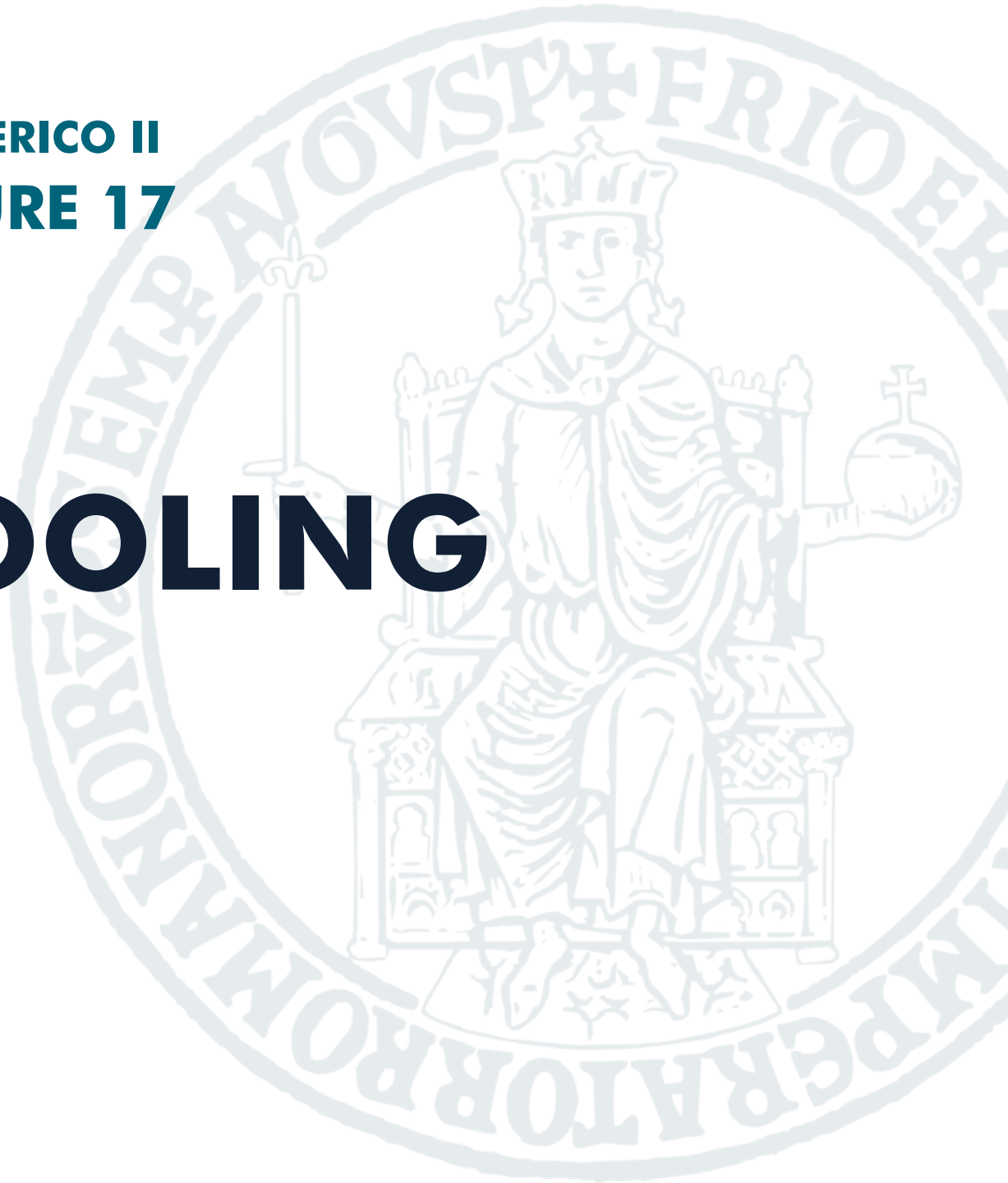
UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 17

FRONTEND TOOLING



Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

https://luistar.github.io

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

THE ROAD SO FAR

- We started our journey with foundational technologies: HTML, CSS, JS.
- We built **static** web pages
- We moved to **server-side** programming
- Still, the web pages we dynamically built were simple
  - A single stylesheet
  - Very little (if any) client-side JS
  - No external libraries/modules

THE ROAD AHEAD

Modern frontends can be much more **complex**!

- Client-side JavaScript code is becoming increasingly complex
  - Many **libraries/modules** are used
  - **Transpiling**, **Downleveling** code
  - **Optimizing** served files
- Often, **stylesheet pre-/post-processors** such as SASS are used

Tooling is required to **support devs** and **automate** these steps!

FRONTEND TOOLING

Today, we will go over the main challenges and tools to develop complex modern frontends

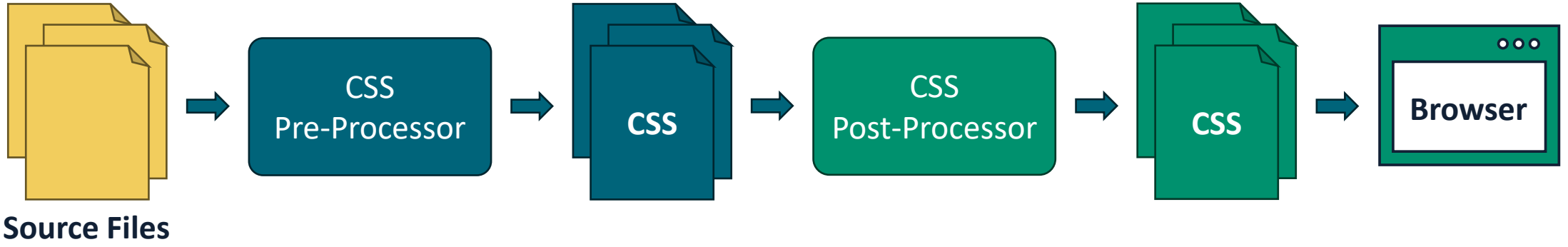
Agenda:

- CSS Pre-/Post-processors
  - SASS
- CSS Frameworks
- **Bundling** and **Minification**
- Development/Build servers

