

L'organisation d'une application React Native

Table des matières

I. Contexte	3
II. L'organisation des dossiers et fichiers dans une application React Native	3
III. Exercice : Appliquez la notion	8
IV. Les outils pour organiser et maintenir la qualité de son code	9
V. Exercice : Appliquez la notion	11
VI. Essentiel	12
VII. Auto-évaluation	12
A. Exercice final	12
B. Exercice : Défi.....	14
Solutions des exercices	14
Contenus annexes	19

I. Contexte

Durée : 1 h

Environnement de travail : Un carnet pour prendre des notes.

Pré-requis : Quelques notions avec GIT (versioning) et des notions sur le thème des composants React.

Contexte

Le sujet de l'organisation du code dans une application est un sujet très polémique, car, généralement, chaque développeur a son propre avis sur la question. Cependant, avec l'expérience et la pratique, on peut remarquer que certains patterns de développement fonctionnent mieux que d'autres au fur et à mesure qu'un projet prend de l'ampleur et que le nombre de fichiers augmente.

Nous allons voir dans ce module quelques exemples de « bonnes pratiques » qui pourront inspirer et permettre d'alimenter des échanges pendant des ateliers de conception au sujet de l'organisation d'un projet React Native.

Quoi qu'il en soit, c'est un sujet à partager avec son équipe afin de trouver un consensus qui convienne au plus grand nombre. En effet, il est important de garder une cohérence tout au long d'un projet, alors autant que chacun se sente à l'aise avec les règles prises ensemble.

II. L'organisation des dossiers et fichiers dans une application React Native

Objectifs

- Étudier les points clés à prendre en considération dans l'organisation de son code source
- Voir une structure de fichiers permettant de sécuriser le futur de son projet
- Comprendre le découpage en composants d'une interface utilisateur
- Comprendre la spécialisation des composants

Mise en situation

Généralement polémique, le sujet de l'organisation du code dans une application est à aborder en début de projet avec son équipe. En effet, plus on tarde à l'aborder, plus on a de chances d'avoir des divergences dans les pratiques, et donc du code à modifier (on appelle souvent cela « refactor »). Et, qui dit beaucoup de modifications, dit risque de régression, et surtout perte de temps.

Voyons ensemble quelques principes importants à garder à l'esprit.

Ranger ses fichiers : le fonctionnement en modules

L'univers JavaScript, et plus particulièrement celui de NodeJS, pousse vers l'utilisation de modules autonomes. Et ce n'est pas parce que l'on ne crée pas de modules publiés sur npm (le gestionnaire de paquets de NodeJS) qu'il ne faut pas s'intéresser à l'approche modulaire du code.

Concrètement, on peut rencontrer deux grands types de projets au cours de nos développements :

- Les projets qui organisent le code par ce qu'il est (approche sémantique) : par exemple, les composants dans un dossier `components`, les modèles de données avec les autres dans un dossier `models`, etc.
- Les projets qui organisent le code par entités métier : par exemple, nous aurons un module `user` qui contiendra des composants graphiques pour l'affichage des utilisateurs, le modèle de données des utilisateurs, etc. On parle alors de *Feature Driven Development* (« Développement Guidé par les Fonctionnalités »).

Quel est problème rencontré par une approche sémantique ?

Tout d'abord, voici un exemple d'approche sémantique dans l'organisation de ses fichiers.

Exemple Une arborescence par approche sémantique

```

1 src
2 | components
3 |   | UiCard.js
4 |   | UserProfile.js
5 | models
6 |   | CourseModel.js
7 |   | UserModel.js
8 | reducers
9 |   | userReducer.js
10 | services
11 |   | userService.js

```

Tant que la taille d'un projet reste modeste, cette approche peut sembler pertinente. Mais il s'avère qu'au fur et à mesure de l'avancement dans le développement d'un projet, ce genre d'approche tend à montrer ses limites :

- Les imports relatifs de fichiers deviennent de plus en plus longs (`../../../../../../../../components/Coucou.js` par exemple), les fichiers étant dispersés dans une arborescence plutôt verticale où l'on crée des sous-dossiers de sous-dossiers. Il devient donc difficile de naviguer dans cette dernière et de s'y retrouver.
- On augmente le risque de conflits de *versioning* avec Git, car il y a plus de chances que des développeurs touchent les mêmes fichiers/dossiers.
- Les modules ne sont pas clairement identifiables, non exportables à d'autres projets, et on perd pas mal de temps à chercher où se trouvent les choses dans le projet, en particulier lorsqu'on débarque sur un projet en cours.

Bref ! Globalement, cela ne *scale* pas correctement.

Voici maintenant une approche modulaire par fonctionnalité :

Exemple Une arborescence par module

```

1 src
2 | authentication
3 |   | authentication.model.js
4 |   | components
5 |   |   | LoginForm.js
6 |   | services
7 |   |   | login-form-validator.js
8 | homepage
9 |   | components

```

```
10 |       └─ HomeCarousel.js
11 |     └─ user
12 |         └─ components
13 |             └─ badges
14 |                 └─ UserBadge.js
15 |             └─ profile
16 |                 └─ UserAvatar.js
17 |         └─ user.model.js
18 |         └─ user.reducer.js
```

Ici, on constate que chaque fonctionnalité est isolée dans son module et permet potentiellement d'exporter le module sur d'autres projets. Quand on développe avec ce genre d'architecture, on ne perd pas beaucoup de temps à chercher où se trouvent les fichiers et telle ou telle fonctionnalité dans l'application.

On peut également rajouter un fichier texte `README.md` à la racine du module qui spécifie les dépendances du module vis-à-vis d'un autre module. Ou, plus avancé, on peut utiliser un injecteur de dépendances pour le faire à notre place comme le propose le framework Angular¹.

Il est fortement recommandé de s'imposer une structure et de s'y tenir entre membres de l'équipe, sans quoi, chacun fait ce qu'il souhaite et le projet devient impossible à maintenir.

Le nommage des fichiers et leur contenu

De manière générale, ce genre de choix est également à prendre avec son équipe en début de projet. Une pratique commune est de nommer les fichiers en leur donnant un préfixe lié à leur module et à ce qu'ils sont.

Imaginons que nous créons une carte qui affichera un profil utilisateur : nous pourrions la découper comme suit, en plusieurs fichiers.

