

# **FRONTEND DEVELOPMENT WINTERSEMESTER 2020**

A background image of a roller coaster track with a car at the bottom, overlaid with a red tint. The text "ASYNCHRONOUS JAVASCRIPT" is written in large, bold, white capital letters across the center of the image.

# ASYNCHRONOUS JAVASCRIPT

# SHORT DISCLAIMER

» You'll see es6 arrow functions in the slides

» For now:

» all 4 variants can be seen as equal

» we'll look at the difference in the next lecture

```
function myFunction() { return 'some value' } // function
const myFunction = function () { return 'some value' } // anonymous function
const myFunction = () => { return 'some value' } // es6 arrow functions
const myFunction = () => 'some value' // es6 arrow functions with implicit return
```



A background image of a roller coaster track with a car at the bottom, overlaid with a red tint. The text "ASYNCHRONOUS JAVASCRIPT" is written in large, bold, white capital letters across the center of the image.

# ASYNCHRONOUS JAVASCRIPT



# JS & CONCURRENCY

- » JS is single threaded
- » eliminate lots of concurrency issues
- » Only one thing can happen at a time
  - » long running operations would block execution
  - » eg. network requests





# JS & CONCURRENCY

- » Network requests would block main thread
- » no user interaction would be possible
- » button clicks wouldn't be registered
- » scrolling would not work
- » ...

# SYNCHRONOUS CODE

**“IN A SYNCHRONOUS  
PROGRAMMING  
MODEL, THINGS  
HAPPEN ONE AT A  
TIME.”<sup>5</sup>**

<sup>5</sup> [https://eloquentjavascript.net/11\\_async.html](https://eloquentjavascript.net/11_async.html)



# JS AND SYNCHRONOUS CODE EXECUTION<sup>3</sup>

```
const second = () => {  
  console.log('Hello there!');  
}  
const first = () => {  
  console.log('Hi there!');  
  second();  
  console.log('The End');  
}
```

<sup>3</sup> Visualisation of code execution

A background image of a roller coaster track with a car at the bottom, overlaid with a red tint. The text "ASYNCHRONOUS CODE EXECUTION" is centered in white.

# ASYNCHRONOUS CODE EXECUTION

**“AN ASYNCHRONOUS  
MODEL ALLOWS  
MULTIPLE THINGS TO  
HAPPEN AT THE SAME  
TIME.”<sup>5</sup>**

<sup>5</sup> [https://eloquentjavascript.net/11\\_async.html](https://eloquentjavascript.net/11_async.html)





# ASYNC JS

- » multiple non-cpu bound computations can happen at the same time
- » network requests
- » reading files from disk
- » waiting for a `setTimeout`

# ASYNCHRONOUS JS

- » when waiting for network response
  - » js continues doing other tasks
  - » once the response is there
  - » and js has nothing else to do
  - » js continues processes network response

# CONTINUATION PASSING STYLE



**NO PROCEDURE IS ALLOWED TO RETURN TO ITS CALLER--EVER.<sup>1</sup>**

<sup>1</sup> pre-es6

# CONTINUATION PASSING STYLE

- » or programming with  
callbacks
- » a callback is a function  
which is called when  
something happened

# CONTINUATION PASSING STYLE

» real world example:

» you wait for a package and call the delivery service

» the delivery service notes your "contact details" (callback)

» and he'll call you back once he has additional information



# CONTINUATION PASSING STYLE

```
function multiplyBy2(x, whenDone) {  
    //  
    // callback is defined which will receive the result  
    whenDone(x * 2)  
}
```

```
function logResult(value) {  
    console.log(value)  
}
```

```
multiplyBy2(4, logResult) // 8 will be logged
```

# CONTINUATION PASSING STYLE IN THE BROWSER

- » setTimeout does not block js execution
- » other scripts can still execute
- » after ~1 second callback will be invoked

```
setTimeout(() => {  
    // will be called in the future  
}, 1000)
```

# CONTINUATION PASSING STYLE IN NODE.JS

- » `fs.readFile` does not return to it's caller
- » accepts a callback which is invoked once the file was read

```
// in node.js
fs.readFile('file.txt', (err, data) => {
  if (err) throw err; // throws on error (eg. file not found)
  console.log(data); // logs the contents of file.txt
})
```



# USECASE:

- » fetch function which fetches data from an api
- » does a HTTP GET request
- » accepts URL and callback
- » callback receives server response

```
fetch(url, (err, response) => {})
```

# USECASE:

```
function logCurrentUser() {  
  fetch('/api/currentUser', {currentUser}) => {  
    console.log(currentUser)  
  })  
}
```

A roller coaster track with a car at the bottom, overlaid with a red tint and the text 'WHAT IF ADDITIONAL DATA NEEDS TO BE FETCHED?'.

