JavaScript pt. 2: DOM, Elements, & Fetch API

Kianoosh Abbasi

CSC309 Fall 2022

Some content is from Dr. Sadia Sharmin's slides of CSC309 Winter 2021: www.rainsharmin.com

So far

- Front-end:
  HTML: tags and forms
  CSS: styles, selectors, layout

- JavaScript Intro:
  Objects, functions
  Scopes, closures, arrow functions

This week

- DOM
  Getting and manipulating elements
- jQuery
- Asynchronous requests: Ajax
- Event loop
- Fetch API and Promises

Manipulating the web page

```
alert("Are you REALLY sure you want to leave??")
```

Where to put JS

- JS code should be placed inside the <script> tag

- Inline JS:

```
<script>
   console.log(1 + 2 + 3)
</script>
```

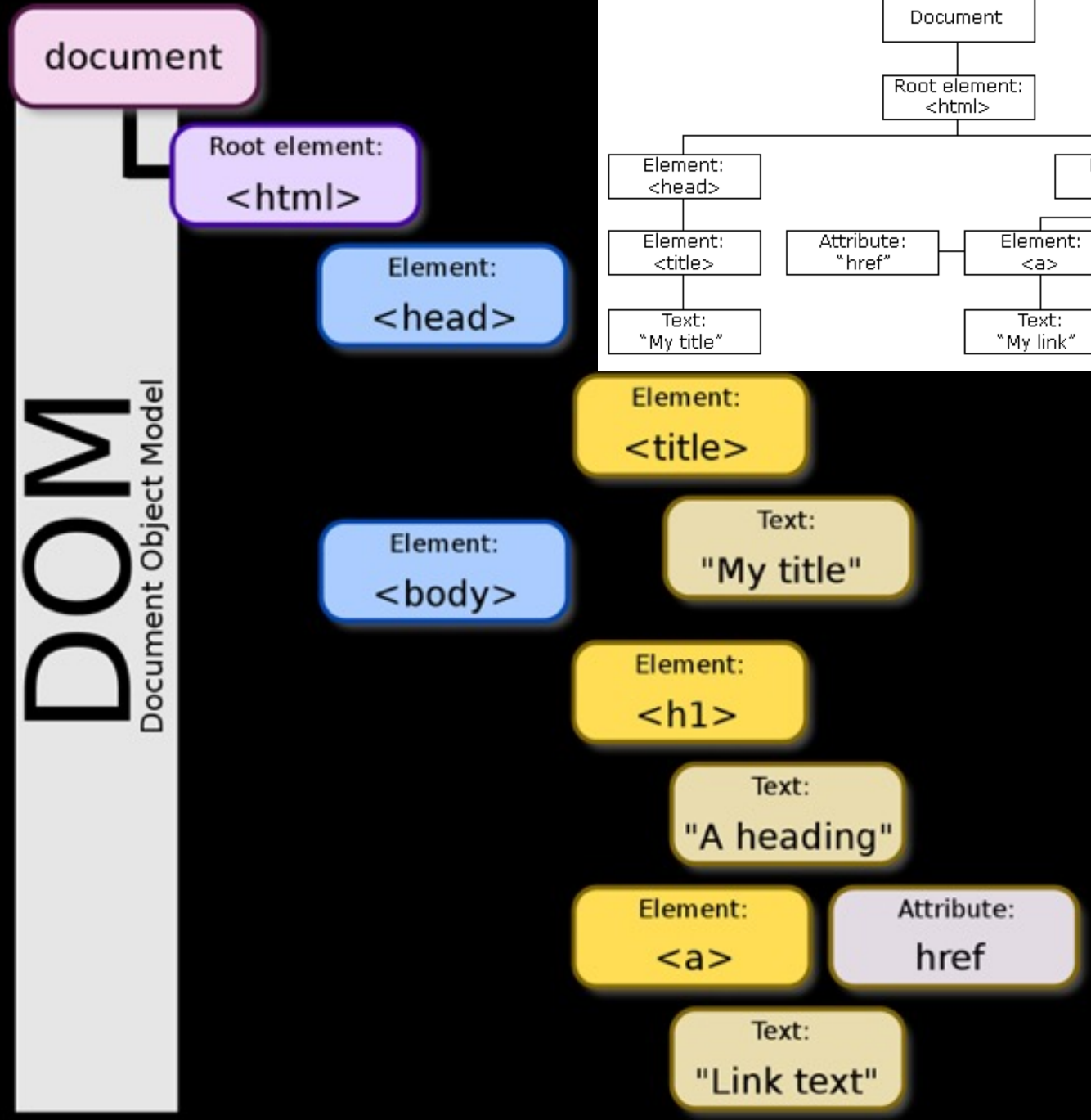
- JS file

```
<script src="city_region_dropdown.js"></script>
```

