

React Native



# Cours Cross-Platform

Amina Abdelkafi

# Notre Objectif

<https://www.figma.com/file/yeok5btXQmVKR2LUtplygX/Cours?node-id=0%3A1>

- Développer une application mobile en Cross Platform
- respecter les règles de nommage
- respecter les règles de composition des composants
- respecter les bonnes pratiques sur la qualité du code
- travailler en équipe
- Utiliser le gestionnaire de version git
- respecter git flow

# Introduction

## Développement d'applications cross-platform vs natives

Auparavant, les développeurs étaient contraints de développer des applications spécifiques ou natives pour des appareils spécifiques, des systèmes d'exploitation et toutes les plates-formes sur lesquelles ils devaient être construits.

Au fur et à mesure que le marché et l'écosystème des applications mobiles se développent, des nouvelles approches sont apparues, notamment la création de la même application pour différentes plates-formes. C'est ainsi que le développement cross-platform a émergé.

Le développement cross-platform est le processus de création d'une application(généralement une application mobile) qui peut être déployée sur plusieurs plateformes à partir d'un code source unique.

# Introduction

## Qu'est-ce que React?

React, ou ReactJS, est une bibliothèque JavaScript open-source permettant de construire des interfaces utilisateur (UI) spécifiquement pour les applications à **page unique (SPA)**.

Son objectif principal est de simplifier le processus de développement et de construction des applications web rapides, évolutives et simples.

En React, on utilise l'extension JSX (ou TSX pour TypeScript).

Donc JSX est une extension de syntaxe pour JavaScript. Il a été écrit pour être utilisé avec React. Le code JSX ressemble beaucoup au HTML.

### **Que signifie "extension de syntaxe" ?**

Dans ce cas, cela signifie que JSX n'est pas un JavaScript valide. Les navigateurs Web ne peuvent pas le lire !

Si un fichier JavaScript contient du code JSX, ce fichier devra être compilé . Cela signifie qu'avant que le fichier n'atteigne un navigateur Web, un [compilateur](#) (babel) JSX traduira n'importe quel JSX en JavaScript normal.

### **Exemple: react class component**



<https://app.components.studio/edit/9PfKNQ5cdTLgrgBxQswH/srC/index.tsx?p=stories>

4

### **Exemple: react functional component**



<https://app.components.studio/edit/79yzXXenHCEX7o5LguVo/srC/index.tsx?p=stories>

## React component without props

```
// Functional Components
export const MyComponentClass = () => {
  return <h1>Hello world</h1>;
}

// Works the same either way:
ReactDOM.render(
  <MyComponentClass />,
  document.getElementById('app')
);
```

- *Les composants de fonction* sont des composants React définis comme des fonctions JavaScript
- Les composants de la fonction doivent retourner JSX
- Les composants fonctionnels peuvent accepter un `props` paramètre.

## React component with props

```
// Functional Components
export const MyComponentClass = (props) => {
  return <h1>Hello world {props.name} </h1>;
}

// Works the same either way:
ReactDOM.render(
  <MyComponentClass name="Arnaud" />,
  document.getElementById('app')
);
```

## Pourquoi utiliser Hooks?

Les React Hooks sont des fonctions qui nous permettent de gérer l'état interne des composants et de gérer les effets secondaires post-rendering directement à partir de nos **composants fonctionnels**.

React propose un certain nombre de Hooks intégrés.

Quelques-uns d'entre eux incluent useState(), useEffect(), useContext(), useReducer() et useRef(). Voici [la liste complète dans les docs](#).

## Exemple



<https://gist.github.com/amina-tekab/ccfdccb0df8c9e0b8bcb2d1aa9d99b0f>

## useState()

`useState()` est une fonction JavaScript définie dans la bibliothèque React. Lorsque nous appelons cette fonction, elle renvoie un tableau avec deux valeurs :

- *current state* - la valeur actuelle de cet état
- *state setter* - une fonction que nous pouvons utiliser pour mettre à jour la valeur de cet état

Parce que React renvoie ces deux valeurs dans un tableau, nous pouvons les affecter à des variables locales.

```
import React, { useState } from "react";

function Toggle() {
  const [toggle, setToggle] = useState();

  // Initialize State
  // const [toggle, setToggle] = useState("On");
  const handleChange = (event) => {
    setToggle("Off");
  }

  return (
    <div>
      <p>The toggle is {toggle}</p>
      <button onClick={() => setToggle("On")}>On</button>
      <button onClick={() => setToggle("Off")}>Off</button>
      <button onClick={handleChange}>Off: outside jsx </button>
    </div>
  );
}
```

## Set From Previous State

### useState()

`useState()` est une fonction JavaScript définie dans la bibliothèque React. Lorsque nous appelons cette fonction, elle renvoie un tableau avec deux valeurs :

- *current state* - la valeur actuelle de cet état
- *state setter* - une fonction que nous pouvons utiliser pour mettre à jour la valeur de cet état

Parce que React renvoie ces deux valeurs dans un tableau, nous pouvons les affecter à des variables locales.

```
import React, { useState } from 'react';
export default function QuizNavBar({ questions }) {
  const [questionIndex, setQuestionIndex] = useState(0);
  // define event handlers
  const goBack = () => setQuestionIndex(prevQuestionIndex => prevQuestionIndex - 1);
  const goToNext = () => setQuestionIndex(prevQuestionIndex => prevQuestionIndex + 1);
  // determine if on the first question or not
  const onLastQuestion = questionIndex === questions.length - 1;
  const onFirstQuestion = questionIndex === 0;
  return (
    <nav>
      <span>Question #{questionIndex + 1}</span>
      <div>
        <button disabled={onFirstQuestion} onClick={goBack}> Go Back</button>
        <button onClick={goToNext} disabled={onLastQuestion}> Next Question</button>
      </div>
    </nav>
  );
}
```



## useState()

`useState()` est une fonction JavaScript définie dans la bibliothèque React. Lorsque nous appelons cette fonction, elle renvoie un tableau avec deux valeurs :

- *current state* - la valeur actuelle de cet état
- *state setter* - une fonction que nous pouvons utiliser pour mettre à jour la valeur de cet état

Parce que React renvoie ces deux valeurs dans un tableau, nous pouvons les affecter à des variables locales.

## Arrays in State



<https://gist.github.com/amina-tekab/c50039c48b27ed096b12fe959a5bf29b>

```
const [selected, setSelected] = useState([]);
```

## Object in State



<https://gist.github.com/amina-tekab/ab60b36e8c063746f782bbf432beb3d32>

```
const [profile, setProfile] = useState({});
```

- Avec React, nous fournissons des modèles de données statiques et dynamiques à JSX pour afficher une vue à l'écran
  - Utilisez des Hooks pour "s'accrocher" à l'état des composants internes pour gérer les données dynamiques dans les composants de fonction
  - Nous utilisons le State Hook en utilisant le code ci-dessous :
    - `currentState` pour référencer la valeur actuelle de l'état
    - `stateSetter` pour référencer une fonction utilisée pour mettre à jour la valeur de cet état
    - l' `initialState` argument pour initialiser la valeur de l'état pour le premier rendu du composant
- ```
const [currentState, stateSetter] = useState( initialState );
```
- Appeler les setters d'état dans les gestionnaires d'événements (handle event)
  - Définissez des gestionnaires d'événements (handle event) simples en ligne avec nos écouteurs d'événements JSX et définissez des gestionnaires d'événements complexes (handle event) en dehors de notre JSX
  - Utiliser une fonction de rappel d'état setter lorsque notre valeur suivante dépend de notre valeur précédente
  - Utilisez des tableaux et des objets pour organiser et gérer les données associées qui ont tendance à changer ensemble
  - Diviser l'état en plusieurs variables plus simples au lieu de tout jeter dans un seul objet d'état

## useEffect()

useEffect() est utilisé pour appeler une autre fonction qui fait quelque chose pour nous, donc rien n'est renvoyé lorsque nous appelons la `useEffect()` fonction.

Le premier argument passé à la `useEffect()` est **Callback Function** que nous voulons que React appelle après chaque rendu de ce composant.

