Introduction à StencilJS : créer des composants Web modernes

Formation Professionnelle

2024 - v1.0.0

Copyright et Licence

Auteur : Noël Macé
Année de création : 2024
Contact : contact@noelmace.com

Votre formateur

Noël Macé

- Développeur / Architecte Senior Fullstack
- Consultant-Formateur expert en développement Web
- **Expérience :**
  - Plus de 15 années dans l'informatique et l'enseignement
  - Focus sur les évolutions des standards et pratiques web
  - Auteur d'un livre sur les architectures web modernes
- **Motivations :**
  - Partager des connaissances et encourager les bonnes pratiques de développement.
  - Promouvoir les technologies ouvertes et respectueuses des utilisateurs.
- **Contact :** contact@noelmace.com

Objectifs de la formation

- Comprendre les fondamentaux de **StencilJS**.
- Apprendre à créer des composants Web modernes et performants.
- Explorer les concepts avancés : gestion de l'état, interactions, routage, formulaires, et Service Workers.
- Comparer StencilJS avec d'autres outils et bibliothèques front-end.

Contexte et déroulement

- **Audience :** Développeurs front-end avec des notions de base en JavaScript et Web Components.
- **Structure :**
  1. Présentation de Stencil.
  2. Premiers pas avec les composants Stencil.
  3. Propriétés, méthodes et gestion de l'état.
  4. Gestion des évènements et interactions utilisateur.
  5. Routage.
  6. Formulaires.
  7. Service Workers.
  8. Alternatives : tour d'horizon des outils front-end modernes.
- **Déroulement :**
  - Présentations théoriques.
  - Démonstrations pratiques.
  - Exercices guidés.
  - Discussions et Q/A.

Présentation de Stencil

Qu'est-ce que StencilJS ?

- Un compilateur moderne pour créer des **Web Components**.
- Conçu par **Ionic Framework**.
- Objectifs principaux :
  - Optimisation des performances.
  - Facilité d'utilisation.
  - Intégration dans n'importe quel projet ou framework (React, Angular, Vue, etc.).

Fonctionnalités clés

1. **Base Web Components** : créez des composants standard.
2. **React-like Syntax** : développeurs familiers avec React apprécieront sa syntaxe JSX.
3. **Performance** : Lazy Loading, réduction du bundle, DOM virtuel.
4. **Interopérabilité** : compatible avec tous les frameworks modernes.
5. **Support TypeScript** : robustesse grâce à TypeScript intégré.

Premiers pas avec Stencil

Installation et configuration

Prérequis

- Node.js (version LTS recommandée).
- Un éditeur comme VS Code.

Créer un projet Stencil**

```
npm init stencil
```

Structure du projet

- `src/components/` : contient vos composants.
- `stencil.config.ts` : configuration du projet.

Propriétés, méthodes et gestion de l'état

Ajouter des propriétés et méthodes

- Utilisez les **decorators** TypeScript pour définir des propriétés :

```
@Prop() title: string;
@State() count: number = 0;
```

Gestion de l'état interne

- Le décorateur `@State` est utilisé pour des variables locales et réactives.
- Exemple :

```
@State() likes: number = 0;

incrementLikes() {
  this.likes += 1;
}
```

Gestion des évènements et interactions utilisateur

Émettre et écouter des évènements

Utilisez le décorateur `@Event()` pour émettre des évènements :

```
@Event() myEvent: EventEmitter<string>;
```

Abonnez-vous à des évènements dans vos composants ou via JavaScript natif :

```
<my-component onMyEvent={(e) => console.log(e.detail)}></my
```

Routage

Pourquoi un routeur dans une application ?

- Permet de gérer la navigation entre différentes pages ou vues.
- Nécessaire pour structurer les applications monopages (SPA).
- Facilite l'organisation et la lisibilité du code.
- Adapté à la programmation web orientée composants.

Histoire de @stencil/router

- Initialement créé par l'équipe de **Ionic Framework**.
- Inspiré des routeurs React et Angular.
- Léger et optimisé pour les Web Components.
- Utilisé dans les projets Ionic mais également indépendant du framework.

