



★ Course information

Assignments & Practicing

- ★ Assignment HTTP+Design
- ★ Assignment JS+Node
- ★ Assignment CSS+Node
- ★ Assignment rubric
- ★ How to use VSC
- ★ Nodeschool exercises
- ★ Sample exams

Lectures

- ★ HTTP
- ★ HTML
- ★ JavaScript
- ★ Node.js I
- ★ CSS
- ★ Node.js II
- ★ Sessions et al.
- ★ Web security

The course notes can be  highlighted by holding  and selecting text. Hovering over a highlight opens a notepad. A double-click on a highlight removes it.

Delete all highlights on this page.

Advanced Node.js




At times we use  and  to make it clear whether an explanation refers to the code snippet above or below the text. The **!!** sign is added to code examples you should run yourself. When you come across a `DEBUG` box, we offer advice on how to debug your code—this may help you with your assignments. Neither `DEBUG` nor `OPTIONAL` content are exam material.

Table of Contents

- ★ ➤ Required & recommended readings and activities
- ★ ➤  Learning goals
- ★ ➤ Organization and reusability of Node.js code
 - ★ ➤ A file-based module system
 - ★ ➤ **!!** A first module example
 - ★ ➤ **!!** `require` is blocking
 - ★ ➤ **!!** `module.exports` vs. `exports`
- ★ ➤ Creating and using a (useful) module
 - ★ ➤ CommonJS vs. ECMAScript modules
- ★ ➤ Middleware in Express
 - ★ ➤ **!!** Logger example
 - ★ ➤ **!!** Authorization component example
 - ★ ➤ Components are configurable
- ★ ➤ Routing
 - ★ ➤ Routing paths and string patterns
 - ★ ➤ Routing parameters
 - ★ ➤ Organizing routes

- ★ ➤ Templating with EJS
 - ★ ➤ !! A first EJS example
 - ★ ➤ !! EJS and user-defined functions
 - ★ ➤ !! JavaScript within EJS templates
 - ★ ➤ !! Express and templates
- ★ ➤ Node.js in production
- ★ ➤ Self-check

Required & recommended readings and activities

- ★ Required readings: *none*
- ★ Recommended activities:
 - ★ An interactive ejs playground can be found ➤ [here](#).
- ★ Recommended readings:
 - ★ Chapters 10, 14 and 20 of the ➤ [Web Development with Node & Express book](#).
 - ★ To learn more about ejs, take a look at its ➤ [GitHub repository](#).
 - ★ To learn more about middleware and Express, take a look at the ➤ [Express documentation](#).
 - ★ An overview of ➤ [best practices in Node.js](#).
- ★ Relevant scientific publications:
 - ★ Fard, Amin Milani, and Ali Mesbah. ➤ [JSNose: Detecting javascript code smells](#). 13th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, 2013.
 - ★ Nasehi, Seyed Mehdi, et al. ➤ [What makes a good code example?: A study of programming Q&A in StackOverflow](#). 28th IEEE International Conference on Software Maintenance (ICSM). IEEE, 2012.

Learning goals

- ★ Organize Node.js code into modules.



- ★ Understand and employ the concept of *middleware*.
- ★ Employ routing.
- ★ Employ templating.

Organization and reusability of Node.js code

So far, we have organized all server-side code in a single file, which is only a feasible solution for small projects. In larger projects, this quickly ends in unmaintainable code, especially when working in a team.

These issues were recognized early on by the creators of Node.js. For this reason, they introduced the concept of **modules**. A Node.js module is (1) a single file or (2) a directory of files and all code contained in it.

By default, *no code in a module is accessible to other modules*. Any property or method that should be visible to other modules has to be **explicitly** marked as such - you will learn shortly how exactly. Node.js modules can be published to ↗ npmjs.com, the most important portal to discover and share modules with other developers. This is ↗ [Express' page on npm](#):



Screenshot taken September 19, 2020.

👉 You see here that modules often depend on a number of other modules (in this case: 30 dependencies). As Express is a very popular module, it is listed as dependency in more than 46,000 other modules.

You already know how to install modules, e.g. `npm install winston` installs one of the more popular [Node logging libraries](#). You can also use the command line to search for modules to install, e.g. `npm search winston`.

While it is beyond the scope of this course to dive into the details of the npm registry, it should be mentioned that it is not without issues; the story of how 17 lines of code—a single npm module—nearly broke much of the modern web for half a day can be found [here](#).

A file-based module system



In Node.js each file is its own module. This means that the code we write in a file does not pollute the *global namespace*. In Node.js we get this setup “for free”. When we write client-side JavaScript, we have to work hard to achieve the same effect (recall the module pattern covered in the JavaScript lecture).

The module system works as follows: each Node.js file can access its so-called **module definition** through the `module` object. The `module` object is your entry point to modularize your code. To make something available from a module to the outside world, `module.exports` or `exports` is used as we will see in a moment. The `module` object looks as follows (depending on the Node version and underlying operating system the property values will vary) 🙌:

Copy me

```
1  Module {
2    id: '.',
3    path: '/Users/Node/Web-Teaching',
4    exports: {},
5    parent: null,
6    filename: '/Users/Node/Web-Teaching/myfile.js',
7    loaded: false,
8    children: [],
9    paths: [
10     '/Users/Node/GitHub/Web-Teaching/node_modules',
11     '/Users/Node/GitHub/node_modules',
12     '/Users/Node/node_modules',
13     '/Users/node_modules',
14     '/node_modules'
15   ]
16 }
```

To see for yourself how the `module` object looks on your machine you can do one of two things:

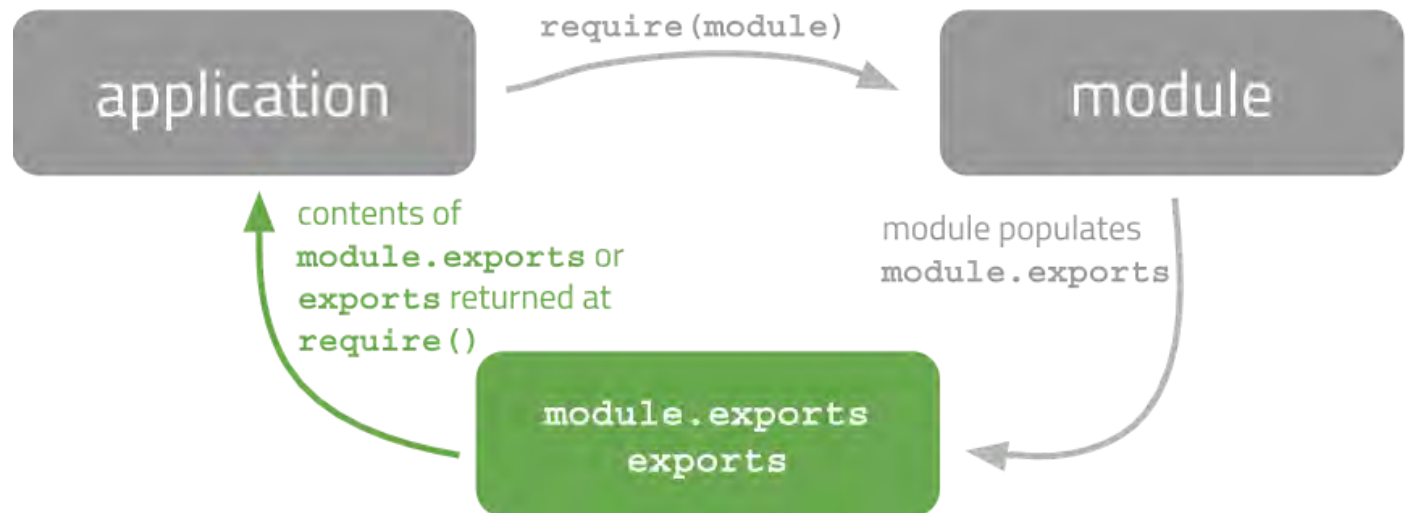


- i. Create a Node.js script containing only the line `console.log(module);` and run it.
- ii. Start the node REPL (just type `node` into the terminal) and then type `console.log(module);` .

We see that in our module object above 🙌 nothing is currently being *exported* as the `exports` property has an empty object (`{}`) as value.

Once you have defined your own module, the globally available `require` function is used to import a module. At this stage, you should recognize that you have been using Node.js modules since your first attempts with Node.js.

Here is a graphical overview of the connection between `require` and `module.exports` :



🙌 An application uses the `require` function to import module code. The module itself populates the `module.exports` variable to make certain parts of the code base in the module available to the outside world. Whatever was assigned to `module.exports` (or `exports` —we will get to it) is then returned to the application when the application calls `require()` .

!! A first module example



Let's consider files `foo.js` 📌:

Copy me

```
1  const fooA = 1;
2  module.exports = "Hello!";
3  module.exports = function () {
4    console.log("Hi from foo!");
5  };
```

and `bar.js` 📌:

Copy me

```
1  const foo = require("./foo");
2  foo(); //CASE 1: Hi from foo!
3  require("./foo")(); //CASE 2: Hi from foo!
4  console.log(foo); //CASE 3: [Function]
5  console.log(foo.toString()); //CASE 4: function () {console.log("Hi fro
6  console.log(fooA); //CASE 5: ReferenceError (you will have to remove th
7  console.log(module.exports); //CASE 6: {}
```

📌 Here, `foo.js` is our `foo` module and `bar.js` is our application that imports the module to make use of the module's functionality (you can run the script as usual with `node bar.js`). In the `foo` module, we define a variable `fooA` in line 1. In lines 2 and 3, you can see how a module uses `module.exports` to make parts of its code available: in line 2, we assign a string to `module.exports`, in line 3 we make a new assignment to `module.exports` and define a function that prints out *Hi from foo!* on the console. Which of the two assignments will our application `bar` end up with? In `bar.js` the first line calls the `require()` function and assigns the returned value to the variable `foo`. In line 1 we used as argument `./foo` instead of `./foo.js` - you can use both variants. The dot-slash indicates that `foo.js` resides in the current directory.



Node.js runs the referenced JavaScript file 🙌 (here: `foo.js`) in a **new scope** and **returns the final value** of `module.exports` . What then is the final value after executing `foo.js` ? It is the function we defined in line 3.

🙌 As you can see in lines 2 and beyond of `bar.js` there are several ways to access whatever `require` returned:

- ✧ **CASE 1:** We can call the returned function and this results in *Hi from foo!* as you would expect.
- ✧ **CASE 2:** We can also combine lines 1 and 2 into a single line with the same result.
- ✧ **CASE 3:** If we print out the variable `foo` , we learn that it is a function.
- ✧ **CASE 4:** Using the `toString()` function prints out the content of the function.
- ✧ **CASE 5:** Next, we try to access `fooA` - a variable defined in `foo.js` . Remember that Node.js runs each file in a new scope and only what is assigned to `module.exports` is available. Accordingly, `fooA` is not available in `bar.js` and we end up with a reference error. Note, that Visual Studio Code flags up this error (`fooA` is not defined) already at the code writing stage.
- ✧ **CASE 6:** Finally, we can also look at the `module.exports` variable of `bar.js` - this is always available to a file in Node.js. In `bar.js` we have not assigned anything to `module.exports` and thus it is an empty object.

!! require is blocking

This module setup also explains why `require` is **blocking**: once a call to `require()` is made, the referenced file's code is executed and only once that is done, does `require()` return. This is in contrast to the usual *asynchronous* nature of Node.js functions.

