



formation.tech

Formation Stencil

Romain Bohdanowicz

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



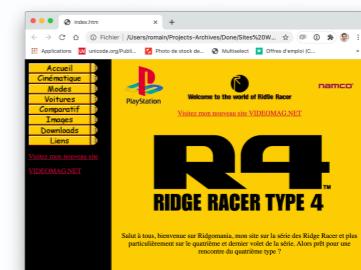
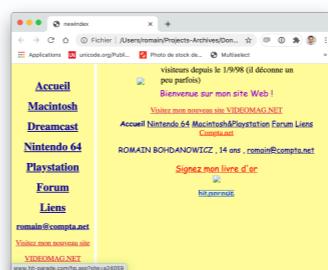
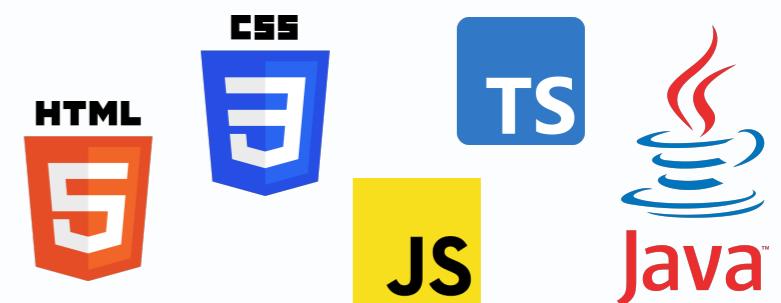
formation.tech

Introduction



Introduction - Formateur

- Romain Bohdanowicz
Ingénieur EFREI 2008, spécialité en Ingénierie Logicielle
- Expérience
Formateur/Développeur Freelance depuis 2006
Près de 3000 jours de formation 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 - formation.tech



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



Introduction - Autres activités



- Développement
- Coaching
- Audit de code
- CTO externalisé
- ...

Introduction - Et vous ?



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



formation.tech

Web Components

Web Components - Introduction



- Les Web Components sont implémentés par un ensemble de Web APIs permettant de créer des balises/éléments HTML customs
- 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 encapsuler dans des Web Components et inversement

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



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 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 `fromAssociated`
- 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 sont désactivés
- › `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/>



