React Native

Développez des applications mobiles modernes

Commencer le TP

TP React Native

📱 Introduction

Bienvenue dans ce TP de React Native ! Vous allez apprendre à créer des applications mobiles natives en utilisant vos connaissances en React. Ce TP vous guidera à travers plusieurs exercices pratiques, du plus simple au plus complexe.

📚 Ressources

- Github: https://github.com/tanguyvans/tp-react-native
- Documentation: https://tanguyvans.github.io/tp-react-native/

🎯 Objectifs

À la fin de ce TP, vous serez capable de créer des applications mobiles complètes avec React Native. Vous maîtriserez :

Compétence	Description
🛠️ Setup	Configuration de l'environnement et des outils
📱 UI	Création d'interfaces utilisateur natives
🧩 Components	Création et réutilisation de composants
🔄 État	Gestion de l'état avec useState
🌐 API	Interaction avec des services externes
🔐 Auth	Implémentation de l'authentification

📚 Structure du TP

Le TP est divisé en exercices progressifs :

🚀 Exercice 0 : Lancement de l'application

Dans ce premier exercice, vous découvrirez l'environnement React Native. Vous apprendrez à configurer votre espace de travail, installer Expo Go sur votre téléphone et créer votre première application mobile.

🔢 Exercice 1 : Application Counter

Commencez par une application simple pour comprendre les bases de React Native. Vous créerez un compteur interactif qui vous permettra de maîtriser la gestion d'état avec useState et les interactions utilisateur de base.

✅ Exercice 2 : Application Todo List

Passez à un niveau supérieur avec une application de gestion de tâches. Vous apprendrez à gérer une liste d'éléments, utiliser le stockage local et implémenter les opérations CRUD (Create, Read, Update, Delete).

🌟 Exercice 3 : DevHub

Dans cet exercice final, vous créerez une application complète qui interagit avec l'API GitHub. Vous implémenterez l'authentification avec Supabase, créerez une interface moderne et gérerez la navigation entre différents écrans.

🛠️ Prérequis

Pour suivre ce TP, vous aurez besoin de :

- **Node.js** installé sur votre machine
- **VS Code** ou un autre éditeur de code
- **Smartphone** avec Expo Go installé

🚀 Commençons !

Choisissez un exercice dans le menu de gauche pour commencer. Bon développement !

Prérequis

Avant de commencer les exercices, assurez-vous d'avoir installé les outils et logiciels suivants sur votre machine.

Logiciels requis

Node.js

Vérifiez que vous avez bien la version 22.14.0 ou une version supérieure installée:

```
node -v
```

Si vous n'avez pas Node.js installé, vous pouvez le télécharger ici. Je recommande d'installer `nvm` (Node Version Manager). `nvm` est un outil qui vous permet d'installer et de gérer différentes versions de Node.js. Pour ce workshop, nous utiliserons la version 22.14.0.

Si vous avez déjà nvm installé, vous pouvez l'installer avec:

```
nvm install 22.14.0
```

Git

Assurez-vous d'avoir Git installé. Si ce n'est pas le cas, vous pouvez l'installer ici.

```
git --version
```

Environnement de développement React Native

En fonction de votre système d'exploitation, vous devrez installer différents outils. Suivez les instructions ici.

Pour ce workshop, nous utiliserons Expo. Installez l'application Expo Go sur votre téléphone.

Clonage du projet

Clonez le projet avec la commande suivante:

```
git clone https://github.com/Tanguyvans/tp-react-native.git
```

Ouvrez le projet avec votre environnement de développement React Native.

```
cd tp-react-native
```

Pour fork le projet:

```
git remote -v
```

Edit this page

Créer un projet

🎯 Objectifs

Dans ce premier exercice, vous apprendrez à :

Compétence	Description
🛠️ Installation	Créer un nouveau projet Expo React Native
📦 Dépendances	Gérer les dépendances avec npm
🚀 Déploiement	Lancer et tester votre application

🎯 Étape 1: Créer un nouveau projet

Avant que vous avez remplis les prerequis.

Utilisez la commande suivante pour créer un nouveau projet :

```
npx create-expo-app@latest [nom-du-projet]
```

📦 Étape 2 : Installer les dépendances

Naviguez vers le dossier du projet et installez les dépendances :

```
cd [nom-du-projet]
npm install
```

🚀 Étape 3 : Lancer le projet

Pour lancer le projet, utilisez la commande suivante. Assurez-vous que votre ordinateur et votre appareil mobile sont connectés au même réseau Wi-Fi.

```
npx expo start
```

🔧 Résolution des problèmes

Si vous rencontrez des problèmes de connexion, essayez :

```
npx expo start --tunnel
```

Cette commande créera un tunnel permettant d'accéder à votre projet depuis votre appareil mobile, même sur un réseau différent.

