COURSEWARE

Professional Skills
Agile Fundamentals
Jira
Git
Databases Introduction
Java Beginner
Maven
Testing (Foundation)
Java Intermediate
HTML
CSS
Javascript
Spring Boot
Selenium
Sonarqube
Advanced Testing (Theory)
Cucumber
MongoDB
Express
- Intro
- Handling Requests
- Middleware
- Error Handling
- Router
- Mongoose
NodeJS
React
Express-Testing
Networking
Security

Handling Requests

Contents

- Overview
- Request Methods
  - GET
  - POST
  - PUT
  - PATCH
  - DELETE
- req
  - URL parameters
  - Query parameters
  - Request body
- res
  - res.send
  - res.status
- Tutorial
- Exercises

Overview

Express comes pre-equipped with functionality for dealing with requests.

Request Methods

Requests have two major components we are interested in: the *path* and the *method*.
The *path* is the location the request will be sent to and the *method* is the type of request.

For example, in order to listen for **GET** requests at **/hello** then we would create this function:

```
app.get('/hello', (req, res) => console.log('Hello, World!'));
```

We can see from this example that the *method* is a function (**.get()**), the first parameter of this function is the *path* (**/hello**) and the second parameter is a callback function that just logs **Hello, world!** to the console.

GET

GET requests are used to *fetch* data, to listen for a **GET** request we use the **.get()** function.

```
app.get('/read', (req, res) => console.log('fetch'));
```

POST

POST requests are used to *create* data, to listen for a **POST** request we use the **.post()** function.

```
app.post('/create', (req, res) => console.log('create'));
```

PUT

PUT requests are used to *replace* data, to listen for a **PUT** request we use the **.put()** function.

Cloud Fundamentals
AWS Foundations
AWS Intermediate
Linux
DevOps
Jenkins Introduction
Jenkins Pipeline
Markdown
IDE Cheatsheet

```
app.put('/replace', (req, res) => console.log('replace'));
```

PATCH

PATCH requests are used for *partial updates*, to listen for a **PATCH** request we use the **.patch()** function.

```
app.patch('/update', (req, res) => console.log('update'));
```

DELETE

DELETE requests are used to *delete* data, to listen for a **DELETE** request we use the **.delete()** function.

```
app.delete('/delete', (req, res) => console.log('delete'));
```

req

In the previous examples you may have noticed that all of the callback functions take in a **req** variable.
req is an object that represents the received request and can be used to view sent data and request metadata such as headers, cookies, etc.

Data can be received in three different ways:

URL parameters

One way to send data is as part of the URL itself.
For example, a request sent to **/delete/1** would normally indicate that an element with an id of 1 should be deleted.

To use URL parameters in Express they must first be declared in the *path* using a : and then access via the **req.params** object.

```
app.delete('/delete/:id', (req, res) => {
    console.log(req.params.id);
});
```

Query parameters

Query parameters are appended to the end of a URL and take a very particular format.
Query parameters always start with a **?**.
Each parameter takes the format of **key=value**.
Parameters are separated by an **&**.

For example, if we had this object:

```
{
    "id": 1,
    "name": "Barry",
    "age": 44,
    "job": "Receptionist"
}
```

and we wanted to send a request to update the **name** and **age** we might send it to a path like this:

```
/update/1?name=barry&age=22
```

In Express query parameters can be accessed from the **req.query** object.

```
app.patch('/update/:id', (req, res) => {
    const id = req.params.id;
    console.log('id: ', id);
    const name = req.query.name;
    console.log('Name: ', name);
    const age = req.query.age;
    console.log('Age: ', age);
});
```

Request body

Larger or more complex data can be sent via the request body, typically this is done in *JSON* (JavaScript Object Notation) format.
Express cannot parse this by default and requires the edition of *body-parsing middleware* which will be covered in more detail in a different module.

In order to parse JSON from the request body we must **use** the json-parser provided by **express.json()**.
Make sure this line goes *above* any request handling.

```
app.use(express.json())
```

If we remember Barry from the previous example:

▶ Barry
We are going to replace him with Bill:
▶ Bill
This time we want to overwrite Barry entirely which we will do by **PUT**-ing with a new JSON object in the request body.

```
app.put('/replace/:id', (req, res) => {
    const id = req.params.id;
    console.log('id: ', id);
    const body = req.body;
    console.log('Body: ', req.body);
});
```

res

The **res** object is used to control the response Express will send to a particular request.

res.send

res.send() will send the response with any data passed into this function passed into the response body.

If you don't send a response then you must call the next function or *your request* ***will*** *hang*.

Here we see a modified version of an earlier example, except this time the message will be sent in the response.

```
app.get('/hello', (req, res) => {
    res.send('Hello, World!');
});
```

res.status

By default responses will have a status code of 200 for OK. OK is obviously not correct if we're trying communicate an error (4xx or 5xx) and even if everything worked there are much more descriptive statuses than 'OK'.

Default:

```
app.get('/hello', (req, res) => {
    res.status(200).send('Hello, World!');
});
```

Example error:

```
app.get('/error', (req, res) => {
    res.status(500).send('Mistakes were made');
});
```

Tutorial

There is no tutorial for this module.

Exercises

1. Create a new folder called 'express_request_handling' and initialise it as a node package.

▶ Click for solution

2. Install Express

▶ Click for solution

3. Create *index.js*, import Express and initialise it.

▶ Click for solution

4. Create a request handler that listens for **GET** requests at **/**, make it send a response of 'Hello, my name is !'.

▶ Click for solution

5. Create a non-constant array containing the names of everyone on your row.

▶ Click for solution

6. Create a request handler that listens at **/getAll** and responds with the whole array.

▶ Click for solution

7. Create a request handler that *fetches* the name at the index specified in a *URL parameter*.

▶ Click for solution

8. Create a request handler that *deletes* the name at the index specified in a *URL parameter*.

▶ Click for solution

9. Create a request handler that *creates* a new name in the array by sending a JSON object in the request body.

Remember to add any necessary middleware

▶ Click for solution

10. Create a request handler that *replaces* a name in the array with a name *specified in a query parameter* at an index *specified in a URL parameter*.

▶ Click for solution