formation.tech

# Web Components

# Web Components - Introduction

- Web Components is a set of Web APIs allowing to create custom HTML tags/ elements
- It is composed of 3 Web APIS
  - Custom Elements
  - Shadow DOM
  - HTML Templates and Slots
- Support in modern browsers is complete
- Many libraries can be used to fasten the development : Lit, Stencil, X-Tag...
- React, Angular or Vue components can be wrapped in native Web Components

| Browser support | | | | | |
|---|---|---|---|---|---|
| HTML TEMPLATES | ✓ | ✓ | ✓ | ✓ | ✓ |
| CUSTOM ELEMENTS | ✓ | ✓ | ✓ | ✓ | ✓ |
| SHADOW DOM | ✓ | ✓ | ✓ | ✓ | ✓ |
| ES MODULES | ✓ | ✓ | ✓ | ✓ | ✓ |

formation.tech

# Custom Elements

# Custom Elements - Introduction

- Custom Elements is a Web API that enable the possibility to create custom tags in your app
- It is fully native so we don't need to load an external library like React, Angular or Vue
- We can create our own elements (Autonomous custom element) or extend existing ones (Customized built-in element)
- Lifecycle callbacks are called at specific moment of the element life (when it appears in the DOM, when it receives new attributes...)

# Custom Elements - Element name

- The element name (e.g. `<my-counter>`) :
  - must start with a letter
  - must be lowercase
  - must contain a dash (-) to be forward compatible with future HTML, SVG or MathML names
  - must not be one of the following : annotation-xml, color-profile, font-face, font-face-src, font-face-uri, font-face-format, font-face-name, missing-glyph
  - can contain any letter or emoji (e.g. `<math-α>` or `<emotion-😍>` are valid)

# Custom Elements - Autonomous custom element

- To create an Autonomous custom element we have to create a class that inherits from the HTMLElement interface
- Then you have to register your new tag name with a CustomElementRegistry (the global variable customElements)

```
class Counter extends HTMLElement {}

customElements.define('my-counter', Counter);
```

# Custom Elements - Autonomous custom element

- An Autonomous custom element can be used
  - As any other HTML element :

```
<body>
  <my-counter></my-counter>
</body>
```

  - Using the document.createElement method

```
const myCounterEl = document.createElement('my-counter');
document.body.append(myCounterEl);
```

# Custom Elements - Customized built-in element

- A customized build-in element must inherit from the HTML*Element interface it customize
- The option extends must be pass to the CustomElementRegistry define method :

```
class CounterHTMLElement extends HTMLButtonElement {}

customElements.define('my-counter', CounterHTMLElement, { extends: 'button' });
```

# Custom Elements - Customized built-in element

- A customized built-in element can be used
  - In HTML

```
<body>
    <button is="my-counter"></button>
</body>
```

  - Using the document.createElement method

```
const myCounterEl = document.createElement('button', { is: 'my-counter' });
document.body.append(myCounterEl);
```

# Custom Elements - Lifecycle callbacks

- Lifecycle callbacks are methods that are called automatically during the lifetime of a web component

```
class Counter extends HTMLElement {
  constructor() { super(); console.log('constructor'); }
  connectedCallback() { console.log('connectedCallback'); }
  disconnectedCallback() { console.log('disconnectedCallback'); }
  adoptedCallback() { console.log('adoptedCallback'); }
  attributeChangedCallback() { console.log('attributeChangedCallback'); }
  formAssociatedCallback() { console.log('formAssociatedCallback'); }
  formDisabledCallback() { console.log('formDisabledCallback'); }
  formResetCallback() { console.log('formResetCallback'); }
  formStateRestoreCallback() { console.log('formStateRestoreCallback'); }
}

customElements.define('my-counter', Counter);
```

