Développez un tableau de bord d'analytics avec React

Parcours Développeur d’application Javascript React

Hye-jin Cho-Drugeon

Ce que vous allez apprendre

•**Objectif du projet:**

Développer une page de profil utilisateur dynamique pour une plateforme d'analyse sportive en utilisant React.

Composants clés:

- **Composants React:** Construire des éléments UI réutilisables pour structurer la page.
- **Gestion d'état:** Gérer les changements de données au sein des composants à l'aide de l'état React.
- **Récupération de données:** Utiliser Fetch ou Axios pour récupérer des données depuis un backend Node.js.
- **Visualisation de données:** Créer des graphiques et diagrammes interactifs avec D3 ou Recharts.
- **Style CSS:** Appliquer des styles CSS aux composants pour un design visuellement attrayant.
- **Routage:** Implémenter la navigation entre différentes sections de l'application.

Concepts clés:

•**JSX:** Écrire des structures HTML-like au sein de JavaScript.

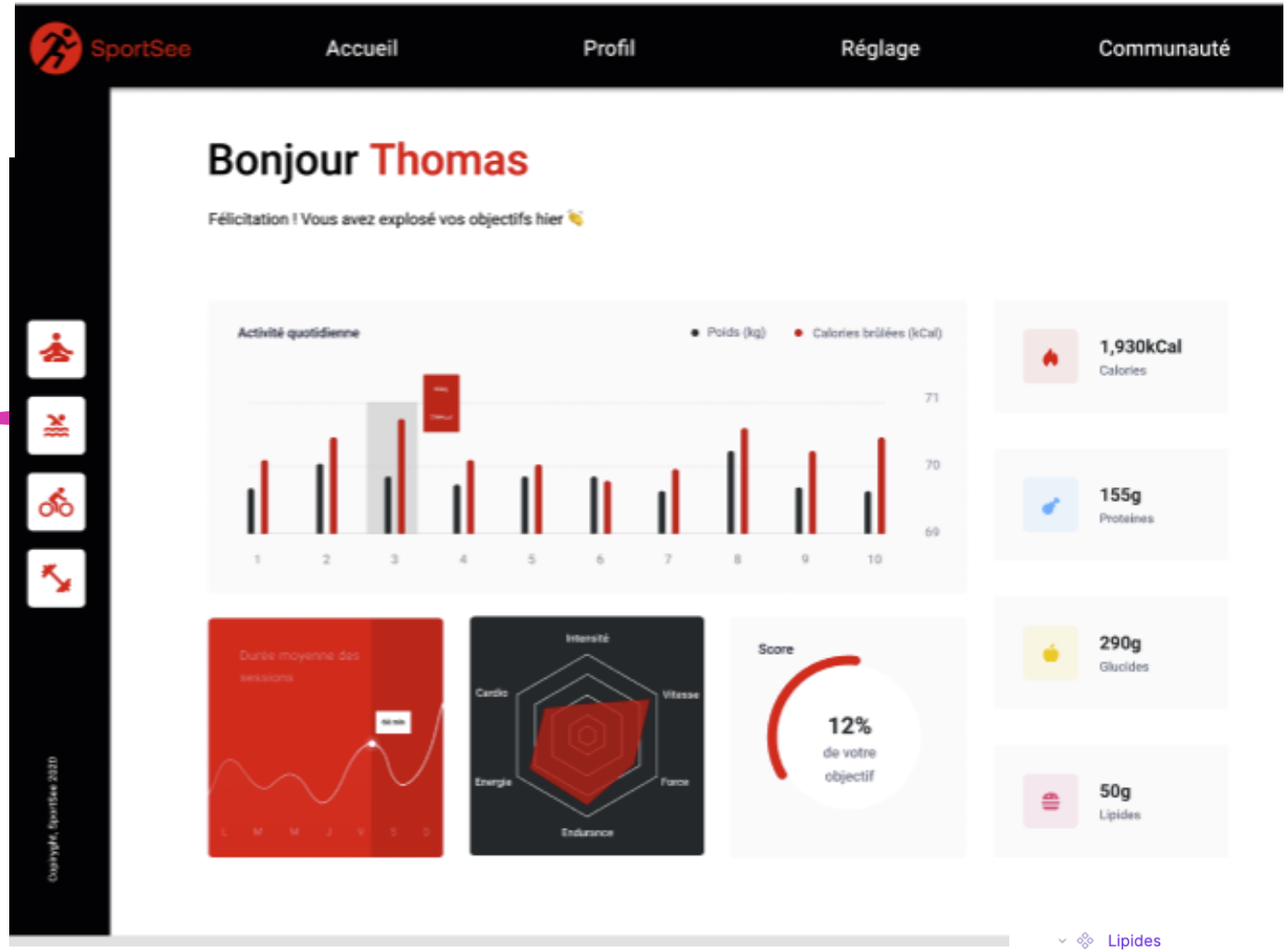
•**Props:** Transmettre des données des composants parents aux composants enfants.

