Adv Web Dev Arch
Lecture 6 (updated 2026)

how to implement
Microservices communication asynchronously
(intro to promises in JS)

Amir Amintabar, PhD

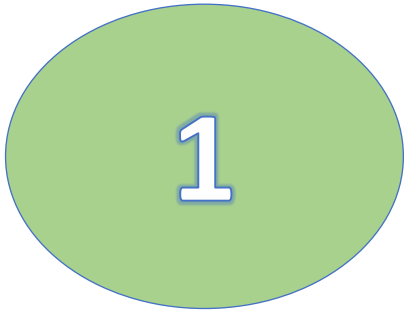
Outline

- 1 Review
- 2 Asynch operations
- 3 Promises in JS
- 4 Promise status
- 5 .catch() and .finally()
- 6. In class activity
- 7. remarks on **response size** of GET vs POST requests
- 8. W3Schools website seems inaccurate

- Quizzes on promise during labs, this week
- No new labs this week. Next lab will be assigned after the midterm
- Midterm exam administrated on Learning Hub

review

- myDomain.com/api/customers



- Every http request has a verb or method that determines its intention
- GET,PUT,DELETE,POST
- PUT: replacing an object entirely with something new
- PATCH: updating only a property of an object
- PUT vs POST
- PUT is idempotent, POST is not
- calling PUT once or several times successively has the same effect (that is no side effect), whereas successive identical POST requests may have additional effects, e.g. leading to placing an order several times.

Idempotent: can be applied multiple times without changing the result

Remember in JS can a function receive another function as parameter? first-class functions

- Example:

```
function foo(x) {
    alert(x);
}
function bar(func) {
    func("Hello World!");
}

//alerts "Hello World!"
bar(foo);
```

Remember Call backs

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}
function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}
myCalculator(5, 5, myDisplayer);
```

- Example above says
- "here are arguments to do sth with them in this function"
- "here is the function to call once you are done"

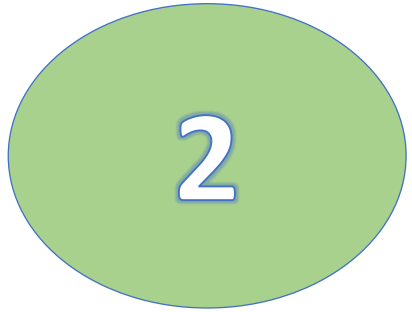
Function returning an object which has methods

- Q: What does this function return?

(what data type?)

```
function math() {
  return {
    add: function(x, y) {
      return x + y;
    },
    multiply: function(x, y) {
      return x * y;
    }
  };
}
```

- Q: how would you add two numbers using the code snipped above?
- math().add(2,3)



Asyc operations and callback hell!

What if a function does asynchronous operations ?

- Microservices communicate via the net using API calls
- and any communication via net is asynchronous

problem

- In js Functions can only return a single value, whether of primitive data type (number, string etc) or a non- primitive data type such as an object.

