ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

ΚΕΝΤΡΟ ΕΠΙΜΟΡΦΩΣΗΣ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗΣ

JavaScript Asynchronous Programming

Αθανάσιος Ανδρούτσος

Κατανεμημένα Συστήματα

- Στα κατανεμημένα συστήματα με αρχιτεκτονική client-server, οι clients στέλνουν requests στον Server ο οποίος απαντά με responses
- Υπάρχουν δύο προσεγγίσεις στο πως ο Server εξυπηρετεί τα αιτήματα των πελατών
  - Iterative Servers
  - Concurrent Servers

Iterative Approach

- Στους επαναληπτικούς εξυπηρετητές κάθε αίτηση εξυπηρετείται κατ' αποκλειστικότητα και με τη σειρά που έρχεται (FIFO Queue)
- Αυτό υπονοεί ότι ο server μπλοκάρει (blocking mode) όταν εξυπηρετεί μία αίτηση, περιμένοντας να τελειώσει, ώστε να εξυπηρετήσει την επόμενη αίτηση
- Η προσέγγιση αυτή είναι καλή όταν δεν έχουμε υψηλή συχνότητα ταυτόχρονων requests. Όταν έχουμε πολλά requests ταυτόχρονα, τα requests περιμένουν σε μια ουρά (FIFO Queue) για να εξυπηρετηθούν σε First-Come First-Served basis με αποτέλεσμα να δημιουργούνται καθυστερήσεις (latency) κάτι το οποίο κάνει downgrade το Quality of Service (QoS) και επομένως δεν είναι αποδεκτό.

Concurrent Approach

- Οι ‘Ταυτόχρονοι’ ή ασύγχρονοι εξυπηρετητές, εξυπηρετούν ‘ταυτόχρονα’ τις αιτήσεις των πελατών (clients). Οι δύο βασικές προσεγγίσεις στον ‘ταυτοχρονισμό’ είναι οι ακόλουθες
  - **Multi-threaded**. Στην προσέγγιση αυτή για κάθε αίτημα δημιουργείται ένα thread (νήμα) από το Λειτουργικό Σύστημα. Πολλοί Web Servers λειτουργούν με τη λογική one-thread per connection
  - **Single-Threaded Event-Loop**. Στην προσέγγιση αυτή υπάρχει ένα ‘main’ thread ενώ υπάρχει και μία ουρά (queue). Κάθε request εισάγεται στην ουρά και εξυπηρετείται ασύγχρονα. Το main thread δεν κάνει blocking

Multi-threaded apps

- Η δημιουργία πολλών threads έχει πλεονεκτήματα και μειονεκτήματα
  - **Πλεονεκτήματα**. Εκμεταλλεύεται τα σύγχρονα multi-core συστήματα και κάνει dispatch τα threads σε διαφορετικούς πυρήνες, οπότε έχουμε πραγματικό παραλληλισμό. Επομένως σε εφαρμογές που είναι **CPU-bounded** (δηλ. εξαρτώνται από το CPU όπως Scientific simulations, data analytics, cryptocurrency mining, 3D Rendering, video-audio processing) τα multi-threaded συστήματα έχουν καλό performance
  - **Μειονεκτήματα**. Οι επεξεργαστές χρησιμοποιούν την τεχνική context switching όταν τα threads είναι idle (π.χ. όταν περιμένουν τα αποτελέσματα από ένα αρχείο ή ΒΔ). Το context switching είναι ακριβό σε όρους χρόνου. Επομένως όταν έχουμε εφαρμογές που είναι **I/O Bounded** (δηλ. εξαρτώνται από το I/O rate με το Filesystem, όπως APIs, Microservices, Online Gaming, Collaborative apps), οι επιδόσεις των multi-threaded συστημάτων είναι χαμηλές