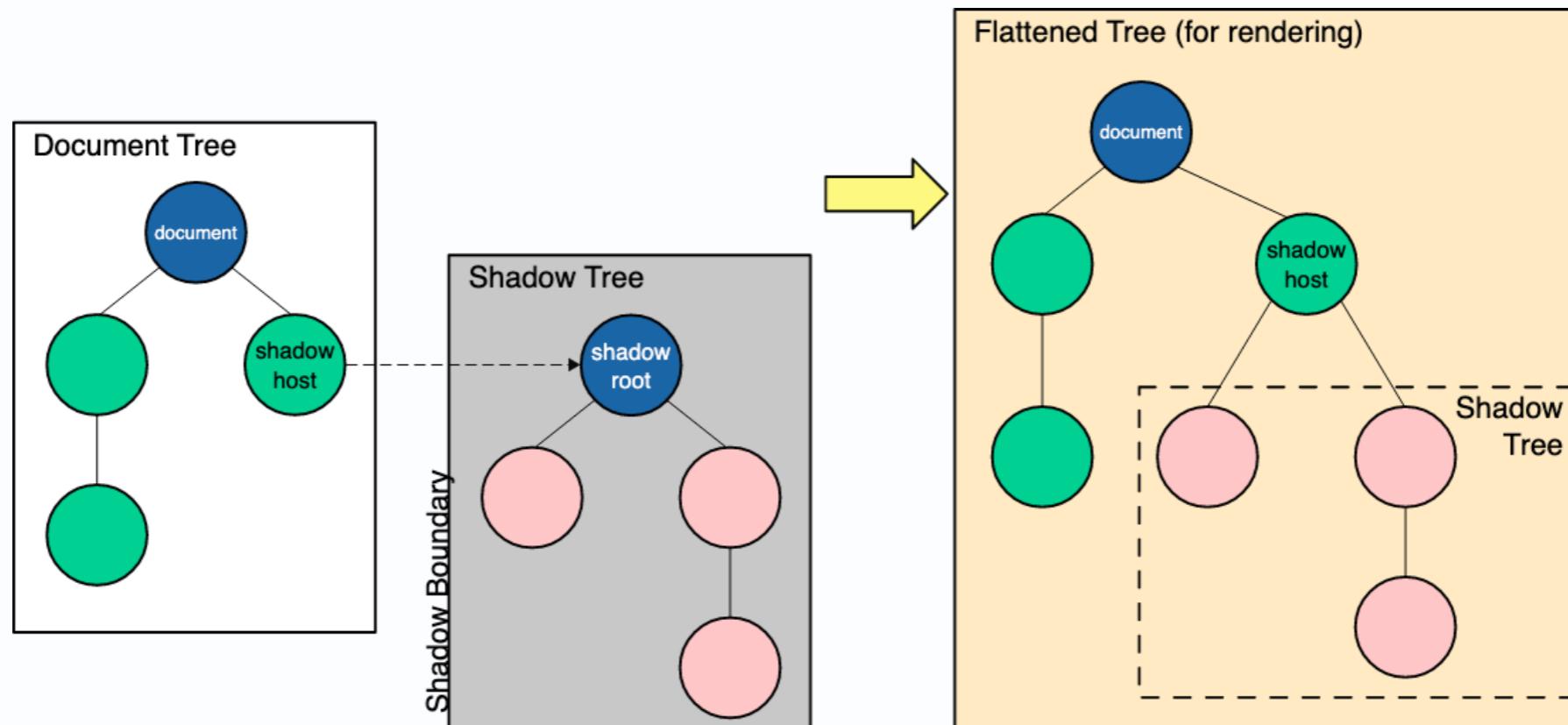
formation.tech

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, style et comportement caché 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 passé pour chaque instance
- A la place on privilégie les HTML Templates
- Les templates ne sont pas rendus par le navigateur
- Leur contenu n'est parse 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>
`;
```

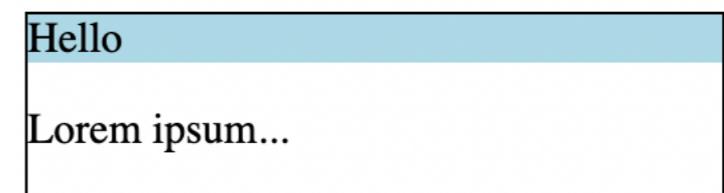
```
class Card extends HTMLElement {
  constructor() {
    super();
    const shadow = this.attachShadow({ mode: 'open' });
    shadow.append(templateEl.content.cloneNode(true));
```

2

10

30

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





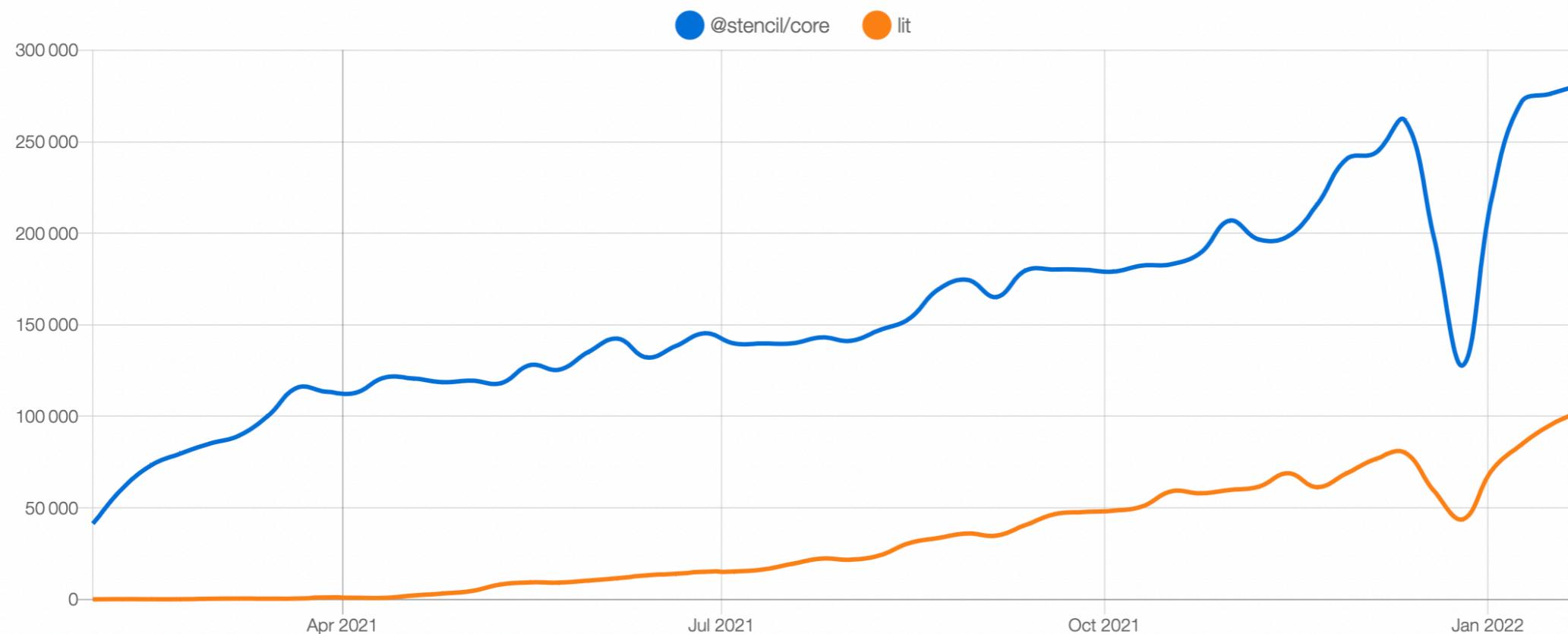
formation.tech

Web Component Librairies



Web Component Librairies

- Plusieurs bibliothèques sont dédiés aux Web Components :
 - Lit : créé par Google, successeur de lit-html et Polymer
 - Stencil : créé par Ionic
 - Lightning Web Components : créé par SalesForce



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://developers.google.com/web/fundamentals/web-components?hl=en>
https://developer.mozilla.org/en-US/docs/Web/Web_Components



formation.tech

Stencil

Stencil - Introduction



- Crée par les développeurs de Ionic Framework
- Présenté pour la première fois en août 2017 au Polymer Summit de Copenhague
<https://www.youtube.com/watch?v=UfD-k7aHkQE>
- N'est pas un nouveau framework mais un compilateur pour les Web Components (zéro dépendance une fois compilé)
- Permet de créer des custom elements natifs avec :
Virtual DOM, SSR, Async Rendering, AOT compilation, Data-Binding
- Inspiré des meilleures partie d'Angular, React, VueJS et Polymer
- Compatible avec tous les frameworks
- Utilise TypeScript et JSX
- Permet de créer des composants ou des apps complètes
- Licence MIT

Stencil - Web components



- Ensemble de 4 normes
 - HTML Imports
 - Custom Elements
 - Shadow DOM
 - HTML Templates
- Crée par Google en 2012 et normé depuis
- Shadow DOM et Custom Elements ont connues 2 versions (v0 puis v1), d'autres des modifications (HTML Imports)

Stencil - Web components



- Compatibilité (septembre 2018) :

<https://www.webcomponents.org>

<https://caniuse.com/#search=components>

Browser support	CHROME	OPERA	SAFARI	FIREFOX	EDGE
HTML TEMPLATES	STABLE	STABLE	STABLE	STABLE	STABLE
CUSTOM ELEMENTS	STABLE	STABLE	STABLE	POLYFILL DEVELOPING	POLYFILL CONSIDERING
SHADOW DOM	STABLE	STABLE	STABLE	POLYFILL DEVELOPING	POLYFILL CONSIDERING
ES MODULES	STABLE	STABLE	STABLE	STABLE	STABLE
HTML IMPORTS	STABLE	STABLE	POLYFILL	POLYFILL	POLYFILL

Stencil - Lazy Loading



- Les composants Stencil sont lazy-loadés, pas besoin de charger l'ensemble des composants au démarrage
- N'utilise pas Webpack ou les HTML imports
- C'est le navigateur qui décide quels sont les fichiers à charger

Stencil - Collections

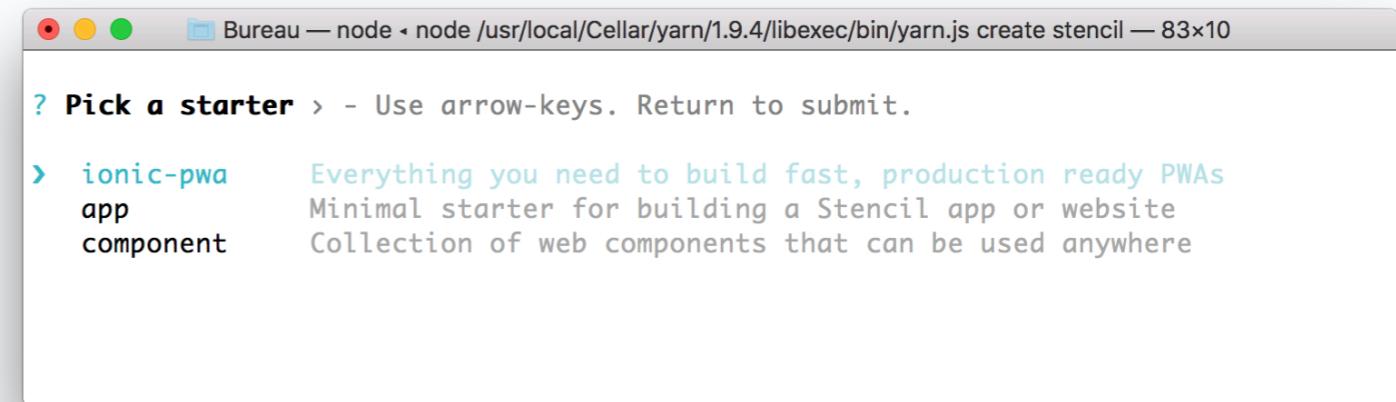


- Stencil facilite la création de collections de composants d'UI
 - Fin des frameworks d'UI monolithiques type Bootstrap avec des bindings pour chaque framework JS (ui-bootstrap, ng-bootstrap, react-bootstrap...)
- Publication sur npm
- Facilite le partage de ressources statiques (svg, icônes, etc...)
- Création de catégories de composant pour réduire le nombre de requêtes
- Une collection est un regroupement de composants lazy-loadés
- Ex : @ionic/core

Stencil - Installation



- Prérequis :
 - Node.js 16+
 - npm, yarn ou pnpm
- Création d'un projet
 - Avec npm
 npm init stencil
 - Avec Yarn
 yarn create stencil
- 3 choix :
 - ionic-pwa, une application ionic
 - app, une app stencil
 - component, une collection de composants stencil





Stencil - Génération

- Une fois un projet créé, on peut générer un nouveau composant avec la commande :
`stencil generate`

