GeekBrains

ReactJS. Базовый курс

Работа с API.

[React 17.0.1]

На этом уроке

1. Научимся работать с fetch в Реакт.
2. Познакомимся с подходами к обработке результата запроса, состояний ошибки и загрузки запроса.
3. Научимся выполнять запрос к API из redux-thunk.

Оглавление

Теория урока

Работа с API

API и клиент-серверное взаимодействие

API (application programming interface) - в широком понимании, описание способов, которыми одна программа будет взаимодействовать с другой. В контексте взаимодействия клиент-сервер, как

правило, под API понимают набор эндпоинтов (endpoint) - url, вызывая которые, клиент может передавать серверу или получать от него некоторые данные.

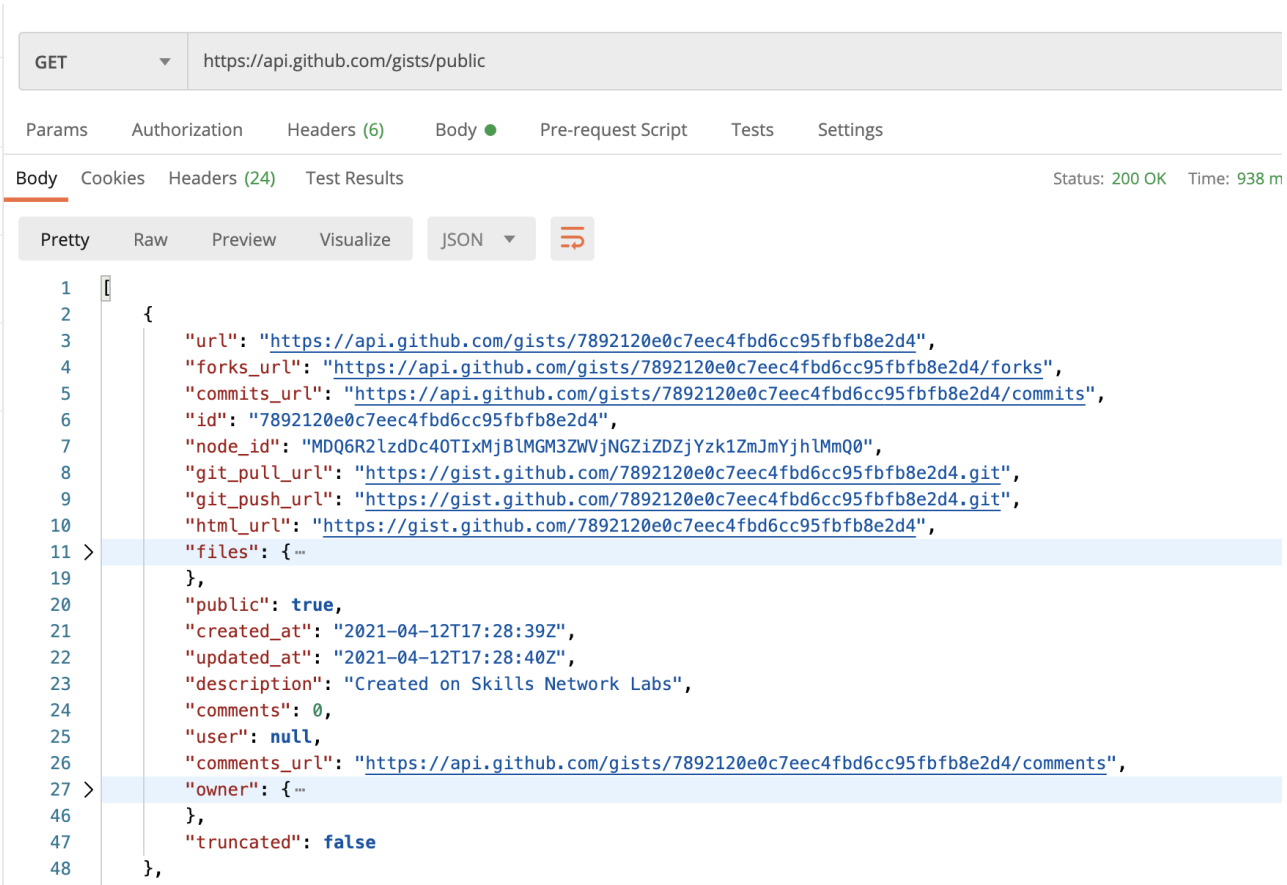
К примеру, github предоставляет публичный REST API со следующими эндпоинтами:

1. GET https://api.github.com/gists/ - возвращает список gists
2. POST https://api.github.com/gists/ - создает новый gist (требует аутентификации пользователя)
3. GET https://api.github.com/gists/public - возвращает список публичных gists
4. GET https://api.github.com/gists/{gistID} - возвращает данные об одном gist по переданному id

Полный список эндпоинтов можно увидеть в документации github.

