JavaScript pt. 2: Ajax and Advanced JS

Kianoosh Abbasi

CSC309 Winter 2023

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

- Back-end: Django

- JavaScript Intro:
  Basic JS, DOM, elements, jQuery

This session

- Asynchronous requests: Ajax

- Advanced JS (React prep)
  - Arrow functions
  - Promises
  - Fetch API

Asynchronous requests

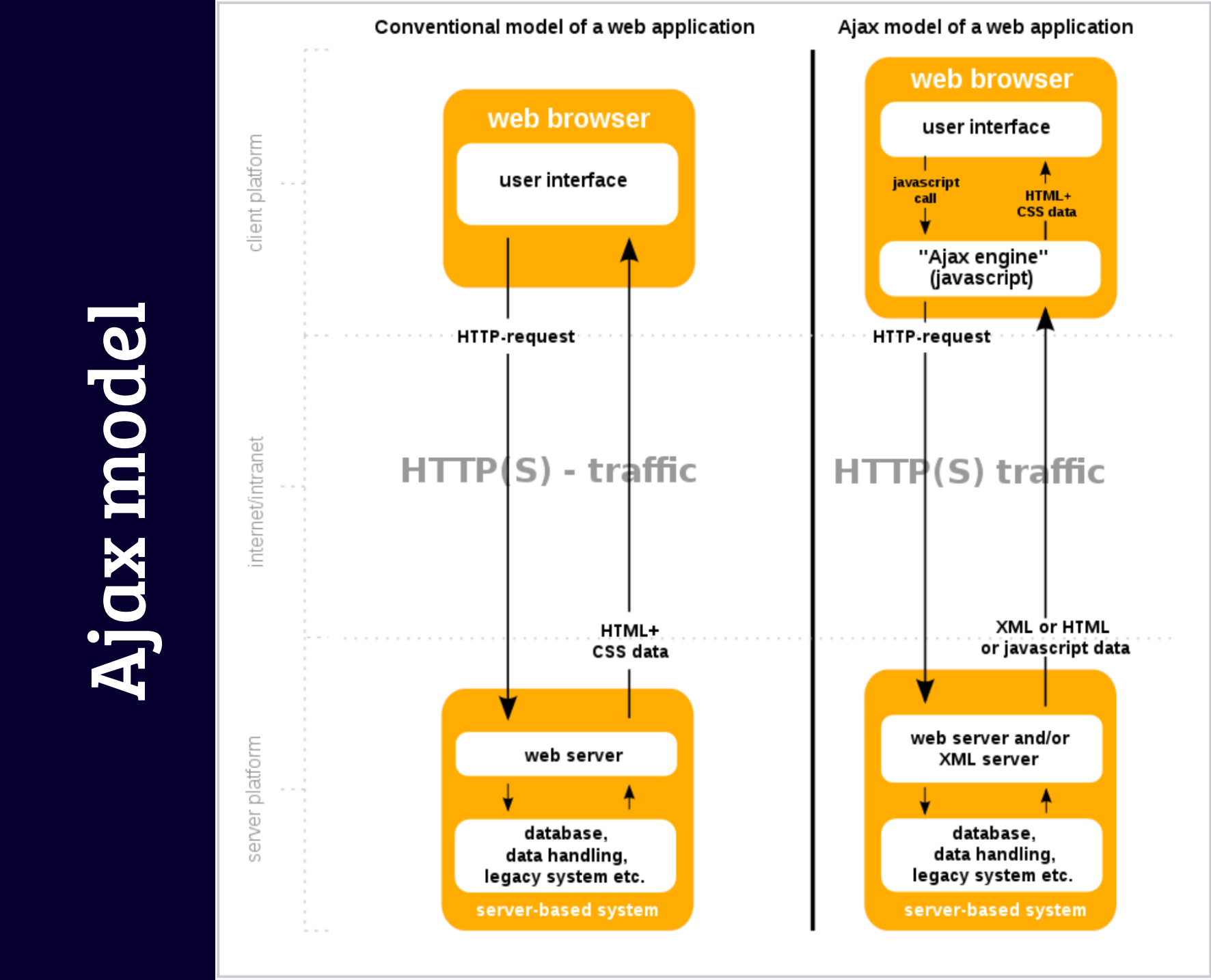
Requests

- 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();
            }
        }
    }
})
```

Advanced JS

Sessions

- If your project uses session auth, browser already stores and sends the cookies headers

- If it uses token auth, you are responsible for storing and using the token

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

- Set `Authorization` header with the appropriate value

- This key-value store is Persistent, even after closing tab/browser
  - Unless you clear history

Let vs var in for loops (optional)

- let does not support redeclaration
- But what happens in a for loop?
  - It is in fact redeclared each time
- Not the case with var

Let vs var in for loops (optional)

- What is the difference between these two codes?

- The top code print the same value of i

```
for(var i = 1; i <= 5; i++) {
    setTimeout(function() {
        console.log('Value of i : ' + i);
    },100);
}


for(let i = 1; i <= 5; i++) {
    setTimeout(function() {
        console.log('Value of i : ' + i);
    },100);
}
```

Arrow functions

- A more convenient way to define functions
- Almost equivalent to regular functions
  - More on that later

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

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

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

Simplify even further

- Today, for loops and if statements are rarely used
- Instead of a for loop, use forEach or map
- Example:

```
var names = ["ali", "hassan"]
names.forEach((item, index) => console.log(item + " at " + index))
upper = names.map(item => item.toUpperCase())
```

Simplify even further

- Take out elements with a specific condition
- Use the filter method instead of for loop and if
- Example:

```
let students = [{name: "John", id: 1}, {name: "Ali", id:2}]