Installation du routeur Stencil

1. Installez le package via npm

```
npm install @stencil/router
```

2. Importez et configurez le routeur dans votre projet :

- Ajoutez le module de routeur dans un composant principal (`app-root` par exemple).
- Configurez vos routes à l'aide de `<stencil-router>`.

Exemple de configuration des routes

```
<stencil-router>
  <stencil-route url="/" component="app-home"></stencil-rou
  <stencil-route url="/about" component="app-about"></stenc
</stencil-router>
```

Fonctionnalités de base : Définir et afficher une route

Fonctionnement

1. Chaque **`stencil-route`** correspond à une URL spécifique.
2. Le paramètre `component` indique le composant à charger.
3. Le routeur utilise l'historique du navigateur pour changer de vue.

Exemple simple

```
<stencil-route url="/" component="app-home"></stencil-route
```

Pièges à éviter

1. **Routes non correspondantes :**
   - Si aucune route ne correspond à l'URL actuelle, rien ne sera affiché.
   - Utilisez une route par défaut avec `component` et `exact=false`.
2. **Chargement conditionnel des composants :**
   - Assurez-vous que chaque composant de route est défini avant d'être utilisé.

Fonctionnalités avancées : Navigation dynamique

Utilisation d'un lien pour naviguer

`<stencil-route-link>` permet de naviguer sans recharger la page.

```
<stencil-route-link url="/about">Aller à la page "À propos"
```

Navigation programmée avec JavaScript

Vous pouvez également naviguer via le code en utilisant l'API du routeur.

```
import { Router } from '@stencil/router';

Router.push('/about');
```

Gestion des paramètres de route

Ajouter des paramètres à une route

Les paramètres permettent de transmettre des données via l'URL.

```
<stencil-route url="/user/:id" component="user-profile"></s
```

Lire les paramètres dans un composant

Vous pouvez accéder aux paramètres via `match.params`.

```
@Prop() match: MatchResults;

connectedCallback() {
  console.log(this.match.params.id);
}
```

Résumé

- Le routeur Stencil est simple, léger et conçu pour les Web Components.
- Fonctionnalités principales :
  1. Gestion des URL et navigation.
  2. Support des paramètres dynamiques.
  3. Navigation conditionnelle et programmée.
- Pièges courants :
  - Routes non correspondantes.
  - Mauvaise gestion des composants dynamiques.

Questions avant de passer au prochain sujet ?

Formulaires avec StencilJS

Pourquoi les formulaires sont importants ?

- Permettent de collecter et traiter des données utilisateur.
- Utilisation fréquente dans les applications modernes (authentification, saisie de données).
- Enjeux :
  1. Gestion des champs et des événements associés (validation, soumission).
  2. Performance et accessibilité.
  3. Intégration avec des API ou un backend.

Fonctionnalités de base

Points essentiels

- Les formulaires utilisent les éléments natifs du DOM HTML.
- StencilJS facilite :
  - La gestion des événements (`onInput`, `onSubmit`, etc.).
  - La liaison entre les champs de formulaire et les données de l'état (`@State`).
  - La validation des entrées utilisateur.

Exemple de formulaire simple

```
<form onSubmit={(e) => this.handleSubmit(e)}>
  <input type="text" placeholder="Nom" onInput={(e) => this
  <button type="submit">Envoyer</button>
</form>
```

Gestion des événements des champs de formulaire

Principe

Utiliser les événements natifs pour réagir aux changements de valeur :

- Input
- Change

onInput

Déclenché à chaque modification du champ :

```
handleNameInput(event: InputEvent) {
  const input = event.target as HTMLInputElement;
  this.name = input.value;
  console.log(`Nom saisi : ${this.name}`);
}
```

onChange

Déclenché lorsque la modification est terminée.

Gestion des valeurs via `@State`

- Les données des champs peuvent être stockées dans l'état local.
- Utilisez `@State` pour les rendre réactives.

