Advanced Web Programming

Data oriented programming

Array, Object, JSON

Grzegorz Rogus
rogus@agh.edu.pl

Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
AGH University of Science and Technology

Data in JS code

- After lab2 we knew about event programming and DOM manipulation.
- This is the foundation of reactivity in our web pages (web app).
- It is great and amazing

But …

In web applications, we mainly modify data

so today's lecture is dedicated to the data. Data in JS, where they are transformed and modified.

Agenda

- Array – methods
- Higher-order function – Functional programming in JS
- Object
- Manipulation of object
- Array of Object
- JSON
- FetchAPI
- Asynchronous programming in JS

Array and Object

The two most used data structures in JavaScript are:
Object and Array.

Objects allow us to create a single entity that stores data items by key.
Arrays allow us to gather data items into an ordered list.

Object

Objects are variables too.
But objects can contain many values.

```
const car = {type:"Fiat", model:"500", color:"white"};
```

The values are written as **name:value** pairs
(name and value separated by a colon).

Accessing JavaScript Properties:

```
objectName.property      // person.age

or

objectName["property"]   // person["age"]
```

Example

```
let fidgetSpinner = {
  owner: " GR",                        // string
  grade: "Junior",                          // string
  "zip-code": 90210,                          // number
  price: 3.95,                              // number
  colors: ["red", "green", "purple"],               // array
  getChoice: function() { return this.owner + " bought " + this.colors[1]; }
};
console.log(fidgetSpinner.price);         // 3.95
console.log(fidgetSpinner.colors[2]));    // purple
console.log(fidgetSpinner.getChoice());   // GRbought green
console.log(fidgetSpinner["zip-code"]);   // 90210
console.log(fidgetSpinner.zip-code]);     // error
```

An object can have methods (function properties) that refer to itself as this can refer to the fields with .fieldName or ["fieldName"] syntax

Displaying Object Properties

```
const person = {
  name: "John",
  age: 30,
  city: "New York"
};
document.getElementById("demo").innerHTML = person.name + "," + person.age + "," +
person.city;
```

Displaying the Object in a Loop

```
const person = {    name: "John",    age: 30,    city: "New York"  };

let txt = "";
for (let x in person) {
    txt += person[x] + " ";
};

document.getElementById("demo").innerHTML = txt;
```

Collection of data - Array (reminder)

- Array - structure to store ordered collections.
- How to create array in JS (3 solustions):
  - `let arr = new Array(ele0, ele1, ..., eleN)`
  - `let arr = Array(ele0, ele1, ..., eleN)`
  - `let arr = [ele0, ele1, ..., eleN]; // preferred`

  - `let fruits = ["Apple", "Orange", "Plum"];`

- How to get or set array elements

```
arr[0] = "Mango",
let x = arr[1];
```

Convert an Object to an Array in JavaScript

For plain objects, the following methods are available:

- Object.keys(obj) – returns an array of keys.
- Object.values(obj) – returns an array of values.
- Object.entries(obj) – returns an array of [key, value] pairs.

```
let user = {
  name: "John",
  age: 30
};

Object.keys(user) = ["name", "age"]
Object.values(user) = ["John", 30]
Object.entries(user) = [ ["name","John"], ["age",30] ]
```

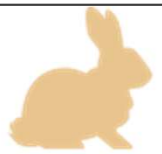
Array - List operations

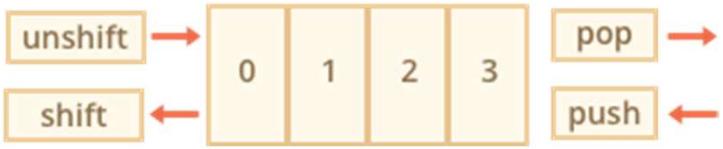
Method	Description
`list.push(`*element*`)`	Add ***element*** to back
`list.unshift(`*element*`)`	Add ***element*** to front

Method	Description
`list.pop()`	Remove from back
`list.shift()`	Remove from front

Method	Description
`list.indexOf(`*element*`)`	Returns numeric index for ***element*** or -1 if none found

slow method 

 Fast method



Array - List operations part 2

slice(start_index, upto_index) extracts a section of an array and returns a new array.

```
let myArray = ['a', 'b', 'c', 'd', ,e’];
myArray = myArray.slice(1, 4)  // starts at index 1 and extracts all elements
until index 3, returning [ "b", "c", "d"]
```

splice(index, count_to_remove, addElem1, addElem2, ...) removes elements from an array and (optionally) replaces them. It returns the items which were removed from the array.

```
let myArray = ['1', '2', '3', '4', ,5’];
myArray.splice(1, 3, 'a', 'b')
// myArray is now ["1", "a", "b", "5"]
// This code started at index one (or where the "2" was), removed 3 elements there,
and then inserted all consecutive elements in its place.
```

concat() joins two or more arrays and returns a new array.

```
let myArray = ['1', '2', ,3’];
myArray = myArray.concat('a', 'b', 'c')
// myArray is now ["1", "2", "3", "a", "b", "c"]
```

Loop through the array

Array iteration methods operate on every array item.