Si nous voulons appeler notre effet uniquement après le premier rendu, nous passons un tableau vide à `useEffect()` comme deuxième argument. Ce deuxième a

```
import React, { useState, useEffect } from 'react';

export default function PageTitle() {
  const [name, setName] = useState("");
  useEffect(() => {
    document.title = `Hi, ${name}`;
  }, [name]);
  return (
    <div>
      <p>Use {name} input field below to rename this page!</p>
      <input
        onChange={({target}) => setName(target.value)}
        value={name}
        type='text' />
    </div>
  );
}
```

```
useEffect(() => {  
    // Called only once  
}, []);
```

```
useEffect(() => {  
    // It will be called before unmounting  
    return () => [  
        // It will be called while unmounting  
    ];  
}, []);
```

```
useEffect(() => {  
    // Called everytime  
});
```

```
useEffect(() => {  
    // pass dependencies  
}, [dependencies]);
```

## Qu'est-ce que React Native ?

React Native est le petit frère de React. Comme React, React Native est également basé sur JavaScript mais est conçu pour construire des applications mobiles natives avec des composants réutilisables. La passerelle React Native est chargée d'effectuer le rendu natif des API en Java (pour Android) et Swift (pour iOS). Les applications sont rendues avec des composants d'interface utilisateur mobiles au lieu de webviews et fonctionnent de la même manière que les autres applications mobiles.

React Native peut afficher des interfaces JavaScript pour les API de plate-forme. Ainsi, les applications React Native peuvent accéder à des fonctionnalités de la plate-forme mobile telles que la localisation de l'utilisateur et l'appareil photo du téléphone.

Bien que "**native**" fasse partie du nom, il ne s'agit pas d'un pur développement d'applications natives : Expo et React Native utilisent toujours JavaScript pour exécuter votre application. Au lieu de rendre ce JavaScript avec un moteur Web, ils utilisent les composants natifs réels de la plate-forme.

En utilisant React, vous pouvez réutiliser vos connaissances antérieures en matière de développement Web pour créer un mobile avec les fonctionnalités natives des applications mobiles traditionnelles. Il permet également aux développeurs de partager la majeure partie du code sur toutes les plates-formes, ce qui accélère le développement. Mais, pour passer de l'idée à l'application, une connaissance de base des plateformes natives est nécessaire.

## Quelle est la différence entre React et React Native ?

Comme tous les frères et sœurs, React et Réagir aux autochtones partagent des similitudes mais présentent aussi des différences distinctes.

- **React** est une bibliothèque JavaScript à code source ouvert, principalement utilisée pour créer des interfaces utilisateur pour les applications Web. Alors que **React native** a été explicitement conçu pour créer des interfaces utilisateur mobiles réactives.
- **React** ne dépend pas d'une plateforme, il peut donc être exécuté sur toutes sortes de plateformes. **React native** est plateforme-dépendante (jusqu'à un certain point). Alors qu'un grand pourcentage du code peut être partagé à un différent plateforme ou à travers un multiple plateforme set-up, il y a aussi un pourcentage de code spécifique qui est adapté à chaque plateforme c'est-à-dire iOS , Android et Web..

## Expo et React Native

[Expo](#) est une plate-forme pour créer des applications React universelles qui vous aident à développer, **créer, déployer et itérer** rapidement sur des applications mobiles. Il fournit une gamme d'outils pour faciliter encore plus le développement avec React Native.

Bien qu'Expo ne soit pas obligé de créer des applications React Native, il aide les développeurs en supprimant le besoin de connaissances natives de base.

- [Expo Go](#) est une application que vous pouvez télécharger sur votre téléphone pour « visualiser » votre application en cours de développement.
- [Expo CLI](#) est un outil pour créer, gérer et développer vos applications.
- [Expo SDK](#) est un ensemble modulaire de packages qui permettent d'accéder à des API natives, telles que `Camera` ou `Notifications`.
- [Expo Snack](#) est un éditeur basé sur le Web où vous pouvez écrire des extraits React Native et les exécuter dans le navigateur, ce que vous verrez dans le prochain exercice !

Vous verrez parfois Expo mentionnée dans la [documentation de React Native](#) . Nous vous recommandons généralement d'utiliser Expo lors de la création d'applications React Native. C'est le moyen le plus simple de commencer à créer votre application, et il n'y a pas de verrouillage. Une fois que vous démarrez avec Expo, vous pouvez toujours éjecter vers React Native uniquement.

Expo et React Native fournissent un cadre complet avec une grande communauté. Il existe des milliers **de packages prédéfinis** que vous pouvez utiliser directement dans votre application.

Pour la plupart des applications, Expo et React Native sont un bon choix :

- Les applications créées avec Expo ou React Native peuvent s'exécuter sur plusieurs plates-formes. Cela signifie un développement plus rapide et moins de code à maintenir tout en partageant la majeure partie du code.
- Il fournit un accès direct aux fonctionnalités natives, permettant aux développeurs de rendre l'application aussi performante que les applications natives pures.
- Démarrer avec Expo et React Native ne nécessite qu'un développement Web de base et une compréhension de base de la plate-forme native.

De grandes entreprises comme [Bloomberg](#) , [Shopify](#) et [Coinbase](#) utilisent React Native pour leurs applications mobiles. Coinbase a commencé avec native et est passé à React Native [avec beaucoup de succès](#) . Vous pouvez trouver plus d'entreprises répertoriées dans la [documentation de React Native](#) !



## **Component**

Toutes les applications d'Expo et de React Native sont constituées de *composants* . Ces *composants* sont de petits éléments réutilisables de votre application, qui fonctionnent tous ensemble. Chacun de ces composants a généralement une responsabilité unique. Cela peut varier du rendu du texte stylé au rendu d'autres composants pré-stylés pour créer un formulaire.

Tout comme React normal, l'application React Native s'ouvre entièrement d'un seul composant. Ce composant rend tous les autres composants de l'application, des écrans au texte simple. Expo et React Native utilisent le concept [entry point](#) pour ouvrir l'application.

## Cross-Platform Differences

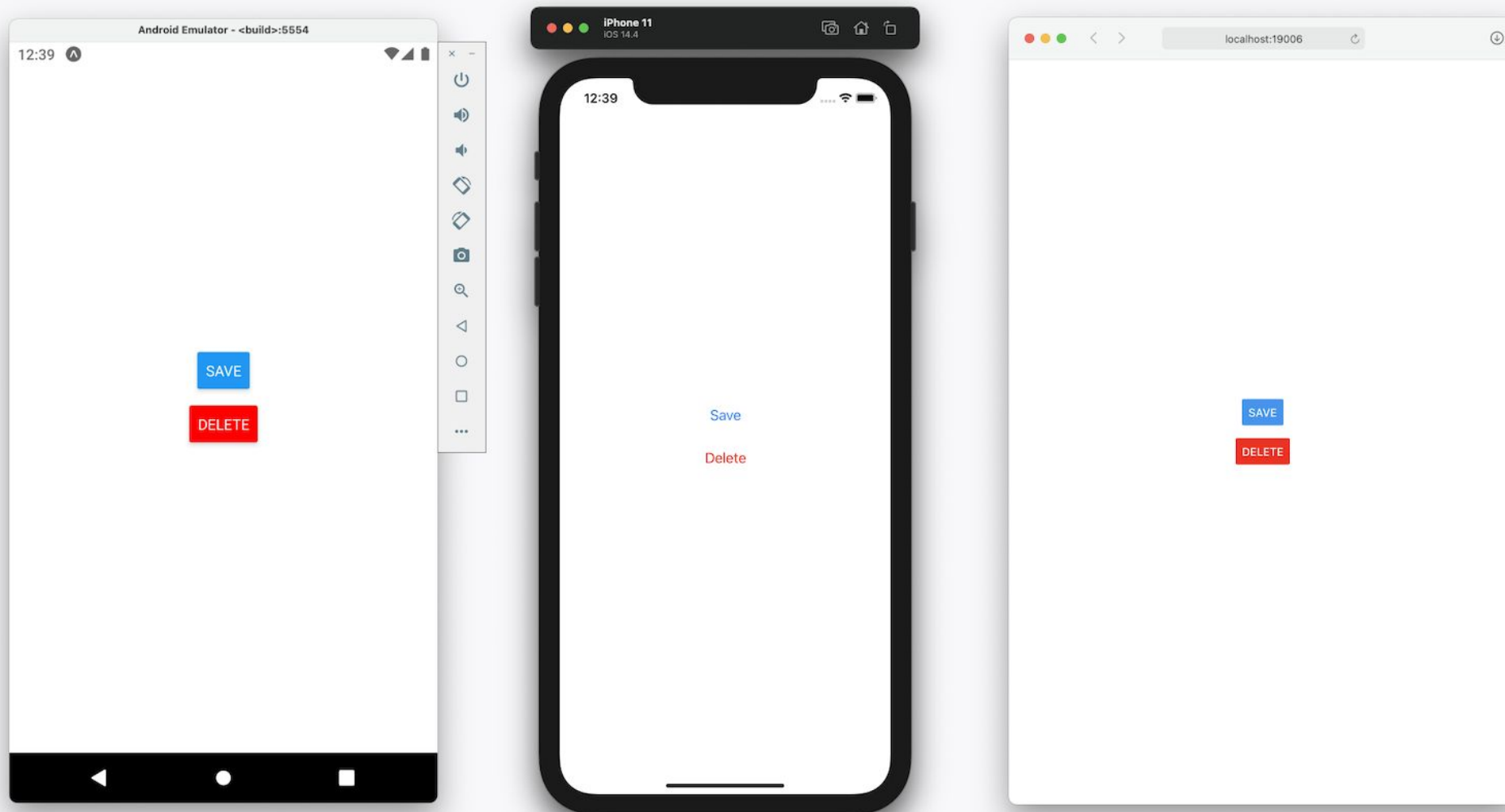
Une grande partie du code Expo et React Native peut être réutilisé sur plusieurs plates-formes natives, mais il existe certaines différences dont vous devez être conscient.