Single-thread event-loop

- Σε εφαρμογές που είναι I/O bounded, θα θέλαμε όλα τα I/O operations να είναι non-blocking, δηλαδή να εκτελούνται ταυτόχρονα (με βάση πάντα τους περιορισμούς που ισχύουν για όλα τα συστήματα ΒΔ)
- Σε αυτές τις περιπτώσεις μπορεί να έχουμε ένα main thread με μία queue όπου κάθε ασύγχρονο αίτημα να εισάγεται στην ουρά και να εκτελείται σε ξεχωριστό thread και να ενημερώνει (event) όταν εκτελεσθεί ελαχιστοποιώντας τον αριθμό των threads καθώς και του context-switching

JavaScript

- Τόσο στον browser όσο και στο Node.js το runtime της JavaScript χρησιμοποιεί **single-thread event loop** παρέχοντας καλύτερο performance σε εφαρμογές που είναι I/O bounded όπως εφαρμογές API με ΒΔ
- Επομένως η **JavaScript χρησιμοποιεί εγγενώς asynchronous programming** ώστε να μπορεί το main thread να μην κάνει blocking σε long-running tasks (HTTP Requests, User input) αλλά να είναι *responsive* και σε άλλα events

Synchronous Programming (1)

primes-sync.html

testbed > primes-sync.html > html > body > div > input#quota

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Document</title>
7  </head>
8  <body>
9      <div>
10         <label for="quota"></label><br />
11         <input type="text" id="quota" value="1000000">
12         <input type="text" name="random-txt" id="randomTxt">
13     </div>
14     <button type="button" id="generateBtn">Generate Primes</button>
15     <button type="button" id="resetBtn">Reset</button>
```

Synchronous Programming (2)

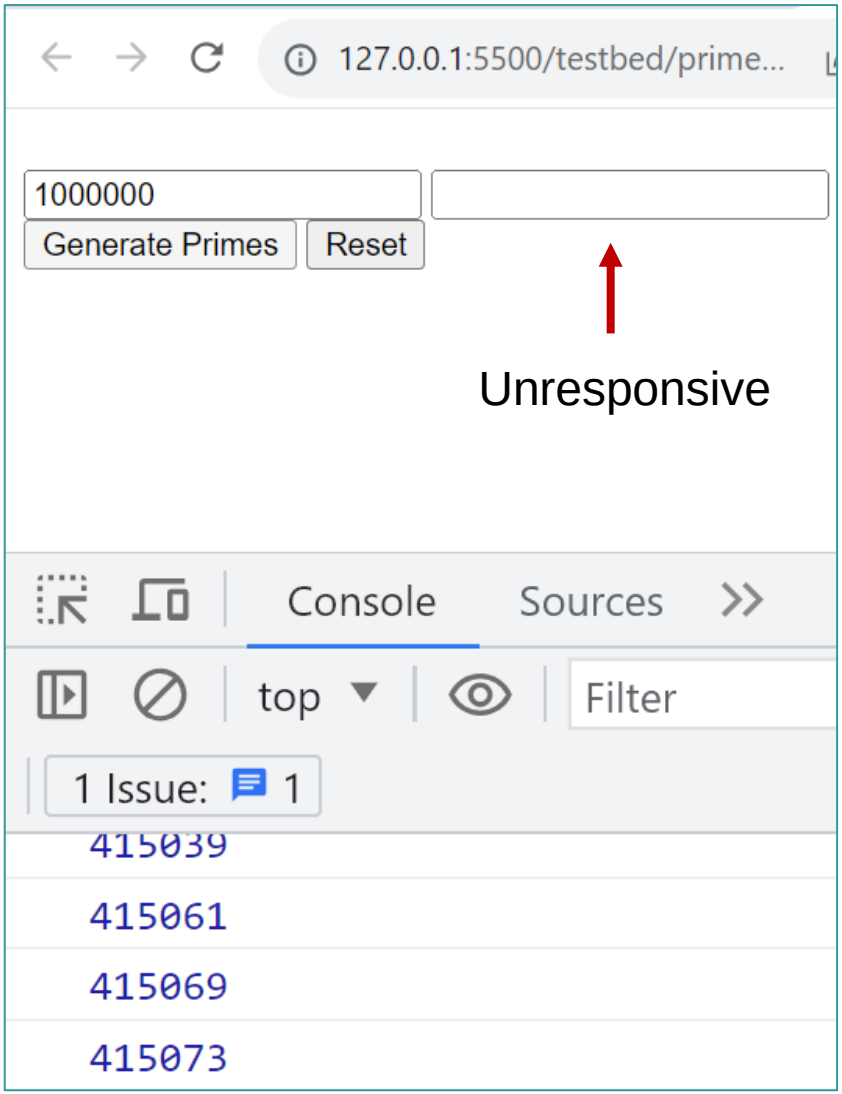

