GeekBrains

ReactJS. Базовый курс

Знакомство с Firebase

[React 17.0.1]

На этом уроке

1. Узнаем о firebase и создадим проект в нем.
2. Научимся подключать авторизацию через email и пароль.
3. Научимся работать с RealtimeDB для сохранения чатов и сообщений.

Оглавление

Теория урока

Firebase. Создание проекта. Настройка и подключение

Firebase - продукт Google, который предоставляет множество возможностей, полезных для разработки веб- и мобильных приложений. В число предоставляемых firebase сервисов входят авторизация, базы данных (NoSQL), аналитика и многие другие. Кроме того, google предоставляет удобную в использовании SDK для веб-приложений.

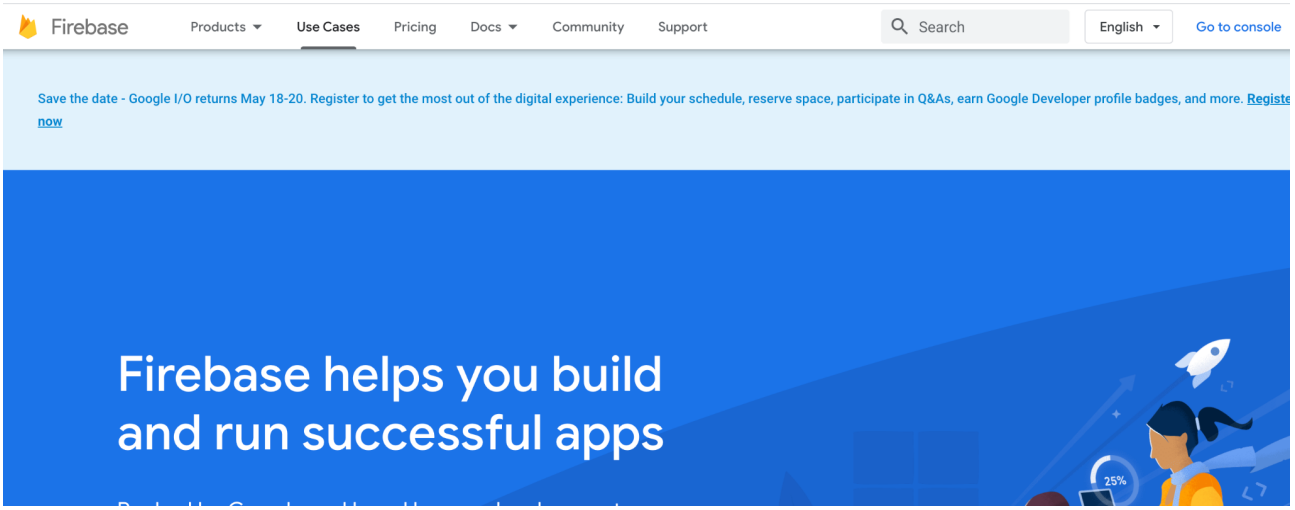
Для работы с firebase необходим аккаунт google.

При подключении сервисов firebase необходимо создать т.н. проект firebase и подключить его к своему приложению (эти шаги подробно рассмотрены в следующих разделах). Далее под проектом будет пониматься проект firebase - т.е. то, что создается и редактируется нв консоли разработчика на сайте firebase.google.com, а под приложением - разрабатываемое приложение на React.

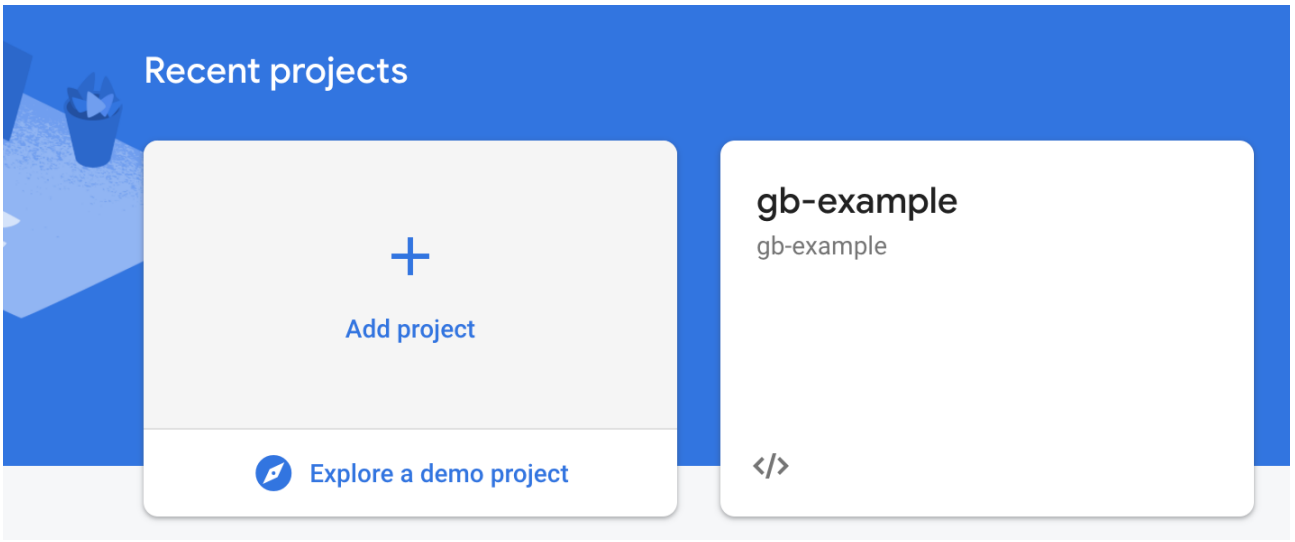
Настройка проекта Firebase в консоли разработчика

Для подключения возможностей firebase к своему приложению необходимо сделать следующее:

1. На странице firebase.google.com перейти в консоль разработчика



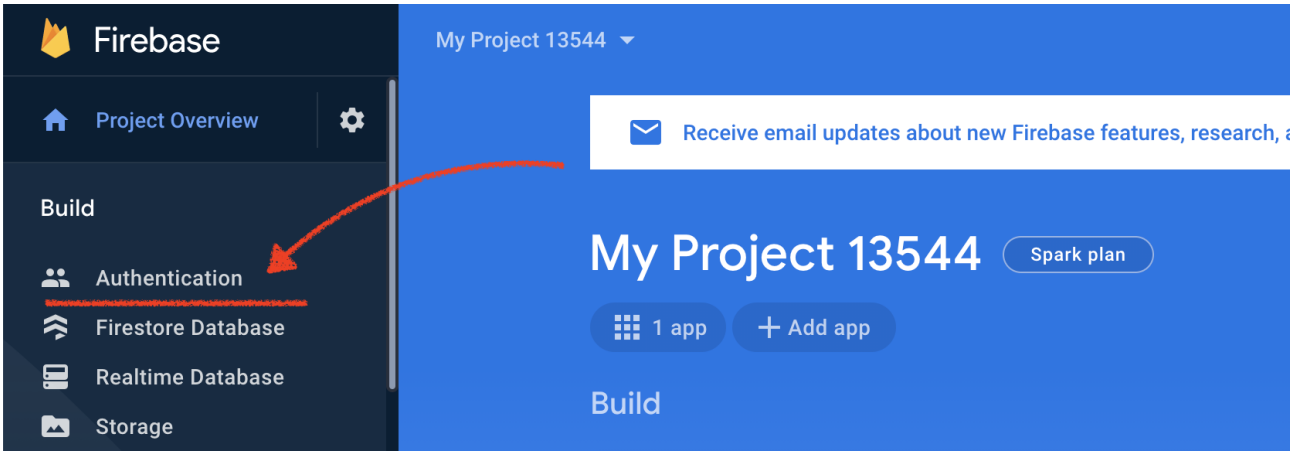
2. Создать новый проект (или выбрать существующий)



