



formation.tech

Formation Web Component et LitElement @Rolex

Romain Bohdanowicz

Twitter : @bioub - <https://github.com/bioub>

<http://formation.tech/>



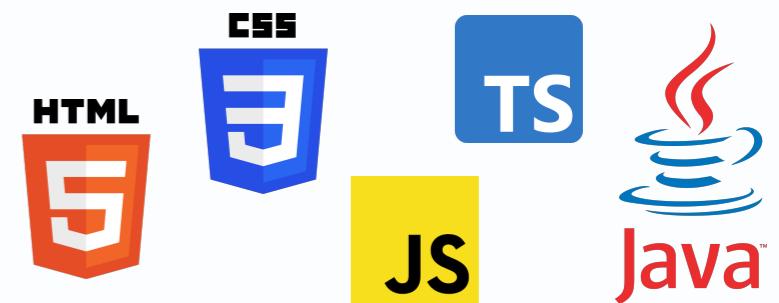
formation.tech

Introduction



Introduction - Formateur

- Romain Bohdanowicz
Ingénieur EFREI 2008, spécialité en Ingénierie Logicielle
- Expérience
Formateur/Développeur Freelance depuis 2006
Plus de 800 formations animées
- Langages
Expert : HTML / CSS / JavaScript / TypeScript / PHP / Java
Notions : C / C++ / Objective-C / C# / Python / Bash / Batch
- Certifications
PHP / Zend Framework / Node.js
- A propos
Premier site web à 12 ans (HTML/JS/PHP)
Triathlète du dimanche



Introduction - Horaires



- Matin
 - 8h30 - 10h20
 - Pause 20 minutes
 - 10h40 - 12h30
- Après-midi
 - 13h30 - 14h50
 - Pause 20 minutes
 - 15h10 - 16h30
- Questionnaire de satisfaction à remplir en fin de formation :
<https://stagiaire.formation.tech/>

Introduction - formation.tech



- Organisme de formation depuis 2016
- Certifié Qualiopi
- ~50 formations au catalogue
- Une dizaine de formateurs indépendants
- Formations en français ou anglais
- <https://formation.tech/>



Introduction - Et vous ?



- Pré-requis ?
- Rôle dans votre société ?
- Intérêt / objectif de cette formation ?



formation.tech

Web Components

Web Components - Qu'est-ce qu'un composant ?

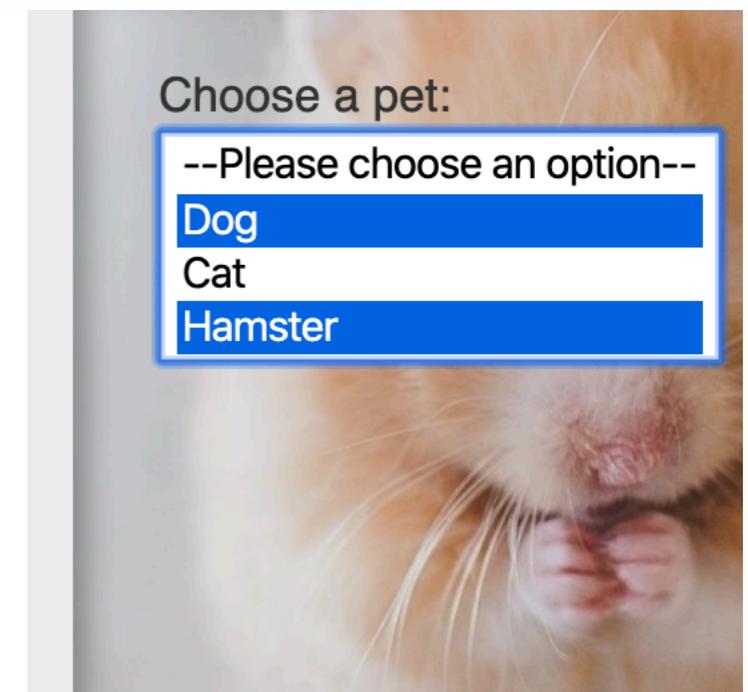


- Un composant est un moyen simple et isolé de regrouper :
 - Du HTML pour la structure de données
 - Du CSS pour la mise en forme
 - Du JavaScript pour le comportement

- Exemple : la balise select

<https://developer.mozilla.org/fr/docs/Web/HTML/Element/select>

```
<label for="pet-select">Choose a pet:</label>  
  
<select name="pets" id="pet-select" multiple>  
  <option value="">--Please choose an option--</option>  
  <option value="dog">Dog</option>  
  <option value="cat">Cat</option>  
  <option value="hamster">Hamster</option>  
  <option value="parrot">Parrot</option>  
  <option value="spider">Spider</option>  
  <option value="goldfish">Goldfish</option>  
</select>
```



Web Components - Un ensemble d'APIs Web



- › Les Web Components nous permettent de créer nos propres balises modernes
- › Ils sont implémentés via 3 APIs Web :
 - Custom Elements
 - Shadow DOM
 - HTML Templates and Slots
- › Tous les navigateurs modernes les supportent
- › Plusieurs bibliothèques peuvent être utilisées pour accélérer leur développement : Lit, Stencil, X-Tag...
- › Les composants React, Angular ou Vue peuvent être encapsulés dans des Web Components et inversement

Browser support	Chrome	Safari	Firefox	Edge
HTML TEMPLATES	✓	✓	✓	✓
CUSTOM ELEMENTS	✓	✓	✓	✓
SHADOW DOM	✓	✓	✓	✓
ES MODULES	✓	✓	✓	✓

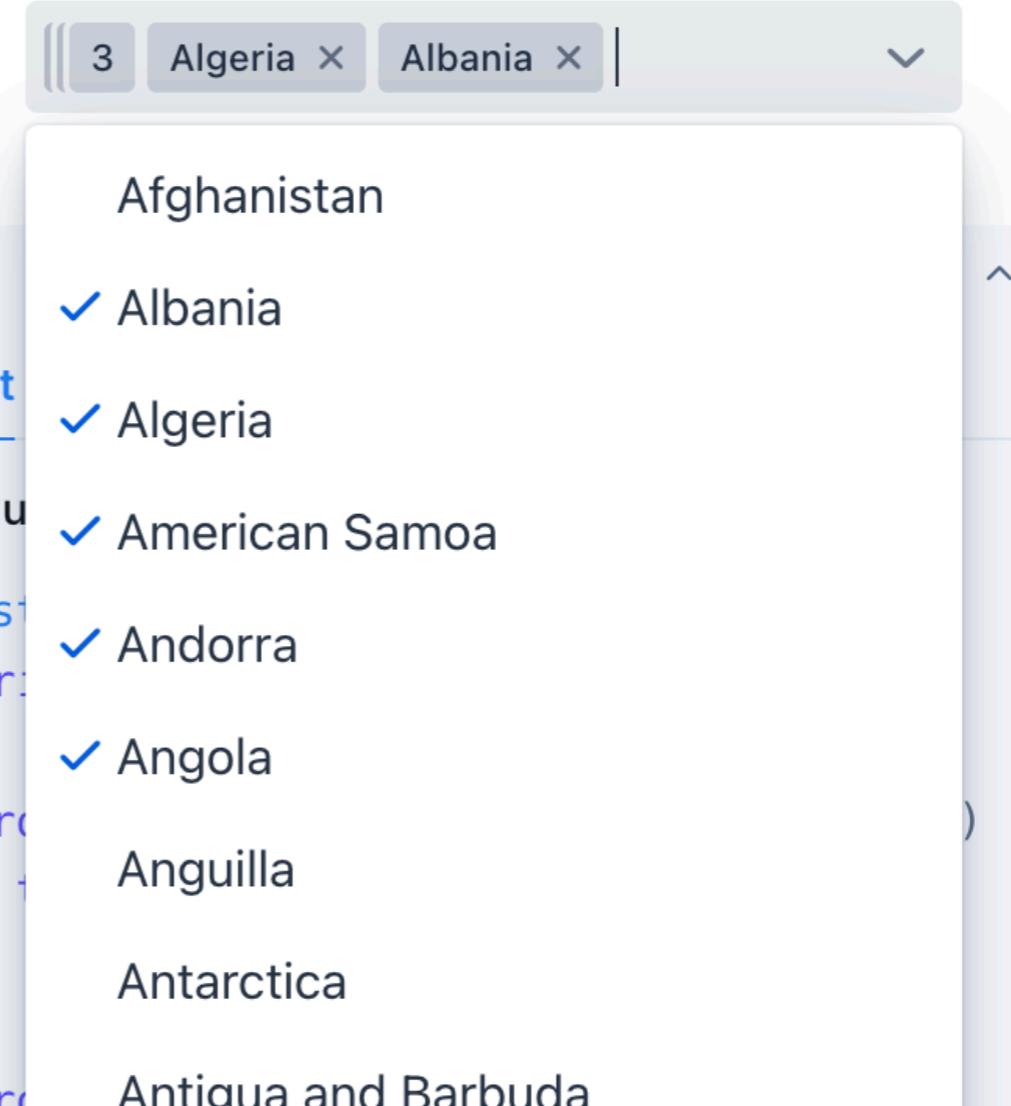
Web Components - Exemple



- Exemple de Web Component
<https://vaadin.com/docs/latest/components/multi-select-combo-box>
- Ici avec la syntaxe de lit-html :

```
<vaadin-multi-select-combo-box
  label="Countries"
  item-label-path="name"
  item-id-path="id"
  .items="${this.items}"
></vaadin-multi-select-combo-box>
```

Countries



The screenshot shows a Vaadin MultiSelectComboBox component with the label "Countries". The input field displays "3" selected items: "Algeria" and "Albania". The dropdown menu lists a total of 14 countries, each preceded by a blue checkmark indicating it is selected. The list includes: Afghanistan, Albania (selected), Algeria (selected), American Samoa, Andorra, Angola, Anguilla, Antarctica, and Antigua and Barbuda.

Country
Afghanistan
✓ Albania
✓ Algeria
✓ American Samoa
✓ Andorra
✓ Angola
Anguilla
Antarctica
Antigua and Barbuda



Web Components - Types de composants

- Dans un application on peut retrouver plusieurs types de composants :
 - des composants d'UI génériques, en général regroupés au sein d'une bibliothèque
Exemples : multi-select, button-group, modal-popup, data-grid...
 - des composants qui représentent des fragments de l'UI de notre application :
Exemples : top-bar, login-form, product-card...
 - des composants qui font office de pages, associés à des URL
Exemple : home-page, products-list, product-details
 - (parfois) un composant qui représente la racine de notre application
Exemple : app-root



formation.tech

Custom Elements

Custom Elements - Introduction



- › Custom Elements est un API Web qui permet de créer de nouvelles balises
- › Il est complètement natif donc pas besoin d'utiliser un API externe comme React, Angular ou Vue
- › On peut créer nos propres éléments (Autonomous custom element) ou étendre des éléments existant (Customized built-in element)
- › Des Lifecycle callbacks sont appelés à moment spécifique de la vie de l'élément (lorsqu'il apparaissent dans le DOM, lorsqu'il reçoivent de nouveaux attributs...)