•**Cycle de vie des composants:** Comprendre comment les composants se montent, se mettent à jour et se démontent.

•**Hooks:** Utiliser useState et useEffect pour la gestion d'état et les effets secondaires.

SportSee

- Figma
- Width 1440px,
- Height 1024px



- Nav
  - .logo
  - .nav-t
  - Copiryght, SportSee 2020
  - .nav-l
    - .icon
    - .icon
    - .icon
    - .icon
  - BG
  - BG

- Dashboard
  - Header
    - Félicitation ! Vous avez e
    - Bonjour Thomas
  - Radar
  - KPI
  - Objectifs
  - poids
    - .tooltips
    - Body weight
    - legend
    - Y
    - bars
    - X
    - graph-background
    - background

- Lipides
  - 50 Avg fat
  - fat-icon
  - background
- Glucides
- Proteines
- Calories

1. Récupération des données auprès de l'API

Hook React personnalisé: simplifie, données depuis une API aux composants

useState: Permet de gérer l'état interne de vos composants

1. data: Stocke les données récupérées.

2. **loading:** Indique si les données sont toujours en cours de récupération.

3. **error:** Stocke toute erreur rencontrée lors de la récupération.

useEffect: Fournit un moyen de gérer les effets secondaires, tels que la récupération de données, l'abonnement à des événements ou l'interaction avec le DOM.

1. La fonction **fetchData** est appelée au sein du hook useEffect.

2. **fetchData** est une fonction asynchrone utilisant **async/await.**

3. **await fetch(...):** Attend la réponse de l'API.

4. **await response.json():** Attend que les données de la réponse soient analysées au format JSON.

5. Le bloc **try...catch** gère les erreurs potentielles lors de la récupération.

6. Le bloc finally garantit que loading est mis à false indépendamment du succès ou de l'échec.

useFetch: Hook React personnalisé

```
src > hooks > JS useFetch.js > [@] useFetch > useEffect() callback > [@] fetchData
45
46  const useFetch = (userId, type = "") => {
47    const [data, setData] = useState(null);
48    const [isLoading, setIsLoading] = useState(true);
49
50    useEffect(() => {
51      const fetchData = async () => {
52        try {
53          setIsLoading(true);
54          const response = await fetch(`${API_BASE_URL}${userId}/${type}`);
55
56          if (!response.ok) {
57            throw new Error("API request failed");
58          }
59
60          const responseData = await response.json();
61
62          if (responseData.error) {
63            throw new Error(responseData.error);
64          }
65          setData(responseData);
66        } catch (err) {
67          console.warn(
68            `Failed to fetch from API: ${err.message}. Using mock data instead.`
69          );
70          setData(getMockData(userId, type));
71        } finally {
72          setIsLoading(false);
73        }
74      };
75
76      fetchData();
77    }, [userId, type]);
```

useFetch.js

2. Création et affichage des graphiques

- Recharts est une bibliothèque de graphiques React qui permet de créer facilement des visualisations de données interactives et personnalisables, telles que des graphiques en courbes, des diagrammes en barres, des graphiques en secteurs et bien plus encore.
- Prenons l'exemple du graphique d'activité dans `ActivityChart.jsx`

```jsx
*/
const ActivityChart = ({ userId }) => {
  const { data, isLoading } = useFetch(userId, 'activity');

  if (isLoading) {
    return <div>Loading...</div>;
  }
```