Внимание!

Для работы с API, тестирования запросов и проверки ответов удобно использовать программу Postman. С ее помощью можно получить представление о том, в каком формате будут приходить данные от API. К примеру, результат GET *запроса на эндпоинт* https://api.github.com/gists/ *в Postman выглядит следующим образом:*



В данном проекте мы будем использовать GET https://api.github.com/gists/public и GET https://api.github.com/gists/{gistID} для получение списка публичных gists и данных о конкретном gist.

Следует также отметить, что существуют различные способы и протоколы взаимодействия клиент-сервер (например, websocket), но в данном курсе мы будем рассматривать только работу с https.

Кратко о Promise и fetch

JavaScript - однопоточный язык. Это значит, что в любой момент времени мы можем выполнять лишь одно действие, а если поток ждет результата вызова некоторой функции, то он оказывается заблокирован и не может выполнять другие действия параллельно. Для обхода этого ограничения движки JS (такие как V8 в Chrome) используют event loop (подробнее о нем можно прочитать здесь).

Для запуска асинхронных действий используют Promise (промис). Промис - специальный объект, который содержит свое состояние. При изменении состояния промис вызовет, по цепочке, коллбэки, переданные ему через специальные методы - then, catch и finally:

```
const p = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Hey!'), 1000);
});

p.then((value) => console.log(value + ' All good')) // выведет 'Hey! All good'
  .catch(err => console.warn(err))
  .finally(() => console.log('cleanup'));
```

Для отправки запроса используют функцию js, работа которой основана на промисах - fetch. Ее использование выглядит следующим образом:

```
// По умолчанию - GET
fetch('https://api.github.com/gists/public')
  .then((response) => {
    if (!response.ok) {
      throw new Error(`Request failed with status ${response.status}`);
    }

    return response.json();
  })
  .then((result) => console.log(result))
  .catch((err) => console.log(err));

// С помощью второго аргумента указываются опции запроса, напр., метод
fetch('https://api.github.com/gists/public', {
    method: 'POST',
    mode: 'cors',
    cache: 'no-cache',
    headers: {
      'Content-Type': 'application/json'
    }
  })
```

```
  .then((response) => {
    if (!response.ok) {
      throw new Error(`Request failed with status ${response.status}`);
    }

    return response.json();
  })
  .then((result) => console.log(result))
  .catch((err) => console.log(err));
```

Здесь в коллбэке then можно обрабатывать результаты запроса. Причем в первый коллбэк попадает полученный результат запроса - **но не полученные данные.** Для получения данных необходимо вызвать от результата запроса метод json (в случае, если мы ожидаем данные в формате json).

Также достаточно часто используется библиотека axios (построена на основе XMLHttpRequest), однако в данном курсе для запросов к серверу мы будем использовать fetch.

Получение данных от API из компонента

Добавим в наше приложение страницу, отображающую список gists, полученный от API github. На начальном этапе будем вызывать API и обрабатывать ответ в компоненте.

Сперва добавим несколько строковых констант для эндпоинтов:

```
export const API_URL_PUBLIC = "https://api.github.com/gists/public";
export const API_URL_GIST = "https://api.github.com/gists/";
```

Затем создадим новую страницу для отображения списка gists (пока в качестве данных используем пустой массив):

```
const gists = [];

export const GistsList = () => {

  const renderGist = useCallback(
    // gist.description может быть пустой строкой
    (gist) => <li key={gist.id}>{gist.description || 'No description'}</li>,
    []
  );

  return <ul>{gists.map(renderGist)}</ul>;
};
```

Добавим ее в роутер, а в хэдере добавим ссылку на нее:

```
export default function Router() {
  return (
    <BrowserRouter>
      <header>
        <ul>
          <li>
            <Link to="/profile">Profile</Link>
          </li>

          <li>
            <Link to="/chats">Chats</Link>
          </li>

          <li>
            <Link to="/">Home</Link>
          </li>

          <li>
            <Link to="/gists">Gists</Link>
          </li>
        </ul>
      </header>

      <Switch>
        <Route path="/profile">
          <Profile />
        </Route>

        <Route
          exact
          path="/chats"
          render={({ match }) => <Redirect to="/chats/id1" />}
        />

        <Route path="/chats/:chatId">
          <Chats />
        </Route>

        <Route path="/nochat">
          <NoChat />
        </Route>

        <Route path="/gists">
          <GistsList />
        </Route>

        <Route exact path="/">
          <Home />
        </Route>

        <Route>
```

```
          <h3>Page not found</h3>
        </Route>
      </Switch>
    </BrowserRouter>
  );
}
```

