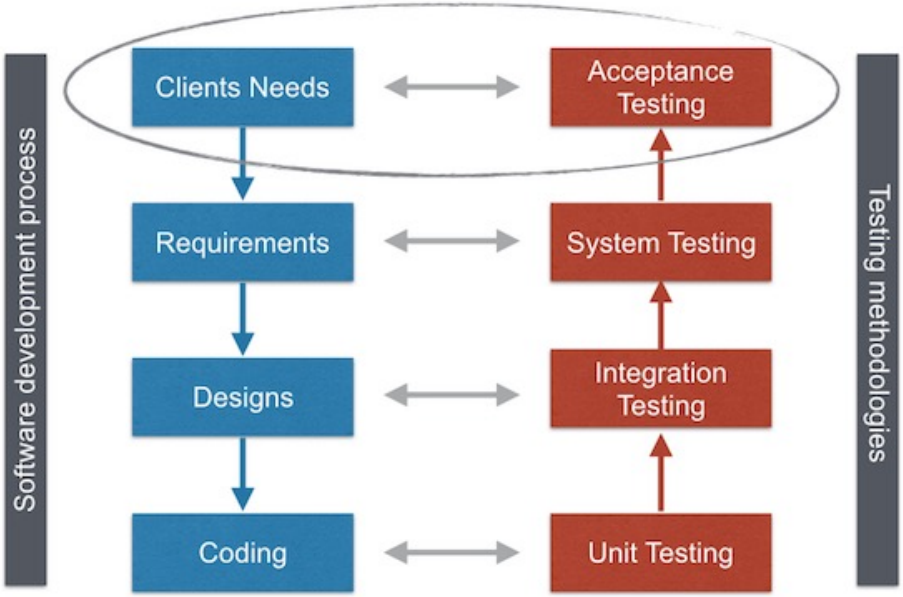
- One module can have an adverse effect on another
  - Subfunctions, when combined, may not produce the desired major function
  - Individually acceptable imprecision in calculations may be magnified to unacceptable levels
- Interfacing errors not detected in unit testing may appear
- Timing problems (in real-time systems) are not detectable by unit testing
- Resource contention (concurrency) problems are not detectable by unit testing

System Testing

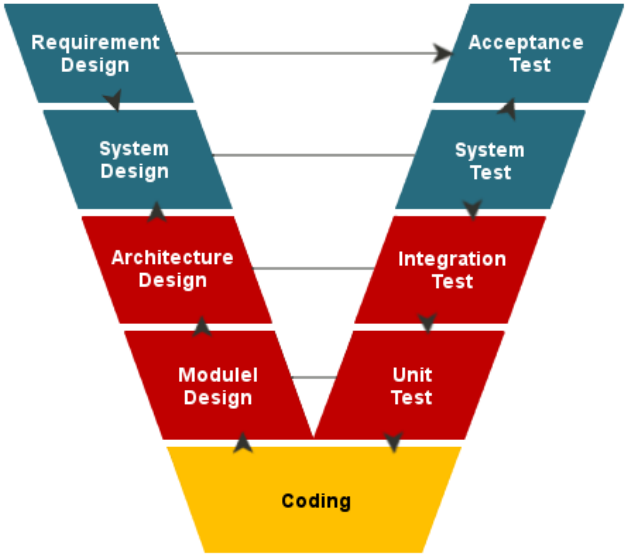
- Part of Validation
- Determine if the software meets all the requirements defined in the software requirements specification (SRS)

Acceptance Testing

- Similar to system testing except that customers are present or directly involved.
- Usually, the tests are developed and run by the customer



https://usersnap.com/blog/types-user-acceptance-tests-frameworks/



Acceptance Testing - Alpha Testing

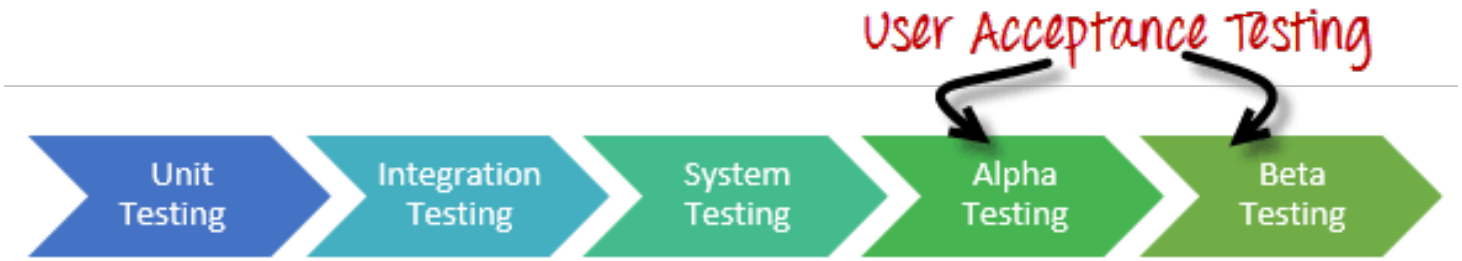
- Performed to identify all possible issues/bugs before releasing the product
- The focus is to simulate real users by using black box and white box techniques.
- Alpha testing is carried out in a lab environment and usually, the testers are internal employees of the organization.
- Is called alpha because it is done early on, near the end of the development of the software.

Acceptance Testing - Beta Testing

- Is performed by "real users" of the software application in a "real environment“.
- Can be considered as a form of external User Acceptance Testing.
- Beta version of the software is released to a limited number of end-users of the product to obtain feedback on the product quality.
- Beta testing reduces product failure risks and provides increased quality of the product through customer validation.
- Direct feedback from customers is a major advantage of Beta Testing. This testing helps to tests the product in customer's environment.

Alpha and Beta Testing

- It's best to provide customers with an outline of the things that you would like them to focus on and specific test scenarios for them to execute.
- Provide with customers who are actively involved with a commitment to fix defects that they discover.



https://www.guru99.com/alpha-beta-testing-demystified.htm

Regression Testing

NB: Automatic tests are regression tests!

- Type of software testing to confirm that a recent program or code change has not adversely affected existing features.
- Regression Testing is nothing but a full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine.
- Regression Testing is required when there is a:
  - Change in requirements and code is modified according to the requirement
  - New feature is added to the software
  - Defect fixing
  - Performance issue fix

Good Testing Practices

- A good test case is one that has a high probability of detecting an undiscovered defect, not one that shows that the program works correctly
- It is impossible to test your own program. Assign your best people to testing
- A necessary part of every test case is a description of the expected result
- Write test cases for valid as well as invalid input conditions.
- Thoroughly inspect the results of each test
- As the number of detected defects in a piece of software increases, the probability of the existence of more undetected defects also increases
- Ensure that testability is a key objective in your software design
- Never alter the program to make testing easier

s.t.U(ntestability).p.i.d.

Unit vs. Integration Testing

Example

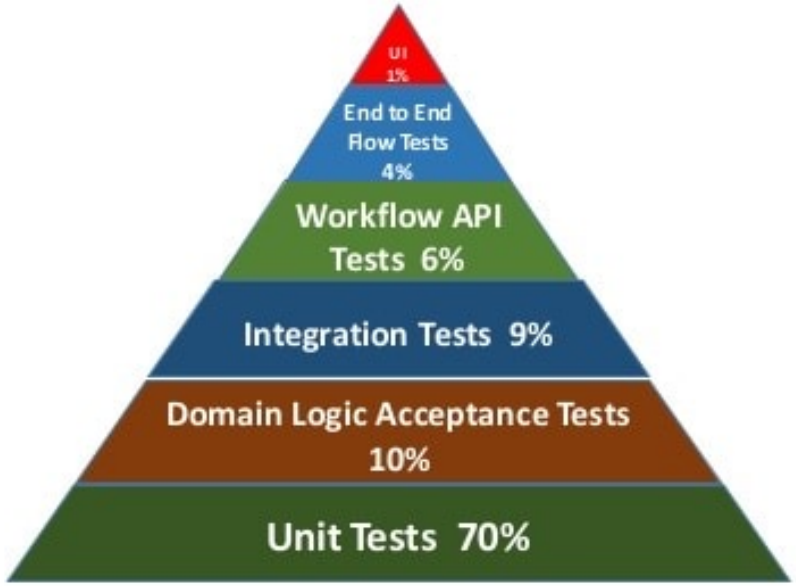
Unit vs. Integration tests

- A **unit test** tests a logical unit, a single isolated component, a function, or a feature. A unit test isolates this component to test it without any dependencies like I did in the last post. First, I tested the actions of a controller, without testing the actual service in behind. Then I tested the service methods in an isolated way with a faked DbContext. Why? Because unit tests shouldn't break because of a failing dependency. A unit test should be fast in development and in execution. It is a development tool. So it shouldn't cost a lot of time to write one. And, in fact, setting up a unit test is much cheaper than setting up an integration test. Usually, you write a unit test during or immediately after implementing the logic. In the best case, you'll write a unit test before implementing the logic. This would be the TDD way, test-driven development or test-driven design.
- An **integration test** does a lot more. It tests the composition of all units. It ensures that all units are working together in the right way. This means it may need a lot more effort to set up a test because you need to set up the dependencies. An integration test can test a feature from the UI to the database. It integrates all the dependencies. On the other hand, an integration test can be isolated on a hot path of a feature. It is also legit to fake or mock aspects that don't need to be tested in this special case. For example, if you test a user input from the UI to the database, you don't need to test the logging. Also, an integration test shouldn't fail because on an error outside the context. This also means to isolate an integration test as much as possible, maybe by using an in-memory database instead of a real one.

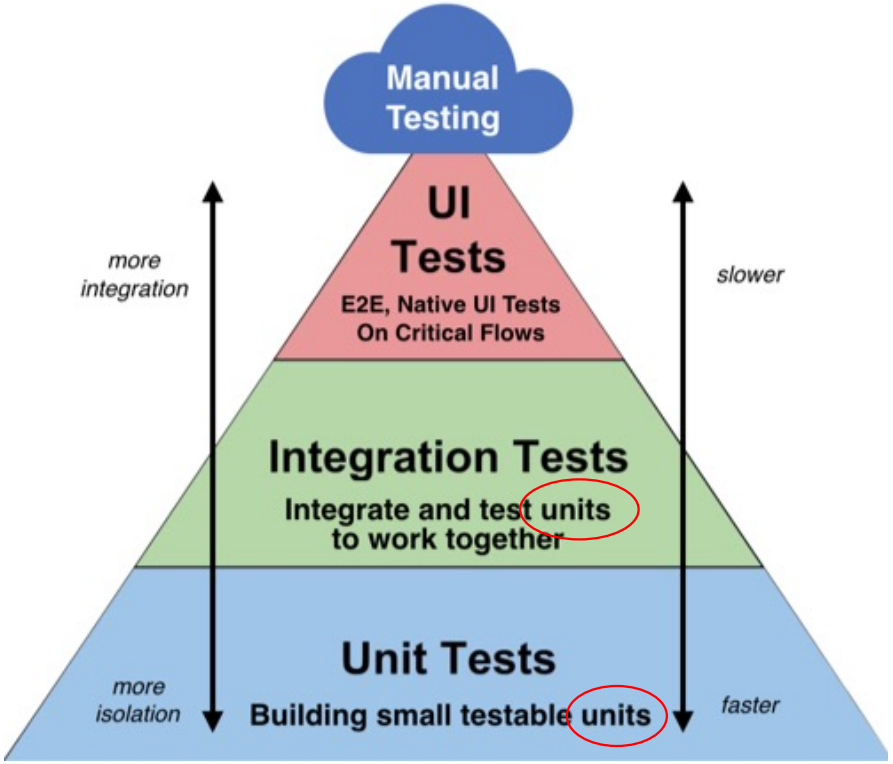
https://dzone.com/articles/integration-testing-data-access-in-aspnet-core

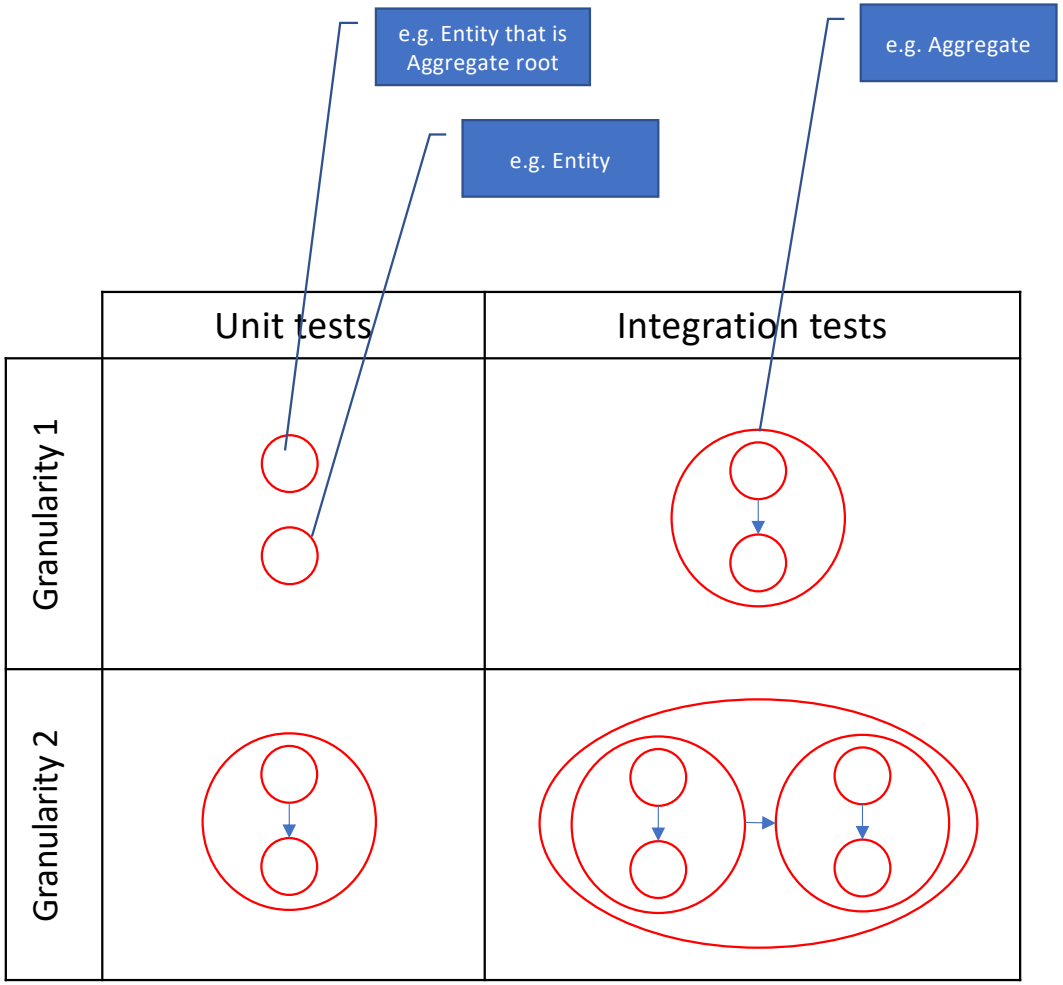
Unit vs. Integration tests

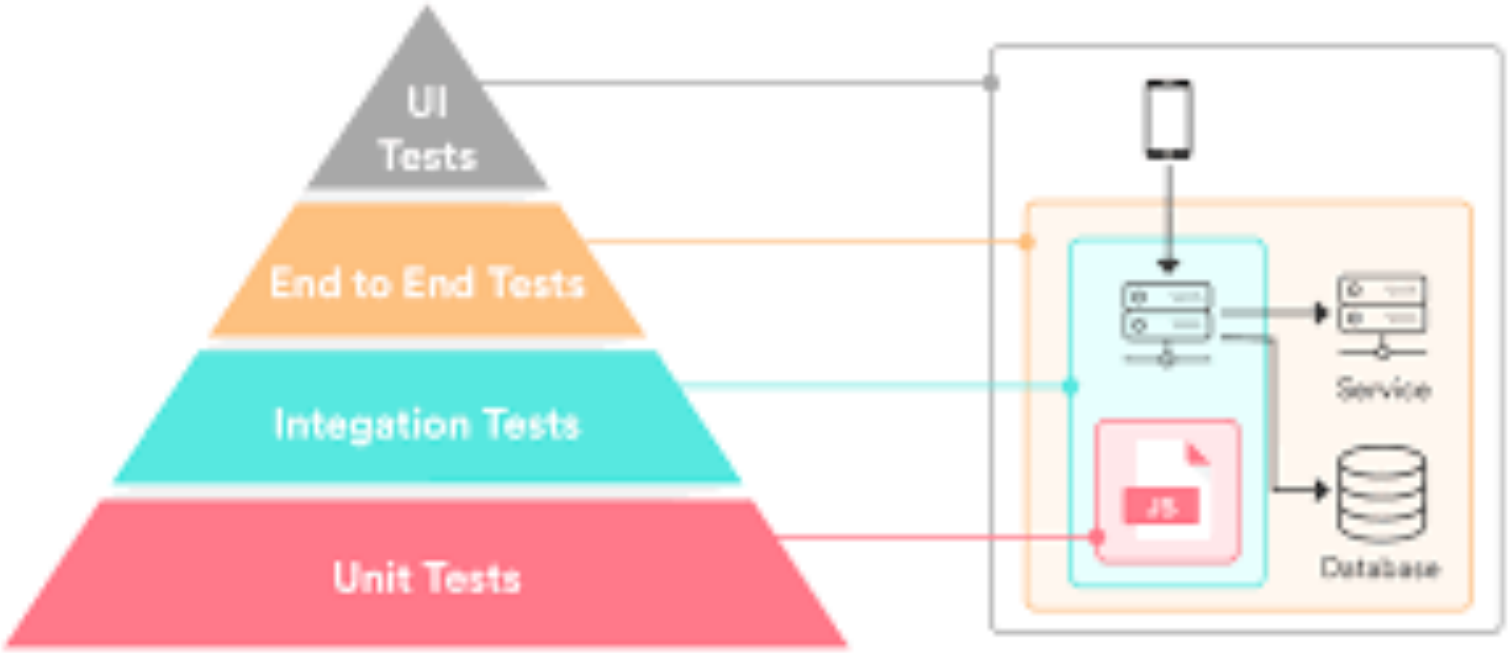
	Unit Testing	Integration Testing
Definition	Tests an atomic module to ensure it works as expected	Tests several modules to ensure they work together as expected
Condition	Checks a single component	Checks multiple components in combination
Complexity	Quite simple, covers a small piece of code, doesn't require additional integration preparations	Complex, covers the whole application or module, requires integration preparations before testing
Dependency	Has no dependencies from the outside	Has a dependency on store, database, etc. to be sure all the application's functionality is properly tested
Order of implementation	First step of testing	Second step of testing, conducted after unit testing



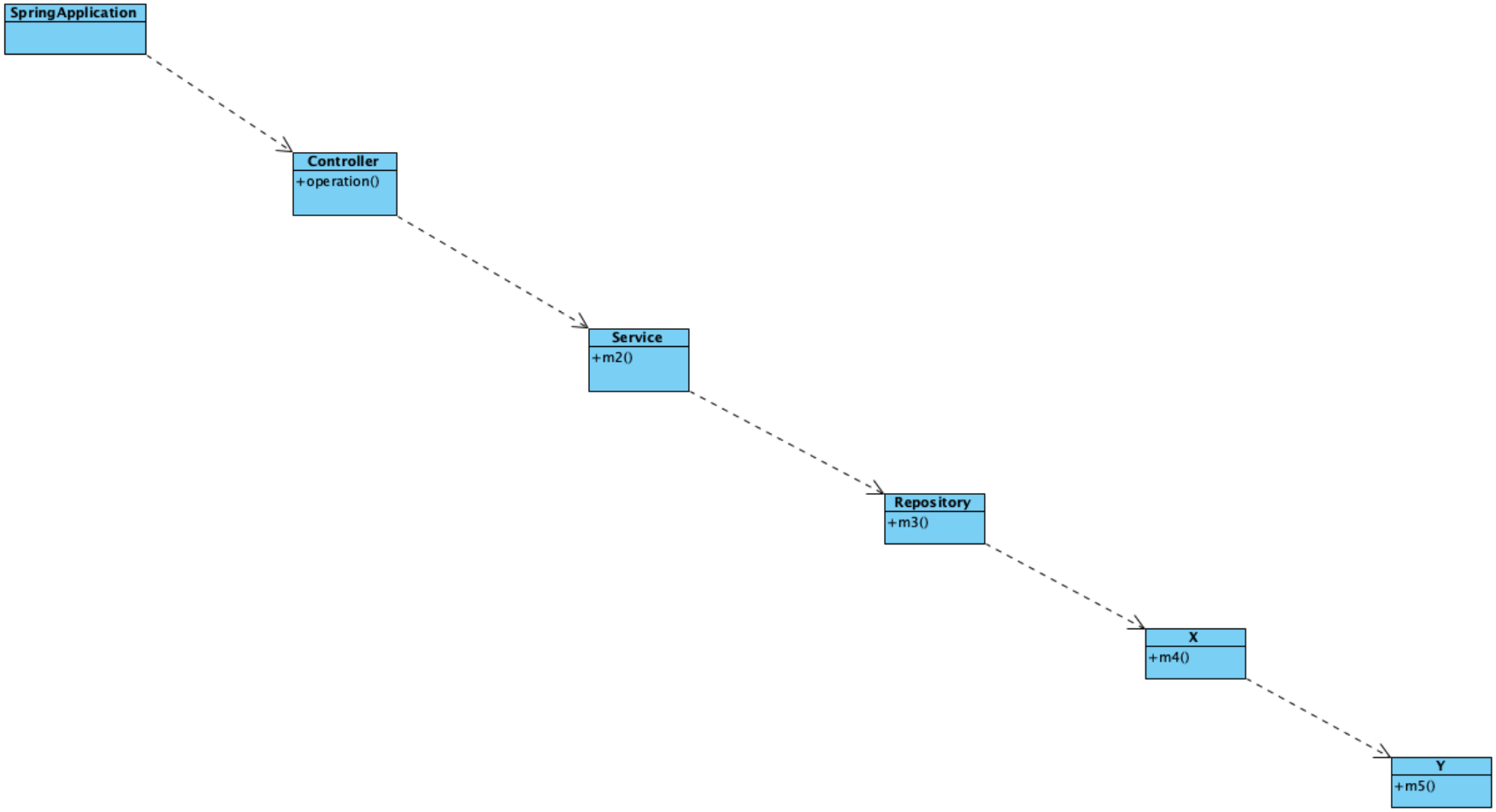
What is a unit?






UI
Tests
End to End Tests
Integation Tests
Unit Tests
JS
Service
Database

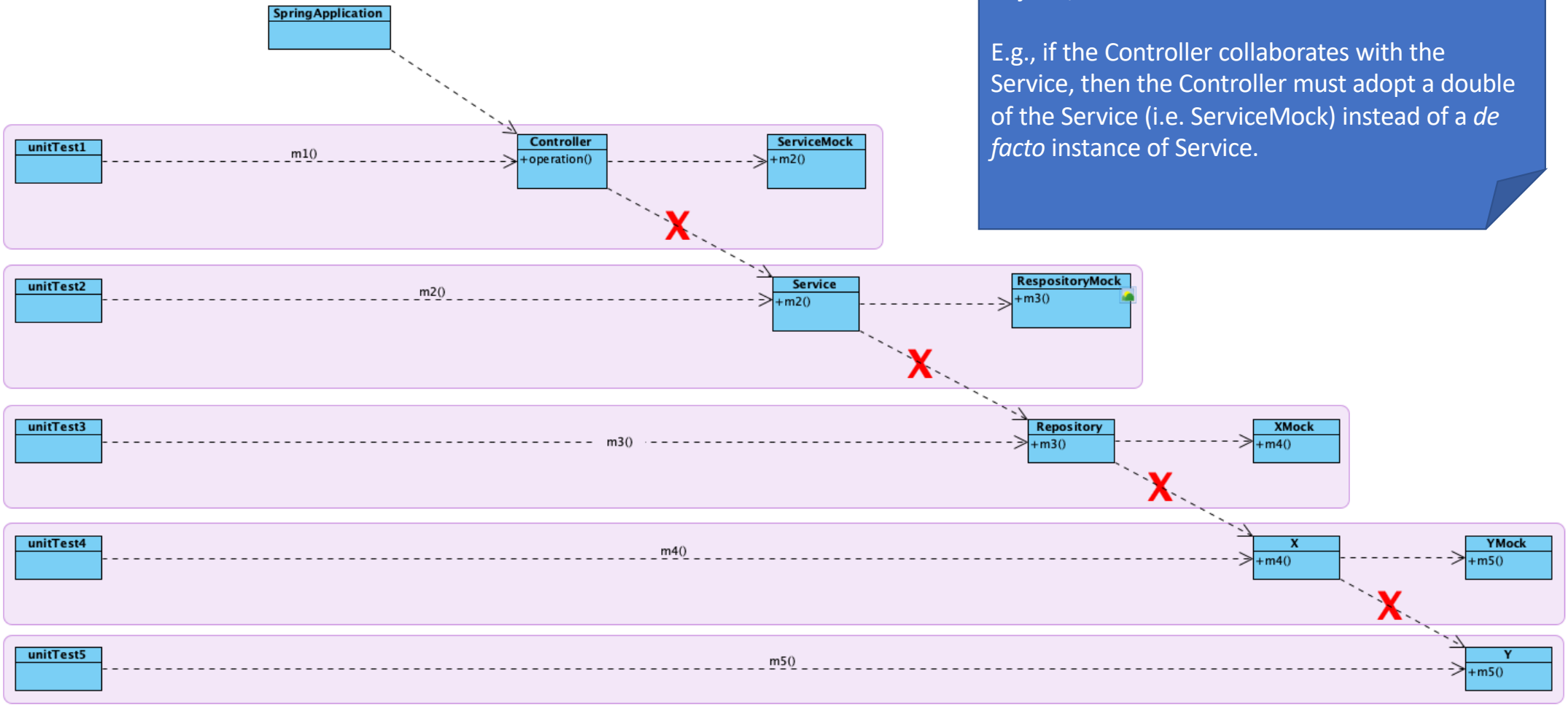
Scenario/Application (example)



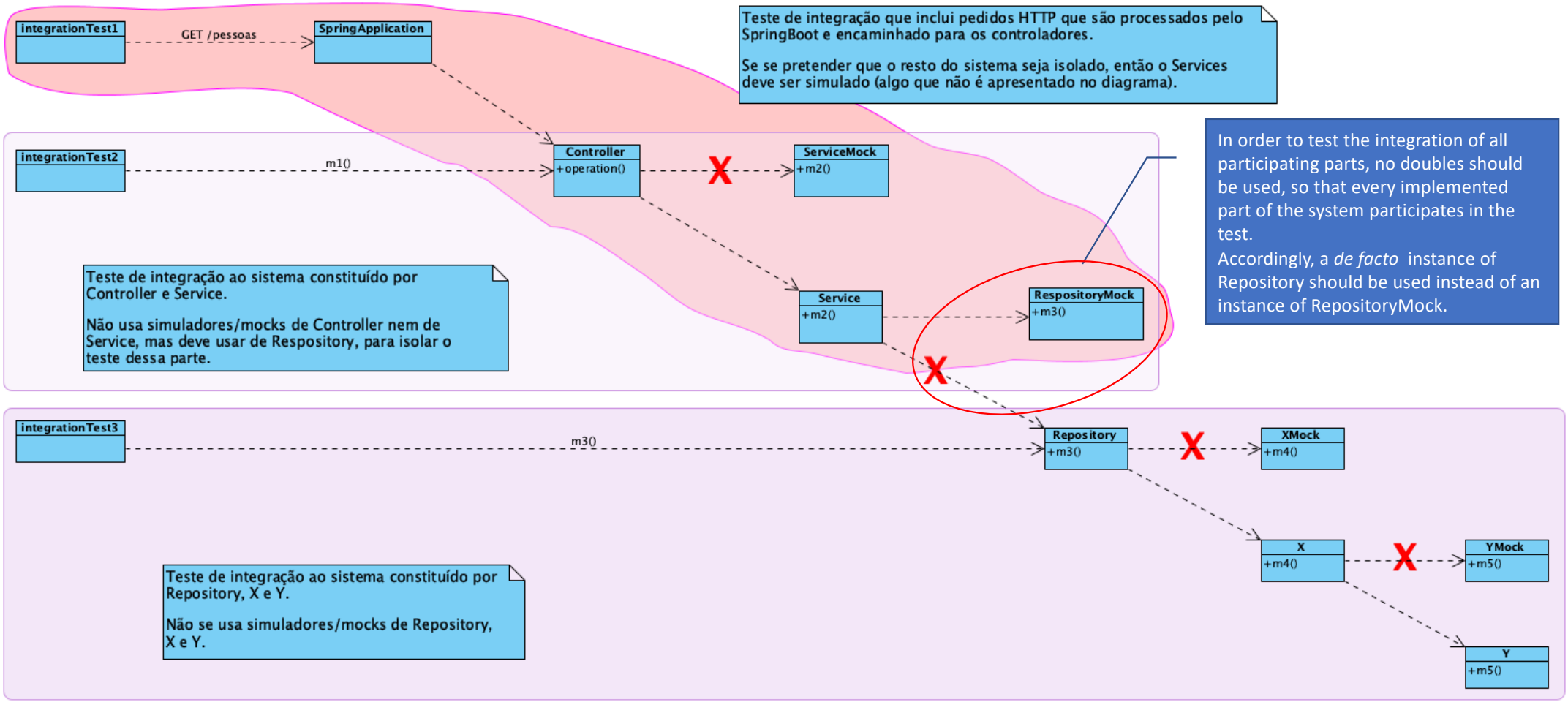
Unit tests (examples)

For each unit test, one must isolate the method/object under test with doubles/simulations of the collaborating objects/methods.

E.g., if the Controller collaborates with the Service, then the Controller must adopt a double of the Service (i.e. ServiceMock) instead of a *de facto* instance of Service.



Integration tests (examples)



Doubles (Duplos)

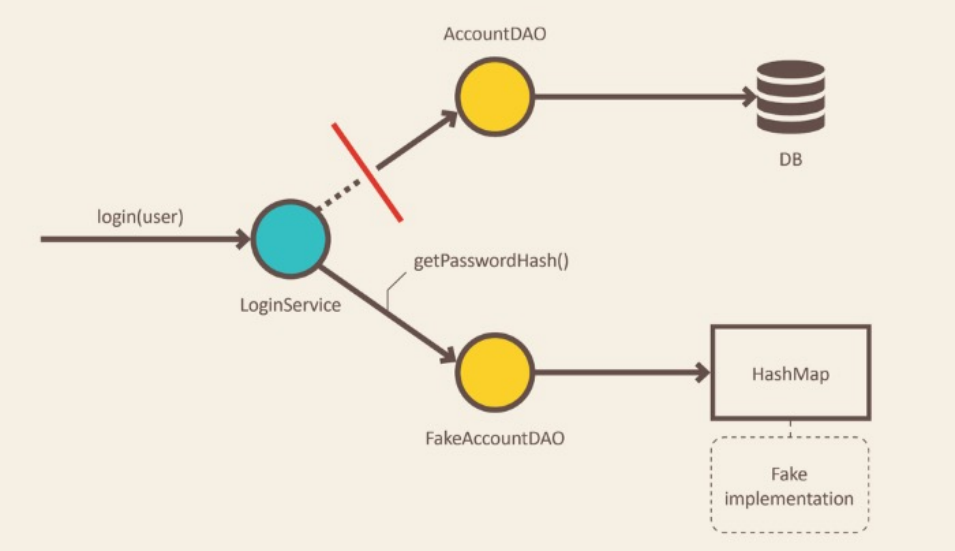
Definitions

- **Dummy** objects are passed around but never actually used. Usually, they are just used to fill parameter lists.
- **Fake** objects that actually have working implementations, but usually take some shortcut which makes them not suitable for production (an InMemoryTestDatabase is a good example).
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
  - ***Stub is an object that holds predefined data and uses it to answer calls during tests. It is used when we cannot or don't want to involve objects that would answer with real data or have undesirable side effects.***
- **Spies** are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
- **Mocks** are pre-programmed with expectations which form a specification of the calls [and order] they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting.

In https://martinfowler.com/bliki/TestDouble.html

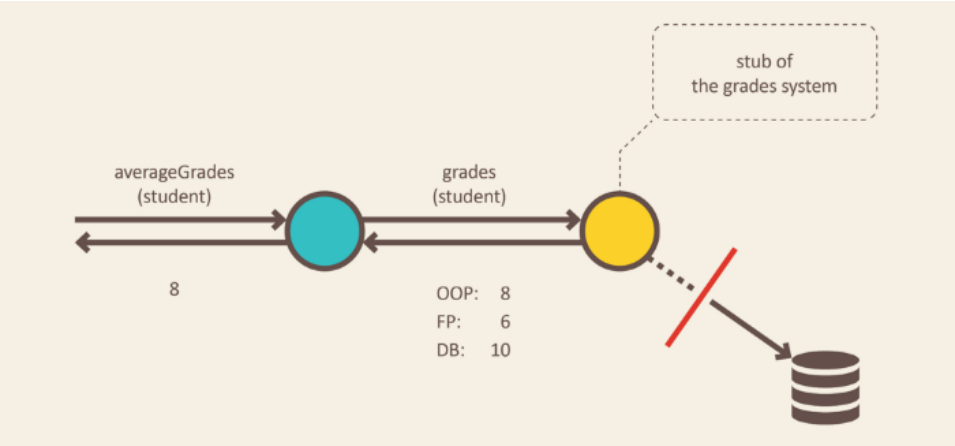
Fake

- Fakes are objects that have working implementations, but not same as production one. Usually, they take some shortcut and have simplified version of production code.



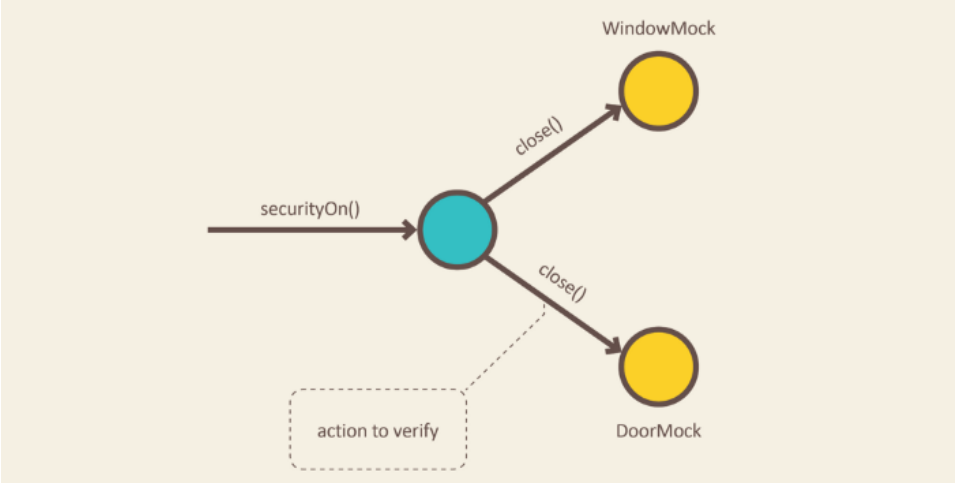
Stub

- Stub is an object that holds predefined data and uses it to answer calls during tests. It is used when we cannot or don't want to involve objects that would answer with real data or have undesirable side effects.



