

# INTRODUCTION À REACT



## SOMMAIRE 1/3

### 001

#### L'ENVIRONNEMENT

Langage, stack, ...

### 002

#### TYPESCRIPT : LES BASES

Types, opérateur spread, classes

### 003

#### REACT : LES BASES

Philosophie, éléments,  
composants

### 004

#### REACT : TEMPLATES

Evénements, affichage  
conditionnel



## SOMMAIRE 2/3

### 005

#### REACT : HOOKS

useState, useEffect, ...

### 006

#### REACT : NAVIGATION

Input/Output, routing, guards  
TP

### 007

#### REACT: THEMES

Thèmes, Theme Switcher

### 008

#### CI/CD : Générer un APK auto

GitHub+Expo Go, WebHooks



## SOMMAIRE 2/3

# 009

TYPESCRIPT : ASYNCHRONE

Asynchronisme



# ENVIRONNEMENT

001



Télécharger et installer les pré-requis.

Ce qui ont déjà installé : vérifier les versions → Mauvaise version ou plusieurs versions  
= on va perdre beaucoup de temps

IDE : Chacun est libre d'utiliser l'IDE de son choix, je propose de l'aide sur VS Code

Git :

URL : <https://git-scm.com/downloads>

Credentials : <https://support.atlassian.com/bitbucket-cloud/docs/configure-your-dvcs-username-for-commits> / <https://git-scm.com/book/fr/v2/Utilitaires-Git-Stockage-des-identifiants>

Connexion à github depuis votre machine :

<https://docs.github.com/fr/authentication/connecting-to-github-with-ssh>

MacOs : pb de droits EACCESS <https://docs.npmjs.com/resolving-eacces-permissions-errors-when-installing-packages-globally>

NodeJS LTS :

URL : <https://nodejs.org/en/download/>

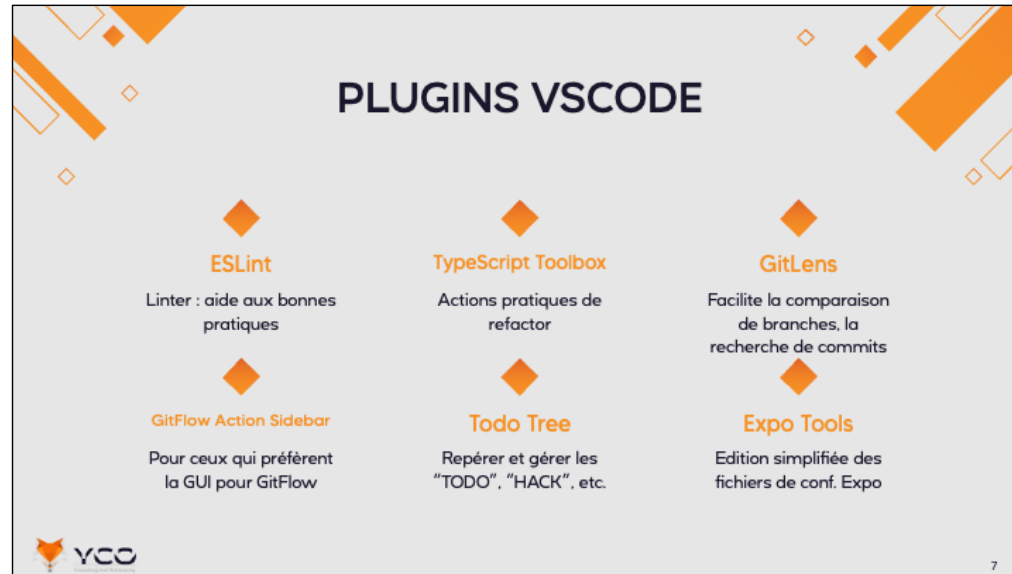
Attention, Angular ne fonctionne que sur la LTS

Expo :

<https://docs.expo.dev/get-started/installation/>

npm install --global expo-cli

npx expo -h pour avoir la liste des commandes dispo



1<sup>ère</sup> ligne : plugins « obligatoires » pour travailler de manière fluide

2<sup>ème</sup> ligne : selon les goûts, peuvent être utiles

⚠ Attention dans VS Code de ne pas activer tous les plugins en même temps si vous l'utilisez déjà pour d'autres langages.



- Niveau 0 de git : On utilise juste la branche master (ou main maintenant) pour garder ses modifications
- Déjà mis en place régulièrement : une branche Dev destinée à partager les avancées de l'équipe de développement sans perturber la stabilité de la branche master. Elle peut être déployée sur un serveur d'intégration. Elle peut être merge sur le master avec une/des pull-request(s) à chaque livraison (ne pas oublier les tags)
- Spécifique à GitFlow :
  - Pour chaque feature → Création d'une branche (si possible avec le numéro de ticket) avec le préfixe feature/, à partir de la branche dev (on merge régulièrement la branche dev sur sa branche de feature)
  - Quand la feature est terminée, la branche est cloturée et mergée sur la branche dev
  - Pour un hotfix (corriger exceptionnellement une urgence de production) → Création d'une branche avec le préfixe hotfix/, **à partir du master**, qui doit être cloturée très rapidement, mergée sur le master, puis merger le master sur le dev, puis le dev sur les branches features.



002

## **TYPESCRIPT : LES BASES**



# TYPESCRIPT

## C'EST QUOI ?

- Langage orienté objet (typé)
- Transpilé en JS (via tsc)
- Implémente la norme ES6
- Supporté par Microsoft
- Open source
- Utilisable backend/frontend



# TYPES

Déclaration d'une fonction

```
echoName(name: string) {  
  let punctuation: string = '!?.';  
  let nbOccurrences: number = 0; // int, decimal, float...  
  let isEnabled: boolean = false;  
  const numArray: Array<number> = new Array<number>();  
  // ou const numArray: number[] = [];  
  numArray.push(2.3); // numArray[0]  
  numArray.push(42); // numArray[1]  
  let tuple: [number, string] = [1, 'J\'aime le TypeScript'];  
  // tuple[0]: number = 1;  
  // tuple[1]: string = 'J\'aime le TypeScript'  
  
  let test: {x: number, y: number} = {x: 0, y: 1};  
  // Type anonyme  
  
  let badVariable: any;  
  
  console.log('Bonjour ' + name + punctuation);  
}
```

YCO

Il est possible de ne pas écrire le type dans certains cas (types de base du langage), c'est fortement déconseillé dans tous les autres cas.  
On préférera toujours typer les variables, ne jamais utiliser any quand ce n'est pas indispensable (très rare, et seulement pour s'adapter à du code externe qui serait mal fait).  
On peut configurer le linter et le compilateur pour nous obliger à typer les variables.  
(Avec la règle noImplicitAny, qui force à écrire any si l'on ne précise rien, pour prendre la responsabilité du code horrible que l'on laisse à ses collègues),

Il peut être intéressant de s'habituer à la syntaxe Array<type> afin de se préparer aux Types Génériques

## LET & CONST

Déclaration  
d'une  
fonction

```
echoName(name: string) {  
  const punctuation: string = '!?.';  
  punctuation = '!'; // KO 🚫  
  let nbOccurrences: number = 0; // int, decimal, float...  
  nbOccurrences++; // OK 🟢  
  
  const numArray: Array<number> = new Array<number>();  
  numArray.push(2.3); // OK 🟢  
  numArray.push(42); // OK 🟢  
  numArray = [6, 8, 10]; // KO 🚫  
  
  const test: {x: number, y: number} = {x: 0, y: 1};  
  test.x = 12; // OK 🟢  
  test = {x: 14, y: 16}; // KO 🚫  
  
  console.log('Bonjour ' + name + punctuation);  
}
```

## OPÉRATEUR "SPREAD" (...)

Déclaration  
d'une  
fonction

```
uselessFunc() {  
  // Déstructuration  
  let [a, b, c, ...restant] = [1, 14, 3, 18, 19];  
  // a = 1, b = 14, c = 3, restant = [18, 19]  
  
  // Tableaux  
  let liste1: Array<Number> = [1, 2];  
  liste1 = [...liste1, 3];  
  
  // Extension  
  let point2D: { x: number, y: number } = { x: 2, y: 3 };  
  let point3D: { x: number, y: number, z: number } = { z: 5, ...point2D };  
  // point3D = { x: 2, y: 3, z: 5 }  
  // ⚠ Ce n'est pas une Deep-Copy  
  point2D.x = 10;  
  // point3D = { x: 10, y: 3, z: 5 }  
  
  // Fusion  
  let foo = { w: 0 };  
  let bar = { t: 1, f: 2 };  
  let fooBar = { ...foo, ...bar };  
  // fooBar = { w: 0, t: 1, f: 2 }  
  let autrePoint2D = { x: 20, y: 30 };  
  point3D = { ...point3D, ...autrePoint2D };  
  // point3D = { x: 10, y: 20, z: 5 }  
}
```



Peut être utilisé pour passer des valeurs en paramètre d'une fonction depuis un tableau de valeurs mais c'est à éviter. On peut en trouver dans du code legacy.