```
stencil-library — roman@MacBook-Pro-de-Romain — ..encil-library — -zsh — 113x13
[→ stencil-library git:(master) npx stencil generate
[33:30.4]  @stencil/core
[33:30.6]  v4.11.0 🎉
[✓ Component tag name (dash-case): ... my-button
[✓ Which additional files do you want to generate? > Stylesheet (.css), Spec Test (.spec.tsx), E2E Test (.e2e.ts)

$ stencil generate my-button

The following files have been generated:
- src/components/my-button/my-button.tsx
- src/components/my-button/my-button.css
- src/components/my-button/test/my-button.spec.tsx
- src/components/my-button/test/my-button.e2e.ts
```

Stencil - Component



- › Un composant Stencil :
 - est écrit en TypeScript
 - utilise un décorateur comme Angular
 - utilise le JSX comme React

```
import { Component, ComponentInterface, Prop, Host, h } from '@stencil/core';

@Component({
  tag: 'my-duration',
  styleUrl: 'duration.scss',
  shadow: true,
})
export class Duration implements ComponentInterface {
  @Prop() time = 0;
  @Prop() isMobilePlayer = false;

  private formatDuration(duration: number) {
    const minutes = Math.floor(duration / 60);
    const seconds = Math.floor(duration - minutes * 60);
    return `${String(minutes).padStart(2, '0')}:${String(seconds).padStart(2, '0')}`;
  }

  render() {
    return <Host>{this.formatDuration(this.time)}</Host>;
  }
}
```

Stencil - Component



- Options du décorateur :
 - tag: nom de la balise
 - shadow: utilisation du Shadow DOM
 - scope: simule le Shadow DOM avec les CSS Modules
 - styles, styleUrls, stylesUrl : chemin vers le CSS/SCSS ou CSS/SCSS inline
 - assetsDirs: répertoire assets (images...) du composant
 - formAssociated: si control de formulaire (voir <https://stenciljs.com/docs/form-associated>)



formation.tech

JSX

JSX - Introduction



- JSX est une syntaxe inventée par Facebook pour React qui étend la syntaxe JavaScript
- Le JSX permet de créer un arbre d'éléments en mémoire appelé Virtual DOM
- Il permet pour des applications JavaScript de programmer l'interface de façon déclarative comme en HTML
- La syntaxe utilisée est XML (attention aux balises vide comme ``)
- Le JSX n'a pas pour vocation à être implémenté par les navigateurs mais d'être transformé en JavaScript par un transpileur (Babel, esbuild...)
- D'autres projets utilisent également JSX aujourd'hui : Vue.js, preact, Solid, Qwik...
- Le JSX se transforme en JavaScript à l'aide de transpilers comme Babel ou esbuild

JSX - Runtime classic



- Avec le runtime classic ce code