Let's now consider what happens if a module is imported more than once. Keep `foo.js` intact; `bar.js` now becomes 🙌:


```

1  const t1 = process.hrtime()[1]; //returns an array with [seconds, nanos
2  const foo1 = require("./foo");
3  console.log(process.hrtime()[1] - t1); //303914
4
5  const t2 = process.hrtime()[1];
6  const foo2 = require("./foo");
7  console.log(process.hrtime()[1] - t2); //35012

```

Run `node bar.js` and observe the execution times logged to the terminal. 🙌 Here, we execute `require('./foo')` twice and log both times the time it takes for `require` to return. The first time the line `require(foo.js)` is executed, the file `foo.js` is read from disk (this takes some time). In subsequent calls to `require(foo.js)`, however, the **in-memory object is returned**. Thus, `module.exports` is **cached**.

🙌 We here resort to using `process.hrtime()` which returns an array whose first value is the time in seconds and the second is the time in nanoseconds relative to “*an arbitrary time in the past*” (🔗 [Node documentation](#)). While it is not possible to compute an absolute time in this manner, we can accurately measure the duration of code as seen in the above example. Depending on your machine, the reported nanoseconds intervals will differ, though the first `require()` statement will always take significantly (10x-100x) longer than the second `require()` statement.

!! module.exports vs. exports

Every Node.js file has access to `module.exports`. If a file does not assign anything to it, it will be an empty object, but it is **always** present. Instead of `module.exports` we can use `exports` as `exports` is an **alias** of `module.exports`. This means that the following two code snippets are equivalent 🙌:

Copy me

```

1  //SNIPPET 1
2  module.exports.foo = function () {

```



```

3     console.log("foo called");
4 };
5
6 module.exports.bar = function () {
7     console.log("bar called");
8 };

```

Copy me

```

1 //SNIPPET 2
2 exports.foo = function () {
3     console.log("foo called");
4 };
5
6 exports.bar = function () {
7     console.log("bar called");
8 };

```

👉 In the first snippet, we use `module.exports` to make two functions (`foo` and `bar`) accessible to the outside world. In the second snippet, we use `exports` to do exactly the same. Note that in these two examples, **we do not assign something to `exports` directly**, i.e. we do not write `exports = function`. This is in fact **not possible** as `exports` is only a reference (a short hand if you will) to `module.exports` : if you directly assign a function or object to `exports` , then its reference to `module.exports` will be **broken**. You can only **assign directly** to `module.exports` , for instance, if you only want to make a single function accessible.

Creating and using a (useful) module

In the example above 👉, `foo.js` is a module we created. Not a very sensible one, but still, it is a module. Modules can be either:



- ✪ a **single file**, or,
- ✪ a **directory of files**, one of which is `index.js`.

A module can contain other modules (that's what `require` is for) and should have a specific purpose. For instance, we can create a *grade rounding module* whose functionality is the rounding of grades in the Dutch grading system. Any argument that is not a number between 1 and 10 is rejected 🙅:

[Copy me](#)

```
1  /* not exposed */
2  const errorString = "Grades must be a number between 1 and 10.";
3
4  function roundGradeUp(grade) {
5      if (isValidNumber(grade) == false) {
6          throw errorString;
7      }
8      return Math.ceil(grade) > 10 ? 10 : Math.ceil(grade); //max. is alwa
9  }
10
11 function isValidNumber(grade) {
12     if (
13         isNaN(grade) == true ||
14         grade < exports.minGrade ||
15         grade > exports.maxGrade
16     ) {
17         return false;
18     }
19     return true;
20 }
21
22 /* exposed */
23 exports.maxGrade = 10;
24 exports.minGrade = 1;
25 exports.roundGradeUp = roundGradeUp;
```



```
26 exports.roundGradeDown = function (grade) {
27   if (isValidNumber(grade) == false) {
28     throw errorString;
29   }
30   return Math.floor(grade);
31 };
```

We can use the grading module in an Express application as follows 📌:

Copy me

```
1  const express = require("express");
2  const url = require("url");
3  const http = require("http");
4  const grading = require("./grades"); // our module file resides in the
5
6  const port = process.argv[2];
7  const app = express();
8  http.createServer(app).listen(port);
9
10 app.get("/round", function (req, res) {
11   const query = url.parse(req.url, true).query;
12   const grade = query["grade"] != undefined ? query["grade"] : "0";
13
14   //accessing module functions
15   res.send(
16     "UP: " +
17       grading.roundGradeUp(grade) +
18       ", DOWN: " +
19       grading.roundGradeDown(grade)
20   );
21 });
```



Assuming the Node script is started on `localhost` and port `3000`, we can then test our application with several valid and invalid queries:

- ★ `http://localhost:3000/round?grade=2.1a`
- ★ `http://localhost:3000/round?grade=2.1`
- ★ `http://localhost:3000/round?grade=`
- ★ `http://localhost:3000/round?grade=10`

OPTIONAL

CommonJS vs. ECMAScript modules

When Node.js was created, there was no standardized format for modules and Node.js went with the format introduced above—this is known as the [CommonJS](#) module formatting.

As modules are important to enable code encapsulation, improve code maintenance, etc. eventually modules were added to the ECMAScript specification: not Node.js' choice of module formatting though; **ECMAScript modules** were introduced in ES6. Those files are usually recognizable by the file ending `*.mjs` (though [MDN argues against it](#)). Node.js supports both module systems; in this class we stick to the CommonJS formatting which remains the more common one in Node.js.

All modern browsers [support ECMAScript modules](#). We have not introduced them in the client-side JavaScript lecture due to a lack of time. You can read more about them on [MDN](#).

Middleware in Express

Middleware components are small, self-contained and reusable code pieces across applications. Imagine you have written an Express application with tens of different routes



and now decide to log every single HTTP request coming in. You could add 2-3 lines of code to every route to achieve this logging (which leads to code duplication, generally a bad idea) or you write a middleware logging component that gets called before any other route is called (code is written only once!). How exactly the latter works in Express is discussed now.