let john = students.filter(item => item.name === "John")
```

Simplify even more further!

- reduce lets you do a lot of cool things with just 1 inline arrow function

- Example:

```
let maxCredit = employee.reduce((acc, cur) =>
Math.max(cur.credit, acc), Number.NEGATIVE_INFINITY)
```

Power of arrow functions!

Regular functions

```
var totalJediScore = personnel
  .filter(function (person) {
    return person.isForceUser;
  })
  .map(function (jedi) {
    return jedi.pilotingScore + jedi.shootingScore;
  })
  .reduce(function (acc, score) {
    return acc + score;
  }, 0);
```

Arrow functions

```
const totalJediScore = personnel
  .filter(person => person.isForceUser)
  .map(jedi => jedi.pilotingScore + jedi.shootingScore)
  .reduce((acc, score) => acc + score, 0);
```

Source: https://medium.com/poka-techblog/simplify-your-javascript-use-map-reduce-and-filter-bd02c593cc2d

Subtlety

- Regular functions have their own this value
- The object that called the function
  - Methods and event listeners: the actual object/element
  - Global function: global object (window)
- Arrow functions do not have their own this

- Do not use arrow functions as event listeners or object methods

  You can use them as class methods though. PERFECTLY WEIRD ISN'T IT?

- However, unlike regular functions, they can bind (capture) this like any other closure value

```
const person = {
    name: 'Kianoosh',
    sayHi() {
        setTimeout(() =>
            console.log(this.name + " says hi!"), 500);
    }
}
```

- For more information, visit https://www.javascripttutorial.net/es6/when-you-should-not-use-arrow-functions/

Destructuring

Visit https://dmitripavlutin.com/javascript-object-destructuring/

```
const hero = {
  name: 'Batman',
  realName: 'Bruce Wayne'
};

const { name, realName } = hero;

name;     // => 'Batman',
realName; // => 'Bruce Wayne'
```

```
const hero = {
  name: 'Batman',
  realName: 'Bruce Wayne'
};

const { name, ...realHero } = hero;

realHero; // => { realName: 'Bruce Wayne' }
```

```
const heroes = [
  { name: 'Batman' },
  { name: 'Joker' }
];

const names = heroes.map(
  function({ name }) {
    return name;
  }
);