**WHAT IF ADDITIONAL DATA NEEDS TO BE FETCHED?**



# CALLBACKS:

```
function myBestFriendsAddress() {  
  fetch('/api/currentUser', {currentUser}) => {  
    // will be executed once we get a response from `currentUser`  
    fetch(`/api/user/${currentUser.id}/bestFriend`, {bestFriend}) => {  
      // will be executed once we get a response from `bestFriend`  
      fetch(`/api/user/${bestFriend.id}/address`, {bestFriendsAddress}) => {  
        // ...  
      }  
    }  
  }  
}
```

# CALLBACK HELL:

```
1  function hell(win) {  
2    // for listener purpose  
3    return function() {  
4      loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {  
5        loadLink(win, REMOTE_SRC+'/lib/async.js', function() {  
6          loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {  
7            loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {  
8              loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {  
9                loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {  
10               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {  
11                loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {  
12                 loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {  
13                  async.eachSeries(SCRIPTS, function(src, callback) {  
14                   loadScript(win, BASE_URL+src, callback);  
15                  });  
16                 });  
17                });  
18               });  
19              });  
20             });  
21            });  
22           });  
23          });  
24         });  
25        });  
26       }  
    }  
  }
```



**HOW COULD WE FLATTEN THIS TREE?**



# ESG PROMISES TO RESCUE



# PROMISES

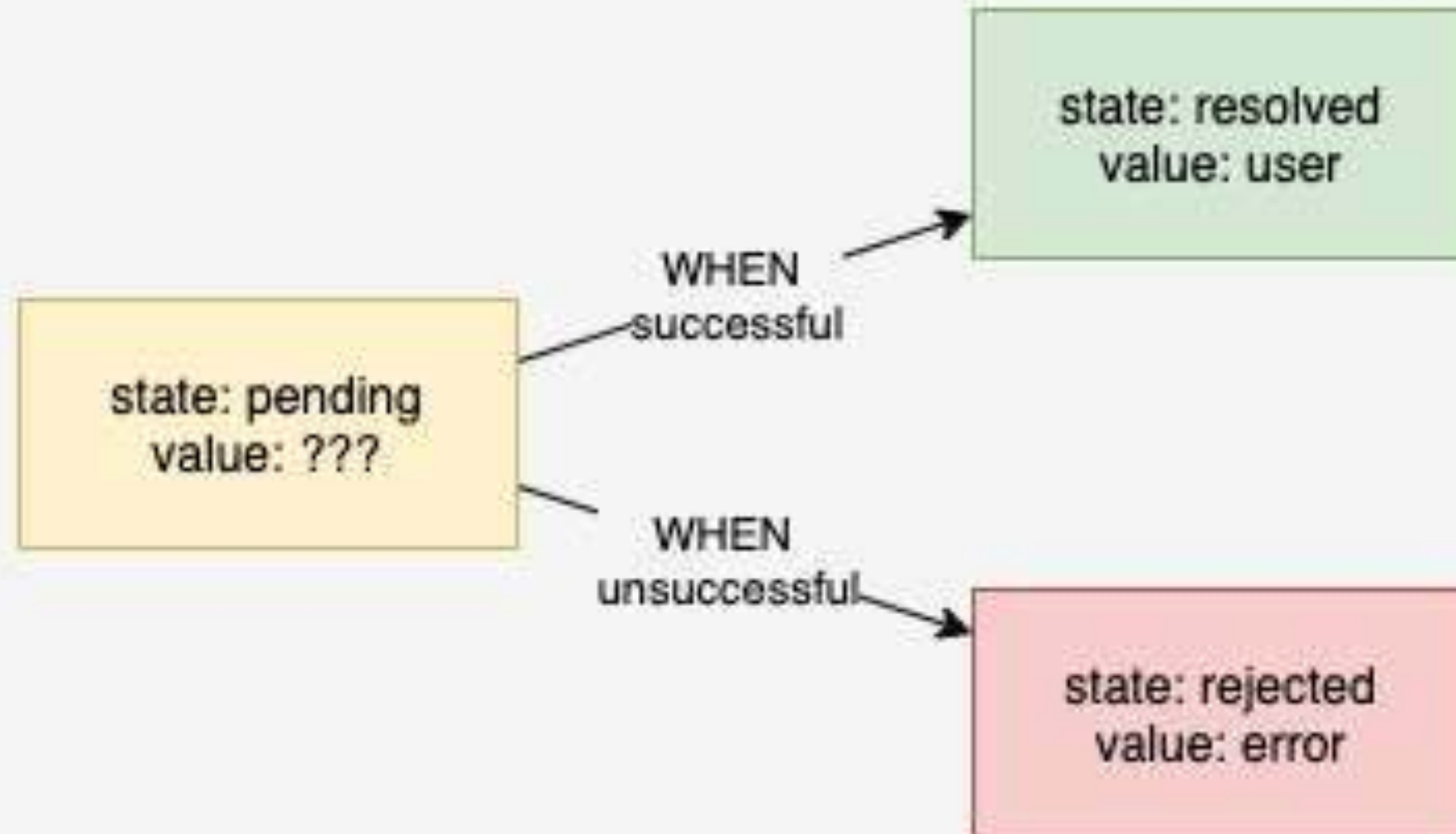
The Promise object represents the eventual completion (or failure) of an asynchronous operation and its resulting value.

