

Web Programming

Lecture 0

Introduction to Web Technologies

Josue Obregon 

jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

 [Lecture slides index](#)

March 4, 2025



Agenda

- Lecturer and students
- Course logistics
 - Objectives
 - Methodology
 - Class contents
 - Evaluation
 - Resources
 - Web Development Tools
- Introduction to Web Technologies

Lecturer Presentation

- BSC in Computer Engineering
 - Universidad de San Carlos de Guatemala
- MSc in Industrial & Management Systems Engineering
 - Business Process Management Laboratory
 - Process mining applied to SNS
- PhD in Industrial & Management Systems Engineering
 - Industrial Artificial Intelligence Laboratory
 - Interpretable Machine Learning for tree ensembles

 Prof. Josue Obregon

Tel: 02-970-7291

Office: Changhak Hall (3), Room 334-1

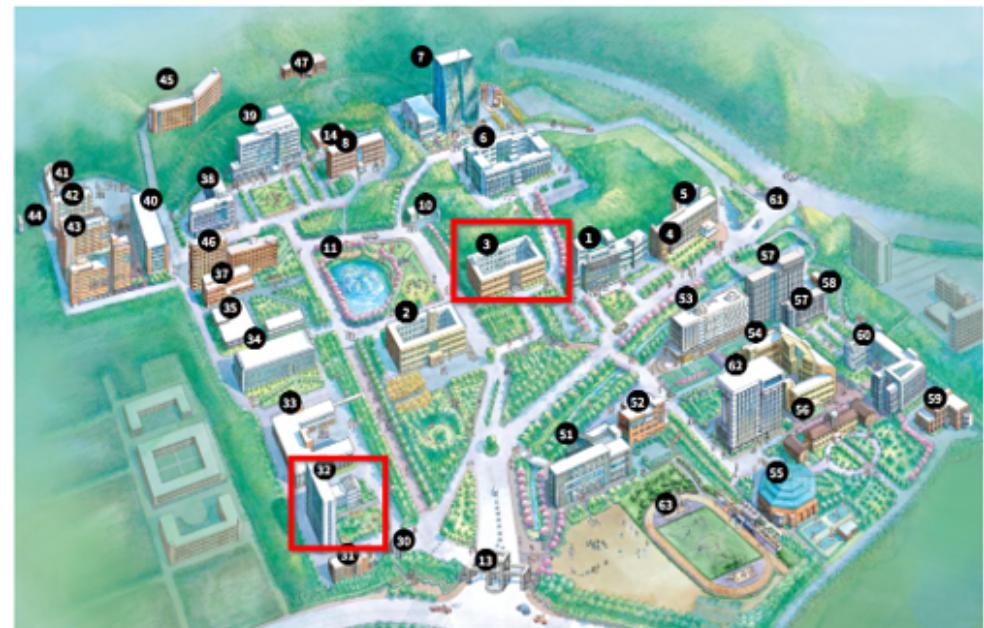
Office Hours: Monday 14:00 – 17:00

(come anytime!) **Home-page:**

<https://eis.seoultech.ac.kr/>

Email: jobregon@seoultech.ac.kr

Publications: [Google Scholar Profile](#)



Students

- Question for students:
 - Which part of a website (design, interactivity, or data handling) are you most curious about and why?
 - What's one programming concept from Java that you think will help you in web development?
 - Have you heard of or used any web tools (like HTML, CSS, or JavaScript) before? If so, which one interests you the most?

Every great developer you know got there by solving problems they were unqualified to solve until they actually did it.

— Patrick Mackenzie, Software Engineer

Course objectives

By the end of this course, students will be able to:

1. Design and implement web applications using **JavaScript**, **SQL**, **HTML**, **CSS**, and **React**.
2. Apply best practices in application design, including scalability, security, and user experience optimization.
3. Write and utilize APIs, manipulate the **DOM**, and create **dynamic**, **interactive** user interfaces.
4. Test, containerize, and deploy web applications on the Internet using modern CI/CD workflows.

The end result? A better understanding of the web, important technologies, and a portfolio for you to show!

Methodology



- Flipped Clasroom style
- (1 ~ 1.5 hours) Watch online lectures and complete readings before class.
- (1.5 ~ 2 hours) Offline activities (11:00 am)
 - Initial Quiz: Formative assessment of basic concepts and syntax.
 - Interactive Q&A: discussions and to clarify doubts.
 - Programming Quizzes: Take regular quizzes to reinforce concepts.
 - Hands-On Practice: Engage in coding exercises to apply what you've learned.
- Teaching Philosophy

active learning ↔ passive learning

Class contents *whole Semester*

1. Intro to Web Technologies + HTML
2. CSS
3. Introduction to JavaScript
4. DOM Manipulation with JavaScript
5. Asynchronous JavaScript, JSON and APIs
6. Introduction to Node.js
7. Node.js and Express
8. **Midterm**
9. Data Persistence in Node JS
10. User Authentication and Securing Web applications
11. Introduction to React
12. Deploying Web Applications - Testing and CI
13. Deploying Web Applications - CD, Containerization and more on Security
14. Project Presentation
15. **Final examination**

Evaluation

Point Distribution

Assesment	Points
-----------	--------

Midterm	30%
Homework	15%
Term project	25%
Final exam	30%
Total	100%

focusing on coding
mini project
n projects ?
focusing on concepts

Grading guidelines

SeoulTech Grades	Marks (100)	NU Grades
A+ (4.5)	above 70	First
A0 (4.0)		
B+ (3.5)	above 60	Upper Second
B0 (3.0)		
C+ (2.5)	above 50	Lower Second
C0 (2.0)		
D+ (1.5)	above 40	Third
D0 (1.0)		
F (0.0)	under 40	Fail

Resources

- **Class website** (continuously updating)
- eclass for homeworks, announcements, quizzes and attendance
- Recommended books
 - [Learning JavaScript Design Patterns, 2nd Edition](#), Addy Osmani Published by O'Reilly Media, Inc.
 - [Node.js for Beginners](#), Ulises Gascon, Published by Packt Publishing

js → node.js

Web development tools

- **Chrome:** a browser to view and debug web pages
- **VSCode:** a text editor to write HTML/CSS/JS/SQL (with various helpful packages available)
- **Command Line with Git:** to clone/push assignments repositories (GitHub Clasroom)

Introduction web technologies

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <title>Web Programming ITM</title>
5   </head>
6   <body>
7     Welcome to Spring Semester!
8   </body>
9 </html>
```

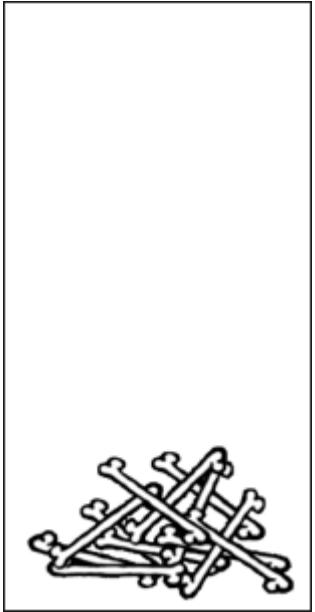
Global overview

" interact with customer through web app.

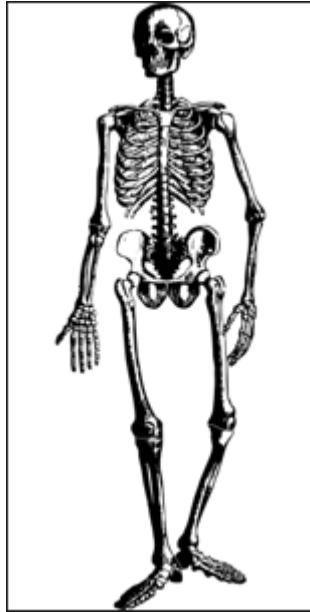
- Why do we study Web Programming in ITM?
 - In-demand skills
 - Allows businesses to establish an online presence, welcoming traffic and converting it into leads
 - It is fun to create websites (well, sort of...)
- What are the tools and technologies developers are currently using and the ones they want to use.
 - Stack Overflow Developer Survey 2024

React / Node.js

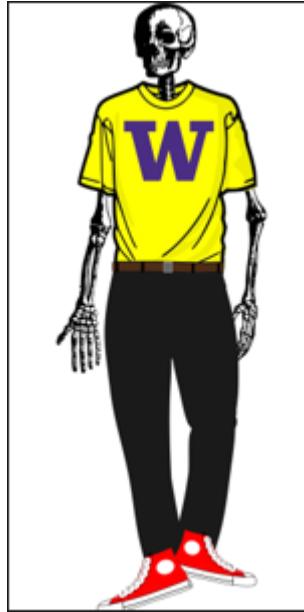
What is a web page?



Content



Structure



Style

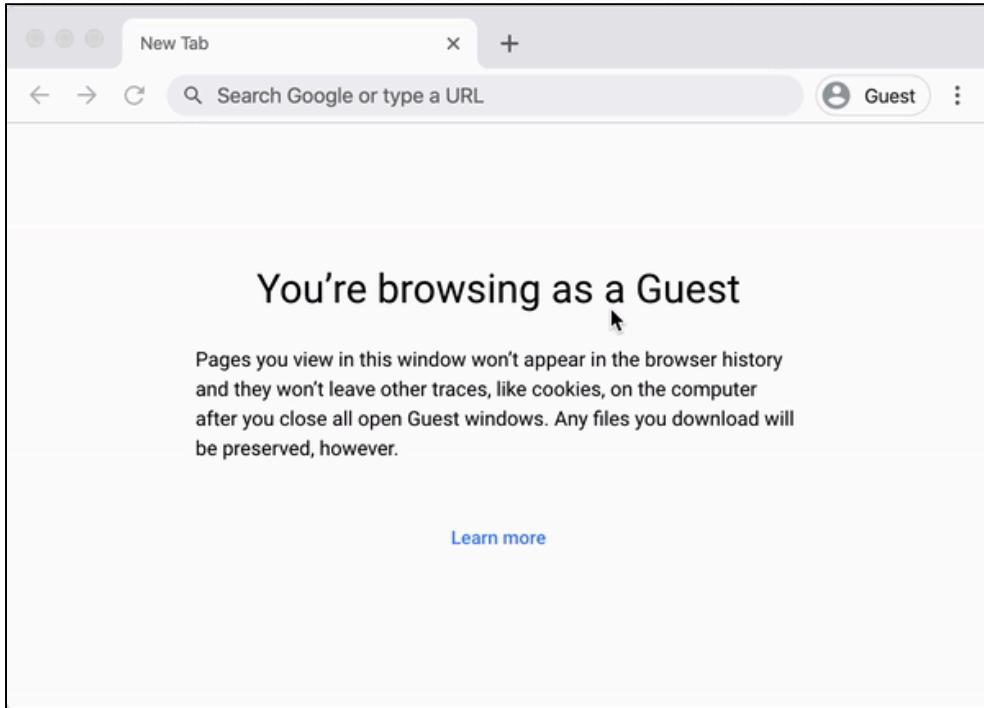


dancing skeletons

Behavior

A real web page?

What's everything involved here?



It's just this, right?

1. Decide on URL...
2. Type it in...
3. Hit enter...
4. Website loads!



But what happens between 3 and 4?

Behind the scenes

You don't have Google.com on your computer. So, where does it come from?

1. Figure out where it is
2. Ask for it to be sent to us
3. Check and verify what we get
4. Show it

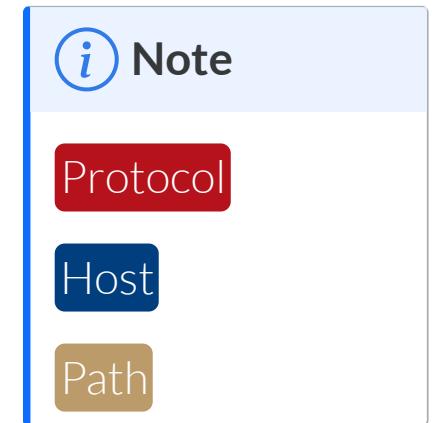
The text in the address bar

- It indicates where to find the website
- **Uniform Resource Locator (URL)**: An identifier for the location of a document


host contents

https://itm.seoultech.ac.kr/about_itm/about_professor

https://jobregon1212.github.io/lecture-slides/webprog/



URL elements

https://itm.seoultech.ac.kr/about_itm/about_professor

We've handled the host to IP address (so we know who to ask for the web page) The **“protocol”** tells us how:

set of rules to communicate.

- **HTTP: HyperText Transfer Protocol**
- Gives us the instructions (protocols) for how to share (transfer) web content (“hypertext”)

HTML 웹페이지 전송 규칙.

And the rest tells us what:

- From the itm.seoultech.ac.kr server (aka **host**)...
- Now we specifically want about_itm/about_professor (aka **path** or resource)

Internet vs. The Web

Internet

Computers (servers) connected to each other via a series of networks

Powered by layers upon layers:

- Physical: The cables between them
- Data & Network: The packets of information
- Transport (TCP/IP): Providing connections and reliability
- Application: Tying everything together to be useful

The Web

- Collection of pages of information
- Text... but with some “Hyper” around it
- Pages can link to each other
- Pages have style and interactivity

Remember that URL? <https://google.com/>

Need to go out to the internet to get the webpage.

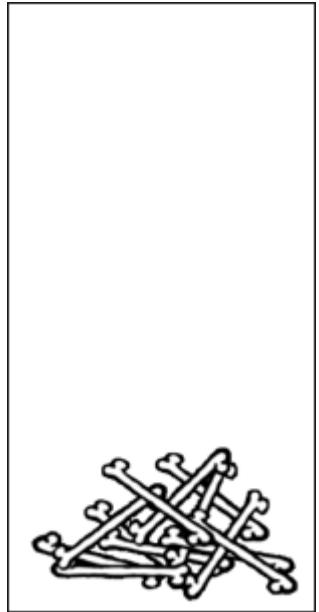
Internet is low-level: based on numbers (IP addresses), not names.

Domain Name System (DNS)

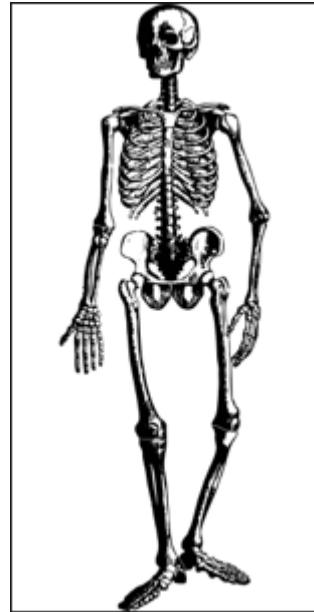
A Domain Name System translates human-readable names to IP addresses

- Example: `itm.seoultech.ac.kr` → `203.246.83.174`
- Hostname of `itm.seoultech.ac.kr` (which we might put into the browser's address bar) ... has IP address of 203.246.83.174 (which will be used to contact the server via the internet)

Then we have a web page right?



Content
Words + images



Structure
HTML



Style
CSS



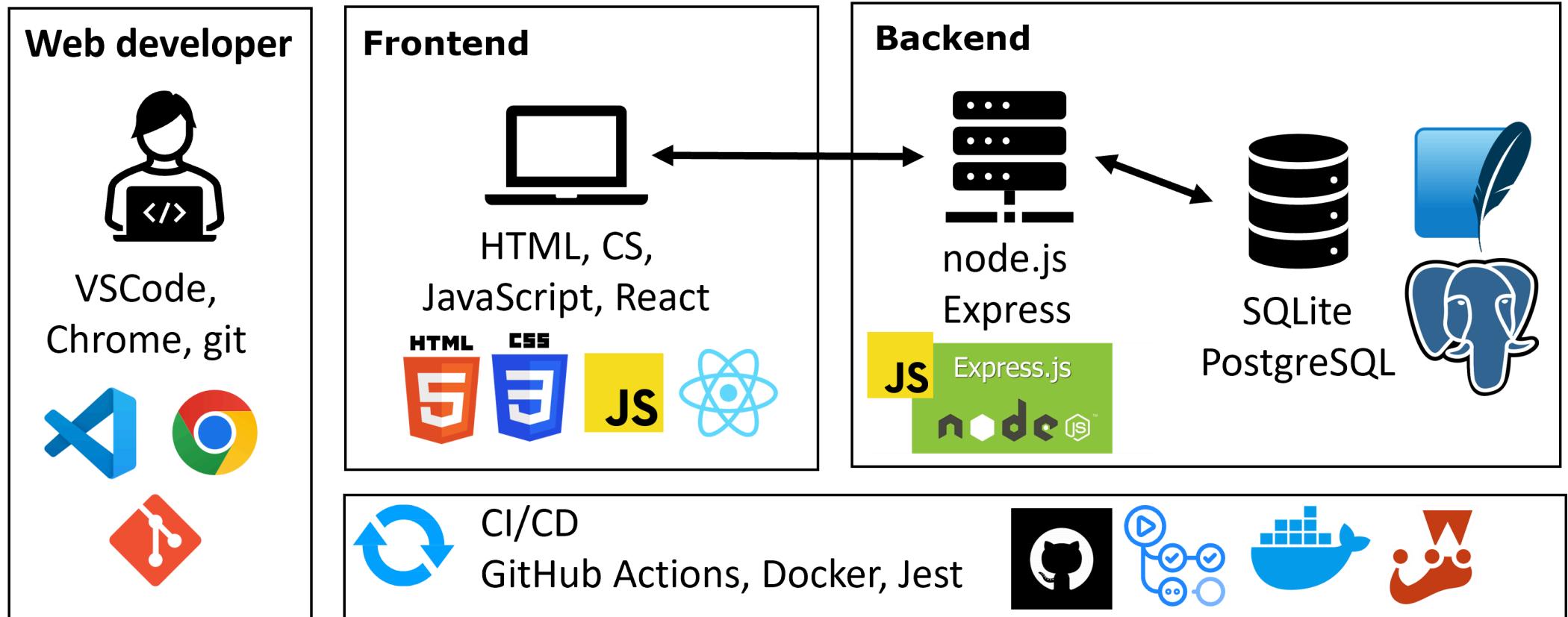
behavior
JavaScript

What's in a web page?

- **Hypertext Markup Language (HTML)**: semantic markup for web page content
- **Cascading Style Sheets (CSS)**: styling web pages
- **Client-side JavaScript**: adding programmable interactivity to web pages
- **Asynchronous Javascript, XML adn JSON**: fetching data from web services using **JavaScript fetch API**

What are we going to learn?

Modern Web Development Technology Stack



Types of web developers: **Frontend**, **backend** and **fullstack**

Acknowledgements

- Some contents of this lecture are partially adapted from:
 - Harvard [CS50's Web Programming with Python and JavaScript](#), licensed under [CC BY-NC-SA 4.0](#).
 - Materials from University of Washington's [CSE 154 Web Programming](#) (used with permission).
 - [The Odin Project](#) (main website code under MIT license and curriculum licensed under a [CC BY-NC-SA 4.0](#))
 - [The Fundamentals of Web Application Development \(Web Edition\)](#) ©2025 Nicholas D. Freeman. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.

► Back to title slide

► Back to lecture slides index

Web Programming

Lecture 1

HTML

Josue Obregon 

jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

[▶ Lecture slides index](#)

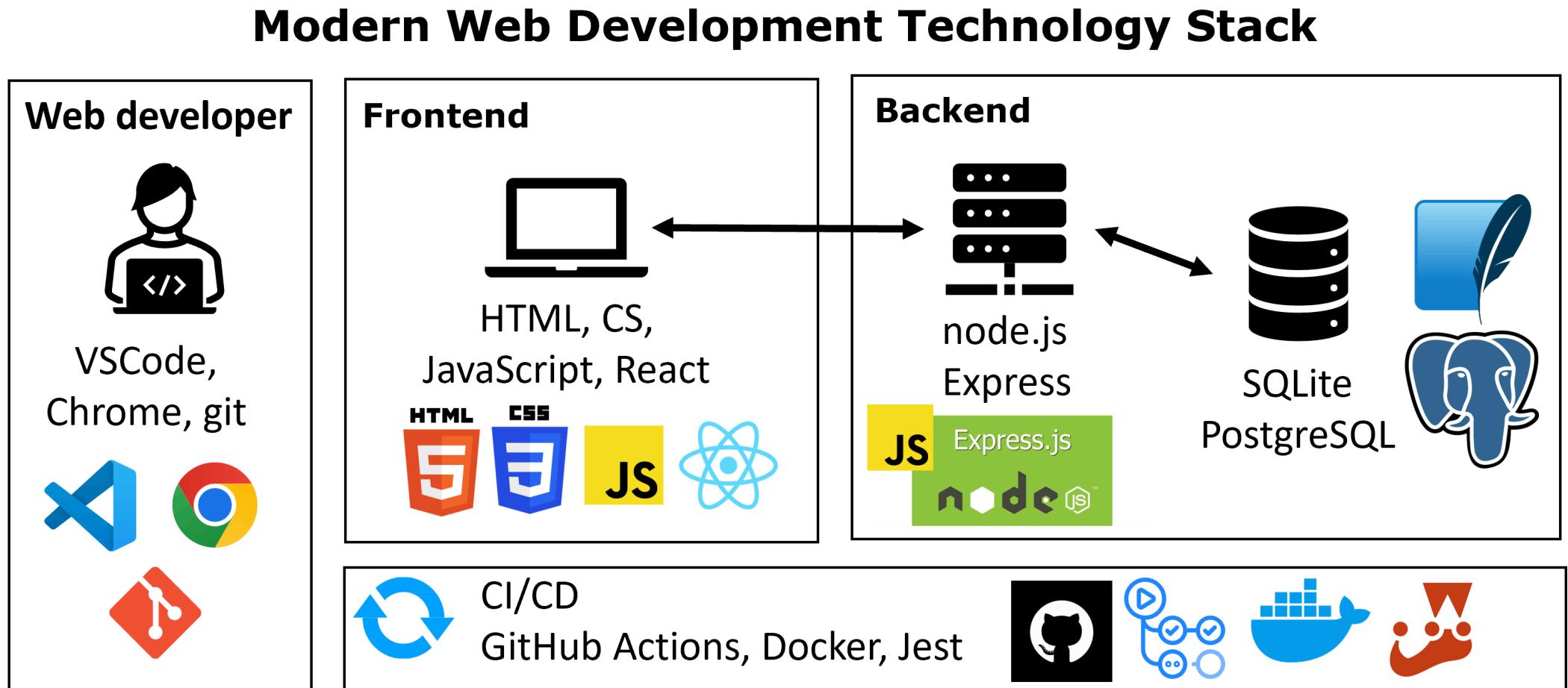
March 4, 2025



Agenda

- HTML
 - What is HTML
 - Semantic tags
 - Common HTML tags

What are we going to learn?

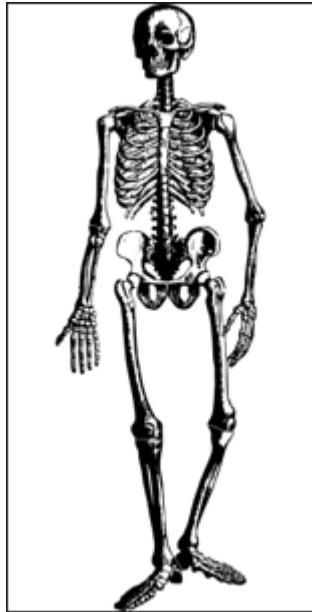


Roadmaps: **Frontend**, **backend** and **fullstack**

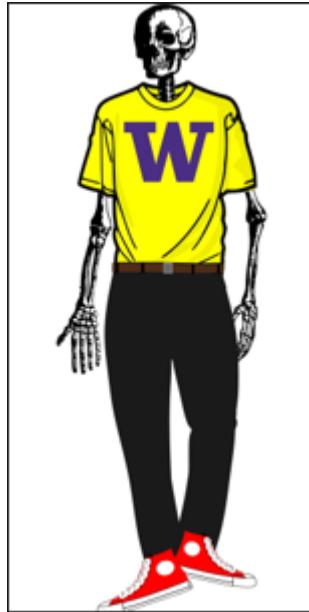
Analogy of a web page



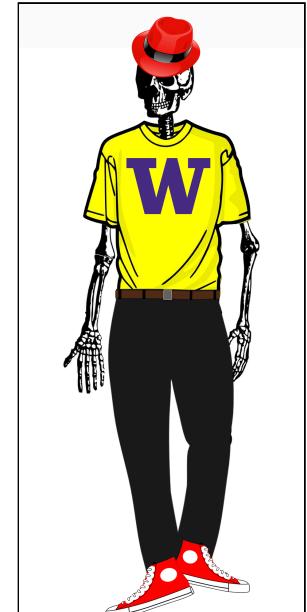
Words + images



HTML



CSS



JavaScript

HTML

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <title>Web Programming ITM</title>
5   </head>
6   <body>
7     Hello, world!
8   </body>
9 </html>
```

how
arrange/display
content

Hyper: over/above/beyond

Text: words and/or alphanumeric characters

Markup: the result of preparing text or indicating the relationship between parts of text before displaying

Language: a system of symbols used to communicate ideas

In other words... HTML defines the meaning and structure of web content(i.e., text, images, etc.)

programming language → make computer do something

Hypertext Markup Language

- Describes the content and structure of information on a web page
- Not the same as the presentation (appearance on screen)
- Surrounds text content with opening and closing tags
- Most whitespace is insignificant in HTML (ignored or collapsed to a single space)
- We will use a newer version called HTML5

SYNTAX:

```
1 <tag>content</tag>
```

opening

closing

EXAMPLE:

```
1 <p>This is a paragraph</p>
```

Structure of an HTML page

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <title>Web Programming ITM</title>
5   </head>
6   <body>
7     Welcome to Spring Semester!
8   </body>
9 </html>
```

Main HTML tags

- `<!DOCTYPE html>` tells the browser to interpret our page's code as HTML5
- `<html>` Represents the root (top-level element) of an HTML document
- `<head>` Contains machine-readable information (metadata) about the document (not rendered)
- `<body>` contains the page's content

General Outline with HTML5 <body>

```
1 <body>
2   <header>
3     <!-- Header of the webpage body (e.g. logo, navigation bar) -->
4   </header>
5   <main>
6     <!-- Main section of the webpage body (where most content is) -->
7   </main>
8   <footer>
9     <!-- Footer of the webpage body (e.g. copyright info) -->
10  </footer>
11 </body>
```

HTML5 Semantic Tags

A semantic element clearly describes its meaning to both the browser and the developer

- `<main>`
 - Main content of the document
 - there can only be one main element in the `<body>`
 - The content inside should be unique and not contain content that is repeated across pages (e.g. sidebars, nav link, search bars, etc.)
- `<header>`
 - contains header information for page body or section/article, including logos, navigation bar, etc.
- `<footer>`
 - contains footer information for page body or section/article, including copyright information, contact, links, etc.

HTML5 example semantic structure



Common HTML tags

- <h1>, <h2>, ..., <h6>
- ,
-
- <a>
- <table>
- <form>

HTML heading levels <h1>...<h6>

Description

Example code

Output

- Represent heading information for portions and/or subportions of the page
- <h1>: highest importance, <h2>: second highest importance etc.
- Do not choose based off of the rendered size
 - Remember, what is the purpose of HTML? -Do not skip heading levels



Do not confuse these tags

<head> vs. <header> vs. <h1>...<h6>

Ordered `` and unordered `` lists

Description	Example	Output
<ul style="list-style-type: none">• <code></code> for an unordered list (typically a bulleted list)• <code></code> for an ordered list (typically a numbered list)• Both represent a list of items• <code></code> tags are used within both unordered AND ordered lists• <code></code> list item tag. Representative of an item in a list• A list can contain another list		

- `` for an unordered list (typically a bulleted list)
- `` for an ordered list (typically a numbered list)
- Both represent a list of items
- `` tags are used within both unordered AND ordered lists
- `` list item tag. Representative of an item in a list
- A list can contain another list

Images

Inserts a graphical image onto the page (inline) - The `src` attribute specifies the image URL - The `alt` attribute describing the image, which improves accessibility for users who can't otherwise see it.

Example

```
1 
```



 The shield of Link

Link (Anchors) <a>

Links, or “anchors”, to other pages (inline)¹

Uses the href (Hypertext REference) attribute to specify the destination URL Can be absolute (to another web site) or relative (to another page on this site)

- Absolute example: “<https://www.google.com/>”
- Relative example: “/img/figure.jpg”

Example

```
1 Search for it on <a href="https://www.google.com/">Google</a>!
```

Output

Search for it on [Google](https://www.google.com/)!

Relative vs absolute paths for links and images

Relative: paths are relative to the document linking to the path. - Linked files within the same directory: “filename.png”

```
1 <img href="my-other-page.html">Check out my other page!</a>
```

- Linked files within a subdirectory (e.g. “img”): “img/filename.jpg”

```
1 
```

Absolute: paths refer to a specific location of a file, including the domain and protocol.

- Typically used when pointing to a link that is published online (not within your own website).
- Example: "<https://validator.w3.org/>"

Tables <table>

Description

Example

Output

Allows web developers to arrange data into rows and columns.

- <td> defines a table cell (table data)
- <tr> defines a table row
- <th> defines a table header (optional)
- <thead> groups the header content in a table
- <tbody> groups the body content in a table

Forms <form>

Description

Example

Output

It is used to create an HTML form for user input.

- <input type="text"> displays a single-line text input field
- <input type="submit"> Displays a submit button (for submitting the form)
- <input type="button"> Displays a clickable button

<input name=" " send to backend
identify the data >

Next week

Give style to our HTML document with CSS

CSS



Acknowledgements

- Some contents of this lecture are partially adapted from:
 - Harvard [CS50's Web Programming with Python and JavaScript](#), licensed under [CC BY-NC-SA 4.0](#).
 - Materials from University of Washington's [CSE 154 Web Programming](#) (used with permission).
 - [The Odin Project](#) (main website code under MIT license and curriculum licensed under a [CC BY-NC-SA 4.0](#))
 - [The Fundamentals of Web Application Development \(Web Edition\)](#) ©2025 Nicholas D. Freeman. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.

► Back to title slide

► Back to lecture slides index

Web Programming

Lecture 2

CSS

Josue Obregon 

jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

[▶ Lecture slides index](#)

March 10, 2025

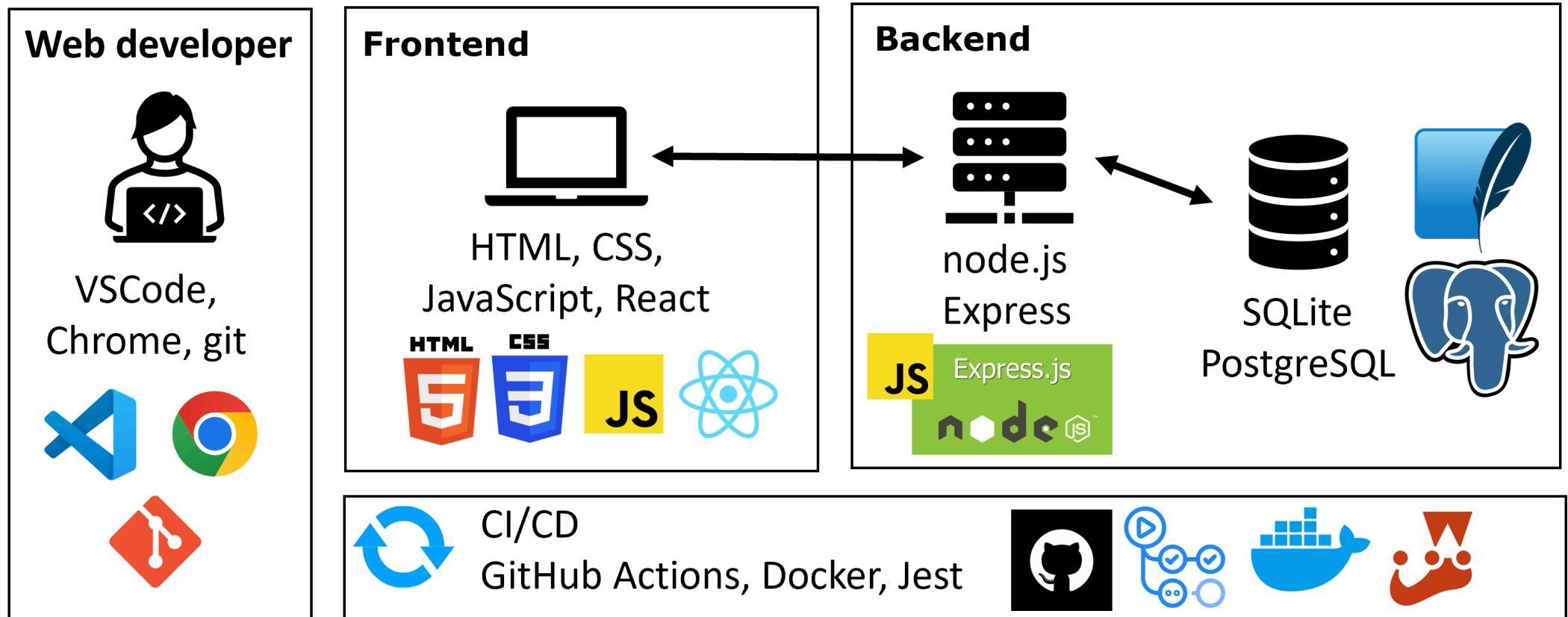


Agenda

- HTML
 - CSS
 - Specificity
 - Responsive web design
 - Bootstrap

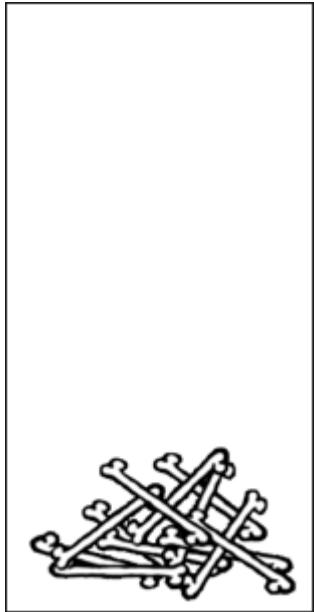
What are we going to learn?

Modern Web Development Technology Stack

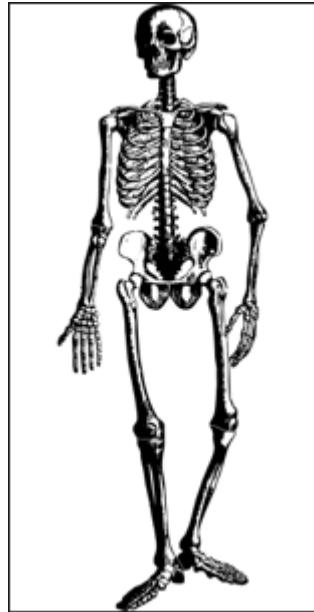


Roadmaps: **Frontend**, **backend** and **fullstack**

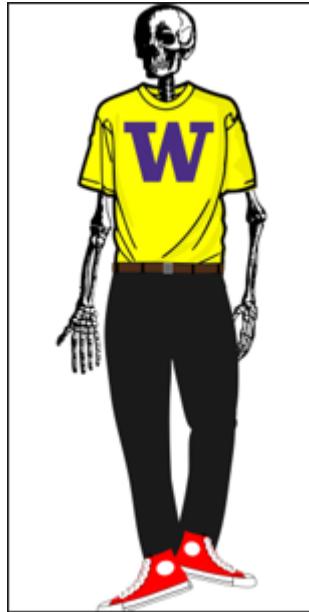
Analogy of a web page



Words + images



HTML



CSS



JavaScript

Cascading Style Sheets

```
1 <head>
2 ...
3   <link href="filename" rel="stylesheet">
4 ...
5 </head>
```

- CSS describes the appearance and layout of information on a web page (as opposed to HTML, which describes the content structure)
- Can be **embedded in HTML (bad)** or **placed into separate .css file (much better)**

Best Practice

To keep your code clean and easier to maintain, separate your styling into its own file.

Basic CSS Rule Syntax

Template

```
1 selector {  
2 <property: value; > declaration  
3   property: value;  
4   ...  
5   property: value;  
6 }
```

Example

```
1 h1 {  
2   color: red;  
3   text-decoration: underline;  
4   font-family: sans-serif;  
5 }
```

- A CSS file consists of one or more rulesets containing one or more rules
- **Selectors** designate exactly which element(s) to apply styles to
- **Properties** determine the styles that will be applied to the selected element(s)
- Each property has a set of **values** that can be chosen
- There are currently over **200 possible style properties** to choose from - VSCode is useful for autocompleting property values!
- A rule is considered a **selector** and the list of **properties** applied to it.

style attribute and tag embedded in HTML

Inline

```
1 <h1 style="color:red;">  
2     This text will be red!  
3 </h1>
```

In the <head>

```
1 <head>  
2     <style>  
3         h1 {  
4             color: red;  
5         }  
6     </style>  
7 </head>
```

- In general, it is better to put your styles in external stylesheets and apply them using <link> elements.

Common CSS properties

CSS Property	Example Values
color	red, blue, #000000 (black)
text-align	left, right, center, justify
width, height	auto, 100px, 50%, 100%
margin, padding	0, 10px, 20px, 1em
font-family	Arial, Times New Roman, sans-serif, serif
font-size	12px, 16px, 1.5em,
font-weight	normal, bold, 400, 700
border	none, 1px solid black, 2px dashed red, 3px double blue

Grouping selectors

Description	Example HTML	Output with style applied
<pre>1 a, p, h3 { 2 color: green; 3 font-size: 14pt; 4 } 5 6 p { 7 text-decoration: underline; 8 }</pre>		

Allows us to reduce redundancy by grouping together rules into rulesets to style multiples types of elements with common styles. What will text inside the `<p>` tag look like here?

id and **class**

id

- Unique identifier for an element
- Each unique id can only appear once per page
- Each element can only have one id

class

- Non-unique grouping attribute to share with many elements
- Many elements (even of different types) can share the same class
- Each element can have many different classes

Cascade and specificity

All styles “**cascade**” from the top of the sheet to the bottom

Source Order: If multiple CSS rules have the same specificity, the one that appears later in the stylesheet overrides earlier ones.

Specificity: More specific selectors (like IDs) override less specific ones (classes, tags).

1. inline
2. id
3. class
4. type

Importance (`:important`): Styles marked with !important override specificity and source order, becoming top priority.

RIORITY

property : value !important ;

The Document Object Model (DOM)

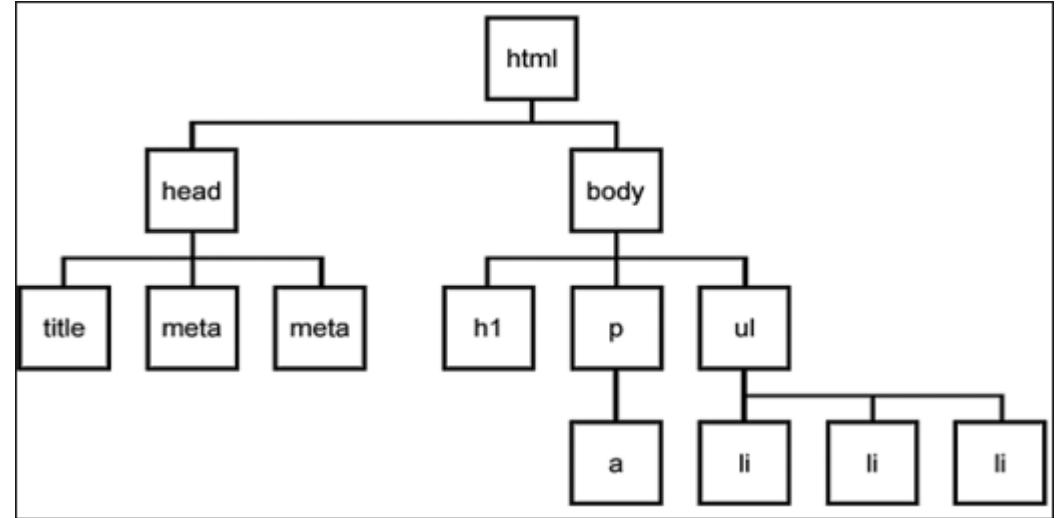
How the browser represents the hierarchy of a page. Very useful when thinking about selectors!

For example:

- Every HTML tag (`<p>`, ``, `<body>`, etc.) is part of the DOM, and has a place in a tree hierarchy.
- They all have a parent (`<body>`'s is `<html>`) and almost always have a child (`<title>` is a child of `<head>`).
- You don't build a DOM, but the browser does, which allows you to access it.

Example DOM

```
1 <html>
2   <head>
3   ...
4   <head>
5   <body>
6     <h1>heading</h1>
7     <p>
8       hello <a href="">there</a>
9     </p>
10    <ul>
11      <li>one</li>
12      <li>two</li>
13      <li>three</li>
14    </ul>
15  </body>
16 </html>
```



CSS Selectors

Syntax	Name	Description
a, b	Multiple Element Selector	
a b	Descendant Selector	Selects all b elements located anywhere inside of a
a > b	Child Selector	Selects all b descendants directly inside a elements
a + b	Adjacent Sibling Selector	Selects b only if it shares a parent with a and is immediately after it in the DOM
[a=b]	Attribute Selector	
a:b	Pseudoclass Selector	
a::b	Pseudoelement Selector	

Responsive web design

Adjusts a website's layout to fit the screen size of the device it's being viewed on.

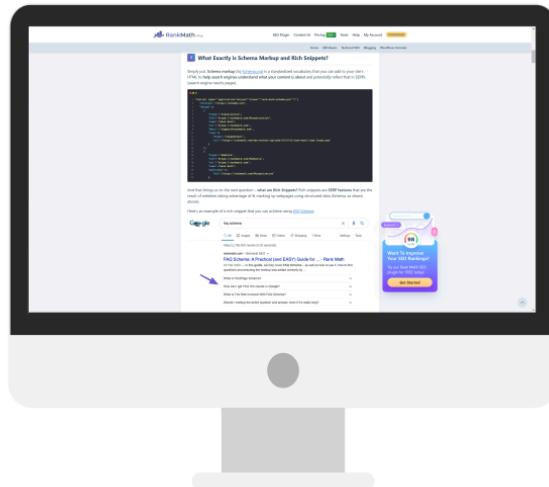
It allows a single website to work across many devices, including desktops, tablets, and phones.

Responsive design related concepts:

- Viewport
- Media queries
- Layout techniques in CSS

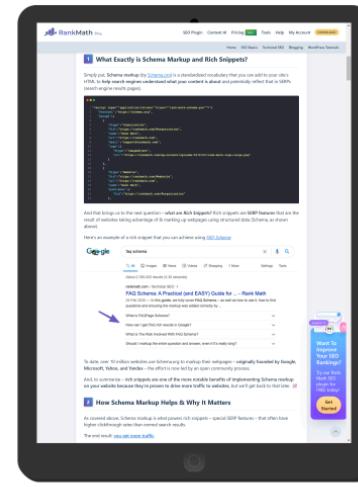
Viewport

Desktop



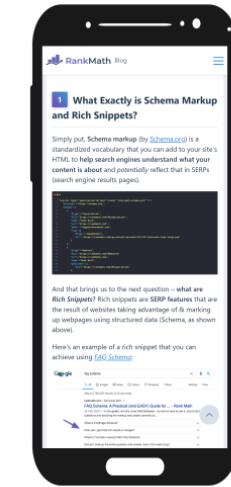
1920 x 1080 px

Tablet



1024 x 1388 px

Smartphone



430 x 932 px

- The viewport is the user's visible area of a web page.
- The viewport varies with the device, and will be smaller on a mobile phone than on a computer screen.

Media queries

Include this line in the `<head>` tag of your HTML document.