```
function add (a, b) {
  return a + b;
}
```

```
function person (name, age) {
  return {
    name: name,
    age: age
  };
}
```

- Usually, we call the function in our code, and do stuff based on the retuned value. But what if the operation performed by a function is *asynchronous?* ☹
- Q: why functions doing asynchronous operations are more complicated?

- Dealing with functions doing asynchronous stuff, we are not only interested in the returned value, but **also in how it was returned**, was the operation successful or failed (timed-out, DB is down, authentication error etc)
- 1- Do something with the success
- 2- Or deal with the failure gracefully

- But functions return only one thing! (read with scream)
- Q: Whats your solution ?

- Lets simply return an object that has two callback methods, resolve() and reject()! We simply override them based on what we want ☺

Promise in JavaScript

Understanding promises

Understanding Promises

- "Imagine yourself working in a company. Your boss **promises** you a raise someday.
- You don't know if you will get that raise until next quarter, you just wait(*pending*). Your boss can either really get you the raise(*fulfilled*), or stand you up and withhold (*reject*)it if she/he is not happy :(.

- That is a **promise**. A promise has 3 states:

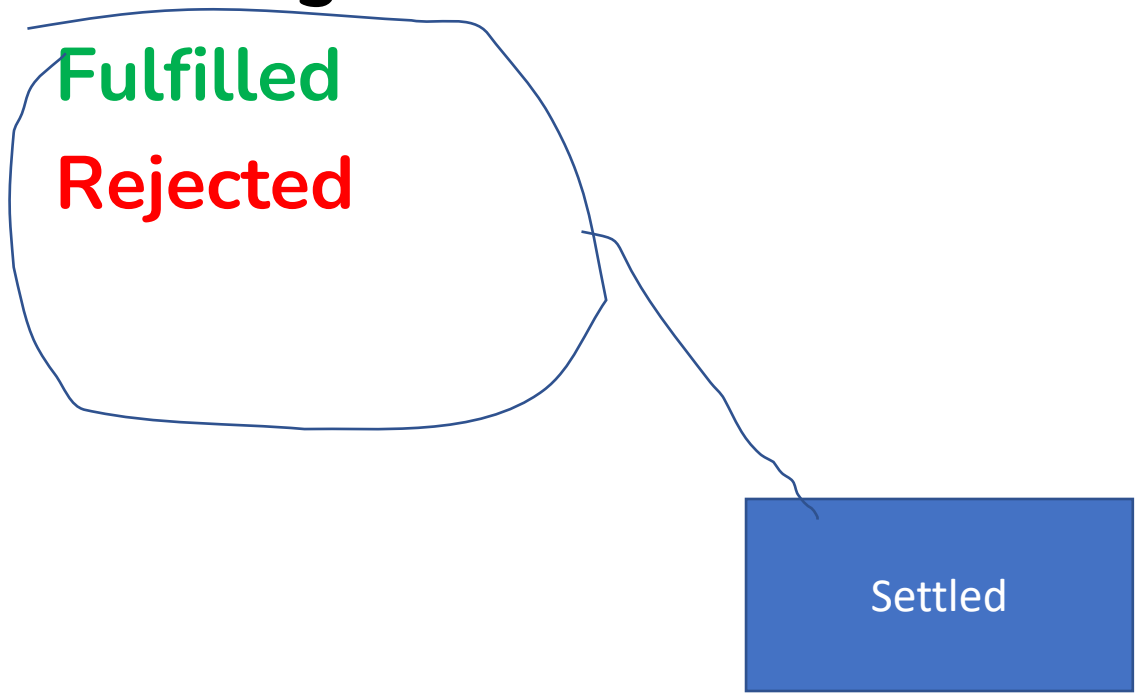
1. **Pending**: You *don't know* if you will get the raise
2. **Resolved (Fulfilled)**: Boss is happy, gets you the raise. describe what you would do in a function called resolve()
3. **Rejected**: Your boss is unhappy, withholds the raise. Lets describe what you would do in a function called reject()

Remember the three possible status of a JS promise

Pending

Fulfilled

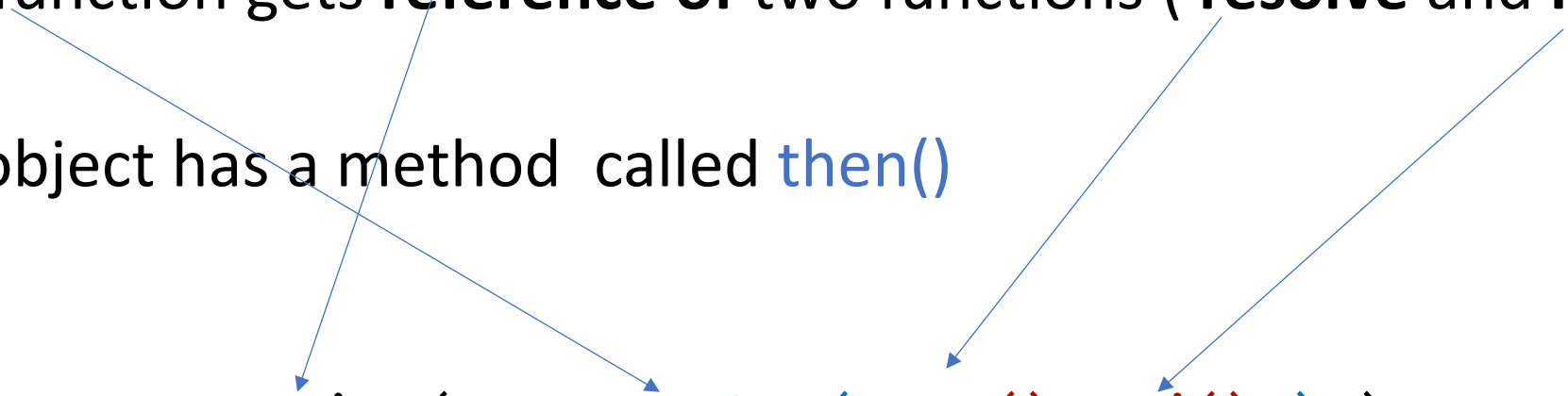
Rejected



- Js runs asynchronously. That means you never exactly know when things happen. Remember the restaurant with single waiter analogy. When you ask the waiter for sth, you never know when he is going to get back to you.
- You only get a promise. (he might get back to you , he might forget)
- When you make an AJAX call you never know when you are going to get back the result. Imagine other part of your code is pending for the result of this AJAX.

Promise() is an object in JavaScript

1. The Promise object is used for asynchronous computations.
2. A Promise represents a single asynchronous operation that hasn't completed yet, but is expected in the future.
3. You pass a function to the constructor which gets executed the moment the promise object gets created
4. The executer function gets **reference of** two functions (**resolve** and **reject**) as input
5. The Promise object has a method called then()



