Atelier Développement Web et Mobile Moderne

Construire des interfaces réactives de A à Z avec Vite

Présenté par Dr. Abdelweheb Gueddes

Nuit de l'info 2025

26-11-2025
ISITcom, Sousse

Programme de l'Atelier (2 Heures)

1. **Partie 1 : Backend Express via Sails.js (15 minutes)**
   - Génération d'une API REST complète en 3 commandes.
2. **Partie 2 : Le Web avec React & Vite (60 minutes)**
   - Évolution du code étape par étape : UI statique, état, formulaires.
   - Composants réutilisables, Props.
   - Connexion à notre API : Lire et Écrire des données.
3. **Partie 3 : Le Mobile avec React Native (40 minutes)**
   - L'écosystème Expo et les composants natifs.
   - Réutiliser 90% de notre code pour une app mobile.
4. **Conclusion, Défi et Q&R (5 minutes)**

Architecture Générale

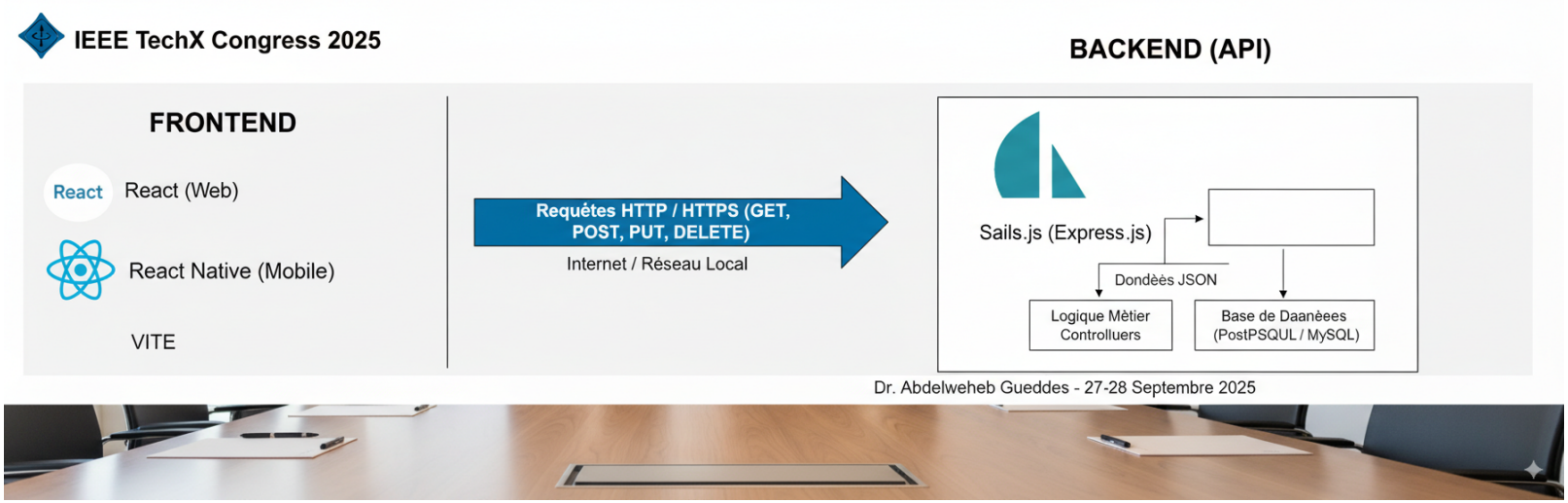

Figure – Architecture Générale : Communication Frontend-Backend

Générer une API REST en 3 Commandes

3. Lancer le serveur

Objectif

Créer une API fonctionnelle pour gérer des "tâches" en moins de 5 minutes.

```
npm install -g sails
sails new task-api --no-frontend
cd task-api
```

Générer une API REST en 3 Commandes

3. Lancer le serveur

Objectif

Créer une API fonctionnelle pour gérer des "tâches" en moins de 5 minutes.

```
npm install -g sails
sails new task-api --no-frontend
cd task-api
```

```
sails generate api task title:string isCompleted:boolean
```

Générer une API REST en 3 Commandes

3. Lancer le serveur

Objectif

Créer une API fonctionnelle pour gérer des "tâches" en moins de 5 minutes.

```
npm install -g sails
sails new task-api --no-frontend
cd task-api
```

```
sails generate api task title:string isCompleted:boolean
```

```
sails lift
```

Notre API est prête sur `http://localhost:1337`

Les routes `GET`, `POST`, `PATCH`, `DELETE` pour `/task` sont automatiquement créées.

