VANILLA JS & DOM

YOU DON'T NEED THAT LIBRARY

MAXIMILIANO FIRTMAN

MAXIMILIANO FIRTMAN

MOBILE+WEB DEVELOPER

HTML since 1996

JavaScript since 1998

AUTHOR

Authored 13 books and +70 courses

Published +150 webapps

@FIRT · FIRT.DEV

What we'll cover

- Vanilla JS
- DOM API
- Fetch
- Design Patterns
- Single Page Applications
- Web Components
- Reactive Programming
- Routing

Pre-requisites

firtman.github.io/vanilla

Questions?

You don't need that library

But I love

React

But I love

Angular

But I love

Vue

But I love

Svelte

I want you to also love

Vanilla JS

DEFINITION

Vanilla JavaScript

The usage of the core language and browser APIs to create web apps without any additional libraries or frameworks added on top

IMPORTANT

Vanilla JavaScript has been as a concept for a decade now

Why do we need to care about VanillaJS?

- Add one more tool to your toolbox
- Understand what your library is doing
- Extend your library with plugins
- Add be a better web developer
- To mix with libraries
- **To use it!**

  You can create simple and fast webapps with no CLI, no build process

Ben Holmes 
@BHolmesDev

Genuinely think every frontend dev should build their own framework at some point

Ben Holmes  @BHolmesDev · 11h

Building @slinkitydotdev taught me more JS and web APIs in a month than years of React ever did. ES modules, SPA routing from scratch, bundling strategies... this is the good stuff 🏆

Main Advantages of Vanilla JS

- Lightweight
- Control and Power
- Simplicity
- Flexibility
- Performance
- Compatibility
- No node-modules 🤣

Main Fears of Vanilla JS

- Routing for Single Page Applications
- Too Verbose and Time Consuming
- State Management
- Templating
- Complexity
- Reusable Components
- Maintenance
- Learning Curve
- Browser Compatibility
- Reinventing the Wheel every time
- Scalability

IMPORTANT

Learn the tool and use it when it's the best option.

We won't advocate for using Vanilla JS on every project.

WARNING

In this workshop, we will be using different techniques and design patterns in parallel.

DOM Essentials

DEFINITION

DOM

The Document Object Model connects web pages to JavaScript by representing the structure of a document in memory

DEFINITION

DOM API

A browser API exposed to developers to manipulate the DOM from an scripting language.

The DOM API is available on many objects

`window` global object

`document` object

One object per HTML element and other nodes in your document

```
<html>

  <body>

    <header>

      <h1>The DOM</h1>

      <p class="tagline">
       Document Object Model
      </p>

      <img hidden src="dom.png">

    <header>

  </body>

</html>
```

```
<html>

  <body>

    <header>

      <h1>The DOM</h1>

      <p class="tagline">
       Document Object Model
      </p>

      <img hidden src="dom.png">

    <header>

  </body>

</html>
```

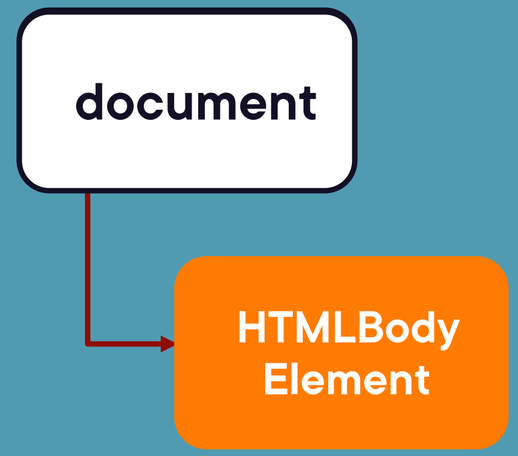
- Each element is represented by an object of **HTMLElement** interface or other interface that inherits it

- They have instance properties and methods

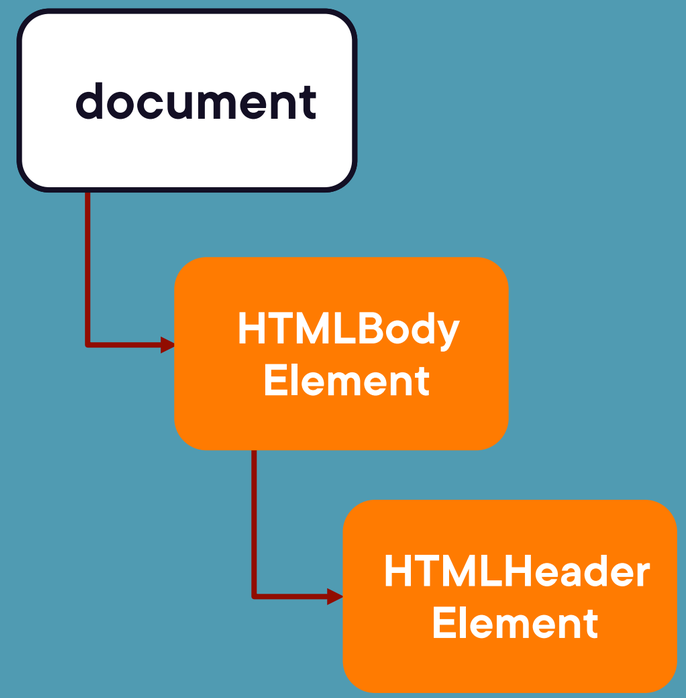
- Changes in properties or children will trigger updates in the user interface **when you release the thread**.

- We can listen to events happening in that element and react in consequence

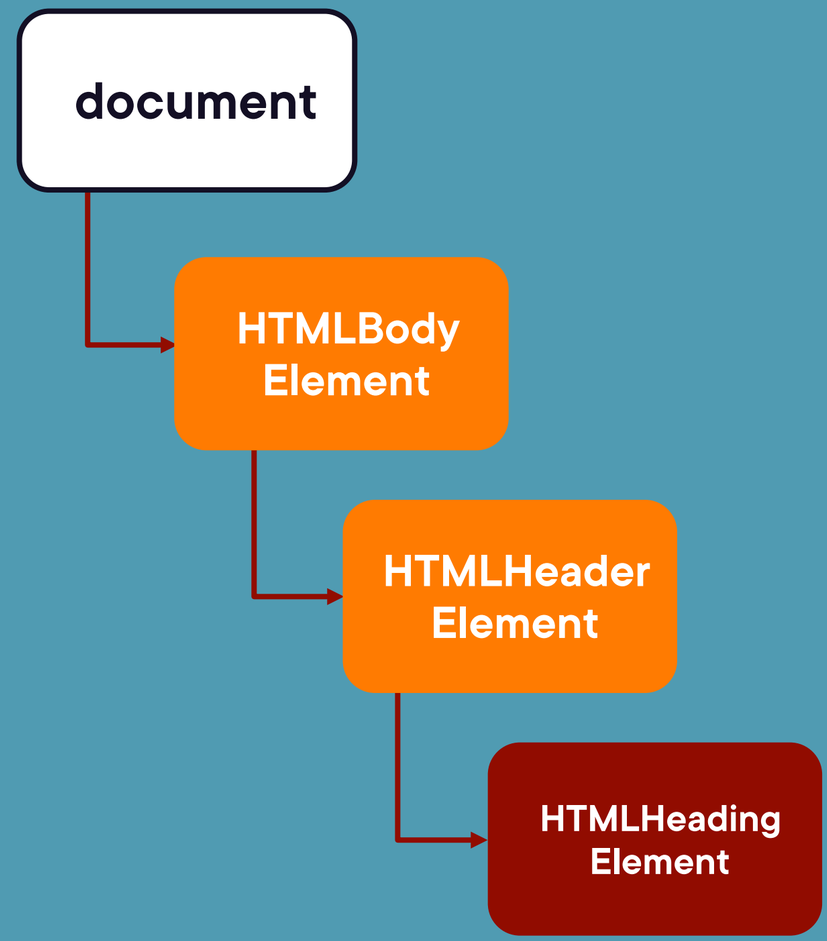
```
<html>

  <body>

    <header>

      <h1>The DOM</h1>

      <p class="tagline">
       Document Object Model
      </p>

      <img hidden src="dom.png">

    <header>

  </body>

</html>
```



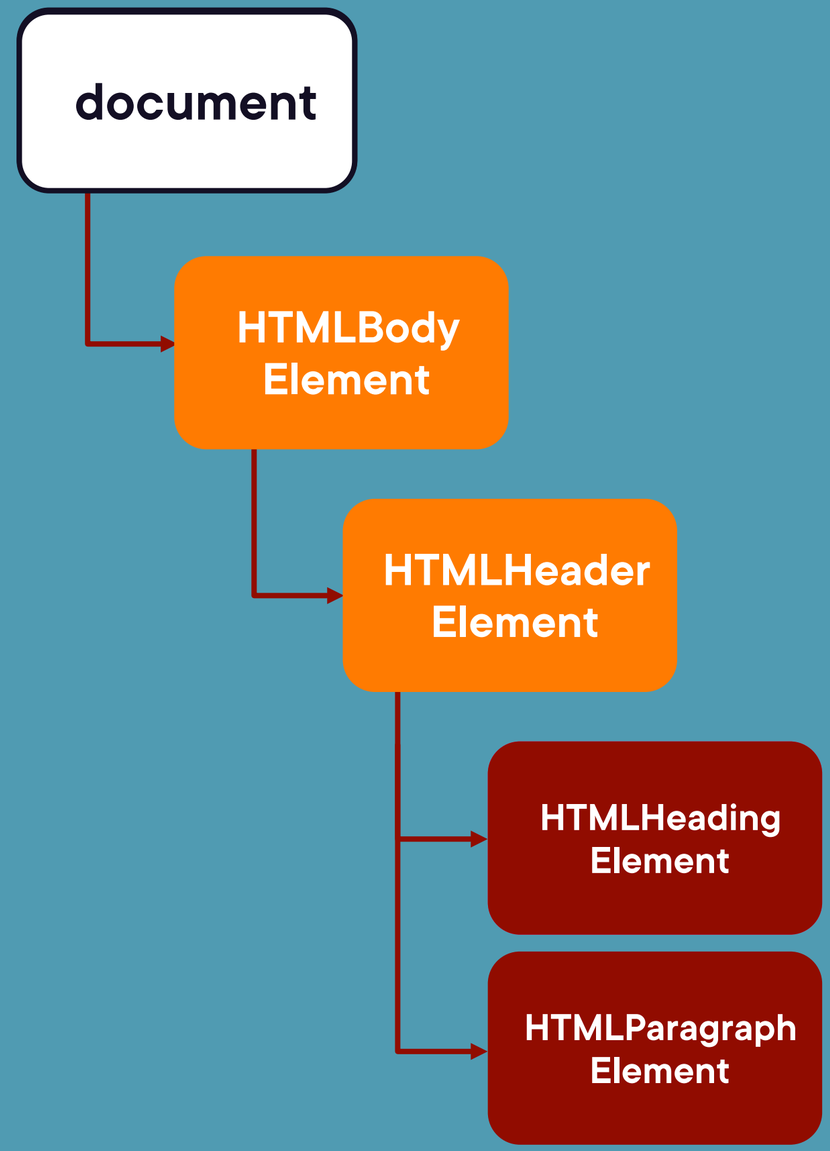
```
<html>

  <body>

    <header>

      <h1>The DOM</h1>

      <p class="tagline">
       Document Object Model
      </p>

      <img hidden src="dom.png">

    <header>

  </body>

</html>
```



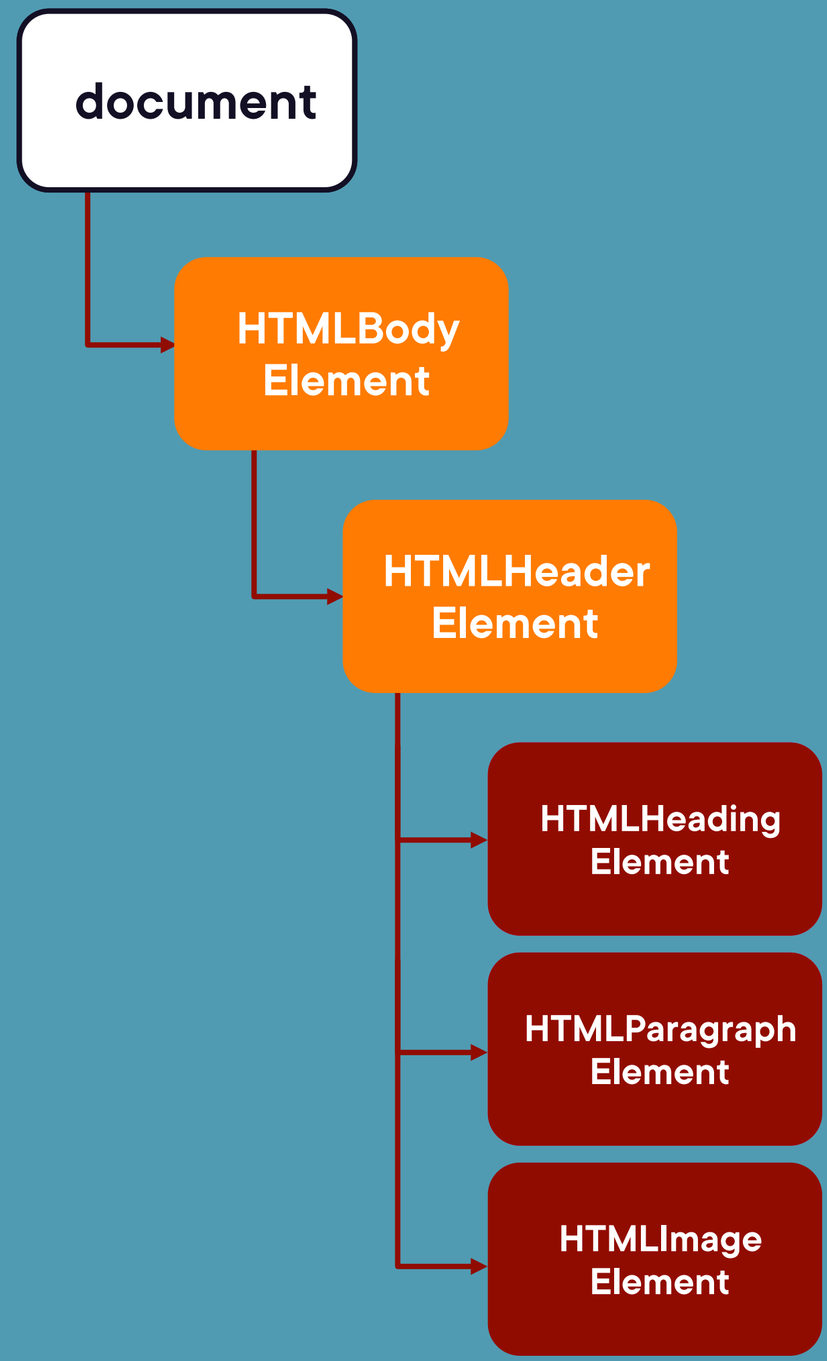
```
<html>

  <body>

    <header>

      <h1>The DOM</h1>

      <p class="tagline">
       Document Object Model
      </p>

      <img hidden src="dom.png">

    <header>

  </body>

</html>
```



```
<html>
  <body>
    <header>
      <h1>The DOM</h1>
      <p class="tagline">
       Document Object Model
      </p>
      <img hidden src="dom.png">
    <header>
  </body>
</html>
```



```
<html>

  <body>

    <header>

      <h1>The DOM</h1>

      <p class="tagline">
       Document Object Model
      </p>

      <img hidden src="dom.png">

    <header>

  </body>

</html>
```



To work with DOM elements, we can...