```
let promise = new Promise( myExecuter( res(),rej() ) );
```

Promise object constructor

```
let promise = new Promise(
    (resolve, reject)=> {
      // some condition to decide outcome of promise (reject or resolve)
      reject("Rejected!");
      resolve("Resolved!");
    }
);
promise.then(
    (resolveMes)=> {
      console.log(resolveMes);
    }
  ,
    (rejectMes)=> {
      console.log(rejectMes)
    }
);
```

1. Promise constructor to which we can pass a function called executer. This function itself gets two functions first one points to resolve, the second one points to reject

3. Promise executer function. Gets executed **immediately** upon creation of the promise object

4. resolve and reject invoke handlers **asynchronously** and **mutually exclusive** And run only once

Resolve, reject change promise status synchronously which lead to put handlers in microtask and run only after the current call stack finishes
Promise handlers do **NOT** wait for "all code in the program" to finish.
They wait only for the **current call stack** to unwind.

2. Definition of the two handlers (functions) resolve and promise are given here, or could be given elsewhere but passed by reference

In this example, when reject handlers invokes, the resolve will be ignored because the promise is already settled.

5. The then() method itself returns a promise

Note: blue color indicates resolve contents, Red indicates reject ones

- resolve / reject do not invoke handlers directly; they change the promise's state , JavaScript then queues the handlers (as you know queuing ➔ asynch)

- In fact resolve(),reject() **create** the promise's value; .then()handlers **receives** the promise's value
-

  resolve is one specific function created by js internally, the first callback passed to .then(callback1(), callback(2)) is a different function they exist at different times they play different roles

Eager vs lazy

- Promises are eager. This means that **when a new Promise is constructed, it immediately starts executing, attempting to resolve the** asynchronous value (meaning the resolve, reject methods are called asynchronously by a synchronous executer method)

Promise Examples

Example 1:

- In what order the strings below will be logged?

```
let promise = new Promise(
    function (resolve, reject) {
        // do some stuff
        console.log("2 called immediately ");
        resolve("4 Resolved!");// called asynchronously & mutually exclusive
        reject("3 Rejected!");// called asynchronously & mutually exclusive
        console.log("5 called immediately ");//pending ..
    }
);
console.log("1 Main execution stack");
promise.then(
    function (resMessage) {
        console.log(resMessage);
    },
    function (rejMessage) {
        console.log(rejMessage)
    }
);
```

Resolve and reject are called synchronously to change the state of
Promise immediately which in turn invokes the corresponding handlers asynchronously and mutually exclusive

A:
2 (we said the executor will invoke the moment the promise constructor is called.
5 (2 and 5) execute synchronously
1 then this like executes synchronously as it is in the same execution stack
4 invokes asynchronously and mutually exclusive which means the main execution stack finishes first, then one of these two the invokes first, will block the other one from execution .

Example 2: in what order the numbers will be logged?