1. Old method

```
const tab = ["Marcin", "Ania", "Agnieszka"];
for (let i=0; i<tab.length; i++) {
      console.log(tab[i]);     //"Marcin", "Ania"...
}
```

2. Beter
for.each

```
const tab = ["Marcin", "Monika", "Magda"];

tab.forEach(el => {
   console.log(el.toUpperCase());    //"MARCIN", "MONIKA", ....
});
```

3. The Best
Loop for of

```
const tab = ["Marcin", "Ania", "Agnieszka"];

for (const el of tab) { //el  - variable name by us
   console.log(el);       //"Marcin", "Ania"...
}
```

Searching in array

Special method	result
• Array.includes	// true/false
• Array.find	// finding element or null
• Array.indexOf	// index number
• Array.findIndex	// index number
• Array.lastIndexOf	// index number
• Array.filter	// new array
• Array.Map()	// new array
• Array.some()	// true/false

Problem 1 – how to find elements that meet a certain condition

```
const ages = [11, 34, 8, 9, 23, 51, 17, 40, 14];
let tab = [];
 for (let i = 0; i < ages.length; i++) {
    if(ages[i] > 18){
        tab.push(ages[i])
} }
console.log(tab);   // [34, 23, 51, 40]
```

Old fashion –

don't do that

```
const ages = [11, 34, 8, 9, 23, 51, 17, 40, 14];

let tab = ages.filter((age) => age > 18);

console.log(tab); //[34, 23, 51, 40]
```

New approach -

do like that

The filter() method takes each element in an array and it applies a conditional statement against it.
If this conditional returns true, the element gets pushed to the output array.
If the condition returns false, the element does not get pushed to the output array.

Filter function

A higher order function is **a function that either takes a function as an argument or returns a function**

```
let results = arr.filter(function(item, index, array)
{   // if true item is pushed to results and the iteration continues
    // returns empty array if nothing found
});
```

```
let tab = ages.filter((age) => age > 18);
```

Arrow function
Thanks ES2015!!!

function fufu(arg1) { expresion} == (arg1, arg2, ...argN) => expression

```
const students = [
  { name: 'Quincy', grade: 96 },    { name: 'Jason', grade: 84 },   { name: 'Alexis', grade: 100 },
  { name: 'Sam', grade: 65 },   { name: 'Katie', grade: 90 }
];

const studentGrades = students.filter(student => student.grade >= 90);
return studentGrades;
// [ { name: 'Quincy', grade: 96 }, { name: 'Alexis', grade: 100 }, { name: 'Katie', grade: 90 } ]
```

Problem 2 – how to modify all elements in array

The map() method is used for creating a new array from an existing one, applying a function to each one of the elements of the first array.

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(item => item * 2);
console.log(doubled);       // [2, 4, 6, 8]
```

```
const tab = ["Marcin", "Monika", "Magda"];

const tab2 = tab.map(el => el.toUpperCase());

console.log(tab); //[Marcin, Ania, Agnieszka]
console.log(tab2); //[MARCIN, ANIA, AGNIESZKA]
```

```
function multiple3(number) {
   return number * 3;
}
var ourTable = [1, 2, 3];
console.log(ourTable.map(multiple3)); //[3, 6, 9]
```

Problem 3 – how to calculate value from all items

- The reduce() method runs a function on each array element to produce (reduce it to) a single value.

- The reduce() method works from left-to-right in the array.
- ReduceRight() work in revers direction.

- The reduce() method does not reduce the original array.

```
const tab = [3, 2, 4, 2];

const result = tab.reduce((a, b) => a * b);

//1 iteration => prev = 3,  curr = 2
//2 iteration => prev = 6,  curr = 4
//3 iteration => prev = 24, curr = 2
//resullt = 48
```

Problem 4 – how to check if an elem exsist

The indexOf(), lastInexOf, include methods searches an array for an element value and returns its position.

- arr.indexOf(item, from) – looks for item starting from index from, and returns the index where it was found, otherwise -1.
- arr.lastIndexOf(item, from) – same, but looks for from right to left.
- arr.includes(item, from) – looks for item starting from index from, returns true if found.