CSS PRE-/POST-PROCESSORS

CSS PRE-/POST-PROCESSORS

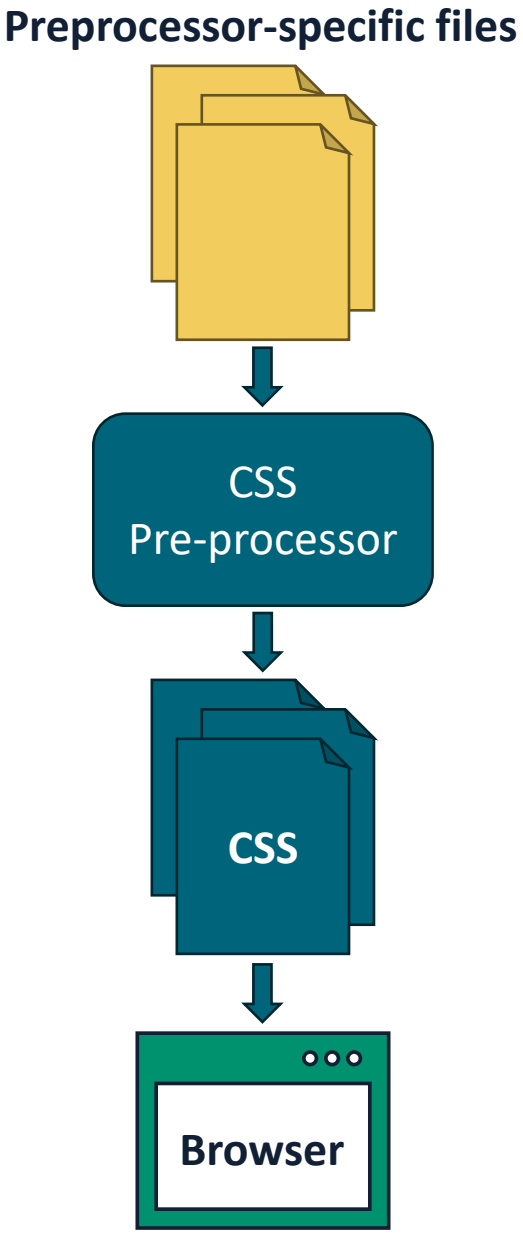
- CSS on its own can be fun, but stylesheets soon get **large**, **complex**, and **hard to maintain**.
- CSS Pre-/Post-Processing tools are essential in modern web development, and allow devs to efficiently creating **robust**, **scalable**, **optimized** stylesheets.



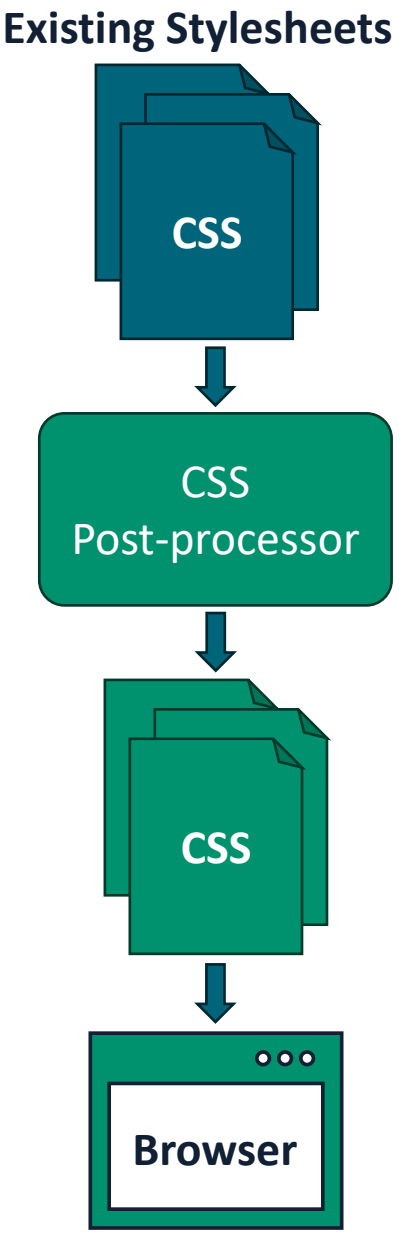
CSS PRE–PROCESSORS

CSS Pre-processors are special «compilers» that can generate CSS code starting from files written in their own specific syntax

- Pre-processor-specific syntax can be seen as an **extesion** of base CSS, introducing new features (e.g.: variables) to help write maintable stylesheets more efficiently
- Widely-used CSS processors include: Sass, LESS, Stylus…



CSS POST-PROCESSORS



Post-processors can automatically **process** and **transform** existing stylesheets, typically used for:

- **Downleveling** (see https://preset-env.cssdb.org/)

```
body {
  color: oklch(61% 0.2 29);
}
```

→

```
body {
  color: rgb(225, 65, 52);
}
```

- **Autoprefixing** (see https://autoprefixer.github.io/)

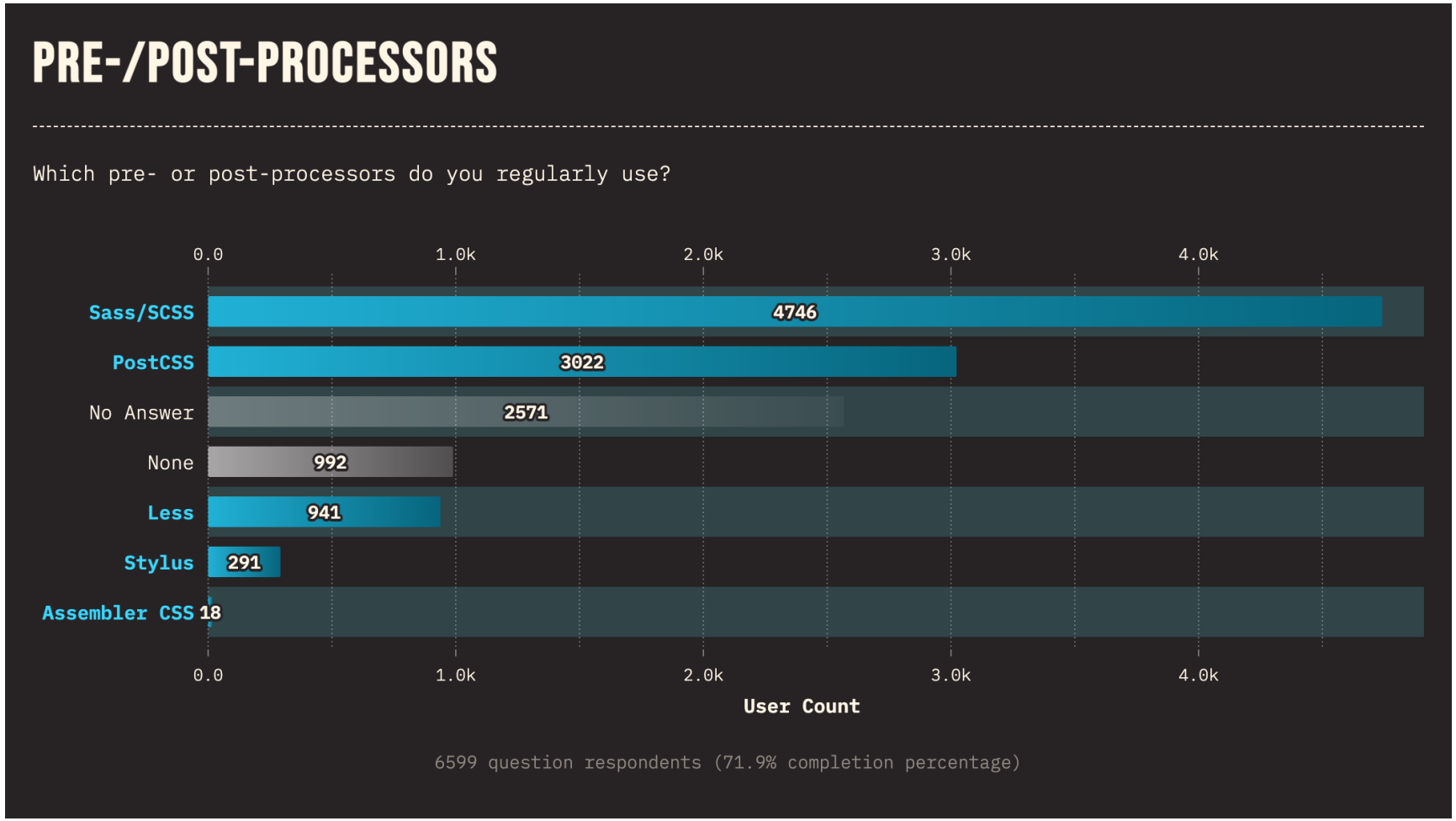
```
.example {
  transition: all .5s;
}
```

