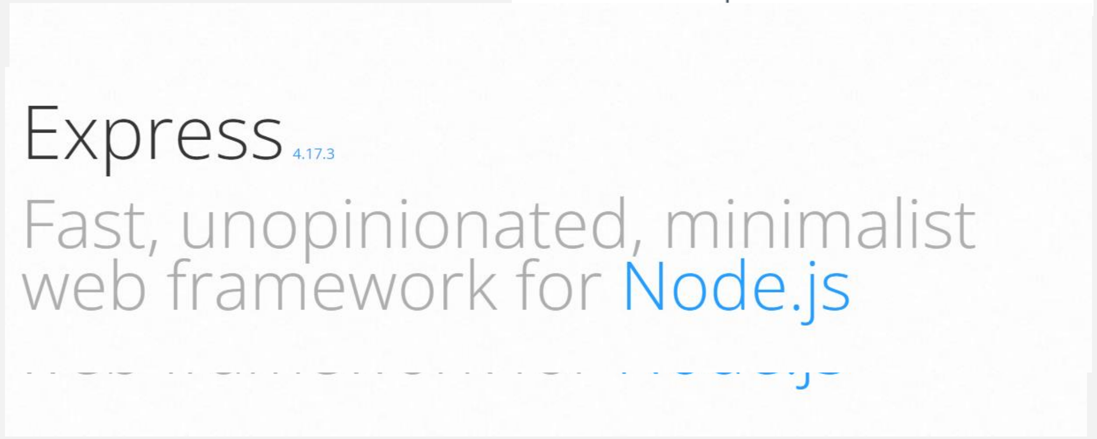
POLITÉCNICO
DO PORTO
ESMAD

PROGRAMAÇÃO WEB II

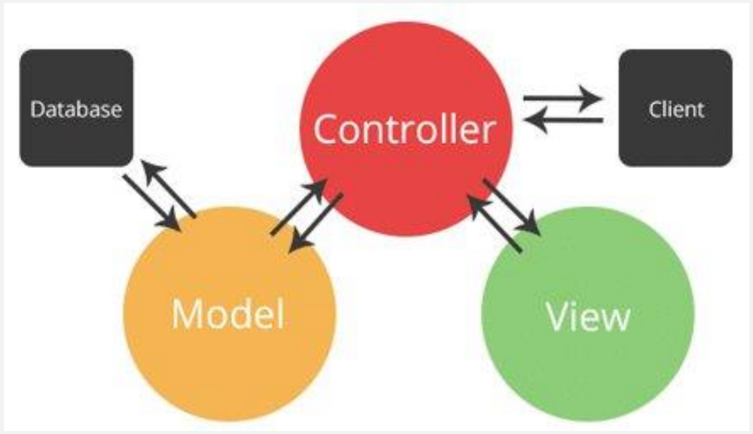
TSIW

SUMMARY

- Introduction to Express
- MVC: Model, View e Controller


Express 4.17.3

Fast, unopinionated, minimalist web framework for Node.js



Express – introduction

Express 4.17.3
Fast, unopinionated, minimalist web framework for Node.js

- Because vanilla Node.js can be **verbose**, **confusing** and **limited in features**, Express is a framework that acts as a **light layer** atop the Node.js web server making it easier to develop Node.js web applications
- It is used by many companies
- It simplifies the APIs of Node.js, adds **helpful features**, helps organizes the application's functionality with **middleware and routing**, adds helpful utilities to Node.js's HTTP objects and facilitates rendering of dynamic HTML views
- It helps to wrap a standard Node.js application around an **MVC architecture**

Express – how to install

- Assuming Node is already installed, create a new Node project in a folder of your choice and set up a default package.json file
  - ➢ `npm init -y`
- Install Express locally and check that it is added to dependencies parameter of package.json
  - ➢ `npm install express`
- How to use in your Node application? Just include

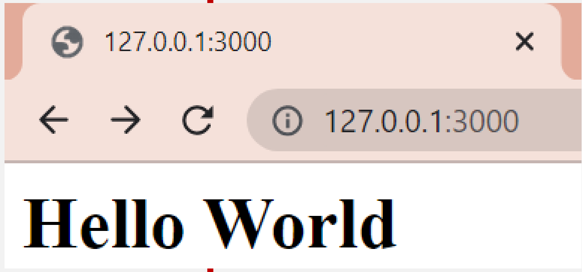
```
const express = require('express');
```

- To appreciate how much easier Express makes Node apps development, let's use it to **repeat some examples from what we have previously done**

Express – Hello World

- Example 1: create a simple web server

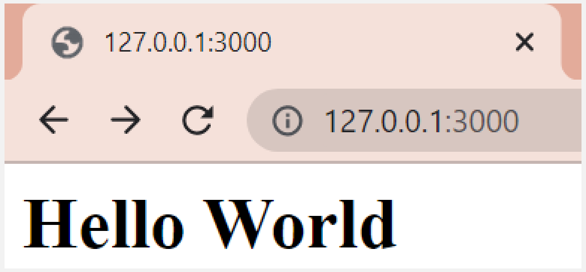
```
// import Express
const express = require('express');

const app = express();
const hostname = '127.0.0.1';
const port = process.env.PORT || 3000;

// sets the server response to a GET request on URL “/”
app.get('/', (req, res) => {
    res.send('<html><body><h1>Hello World</h1></body></html>');
})

//  server creation and listening for any incoming requests
app.listen(port, hostname, (error) => {
     console.log(`App listening at http://${hostname}:${port}/`)
})
```