```
const fruits = ["Apple", "Orange", "Apple", "Mango"];


let position = fruits.indexOf("Apple") + 1;        // 1

let position = fruits.lastIndexOf("Apple") + 1;    // 3

fruits.includes("Mango");     // is true
```

Problem 4 – how to check if an elem exsist

Check If Every Element Passes a Test: every()
Check If At Least One Element Passes a Test: some()

The every() method check if all array values pass a test.
The some() method check if some array values pass a test.

Both methods return true or false as a result.
Every () will return true when the function passed in the parameter returns true for every item in the array.

```
const numbers = [45, 4, 9, 16, 25];

let someOver18 = numbers.some(el => el >= 18 );

let allOver18 = numbers.every(el => el >= 18 );
```

This example check if some array values are larger than 18: // true

This example check if all array values are larger than 18: // false

Problem 5 – how to find element

The find() method returns the value of the first array element that passes a test function

```
const tab = [12, 5, 8, 130, 44];

const bigNr = tab.find(el => el >= 15);
console.log(bigNr);      //130
```

```
const tab = [
   { name : "Karol", age: 10, gender: "m" },
   { name : "Beata", age: 13, gender: „f" },
   { name : "Marcin", age: 18, gender: "m" },
   { name : "Ania", age: 20,gender: „f" }
];

const firstWoman = tab.find(el => el.gender === „f");
console.log(firstWoman); //{ name : "Beata", age: 13, gender: „f" }
```

Method chaining

Method chaining is a way of launching subsequent methods (functions) that we write after the dot.
Running the function returns some value.
If such a returned value matches for the next method, we can run this method immediately after the dot:

```
const tab = ["Marcin", "Monika", "Magda"];

const newTab = tab
    .map(el => el.toLowerCase())       //return new array...
    .filter(el => el.endsWith("a"))     //...so I can filter
    .map(el => el + "!")                 //...filter return array which I use with map
    .forEach(el => console.log(el))    //...map return array so forEach match to do it

console.log(newTab)
```

Array of objects Method chaining

```
const students = [
  { name: "Nick", grade: 10 },
  { name: "John", grade: 15 },
  { name: "Julia", grade: 19 },
  { name: "Nathalie", grade: 9 },
];

const aboveTenSum = students
  .map(student => student.grade) // we map the students array to an array of their grades
  .filter(grade => grade >= 10) // we filter the grades array to keep those 10 or above
  .reduce((prev, next) => prev + next, 0); // we sum all the grades 10 or above one by one

console.log(aboveTenSum) // 44 -- 10 (Nick) + 15 (John) + 19 (Julia), Nathalie below 10 is ignored
```

Destructuring assignment

- Destructuring assignment is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables, as sometimes that’s more convenient.
- Destructuring also works great with complex functions that have a lot of parameters, default values, and so on.

```
const person = {
  firstName: "Nick",
  lastName:
"Anderson",
  age: 35,
  sex: "M"
}
```

Without destructuring

```
const first = person.firstName;
const age = person.age;
const city = person.city || "Paris";
```

With destructuring, all in one line:

```
const { firstName: first, age, city = "Paris" } = person; // That's it !
```

Summary

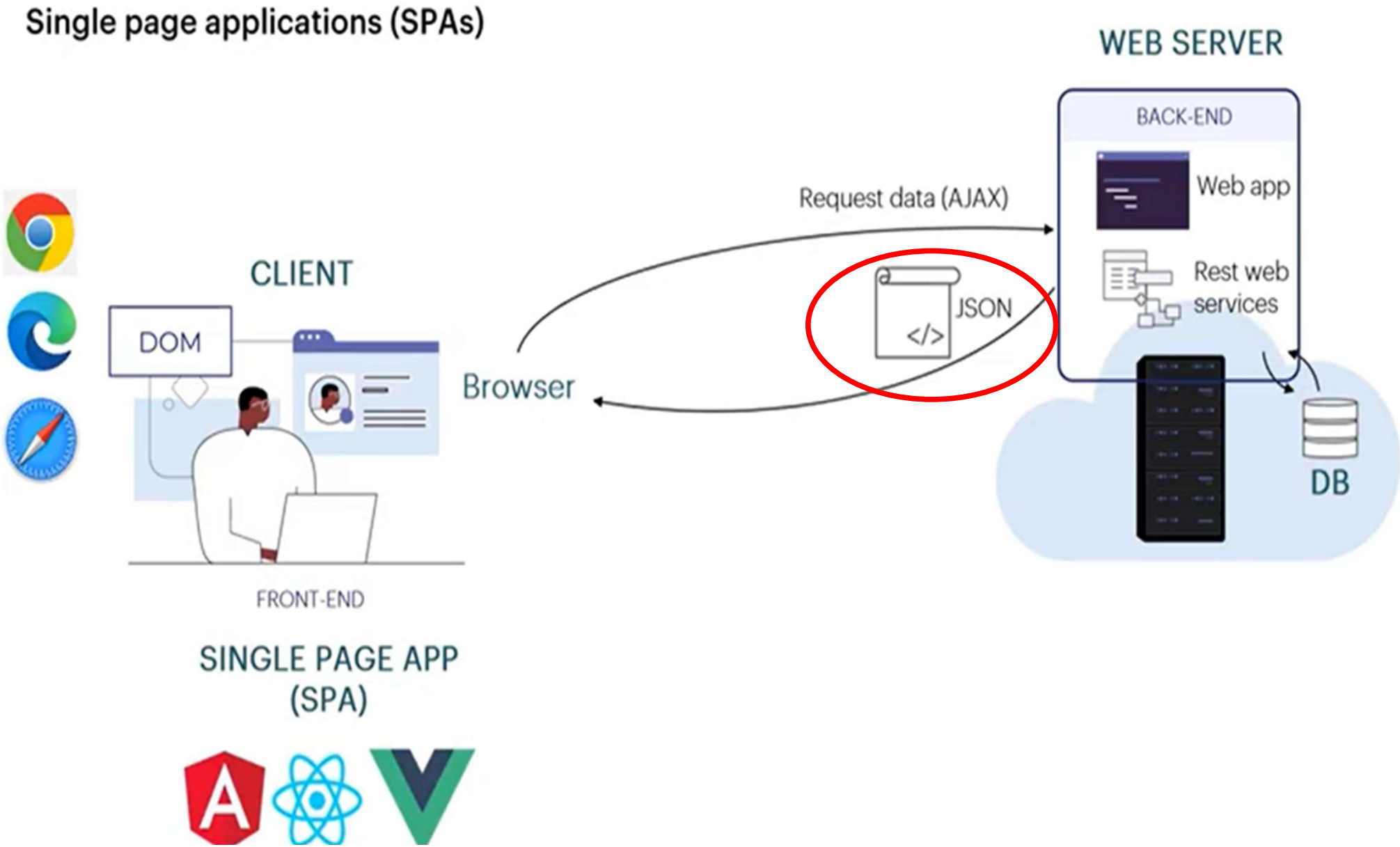
- When we need to iterate over an array – we can use forEach, for or for..of.
- When we need to iterate and return the data for each element – we can use map
- When we need check if items exsist we can use: indexOf, include, every, some
- When we need get some items which meet a certain conditio we use Filter.

problem

the data is still embedded in our code.
We would like to get data from external enviroment.

How to Loading data from files

Do you remember?
Modern web application load json files.



JSON

JavaScript Object Notation

JSON is a text format for storing and transporting data
JSON is "self-describing" and easy to understand

A data format that represents data as a set of JavaScript objects natively supported by all modern browsers (and libraries to support it in old ones)

JSON Rules!

JSON has a few rules that differ from regular JS:

Strings must be quoted
All property/field names must be quoted
Values can be:
- Number (23)
- String ("string has to be quoted, like this")
- Boolean (true, false)
- Array (["value1", 24, true, "hallo"])
- Object ({ "nested" : "object", "which" : { "can" : "nested objects" } })
- null

Numerous validators/formatters available, eg JSONLint

JSON - example

The JSON format is very similar to classic JavaScript objects:

Valid JSON

```
{                       // no variable assignment
   "first_name": "Bart",    // strings and properties must be quoted
  "last_name": "Simpson",  // with double quotes
  "age" : 13,                    // numbers can be here without quotes
  "cowabunga": true       // booleans can be here without quotes
}                       // no semicolon at the end
```

Object array in JSON

Example Objects array

```
{ "samochod": [
    {
        "Marka": "VW",
        "Model": "Golf",
        "Rocznik": 1999
    },
    {
        "Marka": "BMW",
        "Model": "S6",
        "Rocznik": 2007
    },
    {
        "Marka": "Audi",
        "Model": "A4",
        "Rocznik": 2009
    }
]}
```

The JSON format is very similar to classic JavaScript objects

JavaScript Objects and JSON

JSON is a way of saving or storing JavaScript objects.

Browser JSON methods:

- JSON.parse(/* JSON string */) -- converts JSON string into Javascript object

- JSON.stringify(/* Javascript Object */) -- converts a Javascript object into JSON text

JSON stringify/parse example

```
const ob = { name : "Grzegorz", surname : "Rogus" }


const obStr = JSON.stringify(ob);
 console.log(obStr); //"{"name":"Grzegorz","subname":"Rogus"}"

console.log( JSON.parse(obStr) );  //our previously object

                                   //{name:"Grzegorz",subname:"Rogus"}
```

JSON stringify/parse example

```
> let point = {x: 1, y: 2, z: 3}
> point
> {x: 1, y: 2, z: 3}

> let s = JSON.stringify(point);
> s
> "{"x":1,"y":2,"z":3}"

