Front-End Development

OPO

Syllabus

Document

1.0

Version

Leen Van Houdt

Author(s)

Academic year 2023-2024

PREFACE

The following styles and conventions are used in this syllabus.

Additional information.

Idea, example or tip.

Warning, important information.

Deep-dive.

Question to the reader.

Objective.

Exercise, task or assignment for the reader.

Additional material or reference.

TABLE OF CONTENT

1 Introduction

This syllabus was developed as part of the Front-End Development course. In this course we use a range of materials to teach you the basics of front-end development. Previously, you have been learning about HTML, CSS and a little bit of JavaScript by following along on the website. This syllabus will pick up where the website left off and covers additional material. To be able to follow along with this text, you will have to finish reviewing the contents of the website first, including the sections on JavaScript, as these won't be repeated here.

In this syllabus, we will focus on JavaScript concepts that weren't included on the website: objects, arrays, and functions. We will also introduce you to the asynchronous web model and discuss how to handle errors. This text serves as a theoretical overview of these concepts. On Toledo you will find accompanying videos that demonstrate how to apply them. Throughout the syllabus you will also find links to external sources where you can find additional information and examples.

For a full overview of all course materials, please refer to the Toledo course.

2 Objects and arrays

2.1 Objects

In this course we have previously discussed data types, such as strings and numbers. Most data types in JavaScript are primitive, meaning their values only contain one thing such as a string, number, or Boolean. Objects differ from these primitive types because they can be used to store collections of multiple values. Each of these values has a specific key.

There are two ways to declare an empty object, of which the second one is most commonly used:

```
let object = new Object();          // "object constructor" syntax
let object = {};                    // "object literal" syntax
```

An object can be created with an optional list of properties. A property is a *key: value* pair, where the key is a string that can be used to store, retrieve, and alter the value. The value of a property can be anything: it can be a simple string or number, but also another object or a function.

Here's an example of an object being declared with a list of properties and subsequently accessed using its keys:

```
const identity = { name: 'John', age: 45 }
console.log(`Hello ${identity.name} who is ${identity.age} years old.`)
```

It is also possible to **destructure** the properties of an object into separate variables. This means we unpack the object and assign its properties to new variables, like so:

```
const identity = { name: 'John', age: 45 }
const { name, age } = identity
console.log(`Hello ${name} who is ${age} years old.`)
```

https://javascript.info/object

2.2 Arrays

We've previously discussed objects, which are collections of keyed values. However, sometimes we want the values in a collection to be numbered in a specific order. JavaScript has a special data structure to store ordered collections: *Array*. An array contains a numbered collection of values, so that the order of elements can be managed. In this section we will discuss how to declare and manipulate arrays.

2.2.1 Creating arrays

There are two ways to create an empty array in JavaScript. The second one is most commonly used:

```
let persons = new Array();
let persons = [];            // most common method
```

Upon creation the array can be initialized with elements of any type:

```
let persons = ["John", "Frank", "Annie"];     //array of strings
let persons = [{name: "John", age: 45},
           {name: "Annie", age: 23}]          //array of objects
let array = ["Apple",
           {name: "John"},
           true,
           function() { console.log("hello"); } ];  // mix of values
```

An array is an ordered collection, meaning array elements are numbered starting with 0. Elements can be retrieved by index. The total count of elements in an array is its *length*.

```
let persons = ["John", "Frank", "Annie"];
console.log(persons[0]);     // John
console.log(persons[1]);     // Frank
console.log(persons[2]);     // Annie
console.log(persons.length); // 3
```

The index can also be used to replace elements:

```
let persons = [“John”, “Frank”, “Annie”];
persons[0] = “Hannah”
console.log(persons);  // “Hannah”, “Frank”, “Annie”
```

Or to add a new element to the array:

```
let persons = [“John”, “Frank”, “Annie”];
persons[3] = “Martha”
console.log(persons);  // “John”, “Frank”, “Annie”, “Martha”
persons[persons.length] = “Jane”
console.log(persons);  // “John”, “Frank”, “Annie”, “Martha”, “Jane”
```

Note that *persons.length* will return the count of elements in the array (which in the above example equals 4, after adding “Martha”). Because arrays are numbered starting with 0, using *persons[persons.length] =* … will add an element at the end of the array.

2.2.2 Manipulating arrays

The following operations can be used to manipulate arrays.

