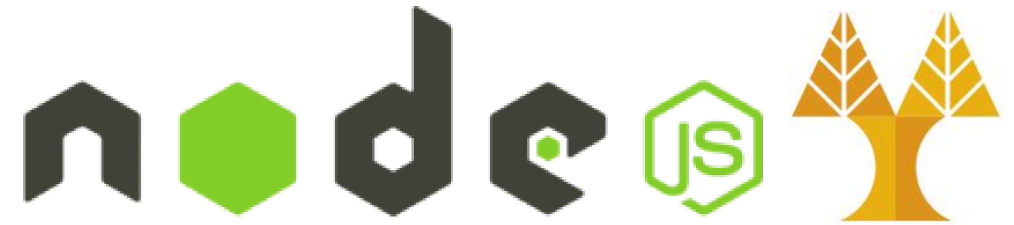
Internet Technologies

Introduction to Node.js and Express

University of Cyprus
Department of Computer Science

Node.js



- Open-source, cross-platform runtime environment that allows developers to create all kinds of server-side tools and applications in JavaScript
  - Runtime is intended for use outside of a browser context (i.e. running directly on a computer or server OS) → Download & install: https://nodejs.org/en/download
  - relies on modules: built-in (e.g. http), third-party (e.g. express), user-defined
- Creating your own module is easy
  - Just put your JavaScript code in a separate js (e.g. mymodule.js) file and include it in your code by using keywork require, like

```
var modulex = require('./mymodule');
```

- Libraries in Node.js are called packages and they can be installed using NPM by typing: `npm install package_name` on terminal

Benefits

- **Great performance!** Node was designed to optimize throughput and scalability in web applications and is a good solution for many common web-development problems (e.g. real-time web applications).
- **Fast development!** Code is written in "plain old JavaScript", which means that less time is spent dealing with "context shift" between languages when you're writing both client-side and server-side code.
- **Access to numerous packages!** The node package manager (npm) provides access to hundreds of thousands of reusable packages. It also has best-in-class dependency resolution and can also be used to automate most of the build toolchain.
- **Portable development!** It is available on Microsoft Windows, macOS, Linux, Solaris, FreeBSD, OpenBSD, WebOS, and NonStop OS. Furthermore, it is well-supported by many web hosting providers, that often provide specific infrastructure and documentation for hosting Node sites.
- It has a very **active third-party ecosystem** and developer community, with lots of people who are willing to help.

Node.js built-in modules

Module name	Description
console	console module is used to print information on stdout and stderr.
fs	fs module is used for File I/O (read/write data on disk).
http	http module is used transfer data over the HTTP protocol. The HTTP module creates an HTTP server that listens to server ports and gives a response back to the client.
net	net module provides servers and clients as streams. Acts as a network wrapper.
os	os module provides basic operating system related utility functions.
path	path module provides utilities for handling and transforming file paths.
process	process module is used to get information on current process.

Create a Node.js application/project

- Create a folder to hold your application/project
- Open VS Code
- Open newly created folder (1) within VS Code
- Open terminal (2) to run commands

File Edit Selection View Go Run Terminal Help
Search
Welcome
2
Visual Studio Code
Editing evolved
Start
New File...
Open File...
Open Folder...
Clone Git Repository...
Connect to...
1
Recent
Lab12 - Nodejs and Express C:\Users\admin\Dropbox\EPL425 - Internet Technol...
AdminApp C:\Users\admin\OneDrive - University of Cyprus\MySoftwareProjects...
AdminApp C:\Users\admin\OneDrive - University of Cyprus\MySoftwareProjects
MySoftwareProjects C:\Users\admin\OneDrive - University of Cyprus
AdminPanel C:\Users\admin\OneDrive - University of Cyprus\MySoftwareProjec...
More...
Walkthroughs
Get Started with VS Code
Customize your editor, learn the basics, and start coding
Learn the Fundamentals
Get Started with Python Development Updated
Get Started with Jupyter Notebooks Updated
Get Started with WSL Updated
More...
Show welcome page on startup