Middleware components have **three parameters**:

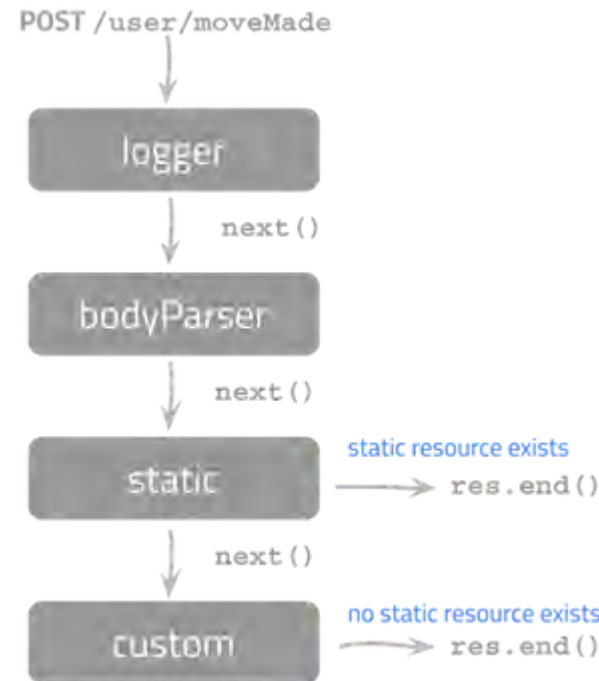
- ✧ an HTTP request object;
- ✧ an HTTP response object;
- ✧ an optional callback function— `next()` —to indicate that the component is finished; the **dispatcher** which orchestrates the order of middleware components can now move on to the next component.

Middleware components have a number of abilities. They can:

- ✧ execute code;
- ✧ change the request and response objects;
- ✧ end the request-response cycle;
- ✧ call the next middleware function in the middleware stack.

As a concrete example, imagine an Express application with a POST route `/user/moveMade` 📌:





👉 The first middleware to be called is the logger component, followed by the bodyParser component which parses the HTTP request body; next, the static component is probed (is there a static resource that should be served to the user?) and if no static resource exists, a final custom component is called. **When an HTTP response is sent (`res.end()`), the middleware call chain is complete.** Once the call chain is complete, it cannot be activated again. It is thus not possible to send two or more responses for a given HTTP request: for each request, at most one response can be generated.

!! Logger example

Our goal is to create a logger that records every single HTTP request made to our application as well as the URL of the request. We need to write a function that accepts the HTTP request and response objects as arguments and `next` as callback function. Here, we write two functions to showcase the use of several middleware components 👉:

Copy me

```
1  const express = require("express");
2
```

```

3 //a middleware logger component
4 function logger(request, response, next) {
5   console.log(`${new Date()}`, ${request.method}, ${request.url}`);
6   next(); //control shifts to next middleware function
7 }
8
9 //a middleware delimiter component
10 function delimiter(request, response, next) {
11   console.log("-----");
12   next();
13 }
14
15 const app = express();
16 app.use(logger); //register middleware component
17 app.use(delimiter);
18 app.listen(3001);

```

If you start the script and then make the following HTTP requests:

- ★ http://localhost:3001/first
- ★ http://localhost:3001/second
- ★ http://localhost:3001/

your output on the terminal will look something like this:

```

1 2020-01-14T19:17:16.577Z      GET    /first
2 -----
3 2020-01-14T19:17:19.111Z      GET    /second
4 -----
5 2020-01-14T19:17:21.159Z      GET    /
6 -----

```



👉 Importantly, `next()` enables us to move on to the next middleware component while `app.use(...)` registers the middleware component with the dispatcher. Try out this code for yourself and see what happens if:

- ✧ `app.use` is removed;
- ✧ the order of the middleware components is switched, i.e. we first add `app.use(delimiter)` and then `app.use(logger)` ;
- ✧ in one or both of the middleware components the `next()` call is removed.

You will observe different behaviors of the application that make clear how the middleware components interact with each other and how they should be used in an Express application.

In the example above we did not actually sent an HTTP response back, but you know how to write such a code snippet yourself. Up to this point, none of our routes had contained `next` for a simple reason: all our routes ended with an HTTP response being sent and this completes the request-response cycle; in this case there is no need for a `next()` call.

!! Authorization component example

In the ↗ [node-component-ex](#) example application, we add an authorization component to a wishlist application back-end: only clients with the **correct username and password** (i.e. authorized users) should be able to receive their wishlist when requesting it. We achieve this by adding a middleware component that is activated for every single HTTP request and determines:

- ✧ whether the HTTP request contains an authorization header (if not, access is denied);
- ✧ and whether the provided username and password combination is the correct one.

Before we dive into the code details, install and start the server as explained ↗ [here](#). Take a look at `app.js` before proceeding.

Once the server is started, open another terminal and use ↗ [curl](#), a command line tool that provides us with a convenient way to include username and password as you will see:



- ★ Request the list of wishes without authorization, i.e. `curl -w "\n" http://localhost:3000/wishlist` - you should see an `Unauthorized access error`. The `-w "\n"` option is not strictly necessary; it adds a newline after a completed transfer to make the console output a bit more readable.
- ★ Request the list of wishes with the correct username and password (as hard-coded in our demonstration code): `curl -w "\n" --user user:password http://localhost:3000/wishlist`. The option `--user` allows us to specify the username and password to use for authentication in the `[USER]:[PASSWORD]` format.
- ★ Request the list of wishes with an incorrect username/password combination: `curl -w "\n" --user test:test http://localhost:3000/wishlist`. You should receive a `Wrong username/password combination error`.
- ★ Add a wish to the wishlist: `curl -w "\n" --user user:password 'http://localhost:3000/addWish?type=board%20game&name=Treasure%20Hunt&priority=low'` (note that whitespaces are replaced by `%20` in URLs and the URL has to appear within quotes due to the special characters `&` in it). Did it work? If so, another execution of `curl -w "\n" --user user:password http://localhost:3000/wishlist` should result in a wishlist with now three wishes.

Let's take a look at what your terminal session should look like:





node-component-ex and curl .