# Custom Elements - Lifecycle callbacks

- constructor
  - it is not strictly a lifecycle callback, it is called when the class is instantiated
  - has to call super() at the beginning
  - doesn't have access to the DOM
  - may define event listeners
  - can call attachInternals or attachShadow

```
class Counter extends HTMLElement {
  constructor() {
    super();
    this.count = 0;
    this._internals = this.attachInternals();
    this.addEventListener('click', this._onClick.bind(this));

    this._internals.role = 'button';
  }
}
```

# Custom Elements - Lifecycle callbacks

- connectedCallback
  - called when the element is appended to the DOM
  - also called when the element is moved in the document tree (appended elsewhere)
  - has access to the DOM

```
class Counter extends HTMLElement {
  constructor() {
    super();
    this.count = 0;
  }
  connectedCallback() {
    this.innerText = this.count;
  }
}
```

```
<my-counter></my-counter> <!-- connectedCallback called -->
<div></div>
<script>
  const myCounterEl = document.querySelector('my-counter');
  const divEl = document.querySelector('div');
  setTimeout(() => {
    divEl.appendChild(myCounterEl); // connectedCallback called
  }, 2000);
</script>
```

# Custom Elements - Lifecycle callbacks

- disconnectedCallback
  - called when the element is removed from the DOM
  - also called when the element is moved in the document tree (appended elsewhere)
  - must be used to prevent optimization issues and memory leaks

```
class Counter extends HTMLElement {
  constructor() {
    super();
    this.count = 0;
    this._handleClick = this._handleClick.bind(this);
  }
  _handleClick = () => {
    this.count++;
    this._updateRendering();
  }
  connectedCallback() {
    this._updateRendering();
    this.addEventListener('click', this._handleClick);
  }
  disconnectedCallback() {
    this.removeEventListener('click', this._handleClick);
  }
  _updateRendering() {
    this.innerText = this.count;
  }
}
```

# Custom Elements - Lifecycle callbacks

- attributeChangedCallback
  - called when an attribute is set of modified
  - watched attributes must be defined with the static observedAttributes property

```
class Counter extends HTMLElement {
  count = 0; // ES2022 class properties
  static observedAttributes = ["count"];
  attributeChangedCallback(name, oldValue, newValue) {
    if (name === 'count') {
      this.count = Number(newValue);
    }
    this._updateRendering();
  }
  _updateRendering() {
    this.innerText = this.count;
  }
}
```

```
<my-counter count="3"></my-counter> <!-- attributeChangedCallback called -->
<script>
  const myCounterEl = document.querySelector('my-counter');
  setTimeout(() => {
    myCounterEl.setAttribute('count', '4'); // attributeChangedCallback called
  }, 2000);
</script>
```

# Custom Elements - Lifecycle callbacks

- adoptedCallback
  - called when a custom element is adopted by another document (e.g. in an iframe)

```
<my-counter></my-counter>
<iframe></iframe>
<script>
  const myCounterEl = document.querySelector('my-counter');
  const iframeEl = document.querySelector('iframe');
  setTimeout(() => {
    iframeEl.contentDocument.body.appendChild(myCounterEl); // adoptedCallback
called
  }, 2000);
</script>
```

# Custom Elements - Syncing props and attrs

- To keep properties and attributes synchronized we use JavaScript get and set syntax
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set

```
class Counter extends HTMLElement {
  static observedAttributes = ["count"];
  get count() {
    return this.getAttribute('count');
  }
  set count(val) {
    this.setAttribute('count', val);
  }
  connectedCallback() {
    this._updateRendering();
  }
  attributeChangedCallback(name, oldValue, newValue) {
    this._updateRendering();
  }
  _updateRendering() {
    this.innerText = this.count;
  }
}
```

# Custom Elements - Syncing property and attribute

```
<my-counter count="1"></my-counter>
<script>
  const myCounterEl = document.querySelector('my-counter');
  console.log(myCounterEl.count); // 1
  setTimeout(() => {
    myCounterEl.count = '2';
    console.log(myCounterEl.getAttribute('count')); // 2
    setTimeout(() => {
      myCounterEl.setAttribute('count', '3');
      console.log(myCounterEl.count); // 3
    }, 1000);
  }, 1000);
</script>
```