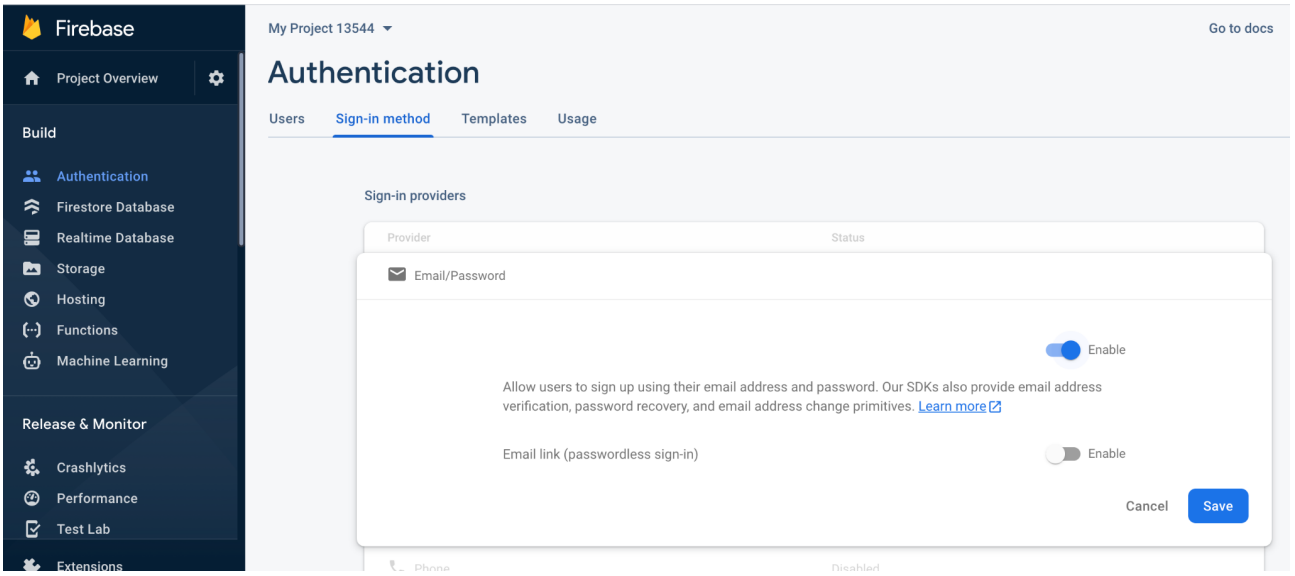
При создании нового проекта будет необходимо пройти несколько шагов:

a) Ввести название нового проекта
b) Включить/отключить сбор данных для аналитики (на данном этапе рекомендуется отключить аналитику, при необходимости ее можно включить позже)

3. После завершения процесса создания проекта firebase необходимо включить в данной консоли сервисы, которые необходимы нашему приложению. В данном случае мы будем использовать аутентификацию по email и паролю, а также базу данных Realtime Database (подробнее об этих сервисах в следующих разделах). Сперва перейдем в раздел Authetication:

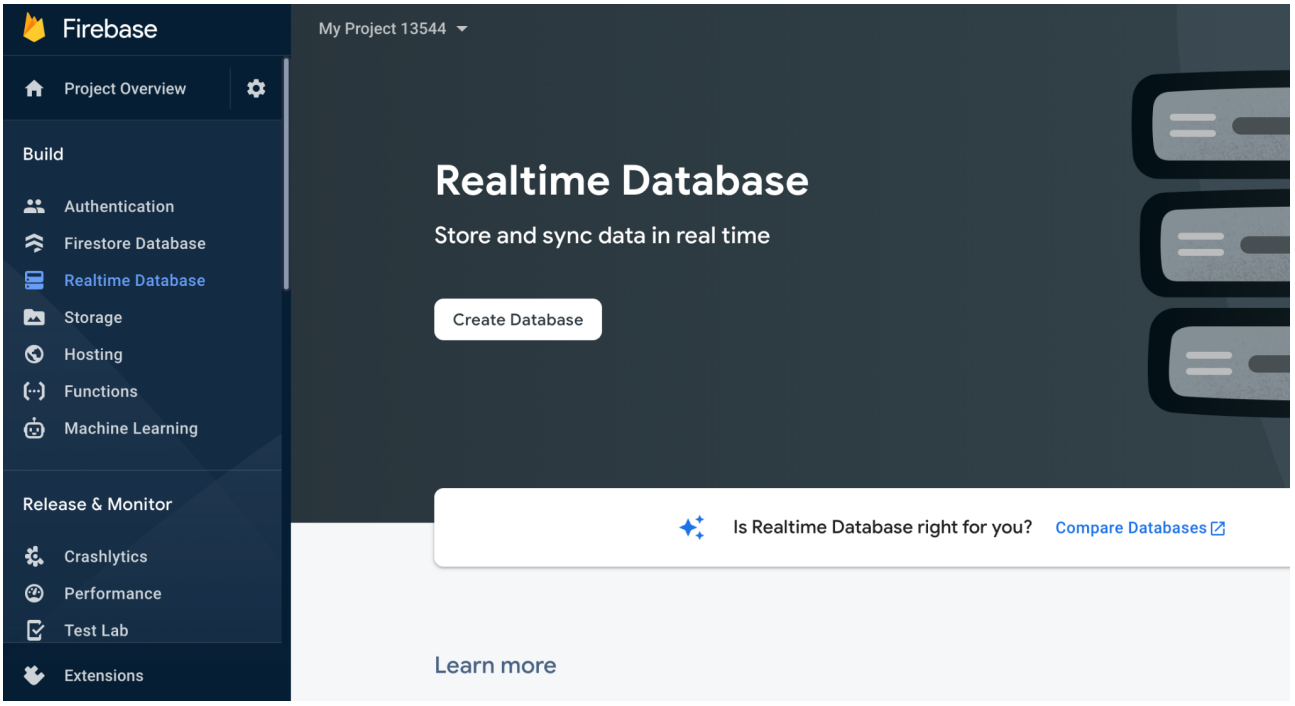


И активируем сервис, нажав Get Started. После этого перейдем на вкладку Sign-in method и выберем пункт Email/Password:



Установим флаг Enable.

