

# Web Components

Custom Elements, Shadow DOM, Templates, lit-html



**SoftUni Team**  
Technical Trainers



**SoftUni**



Software University

<https://softuni.org>

1. What are Web Components?
2. Creating Elements & Shadow DOM
3. HTML Templates & Slots
4. Component Lifecycle
5. Extending HTML Elements
6. Lit HTML





sli.do

#js-advanced



# Web Components

Web Platform API

# What are Web Components?



- Web components are a set of **web platform APIs** that allow you to create:
  - Custom, reusable, **encapsulated HTML tags** to use in web pages and web apps
- Custom web components **benefits**:
  - Will work across **modern browsers**
  - Can be used with any JavaScript **library** or **framework** that works with HTML

# Web Components - Specification

- Web Components are based on **four** main specifications:
  - Custom Elements – lays the foundation of **designing** and using **new** types of **DOM elements**
  - Shadow DOM – defines how to use **encapsulated** style and markup
  - ES Modules – import/export
  - HTML Template – declare **fragments of markup** that go unused on page load, but are instantiated later





# Creating Web Components

Shadow DOM

- Use JavaScript to define a new HTML element and its tag with the **customElements** global

```
class AppRoot extends HTMLElement {...}
```

```
window.customElements.define('app-root', AppRoot);
```

- To use the new tag

```
<app-root></app-root>
```



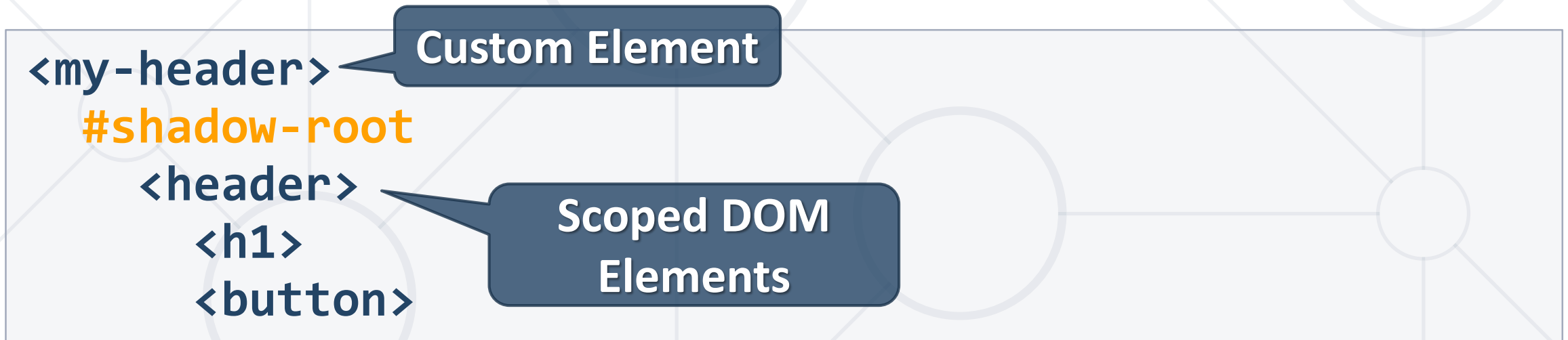
# Shadow DOM



- Shadow DOM is a new DOM feature that helps you build components
  - You can think of shadow DOM as a **scoped subtree** inside your element
- Shadow DOM lets you place the children in a scoped subtree, so document-level CSS **can't restyle** it
- The **shadow root** is the top of the shadow tree

# Shadow DOM Example

- Consider a header component that includes a **page title** and a **menu button**
- The subtree below shadow root is called a shadow tree:



# Shadow Root & Host

- The shadow root is the top of the shadow tree
- The element that the tree is attached to (**<my-header>**) is called the shadow host
  - Has a property called **shadowRoot** that refers to the shadow root
- The shadow root has a **host** property that identifies its host element



- You can add a shadow tree to an element imperatively by calling **attachShadow**:

```
let div = document.createElement('div');
```