```
1  console.log("7");
2  let promise = new Promise(
3      function (resolve, reject) {
4          resolve("4");
5          console.log("2");
6          reject("3");
7          console.log("5");
8      }
9  );
10 console.log("1");
11 promise.then(
12     function (resMessage) {
13         console.log(resMessage);
14     },
15     function (rejMessage) {
16         console.log(rejMessage)
17     }
18 );
19 console.log("8");
```

```
7
2
5
1
8
4
```

Example 3

```
let promise = new Promise(
    function (resolve, reject) {
        console.log(" 2 ");
        reject(" 3 ");
        resolve(" 4 ");
        console.log(" 5 ");
    }
);
promise.then(
    function (st) {
        console.log(st);
    },
    function (st) {
        console.log(st)
    }
);
console.log(" 1 ");
```

Q: In what order the console logs the numbers?

A:
2
5
1
3

Q: what happened to 4?

Example 4: How many times 'Hi' will be logged?

```
// How many times "Hi" will be logged?
let promise = new Promise((res) => {
    setInterval(() => {
        res("Hi");
    }, 100);
});
promise.then((mes)=>{console.log(mes)});
```

Tip: Remember we said promises are either pending or settled (settled: resolved or rejected). Once they are settled, their state will not change

If the promise has already been resolved to a value to a rejection or to another promise, then the res method does nothing .

A: Hi will be logged only once.

Example 5:

- Promise.resolve() and Promise.reject() are shortcuts to manually create an already resolved or rejected promise respectively.
- In what order the numbers will be logged?

```
Promise.resolve().then(() => console.log(1));
console.log(2);
```

- Remember the note in the gray slide about functions passed to then(), which means resolve and reject, will never be called synchronously, even with an already-resolved promise
- Therefore the order is:

2

1

Chaining promises

If you remember the note 5 of the gray slide, The then() method returns a Promise. That's why we can chain promises .

The then() method returns a Promise. It takes up to two arguments: callback functions for the success and failure cases of the Promise

function return (new Promise)).then(()=>{return (new Promise)}).then

promise.then()

Example 6 (in-class activity)

```
new Promise(resolve => {
  resolve(1);
})
  .then(a => {
    console.log("First:", a);
    return a * 2;
  })
  .then(b => {
    console.log("Second:", b);
    return b * 3;
  })
  .then(c => {
    console.log("Final:", c);
  });
```

1. What will be logged ?

Example 7

- In what order the numbers will be logged?

```
const wait = ms => new Promise(resolve => setTimeout(resolve, ms));
wait(0).then(() => console.log(2));
Promise.resolve().then(() => console.log(3)).then(() => console.log(1));
console.log(4);
```

Example 8 (in-class activity)

```
const p = new Promise((resolve, reject) => {
  const success = false;
  if (success) {
    resolve("Everything worked!");
  } else {
    reject("Something went wrong");
  }
});


p.then(result => {
  console.log("SUCCESS:", result);
})
  .catch(error => {
    console.log("ERROR:", error);
  });
```

1. What will be logged ?
2. What happens if we don't handle reject and remove catch() ?

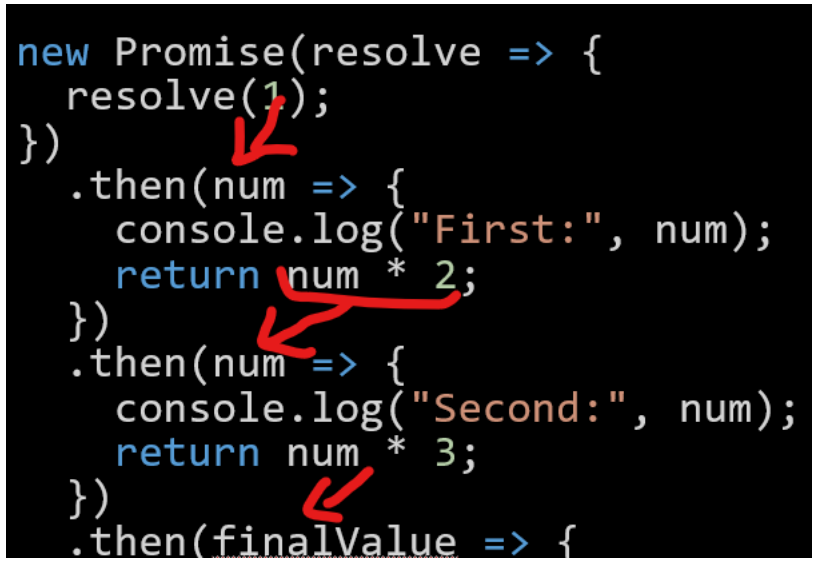
A: Unhandled reject error

Example 9 (in-class activity)

