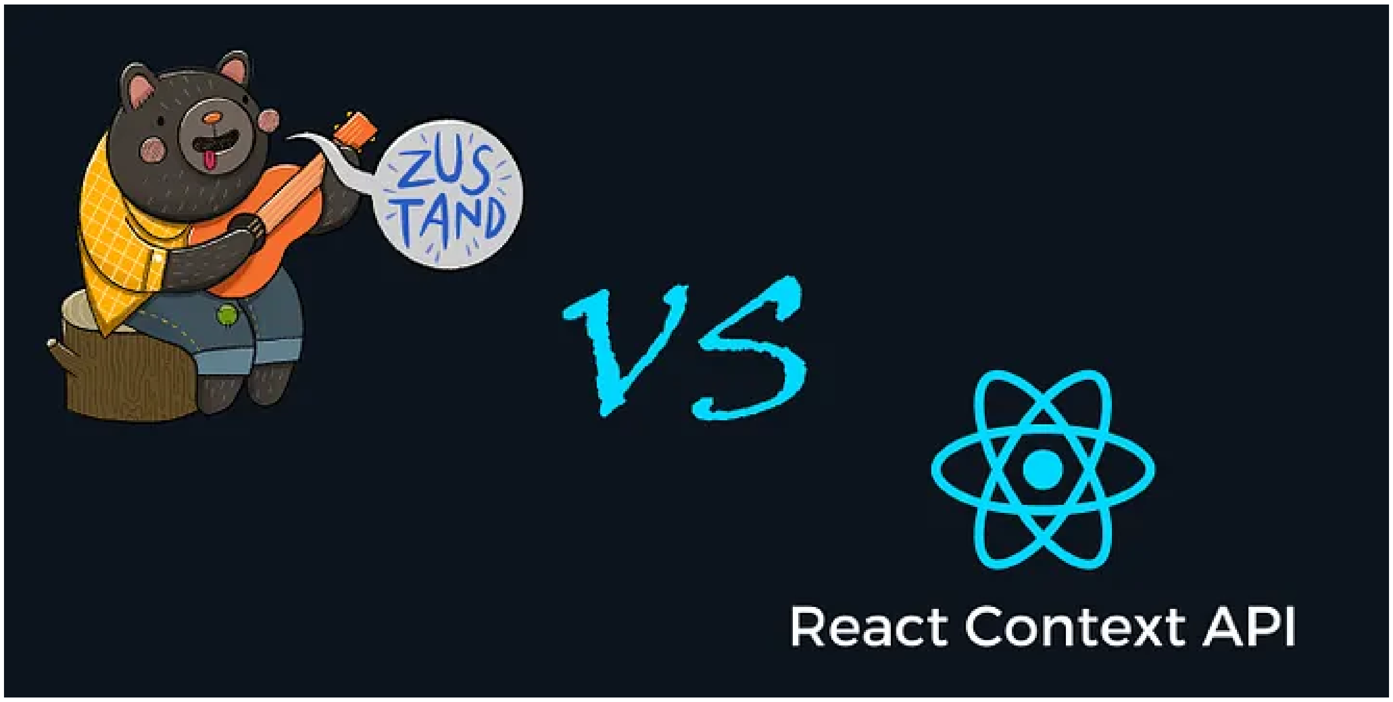
React Context API vs Zustand State Manager

