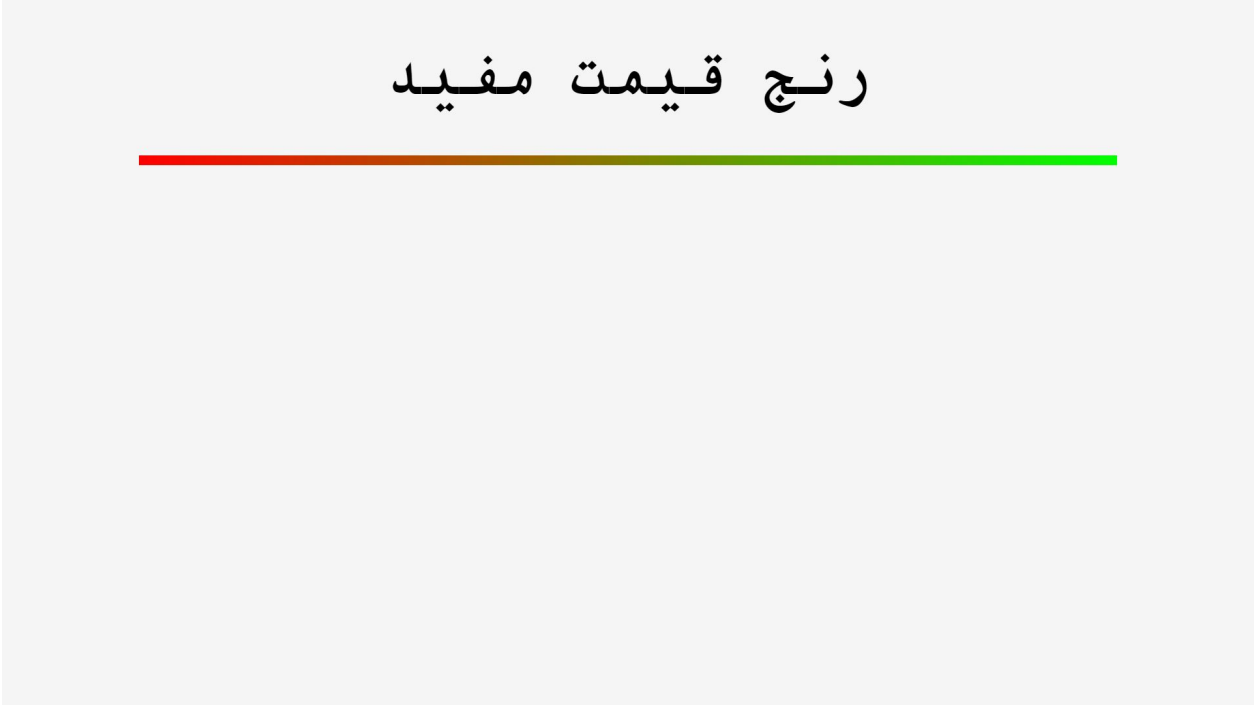
رِنج قیمت مفید

کارگزاری مفید در جهت بهبود رابط کاربری خود شروع به اعمال تغییرات ریز و درشتی کرده است. یکی از این تغییرات مربوط به نمایش محدوده قیمتی یک سهم است. در حال حاضر این بخش یا خیلی قرمز است یا خیلی سبز؛ به همین جهت تصمیم گرفته شده که این بخش بازطراحی شود.



جزئیات پروژه

پروژه‌ی اولیه را از این لینک دانلود کنید.

▼ ساختار فایل‌های پروژه

```
price-range
├── index.html
└── style.css
```

موارد خواسته‌شده

- در داخل `div.container` یک المان از نوع `div` با کلاس `price` ایجاد کنید.
- پس‌زمینه‌ی مربوط به کلاس `price` با استفاده از ویژگی گرادیان خطی *CSS* ، **به سمت راست** و از قرمز خالص یعنی `rgb(255, 0, 0)` شروع شده و با تغییرات ملایم به سبز خالص یعنی `rgb(0, 255, 0)` ختم شود. همچنین عرضی برابر با `500px` و ارتفاعی برابر با `5px` داشته باشد.

توجه: رنگ‌ها دقیقاً همان‌طور که در صورت سؤال ذکر شده باید به‌صورت `rgb` وارد شوند.

آن‌چه باید آپلود کنید

پس از اعمال تغییرات، فایل‌های `index.html` و `style.css` را زیپ کرده و آپلود کنید.

تایپینگ افکت

در این سؤال قرار است افکت تایپ کردن و حذف متن تایپ‌شده را در جاوااسکریپت پیاده‌سازی کنیم.

ظاهر کلی برنامه به‌صورت زیر است:

جزئیات پروژه

پروژه‌ی اولیه را از این لینک دانلود کنید.

