[Lesson 10-11]

Roi Yehoshua 2018

[What we learnt last time?]

- JSON format
- AJAX
- XHR
- jQuery.ajax()

[Our targets for today]

- Cookies
- Working with Promise
- Processing promise answers
- Async/Await construction
- Fetch API

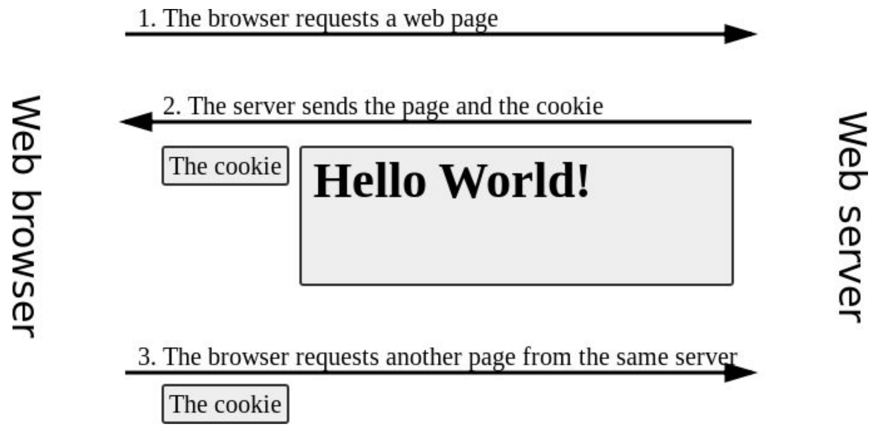
State Management

- HTTP is stateless
- In order to associate a request to any other request, you need a way to store user data between HTTP requests
- Cookies are used to transport data between the client and the server
- Sessions allow you to store information associated with the client on the server

Cookies	Session
Stored on the client side	Stored on the server side
Can only store strings	Can store objects
Can be set to a long lifespan	When users close their browser, they also lose the session

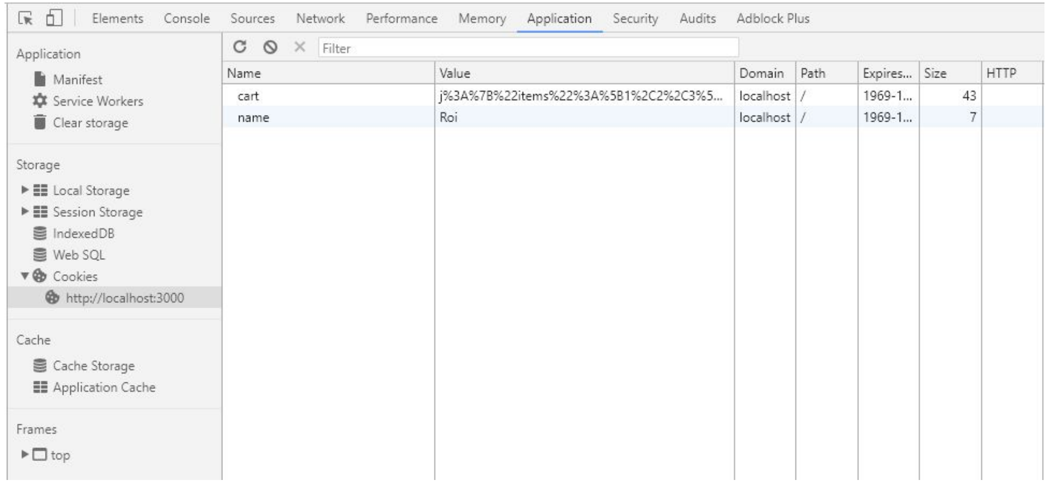
Cookies

→ Cookies are simple, small files/data that are stored on the client side
→ Every time the user loads the website back, this cookie is sent with the request
→ This helps us keep track of the user’s actions


1. The browser requests a web page
2. The server sends the page and the cookie
The cookie
Hello World!
3. The browser requests another page from the same server
The cookie
Web browser
Web server