Having found out how the code *behaves*, let us look at the authorization component. We here define it as an anonymous function as argument to `app.use` :

Copy me

```
1  app.use(function (req, res, next) {
2    var auth = req.headers.authorization;
3    if (!auth) {
4      return next(new Error("Unauthorized access!"));
5    }
6
7    //extract username and password
8    var parts = auth.split(" ");
9    var buf = new Buffer(parts[1], "base64");
10   var login = buf.toString().split(":");
11   var user = login[0];
12   var password = login[1];
13
14   //compare to 'correct' username/password combination
```



```
15 //hardcoded for demonstration purposes
16 if (user === "user" && password === "password") {
17     next();
18 } else {
19     return next(new Error("Wrong username/password combination!"));
20 }
21 });
```

👉 This code snippet first determines whether an authorization header was included in the HTTP request (accessible at `req.headers.authorization`). If no header was sent, we pass an error to the `next()` function, for Express to catch and process, i.e. sending the appropriate HTTP response. If an authorization header is present, we now extract the username and password (it is base64 encoded!) and determine whether they match `user` and `password` respectively. If they match, `next()` is called and the next middleware component processes the request, which in our `app.js` file is `app.get("/wishlist",...)`.

Components are configurable

One of the design goals of middleware is **reusability across applications**: once we define a logger or an authorization component, we should be able to use it in a wide range of applications without additional engineering effort. Reusable code usually has parameters that can be set. To make this happen, we can wrap the original middleware function in a *setup function* which takes the function parameters as input 👉:

Copy me

```
1 function setup(options) {
2     // setup logic
3     return function (req, res, next) {
4         // middleware logic
5     };
}
```



```
6   }  
7   app.use(setup({ param1: "value1" }));
```

Routing

Routing is the mechanism by which requests are routed to the code that handles them. The routes are **specified by a URL and HTTP method** (most often `GET` or `POST`). You have employed routes already—every time you wrote `app.get()` you specified a so-called **route handler** for HTTP method `GET` and wrote code that should be executed when that route (or URL) is called.

This routing paradigm is a significant departure from the past, where **file-based** routing was commonly employed. In file-based routing, we access files on the server by their actual name, e.g. if you have a web application with your contact details, you typically would write those details in a file `contact.html` and a client would access that information through a URL that ends in `contact.html`. Modern web applications are not based on file-based routing, as is evident by the fact URLs these days do not contain file endings (such as `.html` or `.asp`) anymore.

In terms of routes, we distinguish between request **types** (`GET /user` differs from `POST /user`) and request **routes** (`GET /user` differs from `GET /users`).

Route handlers are middleware. So far, we have not introduced routes that include `next` as third argument, but since they *are* middleware, we can indeed add `next` as third argument.

Let's look at an example where this makes sense 📌:

Copy me

```
1   //clients request their wishlists  
2   app.get("/wishlist", function (req, res, next) {  
3       //hardcoded "A-B" testing
```



```

4      if (Math.random() < 0.5) {
5          return next();
6      }
7      console.log("Wishlist in schema A returned");
8      res.json(wishlist_A);
9  });
10
11  app.get("/wishlist", function (req, res, next) {
12      console.log("Wishlist in schema B returned");
13      res.json(wishlist_B);
14  });

```

👉 We define two route handlers for the same route `/wishlist`. Both anonymous functions passed as arguments to `app.get()` include the `next` argument. The first route handler generates a random number between 0 and 1 and if that generated number is below 0.5, it calls `next()` in the return statement. If the generated number is ≥ 0.5 , `next()` is not called, and instead a response is sent to the client making the request. If `next` was used, the dispatcher will move on to the second route handler and here, we do not call `next`, but instead send a response to the client. What we have done here is to hardcode so-called *A/B testing*. Imagine you have an application and two data schemas and you aim to learn which schema your users prefer. Half of the clients making requests will receive schema A and half will receive schema B.

We can also provide multiple handlers in a single `app.get()` call 👉:

Copy me

```

1  //A-B-C testing
2  app.get(
3      "/wishlist",
4      function (req, res, next) {
5          if (Math.random() > 0.33) {
6              return next();

```



```

7      }
8      console.log("Wishlist in schema A returned");
9      res.json(wishlist_A);
10     },
11     function (req, res, next) {
12         if (Math.random() < 0.5) {
13             return next();
14         }
15         console.log("Wishlist in schema B returned");
16         res.json(wishlist_B);
17     },
18     function (req, res) {
19         console.log("Wishlist in schema C returned");
20         res.json(wishlist_C);
21     }
22 );

```

👉 This code snippet contains three handlers - and each handler will be used for about one third of all clients requesting `/wishlist`. While this may not seem particularly useful at first, it allows you to create generic functions that can be used in any of your routes, by dropping them into the list of functions passed into `app.get()`. What is important to understand *when* to call `next` and *why* in this setting we have to use a `return` statement—without it, the function’s code would be continued to be executed.

Routing paths and string patterns

When we specify a path (like `/wishlist`) in a route, the path is eventually converted into a **regular expression** (short: regex) by Express. Regular expressions are patterns to match character combinations in strings. They are very powerful and allow us to specify **matching patterns** instead of hard-coding all potential routes. For example, we may want to allow users to access wishlists via a number of similar looking routes (such as `/wishlist`, `/whishlist`, `/wishlists`). Instead of duplicating code three times for three



routes, we can employ a regular expression to capture all of those similarly looking routes in one expression.

Express distinguishes three different types of route paths:

- i. strings;
- ii. string patterns; and
- iii. regular expressions.

So far, we have employed just strings to set route paths. **String patterns** are routes defined with strings and a subset of the standard regex meta-characters, namely: `+` `?` `*` `()` `[]` `.`

Regular expressions contain the full range of common regex functionalities (routes defined through regular expressions are enclosed in `/ /` instead of `' '`), allowing you to create arbitrarily complex patterns. If you are curious how complex regular expressions can become, take a look at the size of regular expressions to [validate email addresses](#).

