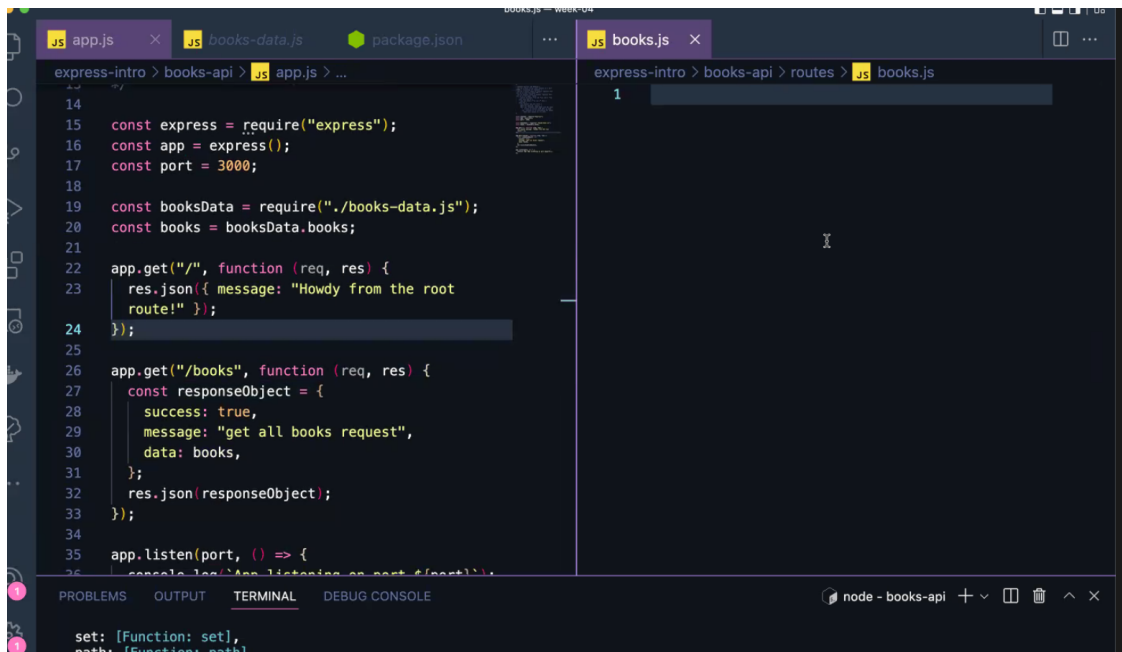


Modularising our server code by separating request handlers into separate routers

Make a folder in root folder (normally called “routes” but can be anything)

Inside, make a file called *something.js* where *something* is an appropriate name for what you are routing e.g. books.js

Before moving routers



```
express-intro > books-api > JS app.js > ...
14
15 const express = require("express");
16 const app = express();
17 const port = 3000;
18
19 const booksData = require("../books-data.js");
20 const books = booksData.books;
21
22 app.get("/", function (req, res) {
23   res.json({ message: "Howdy from the root
24     route!" });
25 });
26
27 app.get("/books", function (req, res) {
28   const responseObject = {
29     success: true,
30     message: "get all books request",
31     data: books,
32   };
33   res.json(responseObject);
34 });
35
36 app.listen(port, () => {
37   console.log(`App listening on port ${port}`);
38 });
```

```
express-intro > books-api > routes > JS books.js
1
```

node - books-api

Plan and first steps screenshot:

```
package.json
express-intro > books-api > app.js > ...
books-data.js
const books = booksData.books;
app.get("/", function (req, res) {
  res.json({ message: "Howdy from the root route!" });
});
app.get("/books", function (req, res) {
  res.json({
    success: true,
    message: "get all books request",
    data: books,
  });
  res.json(responseObject);
});
app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

```
books.js
1 /*
2 - Create a books router using Express's router method ✓
3 - Imported in the books using require ✓
4 - Handle any requests to path '/books/' here in the books router - move the logic out of app.js to this file
5 - Make that router available to use outside of this file and use in app.js
6 */
7
8 const express = require("express");
9 const router = express.Router();
10
11 const books = require("../books-data.js");
12
13
```

```
package.json
express-intro > books-api > app.js > ...
books-data.js
const books = booksData.books;
app.get("/", function (req, res) {
  res.json({ message: "Howdy from the root route!" });
});
app.listen(port, () => {
  console.log(`App listening on port ${port}`);
});
```