# PROMISES

- » A promise is in one of 3 states
  - » pending
  - » fulfilled
  - » rejected
- » Other languages call it futures



# PROMISES



# TRANSFORM CALLBACKS TO PROMISES

```
function myBestFriendsAddress() {  
  fetch('/api/currentUser', {currentUser}) => {  
    // will be executed once we get a response from `currentUser`  
    fetch(`/api/user/${currentUser.id}/bestFriend`, {bestFriend}) => {  
      // will be executed once we get a response from `bestFriend`  
      fetch(`/api/user/${bestFriend.id}/address`, {bestFriendsAddress}) => {  
        // ...  
      }  
    }  
  }  
}
```

# TRANSFORM CALLBACKS TO PROMISE

## WRAP FETCH FUNCTION IN PROMISE

```
const fetchAsPromise = (url) => {  
  return new Promise((resolve) => { // create a new promise  
    // call the fetch function and *WHEN* the request is done, resolve the promise  
    fetch(url, resolve)  
  })  
}
```

# TRANSFORM CALLBACKS TO PROMISE

## USE FETCHASPromise

```
const fetchAsPromise = (url) => new Promise((resolve) => {
  fetch(url, resolve)
})

fetchAsPromise('/api/currentUser')
  .then((currentUser) => fetchAsPromise(`/api/user/${currentUser.id}/bestFriend`))
  .then((bestFriend) => fetchAsPromise(`/api/user/${bestFriend.id}/address`))
  .then((bestFriendsAddress) => console.log(bestFriendsAddress))
```



# TRANSFORM CALLBACKS TO PROMISE

## USE FETCHASPromise

```
const fetchAsPromise = (url) => new Promise((resolve) => {
  fetch(url, resolve)
})

fetchAsPromise('/api/currentUser')
  .then((currentUser) => fetchAsPromise(`/api/user/${currentUser.id}/bestFriend`))
  .then((bestFriend) => fetchAsPromise(`/api/user/${bestFriend.id}/address`))
  //      ^^^^^^^^^^^
  // value of previous result is the first argument
  .then((bestFriendsAddress) => console.log(bestFriendsAddress))
```

# PROMISES ERROR HANDLING

```
const fetchAsPromise = (url) => new Promise((resolve, reject) => {  
    //  
    // enhance previous fetchAsPromise with reject param  
  
    fetch(url, {data}) => {  
        if (data.status === 200) { resolve(data) }  
        else { reject(data) }  
    }  
})  
})
```

# PROMISES ERROR HANDLING

```
const fetchAsPromise = (url) => new Promise((resolve, reject) => {
  fetch(url, {data}) => {
    if (data.status === 200) { resolve(data) }
    else { reject(data) }
  })
})

fetchAsPromise('/api/currentUser')
  .then((currentUser) => fetchAsPromise(`/api/user/${currentUser.id}/bestFriend`))
  // ...
  .catch(() => alert('something went wrong 🥲'))
// ^^^^^
// catch all promise rejections
```

# PROMISES ERROR HANDLING

- » remember promise has 3 states (pending, resolved, rejected)
- » `.catch` converts a rejected promise to a resolved one

```
fetchAsPromise('/api/currentUser')  
  // state: resolved  
  .then(() => fetchAsPromise(`/api/someUnknownURL`))  
  // state: rejected  
  .catch(() => alert('something went wrong 🥹'))  
  // state: resolved  
  .then(() => console.log('will be logged'))
```