Hello World

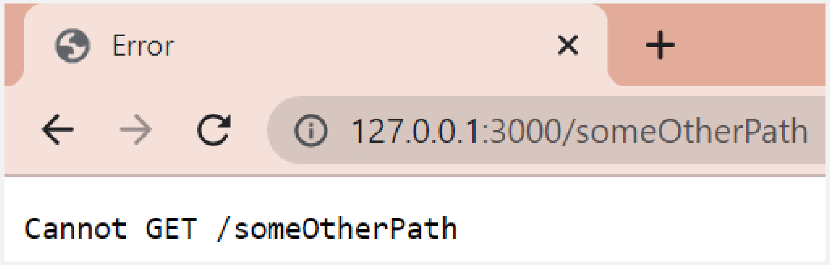
Express – Hello World

- Example 1: create a simple web server
  - Express takes care of the `http`, `request` and `response` objects "behind the scenes"
  - The callback function provided in the last argument in `app.listen()` is executed when the servers starts listening
  - Express allows greater flexibility in responding to browser HTTP requests
  - The app responds with "Hello World!" for requests to the root URL (/) or route
  - For every other path, it will respond with a 404 Not Found



Hello World

Success response to HTTP request
`GET /`

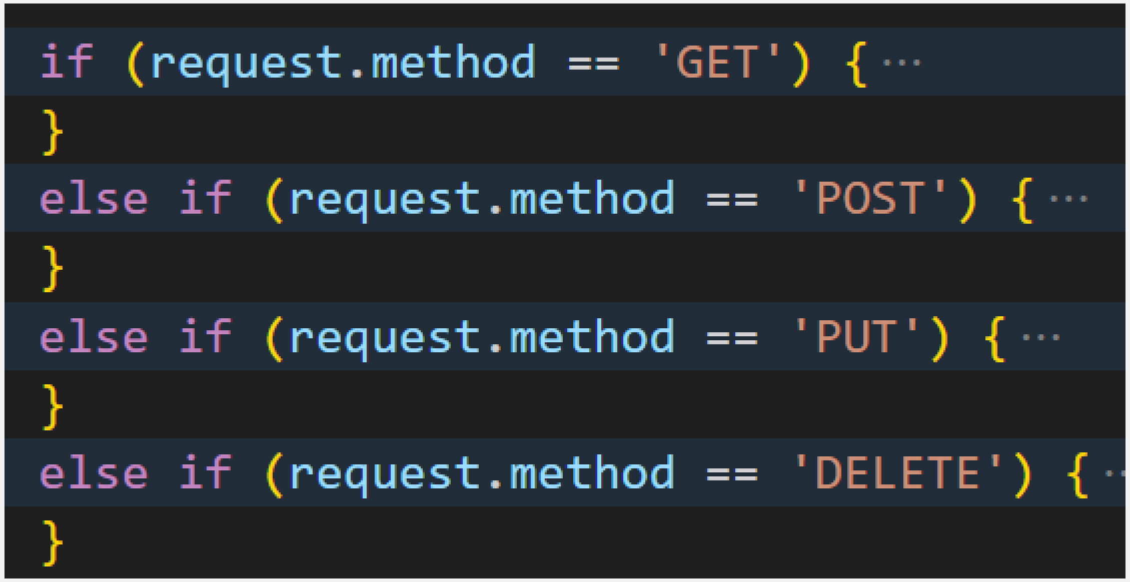


```
Cannot GET /someOtherPath
```

Error response to HTTP request
`GET /someOtherPath`

Express – how does it help?

- **Routing**: refers to determining how an application responds to a client request to a particular endpoint
  - Endpoint: a URI (or path) + a specific HTTP request method (GET, POST, …)
- Previously without Express, to respond to individual routes required an **extended *if-else* statement in one big request handler**



```
if (request.method == 'GET') {···
}
else if (request.method == 'POST') {···
}
else if (request.method == 'PUT') {···
}
else if (request.method == 'DELETE') {·
}
```

Express – how does it help?

- With Express, it is possible to refactor one big request handler function into **many smaller request handlers** that each handle a specific function
  - This allow to build apps in a more **modular** and **maintainable** way
  - Route definition takes the following structure

```
app.METHOD(PATH, HANDLER)
```

app is an instance of Express

METHOD is an HTTP request method, in lowercase

PATH is the request URI

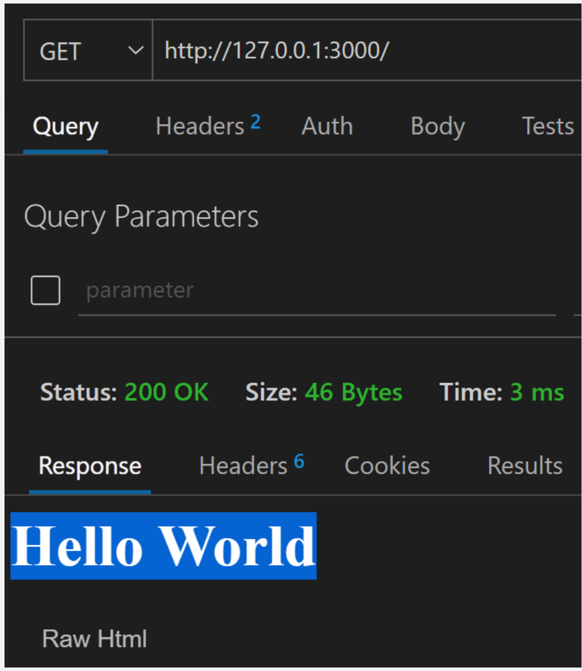
HANDLER is the callback function executed when the route is matched

Express – routing

- The following examples illustrate defining simple routes:

```
// respond with Hello World on the root route (/), the
application's home page
app.get('/', (req, res) => {
    res.send('Hello World');
})
```

```
// respond to a POST request on route /users
app.post('/users', (req, res) => {
    res.send('Got a POST request');
})
```



Hello World

- Express supports methods that correspond to all HTTP request methods: GET, POST, PUT, DELETE...
  - For a full list, see the documentation about app.METHOD

Express – routing

There are two special routing methods:

1. `app.all()`, used to handle **all HTTP** request **methods**
2. `app.use()`, used to specify **middleware** as the callback function
   - Middleware functions are functions that have access to the request object (**`req`**), the response object (**`res`**), and the **`next`** function in the application's request-response cycle
   - The **`next`** function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware
   - For example, the following handler is executed for **all requests** using **any HTTP request method** supported in the Node `http` module

```
// middleware function executed for EVERY request to the app
app.use( function (req, res, next) => {
  console.log('Time: %d ...', Date.now())
  next() // pass control to the next handler
})
```

