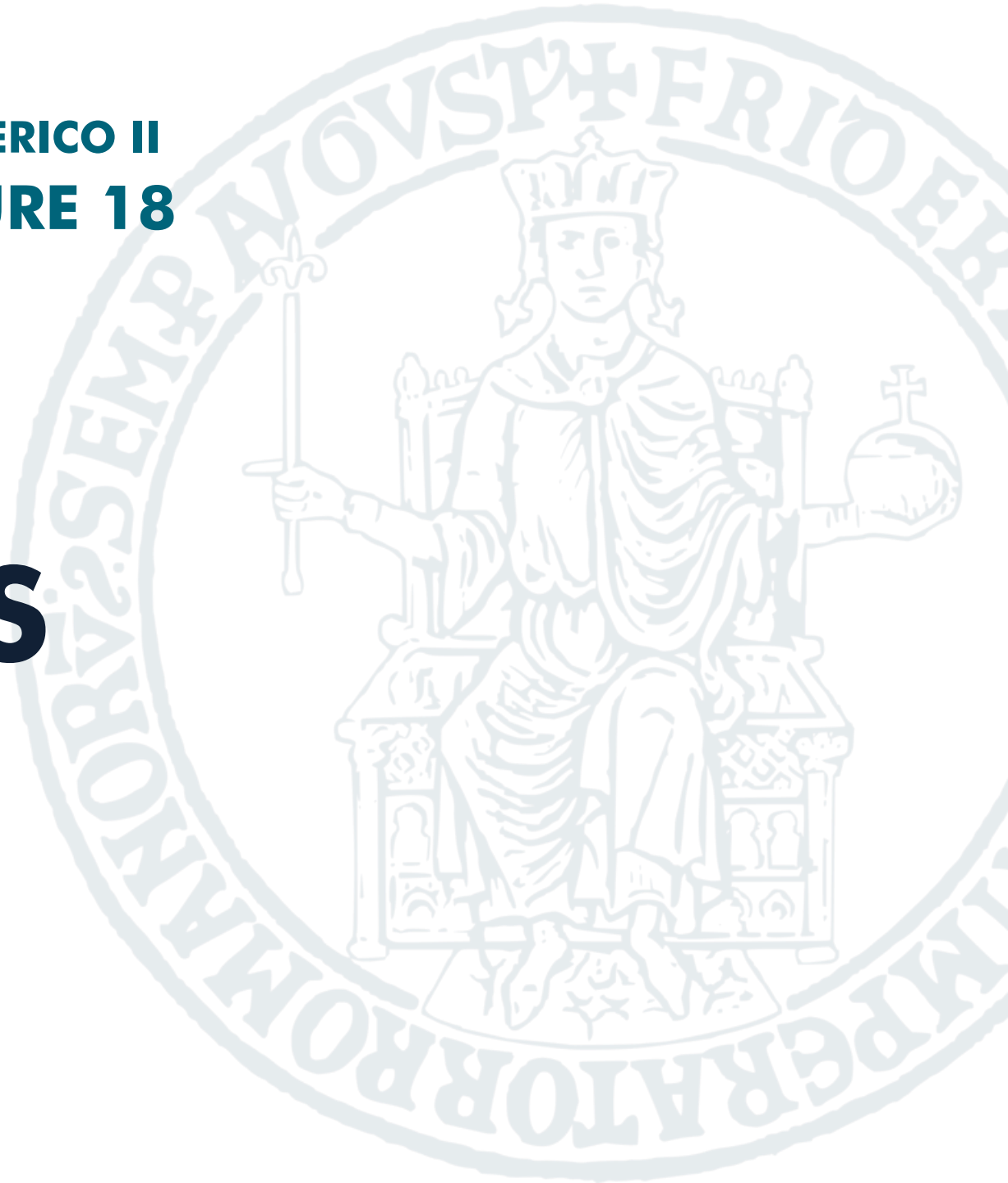
UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

WEB TECHNOLOGIES — LECTURE 18

# SINGLE PAGE APPLICATIONS




Luigi Libero Lucio Starace, PhD

luigiliberolucio.starace@unina.it

https://luistar.github.io

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

# PREVIOUSLY, ON WEB TECHNOLOGIES

We learned how to develop **traditional web applications**

- Using CGI, PHP, and plain Node.js
- And using a Framework (Express)

In the web apps we developed, pages are **dynamically generated on the server** and sent to the web browser for rendering.