▼ ساختار فایل‌های پروژه

```
typing-effect
├── index.html
├── main.js
└── styles.css
```

موارد خواسته شده

- یک `input` با شناسه‌ی `user-caption` وجود دارد که مقدار درون آن پس از کلیک کردن دکمه ای که شامل شناسه‌ی `test-typing` است **باید با افکت تایپ کردن در تگ `span` با شناسه‌ی `caption` نوشته شود.**
- دقت شود اگر در تگ `span` با شناسه‌ی `caption` از قبل متنی وجود داشته باشد، باید متن جدید جایگزین شود و **با افکت نوشته شود**.
- اگر متنی داخل تگ `span` با شناسه‌ی `caption` وجود داشته باشد و روی دکمه‌ای با شناسه‌ی `test-erasing` کلیک شود، باید **متن با افکت حذف شود**. روند حذف به این‌صورت است که هر بار آخرین کاراکتر رشته حذف می‌شود.
- اگر `input` خالی باشد و روی دکمه‌ی نوشتن کلیک شود، باید متن زیر در تگ `span` با شناسه‌ی `caption` **با افکت نوشته شود**:

```
typing test!
```

- اگر متنی برای حذف کردن وجود نداشته باشد، ابتدا باید متن زیر در تگ `span` با شناسه‌ی `caption` **بدون افکت (و بدون تأخیر)** نوشته شود و سپس متن **با افکت حذف شود**:

```
erasing test!
```

نکات

- سرعت تایپ هر کارکتر ۳۰ میلی‌ثانیه است.
- هر دو عملیات نوشتن و حذف کردن متن باید با ۳۰ میلی‌ثانیه تأخیر شروع شوند.
- در گیفی که برای این سؤال قرار داده شده، یک *space* قبل و یک *space* بعد از رشته ای که با افکت تایپ می‌شود مشاهده می‌شود. دلیل این فاصله‌ها، ساختار *HTML* پروژه است. بنابراین به فاصله‌ها توجه نکنید.
- در این سؤال فقط سرعت تایپ هر کارکتر تست می شود، به علاوه مقدار نهایی رشته که در تگ `span` با شناسه‌ی `caption` وجود دارد **(از قرار دادن *space* اضافه در تگ `span` با شناسه‌ی `caption` خودداری کنید)**.

- شما تنها مجاز به اعمال تغییرات در فایل `main.js` هستید.

آن‌چه باید آپلود کنید

پس از پیاده‌سازی موارد خواسته‌شده، فایل `main.js` را آپلود کنید (آن را زیپ کنید).

بشمار

آمیرزا با ۸۰ سال سن به برنامه‌نویسی علاقه‌مند شده است. او شنیده است که یکی از زبان‌های موردعلاقه‌ی جوانان جاوااسکریپت است، اما او از عالم کامپیوتر یک کامپیوتری دارد که با زغال کار می‌کند! این کامپیوتر حتی عملیات‌های ساده‌ی ریاضی همچون $2 + 2$ را به کندی انجام می‌دهد. وی که از این امر عاصی شده است، از شما می‌خواهد برای او کاری کنید که بتواند ببیند هر تابع چه زمانی برحسب میلی‌ثانیه طول می کشد تا اجرا شود.

جزئیات پروژه

در این سؤال باید تابعی با نام `timeit` طراحی شود که به‌صورت زیر عمل کند:

```
const fn = (a, b) => a + b;
timeit(fn)(5, 10).then(ans => {
    ans === {value: 15, time: 500} // true
})
```

در مثال بالا، میزان زمان ذکرشده براساس پرفورمنس سیستم آمیرزا است و روی کامپیوتر شما بسیار سریع‌تر خواهد بود (زمان تقریباً برابر با ۰٫۱ میلی‌ثانیه خواهد بود).

جالب است بدانید آمیرزا علاوه بر توابع معمولی از توابعی با رفتار `async` نیز استفاده می‌کند؛ پس انتظار می‌رود تابع شما برای این دسته از توابع نیز به خوبی کار کند. مثال:

```
function fn(t) {
    return new Promise((res, rej) => {
        setTimeout(() => res(`done after ${t}ms`), t);
    });
}

timeit(fn)(25).then(ans => {
    ans === {value: "done after 25ms", time: 25}
})
```

نکته: از آن‌جایی که زمان اجرایی توابع وابستگی به کلاک کاری پردازنده دارند، ممکن است میزان زمان محاسبه‌شده دارای مقداری خطا باشد. این میزان خطا در روند داوری لحاظ شده است.

آن‌چه باید آپلود کنید

فایل `main.js` که تابع `timeit` در آن پیاده‌سازی شده است را به صورت *ZIP* شده ارسال نمایید.

کانتکست‌ساز

روح‌الله در حال ساخت یک اپلیکیشن جدید است. او سعی دارد استیت‌های گلوبال اپلیکیشن خود را با کانتکست بنویسد، اما چون تنبل است، دوست ندارد هر بار این عمل را تکرار کند. تابعی به نام `createGlobalState` بنویسید که با گرفتن دستوراتی، این کار را برای او انجام دهد.

جزئیات پروژه

پروژه‌ی اولیه را از این لینک دانلود کنید.

▼ ساختار فایل‌های پروژه

```
context-maker
├── package.json
├── package-lock.json
├── public
│   └── index.html
├── README.md
└── src
    ├── App.js
    ├── contexts
    │   └── count.js
    ├── index.js
    ├── lib
    │   └── createGlobalState.js
    └── __tests__
        └── sample.test.js
```

▼ راه‌اندازی پروژه

- پروژه‌ی اولیه را دانلود و از حالت فشرده خارج کنید.
- با اجرای دستور `npm i` پکیج‌های مورد نیاز را نصب کنید.

- با اجرای دستور `npm start` می‌توانید پروژه را اجرا کنید، اما توجه داشته باشید که تا قبل از تکمیل فایل `src/lib/createGlobalState.js` ، پروژه به درستی نمایش داده نمی‌شود.

تابع `createGlobalState` هر بار که روح‌الله بخواهد استیتی را با کانتکست ایجاد کند باید قابل استفاده باشد.

