Κεφάλαιο 11. Προγραμματισμός στην πλευρά του εξυπηρετητή

Σύνοψη

Στο κεφάλαιο θα χρησιμοποιήσουμε την JavaScript ως γλώσσα προγραμματισμού εφαρμογών από την πλευρά του εξυπηρετητή στο προγραμματιστικό περιβάλλον της Node.js. Η Node.js παρέχει ένα οικοσύστημα για την υποστήριξη της ανάπτυξης εφαρμογών και συνοδεύεται από εργαλεία όπως το npm για τη διαχείριση πακέτων. Θα παρουσιαστούν τα δύο κύρια συστήματα βιβλιοθηκών στην JavaScript, το CommonJS και το ES6 Modules (ESM), καθώς και το πώς μπορούμε να γράψουμε και να χρησιμοποιήσουμε βιβλιοθήκες ESM. Θα χρησιμοποιήσουμε κάποιες βασικές βιβλιοθήκες της Node.js, την fs και την http, στα παραδείγματά μας σε τρεις εκδοχές, σύγχρονα, ασύγχρονα με συναρτήσεις επιστροφής και ασύγχρονα με τη διεπαφή των Promises. Με την ευκαιρία αυτή θα χρησιμοποιήσουμε τη διεπαφή async/await σε κώδικα που χρησιμοποιεί τα Promises.

Επιπλέον, θα παρουσιαστούν μερικά πολύ βασικά εργαλεία και βοηθήματα για την ανάπτυξη εφαρμογών, όπως το nodemon και το Visual Studio Code, για την αποσφαλμάτωση των προγραμμάτων μας, καθώς και τη δημοσίευση των εφαρμογών διαδικτύου μας στο Heroku.

Προαπαιτούμενη γνώση

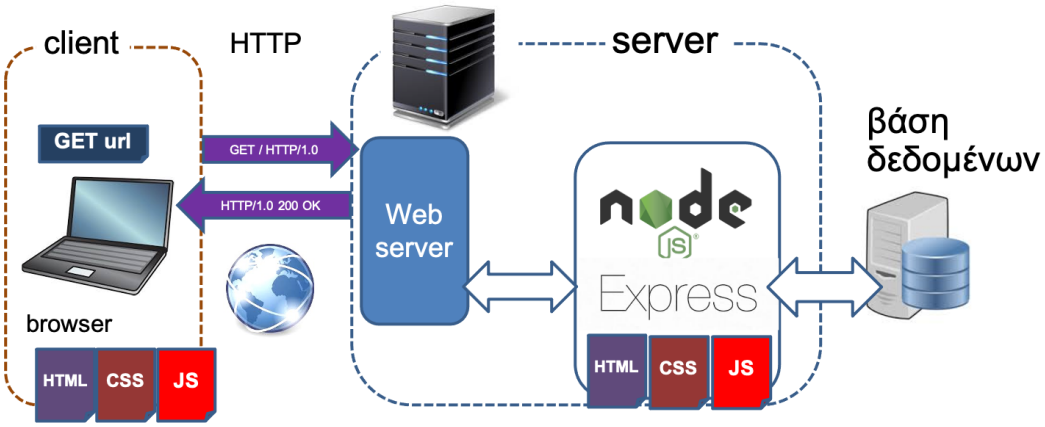
Για τη μελέτη αυτού του κεφαλαίου είναι απαραίτητη η εξοικείωση με τη γλώσσα JavaScript, όπως έχει παρουσιαστεί στα προηγούμενα κεφάλαια (7, 8, 9 και 10). Απαραίτητη είναι και η μελέτη του κεφαλαίου 1, όπου παρουσιάζεται η αρχιτεκτονική του διαδικτύου και το πρωτόκολλο HTTP.

11.1 Εισαγωγή

Έχοντας φτάσει στο κεφάλαιο αυτό, έχουμε αποκτήσει μια πολύ καλή γνώση των βασικών συστατικών μιας διαδικτυακής εφαρμογής από την πλευρά του φυλλομετρητή, δηλαδή του *πελάτη*, για να χρησιμοποιήσουμε την ορολογία του μοντέλου πελάτη-εξυπηρετητή. Όπως είδαμε, το περιεχόμενο προβάλλεται στον φυλλομετρητή χρησιμοποιώντας τις τεχνολογίες HTML, CSS και JavaScript. Το περιεχόμενο αυτό είναι μία ή περισσότερες *στατικές* σελίδες, δηλαδή σελίδες που είναι γραμμένες από πριν και αποθηκευμένες σε κάποιον υπολογιστή, τον εξυπηρετητή. Ο εξυπηρετητής τις παρέχει σε όποιον πελάτη τις ζητήσει, χωρίς μεταβολές και χωρίς να παρέμβει στο περιεχόμενο. Αυτή είναι και η αρχική ιδέα πίσω από τον παγκόσμιο ιστό, δηλαδή αρχεία HTML που βρίσκονται σε διαφορετικούς εξυπηρετητές και τα οποία είναι συνδεμένα μέσω υπερσυνδέσμων.

Στις περισσότερες περιπτώσεις είναι πιο χρήσιμο η εφαρμογή μας να επιστρέφει περιεχόμενο που να είναι προσαρμοσμένο στον χρήστη, να είναι δηλαδή αποτέλεσμα κάποιας επεξεργασίας. Σε αυτό και στα επόμενα κεφάλαια θα παρουσιαστούν οι διαδικτυακές εφαρμογές από την πλευρά του εξυπηρετητή. Η επεξεργασία πραγματοποιείται στον εξυπηρετητή χρησιμοποιώντας κάποια γλώσσα προγραμματισμού. Στο βιβλίο αυτό η γλώσσα με την οποία γράφουμε εφαρμογές στην πλευρά του εξυπηρετητή θα συνεχίσει να είναι η JavaScript, την οποία έχουμε ήδη δει εκτεταμένα στα προηγούμενα κεφάλαια. Άλλες δημοφιλείς τεχνολογίες για τη συγγραφή προγραμμάτων στην πλευρά του εξυπηρετητή είναι η PHP, η Python, η Java, C# κ.λπ. Στην πραγματικότητα η JavaScript είναι η μοναδική γλώσσα με την οποία μπορούμε να γράψουμε προγράμματα τόσο στην πλευρά του φυλλομετρητή όσο και του εξυπηρετητή.

Στην ***Εικόνα 11.1*** φαίνεται η αρχιτεκτονική πελάτη-εξυπηρετητή όπου παράγονται δυναμικά οι ιστοσελίδες, όπως είδαμε και στην ενότητα 1.5. Ο εξυπηρετητής στο αίτημα του πελάτη (GET) δεν επιστρέφει μια στατική σελίδα HTML, αλλά αναθέτει στη Node.js να ετοιμάσει ένα έγγραφο HTML, ενδεχομένως διαβάζοντας πληροφορίες που υπάρχουν σε κάποια βάση δεδομένων, να ενσωματώσει τυχόν CSS και JavaScript και έπειτα να το επιστρέψει στον πελάτη.



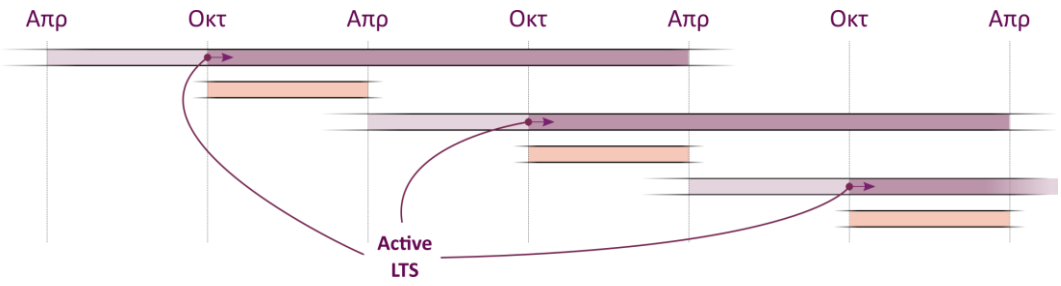
Εικόνα 11.1 *Αρχιτεκτονική μιας τυπικής εφαρμογής που θα αναπτύξουμε στο πλαίσιο του βιβλίου.*

Βέβαια, αν και σημαντική, η εξοικείωση με μια γλώσσα προγραμματισμού δεν είναι πάντα ο πιο καθοριστικός παράγοντας για την επιλογή της στην ανάπτυξη εφαρμογών εξυπηρετητή. Μεγάλη σημασία έχει και το ευρύτερο οικοσύστημα, τα εργαλεία που συνοδεύουν και συμπληρώνουν τη γλώσσα για την ανάπτυξη στον εξυπηρετητή. Όσον αφορά την JavaScript, το κύριο εργαλείο μας είναι η πλατφόρμα Node.js. Η Node.js είναι ένα περιβάλλον με το οποίο μπορούμε να εκτελέσουμε κώδικα γραμμένο σε JavaScript. Μέχρι τώρα έχουμε δει πως ο κώδικας JavaScript εκτελείται από τον φυλλομετρητή από μια μηχανή εκτέλεσης κώδικα JavaScript (Ενότητα 7.1.4). Η Node.js χρησιμοποιεί τη μηχανή JavaScript του φυλλομετρητή Chrome, τη μηχανή V8, με κάποιες αλλαγές που θα δούμε στην πορεία.

11.2 Λήψη και εγκατάσταση της Node.js

Για την παρακολούθηση αυτού του κεφαλαίου είναι απαραίτητη η λήψη και εγκατάσταση της Node.js στον υπολογιστή, μια σχετικά απλή διαδικασία που μπορούμε να ολοκληρώσουμε ακολουθώντας τις οδηγίες στον ιστότοπο της Node.js.

Η Node.js διατίθεται σε διάφορες εκδόσεις, που δημοσιεύονται 2 φορές το χρόνο, κάθε Απρίλιο (με ζυγό αριθμό έκδοσης: 2, 4, 6, 8…) και κάθε Οκτώβριο (με μονό αριθμό έκδοσης 1, 3, 5, 7…). Επιλέγουμε κατά προτίμηση την έκδοση του πιο πρόσφατου Απρίλη, δηλαδή την έκδοση με τον μεγαλύτερο άρτιο αριθμό. Οι εκδόσεις αυτές ονομάζονται LTS (Long-term support - υποστήριξη μεγάλης διάρκειας) και συντηρούνται για διάστημα 3 ετών όσον αφορά κρίσιμα θέματα ασφάλειας. Από τον Απρίλιο του 2022 η τρέχουσα έκδοση LTS είναι η 18. Για την ακρίβεια, η σύσταση είναι να επιλέγουμε την πιο πρόσφατη LTS αφού έχουν περάσει έξι μήνες από τη διάθεσή της (**Εικόνα *11.2***).



Εικόνα 11.2 *Οι εκδόσεις της Node.js που διατίθενται κάθε Απρίλιο υποστηρίζονται για τρία χρόνια (LTS) και έχουν ζυγό αριθμό. Αυτές που διατίθενται κάθε Οκτώβριο έχουν μονό αριθμό.*

Για τη χρήση της Node.js απαιτείται η εξοικείωση με βασικές λειτουργίες σε περιβάλλον γραμμής εντολών, που στα Windows είναι προσβάσιμο από την εφαρμογή PowerShell ή τη γραμμή εντολών και στους υπολογιστές Mac και Linux από το Terminal ή τερματικό.

Μαζί με τη Node.js θα εγκατασταθεί και το εργαλείο **npm** (node package manager) που βοηθά στη δημιουργία και τη διαχείριση του προγράμματός μας, το οποίο θα είναι ένα «πακέτο», καθώς και στην

εγκατάσταση πακέτων που έχουν δημιουργήσει άλλοι.

11.2.1 Μια απλή εφαρμογή σε Node.js

Έχοντας εγκαταστήσει τη Node.js, μπορούμε να γράψουμε και να εκτελέσουμε το πρώτο πρόγραμμά μας και να το αποθηκεύσουμε σε ένα αρχείο με όνομα, για παράδειγμα, index.js:

```
console.log('Καλή σας μέρα!`);
```

Για να τρέξει το πρόγραμμα αυτό αρκεί να εκτελέσουμε στο τερματικό την εντολή

```
node index.js
```

Προϋπόθεση βέβαια είναι το τερματικό μας να είναι στον φάκελο που έχουμε αποθηκεύσει το αρχείο index.js, κάτι που μπορούμε να εξασφαλίσουμε αν ανοίξουμε το τερματικό που είναι ενσωματωμένο στο Visual Studio Code.

11.2.2 Διαφορές με την JavaScript στον φυλλομετρητή

Η JavaScript που εκτελείται από τη Node.js είναι η ίδια με αυτή του φυλλομετρητή, ειδικά αν ο φυλλομετρητής μας είναι ο Goole Chrome ή χρησιμοποιεί τη V8, όπως ο Microsoft Edge, ο Opera κ.ά.

Ωστόσο, είναι σαφές πως το περιβάλλον εκτέλεσης της Node.js είναι τελείως διαφορετικό από τον φυλλομετρητή, όπου συνήθως χρησιμοποιούμε την JavaScript για να αλληλεπιδράσουμε με το DOM και για να χρησιμοποιήσουμε διάφορες ενσωματωμένες λειτουργίες όπως το Cookies API κ.λπ. Το DOM και τα διάφορα API προφανώς δεν είναι διαθέσιμα όταν εκτελείται η JavaScript έξω από τον φυλλομετρητή. Επιπλέον, δεν έχουμε στη διάθεσή μας ούτε τα αντίστοιχα αντικείμενα του DOM, όπως το document και το window.

Από την άλλη πλευρά, όπως θα δούμε (Ενότητα 11.3.1), η Node.js περιλαμβάνει μια σειρά από βιβλιοθήκες με τις οποίες μπορούμε να κάνουμε πράγματα που δεν είναι δυνατό να γίνουν στον φυλλομετρητή, όπως να επεξεργαστούμε αρχεία στον δίσκο (βιβλιοθήκη fs) ή να δημιουργήσουμε έναν εξυπηρετητή (βιβλιοθήκη http).

Επιπλέον, δεν γνωρίζουμε από πριν ποιον φυλλομετρητή θα χρησιμοποιήσει ο επισκέπτης της ιστοσελίδας μας, ούτε σε ποια έκδοση θα είναι αυτός. Αυτό σημαίνει πως δεν ξέρουμε ποια έκδοση ή παραλλαγή της JavaScript μπορεί να εκτελέσει ο φυλλομετρητής και ενδεχομένως χρειάζεται να πάρουμε ειδικά μέτρα για να το αντιμετωπίσουμε. Τη Node.js όμως την εγκαθιστούμε σε ένα σημείο, στον εξυπηρετητή, και έχουμε απόλυτη επίγνωση για την έκδοσή της και τις δυνατότητές της.

Ένα τέταρτο σημείο στο οποίο διαφέρουν, αν και αυτό τείνει να εξαλειφθεί, είναι το σύστημα φόρτωσης εξωτερικών βιβλιοθηκών (modules), που στον φυλλομετρητή είναι το ECMAScript 6, ενώ η Node.js υποστηρίζει δύο συστήματα, το παλιότερο CommonJS αλλά και το ES6 (Ενότητα 11.3).

Τέλος, υπάρχουν και κάποιες διαφορές στον βρόχο ελέγχου συμβάντων (Ενότητα 10.1.2).

11.2.3 Ένας απλός εξυπηρετητής σε Node.js

Αν και η Node.js έχει πολλά χρήσιμα χαρακτηριστικά που την κάνουν ένα ενδιαφέρον περιβάλλον προγραμματισμού, ο λόγος που κυρίως μας ενδιαφέρει είναι η χρήση της ως γλώσσα από την πλευρά του εξυπηρετητή. Όπως είδαμε στην ενότητα 1.3.1, η Node.js έχει έναν ενσωματωμένο εξυπηρετητή που μπορούμε να χρησιμοποιήσουμε άμεσα:

```
// Αρχείο app.js
const  http = require('http');

http.createServer(function (req, res) {
  res.write('Καλή σας μέρα!');
  res.end();
}).listen(8080); // ο εξυπηρετητής αποκρίνεται στη θύρα 8080
```

Εκτελούμε στο τερματικό ή τη γραμμή εντολών **$ node app.js** και, αν δεν υπάρξει κάποιο σφάλμα, μπορούμε να πλοηγηθούμε με τον φυλλομετρητή στη διεύθυνση http://127.0.0.1:8080, όπου θα δούμε το μήνυμα «Καλή σας μέρα!».

