Les Web Components et l'accessibilité

Paris Web

26 septembre 2025

Intro

Qui sommes-nous ?

Simon Duhem

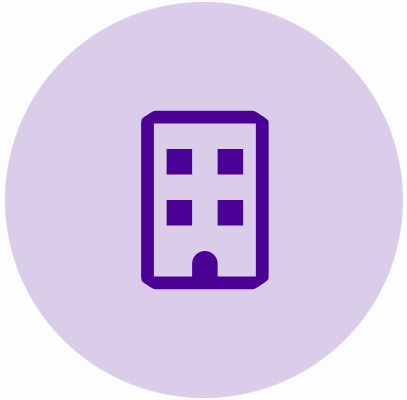
Architecte front-end

Nicolas Jouanno Daniel

Lead front-end

Intro

Que faisons-nous ?



mgdis

Core UI

Design System

mg-components

Les Web Components

Qu'est-ce qu'un composant ?

- Un composant est une brique d'interface autonome.
- Regroupe structure, style et comportement.
- Il facilite la réutilisation et la maintenance du code.

Qu'est-ce que les Web Components ?

Historique des Web Components

- 2011 : Proposés par Google.
- 2014 : Spécifications officielles publiées.
- 2016 : Adoption progressive dans les navigateurs modernes.
- Aujourd'hui : Standard mature, en évolution constante.

Les clés des Web Components

- Un standard web natif pour créer des composants personnalisés et encapsulés.
- Indépendants des frameworks.
- Compatibles avec tous les navigateurs modernes.
- Trois technologies clés :
  - Custom Elements
  - **Shadow DOM**
  - Templates HTML

Custom Elements

```
<paris-web></paris-web>
```

- Permettent de créer ses propres balises HTML avec un comportement spécifique.
- Le nom doit contenir un tiret (`-`).
- S'utilisent comme n'importe quel élément du DOM.
- Doivent être enregistrés via `customElements.define()`.

Shadow DOM

```
<paris-web>
  #shadow-root
  <p>Contenu encapsulé</p>
</paris-web>
```

- Permet d'encapsuler le style et le code d'un composant.
- Crée une frontière entre le contenu interne et le reste du document.
- Les styles définis à l'intérieur ne s'appliquent pas à l'extérieur, et ceux de l'extérieur n'affectent pas l'intérieur.
- Deux modes :
  - `open` : accessible via `.shadowRoot` en JavaScript.
  - `closed` : inaccessible depuis l'extérieur.

Templates

```
<template id="paris-web-template">
  <p>Hello Paris Web!</p>
</template>
```

- Contient du HTML.
- N'est pas affiché.
- On l'insère dans le Shadow DOM via JavaScript.

Exemple de composant

```
<template id="paris-web-template">
  <p>Hello Paris Web!</p>
</template>

<script>
  class ParisWeb extends HTMLElement {
    constructor() {
      super();
      const template = document.getElementById('paris-web-template');
      const content = template.content.cloneNode(true);

      const shadow = this.attachShadow({ mode: 'open' });
      shadow.appendChild(content);
    }
  }

  customElements.define('paris-web', ParisWeb);
</script>

<paris-web></paris-web>
```

Librairies

- Lit
- Stencil

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

@Component({
  tag: 'paris-web',
  shadow: true,
})
export class ParisWeb {
  render(): HTMLElement {
    return <p>Hello Paris Web!</p>;
  }
}
```

Pourquoi les Web Components ?

- Un standard du W3C qui reste stable dans le temps.
- Des composants réutilisables partout, peut importe le framework.

Shadow DOM & Accessibilité : quand l'encapsulation brise les liens ARIA

Le contexte : notre approche atomic design

```
<label for="input">Nom</label>
<input type="text" id="input" aria-describedby="aide erreur" />
<span id="aide">Format attendu</span>
<span id="erreur">Champ requis</span>
```

Les liens ARIA fonctionnent parfaitement

Mais quand on passe aux Web Components...