```
17      <script>
18          const QUOTA = 1000000
19
20          document.querySelector('#generateBtn').addEventListener('click', function() {
21              generateBtnClicked()
22          })
23
24          document.querySelector('#resetBtn').addEventListener('click', function() {
25              resetBtnClicked()
26          })
27
28          function generateBtnClicked() {
29              let primesDom = document.querySelector('#quota').value
30              if (primesDom > QUOTA) {
31                  console.log('Error. Large prime limit')
32                  return
33              }
34              generatePrimes(primesDom)
35          }
37          function resetBtnClicked() {
38              document.querySelector('#quota').value = QUOTA
39          }
```

Synchronous Programming (3)


```
41      function generatePrimes(quota) {
42          const primes = []
43          let isPrime
44
45          for (let i = 1; i <= quota; i++) {
46              isPrime = true
47              for (let j = 2; j <= Math.sqrt(i); j++) {
48                  if (i % j == 0) {
49                      isPrime = false
50                      break
51                  }
52              }
53              if (isPrime) {
54                  primes.push(i)
55                  console.log(i)
56              }
57
58          }
59      }
```

- Πρώτος είναι ένας αριθμός που διαιρείται με τον εαυτό του και την μονάδα
- Πρακτικά μπορούμε να ελέγξουμε τους αριθμούς μέχρι την τετραγωνική ρίζα του αριθμού

Synchronous Programming (4)



- Κατά τη διάρκεια που τρέχει το Generate Primes, αν κάνουμε κλικ στο **δεξί textbox, είναι πλήρως unresponsive**

Async Programming (1)

- Χρειαζόμαστε επομένως ένα μηχανισμό που:

1. Να επιτρέπει στο main thread να επιστρέφει αμέσως και να μην κάνει blocking περιμένοντας μία long-running function όπως η generatePrimes() να ολοκληρώσει, και
2. Να μας ενημερώνει –και να επιστρέφει το αποτέλεσμα- όταν η εκτέλεση της long-running function ολοκληρωθεί

Async Programming (2)

- Για να ενημερωνόμαστε όταν μία long-running process ολοκληρώσει την εκτέλεσή της, στον ασύγχρονο προγραμματισμό χρησιμοποιούμε events και event handlers
- Το event μπορεί να είναι κάτι σαν: *long-running process has completed* και ο event handler μόλις το *long-running process has completed* να επεξεργάζεται το result ή/και να το επιστρέφει

Async Programming (2)