```
1 <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

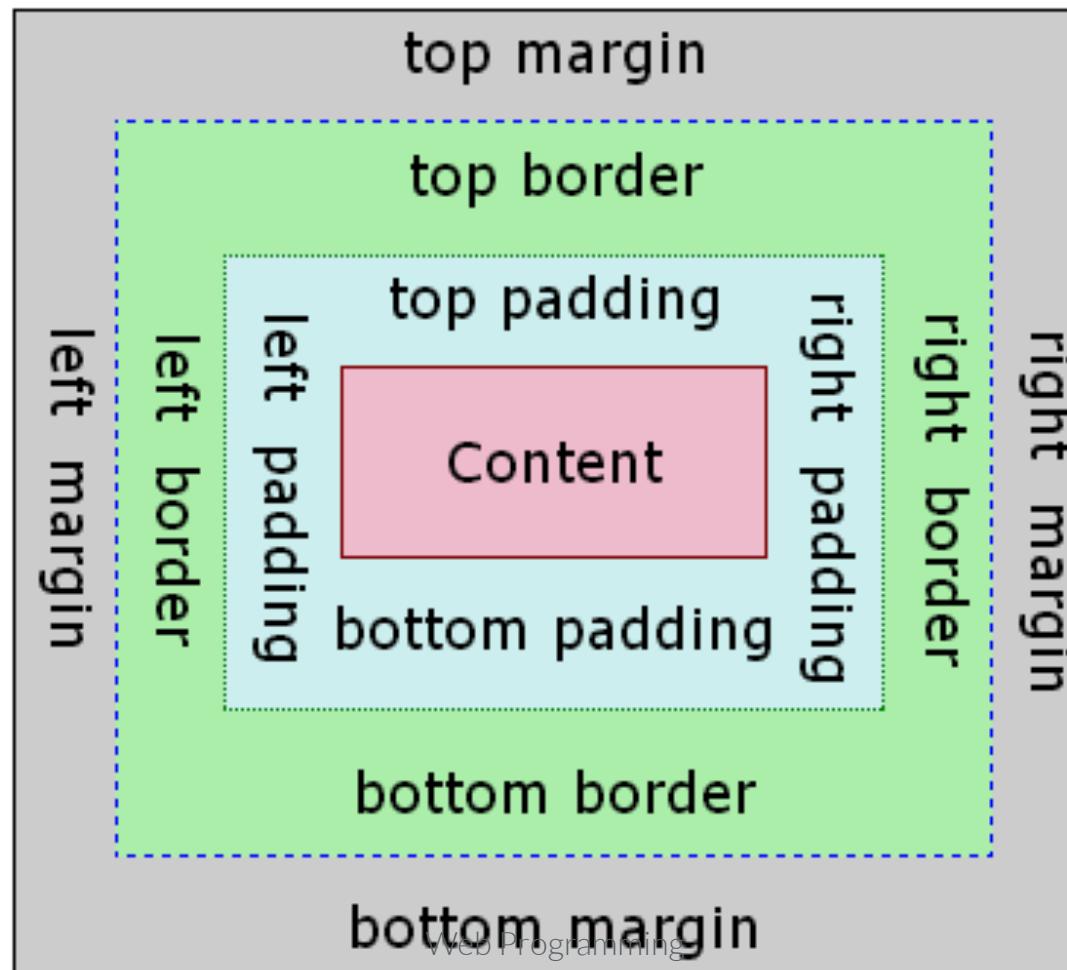
- `<meta name=viewport>` tag is essential for controlling how a webpage fits into different screen sizes
- `width=device-width` match the width of the viewport to the width of the device
- `initial-scale=1.0` defines the zoom of the page the moment it loads
- Media types:
 - `print`, `screen`, `hover`, etc.
- Media features:
 - `height`, `width`, `orientation`, etc.

Layout techniques in CSS

- Box Model (margin/padding/border)
- Flex
- Grid
- Others (we are not going to study them)
 - Positioning
 - Float (less common today)

CSS box model

The **CSS box model** module defines the `margin` and `padding` properties, which along with the `height`, `width` and `border` properties, make up the CSS box model.



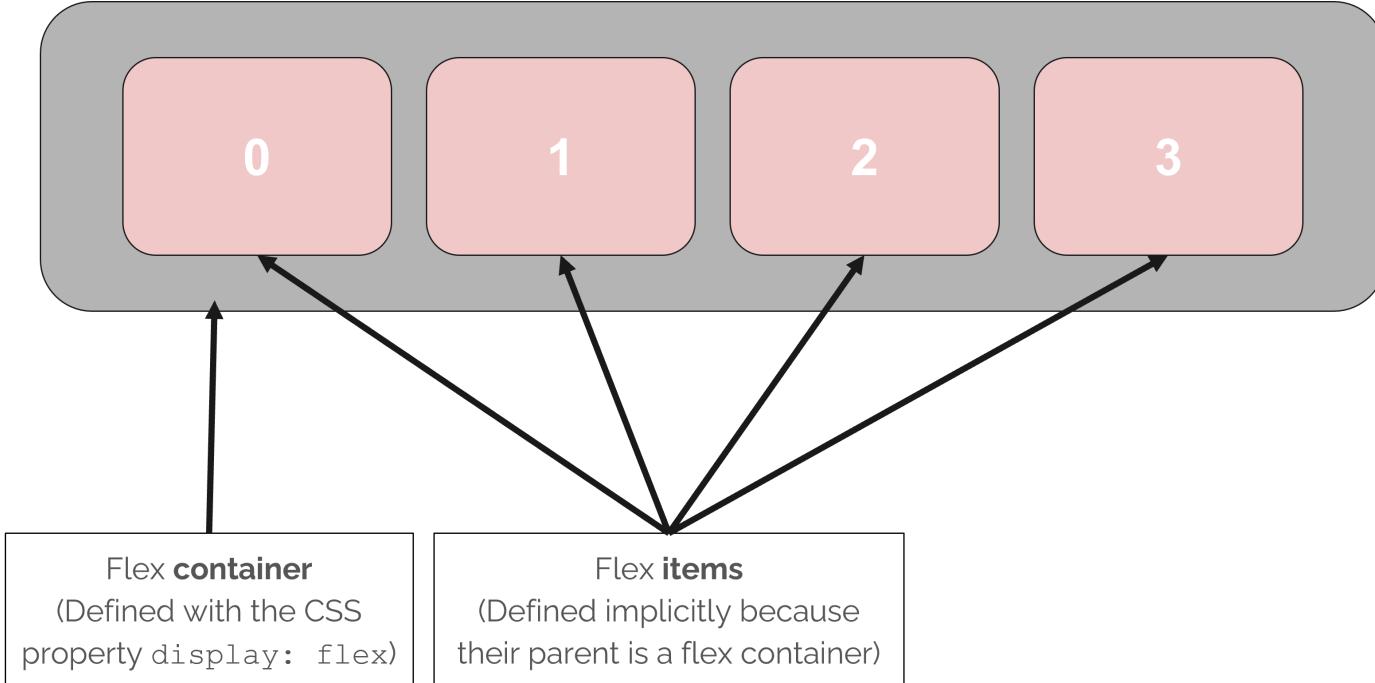
Flexbox

- Flexbox is a set of CSS properties for aligning block level content.
- Flexbox defines two types of content: “containers” and “items”.
 - “Container” (sometimes referred to as the flex parent): the direct parent element of the HTML elements whose position you would like to control
 - “Items” (sometimes referred to as the flex children): the directly nested elements inside the parent container whose position you are controlling.
- Most properties on the container can be set to determine how its items are laid out. Some properties can be directly applied to the flex items.

To label the container as a flex container we use the CSS set

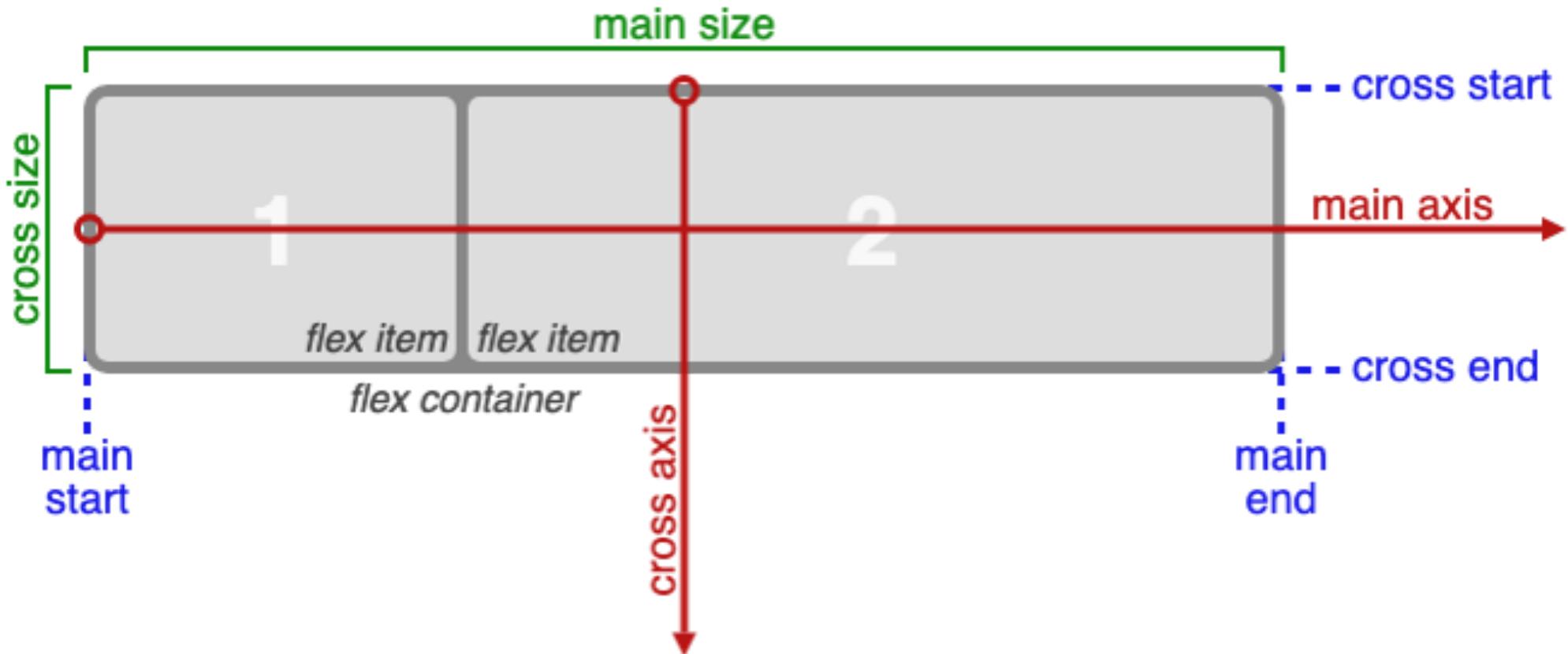
```
1 div {  
2   display: flex;  
3 }
```

Flexbox



```
1 <section>
2   <p>0</p>
3   <p>1</p>
4   <p>2</p>
5   <p>3</p>
6 </section>
```

The aftermath of `display:flex`



```
1 <section>
2   <p>1</p>
3   <p>2</p>
4 </section>
```

Basic properties of a flex container

`display: flex;`

-makes an element a “flex container”, items inside automatically become “items” - by default, starts as a row

`justify-content: flex-end; (flex-start, space-around,...)`

- indicates how to space the items inside the container along the main axis

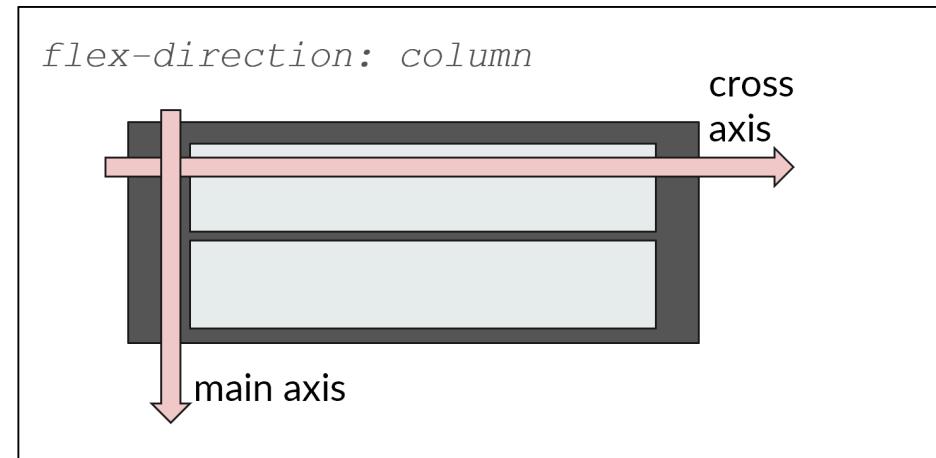
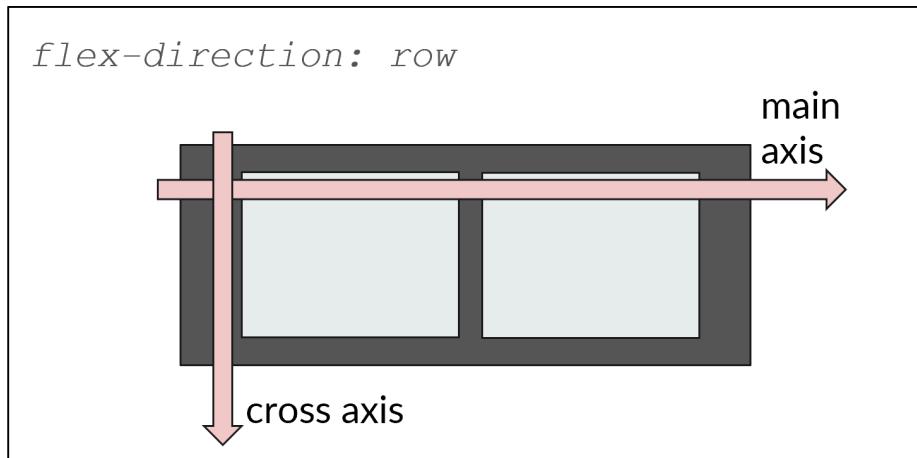
`align-items: flex-end; (flex-start, center, baseline,...)` - indicates how to space the items inside the container along the cross axis

`flex-direction: row; (column)` - indicates whether the container flows horizontally or vertically (default row)

`flex-wrap: wrap; (no-wrap, ...)` - indicates whether the container’s children should wrap on new lines

The Impact of Adjusting `flex-direction`

- The default `flex-direction` is **row**. If you change the `flex-direction` to **column** you are changing the orientation of the axes.
- The properties arranging elements along each axis does not change
 - Example: `justify-content` still controls items along the main axis. The main axis is just vertical now



CSS grid layout

It divides the page into major regions and defines the relationship in terms of size, position, and layering between parts of a control built from HTML primitives.

To label the container as a grid container we use the CSS set

```
1 div {  
2   display: grid;  
3 }
```

- **grid-column-gap** and **grid-row-gap** specifies the gap between the columns and rows
- **grid-template-columns** and **grid-template-rows** specifies the size of the rows in a grid layout

Bootstrap

Description

Example with Bootstrap

Example without Bootstrap

Bootstrap is one of the most popular HTML, CSS, and JavaScript framework for developing responsive, mobile-first websites.

Bootstrap is opensource under the MIT license .

- Include this code in your `<head>` element of your HTML document

```
1 <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.
```



Next week

Say hello to our new friend, JavaScript...



Acknowledgements

- Some contents of this lecture are partially adapted from:
 - Harvard [CS50's Web Programming with Python and JavaScript](#), licensed under [CC BY-NC-SA 4.0](#).
 - Materials from University of Washington's [CSE 154 Web Programming](#) (used with permission).
 - [The Odin Project](#) (main website code under MIT license and curriculum licensed under a [CC BY-NC-SA 4.0](#))
 - [The Fundamentals of Web Application Development \(Web Edition\)](#) ©2025 Nicholas D. Freeman. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.

► Back to title slide

► Back to lecture slides index

Web Programming

Lecture 3

Introduction to JavaScript

Josue Obregon 

jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

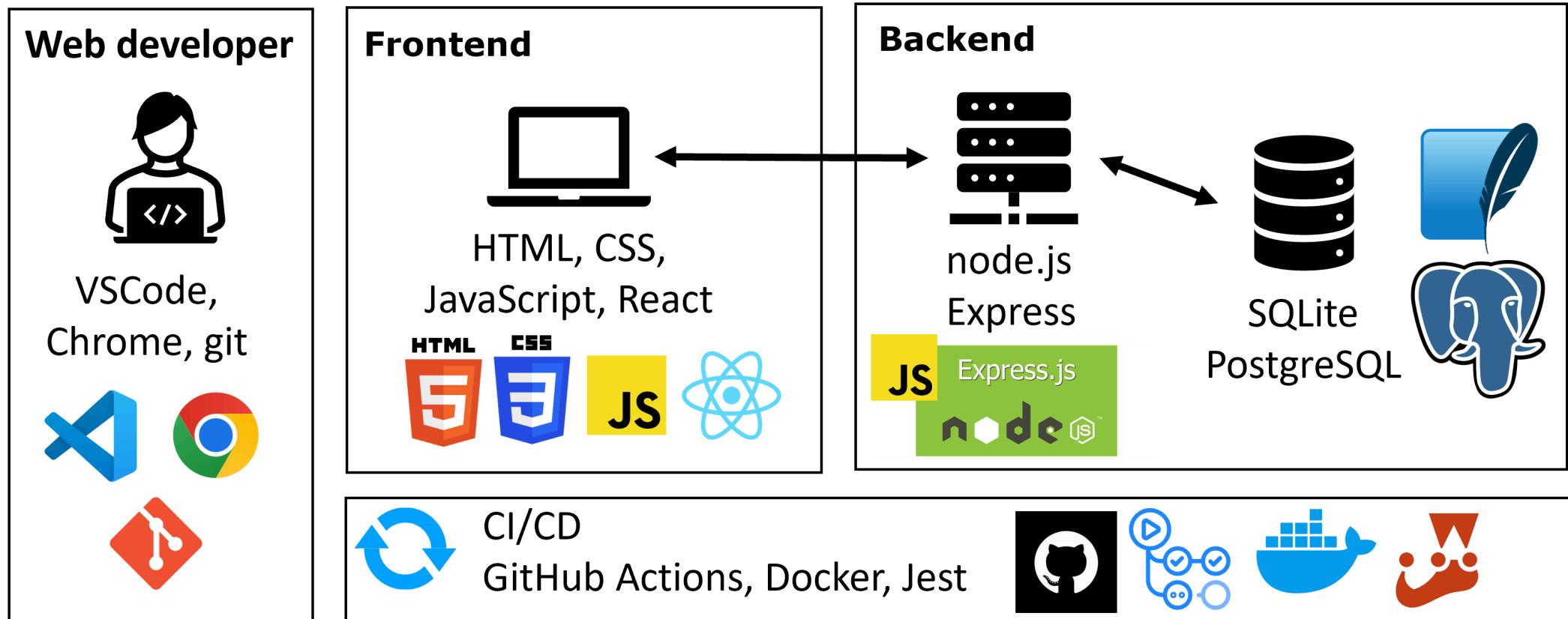
 [Lecture slides index](#)

March 14, 2025



Course structure

Modern Web Development Technology Stack



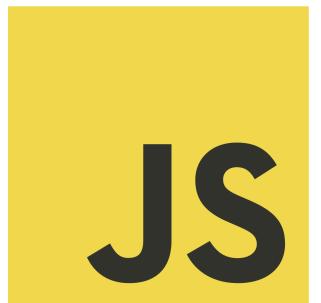
Roadmaps: **Frontend**, **backend** and **fullstack**

Agenda

- JavaScript
 - Variables and data types
 - Arrays
 - Control and repetition statements
 - Functions
 - Objects

JavaScript

- A lightweight “scripting” programming language
- Created in 1995 by Brendan Eich
 - Originally called Mocha, but it was renamed LiveScript and finally JavaScript
- First version was called ECMAScript¹ 1 (ES1)
- It releases versions every year (evolves fast)
- Current version is ECMAScript 2024 (ES2024)
- NOT related to Java other than by name and some syntactic similarities

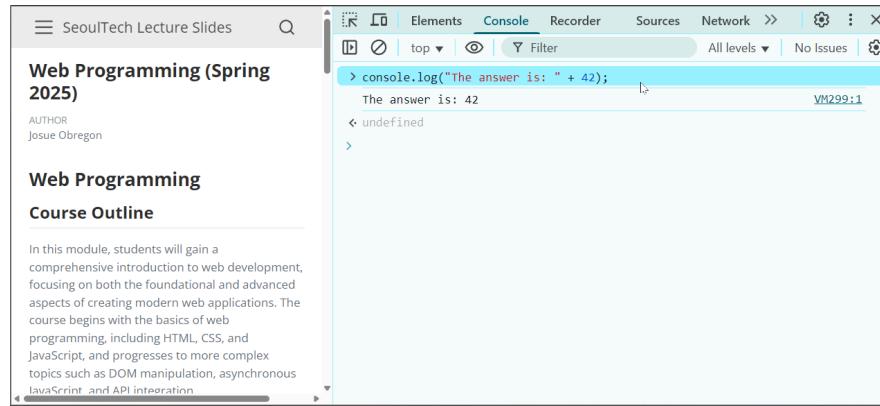


Why are we learning JavaScript?

- We can use JavaScript in **frontend development** to:
 - Insert dynamic text into HTML (e.g., username)
 - React to events (e.g., page load, user's mouse click)
 - Retrieve information about a user's computer (e.g., what browser they are using)
 - Request additional data needed for the page (e.g., from an API; more on this in a couple weeks)
- Thanks to **Node.js** (a runtime environment for running JavaScript programs outside the browser) we can also use JavaScript in **backend development!**

JavaScript, an interpreted language

- As an interpreted programming language, JS is great for interacting with a line at a time (similar to Python, but very different than Java). Where do we start?
- The easiest way to dive in is with the Chrome browser's [Console tab](#) in the same inspector tool we've used to inspect our HTML/CSS (press F12).

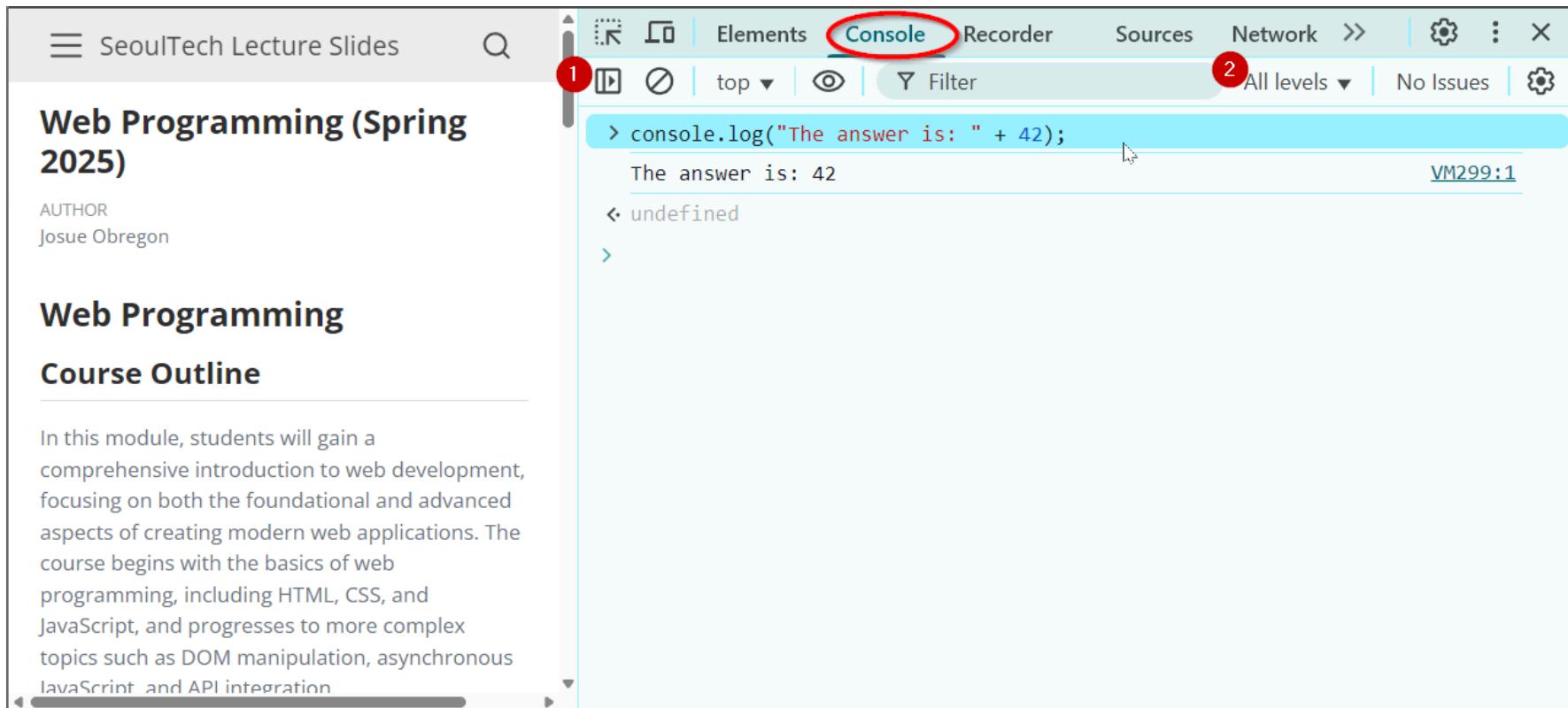


- Until we learn how to interact with the HTML DOM with JS, it is recommended experimenting with the following code examples using this console to get comfortable with the basic syntax and behavior.

Console output in JavaScript

- `console.log()` command is used to output values to the browser console, most often used to debug JS programs.
- You can think of this as `System.out.println` in Java or `print` in Python.
- Make sure you “Hide console sidebar” (1) and select “All levels” in the dropdown menu (2)

```
1 console.log("The answer is: " + 42);
```



Comments

```
1 // single-line comment  
2  
3 /**  
4 * multi-line  
5 * comment  
6 */  
7
```

- Identical to Java's comment syntax



Comments in HTML and CSS

- HTML: `<!-- comment -->`
- CSS: `/* comment */`

Variables and constants

```
1 var name = expression; // define a variable globally  
2 let name = expression; // define a variable with limited scope  
3 const name = expression; // define a constant (cannot change)
```

- Variables are declared with the `let` keyword (case-sensitive).
- Constants are declared with the `const` keyword
- You may also see `var` used instead of `let`
 - this is an older convention with weaker scope - it is not recommended to use
- Naming convention:
 - Use `camelCase` for variable (and function) names
 - Don't start with a number or symbol (except `_` or `$`)
 - Variable names are case-sensitive (i.e. `age` is not the same as `Age`)
 - Do not use JS keywords as names (e.g., `let`, `const`, `function`, or `return`)

Variables and constants

```
1 let level = 23;           // number
2 let accuracyRate = 0.99;   // number
3 let name = "ITM WebProg"; // string
4 let flag = true;          // boolean
5 let temps = [55, 60, 57.5]; // array
6 typeof(name);            // 'string'
```

- Types are not specified, but JS does have types (“loosely-typed”)

- Primitives: number / string / boolean / array / null / undefined
 - object, function,
 - The typeof function returns a variable type
- (key, value)
only mutable!*
- collections of*
- declared but no value yet*
- get to nothing*

Note

Type conversion isn't always what you expect as you will see in a moment.

A Note about declaring types in JavaScript

- If you've programmed in a statically-typed language like Java, you may recall that when declaring variables, you must specify their type which stays the same during runtime.

```
1 boolean isValid = "hello!"; // error in JavaScript. boolean keyword doesn't
```

- In a dynamically-typed language like JavaScript, you don't need to specify the type (just use let or const) and you may change the type the variable refers to later in execution.

```
1 let isValid = true; // no error
2 isValid = "hello!";
3 isValid = 1;
4 isValid = true;
```

Number type

```
1 let enrollment = 99;  
2 let medianGrade = 2.8;  
3 let credits = 5 + 4 + (2 * 3);
```

- Integers and real numbers are the same type (no `int` vs. `double`). All numbers in JS are floating point numbers.
- Same operators: `+` `-` `*` `/` `%` `++` `--` `=` `+=` `-=` `*=` `/=` `%=` and similar precedence to Java.
- Many operators auto-convert types: `"2" * 3 is 6`
- `NaN` (“Not a Number”) is a return value from operations that have an undefined numerical result (e.g. dividing a String by a Number).

String type

```
1 let name = "Yuna Kim";                                // "Yuna Kim"
2 let fName = name.substring(0, name.indexOf(" ")); // "Yuna"
3 let len = nickName.length;                            // 8
4 let sport = 'Figure Skating';                      // can use "" or '
5 let message = `${name}, practices ${sport}`;        // `` template literals
6 console.log(message) // 'Yuna Kim, practices Figure Skating'
```

- Methods:
 - `charAt`, `charCodeAt`, `fromCharCode`, `indexOf`, `lastIndexOf`, `replace`, `split`, `substring`, `toLowerCase`, `toUpperCase`
- Template literals
 - Literals delimited with backtick (`) characters
 - Can also contain other parts called **placeholders**, which are embedded expressions delimited by a dollar sign and curly braces: ``${expression}``.

More about strings

```
1 let count = 10;                                // 10
2 let stringedCount = "" + count;                // "10"
3 let puppyCount = count + " puppies, yay!";    // "10 puppies, yay!"
4 let magicNum = parseInt("42 is the answer");   // 42
5 let mystery = parseFloat("Am I a number?");    // NaN
```

- To convert between Numbers and Strings:
- Escape sequences behave as in Java: \ ' \" \& \n \t \\
- To access characters of a String s, use s[index] or s.charAt(index):

```
1 let firstLetter = puppyCount[0];                // "1"
2 let fourthLetter = puppyCount.charAt(3);          // "p"
3 let lastLetter = puppyCount.charAt(puppyCount.length - 1); // "!"
```

Common bugs when using strings

```
1 let sumNum = 1 + 1;          // "2"  
2 let sumStrNum = "1" + 1;    // "11"  
3 let sumNumStr = 1 + "1";   // "11"
```

- While Strings in JS are fairly similar to those you'd use in Java, there are a few special cases that you should be aware of.
 - Remember that `length` is a property (not a method, as it is in Java)
 - Concatenation with `+` (see the code above)

Special values: `null` and `undefined`

```
1 let foo = null;  
2 let bar = 9;  
3 let baz;  
4  
5 /* At this point in the code,  
6  * foo is null  
7  * bar is 9  
8  * baz is undefined  
9 */
```

- `undefined`: declared but has not yet been assigned a value
- `null`: exists, but was specifically assigned an empty value or null.
 - Expresses intentional a lack of identification.
- A good motivating overview of **null vs. undefined**

Note

This takes some time to get used to, and remember this slide if you get confused later.