```
new Promise(resolve => {
  resolve(1);
})
  .then(num => {
    console.log("First:", num);
    return num * 2;
  })
  .then(num => {
    console.log("Second:", num);
    return num * 3;
  })
  .then(finalValue => {
    console.log("Final:", finalValue);
  });
```

1. What will be logged ?

- Then() returns another promise and passes num*3 to the resolve of that new promise
- Q: why it does not pass the num*3 to the reject of the new promise?
- The value returned from .then() **is passed to the internal resolve function of the NEW promise created by .then()**.



- The core rule (from the Promise spec, simplified)
- Returning a value is defined as success (passing to resolve of new promise)
  Throwing an error is defined as failure (passing to reject of new promise)

-

Example 10 (in-class activity)

```
new Promise(resolve => {
  setTimeout(() => resolve(2), 500);
})
  .then(num => {
    console.log("First:", num);
    return num * 3;
  })
  .then(num => {
    console.log("Second:", num);
    return num + 1;
  })
  .then(finalValue => {
    console.log("Final:", finalValue);
  });
```

1. What will be logged ?

- What is the difference between try catch error and promise resolve reject?
- Promises were designed to behave like **async try/catch blocks**.

```
// Synchronous code

try {
  const x = work();
  return x;   // success path
} catch (e) {
  throw e;    // failure path
}
```

```
//Promise .then() handler

.then(x => {
  return x;  // resolve
})
  .catch(e => {
    throw e;  // reject
  })
```

Example 11 (in-class activity)

```
f1 = () =>{console.log(1)};
f2 = () =>{console.log(2)};
f3 = () =>{console.log(3)};

// async() does things asynchronously for us
function async(callMeBack) {
    setTimeout(() => {
        callMeBack();
    }, Math.floor(Math.random() * 1000)
    );
}
//we want f1,f2 and f3 execute in f1,f2, f3
// asynchronously How can we guarantee ?
// would below work? if not what to do?
async(f1);
async(f2);
async(f3);
```

```
we want f1,f2 and f3 execute
in f1,f2, f3
asynchronously How can we
guarantee ?
 would below work? if not
what to do?
```

Lets observe
how

promise status

changes

In JavaScript, **"promise states"** and **"promise status"** refer to the same concept: the internal, unchangeable condition of a `Promise` object at any given time. There are three possible states (or statuses): **`pending`**, **`fulfilled`**, and **`rejected`**.

Try this code example 12

and observe the changes in the property values of the instantiated promse object over time

```
const prom = new Promise( (res, rej)=>{
    setTimeout(() => {
        res('status changed now!');
    }, 5000);
})
```

Example: observe how status and value change after 5 sec

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => resolve("status changed now!"), 5000);
});
promise.then(alert);
```

```
Promise {<pending>}
    __proto__: Promise
    [[PromiseStatus]]: "pending"
    [[PromiseValue]]: undefined
> promise
Promise {<resolved>: "status changed now!"}
    __proto__: Promise
    [[PromiseStatus]]: "resolved"
    [[PromiseValue]]: "status changed now!"
>
```

After around a 5 second gap, the status changes from pending to resolved

Example 9: using default definition of resolve method

- what will be the status and value of the object promise after execution of this code?

```
let promise = new Promise(function(resolve, reject) {
  resolve ("nicely done!");
});
```

```
> //1 using default definition of resolve
  let promise = new Promise(function(resolve, reject) {
    resolve ("nicely done!");
  });
< undefined

> promise
< ▼Promise {<resolved>: "nicely done!"} i
    ▶ __proto__: Promise
      [[PromiseStatus]]: "resolved"
      [[PromiseValue]]: "nicely done!"
```