Inspecting Cookies in Chrome Developer Tools

→ In the Google Chrome developer tools, you can view the cookies sent to the browser under the Application tab:



Name	Value	Domain	Path	Expires...	Size	HTTP
cart	j%3A%7B%22items%22%3A%5B1%2C2%2C3%5...	localhost	/	1969-1...	43	
name	Roi	localhost	/	1969-1...	7	

[Promise]

→ Often in programming we have a “producing code” that does something that needs time (e.g., load a remote script) and a “consuming code” that wants the result when it’s ready

→ A **promise** is a special JavaScript object that links them together

→ The producing code creates the promise and gives to everyone who needs the result, so that they can subscribe for the result

→ The constructor syntax for a promise object is:

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

→The function passed to new Promise is called **executor**

- →When the promise is created, the executor is called immediately
- →It contains the producing code, that should eventually finish with a result

[Promise]

→The resulting promise object has internal properties:

→**state** – initially is “pending”, then changes to “fulfilled” or “rejected”

→**result** – the result of the computation, initially undefined

→When the executor finishes its job, it should call one of:

→**resolve(value)** – to indicate that the job finished successfully:

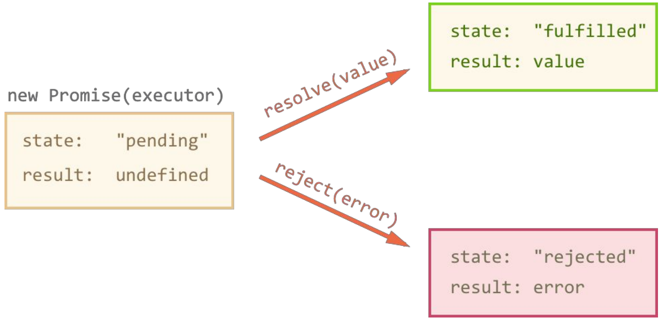
→ sets state to "fulfilled"

→ sets result to value

→**reject(error)** – to indicate that an error occurred:

→ sets state to "rejected"

→ sets result to error



Promise Example

→ An example for a simple executor:

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

→ And now an example where the executor rejects promise with an error:

```
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);
});
```

→ There can be only one result or an error

- →The executor should call only resolve or reject. The promise state change is final

→ Technically we can call reject (just like resolve) with any type of argument

- →It is recommended to use Error objects in reject

[Consumers: “.then” and “.catch”]

→ A promise object serves as a link between the producing code (executor) and the consuming functions – those that want to receive the result/error

→ Consuming functions can be registered using **promise.then()** and **promise.catch()**

→ The syntax of **.then** is:

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

→ Example:

```
let promise = new Promise(function (resolve, reject) {
    setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
    result => alert(result), // shows "done!" after 1 second
    error => alert(error) // doesn't run
);
```

Consumers: “.then” and “.catch”

→ If we’re interested only in successful completions, then we can provide only one argument to .then:

```
let promise = new Promise(resolve => {
     setTimeout(() => resolve("done!"), 1000);
});

promise.then(alert); // shows "done!" after 1 second
```

→ If we’re interested only in errors, then we can use .then(null, function) or an “alias” to it: **.catch(function)**:

```
let promise = new Promise((resolve, reject) => {
     setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

Example: loadScript

→ For instance, take a look at the function loadScript(src):

```
function loadScript(src) {
    let script = document.createElement('script');
    script.src = src;
    document.head.append(script);
}
```

→The purpose of this function is to load a new script

→When <script src="…"> is added to the document, the browser loads the script asynchronously and executes it

→ We'd like to know when the script has finished loading, as when to use new functions and variables from that script

→ For that purpose, we can add a callback function as a second argument to loadScript() that should execute when the script loads:

[Example: loadScript]

```
function loadScript(src, callback) {
    let script = document.createElement('script');
    script.src = src;
    script.onload = () => callback(script);

    document.head.append(script);
}
```

→ Now if we want to call new functions from the script, we should write them in the callback:

```
loadScript('/my/script.js', function () {
    // the callback runs after the script is loaded
    newFunction(); // so now it works
    ...
});
```

[Example: loadScript]

→Let’s rewrite it using promises

- → The new function loadScript() will not require a callback
- → Instead it will create and return a promise object that settles when the loading is complete

```
function loadScript(src) {
    return new Promise(function (resolve, reject) {
        let script = document.createElement('script');
        script.src = src;
        script.onload = () => resolve(script);
        script.onerror = () => reject(new Error("Script load error: " + src));
        document.head.append(script);
    });
}

// Usage:
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js");
promise.then(
    script => alert(`${script.src} is loaded!`),  error => alert(`Error: ${error.message}`)
);
promise.then(script => alert('One more handler to do something else!'));
```

[Callbacks vs. Promises]

→ We can immediately see few benefits of promises over the callback-based syntax:

Callbacks	Promises
We must have a ready callback function when calling loadScript. In other words, we must know what to do with the result *before* loadScript is called.	Promises allow us to code things in the natural order. First we run loadScript, and .then write what to do with the result.
There can be only one callback.	We can call .then on a promise as many times as we want, at any time later.

Exercise (1)

→ The built-in function setTimeout() uses callbacks

→ Create a promise-based alternative

→ The function **delay(ms)** should return a promise

→ That promise should resolve after ms milliseconds,

so that we can add .then to it:

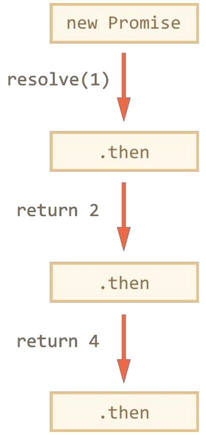
```
function delay(ms) {
    // your code
}

delay(3000).then(() => alert('runs after 3 seconds'));
```

[Promises Chaining]

→ **Promises chaining** allows you to have a sequence of asynchronous tasks to be done one after another (e.g., for loading scripts)

```
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;
});
```



→ The idea is that the result is passed through the chain of .then handlers

→ This works because a call to promise.then() returns a promise, so that we can call the next .then() on it

[Returning Promises]

→ Normally, a value returned by a .then handler is immediately passed to the next handler

→ However, if the returned value is a promise, then the further execution is suspended until it settles. After that, the result of that promise is given to the next .then handler

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

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

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

The output is again 1 → 2 → 4, but now with 1 second delay between alert calls

[Example: loadScript]

→ Let’s use this feature with loadScript to load scripts one by one, in sequence:

```
loadScript("/promises/one.js")
    .then(function (script) {
        return loadScript("/promises/two.js");
    })
    .then(function (script) {
        return loadScript("/promises/three.js");
    })
    .then(function (script) {
        one();
        two();
        three();
    });
```

→ Here each loadScript call returns a promise, and the next .then runs when it resolves

→ Then it initiates the loading of the next script. So scripts are loaded one after another

→ Note that the code is still “flat”, it grows down, not to the right (as opposed to using callbacks)

[DAN.IT] EDUCATION

[async/await]

→ The newest way to write asynchronous code in JavaScript (from ES7)

→ Async/await simplify the process of working with promises

→ **Async functions** always return a Promise

- → If the function throws an error, the Promise will be rejected
- → If the function returns a value, the Promise will be resolved

```
async function func() {
    return 1; // the same as: return Promise.resolve(1);
}

func().then(alert); //
```

- → The word “async” before a function means that the function always returns a promise
- → If the code has return <non-promise> in it, JS automatically wraps it into a resolved promise

[await]

→ The keyword await makes JS wait until the promise settles, and returns its result

→That doesn’t cost any CPU resources, because the engine can do other jobs meanwhile

→It’s just a more elegant syntax of getting the promise result than promise.then

→ await can be used only inside async functions

```
async function f() {
    let promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve("done!"), 1000)
    });
    let result = await promise; // wait till the promise resolves

    alert(result); // "done!"
}

f();
```

[Async Methods]

→ A class method can also be async, just put async before it:

```
class Waiter {
    async wait() {
        return await Promise.resolve(1);
    }
}

new Waiter()
    .wait()
    .then(alert); // 1
```

→ The meaning is the same: it ensures that the returned value is a promise and enables await

Error Handling

→ In case that the promise was rejected, await throws the error, just if there were a throw statement at that line

```
async function f() {
    await Promise.reject(new Error("Whoops!"));
    // the same as:
    // throw new Error("Whoops!");
}
```

→ We can catch that error using try..catch, the same way as a regular throw:

```
async function f() {
    try {
        let response = await fetch('http://no-such-url');
    } catch (err) {
        alert(err); // TypeError: failed to fetch
    }
}
f();
```

[Error Handling]

→ If we don’t have try..catch, then the promise generated by the call of the async function f() becomes rejected

→ We can append .catch to handle it:

```
async function f() {
    let response = await fetch('http://no-such-url');
}

// f() becomes a rejected promise
f().catch(alert); // TypeError: failed to fetch
```

→ If we forget to add .catch there, then we get an unhandled promise error (and can see it in the console)

→ Note that at the top level of the code, when we’re outside of any async function, we’re syntactically unable to use await, so it’s a normal practice to add .then/catch to handle the final result or falling-through errors

[Exercise (2)]

→ Rewrite the following code using async/await instead of .then:

```
function resolveAfter2Seconds() {
    return new Promise(resolve =>
        {  setTimeout(() => {
             resolve('resolved');
        }, 2000);
    });
}

function asyncCall() {
    console.log('calling');
    resolveAfter2Seconds().then(result =>
    {
        console.log(result);
        // expected output: "resolved"
    });
}

asyncCall();
```

[Fetch]

- ➔ Fetch API is an interface for fetching data
- ➔ It replaces older XMLHttpRequest API, providing more options and more powerful workflow
- ➔ Fetch API is implemented by window object and also available from the web workers
- ➔ Fetch requests returns a promise that resolves to the Response object
- ➔ Response object contains data stream that can be read as data types like text, JSON, blob, and others
- ➔ Fetch requests does not send credentials cookies by default, so they must be provided explicitly

[DAN.IT]
EDUCATION

[Fetch]

- ➔ Fetch can be done by the fetch() method
- ➔ It receives two arguments:
  - → Input - USVString or a Request object
  - → Init - optional argument with custom settings like method, headers, body and others
  - → Init settings will override request object settings

[Request]

- ➔ Request object can be created via Request constructor
- ➔ It receives two arguments (just like a fetch() method itself):
  - → Input - USVString or a Request object
  - → If another Request object is provided - a copy of it will be created
  - → Init - optional argument with custom settings like method, headers, body and others

[Init settings]

- ➔ Init settings of the fetch method or request constructor contains connections params related to the fetch requests like a request method, caching, credentials and others
- ➔ Some of this settings are:
  - → Method - request method like GET, POST, PUT and others
  - → Headers - additional information to the server like content-type, authorization, cors-related headers and others
  - → Body - body of the request. Can be blob, FormData, USVString and others
  - → Mode - mode of the request like cors, no-cors or same-origin
  - → Credentials - settings of send request credentials, can be omit, same-origin or include. Must be provided in order for the cookies to be send.
  - → Cache - cache mode that will be used for the request.

[Headers]

- ➔ Headers object can be created via headers constructor
- ➔ It can be than attached to the fetch() method or a request object init settings object
- ➔ Not all headers are available in headers object for the fetch api for the security reasons. Such headers are: Cookie, Date, Keep-Alive, Origin, Referer and others
- ➔ Formation of headers could be done via headers object methods
- ➔ Some of them are:
  - → Set - sets a new value for the headers object
  - → Get - returns a specified header
  - → Delete - removes a header
  - → Has - returns true if the header exists

[DAN.IT] EDUCATION

[Headers examples]

```
const myHeaders = new Headers();

//adds a new header
myHeaders.set('Content-Type', 'text/xml');

//returns a specified header
myHeaders.get('Content-Type');

//deletes a specified header
myHeaders.delete('Content-Type');