Arrays *mutable*

```
1 let name = [];                                // empty array
2 let names = [value, value, ..., value]; // pre-filled
3 names[index] = value;                      // store element
4
5 let types = ["Electric", "Water", "Fire"];
6 let pokemon = [];      // []
7 pokemon[0] = "Pikachu"; // ["Pikachu"] 0-indexed
8 pokemon[1] = "Squirtle"; // ["Pikachu", "Squirtle"]
9 pokemon[3] = "Gengar";  // ["Pikachu", "Squirtle", undefined, "Gengar"]
10 pokemon[3] = "Abra";   // ["Pikachu", "Squirtle", undefined, "Abra"]
```

- Two ways to initialize an array
- **length** property (grows as needed when elements are added)
- You can also create an array from other data types
 - For example, when you split a string using the **string.split()** method

Arrays methods

```
1 array.indexOf(element, fromIndex) optional
2 array.includes(element, fromIndex)
3
4 let pokemon = ["Pikachu", "Squirtle", "Gengar", "Abra"];
5 let index = pokemon.indexOf("Gengar");
6 console.log(index); // 2
7
8 console.log(pokemon.includes("Squirtle")); // true
9 console.log(pokemon.includes("Jigglypuff")); // false
```

- `indexOf()` method is useful for finding the first index of a specific element within an array
 - If the element cannot be found, then it will return -1
- `includes()` method is a simple and efficient way to check if an array contains a specific value

Conditional statements

```
1 const condition = true
2 const condition2 = true
3 if(condition) {
4     console.log("The condition is true")
5 } else if (condition2) {
6     console.log("The condition2 is true")
7 } else {
8     console.log("The condition and condition2 are false")
9 }
```

- Identical structure to Java's if/else statement
- Logical operators: > < >= <= && || ! object, array, function, NaN
- Strict equality operators: === !== ⇒ cannot use for comparing non-primitive types
- Non-strict equality operators: == != ⇒ Don't check data types

More on logical operators

```
1 5 < "7"          // true implicit conversion "7" to 7  
2 42 == 42.0        // true  
3 "5.0" == 5        // true  
4 "5.0" === 5       // false string ≠ number
```

- Most logical operators automatically convert types
- In the case of equality operator attempt to convert types
- `====` and `!==` are strict equality tests; checks both type and value

Boolean type

```
1 let iLikeWebProg = true;
2 let myGrade = "A+" > 0; // false
3 if ("web programming is great") {
4   console.log("true");
5 }
6 if (0) { /* false */ }
7
8 //converting a value into a `boolean` explicitly:
9 let boolValue = Boolean(otherValue);
10 let boolValue = !(otherValue); //double not
```

- Any value can be used as a **boolean**
 - “**falsey**” values: 0, 0.0, NaN, "", null, and undefined
 - “**truthy**” values: anything else

Repetition statements

while

```
1 let i = 1;  
2 while (i <= 10) {  
3     console.log(i);  
4     i++;  
5 }
```

for

```
1 for (let i = 0; i < 10; i++) {  
2     console.log(i);  
3 }  
4  
5
```

- The most common repetition statements in JS are the `for` and `while` statement
- Syntax is quite similar to Java

Defining functions

```
1 // template
2 function name(params) {
3     statement;
4     return;
5 }
6
7 // defining a function
8 function greet(name) {
9     console.log(`Hello ${name}`);
10 }
11 //anonymous function
12 const sum = function (num1, num2) {
13     return num1 + num2;
14 };
15
16 // calling a function
17 greet('WebProg'); // Hello Webprog
18 console.log(sum(3, 4)); // 7
```

- Functions are declared using the **function** keyword.
- Statements placed into functions can be evaluated in response to user events
- An **anonymous function** is a function without a name that can be assigned to a variable

Arrow function expressions

```
1 () => expression
2
3 param => expression
4
5 (param) => expression
6
7 (param1, paramN) => expression
8
9 () => {
10   statements
11 }
12
13 param => {
14   statements
15 }
16
17 (param1, paramN) => {
18   statements
19 }
```

always anonymous

- Arrow function is a concise way of writing JS functions in shorter way.
- They were introduced in the ES6 version.
- They make our code more structured and readable (if you are familiar with the syntax)

More Array methods

```
1 const numbers = [1, 2, 3, 4, 5]
2 const mapTransformation = numbers.map(el => el * 10) //10,20,30,40,50
3 const forEachTransformation = []
4 numbers.forEach(el => {
5     forEachTransformation.push(el * 10)
6 })
7 console.log(mapTransformation) // 10,20,30,40,50
8 console.log(forEachTransformation) // 10,20,30,40,50
```

- The `map(function)` directly returns a new array with the applied transformation.
- The `forEach(function)` method calls a function for each element in an array. ⇒ No return new array

Using Objects

```
1 let person = {  
2   properties }   name: "Philip J. Fry",           // string  
3   age: 23,          // number  
4   "weight": 172.5,      // number  
5   friends: ["Farnsworth", "Hermes", "Zoidberg"], // array  
6 };  
7  
8 person.age;  
9 person["weight"]; // 172.5  
10 person.friends[2]; // "Zoidberg"  
11 let propertyName = "name";  
12 console.log(person[propertyName]); // "Philip J. Fry"
```

- Objects are the most versatile structure in JS
- You can access the fields with dot notation (`.fieldName`) or bracket notation (`["fieldName"]`) syntax
- You can create a new property or overwrite existing ones in an object by assigning a value

more flexible
→ access property dynamically

Optional chaining ? .

```
1 const user = {  
2   name: "John",  
3   address: {  
4     street: "Main Street",  
5   },  
6 };  
7 const otherUser = {  
8   name: "Jane",  
9 };  
10 console.log(user.address?.street); // Main Street  
11 console.log(otherUser.address?.street); // undefined  
12 // without optional chaining:  
13 console.log(user.address.street); // Main Street  
14 console.log(otherUser.address.street); // TypeError: Cannot read properties of undefined (reading 'street')
```

- New operator introduced in ES2020
- It allows you to access deeply nested properties of an object without worrying about whether the property exists or not

Next week

DOM manipulation and event-driven programming with JavaScript



Acknowledgements

- Some contents of this lecture are partially adapted from:
 - Harvard [CS50's Web Programming with Python and JavaScript](#), licensed under [CC BY-NC-SA 4.0](#).
 - Materials from University of Washington's [CSE 154 Web Programming](#) (used with permission).
 - [The Odin Project](#) (main website code under MIT license and curriculum licensed under a [CC BY-NC-SA 4.0](#))
 - [The Fundamentals of Web Application Development \(Web Edition\)](#) ©2025 Nicholas D. Freeman. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [freeCodeCamp.org](#) © 2025 freeCodeCamp.org. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.

Web Programming

Lecture 4

DOM manipulation with JavaScript

Josue Obregon 

jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

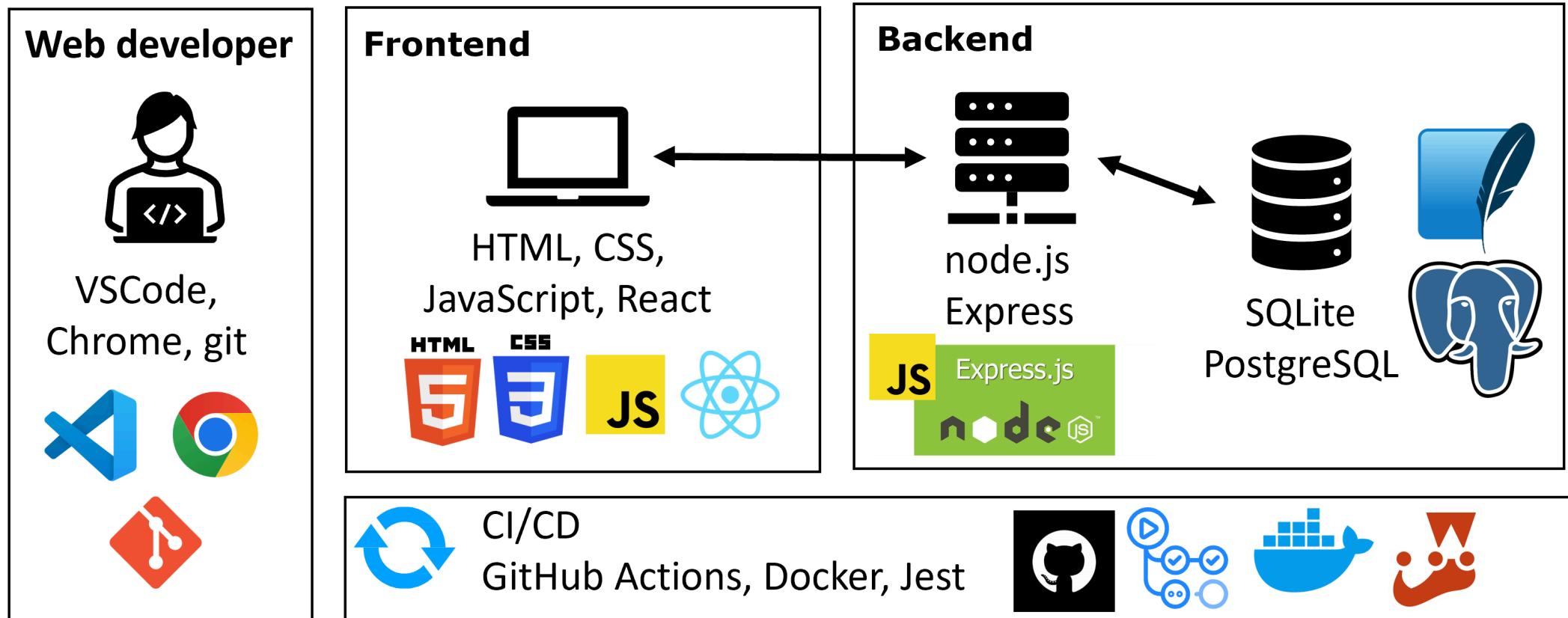
 [Lecture slides index](#)

March 20, 2025



Course structure

Modern Web Development Technology Stack



Roadmaps: **Frontend**, **backend** and **fullstack**

Agenda

- JavaScript
 - Event-driven programming
 - DOM manipulation

Separation of concerns

- Separation of Concerns: a concept from Software Engineering that every part of a program should address a separate “concern”.
- In web programming, we define those concerns as:
 - Content (words, images)
 - Structure/Meaning (HTML)
 - Style/Appearance (CSS)
 - Behavior *⇒ by using JS*
- What happens if we don't separate them?
 - redundant codes*
 - inefficient modifying*
 - lose maintainability*
- How do we improve this? *avoid combining diff concerns in the same file*

JavaScript/HTML Connection

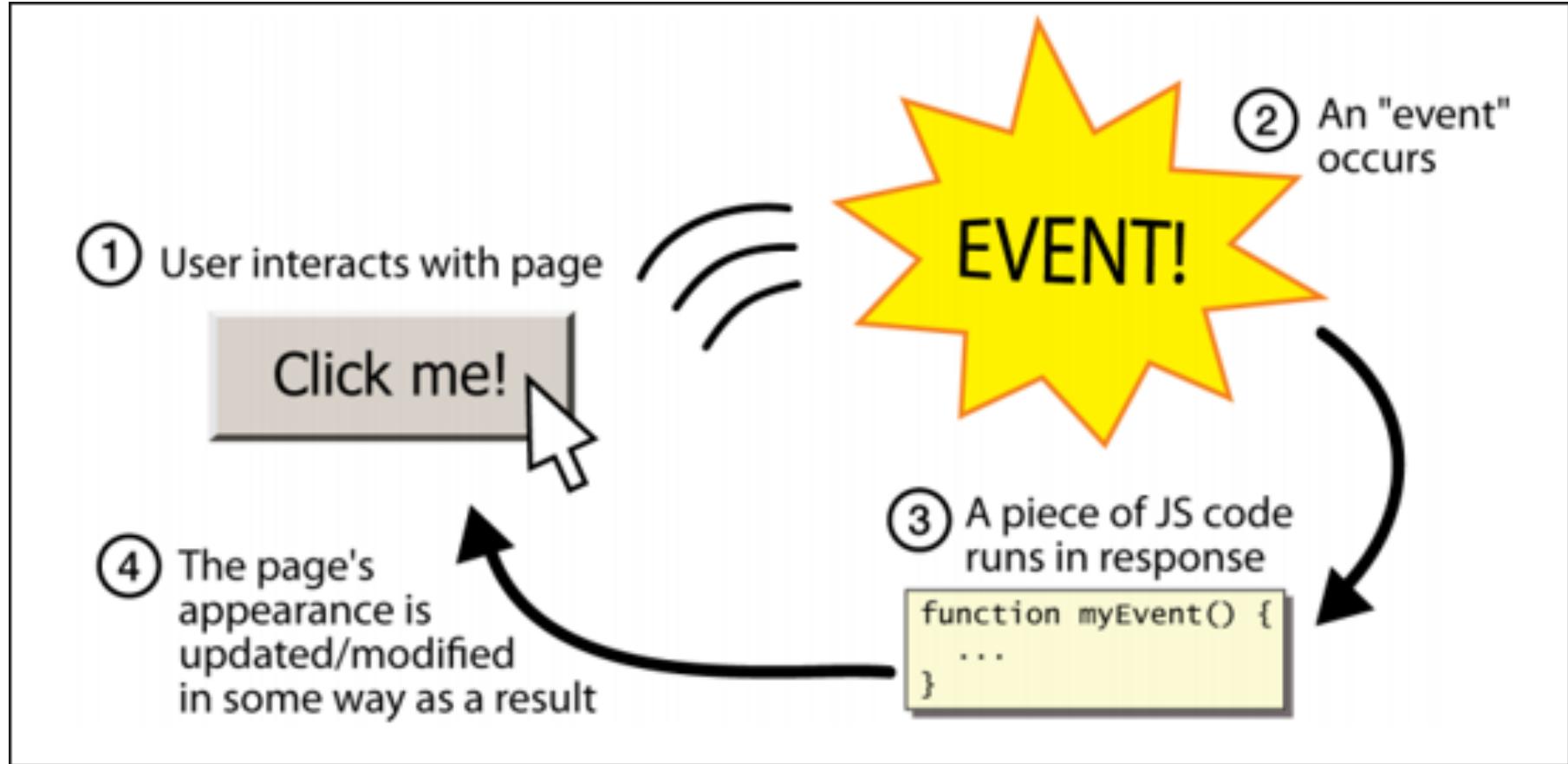
- In general, to add interactivity to our HTML/CSS websites we need to:
 - Link a JavaScript program to our HTML
 - Identify the elements we want to interact with (e.g., ``, `<button>`, page)
 - Identify the events to respond to (e.g., click, hover, input)
 - Define the response functions that handle the desired behavior
 - Assign event listeners to the elements

Linking to a JS file: <script>

```
1 <!--template-->
2 <script src="filename"></script>
3
4 <!--example-->
5 <script src="example.js"></script>
```

- The <script> tag should be placed in the HTML page's <head>. All JS code used in the page should be stored in a separate .js file
- JS code can be placed directly in the HTML file's body or head (like CSS), but this is poor code quality. You should always separate content, presentation, and behavior by keeping these “concerns” in separate files!!

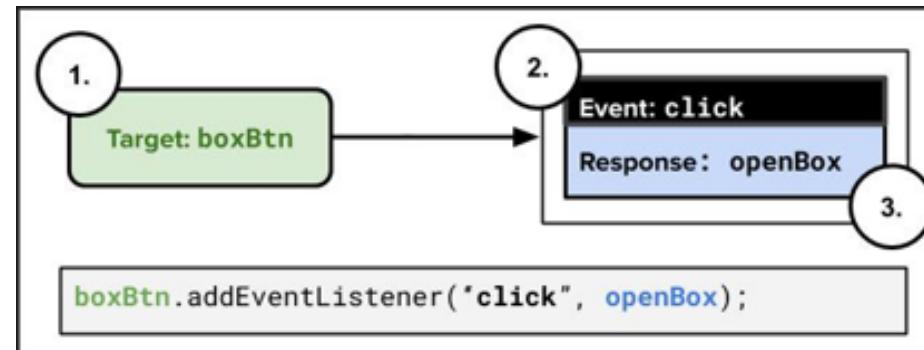
Event-driven programming



- Unlike Java programs, JS programs have no main; they respond to user actions called events
- **Event-Driven Programming**: writing programs driven by user events

Event handling

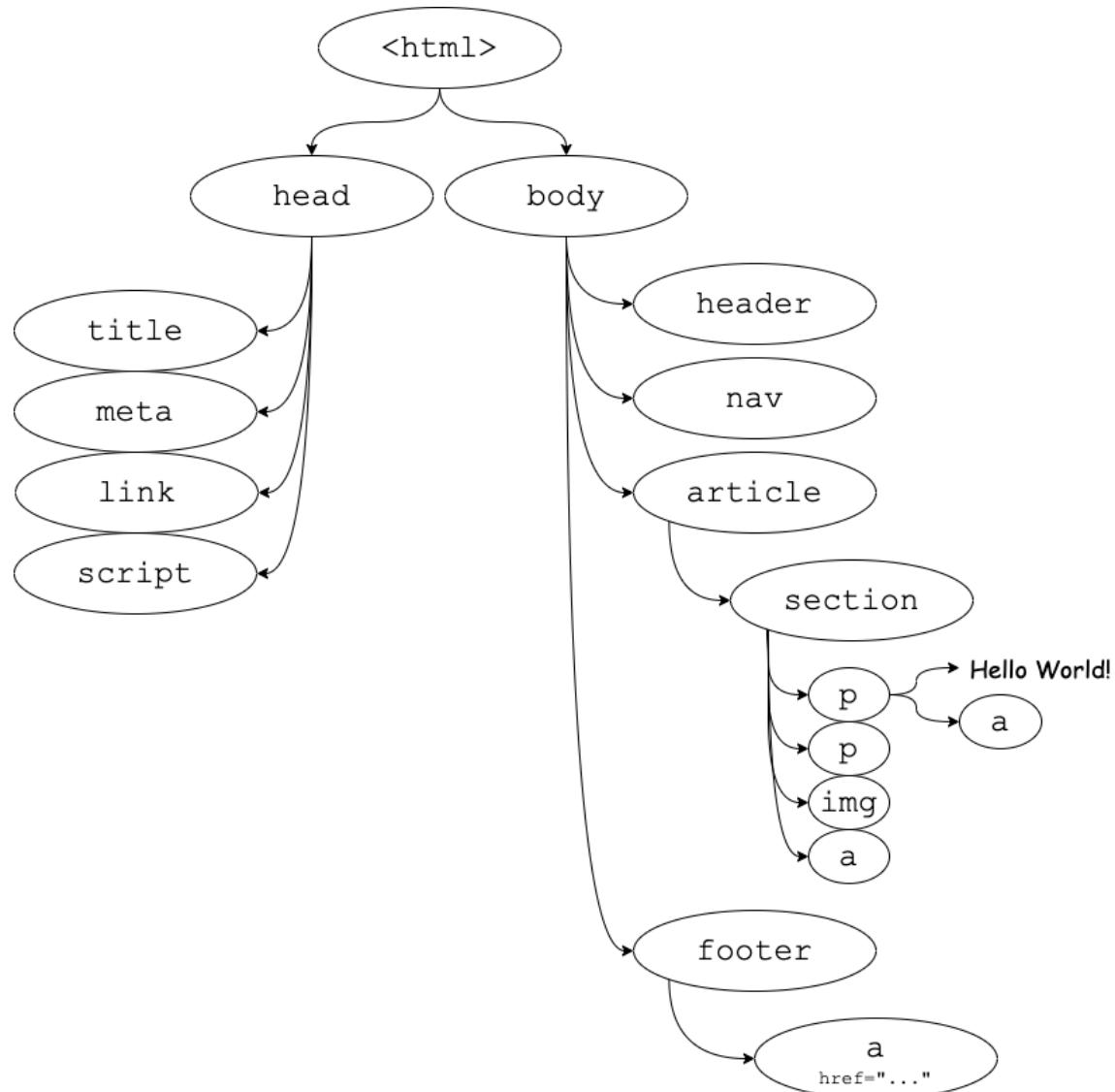
- We can use JavaScript to attach functions to elements when an event (e.g. “click”) occurs.
- To do so, we need to:
 1. Identify the source element to listen for an event
 2. Indicate the event to trigger a response
 3. Specify the response function(s) to call when the event is triggered.
- Once we have this information, we can use `addEventListener` to hook everything up!



Identify Source Element

Select the DOM elements to which we will attach event listeners.

DOM tree



```

1 <html>
2 <head>
3   <title>My Fancy Title</title>
4   <meta charset="UTF-8">
5   <link rel="stylesheet" href="styles.css">
6   <script src="script.js"></script>
7 </head>
8 <body>
9   <header>...</header>
10  <nav>...</nav>
11  <article>
12    <section>
13      <p>Hello world: <a href="...">here</a></p>
14      <p>Welcome!</p>
15      
16      <a href="...">citation</a>
17    </section>
18  </article>
19  <footer>
20    <a href="..."></a>
21  </footer>
22 </body>
23 </html>
  
```

- Selectors (class ., id # or type)

Document interface

- The `Document`¹ interface represents any web page loaded in the browser and serves as an entry point into the web page's content, which is the **DOM tree**
- We can access and manipulate the DOM through the `Document` instance of an HTML page
 - `document` is the actual object for your html page loaded in browser. This is a DOM object.
 - `DOMContentLoaded` event fires when the document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading.
 - The window `load` event is also used for this, it fires when the resources have successfully loaded.
 - `getElementById`, `querySelector` and `querySelectorAll` instance methods are used to access elements on the DOM

Accessing an element by id

```
1 let name = document.getElementById("id");
```

- `document.getElementById()` returns the DOM object for an element with a given id (note that you omit the `#` when giving an id)
- If the id does not exist, there is no error. Instead the returned result is `null`

Accessing a single node using querySelector

```
1 let name = document.querySelector(selector) → '#, .' are required!
```

- `document.querySelector()` returns **only** the first element that would be matched by the given CSS selector string
- If no element matches the passed selector, there is no error. Instead the returned result is **null**

Accessing a collection of nodes using querySelectorAll

```
1 let name = document.querySelectorAll(selector)
```

- `document.querySelectorAll()` returns all elements that would be matched by the given CSS selector string
- If no element matches the passed selector, there is no error. Instead an **empty array** is returned.

Handy shortcut functions

- It's handy to declare a shortcut to help us out as we will use these methods A LOT.
- You may use the following in your JS programs (these are exceptions to the rule of having descriptive function names):

```
1 function id(id) {  
2     return document.getElementById(id);  
3 }  
4  
5 function qs(selector) {  
6     return document.querySelector(selector);  
7 }  
8  
9 function qsa(selector) {  
10    return document.querySelectorAll(selector);  
11 }
```

Identify the events to respond to

Determine the specific user or page events (e.g., click, hover, input) that should trigger a response.

Common types of JavaScript events

Event	Description
load	Webpage has finished loading the document
scroll	User has scrolled up or down the page
click	A pointing device (e.g., mouse) has been pressed and released on an element
dblclick	A pointing device button is clicked twice on the same element
keydown	Any key is pressed
keyup	Any key is released

You can find a full list [here](#)

Assign event listeners to the elements

Attach handling functions to the selected elements so that they are triggered when the corresponding events occur

Listening & Responding to Events

```
1 // attaching a named function
2 element.addEventListener("click", handleFunction);
  event name           function to trigger
3
4 function handleFunction(e) {
5   // event handler code
6   // parameter e is optional and captures
7   // more information about the event
8 }
9
```

```
1 // attaching an anonymous function
2 element.addEventListener("click", function (e) {
3   // event handler code
4 });
5
6 // attaching an arrow function
7 element.addEventListener("click", (e) => {
8   // event handler code
9 });
```

- JS functions can be set as event handlers (also known as “callbacks”¹)
 - A callback is just a function that’s passed into another function, with the expectation that the *callback* will be called at the appropriate time
- When you interact with the element and trigger the event, the *callback* function will execute \Rightarrow *asynchronous function*: moving on without waiting for job finish
- **click** is just one of many event types we’ll use

Event handler function syntax

What's the difference between Statement 1 and Statement 2?

Statement 1 *wrong!*

just name of function

```
1 element.addEventListener("click", openBox);
```

Statement 2

```
1 element.addEventListener("click", openBox());
```

Event handler function syntax

What's the difference between Statement 1 and Statement 2?

Statement 1

```
1 element.addEventListener("click", openBox);
```

Statement 2

```
1 element.addEventListener("click", openBox());
```

addEventListener with multiple events

```
1 myBtn.addEventListener("click", handleClick);
2 myBtn.addEventListener("click", logClick);
3 myBtn.addEventListener("mouseover", handleMouseOver);
```

- We can add **multiple eventListeners** to the same element.
 - Multiple **eventListeners** for the same event but with different *callback* functions
 - Multiple **eventListeners** each for different events

removeEventListener after adding one

```
1 myBtn.addEventListener("click", function1);
2 myBtn.addEventListener("click", function2);
3
4 myBtn.removeEventListener("click", function1);
```

- After adding an **eventListener** we can remove it so that the behavior associated with the event is no longer triggered.
- You cannot remove anonymous functions, since you dont have a name to refer to them
- Calling removeEventListener() with arguments that do not identify any currently registered event listener on the EventTarget has no effect.

Define the response functions

Create functions that handle the desired behavior when an event occurs,
usually manipulating the DOM

What's inside a DOM Node?

```
1   
2 <p>picture taken from google</p>
```

- For starters, the HTML attributes. This HTML above is a DOM object
- Let's assume a variable for the tag called `myImg` is properly defined) with these two properties:
 - `myImg.src` – set by the browser to `images/duck.png`
 - `myImg.alt` – set by the browser to “A useful rubber duck photo”
- Let's assume a variable for the `<p>` tag called `para` is properly defined) with this property
 - `para.textContent` – set by the browser to “picture taken from google”

Manipulating DOM elements

- Most JS code manipulates elements on an HTML page
- We can examine elements' state
 - e.g. see whether a box is checked
- We can change the state of elements
 - e.g. insert some new text into a div
- We can change the style of an element
 - e.g. make a paragraph red

Listening to load events

- You can only access document element after the window “*load*” event has fired
- You can also access the DOM after the document *DOMContentLoaded* event has fired
 - without waiting for stylesheets, images, and subframes to finish loading

```
1 window.addEventListener("load", init);
2 //or
3 document.addEventListener("DOMContentLoaded", init);
4
5 // no access to the document here
6
7 function init() {
8     // we now have access to the DOM tree!
9     // set up your initial document event handlers here.
10 }
11
```

Modifying DOM elements

- We can modify HTML element properties and style

DOM Element

```
1 <p id="text">  
2   This is the original text.  
3 </p>
```

JS

```
1 document.getElementById("text").textContent = "Text has been changed!";  
2  
3 document.querySelector("#text").style.backgroundColor = "lightblue";
```

access CSS

Creating DOM elements

- For creating a new DOM element, we use the `document.createElement("tag")` function which creates and returns a new empty DOM node representing an element of that type

```
1 // create a new <h2> node
2 let newHeading = document.createElement("h2");
3 newHeading.textContent = "This is a new heading!";
```

! Important

- Merely creating an element does not add it to the page
- You must add the new element as a child of an existing element on the page.

Adding and Moving Nodes on the DOM

- When you have a parent DOM node, you can add or remove a child DOM node using the following functions:

Name	Description
<code>parent.appendChild(node)</code>	Places the given node at the end of this node's child list
<code>parent.insertBefore(new, old)</code>	Places the given node in this node's child list just before the old child
<code>parent.replaceChild(new, old)</code>	Replaces the given child with new nodes
<code>parent.insertAdjacentElement(location, new)</code>	Inserts a new element at the specified location

Example

```
1 let li = document.createElement("li");
2 li.textContent = "A list item!";
3 id("my-list").appendChild(li);
```

Removing DOM elements

- When you have a parent DOM node, you can remove a child DOM node using the following functions:

Name	Description
parent.removeChild(node)	Removes the given node from this node's child list
node.remove()	Removes the node from the page

Example

```
1 document.querySelector(".list-element").remove();
2 /* or */
3 let li = document.querySelector(".list-element");
4 li.parentElement.removeChild(li);
```

Hiding/Showing DOM elements

How can we hide an HTML element?

```
1 #my-img {  
2   display: none;  
3 }
```

In JS, it's possible to modify the style properties of an element directly

```
1 document.getElementById("my-img").style.display = "none";
```

The <button>

```
1 <button id="my-btn">Go!</button>
```

- Text inside of the button tag renders as the button text
- To make a responsive button (or other UI controls):
 1. Choose the control (e.g., button) and event (e.g., mouse click) of interest
 2. Write a JS function to run when the event occurs
 3. Attach the function to the event on the control

Open the Box

- In this example code, we modify an image by pressing a button.

Code example

Output

HTML

```
1 
2 <button id="box-btn">Click me!</button>
```

JS

```
1 let boxBtn = document.getElementById("box-btn");
2 boxBtn.addEventListener("click", openBox);
3
4 function openBox() {
5   let box = document.getElementById("mystery-box"); // 1. Get the box image
6   box.src = "star.png"; // 2. Change the box image's src attribute!
7 }
```

Next week

Asynchronous JavaScript, JSON and APIs



Acknowledgements

- Some contents of this lecture are partially adapted from:
 - Harvard **CS50's Web Programming with Python and JavaScript**, licensed under **CC BY-NC-SA 4.0**.
 - Materials from University of Washington's **CSE 154 Web Programming** (used with permission).
 - **The Odin Project** (main website code under MIT license and curriculum licensed under a **CC BY-NC-SA 4.0**)
 - **The Fundamentals of Web Application Development (Web Edition)** ©2025 Nicholas D. Freeman. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - **freeCodeCamp.org** © 2025 freeCodeCamp.org. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.

Web Programming

Lecture 5

*Asynchronous Programming in JavaScript,
JSON and AJAX*

Josue Obregon 

jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

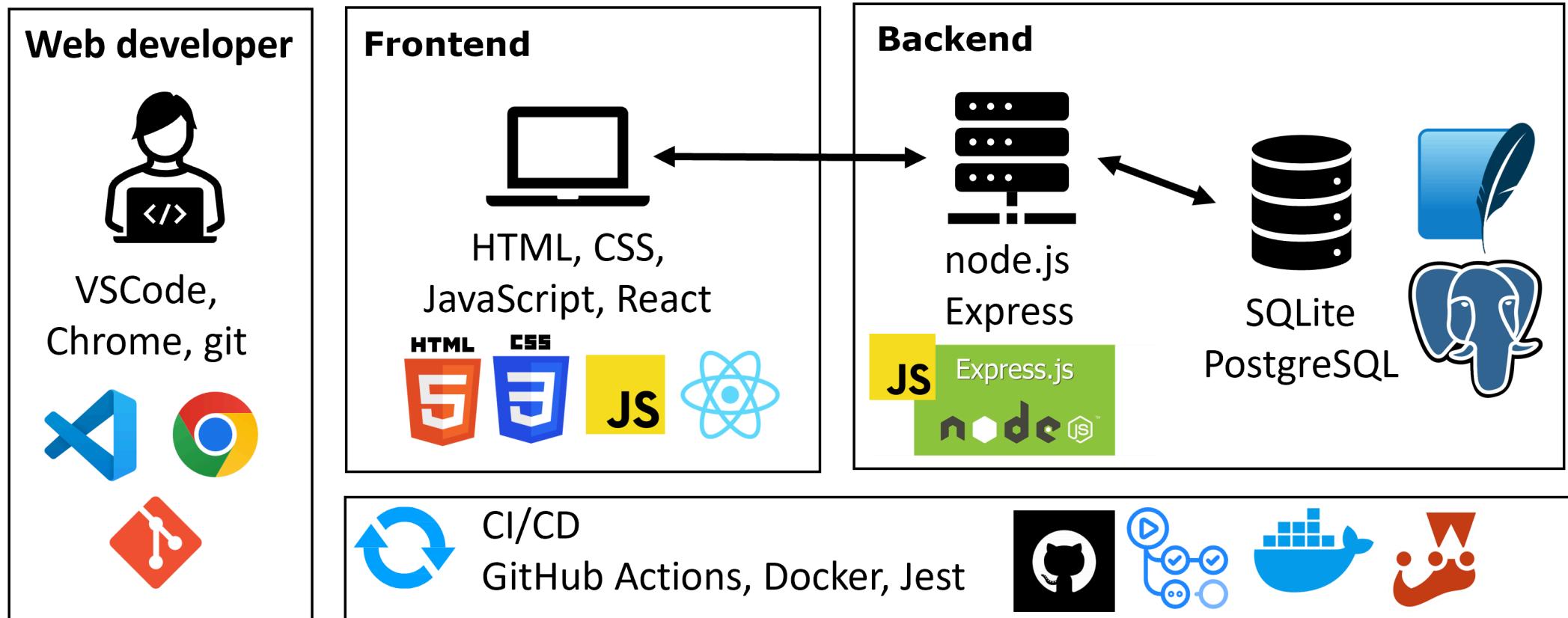
 [Lecture slides index](#)

April 1, 2025



Course structure

Modern Web Development Technology Stack



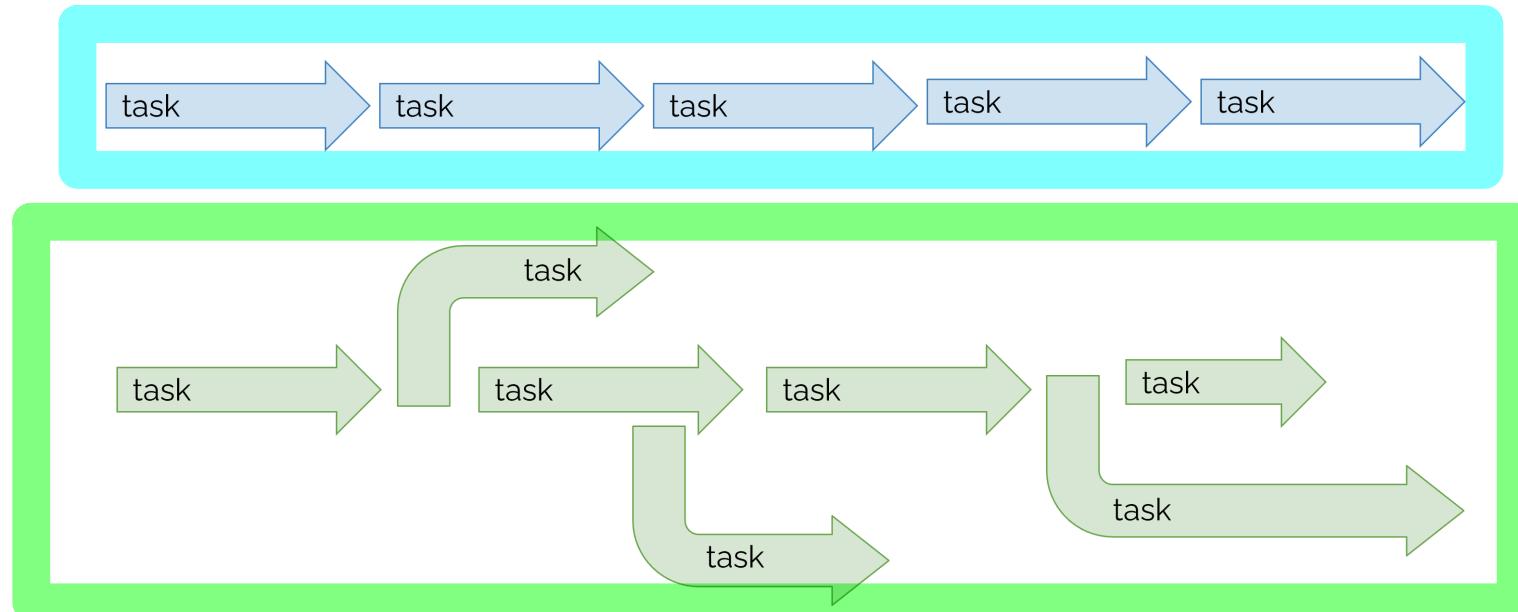
Roadmaps: **Frontend**, **backend** and **fullstack**

Agenda

- JavaScript
 - Asynchronous programming
 - Timers
 - Callback
 - Promises
 - asynch/await
 - JSON
 - AJAX

Asynchronous programming

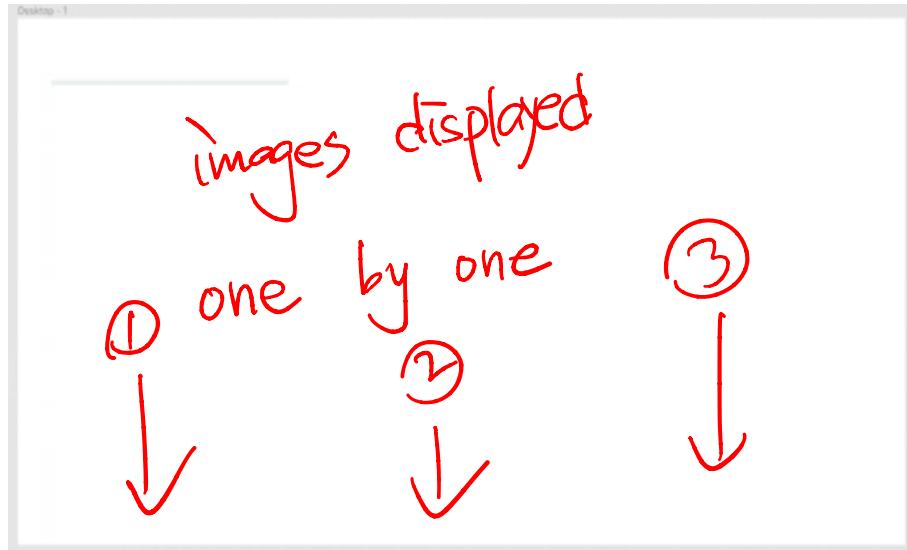
- In JavaScript, synchronous code is executed in a blocking manner, meaning that the code is executed serially, one statement at a time.
- In JavaScript, asynchronous programming is a fundamental part of the language. It is the mechanism that allows us to perform operations in the background, without blocking the execution of the main thread.
- This is especially important in the browser, where the main thread is responsible for **updating the user interface and responding to user actions**.
- In the figure below, the blue arrow tasks represent **synchronous programming**, whereas the green arrow tasks represent **asynchronous programming**



Synchronous vs. Asynchronous systems

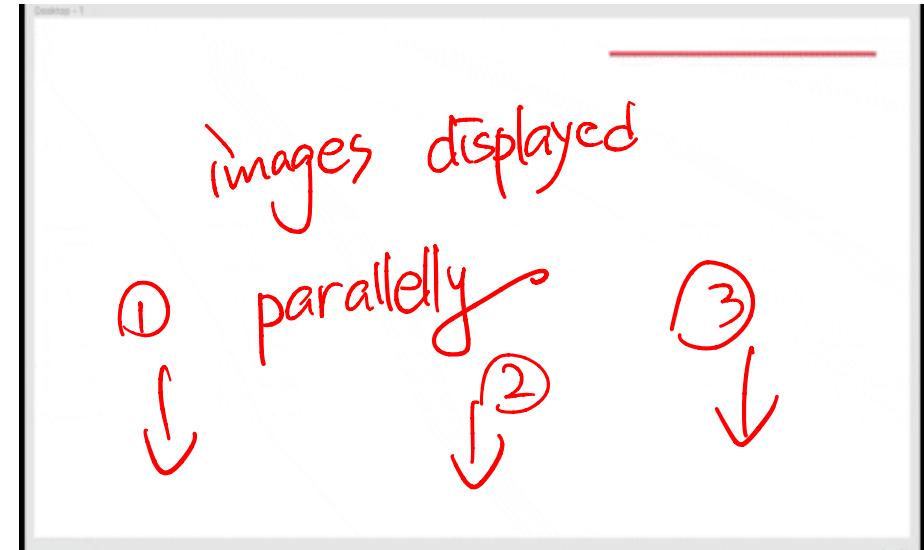
Synchronous system

- Images are loaded serially, one image at a time.



Asynchronous system

- Images are loaded parallelly, every image at its own time.



Why is JavaScript so different?