Express – routing

- One can create chainable route handlers for a route path by using **app.route()**
  - Because the path is specified at a single location, creating modular routes is helpful, as is reducing redundancy and typos

```
app.route('/book')
  .get((req, res) => {
    res.send('Get a random book')
  })
  .post((req, res) => {
    res.send('Add a book')
  })
  .put((req, res) => {
    res.send('Update a book')
  })
  .delete((req, res) => {
    res.send('Delete a book')
  })
```

Express – routing

- Route parameters are named URL segments that are used to capture the values specified at their position in the URL
  - In Express these identifiers are defined using colons **:**
  - The captured values are populated in the `req.params` object, with the name of the route parameter specified in the path as their respective keys

```
app.get('/users/:id', (req, res) => {
     res.send('The id you specified was ' + req.params.id);
})
```

Request URL: http://localhost:3000/users/34
`req.params` { "id": "34" }

```
app.get('/users/:userId/books/:bookId', (req, res) => {
     res.send(req.params);
})
```

Request URL: http://localhost:3000/users/34/books/8989
`req.params` { "userId": "34", "bookId": "8989" }

Express – route paths

- Pattern matched routes: routes can also be **string patterns** or **regular expressions** (it is also possible to use regex to restrict URL parameter matching):

```
// this route path will match abcd, abANYTHING_HEREcd, …
app.get('/ab*cd', (req, res) => {
   res.send(req.originalUrl);
});
```

```
// this route path will match acd and abcd
app.get('/ab?cd', (req, res) => {
   res.send(req.originalUrl);
});
```

```
// this route path will match anything finished with fly
app.get(/.*fly$/, (req, res) => {
   res.send(req.originalUrl);
});
```

Handy tool for testing basic Express routes, although it does not support pattern matching: https://forbeslindesay.github.io/express-route-tester/

Express – route paths

- Since the hyphen (-) and the dot (.) are interpreted literally, they can be used along with route parameters for useful purposes

```
app.get('/flights/:from-:to', (req, res) => {
   res.send('Flight from: ' + req.params.from + ' to: ' + req.params.to);
});
```

Request URL: http://localhost:3000/flights/LIS-OPO
`req.params`{ "from": "LIS", "to": "OPO"}

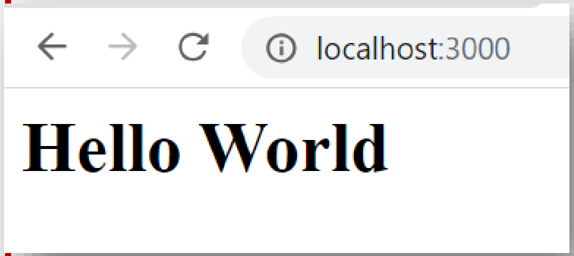
```
app.get('/plantae/:genus.:species', (req, res) => {
   ...;
});
```

Request URL: http://localhost:3000/plantae/Prunus.persica
`req.params` { "genus": "Prunus", "species": "persica"}

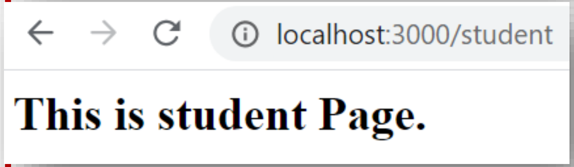
Express – how does it help?

- **Example 2**: create a simple web server (with more routes)
  - ➢ define specific routes and the response the server gives when a route is hit

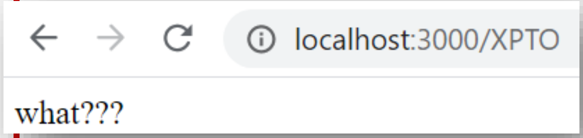
```
const express = require('express');
const app = express();

const hostname = '127.0.0.1';
const port = process.env.PORT || 3000;

// sets the server response to a GET request to URL "/"
app.get('/', (req, res) => {
    res.send('<html><body><h1>Hello World</h1></body></html>');
})
// sets the server response to a GET request to URL "/student"
app.get('/student', (req, res) => {
    res.send('<html><body><h2>This is student Page.</h2></body></html>');
})
//send a predefined error message if client asks for invalid route
app.all('*', function (req, res) {
     res.status(404).send('what???');
})

app.listen(port, hostname, () => {
    console.log(`App listening at http://${hostname}:${port}/`)
})
```