Les *composants natifs* peuvent sembler différents en raison des directives de conception fournies par la plate-forme. Sur iOS, Apple a mis en œuvre ses [directives d'interface humaine](#) tandis qu'Android fournit ses [directives de conception matérielle](#). Certains des *composants de base* peuvent sembler différents sur d'autres plates-formes à cause de cela.

Jetons un coup d'œil au `<Button>` composant. Dans cette image, vous pouvez voir comment cela est rendu sur Android et iOS.

Il existe également des différences dans les fonctionnalités fournies par chaque plate-forme. Apple souhaite que les développeurs d'applications utilisent leur fonctionnalité "Authentifier avec Apple", mais celle-ci n'est pas disponible sur Android. Pour cette raison, il n'est pas toujours possible de fournir une API unifiée pour les fonctionnalités natives.

[link](#)



- React Native est une bibliothèque qui utilise React pour le développement d'applications mobiles afin de créer des applications performantes avec JavaScript.
- Expo est une plate-forme pour les applications React universelles qui contient React Native et vous aide à itérer rapidement, sans aucune connaissance de la plate-forme native.
- Les composants de React Native ont un équivalent de *composant natif* qui est rendu sur la plate-forme native.
- Tout comme React, Expo et React Native utilisent le concept **entry point** de votre application pour la rendre sur différentes plates-formes natives.
- Étant donné que différentes plates-formes natives ne sont pas identiques, certains composants se comportent différemment sur certaines plates-formes.

# Configuration locale

L'objectif de cette partie est de:

- Installez le logiciel nécessaire pour faire fonctionner React Native
- Générer un projet de base avec les outils en ligne de commande d'Expo
- Prévisualisez-le sur votre ordinateur et, en utilisant Expo Go, prévisualisez-le également sur votre smartphone
- Modifiez l'application et voyez les changements en direct



<https://app.components.studio/edit/AkUn9pMhApjoXJk3DyfO/src/index.jsx?p=stories>

C'est un lien qui contient des exemples des composants de test

# Créer une application Expo

## Requirement

Pour créer l'application, vous devez d'abord installer quelques éléments sur votre ordinateur :

- [Node.js](#) - Un environnement d'exécution JavaScript
- [Visual Studio Code](#) - Un éditeur de texte

## Commandes utiles

- `node -v` - Vérifie votre version de Node.js
- `npm -v` - Vérifie votre version du gestionnaire de paquets Node
- `npm install -g expo-cli` - Installe les outils en ligne de commande d'Expo
  - Les outils de ligne de commande Expo facilitent la création, la maintenance et le test d'un projet Expo !
- `expo init hello-world` - Crée une nouvelle application Expo
- `cd hello-world` - Vous déplace dans le dossier **hello-world**

## Tester l'application avec Expo Go

### Installez l'application Expo Go

Vous testerez l'application sur votre appareil mobile personnel. Pour ce faire, vous devrez installer l'application appelée Expo Go.

L'application Expo Go se synchronisera avec les outils de ligne de commande Expo que nous avons utilisés pour créer notre application.

- [Expo Go pour Android](#)
- [Expo Go pour iOS](#)

Une fois l'application Expo Go installée, lançons le projet React Native. Depuis le terminal Visual Studio Code, tapez ce qui suit :

- `npm start` - Démarre l'application

Ouvrez maintenant l'application Expo Go que vous venez d'installer et scannez le code QR qui aurait dû apparaître dans le terminal.

Le chargement peut prendre une seconde ou deux, mais votre nouveau projet React Native devrait maintenant être exécuté sur votre appareil !

- Le développement d'applications Expo sur votre propre ordinateur nécessite [Node.js](#) et un éditeur de code.
- `npm install -g expo-cli` installe les outils de ligne de commande Expo.
- `expo init hello-world` crée un nouveau projet Expo nommé **hello-world**.
- Expo Go, l'application mobile pour [Android](#) et [iOS](#), facilite la prévisualisation de votre application sur votre smartphone.



# CORE COMPONENTS

## Qu'est-ce qu'un core component ?

Dans Expo et React Native, les composants sont traduits en composants natifs pour la plate-forme sur laquelle ils s'exécutent. Pour vous aider à démarrer, React Native fournit un ensemble de composants prêts à l'emploi pour votre application. Ces composants sont appelés "**core component**" et la plupart d'entre eux ont des implémentations Android et iOS intégrées.

Vous pouvez importer ces composants à partir du `react-native` package. Au cours de cette leçon, nous approfondirons `View`, `Text`, `Image`, `Button`, `TextInput`, et `ScrollView`. Ces composants sont essentiels pour toutes les applications Expo et React Native.

Nous couvrons certains des composants de base les plus courants dans cette leçon. Vous pouvez trouver la liste complète des composants de base dans la documentation de l' [API Expo](#) et la documentation [des composants React Native](#) .

| REACT NATIVE UI COMPONENT       | ANDROID VIEW                    | IOS VIEW                          | WEB ANALOG                                   | DESCRIPTION                                                                                           |
|---------------------------------|---------------------------------|-----------------------------------|----------------------------------------------|-------------------------------------------------------------------------------------------------------|
| <code>&lt;View&gt;</code>       | <code>&lt;ViewGroup&gt;</code>  | <code>&lt;UIView&gt;</code>       | A non-scrollling<br><code>&lt;div&gt;</code> | A container that supports layout with flexbox, style, some touch handling, and accessibility controls |
| <code>&lt;Text&gt;</code>       | <code>&lt;TextView&gt;</code>   | <code>&lt;UITextView&gt;</code>   | <code>&lt;p&gt;</code>                       | Displays, styles, and nests strings of text and even handles touch events                             |
| <code>&lt;Image&gt;</code>      | <code>&lt;ImageView&gt;</code>  | <code>&lt;UIImageView&gt;</code>  | <code>&lt;img&gt;</code>                     | Displays different types of images                                                                    |
| <code>&lt;ScrollView&gt;</code> | <code>&lt;ScrollView&gt;</code> | <code>&lt;UIScrollView&gt;</code> | <code>&lt;div&gt;</code>                     | A generic scrolling container that can contain multiple components and views                          |
| <code>&lt;TextInput&gt;</code>  | <code>&lt;EditText&gt;</code>   | <code>&lt;UITextField&gt;</code>  | <code>&lt;input type="text"&gt;</code>       | Allows the user to enter text                                                                         |

# workshop 1 : Environnement, pré requis & Hello world !

## Objectif

- Configurer l'environnement de développement
- Générer le squelette basique d'une application
- Tester son application sur son smartphone

## Prérequis

Il vous faudra de base :

- NodeJS

Installez ensuite la CLI de Expo via une console :

```
$ npm install -g expo-cli
```

npm, qui signifie « Node Package Manager », est le gestionnaire de paquets de Node.js. Il permet d'**installer des outils pour le développement**.

Le paramètre `-g` installe de manière **globale dépendance** dans tout votre ordinateur et sa session courante. Vous n'aurez donc pas à refaire cette action lors de prochains projets.