Let's zoom in on **string patterns** (the full regex functionalities are beyond the scope of this class). Express' **string pattern meta-characters** have the following interpretations:

Character	Description	Regex	Matched expressions
<code>+</code>	one or more occurrences	<code>ab+cd</code>	<code>abcd</code> , <code>abbc</code> d, ...
<code>?</code>	zero or one occurrence	<code>ab?cd</code>	<code>acd</code> , <code>abcd</code>
<code>*</code>	zero or more occurrences of any char (wildcard)	<code>ab*cd</code>	<code>abcd</code> , <code>ab1234cd</code> , ...
<code>[...]</code>	match anything inside for one character position	<code>ab[cd]?e</code>	<code>abe</code> , <code>abce</code> , <code>abde</code>
<code>(...)</code>	boundaries	<code>ab(cd)?e</code>	<code>abe</code> , <code>abcde</code>



It is important to realize that the use of `*` in Express' string patterns is somewhat unique. In most other languages/frameworks, whenever `*` is mentioned in relation to regular expressions, it refers to zero or more occurrences of the preceding element. In Express' string patterns, `*` is a wildcard.

These meta-characters can be combined as seen here 🙌:

Copy me

```
1  app.get('/user(name)?s+', function(req,res){
2      res.send(...)
3  });
```

🙌 The string pattern can be deciphered by parsing the pattern from left to right. It matches all routes that:

- ★ start with `user` ,
- ★ are followed by `name` or `"`, and
- ★ end with one or more occurrences of `s` .

And thus `/user` or `/names` do *not* match this pattern, but `/users` or `/usernames` do indeed match.

If you want to play around with your own string patterns, we have a small demo application for this purpose 🙌. You find the installation instructions ➦ [here](#).





Express string patterns demo application.

Routing parameters

Apart from regular expressions, routing parameters can be employed to enable **variable input** as part of the route. Consider the following code snippet 📌:

Copy me

```
1  const express = require("express");
2  const app = express();
3
4  const wishlistPriorities = {
5    high: ["Wingspan", "Settlers of Catan", "Azul"],
6    medium: ["Munchkin"],
7    low: ["Uno", "Scrabble"],
8  };
9
10 app.get("/wishlist/:priority", function (req, res, next) {
11   let list = wishlistPriorities[req.params.priority];
12   if (!list) {
```



```

13         return next();
14     }
15     res.send(list);
16 });
17
18 app.get("*", function (req, res) {
19     res.send("No wishlist to return");
20 });
21
22 app.listen(3000);

```

👉 We have defined an object `wishlistPriorities` which contains `high`, `medium` and `low` priority wishes. We can hardcode routes, for example `/wishlist/high` to return only the high priority wishes, `/wishlist/medium` to return the medium priority wishes and `/wishlist/low` to return the low priority wishes. This is not a maintainable solution though (just think about objects with hundreds of properties). Instead, we create a single route that, dependent on a **routing parameter**, serves different wishlists. This is achieved in the code snippet shown here. The routing parameter `priority` (indicated with a starting colon `:`) will match any string that does **not** contain a slash. The routing parameter is available to us in the `req.params` object. Since the route parameter is called `priority` (what name we give this parameter is up to us), we access it as `req.params.priority`. The code snippet checks whether the route parameter matches a property of the `wishlistPriorities` object and if it does, the corresponding wishlist is returned in an HTTP response. If the parameter does not match any property of the `wishlistPriorities` object, we make a call to `next` and move on the next route handler.

Routing parameters can have various levels of nesting. Route paths can also contain optional parameters—those have an `?` appended at the end of the routing parameter name 👉:

```
1  const express = require("express");
2  const app = express();
3
4  const wishlistPriorities = {
5    high: {
6      partyGame: [],
7      gameNightGame: ["Wingspan", "Settlers of Catan", "Azul"],
8    },
9    medium: {
10     partyGame: ["Munchkin"],
11     gameNightGame: [],
12   },
13   low: {
14     partyGame: ["Uno"],
15     gameNightGame: ["Scrabble"],
16   },
17 };
18
19 //the second route parameter is optional
20 app.get("/wishlist/:priority/:gameType?", function (req, res, next) {
21   const prio = req.params.priority;
22   const type = req.params.gameType;
23
24   let list;
25
26   if(list = wishlistPriorities[req.params.priority][req.params.gameType])
27     res.send(list);
28 }
29 else if(list = wishlistPriorities[req.params.priority]){
30   res.send(list);
31 }
32 else {
33   return next(); // will eventually fall through to 404
34 }
```



```
35     });  
36  
37     app.listen(3000);
```

👉 Here, we do not only use the priorities for our wishlist, but also partition them according to whether the wanted games are party games or rather something for a long game night. The route handler now contains two routing parameters, `:priority` and `:gameType`. The `?` at the end of `:gameType` indicates that this is an optional parameter. Both parameters are accessible through the HTTP request object. We use the two parameters to access the contents of the `wishlistPriorities` object. If our corresponding Node.js script runs on our own machine, we can access high priority games for a game night via `http://localhost:3002/wishlist/high/gameNightGame`. If the second route parameter is not provided, we return the wishlist according to the priority level only, e.g. `http://localhost:3002/wishlist/high`. Otherwise, we call `next()`.

DEBUG

Organizing routes

Lastly, a word on how to organize your routes. Adding routes to the main application file becomes unwieldy as the codebase grows. Based on the knowledge of this lecture, you can move routes into a separate module. All you need to do is to pass the `app` instance into the module (here: `routes.js`) as an argument 👉:

Copy me

```
1  /* routes.js */  
2  module.exports = function(app){  
3  
4      /* Route 1 */  
5      app.get('/', function(req,res){  
6          res.send(...);  
7      })
```



```
8
9      /* Route 2, ... */
10     };
```

Copy me

```
1  /* app.js */
2  //...
3  require("./routes.js")(app);
4  //...
```

👉 `routes.js` is a route module in which we assign a function to `module.exports` which contains the routes. In `app.js` we add the routes to our application through the `require` function and passing `app` in as an argument.