Создадим переменную стейта, и инициализируем ее пустым массивом:

```
const [gists, setGists] = useState([]);
```

Запрос данных из компонента выглядит следующим образом:

```
  useEffect(() => {
    fetch(API_URL_PUBLIC)
      .then((response) => response.json())
      .then((result) => setGists(result));
  }, []);
```

Обратите внимание, что установка полученных данных в стейт происходит в коллбэке then (причем сперва необходимо преобразовать ответ на запрос с помощью метода json, что также является асинхронной операцией).

Также следует помнить, что сетевые запросы - операция, подверженная ошибкам (к примеру, сервер может быть недоступен, соединение может быть слишком медленное и т.п.), поэтому необходимо поймать и обработать ошибку. Для этого будем использовать блок catch:

```
  useEffect(() => {
    fetch(API_URL_PUBLIC)
      .then((response) => {
        if (!response.ok){
          throw new Error(`Request failed with status ${response.status}`);
        }

        return response.json();
      })
      .then((result) => setGists(result))
      .catch((err) => console.log(err));
  }, []);
```

Обратите внимание, что в первом then мы проверяем поле ok объекта response. Это необходимо, так как даже в случае неудачного запроса fetch может не выбросить исключение (например, при ошибке 404).

На данный момент ошибка просто выводится в консоль. Однако, пользователь при этом об ошибке никак не оповещен - UI не изменяется.

Исправим это, добавив еще одну переменную стейта - error:

```
  const [error, setError] = useState(false);
```

И в блоке catch (т.е., в случае, когда запрос завершился неуспешно) будем устанавливать эту переменную. Перед началом нового запроса переменную error необходимо “сбросить” (иначе, в случае даже успешного повторного запроса, пользователю все равно будет отображено сообщение об ошибке). Кроме того, вынесем запрос в отдельный коллбэк для возможности удобного его перезапуска в случае ошибки, а на странице будем отображать соответствующее сообщение:

```
export const GistsList = () => {
  const [gists, setGists] = useState([]);
  const [error, setError] = useState(false);

  const requestGists = () => {
    setError(false);
    fetch(API_URL_PUBLIC)
      .then((response) => {
        if (!response.ok) {
          throw new Error(`Request failed with status ${response.status}`);
        }

        return response.json();
      })
      .then((result) => setGists(result))
      .catch((err) => {
        setError(true);
        console.log(err);
      });
  };

  useEffect(() => {
    requestGists();
  }, []);

  const renderGist = useCallback(
    (gist) => <li key={gist.id}>{gist.description}</li>,
    []
  );
```

```
  if (error) {
    return (
      <>
        <h3>Error</h3>
        <button onClick={requestGists}>Retry</button>
      </>
    );
  }

  return <ul>{gists.map(renderGist)}</ul>;
};
```

Также хорошим тоном является оповещение пользователя о том, что запрос еще идет (например, отображение спиннера или лоадера). Сделаем это с помощью новой переменной стейта isLoading:

```
export const GistsList = () => {
  const [gists, setGists] = useState([]);
  const [error, setError] = useState(false);

  const requestGists = () => {
    fetch(API_URL_PUBLIC)
      .then((response) => {
        if (!response.ok) {
          throw new Error(`Request failed with status ${response.status}`);
        }

        return response.json();
      })
      .then((result) => setGists(result))
      .catch((err) => {
        setError(true);
        console.log(err);
      });
  };

  useEffect(() => {
    requestGists();
  }, []);

  const renderGist = useCallback(
    (gist) => <li key={gist.id}>{gist.description}</li>,
    []
  );

  if (error) {
    return (
      <>
        <h3>Error</h3>
        <button onClick={requestGists}>Retry</button>
      </>
```

```
    );
  }

  return <ul>{gists.map(renderGist)}</ul>;
};
```

Она устанавливается в true перед началом запроса и в блоке finally - то есть, когда запрос завершен - сбрасывается в false.