Vimukthi Jayasinghe · Follow
5 min read · Feb 16, 2022

Listen Share

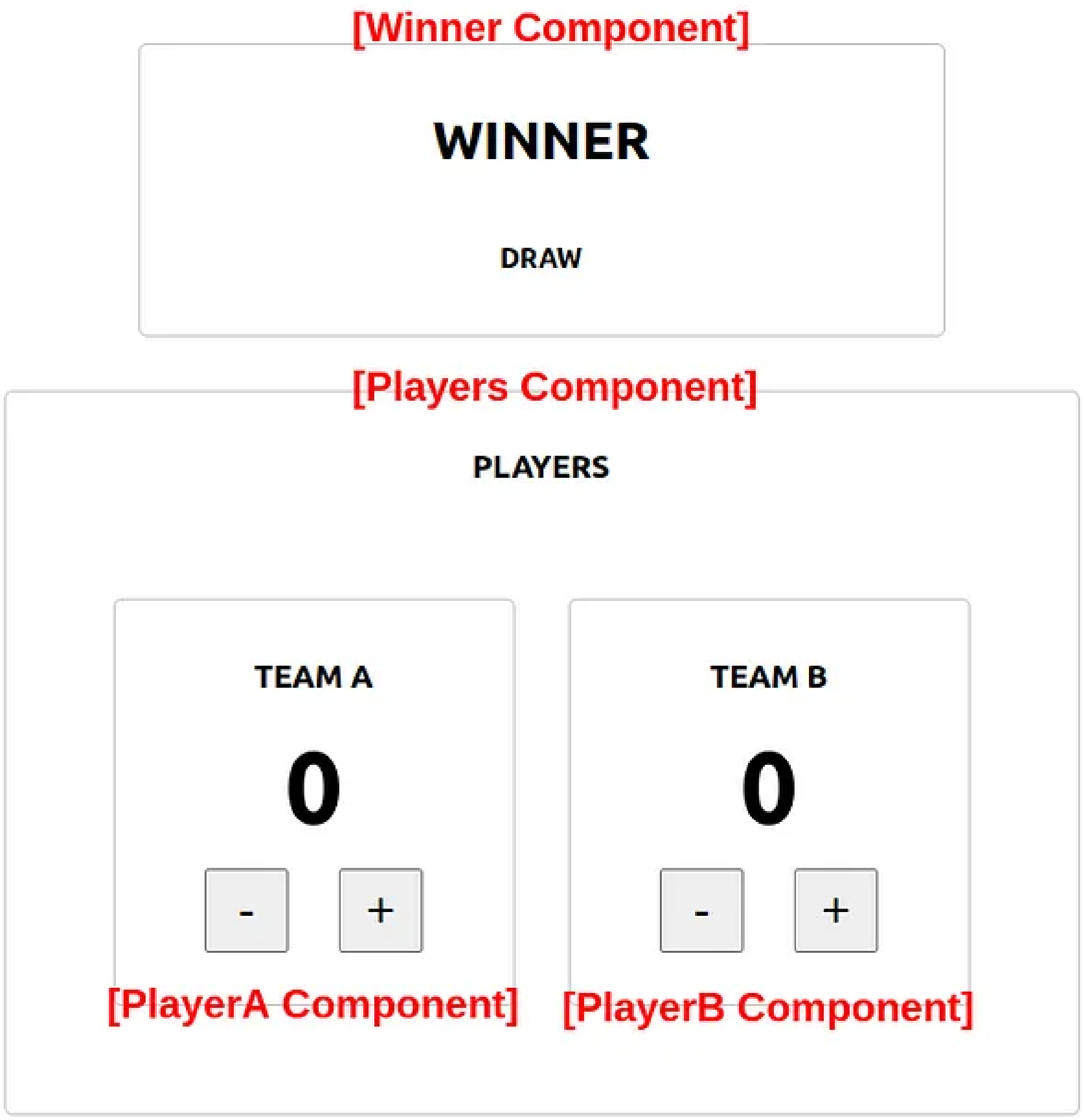


react context api vs Zustand

This article will show a practical performance comparison between the usage of React Context API and the Zustand State Manager library by solving one scenario with the above-mentioned approaches.

In the production level frontend application that uses React JS, we may need to manage the state. Indeed that is why we focus on frontend frameworks or libraries most of the time. When it comes to the manage our application state globally, we meet lots of state managers or state management patterns and utilities. As we know React JS is quite statable in the market and because of that we can get multiple approaches to manage our frontend application’s state. In this article, I am going to compare the React Context API and the Zustand State Manager with a practical example.

First of all, let's understand what we are going to build.

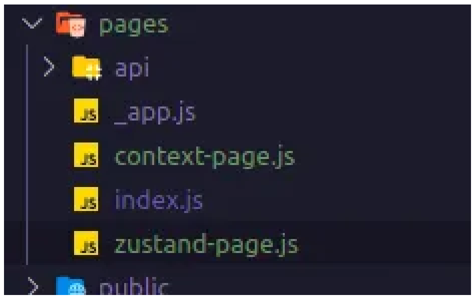


This application contains two main components that show the final winner called the winner component. It will compare the scores achieved by Player A and Player B then show the real-time results. Individual Player' component has their own behavior. By pressing those plus and minus icon buttons scores of each team will be increased or decreased respectively.