## Initialisation du squelette

On initialise une nouvelle application via la commande :

```
$ expo init "Nom_de_votre_app"
```

Pour notre cours, on va utiliser la version typescript.  
Merci de choisir template blank (Typescript)

Vous aurez l'architecture suivante :

- assets/ : dossier contenant vos images
- app.json : fichier de configuration du projet
- app.tsx : fichier d'entrée de votre application
- package.json : contient l'ensemble des dépendances et scripts du projet
- tsconfig.json : fichier de configuration de typescript

Lancement du serveur de dev configuration pour la gen du bundle ( une sorte de webpack )

Simple :

```
$ npm start
```

Un nouvel onglet dans votre navigateur va s'ouvrir, vous proposant divers options :

```
gitpod /workspace/expo-project (main) $ expo init "Nom_de_votre_app"
```

Migrate to using:

```
> npx create-expo-app --template
```

```
? Choose a template: > - Use arrow-keys. Return to submit.
```

```
----- Managed workflow -----
```

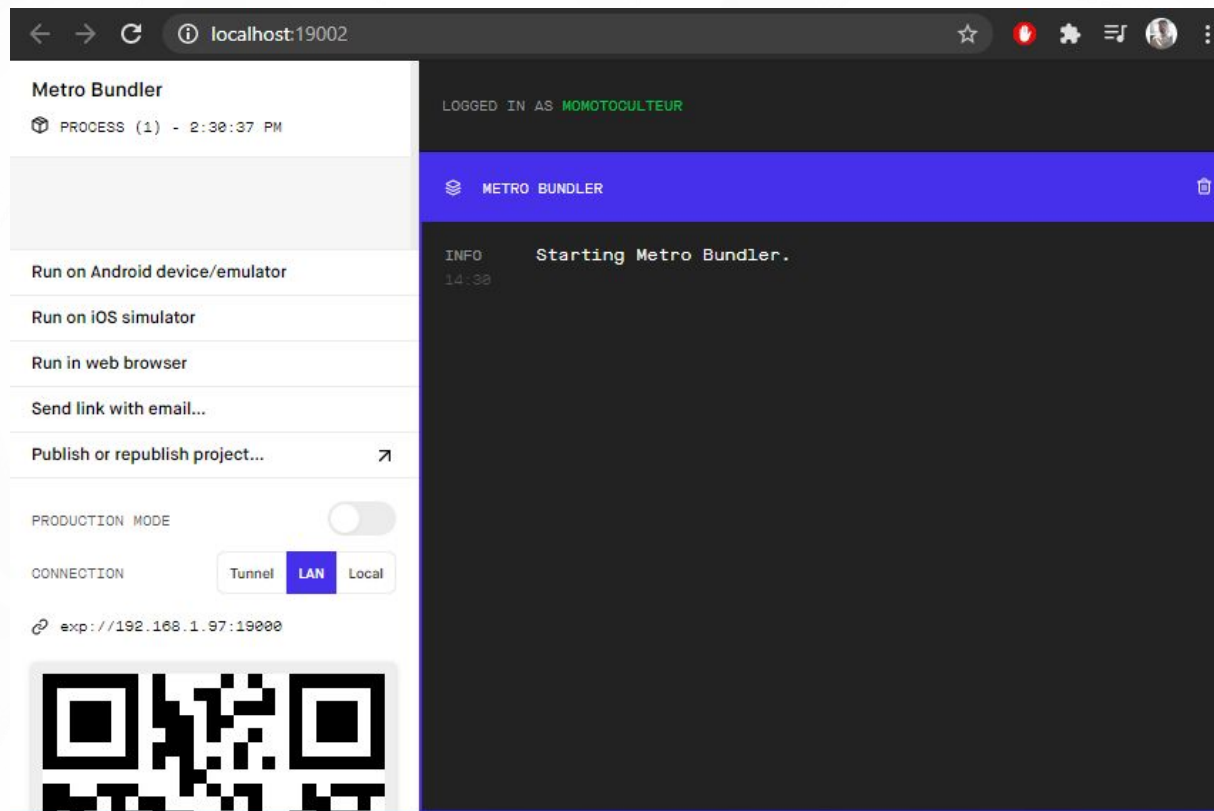
```
blank a minimal app as clean as an empty canvas
```

```
> blank (TypeScript) same as blank but with TypeScript configuration
```

```
tabs (TypeScript) several example screens and tabs using react-navigation and TypeScript
```

```
----- Bare workflow -----
```

```
minimal bare and minimal, just the essentials to get you started
```



Pour faire fonctionner le simulateur Android ou iOS, vous allez devoir installer les biblio natives. Mais on peut éviter cela en utilisant notre propre smartphone. On va devoir connecter notre smartphone au même réseau wifi que l'ordinateur sur lequel on développe.

Installez l'application EXPO, disponible sur l'iOS Store et GooglePlay store, selon votre type de device. Une fois l'application lancé sur votre smartphone, vous allez pouvoir scanner le QR code affiché sur la console, comme montré sur la capture d'écran précédent. Cela va permettre de transférer le bundle directement sur le smartphone.

Vous allez pouvoir ainsi voir vos modification en temps réel sur votre téléphone, de l'application que vous codez, à chaque sauvegarde.

## Icone & Splash screen

Vous pouvez changer l'icone de votre application via le fichier « **app.json** ».

De même pour le splash screen, qui correspond a l'image qui sera affiché le temps que votre application soit chargé par votre smartphone.

## Rechargement d'une application

Ouvrez le fichier App.js, vous devriez voir ceci :  
Vous retrouvez les textes visibles sur votre device. Ne vous attardez pas sur le code, on le verra en détail dans la partie 2. L'objectif actuellement c'est de changer le texte. Par exemple, remplacez :

```
<Text>Open up App.tsx to start working on your  
app!</Text>
```

Par

```
<Text>Hello Word!</Text>
```

Retournez dans votre application et constatez le changement de texte.

⇒ **Le rechargement de l'application, avec notre nouveau texte, est instantané. C'est ce que l'on appelle le [Live Reloading](#).**

```
Nom_de_votre_app > TS App.tsx > ...  
1  import { StatusBar } from 'expo-status-bar';  
2  import { StyleSheet, Text, View } from 'react-native';  
3  
4  export default function App() {  
5    return (  
6      <View style={styles.container}>  
7        <Text>Open up App.tsx to start working on your app!</Text>  
8        <StatusBar style="auto" />  
9      </View>  
10   );  
11 }  
12  
13 const styles = StyleSheet.create({  
14   container: {  
15     flex: 1,  
16     backgroundColor: '#fff',  
17     alignItems: 'center',  
18     justifyContent: 'center',  
19   },  
20 });  
21
```

Si vous ne constatez pas de modification, il faut forcer le rechargement de votre application. Si vous êtes sur smartphone, simulateur, émulateur, iOS, Android, le fonctionnement est différent :

- Sur smartphone / tablette pour iOS / Android : Vous devez **secouer** votre appareil. Jusqu'à l'apparition d'une pop-up (**menu debug**) et sélectionnez "**Reload**".
- Sur simulateur iOS, il faut faire ⌘ + R ou ⌘ + D pour afficher le **menu debug** et sélectionner "**Reload**".
- Sur émulateur Android, il faut faire R + R ou Ctrl (⌘ sur Mac) + M pour afficher le **menu debug** et sélectionner "**Reload**".

**Retenez bien la démarche.** Le rechargement est, par défaut, automatique, mais malheureusement, il est un peu hasardeux. Par exemple, lorsque vous aurez des erreurs dans votre code, il faudra très souvent forcer le rechargement de votre application pour que l'erreur disparaisse.



## Créer un dépôt Git

Un dépôt Git (repository) est tout simplement un projet versionné avec Git. Avant de créer un dépôt Git, il faut d'abord installer Git. Pour cela, rends toi sur <https://git-scm.com/downloads> et en fonction de ton OS, choisis l'installable qu'il te faut et installe le tout simplement. Une fois l'installation terminée, pour vérifier que tu l'installation est effective, ouvre un terminal et exécute la commande:

```
$ git --version
```

Maintenant que nous avons installé Git, on peut créer notre premier dépôt Git.

Pour faire de ce projet un dépôt Git, c'est à dire donc le versionner avec Git, il faut donc ouvrir un terminal et te rendre dans ton projet puis il faut exécuter la commande:

```
$ git init
```

Maintenant que nous avons un dépôt Git, nous avons un début. Il faut ensuite dire à Git quels fichiers il doit traquer (suivre), il ne va pas le deviner au fait. Pour suivre un fichier, il faut utiliser la commande:

```
$ git add fileName1.ext fileName2.ext
```

La commande git add prend donc un ensemble de fichiers en paramètre. Mais vous pouvez ajouter tous les fichiers du dossier courant en faisant:

```
$ git add .
```



Le problème avec cette commande pour un tout nouveau projet, c'est que tu risques de rajouter des fichiers qui ne doivent pas être suivis par Git, c'est le cas des fichiers qui contiennent les configurations avec des valeurs secrètes comme le mot de passe de ta base de données ou une clé d'API (fichier **.env** par exemple). Pour éviter cela, il faut donc rajouter un fichier qui s'appelle **.gitignore**.

Le fichier **.gitignore** est le fichier qui va nous permettre de dire à Git de ne pas suivre ces fichiers ou dossiers. Nous allons donc rajouter un fichier qui s'appelle **.gitignore** ( le **.** fait partie du nom du fichier) à la racine du projet et définir les fichiers et dossiers à ignorer.