→

```
.example {
  -webkit-transition: all .5s;
  -o-transition: all .5s;
  transition: all .5s;
}
```

- **Optimizations** (e.g.: **minification** or delete unused parts)
- Enforce conventions and detect errors
- One of the most popular post-processors is PostCSS

CSS PRE-/POST-PROCESSORS: ADOPTION



State of CSS Survey 2023

SASS

- Self-defines as «*CSS with superpowers*»
- Currently the most popular pre-processor for CSS
- Been around since 2006 (that's almost 20 years ago!)
- Other preprocessors offer roughly the same features, with a slightly different syntax

INSTALLING SASS

- Install options are documented on the official website
- The easiest way to install SASS is to install it via **npm**

```
@luigi ➜ D/O/T/W/2/e/tooling $ npm install -g sass

added 17 packages in 5s
```

- This makes the **sass** binary available in the system path

```
@luigi ➜ D/O/T/W/2/e/tooling $ sass --help
Compile Sass to CSS.

Usage: sass <input.scss> [output.css]
       sass <input.scss>:<output.css> <input/>:<output/> <dir/>
```

SASS: TWO SYNTAXES

Sass supports two different syntaxes: **SCSS** and the **Indented Syntax**

SCSS Syntax

- Uses the **.scss** file extension
- It's a superset of CSS
- Most popular syntax

```scss
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

body {
  font: 100% $font-stack;
  color: $primary-color;
}
```

Indented Syntax

- Uses the .sass file extension
- Indentation-based, like Python
- Was the original syntax

```sass
$font-stack: Helvetica, sans-serif
$primary-color: #333

body
  font: 100% $font-stack;
  color: $primary-color;
```

SASS: VARIABLES

- Sass supports **variables**
- They are ways to **re-use** values throughout our spreadsheets
- When we compile a Sass file, variables are substituted by their value

```scss
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

body {
  font: $font-stack;
  color: $primary-color;
}
```

```css
body {
  font: Helvetica, sans-serif;
  color: #333;
}
```

```
@luigi ➜ D/O/T/W/2/e/tooling/sass $ sass file.scss file.css
```

SASS: NESTING

- In HTML, we have a clear nested element hierarchy
- In plain CSS, we cannot have nested rules
- Sass introduces support for **nesting**. It can make code more **readable**
- Too much nesting is a bad practice though! Leads to over-specific CSS!

```
.elem {
  color: red;
  span {
    color: blue;
    a {
      color: green; //applies to <a>s contained
    }
  }
}
```

```
.elem {
  color: red;
}
.elem span {
  color: blue;
}
.elem span a {
  color: green;
}
```

SASS: NESTING

```scss
/* nesting.scss */

nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

```css
/* nesting.css */

nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}

nav li {
  display: inline-block;
}

nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}
```

SASS: MIXINS

- Mixins are a way to re-use a group of CSS/Sass declarations
- Can be declared using the at-rule **@mixin** followed by an identifier
- Can then be included using the **@include** rule

```
@mixin clear-list {
  list-style-type: none;
  padding: 0; margin: 0;
  li {
    display: inline-block;
  }
}

nav ul {
  background-color: aliceblue;
  @include clear-list;
}
```

```
nav ul {
  background-color: aliceblue;
  list-style-type: none;
  padding: 0;
  margin: 0;
}
nav ul li {
  display: inline-block;
}
```

SASS: MIXINS

- Mixins can also optionally include one or more parameters

```
@mixin theme($color: DarkBlue) {
  background: $color;
  color: #fff;
}

.info {
  @include theme;
}

.danger {
  @include theme($color: DarkRed);
}
```

```
.info {
  background: DarkBlue;
  color: #fff;
}

.danger {
  background: DarkRed;
  color: #fff;
}
```

SASS: MODULES

```scss
/* base.scss */
$font-stack: Helvetica, sans-serif;
$primary-color: #333;

body {
  font: $font-stack;
  color: $primary-color;
}
```

```scss
/* example.scss */
@use 'base';

.inverse {
  background: base.$primary-color;
  color: white;
}
```

```css
/* example.css */
body {
  font: Helvetica, sans-serif;
  color: #333;
}

.inverse {
  background-color: #333;
  color: white;
}
```

SASS: PARTIALS

- Sass files starting with an underscore are called **partials**
- These files are meant to only be included by other files
- Sass will not generate a stand-alone CSS files for partials
- A Sass file including all variables could be a good example of a partial

```
/* _variables.scss */
$primary-color: purple;
$accent-color: orange;
$background-color: beige;
```

SASS: LISTS AND MAPS

- **Lists** are sequences of values separated by commas, spaces, or slashes as long as it is consistent across the list.
- **Maps** are sequences of key-value pairs.
  - Each pair has the form **<key>:<value>**.
  - Keys must be unique within a map.

```
$list: (120px, 240px, 480px, 800px);
$map:  ("xs": 480px, "md": 768px, "lg": 1024px);
```

SASS: CONTROL FLOW RULES

- Sass provides rules to control whether styles get emitted, or to emit them multiple times with small variations
- Think of it as a **template engine** for generating CSS files!
- Same idea as control flow constructs in template engines
  - **@if/@else** can be used to conditionally emit styles
  - **@each** can be used to iterate over all elements in a list/map
  - **@for** and **@while** have the usual semantics

SASS: @IF/@ELSE

```
@use "sass:math";

@mixin avatar($size, $circle: false) {
  width: $size;
  height: $size;

  @if $circle {
    border-radius: math.div($size, 2);
  }
}

.square-av {
  @include avatar(100px, $circle: false);
}
.circle-av {
  @include avatar(100px, $circle: true);
}
```