Custom Elements - Nom de balise



- Le nom de l'élément (i.e. : <my-counter>):
 - doit démarrer par une lettre
 - doit être en minuscule
 - doit contenir au moins un tiret (-) pour rester compatible avec des noms d'éléments HTML, SVG or MathML futurs (qui eux n'en contiennent pas)
 - ne doivent pas être un des noms réservés suivants : annotation-xml, color-profile, font-face, font-face-src, font-face-uri, font-face-format, font-face-name, missing-glyph
 - peuvent contenir n'importe quelle lettre ou emoji (i.e. <math-α> or <emotion-😍> sont valides)

Custom Elements - Autonomous custom element



- Pour créer un Autonomous custom element nous devons créer une classe qui hérite de l'interface HTMLElement
- On doit ensuite enregistrer notre nouveau nom de balise avec un CustomElementRegistry (la variable globale customElements)

```
class Counter extends HTMLElement {}  
customElements.define('my-counter', Counter);
```

Custom Elements - Autonomous custom element



- Un Autonomous custom element peut être utilisé

- Comme les autres balises en HTML :

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

- En utilisant la méthode document.createElement

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



Custom Elements - Customized built-in element

- Un customized build-in element doit hériter d'une HTML*Element interface et la customiser
- L'option extends doit être passé à la méthode register du CustomElementRegistry :

```
class CounterHTMLInputElement extends HTMLButtonElement {}

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



Custom Elements - Customized built-in element

- › Un customized built-in element peut être utilisé
 - En HTML avec l'attribut `is`

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

- En utilisant la méthode `document.createElement` et l'option `is` :

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

Custom Elements - Lifecycle callbacks



- Les Lifecycle callbacks sont des méthodes qui sont appelées automatiquement pendant la vie du 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

- constructeur
 - n'est pas tout à fait un lifecycle callback, il est appelé lorsque la classe est instanciée
 - doit appeler super() avant tout autre instruction
 - n'a pas accès au DOM
 - peut définir des events listeners
 - peut appeler attachInternals ou 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
 - appelé une fois que l'élément est ajouté au DOM
 - aussi appelé lorsque l'élément est déplacé (ajouté ailleurs dans l'arbre)
 - a accès au 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
 - appelé lorsque l'élément est retiré du DOM tree
 - aussi appelé lorsque l'élément est déplacé (ajouté ailleurs dans l'arbre)
 - doit être utilisé pour éviter des problèmes de performance et/ou fuite mémoire

```
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
 - appelé lorsqu'un attribut est défini ou modifié
 - les attributs ainsi surveillés doivent être définis en utilisant la propriété statique observedAttributes

```
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
 - appelé lorsqu'un custom element est adopté par un autre document (i.e. dans un 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

- Pour garder les propriétés et attributs synchronisés on utilise les syntaxes JavaScript get et set :

<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 - Sync 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



- Les champs de formulaire custom doivent être déclarés en utilisant la propriété statique `formAssociated`
- La méthode `attachInternals` doit être appelée par le constructeur ou une ES2022 class property
- `attachInternals` retourne un objet `ElementInternals` qui fournit des utilitaires pour le custom element dans un contexte de formulaire

```
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



- Votre élément peut maintenant être utilisé comme champ de formulaire :

```
<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>
```

- On peut lier internals à l'éléments via la syntaxes get :

```
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 : lorsque le champ est associé au formulaire
- › formDisabledCallback : lorsque le champs ou son fieldset parent est désactivé
- › formResetCallback : lorsque le bouton reset est enfoncé
- › formStateRestoreCallback : lorsque le navigateur rempli le champ (i.e. autocomplete)
- › En savoir plus : <https://web.dev/more-capable-form-controls/>



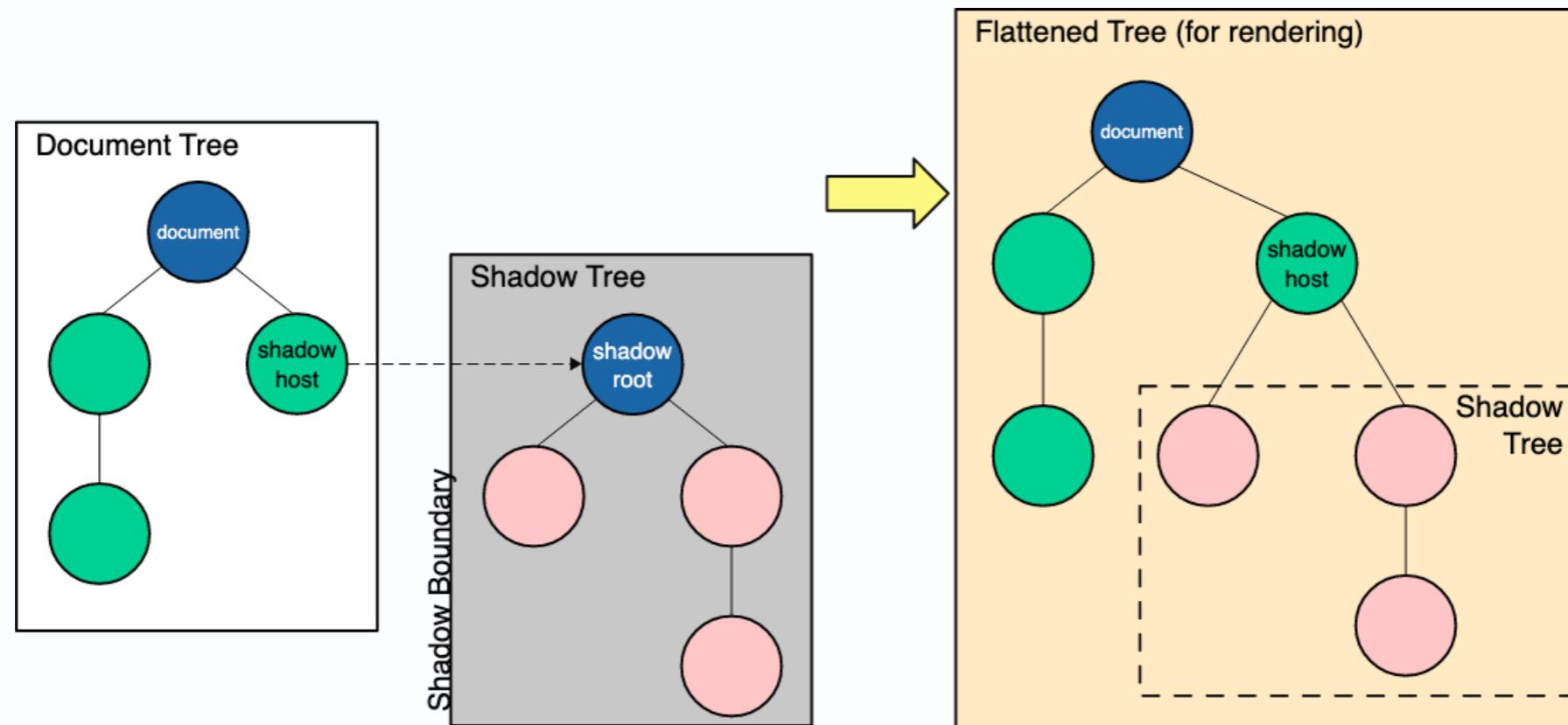
formation.tech

Shadow DOM

Shadow DOM - Introduction



- Shadow DOM est un Web API utilisé pour créer un DOM séparé et masqué du reste de la page à l'intérieur d'un custom element
- Il garde les balises, styles et comportements cachés du reste de la page et réduit ainsi les risques de conflits de HTML, CSS ou JS
- Certains éléments HTML suivent déjà ce principe comme les champs de formulaire ou les éléments audio et video



Shadow DOM - Crédit



- Shadow DOM peut être créé en utilisant la méthode attachShadow d'un élément
- Cette méthode contient une option mode qui peut contenir les valeurs :
 - open : le DOM parent a accès au shadow DOM via la propriété myCustomElem.shadowRoot
 - closed : le shadow DOM n'est pas 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



- Il suffit de créer un élément style ou link à l'intérieur d'un shadow DOM pour que le style ne s'applique qu'à celui-ci :

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

DOM

Shadow DOM - CSS Selectors



- Il y a 4 pseudo-classes dans les sélecteurs CSS liés aux Web Components :
 - :defined qui matches à tous les custom éléments définis avec `customElements.define()`
 - :host, à l'intérieur d'un Shadow DOM, qui fait référence au custom element qui contient le CSS
 - :host(), à l'intérieur d'un Shadow DOM, qui fait référence au custom element qui contient le CSS combiné avec un autre sélecteur passé en paramètre de la fonction
 - :host-context(), à l'intérieur d'un Shadow DOM, qui fait référence au custom element qui contient le CSS combiné avec un autre sélecteur passé en paramètre de la fonction qui s'applique aux ancêtres

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

- Pour personnaliser un Web Component qui utilise le Shadow DOM depuis le document parent il faut utiliser des Custom properties (CSS Variables)
- Les Custom properties sont préfixées par 2 tirets --
- Pour utiliser la Custom property on utilise la fonction var()
`var(--my-custom-property)`
`var(--my-custom-property, default-value)`
- Les Custom properties traversent les Shadow DOM

Shadow DOM - CSS Properties



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



- Utiliser innerHTML pour remplir un element est peu performant avec les web components car le HTML serait parsé à chaque instance du composant
- A la place on privilégie les HTML Templates
- Les templates ne sont pas rendus par le navigateur
- Leur contenu n'est pas rendu qu'une seule fois

```
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

- Les Slots permettent de projeter du contenu à l'intérieur du 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

- Les slots peuvent contenir des valeurs par défaut :

```
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

- On peut utiliser plusieurs slots en utilisant l'attribut name

```
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
Lore ipsum...



formation.tech

Web Component Librairies

Web Component Librairies



- Plusieurs bibliothèques sont dédiés aux Web Components :
 - Lit : créé par Google, successeur de Polymer
 - Stencil : créé par Ionic
 - Catalyst : créé par Github
 - FAST : créé par Microsoft (utilisé par VSCode)
 - Lightning Web Components : créé par SalesForce



Web Component Librairies

npm trends

@github/catalyst vs @microsoft/fast-element vs @stencil/core vs lit

Enter an npm package...