Στην πρώτη γραμμή του προγράμματός μας, με τη συνάρτηση **require()** φορτώνουμε τη βιβλιοθήκη (module) **http** που μας διαθέτει έναν απλό εξυπηρετητή για το πρωτόκολλο HTTP.

11.3 Βιβλιοθήκες και πακέτα στη Node.js

Έχουμε λοιπόν κατασκευάσει έναν απλό εξυπηρετητή και μπορούμε να τον χρησιμοποιήσουμε για να παρουσιάσουμε, αντί για το απλό μήνυμα «Καλή σας μέρα!», ολόκληρες σελίδες HTML, που παράγονται δυναμικά. Ένα τέτοιο εγχείρημα θα πάρει γρήγορα μεγάλο μέγεθος και θα χρειαστεί να χωρίσουμε το πρόγραμμα σε αυτοτελείς μονάδες ώστε να τις συντηρούμε πιο εύκολα. Στην JavaScript οι μονάδες αυτές ονομάζονται βιβλιοθήκες (modules) και συνήθως μια βιβλιοθήκη περιέχει κώδικα που εξυπηρετεί έναν συγκεκριμένο σκοπό. Μια βιβλιοθήκη είναι αποθηκευμένη σε ένα αρχείο.

11.3.1 Ενσωματωμένες βιβλιοθήκες

Ο μηχανισμός των modules μάς επιτρέπει να χρησιμοποιήσουμε βιβλιοθήκες με κώδικα τρίτων, επιταχύνοντας και διευκολύνοντας κατά πολύ την ανάπτυξη της εφαρμογής μας. Η Node.js έχει ενσωματωμένα έναν αριθμό από modules που μας βοηθούν να επιτελέσουμε μερικές πολύ χρήσιμες εργασίες. Για παράδειγμα, το module **http** που χρησιμοποιήσαμε στο προηγούμενο παράδειγμα είναι αυτό που μας παρέχει έναν απλό εξυπηρετητή HTTP. Μερικά ακόμη ενσωματωμένα modules είναι (η πλήρης λίστα στην τεκμηρίωση της Node.js):

- **https**: Υλοποιεί έναν εξυπηρετητή στο πρωτόκολλο HTTPS.
- **http2**: Υλοποιεί έναν εξυπηρετητή στο πρωτόκολλο HTTP2.
- **url**: Εργαλεία για το διάβασμα και τον χειρισμό URL.
- **process**: Πληροφορίες και εργαλεία για την τρέχουσα διεργασία της Node.js.
- **fs**: Δυνατότητες πρόσβασης σε αρχεία.
- **zlib**: Εργαλεία για τη συμπίεση και αποσυμπίεση αρχείων.
- **path**: Εργαλεία για τον χειρισμό διαδρομών και ονομάτων αρχείων στον δίσκο.
- **crypt**: Παρέχει κρυπτογραφικές λειτουργίες.
- **module**: Εργαλεία για την αλληλεπίδραση με τις βιβλιοθήκες.

Οι ενσωματωμένες βιβλιοθήκες δεν χρειάζονται εγκατάσταση, αλλά είναι διαθέσιμες απευθείας.

Για να δούμε τις διαθέσιμες ενσωματωμένες βιβλιοθήκες, μπορούμε να χρησιμοποιήσουμε τη βιβλιοθήκη module:

```
// Αρχείο modules.mjs
import * as module from 'module';

console.log(module.builtinModules)
```

Το αποτέλεσμα θα είναι η πλήρης λίστα των ενσωματωμένων βιβλιοθηκών.

11.3.2 Τοπικές βιβλιοθήκες

Τα ενσωματωμένα modules είναι πολύ χρήσιμα, αλλά πολλές φορές θα χρειαστεί να γράψουμε δικά μας, είτε για να τα χρησιμοποιήσουμε στην εφαρμογή μας είτε για να τα διαθέσουμε σε τρίτους.

11.3.3 Βιβλιοθήκες ECMAScript6 και CommonJS

Η Node.js υποστηρίζει δύο συστήματα για module που δεν είναι τελείως συμβατά μεταξύ τους. Ο λόγος είναι πως όταν κατασκευάστηκε η Node.js, η JavaScript δεν είχε ενσωματωμένο μηχανισμό στη γλώσσα που να υποστηρίζει module. Η Node.js χρησιμοποίησε λοιπόν τον μηχανισμό που βασίζεται στο πρότυπο CommonJS. Αυτός ο μηχανισμός είναι που φορτώνει ένα module με τη συνάρτηση `require()`.

Το 2015, όμως, με την έκδοση ECMAScript 6 (ES6), η JavaScript απέκτησε ενσωματωμένο σύστημα module. Αυτός είναι ο μηχανισμός που ονομάζεται συνήθως ES6 modules και χρησιμοποιεί την οδηγία import για το φόρτωμα. Ο μηχανισμός είναι ο ίδιος που μπορούμε να χρησιμοποιήσουμε για να φορτώσουμε module στον φυλλομετρητή. Επιπλέον, με τη σύνταξη import της ES6 μπορούμε να φορτώσουμε βιβλιοθήκες που είναι γραμμένες σύμφωνα με το CommonJS. Αντίθετα, με τη σύνταξη `require()` του CommonJS δεν μπορούμε να φορτώσουμε βιβλιοθήκες γραμμένες σύμφωνα με το ES6.

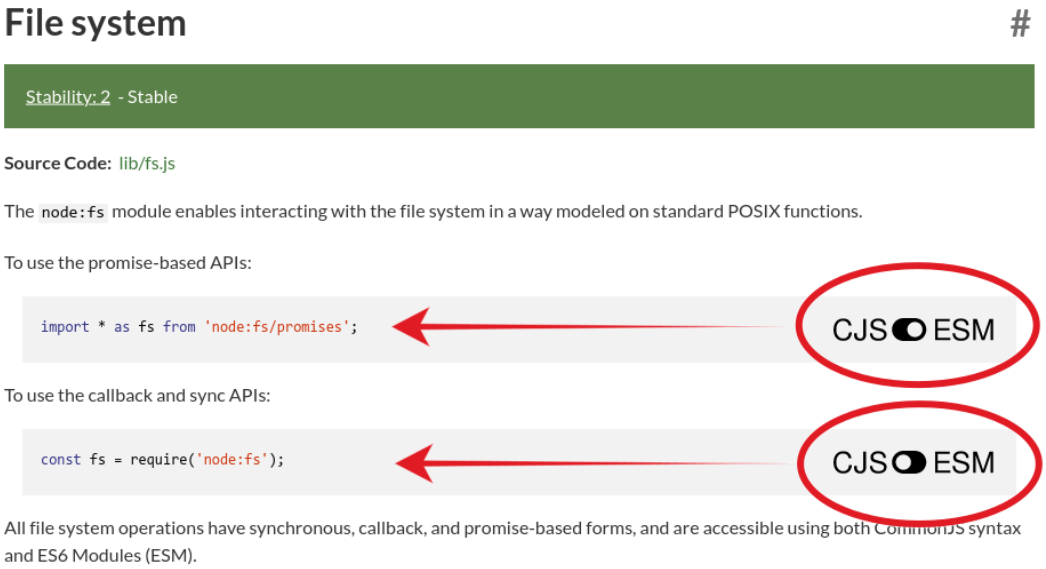
Η Node.js χρησιμοποιεί σαν προκαθορισμένο σύστημα το CommonJS. Αν θέλουμε να χρησιμοποιήσουμε το σύστημα ES6, τότε θα πρέπει να αποθηκεύσουμε το αρχείο μας με κατάληξη **.mjs**, για παράδειγμα **index.mjs**:

```
// αρχείο με κατάληξη .mjs
import http from 'http';

http.createServer(function (req, res) {
  res.write('Καλή σας μέρα!');
  res.end();
}).listen(8080); // ο εξυπηρετητής αποκρίνεται στη θύρα 8080
```

Εναλλακτικά, μπορούμε να ορίσουμε πως θέλουμε στο πρότζεκτ μας να χρησιμοποιήσουμε το πρότυπο ES6 δηλώνοντάς το στο αρχείο package.json (Ενότητα 11.3.4).

Στην τεκμηρίωση που παρέχει η Node.js για τις ενσωματωμένες βιβλιοθήκες της μπορούμε να δούμε παραδείγματα και για τους δύο τρόπους (**Εικόνα *11.3***):



Εικόνα 11.3 *Στιγμιότυπο από την τεκμηρίωση των ενσωματωμένων βιβλιοθηκών της Node.js. Με τον διακόπτη μπορούμε να επιλέξουμε τον κώδικα για την εισαγωγή του module στο πρόγραμμά μας.*

Ο μηχανισμός ES6 παρουσιάζεται πιο αναλυτικά στη συνέχεια (Ενότητα 11.7). Είναι όμως αρκετά απλός ώστε να συνεχίσουμε με τη βασική χρήση της Node.js χωρίς να μπούμε ακόμα σε λεπτομέρειες.

Μια απλή τοπική βιβλιοθήκη

Ο παρακάτω κώδικας περιγράφει ένα ES6 module που περιέχει μια λίστα για ψώνια και τη συνάρτηση isInList() που μας απαντάει αν κάποιο προϊόν είναι ή όχι στη λίστα μας:

```
// Αρχείο groceryList.mjs
const items = ['μακαρόνια', 'ελαιόλαδο', 'σκόρδο']

class GroceryList {
    isInList(item) {
        return items.includes(item)
    }
};

export { GroceryList }
```

Για να χρησιμοποιήσουμε αυτό το module θα πρέπει να το κάνουμε import, για παράδειγμα:

```
// Αρχείο app.mjs
import {GroceryList}  from './groceryList.mjs';

const gList = new GroceryList();
console.log(gList.isInList("σκόρδο"));
```

Αν εκτελέσουμε το πρόγραμμά μας με **node app.mjs**, θα δούμε το αποτέλεσμα true στο τερματικό μας.

11.3.4 Πακέτα και το εργαλείο npm

Το τρίτο και ίσως πιο χρήσιμο είδος βιβλιοθήκης στη Node.js είναι τα πακέτα. Τα πακέτα δεν είναι τίποτε άλλο από μια ομάδα αρχείων σε έναν φάκελο. Στον αρχικό φάκελο κάθε πακέτου υπάρχει ένα αρχείο με όνομα **package.json** που περιέχει βασικές πληροφορίες και οδηγίες που αφορούν το πακέτο αυτό.

Ένα πακέτο που το φορτώνουμε με την import ή με τη require είναι και module. Είναι σημαντικό να τονίσουμε πως από εδώ και στο εξής κάθε πρόγραμμά μας σε Node.js θα είναι και ένα πακέτο.

Δημιουργία πακέτου

Ο πιο απλός τρόπος δημιουργίας αυτού του αρχείου είναι να χρησιμοποιήσουμε το εργαλείο **npm** (node package manager) που εγκαθίσταται μαζί με τη Node.js (Ενότητα 11.2). Στο τερματικό, στον φάκελο που θέλουμε να δημιουργήσουμε το πακέτο μας, γράφουμε:

```
npm init -y
```

Με τις παραμέτρους init και -y θα δημιουργηθεί ένα βασικό αρχείο package.json και θα συμπληρωθεί με κάποιες πληροφορίες.

Το αρχείο αυτό θα περιέχει τα εξής:

```
{
  "name": "node",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Το πιο σημαντικό πεδίο είναι το main, που προσδιορίζει ποιο θα είναι το αρχείο που περιέχει το σημείο εισόδου του πακέτου μας, τι θα φορτωθεί όταν κάποιος φορτώσει το πρόγραμμά μας σαν πακέτο στο δικό του πρόγραμμα. Στην πορεία, το package.json θα εμπλουτιστεί και με άλλα πεδία και τιμές που θα μας βοηθήσουν στην ανάπτυξη.

Ένα τέτοιο πεδίο είναι το **type**. Με το type μπορούμε να προσδιορίσουμε ποιο σύστημα module προτιμούμε. Αν δεν γράψουμε τίποτα, όπως στο παράδειγμα πιο πάνω, τότε το σύστημά μας θα είναι το CommonJS και θα πρέπει να έχουμε στα αρχεία μας κατάληξη .mjs αν θέλουμε να χρησιμοποιήσουμε βιβλιοθήκες τύπου ES6. Μπορούμε όμως να δώσουμε την τιμή module:

```
{
  ...
  "type": "module",
  ...
}
```

Με την τιμή “module” η Node.js θα χρησιμοποιήσει το σύστημα ES6 για να φορτώσει τα modules και μπορούμε

να τα ονομάσουμε και με τη συνήθη κατάληξη .js.

Αν δεν υπάρχει η ιδιότητα type ή αν έχει τιμή "commonjs", τότε θα θεωρηθεί πως ακολουθούμε το CommonJS.

11.3.5 Εγκατάσταση πακέτων από το npmjs.com

Ένα από τα πιο ελκυστικά χαρακτηριστικά της Node.js είναι η μεγάλη δημόσια βιβλιοθήκη πακέτων, που είναι διαθέσιμη από την ιστοσελίδα https://www.npmjs.com. Ενδεικτικά, στο αποθετήριο αυτό υπάρχουν περίπου 2.000.000 πακέτα (Μάιος 2022), από τα περίπου 1.600.000 που υπήρχαν τον Μάιο του 2021 και προστίθενται πάνω από 1.000 νέα πακέτα κάθε μέρα.

Αφού εντοπίσουμε ένα πακέτο που μας ενδιαφέρει να χρησιμοποιήσουμε στο δικό μας πακέτο, χρησιμοποιούμε το npm για να το εγκαταστήσουμε, για παράδειγμα η παρακάτω εντολή στο τερματικό θα εγκαταστήσει στον τρέχοντα φάκελο το πακέτο yodasay:

```
npm install yodasay
```

Το πακέτο θα αποθηκευτεί στον υποφάκελο node_modules στον τρέχοντα φάκελο. Καθώς το yodasay έχει εξαρτήσεις και σε άλλα πακέτα, το npm αναλαμβάνει να αποθηκεύσει και αυτά. Στο τέλος, στον υποφάκελο node_modules θα βρίσκεται το yodasay καθώς και όλα τα πακέτα που χρειάζεται για να λειτουργήσει, συνολικά 6 πακέτα.

Επιπλέον, στο αρχείο package.json θα προστεθεί η ιδιότητα dependencies που γράφει πως το δικό μας πακέτο εξαρτάται από το yodasay, στην έκδοση τουλάχιστον 1.1.9:

```
{
  ...
  "dependencies": {
    "yodasay": "^1.1.9"
  },
  ...
}
```

Από εδώ προκύπτει μια πολύ χρήσιμη λειτουργία, δηλαδή να μπορούμε να κατεβάσουμε όλες τις εξαρτήσεις μας (τα πακέτα που χρειάζεται το δικό μας πρόγραμμα) ακόμη και αν δεν υπάρχει ο φάκελος node_modules. Με την εντολή

```
npm install
```

θα εγκατασταθούν όλες οι εξαρτήσεις που περιγράφονται στο package.json.

Το yodasay είναι ένα πακέτο που ζωγραφίζει στο τερματικό τον Yoda, χαρακτήρα των ταινιών Star Wars, που λέει ό,τι του ζητήσουμε, για παράδειγμα:

```
//αρχείο με όνομα my_yoda.mjs
import yodasay from 'yodasay';

console.log(yodasay.say({
  text: 'Use yodasay, you will.'
}));
```

Αν εκτελέσουμε στο τερματικό **node my_yoda.mjs** θα εμφανίσει το εξής:

```
 **
< Use yodasay, you will. >
 ------------------------
      \
       \
          .--.
  \`--._,'.::.`._.--'/
    .   ` **::** '   .
      -:.`'..`'.:-
```

