Promise, Async& Await, & Fetch

Rujuan Xing

Maharishi International University - Fairfield, Iowa

Introduction

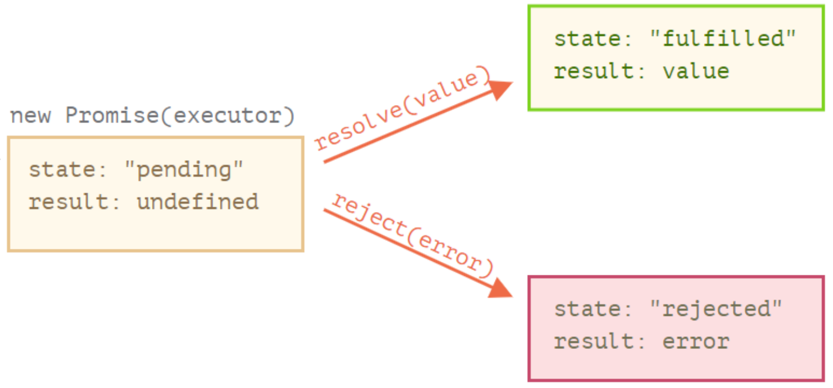
- JavaScript executes code in a single thread, which brings a risk of blocking the thread if a single line takes long time to process. This means until that line finishes, the next line of code won’t be processed.
- For example, handling of AJAX request should be done on a different thread, otherwise our main thread would be blocked until the network response is received.

What is a Promise?

- A promise is an object that represents something that will be available in the future. In programming, this "something" is values.
- Promises propose that instead of waiting for the value we want (e.g. the image download), we receive something that represents the value in that instant so that we can "get on with our lives" and then at some point go back and use the value generated by this promise.
- Promises are based on time events and have some states that classify these events:
  - Pending: still working, the result is undefined;
  - Fulfilled: when the promise returns the correct result, the result is a value.
  - Rejected: when the promise does not return the correct result, the result is an error object.

Create a Promise Object

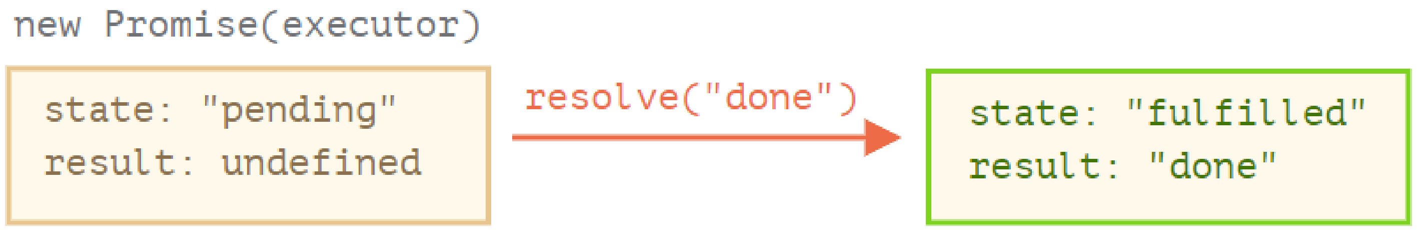
```
let promise = new Promise(function(resolve, reject) {
    // executor
});
```



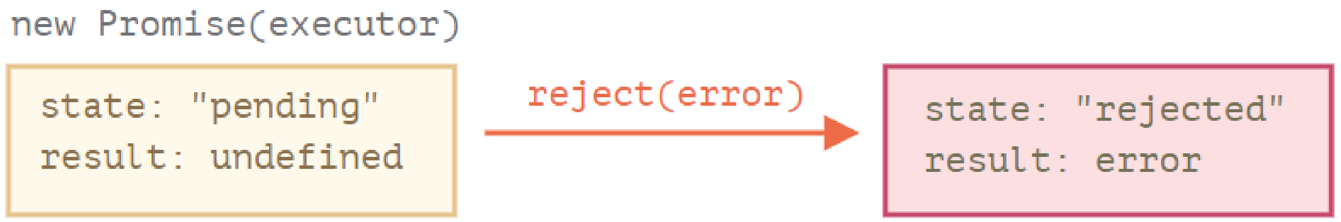
- The function passed to new Promise is called the **executor**. When new Promise is created, **the executor runs automatically**. Only the parts of resolve and reject are going to be asynchronous.
- Its arguments `resolve` and `reject` are callbacks provided by JavaScript itself.
- When the executor obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:
  - `resolve(value)` — if the job is finished successfully, with result `value`.
  - `reject(error)` — if an error has occurred, `error` is the error object.
- The promise object returned by the new Promise constructor has these internal properties:
  - `state` — initially "`pending`", then changes to either "`fulfilled`" when `resolve` is called or "`rejected`" when `reject` is called.
  - `result` — initially `undefined`, then changes to `value` when `resolve(value)` called or `error` when `reject(error)` is called.