Pour notre cas le contenu de fichier **.gitignore** est

```
Nom_de_votre_app > .gitignore
1  node_modules/
2  .expo/
3  dist/
4  npm-debug.*
5  *.jks
6  *.p8
7  *.p12
8  *.key
9  *.mobileprovision
10 *.orig.*
11 web-build/
12 |
13 # macOS
14 .DS_Store
15
```

## Faire un commit

Vous avez maintenant initialiser un dépôt Git, modifier les fichiers que vous souhaitez et vous les avez ajouter à Git avec **git add**. Il faut ensuite enregistrer les changements que vous avez effectuer et pour cela il faut faire un commit.

Pour faire un commit, il faut utiliser la commande **git commit**:

```
$ git commit -m 'Premier commit'
```

On utilise l'option **-m** pour spécifier le message du commit, avec Git, on ne peut pas faire un commit sans spécifier de message. Il faut donc dans ce message expliquer pourquoi vous avez fais des changements. Ici j'utilise juste le message " Premier commit" pour dire ce que ça veut vraiment dire quoi: C'est notre premier commit

Jusque là, ton code est toujours sur ton ordinateur, moi je n'y ai pas encore accès, il faut donc le publier sur Github.

## Créer un dépôt sur Github

Pour commencer connectez vous sur <https://github.com>, si vous avez pas de compte, crées en un, c'est gratuit.

**Pour ce cours, on va utiliser une organisation bien déterminer. Merci de m'envoyer votre identifiant de github, pour que je puisse vous envoyer une invitation.**

Après l'acceptation de l'invitation, Cliquez sur le bouton [New](#). Tu arrives sur une page qui ressemble à ceci:



Search or jump to...



[Pull requests](#) [Issues](#) [Marketplace](#) [Explore](#)



## Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

### Repository template

Start your repository with a template repository's contents.

No template ▾

Owner \*

Repository name \*

 iit-cross-platform ▾

/

Great repository names are short and memorable. Need inspiration? How about [silver-adventure](#)?

### Description (optional)

☐  **Public**

Anyone on the internet can see this repository. You choose who can commit.

☒  **Private**

You choose who can see and commit to this repository.

### Initialize this repository with:

Skip this step if you're importing an existing repository.

☐ **Add a README file**

This is where you can write a long description for your project. [Learn more](#).

### Add .gitignore

Choose which files not to track from a list of templates. [Learn more](#).

.gitignore template: None ▾

### Choose a license

A license tells others what they can and can't do with your code. [Learn more](#).

License: None ▾

 You are creating a private repository in the iit-cross-platform organization.

Create repository



© 2022 GitHub, Inc. [Terms](#) [Privacy](#) [Security](#) [Status](#) [Docs](#) [Contact GitHub](#) [Pricing](#) [API](#) [Training](#) [Blog](#) [About](#)

Dans le champ *Repository name*, saisi le nom de votre projet (sans espace et pas de caractères accentués),(chaque groupe doit respecter le nom suivant **GI\_3\_2022\_G\_[Numero de Groupe]\_[TeamName]**). En fait, la description n'est pas obligatoire, il faut ensuite choisir une portée pour votre projet:

- *Public* - tout le monde pourra voir votre projet (plus de 50 millions de personnes qui utilisent Github)
- *Privé* - vous contrôlez qui peut voir votre projet en envoyant une invitation à chaque personne

Pour notre cas, il faut créer un dépôt privé avec le nom **GI\_3\_2022\_G\_[Numero de Groupe]\_[TeamName]** , cliquez ensuite sur le bouton *Create repository* pour valider la création du dépôt.

Une fois terminer, vous aurez une interface comme ici qui vous montre les premiers pas pour publier votre projet sur ce dépôt:

[Pull requests](#) [Issues](#) [Marketplace](#) [Explore](#)[iit-cross-platform / GI\\_3\\_2022\\_G\\_1](#) Private[Sign in to Sourcegraph](#)[Watch](#) 1[Fork](#) 0[Star](#) 0[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Security](#) [Insights](#) [Settings](#)

### Quick setup — if you've done this kind of thing before

[Gitpod](#)[Set up in Desktop](#)

or

[HTTPS](#)[SSH](#)

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

### ...or create a new repository on the command line

```
echo "# GI_3_2022_G_1" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/iit-cross-platform/GI_3_2022_G_1.git
git push -u origin main
```



### ...or push an existing repository from the command line

```
git remote add origin https://github.com/iit-cross-platform/GI_3_2022_G_1.git
git branch -M main
git push -u origin main
```



### ...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#)

**ProTip!** Use the URL for this page when adding GitHub as a remote.



© 2022 GitHub, Inc.


[Terms](#)[Privacy](#)[Security](#)[Status](#)[Docs](#)[Contact GitHub](#)[Pricing](#)[API](#)[Training](#)[Blog](#)[About](#)

38

**TEKAB.DEV**

Pour commencer, il faut choisir le mode HTTPS, Ensuite, recherche la section ou il est écrit "...or push an existing repository from the command line", c'est la quatrième section en partant du haut, la deuxième en partant du bas, l'avant dernière section quoi.

#### ...or create a new repository on the command line



```
echo "# GI_3_2022_G_1" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/iit-cross-platform/GI_3_2022_G_1.git
git push -u origin main
```

Copie les lignes une par une puis vous les exécutez sur votre ordinateur, dans le dossier de votre dépôt Git:

```
$ git remote add origin https://github.com/iit-cross-platform/GI_3_2022_G_1.git
```

Cette ligne ajoute le dépôt qui se trouve sur le serveur [https://github.com/iit-cross-platform/GI\\_3\\_2022\\_G\\_1.git](https://github.com/iit-cross-platform/GI_3_2022_G_1.git) comme étant notre dépôt distant et nous l'appelons origin. Ce lien est différent dans votre cas, il faut donc copier le lien qui vous a été donné par Github.

Puis la deuxième ligne:

```
$ git branch -M main
```

Cette ligne modifie la branche par défaut de master à main, et enfin la troisième ligne:

```
$ git push -u origin main
```

Cette ligne va permettre d'envoyer notre code à la source origin ([https://github.com/iit-cross-platform/GI\\_3\\_2022\\_G\\_1.git](https://github.com/iit-cross-platform/GI_3_2022_G_1.git)) sur la branche main.

Dès que vous exécutez la commande, le terminal vous demande vos identifiants Github avant de valider le push, rentrez votre username (ou email), validez avec Entrer, puis saisissez votre mot de passe (il est caché, il ne va donc pas s'afficher) et validez avec Entrer, si tout est OK, vous avez un message comme sur l'image ci-dessous qui dit que votre code a été publié avec succès:



```
gitpod /workspace/expo-project (main) $ cd Nom_de_votre_app/
gitpod /workspace/expo-project/Nom_de_votre_app (main) $ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /workspace/expo-project/Nom_de_votre_app/.git/
gitpod /workspace/expo-project/Nom_de_votre_app (master) $ git add .
gitpod /workspace/expo-project/Nom_de_votre_app (master) $ git commit -m "first commit"
[master (root-commit) 7e98f15] first commit
12 files changed, 6778 insertions(+)
create mode 100644 .expo-shared/assets.json
create mode 100644 .gitignore
create mode 100644 App.tsx
create mode 100644 app.json
create mode 100644 assets/adaptive-icon.png
create mode 100644 assets/favicon.png
create mode 100644 assets/icon.png
create mode 100644 assets/splash.png
create mode 100644 babel.config.js
create mode 100644 package.json
create mode 100644 tsconfig.json
create mode 100644 yarn.lock
gitpod /workspace/expo-project/Nom_de_votre_app (master) $ git branch -M main
gitpod /workspace/expo-project/Nom_de_votre_app (main) $ git remote add origin https://github.com/iit-cross-platform/GI_3_2022_G_1.git
gitpod /workspace/expo-project/Nom_de_votre_app (main) $ git push -u origin main
Enumerating objects: 16, done.
Counting objects: 100% (16/16), done.
Delta compression using up to 16 threads
Compressing objects: 100% (15/15), done.
Writing objects: 100% (16/16), 192.96 KiB | 9.65 MiB/s, done.
Total 16 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/iit-cross-platform/GI_3_2022_G_1.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
gitpod /workspace/expo-project/Nom_de_votre_app (main) $
```

Retournez sur le navigateur, sur la page du dépôt puis actualisez la page et vous allez trouver votre projet en ligne.