- Once they reach the browser, web pages are inherently **static**.

# MULTI–PAGE WEB APPLICATIONS

In our traditional web apps, when we change page (e.g.: click on a link):

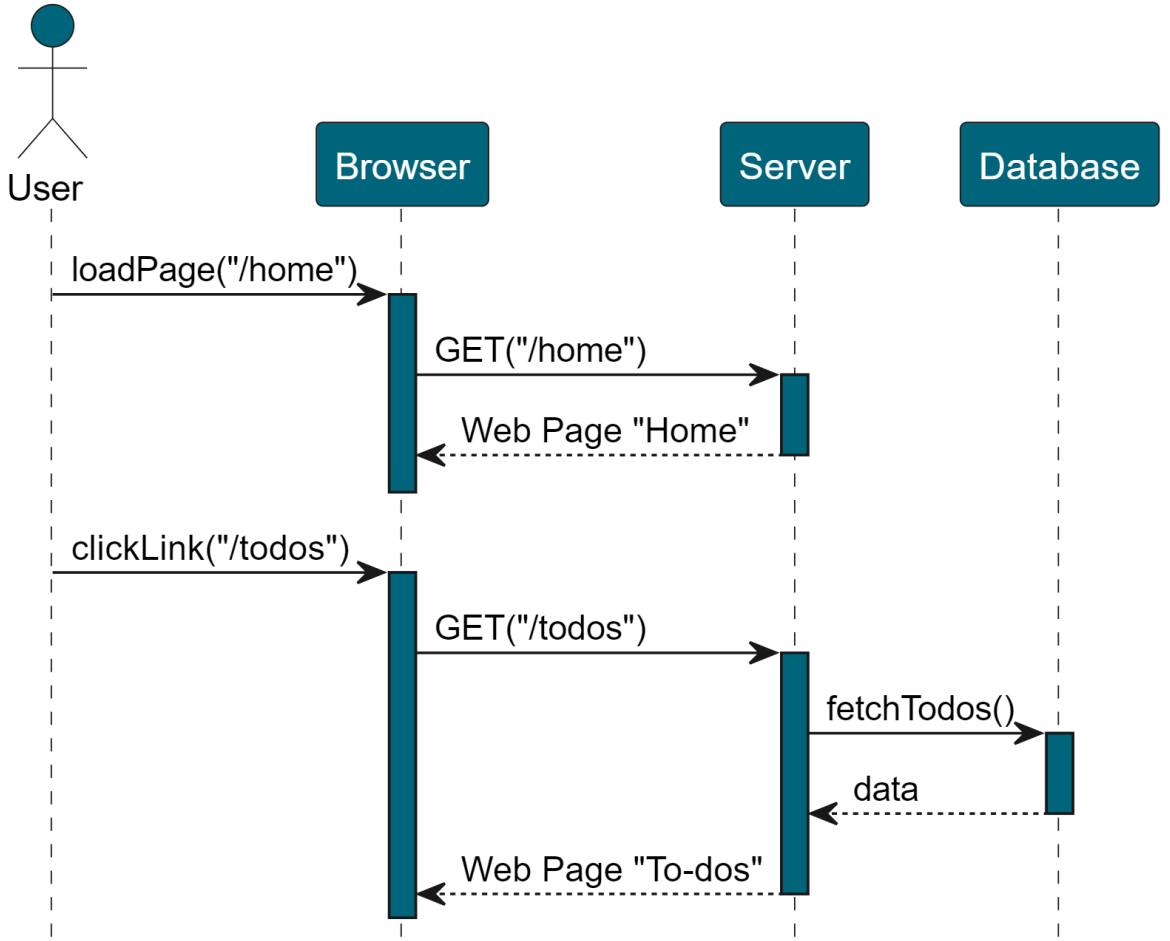
- A **new request** is sent to the server
- The **new page** is returned in response and **displayed** by the browser

This approach is also known as the **multi-page** approach, leading to **multi-page web applications**.