localhost:3000

Hello World


localhost:3000/student

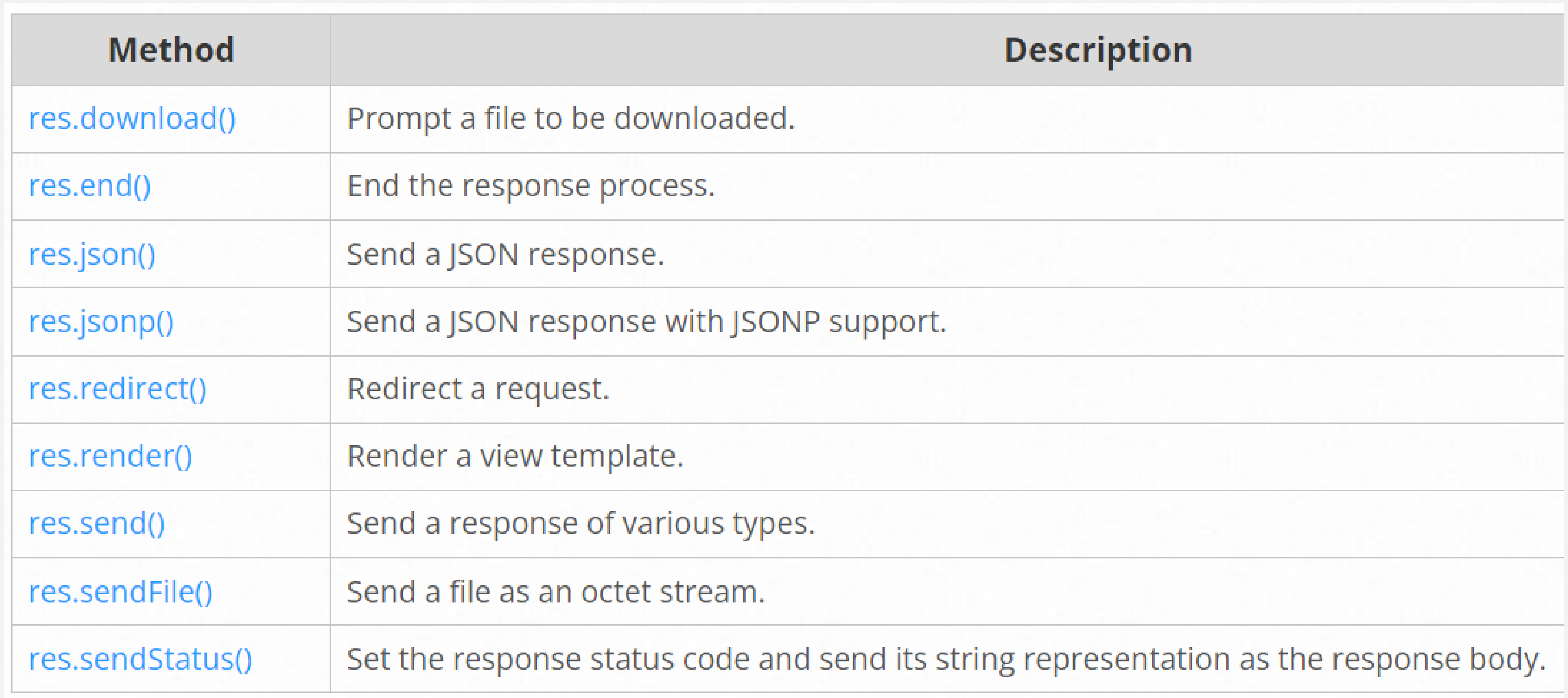
This is student Page.


localhost:3000/XPTO

what???

Express – how does it help?

- **Response methods**: methods used by the response object to send a response to the client and end 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.
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.

Express – how does it help?

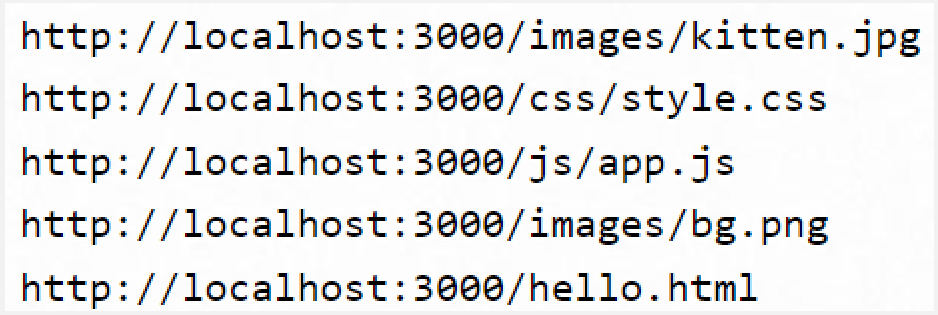
- **Example 3**: alter the type of response to HTML **static files**
  - to include serving static files like fonts, images, CSS or others with Express, one must add the Express built-in middleware **express.static**

```
express.static(root, [options])
```

- `root`: root directory from which to serve static assets

Example #01

```
// all files in a '/public' directory can be sent by the server, using GET requests to
the app root path, followed by the filename
app.use(express.static('public'));
```



```
http://localhost:3000/images/kitten.jpg
http://localhost:3000/css/style.css
http://localhost:3000/js/app.js
http://localhost:3000/images/bg.png
http://localhost:3000/hello.html
```

Express – how does it help?

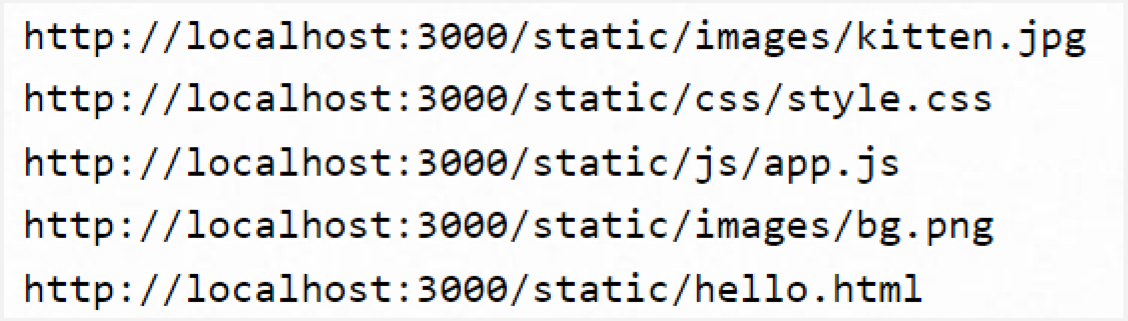
- **Example 3**: alter the type of response to HTML **static files**
  - ➢ to include serving static files like fonts, images, CSS or others with Express, one must add the Express built-in middleware **express.static**

```
express.static(root, [options])
```

  - ○ `root`: root directory from which to serve static assets

Example #02

```
// all files in directory '/public' can be sent by the server, using GET requests with
the path '/static', followed by the filename
app.use('static', express.static('public'));
```



```
http://localhost:3000/static/images/kitten.jpg
http://localhost:3000/static/css/style.css
http://localhost:3000/static/js/app.js
http://localhost:3000/static/images/bg.png
http://localhost:3000/static/hello.html
```

Express – how does it help?