Ex. : let params = [1, « toto », false];

this.maFonction(...params);

Pour une deep-copy :

let object2 = JSON.parse(JSON.stringify(object1));

# CLASSES & INTERFACES

Déclaration  
d'une  
classe

```
export enum EnumMarqueVoiture {  
  CITROEN = 1,  
  PEUGEOT = 2,  
  RENAULT = 3  
}  
  
export interface IVoiture {  
  marque: EnumMarqueVoiture;  
  modele: string;  
  kilometrage: number | null;  
}  
  
export class Voiture implements IVoiture {  
  marque: EnumMarqueVoiture;  
  modele: string;  
  kilometrage: number;  
  
  constructor(marque: EnumMarqueVoiture, modele: string, kilometrage: number) {  
    this.marque = marque;  
    this.modele = modele;  
    this.kilometrage = kilometrage;  
  }  
  
  klaxonner() {  
    console.warn('BOUGE DE LÀ !');  
  }  
}
```

# CLASSES & INTERFACES

app.component.ts

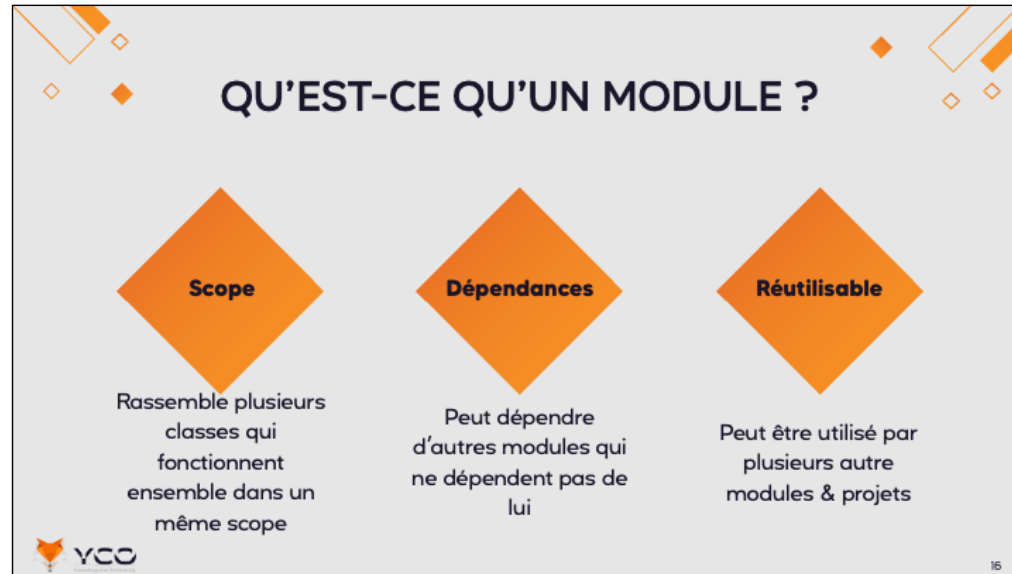
Déclaration d'une classe

```
import { EnumMarqueVoiture, Voiture } from './models/voiture.model';  
  
export class AppComponent {  
  modele: string = '307';  
  voiture: Voiture | undefined;  
  
  uselessFunc() {  
    this.voiture = new Voiture(EnumMarqueVoiture.PEUGEOT, this.modele, 20000);  
    this.voiture.klaxonner();  
  }  
}
```

YCO

15

Pourquoi export ? → Modules



Ils remplacent les namespaces (obsolètes), mais dépendent d'un injecteur de modules (CommonJS, Require.js, ...) ou d'un runtime qui gère les modules (Node.js)

Un module correspond à un fichier → On peut les regrouper en important puis ré-exportant depuis un même fichier.

Intérêts :

- Peut être transpilé dans un fichier séparé
- Peut être chargé seulement au moment où il est utilisé
- Peut être mis à jour sans mettre à jour tout le projet

Ne pas confondre avec les modules Angular ou npm



# MODULES

validator.module.ts

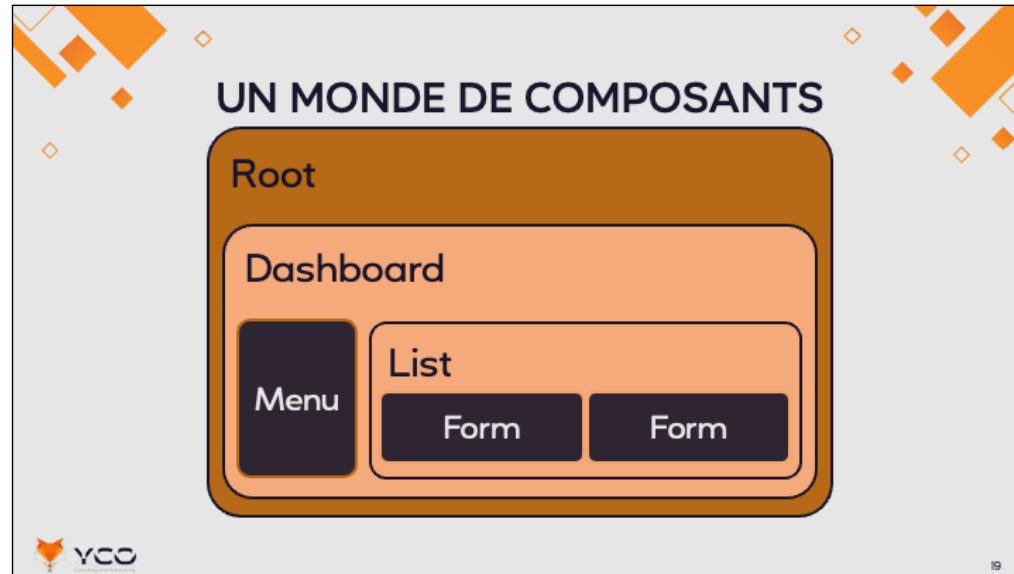
Déclaration  
d'un  
module

```
export * from './StringValidator'; // exports 'StringValidator'  
interface  
export * from './ZipCodeValidator'; // exports 'ZipCodeValidator'  
class and 'numberRegexp' constant value  
export * from './ParseIntBasedZipCodeValidator'; // exports the  
'ParseIntBasedZipCodeValidator' class
```



Un monde « tout composant »

Ne pas hésiter à consulter : <https://www.reactnative.express>

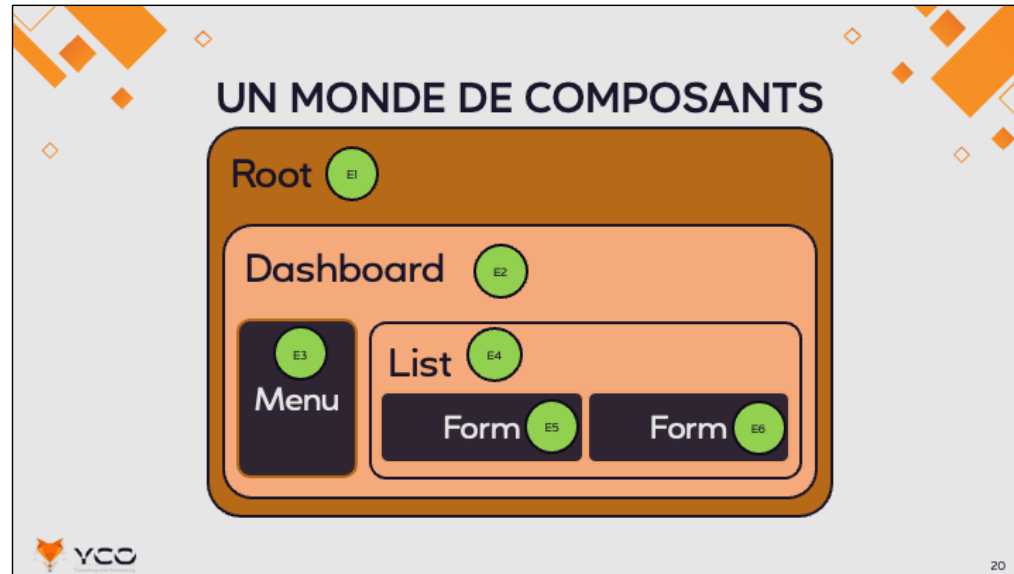


En React on crée techniquement une application One Page, qui va simuler la navigation via un système interne de routers.