Simple HTTP server in Node.js

- Create a web server that listens for any HTTP request on http://127.0.0.1:8000/
- When a request is received, the script responds with the string: "Hello World“
- Execute the command `node <filename>` on terminal

hello_nodejs.js

```javascript
// Load HTTP module
const http = require("http");

const hostname = "127.0.0.1";
const port = 8000;

// Create HTTP server
const server = http.createServer(function (req, res) {
    // Set the response HTTP header with HTTP status and Content type
    res.writeHead(200, { "Content-Type": "text/plain" });
    // Send the response body "Hello World"
    res.end("Hello World\n");
});

// Prints a log once the server starts listening
server.listen(port, hostname, function () {
    console.log(`Server running at http://${hostname}:${port}/`);
});
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS node

```
PS C:\Users\admin\Dropbox\EPL425 - Internet Technologies\05 - Labs\Lab12 - Nodejs and Express> node hello_nodejs.js
Server running at http://127.0.0.1:8000/
```

Simple HTTP server in Node.js

Home Workspaces API Network Upgrade

My Workspace New Import

GET localhost:8000

No environment

Collections

Environments

History

Your collecton

Your collection

Authorization

Type API Key

Create a collection for your requests

A collection lets you group related requests and easily set common authorization, tests, scripts, and variables for all requests in it.

Create Collection

localhost:8000 Save

GET localhost:8000 Send

Params Auth Headers (9) Body Pre-req. Tests Settings Cookies

Query Params

Key	Value	Description	Bulk Edit

Body 200 OK 37 ms 169 B Save as example

Pretty Raw Preview Visualize Text

```
1 Hello World
2
```

Online Find and replace Console Postbot Runner Start Proxy Cookies Trash

Web frameworks

- Common web-development tasks are not directly supported by Node itself:
  - Add specific handling for different HTTP methods (e.g. GET, POST, DELETE, etc.)
  - Separately handle requests at different URL paths ("routes" or endpoints)
  - Serve static files, or use templates to dynamically create the response
- Node won't be of much use on its own
- You will either need to write the code yourself, or you can avoid reinventing the wheel and use a web framework!

Express

- The most popular Node web framework. It provides mechanisms to:
  - Write handlers for requests with different HTTP methods at different URL paths (routes)
  - Integrate with "view" rendering engines in order to generate responses by inserting data into templates – for static and dynamic websites
  - Add additional request processing "middleware" at any point within the request handling pipeline
- Express itself is fairly minimalist but developers have created compatible middleware packages to address almost any web development problem:
  - Libraries to work with cookies, sessions, user logins, URL parameters, POST data, security headers, and many more

Create an Express application/project

- Create a folder to hold your application/project
- Open VS Code
- Open newly created folder within VS Code
- Open terminal and run node install express to install Express

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\admin\Dropbox\EPL425 - Internet Technologies\05 - Labs\Lab12 - Nodejs and Express> npm install express

added 64 packages in 6s

12 packages are looking for funding
  run `npm fund` for details
PS C:\Users\admin\Dropbox\EPL425 - Internet Technologies\05 - Labs\Lab12 - Nodejs and Express>
```

package.json and package-lock.json

- After Express installation, 2 new files are automatically created:
  - package.json: It contains the list of dependencies required for the project, along with their desired version ranges specified using semantic versioning or specific version numbers.
  - package-lock.json: It includes the specific resolved versions of all the dependencies, their sub-dependencies, and their exact installation locations. It acts as a snapshot of the dependency tree for ensuring consistent installations.

```json
{
  "dependencies": {
    "express": "^4.19.2"
  }
}
```

Simple HTTP server in Express

- Fewer lines of source code compared to Node

hello_express.js

```
// Load Express module
const express = require("express");
// Create Express application
// app object has methods for routing HTTP requests, configuring middleware,
// rendering HTML views, registering a template engine, and modifying application
// settings that control how the application behaves
const app = express();
const port = 8001;

// Route definition
// Callback function will be invoked whenever there is an HTTP GET request with
// a path ('/') relative to the site root
app.get("/", function (req, res) {
    // send() is used to reply with the string "Hello World!"
    res.send("Hello World!");
});

// Launches the server on the specified port; prints a log comment to console
app.listen(port, function () {
    console.log(`Example app listening on port ${port}!`);
});
```