Étape 1 : Projet et UI Statique

2. Notre premier composant : src/App.jsx

```
npm create vite@latest ma-weblist -- --template react
cd ma-weblist
npm install
npm run dev
```

Étape 1 : Projet et UI Statique

2. Notre premier composant : src/App.jsx

```
npm create vite@latest ma-weblist -- --template react
cd ma-weblist
npm install
npm run dev
```

On commence avec des données "en dur" pour construire l'interface.

```jsx
// src/App.jsx
import './App.css';
function App() {
  const tasks = [
    { id: 1, title: 'Apprendre React', isCompleted: true },
    { id: 2, title: 'Boire un café', isCompleted: false },
  ];
  return (
    <div className="App-container">
      <h1>Liste des Tâches</h1>
      <ul className="task-list">
        {tasks.map(task => (
          <li key={task.id}>{task.title}</li>
        ))}      </ul>    </div>  ); }
export default App;
```

Étape 2 : Introduction de l'État ('useState')

Modification de src/App.jsx

Pour que les données puissent changer, on les place dans l'état.

```
1  import React, { useState } from 'react'; // 1. Importer useState
2  import './App.css';
3  function App() {
4    // 2. Remplacer la constante par un état
5    const [tasks, setTasks] = useState([
6      { id: 1, title: 'Apprendre React', isCompleted: true },
7      { id: 2, title: 'Boire un café', isCompleted: false },
8    ]);
9    return (
10     <div className="App-container">
11       <h1>Liste des Tâches</h1>
12       <ul className="task-list">
13         {tasks.map(task => (
14           <li key={task.id}>{task.title}</li>
15         ))}
16       </ul>
17     </div>
18   );
19 }
20 export default App;
```

Étape 3 : Gestion du Formulaire

JSX dans App.jsx

On ajoute un état pour le champ de saisie et une fonction pour gérer l'ajout.

```jsx
import React, { useState } from 'react';
import './App.css';
function App() {
  const [tasks, setTasks] = useState([]);
  // 1. État pour le texte de l'input
  const [newTodoText, setNewTodoText] = useState('');
  // 2. Fonction pour ajouter une tâche
  const handleAddTask = (e) => {
    e.preventDefault(); // Empêche le refresh
    if (!newTodoText.trim()) return;
    const newTask = {
      id: Date.now(), // ID temporaire
      title: newTodoText,
      isCompleted: false,
    };
    setTasks([...tasks, newTask]);
    setNewTodoText(''); // Vider le champ
  };
  // ... return JSX ...
}
```

```jsx
// ...
return (
  <div className="App-container">
    <h1>Liste des Tâches</h1>
    {/* 3. Formulaire pour ajouter */}
    <form onSubmit={handleAddTask} className="task-form">
      <input
        type="text"
        className="task-input"
        value={newTodoText}
        onChange={(e) => setNewTodoText(e.target.value)}
      />
      <button type="submit">Ajouter</button>
    </form>
    <ul className="task-list">
      {tasks.map(task => (
        <li key={task.id}>{task.title}</li>
      ))}
    </ul>  </div>   );
//
```

Étape 4 : Connexion à l'API (GET)

Le Hook 'useEffect' dans src/App.jsx

On remplace les données en dur par un appel à notre API Sails.js.

```
1  import React, { useState, useEffect } from 'react'; // 1. Importer useEffect
2  import './App.css';
3  const API_URL = 'http://localhost:1337/task';
4  function App() {
5    const [tasks, setTasks] = useState([]);
6    const [newTodoText, setNewTodoText] = useState('');
7
8    // 2. Ce code s'exécute une seule fois au chargement
9    useEffect(() => {
10     fetch(API_URL)
11       .then(res => res.json())
12       .then(data => setTasks(data))
13       .catch(err => console.error("Erreur de chargement:", err));
14   }, []); // Le tableau vide [] est crucial
15
16   const handleAddTask = (e) => { /* ... inchangé pour l'instant ... */ };
17
18   return ( /* ... JSX inchangé ... */ );
19 }
20 export default App;
```