Pick them from the current DOM

Create them, and then inject then into the DOM

When we have a reference to an Element we can

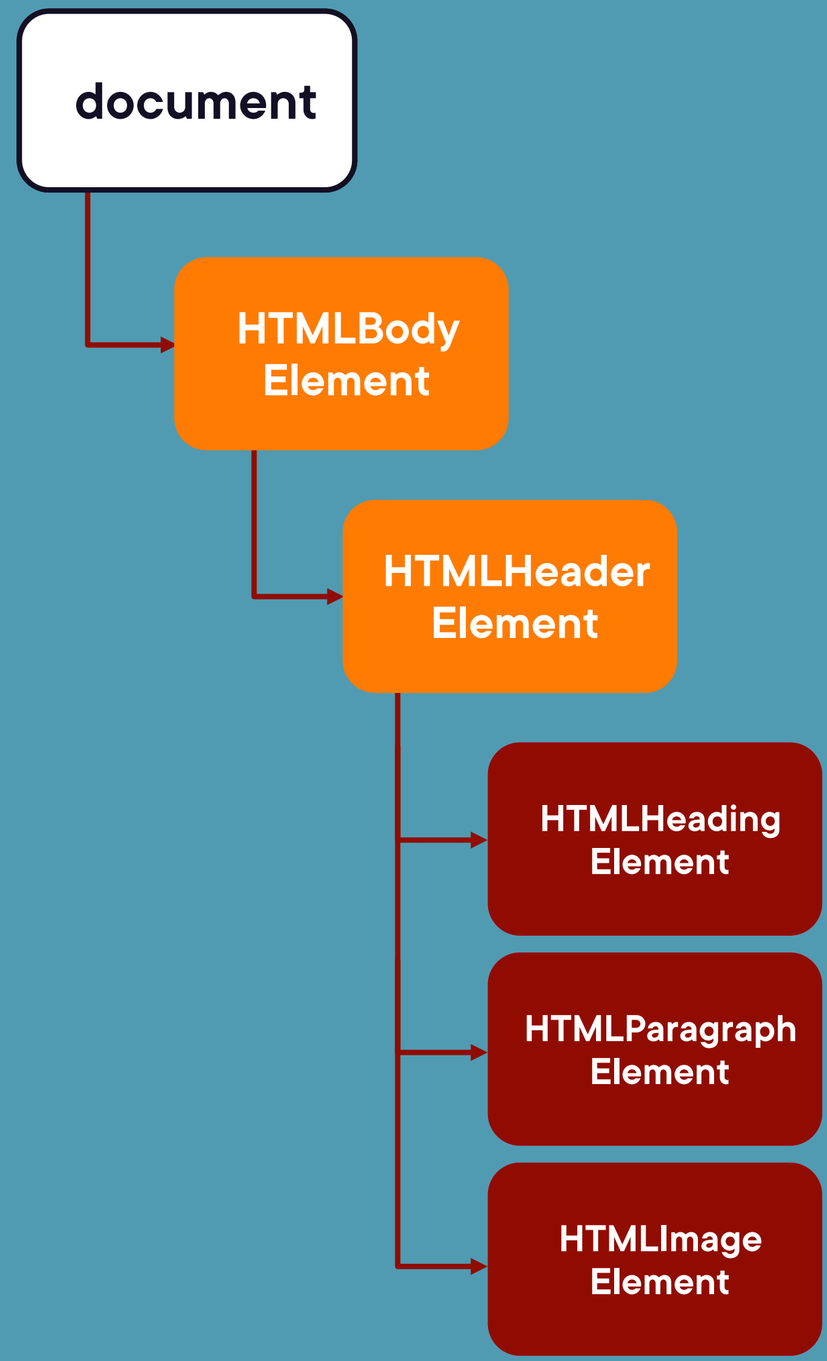
Read its content

Change its content

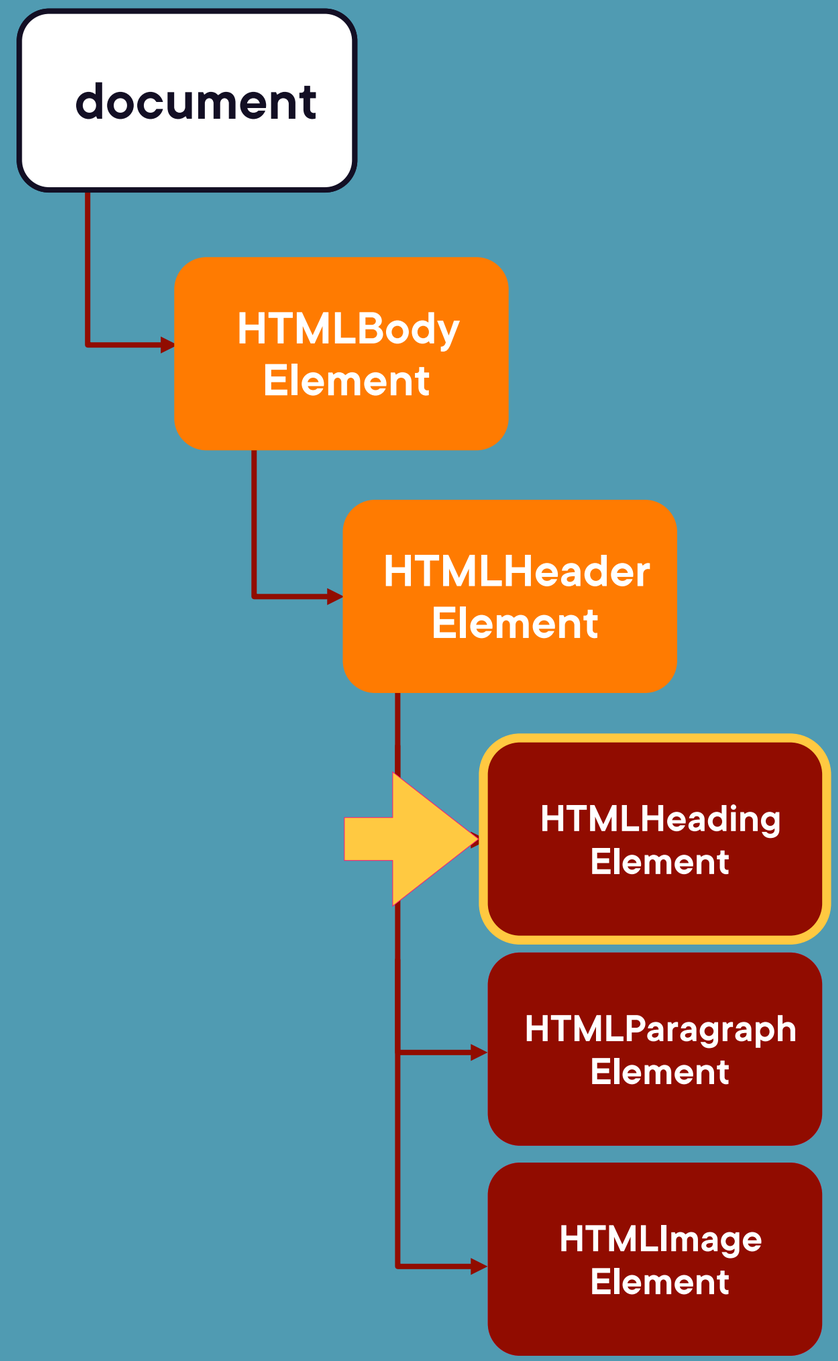
Remove it

Add new elements to it

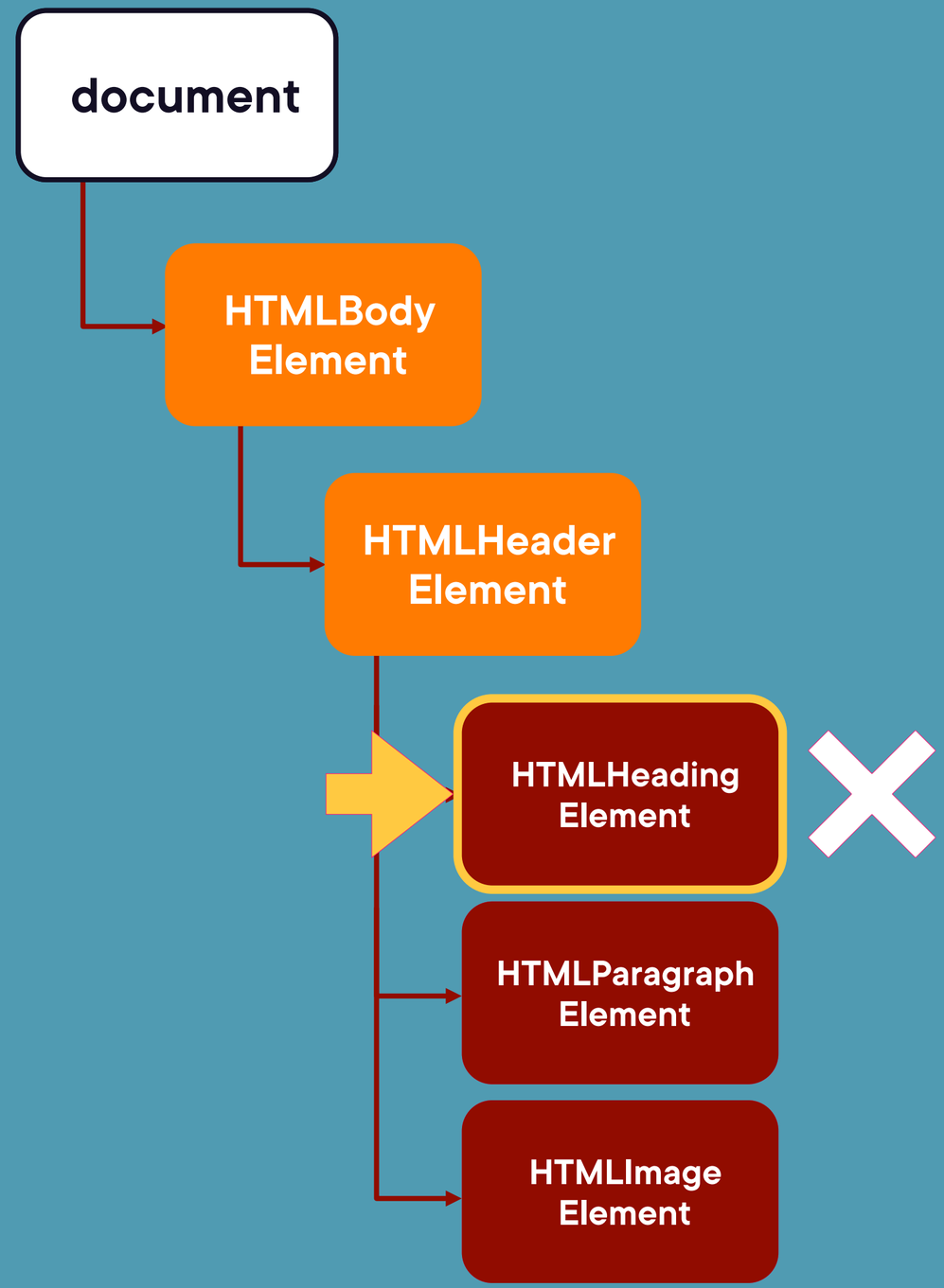
```
<html>

  <body>

    <header>

      <h1>The DOM</h1>

      <p class="tagline">
       Document Object Model
      </p>

      <img hidden src="dom.png">

    <header>

  </body>

</html>
```



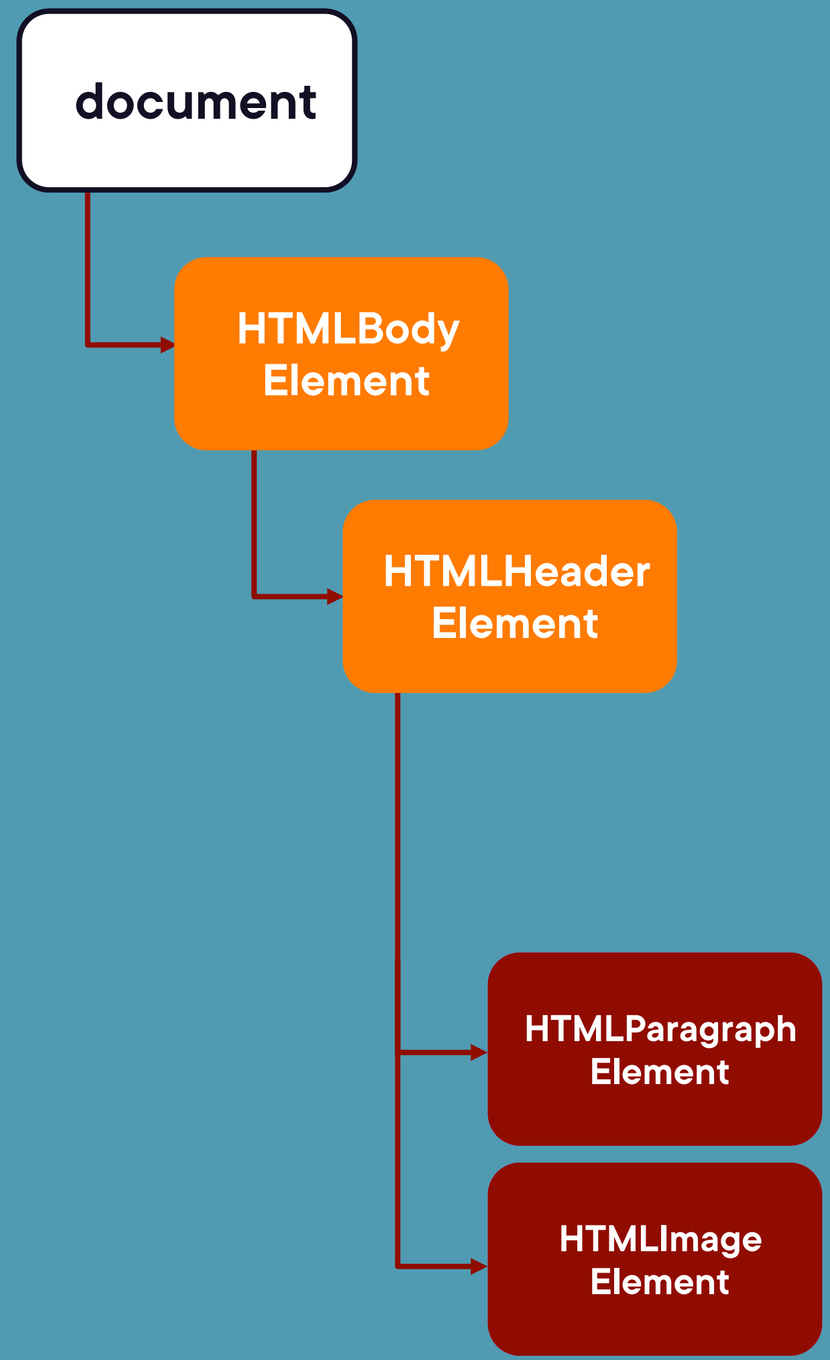
```
<html>

  <body>

    <header>

      <h1>The DOM</h1>

      <p class="tagline">
       Document Object Model
      </p>

      <img hidden src="dom.png">

    <header>

  </body>

</html>
```



