MERN stands for:

MongoDB	JSON Database
Express	Node.js Web Framework
React	Client-side JavaScript Framework
NodeJS	JavaScript Webserver

In these lessons we will be making an Inventory App that store's book data and users in a MongoDB database. NodeJS and Express will be the server side that acts like a bridge between the MongoDB database and the React client. The React is the client side that will interact with the Web Browser and uses the nodejs server to read and write to the MongoDB database.

Conventions used in these lessons:

bold - headings, keywords, code

italics - code syntax

underline - important words

How the Lessons work:

(1) We can teach you step by step on Skype or Zoom where we instruct you, what to do. We work with you interactively so that you can get your app and running quickly. We explain each step to you so that you can complete your app and understand your app. There are many home works to accomplish for each Lesson. We will always take up the home work before we start the next lesson. Each lesson is designed for 1 hour. You may progress faster or slower depending on your learning speed. Each Lesson is $15 per hour. The lessons are a courtesy to students so that they can learn programming conveniently.
Lesson appointments can be booked online through students@cstutoring.com or from our web site https://www.onlineprogramminglessons.com/

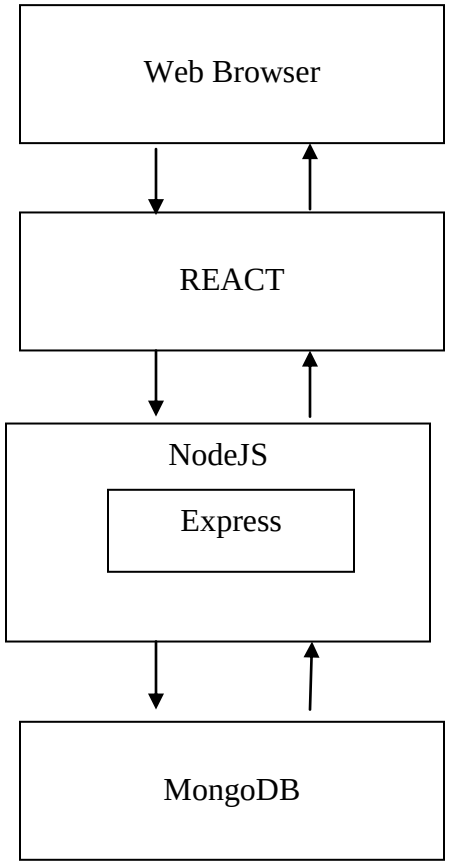
(2) Individuals may work directly from the lesson pdf file and then make an appointment when they need further explanation. We charge $15 per hour or part of for each appointment.
Programming help is available at $20 per hour for lesson program help and $25 per hour for non-lesson program help.
Lesson or homework appointments can be booked directly through students@cstutoring.com

Lesson1
Introduction to MERN,NodeJS, MongoDB and setting up NodeJS Server

Introduction to MERN

MERN is a 3-tier architecture with frontend, backend and database entirely using JavaScript and JSON.

Tier	Component(s)	Purpose
FrontEnd	React	Web Client
BackEnd	NodeJS Express	Web Server
Database	MongoDB	Store Data for future use



Backend

NodeJS is basically a web server that you connect to. You can connect to nodeJS directly from a web browser. Nodejs is considered server side because it is doing many internal calculations independent of a web browser. NodeJS has a port where in connects to a client. The client can be a web browser or a React app.

Express is a Nodejs web framework that runs inside the Node.js server. Express performs URL routing that matches an incoming URL with server functions like storing data, lookup updating and deleting. Express handles HTTP GET POST, PUT And DELETE requests and responses.

Frontend

React is a frame work that displays web pages to a web browser. React is considered client side because it interacts mainly with the web browser client, displaying web pages. React has its own communication port independent of nodeJS where it communicates to the web browser. React uses JavaScript modules from NodeJS.

Database

MongoDB is a data base that reads and writes data using JSON. JSON stores data in a string format the can be easily read and written to. Mongodb does not use SQL but instead has dedicated functions to save, update, find and store data in database tables using the JSON format. MongoDB easily works with as React, Express, and Node.

NodeJS, Express React and MongoDB are all written in JavaScript and use JSON to transfer data between them. We will now install and set up NodeJS, React and MongoDB.

Installing and Setting up NodeJS

You first need to install node.js.

Step 1 Install node.js

Node js is available for Windows, MacOS and Linux free of charge.
https://nodejs.org/en/download/

We use Windows in these course lessons. Downloaded the Windows Installer (.msi) installer file either 32 bit or 64 bit depending on your computer set up, and run it. **nodejs** installs with no problems and all paths are set up for you automatically. You can easily do the same.

If you all ready have nodejs installed on your computer you may want to reinstall to the latest version. You should be using node.js version 10 or greater.

You can check your node js version like this:

node –v
v10.15.3

NPM is a package manager for Node.js.
NPM is used to install additional node.js modules.
The NPM program is installed on your computer when you install Node.js

Step 2: Make Inventory App folders

Make a top level folder called **inventory-app**.
Inside make two folders called **server** and **client.**

```
C:\Users\ADMIN>d:
D:\>mkdir inventory-app
D:\>cd inventory-app
D:\inventory-app>mkdir server
D:\inventory-app>mkdir client
D:\inventory-app>dir
 Volume in drive D is New Volume
 Volume Serial Number is F685-E7C2
```

```
 Directory of D:\inventory-app

2021-03-26  07:27 PM    <DIR>          .
2021-03-26  07:27 PM    <DIR>          ..
2021-03-26  07:27 PM    <DIR>          client
2021-03-26  07:27 PM    <DIR>          server
               0 File(s)              0 bytes
               4 Dir(s)  281,664,606,208 bytes free

D:\inventory-app>\
```

Step 3 make the nodejs server

Go to the server folder and then initialize the nodejs server. We use the –y option when means accept all default values.

```
D:\inventory-app>cd server
D:\inventory-app\server>npm init -y
Wrote to D:\inventory-app\server\package.json:
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Step 4 Next we need to install the following the node.js module:

Install express:

```
D:\inventory-app\server>npm install express
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN server@1.0.0 No description
npm WARN server@1.0.0 No repository field.

+ express@4.17.1
added 50 packages from 37 contributors and audited 50 packages in 2.635s
found 0 vulnerabilities
```

Step 5 Write the server.js program

Open notepad and copy/paste or type in the following program server.js and place in the clients folder.

D:\inventory-app\server>**notepad server.js**

```
// server.js

// import  required modules
const express = require('express');

// create server app
const app = express();

// use port 4000
const apiPort = 4000;

// return server info as a default get requests
app.get('/', (req, res) => {
        res.send('Inventory App Server');
});

// start the server, listen for requests
app.listen(apiPort, () => console.log(`Server running on port ${apiPort}`));
```

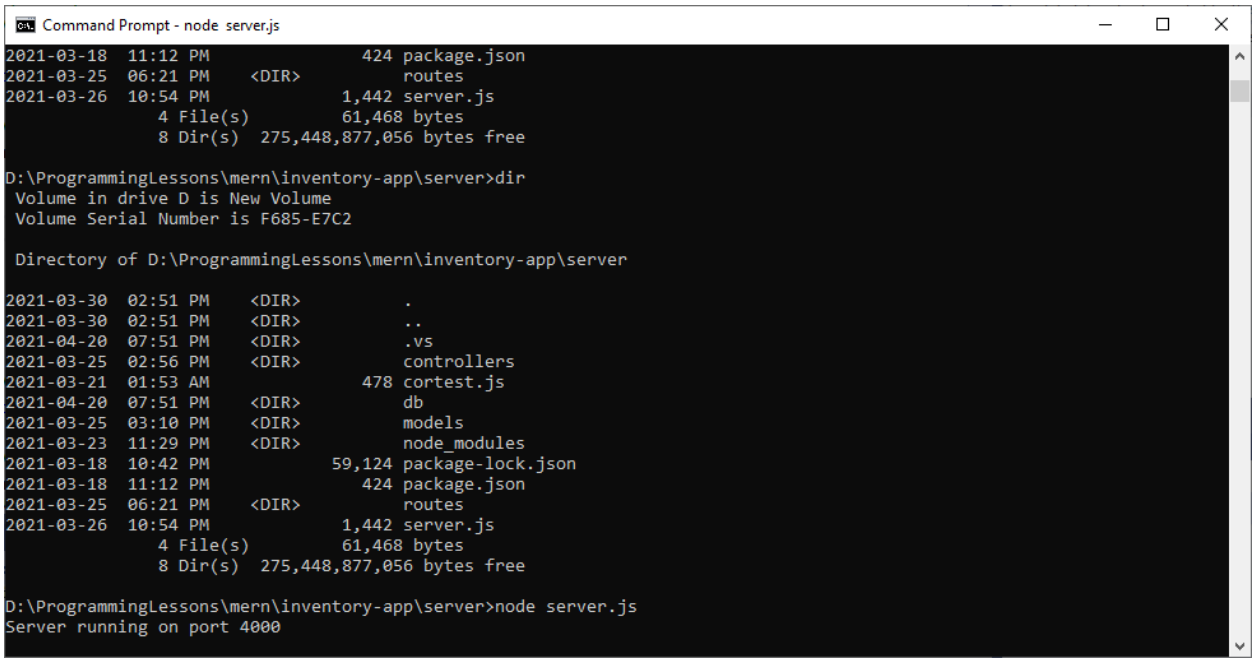
Step6 run server

From your ms-dos prompt type in the following to start the server:

D:\inventory-app\server>**node server.js**

You should get something like this:

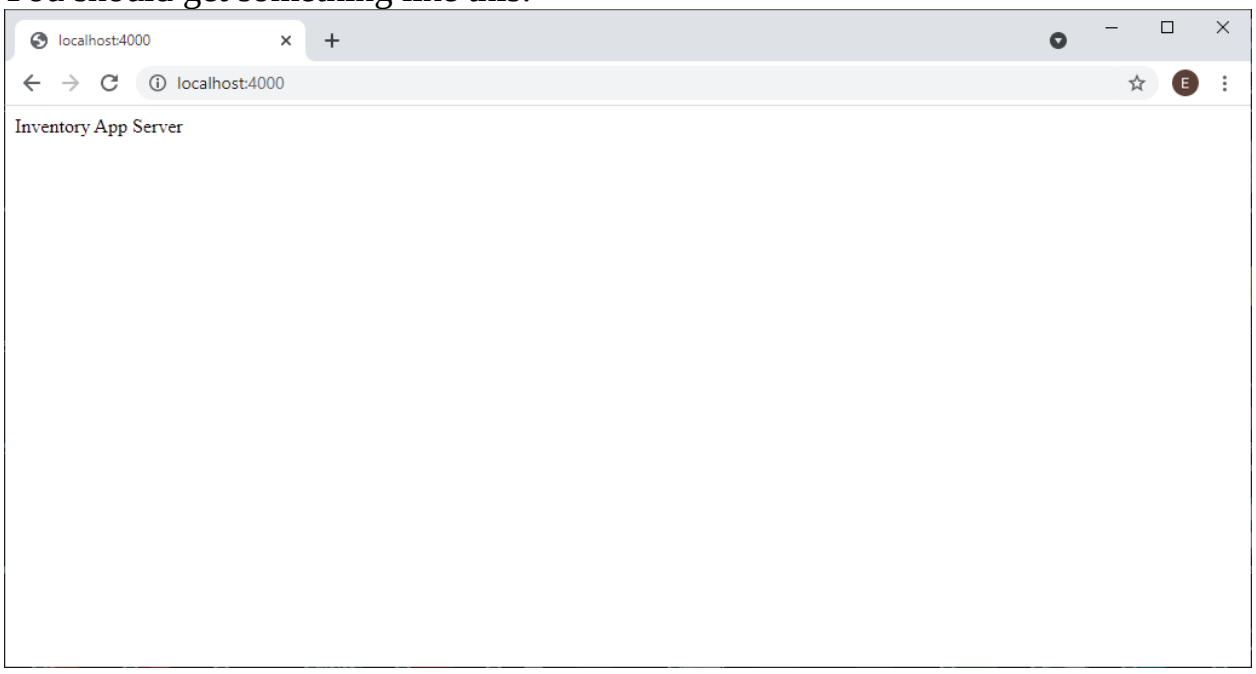
Server running on port 4000



Step 7 go to web browser localhost port 4000

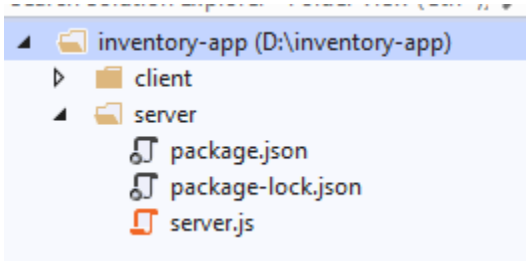
In the url address bar Type in: http://localhost:4000/

You should get something like this:



If you did not get the web page then you typed in the program incorrectly, or node.js is installed improperly. Or you typed in the wrong url in the address bar or your web browser is not working properly or local host is not working properly. If you cannot resolve this issue then you can book an appointment with one of our technical people at $25 per hour or part of. Send us an email at students@cstutoring.com

Your inventory-app folders should now look like this:



How the server code works:

We first import the required modules using the **require** directive.
we use **const** rather than **var** or **let** because these module variables do not changed once assigned.

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

Next we create the express module to run our server app.

```
const app = express()
```

We want to use port 4000.

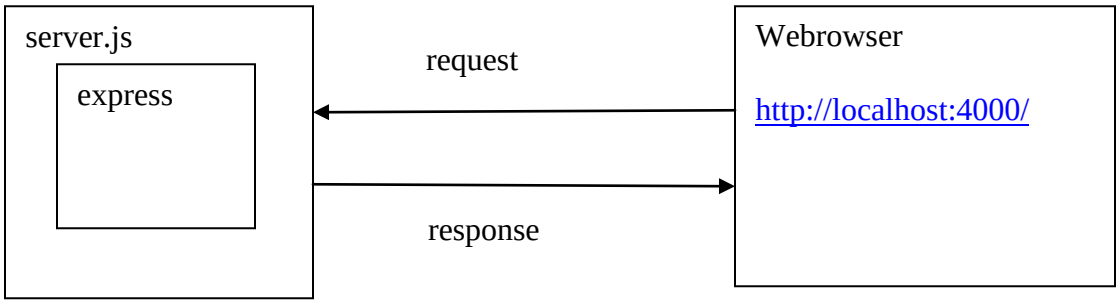
```
const apiPort = 4000
```

When we get a default get request at our port 4000, the request object req would contain the request data. We then send back a message 'Inventory App Server' back to the web browser using the **res** respond object **send** method. The express module handles requests for us.

```
app.get('/', (req, res) => {
        res.send('Inventory App Server')
})
```

We then start the server and listen for requests.

```
app.listen(apiPort, () => console.log(`Server running on port ${apiPort}`))
```



LESSON 2 INSTALLING MONGODB DATABASE

Install the MongoDB community Server community version it is free.
https://www.mongodb.com/try/download/community

select the **msi** installer package file to down load.

We have downloaded the windows msi installer and ran it.
Select the Complete setup install option and install as a service if that option is available. Mongodb installs with no problems.
Once installed you may have to make a data directory in your C drive

Open up a msdos prompt:

```
C:\Users\ADMIN>cd ..
C:\Users>cd ..
C:\>mkdir data
C:\>cd data
C:\data>mkdir db
C:\data>
C:\data>dir
 Volume in drive C has no label.
 Volume Serial Number is 0A37-1D0C

 Directory of C:\data

2021-03-27  01:26 PM    <DIR>          .
2021-03-27  01:26 PM    <DIR>          ..
2021-03-27  01:26 PM    <DIR>          db
               0 File(s)              0 bytes
               3 Dir(s)  169,986,711,552 bytes free

C:\data>
```

If your mongodb is not a service then you need to start the mongodb server manually as follows:

```
C:\data>cd ..
C:\>cd "Program Files"
C:\Program Files>cd MongoDB
C:\Program Files\MongoDB>cd server
C:\Program Files\MongoDB\Server>cd 4.0
C:\Program Files\MongoDB\Server\4.0>cd bin
C:\Program Files\MongoDB\Server\4.0\bin>mongod
```

or may need to specify the database file path as well

```
mongod.exe --dbpath="c:\data\db"
```

The --dbpath option points to your database directory.

You should get the message:

```
Waiting for connections on port 27017
```

If this is not working then you need to trouble shoot on your own.

If you cannot resolve this issue then you can book an appointment with one of our technical people at $25 per hour or part of. Send us an email at students@cstutoring.com

Using Mongo Command Shell

The Mongo Command shell is used to execute MongoDB commands

Basic MongoDB CRUD operations

- **C**reate Operations
- **R**ead Operations
- **U**pdate Operations
- **D**elete Operations

To start the Mongo Command Shell navigate to C:\Program Files\MongoDB\Server\4.4\bin or your particular version.

```
"C:\Program Files\MongoDB\Server\4.4\bin\mongo.exe"
```

You will get the following prompt: (it is the '>' on the next line)

```
>
```

Our first command we will print out our data base name.

```
> db.getName()

test
>
```

Next we create a new data base called grocery_store. The data base will be made automatically for us. MongoDB may contain many databases and each data base may contain many collections (tables)

```
> use grocery_store

switched to db grocery_store

print out the data base name

> db.getName()

grocery_store
>
```

We will now test all the CRUD operations

Create Operations

We use the **insert** command to add new documents (rows) to a collection (tables). If the collection does not currently exist, the insert command will create the collection automatically.

We will have a products table containing an id, name, price and department.