مثلاً اگر روح الله بخواهد استیتی به نام `count` را با کانتکست ایجاد کند، باید به شکل زیر بتواند این کار را انجام دهد:

src/contexts/count.js

```
1  import { createGlobalState } from "../lib/createGlobalState";
2
3  const [CountProvider, useCount] = createGlobalState((set) => ({
4    count: 0,
5    increment(num) {
6      set((count) => count + num);
7    },
8    decrementByOne() {
9      set((count) => count - 1);
10   },
11   backToZero() {
12     set(0);
13   },
14 }));
15
16 export { CountProvider, useCount };
```

تابع `createGlobalState` به‌عنوان ورودی باید یک تابع بپذیرد. تابع ورودی نیز باید یک تابع دیگر به نام `set` دریافت کند.

تابعی که به `createGlobalState` پاس داده می‌شود باید یک آبجکت برگرداند که شامل نام استیت با مقدار اولیه‌ی دلخواه و تعدادی تابع به‌منظور ایجاد تغییر در استیت باشد.

تضمین می‌شود که در هر بار استفاده از `createGlobalState` ، آبجکت فوق فقط دارای یک استیت باشد و مابقی پراپرتی‌ها توابعی برای تغییر همان استیت باشند.

توابعی که برای تغییر استیت تعریف می‌شوند باید بتوانند با استفاده از تابع `set` مقدار استیت را تغییر دهند.

نکته: تابع `set` دقیقاً مانند `setter` هوک `useState` رفتار می‌کند.

خروجی تابع `createGlobalState` باید یک آرایه باشد که این آرایه به‌ترتیب شامل یک کامپوننت `Provider` و یک هوک است.

کامپوننت `Provider` باید یک آبجکت، مشابه همان آبجکتی که در `createGlobalState` قرار دادیم را در قالب `Context` در دسترس فرزندانش قرار دهد.

فرزندان کامپوننت `Provider` باید بتوانند با استفاده از هوکی که از آرایه‌ی خروجی تابع `createGlobalState` دریافت می‌کنیم از مقدار `Context` استفاده کنند.

مثالی از نحوه‌ی استفاده از کامپوننت `Provider` :

index.js

```
import ReactDOM from "react-dom";
import App from "./App";
import { CountProvider } from "./contexts/count";

ReactDOM.render(
  <CountProvider>
    <App />
  </CountProvider>,
  document.getElementById("root")
);
```

روح‌الله باید مجبور باشد از `Provider` مخصوص هر کانتکست استفاده کند. در غیر این‌صورت، باید یک ارور جدید از نوع آبجکت `Error` دریافت کند که باعث توقف در ادامه اجرای برنامه شود.

برای مثال اگر او بدون استفاده از `CountProvider` سعی کند از هوک `useCount` استفاده کند، باید چنین اروری را دریافت کند:

```
The 'count' global state must be used within it's relevant context provider
```

نکته: متن دقیق ارور مهم نیست، اما حتماً باید شامل نام استیت باشد.

روح‌الله باید بتواند با استفاده هوکی که از آرایه‌ی خروجی تابع `createGlobalState` دریافت می‌کند به‌شکل زیر از مقدار استیت و توابعی که تعریف کرده بود استفاده کند:

App.js

```jsx
import { useCount } from "./contexts/count";

function App() {

  const {count, increment, decrementByOne, backToZero} = useCount();

  return (
    <div>
      <h1>{count}</h1>
      <div>
        <button onClick={() => increment(5)}>
          Increment by 5
        </button>
        <button onClick={() => decrementByOne()}>
          Decrement by 1
        </button>
        <button onClick={() => backToZero()}>Back To Zero</button>
      </div>
    </div>
  );
}

export default App;
```

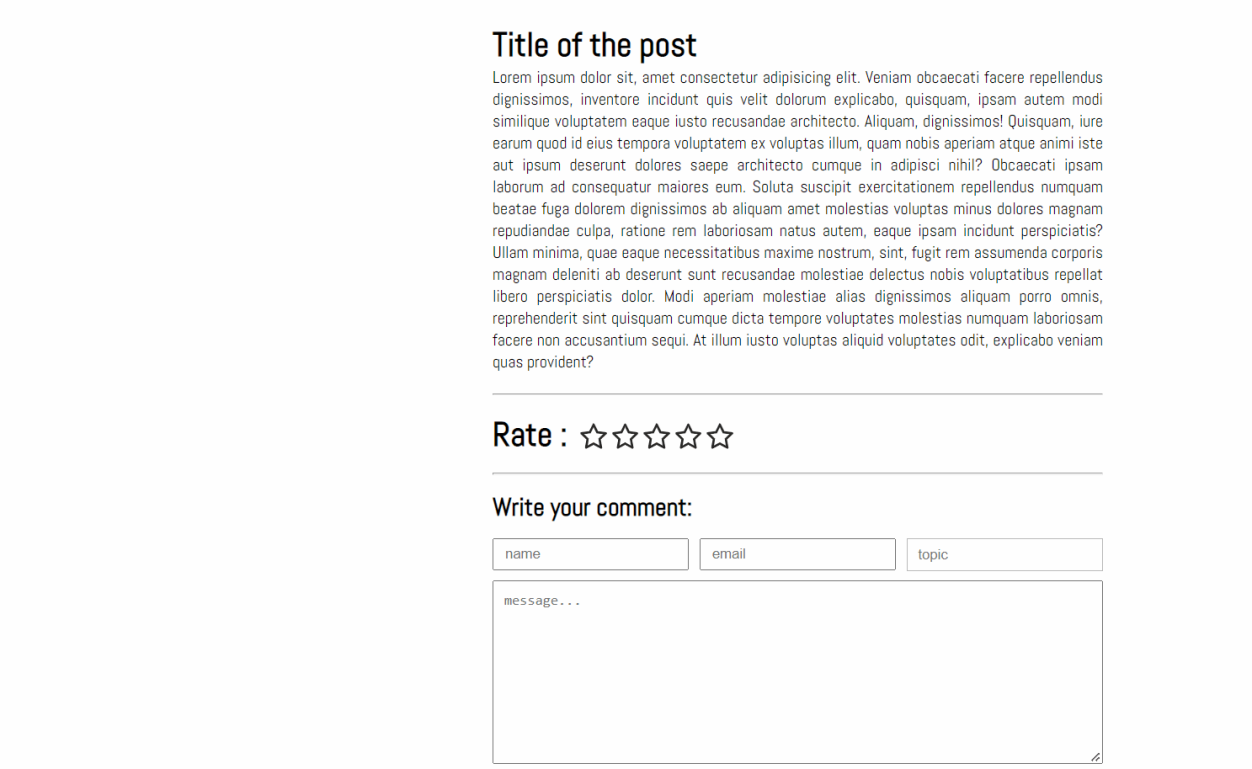
نکته: بدیهی است که مثال `count` بالا فقط یک نمونه از نحوه‌ی کار تابع `createGlobalState` است. شما باید این تابع را طوری پیاده‌سازی کنید که با ورودی‌های متفاوت از جمله تفاوت در نام استیت، مقدار اولیه‌ی استیت و توابع مربوط به تغییر در استیت به درستی عمل کند.