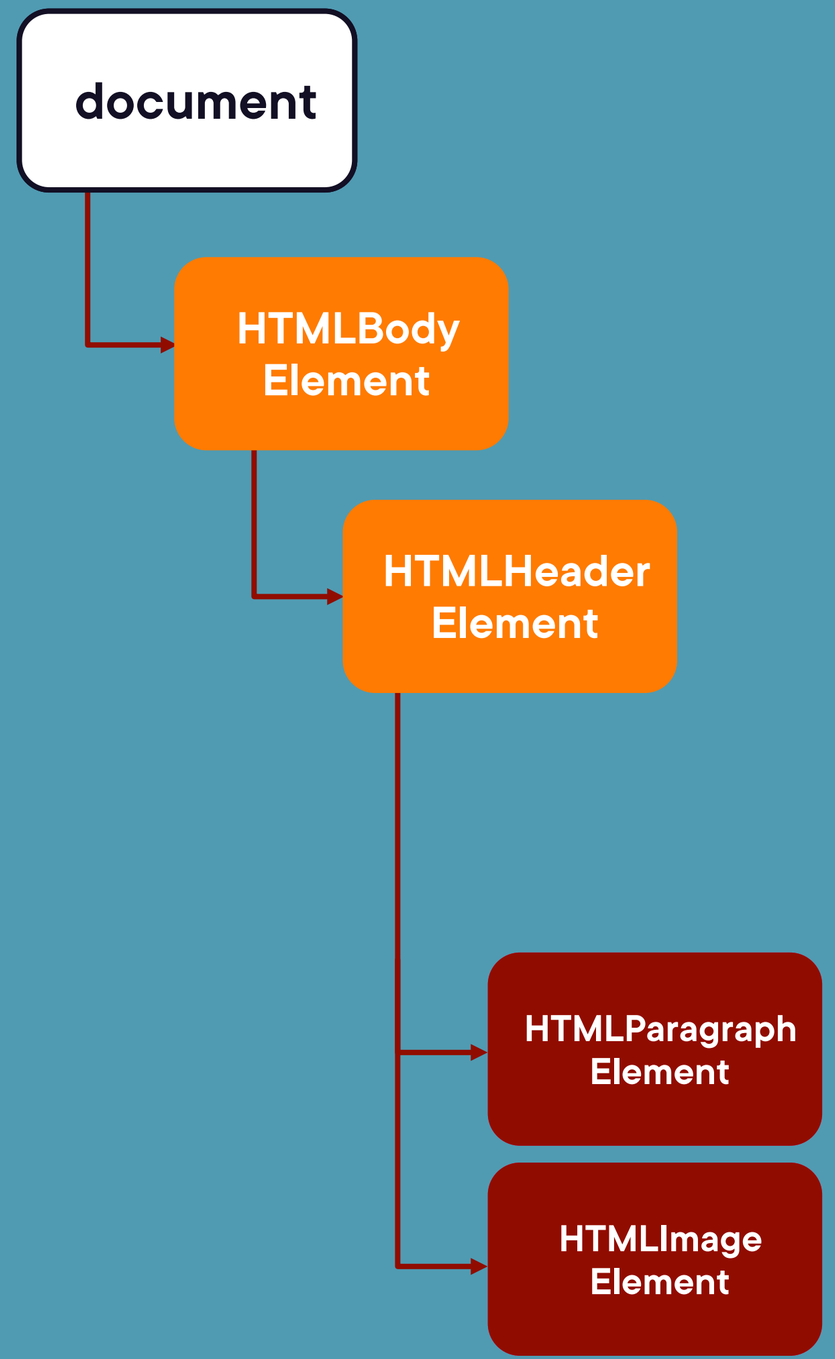
```
<html>

  <body>

    <header>

      <h1>The DOM</h1>

      <p class="tagline">
       Document Object Model
      </p>

      <img hidden src="dom.png">

    <header>

  </body>

</html>
```



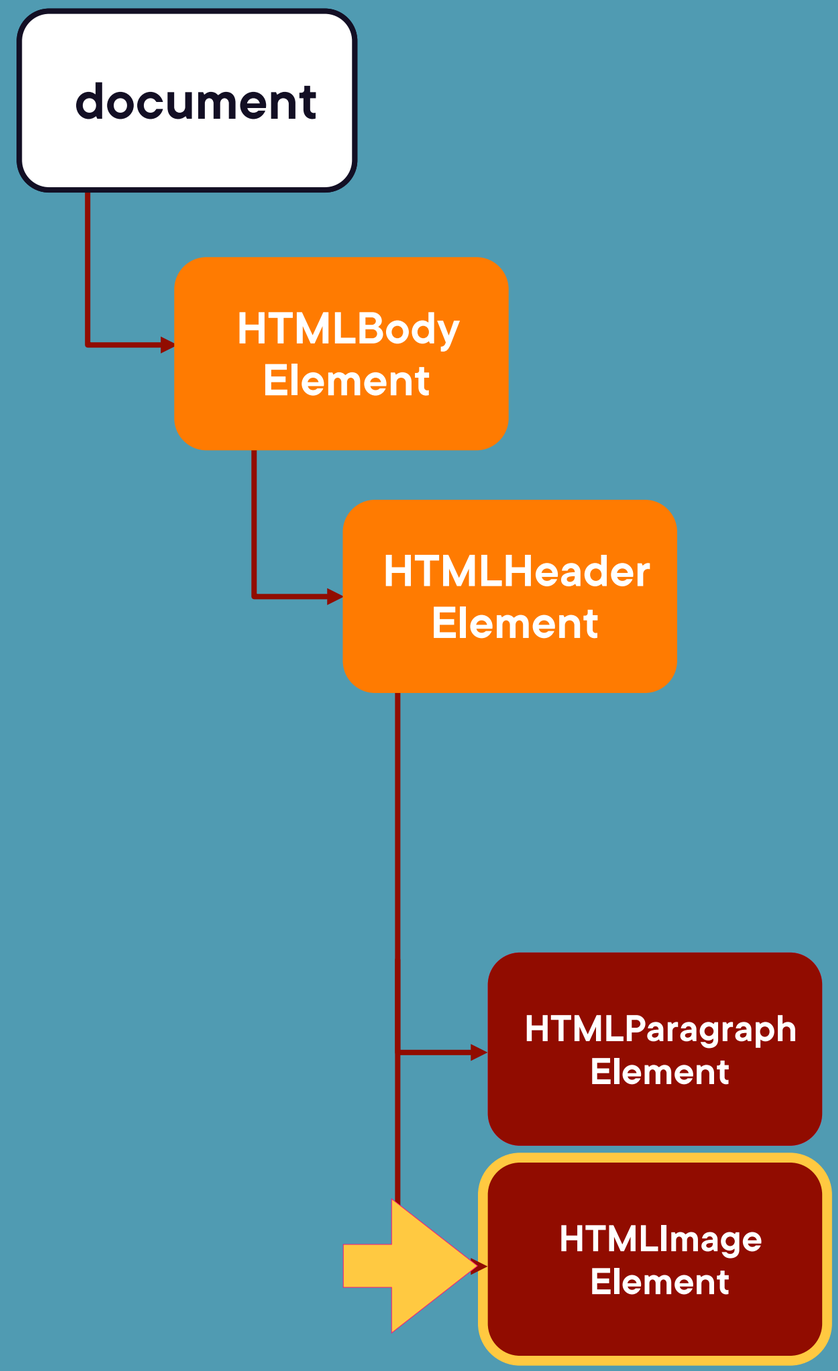
```
<html>

  <body>

    <header>

      <h1>The DOM</h1>

      <p class="tagline">
       Document Object Model
      </p>

      <img hidden src="dom.png">

    <header>

  </body>

</html>
```



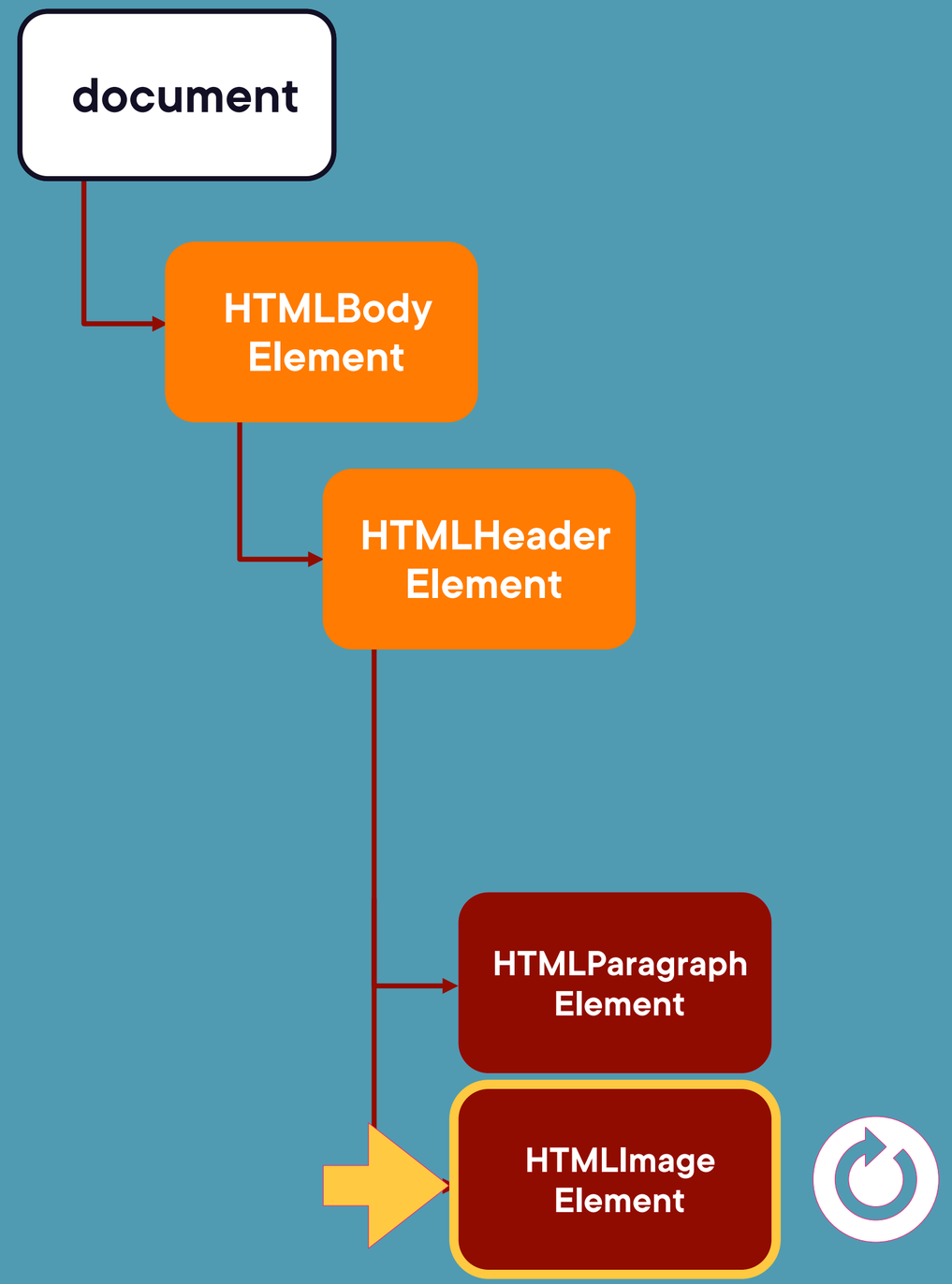
```
<html>
  <body>
    <header>
      <h1>The DOM</h1>
      <p class="tagline">
       Document Object Model
      </p>
      <img hidden src="dom.png">
    <header>
  </body>
</html>
```



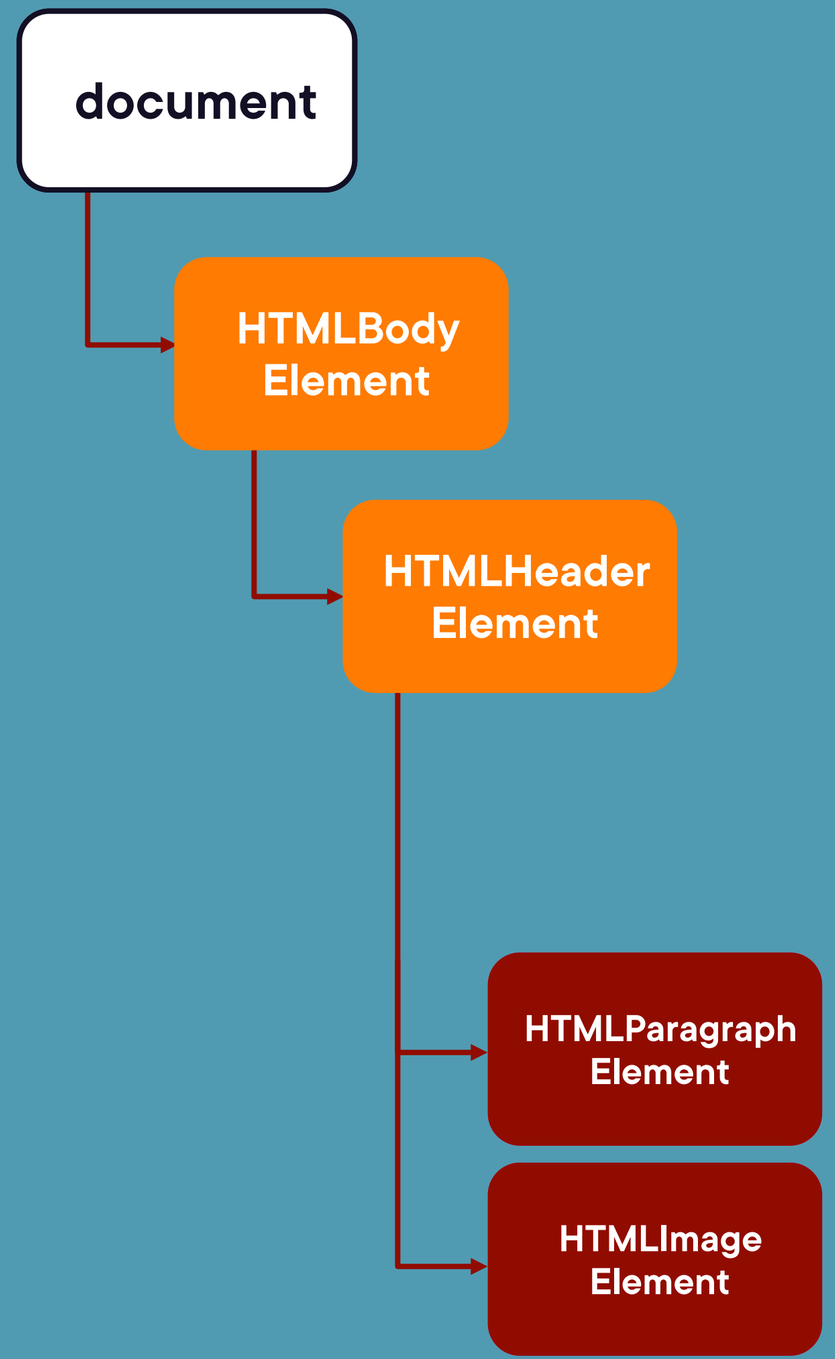
```
<html>
  <body>
    <header>
      <h1>The DOM</h1>
      <p class="tagline">
       Document Object Model
      </p>
      <img hidden src="dom.png">
    <header>
  </body>
</html>
```



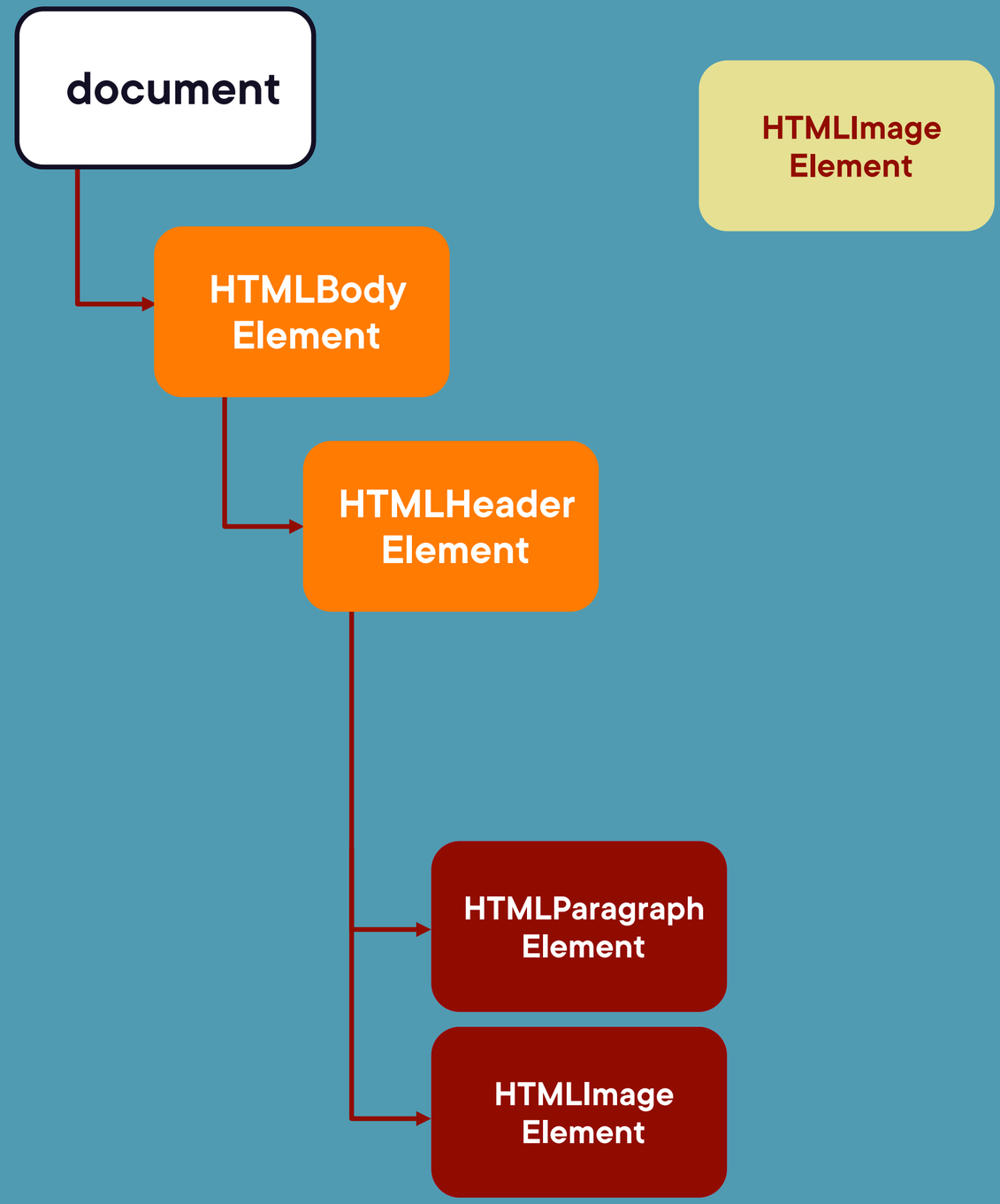
```
<html>
  <body>
    <header>
      <h1>The DOM</h1>
      <p class="tagline">
       Document Object Model
      </p>
      <img hidden src="dom.png">
    <header>
  </body>
</html>
```