```
const now = new Date();
const prenom = 'Romain';

return (
  <div class="App" id="App">
    <p>Heure : {now.toLocaleTimeString()}</p>
    <p>Prénom : {prenom}</p>
  </div>
);
```

- Devient

```
const now = new Date();
const prenom = 'Romain';

return h(
  'div',
  { class: 'App', id: 'App' },
  h('p', null, 'Heure : ', now.toLocaleTimeString()),
  h('p', null, 'Prénom : ', prenom),
);
```

JSX - Accolades



- En JSX, on utilise les accolades pour utiliser des expressions JavaScript
- Les accolades créées des noeuds de texte si on utilise les types string, number ou JSX.Element :

```
function WillRender() {  
  const name = 'Romain';  
  const age = 38;  
  const element = <div>React Element</div>;  
  
  return (  
    <div className="App">  
      <p>Hello {name}, I'm {age}</p>  
      {element}  
    </div>  
  );  
}
```

- Les types boolean, null ou undefined peuvent être utilisés mais il ne créent pas de noeud

```
function WillDoNothing() {  
  return (  
    <div className="App">  
      {undefined} {null} {false} {true}  
    </div>  
  );  
}
```



JSX - Accolades

- On peut également utiliser des types itérables qui vont créer plusieurs noeuds

```
function WillLoop() {  
  return (  
    <div className="App">  
      {'ABC', 123, <div>Element</div>}  
    </div>  
  );  
}
```

- Le même comportement s'applique :
 - les types string, number et JSX.Element s'affichent
 - les types boolean, null et undefined n'affichent rien



JSX - Accolades

- Attention les objets autres que string, React.element et Iterable provoquent une erreur :

```
function WillThrow() {  
  const coords = { x: 1, y: 2 };  
  const now = new Date();  
  return (  
    <div className="App">  
      {coords} {now}      ← Erreur : vNode passed as children has unexpected type  
    </div>  
  );  
}
```



JSX - Accolades

- Les commentaires s'écrivent de la manière suivante (valeur === undefined) :
{/* Mon commentaire */}

```
function Comment() {  
  return (  
    <div className="App">  
      {/* Comment */}  
    </div>  
  );  
}
```

JSX - Attributs JSX



- Les attributs JSX sont passés en 2ème paramètre de h

```
// En JSX :  
<input type="text" maxLength={10} required />
```

```
// Une fois buildé :  
h('input', {  
  type: 'text',  
  maxLength: 10,  
  required: true,  
}),
```

```
// En JSX :  
h('my-hello', {  
  name: 'Romain',  
  age: 38,  
  isActive: true,  
})
```

```
// Une fois buildé :  
<my-hello name="Romain" age={38} isActive />
```

- Lorsqu'on passe une constante de type string (ici "text" ou "Romain") les accolades sont optionnelles
- Lorsqu'on ne passe pas de valeur (ici required ou isActive) cela revient à passer true

JSX - Attributs JSX



- On peut également passer directement un object contenant plusieurs attributs JSX avec la syntaxe :

```
{...obj}
```

```
function Attributes() {
  const inputProps = {
    type: 'text',
    maxLength: 10,
  };

  return (
    <div className="App">
      <input {...inputProps} required />
    </div>
  );
}
```

- {...inputProps} traduit type="text" maxLength={10}



JSX - Rendu conditionnel

- Pour rendre des éléments de manière conditionnelle on réutilise ce qu'on a appris à propos des accolades JSX

```
function App() {  
  let element;  
  
  if (condition) {  
    element = <div>If true</div>  
  }  
  
  return (  
    <div className="App">  
      {element}  
    </div>  
  );  
}
```

```
function App() {  
  let element;  
  
  if (condition) {  
    element = <div>If true</div>  
  } else {  
    element = <div>If not</div>  
  }  
  
  return (  
    <div className="App">  
      {element}  
    </div>  
  );  
}
```

- Un élément sera affiché, undefined sera ignoré



JSX - Rendu conditionnel

- › On peut également utiliser des expressions conditionnelles

```
function App() {  
  const element = condition && <div>If true</div>;  
  
  return (  
    <div className="App">  
      {element}  
    </div>  
  );  
}  
  
function App() {  
  return (  
    <div className="App">  
      {condition && <div>If true</div>}  
    </div>  
  );  
}
```



JSX - Rendu conditionnel

- › On peut également utiliser des expressions conditionnelles

```
function App() {  
  const element = condition ? <div>If true</div> : <div>If not</div>;  
  
  return (  
    <div className="App">  
      {element}  
    </div>  
  );  
}  
  
function App() {  
  return (  
    <div className="App">  
      {condition ? <div>If true</div> : <div>If not</div>}  
    </div>  
  );  
}
```

JSX - Listes



- Pour afficher plusieurs éléments il suffit d'utiliser un tableau

```
function App() {  
  const names = ['Romain', 'Edouard'];  
  const elements = [];  
  
  for (const name of names) {  
    elements.push(<div key={name}>{name}</div>)  
  }  
  