Exemple

```
1 src
2   └─ profile
3       └─ components
4           └─ ProfileAvatar.js
5           └─ ProfileCard.js
6   └─ ui
7       └─ components
8           └─ UiBadge.js
9           └─ UiText.js
```

L'arborescence de fichiers présentée en exemple permet de séparer les fichiers qui contiennent des éléments spécifiques à une fonctionnalité, de ceux réutilisables dans toute l'application.

En effet, les éléments présents dans `src/profile/components` sont spécifiques à l'affichage du profil utilisateur, tandis que ceux présents dans `src/ui/components` sont des modules généralistes, et donc réutilisables dans d'autres pages de l'application.

Remarque

Remarquons le nommage des fichiers. Sur l'approche module, c'est simple, ils sont préfixés par le nom de leur module. Pratique, n'est-ce pas ?

¹ <https://angular.io/guide/dependency-injection>

Des composants pour l'affichage des écrans

Nous le verrons dans le module dédié à la navigation, mais un certain type de composants permet de représenter l'affichage d'un écran et est utilisé par le routeur pour gérer la navigation de l'application. Ces composants importent d'autres composants graphiques pour l'affichage et sont généralement associés à des métadonnées de navigation.

L'organisation modulaire des composants écran peut se faire selon deux méthodes : les stocker dans un répertoire `screens` à la racine du projet, ou créer un répertoire `screens` pour chacun des modules le nécessitant.

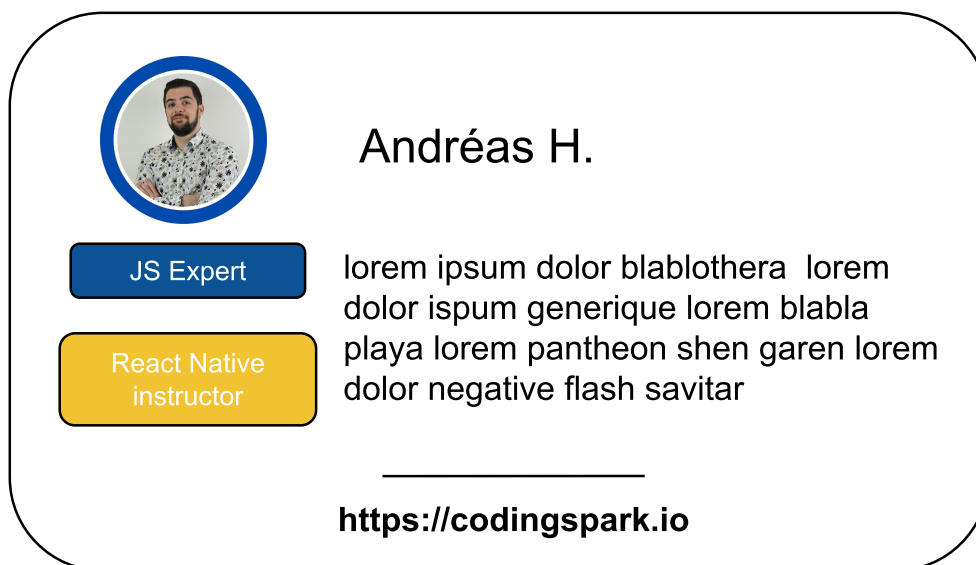
Ainsi, en suivant l'arborescence présentée un peu plus haut, dans le module `profile`, on pourrait avoir un dossier `screens` avec un fichier `ProfileScreen.js` à l'intérieur.

Séparation en conteneurs de logique et composants d'affichage

Une autre pratique très répandue est d'utiliser des composants spécifiques à de la logique (notamment les fameux *Higher Order Components*, ou HOC, dont nous allons parler un peu plus loin) et des composants spécialisés dans l'affichage de données, uniquement alimentés par des props.

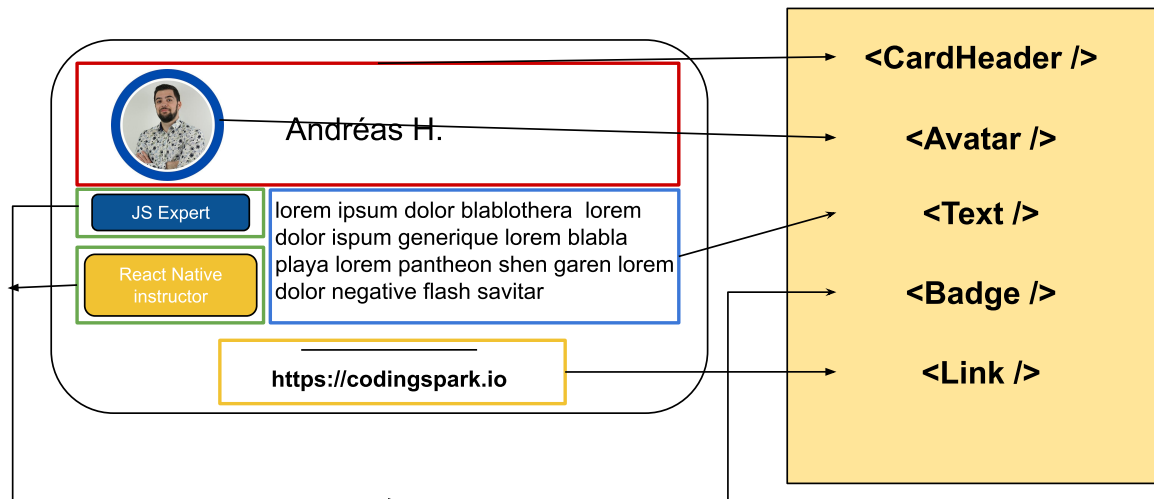
L'idée est d'appliquer le principe de responsabilité unique¹ d'un composant en le spécialisant : de cette manière, on permet une réutilisation maximale du code.

Prenons par exemple l'affichage ci-dessous :



Ici, le composant contenant la logique applicative qui permet d'aller récupérer les données et les informations de la carte (à savoir, le titre, les badges, etc.) pourrait s'appeler `ProfileCard.js`. Ensuite, découpons les composants d'affichage.

¹ https://fr.wikipedia.org/wiki/Principe_de_responsabilit%C3%A9_unique



Ici, chaque élément va générer un composant personnalisé qui pourra éventuellement être réutilisé. D'ailleurs, les composants `Text` et `Badge` se trouvent dans notre module nommé `ui`, dans les exemples d'organisation de fichiers vus précédemment. Ces composants d'affichage ne prennent que des props en paramètres et ne font qu'afficher le contenu donné, sans aucune logique métier à l'intérieur.