توجه: شما تنها مجاز به اعمال تغییرات در فایل `src/lib/createGlobalState.js` هستید.

آن‌چه باید آپلود کنید

پس از پیاده‌سازی موارد خواسته‌شده، پوشه‌ی `src` را زیپ کرده و آپلود کنید.

این که ناقصه!



چند وقت پیش عرفان پروژه‌ای را قبول کرد؛ پروژه‌ای که در آن باید صفحه‌ی یک مقاله را با ری‌اکت طراحی می‌کرد؛ صفحه‌ای که از چهار قسمت اصلی تشکیل شده که به شرح زیر است:

۱. عنوان و توضیحات مقاله
۲. سیستم امتیازدهی به مقاله
۳. سیستم ثبت نظر
۴. قسمت نمایش نظرات به‌صورت تودرتو

متأسفانه در اواسط پروژه بود که عرفان درگیر ویروس کرونا شد و پروژه ناقص ماند، اما از آن‌جایی که عرفان روی قولی که می‌دهد بسیار حساس است، از شما خواسته تا قسمت‌هایی از پروژه را که باقی مانده کامل کنید تا مبادا بد قول شود و بتواند پروژه را سر وقت تحویل دهد.

جزئیات پروژه

پروژه‌ی اولیه را از این لینک دانلود کنید.

▼ ساختار فایل‌های پروژه

```
comments
├── package.json
├── package-lock.json
├── public
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
│   ├── manifest.json
│   └── robots.txt
├── README.md
├── server
│   ├── app.js
│   ├── posts.json
│   └── techs.json
└── src
    ├── App.css
    ├── App.jsx
    ├── assets
    │   └── avatar.png
    ├── components
    │   ├── AddComment.jsx
    │   ├── Comment.jsx
    │   ├── Post.jsx
    │   ├── Rate.jsx
    │   └── SelectBox.jsx
    ├── container
    │   └── Comments.jsx
    ├── data
    │   └── data.js
    ├── index.js
    └── __tests__
        └── sample.test.js
```

▼ راه‌اندازی پروژه

- پروژه‌ی اولیه را دانلود و از حالت فشرده خارج کنید.
- اجرای دستور `npm i` پکیج های مورد نیاز را نصب کنید.
- با اجرای دستور `npm run server` سرور را اجرا کنید.
- با اجرای دستور `npm start` پروژه را اجرا کنید.

قسمت‌های باقی‌مانده از پروژه که باید پیاده‌سازی شوند

پاسخ به نظر دیگران

در کامپوننت `Comment` دکمه‌ای تحت عنوان `reply` وجود دارد که با کلیک کردن روی آن به وضعیت پاسخ می‌رویم. در وضعیت پاسخ، اتفاقاتی رخ می دهد که به شرح زیر است:

۱. کاربر باید به المانی با کلاس `ac-wrapper` اسکرول شود. این المان در فایل `AddComment` وجود دارد.

۲. در فایل `AddComment` یک تگ `h2` وجود دارد که محتوای آن در وضعیت پاسخ باید برابر باشد با:

```
Write your comment in response to {name}:
```

که `{name}` برابر است با نام فردی که قرار است به او پاسخ داده شود.

و در وضعیت ثبت نظر باید برابر باشد با:

```
Write your comment:
```

۳. در وضعیت پاسخ باید بعد از دکمه‌ی `Send` در فایل `AddComment` دکمه‌ای تحت عنوان `Cancel` اضافه شود که در صورت کلیک کردن روی این دکمه، باید از وضعیت پاسخ به وضعیت ثبت نظر برگردیم.

۴. اگر در وضعیت ثبت نظر باشیم، در فایل `AddComment` بعد از `input` با نوع `email` باید کامپوننت `SelectBox` فراخوانی شود، اما اگر در وضعیت پاسخ باشیم، این کامپوننت نباید در صفحه وجود داشته باشد.