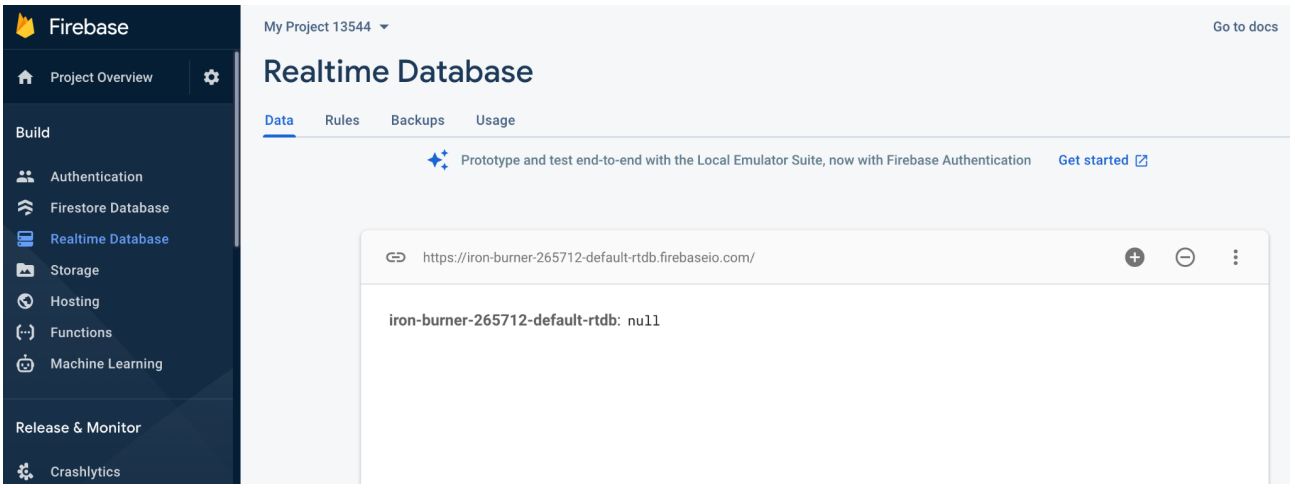
Аналогично активируем базу данных. Перейдем на вкладку Realtime Database:



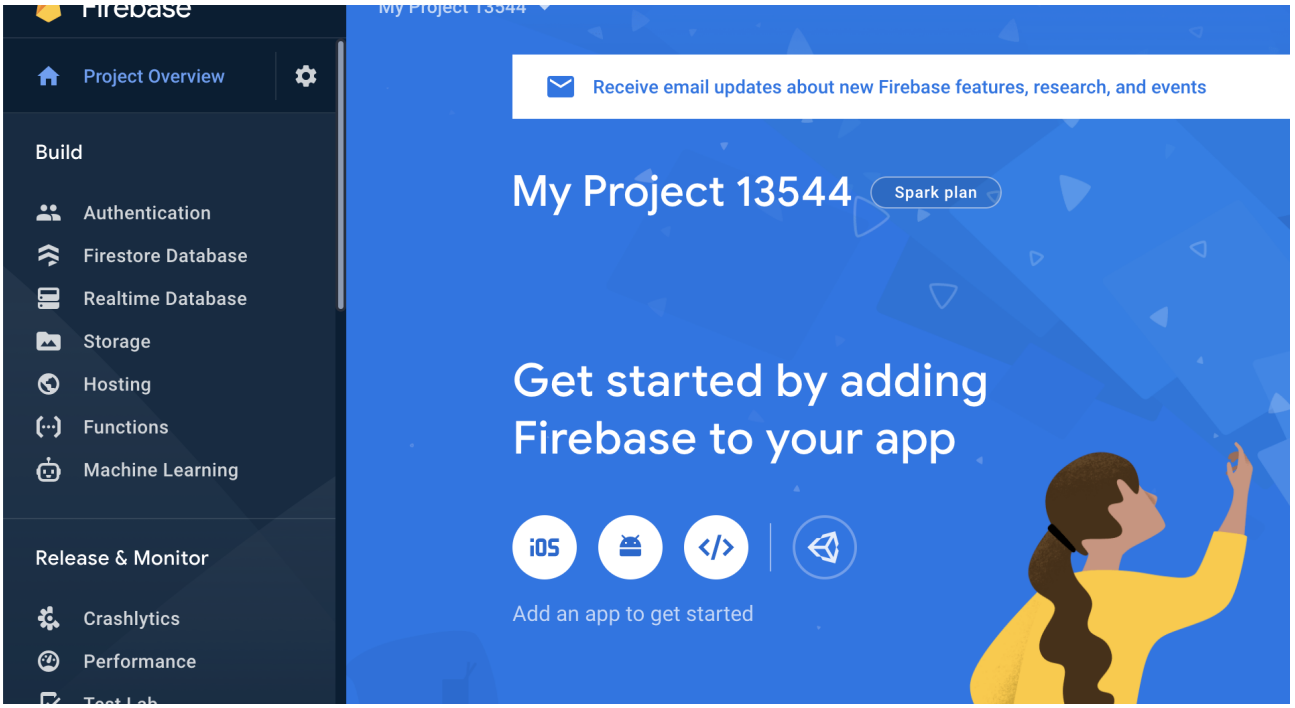
Нажмем Create Database, после чего необходимо выбрать локацию для хранения данных и правила доступа к базе:

На данном этапе можно оставить настройки по умолчанию.

После нажатия Enable создана пустая база данных. Она пока доступна для редактирования только из консоли разработчика:



4. После того как проект создан и настроен, требуется “соединить” его с нашим приложением - для этого в коде необходимо указать данные и ключи проекта. Перейдем на вкладку Project Overview и выберем “Добавить веб-приложение”:



Необходимо ввести название вашего приложения (требуется для идентификации вашего приложения в консоли разработчика). Firebase сформирует и отобразит конфигурационный скрипт:

2 Add Firebase SDK

Copy and paste these scripts into the bottom of your <body> tag, but before you use any Firebase services:

```
<!-- The core Firebase JS SDK is always required and must be listed first -->
<script src="https://www.gstatic.com/firebasejs/8.4.1/firebase-app.js"></script

<!-- TODO: Add SDKs for Firebase products that you want to use
     https://firebase.google.com/docs/web/setup#available-libraries -->

<script>
  // Your web app's Firebase configuration
  var firebaseConfig = {
    apiKey: "AIzaSyDTBA7hv9c064cc7JIKJWJxnaJoO-EdaKo",
    authDomain: "iron-burner-265712.firebaseapp.com",
    databaseURL: "https://iron-burner-265712-default-rtdb.firebaseio.com",
    projectId: "iron-burner-265712",
    storageBucket: "iron-burner-265712.appspot.com",
    messagingSenderId: "481179213014",
    appId: "1:481179213014:web:7ed45c7601e26c37c9fd77"
  };
  // Initialize Firebase
  firebase.initializeApp(firebaseConfig);
</script>
```

Learn more about Firebase for web: Get Started, Web SDK API Reference, Samples

Continue to console

Скопируйте только объект firebaseConfig.