  return (  
    <div className="App">  
      {elements}  
    </div>  
  );  
}
```

- L'attribut key est optionnel, en dev un warning s'affichera dans la console s'il n'est pas présent
- En prod le warning disparait

JSX - Listes



- L'attribut key sert à Stencil à améliorer le diffing et mieux déterminer les changements apportés à des listes
- La valeur passée doit être unique (pour ce tableau, pas pour l'ensemble de l'app)
- L'attribut key s'utilise sur les éléments à la racine du tableau uniquement
- La valeur doit être la plus stable possible dans le temps :
 - l'élément du tableau d'origine si les valeurs ne sont pas éditables et uniques
 - l'indice du tableau d'origine si on ne peut pas le trier ou supprimer des éléments
 - l'id est idéal si tableau d'enregistrement provenant de la base de données
 - si besoin de générer une valeur (uniq(), uuid(), Math.random()) ne pas le faire à l'intérieur du composant (au prochain rendu, la valeur changera)

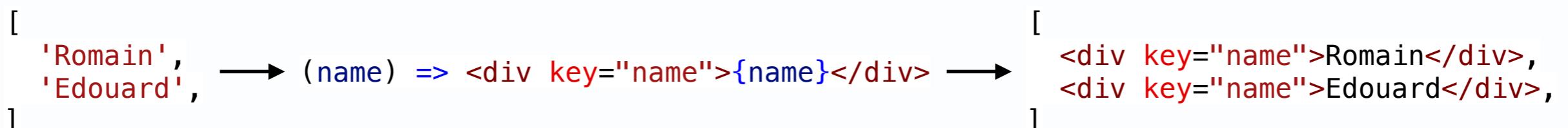
JSX - Listes



- Comme avec les listes on peut créer le tableau en une seule expression en utilisant .map

```
function App() {  
  const names = ['Romain', 'Edouard'];  
  const elements = names.map((name) => <div key="name">{name}</div>);  
  
  return (  
    <div className="App">  
      {elements}  
    </div>  
  );  
}
```

- Avec .map chaque élément du tableau est placé dans un nouveau tableau en étant transformé par la fonction de transformation :





JSX - Listes

- Utilisation de .map directement en JSX

```
function App() {
  const names = ['Romain', 'Edouard'];

  return (
    <div className="App">
      {names.map((name) => <div key="name">{name}</div>)}
    </div>
  );
}
```

Stencil - Ref



- › On peut récupérer une Ref en JSX avec ref
- › La ref ne sera disponible qu'un fois l'élément rendu dans le DOM

```
<staytuned-slider
  ref={(el) => (this.minifiedSlider = el as HTMLStaytunedSliderElement)}
  {...scrubberOpts}
  class="main-scrubber"
/>
```



formation.tech

Décorateurs



Décorateurs - Props

- Pour passer des paramètres sous forme d'attribut ou de propriétés au composant on utilise des propriétés décorés avec `@Prop`
- Le camelCase se transforme en kebab-case pour les attributs
- Les props peuvent être muable en passant l'option `mutable: true`
- Attention les props de type Object et Array ne peut s'utiliser qu'en JSX ou DOM (pas HTML)

```
import { Component, ComponentInterface, Prop, h } from '@stencil/core';

@Component({
  tag: 'my-duration',
  styleUrl: 'duration.scss',
  shadow: true,
})
export class Duration implements ComponentInterface {
  @Prop() time = 0;
  @Prop() isMobilePlayer = false;

  // ...

<staytuned-duration time={this.currentTime} />
```

Décorateurs - State



- Le state permet de définir des valeurs internes au composant qui provoquera son rafraîchissement en cas de mise à jour

```
@State() format: Format = Format.collapsed; // collapsed, normal, expand
```



Décorateurs - Watch

- On peut surveiller des changements de Prop ou State avec Watch

```
@State() format: Format = Format.collapsed; // collapsed, normal, expand
@Watch('format')
setAnimation() {
    this.animating = true;
    clearTimeout(this.animationTimeout);
    this.animationTimeout = setTimeout(() => {
        this.animating = false;
    }, 300);
}
```



Décorateurs - Element

- On peut récupérer une référence sur l'Element hôte avec @Element

```
@Element() host: HTMLMyPlayerElement;
```

Décorateurs - Method



- On peut exposer une méthode dans l'API public avec @Method

```
@Component({
  tag: 'my-player',
  shadow: true,
})
export class Player {
  @Host() host: HTMLMyPlayerElement;

  @Method()
  play() {
    return this.host.querySelector('video').play();
  }
}

const playerEl = document.createElement('my-player');
playerEl.play();
```



formation.tech

Framework Integration

Framework Integration - Introduction



- Stencil propose des exports vers les principaux frameworks :
 - Angular
 - React
 - Vue
- On utilise pour ça les paquets :
 - `@stencil/angular-output-target`
 - `@stencil/react-output-target`
 - `@stencil/vue-output-target`

Framework Integration - Config



- Il est recommandé d'utiliser l'intégration avec les frameworks via des monorepos, idéalement géré via des outils dédiés : Lerna, Nx, Turborepo
- La configuration se fait dans stencil.config.ts, exemple :
<https://github.com/ionic-team/stencil-ds-output-targets>
<https://github.com/ionic-team/ionic-framework>
- ```
export const config: Config = {
 namespace: 'component-library',
 taskQueue: 'async',
 outputTargets: [
 angularOutputTarget({
 componentCorePackage: 'component-library',
 directivesProxyFile: '../component-library-angular/src/directives/proxies.ts',
 }),
 reactOutputTarget({
 componentCorePackage: 'component-library',
 proxiesFile: '../component-library-react/src/components.ts',
 }),
 vueOutputTarget({
 componentCorePackage: 'component-library',
 proxiesFile: '../component-library-vue/src/proxies.ts',
 componentModels: vueComponentModels,
 }),
 // ...
],
};
```

# Framework Integration - Angular



- Les versions récentes de l'export Angular propose quelques options :  
<https://stenciljs.com/docs/angular#api>
- componentCorePackage : le nom de la lib exportée
- outputType :
  - component : tous les composants déclarés dans un seul NgModule
  - scam : un composant exporté + un NgModule par composant
  - standalone : des composants exportés en standalone (sans NgModule)
- directivesProxyFile : le chemin vers le fichier contenu les composants exportés
- directivesArrayFile : le chemin vers un tableau de composants exportés (si besoin de les inclure dans un NgModule)

# Framework Integration - Angular



- customElementsDir : le nom du dossier contenant les éléments personnalisés (dans le projet Stencil)
- excludeComponents : les composants à ne pas exporter (certains nécessitent d'être écrits manuellement, si besoin de services...)
- valueAccessorConfigs : configure les ControlValueAccessor d'Angular, c'est à dire les composants qui seront utilisés comme contrôles de formulaire :  
<https://angular.io/api/forms/ControlValueAccessor>

# Framework Integration - Angular



- On peut avoir plusieurs configs Angular (ex : standalone + component)

