6 - React (API)

React education, 2024.

MITHRIL

Overview

- Fetch
- Promises
- Async/Await

Fetch

Fetch

- Fetch API provides an interface for fetching resources and it is a modern alternative to XMLHttpRequest.

- *fetch()* method takes one mandatory argument, the path to the resource you want to fetch. It returns a Promise that resolves to the Response to that request — as soon as the server responds with headers — even if the server response is an HTTP error status.

Fetch

- On the top of the right image we see an example of GET request using fetch.
- The `json()` is a method of the body mixin that returns a *promise* that *resolves* with the result of parsing the body text as JSON.
- Because `json()` returns a *promise* we can chain another `then()` which logs the parsed response.

```
fetch("https://jsonplaceholder.typicode.com/posts")
  .then((response) => response.json())
  .then((json) => console.log(json));

// fetch("https://jsonplaceholder.typicode.com/posts", {
//   method: "POST",
//   body: JSON.stringify({
//     title: "foo",
//     body: "bar",
//     userId: 1,
//   }),
//   headers: {
//     "Content-type": "application/json; charset=UTF-8",
//   },
// })
//   .then((response) => response.json())
//   .then((json) => console.log(json));
```

MITHRIL

Fetch

- On the bottom of the image we see an example of POST request using fetch.
- It’s a bit more complex than GET.
- We have to specify a method “POST”
- Here we are using Content-type to specify which type of data we are sending to the server.
- Body defines data we are seeing, if we are sending JS object or value it must be wrapped with *JSON.stringify* method
- then() is also used to get an actual response from the server

```
// fetch("https://jsonplaceholder.typicode.com/posts")
//   .then((response) => response.json())
//   .then((json) => console.log(json));

fetch("https://jsonplaceholder.typicode.com/posts", {
  method: "POST",
  body: JSON.stringify({
    title: "foo",
    body: "bar",
    userId: 1,
  }),
  headers: {
    "Content-type": "application/json; charset=UTF-8",
  },
})
  .then((response) => response.json())
  .then((json) => console.log(json));
```

MITHRIL

Fetch

- fetch() allows us to make other types of requests as well like PATCH, PUT, DELETE and so on.
- We can provide the headers (most often Content-Type or the Authorization), body of a request in case of POST and PUT requests.

Promises

Promises

- Object that represent the eventual completion or failure of **asynchronous** operation, and its resulting value. Generally used for easier handling of asynchronous operations such as file operations, API calls, DB calls, IO calls…read more here.
- Promise can be created like it is shown on code below.
- The constructor accepts a function called executor which accepts two parameters resolve and reject. `resolve()` function should be executed when expected result is returned, and `reject()` in case when unexpected error occured.

```
new Promise( /* executor */ function(resolve, reject) {});
```