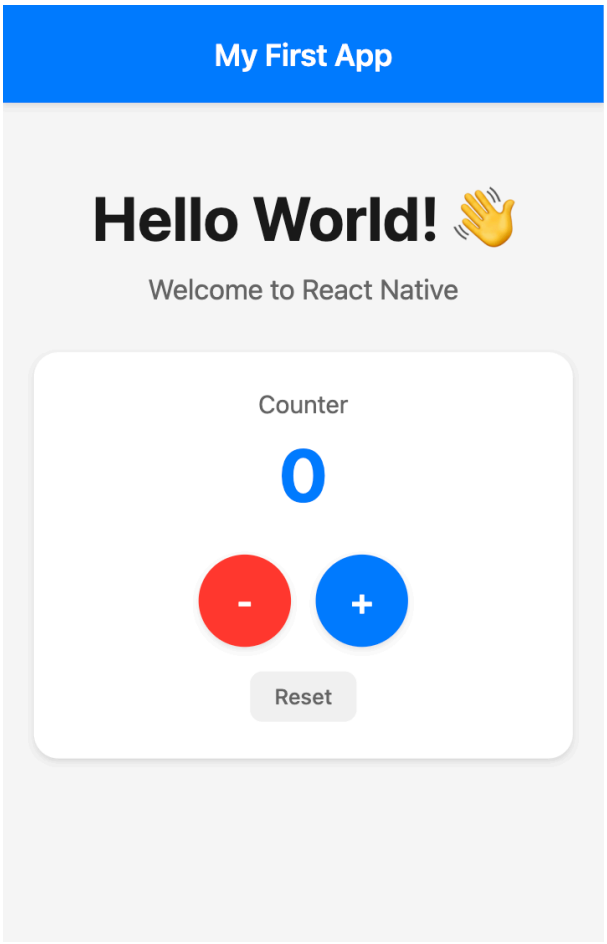
Hello World

🎯 Objectifs

Dans ce premier exercice, vous apprendrez à :

Compétence	Description
📱 Interface	Créer une interface utilisateur structurée avec en-tête et zone de contenu
🔄 État	Gérer l'état local avec useState pour un compteur interactif
💅 Style	Appliquer des styles professionnels et créer des composants réutilisables
🎨 Design	Implémenter des retours visuels et des animations de pression

Voici à quoi ressemblera votre application finale :



📋 Étape 1 : Lancer le projet

Commencez par naviguer vers le dossier de l'exercice et installer les dépendances :

```
cd 01-hello-world
npm install
```

Puis lancez le projet :

```
npx expo start
```

> **INFO**
>
> Si vous rencontrez des problèmes de connexion, essayez de lancer le projet avec le tunnel :
>
> ```
> npx expo start --tunnel
> ```

📱 Étape 2 : Créer la page Hello World

Votre projet tourne, mais il est vide. Ajoutons une page simple avec un composant texte. Allez dans le fichier `app/index.tsx` et ajoutez le code suivant :

```tsx
import { StyleSheet, Text, View } from "react-native";

export default function Page() {
  return (
    <View>
      <Text>Hello World</Text>
    </View>
  );
}
```

🔄 Étape 3 : Ajouter un compteur

Ajoutons de l'interactivité à notre application avec un compteur. Nous utiliserons le hook `useState` pour gérer l'état du compteur. Le hook `useState` permet de suivre les valeurs qui peuvent changer au fil du temps.

Importez le hook `useState` et ajoutez-le à votre composant :

```
import { StyleSheet, Text, View, Pressable, SafeAreaView } from "react-
native";
import { useState } from "react";
```

Ajoutez un composant `SafeAreaView` pour éviter que le contenu soit masqué par les trous noirs :

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

  return (
    <SafeAreaView>
      <View>
        <Text>Hello World</Text>
      </View>
    </SafeAreaView>
  );
}
```

> **INFO**
>
> `SafeAreaView` automatiquement ajoute un padding pour éviter que le contenu soit masqué par :
>
> - Les trous noirs sur les iPhones
> - Les coins arrondis

Maintenant que nous avons une variable `count`, nous pouvons l'utiliser dans le composant `Text` :

```
<Text>{count}</Text>
```

Créons des boutons pour contrôler le compteur. Nous utiliserons `Pressable` au lieu du composant `Button` de base car il offre de meilleures options de style et de retour visuel :

```
<View>
  <Pressable onPress={() => setCount(count + 1)}>
    <Text>Increment</Text>
  </Pressable>
</View>
```

Il est possible d'utiliser le composant `Button` de base, mais il ne permet pas de personnaliser les styles et les retours visuels :

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

🎯 Tâche

Créez un bouton de decrementation qui :

- Décrémente le compteur lorsqu'il est pressé
- Utilise le composant `Pressable`
- Utilise `Math.max(0, prev - 1)` pour éviter que le compteur soit négatif

Créez un bouton de réinitialisation qui :

- Réinitialise le compteur à 0 lorsqu'il est pressé
- Utilise le composant `Pressable`

🎨 Étape 4 : Appliquer des styles à l'application

Ajoutons des styles à notre application étape par étape :

À ce stade, votre code devrait ressembler à ceci :

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

  return (
    <SafeAreaView>
      <View>
        <Text>Hello World</Text>
      </View>
      <View>
        <Text>{count}</Text>
        <View>
          ...
          <Pressable onPress={() => setCount((prev) => prev + 1)}>
            <Text>+</Text>
          </Pressable>
          ...
        </View>
      </View>
```

```
    </SafeAreaView>
  );
}
```

Pour ajouter des styles, nous devons simplement créer une constante `styles` et ajouter les styles aux composants.