> s = s.replace("1", "4");
> "{"x":4,"y":2,"z":3}"

> let point2 = JSON.parse(s);
> point2
> {x: 4, y: 2, z: 3}
```

What is JSON used for?

JSON data comes from many sources on the web:

- web services use JSON to communicate
- web servers store data as JSON files
- databases sometimes use JSON to store, query, and return data

JSON is the de facto universal format for exchange of data

AJAX

Asynchronous JavaScript and XML

Modern Web Applications

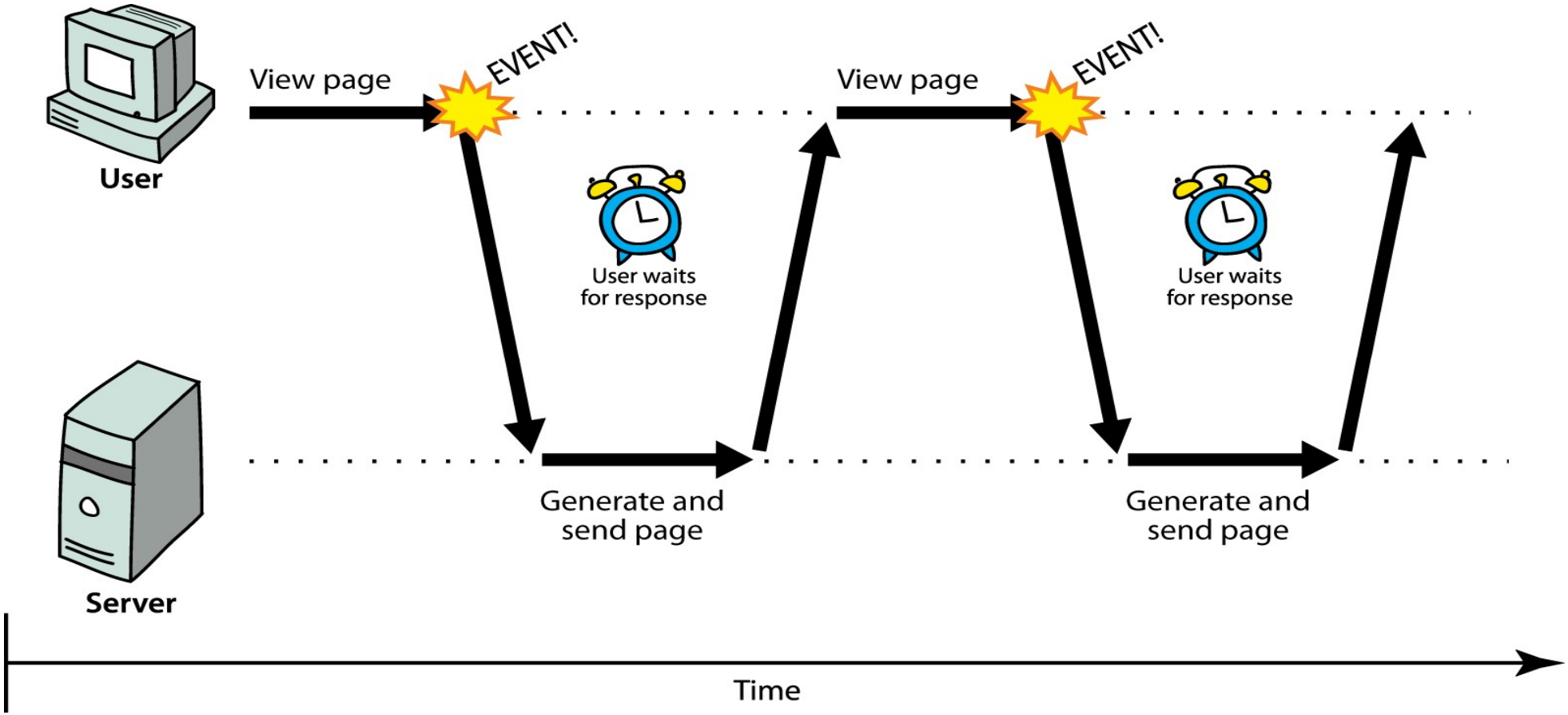
All of the pages that we've made up until now have content, style, and behavior.