Nous avons également implémenté d'autres types de graphiques :

- Un graphique radar pour les performances (`PerformanceChart.jsx`)

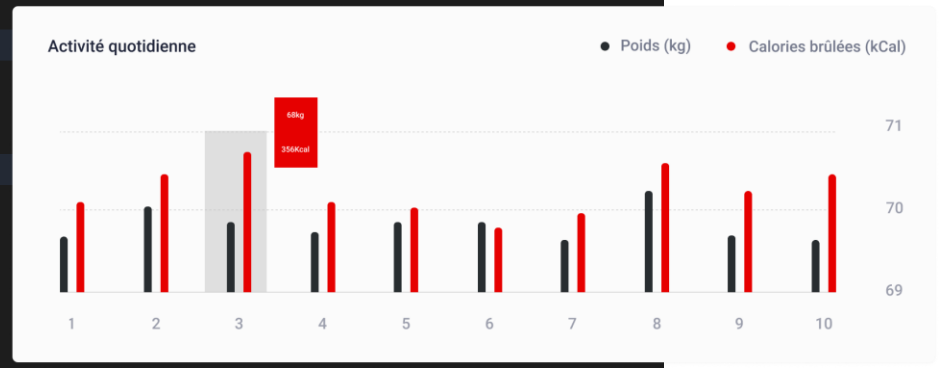
- Un graphique linéaire pour les sessions (`SessionsChart.jsx`)

- Un graphique circulaire pour le score global (`ScoreChart.jsx`)

Chaque graphique est adapté pour représenter au mieux les données spécifiques.

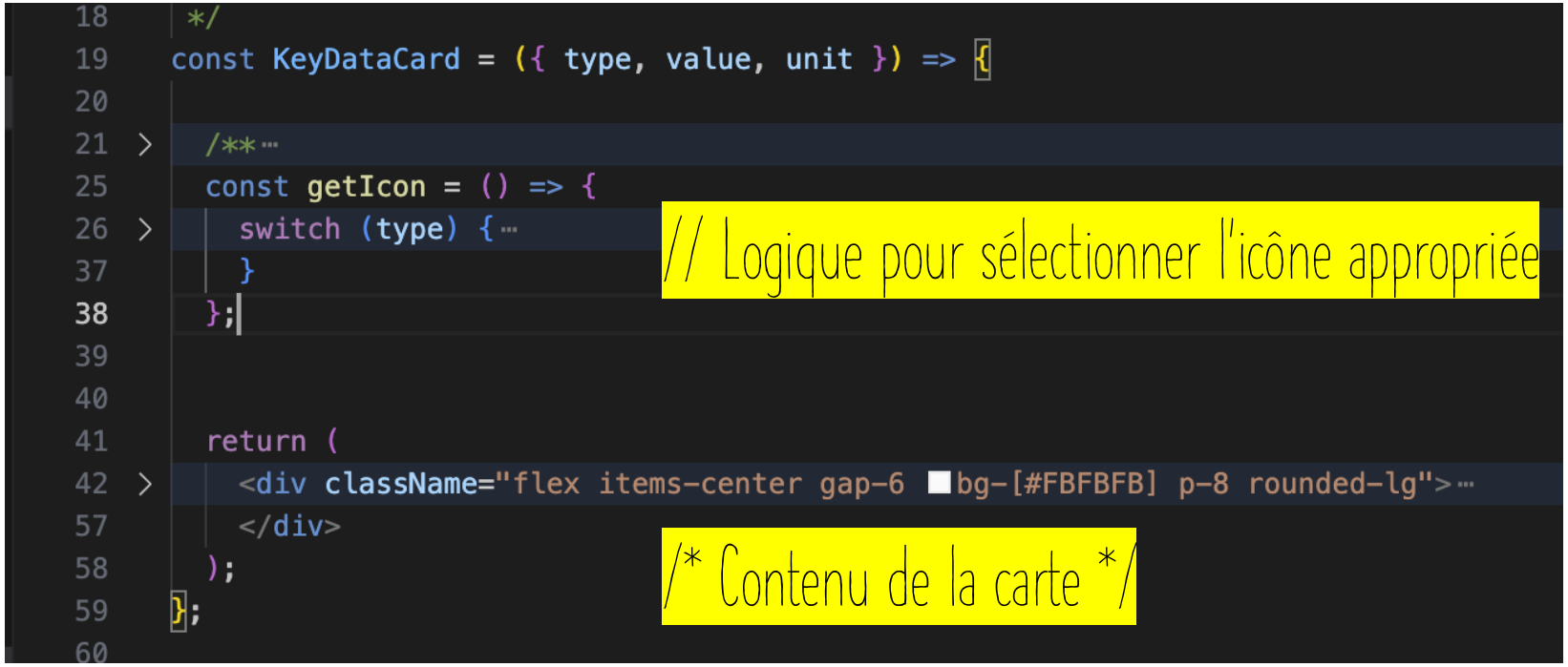
```jsx
return (
  <div className="bg-[#FBFBFB] p-6 rounded-lg h-full">
    <div className="flex justify-between items-center mb-8">…
    </div>
    <ResponsiveContainer width="100%" height="85%">
      <BarChart data={formatActivityData(data)} barGap={8} barCategoryGap="35%">
        <CartesianGrid strokeDasharray="3 3" vertical={false} stroke="#DEDEDE" />
        <XAxis
          dataKey="name"…
          dy={15}
        />
        <YAxis
          yAxisId="right"…
          dx={15}
        />
        <YAxis yAxisId="left" orientation="left" hide={true} />
        <Tooltip
          contentStyle={{…
          labelStyle={{ display: 'none' }}
        />
        <Bar
          yAxisId="right"…
          barSize={7}
        />
        <Bar
          yAxisId="left"…
          barSize={7}
        />
      </BarChart>
    </ResponsiveContainer>
  </div>
);
```