Exemple avec liaison entre champ et état

```
@State() name: string = '';

render() {
  return (
    <form>
      <input type="text" value={this.name} onInput={(e) =>
      <p>Vous avez saisi : {this.name}</p>
    </form>
  );
}
```

Validation des champs de formulaire

Ajouter des règles de validation HTML

- Utilisez les attributs HTML natifs pour la validation :
  - `required`
  - `pattern`
  - `min`, `max`
  - `maxlength`, `minlength`

Exemple de validation native

```
<input type="email" required placeholder="Entrez votre emai
```

Validation personnalisée avec StencilJS

```
validateEmail(email: string): boolean {
  const regex = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]+
  return regex.test(email);
}

handleEmailInput(event: InputEvent) {
  const email = (event.target as HTMLInputElement).value;
  if (!this.validateEmail(email)) {
    console.error('Email invalide');
  }
}
```

Soumission de formulaire

Gestion de l'événement `onSubmit`

- Interceptez la soumission du formulaire avec `onSubmit`.
- Empêchez le comportement par défaut pour gérer la soumission manuellement.

Exemple

```
handleSubmit(event: Event) {
  event.preventDefault();
  console.log('Formulaire soumis avec succès !');
}
```

Intégration avec une API

Soumettez les données du formulaire à un backend via `fetch`.

```
async handleSubmit(event: Event) {
  event.preventDefault();
  const response = await fetch('https://example.com/api', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ name: this.name, email: this.ema
  });
  const result = await response.json();
  console.log('Réponse API :', result);
}
```

Accessibilité des formulaires

Bonnes pratiques

1. Utiliser des labels associés aux champs

```
<label for="name">Nom</label> <input id="name" type="text"
```

2. Gérer les erreurs

Fournissez des retours visuels (ex. : messages d'erreur, encadrés rouges).

```
@State() errorMessage: string = '';

render() {
  return (
    <form>
      <input type="email" onInput={(e) => this.handleEmail
      {this.errorMessage && <p class="error">{this.errorMes
    </form>
  );
}
```

Résumé du chapitre “Formulaires”

- **Champs natifs et réactifs :**
  - Utilisez les événements `onInput` et l'état local `@State`.
- **Validation :**
  - Combinez validations natives et personnalisées.
- **Soumission et intégration :**
  - Gérez `onSubmit` et utilisez `fetch` pour interagir avec des API.
- **Accessibilité :**
  - Utilisez des labels et fournissez des retours d'erreur visuels.

Questions ou démonstrations spécifiques avant de continuer ?

Service Workers

Cf. ma conférence PRPL pour des compléments :

- *support*
- *archive*

Définition

- Les **Service Workers** sont des scripts qui s'exécutent en arrière-plan, indépendamment de la page web.
- Ils permettent de gérer des fonctionnalités avancées comme :
  1. La mise en cache des ressources.
  2. Les notifications push.
  3. La synchronisation en arrière-plan.

Avantages

1. **Performance :**
   - Les ressources sont chargées depuis le cache, accélérant les temps de chargement.
2. **Expérience utilisateur :**
   - Fonctionnement hors ligne grâce à la mise en cache.
3. **Fiabilité :**
   - Les ressources critiques peuvent être disponibles même en cas de perte de connexion.
4. **Progressive Web Apps (PWA) :**
   - Indispensables pour transformer une application en PWA.

Fonctionnement

Cycle de vie

1. **Installation :**
   - Enregistre le Service Worker dans le navigateur.
2. **Activation :**
   - Met en place le cache et les événements.
3. **Interception des requêtes :**
   - Permet de servir des ressources depuis le cache.

Exemple

```
# Exemple de cycle de vie d'un Service Worker
self.addEventListener('install', (event) => {
  console.log('Service Worker installé.');
});

self.addEventListener('activate', (event) => {
  console.log('Service Worker activé.');
});

self.addEventListener('fetch', (event) => {
  console.log('Requête interceptée :', event.request.url);
});
```

Construire un Service Worker pas-à-pas

Étape 1 : Créer un Service Worker manuellement

1. Créez un fichier `service-worker.js` à la racine du projet.
2. Configurez les événements `install`, `activate` et `fetch`.