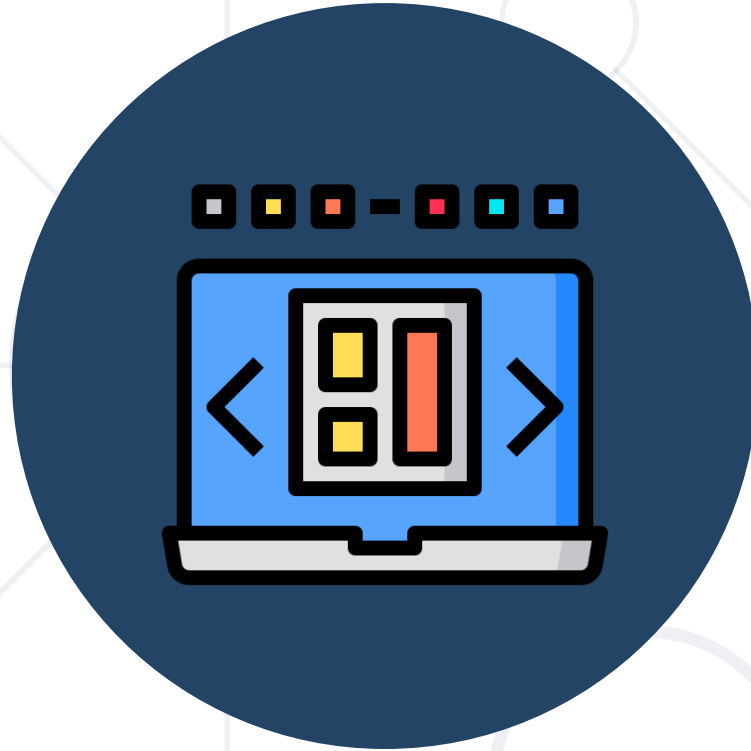
```
let shadowRoot = div.attachShadow({mode: 'closed'});
```

```
shadowRoot.innerHTML = '<h1>Hello Shadow DOM</h1>';
```

Could be **open**  
or **closed**

# First Component: Example

```
class AppRoot extends HTMLElement {  
  constructor() {  
    super();  
  
    const root = this.attachShadow({ mode: 'closed' });  
  
    const div = document.createElement('div');  
    div.innerHTML = '<h1>Hello Shadow DOM</h1>';  
  
    root.appendChild(div);  
  }  
}  
  
customElements.define('app-root', AppRoot);
```




# HTML Templates

Creating Templates & Passing Slots

# The Template Tag

- The **<template>** tag is used as a container to hold some HTML content hidden from the user when the page loads
- The content inside **<template>** can be rendered later with a JavaScript



```
<template>
  <div class="container">
    <h1>App Root Name</h1>
  </div>
</template>
```

- `<app-root></app-root>`

```
const template = document.createElement('template');
const div = document.createElement('div');
const h1 = document.createElement('h1');
h1.textContent = this.getAttribute('app-name');
div.appendChild(h1);
template.innerHTML = div.innerHTML;

root.appendChild(template.content.cloneNode(true));
```

Retrieve passed  
attribute

Node **deep** cloning



# Slots & Named Slots

- The HTML `<slot>` element is a placeholder inside a web component that you can fill with your own markup
- This lets you create **separate DOM trees** and present them together
- A named slot is a `<slot>` element with a **name attribute**



# Named Slots: Example

- Omit the passing of the attribute:

```
<app-root>  
  <h1 slot="title">My App Name</h1>  
</app-root>
```

```
const appRootTemplate = html`  
  <div>  
    <slot name="title"></slot>  
  </div>  
`
```



# Component Lifecycle

Handling Certain Events

# Component Lifecycle



- Web Components have their own **lifecycle**
- The following events happen in a Web Component's lifecycle:
  - Element is **inserted** into the DOM
  - Updates when UI **event** is **triggered**
  - Element **deleted** from the DOM
- There are **callback functions** that capture these lifecycle events and let us handle them accordingly

# Lifecycle Hooks

- The following **lifecycle hooks** are in a web component:
  - `constructor()`
  - `connectedCallback()`
  - `disconnectedCallback()`
  - `attributeChangedCallback()`
  - `adoptedCallback()`