On parle de *Dumb components* (« composants bêtes »).

Higher Order Component (HOC)

Un type de composant également très populaire, notamment avant l'apparition des hooks pour les composants fonction, est le *Higher Order Component* (HOC).

Ils sont généralement assez populaires dans les projets. Ce type de composant répond à la problématique de factorisation de code métier s'appliquant au cycle de vie des composants. Ils prennent généralement un composant en paramètre (et une configuration) et lui injectent un certain nombre de props. De cette manière, ils permettent une façon de réutiliser du code assez pratique.

Exemple

```

1 import * as React from 'react';
2 import { Button, View } from 'react-native';
3
4 // Ce composant reçoit en paramètre un composant et retourne un composant.
5 // C'est un HOC, normalement il devrait être dans son propre fichier.
6 function logProps(WrappedComponent) {
7   // Une classe anonyme, oui c'est possible !
8   return class extends React.Component {
9     componentDidMount(prevProps) {
10       console.log('Props actuelles : ', this.props);
11       console.log('Props précédentes : ', prevProps);
12     }
13
14     render() {
15       // Enrobe le composant initial dans un conteneur, sans le modifier. Mieux !
16       return <WrappedComponent {...this.props} />;
17     }
18   };
19 }
20
```

```

21 // Le composant de base qui va être amélioré
22 function MyButton(props) {
23   return <Button title={props.title} onClick={() => console.log('cliqué')} />;
24 }
25
26 // Notre composant qui est emballé par notre HOC, il s'utilise comme notre composant normal vu
   que les props sont passés.
27 const MyEnhancedButton = logProps(MyButton);
28
29 export default function App() {
30   return (
31     <View>
32       <MyEnhancedButton title="Mon bouton personnalisé" />
33     </View>
34   );
35 }

```

Ce composant affiche dans la console JavaScript les valeurs précédentes et actuelles des props de chaque composant qu'il englobe.

Syntaxe À retenir

- Les questions d'architecture logicielle sont à discuter avec son équipe en début de projet, afin que tout le monde s'aligne sur les mêmes pratiques.
- La communauté propose de nombreux patterns de conception et d'organisation du code, qui peuvent servir de socle à l'organisation de fichiers dans un projet : l'approche modulaire du code *Feature Driven Development* ou les *Higher Order Components* en sont un bon exemple.
- Il faut toujours penser à l'évolution du logiciel et à sa maintenance lorsque l'on prend des décisions techniques. Cela permettra de faciliter la reprise en main d'un projet par d'autres développeurs ou de reprendre son propre travail dans plusieurs mois.

Complément

- <https://fr.reactjs.org/docs/higher-order-components.html>
- <https://reactjs.org/docs/thinking-in-react.html>¹
- <https://fr.reactjs.org/docs/higher-order-components.html>²³

III. Exercice : Appliquez la notion

Question

[solution n°1 p.15]

Créons un *Higher Order Component* (composant de haut niveau) nommé `withTodoHttpRequest` qui effectue une requête au moment où il s'instancie (grâce au hook `useEffect`) sur une URL **`https://jsonplaceholder.typicode.com/todos/<todoId>`**.

Il faudra remplacer `<todoId>` par un nombre entre 1 et 10.

Notre composant personnalisé reçoit une props `todoId` qui permet de spécifier l'identifiant du todo à récupérer : il faut l'utiliser en coordination avec l'URL précisée plus haut. Si la props `todoId` change, la requête doit se réexécuter pour afficher les nouvelles données liées aux changements de props.

1 <https://fr.reactjs.org/docs/higher-order-components.html>

2 <https://reactjs.org/docs/thinking-in-react.html>

3 <https://fr.reactjs.org/docs/higher-order-components.html>

Une fois le contenu du todo récupéré, il faut simplement afficher le contenu du JSON retourné dans un composant `Text`, qui peut recevoir en props la couleur que le texte doit prendre (par défaut, le texte sera noir). Pour cela, il est possible d'utiliser l'instruction `JSON.stringify(props, null, 2)`.

Tant que la requête n'a pas reçu de réponse, il faut afficher un loader grâce au composant `ActivityIndicator`¹ afin de faire patienter l'utilisateur.

En indice se trouve le code pour faire la requête HTTP sur une API de test, et il faudra utiliser son résultat dans une clé de state personnalisée grâce à `useState`.

Indice :

La requête HTTP peut être réalisée grâce au code suivant :

```
1 fetch('https://jsonplaceholder.typicode.com/todos/1')
2   .then(response => response.json())
3   .then(json => console.log(json))
4
```

Le retour de la requête HTTP suit le format suivant :

```
1 const result = {
2   "userId": 1,
3   "id": 1,
4   "title": "delectus aut autem",
5   "completed": false
6 }
```

IV. Les outils pour organiser et maintenir la qualité de son code

Objectifs

- Voir le panel d'outils qui s'offrent à nous pour maintenir son code dans un état cohérent
- Observer les bonnes pratiques en matière de vérification de son code

Mise en situation

Nous parlons de structure de fichiers, et plus généralement de bonnes pratiques de code, et, maintenant que nous nous sommes entendus sur un format et des pratiques spécifiques, il faut permettre à chacun d'en avoir connaissance, de s'en rappeler et de pouvoir les appliquer le plus simplement possible. Il peut être un peu lassant de devoir repasser derrière son code pour rajouter de l'indentation ou réorganiser des blocs de code afin de concorder parfaitement à une spécification. Pour cela, il existe des outils très pratiques, que nous allons voir.

L'utilisation d'un linter et/ou d'un formater

Pour l'organisation du code, il existe deux types de règles :

- Les règles de formatage : par exemple, s'il faut mettre ou non un point-virgule à la fin d'une instruction, la longueur maximum d'une ligne de code...
- Les règles de qualité du code : par exemple, ne pas avoir de variable déclarée ou importée qui soit non utilisée dans le code, ou encore nommer les clés de ses objets par ordre alphabétique.

Certains outils influent plutôt sur l'un ou l'autre type de règle.

Nous allons voir dans ce cours les deux principaux outils utilisés dans la communauté JavaScript pour répondre à ces besoins. Ce sont des outils gratuits et *open source*, présents sur de nombreux projets. Le premier outil est un « formateur » qui se nomme Prettier.