```
26  * Fetches a movie from the Movies API.
27  * This function defines handling for both successful and failed(movie not found, api ur
28  */
29 function fetchMovieFromApi(title) {
30     let ajaxRequest = new XMLHttpRequest()
31     ajaxRequest.open("GET", `http://www.omdbapi.com/?t=${title}&apikey=c[redacted]`, true)
32     ajaxRequest.timeout = 5000  //timeout after 5 seconds
33     ajaxRequest.ontimeout = (e) => onApiError()
34     ajaxRequest.onreadystatechange = function() {
35         if (ajaxRequest.readyState === 4)
36         {
37             if(ajaxRequest.status === 200) {
38                 handleResults(JSON.parse(ajaxRequest.responseText))
39             }
40             else {
41                 onApiError()
42             }
43         }
44     }
45     ajaxRequest.send()
46 }
```

- Για παράδειγμα το XMLHttpRequest API μας επιτρέπει να κάνουμε HTTP requests. Μιας και τα HTTP Requests μπορεί να χρειάζονται αρκετό χρόνο για να ολοκληρωθούν κάνουμε register ένα *onreadystatechange* event listener στο XMLHttpRequest και όταν το event του *readyState* γίνει fire με τιμή === 4, τότε έχει ολοκληρωθεί το request

Async Programming (3)

Asynchronous development model

Callbacks

Synchronous

```
public int compute(int a, int b) {
   return ...;
}
```

```
int res = compute(1, 2);
```

Asynchronous

```
public void compute(int a, int b,
   Handler<Integer> handler) {
   int result = a + b;
   handler.handle(result);
}
```

```
compute(1, 2, res -> {
   // Called with the result
});
```

- Γενικά στον ασύγχρονο προγραμματισμό, περνάμε τυπικά και μία επιπλέον παράμετρο που είναι το **callback (event handler)**, που ενεργοποιείται όταν είναι έτοιμο το result και έχει πρόσβαση στο result, το οποίο στη συνέχεια το κάνει handle. Ο Handler μπορεί να επεξεργαστεί το result και μπορεί να το επιστρέψει στον caller κάνοντάς τον notify ότι το result είναι έτοιμο

Async Programming (4)

- Στον ασύγχρονο προγραμματισμό ωστόσο δεν μπορούν όλα τα tasks να τρέχουν παράλληλα, όπως για παράδειγμα όταν εξαρτάται το ένα από το άλλο
- Αν για παράδειγμα έχουμε την ακολουθία συναρτήσεων: fetch -> handle -> show, για να τρέξει η handle πρέπει να έχει ολοκληρωθεί η fetch και για να τρέξει η show πρέπει να έχει ολοκληρωθεί η handle
- Σε αυτές τις περιπτώσεις θα έχουμε το ακόλουθο pattern στον σύγχρονο και τον ασύγχρονο προγραμματισμό (βλ. επόμενη διαφάνεια)

Sync vs Async Programming


```
function doTheSyncSequence() {
    let result = fetchSyncHello('Hello')
    result = handleSync(result)
    showSync(result)
}

function fetchSyncHello(token) {
    return token + " world of"
}

function handleSync(res) {
    return res + " Coding Factory"
}

function showSync(res) {
    const result = res + ' AUEB'
    console.log(`${result}`)
}

doTheSyncSequence()
```

- Sync – Async Programming →

```
function doTheAsyncSequence() {
    fetchAsyncHello('Hello', res1 => {
        handleAsync(res1, res2 => {
            showAsync(res2, res3 => {
                console.log(`${res3}`)
            })
        })
    })
}

function fetchAsyncHello(helloToken, callbackHello) {
    const result = helloToken + " world of"
    callbackHello(result)
}

function handleAsync(res, callbackHandle) {
    const result = res + " Coding Factory"
    callbackHandle(result)
}

function showAsync(res, callbackShow) {
    const result = res + " - AUEB"
    callbackShow(result)
}