```
.square-av {
  width: 100px;
  height: 100px;
}

.circle-av {
  width: 100px;
  height: 100px;
  border-radius: 50px;
}
```

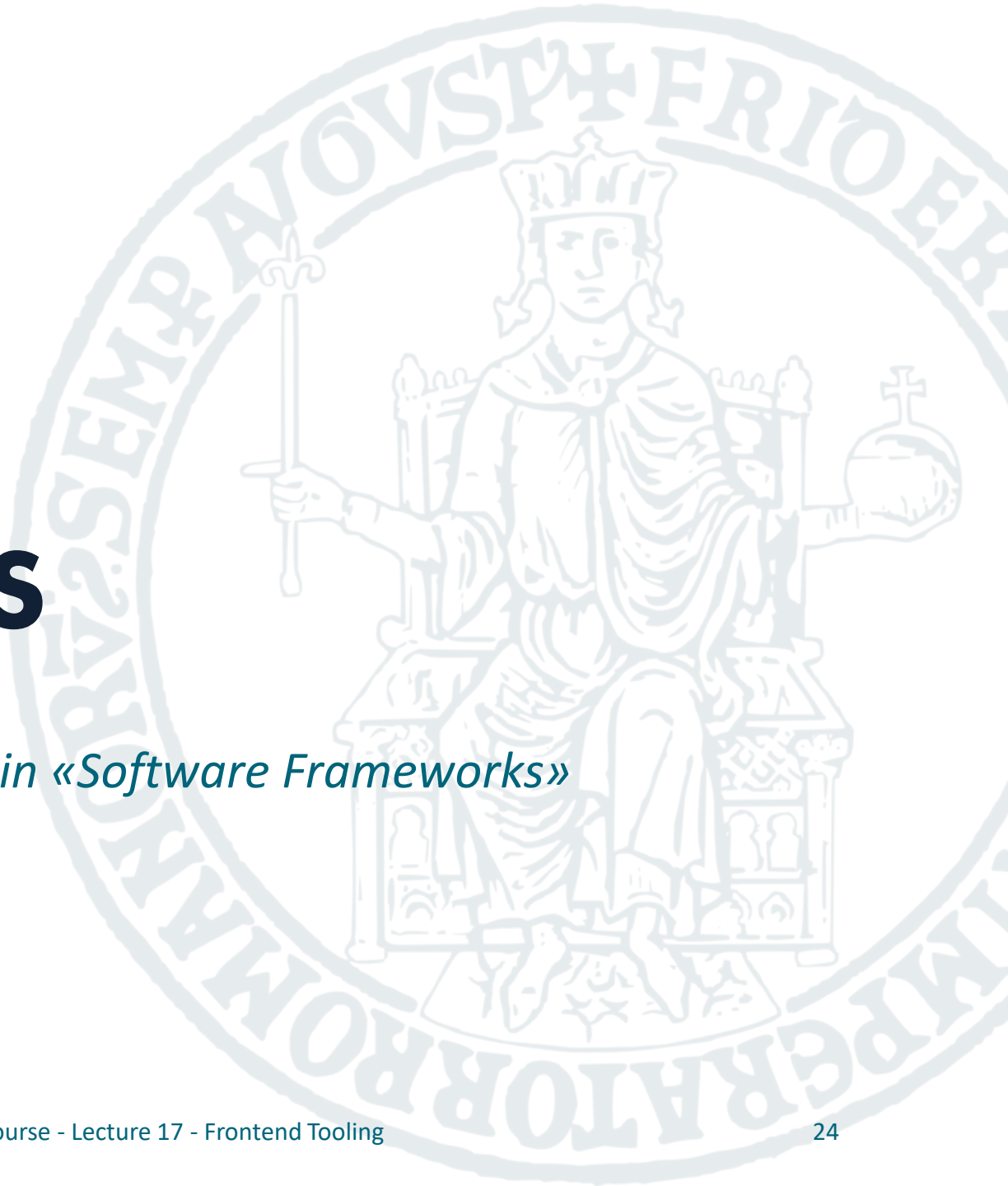
SASS: @EACH

```
$sizes: ("xs": 480px, "md": 768px, "lg": 1024px);

@each $sizeName, $sizeMaxWidth in $sizes {
  @media screen and (max-width: $sizeMaxWidth) {
    .hidden-#{$sizeName} {
      display: none;
    }
  }
}
```

```
@media screen and (max-width: 480px) {
  .hidden-xs {
    display: none;
  }
}
@media screen and (max-width: 768px) {
  .hidden-md {
    display: none;
  }
}
/*.hidden-lg omitted for the sake of brevity*/
```

CSS FRAMEWORKS

Well, these are not really «Frameworks» as in «Software Frameworks»



CSS FRAMEWORKS

- Also when writing CSS code for our front-end, we might find ourselves solving the same problems over and over again
  - Create a layout system
  - Create a consistent typography
  - Style our tables, list, UI-components
  - Reset user agent styles to get a consistent appearence on any browser
- Luckily, there is no need to start from scrach every time!
- Many CSS frameworks exist, providing a set of **general-purpose** styles
  - We can use the provided styles as-is, or we can do some fine-tuning on top!

CSS FRAMEWORKS

Widely-used CSS frameworks include:

- Bootstrap (the most popular framework, originally from Twitter)
- Tailwind (known for its **utility-first** approach)
- Materialize, Foundation, Bulma, PureCSS, …

USING CSS FRAMEWORKS

- Frameworks are a set of one or more CSS files
- We just need to import these CSS files in our pages, and use the classes provided by the framework (as described in the docs)
- We can download the CSS files and serve them from our own server
- Or we can use CDNs (Content Delivery Networks) that host the CSS files for us.
  - Quick and convenient for prototyping, but raises some privacy concerns…

BOOTSTRAP EXAMPLE

- Let’s look at a page featuring Bootstrap 5.3
- Live demo time!

Web Technologies Course

Features Enterprise Support Pricing

Pricing

Discover our competitive plans and master Web Technologies.

Free	Pro	Enterprise
$0/mo	$15/mo	$29/mo
5 hello-world examples included Learn about HTML4, CSS2, CGI Email support	20 hello-world examples included Learn about server-side frameworks Priority email support	100+ hello-world examples included Full content up-to Angular 17 and Vite Email and Teams support
Sign up for free	Get started	Contact us

Compare plans

	Free	Pro	Enterprise
Hello World Examples	5	20	100+
Modern Web Technologies included		✓	✓
Animated Gifs of Cats on a Computer			✓

CUSTOMIZING CSS FRAMEWORKS

- We might not want our website to look the same as a number of other websites built using the same framework.
- The pre-defined styles in the framework might not meet our needs for some particular aspects
- Most CSS frameworks are built using Sass or a similar tool. One way to customize a CSS framework is to change the build variables (e.g.: colors, fonts, sizes, etc…) according to our need, and then build the framework again.
- Another approach is to define custom CSS styles that override some of the framework's styles.

CUSTOMIZING BOOTSTRAP WITH SASS

- We can import the main bootstrap Sass file (or only some of its modules such as the grid system, the typography, etc…), and override some variables
- After compiling with Sass, we will get our own, custom Bootstrap flavor

```
//overriding Bootstrap variables

$primary:       #00647a;
$secondary:     rgb(254, 203, 110);

