



343



88



632



Sam Magura

Posté sur 9 avr. • Mis à jour le 15 avr.

Mauvaises habitudes des développeurs React de niveau intermédiaire

#réagir #javascript #manuscrit

Si vous êtes un développeur React de niveau intermédiaire cherchant à devenir un développeur React avancé, ce poste est pour vous !

J'examine quotidiennement le code React écrit par des développeurs juniors et de niveau intermédiaire depuis quelques années maintenant, et cet article couvre les erreurs les plus courantes que je vois. Je suppose que vous connaissez déjà les bases de React et que vous ne couvrirez donc pas les pièges tels que "ne pas muter les accessoires ou l'état".

Mauvaises habitudes

Chaque rubrique de cette section est une mauvaise habitude à éviter !

J'utiliserai l'exemple classique d'une application de liste de tâches pour illustrer certains de mes points.

État de duplication

Il devrait y avoir une seule source de vérité pour chaque élément d'état. Si la même information est stockée deux fois dans l'état, les deux éléments d'état peuvent se désynchroniser. Vous pouvez essayer d'écrire du code qui synchronise les deux éléments d'état, mais il s'agit d'un pansement sujet aux erreurs plutôt que d'une solution.

Voici un exemple d'état en double dans le contexte de notre application de liste de tâches. Nous devons suivre les éléments de la liste de tâches ainsi que ceux qui ont été cochés. Vous pouvez stocker deux tableaux dans l'état, avec un tableau contenant toutes les tâches et l'autre contenant uniquement celles terminées :

```
const [todos, setTodos] = useState<Todo[]>([])
const [completedTodos, setCompletedTodos] = useState<Todo[]>([])
```

Mais ce code est bogué au pire et malodorant au mieux ! Les tâches terminées sont stockées deux fois dans l'état, donc si l'utilisateur modifie le contenu textuel d'une tâche et que vous appelez uniquement `setTodos`, `completedTodos` contient maintenant l'ancien texte qui est incorrect !

Il existe plusieurs façons de dédupliquer votre état. Dans cet exemple artificiel, vous pouvez simplement ajouter un `completed` booléen au `Todo` type afin que le `completedTodos` tableau ne soit plus nécessaire.

Sous-utilisation des réducteurs

React a deux façons intégrées de stocker l'état : `useState` et `useReducer`. Il existe également d'innombrables bibliothèques pour gérer l'état global, Redux étant la plus populaire. Étant donné que Redux gère toutes les mises à jour d'état via des réducteurs, j'utiliserais le terme "réducteur" pour désigner



Sam Magura

Suivre

204 Pas de contenu

EMPLACEMENT

Raleigh, Caroline du Nord, États-Unis

ÉDUCATION

MS Mathématiques Appliquées, North Carolina State University

TRAVAIL

Ingénieur logiciel chez Spot

REJOINT

22 sept. 2021

Plus de Sam Magura

Présentation de SpotCoders, un Meetup de codage gratuit

#javascript # tapuscrit # réagir # débutants

Le moyen le plus simple de créer une application mobile ? CodeSandbox !

débutants # javascript # mobile

Le grand débat sur la boîte à outils Redux

redux # réagir # javascript # tapuscrit

à la fois les `useReducer` réducteurs et les réducteurs Redux.

`useState` est parfaitement bien lorsque les mises à jour d'état sont simples. Par exemple, vous pouvez `useState` suivre si une case est cochée ou suivre la valeur saisie d'un texte.

Cela étant dit, **lorsque les mises à jour d'état deviennent même légèrement complexes, vous devez utiliser un réducteur.** En particulier, **vous devez utiliser un réducteur chaque fois que vous stockez un tableau dans l'état et l'utilisateur peut modifier chaque élément du tableau.** Dans le cadre de notre application de liste de tâches, vous devez absolument gérer le tableau des tâches à l'aide d'un réducteur, que ce soit via `useReducer` ou Redux.

Les réducteurs sont bénéfiques car :

- Ils fournissent un emplacement centralisé pour définir la logique de transition d'état.
- Ils sont extrêmement faciles à tester unitairement.
- Ils déplacent la logique complexe hors de vos composants, ce qui donne des composants plus simples.
- Ils empêchent les mises à jour d'état d'être écrasées si deux modifications se produisent simultanément. Passer une fonction à `setState` est un autre moyen d'éviter cela.
- Ils permettent des optimisations de performances puisqu'ils `dispatch` ont une identité stable.
- Ils vous permettent d'écrire du code de style mutation avec `Immer`. Vous pouvez utiliser Immer avec `useState`, mais je ne pense pas que beaucoup de gens le fassent.

Ne pas écrire de tests unitaires pour les fruits à portée de main

Les développeurs sont des gens occupés et écrire des tests automatisés peut prendre du temps. Lorsque vous décidez si vous devez passer un test, demandez-vous : "Ce test aura-t-il suffisamment d'impact pour justifier le temps que j'ai passé à le rédiger ?" Quand la réponse est oui, écrivez le test !

Je trouve que les développeurs React de niveau intermédiaire n'écrivent généralement pas de tests, **même lorsque le test prendrait 5 minutes à écrire et aurait un impact moyen ou élevé !** Ces situations sont ce que j'appelle le "fruit à portée de main" des tests. **Testez les fruits à portée de main !!!**

En pratique, cela signifie écrire des tests unitaires pour toutes les fonctions "autonomes" qui contiennent une logique non triviale. Par autonome, j'entends des fonctions pures qui sont définies en dehors d'un composant React.

Les réducteurs en sont le parfait exemple ! Tous les réducteurs complexes de votre base de code doivent avoir une couverture de test de près de 100 %. Je recommande fortement de développer des réducteurs complexes avec le développement piloté par les tests. Cela signifie que vous allez écrire au moins un test pour chaque action gérée par le réducteur, et alterner entre l'écriture d'un test et l'écriture de la logique du réducteur qui fait passer le test.

Sous -utilisation `React.memo` , `useMemo` et `useCallback`

Les interfaces utilisateur alimentées par React peuvent devenir lentes dans de nombreux cas, en particulier lorsque vous associez des mises à jour d'état fréquentes à des composants coûteux à rendre (React Select et FontAwesome, je vous regarde.) Les React DevTools sont parfaits pour identifier les problèmes de performances de rendu , soit avec la case à

cocher "Mettre en surbrillance les mises à jour lors du rendu des composants", soit l'onglet du profileur.

Votre arme la plus puissante dans la lutte contre les mauvaises

performances de rendu est `React.memo`, qui ne restitue le composant que si ses accessoires ont changé. Le défi ici est de s'assurer que les accessoires ne changent pas à chaque rendu, auquel cas `React.memo` cela ne fera rien. Vous devrez utiliser les crochets `useMemo` et pour éviter cela `useCallback`.

J'aime utiliser de manière proactive `React.memo`, `useMemo` et `useCallback` pour prévenir les problèmes de performances avant qu'ils ne surviennent, mais une approche réactive - c'est-à-dire attendre pour effectuer des optimisations jusqu'à ce qu'un problème de performances soit identifié - peut également fonctionner.

Écrire `useEffect` des s qui s'exécutent trop souvent ou pas assez souvent

Ma seule plainte avec React Hooks est qu'il `useEffect` est facile d'en abuser.

Pour devenir un développeur React avancé, vous devez bien comprendre le comportement `useEffect` et les dépendances des tableaux.

Si vous n'utilisez pas le [plugin React Hooks ESLint](#), vous pouvez facilement manquer une dépendance de votre effet, ce qui se traduit par un effet qui ne s'exécute pas aussi souvent qu'il le devrait. Celui-ci est facile à corriger - utilisez simplement le plugin ESLint et corrigez les avertissements.