Web applications are webpages that pull in additional data and information as the user progresses through them, making it feel similar to a desktop application.

Some motivations for making web pages into web applications:

- Better user experience
- Less data sent across the wire
- Leads to good software architecture:
- Client handles display, server serves up data

Synchronous requests

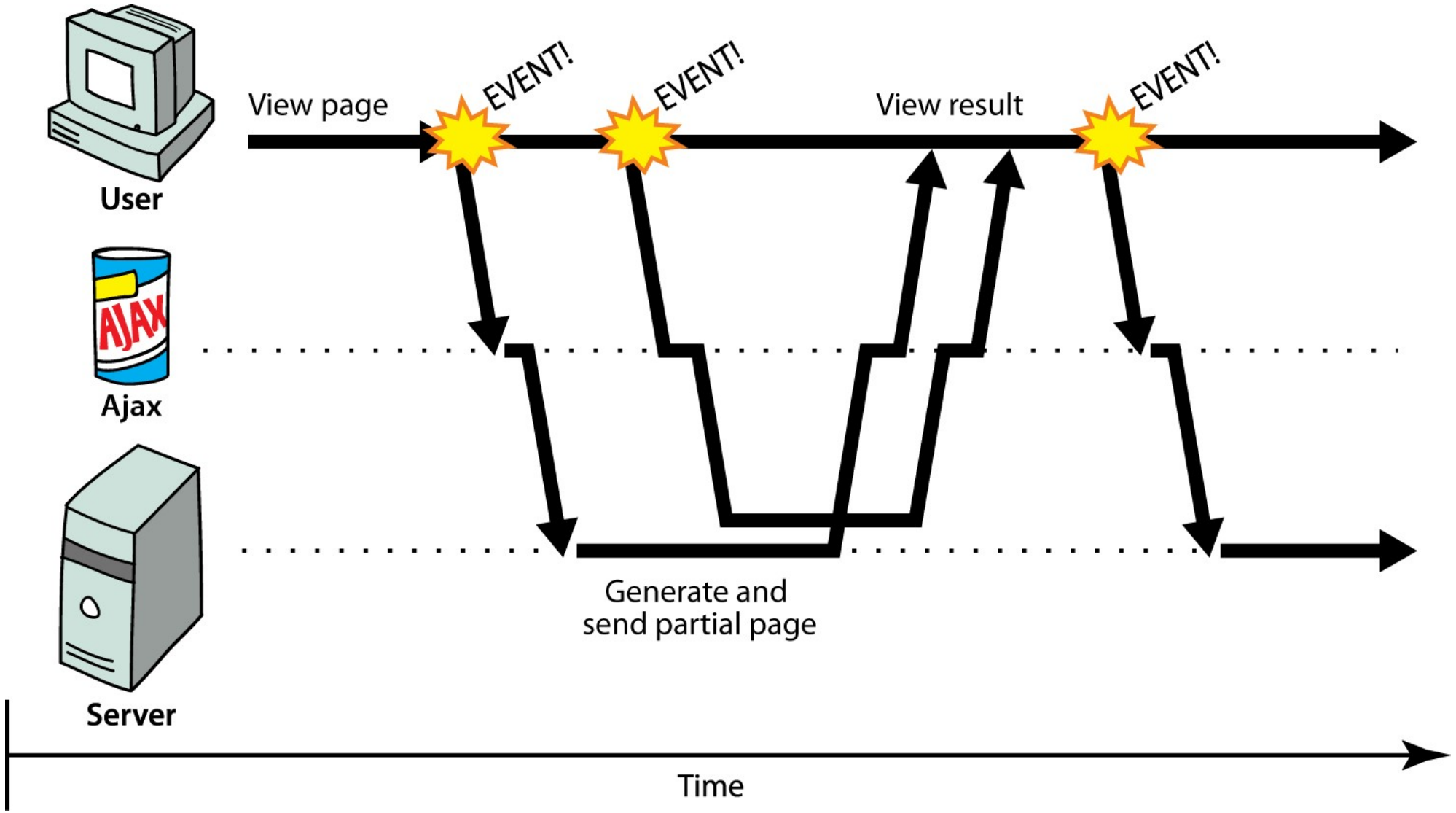


Why are synchronized requests are problematic?

Your code waits for the request to completely finish before proceeding.

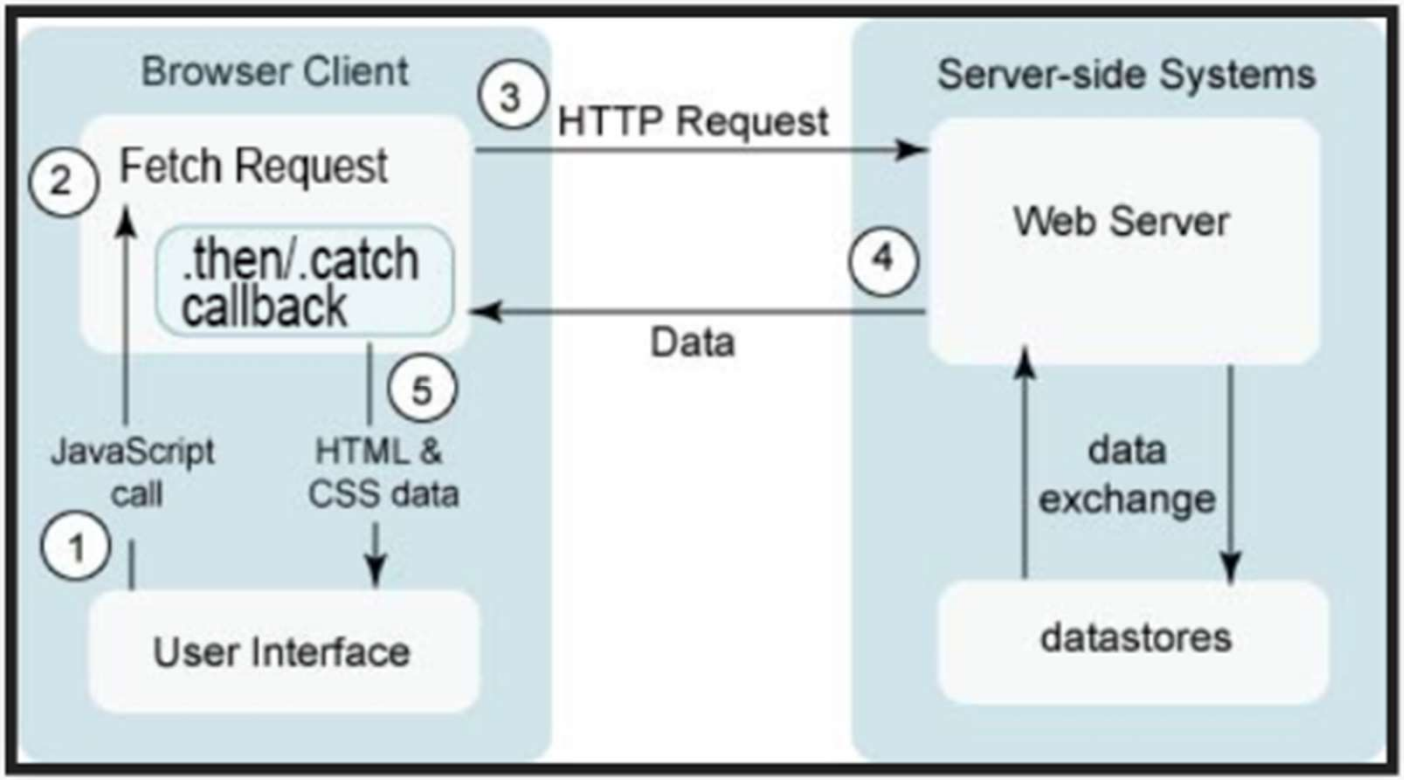
It is easier to program for synchronous behavior, but the user's entire browser LOCKS UP until the download is completed, which is a terrible user experience (especially if the page is very large or slow to transfer)

Asynchronous requests



Why use AJAX?

- You can use AJAX to download information from a server in the background
- It allows dynamically updates to a page without making the user wait
- It avoids the "click-wait....refresh" which would frustrate users



Fetch API

The Fetch API is the API to use to load external resources (text, JSON, etc) in the browser.

The Fetch API is made up of one function, and its syntax is is concise and easy to use:

```
fetch('images.txt');
```

- The `fetch()` method takes the string path to the resource you want to fetch as a parameter
- It returns a `Promise`

Fetch API

```
fetch(url, [options]);
```

```
fetch("https://test.com/zasob")
   .then(response => {
       console.log(response);
    }
)
```

Object Response

ok
status (200, 404, 301 itp.)
statusText status connection in text format (for example Not found)
type connection type
url
body

response.text() Format response

response.json()

response.formData()

response.blob()

response.arrayBuffer()

Processing the returned data

Now that we've done a fetch, we need to do something with the data that comes back from the server.

But we don't know how long that will take or if it even will come back correctly!

The fetch call returns a Promise object which will help us with this uncertainty.

Real world promises

Promises have three states:

Pending
Fulfilled
Rejected

Example: “I promise to post HomeWork 1”

Pending: Not yet posted

Fulfilled: HW 1 posted

Rejected: Wrong homework posted, or not posted in time

JS promises

promise
A JS object that executes some code that has an uncertain outcome

Promises have three states:

Pending
Fulfilled
Rejected

```
...
  let promise = new Promise(action);
...
function action(resolve, reject) {
  // do pending uncertain action
  //  (like make an AJAX call)

  if (success) {
    resolve();      // Fulfilled
  } else {
    reject();       // Rejected
  }
}
```

Chaining Promises

We want the following asynchronous actions to be completed in this order:

1. When the `fetch` completes, run `onResponse`
2. When `response.text()` completes, run `onStreamProcessed`

```
function onStreamProcessed(text) { … }
function onResponse(response) {
  response.text().then(onStreamProcessed);
}
fetch('images.txt').then(onResponse, onError);
```

Rewrite Code

```
function onStreamProcessed(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt')
    .then(onResponse, onError)
    .then(onStreamProcessed);
```

Finish code

```
function onStreamProcessed(text) {
  const urls = text.split('\n');
  for (const url of urls) {
    const image = document.createElement('img');
    image.src = url;
    document.body.append(image);
  }
}
function onSuccess(response) {
  response.text().then(onStreamProcessed)
}
function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt').then(onSuccess, onError);
```

AJAX fetch Code Skeleton (text response)

A Promise is returned from a fetch call

We will be using promises when we fetch information from a server, which is an uncertain task

I give you „template" starting code because you will use this frequently

```
const BASE_URL = /* put base url string here */;
...
function callAjax() {
  let url = BASE_URL /* + any query parameters */;
  fetch(url)
  .then(checkStatus)
  .then(handeResponse)
  .catch(handleErrror);
}
function handleResponse(responseText) {
  //success: do something with the responseText
}

function handleError(error) {
  //error: do something with error
}

function checkStatus(response) {    // boiler plate code given out
  ...
}
```

AJAX fetch Code Skeleton (json response)

```
const BASE_URL = /* put base url string here */;
...
function callAjax() {
  let url = BASE_URL /* + any query parameters */;
  fetch(url)
  .then(checkStatus)
  .then(JSON.parse)      // parse the response string into a JSON object
  .then(handeResponse)
  .catch(handleErrror);
}
function handleResponse(responseJSON) {
  // now handle this response as a JSON object.
}

function handleError(error) {
  // error handling doesn't change
}
```

How to get Data from Fetch API

```
fetch("https://restcountries.eu/rest/v2/name/Poland")
    .then(res => res.json())
    .then(res => {
        console.log(res);   // display array of object in console
 })
```

Mechanics of Fatch API

We initiate a fetch of a URL

A fetch call returns a Promise object

- The .then method on a Promise object returns a Promise object
- Our first .then (checkStatus) checks the status of the response to makes sure the server responded with an OK. The result of that first .then is another Promise object with the response (text, JSON, ...) as the value of the Promise.
- We may .then(JSON.parse) which also returns a Promise object with a JSON object as the value
- We .then(handleResponse) which will do something with the response from the server.
- If at any time there is an error, the execution falls down to the .catch method on the Promise chain

Summary Fatch API

We initiate a fetch of a URL

A fetch returns a Promise

Promises are good because...

- They help deal with code that has an uncertain outcome
- They separate the completion of the fetch request from the page logic
- We can reuse the same logic and handle completion in different ways (e.g. refactor the AJAX logic or the function to handle the response)

Local json Example

Now let's assume we have "pizza.json" that contains a menu of pizzas, and other data, in the following format:

```
{
  "pizzaria": "Cottage Inn",
  "location": "Ann Arbor, MI",
  "pizzas": [
   {
     "name": "Roma Pesto",
     "description": "Sun dried ..."
   },
   {
     "name": "Motor City Meatball",
     "description": "Meatballs, pepperoni, ...
   },...
```

You want to fetch this information and load it on the page with the function loadJSONMenu and you want to handle errors with

Answer...

```
function loadJSONMenu() {
   const URL = "pizza.json";
   fetch(URL)
     .then(checkStatus)
     .then(JSON.parse)
     .then(loadJSONMenu)
     .catch(handleRequestError);
}
```

There's no longer any need to parse/split the result in loadJSONMenu - why?

Because the data is well structured

Local files

When we load a web page in the browser that is saved on our computer, it is served via `file://` protocol:

We are **not allowed** to load files in JavaScript from the `file://` protocol, which is why we got the error.

Serve over HTTP

When we load a web page in the browser that is saved on our computer, it is served via `file://` protocol:

We are **not allowed** to load files in JavaScript from the `file://` protocol, which is why we got the error.

Serve over HTTP

- Local server

```
npm i http-server -g
//lub
npm i live-server -g
```

- json-server

```
npm install json-server -g
```

```
json-server --watch nazwa-pliku.json
```

- Fake serwer in the cloude np. https://jsonplaceholder.typicode.com/posts