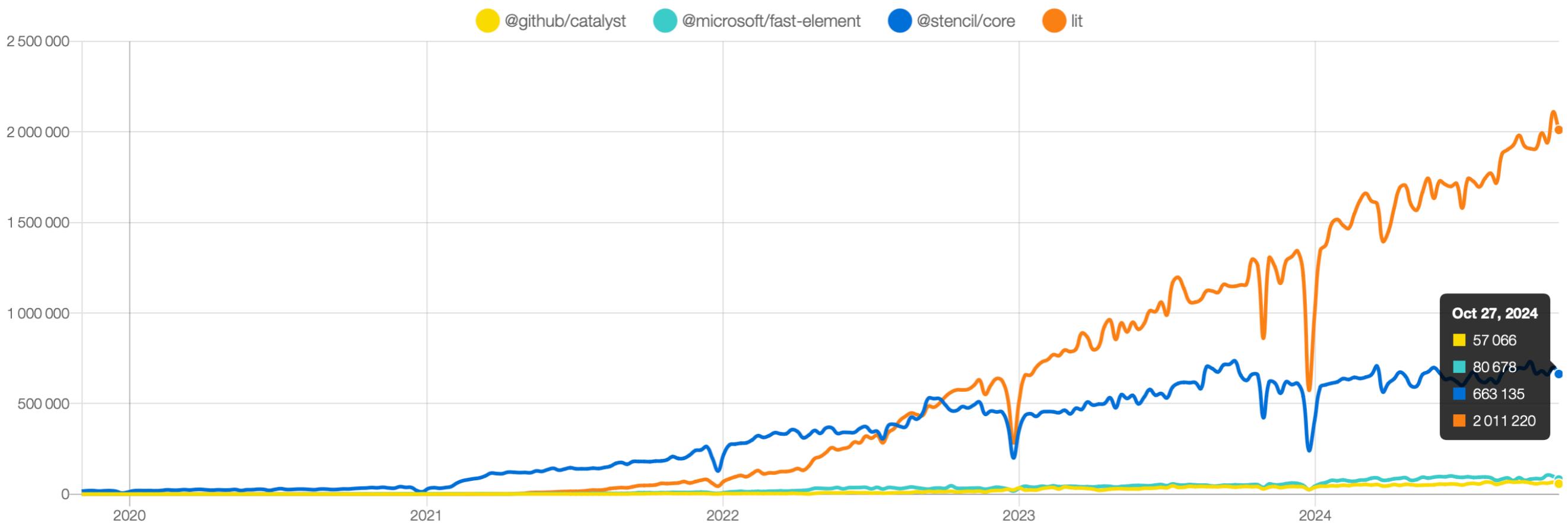
@github/catalyst x

@microsoft/fast-element x

@stencil/core x

lit x

Downloads in past 5 Years ▾





Web Component Librairies



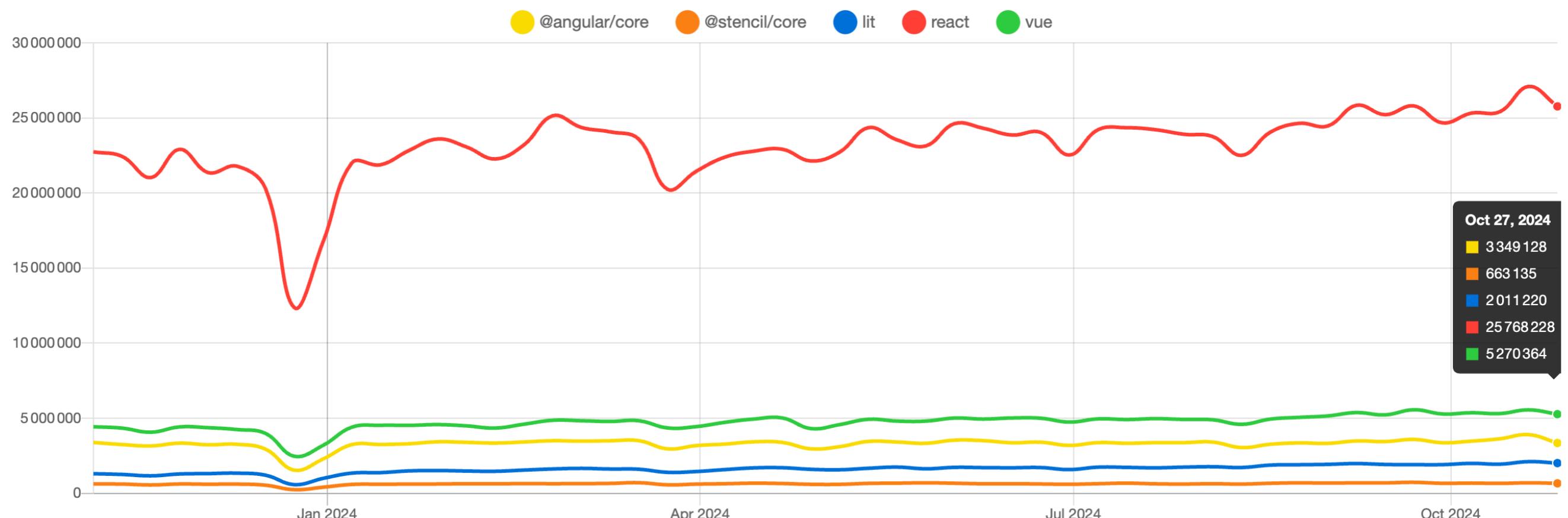
npm trends

@angular/core vs @stencil/core vs lit vs react vs vue

Enter an npm package...

@angular/core ✕ @stencil/core ✕ lit ✕ react ✕ vue ✕

Downloads in past 1 Year ▾





Web Component Librairies

- › On peut également encapsuler des Web Components dans les principaux frameworks frontend :
 - 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>
- › Compatibilité :
<https://custom-elements-everywhere.com/>



Web Component Librairies

- Un IDE en ligne pour les Web Components :
<https://webcomponents.dev/new>
- Un catalogue de Web Components :
<https://www.webcomponents.org/>
- Exemples de Components sans bibliothèques :
<https://github.com/vanillawc>
- Ressources :
<https://web.dev/articles/web-components?hl=en>
https://developer.mozilla.org/en-US/docs/Web/API/Web_components



formation.tech

Lit

Lit - Introduction



- Crée en 2018 comme successeur du projet Polymer né en 2015
- Lit est une bibliothèque qui simplifie la création de Web Components avec de la réactivité, un gestion des templates et du CSS plus efficace
- Lit est idéal pour :
 - créer des UIKit de composants
 - créer des applications comme on pourrait créer des apps React ou Vue.js
 - mettre à jour de façon progressive des sites HTML vers des concepts de développement plus modernes



Lit

Lit - lit-html



- lit-html est un bibliothèque de templates qui permet une mise à jour efficace du DOM
- elle permet d'exprimer l'interface web en fonction des données de façon claire et intuitive comme en HTML
- lit réexporte les fonctions de lit-html
- on retrouve lit-html dans un certain nombre d'autres bibliothèques de composants

Lit - lit-html



- Si on modifiait directement le DOM avec innerHTML, à chaque refresh de notre template, l'ensemble des balises serait recréées :

```
function render(htmlString, element) {
  element.innerHTML = htmlString;
}

const rootEl = document.getElementById('root');

const myTemplate = (name) => `
  <h1>Hello, ${name}</h1>
  <p>Current time: ${new Date().toLocaleString()}</p>
`;

render(myTemplate('World'), rootEl);

setInterval(() => {
  render(myTemplate('World'), rootEl);
}, 1000);
```

Hello, World!

Current time: 30/10/2024 13:07:07

The screenshot shows the browser's developer tools with the "Elements" tab selected. The DOM tree is displayed, starting with the root element `<!DOCTYPE html>`. It includes the `<html lang="en">`, `<head>`, and `<body>` sections. The `<div id="root">` element is highlighted with a blue selection bar. Inside this div, there is an `<h1>` element containing the text "Hello, World!" and a `<p>` element containing the current time "30/10/2024 13:07:07". The entire `<body>` section is closed by a matching `</body>` tag, which is further enclosed by the `<html>` and `</html>` tags.

Lit - lit-html



- Avec le tagged template html et la fonction render de lit-html seuls les noeuds qui contiennent des modifications sont recréés

```
import { html, render } from 'lit-html';

const rootEl = document.getElementById('root');

const myTemplate = (name) => html`<div>
  <h1>Hello, ${name}</h1>
  <p>Current time: ${new Date().toLocaleString()}</p>
</div>`;

render(myTemplate('World'), rootEl);

setInterval(() => {
  render(myTemplate('World'), rootEl);
}, 1000);
```

Hello, World!

Current time: 30/10/2024 13:08:18

The screenshot shows the browser's developer tools with the "Elements" tab selected. The DOM tree is displayed, starting with the root element `<!DOCTYPE html>`. Inside the `<html>` element, there is a `<body>` element containing a `<div id="root">`. This `<div>` has two children: a `<h1>` element and a `<p>` element. The `<h1>` element contains the text "Hello, " followed by a comment `<!--?lit533364337-->`, the word "World", and an exclamation mark "!". The `<p>` element contains the text "Current time: " followed by another comment `<!--?lit533364337-->`, the date "30/10/2024 13:08:18", and a closing tag `</p>`.