Une fois que vous avez toutes les dépendances répertoriées dans le tableau de dépendances, vous pouvez constater que votre effet s'exécute trop souvent. Par exemple, l'effet peut s'exécuter sur chaque rendu et provoquer une boucle de mise à jour infinie. Il n'y a pas de solution unique à ce problème, vous devrez donc analyser votre situation spécifique pour déterminer ce qui ne va pas. Je dirai que, si votre effet dépend d'une fonction, stocker cette fonction dans une référence est un modèle utile.

Comme ça:

```
const funcRef = useRef(func)

useEffect(() => {
  funcRef.current = func
})

useEffect(() => {
  // do some stuff and then call
  funcRef.current()
}, [/* ... */])
```

Ne pas tenir compte de la convivialité

En tant que développeur frontend, vous devez vous efforcer d'être plus qu'un simple programmeur. **Les meilleurs développeurs frontaux sont également des experts de la convivialité et de la conception Web, même si cela ne se reflète pas dans leurs intitulés de poste.**

La convivialité fait simplement référence à la facilité d'utilisation d'une application. Par exemple, est-il facile d'ajouter une nouvelle tâche à la liste ?

Si vous avez la possibilité d'effectuer des tests d'utilisabilité avec de vrais utilisateurs, c'est génial. La plupart d'entre nous n'ont pas ce luxe, nous devons donc concevoir des interfaces basées sur notre intuition de ce qui est convivial. **Cela revient en grande partie au bon sens et à l'observation de ce qui fonctionne ou ne fonctionne pas dans les applications que vous utilisez au quotidien.**

Voici quelques bonnes pratiques d'utilisation simples que vous pouvez mettre en œuvre dès aujourd'hui :

- Assurez-vous que les éléments cliquables semblent cliquables. Déplacer votre curseur sur un élément cliquable devrait changer légèrement la couleur de l'élément et faire en sorte que le curseur devienne une "main pointée", c'est-à-dire `cursor: pointer` en CSS. [Passez la souris sur un bouton Bootstrap pour voir ces meilleures pratiques en action.](#)
- Ne masquez pas les éléments importants de l'interface utilisateur. Imaginez une application de liste de tâches où le bouton "X" qui supprime une tâche est invisible jusqu'à ce que vous survoliez cette tâche spécifique. Certains concepteurs aiment à quel point c'est "propre", mais cela oblige l'utilisateur à chercher pour comprendre comment effectuer une action de base.
- Utilisez la couleur pour transmettre le sens. Lors de l'affichage d'un formulaire, utilisez une couleur en gras pour attirer l'attention sur le bouton d'envoi ! S'il y a un bouton qui supprime définitivement quelque chose, il vaut mieux qu'il soit rouge ! Consultez [les boutons](#) et les [alertes](#) de Bootstrap pour en avoir une idée.

Ne travaille pas vers la maîtrise du CSS et de la conception Web

Si vous souhaitez créer efficacement de belles interfaces utilisateur, vous devez maîtriser le CSS et la conception Web. Je ne m'attends pas à ce que les développeurs de niveau intermédiaire soient immédiatement capables de créer des interfaces propres et conviviales tout en maintenant leur efficacité à un niveau élevé. Il faut du temps pour apprendre les subtilités du CSS et créer une intuition pour ce qui semble bon. Mais vous devez travailler dans ce sens et vous améliorer avec le temps !

Il est difficile de donner des conseils spécifiques pour améliorer vos compétences en matière de style, mais en voici un : **master flexbox**. Bien que flexbox puisse être intimidant au début, c'est un outil polyvalent et puissant que vous pouvez utiliser pour créer pratiquement toutes les mises en page dont vous aurez besoin dans le développement quotidien.

Cela couvre les mauvaises habitudes! Voyez si vous êtes coupable de l'un d'entre eux et travaillez à vous améliorer. Maintenant, je vais faire un zoom arrière et discuter de quelques meilleures pratiques globales qui peuvent améliorer vos bases de code React.

Meilleures pratiques générales

Utiliser exclusivement TypeScript

Le JavaScript normal est un langage acceptable, mais la vérification du type de manque en fait un mauvais choix pour tout sauf les petits projets de loisirs. Écrire tout votre code en TypeScript augmentera considérablement la stabilité et la maintenabilité de votre application.

Si TypeScript vous semble trop complexe, continuez à travailler. Une fois que vous aurez acquis la maîtrise, vous pourrez écrire TypeScript aussi rapidement que vous pouvez écrire JavaScript maintenant.

Utiliser une bibliothèque de récupération de données

Comme je l'ai dit dans la section "Mauvaises habitudes" de cet article, il `useEffect` est difficile d'écrire correctement. Cela est particulièrement vrai lorsque vous utilisez `useEffect` directement pour charger des données à partir de l'API de votre backend. Vous vous épargnerez d'innombrables maux de tête en utilisant une bibliothèque qui résume les détails de la récupération des données. Ma préférence personnelle est [React Query](#), bien que [RTK Query](#), [SWR](#) et [Apollo](#) soient également d'excellentes options.

N'utilisez le rendu serveur que si vous en avez vraiment besoin

Le rendu côté serveur (SSR) est l'une des fonctionnalités les plus intéressantes de React. Cela ajoute également une énorme quantité de complexité à votre application. Bien que des frameworks comme Next.js facilitent grandement la SSR, il reste une complexité inévitable à gérer. Si vous avez besoin de SSR pour le référencement ou de temps de chargement rapides sur les appareils mobiles, utilisez-le par tous les moyens. Mais si vous écrivez une application métier qui n'a pas ces exigences, veuillez simplement utiliser le rendu côté client. Vous me remercierez plus tard.

Colocaliser des styles avec des composants

Le CSS d'une application peut rapidement devenir un désordre tentaculaire que personne ne comprend. Sass et d'autres préprocesseurs CSS ajoutent quelques avantages mais souffrent toujours en grande partie des mêmes problèmes que le CSS vanille.

Je pense que les styles doivent être limités à des composants React individuels, avec le CSS colocalisé avec le code React. Je recommande fortement de lire [l'excellent article de blog de Kent C. Dodds sur les avantages de la colocation](#). Le fait de limiter le CSS à des composants individuels conduit à la réutilisation des composants comme méthode principale de partage des styles et évite les problèmes où les styles sont accidentellement appliqués aux mauvais éléments.

Vous pouvez implémenter des styles colocalisés à l'échelle des composants à l'aide de [Emotion](#), [styled-components](#) ou [CSS Modules](#), entre autres bibliothèques similaires. Ma préférence personnelle est l'emotion avec l'`css` accessoire.

Mise à jour 2022-04-15 : Clarification de ma déclaration selon laquelle vous devez "toujours" utiliser un réducteur lorsque l'état est un tableau.

Débat (65)

S'abonner



Ajouter à la discussion



Taha Ben Massaud • 10 avril

Bien que je sois d'accord avec la plupart des conseils donnés, certains sont subjectifs et opiniâtres et ne reflètent pas nécessairement les meilleures pratiques universellement acceptées.

Par exemple, je préfère SASS/LESS à css-in-js. Mais encore une fois, c'est ma propre préférence subjective. De même, je préfère utiliser Elm, ReScript ou PureScript si je suis trop préoccupé par l'exactitude de l'application. Sinon, (JS + tests) est meilleur que TS pour un développement rapide et moins de frais généraux liés à la maintenance d'une montagne de types TS qui fuient souvent de toute façon.

De plus, il serait préférable de faire un zoom arrière et de réfléchir à l'utilité de React.memo. Peut-être que la restructuration de l'arborescence de réaction rendrait plusieurs instances de React.memo redondantes.