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: "#f5f5f5",
  },
  header: {
    backgroundColor: "#007AFF",
    padding: 20,
    alignItems: "center",
  },
  headerText: {
    color: "white",
    fontSize: 20,
    fontWeight: "600",
  },
  main: {
    flex: 1,
    alignItems: "center",
    justifyContent: "center",
    padding: 20,
  },
  counterNumber: {
    fontSize: 48,
    fontWeight: "bold",
    color: "#007AFF",
    marginBottom: 24,
  },
  buttonGroup: {
    flexDirection: "row",
    gap: 16,
    marginBottom: 16,
  },
  button: {
    width: 60,
    height: 60,
    borderRadius: 30,
    justifyContent: "center",
    alignItems: "center",
    backgroundColor: "#FF3B30",
  },
  incrementButton: {
    backgroundColor: "#007AFF",
```

```
  },
  buttonText: {
    color: "white",
    fontSize: 24,
    fontWeight: "600",
  },
  resetButton: {
    padding: 10,
    backgroundColor: "#f0f0f0",
    borderRadius: 8,
  },
  resetButtonText: {
    color: "#666",
    fontSize: 14,
  },
});
```

En React Native, vous pouvez combiner plusieurs styles en utilisant un tableau. Ceci est utile lorsque vous souhaitez :

- Appliquer des styles de base et des variations
- Ajouter des styles conditionnels
- Remplacer des propriétés spécifiques

Voici comment combiner des styles :

```
// Style de base application
<View style={styles.button}>
  <Text>Basic Button</Text>
</View>

// Combinaison de deux styles
<View style={[styles.button, styles.incrementButton]}>
  <Text>Combined Styles</Text>
</View>
```

Dans notre application de compteur, nous utilisons ceci pour créer différentes variations de boutons :

```
// Bouton de decrementation (rouge)
<Pressable style={styles.button}>
  <Text>-</Text>
</Pressable>

// Increment button (blue)
<Pressable style={[styles.button, styles.incrementButton]}>
```