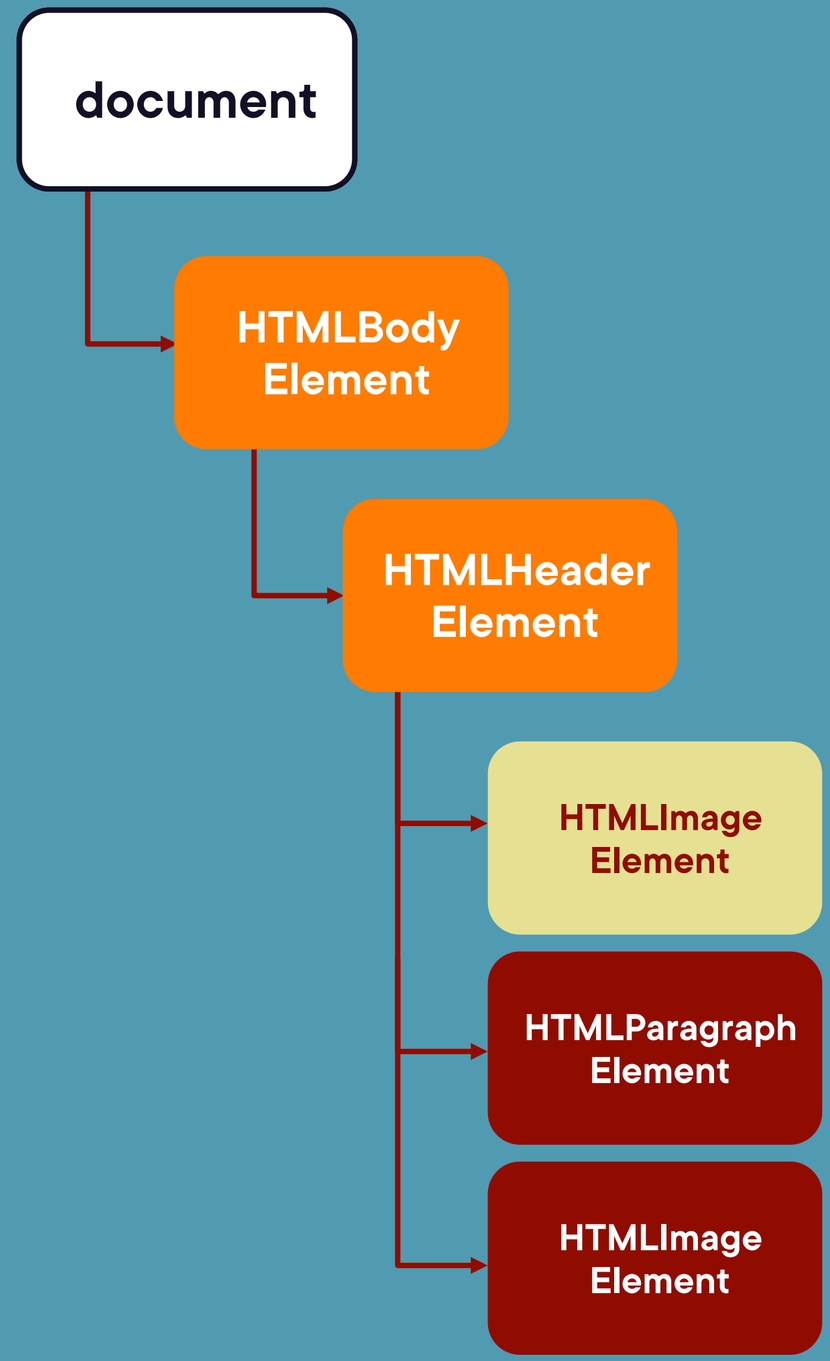
```
<html>
  <body>
    <header>
      <h1>The DOM</h1>
      <p class="tagline">
       Document Object Model
      </p>
      <img hidden src="dom.png">
    <header>
  </body>
</html>
```



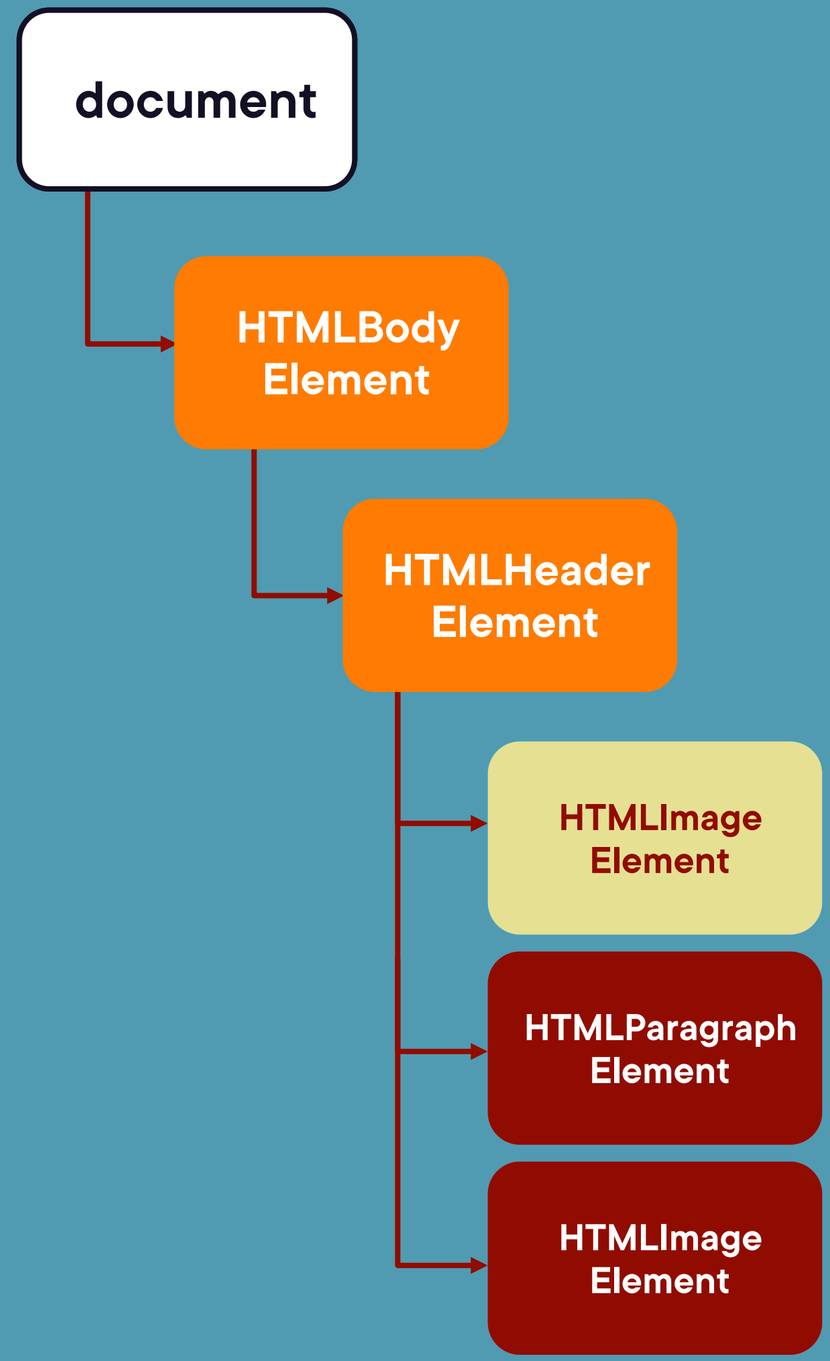
```
<html>

  <body>

    <header>


      <p class="tagline">
       Document Object Model
      </p>

      <img src="dom.png">

    <header>

  </body>

</html>
```



```
<html>

  <body>

    <header>


      <p class="tagline">
       Document Object Model
      </p>

      <img src="dom.png">

    <header>

  </body>

</html>
```



```
<html>

  <body>

    <header>

      <img src="header.png">

      <p class="tagline">
       Document Object Model
      </p>

      <img src="dom.png">

    <header>

  </body>

</html>
```



Working with the DOM API

Select elements from the DOM

By ID

By Class Name

By Name

By CSS Selector

Navigating DOM structure

When selecting elements, some functions return...

One HTML Element (`HTMLElement`)

A Live HTML Element Collection (`HTMLCollection`)

Static Element Collection (`NodeList`)

Functions to get a reference to one DOM element

getElementById

querySelector

WARNING

When you use a function that returns one element, it can also return null if no node was found

Finding one element

Always check for null in case it's possible the element does not exist

```
const element = document.getElementById("one-item");

if (element ≠ null) {
    // element found
}
```

Finding one element

Always check for null in case it's possible the element does not exist

```
const element = document.querySelector("section>header a");

if (element ≠ null) {
    // element found
}
```

Functions to get a reference to multiple DOM elements

getElementsByTagName

getElementsByClassName

querySelectorAll()
static collection

getElementsByName

WARNING

When you use a function that returns multiple elements, it can also return an empty collection if no elements were found

Finding multiple elements

The collection we receive has a length property ,can be accessed as an array using [ ] and it has all the array modern interface we know

```
// elements is an static NodeList
const elements = document.querySelectorAll("#nav-menu li");

if (elements.length > 0) {
    // elements found
    const firstElement = elements[0];
}
```

Finding multiple elements

The collection we receive has a length property and can be accessed as an array using [ ] but it's not an Array, so no access to all methods.

```
// elements is a live HTMLCollection
const elements = document.getElementsByClassName("important");

for (let currentElement of elements) {
   // we process each currentElement
}
```

WARNING

HTMLCollections (live) don't have all the modern Array interface, such as `filter`, `map`, `reduce` or `forEach`.

IDEAS

You can add modern array functions to HTMLCollection by creating a Array from it using

```
Array.from(collection)
```

Finding multiple elements

The collection we receive has a length property and can be accessed as an array using [ ] but it's not an Array, so no access to all methods.

```
// we enhance the HTMLCollection into an Array
const elements = Array.from(
                 document.getElementsByClassName("important"));

// we can use array methods now
elements.filter(e⇒e.tagName=="p");
```

With an HTML Element in JavaScript you can

- Read and change attributes' values
- Read and change styles
- Hook event listeners
- Add, remove or move children elements
- Read and change its contents
- More APIs

How to read or change attributes of a DOM element

You can use dot syntax to access properties mapped from HTML attributes.

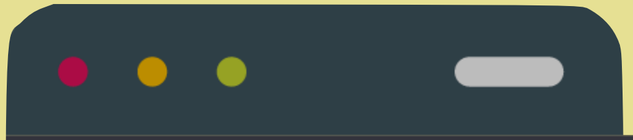
```
// Most attributes have the same name as JavaScript properties
// but exceptions apply, such as className and htmlFor

element.hidden = false;
element.src = "logo.png";
element.className = "myClass";              // "class" HTML attribute
```

How to read or change styles of a DOM element

You can use dot syntax to access a **style** object.
That object will have a map to every CSS property you can inline in HTML.

```
/*  Change kebab syntax (-) to camel case syntax.  */

element.style.color = "blue";
element.style.fontSize = "1.2em";                  // font-size
element.style.borderRightColor = "#FCFCFC";    // border-right-color
```

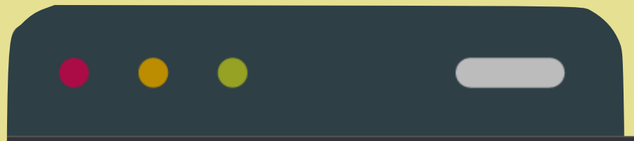
How to listen to events of an HTML element



You can use dot syntax and bind a function to an event name using addEventListener. There is a list of standard events.

```
element.addEventListener("event name", function);
```

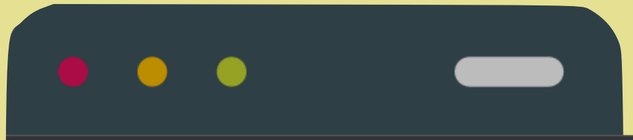
How to listen to events of an HTML element



You can use dot syntax and bind a function to an event name using addEventListener. There is a list of standard events.

```
function eventHandler(event) {
    // do something
}

element.addEventListener("click", eventHandler);
```

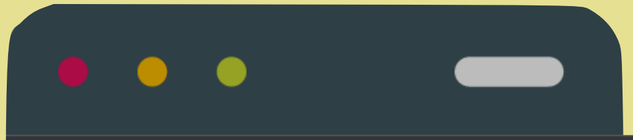
How to listen to events of an HTML element



You can use dot syntax and bind a function to an event name using addEventListener. There is a list of standard events.

```
element.addEventListener("click", function(event) {
    // do something
});
```

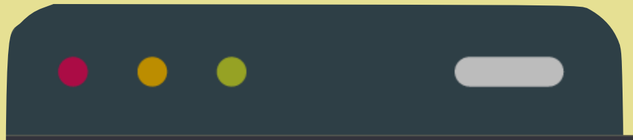
How to listen to events of an HTML element



You can use dot syntax and bind a function to an event name using addEventListener. There is a list of standard events.

```
element.addEventListener("click", (event) ⇒ {
    // do something
});
```

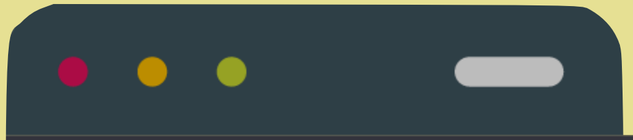
How to listen to events of an HTML element



You can use dot syntax and bind a function to an event name using addEventListener. There is a list of standard events.

```
element.addEventListener("click", (event) ⇒ {
    // do something
});
```

How to listen to events of an HTML element



You can use dot syntax and bind a function to an event name using addEventListener. There is a list of standard events.

```
element.addEventListener("click", () ⇒ {
    // do something
});
```

Accessing and Editing Contents of the Elements

Accessing textContent

Accessing innerHTML

Using DOM APIs to create new nodes

Working with textContent

```
const element = document.querySelector("#message");

// We read the current element's content as string
const contents = element.textContent;

// We change the contents of the element with a new string
element.textContent = "The text has been changed";
```

Working with innerHTML

```
const element = document.querySelector("#section-6 header");

// We read the current element's HTML content as string
const contents = element.innerHTML;

// We change the contents of the element with a new HTML string
element.innerHTML = `
   <h1>My App</a>
   <p>The best platform for learning frontend</p>
`;
```

Working with DOM APIs to create or edit content

```
const element = document.querySelector("#section-6 header");

const h1 = document.createElement("h1");
h1.textContent = "My App";
element.appendChild(h1);

const p = document.createElement("p");
p.textContent = "The best platform for learning frontend";
element.appendChild(p);
```

Our Project

- App: **Coffee Masters**
- Web app for a coffee shop
- Menu of products
- Details of each product
- Order with Cart
- HTML and CSS ready
- Data in JSON
- No JavaScript written

Let's code!