doTheAsyncSequence()
```

Callback hell – Pyramid of doom


```
22  function doTheAsyncSequence() {
23      fetchAsyncHello('Hello', res1 => {
24          handleAsync(res1, res2 => {
25              showAsync(res2, res3 => {
26                  console.log(`${res3}`)
27              })
28          })
29      })
30  }
31
32  function fetchAsyncHello(helloToken, callbackHello) {
33      const result = helloToken + " world of"
34      callbackHello(result)
35  }
36
37  function handleAsync(res, callbackHandle) {
38      const result = res + " Coding Factory"
39      callbackHandle(result)
40  }
41
42  function showAsync(res, callbackShow) {
43      const result = res + " - AUEB"
44      callbackShow(result)
45  }
46
47  doTheAsyncSequence()
```

- Όπως παρατηρούμε, σε αυτές τις περιπτώσεις των sequential tasks, έχουμε callbacks μέσα σε callbacks, με αποτέλεσμα ο κώδικας να γίνεται δυσανάγνωστος
- Αυτή η μορφή του κώδικα ονομάζεται callback hell ή pyramid of doom (μιας και οι εσοχές της ενθυλάκωσης σχηματίζουν μία πυραμίδα)

Callback Hell

- Για αυτό τον λόγο, του callback hell, στον ασύγχρονο προγραμματισμό, τα πρόσφατα API δεν χρησιμοποιούν callbacks αλλά ένα άλλο μηχανισμό, τον μηχανισμό των **Promises**
- Το **Promise** λειτουργεί ως **wrapper κλάση** του αποτελέσματος, **ενώ παρέχει και μεθόδους** (handlers) για να έχουμε πρόσβαση στο αποτέλεσμα είτε είναι επιτυχές (success / resolved) είτε όχι (failure / rejected)

Promises (1)

- Τα promises είναι JavaScript objects που περιέχουν το αποτέλεσμα ενός ασύγχρονου task
- Περιέχουν τόσο **το αποτέλεσμα**, όσο και **μεθόδους** για να έχουμε πρόσβαση στο αποτέλεσμα ή σε τυχόν λάθη που μπορεί να συμβούν (errors)
- Τα Promises είναι ο βασικός πυλώνας του ασύγχρονου προγραμματισμού στη σημερινή JavaScript

Promises (2)

- Τα promises είναι επομένως wrapper objects στα οποία μπορούμε να κάνουμε attach και handlers
- Κατά την διάρκεια εκτέλεσης ενός ασύγχρονου task μπορεί ένα **promise** να βρίσκεται σε μία από τις τρεις παρακάτω φάσεις:
  - **Pending**. Το ασύγχρονο task έχει ξεκινήσει (promise initiated) αλλά δεν έχει ολοκληρωθεί
  - **Fulfilled**. Το async task έχει ολοκληρωθεί (promise resolved) είτε επιτυχώς ή με error
  - **Rejected**. Το async task δεν έχει γίνει fulfilled (promise rejected), λόγω λάθους

Create a Promise - Constructor

promisestest > JS promise1.js > ...

```
1  let myPromise = new Promise((resolve, reject) => { })
2  let myPromise2 = new Promise(function(resolve, reject) { })
```

```
let myPromise = new Promise(function(resolve, reject) {
    const result = Math.floor(Math.random() * 5)    // range: 0 - 4
    if (result === 0) resolve(result)
    else reject('Error in guess')
})
```

- Μπορούμε να δημιουργήσουμε ένα promise object με τον Promise constructor. Η παράμετρος του Promise constructor είναι ένα **executor function** (είτε ως arrow function ή ως anonymous function) που εκτελείται αυτόματα και παρέχει τα resolve και reject functions που κάνουν populate το state του promise. Το resolve είναι το function που αλλάζει το state του promise σε fulfilled (resolved) ενώ το reject είναι το function που αλλάζει το state του promise σε rejected
- Ο executor αναμένουμε να εκτελέσει κάποιο asynchronous task (στο παράδειγμα έστω ότι είναι η Math.random()) και μόλις ολοκληρωθεί να καλέσει την resolve ή την reject

Create a promise – static factory


```
4   let myPromise3 = Promise.resolve('a value')
5   let myPromise4 = Promise.reject('an error')
```

- Μπορούμε να δημιουργήσουμε promises και με τη χρήση των static factories Promise.resolve() και Promise.reject() όταν υπάρχουν ήδη οι τιμές (resolved ή rejected) χωρίς executor function

Then-ables (1)