# Example: constructor()

- The **constructor()** is called when the web component is **created**
- It's called when we create the **shadow DOM** and it's used for setting up listeners and **initialize** a component's **state**

```
this._root = this.attachShadow({ mode: 'closed' });  
  
this.state = {  
  title: this.getAttribute('app-title')  
}
```

# Example: `connectedCallback()`

- This is called when an element is **added** to the DOM
- It means that we can safely **set attributes**, **fetch resources**, run set up code or **render templates**

```
connectedCallback() {  
  // Load some data using fetch or axios  
}
```

# Example: disconnectedCallback()

- This is called when the element is **removed** from the DOM
- Therefore, it's an ideal place to add **cleanup logic** and to free up resources

```
disconnectedCallback() {  
    // clear timers or intervals  
}
```



# Example: attributeChangedCallback()

- In this callback, we can get the **value** of the **attributes** as they're assigned in the code
- We can add a **static get observedAttributes()** hook to define what attribute values we observe:

```
static get observedAttributes() {  
  return ['app-title', 'foo', 'bar']  
}
```

# Example: attributeChangedCallback()

- The callback receives **three** parameters:

```
attributeChangedCallback(name, oldValue, newValue) {  
  console.log(`${name}'s value has been changed  
    from ${oldValue} to ${newValue}`);  
}
```



# Extending Native HTML Elements

# Extending HTML Elements



- Custom elements allows you to **extend existing** (native) HTML elements as well as other custom elements
- If you aren't happy with the regular **<button>** element, for example, you can override it
- **NOTE:** This feature is **not supported** in WebKit (*August 2022*)

# Example: Extending Button

- Extend the native element, and add a third parameter to the define method:

```
class FancyButton extends HTMLElement {  
  constructor() {  
    self = super();  
    self.textContent = 'Custom Button';  
  }  
}  
customElements.define('fancy-button', FancyButton,  
  { extends: 'button' }  
);
```

# Example: Extending Button

- After that add the **"is" attribute** and the name of the custom button element:

```
<button is="fancy-button">  
</button>
```

- This should render a button with text content **"Custom Button"**



# Lit-html

Creating templates with ease

# What is lit-html?

- Simple, modern, safe, small and fast **HTML templating library** for JavaScript
- Lets you write HTML templates in JavaScript using **template literals** with embedded JavaScript expressions
- Identifies the **static** and **dynamic parts** of your templates so it can efficiently **update** just the changed portions





- Installation:

```
npm install lit-html
```

- To use lit-html, import it via a path:

```
<script type="module">  
  import { html, render }  
    from './node_modules/lit-html/lit-html.js';  
  ...  
</script>
```

Path to main file (use  
**live-server** to start)

- lit-html has two main APIs:
  - The **html template tag** used to write templates.
  - The **render()** function used to render a template to a DOM container


```
const appRootTemplate = // Same as previous template
```

```
render(  
  appRootTemplate(this.state), this._root,  
  { host: this }  
);
```

Template **event**  
**context**

# Tag Functions / Tagged Templates

- A tagged template is a **function call** that uses a **template literal** from which to get its arguments



```
// Tag Function Call  
greet`I'm ${name}. I'm ${age} years old.`
```

- Create a greet function and just log the arguments:

```
function greet() {  
  console.log(arguments[0]); // array  
  console.log(arguments[1]); // name  
  console.log(arguments[2]); // age  
}
```

- In addition to using expressions in the text content of a node, you can bind them to a node's attribute and property values, too:

```
const myTemplate = (data) => html`<div  
  class=${data.cssClass}>Stylish text.</div>`
```

- Use the **?** prefix for a **boolean** attribute binding:

```
const myTemplate = (data) => html`<div  
  ?disabled=${!data.active}>Stylish text.</div>`
```

- You can also bind to a **node's JavaScript properties** using the **.** **prefix** and the property name:

```
const myTemplate = (data) => html`<input  
  .value=${data.value}></input>`;
```

- You can use property bindings to **pass** complex data **down** the tree to subcomponents:

```
const myTemplate = (data) => html`<my-list  
  .listItems=${data.items}></my-list>`;
```