Exemple complet

```
# Exemple de Service Worker minimal
const CACHE_NAME = 'my-cache-v1';
const ASSETS = ['index.html', 'styles.css', 'app.js'];

self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME).then((cache) => {
      console.log('Caching des ressources.');
      return cache.addAll(ASSETS);
    })
  );
});

self.addEventListener('activate', (event) => {
  event.waitUntil(
```

Étape 2 : Enregistrer le Service Worker dans votre projet

Enregistrez le Service Worker dans votre application via `navigator.serviceWorker`:

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker
    .register('/service-worker.js')
    .then(() => console.log('Service Worker enregistré.'))
    .catch((error) => console.error("Erreur lors de l'enreg
}
```

Utiliser Workbox pour simplifier la gestion des Service Workers

Pourquoi Workbox ?

- **Workbox** est une bibliothèque de Google qui simplifie l'écriture et la gestion des Service Workers.
- Fonctionnalités clés :
  1. Prise en charge des stratégies de cache.
  2. Gestion simplifiée des ressources statiques et dynamiques.
  3. Génération automatique des fichiers de cache.

Étape 1 : Installer Workbox

Installez Workbox dans votre projet avec npm :

```
npm install workbox-cli --save-dev
```

Étape 2 : Configurer Workbox

1. Ajoutez un fichier de configuration `workbox-config.js` à la racine du projet.
2. Définissez les ressources à mettre en cache.

```
# Exemple de configuration Workbox
module.exports = {
  globDirectory: './',
  globPatterns: ['**/*.{html,js,css}'],
  swDest: 'service-worker.js',
  runtimeCaching: [
    {
      urlPattern: ({ url }) => url.origin === self.location
      handler: 'CacheFirst',
      options: {
        cacheName: 'static-resources',
        expiration: {
          maxEntries: 50,
        },
      },
    },
  ],
};
```

Étape 3 : Générer le Service Worker avec Workbox

```
npx workbox generateSW
```

Pièges courants

1. **Cache obsolète :**
   - Les utilisateurs peuvent voir une ancienne version de votre application si le Service Worker n'est pas mis à jour correctement.
   - Solution : invalidez les anciens caches lors de l'activation.
2. **Erreurs silencieuses :**
   - Le Service Worker peut échouer sans que vous le remarquiez.
   - Solution : ajoutez des logs et surveillez les événements.
3. **Support limité des navigateurs :**
   - Les Service Workers ne fonctionnent pas sur les très vieux navigateurs.
   - Solution : vérifiez la compatibilité avec `if ('serviceWorker' in navigator)`.

Stratégies

Définition

Une stratégie réseau définit comment un Service Worker gère les requêtes :

1. **Prise en charge des ressources en ligne (online).**
2. **Mise en cache des ressources (offline).**
3. **Combinaison des deux.**

Principales stratégies

1. Cache First (Cache prioritaire)
2. Network First (Réseau prioritaire)
3. Stale While Revalidate (*“Consomme et revalide” / “Sert puis met à jour”*) :
4. Pre-caching (Pré-mise en cache)

1. Cache First (Cache prioritaire)

- Cherche d'abord la ressource dans le cache.
- Si elle n'existe pas, la télécharge depuis le réseau.

Pros & Cons

- **Avantage :**
  - Réduit la latence en évitant les requêtes réseau.
- **Inconvénient :**
  - Les ressources peuvent devenir obsolètes.

Exemple Vanilla

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      return response || fetch(event.request);
    })
  );
});
```

Exemple Workbox

```
workbox.routing.registerRoute(
  ({ request }) => request.destination === 'script',
  new workbox.strategies.CacheFirst({
    cacheName: 'scripts-cache',
    plugins: [
      new workbox.expiration.ExpirationPlugin({
        maxEntries: 50,
        maxAgeSeconds: 30 * 24 * 60 * 60, // 30 jours
      }),
    ],
  })
);
```

2. Network First (Réseau prioritaire)

- **Avantage :**
  - Retourne toujours les ressources les plus à jour (si le réseau est disponible).
- **Inconvénient :**
  - Plus lent, dépend fortement du réseau.