```
> db.products.insert({ _id: 1, name: "apple", price: 1.19, department: "fruit" })

WriteResult({ "nInserted" : 1 })
```

We can then query the data base using the **find** function with a name of the product to check if it got inserted into the table

```
> db.products.find( { name: "apple" } )

{ "_id" : 1, "name" : "apple", "price" : 1.19, "department" : "fruit" }
```

With the **insertMany** command we can add many more products all at once

```
> db.products.insertMany([
...    { _id: 2, name: "chicken", department: "meat" },
...    { _id: 3, name: "corn", price: 1.35, department: "vegetables" },
...    { _id: 4, name: "orange", price: 1.29, department: "fruit" }
... ])

{ "acknowledged" : true, "insertedIds" : [ 2, 3, 4 ] }
```

We can use **find** to printout all the products at once

```
> db.products.find()

{ "_id" : 1, "name" : "apple", "price" : 1.19, "department" : "fruit" }
{ "_id" : 2, "name" : "chicken", "department" : "meat" }
{ "_id" : 3, "name" : "corn", "price" : 1.35, "department" : "vegetables" }
{ "_id" : 4, "name" : "orange", "price" : 1.29, "department" : "fruit" }
```

We can also printout in **json** format (which seems to be the default as above)

```
> db.products.find({}).forEach(printjson);

{ "_id" : 1, "name" : "apple", "price" : 1.19, "department" : "fruit" }
{ "_id" : 2, "name" : "chicken", "department" : "meat" }
{ "_id" : 3, "name" : "corn", "price" : 1.35, "department" : "vegetables" }
{ "_id" : 4, "name" : "orange", "price" : 1.29, "department" : "fruit" }
```

Updating a Product

We will change the price of the apple to 1.39

```
> db.products.update(
...    { _id: 1 },
...    {
...      $set: {
...        price: 1.39
...      }
...    }
... )

WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

We now verify to see if the price of the apple was updated:

```
> db.products.find()

{ "_id" : 1, "name" : "apple", "price" : 1.39, "department" : "fruit" }
{ "_id" : 2, "name" : "chicken", "department" : "meat" }
{ "_id" : 3, "name" : "corn", "price" : 1.35, "department" : "vegetables" }
{ "_id" : 4, "name" : "orange", "price" : 1.29, "department" : "fruit" }
```

Delete a row

Now we delete the apple product

```
> db.products.deleteOne( { _id : 1 } )

{ "acknowledged" : true, "deletedCount" : 1 }
```

We now check if the apple has been deleted

```
> db.products.find()

{ "_id" : 2, "name" : "chicken", "department" : "meat" }
{ "_id" : 3, "name" : "corn", "price" : 1.35, "department" : "vegetables" }
{ "_id" : 4, "name" : "orange", "price" : 1.29, "department" : "fruit" }
```

Drop a table

Sometimes you need to drop a table if you want to add fresh data, or you do not need the table any more.

```
> db.products.drop()

true
```

We now Check if the table is gone

```
> db.products.find()
```

Other useful commands

Print out Collection Info

The db.getCollectionInfos() will print out the collection information

```
> db.getCollectionInfos()
[
	{
		"name" : "collection_name",
		"type" : "collection",
		"options" : {

		},
		"info" : {
			"readOnly" : false,
			"uuid" : UUID("a153d925-c7a7-4f2d-9de5-584725ea3c72")
		},
		"idIndex" : {
			"v" : 2,
			"key" : {
				"_id" : 1
			},
			"name" : "_id_",
			"ns" : "test.collection_name"
		}
	},
	{
		"name" : "user",
		"type" : "collection",
		"options" : {
		},
		"info" : {
			"readOnly" : false,
			"uuid" : UUID("61cd3f76-3903-4075-bddd-44ebdc430475")
		},
		"idIndex" : {
			"v" : 2,
			"key" : {
				"_id" : 1
			},
			"name" : "_id_",
			"ns" : "test.user"
		}
	}
]
>
```

Show Database Tables

The db.show_tables() command will show all the tables in the current data base

```
> show tables
collection_name
user
```

Show Database Names

The db.show_databases() command will show all the databases stored in MongoDB

```
> show databases
admin            0.000GB
bezkoder_db      0.000GB
cinema           0.000GB
config           0.000GB
grocery_store    0.000GB
info3069db       0.000GB
info3089db       0.000GB
inventory        0.000GB
local            0.000GB
mongoose_basics  0.000GB
shopping         0.000GB
test             0.000GB
```

Using default ID'S

If you do not supply a ID then the mongodb make a default unique id for you. It is quite a large number. Most people use the default id's for convenience.

```
> db.products.insert({name: "apple", price: 1.19, department: "fruit" })

WriteResult({ "nInserted" : 1 })

> db.products.find()

{ "_id" : ObjectId("60636cb45de9b86effa84858"), "name" : "apple", "price" :
1.19, "department" : "fruit" }
>
```

Printing out Collection Names and Rows

Here is some handy code to print out collection names and row data

```
> db.getCollectionNames().forEach(c => {
...     db[c].find().forEach(d => {
...         print(c);
...         printjson(d)
...     })
... })
products
{
        "_id" : ObjectId("60636cb45de9b86effa84858"),
        "name" : "apple",
        "price" : 1.19,
        "department" : "fruit"
}
>
```

Lesson 3 Adding MongoDB to our Node.js Server

Now that we have installed MongoDB we need to connect it to our nodejs server The main purpose of our server is to connect out **react** Client to our mongoDB database located in our server.

The next thing we have to do is to define our database tables. MongoDB tables are designed in a Mongo Schema.

A mongo schema is a JSON object that defines the column fields used in a table.

We will have 2 tables, a **books** table that stores books for sale and a **users** table that is used to log in users that can use the inventory app.

We will put our schemas models in a models folder

Todo: In your server folder make a **models** folder.

book table

The book table will store isbn, title, author, price, quantity and image.
Here is the book-model schema file called book-model.js

```
// book-model.js
// define book database table
const mongoose = require('mongoose')
const Schema = mongoose.Schema

// define book table
const Book = new Schema(
	{
		isbn: { type: String, required: true },
		title: { type: String, required: true },
		author: { type: String, required: true },
		price: { type: Number, required: true },
		quantity: { type: Number, required: true },
		image: { type: String, required: true },

	},
	{ timestamps: true },
)

// export book database model
module.exports = mongoose.model('books', Book)
```

The module exports statement creates the model with the table name books and with the JSON schema Book.

todo: put the book-model in a file called book-model.js and put in the models folder.

users table

The user table will store user name, email, password and creation date.
Here is the user-model schema file book-model.js

```
//user-model.js
//define user database table

const mongoose = require("mongoose");

const User = mongoose.Schema({
    username: {
        type: String,
        required: true
    },
    email: {
        type: String,
        required: true
    },
    password: {
        type: String,
        required: true
    },
    createdAt: {
        type: Date,
        default: Date.now()
    }
});

// export user database model
module.exports = mongoose.model("users", User);
```

The module exports statement created the model with the table name **users** with the JSON schema **User**.

todo: put the book-model in a file called user-model.js and put in the models folder.

Database Connection file db.js

The next thing we have to do is make a small JavaScript file to connect to the mongoDB data base.

Our database name is: **inventory.**

Todo:

First make a folder inside the sever folder called **db**.

Then make a **db.js** file to store the mongodb connection code.

Here are the connection details:

We require mongoose to connect to our Mongodb data base

```
const mongoose = require('mongoose')
```

We connect to localhost port 270217 where the mongoDB is listening for commands.

The Mongoose uses a Promise when it tries to connect to the MongoDB data base. A Promise is an object that returns a value at a later time when the result value is known through a call back. A call back is when another function calls another function when some thing is ready.

Promises allow you to use an asynchronous method, get a final value, and return value in the form of .**then()** function calls at a later time. The promise will call the function passed to its **then** function as a callback. In our case the **then** function is called when the mongoose connects to the mongoDB.

The **then** call back function is an anonymous function that prints out the connection success message.

Note: Anonymous function is a function with no name.

```
() => {console.log("You are connected to the database inventory!");}
```

really means

```
f() {console.log("You are connected to the database inventory!");}
```

An Anonymous function is a compact short form of a function. The function does not need a name since only code is needed.

In case of an error the **catch** function is called. The promise will call the function passed to the **catch** as a callback. In our case if the mongoose cannot connect to the mongoDB data base. The catch call back function is an anonymous function that prints out the connection error message.

Note: Anonymous function is a function with no name.

e => {console.error('Connection error', e.message)}

really means

f(e) { console.error('Connection error', e.message)}

An Anonymous function is a compact short form of a function. The function does not need a name since only code is needed.

Here is the mongoose connection that uses a promise.

```
// connect to mongo db
mongoose
.connect('mongodb://127.0.0.1:27017/inventory',
      { useNewUrlParser: true, useUnifiedTopology: true  })
.then(() => {
       console.log("You are connected to the database inventory!");
})

.catch(e => {
       console.error('Connection error', e.message)
})
```

We set **useNewUrlParser**: to true to use the new parser and we also set **useUnifiedTopology** to true if we are using the newer connection management engine. If we cannot connect an 'error' event is issued which is caught and report an error. If we do connect then the connection message is displayed:

You are connected to the database inventory!"

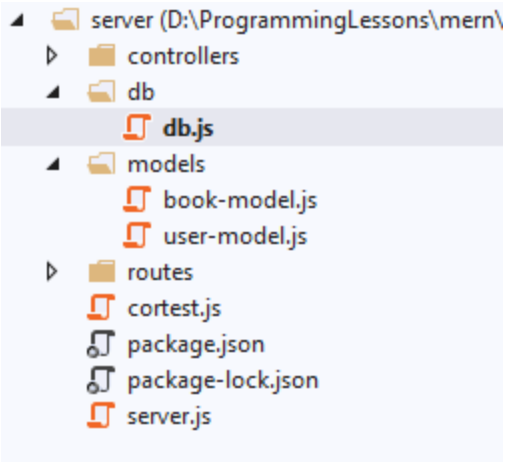
Finally we export the db connection stored in the mongoose object.

```
// save connection
const db = mongoose.connection

// export connection
module.exports = db
```

Here is the complete db.js file:

```
// db.js
// connect to mongo db using mongoose

const mongoose = require('mongoose')

// connect to mongo db
mongoose
	.connect('mongodb://127.0.0.1:27017/inventory', { useNewUrlParser: true,
useUnifiedTopology: true })
	.then(() => {
		console.log("You are connected to the database inventory!");
	})
	.catch(e => {
		console.error('Connection error', e.message)
	})

// save connection
const db = mongoose.connection

// export connection
module.exports = db
```

Your folders should now look like this:



Updating server code:

We have to update the server code to connect to our mongodb database

We first need to install the **mongoose** module.

```
npm install mongoose
npm WARN mern_lesson2@1.0.0 No description
npm WARN mern_lesson2@1.0.0 No repository field.

+ mongoose@5.12.2
added 31 packages from 94 contributors and audited 81 packages in 3.85s
found 0 vulnerabilities
```

Next we add the import to out server.js file

```
const db = require('./db/db')
```

and then we connect to mongodb with the on function

```
// connect to mongodb data base
db.on('error', console.error.bind(console, 'MongoDB connection error:'))
```

Here is the complete updated server.js

```
// server.js

// import  required modules
const express = require('express');
const db = require('./db/db')

// create server app
const app = express();

// use port 4000
const apiPort = 4000;

// connect to mongodb data base
db.on('error', console.error.bind
    (console, 'MongoDB connection error:'))

// return server info as a default get requests
app.get('/', (req, res) => {
        res.send('Inventory App Server');
});

// start the server, listen for requests
app.listen(apiPort, () => console.log
    (`Server running on port ${apiPort}`));
```

You can stop the node.js server with ctrl ^C (twice)

Now Run the server

```
node server.js
```

You should get something like this:

```
D:\ProgrammingLessons\mern\mern_lessons\mern_lesson2>node server.js
Server running on port 4000
You are connected to the database inventory!
```

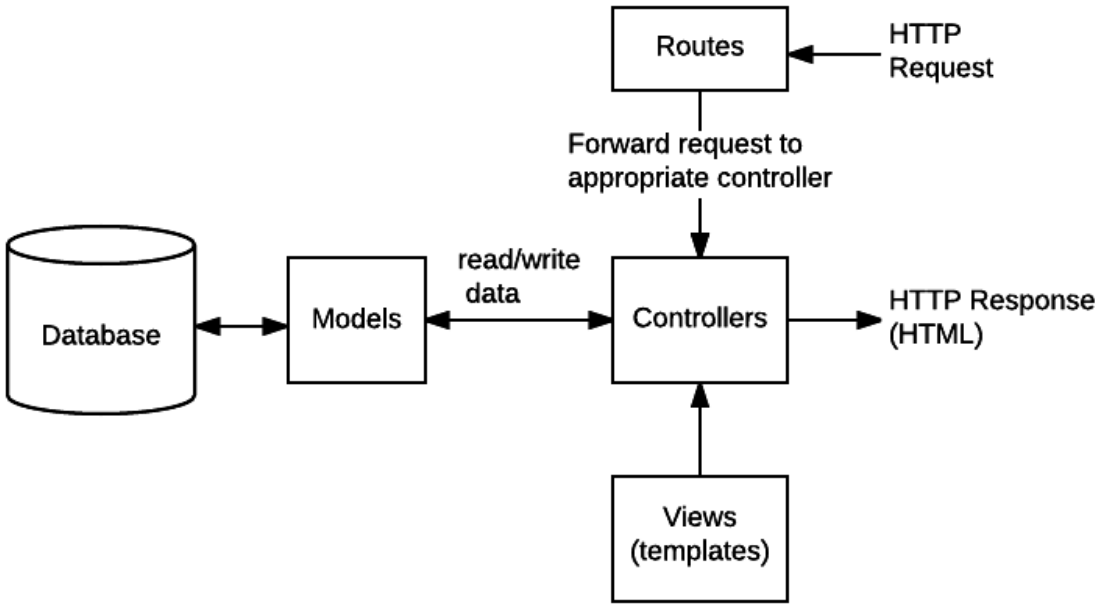
LESSON 4 ROUTING AND SETTING UP OUR NODE.JS ROUTER

Routing lets you connect requests to specified responses. A request could be registering a user, logging in a user, looking up a book etc. A response can be a OK, or a send back book data for looking up a book. Requests are made to a particular endpoint. An end point is a url with designated path like:

http://localhost/api/books

The above request would respond with a list of books. Each path would route to the appropriate respond function. A respond can be a piece of code or a designated function. A Controller is used to process the request and provide the response. The controller access the data base using the models. The views are used to format html the response. This architecture is called MVC **M**odel **V**iew **C**ontroller. In this application we do not have any views since our React client is providing the views for us. Our responses is mostly data from the data base.

MVC Architecture:



Each request has a specified methods. The request methods are POST, GET, PUT and DELETE. Different request methods respond in different ways. Here are the request methods that we use in our inventory app.

Method	Op	Description	URL Example
POST	C	Insert data into database	http://localhost/api/book
GET	R	Return a value from database	http://localhost/api/books
PUT	U	Update a data base with a value	http://localhost/api/book/id
DELETE	D	Delete a row from a database	http://localhost/api/book/id

The GET and DELETE method receive data on the **url** path with an id like:

http://localhost/api/books/12345

or the url path followed by a '?' and name value pairs (name=value)

http://localhost/api/books?id=12345

POST and PUT requests send data inside a message body where the url path is just for routing. To receive data on a message body a **body parser** is needed. A body parser parses the body message and extracts a data object from it.

CONTROLLERS

Before we can make the router we need to make some controllers.

Todo:

Make a folder called **controllers** and then make a empty **book-ctrl.js** file and a **empty user-ctrl.js** file.

Book Controller

The book controller handles all book data requests. Creating, updating, deleting and retrieving books. Some requests are GET, POST,PUT AND DELETE. The following chart shows the required routing and response:

Request	method	Example	Response
Create book	POST	api/book	Return created book
Get all books	GET	api/books'	Return list of books
Get book by id	GET	api/book/:id'	Return book for id
Update book	PUT	api/book/:id'	Return updated book
Delete book by id	DELETE	api/book/:id'	Delete updated book

:id means a book id

The user controller is used for registering a user and logging in users,

Request	method	Example	Response
Register user	POST	api/register	Return created user
Login user	GET	api/login	Return logged in user

For now theses file will just have basic code that will be filled in later.
We just print out the request on the server and then send a message back to the client.

Here is the book-ctrl starter code

```
// book-ctrl.js
// book controller
// handles all book data base requests from client

// book table
const Book = require('../models/book-model')

// create book
// router.post('/book', BookCtrl.createBook)
createBook = (req, res) => {

    console.log("create  book")
    res.send('create  book');

}

// update book
// router.put('/book/:id', BookCtrl.updateBook)
updateBook = async (req, res) => {

    console.log("update  book")
    res.send('update  book');
}
```

```
// delete book
// router.delete('/book/:id', BookCtrl.deleteBook)
deleteBook = async (req, res) => {

    console.log("delete book")
    res.send('delete book');
}

// get book by id
// router.get('/book/:id', BookCtrl.getBookById)
getBookById = async (req, res) => {

    console.log("get book by id")
    res.send('get book by id');
}

// return all books
// router.get('/books', BookCtrl.getBooks)
getBooks = async (req, res) => {

    console.log("get books");
    res.send('get books');
}

// export all book controller functions
module.exports = {
    createBook,
    updateBook,
    deleteBook,
    getBooks,
    getBookById,
}
```

to do:

Put the above code in the book-ctrl.js file located in the controllers folder.