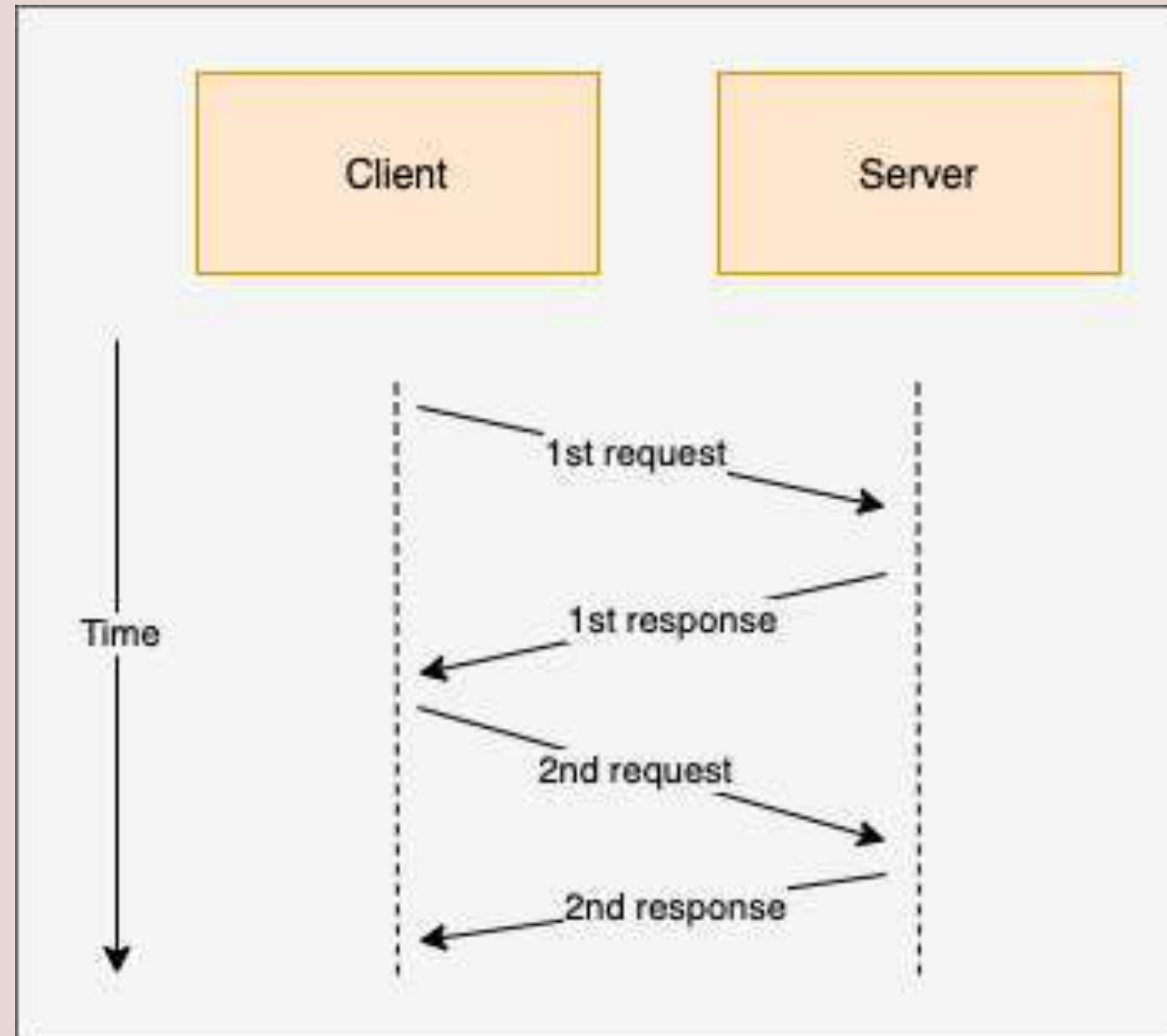


# CREATING PROMISES OUT OF STATIC VALUES

```
Promise.resolve(1)  
  .then((value) => { console.log(value) }) // 1 will be logged
```