Exemple Vanilla

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    fetch(event.request)
      .then((response) => {
        return caches.open('dynamic-cache').then((cache) =>
          cache.put(event.request, response.clone());
          return response;
        });
      })
      .catch(() => caches.match(event.request))
  );
});
```

Exemple Workbox

```
workbox.routing.registerRoute(
  ({ request }) => request.destination === 'document',
  new workbox.strategies.NetworkFirst({
    cacheName: 'pages-cache',
    plugins: [
      new workbox.expiration.ExpirationPlugin({
        maxEntries: 10,
        maxAgeSeconds: 7 * 24 * 60 * 60, // 7 jours
      }),
    ],
  })
);
```

3. Stale While Revalidate

- **Avantage :**
  - Offre une expérience utilisateur rapide.
  - Met à jour les ressources en arrière-plan.
- **Inconvénient :**
  - Peut servir des données obsolètes.

Exemple Vanilla

```
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.open('dynamic-cache').then((cache) => {
      return cache.match(event.request).then((response) =>
        const fetchPromise = fetch(event.request).then((net
          cache.put(event.request, networkResponse.clone()
          return networkResponse;
        });
        return response || fetchPromise;
      });
    })
  );
});
```

Exemple Workbox

```
workbox.routing.registerRoute(
  ({ request }) => request.destination === 'image',
  new workbox.strategies.StaleWhileRevalidate({
    cacheName: 'images-cache',
    plugins: [
      new workbox.expiration.ExpirationPlugin({
        maxEntries: 100,
        maxAgeSeconds: 30 * 24 * 60 * 60, // 30 jours
      }),
    ],
  })
);
```

Pre-caching (Pré-mise en cache)

- **Avantage :**
  - Assure que les ressources critiques sont disponibles hors ligne.
- **Inconvénient :**
  - Nécessite une liste prédéfinie de ressources.

Exemple Vanilla

```
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('precache').then((cache) => {
      return cache.addAll(['/index.html', '/styles.css', '/
    })
  );
});
```

Exemple Workbox

```
workbox.precaching.precacheAndRoute([
  { url: '/index.html', revision: '123456' },
  { url: '/styles.css', revision: '123456' },
  { url: '/app.js', revision: '123456' },
]);
```

Résumé des stratégies réseau

1. **Cache First :**
   - Priorise le cache pour la vitesse.
   - Utilisez-le pour les ressources rarement mises à jour (images, scripts).
2. **Network First :**
   - Priorise le réseau pour des données à jour.
   - Utile pour des pages dynamiques.
3. **Stale While Revalidate :**
   - Combine rapidité et actualisation en arrière-plan.
   - Adapté pour les données qui évoluent lentement.
4. **Pre-caching :**
   - Prérequis pour les Progressive Web Apps (PWA).
   - Idéal pour des ressources critiques (index.html, CSS global).

Questions ou retours sur ces stratégies ?

Résumé du chapitre “Service Workers”

1. Créer et enregistrer un Service Worker en vanilla js
2. Gérer les événements `install`, `activate` et `fetch`.
3. Introduction à Workbox
   - Simplifie la gestion du cache et les stratégies avancées.
   - Génération automatique des fichiers.
4. Stratégies
   - 4.1 Cache First
   - 4.2 Network First
   - 4.3 Stale While Revalidate
   - 4.4 Pre-caching

Questions ou clarifications avant de continuer ?

Alternatives

- Il n'existe pas d'outil de développement Web "ultime".
- Bien que StencilJS soit un outil puissant pour créer des Web Components, il existe d'autres bibliothèques ou frameworks adaptés à des besoins variés.
- Comprendre leurs avantages et inconvénients permet de mieux choisir l'outil selon :
  1. La complexité du projet.
  2. Les besoins en performances.
  3. Les connaissances de l'équipe.

Lit

- Lit est une bibliothèque légère développée par Google pour créer des Web Components.
- Basée sur les standards natifs du DOM et des templates HTML.

Avantages de Lit

1. **Léger et rapide :**
   - Taille de base de ~5KB.
   - Syntaxe simple et claire.
2. **Interopérabilité :**
   - Fonctionne avec tous les frameworks modernes.
3. **Réactivité intégrée :**
   - Mise à jour efficace du DOM grâce à des propriétés réactives.

Exemple avec Lit : Composant simple

- Déclaration d'un composant avec `@customElement`.
- Utilisation des propriétés réactives avec `@property`.

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

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

  static styles = css`
    p {
      color: blue;
    }
  `;

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

Svelte

- Framework innovant basé sur une approche de compilation.
- Contrairement à React ou Angular, Svelte génère du code JavaScript optimisé, sans virtual DOM.

Avantages de Svelte

1. **Performance :**
   - Les composants sont compilés en JavaScript pur.
2. **Syntaxe intuitive :**
   - Moins de boilerplate, code facile à lire et écrire.
3. **Flexibilité :**
   - Convient aussi bien pour des petits widgets que des applications complexes.

Exemple avec Svelte : Composant simple

- Déclaration des données avec une syntaxe réactive.
- Affichage dynamique basé sur les données.