Back in **Lecture 7**, we learned about JavaScript in a Browser environment: DOM manipulation, fetching data, …

- So far, however, we did not make particular use of those features

# MULTI–PAGE APPLICATIONS

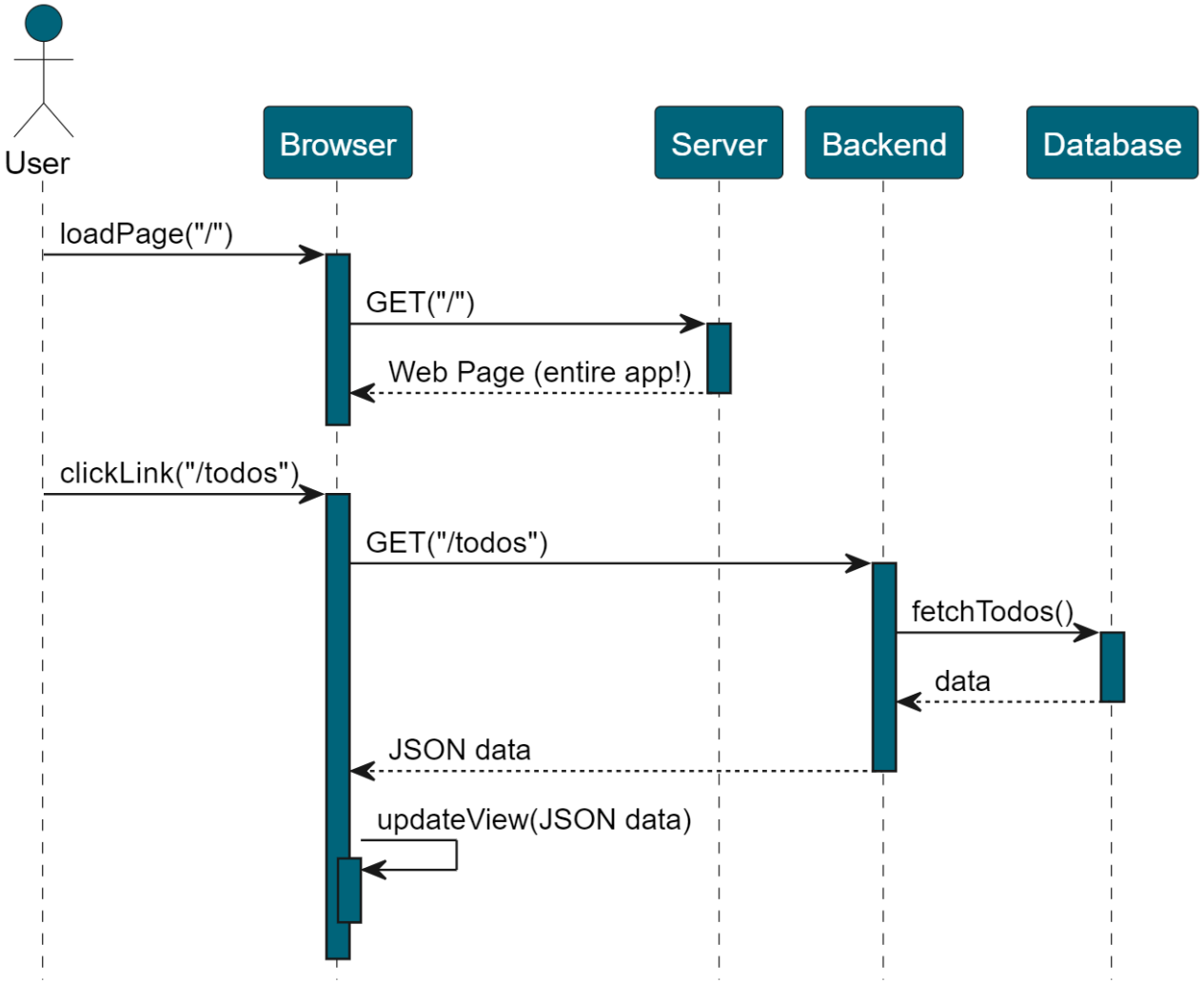



Navigating to a different page always triggers a full-reload of the web page

# SINGLE–PAGE WEB APPLICATIONS

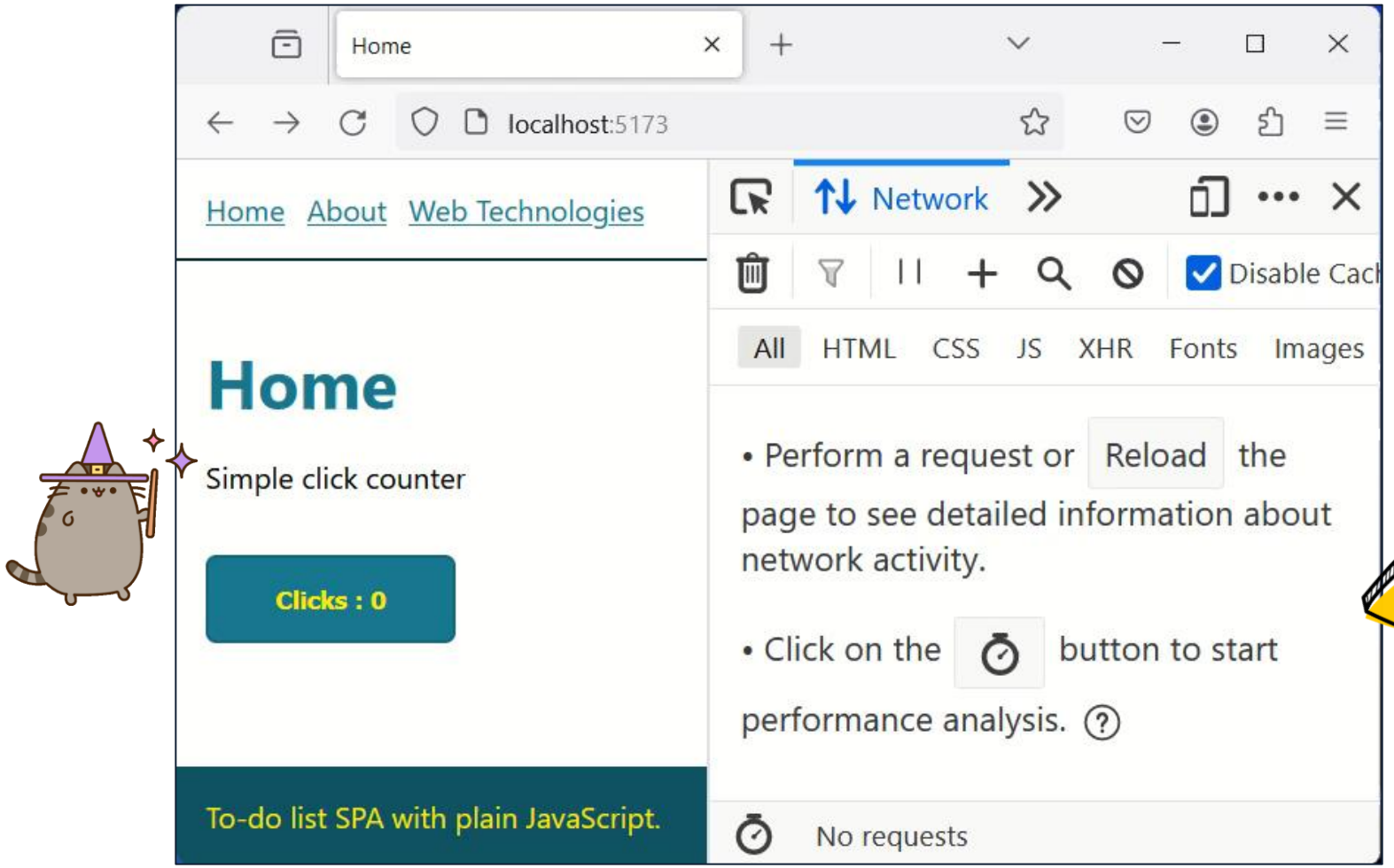
- A **Single-Page Application** (SPA) reacts to user input by **dynamically updating the content of the current page,** rather than loading a new page from the server.
- The **entire** application is loaded in the initial request, and then the page is never reloaded
- The complexity of view rendering is **shifted** from the server to the client-side code (thin clients to thick clients)
- JavaScript-intensive applications