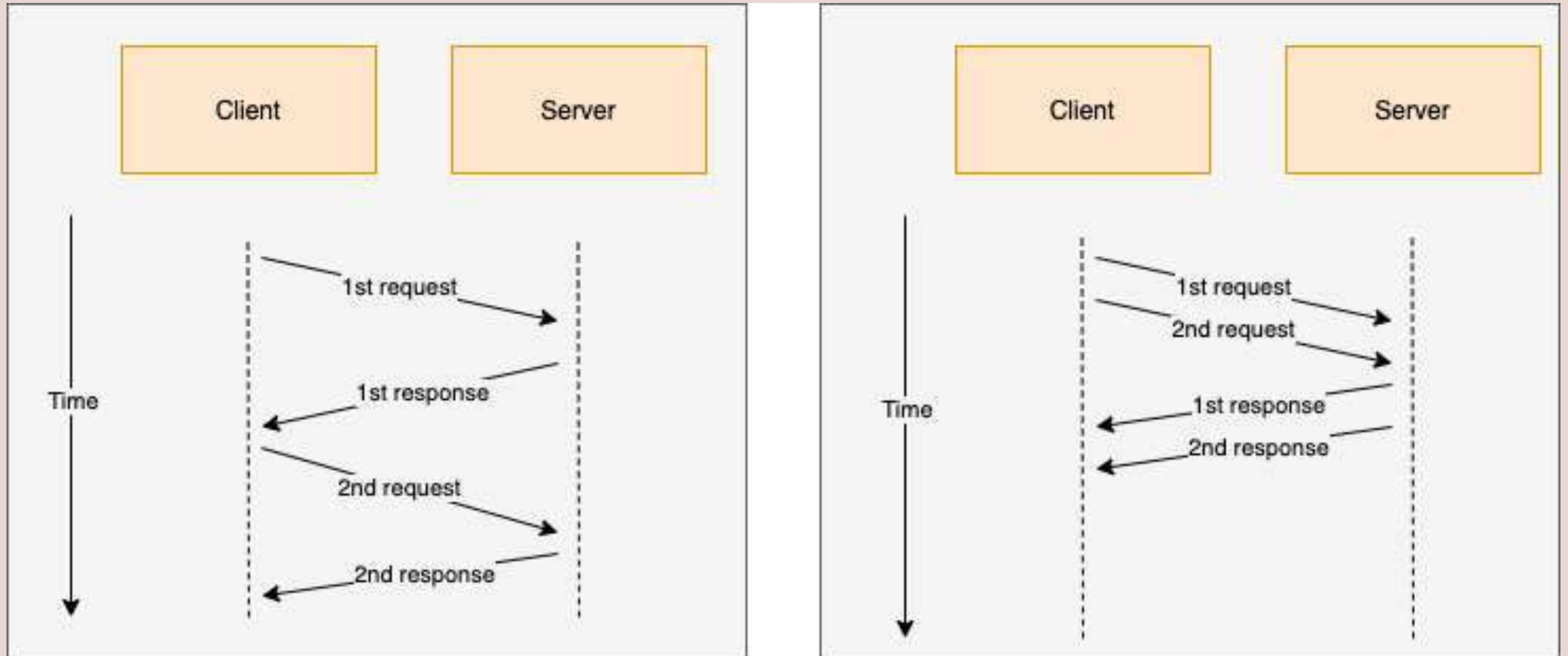
```
Promise.reject(1)  
  .catch((value) => { console.log(value) }) // 1 will be logged
```

# WAITING FOR MULTIPLE PROMISES



**DIDN'T YOU SAY NETWORK  
REQUESTS CAN BE DONE  
CONCURRENTLY?**

# WAITING FOR MULTIPLE PROMISES





# WAITING FOR MULTIPLE PROMISES

- » waiting for each request before doing the next one is slow
- » `Promise.all` makes it possible to run and wait for multiple promises concurrently
- » if possible try to parallelize promises via `Promise.all`

# WAITING FOR MULTIPLE PROMISES

```
Promise.all([
  fetchAsPromise(`/api/currentUser`),
  fetchAsPromise(`/api/weather`)
]).then([ currentUser, weather ] => {
  console.log(currentUser)
  console.log(weather)
})
```