Example 10: using default definition of reject method

- what will be the status and value of the object promise after execution of this code?

```
let promise = new Promise(function(resolve, reject) {
    console.log("Doing something before rejection");
    reject ("Whoops!");
});
```

```
> let promise = new Promise(function(resolve, reject) {
    console.log("Doing something before rejection");
    reject ("Whoops!");
  });
  Doing something before rejection
< undefined
✖ ▶Uncaught (in promise) Whoops!
```

Why?
Because we did not catch the error (we did not define the reject method)!
The js engine throws the error using default definition of reject! In fact rejection naturally means something went wrong

Q: How to execute two successive AJAX calls (second one only after first one is guaranteed finished)?

- Suppose we wish to
- make an Ajax call to server A to get something
- And **then**
- making another AJAX call to server B based on what you got from server A.
- **Q:** How can you **guarantee** the **right order**?
- **A**: calling myFunc2 inside myFunc1, line 65

Have you heard about callback hell?

```
57  const xhttp = new XMLHttpRequest();
58  let score = 0;
59  function myFunc1() {
60      xhttp.open("GET", "http://localhost:8888/getscore/?name=john", true);
61      xhttp.send();
62      xhttp.onreadystatechange = function () {
63          if (this.readyState == 4 && this.status == 200) {
64              document.getElementById("demo").innerHTML =
65                  score = parseInt(this.responseText);
66          }
67      };
68  }
69
70  function myFunc2() {
71      xhttp.open("PATCH", "http://localhost:7777/updatescore/" + str, true);
72      xhttp.send("name=john;score=" + score);
73      xhttp.onreadystatechange = function () {
74          if (this.readyState == 4 && this.status == 200) {
75              document.getElementById("demo").innerHTML =
76                  this.responseText;
77          }
78      };
79  }
```

.finally

3 main methods of the Promise object are

.then .catch .finally

What are these methods?

```
> let promise = new Promise(function(resolve, reject) {
    resolve ("nicely done!");
  });
< undefined
> promise
< ▼Promise {<resolved>: "nicely done!"}
    ▼__proto__: Promise
      ▶catch: ƒ catch()
      ▶constructor: ƒ Promise()
      ▶finally: ƒ finally()
      ▶then: ƒ then()
       Symbol(Symbol.toStringTag): "Promise"
      ▶__proto__: Object
    [[PromiseStatus]]: "resolved"
    [[PromiseValue]]: "nicely done!"
```

.then method (we already know)

```
promise.then(
    function (result) { /* handle a successful result */ },
    function (error) { /* handle an error */ }
);
```

- The first argument of .then is a function that:
- runs when the Promise is resolved, and
- receives the result.

- The second argument of .then is a function that:
- runs when the Promise is rejected, and
- receives the error.

Example: What's the output of the code below?

```
function foo(st) {
    document.write(st)
}
let promise = new Promise(function (resolveFunc, rejectFunc) {
    resolveFunc("<h1>H</h1>ello ");
})
promise.then(foo);
```

A: When we pass only one function to .then method, that function replaces the default definition of resolve method

So the code above will display 1 on the browser window

Example: What's the output of the code below?

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => resolve(2), 1000);
    resolve(1);
});
promise.then(alert);
```

it will alert "1" on the browser window and ignore the second resolve

.catch method

- If we're interested only in errors, then we can use null as the first argument:
- .then(null, **errorHandlingFunction**)
- Is the same as
- .catch(**errorHandlingFunction**)

```
> let promise = new Promise((resolve, reject) => {
    setTimeout(() => reject(new Error("Whoops!")), 5000);
  });

  // .catch(f) is the same as promise.then(null, f)
  promise.catch(alert); // shows "Error: Whoops!" after 5 second
<· ▼Promise {<pending>} i
    ▶__proto__: Promise
     [[PromiseStatus]]: "pending"
     [[PromiseValue]]: undefined

> promise
<· ▼Promise {<rejected>: Error: Whoops!
        at setTimeout (<anonymous>:2:27)} i
    ▶__proto__: Promise
     [[PromiseStatus]]: "rejected"
    ▶[[PromiseValue]]: Error: Whoops! at setTimeout (<anonymous>:2:27)