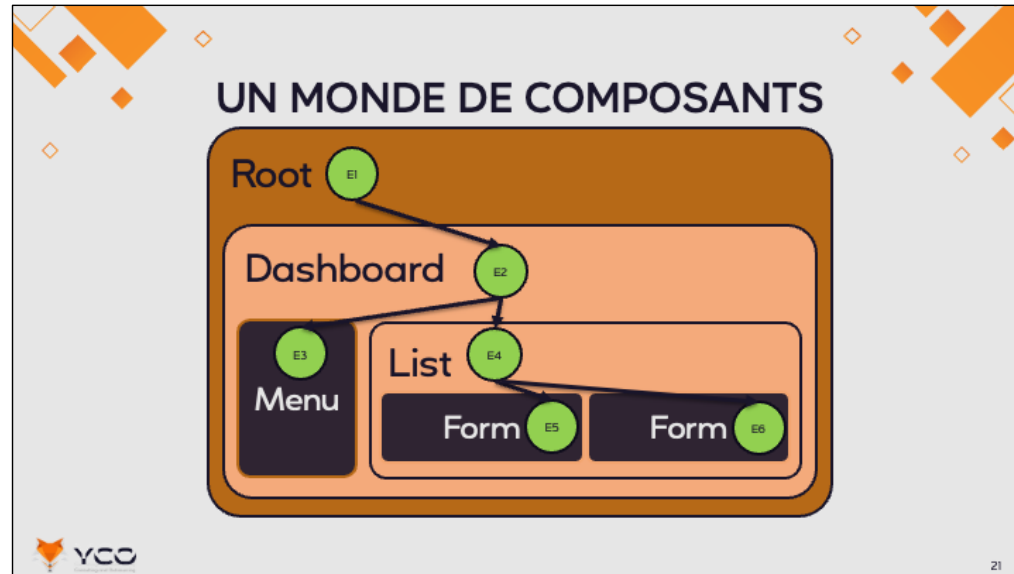
On va définir des composants qui sont : Une vue + un comportement (ViewModel), mêlés dans un même fichier.

Contrairement à Angular, React n'impose pas de structure applicative. Tout est composant. Le but est de découper les composants de manière à ce qu'ils contiennent le moins de code possible et à utiliser un maximum la composition de composants pour les imbriquer les uns dans les autres.

Les composants sont réutilisables et peuvent apparaître autant de fois qu'on le désire sur un même écran ou dans différents écrans.

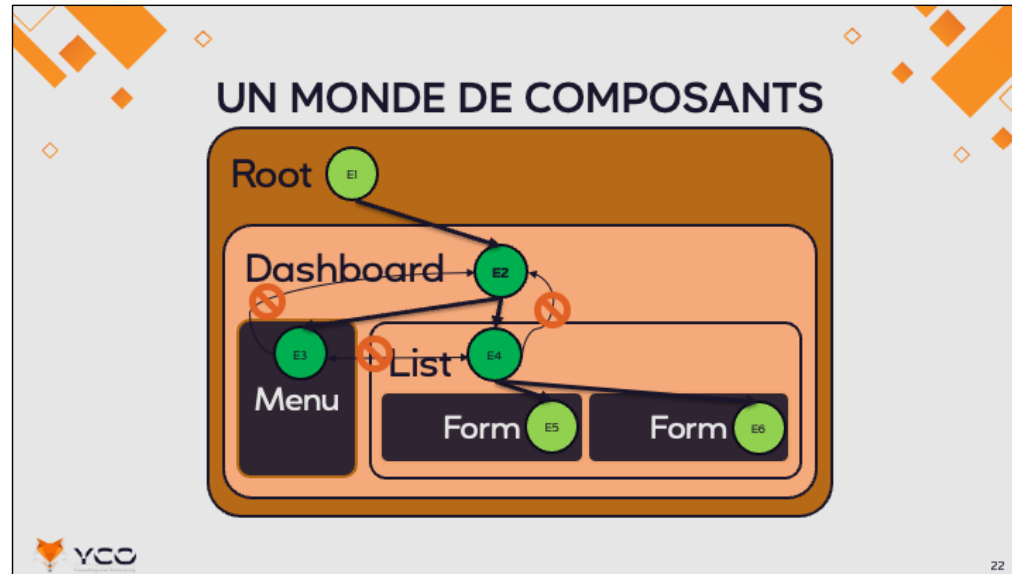


Chaque composant maintient un état local, avec ses propres variables



Pour transmettre une information d'un composant à l'autre, l'information peut uniquement descendre d'un composant parent à un composant enfant. L'idée est de déclarer la variable et les fonctions qui vont la manipuler dans le plus proche parent.

Dans cet exemple, si je veux faire varier un élément de ma liste quand je clique dans le menu, il me faut placer mon code dans le dashboard et faire descendre l'information dans les 2 composants.



Pour transmettre une information d'un composant à l'autre, l'information peut uniquement descendre d'un composant parent à un composant enfant. L'idée est de déclarer la variable et les fonctions qui vont la manipuler dans le plus proche parent.

Dans cet exemple, si je veux faire varier un élément de ma liste quand je clique dans le menu, il me faut placer mon code dans le dashboard et faire descendre l'information dans les 2 composants.

# PREMIÈRE APPLICATION

Générer une application Expo

```
$ npx
```

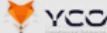
Choisir Navigation (TypeScript)

Installer les dépendances (dans le dossier qu'on vient de générer)

```
$ npx
```

Lancer l'application

```
$ npx
```

 YCO

23

- Si le projet est déjà créé et se base sur JS plutôt que TypeScript : <https://reactnative.dev/docs/typescript>

TP



Créer un application Expo sur votre poste



# STRUCTURE D'UNE APPLICATION EXPO

```
✓ COURSE-REACT
  ✓ app
    ✓ (tabs)
    18 _layout.jsx
    18 index.jsx
    18 two.jsx
    18 _layout.jsx
    18 [...missing].tsx
    18 model.tsx
  ✓ assets
    > fonts
    > images
  ✓ components
    > __tests__
    18 EditScreenInfo.tsx
    18 ExternalLink.tsx
    18 StyledText.tsx
    18 Themed.tsx
  ✓ constants
    18 Colors.ts
    > node_modules
    > .gitignore
    1 app.json
    1 babel.config.js
    18 index.ts
    1 package-lock.json
    1 package.json
    1 tsconfig.json
```



Under Construction



25

- TODO

# LES COMPOSANTS NATIFS

- View
- Layout
- Text
- Image
- Button
- Touchable
- ScrollView

Représente un conteneur natif d'affichage

Permet d'appliquer un style à tous ses sous-éléments



YCO

26

- Ils s'utilisent avec une balise qui correspond à leur nom et possèdent des propriétés de base accessible via l'auto complétion

# LES COMPOSANTS NATIFS

- View
- Layout
- Text
- Image
- Button
- Touchable
- ScrollView

Conteneur de style « flexbox »  
Permet d'organiser du contenu



# LES COMPOSANTS NATIFS

- View
- Layout
- Text
- Image
- Button
- Touchable
- ScrollView

Affiche un label

En général on en crée une surcharge qui prend  
En compte notre Theme



# LES COMPOSANTS NATIFS

- View
- Layout
- Text
- Image
- Button
- Touchable
- ScrollView

Affiche une image

En général on en crée plusieurs surcharges pour prendre en compte nos différents types d'affichages.



# LES COMPOSANTS NATIFS

- View
- Layout
- Text
- Image
- Button
- Touchable
- ScrollView

Affiche un bouton, par défaut il n'y a pas de fond/bordure mais un hover existe.