# RACING MULTIPLE PROMISES

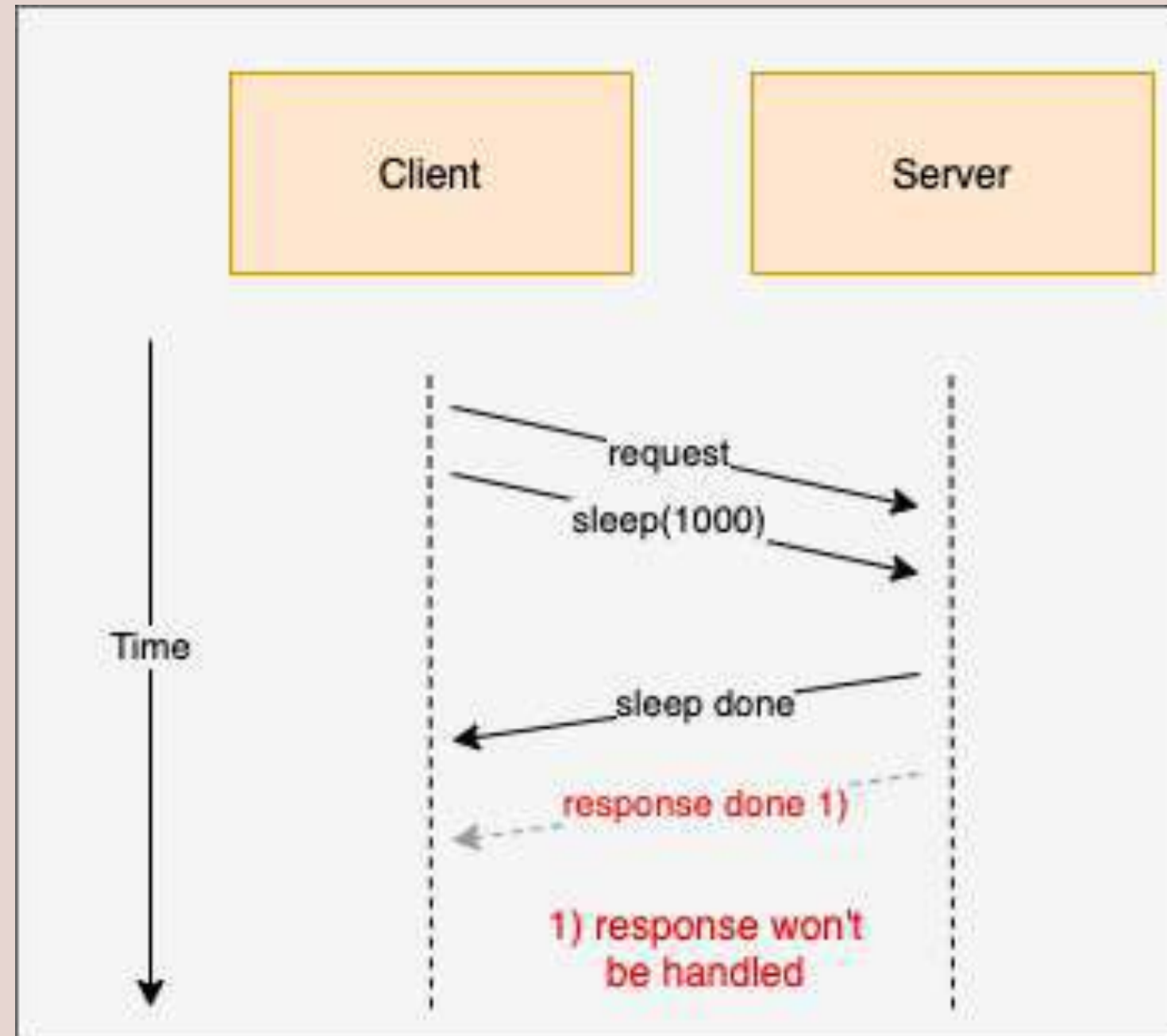
- » `Promise.race` returns the fastest promise <sup>2</sup>
- » can be used to implement a timeout for slow requests

```
const sleep = (timeout) => new Promise((resolve) => setTimeout(resolve, timeout))
```

```
Promise.race([  
  fetchAsPromise(`/api/currentUser`),  
  sleep(1000).then(() => ({ error: 'timeout' })),  
]).then((result) => {  
  if (result.error) {  
    throw new Error('Timeout')  
  }  
  return result;  
})
```

<sup>2</sup> I personally only used it once

# RACING MULTIPLE PROMISES





# PROMISE IN THE WILD

## FETCH API

- » Promise based Browser API for HTTP requests
- » replaces/enhances XHR Request
- » based on promises

```
// Getting data via fetch
```

```
fetch('/api/users')  
  .then((httpResult) => httpResult.json())  
  //                      ^^^^^^^  
  // required as fetch resolves when HTTP headers are sent  
  .then((data) => console.log(data))
```

# PROMISE IN THE WILD

## POSTING DATA VIA FETCH

```
// Posting data via fetch
```

```
fetch('/api/users', {  
  method: 'POST', // specify HTTP method  
  headers: {  
    'Content-Type': 'application/json' // setting the content type  
  },  
  body: JSON.stringify({ my: 'payload' }) // adding payload  
})  
  .then((httpResult) => httpResult.json())  
  .then((data) => console.log(data))
```

# ASYNC/AWAIT

# ASYNC/AWAIT

- » Syntactic Sugar for Promises
- » 2 new keywords
  - » `async` marks a function to be `async`
  - » `await` pauses execution inside an `async` function
- » `await` can't be used outside an `async` function

# ASYNC/AWAIT

```
async function someAsyncFunction () {  
  // ^^  
  // async keyword defines an async function  
  // in here we can use await  
  
  const response = await myHTTPRequest()  
  //          ^^^^^  
  // wait until the http request is done and the promise resolves/rejects  
  console.log(response)  
}
```



# ASYNC/AWAIT PROMISE EXAMPLE

```
const fetchAsPromise = (url) => new Promise((resolve) => {  
  fetch(url, resolve)  
})
```

```
function bestFriendsAddress() {  
  return fetchAsPromise('/api/currentUser')  
    .then((currentUser) => fetchAsPromise(`/api/user/${currentUser.id}/bestFriend`))  
    .then((bestFriend) => fetchAsPromise(`/api/user/${bestFriend.id}/address`))  
    .then((bestFriendsAddress) => console.log(bestFriendsAddress))  
}
```

# ASYNC/AWAIT PROMISE EXAMPLE

```
const fetchAsPromise = (url) => new Promise((resolve) => {  
  fetch(url, resolve)  
})
```

```
async function bestFriendsAddress() {  
  const currentUser = await fetchAsPromise('/api/currentUser')  
  const bestFriend = await fetchAsPromise(`/api/user/${currentUser.id}/bestFriend`)  
  const bestFriendsAddress = await fetchAsPromise(`/api/user/${bestFriend.id}/address`)  
  console.log(bestFriendsAddress)  
}
```