```
import { Config } from '@stencil/core';
import { angularOutputTarget } from '@stencil/angular-output-target';

export const config: Config = {
 namespace: 'stencil-library',
 outputTargets: [
 angularOutputTarget({
 componentCorePackage: 'stencil-library',
 outputType: 'component',
 directivesProxyFile: '../component-library/src/proxies.ts',
 directivesArrayFile: '../component-library/src/array.ts',
 }),
 angularOutputTarget({
 componentCorePackage: 'stencil-library',
 outputType: 'standalone',
 directivesProxyFile: '../component-library/src/proxies.ts',
 }),
 {
 type: 'dist-custom-elements',
 copy: [
 {
 src: '../scripts/custom-elements',
 dest: 'components',
 warn: true
 },
 {
 dir: 'components',
 includeGlobalScripts: false
 }
],
 type: 'dist',
 esmLoaderPath: '../loader',
 },
],
};
```



**formation.tech**

# Tests Automatisés

# Tests Automatisés - Introduction



- Comment tester son code ?
  - Manuellement : une personne effectue les tests
  - Automatiquement : les tests ont été programmés
- Historique
  - à partir de 1989 en Smalltalk et le framework SUnit
  - à partir de 1997 en Java avec JUnit
  - à partir de 2004 dans le navigateur avec Selenium

# Tests Automatisés - Pourquoi ?



- Pourquoi automatiser les tests ?
  - plus l'application grandit, plus le risque d'introduire une régression est grand  
ex: modifier une fonction qui est partagé par différentes
  - tester manuellement à chaque itération prendra à terme plus de temps qu'écrire le code du test
  - les tests automatisés peuvent se lancer sur différentes plate-formes et navigateurs très simplement
  - les tests aident à la compréhension du code, les lire permet de comprendre des comportements qui n'ont pas toujours été documentés
- Pourquoi continuer de tester manuellement ?
  - certains tests peuvent être simple à faire manuellement mais compliqués à automatiser (drag-n-drop...)
  - automatiser permet d'avoir accès à des choses inaccessibles manuellement (bouton caché par une popup...)



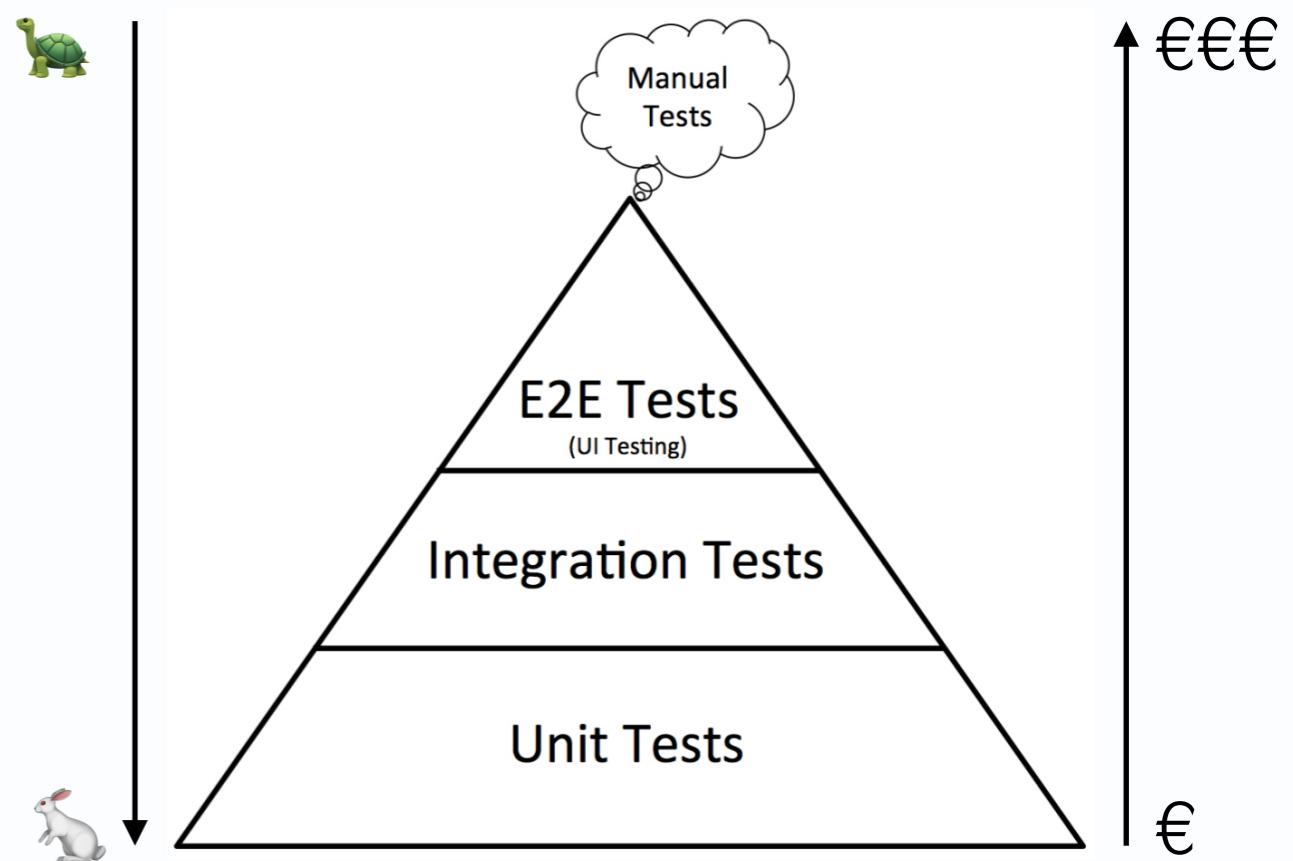
# Tests Automatisés - Types de tests

- Types de tests :
  - tests de code statiques / linters
  - tests de code dynamiques / tests unitaires...
  - tests de déploiement
  - tests de sécurité (pentests)
  - tests de montée en charge
  - ...



# Tests Automatisés - Pyramide des tests

- 3 types de tests automatisés au niveau code côté Front :
  - Test unitaire  
Permet de tester les briques d'une application (classes / fonctions)
  - Test d'intégration  
Teste que les briques fonctionnent correctement ensembles
  - Test End-to-End (E2E)  
Vérifie l'application dans le client
- Une pyramide
  - plus le test est haut plus il est lent
  - plus le test est haut plus il coûte cher





# Tests Automatisés - Quand exécuter ?

- Quand exécuter ?
  - tout le temps si on arrive à maintenir des tests performants (max 1-2 minutes)
  - avant un commit
  - avant un push
  - sur une plateforme d'intégration ou de déploiement continu (CI/CD)



# Tests Automatisés - Organisation

- Ou placer ses tests ?
  - dans le même répertoire que le code testé
  - dans un répertoire *test* en préservant l'arborescence du répertoire *src*
  - dans un répertoire *test* sans lien avec l'arborescence



**formation.tech**

# Jest / Vitest

# Jest - Introduction



- Framework de test créé en 2014 par Facebook
- Sous Licence MIT depuis septembre 2017
- Permet de lancer des tests :
  - unitaires / d'intégration (dans Node.js)
  - fonctionnels / E2E (via Puppeteer ou PlayWright)
- Peut s'utiliser avec ou sans configuration
- Les tests se lancent en parallèle dans les Workers Node.js
- Intègre par défaut :
  - Calcul de coverage (via Istanbul)
  - Mocks (natifs ou en installant Sinon.JS)
  - Snapshots



# Jest - Installation

- › Installation