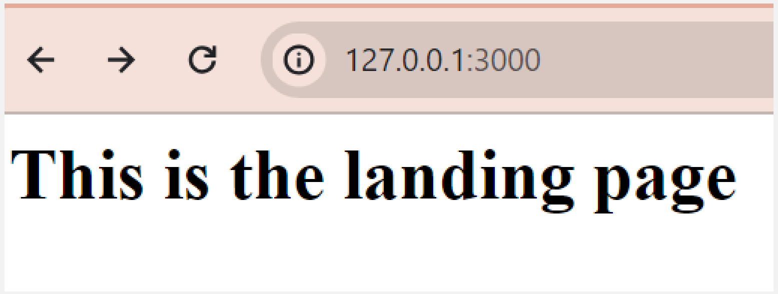
- **Example 3**: alter the type of response to HTML static files
  - path is a Node module that helps getting the specific path to a file
  - The first parameter of the `sendFile` method of Express requires to be the absolute path of a file, unless the `root` option is used
  - Remember that the **absolute path may change** on different OS

```
// import module PATH
const path = require('path');

// sends static HTML file to GET requests on the root URL
app.get('/', (req, res) => {
      //path.join(): joins all given paths
      res.sendFile(path.join(__dirname, 'index.html'));
})

// sends static HTML files from a directory named ‘public’ for GET requests on the URL
‘files’, given the relative file paths in a query string
app.get('/files', (req, res) => {
      //path.join(): joins all given paths
      res.sendFile(req.query.filename, {root: path.join(__dirname, 'public') });
})
```

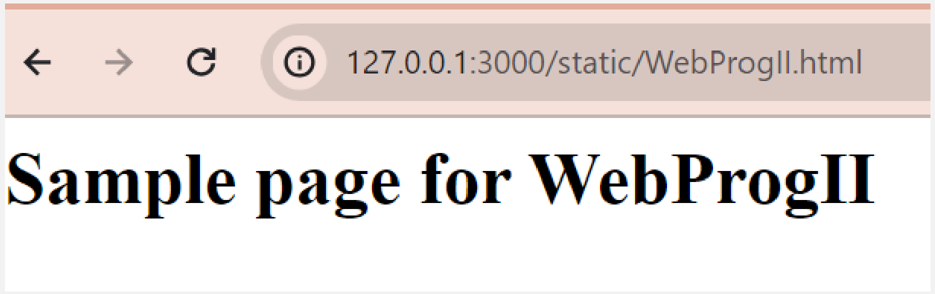
Check into your Node console the value of `__dirname`

Express – how does it help?

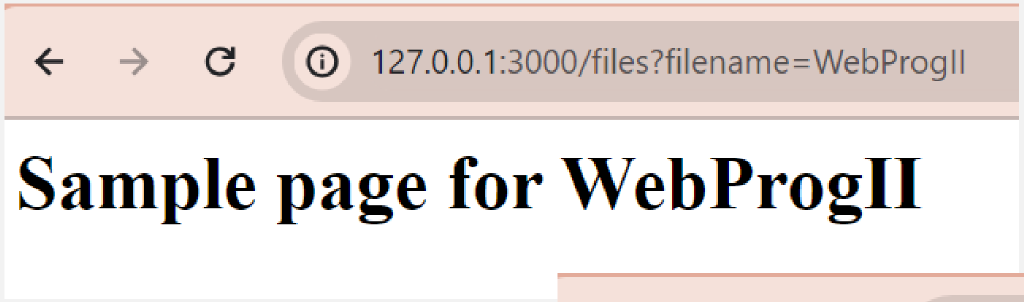
- **Example 3**: alter the type of response to HTML static files


127.0.0.1:3000

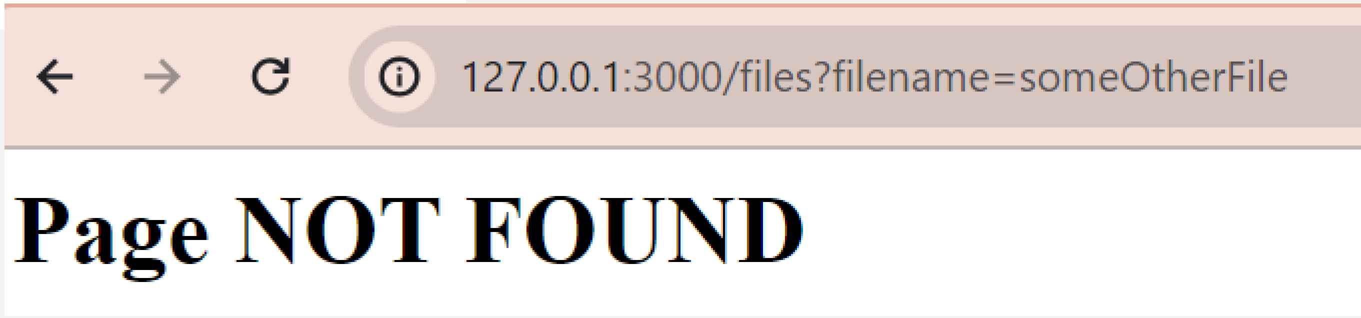
This is the landing page


127.0.0.1:3000/static/WebProgII.html

Sample page for WebProgII


127.0.0.1:3000/files?filename=WebProgII

Sample page for WebProgII


127.0.0.1:3000/files?filename=someOtherFile

Page NOT FOUND

Express – middleware

- Express is a **routing** and **middleware web framework** that has minimal functionality of its own
  - An Express application is essentially a **series of middleware function calls**
- Middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the *next* middleware function in the application's request-response cycle
  - If the current middleware function does not end the request-response cycle, it must call `next()` to pass control to the next middleware function; otherwise, the request will be left hanging
- Middleware functions can perform the following tasks:
  - Execute any code
  - Make changes to the request and the response objects
  - End the request-response cycle
  - Call the next middleware function in the stack

Express – middleware

- Bind application-level middleware to an instance of the app object by using the `app.use()` and `app.METHOD()` functions

```
// middleware function with no mount path, executed EVERYTIME the app receives a request
app.use( (req, res, next) => {
  console.log('Time:', Date.now())
  next() // pass control to the next handler
})
```

```
// middleware function mounted on /users/:id path, executed for ALL HTTP requests
app.use('/users/:id', (req, res, next) => {
  console.log('Request Type:', req.method);
  next() // pass control to the next handler
}, (req, res) => { // 2nd handler
  res.send('Finished');
})
```

on the root route (/), the application's home page

Express – middleware

- With Express one can provide multiple callback functions that behave like middleware to handle a request
  - The only exception is that these callbacks might invoke `next('route')` to bypass the remaining route callbacks
  - One can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there's no reason to proceed with the current route
  - Route handlers can be in the form of a function, an array of functions, or combinations of both