```
books.js
1 /*
2 - Create a books router using Express's router method ✓
3 - Imported in the books using require ✓
4 - Handle any requests to path '/books/' here in the books router - move the logic out of app.js to this file ✓
5 - Make that router available to use outside of this file ✓
6 - use in app.js
7 - Hook up router to correct path in app.js
8 */
9
10 const express = require("express");
11 const router = express.Router();
12
13 const books = require("../books-data.js");
14
15 router.get("/books", function (req, res) {
```

After moving routers:

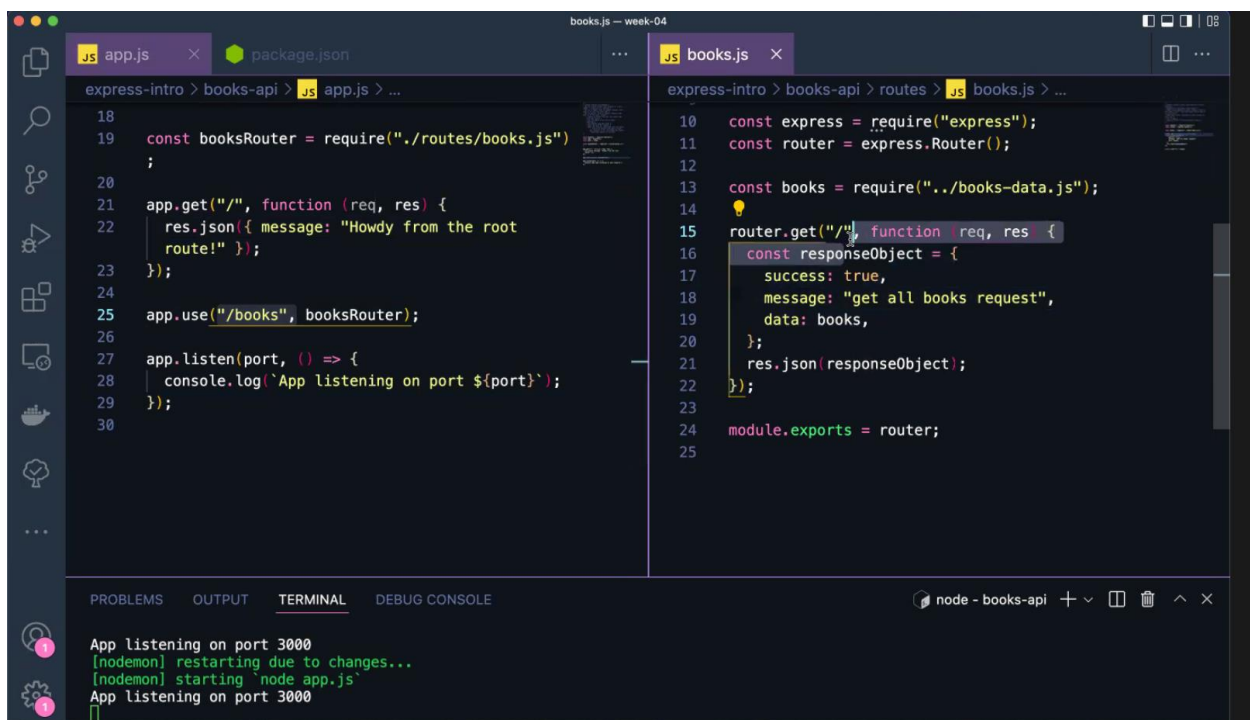
Import express module (const express = require("express");)
Set express.Router() to router variable (const router = express.Router())
Import books item (const books = require("../books-data.js" – be careful with this!)

Copy `app.get` code for path `/books` from `app.js` and refactor to use router variable (change `app.get` to `router.get`)
Export router object (`module.exports = router`)

In `app.js` – import router object (using `const booksRouter = require("./routes/books.js")`)

Use the router object – `app.use("/books", booksRouter)` – 1st argument is the path (which is the URL – `/` equals root path so `localhost: 3000` in this case), 2nd argument is which handler to use

Don't use `/books` in `books.js` as well as in `app.js` otherwise you're making the path `localhost:3000/books/books` and just using `localhost:3000/books` won't work and will return an error



```
books.js — week-04

app.js
18
19 const booksRouter = require("./routes/books.js")
20 ;
21 app.get("/", function (req, res) {
22   res.json({ message: "Howdy from the root route!" });
23 });
24
25 app.use("/books", booksRouter);
26
27 app.listen(port, () => {
28   console.log(`App listening on port ${port}`);
29 });
30