```
npm install --save-dev jest
```

```
yarn add --dev jest
```



# Jest - Hello, world !

- Sans configuration, les tests doivent se trouver dans un répertoire `__tests__`, ou bien se nommer `*.test.js` ou `*.spec.js`

```
// src/hello.js
const hello = (name = 'World') => `Hello ${name} !`;

module.exports = hello;
```

```
// __tests__/hello.js
const hello = require('../src/hello');

test('Hello, world !', () => {
 expect(hello()).toBe('Hello World !');
 expect(hello('Romain')).toBe('Hello Romain !');
});
```



# Jest - Lancements des tests

- › Si Jest localement  
node\_modules/.bin/jest
- › Si Jest globalement  
jest
- › Avec un script test dans package.json  
npm run test  
npm test  
npm t

```
// package.json
{
 "devDependencies": {
 "jest": "^22.0.6"
 },
 "scripts": {
 "test": "jest"
 }
}
```

```
MacBook-Pro:hello-jest romain$ node_modules/.bin/jest
PASS __tests__/_hello.js
 ✓ Hello, world ! (3ms)

Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 0.701s, estimated 1s
Ran all test suites.
```



# Jest - Watchers

- › En mode Watch

```
node_modules/.bin/jest --watchAll
jest --watchAll
npm t -- --watchAll
```

```
MacBook-Pro:hello-jest romain$ npm t -- --watchAll
PASS __tests__/hello.js
PASS __tests__/calc.js
```

```
Test Suites: 2 passed, 2 total
Tests: 3 passed, 3 total
Snapshots: 0 total
Time: 0.65s, estimated 1s
Ran all test suites.
```

## Watch Usage

- › Press **f** to run only failed tests.
- › Press **o** to only run tests related to changed files.
- › Press **p** to filter by a filename regex pattern.
- › Press **t** to filter by a test name regex pattern.
- › Press **q** to quit watch mode.
- › Press **Enter** to trigger a test run.



# Jest - Coverage

- Avec calcul du coverage  
node\_modules/.bin/jest --coverage  
jest --coverage  
npm t -- --coverage
- Par défaut le coverage s'affiche dans la console et génère des fichiers Clover, JSON et HTML dans le dossier coverage

```
MacBook-Pro:hello-jest romain$ npm t -- --coverage
PASS __tests__/calc.js
PASS __tests__/hello.js

Test Suites: 2 passed, 2 total
Tests: 3 passed, 3 total
Snapshots: 0 total
Time: 0.722s, estimated 1s
Ran all test suites.

-----|-----|-----|-----|-----|-----|
File	%Stmts	%Branch	%Funcs	%Lines	Uncovered Lines
All files | 86.67 | 100 | 60 | 100 |
calc.js | 83.33 | 100 | 50 | 100 |
hello.js | 100 | 100 | 100 | 100 |
```



# Jest - Partager des variables entre les tests

- › On peut utiliser la portée de closure

```
describe('A suite is just a function', () => {
 let a;

 test('and so is a spec', () => {
 a = true;

 expect(a).toBe(true);
 });
});
```

# Jest - Hooks



- Dans une suite de tests, certaines méthodes seront appelées automatiquement durant la vie du test
  - `beforeEach`, avant chaque test de la suite (*avant chaque test*)
  - `afterEach`, après chaque test
  - `beforeAll`, avant le premier test de la suite (*avant le premier test*)
  - `afterAll`, après le dernier test de la suite

```
describe('A suite with some shared setup', () => {
 let foo = 0;
 beforeEach(() => {
 foo += 1;
 });
 afterEach(() => {
 foo = 0;
 });
 beforeAll(() => {
 foo = 1;
 });
 afterAll(() => {
 foo = 0;
 });
});
```



# Jest - Désactiver les tests

- › Pour désactiver certains tests on peut :
  - les commenter
  - utiliser la fonction `test.skip` au lieu de `test`
  - utiliser la fonction `describe.skip` au lieu de `describe`

```
describe('sum function', () => {
 test.skip('should add positive number', () => {
 expect(sum(1, 2)).toEqual(3);
 });
 test('should convert strings to numbers', () => {
 expect(sum('1', '2')).toEqual(3);
 });
}); ➔ hello-jasmine npm t

> @ test /Users/romain/Desktop/prepa-angular-tests/hello-jasmine
> jasmine

Randomized with seed 01882
Started
.