انتخاب *topic*

مقالات موضوعات مختلفی را شامل می‌شوند. کاربر هنگام ثبت نظر باید *topic* یا موضوعی را مشخص کند. برای این کار، عرفان سروری آماده کرده است که یک رشته دریافت می‌کند و بین *topic* های موجود جست‌وجو می‌کند و لیست *topic* هایی که رشته‌ی ارسال‌شده در آن‌ها وجود دارد را بر می‌گرداند. برای مثال اگر *topic* ها در سرور به‌صورت زیر باشند:

```
php , python , java , c++ , Go
```

اگر حرف `p` برای سرور ارسال شود، سرور لیست زیر را برمی‌گرداند:

```
{
    "data": {
        "matchedTechs": [
            { "id": 1, "name": "php" },
            { "id": 2, "name": "python" }
        ]
    },
    "status": "success"
}
```

در فایل `components/SelectBox` یک `input` با کلاس `tpc` وجود دارد. در صورت تغییر `value` این `input` ، باید یک درخواست *GET* به آدرس `http://127.0.0.1:8000/` ارسال شود و مقدار `input` در *query string* با کلید `search` ارسال کنید تا لیست *topic* ها دریافت شوند.

عرفان برای نمایش *topic* ها یک المان `div` با کلاس `c-selectbox` ساخته است. این المان به نحوی طراحی و استایل‌دهی شده است که شبیه به یک *dropdown* باشد. اگر استایل های زیر را داشته باشد:

```
padding: 0;
height: 0;
overflow: "hidden";
```

لیست topic ها نمایش داده نمی‌شود و *dropdown* در وضعیت بسته است و اگر این استایل ها را نداشته باشد، *dropdown* در وضعیت باز است و لیست *topic* ها در صفحه نمایش داده می شود.

در صورتی که لیست *topic* ها خالی باشد، *dropdown* باید بسته باشد.

در صورتی که لیست *topic* ها خالی نباشد، باید لیست را در المان `div` با کلاس `c-selectbox` به‌صورت زیر رندر کنید:

```
<div className="item">
    <label htmlFor={نام تاپیک}{آیدی تاپیک}></label>
    <input type="radio" name="" id={آیدی تاپیک} />
</div>
```

با کلیک بر روی المان `div` با کلاس `item` که رندر کرده‌اید، اتفاقات زیر باید رخ دهد:

ابتدا باید dropdown بسته شود و سپس باید نام کامل آن *topic* به عنوان مقدار `input` با کلاس `tpc` قرار بگیرد

دقت کنید در زمانی که لیست تاپیک ها خالی نیست و ما در حال نمایش *topic* ها هستیم در صورتی که کاربر به‌صورت کلی روی `body` کلیک کند، باید *dropdown* بسته شود و مقدار `input` با کلاس `tpc` تغییر نکند.

امتیازدهی

در این بخش از سؤال، شما باید سیستم امتیازدهی به مقاله را پیاده‌سازی کنید. در فایل `Rate.jsx` یک استیت به نام `star` وجود دارد که مقدار اولیه‌ی آن به صورت زیر است:

```
[
    { id: 1, hover: false, clicked: false },
```

```
    { id: 2, hover: false, clicked: false },
    { id: 3, hover: false, clicked: false },
    { id: 4, hover: false, clicked: false },
    { id: 5, hover: false, clicked: false }
]
```

در صورتی که مقدار `hover` و `clicked` برابر با `true` باشد، ستاره‌های تو پر و در غیر این‌صورت، ستاره‌های تو خالی نمایش داده می‌شوند. در این بخش چهار تابع داریم که دو تا از آن‌ها به‌صورت کامل نوشته شده و دو تابع را باید شما پیاده‌سازی کنید. عملکرد دو تابعی که به صورت کامل از قبل نوشته شده به‌شرح زیر است:

▼ تابع hoverHandler

این تابع رویداد `onMouseEnter` روی ستاره‌ها را هندل می‌کند، به‌طوری که یک شناسه دریافت می‌کند و مقدار `hover` هر آبجکتی که شناسه‌ی آن‌ها از نظر عددی کوچک‌تر از شناسه‌ی دریافتی است را برابر با `true` می‌کند. برای مثال، فرض کنید کاربر ستاره‌ی چهارم را *hover* کرده است. ما باید ستاره‌هایی با شناسه‌ی `1` تا `4` را آپدیت کنیم و مقدار `hover` آن‌ها را `true` کنیم.

▼ تابع blurHandler

این تابع رویداد `onMouseLeave` را روی ستاره‌ها هندل می‌کند، به‌طوری که مقدار `hover` هر آبجکت را آپدیت کرده و آن را برابر با `false` قرار می‌دهد.

و اما دو تابعی که شما باید آن‌ها را کامل کنید:

تابع `submitRateHandler`

این تابع در قدم اول مانند تابع `hoverHandler` کار می کند، با این تفاوت که به جای `true` کردن مقدار `hover` هر آبجکت، باید مقدار `clicked` هر آبجکت را `true` کند. در ادامه، باید یک درخواست از نوع *PATCH* به آدرس `http://127.0.0.1:8000/posts/` ارسال کند و در بدنه‌ی درخواست، شناسه‌ی آخرین آبجکتی که ویژگی `clicked` آن برابر با `true` شده را به عنوان `rate` به صورت *JSON* به سرور ارسال کند.