Je sais que je suis minoritaire mais je déteste les crochets à cause de tous les pièges dont il faut se souvenir lors de leur utilisation, surtout lorsqu'il y a 2 niveaux ou plus imbriqués. Découvrez RefractJs; pas besoin de crochets. Les HoC sont meilleurs parce qu'ils sont un modèle universel; pas l'API d'une bibliothèque.

20 aime ▾ Réponse



Nath • 10 avril • Modifié le 10 avril

TypeScript n'est pas parfait mais il vous offre de nombreux tests gratuits. Pourquoi perdre du temps à écrire des tests que vous obtenez gratuitement ? Les tests doivent tester la logique et non l'utilisation.

Peut-être que JS + Tests est plus rapide pour les développeurs juniors qui ne comprennent pas les systèmes de type. Mes stagiaires s'en sortent bien..

De plus, le typage statique aide à documenter l'utilisation des composants. Mes stagiaires l'adorent car il leur dit quand ils manquent un attribut requis ou quel type il attend comme paramètre.

D'accord sur Purescript / Elm etc.

10 aime Réponse



Joël BonetR • 13 avril

Le code auto-documenté est un mensonge dont je remercie sincèrement la communauté depuis longtemps.

2 aime Fil de discussion



Nath • 13 avril • Modifié le 13 avril

À peine. Donc, vous n'utilisez pas l'interface utilisateur Swagger ?

Des outils comme Swagger et le typage statique ne remplacent bien sûr pas toute la documentation. La documentation automatique est extrêmement précieuse.

Trop de développeurs JavaScript uniquement qui n'ont pas travaillé sur des projets d'entreprise à grande échelle.

4 aime Réponse



Collines propres • 10 avril

CSS in JS et Tapuscrit sont les meilleures pratiques et chaque jour, de plus en plus d'entreprises rejoignent le train en marche pour demander ces compétences. Bien que personnellement j'aime aussi prendre du recul dans le train en marche, le CSS in JS et le Tapuscrit deviennent rapidement des standards de l'industrie (le Tapuscrit l'est déjà) et vous allez nuire à votre carrière si vous refusez de maîtriser ces compétences.

4 aime Réponse



Taha Ben Massaud • 10 avril

Affirmer que CSS-in-JS et TS sont les meilleures pratiques est incorrect. Cependant, prétendre qu'il y a un élan derrière ces technologies est correct.

TS, à mon avis, est le nouveau Java. Cela construira une énorme pile de bases de code horribles que les futurs développeurs devront maintenir. C'est une opportunité autant qu'une malédiction 😅. Les langages FP purs avec un FFI clair sont meilleurs pour garantir l'exactitude des applications.

12 aime Fil de discussion



Collines propres • 11 avril

J'ai beaucoup travaillé avec les bases de code existantes en Java, TypeScript et Javascript. Votre affirmation selon laquelle TypeScript

est le nouveau Java est juste un peu ridicule... Si vous voulez faire une comparaison aussi tordue, Typescript est plus comparable à C# qu'à Java. La pile d'horribles bases de code dont vous parlez, je les ai vues mais elles étaient toutes en Javascript.

♥ 4 aime Q Réponse



Lars Rye Jeppesen • 13 avril

...



css-in-js est horrible, arrêtez-vous s'il vous plaît.

♥ 4 aime Q Réponse



Kanishka • 10 avril • Modifié le 10 avril

...



Sinon, (JS + tests) est meilleur que TS pour un développement rapide et moins de frais généraux liés à la maintenance d'une montagne de types TS qui fuient souvent de toute façon.

J'ai pensé à cela aussi. Je voudrais seulement utiliser rescript ou elm, ou juste js. De ces trois options, je penche légèrement vers js avec des tests car je n'ai pas à gérer d'outils spéciaux. Si elm prend de l'ampleur, ou si rescript lance un énorme framework développé en rescript, j'en ferais ma valeur par défaut. Je suis déçu qu'Elm n'ait pas gagné (encore ?). (Je construis très rarement de petits frontaux, donc mes opinions ici n'ont pas grand-chose pour les étayer).

J'ai également pensé à laisser les données d'objet métier au format json (par exemple [package.elm-lang.org/packages/1602...](#)) au lieu de les convertir en enregistrements, lors de l'utilisation d'elm ou de rescript, car je pense que la plupart des objets métier voudront éventuellement certains champs définis par l'utilisateur.

♥ 3 aime Q Réponse



Taha Ben Massaud • 10 avril

...



J'ai récemment lancé un projet JS et j'ai eu des problèmes avec des versions conflictuelles des dépendances npm. Il m'a fallu une journée pour équilibrer les versions correctes et je ne pouvais toujours pas utiliser la dernière version si l'une des dépendances principales. A un moment, j'ai été très tenté de passer à Elm car la gestion des dépendances est plus facile. Je ne voulais tout simplement pas gérer l'encodage/décodage de JSON ; paresseux 😅

♥ 4 aime ⌂ Fil de discussion



Kanishka • 10 avril

...



J'aurais juste utilisé une trappe d'évacuation comme [package.elm-lang.org/packages/1602...](#)

♥ 3 aime ⌂ Fil de discussion



Commentaire supprimé



Taha Ben Massaud • 10 avril

...



♥ 2 aime Q Réponse



Jared M. Smith • 13 avril

...



Elm est ma référence pour une belle expérience de développeur. Pour moi c'est "gagné".

Je laisse mon IDE (Atom/ [elmjutsu](#)) générer des décodeurs/encodeurs pour moi. Vous pouvez utiliser [korban.net/elm/json2elm/](#) si votre éditeur préféré ne prend pas en charge leur génération. Je ne recommande pas de contourner les décodeurs.

Je ne décode/encode que les données dont j'ai besoin. Parfois (généralement), je veux que mon modèle soit différent des données de l'API, alors je fais des ajustements pour cela. Le décodage m'aide même à trouver des failles dans mes hypothèses sur les données provenant de l'API !

Il y a [elm-graphql](#) si c'est une option pour vous.

3 aime Réponse



Lars Rye Jeppesen • 13 avril • Modifié le 14 avril

...

tailwindcss est la merde. css-in-js est le pire et fonctionne vraiment mal.

2 aime Réponse



Réactivité • 14 avril • Modifié le 14 avril

...

Je suis d'accord avec tout. Cependant, j'aime toujours les crochets

1 aimer Réponse



Joël BonetR • 9 avril • Modifié le 9 avril

...

Bon Dieu ! utiliser TS comme "bonne pratique"... Le message était bon puisque j'ai lu une chose aussi stupide.

Je vous encourage à apprendre JS correctement à la place.
Nous avons des outils en JS à utiliser selon nos besoins, ce n'est pas le langage de programmation le plus utilisé dans le monde par hasard.

réalités

Voulez-vous une vérification de type ? Ajoutez simplement

```
// @ts-check
```

en haut de votre fichier, VSCode le générera pour vous au moment du développement[*1], tout ce que vous avez à faire est d'ajouter JSDoc à vos fonctions/méthodes et variables, ce qui est l'une des meilleures pratiques les plus importantes : Documentez votre code.

Voulez-vous des types et une inférence de fonction/méthode ? Encore une fois, JSDoc.

Mieux intellisens ? Devinez quoi... JSDoc.

Vous souhaitez produire un code maintenable ? Appliquez les motifs SOLID et KISS.

Puisque nous parlons de réaction, vous devriez également connaître les prop-types, plus utiles que la vérification de type si vous tenez compte du contexte de React[*2].