Étape 5 : Envoyer des données à l'API (POST)

Mise à jour de 'handleAddTask' dans App.jsx

On modifie 'handleAddTask' pour qu'elle envoie la nouvelle tâche au serveur.

```
1  // ...
2  const handleAddTask = (e) => {
3    e.preventDefault();
4    if (!newTodoText.trim()) return;
5
6    const newTask = {
7      title: newTodoText,
8      isCompleted: false,
9    };
10
11   // Envoi de la requête POST
12   fetch(API_URL, {
13     method: 'POST',
14     headers: { 'Content-Type': 'application/json' },
15     body: JSON.stringify(newTask),
16   })
17     .then(res => res.json())
18     .then(addedTask => {
19       // On met à jour l'UI avec la réponse du serveur
20       setTasks([...tasks, addedTask]);
21       setNewTodoText('');
22     });
23 };
24 // ...
```

Prérequis : Votre Téléphone et Expo Go

Avant de coder, préparez votre téléphone !

Cette étape est essentielle pour voir votre application en direct.

1. Prenez votre smartphone personnel (iOS ou Android).

Prérequis : Votre Téléphone et Expo Go

Avant de coder, préparez votre téléphone !

Cette étape est essentielle pour voir votre application en direct.

1. Prenez votre smartphone personnel (iOS ou Android).
2. Assurez-vous qu'il est connecté au **même réseau Wi-Fi** que votre ordinateur.

Prérequis : Votre Téléphone et Expo Go

Avant de coder, préparez votre téléphone !

Cette étape est essentielle pour voir votre application en direct.

1. Prenez votre smartphone personnel (iOS ou Android).
2. Assurez-vous qu'il est connecté au **même réseau Wi-Fi** que votre ordinateur.
3. Téléchargez l'application gratuite « **Expo Go** » depuis l'App Store ou le Google Play Store.

C'est cette application qui exécutera notre code en temps réel.

Mobile : Étape 1 - Installation et Création du Projet

3. Naviguer dans le dossier du projet

Cet outil, comme Vite pour le web, gère la création et le lancement du projet.

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

Mobile : Étape 1 - Installation et Création du Projet

3. Naviguer dans le dossier du projet

Cet outil, comme Vite pour le web, gère la création et le lancement du projet.

```
1 npm install -g expo-cli
```

```
1 expo init ma-mobilelist
```

Mobile : Étape 1 - Installation et Création du Projet

3. Naviguer dans le dossier du projet

Cet outil, comme Vite pour le web, gère la création et le lancement du projet.

```
1 npm install -g expo-cli
```

```
1 expo init ma-mobilelist
```

Que choisir ?

Quand l'outil vous le demande, sélectionnez le template **'blank'** (minimal).

Mobile : Étape 1 - Installation et Création du Projet

3. Naviguer dans le dossier du projet

Cet outil, comme Vite pour le web, gère la création et le lancement du projet.

```
1                    npm install -g expo-cli
```

```
1                    expo init ma-mobilelist
```

Que choisir ?

Quand l'outil vous le demande, sélectionnez le template **'blank'** (minimal).

```
1                    cd ma-mobilelist
```

Du Web au Mobile : Le Code Complet

Voici à quoi ressemble App.js sur mobile

On réutilise toute la logique (useState, useEffect, fetch) et on adapte seulement la "Vue" (le JSX) avec des composants natifs.

```
1  import React, { useState, useEffect } from 'react';
2  import { StyleSheet, Text, View, TextInput, Button, FlatList, SafeAreaView,
   ↪  TouchableOpacity, Platform } from 'react-native';
3
4  const API_URL = 'http://VOTRE_IP_LOCALE:1337/task';
5
6  const Banner = ({ text }) => (
7    <View style={styles.bannerContainer}><Text
   ↪  style={styles.bannerText}>{text}</Text></View>
8  );
9
10 export default function App() {
11   // --- LA LOGIQUE EST 100% IDENTIQUE ---
12   const [tasks, setTasks] = useState([]);
13   const [newTodoText, setNewTodoText] = useState('');
14   useEffect(() => { /* ... fetch GET ... */ }, []);
15   const handleAddTask = () => { /* ... fetch POST ... */ };
16
17   // --- LA VUE EST DIFFÉRENTE ---
18 }
```