@import "../node_modules/bootstrap/scss/bootstrap.scss";
```

BOOTSTRAP EXAMPLE

- Let’s compile a custom version of Bootstrap
- Live demo time!

Web Technologies Course

Features Enterprise Support Pricing

Pricing

Discover our competitive plans and master Web Technologies.

Free	Pro	Enterprise
$0/mo	$15/mo	$29/mo
5 hello-world examples included Learn about HTML4, CSS2, CGI Email support	20 hello-world examples included Learn about server-side frameworks Priority email support	100+ hello-world examples included Full content up-to Angular 17 and Vite Email and Teams support
Sign up for free	Get started	Contact us

Compare plans

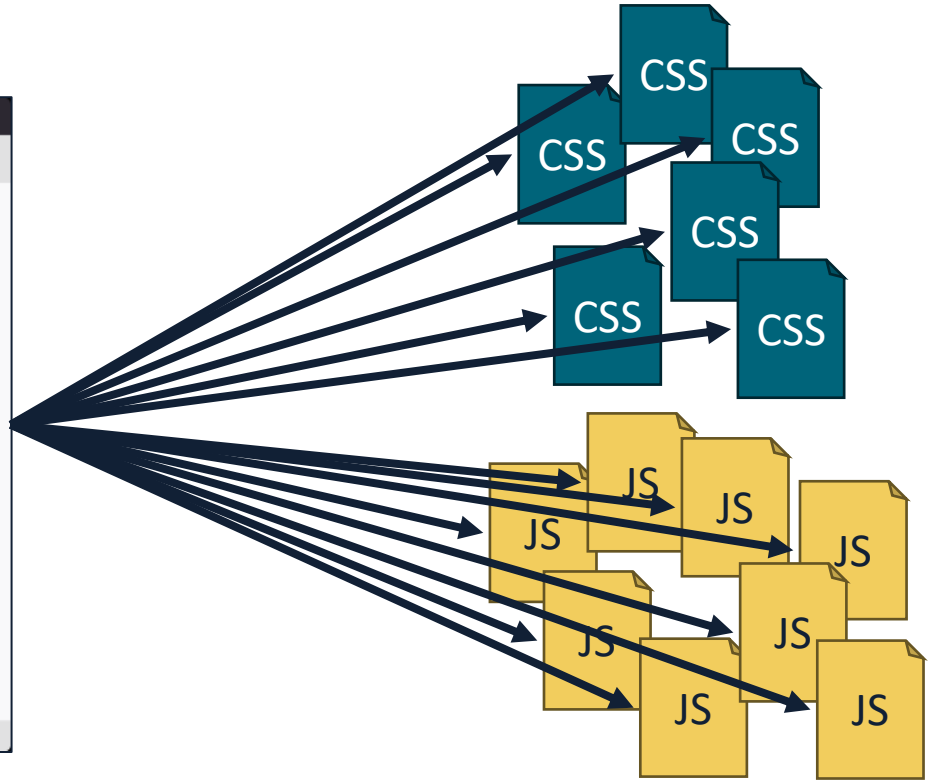
	Free	Pro	Enterprise
Hello World Examples	5	20	100+
Modern Web Technologies included		✓	✓
Animated Gifs of Cats on a Computer			✓

BUNDLING AND MINIFICATION

BUNDLING

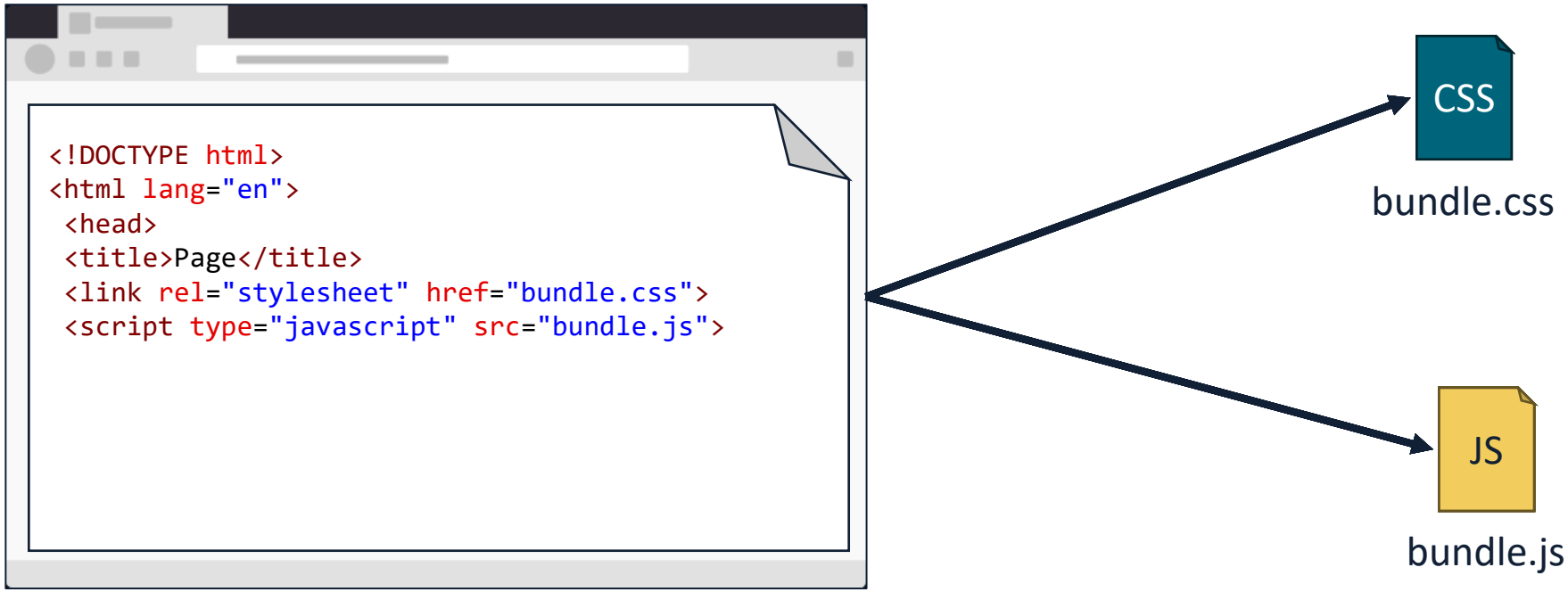
- Bundling combines multiple css or js files into a single one
- This way, browsers perform fewer HTTP requests to load a page