- Templates can also include declarative event listeners
- An event listener looks like an attribute binding, but with the **prefix @** followed by an **event name**:

```
const appRootTemplate = (ctx) => html`  
  <div>  
    <h1 @click=${ctx.handleClick}>${ctx.title}</h1>  
  </div>  
`
```

- lit-html has **no built-in control-flow** constructs. Instead you use normal JavaScript expressions and statements:

```
html`  
  ${user.isloggedIn  
    ? html`Welcome ${user.name}`  
    : html`Please log in`  
  }  
`;  
`;
```

- To render lists, you can use **Array.map** to transform a list of data into a list of templates:

```
html`  
  <ul>  
    ${items.map((item) => html`<li>${item}</li>`)}  
  </ul>  
`;  
`;
```



- The **classMap directive** lets you set a **group of classes** based on an object:

```
import { classMap } from './node_modules/lit-html/directives/class-map.js';

const itemTemplate = (item) => {
  const classes = { selected: item.selected };
  return html`<div class="menu-item
    ${classMap(classes)}">Classy text</div>`;
}
```

# Directives: styles and styleMap

- You can use the **styleMap directive** to set **inline styles** on an element in the template:

```
import { styleMap } from './node_modules/lit-html/directives/style-map.js';
```

```
const styles = {  
  color: myTextColor,  
  backgroundColor: highlight ? myHighlightColor :  
    myBackgroundColor  
};
```

```
html`<div style=${styleMap(styles)}>Hi there!</div>`;
```

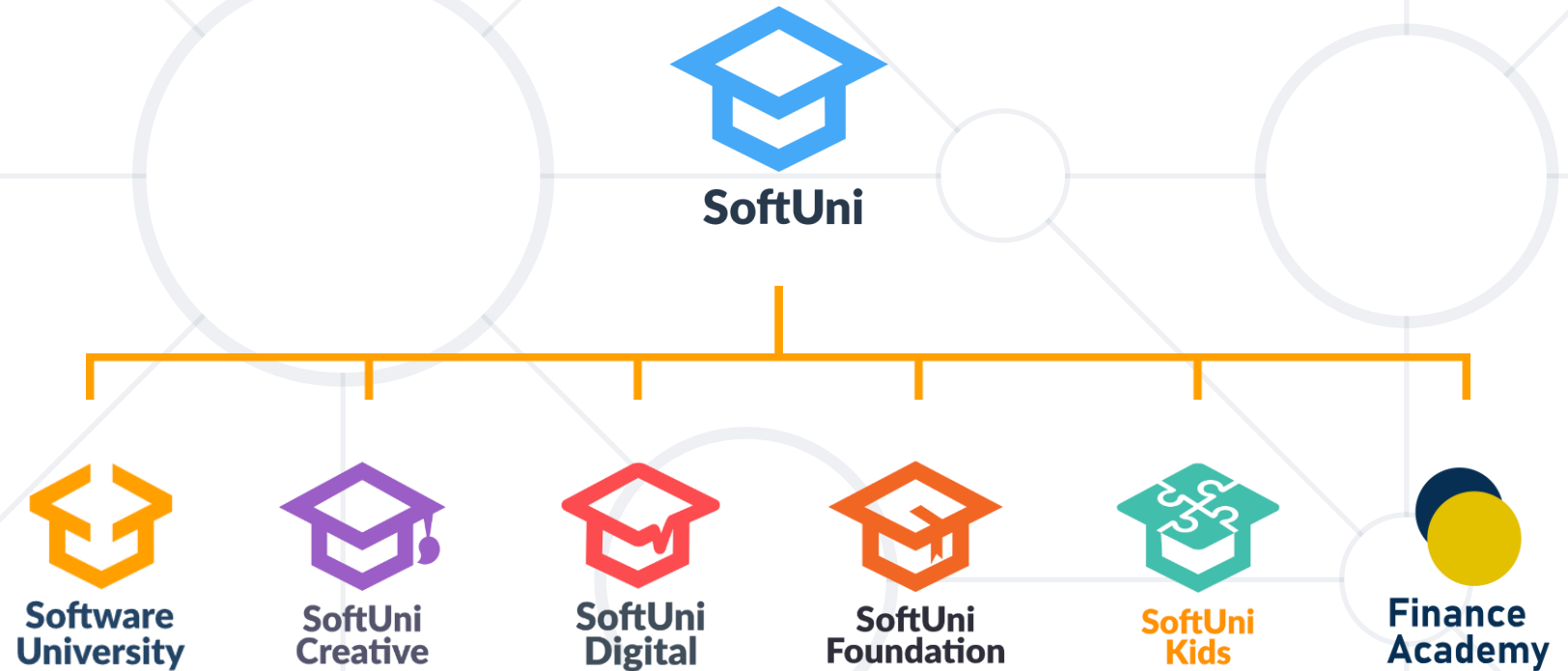
- Repeats a **series of values** generated from an iterable, and **updates** those items **efficiently** when the iterable changes:

```
import { repeat } from './node_modules/lit-html/directives/repeat';

const myTemplate = () => html`
  <ul>
    ${repeat(items, (i) => i.id, (i, index) => html`
      <li>${index}: ${i.name}</li>`)}
  </ul>
`;
```