Ran 1 of 5 specs
1 spec, 0 failures
Finished in 0.011 seconds
```



# Jest - Désactiver les tests

- › On peut également forcer l'exécution d'un test (et pas les autres) avec
  - *test.only* pour un test en particulier
  - *describe.only* pour un groupe de test



# Jest - Tester les erreurs

- › Pour tester les erreurs on utilise
  - un callback dans le expect
  - toThrow

```
export function throwError() {
 throw new Error('Error Message');
}

import { expect, test } from 'vitest';
import { throwError } from './throwError';

test('throwError function', () => {
 expect(() => throwError()).toThrow();
 expect(() => throwError()).toThrow('Error Message');
 expect(() => throwError()).toThrow(/error message/i);
 expect(() => throwError()).toThrow(Error);
 expect(() => throwError()).toThrow(new Error('Error Message'));
});
```



# Jest - Mocks

- Créer une fonction de test

```
export function withCallback(cb: (val: string) => void) {
 cb('ABC');
}
```

```
import { expect, test, vitest } from 'vitest';
import { withCallback } from './withCallback';

test('withCallback function', () => {
 const mockFn = vitest.fn();

 withCallback(mockFn);

 expect(mockFn).toHaveBeenCalled();
 expect(mockFn).toHaveBeenCalledOnce();
 expect(mockFn).toHaveBeenCalledTimes(1);
 expect(mockFn).toHaveBeenCalledWith('ABC');
});
```



# Jest - Mocks

- Avec une implémentation

```
export function withCallbackReturn(cb: (val: string) => string): string {
 return cb('ABC');
}
```

```
import { expect, test, vitest } from 'vitest';
import { withCallbackReturn } from './withCallbackReturn';

test('withCallbackReturn function', () => {
 const mockFn = vitest.fn().mockImplementation(() => 'XYZ');
 // en raccourci :
 // const mockFn = vitest.fn().mockReturnValue('XYZ');

 const val = withCallbackReturn(mockFn);

 expect(mockFn).toHaveBeenCalled();
 expect(mockFn).toHaveBeenCalledOnce();
 expect(mockFn).toHaveBeenCalledTimes(1);
 expect(mockFn).toHaveBeenCalledWith('ABC');
 expect(val).toBe('XYZ');
});
```



# Jest - Mocks

- › Pour espionner une méthode d'un objet

```
export function secondsFromNow(timestamp: number) {
 return Date.now() - timestamp;
}
```

```
import { expect, test, vitest } from 'vitest';
import { secondsFromNow } from './secondsFromNow';

test('secondsFromNow function', () => {
 vitest.spyOn(Date, 'now').mockReturnValue(new Date(2023, 9, 1, 0, 0, 0, 30).getTime())

 expect(secondsFromNow(new Date(2023, 9, 1, 0, 0, 0).getTime())).toBe(30)
});
```

# Jest - Mocks



- › Pour espionner un module

```
export async function fetchUsers() {
 const res = await fetch('https://jsonplaceholder.typicode.com/users');
 return await res.json();
}
```

```
import { fetchUsers } from './fetchUsers';

export async function listUsers() {
 const users = await fetchUsers();
 return users;
}
```

```
import { expect, test, vitest } from 'vitest';
import { listUsers } from './listUsers';
import { fetchUsers } from './fetchUsers';

vitest.mock('./fetchUsers');

test('listUsers function', async () => {
 vitest.mocked(fetchUsers).mockResolvedValue([{id: 1, name: 'Toto'}])
 const users = await listUsers();
 expect(users).toEqual([{id: 1, name: 'Toto'}])
});
```



# Jest - Tester les timers

- La fonction `jest.useFakeTimers()` transforme les timers (`setTimeout`, `setInterval...`) en mock

```
export function timeout(delay: number, arg: any) {
 return new Promise((resolve) => {
 setTimeout(resolve, delay, arg);
 });
}
```

```
import { expect, test, vitest } from 'vitest';
import { timeout } from './timeout';

// ✓ hello function 1002ms
test('hello function', async () => {
 const val = await timeout(1000, 'ABC');
 expect(val).toBe('ABC');
});

// ✓ hello function
test('hello function', async () => {
 vitest.useFakeTimers();
 const promise = timeout(1000, 'ABC');
 vitest.advanceTimersByTime(1000);
 const val = await promise;
 expect(val).toBe('ABC');
 vitest.useRealTimers();
});
```



**formation.tech**

# Tests Stencil



# Tests Stencil - Introduction

- Dans Stencil on lance les tests via la commande :  
`stencil test`
- 2 options :
  - Unitaires  
`stencil test --spec`
  - End to End  
`stencil test --e2e`
- On peut lancer les 2 à la fois :  
`stencil test --spec --e2e`
- Il existe des utilitaires pour se simplifier les tests dans `@stencil/core/testing`



# Tests Stencil - Unitaires

```
import { newSpecPage } from '@stencil/core/testing';
import { HelloWorld } from '../hello-world';

describe('hello-world', () => {
 it('renders', async () => {
 const page = await newSpecPage({
 components: [HelloWorld],
 html: `<hello-world></hello-world>`,
 });
 expect(page.root).toEqualHtml(
 <hello-world>
 <mock:shadow-root>
 <slot></slot>
 </mock:shadow-root>
 </hello-world>
);
 });
});
```



# Tests Stencil - E2E

```
import { newE2EPage } from '@stencil/core/testing';

describe('hello-world', () => {
 it('renders', async () => {
 const page = await newE2EPage();
 await page.setContent('<hello-world></hello-world>');

 const element = await page.find('hello-world');
 expect(element).toHaveClass('hydrated');
 });
});
```