```
Promise.then(onFulfilled, onRejected);
```

- Αντί για τα callbacks που δημιουργούν το callback hell, στα promises έχουμε τα **.then()** που είναι μέθοδοι με δύο παραμέτρους, που αντιστοιχούν σε δύο handlers, έστω: successHandler και failureHandler που ανταποκρίνονται στα events onFulfilled και onRejected
- Τα **.then** τα κάνουμε attach στο promise, **επιστρέφουν promise**, οπότε μπορούν να γίνουν και chaining.
- Ο successHandler ενεργοποιείται όταν και εάν γίνει resolve (fulfill) το promise ενώ ο failureHandler ενεργοποιείται όταν το promise γίνει reject

Then-ables (2)


```
let myPromise = new Promise(function(resolve, reject) {
    const result = Math.floor(Math.random() * 5)    // range: 0 - 4
    if (result === 0) resolve(result)
    else reject('Error in guess')
})

myPromise.then(response => {
    const result = "CF found the random num: " + response
    console.log(`Result:  ${result}`)
}, error => console.log(error))
```

- Το .then παρέχει handlers και για το success (resolve) και για το failure (reject)

Error Handling in Promise


```
myPromise.then(response => {
    const result = "CF found the random num: " + response
    console.log(`Result:  ${result}`)
}, error => console.log(error))
```

- Όπως βλέπουμε παραπάνω η 2η παράμετρος του **then**, δηλαδή ο onRejected handler, λειτουργεί όπως η catch σε ένα κλασσικό try/catch block.
- Αν δεν έχουμε onReject handler σε ένα then, το τυχόν error γίνεται catch στο 1ο επόμενο onReject handler σε επόμενο / μεθεπόμενο κλπ. then μιας και τα then μπορούν να γίνουν chaining
- Αντί να έχουμε onReject handlers σε κάθε then, μπορούμε να ορίσουμε ένα .catch στο τέλος των thenables . Αν λοιπόν δεν ορίσουμε onRejection callbacks στα .then, τότε όλα τα errors γίνονται catch στην .catch(onRejected)

Catch


```
62  let myPromise = new Promise(function(resolve, reject) {
63      const result = Math.floor(Math.random() * 5)    // range: 0 - 4
64      if (result === 0) resolve(result)
65      else reject('Error in guess')
66  })
67
68  myPromise
69      .then(response => {
70          const result = "CF found the random num: " + response
71          console.log(`Result:  ${result}`)
72      })
73      .catch(error => console.log(error))
```

- Τα .catch είναι μία μέθοδος που μπορούμε να κάνουμε attach σε ένα promise (τα then επιστρέφουν promise) ώστε να έχουμε ένα σημείο χειρισμού των λαθών

Multiple then and one catch


```
62  let myPromise = new Promise(function(resolve, reject) {
63      const result = Math.floor(Math.random() * 5)    // range: 0 - 4
64      if (result === 0) resolve(result)
65      else reject('Error in guess')
66  })
67
68  myPromise
69      .then(response => "CF found the random num: " + response)
70      .then(res => console.log(`Result:  ${res}`))
71      .catch(error => console.log(error))
```

- Παρατηρούμε το chaining των then. Το catch χειρίζεται οποιοδήποτε error συμβεί, σε οποιοδήποτε then, εφόσον στα then δεν υπάρχουν onReject handlers
- Παρατηρήστε ότι στο 1ο then, το return υπονοείται επειδή δεν υπάρχουν { }

Promises και Async actions

Τα promises είναι ο βασικός μηχανισμός στον ασύγχρονο προγραμματισμό.

Όταν έχουμε κάποιο long running task, **τυπικά να λάβουμε δεδομένα από κάποιο API**, θα θέλαμε να τρέξει ασύγχρονα ώστε να μην μπλοκάρει το main thread

Τα ασύγχρονα tasks σε ένα executor function είναι συνήθως κλήσεις GET, POST, PUT, DELETE προς κάποιο URI-based resource

Products

mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json