```
<!-- Maintenant : chaque élément devient un composant -->
<mg-label for="input">Nom</mg-label>
<mg-input id="input" aria-describedby="aide erreur"></mg-input>
<mg-help-text id="aide">Format attendu</mg-help-text>
<mg-message id="erreur">Champ requis</mg-message>
```

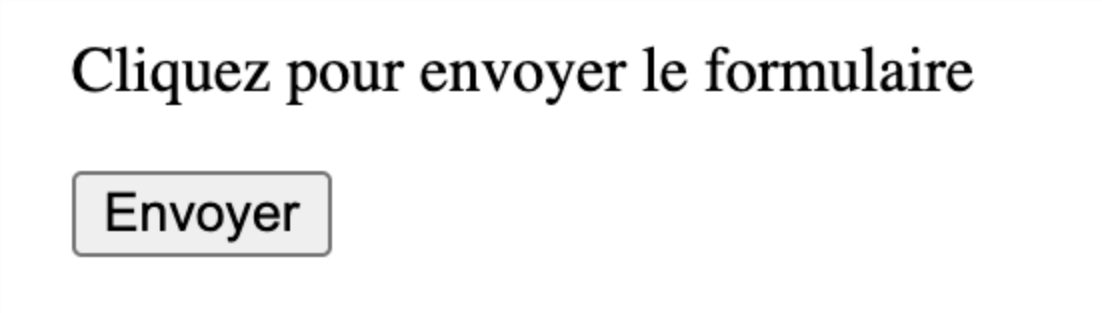
- Logique parfaite... en théorie

Le problème révélé

```
<p id="info">Remplissez le formulaire et cliquez sur le bouton pour envoyer le formulaire</p>
<my-button aria-describedby="info">Envoyer</my-button>
```

✅ VISUEL

Le contenu est bien rendu visuellement



🚫 VoiceOver

```
❌ "Envoyer, bouton"
✅ Attendu : "Envoyer, Instructions importantes, bouton"
```

Le Shadow DOM isole chaque composant !

Pourquoi ça ne marche plus ?

```
<p id="info">Remplissez le formulaire et cliquez sur le bouton pour envoyer le formulaire</p>
<my-button aria-describedby="info">
  #shadow-root (open)
  <button><slot></slot></button>
  "Envoyer"
</my-button>
```

- Le `<p>` porte l' `id="info"`
- L'attribut `aria-describedby="info"` est sur le host de `<my-button>`
- Le focus est sur l'élément `<button>` dans le Shadow DOM

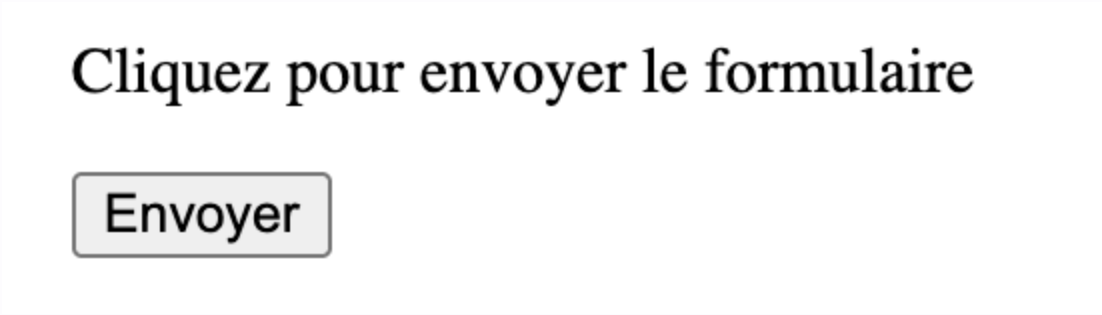
Solution 1 : Encapsuler complètement la logique ARIA

```
<my-button description="Remplissez le formulaire et cliquez sur le bouton pour envoyer le formulaire">Envoyer</my-button>

<template id="mg-button-template">
  <p id="info"></p>
  <button aria-describedby="info">
    <slot></slot>
    <!-- Description interne, cachée visuellement mais lisible par screen readers -->
  </button>
</template>
<script>
  class MyButton extends HTMLElement {
    [...]
    connectedCallback() {
      // Remplit la description avec la prop/attribut "description"
      this.shadowRoot.querySelector("#info").textContent = this.getAttribute("description");
    }
  }
</script>
```

DOM

```
<my-button description="Remplissez le formulaire et cliquez sur le bouton pour envoyer le formulaire">
  #shadow-root (open)
  <p id="info">Remplissez le formulaire et cliquez sur le bouton pour envoyer le formulaire</p>
  <button aria-describedby="info"><slot></slot></button>
  "Envoyer"
</my-button>
```

✅ Résultat : Fonctionne parfaitement