# SINGLE–PAGE APPLICATIONS




- Navigating to a different page does not trigger a full page reload.
- Required data is fetched from a backend (e.g.: REST)
- Client-side JavaScript dynamically creates the new web page

# SINGLE-PAGE APPLICATIONS IN ACTION



# CLIENT–SIDE ROUTING

- Different «pages» in a SPA are typically referred to as **views**
- In SPAs, navigation and view updates are managed entirely on the client-side, without reloading the entire page from the server
- This practice is called **client-side routing**
- Client-side routing involves:
  - Updating the Browser url
  - Loading and rendering the appropriate content (**view**) on the web page
  - Interact with the Browser History API to allow users to seamlessly use the back and forward buttons to navigate between different views

# CLIENT–SIDE ROUTING

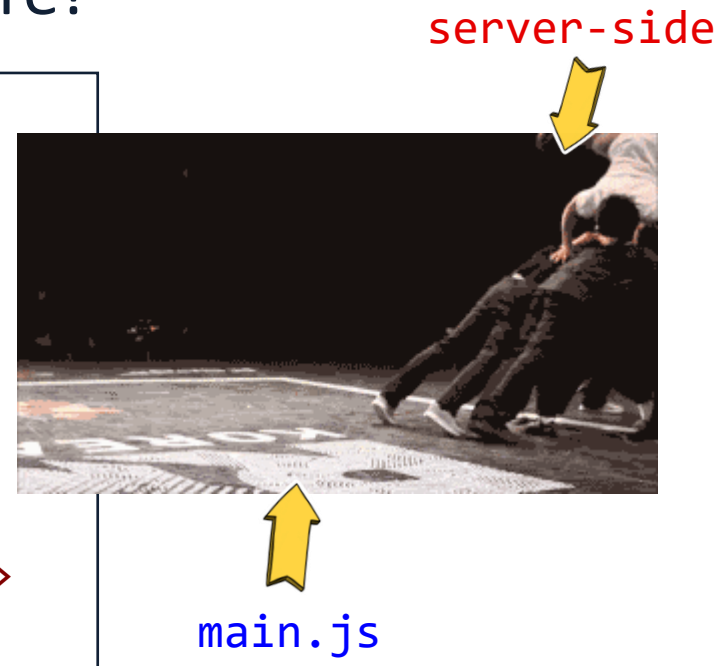
Key components of client-side routing include:

- **Router:** module that handles the mapping between URLs and Views
- **View Rendering:** a mechanism to render different Views
- **History Management:** a mechanism to use the History API to manipulate the browser's navigation history.
  - This is because we want users to interact with SPAs as with any other Multi-page app, and the back and forward buttons should work as intended.

# OUR FIRST SINGLE PAGE APPLICATION

- There is a single HTML document, shown below
- It does not include any content at all, just an empty «app» !
- main.js is the one that does the heavy lifting here!

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Single Page Application</title>
    <link rel="stylesheet" href="src/css/style.scss"/>
  </head>
  <body>
    <div id="app"></div>
    <script type="module" src="./src/js/main.js"></script>
  </body>
</html>
```



# TAKING A LOOK AT MAIN.JS

- The main.js script starts by creating three new elements in the app `<div>` container
- A navbar, a container for the main content, and a footer
- The navbar and the footer won't change when switching between different views, while the content will change.