```
  <Text>+</Text>
</Pressable>
```

Le second style dans le tableau remplacera les propriétés en double du premier style.

🎯 Essayez-vous-même !

1. Ajoutez les styles aux composants corrects
2. Créez une nouvelle variation de bouton avec une couleur différente
3. Essayez de combiner trois ou plus de styles
4. Créez un état sélectionné pour les boutons

DevNotes

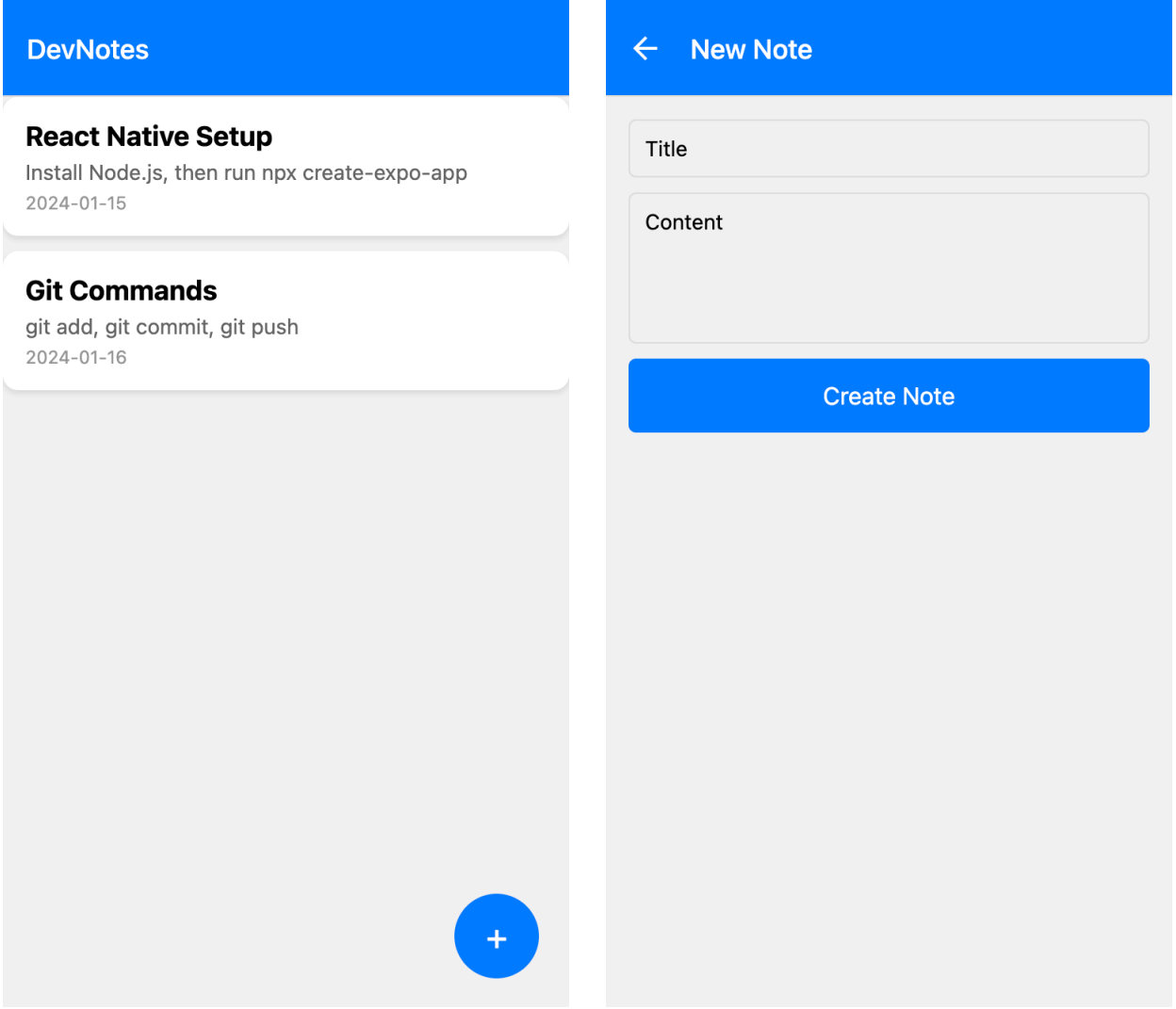
Dans ce deuxième exercice, nous allons créer une app de notes pour développeurs. Vous apprendrez :

🎯 Objectifs

Dans cet exercice, vous apprendrez à :

Compétence	Description
🧩 Composants	Créer et utiliser des composants réutilisables
📱 Navigation	Gérer la navigation entre différentes pages de l'application
🔄 Communication	Maîtriser le passage de données entre les écrans
📂 Architecture	Organiser efficacement la structure de votre code

Voici à quoi ressemblera votre app :



📋 Étape 0 : Lancer le projet

Commencez par naviguer vers le dossier de l'exercice et installer les dépendances :

```
cd 02-devnotes
npm install
```

Puis lancez le projet :

```
npx expo start
```

> **INFO**
>
> Si vous rencontrez des problèmes de connexion, essayez de lancer le projet avec le tunnel :
>
> ```
> npx expo start --tunnel
> ```

📋 Étape 1 : Création du composant

Commençons par créer le composant réutilisable qui affichera chaque note dans la liste.

1.1 Structure des types

Créez un fichier `types.ts` à la racine du projet. Nous pourrons ainsi faire appel au type `Note` dans tous les fichiers.

```
export type Note = {
  id: string;
  title: string;
  content: string;
  date: string;
};
```

1.2 Création du composant

Modifiez le composant `components/NoteCard.tsx` pour afficher les notes dans une card:

```
import { View, Text, Pressable, StyleSheet } from "react-native";
import { Link } from "expo-router";
import { Note } from "../types";

type Props = {
  note: Note;
};

export default function NoteCard({ note }: Props) {
  return (
    // TODO: Ajouter un lien vers la page de détail de la note
    // Faite l'affichage de la note dans une card: Pressable, View, Text
    <View></View>
  );
}
```

🎯 Tâche

Terminez le composant `NoteCard.tsx` pour qu'il affiche:

- Le titre de la note
- La date de la note
- Le contenu de la note

> **CONSEIL**
>
> Inspirez-vous du code de l'exercice précédent pour créer le composant `NoteCard`. Pour afficher le titre de la node vous devez faire appel à `note.title`.

1.3 FlatList

Pour afficher la liste des notes, nous allons utiliser le composant `FlatList`. Une `FlatList` est un composant qui permet de afficher une liste de données de manière dynamique. Dans app/index.tsx, nous allons ajouter un `FlatList` qui fera appel au composant `NoteCard`.

Ne pas oublier d'importer le composant `NoteCard.tsx` dans le fichier index.tsx.