```
[
  {
    "name" : "baked beans",
    "price" : 0.40,
    "image" : "beans.jpg",
    "type" : "vegetables"
  },
  {
    "name" : "hot dogs",
    "price" : 1.99,
    "image" : "hotdogs.jpg",
    "type" : "meat"
  },
  {
    "name" : "spam",
    "price" : 2.85,
    "image" : "spam.jpg",
    "type" : "meat"
  },
  {
    "name" : "refried beans",
    "price" : 0.99,
    "image" : "refried.jpg",
    "type" : "vegetables"
  },
  {
    "name" : "kidney beans",
    "price" : 0.58,
    "image" : "kidney.jpg",
    "type" : "vegetables"
  },
  {
    "name" : "garden peas",
    "price" : 0.52,
    "image" : "gardenpeas.jpg",
    "type" : "vegetables"
  },
```


- Έστω το products.json resource που θέλουμε να το κάνουμε GET ασύγχρονα και στη συνέχεια να εμφανίσουμε την 1η εγγραφή του πίνακα

Fetch API (1)


```
1  $(document).ready(function() {
2      fetchAProduct()
3  })
4
5  /**
6   * Fetches the first product of an array of products.
7   */
8  function fetchAProduct() {
9      fetchProducts()
10     .then(response => {
11         if (!response.ok) {
12             return Promise.reject(`HTTP status error: ${response.status}`)
13         }
14         return response.json()
15     })
16     .then(data => console.log(data[0]))
17     .catch(error => console.log(error.toString()))
18 }
19
20 /**
21  * Fetches all products from a store.
22  * @returns JSON representation of the products
23  */
24 function fetchProducts() {
25     return fetch(`https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json`)
26 }
```

- Το Fetch API (γραμμή 25) είναι ο promise-based ‘αντικαταστάτης’ του XMLHttpRequest. Επιστρέ**φει ένα promise**

Fetch API (2)


```
24 function fetchProducts() {
25     return fetch(`https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json`)
26 }
```

- Στην απλή του μορφή η μέθοδος fetch έχει μία παράμετρο, το URL, ενώ το HTTP method by default είναι το GET

Fetch API (3)


```
// URL of the API endpoint
const apiUrl = 'https://jsonplaceholder.typicode.com/posts';

// Headers for the request
const headers = new Headers();
headers.append('Content-Type', 'application/json');
headers.append('Authorization', 'Bearer ACCESS_TOKEN');

// Data to send in the POST request
const postData = {
    userId: 12,
    id: 101,
    title: 'Coding Factory AUEB',
    body: 'Software Engineering Education'
};


// Request configuration
const requestOptions = {
    method: 'POST',
    headers: headers,
    body: JSON.stringify(postData)
};
```

- Σε ένα πιο εκτεταμένο παράδειγμα, μπορούμε να κάνουμε populate τους headers και τα data (payload) καθώς και στη συνέχεια τα requestOptions
- Χρησιμοποιούμε το typicode, ένα REST API για testing

Fetch API (4)


```
25  /**
26   * Posts the request and fetches back the id
27   * of the newely created resource from the API.
28   */
29  fetch(apiUrl, requestOptions)
30      .then(response => {
31          if (!response.ok) {
32              throw new Error(`HTTP error! Status: ${response.status}`);
33          }
34          console.log('Post request successful');
35          return response.json()
36      })
37      .then(data => {
38          const newId = data.id;
39          console.log('New post ID:', newId);
40      })
41      .catch(error => {
42          console.error('Fetch error:', error);
43      });
```

- Παρατηρούμε ότι στο FETCH API, errors θεωρούνται τα connection errors, όχι τα HTTP 404, κλπ.
- Επομένως θα πρέπει να ελέγξουμε στο then για response.ok (γρ. 31)

Fetch API vs XMLHttpRequest

- Το Fetch API δεν έχει εύκολο τρόπο να ορίζουμε timeout ενώ όπως είδαμε θεωρεί valid όλα τα HTTP codes, (error όπως είδαμε θεωρεί μόνο τα network errors)
- Μπορούμε ωστόσο να χρησιμοποιούμε και το κλασσικό XMLHttpRequest με promises (βλ. επόμενη διαφάνεια)

Promisified XHR (1)