Here is the user-ctrl starter code

```
// user-ctrl.js
// user controller handles login and registration from client

// user data base model
const User = require('../models/user-model')

// login user
// router.post('/login', UserCtrl.loginUser)
loginUser = async (req, res) => {

    console.log("login: " + JSON.stringify(req.body))
    res.send('login');
}
```

```
// register user
// router.post('/register', UserCtrl.registerUser)
registerUser = (req, res) => {

    console.log("register  user")
    res.send('register  user');
}


// export all book controller functions
module.exports = {
    loginUser,
    registerUser
}
```

to do:

Put the above code in the user-ctrl.js file located in the controllers folder.

ROUTER

The next step is to make the router.

TODO:

Make a folder called **routes** and a file called **router.js**
Then put the following code in the router.js file

```
// router.js

// the router handles all requests
// and calls the appropriate controller function

// use express
const express = require('express')

// use these controllers
const UserCtrl = require('../controllers/user-ctrl')
const BookCtrl = require('../controllers/book-ctrl')

// make router
// routing refers to how an application's endpoints (URIs)
// respond to client requests.
const router = express.Router()
```

```
// user routes
router.post('/login', UserCtrl.loginUser)
router.post('/register', UserCtrl.registerUser)

// book routes
router.post('/book', BookCtrl.createBook)
router.put('/book/:id', BookCtrl.updateBook)
router.delete('/book/:id', BookCtrl.deleteBook)
router.get('/book/:id', BookCtrl.getBookById)
router.get('/books', BookCtrl.getBooks)

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

Our router connects all the routes to our controllers. Our Router uses **Cross-Origin Resource Sharing.**

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any other origins (domain, scheme, or port) than its own from which a browser should permit loading of resources.

Our router also uses a Body Parser. The PUT and POST requests send data inside a body message in JSON format string. What the body parse does is to parse the JSON into a JavaScript object so we can access the fields, like reg.body.title.
We first need to install the cors and body parser modules using NPM

TODO:

You need to install **cors** module as follows:

```
npm install cors
npm WARN mern_lesson2@1.0.0 No description
npm WARN mern_lesson2@1.0.0 No repository field.

+ cors@2.8.5
added 2 packages from 2 contributors and audited 83 packages in 1.634s
found 0 vulnerabilities
```

You will need to install **body-parser** module as follows:

```
npm install body-parser
npm WARN server@1.0.0 No description
npm WARN server@1.0.0 No repository field.

+ body-parser@1.19.0
updated 1 package and audited 51 packages in 1.21s
found 0 vulnerabilities
```

We now need to set up our server to handle routing, body parser and **Cross-Origin Resource Sharing** (CORS)

```
const bodyParser = require('body-parser')
const cors = require('cors')
const Router = require('./routes/router')
```

todo: add the above requires in your server.js file in the same place as the other requires.

Using CORS allows our React Client running Port 3000 to talk to our server at port 4000.

```
// allow port 3000 to connect to this server
var corsOptions = {
  origin: "http://localhost:3000"
};

// enable Cross-Origin Resource Sharing (CORS)
// allows other port to connect to this server
app.use(cors(corsOptions));
```

todo: add the cors code right after const apiPort = 4000

Next we want the express module to use the body parser. The body parser extracts the incoming portion of the incoming request and attaches it to the request object.

```
app.use(bodyParser.json())
```

The line app.use(bodyParser.json()) above tells the Express to use **json** when body parsing.

Our react client is sending the body message in **json** format.

```
// the body parser extracts the entire body portion of an incoming
// request stream and makes it available on req.body.
app.use(bodyParser.urlencoded({ extended: true }))
```

The extended: true above means that the req.body object will contain values of any type instead of just strings.

To do: type in or copy paste the above two lines right after the cors code

The last thing we need to do is tell the express to use the router.

```
// use the router starting with the api prefix
app.use('/api', Router)
```

Our base url routing is "http://localhost:4000/api"

Here is the complete updated server code:

```
// server.js
// node js server
// node js server connects a client to the mongodb
// data base user and book table

// import required modules
const express = require('express')
const bodyParser = require('body-parser')
const cors = require('cors')
const db = require('./db/db')
const Router = require('./routes/router')


// express is a framework that works with node.js
// to handle multiple types of requests
// like the GET, PUT, and POST and DELETE requests.
const app = express()

// use port 4000
const apiPort = 4000


// allow port 3000 to connect to this server
var corsOptions = {
    origin: "http://localhost:3000"
};
```

```
// enable Cross-Origin Resource Sharing (CORS)
// allows other port to connect to this server
app.use(cors(corsOptions));

//app.use(cors()) // allows all ports
app.use(bodyParser.json())

// the body parser extracts the entire body portion
// of an incoming request stream and exposes it on req.body.
app.use(bodyParser.urlencoded({ extended: true }))


// connect to mongodb data base
db.on('error', console.error.bind
     (console, 'MongoDB connection error:'))

// get server info as a default get requests
app.get('/', (req, res) => {
    res.send('Inventory App Server')
})

// use the router starting with the api prefix
app.use('/api', Router)

// start the server, listen for requests
app.listen(apiPort, () => console.log
      (`Server running on port ${apiPort}`))
```

Testing the Router and Controller

Testing with A web Browser

Note: You can only do GET requests with a web browser,

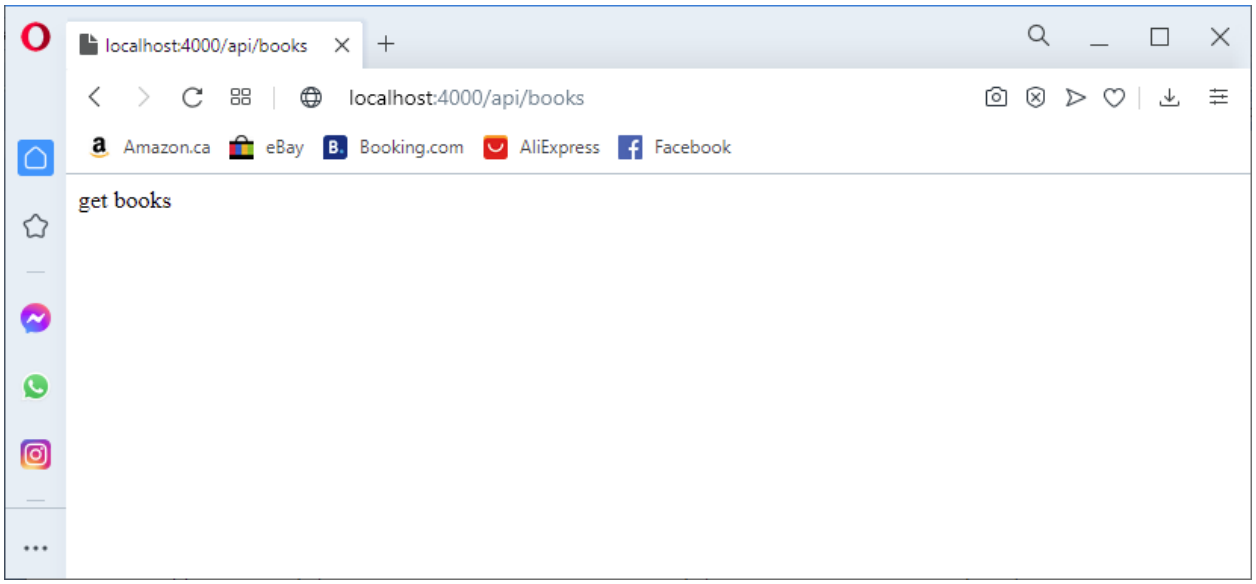
Run the updated server

Node server.js

Open a web browser at http://localhost:4000/api/books

local host port 4000 path /api/books

You should get something like this if everything is working okay:



To do post request you need to down load postman

You can download postman from here:

```
https://www.postman.com/
```

Request	Method	Example	Response
Create book	POST	api/book	Return created book
Get all books	GET	api/books	Return list of books
Get book by id	GET	api/book/:id	Return book for id
Update book	PUT	api/book/:id	Return updated book
Delete book by id	DELETE	api/book/:id	Delete updated book

:id means a book id

The user controller is used for registering a user and for logging in a user.

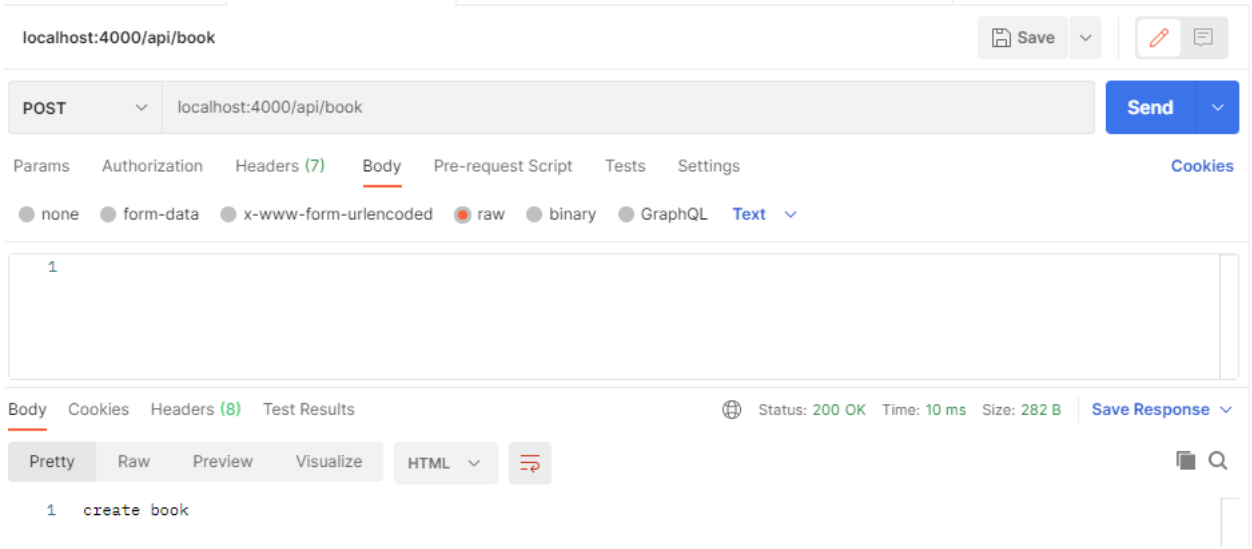
Request	Method	Example	Response
Register user	POST	api/register	Return created user
Login user	POST	api/login	Return logged in user

Testing With Postman

With postman we can do all the GET, PUT, POST and DELETE tests, All you have to do is to type in the request URL, select the Method and press the Send button. The server will respond with our respond test message.

Like: http://localhost:4000/api/book

Todo: try all the POST commands from the above chart



LESSON 5 BOOK CONTROLLER

We now complete the book controller step by step.
We need to complete the following tasks

Request	Method	Example	Response
Create book	POST	api/book	Return created book
Get all books	GET	api/books	Return list of books
Get book by id	GET	api/book/:id	Return book for id
Update book	PUT	api/book/:id	Return updated book
Delete book by id	DELETE	api/book/:id	Delete updated book

Create Book

Create Book request must receive book info in JSON format. The body parser will provide the book info to the request object as a body field. If book data is not provided a 400 status is returned with the error message 'You must provide a book'

We then proceed to create the book on the mongoDB database. If the book cannot be created than a error 400 is returned with the message 'cannot create book'. Finally the book is saved to the mongoDB database. If successful a 201 status is returned with the message 'Book created!' if not successful then a 400 status is returned with the message 'Book not created!'. The book object uses a promise. If everything is okay then the promise calls its **then** function else it calls its catch function. When the promise calls its **then** function it calls our callback function () (a anonymous function without a name and no parameters). Our callback function returns the book object with the status message 201 (resource created). Our catch callback returns the error mongodb error message and status 400 and message Book not created.

```
// create book
// router.post('/book', BookCtrl.createBook)
createBook = (req, res) => {
   console.log("create  book")
   // get book info
   const body = req.body
   console.log(body);

   // validate book info
   if (!body) {
      return res.status(400).json({
         success: false,
         error: 'You must provide a book',
      })
   }

   // make new book
   const book = new Book(body)
   console.log(book);

   // book created
   if (!book) {
      return res.status(400).json({ success: false, error:  ' cannot create book'})
   }
   // save book in data base
   book
      .save()
      .then(() => {
         console.log("Book created!");
         return res.status(201).json({
            success: true,
            id: book._id,
            message: 'Book created!',
         })
      }).catch(error => {
         console.log("Book not created!");
         return res.status(400).json({
            error,
            message: 'Book not created!',
         })
      })
}
```

To do: put the above code in your book controller paste over the CreateBook function.

Then restart the node.js server, fix up any errors if they arise.

You can test the CreateBook request right away using Postman using POST and the following JSON Body message In Postman select method POST, Body and RAW, and JSON data.

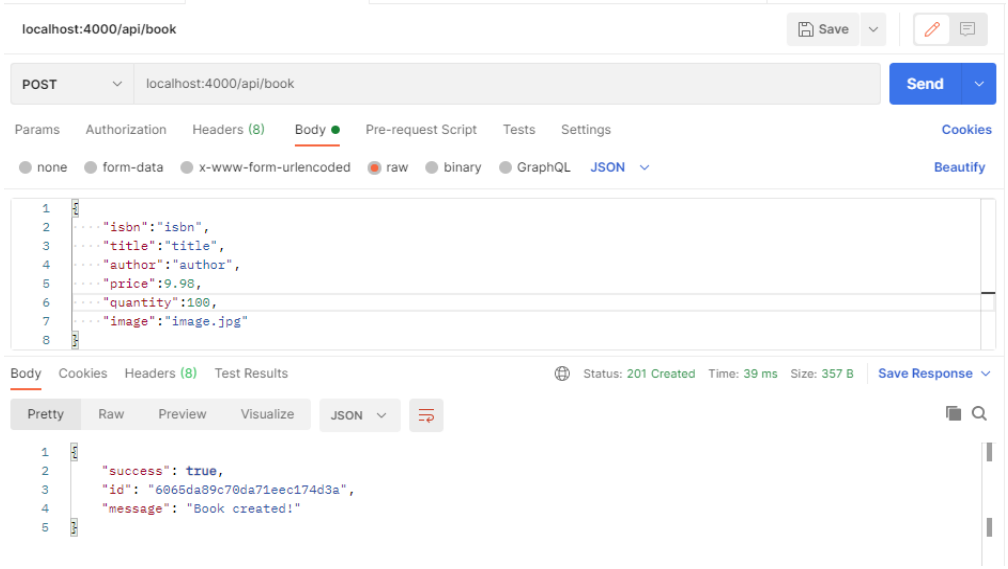
Use this url in postman that will create a book:

```
localhost:4000/api/book
```

Put this in the body section of Postman:

```
{
    "isbn":"isbn",
    "title":"title",
    "author":"author",
    "price":9.98,
    "quantity":100,
    "image":"image.jpg"
}
```

Make sure you select method POST, Body and RAW, and JSON data.
Here is the Postman Results:



UPDATE BOOK

The Update Book request must receive the book id as a parameter in the URL and the updated book info in JSON format as a body message. The entire data base record is rewritten to the data base that will contain updated and old book data. The body parser will provide the book info to the request object as a body field. If book data is not provides a 400 status is returned with the error message 'You must provide a body to update'

Next we find the book in the data base using the supplied parameter id in the url which id the is the id on the book in the data base.

If the book cannot be found then a 404 status message is retuned with the message 'Book not found!',

We then copy the book data from the req.body book object to the found book object that is to be written back to the database.

We then proceed to save update the book. If the book cannot be saved than a error 404 is returned with the message 'Book not updated!'. If successful a 200 status and the message 'Book updated!' is returned. The **async** keyword waits for the updateBook operation to complete.

```
// update book
// router.put('/book/:id', BookCtrl.updateBook)
updateBook = async (req, res) => {

  console.log("update  book")

  // get updated book info
  const body = req.body

  // validate book info
  if (!body) {
    return res.status(400).json({
      success: false,
      error: 'You must provide a body to update',
    })
  }
```

```
    // find book in data base
    Book.findOne({ _id: req.params.id }, (err, book) => {
      if (err) {
        return res.status(404).json({
          err,
          message: 'Book not found!',
        })
      }

      // update existing book
      book.isbn = body.isbn
      book.title = body.title
      book.author = body.author
      book.price = body.price
      book.quantity = body.quantity
      book.image = body.image

      // update book in data base
      book
        .save()
        .then(() => {

          // return success
          return res.status(200).json({
            success: true,
            id: book._id,
            message: 'Book updated!',
          })
        })
        .catch(error => {
          return res.status(404).json({
            error,
            message: 'Book not updated!',
          })
        })
    })
  }
```

To do: put the above code in your book controller paste over the previous update book function. Then restart the node.js server, fix up any errors if they arise.

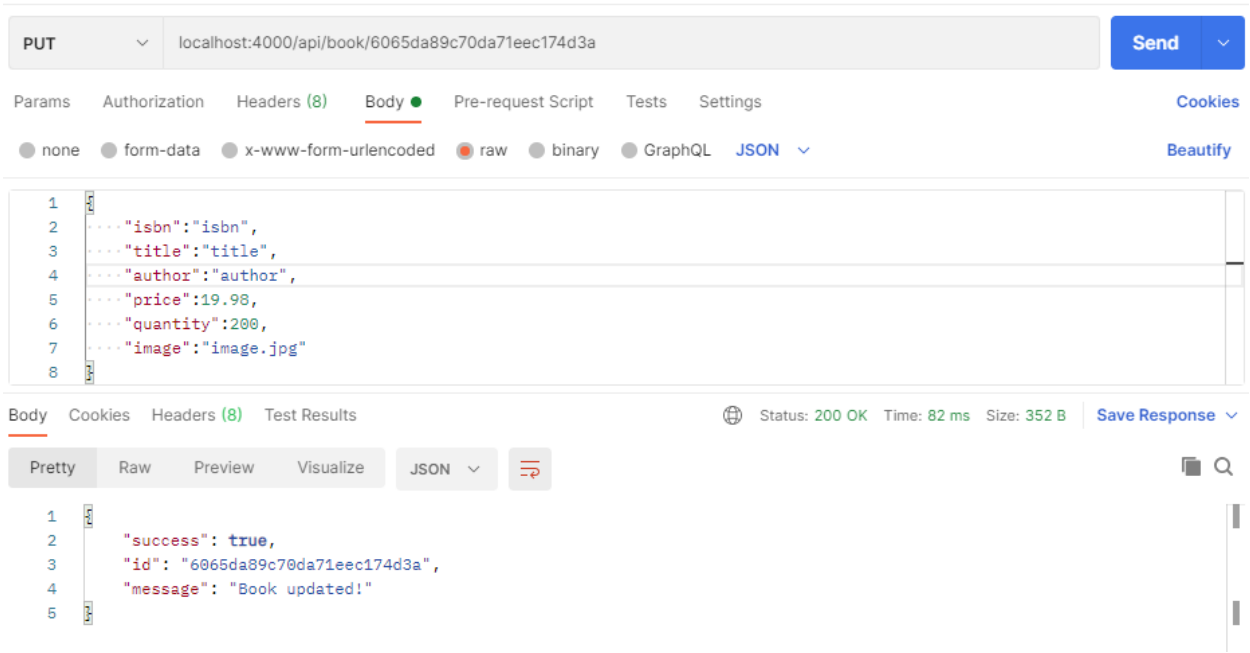
You can test the update book function right away using Postman and using the PUT method and the following JSON Body message In Postman select method PUT, Body and RAW, and JSON data. Use this URL to update the book using the precious create book id.

localhost:4000/api/book/6065da89c70da71eec174d3a

Put this in the body section of Postman to update price and quantity.