Push: add an item to the end of the array.

```
let persons = [“John”, “Frank”, “Annie”];
persons.push(“Martha”) // result: “John”, “Frank”, “Annie”, “Martha”
```

Pop: remove the last item from the array.

```
let persons = [“John”, “Frank”, “Annie”];
let last = persons.pop();
console.log(last)           // “Annie”
console.log(persons)        // “John”, “Frank”
```

Shift: extract first element.

```
let persons = [“John”, “Frank”, “Annie”];
let first = persons.shift();
console.log(first)          // “John”
```

```
console.log(persons)          // “Frank”, “Annie”
```

Unshift: add element to the beginning.

```
let persons = [“John”, “Frank”, “Annie”];
persons.unshift(“Martha”);
console.log(persons)          // “Martha, “John”, “Frank”, “Annie”
```

Try it out yourself: define an array and test out the operations above. Think about what the array will look like after each operation and execute them to see if you’re right.

2.2.3 Iterating arrays

There are several ways to iterate over the elements of an array. One of the oldest ways is by using a *for* loop to loop over the indexes of the array:

```
const persons = [“John”, “Frank”, “Annie”];
for(let i=0; i<persons.length; i++) {
  console.log(persons[i]); // “John”, “Frank”, “Annie”
}
```

Another form of loop that can be used for arrays is *for … of …:*

```
const persons = [“John”, “Frank”, “Annie”];
for(const person of persons) {
  console.log(person); // “John”, “Frank”, “Annie”
}
```

Two modern ways of iterating arrays are by using the *filter(…)* and *forEach(…)* higher-order functions. These will be discussed in the next chapter on functions.

2.2.4 Spreading and destructuring arrays

“Spread” elements of one array into another one:

```
let men = [“John”, “Frank”];
let women = [“Annie”, “Martha”];
let persons = […men, …women]
     // result persons: [“John”, “Frank”, “Annie”, “Martha”]
```

The “…” takes all elements of the array and pushes them into another one. This is a **common** pattern in JavaScript.

Destructuring: unpack elements out of an array and assign them to a variable.

```
let persons = [“John”, “Frank”, “Annie”];
const [one, two, three] = persons;
console.log(one);      // “John”
console.log(two);      // “Frank”
console.log(three);    // “Annie”
```

Array destructuring and spreading can be combined to add the remaining elements to a new array:

```
let persons = [“John”, “Frank”, “Annie”];
const [one, …rest] = persons;
console.log(one);      // “John”
console.log(rest);     // [“Frank”, “Annie”]
```

JavaScript detects more elements than it can destruct in variables, so all the remaining elements are pushed in a new array called “rest”.

This chapter on arrays is partly based on https://javascript.info/array, which contains more in depth reading.

3 Functions

If you need to execute a specific action multiple times in your script, it's useful to turn it into a function. Functions are the building blocks of our code and can be called repeatedly throughout the program. This way we don't have to write duplicate code and can instead reuse the same function every time.

A function is created with a function declaration. This is the classic way of declaring a function:

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

console.log(calculateSum(2,4));    // 6
```

This function takes two arguments *a* and *b*, evaluates the expression *a + b*, and returns the result.

Try to use JavaScript as a "functional" language and express your functionality with functions (even if they are only one line).

Read more about the basics of functions: https://javascript.info/function-basics

3.1 Arrow functions

Instead of declaring functions in the classic way, there is also a more modern way of creating functions: by using **arrow functions**. Arrow functions have a more simple and concise syntax. Let's rewrite the previous example as an arrow function:

```
const calculateSum = (a, b) => {
  return a + b;
}

console.log(calculateSum(2,4));    // 6
```

As you can see, the function accepts the bracketed arguments *a* and *b*, and returns *a + b* as its output. Compare this example with the previous classic function declaration and note the differences in syntax.

Arrow functions are especially concise if a function only contains one line. In this case you can use the shorthand notation and omit the curly brackets:

```
const calculateSum = (a, b) => a + b;

console.log(calculateSum(2,4));    // 6
```

When a function only takes one argument, we can omit the brackets around the arguments:

```
const double = n => n * 2;

console.log(double(3));      // 6
```

If the function doesn't take any arguments, we have to write the brackets but they will be empty, like so:

```
const hello = () => console.log("Hello world");

hello();   // "Hello world"
```

Read more about arrow functions: https://javascript.info/arrow-functions-basics

3.2 First-class functions

In JavaScript, functions are first-class functions: they can be treated like variables. This means the function can be passed as an argument to another function, can be returned as the result of a function, and can be assigned to another variable. For example:

```
const calculateSum = (a, b) => a + b;

let other = calculateSum;    // assign function to another variable
console.log(other(2,4))            // 6
console.log(calculateSum(2,1))     // 3
```

https://developer.mozilla.org/en-US/docs/Glossary/First-class_Function

3.3 Higher-order functions

A higher-order function is a function that leverages other functions by either receiving them as an argument or returning them as a result.

Here's an example of passing functions as argument to a higher-order function:

```
const sum = (a, b) => a + b;
const multiply = (a, b) => a * b;

const calculate = (a, b, operation) => "Result: " + operation(a,b);

let res = calculate(2, 3, sum); // "Result: 5"
res = calculate(2, 3, multiply); // "Result: 6"
```

In the previous chapter on arrays, we briefly mentioned the *forEach(…)* and *filter(…)* functions as modern ways to iterate over the elements of an array. Both methods are higher-order functions, because they take a function as argument.

Looping over arrays with the *forEach* higher-order function can be done like this:

```
const persons = [“John”, “Annie”];

const greet = (person) => console.log(“Hello ” + person);

// Execute greet function for every element in array
persons.forEach(greet);      // “Hello John”, “Hello Annie”

// Inline as an anonymous function
persons.forEach((person) => console.log(“Hello ” + person))
```

We can filter arrays with the *filter* higher-order function:

```
const persons = [“John”, “Annie”, “Martha”];

// Only retain persons whose length is more than 4 characters
const filtered = persons.filter((person) => person.length > 4);