À propos des bibliothèques de récupération de données, il existe une API de récupération par défaut dans JS, qui peut gérer TOUTES les fonctionnalités existantes fournies par les requêtes HTTP (de plus, l'API de récupération arrive également sur Node). Pourquoi en ajouter un supplémentaire ?

Une bonne pratique importante (et réelle) consiste à ajouter le minimum de bibliothèques tierces à votre projet. Cela rend votre projet plus robuste, plus rapide à construire/déployer et généralement plus sécurisé.[*3]

opinion personnelle

Je trouve les composants stylés utiles et probablement la meilleure option pour plusieurs raisons (tout est un composant, pas de CSS inutilisé, beaucoup plus propre que tailwind, capacités sass/scss par défaut, JSX à l'intérieur de la définition des styles, il peut gérer les accessoires JS (utile pour les interactions que CSS seul ne peut pas gérer et vous pouvez le placer sur un autre fichier).

J'aime avoir 3 fichiers dans chaque dossier de composant, ce sont, par exemple :

monComposant.js
monComposant.styles.js
monComposant.utils.js

Ce qui, je suppose, est explicite (demandez-moi si vous voulez des détails).

[*1] Cela signifie que vous n'avez pas besoin d'ajouter TS à votre projet et d'attendre le temps de construction supplémentaire, cela fonctionnera, disons, en cours d'exécution, pendant que vous développez à la place. 0 temps perdu et la même vérification de type que l'ajout de l'énorme bibliothèque TS à votre projet, n'est-ce pas incroyable ? 😊

[*2] La majeure partie de la logique doit se trouver dans le backend. Vous ne devez gérer la logique d'interaction que dans le frontend, ce qui facilite la portée de tout ce qui lui appartient afin que vous puissiez échanger votre backend ou frontend sans l'enfer d'un code avec un couplage lourd. Il peut sembler que React existera pour toujours, mais les personnes qui migrent de JQuery-UI remercient peut-être la même chose ces jours-ci 😅

[*3] Par robustesse, cela signifie que vous n'avez pas besoin de mettre à jour votre projet chaque fois qu'une bibliothèque change sa façon de travailler, s'appuyer sur l'API du langage (si cela convient aux besoins du projet) est généralement préférable.

Par sécurisé, j'entends que la probabilité qu'un tiers traite une faille de sécurité est plus élevée que l'utilisation de l'API de langage elle-même en règle générale.

Je trouve plus rapide à construire/déployer comme explicite.

♡ 12 aime ⚒ Réponse



Maroš Betko • 10 avril

...

Apprendre JS correctement comme vous l'avez dit et utiliser Ts ne sont en aucun cas exclusifs. Vous écrivez pour ne pas utiliser Ts mais suggérez immédiatement d'utiliser le pragma ts-check. Bien que JSDoc soit un outil puissant, il ne doit pas être utilisé pour la vérification de votre code, c'est encore une fois grâce à TS qui extrait des informations de votre documentation js et n'est en aucun cas une solution solide pour les projets plus importants.

Aussi à propos des bibliothèques de récupération de données, vous n'avez clairement pas compris à quoi elles servent. React Query n'est pas là pour remplacer fetch, il utilise fetch pour travailler intelligemment et efficacement avec des données asynchrones.

♡ 7 aime ⚒ Réponse



Joël BonetR • 10 avril • Modifié le 10 avril

...

Apprendre JS correctement comme vous l'avez dit et utiliser Ts ne sont en aucun cas exclusifs.

Bien sûr que non, je suis totalement d'accord avec cela. Je ne modifie pas le commentaire précédent pour garder les choses telles qu'elles sont, j'aurais dû éviter le mot "au lieu" à la fin ou le communiquer d'une manière différente.

Vous écrivez pour ne pas utiliser Ts mais suggérez immédiatement d'utiliser le pragma ts-check.

J'ai dit que si vous voulez une vérification de type pour une raison quelconque, vous pouvez simplement l'utiliser à la place en chargeant l'intégralité du TS en tant que dépendance dans votre projet.

Bien que JSDoc soit un outil puissant, il ne doit pas être utilisé pour la vérification de votre code

Pourquoi pas? Les annotations JSDoc vivent dans les commentaires, plutôt que directement dans la syntaxe, ce que je préfère mais c'est totalement opiniâtre, vous pouvez choisir l'un ou l'autre ou aucun des deux et ce sera OK si cela convient aux besoins de votre projet.

c'est encore une fois grâce à TS qui extrait des informations de votre documentation js et n'est en aucun cas une solution solide pour les projets plus importants. Aussi à propos des bibliothèques de récupération de données, vous n'avez clairement pas compris à quoi elles servent. React Query n'est pas là pour remplacer fetch, il utilise fetch pour travailler intelligemment et efficacement avec des données asynchrones.

Cet article porte sur les bonnes pratiques, donc mon commentaire vise à éviter les choses opiniâtres dans les bonnes pratiques plutôt que de discuter de l'utilité d'une bibliothèque donnée.

Nous devons séparer ce que nous "aimons" ou ce que nous avons trouvé "utile" ou ce qui a "bien fonctionné pour nous dans un cas d'utilisation spécifique" afin de pouvoir différencier les bonnes pratiques réelles de nos préférences.

Préférez-vous TS pour certaines raisons ? C'est bon, allez-y. Préférez-vous éviter d'utiliser TS ? C'est tout aussi bon, allez-y. Il en va de même pour la récupération des bibliothèques.

Est-ce que la frappe forte est bonne ? Bien sûr, cela peut éviter les erreurs d'exécution, mais ce n'est pas le seul moyen d'atteindre le même résultat.

Dire que TS ou récupérer des bibliothèques sont de "bonnes pratiques" est tellement hors de la réalité que cela ne peut tout simplement pas convenir. Bien sûr, les meilleures pratiques ne sont pas spécifiques à une langue, c'est pourquoi l'ajout de TS ou d'une bibliothèque donnée n'est qu'une opinion (à laquelle vous pouvez être d'accord ou non, mais ce n'est pas le sujet de la discussion).

♡ 3 aime ↗ Fil de discussion



Collines propres • 11 avril

...

La raison pour laquelle nous pouvons dire ce qu'est une meilleure pratique est que nous pouvons voir Typescript gagner massivement en popularité. Nous pouvons voir du code écrit en Typescript échouer beaucoup moins fréquemment en production. Nous pouvons voir des développeurs juniors apprendre beaucoup plus rapidement à écrire du code testable approprié.

Javascript a engendré tous ces développeurs cowboys qui ont commencé leur propre religion d'écriture de code approprié en Javascript (avec 1 adepte par religion), car le langage lui-même n'a aucune opinion sur ce qu'est le code approprié. Il y a une raison pour laquelle tous ces "outils" que vous avez mentionnés plus tôt existent maintenant. Typescript est l'évolution naturelle de certains de ces outils.

Et oui, les meilleures pratiques peuvent certainement être spécifiques à une langue. Par exemple, sa meilleure pratique consiste à utiliser une indentation appropriée dans Typescript, en Python.

votre code ne fonctionne tout simplement pas et dans d'autres langages comme le cobal, apparemment, sa meilleure pratique consiste à ne pas utiliser d'indentation du tout : P

 1 aimer  Fil de discussion

 Joël BonetR • 11 avril • Modifié le 11 avril

Ce script a l'air bien, mais vous vous rendez compte que JSDoc existe depuis les années 90 et que les capacités de Doc sont dans presque toutes les langues...

Je suis toujours du côté JS ici dans la communauté Dev pour une bonne raison. Vous défendez TS avec des préférences personnelles plutôt que d'analyser correctement les besoins du projet.