Express – middleware

```
// an array of callbacks to handle a route
let cb0 = function (req, res, next) {
  console.log('CB0')
  next()
}

let cb1 = function (req, res, next) {
  console.log('CB1')
  next()
}

let cb2 = function (req, res) {
  res.send('Hello from C!')
}

app.get('/example/c', [cb0, cb1, cb2])
```

Express – middleware

```
//1st route handler, which prints the user ID
app.get('/user/:id', (req, res, next) =>  {
  console.log('ID:', req.params.id)
  next()
}, (req, res, next) => {
  res.send('User Info')
})

//2nd route handler: WILL NEVER GET CALLED because the 1st route ends the request-response cycle
app.get('/user/:id', (req, res, next) => {
  res.send(req.params.id)
})
```

```
app.get('/user/:id', (req, res, next) => {
  //if user ID is 0, skip to the 2nd route handler
  if (req.params.id === '0') next('route')
  // otherwise pass the control to the next middleware function in this stack
  else next()
}, function (req, res, next) {
  res.send('User Info')
})

//2nd route handler which sends a special message
app.get('/user/:id', (req, res, next) => {
  res.send('special route')
})
```

Express – error-handling middleware

- Define error-handling middleware functions in the same way as other middleware functions, except with four arguments instead of three: `(err, req, res, next)`
    - ➢ One must provide four arguments to identify it as an error-handling middleware function
    - ➢ Even if it is not necessary to use the next object, one must specify it to maintain the signature; otherwise, the next object will be interpreted as regular middleware and will fail to handle errors

```
// a single callback function to handle a route
app.use( (err, req, res, next) => {
  console.error(err.stack)
  res.status(500).send('Something broke!')
})
```

    - ➢ You define error-handling middleware last, after other `app.use()` and routes calls
    - ➢ For details about error-handling middleware, see Error handling

Express – built-in middleware

- Express has the following built-in middleware functions
  - express.static: serves static assets such as HTML files, images, and so on
  - express.json: parses incoming requests with JSON payloads
  - express.urlencoded: parses incoming requests with URL-encoded payloads

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

// all request will now be able to parse JSON and process requests with Content
Header of of type application/json
app.use(express.json())

// all request will now be able to parse data from HTML forms (process request with
Content Header of type application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }))

app.post('/profile', (req, res, next) => {
  console.log(req.body) // req.body: access the body data key-value pairs
  res.json(req.body) // req.json(): send a JSON response
})
```

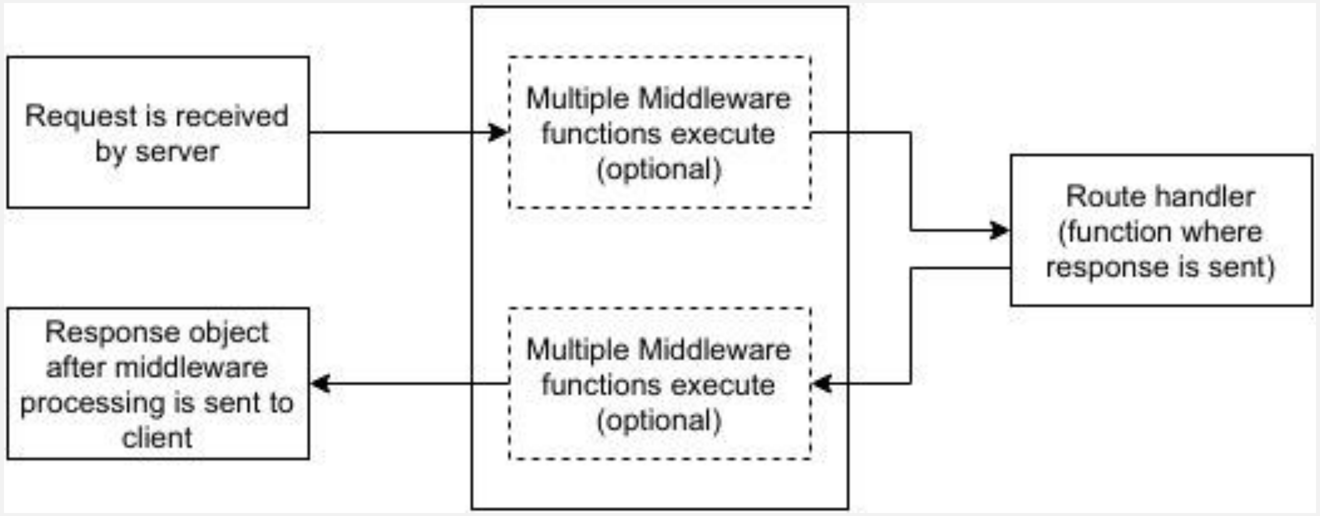
Express – third-party middleware

- Third-party middleware extends Express functionalities
- One must install the external module, load it into the app (at the application level or router level)
  - ➢ For a partial list of third-party middleware functions that are commonly used with Express, see Third-party middleware

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

// enable cross-origin resource sharing (CORS)
const cors = require("cors");

// all requests to this app, will notice a new header being returned:
Access-Control-Allow-Origin: *
app.use(cors());
```

Express – router middleware

- Defining all routes is very tedious to maintain
- One should separate the routes from the main application file, using the router middleware **Express.Router**
  - creates modular, mountable and testable route handlers
  - it allows to group the route handlers for a particular part of an application together and access them using a common route-prefix


Request is received by server
Multiple Middleware functions execute (optional)
Route handler (function where response is sent)
Multiple Middleware functions execute (optional)
Response object after middleware processing is sent to client

Express – router middleware

index.js - main app file

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

const things =
require('./routes/things.routes.js');