```
{
    "isbn":"isbn",
    "title":"title",
    "author":"author",
    "price":19.98,
    "quantity":200,
    "image":"image.jpg"
}
```

Make sure you select method PUT, Body and RAW, and JSON data.



GET BOOK by ID

The Get Book ID request must receive the book id as A parameter in the URL, we then find the book in the data base using the supplier request id which is the id in the data base for that book record. If the book cannot be found a 400 status is returned along with the error message. If no book is retrieved from the data base then a 404 status message is retuned with the message 'Book not found!'. Else a 200 status is returned along with the book data. The **async** keyword waits for the getBookById operation to complete.

```
// get book by id
// router.get('/book/:id', BookCtrl.getBookById)
getBookById = async (req, res) => {

  console.log("get book by id")

  // find book by id
  await Book.findOne({ _id: req.params.id }, (err, book) => {
    if (err) {
      return res.status(400).json({ success: false, error: err })
    }

    // book no found
    if (!book) {
      return res
        .status(404)
        .json({ success: false, error: `Book not found` })
    }

    // return success
    return res.status(200).json({ success: true, data: book })
  }).catch(err => console.log(err))
}
```

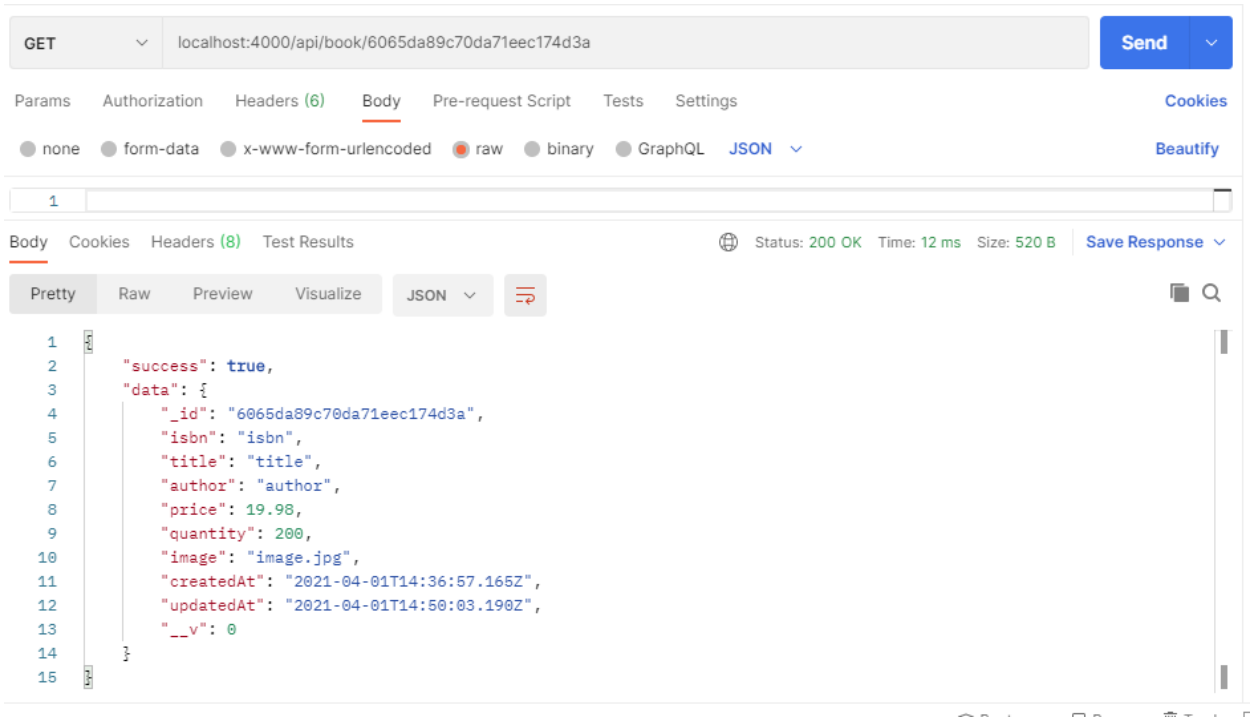
To do: Place the above code in your book controller paste over the previous get book by id function.

Then restart the node.js server, fix up any errors if they arise.

You can test get **book by id** function right away using Postman using method GET and the following url

```
localhost:4000/api/book/6065da89c70da71eec174d3a
```

Make sure you select method GET



GET ALL BOOKS /books

Once the /books request is received all the books from the mongodb data base is retrieved using the find command and the query {}. If an error occurs a 400 status message and error is returned. If no books are retrieved then a 404 status message is returned with the message no 'books available`, otherwise a 200 status message is returned and the book data. getBookById. The **async** keyword waits for the getBooks operation to complete.

```
// return all books
// router.get('/books', BookCtrl.getBooks)
getBooks = async (req, res) => {

   console.log("get books");

   // get all books in table {}
   await Book.find({}, (err, books) => {
      if (err) {
         return res.status(400).json({ success: false, error: err })
      }
      // no books in database
      if (!books.length) {
         return res
            .status(404)
            .json({ success: false, error: `no books available` })
      }

      // return success
      return res.status(200).json({ success: true, data: books })
   }).catch(err => console.log(err))
}
```

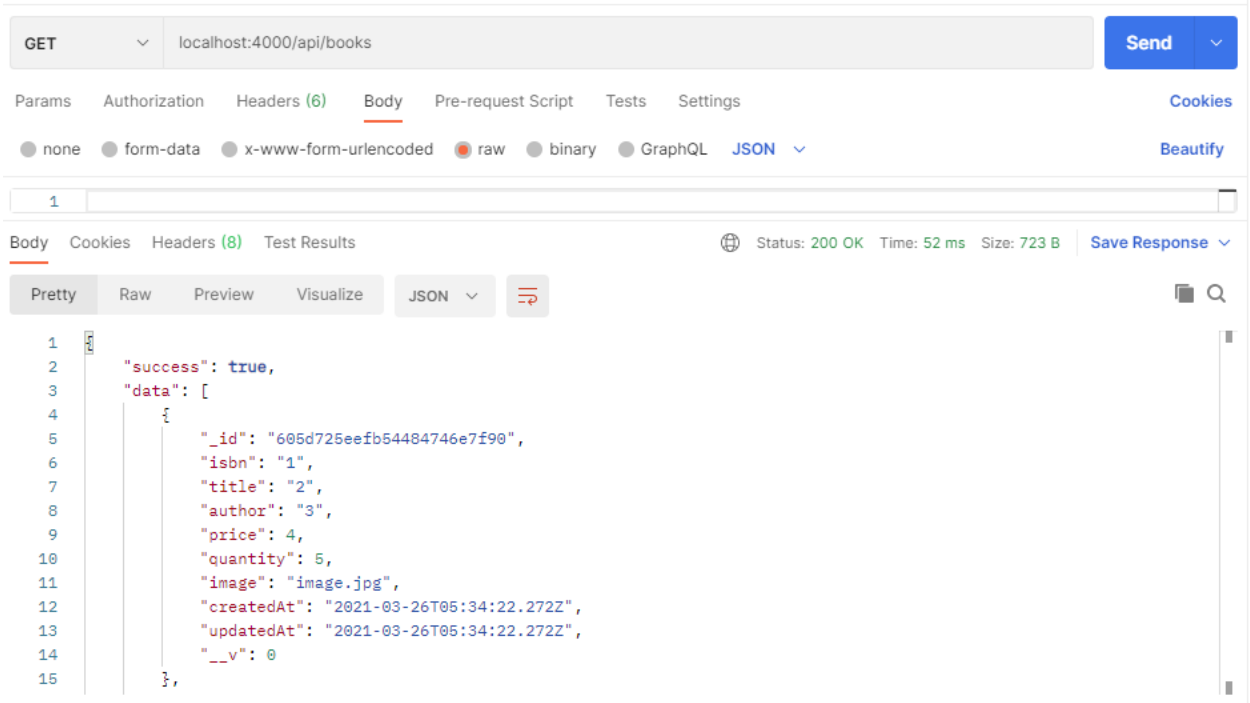
To do: Place the above code in your book controller paste over the previous get book by id function.

Then restart the node.js server, fix up any errors if they arise.

You can test right away using Postman using GET and the following url

```
localhost:4000/api/books
```

Make sure you select method GET



DELETE BOOK by ID

The Delete Book ID request must receive the book id as a parameter in the URL, We then use the book id and the mongoDB findOneAndDelete method to delete the book. If an find error occurs a 400 status is return along with the error message. If no book is find from the data base then a 404 status message is retuned with the message 'Book not found!'. Else a 200 status is returned along with the book data that was deleted. The **async** keyword waits for the deleteBook operation to complete.

```
// delete book
// router.delete('/book/:id', BookCtrl.deleteBook)
deleteBook = async (req, res) => {
  console.log("delete book")

  // find and delete book in data base
  await Book.findOneAndDelete({ _id: req.params.id }, (err, book) => {
    if (err) {
      console.log("400 error: " + err);
      return res.status(400).json({ success: false, error: err })
    }
```

```
    // book not found
     if (!book) {
        console.log("404 error: " + 'Book not found');
        return res
          .status(404)
          .json({ success: false, error: `Book not found` })
    }

    // print deleted book details
    console.log(book)

    // return success
    return res.status(200).json({ success: true, data: book })
  }).catch(err => console.log(err))
}
```

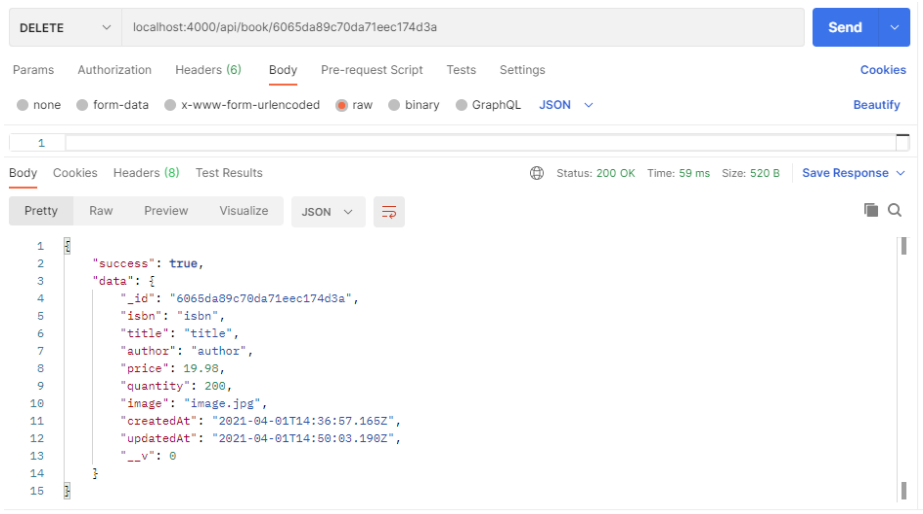
To do: Place the above code in your book controller paste over the previous get book by id function.

Then restart the node.js server, fix up any errors if they arise.

You can test right away using Postman using DELETE and the following url

```
localhost:4000/api/book/6065da89c70da71eec174d3a
```

Make sure you select method DELETE



Here is the complete Book Controller code:

```
// book-ctrl.js
// book controller
// handles all book data base requests from client

// book table
const Book = require('../models/book-model')

// create book
// router.post('/book', BookCtrl.createBook)
createBook = (req, res) => {

   console.log("create  book")

   // get book info
   const body = req.body
   console.log(body);

   // validate book info
   if (!body) {
      return res.status(400).json({
         success: false,
         error: 'You must provide a book',
      })
   }

   // make new book
   const book = new Book(body)
   console.log(book);

   // book created
   if (!book) {
      return res.status(400).json({ success: false, error: err })
   }

   // save book in data base
   book
      .save()
      .then(() => {
         console.log("Book created!");
         return res.status(201).json({
```

```
            success: true,
            id: book._id,
            message: 'Book created!',
          })
        })
        .catch(error => {
          console.log("Book not created!");
          return res.status(400).json({
            error,
            message: 'Book not created!',
          })
        })
}

// update book
// router.put('/book/:id', BookCtrl.updateBook)
updateBook = async (req, res) => {

  console.log("update  book")

  // get updated book info
  const body = req.body

  // validate book info
  if (!body) {
    return res.status(400).json({
      success: false,
      error: 'You must provide a body to update',
    })
  }

  // find book in data base
  Book.findOne({ _id: req.params.id }, (err, book) => {
    if (err) {
      return res.status(404).json({
        err,
        message: 'Book not found!',
      })
    }
```

```
      // update existing book
      book.isbn = body.isbn
      book.title = body.title
      book.author = body.author
      book.price = body.price
      book.quantity = body.quantity
      book.image = body.image
      // update book in data base
      book
        .save()
        .then(() => {

          // return success
          return res.status(200).json({
            success: true,
            id: book._id,
            message: 'Book updated!',
          })
        })
        .catch(error => {
          return res.status(404).json({
            error,
            message: 'Book not updated!',
          })
        })
    })
}

// delete book
// router.delete('/book/:id', BookCtrl.deleteBook)
deleteBook = async (req, res) => {
  console.log("delete book")

  // find and delete book in data base
  await Book.findOneAndDelete({ _id: req.params.id }, (err, book) => {
    if (err) {
      console.log("400 error: " + err);
      return res.status(400).json({ success: false, error: err })
    }
```

```
      // book not found
       if (!book) {
          console.log("404 error: " + 'Book not found');
          return res
            .status(404)
            .json({ success: false, error: `Book not found` })
      }

      // print deleted book details
      console.log(book)

      // return success
      return res.status(200).json({ success: true, data: book })
   }).catch(err => console.log(err))
}

// get book by id
// router.get('/book/:id', BookCtrl.getBookById)
getBookById = async (req, res) => {

   console.log("get book by id")

   // find book by id
   await Book.findOne({ _id: req.params.id }, (err, book) => {
      if (err) {
         return res.status(400).json({ success: false, error: err })
      }

      // book not found
      if (!book) {
         return res
            .status(404)
            .json({ success: false, error: `Book not found` })
      }

      // return success
      return res.status(200).json({ success: true, data: book })
   }).catch(err => console.log(err))
}
```

```
// return all books
// router.get('/books', BookCtrl.getBooks)
getBooks = async (req, res) => {
   console.log("get books");

   // get all books in table {}
   await Book.find({}, (err, books) => {
      if (err) {
         return res.status(400).json({ success: false, error: err })
      }
      // no books in database
      if (!books.length) {
         return res
            .status(404)
            .json({ success: false, error: `no books available` })
      }

      // return success
      return res.status(200).json({ success: true, data: books })
   }).catch(err => console.log(err))
}

// export all book controller functions
module.exports = {
   createBook,
   updateBook,
   deleteBook,
   getBooks,
   getBookById,
}
```

LESSON 6 USER CONTROLLER

We now complete the User controller step by step.
We need to complete the following tasks

Request	Method	Example	Response
Register user	POST	api/register	Return created user
Login user	POST	api/login	Return logged in user

REGISTER USER

The Register User request must receive user info in JSON format. The body parser will provide the user registration info to the request object as a body field. If user registration data is not provided a 400 status is returned with the error message 'You must provide a user' We then proceed to register the user on the mongoDB database. If the user cannot be created than a error 400 is returned with the message 'cannot register user'. Finally the user registration is saved to the mongoDB database. If successful a 201 status is returned with the message 'User registered!' if not successful then a 400 status is returned with the message User not registered!'. The **async** keyword waits for the getBookById operation to complete.