```
<!DOCTYPE html>
<html lang="en">
 <head>
 <title>Page</title>
 <link rel="stylesheet" href="a.css">
 <link rel="stylesheet" href="b.css">
 <!-- other stylesheets -->
 <script type="javascript" src="a.js"></script>
 <script type="javascript" src="b.js"></script>
 <script type="javascript" src="c.js"></script>
 <!-- other scripts -->
```



BUNDLING

- Bundling combines multiple css or js files into a single one
- This way, browsers perform fewer HTTP requests to load a page



BUNDLING

- Additional HTTP requests introduce **latency**
- Caching strategies can help on subsequent page loads
- Still, including many stylesheet/scripts has an impact on the first page load
- Bundling is a widely-adopted performance optimization strategy
- Beware: the order in which the single files are appended in the bundle matters!

BUNDLING: 'HISTORICAL' NOTE

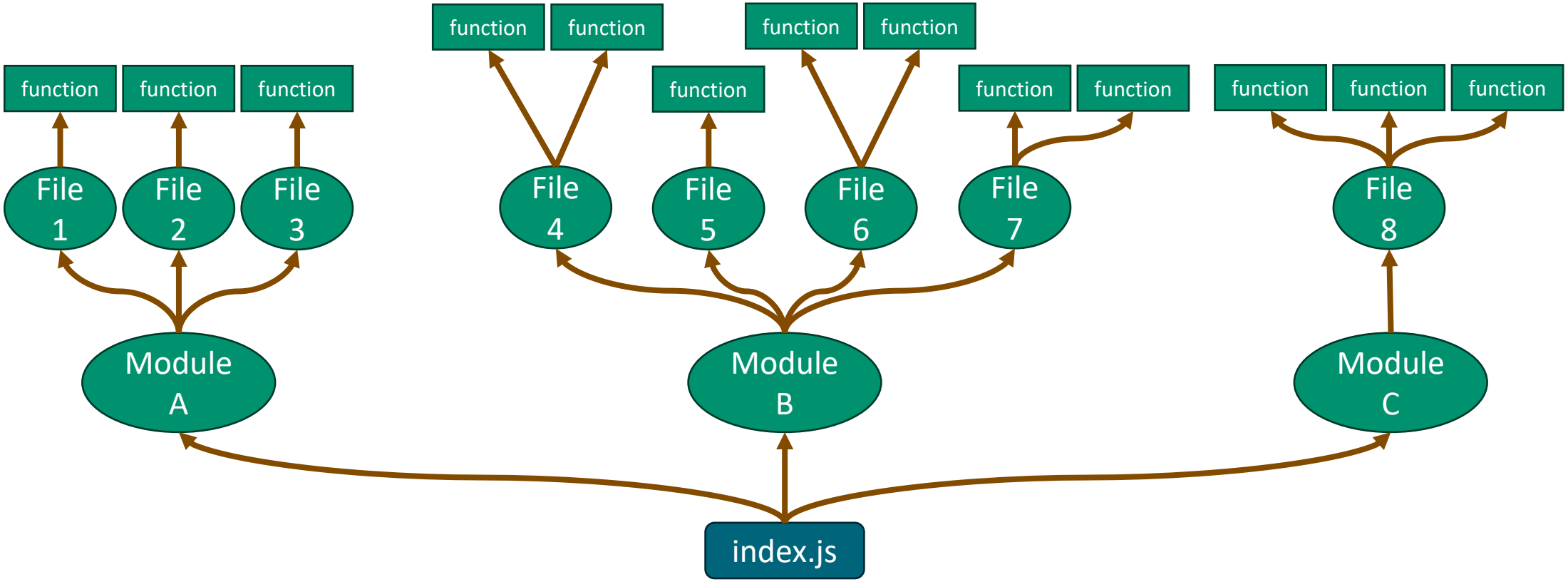
- Before ES modules were available in browsers, there was no way of writing modular JS code for browsers
- Bundlers (e.g.: tools like Webpack, Rollup, or Parcel) took care of concatenating all source modules in a way that browsers could understand

TREE SHAKING

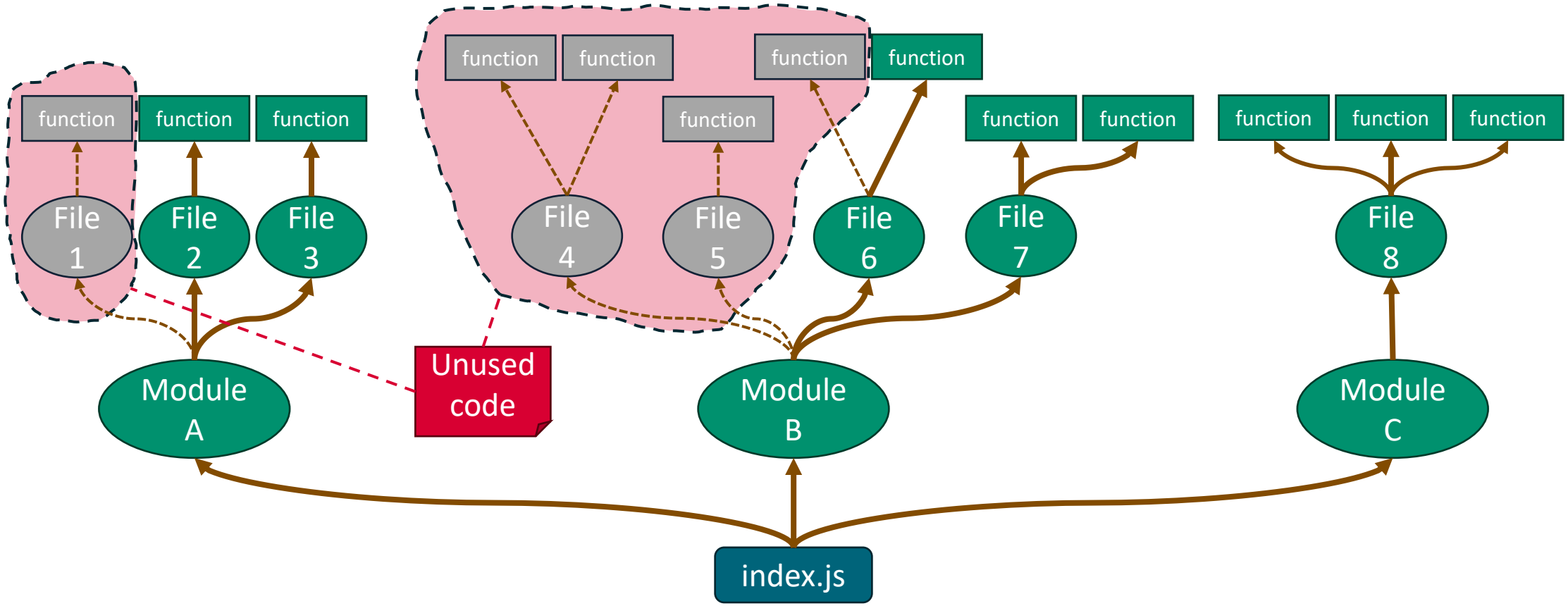
Bundlers often perform also **Tree Shaking** (i.e., dead code elimination)

- We often need only some of the functionality of the modules we import
- Modules can be quite large. Including an entire large module in the JavaScript code we distribute with our web pages, when we only use one of its functions, is ineffecient.
- Tree Shaking analyzes imports and export to prune out all unused code from the final bundle
- This can have a significant impact on bundle size (and thus on page load performance)!

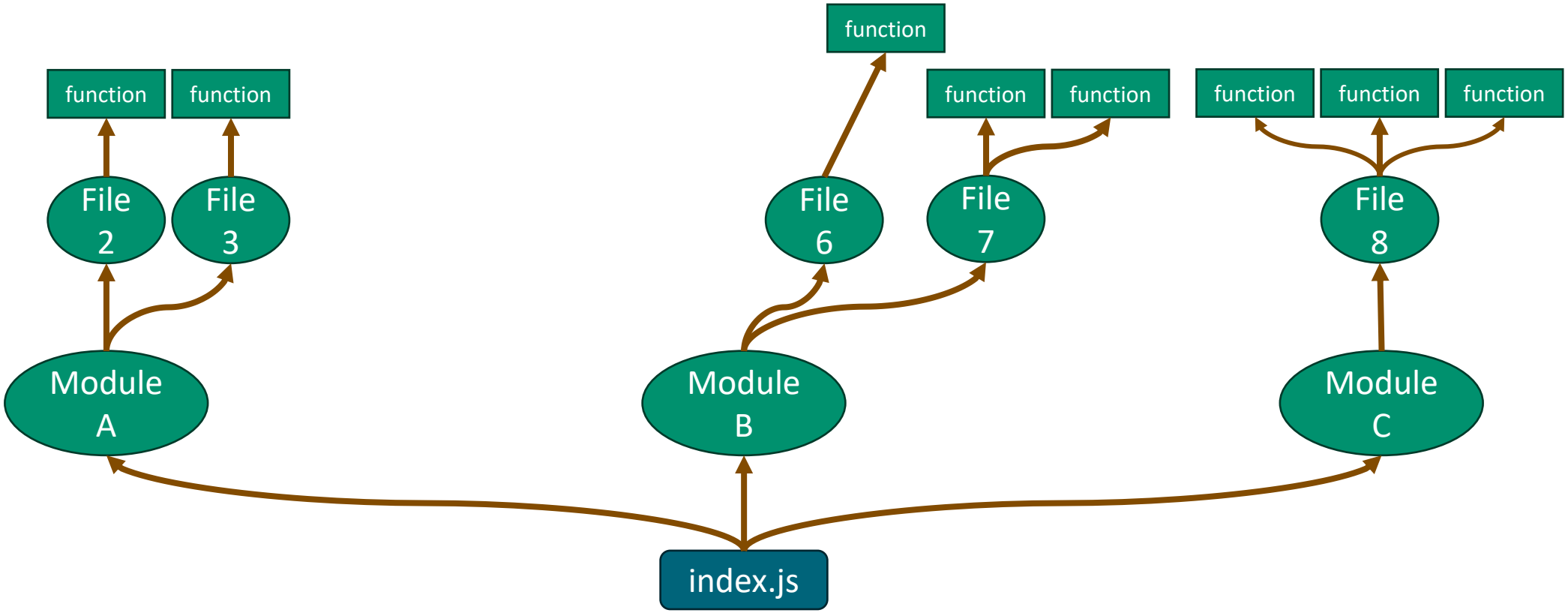
TREE SHAKING



TREE SHAKING



TREE SHAKING



MINIFICATION

- When humans write code, indentation and proper variable names are **crucial**
  - readability, understandability, maintainability…
- Browsers don’t really need readable code
- We could optimize file transfer by removing «unnecessary» whitespaces and shortening our source code without impacting its functionality
- This process is called **minification**, and the resulting compressed files are referred to as **minified**

A Steamroller «minifying» code.
Image generated by DALL-E 3

MINIFICATION: EXAMPLE