//returns false, since Content-Type header is deleted
myHeaders.has('Content-Type');
```

[DAN.IT]
EDUCATION

[Fetch examples]

```
fetch("header.jpg").then(...)
```

```
const myInit = {
   method: 'GET',
   cache: 'default'
};
const myRequest = new Request('header.jpg', myInit);
fetch(myRequest).then(...)
```

```
const myHeaders = new Headers();
myHeaders.append('Content-Type', 'image/jpeg');

const myRequest = new Request('header.jpg');

const myInit = {
   method: 'GET',
   headers: myHeaders,
   mode: 'cors',
   cache: 'default'
};


fetch(myRequest, myInit).then(...)
```

[Response]

- ➔ Result of the fetch request is a promise that resolves to the response object
- ➔ It is also possible to create response object via Response constructor
- ➔ Response object contains information about response like status code, status message or associated headers
- ➔ It also contains methods for operations with response object itself and reading an answer stream data

[Response]

- ➔ Response properties are:
  - → Response.status - Contains the status code of the response
  - → Response.statusText - Contains the status message corresponding to the status code
  - → Response.url - Contains the URL of the response
  - → Response.headers - Contains the Headers object associated with the response
  - → Response.ok - Contains a boolean stating whether the response was successful (status in the range 200-299) or not.
  - → Response.redirected - Indicates whether or not the response is the result of a redirect
  - → Response.type - Contains the type of the response (basic, cors)
  - → Response.useFinalURL - Contains a boolean stating whether this is the final URL of the response
  - → Response.body - A simple getter used to expose a ReadableStream of the body contents
  - → Response.bodyUsed - Stores a Boolean that declares whether the body has been used in a response yet

[Response]

- ➔ General methods:
  - → Response.clone() - Creates a clone of a Response object
  - → Response.error() - Returns a new Response object associated with a network error
  - → Response.redirect() - Creates a new response with a different URL

- ➔ Method for reading stream data:
  - → Body.arrayBuffer() - Takes a Response stream and reads it to completion. It returns a promise that resolves with an ArrayBuffer
  - → Body.blob() - Takes a Response stream and reads it to completion. It returns a promise that resolves with a Blob
  - → Body.formData() - Takes a Response stream and reads it to completion. It returns a promise that resolves with a FormData object
  - → Body.json() - Takes a Response stream and reads it to completion. It returns a promise that resolves with the result of parsing the body text as JSON
  - → Body.text() - Takes a Response stream and reads it to completion. It returns a promise that resolves with a USVString (text)

[Response examples]

```
fetch(myRequest)
   .then(response => response.json())
   .then(data => {
       for (let i = 0; i < data.products.length; i++) {
           const listItem = document.createElement('li');
           listItem.innerHTML = `<strong>${data.products[i].Name}</strong> Cost: <strong>£${data.products[i].Price}</strong>`;
           myList.appendChild(listItem);
       }
   }
);
```

```
function getText(pageId) {
   const myRequest = new Request(pageId + '.txt');
   fetch(myRequest)
       .then(response => response.text())
       .then(text => {
             myArticle.textContent = text;
       });
}
```

Axios

- Axios is a library for both client and node.js server for making HTTP requests
- It is built over XMLHttpRequest on the client and on HTTP from node.js
- Axios requests return promise
- It supports manipulations with requests like interception and transformation
- It can automatically transform data into JSON
- Axios provides buildin protection against XSRF attacks

[Installation]

- ➔ Axios can be installed via npm
  → `npm install axios`

- ➔ Via bower
  → `bower install axios`

- ➔ Or via CDN
  → `<script src="https://unpkg.com/axios/dist/axios.min.js"></script>`

[Making request]

- ➔ Axios Config object can specify request params including URL
- ➔ It can be provided to the axios method to make a request
- ➔ But for convenience axios contains many aliases than can instead take only URL and set the rest depending on predefined defaults

General request methods:

- ➔ axios(config) - most basic usage, accepts config object.
- ➔ axios(url[, config]) - only URL and optional partial config object. By default will make a GET request
- ➔ axios.request(config) - alias of axios(config)

[Convenience methods]

- ➔ These methods require only URL param and data argument for some request methods
- ➔ Optional config object can also be provided, but request params required to make corresponding request method will be set automatically, so there is no need to set every param by yourself

[Convenience methods]

All of this methods makes a corresponding HTTP method request

- ➔ axios.get(url[, config])
- ➔ axios.delete(url[, config])
- ➔ axios.head(url[, config])
- ➔ axios.options(url[, config])
- ➔ axios.post(url[, data[, config]])
- ➔ axios.put(url[, data[, config]])
- ➔ axios.patch(url[, data[, config]])

[DAN.IT] EDUCATION

[Concurrency methods]

And finally concurrency methods for working with promises

- ➔ axios.all(iterable)
- ➔ axios.spread(callback)

[Examples]