I am going to create a Next.js Application for this demo. with that, I will create two routes one for the solution with Context API and the other with the Zustand.

```
npx create-next-app@latest
# or
yarn create next-app
```

Create a new Next.js Application using the above command and give it a proper name in the installation process. after completing the installation get your app up and running. Then under the pages directory, I created two files called "***context-page***" and "***zustand-page***".



```
pages
  api
  _app.js
  context-page.js
  index.js
  zustand-page.js
public
```

Let's take our first approach using Context API.

This is how we keep the Player scores in the store.

```
{
     teamA, // Team A's score
     teamB, // Team B's score
     increaseTeamAScore, // Increase Team A's score by one
     decreaseTeamAScore, // Decrease Team A's score by one
     increaseTeamBScore, // Increase Team B's score by one
```

```
        decreaseTeamBScore, // Decrease Team B's score by one
    }
```

By keeping in mind of above store design, let's create our context.

```
import { createContext, useState, useContext } from "react";

const Context = createContext(null);

const useStore = () => {
  const [teamA, setTeamA] = useState(0);
  const [teamB, setTeamB] = useState(0);
  return {
    teamA, // Team A's score
    teamB, // Team B's score
    increaseTeamAScore: () => setTeamA((v) => v + 1), // Increase Team A's score by one
    decreaseTeamAScore: () => setTeamA((v) => v - 1), // Decrease Team A's score by one
    increaseTeamBScore: () => setTeamB((v) => v + 1), // Increase Team B's score by one
    decreaseTeamBScore: () => setTeamB((v) => v - 1), // Decrease Team B's score by one
  };
};

const StoreContextProvider = ({ children }) => {
  return (
    <Context.Provider value={useStore()} r>
      {children}
    </Context.Provider>
  );
};

export const useStoreContext = () => useContext(Context);

export default StoreContextProvider;
```