# ASYNC/AWAIT PROMISE EXAMPLE

```
const fetchAsPromise = (url) => new Promise((resolve) => {
  fetch(url, resolve)
})

// define an async function
async function bestFriendsAddress() {
  const currentUser = await fetchAsPromise('/api/currentUser')
  //          ^^^^^
  // wait until the promise is resolved and continue once it is done
  const bestFriend = await fetchAsPromise(`/api/user/${currentUser.id}/bestFriend`)
  const bestFriendsAddress = await fetchAsPromise(`/api/user/${bestFriend.id}/address`)
  console.log(bestFriendsAddress)
}
```

# ASYNC/AWAIT ERROR HANDLING PROMISES

```
const fetchAsPromise = (url) => new Promise((resolve) => {  
  fetch(url, resolve)  
})
```

```
async function bestFriendsAddress() {  
  try {  
    const currentUser = await fetchAsPromise('/api/currentUser')  
    const bestFriend = await fetchAsPromise(`/api/user/${currentUser.id}/bestFriend`)  
    const bestFriendsAddress = await fetchAsPromise(`/api/user/${bestFriend.id}/address`)  
    console.log(bestFriendsAddress)  
  } catch (e) {  
    console.error('something went wrong')  
  }  
}
```

# ASYNC/AWAIT PITFALLS

# ASYNC/AWAIT PITFALLS

## `await` IN LOOPS

- » API is called sequentially
- » waits until promise is resolved before continuing
- » each iteration waits until the promise is done
- » user needs to wait longer until all data is present



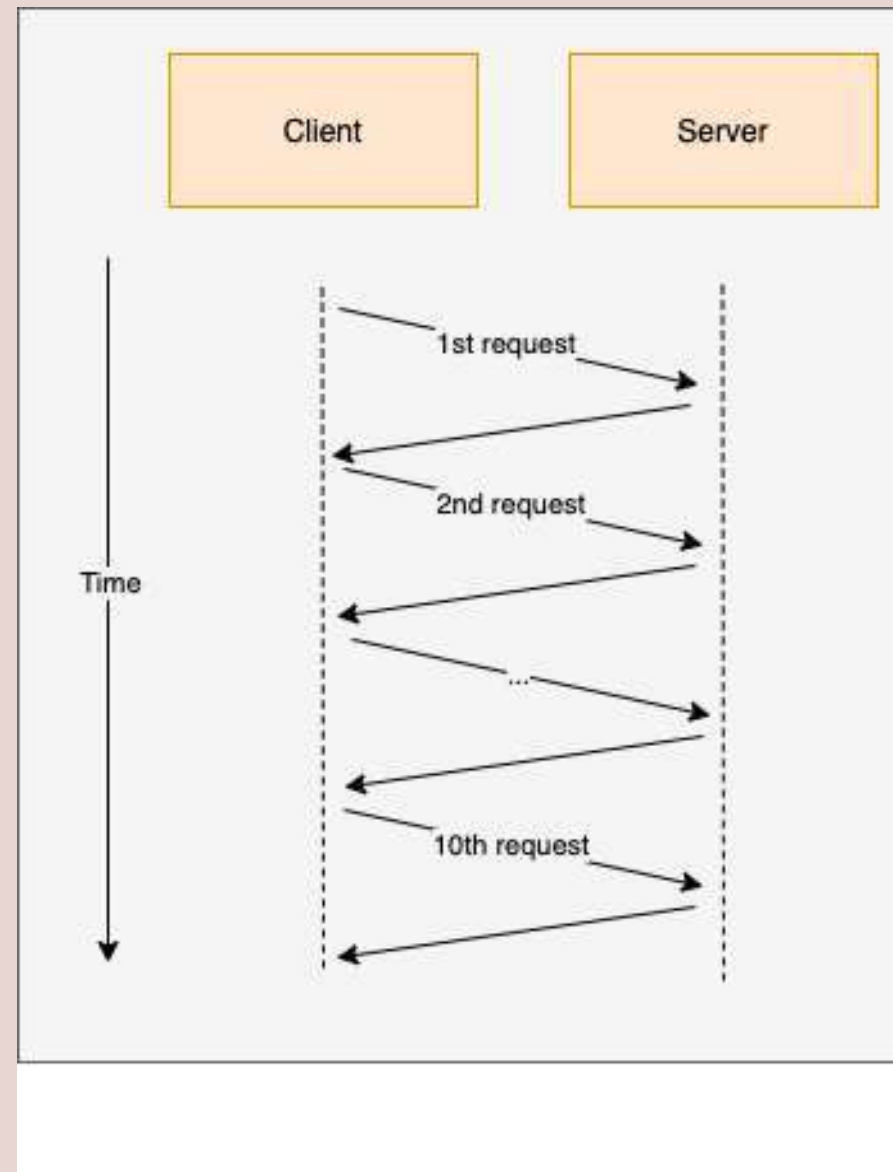
# ASYNC/AWAIT PITFALLS

## await IN LOOPS

```
async function awaitInLoops() {  
  for (let userId; userId < 10; userId++) {  
    const user = await fetchAsPromise(`/api/user/${userId}`)  
    //          ^^^^  
    // for each user we have to wait until http request is done  
    console.log(user)  
  }  
}
```