Подключение firebase к приложению

Внимание!

В данном уроке рассматривается работа с firebase SDK версии 8. На момент написания методички версия 9 находится на стадии бета-тестирования.

Сперва установим SDK для работы с firebase:

```
npm i --save firebase
```

Добавим файл services/firebase.js - в нем настроим и инициализируем firebase:

```
import firebase from "firebase";

const config = {
  apiKey: "AIzaSyDTBA7hv9c064cc7JIKJWJxnaJoO-EdaKo",
  authDomain: "iron-burner-265712.firebaseapp.com",
  databaseURL: "https://iron-burner-265712-default-rtdb.firebaseio.com",
  projectId: "iron-burner-265712",
  storageBucket: "iron-burner-265712.appspot.com",
  messagingSenderId: "481179213014",
  appId: "1:481179213014:web:7ed45c7601e26c37c9fd77",
};

firebase.initializeApp(config);
```

Инициализация нашего приложения - то есть “связывание” приложения с проектом firebase осуществляется с помощью функции firebase.initializeApp, которой передается объект настроек с ключами, url и т.п. - тот объект, который мы получили в консоли разработчика.

Теперь в приложении мы имеем возможность взаимодействовать с проектом - например, использовать аутентификацию.

Аутентификация в приложении с помощью firebase через email и пароль

Все функции и данные, связанные с аутентификацией, доступны из класса firebase.auth. Из него можно как получать статические поля и методы (напр., firebase.auth.GoogleAuthProvider(), служащий для создания экземпляра провайдера для аутентификации через аккаунт Google), так и использовать его экземпляр (напр., firebase.auth().currentUser, который возвращает текущего пользователя). Причем firebase.auth() - синглтон, т.е. все его вызовы будут возвращать ссылку на один и тот же объект.

Настраиваем доступ к страницам приложения

Добавим возможность регистрации и входа, причем страницы чата и профиля сделаем доступными только для авторизованных пользователей. Используем для этого следующий подход: все наши компоненты Route обернем в HOC, который будет проверять состояние аутентификации пользователя, и отображать соответствующий компонент:

src/hocs/PrivateRoute.js

```
import React from "react";
import { Route, Redirect } from "react-router-dom";

export default function PrivateRoute({ authenticated, ...rest }) {
  return authenticated ? (
    <Route {...rest} />
  ) : (
    <Redirect to={{ pathname: "/login" }} />
  );
}
```

src/hocs/PublicRoute.js

```
import React from "react";
import { Route, Redirect } from "react-router-dom";

export default function PublicRoute({ authenticated, ...rest }) {
  return !authenticated ? <Route {...rest} /> : <Redirect to="/chats" />;
}
```

Компонент PrivateRoute служит для обертки маршрутов, доступных только авторизованному пользователю. Он принимает проп authenticated и, в зависимости от него, возвращает либо исходный компонент, либо Redirect на страницу входа. Компонент PublicRoute работает аналогично, но отображает исходный компонент в случае, если пользователь **не** аутентифицирован.

Добавим теперь в список маршрутов страницы входа и регистрации, а на главной странице добавим ссылки на них:

src/components/routes.js

```
export const Routes = () => {

  return (
    <BrowserRouter>
      <Header />
      <ul>
        <li>
```

```
          <Link to="/chats">Chats</Link>
        </li>
        <li>
          <Link to="/profile">Profile</Link>
        </li>
        <li>
          <Link to="/gists">Gists</Link>
        </li>
        <li>
          <Link to="/signup">Registration</Link>
        </li>
        <li>
          <Link to="/login">Login</Link>
        </li>
      </ul>
      <ChatList />
      <Switch>
        <Route exact path="/">
          <Home />
        </Route>
        <Route exact path="/login">
          <Login />
        </Route>
        <Route exact path="/signup">
          <SignUp />
        </Route>
        <Route path="/chats/:chatId?">
          <App />
        </Route>
        <Route path="/profile">
          <Profile />
        </Route>
        <Route path="/gists">
          <GistsList />
        </Route>
      </Switch>
    </BrowserRouter>
  );
};
```

src/component/home.js

```
export const Home = () => (
  <>
    <h3>Home</h3>
    <div>
      <Link to="/login">Sign In</Link>
    </div>
    <div>
      <Link to="/signup">Sign Up</Link>
    </div>
```

```
  </>
);
```

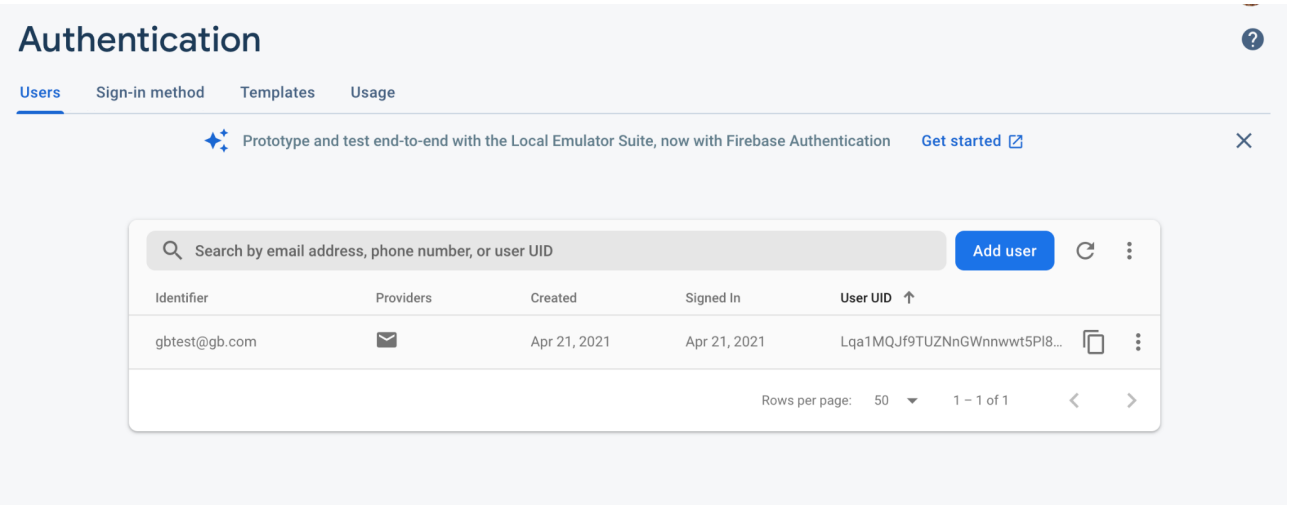
Добавляем авторизацию через firebase

Теперь создадим компонент для страницы регистрации:

```
export const Signup = () => {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [error, setError] = useState("");

  const handlePassChange = (e) => {
    setPassword(e.target.value);
  };

  const handleEmailChange = (e) => {
    setEmail(e.target.value);
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    setError("");

    try {
      await firebase.auth().createUserWithEmailAndPassword(email, password);
    } catch (error) {
      setError(error.message);
    }
  };

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <p>Fill in the form below to register new account.</p>
        <div>
          <input
            placeholder="Email"
            name="email"
            type="email"
            onChange={handleEmailChange}
            value={email}
          />
        </div>
        <div>
          <input
            placeholder="Password"
            name="password"
            onChange={handlePassChange}
            value={password}
```

```
            type="password"
          />
        </div>
        <div>
          {error && <p>{error}</p>}
          <button type="submit">Login</button>
        </div>
        <hr />
        <p>
          Already have an account? <Link to="/login">Sign in</Link>
        </p>
      </form>
    </div>
  );
```