Importing and creating modules

- A module is a JavaScript library/file that you can import into other code using Node's require() function e.g. `require("express")`
- User-defined modules can be imported in the same way:
  - To make objects available outside of a module you just need to expose them as additional properties on the exports object
    - For example, the square.js module below is a file that exports area() and perimeter() methods

square.js

```
exports.area = function (width) {
  return width * width;
};
exports.perimeter = function (width) {
  return 4 * width;
};
```

otherfile.js

```
// require() the name of the file
without the (optional) .js extension
const square = require("./square");

console.log(`The area of a square with
a width of 4 is ${square.area(4)}`);
```

Importing and creating modules

- If you want to export a complete object in one assignment instead of building it one property at a time, assign it to module.exports

square.js

```
module.exports = {
  area(width) {
    return width * width;
  },

  perimeter(width) {
    return 4 * width;
  },
};
```

Creating route (endpoint) handlers

- Routing refers to how an application's endpoints (URIs) respond to client requests
- Routing is defined using methods of the Express app object that correspond to HTTP methods
    - app.get() to handle GET requests
    - app.post to handle POST requests
- Routing methods specify a callback function (handler function) called when the application receives a request to the specified route (endpoint)

```
const express = require('express')
const app = express()

// GET method route
app.get('/books', function (req, res) => {
  // do something
})

// POST method route
app.post('/books', function (req, res) => {
  // do something
})

// PUT method route
app.put('/books', function (req, res) => {
  // do something
})
```

Callback functions (handlers)

- Invoked when an HTTP request is received on a specific path

```
// Callback function must take a request and a response object as arguments
app.get("/books", function (req, res) {
    // send() method called on response object to issue a reply message
    res.send({'id':1, 'title':'Odyssey');
});
```

- Response object can be used to send some response back to the requester
  - `res.send([body])` sends data and ends the request
    - body can be any of the following Buffer, String, Object, Array
    - Automatically sets the Content-Type response header based on the argument passed. However, we can programmatically set the Content-Type header is possible via the set() method on the res object: `res.set('Content-Type', 'application/json');`
  - `res.json()` sends data in JSON format and ends the request
    - identical to res.send() when an object or array is passed, but it also converts non-objects to json

SEE ALL RESPONSE METHODS

Setting a callback function (handler)

- First way

```
app.get("/books", function (req, res) {
    // do something
});
```

- Second way

```
app.get("/books", (req, res) => {
    // do something
});
```

- Third way

```
const f1 = function (req, res) {
    // do something
}
app.get("/books", f1);
```

Route parameters

- Pass values (route parameters) via the endpoint url

```
app.get("/books/:bid/reviews/:rid", function (req, res) {
    res.send("Book id: " + req.params['bid'] + " and review id: " + req.params['rid']);
});
```

- Captured values are populated in the `req.params` object
    - names of route parameters specified in the path are found as object keys
    - The name of route parameters must be made up of “word characters” ([A-Za-z0-9_]).

Using middleware

- Route functions (we have already seen) end the HTTP request-response cycle by returning some response to the HTTP client
- Middleware functions typically perform some operation on the request or response and then call the next function in the "stack", which might be more middleware or a route handler
    - The order in which middleware is called is up to the app developer.

Callback functions as middleware

- We can provide multiple callback functions that behave like middleware to handle a request. Need to invoke next() to proceed to the next callback.

```
const f1 = function (req, res, next) {
    // do something, e.g. checks on the received data
    next(); // proceed to the next callback function
}

const f2 = function (req, res) {
    res.send("John Smith");
}

app.get("/users", [f1, f2]);
```

Third-party middleware package

- Install [morgan]() HTTP request logger middleware:
  - npm install morgan
- Use morgan middleware

```
const express = require("express");
const logger = require("morgan");
const app = express();

// call use() on the Express application object to add the middleware to the stack
// i.e. to the processing chain of all responses
app.use(logger("dev"));
```

Effect after applying the morgan middleware: whenever a request is received at the listening server-side program, morgan logger prints a dedicated message to console

```
Example app listening on port 8001!
GET /books/23/reviews/13 200 7.023 ms - 29
GET /books 200 1.827 ms - 26
```

User-defined function as middleware for handling error

- Errors are handled by one or more special middleware functions that have four arguments (err, req, res, next) instead of the usual three. For example:

```
app.use(function (err, req, res, next) {
  console.error(err.stack);
  res.status(500).send("Something broke!");
});
```