Document object model

- Browser creates the DOM tree of the page
- Each element is a node
- Child elements are children of the parent node
- Scripts access DOM through the document variable

```
<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <h1>A heading</h1>
    <a href="...">Link
    text</a>
  </body>
</html>
```



Getting elements

- Various ways to access an element

```
document.getElementById("st-2")
document.getElementsByClassName("ne-share-buttons")
document.getElementsByTagName("ul")

document.querySelector("#submit-btn")
document.querySelectorAll(".col-md-12")
```

- Good exercise at:

  https://javascript.info/task/find-elements/table.html

Navigating through DOM

- Relevant nodes can be accessed through properties

  `parentNode, firstChild, lastChild, childNodes, nextSibling`

- Example

```
let img = document.querySelector(
                "body section:first-child > img")
let par = img.parentNode

console.log(par.childNodes.length);
```

Manipulating elements

- Element properties
  innerHTML, style, getAttribute()

- Example

```
let body = document.body
body.innerHTML = "<h3>hello!</h3>"

h3 = document.getElementsByTagName("h3")
h3.style.color = "green"
h3.setAttribute("class", "title")
console.log(h3.getAttribute("style"))
```

Events

Visit https://www.w3schools.com/tags/ref_eventattributes.asp

- Various events are monitored by the browser

- document events

  onload, onkeydown, onkey

- Element events

  onclick, onmouseover, ondrag, oncopy, onfocus, onselect, onsubmit

Events

- You can define a function

```
h3.onclick = function() {
    this.innerHTML = "you just clicked on me!"
}
```

- Alternative:

```
<script>
    function h3click(h3){
        h3.style.color = "blue"
    }
</script>

<h3 onclick="h3click(this)" onmouseover="console.log(new
Date())"></h3>
```

Exercise: A form with client-side validation

- Examples:

  Checks if a security question is answered correctly

  Checks if the email input is valid

  Checks password and repeat password are the same

- Errors should appear dynamically and disappear if user has fixed the issue

jQuery

- Pure JS codes can be so verbose
- jQuery is a library that provides a lot of shortcuts to do the same things
- Add it to the project
  - Download source from https://jquery.com/download/
  - Import via `<script src="jquery-3.6.1.min.js"></script>`

Syntax

- Everything is done through the $ function (also called jQuery)

- Based on query selectors

- Examples:

```
$("p").hide()
$("#colorbox").removeClass("row")
$(".form").attr("method", "POST").submit()
```

Actions

- Run your scripts after the webpage has finished loading

```
$(document).ready(function(){
  // jQuery methods go here...
});
```

- A lot of shortcuts for events

```
$("p").click(function(){
  // action goes here
});
```

Note

- jQuery is effectively just a wrapper around plain JS
- But jQuery objects have different methods/properties than JS
- Example

```
document.querySelector("#title").innerHTML = "<h1>Hello</h1>"
$("#title").html("<h1>Hello</h1>")
```

Asynchronous requests

Requests

- Currently, one main request is made to the server (upon entering the URL or submitting a form)
- Response is rendered and additional requests made by browser to fetch static data (js, css, images, fonts, etc.)
- This way entails a full reload for just one request!!