- Java, and other compiled languages, are often used to build systems.
 - Objects are great to compose together to build complex systems.
 - Systems must be reliable - a benefit of strict types, compiling, and well-defined behavior in Java.
- JavaScript is used to interact and communicate.
 - It listens.
 - It responds.
 - It requests.
- Whereas in Java, programs often have a well-defined specification (behavior), JS has to deal with uncertainty (unusual user behavior, unavailable servers, no internet connection, etc.)

The asynchronous mindset

- To master asynchronous programming, start thinking about your code in a non-linear way.
- Instead of writing linear code, **structure it around events and outcomes.**

JavaScript offers three main asynchronous tools:

- **Callbacks:** Functions passed and executed when events occur. Simple but can lead to “callback hell.”
- **Promises:** Introduced in ES6, provide a cleaner way to manage async flows with better readability and state tracking.
- **Async/Await:** Syntax sugar over promises, enabling more readable and maintainable code.

Event listeners are asynchronous

- Event listeners can be specified as callback function, which is asynchronous programming
- Think about it, when we add the event listener, we are not executing `callbackFn` or the anonymous function right away.
- Those functions are executed when the click event happens.

```
1 btn.addEventListener('click', callbackFn);
2 btn.addEventListener('click', function() {
3   ...
4 });
5 btn.addEventListener('click', () => {
6   ...
7 });
```

JS treat function as first class function.

↳ assign to var
possible | argument of function
return value of function.

Callback functions

- Callbacks exploit JavaScript's capability to pass functions.
- There are two essential parts to this technique:
 1. A function that is passed as an argument to another function
 2. The passed function is executed when a certain event happens
- Let's see an example of a callback function

```
1 const doSomething = (cb) => {  
2   console.log('Doing something...');  
3   cb(); call callback function.  
4 };
```

```
1 const nextStep = () => {  
2   console.log('Callback called');  
3 };  
4 doSomething(nextStep);
```

- Callbacks are just a pattern where we expect that the next function to be executed is actually called as the final step (call me back when you are done - callback)
- Note that the function that is passed as an argument is not executed immediately.

Callback functions with parameters

- It is also possible to pass a function that receives arguments.

```
1 const calculateNameLength = (name, cb) => {
2   const length = name.length;
3   cb(length);
4 };
5 calculateNameLength('John', (length) => {
6   console.log(`The name length is ${length}`); // The name length is 4
7 });
```

Timers and intervals

Name	Description
<code>setTimeout(callBack, delayMS)</code>	Arranges to call given function after given delayMS, returns timer id
<code>setInterval(callBack, delayMS)</code>	Arranges to call function repeatedly every delayMS ms, returns timer id
<code>clearTimeout(timerID)</code>	Stop the given timer
<code>clearInterval(timerID)</code>	Stop the given interval

- There are two functions that are commonly used to delay the execution of a function, `setTimeout` and `setInterval`.
- Both functions receive a callback as an argument and execute it after a certain amount of time.
- Both `setTimeout` and `setInterval` return an ID representing the timer.
- A unique identifier the window has access to in order to manage the page timers.
- If you have access to the id, you can tell the window to stop that particular timer by passing it to `clearTimeout/clearInterval` later

setTimeout example

HTML code

```
1 <button id="demo-btn">Click me!</button>
2 <p id="output-text"></p>
```

JS code

```
1 document.addEventListener('DOMContentLoaded', init);
2 function init() {
3     id("demo-btn").addEventListener("click", delayedMessage);
4 }
5
6 function delayedMessage() {
7     id("output-text").textContent = "It's gonna be legend...wait for it...";
8     setTimeout(sayHello, 3000);
9 }
10
11 function sayHello() { // called when the timer goes off
12     id("output-text").textContent = "dary... Legendary!";
13 }
```

setInterval example

HTML code

```
1 <p id="timer-text"></p>
```

JS code

```
1 let timerId = null; // stores ID of interval timer
2 function repeatedMessage() {
3     timerId = setInterval(sayAnnyeong, 1000);
4 }
5
6 function sayAnnyeong() {
7     id("timer-text").textContent += "안녕!";
8 }
```

Error first callbacks

- The most common way to handle errors in callbacks is to use the error first pattern.
- This pattern consists of passing an error as the first argument of the callback, and the result as the second argument.
- In the example below, `cb` receives two arguments, an `error` and a `result`.
- In the example on the left, we pass an error, whereas in the example of the right, we pass a success result.

```
1 const doSomething = (cb) => {
2   const error = new Error('Something went wrong');
3   cb(error, null);
4 };
5
6 doSomething((error, result) => {
7   if (error) {
8     console.log('There was an error');
9     return;
10 }
11 console.log('Everything went well');
12});
```

```
1 const doSomething = (cb) => {
2   const result = 'It worked!';
3   cb(null, result);
4 };
5 doSomething((error, result) => {
6   if (error) {
7     console.log('There was an error');
8     return;
9   }
10  console.log(result);
11  console.log('Everything went well');
12});
```

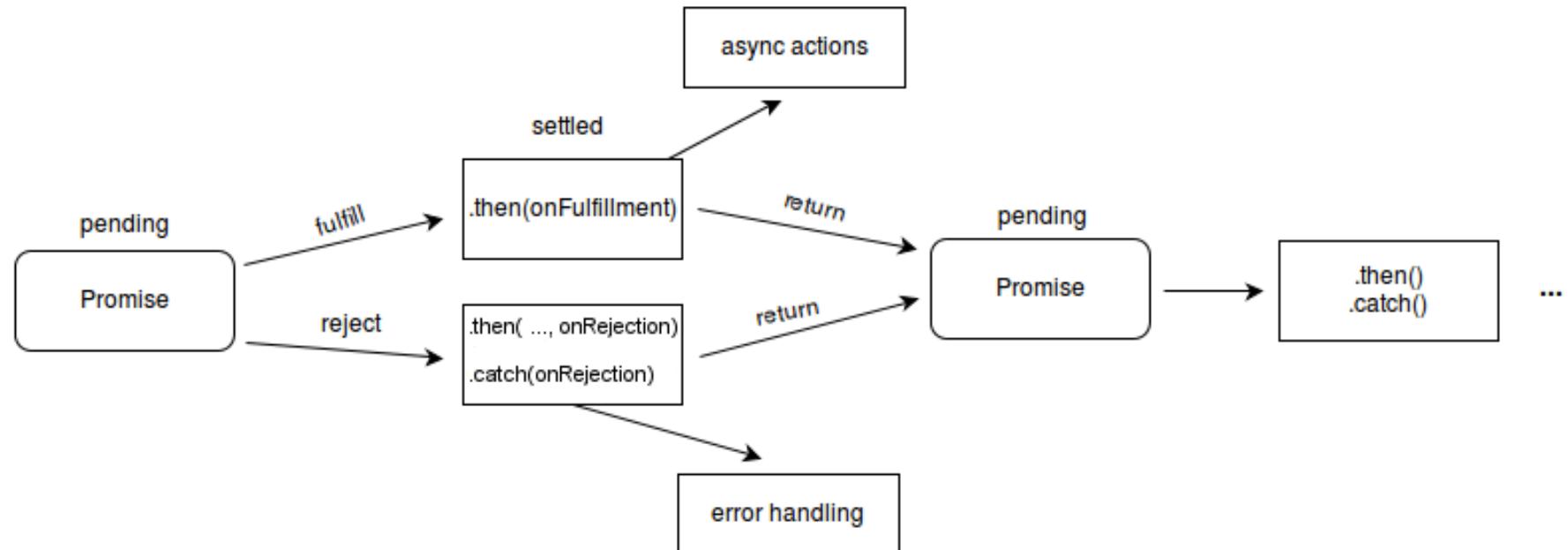
Callback hell

- Callbacks are not easy to read, and when there are a lot of nested callbacks, the code becomes very hard to understand.
- This is called callback hell, and it is a common problem when using callbacks.

```
1  readFile("docs.md", (err, mdContent) => {
2      convertMarkdownToHTML(mdContent, (err, htmlContent) => {
3          addCssStyles(htmlContent, (err, docs) => {
4              saveFile(docs, "docs.html", (err, result) => {
5                  ftp.sync((err, result) => {
6                      // ...
7                  })
8              })
9          })
10     })
11 })
```

Promises

- The `Promise` object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.
- A promise has states, and it can exist in any of three states: pending, fulfilled, or rejected.
- When a promise is created, it is in the pending state. When a promise is fulfilled, it is in the fulfilled state. When a promise is rejected, it is in the rejected state.
- After a promise is fulfilled or rejected, it becomes unchangeable
 - To manage fulfillment, the `then` method is employed, while the `catch` method is used to address the rejection of the promise.



Example of Promises

- Let's see how this works with a simple example using `fetch` to make a request to an external application programming interface (API).

```
1 fetch('https://pokeapi.co/api/v2/pokemon/gengar')  
2   .then(response => response.json())  
3   .then(json => console.log(json))  
4   .catch(error => console.log(error));
```



Creating promises

- You can create a promise using the Promise constructor, which receives a callback function as an argument (**executorFn** in the code example).
- This callback function receives two arguments, **resolve** and **reject**.
 - The resolve function is used to resolve the promise, and the reject function is used to reject the promise.
- **executorFn** eventually determines the state of the promise
- It is up to us to define this executor function
- When created, the **Promise** always initially has a state of pending

```
1 let promiseObj = new Promise(executorFunction);
2
3 function executorFunction(resolve, reject) {
4     // ...
5     if (conditionMet) {
6         resolve(); // Passed by the Promise object
7     } else {
8         reject(); // Passed by the Promise object
9     }
10 }
```

Example of creating a promise

- In this example, we have a function called `setTimeoutPromise` that receives a time as an argument.
- This function returns a promise that will be resolved after the specified time. When the promise is resolved, we print one second later to the console.

```
1  function setTimeoutPromise(time) {
2      function executorFunction(resolve, reject) {
3          setTimeout(function () {
4              resolve();
5          }, time);
6      }
7
8      return new Promise(executorFunction);
9  }
10
11 console.log('Before setTimeoutPromise');
12 setTimeoutPromise(1000).then(function () {
13     console.log('one second later');
14 });
15 console.log('After setTimeoutPromise');
```

Callback hell with promises

- Promises are a great way to deal with the limitations that callbacks introduce when we need to perform multiple asynchronous operations that should be executed in a consecutive order.
- Promises will handle errors more easily, so the readability of the code should be clearer and easier to maintain in long term.

```
1 readFile("docs.md")           // returns a Promise
2   .then(convertMarkdownToHTML) // gets the content of docs.md
3   .then(addCssStyles)        // gets the HTML from the previous step
4   .then(docs => saveFile(docs, "docs.html")) // saves the final version
5   .then(ftp.sync)            // syncs to the server
6   .then(result => {
7     // do something with result
8   })
9   .catch(error => console.log(error));
```

- Note that Each `.then(...)` receives the resolved value from the previous Promise, and passes its own result to the next `.then()`.
 - This means that when you pass a function reference like in the left code block, is the same as writing the contents of the right code block

```
1 .then(convertMarkdownToHTML)
```

```
1 .then(data => convertMarkdownToHTML(data))
```

async/await

- ES2017 introduced a new way to handle asynchronous code, the `async` and `await` keywords.
- These keywords are syntactic sugar for promises
 - they are not a new way to handle asynchronous code, but they make the code much easier to read and write
- `async` wraps the return value in a Promise whose resolved value is the return value
- `await` halts execution of the code within scope of the function it's defined in until the Promise is resolved and then returns the resolved value of the promise
- async/await are useful because it makes our code “look” synchronous again
 - You don't need callbacks, you can just have regular functions

Using `async` keyword (trivial example)

- Labeling a function as `async` wraps the function in a `Promise`
- Take a look in the differences between what gets printed to the console

```
1 function sillyExample1() {  
2     return 'look at what I return regular';  
3 }  
4  
5 async function sillyExample2() {  
6     return 'look at what I return async';  
7 }  
8  
9 console.log(sillyExample1());  
10 console.log(sillyExample2());
```

Using `await` keyword (trivial example)

```
1 function newPromise(time) {
2     return new Promise((resolve, reject) => {
3         setTimeout(resolve, time, 'wow');
4     });
5 }
6 // Example 1: Using .then() to handle the resolved value
7 let example1 = newPromise(2000);
8 example1.then(result => {
9     console.log(result + ' that was simple (promise)');
10});
11
12 // Example 2: Using await (inside an async function)
13 async function runExample2(time) {
14     let example2 = await newPromise(time);
15     console.log(example2 + ' that was simple (async/await)');
16 }
17 runExample2(4000);
```

- Using `await` means that we are halting the execution of our code until the Promise has resolved (or rejected)
- Look at the differences between the log statements
- If you use the `await` keyword inside of a function, the function needs to be labeled as `async`
- All `async` functions need to be awaited when called

Error Handling with `async/await`

- Utilize the try/catch block
- All code to execute in success cases goes in the try block
- Error handling code goes in the catch block
- Anything awaited belongs in a try block

```
1 try {  
2     someImportantFun();  
3     await someFuncThatReturnsAPromise();  
4     yetAnotherFuncThatIsImportant();  
5 } catch(err) {  
6     // replace w/ elegant error handling  
7     console.log(err);  
8 }
```

JavaScript Object Notation (JSON)

- Data format that represents data as a set of JavaScript objects
- Natively supported by all modern browsers (and libraries to support it in old ones)
- ~~Not yet as popular as XML, but steadily rising due to its simplicity and ease of use~~
- **Updated:** JSON is a lightweight substitute for XML and it is more popular than XML because of JavaScript's dominance as the most widely used language of today.

JavaScript objects

```
1 let person = {  
2     name: "Philip J. Fry",           // string  
3     age: 23,                      // number  
4     "weight": 172.5,               // number  
5     friends: ["Farnsworth", "Hermes", "Zoidberg"], // array  
6 };  
7  
8 person.age;  
9 person["weight"]; // 172.5  
10 person.friends[2]; // "Zoidberg"  
11 let propertyName = "name";  
12 console.log(person[propertyName]); // "Philip J. Fry"
```

- Objects are the most versatile structure in JS
- You can access the fields with dot notation (.fieldName) or bracket notation ([“fieldName”]) syntax
- You can create a new property or overwrite existing ones in an object by assigning a value

Example of JSON

```
1 {  
2   "name": "Philip J. Fry",  
3   "age": 23,  
4   "weight": 172.5,  
5   "friends": [  
6     "Farnsworth",  
7     "Hermes",  
8     "Zoidberg"  
9   ]  
10 }
```

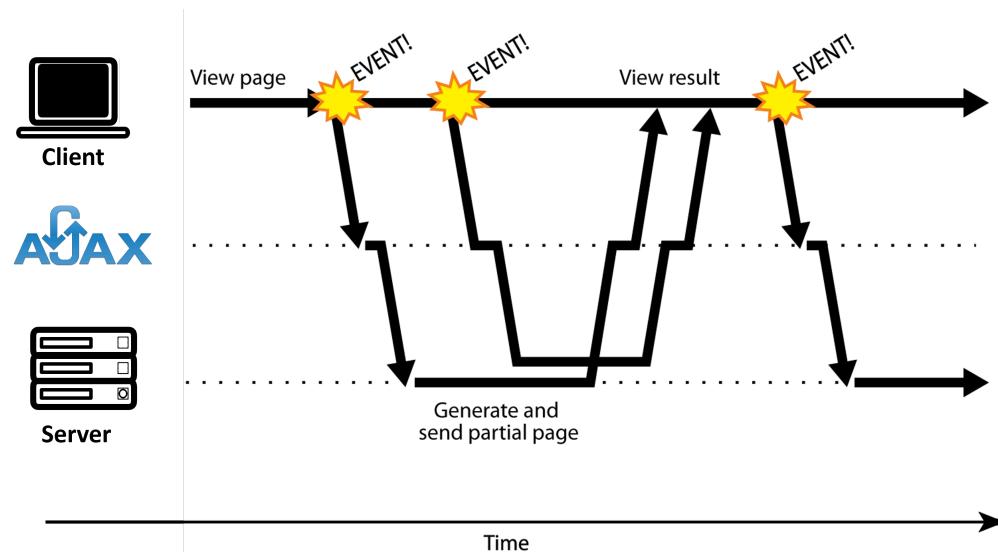
- Keys (attributes) must be in double quotes:
 - name: "Fry" → Invalid
 - "name": "Fry" → Valid
- Strings must use double quotes (not single quotes):
 - 'Fry' → Invalid
 - "Fry" → Valid
- No functions, comments, or trailing commas allowed:
 - JSON is pure data – not code.
 - You can't include `function() {}` or comments like `// this is a comment.`

JavaScript Objects vs. JSON

- JSON is a way of representing objects, or structured data.
 - The technical term is “serializing” which is just a fancy word for turning an object into a savable string of characters
- Browser JSON methods:
 - `JSON.parse(/* JSON string */)` – converts JSON string into Javascript object
 - `JSON.stringify(/* Javascript Object */)` – converts a Javascript object into JSON text

AJAX

- Web application: a dynamic web site that mimics the feel of a desktop app presents a continuous user experience rather than disjoint pages
 - Examples: Gmail, Facebook, etc.
- AJAX: Asynchronous JavaScript and XML
 - It is not a technology, but rather a programming pattern
 - Downloads data from a server in the background
 - Allows dynamically updating a page without making the user wait
 - Avoids the “click-wait-refresh” pattern
 - Currently it should be called “AJAJ”??



AJAX

- **XMLHttpRequest**(callback-based)
 - XMLHttpRequest (XHR) objects are used to interact with servers. You can retrieve data from a URL without having to do a full-page refresh.
 - This enables a Web page to update just part of a page without disrupting what the user is doing.
- **Fetch API** (promise-based)
 - It provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network.
 - Unlike XMLHttpRequest that is a callback-based API, Fetch is promise-based and provides a better alternative

Next week

Backend development and Introduction to Node.js



Acknowledgements

- Some contents of this lecture are partially adapted from:
 - Harvard [CS50's Web Programming with Python and JavaScript](#), licensed under [CC BY-NC-SA 4.0](#).
 - Materials from University of Washington's [CSE 154 Web Programming](#) (used with permission).
 - [The Odin Project](#) (main website code under MIT license and curriculum licensed under a [CC BY-NC-SA 4.0](#))
 - [The Fundamentals of Web Application Development \(Web Edition\)](#) ©2025 Nicholas D. Freeman. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [freeCodeCamp.org](#) © 2025 freeCodeCamp.org. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [Node.js for Beginners](#) by Ulises Gascón. Some contents adapted under fair educational use for instructional purposes.

Web Programming

Lecture 6

AJAX, JSON and RESTful APIs

Josue Obregon 

jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

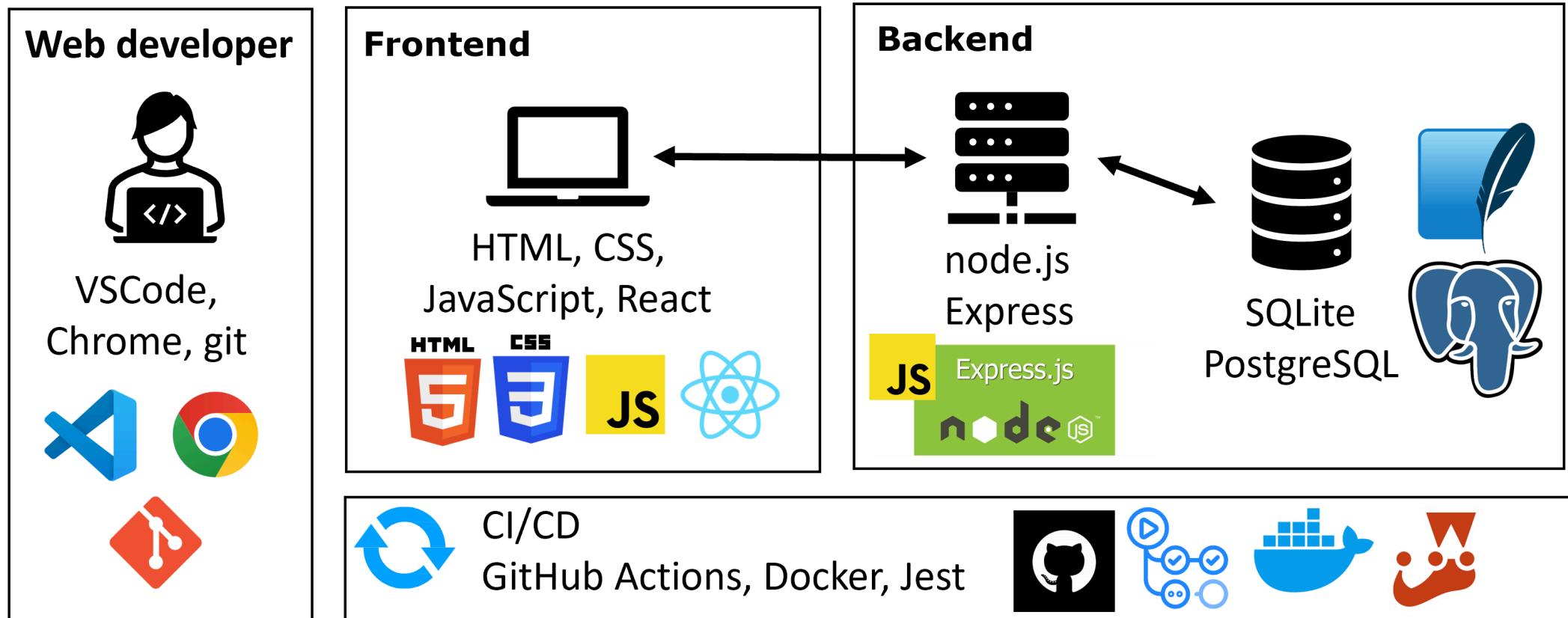
 [Lecture slides index](#)

April 8, 2025



Course structure

Modern Web Development Technology Stack



Roadmaps: **Frontend**, **backend** and **fullstack**

Agenda

- HTTP
- APIs
 - RESTful APIs
- JSON
- AJAX
- fetch

HTTP – server and client relationship

- Although web development can be quite complex, we can simplify it by focusing on how the server and client interact in a typical web application.

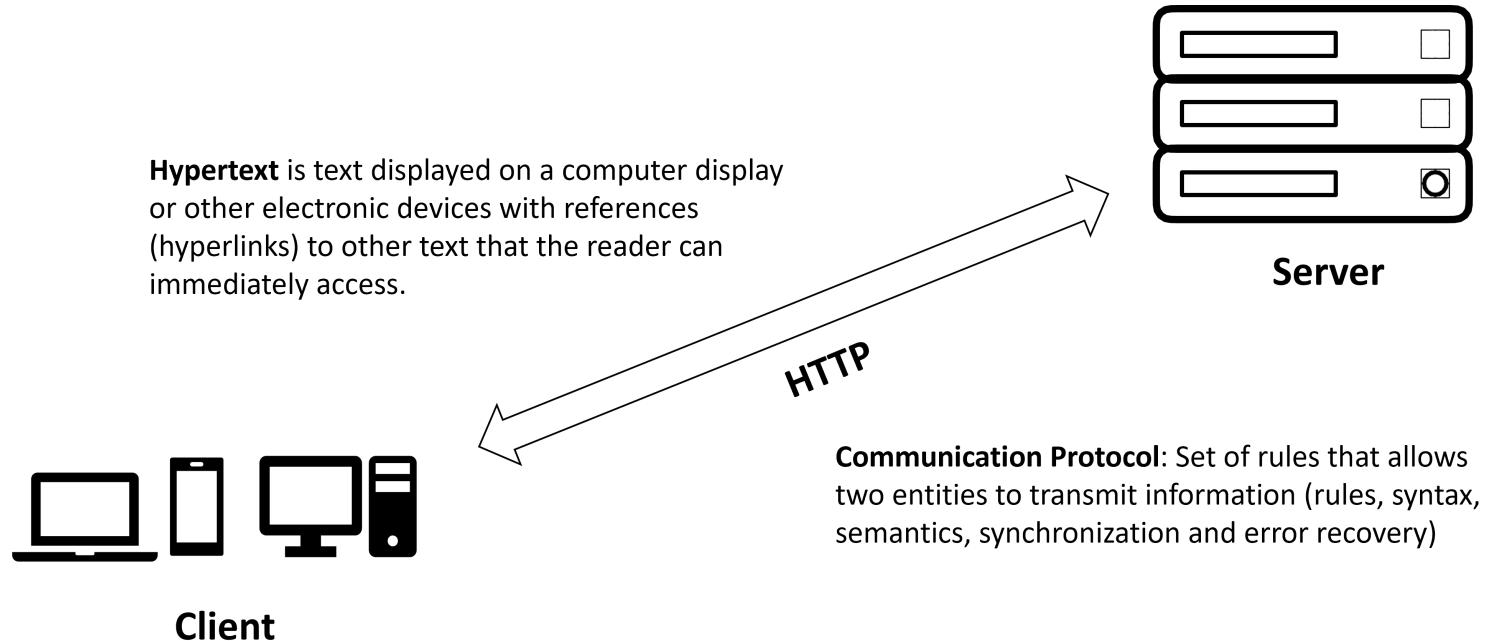
Two main participants:

- **Server**: The machine running the application, handling database queries and other backend tasks.
- **Client**: The software on the user's device—usually a browser executing HTML, CSS, and JavaScript—also called the frontend.

They communicate over **HTTP**: the client issues a request, and the server returns a response, forming the standard request/response cycle.

Request and response

- HTTP revolves around two primary elements: the **request** and the **response**.
- The client issues a request to the server, and the server processes it before returning a response.
 - Each message is composed of multiple parts, which we will explore in the following slides.
- As shown, a single server can serve many clients at once. This is the standard web application architecture: the server handles incoming requests and delivers the corresponding responses.



HTTP – Multiple Client Requests to Multiple Servers

- A client often sends multiple HTTP requests to one or more servers.
- Check the following HTTP snippet

```
1 <head>
2   <link rel="stylesheet" href="https://server1.com/style.css">
3 </head>
4 <body>
5   
6   <script src="https://server2.com/script.js"></script>
7 </body>
```

- In this case, requests go to `server1.com`, `server2.com`, and `server3.com`.
- Each server returns its specific resource.
- Open your browser's DevTools (Network tab) on a site like <https://itm.seoultech.ac.kr> to observe all requests and responses.

HTTP request and response to the ITM website

The screenshot shows the Network tab in the Chrome DevTools. The left pane displays the ITM website's homepage, featuring the SeoulTech logo, a banner with Korean text "차세대 글로벌 IT융합기술 리더 양성" and English text "Lead the way of future", and a "공지사항" (Notice) section. The right pane shows the Network timeline and a detailed list of resources. The timeline at the top shows various requests starting from 2.000 ms and ending at 12.000 ms. Below the timeline is a table listing 21 resources, each with columns for Name, Status, Type, Initiator, Size, and T. The resources include CSS files like main.css, jquery.smartPop.css, ui.all.css, etc., and JS files like jquery-1.8.3.js, jquery.smartPop.js, ui.datepicker.js, etc. Most resources have a status of 200 and are of type document or stylesheet.

Name	Status	Type	Initiator	Size	T.
itm.seoultech.ac.kr	200	document	Other	50.1 kB	4.
main.css	200	stylesheet	(index):28	(memory ca... 0.	0.
jquery.smartPop.css	200	stylesheet	(index):32	(memory ca... 0.	0.
ui.all.css	200	stylesheet	(index):33	(memory ca... 0.	0.
jquery.fancybox.css	200	stylesheet	(index):34	(memory ca... 0.	0.
jquery-1.8.3.js	200	script	(index):36	(memory ca... 0.	0.
jquery.smartPop.js	200	script	(index):37	(memory ca... 0.	0.
ui.datepicker.js	200	script	(index):38	(memory ca... 0.	0.
ui.datepicker-ko.js	200	script	(index):39	(memory ca... 0.	0.
slides.min.jquery.js	200	script	(index):40	(memory ca... 0.	0.
jquery.fancybox.pack.js	200	script	(index):41	(memory ca... 0.	0.
common.js	200	script	(index):42	(memory ca... 0.	0.
slick.js	200	script	(index):44	(memory ca... 0.	0.
slick.css	200	stylesheet	(index):45	(memory ca... 0.	0.
m_menu.png	200	png	(index):75	(memory ca... 0.	0.
logo.png	200	png	(index):76	(memory ca... 0.	0.

84 requests | 56.0 kB transferred | 30.2 MB resources | Finish: 11.18 s | DOMContentLoaded: 295 ms | Load: 624 ms

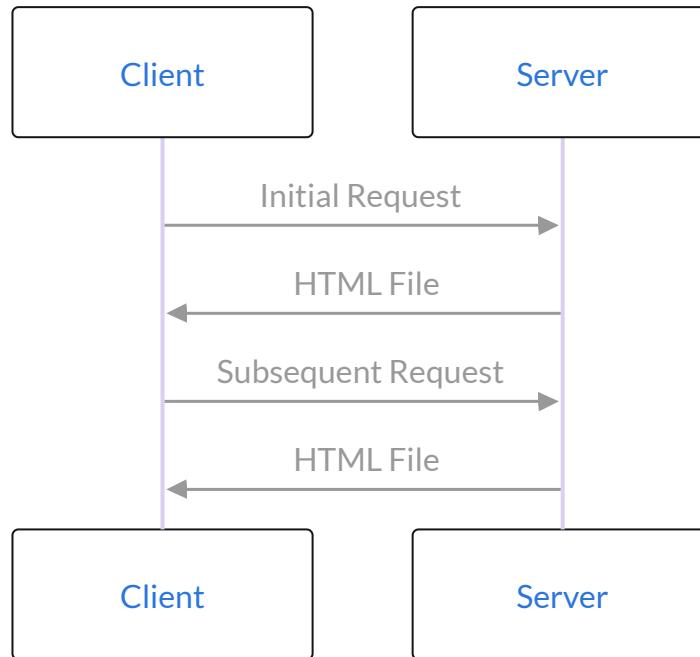
- Pay attention at the bottom part you can easily see that this page is sending 84 requests targeting different servers to render the page.
 - Key resources include: CSS files, favicons, JS files, images, videos and raw data

Server-side rendering (SSR)

- Early web apps were simple and JavaScript usage was limited.
- Pages were rendered on the **server**, and the client only received HTML, CSS, and JS.
- This approach is called **server-side rendering** and is still used today.

Disadvantages:

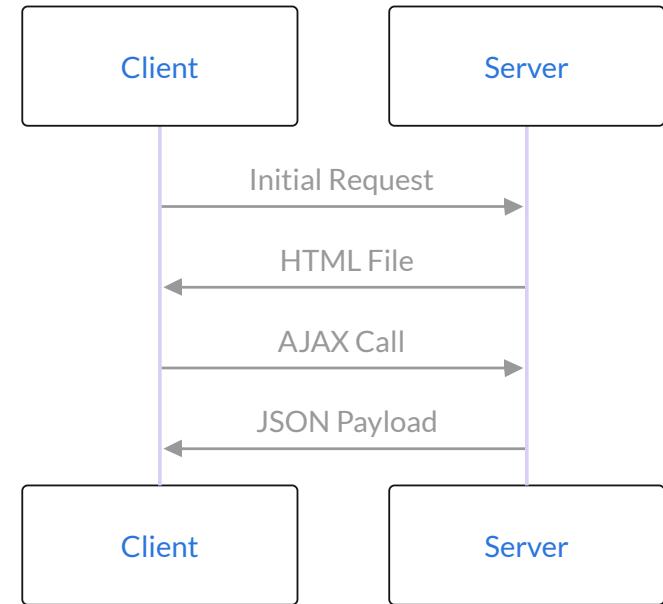
- Every user interaction requires the server to re-render and resend the page.
- Generates high network traffic.
- Users experience blank screens during refreshes.
- Especially poor on early smartphones (low processing power, slow connections).



The relationship between the server and client in the server-side rendering approach

Client-side Rendering (CSR)

- In client-side rendering, the server sends the initial HTML, CSS, and JS files to the client.
- JavaScript then takes control and renders the views on the client side.
- The server only sends data, and the client renders the page
 - **Single-Page Application (SPA)** is the most common pattern today (it is not absolutely required to be CSR)
- Initially complex to implement, but frameworks like **Angular**, **React**, and **Vue.js** have made SPA development popular.
- The **SPA pattern** still uses HTTP, but via **Asynchronous JavaScript and XML (AJAX)** requests.
- Backend applications have evolved into **APIs** that serve data to clients—and even other servers.



The relationship between the server in the AJAX approach

Mastering HTTP

$C \rightarrow S$

$S \rightarrow C$

- You may be surprised to learn that HTTP **requests** and **responses** are largely **human-readable plaintext**.
- Currently, the most used version of the protocol is the HTTP/1.1 version, but the HTTP/2 version is gaining popularity
- Let's take a deeper look at the different parts that compose the request and the response:
 1. headers
 2. methods
 3. payloads

HTTP headers (request)

- Each request and response has a set of headers.
- These are key-value pairs and provide additional information about the request or the response.

- **Representation headers**

- `content-type`, `content-length`

- **General headers**

- `keep-alive`, `upgrade-insecure-requests`

- **Request headers**

- `accept`, `accept-encoding`, `accept-language`, `host` and `user-agent`

- We can understand many things about a request, such as the type of content the client is expecting, the language, and the browser used

POST / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh; ...)... Firefox/51.0
Accept: text/html,application/xhtml+xml,...,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-12656974
Content-Length: 345

-12656974
(more data)

Request headers: Host, User-Agent, Accept, Accept-Language, Accept-Encoding, Connection, Upgrade-Insecure-Requests, Content-Type, Content-Length

General headers: Content-Type, Content-Length

Representation headers: none

HTTP headers (response)

- Representation headers

- `content-type`, `content-encoding`, `last-modified`

- General headers

- `connection`, `date`, `keep-alive`, `transfer-encoding`

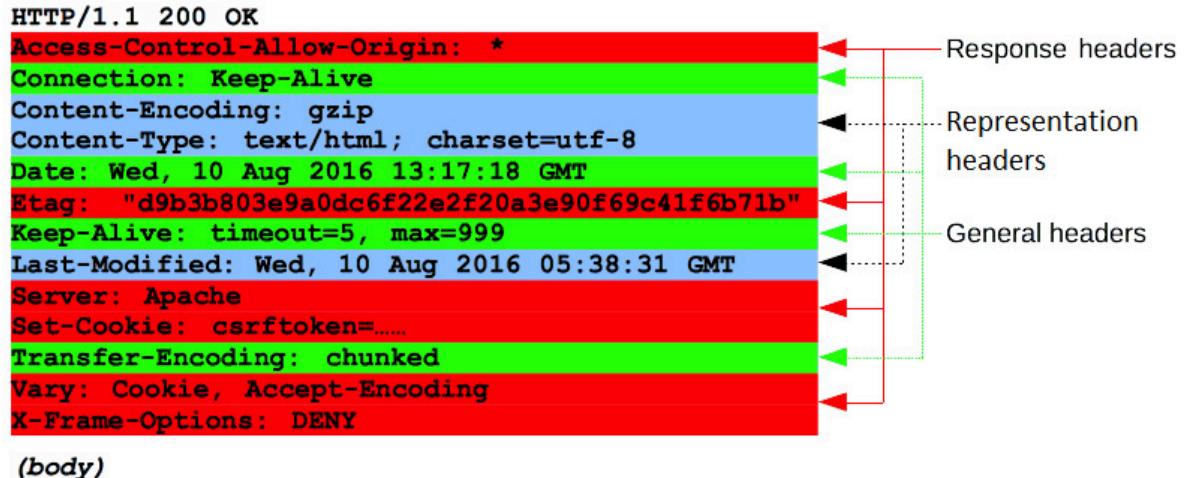
- Response-specific headers

- `access-control-allow-origin`, `etag`, `server`, `set-cookie`, `vary`, `x-frame-options`

- Provide metadata to help browsers and apps digest and render responses

- Critical for security:

- `X-Frame-Options` prevents framing
 - `Feature-Policy` can disable camera/microphone, etc.



HTTP status codes

- With all responses that a web server sends, there is a response code that allows us to understand whether the request was successful or not and can even provide more detailed feedback.

The most common status codes are:

- 200 OK**
- 201 Created -> The request succeeded, and a new resource was created as a result.
- 301 Moved Permanently -> The URL of the requested resource has been changed permanently. The new URL is given in the response.
- 400 Bad Request -> The server cannot or will not process the request due to something that is perceived to be a client error
- 401 Unauthorized -> semantically this response means “unauthenticated”
- 403 Forbidden -> The client does not have access rights to the content; that is, it is unauthorized
- 404 Not Found**
- 429 Too Many Requests
- 500 Internal Server Error -> The server has encountered a situation it does not know how to handle.
- 503 Service Unavailable.