Lit - LitElement



- Nos composants vont hériter de LitElement (intègre lit-html) qui lui même hérite de ReactiveElement (synchronise les propriétés avec l'UI)

```
import { LitElement, html } from 'lit';

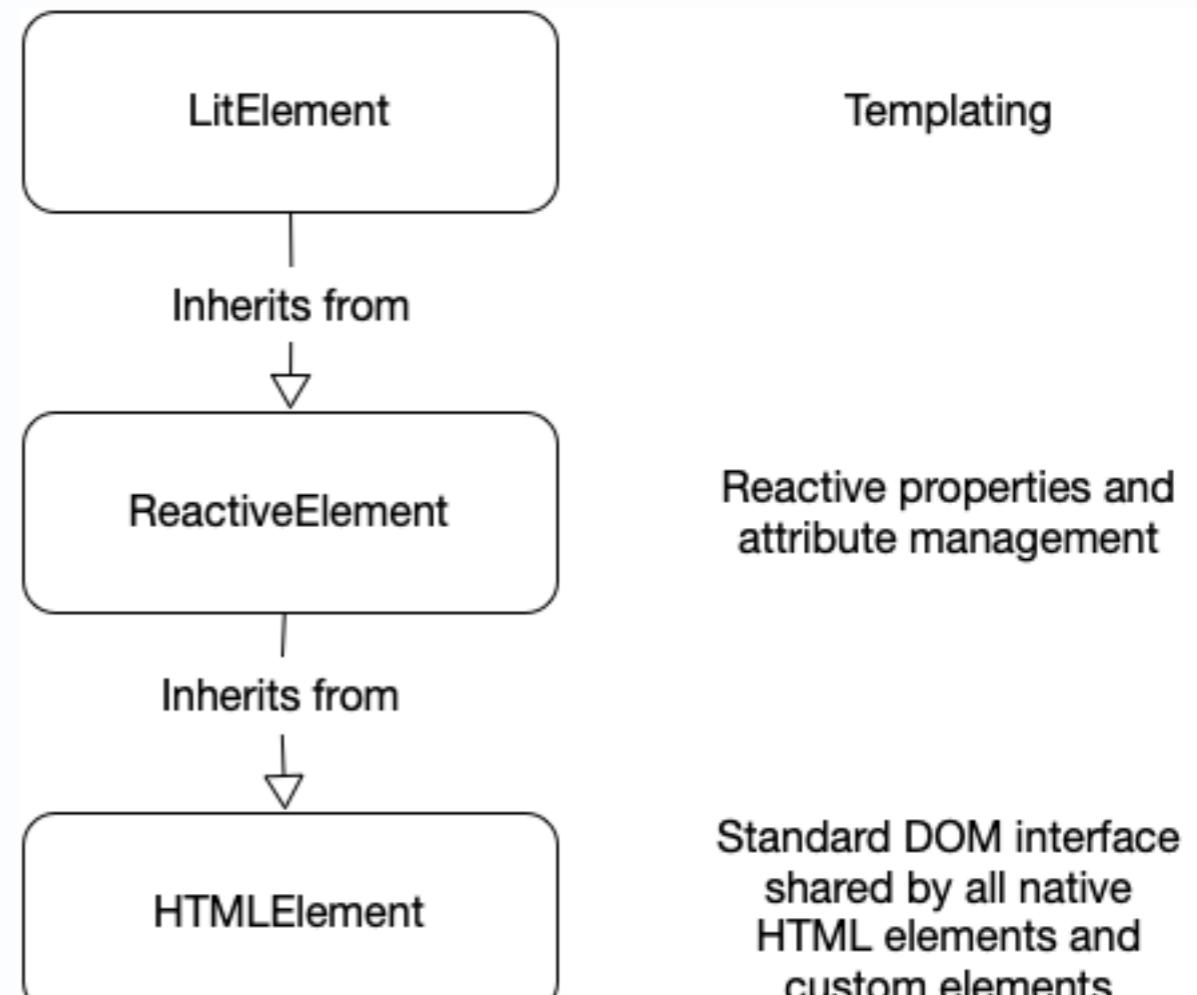
export class MyHello extends LitElement {
  render() {
    return html`<h1>Hello, World!</h1>`;
  }
}

customElements.define('my-hello', MyHello);

import { ReactiveElement } from 'lit';

export class MyHello extends ReactiveElement {
  createRenderRoot() {
    const shadowRoot = super.createRenderRoot();
    const h1El = document.createElement('h1');
    h1El.innerText = 'Hello, World!';
    shadowRoot.append(h1El);
    return shadowRoot;
  }
}

customElements.define('my-hello', MyHello);
```



Lit - LitElement



- Un composant Lit va définir son template si besoin via la méthode render
- Il faut penser à le définir via customElements.define

```
import { LitElement, html } from 'lit';

export class MyHello extends LitElement {
  render() {
    return html`<h1>Hello, World!</h1>`;
  }
}

customElements.define('my-hello', MyHello);
```

Lit - LitElement



- La méthode render peut retourner :
 - des valeurs primitives comme string, number ou boolean
 - des objets TemplateResult créés par la fonction html
 - des noeuds DOM
 - les valeurs spéciales nothing ou noChange (exportées par lit, à privilégier plutôt que empty string, null ou undefined)
 - des tableaux ou iterables de valeurs mentionnées ci-dessus

Lit - Composition



- › Pour décomposer une UI en des fragments plus petits on utilise la composition
- › Soit en imbriquant des appels à html :

```
import { html, LitElement } from 'lit';

export class MyList extends LitElement {
  renderItem(index) {
    return html`<li>Item ${index}</li>`;
  }

  render() {
    return html`<ul>
      ${this.renderItem(1)}
      ${this.renderItem(2)}
      ${this.renderItem(3)}
    </ul>`;
  }
}

customElements.define('my-list', MyList);
```

Lit - Composition



- Soit en faisant appel à d'autres composants :

```
import { html, LitElement } from 'lit';
import './my-item.js';

export class MyList extends LitElement {
  render() {
    return html`<div>
      <my-item index="1"></my-item>
      <my-item index="2"></my-item>
      <my-item index="3"></my-item>
    </div>`;
  }
}

customElements.define('my-list', MyList);
```

Lit - Properties



- Les propriétés du composant sont définies via la propriété statique properties :

```
import { html, LitElement } from 'lit';

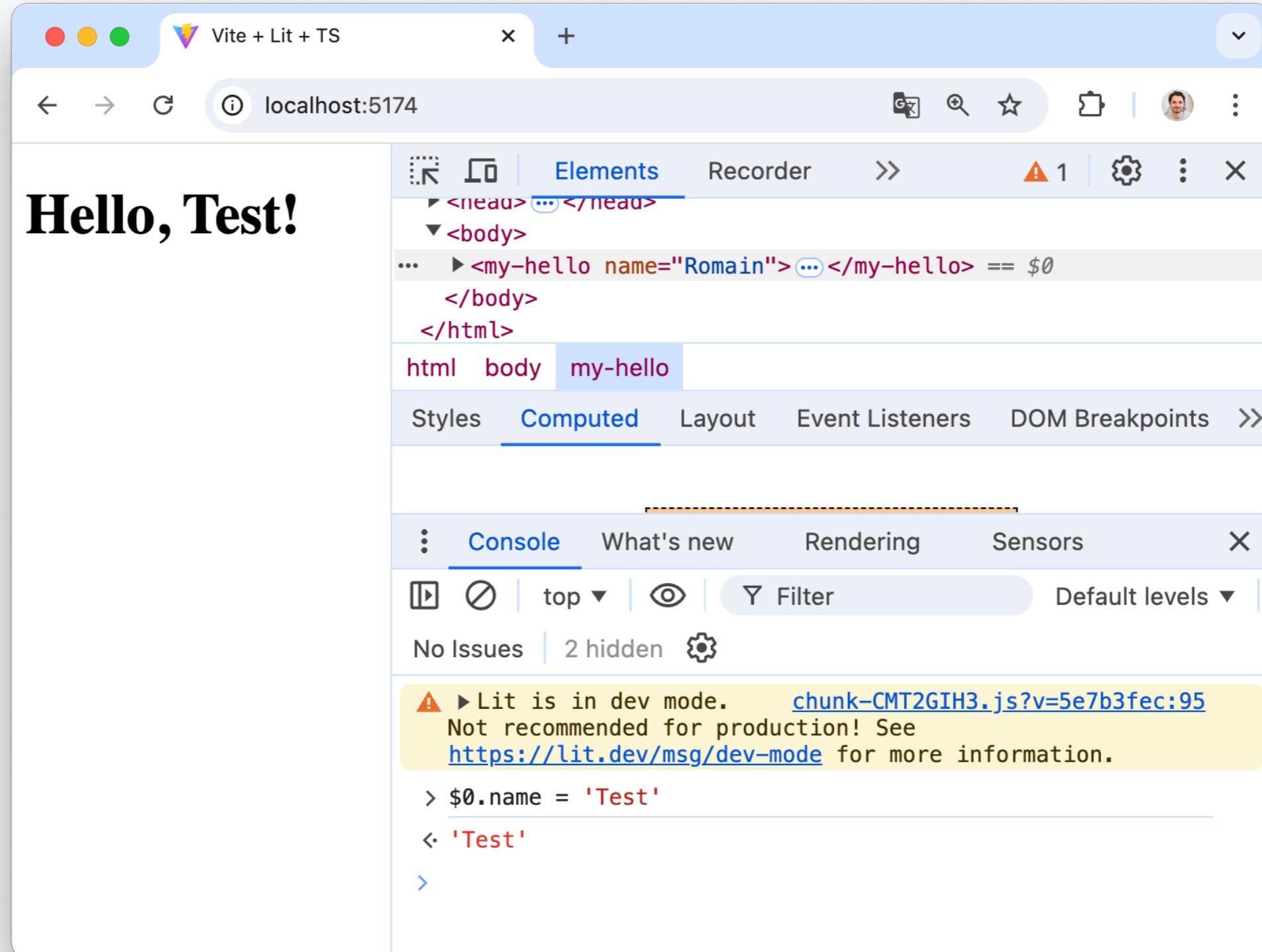
export class MyHello extends LitElement {
  static properties = {
    name: { type: String },
  };

  render() {
    return html`<h1>Hello, ${this.name}!</h1>`;
  }
}

customElements.define('my-hello', MyHello);
```

- Le composant est "réactif" aux changements de propriétés, si la propriété change, l'UI se rafraîchit en conséquence

Lit - Properties



The screenshot shows a browser window with the title "Vite + Lit + TS" at "localhost:5174". The main content area displays the text "Hello, Test!". The browser's developer tools are open, specifically the Elements and Console tabs.

Elements Tab: Shows the DOM structure:

- <head> (highlighted)
- <body>
- ... > <my-hello name="Romain"> ... </my-hello> == \$0
- </body>
- </html>

The "my-hello" element is selected in the tree. The bottom navigation bar for the Elements tab includes "Styles", "Computed" (which is selected), "Layout", "Event Listeners", and "DOM Breakpoints".

Console Tab: Shows the following output:

- A warning message: "⚠ Lit is in dev mode. [chunk-CMT2GIH3.js?v=5e7b3fec:95](#)
Not recommended for production! See <https://lit.dev/msg/dev-mode> for more information."
- Log entries:
 - > \$0.name = 'Test'
 - < 'Test'
 - >

Lit - Properties



- En JavaScript pour améliorer la complétion on peut typer avec JSDoc

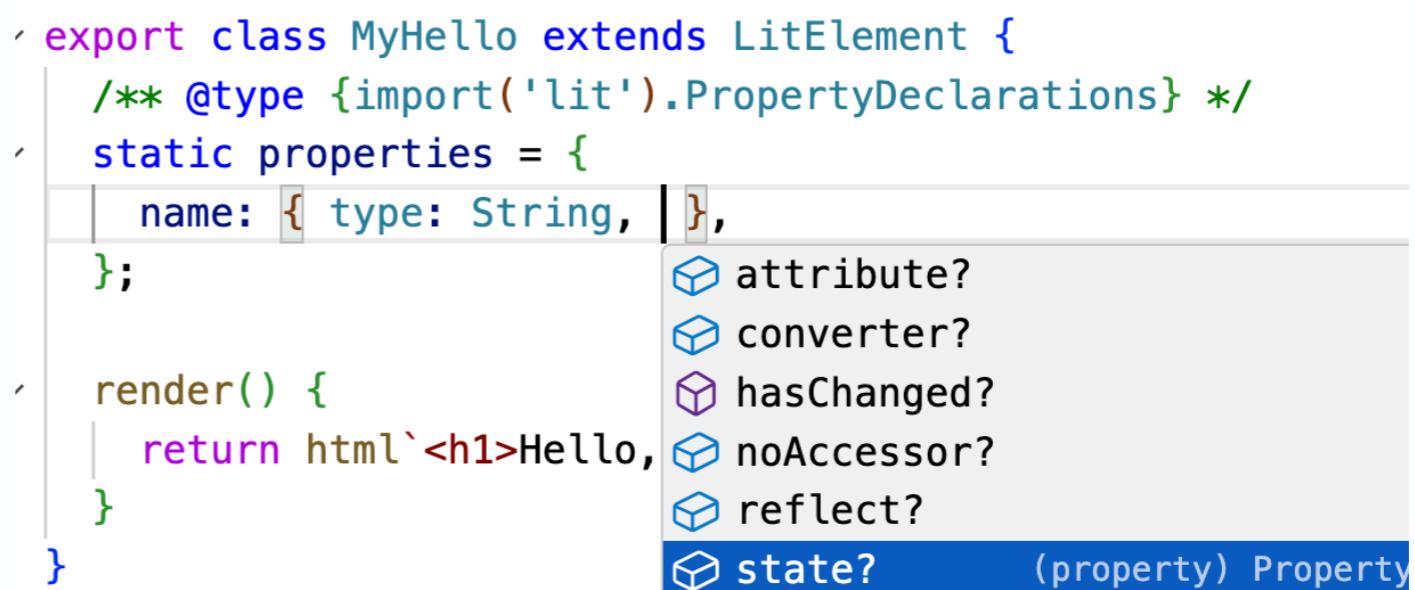
```
import { html, LitElement } from 'lit';

export class MyHello extends LitElement {
  /** @type {import('lit').PropertyDeclarations} */
  static properties = {
    name: { type: String },
  };

  render() {
    return html`<h1>Hello, ${this.name}</h1>`;
  }
}

customElements.define('my-hello', MyHello);
```

```
  export class MyHello extends LitElement {
    /** @type {import('lit').PropertyDeclarations} */
    static properties = {
      name: { type: String, | },
```