Mock

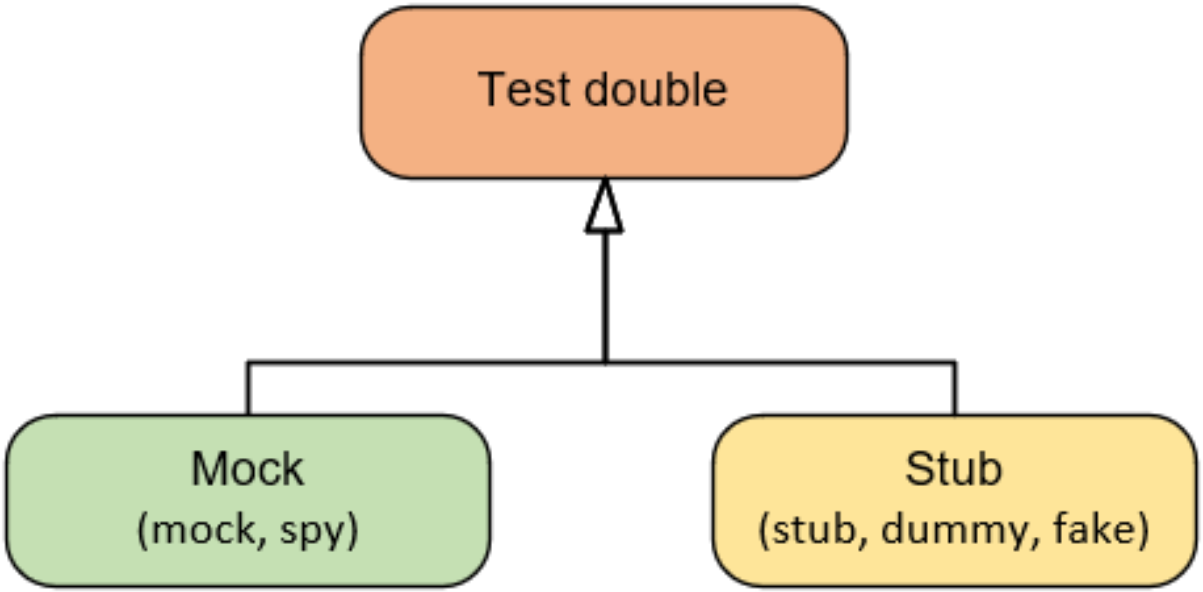
- Mocks are objects that register calls they receive.
- In test assertion we can verify on Mocks that all expected actions were performed



Algo mais em http://xunitpatterns.com/Mock%20Object.html

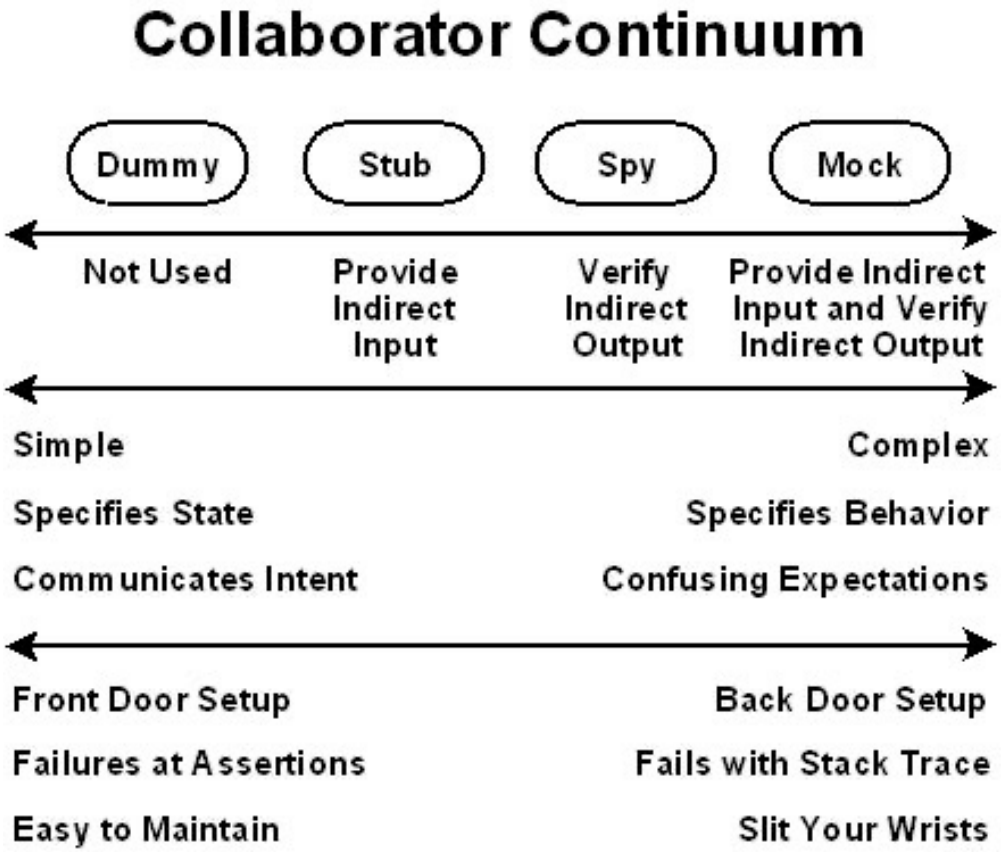
Spy

- A stub that uses behavior.
- It is a simple mock.



Mocks vs. Stubs vs. Spies

- Mock objects always use behaviour verification
- A stub can go either way.
- Meszaros refers to stubs that use behaviour verification as a Test Spy.



https://newbedev.com/what-is-wrong-with-stubs-for-unit-testing

End2End Tests

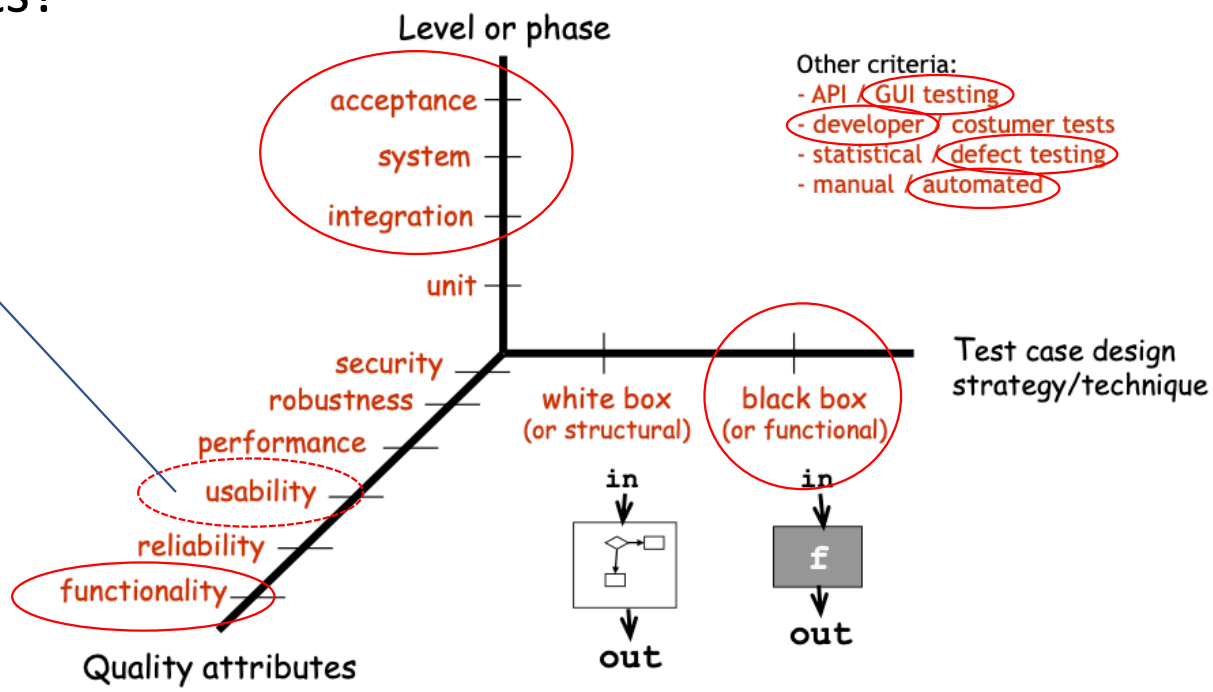
aka e2e tests

Automatic end-to-end tests

- No isolation of parts exists → Doubles are not used
- How to characterize e2e tests?

Usability is partially testable by automatic tests. E.g. the ARIA (https://www.w3.org/TR/wai-aria-1.1/) text in should be [descriptive text].

Yet, usability is a human decision, and therefore should be ultimately validated by users.



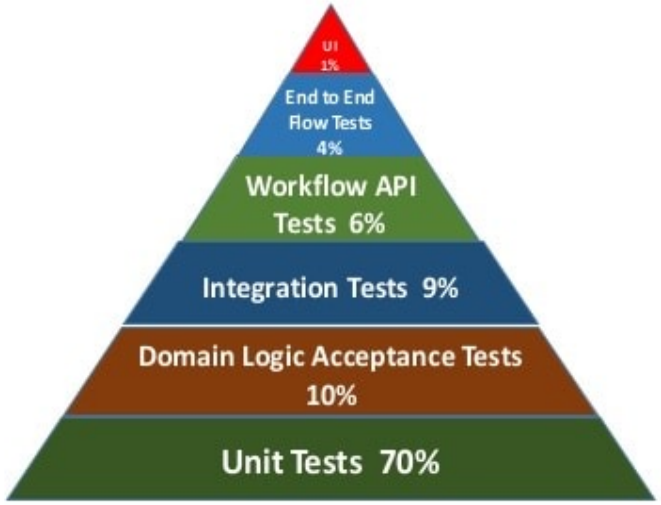
Introduction to E2E tests

- These tests make expectations about what the user sees and reads in the browser.
- Simulate a user interacting with the application:
  - Navigating to an address
  - Reading text
  - Clicking on a link or button
  - Filling out a form
  - Moving the mouse
  - Typing on the keyboard.

Introduction to E2E tests (ii)

- From the user's perspective, it does not matter that your application is implemented in Angular.
- Technical details like the inner structure of your code are not relevant.
- There is no distinction between front-end and back-end, between parts of your code. The full experience is tested.
- These tests are called **end-to-end (E2E) tests** since they integrate all parts of the application from one end (the user) to the other end (the darkest corners of the back-end).
- End-to-end tests also form the automated part of **acceptance tests** since they tell whether the application works for the user.

Introduction to E2E tests (ii)



- Speed: Expensive and slow
- Reliability: unreliable and indeterministic
  - It is often hard to find the root cause of the problem
  - Often, they fail for no apparent reason; next execution suddenly pass
- Setup
  - Use a real browser, even if there is no UI (headless)
  - Need to deploy frontend, backend, databases, caches, network, etc. in nodes

End-to-end testing frameworks

- Since version 12 (?) Angular do not provide a default e2e testing framework
- Protractor
  - Angular-specific e2e testing framework
  - Not recommended because it is deprecated (after Angular version 12)
  - Free
- Cypress
  - Generic e2e testing framework
  - Free, but some features (e.g. dashboard for CI/CD) is a paid service

Final remarks

Good Testing Practices

This collides with many development paradigms, e.g. eXtreme Programming.

- Having written requirements is essential
- A good test case is one that has a high probability of detecting an undiscovered defect, not one that shows that the program works correctly
- It is impossible to test your own program
- A necessary part of every test case is a description of the expected result

Good Testing Practices (cont'd)

Better use automated (programmed) tests.

- Avoid nonreproducible or on-the-fly testing
- Write test cases for valid as well as invalid input conditions
- Thoroughly inspect the results of each test
- As the number of detected defects in a piece of software increases, the probability of the existence of more undetected defects also increases
- Avoid knowing
  - A lot about a few types of testing
  - A little about some additional types of testing
  - Very little about a sizable number of testing types

Good Testing Practices (cont'd)

- Assign your best people to testing
- Ensure that testability is a key objective in your software design
- Never alter the program to make testing easier
- Testing, like almost every other activity, must start with objectives

MYTH: ***Quality Control = Testing.***	**FACT**: Testing is just one component of software quality control. Quality Control includes other activities such as Reviews.
MYTH: ***The objective of Testing is to ensure a 100% defect- free product.***	**FACT**: The objective of testing is to uncover as many defects as possible while ensuring that the software meets the requirements. Identifying and getting rid of all defects is impossible.
MYTH: ***Testing is easy.***	**FACT**: Testing can be difficult and challenging (sometimes, even more so than coding).
MYTH: ***Anyone can test.***	**FACT**: Testing is a rigorous discipline and requires many kinds of skills.
MYTH: ***There is no creativity in testing.***	**FACT**: Creativity can be applied when formulating test approaches, when designing tests, and even when executing tests.
MYTH: ***Automated testing eliminates the need for manual testing.***	**FACT**: 100% test automation cannot be achieved. Manual Testing, to some level, is always necessary.
MYTH: ***When a defect slips, it is the fault of the Testers.***	**FACT**: Quality is the responsibility of all members/stakeholders, including developers, of a project.
MYTH: ***Software Testing does not offer opportunities for career growth.***	**FACT**: Gone are the days when users had to accept whatever product was dished to them; no matter what the quality. With the abundance of competing software and increasingly demanding users, the need for software testers to ensure high quality will continue to grow. Software testing jobs are hot now.

Bibliography and References

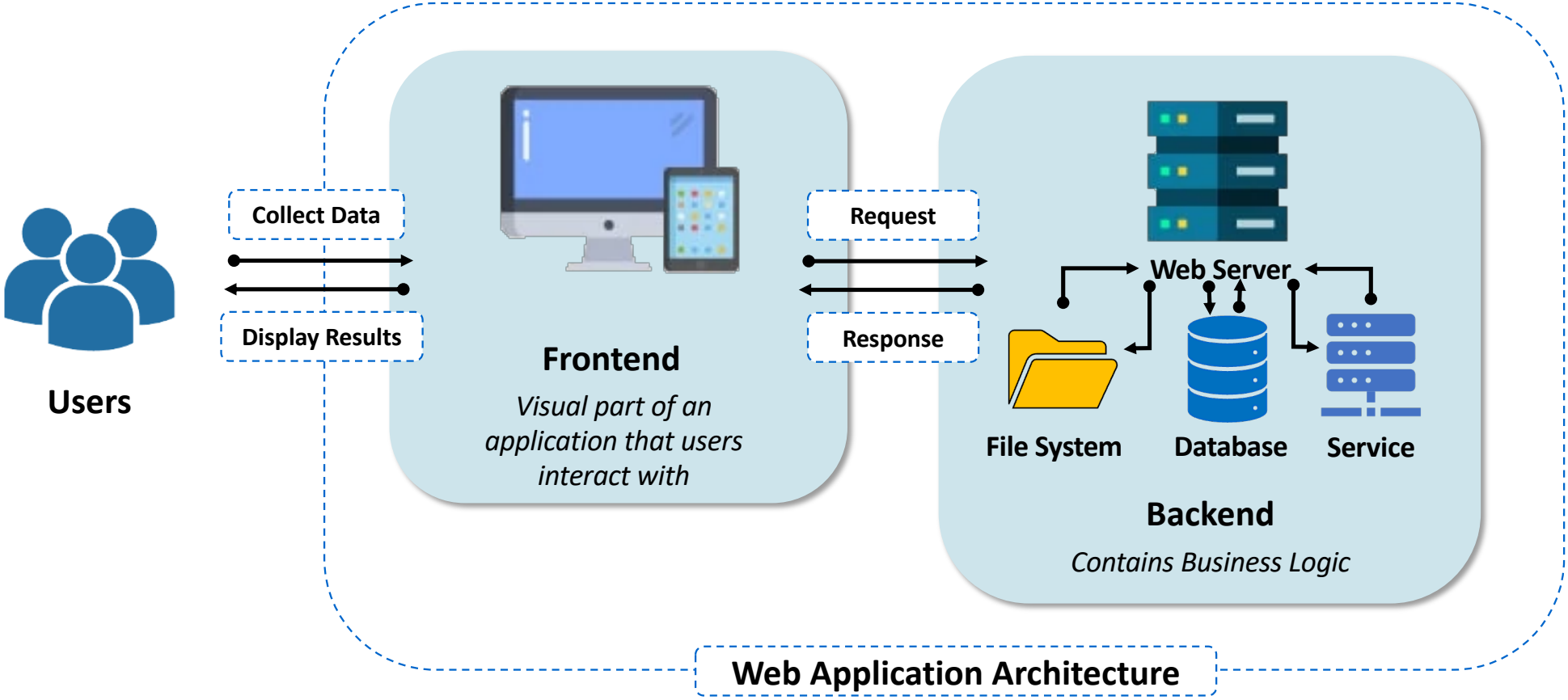
- http://softwaretestingfundamentals.com
- https://www.edureka.co/blog/software-testing-levels/

Introduction to Angular

Web Concepts

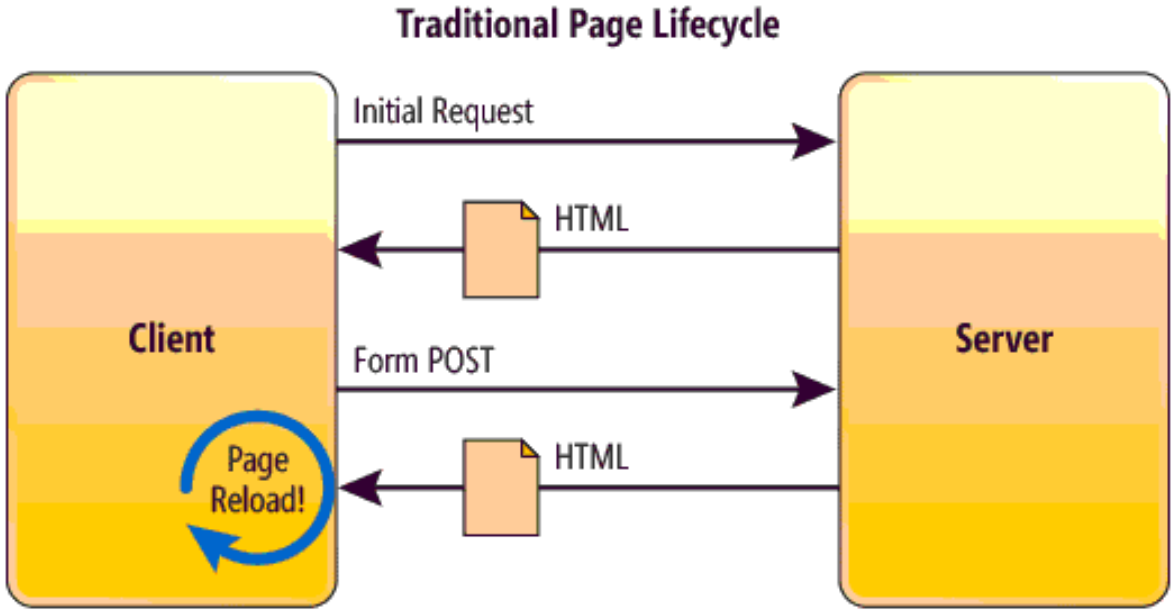
Web Application

- **A web application is interactive**



Types of Web Applications

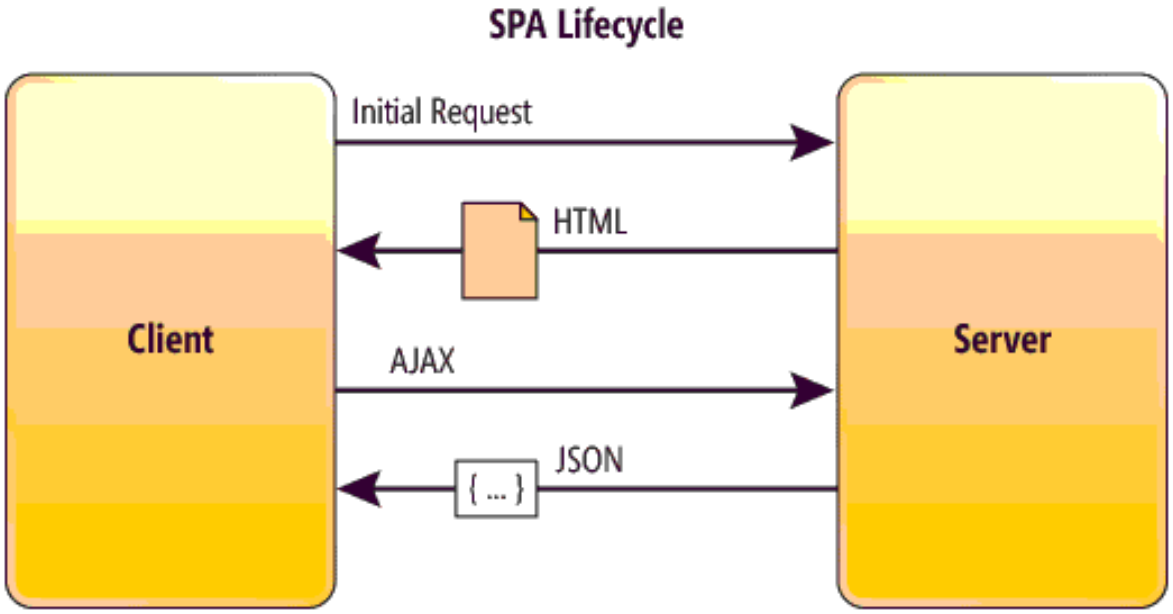
- **Multi Page Application (MPA)**
  - Is the traditional web application architecture that (re)loads the entire web page whenever user interacts with the web application.



cf: https://moz.com/blog/optimizing-angularjs-single-page-applications-googlebot-crawlers

Types of Web Applications

- **Single Page Application (SPA)**
  - Is a web application that re-render its content in response to user interactions without making a request to the server to fetch new web page.



cf: https://moz.com/blog/optimizing-angularjs-single-page-applications-googlebot-crawlers

Angular Framework

What is Angular?

- **Angular is a development platform, built on TypeScript.**
- **As a platform, Angular includes:**
  - A component-based framework for building scalable web applications;
  - A collection of well-integrated libraries that cover a wide variety of features, including routing, forms management, client-server communication, and more;
  - A suite of developer tools to help develop, build, test, and update code.
- **The Angular ecosystem consists of a diverse group of over 1.7 million developers, library authors, and content creators.**

isep Instituto Superior de Engenharia do Porto

Angular applications: The essentials

- **Components**
  - Includes a TypeScript class, an HTML template and stylesheets;
  - The HTML template declares how the component renders (Angular automatically updates the rendered DOM when the component's state changes);
  - The component model offers strong encapsulation and an intuitive application structure;
  - Components make testing easier and increase overall code readability.
- **Dependency injection (DI)**
  - Is a design pattern;
  - DI allows the declaration of the TypeScript classes dependencies without taking care of their instantiation (this instantiation is handled automatically by Angular).
  - This design pattern encourage writing more testable and flexible code.

Angular CLI

- **Is a Command Line Interface (CLI) tool.**
- **Is the fastest, straightforward, and recommended way to develop Angular applications.**

```
> npm install -g @angular/cli
> ng new my-first-angular-app
> cd my-first-angular-app
> ng serve --open
```

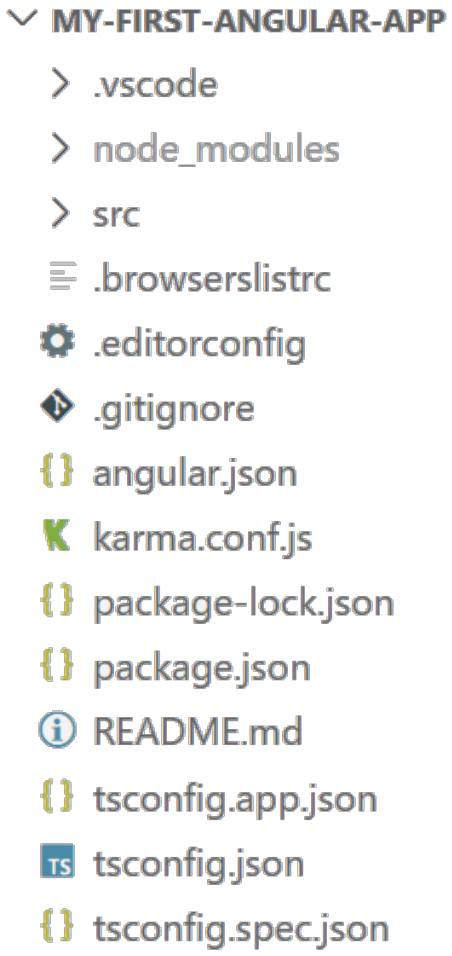
Install the Angular CLI.

Create a new application.

The **`ng serve`** command:

1. builds the app
2. starts the development server
3. watches the source files, and rebuilds the app as you make changes to those files
4. The **`--open`** flag opens a browser to http://localhost:4200

Project Structure



- **`README.md`**
  - Basic documentation for the project.
- **`package.json`**
  - Project metadata including third party packages used by the project.
- **`package-lock.json`**
  - Automatically generated whenever there is a modification in node-modules or package.json. Describes the exact dependency tree that was generated.
- **`tsconfig.json`**
  - TypeScript compiler configuration.
- **`tsconfig.app.json`**
  - Additional config file that allows to adjust configuration on an app basis. Useful when the workspace have multiple apps.
- **`tsconfig.spec.json`**
  - TypeScript configuration for the application tests.

Project Structure – the src folder

- **Includes all Angular components, templates, styles, images, and anything else the app needs**
- **`app`**
  - Contains the component files in which application logic and data are defined.
- **`assets`**
  - Contains image and other asset files to be copied as-is when the application is built.
- **`environment`**
  - Contains build configuration options for target environments. By default, there is an unnamed standard development environment and a production ("prod") environment.
- **`index.html`**
  - The main .HTML page that is served when someone visits the site. Angular automatically renders the root component when building the app.

Project Structure – the src/app folder

- src
  - app
    - app-routing.module.ts
    - app.component.css
    - app.component.html
    - app.component.spec.ts
    - app.component.ts
    - app.module.ts

- **All application components will be placed here (in separated folders).**
- **Each component has a TypeScript class, an HTML template, one or more stylesheets and a unit test file**
- **The AppComponent (root component) files and the app.module.ts file are also in this folder.**
  - The AppComponent is the root component of what will become a tree of nested components as the application evolves;
  - The app.module.ts file, defines AppModule, the root module that tells Angular how to assemble the application

Angular CLI – Command overview

COMMAND	DESCRIPTION
`ng add`	Adds support for an external library to the project
`ng analytics`	Configures the gathering of Angular CLI usage metrics
`ng build`	Compiles an Angular app into an output directory named **`dist/`** at the given output path. Must be executed from within a workspace directory.
`ng cache`	Configure persistent disk cache and retrieve cache statistics.
`ng completion`	Set up Angular CLI autocompletion for terminal.
`ng config`	Retrieves or sets Angular configuration values.
`ng deploy`	Invokes the deploy builder for a specified project or for the default project in the workspace
`ng doc`	Opens the official Angular documentation (angular.io) in a browser, and searches for a given keyword.
`ng e2e`	Builds and serves an Angular app, then runs end-to-end tests using Protractor.

Angular CLI – Command overview

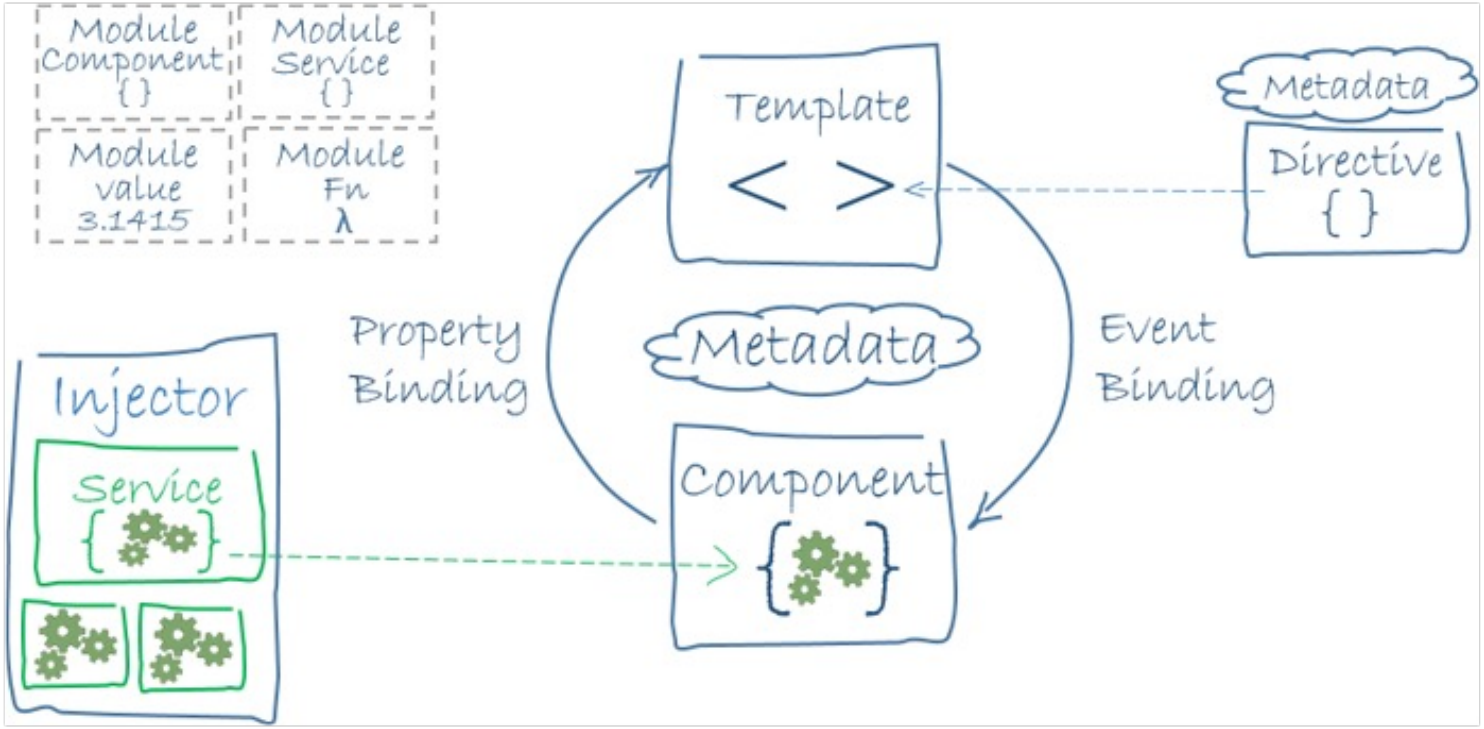
COMMAND	DESCRIPTION
`ng extract-i18n`	Extracts i18n messages from source code.
`ng generate`	Generates and/or modifies files based on a schematic.
`ng lint`	Runs linting tools on Angular app code in a project folder.
`ng new`	Creates a new workspace and an initial Angular app.
`ng run`	Runs a custom target defined in your project.
`ng serve`	Builds and serves the app, rebuilding on file changes.
`ng test`	Runs unit tests in a project.
`ng update`	Updates the application and its dependencies.
`ng version`	Outputs Angular CLI version.

First-party libraries

- **The Angular provides many first-party libraries that make it easy to add functionality to the application.**
- **Some of the available libraries are:**

LIBRARY	DETAILS
Angular Router	Advanced client-side navigation and routing based on Angular components. Supports lazy-loading, nested routes, custom path matching, and more.
Angular Forms	Uniform system for form participation and validation.
Angular HttpClient	Robust HTTP client that can power more advanced client-server communication.
Angular Animations	Rich system for driving animations based on application state.
Angular PWA	Tools for building Progressive Web Applications (PWAs) including a service worker and Web app manifest.
Angular Schematics	Automated scaffolding, refactoring, and update tools that simplify development at large scale.

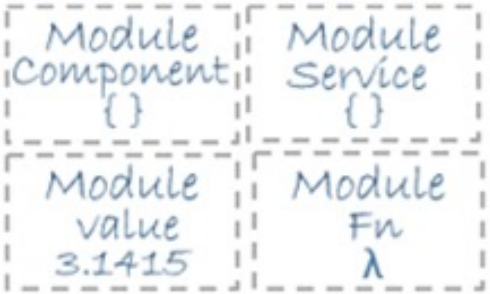
Angular Architecture



cf. https://angular.io/guide/architecture

Angular Architecture - Modules

- **The basic building blocks of an Angular application are NgModules, which provide a compilation context for components.**
- **An NgModule declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a closely related set of capabilities.**
- **Angular app has a root module, conventionally named AppModule, which provides the bootstrap mechanism that launches the application.**
- **Organizing the code into distinct functional modules helps in managing development and improve reusability.**



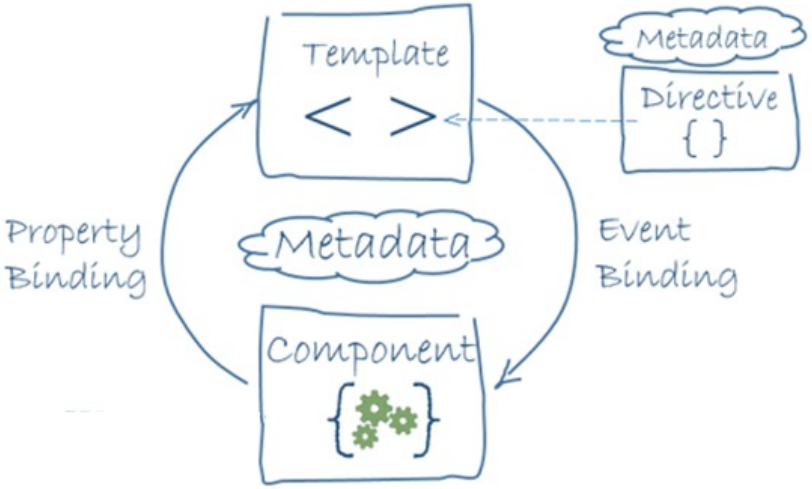
Angular Architecture - Components

- **Angular app has at least one component, the root component.**
  - The root component connects a component hierarchy with the page DOM
- **Components have application logic and define views to be displayed in a target environment.**
- **Components use services which provide specific functionality not directly related to view**
- **The `@Component()` decorator identifies the component TypeScript class**

isep Instituto Superior de Engenharia do Porto

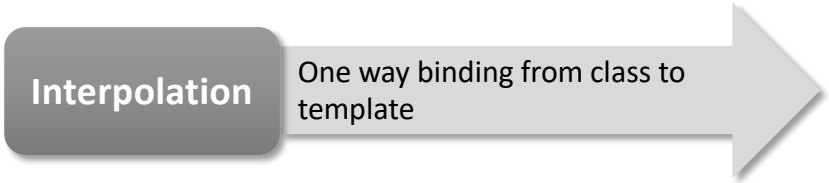
Angular Architecture - Templates

- **Components are defined with a HTML template that tells Angular how to render the component logic.**
- **A template combines HTML with Angular markup that can modify HTML elements before they are displayed**

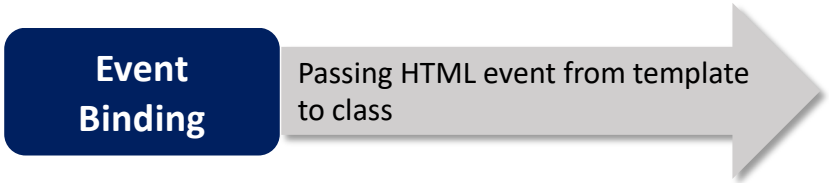


Angular Architecture - Data Binding

- **Three types of data binding:**
  - **Interpolation** binds the data one-way. This means that when value of the field bound using interpolation changes, it is updated in the page as well.
    - The syntax of interpolation is `{{varname}}`.