```
\ `--' /
  ----
```

Άσκηση

Δημιουργήστε ένα νέο πακέτο και χρησιμοποιήστε το πακέτο cowsay για να γράψετε μια μικρή εφαρμογή. Χρησιμοποιήστε το cowsay για να εμφανίσετε στην οθόνη τη λίστα με τα ψώνια των προηγούμενων παραδειγμάτων.

Απεγκατάσταση πακέτων

Μπορούμε να απεγκαταστήσουμε ένα πακέτο απλά σβήνοντάς το από τον υποφάκελο node_modules. Άλλος ένας τρόπος, που σβήνει και όλες τις εξαρτήσεις του πακέτου που απομακρύνουμε, είναι, π.χ., να τρέξουμε npm uninstall --no-save yodasay.

Και στις δύο περιπτώσεις στο package.json θα συνεχίσει να υπάρχει αναφορά στο πακέτο. Στην επόμενη εκτέλεση της npm install θα γίνει πάλι εγκατάσταση.

Για να απομακρύνουμε τελείως ένα πακέτο και οτιδήποτε άλλο εγκαταστάθηκε από το npm για λογαριασμό του, απλά εκτελούμε:

```
npm uninstall package
```

11.3.6 Αρίθμηση εκδόσεων κατά SemVer

Θα παρατηρήσουμε πως η αρίθμηση των εκδόσεων αποτελείται από τρεις ακέραιους αριθμούς και αυτό ισχύει τόσο για τα πακέτα που κατεβάζουμε με το npm όσο και για την έκδοση που έδωσε το npm στο δικό μας πακέτο όταν τρέξαμε την εντολή npm init -y.

Εικόνα 11.4 *Η αρίθμηση κατά SemVer χρησιμοποιεί ακεραίους σε τρεις θέσεις για να σηματοδοτήσει το είδος των αλλαγών μεταξύ εκδόσεων.*

Ο λόγος είναι πως η Node.js ακολουθεί τη σύμβαση Semantic Versioning ή SemVer (**Εικόνα *11.4***). Στο SemVer οι τρεις ακέραιοι σηματοδοτούν αλλαγές ανάλογα με το εύρος τους:

- **Μεγάλη** (Major) αλλαγή: Ο πρώτος ακέραιος αυξάνει κατά ένα όταν η νέα μας έκδοση εισάγει μια αλλαγή που κάνει το πακέτο μας μη συμβατό με προηγούμενες εκδόσεις.
- **Μικρή** (Minor) αλλαγή: Ο δεύτερος ακέραιος αυξάνει κατά ένα όταν η νέα μας έκδοση εισάγει μια νέα λειτουργία, αλλά το πακέτο μας είναι συμβατό με προηγούμενες εκδόσεις, δηλαδή κάποιος που χρησιμοποιούσε την προηγούμενη έκδοση, μπορεί να χρησιμοποιήσει και τη νέα χωρίς να χρειαστεί να κάνει αλλαγές στον δικό του κώδικα.
- **Διόρθωση** (Patch): Ο τρίτος ακέραιος αυξάνει όταν η νέα έκδοση του πακέτου μας δεν εισάγει αλλαγή στη λειτουργικότητα, αλλά διορθώνει ένα σφάλμα της προηγούμενης έκδοσης.

Το SemVer είναι μια αρκετά δημοφιλής και απλή σύμβαση και η υιοθέτησή της έχει κάποια πλεονεκτήματα. Μπορούμε να καταλάβουμε με μια ματιά τι είδους αλλαγή συνέβη από τη μια έκδοση στην άλλη και να προσδιορίσουμε ποιες εκδόσεις πακέτων αποδεχόμαστε.

Αν δούμε ξανά την εγγραφή στο package.json για την εξάρτηση στο yodasay, θα παρατηρήσουμε πως γράφει

```
"yodasay": "^1.1.9"
```

Το σύμβολο ^ απλά λέει πως αποδεχόμαστε οποιαδήποτε έκδοση του yodasay από την 1.1.9 και πάνω, αρκεί να μην αυξηθεί ο πρώτος ακέραιος, άρα δεν επιτρέπουμε εκδόσεις με μεγάλες αλλαγές.

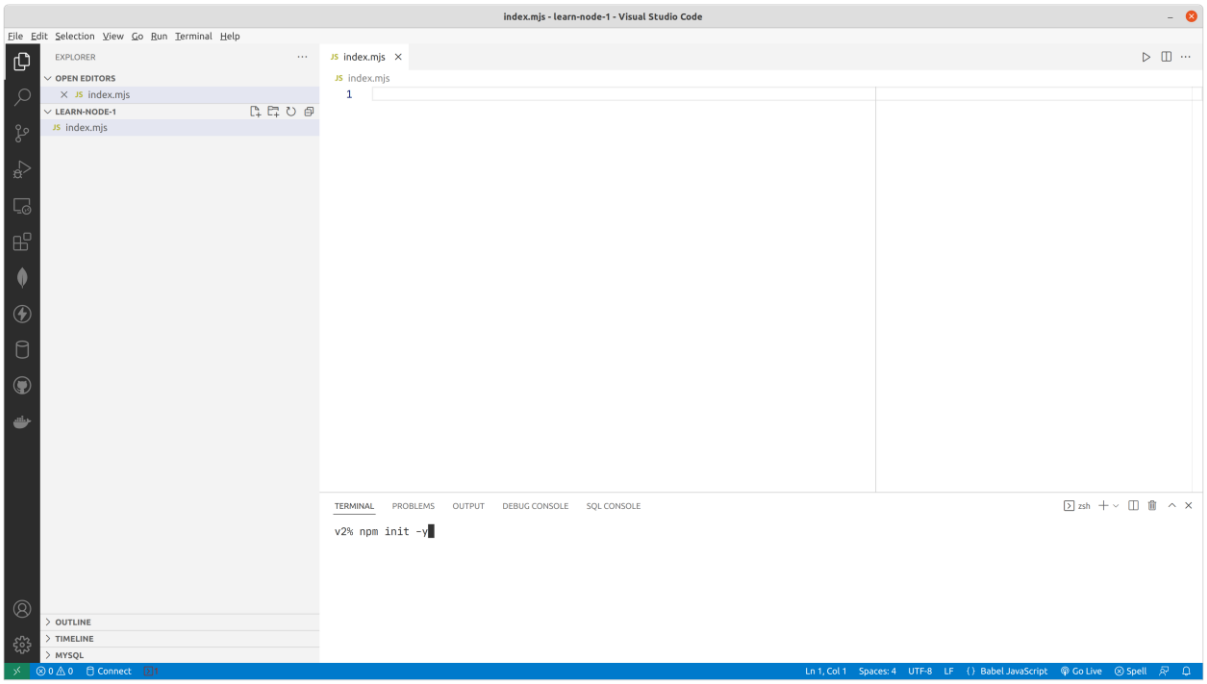
Το εργαλείο npm semver calculator μάς βοηθά αν θέλουμε να διατυπώσουμε πιο λεπτομερείς απαιτήσεις όσον αφορά την έκδοση των πακέτων που αποδεχόμαστε.

11.4 Βασικές εργασίες με αρχεία

Πριν χρησιμοποιήσουμε τη Node.js για τη δημιουργία εφαρμογών διαδικτύου, είναι σκόπιμο να εξοικειωθούμε μαζί της με πιο απλά παραδείγματα, χρησιμοποιώντας την ενσωματωμένη βιβλιοθήκη **fs**.

Η βιβλιοθήκη **fs** μας επιτρέπει να κάνουμε βασικές εργασίες με αρχεία, να ανοίξουμε και να διαβάσουμε ένα αρχείο, να γράψουμε νέο περιεχόμενο κ.λπ.

Αν και δεν είναι αυστηρά απαραίτητο, ωστόσο ακόμη και για απλά παραδείγματα ενδείκνυται να χρησιμοποιήσουμε το npm για να δημιουργήσουμε ένα πακέτο. Σε έναν νέο φάκελο λοιπόν, που μπορούμε να ονομάσουμε learn-node-1, ανοίγουμε το Visual Studio Code. Από το VS Code επιλέγουμε File->Open Folder και ανοίγουμε τον φάκελο που θα περιέχει το πακέτο μας. Στη συνέχεια, στο τερματικό του VS Code πληκτρολογούμε **npm init -y** (**Εικόνα 11.5**) και δημιουργείται το αρχείο package.json.



Εικόνα 11.5 *Το τερματικό είναι προσβάσιμο από το μενού Terminal->New Terminal.*

11.4.1 Εγγραφή σε αρχείο

Η βιβλιοθήκη fs παρέχει πολλές χρήσιμες λειτουργίες για τον χειρισμό αρχείων. Μπορούμε να την ενσωματώσουμε είτε με το σύστημα CommonJS

```
const fs = require('fs')
```

είτε με το σύστημα ES6 modules, που θα προτιμήσουμε σε αυτό το βιβλίο. Η βιβλιοθήκη fs μπορεί να λειτουργήσει με τρεις τρόπους:

- **Σύγχρονα**, όπου η κάθε εντολή εκτελείται μόνο αφού τελειώσει η προηγούμενη. Σε αυτή την περίπτωση, αν για παράδειγμα έχουμε να ανοίξουμε ένα πολύ μεγάλο αρχείο θα έχουμε καθυστερήσεις.

Όλες οι υπόλοιπες εντολές, που δεν εξαρτώνται από το αποτέλεσμα του ανοίγματος του αρχείου, θα περιμένουν να τελειώσει αυτό για να εκτελεστούν.

- **Ασύγχρονα**, όπου μπορούμε να εκτελέσουμε στο παρασκήνιο την ενέργεια που μπορεί να μπλοκάρει την εκτέλεση του προγράμματός μας, ώστε να εκτελεστεί *ασύγχρονα*. Η βιβλιοθήκη fs μπορεί να λειτουργήσει και με τους δύο μηχανισμούς που είδαμε στο προηγούμενο κεφάλαιο πως παρέχει η JavaScript για την ασύγχρονη εκτέλεση κώδικα:
  - τη **συνάρτηση επιστροφής** και
  - τις **υποσχέσεις**.

Σύγχρονα

Με την πιο απλή σε χρήση, σύγχρονη εκδοχή, όταν η Node.js ζητήσει πρόσβαση στο αρχείο από το λειτουργικό σύστημα, θα περιμένει μέχρι να τερματίσει αυτή η ενέργεια. Μόνο όταν τελειώσει η πρόσβαση και η Node.js πάρει το αποτέλεσμα από το λειτουργικό σύστημα θα συνεχίσει η εκτέλεση των επόμενων εντολών.

```
//αρχείο write_sync.mjs
import *  as fs from 'fs'

console.log("Αρχή")
try {
    const fd = fs.openSync('shopping-list.txt', 'a+')
    fs.writeSync(fd, "Μπανάνες\n");
    fs.closeSync(fd);
}
catch (err) {
    if (err.code == 'EACCES')
        console.error("Δεν έχω πρόσβαση")
    else
        throw err
}
//Ανάγνωση των περιεχομένων του αρχείου
try {
    const data = fs.readFileSync('shopping-list.txt', 'utf-8');
    console.log('Δεδομένα: ', data.trim());
}
catch(err) {
    throw err
}
console.log("Τέλος")
```

Στο παράδειγμα αυτό χρησιμοποιείται η readFileSync(), που διαβάζει το περιεχόμενο ολόκληρου του αρχείου σύγχρονα. Αν το αρχείο είναι μεγάλο, μπορεί, εκτός από το μπλοκάρισμα του event loop, να προκύψει και πρόβλημα με τη διαθέσιμη μνήμη, καθώς η συνάρτηση αποπειράται να φορτώσει όλα τα περιεχόμενα του αρχείου στη μνήμη.

Με συναρτήσεις επιστροφής

Με τη χρήση συναρτήσεων επιστροφής μπορούμε να εκτελέσουμε τις ενέργειές μας ασύγχρονα, χωρίς να μπλοκάρουμε το event loop. Όταν ολοκληρωθούν, θα κληθεί η συνάρτηση επιστροφής. Το ίδιο προηγούμενο παράδειγμα μπορούμε να το γράψουμε με συναρτήσεις επιστροφής.

```
//αρχείο write_cb.mjs
import *  as fs from 'fs'

console.log("Αρχή")
fs.open('shopping-list.txt', 'a+', (err, fd) => {
    if (err)
        throw err
    else {
```

```
            fs.write(fd, "Μπανάνες\n", (err, written, string) => {
                if (err)
                    throw err
                else
                    fs.readFile('shopping-list.txt', 'utf-8', (err, data) => {
                        if (err)
                            throw err
                        else
                            console.log('Δεδομένα: ', data.trim());
                        });
            })
        }
    })
    console.log("Τέλος")
```

Αυτό το μικρό πρόγραμμα χρησιμοποιεί συναρτήσεις επιστροφής που είδαμε σε προηγούμενη ενότητα (Ενότητα 10.1.2). Η συνάρτηση fs.open(), λοιπόν, καλείται με τρία ορίσματα, **όνομα αρχείου**, **flag** και μια **συνάρτηση επιστροφής**. Με το δεύτερο όρισμα, το **flag**, ορίζουμε τι μπορεί να γίνει με το αρχείο, για παράδειγμα:

- **a+**: Ανοίγει το αρχείο για συμπλήρωση και ανάγνωση. Το + σημαίνει πως αν δεν υπάρχει το αρχείο θα δημιουργηθεί.

Το fs δίνει πολλές άλλες επιλογές για το flag, για παράδειγμα:

- **r**: Ανοίγει για ανάγνωση, αν όμως δεν υπάρχει θα προκύψει εξαίρεση.
- **w+**: Ανοίγει για ανάγνωση και εγγραφή. Το αρχείο, αν δεν υπάρχει, θα δημιουργηθεί. Αν υπάρχει, τότε θα σβηστούν τα περιεχόμενά του.

Η τρίτη παράμετρος, η συνάρτηση επιστροφής, θα κληθεί από τη fs.open(). Η `open()` θα περάσει δύο παραμέτρους στη συνάρτηση επιστροφής. Η πρώτη θα έχει κάποια τιμή αν υπήρξε σφάλμα κατά το άνοιγμα, η δεύτερη είναι ένας **δείκτης προς το αρχείο** που άνοιξε η `open()`, που συνήθως ονομάζεται file descriptor.

Στο σώμα της συνάρτησης επιστροφής ελέγχουμε αν έχει προκύψει κάποιο σφάλμα και, στο παράδειγμα, το χειριζόμαστε υποτυπωδώς, καθώς απλά ρίχνουμε ξανά το σφάλμα με τη throw. Για παράδειγμα, θα μπορούσαμε να μην έχουμε δικαιώματα εγγραφής στον δίσκο ή στο συγκεκριμένο αρχείο.

Αν όλα πήγαν καλά και το αντικείμενο err που πέρασε η `open()` στη συνάρτηση επιστροφής έχει τιμή null, θα συνεχίσει η εκτέλεσή της. Σε αυτή την περίπτωση θα κληθεί η συνάρτηση fs.write().

Η `write()` καλείται και αυτή με τρία ορίσματα, τον **δείκτη στο αρχείο**, το **αλφαριθμητικό** που θέλουμε να γράψουμε και μια **συνάρτηση επιστροφής**. Στο απλό μας παράδειγμα χρησιμοποιούμε τη συνάρτηση επιστροφής για την περίπτωση που υπάρξει κάποιο σφάλμα κατά την εγγραφή, κατ' αναλογία με αυτό που κάναμε στη συνάρτηση επιστροφής της open(). Οι δύο επιπλέον παράμετροι που ορίζουμε στη συνάρτηση επιστροφής (**written** και **string**) περιέχουν το πλήθος των bytes που χρειάστηκαν για το αλφαριθμητικό που γράφτηκε.

Με Promises

Η τρίτη εκδοχή του παραδείγματός μας χρησιμοποιεί το Promise API, που είδαμε πιο αναλυτικά στην προηγούμενη ενότητα, όπως φαίνεται από τη γραμμή του import. Στη συνέχεια, καλούμε την writeFile(), που αυτή τη φορά επιστρέφει ένα promise, και χρησιμοποιούμε μια αλυσίδα καταναλωτών (Ενότητα 10.4.2). Στο επόμενο βήμα διαβάζουμε από το αρχείο με τη readFile() που και αυτή επιστρέφει ένα promise και, όταν επιλυθεί και αυτή, τυπώνουμε το αποτέλεσμα, που περιέχεται στη μεταβλητή data.