The image shows an IDE's code completion tooltip for the 'name' property declaration. It lists several optional properties: 'attribute?', 'converter?', 'hasChanged?', 'noAccessor?', 'reflect?', and 'state?'. The 'state?' option is highlighted with a blue background.

```
      };
      render() {
        return html`<h1>Hello,
      }
    }
```

Lit - Properties



- Pour donner une valeur par défaut on utilise un constructeur
- Attention cela ne fonctionne pas avec des ES2022 Class Properties car elles écrasent les accesseurs get/set qu'utilise LitElement pour détecter les changements de valeur

```
import { LitElement, html } from 'lit';

export class MyHello extends LitElement {
  /** @type {import('lit').PropertyDeclarations} */
  static properties = {
    name: { type: String },
  };

  constructor() {
    super();
    this.name = 'world'; // ✅ Initial value
  }

  render() {
    return html`<h1>Hello, ${this.name}</h1>`;
  }
}

customElements.define('my-hello', MyHello);
```

```
import { LitElement, html } from 'lit';

export class MyHello extends LitElement {
  /** @type {import('lit').PropertyDeclarations} */
  static properties = {
    name: { type: String },
  };

  name = 'world'; // ❌ Error: The following property
  // element my-hello will not trigger updates as ex
  // because they are set using class fields: name.
  // Native class fields and some compiled output wi
  // overwrite accessors used for detecting changes.
  // https://lit.dev/msg/class-field-shadowing for m

  render() {
    return html`<h1>Hello, ${this.name}</h1>`;
  }
}

customElements.define('my-hello', MyHello);
```

Lit - Properties



- Les propriétés peuvent être déclarées internes avec state: true
- Elle ne font pas alors partie de l'API public du composant
- Par convention on les préfixe par un underscore ou les déclare private en TypeScript
- Les propriétés interne n'ont pas d'attribut correspondant mais peuvent néanmoins être initialisés via le DOM

```
import { html, LitElement } from 'lit';

export class MyCounter extends LitElement {
    /** @type {import('lit').PropertyDeclarations} */
    static properties = {
        _count: { type: Number, state: true },
    };

    constructor() {
        super();
        this._count = 0;
    }

    render() {
        return html`

Count: ${this._count}

`;
    }
}
```

Lit - Properties



- Les types permettent de définir comment passer de l'attribut à la propriété et inversement, ils masquent 2 fonctions de conversions *fromAttribute* et *toAttribute*
- Les types suivants peuvent être utilisés :
 - String (défaut) : propriété null ou undefined supprime l'attribut, sinon la valeur se synchronise dans un sens ou l'autre
 - Number : propriété null ou undefined supprime l'attribut, une conversion Number est appliquée dans le sens attribut -> propriété et String dans l'autre
 - Boolean : si l'attribut existe propriété true, sinon false. Si la propriété est truthy empty attribute, falsy l'attribut est supprimé
 - Array / Object : propriété null ou undefined supprime l'attribut. Sinon conversion via JSON.parse / JSON.stringify



Lit - Properties

- Attention en TypeScript le type indiqué doit exister au runtime (pas d'interface ou de type alias)

```
import { LitElement, html } from 'lit';
import { customElement, property } from 'lit/decorators.js';

interface User {
  name: string;
}

@customElement('my-hello')
export class MyHello extends LitElement {
  @property() // utiliser @property({ type: Object })
  user!: User;

  render() {
    return html`<h1>Hello, ${this.user.name}!</h1>`; // undefined car par défaut type: String
  }
}
```

Lit - Properties



- Autres options :
 - attribute : par défaut true, si false la propriété existera mais pas l'attribut (donc pas possible de passer une valeur en HTML)
 - converter : permet de définir les fonctions pour passer de propriété à attribut et inversement
<https://lit.dev/docs/components/properties/#conversion-converter>
 - hasChanged : une fonction qui permet de définir sur quel critère la propriété a changé, par défaut newValue !== oldValue. Pourrait être un *deepEqual* sur des tableaux ou des objects
 - noAccessor : rare, pour des héritages, la classe fille pourrait définir une propriété qui ne modifie les options de la propriété sans modifier celle de la classe parent
 - reflect : synchronise l'attribut avec la propriété

Lit - Binding



- Binding sur des noeuds enfants

```
html`  
<h1>Hello ${name}</h1>  
<ul>  
  ${listItems}  
</ul>`
```

- Binding sur des attributs

```
html`<div class=${highlightClass}></div>`
```

- Binding sur des attributs booléens

```
html`<div ?hidden=${!show}></div>`
```

- Binding sur des propriété

```
html`<input .value=${value}>`
```

Lit - Conditional Rendering



- Il existe différentes techniques pour masquer un élément dans un template lit
 - la propriété hidden
<https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/hidden>
 - l'attribut style et les propriétés CSS display, opacity ou visibility
https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/style
 - l'attribut class
https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/class
 - l'opérateur AND (&&)
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND
 - l'opérateur ternaire (?:)
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_operator
 - l'opérateur OR (||) ou Nullish coalescing (??)
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_OR
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing

Lit - Conditional Rendering



- Avec la propriété hidden

```
import { LitElement, html } from 'lit';

export class SelectComponent extends LitElement {
  static properties = {
    _menuOpen: { type: Boolean, state: true },
  };

  constructor() {
    super();
    this._menuOpen = false;
  }

  render() {
    return html`

Item 2


      <div class="menu" .hidden=${!this._menuOpen}>
        <div class="menu-item">Item 1</div>
        <div class="menu-item">Item 2</div>
        <div class="menu-item">Item 3</div>
      </div>
    `;
  }
}

customElements.define('my-select', SelectComponent);
```

Lit - Conditional Rendering



- Avec l'attribut style et la directive styleMap

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

export class SelectComponent extends LitElement {
  static properties = {
    _menuOpen: { type: Boolean, state: true },
  };

  constructor() {
    super();
    this._menuOpen = false;
  }

  render() {
    return html`

Item 2



<div class="menu-item">Item 1</div>
  <div class="menu-item">Item 2</div>
  <div class="menu-item">Item 3</div>

`;
  }
}

customElements.define('my-select', SelectComponent);
```

Lit - Conditional Rendering



- Avec l'attribut class et la directive classMap

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

export class SelectComponent extends LitElement {
  static properties = {
    _menuOpen: { type: Boolean, state: true },
  };

  static styles = css`  

    .close {  

      display: none;  

    }  

`;  

  constructor() {  

    super();  

    this._menuOpen = false;
  }
  

  render() {  

    return html`  

      <div class="selected">Item 2</div>  

      <div class="${classMap({ menu: true, close: !this._menuOpen })}">  

        <div class="menu-item">Item 1</div>  

        <div class="menu-item">Item 2</div>  

        <div class="menu-item">Item 3</div>  

      </div>
`;  

  }
}
```

Lit - Conditional Rendering



- Avec l'opérateur ternaire (nothing est un valeur spéciale lit-html qui ne crée pas de noeud dans le DOM)

```
import { LitElement, html, nothing } from 'lit';

export class SelectComponent extends LitElement {
  static properties = {
    _menuOpen: { type: Boolean, state: true },
  };

  constructor() {
    super();
    this._menuOpen = false;
  }

  render() {
    return html`

Item 2


${this._menuOpen
  ? html`<div class="menu">
    <div class="menu-item">Item 1</div>
    <div class="menu-item">Item 2</div>
    <div class="menu-item">Item 3</div>
  </div>`
  : nothing}
`;
  }
}

customElements.define('my-select', SelectComponent);
```

Lit - Conditional Rendering



- Avec un if/else

```
import { LitElement, html, nothing } from 'lit';

export class SelectComponent extends LitElement {
  static properties = {
    _menuOpen: { type: Boolean, state: true },
  };

  constructor() {
    super();
    this._menuOpen = false;
  }

  render() {
    let menu = nothing;

    if (this._menuOpen) {
      menu = html`

<div class="menu-item">Item 1</div>
        <div class="menu-item">Item 2</div>
        <div class="menu-item">Item 3</div>
      </div>
    `;
    }

    return html`

Item 2</div>
    ${menu}
  `;
  }
}


```

Lit - Conditional Rendering



- On peut améliorer les performances en utilisant la directive cache :

```
import { LitElement, html, nothing } from 'lit';
import { cache } from 'lit/directives/cache.js';

export class SelectComponent extends LitElement {
  static properties = {
    _menuOpen: { type: Boolean, state: true },
  };

  constructor() {
    super();
    this._menuOpen = false;
  }

  render() {
    return html`

Item 2


      ${cache(this._menuOpen
        ? html`<div class="menu">
          <div class="menu-item">Item 1</div>
          <div class="menu-item">Item 2</div>
          <div class="menu-item">Item 3</div>
        </div>
      : nothing)}
    `;
  }
}

customElements.define('my-select', SelectComponent);
```

Lit - List



- Lit sait déjà traiter les tableaux ou autre type iterable, ici un tableau de string sera rendu en tableau de `TextNode` :

```
import { LitElement, html } from 'lit';

export class SelectComponent extends LitElement {
  static properties = {
    items: { type: Array },
  };

  constructor() {
    super();
    this.items = ['Item 1', 'Item 2', 'Item 3'];
  }

  render() {
    return html`

Item 2


      <div class="menu">${this.items}</div>
    `;
  }
}

customElements.define('my-select', SelectComponent);
```