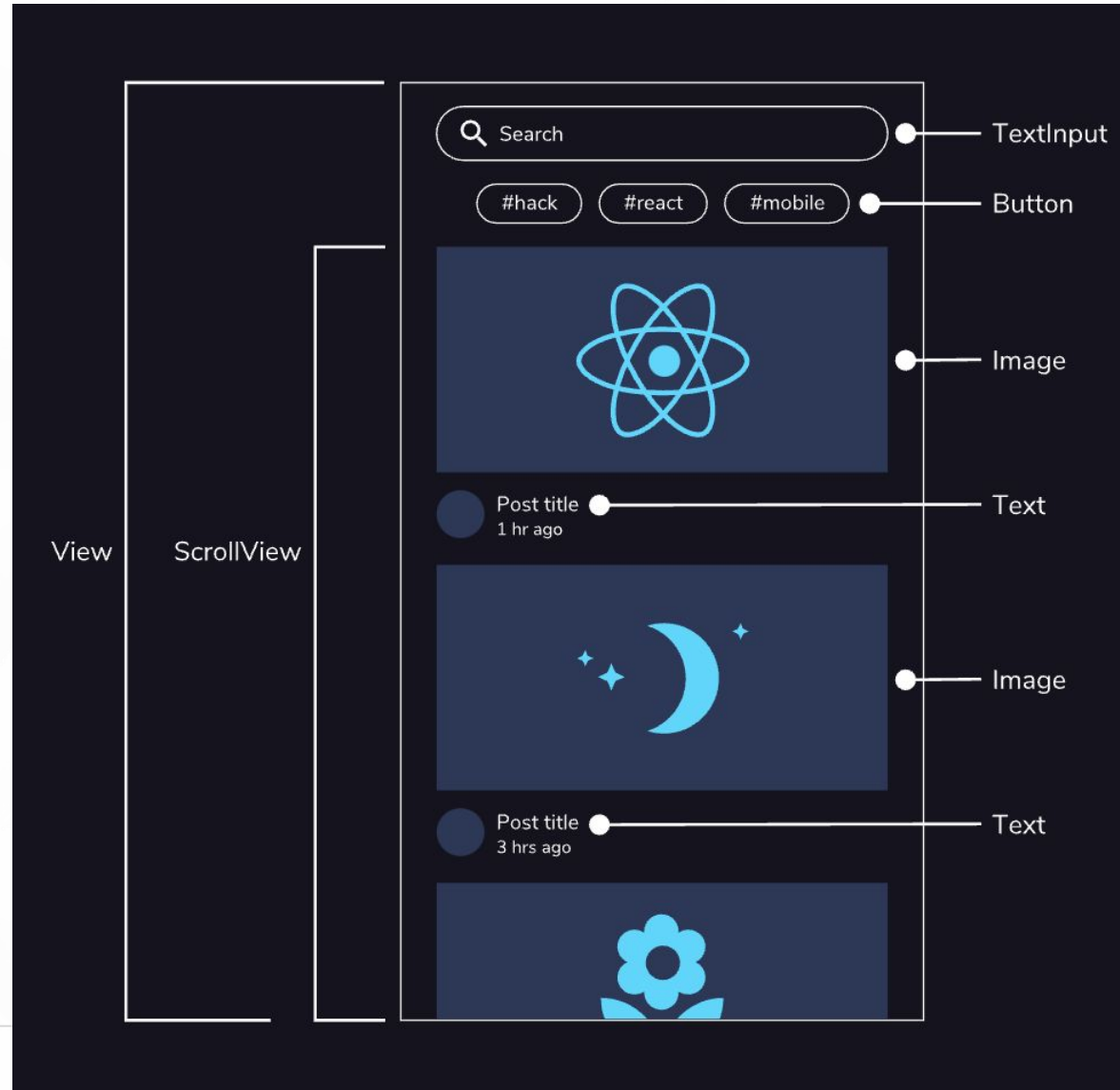
A partir de maintenant, vous pouvez copier le lien sur la barre d'URL, me l'envoyer par mail et moi je pourrais accéder à votre code, le cloner sur mon ordinateur pour vous aider à corriger vos problèmes.

# workshop 2 : Création composant React

## Objectif

- Tester les core component
- Créer ce composant

Les composants React Native, correspondent aux éléments graphiques simples que l'on retrouve sur les applications mobiles natives : Text, Button, Image, ScrollView, View, WebView, etc. **Ces composants existent déjà** et sont mis à disposition par React Native. La liste est disponible sur la [documentation des composants React Native](#). **Nous allons construire nos vues avec ces composants.**



## View component

L'un des composants de base les plus fondamentaux est le `View` [component](#). Avec `View`, vous pouvez créer des mises en page réactives à l'aide [de flexbox](#) ou ajouter un style de base aux composants imbriqués (**nested components**).

Le composant est mieux comparable à un `div` élément HTML. Tout comme `div`, le `View` **component** n'est visible que si on applique un style. Nous pouvons appliquer ce style à travers la `style` propriété.

```
<View style={{ width: 250, backgroundColor: 'yellow' }}>
  ...
</View>
```

Expo et React Native n'utilisent pas de CSS. Cette méthode de style n'est pas disponible sur les plates-formes natives. Au lieu d'utiliser CSS, nous pouvons écrire notre style en utilisant des objets JavaScript simples. Ces objets utilisent les mêmes propriétés CSS mais ils sont écrits en « **camelCase** ». Toutes les propriétés de style sont expliquées dans la [documentation de style](#) .

1. Commençons par création de dossier composants dans votre projet
2. Puis il faut créer dans le dossier **composants** un fichier **FirstComponents.tsx**
3. Il faut intégrer dans le fichier **FirstComponents.tsx** le code suivants:

```
export default function FirstComponents() {  
  return (  
    <></>  
  );  
}
```

< >  
angle brackets

1. Votre composant **FirstComponents.tsx** créé ne sera d'aucune utilité tant que vous ne l'importerez pas dans votre fichier **"App.tsx"** et n'enveloppez pas les composants créés avec **the angle brackets**.

```
Settings  TS App.tsx M X  README.md U  TS FirstComponents.tsx U  
de_votre_app > TS App.tsx > App  
import { StatusBar } from 'expo-status-bar';  
import { StyleSheet, Text, View } from 'react-native';  
import FirstComponents from './components/FirstComponents';  
export default function App() {  
  return (  
    <View style={styles.container}>  
      <FirstComponents></FirstComponents>  
      <Text>Hello Word!</Text>  
      <StatusBar style="auto" />  
    </View>  
  );  
}  
  
const styles = StyleSheet.create({  
  container: {  
    flex: 1,  
    backgroundColor: 'fff',  
    alignItems: 'center',  
    justifyContent: 'center',  
  },  
});
```

Ici, nous avons importé le composant "FirstComponents" et ajouté le composant spécifié en tant que <> dans la fonction "return()" de l'App() :



5. créer un `View` sans aucun style dans le fichier **FirstComponents.tsx** , puis il faut faire une capture d'écrans de l'application mobile et d'ajouter un l'image dans le repo git dans un dossier review.

Il faut faire un commit de message **"first step : exercice create view component"** qui contient la nouvelle modification de composant **FirstComponents** et l'image review puis il faut faire push.

6. N'oubliez pas que sans style, le `View` components n'est pas visible.

Modifions le code pour afficher notre `View` comme une boîte rouge. Pour cela, nous devons appliquer une largeur, une hauteur et une couleur de fond fixes. Définissez la largeur et la hauteur sur `100` et la couleur d'arrière-plan sur `red`.

**De même, il faut refaire commit et push pour cette modification et pour la capture d'écrans de résultat: le message de commit doit être "red view: exercice create view component".**

7. En remarquant que cette boîte rouge est coincée dans **le coin supérieur gauche** de notre application. Nous pouvons résoudre ce problème en utilisant **flexbox**.

Expo et React Native prennent en charge une forme de flexbox légèrement différente de celle du Web. Au lieu d'écrire `display: flex`, tous les composants sont une flexbox par défaut. Nous pouvons activer le comportement flexbox en utilisant la propriété `flex` dans notre objet de style. Toutes les propriétés flexbox prises en charge sont expliquées dans la [documentation flexbox](#) .

```
<View style={{ flex: 1 }}>
  ...
</View>
```

Ajoutons un composants `view` **invisible** qui positionne notre boîte rouge en plein milieu de notre écran, en utilisant les propriétés `flex`, `justifyContent` et `alignItems`.

Merci de refaire un autre commit et push qui contient bien évidemment une capture d'écrans et le nouveau code.

Le message doit être : **“flexbox style view: exercice create view component”**.

**8.** L'utilisation de flexbox est importante pour créer un style réactif dans React Native. La création d'applications pour Android et iOS implique également la création de plusieurs tailles d'écran. Que se passe-t-il lorsqu'un autre `View` est ajouté à la flexbox ?

Ajoutez une autre boîte avec des propriétés de style similaires à notre boîte rouge. Pour le rendre plus distinct, remplacez le `backgroundColor` par quelque chose d'autre, comme `blue`.

