14/11/2024

Étude Comparative des Solutions pour la Gestion des Tâches

Objectifs

1. *Comparer différentes approches pour la création d'une application de gestion des tâches en utilisant JavaScript et React.*
2. *Illustrer comment les diagrammes UML peuvent être appliqués à chaque solution pour la modélisation et l'analyse.*
3. *Montrer l'évolution de l'architecture d'une application, de JavaScript simple vers une application modulaire avec React.*

Mahdi KELLOUCH
ISMO TETOUAN

Table des matières

1. Introduction

La création d'applications web dynamiques et modulaires est essentielle pour répondre aux besoins modernes des utilisateurs. Les technologies comme JavaScript et React permettent de concevoir des applications réactives, modulaires et maintenables. Ce document explore les différentes façons d'implémenter une application de **gestion de tâches** en utilisant JavaScript et React.

Nous proposons une étude comparative en présentant quatre solutions, allant de la plus simple (JavaScript basique) à une application modulaire en React, pour mieux comprendre les évolutions possibles et l'importance de la structuration du code.

2. Présentation de l'Étude de Cas

L'étude de cas consiste en une application de **gestion des tâches**, comprenant les fonctionnalités suivantes :

- **Ajouter une tâche**
- **Supprimer une tâche**
- **Marquer une tâche comme en cours**
- **Marquer une tâche comme terminée**

L'application doit permettre aux utilisateurs de gérer des tâches avec différents états et de suivre leur progression. Chaque solution sera accompagnée des **diagrammes UML nécessaires** pour la modélisation de l'application.

3. Solutions de l'Application

Solution 1 : Application en JavaScript à Un Seul Niveau

Description

La première solution est une application réalisée en **HTML5, CSS3 et JavaScript** sous une seule structure, avec tout le code JavaScript dans un fichier unique. Elle permet d'illustrer les concepts de base avec JavaScript sans structuration avancée.

Fonctionnalités Implémentées

- Ajout de tâche
- Suppression de tâche
- Changement d'état de la tâche (en cours, terminée)

Diagrammes UML Requis

1. **Diagramme de Séquence Système** : Décrit les interactions de l'utilisateur avec l'application pour chaque action.
2. **Diagramme d'Activité** : Décompose chaque fonctionnalité en étapes simples.
3. **Diagramme d'État** (simple) : Présente les transitions d'état d'une tâche.

Détails de l'Implémentation

L'application est structurée avec des **fonctions globales** pour chaque fonctionnalité (ajouter, supprimer, mettre à jour), et l'état les tâches sont stockées dans un tableau global.

Structure du Code JavaScript

```
let tasks = [];

function addTask(taskName) {
  const task = { name: taskName, status: "à faire" };
  tasks.push(task);
  renderTasks();
}

function deleteTask(index) {
  tasks.splice(index, 1);
  renderTasks();
}

function updateTaskStatus(index, status) {
  tasks[index].status = status;
  renderTasks();
}

function renderTasks() {
  const taskList = document.getElementById("taskList");
  taskList.innerHTML = "";
  tasks.forEach((task, index) => {
    const taskItem = document.createElement("li");
    taskItem.innerHTML = `${task.name} - ${task.status}
      <button onclick="updateTaskStatus(${index}, 'en cours')">En cours</button>
      <button onclick="updateTaskStatus(${index}, 'terminée')">Terminée</button>
      <button onclick="deleteTask(${index})">Supprimer</button>`;
    taskList.appendChild(taskItem);
  });
}
```

Solution 2 : Application JavaScript avec Classes Métier

Description

La seconde solution utilise **JavaScript avec des classes** pour mieux structurer le code. Les classes Task et TaskManager sont introduites pour représenter les tâches et gérer la logique de l'application.

Fonctionnalités Implémentées

1. **Classe Task** : Représente une tâche avec un nom et un état.
2. **Classe TaskManager** : Gère la liste des tâches avec les méthodes d'ajout, de suppression, et de mise à jour.

Diagrammes UML Requis

1. **Diagramme de Classe** : Définit les attributs et méthodes de Task et TaskManager.
2. **Diagramme de Séquence Système** : Montre les interactions de l'utilisateur avec les classes pour chaque action.

3. **Diagramme d'Activité** : Décompose les actions en étapes liées aux méthodes de chaque classe.
4. **Diagramme d'État** : Décrit les états de chaque tâche et les transitions.

Détails de l'Implémentation

L'implémentation est basée sur des **classes JavaScript**, avec Task pour chaque tâche et TaskManager pour gérer la liste. Cela permet une meilleure organisation du code et une maintenance facilitée.

Structure des Classes JavaScript

```
class Task {
  constructor(name) {
    this.name = name;
    this.status = "à faire";
  }

  setStatus(status) {
    this.status = status;
  }
}

class TaskManager {
  constructor() {
    this.tasks = [];
  }

  addTask(taskName) {
    const task = new Task(taskName);
    this.tasks.push(task);
    this.renderTasks();
  }

  deleteTask(index) {
    this.tasks.splice(index, 1);
    this.renderTasks();
  }

  updateTaskStatus(index, status) {
    this.tasks[index].setStatus(status);
    this.renderTasks();
  }

  renderTasks() {
    const taskList = document.getElementById("taskList");
    taskList.innerHTML = "";
    this.tasks.forEach((task, index) => {
      const taskItem = document.createElement("li");
      taskItem.innerHTML = `${task.name} - ${task.status}
        <button onclick="taskManager.updateTaskStatus(${index}, 'en cours')">En
cours</button>
        <button onclick="taskManager.updateTaskStatus(${index},
'terminée')">Terminée</button>
```

```
        <button onclick="taskManager.deleteTask(${index})">Supprimer</button>`;
      taskList.appendChild(taskItem);
    });
  }
}

const taskManager = new TaskManager();

// Écouteur pour le formulaire d'ajout de tâche
document.getElementById("taskForm").addEventListener("submit", (event) => {
  event.preventDefault();
  const taskName = document.getElementById("taskName").value;
  taskManager.addTask(taskName);
  document.getElementById("taskForm").reset();
});
```

Solution 3 : Application Décomposée en Composants JavaScript

Description

La troisième solution introduit une **structure modulaire** en décomposant l'application en plusieurs composants JavaScript, similaire à une application React, mais sans l'utilisation de React.