```
//basic request
axios.get('/user?ID=12345')
   .then(response => {
       // handle success
       console.log(response);
   })
   .catch(error => {
       // handle error
       console.log(error);
   })
   .then(() => {
       // always executed
   });
```

```
//post request
axios.post('/user', {
   firstName: 'Fred',
   lastName: 'Flintstone'
})
   .then(response => {
       console.log(response);
   })
   .catch(error => {
       console.log(error);
   });
```

[Concurrency examples]

```
//waiting for both requests to complete
function getUserAccount() {
   return axios.get('/user/12345');
}

function getUserPermissions() {
   return axios.get('/user/12345/permissions');
}

axios.all([getUserAccount(), getUserPermissions()])
   .then(axios.spread((acct, perms) => {
       // Both requests are now complete
   }));
```

Handling errors example

```
axios.get('/user/12345')
   .catch(error => {
       if (error.response) {
           // The request was made and the server responded with a status code
           // that falls out of the range of 2xx
           console.log(error.response.data);
           console.log(error.response.status);
           console.log(error.response.headers);
       } else if (error.request) {
           // The request was made but no response was received
           // `error.request` is an instance of XMLHttpRequest in the browser and an instance of
           // http.ClientRequest in node.js
           console.log(error.request);
       } else {
           // Something happened in setting up the request that triggered an Error
           console.log('Error', error.message);
       }
       console.log(error.config);
   });
```

Axios Config

- ➔ Axios config is an object filled with optional params to describe the desired request
- ➔ It can be provided into methods to configure the request
- ➔ It is possible to set a user default config that will apply to any axios request that does not override default params
- ➔ It can be done by setting params of axios.defaults object
- ➔ The final config of the axios request will be a combination of axios library defaults, user defaults and finally the optional request config
- ➔ Config will be merged with an order of precedence. The latter will take precedence over the former

Common config params

- url - server URL of the request. Can be both relative and absolute. for example `/user` and `https://example.com/api/user`
- method - method of the request, like GET, POST, DELETE
- baseURL - this param will prepend the url param, unless url is an absolute. For example you can set baseURL to the `https://example.com/api/` and url just to `/user`
- headers - object containing request headers. example - {'Content-Type': 'text/xml'}
- params - is an object containing URL parameters to be sent with the request. So setting it to {id: 12345} will result in url ending in ?id=12345

[Common config params]

- ➔ data - data to be sent as the request body. Only applicable for request methods 'PUT', 'POST', and 'PATCH'.
- ➔ timeout - number of milliseconds before the request times out. If the request takes longer than `timeout`, the request will be aborted
- ➔ withCredentials - indicates whether or not cross-site Access-Control requests should be made using credentials
- ➔ auth - indicates that HTTP Basic auth should be used, and supplies credentials. This will set an `Authorization` header, overwriting any existing `Authorization` custom headers you have set using `headers`.
- ➔ validateStatus - defines whether to resolve or reject the promise for a given. HTTP response status code. If `validateStatus` returns `true` (or is set to `null` or `undefined`), the promise will be resolved; otherwise, the promise will be rejected.
- ➔ cancelToken - specifies a cancel token that can be used to cancel the request

[Advanced config params]

- ➔ transformRequest - allows changes to the request data before it is sent to the server
- ➔ transformResponse - allows changes to the response data to be made before
- ➔ paramsSerializer - optional function in charge of serializing `params`
- ➔ adapter - allows custom handling of requests which makes testing easier
- ➔ xsrfCookieName - is the name of the cookie to use as a value for xsrf token
- ➔ xsrfHeaderName - is the name of the http header that carries the xsrf token value
- ➔ responseType - indicates the type of data that the server will respond with. options are 'arraybuffer', 'blob', 'document', 'json', 'text', 'stream'

[Advanced config params]

- ➔ responseEncoding - indicates encoding to use for decoding responses
- ➔ maxContentLength - defines the max size of the http response content in bytes allowed
- ➔ onUploadProgress - allows handling of progress events for uploads
- ➔ onDownloadProgress - allows handling of progress events for downloads
- ➔ maxRedirects - defines the maximum number of redirects to follow in node.js. If set to 0, no redirects will be followed
- ➔ socketPath - defines a UNIX Socket to be used in node.js
- ➔ httpAgent - define a custom agent to be used when performing http and https requests, respectively, in node.js. This allows options to be added like `keepAlive` that are not enabled by default.
- ➔ httpsAgent - same as httpAgent but for https protocol
- ➔ proxy - defines the hostname and port of the proxy server

[DAN.IT] EDUCATION

[Config examples]

