📘 COURS COMPLET : LES COMPOSANTS ET LES PROPS EN REACT NATIVE

🎯 Objectifs pédagogiques

À la fin de ce cours, tu seras capable de :

- Comprendre la structure et le rôle des **composants** dans une application React Native.
- Créer des **composants personnalisés et réutilisables**.
- Utiliser les **props** pour transmettre des données et des fonctions entre composants.
- Organiser ton code pour une meilleure lisibilité et modularité.

🧩 1. Qu'est-ce qu'un composant ?

Un **composant** est une **brique de base** de toute application React ou React Native.
C'est une **fonction (ou classe)** qui renvoie une interface utilisateur (UI).

➞ Chaque écran, bouton, formulaire ou carte d'affichage peut être un composant.

◆ Exemple simple

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

export default function Bonjour() {
  return (
    <View>
      <Text>Bonjour, monde 👋</Text>
    </View>
  );
}
```

Ici :

- `Bonjour` est un composant fonctionnel.
- Il retourne un peu de JSX (`<View>` et `<Text>`).

⚙️ 2. Types de composants

◈ a. Composants fonctionnels (modernes et recommandés)

Ce sont des fonctions JavaScript qui renvoient du JSX.

```
function Header() {
  return <Text>Ceci est un en-tête</Text>;
}
```

✅ Avantages :

- Simples à lire et à maintenir.
- Compatibles avec les **Hooks** (`useState`, `useEffect`, etc.).
- Performants et modernes.

◈ b. Composants de classe (anciens)

Ils étaient utilisés avant l'arrivée des Hooks.

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

class Header extends Component {
  render() {
    return <Text>Ceci est un en-tête</Text>;
  }
}
```

❌ Aujourd'hui, on préfère les **composants fonctionnels**.

🧱 3. Structure d'un projet avec composants

Un projet React Native peut être organisé ainsi :

```
UserApp/
│
├── App.js
└── components/
    ├── UserItem.js
    ├── UserForm.js
    └── UserList.js
```

- `App.js` : le **composant principal** (parent).
- `components/` : contient les **composants enfants réutilisables**.

💡 4. Pourquoi utiliser des composants ?

- **Réutilisation** : le même bouton ou formulaire peut être utilisé plusieurs fois.
- **Lisibilité** : chaque composant a un rôle clair.
- **Maintenance** : modifier un composant le met à jour partout où il est utilisé.
- **Séparation logique** : UI et logique métier bien séparées.

🔗 5. Communication entre composants

a. Parent → Enfant : avec les props

Le parent envoie des données à l'enfant sous forme de **propriétés** (props).

```
// Parent
<User name="Tomi" age={25} />
// Enfant (User.js)
export default function User({ name, age }) {
  return <Text>{name} a {age} ans.</Text>;
}
```

b. Enfant → Parent : avec les callbacks (fonctions)

L'enfant peut exécuter une fonction transmise par le parent.

```
// Parent
<UserButton onPressAction={() => alert('Clic reçu !')} />
// Enfant
<Button title="Appuie ici" onPress={onPressAction} />
```

🧠 6. Les props en détail

Définition

Les **props** (pour *properties*) sont des **paramètres passés à un composant**.
Elles servent à personnaliser son contenu ou son comportement.

Les props :

- sont **en lecture seule** (on ne peut pas les modifier directement) ;
- peuvent être **de tout type** : chaîne, nombre, tableau, objet, fonction, etc.

Exemples de passage de props

Type de prop	Exemple d'appel	Exemple d'utilisation
Texte	`<Title text="Bienvenue" />`	`{text}`
Nombre	`<Timer seconds={10} />`	`{seconds}`

Type de prop	Exemple d'appel	Exemple d'utilisation
Objet	`<User data={{name: 'Ali', age: 25}} />`	`{data.name}`
Tableau	`<List items={['A', 'B']} />`	`{items.map(...)}`
Fonction	`<Button onClick={handlePress} />`	`onClick()`

Réception des props

Deux méthodes :

```
// 1️⃣ Déstructuration (recommandée)
function User({ name, age }) {
  return <Text>{name} - {age} ans</Text>;
}

// 2️⃣ Sans déstructuration
function User(props) {
  return <Text>{props.name} - {props.age} ans</Text>;
}
```

💬 7. Exemple concret : composant avec props

Exemple 1 : le composant `UserItem`

Objectif :

Afficher les informations d'un utilisateur (nom + email) et proposer deux boutons : **modifier** et **supprimer**.

Code :

```
// components/UserItem.js
import React from 'react';
import { View, Text, Button, StyleSheet } from 'react-native';

const UserItem = ({ user, onEdit, onDelete }) => {
  return (
    <View style={styles.item}>
      <Text style={styles.text}>{user.name} ({user.email})</Text>
      <View style={styles.actions}>
        <Button title="Modifier" onPress={onEdit} />
        <Button title="Supprimer" color="red" onPress={onDelete} />
      </View>
    </View>
  );
};

const styles = StyleSheet.create({
  item: { padding: 10, backgroundColor: '#f8f8f8', marginVertical: 5,
borderRadius: 8 },
  text: { fontSize: 16 },
```

```
  actions: { flexDirection: 'row', justifyContent: 'space-between',
marginTop: 5 },
});

export default UserItem;
```

Explications :

- `user` est **une donnée passée depuis le parent (`App.js`)** via une **propriété (prop)**.
- `onEdit` et `onDelete` sont aussi des **fonctions passées en props**, que le composant exécute lors des clics sur les boutons.