store-context.js hosted with ❤ by GitHub view raw

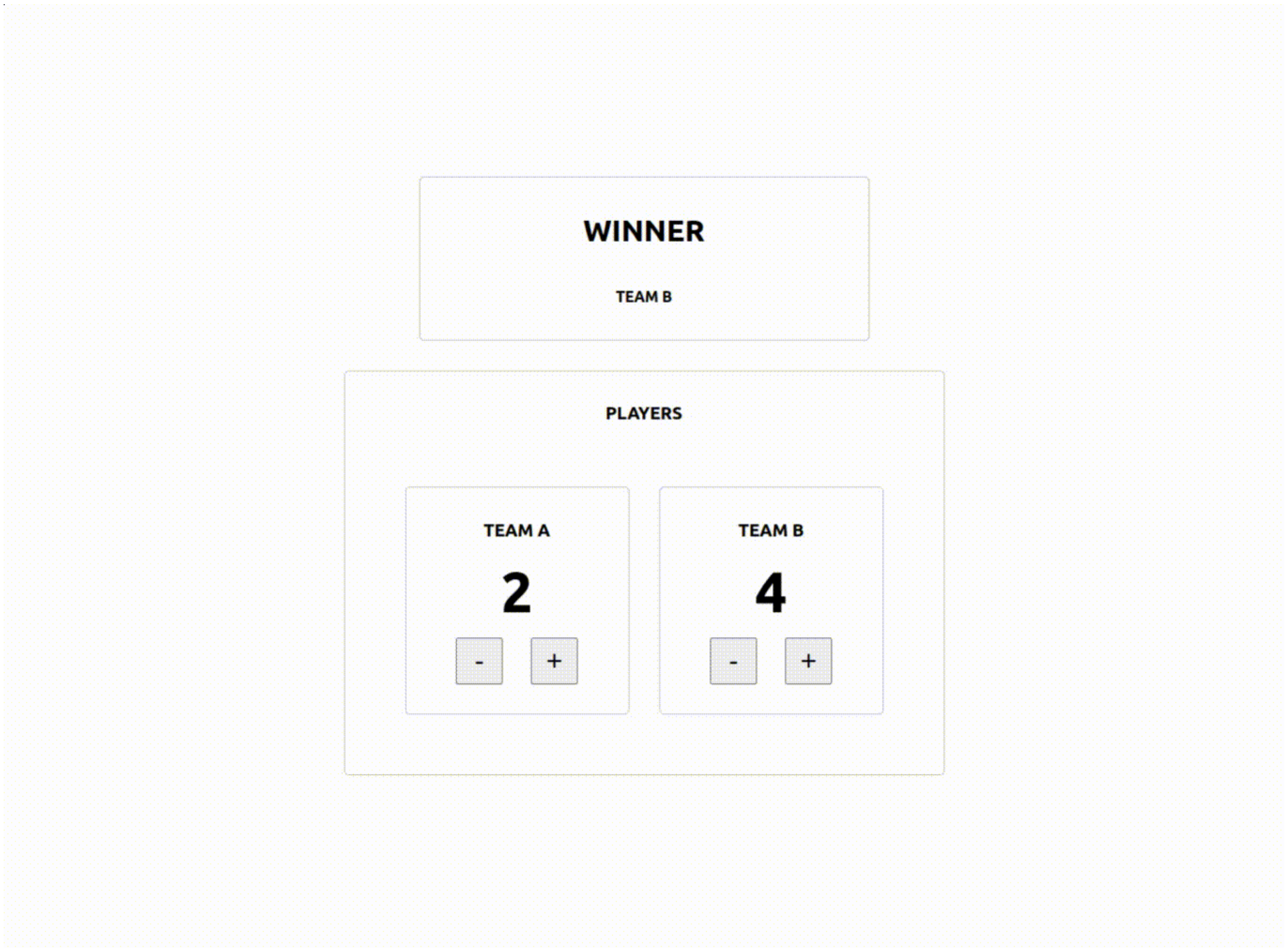
Now we are going to create *context-page.js* file. This will include the components that we talk earlier with a diagram.

```
import StoreContextProvider, {
  useStoreContext,
} from "../context/store-context";
import styles from "../styles/Home.module.css";

// Winner Component that shows the final results.
const Winner = () => {
  const { teamA, teamB } = useStoreContext();
  return (
    <div className={styles.winner}>
      <h1 className={styles.text}>Winner</h1>
      <h4 className={styles.text}>
        {teamA === teamB ? "DRAW" : teamA > teamB ? "TEAM A" : "TEAM B"}
      </h4>
    </div>
  );
};

// Reusable player component that render Players details and actions.
const Player = ({ label, score, onIncrease, onDecrease }) => (
  <div className={styles.playerContainer}>
    <h3 className={styles.text}>{label}</h3>
    <h4 className={styles.score}> {score} </h4>

    <div className={styles.btnWrapper}>
      <button className={styles.btn} onClick={onDecrease}>
        -
      </button>
      <button className={styles.btn} onClick={onIncrease}>
        +
      </button>
    </div>
  </div>
);

// Player A Components that subscribed the Context.
const PlayerA = () => {
  const { teamA, increaseTeamAScore, decreaseTeamAScore } = useStoreContext();

  return (
    <Player
      label={"Team A"}
      score={teamA}
      onIncrease={increaseTeamAScore}
      onDecrease={decreaseTeamAScore}
      onDecrease={decreaseTeamAScore}
    />
  );
};

// Player B Components that subscribed the Context.
const PlayerB = () => {
  const { teamB, increaseTeamBScore, decreaseTeamBScore } = useStoreContext();

  return (
    <Player
      label={"Team B"}
      score={teamB}
      onIncrease={increaseTeamBScore}
      onDecrease={decreaseTeamBScore}
    />
  );
};

// Players component will contains all active player components
const Players = () => {
  console.log("players");
  return (
    <div className={styles.playersContainer}>
      <h3 className={styles.text}>Players</h3>
      <div className={styles.playersWrapper}>
        <PlayerA />
        <PlayerB />
      </div>
    </div>
  );
};

export default function ContextComponent() {
  return (
    <div className={styles.container}>
      <main className={styles.main}>
        <StoreContextProvider>
          <Winner />
          <Players />
        </StoreContextProvider>
      </main>
    </div>
  );
}
```