```
// register user
// router.post('/register', UserCtrl.registerUser)
registerUser =   async(req, res) => {
   console.log("register  user")

   // get registration info
   const body = req.body
   console.log(body);

   // validate registration info
   if (!body) {
      return res.status(400).json({
         success: false,
         error: 'You must provide a user',
      })
   }
```

```
  // make new user
  const user = new User(body)
  console.log(user);

  // user not created
  if (!user) {
    return res.status(400).json({ success: false, error: err })
  }

  // save user in data base
  user
    .save()
    .then(() => {
      console.log("User registered!");
      return res.status(201).json({
        success: true,
        id: user._id,
        message: 'User registered!',
      })
    })
    .catch(error => {
      console.log("User not registered!");
      return res.status(400).json({
        error,
        message: 'User not registered!',
      })
    })
}
```

To do: put the above code in your book controller paste over the RegisterUser function.

Then restart the node.js server, fix up any errors if they arise.

You can test the RegisterUser request right away using Postman using POST and the following JSON Body message In Postman select method POST, Body and RAW, and JSON data

Use this POST url in postman that will register a user

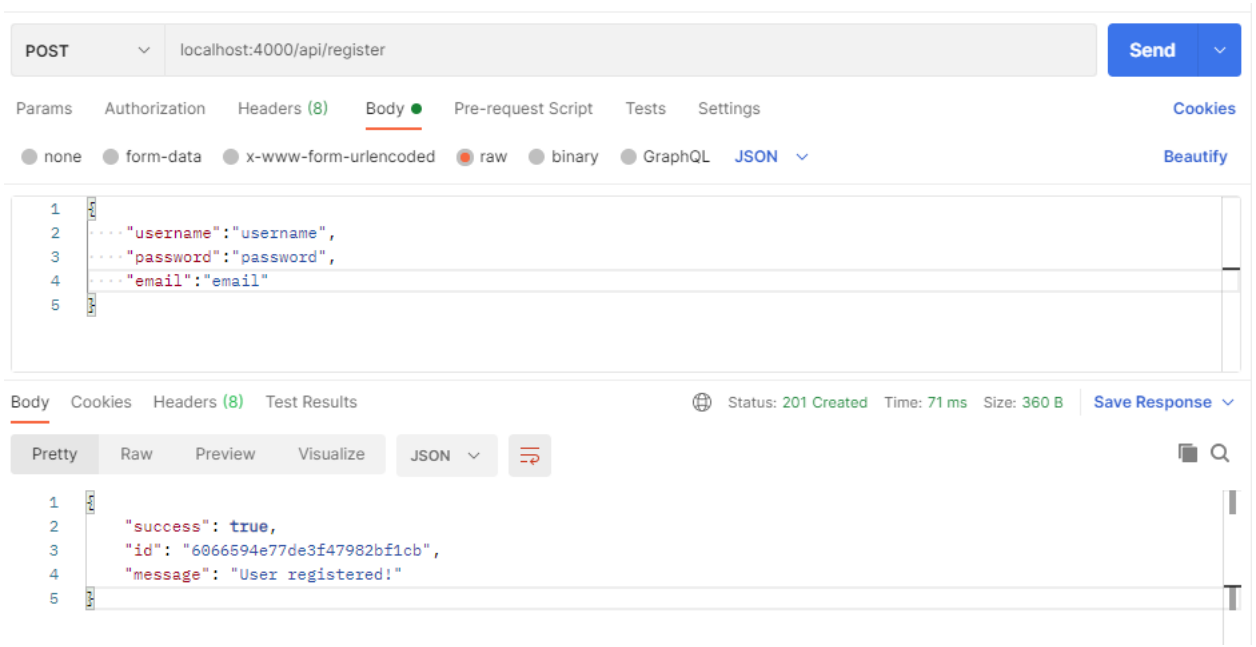
localhost:4000/api/register

Put this in the body section of Postman:

```
{
    "username":"username",
    "password":"password",
    "email":"email"
}
```

Make sure you select method POST, Body and RAW, and JSON data.

Here is the Postman Results:



LOGIN USER

The Login User request the login info username and password through a body message. The mongodb findOne function is used to look up the user. If an error occurs then a 400 status and error message is returned. If the user is not found then a 404 status and User not found message is returned. If the user is found then a 200 status user data is returns as a authorization token. The **async** keyword waits for the getBookById operation to complete.

```
// login user
// router.post('/login', UserCtrl.loginUser)
loginUser = async (req, res) => {

   console.log("login: " + JSON.stringify(req.body))

   // find user in data base
   await User.findOne( req.body , (err, user) => {
      if (err) {
         return res.status(400).json({ success: false,  error: err })
      }
     console.log(user)

      // user not found
      if (!user) {
         return res
            .status(404)
            .json({ success: false, error: `User not found` })
      }

    // success, return user id as a token
      return res.status(200).json({ success: true, token: user._id, data: user })
   }).catch(err => console.log(err))
}
```

To do: put the above code in your book controller paste over the loginUser function. Then restart the node.js server, fix up any errors if they arise.

You can test the Login request right away using Postman using POST and the following JSON Body message In Postman select method POST, Body and RAW, and JSON data

Use this POST url in postman that will login a user

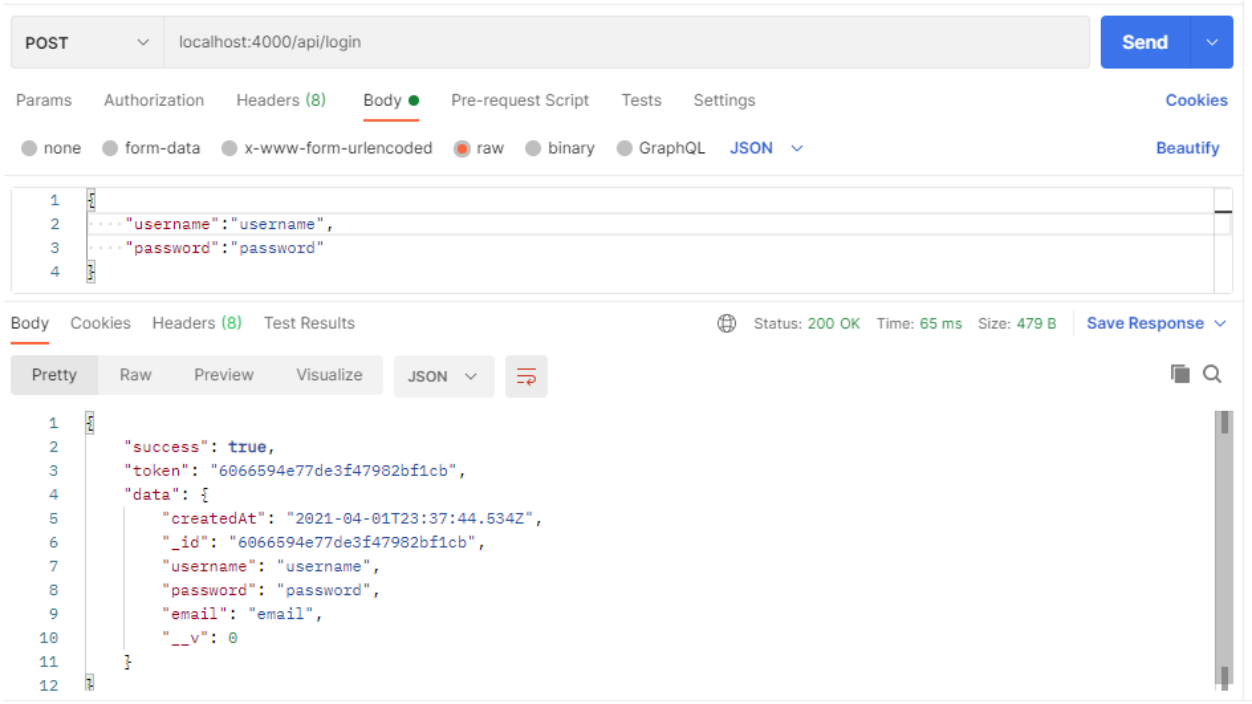
localhost:4000/api/login

Put this in the body section of Postman:

```
{
    "username":"username",
    "password":"password"
}
```

Make sure you select method POST, Body and RAW, and JSON data.

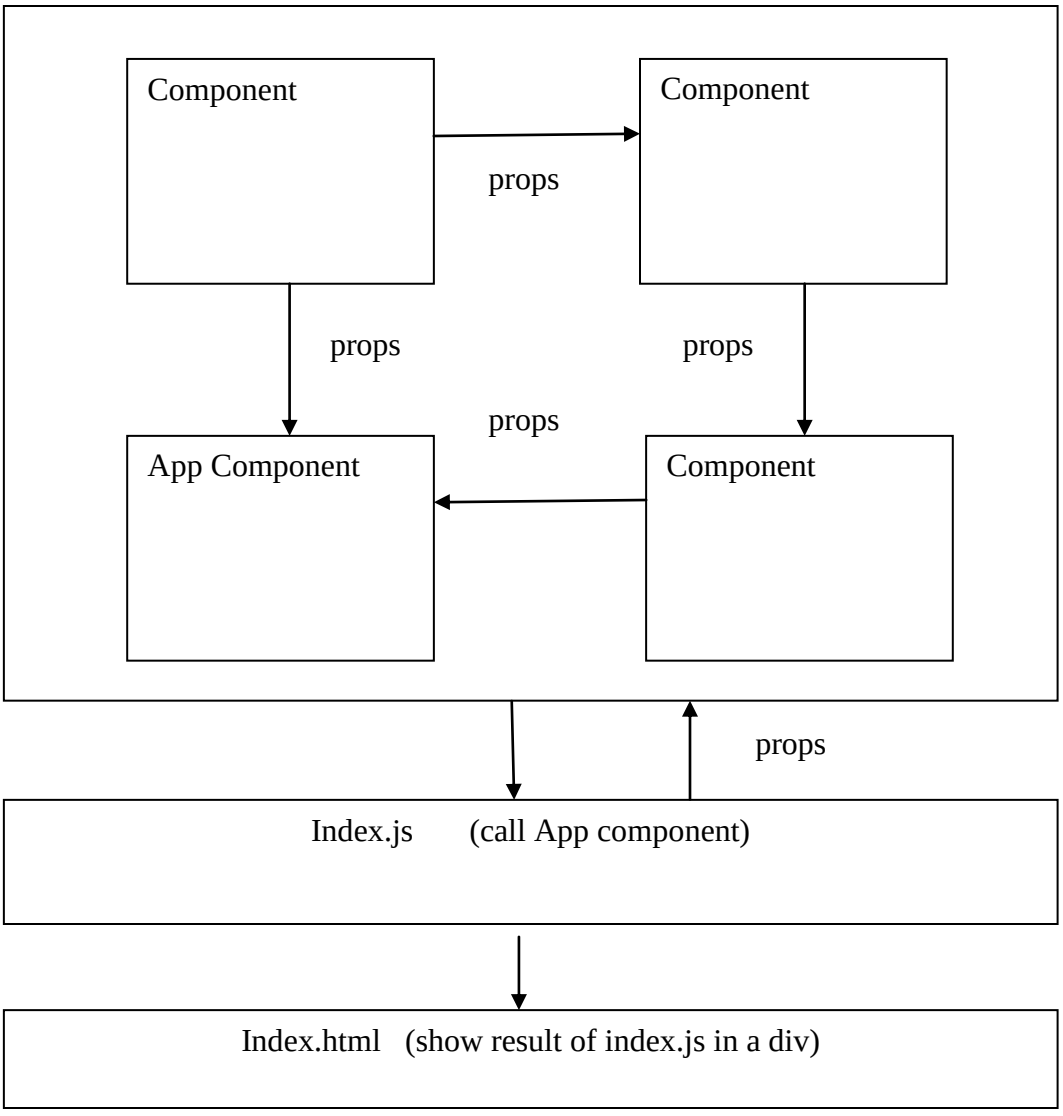
Here is the Postman Results:



LESSON 7 REACT CLIENT AND INSTALLING REACT

Lesson1 Introduction to React

React let's you create web pages using components. Components contain data values known as **states**. Components also send data known as **props** (properties) to other components. Components display web content on the screen, using its **render** method. Components are like building blocks. A web page using React components can be visualized like this:



Installing React

You need node.js and NPM to install React We already have installed node.js in Lesson 1.

Step 1 Install node.js (if you do not have nodejs installed on your computer)

https://nodejs.org/en/download/

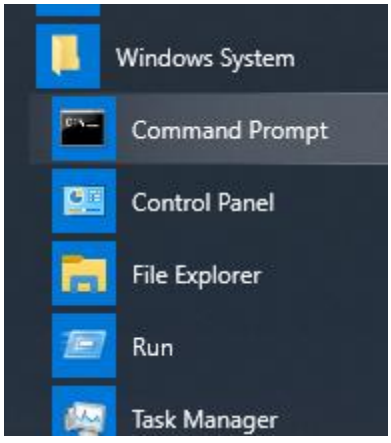
On our windows machine we download the windows installer msi file, and installs with no problems. If you all ready have node js installed on your computer you may want to reinstall to the latest version. React only work on node.js versions 10 or greater.

Node.js is an open source server environment.
Node.js allows you to run JavaScript on the Node.js server.
NPM is used to create and run React apps and to install additional modules.

NPM is a package manager for Node.js packages and modules.
The NPM program is installed on your computer when you install Node.js
React uses node.js modules to create react apps and also uses modules provided by node.js

Step 2 install create-react-app

From the start menu under Windows System open up a Command prompt



Navigate to the root of your C drive

C:\Users\admin>cd ..
C:\Users>cd ..

If you do not have a inventory-app folder then make one

mkdir inventory-app

if you do have one navigate to it

cd inventory-app

In the command prompt window type on:

npm install -g create-react-app

This installs the create-react-app module into the nodes modules, so you can create react apps but does not create the react app,

D:\inventory-app>**npm install -g create-react-app**
C:\Users\ADMIN\AppData\Roaming\npm\create-react-app ->
C:\Users\ADMIN\AppData\Roaming\npm\node_modules\create-react-app\index.js
+ create-react-app@4.0.3
updated 1 package in 4.108s

Step 3 create react app

On the windows prompt command line type

npx create-react-app client

It will start out like this:

```
D:\inventory-app>npx create-react-app client
```

If successful you will end up something like this:

```
Creating a new React app in D:\inventory-app\client.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

removed 1 package and audited 1954 packages in 28.885s
found 0 vulnerabilities

Success! Created client at D:\inventory-app\client
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd client
  npm start

Happy hacking!