Je préférerais utiliser TS dans un backend Node qui fournit une logique métier mais je trouve un non-sens en l'ajoutant au service d'authentification (qui validera simplement les jetons de Google IDP par exemple).

J'éviterais probablement d'ajouter TS à un microservice qui regroupe 2 tiers (vous n'obtiendrez de toute façon pas un typage fort à l'exécution, donc tout ce que vous avez à faire est une prop map, et le résultat sera le même avec vanilla JS qu'avec TS, vous ajouterez probablement une bibliothèque de validation comme Joi).

Donc non, on ne peut pas dire que l'utilisation de TS est une bonne pratique, comme toute tech, c'est bien là où ça a du sens. Je suis fatigué de dire qu'il n'y a pas de technologie unique pour les gouverner tous, et que nous travauillons dans un domaine scientifique. Ajouter des préférences personnelles à vos projets leur nuit d'une manière ou d'une autre. C'est-à-dire que ne pas utiliser TS alors que vous devriez rendre le produit moins robuste.

L'ajout de TS là où vous ne devriez pas augmenter vos temps de développement pour une raison quelconque.

Allez, vous pouvez trouver des définitions appropriées sans opinion sur les [meilleures pratiques](#) sur Internet.

 2 aime  Fil de discussion

 Collines propres • 11 avril • Modifié le 11 avril

Nous parlions ici de projets React. La page Wikipédia que vous liez contient la définition des meilleures pratiques. Il ne tient pas une liste exhaustive de toutes les meilleures pratiques utilisées dans le domaine, ni ne prétend que tout ce qui n'y est pas mentionné n'est pas une meilleure pratique.

"Les meilleures pratiques de codage sont un ensemble de règles informelles que la communauté de développement de logiciels utilise pour aider à améliorer la qualité des logiciels"

Améliorer la qualité des logiciels, c'est ce que fait Typescript, en particulier pour les projets React.

 2 aime  Fil de discussion

 Joël BonetR • 11 avril • Modifié le 11 avril

Ceci est une opinion, pas une définition. En supposant que TS améliore la qualité du logiciel, nous pouvons extrapoler cela et dire que Java (qui inclut les "côtés forts" de TS par défaut) améliore la qualité du logiciel et, par extension, tout le code Java a de la qualité.

J'ai travaillé 4 ans en utilisant Java et je suis sûr de dire que ce n'est pas vrai.

Un typage fort donne ainsi à votre code une **robustesse** qui n'est

absolument pas la même que la **qualité**.

Les principes SOLID, les modèles de conception (quand ils conviennent), la documentation, les tests, les bonnes conventions de nommage, etc. sont des éléments qui améliorent la qualité du logiciel et certains d'entre eux ajoutent également de la robustesse.

De plus, TS brille lorsqu'il s'agit d'une logique complexe (car il ajoute de la robustesse, que ce soit avec le pragma ts-check ou en ajoutant TS dans votre projet) et vous n'êtes pas censé ajouter une logique complexe dans le frontent (cela ajoute du couplage et rend difficile la migration vers une nouvelle technologie frontale quand réagir meurt dans le futur) donc je dirais plutôt "spécialement dans les services de noeuds complexes" plutôt que "spécialement dans React", où vous n'obtiendrez que la logique d'interaction.

 2 aime  [Fil de discussion](#)

 **Collines propres** • 11 avril

Je vais simplement ignorer la comparaison absurde avec Java (la manière dont Typescript traite la sécurité des types est complètement différente de Java) et ne répondrai qu'à la partie la plus sensée de votre réponse. SOLID est génial, en particulier les parties qui se répercutent sur la programmation fonctionnelle, oui les modèles de conception, la documentation, les tests, les bonnes conventions de dénomination, tout à fait d'accord et il n'y a aucune raison pour laquelle vous ne pouvez pas le faire dans Typescript. La deuxième partie de votre réponse ... Ce n'est pas le but de Typescript, le but de Typescript est que je peux avoir confiance que lorsqu'un membre de l'équipe écrit du code ou qu'un stagiaire écrit du code et après un certain temps, je dois apporter des modifications à cela code, je ne suis pas confronté à un grand désordre de différents styles de codage et de déclarations de variables qui sont modifiés partout, contenant des valeurs que je ne peux pas prédire. Typescript m'a rendu la vie beaucoup plus facile et moins remplie de corrections de bogues et de fautes de frappe. Si vous voulez rejeter cela comme une opinion personnelle, allez-y lol

 2 aime  [Fil de discussion](#)

 **Joël BonetR** • 11 avril • Modifié le 11 avril

Je connais cette différence entre Java et TS, mais faire l'hypothèse que TS traite toujours un code de qualité est absurde de la même manière.

Je suis d'accord avec vous dans la plupart des cas pour être honnête, c'est juste que, pour ajouter de l'objectivité à l'équation, vous pouvez obtenir la même chose avec les définitions JSDoc et en utilisant le pragma ts-check au lieu de charger l'intégralité du TS en tant que dépendance du projet.

Encore une fois, je ne veux pas dire que l'utilisation de TS est mauvaise ou toute autre chose, c'est juste que fanboyer une technologie vous fait penser que c'est bon pour tout le monde dans chaque situation et pour chaque cas d'utilisation auquel vous devez faire face et c'est simplement pas vrai.

Pour obtenir la bonne pile technologique pour un projet, vous devez vérifier différentes métriques ; le ou les cas d'utilisation spécifiques à couvrir, la planification future, les ressources humaines disponibles (quantité, ancienneté, expérience antérieure), l'expérience globale de l'équipe sur chaque pile technologique, etc.

Ensuite, dans un monde parfait, tous les projets auraient des tests unitaires, des tests e2e, une bonne documentation, une confluence à jour avec les critères d'acceptation à revoir avant de refactoriser

quelque chose et un énorme etc. de choses.

La vraie vie pour la plupart des projets est que les besoins de l'entreprise passent en premier et cela signifie généralement "Nous avons besoin que cela soit fait avant [DATE LIMITE]", ce qui conduit inévitablement à réduire le temps dans certaines de ces choses, vous devez donc choisir.

Préférez-vous un temps de développement prolongé en utilisant TS ? Vous préférez les tests unitaires ? Appliquez-vous la documentation ?

Il existe différentes approches qui vous conduisent -presque- au même résultat, par exemple, si vous adoptez l'approche "tout est un composant" dans React, l'utilisation de propTypes est plus flexible que la frappe forte de votre code, en plus vous ajoutez une définition explicite de ce qui est facultatif et ce qui est requis et ainsi de suite, juste pour ajouter un exemple incomplet.

Ce n'est pas tout noir ou blanc, ce n'est pas une "bonne pratique", ce n'est pas une utilisation TS ou ne pas utiliser TS, c'est une définition de pile technologique et ce sera la bonne ou non en fonction de ces différentes métriques que j'ajoute avant et cela peut être une question sur "combien de TS voulez-vous impliquer dans votre projet" et pour quelle raison.

J'ajoute 2 cas d'utilisation d'il y a quelques mois juste pour illustrer.

Application de chat en temps réel avec Node + React utilisant Websockets

et la persistance des données dans une base de données MySQL.

Il était destiné à être intégré dans une application Web différente en tant qu'application Web autonome.

Le code React n'est qu'un point d'entrée pour authentifier l'utilisateur dans le contexte + le composant de chat (divisé en deux, l'endroit où vous tapez et l'endroit où vous lisez). Environ 200 lignes de code autant, vous ne traitez que des chaînes.

Node backend c'est juste l'émetteur d'événements, le persister (pour stocker) et le lecteur (pour obtenir l'historique des discussions précédentes), environ 150 lignes de code autant si vous assemblez les fichiers, encore une fois, vous ne traitez que des chaînes.