```
import NoteCard from "../components/NoteCard";

export default function Home() {
  const [notes, setNotes] = useState<Note[]>(initialNotes);

  return (
    <View style={styles.container}>
      {/* TODO: La FlatList doit être ici */}
    </View>
  );
}
```

🎯 Tâche

Dans le fichier `index.tsx`, ajoutez un `FlatList` qui fera appel au composant `NoteCard`:

Pour ce faire allez dans la documentation officielle : FlatList

📝 Étape 2 : Page de détail d'une note

Maintenant que nous avons la liste des notes, nous allons créer la page de détail d'une note. Pour cela nous devons créer une page qui affichera les informations d'une note. Cependant, il faut faire cela de façon dynamique, c'est-à-dire que nous devons pouvoir accéder à la page de détail d'une note en utilisant son id.

2.1 Configuration du routage

Nous avons créé un dossier `notes` dans `app/`. Dans ce dossier il faut créer un fichier `[id].tsx` à l'intérieur. Pour l'affichage de la note voici un exemple **très basique** :

```
import { View, Text } from "react-native";
import { useLocalSearchParams } from "expo-router";

export default function NoteDetail() {
  const { id } = useLocalSearchParams();

  return (
    <View>
      <Text>Note Detail {id}</Text>
    </View>
  );
}
```

🎯 Tâche

- Créez un fichier `app/notes/[id].tsx` qui affichera les informations d'une note.
- Ajouter un `Text` qui affichera le titre de la note
- Ajouter un `Text` qui affichera la date de la note
- Ajouter un `Text` qui affichera le contenu de la note

2.2 Comment aller vers la page de détail d'une note

La page note est créée mais comment aller vers cette page ? Il faut ajouter un lien vers celle-ci pour chaque note de la page d'accueil. Pour cela nous allons utiliser le composant `Link` de `expo-router`.

```
export default function NoteCard({ note }: Props) {
  return (
    <Link href={`/notes/${note.id}`} asChild>
      {/* Code précédent */}
    </Link>
  );
}
```

🎯 Tâche

Allez dans le fichier `NoteCard.tsx` et ajoutez un lien vers la page de détail d'une note.

Pour ce faire allez dans la documentation officielle : Link

> **CONSEIL**
>
> Comme la page note est dynamique, il faut passer l'id de la note en paramètre. Voici comment définir un paramètre dynamique : `/notes/${note.id}`

> **INFO**
>
> A ce stade, vous pouvez accéder à la page de détail d'une note en cliquant sur une note de la liste. Cependant, il manque la barre de navigation en haut et le bouton retour automatique. Pour cela nous allons devoir utiliser le Layout.

2.3 Pourquoi un Layout ?

Le Layout joue un rôle crucial dans notre application. Voici la différence entre une navigation avec et sans Layout :

Sans Layout

- ❌ Pas de barre de navigation en haut
- ❌ Pas de bouton retour automatique
- ❌ Pas d'animations de transition
- ❌ Pas de gestion native du "swipe to go back" sur iOS

Avec Layout

- ✅ Une barre de navigation professionnelle
- ✅ Un bouton retour automatique
- ✅ Des animations de transition fluides
- ✅ Une gestion native des gestes

> **IMPORTANT**
>
> Sans Layout, votre application fonctionnera mais ne ressemblera pas à une vraie application mobile native !

✨ Étape 3 : Layout

le layout est le composant racine de l'application. Il est important de le créer pour que l'application fonctionne correctement. Il permet de gérer la navigation entre les différentes pages de l'application.

Créez un fichier `_layout.tsx` dans `app/` avec le code suivant:

```
import { Stack } from "expo-router";

export default function Layout() {
  return (
    <Stack
      screenOptions={{
        headerStyle: {
          backgroundColor: "#007AFF",
        },
        headerTintColor: "#fff",
      }}
    >
      <Stack.Screen name="index" options={{ title: "DevNotes" }} />
      <Stack.Screen name="notes/[id]" options={{ title: "Note Details" }} />
    </Stack>
  );
}
```

✨ Étape 4 : Nouvelle note

4.1 Formulaire de création

Dans le app/new/index.tsx, nous allons créer un formulaire pour créer une nouvelle note. Pour cela nous allons utiliser les composants `TextInput`. Le composant `Pressable` sera utilisé pour le bouton de création.

```
import { View, TextInput, Pressable, Text, StyleSheet } from "react-native";
import { useRouter } from "expo-router";
import { useState } from "react";
import { addNoteEvent } from "../index";

export default function NewNote() {
  const router = useRouter();
  const [title, setTitle] = useState("");
  const [content, setContent] = useState("");

  const handleCreate = () => {
    if (title && content) {
      // Émettre l'événement avec les données de la nouvelle note
      addNoteEvent.emit("newNote", { title, content });
```