D:\inventory-app>
```

Step 4: run client

cd client
npm start

You will get something like this:

Compiled successfully!

You can now view client in the browser.

Local: http://localhost:3000
On Your Network: http://192.168.2.12:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

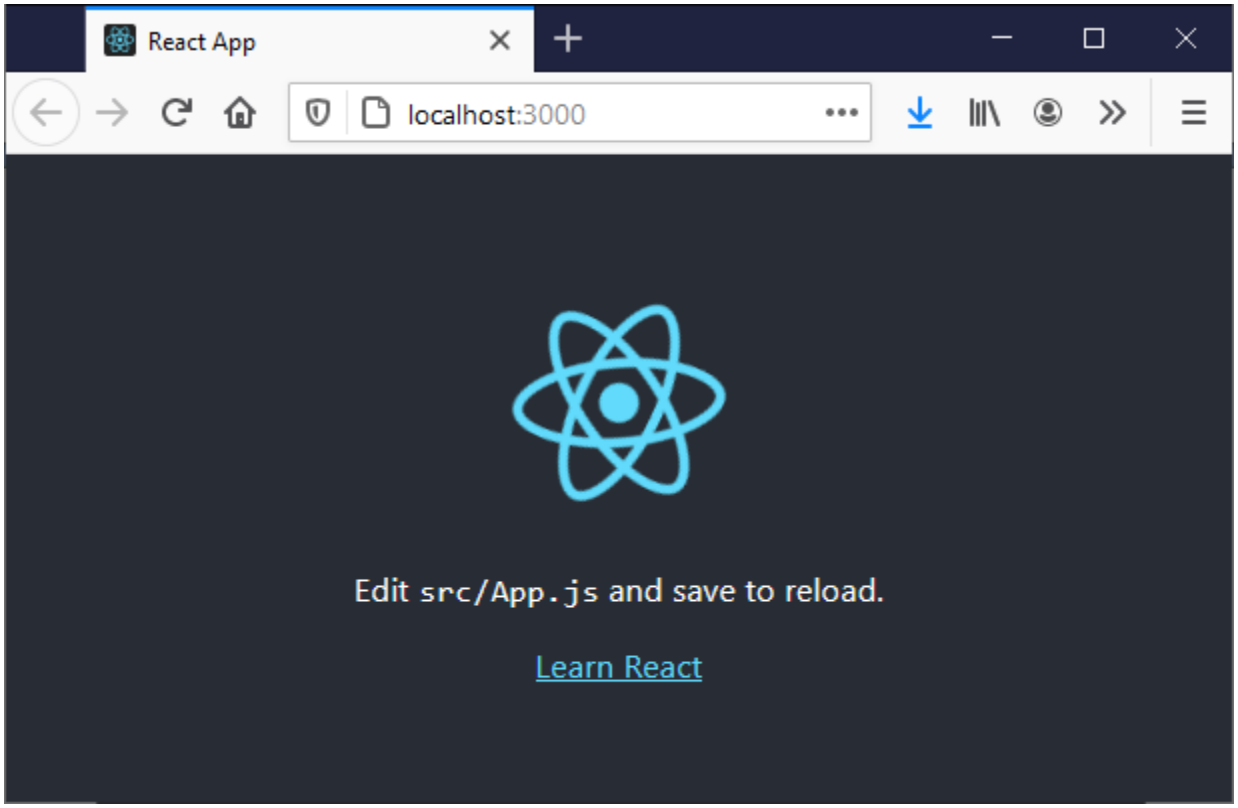
Step 5 view react client

Start your web browser at localhost: 3000

http://localhost:3000/ actually is calling http://localhost:3000/index.html

Usually this is an automatic process, react usually automatically starts the web browser opened at localhost:3000. If it does not start automatically then go to your web browser ant type localhost:3000 in the address bar.

You should get something like this:

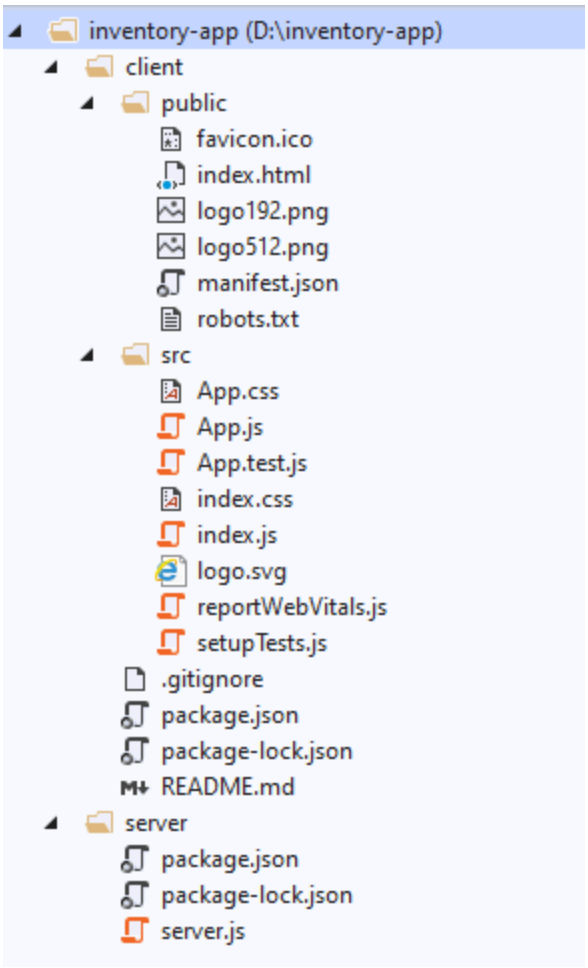


If you do not then repeat all the above steps.

Client app file structure

The Client App has created many files. The public folder contains **index.html** what the web browser runs where as the src folders contains the **index.js** file that is used to display the web content rendered from the react components. The node_modules contain all the modules that react will use. (too many files to show.

Our Inventory App project now looks like this with the React Client installed:



Basic Folder Structure Explained

1. **package.json:** This File has the list of node dependencies which are needed.

2. **public/index.html:** When the application starts this is the first page that is loaded. This will be the only html file in the entire application since React is generally written using **JSX** which I will explain later. Also, this file has a line of code ** **. All the application components are loade**d i**nto this div.

3. **src/index.js**: This is the javascript file corresponding to index.html. This file has the following line of code which is used to call App component to display. **ReactDOM.render(<App />, document.getElementById('root'));**

The above line of code is telling that **App** Component to be loaded into an html div element with id **root** located into the **index.html** file.

4. **src/index.css**: The CSS file corresponding to index.js.

5. **src/App.js** : This is the file for **App** Component. **App** Component is the main component in React which acts as a container for all other components.

6. **src/App.css** : This is the CSS file corresponding to **App** Component

7. **build:** This is the folder where the built files are stored. React Apps can be developed using either JSX, or normal JavaScript itself, but using JSX definitely makes things easier to code for the developer :). But browsers do not understand JSX. So JSX needs to be converted into javascript before deploying. These converted files are stored in the build folder after bundling and minification.

Here are the index.html. index.js and App.js files created by React. The App.js file renders the webpage contents and uses index.js to send it to the index.html file to be displayed in an div element.

// index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <!--
      manifest.json provides metadata used when your web app is installed on a
      user's mobile device or desktop.
      See https://developers.google.com/web/fundamentals/web-app-manifest/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
```

```
      <!--
      Notice the use of %PUBLIC_URL% in the tags above.
      It will be replaced with the URL of the `public` folder during the build.
      Only files inside the `public` folder can be referenced from the HTML.

      Unlike "/favicon.ico" or "favicon.ico", "%PUBLIC_URL%/favicon.ico" will
      work correctly both with client-side routing and a non-root public URL.
      Learn how to configure a non-root public URL by running `npm run build`.
    -->


<title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

      You can add webfonts, meta tags, or analytics to this file.
      The build step will place the bundled scripts into the <body> tag.

      To begin the development, run `npm start` or `yarn start`.
      To create a production bundle, use `npm run build` or `yarn build`.
    -->
  </body>
</html>
```

The index.js file basically just shows the react content by rendering the React App component which shows the react logo image.

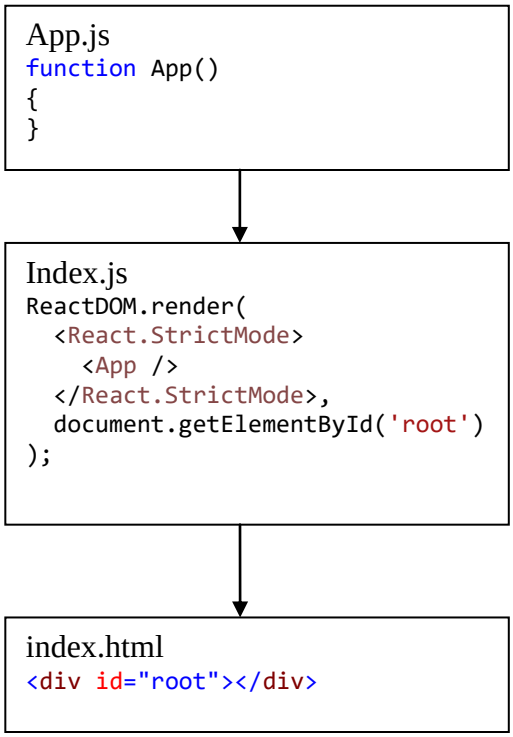
// **index.js**

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

App.js for now just displays the React logo

```
// App.js

import logo from './logo.svg';
import './App.css';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.js</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

export default App;
```

Here is the program code flow



The index.js file renders the App component using the ReactDOM.render function.

```
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Using the <App /> component and a reference to the root div located in the index.html file.
React.StrictMode is used for highlighting potential problems in an application.
document.getElementById('root') is the element that is used to display the App.

Lesson 8 React Components

React components are used to build web pages. A Component display web page content. Components can be made from JavaScript classes or JavaScript functions.

React Components made from JavaScript classes are known as React Class Components.

React Components made from JavaScript functions are known as React Function Components.

A Component combines HTML tags and JavaScript to render content.
A React Class Component has a explicit **render** method where as a React Function Component does not and implicitly renders. The **render** method in the React Class Component returns HTML tags converted to JavaScript code that can be rendered.

Like wise the React Function Component returns HTML tags converted to JavaScript code that can also be rendered. Rendering means displaying HTML tags on a web browser.

Here is a simple React Class Component that renders the message "I like React" using a <h1> tag.

```
import React, { Component } from 'react';

class MyApp extends Component {
    render() {
        return (

                <div>
                    <h1>I Like React</h1>
                </div>

        );
    }
}
```

Here is React Component function that renders "I like React" also inside a <h1> tag.

```
function MyApp() {
  return (
    <div>
          <h1>I like React</h1>
    </div>
  );
}
```

The **return** keyword returns the HTML tags converted to JavaScript code that can render content.

Here is a functional component using the arrow function definition that renders "I like React". The arrow function is similar to the JavaScript function but omits the **function** keyword and the function code follows the => (arrow). They both do the same thing that renders "I like React" in a <h1> tag. The arrow function is just a short compact form of a JavaScript function definition.

```
const MyApp = () => {
    return (
        <div>
            <h1> I Like React </h1>
        </div>
    );
}
```

We use the **const** keyword rather than the **let** or **var** keyword because a function is considered a constant. **const** means code that will not be changed.

A functional component does the same thing a class component does but it is more simpler to code and uses less overhead for React to execute. We use functional components just when we need to renders something simple that does not require **props** and **states**

JavaScript XML (JSX)

The HTML code in the Components render method is known as JSX. JSX is a syntax extension to JavaScript that produces React "elements" that is used for rendering content.
JSX allows us to write HTML tags directly within the JavaScript code. The JSX is translated into JavaScript code down to **React.createElement()** calls.

For example the following html code:

```
<div>
   <h1> I like React</h1>
</div>
```

Would be translated into:

```
React.createElement(
    "div",
    null,
    React.createElement(
        "h1",
        null,
        " I like React"
        )
);
```

JSX always need a top html tag element like , if you want to include additional html tags.

```
<div>
   <h1> I like React</h1>
   <h1> I like Programming</h1>
</div>
```

Rendering a Component

The **ReactDOM.render()** function is used to render a component.

The ReactDOM.render() function takes two arguments, a HTML code or React components and an HTML element.

The purpose of the render function is to display the specified HTML code inside the specified HTML element.

You can render a component in the index.js file by specifying the component name as a HTML tag and calling the **ReactDOM.render method.**

```
ReactDOM.render(
  <React.StrictMode>
    <MyApp />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Alternately you can render JSX hardcoded

```
ReactDOM.render(<div>
    <h1> I like React</h1></div>
, document.getElementById('root'));
```

Or equate JSX to a const variable and render it.

```
const myElement = <div><h1> I like React</h1></div >;
ReactDOM.render(
   myElement
    , document.getElementById('root'));
```

props

Props sends data to other components as parameters. **Props** stand for properties. Components can send props and receive **props**. **Props** cannot be changed once they are received and are read only.

In this example we send a name as a **prop** to the MyApp component.

<MyApp name="Tom Smith" />

Here is the complete render example sending an **prop** to a app

```
ReactDOM.render(
  <React.StrictMode>
    <MyApp name=”Tom Smith” />
  </React.StrictMode>,
  document.getElementById('root')
);
```

The MyApp class component receives the props and renders the name from the props object members.

The name value from the props object is placed in curly brackets like this {pops.name} and the <h2> is now this:

```
<h2>My Name is {props.name} </h2>
```

Our MyApp class component is now this:

```
import React, { Component } from 'react';

class MyApp extends Component {
    render() {
        return (
          <div>
             <h1>I like React</h1>
             <h2>My Name is {props.name}</h2>
          </div>
        );
    }
}
```

Passing props to class components that have constructors

A constructor in a class component is used to receive prop and initialize other variables. If a class component has a constructor then it must receive props.

```
import React, { Component } from 'react';

class MyApp extends Component {

    constructor(props) {
        super(props);
     }


  render() {
    return (

      <div>
        <h1>I like React</h1>
        <h2>My Name is {props.name}</h2>
      </div>

    );
  }
}
```

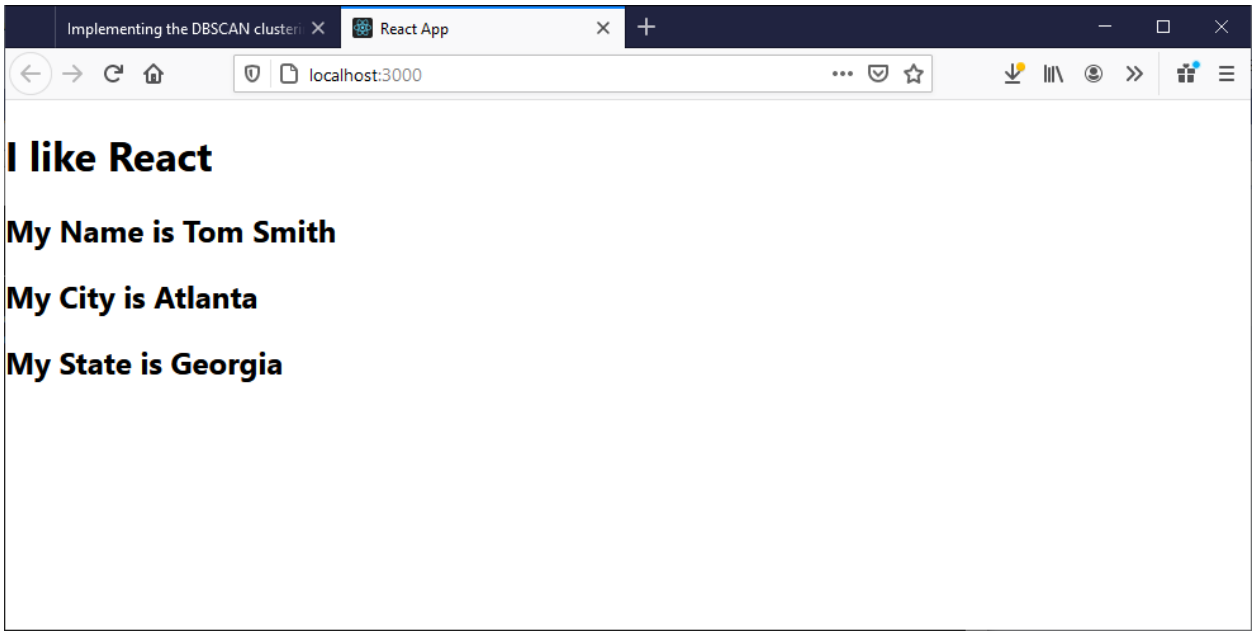
functional components also receive props

```
function MyApp(props) {
 return (
  <div>
      <h1>I like React</h1>
      <h2>My Name is {props.name}</h2>
  </div>
 );
}
```

LESSON8 Homework Question 1

Create the Name App in React. When you call the App component from index.js send the App Component the name, city and states as props. In the App component controller uses the received props display the name, state and city for rendering.

Note:
Although the web browser will automatically update when you make changes to your react code. there may be instances where you want to start fresh.
To return control back to the command prompt: type ctrl C [^C]
Terminate batch job (Y/N)? Y
And you will have the command line back again.
To restart the app type:
npm start
You should get something like this:



States

A class component has a state object that allow a class component to store values. Here is a state object that stores information about a person.

```
this.state = {
      name: "Tom Smith",
      city: "Atlanta",
      state: "Georgia"
    };
```

When the state object changes, the component re-renders by calling the **render** method. Only class components can have states. Functional components do not have states.

reading state values

You can read values from the state object using the state object reference and the member variable, and embed into the HTML tag's like this

```
<h1>Name: {this.state.name} </h1>
```

Or use a constant like this

```
const name = this.state.name
```

then embed the const value into the HTML tag

```
<h1>Name: {name} </h1>
```

Alternatively you can read all the values from the state object into constant variables like this

```
const {name, state, city} = this.state
```

And then put into the HTML tags like this

```
<div>
 <h1>Name: {name} </h1>
 <h1>State: {state} </h1>
 <h1>City: {city} </h1>
</div>
```

Changing state values

To change a value in the state object, use the **setState** method.

When a value in the state object changes, the component will re-render, meaning that the output will change according to the new value(s).

You will change a state value like this:

```
this.setState({name: "Sue Jones"});
```

The state object inside the Person component is used to store information about a Person. The render method will now display all the information about the person when it renders.

```
import React, { Component } from 'react';

class Person extends Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Tom Smith",
      city: "Atlanta",
      state: "Georgia"
    };
  }
  render() {

    const {name, state, city} = this.state

    return (
      <div>
        <h1>Name: {name} </h1>
        <h1>State: {state} </h1>
        <h1>City: {city} </h1>
      </div>
    );
  }
}
```

To use the above Person class it must be called from index.js or another Component like this:

```
<Person />
```

Initializing a Component state with props from another component

There may be situation when you want to initialize state data from props. Props can send data to a component that can be used to initialize it's state. For example we can send data to a person component in the index.js and then the Person component can store this data in its state. The advantage is the props data that is sent to the Components can be stored in its state where it can be displayed and changed for later times. Prop data cannot be changed but state data can.

```
ReactDOM.render(
  <React.StrictMode>
     <Person name="Tom Smith" city="Atlanta" state="Georgia" />
  </React.StrictMode>,
  document.getElementById('root')
);
```

The Person Component will receive the props from the index.js file when the Person component is called. Here is the Person Component that receives props and stores them it is state.

```
import React, { Component } from 'react';

class Person extends Component {
    constructor(props) {
        super(props);
        this.state = {
            name: props.name,
            city: props.city,
            state: props.state
        };
    }
    render() {
        const { name, state, city } = this.state
        return (
            <div>
                <h1>Name: {name} </h1>
                <h1>State: {state} </h1>
                <h1>City: {city} </h1>
            </div>
        );
    }
}
```

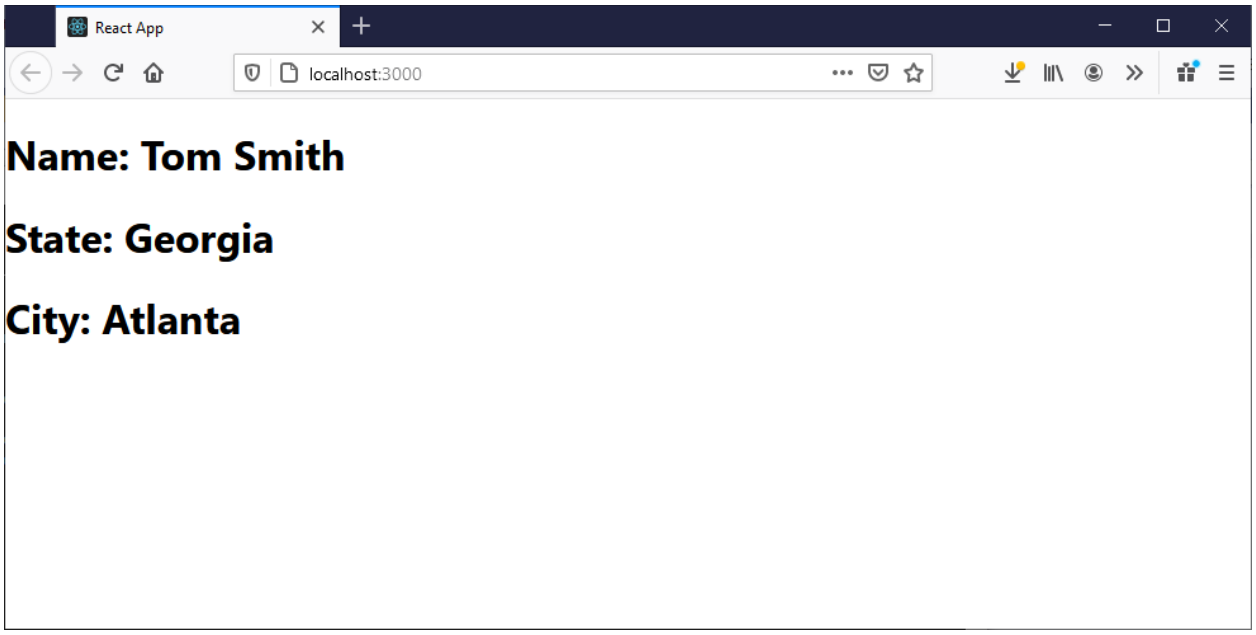
```
export default Person;
```

Running the program you should get something like this:

LESSON6 Homework Question 2

Call the App component from index.js, send the name, city and states as props. In the App component controller send the received props to the Person Component and have the Person Component to store the name, state and city in its state for rendering.