Il n'y a aucun avantage logique à ajouter TS. Vous pouvez fantasmer avec des choses comme "si le projet grandit alors...", "si bla bla... alors" mais la réalité est que ce projet restera tel quel pendant toute sa vie.

Service pour gérer la logique métier

Entre un frontend React et un CMS headless.

- Prévision du temps de support minimum du projet ? 5 années.
 - Prévision de croissance du projet ? Gros.
 - Ce cas d'utilisation convient-il à TS dans le service Node ? Bien sûr que c'est le cas ! Je ne recommanderais pas de le faire sans TS.
 - Ce cas d'utilisation **doit** -il avoir TS dans la partie React (o/n) ? Non.
 - Sommes-nous sur le point d'utiliser TS ici ? Voyons le reste des métriques, oh, l'équipe a de l'expérience avec JS mais peu en TS, nous sommes sur le point d'ajouter uniquement la logique d'interaction à l'interface afin de pouvoir couvrir la robustesse avec JSDoc, TS-pragma et prop-types.
-

C'est la vérité objective à ce sujet en ce qui concerne les métriques, mais vous semblez être le seul développeur qui pleure à l'arrière en disant "mais nous DEVONS utiliser TS" (généralement simplement parce qu'il y est habitué et par préférence personnelle) et ne peut pas donner aucune bonne raison pour cela aux gens de l'architecture et de la gestion.

2 aime Fil de discussion



Long Tao • 11 avril • Modifié le 11 avril

Je ne connais pas l'homme, jsdocs, proptypes .. Je vois cela trop souvent comme un vœu pieux, car un seul gars dogmatique de toute l'équipe les maintient à jour et cohérents. Strict ts est plus convivial pour une équipe de niveau mixte, car il impose moins de contraintes aux ingénieurs seniors lors de la révision du code pour toujours vérifier si les "meilleures pratiques" sont respectées (et réduit les frictions dues au blocage du code pendant l'étape de rétroaction du code)

Une base de code avec jsdoc/proptypes/etc obsolète me fait pleurer.

3 aime Fil de discussion



Joël BonetR • 11 avril

Vous pouvez utiliser un linter pour vous assurer que ceux-ci sont présents et valider également ts-pragma, de sorte que vous ne pouvez pas avoir une définition invalide

1 aimer Fil de discussion



Collines propres • 14 avril

Ou vous ne pouvez pas utiliser toutes ces choses et n'utiliser qu'une seule chose : Typescript :P

1 aimer Fil de discussion



Joël BonetR • 14 avril

Bien sûr, c'est une option que je suis en fonction du cas d'utilisation :)

1 aimer Réponse



MehYam • 9 avril

Erreur majeure dans votre message : vous avez centré votre en-tête "opinion personnelle" au lieu de l'aligner vers le haut.

15 aimer Réponse



Sam Magura ♂ • 10 avril

[@mehyam](#) _

3 aimer Réponse



Joël BonetR • 10 avril

C'est une introduction 😊

1 aimer Réponse



Nicolas Peluchetti • 11 avril

Je ne suis vraiment pas d'accord avec vous sur ce point, écrire jsDocs ou PHPDocs est l'une des plus grosses pertes de temps, à mon avis : à quand remonte la dernière fois que vous avez lu un commentaire utile dans un jsDoc ?

Je pense que tout ce que jsDocs fait est soit:

- agrandissez vos fichiers sans raison particulière
- vous faire réfléchir à deux fois avant d'écrire une fonction, car vous êtes obligé de mettre dans le commentaire passe-partout ans : je comprends que WordPress a besoin d'une documentation appropriée, c'est une bibliothèque, mais le code que vous écrivez pour les clients doit être clair, ne pas avoir de commentaires standard, donc fondamentalement, il suffit de commenter lorsque vous faites quelque chose de mal. Pour moi, la norme d'or devrait être de tout taper, en PHP, vous l'obtenez gratuitement, en JS, vous avez besoin de tapuscrit.

4 aime Réponse

Joël BonetR • 11 avril • Modifié le 11 avril

C'est la première fois de ma vie que je lis quelque chose qui dit que "documenter votre code est mauvais". Vous me surprenez plus chaque jour 😅

Bien sûr, vous devez réfléchir avant d'écrire une fonction, et comme meilleure pratique, vous réécrirez probablement cette fonction en la divisant en deux ou trois parce que vous avez oublié d'appliquer le S sur SOLID (responsabilité unique).

N'oubliez pas que la première moitié du travail d'un développeur consiste à faire fonctionner le code, l'autre moitié à le rendre maintenable, lisible, documenté et testé.

4 aime Fil de discussion

Collines propres • 13 avril

Oui, un bon code n'a pas besoin de documentation. Lisez Clean Code de Martin Robert. Si votre code a besoin de documentation, ce n'est pas facile à lire/comprendre et c'est donc un mauvais code.

1 aimer Fil de discussion

Joël BonetR • 14 avril • Modifié le 14 avril

Cela décrit un concept et non une réalité. La perfection maximale dans une équipe serait de comprendre le code de l'autre à la perfection, tous avec le même niveau/quantité de connaissances et travaillant ensemble depuis des années.

Dans un projet réel de travail en équipe, vous obtiendrez généralement différents niveaux d'expertise, il y a une rotation des personnes et vous devrez maintenir le code pendant longtemps.

Je peux coder une composition de fonction dans mon code et le junior viendra dire "bruh, WTF". Si nous pouvons prendre le temps et les ressources nécessaires pour éléver l'ancienneté à un niveau "minimum" donné pour le projet, ce serait bien (je suppose que je ne l'ai jamais vu IRL).

Mais même cela, il n'y a pas de modèle unique ou de meilleure pratique autre que la documentation de votre code qui peut vous libérer du fardeau de tout mémoriser à une étape ultérieure.

Disons que vous travaillez 5 ans dans un projet et que quelqu'un de nouveau arrive. Comprendre ce qu'il fait dans l'ensemble est une question de lecture des noms de fonction (si la base de code est bonne).

c'est-à-dire `getUserId(user)` obtiendra évidemment l'ID utilisateur, probablement à partir d'un objet utilisateur. Si vous utilisez TS, vous pouvez voir que l'ID utilisateur est un nombre et que le paramètre utilisateur est un objet utilisateur. C'est toute la connaissance que vous pouvez tirer de la partie déclarative du code.

Généralement, vous verrez quelque chose comme :

```
const getNewNickname = (user) => generateNickname(randomize(cap
```

L'ajout de doc serait quelque chose comme:

```
/**  
 * Generates a new random nickname based on the current user  
 * @param {Object<User>} user  
 * @returns {string} Nickname  
 */  
const getNewNickname = (user) => capitalize(generateNickname(
```

N'est-ce pas beau ?

Allons un peu plus loin en ajoutant la composition de la fonction :

```
const compose = (...fns) => x => fns.reduceRight((y, f) => f(y))  
const getNewNickname = compose(capitalize, generateNickname, ra
```

Est-ce que tout le monde peut prédire ce que cela fera exactement sans avoir à réfléchir pendant quelques secondes ? Je peux honnêtement dire qu'un quart de mon équipe actuelle ne le ferait pas.

Ajoutons-y Doc :

```
/**  
 * Executes a sort of functions from right to left, passing the  
 * @param {...any} fns functions  
 * @returns {any} execution result  
 */  
const compose = (...fns) => x => fns.reduceRight((y, f) => f(y))  
  
/**  
 * Generates a new random nickname based on the current user na  
 * @type {string} nickname  
 */  
const getNewNickname = compose(capitalize, generateNickname, ra
```

N'est-ce pas mieux, plus rapide et plus rapide ?