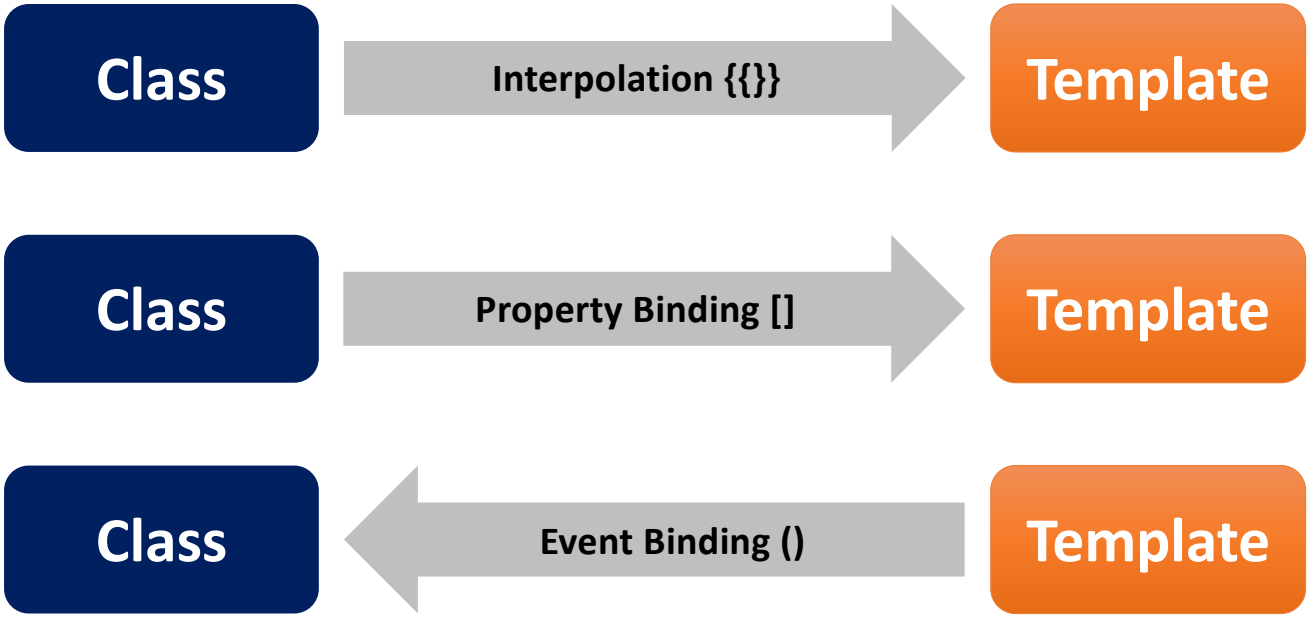
  - **Event binding** lets the application respond to user input in the target environment by updating application data.
    - The syntax behind Event Binding is `(eventname)`.



  - **Property binding** allows interpolation of values that are computed from application data into the HTML.
    - The syntax for this is `[propertyname]`.

Angular Architecture - Data Binding

- **In event binding, data flows from template to class**
- **In property and interpolation, data flows from class to template**



Angular Architecture - Data Binding

- **Binding example**

app.component.ts

```
(…)
export class AppComponent {

  title = 'This is a cool title';
  bHeight = 100;
  bWidth = 100;

  addProduct() {
     console.log('add product');
  }

}
(…)
```

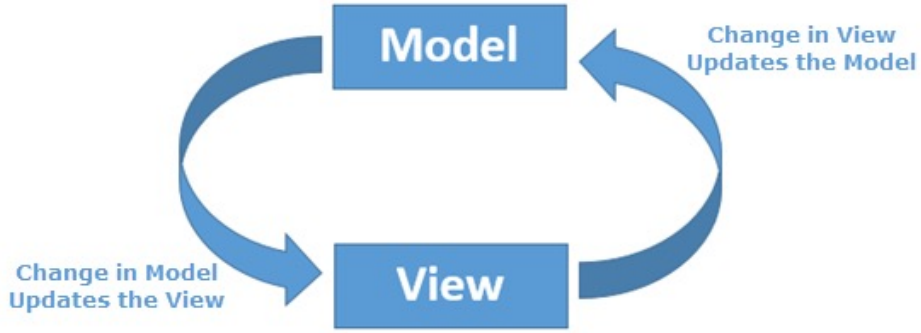
app.component.html

```
<h1>{{title}}</h1>

<button [style.height.px] = 'bHeight'
         [style.width.px] = 'bWidth'
                  (click) = 'addProduct()'>
    Add Product
</button >
```

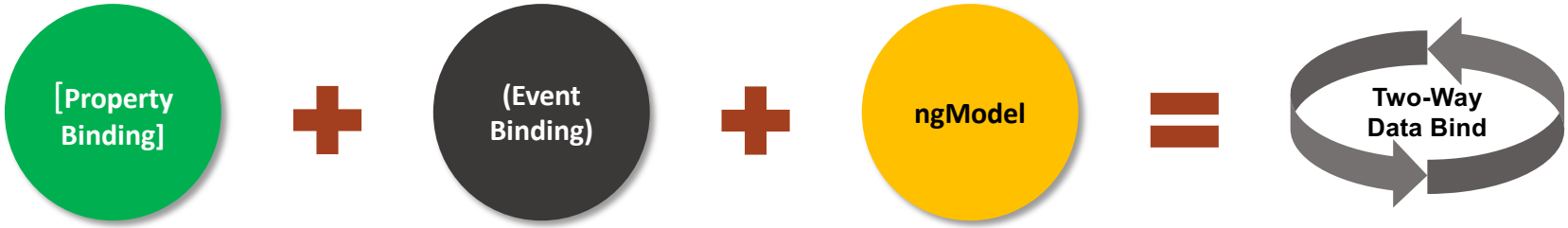
isep Instituto Superior de Engenharia do Porto

Angular Architecture - Data Binding

- **Angular does not have built-in Two-Way data binding.**



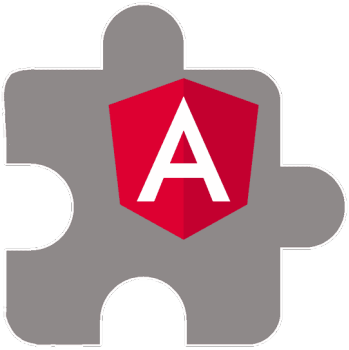
- **However, the combination use of Property Binding, Event Binding and the `ngModel` directive generates Two-Way data binding.**



Angular Components

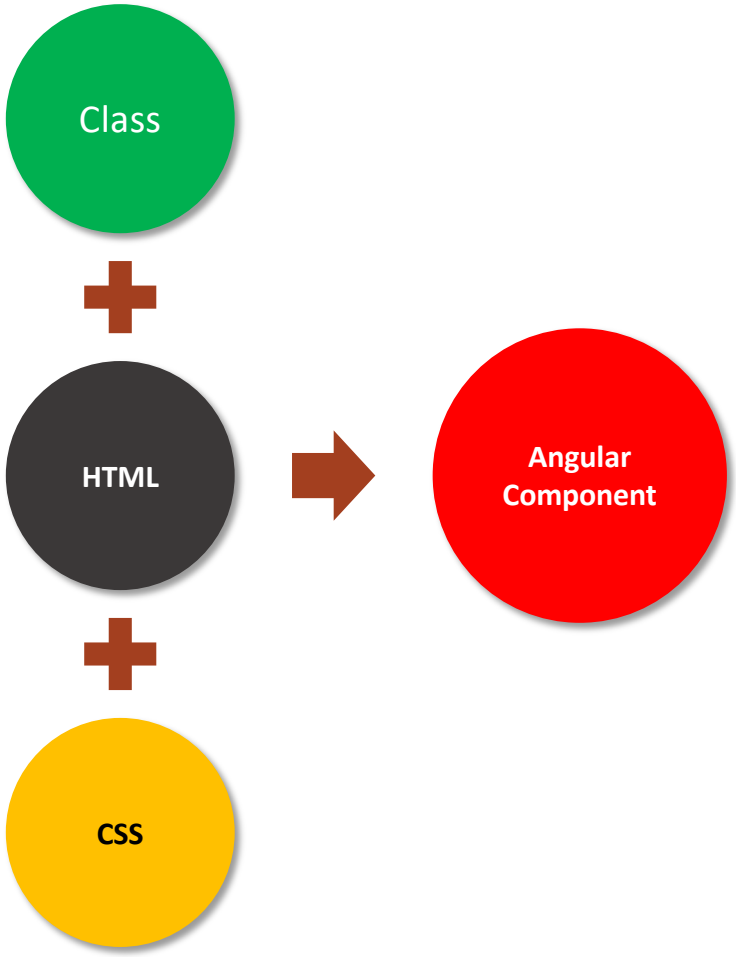
Angular Components

- **Components are the fundamental building blocks of Angular applications**
- **They:**
  - display data on the screen;
  - listen for user input;
  - take actions based on that input.
- **The AppComponent is the base component**
  - The shell is controlled by this Angular component



Angular Components

- **A component is distributed over three files**
  - `*.component.ts`
    - the component class code, written in TypeScript
  - `*.component.html`
    - the component template, written in HTML
  - `*.component.css`
    - the component's private CSS styles
- **Component files are located in a componente folder in `src/app`.**
- **It is good practice to create a folder to store all component folders.**



Create a new Component

- **Command:**

```
ng generate component component-name
```

- **Example:**

```
> ng generate component Components/CursoAngular
```

to comply with good practices.

- Run the command in the root of the project
  - The Angular CLI creates the new folder `src/app/Components/curso-angular/` and the three files related to CursoAngularComponent:
    - `curso-angular.component.ts` ← the component class code, written in TypeScript
    - `curso-angular.component.html` ← the component template, written in HTML
    - `curso-angular.component.css` ← the component's private CSS styles
  - Then updates the file `src/app/app.module.ts`

isep Instituto Superior de Engenharia do Porto

The Component Class file

- **The import statement allows to use classes defined elsewhere.**
- **In the example, the @Component decorator and the OnInit class are required for this component class.**

curso-angular.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-curso-angular',
  templateUrl: './curso-angular.component.html',
  styleUrls: ['./curso-angular.component.css']
})

export class CursoAngularComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

Decorators are a part of the typescript language.
They offer a simple way to add meta information to a class.
Decorators are marked by starting with an **@**.

isep Instituto Superior de Engenharia do Porto

The Component Class file

- **The `@Component` decorator marks a class as an Angular component and provides configuration metadata that determines how the component should be:**
  - processed,
  - Instantiated,
  - used at runtime.

curso-angular.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-curso-angular',
  templateUrl: './curso-angular.component.html',
  styleUrls: ['./curso-angular.component.css']
})

export class CursoAngularComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

The Component Class file

- **The selector matches the name of the HTML element that identifies the component within a parent component's template.**

curso-angular.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-curso-angular',
  templateUrl: './curso-angular.component.html',
  styleUrls: ['./curso-angular.component.css']
})

export class CursoAngularComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

Other.component.html

```
(…)
<app-curso-angular></app-curso-angular>
(…)
```

isep Instituto Superior de Engenharia do Porto

The Component Class file

- **The templateUrl is the module-relative address of this component's HTML template.**
- **This template defines the component's host view.**

curso-angular.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-curso-angular',
  templateUrl: './curso-angular.component.html',
  styleUrls: ['./curso-angular.component.css']
})

export class CursoAngularComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

isep Instituto Superior de Engenharia do Porto

The Component Class file

- **The styleUrls is an array module-relative address of the stylesheet files used to style the component view.**

curso-angular.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-curso-angular',
  templateUrl: './curso-angular.component.html',
  styleUrls: ['./curso-angular.component.css']
})

export class CursoAngularComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

isep Instituto Superior de Engenharia do Porto

The Component Class file

- **The component class must be exported so it can be imported elsewhere.**
  - ... like in `AppModule`.

curso-angular.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-curso-angular',
  templateUrl: './curso-angular.component.html',
  styleUrls: ['./curso-angular.component.css']
})

export class CursoAngularComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

app.module.ts

```
(…)
import { CursoAngularComponent } from
'./Components/curso-angular/curso-angular.componente';
(…)
```

isep Instituto Superior de Engenharia do Porto

The Component Class file

- **Constructor in Angular is used to inject dependencies into the component class.**
  - A dependency type constructor argument injects the dependencies.
- **It is a must to keep the Angular component constructor simple throughout.**
  - The reason is that simple logic in Angular constructor helps in easy operation of unit tests.

curso-angular.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-curso-angular',
  templateUrl: './curso-angular.component.html',
  styleUrls: ['./curso-angular.component.css']
})

export class CursoAngularComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

The Component Class file

- The `ngOnInit` is a lifecycle hook.
- Angular calls `ngOnInit` shortly after creating a component.
- It's a good place to put initialization logic…

curso-angular.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
  selector: 'app-curso-angular',
  templateUrl: './curso-angular.component.html',
  styleUrls: ['./curso-angular.component.css']
})

export class CursoAngularComponent implements OnInit {

  constructor() { }

  ngOnInit(): void {
  }

}
```

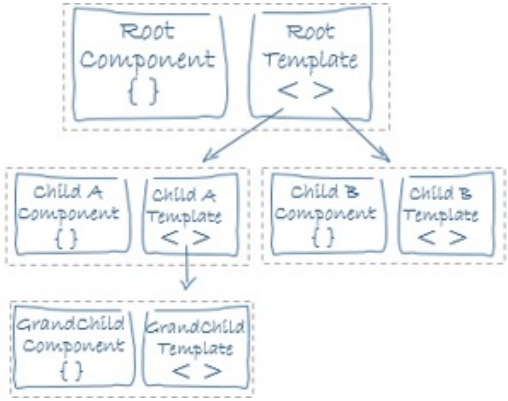
The Component Template file

- **A template is a form of HTML that tells Angular how to render the component.**

- **A template combines ordinary HTML with Angular directives and binding markup that allow Angular to modify the HTML before rendering it for display.**
- **A template defines a view.**
  - sets of screen elements that Angular can choose among and modify according to program logic and data

The Component Template file

- **Views are typically organized hierarchically, allowing to modify or show and hide entire UI sections or pages as a unit.**
- **The template immediately associated with a component defines that component's host view.**
- **The component can also define a view hierarchy, which contains embedded views, hosted by other components.**



The Component Template file

- **A template looks like regular HTML, except that it also contains Angular template syntax, which alters the HTML based on application's logic and the state of application and DOM data.**
- **The Component template can use**
  - data binding to coordinate the application
  - DOM data
  - pipes to transform data before it is displayed
  - directives to apply application logic to what gets displayed

The Component Template file

- **Template uses typical HTML elements like `<h1>` and `<p>`.**
- **It also includes Angular template-syntax elements:**
  - `*ngFor`
  - `{{session.number}}`
  - `(click)`
  - `[session]`
  - `<app-curso-detail>`
- **The template-syntax elements tell Angular how to render the HTML to the screen, using program logic and data**

curso-angular.component.html

```
<h1>Sessões:</h1>

<p><em>Selecione uma sessão para obter o seu plano.</em></p>
<ul>
  <li *ngFor="let session of sessions">
    <button type="button" (click)="selectSession(session)">
      {{session.number}} – {{session.topic}}
    </button>
  </li>
</ul>

<app-curso-detail
        *ngIf="selectedsession"
        [session]="selectedsession">
</app-curso-detail>
```

The Component Template file

- **The *ngFor directive tells Angular to iterate over a list.**

1 - TypeScript

2 – Angular Basics

3 – First Application

4 – Angular Structure

curso-angular.component.html

```
<h1>Sessões:</h1>

<p><em>Selecione uma sessão para obter o seu plano.</em></p>
<ul>
  <li *ngFor="let session of sessions">
    <button type="button" (click)="selectSession(session)">
      {{session.number}} – {{session.topic}}
    </button>
  </li>
</ul>

<app-curso-detail
        *ngIf="selectedsession"
        [session]="selectedsession">
</app-curso-detail>
```

isep Instituto Superior de Engenharia do Porto

The Component Template file

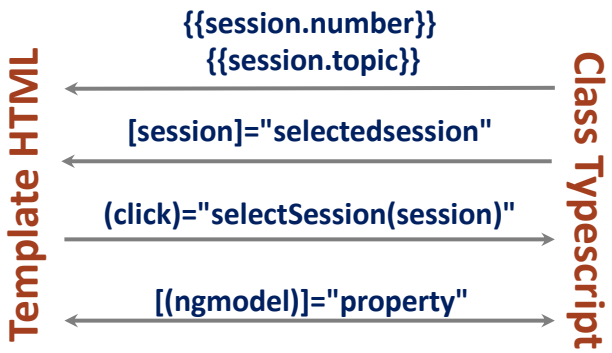
- **The `*ngIf` directive conditionally shows or hides content based on the value of a boolean expression.**

curso-angular.component.html

```
<h1>Sessões:</h1>

<p><em>Selecione uma sessão para obter o seu plano.</em></p>
<ul>
  <li *ngFor="let session of sessions">
    <button type="button" (click)="selectSession(session)">
      {{session.number}} – {{session.topic}}
    </button>
  </li>
</ul>

<app-curso-detail
        *ngIf="selectedsession"
        [session]="selectedsession">
</app-curso-detail>
```

The Component Template file

- `(click)`, `[session]`, `{{session.number}}` and `{{session.topic}}` bind program data to and from the DOM, responding to user input.



cf. https://angular.io/guide/architecture

curso-angular.component.html

```
<h1>Sessões:</h1>

<p><em>Selecione uma sessão para obter o seu plano.</em></p>
<ul>
  <li *ngFor="let session of sessions">
    <button type="button" (click)="selectSession(session)">
      {{session.number}} - {{session.topic}}
    </button>
  </li>
</ul>

<app-curso-detail
        *ngIf="selectedsession"
        [session]="selectedsession">
</app-curso-detail>
```

The Component Template file

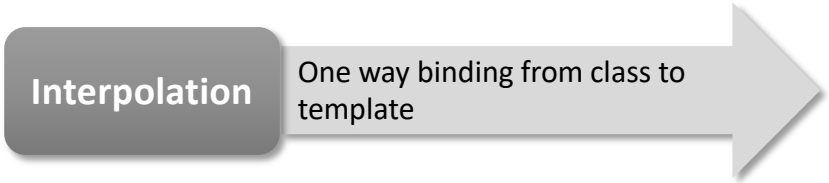
- **The `<app-curso-detail>` element tag represents the, CursoDetailComponent component.**
  - The *CursoDetailComponent* defines the curso-detail portion of the rendered DOM structure specified by the *CursoAngularComponent* component.

curso-angular.component.html

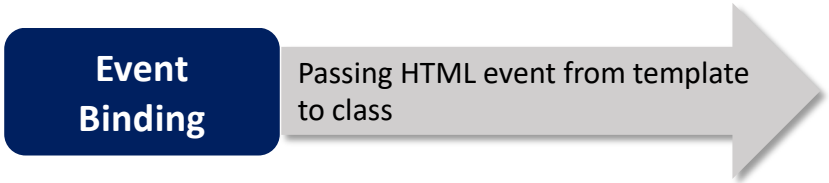
```
<h1>Sessões:</h1>

<p><em>Selecione uma sessão para obter o detalhe.</em></p>
<ul>
  <li *ngFor="let session of sessions">
    <button type="button" (click)="selectSession(session)">
      {{session.number}} – {{session.topic}}
    </button>
  </li>
</ul>

<app-curso-detail
        *ngIf="selectedsession"
        [session]="selectedsession">
</app-curso-detail>
```

Data Binding

- **Three types of data binding:**
  - **Interpolation** binds the data one-way. This means that when value of the field bound using interpolation changes, it is updated in the page as well.
    - The syntax of interpolation is `{{varname}}`.



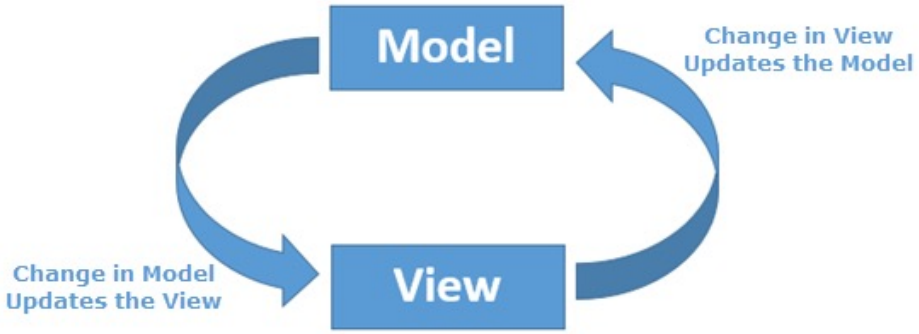
  - **Event binding** lets the application respond to user input in the target environment by updating application data.
    - The syntax behind Event Binding is `(eventname)`.



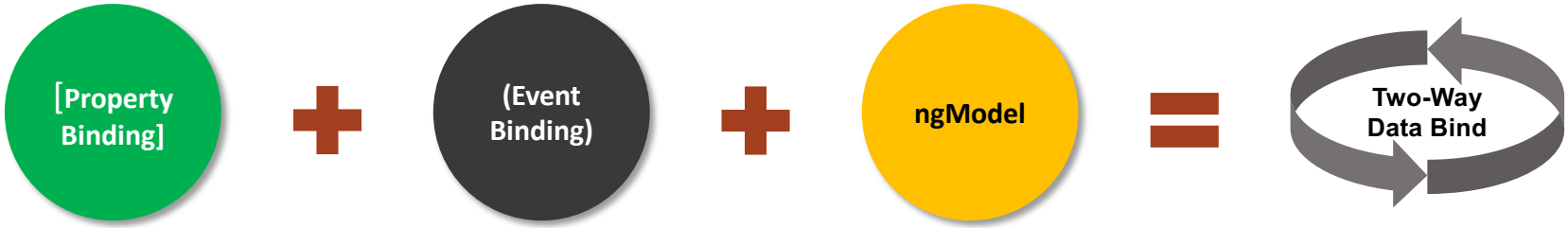
  - **Property binding** allows interpolation of values that are computed from application data into the HTML.
    - The syntax for this is `[propertyname]`.

Data Binding

- **Angular does not have built-in Two-Way data binding.**

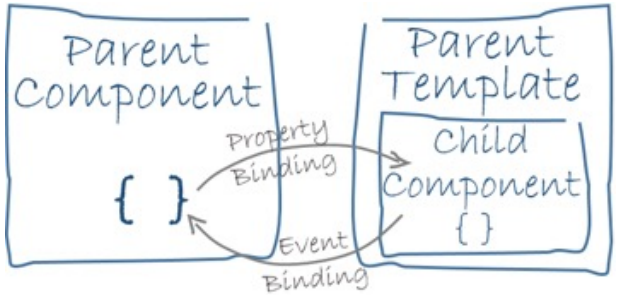


- **However, the combination use of Property Binding, Event Binding and the `ngModel` directive generates Two-Way data binding.**



Data binding

- **Data binding plays an important role in communication between a template and its component and is also important for communication between parent and child components.**



- **This feature is implemented using the `@Input()` and `@Output()` decorators.**

isep Instituto Superior de Engenharia do Porto

Sending data to a child component

- **The `@Input()` decorator must be imported from Angular core library.**
- **The `@Input()` decorator in a child component or directive signifies that the property can receive its value from its parent component.**

curso-detail.component.ts

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'curso-detail-angular',
  templateUrl: './curso-detail.component.html',
  styleUrls: ['./curso-detail.component.css']
})

export class CursoDetailComponent implements OnInit {
  @Input() session? : Session;

  constructor() { }

  ngOnInit(): void {
  }

}
```

Sending data to a child component

- **The session property must be bonded in the parent's component.**

curso-detail.component.ts

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'curso-detail-angular',
  templateUrl: './curso-detail.component.html',
  styleUrls: ['./curso-detail.component.css']
})

export class CursoDetailComponent implements OnInit {
  @Input() session? : Session;

  constructor() { }

  ngOnInit(): void {
  }

}
```

TARGET.
@Input() property from child component.

child component selector.

SOURCE.
Property from parent component.

curso-angular.component.html

```
<curso-detail-angular [session]="selectedSession">
</curso-detail-angular>
```

Sending data to a child component

- **The parent component class handles `selectSession`**

curso-angular.component.ts

```
(…)
export class CursoAngularComponent {

  selectedSession?: Session:

  onSelect(session: Session): void {
    this.selectedSession = session;
  }

  (…)
}
```

curso-detail.component.ts

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'curso-detail-angular',
  templateUrl: './curso-detail.component.html',
  styleUrls: ['./curso-detail.component.css']
})

export class CursoDetailComponent implements OnInit {
  @Input() session? : Session;

  constructor() { }

  ngOnInit(): void {
  }

}
```

curso-angular.component.html

```
<curso-detail-angular [session]="selectedSession">
</curso-detail-angular>
```

isep Instituto Superior de Engenharia do Porto

Sending data to a parent component

- The `@Output()` decorator must be imported from Angular core library.
- The `@Output()` decorator marks a property in a child component as a doorway through which data can travel from the child to the parent.
- The child component uses the @Output() property to raise an event to notify the parent of the change.

curso-output.component.ts

```
import {Component, Output, EventEmitter} from '@angular/core';

@Component({
  selector: 'curso-output',
  templateUrl: './curso-output.component.html',
  styleUrls: ['./curso-output.component.css']
})

export class CursoOutputComponent implements OnInit {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
  (…)
}
```

Sending data to a parent component

- **To raise an event, the @Output() directive must have the type of `EventEmitter`, which is a class in @angular/core.**

curso-output.component.ts

```
import {Component, Output, EventEmitter} from '@angular/core';

@Component({
  selector: 'curso-output',
  templateUrl: './curso-output.component.html',
  styleUrls: ['./curso-output.component.css']
})

export class CursoOutputComponent implements OnInit {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
  (…)
}
```

Sending data to a parent component

- **The @Output() decorated property is an event.**

The decorator makes the property a way for data to pass from the child to the parent.

Tells Angular to create a new event emitter and that the data it emits is of type string.

curso-output.component.ts

```
import {Component, Output, EventEmitter} from '@angular/core';

@Component({
  selector: 'curso-output',
  templateUrl: './curso-output.component.html',
  styleUrls: ['./curso-output.component.css']
})

export class CursoOutputComponent implements OnInit {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
  (…)
}
```

isep Instituto Superior de Engenharia do Porto

Sending data to a parent component

- **The `addNewItem()` function uses the `newItemEvent`, to raise an event with the value the user types into the `<input>`.**

curso-output.component.ts

```
import {Component, Output, EventEmitter} from '@angular/core';

@Component({
  selector: 'curso-output',
  templateUrl: './curso-output.component.html',
  styleUrls: ['./curso-output.component.css']
})

export class CursoOutputComponent implements OnInit {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
  (…)
}
```

curso-output.component.html

```
<label for="item-input">Add an item:</label>
<input type="text" id="item-input" #newItem>
<button type="button" (click)="addNewItem(newItem.value)">Add
to parent's list</button
```

isep Instituto Superior de Engenharia do Porto

Sending data to a parent component

- **The parent's template binds the parent's method to the child's event**

app.component.html

```
(…)
<curso-output (newItemEvent)="addItem($event)">
</curso-output>
```

The event binding connects the event in the child, `newItemEvent`, to the method in the parent, `addItem()`.

The `$event` contains the data that the user types into the <input> in the child template UI.

`$` suffix is used to indicate that the variable is an Observable.

curso-output.component.ts

```
import {Component, Output, EventEmitter} from '@angular/core';

@Component({
  selector: 'curso-output',
  templateUrl: './curso-output.component.html',
  styleUrls: ['./curso-output.component.css']
})

export class CursoOutputComponent implements OnInit {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
  (…)
}
```

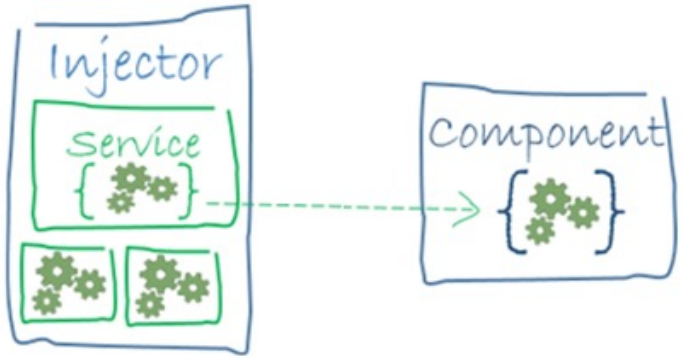
curso-output.component.html

```
<label for="item-input">Add an item:</label>
<input type="text" id="item-input" #newItem>
<button type="button" (click)="addNewItem(newItem.value)">Add
to parent's list</button
```

isep Instituto Superior de Engenharia do Porto

Angular Architecture - Services

- **Is used to encapsulate data or logic that isn't associated with a specific view and is intended to be shared across components.**
- **A service is a class with a narrow, well-defined purpose.**
- **A service class definition is preceded by the `@Injectable()` decorator.**
  - The decorator provides the metadata that allows the service to be injected into client components as a dependency.



Angular Services

Angular Services

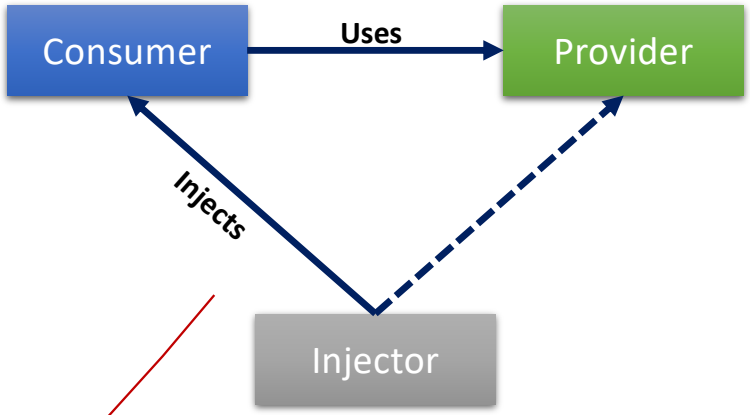
- **Angular Services encourage code and functionality reuse across the Angular project components.**
- **Services helps with the abstraction of logic and data that is hosted independently but can be shared across other components.**
- **Angular distinguishes components from services to increase modularity and reusability.**
  - A component should use services for tasks that don't involve the view or application logic;
  - Services are good for tasks such as fetching data from the server, validating user input, or logging directly to the console.
- **The Angular Services are available to any component because they are injected into the root provider.**

Dependency Injection (DI)

- **Dependency injection is a design pattern that makes a class independent of its dependencies.**
- **It allows the creation of dependent objects outside of a class and provides those objects to the class**
- **It achieves that by decoupling the usage of an object from its creation. This helps to follow SOLID's dependency inversion and single responsibility principles.**
  - The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.
  - The Single Responsibility Principle (SRP) states that every class, module, or function in a program should have one, and only one, responsibility/purpose.

Dependency Injection (DI)

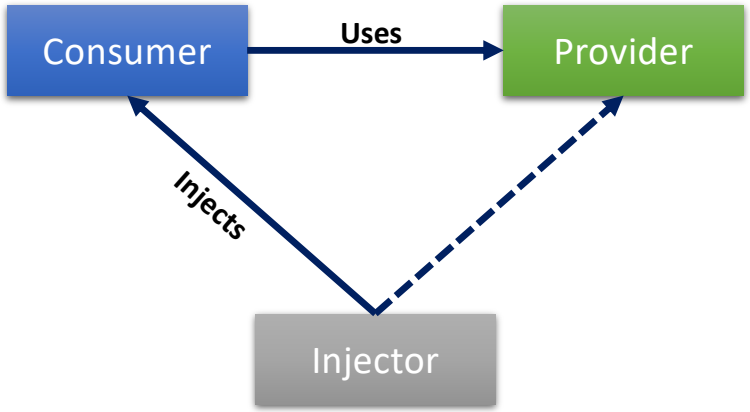
- **The DI pattern involves 3 types of classes:**
  - Consumer Class: The consumer class is a class which depends on the provider class
  - Provider Class: This class provides service to the consumer class.
  - Injector Class: The injector class injects the provider class object into the consumer class.
- **The DI pattern separates the responsibility of creating an object of the service class out of the client class**



The injector class creates an object of the provider class and injects that object to a consumer object.

Dependency Injection (DI)

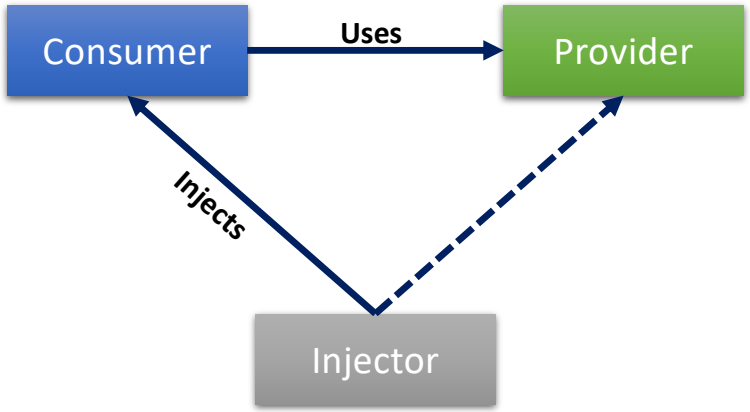
- **Dependency injection (DI) is the part of the Angular framework that provides Components with access to services and other resources.**
- **Angular facilitates the interaction between dependency consumers and dependency providers using an abstraction called Injector.**



isep Instituto Superior de Engenharia do Porto

Dependency Injection (DI)

- **When a dependency is requested, the injector checks its registry to see if there is an instance already available there.**
- **If not, a new instance is created and stored in the registry.**
- **Angular creates an application-wide injector (also known as "`root`" injector) during the application bootstrap process, as well as any other injectors as needed.**
  - Although in most cases there is no need to create injectors manually there is a layer that connects providers and consumers.