//now the app will answer to GET and POST
requests to route /things
app.use('/things', things);

app.listen(3000);
```

things.routes.js - file stored in folder *routes*

```
const express = require('express');
const router = express.Router();

// middleware that is specific to this router
router.use((req, res, next) => {
  console.log('Time: ', Date.now())
  next()
})
// route handlers for a GET request on route
path / (mounted on application path /things)
router.get('/', (req, res) => {
   res.send('GET route on things.');
});
// route handlers for a POST request on
route path / (mounted on application
path /things)
router.post('/', (req, res) => {
   res.send('POST route on things.');
});

//export this router
module.exports = router;
```

Express – router middleware

index.js - main app file

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

const things =
require('./routes/things.routes.js');

//now the app will answer to GET and POST
requests to route /things
app.use('/things', things);

app.listen(3000);
```

things.routes.js - file stored in folder *routes*

```
const express = require('express');
const router = express.Router();

// middleware that is specific to this router
router.use((req, res, next) => {
  console.log('Time: ', Date.now())
  next()
})
// chainable route handlers for a route path /
mounted on application path /things
router.route('/')
   .get((req, res) => {
      res.send('GET route on things.');
   })
   .post((req, res) => {
      res.send('POST route on things.');
   });

//export this router
module.exports = router;
```

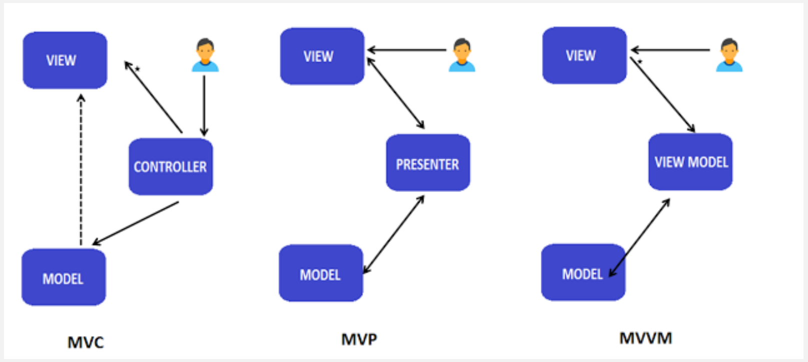
Express – exercise

- Lets create a simple MOVIES API, with data transported in JSON format, using the MVC architecture

Method	URI	Description
GET	/movies	Gets the list of all movies and their details.
GET	/movies/{id}	Gets details on a particular movie. Return 404 error if movie does not exist.
POST	/movies	Creates a new movie with the details provided in the request body. Response contains the URI for this newly created resource. Return 201 on success (with the path to access the new movie) or 400 error for incomplete or invalid movie data
PUT	/movies/{id}	Modifies a particular movie. Return 400 error for incomplete or invalid moovie data or 404 if movie is not found
DELETE	/movies/{id}	Delete a particular movie, if it exists.

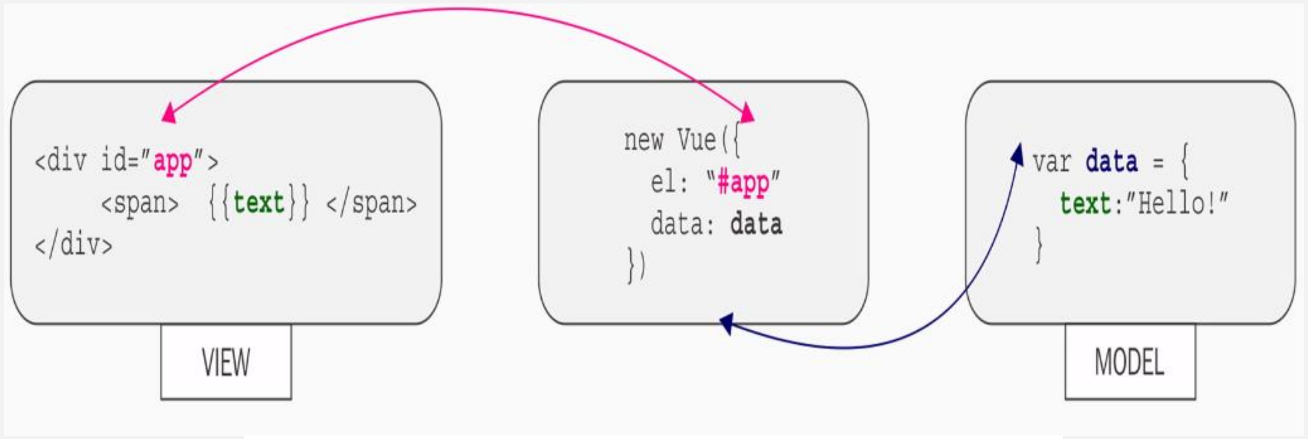
MVC architecture

- When developing of an application, it is essential to **define the platform** and how the **system components** will interact and organize
- In this sense, it is important to **create an architecture** that defines the software elements and that tells how they interact with each other
- Nowadays there are **several architectures,** and each one has a very specific function within the structure of a project



MVVM architecture

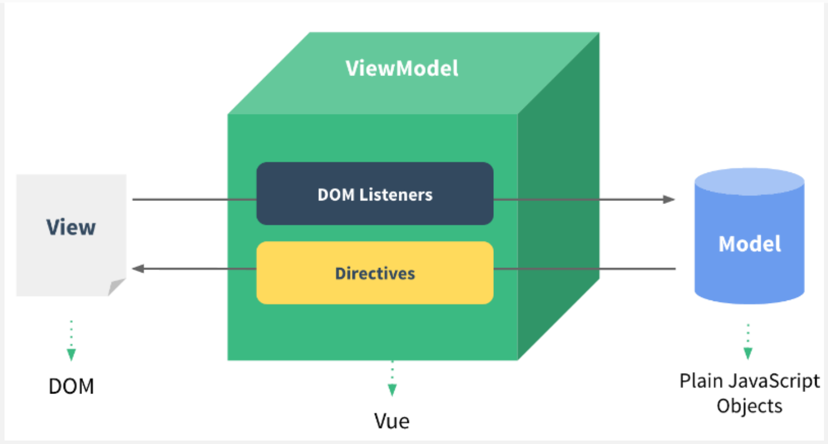
- Example: Vue framework



```
<div id="app">
    <span> {{text}} </span>
</div>
```

```
new Vue({
  el: "#app"
  data: data
})
```

```
var data = {
  text:"Hello!"
}
```