>
```

5 sec

.catch example

.finally

- Just like there's a finally clause in a regular try {...} catch {...}, there's finally in promises.

- The call **.finally(f)**
- is similar to
- .then(f, f)
- in the sense that it always runs when the promise is settled: be it resolve or reject. (src: javascript.info)

- Note: there are more into .finally method that is outside the scope of this class.

.finally method example

- Remember if resolve function executes, reject function will be ignore and vice versa. However regardless of how the promise settles the finally method will be called

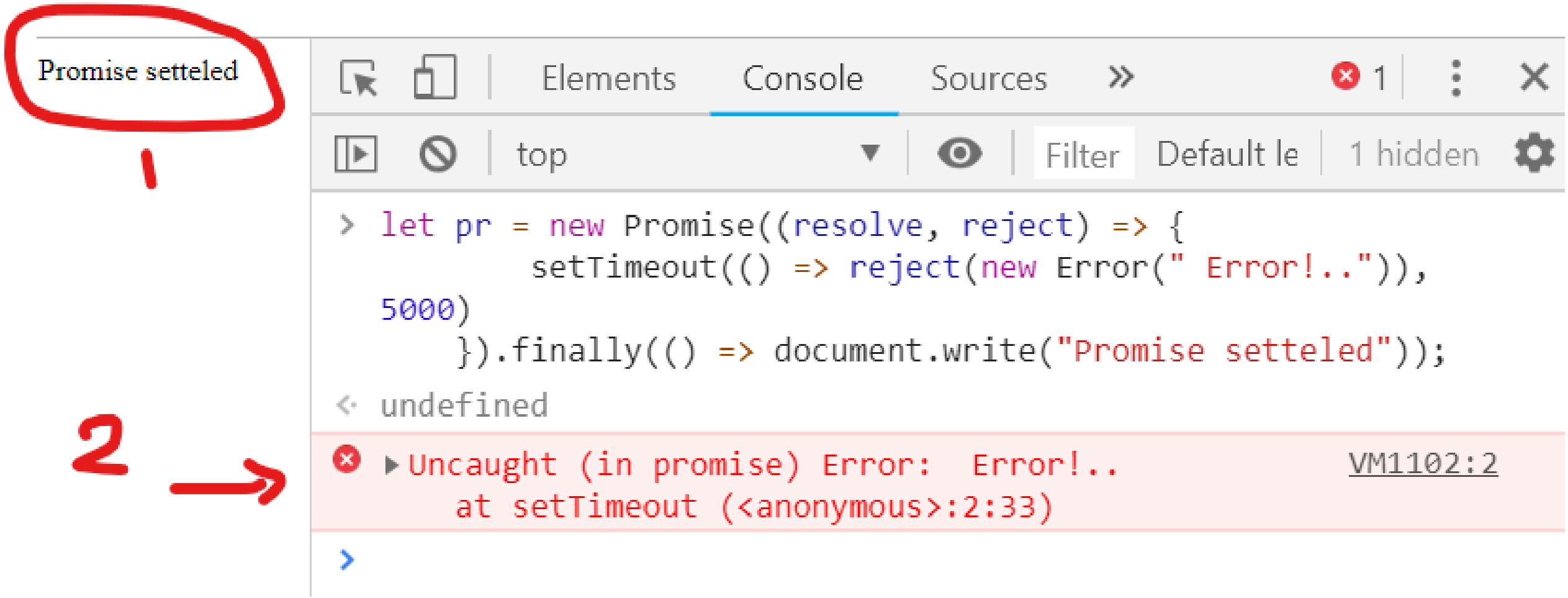
```
let pr = new Promise((resolve, reject) => {
    setTimeout(() => resolve("result of resolve "), 5000)
})

    .then(result => alert(result))// <-- .then handles the result
    .finally(() => alert("Promise ready"));
```

.finally method example 2

- Remember if resolve function executes, reject function will be ignore and vice versa. However regardless of how the promise settles the finally method will be called .
- In example below we did not catch error nor we defined reject function.
- Remember the default function would terminate the script