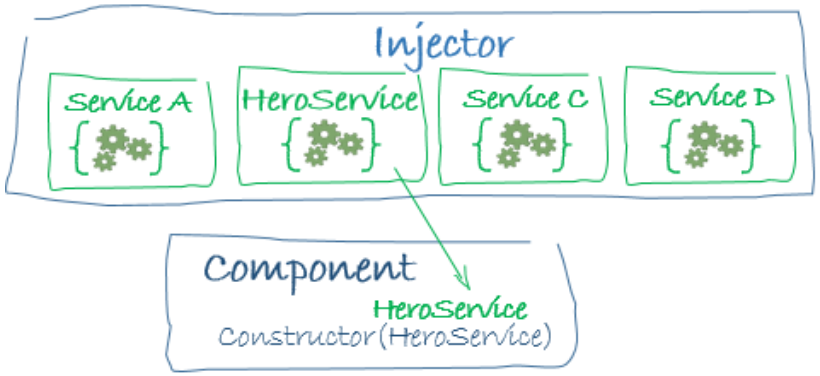
isep Instituto Superior de Engenharia do Porto

Providing dependency

- **The `@Injectable()` decorator defines a class as a service and allows Angular to inject it into a component as a dependency.**

HeroService.service.ts

```
import { Injectable } from '@angular/core';

@Injectable()
export class HeroService {

  (…)

}
```



Providing dependency

- **A dependency can be provided in multiple places:**
  - At the **Component level**, using the providers field of the `@Component` decorator.

```
@Component({
  selector: 'hero-list',
  template: '...',
  providers: [HeroService]
})
class HeroListComponent {}
```

In this case the HeroService becomes available to all instances of this component and other components and directives used in the template.

isep Instituto Superior de Engenharia do Porto

Providing dependency

- **A dependency can be provided in multiple places:**
  - At the **NgModule level**, using the providers field of the `@NgModule` decorator.

```
@NgModule({
  declarations: [HeroListComponent]
  providers: [HeroService]
})
class HeroListModule {}
```

In this scenario, the HeroService is available to all components, directives and pipes declared in this NgModule

isep Instituto Superior de Engenharia do Porto

Providing dependency

- **A dependency can be provided in multiple places:**
  - At the **application root level**.
    - Allows injecting into other classes in the application.

HeroService.service.ts

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class HeroService {

  (…)

}
```

With this solution, Angular creates a single, shared instance of the *HeroService* and injects it into any class that asks for it

Injecting a dependency

- **The most common way to inject a dependency is to declare it in a class constructor.**
  - When Angular creates a new instance of a component class, it determines which services or other dependencies that component needs by looking at the constructor parameter types.

HeroList.component.ts

```
import { Component, OnInit } from '@angular/core';
Import { HeroService } from './hero.service.ts';

@Component({
  selector: 'app-hero-list-component',
  templateUrl: './hero-list-component.component.html',
  styleUrls: ['./hero-list-component.component.css']
})
export class HeroListComponent implements OnInit {

  constructor(private service: HeroService) { }

  ngOnInit(): void {
  }
}
```

When Angular discovers that a component depends on a service, it first checks if the injector has any existing instances of that service.
If a requested service instance doesn't yet exist, the injector creates one using the registered provider, and adds it to the injector before returning the service to Angular.
When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments.

Injecting a dependency

- **Another way to inject a dependency is to use the `@Inject()` decorator.**
- **`@Inject()` is used to manually identity dependencies and the token an injector should use for this dependency.**
  - This approach should be avoided because the code can get messy.
  - Tokens are keys used to reference a Provider.

HeroList.component.ts

```
import { Component, OnInit } from '@angular/core';
Import { HeroService } from './hero.service.ts';

@Component({
        (…)
})
export class HeroListComponent implements OnInit {

  constructor(@Inject(HERO_SERVICE_TOKEN) private service: HeroService) { }

  ngOnInit(): void {
  }
}
```

This approach explicitly tells Angular that it should use a specific provider factory function to create this dependency.

isep Instituto Superior de Engenharia do Porto

How does the injector know what is needed?

- **Costumer must put its need**
  - It can do so by requesting the need (dependency) in its constructor

coffeCustomer.ts

```
(…)
export class coffeCustomer() {

  constructor(coffee: CoffeService) { }

  (…)
}
```

Here, the injector uses *CoffeeService* type as the token for looking up *coffee*.

This can be compared to the waiter jotting down *"1 cap"* for *"1 cup of cappuccino"* in the notebook

  - Injector collects all orders from Angular application (create a map of any internal dependencies)
    - The DI tokens act as a key to that map.
    - The dependency value is an instance, and the class type serves as a lookup key.

Angular Services

- **The `@Injectable()` decorator must be imported from Angular core library.**
- **The `@Injectable()` decorator marks the class as member of the dependency injection system.**

data.service.ts

```
import { Injectable } from '@angular/core’;

@Injectable({
  providedIn: 'root'
})

export class DataService {
constructor() { }
getList(){
    return[
      {'grade':1, 'name':'Davido', 'country':'Nigeria'},
      {'grade':2, 'name':'Burna', 'country':'Nigeria'},
      {'grade':3, 'name':'Diamoz', 'country':'Tanzania'},
      {'grade':4, 'name':'Sarkodie', 'country':'Ghana'},
      {'grade':5, 'name':'Mr. Eazi', 'country':'Nigeria'}
    ];
  }
}
```

Angular Services

- **The registered provider is the root injector.**
- **When the service is provided at the root level, Angular creates a single, shared instance of the service (DataService) and injects into any class that asks for it.**
- **Registering the provider in the `@Injectable` decorator also allows Angular to optimize an application by removing the service if it isn't used.**

data.service.ts

```
import { Injectable } from '@angular/core’;

@Injectable({
  providedIn: 'root'
})

export class DataService {
constructor() { }
getList(){
    return[
      {'grade':1, 'name':'Davido', 'country':'Nigeria'},
      {'grade':2, 'name':'Burna', 'country':'Nigeria'},
      {'grade':3, 'name':'Diamoz', 'country':'Tanzania'},
      {'grade':4, 'name':'Sarkodie', 'country':'Ghana'},
      {'grade':5, 'name':'Mr. Eazi', 'country':'Nigeria'}
    ];
  }
}
```

Angular Services

- **The service class must be exported so it can be imported elsewhere.**
  - ... like in a component class.

the.component.ts

```
import { Component, OnInit } from '@angular/core';

import { DataService } from '../data.service';

@Component({
  selector: 'app-the',
  templateUrl: ‘./the.component.html',
  styleUrls: [‘./the.component.css']
})

(…)
```

data.service.ts

```
import { Injectable } from '@angular/core’;

@Injectable({
  providedIn: 'root'
})

export class DataService {
constructor() { }
getList(){
    return[
      {'grade':1, 'name':'Davido', 'country':'Nigeria'},
      {'grade':2, 'name':'Burna', 'country':'Nigeria'},
      {'grade':3, 'name':'Diamoz', 'country':'Tanzania'},
      {'grade':4, 'name':'Sarkodie', 'country':'Ghana'},
      {'grade':5, 'name':'Mr. Eazi', 'country':'Nigeria'}
    ];
  }
}
```

Angular Services

- **To inject the service in the component, add a service type parameter to the constructor.**

the.component.ts

```
import { Component, OnInit } from '@angular/core';

import { DataService } from '../data.service';

@Component(
(…)
)
export class TheComponent implements OnInit {

  constructor(private datSrv: DataService) { }

(…)
}
```

data.service.ts

```
import { Injectable } from '@angular/core’;

@Injectable({
  providedIn: 'root'
})

export class DataService {
constructor() { }
getList(){
    return[
      {'grade':1, 'name':'Davido', 'country':'Nigeria'},
      {'grade':2, 'name':'Burna', 'country':'Nigeria'},
      {'grade':3, 'name':'Diamoz', 'country':'Tanzania'},
      {'grade':4, 'name':'Sarkodie', 'country':'Ghana'},
      {'grade':5, 'name':'Mr. Eazi', 'country':'Nigeria'}
    ];
  }
}
```

Angular Services

- **To consume the service, call the service's business method.**

the.component.ts

```
import { Component, OnInit } from '@angular/core';
import { DataService } from '../data.service';

@Component(
(…)
)
export class TheComponent implements OnInit {

  constructor(private datSrv: DataService) { }

  getListFromService(){
        this.data = this.datSrv.getList();
  }
(…)
}
```

data.service.ts

```
import { Injectable } from '@angular/core’;

@Injectable({
  providedIn: 'root'
})

export class DataService {
constructor() { }
getList(){
    return[
      {'grade':1, 'name':'Davido', 'country':'Nigeria'},
      {'grade':2, 'name':'Burna', 'country':'Nigeria'},
      {'grade':3, 'name':'Diamoz', 'country':'Tanzania'},
      {'grade':4, 'name':'Sarkodie', 'country':'Ghana'},
      {'grade':5, 'name':'Mr. Eazi', 'country':'Nigeria'}
    ];
  }
}
```

Angular Modules

What are Angular Modules?

- **NgModules are containers for a cohesive block of code dedicated to an application domain or workflow and with a closely related set of capabilities.**
- **They can contain components, service providers, and other code files whose scope is defined by the containing NgModule.**
- **They can import functionality that is exported from other NgModules, and export selected functionality for use by other NgModules.**
- **Angular application has at least one NgModule class.**
  - The root module, named AppModule (`app.module.ts`).
  - The AppModule defines a NgModule hierarchy of any depth.

Angular modularity

- **Modules are a great way to organize an application and extend it with capabilities from external libraries.**
  - Angular libraries are NgModules, such as `FormsModule`, `HttpClientModule`, and `RouterModule`.
  - Many third-party libraries are available as NgModules such as *Material Design*.
- **Modules can also add services to the application.**
  - Such services might be internally developed or come from outside sources, such as the Angular router and HTTP client.
- **Modules can be loaded eagerly when the application starts or lazy loaded asynchronously by the router.**

NgModule metadata

- **NgModule is defined by a class decorated with `@NgModule()`.**
- **This decorator is a function that takes a single metadata object whose properties describe the module. The most relevant properties are:**

Properties	Details
`declarations`	The components, directives, and pipes that belong to this NgModule.
`exports`	The subset of declarations that should be visible and usable in the component templates of other NgModules.
`imports`	Other modules whose exported classes are needed by component templates declared in this NgModule.
`providers`	Service providers supplied by NgModule for the global collection of services. They become accessible from all parts of the application.
`bootstrap`	The main application view (called the root component, AppComponent) which hosts all other application views. Only the root NgModule should set the bootstrap property.

The basic NgModule

- **The Angular CLI generates the following basic AppModule when creating a new application.**

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Feature Modules

- **Aim to organize the code.**
  - Helps enforce clear boundaries for features (keep code related to a specific functionality or feature separate from other code);
  - Allows to delineate intervention areas in the collaboration between developers and teams, separating directives and managing the size of the root module.
- **The feature module collaborates with the root module and with other modules through the services it provides and the components, directives, and pipes that it shares.**

Create a Feature Module

- **Using the Angular CLI, run the following command:**

to comply with good practices.

HeroDashboard is the name of the module

```
> ng generate module Modules/HeroDashboard
CREATE src/app/Modules/hero-dashboard/hero-dashboard.module.ts (199 bytes)
```

hero-dashboard folder is created.

hero-dashboard.module.ts is the module class.

- The hero-dashboard.module.ts has the following contents:

hero-dashboard.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  declarations: [],
  imports: [
    CommonModule
  ]
})
export class HeroDashboardModule { }
```

Exports all the basic Angular directives and pipes, such as **NgIf**, **NgFor, NgStyle**, **DecimalPipe**, …

The declarations array is available to add declarables (components, directives, and pipes) that belong exclusively to this module.

Create a Feature Module

- **Add a component to the Feature Module :**

Create component inside the Module folder.

```
> ng generate component Modules/hero-dashboard/HeroDashboard
CREATE src/app/Modules/hero-dashboard/hero-dashboard/hero-dashboard.component.html (…)
CREATE src/app/Modules/hero-dashboard/hero-dashboard/hero-dashboard.component.spec.ts (…)
CREATE src/app/Modules/hero-dashboard/hero-dashboard/hero-dashboard.component.ts (…)
CREATE src/app/Modules/hero-dashboard/hero-dashboard/hero-dashboard.component.css (…)
UPDATE src/app/Modules/hero-dashboard/hero-dashboard.module.ts (…)
```

- This generates the component folder and files, and updates the Feature Module:

hero-dashboard.module.ts

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { HeroDashboardComponent } from './hero-dashboard/hero-dashboard.component'

@NgModule({
  declarations: [ HeroDashboardComponent ],
  imports: [ CommonModule ]
})

export class HeroDashboardModule { }
```

isep Instituto Superior de Engenharia do Porto

Importing a Feature Module

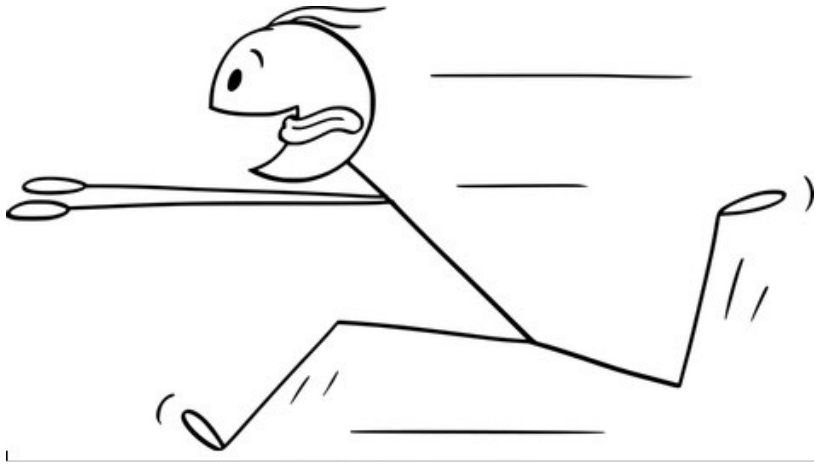
- **To use the Feature Module, AppModule needs to know about it.**
  - Import it in the AppModule and add it to the `imports` array

app.module.ts

```
(…)
import { HeroDashboardModule } from './Modules/hero-dashboard/hero-dashboard.module';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HeroDashboardModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Lazy loading Feature Modules

- **By default, NgModules are eagerly loaded.**
  - Once the app is loaded, all NgModules will be loaded as well, whether they are needed immediately or not.
- **For large applications with lots of routes, consider lazy loading.**
  - Design pattern that loads NgModules as needed.
  - Lazy loading helps keep initial bundle sizes smaller, which in turn helps decrease load times.



Lazy loading Feature Modules

- **Using the Angular CLI, run the following command:**

```
> ng generate module Modules/LazyHeroes --route LazyRoute --module app.module
CREATE src/app/Modules/lazy-heroes/lazy-heroes-routing.module.ts (360 bytes)
CREATE src/app/Modules/lazy-heroes/lazy-heroes.module.ts (387 bytes)
CREATE src/app/Modules/lazy-heroes/lazy-heroes.component.html (26 bytes)
CREATE src/app/Modules/lazy-heroes/lazy-heroes.component.spec.ts (628 bytes)
CREATE src/app/Modules/lazy-heroes/lazy-heroes.component.ts (294 bytes)
CREATE src/app/Modules/lazy-heroes/lazy-heroes.component.css (0 bytes)
UPDATE src/app/app-routing.module.ts (1069 bytes)
```

- This command:
  - will generate a lazy-loadable Feature Module with routing enabled and a component to route to.
  - automatically declares the `LazyHeroes` component and imports `LazyHeroesRoutingModule` inside the new feature module.
  - **does not** add a reference to the new feature module in the `AppModule` (the new module is meant to be lazy-loaded).
  - adds the declared route (LazyRoute) to the routes array declared in `app-routing.module.ts`.

Lazy loading Feature Modules

- **The AppRoutingModule file**

app-routing.module.ts

```
(…)

const routes: Routes = [
  (…)
  { path: 'LazyRoute',
    loadChildren: () =>
         import('./Modules/lazy-heroes/lazy-heroes.module').then(m => m.LazyHeroesModule) }
];

(…)
```

Lazy-loading syntax uses loadChildren followed by a function that uses the import('...') syntax for dynamic imports.

when route 'LazyRoute' is visited, the respective module is load lazily.

isep Instituto Superior de Engenharia do Porto

Providing dependencies in modules

- **When a service provider is added to the root application injector, it is available app-wide.**
  - This provider is also available to all the classes in the application as long they have the lookup token.
- **A service must always be provided on the root injector unless it is required that the service be available in the context of a given `@NgModule`.**
  - if you don't want the service to be available to applications unless they import a certain module, you can specify that the service should be provided in the module.

the.service.ts

```
import { Injectable } from '@angular/core';
import { TheModule } from './the.module';

@Injectable({
  providedIn: TheModule
})
export class TheService {
}
```

Providing dependencies in modules

- **The preferred way to provide a service in a module is to explicitly refer to the module in the `providedIn` property of the `@Injectable()` decorator.**
  - This method is preferred because it allows the service to tree-shake if nothing injects it.
- **If it is not possible to specify in the service which module should provide it, a provider for the service can be declared within the module:**

the.module.ts

```
import { NgModule } from '@angular/core';
import { TheService } from './the.service';

@NgModule({
  providers: [UserService]
})
export class UserModule {
}
```

isep Instituto Superior de Engenharia do Porto

Limiting provider scope by lazy loading modules

- **In lazy loading Modules, any services listed in their provider arrays aren't available because the root injector doesn't know about these modules.**
- **When the Angular router lazy-loads a module, it creates a new injector.**
  - This injector is a child of the root application injector (there is a single root injector and then a child injector for each lazy loaded module).
  - The child injector gets populated with all the module-specific providers.
- **Any component created within a lazy loaded module's context, such as by router navigation, gets its own local instance of child provided services.**
  - not the instance in the root application injector.

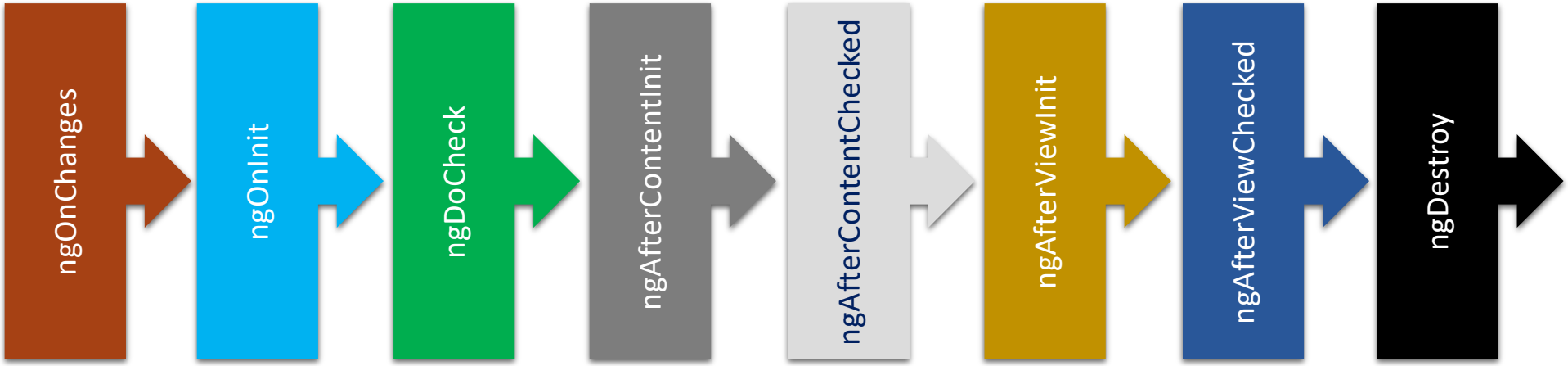
Lifecycle hooks

Angular lifecycle

- **A component in Angular goes through several different phases from creation to destruction.**
  - The lifecycle of a component instance starts when Angular instantiates the component class and renders the component view along with its child views;
  - The lifecycle continues with change detection, as Angular checks to see when data-bound properties change, and updates both the view and the component instance as needed;
  - The lifecycle ends when Angular destroys the component instance and removes its rendered template from the DOM.
- **Directives have a similar lifecycle, as Angular creates, updates, and destroys instances during execution.**

Lifecycle hook methods

- **An interface that allows to tap into the lifecycle of directives and components as they are created, updated, and destroyed.**
- **Each interface has a single hook method whose name is the interface name prefixed with *ng*.**
  - For example, the `OnInit` interface has a hook method named `ngOnInit`.
- **Angular runs these hook methods in the following order:**



Lifecycle hook methods

#	Hook Method	Details
1	`ngOnChanges`	When an input or output binding value changes.
2	`ngOnInit`	After the first ngOnChanges.
3	`ngDoCheck`	Developer's custom change detection.
4	`ngAfterContentInit`	After component content initialized.
5	`ngAfterContentChecked`	After every check of component content.
6	`ngAfterViewInit`	After the views of a component are initialized.
7	`ngAfterViewChecked`	After every check of the views of a component.
8	`ngOnDestroy`	Just before the directive is destroyed.

Angular Routing

Routing

- **Routing in Angular allows the users to create a single-page application (SPA) with multiple views and allows navigation between them.**
  - Users can switch between these views without losing the application state and properties.
  - This switching of views is accomplished by showing or hiding parts of the screen that correspond to specific components, rather than going out to the server to get a new page.
- **To handle the navigation from one view to the next, Angular Router is used.**
  - The Router enables navigation by interpreting a browser URL as an instruction to change the view.

isep Instituto Superior de Engenharia do Porto

Generate an application with routing enabled

- **Using the Angular CLI, run the following command:**

```
> ng new a-new-routing-app --routing --defaults
```

The name of the new workspace and initial project

Generate a routing module (**AppRoutingModule**) for the project.

Disable interactive input prompts for options with a default

The basic Routing Module

The Angular CLI automatically imports ***RouterModule*** and ***Routes*** from @angular/router.

The Angular CLI sets up a Routes array for the application routes.
This array collects a set of routes.
The router attempts to match segments of a given URL against each route, using the configuration options defined in this object

The Angular CLI automatically configures the imports and exports arrays for @NgModule().

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

The **forRoot()** method creates and configures a module with all the router providers and directives. Optionally sets up an application listener to perform an initial navigation

isep Instituto Superior de Engenharia do Porto

Routes definitions

- **A route definition is a JSON object.**
- **Each route, typically, has two properties.**
  - The first property, path, is a string that specifies the URL path for the route.
  - The second property, component, is a string that specifies what component the application should display for that path.

app-routing.module.ts

```
(…)

const routes: Routes = [
        {path: 'crisis-list', component: CrisisListComponent},
        {path: 'heroes-list', component: HeroesListComponent}
];

(…)
```

path
Property that describes the URL this route will handle. For example, if **path: 'xpto'**, the handled URL will be something like ***myapp.com/xpto***

component
This is the component that the router should create when navigating to this route

isep Instituto Superior de Engenharia do Porto

Routes definitions

- **Create a default route**
  - To set up a redirect, configure a route with
    - the `path` to redirect from;
    - the `component` to redirect to;
    - a `pathMatch` value that tells the router how to match the URL.

app-routing.module.ts

```
const routes: Routes = [
        (…)
        { path: '', redirectTo: '/main', pathMatch: 'full' }
];
```

This route redirects a URL that fully matches the empty path to the route whose path is 'main'.

isep Instituto Superior de Engenharia do Porto

Routes definitions

- **Setting up wildcard routes**
  - A well-functioning application should gracefully handle when users attempt to navigate to a part of the application that does not exist.
  - To add this functionality to the application, set up a **wildcard route**.
  - The Angular router selects this route any time the requested URL doesn't match any router paths.

app-routing.module.ts

```
const routes: Routes = [
        (…)
        { path: '**', component: PageNotFoundComponent }
];
```

The two asterisks, **, indicate to Angular that this routes definition is a wildcard route

This component is defined to display a 404 Page or redirect to application's main component

isep Instituto Superior de Engenharia do Porto

Routes order

- **The order of routes is important because the Router uses a first-match wins strategy when matching routes.**
  - So **more specific routes should be placed above less specific routes**.
- **The routes order should be:**
  - All routes with a static path;
  - The empty path route, which matches the default route;
  - The wildcard route
    - This should be the last route in the list because it matches every URL and the Router selects it only if no other routes match first.

Routes order

app-routing.module.ts

```
(…)

const routes: Routes = [
        { path: 'crisis-list', component: CrisisListComponent },
        { path: 'heroes-list', component: HeroesListComponent },
        { path: '', redirectTo: '/heroes-list', pathMatch: 'full' }, // Redirect to
`HeroesListComponent`
        { path: '**', component: PageNotFoundComponent },  // Wildcard route for a 404 page
];

(…)
```

The Angular Router-Outlet

- **Router-Outlet is an Angular directive used to insert the component matched by routes to be displayed on the screen.**
  - Acts as a placeholder that Angular dynamically fills based on the current router state.
- **It's exported by the *RouterModule* and added to a component template.**

app.compoenet.html

```
<router-outlet></router-outlet>
<router-outlet name='left'></router-outlet>
<router-outlet name='right'></router-outlet>
```

Each outlet can have a unique name, determined by the optional name attribute.
The name cannot be set or changed dynamically.
If not set, default value is "primary".

- Named outlets can be the targets of secondary routes.
- The Route object has an outlet property to identify the target outlet:

```
{path: <base-path>, component: <component>, outlet: <target_outlet_name>}
```

isep Instituto Superior de Engenharia do Porto

The Angular Router-Outlet

- **A router outlet emits:**
  - an activate event when a new component is instantiated;
  - an deactivate event when a component is destroyed;
  - an attached event when the outlet is instructed to reattach the subtree;
  - a detached event when the outlet is instructed to detach the subtree.

app.compoenet.html

```
<router-outlet
  (activate)='onActivate($event)'
  (deactivate)='onDeactivate($event)'
  (attach)='onAttach($event)'
  (detach)='onDetach($event)'></router-outlet>
```

RouterLink

- **When applied to an element in a template, makes that element a link that initiates navigation to a route.**
- **Example:**

  Given the route configuration

  Parameter

  ```
  [{ path: 'user/:name', component: UserCmp }]
  ```

  the following creates a static link to the route:

  ```
  <a routerLink="/user/bob">link to user component</a>
  ```

isep Instituto Superior de Engenharia do Porto

Nesting routes

- **As the application becomes more complex, it becomes necessary to create routes relative to a component other than the root component.**
  - These types of nested routes are called **child routes**.

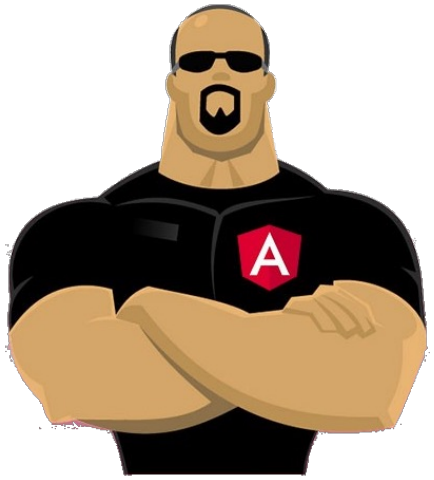
app-routing.module.ts

```
const routes: Routes = [
  (…)
  { path: 'parent-component',
    component: ParentComponent, // this component has a <router-outlet> in the template
    children: [
                {
                  path: 'child-a', // child route path
                  component: ChildAComponent, // child route component that the router renders
                },
                {
                  path: 'child-b', // another child route path
                  component: ChildBComponent, // another child route component that the router renders
                }
               ]
  }
];
```

isep Instituto Superior de Engenharia do Porto

What are Route Guards?

- **Interfaces which can tell the router whether it should allow navigation to a requested route**
- **They make this decision by looking for a `true` or `false` return value from a class which implements the given guard interface**
- **Angular provides 5 types of route guards:**
  - `CanActivate`
  - `CanActivateChild`
  - `CanLoad`
  - `CanDeactivate`
  - `Resolve`



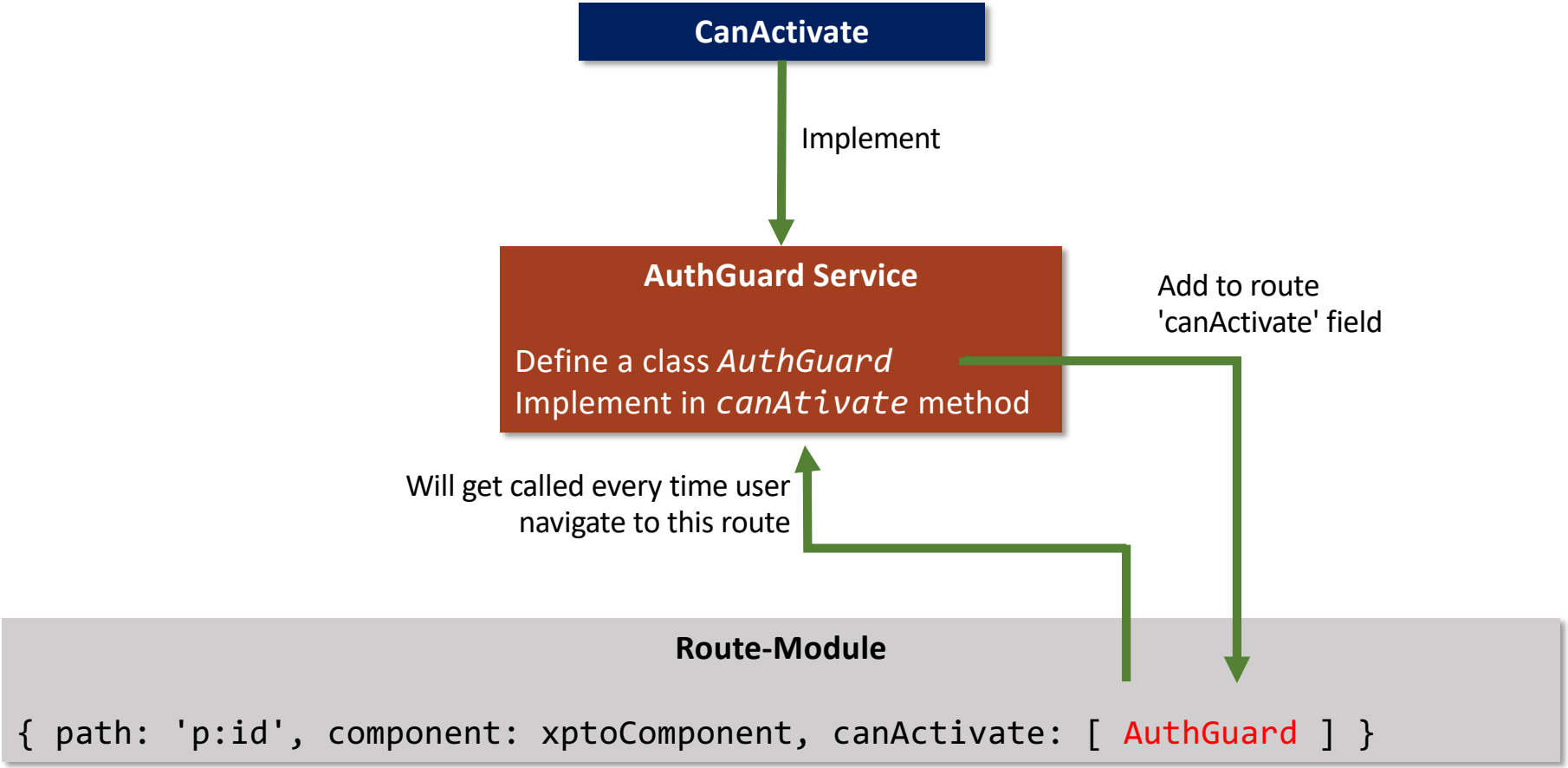
CanActivate Route Guard

- **The CanActivate interface implements route guard preventing access to the specific route**
  - If all guards return `true`, navigation continues;
  - If any guard returns `false`, navigation is cancelled;
  - If any guard returns a `UrlTree`, the current navigation is cancelled and a new navigation begins to the `UrlTree` returned from the guard.
- **Method Signature :**

```
interface CanActivate {

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
            Observable<boolean | UrlTree> |
            Promise<boolean | UrlTree> |
            boolean |
            UrlTree

}
```

isep instituto superior de engenharia do porto

`CanActivate` - How to implement it?



CanActivate - How to implement it?