```
      router.back();
    }
  };

  return (
    <View>
      <TextInput
        placeholder="Title"
        // TODO: Ajouter les éléments du formulaire
      />
      <TextInput
        placeholder="Content"
        // TODO: Ajouter les éléments du formulaire
        multiline
      />
      <Pressable>
        <Text>Create Note</Text>
      </Pressable>
    </View>
  );
}
```

🎯 Tâche

Créez un fichier `app/new/index.tsx` qui sera un formulaire pour ajouter une nouvelle note.

Il faut utiliser les composants `TextInput` et `Pressable`.

Utilisez la documentation officielle pour comprendre comment utiliser ces composants : TextInput et Pressable.

- Compléter `TextInput` pour le titre et le contenu
- Compléter le bouton de création

> **CONSEIL**
>
> Pour `textInput` utiliser les éléments `value` et `onChangeText` pour gérer les données. Il faut faire appel à la fonction `handleCreate` lorsque l'utilisateur appuie sur le bouton.

4.2 Ajout du bouton de création

🎯 Tâche

Dans, nous devons ajouter un bouton pour créer une nouvelle note:

- Ajouter un bouton dans `app/index.tsx`
- Mettez à jour le `app/_layout.tsx` pour ajouter la page de création de note

> **CONSEIL**
>
> Pour ajouter un lien, il suffit d'utiliser le composant `Link` de `expo-router`. C'est similaire à ce qui a été fait dans NoteCard.tsx.

4.3 Configuration du routage

Comme nous avons créé une nouvelle page, il faut la rendre accessible. Pour cela, il faut ajouter la page dans le layout. Inspirez-vous du code du layout pour ajouter la page.

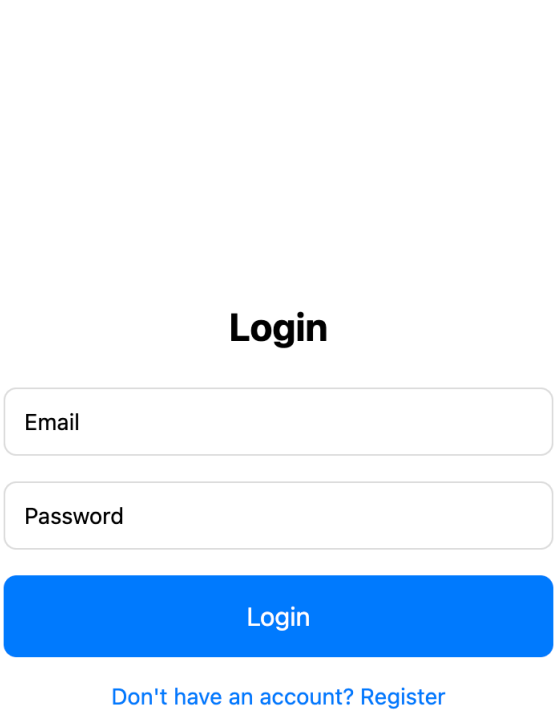
DevHub

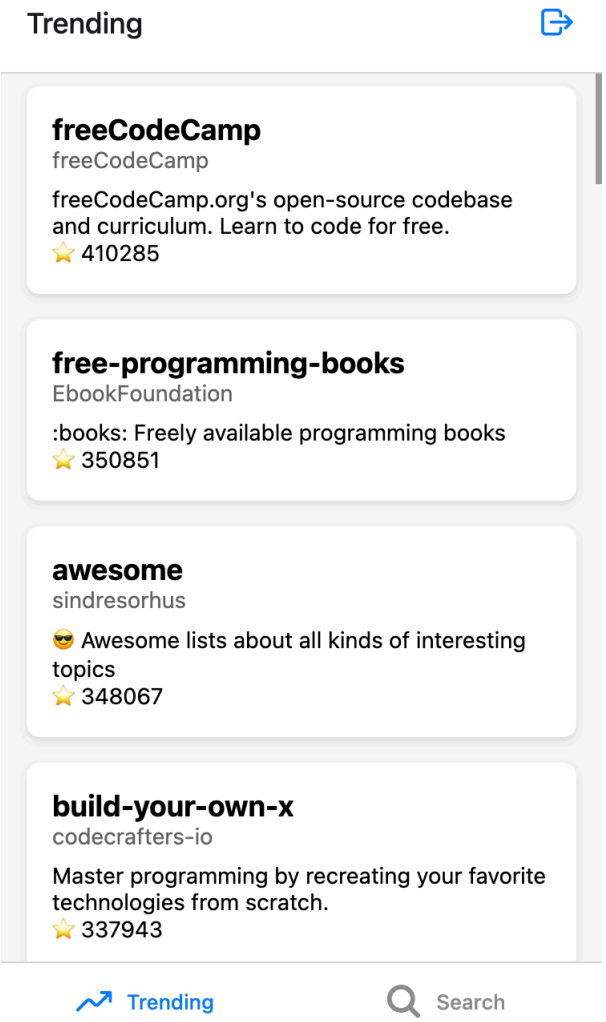
🎯 Objectifs

Dans cet exercice, vous apprendrez à :

Compétence	Description
🔐 Auth	Implémenter l'authentification avec GitHub
📡 API	Intégrer l'API GitHub et Supabase
🔄 État	Synchroniser les données entre le client et le serveur

Voici à quoi ressemblera votre application finale :





📋 Étape 0 : Lancer le projet

Commencez par naviguer vers le dossier de l'exercice et installer les dépendances :

```
cd 03-devhub
npm install
```

Puis lancez le projet :

```
npx expo start
```

> **INFO**
>
> Si vous rencontrez des problèmes de connexion, essayez de lancer le projet avec le tunnel :
>
> ```
> npx expo start --tunnel
> ```

📋 Étape 1 : Créer un projet Supabase

1.1 Installation des dépendances

Naviguez vers le dossier de l'exercice et installez les dépendances.