# Custom Elements - Custom form controls

- Custom form controls must be declared using the static fromAssociated property
- The attachInternals method must be called in the constructor or in a ES2022 class property
- attachInternals returns a ElementInternals object which provides utilities for the custom element to work in a form context

```
class Counter extends HTMLElement {
  static formAssociated = true;
  static observedAttributes = ["count"];
  #internals = this.attachInternals(); // ES2022 Private class property
  connectedCallback() {
    this._updateRendering();
  }
  attributeChangedCallback(name, oldValue, newValue) {
    this.#internals.setFormValue(newValue);
    this._updateRendering();
  }
  _updateRendering() {
    this.innerText = this.getAttribute('count');
  }
}
```

# Custom Elements - Custom form controls

- Your element can now be used as a form control :

```
<form>
  <my-counter name="age" count="18"></my-counter>
  <button>Send</button>
</form>
<script>
  const formEl = document.querySelector('form');
  formEl.addEventListener('submit', (event) => {
    event.preventDefault();
    const formData = new FormData(formEl);
    console.log(Object.fromEntries(formData)); // {age: '18'}
  });
</script>
```

- You can bind internals to the element using get syntax :

```
class Counter extends HTMLElement {
  static formAssociated = true;
  #internals = this.attachInternals(); // ES2022 Private class property
  get form() { this.#internals.form }
  get validity() { this.#internals.validity }
  get name() { this.getAttribute('name') }
}
```

# Custom Elements - Custom form controls

- Custom Form lifecycle callbacks

```
class Counter extends HTMLElement {
  static formAssociated = true;
  #internals = this.attachInternals();
  formAssociatedCallback() { console.log('formAssociatedCallback'); }
  formDisabledCallback() { console.log('formDisabledCallback'); }
  formResetCallback() { console.log('formResetCallback'); }
  formStateRestoreCallback() { console.log('formStateRestoreCallback'); }
}
```