/* Configuration du graphique */

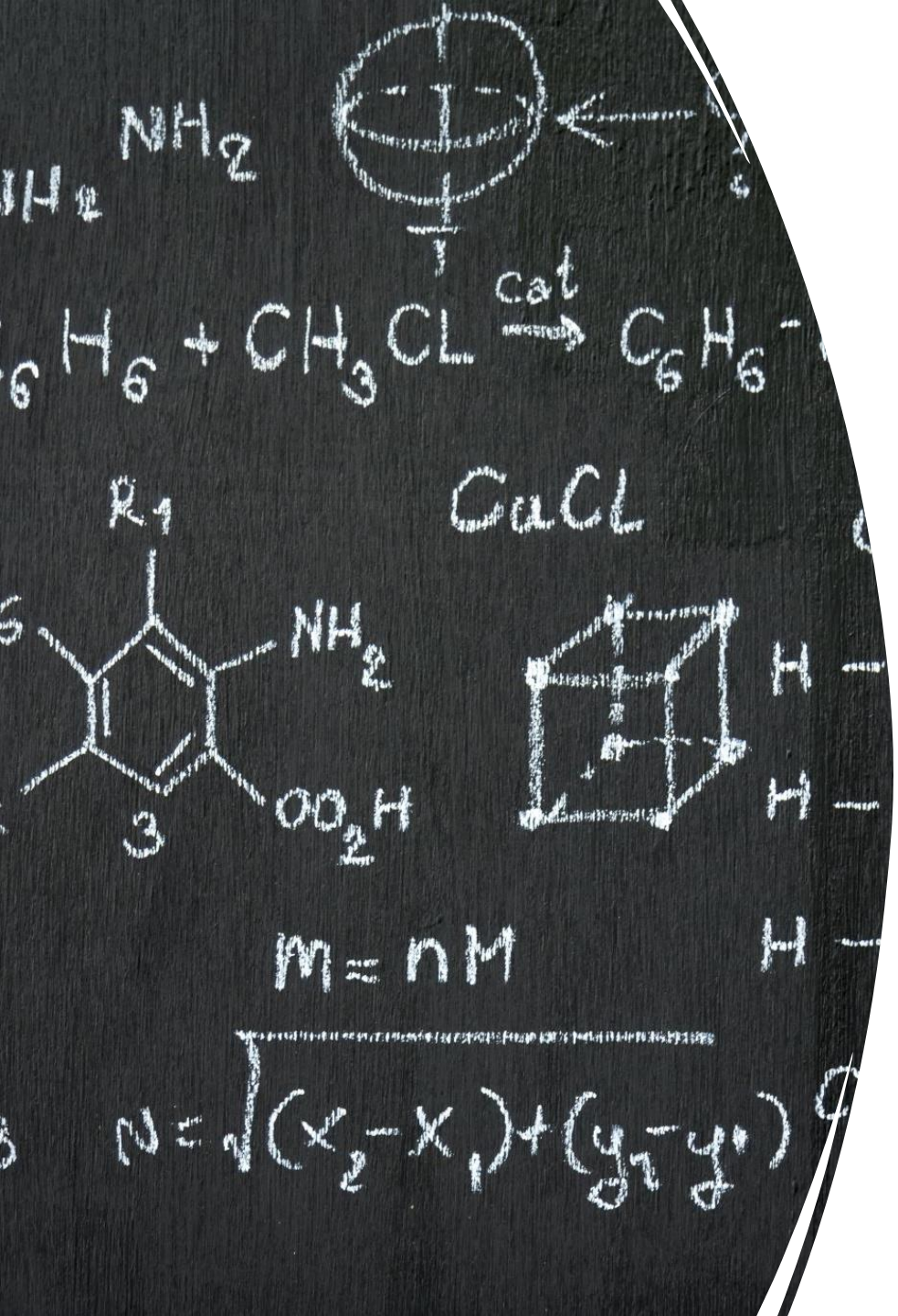


3. Séparation logique du code dans des composants réutilisables

- Notre application est structurée en composants réutilisables. Prenons l'exemple de `KeyDataCard.jsx` :