```
cd exercises/03-devhub
npm install
```

1.2 Créer un projet Supabase

Créez un nouveau projet sur Supabase :

1. Connectez-vous à votre compte Supabase
2. Créez un nouveau projet
3. Notez l'URL et la clé d'API de votre projet

1.2 Variables d'environnement

Naviguez vers le dossier de l'exercice et installez les dépendances. Ensuite on va copier le fichier `.env.example` et le renommer en `.env`. C'est dans ce fichier que vous mettrez votre clé d'API

Supabase.

```
cp .env.example .env
```

Ouvrez le fichier `.env` et remplacez les valeurs par vos propres valeurs.

```
EXPO_PUBLIC_SUPABASE_URL=
EXPO_PUBLIC_SUPABASE_ANON_KEY=
```

1.3 Gestion du Login et Register

Pour la gestion du login et register, nous allons utiliser le SDK de Supabase. Pour cela dans le dossier `lib` on va créer un fichier `supabase.ts` et on va y implémenter le SDK de Supabase. Dans ce dossier on retrouve nos variables d'environnement. Expo permet de faire appel à ces variables pour autant qu'elles possèdent le préfixe `EXPO_PUBLIC_`.

```
import "react-native-url-polyfill/auto";
import { createClient } from "@supabase/supabase-js";

const supabaseUrl = process.env.EXPO_PUBLIC_SUPABASE_URL;
const supabaseAnonKey = process.env.EXPO_PUBLIC_SUPABASE_ANON_KEY;

if (!supabaseUrl || !supabaseAnonKey) {
  throw new Error("Missing Supabase environment variables");
}

export const supabase = createClient(supabaseUrl, supabaseAnonKey);
```

Maintenant qu'on a notre client Supabase, on va pouvoir l'utiliser dans notre application. Nous allons créer une page de login et de register dans le dossier `app/auth`.

Les deux pages auront un formulaire et un bouton pour se connecter ou se register (similaire à l'exercice 2). la nouveauté ici est que nous allons utiliser le SDK de Supabase pour gérer l'authentification.

La fonction SignUp ressemble a ceci. Il manque la partie ou on appelle le SDK de Supabase pour faire le SignUp. Pour cela regardez la documentation de Supabase.

```
async function signUp() {
  if (!email || !password) {
    Alert.alert("Error", "Please fill in all fields");
    return;
  }
```

```
  setLoading(true);
  try {
    const { data, error } = await supabase.auth.signUp({
      email,
      password,
    });

    if (error) throw error;

    if (data.session) {
      router.replace("/(tabs)");
    } else {
      Alert.alert(
        "Success",
        "Registration successful. Please check your email.",
        [{ text: "OK", onPress: () => router.replace("/auth/login") }]
      );
    }
  } catch (error) {
    Alert.alert("Error", (error as Error).message);
  } finally {
    setLoading(false);
  }
}
```

Pour la fonction de connexion, il faut utiliser la fonction `signInWithPassword` de Supabase. Pour cela regardez la documentation de Supabase.

```
async function signIn() {
  if (!email || !password) {
    Alert.alert("Error", "Please fill in all fields");
    return;
  }

  setLoading(true);
  try {
    // TODO: Ajouter la connexion avec le SDK de Supabase

    if (error) throw error;

    router.replace("/(tabs)");
  } catch (error) {
    Alert.alert("Error", (error as Error).message);
  } finally {
    setLoading(false);
  }
}
```

1.4 Gestion du Layout

L'application aura 2 layouts. Le layout qui se trouve dans le dossier `app` est le layout racine de l'application. Il est utilisé pour l'authentification et le layout `(tabs)/_layout.tsx` sera utilisé pour les onglets (trending et recherche).

Le premier layout est le suivant. Il permet de rediriger l'utilisateur vers la page de login si il n'est pas connecté et vers la page des onglets si il est connecté. Pour cela la variable session est utilisée.

```
export default function RootLayout() {
  const [session, setSession] = useState<Session | null>(null);
  const segments = useSegments();
  const router = useRouter();

  useEffect(() => {
    //1. verifier si l'utilisateur est connecté
    supabase.auth.getSession().then(({ data: { session } }) => {
      setSession(session);
    });

    //2. écouter les changements d'état de la session
    supabase.auth.onAuthStateChange((_event, session) => {
      setSession(session);
    });
  }, []);

  useEffect(() => {
    const inAuthGroup = segments[0] === "auth";

    if (!session && !inAuthGroup) {
      router.replace("/auth/login");
    } else if (session && inAuthGroup) {
      router.replace("/(tabs)");
    }
  }, [session, segments]);

  return <Slot />;
}
```

Le deuxième layout est le suivant. Il permet de gérer les onglets (trending et recherche).

