Cahier de corrections d'exercice - Introduction à StencilJS

> Ce document n'est qu'un premier brouillon en attente d'une relecture complète et de tests en situation réelle. Merci de me signaler toute erreur, incohérence ou manque de clarté.

Exercice 1 : Installation et Configuration de StencilJS

Étapes

1. Vous avez créé un projet Stencil en utilisant la commande `npm init stencil`.
2. Vous avez initialisé le projet avec les options par défaut ou personnalisées.
3. Une fois le projet configuré, vous avez pu l'ouvrir dans votre éditeur (VS Code recommandé).
4. Vous avez démarré l'application avec `npm start`, et vérifié que l'application était accessible sur votre navigateur via le `localhost`.

Questions

Quelles sont les principales répercussions de l'utilisation de StencilJS dans un projet par rapport à un framework classique ?

Exercice 2 : Créer un Web Component simple avec Stencil

Étapes

1. Vous avez créé un composant `my-greeting` avec une propriété `name`.
2. Le composant affiche "Hello, [name]" en utilisant la syntaxe JSX de Stencil.

Code

```
// Exemple de composant Stencil avec une propriété `name`
import { Component, Prop } from '@stencil/core';

@Component({
  tag: 'my-greeting',
  styleUrl: 'my-greeting.css',
  shadow: true,
})
export class MyGreeting {
  @Prop() name: string;

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

```
  }
}
```

Questions

Quelles sont les différences entre `@Prop` et `@State` dans la gestion des données au sein d'un composant ?

- Le décorateur `@Prop()` permet de créer des propriétés passées depuis l'extérieur du composant.
- Le décorateur `@State()` aurait pu être utilisé si nous voulions avoir un état interne réactif.

Exercice 3 : Ajouter un Service Worker à votre application

1. Vous avez créé un fichier `service-worker.js` et l'avez enregistré dans votre application via `navigator.serviceWorker.register('/service-worker.js')`.
2. Vous avez mis en cache les ressources de base comme `index.html` et `app.js` dans le Service Worker.

Questions

Expliquez comment un Service Worker peut améliorer la performance d'une application web.

- Un **Service Worker** est un script qui s'exécute en arrière-plan dans le navigateur, indépendant des pages, ce qui lui permet de gérer des fonctionnalités comme la mise en cache des ressources et la gestion des requêtes réseau.
- L'enregistrement du Service Worker dans votre application permet à celle-ci de fonctionner hors ligne en servant les ressources mises en cache lorsqu'Internet est inaccessible.

Code

```
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('my-cache').then((cache) => {
      return cache.addAll(['index.html', 'app.js']);
    })
  );
});

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

```
    })
  );
});
```

- `install` : Lors de l'installation, les ressources de base (`index.html` et `app.js`) sont mises en cache pour être accessibles hors ligne. Cette étape garantit que les ressources essentielles sont disponibles dès le début.
- `fetch` : Lors de la réception des requêtes réseau, le Service Worker vérifie d'abord si la ressource est déjà dans le cache (via `caches.match()`). Si elle y est, il la retourne ; sinon, il va chercher la ressource sur le réseau avec `fetch()`.

```
// Exemple d'enregistrement du Service Worker dans le projet Stencil
if ('serviceWorker' in navigator) {
  navigator.serviceWorker
    .register('/service-worker.js')
    .then(() => console.log('Service Worker enregistré avec succès.'))
    .catch((error) => console.error("Erreur lors de l'enregistrement du Service Worker :", e
}
```

- Le code ci-dessus enregistre le Service Worker dans l'application. Si l'enregistrement est réussi, un message est loggé dans la console, indiquant que le Service Worker a bien été activé.
- L'enregistrement se fait avec la méthode `navigator.serviceWorker.register()`, qui prend en paramètre l'URL du fichier `service-worker.js`.

Exercice 4 : Validation des formulaires

Étapes

1. Vous avez créé un formulaire avec les champs `email` et `password`.
2. Vous avez implémenté une validation côté client pour vérifier que les champs ne sont pas vides et que l'email est valide.

Questions

> Quelle est l'importance de valider les formulaires côté client et côté serveur ?

La validation des formulaires côté client permet de garantir que les données envoyées au serveur sont correctes avant même leur transmission.

- Il est important de valider les champs de formulaire pour s'assurer que l'utilisateur a bien rempli les informations demandées, tout en améliorant l'expérience utilisateur.
- Dans ce cas, nous allons vérifier que l'email est valide avec une expression régulière et que le mot de passe n'est pas vide.

Code

```
// Exemple de validation d'email dans un formulaire
validateEmail(email: string): boolean {
  const regex = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
  return regex.test(email);
}

handleSubmit(event: Event) {
  event.preventDefault();
  if (!this.validateEmail(this.email)) {
    this.errorMessage = 'Email invalide.';
  } else {
    this.errorMessage = '';
    // Soumettre le formulaire
  }
}
```

- `validateEmail()` : La fonction utilise une expression régulière pour vérifier si l'email a le format correct. Cette méthode retourne `true` si l'email est valide et `false` sinon.
- `handleSubmit()` : Lorsque le formulaire est soumis, cette fonction vérifie si l'email est valide. Si l'email est invalide, un message d'erreur est affiché. Sinon, l'erreur est réinitialisée et le formulaire peut être soumis.