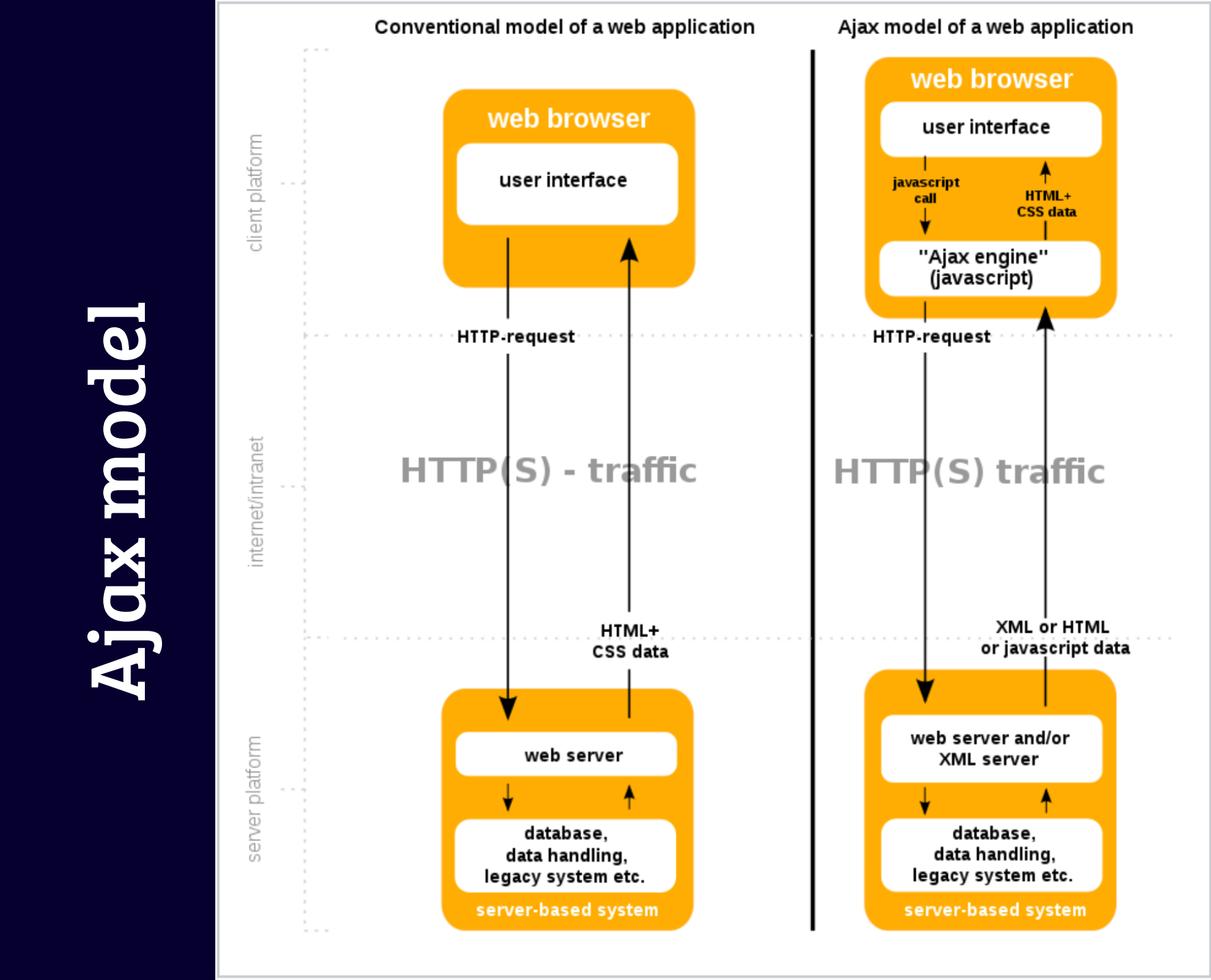
Solution

- Asynchronous JavaScript and XML (Ajax)