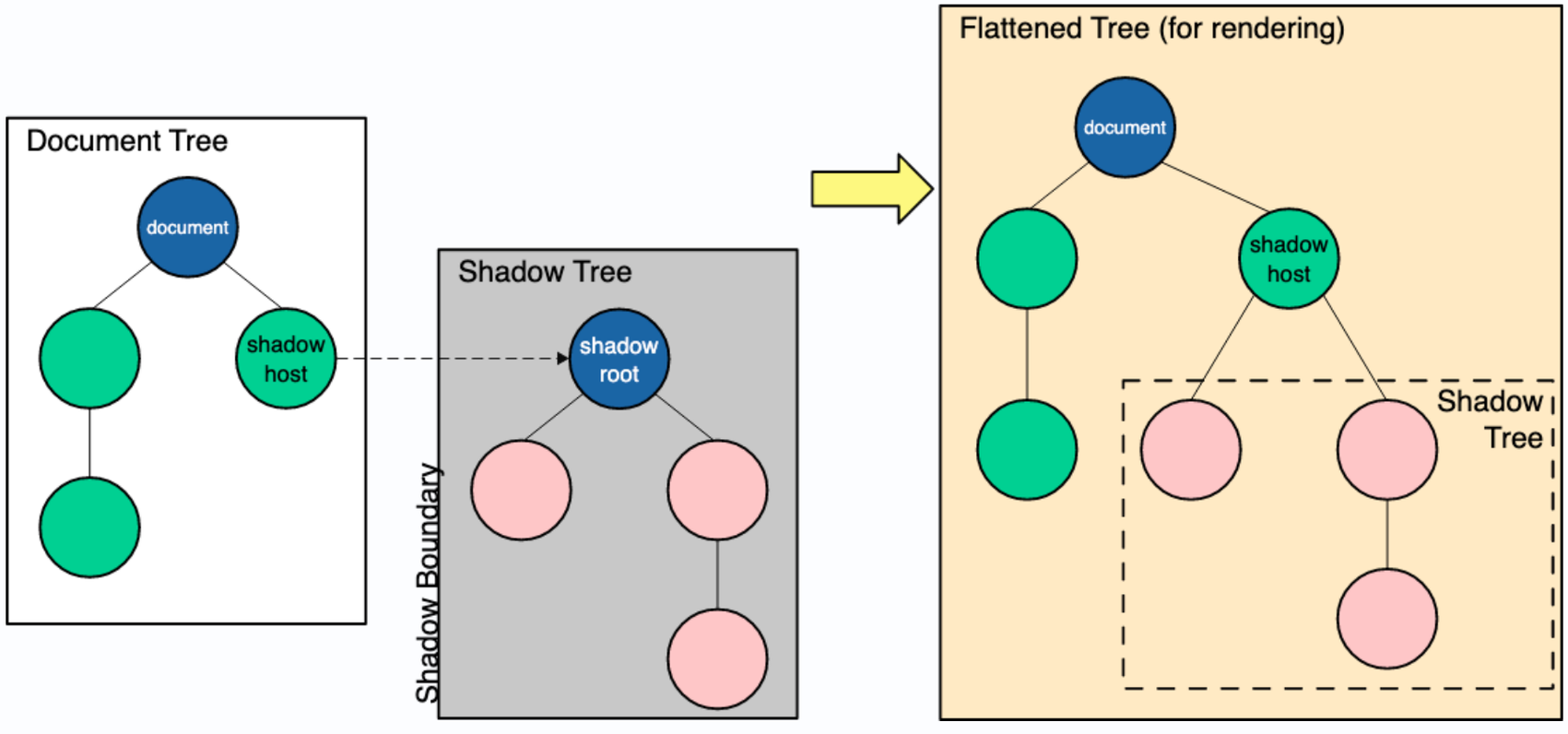
- formAssociatedCallback : when the control is attached to the form
- formDisabledCallback : when the control or parent fieldset is disabled
- formResetCallback : when the button reset is pressed
- formStateRestoreCallback : when the browser fills the controls (e.g. autocomplete)
- Learn More : https://web.dev/more-capable-form-controls/

formation.tech

# Shadow DOM

# Shadow DOM - Introduction

- Shadow DOM is a Web APIs that is used to create a hidden separated DOM in a custom element
- It keeps markup, style and behavior hidden and separated from the rest of the page so reduce risk of conflicts
- Some HTML elements already follow this principle such as the controls of a video element



# Shadow DOM - Creation

- Shadow DOM can be created using the attachShadow method of an Element
- This method takes a mode option which can be set to :
  - open : the parent DOM can access to the shadow DOM using the myCustomElem.shadowRoot property
  - closed : the shadow DOM is not accessible (myCustomElem.shadowRoot === null)

```
class Counter extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });
    shadow.innerHTML = `<button>0</button>`;
  }
}
```

```
▼<my-counter>
  ▼#shadow-root (closed)
      <button>0</button>
  </my-counter>
```

# Shadow DOM - Scoped CSS

- Simply create a style or link tag inside the shadow DOM and it will apply only to your component :

```
class Counter extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });

    const styleEl = document.createElement('style');
    styleEl.innerText = `
      button {
        background: yellow;
      }
    `;

    const buttonEl = document.createElement('button');
    buttonEl.innerText = 'Shadow DOM';
    shadow.append(styleEl, buttonEl);
  }
}
```

```
<body>
  <my-counter></my-counter>
  <button>DOM</button>
</body>
```

# Shadow DOM - CSS Selectors

- There a 4 CSS Selectors pseudo-classes related to Web Components :
  - :defined that matches all custom elements defined with customElements.define()
  - :host, inside a Shadow DOM, refers to the custom element containing the CSS
  - :host(), inside a Shadow DOM, refers to the custom element containing the CSS combined with another selector in the function parameter
  - :host(), inside a Shadow DOM, refers to the custom element containing the CSS combined with another selector in the function parameter that apply to ancestors