Status Code	Description
1xx	informational
2xx	Successful
3xx	Redirection
4xx	Client error
5xx	Server error

- Error 418 and some fun hacker and open source culture

HTTP methods

- HTTP defines a set of **request methods** to indicate the purpose of the request and what is expected if the request is successful.
- Although they can also be nouns, these request methods are sometimes referred to as HTTP verbs.
- GET: Retrieve a resource (default method used by the browser)
- POST: Create a resource
- PUT: Update a resource
- PATCH: Partially update a resource
- DELETE: Delete a resource

★ CRUD

Method	Action
POST or PUT	Create
GET	Read
PUT	Update
DELETE	Delete

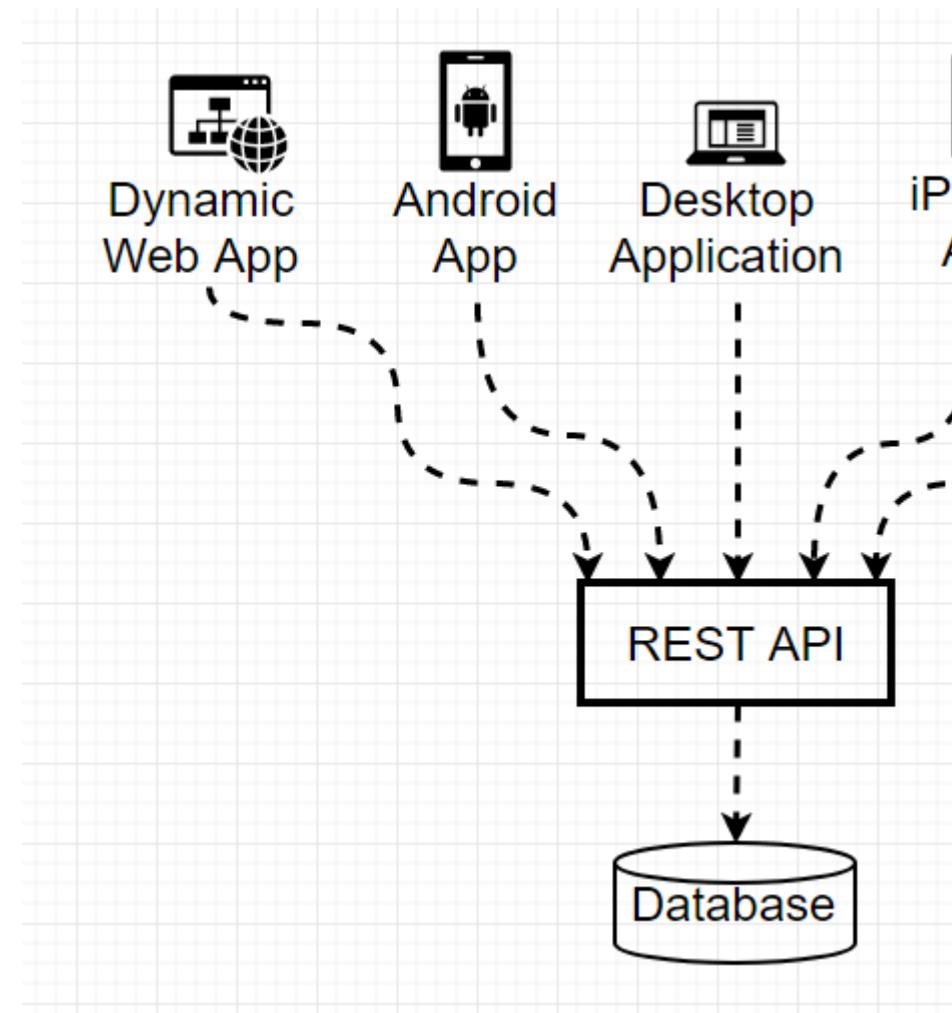
(CRUD) are the four basic operations or actions of persistent storage.

HTTP payloads

- HTTP messages can carry a payload, which means that we can send data to the server, and servers likewise can send data to their clients.
- This is often done with **POST requests**.
- Payloads can be in many formats, but the most common are the following:
 - **application/json**: Used when sharing JSON data
 - **application/x-www-form-urlencoded**: Used when sending simple texts in ASCII, sending data in the URL
 - **multipart/form-data**: Used when sending binary data (such as files) or non-ASCII texts
 - **text/plain**: Used when sending plain text, such as a log file

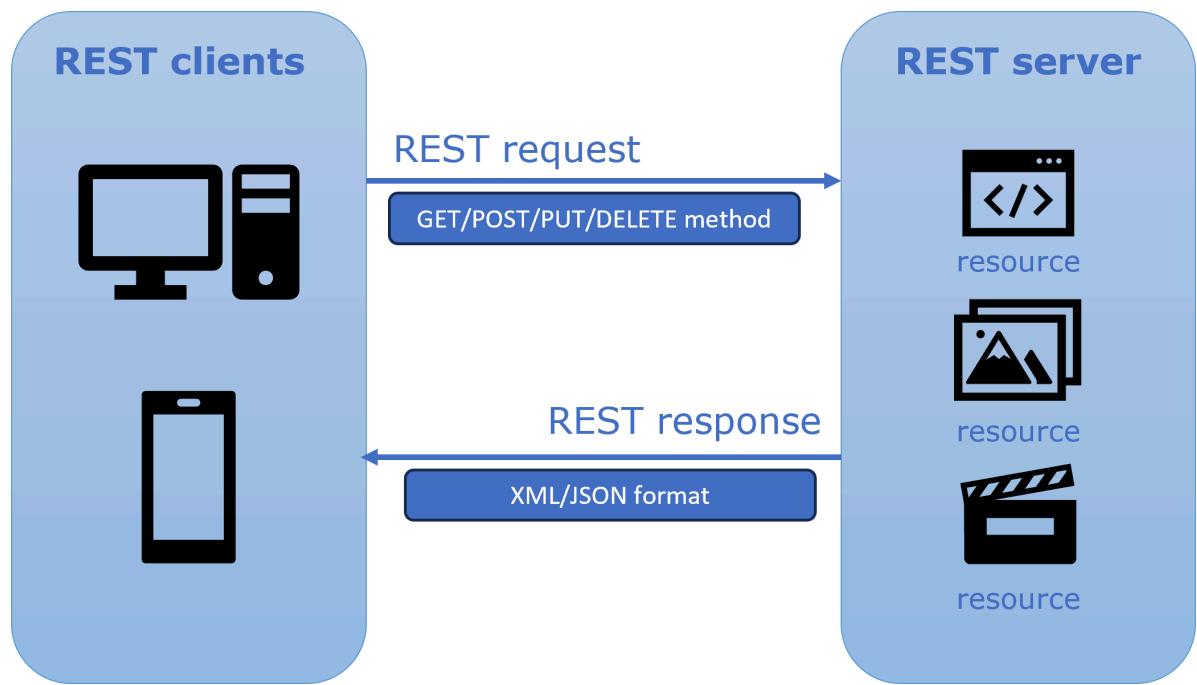
Application Program Interface (API)

- An application programming interface (API) is a connection between computers or between computer programs.
- A web API is an API for either a web server or a web browser.
- A **client-side web API** is a programmatic interface to extend functionality within a web browser or other HTTP client.
- A **server-side web API** consists of one or more publicly exposed endpoints to a defined request–response message system, typically expressed in **JSON** or **XML** by means of an HTTP-based web server.
- APIs come in all shapes and sizes:
 - REST (what we'll focus on)
 - SOAP
 - RPC/gRPC
 - GraphQL



REST APIs

- REST stands for Representational State Transfer
- Architectural style for designing web APIs
- Coined by Roy Fielding in his [2000 PhD dissertation](#)
- **Resources**
 - Identified by unique URLs, that denote endpoints
 - Accessed and manipulated via HTTP methods
- **Server Response**
 - Status code (e.g., 200, 404, 500)
 - Payload (JSON, XML, etc.) when applicable



Example of REST API

- Let's say that we have a REST API to manage a database of movies.
- We can define the following resources:
 - /movies: This resource represents the collection of movies
 - /movies/:id: This resource represents a single movie
- The client can perform the following operations on these resources using the aforementioned HTTP methods:
 - GET /movies: Get all the movies
 - GET /movies/:id: Get a single movie
 - POST /movies: Create a new movie
 - PUT /movies/:id: Update a movie
 - DELETE /movies/:id: Delete a movie
- Most of the time, the server will respond with a JSON payload, but it can also respond with other formats such as XML or HTML.

JavaScript Object Notation (JSON)

- Data format that represents data as a set of JavaScript objects
- Natively supported by all modern browsers (and libraries to support it in old ones)
- ~~Not yet as popular as XML, but steadily rising due to its simplicity and ease of use~~
- **Updated:** JSON is a lightweight substitute for XML and it is more popular than XML because of JavaScript's dominance as the most widely used language of today.

JavaScript objects

```
1 let person = {  
2     name: "Philip J. Fry",           // string  
3     age: 23,                      // number  
4     "weight": 172.5,               // number  
5     friends: ["Farnsworth", "Hermes", "Zoidberg"], // array  
6 };  
7  
8 person.age;  
9 person["weight"]; // 172.5  
10 person.friends[2]; // "Zoidberg"  
11 let propertyName = "name";  
12 console.log(person[propertyName]); // "Philip J. Fry"
```

- Objects are the most versatile structure in JS
- You can access the fields with dot notation (.fieldName) or bracket notation ([“fieldName”]) syntax
- You can create a new property or overwrite existing ones in an object by assigning a value

Example of JSON

```
1 {  
2   "name": "Philip J. Fry",  
3   "age": 23,  
4   "weight": 172.5,  
5   "friends": [  
6     "Farnsworth",  
7     "Hermes",  
8     "Zoidberg"  
9   ]  
10 }
```

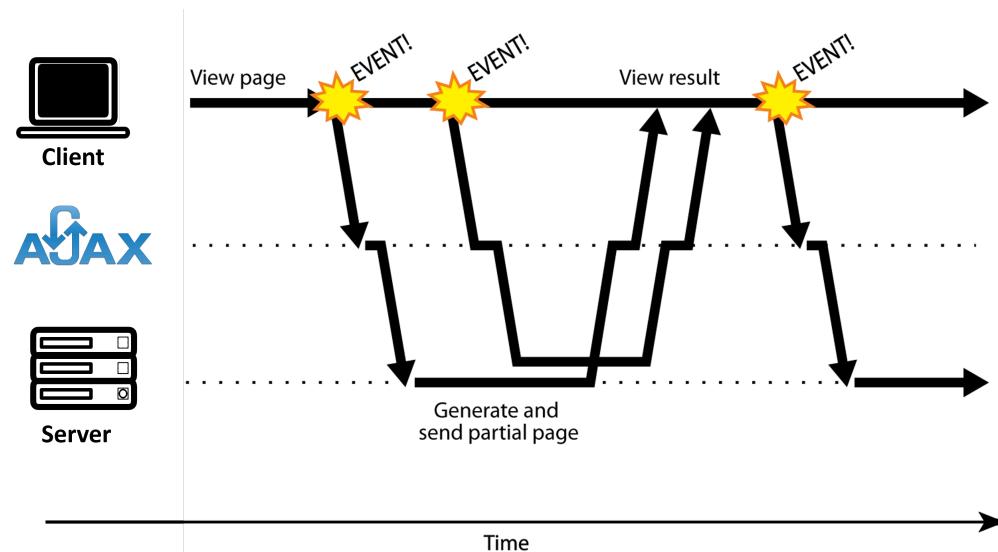
- Keys (attributes) must be in double quotes:
 - name: "Fry" → Invalid
 - "name": "Fry" → Valid
- Strings must use double quotes (not single quotes):
 - 'Fry' → Invalid
 - "Fry" → Valid
- No functions, comments, or trailing commas allowed:
 - JSON is pure data – not code.
 - You can't include `function() {}` or comments like `// this is a comment.`

JavaScript Objects vs. JSON

- JSON is a way of representing objects, or structured data.
 - The technical term is “serializing” which is just a fancy word for turning an object into a savable string of characters $\text{JS Object} \rightarrow \text{String (JSON)}$
- Browser JSON methods:
 - `JSON.parse(/* JSON string */)` – converts JSON string into Javascript object
 - `JSON.stringify(/* Javascript Object */)` – converts a Javascript object into JSON text

AJAX

- Web application: a dynamic web site that mimics the feel of a desktop app presents a continuous user experience rather than disjoint pages
 - Examples: Gmail, Facebook, etc.
- AJAX: Asynchronous JavaScript and XML
 - It is not a technology, but rather a programming pattern
 - Downloads data from a server in the background
 - Allows dynamically updating a page without making the user wait
 - Avoids the “click-wait-refresh” pattern
 - Currently it should be called “AJAJ”??



AJAX

- **XMLHttpRequest**(callback-based)
 - XMLHttpRequest (XHR) objects are used to interact with servers. You can retrieve data from a URL without having to do a full-page refresh.
 - This enables a Web page to update just part of a page without disrupting what the user is doing.
- **Fetch API** (promise-based)
 - It provides a global fetch() method that provides an easy, logical way to fetch resources asynchronously across the network.
 - Unlike XMLHttpRequest that is a callback-based API, Fetch is promise-based and provides a better alternative

AJAX with Fetch

- The Fetch API provides a JavaScript interface for making HTTP requests and processing the responses.
- `fetch` takes a URL string as a parameter to request data (e.g. pokemon information in JSON format) that we can work with in our JS file.

```
1 async function populateInfo() {  
2   const URL = "https://pokeapi.co/api/v2/pokemon/ditto";  
3   await fetch(URL) // returns a Promise!  
4   ...  
5 }
```

- We need to do something with the data that comes back from the server.
- But we don't know how long that will take or if it even will come back correctly!
- The `fetch` call returns a `Promise` object which will help us with this uncertainty.

More on `fetch()`

- The `fetch()` function returns a Promise
- When resolved, the value of this Promise is the Response object representing the response from the API
- It contains a lot of information (more than just the pure information we want to get back)
- The information is critical to our implementation

Checking the http status with `statusCheck()`

- NOT (!!?) built into JavaScript (we need to define it)
- But why?? Wouldn't the promise return by fetch just reject if something fails??

```
1  async function statusCheck(res) {  
2      if (!res.ok) {  
3          throw new Error(await res.text());  
4      }  
5      return res;  
6  }
```

.json() and .text()

- Called on the Response object
- Both of these methods return Promises
- **.json():**
 - Returns a Promise that resolves to a JavaScript object
 - Parsing the contents of the Response object's body (received as JSON) into an object
- **.text():**
 - Returns a Promise that resolves to a String
 - Parsing the contents of the Response object's body (received as text) into a String

Generic fetch template

- Here's the general template you will use for most AJAX code:

```
1 const BASE_URL = "some_base_url.com"; // you may have more than one
2
3 async function makeRequest() {
4   let url = BASE_URL + "?query0=value0"; // some requests require parameters
5   try {
6     let result = await fetch(url)
7     await statusCheck(result)
8     // result = await result.text(); // use this if your data comes in text
9     // result = await result.json(); // or this if your data comes in JSON
10    processData(result)
11  } catch(err) {
12    handleError(err); // define a user-friendly error-message function
13  }
14}
15
16 function processData(responseData) {
17   // Do something with your responseData! (build DOM, display messages, etc.)
18 }
19
20 async function statusCheck(res) {
21   if (!res.ok) {
22     throw new Error(await res.text());
```

Next week

Backend development and Introduction to Node.js



Acknowledgements

- Some contents of this lecture are partially adapted from:
 - Harvard [CS50's Web Programming with Python and JavaScript](#), licensed under [CC BY-NC-SA 4.0](#).
 - Materials from University of Washington's [CSE 154 Web Programming](#) (used with permission).
 - [The Odin Project](#) (main website code under MIT license and curriculum licensed under a [CC BY-NC-SA 4.0](#))
 - [The Fundamentals of Web Application Development \(Web Edition\)](#) ©2025 Nicholas D. Freeman. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [freeCodeCamp.org](#) © 2025 freeCodeCamp.org. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [Node.js for Beginners](#) by Ulises Gascón. Some contents adapted under fair educational use for instructional purposes.

Web Programming

Lecture 9

*Web application architectures
and introduction to node.js*

Josue Obregon 

jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

► [Lecture slides index](#)

May 9, 2025

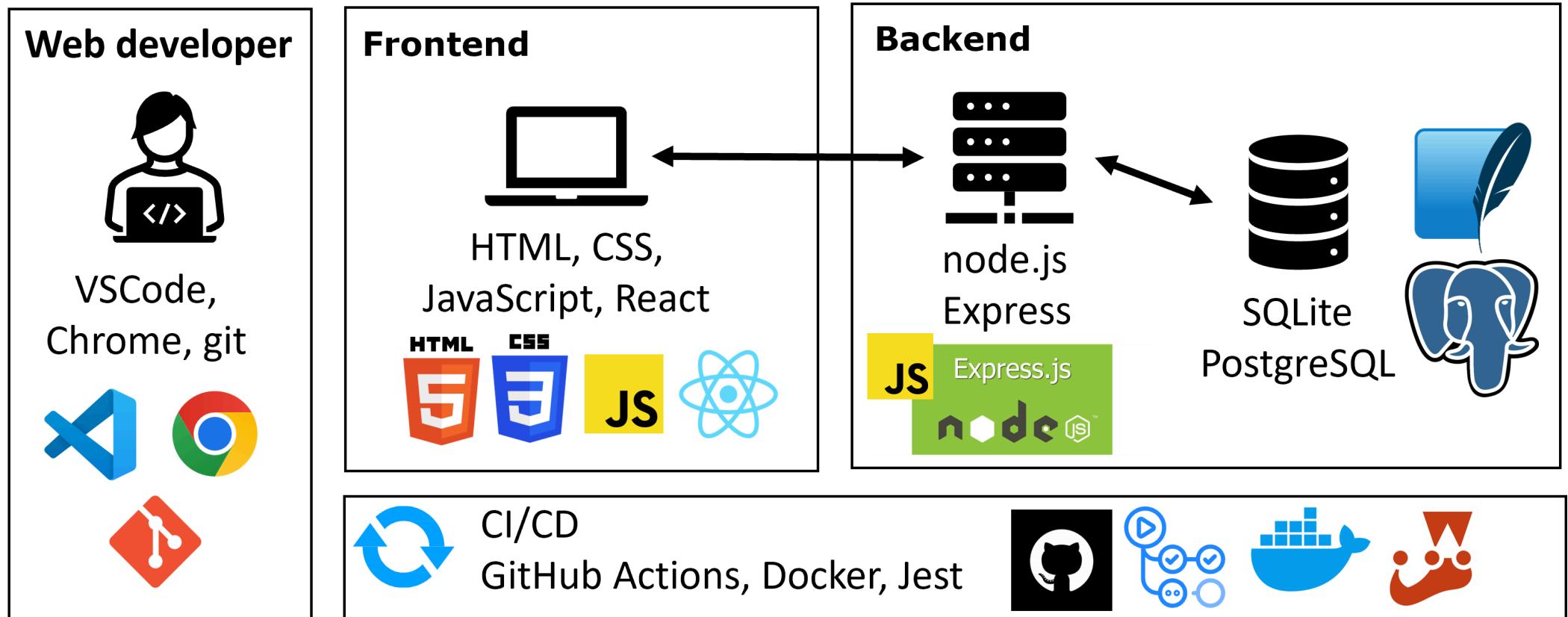


Agenda

- Web applications architecture
- Web service
- Server-side web frameworks
- Node.js
- Starting a simple Node.js project

Course structure

Modern Web Development Technology Stack



Roadmaps: **Frontend**, **backend** and **fullstack**

What we have learned so far

- Web page structure and appearance with HTML5 and CSS.
- Client-side interactivity with JS DOM and events.
- Using web services (APIs) as a client with JS.
- Writing JSON-based web services with a server-side language.

Core layers of a web application

- Presentation Layer (Frontend)
 - What users see and interact with (UI)
 - Built with HTML, CSS, JavaScript (or frameworks like React, Angular)
- Application Layer (Backend / Business Logic)
 - Processes user requests, applies business rules
 - Handles authentication, routing, and service orchestration
 - Many languages and framework available (Node.js, Django, Laravel, etc.)
- Data Layer (Database)
 - Stores, retrieves, and manages application data
 - Can be relational (MySQL, PostgreSQL) or NoSQL (MongoDB, Redis)

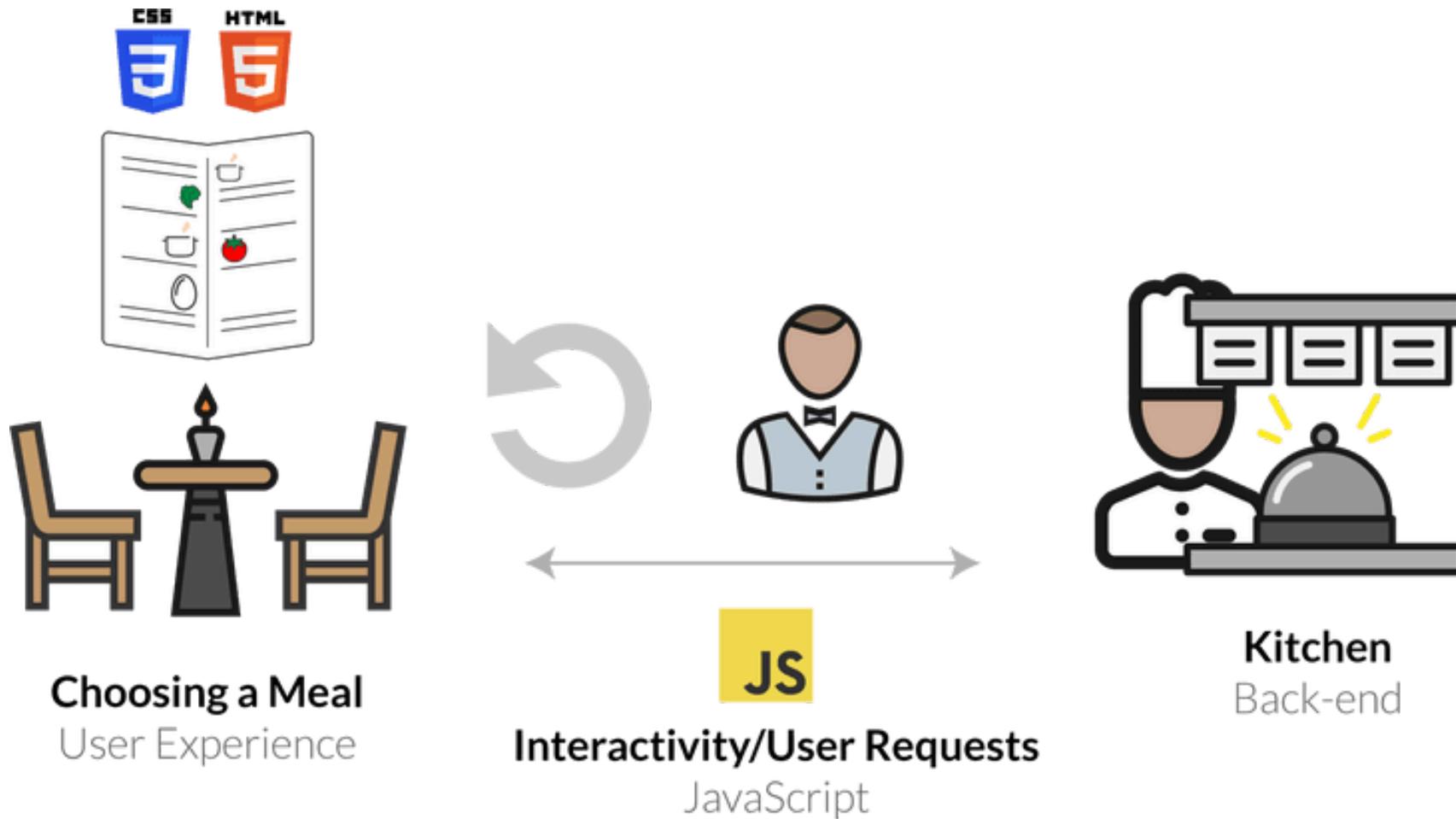
Web application architecture?

- The structure that defines how web components –frontend, backend, and database— interact to deliver a complete web application.
 - Ensure scalability, maintainability, and performance
 - Support modularity and evolution of applications
- Evolution of web application architectures
 - **Monolithic** (before 2000): Single unit of code, sharing same resources and memory.
 - **Tiers and Layers** (Around 2000s): Separates the application into different layers/tiers, with each layer performing a specific function (presentation, business, data) *Separation of concerns*
 - **Service Oriented Architecture** (Early 2000s): build distributed systems using web services. This architecture is based on the idea of breaking down an application into smaller, reusable services that can be combined to create more complex applications
 - **Microservices** (Mid 2010s): is an evolution of SOA that emerged in the mid-2010s. This architecture breaks down an application into a collection of small, independent services that communicate with each other using APIs. *no need to be same language or framework for communication*

Web service

- Web service definitions:
 - software functionality that can be invoked through the internet using common protocols.
 - a server running on a computer device, listening for requests at a particular port over a network, serving web documents
- Done by contacting a program on a web server
 - Web services can be written in a variety of languages
 - Many web services accept parameters and produce results
 - Clients contact the server through the browser using XML over HTTP and/or AJAX Fetch code
 - The service's output might be HTML but could be text, XML, JSON, or other content

How does a web service respond to requests?

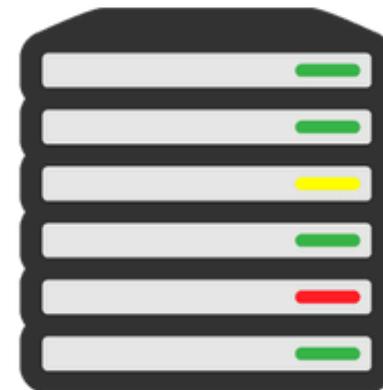


Why do we need a server to handle web service requests?

- Servers are dedicated computers for processing data efficiently and delegating requests sent from many clients (often at once).
- These tasks are not possible (or appropriate) in the client's browser.



Kitchen
Back-end



Server
Back-end

Server-side web frameworks

- Server-side web frameworks (a.k.a. “web application frameworks”) are software frameworks that make it easier to write, maintain and scale web applications by providing tools and libraries that simplify common web development tasks
 - **Work directly with HTTP requests and responses:** Handle HTTP requests and responses without dealing with low-level networking code.
 - **Route requests to the appropriate handler:** Map URL patterns to specific handler functions for organized and maintainable code.
 - **Make it easy to access data in the request:** Provide easy access to request data like URL parameters, POST data, cookies, and session info through built-in objects.
 - **Abstract and simplify database access:** Simplify database operations with built-in Object-Relational Mappers (ORMs).
 - **Rendering data:** Generate dynamic content using templates and easily render data as HTML, JSON, or XML.
 - **Supporting sessions and user authorization**

Popular server-side web frameworks

- Opinionated vs. Unopinionated:
 - Opinionated frameworks guide you with built-in best practices, while unopinionated ones give more flexibility in how you structure your code.
- Batteries Included vs. Lightweight:
 - Batteries included frameworks offer built-in tools for most tasks, while lightweight ones let you choose and add only what you need.
- Server-side applications are written using various programming languages:
 - JavaScript, Java, Ruby, Python, PHP among others

Language					
Framework		 Laravel		  FastAPI	express  front-end & back-end with React

Client-side JavaScript

- So far, we have used JS on the browser (client) to add interactivity to our web pages
- “Under the hood”, your browser requests the JS (and other files) from a URL resource, loads the text file of the JS, and interprets it realtime in order to define how the web page behaves.
- In Chrome, it does this using the **V8 JavaScript engine**, which is an open-source JS interpreter made by Google.
 - Other browsers have different JS engines (e.g. Firefox uses SpiderMonkey).
- Besides the standard JS language features, you also have access to the DOM when running JS on the browser - this includes the `window` and `document`

Server-side language: JavaScript **node**

→ run JS without browser

- Node.js is a lightweight and fast **runtime** based on the **V8 JavaScript engine** (same engine that powers Google Chrome and Microsoft Edge).
- **cross-platform**, which means that we can run it on any operating system and architecture available in the modern market
- Flourishing **package ecosystem**
- Designed for **efficient, asynchronous server-side programming**
- **Open-source** with a big and active developer community
 - The Node.js Foundation merged with the JS Foundation in 2019 to create the **OpenJS Foundation**
 - Google, IBM, Microsoft, Netflix, Red Hat, GitHub

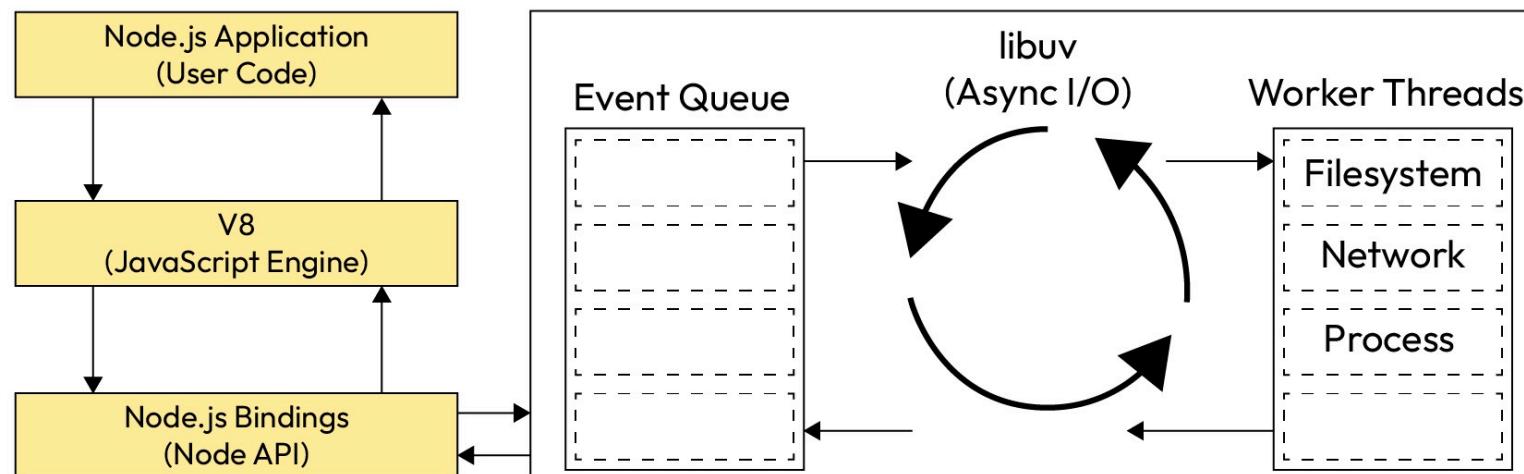


The Node.js single-thread architecture & Non-blocking I/O

- Uses a single-threaded, non-blocking I/O model for efficiency and scalability.
- Replaces traditional multi-threading with event-driven architecture using a central event loop.
- Built on V8 engine (executes JS) and libuv (handles async I/O via thread pool).
- Key components:
 - JavaScript code: Your app logic
 - V8: Runs your JS
 - Node APIs + C++ bindings: Interface between JS and system-level code
 - libuv: Manages async operations (file access, networking)

메인 스레드 (기본)

I/O 작업은 libuv가 관리하는 스레드 끝에 위임.
I/O 작업 완료 기다리지 않고 다음 작업 처리
I/O 작업 완료 시 callback 함수가 이벤트 큐로 전달.
메인 스레드 준비되면 callback 실행.



Node.js: Server-side JavaScript

- Node.js uses the same open-source V8 JavaScript engine as Chrome
- Node.js is a runtime environment for running JS programs using the same core language features, but outside of the browser.
- When using Node, you do not have access to the browser objects/functions (e.g. document, window, addEventListener, DOM nodes).
- Instead, you have access to functionality for managing HTTP requests, file I/O, and database interaction.
- This functionality is key to building REST APIs!

Client-side vs. Server-side JavaScript

Client-side

- Adheres (mostly) to the ECMAScript standards
- Parsed and run by a browser and therefore has access to document, window, and more
- Calls APIs using fetch, which sends HTTP requests to a server
- Runs only while the web page is open
- Handles “events” like users clicking on a button

Server-side

- Adheres (mostly) to the ECMAScript standards
- Parsed and run by Node.js and therefore has access to File I/O, databases, and more
- Handles requests from clients, sending HTTP responses back
- Runs as long as Node is running and listening
- Handles HTTP requests (GET/POST etc)

Getting started with Node.js

- When you have Node installed, you can run it immediately in the command line.

1. Start an interactive REPL with node (no arguments). This REPL is much like the Chrome browser's JS console tab.

- “REPL” stands for “Read Evaluate Print Loop”.
- It is a simple interactive computer programming environment that takes single user inputs, executes them, and returns the result to the user.
- REPL-style tools exist in most languages.

2. Execute a JS program in the current directory with node file.js

Starting a Node.js Project

- There are a few steps to starting a Node.js application, but luckily most projects will follow the same structure.
 - When using Node.js, you will mostly be using the command line
1. Start a new project directory (e.g. intro-node)
 2. Create your `app.js` file in your project directory
 3. Inside the directory, run `npm init` to initialize a `package.json` configuration file (you can keep pressing Enter to use defaults)
 4. Install any modules with `npm install <package-name>`
 5. Write your Node.js file! (e.g. `app.js`)
 6. Include any front-end files in a public directory within the project.
- Along the way, a tool called `npm` will help install and manage packages that are useful in your Node app.

Starting app.js

- Run `npm install express` to install the **Express.js** package
- Build your basic app with the following file
 - Check if your service is running correctly accessing this in your browser `http://localhost:8080/posts`

```
1 const express = require('express');
2 const app = express();
3
4 const PORT = 8080;
5
6 app.get('/posts', function(req, res) {
7   console.log("Endpoint /posts received a GET request.");
8   res.type("text").send("Hello World");
9 });
10
11 app.listen(8080, () => {
12   console.log(`Our first node app listening on port ${PORT}`);
13 })
```

Handwritten annotations on the code:

- Line 5: "request of client" is written above "req".
- Line 5: "response of server" is written above "res".
- Line 6: "endpoint" is written above the "/posts" path.
- Line 11: "Express app listens client's request at port 8080" is written below the line.

! Important

package.json: This file is primarily used for managing and documenting metadata about the project, including its name, version, author, dependencies, scripts, and other configuration details. It acts as a manifest for the project.

package-lock.json: This file is generated and updated automatically by `npm` when installing or updating packages. It is used to lock the exact versions of dependencies installed in the project, ensuring reproducibility and consistent installations across different environments. **No Touch !**

Node.js modules

- When you run a `.js` file using Node.js, you have access to default functions in JS (e.g. `console.log`)
- In order to get functionality like file I/O or handling network requests, you need to import that functionality from modules - this is similar to the `import` keyword you have used in Java or Python.
- In Node.js, you do this by using the `require()` function, passing the string name of the module you want to import.
- For example, the module we'll use to respond to HTTP requests in Node.js is called `express`. You can import it like this:

```
1 const express = require("express");
```

Quick reminder on `const` Keyword

- Using const to declare a variable inside of JS just means that you can never change what that variable references.
- We've used this to represent "program constants" indicated by ALL_UPPERCASE naming conventions
- For example, the following code would not work:

```
1 const specialNumber = 1;  
2 specialNumber = 2; // TypeError: Assignment to constant variable.
```

- When we store modules in Node programs, it is conventional to use const instead of let to avoid accidentally overwriting the module.
- Unlike the program constants we define with const (e.g. BASE_URL), we use camelCase naming instead of ALL_CAPS.

app.listen()

- To start the localhost server to run your Express app, you need to specify a port to listen to.
- The express app object has a function app.listen which takes a port number and optional callback function
- At the bottom of your `app.js`, add the following code - (process.env.PORT is needed to use the default port when hosted on an actual server)

```
1 // Allows us to change the port easily by setting an environment variable
2 const PORT = process.env.PORT || 8080;
3 app.listen(PORT);
```

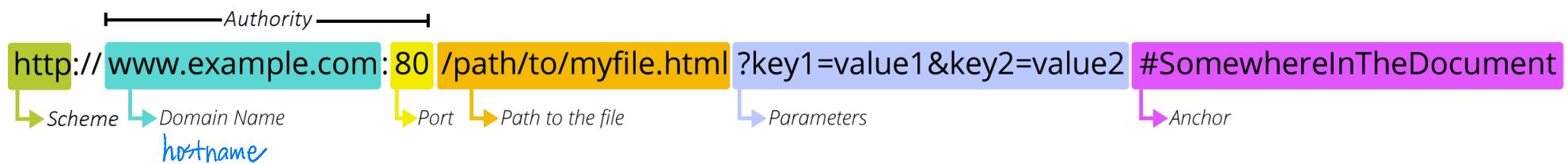
Note

localhost is a hostname that refers to the current computer used to access it. - A hostname is a label assigned to device connected to a computer network - The name localhost is reserved for loopback purposes

Basic Routing in Express

- Routes are used to define endpoints in your web service
- Express supports different HTTP requests - we will learn **GET** and **POST**
- Express will try to match routes in the order they are defined in your code
- **app.get** allows us to create a **GET endpoint**.
- It takes two arguments: **The endpoint URL path**, and a **callback function for modifying/sending the response**.

```
1 app.get(path, (req, res) => {  
2   ...  
3 });
```



Adding Routes in Express.js (I)

```
1 app.get(path, (req, res) => {  
2   ...  
3 });
```

- `req` is the request object, and holds items like the request parameters.
- `res` is the response object, and has methods to send data to the client.
- `res.set(...)` sets header data, like “content-type”.
 - Always set either “text/plain” or “application/json” with your response.
- `res.send(response)` returns the response as HTML text to the client.
- `res.json(response)` does the same, but with a JSON object.
- When adding a route to the path, you will retrieve information from the request, and send back a response using res (e.g. setting the status code, content-type, etc.)
- If the visited endpoint has no matching route in your Express app, the response will be a **404 (resource not found)**

Useful Request Properties/Methods

Name	Description
req.params	Endpoint “path” parameters from the request
req.query	Query parameters from the request

Useful Response Properties/Methods

Name	Description
res.write(data)	Writes data in the response without ending the communication
res.send()	Sends information back (default text with HTML content type)
res.json()	Sends information back as JSON content type
res.set()	Sets header information, such as “Content-type”
res.type()	A convenience function to set content type (e.g., "text" or "json")
res.status()	Sets the response status code
res.sendStatus()	Sets the response status code with the default status text

Setting the Content Type

- By default, the content type of a response is HTML - we will only be sending plain text or JSON responses though in our web services
- You can use res.type("text") and res.type("json") which is used to set response header content type information

```
1 app.get('/hello', function (req, res) {  
2   // res.set("Content-Type", "text/plain");  
3   res.type("text"); // same as above  
4   res.send('Hello World 2!');  
5 });
```

```
1 app.get('/hello-json', function (req, res) {  
2   // res.set("Content-Type", "application/json");  
3   res.type("json");  
4   res.send({ "msg" : "Hello world with json!" });  
5   // can also do res.json({ "msg" : "Hello world!"});  
6   // which also sets the content type to application/json  
7  
8 });
```

Request Parameters: Path Parameters

- Act as wildcards in routes, letting a user pass in “variables” to an endpoint
- Define a route parameter with :param

```
1 Route path: /majors/:major/courses/:course  
2 Request URL: http://localhost:8080/majors/ITM/courses/WebProg  
3 req.params: { "major": "ITM", "course": "WebProg" }
```

- These are attached to the request object and can be accessed with req.params

```
1 app.get("/majors/:major/courses/:course", function (req, res) {  
2     res.type("text");  
3     res.send("You sent a request for the major " + req.params.major +  
4             ", for the course " + req.params.course);  
5 });
```

Request Parameters: **Query Parameters**

- You can also use query parameters in Express using the req.query object, though they are more useful for optional parameters.

```
1 Route path: /courseInfo
2 Request URL: http://localhost:8080/courseInfo?major=ITM&course=WebProg
3 req.query: { "major": "ITM", "course": "WebProg" }
```

```
1 app.get("/courseInfo", function (req, res) {
2   let major = req.query.major; // ITM
3   let course = req.query.course; // WebProg
4   // do something with variables in the response
5 });
```

- Unlike path parameters, these are not included in the path string (which are matched using Express routes) and we can't be certain that the accessed query key exists.
- If the route requires the parameter but is missing, you should send an error to the client in the response.