متأسفانه در حال حاضر سرور خراب است و عرفان از دوستش سینا خواسته تا سرور را درست کند، اما سینا در سفر به سر می‌برد و شما باید با همین سرور خراب کار کنید! اشکال سرور این است که با احتمال ۵۰ درصد، `rate` کاربران را ثبت می‌کند و با احتمال ۵۰ درصد، درخواست شما با خطا مواجه می‌شود. علاوه بر این، سرور یک تأخیر یک ثانیه ای نیز دارد. اگر درخواست شما با موفقیت ثبت شود پیام سرور به شما به صورت زیر خواهد بود:

```
{
    "message": "Your rate for this post has been registered.",
    "status": "success"
}
```

پس از دریافت پیام بالا، باید مقدار `message` را به‌صورت یک *toast* نمایش دهید. برای این کار، عرفان به پیشنهاد مهیار از پکیج `react-toastify` استفاده کرده است. البته کدهای این قسمت را نیز در اختیار شما قرار داده است. کافی است در صورت دریافت پاسخ موفقیت‌آمیز، از تکه‌کد زیر استفاده کنید:

```
toast.success(پیامی که از سرور دریافت کرده اید, {
    position: "top-left",
});
```

در صورت *fail* شدن درخواست، پاسخ سرور به‌صورت زیر خواهد بود:

```
{
    "message": "Rating registering failed, try again.",
    "rate": آخرین امتیازی که در سرور ثبت شده است,
    "status": "error"
}
```

مقدار `rate` برابر با آخرین امتیاز ثبت‌شده‌ی کاربر است. شما باید این مقدار را به تابع `stepBackward` ارسال کنید.

به دلیل این که سرور تأخیر دارد، ما در زمان تآخیر باید امتیازی که کاربر ثبت کرده است را آپدیت و رندر کنیم، اما به محض این که پاسخ از سرور دریافت شود، در صورتی که درخواست *fail* شده باشد، باید

سیستم *rating* را مجدد آپدیت کنیم و به `rate` قبلی برگردیم **(منظور آخرین امتیازی است که کاربر وارد کرده و در سرور ثبت شده)**. به این تکنیک، *optimistic rendering* گفته می‌شود. تابع `stepBackward` در واقع قرار است این کار را برای ما هندل کند. پس از فراخوانی تابع `stepBackward` باید مجدداً یک *toast* به کاربر نمایش دهید. برای این کار، از تکه‌کد زیر استفاده کنید:

```
toast.error(پیامی که از سرور دریافت کرده اید, {
    position: "top-left",
});
```

تابع `stepBackward`

این تابع یک *rate* دریافت می‌کند و باید استیت مربوط به ستاره‌ها را آپدیت کند، به طوری که مقدار `clicked` مربوط به المان‌های دارای شناسه‌های کوچک‌تر یا مساوی `rate` را `true` کند و مقدار `hover` آن‌ها را `false` کند. برای مثال، استیت ستاره‌ها در هر وضعیتی که باشد، در صورت *fail* شدن درخواست اگر سرور مقدار `rate` را `3` برگرداند، باید آبجکت‌هایی با شناسه‌های `1` تا `3` طوری آپدیت شوند که مقدار `clicked` شان `true` و مقدار `hover` شان `false` شود.

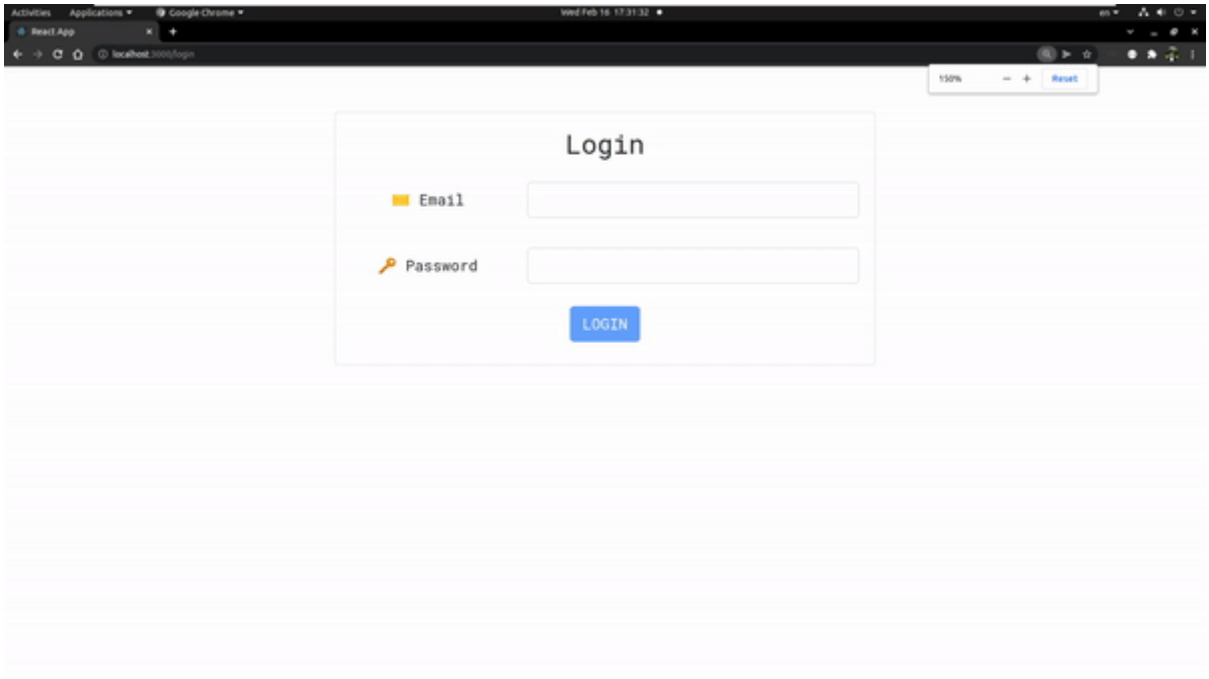
توجه: شما تنها مجاز به اعمال تغییرات در فایل‌های زیر هستید:

```
src/App.jsx
src/components/AddComment.jsx
src/components/Comment.jsx
src/components/Rate.jsx
src/components/SelectBox.jsx
```