names; // => ['Batman', 'Joker']
```

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

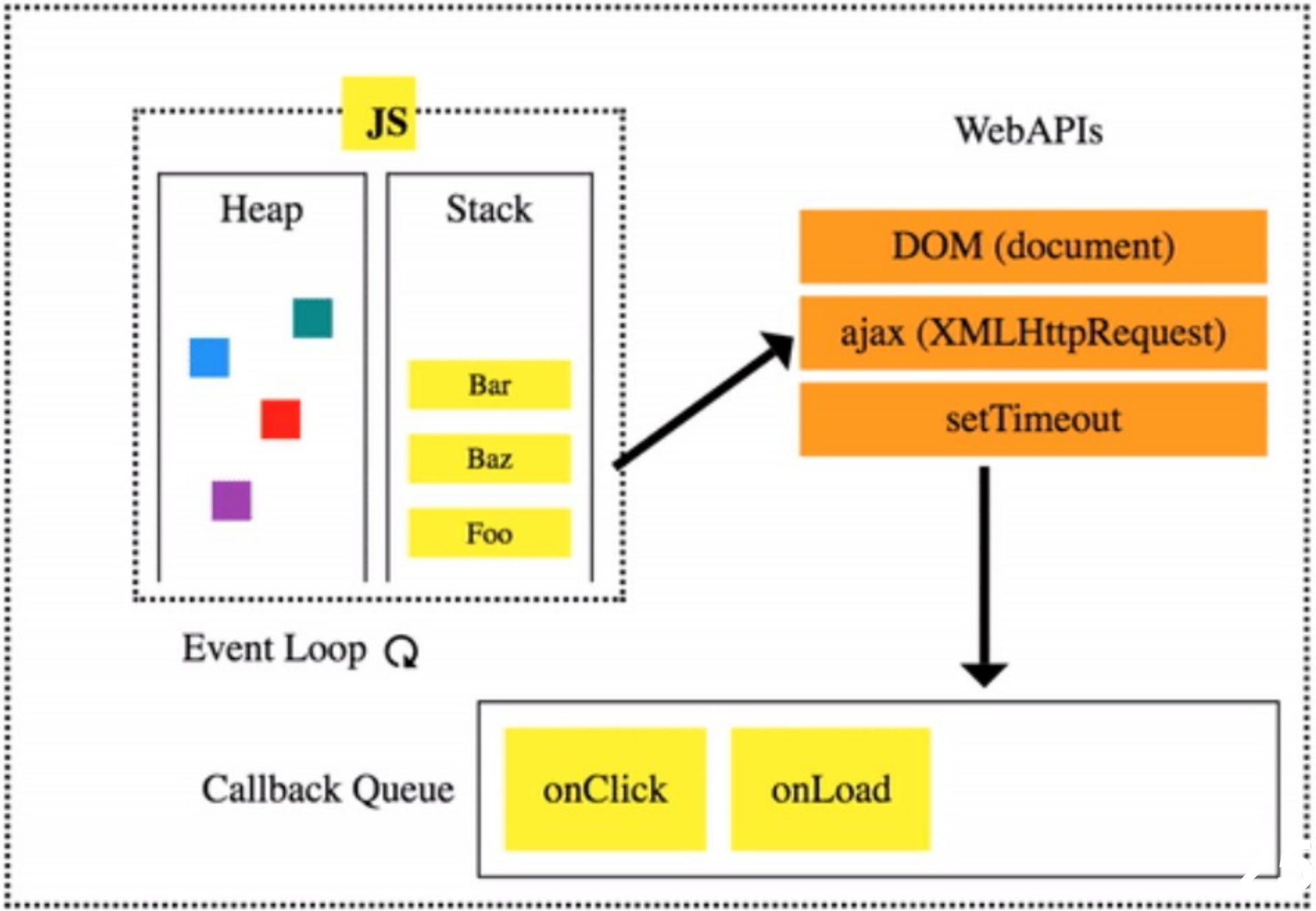
```
$(document).ready(…)
element.addEventListener(…)
$("p > button").click(function(){…})
```

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 session

- Asynchronous requests: Ajax

- Advanced JS (React prep)
  - Arrow functions
  - Promises
  - Fetch API

Next session

- Single-page applications
  - React intro
- JSX
- React application
  - Props
  - Events
  - State