- This function can return any content required, but **must be called after all other app.use() and routes calls** so that it is the last middleware in the request handling process!
- In order to have this function called, an error must be sent via the next() function from a previous middleware or a route handler (see example later)

Body parsers as middleware

- Some messages (e.g. POST, PUT) usually carry data in body
- Data are expressed in multiple formats: JSON, URL-encoded, plain text, multipart/form-data, etc.
- Appropriate body parser(s) need to be added as middleware prior defining route handlers:
  - Body parsers add a body property to the Express request `req` so data is accessible via `req.body`
  - For JSON data: `app.use(express.json());`
    - JSON parser is applied when received message header `Content-Type` is `application/json`
    - If JSON body is malformed, custom error handling middleware will be called (if any, see previous slide) otherwise the default Express error handling middleware will respond with an HTTP 400.

Body parsers as middleware: example

- Receive JSON data from a POST message

```
// Parse JSON bodies
// Make sure you put `app.use(express.json())` **before** your route handlers!
app.use(express.json());

app.post("/books", function (req, res) {
    // The json() middleware adds a body property (object) to the Express request req
    console.log(req.body);
});
```

Using databases

- Express apps can use any database mechanism supported by Node.js including: PostgreSQL, MySQL, MariaDB, MSSQL, Redis, SQLite, MongoDB
  - Database driver must be installed using npm in advance
- Install MySQL driver (can be used to connect to MySQL and MariaDB)
  - npm install mysql
- Use MySQL

```
const mysql = require("mysql");
```

Connect to database

- Create a single connection to database and use it throughout the program (to make multiple queries on the database tables)

```
const con = mysql.createConnection({
    host: "localhost",
    user: "root",
    password: "",
    database: "library"
});

con.connect(function(err) {
    if (err) {
        // send error to the next middleware (for handling errors)
        return next(err);
        // return next(); will call the default Express error handling middleware
    }
});
```

Query the database

- Use the single connection that was initially created to run queries on the database tables

```
app.get("/books", function (req, res, next) {
    con.query("SELECT * FROM books", function (err, result, fields) {
        if (err) // error on query
            return next(err);
        else
            res.send(result);
    });
})
```

Array of objects

- What happens if a query to database takes a long time to return results? Is your system able to serve an HTTP request from another client?
  - No, only one connection to db is available for all incoming HTTP requests

Connect to database using connection pool

- Connection pool: set of database connections maintained so that the connections can be reused when future requests to the database are required - Connection pools are used to enhance the performance of executing commands on a database

```
const pool  = mysql.createPool({
    connectionLimit: 10, // in this example, pool allows up to 10 parallel connections
    host: 'localhost',
    user: 'root',
    password: '',
    database: 'library'
});
app.get("/books", function (req, res, next) {
    pool.query("SELECT * FROM books", function (err, result, fields) {
        if (err) return next(err);
        else
            res.send(result);
    }); // when a query is over, the connection is released and is returned to the pool
});
```

More info on connection pooling can be found here.

Exercise

- Create a REST API server (with connection pool) in Node.js/Express that defines the following endpoints:

Method	Endpoint	Description	Status response code	Returned data
GET	/books	Returns all books*	200 OK	Array of books
GET	/books/:id	Returns a single book indicated by its id*	200 OK / 404 Not found^	A book / {'status': 'ok’}^
POST	/books	Creates a new book**	201 Created	{'status': 'ok'}
PUT	/books/:id	Updates a book**	200 OK	{'status': 'ok'}
DELETE	/books	Deletes all books	204 No Content	{'status': 'ok'}
DELETE	/books/:id	Deletes a single book indicated by its id	204 No Content	{'status': 'ok'}

This exercise requires a database with a books table with the same structure as described in the previous lab.
You can create the database and the books table in XAMPP (via phpMyAdmin) by importing the library.sql file provided in EPL425 lab web page (if it is not already in place).

* Data is returned to client in JSON format
** Data is provided by client in JSON format
^ If no book found in database

Appendix

Response methods

- Methods on the response object (res) can send a response to the client, and terminate the request-response cycle. If none of these methods are called from a route handler, the client request will be left hanging.

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process. No response message will be sent.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a view template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.