Templating with EJS

When we started our journey with Node.js and Express, we discussed that writing HTML in this manner (i.e. as part of our server-side script) 👉:

Copy me

```
1  const express = require("express");
2  const url = require("url");
3  const http = require("http");
4
5  const port = process.argv[2];
6  const app = express();
7  http.createServer(app).listen(port);
8
9  const htmlPrefix = "<html><head></head><body><h2>";
```



```

10  const htmlSuffix = "</h2></body></html>";
11  app.get("/greetme", function (req, res) {
12    let query = url.parse(req.url, true).query;
13    let name = query["name"] !== undefined ? query["name"] : "Anonymous";
14    res.send(`${htmlPrefix}Greetings ${name}!${htmlSuffix}`);
15  });
16
17  app.get("*", function (req, res) {
18    res.send(`${htmlPrefix}Goodbye!${htmlSuffix}`);
19  });

```

is a poor choice, as the code quickly becomes unmaintainable, hard to debug and generally a pain to work with.

One approach to solve this problem is the use of **Ajax**: the HTML code is *blank* in the sense that it does not contain any user-specific data. The HTML and JavaScript (and other resources) are sent to the client and the client makes an Ajax request to retrieve the user-specific data from the server-side.

An alternative to Ajax is **templating**. With **templating**, we are able to **directly send HTML with user-specific data to the client** and thus remove the extra request-response cycle that Ajax requires:



With templates, our goal is to write as little HTML by hand as possible. Instead:

- ✧ we create a **HTML template** void of any data,
- ✧ add data, and
- ✧ from template+data generate a rendered HTML view.




This approach keeps the code clean and separates the coding logic from the presentation markup. Templates fit naturally into the *Model-View-Controller* paradigm which is designed to keep logic, data and presentation separate.

This concept exists in several languages and even for Node.js alone, several template engines exist. In this course, we teach the basics of **EJS**—*Embedded JavaScript*—a relatively straightforward template engine and language.

!! A first EJS example

Let's take a first look at EJS. For this exercise, we will use Node's **REPL** (*Read-Eval-Print Loop*). It is the **Node.js shell**; any valid JavaScript which can be written in a script can be passed to the REPL as well. It is useful for experimenting with Node.js, and playing around with some of JavaScript's more eccentric behaviors.

To start the REPL, type `node` in the terminal. The Node shell becomes available, indicated by the `>` prompt. Start your REPL and try a few things (declare a variable, define a function, etc.) by typing into the shell, ending each line with `Enter`. You should see something like this .



👉 If you want to avoid the constant `undefined` messages on the REPL (after every executed command, the REPL prints out the return value—and it prints `undefined` if there is no return value), start the REPL as follows:

```
1 node -e "require('repl').start({ignoreUndefined:true})"
```

Back to EJS. Before we can use the EJS module, we need to install it as it is not part of core Node.js modules. You already know how to do this: `npm install ejs`. Then, you can start the REPL and type out the following EJS snippet 👉:

Copy me

```
1 const ejs = require("ejs");
2 const template = "<%= message %>"; //<%= outputs the value into the tem
3 const context = { message: "Hello CSE1500!" };
4 console.log(ejs.render(template, context));
```

👉 Here, we first make the EJS object available via `require()`. Next, we define our template string. In this template we aim to replace the message with the actual data. Our `context` variable holds an object with a property `message` and value `Hello CSE1500!`. Lastly, we have to bring the template and the data together by calling `ejs.render()`. The output will be the **rendered view**. The template contains `<%=`, a so-called *scriptlet tag* to indicate the start of an element to be replaced with data as well as an ending tag `%>`.

There are two types of scriptlet tags that **output values**:

- ✪ `<%= ... %>` output the value into the template in **HTML escaped** form. This means that the characters that are indicating the start/end of markup sequences (such as `<script>` and `</script>`) are converted in such a way that they are rendered as content instead of being interpreted as markup.



- ★ <%- ... %> output the value into the template in **unescaped** form. This means that a value such as <script> remains as-is. This enables cross-site scripting attacks, which we will discuss in the [security lecture](#).

In order to see the difference between the two types of tags, go back to Node's REPL and try out the following code snippet 👉:

Copy me

```
1  const ejs = require("ejs");
2  const unescapedTemplate = "<%- message %>";
3  const escapedTemplate = "<%= message %>";
4  const context = { message: "<script>alert('Hello CSE1500!');</script>"
5
6  console.log("[UNESCAPED] " + ejs.render(unescapedTemplate, context));
7  console.log("[ESCAPED] " + ejs.render(escapedTemplate, context));
```

You should see the following:



Node REPL and (un)escaped EJS templates.



👉 Here, we observe that the special characters in HTML are replaced by so-called **HTML entities**, each one being a string starting with `&` and ending with `;`. The opening tag bracket `<` becomes `<`, a single quotation mark `'` becomes `'`. The browser's rendering engine *renders* these HTML entities as special characters.

Try it out for yourself! Save the following snippet in a `*.html` file and open it in your browser 👉:

Copy me

```
1  <html>
2    <head></head>
3    <body>
4      &lt;script&gt;alert(&#39;Hello CSE1500!&#39;);&lt;/script&gt;
5    </body>
6  </html>
```

In contrast, the un-escaped variant produces `<script>alert('Hello CSE1500!');`
`</script>` as output. In the latter case, this is code that the browser will execute. Once more, try it out yourself by replacing the escaped output in the HTML document above with the unescaped output. Loading the page should now result in a popup appearing in the browser.

!! EJS and user-defined functions

In order to make templates maintainable, it is possible to provide user-defined functions to a template as follows 👉:

Copy me

```
1  const ejs = require("ejs");
2  const people = ["wolverine", "paul", "picard"];
3
4  const transformUpper = function (inputString) {
```



```

5         return inputString.toUpperCase();
6     };
7
8     const template = '<%= helperFunc(input.join(", ")); %>';
9     const context = {
10         input: people,
11         helperFunc: transformUpper, //user-defined function
12     };
13     console.log(ejs.render(template, context));

```

👉 `transformUpper` is a user-defined function that expects a string as input and transforms it to uppercase. The `context` object has a property `helperFunc` which is assigned a user-defined function as value. In the template, we use the properties of the `context` object; `ejs.render` brings template and data together.