```
let nav = document.createElement("nav");
nav.id = "nav";
let content = document.createElement("main");
content.id = "content";
let footer = document.createElement("footer");
footer.id = "footer";

app.append(nav, content, footer);
```

# TAKING A LOOK AT MAIN.JS

- Then, the **routes** in the app are defined.
- Each route associates a view to a given URL
- Each route is indexed by its URL path, and is associated with
  - **title:** will be title of the page when that view is loaded
  - **render:** a function returning an HTML string rendering the view

```
import about from "./views/about.js"; /*other imports omitted for brevity*/

const routes = {
    "/": { title: "Home", render: home },
    "/about": { title: "About", render: about },
    "/webtech": { title: "Web Technologies", render: contact },
};
```

# TAKING A LOOK AT VIEWS/ABOUT.JS

- View rendering is as simple as possible
- A function returning an HTML string to be rendered for that view
- It may get way more complex!
- For a starter, we may use a **template engine**!

```
export default () => /*html*/`
    <h1>About</h1>
    <p>
       This is a single page app. Navigating to different pages does not trigger
       a full-page reload!
    </p>
`;
```

# TAKING A LOOK AT MAIN.JS: ROUTER FUNC

- Then, we define a router function

```
function router() {
  let view = routes[location.pathname]; //get route corresponding to current path

  if (view) { //if a route matches the current path
    document.title = view.title; //update the page title to the one of the route
    content.innerHTML = view.render(); //update the content with the result of
                                       //the render function
  } else { //no route matches the path
    history.replaceState("", "", "/"); //replace current history entry with an
                                       //emtpy state and change url path to /
    router(); //call router() again
  }
};
```

# TAKING A LOOK AT MAIN.JS

- Then, we populate the nav and footer elements a router function
- **renderFooter()** is a function returning an HTML string

```
// Add Navigation links to the page
for(let route in routes){
    let link = document.createElement("a");
    link.href = route;
    link.innerText = routes[route].title;
    link.setAttribute("data-link",""); // notice we're adding a custom attribute!
    nav.append(link);
}

// Render footer
footer.innerHTML = renderFooter();
```

# TAKING A LOOK AT MAIN.JS: NAVIGATION

- In a SPA, we might have links that should behave normally (e.g.: links to external websites), and links that should just trigger a view update
- In our example, we identify the latters with the `data-link` attribute

```
// Handle navigation
window.addEventListener("click", e => {
  if (e.target.matches("[data-link]")) {
    e.preventDefault(); // do not trigger page reload
    history.pushState("", "", e.target.href); // add new state on history stack
    router(); // call router to update the view according to current url path
  }
});
```

# TAKING A LOOK AT MAIN.JS: NAVIGATION

Last, we take care of executing the **router()** function to update the view to match the current URL path

- Every time the user clicks on the «back» button in the browser
  - popstate event on the Window interface
- Every time the main web page is fully reloaded (e.g.: if the user presses the refresh button)
  - DOMContentLoaded event

```
// Update router
window.addEventListener("popstate", router);
window.addEventListener("DOMContentLoaded", router);
```

# INTRODUCING COMPONENTS

- As the complexity of front-ends grows, managing entire views as **monoliths** becomes rapidly unfeasible
  - Codebase gets hard to maintain
  - It’s difficult to **re-use** pieces of functionalities in different views
- Decomposing the UI into modular components is the way to go
- Each component encapsulates a specific functionality
  - A component should be the sole responsible for **rendering its UI (HTML)** and managing its **internal state**
  - Views using a component should not bother about these details
- Components can then be easily re-used in multiple views

# COMPONENTS

Components play a central role in all front-end frameworks (we'll see!)

- Key characteristics of components include:
  - **Modularity:** they should be self-contained, and **encapsulate** a specific piece of functionality
  - **Encapsulation:** they should encapsulate their own business logic, state, layout and styles
  - **Reusability:** they should be designed to be re-usable across different parts of the app

# CREATING A COMPONENT FROM SCRATCH