# LES COMPOSANTS NATIFS

View

Layout

Text

Image

Button

Touchable

ScrollView

Il y a 2 types de Touchable avec leur propriétés :

- TouchableOpacity (activeOpacity)
- TouchableHighlight (underlayColor)

Permet de définir un callback sur le onPress

Permet de créer des boutons avec plus de liberté



# LES COMPOSANTS NATIFS

View

Layout

Text

Image

Button

Touchable

ScrollView

Zone scrollable.

Si le contenu dépasse une scrollbar apparaît

Peut être horizontal/vertical/les deux.





# COMPOSANT PERSONNALISÉ

styled-text.component.tsx

```
import { Text, TextProps } from './Themed';  
  
export function MonoText(props: TextProps) {  
  return <Text {...props} style={[props.style, { fontFamily: 'SpaceMono' }] } />;  
}
```



33

- On peut surcharger des composants existants pour les spécialiser, ou en créer de nouveaux.
- On remarque une extension .tsx (ou .jsx si on n'utilise pas TypeScript)
  - C'est un mix entre du code et du markup
  - Le markup qui ressemble à du HTML génère automatiquement du code correspondant à la transpilation
- La façon maintenant recommandée est de créer des composants fonction ( « export function » ) mais il existe des composants sous forme de classe.
  - La syntaxe est simplifiée avec la syntaxe fonction
  - La syntaxe par classe va devenir deprecated
- Le nom de la fonction va déterminer la balise qui sera utilisable pour utiliser notre composant au sein d'un autre composant
  - Le nom de la fonction doit commencer par une Majuscule pour être reconnu par React comme un composant
  - Le premier paramètre contient toujours les « props » : ce sont les propriétés utilisables dans notre composant données par le composant parent. React fournit des interfaces standard, on peut définir le type que l'on souhaite.
- {...props} utilise l'opérateur spread pour dépiler tous les sous-paramètres qui sont passés dedans et les mettre à cet endroit.
- Dans ce « pseudo-HTML » pour utiliser des valeurs qui viennent du composant, il

faut utiliser des accolades « { } »

- Dans ces accolades on peut utiliser des tableaux « [] »
- Dans ces accolades on peut utiliser des objets avec une nouvelle paire d'accolades « {} »

## TP



Créer un composant personnalisé pour afficher du texte en couleur et l'afficher sur le premier écran de l'application.

On veillera à placer ce composant dans un Layout contenu dans une View.

Créer un bouton personnalisé à l'aide d'un Touchable

# LES COMPOSANTS LISTE NATIFS



# LES COMPOSANTS LISTE NATIFS

FlatList

SectionList

Liste avec notion de hiérarchie

Les items peuvent être homogènes mais ce n'est pas une obligation



## COMPOSANT FLAT LIST

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

const DATA = [{id: '1', title: 'First Item'}, {id: '2', title: 'Second Item'}, {id: '3', title: 'Third Item'}];

const Item = ({title}) => (
  <View style={styles.item}>
    <Text style={styles.title}>{title}</Text>
  </View>
);

const App = () => {
  return (
    <>
      <FlatList data={DATA} renderItem={({item}) => <Item title={item.title} />} keyExtractor={item => item.id} />
    </>
  );
};

const styles = StyleSheet.create({item: { backgroundColor: '#64EF37', padding: 20, marginVertical: 8, marginHorizontal: 16},
title: { fontSize: 32 },
});

export default App;
```



37

- On passe notre tableau de données au composant
- On explique au composant comment afficher un item
- Notre template fait référence à un composant défini plus haut

## COMPOSANT SECTION LIST

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

const DATA = [
  {
    title: 'Main dishes',
    data: ['Pizza', 'Burger', 'Risotto'],
  },
  {
    title: 'Sides',
    data: ['French Fries', 'Onion Rings', 'Fried Shrimps'],
  },
  {
    title: 'Drinks',
    data: ['Water', 'Coke', 'Beer'],
  },
  {
    title: 'Desserts',
    data: ['Cheese Cake', 'Ice Cream'],
  },
];

const App = () => (
  <>
    <SectionList sections={DATA} keyExtractor={({item, index}) => item + index}
      renderItem={({item}) => (
        <View>
          <Text>{item}</Text>
        </View>
      )}
      renderSectionHeader={({section: {title}}) => (
        <Text>{title}</Text>
      )} />
  </>
);

export default App;
```

- De la même manière que pour la liste simple on passe la datasource au composant mais il faut cette-fois lui donner les directives pour afficher les items mais aussi les sections.

TP



Utiliser une list dans votre composant principal qui utilise votre  
composant de texte coloré pour afficher ses items.



004

**REACT :  
ÉVÈNEMENTS &  
CONDITIONS**



# ÉVÈNEMENTS

app.component.tsx

```
import React from 'react'
import { Button } from 'react-native'

export default function App() {
  return (
    <Button
      title="Bouton de test"
      onPress={() => {
        console.log('TEST OK');
      }}
    />
  )
}
```



- Button propose nativement un évènement « onPress » auquel on peut raccorder une fonction anonyme pour exécuter un callback.

## ÉVÉNEMENTS PERSONNALISÉS

app.component.tsx

```
import React from 'react'
import { Button } from 'react-native'

function HelloButton({ title, sayHello }) {
  return <Button title={title} onPress={sayHello} />
}

export default function App() {
  return (
    <HelloButton
      title={ 'Click HERE to say hello' }
      sayHello={() => console.warn('HELLO')}
    />
  )
}
```



42

- C'est une sorte de cas particulier de l'utilisation des « props »
- On remarque que la fonction HelloButton n'a qu'un seul paramètre et que c'est un objet qui a 2 paramètres.
- En définissant le nom du paramètre de la fonction, on définit le nom du paramètre en « pseudo-HTML »

TP



Créer un évènement qui écrit dans la console et l'appeler  
depuis un bouton dans votre composant

## AFFICHAGE CONDITIONNEL - &&/||

app.component.tsx

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

const Card = ({ title, showButton }) => (
  <View>
    <Text style={{ fontSize: 60 }}>{title}</Text>
    {showButton && <Button title="Press me!" />}
  </View>
)

export default function App() {
  return (
    <View>
      <Card title="Title" showButton={false} />
      <Card title="Title with button" showButton={true} />
    </View>
  )
}
```

## AFFICHAGE CONDITIONNEL - TERNAIRE

app.component.tsx

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

const Card = ({ title, buttonText }) => (
  <View>
    <Text style={{ fontSize: 60 }}><title></Text>
    {buttonText ? <Button title={buttonText} /> : null}
  </View>
)

export default function App() {
  return (
    <View>
      <Card title="Title" />
      <Card title="Title with button" buttonText="Press me!" />
    </View>
  )
}
```

## AFFICHAGE CONDITIONNEL – IF/ELSE

app.component.tsx

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

const Card = ({ loading, error, title }) => {
  let content

  if (error) {
    content = <Text style={{ fontSize: 24, color: 'red' }}>Error</Text>
  } else if (loading) {
    content = <Text style={{ fontSize: 24, color: 'gray' }}>Loading...</Text>
  } else {
    content = (
      <View>
        <Text style={{ fontSize: 60 }}>{title}</Text>
      </View>
    )
  }

  return <View style={{ padding: 24 }}>{content}</View>
}

export default function App() {
  return (
    <View>
      <Card error={true} />
      <Card loading={true} />
      <Card loading={false} title="Title" />
    </View>
  )
}
```

- Pour utiliser la syntaxe if/else, on doit séparer la logique de l'affichage (même si c'est au sein du même fichier)
- Ici, dans le composant Card, on définit le contenu de la variable content dans notre structure if/else et on l'utilise dans son return

TP



Conditionnez l'affichage de votre bouton à un paramètre du composant



# LISTES

```
list.component.tsx
import React from 'react'
import { View, Text } from 'react-native'

const data = [
  { id: 'a', name: 'Devin' },
  { id: 'b', name: 'Gabe' },
  { id: 'c', name: 'Kim' },
]

export default function App() {
  return (
    <View>
      {data.map((item) => (
        <Text key={item.id}>{item.name}</Text>
      ))}
    </View>
  )
}
```