В зависимости от значения этой переменной мы можем отображать на странице, к примеру, CircularProgress из Material UI.

```
const [loading, setLoading] = useState(false);

// ...

const requestGists = () => {
    setLoading(true);
    fetch(API_URL_PUBLIC)
      .then((response) => {
        if (!response.ok) {
          throw new Error(`Request failed with status ${response.status}`);
        }

        return response.json();
      })
      .then((result) => setGists(result))
      .catch((err) => {
        setError(true);
        console.log(err);
      })
      .finally(() => setLoading(false));
  };

// ...

 if (loading) {
    return <CircularProgress />;
  }

// ...
```

Этот пример можно переписать с использованием более современного синтаксиса - async/await:

```
  const requestGists = async () => {
    setLoading(true);

    try {
```

```
      const response = await fetch(API_URL_PUBLIC);

      if (!response.ok) {
        throw new Error(`Request failed with status ${response.status}`);
      }

      const result = await response.json();
      setGists(result);
    } catch (err) {
      setError(true);
      console.warn(err);
    } finally {
      setLoading(false);
    }
  };
```

Обратите внимание, что использование такого синтаксиса возможно, только если функция для отправки запроса вынесена в отдельный коллбэк - как упоминалось ранее, коллбэк useEffect нельзя объявлять асинхронным (см. Методичку 2).

Для проверки верности работы данной страницы мы можем воспользоваться вкладкой Network в инструментах разработчика - к примеру, эмулировать медленное соединение для проверки работы лоадера.

Обратите внимание, что запрос отправляется после первого рендера:

```
useEffect(() => {
  requestGists();
}, []);
```

Это рекомендованный авторами библиотеки способ выполнению асинхронных действий в компоненте (в классовом компоненте вызов этой функции находился бы в componentDidMount). Такой подход может показаться контринтуитивным - ведь, в некотором смысле, логичнее было бы выполнить его до первого рендера. Однако, на практике, первый рендер все равно завершится быстрее, чем сетевой запрос, и данные не будут отображены в любом случае. Вместе с этим, выполнение действий, связанных с побочными эффектами перед рендером противоречит механике Реакт.

Ну что ж, теперь у нас есть страница с загружаемыми данными. Аналогично, изменяя url API, можно загружать и отображать информацию из любого API (при условии, что он является открытым и возвращает данные в формате JSON). К примеру, список доступных API приведен здесь.

Однако, на данный момент все данные хранятся в компонентах. Перенесем их в стор.

Выполнение запросов из миддлвар

К примеру, при использовании redux-thunk мы можем изменить миддлвар из предыдущего примера следующим образом:

```
const GET_GISTS = 'GISTS::GET_GISTS';
const GET_GISTS_SUCCESS = 'GISTS::GET_GISTS_SUCCESS';

const getGists = () => ({
  type: GET_GISTS,
});

const getGistsSuccess = (gists) => ({
  type: GET_GISTS_SUCCESS,
  Payload: gists,
});

const getAllGists = () => async (dispatch, getState) => {
      const response = await fetch(API_URL_PUBLIC);
      const result = await response.json();
      dispatch(getGistsSuccess(result));
    }
}
```

Работая с запросами из компонента, мы также обрабатывали состояния загрузки и ошибки. Добавим эти обработки и здесь. Для начала создадим константу со статусами запроса:

```
export const STATUSES = {
  IDLE: 0,
  REQUEST: 1,
  SUCCESS: 2,
  FAILURE: 3,
}
```

Как правило, это делается с помощью создания нескольких экшенов для каждого запроса. Сперва создадим новый редьюсер:

```
const initialState = {
  gists: [],
  request: STATUSES.IDLE,
  error: null,
};

const gistsReducer = (state = initialState, action) => {
  switch (action.type) {
    default:
      return state;
  }
};
```

```
export default gistsReducer;
```

Теперь создадим типы экшенов и action creators:

```
export const GET_GISTS_REQUEST = "GISTS::GET_GISTS_REQUEST";
export const GET_GISTS_SUCCESS = "GISTS::GET_GISTS_SUCCESS";
export const GET_GISTS_FAILURE = "GISTS::GET_GISTS_FAILURE";

export const getGistsRequest = () => ({
  type: GET_GISTS_REQUEST,
});

export const getGistsSuccess = (data) => ({
  type: GET_GISTS_SUCCESS,
  payload: data,
});

export const getGistsFailure = (err) => ({
  type: GET_GISTS_FAILURE,
  payload: err,
});
```