آن‌چه باید آپلود کنید

پس از پیاده‌سازی موارد خواسته‌شده، پوشه‌ی `src` را زیپ کرده و آپلود کنید.

توکن‌چی



چند وقتی است که بحث توکن های *JWT* برای احراز هویت داغ شده است؛ به دلایل متعدد زیادی که بسیار هم موجه هستند. نیما تصمیم گرفته که مشقی به مهیار بدهد که این شیوه‌ی احراز هویت را در ری‌اکت به واسطه‌ی یک هوک پیاده کند. همچنین نیما زحمت پیاده‌سازی سروری که از این شیوه‌ی احراز هویت پشتیبانی می‌کند را قبول کرده است.

این سرور وظیفه‌ی احراز هویت کاربران را برعهده دارد و همواره توکنی که برمی‌گرداند زمان انقضای مشخصی دارد، به این معنا که پس از زمان تعیین‌شده، دیگر توکن قابل استفاده نیست. از این توکن تحت عنوان *Access Token* نیز یاد می‌شود. در کنار این توکن، یک توکن برای دریافت مجدد توکن جدید و معتبر نیز در اختیار کاربر قرار می‌دهد. از این توکن تحت عنوان *Refresh Token* یاد می‌شود. با ارسال این توکن به سرور، در صورت معتبر بودن، یک *Access Token* و همچنین *Refresh Token* جدید در اختیار کاربر قرار می‌گیرد.

مهیار نه تنها بسیار در امر زمان‌بندی بسیار نامنظم است، بلکه بسیار تنبل و پر مشغله نیز هست! به همین دلیل، تصمیم گرفته است که از شما در انجام این تمرین کمک بگیرد.

جزئیات پروژه

پروژه‌ی اولیه را از این لینک دانلود کنید.

▼ ساختار فایل‌های پروژه

```
jwt-auth
├── package.json
├── package-lock.json
├── public
│   └── index.html
├── README.md
├── server
│   ├── index.js
│   └── requests.rest
└── src
    ├── App.css
    ├── App.js
    ├── Components
    │   └── ...
    ├── Hooks
    │   └── useJWT.js
    ├── index.js
    ├── Pages
    │   ├── Login.jsx
    │   └── Profile.jsx
    ├── setupTests.js
    └── __tests__
        └── sample.test.js
```

▼ راه‌اندازی پروژه

- پروژه‌ی اولیه را دانلود و از حالت فشرده خارج کنید.
- با اجرای دستور `npm i` پکیج‌های موردنیاز را نصب کنید.
- با اجرای دستور `node server/index.js` سرور را اجرا کنید.
- با اجرای دستور `npm start` پروژه را اجرا کنید.

هوک useJWT

نیما از مهیار خواسته است که یک هوک برای این شیوه‌ی احراز هویت طراحی کند که بتوان از آن در پروژه‌های بعدی نیز استفاده کرد. جزییات این هوک که نامش `useJWT` است به‌شرح زیر است:

این هوک باید چهار تابع با نام‌های زیر برگرداند. در ادامه به جزئیات هر یک از آن‌ها خواهیم پرداخت:

```
const { login, logout, refreshToken, sendPostRequest } = useJWT()
```

نکته‌ای که نیما از مهیار خواسته است در پیاده‌سازی این توابع به خاطر داشته باشد، این است که بتوان از همه‌ی آن‌ها به شکل زیر استفاده کرد:

```
f(...args).then(data => {}).catch(err => {});
```

تابع login

این تابع به‌ترتیب دو آرگومان `email` و `password` دریافت می‌کند و آن‌ها را در قالب یک شی و متد *POST* به آدرس `http://127.0.0.1:4000/api/login` ارسال می‌کند. در صورتی که رمز و ایمیل کاربر درست وارد شده باشد، سرور یک شی به‌صورت زیر برمی‌گرداند:

```
{
    "data": {
        "access": "ACCESS_TOKEN",
        "refresh": "REFRESH_TOKEN"
    }
}
```

همچنین مقادیر `access` و `refresh` باید به‌صورت مجزا و **با همان نام** در فضای محلی مرورگر (`localStorage`) ذخیره شوند (مقدار `access` در کلید `access` و مقدار `refresh` در کلید `refresh` ذخیره شود).