```
//αρχείο write_promise.mjs
import *  as fs from 'fs/promises'

console.log("Αρχή")
fs.writeFile('shopping-list.txt', "Αχλάδια\n", { flag: 'a+' })
    .then(() => fs.readFile('shopping-list.txt', 'utf-8'))
    .then((data) => {
        console.log("Δεδομένα: ", data)
    })
```

```
    .catch(err => {
        console.error("Παρουσιάστηκε σφάλμα: ", err)
    });

console.log("Τέλος")
```

Με async/await

Η εκδοχή με τις συναρτήσεις επιστροφής οδηγεί σε κώδικα που μπορεί να είναι δυσνόητος και, μάλιστα, όπως είδαμε, αυτή η κατάσταση είναι τόσο συχνή που έχει όνομα, η «κόλαση των callback». Η εισαγωγή του Promise API είναι μια μεγάλη βελτίωση στη συγγραφή ασύγχρονου κώδικα, ωστόσο πολλοί χρήστες (προγραμματιστές) δεν τη βρίσκουν ιδιαίτερα ευχάριστη.

Υπάρχει λοιπόν μια εναλλακτική σύνταξη για το Promise API, που είναι πιο εύχρηστη, η σύνταξη "async/await". Ας δούμε πρώτα το παράδειγμα εγγραφής και ανάγνωσης σε αρχείο, πριν την εξηγήσουμε.

```
//αρχείο write_async_await.mjs
import *  as fs from 'fs/promises`

try {
    await fs.writeFile('./shopping-list.txt', 'Μήλα\n', { flag: 'a+' })
    const data = await fs.readFile('shopping-list.txt', 'utf-8')
    console.log("Δεδομένα: ", data)
}
catch (err) {
    throw err
}

console.log("Τέλος")
```

Είναι αμέσως εμφανές πως ο κώδικάς μας είναι πιο ευανάγνωστος με τη σύνταξη async/await. Με τη λέξη-κλειδί **await** μπροστά από μια εντολή δηλώνουμε πως αυτή θα εκτελεστεί ασύγχρονα. Αυτό σημαίνει πως η readFile() θα εκτελεστεί μόνο αφού τελειώσει η writeFile(). Ωστόσο, σε αντίθεση με τη σύγχρονη εκτέλεση που είδαμε στο πρώτο παράδειγμα, το event loop δεν θα μπλοκαριστεί για όσο χρόνο χρειάζεται να εκτελεστεί η writeFile(). Η μεταβλητή **data** θα πάρει τιμή μόνο όταν ολοκληρωθεί (fulfill, στην ορολογία του Promise API) η ασύγχρονη readFile(). Όλο το μπλοκ κώδικα είναι μέσα σε μια δομή **try/catch** (Ενότητα 8.6.4), που κάνει και τη διαχείριση σφαλμάτων αρκετά πιο εύκολη.

11.5 Async/await

Η σύνταξη async/await είναι ένας εναλλακτικός τρόπος χρήσης του Promise API και κάνει τη συγγραφή ασύγχρονου κώδικα αισθητά πιο απλή. Ο βασικός μηχανισμός όμως συνεχίζει να είναι αυτός που παρέχει το Promise API (Ενότητα 10.4).

11.5.1 Η λέξη-κλειδί async

Η λέξη-κλειδί **async** μπροστά από μια συνάρτηση δηλώνει πως αυτή επιστρέφει ένα promise. Για να δούμε τη χρήση της θα ξαναγράψουμε το παράδειγμα χρήσης καταναλωτών από την ενότητα 10.4.2:

```
const p = new Promise((resolve, reject) => {
    //συναρτήσεις που ενεργοποιούνται σε περίπτωση επιτυχούς ή όχι
ολοκλήρωσης
    if (Math.random() > 0.5) { //προσομοίωση τυχαιότητας λειτουργίας
        resolve('Επιτυχής ολοκλήρωση')
    }
    else {
        reject('Σφάλμα')
    }
})
```

```
async function f() {
    return p
}

f().then(result => console.log(result),
    error => console.error(error))
```

Η συνάρτησή μας f() έχει σημειωθεί με τη λέξη-κλειδί **async**, καθώς επιστρέφει ένα promise.

Θα μπορούσαμε ωστόσο να μην επιστρέψουμε ένα promise, αλλά κατευθείαν μια τιμή:

```
async function f() {
    if (Math.random() > 0.5) { //προσομοίωση τυχαιότητας λειτουργίας
        return ('Επιτυχής ολοκλήρωση')
    }
    else {
        throw Error("Σφάλμα")
    }
}

f().then(result => console.log(result))
     .catch(error => console.error(error.message))
```

Σε αυτή την περίπτωση, λοιπόν, η συνάρτησή μας δεν επιστρέφει ρητά ένα promise. Η async, παρ' όλ' αυτά, αναλαμβάνει να τοποθετήσει το αποτέλεσμα σε ένα promise και μας επιστρέφει αυτό.

11.5.2 Η λέξη-κλειδί await

Η async, ωστόσο, χρησιμοποιείται σε συνδυασμό με τη λέξη-κλειδί **await**. Με την **await** η JavaScript περιμένει να ολοκληρωθεί η υπόσχεση ώστε να συνεχίσει την εκτέλεση. Πλέον, το παράδειγμά μας μπορεί να γραφτεί ως εξής:

```
async function f() {
    if (Math.random() > 0.5) { //προσομοίωση τυχαιότητας λειτουργίας
        return ('Επιτυχής ολοκλήρωση')
    }
    else {
        throw Error("Σφάλμα")
    }
}

try {
    // καλείται η f() και περιμένουμε μέχρι να επιλυθεί η υπόσχεση:
    const result = await f()
    console.log(result)
}
catch(error) {
    console.log(error.message)
}
```

Όταν εκτελείται η γραμμή **const** result = **await** f(), η JavaScript δεν εκτελεί την επόμενη γραμμή, αλλά περιμένει μέχρι να επιλυθεί η υπόσχεση που επιστρέφει η f(). Η αναμονή αυτή όμως δεν μπλοκάρει τον βρόχο ελέγχου συμβάντων, το event loop.

Η σύνταξη αυτή είναι σαφώς πιο εύκολη στην ανάγνωση από τη σύνταξη του Promise API, όπως γίνεται εμφανές αν γράψουμε ένα κάπως πιο σύνθετο παράδειγμα, όπως αυτό της ενότητας 10.4.3, χρησιμοποιώντας async/await:

```
console.log('αρχή προγράμματος');

// Καθώς η setTimeout δεν είναι συνάρτηση async (δεν επιστρέφει promise),
```

```
// για να πάρουμε το αποτέλεσμα μετά από 1000 ms είτε θα καλέσουμε μια
συνάρτηση
// επιστροφής είτε θα χρησιμοποιήσουμε promise:
function sleep() {
  return new Promise((resolve) => {
    setTimeout(() => resolve(10), 1000)
  })
}

// Τώρα μπορούμε να περιμένουμε μέχρι να ολοκληρωθεί η υπόσχεση
let result = await sleep();

// Υλοποιούμε τρεις φορές τη συνάρτηση με ελαφρώς διαφορετική σύνταξη
// Πρακτικά, οι τρεις συναρτήσεις κάνουν το ίδιο: διπλασιάζουν την τιμή
// εισόδου
async function function1(aValue) {
  console.log('async-1: ', aValue);
  return aValue * 2;
}

const function2 = async aValue => {
  console.log('async-2: ', aValue);
  return aValue * 2;
}

const function3 = async function (aValue) {
  console.log('async-3: ', aValue);
  return aValue * 2;
}

let res1 = await function1(result);
let res2 = await function2(res1);
await function3(res2);
console.log('τέλος προγράμματος');
```

Όπως βλέπουμε, η εκτέλεση των ασύγχρονων συναρτήσεων είναι εύκολα κατανοητή. Οι τρεις ασύγχρονες συναρτήσεις κάνουν τον ίδιο υπολογισμό, διπλασιάζουν την είσοδό τους, με ελαφρώς διαφορετική σύνταξη. Το αποτέλεσμα από την εκτέλεση του προγράμματος θα είναι:

```
αρχή προγράμματος
await-1:  10
await-2:  20
await-3:  40
τέλος προγράμματος
```

Η χρήση της await στον κώδικα

Σε παλιότερες εκδόσεις της Node.js αλλά και σε παλιότερους φυλλομετρητές η χρήση της await δεν επιτρεπόταν αν δεν ήταν μέσα σε μια συνάρτηση async. Για παράδειγμα, ο παρακάτω κώδικας δεν μπορούσε να εκτελεστεί:

```
let aValue = await someAsyncFunction();
let bValue = await anotherAsyncFunction(aValue);
```

Αυτός ο περιορισμός ξεπερνιόταν χρησιμοποιώντας μια σύνταξη όπου η συνάρτηση εκτελείται την ώρα που ορίζεται. Η σύνταξη αυτή ονομάζεται Immediately Invoked Function Expression ή άμεσα καλούμενη συναρτησιακή έκφραση:

```
(async () => {
    let aValue = await someAsyncFunction();
```

```
    let bValue = await anotherAsyncFunction(aValue);
})();
```

Στη Node.js επιτρέπεται πλέον η χρήση της await στο ανώτερο επίπεδο, έξω από συνάρτηση async, ακριβώς όπως στα προηγούμενα παραδείγματα. Για να γίνει αυτό, όμως, θα πρέπει ο κώδικάς μας να είναι σε μορφή ECMAScript6, δηλαδή

- να είναι αποθηκευμένος σε αρχείο με κατάληξη **.mjs** ή
- να έχει οριστεί ο τύπος του πακέτου μας στο αρχείο package.json σε module, όπως περιγράφεται στην ενότητα 11.3.4.

Top-level await στον φυλλομετρητή

Αντίστοιχα, στον φυλλομετρητή η χρήση των await στο ανώτερο επίπεδο επιτρέπεται αν φορτώσουμε τον κώδικά μας σαν ECMAScript6 module, χρησιμοποιώντας το γνώρισμα **type="module**, όπως στο παρακάτω παράδειγμα που τυπώνει τον πληθυσμό της Ελλάδας στην κονσόλα του φυλλομετρητή.

```
<script type="module">
    let response = await
fetch("https://restcountries.com/v3.1/name/greece")
    let responseJson = await response.json()

    console.log(responseJson[0].population)
</script>
```

11.5.3 Χειρισμός σφαλμάτων

Συνοψίζοντας, η χρήση της λέξης **async** μπροστά από μια συνάρτηση έχει σαν αποτέλεσμα η συνάρτηση να επιστρέφει πάντα μια υπόσχεση. Με τη λέξη **await** μπροστά από μια υπόσχεση, η JavaScript θα περιμένει μέχρι να υλοποιηθεί αυτή.

Αν κατά την υλοποίηση της υπόσχεσης συμβεί κάποιο σφάλμα, μπορούμε να το χειριστούμε με τον συνήθη τρόπο του Promise API. Όπως είδαμε, το async/await είναι απλά μια εναλλακτική σύνταξη για να γράφουμε ασύγχρονο κώδικα με το Promise API. Συνεπώς, ο χειρισμός σφαλμάτων μπορεί να γίνει με τον ειδικό καταναλωτή .catch(), όπως ακριβώς και στις υποσχέσεις.

Είναι όμως συνήθως πιο βολικό να χειριστούμε το σφάλμα χρησιμοποιώντας τη δομή **try/catch** (+@#sec:try_catch). Για παράδειγμα, έστω μια συνάρτηση async που υπό προϋποθέσεις μπορεί να εγείρει ένα νέο σφάλμα:

```
async function myDivision(a, b) {
  if (b==0)
      throw new Error("Δεν μπορεί να γίνει διαίρεση με το 0")
  return a/b
}
```

Ο χειρισμός μπορεί να γίνει με την try/catch:

```
try {
  console.log(await myDivision(2, 1))
}
catch(error) {
  console.log("Συνέβη σφάλμα", error.message)
}
```

ή με τον καταναλωτή catch():

```
myDivision(2, 0)
    .then( result => console.log(result))
    .catch( error => console.log(error))
```

11.6 Η Node.js ως εξυπηρετητής διαδικτύου

Μέχρι τώρα έχουμε συζητήσει μερικά βασικά χαρακτηριστικά του περιβάλλοντος που παρέχει η Node.js. Έχουμε δει τα πακέτα και τις βιβλιοθήκες, τον ασύγχρονο προγραμματισμό με το παράδειγμα της βασικής βιβλιοθήκης **fs** και χρησιμοποιήσαμε την ευκαιρία για να συμπληρώσουμε την εικόνα του Promise API με τη βολικότερη σύνταξη async/await.

Στη συνέχεια θα εστιάσουμε στη χρήση της Node.js ως περιβάλλον ανάπτυξης εφαρμογών διαδικτύου από την πλευρά του εξυπηρετητή.

Στην αρχή του κεφαλαίου είδαμε ήδη έναν απλό εξυπηρετητή Node.js, που μπορούμε να ξεκινήσουμε με node <όνομα αρχείου.mjs> από το τερματικό:

```
// αρχείο με κατάληξη .mjs
import http from 'http';

http.createServer(function (req, res) {
  res.write('Καλή σας μέρα!');
  res.end();
}).listen(8080); // ο εξυπηρετητής αποκρίνεται στη θύρα 8080
```

Κάθε φορά που θα φτάνει ένα αίτημα στον εξυπηρετητή μας (π.χ. αν ανοίξουμε τον φυλλομετρητή και φορτώσουμε τη σελίδα http://127.0.0.1:8080) θα εκτελείται η συνάρτηση που περάσαμε σαν όρισμα στην createServer(). Η συνάρτηση αυτή είναι η συνάρτηση χειρισμού αιτημάτων (request handler). Η Node.js, όταν φτάσει ένα αίτημα στον εξυπηρετητή μας, καλεί τον χειριστή αιτημάτων με δύο ορίσματα, το αντικείμενο του αιτήματος HTTP και το αντικείμενο της απάντησης HTTP. Συνηθίζεται να ονομάζουμε τις παραμέτρους αυτές req και res ή request και response αντίστοιχα.

Κάθε αίτημα λοιπόν συνοδεύεται από το **request object**. Με βάση τις πληροφορίες που περιέχονται στο αίτημα, ο εξυπηρετητής μπορεί να αποφασίσει τι θα επιστρέψει σαν απάντηση. Όπως έχουμε δει (Ενότητα 1.4.3), ένα αίτημα χαρακτηρίζεται από τη μέθοδο, τη διαδρομή του πόρου που ζητήθηκε, την έκδοση του πρωτοκόλλου, τις κεφαλίδες και το σώμα του μηνύματος.

Μπορούμε να ανακτήσουμε όλες αυτές τις πληροφορίες μέσω του αντικείμενου αιτήματος:

```
// αρχείο server.mjs
import http from 'http';

http.createServer(function (req, res) {
    console.log("Μέθοδος: ", req.method)
    console.log("Διαδρομή: ", req.url)
    console.log("Πρωτόκολλο: ", req.httpVersion)
    console.log("Κεφαλίδες: ", req.headers)
    console.log("Σώμα: ", req.body)
    res.write('Καλή σας μέρα!');
    res.end();
}).listen(8080); // ο εξυπηρετητής αποκρίνεται στη θύρα 8080
```

Συνήθως, τα αιτήματα με τη μέθοδο **GET** συνοδεύονται και από μια λίστα παραμέτρων. Για παράδειγμα, ένα αίτημα προς τη μηχανή αναζήτησης Google έχει τη μορφή https://www.google.com/search?q=καιρός, όπου η παράμετρος q έχει την τιμή καιρός. Αν έχουμε περισσότερες παραμέτρους, αυτές χωρίζονται με τον χαρακτήρα **&**:

```
var1=value1&var2=value2&var3=value3
```

Ο εξυπηρετητής διαμορφώνει και την απάντηση που θα δώσει στο αίτημα με βάση τις παραμέτρους του αιτήματος. Η βιβλιοθήκη http δεν παρέχει βοηθήματα για την ανάγνωση των παραμέτρων, αλλά μπορούμε να καταφύγουμε στην επίσης ενσωματωμένη στη Node.js βιβλιοθήκη url. Εδώ μπορεί να δημιουργηθεί μια μικρή σύγχυση, καθώς η βιβλιοθήκη url περιλαμβάνει το αντικείμενο URL, το οποίο μας ενδιαφέρει. Το αντικείμενο αυτό είναι διαθέσιμο στο global object, οπότε δεν χρειάζεται να το φορτώσουμε με import.

Αρχικά, κατασκευάζουμε ένα νέο στιγμιότυπο του URL με βάση τη διαδρομή του αιτήματός μας, που αποτελείται από το πρωτόκολλο του αιτήματος, που χάριν απλότητας το θέτουμε σε http, το όνομα του

εξυπηρετητή μας (στον οποίο απευθύνθηκε το αίτημα) και τη διαδρομή του αιτήματος:

```
const myURL = new URL("http://" + req.headers.host + req.url)
```

Έχοντας κατασκευάσει έτσι ένα αντικείμενο URL, μπορούμε να ανακτήσουμε τις τιμές των παραμέτρων που μας ενδιαφέρουν με την κλάση URL.searchParams και τη συνάρτησή της get():

```
const name = myURL.searchParams.get('onoma')
```

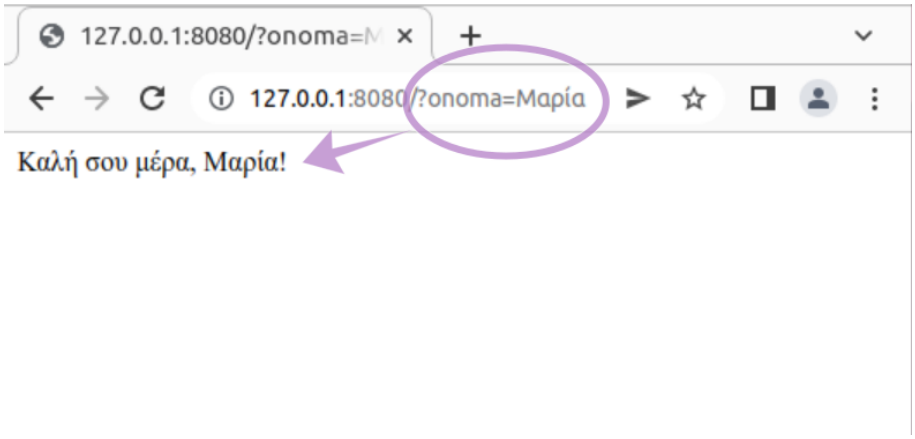
Έτσι, αν το αίτημα ήταν στο URL http://127.0.0.1:8080/?onoma=Μαρία, η μεταβλητή name θα έχει την τιμή «Μαρία». Πριν στείλουμε την απάντησή μας στον φυλλομετρητή, λόγω των ελληνικών, θα πρέπει να ειδοποιήσουμε τον φυλλομετρητή πως η απάντηση που στέλνουμε είναι κωδικοποιημένη σε UTF-8, ορίζοντας την κατάλληλη κεφαλίδα στην απάντηση που θα στείλουμε.

```
// αρχείο server.mjs
import http from 'http';

http.createServer(function (req, res) {
    const myURL = new URL("http://" + req.headers.host + req.url)
    const name = myURL.searchParams.get('onoma')

    res.setHeader('Content-Type',  'text/html; charset=utf-8')
    res.write(`Καλή σου μέρα, ${name}!`);
    res.end();
}).listen(8080); // ο εξυπηρετητής αποκρίνεται στη θύρα 8080
```

Η πολύ απλοϊκή εφαρμογή μας θα εμφανίσει στον φυλλομετρητή ένα μήνυμα καλωσορίσματος (**Εικόνα *11.6***)



Εικόνα 11.6 *Η εφαρμογή παράγει διαφορετική απόκριση με βάση την τιμή της παραμέτρου onoma.*

11.7 ES6 export και import

Όπως είδαμε πριν (Ενότητα 11.3.3), η Node.js υποστηρίζει το πιο πρόσφατο πρότυπο βιβλιοθηκών ECMAScript6, γνωστό και ως ES6. Με αυτό το πρότυπο μπορούμε να μοιράσουμε τον κώδικά μας σε λογικές ενότητες, πρακτικά σε διαφορετικά αρχεία, χρησιμοποιώντας τον ίδιο μηχανισμό που έχουμε και στην πλευρά του φυλλομετρητή. Έχουμε ήδη δει σε προηγούμενα παραδείγματα πώς μπορούμε να φορτώσουμε βιβλιοθήκες με τον μηχανισμό ES6.

Σε αυτή την ενότητα θα δούμε το πρότυπο με μεγαλύτερη λεπτομέρεια έτσι ώστε να μπορούμε να γράφουμε δικές μας βιβλιοθήκες και να φορτώνουμε βιβλιοθήκες τρίτων με μεγαλύτερη ευελιξία. Η χρήση των βιβλιοθηκών ES6 είναι απλή, αλλά ο μηχανισμός παρέχει κάμποσους εναλλακτικούς τρόπους για να ορίσουμε και να φορτώσουμε βιβλιοθήκες, που καλό είναι να τους έχουμε υπόψη.

11.7.1 Ονοματισμένα export

Για να ορίσουμε δικές μας βιβλιοθήκες χρησιμοποιούμε τη δήλωση export, μπροστά από τα στοιχεία που θέλουμε να κάνουμε διαθέσιμα για φόρτωμα. Αυτά τα στοιχεία μπορεί να είναι μεταβλητές (**var**, **let** ή **const**), συναρτήσεις ή κλάσεις:

```
//αρχείο my-module.mjs
export let myVariable = "Καλημέρα"

export function myFunction() {
    console.log("Δουλεύω...")
}

export class MyClass {
    constructor() {
        console.log("Κατασκευάζω...")
    }
}
```

Εναλλακτικά, και μερικές φορές ίσως είναι πιο βολικό, μπορούμε να έχουμε μια μοναδική δήλωση export που να περιέχει σε άγκιστρα τα στοιχεία που θέλουμε να διαθέσουμε:

```
export {myVariable, myFunction, myClass}
```

Έχουμε λοιπόν **ονοματίσει** ποια στοιχεία θέλουμε να κάνουμε διαθέσιμα για φόρτωμα σε άλλα αρχεία. Για να τα φορτώσουμε χρησιμοποιούμε τη δήλωση import, ονοματίζοντας και πάλι τα στοιχεία που χρειαζόμαστε για να φορτώσουμε:

```
//αρχείο use-my-module.mjs
//το my-module-mjs βρίσκεται στον ίδιο φάκελο ("./")
import {myVariable, myFunction, MyClass}  from './my-module.mjs';

console.log(myVariable)  //τυπώνει "Καλημέρα"
myFunction() //τυπώνει "Δουλεύω..."
const myObj = new MyClass()
```

11.7.2 Μετονομασία των στοιχείων

Συχνά θα χρειαστεί να φορτώσουμε βιβλιοθήκες που παρέχουν τρίτοι, όπου ενδεχομένως χρησιμοποιούνται ονόματα για τα στοιχεία που εξάγουν ίδια με τα δικά μας ονόματα. Ο μηχανισμός ES6 μάς δίνει τη δυνατότητα μετονομασίας των στοιχείων που φορτώνουμε με τη λέξη-κλειδί as:

```
import {myVariable as myOtherVariable, myFunction as anotherFunction}
from './my-module.mjs';

myVariable="Καληνύχτα"
console.log(myOtherVariable) //τυπώνει "Καλημέρα"
anotherFunction()  //τυπώνει "Δουλεύω..."
```

Η ίδια λέξη-κλειδί as μπορεί να χρησιμοποιηθεί και στην πλευρά του export:

```
export {myVariable as myOtherVariable, myFunction, myClass}
```

11.7.3 Default exports

Ο λόγος όμως που χρησιμοποιούμε κάποιο σύστημα βιβλιοθηκών εξαρχής είναι πως επιδιώκουμε τα προγράμματά μας να είναι αρθρωτά. Να αποτελούνται δηλαδή από καλά ορισμένα επιμέρους τμήματα, καθένα από τα οποία να επιτελεί μια εργασία. Αν ένα τμήμα ή βιβλιοθήκη επιτελεί περισσότερες εργασίες, ίσως είναι καλή ιδέα να τη διασπάσουμε σε επιμέρους αρχεία.

Στο ES6 μπορούμε όμως να ακολουθήσουμε έναν ενδιάμεσο δρόμο και να ορίσουμε *ένα* στοιχείο που

να φορτώνεται χωρίς να ονοματιστεί, το λεγόμενο **default export**. Θα μπορούμε να έχουμε πολλά export, αλλά μόνο ένα θα είναι το **default**:

```
//αρχείο my-module.mjs
export let myVariable = "Καλημέρα"

export default function myDefaultFunction() {
    console.log("Δουλεύω πάντα...")
}

export class MyClass { ... }
```

ή εναλλακτικά:

```
export {myVariable, myFunction as default, myClass}
```

Πλέον, πέρα από τα *ονοματισμένα* import, μπορούμε να φορτώσουμε και το default ή μόνο αυτό:

```
import myDefaultFunction from './my-module.mjs'

...

myDefaultFunction() //τυπώνει "Δουλεύω πάντα..."
```

ή

```
import { default as anotherFunction } from './my-module.mjs'

...

anotherFunction() //τυπώνει "Δουλεύω πάντα..."
```

Τέλος, μπορούμε να συνδυάσουμε default και ονοματισμένα import, αρκεί το default να είναι πρώτο:

```
import myDefaultFunction, {myVariable, myFunction, MyClass}  from './my-
module.mjs';
```

11.7.4 Εισαγωγή της βιβλιοθήκης ως αντικειμένου

Ο πιο πρακτικός τρόπος για να φορτώσουμε τη βιβλιοθήκη είναι να τη φορτώσουμε ως αντικείμενο και είναι πολύ απλός:

```
// Τα στοιχεία που κάνει export το my-module θα είναι διαθέσιμα
// στο αντικείμενο με όνομα MyModule
import * as MyModule from './my-module.mjs'

console.log(MyModule.myOtherVariable) //τυπώνει "Καλημέρα"
MyModule.myFunction()  //τυπώνει "Δουλεύω..."
```

Με τον τρόπο αυτό, έχουμε δημιουργήσει ένα νέο αντικείμενο, με όνομα MyModule, μέλη του οποίου είναι όλα τα export του my-module.mjs. Και πάλι, μπορούμε να συνδυάσουμε την εισαγωγή ως αντικείμενο με την εισαγωγή default, αρκεί το default να είναι πρώτο:

```
import myDefaultFunction, * as MyModule from './my-module.mjs';
```

11.7.5 Φόρτωση και εκτέλεση βιβλιοθήκης

Πιο σπάνια, μπορεί να χρειαστεί να φορτώσουμε μια βιβλιοθήκη όχι για να εισαγάγουμε κάποια στοιχεία από αυτή, αλλά για να εκτελεστεί κάποιος κώδικας που περιέχει. Ένα τέτοιο παράδειγμα υπάρχει πιο κάτω στο κεφάλαιο, στη χρήση του πακέτου dotenv (11.8.4). Σε αυτή την περίπτωση δεν χρειάζεται να προσδιορίσουμε ποια στοιχεία θέλουμε να εισαγάγουμε, απλά φορτώνουμε τη βιβλιοθήκη. Έστω μια απλή βιβλιοθήκη:

```
//αρχείο my-module.mjs
export let myVariable = "Καλημέρα"

export function myFunction() {
    console.log("Δουλεύω...")
}

export class MyClass {
    constructor() {
        console.log("Κατασκευάζω...")
    }
}
console.log("Εκτέλεση κώδικα στο my-module.mjs")
```

Φορτώνοντας λοιπόν τη βιβλιοθήκη μας

```
import './my-module.mjs'
//Θα εμφανιστεί το μήνυμα "Εκτέλεση κώδικα στο my-module.mjs"
//τα exports δεν θα είναι διαθέσιμα.
```

θα εκτελεστεί ο κώδικας που αυτή περιέχει, αλλά τα export δεν θα μας είναι διαθέσιμα.

11.7.6 Δυναμικά import

Από την έκδοση EcmaScript 2020 του προτύπου μπορούμε να χρησιμοποιήσουμε την έκφραση import(), με σύνταξη που μοιάζει με συνάρτηση, για να φορτώσουμε βιβλιοθήκες. Έτσι, μπορούμε να χρησιμοποιήσουμε την import() μέσα σε μια συνάρτηση ή σαν μέρος ενός λογικού ελέγχου συνθήκης με **if** κ.ο.κ.

Με την έκφραση import() επιστρέφεται ένα Promise, το οποίο ολοκληρώνεται (fullfil) όταν φορτωθεί η βιβλιοθήκη. Μπορούμε έτσι, για παράδειγμα, να γράψουμε προγράμματα που φορτώνουν βιβλιοθήκες ανάλογα με την τιμή κάποιας παραμέτρου από το περιβάλλον:

```
const PLATFORM = process.env.PLATFORM // μπορεί να είναι Windows, Mac,
Linux κ.λπ.

if (PLATFORM=="Windows") {
    await import('./module_with_windows_functionality.mjs')
    doStuff()
    ...
} else {
    ...
}
```

11.7.7 Ιδιότητες και περιορισμοί

Συνοπτικά, ο μηχανισμός των βιβλιοθηκών ECMAScript είναι κατανοητός και απλός. Πέρα από τα παραπάνω παραδείγματα, μερικά άλλα σημαντικά σημεία είναι πως, με εξαίρεση τα δυναμικά import –με import()–, όλα τα export και import πρέπει να είναι στο ανώτερο επίπεδο στον κώδικά μας, π.χ. δεν μπορεί να γίνει export σε μια συνάρτηση που ανήκει σε μια κλάση, ούτε να χρησιμοποιηθεί η import μέσα σε μια συνάρτηση – η import() όμως ναι.

Επιπλέον, δεν έχει σημασία σε ποια γραμμή γίνεται το import, θα γίνει πάντα hoist (Ενότητα 7.3.4), δηλαδή θα μετακινηθεί αυτόματα πάνω και η βιβλιοθήκη που κάνουμε import θα είναι διαθέσιμη από την πρώτη γραμμή του κώδικά μας.

Τέλος, ένα σημαντικό χαρακτηριστικό του μηχανισμού είναι πως τα στοιχεία που κάνουμε import είναι μόνο για ανάγνωση και δεν μπορούμε να μεταβάλουμε τις τιμές τους.

Άσκηση
Γράψτε μια εφαρμογή διαδικτύου όπου η λίστα με τα ψώνια να εμφανίζεται, αντί για την κονσόλα, σαν απάντηση σε ένα αίτημα HTTP.

11.8 Βοηθητικά εργαλεία για την ανάπτυξη εφαρμογών Node.js

Στο κλείσιμο αυτού του κεφαλαίου θα αναφερθούμε σε βοηθητικά εργαλεία που είναι χρήσιμα για την ανάπτυξη εφαρμογών Node.js, τα οποία ξεπερνούν τα όρια του τετριμμένου παραδείγματος. Όσο αυξάνεται η πολυπλοκότητα της εφαρμογής μας χρειάζεται να χρησιμοποιήσουμε τεχνικές και εργαλεία που κάνουν τη ζωή μας ως προγραμματιστές πιο εύκολη. Επίσης, θα περιγράψουμε πώς μπορούμε με απλά βήματα να δημοσιεύσουμε την εφαρμογή μας ώστε να είναι διαθέσιμη στο κοινό, μέσα από την υπηρεσία Heroku.

11.8.1 Dependencies και DevDependencies

Όπως είδαμε στην ενότητα 11.3.5, το εργαλείο npm μας βοηθά να προσδιορίσουμε και να εγκαταστήσουμε τις εξαρτήσεις της εφαρμογής μας. Μερικές από τις εξαρτήσεις, όμως, δεν χρειάζονται για τη λειτουργία της εφαρμογής, αλλά είναι πολύ χρήσιμες για τον προγραμματιστή κατά την ανάπτυξή της. Τέτοιο παράδειγμα είναι το πακέτο nodemon που παρουσιάζεται αμέσως μετά.

Το npm μας δίνει τη δυνατότητα να έχουμε εξαρτήσεις δύο ειδών, τις απλές, που λειτουργούν όπως ήδη γνωρίζουμε, και τις εξαρτήσεις ανάπτυξης (DevDependencies). Οι εξαρτήσεις ανάπτυξης βρίσκονται και αυτές στο package.json, αλλά στο αντικείμενο "devDependencies".

Οι εξαρτήσεις ανάπτυξης εγκαθίστανται με την εντολή