Create a Promise Object (cont.)

```
let promise = new Promise(function(resolve, reject) {
    // the function is executed automatically when the promise is constructed
    // after 1 second signal that the job is done with the result "done"
    setTimeout(() => resolve("done"), 1000);
});
```



```
let promise = new Promise(function (resolve, reject) {
    // after 1 second signal that the job is finished with an error
    setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```



Consumers: `then`, `catch`, `finally`

- A Promise object serves as a link between the executor and the consuming functions, which will receive the result or error. Consuming functions can be registered (subscribed) using methods `.then`, `.catch` and `.finally`.

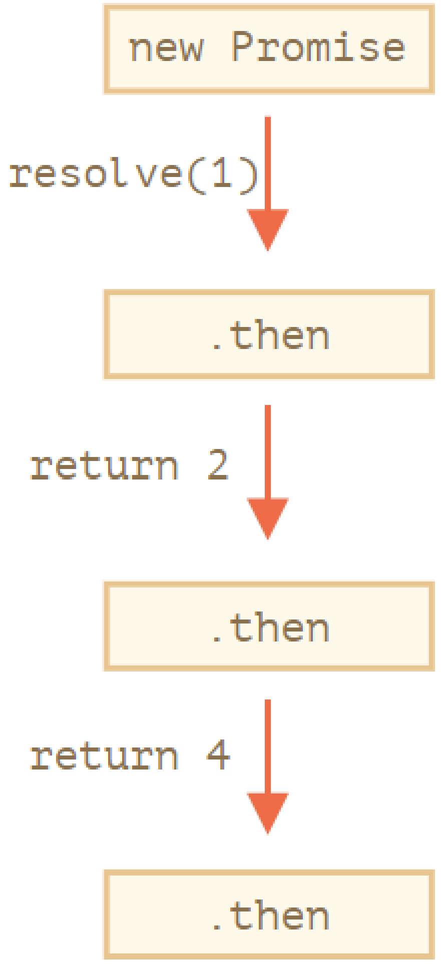
```
let promise = new Promise(function(resolve, reject) {
    const random = Math.random();
    console.log('random: ', random);
    if (random > 0.5) {
        setTimeout(() => resolve("done!"), 1000);
    } else {
        setTimeout(() => reject(new Error("Whoops!")), 1000);
    }

});

promise.then(result => console.log(result))
    .catch(error => console.log(error))
    .finally(() => console.log("Promise ready!"));
```

Promises chaining

```
new Promise(function(resolve, reject) {
    setTimeout(() => resolve(1), 1000); // (*)
}).then(function(result) { // (**)
    alert(result); // 1
    return result * 2;
}).then(function(result) { // (***)
    alert(result); // 2
    return result * 2;
}).then(function(result) {
    alert(result); // 4
    return result * 2;
});
```



Async/await

- It’s a special syntax to work with promises in a more comfortable fashion
- The `async` keyword: when you put `async` keyword in front of a function declaration, it turns the function into an `async` function.
- The `await` keyword: `await` only works inside `async` functions. `await` can be put in front of any `async` promise-based function to pause your code on that line until the `promise` fulfills, then return the resulting `value`.

Async functions

- `async` can be placed before a function. An `async` function always returns a `promise`:
  - When no return statement defined, or return without a value. It turns a resolving a promise equivalent to `return Promise.Resolve()`
  - When a return statement is defined with a value, it will return a resolving promise with the given return `value`, equivalent to `return Promise.Resolve(value)`
  - When an error is thrown, a rejected promised will be returned with the thrown error, equivalent to `return Promise.Reject(error)`