```
let msg = "Hello";
let name = "Web Technologies";
let str = `${msg}, ${name}!`;
let b = true;
if(b === true)
  document.getElementById('msg').innerHTML = str;
```

- 188 Bytes

```
let msg="Hello",name="Web
Technologies",str=`${msg},${name}!`,b=!0;!0===b&&
(document.getElementById("msg").innerHTML=str);
```

- 123 Bytes
- 34,57% compression
- Saved 65 Bytes

MINIFICATION: EXAMPLES

The effects of minification are greater the larger the source files

- **React** 18.2.0 weights ~85kB
- The minified React library weights ~25kB
- Compression rate: ~70%, saving ~60kB in file transfer size

- **Moment.js** 2.29.4 weights ~151kB
- The minified version weights ~75kB
- Compression rate: ~50%, saving ~76kB in file transfer size

DEVELOPMENT AND BUILD WITH VITE

VITE

- Pronounced /vit/ (like «veet»)
- A tool to support web developers in developing and deploying their projects
- Two key components:
  - A **development server** with a number of advanced features to streamline development
  - A **build tool** to create highly optimized static assets, ready for production

CREATING A VITE PROJECT

```
@luigi ➜ D/O/T/W/2/e/tooling/vite $ npm create vite@latest

✓ Project name: ... hello-vite
✓ Select a framework: » Vanilla
✓ Select a variant: » TypeScript

Scaffolding project in D/O/T/W/2/e/tooling/vite...

Done. Now run:

  cd hello-vite
  npm install
  npm run dev
```

LET'S TAKE A LOOK AT THE SCAFFOLDING