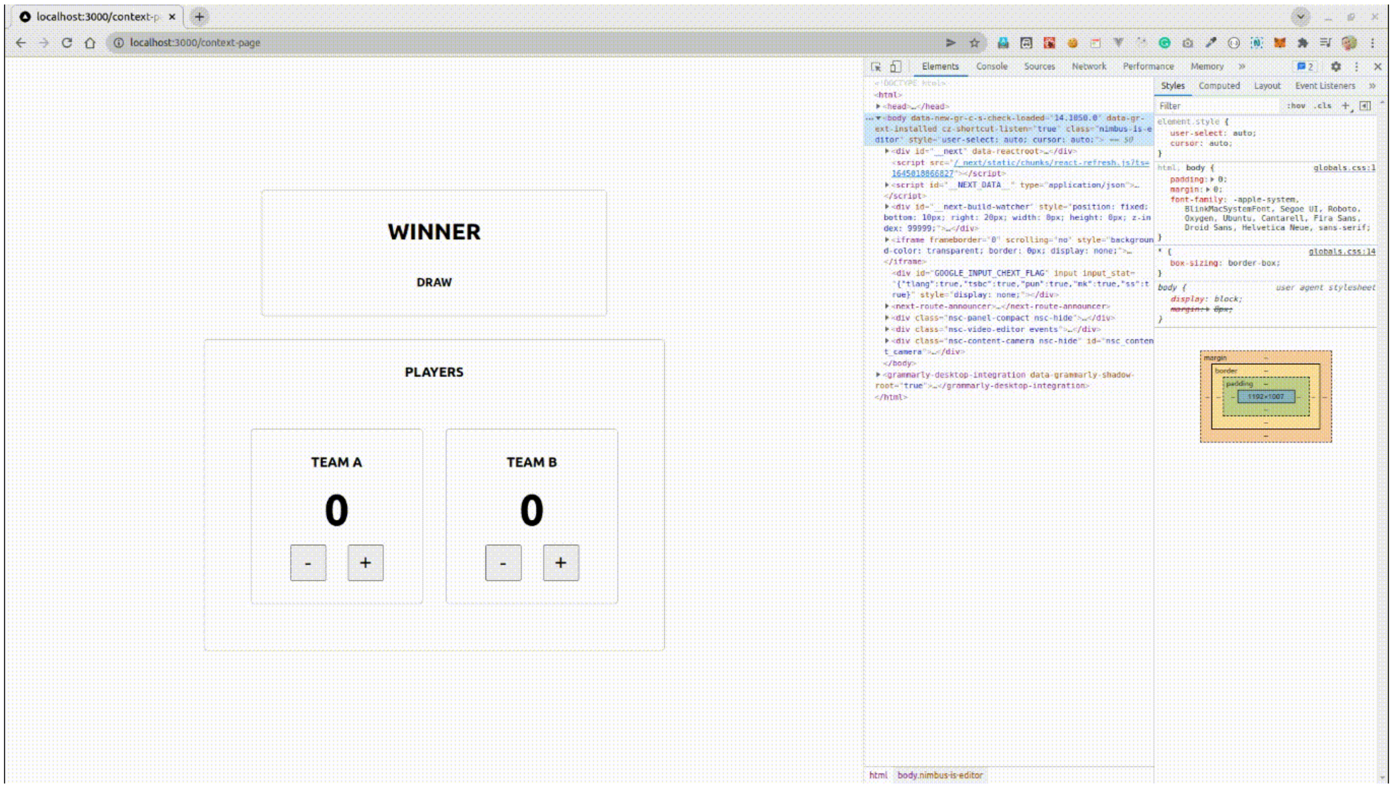
context-page.js hosted with ❤ by GitHub view raw

Easy Peasy..!! We Created our first solution with Context API. Go to the route below and check the functionality. Since we are