Du Web au Mobile : Le Code Complet

Voici à quoi ressemble App.js sur mobile

On réutilise toute la logique (useState, useEffect, fetch) et on adapte la "Vue" (le JSX) => composants natifs.

```
1  // --- LA VUE EST DIFFÉRENTE ---
2  return (
3    <SafeAreaView style={styles.container}>
4      <Banner text="Atelier IEEE TechX 2025 !" />
5      <View style={styles.content}>
6        <Text style={styles.title}>Tâches Mobiles</Text>
7        <View style={styles.form}>
8          <TextInput style={styles.input} value={newTodoText}
           ↪  onChangeText={setNewTodoText} />
9          <Button title="Ajouter" onPress={handleAddTask} />
10       </View>
11       <FlatList
12         data={tasks}
13         keyExtractor={item => item.id.toString()}
14         renderItem={({ item }) => (
15           <TouchableOpacity>
16             <View style={styles.taskItem}><Text>{item.title}</Text></View>
17           </TouchableOpacity>
18         )}
19       />      </View>    </SafeAreaView>  ); }
20       const styles = StyleSheet.create({ /* ... tous les styles ... */ });
```

Mobile : Étape 2 - Lancement et Connexion

1. Démarrer le serveur de développement

Cette commande va lancer un serveur sur votre ordinateur et ouvrir une page dans votre navigateur.

```
npm start
```

Mobile : Étape 2 - Lancement et Connexion

1. Démarrer le serveur de développement

Cette commande va lancer un serveur sur votre ordinateur et ouvrir une page dans votre navigateur.

```
npm start
```

Le terminal va maintenant afficher un QR Code sur la gauche.

Action : Connectez votre téléphone !

1. Ouvrez l'application **Expo Go** que vous venez de télécharger.
2. Allez dans l'onglet "Scan" (ou appuyez sur "Scan QR Code").
3. Visez le QR Code affiché dans votre terminal avec l'appareil photo.

Bonus : Le Défi (Web)

Pistes de solution

Objectif : Rendre les tâches interactives sur le Web

Ajouter une requête `PATCH` au clic pour changer l'état 'isCompleted' et appliquer un style conditionnel.

```
1  // La fonction de mise à jour
2  const handleToggleStatus = (taskToUpdate) => {
3    const newStatus = !taskToUpdate.isCompleted;
4    fetch(`${API_URL}/${taskToUpdate.id}`, {
5      method: 'PATCH',
6      headers: { 'Content-Type': 'application/json' },
7      body: JSON.stringify({ isCompleted: newStatus }),
8    }).then(() => { /* Mettre à jour l'état local */ });
9  };
10
11 // Le JSX avec style et événement
12 <li className={task.isCompleted ? 'completed' : ''} onClick={() =>
   ↪  handleToggleStatus(task)}>
13   {task.title}
14 </li>
15
16 // Le CSS
17 .task-list li.completed { text-decoration: line-through; color: grey; }
```

Bonus : Le Défi (Mobile)

Pistes de solution

Objectif : Transposer l'interactivité sur Mobile

Même logique, mais avec les outils de React Native : '<TouchableOpacity>' et les styles en tableau.

```
1  // Utiliser TouchableOpacity
2  <TouchableOpacity onPress={() => handleToggleStatus(item)}>
3    <View style={[styles.taskItem, item.isCompleted &&
   ↪  styles.completedItem]}>
4      <Text style={item.isCompleted ? styles.completedText : {}}>
5        {item.title}
6      </Text>
7    </View>
8  </TouchableOpacity>
9
10 // Les styles dans StyleSheet
11 completedItem: { backgroundColor: '#d1e7dd' },
12 completedText: { textDecorationLine: 'line-through', color:
   ↪  'grey' }
```

Ressources & Fin

Pour Aller Plus Loin

- Documentation officielle React : react.dev
- Documentation officielle React Native : reactnative.dev
- React Native Directory (librairies et outils) : reactnativedirectory.com
- Tutoriels interactifs : react-tutorial.app
- Cours vidéo (gratuit) : FreeCodeCamp - React

Merci ! Questions & Réponses