Для регистрации нового пользователя используется функция из модуля auth библиотеки firebase - createUserWithEmailAndPassword. Обратите внимание, что функция асинхронная (“под капотом” - запрос к серверу), и может выбросить исключение. Здесь ее вызов аналогичен вызову fetch в предыдущем уроке - мы дожидаемся завершения ее работы (с помощью async/await), а также отлавливаем возможную ошибку. Сообщение об ошибке отображается в компоненте с помощью поля стейта error. В случае, если регистрация прошла успешно, в консоли разработчика firebase можно увидеть нового пользователя:



Аналогично создадим страницу входа - единственным отличием на данный момент будет являться то, что мы будем вызывать функцию signInWithEmailAndPassword:

```
export const Login = () => {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [error, setError] = useState("");

  const handlePassChange = (e) => {
    setPassword(e.target.value);
```

```
  };

  const handleEmailChange = (e) => {
    setEmail(e.target.value);
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    setError("");

    try {
      await firebase.auth().signInWithEmailAndPassword(email, password);
    } catch (error) {
      setError(error.message);
    }
  };

  return (
    <div>
      <form onSubmit={handleSubmit}>
        <p>Fill in the form below to login to your account.</p>
        <div>
          <input
            placeholder="Email"
            name="email"
            type="email"
            onChange={handleEmailChange}
            value={email}
          />
        </div>
        <div>
          <input
            placeholder="Password"
            name="password"
            onChange={handlePassChange}
            value={password}
            type="password"
          />
        </div>
        <div>
          {error && <p>{error}</p>}
          <button type="submit">Login</button>
        </div>
        <hr />
        <p>
          Don't have an account? <Link to="/signup">Sign up</Link>
        </p>
      </form>
    </div>
  );
};
```

Обратите внимание, что в этих компонентах мы не используем результаты вызовов функций createUserWithEmailAndPassword и signInWithEmailAndPassword - не сохраняем возвращенного пользователя, токен, и т.п. Все это за нас “под капотом” делает firebase - нам же осталось только добавить отслеживание состояния аутентификации. Для этого в компоненте Routes добавим следующую логику:

```
export const Routes = () => {
  const [authed, setAuthed] = useState(false);

  useEffect(() => {
    firebase.auth().onAuthStateChanged((user) => {
      if (user) {
        setAuthed(true);
      } else {
        setAuthed(false);
      }
    })
  }, []);

  return (
    <BrowserRouter>
      <Header />
      <ul>
        <li>
          <Link to="/chats">Chats</Link>
        </li>
        <li>
          <Link to="/profile">Profile</Link>
        </li>
        <li>
          <Link to="/gists">Gists</Link>
        </li>
        <li>
          <Link to="/signup">Registration</Link>
        </li>
        <li>
          <Link to="/login">Login</Link>
        </li>
      </ul>
      <ChatList />
      <Switch>
        <PublicRoute authenticated={authed} exact path="/">
          <Home />
        </PublicRoute>
        <PublicRoute authenticated={authed} path="/login">
          <Login />
        </PublicRoute>
        <PublicRoute authenticated={authed} path="/signup">
          <SignUp />
        </PublicRoute>
        <PrivateRoute authenticated={authed} path="/chats/:chatId?">
```

```
          <App />
        </PrivateRoute>
        <PrivateRoute authenticated={authed} path="/profile">
          <Profile />
        </PrivateRoute>
        <PublicRoute authenticated={authed} path="/gists">
          <GistsList />
        </PublicRoute>
      </Switch>
    </BrowserRouter>
  );
};
```

Здесь при монтировании компонента мы с помощью функции onAuthStateChanged добавляем подписку на состояние аутентификации - коллбэк, переданный этой функции, будет вызываться каждый раз, когда это состояние изменяется (т.е. когда пользователь регистрируется, либо выполняет вход или выход в приложении). Если пользователь аутентифицирован (зарегистрировался или выполнил вход) - коллбэк будет вызван с объектом user, содержащим данные о пользователе (его id, email и т.п.). Если пользователь вышел из приложения или удалил аккаунт - коллбэк будет вызван с аргументом null.

Благодаря этому мы можем выполнить проверку состояния аутентификации и соответствующим образом обновить состояние приложения - в данном случае просто изменяем переменную стейта authed. При его изменении компоненты PrivateRoute и PublicRoute получат обновленные пропсы - таким образом, компонент PublicRoute, получив значение пропса authed = true, вернет Redirect на страницу chats. Если же в какой-то момент состояние аутентификации вновь изменится (например, пользователь нажмет Sign Out), коллбэк функции onAuthStateChanged вновь выполнится с аргументом null, что вызовет обновление переменной authed, и любой компонент, обернутый в PrivateRoute, перенаправит пользователя на главную страницу.

Подключение Realtime Database для хранения чатов и сообщений

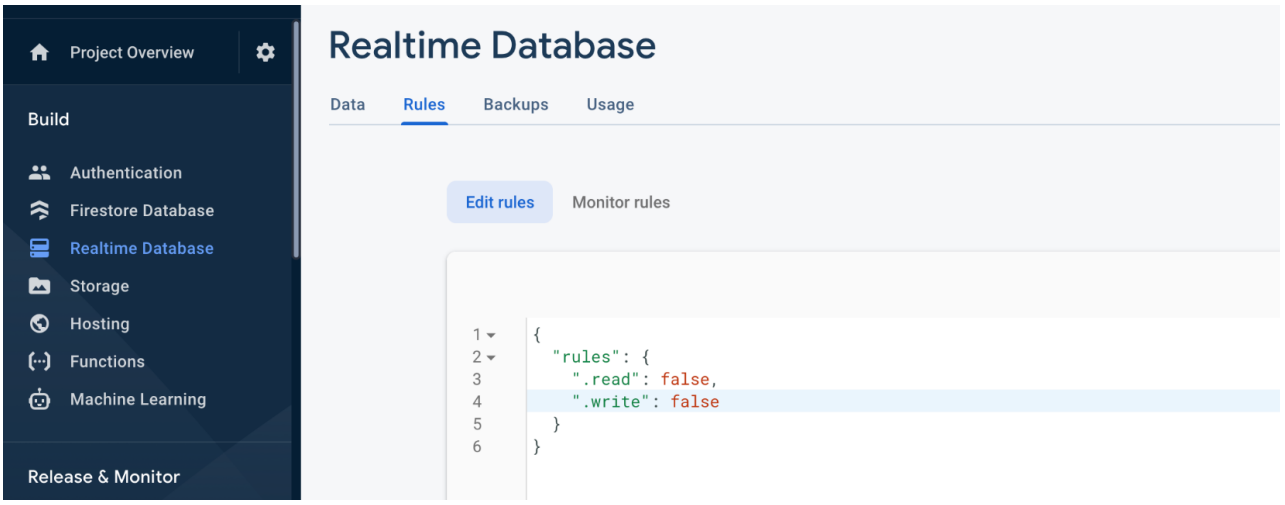
Работа Realtime Database основана на WebSocket - и в этом заключается одно из главных ее преимуществ. Это позволяет использовать получение данных в реальном времени - firebase автоматически уведомит ваше приложение, если данные на сервере изменены. В RealtimeDB можно записать только объект или примитив. При попытке записать в базу, например, массив, значение сохранено не будет.