```
<View>
  <Text key="a">Devin</Text>
  <Text key="b">Gabe</Text>
  <Text key="c">Kim</Text>
</View>
```



48

- Utilisation de la fonction « map » pour générer du template
  - La fonction map est une fonction standard de TypeScript
  - Elle crée une copie modifiée d'une liste en appliquant une même transformation à chaque élément
  - Permet de mapper un type vers un autre par exemple
- La propriété « key » est comme « id » en HTML, elle permet d'identifier de manière unique un élément
  - Si on ne la remplit pas, React crée une sorte d'arborescence d'ids comme une liste numérotée, à plusieurs niveaux :
  - View: 0
    - Text: 0.0
    - Text: 0.1
    - Text: 0.2
  - On peut demander à React d'utiliser un index généré automatiquement (cf. slide suivante)

# LISTES

list.component.tsx

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

const data = [
  { id: 'a', name: 'Devin' },
  { id: 'b', name: 'Gabe' },
  { id: 'c', name: 'Kim' },
]

export default function App() {
  return (
    <View>
      {data.map((item, index) => (
        <Text key={index.toString()}>{item.name}</Text>
      ))}
    </View>
  )
}
```

<View>  
 <Text key="0">Devin</Text>  
 <Text key="1">Gabe</Text>  
 <Text key="2">Kim</Text>  
</View>



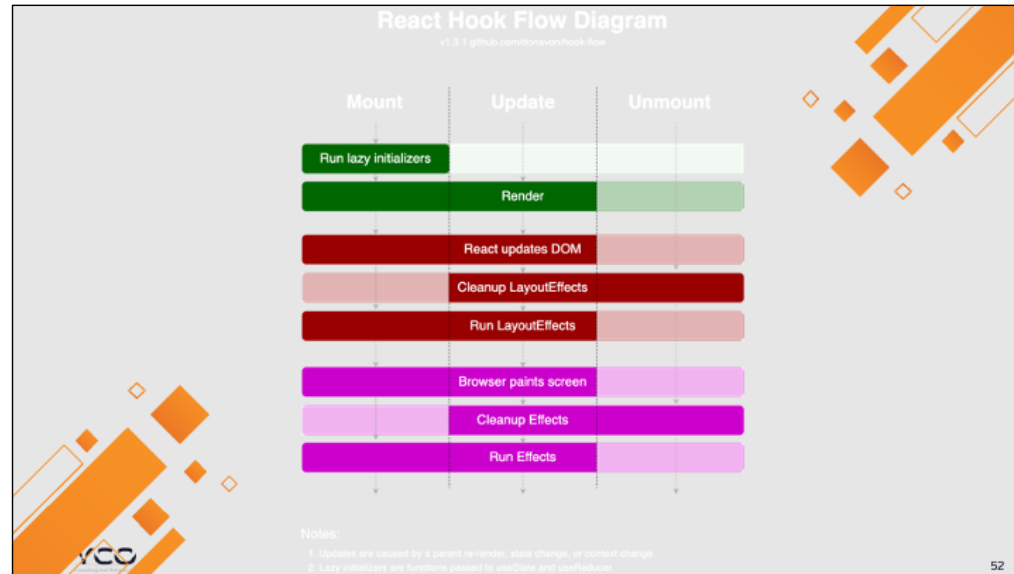
TP



Afficher une liste dans un template React

# REACT : HOOKS

005



- Le cycle de vie des composants React se compose de 3 phases:
  - Mount : Initialisation quand on affiche le composant
  - Update : Déclenché sur chaque modification de données
  - Unmount : Quand le composant ne fait plus partie de la page affichée
- Les Hooks sont des fonctions particulières qui sont appelées par React à certains moments de son cycle de vie pour nous permettre d'agir
- Dans chaque composant les Hooks sont toujours appelés dans le même ordre, c'est important



# LES HOOKS

## DO

- ✓ Appeler depuis la racine d'un fonction composant
- ✓ Appeler depuis un Hook personnalisé
- ✓ Utiliser le linter pour forcer l'utilisation correcte

## DON'T

- Appeler à l'intérieur d'une condition
- Appeler dans une boucle
- Appeler depuis une fonction en dehors d'un composant



# USESTATE

Persiste une donnée entre 2 rendus

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

const randomDiceRoll = () => Math.floor(Math.random() * 6) + 1;

export default function App() {
  const [diceRolls, setDiceRolls] = useState([]);

  return (
    <View>
      <Button title="Roll dice!" onPress={() => {
        setDiceRolls([...diceRolls, randomDiceRoll()])
      }} />
      {diceRolls.map((diceRoll, index) => (
        <Text style={{ fontSize: 24 }} key={index}>
          {diceRoll}
        </Text>
      ))}
    </View>
  )
}
```

YCO

54

Sur chaque update de données, on refait un rendu de notre composant.

useState nous permet d'avoir une variable dont la valeur est préservée d'un rendu sur l'autre.

On ne peut pas faire d'affectation directe de la valeur, il faut passer par le setter créé à l'instanciation.

La modification de la valeur (via le setter) déclenche un nouveau rendu du composant.

Plusieurs concepts ici :

1. Affectation du résultat d'une fonction à un tableau : Cela crée 2 const : diceRolls, un tableau, et setDiceRolls un callback qui permet d'affecter une nouvelle valeur à diceRolls
2. Affectation d'une valeur par défaut
3. Remplacement de la valeur existante par une nouvelle (donc pour ajouter une valeur au tableau on utilise l'opérateur spread)

Il est possible d'utiliser autant de useState que l'on veut sur un composant, il suffit de les mettre les uns à la suite des autres au début du composant.

TP



Créer un compteur que l'on peut incrémenter et décrémenter  
via des boutons : +1, +2, -1, -2 et Reset



# USEREF

Persiste une donnée entre 2 rendus

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

const randomDiceRoll = () => Math.floor(Math.random() * 6) + 1;

export default function App() {
  const diceRolls = useRef([]);

  return (
    <View>
      <Button title="Roll dice!" onPress={() => {
        diceRolls.current = [...diceRolls, randomDiceRoll()];
      }} />
      {diceRolls.map((diceRoll, index) => (
        <Text style={{ fontSize: 24 }} key={index}>
          {diceRoll}
        </Text>
      ))}
    </View>
  )
}
```

YCO

56

useRef permet comme useState de conserver une valeur d'un rendu du composant à l'autre.

Contrairement à useState, il ne déclenche pas un nouveau rendu quand on modifie la valeur.

Contrairement à useState, on peut le réaffecter directement.

Cela permet de différer l'effet de la modification, et de gagner en performances. (Par exemple appliquer les modifications à la validation d'un formulaire plutôt que chaque touche tapée).

TP



Sur le compteur précédent, utiliser `useRef`, pour ne mettre à jour  
l'affichage que lorsque l'on clique sur un bouton « Mettre à  
jour »

# USERREDUCER

Persiste une donnée complexe ou plusieurs états

```
function ShoppingList() {
  const inputRef = useRef();
  const [items, dispatch] = useReducer((state, action) => {
    switch (action.type) {
      case 'add': return state.push(action.payload);
      case 'remove': // keep every item except the one we want to remove
        return state.filter((_, index) => index !== action.index);
      default:
        return state;
    }
  }, []);

  function handleSubmit(e) { /*...*/ }

  return (
    <>
      <form onSubmit={handleSubmit}>
        <input ref={inputRef} />
      </form>
      <ul> {items.map((item, index) => (
        <li key={item.id}>
          {item.name} <button onClick={() => dispatch({ type: 'remove', index })}>X</button>
        </li> ) )}
      </ul>
    </>
  );
}
```

YCO

58

Le plus souvent le corps du reducer sera composé d'un switch case.

Le but est d'appeler une seule fonction avec un paramètre différent pour réaliser différentes actions : mettre à jour une propriété différente d'un même objet, effectuer différentes actions sur un même objet/tableau.

On note la présence de tags vides : ce sont des raccourcis pour `<React.Fragment>` et `</React.Fragment>`, qui servent de wrapper car un composant React ne doit renvoyer qu'un seul élément (qui peut lui en contenir plusieurs)

TP



Reprenez le composant liste créé au chapitre précédent et ajoutez-y des boutons pour ajouter/supprimer des lignes.  
Gérer l'action de ces boutons via useReducer.