```
<script>
  let name = 'World';
</script>

<style>
  p {
    color: green;
  }
</style>

<p>Hello, {name}!</p>
<input bind:value={name} placeholder="Votre nom" />
```

SolidJS

- SolidJS est un framework moderne basé sur une **approche réactive** sans virtual DOM.
- Utilise une compilation optimisée pour transformer les composants en JavaScript natif.

Avantages de SolidJS

1. **Performance extrême :**
   - Pas de virtual DOM, chaque changement est directement appliqué au DOM.
2. **Syntaxe JSX :**
   - Familiarité pour les développeurs habitués à React.
3. **Interopérabilité :**
   - Peut intégrer des Web Components natifs.

Compatibilité avec les Web Components

- **Prise en charge directe :**
  - Peut consommer des Web Components comme n'importe quel élément HTML.
  - Possibilité d'exporter des composants en tant que Web Components.

Exemple avec SolidJS : Consommer un Web Component

Intégration d'un Web Component standard dans SolidJS.

```
import { render } from 'solid-js/web';

function App() {
  return (
    <div>
      <my-component name="SolidJS"></my-component>
    </div>
  );
}

render(() => <App />, document.getElementById('root'));
```

Preact

- Une alternative légère à React (~3KB), avec une API compatible.
- Recommandé pour des projets nécessitant une taille minimale.

Avantages de Preact

1. **Léger et rapide :**
   - Taille très réduite, idéal pour les projets où la performance est critique.
2. **Compatibilité React :**
   - Peut réutiliser les bibliothèques et composants React existants.

Compatibilité avec les Web Components

- **Interopérabilité moyenne :**
  - Preact peut consommer des Web Components, mais nécessite parfois un wrapper pour la gestion des événements.

Exemple avec Preact : Utiliser un Web Component

Utilisation d'un Web Component natif dans une application Preact.

```
import { h, render } from 'preact';

function App() {
  return (
    <div>
      <my-component name="Preact"></my-component>
    </div>
  );
}

render(<App />, document.getElementById('root'));
```

React, Next.js et Remix

- **Next.js** et **Remix** sont des frameworks full-stack modernes basés sur React.
- Conçus pour le rendu côté serveur (SSR) et les performances globales.

Avantages

1. **Rendu côté serveur :**
   - Améliore le SEO et les performances initiales.
2. **Routage intégré :**
   - Gestion simplifiée des routes et des pages.
3. **Écosystème React :**
   - Compatibles avec les bibliothèques et Web Components React.

Compatibilité avec les Web Components

- **Interopérabilité indirecte :**
  - Nécessite des ajustements pour intégrer des Web Components natifs.
  - React reste au cœur de l'interaction.

Angular

- Framework complet pour des applications complexes.
- Conçu pour des projets d'entreprise avec des besoins structurés.

Avantages de Angular

1. **Architecture robuste :**
   - Basé sur des modules et services.
2. **Typescript natif :**
   - Utilisé par défaut pour une meilleure maintenabilité.
3. **Interopérabilité avec les Web Components :**
   - Intègre facilement les Web Components via `CUSTOM_ELEMENTS_SCHEMA`.

Exemple avec Angular : Intégrer un Web Component

Déclaration d'un schéma pour consommer des Web Components.