Structure

1. index.js : Fichier d'entrée.
2. App.js : Composant principal.
3. TaskManager.js : Gestionnaire des tâches.
4. TaskForm.js : Formulaire pour l'ajout de tâche.
5. TaskList.js : Affichage de la liste des tâches.

Diagrammes UML Requis

1. **Diagramme de Classe** : Décrit chaque composant avec ses attributs et méthodes.
2. **Diagramme de Séquence Détaillé** : Décompose les interactions entre composants.
3. **Diagramme d'État** : Illustre les transitions d'état de chaque tâche en fonction des actions.
4. **Diagramme d'Activité** : Décompose les étapes de chaque fonctionnalité par composant.

Détails de l'Implémentation

Cette solution utilise **plusieurs classes JavaScript** pour diviser les responsabilités en composants. Chaque composant gère son état et ses méthodes, avec un flux d'interaction plus clair entre eux.

Exemple de Code

```
// Task.js
class Task {
  constructor(name) {
    this.name = name;
    this.status = 'à faire';
```

```
  }
}

// TaskManager.js
class TaskManager {
  constructor() {
    this.tasks = [];
  }
  // ... méthodes similaires à la solution 2
}

// TaskForm.js
class TaskForm {
  constructor(onAddTask) {
    this.onAddTask = onAddTask;
  }
  // Méthode render() pour afficher le formulaire
}

// TaskList.js
class TaskList {
  constructor(tasks, onDeleteTask, onUpdateStatus) {
    this.tasks = tasks;
    this.onDeleteTask = onDeleteTask;
    this.onUpdateStatus = onUpdateStatus;
  }
  // Méthode render() pour afficher les tâches
}
```

Solution 4 : Application en React avec Composants

Description

La dernière solution est réalisée en **React**, avec des composants pour chaque partie de l'application. Cette structure est la plus modulaire et la plus proche des standards modernes.

Composants

1. App : Composant principal qui gère l'état global.
2. TaskForm : Composant pour ajouter une tâche.
3. TaskList : Composant pour afficher la liste.
4. Task : Composant pour chaque tâche individuelle.

Diagrammes UML Requis

1. **Diagramme de Classe** : Décrit les composants React avec props et state.
2. **Diagramme de Séquence Détaillé** : Montre le flux de données et d'interactions entre les composants.
3. **Diagramme d'État** : Géré par setState dans chaque composant.
4. **Diagramme d'Activité** : Spécifie les actions de chaque composant de manière détaillée.

Détails de l'Implémentation

Avec React, l'application est organisée en composants fonctionnels ou classes, chacun avec son propre state. React permet une **mise à jour efficace** de l'interface via le **Virtual DOM**, ce qui rend l'application plus performante et plus facile à maintenir.

Exemple de Code

```
// App.js
import React, { useState } from 'react';
import TaskForm from './TaskForm';
import TaskList from './TaskList';

function App() {
  const [tasks, setTasks] = useState([]);
  // Fonctionnalités pour ajout, suppression et mise à jour de tâches
}
export default App;
```

4. Analyse Comparative des Solutions

Comparaison des Fonctionnalités

Chaque solution implémente les fonctionnalités de base mais avec un niveau de complexité et de modularité croissant.

Solution	Modèle de Gestion	Niveau de Modularité	Facilité de Maintenance
JavaScript Simple	Fonctions globales	Faible	Limitée
JavaScript avec Classes	Classes métier	Moyen	Modéré
JavaScript en Composants	Composants classes	Élevé	Bonne
React	Composants React	Très élevé	Excellente

Comparaison des Diagrammes UML

Les solutions avec plus de modularité nécessitent des diagrammes plus détaillés.

Solution	Diagrammes Requis
JavaScript Simple	Séquence système, activité simple, état simple
JavaScript avec Classes	Séquence système, classes, activité, état
JavaScript en Composants	Séquence détaillée, classes, activité, état détaillé
React	Séquence détaillée, classes, activité, état avec state

5. Conclusion et Recommandations

Cette étude a permis d'analyser et de comparer les différentes solutions pour une application de gestion des tâches, en passant de JavaScript simple à React. Chaque solution a des avantages et des inconvénients :

- **JavaScript Simple** : Convient pour des applications de petite taille.
- **JavaScript avec Classes** : Offre une meilleure organisation du code.

- **JavaScript en Composants** : Approche modulaire proche de React, facilite l'évolution.
- **React** : Solution optimale pour des applications dynamiques, réactives et de grande envergure.

Nous recommandons d'utiliser **React** pour les applications nécessitant une forte modularité et des interactions complexes, tandis que JavaScript simple convient aux applications légères et moins complexes.

6. Annexes : Diagrammes UML

- **Diagrammes de Séquence**
- **Diagrammes de Classe**
- **Diagrammes d'État et d'Activité**