- **Implement the Guard Service**

auth-guard.service.ts

```
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router } from '@angular/router';
import { Observable } from 'rxjs/Observable';

export class AuthGuardService implements CanActivate {
     constructor(private authService: AuthService, private router: Router) {}

     canActivate(route: ActivatedRouteSnapshot, state:RouterStateSnapshot): Observable<boolean
                                                                    //| Promise<boolean> | boolean
     {
           // return true if you want to navigate, otherwise return false
     }
}
```

- **Use Guard Service in route definition**

app-routing.module.ts

```
const routes: Routes = [
        { path: 'heroes-list', component: HeroesListComponent, canActivate: [AuthGuardService] }
];
```

isep Instituto Superior de Engenharia do Porto

Example – use of **canActivate**

auth-guard.service.ts

```
import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable()
export class AuthGuardService implements CanActivate {

  constructor(public auth: AuthService, public router: Router) {}

  canActivate(): boolean {
    if (!this.auth.isAuthenticated()) {
      this.router.navigate(['login']);
      return false;
    }
    return true;
  }

}
```

Example – use of **canActivate**

app-routing.module.ts

```
import { Routes, CanActivate } from '@angular/router';
import { ProfileComponent } from './Components/profile/profile.component';
import { AuthGuardService } from './Services/auth-guard.service';

export const ROUTES: Routes = [
  { path: '', component: HomeComponent },
  {
    path: 'profile',
    component: ProfileComponent,
    canActivate: [AuthGuardService]
  },
  { path: '**', redirectTo: '' }
];
```

`CanActivateChild` Route Guard

- **The CanActivateChild interface is used to prevent access to child routes of a given route.**
  - Similar to `CanAtivate` interface.
  - It is only needed to add **`CanActivateChild`** to the parent route.
    - it is not necessary to add CanAtivate to each child route.
- **Method Signature :**

```
interface CanActivateChild {

  canActivateChild(childRoute: ActivatedRouteSnapshot, state: RouterStateSnapshot):
                  Observable<boolean | UrlTree> |
                  Promise<boolean | UrlTree> |
                  boolean |
                  UrlTree

}
```

isep Instituto Superior de Engenharia do Porto

Example – use of **canActivateChild**

auth-guard.service.ts

```
import { Injectable } from '@angular/core';
import { Router, CanActivateChild } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable()
export class AuthGuardService implements CanActivateChild {

  constructor(public auth: AuthService, public router: Router) {}

  canActivateChild(): boolean {
    if (!this.auth.isAuthenticated()) {
      this.router.navigate(['login']);
      return false;
    }
    return true;
  }

}
```

Example – use of **canActivateChild**

app-routing.module.ts

```
import { Routes, CanActivateChild } from '@angular/router';
import { ProfileComponent } from './Components/profile/profile.component';
import { AuthGuardService } from './Services/auth-guard.service';

export const ROUTES: Routes = [
  { path: '', component: HomeComponent },
  {
        path: 'dashboard', component: DashboardComponent,
        canActivateChild: [AuthGuardService],
        children: [
                { path: ':id', component: InfoComponent},
                { path: ':id/edit', component: EditInfoComponent}
        ]
  },
  { path: '**', redirectTo: '' }
];
```

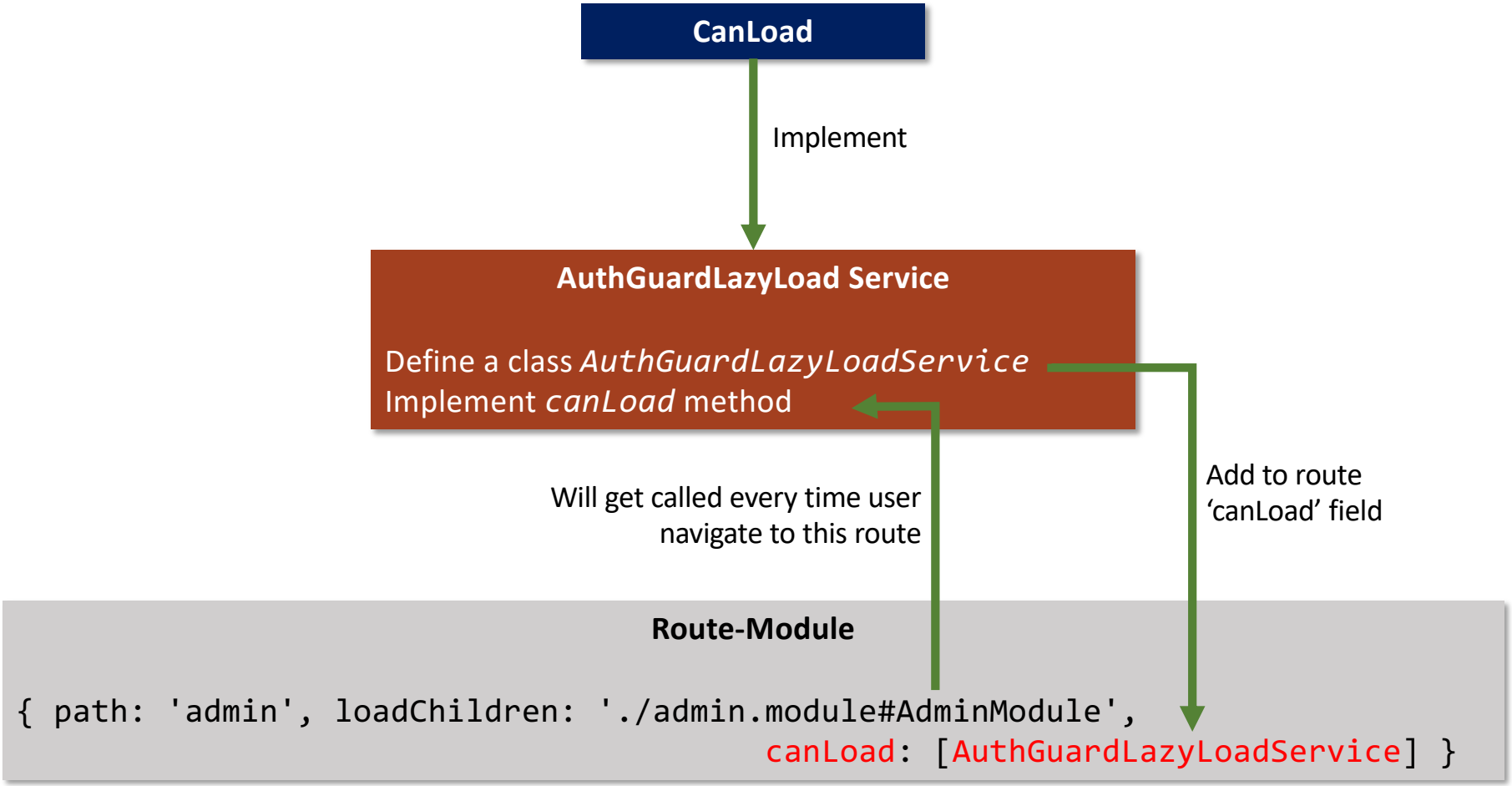
CanLoad Route Guard

- **The CanLoad interface prevents the loading of a lazy-load module.**
  - Use this guard when you don't want unauthorized user to navigate to any of the module's routes and even download the module.
  - Angular provides `canActivate` Guard, which prevents unauthorized user from accessing the route. But it does not stop the module from being downloaded.
- **Method Signature :**

```
interface CanLoad {

  canLoad(route: Route, segments: UrlSegment[]):
                  Observable<boolean | UrlTree> |
                  Promise<boolean | UrlTree> |
                  boolean |
                  UrlTree

}
```

isep instituto superior de engenharia do porto

CanLoad - How to implement it?



CanLoad - How to implement it?

- **Implement the Guard Service**

auth-guard-lazy.service.ts

```
import {CanLoad, Route, Router} from '@angular/router';

export class AuthGuardLazyService implements CanLoad {
   constructor(private router: Router) {}

   canLoad(route:Route,segments:UrlSegment[]): Observable<boolean> //| Promise<boolean> | boolean
   {
           // return true or false depending on whether you want to load that
           // module or not.
   }
}
```

- **Use Guard Service in route definition**

app-routing.module.ts

```
const routes: Routes = [
        { path: 'heroes-list', component: HeroesListComponent, canLoad: [AuthGuardLazyService] }
];
```

isep Instituto Superior de Engenharia do Porto

Example – use of **canLoad**

auth-guard.service.ts

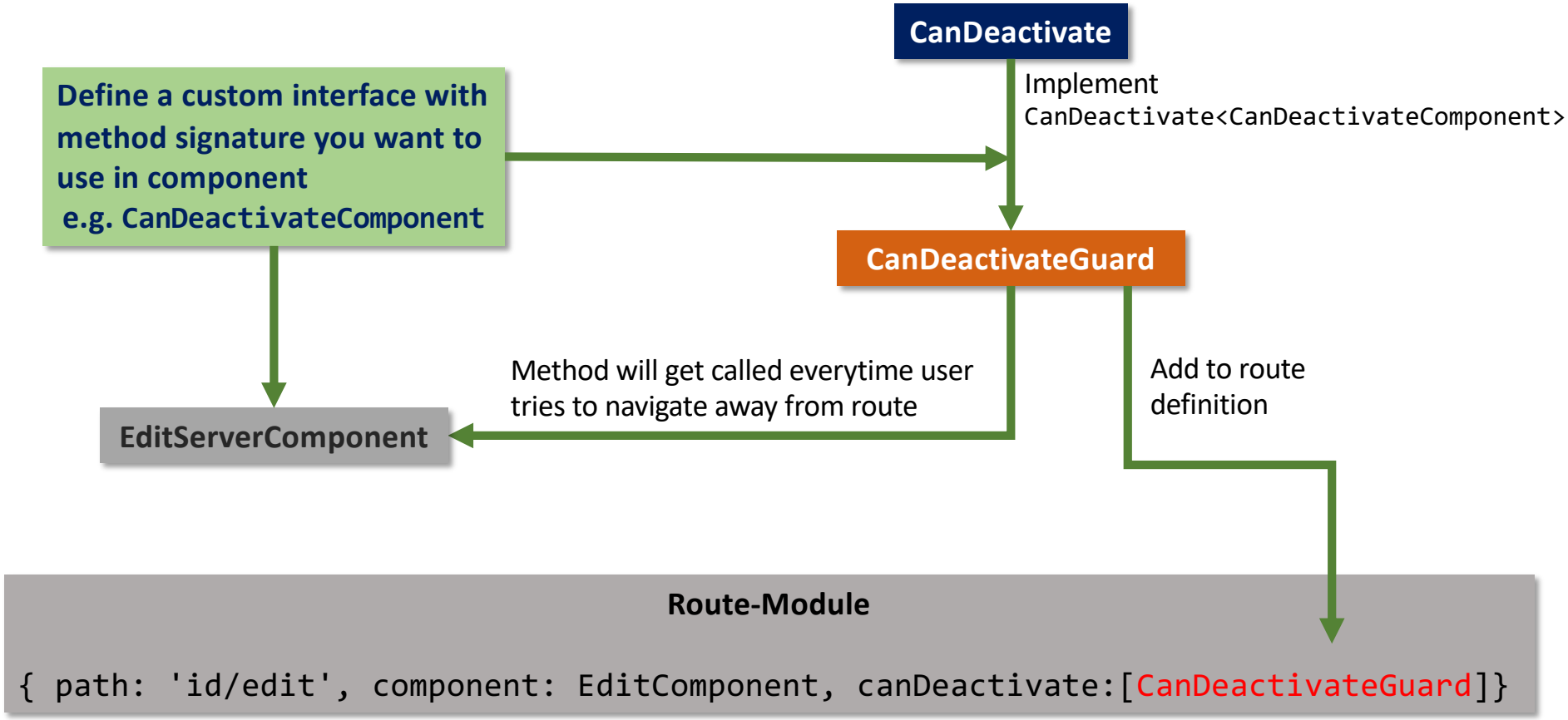
```
import { Injectable } from '@angular/core';
import { Router, Route, CanLoad } from '@angular/router';

@Injectable()
export class AuthGuardService implements CanLoad {

  constructor(public router: Router) {}

  canLoad(route: Route): boolean {
    let url: string = route.path;
    if (url=='admin') {
      alert('You are not authorized to visit this page');
      return false;
    }
    return true;
  }

}
```

CanDeactivate Route Guard

- **The CanDeactivate interface prevents users from accidentally leaving a route/page in the application.**
  - This guard can be useful when you want to prevent a user from accidentally leave the current view without saving or some other undone tasks.
  - Note that this guard doesn't prevent the user from closing the browser tab or navigating to a different address. It only prevents actions from within the application itself.
- **Method Signature :**

```
interface CanDeativate {

  canDeactivate(component: T, currentRoute: ActivatedRouteSnapshot,
                currentState: RouterStateSnapshot, nextState?: RouterStateSnapshot):
        Observable<boolean | UrlTree> |
        Promise<boolean | UrlTree> |
        boolean |
        UrlTree

}
```

CanDeactivate - How to implement it?



CanDeactivate - How to implement it?

- **Create an interface that will be used in component and use it in CanDeactivateGuard service**

can-component-deactivate.ts

```
import { CanDeactivate } from '@angular/router';

export interface CanComponentDeactivate {
    canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}
```

isep Instituto Superior de Engenharia do Porto

CanDeactivate - How to implement it?

- **Implement the Guard Service**

can-deactivate.service.ts

```
import { CanDeactivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { CanComponentDeactivate } from './can-component-deactivate';

export class CanDeactivateService implements CanDeactivate<CanComponentDeactivate> {
    canDeactivate(component:CanComponentDeactivate, currentRoute:ActivatedRouteSnapshot,
                  currentState:RouterStateSnapshot, nextState?: RouterStateSnapshot):
                                Observable<boolean> //| Promise<boolean> | boolean
    {
         return component.canDeactivate();
    }
}
```

isep Instituto Superior de Engenharia do Porto

CanDeactivate - How to implement it?

- **Implement the component class**

edit.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params, Router } from '@angular/router';
import { Observable } from 'rxjs/Observable';
import { CanComponentDeactivate } from './can-component-deactivate';

@Component({
   selector: 'app-edit',
   templateUrl: './edit.component.html',
   styleUrls: ['./edit.component.css']
})

export class EditComponent implements OnInit, CanComponentDeactivate {
   (…)
   canDeactivate(): Observable<boolean>
                  //| Promise<boolean> | boolean
   {
        // return true or false depends on whether you want user
        // to navigate away from this route or not.
   }
}
```

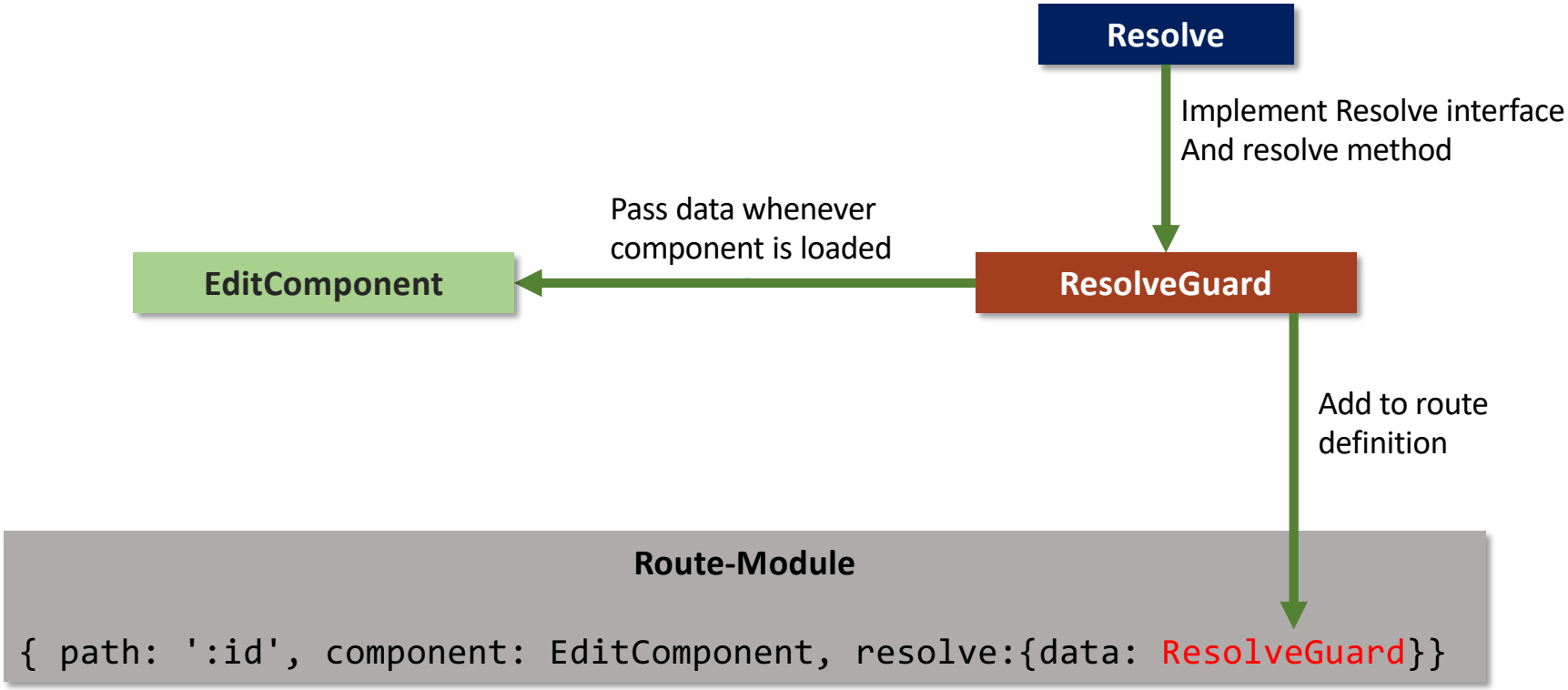
Resolve Route Guard

- **The Resolve interface can be implemented to be a data provider.**
  - Sometimes the data shared between components is so heavy that it is not possible to pass it through query parameters.
  - To handle this situation angular has provided `Resolve` Guard.
  - The interface defines a `resolve()` method that is invoked right after the `ResolveStart` router event.
  - The router waits for the data to be resolved before the route is finally activated.
- **Method Signature :**

```
interface Resolve<T> {

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
        Observable<T> |
        Promise<T> |
        T

}
```

isep Instituto Superior de Engenharia do Porto

Resolve - How to implement it?



isep Instituto Superior de Engenharia do Porto

Resolve - How to implement it?

- **Implement the Guard Service**

resolve.service.ts

```
import { Resolve, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { Observable } from 'rxjs/Observable';
import { Injectable } from '@angular/core';
import { FetchDataService } from '../fetch-data.service';

interface UserName {
    id: number;
    name: string;
}

@Injectable()

export class ResolveService implements Resolve<UserName> {
    constructor(private fetchService: FetchDataService) {}

    resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
                            Observable<UserName> //| Promise<UserName> | UserName
    {
        return this.fetchService.getUser( route.params['id'] );
    }
}
```

Resolve - How to implement it?

- **Use Guard Service in route definition**

app-routing.module.ts

```
const routes: Routes = [

        { path: ':id', component: EditComponent, resolve: {data: ResolveService} }

];
```

Resolve route takes an object as other route guards accept an array

isep Instituto Superior de Engenharia do Porto

Resolve - How to implement it?

- **Implement the component class**

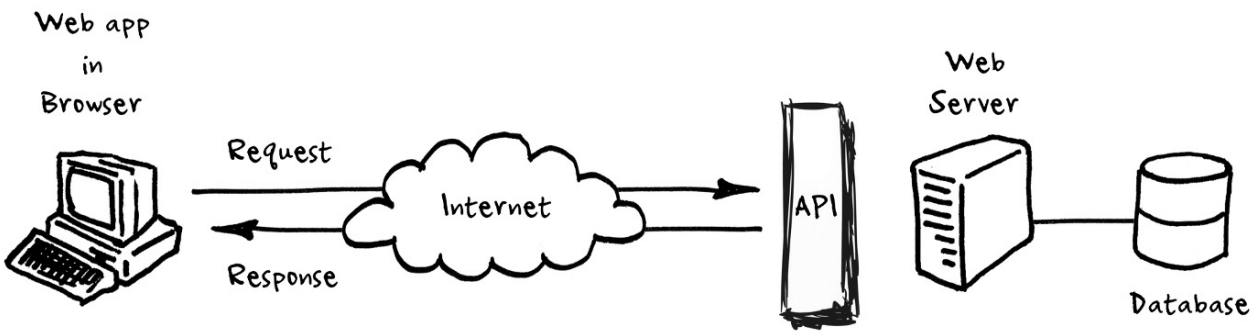
edit.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params, Router, Data } from '@angular/router';

@Component({
   selector: 'app-edit',
   templateUrl: './edit.component.html',
   styleUrls: ['./edit.component.css']
})

export class EditComponent implements OnInit {

   userName: {id: number, name: string};

   constructor(private route: ActivatedRoute, private router: Router) {}

   ngOnInit() {
       this.route.data.subscribe( (data: Data) => { this.userName = data['data']; } );
   }
}
```

Communicating with backend services using HTTP

HttpClient

- **Angular provides the `HttpClient` service class in *`@angular/common/http`*.**
  - A client HTTP API for Angular applications
- **This HTTP client service offers the following major features:**
  - The ability to request typed response objects;
  - Streamlined error handling;
  - Testability features;
  - Request and response interception.



Setup for server communication

- **Import `HttpClientModule` in the module typescript file.**
- **Inject the `HttpClient` service as a dependency of an application class.**
- **The `HttpClient` service makes use of observables for all transactions. So, import the `RxJS` `observable` and the `catchError` operator.**

xyz.module.ts

```
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
        HttpClientModule,
  ],
})
export class XyzModule {}
```

xyz.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http’;
import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';

@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) { }
}
```

Requesting data from a server

- **Use the `HttpClient.get()` method to fetch data from a server.**
  - The asynchronous method sends an HTTP request and returns an *Observable* that emits the requested data when the response is received.
  - The return type varies based on the *observe* and *responseType* values passed to the call.
- **The `get()` method takes two arguments:**
  - the endpoint URL from which to fetch;
  - an options object that is used to configure the request.

```
options: {
  headers?: HttpHeaders | {[header: string]: string | string[]},
  observe?: 'body' | 'events' | 'response',
  params?: HttpParams|{[param: string]: string | number | boolean |
ReadonlyArray<string | number | boolean>},
  reportProgress?: boolean,
  responseType?: 'arraybuffer'|'blob'|'json'|'text',
  withCredentials?: boolean,
}
```

The ***observe*** option specifies how much of the response to return.

The ***responseType*** option specifies the format in which to return data.

isep Instituto Superior de Engenharia do Porto

Requesting data from a server

- **Example (*get all books*):**

book.service.ts

```
const theServerURL = 'http://www.boostore.com/api/book/';

getBooks(): Observable<Book[]> {

    return this.http.get<Book[]>(
                theServerURL,
                {observe: 'body', responseType: 'json’}
    );

}
```

Receives a typed data object, which is easier and safer to consume.

options object that is used to configure the request.
These are the default values for these options. They could, therefore, have been omitted.

The ***endpoint URL*** from which to fetch.

isep Instituto Superior de Engenharia do Porto

Requesting data from a server

- **Example (*get a book by isbn*):**

book.service.ts

```
getBook(isbn: string): Observable<Book[]> {
   const theUrl = `${this.theServerURL}/${isbn}`;
   return this.http.get<Book[]>(
            theUrl,
            {observe: 'body', responseType: 'json'}
        );

 }
```

The ***endpoint URL*** to fetch data.
The isbn of the book to be fetched is passed in the request URL.
E.g.: *http://www.boostore.com/api/book/123456789*

options object that is used to configure the request.
These are the default values for these options. They could, therefore, have been omitted.

isep Instituto Superior de Engenharia do Porto

Reading the full response

- **By default, `HttpClient.get()` returns JSON data contained in the response body. However, it may be necessary to access information about the transaction.**
  - Servers return special headers or status codes to indicate certain conditions that are important to the application workflow.

book.service.ts

```
getBooks(): Observable<HttpResponse<Book[]>> {

   return this.http.get<Book[]>(
           theServerURL,
           {observe: 'response'}
   );

}
```

Returns an **Observable** of type ***HttpResponse*** rather than just the JSON data contained in the body

isep Instituto Superior de Engenharia do Porto

Reading the full response

- **By default, `HttpClient.get()` returns JSON data contained in the response body. However, it may be necessary to access information about the transaction.**
  - Servers return special headers or status codes to indicate certain conditions that are important to the application workflow.

resp is of type `HttpResponse<Book[]>

book.service.ts

```
getBooks(): Observable<HttpResponse<Book[]>> {

   return this.http.get<Book[]>(
            theServerURL,
            {observe: 'response'}
   );

}
```

xpto.component.ts

```
showBookResponse() {
  this.bookService.getBooks()
        .subscribe(resp => {
           const keys = resp.headers.keys();
           this.headers = keys.map(key =>
                `${key}: ${resp.headers.get(key)}`);
           this.book = resp.body!;
        });
}
```

Extract the headers

The exclamation mark **!** is known as the ***non-null assertion operator*** in TypeScript. This operator tells the TypeScript compiler that a value typed as optional cannot be null or undefined.

Sending data to a server

- `HttpClient` **also supports PUT, POST and DELETE methods used to modify remote data.**

Method	Details	
POST	Creates a new resource	***C***
GET	Retrieves a resource	***R***
PUT	Updates an existing resource	***U***
DELETE	Removes a Resource	***D***

isep Instituto Superior de Engenharia do Porto

Sending data to a server – create data

- **Use the `HttpClient.post()` method to send data to a server.**
  - The asynchronous method sends an HTTP post request and returns an *Observable*.
- **The `post()` method takes three arguments:**
  - the endpoint URL to send to;
  - the data to post in the body of the request;
  - an options object that is used to configure the request.
- ***Options* object example:**

```
const httpOptions = {
  headers: new HttpHeaders({
    'Content-Type':  'application/json',
    'Authorization': 'my-auth-token'
  })
};
```

Sending data to a server – create data

- **Example:**

book.service.ts

```
addBook(book: Book): Observable<Book> {
   return this.http.post<Book>(
            this.theServerURL,
            book,
            this.httpOptions
   );
}
```

Data sent in the request body.

The ***endpoint URL*** to send data.

options object that is used to configure the request.

isep Instituto Superior de Engenharia do Porto

Sending data to a server – delete data

- **Use `HttpClient.delete()` method.**
- **Example:**

book.service.ts

```
deleteBook(isbn: string): Observable<unknown> {
  const theUrl = `${this.theServerURL}/${isbn}`;
  return this.http.delete(
            theUrl,
            this.httpOptions
  );
}
```

The ***endpoint URL*** to delete data.
The isbn of the book to be deleted is passed in the request URL.
E.g.: *http://www.boostore.com/api/book/123456789*

options object that is used to configure the request.

isep Instituto Superior de Engenharia do Porto

Sending data to a server – delete data

- **The component isn't expecting a result from the delete operation, so it subscribes without a callback.**
  - Calling the `subscribe()` method executes the observable, which is what initiates the DELETE request.

xpto.component.ts

```
deleteBook(isbn: string) {
  this.bookService
      .deleteBook(isbn)
      .subscribe();
}
```

isep Instituto Superior de Engenharia do Porto

Sending data to a server – update data

- **Use the `HttpClient.put()` method to update data on a server.**
- **Example:**

book.service.ts

```
updateBook(book: Book): Observable<Book> {
  const theUrl = `${this.theServerURL}/${book.isbn}`;
  return this.http.put<Book>(
            theUrl,
            book,
            this.httpOptions
  );
}
```

Data sent in the request body.

The ***endpoint URL*** to update the data.
The isbn of the book to be updated is passed in the request URL.
E.g.: *http://www.boostore.com/api/book/123456789*

options object that is used to configure the request.

isep Instituto Superior de Engenharia do Porto

Sending data to a server – update data

- **As for any of the HTTP methods that return an observable, the caller, must `subscribe()` to the observable returned from the `HttpClient.put()` in order to initiate the request.**

xpto.component.ts

```
updateBook(){
     if (this.book) {
             this.bookSrv
                 .updateBook(this.book)
                 .subscribe(() => this.book=undefined);
     }
}
```

isep Instituto Superior de Engenharia do Porto

Updating Headers

- **Existing headers within the options object cannot be directly modified because instances of the `HttpHeaders` class are immutable.**
- **Use the `set()` method to return a clone of the current instance with the changes applied.**

```
httpOptions.headers =
       httpOptions.headers
                  .set('content-type', 'application/json')
                  .set('Access-Control-Allow-Origin', '*')
                  .set('Authorization', 'my-new-auth-token');
```

isep Instituto Superior de Engenharia do Porto

Configuring HTTP URL parameters

- **Use the `HttpParams` class with the `params` request option to add URL query strings in the `HttpRequest`.**

The ***HttpParams*** object is immutable. Use ***set()*** method.

Import ***HttpParams*** class.

if there is a search author, construct an option object with an HTML URL-encoded search parameter.

book.service.ts

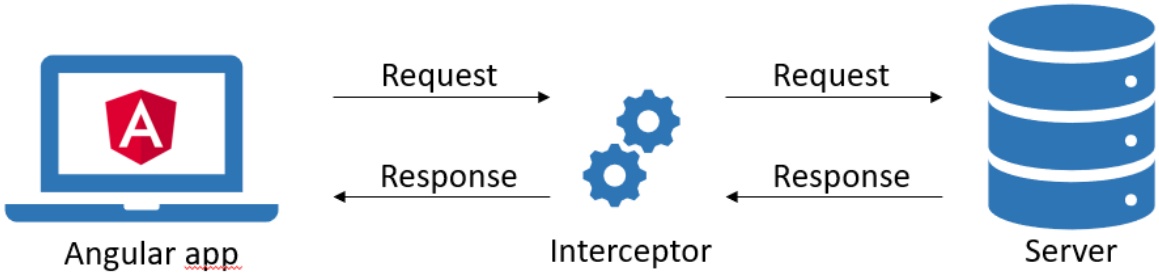
```
import {HttpParams} from "@angular/common/http";

searchBooks(author: string): Observable<Book> {
    const newOptions = author ?
         { params: new HttpParams().set('authorname', author) } : {};

    return this.http.get<Book>(
            theServerURL,
            newOptions
    );
}
```

- **Another solution would be creating an HTTP parameters directly from a query string by using the `fromString` variable:**

```
const params = new HttpParams({fromString: 'authorname=Jonh'});
```

Intercepting requests and responses

- **Interception allows to inspect and transform HTTP requests to a server.**
  - This task is performed by ***interceptors***;
  - *Interceptors* can also inspect and transform a server's response on its way back to the application;
  - Multiple interceptors form a ***forward-and-backward chain*** of request/response handlers;
  - Interceptors can perform a variety of implicit tasks, from authentication to logging, in a routine, standard way, for every HTTP request/response;
  - Without interception, developers would have to implement these tasks explicitly for each `HttpClient` method call



Write an interceptor

- **Implement the `intercept()` method of the `HttpInterceptor` interface.**
  - Most interceptors transform the outgoing request before passing it to the next interceptor in the chain, by calling `next.handle(transformedReq)`.
  - An interceptor may transform the response event stream as well, by applying additional RxJS operators on the stream returned by `next.handle()`.
  - More rarely, an interceptor may handle the request entirely, and compose a new event stream instead of invoking `next.handle()`. This is acceptable behavior, but additional interceptors will be ignored completely.
  - It is also rare but valid for an interceptor to return multiple responses on the event stream for a single request.
  - To use the same instance of `HttpInterceptors` for the entire app, import the `HttpClientModule` only in `AppModule`, and add the interceptors to the root application injector.
    - If you import `HttpClientModule` multiple times across different modules (e.g: in lazy loading modules), each import creates a new copy of the `HttpClientModule`, which overwrites the interceptors provided in the root module.

Write an interceptor

- **Example:**

do-nothing.interceptor.ts

```
import { Injectable } from '@angular/core';
import { HttpEvent, HttpInterceptor, HttpHandler, HttpRequest} from '@angular/common/http';
import { Observable } from 'rxjs';

/* Pass untouched request through to the next request handler. */
@Injectable()
export class DoNothing implements HttpInterceptor {

    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
        return next.handle(req);
    }

}
```

Most interceptors inspect the request on the way in and forward the potentially altered request to the handle() method of the next object which implements the *HttpHandler* interface.
Like *intercept()*, the *handle()* method transforms an HTTP request into an Observable of *HttpEvents* which ultimately include the server's response.
The *intercept()* method could inspect that observable and alter it before returning it to the caller.

The intercept method transforms a request into an Observable that eventually returns the HTTP response.
In this sense, each interceptor is fully capable of handling the request entirely by itself.

isep Instituto Superior de Engenharia do Porto

The next object

- **The `next` object represents the next interceptor in the chain of interceptors.**
  - The final next in the chain is the `HttpClient` backend handler that sends the request to the server and receives the server's response;
  - An interceptor could skip calling `next.handle()`, short-circuit the chain, and return its own Observable with an artificial server response.

isep Instituto Superior de Engenharia do Porto

Provide the interceptor

- **Interceptors are services managed by Angular's DI system. So, the interceptor class must be provided before app ca use it.**
- **Since interceptors are optional dependencies of the `HttpClient` service, they must be provided in the same injector or in a parent of the injector that provides `HttpClient`.**
  - Interceptors provided after DI creates the `HttpClient` are ignored.
- **Consider creating a file that gathers all the interceptor providers into an array**

```
import { HTTP_INTERCEPTORS } from '@angular/common/http';
import { DoNothing } from './do-nothing.intreceptor';

/* Http interceptor providers in outside-in order */
export const httpInterceptorProviders = [
  { provide: HTTP_INTERCEPTORS, useClass: DoNothing, multi: true }
];
```

Import and add it to the *AppModule* provider's array.

```
providers: [
  httpInterceptorProviders
],
```

Tells Angular that HTTP_INTERCEPTORS is a token for a multiprovider that injects an array of values, rather than a single value.

isep Instituto Superior de Engenharia do Porto

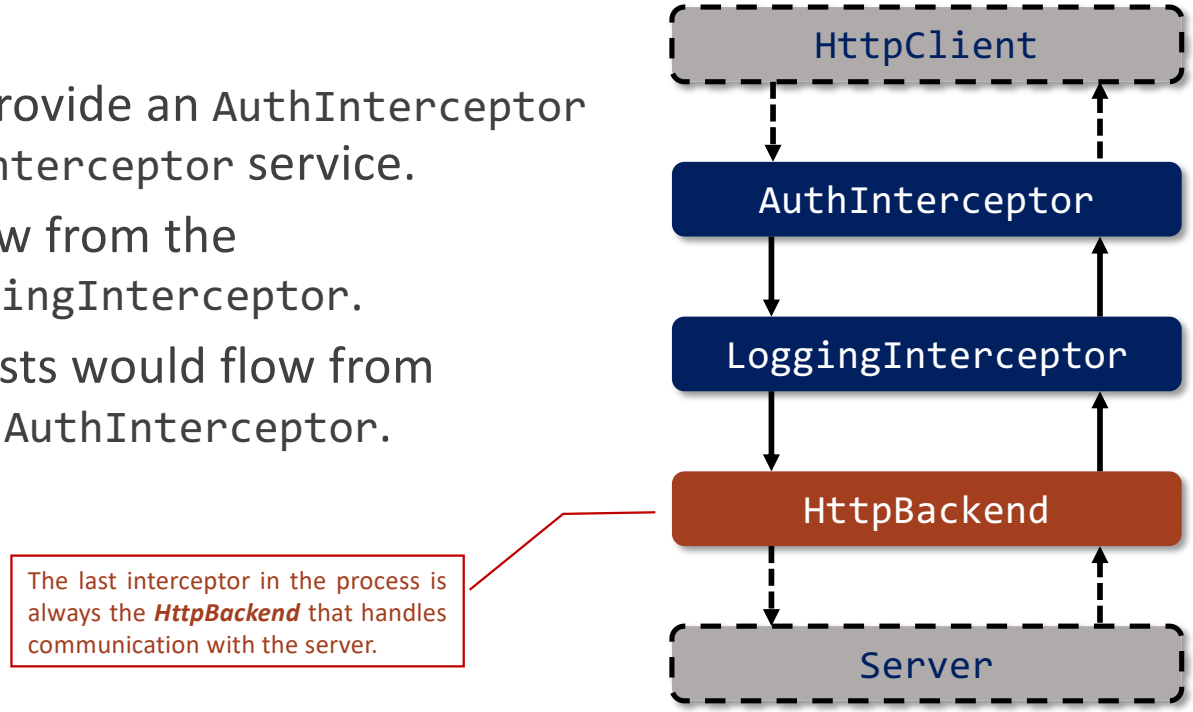
Provide the interceptor

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HttpClientModule } from '@angular/common/http’;

import { httpInterceptorProviders } from './Interceptors/httpInterceptorProviders';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule
  ],
  providers: [
    httpInterceptorProviders
  ],
  bootstrap: [AppComponent]
})
```

Interceptor order

- **Angular applies interceptors in the order they are provided.**
- **For example, consider a situation where we want to handle the authentication of the HTTP requests and log them before sending them to a server.**
  - To accomplish this, we can provide an `AuthInterceptor` service and then a `LoggingInterceptor` service.
  - Outgoing requests would flow from the `AuthInterceptor` to the `LoggingInterceptor`.
  - Responses from these requests would flow from `LoggingInterceptor` back to `AuthInterceptor`.



Handling interceptor events

- **Most `HttpClient` methods return observables of `HttpResponse<any>`.**
  - The `HttpResponse` class is an event, whose type is `HttpEventType.Response`.
- **A single HTTP request can generate multiple events of other types, including upload and download progress events.**
  - The methods `HttpInterceptor.intercept()` and `HttpHandler.handle()` return observables of `HttpEvent<any>`.
- **Many interceptors are only concerned with the outgoing request and return the event stream from `next.handle()` without modifying it.**
- **Some interceptors, however, need to examine and modify the response from `next.handle()`.**
  - these operations **can see all these events** in the stream.