Et de même il faut partager avec moi votre code et le capture d'écrans en git : le message doit être **“multi view in flexbox: exercice create view component”**.



## **Text component**

Expo et React Native traduisent les composants en leur homologue natif, ils ne sont pas exactement similaires à React.

Avec le Web, vous pouvez afficher du texte n'importe où dans le document sans ajouter d'éléments parents. Sur les plates-formes natives comme Android et iOS, ce n'est pas possible.

Pour afficher du texte sur Android et iOS, la chaîne doit être enveloppée dans un `Text` [component](#). Avec ce composant, vous pouvez rendre et styliser le texte. Il peut également hériter du style d'un `Text` [component](#) parent, pour mettre en valeur certains mots.

## Example 1

```
import React from 'react';
import { View, Text } from 'react-native';

const App = () => (
  <View style={{ flex: 1, justifyContent: 'center' }}>
    <Text>
      The quick brown fox jumps over the lazy dog
    </Text>
  </View>
);

export default App;
```

## Exemple 2

```
import React from 'react';
import { View, Text } from 'react-native';

const App = () => (
  <View style={{ flex: 1, justifyContent: 'center' }}>
    <Text style={{fontSize : 16}}>
      The <Text style={{ fontFamily: 'Cochin',fontWeight: 'bold'}}>quick brown </Text>fox jumps over the lazy dog
    </Text>
  </View>
);

export default App;
```

1. De même ici , il faut créer dans le dossier components un composant **SecondComponents.tsx** et il faut l'intégrer dans le fichier **App.tsx**
1. Avec `Text`, vous pouvez restituer n'importe quelle chaîne de votre choix. Bien que le texte soit visible, il n'est pas parfait en termes de lisibilité. Améliorons cela en créant un text de taille de police `16` pixels.
1. Parfois, vous voulez aider le lecteur en mettant l'accent sur des parties spécifiques du texte. Rendons une partie de text en gras en l'enveloppant dans un `Text` component imbriqué.
1. Il y a beaucoup plus de règles de style pour `Text` attendre d'être appliquées. Vous pouvez tous les voir dans la documentation [des accessoires de style de texte](#) .

Merci de faire commit lorsque vous terminez l'exercice.

## Image component

Presque toutes les applications affichent des images. Ce contenu peut être rendu dans React Native à l'aide du `Image` [component](#). Il est similaire à l'`<img>` [élément HTML](#), mais le `Image` [component](#) a plus de fonctionnalités.

L'une des fonctionnalités supplémentaires de `<Image>` est la possibilité de charger des images à partir de différentes sources. `https://` Il peut s'agir d'un lien accessible au public, d'une `file://` référence locale, d'une chaîne encodée en Base64 ou d'une image importée en tant que module avec `require`. Chacune des sources d'images a ses propres avantages.

Dans cet exercice, nous utiliserons les deux méthodes les plus fréquemment utilisées - en utilisant des images à partir d'URL HTTP et des images importées localement.

```
import React from 'react';
import { Image, View } from 'react-native';

const App = () => (
  <View style={{ flex: 1, justifyContent: 'center' }}>
    <Image style={{ width: 100, height: 100 }} source={{ uri: 'https://picsum.photos/100/100' }} />
    { /* Render the `Image` here */ }
  </View>
);

export default App;
```

```
import React from 'react';
import { Image, View } from 'react-native';

const App = () => (
  <View style={{ flex: 1, justifyContent: 'center' }}>
    <Image style={{ width: 100, height: 100 }} source={require('./react-native.jpg')} />
    {/* Render the `Image` here */}
  </View>
);
export default App;
```

1. De même ici , il faut créer dans le dossier components un composant **ThirdComponents.tsx** et il faut l'intégrer dans le fichier **App.tsx**
1. Commençons par rendre un `Image` composant avec une largeur et une hauteur fixes de 100.
1. Merci de tester les différents de méthode d'importation des images.

Merci de faire commit lorsque vous terminez l'exercice.



## **ScrollView component**

La création de contenu déroulant sur le Web est assez simple. Il vous suffit d'appliquer une largeur et une hauteur fixes à un conteneur pour rendre le contenu hors limites défilable. Avec le développement natif, cela nécessite un peu plus qu'un `View` component aux dimensions fixes.

`View` components ne peuvent pas défiler dans Expo et React Native. Le rendu du contenu déroulant nécessite des calculs supplémentaires qui pourraient nuire aux performances lorsqu'ils sont appliqués à tous les `View` components.

Expo et React Native ont différents composants déroulants que nous pouvons utiliser, comme `ScrollView`. `ScrollView` nous permet de gérer et de personnaliser entièrement la façon dont le contenu doit être défilé.

Merci d'ajouter un nouveau composant et de tester le code de la page suivants puis il faut faire commit comme d'habitude :)

```
import React from 'react';  
import { Text, View , ScrollView } from 'react-native';
```

```
<View style={{ flex: 1, justifyContent: 'center' }}>  
  <Text style={{ fontSize: 24, textAlign: 'center' }}>  
    Scroll me!  
  </Text>  
  <View style={{ height: 400, backgroundColor: '#e5e5e5' }}>  
    <ScrollView>  
      { /* This is our scrollable area */}  
      <View style={{ width: 300, height: 300, backgroundColor: 'red' }} />  
      <View style={{ width: 300, height: 300, backgroundColor: 'green' }} />  
      <View style={{ width: 300, height: 300, backgroundColor: 'blue' }} />  
    </ScrollView>  
  </View>  
</View>
```

## Button component

Outre tous les composants visuels, chaque application nécessite des composants interactifs. L'une des interactions les plus fréquentes est une interaction « clic » ou « appui ». La plupart des composants de base prennent en charge **press interactions**. Mais, il existe un composant spécialement créé pour gérer les presses simples : le `Button` [component](#).

NB: la personnalisation du style de `Button` est très limitée.

Pour capturer l'utilisateur qui clique sur le bouton, nous devons utiliser le `onPress` gestionnaire d'événements(**event handler**).

Ceci est appelé `onPress` parce qu'il n'y a généralement pas de souris disponible sur un appareil mobile.

```
<Button
  title="Profile page"
  onPress={() => navigate('profile')}
/>
```

Les interactions simples avec la presse peuvent aller de la soumission d'un formulaire à la navigation vers une autre page.

```

import React, { useState } from 'react';
import { Button, Text, View } from 'react-native';

const App = () => {
  const [pressedCount, setPressedCount] = useState(0);

  return (
    <View style={{ flex: 1, justifyContent: 'center' }}>
      <Text style={{ margin: 16 }}>
        {pressedCount > 0
          ? `The button was pressed ${pressedCount} times!`
          : 'The button isn\'t pressed yet'
        }
      </Text>
      <Button
        title='Press me'
        onPress={() => {setPressedCount((prevState) => prevState + 1)}}
      />
    </View>
  );
};

export default App;

```

## TextInput component

La plupart des applications exigent que les utilisateurs fournissent du contenu, comme l'écriture des tweets sur Twitter. Le `TextInput` [component](#) est créé pour capturer les entrées textuelles et numériques. Ce composant est le mieux comparable à l' `<input>` [élément HTML](#) .

Pour écouter les modifications d'entrée de l'utilisateur, nous pouvons utiliser le `onChangeText` gestionnaire d'événements(**event handler**). Ce gestionnaire d'événements reçoit l'entrée sous forme de chaîne, fournie par l'utilisateur.

```
import React, { useState } from 'react';
import { View, Text, TextInput } from 'react-native';

const App = () => {
  const [name, setName] = useState("");
  return (
    <View style={{
      flex: 1,
      alignContent: 'center',
      justifyContent: 'center',
      padding: 16,
    }}>
      <Text style={{ marginVertical: 16 }}>
        {name ? `Hi ${name}!` : 'What is your name?'}
      </Text>
      <TextInput
        style={{ padding: 8, backgroundColor: '#f5f5f5' }}
        onChangeText={text => null}
      />
    </View>
  );
};
```

## Combining components

Lors de la création d'une application, vous devez souvent réutiliser certains éléments de l'interface utilisateur.

L'ajout direct d'un style aux composants sans grouper ces éléments en composants peut entraîner des incohérences. Il est recommandé de créer des composants personnalisés qui appliquent ce style dans votre application.

La création de composants personnalisés est similaire à React. Nous pouvons créer une fonction et la restituer en tant que composant. **Lors de la création d'un composant personnalisé, il est important de penser aux fonctionnalités qu'il doit prendre en charge.** Les composants avec une seule ou quelques fonctionnalités sont plus faciles à réutiliser à plusieurs endroits.