```
console.log('start');
async function f() {
    return 1;
}

f().then(console.log);
console.log('end');
```

Await

- The keyword `await` makes JavaScript wait until that `promise` settles and returns its result.
- `await` literally suspends the function execution until the `promise` settles, and then resumes it with the `promise` result. That doesn’t cost any CPU resources, because the JavaScript engine can do other jobs in the meantime: execute other scripts, handle events, etc.
- It’s just a more elegant syntax of getting the `promise` result than `promise.then`.

```
console.log('start');
async function foo() {
    return 'done!';
}
async function bar() {
    console.log('inside bar - start');
    let result = await foo();
    console.log(result); // "done!"
    console.log('inside bar - end');
}
bar();
console.log('end');
```

Await (cont.)

1. Can't use `await` in regular functions. If we try to use `await` in a non-async function, there would be a syntax error:

```
async function foo() {
    return 'done!';
}

function bar() {
    let result = await foo(); // Syntax error
    console.log(result);
}
bar();
```

2. `await` won't work in the top-level code

```
// syntax error in top-level code
async function baz() {
    return 'baz...';
}

let result = await baz(); //Syntax Error
console.log(result);
```

Error Handling in Async functions

1. Use `await` inside async function

2. Chain async function call with a `.catch()` call.

```
async function thisThrows() {
    throw new Error("Thrown from thisThrows()");
}

async function run() {
    try {
        await thisThrows();
    } catch (e) {
        console.log('Caught the error....');
        console.error(e);
    } finally {
        console.log('We do cleanup here');
    }
}

run();
```

```
async function thisThrows() {
    throw new Error("Thrown from thisThrows()");
}

thisThrows()
    .catch(console.error)
    .finally(() => console.log('We do cleanup here'));
```

Fetch API

- The Fetch API provides a JavaScript interface for accessing and manipulating parts of the HTTP pipeline, such as requests and responses. It also provides a **global** fetch() method that provides an easy, logical way to fetch resources **asynchronously** across the network.
- This kind of functionality was previously achieved using `XMLHttpRequest`.
- Syntax:

```
let promise = fetch(url, [options]);
```

  - `url` – the URL to access.
  - `options` – optional parameters: method, headers etc.
  - Without `options`, this is a simple `GET` request, downloading the contents of the url.

Using fetch() - GET

```
let promise = fetch('http://localhost:3000/products')
```

The browser starts the request right away and returns a promise that the calling code should use to get the result.

- Getting a response is usually a two-stage process.
  - First, the promise, returned by fetch, resolves with an object of the built-in Response class as soon as the server responds with headers.
    - `promise.then(response => console.log(response.ok, response.status));`
  - Second, to get the response body, we need to use an additional method call.
    - response.text() – read the response and return as text,
    - response.json() – parse the response as JSON,
    - …

```
promise.then(response => response.json())
        .then(result => console.log(result));
```

Using fetch() – POST with options