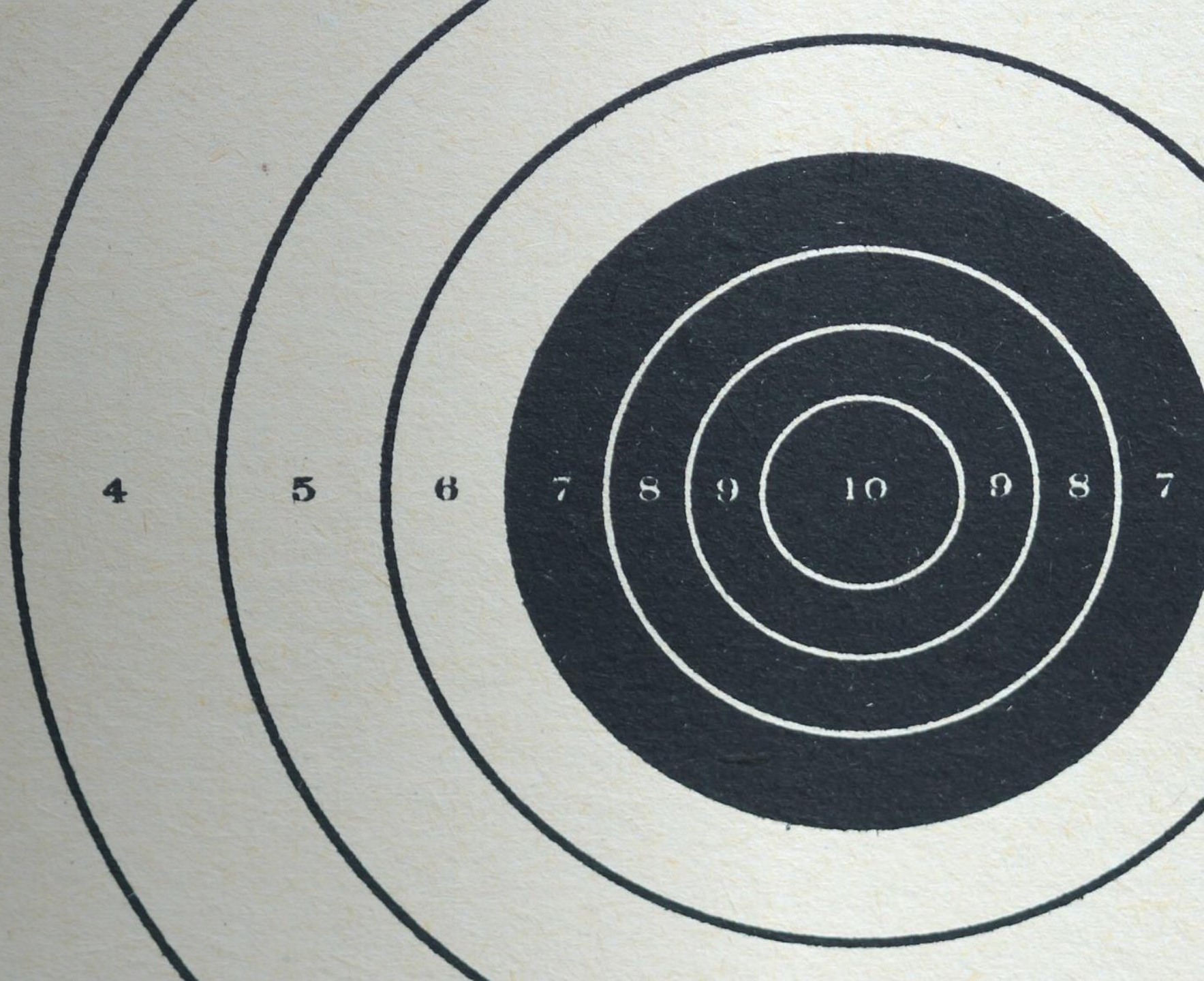
- Ce composant est réutilisable pour afficher différents types de données clés de l'utilisateur. Nous utilisons des props pour personnaliser chaque instance du composant.