Ajout d'un message d'erreur et validation du mot de passe

```
// Validation du mot de passe et gestion des erreurs
handlePasswordValidation(password: string): boolean {
  return password.length >= 6; // Mot de passe doit avoir au moins 6 caractères
}

handleSubmit(event: Event) {
  event.preventDefault();
  if (!this.validateEmail(this.email)) {
    this.errorMessage = 'Email invalide.';
  } else if (!this.handlePasswordValidation(this.password)) {
    this.errorMessage = 'Le mot de passe doit comporter au moins 6 caractères.';
  } else {
    this.errorMessage = '';
    // Soumettre le formulaire
  }
}
```

- **Validation du mot de passe** : La fonction `handlePasswordValidation()` vérifie que le mot de passe contient au moins 6 caractères. Cela garantit que l'utilisateur choisit un mot de passe suffisamment long pour être

sécurisé.
- **Gestion des erreurs** : Les messages d'erreur sont affichés lorsque l'email ou le mot de passe ne respecte pas les conditions définies. Cela améliore l'interaction avec l'utilisateur et l'empêche de soumettre des données incorrectes.

Exercice 5 : Gestion du routage avec Stencil

Étapes

1. Vous avez installé `@stencil/router` et configuré les routes pour l'accueil et la page "À propos".

Code

```
// Exemple de configuration du routage dans Stencil
import { Router, Route } from '@stencil/router';

<Router>
  <Route url="/" component="app-home" />
  <Route url="/about" component="app-about" />
</Router>
```

- Le composant stencil-router gère la navigation entre les pages de l'application.
- Chaque `<Route>` représente une page accessible à partir d'une URL spécifique.
- La structure de base du routage consiste à définir une route pour chaque vue ou page de l'application, puis à y associer le composant à rendre lorsque l'utilisateur navigue vers cette route.

Exemple de structure de projet

- `/src/components/app-home.tsx` : Composant pour la page d'accueil.
- `/src/components/app-about.tsx` : Composant pour la page "À propos".
  - Le routeur stencil-router se charge de rendre le bon composant en fonction de l'URL de la page.

Questions

Quels sont les avantages de l'utilisation d'un routeur dans une application monopage (SPA) ?

- Un routeur dans une application monopage (SPA) permet de maintenir l'état de l'application tout en changeant le contenu visible pour l'utilisateur. Cela évite de recharger entièrement la page à chaque navigation, ce qui améliore les performances et l'expérience utilisateur.

Exercice 6 : Service Worker avec Workbox

Etapes

1. Vous avez installé **Workbox** dans votre projet Stencil via npm.
2. Vous avez ajouté un Service Worker qui met en cache les ressources statiques avec la stratégie `CacheFirst`.
3. Vous avez testé le fonctionnement du cache en accédant à votre application sans connexion réseau.

Questions

Comment **Workbox** simplifie-t-il la gestion des Service Workers et du cache ?

- **Workbox** est une bibliothèque qui simplifie l'utilisation des Service Workers en fournissant des stratégies de mise en cache prêtes à l'emploi.
- La stratégie `CacheFirst` permet de vérifier d'abord si la ressource est présente dans le cache, et de la retourner si elle y est, sinon elle est téléchargée depuis le réseau et mise en cache.
- Cette stratégie est particulièrement utile pour les ressources statiques comme les images, les fichiers CSS ou JavaScript qui ne changent pas fréquemment.

Code

```
// Exemple d'utilisation de Workbox pour la stratégie CacheFirst
workbox.routing.registerRoute(
  ({ request }) => request.destination === 'image',
  new workbox.strategies.CacheFirst({
    cacheName: 'images-cache',
    plugins: [
      new workbox.expiration.ExpirationPlugin({
        maxEntries: 100,
        maxAgeSeconds: 30 * 24 * 60 * 60, // 30 jours
      }),
    ],
  })
);
```

- `workbox.routing.registerRoute()` : Cette méthode permet de définir la stratégie de mise en cache pour des ressources spécifiques. Dans cet exemple, la stratégie `CacheFirst` est appliquée aux images.
- `request.destination === 'image'` : Ce test permet de cibler uniquement les requêtes pour les images. Vous pouvez ajouter d'autres conditions pour gérer différents types de ressources.
- `CacheFirst` : Cette stratégie priorise le cache. Si la ressource est présente dans le cache, elle est immédiatement renvoyée. Sinon, elle est récupérée

via le réseau et ajoutée au cache pour de futures utilisations.

- `plugins` : Ce plugin permet de limiter la durée de vie des ressources en cache. Ici, les ressources expirent après 30 jours ou une fois que 100 entrées sont atteintes dans le cache.

```
// Exemple d'enregistrement du Service Worker avec Workbox
if ('serviceWorker' in navigator) {
  navigator.serviceWorker
    .register('/service-worker.js')
    .then(() => console.log('Service Worker enregistré avec succès.'))
    .catch((error) => console.error('Erreur lors de l\'enregistrement du Service Worker :',
}
```

- `Enregistrement du Service Worker` : Cette étape permet d'enregistrer le fichier `service-worker.js` qui contient la logique du Service Worker. Lorsque l'enregistrement est réussi, le Service Worker prend le contrôle de l'application.
- Si le navigateur prend en charge les Service Workers (via `navigator.serviceWorker`=, il sera enregistré et pourra commencer à intercepter les requêtes réseau et gérer le cache en arrière-plan.