```
✅ VoiceOver : Envoyer, Remplissez le formulaire et cliquez sur le bouton pour envoyer le formulaire, bouton
```

Solution 2 : Le host comme élément ARIA

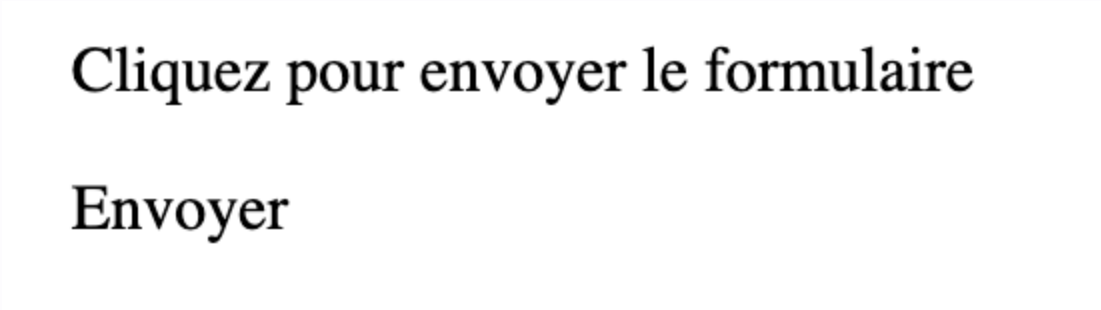
```
<p id="info">Remplissez le formulaire et cliquez sur le bouton pour envoyer le formulaire</p>
<my-button aria-describedby="info">Envoyer</my-button>

<template id="mg-button-template">
  <slot></slot>
</template>
<script>
  class MyButton extends HTMLElement {
    [...]
    connectedCallback() {
      this.setAttribute('role', 'button'); // Le host devient le bouton
      this.setAttribute('tabindex', '0');
      this.addEventListener('keydown', (e) => { // Gestion du clavier
        if (e.key === 'Enter' || e.key === ' ') {
          this.click();
        }
      });
    }
  }
</script>
```

DOM

```
<p id="info">Remplissez le formulaire et cliquez sur le bouton pour envoyer le formulaire</p>
<my-button role="button" aria-describedby="info"> #shadow-root (open) "Envoyer" </my-button>
```

Résultat

- ✅ VoiceOver : Les attributs ARIA fonctionnent (restitution : "Envoyer, Remplissez le formulaire et cliquez sur le bouton pour envoyer le formulaire, bouton")
- ⚠️ Plus de travail pour recréer le comportement natif
- 😬 On perd aussi le rendu natif navigateur



A retenir

Stratégie hybride selon le contexte :

- Composants simples : Privilégier les éléments natifs
- Logique complexe : Encapsuler dans le Shadow DOM
- Toujours : Tester avec les lecteurs d'écran
- Le Shadow DOM n'est pas l'ennemi de l'accessibilité…
  Il nous force juste à être plus rigoureux !

Les Web Components améliorent l'accessibilité

- Centralisation de la logique: corriger une fois = corrigé partout
- Uniformiser le comportement des champs
- Protéger avec l’encapsulation : pas d’effets de bord
- Favoriser l’inclusion 🎯

Les audits

Quand le RGAA est clair

```
<mg-button size="large">Non valide!</mg-button>
```

```
<mg-input-select width="full" items="..."></mg-input-select>
```

> *10.1.2 Dans chaque page web, les attributs servant à la **présentation de l'information** ne doivent pas être présents dans le code source généré des pages. Cette règle est-elle respectée ?*

Quand le RGAA est moins clair

> *7.5 Dans chaque page web, les messages de statut sont-ils correctement restitués par les technologies d'assistance ?*

- Pour l'auditeur, le rôle devait être sur le `<mg-alert>` et non pas dans le Shadow DOM.
- Le cas de test du critère :
  - Retrouver dans le document les messages qui valent pour message de statut.
  - Pour chacun de ces messages, déterminer la nature de l'information dont est porteur le message :
  - Si le message…

Les outils d'audit

- Ne prennent pas en compte le Shadow DOM
  - Nu Html Checker
  - Assistant RGAA
  - Web Developer
  - detectAutocomplete
- Prennent en compte le Shadow DOM :
  - HeadingsMap
  - WCAG Contrast checker

Essentiels mais à faire évoluer

- Utiles pour nos audits
- Indispensables pour gagner du temps
- Mais pas toujours adaptés à certains projets
- Ils doivent évoluer pour mieux prendre en compte les Web components

Conclusion

Conclusion

Pourquoi ne pas adapter le RGAA aux Web Components ?

- Les Web Components sont un standard récent, qui doit coexister avec des spécifications historiques du RGAA.
- Nous avons déposé un dossier pour participer aux prochaines versions du RGAA, afin d'apporter notre expertise sur les Web Components.

Merci

Des questions ?