J'ai donc perdu moins d'une minute pour éviter que d'autres membres de l'équipe n'en perdent quelques-uns à l'avenir.

Et nous venons de voir que la partie déclarative, sur la partie impérative des aides et des fonctions, pourrait être plus difficile à rattraper rapidement car ils auront du code procédural avec des conditions, des boucles, etc., et c'est là que la doc brille le plus.

 2 aime  Fil de discussion

 Collines propres • 14 avril

Le code propre est un concept qui est né d'une tonne d'expérience pratique... Oncle Bob a quoi ? 40 ans d'expérience dans le domaine ? Et ce n'est pas comme s'il était seul, ou que le concept de code propre n'avait pas été adopté et mis en pratique au cours des 14 dernières années...

Vous pouvez réclamer tout ce que vous voulez... Aucun développeur junior ne comprendra ce que vos commentaires JSDoc signifient vraiment, ou ne saura pas comment l'utiliser correctement lui-même. Avec Tapuscrit, ils n'ont pas le choix... Mais plus important encore, ils n'ont pas à s'en soucier du tout, car Typescript fait automatiquement l'inférence de type et il leur dira simplement ce qu'ils font de mal s'ils essaient d'attribuer le mauvais type à un variable ou fonction :B aucune compréhension de JSDoc requise et un tas d'autres outils et plugins de code VS dont vous avez besoin... J'ai l'expérience complètement opposée à ce que vous prétendez... Les développeurs juniors que j'ai rencontrés ont trouvé Typescript très facile à apprendre et utilise.

JSDoc, comme PHPDoc est archaïque, datant d'un siècle avant que nous ayons des systèmes de typage modernes et découle de Javadoc qui est encore plus archaïque...

Je suis d'accord, l'exemple que vous donnez est un horrible obscurcissement de la logique que vous ne devriez pas faire du tout. Si quelqu'un modifie ce qui est renvoyé par l'une des fonctions de votre composition et ne met pas à jour le JSDoc pour cela, Javascript va planter lors de l'exécution et j'aimerais que vous laissiez un développeur junior essayer de trouver et de corriger cette erreur d'exécution avec votre composition de la fonction. Si vous travaillez avec 10 développeurs, je vous garantis qu'au moins 5 d'entre eux ne se soucieront pas beaucoup de la mise à jour des commentaires de code lorsqu'ils écrivent du code. Je préfère de loin les développeurs qui essaient d'écrire du code propre, au lieu d'essayer d'expliquer leur gâchis avec JSDoc.

 1 aimer  Fil de discussion

 Joël BonetR • 14 avril

Donc, pour récapituler, vous utiliserez toujours TS à la place de JS, n'est-ce pas ?

 1 aimer  Fil de discussion

 Collines propres • 15 avril

J'utilise JS puisque TS n'est qu'un sur-ensemble de JS, mais oui, pour la sécurité des types et pour éviter un million d'erreurs d'exécution à chaque fois que vous oubliez quelque chose, j'utiliserais TS jusqu'à ce que quelque chose de mieux se présente.

 1 aimer  Fil de discussion

 Joël BonetR • 16 avril

Vous avez donc pris une décision basée sur d'autres décisions que vous avez prises auparavant (apprendre TS et ignorer autant que possible le vanilla JS).

"Si le seul outil dont vous disposez est un marteau, vous avez tendance à tout voir comme un clou."

En gardant à l'esprit que TS ne vérifie pas les types au moment de l'exécution et connaissant également le contexte sur lequel vous travaillez (dans ce cas, React), où les erreurs d'exécution sont principalement dues à une erreur d'API (le gars du backend a oublié d'ajouter une propriété, le swagger dit que dans ce contrat, vous devriez obtenir une chaîne mais à la place vous avez un entier et ainsi de suite) vous rencontrerez exactement le même nombre d'erreurs d'exécution en utilisant une chose ou une autre mais bon, ce sont vos projets, pas les miens.

 1 aimer  Fil de discussion



Collines propres • 16 avril

La quantité d'hypothèses que vous faites sur mon expérience de travail avec d'autres langues montre à quel point vous êtes vraiment étroit d'esprit.

Il existe des moyens d'éviter les erreurs d'exécution dues aux modifications de l'API, mais je suis sûr que vous avez également une opinion très arrêtée sur ces outils et franchement, j'en suis venu à la conclusion qu'il est totalement inutile de discuter de ces choses avec vous.

2 aime Réponse



Kanishka • 10 avril

En termes de temps de compilation, ReScript est ultra-rapide, si jamais vous voulez vous y essayer. Je ne suggère pas d'y passer, c'est juste une chose intéressante à essayer si la vitesse de compilation est quelque chose qui vous dérange à propos du tapuscrit.

2 aime Réponse



Joël BonetR • 10 avril

Je signale juste certaines choses là-bas 😊 Merci pour votre commentaire, je vais vérifier ça :)

1 aimer Réponse



Vincent S.C. • 13 avril

Je ne suis certainement pas le plus pur de js (en fait, j'adore simplement les types, les interfaces et les classes abstraites) mais je pense que vous avez tout à fait raison avec vos déclarations, en particulier concernant React.

Le framework lui-même n'est pas vraiment idéal pour les projets particulièrement compliqués, et TS n'est qu'une autre couche de construction qui n'est pas vraiment nécessaire si vous êtes à l'aise avec JSDoc.

Je suis TOUJOURS étonné que les documents React ne semblent pas du tout faire apparaître des composants stylés. Je déteste à quel point css-in-js est inflexible. C'est absolument absurde si vous construisez quelque chose de plus grand qu'un site Web personnel ou une série de composants détachés. Avoir un SASS ou LESS est un tel gain de temps et optimise vraiment la façon dont vous pouvez utiliser CSS

1 aimer Réponse



Collines propres • 11 avril

Ou au lieu d'apprendre tous vos "outils en js", vous pouvez simplement apprendre Typescript correctement car il a toutes ces choses intégrées :)

2 aime Réponse



Joël BonetR • 13 avril

En suivant la même logique, vous pouvez simplement apprendre Kotlin, puis le [transpiler en JS](#), car il a encore plus de capacités que TypeScript et vous pouvez également coder des applications Android natives, des services Web, des passerelles, etc., en s'exécutant sur la JVM, ce qui est mieux pour tâches de calcul complexes que JS sur Node.

Peut-être pourriez-vous simplement coder dans Rust et [le compiler en WebAssembly](#).

Il existe de nombreuses options si vous voulez vraiment éviter de creuser profondément dans un langage de programmation ! Si vous préférez simplement gratter la surface, alors vous êtes fatigué pour une raison quelconque et avez besoin d'un défi différent, allez-y.

Je suis à peu près sûr que plus de la moitié des personnes qui lisent ceci ne connaissent même pas JSDoc, j'ai même reçu des commentaires sur la qualité de la documentation de votre code 😊

1 aimer [Fil de discussion](#)



Collines propres • 13 avril

Si votre code a besoin de documentation pour être compréhensible, c'est mauvais. Je recommande vivement la lecture de Clean Code de Robert Martin. Bien sûr, vous pouvez inventer votre propre langage et être un cow-boy de développement solo et personne ne comprendra ce que vous faites, ou vous pouvez simplement utiliser ce qui existe déjà et fonctionne apparemment pour une grande partie de la population de développement. JSDoc ne fonctionne pas, c'est pourquoi personne ne l'utilise, sauf un très petit groupe de prédicateurs. Et oui, je vous recommande fortement de creuser un peu plus dans Typescript jusqu'à ce que vous compreniez que ce n'est pas comparable à Java.