```
fetch('http://localhost:3000/products', {
        method: 'POST',
        body: JSON.stringify({
            title: 'cs445',
            description: 'This is CS445 Lesson 7 Demo',
            price: 100,
        }),
        headers: {
            'Content-type': 'application/json; charset=UTF-8',
        },
    })
    .then((response) => response.json())
    .then((json) => console.log(json));
```

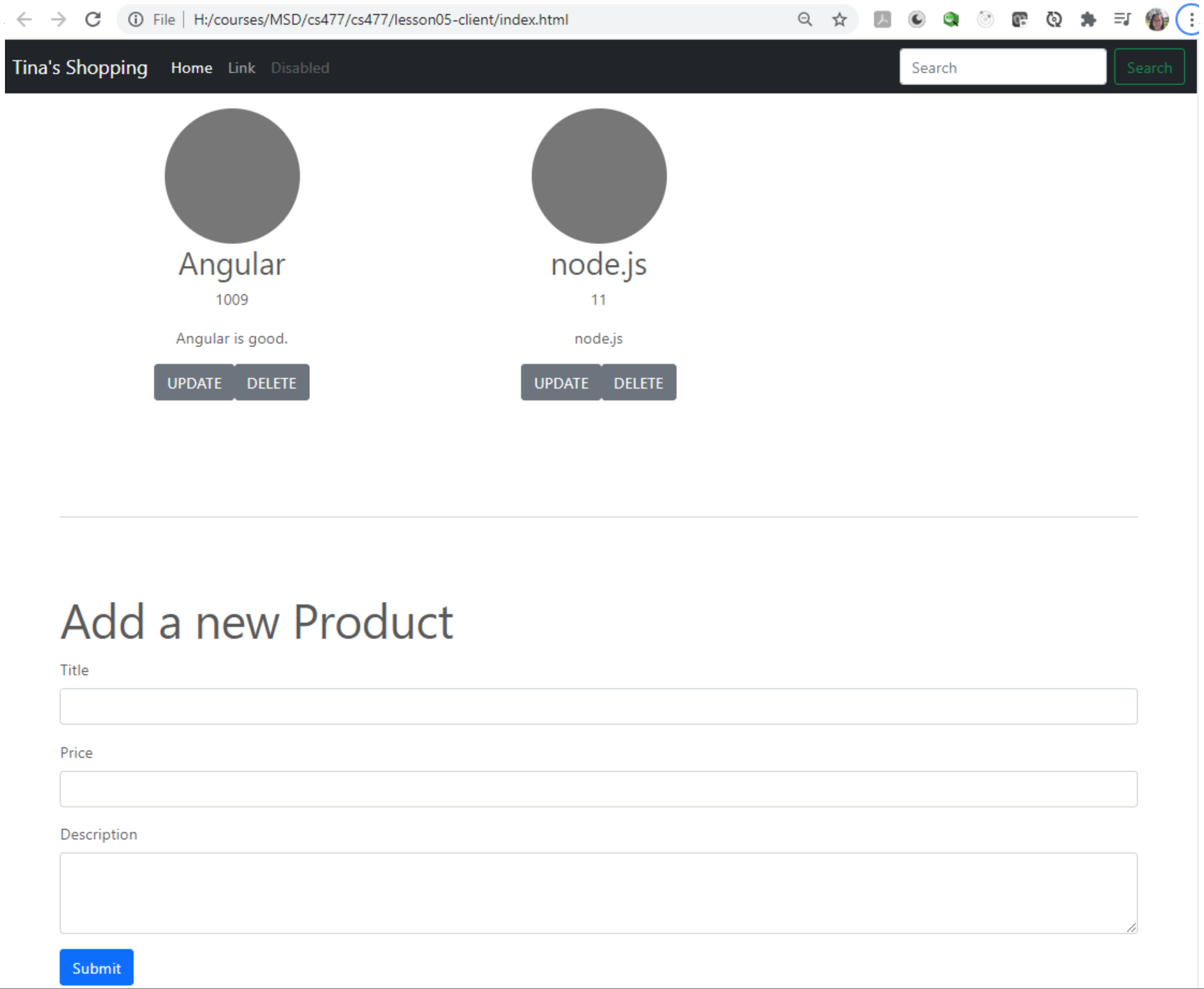
Using Async & Await

```
async function fetchProductById(id) {
    let response = await fetch("http://localhost:3000/products/" + id);
    if (response.ok) {
        let json = await response.json();
        console.log(json);
    } else {
        console.log("HTTP-Error: " + response.status);
    }
}

fetchProductById(1);