Model View Controller (MVC)

- The MVC pattern divides a system into three parts:
  - **Model**: the model can be understood as the domain layer of the application; this part contains the business logic, data persistency, etc.
  - **View**: responsible for presenting the data to the user
  - **Controller**: the controller layer processes and responds to events, receives changes to the model and updates the view layer

Model View Controller (MVC)

- **Model**
  - Model is where the application logic is
    - Will the application send an email? Validate a form? Send or receive data from DB? It does not matter!
  - The model always knows **how to get the data** to perform the most diverse tasks
  - *The model does not need to know when it should be done, nor how to show the data*

Model View Controller (MVC)

- **View**
  - View displays the data
  - View is not just HTML, but any type of data return, such as PDF, JSON, XML, the return of data from the RESTFull server, authentication tokens, among others
  - Any return of data to any interface (e.g. browser) is the responsibility of the view
  - *The view must know **how to render** the data correctly, but does not need to know how to get it or when to render it*

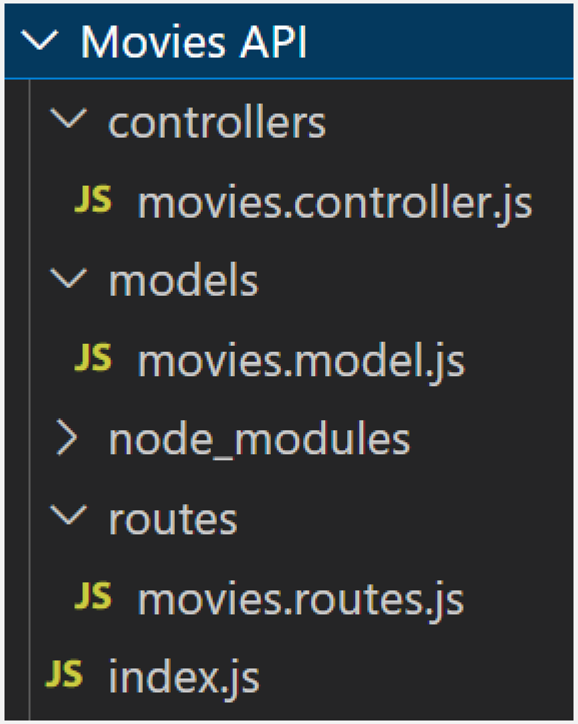
Model View Controller (MVC)

- **Controller**
  - The controller tells you when the processes should happen
  - It is used to intermediate the model and a layer view
    - For example, to get data from the model (stored in the DB) and display it in the view (in an HTML page), or take the data from a view form and send it to someone - model
  - It is also the responsibility of the controller to take care of requests and responses, and this also includes middleware
  - *The controller doesn't need to know how to get the data or how to display it, it just needs to know **when to** do it*

Express – exercise

Folder structure

- controllers folder: responsible for request data validation and for sending the API responses to clients
- models folder: stores all information about the movies
- routes folder: define routes handlers
- index.js file: sets up an Express web server



```
Movies API
  controllers
    movies.controller.js
  models
    movies.model.js
  node_modules
  routes
    movies.routes.js
  index.js
```

Express – exercise

- In the root folder, create file `index.js`:

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

const app = express();
const host = process.env.HOST ||'127.0.0.1'; const port = process.env.PORT || 8080;

app.use(express.json()); //enable parsing JSON body data

// root route -- /api/
app.get('/', function (req, res) {
    res.status(200).json({ message: 'home -- MOVIES api' });
});

// routing middleware for resource MOVIES
app.use('/movies', require('./routes/movies.routes.js'))

// handle invalid routes
app.all('*', function (req, res) {
    res.status(404).json({ message: 'WHAT???' });
})
app.listen(port, host, () => console.log(`App listening at http://${host}:${port}/`));
```

Express – exercise

- Each movie document has the following data: id, name, year and rating
- Since no database is being used, store some movies in memory by declaring an array in a file `movies.model.js` at folder `models`
  - ➢ Since no database is used, every time the server restarts, any changes to this array will vanish
  - ➢ this data will be required by the route handler, so make sure to call it

```
// movie data
let movies = [
   {id: 101, name: "Fight Club", year: 1999, rating: 8.1},
   {id: 102, name: "Inception", year: 2010, rating: 8.7},
   {id: 103, name: "The Dark Knight", year: 2008, rating: 9},
   {id: 104, name: "12 Angry Men", year: 1957, rating: 8.9}
];

//Data will go here
module.exports = movies;
```

New file called *movies.model.js*, stored in folder *models*

- ➢ A movie can be searched within the array using, for example JS array method [filter]

Express – exercise

- In the `routes` folder, create file `movies.routes.js`, to map all API routes to the corresponding controller functions

```
const express = require('express');
const router = express.Router();
// import controller middleware
const moviesController = require("../controllers/movies.controller");

router.route('/')
    .get( moviesController.findAll )
    .post( moviesController.bodyValidator, moviesController.create );

router.route('/:id')
    .get( moviesController.findOne)
    .put( moviesController.bodyValidator, moviesController.update )
    .delete( moviesController.delete );

router.all('*', (req, res) => {
    res.status(404).json({ message: 'MOVIES: what???' }); //send a predefined error message
})

//export this router
module.exports = router;
```

Express – exercise

- In the `controllers` folder, create file `movies.controllers.js`, to validate the body data of POST and PUT requests and send back a response

```
// import movies data
const movies = require("../models/movies.models");

// exports custom request payload validation middleware
exports.bodyValidator = (req, res, next) => {
   …
};

// Display list of all movies
exports.findAll = (req, res) => {
    res.json(movies);
};

// Display only 1 movies
exports.findOne = (req, res) => {
   ...
};

...
```