```
// New component
class Counter extends HTMLElement {
  constructor() {
    super();
    this.innerHTML = `<button>Clicks : ${count}</button>`;
    let btn = this.querySelector("button");
    // State
    btn.onclick = () => {
      btn.innerHTML = "Clicks : " + ++count;
    };
  }
}

let count = 0;

customElements.define("click-counter", Counter);
```

We're defining a new **HTMLElement** called `<click-counter>`, associated with the `Counter` class, using the Web Components API!

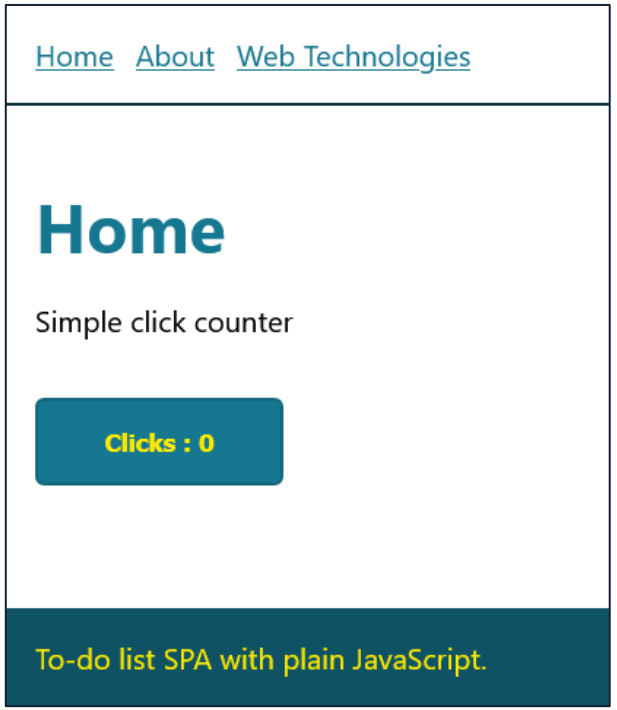
The browser will use the Counter class when a `<click-counter>` element is found in the DOM!

# USING A COMPONENT

```
import "../components/counter.js";

export default () => /*html*/`
    <h1>Home</h1>
    <p>Simple click counter</p>
    <click-counter></click-counter>
`;
```

```
class Counter extends HTMLElement {
  constructor() {
    super();
    this.innerHTML = `<button>Clicks : ${count}</button>`;
    let btn = this.querySelector("button");
    btn.onclick = () => {
      btn.innerHTML = "Clicks : " + ++count;
    };
  }
}
```



# WHY SINGLE–PAGE APPLICATIONS?

SPAs offer some advantages over traditional multi-page apps:

- More **dynamic user experience**, similar to desktop apps
- The initial page load is slower (need to download the entire app), but subsequent ones are way faster
- **Reduced server load**
  - Only required data is fetched from the server, and only when needed.
  - No need to transfer entire HTML documents over and over
  - This can help reduce server loads and bandwidth

# A NOTE ON FETCHING DATA

- Before the Fetch API was introduced in ES6, network requests were sent using AJAX
- **AJAX** stands for **A**synchronous **J**avaScript **A**nd **X**ML
- AJAX was implemented using the **XMLHttpRequest** interface
- While XMLHttpRequest is nowadays being replaced by the Fetch API, it is still quite common and you may find it in some packages

# EXAMPLE USING XMLHTTPREQUEST

- Typical workflow consisted in creating and XMLHttpRequest object
- Setting a callback to execute when the request completes
- Initialize a request (set HTTP method, url, whether it is async or not)
- Send the request to the server