- Browser sends the request in background
  Does not block the main thread

- Response is handled by a series of events and callbacks
  Further changes are made to the document

Ajax model



Source: https://en.wikipedia.org/wiki/Ajax_(programming)

Why is it important

- Offers more control over the web page
  - You lose everything once the browser exits the current page!
- Most modern websites do not use the submit feature
  - Instead, they send an Ajax request and handle response
  - Client-side JS code redirects if necessary
- Basis for single-page frameworks like React

Ajax with jQuery

- Pure JS can send Ajax request (too verbose)
- jQuery's shortcut for Ajax is one of the bests!
- Specify URL, method, data, etc.
  All are optional
- JSON results already parsed at success
  Can be accessed through the data argument

```
$.ajax( options: {
    url: url,
    method: 'PATCH',
    data: {
        username: $('#username-input').val()
    },
    headers: {
        'X-CSRFToken': $('input[name=csrfmiddlewaretoken]').val()
    },
    success: function () {
        $('.show-modal').hide();
    },
    error: function (xhr) {
        if(xhr.status === 400){
            var response = xhr.responseJSON;
            if (response['username']){
                var message = response['username'][0];
                $error_div.html(message).show();
            }
        }
    }
})
```

Local storage

- A key-value storage shared between all pages of the same domain
- Can be used to storage cookies, session data, etc.
- Persistent, even after closing the tab/browser
  Unless you clear history
- Example

```
localStorage.setItem('access_token', access_token);
localStorage.getItem('access_token');
```