// Remember to use try/catch in case of network problem
```

Demo: Shopping Cart – Client side


File | H:/courses/MSD/cs477/cs477/lesson05-client/index.html

Tina's Shopping   Home   Link   Disabled   Search   Search

Angular
1009
Angular is good.
UPDATE DELETE

node.js
11
node.js
UPDATE DELETE

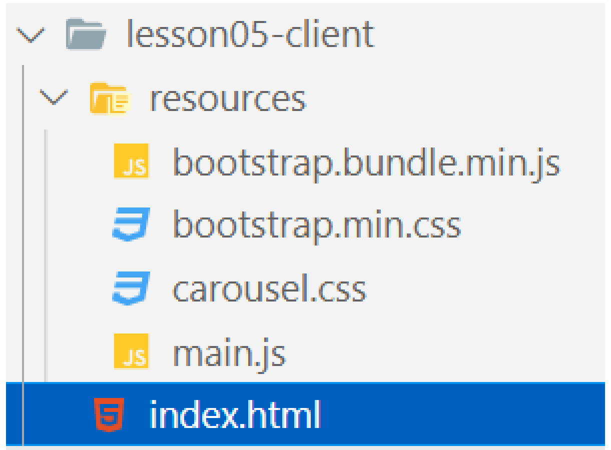
Add a new Product

Title

Price

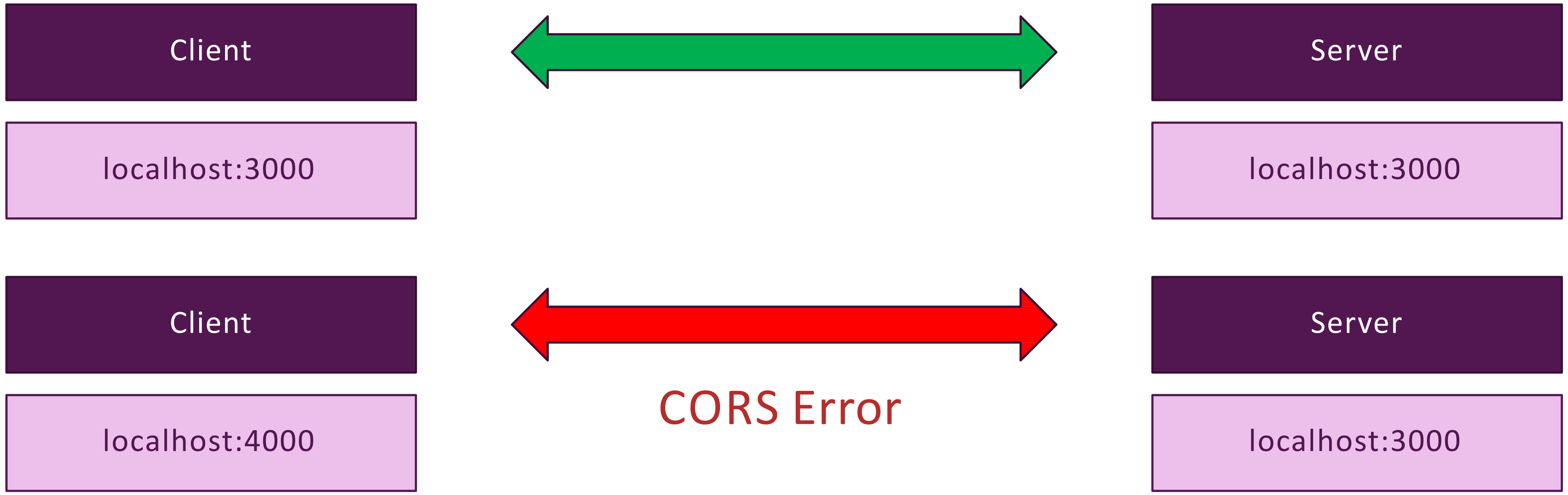
Description

Submit


lesson05-client
resources
bootstrap.bundle.min.js
bootstrap.min.css
carousel.css
main.js
index.html

CORS (Cross-Origin Resource Sharing)

- **Cross-Origin Resource Sharing** (CORS) is a mechanism that uses additional HTTP headers to tell browsers to give a web application running at one origin, access to selected resources from a different origin. A web application executes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, or port) from its own.



cors

- npm install cors
- **Simple Usage (Enable *All* CORS Requests)**
  - `const cors = require('cors');`
  - `app.use(cors());`
- **Enable CORS for a Single Route**
  - `router.post('/users', cors(), userController.insert);`