More on Event Binding

Event Binding

Each DOM element has a list of possible events we can listen to

- Basic: `load, click, dblclick`
- Value: `change`
- Keyboard Events: `keyup, keydown, keypress`
- Mouse Events: `mouseover, mouseout,` etc.
- Pointer and Touch Events
- Scroll, Focus and more APIs

Some specific objects have special events:

- `DOMContentLoaded, popstate` in `window`

IMPORTANT

The spec's naming pattern is to use lowercase with no word separator for event names. That's why we can end up with names like

`webkitcurrentplaybacktargetiswirelesschanged`

Binding functions to events in DOM objects

onevent properties

addEventListener

Using onevent properties for event binding

When you use this technique, only one function can be attached per event/object combination.

Using onevent properties for event binding

When you use this technique, only one function can be attached per event/object combination.

```
function eventHandler(event) {

}
element.oneventname = eventHandler;
```

Using onevent properties for event binding

When you use this technique, only one function can be attached per event/object combination.

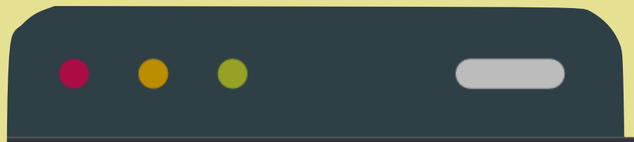
```
function eventHandler(event) {


}

element.onclick = eventHandler;
element.onload = eventHandler;
```

Using onevent properties for event binding

When you use this technique, only one function can be attached per event/object combination.

```
function eventHandler(event) {


}



element.onload = eventHandler;
element.onload = (event) => {
    // it replaces the first handler
};
```

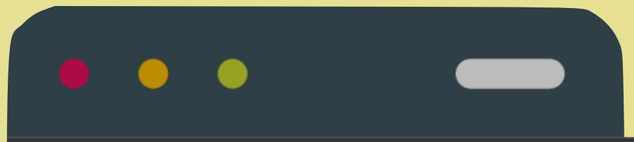
Using addEventListener for event binding



When you use this technique, you can attach more than one event handler per event/object combination.

```
function eventHandler(event) {


}
element.addEventListener("eventname", eventHandler);
```

Using addEventListener for event binding



When you use this technique, you can attach more than one event handler per event/object combination.

```
function eventHandler(event) {


}

element.addEventListener("click", eventHandler);
element.addEventListener("load", eventHandler);
```

Using addEventListener for event binding

When you use this technique, you can attach more than one event handler per event/object combination.

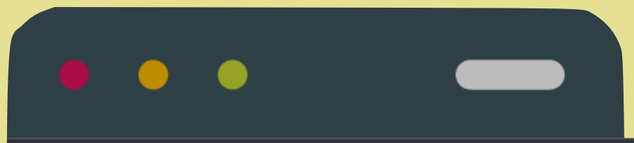
```
function eventHandler(event) {


}



element.addEventListener("load", eventHandler);
element.addEventListener("load", (event) => {
});
```

Advanced Event Handling

When you use this technique, you can attach more than one event handler per event/object combination.

```
function eventHandler(event) {


}
const options = {
    once: true,
    passive: true
}
element.addEventListener("load", eventHandler, options);
```

Advanced Event Handling

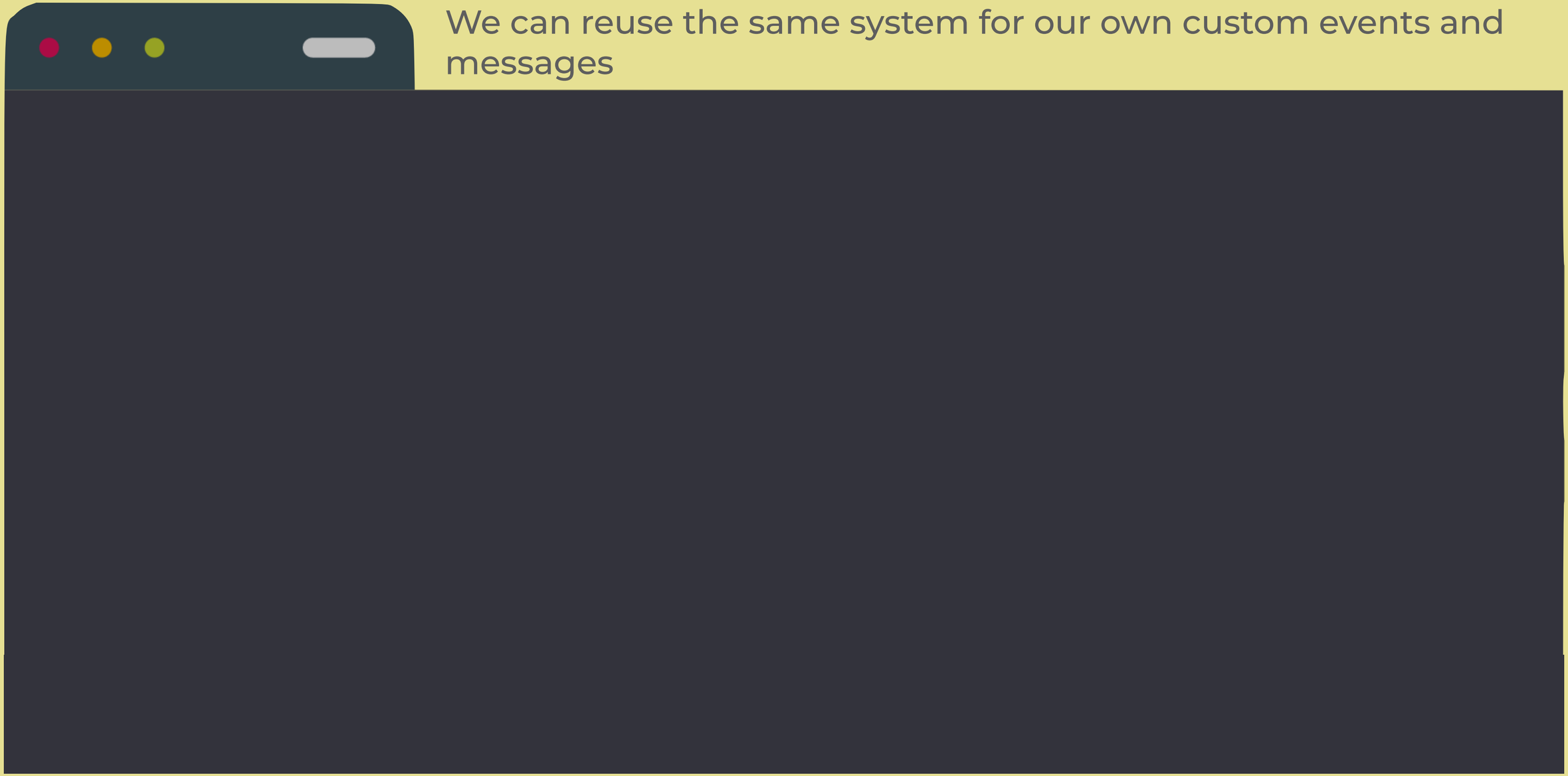


When you use this technique, you can attach more than one event handler per event/object combination.

```
function eventHandler(event) {


}
const options = {
   once: true,
   passive: true
}
element.addEventListener("load", eventHandler, options);
element.removeEventListener("load", eventHandler);
```

Dispatching Custom Events

We can reuse the same system for our own custom events and messages



Dispatching Custom Events

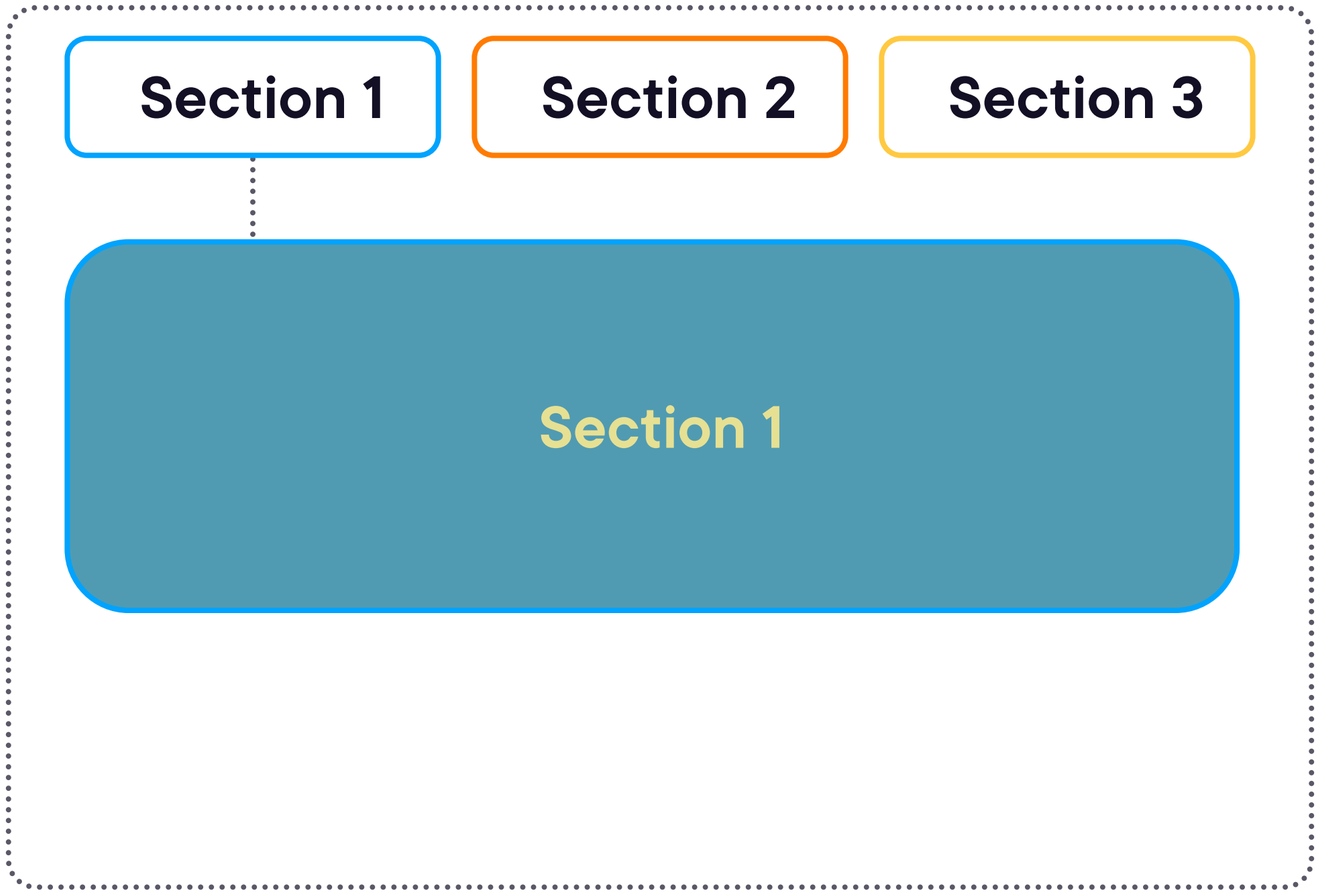
We can reuse the same system for our own custom events and messages

```
const event = new Event("mycustomname");
element.dispatchEvent(event);
```

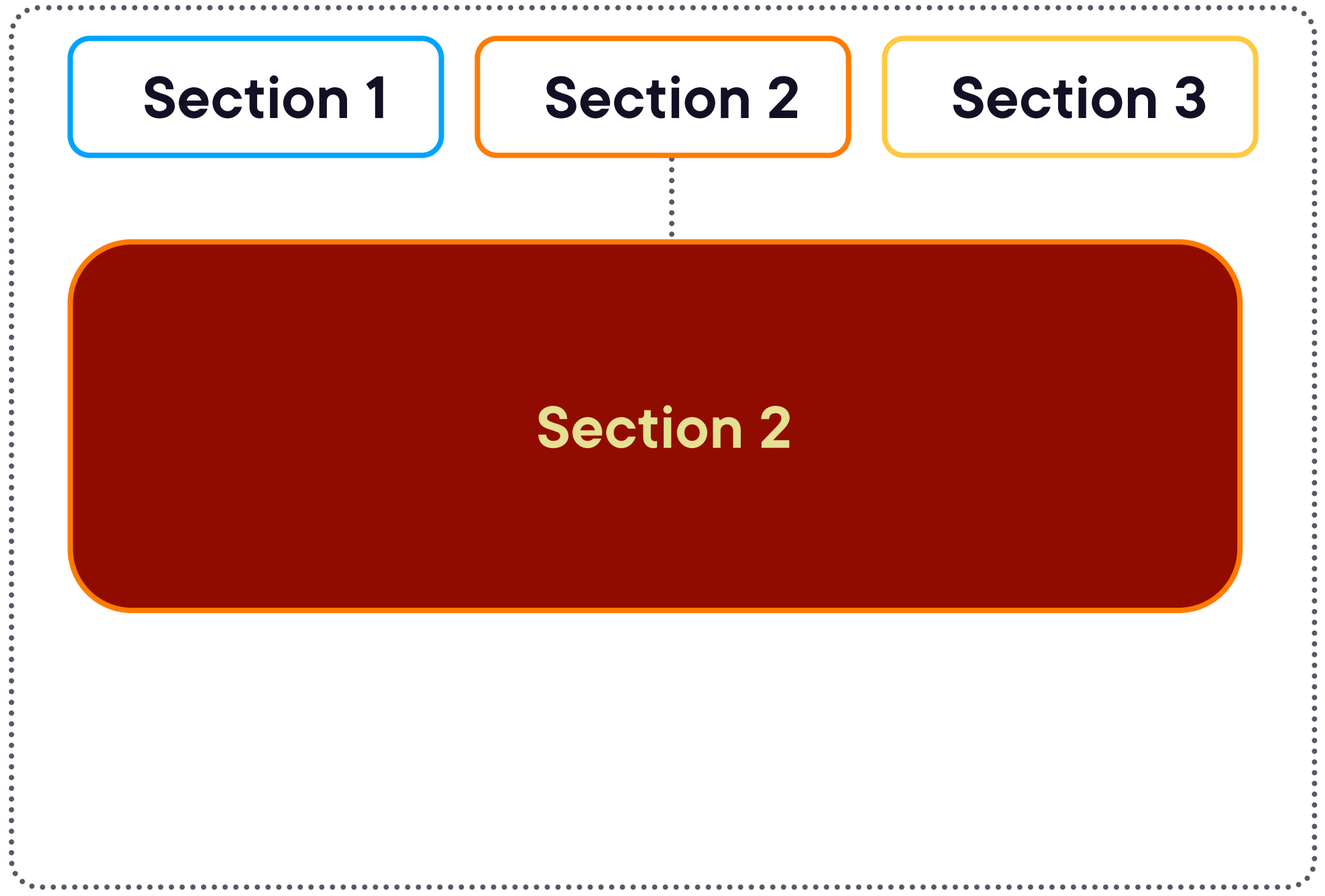
Let's code!