conclusion,

- En conclusion, notre application de suivi de santé démontre une utilisation efficace de React et de ses écosystèmes. Nous avons créé une structure modulaire, géré les appels API de manière asynchrone, et utilisé Recharts pour des visualisations de données attrayantes. À l'avenir, nous pourrions envisager d'ajouter plus de fonctionnalités interactives et d'optimiser les performances pour de grands ensembles de données.

Demo



1. La gestion des calls asynchrones et des promesses

Dans notre hook `useFetch`, nous utilisons `async/await` pour gérer les appels asynchrones :

```
const fetchData = async () => {
  try {
    const response = await fetch(`${API_BASE_URL}${userId}/${type}`);
    // Traitement de la réponse
  } catch (err) {
    // Gestion des erreurs
  }
};
```

Cette approche rend notre code plus lisible et plus facile à maintenir par rapport à l'utilisation de callbacks imbriqués.

2. Le cycle de vie d'une application React avec des données asynchrones

- Notre application gère efficacement le cycle de vie des composants React avec des données asynchrones. Dans `Dashboard.jsx`, nous utilisons `useState` et `useEffect` :

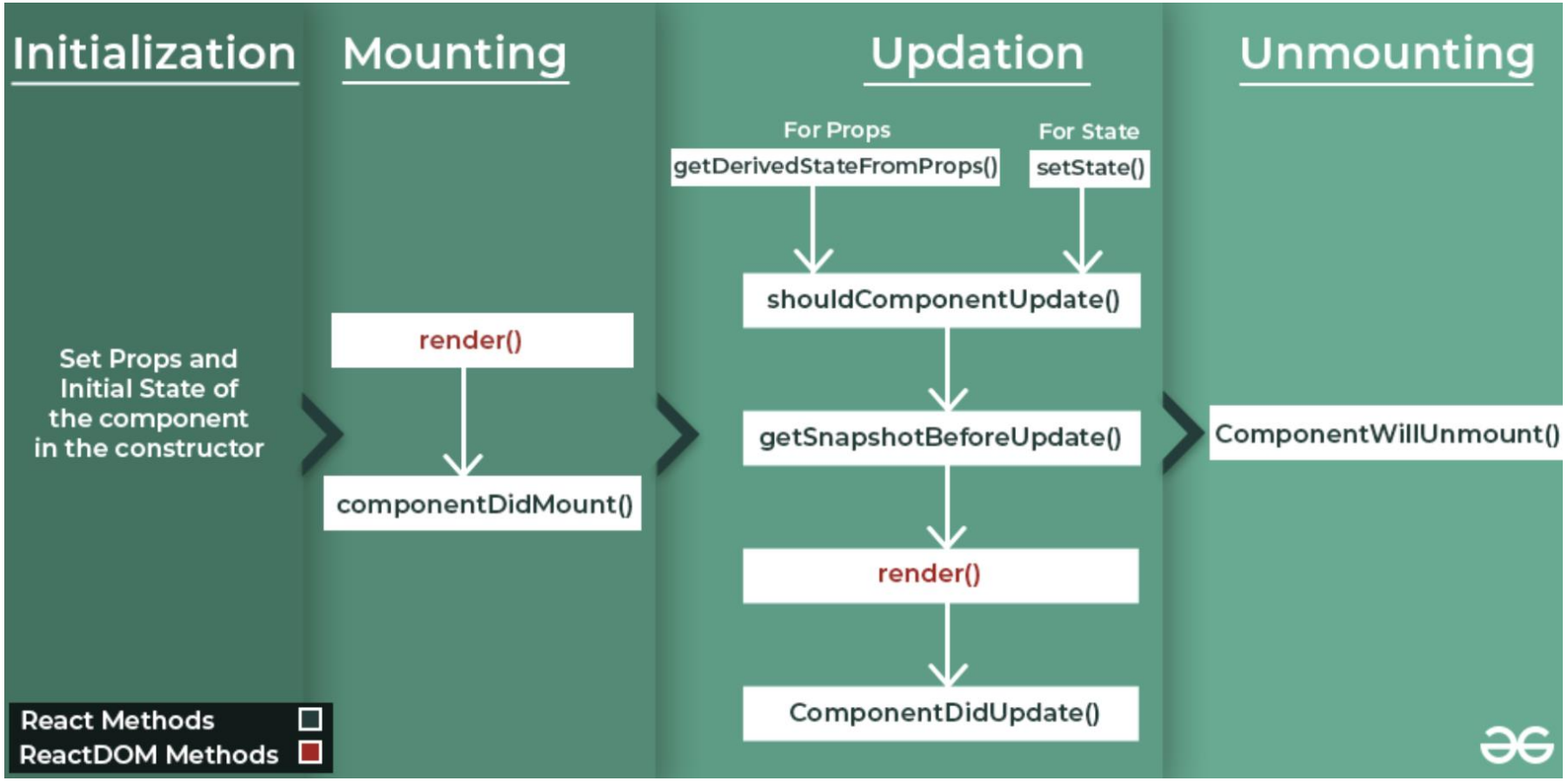
```jsx
// Extrait hypothétique de Dashboard.jsx