Lit - List



- Ce tableau peut contenir des templates créés avec le tagged template html :

```
import { LitElement, html } from 'lit';

export class SelectComponent extends LitElement {
  static properties = {
    items: { type: Array },
  };

  constructor() {
    super();
    this.items = [html`<div>Item 1</div>`, html`<div>Item 2</div>`, html`<div>Item 3</div>`];
  }

  render() {
    return html`
      <div class="selected">Item 2</div>
      <div class="menu">${this.items}</div>
    `;
  }
}

customElements.define('my-select', SelectComponent);
```

Lit - List



- On peut ainsi transformer un tableau de string ou d'objet en tableau de template html, avec une boucle for :

```
import { LitElement, html } from 'lit';

export class SelectComponent extends LitElement {
  static properties = {
    items: { type: Array },
  };

  constructor() {
    super();
    this.items = ['Item 1', 'Item 2', 'Item 3'];
  }

  render() {
    const listItems = [];

    for (const item of this.items) {
      listItems.push(html`<div class="item">${item}</div>`);
    }

    return html`
      <div class="selected">Item 2</div>
      <div class="menu">
        ${listItems}
      </div>
    `;
  }
}

customElements.define('my-select', SelectComponent);
```

Lit - List



- Ou plus directement avec la méthode map du type Array :

```
import { LitElement, html } from 'lit';

export class SelectComponent extends LitElement {
  static properties = {
    items: { type: Array },
  };

  constructor() {
    super();
    this.items = ['Item 1', 'Item 2', 'Item 3'];
  }

  render() {
    return html`

Item 2


      <div class="menu">
        ${this.items.map((item) => html`<div class="item">${item}</div>`)}
      </div>
    `;
  }
}

customElements.define('my-select', SelectComponent);
```



- Pour des lists qui risque d'être modifiée fréquemment, par exemple triées, on préfèrera la directive repeat pour des soucis de performances (où il est possible de spécifier un lien entre le modèle et la vue via une key function) :

```
import { LitElement, html } from 'lit';
import { repeat } from 'lit/directives/repeat.js';

export class SelectComponent extends LitElement {
  static properties = {
    items: { type: Array },
  };

  constructor() {
    super();
    this.items = [
      { id: 1, name: 'Item 1' },
      { id: 2, name: 'Item 2' },
      { id: 3, name: 'Item 3' },
    ];
  }

  render() {
    return html`

Item 2


    <div class="menu">
      ${repeat(
        this.items,
        (item) => item.id,
        (item) => html`<div class="item">${item.name}</div>`,
      )}
    </div>
  `;
  }
}
```

Lit - Directives



- En plus des directives que nous avons déjà vues :
 - classMap
 - styleMap
 - cache
 - repeat
- Lit propose également d'autres directives à consulter ici :
<https://lit.dev/docs/templates/directives/>
- Si besoin on peut également définir ses propres directives :
 - sous forme de fonction pour filtrer une valeur
 - sous forme de classe pour accéder au DOM
- <https://lit.dev/docs/templates/custom-directives/>

Lit - Events



- Pour écouter un événement on utilise la syntaxe @

```
openMenu() {
  this._menuOpen = !this._menuOpen;
}

render() {
  return html`

81


```

Lit - Events



- La fonction ainsi associée à accès à l'objet event

```
import { LitElement } from "lit";

export class MyElement extends LitElement {
  static properties = {
    name: { type: String, state: true },
  };

  constructor() {
    super();
    this.name = 'Titi';
  }

  updateName(event) {
    this.name = event.target.value;
  }

  render() {
    return html`
```

Lit - Events



- Parfois on souhaiter passer une autre valeur à la fonction, il sera nécessaire de définir une fonction supplémentaire pour ignorer event

```
valueSelected(item) {
  this.item = item;
  this._menuOpen = false;
}

render() {
  return html`

83


```

Lit - Custom Events



- › Pour remonter des valeurs aux ancêtres on utilise des CustomEvent

```
valueSelected(item) {  
    this.item = item;  
    this._menuOpen = false;  
    this.renderRoot.dispatchEvent(  
        new CustomEvent('value-selected', {  
            detail: item,  
            bubbles: true,  
            composed: true,  
        }),  
    );  
}
```

- › bubbles permet d'écouter au niveau des ancêtres (et pas uniquement au niveau du composant qui dispatch)
- › composed permet de traverser les shadowRoot



Lit - Custom Events

- Au niveau du parent on récupère la valeur avec event.detail

```
updateName(event) {  
    this.name = event.detail;  
}  
  
render() {  
    return html`  
        <my-select  
            .item=${this.name}  
            .items=${['Toto', 'Titi', 'Tata']}            @value-selected=${this.updateName}  
        ></my-select>  
        <p>You selected : ${this.name}</p>  
    `;  
}
```

Lit - Communication entre les composants



- Parent → Enfant
Reactive properties
- Enfant → Ancêtre
CustomEvent
- Composants éloignés
Context ou Controller custom



- Pour sélectionner un élément du composant, le plus simple est d'utiliser les ref

```
class MyElement extends LitElement {  
  
  inputRef = createRef();  
  
  render() {  
    // Passing ref directive a Ref object that will hold the element in .value  
    return html`<input ${ref(this.inputRef)}>`;  
  }  
  
  firstUpdated() {  
    const input = this.inputRef.value;  
    input.focus();  
  }  
}  
customElements.define('my-element', MyElement);
```

- On peut également utiliser querySelector sur le noeud renderRoot

```
class MyElement extends LitElement {  
  render() {  
    // Passing ref directive a Ref object that will hold the element in .value  
    return html`<input>`;  
  }  
  
  firstUpdated() {  
    const input = this.renderRoot.querySelector('input');  
    input.focus();  
  }  
}  
customElements.define('my-element', MyElement);
```

Lit - Cycle de vie



- En plus du cycle de vie habituel, Lit ajoute les méthodes suivantes :

- `hasChanged()`

A chaque fois une propriété réactive est modifiée

- `requestUpdate()`

Demande la mise à jour du composant

- `shouldUpdate(changedProperties)`

Si la fonction retourne true, le composant sera mis à jour

```
shouldUpdate(changedProperties) {  
    // Only update element if prop1 changed.  
    return changedProperties.has('prop1');  
}
```

- `willUpdate()`

Appelée juste avant l'update, permet de configurer des valeurs dérivées des propriétés modifiées nécessaire à l'update

Lit - Cycle de vie



- `firstUpdated()`
La première fois que le composant est rafraîchi (pour le `.focus()` par exemple)
- `updated()`
A chaque fois que le composant est rafraîchi
- `updateComplete`
Promise qui permet d'attendre que le composant ait été rafraîchi (pour les `expect` des tests unitaires par exemple)

Lit - Controllers



- Un controller est une classe dont l'instance va être associé au composant Lit et qui pourra demander son rafraîchissement
- Ils permettent d'alléger le code du composant
- On les utilise généralement pour
 - manipuler les events globaux (`window.addEventListener`)
 - les calls HTTP (`fetch`, `XHR`)
 - les animations

Lit - Controllers



- Exemple de contrôleur

```
export class ClockController {  
    host;  
  
    value = new Date();  
    timeout;  
    _timerID;  
  
    constructor(host, timeout = 1000) {  
        (this.host = host).addController(this);  
        this.timeout = timeout;  
    }  
    hostConnected() {  
        // Start a timer when the host is connected  
        this._timerID = setInterval(() => {  
            this.value = new Date();  
            // Update the host with new value  
            this.host.requestUpdate();  
        }, this.timeout);  
    }  
    hostDisconnected() {  
        // Clear the timer when the host is disconnected  
        clearInterval(this._timerID);  
        this._timerID = undefined;  
    }  
}
```

Lit - Controllers



- › Pour l'enregistrer

```
import { LitElement, html } from 'lit';
import { ClockController } from './clock-controller.js';

class MyElement extends LitElement {
  // Create the controller and store it
  clock = new ClockController(this, 100);

  // Use the controller in render()
  render() {
    const formattedTime = timeFormat.format(this.clock.value);
    return html`Current time: ${formattedTime}`;
  }
}
customElements.define('my-element', MyElement);

const timeFormat = new Intl.DateTimeFormat('en-US', {
  hour: 'numeric',
  minute: 'numeric',
  second: 'numeric',
});
```

Lit - Async Task



- Pour les opérations qui utilisent les promesses, Lit propose un contrôleur dans le paquet @lit/task.



Lit - Async Task

- Exemple

```
import { Task } from '@lit/task';

class MyElement extends LitElement {
  static properties = {
    productId: {},
  };

  _productTask = new Task(this, {
    task: async ([productId], { signal }) => {
      const response = await fetch(`http://example.com/product/${productId}`, {
        signal,
      });
      if (!response.ok) {
        throw new Error(response.status);
      }
      return response.json();
    },
    args: () => [this.productId],
  });

  render() {
    return this._productTask.render({
      pending: () => html`<p>Loading product...</p>`,
      complete: (product) => html`
        <h1>${product.name}</h1>
        <p>${product.price}</p>
        `,
      error: (e) => html`<p>Error: ${e}</p>`,
    });
  }
}
```

Lit - Decorators



- Lit propose de simplifier un certain nombre de ses concepts en utilisant des décorateurs :
 - customElement pour définir le custom element
 - eventOptions pour ajouter des options à un event listener d'un template (once, passive, capture...)
 - property pour définir ses propriétés reactive
 - query et ses dérivés pour sélectionner un element du template
 - state pour définir des propriétés reactive privées