Navigating Between Pages Using the DOM

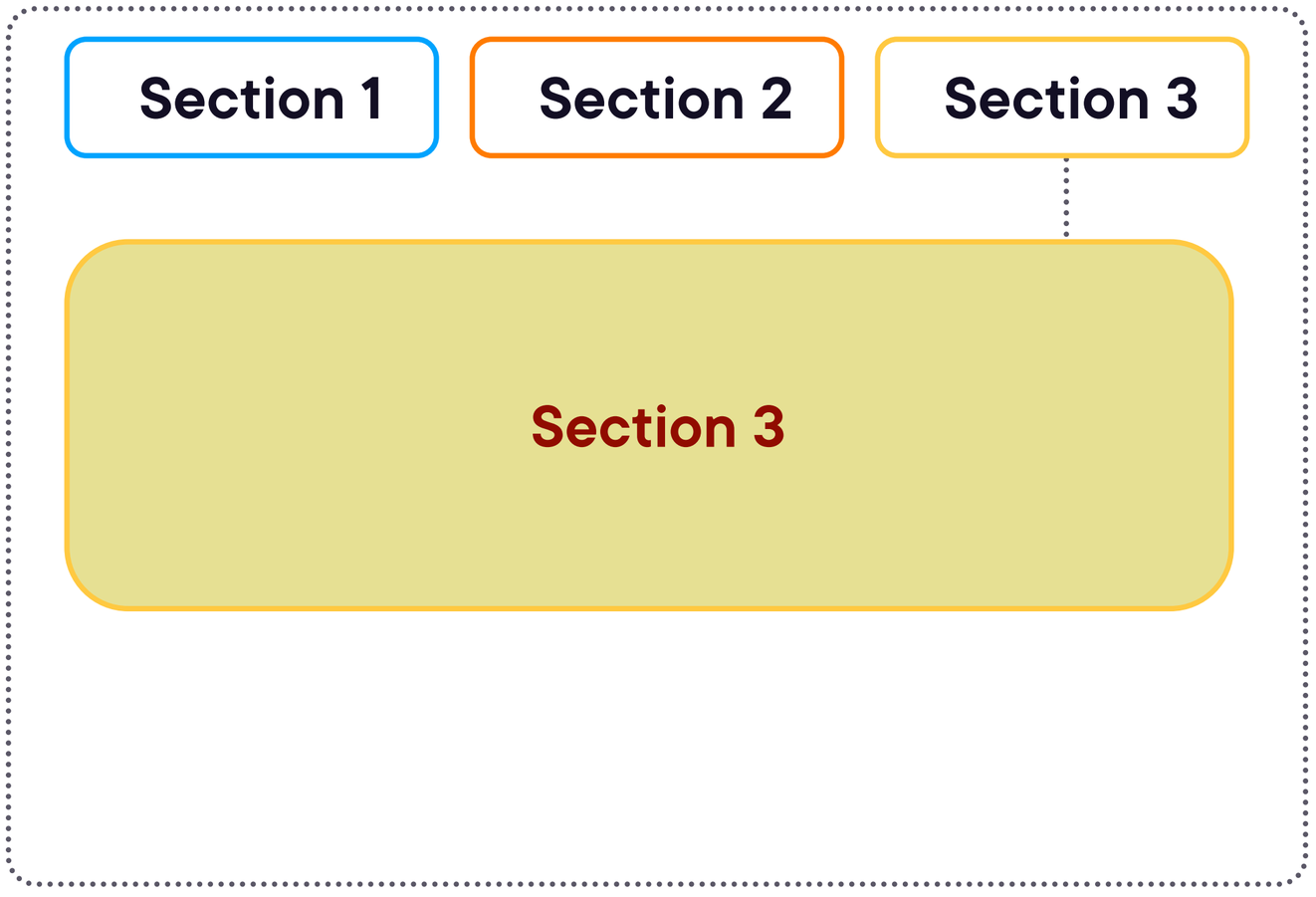
Single Page Application



Single Page Application



Single Page Application



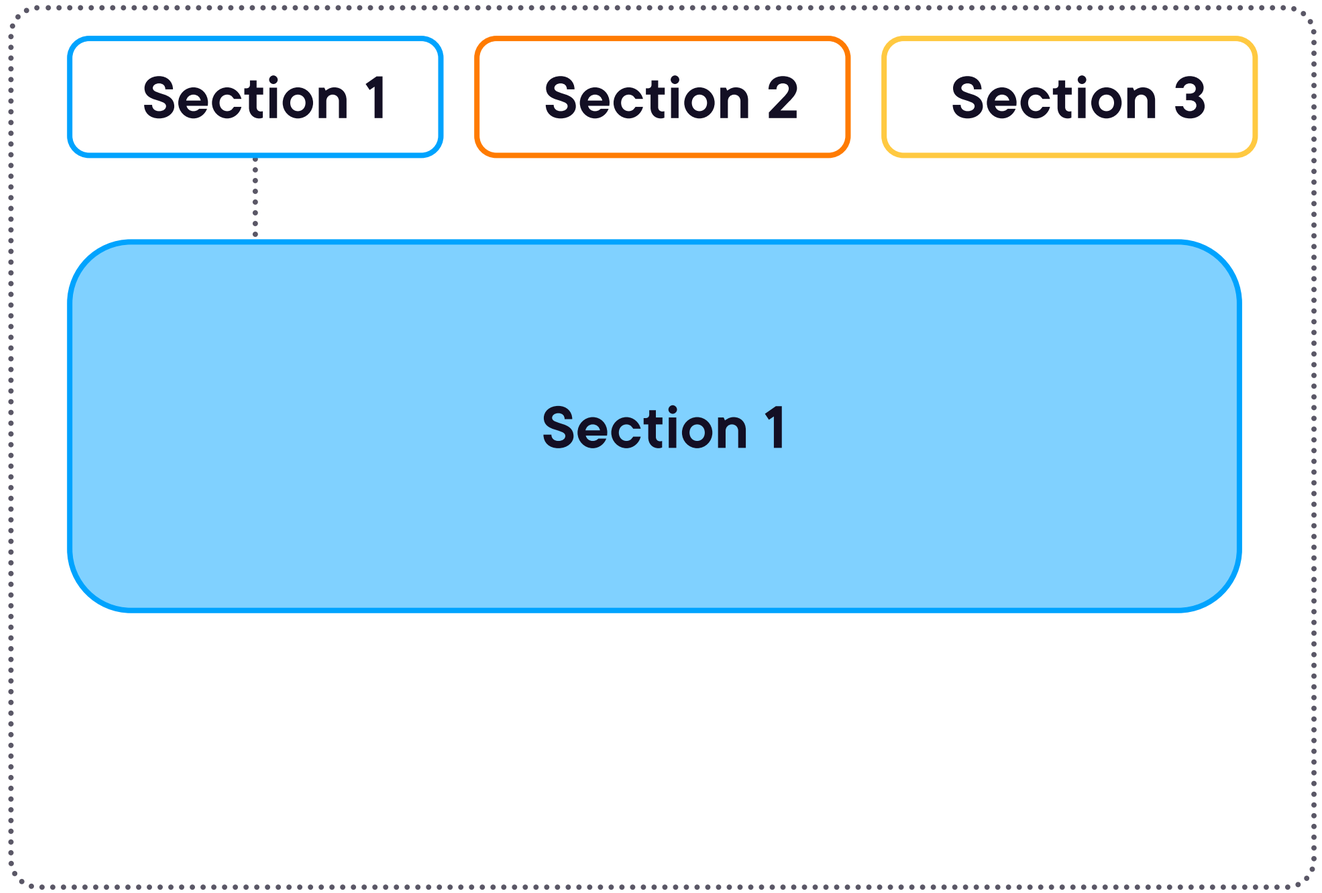
Single Page Applications: how to change content

Remove Previous Page and Inject New Page into the DOM

Hide Previous Page and Show Current Page

Single Page Application

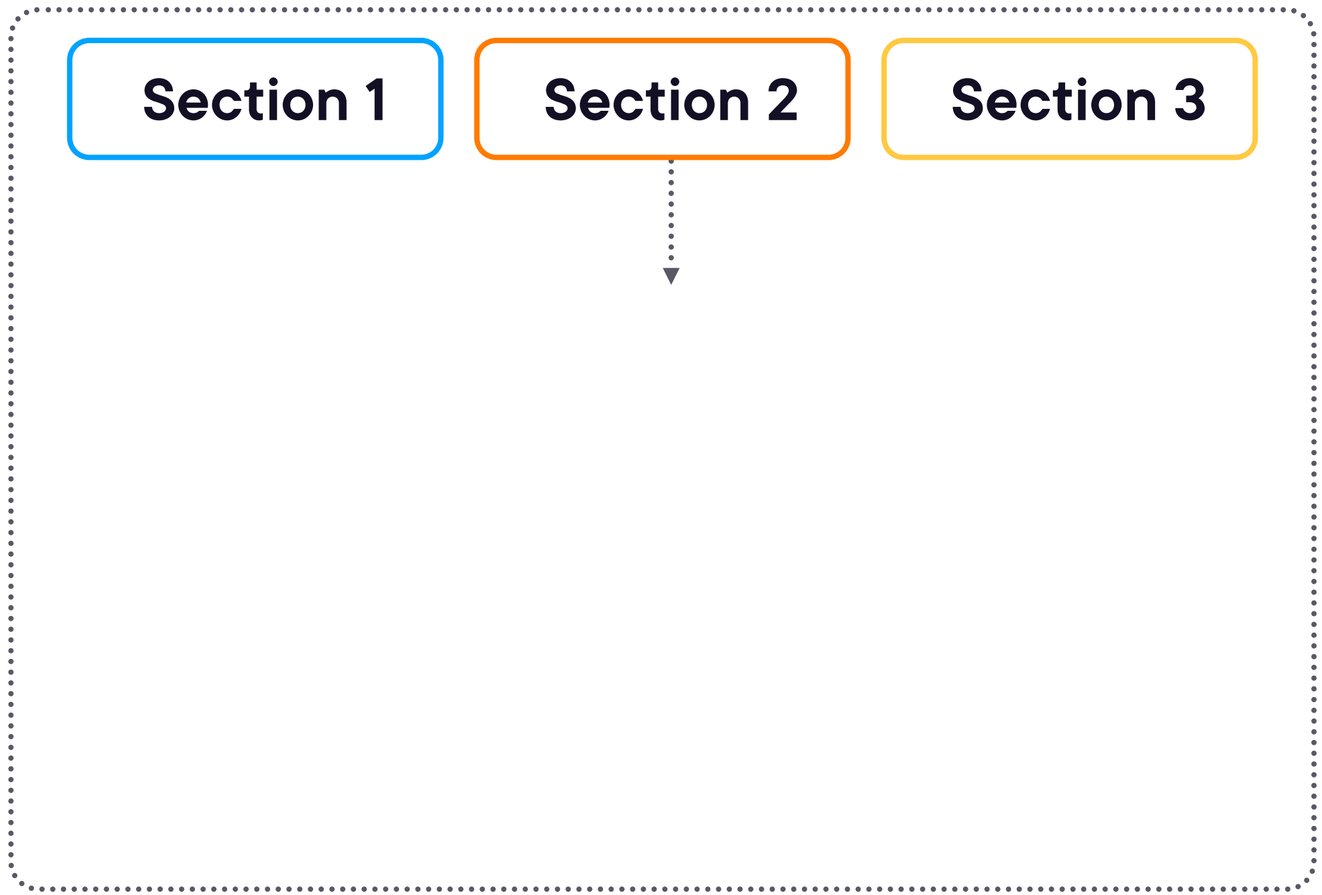
Remove Previous Page and Inject New Page into the DOM




```
<nav>
  <a href="">Section 1</a>
  <a href="">Section 2</a>
  <a href="">Section 3</a>
</nav>


<section id="section1">
</section>
```

Single Page Application

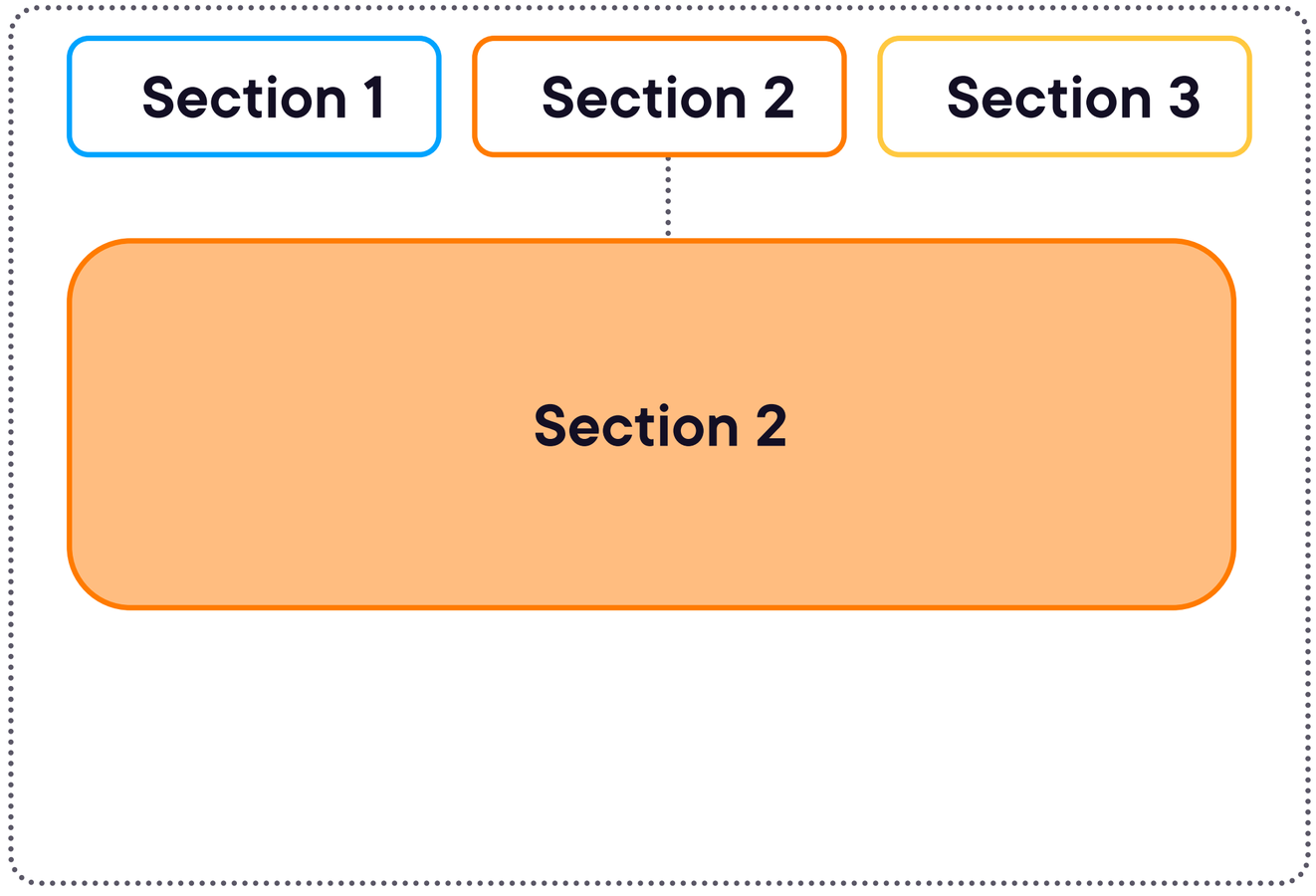
Remove Previous Page and Inject New Page into the DOM




```
<nav>
  <a href="">Section 1</a>
  <a href="">Section 2</a>
  <a href="">Section 3</a>
</nav>


<section id="section1">
</section>
```