!! JavaScript within EJS templates

To make templates even more flexible, we can incorporate JavaScript in the template, using the `<% scriptlet tag` 👉:

Copy me

```

1     const ejs = require("ejs");
2
3     const template =
4         `<%
5             if(movies.length>2){
6                 movies.forEach(function(m){console.log(m.title)})
7             }
8             %>`;
9
10    const context = {
11        movies: [

```



```

12         { title: "The Hobbit", release: 2014 },
13         { title: "X-Men", release: 2016 },
14         { title: "Superman V", release: 2014 },
15     ],
16 };
17
18     ejs.render(template, context);

```

👉 The context is an array of objects, each movie with a title and release date. In the template, we use `Array.prototype.forEach` (it executes a provided function once per array element) to iterate over the array and print out the title and release date if our array has more than two elements (admittedly, not a very sensible example but it shows off the main principle). The `<%` scriptlet tags are used for **control-flow purposes**. If you replace the opening scriptlet tag with `<%-` or `<%=` (try it!), you will end up with an error. Note that we are using backticks here in our `template` to surround the string as backticks allow the use of multiline strings.

!! Express and templates

By now you might be asking why we are doing this. So far, we have only used Node's REPL to show off some of EJS' capabilities. The REPL though is only for playing around with short code snippets, it does not bring us closer to a server-side script with templating enabled. For that, we need Express. *How do templates tie in with the Express framework?* It turns out that so-called **views** can be easily configured with Express. Not only that, an application can also make use of several template engines at the same time.

Three steps are involved:

- i. We set up the *views directory*—the directory containing all templates. Templates are essentially HTML files with EJS scriptlet tags embedded and file ending `.ejs` :


```
app.set("views", __dirname + "/views");
```
- ii. We define the template engine of our choosing: `app.set("view engine", "ejs");`



iii. We create template files.

A functioning Express/EJS demo can be found [here](#). Try it out yourself by installing and running it. Let's consider the application's `app.js` 🙌:

Copy me

```
1  const express = require("express");
2  const url = require("url");
3  const http = require("http");
4
5  const port = process.argv[2];
6  const app = express();
7  http.createServer(app).listen(port, function () {
8    console.log("Ready on port " + port);
9  });
10
11 const wishlist = [];
12 wishlist.push({ name: "Wingspan", type: "game night" });
13 wishlist.push({ name: "Munchkin", type: "party game" });
14 wishlist.push({ name: "Scrabble", type: "game night" });
15 wishlist.push({ name: "Uno", type: "party game" });
16
17 app.set("views", __dirname + "/views");
18 app.set("view engine", "ejs");
19
20 app.get("/wishlist", function (req, res) {
21   res.render("wishlist", { title: "Game wishlist", input: wishlist });
22 });
```

🙌 We first set up the views directory, then the view engine and finally we use Express' `res.render` in order to render a view and send the rendered HTML to the client.

Important to realize in this example is, that the first argument of `res.render` is a view



stored in `views/wishlist.ejs` which the Express framework retrieves for us. The second argument is an object that holds the variables of the template, here `title` and `input` (an array). To confirm this, let's look at the template file itself, `wishlist.ejs` which contains the corresponding variable names 🙌:

Copy me

```
1    <!DOCTYPE html>
2    <html>
3      <head>
4        <title><%= title %></title>
5      </head>
6      <body>
7        <h1>Wishlist</h1>
8        <div>
9          <% input.forEach(function(w) { %>
10           <div>
11             <h3><%=w.name%></h3>
12             <p><%=w.type%></p>
13           </div>
14         <% }) %>
15       </div>
16     </body>
17   </html>
```

🙌 This is mostly HTML, with a few scriptlet tags sprinkled in. While EJS has more capabilities than we present here, for the purposes of our board game project, this excursion into EJS is sufficient.

OPTIONAL

Node.js in production



When deploying a web app in production, there is usually more to it than we can describe in the two Node.js lectures. Testing is a vital aspect we have not yet covered (there will be a whole course on testing later on in your BSc), but also how to reliably run Node.js in production. One tool to help here is ↗ [PM2](#), a popular production process manager with a built-in load balancer.

Self-check

Here are a few questions you should be able to answer after having followed the lecture and having worked through the required readings:

- ▶ True or False? Node.js has a file-based module system.
- ▶ True or False? Node.js runs each module in a separate scope and only returns the final value of `module.exports`.
- ▶ True or False? The `require` module runs asynchronously.
- ▶ True or False? `module.exports` and `exports` can be used in exactly the same way.
- ▶ True or False? Middleware components can execute code.
- ▶ True or False? Middleware components can change the request and response objects.
- ▶ True or False? Middleware components can start the request-response cycle.
- ▶ True or False? Middleware components can call any middleware function in the middleware stack.
- ▶ Name three different routes, the following handler matches.

Copy me

```
1 app.get('/whaa+[dt]s+upp*', function(req,res){
2   res.send(...)
3 });
```



► What is the output of the code snippet above?

Copy me

```
1    var ejs = require("ejs");
2    var people = ["Wolverine", "paul", "picard"];
3    var X = function (input) {
4        if (input) {
5            return input[0];
6        }
7        return "";
8    };
9    var template = "<%= helperFunc(input); %>";
10   var context = {
11       input: people,
12       helperFunc: X,
13   };
14   console.log(ejs.render(template, context));
```

► Consider the following two files, constants.js and bar.js. What is the console output of node bar.js?

Copy me

```
1    //constants.js
2    module.exports.pi = 3.1415;
3    module.exports.password = "root";
```

Copy me

```
1    //bar.js
```



```
2  var constants1 = require("./constants");
3  constants1.password = "admin";
4  var constants2 = require("./constants");
5  console.log(constants2.password);
6  var constants3 = require("./constants");
7  constants2.pi = 3;
8  console.log(constants3.pi);
```



➤ Claudia Hauff, ➤ [GitHub repository](#)