Handling interceptor events

- **Although interceptors are capable of modifying requests and responses, the `HttpRequest` and `HttpResponse` instance properties are readonly, making them immutable.**
  - They are immutable because an app might retry a request several times before it succeeds, which means that the interceptor chain can re-process the same request multiple times.
  - If an interceptor could modify the original request object, the re-tried operation would start from the modified request rather than the original.
  - Immutability ensures that interceptors **see the same request for each try**.
  - To alter a request, clone it first and modify the clone before passing it to `next.handle()`

```
// clone request and replace 'http://' with 'https://' at the same time
const secureReq = req.clone( {url: req.url.replace('http://', 'https://')} );

// send the cloned, "secure" request to the next handler.
return next.handle(secureReq);
```

isep Instituto Superior de Engenharia do Porto

Angular testing

isep

Instituto Superior de
Engenharia do Porto

Disclaimer

- **Some of this material/slides are adapted from several:**
  - Presentations found on the internet;
  - Books;
  - Web sites (in particular the angular.io website);
  - …

Index

- **Angular testing**
  - End-to-end tests

End-to-end testing

isep
Instituto Superior de
Engenharia do Porto

How end-to-end tests work

- **An end-to-end test mimics how a user interacts with the application.**
  - Typically, the test engine launches an ordinary browser and controls it remotely.
  - Once the browser is started, the end-to-end test navigates to the application's URL, reads the page content and makes keyboard and pointer input.
    - For example, the test fills out a form and clicks on the submit button.
  - The end-to-end test makes expectations:
    - Does the page include the right content?
    - Did the URL change?
  - This way, whole features and user interfaces are examined.

End-to-end testing frameworks

- **Frameworks for end-to-end tests allow navigating to URLs, simulating user input and inspecting the page content. Apart from that, they have little in common. The test syntax and the way the tests are run differ widely.**
- **There are two categories of end-to-end testing frameworks:**
  - Those that use WebDriver protocol;
  - Those that do not.
- **Two major e2e testing frameworks:**
  - **Protractor** – based on WebDriver
  - **Cypress** – does not use WebDriver

WebDriver protocol

- **The WebDriver protocol defines a way to control a browser remotely with a set of commands.**
  - It originates from the Selenium browser automation project and is now developed at the World Wide Web Consortium (W3C).
  - All common browsers support the WebDriver protocol.
- **The most important WebDriver commands are:**
  - Navigate to a given URL
  - Find one or more elements in the DOM
  - Get information about a found element:
    - Get an element attribute or property
    - Get the element's text content
  - Click on an element
  - Send keyboard input to a form field
  - Execute arbitrary JavaScript code

Protractor

- **Protractor is an end-to-end testing framework based on WebDriver, made for Angular applications.**
  - Like the Angular framework itself, Protractor originates from Google.
- **Up until Angular version 12, Protractor was the default end-to-end testing framework in projects created with Angular CLI.**
- **Since Angular 12, Protractor is deprecated.**
  - The main reason for Protractor's deprecation is that it was not maintained for years.
- **In new CLI projects, there is no default end-to-end testing solution configured**

Protractor

- **Protractor has two outstanding features designed for testing Angular applications**
  1. Protractor hooks into the Angular app under tests.
     - After sending a WebDriver command, Protractor waits for Angular to update the page;
     - Then it continues with the next command;
     - By synchronizing the test and the application, the test gets faster and more reliable.
  2. Protractor had a feature called control flow…
     - While WebDriver commands are asynchronous by nature, the control flow allows to write tests as if they ran synchronously.
     - Protractor's control flow implementation led to inconsistencies and bugs, so the feature was deprecated…
     - Therefore, you must use `async/await` to explicitly wait for a command to complete.

Cypress

- **Cypress is an end-to-end testing framework that is not based on WebDriver.**
  - There are no Angular-specific features. Any web site can be tested with Cypress.
  - WebDriver-based testing solutions are flexible and powerful but turned out to be slow and unreliable. Cypress aims to improve both the developing experience and the reliability of end-to-end tests.
- **Cypress employs a fundamentally different architecture.**
  - A Node.js application starts the browser.
  - The browser is not controlled remotely, but the tests run directly in the browser, supported by a browser plugin.
  - The test runner provides a powerful user interface for inspecting and debugging tests right in the browser.

Installing Cypress

- **Using the Angular CLI execute the following command in the project directory**
  - Answer *yes* (Y) to all installer questions.

```
> ng add @cypress/schematic
```

- **This command does four important things:**
  - Add Cypress and auxiliary `npm` packages to `package.json`.
  - Add the Cypress configuration file `cypress.config.ts`.
  - Change the `angular.json` configuration file in order to add `ng run` commands.
  - Create a sub-directory named `cypress` with a scaffold for the tests.

Writing an end-to-end test with Cypress

- **The cypress folder contains:**
  - a `tsconfig.json` configuration for all TypeScript files specifically in this directory;
  - an `e2e` directory for the end-to-end tests;
  - a `support` directory for custom commands and other testing helpers;
  - a `fixtures` directory for test data.
- **The tests are structured with the test framework Mocha. The assertions (also called expectations) are written using Chai.**
- **A test file contains one or more suites declared with**

```
describe('…', () => { … })
```

- **Typically, one file contains one describe block, possible with nested describe blocks.**

End-to-end test structure

```
describe('… Feature description …', () => {
  beforeEach(() => {

    // Navigate to the page

  });

  it('… User interaction description …', () => {

    // Interact with the page
    // Assert something about the page content

  });
});
```

spec.cy.ts

```
describe('My First Test', () => {
  beforeEach(() => {
        cy.visit('/')
  });

  it('Visits the initial project page', () => {
        cy.contains('app is running!')
  })
})
```

This is the example created by installing Cypress.

The Books' login page test example

```
describe('login page test',() =>{
    beforeEach(() => {
        cy.visit('/')
    });

    it('has the correct title', () => {
        cy.title().should('equal','Books');
    });

    it('the email and password boxes are empty', () =>{
        cy.get('[id="email"]').should('be.empty');
        cy.get('[id="password"]').should('be.empty');
    });

    it('using the right credentials', () => {
        cy.get('[id="email"]').type('prp@isep.ipp.pt');
        cy.get('[id="password"]').type('teste');
        cy.get('[type="submit"]').click();
        cy.get('button').should('have.length',3);
    });
});
```

`cy.visit` orders the browser to visit the given URL.
Cypress appends the path to the *baseUrl* which, by default, is set to `http://localhost:4200` in the Cypress configuration file, `cypress.config.ts`.

`cy.title` returns the page title (to be more specific, it returns a Cypress chainer that wraps a string).
The chainer has a should method for creating an assertion.

`cy.get` finds an element by a property (id, name class, …).
`cy.get` returns a Chainer, an asynchronous wrapper around the found elements, enriched with useful methods.

`cy.get(…).type` allows it to interact, sending keys, to an element.

`cy.get(…).click` simulates pressing the button

Running the Cypress tests

- **Cypress has two shell commands to run e2e tests:**

```
> npx cypress run
```

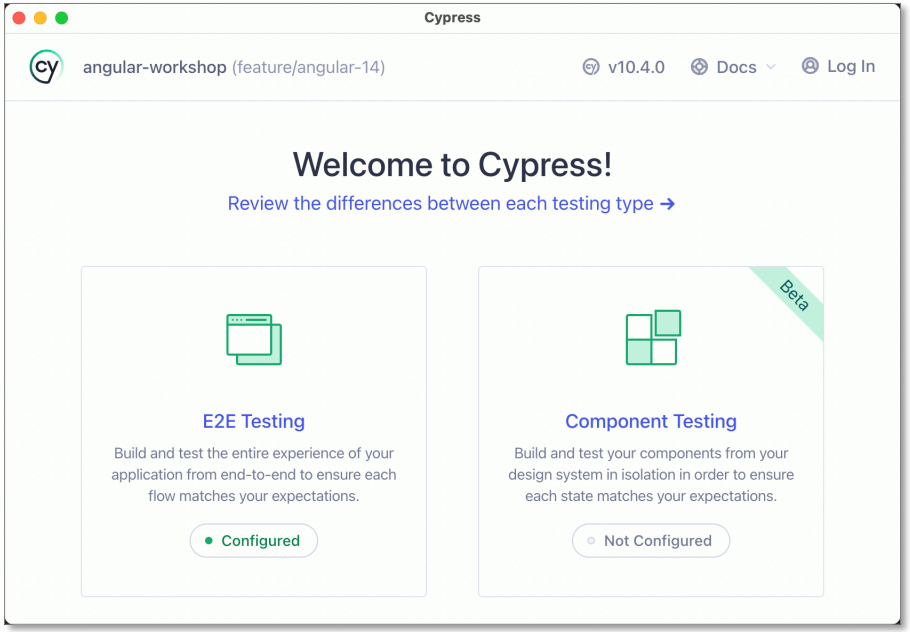
  - Non-interactive test runner.
  - Runs the tests in a “headless” browser. This means the browser window is not visible.
  - The tests are run once, then the browser is closed, and the shell command finishes. The test results are displayed in the shell output.
  - This command is typically used in a continuous integration environment.

```
> npx cypress open
```

  - Interactive test runner.
  - Opens a window to select the browser to use and which tests to run. The browser window is visible and remains visible after completion.
  - The test results are displayed in the browser window. If the test files are changed, Cypress will automatically run the tests again.
  - This command is typically used in the development environment.

Running the Cypress tests

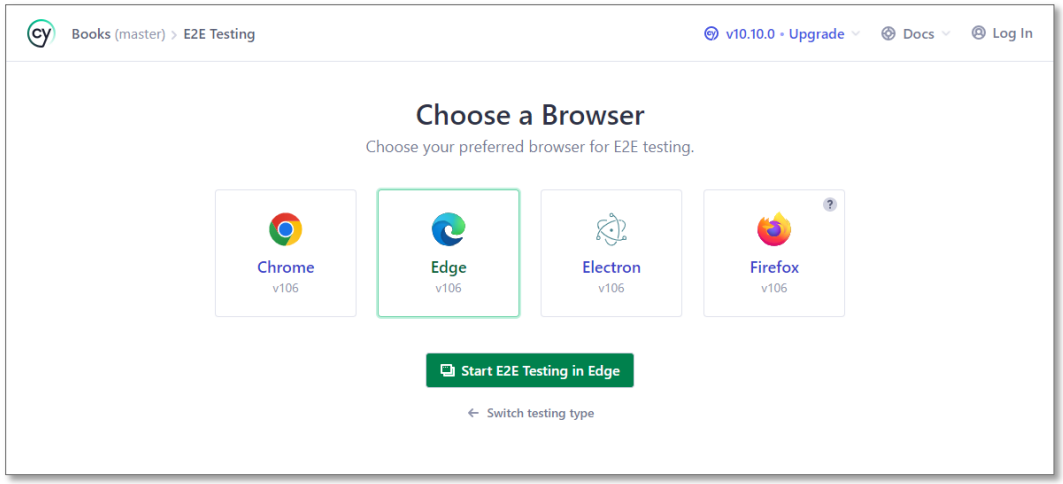
- **The cypress open command will open the test runner**
  - The first choice to make is the type of test.



  - Choose “E2E Testing”

Running the Cypress tests

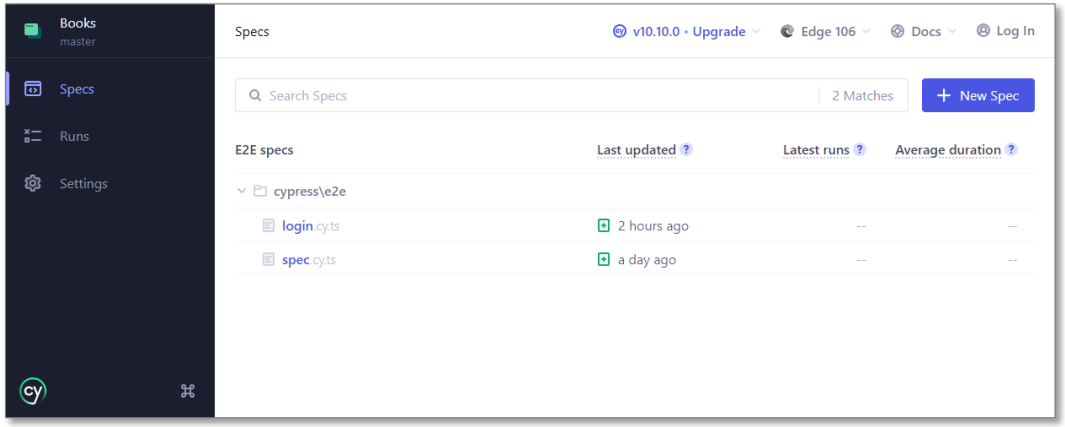
- **Choose the browser for running the tests**

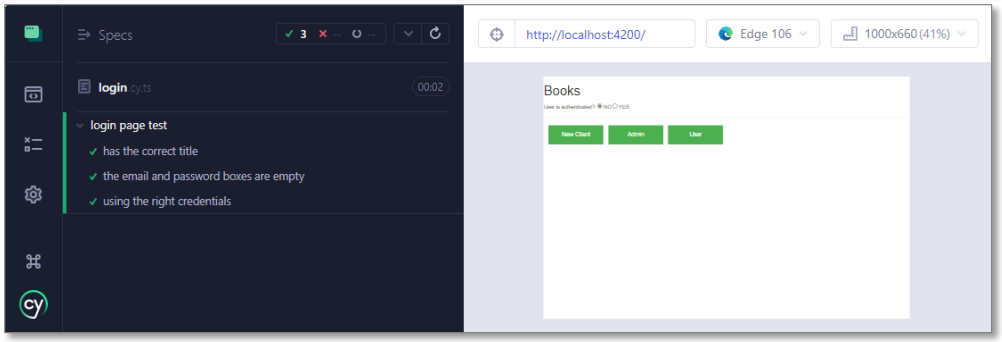


- Cypress automatically lists all browsers it finds on the system.
- In addition, the tests can be run on Electron.
  - Cypress' user interface is an Electron application.
  - Electron is based on Chromium, the open-source foundation of the Chrome browser.

Running the Cypress tests

- **After selecting a browser and clicking the “Start E2E Testing in …” button the testing interface is launched.**
- **Select a test and run it**



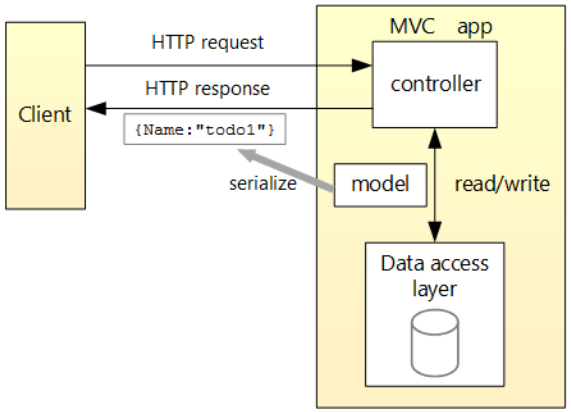


ASP.NET Web API

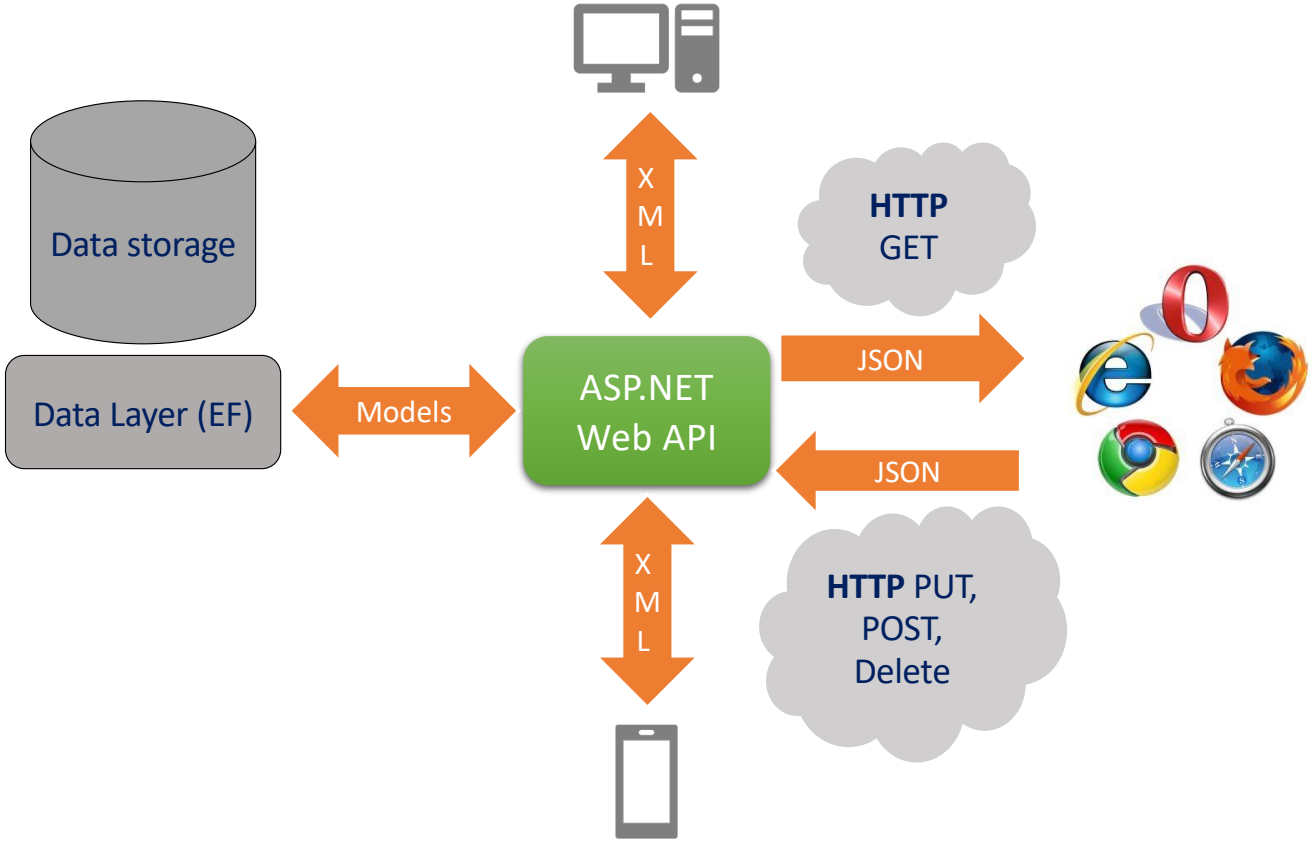
ARQSI

What is ASP.NET Web API?

- Framework that makes it easy to build HTTP services for browsers and mobile devices
- Platform for building RESTful applications on the .NET Framework core using ASP.NET stack



Scenario


Data storage
Data Layer (EF)
Models
ASP.NET
Web API
X
M
L
X
M
L
HTTP
GET
JSON
JSON
HTTP PUT,
POST,
Delete

ASP.NET Web API

- ASP.NET Web API is a Framework for building HTTP Services on top of the .Net Framework.
- Routing in MVC:
  - `{controller}/{action}/{id}`

In MVC routing we have the ***action***

- Routing in ASP.NET Web API :
  - `api/{controller}/{id}`
- In Web API the action is determined by the HTTP verb (POST, GET, PUT, DELETE, ...)

`ControllerBase` class

- A Web API consists of one or more controller classes that derive from `ControllerBase`
  - The Web API project template provides a starter controller
- Don't create a Web API controller derived from the `Controller` class
  - Controller derives from `ControllerBase` and adds support for views
  - It's for handling web pages and not web API requests

ControllerBase class

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class TodoItemsController : ControllerBase
    {
        private readonly TodoContext _context;

        public TodoItemsController(TodoContext context)
        {
            _context = context;

            if (_context.TodoItems.Count() == 0)
            {
                _context.TodoItems.Add(new TodoItem { Name = "Item1" });
                _context.SaveChanges();
            }
        }
    }
}
```

Decorates the class with the **[ApiController]** attribute. This attribute indicates that the controller responds to web API requests.

Inject the database context (**TodoContext**) into the controller. The database context is used in each of the CRUD methods in the controller.

Adds an item to the database if the database is empty. This code is in the constructor, so it runs every time there's a new HTTP request. If you delete all items, the constructor creates Item1 again the next time an API method is called.

Routing

- Web API selects actions based on HTTP methods
- By default, the Web API looks for a match between an HTTP method and the beginning of the controller action method.
  - method `PutCustomers` → HTTP request `HTTP PUT`
- This convention may be overpass by using the annotations:

  - `[HttpDelete]`
  - `[HttpGet]`
  - `[HttpHead]`
  - `[HttpPatch]`
  - `[HttpPost]`
  - `[HttpPut]`
  - `[HttpOptions]`

Routing attributes

- Allows to differentiate routings
  - `[Route ("<expressão>")]`
- Examples:
  - `[Route("{id:int}")]`
    - *api/book/5*
  - `[Route("title/{nome}")]`
    - `api/book/title/Game Of Thrones`
  - `[Route("{pubdate:datetime:regex(^\\d{4}-\\d{2}-\\d{2}$)}")]`
    - `api/book/2018-06-24`

Routing annotations

```
[Route("api/books")]
public class BooksController : ControllerBase
{
    [HttpGet]
    public Task<ActionResult<IEnumerable<Book>>> GetBooks()
    {
       return await _context.Books.ToListAsync();
    }

    [HttpGet("{id}")]
    public Task<ActionResult<Book>> GetBook(long id)
    {
       var book = await _context.Books.FindAsync(id);

       if (book == null)
       {
          return NotFound();
       }

       return book;
    }

    [HttpPost]
    public HttpResponseMessage CreateBook(Book book) { ... }
}
```

These methods implements:

- GET /api/books
  - *http://localhost:3000/api/books*
- GET /api/books/{id}
  - *http://localhost:3000/api/books/1*
- POST /api/books
  - *http://localhost:3000/api/books*

The following HTTP response is produced by the call to *GetBooks*:

```
[
   {
      "id" : 1,
      "title" : "Web API Bible",
      "author" : "John Smith"
   }
]
```

Routing – Route Prefix

- In the same controller it is usual for routing to start in the same route prefix

```
public class BooksController : ControllerBase
{
    [Route("api/books")]
    public Task<ActionResult<IEnumerable<Book>>> GetBooks()
    { ... }

    [Route("api/books/{id}")]
    public Task<ActionResult<Book>> GetBook(long id)
    { ... }

    [Route("api/books")]
    [HttpPost]
    public HttpResponseMessage CreateBook(Book book)
    { ... }
}
```

Routing – Route Prefix

- In the same controller it is usual for routing to start in the same route prefix

```
[RoutePrefix("api/books")]
public class BooksController : ControllerBase
{
    [Route("")]
    public Task<ActionResult<IEnumerable<Book>>> GetBooks()
    { ... }

    [Route("{id}")]
    public Task<ActionResult<Book>> GetBook(long id)
    { ... }

    [Route("")]
    [HttpPost]
    public HttpResponseMessage CreateBook(Book book)
    { ... }
}
```

Routing – Route Prefix

- Use a tilde (~) on the method attribute to override the route prefix:

```
[RoutePrefix("api/books")]
public class BooksController : ControllerBase
{

 (…)

    // GET /api/authors/1/book
    [Route("~/api/authors/{authorId:int}/books")]
    public Task<ActionResult<IEnumerable<Book>>> GetBooksByAuthor(long id)
    { ... }

 (…)

}
```

Routing – QueryString

- How to execute the request:

```
GET : api/categoria/?descricao=armarios de garagem
```

- In other words
  - To execute a route that responds to the GET method in the `categoria` controller.
- Problem:
  - QueryString (in particular the symbol '?') cannot be specified within a route

Routing – QueryString

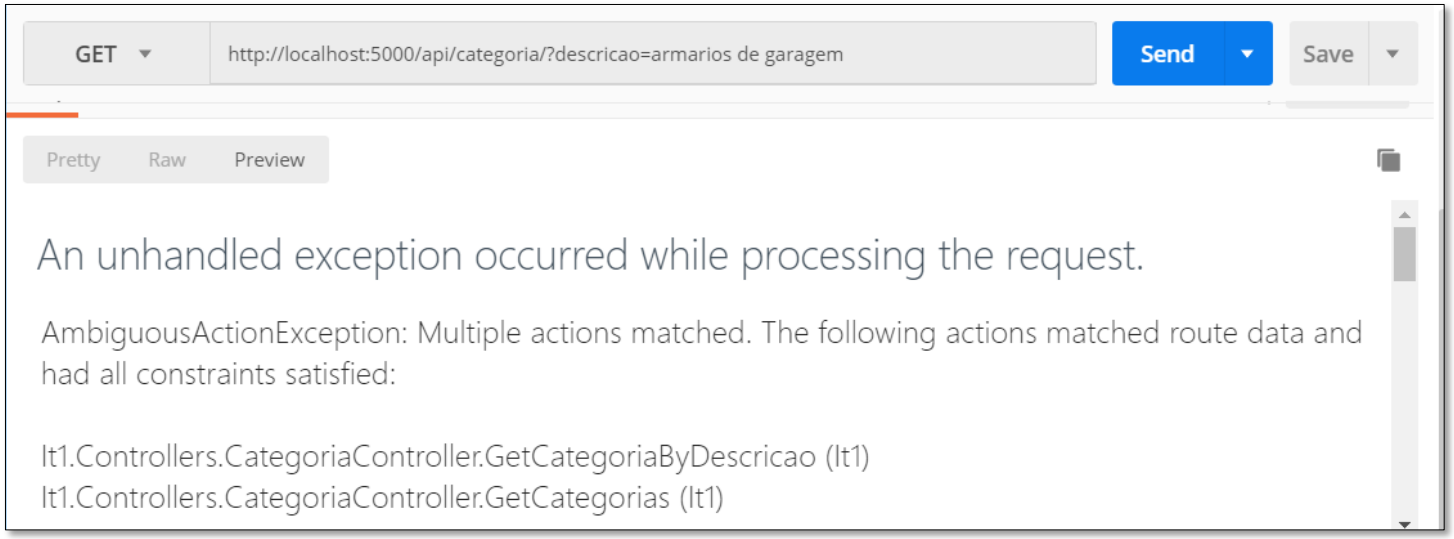
- Obvious solution

```
api/categoria/?descricao=armarios de garagem
```

```
// GET api/category/?descricao={nome}
[HttpGet]
public IQueryable<Categoria> GetCategoriaByDescricao(
                                    [FromQuery(Name = "descricao")] string desc)
{
    IQueryable<Categoria> categoria = _context.Categorias.
        Where(c => c.Descricao.Equals(desc, StringComparison.OrdinalIgnoreCase));

    return categoria;
}
```

Routing – QueryString

- However, there are **two methods** in the controller that respond to GET, which may cause ambiguity in the response.



Routing – QueryString

- The solution is to remove ambiguity:

```
api/categoria/?descricao=armarios de garagem
```

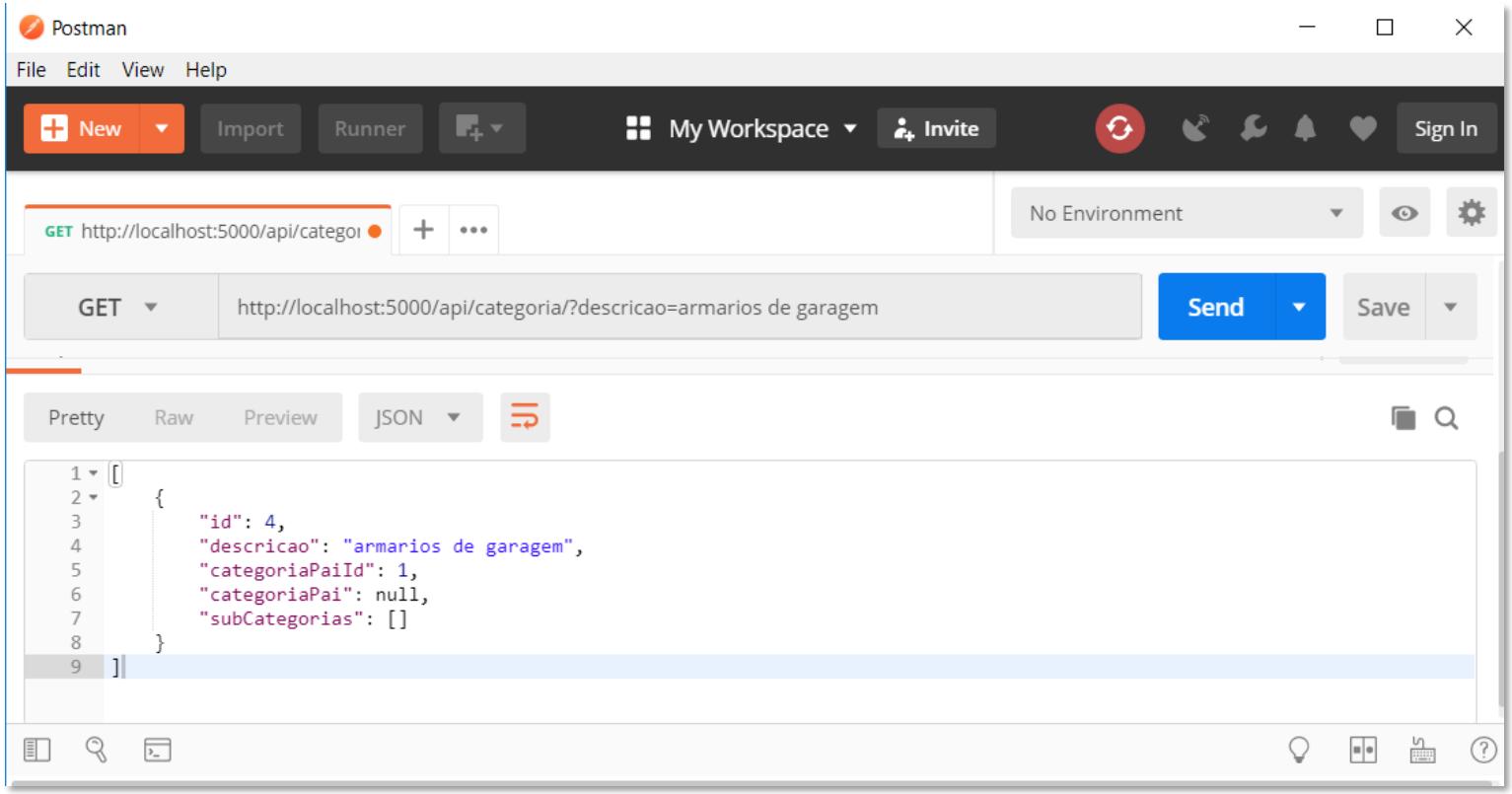
```
(…)

// GET api/Categoria
[HttpGet]
public IEnumerable<Categoria> GetCategorias()
{
     if(Request.Query.ContainsKey("descricao"))
               return GetCategoriaByDescricao(Request.Query["descricao"]);

     //conteúdo restante do GetCategorias
     (…)
}

(…)
```

Routing – QueryString

```
api/categoria/?descricao=armarios de garagem
```

```
(…)

public IQueryable<Categoria> GetCategoriaByDescricao(string desc)
{
    IQueryable<Categoria> categoria = _context.Categorias.
        Where(c => c.Descricao.Equals(desc, StringComparison.OrdinalIgnoreCase));

    return categoria;
}

(…)
```

Routing – QueryString

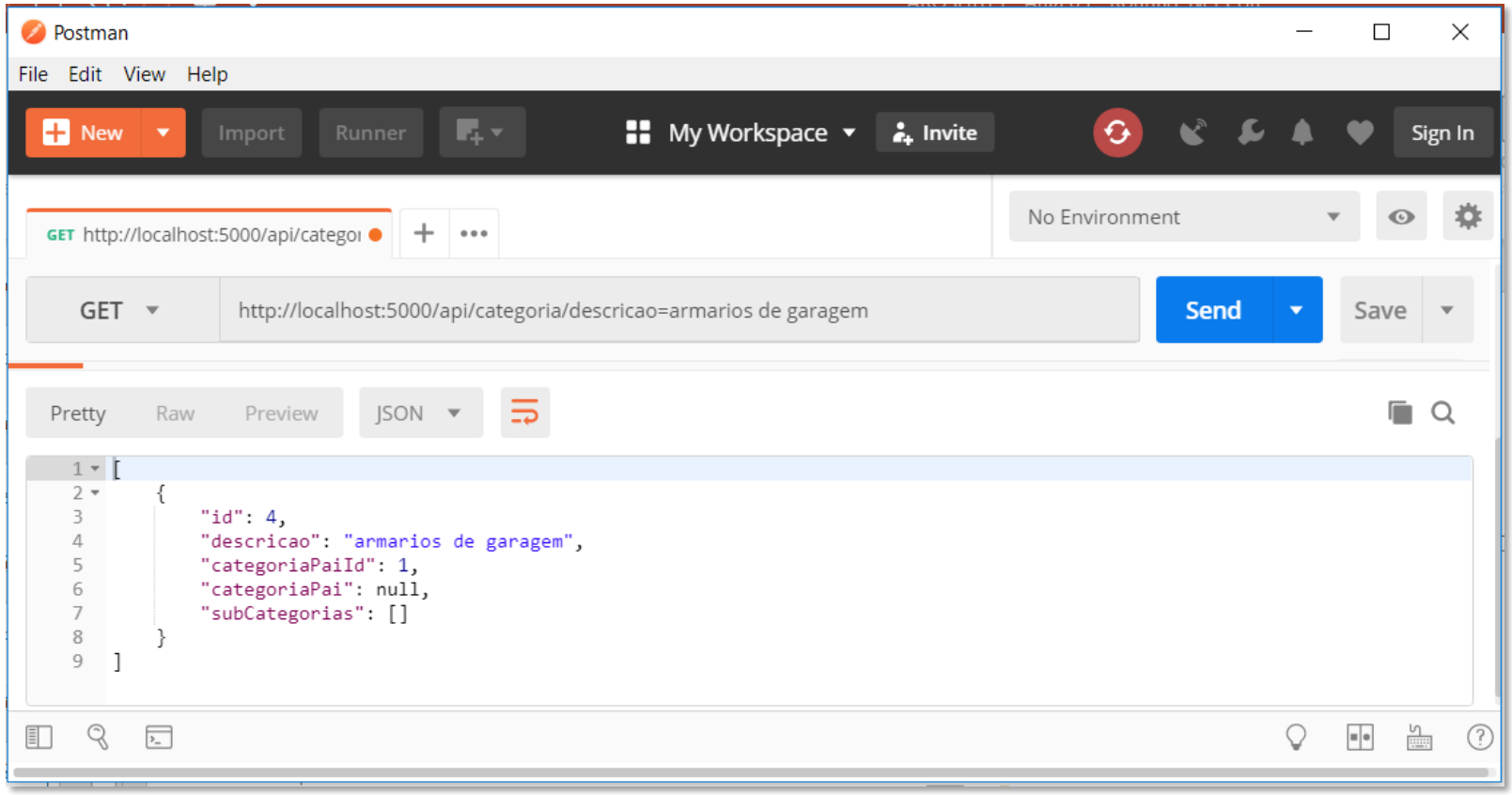


Routing – dropping the “?”