¹ <https://reactnative.dev/docs/activityindicator>

Définition Prettier

Il permet, en se basant sur des règles qu'on lui spécifie, de soulager la mise en place des règles de types formatage. Pour cela, Prettier va réécrire le code de nos fichiers, intégralement, de manière cohérente par rapport aux règles définies. De cette manière, il deviendra impossible pour le développeur de manquer à une de ces règles.

Définition ESLint

Le second outil est un linter de code qui se nomme ESLint.

Cet outil applique des règles qui seraient pour la plupart susceptibles de causer un dysfonctionnement dans l'application. Par exemple, ESLint possède une règle capable de détecter que l'on utilise une variable qui n'existe pas dans le scope du fichier, ce qui déclencherait un plantage de l'application. Il propose d'auto-corriger ces erreurs quand c'est possible, ou il notifie des alertes/erreurs et nous invite à les corriger nous-mêmes.

ESLint permet également de programmer ses propres règles de validation, bien que la plupart des scénarios soient déjà disponibles dans les règles de base et/ou dans les modules *open source* de la communauté.

Installation des outils

Pour installer ces outils, il faudra dans un premier temps les ajouter en dépendances de développement du projet. Pour cela, on utilise les commandes suivantes :

- `npm install --save-dev --save-exact prettier:--save-exact` sert à verrouiller la version afin d'éviter de désynchroniser notre formateur avec des changements dans les valeurs par défaut de sa configuration.
- `npm install --save-dev eslint`

On initialise ensuite les outils :

- `echo {} > .prettierrc.json` pour le fichier de configuration de Prettier
- `npx eslint --init` pour initialiser ESLint, qui va nous poser un certain nombre de questions très simples pour nous aider à générer notre fichier de configuration, que l'on pourra modifier par la suite.

Remarque Qu'est-ce que npx ?

`npx` est livré avec `npm` et permet d'exécuter les outils installés localement. Nous allons laisser de côté la partie `npx` pour le moment, mais, en recherchant sur le Web, on obtient une documentation très explicite à son sujet.

Note : si l'on oublie d'installer Prettier en premier, `npx` téléchargera temporairement la dernière version. Ce n'est pas une bonne idée lorsque l'on utilise Prettier, car nous changeons la façon dont le code est formaté dans chaque version ! Il est important d'avoir une version verrouillée de Prettier dans notre `package.json`. Et c'est aussi plus rapide.

Une fois cela fait, on peut utiliser ces outils pour interagir avec notre code, on peut mettre ces scripts dans des scripts `npm` :

- `npx eslint <nom du fichier|ou dossier>` permet de linter lesdits fichiers
- `npx prettier --write .` (ou spécifier le dossier à la place du `.`, qui est le dossier courant) pour faire travailler Prettier

Pour aller plus loin, il est possible d'améliorer la configuration de son ESLint en utilisant par exemple les règles décrites dans cette page¹ ; il est également possible d'utiliser le plugin React².

1 <https://www.npmjs.com/package/eslint-plugin-react-native>

2 <https://www.npmjs.com/package/eslint-plugin-react>

Bonnes pratiques Git et pull requests

Généralement, les projets de code utilisent un système de *versioning* du code appelé Git. Pour plus d'informations à ce sujet, on peut se renseigner *ici* (cf. p.19).

(cf. p.19) Git permet de gérer des `commits`, qui sont des regroupements d'instructions d'ajout et de retrait de ligne de code dans un fichier. Le développeur crée donc une fonctionnalité, puis versionne le changement de code qu'il a effectué grâce à l'outil en créant un commit avec un message qui identifie la modification. Pour plus d'informations, il est possible de se référer à ce site¹ qui explique très bien le fonctionnement de Git pour les débutants.

- Généralement, il est préconisé de faire un commit par fonctionnalité indépendante et autonome : c'est-à-dire que, si l'on retire le commit de la fonctionnalité, elle disparaît en intégralité et ne laisse pas de reliquat qui pourrait causer des problèmes.
- De la même manière, il faudrait également s'assurer que, lorsqu'un commit est fait, la fonctionnalité est autonome et n'a pas besoin de plus de code que ce dont est composé le commit. Ainsi, on s'assure que, si l'on retire un commit, on aura une application fonctionnelle, quitte à ce qu'il manque une fonctionnalité, mais cela ne plantera pas.
- Pour le message du commit, il faut être explicite afin de pouvoir identifier facilement ce que contient le commit sans avoir forcément à lire l'intégralité du code qui le compose. Il est pertinent de mettre l'identifiant qui permet de faire le rapprochement avec la fonctionnalité dans le logiciel de gestion de projet.
- Enfin, une bonne pratique est d'utiliser des *pull requests* (ou *merge requests*). C'est un procédé qui permet de faire valider son code par d'autres développeurs de son équipe, avant de le rapatrier sur le tronc commun. Cet article explique bien tout ce fonctionnement : <https://www.atlassian.com/fr/git/tutorials/making-a-pull-request>.

Syntaxe À retenir

- Afin de s'assurer de la mise en place des choix d'organisation du code de notre application, il existe des outils qui permettent aux développeurs de se faciliter la vie et d'améliorer leur expérience de développement.
- Pour cela, on utilise généralement, et a minima, deux outils :
 - Un formateur de code, qui permet de corriger les erreurs de style et de formatage en réécrivant automatiquement le code selon le bon fichier de configuration défini,
 - Un linter de code qui se charge de signaler de potentiels problèmes qui pourraient causer des plantages de l'application.
- Enfin, on applique les bonnes pratiques de Git, afin d'avoir du code facilement analysable par ses collègues afin de se faire conseiller et de réduire le risque de régression ou d'introduction d'erreurs en production.

Complément

- <https://eslint.org/>
- <https://prettier.io/>
- <https://www.atlassian.com/fr/git/tutorials/making-a-pull-request>

Exercice : Appliquez la notion

[solution n°2 p.16]

Exercice

¹ <https://www.atlassian.com/fr/git/tutorials/saving-changes>

Pourquoi utilise-t-on `--save-exact` pour l'installation de Prettier ?

- ☐ Parce que c'est la seule manière de procéder
- ☐ Parce que cela permet de figer la version et d'éviter des effets indésirables liés à un changement de configuration par défaut de Prettier
- ☐ Parce que Prettier doit avoir la priorité sur d'autres formateurs de code

Exercice

Il est possible d'utiliser ces outils en ligne de commande dans des scripts npm.