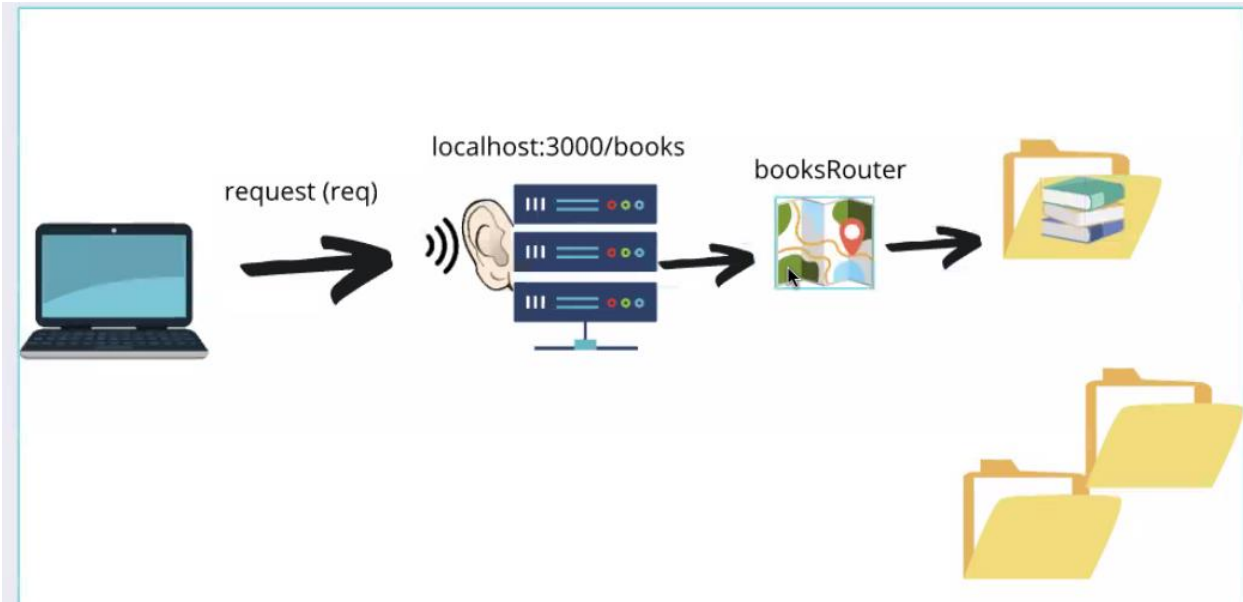
package.json

books.js
10 const express = require("express");
11 const router = express.Router();
12
13 const books = require("../books-data.js");
14
15 router.get("/", function req, res {
16   const responseObject = {
17     success: true,
18     message: "get all books request",
19     data: books,
20   };
21   res.json(responseObject);
22 });
23
24 module.exports = router;
25

TERMINAL
App listening on port 3000
[nodemon] restarting due to changes...
[nodemon] starting node app.js
App listening on port 3000
```

`App.js` is the server

`Books.js` is the `booksRouter` showing the server where to send the request



Making your client more specific (if you want a specific book in this example)

See w4d2_express-books-api

Plan:

- When the client sends a request for a specific book id, respond with that book's information
 - Make another get route (handler) in my books router that listens on a path with an id
 - Get the id param out of the request so that we can use it
 - Logic that looks through the books and finds the right book using the id (for loop?)
 - Variable to store the found book
 - For loop to loop through the array and visit each book
 - Test if the param id matches each book's id with `===` in an if statement
 - When we find the object we want in the array, we can stop looping and store it in the variable we made earlier
 - Create responseObject with data to be sent back
 - Use `res.json()` to send a JSON with the correct book back to the client

In `books.js` setup a new `router.get` (before exports at the bottom):

Use the parameter (params) `:id` after the slash to say "expect client to put something after the slash and whatever is there, use it as the id"

This stores it inside the req (request) object which has a property called “params”. The id is the key and has a value of what the client puts after books/

PARAMS ARE ALWAYS STRING!

To access the id, use req.params.id

To get the book, use a for loop to loop through the array and use the id to select