MITHRIL

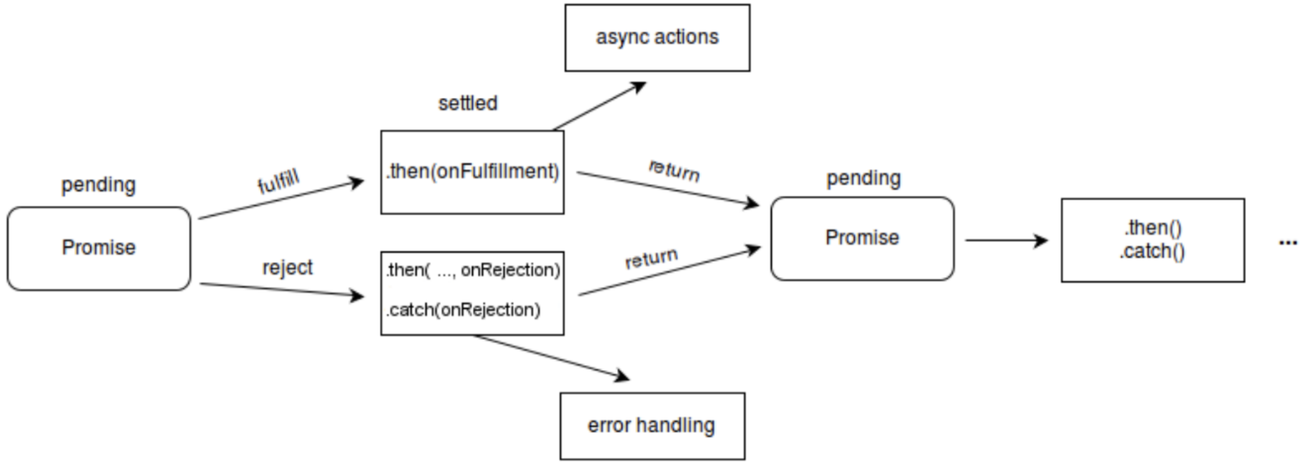
Promises

- In example on the right side, two promises are created. Each promise resolves after specific time expires.
- By use of `then()` we can handle the promise and get the result of execution. With `catch()` we can handle errors, and `finally()` is always executed.
- Promises also can be handled in the way that we want to wait until all promises resolve.
- Try this example to get better understanding.

```
// Promise 1
const promise1 = new Promise((resolve, reject) => {
  setTimeout(function(){
    resolve("Promise 1 finished.")
  }, 2000);
})

// Promise 2
const promise2 = new Promise((resolve, reject) => {
  setTimeout(function(){
    resolve("Promise 2 finished.")
  }, 5000);
})


// Handling of promise 1
promise1.then(result => console.log(result));

// Handling of promise 1 and 2 in paralell
Promise.all([promise1, promise2])
  .then(res => console.log(res))
  .catch(err => console.log(err))
  .finally(() => console.log('Always executed'))
```

Promises

- Three state of promises:
  - Pending - initial
  - Fulfilled - operation completed successfully
  - Rejected - operation failed



MITHRIL

Promise - static methods

- Most common static methods
  - *Promise.all*(iterable) - receives an array of promises and resolve when all promises are resolved of any is rejected
  - *Promise.allSettled*(iterable) - same as .all(), but wait until all promises are settled (resolved or rejected)
  - *Promise.race*(iterable) - receives an array of promises and waits until first promise is resolved or rejected
- Great read on understanding promises.

Async/await

Async/await

- `async` and `await` are extensions of promises.
- Asynchronous function operates asynchronously via *event loop*.
- `async` use an implicit Promise to return its result. Even if you don't return a promise explicitly, async function makes sure that your code is passed through a promise.
- `async` function can contain an await expression that pauses the execution of the async function and waits for the passed Promise's resolution, and then resumes the async function and returns the resolved value.
- `await` can only be used inside `async` function.

MITHRIL

Async/await

- Example explanation:
  - We've defined two async functions which return explicit promise, and our goal here is to create execution one after another.
  - In `returnTimeoutsAwait()` our async methods will be executed one after another, and inside `returnTimeoutsPromises()` will be executed paralell.
- Try this example.

```
async function timeout1(){
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve('Resolved 1');
      console.log('Async function 1 finished');
    }, 2000);
  })
}

async function timeout2(){
  return new Promise((resolve, reject) => {
    setTimeout(() => {
     resolve('Resolved 1');
    console.log('Async function 2 finished');
  }, 5000);
  })
}

async function returnTimeoutsAwait(){
  await timeout1();
  await timeout2();
}

async function returnTimeoutsPromises(){
  let promise1 = timeout1();
  let promise2 = timeout2();
  Promise.all([promise1, promise2])
   .then(() => console.log('Finished in paralel!'));
}

returnTimeoutsAwait();
returnTimeoutsPromises();
```

MITHRIL

Async/await vs. Promises

- These two concepts are similar but each has its own purpose.
  - Await blocks the execution of the code within the async function in which is located.
  - If the output of `function2` is depenedent on output of `function1` then use `await`.
  - If two function can be run in parallel create an array of promises and then use `Promise.all()`.
  - Everytime you use `await` remember that you are writing blocking code (avoid too much blocking code).
- Useful resources:
  - Promises vs async/await
  - Understanding async/await

Hands-on

Hands-on

1. Create new page called Posts.
2. Add this page to the Route in App.tsx.
3. Create new folder which will be used for services, called *services*
4. Inside that folder create file that will have API call to our api.
5. Create one more file that will use created service to get all posts from the https://jsonplaceholder.typicode.com/posts
6. Inside our component/posts page print all of the posts title using react hook.
7. Using CSS style the page that all posts are shown in a grid.