```
{
  "name": "hello-vite",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "preview": "vite preview"
  },
  "devDependencies": {
    "typescript": "^5.2.2",
    "vite": "^5.0.8"
  }
}
```

- **dev** starts the dev server.

  Vite watches for changes to the source files, and automatically reloads the web app when necessary (live reload).
- **build** generates an optimized bundle, ready for production
- **preview** is a simple static server, serving the files generated during the build phase.

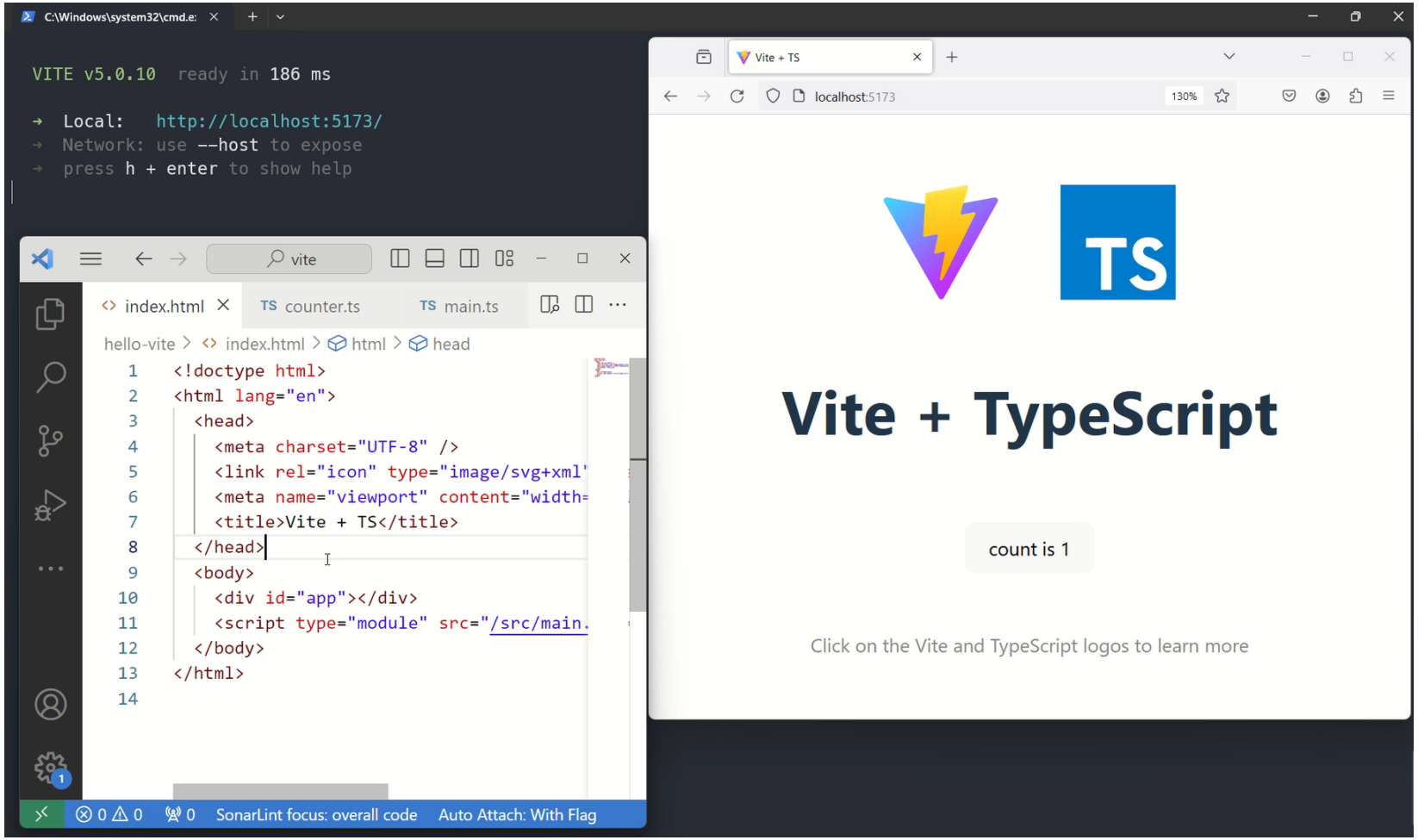
LET'S TAKE A LOOK AT THE SCAFFOLDING

- Let's take a look at the rest of the scaffolding code
- **Live demo time!**
- Will the lecturer be able to actually start the dev server, make a meaningful improvement to the web app, and build the whole app again?

STARTING THE DEV SERVER

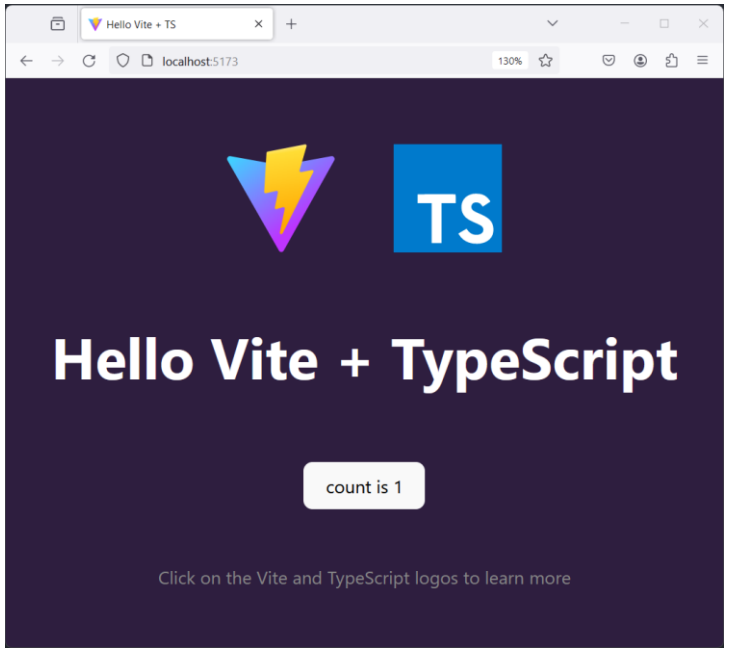
```
@luigi → D/O/T/W/2/e/tooling/vite $ cd .\hello-vite\
@luigi → D/O/T/W/2/e/t/v/hello-vite $ npm install

added 10 packages, and audited 11 packages in 10s

@luigi → D/O/T/W/2/e/t/v/hello-vite $ npm run dev

> hello-vite@0.0.0 dev
> vite

  VITE v5.0.10  ready in 781 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

STARTING THE DEV SERVER



ADDING SASS TO THE MIX

- Suppose we want to include an additional Sass stylesheet
- First, we need to install **sass** in this project dev dependencies
  - `> npm install -D sass`
- Then, we create a sass stylesheet file
- We let Vite know that that file needs to be included in the app
  - We can do that by importing it our **main.ts** file
- Vite automatically will detect a Sass file, compile it using sass, and bundle it after the original CSS file.



BUILDING OUR PROJECT

```
@luigi ➜ D/O/T/W/2/e/t/v/hello-vite $ npm run build

> hello-vite@0.0.0 build
> tsc && vite build

vite v5.0.10 building for production...
✓ 8 modules transformed.
dist/index.html                   0.46 kB │ gzip: 0.29 kB
dist/assets/index-v5kLGx6_.css    1.21 kB │ gzip: 0.63 kB
dist/assets/index-DiAK_ebR.js     3.05 kB │ gzip: 1.62 kB
✓ built in 173ms
```

BUILDING OUR PROJECT

During the build phase, Vite did quite a lot of work for us

- Analyzed what are the static resources we actually use in our app
- Compiled Sass (resp. TypeScript) files to CSS (resp. JavaScript) files
- Bundled Sass and CSS files in a single file
- Bundled JavaScript files in a single file
- Minified the bundle files
- Performed **filename fingerprinting** on the final bundles
- Moved all resources to the **/dist** directory, and updated the **index.html** file with correct imports for the bundles

FILENAME FINGERPRINTING

- The final bundles produced by Vite have a peculiar name:
  - `index-v5kLGx6_.css`, `index-DiAK_ebR.js`
- The last part of the filename is an **hash** of their current contents
- This way, when we build a new version of the app, with some changes to the css or js files, the filename will change
- This techniques can be used to avoid **cache staleness** issues
  - Suppose the files were called simply **index.css** and **index.js**
  - A browser visits our page one minute before we roll out a new version
  - That browser might keep using the old **index.css** and **index.js** files it stored in its cache for some time

REFERENCES

- **Client-side Tooling Overview**
  MDN web docs
  https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Overview
- **Sass: Guide**
  https://sass-lang.com/guide/
- **Bootstrap: Getting Started**
  https://getbootstrap.com/docs/5.3/getting-started/introduction/
- **Get started with Tailwind CSS**
  https://tailwindcss.com/docs/installation
- **Vite: Official Guide**
  https://vitejs.dev/guide/