Guide complet de correction des erreurs TypeScript et ESLint

Ce document présente les erreurs communes rencontrées dans le projet MetaSign et propose des solutions optimales pour les résoudre, en suivant les meilleures pratiques de développement.

1. Imports inutilisés

Erreur

```
'NextRequest' is defined but never used
```

Fichiers concernés:

- `src/app/api/notifications/mark-all-read/route.ts`
- `src/app/api/notifications/route.ts`

Solution

Supprimer complètement l'import inutilisé:

```typescript
// Avant
import { NextResponse, NextRequest } from 'next/server';

// Après
import { NextResponse } from 'next/server';
```

Explication

Lorsqu'un type ou une classe est importé mais jamais utilisé, TypeScript et ESLint signaleront cette importation comme inutile. Au lieu d'utiliser des techniques comme le préfixe underscore, il est préférable de simplement supprimer l'import non utilisé pour garder le code propre.

2. Paramètres non utilisés dans les fonctions

Erreur

```
'request' is defined but never used
```

Fichiers concernés:

- `src/app/api/notifications/mark-all-read/route.ts`
- `src/app/api/notifications/route.ts`

Solution

Supprimer complètement le paramètre:

```typescript
// Avant
export async function GET(request: NextRequest) { ... }

// Après
export async function GET(): Promise<NextResponse> { ... }
```

Explication

Dans les routes API Next.js, si vous n'avez pas besoin du paramètre `request`, vous pouvez simplement l'omettre de la signature de la fonction. Next.js ne vérifie pas strictement si le paramètre est présent, mais uniquement que le nom de la fonction corresponde à une méthode HTTP (GET, POST, etc.).

3. Problèmes avec `exactOptionalPropertyTypes: true`

Erreur

```
L'argument de type '{ status: number | undefined; }' n'est pas assignable au paramètre de type 'ResponseInit'
avec 'exactOptionalPropertyTypes : true'. Pensez à ajouter 'undefined' aux types des propriétés de la cible.
```

Fichiers concernés:

- `src/utils/api-helpers.ts`

Solution

Assurer qu'une valeur non-undefined est toujours utilisée:

typescript

```typescript
// Avant
return NextResponse.json(
  { success: true, data: result },
  { status: mergedOptions.successStatusCode }
);

// Après
const statusCode = mergedOptions.successStatusCode ?? 200;
return NextResponse.json(
  { success: true, data: result },
  { status: statusCode }
);
```

Explication

Avec l'option `exactOptionalPropertyTypes: true` dans TypeScript, il fait une distinction stricte entre une propriété optionnelle (qui peut être absente) et une propriété qui peut être `undefined`. En utilisant l'opérateur de coalescence nulle (`??`), nous garantissons qu'une valeur non-undefined est toujours fournie.

4. Dépendances manquantes ou complexes dans les hooks React

Erreur

```
React Hook useCallback has a missing dependency: 'fetchOptions'.
Either include it or remove the dependency array.


React Hook useEffect has a spread element in its dependency array.
This means we can't statically verify whether you've passed the correct dependencies.
```

Fichiers concernés:

- `src/hooks/useApi.ts`

Solution

Utiliser useRef pour stocker fetchOptions de manière stable:

```typescript
// Stocker fetchOptions dans un useRef pour éviter les changements de référence
const fetchOptionsRef = useRef(fetchOptions);

// Mettre à jour la référence si fetchOptions change
useEffect(() => {
  fetchOptionsRef.current = fetchOptions;
}, [fetchOptions]);

// Utiliser la référence dans les fonctions
const execute = useCallback(
  async (executeOptions?: ApiOptions) => {
    // ...
    const result = await fetchApi<T>(url, {
      ...fetchOptionsRef.current,
      ...executeOptions
    });
    // ...
  },
  [url, checkCache, transform, cacheResult]
);
```

Explication

Les objets comme `fetchOptions` créent une nouvelle référence à chaque rendu, ce qui peut provoquer des boucles infinies s'ils sont inclus dans les dépendances des hooks React. En utilisant `useRef`, nous stockons une référence stable à l'objet et nous pouvons la mettre à jour quand nécessaire sans déclencher de re-renders inutiles.

Pour les tableaux comme `dependencies` qui sont passés avec le spread operator dans les dépendances d'un hook, vous pouvez soit:

1. Créer une variable intermédiaire (`const depsArray = [...dependencies]`)
2. Utiliser un commentaire ESLint pour désactiver l'avertissement quand c'est inévitable

5. Erreurs communes liées aux types strictes

Erreur avec le type `any`

```
Unexpected any. Specify a different type.
```

Solution

Remplacer `any` par `unknown` ou un type plus spécifique:

```typescript
// Avant
function transform(data: any): T { ... }

// Après
function transform(data: unknown): T { ... }
```

Explication

Le type `unknown` est une alternative plus sûre à `any` car il force à vérifier le type avant de manipuler la valeur, tout en gardant la même flexibilité.

Meilleures pratiques pour éviter ces erreurs

1. **Pour les imports:**
   - Utiliser des outils comme ESLint avec auto-fix pour nettoyer automatiquement les imports
   - Configurer l'éditeur pour organiser les imports automatiquement lors de la sauvegarde
2. **Pour les paramètres non utilisés:**
   - Ne pas inclure le paramètre dans la signature s'il n'est pas utilisé
   - Pour les APIs avec une signature fixe, créer des wrappers qui omettent les paramètres inutiles
3. **Pour exactOptionalPropertyTypes:**
   - Toujours utiliser l'opérateur de coalescence nulle (`??`) pour fournir une valeur par défaut
   - Éviter de passer directement des propriétés optionnelles aux fonctions qui attendent une valeur définie
4. **Pour les hooks React:**
   - Utiliser `useRef` pour les objets qui changent fréquemment mais ne doivent pas déclencher de re-renders
   - Éviter d'utiliser des objets ou tableaux directement dans les dépendances des hooks
   - Mémoriser les valeurs complexes avec `useMemo` avant de les utiliser dans les dépendances
   - Pour les tableaux dans les dépendances, les déclarer en dehors de l'effet ou du callback
5. **Pour le typage strict:**
   - Favoriser `unknown` plutôt que `any` quand le type exact n'est pas connu
   - Utiliser des types d'utilitaires comme `Partial<T>`, `Required<T>`, `Pick<T, K>` pour manipuler les types
   - Créer des types d'utilitaires personnalisés pour les cas spécifiques à votre application

L'adoption de ces pratiques contribuera à un code plus robuste, plus sûr et plus facile à maintenir, avec moins d'erreurs lors de la compilation et de l'exécution.