```
16  function getXHRPromise(url) {
17      return new Promise(function(resolve,reject) {
18          let ajaxRequest = new XMLHttpRequest()
19          ajaxRequest.open("GET", url, true)
20          ajaxRequest.timeout = 5000  //timeout after 5 seconds
21          ajaxRequest.ontimeout = (e) => reject()
22          ajaxRequest.onreadystatechange = function() {
23              if (ajaxRequest.readyState == 4)
24              {
25                  if(ajaxRequest.status === 200) {
26                      resolve(JSON.parse(ajaxRequest.responseText))
27                  }
28                  else {
29                      reject()
30                  }
31              }
32          }
33          ajaxRequest.send()
34      })
35  }
```

Πρόκειται για την τεχνική που έχουμε δει, απλώς δημιουργούμε ένα νέο promise και ορίζουμε τις resolve και reject

Ελέγχει το status ενώ μετατρέπει και σε json με JSON.parse()

Promisified XHR (2)


```
1  $(document).ready(function() {
2      fetchAProduct()
3  })
4
5  function fetchAProduct() {
6      fetchProducts()
7      .then(data => console.log(data))
8      .catch(error => console.log(error))
9  }
10
11 function fetchProducts() {
12     let url = `https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json`
13     return getXHRPromise(url)
14 }
15
```

- Η fetchAProduct() δεν χρειάζεται να μετατρέψει σε json ούτε να ελέγξει για ok status μιας και η fetchProducts() καλεί στο σώμα της την getXHRPromise() που όπως αναφέραμε εκτελεί τις παραπάνω ενέργειες

Async / await (1)

- Η JavaScript έχει εισάγει πρόσφατα ένα πιο απλό συντακτικό για να δουλεύουμε με τα promises, πιο κοντά στον κλασσικό τρόπο του σύγχρονου προγραμματισμού
- Ένα function μπορεί να χαρακτηριστεί **async** όταν στο σώμα του περιμένει (**await**) ένα promise να γίνει resolve
- Μία async function πάντα επιστρέφει promise

Async / await (2)


```
1  $(document).ready(function() {
2      fetchAProduct()
3  })
4
5  async function fetchAProduct() {
6      try {
7          const response = await fetchProducts()
8          console.log(response)
9      } catch (error) {
10         console.log(error)
11     }
12 }
13
14 async function fetchProducts() {
15     let url = `https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json`
16     return await getXHRPromise(url)
17 }
```

- Η fetchAProduct() και η fetchProducts() είναι async. Το async ενεργοποιεί χώρο για το await (ότι είναι async πρέπει να έχει στο σώμα του await)

Async / await (3)


```
19  function getXHRPromise(url) {
20      return new Promise(function(resolve,reject) {
21          let ajaxRequest = new XMLHttpRequest()
22          ajaxRequest.open("GET", url, true)
23          ajaxRequest.timeout = 5000  //timeout after 5 seconds
24          ajaxRequest.ontimeout = (e) => reject()
25          ajaxRequest.onreadystatechange = function() {
26              if (ajaxRequest.readyState == 4)
27              {
28                  if(ajaxRequest.status === 200) {
29                      resolve(JSON.parse(ajaxRequest.responseText))
30                  }
31                  else {
32                      reject()
33                  }
34              }
35          }
36          ajaxRequest.send()
37      })
```

- Η getXHRPromise υλοποιεί το promise με τον ασύγχρονο μηχανισμό του XHR
- Δημιουργεί ένα νέο promise και εισάγει τις resolve και reject