```
//Very basic config
{
   method: 'get',
   url: 'http://example.com',
   responseType: 'stream'
}
```

```
//Config object with authorization token attached
to the headers
{
   method: 'get',
   baseURL: 'http://example.com/',
   url: '/login',
   headers: {
       Authorization: `Bearer ${token}`
   }
}
```

[DAN.IT] EDUCATION

Default Config Example

```
//Setting base URL for all request that uses only relative URL
axios.defaults.baseURL = 'https://api.example.com';

//Setting a default authorization token
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;

//Setting post request to send data in a html form format instead of JSON
axios.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded';
```

Response

Response object resolved from the promise will be the following scheme:

- ➔ data - response that was provided by the server
- ➔ status - HTTP status code from the server response
- ➔ statusText - HTTP status message from the server response
- ➔ headers - headers that the server responded with
- ➔ config - config that was provided to `axios` for the request
- ➔ request - request that generated this response. It is the last ClientRequest instance in node.js (in redirects) and an XMLHttpRequest instance in the browser

[Response example]

```
axios.get('/user/12345')
   .then(response => {
       console.log(response.data);
       console.log(response.status);
       console.log(response.statusText);
       console.log(response.headers);
       console.log(response.config);
   });
```

```
axios.get('/post/server', JSON.parse(data))
   .then(response => {
       output.className = 'container';
       output.innerHTML = response.data;
   })
   .catch(err => {
       output.className = 'container text-danger';
       output.innerHTML = err.message;
   });
```

[Interceptors]

- ➔ Interceptor is an axios function that process responses and requests before they are send to their user declared handlers

- ➔ They have many uses such as
  - → Authentication and security management
  - → Operations with routing
  - → Logging
  - → UI events like firing a loading progression animation and ending it

[Interceptors]

```
// Add a request interceptor
axios.interceptors.request.use( config => {
   // Do something before request is sent

   return config;
}, error => {
   // Do something with request error

   return Promise.reject(error);
});
```

```
// Add a response interceptor
axios.interceptors.response.use(response => {
   // Do something with response data

   return response;
}, error => {
   // Do something with response error

   return Promise.reject(error);
});
```

```
//If you may need to remove an interceptor later you can use the eject method

const myInterceptor = axios.interceptors.request.use(function () {/*...*/});
axios.interceptors.request.eject(myInterceptor);
```

[Control questions]

1. What usages do web cookies have?
1. What is Promise?
2. Name Promise internal properties
3. Name Promise object methods
4. What advantages Promise have over Callback?
5. What is async/await?
6. Does async/await makes Promises obsolete?
7. How can you process errors with async/await construction?
8. What is Fetch api and how to work with it?
9. Name prominent features of Axios