```
api/categoria/descricao=armarios de garagem
```

```
// GET api/Categoria/descricao={nome}
[HttpGet("descricao={desc}")]
public IQueryable<Categoria> GetCategoriaByDescricao([FromRoute] string desc)
{
    IQueryable<Categoria> categoria = _context.Categorias.
        Where(c => c.Descricao.Equals(desc, StringComparison.OrdinalIgnoreCase));

    return categoria;
}
```

Routing – dropping the “?”



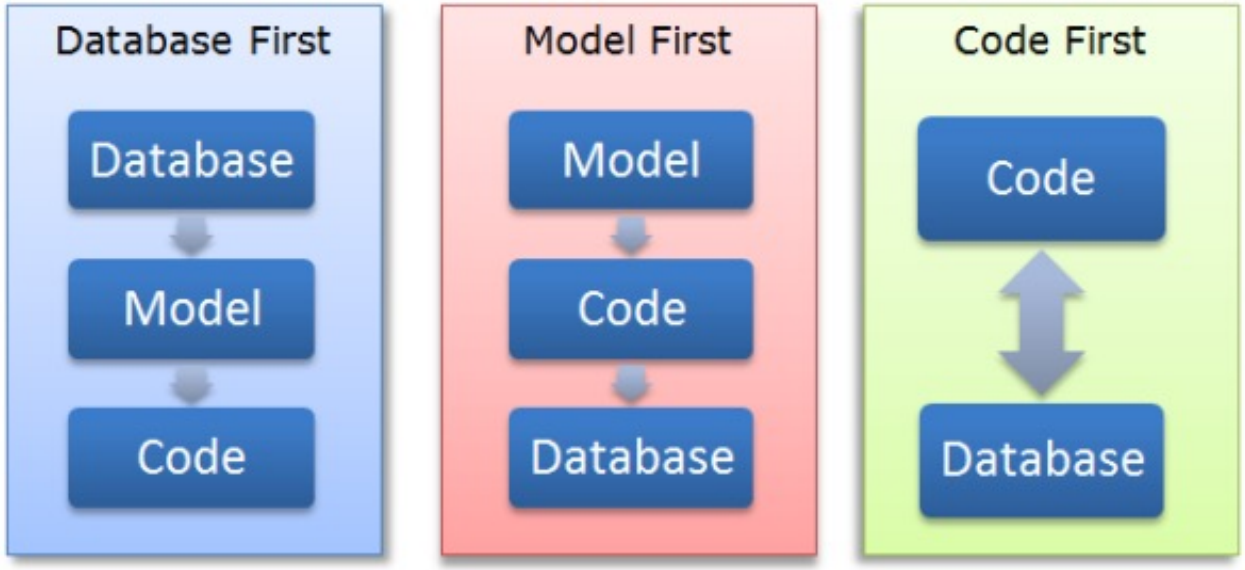
Entity Framework

ARQSI

ORM

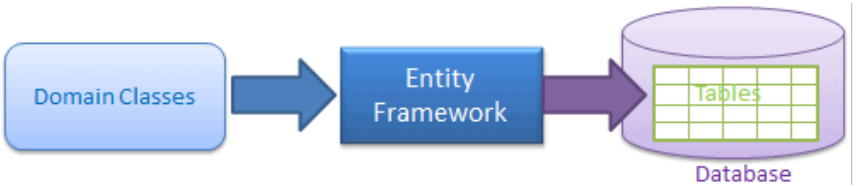
- ORM – Object Relational Mapping
  - For converting data between incompatible type systems in OOP languages.
  - In effect creates “virtual object database”, for using within the OOP language
- .NET ORM systems
  - Entity Framework – Microsoft
  - NHibernate
  - Dapper
  - … and many others

Several Approaches

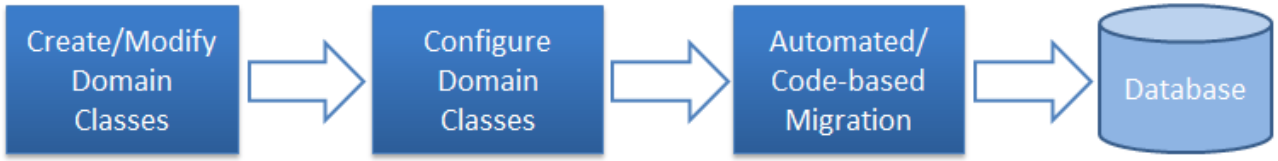


What is Code-First?

- In the Code-First approach, you focus on the application's domain and create classes for the domain entities. The EF will create the database based on domain classes and configurations.

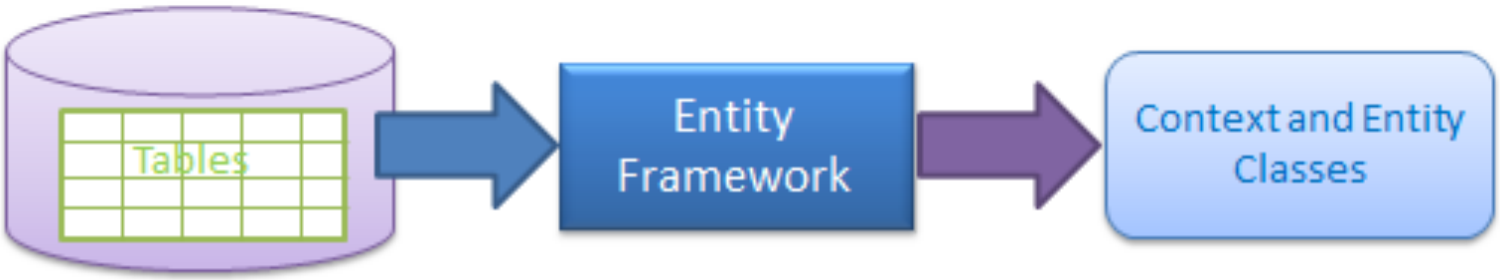


- The code-first development workflow is:



What is Database-First?

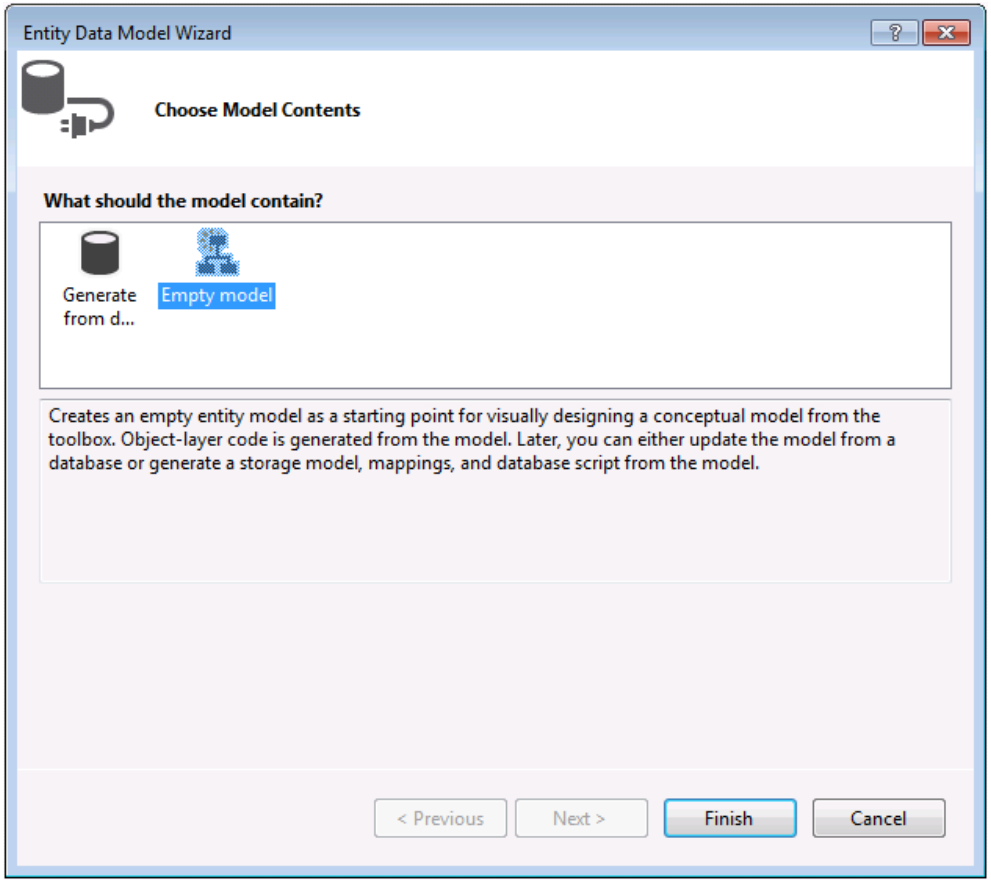
- Usual approach when the data model already exists
- Entity Framework (EF) creates the model based on the database tables and fields



Database-First Approach

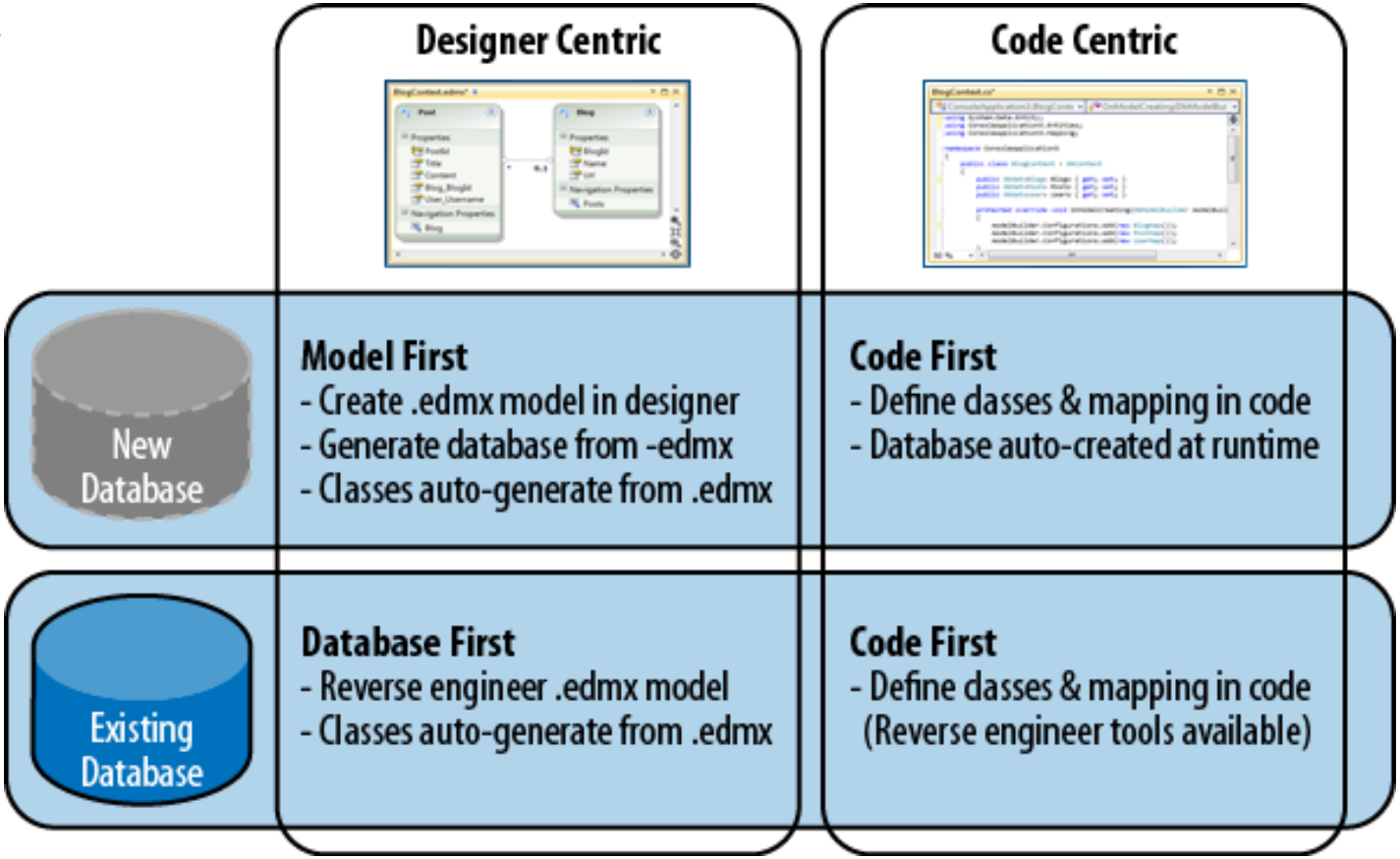
What is Model-First?

- Model design in an IDE tool creating:
  - Entities
  - Relationships
  - Inheritance structures
- The model design implies the creation BD

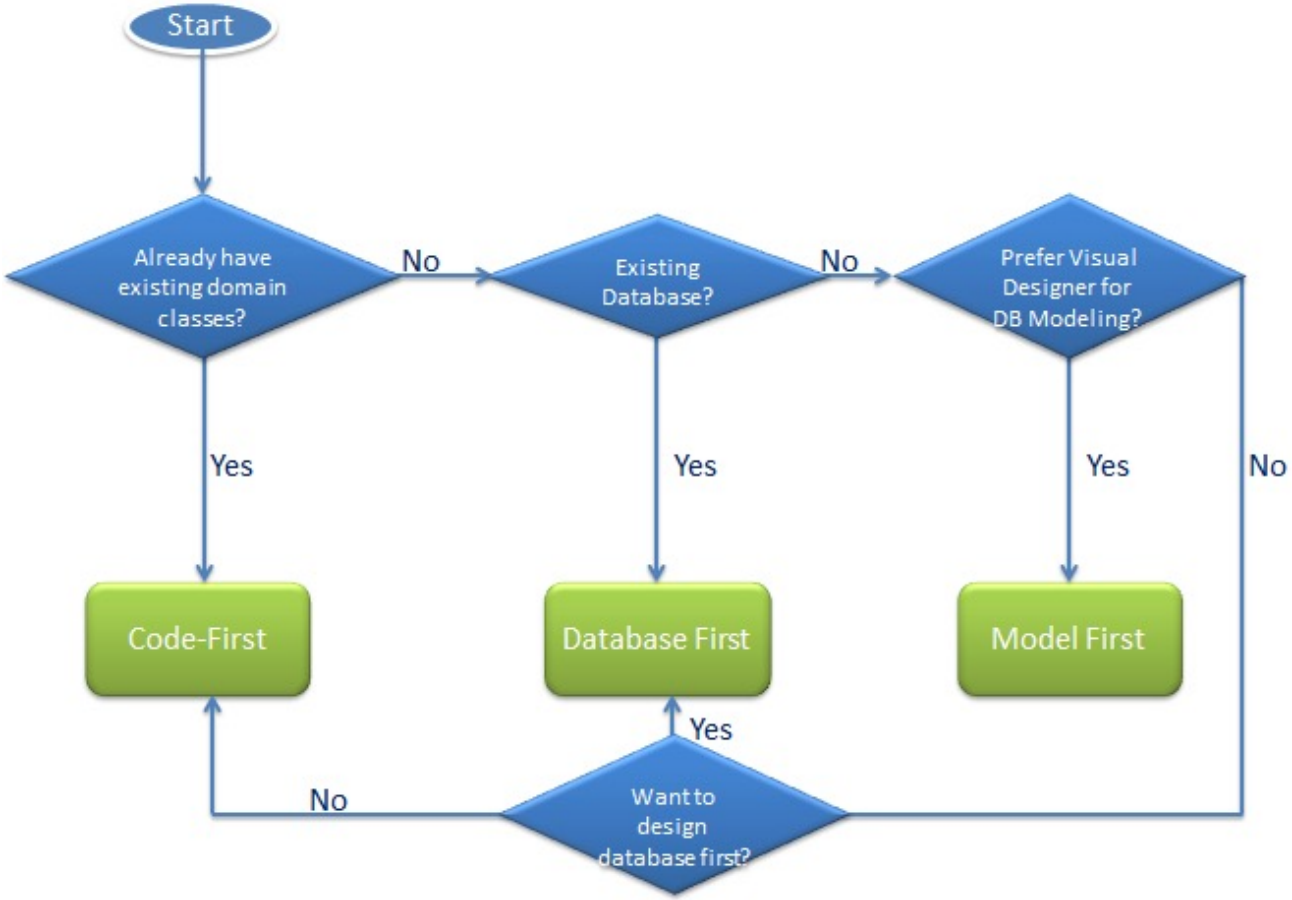


Approach choosing process

- Model,database or code first?



Approach choosing process



EF - Code First

- Code First approach
  - Write your business classes
  - Add relationships between classes
  - Add attributes
  - Add custom getters and setters where needed
  - EF will construct database based on your classes
  - EF will handle all the CRUD, converting code into SQL clauses

Code-First convensions

- Set of rules used to configure the conceptual model based on the code classes
- Namespace:

  `System.Data.Entity.ModelConfiguration.Conventions`

Code-First convensions

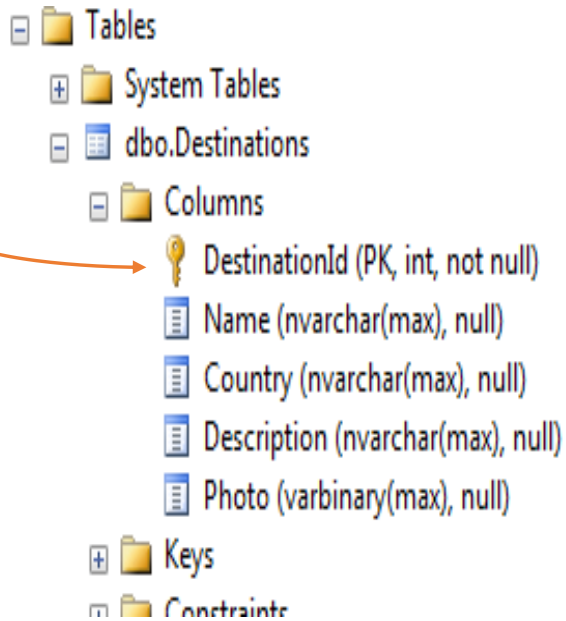
- **Table names Convension**
  - Convention for table naming is to use E.F Pluralization Service (rules for English language)
- **Columns Name Convension**
  - Create the columns using the same names as the class properties they map to
- **Type Discovery**
  - Code-First includes types defined as a DbSet property in context class.
  - Code-First includes reference types included in entity types even if they are defined in different assembly.
  - Code-First includes derived classes even if only the base class is defined as DbSet property.

Code-First convensions

- **Primary Key Convension**
  - Code-First would create a primary key for a property if the property name is `Id` or `<class name>Id` (NOT case sensitive).
- **Relationship Convension**
  - Code First infer the relationship between the two entities using navigation property. This navigation property can be simple reference type or collection type
- **Foreign Key Convension**
  - Code First automatically inserts a foreign key when it encounters a navigation property
- **Complex Type Convension**
  - Code First creates complex type for the class which does not include key property

Code-First convensions – Primary Key

```
public class Destination
{
    public int DestinationId { get; set; }
    public string Name { get; set; }
    public string Country { get; set; }
    public string Description { get; set; }
    public byte[] Photo { get; set; }
    public List<Lodging> Lodgings { get; set; }
}
```



Code-First convensions – Foreign Key

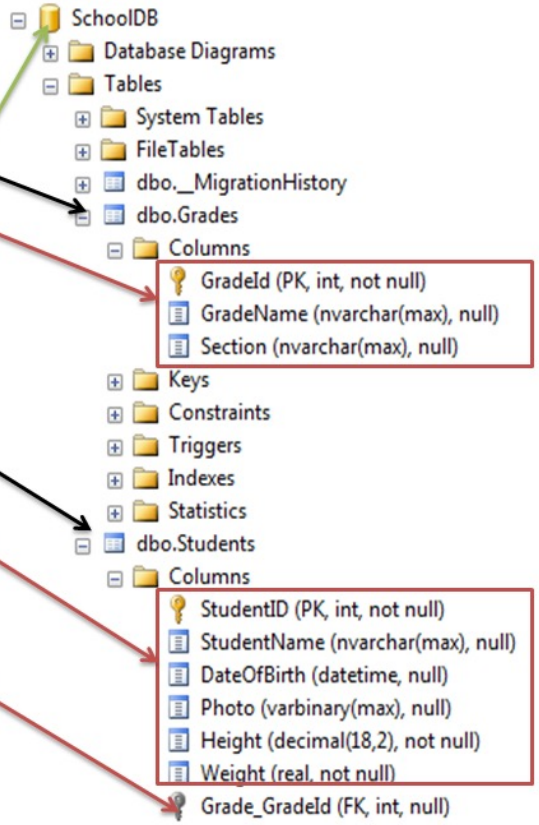
```
public class Jogador{
        public int JogadorId { get; set; }
        public string Nome { get; set; }
        public Equipa Equipa { get; set; }
}

public class Equipa{
        public int EquipaId { get; set; }
        public string Nome { get; set; }
        public string Pais { get; set; }
        public List<Jogador> Jogadores { get; set; }
}
```

Propriedade de navegação Referência

Propriedade de navegação Coleção

Code-First convensions

```
public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

    public ICollection<Student> Students { get; set; }
}
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public Grade Grade { get; set; }
}
public class SchoolContext : DbContext
{
    public SchoolContext() : base("SchoolDB")
    {

    }

    public DbSet<Student> Students { get; set; }
    public DbSet<Grade> Grades { get; set; }
}
```



http://www.entityframeworktutorial.net/code-first/code-first-conventions.aspx

Code-First convensions

- To define a relationship between two classes, we can use navigation properties with or without foreign keys.
  - **Bidirectional relationship** between two classes, if we have a pair of navigation properties that point to each other.
  - **Unidirectional relationship** between two classes, if we have a single navigation property.

Code-First convensions

- One-to-many relationship:
  - If there is a *Reference* navigation property and a *Collection* navigation property or just any of these
- Many-to-many relationship:
  - If there are two navigation properties *Collection*.
- One-to-one relationship:
  - If there are two navigation properties *Reference*.

Required EF Core packages

- ASP.NET Core middleware for EF Core

NuGet	`> Install-Package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore`
.Net CLI	`> dotnet add package Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore`

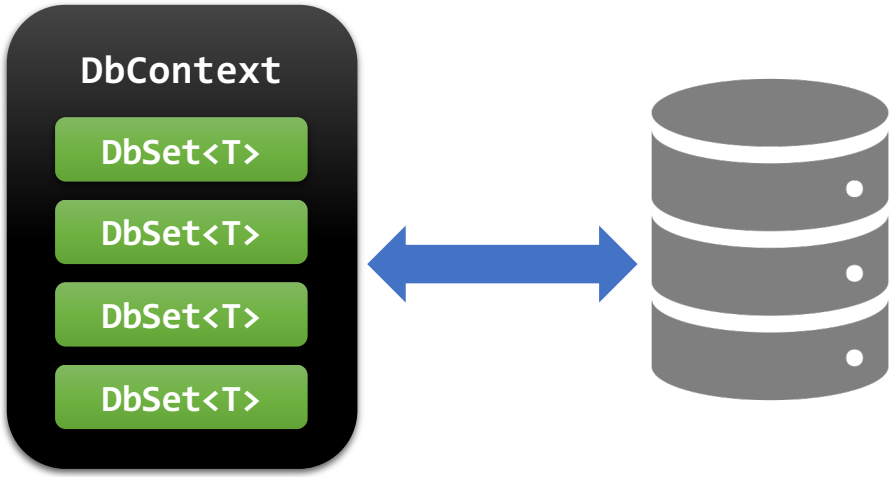
- Database providers
  - **SQL Server**: `Microsoft.EntityFrameworkCore.SqlServer`
  - **SQLite**: `Microsoft.EntityFrameworkCore.Sqlite`
  - **MariaDB**: `Pomelo.EntityFrameworkCore.MySql`
  - **MySQL**: `MySql.EntityFrameworkCore`
  - **Oracle**: `Oracle.EntityFrameworkCore`
  - **InMemory**: `Microsoft.EntityFrameworkCore.InMemory`
- Example for SQLServer

NuGet	`> Install-Package Microsoft.EntityFrameworkCore.SqlServer`
.Net CLI	`> dotnet add package Microsoft.EntityFrameworkCore.SqlServer`

EF and DbContext

- Entities have no idea of EF
- Add class, that inherits from *DbContext*
- Inside *DbContext*, add necessary *DbSet<Entity>* attributes
- Relational tables are created automatically

DbContext – database,
DbSet – table in database



DbContext class

- Derives from *DbContext*
- Refers *DbSet*
  - DbSet is a collection of entities

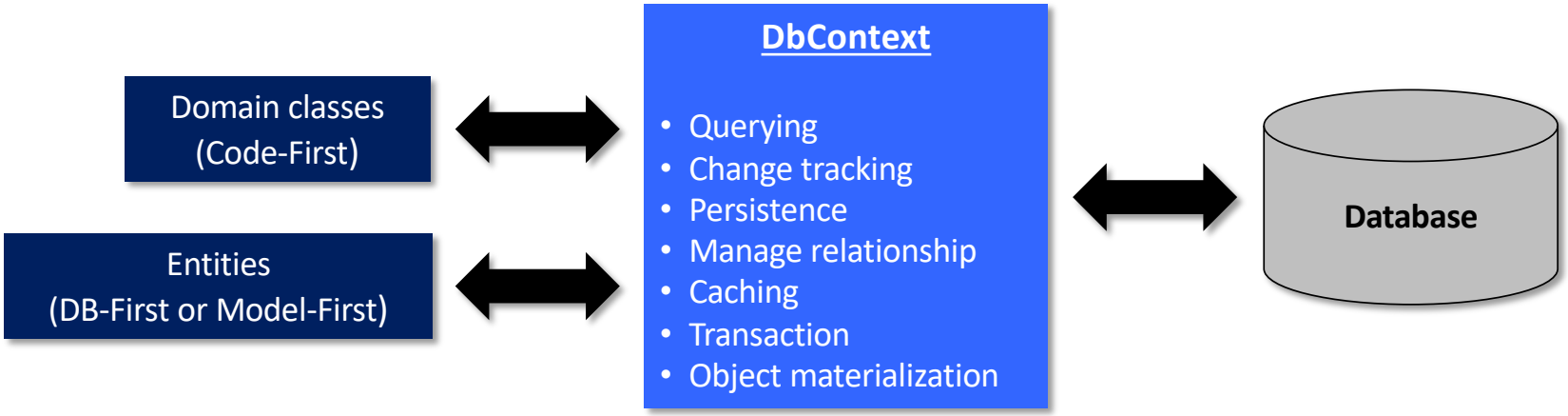
```
public DbSet<ProjectName.Models.Xpto> Xptos { get; set; }
```

- **Exposes *DbSet* properties to the types defined in the model**

```
public class theContext : DbContext
{
    public theContext (DbContextOptions<theContext> options) : base(options){
    }

    public DbSet<ProjectName.Models.Xpto> Xptos { get; set; }
}
```

What is DbContext?

- Class responsible for interacting with information.
- It establishes the bridge between the domain (entities) and the database.



- `Microsoft.EntityFrameworkCore.DbContext`

DbContext is responsible for

- Querying:
  - Converts LINQ-to-Entities queries to SQL query and sends them to the database.
- Change Tracking:
  - Keeps track of changes that occurred on the entities after querying from the database.
- Persisting Data:
  - Performs the Insert, Update and Delete operations to the database, based on entity states.

DbContext is responsible for

- Caching:
  - Provides first level caching by default. It stores the entities which have been retrieved during the lifetime of a context class.
- Manage Relationship:
  - Manages relationships using CSDL, MSL and SSDL in Db-First or Model-First approach and using fluent API configurations in Code-First approach.
- Object Materialization:
  - Converts raw data from the database into entity objects.

Context class

- Derives from *DbContext*
- Refers *DbSet*
  - DbSet is a collection of entities

```
public DbSet<nomeProjeto.Models.Xpto> Xptos { get; set; }
```

- **Exposes *DbSet* properties to the types defined in the model**

```
public class ContactContext : DbContext
{
   public ContactContext (DbContextOptions<ContactContext> options) : base(options){}

   public DbSet<Person> People { get; set; }
   public DbSet<Contact> Contacts { get; set; }
   public DbSet<ContactType> ContactTypes { get; set; }
}
```

Connection to the database

- Defined in the *Program.cs* class

```
using Microsoft.EntityFrameworkCore;
using TodoApi.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

builder.Services.AddDbContext<TodoContext>(opt =>
       opt.UseSqlite("DB/Catalog"));

builder.Services.AddEndpointsApiExplorer();
(…)
```

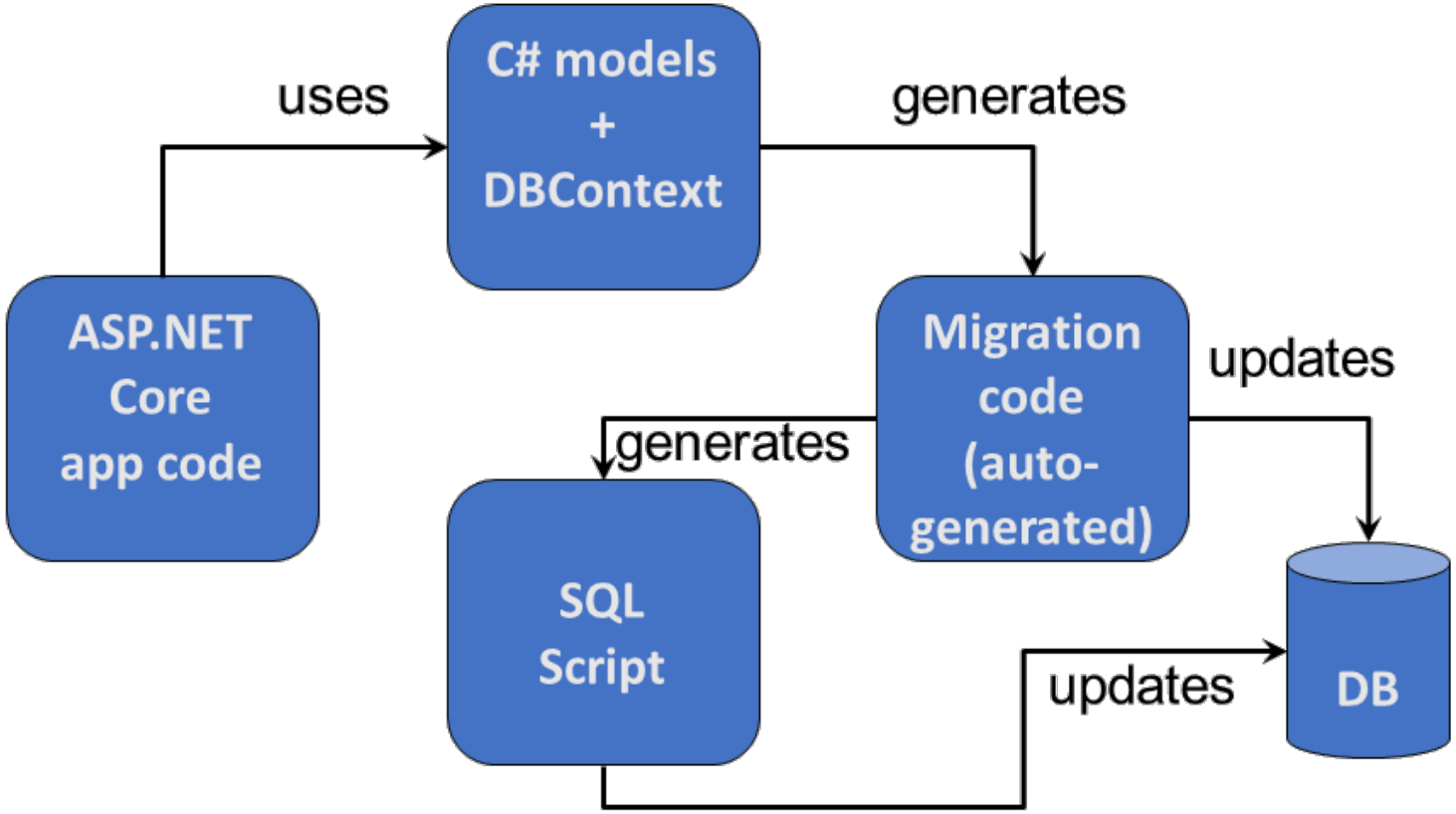
Code-First Migrations

- Code First Migrations is the recommended way to evolve your application's database schema if you are using the Code First workflow.
- Migrations provide a set of tools that allow:
  - Create an initial database that works with your EF model
  - Generating migrations to keep track of changes you make to your EF model
  - Keep your database up to date with those changes

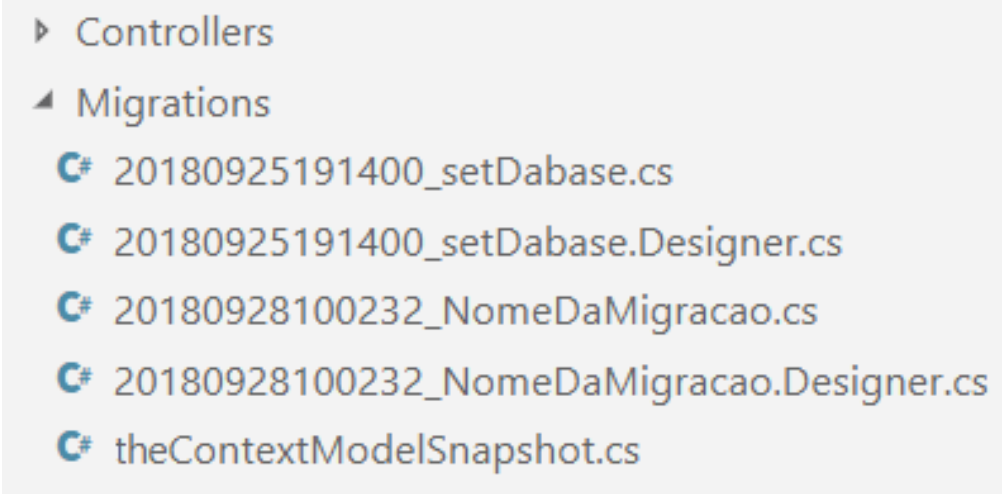
- Console commands:

```
> dotnet ef migrations add NomeDaMigracao
> dotnet ef database update
```

Code-First Migrations

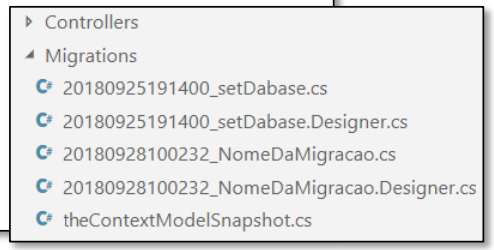


Code-First Migrations



Code-First Migrations – Migration class