Настройка правил доступа к базе данных

По умолчанию запись в базу и чтение из нее недоступны - т.к. в настройках при создании БД мы явно указали это:

То есть, при попытке получить данные из БД или записать в нее данные мы получим ошибку вида “Access denied”.

Чтобы этого избежать, настроим правила доступа. Перейдем на вкладку Rules раздела Realtime Database:



Здесь мы имеем возможность изменять и проверять правила, ограничивающие доступ к нашей БД. К примеру, такой набор правил:

```
{
  "rules": {
      ".read": true,
      ".write": true
  }
```

```
}
```

Разрешает доступ к базе любому пользователю, имеющему на нее ссылку. Такой способ не является безопасным - изменим его так, чтобы доступ к базе имели только аутентифицированные пользователи:

```
{
  "rules": {
      ".read": "auth != null",
      ".write": "auth != null"
  }
}
```

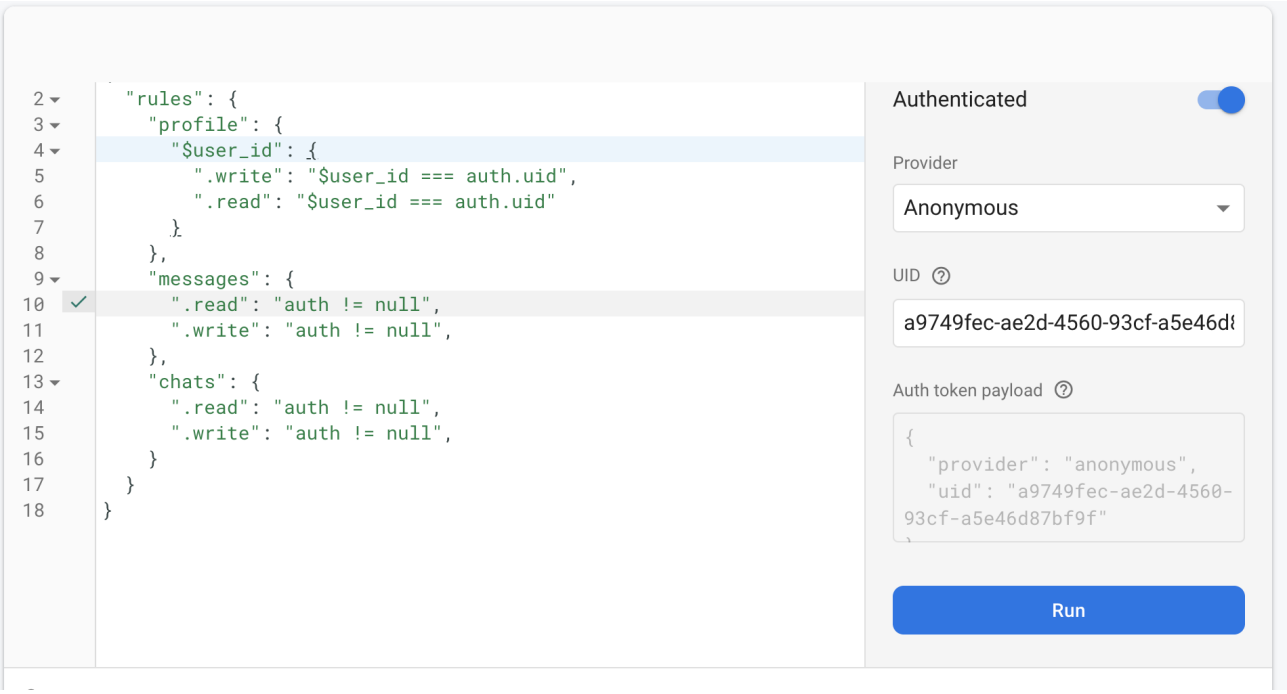
Здесь мы используем объект auth - если пользователь не аутентифицирован, он будет равен null. Для аутентифицированного пользователя в этом объекте будут содержаться его данные. К примеру, следующий набор правил разрешает пользователю доступ только к тем частям раздела profile в БД, в имени которых содержится id пользователя.

```
{
  "rules": {
    "profile": {
      "$user_id": {
        ".write": "$user_id === auth.uid",
        ".read": "$user_id === auth.uid"
      }
    },
    "messages": {
      ".read": "auth != null",
      ".write": "auth != null",
    },
    "chats": {
      ".read": "auth != null",
      ".write": "auth != null",
    }
  }
}
```

Таким образом, любой аутентифицированный пользователь будет иметь доступ на чтение и запись ко всем чатам и сообщениям, но только к своему профилю.

Объект, устанавливающий правила доступа к БД, имеет доступы и к другим переменным (напр., root - ссылка на саму базу данных, data - данные, хранящиеся в текущем разделе, newData - новые данные, которыми пользователь собирается обновить базу, и т.д.). Подробнее об использовании правил можно прочитать в документации.

Кроме того, нажав Rules Playground, можно запустить тестовый запрос к БД с настраиваемыми параметрами и проверить работу правил:



Подключаем Realtime Database в компоненте

Сперва рассмотрим использования базы данных на простом примере в компоненте, а затем перенесем работу с ней в мидлвар.