```
npm install <package-name> --save-dev
```

Με την εντολή npm install θα εγκατασταθούν όλες οι εξαρτήσεις, τόσο οι απλές όσο και οι ανάπτυξης, π.χ. αν διαγράψουμε τον φάκελο node_modules/ και θέλουμε να τον ανασυνθέσουμε.

Ωστόσο, και εδώ είναι η κύρια διαφορά, μπορούμε να ζητήσουμε να εγκατασταθεί το πακέτο μας όχι για ανάπτυξη, αλλά για παραγωγική χρήση. Τότε, με την εντολή

```
npm install --production
```

θα εγκατασταθούν μόνο οι απλές εξαρτήσεις και όχι οι εξαρτήσεις ανάπτυξης.

11.8.2 Το πακέτο nodemon

Στο βασικό παράδειγμα που είδαμε, κάθε φορά που κάνουμε μια αλλαγή στον εξυπηρετητή μας θα πρέπει να τον σταματήσουμε (πατώντας Ctrl-C ή Cmd-C στο παράθυρο του τερματικού) και να ξεκινήσουμε από την αρχή τον εξυπηρετητή όταν κάνουμε τροποποιήσεις στον κώδικα.

Το πακέτο nodemon επανεκκινεί την εφαρμογή μας αυτόματα όταν αποθηκεύσουμε το αρχείο με τον κώδικά μας. Είναι απαραίτητα δύο βήματα, πρώτα να εγκαταστήσουμε το nodemon και έπειτα να ορίσουμε ένα σκριπτ με το οποίο θα ξεκινάει το nodemon. Τα δύο πρώτα βήματα χρειάζεται να τα κάνουμε μόνο μια φορά:

1. Εγκαθιστούμε το πακέτο τοπικά εκτελώντας στο τερματικό (έχουμε αρχικοποιήσει βέβαια το πακέτο μας με npm init -y):

   ```
   npm install nodemon --save-dev
   ```

2. Στο αρχείο package.json του πακέτου μας προσθέτουμε στην ιδιότητα scripts το όνομα του σκριπτ, π.χ. “start”, και την εντολή που θέλουμε να εκτελείται, στην προκειμένη περίπτωση nodemon <όνομα αρχείου>:

   ```
   "scripts": {
       "test": "echo \"Error: no test specified\" && exit 1",
       "start": "nodemon server.mjs"
   },
   ```

3. Κάθε φορά που θέλουμε να δουλέψουμε με το πρότζεκτ μας αρκεί να εκτελέσουμε το σκριπτ μας από το τερματικό εκτελώντας npm run <όνομα σκριπτ>:

   ```
   npm run start
   ```

Για την ακρίβεια, η επανεκκίνηση με το nodemon γίνεται σε κάθε αλλαγή σε αρχεία στον φάκελό μας που έχουν κατάληξη js, mjs, και json. Αυτή η συμπεριφορά μπορεί να αλλάξει ρυθμίζοντας κατάλληλα το nodemon,

π.χ. γράφοντας στο package.json:

```
...
"nodemonConfig": {
  "ignore": ["*.json"]
},
...
```

11.8.3 Ορισμός και χρήση μεταβλητών περιβάλλοντος

Πολύ συχνά είναι σκόπιμο να μη γράφουμε κάποιες τιμές στον κώδικά μας, αλλά να τις παίρνουμε από το περιβάλλον της εφαρμογής μας. Οι μεταβλητές περιβάλλοντος είναι μεταβλητές που δεν ανήκουν στην εφαρμογή μας, αλλά υπάρχουν στο λειτουργικό σύστημα στο οποίο αυτή εκτελείται. Συνήθως αυτές οι μεταβλητές γράφονται με κεφαλαία γράμματα. Η χρήση των μεταβλητών περιβάλλοντος μας επιτρέπει να διαχωρίσουμε την παραμετροποίηση της εφαρμογής μας από την ίδια την εφαρμογή, να αποθηκεύσουμε δηλαδή τις τιμές της παραμετροποίησης ξεχωριστά από την ίδια την εφαρμογή.

Μια τέτοια περίπτωση είναι ο αριθμός της θύρας (port) στην οποία θα δέχεται μηνύματα ο εξυπηρετητής μας. Στα προηγούμενα παραδείγματα την έχουμε ορίσει με 8080. Συχνά όμως αυτή θα είναι μια παράμετρος η οποία εξαρτάται από το λειτουργικό σύστημα στο οποίο εκτελείται η εφαρμογή. Θα μπορούσε, για παράδειγμα, να υπάρχει ήδη μια εφαρμογή που να περιμένει μηνύματα στη θύρα 8080 και, συνεπώς, να πρέπει να αφήσουμε το σύστημα να προσδιορίσει τη θύρα. Αυτό μπορεί να γίνει μέσω μιας μεταβλητής περιβάλλοντος.

Οι μεταβλητές περιβάλλοντος είναι διαθέσιμες στη Node.js μέσα από την ενσωματωμένη βιβλιοθήκη process, που έχει πληροφορίες για τη διεργασία που εκτελεί τη Node.js στον υπολογιστή.

Συνηθίζεται, για παράδειγμα, να χρησιμοποιούμε τη θύρα που ορίζεται στη μεταβλητή περιβάλλοντος **PORT** σαν θύρα στην οποία ο εξυπηρετητής μας περιμένει για συνδέσεις. Η τιμή αυτής της μεταβλητής θα είναι διαθέσιμη στην εφαρμογή μας μέσω της ιδιότητας process.env.PORT. Για να εξασφαλίσουμε πως θα έχουμε πάντα μια τιμή για τη θύρα μας, ανεξάρτητα από το αν υπάρχει ή όχι η μεταβλητή περιβάλλοντος, μπορούμε να γράψουμε:

```
const serverListeningPort = process.env.PORT || 8080

http.createServer(function (req, res) {
    ...
}.listen(serverListeningPort)
```

11.8.4 Μεταβλητές περιβάλλοντος μέσω dotenv

Το πακέτο dotenv επιτρέπει να έχουμε «μεταβλητές περιβάλλοντος» σε εξωτερικά αρχεία. Με άλλα λόγια, μας επιτρέπει να ορίσουμε εμείς κάποιες μεταβλητές, οι οποίες θα είναι διαθέσιμες στην εφαρμογή μας σαν να ήταν μεταβλητές περιβάλλοντος.

1. Εγκαθιστούμε το πακέτο τοπικά εκτελώντας στο τερματικό (έχουμε αρχικοποιήσει βέβαια το πακέτο μας με npm init -y):

   ```
   npm install dotenv
   ```

2. Σε αρχείο με όνομα **.env** (dot-env) γράφουμε τα ζεύγη μεταβλητών-τιμών που θέλουμε να είναι διαθέσιμα στο περιβάλλον της εφαρμογής μας, π.χ.:

   ```
   HOST = localhost
   PORT = 8081
   ```

3. Για να φορτώσουμε τις μεταβλητές στην εφαρμογή μας αρκεί να γράψουμε:

   ```
   import 'dotenv/config'
   //Πλέον οι τιμές που έχουν οριστεί στο .env είναι διαθέσιμες:
   console.log(HOST)
   console.log(PORT)
   ```

Στην περίπτωση που η μεταβλητή περιβάλλοντος που έχουμε ορίσει στο αρχείο .env υπάρχει ήδη στο

περιβάλλον του λειτουργικού συστήματος, θα χρησιμοποιηθεί αυτή του συστήματος.

11.8.5 Φιλοξενία της εφαρμογής στο Heroku

Μια εφαρμογή διαδικτύου δεν έχει πολύ νόημα αν δεν είναι προσβάσιμη στο διαδίκτυο. Για όσο διαρκεί η ανάπτυξη της εφαρμογής μας μπορεί να επαρκεί να χρησιμοποιούμε τον υπολογιστή για την ανάπτυξη και τη φιλοξενία της. Όταν θελήσουμε όμως να τη δείξουμε σε άλλους ή να τη διαθέσουμε στο ευρύτερο κοινό, αυτό δεν θα είναι εύκολα δυνατό από τον υπολογιστή μας. Μπορούμε να χρησιμοποιήσουμε διάφορες υπηρεσίες, από τις οποίες αρκετά δημοφιλής είναι αυτή που παρέχει το Heroku.

Προαπαιτούμενα

Για να χρησιμοποιήσουμε την υπηρεσία του Heroku χρειάζεται να ικανοποιήσουμε μερικά προαπαιτούμενα. Συγκεκριμένα, χρειάζονται

- Ένας λογαριασμός (δωρεάν) στο Heroku.
- Να εγκαταστήσουμε τα εργαλεία Git (όχι Github).
- Να εγκαταστήσουμε το εργαλείο Heroku CLI (command line interface).

Θα πρέπει να σημειωθεί ότι η πλατφόρμα αυτή κατήργησε τη δωρεάν χρήση προς όλους, αφήνοντας όμως τη δυνατότητα δωρεάν χρήσης της υπηρεσίας για μη κερδοσκοπικούς ή εκπαιδευτικούς σκοπούς. Συνεπώς, θα πρέπει να κάνουμε αίτηση για δωρεάν υπηρεσία ως μη κερδοσκοπική ή εκπαιδευτική δραστηριότητα ώστε να μπορέσουμε να τη χρησιμοποιήσουμε.

Με τη χρήση του Heroku, αυτόματα θα έχουμε δύο σημεία στα οποία θα υπάρχει η εφαρμογή μας: Ο υπολογιστής στον οποίο γίνεται η ανάπτυξή της, η διόρθωση σφαλμάτων, η προσθήκη λειτουργιών κ.λπ., και ο χώρος μας στο Heroku, όπου θα ανεβάζουμε ή θα «σπρώχνουμε» (push) τις αλλαγές στη δημόσια έκδοση της εφαρμογής μας. Η διαδικασία αυτή γίνεται στο Heroku μέσα από το δημοφιλές εργαλείο τήρησης εκδόσεων git. Αν και το git δείχνει τις δυνατότητές του σε εφαρμογές που αναπτύσσονται από πολλούς προγραμματιστές ταυτόχρονα, ωστόσο, ακόμα και στην περίπτωση που ο προγραμματιστής είναι ένας είναι χρήσιμο εργαλείο για τη διατήρηση του ιστορικού ανάπτυξης της εφαρμογής, καθώς διατηρεί το σωρευτικό ιστορικό των αλλαγών που πραγματοποιούμε στον κώδικα.

Προετοιμασία της εφαρμογής μας

Πριν τη φιλοξενία της εφαρμογής μας από το Heroku χρειάζεται να έχουμε αρχικοποιήσει το πακέτο μας εκτελώντας npm init -y και να δηλώσουμε στο package.json ποια έκδοση της Node.js θέλουμε να χρησιμοποιήσει το Heroku όταν εκτελεί την εφαρμογή μας:

- Με την εντολή node -v στο τερματικό βρίσκουμε την έκδοση που έχουμε και
- δηλώνουμε στο packages.json την έκδοσή μας:

```
"engines": {
    "node": "16.x"
},
```

Χρειάζεται επίσης να προσδιορίσουμε ποιο είναι το σημείο εισόδου στην εφαρμογή μας. Για να το δηλώσουμε αυτό δημιουργούμε ένα αρχείο με όνομα Procfile.

- Στο αρχείο Procfile γράφουμε (αν, π.χ,. η εφαρμογή ξεκινάει από το αρχείο server.mjs):

  ```
  web: node server.mjs
  ```

- Πριν ανεβάσουμε την εφαρμογή, μπορούμε να δοκιμάσουμε τοπικά χρησιμοποιώντας το Heroku CLI:

  ```
  heroku local web
  ```

- Αν θέλουμε να δοκιμάσουμε σε διαφορετικό PORT, π.χ. 8888, γράφουμε στο αρχείο με όνομα .env (δείτε στην ενότητα 11.8.4 για το πακέτο dotenv).

  ```
  PORT=8888
  ```

- Αν δεν το έχουμε κάνει ήδη, αρχικοποιούμε το git στον φάκελο του πρότζεκτ μας με

  ```
  git init
  ```

- Αν όλα πήγαν καλά, τότε κάνουμε login και ζητάμε από το Heroku να μας δώσει τον χώρο που θα ανεβάσουμε το πρότζεκτ, είτε από τη σελίδα του dashboard στο Heroku είτε απευθείας από τη γραμμή εντολών

```
heroku login
...
heroku create --region=eu
```

Η τελευταία εντολή θα μας δώσει δύο πληροφορίες:
- τον σύνδεσμο στον οποίο θα φιλοξενηθεί το πρότζεκτ μας, δηλαδή το URL με το οποίο θα ανοίγουμε την εφαρμογή στον φυλλομετρητή,
- το **git remote** στο οποίο θα ανεβάζουμε (push) τις νέες μας εκδόσεις. Το git remote είναι ο χώρος στον οποίο θα φυλάσσονται τα αρχεία μας στο Heroku. Κάθε φορά που θα είμαστε έτοιμοι για να δημοσιεύσουμε μια νέα έκδοση θα την ανεβάζουμε στο git remote.

Πολλά από τα αρχεία που συνοδεύουν την εφαρμογή μας δεν χρειάζεται να δημοσιεύονται στο git remote. Ένα παράδειγμα είναι τα περιεχόμενα του υποφακέλου node_modules/ του πρότζεκτ μας. Όπως είδαμε (Ενότητα 11.3.5), μπορούμε όποτε θέλουμε να εγκαταστήσουμε τα πακέτα που περιέχονται εκεί, συνεπώς δεν χρειάζεται να τα φυλάμε στο ιστορικό του git.

- Για να το πετύχουμε αυτό δημιουργούμε ένα αρχείο με όνομα .gitignore, που κάθε γραμμή του ορίζει φακέλους ή αρχεία που θα αγνοούνται από το ιστορικό του git:

```
# Να αγνοούνται τα node modules
/node_modules
# διάφορα αρχεία
npm-debug.log
# αρχείο με πληροφορίες φακέλου στο MacOS
.DS_Store
# το αρχείο dotenv (.env)
*.env
```

Δημοσίευση στο Heroku

Τα προηγούμενα βήματα χρειάζεται να γίνουν μόνο μια φορά για κάθε πρότζεκτ μας. Στη συνέχεια, όταν θέλουμε να δημοσιεύσουμε την εφαρμογή με όλες τις αλλαγές και τροποποιήσεις, αρκεί να επαναλάβουμε τα παρακάτω βήματα.

- Ενημερώνουμε το ιστορικό του git:
  - Αν έχουμε προσθέσει νέα αρχεία, με την εντολή git add . για να επισημάνουμε τα νέα αρχεία που έχουμε δημιουργήσει. Έτσι, αυτά θα συμπεριληφθούν στην επόμενη καταχώριση στο ιστορικό του git. Θα συμπεριληφθούν όλα τα αρχεία που δεν αναγράφονται στο .gitignore,
  - git commit -m «Περιγραφή των αλλαγών» θα κάνει μια νέα καταχώριση στο ιστορικό του git. Θα καταχωριστούν οι αλλαγές στα αρχεία μας μέχρι τώρα σαν μια νέα έκδοση. Η πιο πρόσφατη έκδοση ονομάζεται πάντα "HEAD".
- Ανεβάζουμε (push) την πιο πρόσφατη έκδοσή μας στο Heroku. Με την εντολή git push heroku master ανεβαίνει πάντα η έκδοση "HEAD", δηλαδή αυτή που καταχωρίστηκε την τελευταία φορά που εκτελέσαμε git commit.

Άσκηση

Δημοσιεύστε στο Heroku τη μικρή εφαρμογή διαδικτύου που φτιάξατε στην προηγούμενη άσκηση.

11.8.6 Αποσφαλμάτωση με το Visual Studio Code

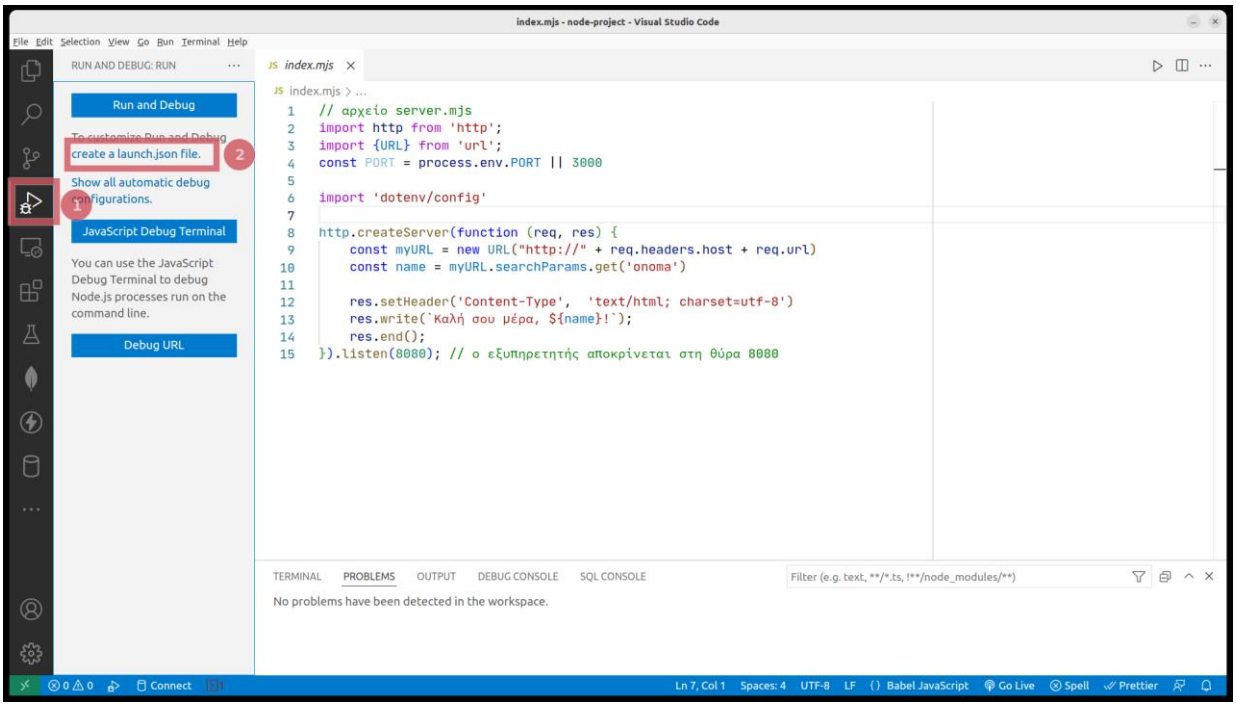
Η εύρεση και αντιμετώπιση σφαλμάτων είναι συχνά χρονοβόρα και επίπονη διαδικασία. Το εργαλείο αποσφαλμάτωσης (debugger) που είναι ενσωματωμένο στο Visual Studio Code είναι εύχρηστο και πολύ χρήσιμο για την εύρεση και την αντιμετώπιση σφαλμάτων.

Ρύθμιση του εργαλείου αποσφαλμάτωσης

Για να ενεργοποιηθεί το εργαλείο αποσφαλμάτωσης χρειάζεται να γίνουν δύο ενέργειες. Αρχικά, να προστεθεί μια ρύθμιση στο αρχείο .vscode/launch.json. Αν το αρχείο αυτό δεν υπάρχει ήδη στο πρότζεκτ μας, μπορεί να

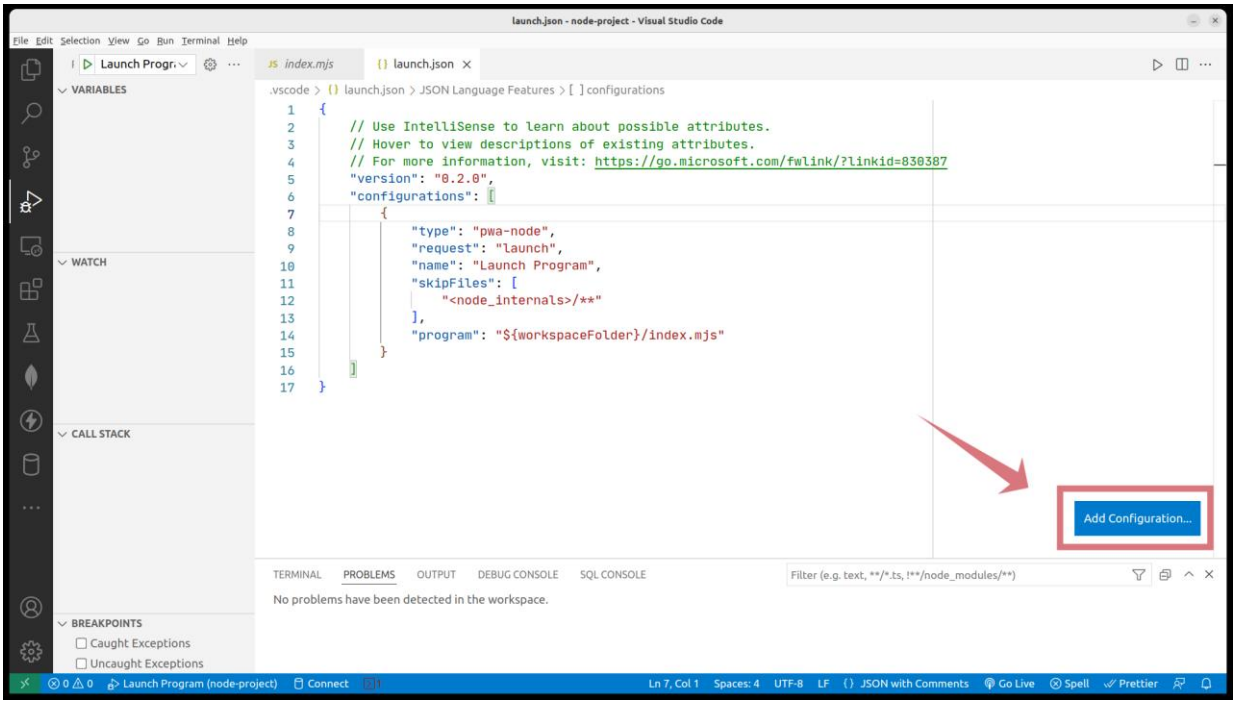
δημιουργηθεί πολύ εύκολα, για παράδειγμα:

- Στο Visual Studio Code επιλέγουμε το εργαλείο “Run and Debug” (ή πατάμε Ctrl+Shift+D (σε Mac Cmd+Shift+D) από την αριστερή στήλη εργαλείων,
- επιλέγουμε τον σύνδεσμο create a launch.json file.



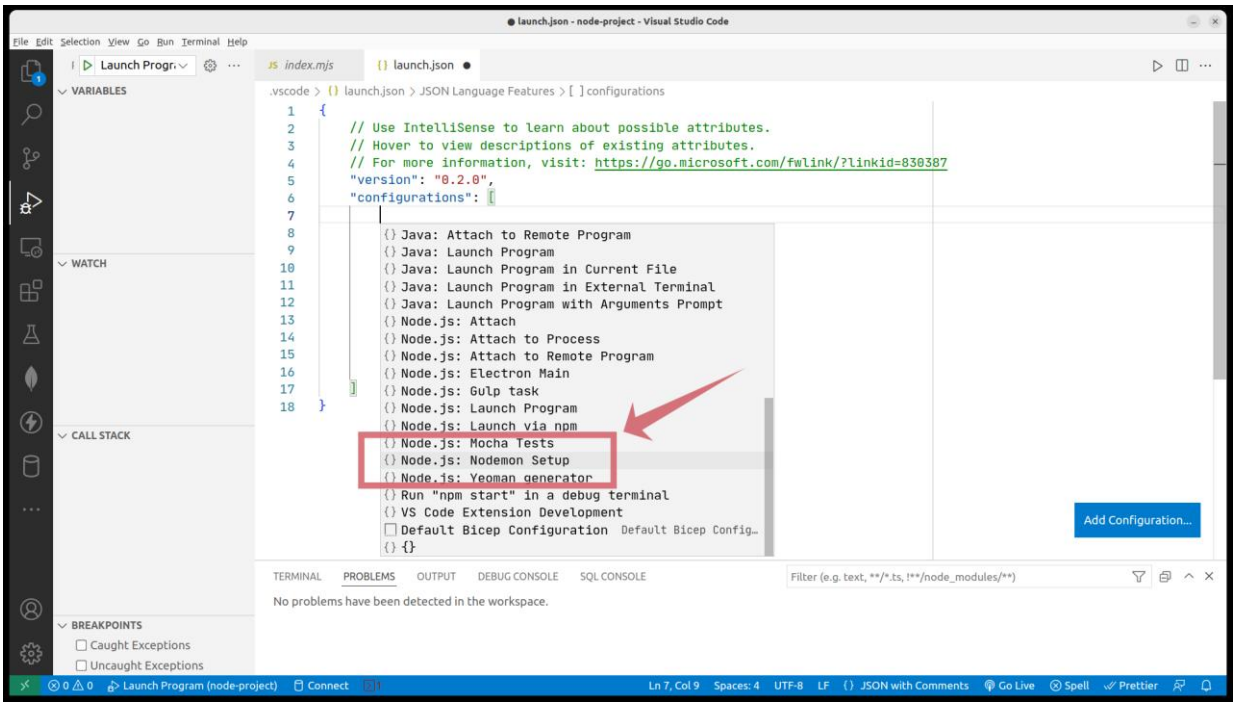
Εικόνα 11.7 *Βήματα για τη δημιουργία του αρχείου .vscode/launch.json, αν δεν υπάρχει ήδη.*

Στη συνέχεια θα εμφανιστεί το αρχείο launch.json που θα δημιουργήσει αυτόματα το Visual Studio Code. Επιλέγουμε “Add Configuration…” (**Εικόνα *11.8***).



Εικόνα 11.8 *Δημιουργούμε ένα νέο launch configuration.*

Έπειτα, επιλέγουμε ένα από τα έτοιμα launch configuration που μας δίνει το εργαλείο (**Εικόνα *11.9***). Μια καλή επιλογή είναι να διαλέξουμε το **nodemon**, όπως φαίνεται στην εικόνα. Αυτό βέβαια χρειάζεται να υπάρχει το nodemon εγκατεστημένο είτε στο σύστημά μας είτε σαν εξάρτηση στο πακέτο μας (Ενότητα 11.8.2).



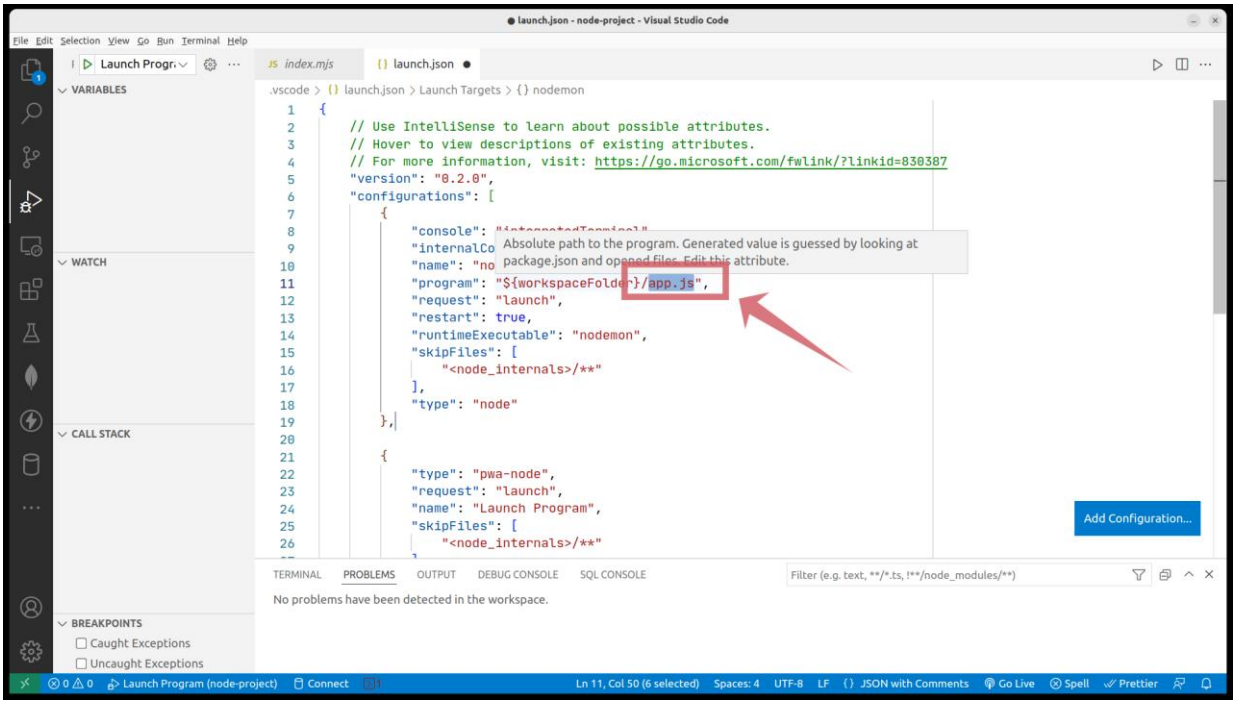
Εικόνα 11.9 *Επιλογή του προτιμώμενου launch configuration.*

Αν προσέξετε τις επιλογές που έχει κάνει το Visual Studio Code, θα δείτε ότι θεωρεί πως το αρχείο που τρέχει την εφαρμογή μας είναι το **app.js**. Προφανώς, αυτό θα πρέπει να το αλλάξουμε, ώστε να ανταποκρίνεται στο όνομα του δικού μας προγράμματος, π.χ. index.mjs ή server.mjs ή όπως αλλιώς ονομάζεται το αρχείο μας. Θα μπορούσαμε εδώ να δώσουμε την τιμή:

```
"program": "${workspaceFolder}/${relativeFile}",
```

Έτσι, όταν εκσφαλματώσουμε θα τρέξει το τρέχον επιλεγμένο αρχείο στο Visual Studio Code.

Μια άλλη ιδιότητα που ίσως θέλουμε να αλλάξουμε είναι η name, που τώρα έχει τιμή “nodemon”. Μπορούμε να την ονομάσουμε «Αποσφαλμάτωση με nodemon» και να αποθηκεύσουμε το αρχείο.



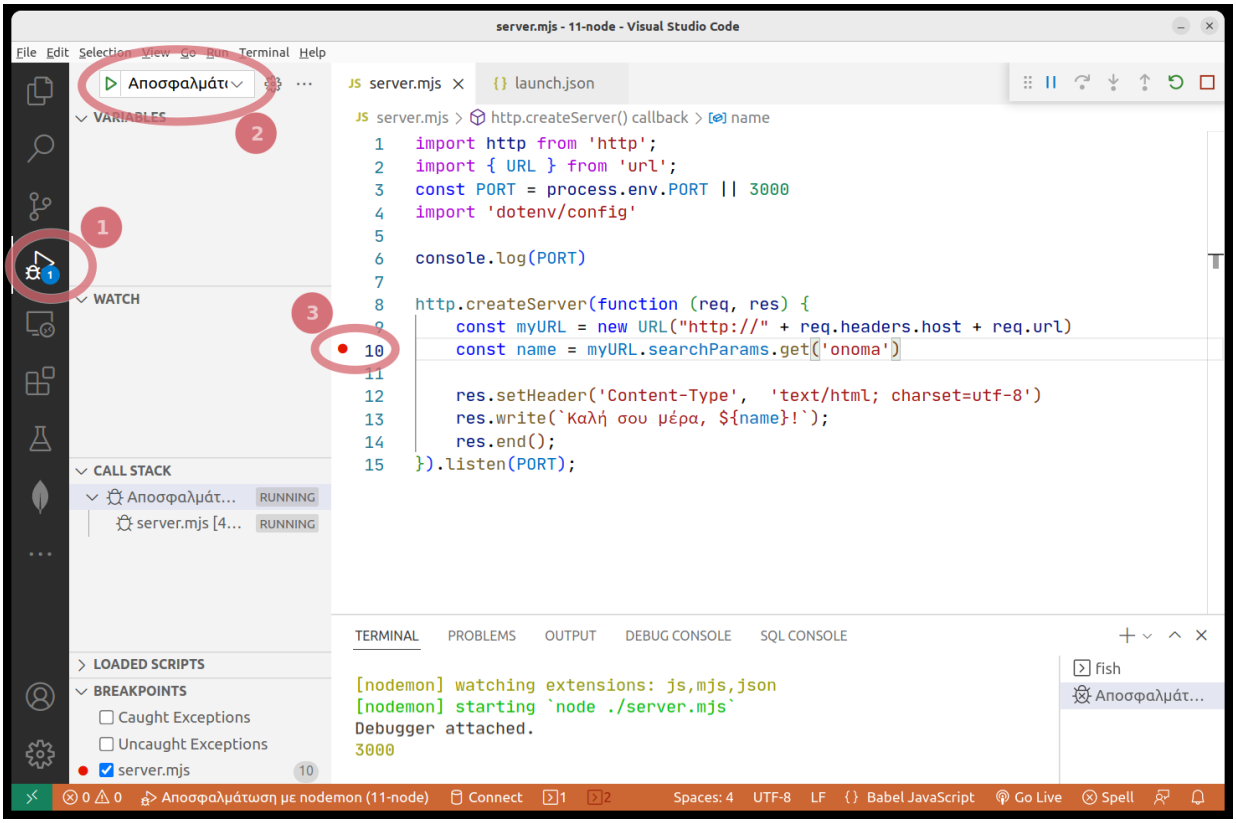
Εικόνα 11.10 *Προσαρμογή των επιμέρους ρυθμίσεων του launch configuration. Προσοχή στο όνομα του αρχείου που θέλουμε να εκσφαλματώσουμε.*

Με το launch configuration έτοιμο, έχουμε πλέον στη διάθεσή μας την επιλογή «Αποσφαλμάτωση με nodemon».

Χρήση του εργαλείου αποσφαλμάτωσης

Πλέον, μπορούμε να εκτελέσουμε τον κώδικά μας βήμα βήμα και να παρατηρήσουμε τις τιμές των μεταβλητών την ώρα της εκτέλεσης.

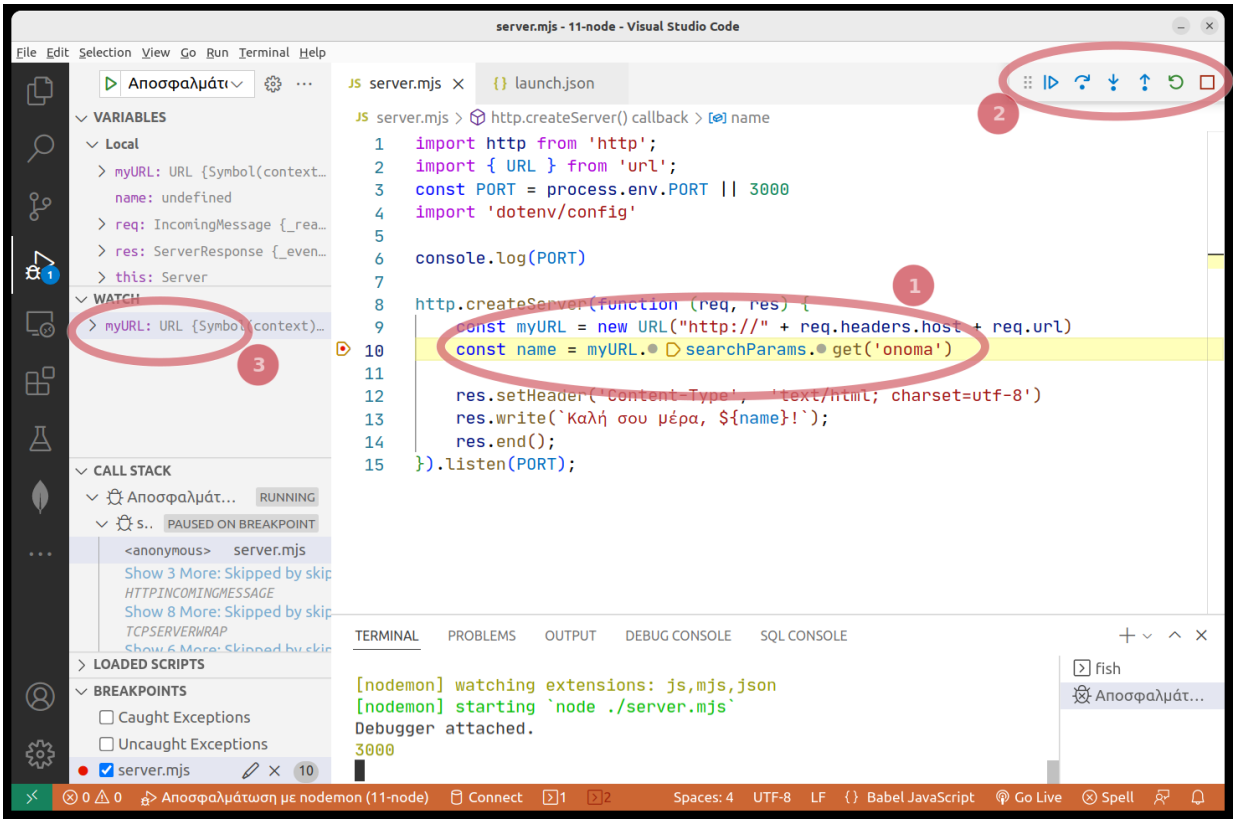
- Ανοίγουμε στο Visual Studio Code το αρχείο που αποτελεί την είσοδο στην εφαρμογή μας, π.χ. το αρχείο server.mjs.
- Μεταβαίνουμε στο εργαλείο αποσφαλμάτωσης με Ctrl+Shift+D (βήμα 1 στην **Εικόνα *11.11***).
- Επιλέγουμε το launch configuration που μόλις δημιουργήσαμε. Στο παράδειγμά μας αυτό είναι το «Αποσφαλμάτωση με nodemon» (βήμα 2 στην εικόνα 114).
- Ξεκινάμε την αποσφαλμάτωση πατώντας το πράσινο τρίγωνο (βήμα 2) ή πατώντας F5.



Εικόνα 11.11 *Το εργαλείο αποσφαλμάτωσης του Visual Studio Code.*

Η εφαρμογή έχει ξεκινήσει και, καθώς είναι εφαρμογή εξυπηρετητή, περιμένει κάποιον πελάτη για να στείλει την απάντησή της. Στο βήμα 3 στην **Εικόνα *11.11*** έχουμε ενεργοποιήσει ένα **breakpoint** με κλικ στη γραμμή 10. Αυτό σημαίνει πως το πρόγραμμα θα σταματήσει να εκτελείται ακριβώς πριν τη γραμμή 10 και θα μπορούμε να δούμε, για παράδειγμα, τις τιμές των μεταβλητών σε αυτό το σημείο.

Μόλις ανοίξουμε τη σελίδα μας στον φυλλομετρητή (http://127.0.0.1:3000), θα εκτελεστεί ο κώδικας μέχρι να φτάσει στο πρώτο breakpoint. Στο παράδειγμα της εικόνας βλέπουμε πως η εκτέλεση έχει σταματήσει προσωρινά στη γραμμή 10 (σημείο 1 στην εικόνα). Έχουμε στη διάθεσή μας διάφορα εργαλεία, τα πιο σημαντικά είναι το εργαλείο με το οποίο ελέγχουμε την εκτέλεση (σημείο 2) και το “Watch” όπου μπορούμε να προσθέσουμε τις μεταβλητές που μας ενδιαφέρουν οι τιμές τους (3).



Εικόνα 11.12 *Η εκτέλεση του κώδικα έχει σταματήσει προσωρινά στο breakpoint.*

11.9 Ερωτήσεις

1. Στη Node.js, παρ' όλο που δεν υπάρχει φυλλομετρητής, συνεχίζουμε να έχουμε πρόσβαση στις ιδιότητες των αντικειμένων **document** και **window**.
   – Σωστό/Λάθος
2. Οι παρακάτω προτάσεις περιέχουν ισχυρισμούς για τις διαφορές και ομοιότητες μεταξύ της JavaScript στη Node.js και της JavasScript στον φυλλομετρητή. Επιλέξετε όσους από τους ισχυρισμούς αληθεύουν.
   1. Ο φυλλομετρητής με τη χρήση της JavaScript μάς δίνει πρόσβαση στα τοπικά αρχεία του υπολογιστή μας.
   2. Ο βρόχος εκτέλεσης συμβάντων είναι ο ίδιος στη Node.js και στον φυλλομετρητή, αφού και στις δύο περιπτώσεις χρησιμοποιείται η μηχανή V8.
   3. Με την JavaScript στον φυλλομετρητή έχουμε δύο μηχανισμούς να φορτώσουμε βιβλιοθήκες, τον CommonJS και τον ES6.
   4. Στη Node.js, όπως και στον φυλλομετρητή, η JavaScript εκτελείται σε ένα νήμα [x].
   5. Τόσο στη Node.js όσο και στον φυλλομετρητή μπορούμε να γράψουμε ασύγχρονο κώδικα με συναρτήσεις επιστροφής ("callback") καθώς και με υποσχέσεις (το Promise API).
   6. Η Node.js δεν μας δίνει πρόσβαση στο Document Object Model (DOM) [x].
3. Για να χρησιμοποιήσουμε τις ενσωματωμένες (core) βιβλιοθήκες της Node.js δεν χρειάζεται να κάνουμε κάτι ιδιαίτερο πέρα από να τις φορτώσουμε.
   – Σωστό/Λάθος
4. Είναι προτιμότερο να χρησιμοποιούμε τη σύγχρονη εκδοχή της βιβλιοθήκης fs.
   – Σωστό/Λάθος
5. Οι ενσωματωμένες (core) βιβλιοθήκες της Node.js παρέχονται από τη μηχανή V8 και γι' αυτό είναι διαθέσιμες και στον φυλλομετρητή.
   – Σωστό/Λάθος
6. Για ποιο λόγο χρησιμοποιούμε το try/catch όταν χειριζόμαστε αρχεία;

1. Για να χειριστούμε σφάλματα που πιθανά προκύψουν κατά τον χειρισμό των αρχείων.
2. Είναι υποχρεωτική η χρήση του μπλοκ try/catch όταν χειριζόμαστε αρχεία.
3. Το try/catch δεν είναι δομή που μπορούμε να χρησιμοποιήσουμε με την JavaScript.

7. Με τις συναρτήσεις επιστροφής ο κώδικας δεν εκτελείται αναγκαστικά με τη σειρά που είναι γραμμένος.
   – Σωστό/Λάθος
8. Με τις υποσχέσεις (Promise API) μπορούμε να κατασκευάσουμε αλυσίδες όπου κάθε βήμα εκτελείται μόνο αν έχει ολοκληρωθεί το προηγούμενο.
   – Σωστό/Λάθος
9. Η Node.js μπορεί να χρησιμοποιήσει modules τόσο με τον μηχανισμό CommonJS όσο και με τον ES6.
   – Σωστό/Λάθος
10. Με τον μηχανισμό CommonJS χρησιμοποιούμε την **import** για να φορτώσουμε βιβλιοθήκες, ενώ με τον μηχανισμό ES6 χρησιμοποιούμε τη **require**.
    – Σωστό/Λάθος
11. Ένας τρόπος που έχουμε για να χρησιμοποιήσουμε το σύστημα ES6 για να φορτώσουμε βιβλιοθήκες είναι να αποθηκεύσουμε τα αρχεία μας με κατάληξη .mjs
    – Σωστό/Λάθος
12. Ένα πακέτο είναι ένας φάκελος που περιέχει ένα αρχείο JSON με όνομα **package.json**.
    – Σωστό/Λάθος
13. Στο αρχείο **package.json**, αν δώσουμε στο πεδίο “type” την τιμή “module”, τότε θα χρησιμοποιηθεί ο μηχανισμός ES6 και αναγκαστικά θα πρέπει να αποθηκεύσουμε το αρχείο μας με κατάληξη **.mjs**
    – Σωστό/Λάθος
14. Όλες οι εξαρτήσεις του πακέτου μας που περιγράφονται στο αρχείο **package.json** κατεβαίνουν και αποθηκεύονται στον υποφάκελο **node_modules**.
    – Σωστό/Λάθος
15. Αν διαγράψουμε τα περιεχόμενα του υποφακέλου **node_modules**, μπορούμε να τα κατεβάσουμε ξανά με την εντολή npm install.
    – Σωστό/Λάθος
16. Οι εξαρτήσεις ανάπτυξης (dev dependencies) δεν επηρεάζονται αν διαγράψουμε τον υποφάκελο **node_modules**, καθώς αποθηκεύονται σε άλλον φάκελο.
    – Σωστό/Λάθος
17. Επιλέξετε όσα από τα παρακάτω είναι συστατικά ενός αιτήματος HTTP:
    1. μέθοδος HTTP (method)
    2. μονοπάτι του πόρου (path)
    3. κωδικοποίηση χαρακτήρων (encoding)
    4. αριθμός έκδοσης του πρωτοκόλλου (protocol version)
    5. κεφαλίδες αιτήματος (request headers)
    6. σώμα του αιτήματος (request body)
    7. μηχανισμός module (commonjs/ES6)
18. Στη Node.js, με κάθε αίτημα HTTP που λαμβάνει η εφαρμογή μας, έχουμε διαθέσιμο το αντικείμενο **request** που περιέχει τις πληροφορίες του αιτήματος. Το αντικείμενο **response** με το οποίο θα στείλουμε την απάντηση δεν είναι διαθέσιμο και θα πρέπει να το κατασκευάσουμε.
    – Σωστό/Λάθος
19. Αν σε ένα αίτημα με τη μέθοδο GET έχουμε τιμές για περισσότερες από μια παραμέτρους, τότε το διαχωριστικό για να τις ξεχωρίζουμε είναι ο χαρακτήρας
    1. ;
    2. ,
    3. &
    4. +
20. Αν σε μια φόρμα δεν οριστεί η μέθοδος αποστολής, τότε η φόρμα
    1. δεν θα αποσταλεί.
    2. θα αποσταλεί με τη μέθοδο GET.
    3. θα αποσταλεί με τη μέθοδο POST.
21. Για να εγκαταστήσουμε ένα πακέτο με το npm έτσι ώστε να είναι ορατό σε όλο το σύστημά μας και ανεξάρτητα από το συγκεκριμένο πακέτο που αναπτύσσουμε, μπορούμε να γράψουμε

1. npm install <όνομα πακέτου>
2. npm install -g <όνομα πακέτου>
3. npm i <όνομα πακέτου>
4. npm g <όνομα πακέτου>

11.10 Βιβλιογραφία και Αναφορές

Ο αναγνώστης μπορεί να βρει λεπτομερείς οδηγίες χρήσης καθώς και σε βάθος περιγραφή των χαρακτηριστκών της Node.js στην επίσημη τεκμηρίωσή της. Πιο προσβάσιμοι οδηγοί για τη χρήση της Node.js υπάρχουν στη σελίδα nodejs.dev, που συντηρείται και ενημερώνεται από την κοινότητα.

Η ιστοσελίδα MDN αποτελεί και για αυτό το κεφάλαιο πολύτιμη πηγή. Έτσι, μπορεί κανείς να μελετήσει αναλυτικά τη χρήση του συστήματος βιβλιοθηκών ES6 Modules καθώς και τη χρήση του async/await.

Στον ιστότοπο του mathesis οι συγγραφείς έχουν δημοσιεύσει ένα διαδικτυακό μάθημα με τίτλο «**Ανάπτυξη διαδικτυακών εφαρμογών με Node.js**», στο περιεχόμενο του οποίου έχει εν μέρει βασιστεί και το υλικό που παρουσιάζεται σε αυτό και στα επόμενα κεφάλαια. https://mathesis.cup.gr/courses/course-v1:ComputerScience+CS3.3+22D/about

Α. Ξενόγλωσσες

Doglio, F. (2018). *REST API Development with Node.js*. Apress.

Herron, D. (2020). *Node.js Web Development* (5th ed.). Packt.

Mardan, A. (2018). *Practical Node.js, Building Real-World Scalable Web Apps* (2nd ed.). Apress.

Συλλογικό (2022). *Node.js API Reference Documentation*. Node.js. Ανακτήθηκε στις 14 Σεπτεμβρίου 2022 από https://nodejs.org/en/docs/

Β. Ελληνόγλωσσες

Σιντόρης, Χ., & Αβούρης, Ν. (2022). Ανάπτυξη διαδικτυακών εφαρμογών με Node.js. Ανοικτό διαδικτυακό μάθημα, https://mathesis.cup.gr.