Single Page Application

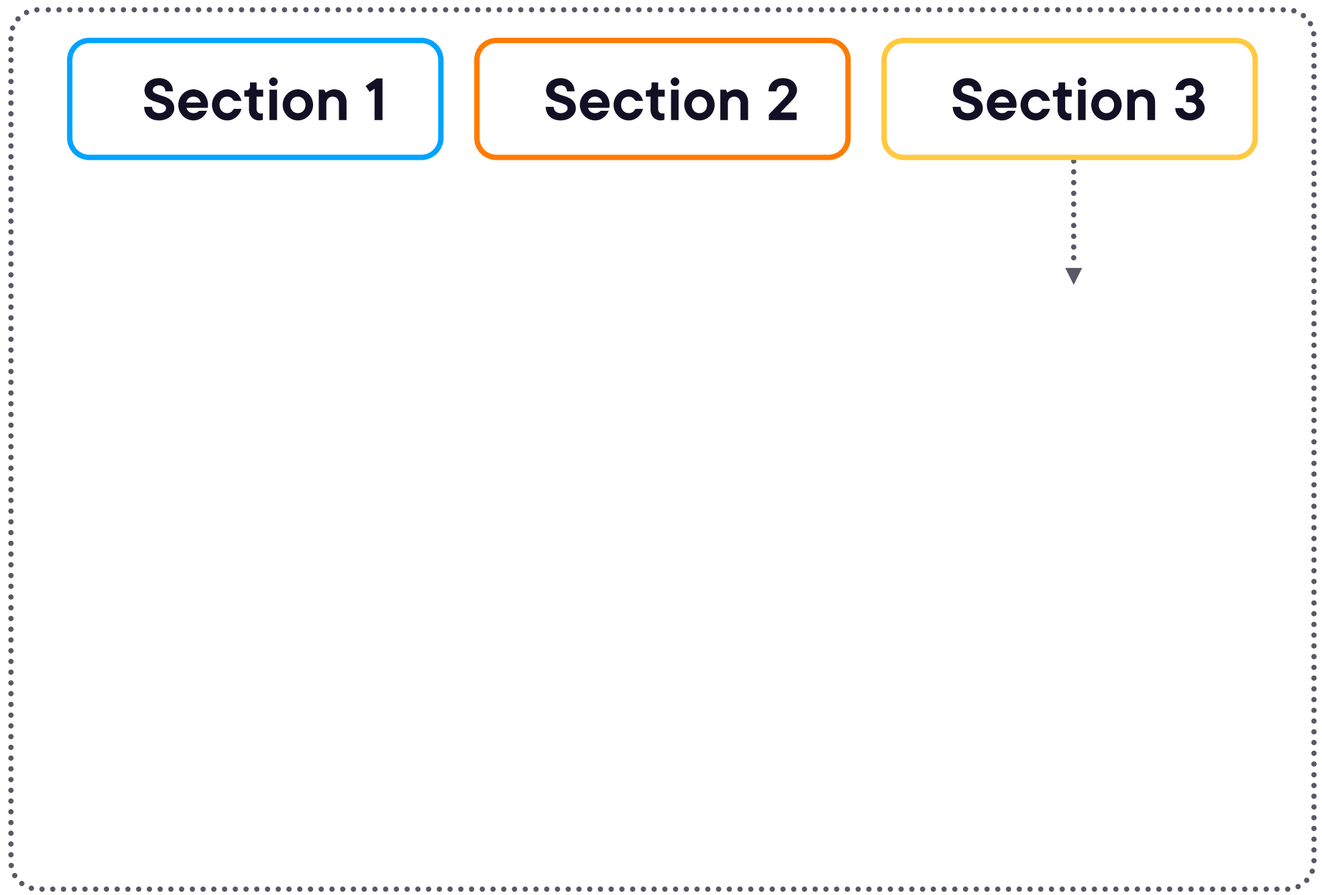
Remove Previous Page and Inject New Page into the DOM




```
<nav>
  <a href="">Section 1</a>
  <a href="">Section 2</a>
  <a href="">Section 3</a>
</nav>

<section id="section2">
</section>
```

Single Page Application

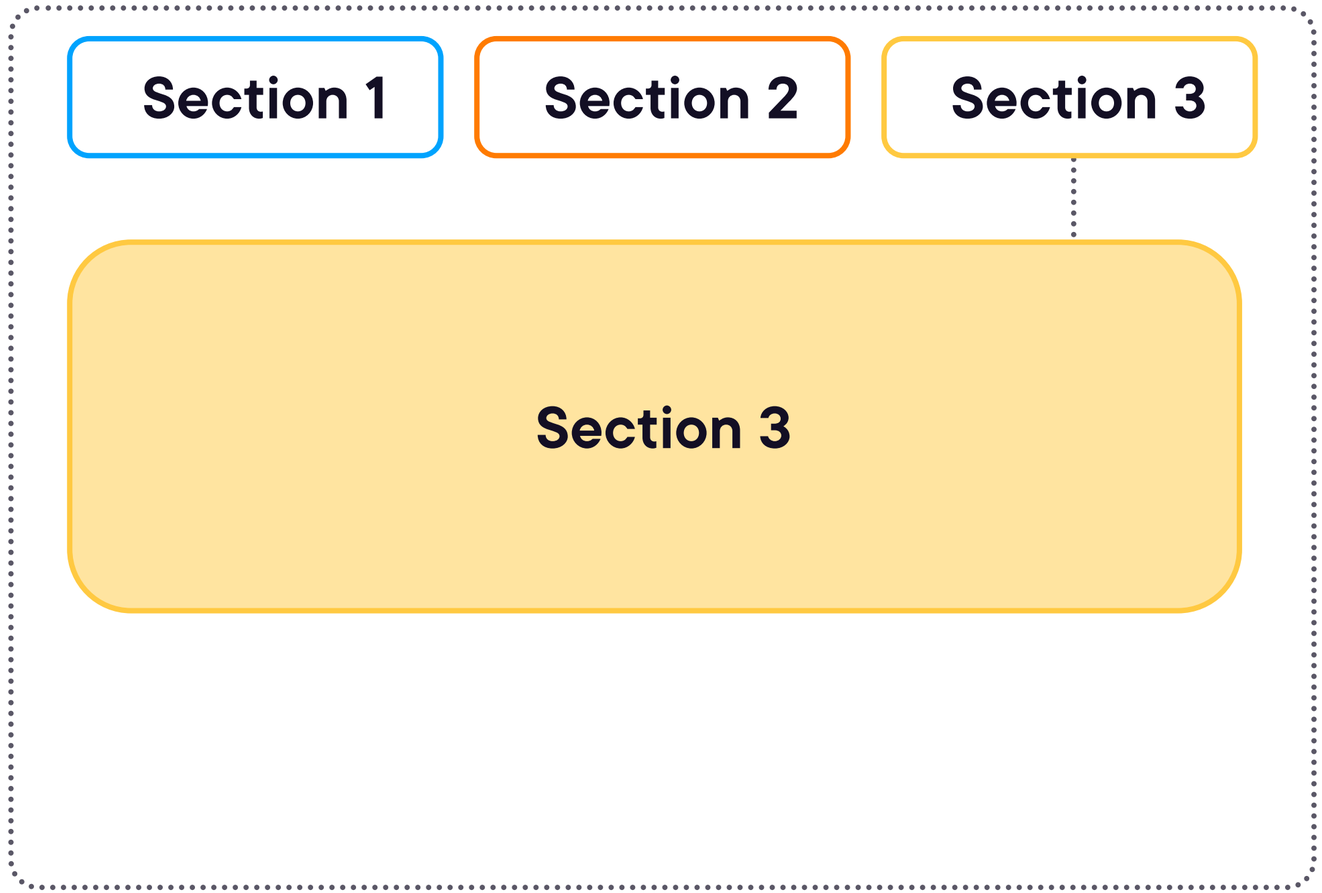
Remove Previous Page and Inject New Page into the DOM




```
<nav>
  <a href="">Section 1</a>
  <a href="">Section 2</a>
  <a href="">Section 3</a>
</nav>


<section id="section2">
</section>
```

Single Page Application

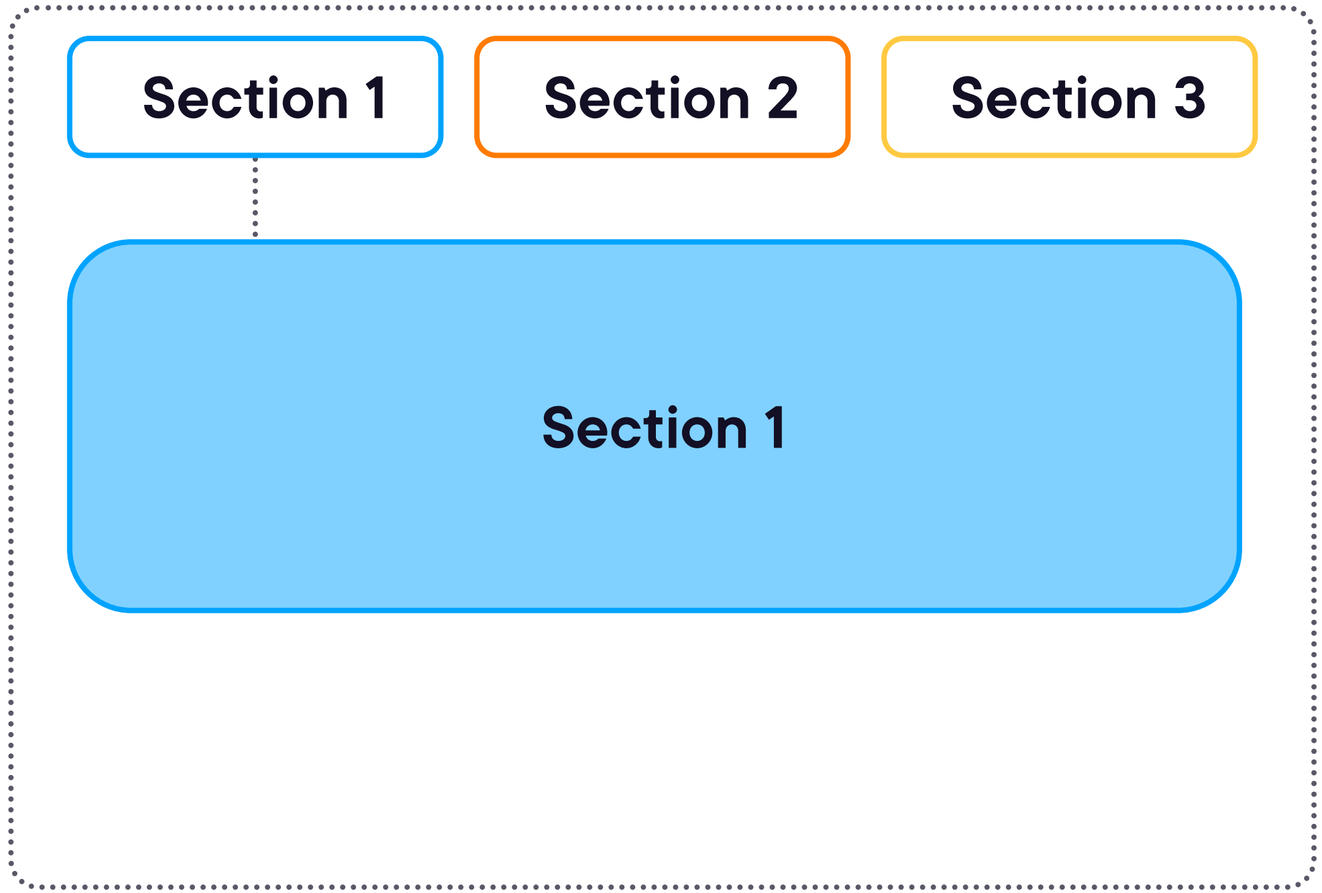
Remove Previous Page and Inject New Page into the DOM




```
<nav>
  <a href="">Section 1</a>
  <a href="">Section 2</a>
  <a href="">Section 3</a>
</nav>

<section id="section3">
</section>
```

Single Page Application

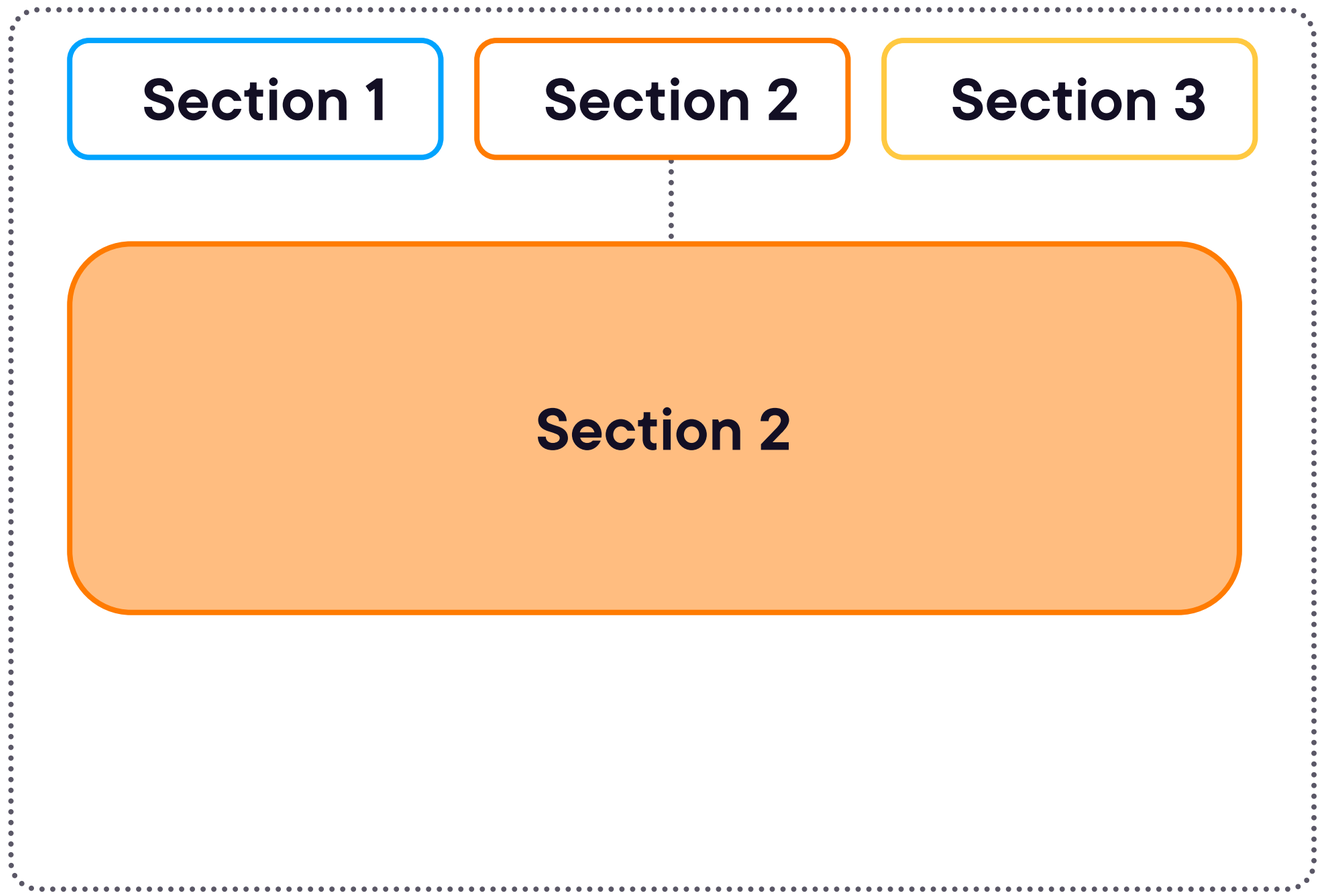
Hide Previous Page and Show Current Page




```
<nav>
  <a href="">Section 1</a>
  <a href="">Section 2</a>
  <a href="">Section 3</a>
</nav>


<section id="section1">
</section>

<section id="section2" hidden>
</section>

<section id="section3" hidden>
</section>
```