# ASYNC/AWAIT PITFALLS

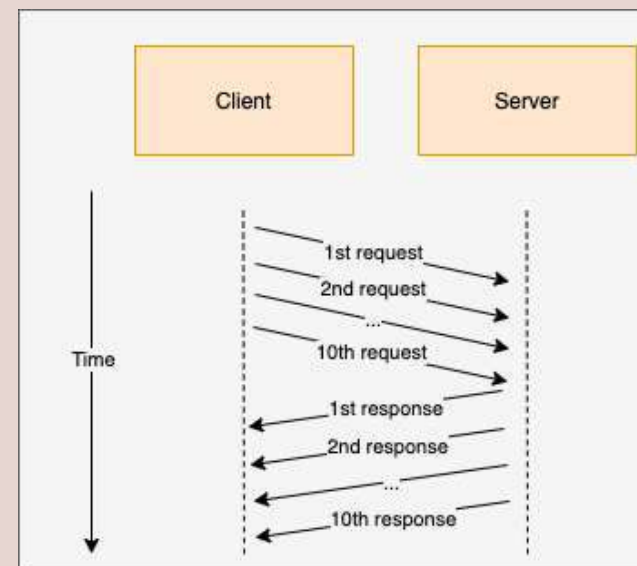
## `await` IN LOOPS



# ASYNC/AWAIT PITFALLS

## await IN LOOPS

- » `Promise.all` can fix this
- » requests to the backend are executed in parallel
- » client waits until all promises resolve



# ASYNC/AWAIT PITFALLS

## await IN LOOPS

```
async function awaitInLoops() {  
  const promises = [] // declare a list of promises  
  
  for (let userId; userId < 10; userId++) {  
    promises.push(fetchAsPromise(`/api/user/${userId}`))  
    // ^^^^^^^  
    // start the promise and add it to the promises  
  }  
  
  await Promise.all(promises)  
  // wait for all promises to finish  
}
```

# ASYNC/AWAIT PITFALLS

## ERROR HANDLING WITHOUT AWAIT

```
async function fetchBestFriend(id) {  
  try {  
    return fetchAsPromise(`/api/user/${id}/bestFriend`)  
    // ^^^^  
    // return promise without awaiting its result  
  } catch (e) {  
    console.error('something went wrong')  
    // ^^^^  
    // when promise fails this error handler won't be called  
  }  
}
```

# ASYNC/AWAIT PITFALLS

## ERROR HANDLING WITHOUT AWAIT

» Remember:

» A promise is a value which might be available in the future

```
async function fetchBestFriend(id) {  
  try {  
    return fetchAsPromise(`/api/user/${id}/bestFriend`)  
    // ^^^^^  
    // returning without an await makes the function return  
    // without waiting for the response and a possible error  
    // when the promise gets rejected the JS engine already  
    // went into a different execution context  
  } catch (e) {  
    console.error('something went wrong')  
    // ^^^^^  
    // when promise fails this error handler won't be called  
  }  
}
```



# ASYNC/AWAIT PITFALLS

## ERROR HANDLING WITHOUT AWAIT

```
async function fetchBestFriend(id) {  
  try {  
    return await fetchAsPromise(`/api/user/${id}/bestFriend`)  
    //      ^^^^  
    // we wait for the promise to be resolved inside the function  
    // and handle the error internally => catch block will be called  
  } catch (e) {  
    console.error('something went wrong')  
  }  
}
```

# ASYNC/AWAIT PITFALLS

## ERROR HANDLING WITHOUT AWAIT

```
async function fetchBestFriend(id) {  
  try {  
    return await fetchAsPromise(`/api/user/${id}/bestFriend`)  
    //      ^^^^  
    // we wait for the promise to be resolved inside the function  
    // and handle the error internally => catch block will be called  
  } catch (e) {  
    console.error('something went wrong')  
  }  
}
```

# **HOMEWORK**

» please see wiki

# FEEDBACK

» Questions: [tmayrhofer.lba@fh-salzburg.ac.at](mailto:tmayrhofer.lba@fh-salzburg.ac.at)

» [Feedback Link](#)