`http://localhost:3000/context-page`



Okay, now you can see our solution's appearance is fine and working as we expected. Let's have a look at this solution from a performance perspective. To do that first of all I am going to term on the "***Highlight updates when components render.***" feature that comes with React Dev Tools Profiler. I'll go to the Profiler tab and then click on the Gear icon placed on the top right side. Then I will check the option.



```
<!DOCTYPE html>
<html>
 ▸<head>…</head>
…▾<body data-new-gr-c-s-check-loaded="14.1050.0" data-gr-
  ext-installed cz-shortcut-listen="true" class="nimbus-is-e
  ditor" style="user-select: auto; cursor: auto;"> == $0
   ▸<div id="__next" data-reactroot>…</div>
    <script src="/_next/static/chunks/react-refresh.js?ts=
    1645018866827"></script>
   ▸<script id="__NEXT_DATA__" type="application/json">…
   </script>
   ▸<div id="__next-build-watcher" style="position: fixed;
   bottom: 10px; right: 20px; width: 0px; height: 0px; z-in
   dex: 99999;">…</div>
   ▸<iframe frameborder="0" scrolling="no" style="backgroun
   d-color: transparent; border: 0px; display: none;">…
   </iframe>
    <div id="GOOGLE_INPUT_CHEXT_FLAG" input input_stat=
    "{"tlang":true,"tsbc":true,"pun":true,"mk":true,"ss":t
    rue}" style="display: none;"></div>
   ▸<next-route-announcer>…</next-route-announcer>
   ▸<div class="nsc-panel-compact nsc-hide">…</div>
   ▸<div class="nsc-video-editor events">…</div>
   ▸<div class="nsc-content-camera nsc-hide" id="nsc_conten
   t_camera">…</div>
  </body>
 ▸<grammarly-desktop-integration data-grammarly-shadow-
 root="true">…</grammarly-desktop-integration>
</html>
```

```
element.style {
  user-select: auto;
  cursor: auto;
}
html, body {                                   globals.css:1
  padding: ▸ 0;
  margin: ▸ 0;
  font-family: -apple-system,
     BlinkMacSystemFont, Segoe UI, Roboto,
     Oxygen, Ubuntu, Cantarell, Fira Sans,
     Droid Sans, Helvetica Neue, sans-serif;
}
* {                                            globals.css:14
  box-sizing: border-box;
}
body {                               user agent stylesheet
  display: block;
  ~~margin: ▸ 8px;~~
}
```

Once that option is enabled, you can see that highlighted sections. It is happening because when we wrap the component tree with Context, it will always rerender all components under the Context Prover when the context change happens.

When we increase or decrease the score of Team A, actually we do not need to rerender the Team B components. since the entire components are wrapped with the StoreContextProvider and actions have done to the Team A component will affect to context, the Team B component also going to reprint. That is why the Team B component also going to be highlighted by the Profiler.

Because scores are compared inside the Winner component, it should rerender. But we do not need to rerender the Team B component. That is the issue with the Context API. It is going to rerender the entire scope that is wrapped with the context provider. But in the real world, we are expecting to rerender changed components only.

To overcome such scenarios, We can take different actions. We can use state managers such as ***Zustand*** or libraries like "***use-context-selector***". In today's post, I am going to use the Zustand state manager library to overcome this issue.

The store design is the same as above and let's have a deep dive in the Zustand library.

Create a Zustand store.

```
import create from "zustand";

const useStore = create((set) => ({
  teamA: 0,
  teamB: 0,
  increaseTeamAScore: () => set((state) => ({ teamA: state.teamA + 1 })),
  decreaseTeamAScore: () => set((state) => ({ teamA: state.teamA - 1 })),
  increaseTeamBScore: () => set((state) => ({ teamB: state.teamB + 1 })),
  decreaseTeamBScore: () => set((state) => ({ teamB: state.teamB - 1 })),
}));
```

create-store.js hosted with ❤ by GitHub view raw

You can see it is easy to create our store with Zustand and create all action methods to deal with the state. Now we will see how to subscribe a component to the store.

```
      <div className={styles.winner}>
        <h1 className={styles.text}>Winner</h1>
        <h4 className={styles.text}>
          {scoreA === scoreB ? "DRAW" : scoreA > scoreB ? "TEAM A" : "TEAM B"}
        </h4>
      </div>
    );
};
```