# Shadow DOM - CSS Selectors

```
class Counter extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });

    const styleEl = document.createElement('style');
    styleEl.innerText = `
      :host {
        display: block;
      }
      :host(:hover) {
        background: yellow;
      }
      :host-context(#box) {
        color: blue;
      }
    `;

    shadow.append(styleEl, 'Shadow DOM');
  }
}
```

```
<body>
  <div id="box">
    <my-counter></my-counter>
  </div>
  <my-counter></my-counter>
  <my-counter></my-counter>
</body>
```

Shadow DOM
Shadow DOM
Shadow DOM

# Shadow DOM - Custom properties

- To customize a Web Component with a Shadow DOM from the parent Document we have to use Custom properties(CSS Variables)
- Custom properties names are prefixed by two dashes `--`
- To use the property we use the `var()` function
  `var(--my-custom-property)`
  `var(--my-custom-property, default-value)`
- Properties can pass through the shadowed component

# Shadow DOM - CSS Properties

```
class Counter extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });

    const styleEl = document.createElement('style');
    styleEl.innerText = `
      button {
        background: var(--myBgColor, yellow);
      }
    `;

    const buttonEl = document.createElement('button');
    buttonEl.innerText = 'Shadow DOM';

    shadow.append(styleEl, buttonEl);
  }
}
```

```
<style>
  #last {
    --myBgColor: lightgreen;
  }
</style>
<my-counter></my-counter>
<my-counter style="--myBgColor: lightblue"></my-counter>
<my-counter id="last"></my-counter>
```

Shadow DOM | Shadow DOM | Shadow DOM

formation.tech

# HTML Templates and slots

# HTML Templates and slots - Template

- Using innerHTML to fill an element is inefficient in a web component as the HTML string would be parsed for each component instance
- Instead we can use HTML Templates
- Templates are not rendered in the browser
- Their content is only parsed once

```
const templateEl = document.createElement('template');
templateEl.innerHTML = `
<style>
button {
  background: var(--myBgColor, yellow);
}
</style>
<button>0</button>
`;

class Counter extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });
    shadow.append(templateEl.content.cloneNode(true));
  }
}
```

# HTML Templates and slots - Slots

- Slot give the possibility to project the content of a web component in the Shadow DOM

```
const templateEl = document.createElement('template');
templateEl.innerHTML = `
<button>
  <slot></slot> <!-- content will appear here -->
</button>
`;

class Counter extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });
    shadow.append(templateEl.content.cloneNode(true));
  }
}
```

```
<my-counter>2</my-counter>
<my-counter>10</my-counter>
<my-counter>30</my-counter>
```

2 10 30

# HTML Templates and slots - Slots

- Slot can define default content

```
const templateEl = document.createElement('template');
templateEl.innerHTML = `
<button>
  <slot>0</slot> <!-- content will appear here -->
</button>
`;

class Counter extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });
    shadow.append(templateEl.content.cloneNode(true));
  }
}
```

```
<my-counter>2</my-counter>
<my-counter>10</my-counter>
<my-counter>30</my-counter>
```

2 10 30

# HTML Templates and slots - Slots

- Multiple slots can be used with names

```
const templateEl = document.createElement('template');
templateEl.innerHTML = `
<style>
  :host {
    display: block;
    border: 1px solid black;
  }
  .title {
    background: lightblue;
  }
</style>
<div class="title">
  <slot name="title"></slot>
</div>
<div class="content">
  <slot name="content"></slot>
</div>
`;

class Card extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });
    shadow.append(templateEl.content.cloneNode(true));
```

2 10 30

```
<my-card>
  <div slot="title">Hello</div>
  <p slot="content">Lorem ipsum...</p>
</my-card>
```