```
let xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
       // Typical action to be performed when the response is ready:
       document.getElementById("demo").innerHTML = xhttp.responseText;
    }
};
xhttp.open("GET", "/url/to/fetch", true);
xhttp.send();
```

# THE TO-DO LIST SINGLE PAGE APP

*Developed from scratch using plain JavaScript!*

# THE TO–DO LIST SINGLE PAGE APP

- Let’s implement our To-do List App as a SPA!
- We’ll use the REST API we developed in the earlier lectures
- And we’ll add a sleek SPA frontend, written in plain JavaScript
- **Live demo time!**

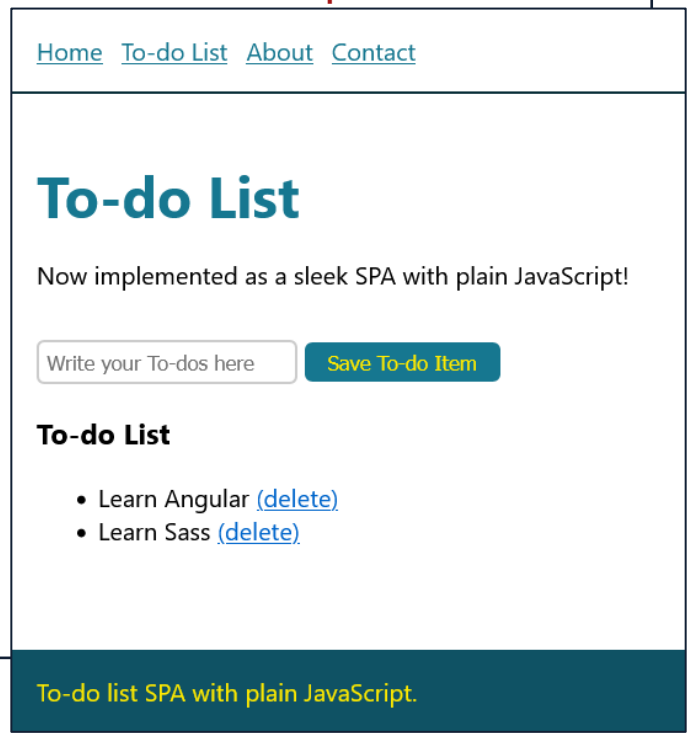
# THE TO-DO LIST COMPONENT

```
class TodoList extends HTMLElement {
  list = [];
  constructor() {
    super();
    this.innerHTML = /*html*/;
    this.list = this.fetchTodoItems().then( (res) => {
      this.updateTodoList(res);
    });

    let btn = this.querySelector("#save-todo");
    let input = this.querySelector("#todo-input");
    btn.onclick = () => {/*...*/};
  }
  async fetchTodoItems(){/*...*/}
  updateTodoList(list){/*...*/  }
  async saveTodo(toSave){/*...*/}
}
```

# THE TO-DO LIST COMPONENT

```
constructor() {
    super();
    this.innerHTML = /*html*/`
      <input id="todo-input" placeholder="Write your To-dos here"></input>
      <button id="save-todo">Save To-do Item</button>
      <h3>To-do List</h3>
      <ul id="the-list"></ul>
    `;
    this.list = this.fetchTodoItems().then( (res) => {
      this.updateTodoList(res);
    });
    let btn = this.querySelector("#save-todo");
    let input = this.querySelector("#todo-input");
    btn.onclick = () => {/*...*/}; //handle save todo
  }
```



# CHALLENGES WHEN DEVELOPING SPAs

Implementing the To-do List App as a SPA from scratch was educational Despite its simplicity, we started to experience some challenges

- **State management**. We needed to keep state and UI always updated and make sure to properly update the views after any state change
- **DOM manipulation**. Manipulating the DOM with the document API requires lots of boilerplate code
- **Client-side Routing.** Our router was pretty basic. What if we were to support parametric routes, redirects, etc.?
- **Efficient rendering.** We just re-draw our entire to-do list every time. What if we wanted to be more efficient?

# FRONTEND FRAMEWORKS

- Frontend frameworks like Angular, React, Vue.js, Ember, Svelte, etc., provide some ready-made solutions to tackle these challenges
- Frameworks also typically include tooling support
  - Generation of boilerplate code
  - Automate build and deployment
- Each framework also provides a somewhat «standardized» way of building frontends
  - A developer who is used to working with a given framework will have less issues moving to a different project using the same framework!
- In the next lecture, we'll get to know **Angular 17**!

# REFERENCES

- **SPA (Single-page application)**
  MDN web docs
  https://developer.mozilla.org/en-US/docs/Glossary/SPA
- **Single page apps in depth**
  By Mikito Takada
  Available at http://singlepageappbook.com/ and on GitHub
  **Relevant parts:** Modern single page apps – an overview
- **Understanding client-side JavaScript frameworks**
  MDN web docs
  https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks
  Relevant parts: Introductory guides: Introduction to client-side frameworks and Framework main features.