Затем добавим их обработку в редьюсер (обратите внимание, что, как и ранее в компоненте, в редьюсере ошибка сбрасывается каждый раз при начале запроса):

```
const gistsReducer = (state = initialState, action) => {
  switch (action.type) {
    case GET_GISTS_REQUEST:
      return {
        ...state,
        request: STATUSES.REQUEST,
      };
    case GET_GISTS_SUCCESS:
      return {
        ...state,
        articles: action.payload,
        request: STATUSES.SUCCESS,
      };
    case GET_GISTS_FAILURE:
      return {
        ...state,
        request: STATUSES.FAILURE,
        error: action.payload,
      };
    default:
      return state;
```

```
  }
};
```

Будем диспатчить их из thunk'а:

```
export const getAllGists = () => async (dispatch) => {
  dispatch(getGistsRequest());

  try {
    const res = await fetch(API_URL_PUBLIC);

    if (!res.ok) {
      throw new Error(`Request failed with status ${res.status}`);
    }
    const result = await res.json();

    dispatch(getGistsSuccess(result));
  } catch (err) {
    dispatch(getGistsFailure(err.message));
  }
};
```

Наконец, создадим селекторы для получения данных о состоянии загрузки в компоненте

```
export const selectGists = (state) => state.gists.gists;
export const selectGistsError = (state) => state.gists.error;
export const selectGistsLoading = (state) => state.gists.loading;
```

Итак, когда компонент будет диспатчить в стор экшен getAllGists (являющийся thunk'ом), то redux-thunk вызовет получившуюся в результате вызова getAllGists функцию, и передаст ей аргументами dispatch и getState. Когда эта функция будет вызвана - нам необходимо начать запрос. Но прежде всего установим в сторе флаг isLoading в true - таким образом компонент сможет обработать состояние загрузки (к примеру, отобразив спиннер до момента окончания запроса). Затем выполним запрос и отправим в стор полученные данные. В случае ошибки сообщим об этом стору из блока catch (варианты обработки ошибок могут существенно различаться - здесь можно просто установить булево значение в сторе, указывающее на наличие ошибки, можно получить сообщение из объекта ошибки и передать его в стор, залогировать его и т.п.).

Компонент же, отправляющий этот запрос, приобретает следующий вид:

```
export const GistsList = () => {
  const dispatch = useDispatch();

  const gists = useSelector(selectGists);
```

```
  const error = useSelector(selectGistsError);
  const loading = useSelector(selectGistsLoading);

  const requestGists = () => {
    dispatch(getAllGists());
  };

  useEffect(() => {
    requestGists();
  }, []);

  const renderGist = useCallback(
    (gist) => <li key={gist.id}>{gist.description}</li>,
    []
  );

  if (loading) {
    return <CircularProgress />;
  }

  if (error) {
    return (
      <>
        <h3>Error</h3>
        <button onClick={requestGists}>Retry</button>
      </>
    );
  }

  return <ul>{gists.map(renderGist)}</ul>;
};
```

Здесь логика аналогична той, что представлена в примере раздела 1, но вместо вызова эндпоинта прямо из компонента мы диспатчим в стор экшен, в ответ на который запрос начнется из миддлвар. Соответственно, и данные для отображения будут получены не из стейта компонента, а из стора через селекторы.

Внимание!

Поскольку такой подход (4 экшена на 1 запрос - для начала запроса, начала загрузки, успешного ответа и ошибки загрузки) достаточно распространен, для создания однотипных экшенов иногда используются т.н. рутины - см. например redux-saga-routines.

Глоссарий

1. Promise (промис) - специальный объект, который содержит свое состояние. Используется для запуска асинхронных задач.
2. Fetch - функция, использующая промисы для выполнения сетевых запросов.

3. API (application programming interface) - в широком понимании, описание способов, которыми одна программа может взаимодействовать с другой. В данном уроке - эндпоинт или набор эндпоинтов, предоставляемых сервером для получения или отправки данных.
4. Endpoint (эндпоинт) - URL, отправляя запрос на который, можно взаимодействовать с сервером (отправлять или получать данные).

Домашнее задание

1. Добавить страницу, которая будет отображать данные, полученные через API (например, gists; можно воспользоваться любым работающим API из списка, название страницы в этом случае изменить соответствующим образом).
2. Добавить миддлвар для отправки запроса и обработки ответа, ошибки и состояния загрузки.
3. Отображать на странице данные из API или соответствующее состояние запроса. При ошибке пользователю должна быть доступна возможность отправить запрос заново без перезагрузки всей страницы.

Дополнительные материалы

1. Событийный цикл (event loop)
2. Axios
3. Redux-saga-routines

Используемые источники

1. Promise
2. Fetch
3. Статья об API
4. Запрос к API с React Hooks, HOC или Render Prop