```
(…)
public partial class NomeDaMigracao : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AddColumn<long>( name: "RefId", table: "theTable", nullable: true);
        migrationBuilder.CreateIndex(name: "IX_RefId", table: "theTable", column: "RefId");
        migrationBuilder.AddForeignKey(name: "FK_RefId",table: "theTable", column: "RefId",
                principalTable: "theTable", principalColumn: "Id",
                onDelete: ReferentialAction.Restrict);
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropForeignKey(name: "FK_RefId", table: "theTable");
        migrationBuilder.DropIndex(name: "IX_RefId", table: "theTable");
        migrationBuilder.DropColumn( name: "RefId", table: "theTable");
    }
}
(…)
```



Code-First Migrations - Commands

- `dotnet ef database update`
  - Apply the Migration to the database to create the schema
- `dotnet ef migrations add AddProductReviews`
  - Adding another migration
  - After making changes to your EF Core model, the database schema will be out of sync
  - To bring it up to date, add another migration. The migration name can be used like a commit message in a version control system
  - For example, if I made changes to save customer reviews of products, I might choose something like AddProductReviews
  - Don´t forget to Update-Database: `dotnet ef database update`

Code-First Migrations - Commands

- `dotnet ef migrations remove`
  - To remove the last migration
- `dotnet ef database update LastGoodMigration`
  - Reverting a migration
  - After making changes to your EF Core model, the database schema will be out of sync
  - dotnet ef migrations script
  - Generating a SQL script
  - To bring it up to date, add another migration. The migration name can be used like a commit message in a version control system
  - For example, if I made changes to save customer reviews of products, I might choose something like `AddProductReviews`

EF - Database initialization

- There are different strategies:
  - ***CreateDatabaseIfNotExists*** – default
  - ***DropCreateDatabaseIfModelChanges***
  - ***DropCreateDatabaseAlways***
  - ***MigrateDatabaseToLatestVersion***
  - … or create your own
- Set DB initializer in your *DbContext* class constructor

```
Database.SetInitializer<MyDBContext>(
        new DropCreateDatabaseIfModelChanges <MyDBContext>()
);
```

EF – Annotations and Fluent API

- **Annotations** are the simplest way to configure entities for EF.
- Several frameworks use the same annotations
  - MVC, Web Api, EF
- Annotations don't cover all options

```
using System.ComponentModel.DataAnnotations;
public class ContactType{
        [Required]
        [MaxLength(128)]
        public String Name { get; set; }
}
```

EF – Annotations and Fluent API

- **Fluent API** is capable of full EF configuration. But only EF sees those configurations.
- In some situations, it is necessary to use Annotations and Fluent API simultaneously

```
public class ContactConfiguration : EntityTypeConfiguration<Contact>
{
      public ContactConfiguration()
      {
            Property(t =>t.Value).IsRequired().HasMaxLength(128);
      }
}
```

EF - Key

- Sets that this attribute is used as primary key in Db
- Annotation:

```
[Key]
```

- Fluent:

```
Entity<T>.HasKey(t=>t.PropertyName)
```

EF - Required

- Attribute value is required, NULL is not allowed in Db
- Annotation:

```
[Required]
```

- Fluent:

```
Entity<T>.Property(t=>t.PropertyName).IsRequired;
```

EF - MaxLength, MinLength

- String minimum or maximum length.
- Annotation:

```
[MaxLength(value)]
[MinLength(value)]
```

- Fluent:

```
Entity<T>
    .Property(t=>t.PropertyName)
    .HasMaxLength(value);
```

EF - NotMapped

- If the attribute is synthetic (computed) and should not be stored in the database.
- Annotation:

```
[NotMapped]
```

- Fluent:

```
Entity.Ignore(d => d.Property);
```

EF - ForeignKey

- Used in case of multiple relationships of the same type or if entity names and key names do not match.
- Annotation:

```
[ForeignKey("fkname")]
```

- Fluent:

```
modelBuilder.Entity<>()
        .HasRequired()
        .WithMany()
        .HasForeignKey(fk => fk.fkname);
```

EF – Annotations and Fluent API

```
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration;
namespace ContactsLibrary
{

    public class PersonConfiguration : EntityTypeConfiguration<Person>{

        public PersonConfiguration()
        {
            Property(p => p.FirstName).IsRequired().HasMaxLength(150);
        }

    }

}
```

EF – Annotations and Fluent API

```
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration;
namespace ContactsLibrary
{
    public class ContactContext : DbContext{

        public DbSet<Person> People { get; set; }
        public DbSet<Contact> Contacts { get; set; }
        public DbSet<ContactType> ContactTypes { get; set; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Configurations.Add(new PersonConfiguration());
        }
    }
}
```

Fluent API only configurations

- Decimal precision (default 18,2)

```
Entity<T>.Property(t=>t.PropertyName).HasPrecision(m,n)
```

- Defining relationships

```
Entity.HasMultiplicity(Property).WithMultiplicity(Property)
```

Multiplicity: **Optional**, **Required**, **One**, **Many**

- Disabling cascade delete

```
Entity.HasRequired().WithMany().WillCascadeOnDelete(false)
```

- Changing conventions

```
modelBuilder.Conventions.Remove<PluralizingTableNameConvention>()
modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>()
```

Fluent API - Relationships

- One-to-Optional (Zero or One)

```
public class Profile {
   public int ProfileId { get; set; }
   public string Name { get; set; }
   public virtual Photo thePhoto { get; set; }
}
```

```
public class Photo {
   public int ProfileId { get; set; }
   public byte[] PhotoInfo { get; set; }
   public virtual Profile theProfile { get; set; }
}
```

```
protected override void OnModelCreating(DbModelBuilder modelBuilder) {
      modelBuilder.Entity<Photo>()
            .HasOne(x => x.theProfile)
            .WithOptional(x => x.thePhoto);
}
```

Fluent API - Relationships

- One-to-One

```
public class User {
   public int UserId { get; set; }
   public String FullName;
   public String Email;
   public Profile theProfile { get; set; }
}
```

```
public class Profile {
   public int ProfileId { get; set; }
   public string Name { get; set; }
   public User theUser { get; set; }
}
```

```
protected override void OnModelCreating(DbModelBuilder modelBuilder) {
      modelBuilder.Entity<User>()
            .HasKey(u => u.UserId);
      modelBuilder.Entity<Profile>()
            .HasRequire(x => x.theUser)
            .WithRequiredPrincipal(x => x.theUser);
}
```

Fluent API - Relationships

- Many-to-Many

```
public class Course {
   public int CourseId { get; set; }
   public String Name;
   public List<Instructor> Instructors { get; } = new();
}
```

```
public class Instructor {
   public int InstructorId { get; set; }
   public string Name { get; set; }
   public List<Course> Courses { get; } = new();
}
```

```
protected override void OnModelCreating(DbModelBuilder modelBuilder) {
      modelBuilder.Entity<Course>()
            .HasMany(t => t.Instructors)
            .WithMany(t => t.Courses)
}
```

EF – Logging

- Override ***OnConfiguring*** method in *DbContext* class

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder
            .LogTo(Console.WriteLine)
            .EnableSensitiveDataLogging()   //to log sensitive data
            .EnableDetailedErrors();        //get detailed query exceptions
```

EF - Db connection

- EF system configuration is in *appsettings.json* file
- Add a connection string

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
       "SqliteCS":"Data Source=DB/Catalog.db;",
       "OracleCS":"Data Source=Catalog;Integrated Security=SSPI;"
  }
}
```

C# - EF, Db connection

- Use it in *Program.cs*

```
(…)
string ConnStr = builder.Configuration.GetConnectionString("SQLiteCS");

builder.Services
       .AddDbContext<JokesContext>(opt => opt.UseSqlite(ConnStr));
(…)
```

References

- *Creating and Configuring a Model - EF Core | Microsoft Learn*
- *Entity Framework - Fluent API | Tutorialspoint*
- *Fluent API - Configuring and Mapping Properties and Types - EF6 | Microsoft Learn*
- *Fluent API in Entity Framework Core (entityframeworktutorial.net)*
- *Many-to-many relationships - EF Core | Microsoft Learn*

Handling Authentication and Authorization with Node and Angular

Authentication vs Authorization

- Authentication: determines who the user claims to be.
- Authorization: determines who (authenticated) has access to what.

1. Authentication

- Verifies you are who you say you are
- Methods:
  - Login form
  - HTTP authentication
  - HTTP digest
  - X.509 certificates
  - Custom authentication method

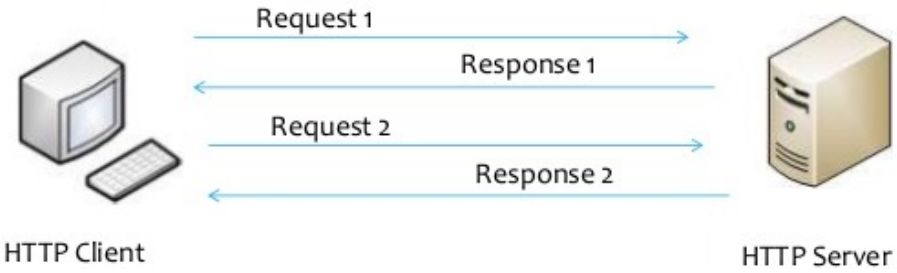
2. Authorization

- Decides if you have permission to access a resource
- Methods:
  - Access controls for URL's
  - Secure objects and methods
  - Access control lists (ACL's)

cf. http://www.differencebetween.net/technology/difference-between-authentication-and-authorization/

Web App Authentication

- HTTP protocol is stateless
  - Don’t remember previous request/response chain

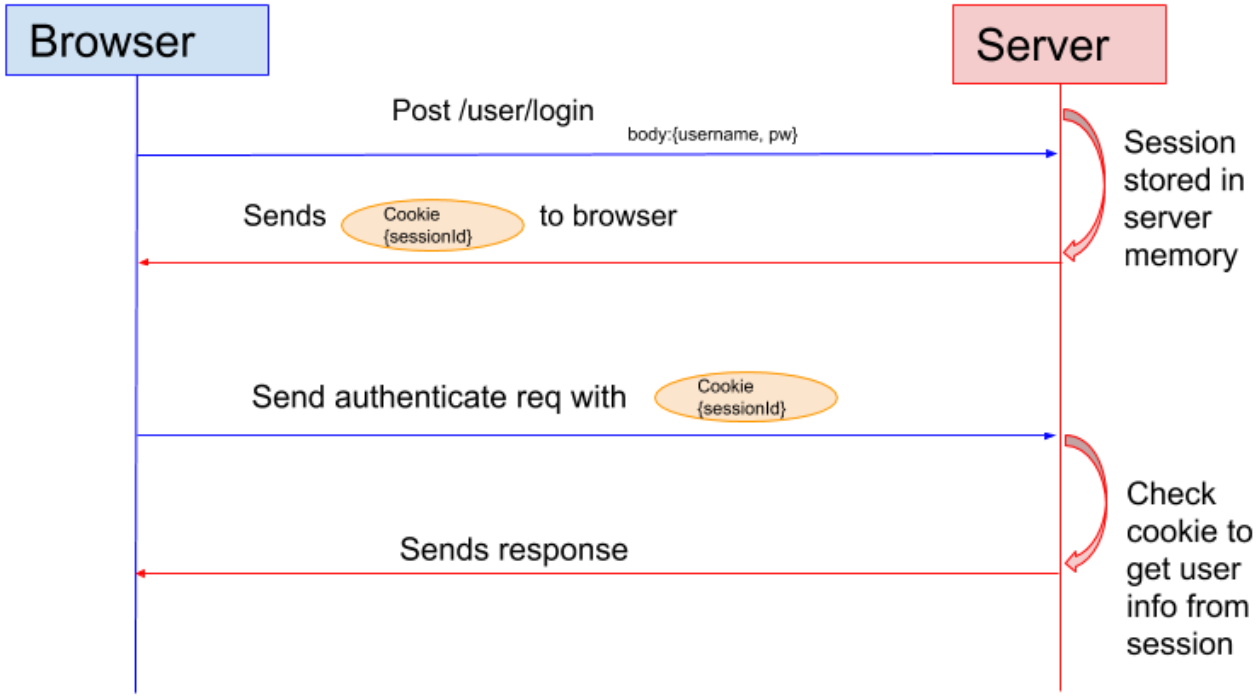


- Therefore, if we authenticate a user with a username and password, then on the next request, our application will not know who they are. He would have to authenticate again, and again, and again, ...

Session Based Authentication

- The server will create a session for the user after the user logs in;
- The session id is then stored on a cookie on the user's browser;
- While the user stays logged in, the cookie would be sent along with every subsequent request. The server can then compare the session id stored on the cookie against the session information stored in the memory to verify user's identity and sends response with the corresponding state
- Issues:
  - *Scalability is affected* - as sessions are stored in the server's memory, scaling becomes an issue when there is a huge number of users using the system at once
  - *Cross-Origin Resource Sharing (CORS)*
  - *Cross-Site Request Forgery (CSRF) attacks*

Session Based Authentication

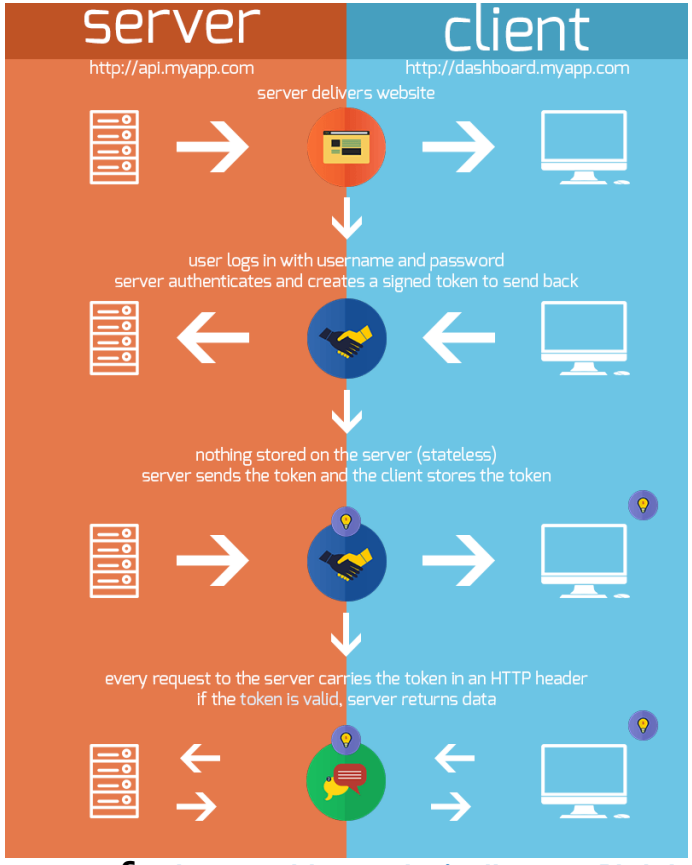


cf. https://medium.com/@sherryhsu/session-vs-token-based-authentication-11a6c5ac45e4

Token-based authentication

- User Requests Access with Username / Password
- Application validates credentials
- Application provides a signed token to the client
- Client stores that token and sends it along with every request
- Server verifies token and responds with data
- Benefits:
  - *Stateless and Scalable* - Tokens are stored on client side
  - *Security* - There is no cookie being sent, this helps to prevent CSRF attacks
  - *Extensibility* - Tokens will allow to build applications that share permissions with anothe

Token-based authentication



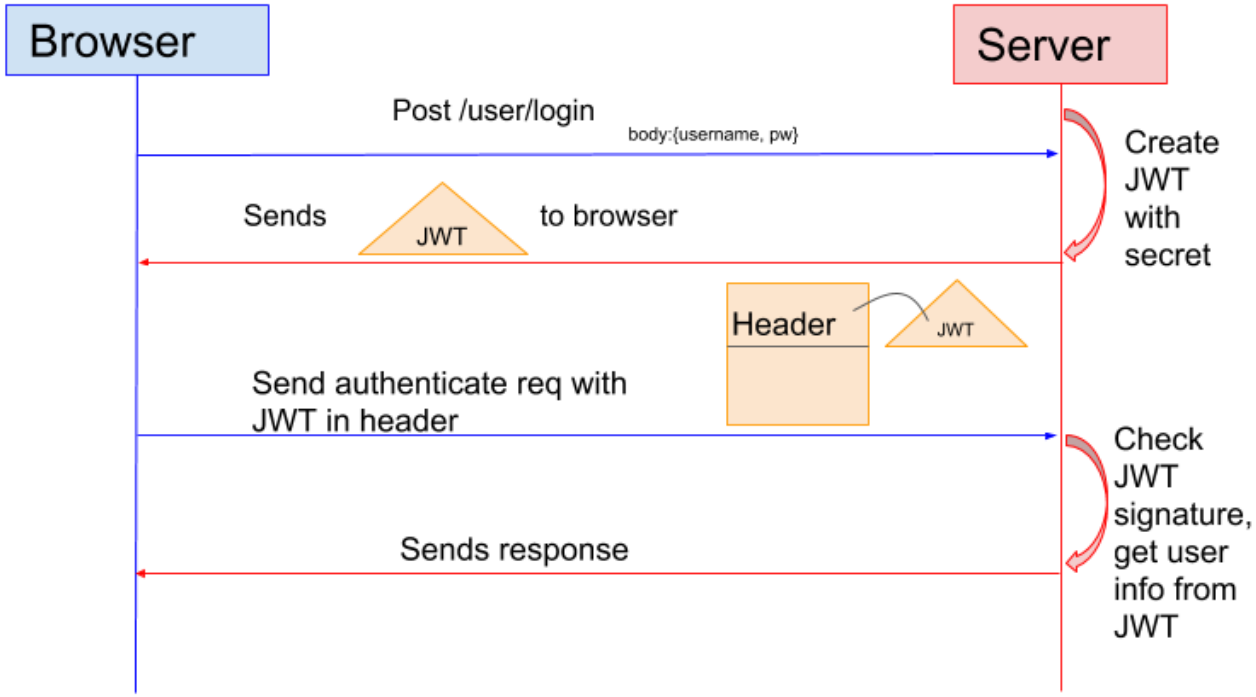
cf. https://scotch.io/bar-talk/the-ins-and-outs-of-token-based-authentication

JSON Web Tokens - JWT

- JSON Web Token is an open standard (RFC 7519) for securely transmitting information between endpoints as JSON object.
- A JSON Web Token consist of three main parts separated by dots (`.`):
  - the header;
  - the payload;
  - the signature.
- On the server end the token is verified by re-encrypting the payload with the crypto info and checking to see if it matches the signature. Any change to the payload will negate the signature and invalidate the token.
- The token is secure because the Salt is only known to the server so resigning a fake token is virtually impossible.

cf. https://scotch.io/tutorials/the-anatomy-of-a-json-web-token

JSON Web Tokens - JWT



cf. https://medium.com/@sherryhsu/session-vs-token-based-authentication-11a6c5ac45e4

JSON Web Tokens - JWT

- JSON Web Token is an open standard (RFC 7519) for securely transmitting information between endpoints as JSON object.
- A JSON Web Token consist of three main parts separated by dots (`.`):
  - the ***header***
    - Has the type of the token which is JWT, and the algorithm being used such as RSA, HMAC, or SHA256.
  - the ***payload***
    - Has the data. Additional optional information known as claims such as the expiration time (exp), subject (sub), or issuer (iss) can be added.
  - the ***signature***
    - Is the sum of the encoded header, the encoded payload, a secret, and lastly the algorithm which is specified in the header.

JSON Web Tokens - JWT

```
// JWT high-level representation
header.payload.signature
```

```
// How JWT looks
eyJhbGciOBJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImpvaG4gZG9lIiwiaWF0IjoxNGYyMjMyNzI2
fQ.NnOv-wHAf59L2WMcDlfNsTThOUY1a0JMFNgJIP67mqU
```

Some tutorials

- Handling Authentication and Authorization with Node
  - https://medium.com/quick-code/handling-authentication-and-authorization-with-node-7f9548fedde8
- Authenticate a Node ES6 API with JSON Web Tokens
  - https://scotch.io/tutorials/authenticate-a-node-es6-api-with-json-web-tokens
- Securing Node.js RESTful APIs with JSON Web Tokens
  - https://medium.freecodecamp.org/securing-node-js-restful-apis-with-json-web-tokens-9f811a92bb52
- route-middleware
  - https://github.com/expressjs/express/blob/master/examples/route-middleware/index.js
- VinceZK
  - https://github.com/VinceZK/authorization

Example

- Required Node.JS Modules
  - `Config`
    - Used to retrieve *`jwtPrivateKey`*
  - `joi`
    - Validation function
  - `jsonwebtoken`
    - Generate and verify JWT
  - `bcrypt`
    - Hashing the password to store in the database
- To Install:

```
C:\> npm install config joi jsonwebtoken bcrypt --save
```

Example

- Model Class `user.js`

user.js

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

module.exports = mongoose.model('User', new Schema({
    name: String,
    password: String,
    email: String,
    isAdmin: Boolean
    })
);
```

Example

- Refer new routes

server.js

```
(…)
var UserRoutes = require('./app/routes/UserRoutes');
app.use('/api/users',UserRoutes);
(…)
```

- Create new routes in `UserRoutes.js`

UserRoutes.js

```
var express = require('express');
var jwt = require('jsonwebtoken');
var bcrypt = require('bcryptjs');
var UserModel = require('../models/user');
var VerifyToken = require('../auth/VerifyToken');

var router = express.Router();
(…)
```

Example

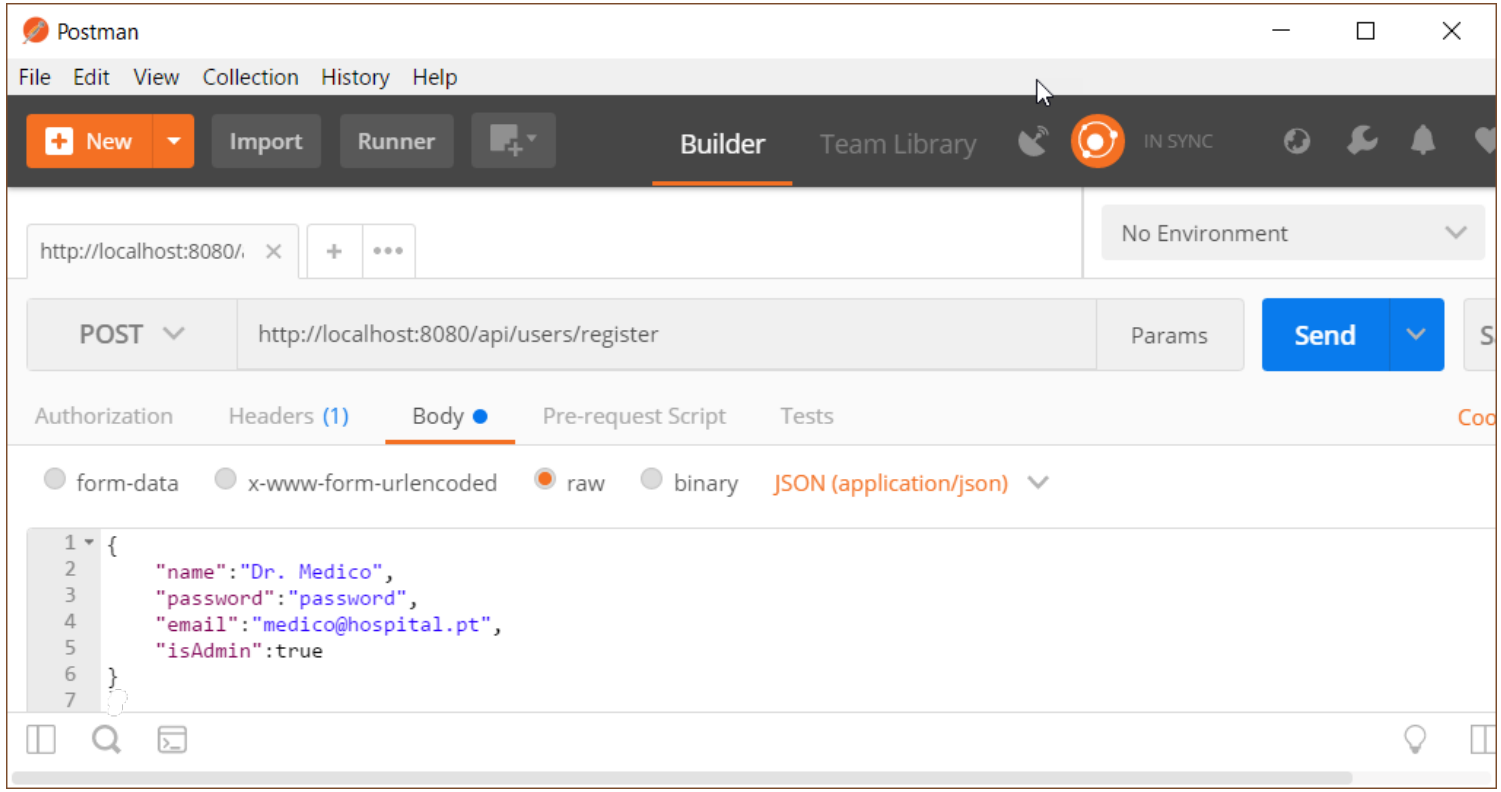
- Route `/api/users/register`

UserRoutes.js

```
(…)

router.post('/register',function(req,res){
    var user= new UserModel();
    user.name = req.body.name;
    user.password = bcrypt.hashSync(req.body.password,8);
    user.email = req.body.email;
    user.isAdmin = req.body.isAdmin;
        user.save(function (err){
        if(err)
            return res.status(500).send("There is a problem registering the user.");
        res.json({message:"User registered!"});
    })
});
```

Example

- Route `/api/users/register`



Example

- Route /api/users/login

UserRoutes.js

```
(…)
router.post('/login',function (req,res){
  UserModel.findOne({email: req.body.email}, function(err,user){
    if (err) throw err;
    if (!user) {res.json({success:false,message:'Authentication failed.'});}
    else if(user){
           if(!bcrypt.compareSync(req.body.password, user.password))
          return res.status(401).send({auth:false,token:null,message: 'Auth failed.'});
       else {
          const payload= {user:user.email};
          var theToken = jwt.sign(payload, 'TheSecret_123456789', {expiresIn:86400});
          res.json({success:true,message:'Enjoy your token!',token:theToken});
       }
    }
  });
});
```

Example

- Route `/api/users/login`

POST http://localhost:8080/api/users/login Params Send

Authorization Headers (1) Body Pre-request Script Tests

```
1 {
2     "password":"password",
3     "email":"medico@hospital.pt"
4 }
```

Body Cookies (1) Headers (5) Test Results Status: 200 OK Time: 122 ms


```
1 {
2     "success": true,
3     "message": "Enjoy your token!",
4     "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
          .eyJ1c2VyIjoibWVkaWNvQGhvc3BpdGFsLnB0IiwiaWF0IjoxNTA4OTIxMjA3LCJleHAiOjE1MDkwMDc2MDd9.wTsCO
          -To_oXOj5j0R3mtkcmVghwEhv7a7UV6PyvWTIw"
5 }
```

Example

- Route `/api/users/`

UserRoutes.js

```
(…)
function hasRole(userEmail, role, func){
    UserModel.findOne({email: userEmail}, function (err, user){
        if (err) throw err;

        if(!user){
            res.json({success: false, message: 'Authentication failed.'});
        }
        else if (user) {
            func(role === 'administrator' && user.isAdmin === true)
        }
    })
}
```

Example

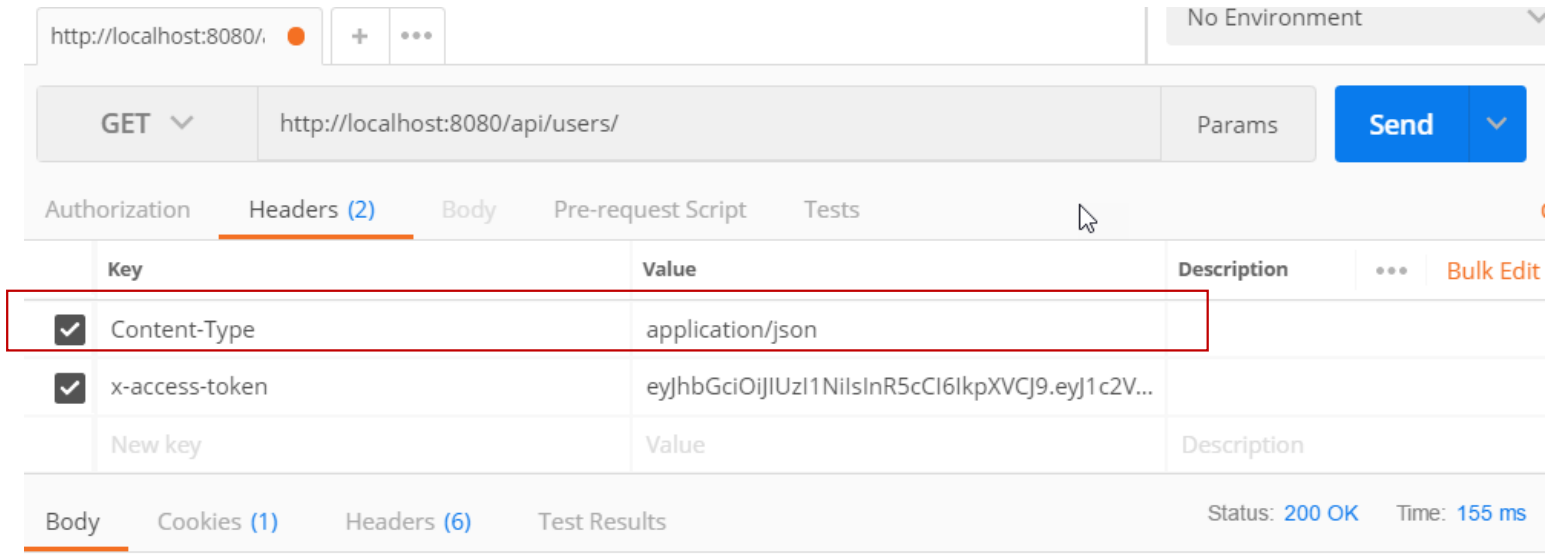
- Route /api/users/

UserRoutes.js

```
(…)

router.get('/', VerifyToken, function(req, res){
    hasRole(req.user, 'administrator', function (decision) {
        if (!decision)
            return res.status(403).send(
                    {auth:false, token: null, message: 'You have no authorization.'}
                );
        else
            UserModel.find(function (err, users){
                if (err) res.send(err);
                res.json(users);
            })
    });
})
module.exports = router;
```

Example

- Route `/api/users/`



Example

- Middleware

VerifyToken.js

```
var jwt = require('jsonwebtoken');

function verifyToken(req, res, next){
    var token = req.headers['x-access-token'];
    if (!token)
        return res.status(403).send({auth:false, message:'No token provided'});

    jwt.verify(token, 'TheSecret_123456789', function(err, decoded){
        if (err)
            return res.status(500).send({auth:false, message:'Failed to authenticate token.'});
        req.user = decoded.user;
        next();
    });
}

module.exports = verifyToken;
```

Angular Authentication using JWT

Authentication service

- Putting all client authentication logic in a single AuthService will help to keep code tidy.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { BehaviorSubject, catchError, Observable, of } from 'rxjs';
import { User } from 'src/app/Domain/User';
import { map } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class AuthService {

    private userSubject: BehaviorSubject<User | null>;
    public user: Observable<User | null>;

    constructor(private http: HttpClient) {
      this.userSubject = new BehaviorSubject<User | null>(
        JSON.parse(localStorage.getItem('user')!)
      );
      this.user = this.userSubject.asObservable();
        }
        (…)
}
```

Authentication service

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { BehaviorSubject, catchError, Observable, of } from 'rxjs';
import { User } from 'src/app/Domain/User';
import { map } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class AuthService {
        (…)
        login(email:string, password:string ) {
                return this.http.post<any>(this.theUrl, {email, password})
                .pipe(
                        map(user => localStorage.setItem('user', JSON.stringify(user))),
                        catchError(this.handleError<User>("login"))
                );
        }
        logout() { localStorage.removeItem('user'); }
        getToken(): string | null  {
                return localStorage.getItem("user") ? JSON.parse(localStorage.getItem("user")!).token :null;
        }
        (…)
}
```

Authentication service

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { BehaviorSubject, catchError, Observable, of } from 'rxjs';
import { User } from 'src/app/Domain/User';
import { map } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class AuthService {

        (…)

        private handleError<T>(operation = 'operation', result?: T) {
        return (error: any): Observable<T> => {
                this.logout();
                console.error(error);
                console.log(`${operation} failed: ${error.message}`);
                return of(result as T);
      };
        }
}
```

Sending the JWT to the server on each request

- The JWT is in the client browser.
- Let's use it to inform the server that a certain HTTP request belongs to a certain user.
- Solution: in every HTTP request sent to the server attach the JWT!
  - To do this, use an Angular HTTP Interceptor.

https://medium.com/engineerbabu/angular-authentication-using-jwt-d846c5ce0ac6

Build the Authentication HTTP Interceptor

```
import { Injectable } from '@angular/core';
import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from '@angular/common/http';
import { Observable } from 'rxjs';
import { AuthService } from '../Modules/authentication/auth.service';

@Injectable()
export class AuthenticationInterceptor implements HttpInterceptor {

  constructor(private authSrv:AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = this.authSrv.getToken()
    if (token) {
      const cloned = req.clone({headers: req.headers.set("Authorization", "Bearer " + token)});
      return next.handle(cloned);
    }
    else {
      return next.handle(req);
    }
  }

}
```

Defining the Guard

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
import { AuthService } from '../Modules/authentication/auth.service';

@Injectable({ providedIn: 'root' })
export class VerifyAuthService implements CanActivate {

        constructor(private authservice: AuthService, private router: Router()) { }

        canActivate(route: ActivatedRouteSnapshot, state:RouterStateSnapshot): boolean{
                // check If the user is logged in
                if(localStorage.getItem('user')) { return true; }
                //not logged in so redirect to login page
                this.router.navigate(['/login'], { queryParams: { returnUrl: state.url }});
                return false;
        }
}
```

app.routing.module.ts

```
const routes: Routes = [
  { path: '', component: BaseComponent , canActivate: [AuthGuard] }
];
```

Error interceptor

- The HTTP error interceptor works with the calling service and the API's
- It intercepts the responses from the API and checks for the status codes.
- Error status:
  - 200: Success
  - 401: Unauthorized Response — the user will be automatically logged out.
  - All other errors are rethrown, to be caught by the calling service.
    - An alert can be displayed by the user service/component on the client's end.

Error interceptor

```
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpHandler, HttpRequest } from '@angular/common/http';
import { Observable } from 'rxjs';
import { AuthService } from '../Modules/authentication/auth.service';

@Injectable()
export class ErrorInterceptor implements HttpInterceptor {

        constructor(private authservice: AuthService) {}

        intercept(req: HttpRequest<any>, next: HttpHandler) Observable<HttpEvent<any>>{
           return next.handle(req).pipe(
                 catchError( err => {
                          if(err.status === 401){// auto logout on unauthorized response
                                   this.authservice.logout();
                                   location.reload(true);
                                   }
                          return throwError(err.error.message || err.statusText);
                 })
            )
        }
}
```