Рассмотрим компонент MessageFieldContainer:

```
const MessageFieldContainer = () => {
  const { chatId } = useParams();

  const chats = useSelector(getChatList);
  const messageList = useSelector((state) => state.messages.messageList);
  const dispatch = useDispatch();

  const onAddMessage = useCallback(
    (message) => {
      dispatch(addMessageWithThunk(chatId, message));
    },
    [chatId]
  );

  const onAddChat = useCallback((newChatName) => {
    dispatch(addChat(newChatName));
  }, []);

  if (!chatId) {
    return (
      <>
```

```
        <ChatList chats={chats} chatId={null} onAddChat={() => {}} />
      </>
    );
  }

  if (!chats[chatId]) {
    return <Redirect to="/nochat" />;
  }

  return (
    <MessageField
      chatId={chatId}
      messages={messageList[chatId]}
      onAddChat={onAddChat}
      onAddMessage={onAddMessage}
    />
  );
};
```

Он получает данные о сообщениях и чатах из стора. Временно изменим его так, чтобы сообщения хранились в массиве в стейте компонента:

```
const [messages, setMessages] = useState([]);
```

В эту переменную мы будем записывать данные, полученные из базы данных. А вместо отправки сообщения в стор будем отправлять его в ту же базу данных. Для работы с realtime database в библиотеке firebase используется модуль database:

```
export const db = firebase.database();
```

Этот модуль с помощью метода ref позволяет получить “живую ссылку” на указанную часть базы данных. Например, если в нашей БД есть раздел messages, мы можем получить ссылку на нее через db.ref("messages").

У объекта, возвращаемого firebase.database().ref, есть несколько методов, в числе которых - метод on. Данный метод позволяет осуществить подписку на некоторое событие - аналогично функции onAuthStateChanged модуля auth. Переданный методу on коллбэк будет выполняться каждый раз, когда происходит указанное событие:

```
db.ref("messages").on("value", (snapshot) => {
  const messages = [];
  snapshot.forEach((snap) => {
    messages.push(snap.val());
```

```
  });

  console.log(snapshot.key, messages);
});
```

Здесь мы отслеживаем событие value - то есть, любое изменение значения в базе данных. Коллбэк при любом изменении будет вызван с аргументом-снепшотом, “снимком состояния” базы данных после изменения. Снепшот - специальный объект, данные из которого получают вызовом метода val. Кроме этого, для снепшота доступен меотд forEach, который, аналогично forEach для массивов, перебирает вс записи в полученном снепшоте. В коллбэке forEach в примере выше получаем данные из “части” снепшота - одной записи в БД, также с помощью val(). Мы также можем получить ключ, по которому в баз хранится полученный снепшот, с помощью snapshot.key.

Мы можем отслеживать и другие события, например: ”child_added” - происходит, когда в БД появляется новая запись, “child_removed” - когда запись удаляется, “child_changed” - запись изменена (дочерняя запись удалена, добавлена новая и т.п.) Полный список событий представлен в документации.

Заметьте, что при такой записи:

```
db.ref("messages").on("child_changed", (snapshot) => {
   // ...
});
```

Мы отслеживаем изменение состояния **только** messages. Соответственно, при изменении другого раздела db - к примеру, chats, данный коллбэк вызван не будет.

Добавление записей происходит с помощью методов

1. push - добавление записи в выбранную часть БД, не изменяя существующие записи
2. set - полностью переписывает указанную часть БД

```
// добавляем сообщение в messages[chatId]
db.ref("messages").child(chatId).push(message);

// заменяем существующее messages[chatId][message.id]
// или создаем его, если его нет
db.ref("messages").child(chatId).child(message.id).set(messages);
```

Для добавления сообщения будем использовать метод set с указанием id сообщения.

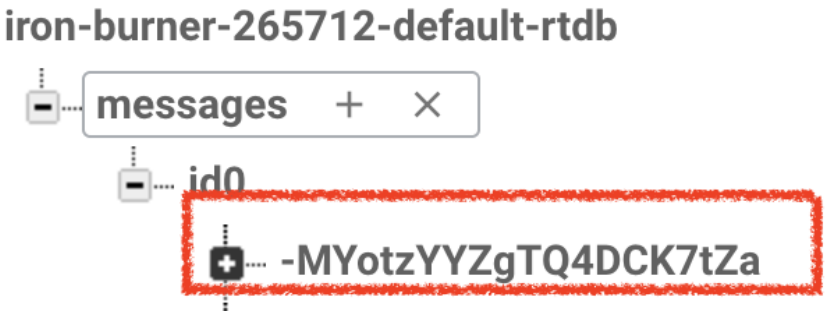
Тогда компонент MessageFieldContainer приобретает следующий вид:

```
export default function MessageFieldContainer() {
  const { chatId } = useParams();

  const [messages, setMessages] = useState([]);

  const onAddMessage = useCallback(
    (message) => {
      firebase.database()
         .ref("messages")
         .child(chatId)
         .child(message.id)
         .set(message);
    },
    [chatId]
  );

  useEffect(() => {
    firebase.database().ref("messages").child(chatId).on("value", (snapshot) =>
{
      const newMessages = [];

      snapshot.forEach(entry => {
        messages.push(entry.val());
      });

      setMessages(newMessages);
    });
  }, []);


  if (!chatId) {
    return (
      <>
        <ChatList chats={chats} chatId={null} onAddChat={() => {}} />
      </>
    );
  }

  if (!chats[chatId]) {
    return <Redirect to="/nochat" />;
  }

  return (
    <>
      <header>Header</header>
      <div className="wrapper">
        <div>
          <ChatList chatId={chatId} />
        </div>
        <div>
          <MessagesList messages={messages} />
          <Input onAddMessage={onAddMessage} />
        </div>
```

```
      </div>
    </>
  );
}
```

Внимание

При первом добавлении записей в БД, если такого раздела в базе не существует, он будет создан. Как было указано выше, массивы в realtime DB добавлять нельзя. При добавлении элемента в базу методом push этому элементу будет автоматически присвоен уникальный ключ:



При добавлении сообщения в БД практически мгновенно появляется новая запись - что приводит к вызову коллбэка, переданного нами методу on. Это, в свою очередь, вызывает обновление стейта компонента и его перерисовку - и новое сообщение отображается на странице без необходимости ее обновлять.

Переносим работу с Firebase в middleware

Итак, мы подключили аутентификацию и базу данных к компонентам. Однако теперь наши компоненты вновь перегружены логикой, в т.ч. асинхронными действиями. Исправим это, перенеся работу с firebase в миддлвар.

Сперва перенесем туда логику авторизации. Уберем стейт, отслеживающий ее состояние, из компонента, а также установку подписки. Вместо этого компонент будет получать эти данные из стора и на монтировании диспатчить экшен.

src/components/MessageField/MessageFieldContainer.js

```
export default function MessageFieldContainer() {
  const { chatId } = useParams();

  const chats = useSelector(getChatList);
  const messageList = useSelector((state) => state.messages.messages);
  const messages = messageList[chatId];
  const dispatch = useDispatch();
```

```
  const onAddMessage = useCallback(
    (message) => {
      dispatch(
        addMessageWithFirebase(chatId, {
          ...message,
          id: `${chatId}-${messages?.length || 0}-${Date.now()}`,
        })
      );

    },
    [chatId]
  );

  useEffect(() => {
    dispatch(initMessageTracking());
  }, []);

  return (
     // ...
  );
}
```

src/store/messages/reducer.js

```
import { CHANGE_MESSAGES } from "./types";

const initialState = {
  messages: {},
};

export const messagesReducer = (state = initialState, action) => {
  switch (action.type) {
    case CHANGE_MESSAGES: {
      return {
        ...state,
        messages: {
          ...state.messages,
          [action.payload.chatId]: action.payload.messages,
        },
      };
    }
    default:
      return state;
  }
};
```