Single Page Application

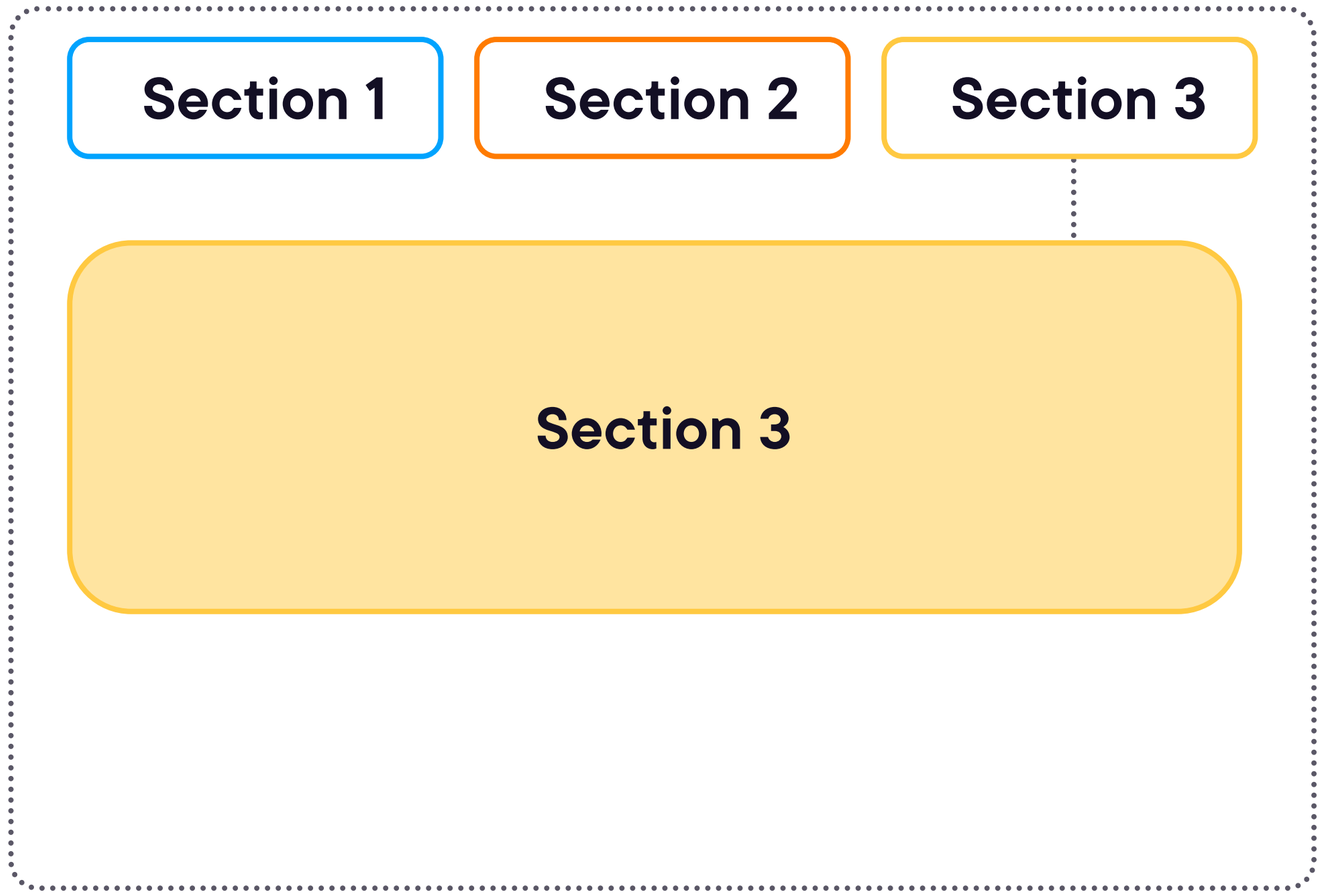
Hide Previous Page and Show Current Page




```
<nav>
  <a href="">Section 1</a>
  <a href="">Section 2</a>
  <a href="">Section 3</a>
</nav>

<section id="section1" hidden>
</section>

<section id="section2">
</section>

<section id="section3" hidden>
</section>
```

Single Page Application

Hide Previous Page and Show Current Page




```
<nav>
  <a href="">Section 1</a>
  <a href="">Section 2</a>
  <a href="">Section 3</a>
</nav>

<section id="section1" hidden>
</section>

<section id="section2" hidden>
</section>

<section id="section3">
</section>
```

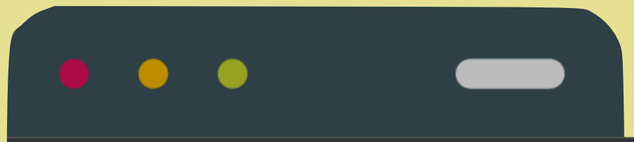
Single Page Application and Router

- We want to change the content of the page based on what the user select:
  - Home Page: menu
  - Details of one particular product
  - Order: cart details and order form

We won't have multiple HTML files, we will use the DOM APIs and Web Components

We will use the History API to push new entries to the navigation history

History API



We can push new "fake" navigation URLs and listen for changes

```
// pushing a new URL; the second argument is unused
history.pushState(optional_state, null, "/new-url");

// to listen for changes in URL within the same page navigation
window.addEventListener("popstate", event ⇒ {
   let url = location.href;
});
```

WARNING

`popstate` won't be fired if the user clicks on an external link or changes the URL manually

IMPORTANT

If you are creating a SPA, configure your server properly and check the URL when the whole page loads for the first time.

Let's code!

IDEAS

Make the Router reusable by receiving a collection of routes (path as a regex and component to render)

Web Components

DEFINITION

Web Component

A modular, reusable building block for web development that encapsulates a set of related functionality and user interface elements.

DEFINITION

Web Component

In short, your own custom HTML tag element.

Web Components

- Compatible with every browser
- It's actually a set of standards:
  - **Custom Elements**
  - **HTML Templates**
  - **Shadow DOM**
  - **Declarative Shadow DOM** 🆕
- It's similar to the idea of components on most of the libraries for JavaScript
- We have freedom of choice on how to define them and use them

DEFINITION

Custom Element

A way to define new, reusable HTML elements with custom behavior and functionality using JavaScript.

Custom Elements

We can define our own HTML tags using the Custom Elements API

```
class MyElement extends HTMLElement {
   constructor() {
      super();
   }
}

customElements.define("my-element",
                      MyElement);
```

```
<body>
  <my-element></my-element>
</body>

<script>
document.createElement("my-element");
</script>
```

WARNING

The HTML tag we define must contain a hyphen (-) to assure future compatibility

Custom Elements with Attributes

We can define our own custom attributes using the data-* spec.

```
class MyElement extends HTMLElement {
   constructor() {
      super();
      this.dataset.language
   }
}

customElements.define("my-element",
                       MyElement);
```

```
<body>
  <my-element data-language="en">
  </my-element>
</body>
```

Custom Elements Lifecycle

We can override some methods of the super class

```
class MyElement extends HTMLElement {

  constructor() {              // Set up initial state, event listeners, etc.
     super();
  }
  connectedCallback() { }     // The element is added to the document
  disconnectedCallback() { } // The element is removed to the document
  adoptedCallback() { }.      // The element has been moved to a new document

  attributeChangedCallback(name, oldValue, new Value() { }

}

customElements.define("my-element", MyElement);
```

Custom Elements with Slots

The slot is the content that we can define as the component's children.
With templates we can have more than one slot.

```
class MyElement extends HTMLElement {
   constructor() {
      super();
   }
}

customElements.define("my-element",
                       MyElement);
```

```
<body>
  <my-element>
     <div>
        <h2>Slot of My Element</h2>
     </div>
  </my-element>
</body>
```

Custom Elements with Customized Builtins

We can extend HTML elements with our own implementation

⛔ **Not available in Safari in 2023**

```
class MyCustomElement extends HTMLElement
{
   constructor() {
      super();
   }
}
customElements.define("my-element",
                       MyElement);
```

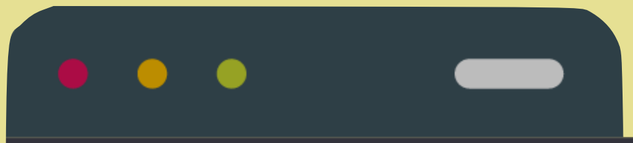
```
<div is="my-element">

</div>
```

DEFINITION

Template Element

Fragments of markup that can be cloned and inserted into the document at runtime, with reusable HTML content that can be rendered and modified dynamically.

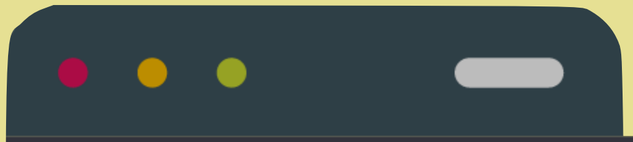
Template Element



It allow us to define HTML content that is not going to be parsed by the browser and it's going to be available for usage only if we clone it.

```
<template id="template1">
   <header>
      <h1>This is a template</h1>
      <p>This content is not rendered initially</p>
   </header>
</template>
```

Template Element Cloning



We clone the template and we append it as a child; typically in **`connectedCallback`** method of the Custom Element.

```
connectedCallback() {
  const template = document.getElementById("template1");
  const content = template.content.cloneNode(true);
  this.appendChild(content);
}
```

WARNING

By default, the nodes of our custom element are part of the same page's DOM, so CSS style declaration applies to all the document.

Template Element

It allow us to define HTML content that is not going to be parsed by the browser and it's going to be available for usage only if we clone it.

```
<template id="template1">
  <style>
    /* this declaration also changes h1s outside of the custom element */
    h1 { color: red }
  </style>
  <header>
      <h1>This is a template</h1>
      <p>This content is not rendered initially</p>
  </header>
</template>
```

DEFINITION

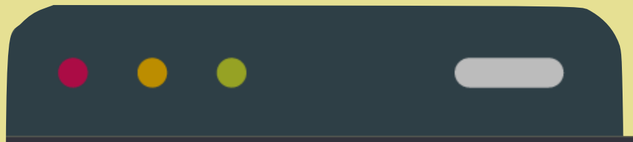
Shadow DOM

A private, isolated DOM tree within a web component that is separate from the main document's DOM tree

Shadow DOM

- Allows more control over styling and encapsulation of functionality of a Custom Element.
- By default, CSS declared in the main DOM won't be applied to the Shadow DOM.
- CSS declared in the Shadow DOM applies only on there.
- There are new pseudo-classes and pseudo-element to allow communication between DOMs in stylesheets.
- It can be opened or closed defining visibility from the outer DOM

Shadow DOM



We have to create a Shadow DOM manually in our Custom Element, typically in the constructor and we save it

```
class MyElement extends HTMLElement {
   constructor() {
      super();
      this.root = this.attachShadow({ mode: "open" });
   }

   connectedCallback() {
      this.root.appendChild( ... )
   }
}
```

Template Element on a Shadow DOM

```
<template id="template1">
  <style>
    /* this declaration ONLY changes h1s inside this custom element */
    h1 { color: red }
  </style>
  <header>
      <h1>This is a template</h1>
      <p>This content is not rendered initially</p>
  </header>
</template>
```

Where to define HTML for a Custom Element

- There are several alternatives
  - Use DOM APIs
  - Use a <template> in the main HTML
  - Use an external HTML file loaded with **fetch** (it can be prefetched)
    - Using innerHTML
    - Using DOMParser

DEFINITION

Declarative Shadow DOM

A way to define Shadow DOM directly in HTML markup using a new set of attributes and tags.

⚠️ *Browser compatibility*

Where to define CSS for a Custom Element

- There are several alternatives
  - Use CSSOM APIs
  - Add a <script> to a <template>
  - Add a <link> in the <template>
  - Use an external CSS file loaded with **fetch** (it can be prefetched) **and injected in the Shadow DOM as a <style>**

Let's code!

IDEAS

Make template engines using tagged literal strings, such as:

```
fem`<input value={0}>`
```

Lit-html uses this technique

Reactive Programming with Proxies

DEFINITION

Proxy

A wrapper object that lets you intercept and modify operations performed on the wrapped object, allowing you to add custom behavior or validation to the object's properties and methods.

Proxy

We wrap an object with a Proxy that can handle different operations, such as when a consumer is setting or getting a property

```
const original = {
  name: 'John Doe',
  age: 30
};

const s = new Proxy(original, handler);

console.log(s.age);   // 30 years old
```

```
const handler = {
  get: function(target, prop) {
    if (property === 'age') {
      return target[prop] + ' years old';
    } else {
      return target[prop];
    }
  }
};
```

Proxy

We can use a proxy for validation, data binding and reactive programming

```
const original = {
  name: 'John Doe',
  age: 30
};

const s = new Proxy(original, handler);
s.age = 40;         // OK
s.age = "hey!";     // Error
```

```
const handler = {
  set: function(target, prop, value) {
     if (property === 'age' &&
         typeof value !== 'number') {
       throw new TypeError('Age not a number');
     } else {
       target[property] = value;
     }
};
```

IMPORTANT

Proxies work with objects only. If you want to do something similar with simpler values, you can use classes with *getter* and *setters*.

DEFINITION

Proxy Handler

Object that contains traps for intercepting and customizing operations performed on a JavaScript proxy object.

DEFINITION

Proxy Trap

Method on a proxy handler object that intercepts and customizes a specific operation performed on the target object.

Most Used Proxy Traps

- get
- set
- has
- deleteProperty
- apply
- construct
- getOwnPropertyDescriptor
- defineProperty
- ownKeys

WARNING

Proxy Traps are also executed when you access a method (function) of the object.

IMPORTANT

With arrays and the usage of [ ] accessor, you can trap `get` and `set` and the index will the the property.

WARNING

To detect changes in arrays, it's not just the accessor [ ], there are functions as well such as `push` that you can trap with `apply`.

Let's code!

Web Animations API

Web Animations API

- For better control in JavaScript you can use the Web Animations API
- It can be used to replace CSS Animations and CSS Transitions
- High-performant
- We need to set an array of keyframes, minimum 2 (initial and final state)
- Each keyframe is an object containing style properties and its values
- The playback rate can be changed
- Playback API for pause/play/finish/reverse

Web Animation Events

- Two events are available: `finish` and `cancel`
- One promise-based event `finished` to use with then() or async/await

Simple Web Animation API

You call animate in the DOM element

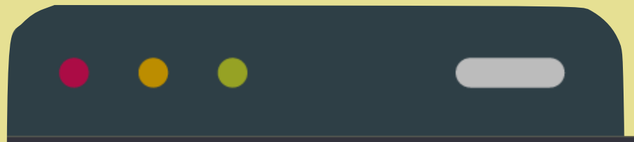
```
const fadeInAnimation = [
  { opacity: 0 },    // initial state
  { opacity: 1 }.    // final state
];
```

Simple Web Animation API

You call animate in the DOM element

```
const fadeInAnimation = [
  { opacity: 0 },   // initial state
  { opacity: 1 }.   // final state
];
element.animate(fadeInAnimation, { duration: 1000 } );
```

Simple Web Animation API



You call animate in the DOM element

```
const myAnimation = [
  { top: "0px", color: "#000" },     // initial state
  { top: "70px", offset: 0.1 },      // executed in first 10%
  { top: "100px", color: "#EEE" }    // final state
];
element.animate(myAnimation, {
   duration: 1000, iterations: 10
} );
```

Simple Web Animation API

Properties: `delay`, `direction`, `duration`, `easing`, `fill`, `iterationStart`, `iterations`, `composite`, `iterationComposite` and `pseudoElement`

```
const myAnimation = [
  { top: "0px", color: "#000" },     // initial state
  { top: "70px", offset: 0.1 },      // executed in first 10%
  { top: "100px", color: "#EEE" }    // final state
];
element.animate(myAnimation, {
   duration: 1000, iterations: 10
} );
```

Simple Web Animation API

There is an API available if you store the animation

```
const animation = element.animate(keyframes,
                                  { duration: 1000 } );

animation.pause();
animation.play();
animation.finish();
animation.onfinish = () ⇒ { };
```

Let's code!

What we've covered

- Vanilla JS
- DOM API
- Fetch
- Design Patterns
- Single Page Applications
- Web Components
- Reactive Programming
- Routing

Main Fears of Vanilla JS

- Routing for Single Page Applications ✅
- Too Verbose and Time Consuming 🧠
- State Management ✅
- Templating ✅
- Complexity ✅
- Reusable Components ✅
- Maintenance 🧠
- Learning Curve ✅
- Browser Compatibility ✅ ❌
- Reinventing the Wheel every time 🧠
- Scalability 🧠

IDEAS

Now that you know Vanilla JS, you can decide to use lightweight libraries and tools one by one, based on your needs.

IDEAS

It's now time to define the design patterns you prefer, and create your own unique library for your needs.

THANKS! 🤗

YOU DON'T NEED THAT LIBRARY

MAXIMILIANO FIRTMAN