Diviser votre conception en composants réutilisables plus petits est subjectif. Il n'y a pas de solution "taille unique". Pour en savoir plus sur le processus de décomposition de la conception, lisez le guide [Thinking in React](#).

```
import React from 'react';
import { View, Text } from 'react-native';

const App = () => (
  <View style={{ flex: 1, justifyContent: 'center' }}>
    <View style={{ width: 100, height: 100, backgroundColor: 'red' }} />
    <View style={{ width: 100, height: 100, backgroundColor: 'green' }} />
    <View style={{ width: 100, height: 100, backgroundColor: 'blue' }} />
  </View>
);

export default App;

export const Box = (props) => (
  <View style={{ width: 100, height: 100, backgroundColor: props.color }} />
  // Move a box `View` component here
);
```



- Le `View` component peut être utilisé pour créer des mises en page réactives à l'aide de flexbox ou ajouter un style de base aux composants imbriqués.
- Le `Text` component affiche du texte. Sur Android et iOS, toutes les chaînes de texte doivent être enveloppées dans l'une d'entre elles.
- L' `Image` component affiche des images. Ils peuvent être référencés à l'aide d'un `https://lien`, d'une `file://référence locale`, d'une chaîne encodée en Base64 ou d'une image importée en tant que module avec `require`.
- Le `ScrollView` component est un conteneur "visible" avec la fonctionnalité de défilement ajoutée. Dans la plupart des endroits, vous n'auriez pas besoin de faire défiler.
- Le `Button` component est utilisé pour les interactions « clic » ou « appui ».
- Le `TextInput` component est utilisé pour capturer l'entrée de l'utilisateur.
- Les composants peuvent être combinés en composants personnalisés pour un style et une réutilisation cohérents.

# workshop 3 : init storybook

Pour notre cas, L'objectif de storybook est de préparer un portail Web pour présenter, documenter, tester et améliorer tous vos composants React Native. C'est un endroit où vous pouvez créer votre propre bibliothèque au fil du temps, puis utiliser cette bibliothèque pour créer et publier rapidement toutes les applications que vous avez.

Storybook est un outil open source permettant de créer des composants et des pages d'interface utilisateur de manière isolée. En d'autres termes, il s'agit d'une bibliothèque que vous pouvez ajouter à votre projet pour tester et documenter vos composants.

C'est un environnement de développement permettant de naviguer dans **un catalogue de composant** (book), **de voir les différents états d'un composant par use-case** (story) tout en apportant des outils de test, de prévisualisation et de documentation du composant.

Storybook fonctionne avec la plupart des frameworks UI ( Vue , React , Angular , React Native ou même les Web Components )

## initialisation du projet:

```
$ npx create-expo-app --template expo-template-storybook my-app-story-book
```

```
$ cd my-app-story-book
```

```
$ yarn
```

```
$ yarn storybook
```

## Création 1ère composant en storybook

<https://gist.github.com/amina21/82f3069b605c0db40dd7c1093be60790>

## Quel sont Storybook controls?

Les storybook controls constituent un mécanisme pour afficher les contrôles graphiques de l'interface utilisateur (par exemple, un sélecteur de couleurs) sur un panneau Storybook dédié pour interagir dynamiquement avec les composants. Ainsi, par exemple, l'utilisateur peut facilement modifier la couleur d'arrière-plan ou la taille de police d'un composant pour faire des test manuel ou de démonstration.

<https://storybook.js.org/docs/react/essentials/controls>

# workshop 4 : yarn workspace

Workspace Yarn nous permettent d'organiser la base de code de notre projet à l'aide d'un référentiel monolithique (monorepo).

Notre Objectif est de créer un projet monorepo, prenant en charge **React Native (via Expo)** et le développement de composants React-Native via Storybook.

Pour configurer les espaces de travail Yarn, créez un **package.json** avec le contenu suivant :

```
// /package.json
{
  "private": true,
  "workspaces": [
    "packages/*"
  ]
}
```

Le champs **workspaces** de cet exemple est configuré de sorte que chaque répertoire à l'intérieur packages agisse comme un package séparé, avec son propre ensemble de dépendances de **package.json** (vous pouvez le remplacer par un autre répertoire parent, ou même une liste spécifique de répertoires).

Créons le dossier **packages** :

```
$ mkdir packages
```

## Création du projet Expo

Il faut vérifier bien que vous avez installé déjà expo (comme on a fait dans le workshop 1)

On va commencer par la création l'application principale, en utilisant Expo. Nous l'appellerons app pour cet exemple :

```
# /  
expo init packages/app
```

Il faut sélectionner le modèle **TypeScript** pour ce projet

Ensuite, on doit assurer que le package a un nom et une version.

Ici, on va créer un **scoped package** **@my[TeamName]** (par exemple @my-iit-g2) , ce qui me permet d'importer tous les packages du monorepo à l'aide d'un identifiant

Modifiez votre **packages/app/package.json** par l'ajouter ces deux champs :

```
// /packages/app/package.json
"name": "@my[TeamName]/app",
"version": "1.0.0",
```

## Configurer Expo pour bien fonctionner avec Yarn Workspaces

Installez **expo-yarn-workspaces** sur votre **app/**.

```
# /packages/app
yarn add -D expo-yarn-workspaces
```

Ajoutez le script suivant (sous la propriété **scripts**) dans **app/package.json** :

```
// /packages/app/package.json
...
  "postinstall": "expo-yarn-workspaces postinstall"
...
```

Créez un **metro.config.js** fichier contenant la configuration suivante :

```
// /packages/app/metro.config.js
const { createMetroConfiguration } = require('expo-yarn-workspaces')

module.exports = createMetroConfiguration(__dirname)
```



## C'est quoi Metro ?

Metro est un **bundler JavaScript** qui accepte des options, un fichier d'entrée et il va nous donner un fichier JavaScript contient tous les fichiers JavaScript. Chaque fois qu'on exécute un projet react-native, une compilation de nombreux fichiers javascript est effectuée dans un seul fichier. Cette compilation est effectuée par un bundler qui s'appelle **Metro** .  
Metro se lance avec le serveur node.

changez le point d'entrée **main**, en celui qui sera généré par **expo-yarn-workspaces** (vous pouvez choisir n'importe quel nom de fichier/emplacement. Ici, j'ai choisi de le générer dans le dossier **.expo** car il sera ignoré par Git) :

```
// /packages/app/package.json
...
  "main": ".expo/__generated__/AppEntry.js",
  ...
```

Exécutez par la suite **yarn postinstall** et **yarn start**

```
# /packages/app
yarn postinstall
yarn start
```

Modifiez votre **packages/ui/package.json** par l'ajouter ces trois champs :

```
// /packages/ui/package.json
"name": "@my[TeamName]/ui",
"version": "1.0.0",
"main": "components/index.tsx",
```

Créez un fichier **index.tsx** dans le dossier **packages/ui/components/** qui contient la liste des composants:

```
// /packages/ui/components/index.tsx
...
export { MyButton } from './Button/Button'
export { IITCard } from './Card/Card'
...
```

Exécutez par la suite **yarn** et **yarn start** dans le dossier **/packages/ui/**

Ajoutez le package **@my[TeamName]/ui** aux liste de dépendances de l'application :

```
# /packages/app  
yarn add @my[TeamName]/ui@1.0.0
```

Puis importez le package ui dans **App.tsx**:

```
# /packages/app/App.tsx  
import { StatusBar } from 'expo-status-bar';  
import { StyleSheet, Text, View } from 'react-native';  
import { MyButton } from '@myiit-g2/ui'  
export default function App() {  
  return (  
    <View>  
      <Text>Open up App.tsx to start working on your app! </Text>  
      <MyButton  
        onPress={() => { }}  
        text="Hello world"  
      />  
      <StatusBar style="auto" />  
    </View>  
  );  
}
```

Pour que notre application `@my[TeamName]/app` puisse importer les composants TypeScript depuis `@my[TeamName]/ui`, il doit être transpilé.

Donc :

Soit on ajoute une étape de construction à `@my/ui`, afin que toute la bibliothèque puisse être transpilée dans un format pouvant être importé. Comme (<https://lerna.js.org/>)

Soit on utilise le package `@expo/webpack-config`, qui nous permet d'importer simplement le composant directement dans le même processus de transpilation (compilation) que l'application utilise déjà.

```
// /packages/app/webpack.config.js
const createExpoWebpackConfigAsync = require('@expo/webpack-config')
module.exports = async function (env, argv) {
  const config = await createExpoWebpackConfigAsync(
    {
      ...env,
      babel: {
        dangerouslyAddModulePathsToTranspile: [
          // Ensure that all packages starting with @my[TeamName] are transpiled.
          '@my[TeamName]',
        ],
      },
    },
    argv,
  )
  return config
}
```