Lit - Decorators



- Exemple avec décorateurs :

```
import { html, LitElement } from 'lit';
import { customElement, property, query } from 'lit/decorators.js';

@customElement('my-hello')
export class MyHello extends LitElement {
  @property()
  name = 'World';

  @query('h1')
  h1El!: HTMLHeadingElement;

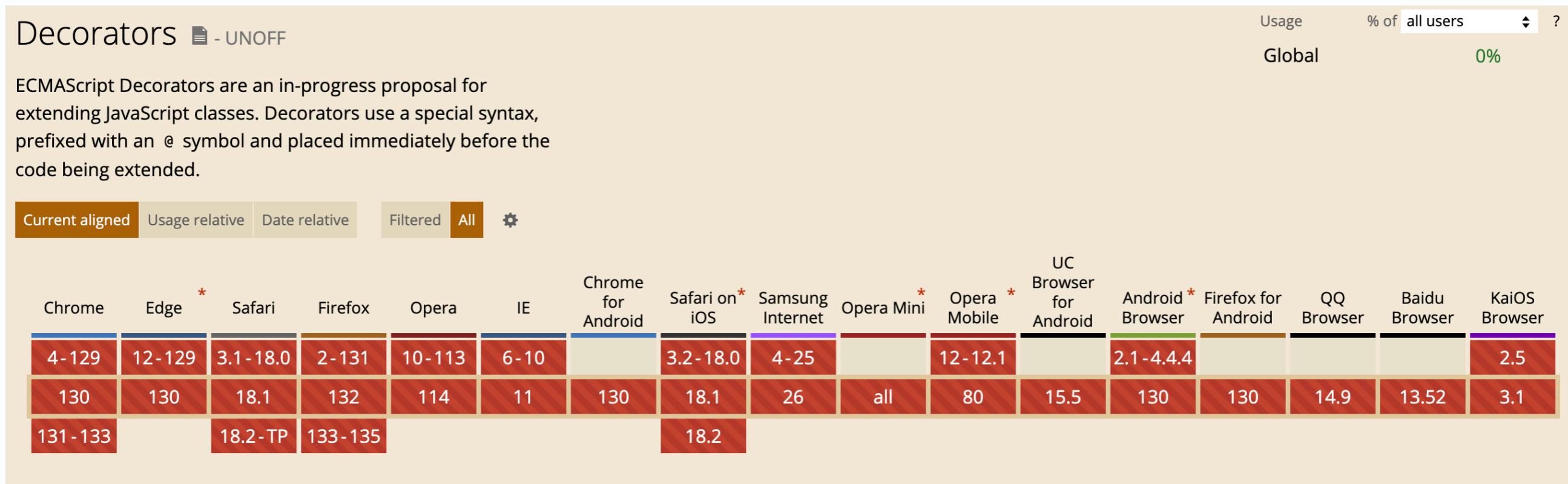
  updated() {
    this.h1El.addEventListener('click', () => {
      console.log('clicked');
    });
  }

  render() {
    return html`<h1>Hello, ${this.name}!</h1>`;
  }
}
```

Lit - Decorators



- A ce jour (novembre 2024) la norme sur les décorateurs n'est pas encore validée
- Elle est au stage 3 du TC39 (release candidate)
- Aucun navigateur ne la supporte pour l'instant
- Il faudra passer par un compilateur comme Babel ou TypeScript
- Les docs Lit en TypeScript utilisent les décorateurs



Lit - HTMLElementTagNameMap



- En TypeScript il est également important de définir le composant dans l'interface HTMLElementTagNameMap afin de le typer

```
declare global {
  interface HTMLElementTagNameMap {
    "my-hello": MyHello;
  }
}
```



formation.tech

Composants Lit @Rolex

Injection de Dépendance - Principes SOLID



- SOLID signifie :
 - **S**ingle responsibility principle
 - **O**pen–closed principle
 - **L**iskov substitution principle
 - **I**nterface segregation principle
 - **D**evelopment dependency inversion principle

Injection de Dépendance - Principes SOLID



- Single responsibility principle

Chaque module, classe ou fonction d'un programme informatique doit être responsable d'une seule partie de la fonctionnalité de ce programme et doit encapsuler cette partie.

Injection de Dépendance - Principes SOLID



- Open-closed principle

Les entités logicielles (classes, modules, fonctions, etc.) devraient être ouvertes à l'extension, mais fermées à la modification.

Injection de Dépendance - Principes SOLID



- › Liskov substitution principle

Si S est un sous-type de T , les objets de type T d'un programme peuvent être remplacés par des objets de type S sans altérer aucune des propriétés souhaitables de ce programme.

Injection de Dépendance - Principes SOLID



- Interface segregation principle

Plusieurs interfaces spécifiques au client valent mieux qu'une seule interface générale

Injection de Dépendance - Principes SOLID



- › Dependency inversion principle

Dépendre d'abstractions (interfaces, classes abstraites...) et non de concrétisations (classes)

Injection de Dépendance - Mauvais Exemple



```
export class CoffeeCup {  
  constructor(  
    private coffeeType: "arabica" | "robusta",  
    private cupCapacity: number  
  ) {}  
}
```

```
import { CoffeeCup } from "./CoffeeCup";  
  
const coffeeCup = new CoffeeCup("arabica", 20);
```



- ✗ Single Responsibility : le café n'est pas séparé de la tasse
- ✗ Open-closed principle : pour créer une tasse de thé, nous devrions modifier le code
- ✗ Dependency inversion principle : CoffeeCup n'a pas de dépendance

Injection de Dépendance - Mauvais Exemple



```
import fs from "fs/promises";

export class FileLogger {
  constructor(private filePath: string) {}

  async log(msg: string) {
    const formatted = `${new Date().toISOString()} - ${msg}\n`;
    await fs.appendFile(this.filePath, formatted);
  }
}
```

```
import { FileLogger } from "./FileLogger";

const logger = new FileLogger("app.log");
await logger.log('Message');
```

- ✗ Single Responsibility : l'écriture de fichiers n'est pas séparée du logging
- ✗ Open-closed principle : pour loguer dans le terminal, il faudrait modifier le code
- ✗ Dependency inversion principle : FileLogger n'a pas de dépendance

Injection de Dépendance - Mauvais Exemple



```
export class Coffee {  
  constructor(  
    private type: "arabica" | "robusta",  
  ) {}  
}
```

```
import { Coffee } from "./Coffee";  
  
export class Cup {  
  private coffee: Coffee;  
  constructor(  
    private capacity: number  
  ) {  
    this.coffee = new Coffee('arabica');  
  }  
}
```

```
import { Cup } from "./Cup";  
  
const coffeeCup = new Cup(10);
```



- ✓ Single Responsibility : 1 classe pour le café, 1 classe pour la tasse
- ✗ Open-closed principle : pour créer une tasse de thé, il faudrait modifier le code
- ✗ Dependency inversion principle : Coffee est une dépendance codée en dur, il ne peut pas être remplacé dynamiquement

Injection de Dépendance - Mauvais Exemple



```
import fs from "fs/promises";

export class FileWriter {
  constructor(private filePath: string) {}

  async write(msg: string) {
    await fs.appendFile(this.filePath, msg);
  }
}
```

```
import { FileWriter } from "./FileWriter";

export class Logger {
  private fileWriter: FileWriter;

  constructor(filePath: string) {
    this.fileWriter = new FileWriter(filePath);
  }

  async log(msg: string) {
    const formatted = `${new Date().toISOString()} - ${msg}\n`;
    await this.fileWriter.write(formatted);
  }
}
```

```
import { Logger } from "./Logger";

const logger = new Logger("app.log");
await logger.log('Message');
```

Injection de Dépendance - Mauvais Exemple



- ✓ Single Responsibility : 1 classe pour l'écriture des fichiers, 1 classe pour le logging
- ✗ Open-closed principle : pour loguer dans le terminal, il faudrait modifier le code
- ✗ Dependency inversion principle : Coffee est une dépendance codée en dur, elle ne peut pas être remplacée dynamiquement

Injection de Dépendance - Mauvais Exemple



```
export class Coffee {  
  constructor(  
    private type: "arabica" | "robusta",  
  ) {}  
}
```

```
import { Coffee } from "./Coffee";  
  
export class Cup {  
  constructor(  
    private capacity: number,  
    private coffee: Coffee  
  ) {}  
}
```

```
import { Coffee } from "./Coffee";  
import { Cup } from "./Cup";  
  
const coffee = new Coffee('arabica');  
const coffeeCup = new Cup(10, coffee);
```



- ✓ Single Responsibility : 1 classe pour le café, 1 classe pour la tasse
- ✓ Open-closed principle : nous pourrions passer une spécialisation de Coffee pour modifier le code (mais cela pourrait briser le principe de substitution de Liskov)
- ✗ Dependency inversion principle : Coffee est une dépendance codée en dur, elle ne peut pas être remplacée dynamiquement

Injection de Dépendance - Mauvais Exemple



```
import fs from "fs/promises";

export class FileWriter {
    constructor(private filePath: string) {}

    async write(msg: string) {
        await fs.appendFile(this.filePath, msg);
    }
}
```

```
import { FileWriter } from "./FileWriter";

export class Logger {
    constructor(private fileWriter: FileWriter) {}

    async log(msg: string) {
        const formatted = `${new Date().toISOString()} - ${msg}\n`;
        await this.fileWriter.write(formatted);
    }
}
```

```
import { FileWriter } from "./FileWriter";
import { Logger } from "./Logger";

const writer = new FileWriter("app.log");
const logger = new Logger(writer);
await logger.log('Message');
```

Injection de Dépendance - Mauvais Exemple

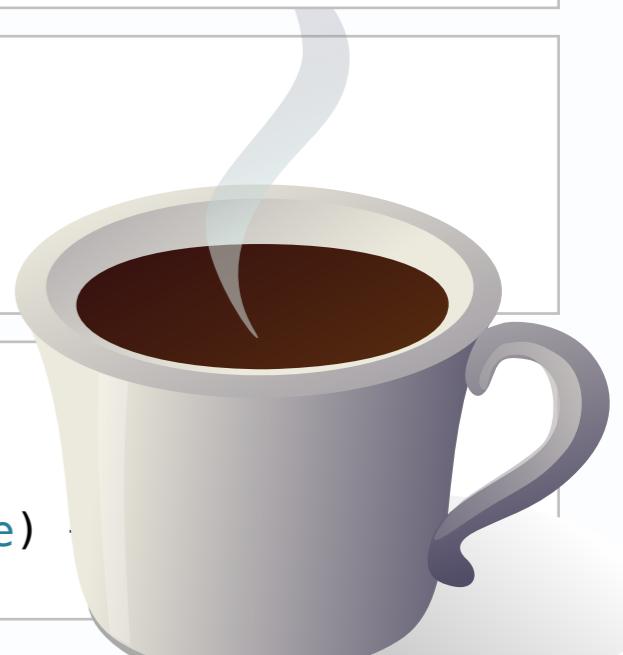


- ✓ Single Responsibility : 1 classe pour l'écriture de fichiers, 1 classe pour le logging
- ✓ Open-closed principle : nous pourrions passer une spécialisation de FileWriter pour modifier le code (mais cela pourrait briser le principe de substitution de Liskov)
- ✗ Dependency inversion principle : Coffee est une dépendance codée en dur, elle ne peut pas être remplacée dynamiquement

Injection de Dépendance - Bon Exemple



```
export interface DrinkInterface {}  
  
import { DrinkInterface } from "./DrinkInterface";  
  
export class Coffee implements DrinkInterface {  
  constructor(private type: "arabica" | "robusta") {}  
}  
  
import { DrinkInterface } from "./DrinkInterface";  
  
export class Cup {  
  constructor(private capacity: number, private drink: DrinkInterface)  
}  
  
import { Coffee } from "./Coffee";  
import { Cup } from "./Cup";  
  
const coffee = new Coffee("arabica");  
const coffeeCup = new Cup(10, coffee);
```



- ✓ Single Responsibility : 1 classe pour le café, 1 classe pour la tasse
- ✓ Open-closed principle : on peut passer une implémentation de DrinkInterface pour modifier le code
- ✓ Dependency inversion principle : la tasse dépend d'une abstraction, nous pourrions facilement créer une tasse de thé