- Here are some additional libraries you can try out:
  - Hybrids - a UI library for creating Web Components with simple and functional API
  - Lit Element - uses lit-html to render into the element's shadow DOM and adds API to help manage element properties and attributes
  - Polymer - provides a set of features for creating custom elements
  - Stencil - an open-source compiler that generates standard-compliant web components

# Questions?



# SoftUni Diamond Partners

**SUPER  
HOSTING  
.BG**



**Coca-Cola HBC  
Bulgaria**



**POKERSTARS**  
POKER | CASINO | SPORTS  
a Flutter International brand

**INDEAVR**  
Serving the high achievers



**AMBITIONED**

 **DRAFT  
KINGS**



**SOFTWARE  
GROUP**

createX



**Postbank**

Решения за твоето утре

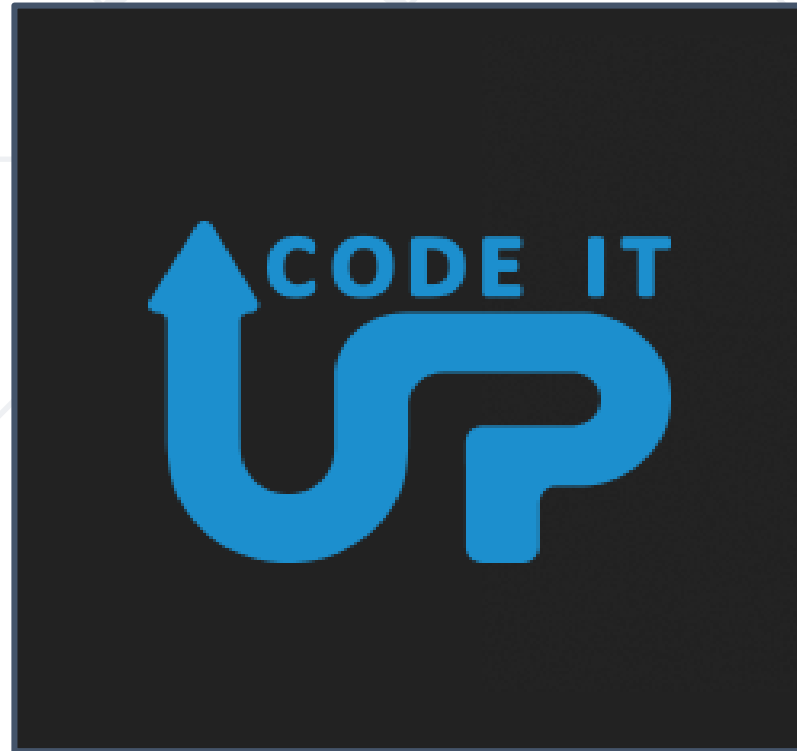


**BOSCH**

**DXC**  
TECHNOLOGY



**SmartIT**



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>





- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg), [about.softuni.bg](http://about.softuni.bg)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](http://forum.softuni.bg)