You should get something like this:



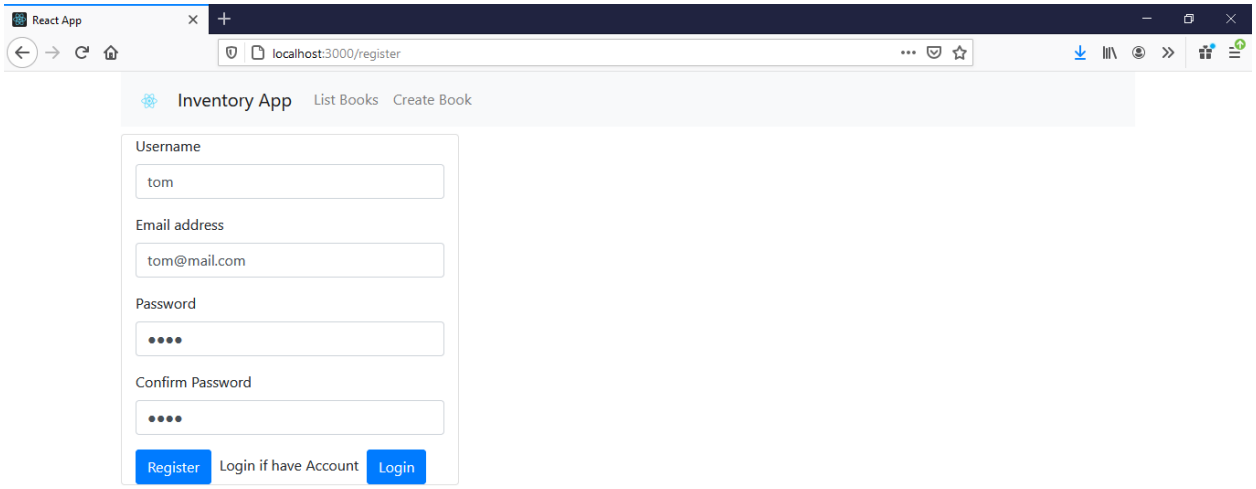
Lesson 7 Client App Components

We will have the following components:

Component	Description
Login	User Login
Register	Register User
DashBoard	Navigate Components
CreateBook	Store Books in Inventory
ListBooks	List Books in Inventory
EditBooks	Edit Books in Inventory
DeleteBook	Delete Books in Inventory
Logout	Logout User

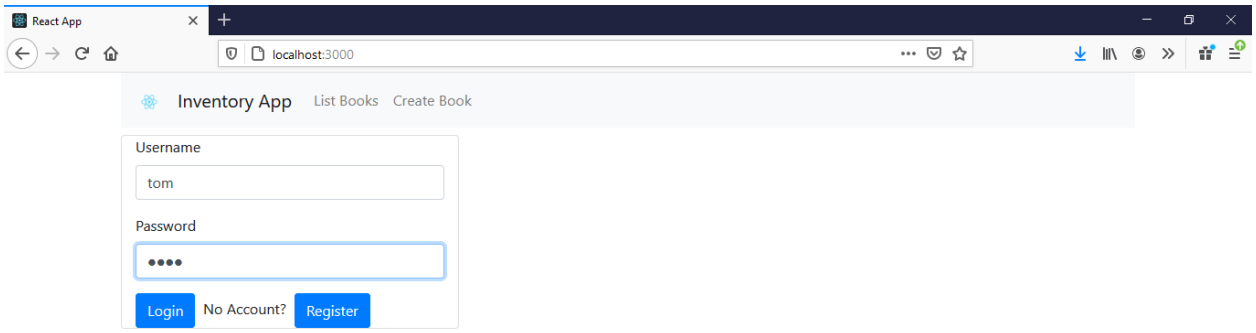
A Register page may look like this:

Register



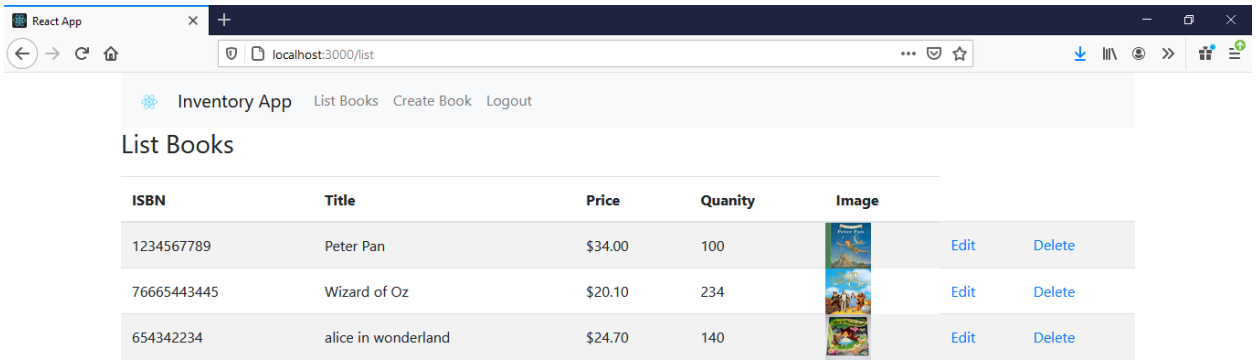
Login

A Login page may look like this:



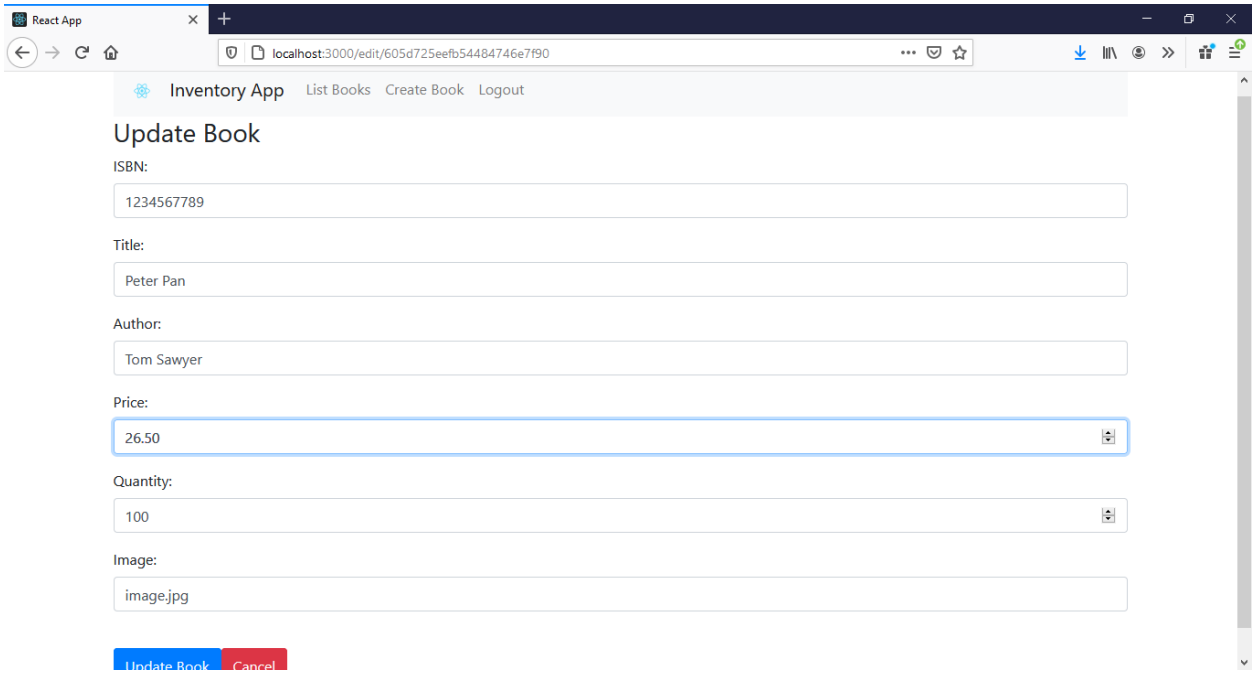
List Books

A List Books page may look like this:



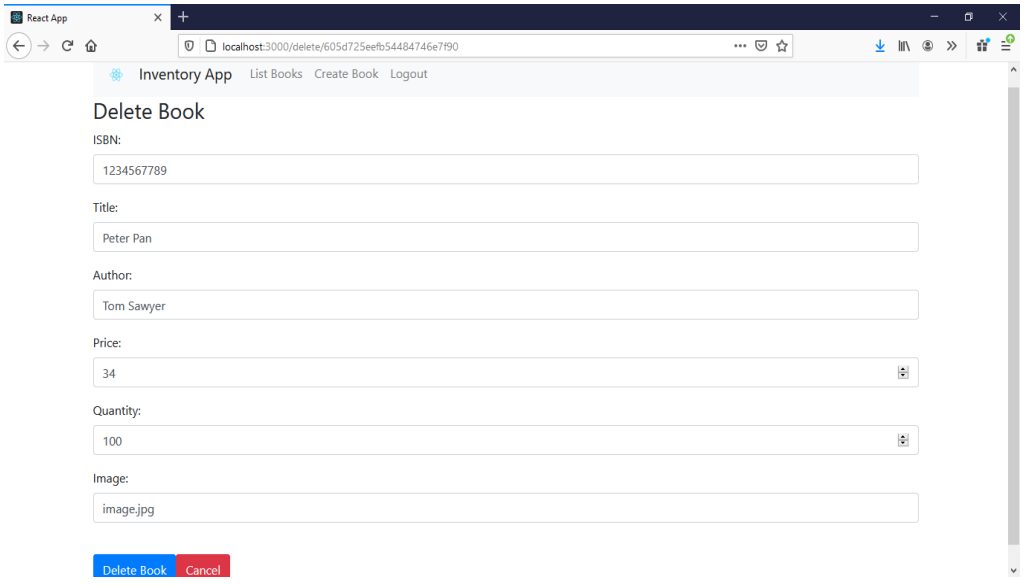
Edit Books

A Edit page may look like this:

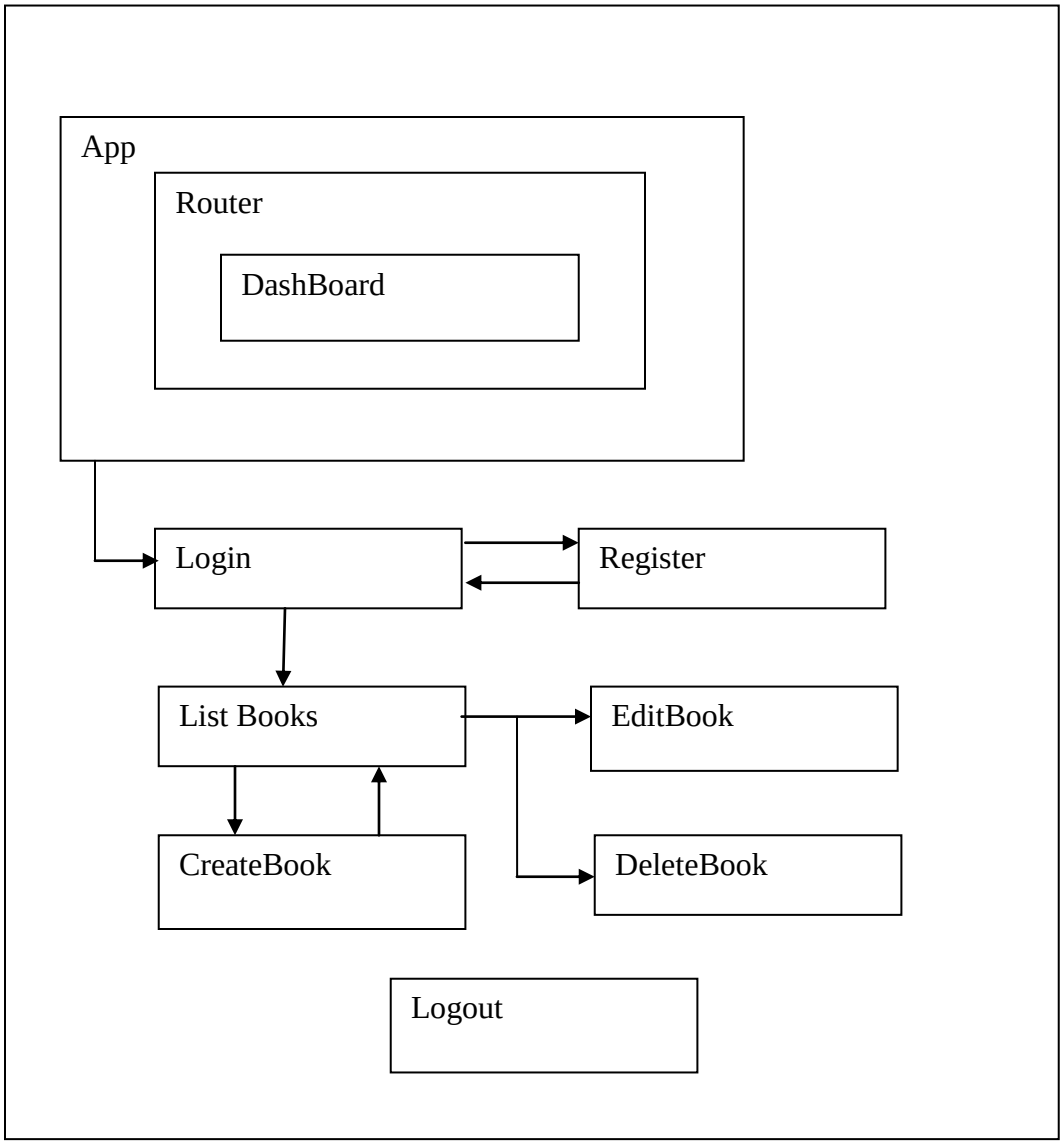


Delete Book

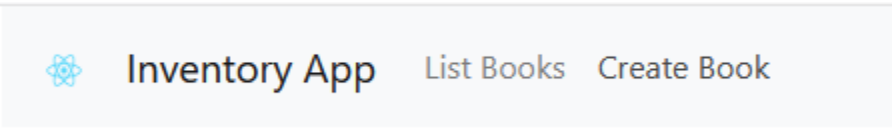
A Delete Book page may look like this:



Our InventoryApp Components block diagram:



The Dashboard Component displays the links. The Route Component picks up the link request and directs to the corresponding component for display. The Login page can only be directed to the Register page. The Register page can only be directed back to the Login page. Once the user logs in, they are directed to the ListBook page. From the ListBook page the user can select EditBook or Delete Book. Once the User is logged in they can select ListBooks or CreateBooks pages from the Dashboard.



Routing

Our first step is to make the preliminary individual components and then link them up with navigation buttons. We will use routing to navigate between all components

The next step is to replace all the App.js code with the following code. The App. Component will provide all the routing between all the other components. It will use a **Router** Component , a nested **Dashboard** Component and a **Switch** Component.

The **Router** handles the routing. The **Switch** component is used to select the component to render. We will select between the Login, Register, CreateBook, ListBooks, Edit Books and DeleteBook, components. The **EditBook and Delete Book** component can only be through the ListBook Component. The CreateBook and ListBook Component can only be accessed after Login. The RegisterComponent can only be accessed through Login Component. The **DashBoard** component produces the link buttons.

Here is the App.js file that is use for navigation. Replace App.js in your src folder with this code:

App.js

```
// App.js
// inventory app
// create, list, update and delete books

import React, { Component } from 'react';
import { BrowserRouter as Router, Route, Switch, Redirect } from "react-router-dom";
import "bootstrap/dist/css/bootstrap.min.css";

import CreateBook from "./components/createbook";
import EditBook from "./components/editbook";
import ListBooks from "./components/listbooks";
import DeleteBook from "./components/deletebook";
import Login from "./components/login";
```

```
import Register from "./components/register";
import Logout from "./components/logout";
import Dashboard from "./components/dashboard";

class App extends Component {

    // show navigation
    // handle routing
    render() {

        return (
            <Router>
                <div className="container">

                    <Dashboard />

                            <Switch>
                                <Route path="/" exact component={Login} />
                                <Route path="/register" exact component={Register} />
                                <Route path="/logout" exact component={Logout} />
                                <Route path="/list" exact component={ListBooks} />
                                <Route path="/edit/:id" component={EditBook} />
                                <Route path="/delete/:id" component={DeleteBook} />
                                <Route path="/create" component={CreateBook} />
                                <Redirect to="/" />
                            </Switch>

                </div>
            </Router>
        );
    }
}

export default App;
```

Components

Next we need to make a folder called **components** to house all our components.