```
let pr = new Promise((resolve, reject) => {
    setTimeout(() => reject(new Error(" Error!..")), 5000)
}).finally(() => document.write("Promise setteled"));
```

.finally method example 2 cont…



Async/Await

- built on top of promises, allowing you to write asynchronous code that looks more like synchronous code.
- async: When a function is declared as async, it automatically returns a promise.
- await: This is used inside async functions to wait for a promise to resolve or reject.

- Good exapls at https://www.w3schools.com/js/js_async.as

In class activity 6

- Q1: For the given API server, measure the time diff from the time you sent a GET request, to the time you receive the response

Q2: Measure the time diff from the time you receive the response header, to the time you receive the response

GET vs POST

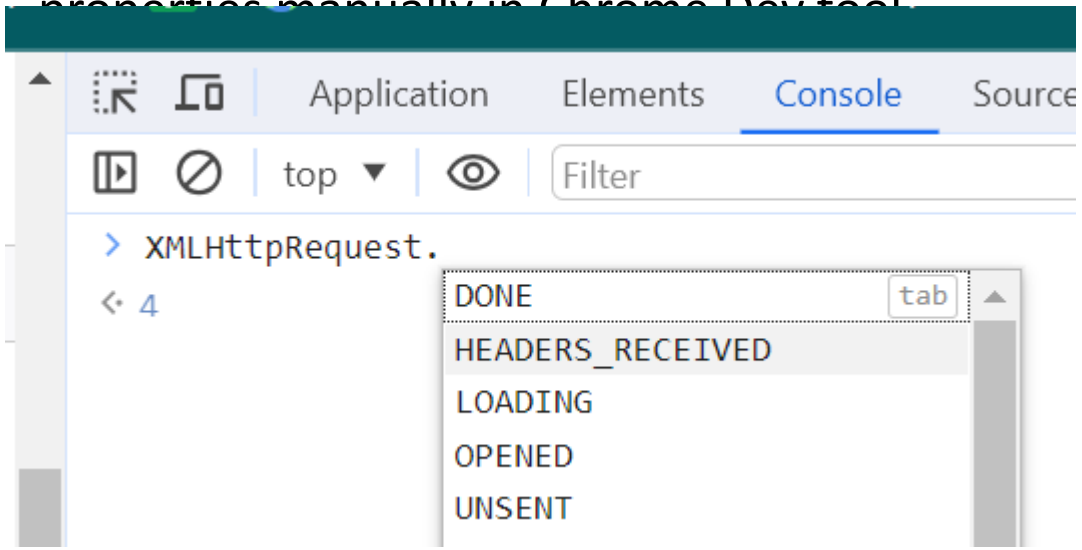
- So we stated that there is a limitation on data size being sent over a single GET **request**. However, there is no such limitation on POST request
- Q: what about the **responses** of GET vs POST? Is there such limit for their responses too?

- A: no! There are no standardized limitations on the size of responses for GET or POST requests specified by the HTTP protocol.
  Needless to mention practical limitations can be influenced by server configurations, network constraints, or browser limitations.

W3schools is inaccurate about onreadystatechanged

- Turns out https://www.w3schools.com/xml/ajax_xmlhttprequest_response.asp is not accurate on ready states status values ! The one we covered in lecture is accurate
- Accurate one:
  0 (UNSENT): The XHR object has been created, but open() has not been called yet.
- 1 (OPENED): open() has been called.
- 2 (HEADERS_RECEIVED): send() has been called, and the headers of the response are available.
- 3 (LOADING): The response is being received. As data comes in, the responseText property is updated.
- 4 (DONE): The operation is complete, and either the request has been successfully completed (status code 2xx) or an error occurred.
- So verified that by looking into the XMLHttpRequest class properties manually in Chrome Dev tool

8



readyState	Holds the status of the XMLHttpRequest. 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready

not accurate!

References and resources

- https://scotch.io/tutorials/javascript-promises-for-dummies#toc-chaining-promises
- Spring.io
- MDN mozlla
- https://medium.com/front-end-weekly/ajax-async-callback-promise-e98f8074ebd7
- https://zapier.com/learn/apis/chapter-6-api-design/
- https://restfulapi.net/versioning/
- RESTful Web API Design with Node.js 10 - Third Edition
- https://phil.tech/2016/http-rest-api-file-uploads/ (for your term project)
- chatGPT