Event loop & Promises

Event loop

Visit https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

- JS is event-driven

- All your scripts is executed at load and the rest are events

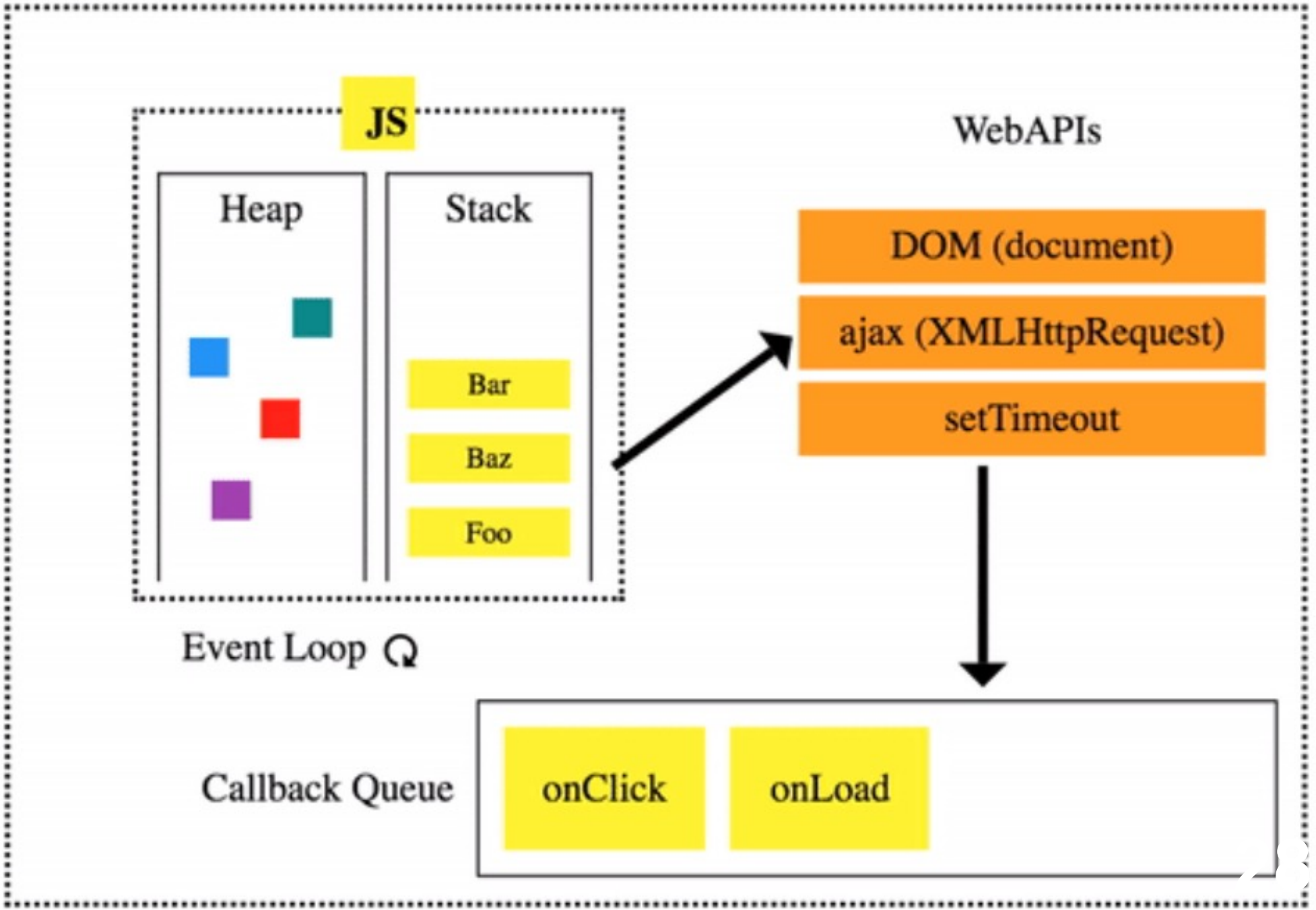
```
$(document).ready(…)
element.addEventListener(…)
$("p > button").click(function(){…})
```

Event loop

Visit https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

- JS is single-threaded
- Event loop provides the illusion of multiple threads
- Events get pushed to the event queue
  Examples: ready, click, ajax, setTimeout
- Event loop constantly checks for a new event and executes its callback
  It's synchronous!

Event Loop



Source: https://medium.com/@Rahulx1/understanding-event-loop-call-stack-event-job-queue-in-javascript-63dcd2c71ecd

Callback hell!

Visit http://callbackhell.com

```
fs.readdir(source, function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

Promises

- An alternative to massive nested callbacks
- Callbacks can make code hard to understand
- Example: jQuery Ajax has at least two callbacks: success and error

Fetch API

- Fetch API returns a promise

- Example:

```
let request = fetch('/account/login/', {
    method: 'POST',
    data: {username: 'Kia', password: '123'}
})

request.then(response => response.text())
       .then(text => console.log(response.json()));
```

Fetch API

- Seems like a mere replacement
- But avoids nested tabs and callbacks
- Promise: a piece of code that can lead to two states: resolved and rejected

Promises

- Promise has two functions: resolve and reject
- The initial state is pending
- Invoke resolve to change the state to resolved
- Invoke reject to changes it to rejected
- Transition is only possible from the pending state

Create a promise

- Example: A trivial promise

```
let test = new Promise(function(resolve, reject) {
    resolve("resolved hahahaha")
})
```

- The code inside of promise gets executed right away
- However, resolve and reject push events to event queue
- Can be handled by appropriate handlers: then and catch

Handling the result

- To handle the result:

```
test.then(message => console.log(message))
```

- Prints out "resolved hahahaha”

- Same situation with reject/catch

What's nice

- then/error will get called even if the promise is already settled

- Chaining promises: Multiple callbacks can be added by calling then several times

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => {
  console.log(`Got the final result: ${finalResult}`);
})
.catch(failureCallback);
```

Example

- What is the output?

```
const add = (num1, num2) => new Promise((resolve) => resolve(num1 + num2))


add(2, 4)
  .then((result) => {
    console.log(result)
    return result + 10
  })
  .then((result) => {
    console.log(result)
    return result
  })
  .then((result) => {
    console.log(result)
  })
```

When it makes sense?