Choosing Error Codes

- Use 400 (Invalid Requests) for client-specific errors.
 - Invalid parameter format (e.g. “Seattle” instead of a required 5-digit zipcode)
 - Missing a required query parameter
 - Requesting an item that doesn’t exist
- Use 500 (Server error) status codes for errors that are independent of any client input.
 - Errors caught in Node modules that are not related to any request parameters
 - SQL Database connection errors (next week!)
 - Other mysterious server errors...

Status Code	Description
1xx	informational
2xx	Successful
3xx	Redirection
4xx	Client error
5xx	Server error

Setting Errors

- The Response object has a status function which takes a status code as an argument.
- A helpful message should always be sent with the error.

```
1 app.get("/cityInfo", function (req, res) {  
2   let state = req.query.state;  
3   let city = req.query.city;  
4   if (!(state && city)) {  
5     res.status(400).send("Error: Missing required city and state query parameters.");  
6   } else {  
7     res.send("You sent a request for " + city + ", " + state);  
8   }  
9 });
```

Next week

- More on Node.js



Acknowledgements

- Some contents of this lecture are partially adapted from:
 - Harvard [CS50's Web Programming with Python and JavaScript](#), licensed under [CC BY-NC-SA 4.0](#).
 - Materials from University of Washington's [CSE 154 Web Programming](#) (used with permission).
 - [The Odin Project](#) (main website code under MIT license and curriculum licensed under a [CC BY-NC-SA 4.0](#))
 - [The Fundamentals of Web Application Development \(Web Edition\)](#) ©2025 Nicholas D. Freeman. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [freeCodeCamp.org](#) © 2025 freeCodeCamp.org. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [Node.js for Beginners](#) by Ulises Gascón. Some contents adapted under fair educational use for instructional purposes.

► Back to title slide

► Back to lecture slides index

Web Programming

Lecture 10

More on Node, POST and Full Stack Websites

Josue Obregon 

jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

 [Lecture slides index](#)

May 13, 2025

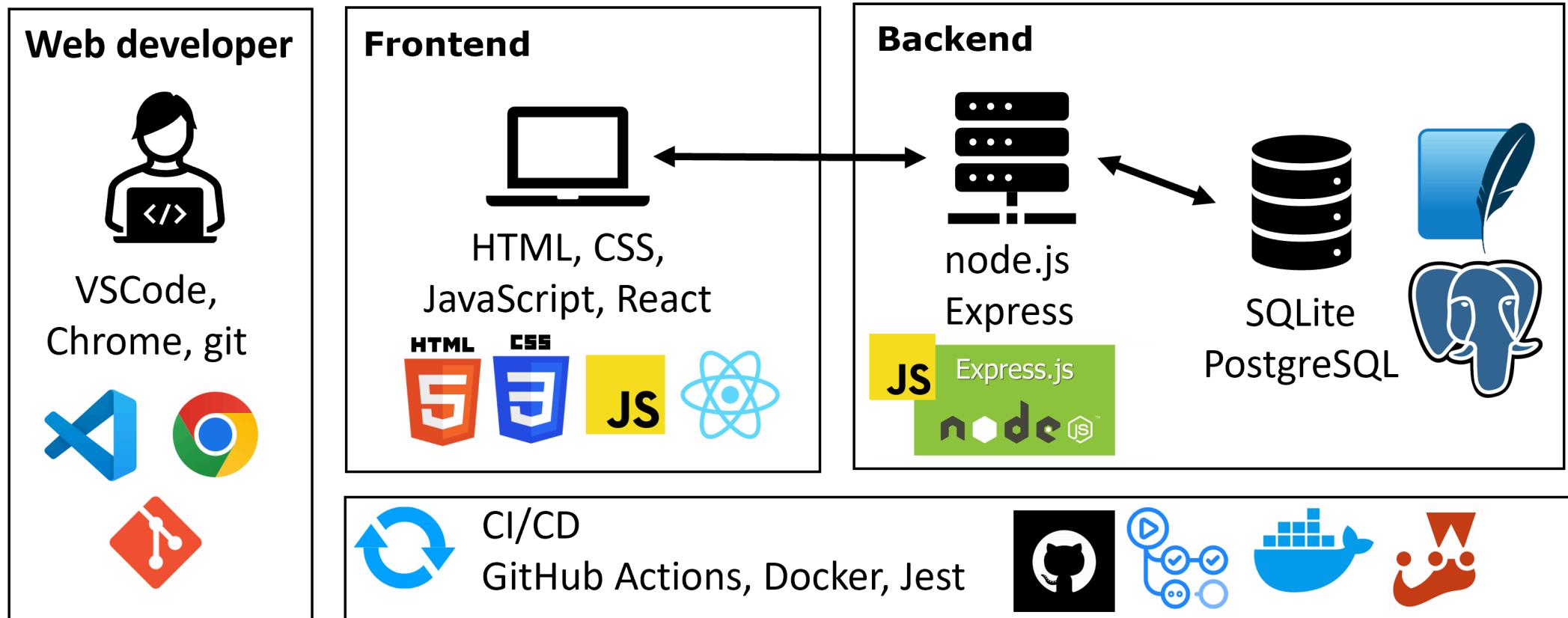


Agenda

- Node.js middleware pattern
- POST and forms
- Full Stack Node.js simple app
- File I/O

Course structure

Modern Web Development Technology Stack



Roadmaps: **Frontend**, **backend** and **fullstack**

Starting a Node.js Project

- There are a few steps to starting a Node.js application, but luckily most projects will follow the same structure.
 - When using Node.js, you will mostly be using the command line
1. Start a new project directory (e.g. intro-node)
 2. Create your `app.js` file in your project directory
 3. Inside the directory, run `npm init` to initialize a `package.json` configuration file (you can keep pressing Enter to use defaults)
 4. Install any modules with `npm install <package-name>`
 5. Write your Node.js file! (e.g. `app.js`)
 6. Include any front-end files in a public directory within the project.
- Along the way, a tool called **npm** will help install and manage packages that are useful in your Node app.

Starting app.js

- Run `npm install express` to install the **Express.js** package
- Build your basic app with the following file
 - Check if your service is running correctly accessing this in your browser `http://localhost:8080/posts`

```
1 const express = require('express');
2 const app = express();
3
4 const PORT = 8080;
5
6 app.get('/posts', function (req, res) {
7   console.log("Endpoint /posts received a GET request.");
8   res.type("text").send("Hello World");
9 });
10
11 app.listen(8080, () => {
12   console.log(`Our first node app listening on port ${PORT}`);
13 })
```

POST Parameters

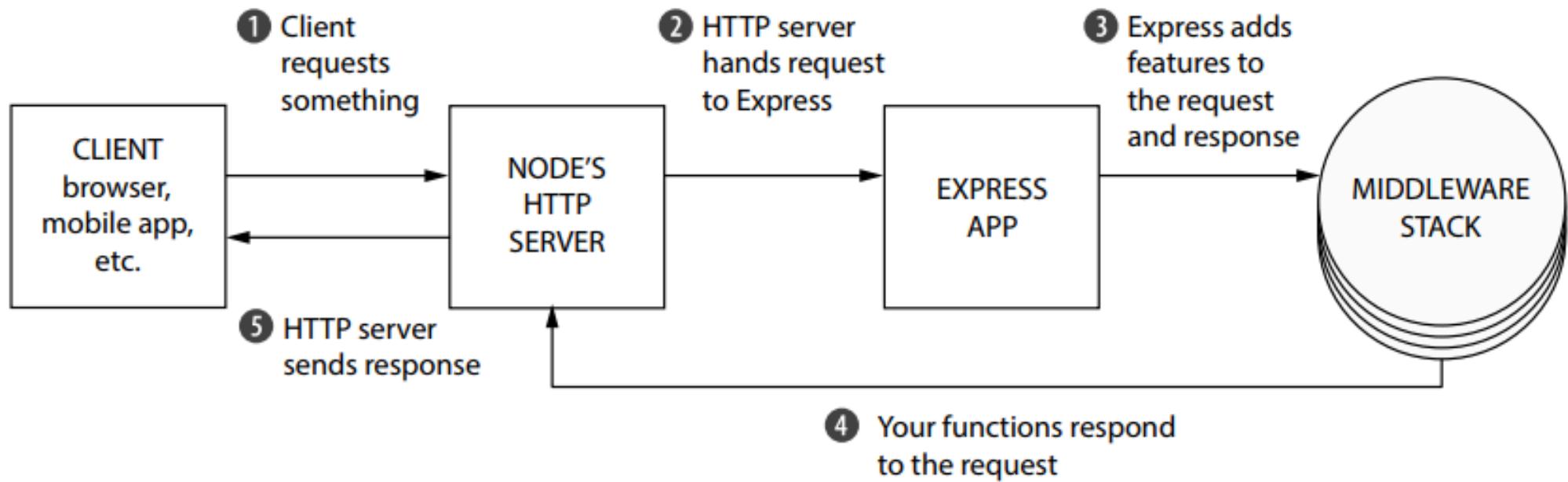
- Since POST parameters are sent through the body of the request, there is no guarantee that they are set, therefore, they must be checked before using them/assuming they exist

```
1 app.post("/contact", (req, res) => {
2   let name = req.body.name;
3   if (name) {
4     ...
5   } else {
6     ...
7   }
8});
```

Handling Different POST Requests

- POST requests can be sent with different data types:
 - application/x-www-form-urlencoded
 - application/json
 - multipart/form-data
- In Express, we use the middleware pattern to extract the POST parameters from the `req.body`.
- For the first two types, there is built-in middleware - we don't need middleware for text/plain.
- With forms and fetch, we use the `FormData` object to send POST parameters, which is always sent as multipart/form-data.
- There is no built-in middleware to access the `req.body` params for multipart content.

Middleware & Request/Response Pipeline



- The heart of Express is **the middleware pattern**, which allows you to extend the functionality of the framework by adding functions that will be executed in the request-response cycle.
- The middleware functions are executed in the order that they are added to the application,
- The middleware pattern as a pipeline, where the request is passed through the pipeline, and each middleware function can modify the request and the response, and pass the request to the next middleware function in the pipeline.

Full Stack Project Directory

- When creating full stack projects (both frontend and backend), your project directory should include all the static forward facing files in a folder named public and everything else at the root.

```
my-node-project/
  app.js
  package.json
  package-lock.json
  node_modules/
    public/
      index.html
      index.js
      index.css
```

Connecting a frontend to a backend

- These three lines of code allow your code to run on a specified port and look into a specific directory for files to serve to the user.
- They must be included at the bottom of your app.js file.

```
1 // tells the code to serve static files in a directory called 'public'  
2 app.use(express.static('public'));  
3  
4 // specify the port to listen on  
5 const PORT = 8080;  
6  
7 // tells the application to run on the specified port  
8 app.listen(PORT);
```

L for client. w/o routing .

Connecting a frontend to a backend

Client side

```
1 fetch('/hi/bob?age=20')
2     .then(statusCheck)
3     .then(res => res.text())
4     .then(handleResponse)
5     .catch(handleError)
6
7 let data = {'id': 1234};
8 fetch('/add',{
9   method: 'POST',
10  headers:{'Content-Type':'application/json'},
11  body: JSON.stringify(payload)})
12  .then(statusCheck)
13  ...
```

Server side

```
1 app.get('/hi/:name', (rq, rs) => {
2   let name = rq.params.name;
3   let age = rq.query.age;
4   ...
5 }
6
7 app.post('/add', (rq, rs) => {
8   let name = rq.body.ssn;
9   ...
10 ...})
```

- This requires you have your directory structure and `app.js` file configured properly AND that you are accessing your HTML document through localhost

Supporting incoming GET and POST requests

- **GET** requests
 - Information sent **through URL**
 - Retrieving non sensitive information

```
1 // GET endpoint
2 app.get('/path/:key', (rq, rs) => {
3     let ex1 = rq.params.key
4     let ex2 = rq.query.key
5 }
```

- **POST** requests
 - Informations sent **through the body**
 - **Sensitive** information and/or updating state

```
1 // POST endpoint
2 app.post('/path', (rq, rs) => {
3     let ex1 = rq.body.key
4 }
```

Full Stack Books app

- Simple book registry (title & author)
- RESTful API endpoints (GET / POST)
- Data stored in-memory
- Static HTML pages consuming API via `fetch()`



In-memory API setup

- Initialize project: npm init -y
- Install deps: express, body-parser, cors
 - body-parser
 - bodyParser.json() → parses JSON request bodies into req.body
 - bodyParser.urlencoded({ extended: false }) → parses HTML form data
 - cors
 - Enables Cross-Origin Resource Sharing
 - Allows browser `fetch()` from different origins ⇒ *client 7000 ↗ allow!*
server 8080 ↘
 - Usage: `app.use(cors());`
- Create app.js and require modules
- Define const books = []

Define Endpoints

- GET /api/books → res.json(books)
- POST /api/books → validate body → books.push(...) → res.status(201).json(...)
- Start server: app.listen(PORT)

```
1 // endpoint for getting a list of books
2 app.get('/api/books', (req, res) => {
3     res.json(books); originally
4 });
5
6 // endpoint for adding a book
7 app.post('/api/books', (req, res) => {
8     const { title, author } = req.body;
9     if (!title || !author) {
10         console.log("Error 400. Title and author required");
11         return res.status(400).json({ error: 'Title and author required' });
12     }
13     const newBook = { id: books.length + 1, title, author };
14     books.push(newBook); == this goes first
15     console.log("Code 201. Book created successful");
16     res.status(201).json(newBook); // HTTP 201 Created successful response status code
17 });
```

Testing the API via Browser

```
1 // index.html
2 fetch('/api/books')
3   .then(res => res.json())
4   .then(json => {
5     const ul = document.getElementById('list');
6     json.forEach(book => {
7       const li = document.createElement('li');
8       li.textContent = `${book.title} by ${book.author}`;
9       ul.appendChild(li);
10    });
11  });
12 
```

```
1 //add.html
2 document.getElementById('addBook').onsubmit = e => {
3   e.preventDefault();
4   const form = e.target;
5   const payload = {
6     title: form.title.value,
7     author: form.author.value
8   };
9   fetch('/api/books', {
10     method: 'POST',
11     headers: { 'Content-Type': 'application/json' },
12     body: JSON.stringify(payload)
13   }).then(() => location.href = 'index.html');
14 };
```

- Create a `public/` folder in the root of the project
- Serve it: `app.use(express.static('public'));`
- `index.html` `<script>` uses `fetch('/api/books')` to list books
- `add.html`
 - `<form>` + JS `fetch()` POSTs to `/api/books`
- Open `http://localhost:8080/index.html` to verify
- Create `public/styles.css`

File I/O in Node.js

- Unlike the browser, we have access to the file system when running Node.js
- We can read all kinds of files, as well as write new files
- For this, we use the `fs` core module
- We saw express as our first module in Node.js to create an API
 - And body-parser to parse POST request bodies
- Another useful module: fs (file system)
- This is a “Core” Module, meaning we don’t have to npm install anything.
built-in
- There are many functions in the fs module (with excellent documentation)
- Most functions rely on error-first callbacks, but we’re going to use the Promise versions

Reading Files and Writing Files

- **fs.readFile(fileName, encodingType)**
 - fileName: (string) file name
 - encodingType: file encoding (usually “utf8”)
 - Returns: Promise with a value holding the file’s contents
- **fs.writeFile(fileName, contents)**
 - fileName: (string) file name
 - contents: (string) contents to write to file
 - Returns: Promise without any resolved arguments

Reading Files and Writing Files with callbacks

- To read and write files, we first need the FileSystem package `const fs = require('fs');`

Write file

```
1 fs.writeFile('hola.txt', 'Hola mundo', (err) => {
2     if (err) {
3         console.log(`Error: ${err}`);
4     } else {
5         console.log('File successfully written!');
6     }
7});
```

Read file

```
1 fs.readFile('hola.txt', 'utf8', (err, contents) => {
2     if (err) {
3         console.log(`Error: ${err}`);
4     } else {
5         console.log(contents);
6     }
7});
```

Reading Files and Writing Files with Promises

- We can ‘promisify’ the readFile function so that it returns a promise instead of taking a callback.
- We use the promises functions of fs `fs.promises`

Using `.then/.catch`

```
1 let contents = fs.promises.readFile('hola.txt', 'utf8');
2 contents
3     .then(console.log)
4     .then(() => { console.log('Done reading file!'); })
5     .catch(err => console.log(`Error: ${err}`));
```

Using `async/await`

```
1 try {
2     let contents = await fs.promises.readFile('hola.txt', 'utf8');
3     console.log(contents);
4     console.log('Done reading file!');
5 }
6 catch (err) {
7     console.log(`Error: ${err}`)
8 }
```

Parsing JSON file contents

- You can read any file, including JSON.
- To parse JSON file contents and use as a JS object, use `JSON.parse`

```
1 let data = await fs.promises.readFile(filePath, 'utf8');  
2 data = JSON.parse(data);
```

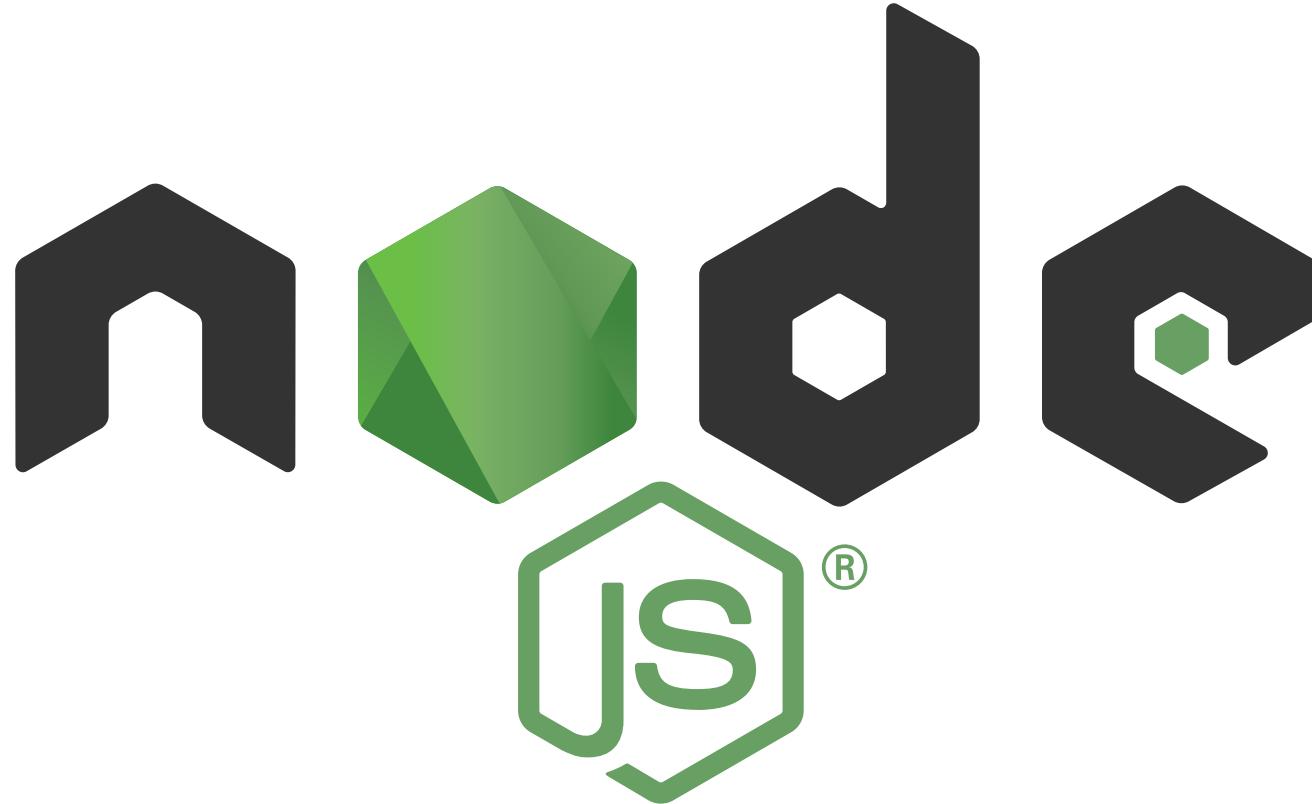
Writing and Reading JSON

- JSON files are text representations of JS objects. When you read from them, you will get a big string.
- When you write to them, you need to write a string.
 - `JSON.parse(jsonString)` to turn a JSON string into an object
 - `JSON.stringify(jsonObj)` to turn a JSON object into a string

```
1 // Read JSON file and return parsed object
2 async function readJSON(filePath) {
3     const data = await fs.promises.readFile(filePath, 'utf8');
4     return JSON.parse(data); String → object
5 }
6
7 // Write object to JSON file
8 async function writeJSON(filePath, jsonObject) {
9     const data = JSON.stringify(jsonObject); object → String
10    await fs.promises.writeFile(filePath, data);
11 }
```

Next week

- Data Persistence in Node.js



Acknowledgements

- Some contents of this lecture are partially adapted from:
 - Harvard [CS50's Web Programming with Python and JavaScript](#), licensed under [CC BY-NC-SA 4.0](#).
 - Materials from University of Washington's [CSE 154 Web Programming](#) (used with permission).
 - [The Odin Project](#) (main website code under MIT license and curriculum licensed under a [CC BY-NC-SA 4.0](#))
 - [The Fundamentals of Web Application Development \(Web Edition\)](#) ©2025 Nicholas D. Freeman. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [freeCodeCamp.org](#) © 2025 freeCodeCamp.org. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [Node.js for Beginners](#) by Ulises Gascón. Some contents adapted under fair educational use for instructional purposes.

Web Programming

Lecture 11

Data persistence in web applications with Node.js

Josue Obregon 

jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

 [Lecture slides index](#)

May 20, 2025

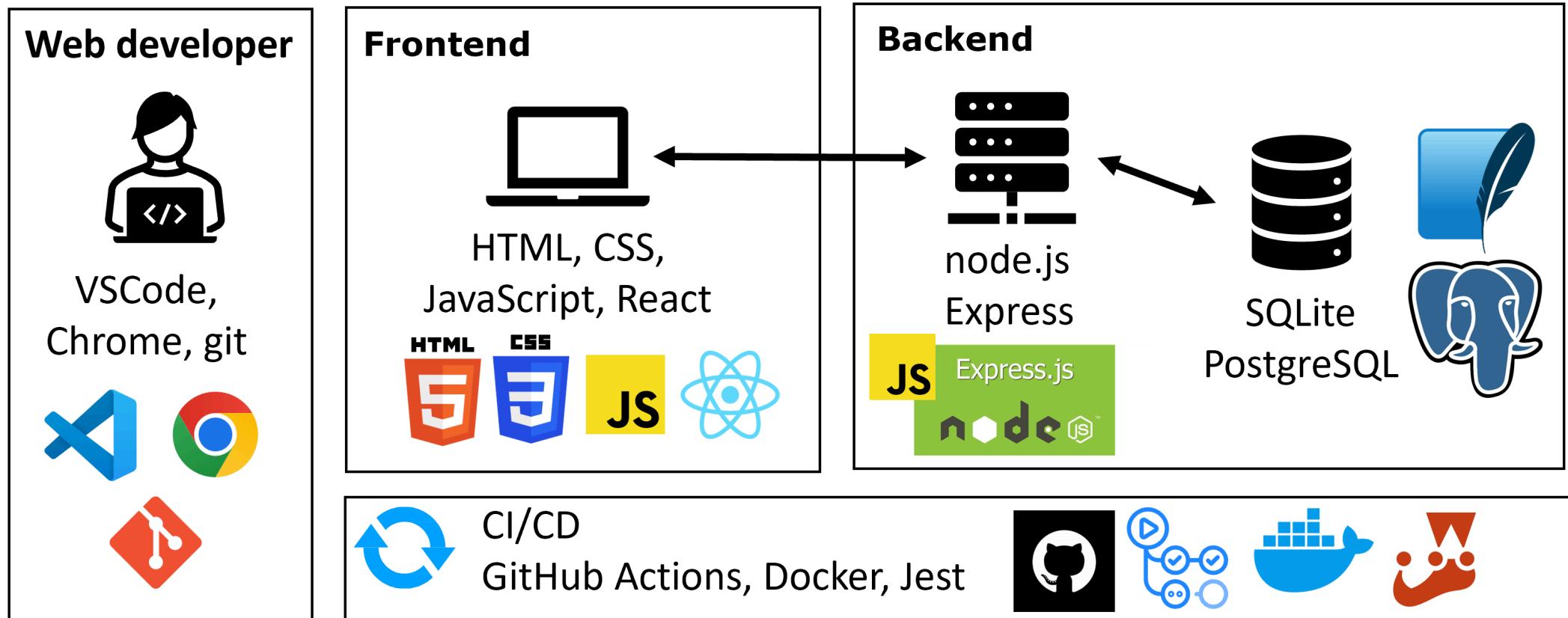


Agenda

- SQL basics
- sqlite module for Node.js

Course structure

Modern Web Development Technology Stack



Roadmaps: **Frontend**, **backend** and **fullstack**

Persistence of data

- **Persistence** is “*the continuance of an effect after its cause is removed.*”
- Persistent data is stored on a long-lasting storage medium, and data remains intact after modifications or changes to the storage medium.
- This is the important distinction between persistent data and non-persistent data: the data *survives after the process ends.*
- In other words, for data to be stored persistently, it must *write to non-volatile storage.*
- Data persistence ensures that **valuable business information remains accessible in a database and consistent across sessions, devices, and applications.**
- Data stored could be:
 - transactional data (dynamic data) on ecommerce platforms
 - customer records (sensitive data)
 - user-generated content
- The point of saving data in a non-volatile storage system is to make sure it can be reliably retrieved later.

Is File-Processing the Best Way?

- We have seen how to store JSON data in a file
- On the server, we have a lot of functionality to access the file system
- JSON files are useful for simple, temporary storage
- But they lack structure, concurrency control, and scalability
 - Processing files can get a bit tedious and it's easy to accidentally overwrite data.

Example - Airlines and flights data

- Consider an airline application that manages flights, where each flight is structured like what's shown on the right
- Each flight entry records its origin, destination, and duration (in minutes).
- This kind of structured data is essential for applications that need to persist flight schedules, search routes, or calculate travel times.

origin	destination	duration
New York	London	415
Shanghai	Paris	760
Istanbul	Tokyo	700
New York	Paris	435
Moscow	Paris	245
Lima	New York	455

Can we use JSON files for this application?

- When working with file-based JSON, things get messy as the application grows.
- What If We Want to Filter Our Data?
 - All flights from New York
 - The shortest flight
 - Flights that arrived in Paris
 - Flights last week
 - Flights longer than 5 hours
 - etc...
- Question: How might you write the code to filter flights in all these different ways using plain files?
 - Loading the entire file every time
 - Parsing and iterating manually
 - Writing custom logic for every case
 - Risking data loss when writing back
- **A database allows us to do this with a simple query!**

flights.json

```
{  
  "flights": [  
    {  
      "origin": "New York",  
      "destination": "London",  
      "duration": 415  
    },  
    {  
      "origin": "Shanghai",  
      "destination": "Paris",  
      "duration": 760  
    },  
    {  
      "origin": "Istanbul",  
      "destination": "Tokyo",  
      "duration": 700  
    }  
  ]  
}
```

What is a Database

- A database is an electronically stored, systematic **collection of data**.
- It can contain any type of data, including words, numbers, images, videos, and files.
- You can use software called a database management system (DBMS) to store, retrieve, and edit data.
- Databases allow:
 - Querying data efficiently
 - Handling multiple users safely
 - Enforcing data types and constraints
 - Persisting and retrieving data reliably
- Ideal for applications with growing data and users
- What are some examples of data you could store in a database?
 - Pokedex, Book Reviews, Store Inventory, User Information, Airlines and flight information

Database Software

- A **database management system (DBMS)** is a software system for creating and managing databases.
- A DBMS enables end users to create, protect, read, update and delete data in a database.
- It also manages security, data integrity and concurrency for databases.

Popular DBMSs

- Oracle
- Microsoft SQL Server (powerful) and Microsoft Access (simple)
- PostgreSQL (powerful/complex free open-source database system)
- SQLite (transportable, lightweight free open-source database system)
- MySQL and MariaDB (simple free open-source database system)
- MongoDB and Apache Cassandra (popular NoSQL databases)

Types of Databases

Types of databases:

- **Hierarchical**: Organizes data in a tree-like structure with parent-child relationships.
- **Relational**: Stores data in tables with rows and columns, using keys to relate them
- **Object-oriented**: Store data as objects, like in object-oriented programming
- **NoSQL**: Handle unstructured or semi-structured data, using flexible formats like JSON or key-value pairs.

Structured Query Language (SQL)

(영역)
특정 domain에 특화된 언어

- A “domain-specific language” (HTML is also a DSL) designed specifically for data access and management.
- SQL is a declarative language: describes what data you are seeking, not exactly how to find it.
 - HTML: markup language, JavaScript: interpreted imperative language
- In SQL, you write statements. The main different types of statements we'll look at:
 - Data Definition Language (DDL): is used to create, modify, or destroy objects within an RDBMS
 - Data Manipulation Language (DML): is the domain of INSERT, UPDATE, and DELETE, which you use to manipulate data.

SQLite data types

- **NULL**: The value is a NULL value. Like in JavaScript, stands for the absence of value
- **INTEGER**: The value is a signed integer, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value.
- **REAL**: The value is a floating-point value, stored as an 8-byte IEEE floating point number.
- **TEXT**: The value is a text string, stored using the database encoding (UTF-8, UTF-16BE or UTF-16LE).
- **BLOB**: The value is a blob of data, stored exactly as it was input (Binary Large Object, e.g., audio and video)



CREATE TABLE

DDL

- **CREATE TABLE** is used to create a new table.

```
1 CREATE TABLE flights (
2     id INTEGER PRIMARY KEY AUTOINCREMENT,
3     origin TEXT NOT NULL,
4     destination TEXT NOT NULL,
5     duration INTEGER NOT NULL
6 );
```

- Constraints - they “constrain” the types of values you can insert in a column.
 - **PRIMARY KEY** (keyname): Used to specify a column or group of columns uniquely identifies a row in a table.
 - Every table should have a column which is used to uniquely identify each row.
 - This improves efficiency and will prove very useful when using multiple tables.
 - **AUTOINCREMENT**: Used with an integer primary key column to automatically generate the “next” value for the key.
 - In SQLite, only available for a primary key that's an INTEGER.
 - **NOT NULL**: prevents NULL entries in a column, requires the value to be set in INSERT statements.
 - **DEFAULT**: specifies default values for a column if not provided in an INSERT statement

INSERT INTO TABLE

DML

- To insert a new record into a table, we use the `INSERT INTO` keyword

```
1 INSERT INTO flights
2   (origin, destination, duration) ⇒ id is omitted due to AUTOINCREMENT
3   VALUES ("New York", "London", 415);
```

- First provide the table name, then optionally the list of columns you want to set (by default it sets all columns).
- Columns left out will be set to `NULL`, unless they have `AUTOINCREMENT` set or some `DEFAULT` value set.
- Then provide the values for each column, which must match the column names specified.

`INSERT INTO` `tablename`
`(column, ...)`
`VALUES (values, ...);`

SELECT statement

DML

- The **SELECT** statement is used to return data from a database.
- It returns the data in a result table containing the **row** data for **column name(s)** given.
- Table and column names are **case-sensitive**.

1 **SELECT * FROM flights;**

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

SELECT some columns

```
1 SELECT origin, destination FROM flights;
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

SELECT rows using the WHERE clause

- The WHERE clause filters out rows based on their columns' data values.
- The WHERE portion of a SELECT statement can use the following operators:
 - =, >, >=, <, <=
 - <> or != (not equal)
 - LIKE pattern
 - IN (value, value, ..., value)

```
1 SELECT * FROM flights WHERE id = 3;
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

SELECT rows based on a text column

```
1 SELECT * FROM flights WHERE origin = "New York";
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

SELECT a few rows based on numerical column

```
1 SELECT * FROM flights WHERE duration > 500;
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

SELECT rows based on a **conjunction** of conditions

```
1 SELECT * FROM flights WHERE duration > 500 AND destination = "Paris";
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

SELECT rows based on a ~~disjunction~~ of conditions

```
1 SELECT * FROM flights WHERE duration > 500 OR destination = "Paris";
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

SELECT rows using the IN operator

- The `IN` operator allows you to specify multiple values in a `WHERE` clause.
- The `IN` operator is a shorthand for multiple OR condition

```
1 SELECT * FROM flights WHERE origin IN ("New York", "Lima");
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

SELECT rows using the LIKE operator

- The SQL **LIKE** operator is used for performing pattern-based searches in a database.
- There are two wildcards often used in conjunction with the LIKE operator:
 - The percent sign **%** represents zero, one, or multiple characters $[0 \sim \infty)$
 - The underscore sign **_** represents one, single character **1**

```
1 SELECT * FROM flights WHERE origin LIKE "%a%"; just include "a" in value
```

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

Grouping functions

- AVERAGE
- COUNT
- MAX
- MIN
- SUM ...

UPDATE

DML

- To update an existing record in a table, we use the **UPDATE** and **SET** keywords.

```
1 UPDATE flights  
2 SET duration = 430  
3 WHERE origin = "New York" AND destination = "London";
```

DELETE

DML

- To delete a record from a table, we use the **DELETE** keyword:

```
1 DELETE FROM flights WHERE destination = "Tokyo";
```

⚠ Warning

- What happens if we forget to add a WHERE clause? ⇒ empty table ...

Other clauses

- LIMIT
- ORDER BY
- GROUP BY
- HAVING

Foreign keys

- A **foreign key** is a column in one table that links to the primary key in another table, creating a relationship between the two tables and ensuring data consistency.
- Let's go back to our flights table

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

Adding the airport code to flights table

id	origin	origin_code	destination	destination_code	durat
1	New York	JFK	London	LHR	415
2	Shanghai	PVG	Paris	CDG	760
3	Istanbul	IST	Tokyo	NRT	700
4	New York	JFK	Paris	CDG	435
5	Moscow	SVO	Paris	CDG	245
6	Lima	LIM	New York	JFK	455

Airports table

id	code	city
1	JFK	New York
2	PVG	Shanghai
3	IST	Istanbul
4	LHR	London
5	SVO	Moscow
6	LIM	Lima
7	CDG	Paris
8	NRT	Tokyo

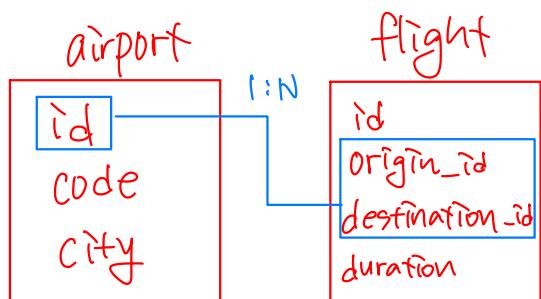
Old flights table

id	origin	destination	duration
1	New York	London	415
2	Shanghai	Paris	760
3	Istanbul	Tokyo	700
4	New York	Paris	435
5	Moscow	Paris	245
6	Lima	New York	455

Normalized flight table

- A **foreign key** is a column in one table that links to the primary key in another table, creating a relationship between the two tables and ensuring data consistency.

id	origin_id	destination_id	duration
1	1	4	415
2	2	7	760
3	3	8	700
4	1	7	435
5	5	7	245
6	6	1	455



Flights table with foreign keys

- Now we also want to store the information of passengers

id	first	last	flight_id
1	Harry	Potter	1
2	Ron	Weasley	1
3	Hermione	Granger	2
4	Draco	Malfoy	4
5	Luna	Lovegood	6
6	Ginny	Weasley	6

People table

id	first	last
1	Harry	Potter
2	Ron	Weasley
3	Hermione	Granger
4	Draco	Malfoy
5	Luna	Lovegood
6	Ginny	Weasley

Normalized passenger table

person_id	flight_id
1	1
2	1
2	4
3	2
4	4
5	6
6	6

How can we use multiple tables in one SQL query?

- When you have relationships defined with **FOREIGN KEY/PRIMARY KEY**, it is often useful to reference both tables to combine data (e.g. displaying the airport info for a flight, or the passenger info for a flight)
- We can reference multiple tables either with an additional **WHERE** constraint or the **JOIN** keyword.

Multi table WHERE syntax

```
1 SELECT people.first, flights.origin, flights.destination  
2 FROM people, flights, passengers  
3 WHERE people.id = passengers.person_id  
4 AND flights.id = passengers.flight_id;
```

- Or we can use joins...

JOIN

```
1 SELECT people.first, flights.origin, flights.destination  
2 FROM passengers  
3 JOIN people ON passengers.person_id = people.id  
4 JOIN flights ON passengers.flight_id = flights.id;
```

first	origin	destination
Harry	New York	London
Ron	New York	London
Hermione	Shanghai	Paris
Draco	New York	Paris
Luna	Lima	New York
Ginny	Lima	New York

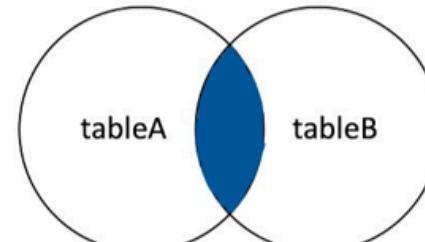
Types of JOINS

- JOIN / INNER JOIN
- LEFT OUTER JOIN
- RIGHT OUTER JOIN
- FULL OUTER JOIN

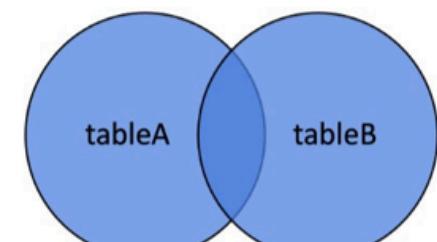
Intersection of tables is denoted by:

```
1 ON tableA.id = tableB.id;
```

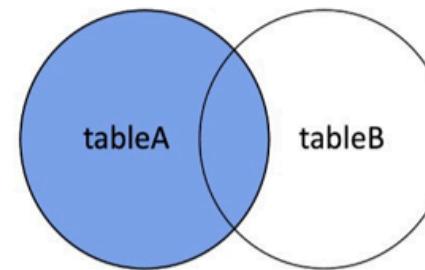
- Not all of these joins are supported by SQLITE.
- For example, full outer join and right joins are not currently supported.



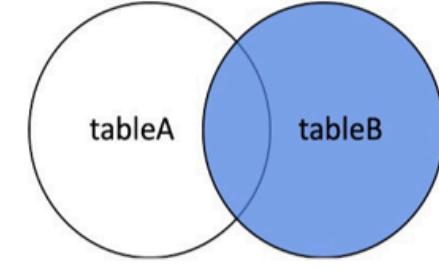
Inner join



Full outer join



Left outer join



Right outer join

SQL Injection

- An **SQL injection** is a security vulnerability where attackers insert malicious SQL code into a query input.
- It can allow attackers to bypass authentication, steal, modify, or delete data in the database.
- SQL injection happens when user input is not properly validated or directly included in SQL statements.
- Let's say we have the following form in a webpage

Username:

Password:

- And we have the following SQL query to retrieve the information about a user

```
1 SELECT * FROM users  
2 WHERE username = username AND password = password;
```

Retrieving harry's information

- Harry login introducing his information

Username:

Password:

- Inserting harry's credentials in our previous query results in the following query:

```
1 SELECT * FROM users
2 WHERE username = "harry" AND password = "12345";
```

SQL injection in action

- Consider the situation where a malicious user enters this as the user input: 'hacker"--'

Username:

`hacker"--`

Password:

- Inserting the hacker's credentials in our previous query results in the following query:

```
1 -- this is a comment in SQL
2 SELECT * FROM users
3 WHERE username = "hacker"--" AND password = "";
```

Data layer of a web application with Node.js

- `sqlite` vs `sqlite3`
- `sqlite3` is the official SQLite bindings for Node.js.
- `sqlite3` is the most popular module used to connect to a SQLite database in Node.js
- Similar to the `fs` module, `sqlite3` functions by default use callbacks for results and errors
- Callbacks?
- We will use `sqlite` which is a promise-based wrapper built on top of `sqlite3`.



Using SQL in Node.js

- First install the `sqlite3` and `sqlite` modules in your project.