- ☐ Vrai
- ☐ Faux

Exercice

On peut créer ses propres règles de lint avec ESLint.

- ☐ Vrai
- ☐ Faux

VI. Essentiel

React Native n'impose pas de structure particulière et se base sur des règles semblables à celles de React Web ou de n'importe quel projet informatique.

Il faut dialoguer avec son équipe et trouver un consensus sur les pratiques à appliquer ensemble, en se basant sur les bonnes pratiques de la communauté et des outils faits pour faciliter la mise en place de ces bonnes pratiques, tels que les formateurs ou les linters.

Une bonne approche universelle est une approche modulaire pilotée par les fonctionnalités : cela se marie parfaitement avec tous les logiciels liés à la réalisation d'un projet, comme un logiciel de suivi de projet qui identifie ces fonctionnalités, ou bien à Git, qui permet de versionner ces fonctionnalités en empilant des commits.

VII. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°3 p.16]

Exercice

En utilisant une structure de fichier en approche sémantique...

- ☐ On obtient un projet qui est facile à maintenir dans le temps
- ☐ On a un projet bien organisé tant que le projet reste petit
- ☐ On a un projet bien organisé seulement quand il devient grand

Exercice

En utilisant une structure de fichier en modules...

- ☐ On a un projet bien organisé du début à la fin, c'est un projet qui scale
- ☐ On obtient un projet qui est facile à maintenir dans le temps
- ☐ On a un projet bien organisé seulement quand il devient grand

Exercice

Pour ranger et nommer ses composants, que faut-il idéalement faire ?

- ☐ Préfixer le nom du fichier par le nom du module
- ☐ Suffixer le nom du fichier par le nom du module
- ☐ Donner un nom générique qui n'identifie pas de manière précise l'utilité du composant
- ☐ Le placer dans un dossier `components` propre à un module
- ☐ Le placer dans un dossier `components` global

Exercice

Que permet un *Higher Order Components* (HOC) ?

- ☐ Il permet de factoriser du code qui pré-remplit certaines propriétés d'un composant sans avoir à le faire soi-même
- ☐ Il permet d'extraire de la logique de code réactive aux cycles de vie d'un composant dans un composant abstrait, qui passe des props liées à cette logique à un composant enveloppé
- ☐ Il permet de cacher conditionnellement le composant enveloppé si une condition externe n'est pas remplie

Exercice

Il est préférable de...

- ☐ Séparer la logique conditionnelle et le code métier de l'affichage d'un composant
- ☐ Regrouper le code métier, la logique conditionnelle et l'affichage dans un même composant

Exercice

L'outil Prettier applique...

- ☐ Des règles de formatage
- ☐ Des règles de qualité du code, notamment celles pouvant causer des plantages en production

Exercice

ESLint permet de corriger automatiquement certains types d'erreurs.

- ☐ Vrai
- ☐ Faux

Exercice

Que peut faire l'exécutable `npx` ?

- ☐ Il peut lancer des outils installés localement (`node_modules/.bin`)
- ☐ Il télécharge les paquets localement et ajoute une ligne à la clé `dependencies` dans le `package.json`
- ☐ Il télécharge les outils s'il n'existent pas localement dans le dossier global de npm pour la session de l'utilisateur courant

Exercice

Un commit au sens de Git, c'est...

- ☐ Un regroupement d'instructions de modifications de code dans des fichiers et dossiers (ajout et suppression)
- ☐ Un petit programme qui vérifie que la syntaxe est conforme à son fichier de configuration
- ☐ Un utilitaire qui permet de corriger automatiquement les erreurs de programmation

B. Exercice : Défi

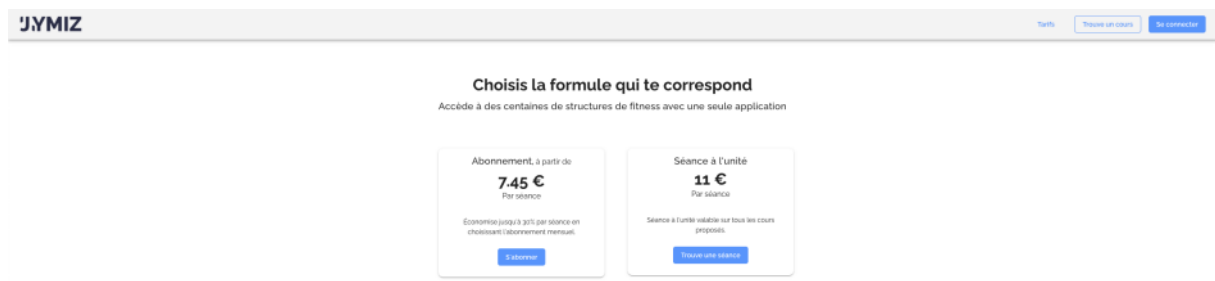
Imaginez un découpage et un rangement pour les dossiers et fichiers de cette interface graphique proposant un abonnement sportif en utilisant la structure en modules évoquée dans ce cours.

Question

[solution n°4 p.18]

Pour achever cet exercice, il faudra rédiger et livrer :

- Une arborescence de fichiers contenant les composants nécessaires pour répliquer l'interface de la capture d'écran ci-dessous,
- Pour chacun des composants : son nom, sa fonction et les props potentielles qu'il pourrait recevoir si cela s'avère nécessaire dans le cadre d'une réutilisation de code.