```
export default function TabsLayout() {
  const router = useRouter();

  async function handleLogout() {
    try {
      const { error } = await supabase.auth.signOut();
```

```
      if (error) throw error;
      router.replace("/auth/login");
    } catch (error) {
      Alert.alert("Error logging out", (error as Error).message);
    }
  }

  return (
    <Tabs
      screenOptions={{
        tabBarActiveTintColor: "#007AFF",
        headerRight: () => (
          <TouchableOpacity onPress={handleLogout} style={{ marginRight: 16
}}>
            <Ionicons name="log-out-outline" size={24} color="#007AFF" />
          </TouchableOpacity>
        ),
      }}
    >
      <Tabs.Screen
        name="index"
        options={{
          title: "Trending",
          tabBarIcon: ({ color, size }) => (
            <Ionicons name="trending-up" size={size} color={color} />
          ),
        }}
      />
      <Tabs.Screen
        name="search"
        options={{
          title: "Search",
          tabBarIcon: ({ color, size }) => (
            <Ionicons name="search" size={size} color={color} />
          ),
        }}
      />
    </Tabs>
  );
}
```

🔐 Étape 2 : API GitHub

Dans la page Home, on affiche la liste des repositories trending. Pour cela on utilise l'API de GitHub. Dans le dossier `lib` on peut voir le fichier `github.ts` qui contient la configuration de l'API GitHub.

Dans ce dossier on retrouve les informations que l'on recupère:

```ts
type Repository = {
  id: number;
  name: string;
  owner: {
    login: string;
  };
  description: string;
  stargazers_count: number;
  html_url: string;
};
```

Nous avons egalement deux fonctions qui permettent de récupérer les repositories trending et les repositories recherchés:

```ts
export const github = {
  // Obtenir les repos tendance (limité à 30 résultats)
  getTrendingRepos: () =>
    fetchFromGitHub<Repository>(
      "/search/repositories?q=stars:>1&sort=stars&order=desc&per_page=30"
    ),

  // Rechercher des repos
  searchRepos: (query: string) =>
    fetchFromGitHub<Repository>(
      `/search/repositories?q=${encodeURIComponent(query)}&per_page=30`
    ),
};
```

2.1 Récupérer les repositories trending

Pour récupérer les repositories trending, on utilise la fonction `getTrendingRepos` qui fait une requête à l'API de GitHub. Dans le fichier `app/(tabs)/index.tsx` on peut voir la fonction `loadTrendingRepos` qui permet de récupérer les repositories trending.

```ts
async function loadTrendingRepos() {
  try {
    // TODO: Récupérer les repositories trending
    // Mettre à jour la variable repos avec les repositories trending
  } catch (error) {
    console.error("Error:", error);
  } finally {
    setLoading(false);
  }
}
```

> **TIP**
>
> Pour récupérer les repositories trending, on utilise la fonction `getTrendingRepos` qui fait une requête à l'API de GitHub. Ensuite il faut mettre à jour le useState `repos` avec les repositories trending.

2.2 Afficher les repositories trending

Toujours dans le fichier `app/(tabs)/index.tsx` on peut voir la fonction `useRepos` qui permet de récupérer les repositories trending.

```
return (
  <SafeAreaView style={styles.container}>
    // TODO: Afficher les repositories trending avec une FlatList
  </SafeAreaView>
);
```

> **TIP**
>
> Pour afficher les repositories trending, on utilise une FlatList. Vous pouvez soit utiliser le composant `RepoCard` ou créer votre propre composant ou utiliser des variables Text et View.

🔐 Étape 3 : Recherche de repositories

Similaire à l'étape 2, on va ajouter une fonction pour rechercher des repositories. Dans le fichier `app/(tabs)/search.tsx` on peut voir la fonction `searchRepos` qui permet de rechercher des repositories.

```
export default function SearchScreen() {
  const [query, setQuery] = useState("");
  const [repos, setRepos] = useState<Repository[]>([]);
  const [loading, setLoading] = useState(false);

  async function searchRepos() {
    if (!query.trim()) return;

    setLoading(true);
    try {
      // TODO: Rechercher des repositories avec la fonction `searchRepos`
    } catch (error) {
      console.error("Error:", error);
    } finally {
      setLoading(false);
```

```
    }
  }

  return (
    <View style={styles.container}>
      <TextInput
        style={styles.input}
        value={query}
        onChangeText={setQuery}
        onSubmitEditing={searchRepos}
        placeholder="Search repositories..."
        returnKeyType="search"
      />
      {loading ? (
        <ActivityIndicator style={styles.center} size="large" />
      ) : (
        // TODO: Afficher les repositories recherchés avec une FlatList
      )}
    </View>
  );
}
```

Edit this page