💡 → Le composant ne connaît pas la logique CRUD ; il se contente d'afficher et de déclencher des actions.

Exemple 2 : le composant `UserForm`

Objectif :

Un formulaire pour **ajouter ou modifier** un utilisateur.

Code :

```
// components/UserForm.js
import React, { useState, useEffect } from 'react';
import { View, TextInput, Button, StyleSheet } from 'react-native';

const UserForm = ({ onSubmit, editingUser }) => {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');

  // Quand on modifie un utilisateur existant, préremplir le formulaire
  useEffect(() => {
    if (editingUser) {
      setName(editingUser.name);
      setEmail(editingUser.email);
    }
  }, [editingUser]);

  const handleSubmit = () => {
    if (!name || !email) return;
    onSubmit({ id: editingUser?.id, name, email });
    setName('');
    setEmail('');
  };

  return (
    <View style={styles.container}>
      <TextInput
        placeholder="Nom"
        value={name}
        onChangeText={setName}
        style={styles.input}
```

```
      />
      <TextInput
        placeholder="Email"
        value={email}
        onChangeText={setEmail}
        style={styles.input}
      />
      <Button
        title={editingUser ? 'Mettre à jour' : 'Ajouter'}
        onPress={handleSubmit}
      />
    </View>
  );
};

const styles = StyleSheet.create({
  container: { marginBottom: 15 },
  input: { borderWidth: 1, borderColor: '#ccc', padding: 8, marginVertical:
5, borderRadius: 5 },
});

export default UserForm;
```

Explications :

- Ce composant possède son propre **état interne (useState)** pour stocker les valeurs des champs.
- Il **reçoit une fonction `onSubmit` en prop**, qui sera appelée par le parent (`App.js`) avec les données saisies.
- `editingUser` permet de **préremplir les champs** si on est en mode édition.

Comment les appeler dans `App.js`

C'est dans le composant parent que tu **appelles tes composants enfants** et que tu **passes les props** :

```
import React, { useState } from 'react';
import { View, Text, FlatList, StyleSheet } from 'react-native';
import UserForm from './components/UserForm';
import UserItem from './components/UserItem';

const App = () => {
  const [users, setUsers] = useState([]);
  const [editingUser, setEditingUser] = useState(null);

  const addUser = (user) => setUsers([...users, { ...user, id:
Date.now().toString() }]);
  const updateUser = (updatedUser) => {
    setUsers(users.map(u => (u.id === updatedUser.id ? updatedUser : u)));
    setEditingUser(null);
  };
  const deleteUser = (id) => setUsers(users.filter(u => u.id !== id));

  return (
    <View style={styles.container}>
```

```
      <Text style={styles.title}>Gestion des utilisateurs</Text>

      {/* Formulaire d'ajout ou d'édition */}
      <UserForm onSubmit={editingUser ? updateUser : addUser}
editingUser={editingUser} />

      {/* Liste des utilisateurs */}
      <FlatList
        data={users}
        keyExtractor={(item) => item.id}
        renderItem={({ item }) => (
          <UserItem
            user={item}
            onEdit={() => setEditingUser(item)}
            onDelete={() => deleteUser(item.id)}
          />
        )}
      />
    </View>
  );
};
```

🧰 8. Validation et valeurs par défaut

Tu peux valider les props avec **PropTypes** :

```
npm install prop-types
import PropTypes from 'prop-types';

function UserItem({ user }) {
  return <Text>{user.name}</Text>;
}

UserItem.propTypes = {
  user: PropTypes.shape({
    name: PropTypes.string.isRequired,
    email: PropTypes.string,
  }),
};

UserItem.defaultProps = {
  user: { name: 'Inconnu', email: 'Non renseigné' },
};
```

✅ Cela permet d'éviter les erreurs de type et de définir des valeurs par défaut.

⚖️ 9. Différence entre state et props

Aspect	State	Props
Qui le définit ?	Le composant lui-même	Le composant parent
Peut être modifié ?	Oui (avec `setState` ou `useState`)	Non (lecture seule)
Objectif	Stocker et gérer les données internes	Recevoir des données externes
Exemple	Champ d'un formulaire	Donnée passée à un enfant

🚀 10. Atelier pratique – Mise en œuvre

🎯 Objectif

Créer une mini application affichant une **liste d'utilisateurs** à l'aide de composants et de props.

💡 Étapes

1. Créer un dossier `components/`
2. Coder :
   - `UserForm` → formulaire d'ajout
   - `UserItem` → affichage d'un utilisateur
   - `UserList` → liste avec recherche
3. Passer les données (`users`) et fonctions (`onEdit`, `onDelete`) via les **props**.
4. Tester la communication Parent ↔ Enfant.
5. Bonus : afficher un message si la liste est vide.

✅ 11. Bonnes pratiques

- Utilise toujours la **déstructuration** dans les composants.
- Ne modifie jamais une prop directement.
- Donne des **noms explicites** à tes props (`onSubmitForm`, `userData`...).
- Garde tes composants **simples et isolés** (1 responsabilité = 1 composant).
- Documente les props importantes (via commentaires ou PropTypes).

📜 12. Résumé du cours

Concept	Description courte
Composant	Bloc réutilisable qui représente une partie de l'interface.
Props	Paramètres passés à un composant (données ou fonctions).
Parent → Enfant	Communication descendante via les props.
Enfant → Parent	Communication ascendante via les callbacks.

Concept	Description courte
State vs Props	State = interne et modifiable ; Props = externe et fixe.