همچنین توکن‌های دریافت شده را باید در قالب یک شی به‌صورت زیر برگرداند:

```
{
    access: "ACCESS_TOKEN",
    refresh: "REFRESH_TOKEN"
}
```

تابع `refreshToken`

این تابع هیچ آرگومانی دریافت نمی‌کند. وظیفه‌ی اصلی این تابع، ارسال *Refresh Token* ذخیره‌شده در مرورگر در قالب بدنه‌ی یک درخواست *POST* به آدرس `http://127.0.0.1:4000/api/token` است. در صورتی که توکن معتبر باشد، سرور یک شی به‌صورت زیر برمی‌گرداند:

```
{
    data: {
        "access": "ACCESS_TOKEN",
        "refresh": "REFRESH_TOKEN"
    }
}
```

مقادیر `access` و `refresh` به صورت مجزا و **با همان نام** باید در فضای محلی مرورگر (`localStorage`) ذخیره شوند. همچنین بدنه درخواست ارسالی باید به صورت زیر باشد:

```
{
    "token": "REFRESH_TOKEN"
}
```

علاوه بر آن، این تابع باید توکن‌های دریافت‌شده را در قالب یک شی به‌صورت زیر برگرداند:

```
{
    access: "ACCESS_TOKEN",
    refresh: "REFRESH_TOKEN"
}
```

تابع `logout`

این تابع هیچ آرگومانی دریافت نمی‌کند. وظیفه‌ی اصلی این تابع، حذف *Access Token* و *Refresh Token* از `localStorage` است. همچنین این تابع باید مقدار *Refresh Token* ذخیره‌شده در `localStorage` را در قالب بدنه‌ی یک درخواست از نوع *DELETE* به آدرس `http://127.0.0.1:4000/api/logout` (در قالب کلیدی تحت عنوان `token`) ارسال کند:

```
1 | const logout = () => { /* TODO: Implement */ }
```

تضمین می‌شود که در این بخش همواره کد پاسخ برابر با `204` است.

تابع `sendPostRequest`

وظیفه‌ی اصلی این تابع، ارسال یک درخواست از نوع *POST* به سرور است. این تابع به عنوان ورودی، ابتدا مسیر درخواست که از نوع رشته است را دریافت می‌کند و به عنوان آرگومان دوم، داده‌ای که قرار است در بدنه درخواست ارسال شود را دریافت می‌کند.

از آن‌جایی که این تابع به منظور راحت‌تر شدن ارسال درخواست به مسیرهای که نیاز به توکن دارند طراحی شده است، دو هدف اصلی را دنبال می‌کند:

- در هدرهای ارسالی باید *Access Token* با کلید `jwt` در هدر قرار بگیرد.
- در صورتی که درخواست اول به دلیل منقضی شدن *Access Token* ناموفق شود، از سرور درخواست یک توکن جدید می‌کند و درخواست اصلی را مجدداً با توکن جدید ارسال می‌کند. لازم به ذکر است این عمل *تنها یک بار* صورت می‌گیرد (در صورتی که در دفعه‌ی دوم هم درخواست ناموفق باشد، نباید خطایی پرتاب شود یا کار دیگری صورت گیرد).
- در صورتی که درخواست موفقیت‌آمیز باشد، نتیجه‌ی درخواست را برمی‌گرداند.

کامپوننت `src/Pages/Login.jsx`

این کامپوننت یک صفحه‌ی ورود است. بخش `return` که بیانگر ساختار صفحه است از قبل نوشته شده است، اما موارد زیر باید تکمیل شوند:

- هنگام فشرده شدن `LoginButton` ، ایمیل و رمز عبور با استفاده از متد مربوطه‌ی هوک طراحی‌شده به سرور ارسال شوند.
- درصورت ورود موفق کاربر، هدایت به آدرس `/` صورت گیرد.
- در صورتی که احراز هویت ناموفق باشد، باید دو کامپوننت `NoRobot` و `ErrorMessage` نمایش داده شوند. همچنین مقدار ورودی `password` باید خالی باشد اما مقدار ورودی `email` همان مقدار قبلی باشد.
- دکمه‌ی `LoginButton` باید در صورتی فعال باشد که ایمیل و رمز عبور هر دو وارد شده باشند. همچنین ایمیل واردشده باید طبق فرمت `user@test.com` باشد و تیک مربوط به `NoRobot` نیز باید فعال باشد.

کامپوننت `src/Pages/Profile.jsx`

- هنگام بارگذاری این کامپوننت، ابتدا باید بررسی شود که کاربر توکنی دارد یا خیر. در صورت داشتن توکن، یک درخواست به آدرس `http://127.0.0.1:4000/api/user` ارسال شود در غیر این صورت کاربر به صفحه `/login` هدایت شود. پاسخ سرور به‌صورت زیر خواهد بود:

```
{
    data: {
        "user": {...values}
    }
}
```

در این‌صورت، شما باید مقدار استیت `user` را برابر با `data` دریافت شده قرار دهید. در غیر این‌صورت (یعنی اگر درخواست ناموفق بود)، کاربر باید به آدرس `/login` هدایت شود

- همچنین در صورت فشرده شدن دکمه‌ی `LogoutButton` باید روند خروج کاربر صورت پذیرد و کاربر به آدرس `/login` هدایت شود.

نکات

- در این پروژه از *React Router* با نسخه `v6` یعنی جدیدترین نسخه، استفاده شده است.

- آدرس درخواست‌ها باید دقیقاً مطابق با آدرس‌های ذکرشده در صورت سؤال باشد.
- داده‌ها باید به‌صورت *JSON* به سرور ارسال شوند.
- برای انجام این پروژه میتوانید از `axios` استفاده کنید.
- شما تنها مجاز به اعمال تغییرات در فایل‌های `useJWT.js` ، `Login.jsx` و `Profile.jsx` هستید.

آن‌چه باید آپلود کنید

پس از پیاده‌سازی موارد خواسته‌شده، پوشه‌ی `src` را زیپ کرده و آپلود کنید.