# USE EFFECT

Modifie le DOM sans persistance

```
import React, { useState, useEffect } from 'react'
import { Button } from 'react-native'

export default function App() {
  const [count, setCount] = useState(0)
  const countEvery3 = Math.floor(count / 3)

  useEffect(() => {
    console.log(countEvery3)
  }, [countEvery3])

  return (
    <Button title={`Increment ${count}`} onPress={() => {
      setCount(count + 1)
    }} />
  )
}
```

YCO

60

Comme pour les autres hooks, il est appelé à chaque rendu du composant.

Il sert à modifier l'état ou le template du composant mais ne renvoie ni ne stocke pas de valeur : effet pour « effet de bord ».

Il appelle son callback (premier paramètre) lors du rendu du composant.  
Les `useEffect` sont appelés dans l'ordre mais de manière asynchrone.

Ici chaque appel à `setCount` demande un nouveau rendu, donc la valeur de « `countEvery3` » est mise à jour puis le `useEffect` est appelé.  
Pour n'afficher le log qu'une fois sur 3, il faut donc trouver autre chose.

Il est possible de ne pas le lancer à chaque rendu grâce au deuxième paramètre :

- S'il est rempli, c'est un tableau qui contient la liste des variables qui vont trigger le `useEffect` quand elles sont modifiées
- S'il est défini mais que le tableau est vide, le `useEffect` ne sera déclenché qu'au premier chargement du composant
- S'il n'est pas défini (ou `undefined`), par défaut, il sera appelé à chaque rendu du composant.

TP



Utiliser `useEffect` et l'affichage conditionnel pour afficher un loader à l'affichage de votre composant principal, qui sera effacé lorsque `useEffect` modifiera une variable booléenne.

# HOOK PERSO – USEINTERVAL

Création d'un  
hook périodique

```
import React, { useState, useEffect, useRef } from 'react'
import { Text } from 'react-native'
```

```
function useInterval(callback, delay) {
  const savedCallback = useRef()
```

```
  // Remember the latest callback.
  useEffect(() => {
    savedCallback.current = callback
  }, [callback])
```

```
  // Set up the interval.
  useEffect(() => {
    if (delay !== null) {
      let id = setInterval(() => {
        savedCallback.current()
      }, delay)
      return () => clearInterval(id)
    }
  }, [delay])
}
```

```
export default function App() {
  const [count, setCount] = useState(0)
```

```
  useInterval(() => {
    setCount(count + 1)
  }, 1000)
```

```
  return <Text style={{ fontSize: 120 }}>{count}</Text>
}
```



TP



Créer un hook personnalisé `useInterval` pour afficher le nb de secondes depuis l'ouverture de l'application.





Pour ce chapitre, on pourra se référer à <https://reactnavigation.org/docs/>

# LES NAVIGATEURS

SwitchNavigator

BottomTabNavigator

StackNavigator

DrawerNavigator

Le plus simple, chaque nouvel écran remplace le précédent

Pas d'éléments d'UI particuliers

Le bouton « précédent » renvoie sur le composant principal (pas d'historique)

YCO

65

- Vidéo de <https://inspeerity.com/blog/integrating-react-native-navigators#:~:text=The%20four%20basic%20navigators%20in,Switch%20Navigator%2C%20and%20Drawer%20Navigator.>

# LES NAVIGATEURS

SwitchNavigator

BottomTabNavigator

StackNavigator

DrawerNavigator

Navigateur par onglets, il rajoute une navbar en bas.

Pas d'historique.

L'onglet sélectionné remplace l'affichage

YCO

66

- Vidéo de <https://dev.to/easybuoy/combining-stack-tab-drawer-navigations-in-react-native-with-react-navigation-5-da>

# LES NAVIGATEURS

- SwitchNavigator
- BottomTabNavigator
- StackNavigator
- DrawerNavigator

Un nouvel écran est « empilé » par-dessus le précédent, un historique est conservé.

Un élément UI de navigation apparaît quand un écran est superposé.

Le bouton « précédent » du smartphone revient à l'écran « en dessous ».



YCO

67

- Vidéo de <https://dev.to/easybuoy/combining-stack-tab-drawer-navigations-in-react-native-with-react-navigation-5-da>

# LES NAVIGATEURS

SwitchNavigator

BottomTabNavigator

StackNavigator

DrawerNavigator

Navigateur par menu accessible au swipe.  
Pas d'historique.  
L'affichage est remplacé par le nouvel écran

Home

This is the home screen  
[Go to About Screen](#)

YCO

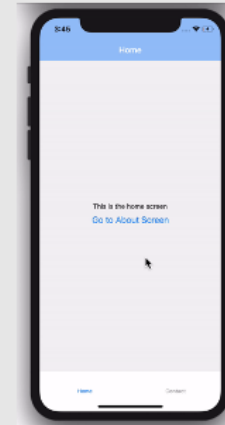
68

- Vidéo de <https://dev.to/easybuoy/combining-stack-tab-drawer-navigations-in-react-native-with-react-navigation-5-da>

## IMBRIQUER DES NAVIGATEURS

Il est possible d'imbriquer tous les types de navigateurs pour obtenir une interface complète.

Chaque navigateur agit de manière indépendante et peut lui-même imbriquer d'autres navigateurs.



- Vidéo de <https://dev.to/easybuoy/combining-stack-tab-drawer-navigations-in-react-native-with-react-navigation-5-da>

# IMPLÉMENTER LA NAVIGATION

```
[...]  
function RootLayoutNav() {  
  const colorScheme = useColorScheme();  
  
  return (  
    <>  
      <ThemeProvider value={colorScheme === 'dark' ? DarkTheme : DefaultTheme}>  
        <Stack>  
          <Stack.Screen name="(tabs)" options={{ headerShown: false }} />  
          <Stack.Screen name="modal" options={{ presentation: 'modal' }} />  
        </Stack>  
      </ThemeProvider>  
    </>  
  );  
}  
[...]
```



On implémente ici une navigation principale sous forme de stack, et on voit que le premier écran comporte des tabs.

# IMPLÉMENTER LA NAVIGATION

```
export default function TabLayout() {
  const colorScheme = useColorScheme();

  return (
    <Tabs screenOptions={{tabBarActiveTintColor: Colors[colorScheme ?? 'light'].tint,}}>
      <Tabs.Screen name="index" options={{title: 'Tab One', tabBarIcon: ({ color }) => <TabBarIcon name="code" color={color} />,
        headerRight: () => (
          <Link href="/modal" asChild>
            <Pressable>
              {{{ pressed }}} => (
                <FontAwesome name="info-circle" size={25} color={Colors[colorScheme ?? 'light'].text}
                  style={{ marginRight: 15, opacity: pressed ? 0.5 : 1 }}/>
              )
            </Pressable>
          </Link>
        ),
      }}
    </Tabs>
    <Tabs.Screen name="two" options={{title: 'Tab Two', tabBarIcon: ({ color }) => <TabBarIcon name="code" color={color} />}}/>
  </Tabs>
);
}
```

On implémente ici une navigation par tabs qui va être imbriquée dans la Stack nav principale.



## DÉCLENCHER LA NAVIGATION

```
import * as React from 'react';
import { Button, View, Text } from 'react-native';
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';

function HomeScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text>Home Screen</Text>
      <Button title="Go to Details" onPress={() => navigation.navigate('Details')} />
    </View>
  );
}
```

## PASSER DES PARAMÈTRES DE NAV

```
function HomeScreen({ navigation }) {
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text>Home Screen</Text>
      <Button title="Go to Details" onPress={() => { navigation.navigate('Details', { itemId: 86,
                                                                                   otherParam: 'anything you want here',
                                                                                   });
      }} />
    </View>
  );
}

function DetailsScreen({ route, navigation }) {
  const { itemId, otherParam } = route.params;
  return (
    <View style={{ flex: 1, alignItems: 'center', justifyContent: 'center' }}>
      <Text>Details Screen</Text>
      <Text>itemId: {JSON.stringify(itemId)}</Text>
      <Text>otherParam: {JSON.stringify(otherParam)}</Text>
      <Button title="Go to Details... again" onPress={() => navigation.push('Details', {
                                                                                   itemId: Math.floor(Math.random() * 100),
                                                                                   })} />
      <Button title="Go to Home" onPress={() => navigation.navigate('Home')} />
      <Button title="Go back" onPress={() => navigation.goBack()} />
    </View>
  );
}
```



TP



Créez un TabNavigator sur votre composant principal pour naviguer entre 2 composants.