console.log(filtered)  // “Annie”, “Martha”
```

4 Asynchronous web model

4.1 Synchronous versus Asynchronous

Up until now, all code examples in this syllabus have been **synchronous**. This means every line of code is executed in the order that it's written in. In the synchronous model, a line of code must be completed before moving on to the next one. However, sometimes we will use functions in our code that might take a lot of time to execute. In this case it might be advantageous to work **asynchronously**. In asynchronous programming, operations can be executed simultaneously or in a different order than the one they were written in. This way, operations that are written after calling a time-consuming function don't have to wait for this function to finish executing.

Further reading on the differences between synchronous and asynchronous programming:
https://www.mendix.com/blog/asynchronous-vs-synchronous-programming/

One of the most common uses of asynchronous programming in JavaScript is retrieving something from a server or database. Say we were to program a website that makes requests to a server using synchronous programming. This means the client must wait for the server to process the request and send a response back without being able to do anything else, resulting in wasted time. Whenever the user refreshes the page, a new request will be made to the server, resulting in extra server processing, higher bandwidth consumption because of redundant page refreshes, and time lost while waiting for a response. For this use case, it would be far more efficient to use asynchronous programming, as shown in Image 1.

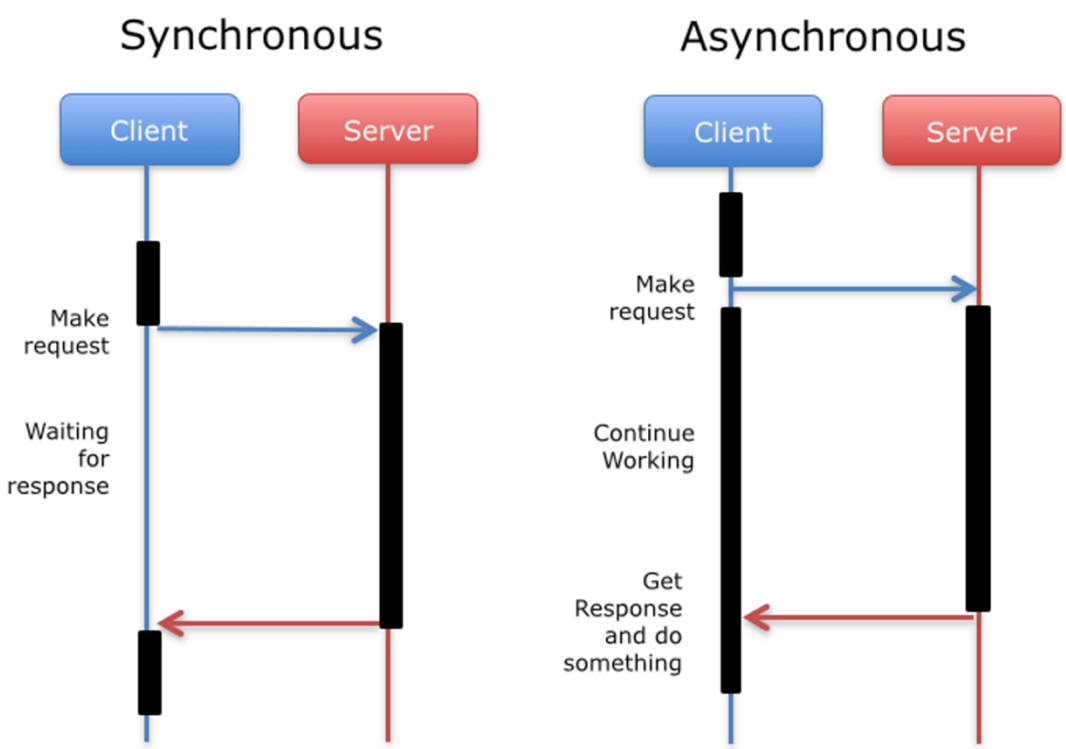

Image 1: comparison of synchronous and asynchronous client-server communication

The asynchronous web model introduces the idea of a partial screen update. This means only the user interface elements that contain new information will be updated, and the rest of the user interface will remain unchanged. This pattern is commonly referred to as **AJAX (Asynchronous JavaScript And XML)**, which is a group of technologies that makes asynchronous server calls possible. Image 2 shows a diagram of the AJAX web model. A similar and more modern technology to fetch data asynchronously is by using the **Fetch API**, which we will discuss further in this chapter.

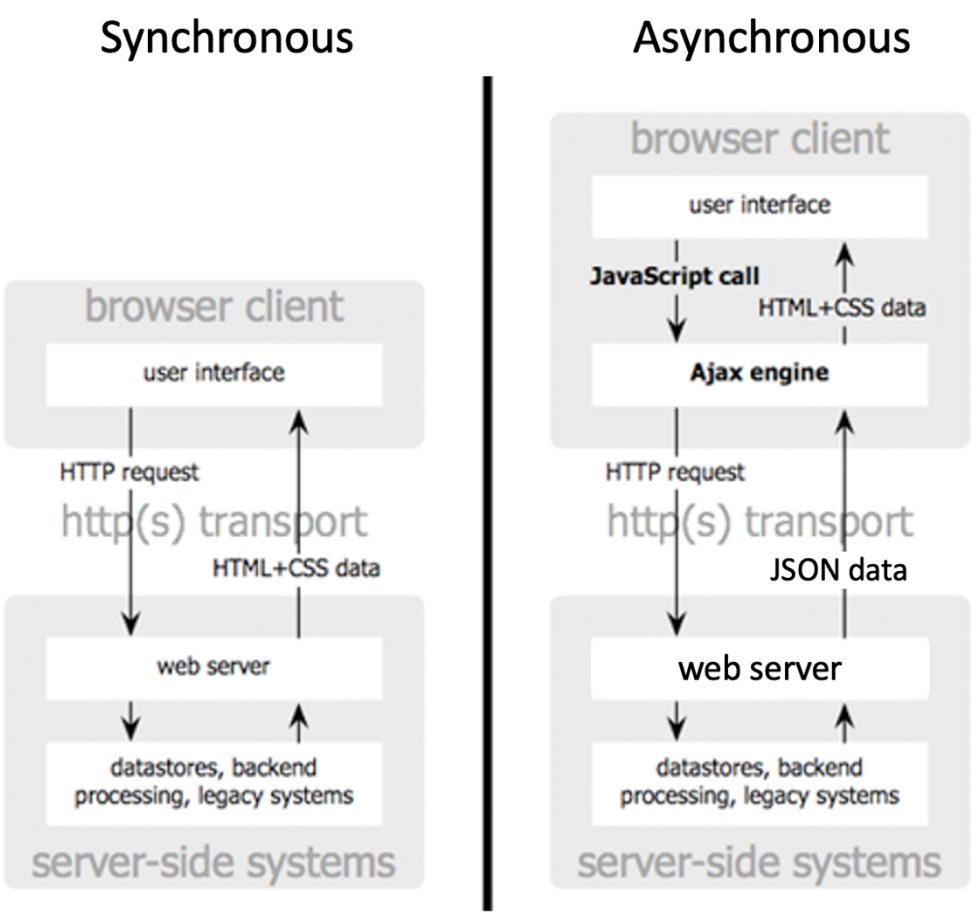

Image 2: AJAX web model (right) compared to the synchronous web model (left)

AJAX explained with a visual example: https://medium.com/free-code-camp/ajax-basics-explained-by-working-at-a-fast-food-restaurant-88d95f5fcb7a

What is AJAX? A brief summary: https://www.ibm.com/docs/en/rational-soft-arch/9.6.1?topic=page-asynchronous-javascript-xml-ajax-overview

4.2 JSON

Before we can dive into the specifics of asynchronous client-server communication, we have to talk about JSON. JSON is a data format short for JavaScript Object Notation. It is a

lightweight data-interchange format that is easy for humans to read and write. As seen in Image 2, this format is often used in the asynchronous web model as a server response to pass data back to the web page.

JSON is built on two structures, which you will recognize from previous chapters:

- a collection of name/value pairs, held by curly brackets: objects
- an ordered list of values, held by square brackets: arrays

Example of a JSON file:

```
[
  {"name":"Witje","type":"Kat","age":3},
  {"name":"Zwartje","type":"Konijn","age":1}
]
```

Note that for name/value pairs in JSON, the name is written between double quotes as shown in the example above.

We can also write more complex nested combinations of objects and arrays in JSON, for example:

```
{
"employees":[
    {"firstName":"John", "lastName":"Doe"},
    {"firstName":"Anna", "lastName":"Smith"},
    {"firstName":"Peter", "lastName":"Jones"}
]
}
```

Read more about JSON: https://www.w3schools.com/whatis/whatis_json.asp

4.3 Fetch API

As previously discussed, the asynchronous web model allows us to asynchronously fetch data from a server without reloading the entire webpage. A group of technologies that originally

allowed us to implement this is AJAX (Asynchronous JavaScript And XML). AJAX uses a **XMLHttpRequest object** to communicate with servers. Data can be sent and retrieved between client and server in a variety of formats, such as JSON, XML and HTML. The server request typically has a **callback** function that specifies what needs to happens when the request is completed. However, a more modern way of implementing the asynchronous web model is by using the Fetch API, which we will discuss in the following sections.

4.3.1 Fetch API: promise

The Fetch API provides an interface for fetching resources across the network by using the *fetch()* function. The *fetch()* function starts the process of fetching a resource from the network and returns a **promise** which is fulfilled once the **response** is available.

A promise is a JavaScript object that links **producing code** and **consuming code**. Producing code is code that does something and can take some time. Consuming code is code that must wait for the result of the producing code. The promise links them together by making the promised result of the producing code available to the consuming code, once a response is ready.

Here's an example of how the Fetch API is implemented using promises to handle operations:

```
window.onload = () => {
    fetchAnimals()
}


function fetchAnimals () {
    fetch("http://localhost:8080")
        .then(response => response.json())
        .then(animals => renderAnimals(animals))
}
```

In this example the asynchronous function *fetchAnimals()* fetches data from the URL "http://localhost:8080" using the *fetch()* function from the Fetch API. This function returns a promise, which is fulfilled once the server sends back a response. We can then use the *.then()* method to specify what needs to happen with that response. In this example the response is converted to JSON format and this data is then rendered on the webpage.

4.3.2 Fetch API: async/await

Instead of using the traditional syntax of promises, we can also use async/await. Async/await makes promises easier to write and read, by making asynchronous code look more like synchronous code. The keyword *async* before a function makes the function return a promise. The keyword *await* before a function makes the function wait for a promise and can only be used inside an *async* function.

Let's rewrite the previous example using async/await:

```
window.onload = async () => {
     renderAnimals(await fetchAnimals())
}


const fetchAnimals = async () => {
    const response = await fetch("http://localhost:8080")
    const animals = await response.json()
    return animals
}
```

Compare this example with the previous one and note the differences in syntax.

In the above example *fetch()* is used to retrieve data from the server. We can also use the fetch API to **post** data to the server, for example when the user submits a form and the data needs to be saved on the server side. The *fetch()* function optionally accepts a second parameter, an object, that we can use to control different settings. When saving data to the server, we use this object to specify the method as POST and add the data that needs to be passed to the server in JSON format:

```
const postJSON = async (data) => {
    const response = await fetch("http://localhost:8080", {
      method: "POST", // or 'PUT'
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(data),
    });
```

```
    const result = await response.json();
    console.log("Success:", result);
}
```

More examples of how the fetch API can be used:
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Writing asynchronous code by using the Fetch API in combination with async/await is the most modern way of implementing the asynchronous web model. Table 1 shows a comparison of the different methods we have discussed in this chapter.

	Oldest way	Old way	Modern way
Communication with server through	XMLHttpRequest object	Fetch API	Fetch API
Data format of the server response	XML	JSON	JSON
Method used to handle asynchronous operations	Callback	Promise	Async/await

Table 1: different implementations of the asynchronous web model

Documentation of the Fetch API:
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

Further reading on AJAX versus Fetch API:
https://medium.com/@reemshakes/is-ajax-getting-replaced-by-fetch-api-55207234793f

Further reading on promises and async/await: https://javascript.info/async

5 Error handling and polling

5.1 Error handling

When programming dynamic webpages and handling data, there are many things that can potentially go wrong. Maybe we try to request data from the server when there is no data available, or the user tries to submit an empty form or a form with invalid data. To offer a smooth user experience to the end-user, it is important that we handle errors and inform users of what went wrong in an appropriate manner.

A simple way of handling specific errors is by adding a conditional statement. Let's take the example of a user entering invalid data in a form. Say we have a form that asks users to submit the month they were born in as a number between 1 and 12. We then use the Fetch API to save this data to the server, like so:

```
const submitForm = async (month) => {
     const response = await fetch("http://localhost:8080", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify(month),
    });

    const result = await response.json();
    console.log("Success:", result);
}
```

For the data to be valid, "month" needs to be a number between 1 and 12. However, what if the user enters an invalid number? In this case we don't want to save the data to the server, but instead display an error message on our web page to communicate the error to the user. We can do this by adding a conditional statement where we check the submitted data before saving it:

```
const submitForm = async (month) => {
     if (month < 1 || month > 12) {
        showErrorMessage(“Month needs to be a number between 1 and 12”);
     } else {
        const response = await fetch("http://localhost:8080", {
         method: "POST",
         headers: {
           "Content-Type": "application/json",
         },
         body: JSON.stringify(month),
       });

       const result = await response.json();
       console.log("Success:", result);
    }
}

const showErrorMessage = (message) => {
     document.getElementById(“status”).innerHTML +=
           “<p>” + message “</p>”;
}
```

In this updated example, the data from the submitted form is only saved to the server if “month” is a number between 1 and 12. If not, we call a new function *showErrorMessage()*, which manipulates an element in our HTML to display the error message.

We can use this same approach of adding a conditional statement for dealing with other types of errors, for example:

- When a user tries to submit a form without data.
- When we want to display data from the server but there is no data available.
- When the input of a field in a form can’t exceed a specific number of characters.
- When the data entered in a form needs to be of a specific type, for example a string.
- …

Of course, the way you need to handle errors will depend on the nature of the application you’re developing and what is and isn’t considered an invalid action.

5.2 Polling

Polling is a technique where we check for fresh data over a given interval by periodically making API requests to a server. This way we can make sure the data shown on our webpage stays up to date.

In JavaScript we can use the *setInterval* or *setTimeout* function to implement polling. Both functions have 2 parameters:

- the function that needs to be executed
- the number of milliseconds until the next poll needs to be done

Here's an example of how we can add polling to the code from the previous chapter, using *setInterval:*

```
window.onload = () => {
    setInterval( async () => {
        renderAnimals( await fetchAnimals())
    }, 10000)
}


const fetchAnimals = async () => {
    const response = await fetch("http://localhost:8080")
    const animals = await response.json()
    return animals
}
```

There is a downside to using *setInterval*: if our server is slow the response time could be longer than the interval between each poll, causing multiple requests to be pending at the same time. You can read more about the difference between *setInterval* and *setTimeout* here:
https://brianchildress.co/simple-polling-using-settimeout/