```
import { NgModule, CUSTOM_ELEMENTS_SCHEMA } from '@angular/

@NgModule({
  declarations: [
    /* vos composants */
  ],
  schemas: [CUSTOM_ELEMENTS_SCHEMA],
})
export class AppModule {}
```

Vue.js

- Framework “progressif” pour le développement d’interfaces utilisateur.
- Connu pour sa simplicité et son efficacité.

Avantages de Vue.js

1. **Facilité d'apprentissage :**
   - Syntaxe intuitive et bien documentée.
2. **Écosystème riche :**
   - Nombreuses extensions et outils communautaires.
3. **Compatibilité avec les Web Components :**
   - Peut consommer directement des Web Components natifs.

Exemple avec Vue.js : Utiliser un Web Component

Intégration d'un Web Component natif dans un projet Vue.

```
<template>
  <my-component name="Vue.js"></my-component>
</template>

<script>
export default {
  name: "App",
};
</script>
```

Synthèse

Framework/Librairie	Prise en charge native	Export de composants en WC	Notes
StencilJS			Conçu spécifiquement pour les Web Components.
Lit			100% basé sur les standards du DOM, dont les Web Components.
Angular			Intègre directement les Web Components depuis sa conception.
Vue.js			Excellente compatibilité.
Svelte			Approche unique de compilation.
Preact			Bonne compatibilité avec des ajustements.
React /			Compatibilité tardive, peut

Résumé du chapitre “Alternatives”

- **SolidJS** et **Lit** offrent des performances exceptionnelles en se rapprochant au maximum des derniers standards et innovations
- **Angular et Vue.js** ont une bonne intégration directe des Web Components, ayant dés le départ adopté une architecture orientée composant proche des standards
- **React, Next.js, et Remix** sont puissants mais nécessitent des ajustements pour interagir avec les Web Components natifs (et les standards du web en général)
- **Preact** est une bonne alternative minimaliste à React, idéale pour des projets à faible empreinte et des équipes orientées programmation réactive.

Questions ou points à approfondir ?

Conclusion de la formation : Web Components et StencilJS

Récapitulatif des notions abordées

StencilJS

- Création de composants modernes avec TypeScript et JSX.
- Optimisation pour les performances grâce au lazy loading et au DOM virtuel.

Fonctionnalités avancées

- Routage avec `@stencil/router`.
- Gestion des formulaires et validation.
- Service Workers

Alternatives

Vue d'ensemble des outils comme Lit, Svelte, React, Angular, et leur compatibilité avec les Web Components natifs.

Points forts des Web Components

Pourquoi continuer à les utiliser ?

- **Interopérabilité universelle :**
  - Fonctionnent indépendamment du framework utilisé.
- **Encapsulation native :**
  - Isolation des styles et des comportements.
- **Standardisation :**
  - Fondés sur des standards ouverts reconnus par tous les navigateurs modernes.

Limites et solutions

1. **Courbe d'apprentissage :**
   - Solution : s'appuyer sur des outils comme Stencil ou Lit pour simplifier la création de composants.
2. **Support des navigateurs :**
   - Les Web Components sont supportés dans les navigateurs modernes, mais certains anciens navigateurs nécessitent des polyfills.
3. **Performance des applications massives :**
   - Solution : utiliser des frameworks comme Svelte ou React pour optimiser les interactions complexes tout en utilisant des Web Components pour les éléments réutilisables.

Perspectives et bonnes pratiques

1. Créez (ou non) votre propre bibliothèque
2. Favorisez (ou non) les standards
3. Collaborez (absolument) avec votre équipe
4. Adoptez une approche modulaire

Ressources complémentaires

- **Documentation officielle :**
  - StencilJS
  - Lit
  - MDN Web Components
- **Tutoriels pratiques :**
  - Création de Web Components avec Stencil.
  - Utilisation des Service Workers pour les PWA.
  - Intégration de Web Components dans des frameworks modernes.
- **Outils recommandés :**
  - Storybook : pour documenter et tester vos composants.
  - Workbox : pour simplifier les Service Workers.

Merci pour votre attention !

Questions ?

N'hésitez pas à partager vos projets ou demander des clarifications.