1 aimer Réponse



Alex Benfaremo • 13 avril • Modifié le 13 avril

En parlant de "// @ts-check", je ne sais pas si vous utilisez le commentaire pour certains moyens, mais en fait, il existe une option dans les paramètres qui l'a activé par défaut pour tous les fichiers javascript.

Aimer Réponse



Joël BonetR • 13 avril

L'ajout du pragma TS le fait fonctionner sur n'importe quel membre de l'équipe VSCode sans avoir à vérifier qu'il l'a activé dans son IDE, c'est donc pratique lorsque vous travaillez en équipe.

2 aime [Fil de discussion](#)



Alex Benfaremo • 13 avril

logique :)

1 aimer Réponse



Lars Rye Jeppesen • 13 avril

Wow, je ne peux pas croire que vous ayez écrit ceci.

1 aimer Réponse



Commentaire marqué comme de mauvaise qualité/non constructif par la communauté. [Voir le code de conduite](#)



kiranmarshall • 11 avril

Dieu tier troll ou divagations d'un fou ?

3 aime Réponse



Edwin • 9 avril

•••

Excellent article, je suis d'accord avec la plupart de ces points. Pas trop sûr d'utiliser des réducteurs uniquement pour les tableaux, car ils ajoutent également un peu de complexité et se passe-partout eux-mêmes. Je dirais que cela aide si vous pouvez identifier des actions uniques dans votre arborescence de composants qui affectent plusieurs parties de l'état.

Une chose que j'ai remarquée, c'est que les gens essaient de trouver leurs propres solutions alors que React a une solution quelque peu opiniâtre à ce problème. Cela se résume principalement à essayer d'optimiser prématûrement ou à stocker des valeurs calculées dans l'état (conduisant aux problèmes que vous décrivez) - ils pensent qu'un rendu serait trop cher en fonction de leur intuition au lieu de le mesurer réellement.

Je ne suis pas un grand fan du code lié aux tests unitaires React (l'état interne de votre application), mais pour une logique métier spécifique, cela pourrait avoir du sens. Je pense que Kent C. Dodds est sur quelque chose avec son [trophée de test](#) - principalement écrire des tests d'intégration afin que vous puissiez vérifier les cas d'utilisation les plus importants de votre application et essayer d'éviter de tester les composants internes.

De plus, l'utilisation [de la bibliothèque de tests](#) (avec Cypress) m'a ouvert les yeux. Plusieurs fois, j'ai essayé d'écrire un test en utilisant des méthodes d'interrogation qui testent l'interface en fonction de ce que l'utilisateur voit (rôle aria, étiquette de formulaire, étiquette aria, texte, etc.), mais j'ai remarqué que je ne pouvais pas car il n'y avait pas d'étiquettes appropriées sur mes éléments. En écrivant le test, l'accessibilité de mon application s'est également améliorée ! Gagner gagner!

5 aime Réponse



Sam Magura ♂ • 9 avril • Modifié le 9 avril

•••

Merci Edwin. Quand je dis "utiliser des réducteurs pour les tableaux", je fais principalement référence au cas d'utilisation "J'ai un tableau d'objets et l'utilisateur peut modifier chaque objet". Pour un simple tableau d'ID, le réducteur n'est pas vraiment nécessaire.

J'aime aussi le trophée de test, donc je suis d'accord avec vous là-bas !

`@testing-library/cypress` est un sujet intéressant... J'ai utilisé React Testing Library et Cypress, mais pas ensemble 😊

3 aime Réponse



Collines propres • 10 avril

•••

Je suis d'accord avec tout, sauf les réducteurs. Bien qu'il soit bon de rester à jour avec une bibliothèque comme redux en raison de son utilisation répandue, et l'utilisation de crochets de réaction intégrés est certainement un bon conseil. Personnellement, je préfère des solutions de gestion d'état plus légères comme jotai et j'espère voir un éloignement du redux dans l'industrie.

2 aime Réponse



Résultats techniques vitaux • 13 avril • Modifié le 13 avril

•••

Il s'agit d'un article écrit de manière très professionnelle qui profite à la communauté dev.to.

Que pensez-vous de TailwindCSS ? J'ai appris à préférer Tailwind aux composants stylés.

Je suis d'accord sur l'importance de Typescript. J'ai récemment travaillé sur un grand site avec plus d'un million de dollars de ventes mensuelles et il n'utilise `.js` que (pas un seul `.tsx` fichier). À cet égard, ce qui différencie une opinion de la meilleure pratique. Je pense que l'utilisation de TypeScript est

la meilleure pratique. Ou est-ce un avis ?

Je partage certainement cela sur Linkedin.

1 aimer Réponse



Nikola Radulaski • 10 avril

- 1. Modules CSS oui, CSS-in-JS non. L'utilisation des modules SCSS vous donne la même encapsulation que vous obtenez avec CSS-in-JS tout en déchargeant le thread principal. Bien sûr, il y a Lerna et des bibliothèques similaires qui compilent en CSS, mais elles ajoutent simplement une complexité inutile au système de construction.
- 2. Si vous utilisez trop la mémorisation, quelque chose ne va pas.
- 3. N'utilisez pas les bibliothèques d'état à moins que votre application en ait vraiment besoin.

2 aime Réponse



Dominique • 11 avril

J'utilise useState pour gérer la plupart de mon état dans mon projet lié à React, mais j'ai besoin d'en savoir plus sur la façon de gérer l'état globalement, comme l'utilisation de redux et autres, quelqu'un peut-il me recommander un bon cours à étudier ?

1 aimer Réponse



Sam Magura ♂ • 11 avril

La [documentation de Redux Toolkit](#) contient de bons tutoriels et guides pour utiliser Redux !

2 aime Réponse



Gabin • 11 avril

L'autre chose à mentionner à propos des réducteurs est d'activer l'état avant l'action (événement). La plupart des développeurs leur écrivent d'abord l'action, ce qui signifie que l'événement peut être déclenché sans condition. Il le mentionne même dans la documentation redux.js.org/style-guide/style-guidelines/#use-the-reducer-to-set-the-state

Aimer Réponse



Gopinath Prasanna • 11 avril

Un nouvel expert JavaScript est né :)

1 aimer Réponse



intermundos • 10 avril

Et le dernier conseil - vous ne passez pas à Vue.

1 aimer Réponse



Jorge Madison • 14 avril

J'ai adoré ce post, j'essaie d'améliorer mon niveau en tant que développeur de réaction et j'ai vraiment aimé les conseils, je le lirai chaque semaine jusqu'à ce que je les ai intériorisés.

1 aimer Réponse



Christian Engel • 26 avril

J'ai trouvé le message principalement énervé, mais l'ai continué jusqu'au "

N'utilisez le rendu du serveur que si vous en avez vraiment besoin".

Ce doit être le pire conseil de tous les temps :(

 1 aimer  Réponse

[Voir la discussion complète \(65 commentaires\)](#)

Code de conduite • Signaler un abus

Lire la suite



Créez un magnifique bouton Connect Wallet avec RainbowKit et React

Anish De - 15 mai



Déboguer n'importe quelle application React en quelques secondes

Chris-15 mai



Présentez-moi sur TypeScript

Ben Halpern -16 mai



JavaScript - Carte

Tanwa Sripan -14 mai

[Communauté de développement](#)— Un réseau social constructif et inclusif pour les développeurs de logiciels. Avec vous à chaque étape de votre voyage.

Construit sur [formulaire](#)- la [Open source](#) logiciel qui alimente [DEV](#) et d'autres communautés inclusives.

Fait avec amour [et Rubis sur rails](#). Communauté DEV © 2016 - 2022.