Solutions des exercices

p.8 Solution n°1

```

1 import * as React from 'react'
2 import {
3   ActivityIndicator,
4   Button,
5   View,
6   StyleSheet,
7   Text,
8 } from 'react-native'
9
10 // Ce composant reçoit en paramètre un composant et retourne un composant.
11 // C'est un HOC, normalement il devrait être dans son propre fichier.
12 function withTodoHttpRequest(WrappedComponent) {
13   return function HOCRequeteTODO(props) {
14     const [isLoading, setLoading] = React.useState(true)
15     const [todo, setTodo] = React.useState(null)
16
17     // Cet effet va être appelé à chaque fois que la valeur de props.todoId change et faire
    une requête
18     React.useEffect(() => {
19       setLoading(true)
20       fetch('https://jsonplaceholder.typicode.com/todos/' + props.todoId)
21         .then((response) => response.json())
22         .then((json) => setTodo(json))
23         .catch((e) => {console.error(e)})
24         .finally(() => setLoading(false))
25     }, [props.todoId])
26     // On affiche notre composant ou un état de chargement
27     return isLoading ? (
28       <ActivityIndicator />
29     ) : (
30       <WrappedComponent {...props} todo={todo} />
31     )
32   }
33 }
34
35 // Ce composant affiche une représentation JSON de notre TODO uniquement
36 function TodoElement(props) {
37   return (
38     <Text style={{ color: props.color || 'black' }}>
39       {JSON.stringify(props.todo, null, 2)}
40     </Text>
41   )
42 }
43
44 // Ce composant permet de faire faire la requête via notre HOC à notre composant
45 const EnhancedTodoElement = withTodoHttpRequest(TodoElement)
46
47 export default function App() {
48   return (
49     <View style={appStyles.globalContainer}>
50       <EnhancedTodoElement todoId={1} color="blue" />
51       <EnhancedTodoElement todoId={2} color="red" />
52       <EnhancedTodoElement todoId={3} />
53     </View>
54   )


```

```
55 }
56
57 const appStyles = StyleSheet.create({
58   globalContainer: {
59     flex: 1,
60   },
61 })
```

Exercice p. 11 Solution n°2


Exercice

Pourquoi utilise-t-on `--save-exact` pour l'installation de Prettier ?

- ☐ Parce que c'est la seule manière de procéder
- ☒ Parce que cela permet de figer la version et d'éviter des effets indésirables liés à un changement de configuration par défaut de Prettier
- ☐ Parce que Prettier doit avoir la priorité sur d'autres formateurs de code
-  C'est simplement une sécurité pour éviter que de nouveaux changements de configuration par défaut introduits par des mises à jour viennent polluer le répertoire de travail.


Exercice

Il est possible d'utiliser ces outils en ligne de commande dans des scripts npm.

- ☒ Vrai
- ☐ Faux
-  Étant donné qu'ils sont installés localement, il est tout à fait possible de les utiliser en créant des scripts personnalisés dans le fichier `package.json`. Par exemple, on pourrait appeler ce script `lint-and-format: eslint src/ && prettier --write src/`.

Exercice

On peut créer ses propres règles de lint avec ESLint.

- ☒ Vrai
- ☐ Faux
-  En effet, on peut créer ses propres règles de validation ou modifier et étendre toutes celles déjà disponibles. Notamment celles issues de la communauté open source.

Exercice p. 12 Solution n°3

Exercice

En utilisant une structure de fichier en approche sémantique...

- ☐ On obtient un projet qui est facile à maintenir dans le temps
- ☒ On a un projet bien organisé tant que le projet reste petit
- ☐ On a un projet bien organisé seulement quand il devient grand

- Q L'utilisation d'une approche sémantique au niveau global complexifie la maintenance d'un projet, du fait qu'on identifie plus difficilement où se trouvent les fonctionnalités corrélées entre elles.

Exercice

En utilisant une structure de fichier en modules...

- ☒ On a un projet bien organisé du début à la fin, c'est un projet qui scale
- ☐ On obtient un projet qui est facile à maintenir dans le temps
- ☐ On a un projet bien organisé seulement quand il devient grand

- Q L'utilisation d'une approche modulaire au niveau global permet de faciliter la réutilisation du code et l'export de modules à d'autres projets. C'est également pratique, car on regroupe des éléments de sémantiques différentes, mais à forte relation entre eux dans des dossiers proches.

Exercice

Pour ranger et nommer ses composants, que faut-il idéalement faire ?

- ☒ Préfixer le nom du fichier par le nom du module
- ☐ Suffixer le nom du fichier par le nom du module
- ☐ Donner un nom générique qui n'identifie pas de manière précise l'utilité du composant
- ☒ Le placer dans un dossier `components` propre à un module
- ☐ Le placer dans un dossier `components` global

- Q Le préfixe permet d'identifier rapidement le fichier et sa position, notamment en utilisant les patterns GLOB de recherche inclus dans les IDE. C'est d'ailleurs pour cela qu'on lui donne un nom qui a du sens et qu'on le place dans un dossier `components` propre à son module. Quitte à faire un module spécifique pour les composants partagés, qu'on pourrait nommer `shared`.

Exercice

Que permet un *Higher Order Components* (HOC) ?

- ☒ Il permet de factoriser du code qui pré-remplit certaines propriétés d'un composant sans avoir à le faire soi-même
- ☒ Il permet d'extraire de la logique de code réactive aux cycles de vie d'un composant dans un composant abstrait, qui passe des props liées à cette logique à un composant enveloppé
- ☒ Il permet de cacher conditionnellement le composant enveloppé si une condition externe n'est pas remplie

- Q On peut en effet faire beaucoup de choses avec un HOC : concrètement, c'est un moyen de factoriser du code lié à des composants qui seraient réutilisables.

Exercice


Il est préférable de...

- ☒ Séparer la logique conditionnelle et le code métier de l'affichage d'un composant
- ☐ Regrouper le code métier, la logique conditionnelle et l'affichage dans un même composant

- Q En effet, on applique le principe de responsabilité unique. De cette manière, on peut donc réutiliser beaucoup plus de code efficacement. De plus, on identifie plus facilement quels composants génèrent des effets de bord.


Exercice

L'outil Prettier applique...

- ☒ Des règles de formatage
- ☐ Des règles de qualité du code, notamment celles pouvant causer des plantages en production
-  Prettier est un formateur qui réécrit votre code pour respecter certains standards. Par contre, les règles de qualité du code sont appliquées par un Linter comme ESLint, notamment celles pouvant causer des plantages en production.


Exercice

ESLint permet de corriger automatiquement certains types d'erreurs.

- ☒ Vrai
- ☐ Faux
-  Selon la nature de l'erreur, ESLint est à même de corriger une erreur directement en altérant le code. Pour cela, il faut lui passer l'argument `--fix`.