const Dashboard = ({ userId }) => {
  const { data, isLoading } = useFetch(userId);

  if (isLoading) {
    return <div>Loading dashboard...</div>;
  }

  return (
    <div>
      <ActivityChart userId={userId} />
      <PerformanceChart userId={userId} />
      {/* Autres composants */}
    </div>
  );
};
```

Cette structure nous permet de gérer facilement les états de chargement et d'affichage des données.

Le **cycle de vie de React :** Chaque composant de React possède un cycle de vie que vous pouvez surveiller et manipuler au cours de ses trois phases principales : **le montage** , **la mise à jour** et **le démontage**



Le **cycle de vie de React (ref https://www.geeksforgeeks.org/reactjs-lifecycle-components/)**

```jsx
// Extrait de ScoreChart.jsx
<PieChart>
  <Pie
    data={data}
    dataKey="value"
    cx="50%"
    cy="50%"
    innerRadius={70}
    outerRadius={80}
    startAngle={90}
    endAngle={450}
  >
    <Cell fill="#FF0000" />
    <Cell fill="#FBFBFB" />
  </Pie>
</PieChart>
```

3. L'utilisation de la librairie graphique

- Recharts s'est avéré être un excellent choix pour notre projet. Sa flexibilité nous permet de créer des graphiques personnalisés et réactifs. Par exemple, dans `ScoreChart.jsx` :