```
router.get("/:id", function (req, res) {  
  const searchedId = req.params.id;           not required but saves typing req.params.id every time!  
  let searchedBook = {};                     declare variable outside of loop otherwise it won't work  
  for (let i = 0; i < books.length; i++) {  
    //Convert searchedId to number otherwise it won't work – see quote below from Arshi why not ==  
    if (Number(searchedId) === books[i].id) {  
      searchedBook = books[i];  
      break;  
    }  
  }  
  const responseObject = {  
    success: true,  
    message: `get book with id ${searchedId}`,  
    data: searchedBook,  
  };  
  
  res.json(responseObject);  
});
```

Quote from Arshi:

Technically you could use ==, but explicitly converting to Number makes your intention clearer to current you, future you and others on your team. Not a hard and fast rule, but try to avoid == at this stage as JS has lots of implicit/quirky type conversion rules which can be “gotchas”.

Adding Search Queries

Take these in as key:value pairs.

In the path you put a question mark e.g. localhost:3000/books?title=night

Title = key and night = value

Could do a separate request handler but in our example, we're adding it inside the books.js request handler

The plan

- If a request comes in to “/books” with a query with the key of title, respond with all books that include the value of the query in their title
 - Get the title query out of the req object: req object has a query property – req.query.title
 - Check if there’s a title and if so, continue but if not, just give everything (last step)
 - Compare title query to the titles in our books object – go through the array (for loop and if statement)
 - Make a variable to store titles – array
 - If statement to compare title from query to title property of each book object – so if the title of the book includes the query string, push it into the array
 - If we find one or more books objects whose title property contains the query string, add them to a new array
 - Make a response object and use the found books array in it
 - Hand response object to res.json()
- If a request comes in to “/books” without a query with the key of title, just respond with all the books (which was our previous functionality already working)

As well as params, req object has a property called query that will contain a key:value pair. E.g for localhost:3000/books?title=night, query: { title: “night” }

Inside books.js, modify the root route already defined to add query functionality to it:

```
router.get("/", function (req, res) {
  //Create a variable that gets the title from req.query property -
  req.query.title
  const titleQuery = req.query.title;
  //Create empty array to store books found in the query search
  const booksFound = [];
  //Check if the title contains a query - if(title)
  // Same as writing title !== undefined as undefined is a “falsey” value (e.g.
  this is asking if there is a title)
  if (titleQuery) {
    //If so, loop through the books and check if the title query matches the titles
    of any books - for loop
    for (let i = 0; i < books.length; i++) {
      //Make both the query and the titles lowercase so you can compare
      if(books[i].title.toLowerCase().includes(titleQuery.toLowerCase())) {
        // -If we find any matching books - push them to the array
        booksFound.push(books[i]);
      }
    }
  }
  const responseObject = {
    success: true,
```

```

    message: `Books found for ${titleQuery}`,
    books: booksFound,
  }
  res.json(responseObject);
// Add a return to the bottom of the function so stops here if has worked
  return;
}
// This is the else bit that we already had before
const responseObject = { success: true, books: books };
res.json(responseObject);
console.log(books);
});

```

Remember the “/book” is the path – in cloud servers this will be the URL but for now we are on localhost:3000/book

Another convention to standardise our APIs - REST

<https://restfulapi.net/>

REST is an acronym for REpresentational State Transfer and an architectural style for distributed hypermedia systems. Roy Fielding first presented it in 2000 in his famous dissertation.

Like other architectural styles, REST has its guiding principles and constraints. These principles must be satisfied if a service interface needs to be referred to as RESTful.

This is a standard way to write API.

Has 6 principles to follow (although the 6th one is optional!)

Read up on this in your own time – we will have a guest talk in a few weeks that delves in to this more.

Different types of request

CRUD actions = Create, Read, Update, Delete

These are the 4 actions we can use to communicate with data within an API or database.

APIs are the interface between clients and servers.

These represent the 4 types of HTTP requests you can make:

Create = POST

Read = GET

Update = PUT, PATCH

Delete = DELETE

Think about a pair of trousers and wardrobe! Trousers = the data and the wardrobe = the server

Get request is like walking to your wardrobe, picking out the trousers you want to wear and getting them out

Post request is like creating your own pair of trousers from scratch and add them to your wardrobe! (but you had no trousers before! Only for new trousers)

Delete is putting your trousers in the bin!

Put is replacing trousers you already own – replace them with a new pair! (But the original existed in the wardrobe first)

Patch is patching a hole in your trousers! (so replacing a bit of them but not completely replacing)