Exercice

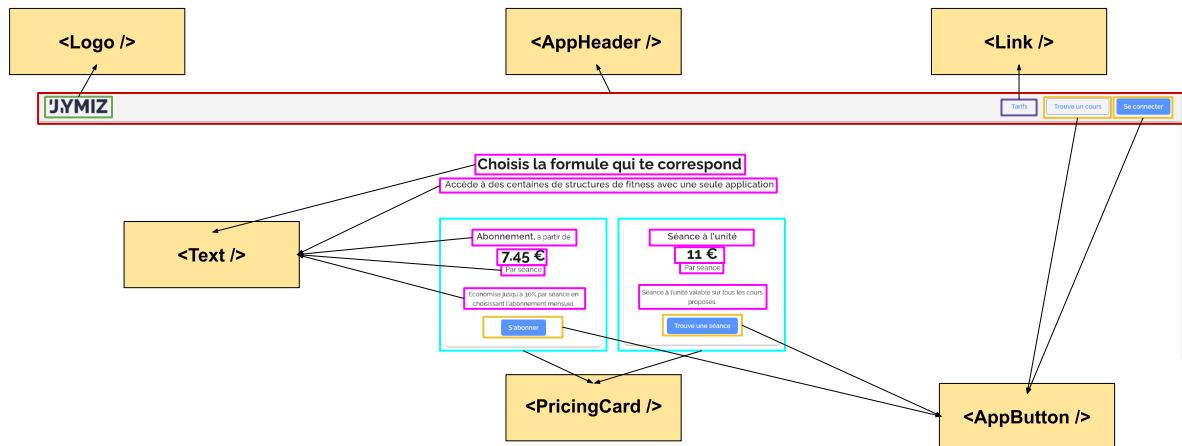
Que peut faire l'exécutable `npm` ?

- ☒ Il peut lancer des outils installés localement (`node_modules/.bin`)
- ☐ Il télécharge les paquets localement et ajoute une ligne à la clé `dependencies` dans le `package.json`
- ☒ Il télécharge les outils s'il n'existent pas localement dans le dossier global de npm pour la session de l'utilisateur courant
-  `npm` est livré avec npm et permet de lancer les outils installés localement, en plus de les télécharger dans le répertoire global de la session utilisateur s'il ne les trouve pas localement.

Exercice

Un commit au sens de Git, c'est...

- ☒ Un regroupement d'instructions de modifications de code dans des fichiers et dossiers (ajout et suppression)
- ☐ Un petit programme qui vérifie que la syntaxe est conforme à son fichier de configuration
- ☐ Un utilitaire qui permet de corriger automatiquement les erreurs de programmation
-  Un commit est un jalon logiciel horodaté qui regroupe un ensemble d'ajouts, modifications ou suppressions de lignes de code. On peut également joindre un message pour décrire ce jalon, qui fait partie d'un outil de gestion de versions nommé Git.



- `AppHeader` peut prendre une props `logo` qui reçoit le composant `Logo` et une props `menuItems` qui prend un tableau d'éléments et qui contiendra un `Link` et deux `AppButtons`.
- `Text` prend en props `bold` et `italic` qui sont des booléens et `size` qui peut être une taille en pixels ou une variante qui sera mappée dans le composant.
- `AppButton` prend une props `title`, une props `variant` qui permet de faire le bouton clair ou foncé et une props `onClick` qui est une fonction indiquant quoi faire lorsqu'on clique sur le bouton.
- `PricingCard` prend une props `title`, une props `price`, une props `description` et une props `callToAction` qui prend un composant, ici un `AppButton`.

En ce qui concerne l'arborescence, on pourrait avoir quelque chose du style :

```

1 src
2 |— miscellaneous
3 |   |— components
4 |     |— AppHeader.js
5 |— pricing
6 |   |— components
7 |     |— PricingCard.js
8 |   |— screen
9 |     |— PricingScreen.js
10 |— ui
11 |   |— components
12 |     |— AppButton.js
13 |     |— Link.js
14 |     |— Logo.js
15 |     |— Text.js

```

Contenus annexes

1. Les avantages de Git

Objectif

- Connaître le DVCS Git et ses avantages

Mise en situation

Lorsqu'il s'agit de choisir un VCS, il est important de connaître les avantages et inconvénients de chacun d'entre eux, ainsi qu'au type auquel ils appartiennent. De plus, il sera sans doute utilisé durant toute la vie du projet. C'est pourquoi nous allons voir ensemble les avantages de l'utilisation de Git par rapport aux autres VCS.

Un DVCS

Git est un DVCS, pour *Distributed Version Control System*. Cela veut dire qu'il peut être utilisé aussi bien en local qu'avec un serveur distant, puisque sa base de données est décentralisée. Par conséquent, il est très différent des CVCS tels que CVS ou Subversion (SVN).