Dans le premier composant, utilisez un StackNavigator pour afficher un message par-dessus au clic sur un bouton.



Tenir compte du Light Mode/ Dark Mode

## CONFIGURER LE PROJET

```
app.json
{
  "expo": {
    "userInterfaceStyle": "automatic",
    ...
    "ios": {
      ...
    },
    "android": {
      "adaptiveIcon": {
        ...
      }
    },
    "web": {
      ...
    }
  }
}
```



76

Il faut bien veiller à ce que la ligne `userInterfaceStyle` soit présente dans le fichier `app.json` dans la section « expo » pour qu'il gère les dépendances liées.  
Si le projet n'a pas été créé via Expo, se référer à la doc :  
<https://docs.expo.dev/guides/color-schemes/>

## CRÉER LES THÈMES

constants/themes.tsx

```
const tintColorLight = '#2f95dc';
const tintColorBlue = '#6677dc';
const tintColorDark = '#fff';

export default {
  light: {
    dark: false,
    colors: {
      primary: 'rgb(0, 122, 255)',
      card: 'rgb(255, 255, 255)',
      border: 'rgb(216, 216, 216)',
      notification: 'rgb(255, 59, 48)',
      text: '#000',
      background: '#fff',
      tint: tintColorLight,
      tabIconDefault: '#ccc',
      tabIconSelected: tintColorLight,
    },
  },
  dark: {
    dark: true,
    colors: {
      primary: 'rgb(0, 122, 255)',
      card: 'rgb(0, 0, 0)',
      border: 'rgb(216, 216, 216)',
      notification: 'rgb(255, 59, 48)',
      text: '#fff',
      background: '#333',
      tint: tintColorDark,
      tabIconDefault: '#333',
      tabIconSelected: tintColorDark,
    },
  },
  blue: {
    dark: false,
    colors: {
      primary: 'rgb(0, 10, 100)',
      card: 'rgb(50, 150, 190)',
      border: 'rgb(216, 216, 216)',
      notification: 'rgb(255, 59, 48)',
      text: '#338',
      background: '#c8ccf7',
      tint: tintColorBlue,
      tabIconDefault: '#ccc',
      tabIconSelected: tintColorBlue,
    },
  },
};
```

77

On définit dans un fichier de constantes tous nos thèmes, de manière à pouvoir utiliser ces couleurs partout sans avoir à les réécrire, et centraliser la modification.

La forme avec le booléen dark et les propriétés de primary à background sont obligatoires.

# CRÉER DES COMPOSANTS STYLISÉS

components/Themed.tsx

```
import { Text as DefaultText, useColorScheme,
  View as DefaultView } from 'react-native';
import Colors from '../constants/Colors';

export function useThemeColor(
  props: { themeName?: keyof typeof Colors },
  colorName: keyof typeof Colors.light.colors & keyof typeof
    Colors.dark.colors & keyof typeof Colors.blue.colors
) {
  const themeNameFromProps = props.themeName ?? null;
  const theme = themeNameFromProps ?? useColorScheme() ?? 'light';
  return Colors[theme] ['colors'][colorName];
}

type ThemeProps = {
  themeName?: keyof typeof Colors
};
export type TextProps = ThemeProps & DefaultText['props'];
export type ViewProps = ThemeProps & DefaultView['props'];

export function Text(props: TextProps) {
  const { style, ...otherProps } = props;

  if (themeName) {
    const color = useThemeColor({ themeName: props.themeName }, 'text');
    return <DefaultText style={[{ color }, style]} {...otherProps} />;
  } else {
    const color = useThemeColor({ }, 'text');
    return <DefaultText style={[{ color }, style]} {...otherProps} />;
  }
}

export function View(props: ViewProps) {
  const { style, ...otherProps } = props;

  if (themeName) {
    const backgroundColor = useThemeColor(
      { themeName: props.themeName }, 'background');
    return <DefaultView style={[{ backgroundColor }, style]}
      {...otherProps} />;
  } else {
    const backgroundColor = useThemeColor({ }, 'background');
    return <DefaultView style={[{ backgroundColor }, style]}
      {...otherProps} />;
  }
}
```

On définit une fonction personnalisée `useThemeColor` : si on lui passe un nom de thème, c'est celui qui sera utilisé, sinon il prendra le résultat de `useColorScheme()` qui est fourni par React, et en dernier recours, le thème « light ».

Le type de la propriété (`keyof...`) va chercher la liste des thèmes que l'on a défini à la diapositive précédente.

On override les props par défaut de `Text` et `View`, pour ajouter le nom du thème dans leurs propriétés.

On override les composants `Text` et `View` natifs et on les réexporte, il faudra les importer de ce fichier plutôt que de React native pour les utiliser dans l'application.

## APPLIQUER LE THÈME DANS L'APP

\_layout.tsx

```
...
function RootLayoutNav() {
  const colorScheme = useColorScheme();

  return (
    <>
      <ThemeProvider value={colorScheme === 'dark' ? Colors.dark : Colors.light}>
        <Stack>
          <Stack.Screen name="(tabs)" options={{ headerShown: false }} />
          <Stack.Screen name="modal" options={{ presentation: 'modal' }} />
        </Stack>
      </ThemeProvider>
    </>
  );
}
```



79

De cette manière le thème sera appliqué partout dans l'application.

Pour aller plus loin il est possible de sauvegarder en localStorage le thème choisi par l'utilisateur et l'appliquer en priorité.



TP



Créer des thèmes light et dark et les appliquer à votre application de manière automatisée.

# STOCKAGE PERSISTANT

Installer le package async-storage

```
$ npx
```

Si vous avez un projet macOS avec cocoa-pods

```
$ npx
```



<https://react-native-async-storage.github.io/async-storage/docs/install>

## STOCKER/RÉCUPÉRER LE THÈME

moncomposant.tsx

```
import AsyncStorage from '@react-native-async-storage/async-storage';

async function storeThemeName(value: string) {
  try {
    await AsyncStorage.setItem('themeName', value)
  } catch (e) {
    // saving error
  }
}

async function getThemeName(): Promise<string> {
  try {
    const value = await AsyncStorage.getItem('themeName')
    if(value !== null) {
      return value;
    } else { throw new Error(); }
  } catch(e) {
    // error reading value
    throw new Error();
  }
}
```



B2

En utilisant async-storage, on peut dans le useEffect/useState d'un composant, stocker le nom du thème à utiliser, et à chaque fois qu'on en aura besoin, il suffira de récupérer le résultat de getThemeName() dans une variable d'état de n'importe quel autre composant.

TP



Créer un composant avec une liste de vos thèmes (keyof ...) et  
au clic, stocker le nom du thème dans le async-storage puis  
l'appliquer à votre application.

008

REACT :  
CI/CD





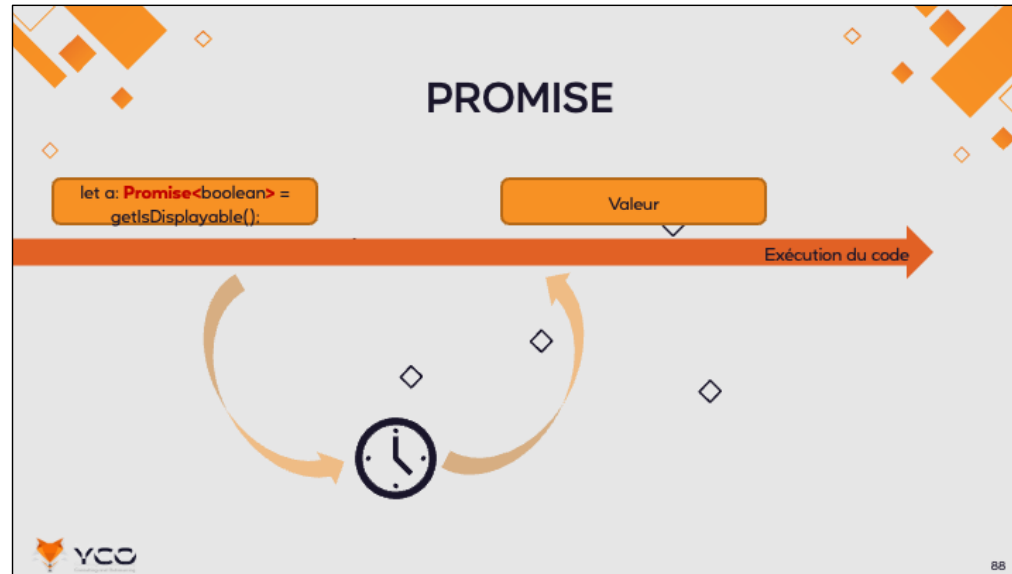


Quand on va chercher un résultat en base de données, cela peut prendre du temps et on ne veut pas forcément bloquer l'affichage pendant ce temps, comment s'y prendre ?