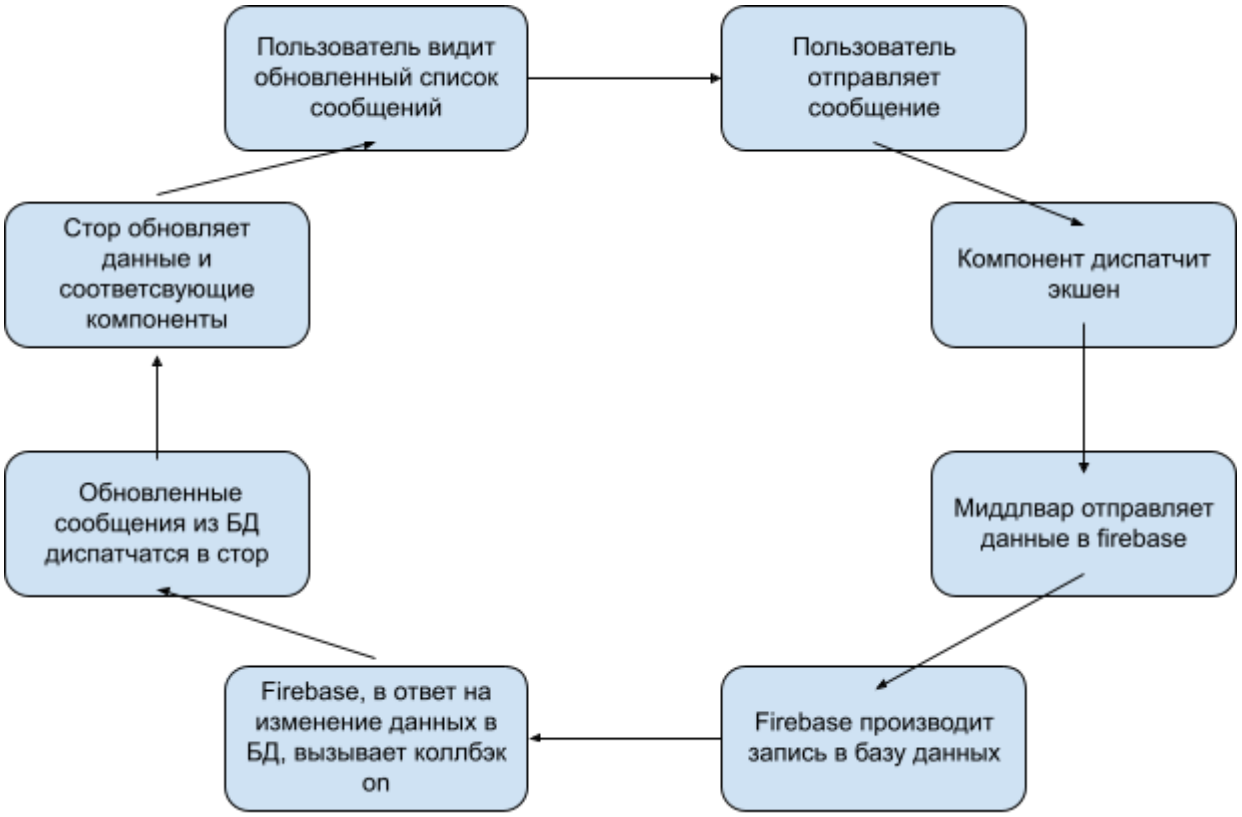
src/store/messages/actions.js

```
const getPayloadFromSnapshot = (snapshot) => {
  const messages = [];

  snapshot.forEach((mes) => {
    messages.push(mes.val());
  });

  return { chatId: snapshot.key, messages }
}

export const addMessageWithFirebase = (chatId, message) => async () => {
  db.ref("messages").child(chatId).child(message.id).set(message);
};

export const initMessageTracking = () => (dispatch) => {
  db.ref("messages").on("child_changed", (snapshot) => {
    const payload = getPayloadFromSnapshot(snapshot);
    dispatch({
      type: CHANGE_MESSAGES,
      payload,
    });
  });

  db.ref("messages").on("child_added", (snapshot) => {
    const payload = getPayloadFromSnapshot(snapshot);
    dispatch({
      type: CHANGE_MESSAGES,
      payload,
    });
  });
};
```

Как и ранее, при изменении состояния авторизации мы производим изменения состояния, но уже не компонента, а стора. Обратите внимание, что мы устанавливаем слушатели для двух событий - “child_added” и “child_changed” - это необходимо, так при добавлении первого сообщения событие child_changed не отработает, но отработает только событие child_added. С другой стороны, при изменении или удалении существующего сообщения не отработает событие child_added.

Аналогично можно логику отправки сообщения, получения сообщений, а также получения и изменения списка чатов.

После данных изменений схема работы нашего приложения выглядит следующим образом:



Пользователь изменяет данные - миддлвар отправляет данные в firebase - firebase проводит изменения БД - в ответ на изменения БД firebase вызывает коллбэк - данные из БД попадают в стор - стор обновляет соответствующие компоненты.

Удаление сообщений и чатов

Для удаления данных в RealtimeDB используется метод remove. Аналогично предыдущим примерам, вместо обработки изменения данных в редьюсере, в миддлваре будем вызывать методы firebase:

src/store/messages/actions.js

```
export const deleteMessageWithFirebase = (chatId, messageId) => async () => {
  db.ref("messages").child(chatId).child(messageId).remove();
};
```

Как можно заметить, теперь редьюсеры чатов и сообщений практически не содержат сложной логики - они просто принимают весь обновленный список и заменяют им существующий.

Внимание!

Очевидным недостатком данного подхода является необходимость полностью заменять весь массив сообщений, хранящийся в сторе (т.е., при получении снапшота обновленной БД приходится

перебирать весь список). При правильном использовании пропа key Реакт сможет оптимизировать рендер списка сообщений и чатов. Другие способы оптимизации приложения будут рассмотрены в следующем уроке.

Глоссарий

1. Firebase - продукт Google, набор сервисов для хранения данных, авторизации пользователей и др.
2. Realtime Database - сервис firebase, NoSQL-БД, получение и отправка данных работает на основе веб-сокет, что обеспечивает обновление данных в реальном времени.

Домашнее задание

1. Создать проект в консоли firebase. Установить и настроить в приложении firebase SDK.
2. Добавить аутентификацию через firebase (email/password).
3. Добавить отправку, хранение, получение сообщений и чатов через Realtime Database.
4. *Добавить в Realtime DB раздел profile. Сохранять в нем данные о профиле пользователя. Рядом с текстом сообщения пользователя отображать его имя или сохраненное пользователем имя.

Дополнительные материалы

1. Статья “Создание чата на Реакт с использованием Firebase”
2. Статья Private Route & Public Route components

Используемые источники

1. Firebase Realtime Database - официальная документация
2. Firebase Auth - официальная документация