```
1 npm install sqlite sqlite3
```

- Second, require both modules with the rest of your modules in your Node.js program.

```
1 const sqlite3 = require('sqlite3');
2 const sqlite = require('sqlite');
```

SQL connection

```
1 const sqlite3 = require('sqlite3');
2 const sqlite = require('sqlite');
3 const fs = require('fs');
4
5 const DB_NAME = "airline.db";
6 /**
7  * Establishes a database connection to the database and returns the database object.
8  * Any errors that occur should be caught in the function that calls this one.
9  * @returns {Promise<sqlite3.Database>} - The database object for the connection.
10 */
11 async function getDBConnection(dbFileName) {
12
13     const db = await sqlite.open({
14         filename: dbFileName,
15         driver: sqlite3.Database
16     });
17     console.log("Connection succesful.")
18     return db;
19 }
```

- If you run `getDBConnection` function, you will see a file called `airline.db` created in your root folder.
- SQLite is a file based, single disk file (read more [here](#))

Methods for Querying the Database

- `.all()`
 - execute an SQL query, returns a Promise which resolves to an array of objects with all the resulting rows
- `.exec()`
 - Executes a SQL query, returns a Promise which resolves to undefined (nothing is returned by this method)
- `.get()`
 - executes an SQL query, returns a Promise which resolves to an object of only the first resulting row
- `.run()`
 - executes a SQL query, returns a Promise which resolves to an object with metadata but no results/data from the query
- Establishing a connection using our `getDbConnection` function
 - Inside an `async` function

```
1 let db = await getDbConnection(DB_NAME); //just for testing
```

- Using `.then/.catch` syntax

```
1 getDBConnection(DB_NAME); //just for testing
```

Loading our dataset

- The following code creates the flights table and insert some files.
 - You can get the `flights_schema.sql` file [here](#)

```
1  async function loadDB(dbFileName, sqlFileName) {
2    try {
3      let db = await getDBConnection(dbFileName);
4      const sql = fs.readFileSync(sqlFileName, 'utf8');
5      const result = await db.exec(sql);
6
7      console.log('Database loaded successfully.');
8      console.log(result);
9
10     await db.close();
11   } catch (err) {
12     console.error('Failed to load database:', err.message);
13   }
14 }
15
16 loadDB(DB_NAME, "flights_schema.sql");
```

db.close()

- The `close` function closes the db connection and ensures that its associated resources are deallocated.
- If we don't close our connections it's possible we could consume all the memory available to our program, which would cause it to crash.
- This is known as a memory leak.
- `.close()` returns a Promise, so it must be awaited

Executing SQL queries with the db object

- Once you have some data and your `db` object, you can now execute SQL queries with `db.all`
- This function takes a SQL query string and an optional array of parameters placeholder values and returns the resulting rows.

```
1  async function executeQuery() {
2      try {
3          let db = await getDBConnection(DB_NAME);
4
5          const sql = "SELECT * FROM flights;";
6          let result = await db.all(sql);
7          console.log(result);
8          await db.close();
9      } catch (err) {
10          console.error('Failed to load database:', err.message);
11      }
12  }
13
14 executeQuery();
```

Using db.exec(sqlString)

- Executes a SQL query, returns a **Promise** which resolves to undefined (nothing is returned by this method)

```
1  async function executeInsert() {
2      try {
3          let db = await getDBConnection(DB_NAME);
4
5          const sql = "INSERT INTO flights (origin, destination, duration) VALUES ('Incheon', \\"Seoul\\", 1000)";
6          let result = await db.exec(sql);
7          console.log(result);
8          await db.close();
9      } catch (err) {
10          console.error('Failed to load database:', err.message);
11      }
12  }
13
14 executeInsert();
```

Using db.run(sqlString)

- Executes an SQL query, returns a **Promise** which resolves to an object with metadata but no results/data from the query

```
1  async function runInsert() {
2      try {
3          let db = await getDBConnection(DB_NAME);
4
5          const sql = "INSERT INTO flights (origin, destination, duration) VALUES ('Guatemala',
6              let result = await db.run(sql);
7              console.log(result);
8              await db.close();
9      } catch (err) {
10          console.error('Failed to load database:', err.message);
11      }
12  }
13
14 runInsert();
```

Extracting the data

- The column (field) names for each row (record) can be accessed using dot notation (it's just an object!)
- Note that only the column names specified in the **SELECT** statement will be accessible
 - In this case we selected all columns with the wildcard `*`

```
1  async function extractData() {
2      try {
3          let db = await getDBConnection(DB_NAME);
4
5          const sql = "SELECT * FROM flights;";
6          let result = await db.all(sql);
7          result.forEach(flight => {
8              console.log(`Flight ${flight.id}, from ${flight.origin} to ${flight.destination} takes ${flight.duration} hours`);
9          });
10         await db.close();
11     } catch (err) {
12         console.error('Failed to load database:', err.message);
13     }
14 }
15
16 extractData();
```

Practice yourself

- Insert three new flights into the flights table
 - From Incheon to San Francisco, Incheon to Amsterdam, Paris to Rome
- Query all flights longer than 600 minutes
- Find all flights arriving in Paris
- List flights originating from New York, Seoul, or Paris
- Create an API endpoint `/flights` that returns a JSON listing all the flights

Next week

- Develop airline webapp and user authentication

Acknowledgements

- Some contents of this lecture are partially adapted from:
 - Harvard [CS50's Web Programming with Python and JavaScript](#), licensed under [CC BY-NC-SA 4.0](#).
 - Materials from University of Washington's [CSE 154 Web Programming](#) (used with permission).
 - [The Odin Project](#) (main website code under MIT license and curriculum licensed under a [CC BY-NC-SA 4.0](#))
 - [The Fundamentals of Web Application Development \(Web Edition\)](#) ©2025 Nicholas D. Freeman. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [freeCodeCamp.org](#) © 2025 freeCodeCamp.org. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [Node.js for Beginners](#) by Ulises Gascón. Some contents adapted under fair educational use for instructional purposes.

Web Programming

Lecture 12

User authentication and authorization

Josue Obregon 

jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

 [Lecture slides index](#)

May 27, 2025

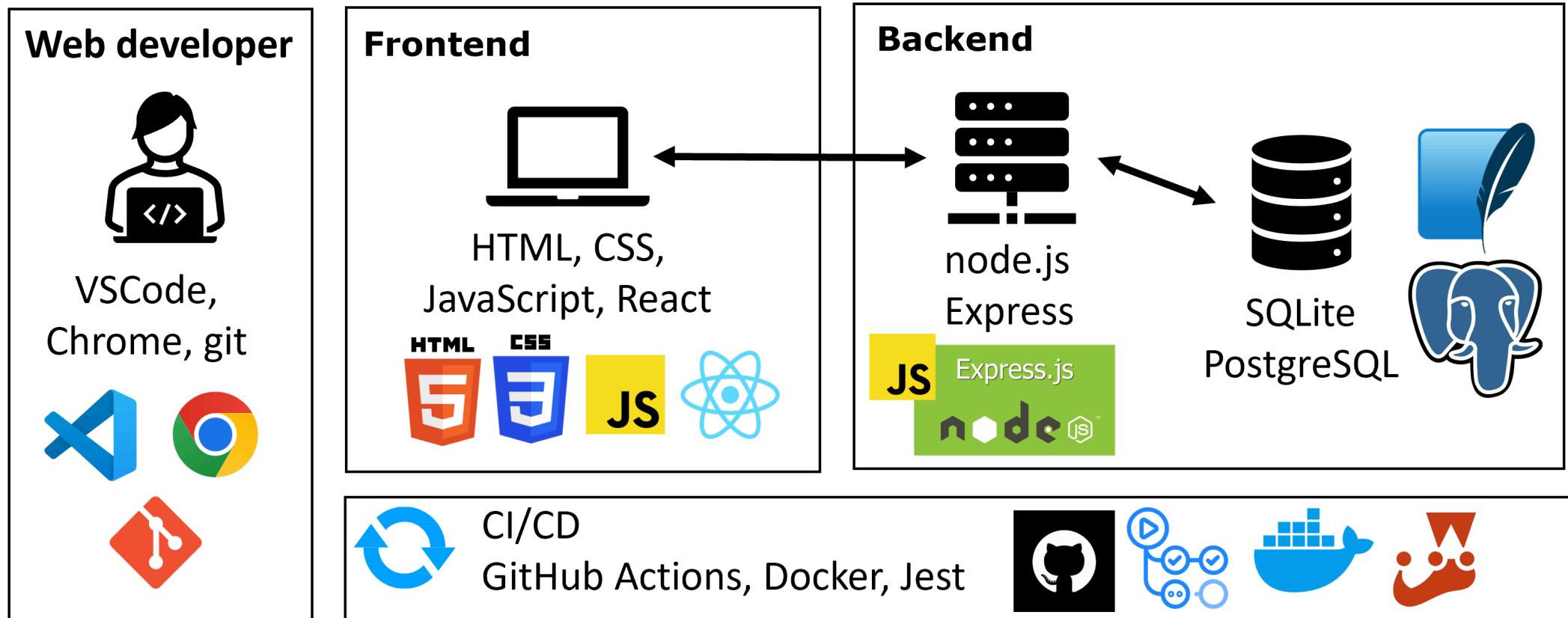


Agenda

- Authentication and authorization
- OAuth
- JWT

Course structure

Modern Web Development Technology Stack



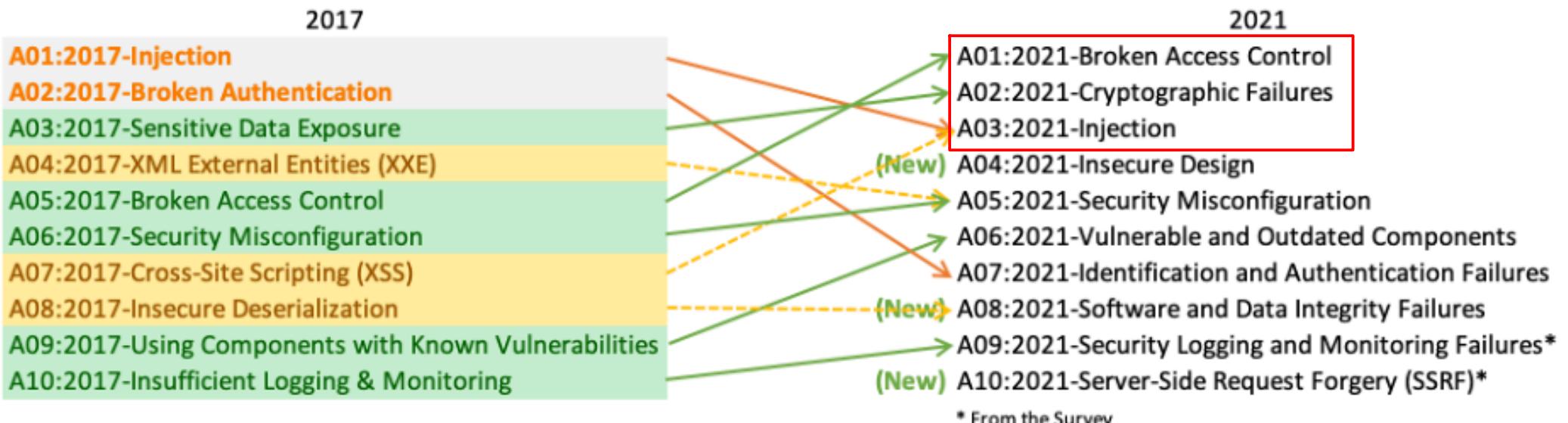
Roadmaps: **Frontend**, **backend** and **fullstack**

The importance of security

- Security was once an afterthought—but not anymore. Apps today are:
 - Exposed 24/7 to the internet
 - Handling sensitive business and user data
 - Built on complex layers and third-party dependencies
- Developers are the first line of defense, even in teams with dedicated security staff.
- Social engineering is now the most common attack vector, it's easier to trick humans than hack code.
- Real business impact: Data breaches cause reputation loss, legal risk, and customer harm.
 - You can check whether your email address has been compromised in a data breach in [this site](#)

Security in web applications and APIs

- APIs often handle sensitive data and business logic.
 - Exposing user info, booking details, payment history.
- Attackers target APIs due to weak endpoints or missing checks.
- Common vulnerabilities presented by the Open Web Application Security Project (OWASP)
WebApps Top 10 and **API Top 10**:



OWASP top 3 vulnerabilities

- **Broken Access Control:** Occurs when users can act outside their authorized roles, accessing or modifying data/functions.
 - Examples: URL tampering, privilege escalation, insecure object references, or broken JWT validation.
 - How to prevent: Enforce access checks server-side; deny access by default except for public resources.
- **Cryptographic Failures:** Happens when sensitive data (e.g., passwords, credit card information) is exposed due to missing or weak encryption.
 - Risks include using outdated algorithms, transmitting data in clear text, or improper key handling.
 - How to prevent: Encrypt all sensitive data in transit (TLS) and at rest using strong, modern standards. Avoid weak/deprecated crypto (e.g., MD5), use secure password hashing (bcrypt, Argon2), and enforce key management best practices.
- **Injection:** Injection flaws happen when untrusted input is sent to an interpreter (e.g., SQL) as part of a command or query. If inputs aren't properly validated or parameterized, attackers can manipulate queries and access or modify data.
 - Use safe APIs or ORMs that support parameterized queries.
 - Validate inputs, avoid dynamic query building, and use query controls like LIMIT to restrict data exposure.

Authentication and Authorization

- **Authentication** confirms the identity of a user, typically through login credentials like username and password.
 - It might include Multi-Factor Authentication (MFA) for added assurance.
- **Authorization** determines what actions or resources a user is allowed to access after they are authenticated.
 - Based on permissions, roles, or scopes.
- These two concepts are often confused but serve distinct purposes in securing web applications.
 - Key Difference: Authentication = “You are user12345.” Authorization = “user12345 can view flights but cannot add new flights.””
- Implementing both correctly ensures that only verified users can access permitted parts of the application.



Types of authentication

- Authentication is typically performed by having a user present a user identifier along with a credential, such as a password, a one-time SMS code, or an assertion signed with a private key.
 - The system then checks the binding between the user identifier and the credential, so it can decide whether or not to authenticate the user.
- Types of authentication information, also called authentication factors, are commonly presented in three categories:
 - Something the user knows, such as a password.
 - Something the user has, such as a phone.
 - Something the user is, such as a thumbprint (biometric identifier).
- Multi-factor authentication (MFA) systems require the user to provide more than one factor: for example, a password combined with a one-time code sent to the user's phone.
- The most common ways for web authentication are:
 - Username and password.
 - Token-based authentication, such as using JSON Web Tokens (JWT), common in RESTful APIs.
 - Session-based authentication.

Authorization

- **Authorization** determines what actions a user is allowed to perform after authentication.
- Common examples:
 - Can the user access a specific page?
 - Can the user create, edit, or delete a post?
- In modern web systems, users often have **roles** (e.g., admin, editor, viewer).
- **Role-Based Access Control (RBAC)** is the most common authorization type:
 - Permissions are grouped by roles.
 - Users are assigned one or more roles.
 - Actions are allowed/denied based on role permissions.
- Skipping authorization checks can expose pages or actions to all users.
- Always verify access rights for each route and action on both front-end and back-end.

OAuth 2.0: Delegated Authorization

- OAuth 2.0 is a token-based **authorization framework**.
- It allows users to grant third-party apps **access to their data without sharing credentials**.
- Used by major providers like Google, Meta, GitHub.
- Ideal for secure delegation: “App A can access my calendar, but not my contacts.”
- Roles in Oauth 2.0:
 - **Resource Owner:** The user who owns the data (e.g., you).
 - **Client:** The application requesting access on the user's behalf (e.g., a calendar app).
 - **Authorization Server:** Authenticates the user and issues access tokens (e.g., Google OAuth server).
 - **Resource Server:** Hosts the protected data and validates access tokens (e.g., Google Calendar API).

 Sign in with Google

 Sign in with Facebook

 Sign in with Twitter

 Sign in with GitHub

 Sign in with email

 Sign in with phone

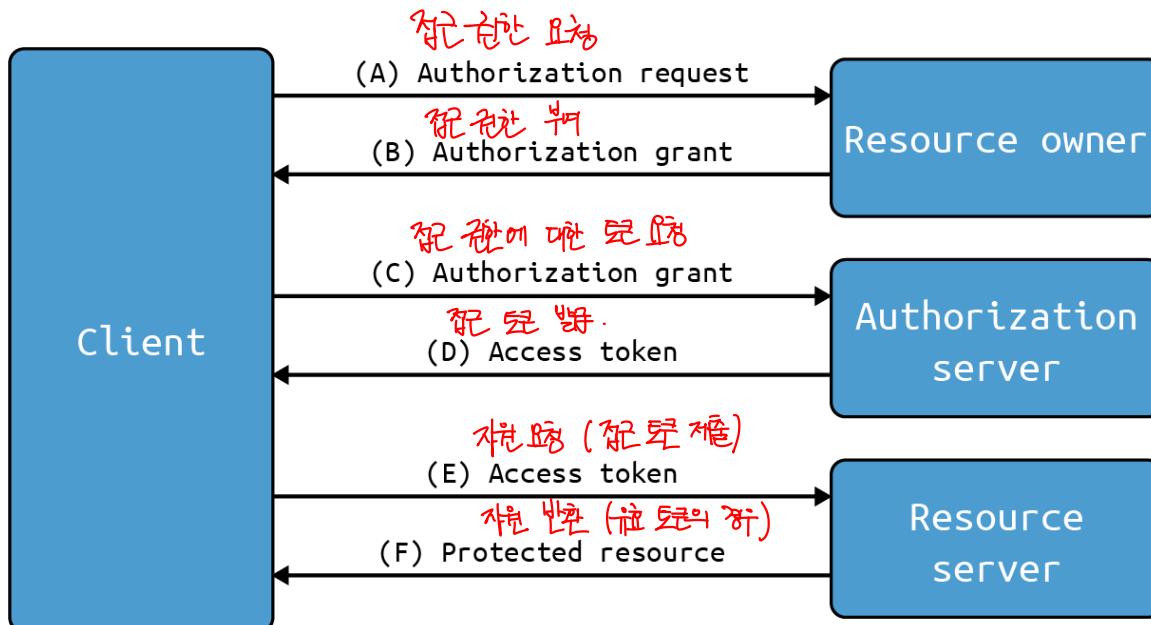
 Sign in with Microsoft

 Sign in with Apple

 Continue as guest

OAuth 2.0 abstract protocol flow

- a. The client requests authorization from the resource owner.
- b. The resource owner will grant or deny the client access to their resources.
- c. The client will ask for an access token from the authorization server for the authorization it has been granted.
- d. The authorization server will issue an access token if the client has been authorized by the resource owner.
- e. The client makes a request for the resource to the resource server, which in our case is the API. The request will send the access token as part of the request.
- f. The resource server will return the resource if the access token is valid.

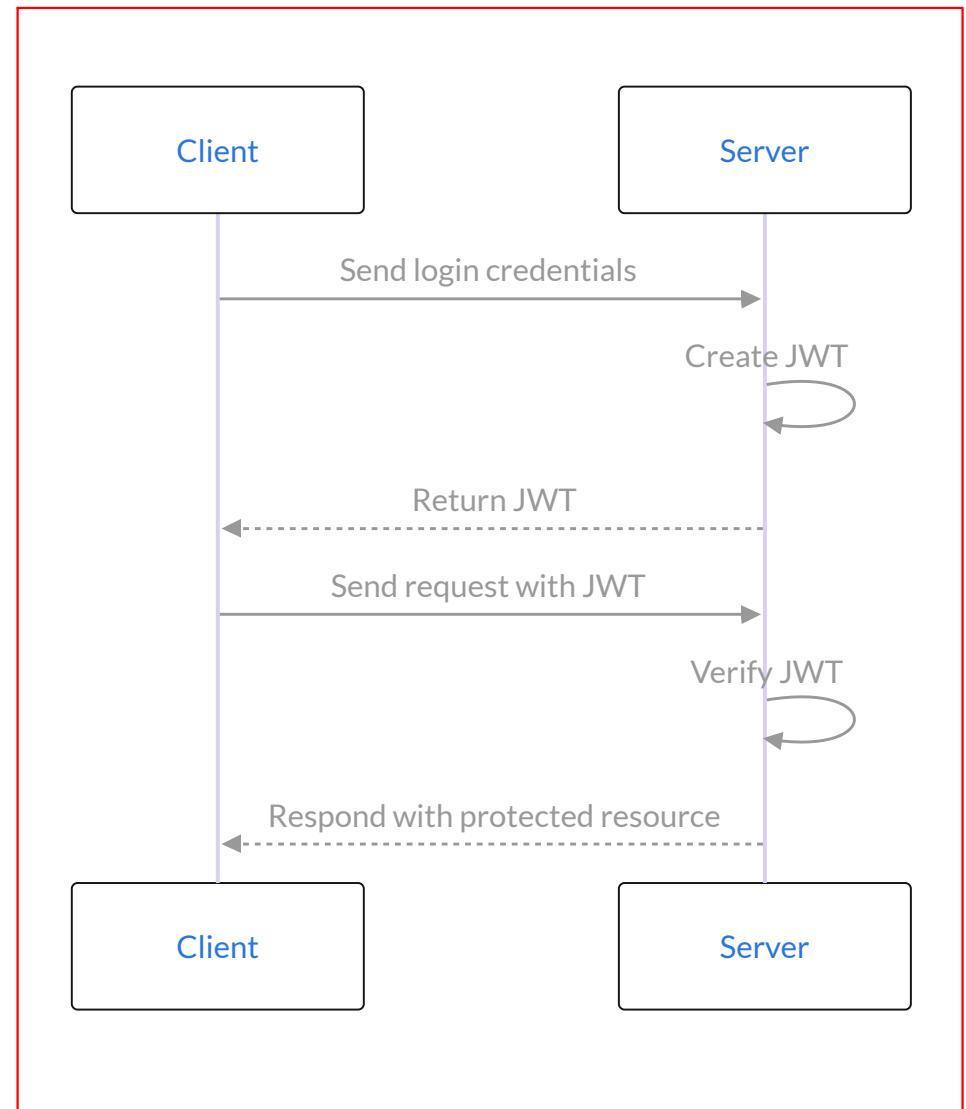


JSON Web Token (JWT)

- One of the most popular ways to implement authentication and authorization in a web application is to use JWT.
- **JSON Web Token** is a proposed internet standard for creating data with optional signature and/or optional encryption whose payload holds JSON that asserts some number of claims. -
The tokens are signed either using a private secret or a public/private key.
- Basically, a **JWT is a string (JSON) that contains information (claims) and is signed using a secret key.**
서버가 서명한 JSON 문자열
- JWTs allow stateless authentication, meaning the server doesn't need to store session data – this makes scaling across multiple servers much easier.
- Each request must be considered untrusted by default, so JWTs include a signed payload that lets the server verify the token's integrity without storing user state.
- JWT signatures protect against unauthorized alterations, ensuring that any changes to the token can be detected, providing both authenticity and security in every request.

JWT process

- In plain terms, the user will authenticate using a username and password, and then the server will return a JWT.
- The user will send the JWT in every request and the server will verify the JWT to authenticate the user
- The JWT is a string with information about the user (such as their name, role, etc.) and is signed using a secret key.
- The server can verify the JWT using the secret key and can then extract the information about the user.
- Any attempt to modify the JWT will invalidate the signature, so the server will reject the request.



Hashing

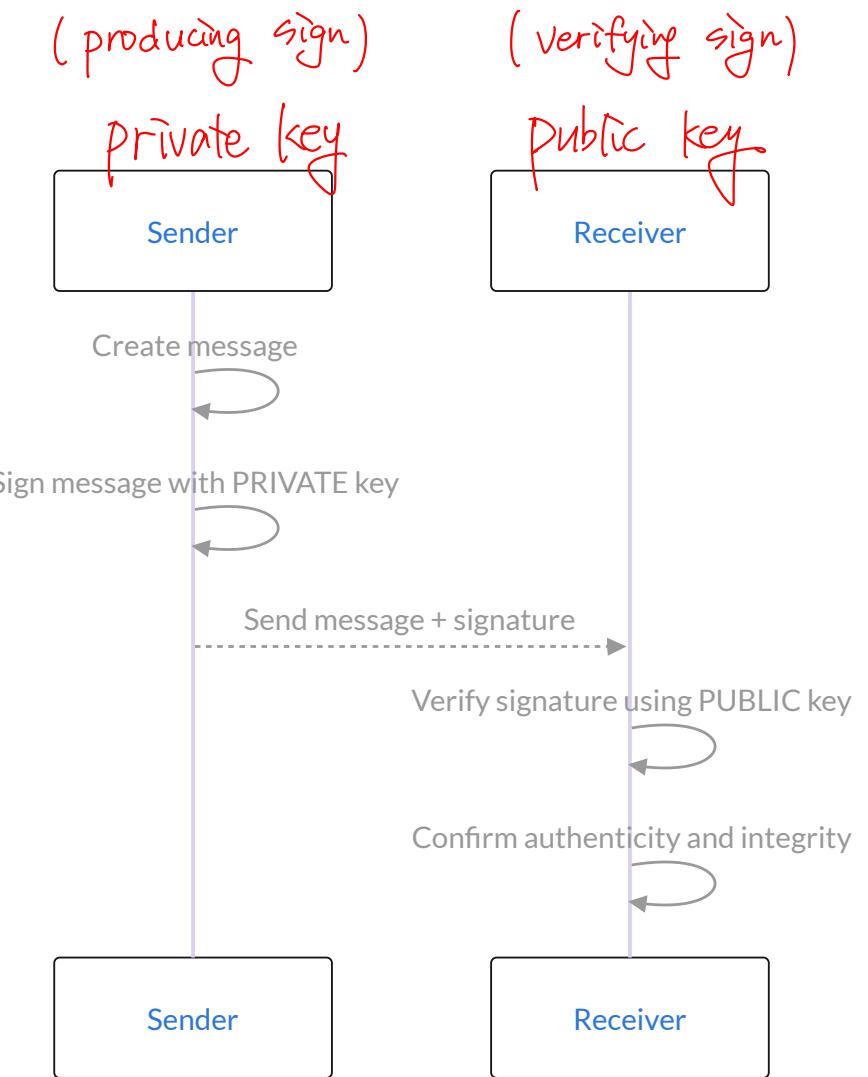
- Hashing transforms any input into a fixed-length string using a one-way algorithm.
 - MD5 (broken), SHA1 (weak), SHA256 (standard), bcrypt (adaptive).
- It's not reversible: you can't get the original input back from the hash.
- Hashes are used to verify data integrity, especially for sensitive info like passwords.

```
1 const bcrypt = require('bcrypt');
2 const hash = await bcrypt.hash('Hello World', 10);
3 console.log(hash);
4 // Output: $2b$10$9Nub0LuSmYa5v0Po.h.9D0h09Cnor/nXSPpakMK2N6D/ZKZdAW1yq
```

- We'll use hashing to store passwords securely during user authentication.
- Why store hashes of passwords instead of the actual passwords?

Signing

- Signing takes content + a secret (or private key) to produce a signature.
- Unlike hashing, signatures can be verified using the original secret or a public key.
- This proves both authenticity (who signed it) and integrity (it wasn't altered).
- For example, Node.js releases include a signed hash file (e.g., **SHASUMS256.txt**)
- Verifying both the hash and signature ensures the file is authentic and untampered.
- In **JWTs**, we use a similar pattern: the payload is hashed and signed (e.g., with HMAC or RSA) instead of using PGP.



JWT structure

- JWT is a string that is composed of three parts separated by a dot. Each part is encoded in base64.
 - **Header**: Contains information about the type of token and the algorithm used to sign the token
 - **Payload**: Contains the claims (information) that we want to store in the token. Example claims: `iss`(issuer, like Google or Auth0), `sub` (subject), `exp`(expiration time), `nbf` (not before), `iat`(issued at) and `jti` (JWT identifier)
 - **Signature**: Contains the signature of the token that is used to verify the token
 - The signature is the result of signing the header and the payload using the secret key.
- We can verify the signature using the secret key, so we can verify the token without needing to store any information on the server.

Header

```
1 {  
2   "alg": "HS256",  
3   "typ": "JWT"  
4 }
```

Payload

```
1 {  
2   "sub": "1234567890",  
3   "name": "John Doe",  
4   "admin": "true",  
5   "iat": 1516239022  
6 }
```

JWT token

```
1 eyJhbGciOiJIUzI1NiIsInR5cC
```

Next week

- CI and CD in web applications

Acknowledgements

- Some contents of this lecture are partially adapted from:
 - Harvard [CS50's Web Programming with Python and JavaScript](#), licensed under [CC BY-NC-SA 4.0](#).
 - Materials from University of Washington's [CSE 154 Web Programming](#) (used with permission).
 - [The Odin Project](#) (main website code under MIT license and curriculum licensed under a [CC BY-NC-SA 4.0](#))
 - [The Fundamentals of Web Application Development \(Web Edition\)](#) ©2025 Nicholas D. Freeman. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [freeCodeCamp.org](#) © 2025 freeCodeCamp.org. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [Node.js for Beginners](#) by Ulises Gascón. Some contents adapted under fair educational use for instructional purposes.

Web Programming

Lecture 13

Testing & Integration in Web Applications

Josue Obregon 

jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

 [Lecture slides index](#)

June 4, 2025

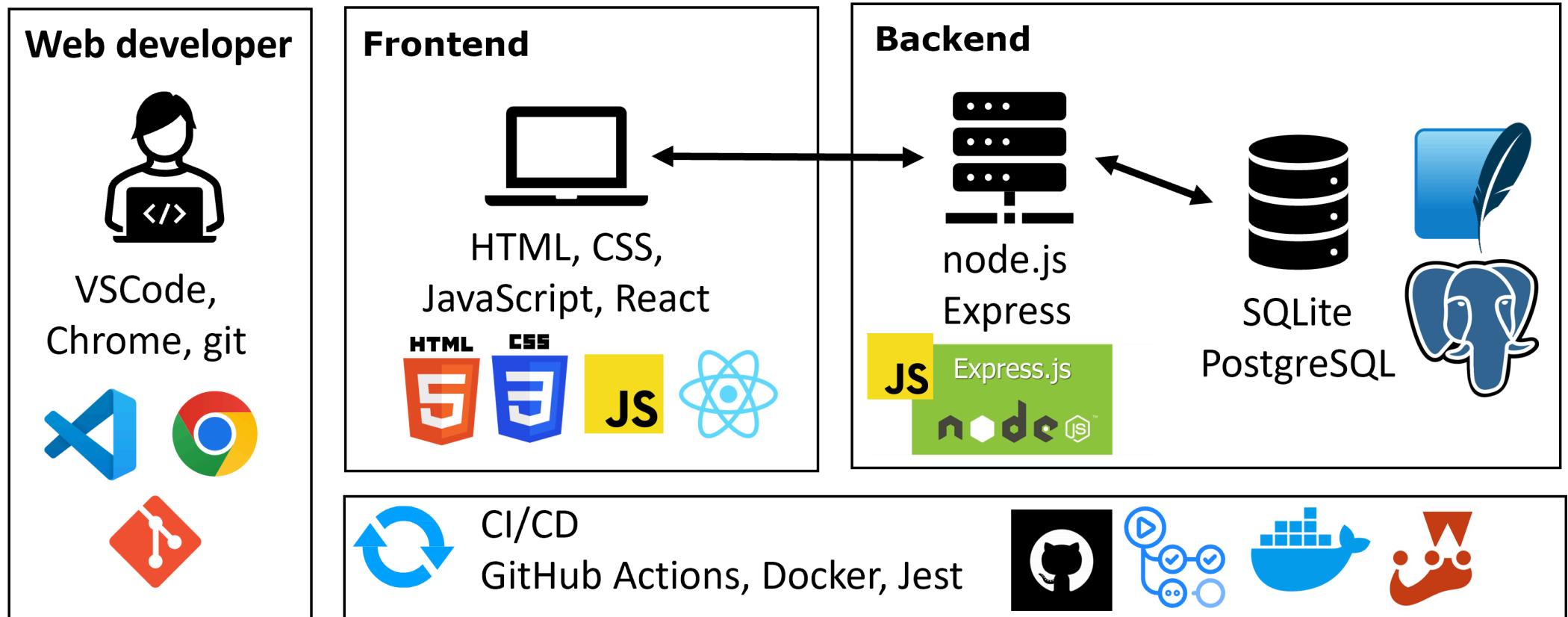


Agenda

- Testing in Web Development
 - Unit tests, service tests and UI tests
- Continuous Integration (CI) and Continuous Delivery (CD)

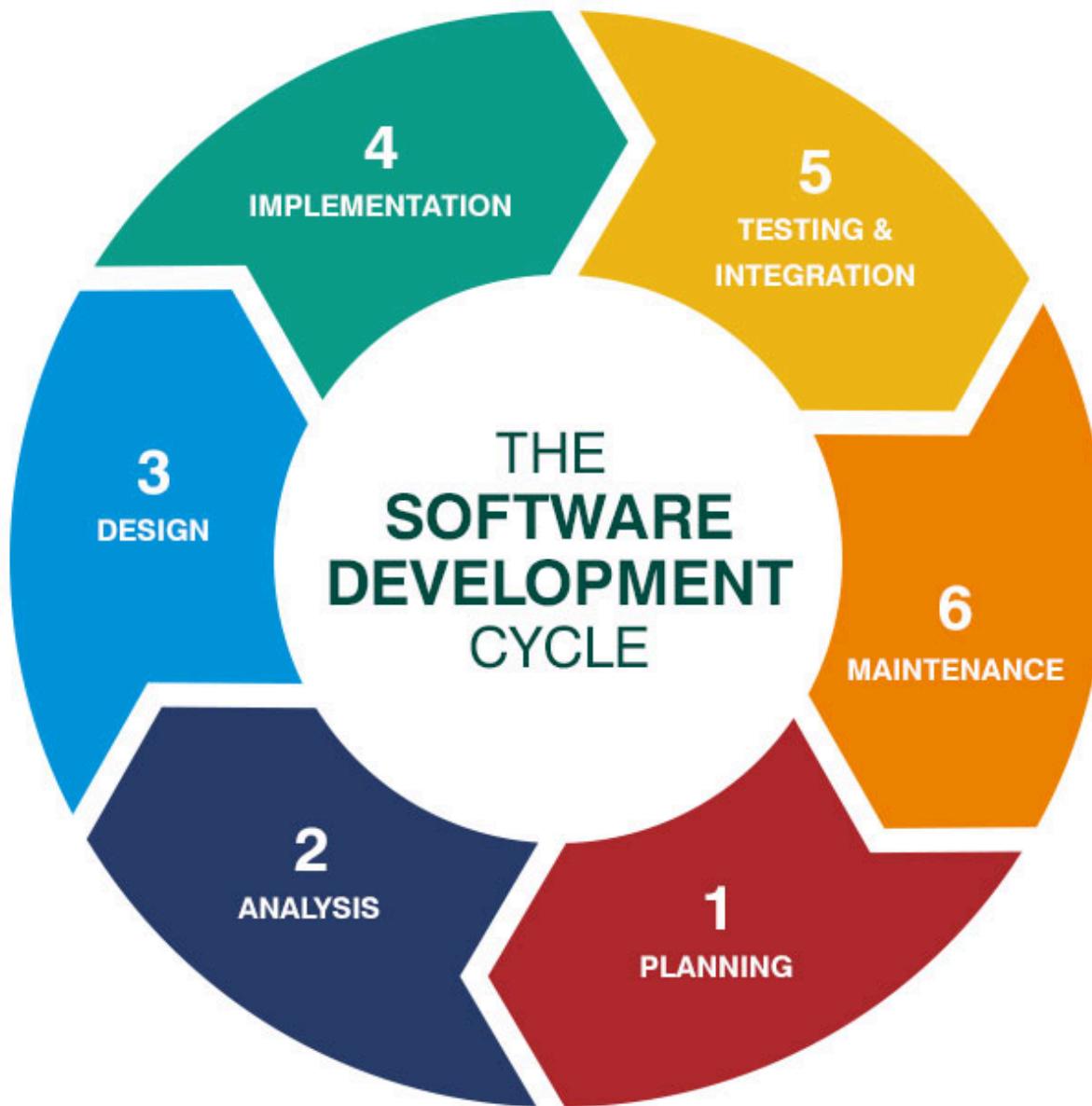
Course structure

Modern Web Development Technology Stack



Roadmaps: **Frontend**, **backend** and **fullstack**

Software Development Life Cycle (SDLC)



Testing in Web Development

- Web applications are complex software with many dependencies and requirements that evolve over time
- Ensures and verifies code correctness
 - Functions work as they are supposed to
 - Webpages behave as intended
- Improves Code Quality
- Facilitates Continuous Integration and Delivery
 - Efficiently and effectively test code as our applications grow large
 - Code changes are continuously tested, leading to faster and safer releases
- Types of Testing: Unit, Integration, System (a.k.a. end-to-end testing)

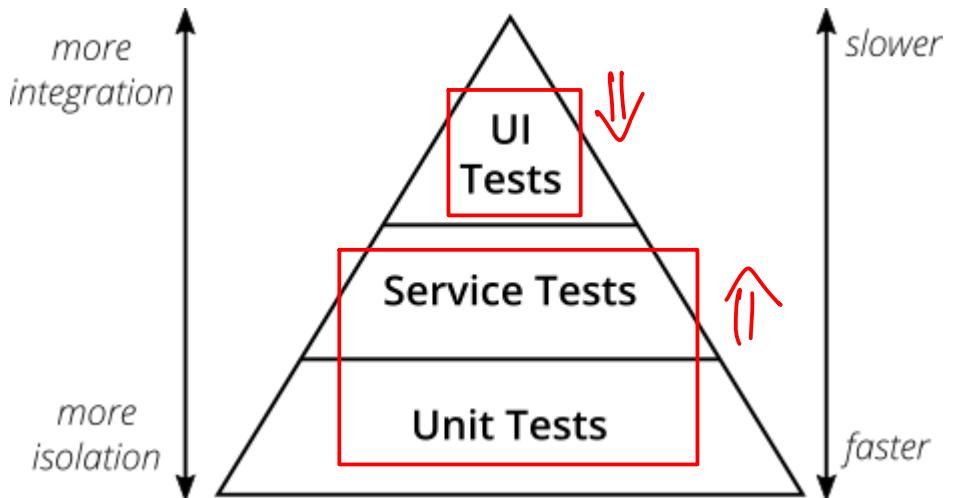
Test Driven Development

- Development style where every time you add a new functionality or fix a bug, you add a test that checks everything works correctly
- The test include a growing set of tests that are run every time you make changes
- When used for adding new functionality (TDD methodology)
 - It emphasizes writing a failing test case first, then writing the minimum amount of code necessary to pass the test, and finally refactoring the code for optimization and clarity.
- When encountering a bug
 - Fix the bug, and then add set of tests

The test pyramid

test type

- **Unit Tests** *DB, Network, UI X,
just logic of each component*
 - Test individual components in isolation.
 - Fast, precise, and cost-effective.
 - Written and maintained by developers.
- **Service Tests** *verifying how different parts of system
work together ⇒ real logic of system*
 - Test application logic and services independent of the UI.
 - Here you test your REST API endpoints, database queries or service functions.
 - More stable and efficient than UI tests.
- **UI Tests**
 - Test the system through the user interface.
 - Slow, fragile, and costly to maintain.
 - Used only for critical user workflows.



How to test JavaScript Code?

- JavaScript lacked a strong testing culture in its early days (browser-only, simple scripts).
- Today, many tools and frameworks support JavaScript testing.
- Knowing how to test lets you switch tools based on your needs.
- `Node.js Test Core` is evolving and promising.
- `Jest` is the most popular and beginner-friendly testing framework.
- For now, use Jest due to better documentation and stable API.

Basic testing with Node.js