Hello

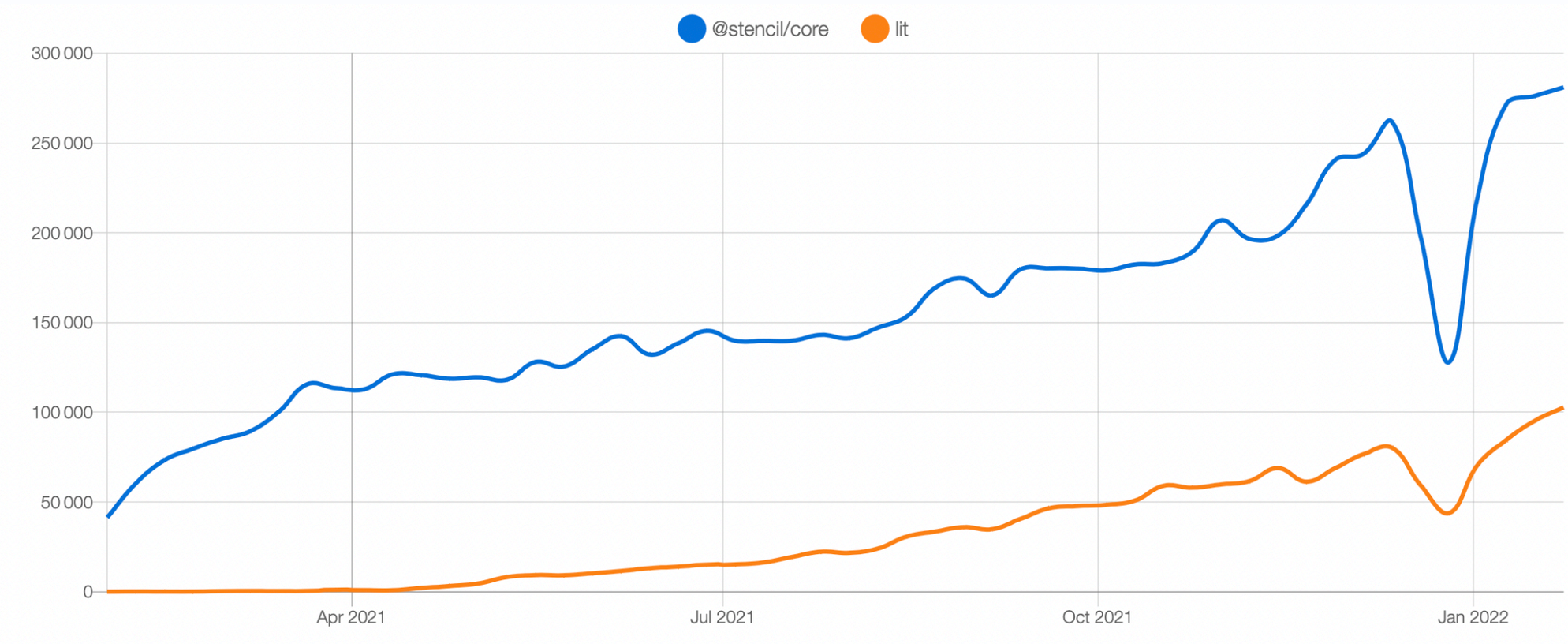
Lorem ipsum...

formation.tech

# Web Component Librairies

# Web Component Librairies

- Some libraries are dedicated to web component creation :
  - Lit : created by Google, successor of lit-html and Polymer
  - Stencil : created by Ionic
  - Lightning Web Components : created by SalesForce



# Web Component Librairies

- We can also wrap components created by popular librairies :
  - Angular
    https://angular.io/guide/elements
  - React
    https://reactjs.org/docs/web-components.html#using-react-in-your-web-components
    https://github.com/bitovi/react-to-webcomponent#readme
  - Preact
    https://github.com/preactjs/preact-custom-element
  - Vue
    https://v3.vuejs.org/guide/web-components.html#definecustomelement

# Web Component Librairies

- A web component online IDE :
  https://webcomponents.dev/new
- A catalog of open source components :
  https://www.webcomponents.org/
- Resources :
  https://developers.google.com/web/fundamentals/web-components?hl=en
  https://developer.mozilla.org/en-US/docs/Web/Web_Components