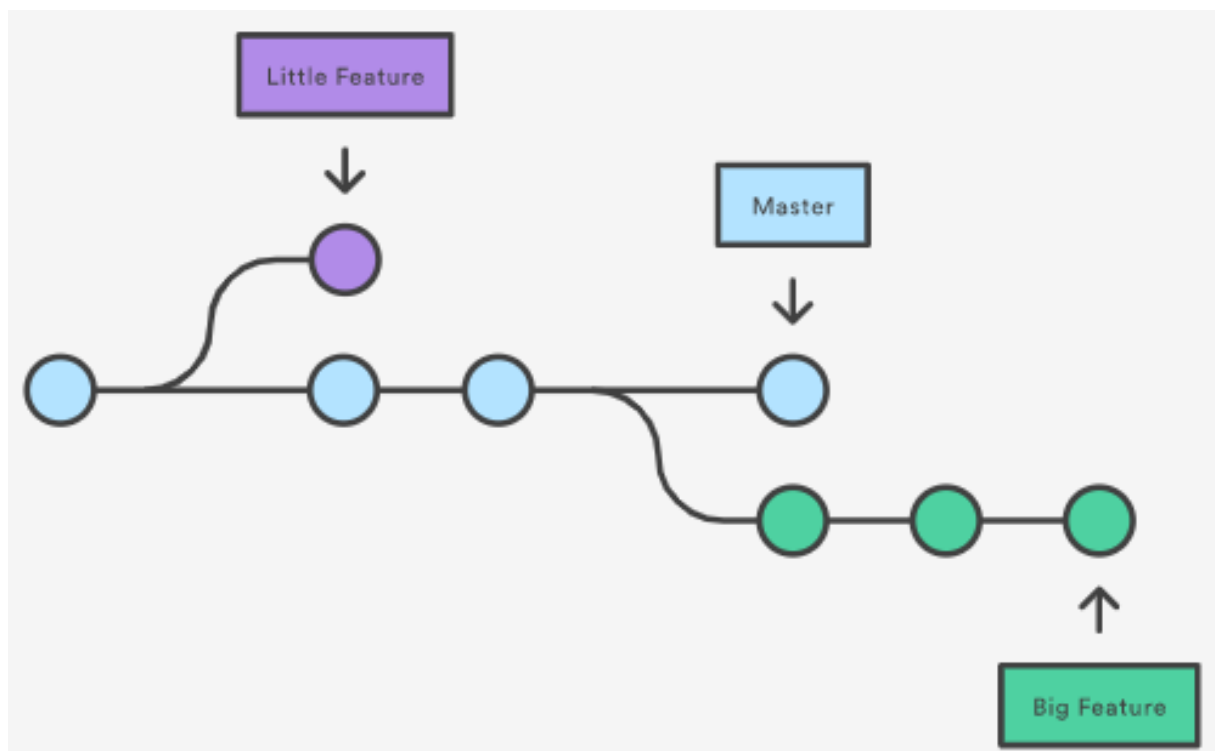
Un outil rapide

Étant donné son architecture distribuée, Git dispose d'une grande rapidité, puisque la majorité des opérations se font en local sur notre machine. C'est-à-dire que nous ne sommes pas tributaire d'une bonne connexion Internet ou d'un serveur centralisé puissant capable de gérer des milliers de requêtes simultanées.

Un des autres systèmes qui rend Git rapide est le fait qu'il travaille avec des instantanés plutôt qu'avec des différences. Là où d'autres VCS stockent la différence d'un fichier entre deux de ses états, Git stocke le fichier en entier lorsqu'il a été modifié, et simplement une référence vers l'état précédent s'il n'a pas changé. Cela le rend très rapide lorsqu'il faut effectuer des recherches dans l'historique ou faire des différences entre deux versions éloignées dans le temps.

Travail en parallèle

Grâce à son système de branches, Git permet à plusieurs développeurs de travailler en simultané sur plusieurs fonctionnalités. La gestion de la fusion de ces branches, le fait d'intégrer une fonctionnalité développée dans un code existant, est largement simplifiée par rapport à d'autres VCS.



Performant pour de grands projets

Git est capable de gérer des projets de grande envergure, tels que le développement du noyau Linux. Celui-ci a initialement été conçu en grande partie par Linus Torvald en 2005 pour gérer ce projet. Il n'a cessé depuis ce jour d'être amélioré et trouve parmi ses utilisateurs des entreprises telles que Google, Facebook ou Microsoft.

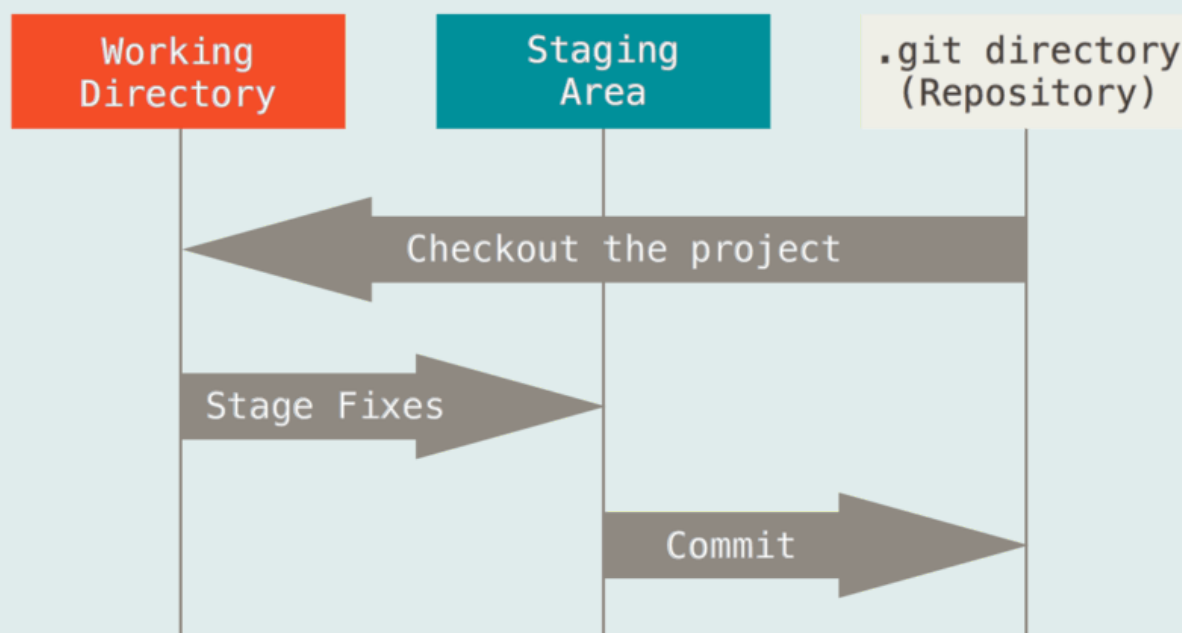
Intégrité des données

En général, Git ne fait qu'ajouter des données. De ce fait, il est assez simple de revenir en arrière lorsque vous pensez avoir cassé quelque chose. Git possède de nombreux garde-fous, qui vous avertiront la plupart du temps lorsque vous êtes sur le point de faire des opérations qui peuvent être dangereuses.

Fondamental Les trois états de fichiers

Git gère les modifications apportées aux fichiers selon trois états : modifié, validé et indexé.

- **Modifié** signifie que le fichier a été modifié en local, mais que ses modifications n'ont pas encore été ajoutées à la base de données locale.
- **Indexé** signifie que la version du fichier a été ajoutée au prochain lot de modifications qui seront intégrées à la base de données locale.
- **Validé** signifie que la version du fichier est présente dans la base de données locale.



Syntaxe À retenir

- Git est un système de gestion de version distribué, développé il y a 15 ans pour le projet du noyau Linux. Il a été depuis continuellement amélioré et convient à tous types de projet, même les plus importants.
- Sa base de données locale et son système de gestion des instantanés le rendent très performant.
- Son système de branches et les trois états de fichiers lui permettent d'être robuste lorsqu'il s'agit de travailler sur un projet, que ce soit seul ou à plusieurs.

Complément

Une rapide histoire de Git¹

Rudiments de Git²

1 <https://git-scm.com/book/fr/v2/D%C3%A9marrage-rapide/Une-rapide-histoire-de-Git>

2 <https://git-scm.com/book/fr/v2/D%C3%A9marrage-rapide/Rudiments-de-Git>