- If your code is synchronous/deterministic (like previous examples), it does not make much sense to use promises

- But if it depends on an external event (i.e., request sent successfully or not), it does make sense
  - No longer need to define multiple callbacks
  - Just one catch and several then callbacks

- That's why FetchAPI returns a promise

Promises vs Callbacks

visit https://dev.to/neisha1618/callbacks-vs-promises-4mi1

```
const makePB&J = () => {
makeBread(function() {
 putPeanutButter(function(...args) {
   spreadJelly(function(...args) {
     sandwichThem(bread,peanutButter,jelly){
    // welcome to hell
     });
   });
  });
});
};
```

```
const makePb&J = () => {
 return makeBread()
   .then(peanut => putPeanutButter(peanut))
   .then(jelly => spreadJelly(jelly))
   .then(sandwich => sandwichThem(sandwich));
 catch((ewww crunchyPeanutButter));
};
```

```
function getFrogsWithVitalSigns(params, callback) {
  let frogIds, frogsListWithVitalSignsData

  api.fetchFrogs(params, (frogs, error) => {
    if (error) {
      console.error(error)
      return
    } else {
      frogIds = frogs.map(({ id }) => id)
      // The list of frogs did not include their health information, so lets fetch that
      api.fetchFrogsVitalSigns(
        frogIds,
        (frogsListWithEncryptedVitalSigns, err) => {
          if (err) {
            // do something with error logic
          } else {
            // The list of frogs health info is encrypted. Our friend texted us the sec
            api.decryptFrogsListVitalSigns(
              frogsListWithEncryptedVitalSigns,
              'pepsi',
              (data, errorr) => {
                if (errorrr) {
                  throw new Error('An error occurred in the final api call')
                } else {
                  if (Array.isArray(data)) {
                    frogsListWithVitalSignsData = data
                  } else {
                    frogsListWithVitalSignsData = data.map(
                      ({ vital_signs }) => vital_signs,
                    )
                    console.log(frogsListWithVitalSignsData)
                  }
                }
              },
            )
          }
        },
```

```
function getFrogsWithVitalSigns(params, callback) {
  let frogIds, frogsListWithVitalSignsData

  api
    .fetchFrogs(params)
    .then((frogs) => {
      frogIds = frogs.map(({ id }) => id)
      // The list of frogs did not include their health information, so lets fetch that
      return api.fetchFrogsVitalSigns(frogIds)
    })
    .then((frogsListWithEncryptedVitalSigns) => {
      // The list of frogs health info is encrypted. Our friend texted us the secret key
      return api.decryptFrogsListVitalSigns(
        frogsListWithEncryptedVitalSigns,
        'pepsi',
      )
    })
    .then((data) => {
      if (Array.isArray(data)) {
        frogsListWithVitalSignsData = data
      } else {
        frogsListWithVitalSignsData = data.map(
          ({ vital_signs }) => vital_signs,
        )
        console.log(frogsListWithVitalSignsData)
      }
    })
    .catch((error) => {
      console.error(error)
    })
  })
}

const frogsWithVitalSigns = getFrogsWithVitalSigns({
  offset: 50,
})
  .then((result) => {
    console.log(result)
```

Source: https://betterprogramming.pub/callbacks-vs-promises-in-javascript-1f074e93a3b5

This week

- DOM
  Getting and manipulating elements
- jQuery
- Asynchronous requests: Ajax
- Event loop
- Fetch API and Promises

Next week

- Back-end development & frameworks
- Django
  Setup, simple views, forms, templates
- MVC Design patterns