```
1 const test = require('node:test');
2 const assert = require('node:assert');
3
4 const sum = (a, b) => a + b;
5
6 test.describe('Operators Test Suite', () => {
7   test.it('Should sum two numbers', () => {
8     assert.strictEqual(sum(1, 2), 3);
9   });
10});
```

- Structure of a Basic JavaScript Test
 - Use `describe` to group related tests (e.g., for a module or function).
 - Use `it` to define individual test cases.
 - Use `assert` to check that the output matches the expected result.
- Test Workflow
 - **Arrange:** Set up the input data needed for the test.
 - **Act:** Run the function or code you want to test.
 - **Assert:** Verify that the output is as expected.

Testing principles

- Fast (*run & write*)
 - Tests should be **quick to run** and **easy to write**.
 - **Slow tests** discourage frequent testing and **slow down development**.
 - **In large projects, test speed becomes critical**—optimize and run in parallel when possible.
- Trustable
 - Tests should be **reliable** and **consistent**.
 - **Avoid flaky** tests that pass or fail randomly.
 - Follow these key principles:
 - **Isolated**: No dependence on external systems (e.g., network, database).
 - **Deterministic**: Same input = same result, every time.
 - **Independent**: Each test runs successfully on its own.
- Maintainable
 - Tests are code too, and must be **easy to update and understand**.
 - Focus on:
 - **Readable and explicit**: Clear logic and purpose.
 - **Single responsibility**: Each test checks one behavior only.
 - **Simplicity**: Prefer many small tests over large, complex ones.

Writing a test suite

- Let's assume we have a project with a utils module (`utils.js`) that perform several arithmetic operations: sum, multiply and divide.
- It looks like this:

```
utils.js
```

```
1 const sum = (a, b) => a + b;
2 const multiply = (a, b) => a * b;
3 const divide = (a, b) => a / b;
4
5
6 module.exports = { sum, multiply, divide };
```

- Basically, we need to test that the `sum` function is summing two numbers, the `multiply` function is multiplying two numbers and the `divide` function is dividing two numbers correctly.

Test suite for utils.js

- Node.js has the core library to build tests: `test` and `assert`.
- First, create a folder `node_test` in the root directory of your project.
- Create a file `utils.test.js`
 - The convention is to name the file `[FILE_TO_TEST].test.js`
- Add several test suites for unit testing `sum`, `multiply` and `divide`.
- Note that we can add several individual tests to a test suite (e.g. the `divide` test suite)

node_test/utils.test.js

```
1 const test = require('node:test');
2 const assert = require('node:assert');
3 const { sum, multiply, divide } = require('../utils');
4
5 test.describe("Utils Test Suite: sum", () => {
6   test.it("Should sum two numbers", () => {
7     assert.strictEqual(sum(1, 2), 3);
8   });
9 });
10
11 test.describe("Utils Test Suite: multiply", () => {
12   test.it("Should multiply two numbers", () => {
13     assert.strictEqual(multiply(5, 3), 15);
14   });
15 });
16
17 test.describe("Utils Test Suite: divide", () => {
18   test.it("Should divide two positive numbers", () => {
19     assert.strictEqual(divide(10, 2), 5);
20   });
21   test.it("Should divide a positive and a negative number", () => {
```

Adding the npm scripts

- Let's add the following NPM scripts to our `package.json` in the `scripts` property.

```
1 {
2   "name": "unit_testing",
3   "version": "1.0.0",
4   "main": "app.js",
5   "scripts": {
6     "node-test": "node --test \"node_test/*.test.js\""
7   },
8   "author": "jobregon",
9   "license": "MIT",
10  "description": ""
11 }
```

- By default if you do not specify a path, Node.js will run all files matching many patterns, including:
 - `**/*.test.{cjs,mjs,js}`
 - `**/*-test.{cjs,mjs,js}`
 - `**/*_test.{cjs,mjs,js}`

Running the test suites

- Now, you can run the tests with the following command: `npm run node-test`
- You will see the following output:

```
F:\webprog\unit_testing>npm run node-test

> unit_testing@1.0.0 node-test
> node --test "node_test/*.test.js"

▶ Utils Test Suite: sum
  ✓ Should sum two numbers (0.7739ms)
▶ Utils Test Suite: sum (2.0312ms)
▶ Utils Test Suite: multiply
  ✓ Should multiply two numbers (0.1265ms)
▶ Utils Test Suite: multiply (0.2332ms)
▶ Utils Test Suite: divide
  ✓ Should divide two positive numbers (0.192ms)
  ✓ Should divide a positive and a negative number (0.1108ms)
  ✓ Should return Infinity when dividing by 0 (0.6883ms)
  ✓ Should return NaN when dividing 0 by 0 (0.255ms)
▶ Utils Test Suite: divide (1.7337ms)

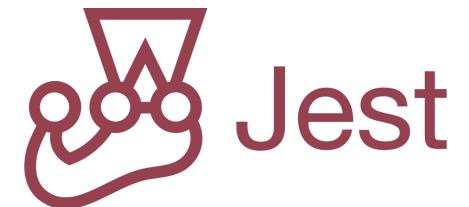
  tests 6
  suites 3
  pass 6
  fail 0
  cancelled 0
  skipped 0
  todo 0
  duration_ms 93.2669
```

- Notice that the terminal uses distinct colors to show us the results of the tests.
 - 3 test suites passed
 - Experiment yourself by making some tests fail and see how does the output change.

Using Jest library

- Jest is a JavaScript testing framework that is very popular in the JavaScript community.
- It's very easy to use and has a lot of features that will help us to build and maintain our test suite
 - Specially for frontend development using modern frameworks, such as Angular, React, or Vue.
- In practice, it is better to only use one library.
- The first step is to install Jest in our project as a development dependency.

```
1 npm install --save-dev jest
```



Configuring Jest

- For the running example, we should add a configuration file `jest.config.js` to specify that we should ignore the `node_test` folder (this is not necessary if you are only using Jest as a test library)

```
jest.config.js
```

```
1 module.exports = {  
2   modulePathIgnorePatterns: ['<rootDir>/node_test/']  
3 };
```

- Let's add the following NPM scripts to our `package.json` file:

```
1 {  
2   "scripts": {  
3     "node-test": "node --test \"node_test/*.test.js\"",  
4     "jest-test": "jest"  
5   }  
6 }
```

Test suite for `utils.js` using Jest

- As you can see, the code is very similar to the code that we created for the Node.js core library.
- The only difference is in how we manage the assertions.
 - `expect()` is used to wrap the actual value you want to test.
 - `.toBe()` checks that the actual value is exactly equal (using `==`) to the expected value.
 - Other matchers `toBeNull()`, `toThrow()`, (more in [this link](#))
- Jest automatically provides the `describe` and `it` functions, so we don't need to import them.

`jest_test/utils.test.js`

```
1 const { sum, multiply, divide } = require('../utils.js');
2
3 describe("Utils Test Suite: sum", () => {
4   test("Should sum two numbers", () => {
5     expect(sum(1, 2)).toBe(3);
6   });
7 });
8
9 describe("Utils Test Suite: multiply", () => {
10  test("Should multiply two numbers", () => {
11    expect(multiply(5, 3)).toBe(15);
12  });
13 });
14
15 describe("Utils Test Suite: divide", () => {
16  test("Should divide two positive numbers", () => {
17    expect(divide(10, 2)).toBe(5);
18  });
19  test("Should divide a positive and a negative number", () => {
20    expect(divide(-10, 2)).toBe(-5);
21  });
22});
```

Running the Jest test suites

- Now, you can run the tests with the following command: `npm run jest-test`
- You will see the following output:

```
F:\webprog\unit_testing>npm run jest-test

> unit_testing@1.0.0 jest-test
> jest

PASS  jest_test/utils.test.js
  Utils Test Suite: sum
    ✓ Should sum two numbers (2 ms)
  Utils Test Suite: multiply
    ✓ Should multiply two numbers
  Utils Test Suite: divide
    ✓ Should divide two positive numbers
    ✓ Should divide a positive and a negative number (1 ms)
    ✓ Should return Infinity when dividing by 0
    ✓ Should return NaN when dividing 0 by 0

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        0.357 s, estimated 1 s
Ran all test suites.
```

- The output is very similar to the output that we saw with the Node.js core library
- Again, experiment yourself by making some tests fail and see how does the output change.

Coverage in testing

- What is **Code Coverage?**
 - A metric that shows **which parts of your code are exercised by tests.**
 - Helps **identify untested or over-tested areas.**
 - Supports quality control of the entire test suite.
- Why It Matters?
 - Reveals **critical logic not covered by tests.**
 - Encourages writing tests for **important scenarios.**
 - Helps **maintain reliable and meaningful test suites.**
- Common Misunderstandings
 - **100% coverage is not always necessary or practical.**
 - **High coverage does not guarantee correctness**—you can cover code with poor tests.
 - Use it as a guide, not a goal.

Configuration of test coverage with Node.js and Jest

- For both libraries, let's add the following NPM scripts to our `package.json`:

```
1 {
2   "scripts": {
3     "node-test": "node --test \"node_test/*.test.js\"",
4     "jest-test": "jest",
5     "jest-test:coverage": "jest --coverage",
6     "node-test:coverage": "node --test --experimental-test-coverage \"node_test/*.test.js\""
7   }
8 }
```

- Node.js has an experimental feature that we can use to generate code coverage.
- We need to use the `--experimental-test-coverage` flag to enable this feature

Adding new functions

- Let's add a new function, `subtract`, to our `utils.js` file:

```
utils.js
```

```
1 const sum = (a, b) => a + b;
2 const multiply = (a, b) => a * b;
3 const divide = (a, b) => a / b;
4 const subtract = (a, b) => a - b;
5
6
7 module.exports = { sum, multiply, divide, subtract };
```

Running the tests

- Let's run the code coverage for both Node.js and Jest to see the results.
- `npm run node-test:coverage` and `npm run jest-test:coverage`
- We have 75% code coverage for the functions, as we don't have any coverage for the `subtract` function.

Node.js

```
F:\webprog\unit_testing>npm run node-test:coverage
> unit_testing@1.0.0 node-test:coverage
> node --test --experimental-test-coverage "node_test/*.test.js"

▶ Utils Test Suite: sum
  ✓ Should sum two numbers (1.1636ms)
▶ Utils Test Suite: multiply
  ✓ Should multiply two numbers (0.1473ms)
▶ Utils Test Suite: divide
  ✓ Should divide two positive numbers (0.1848ms)
  ✓ Should divide a positive and a negative number (0.1013ms)
  ✓ Should return Infinity when dividing by 0 (0.1037ms)
  ✓ Should return NaN when dividing 0 by 0 (0.1608ms)
▶ Utils Test Suite: divide (0.7454ms)
  tests 6
  suites 3
  pass 6
  fail 0
  cancelled 0
  skipped 0
  todo 0
  duration_ms 106.8883
  start of coverage report
  -----
  file           | line % | branch % | funcs % | uncovered lines
  -----
  node_test\utils.test.js | 100.00 | 100.00 | 100.00 |
  utils.js          | 100.00 | 100.00 | 75.00 |
  -----
  all files        | 100.00 | 100.00 | 92.31 |
  -----
  end of coverage report
```

Jest

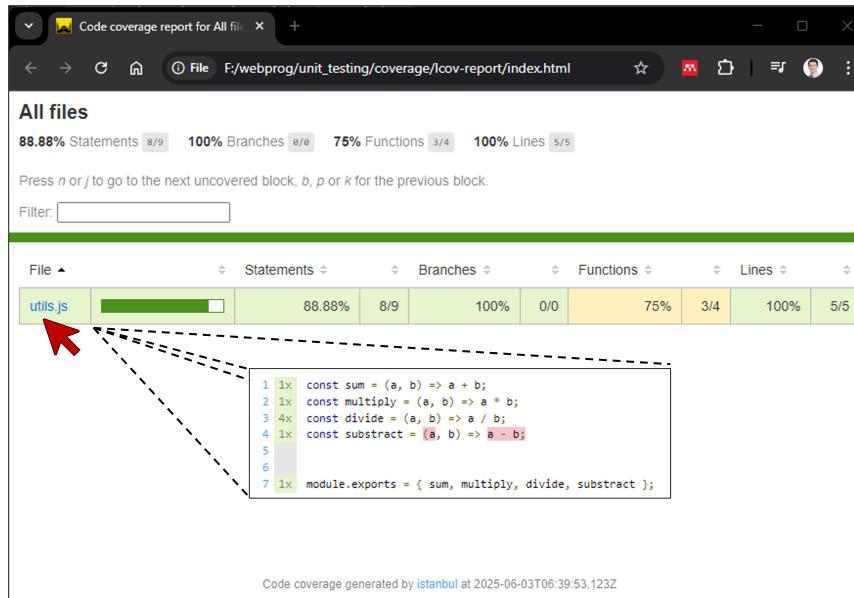
```
F:\webprog\unit_testing>npm run jest-test:coverage
> unit_testing@1.0.0 jest-test:coverage
> jest --coverage

PASS  jest_test/utils.test.js
  Utils Test Suite: sum
    ✓ Should sum two numbers (2 ms)
  Utils Test Suite: multiply
    ✓ Should multiply two numbers
  Utils Test Suite: divide
    ✓ Should divide two positive numbers
    ✓ Should divide a positive and a negative number
    ✓ Should return Infinity when dividing by 0
    ✓ Should return NaN when dividing 0 by 0 (1 ms)

-----
File      | %Stmts | %Branch | %Funcs | %Lines | Uncovered Line #
-----
All files | 88.88 | 100     | 75     | 100    |
utils.js  | 88.88 | 100     | 75     | 100    |
-----
Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        0.603 s, estimated 1 s
Ran all test suites.
```

Coverage UI report

- In both cases, we have generated a coverage folder with the results.
- We can open the `index.html` file located in `coverage/lcov-report` in our browser to see the results.
- We can explore in detail what is and is not covered in `utils.js`
- The code coverage report is a great way to understand your tests, especially when you are working with a large code base.



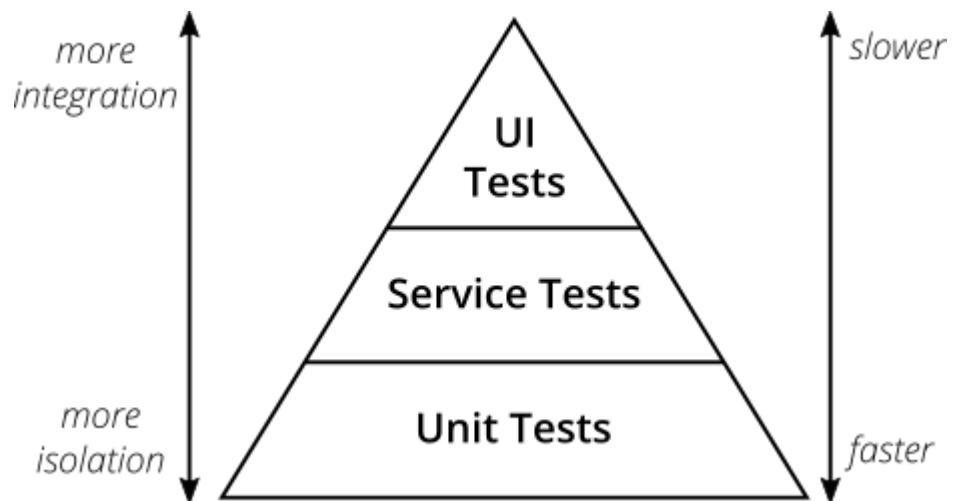
The screenshot shows a browser window displaying a code coverage report titled "Code coverage report for All files". The report includes summary statistics: 88.88% Statements (8/9), 100% Branches (0/0), 75% Functions (3/4), and 100% Lines (5/5). Below this, a table provides detailed coverage per file, with "utils.js" highlighted. A red arrow points to the "utils.js" row. A dashed line connects this row to a callout box containing the following source code:

```
1 1x const sum = (a, b) => a + b;
2 1x const multiply = (a, b) => a * b;
3 4x const divide = (a, b) => a / b;
4 1x const subtract = (a, b) => a - b;
5 
6 
7 1x module.exports = { sum, multiply, divide, subtract };
```

At the bottom of the report, it says "Code coverage generated by [istanbul](#) at 2025-06-03T06:39:53.123Z".

The test pyramid (Service Tests)

- Unit Tests
 - Test individual components in isolation.
 - Fast, precise, and cost-effective.
 - Written and maintained by developers.
- **Service Tests**
 - Test application logic and services independent of the UI.
 - Here you test your REST API endpoints, database queries or service functions.
 - More stable and efficient than UI tests.
- UI Tests
 - Test the system through the user interface.
 - Slow, fragile, and costly to maintain.
 - Used only for critical user workflows.



REST API tests

- How to test that a REST API is working as expected?
- We will learn how to build tests while using Express.
- Let's use the flights application that we implemented in previous lectures
- We need to test this endpoint: **GET /api/flights/:id**
 - Test successful flight information retrieval (status 200)
 - Test that server-side errors are correctly handled (status 500)
 - Test that client side-errors are correctly handled (status 404)
 - We added a 404 response if the id of the flight does not exist (line 9-10)

```
1 app.get(
2   '/api/flights/:id',
3   async (req, res) => {
4     try {
5       const flightId = parseInt(req.params.id);
6       // 1) Flight info
7       const flight = await getFlight(flightId);
8       // check if flight exists
9       if (!flight) {
10         res.sendStatus(404);
11       } else {
12         // 2) Assigned passengers (persons)
13         const passengers = await getPassengers(flightId);
14         res.json({
15           id: flight.id,
16           origin: flight.origin,
17           destination: flight.destination,
18           duration: flight.duration,
19           passengers: passengers
20         });
21       }
22     } catch (err) {
23       res.status(500).json({ message: err.message });
24     }
25   }
26 }
```

SuperTest module

- Provides a high-level abstraction for testing HTTP by simulating HTTP requests directly to your Node.js/Express server.
- It can be used with Jest or node:test.
- Supports: GET, POST, PUT, DELETE, setting headers, sending JSON, etc.
- You can install SuperTest as an npm module

```
1 npm install supertest --save-dev
```

- Example of usage

```
1 const request = require('supertest');
2 const app = require('../app'); // your Express app
3
4 test('GET /api/flights returns 200', async () => {
5   const res = await request(app).get('/api/flights');
6   expect(res.statusCode).toBe(200);
7 });
```

! Important

Do not forget to install Jest in the airline project and add the test scripts to `package.json` as we learned before.

Export your Express app without `.listen()`

- Modify your `app.js` to export the app:

```
1 const express = require('express');
2 const app = express();
3
4 // routes go here...
5
6 module.exports = app;
```

- Create a separate `server.js` if needed, to run the server:

`server.js`

```
1 const app = require('./app');
2 const PORT = 8080;
3 app.listen(PORT, () => {
4   console.log(`Server running at http://localhost:${PORT}`);
5 });
```

- We export `app` and separate `app.listen()` into its own file (e.g., `server.js`) because:
 - SuperTest doesn't need a running server
 - We can use the same app in development, production, and testing.
 - Keep test setup clean and modular.

Test suite for flights API using Jest and SuperTest

- `jest.mock()` allows us to mock modules by erasing the actual implementation of functions and capturing calls to the functions in the module
- Since we are just testing the API logic, we can test the database methods without actually accessing them
- Once we mock the module we can provide a `mockResolvedValue` for `.getFlight` that returns the data we want to use in our tests.
- `it.todo` function marks the tests that we need to add.
 - This is a good practice to keep track of the tests that we need to add, and it does not break the test suite.

tests/flight.test.js

```
1 const request = require('supertest');
2 const app = require('../app');
3
4 // Mock your database functions
5 jest.mock('../db');
6
7 const { getFlight, getPassengers } = require('../db');
8
9 describe('GET /api/flights/:id', () => {
10   it('should return 404 if flight not found', async () => {
11     getFlight.mockResolvedValue(null); // simulate no flight found
12
13     const res = await request(app).get('/api/flights/999');
14     expect(res.statusCode).toBe(404);
15   });
16
17   it('should return flight with passengers', async () => {
18     getFlight.mockResolvedValue({
19       id: 1,
20       origin: 'Seoul',
21       destination: 'Tokyo',
22       duration: 120
23     });
24   });
25 })
```

Running the API tests

- Let's run the tests with code coverage for Jest to see the results.

- `npm run jest-test:coverage`

- There are 2 todo tests

- (`/api/flights/`)

- `app.js` has 50% of coverage
(`app.use` functions were not tested)

- `db.js` has 0% of coverage

- This requires to setup a development database that has the same schema as the production database
 - This is important for integration tests

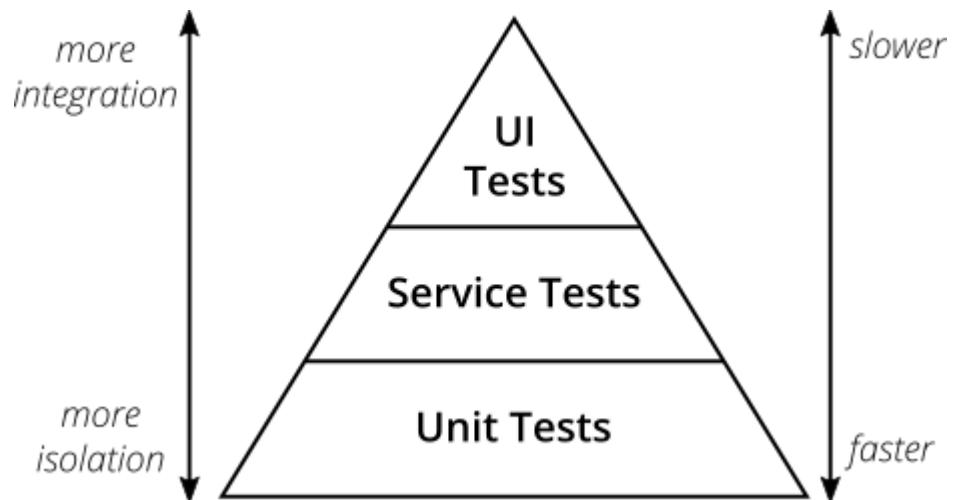
```
F:\webprog\airline>npm run jest-test:coverage
> airline@1.0.0 jest-test:coverage
> jest --coverage

PASS  tests/flight.test.js
  GET /api/flights/:id
    ✓ should return 404 if flight not found (31 ms)
    ✓ should return flight with passengers (14 ms)
    ✓ should return 500 on error (6 ms)
  GET /api/flights
    ✕ todo Should return an empty array when there's no flights data
    ✕ todo Should return all the flights

-----|-----|-----|-----|-----|-----|
File   | % Stmt | % Branch | % Funcs | % Lines | Uncovered Line #
-----|-----|-----|-----|-----|-----|
All files | 53.48 | 100 | 12.5 | 53.48 |
app.js | 94.73 | 100 | 50 | 94.73 | 26
db.js | 20.83 | 100 | 0 | 20.83 | 9-67
-----|-----|-----|-----|-----|-----|
Test Suites: 1 passed, 1 total
Tests: 2 todo, 3 passed, 5 total
Snapshots: 0 total
Time: 1.021 s
Ran all test suites.
```

The test pyramid (UI Tests)

- Unit Tests
 - Test individual components in isolation.
 - Fast, precise, and cost-effective.
 - Written and maintained by developers.
- Service Tests
 - Test application logic and services independent of the UI.
 - Here you test your REST API endpoints, database queries or service functions.
 - More stable and efficient than UI tests.
- **UI Tests**
 - Test the system through the user interface.
 - Slow, fragile, and costly to maintain.
 - Used only for critical user workflows.



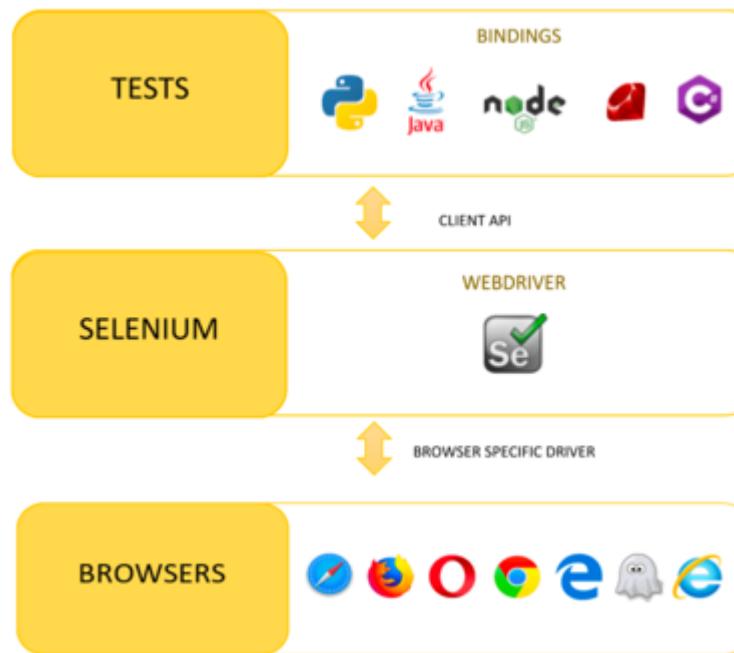
UI/Client-side testing

- Client-side testing verifies that all interactive elements of the user interface (UI) work as intended.
 - This includes buttons, forms, links, and dynamic content.
- Ensures that users can interact with the application as expected, preventing issues like broken buttons, non-responsive forms, and other interaction problems.
- It enhances user experience, identify browser-specific issues and validate client-side logic



Technologies for Client-Side Testing

- **Selenium:** A widely-used open-source tool for automating web browsers.
- It allows you to write scripts in various programming languages (such as Python, Java, and JavaScript) to simulate user interactions with web applications.
- Ideal for end-to-end testing of web applications, verifying that the application works as expected in a real browser environment.



CI/CD

- Stands for Continuous Integration and Continuous Delivery or Deployment
- It aims to streamline and accelerate the software development lifecycle
- CI refers to the practice of automatically and frequently integrating code changes into a shared source code repository
 - Frequent merges to main branch
 - Automated unit testing
- CD is a 2-part process that refers to the integration, testing, and delivery of code changes
Short release cycles



Benefits of CI/CD

- Tackles Small Conflicts Incrementally
 - Many conflicts may arise when multiple features are combined at the same time
- Easier Isolation of Code Issues
 - Unit tests are run with each merge
- Isolates Problems Post-Launch
 - Frequent releasing of new versions help to quickly isolate problems
- Gradual Introduction of New Features
 - Releasing small, incremental changes allows users to slowly get used to new app features
- Competitive Advantage with Rapid Releases

Technologies for CI/CD

GitHub Actions

- GitHub tool integrated into GitHub that automates workflows for building, testing, and deploying code.
- In order to set up a GitHub action, we'll use a configuration language called YAML.
- YAML structures its data around key-value pairs (like a JSON object or Python Dictionary).
- Here's an example of a simple YAML file for node.js:
- `npm ci` is similar to `npm install`, except it's meant to be used in automated environments

```
.github/workflows/node.js.yml

name: Node.js CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

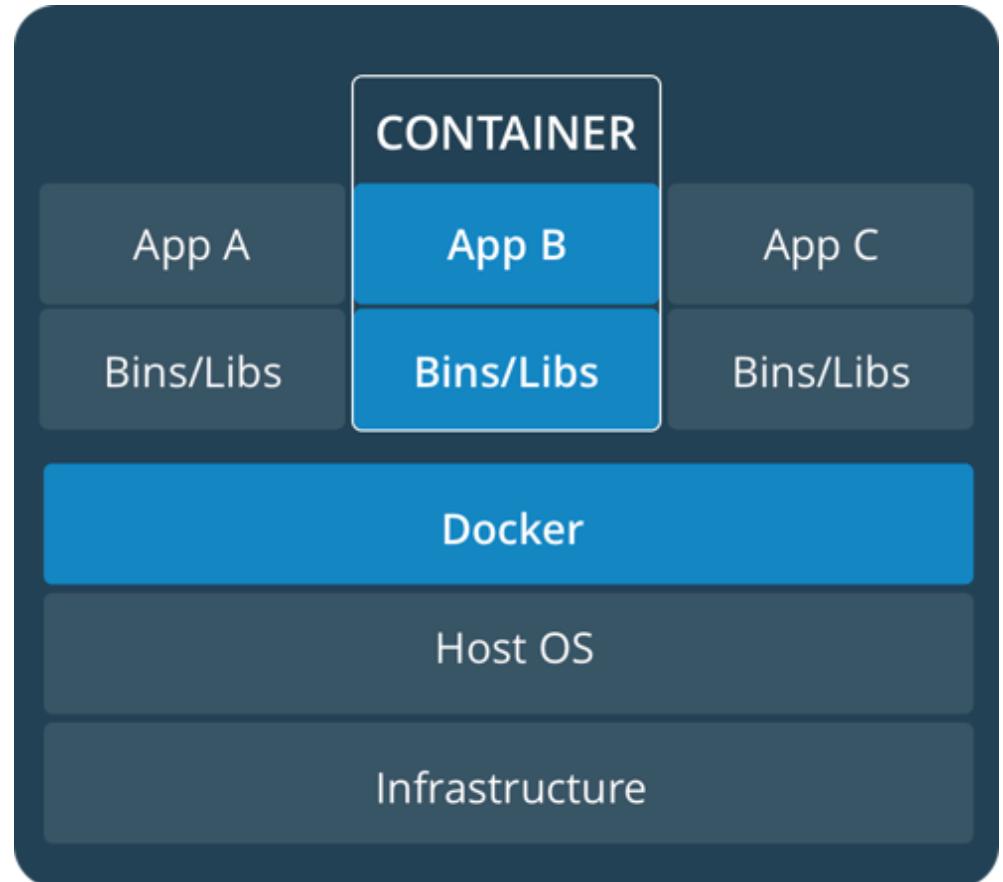
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
```

Technologies for CI/CD

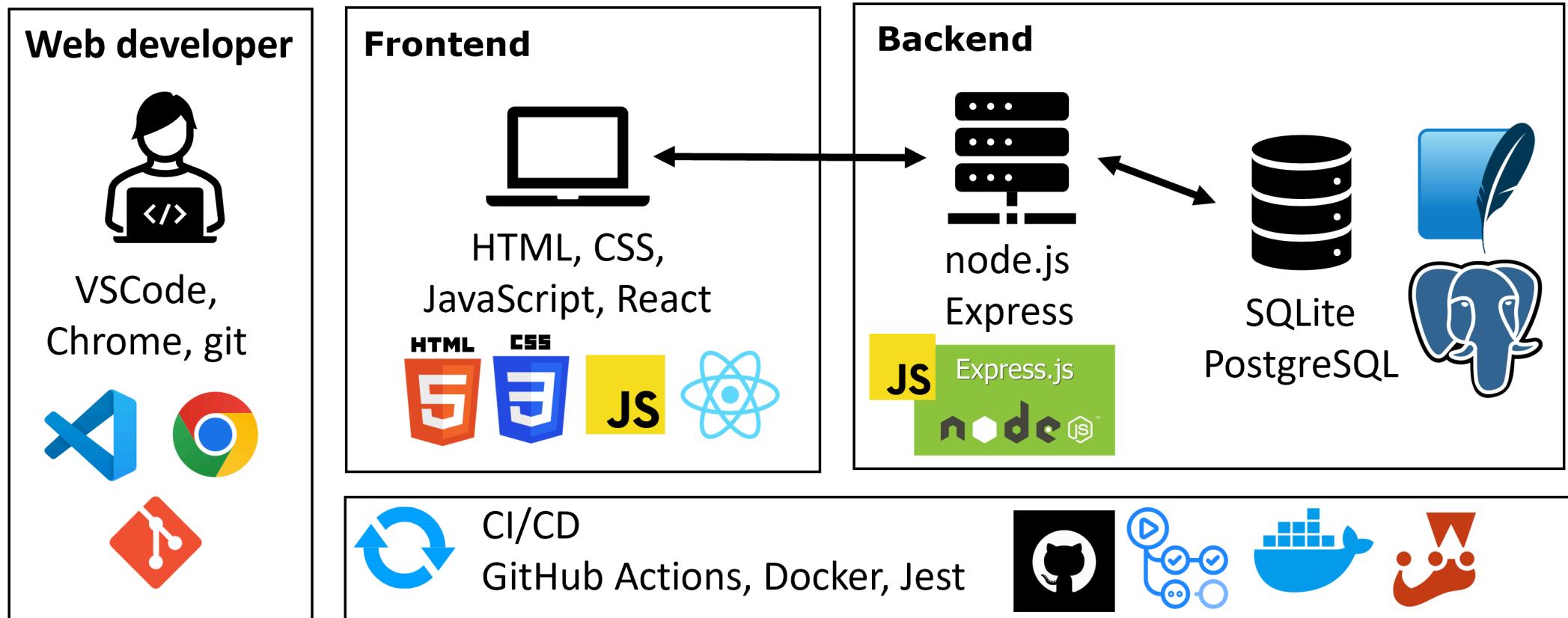
Docker

- Platform that uses containerization to create, deploy, and run applications in isolated environments, ensuring consistency across multiple development and deployment environments.
- Docker ensures that applications run in the same environment from development to production by encapsulating everything needed to run the software, including the code, runtime, libraries, and dependencies.
- Docker containers are lightweight and share the host system's kernel, leading to lower overhead compared to traditional virtual machines.



Course wrap-up

Modern Web Development Technology Stack



Roadmaps: **Frontend**, **backend** and **fullstack**

Acknowledgements

- Some contents of this lecture are partially adapted from:
 - Harvard [CS50's Web Programming with Python and JavaScript](#), licensed under [CC BY-NC-SA 4.0](#).
 - Materials from University of Washington's [CSE 154 Web Programming](#) (used with permission).
 - [The Odin Project](#) (main website code under MIT license and curriculum licensed under a [CC BY-NC-SA 4.0](#))
 - [The Fundamentals of Web Application Development \(Web Edition\)](#) ©2025 Nicholas D. Freeman. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [freeCodeCamp.org](#) © 2025 freeCodeCamp.org. All rights reserved. The content is provided for educational purposes only and is not an exhaustive treatment of the subjects.
 - [Node.js for Beginners](#) by Ulises Gascón. Some contents adapted under fair educational use for instructional purposes.