Injection de Dépendance - Bon Exemple

```
export interface WriterInterface {
  write(msg: string): Promise<void>;
}

import fs from "fs/promises";
import { WriterInterface } from "./WriterInterface";

export class FileWriter implements WriterInterface {
  constructor(private filePath: string) {}

  async write(msg: string) {
    await fs.appendFile(this.filePath, msg);
  }
}

import { WriterInterface } from "./WriterInterface";

export class Logger {
  constructor(private writer: WriterInterface) {}

  async log(msg: string) {
    const formatted = `${new Date().toISOString()} - ${msg}\n`;
    await this.writer.write(formatted);
  }
}

import { FileWriter } from "./FileWriter";
import { Logger } from "./Logger";

const writer = new FileWriter("app.log");
const logger = new Logger(writer);
await logger.log('Message');
```

Injection de Dépendance - Bon Exemple



- ✓ Single Responsibility : 1 classe pour l'écriture des fichiers, 1 classe pour le logging
- ✓ Open-closed principle : on peut passer une implémentation de WriterInterface pour modifier le code
- ✓ Dependency inversion principle : Logger dépend d'une abstraction, nous pourrions facilement créer un logger qui loguerait ailleurs que dans un fichier

Injection de Dépendance - @Rolex



- Pour injecter des dépendances dans vos projets vous pouvez utiliser la classe DependencyInjector

```
import { DependencyInjector } from './lib/dependencies/DependencyInjector.js';
export const di = new DependencyInjector();
```

- Pour fournir une dépendance on peut soit fournir l'objet directement

```
di.provide('router', new Router({
  routes: routes,
  useHistory: true,
}));
```

- Soit passer une fonction :

```
di.provide('router', () => {
  return new Router({
    routes: routes,
    useHistory: true,
  });
});
```

- La clé peut également être un Symbol pour limiter les risques de conflits



Injection de Dépendance - @Rolex

- Pour récupérer sa dépendance on utilise la méthode inject

```
import "./components/top-bar";
import { css, html, LitElement } from "lit";
import { di } from "./di";

export class AppComponent extends LitElement {
  router = di.inject("router");

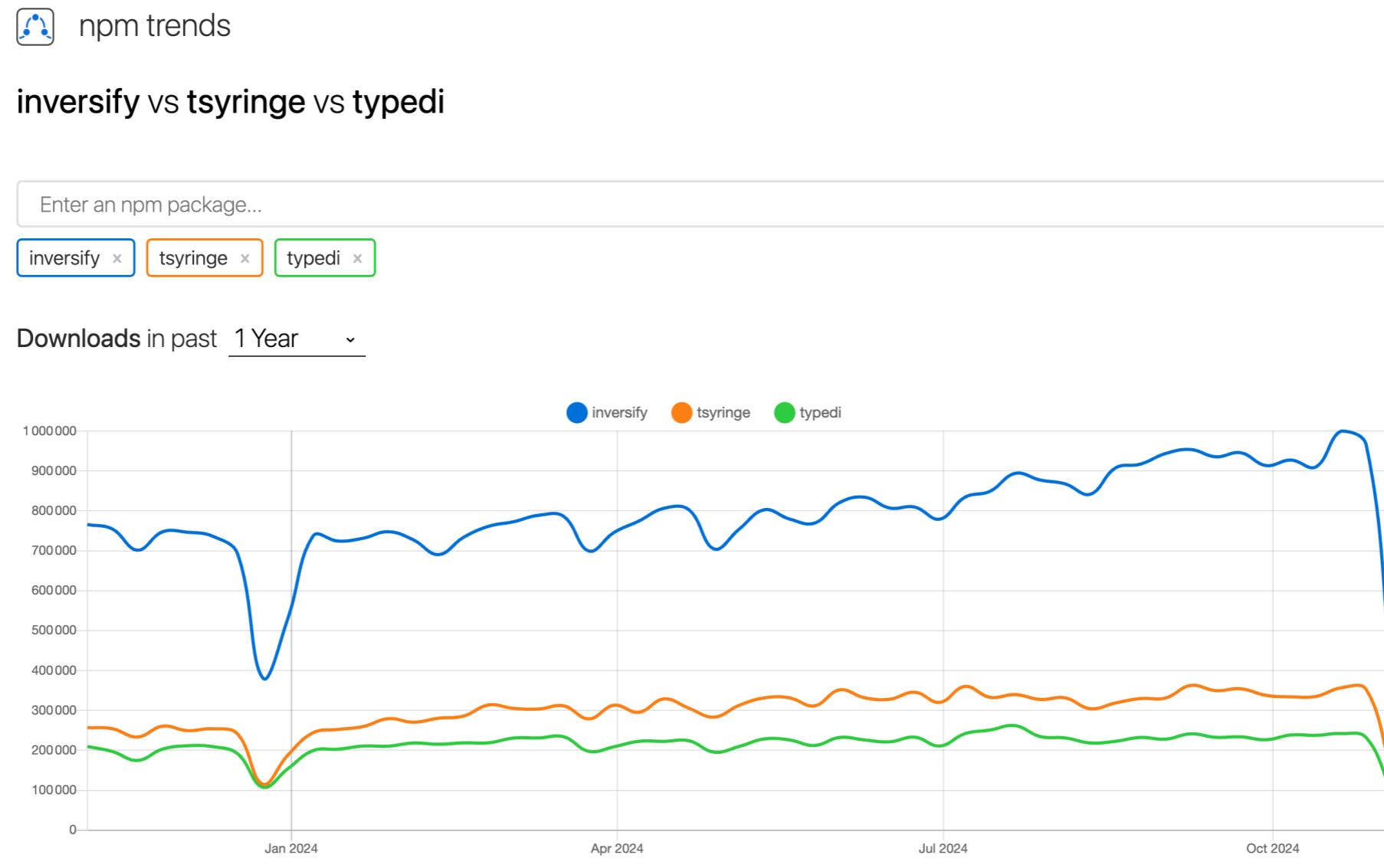
  static styles = css`
    padding: 1rem;
  `;
  render() {
    return html`<my-top-bar></my-top-bar>
<main>
  <rlx-flx-router-view .router=${this.router}></rlx-flx-router-view>
</main>`;
  }
}

customElements.define("my-app", AppComponent);
```

Injection de Dépendance - @Rolex



- Alternatives
 - Les bibliothèques d'Injection de Dépendance :
Basées sur des décorateurs et donc de pouvoir builder le projet



Injection de Dépendance - @Rolex



- Alternatives
 - Le context
Permet de traverser une hiérarchie de composants, mais limité aux composants (autres services ?)
<https://lit.dev/docs/data/context/>

Routeur - Routes



- Pour définir ses routes on associe un path (unique) à un composant :

```
import { SettingsComponent } from "./pages/settings";
import { HomeComponent } from "./pages/home";

/** @type {import('./lib/router/route').Route[]} */
export const routes = [
  { path: "/", name: "home", component: HomeComponent },
  { path: "/settings", name: "settings", component: SettingsComponent },
];
```

- Le name (unique) permettra de générer des liens plus facilement
- Plutôt qu'un composant on peut également lier à une fonction render :

```
/** @type {import('./lib/router/route').Route[]} */
export const routes = [
  { path: "/", name: "home", render: () => html`<h2>Home page</h2>` },
  { path: "/settings", name: "settings", render: () => html`<h2>Home page</h2>` },
];
```

Routeur - Router



- Création du router

```
new Router({  
  routes: routes, // les routes configurées  
  entry: { url: '/home' }, // la route par défaut  
  useHistory: true, // permet d'utiliser l'historique du  
    navigateur back/forward  
  useMemory: true, // permet de stocker la dernière route visitée  
    dans le localStorage pour la prochaine visite  
});
```

- Pour changer de page en la conservant dans l'historique

```
this.router.push(event.target.pathname);  
  
this.router.push({name: 'Home'});  
  
this.router.push({name: 'user-details', parameters: { userId: 3 } });
```

- Pour changer de page sans la conserver dans l'historique

```
this.router.to(event.target.pathname);  
{ path: ':userId' }
```

- Pour accéder aux paramètres de la route :

```
this.router.resolve.route.parameters.userId
```



Routeur - Router

- Pour définir où le composant routé doit s'afficher sur la page :

```
<main>
  <rlx-flx-router-view .router=${this.router}></rlx-flx-router-view>
</main>
```

- Attention à passer la même instance du router partout dans l'app
(Cf chapitre sur l'injection de dépendance)

Routeur - Routes Avancées



- Des fonctions `onBeforeEnter` et `onBeforeLeave` peuvent s'exécuter au moment où le composant routé apparaît ou disparaît de la page :

```
{  
  path: '/',
  name: 'home',
  component: HomeComponent,
  onBeforeEnter: () => console.log('Before entering home'),
  onBeforeLeave: () => console.log('Before leaving home'),
}
```

- Enfin on peut définir des routes imbriquées (le composant parent devra alors à nouveau définir un composant `rlx-flx-router-view` dans son template)

```
{  
  path: '/users',
  name: 'users',
  component: UsersComponent,
  children: [  
    {  
      path: '',
      render: () => html`<p>Select a user from the list</p>`,  
    },  
    {  
      path: ':id',
      name: 'user-detail',
      component: UserDetailsComponent,  
    },  
  ],
}
```

Store - Introduction



- La version de Lit @Rolex ajoute une classe Controller et son alias Store
- Par rapport aux Contrôleurs vus précédemment elle permet de rafraîchir plusieurs composants lors d'un update, permettant ainsi de les faire communiquer

```
/** @typedef {import("lit").LitElement} OriginalLitElement */

export class Controller extends EventTarget {
  /**
   * @type {Set<OriginalLitElement>}
   */
  hosts = new Set();

  /**
   * @param {OriginalLitElement} host
   */
  addHost(host) {
    this.hosts.add(host);
    host.addController(this);
  }

  /**
   * @param {OriginalLitElement} host
   */
  removeHost(host) {
    host.removeController(this);
    this.hosts.delete(host);
  }

  requestUpdate() {
    this.hosts.forEach((host) => host.requestUpdate());
  }
}
```



Store - Introduction

- Exemple

```
import { Store } from "../lib/rlx-lit";

export class SettingsStore extends Store {
  state = {
    title: "My App",
  };

  updateTitle(title) {
    this.state.title = title;
    this.requestUpdate();
  }
}
```

- Pour l'injecter :

```
import { html, LitElement } from "lit";
import { di } from "../di";

export class SettingsComponent extends LitElement {
  store = di.inject("settings-store");

  connectedCallback() {
    super.connectedCallback();
    this.store.addHost(this);
  }
}
```

```
import { LitElement, css, html } from "lit";
import { di } from "../di";
import { SettingsStore } from "../services/SettingsStore";

export class TopBarComponent extends LitElement {
  router = di.inject("router");
  /* @type {SettingsStore} */
  store = di.inject("settings-store");

  connectedCallback() {
    super.connectedCallback();
    this.store.addHost(this);
  }
}
```