Les Promises sont également présentes en JS depuis ES6 mais peu utilisées en vanilla.





Une Promise est typée. On connaît à l'avance le type de retour et on pourra l'utiliser plus tard.

Lorsque l'on demande la valeur, on passe immédiatement à l'exécution du reste des instructions sans attendre le résultat

Si l'on n'a pas besoin d'utiliser le résultat directement dans le code (par exemple directement bindé à la vue), on peut s'arrêter là.

# PROMISE

```
async getRandomInt(): Promise<number> {  
  return new Promise<number>((resolve, reject) => {  
    const value = parseInt((Math.random() *  
      10).toFixed(0));  
  
    if (value === 0) {  
      reject('Ne fonctionne pas pour 0.');    }  
  
    resolve(value);  
  });  
}
```

```
async getRandomInt2(): Promise<number> {  
  const value = parseInt((Math.random() *  
    10).toFixed(0));  
  
  if (value === 0) {  
    throw new Error('Ne fonctionne pas pour 0.');  }  
  
  return value;  
}
```

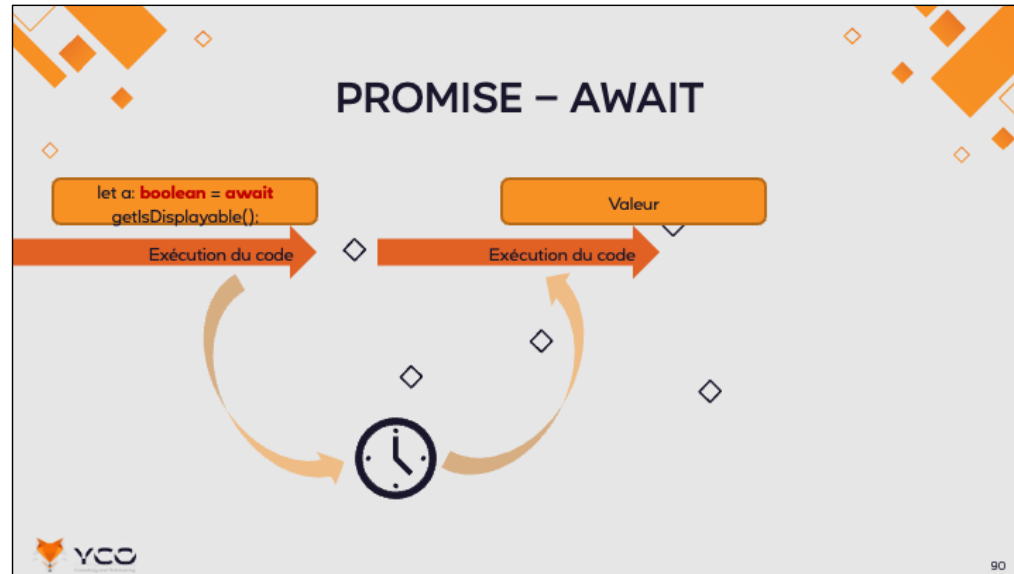


89

La 2<sup>ème</sup> forme est plus idiomatique en TypeScript

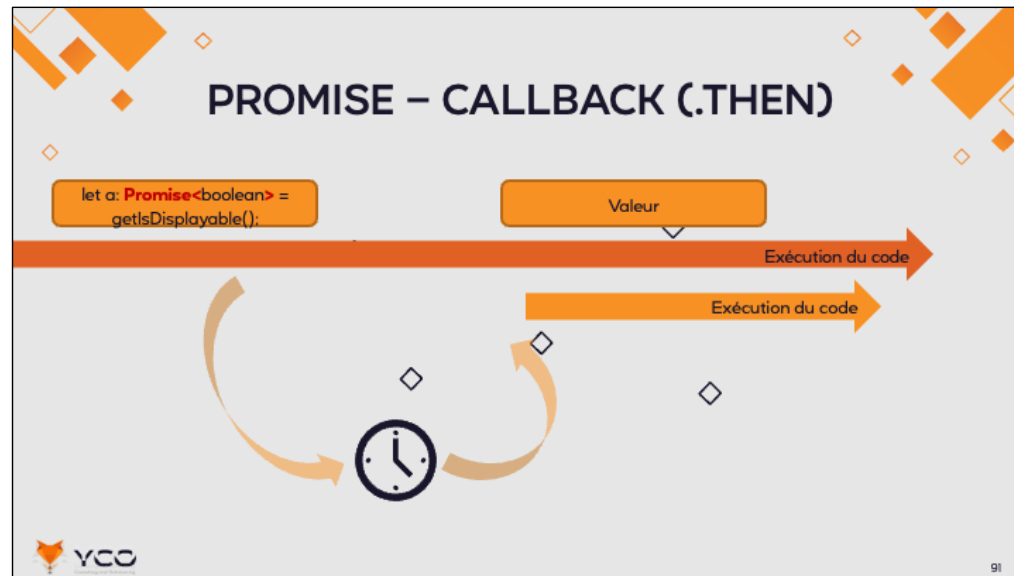
Ici c'est une déclaration de fonction mais ce peut aussi être assigné à une constante :

```
const truc: Promise<number> = () => {  
  const value = parseInt((Math.random() *  
    10).toFixed(0));  
  
  if (value === 0) {  
    throw new Error('Ne fonctionne pas pour 0.');  }  
  
  return value;  
};
```



Pour utiliser le résultat dans son code, 2 solutions :

On resynchronise le processus, l'exécution s'arrête pendant que la valeur est récupérée, on obtient donc le type demandé plutôt qu'une Promise du type demandé.



Une Promise est typée. On connaît à l'avance le type de retour et on pourra l'utiliser plus tard.

Lorsque l'on demande la valeur, on passe immédiatement à l'exécution du reste des instructions sans attendre le résultat

## PROMISE – AWAIT/.THEN()

```
async uselessFunc() {  
  try {  
    let nb: number = await this.getRandomInt();  
    console.log('Vous avez' + (nb > 5 ? 'gagné'  
      : 'perdu'));  
  } catch (err: any) {  
    console.log('Erreur : ', err);  
  }  
}
```

```
uselessFunc2() {  
  this.getRandomInt().then(v => {  
    console.log('Vous avez' + (v > 5 ? 'gagné' :  
      'perdu'));  
  }).catch(err => {  
    console.log('Erreur : ', err);  
  }).finally(() => {  
    console.log('Merci d\'avoir joué')  
  });  
}
```

On ne peut utiliser await qu'à l'intérieur d'une fonction async.  
Si cette fonction doit renvoyer un résultat, il devra lui-même être await, etc.

Une fonction async renvoie forcément une Promise (même si rien n'est précisé).

On peut chaîner les .then() pour appeler plusieurs fois la fonction.



## SUBSCRIBE

```
import { Component, EventEmitter, OnDestroy, OnInit }
from '@angular/core';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-hero-list',
  templateUrl: './hero-list.component.html',
  styleUrls: ['./hero-list.component.css']
})
export class HeroListComponent implements OnInit,
  OnDestroy {

  mySubscribe: Subscription | undefined;
  emitter: EventEmitter<number> = new EventEmitter();
```

```
constructor() { }

ngOnDestroy(): void {
  if (this.mySubscribe) {
    this.mySubscribe.unsubscribe();
  }
}

ngOnInit(): void {
  if (!this.mySubscribe) {
    this.mySubscribe = this.emitter.subscribe(nb => {
      console.log(nb);
    });
  }
}
```



⚠ Ne pas oublier de unsubscribe (sur le OnDestroy en Angular) pour ne pas exécuter plusieurs fois la fonction de callback.

Chaque fois que notre code passe sur la ligne qui contient « .subscribe », le code du callback n'est pas exécuté, il est ajouté à la liste des codes à exécuter lorsque la valeur sera transmise par l'émetteur.