subscript-to-store.js hosted with ❤ by GitHub view raw

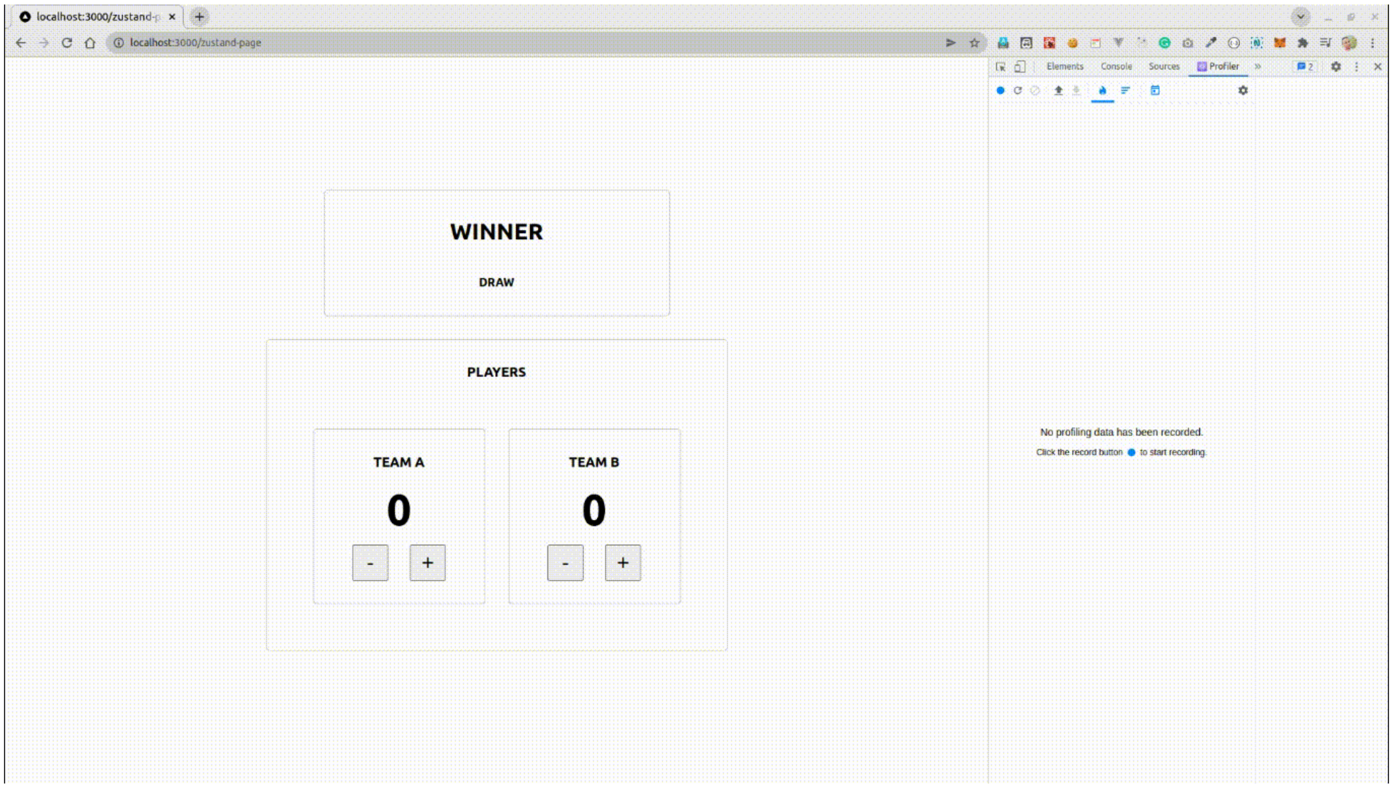
You can see how easily we can subscribe to the store that we created. That is the power of Zustand. It is a small, fast, and scalable barebones state-management solution using simplified flux principles according to their documentation and now we know because we started to use that too.

With the above introduction to Zustand, Let's create our solution with the help of Zustand. First of all, install the package to our Application.

```
yarn add zustan # or dnpm install zustand
```

Then let’s create the “***zustand-page.js***” file with the solution.

Okay, we have addressed the same problem but with a different solution. We can see the expected behavior achieved and now let’s see how the rerendering happens.


localhost:3000/zustand-page
WINNER
DRAW
PLAYERS
TEAM A
0
-
+
TEAM B
0
-
+
Elements
Console
Sources
Profiler
No profiling data has been recorded.
Click the record button ● to start recording.

According to the Profiler, now you can see how the rerenders happen. With the help of Zustand, we have achieved the perfoamce issue too. And the creation of the store and subscribing to the Components are also pretty straightforward.

There are a lot more features offered with the Zustand. You can go through their documentation and find more.

Thank you for reading 🙇.

Feel free to **share** and/or **follow me.** It would help me a lot 😁

React · React Context Api · Prop Drilling · Zustand · Context Api