Todo: make a folder called components in the client fielder.

dashboard

The first component we need to make is the Dashboard Component.

todo: put the following code in a JavaScript file called dashboard.js

```
// dashboard.js
// provides navigation dashboard
import React, { Component } from 'react';
import { Link } from "react-router-dom";
import "bootstrap/dist/css/bootstrap.min.css";
import logo from "../logo.svg";
export default class Dashboard extends Component {
    render() {
        return (
            <nav className="navbar navbar-expand-lg navbar-light bg-light">
                <a className="navbar-brand" href="https://onlineprogramminglessons.com" >
                    <img src={logo} width="30" height="30"
                        alt="onlineprogramminglessons.com" />
                </a>
                <Link to="/" className="navbar-brand">Inventory App</Link>
                <div className="collpase nav-collapse">
                    <ul className="navbar-nav mr-auto">
                        <li className="navbar-item">
                            <Link to="/list" className="nav-link">List Books</Link>
                        </li>
                        <li className="navbar-item">
                            <Link to="/create" className="nav-link">Create Book</Link>
                        </li>
                        <li className="navbar-item">
                            <Link to="/logout" className="nav-link">Logout</Link>
                        </li>
                    </ul>
                </div>
            </nav>
        )
    }
}
```

The rest of the components are skeleton components, they are not finished yet, and to be completed in future lessons.

SKELETON COMPONENTS

todo:

You need to install the **react-router-dom** module from node.js into your application Using NPM like this on the command line:

```
npm install react-router-dom

+ react-router-dom@5.2.0
added 11 packages from 6 contributors and audited 1966 packages in 15.483s
found 0 vulnerabilities
```

Most of our components use bootstrap to do form styling. **Bootstrap** is a framework to includes HTML and CSS based design templates for typography, forms, buttons, tables, navigation, modals, image carousels, etc. .

todo:

You need it install the bootstrap modules

```
npm install bootstrap
+ bootstrap@4.6.0
added 1 package from 2 contributors and audited 1955 packages in 15.364s
found 0 vulnerabilities
```

Here are all the skeleton components and functions you need to make.

Login Function login.js

Todo:

Make a JavaScript file called login.js and type in or copy/paste the following code. Put the login.js file in the components folder.

```
// login.js

import React, { useState } from 'react';
import { withRouter } from "react-router-dom";

function Login(props) {

    return (

    <div>
        <h1>Login</h1>
    </div>
    )
}

export default withRouter(Login);
```

Register Function register.js

Todo:
Make a JavaScript file called register.js and type in or copy/paste the following code. Put the register.js file in the components folder.

```
// register.js
// register new user

import React, { useState } from 'react';
import { withRouter } from "react-router-dom";

function Register(props) {


    // display login form
    return (

        <div>
            <h1>Register</h1>
        </div>
    )

}

export default withRouter(Register);
```

CreateBook Component createbook.js

Todo:
Make a JavaScript file called cratebook.js and type in or copy/paste the following code. Put the createbook.js file in the components folder.

```
// createbook.js
import React, { Component } from 'react';

export default class CreateBook extends Component {

    render() {
        return (
            <div style={{ marginTop: 20 }}>
                <h3>Create Book</h3>
            </div>
        )
    }
}
```

List Books Component listbooks.js

Todo:
Make a JavaScript file called listBooks.js and type in or copy/paste the following code. Put the listbooks.js file in the components folder.

```
// listbook.js
// list all books on books inventory from database server

import React, { Component } from 'react';

export default class ListBooks extends Component {

    // show books
    render() {

        return (
            <div>

                <h3>List Books</h3>
            </div>
        )
    }
}
```

Edit Book Component editbook.js

Todo:
Make a JavaScript file called editbook.js and type in or copy/paste the following code. Put the editbook.js file in the components folder.

```
// editbook.js
// update an existing book in data base

import React, { Component } from 'react';

export default class EditBook extends Component {
    // show book to edit
    render() {

        return (
            <div>
                <h3>Update Book</h3>
            </div>
        )
    }
}
```

Delete Book Component deletebook.js

Todo:
Make a JavaScript file called deletebook.js and type in or copy/paste the following code. Put the deletebook.js file in the components folder.

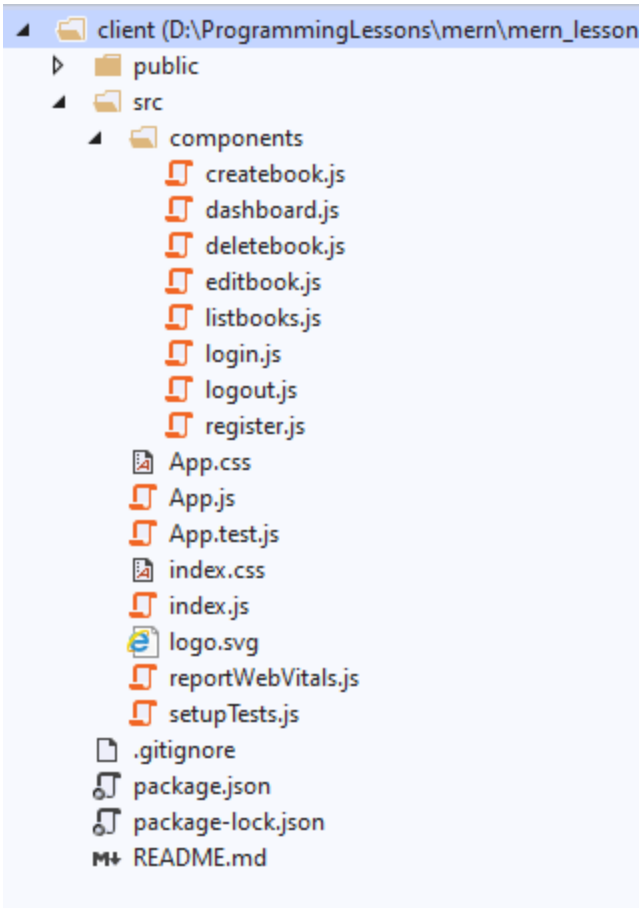
```
// deletebook.js
// delete book from data base

import React, { Component } from 'react';

export default class DeleteBook extends Component {

    // show book to delete
    render() {

        return (
            <div>
                <h3>Delete Book</h3>
            </div>
        )
    }
}
```

Login Function logout.js

Todo:
Make a JavaScript file called logout.js and type in or copy/paste the following code. Put the logout.js file in the components folder.

```
// logout.js
// handle logout
// redirect back to login page

function Logout(props) {

    return "You are logged out";
}

export default Logout;
```

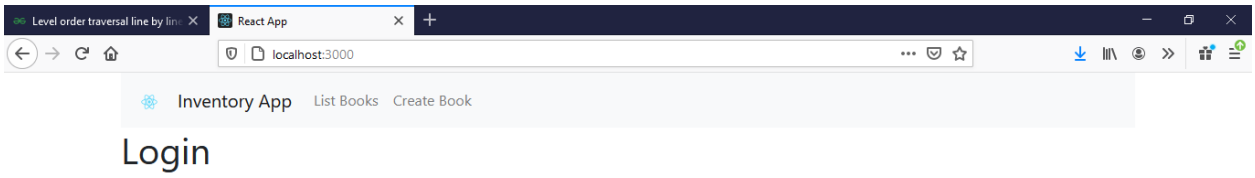
Your Client folder should now look like this:



Todo:
Run the App

npm start

You should get something like this



Note:
To return back to the command type ctrl C [^C]
Terminate batch job (Y/N)? Y
And you will have the command line back again.

LESSON 9 LOGIN and REGISTRATION PAGE

The Login Page allows the user to login into the inventory app. If there are no users, then a user needs to register first.

Our Login Function uses states and uses a hook that lets you add **states** to a function. This is a new feature of react that we wanted to try out.

The useState hook lets you add a React **state** to a function component.
.

```
import React, { useState } from 'react';

function Login(props) {
const [state, setState] = useState({
        username: "",
         password: "",
        successMessage: null
  })
```

handling redirects

We also import **withRouter** from the react-router-dom.

```
import { withRouter } from "react-router-dom";
```

By doing this, the Login component has access to the **this.props.history** so it can redirect the user with the **this.props.history.push** to other routes.

The **withRouter** gives the Login component access to **props.history**, which means the Login component can now redirect the user using the **props.history**. **Props.history** keeps track of previous **urls** like a back button in a web browser.

We must also export the Login Component **withRouter** at the end of our login.js file as follows:

```
export default withRouter(Login);
```

Communicating with the node.js server

The Login Function uses **axios** to communicate with the server. **Axios** is a promise based HTTP client for node.js. Using **axios** we can handle all GET, POST, PUT and DELETE requests and responses quite easily. Since it is promised based we can use **then** and **catch**.

Todo: install axios

```
npm install axios
+ axios@0.21.1
added 1 package from 1 contributor and audited 1967 packages in 14.921s
found 0 vulnerabilities
```

You need to import axios before you can use it:

```
import axios from 'axios';
```

We first set our base URL to the server to port 4000 as a constant.

```
const API_BASE_URL = 'http://localhost:4000';
```

Then we just call the appropriate function like **Post** with our data object. The data object is the body to the **Post** message. The **api** is our base path for our server.

```
axios.post(API_BASE_URL + '/api/login', data)
```

When we receive data from the server we need to update the **state**.

prevState is provided by React along with **props**,

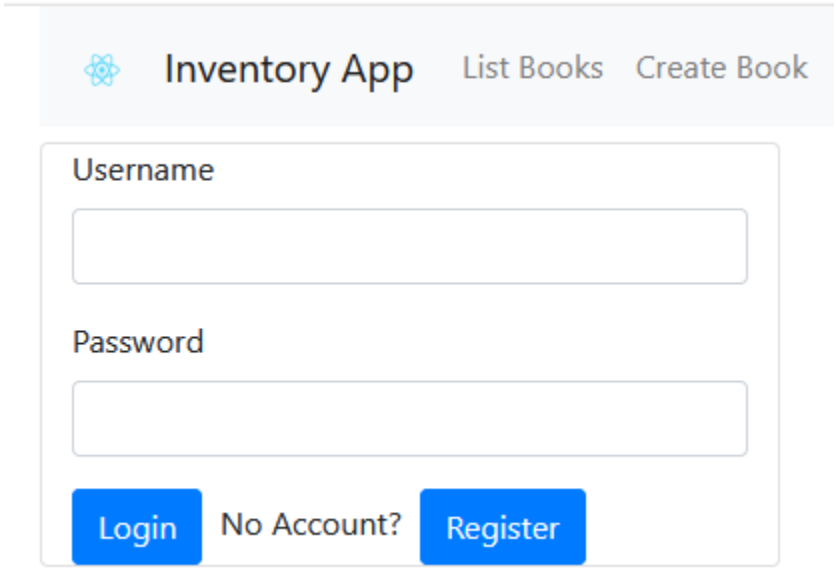
The prevState stores the previous value of state so we can update it with a previous value. We only need to do this when we just want to change a few items in the state not all. Here is an example where Login is successful and we just want to update the successMessage. The three dots in …prevState means all array values in the prevState. We also set **localStorage** to a received authentication token returned from the server.

localStorage is a memory area in the web browser that can store string variables accessed by a key name. We are using localStorage as a session object.

Here is part of the login code that uses **axios** to login. The payload is the username and password obtained from the login form.

```
 axios.post(API_BASE_URL + '/api/login', payload)
.then(function (response) {
   //alert(JSON.stringify(response));
   if (response.status === 200) {
      setState(prevState => ({
         ...prevState,
         'successMessage': 'Login successful. Redirecting to list books'
      }))

      localStorage.setItem("token", response.data.token);
      redirectToListBooks();

   }
```

Storing form data



Every time somebody types there username or password in the textbox the **handleChange** event function is called that receives an **event** object from the login form text box. The **target** object return from the event object contains an id and value. The **id** is the textbox id and the **value** is what was typed into the textbox. Again we use prevState to update the state with the previous value because nothing has been changed and we update the username or password using [id]:value depending what the textbox id is.

```
const handleChange = (e) => {
    const { id, value } = e.target
    setState(prevState => ({
      ...prevState,
      [id]: value
    }))
    }
```

The **handleChange** event handler is initiated by the onChange={handleChange} event attribute in the render method per textbox

The **handleSubmit** function obtains the username and password from the state as a payload to send to the server using axios as previously described.

```
const handleSubmitClick = (e) => {
    e.preventDefault();
    const payload = {
      "username": state.username,
      "password": state.password,
    }
```

The **handleSubmitClick** event handler is initiated by the onChange={handleChange} event attribute in the render method for the submit button.

Here is the complete Login.js code:

```
// login.js
// authenticate user
// receive auth token from server
// go to list books
```

```
import React, { useState } from 'react';
import axios from 'axios';
import { withRouter } from "react-router-dom";

const API_BASE_URL = 'http://localhost:4000';

function Login(props) {

    // store username, password and success message
    const [state, setState] = useState({
        username: "",
        password: "",
        successMessage: null
    })


    // handle textbox change
    const handleChange = (e) => {
        const { id, value } = e.target
        setState(prevState => ({
            ...prevState,
            [id]: value
        }))
    }

    // handle submit button
    const handleSubmitClick = (e) => {
        e.preventDefault();
        const payload = {
            "username": state.username,
            "password": state.password,
        }


       // alert(JSON.stringify(payload.username))

        // login: send username and password to server
        axios.post(API_BASE_URL + '/api/login', payload)

            // success
            .then(function (response) {
                //alert(JSON.stringify(response));
                if (response.status === 200) {
                    setState(prevState => ({
                        ...prevState,
                        'successMessage': 'Login successful. Redirecting to list books'
                    }))

                    localStorage.setItem("token", response.data.token);
                    redirectToListBooks();

                }
```

```
            // incorrect login
            else {
                setState(prevState => ({
                    ...prevState,
                    'successMessage': "Incorrect Login"
                }))

            }
        })
        .catch(function (error) {
           // alert(JSON.stringify(error));
            console.log(error);
            setState(prevState => ({
                ...prevState,
                'successMessage': "Login Error, Try Again"
            }))

        });
}


// redirect to listbooks
const redirectToListBooks = () => {

    window.location.href = '/list';
}

// display login form
return (

        <div className="card col-12 col-lg-4 login-card mt-2 hv-center">
        <form>
            <div className="form-group text-left">
                <label for="username">Username</label>
                <input type="username"
                    className="form-control"
                    id="username"
                    value={state.username}
                    onChange={handleChange}
                />
                <small id="emailHelp" className="form-text text-muted"></small>
            </div>
            <div className="form-group text-left">
                <label for="password">Password</label>
                <input type="password"
                    className="form-control"
                    id="password"

                    value={state.password}
                    onChange={handleChange}
                />
            </div>
            <div className="form-check">
            </div>
```

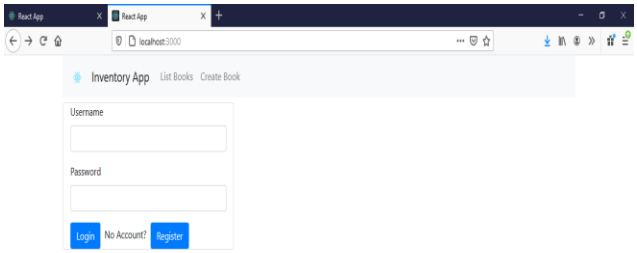
```
                <button
                    type="submit"
                    className="btn btn-primary"
                    onClick={handleSubmitClick}
                >Login</button>

                  No Account?  
                <input
                    type="button" value="Register" className="btn btn-primary"
                    onClick={(e) => {
                        e.preventDefault();
                        window.location.href = '/register';
                    }}
                />
            </form>
            <div className="alert alert-success mt-2"
   style={{ display:  state.successMessage ? 'block' : 'none' }} role="alert">
                {state.successMessage}
            </div>

        </div>
    )
}

export default withRouter(Login);
```

todo: update your login.js file with the above code.

Next run the client

npm start

You should get something like this:



REGISTRATION PAGE

We need to register a user before we can log in. The Registration page is very similar to the Login page operation.

We again **useState** hook that allows us to use states with a function.

```
import React, { useState } from 'react';
```

We again use **axios** so that we can communicate easily with the node.js server

```
import axios from 'axios';
```

We again use **withRouter** so we can use **props.history.push()** to redirect to other pages

```
import { withRouter } from "react-router-dom";
```

we set our base url to the node.js server to 'http://localhost:4000'

```
const API_BASE_URL = 'http://localhost:4000';
```

Our state will contain username, email, password, confirmPassword and a success message.

```
function Register(props) {
  const [state, setState] = useState({
    username: "",
    email: "",
    password: "",
    confirmPassword: "",
    successMessage: null
  })
```

A **handlechange** function initiate by the **onchange={handlechange}** attribute will handle storing of the registration state information where the **id** is the textbox id and the **value** is what was typed into the text box,

```
  const handleChange = (e) => {
    const { id, value } = e.target
    setState(prevState => ({
      ...prevState,
      [id]: value
    }))
  }
```

The registration information is sent to the server as a payload to be stored in the database. If every successful we are redirected back to the login page.

```
// validate inputs and send user data to database server
  const sendDetailsToServer = () => {
    if (state.password.length && state.email.length && state.password.length) {

      const payload = {
        "username": state.username,
        "email": state.email,
        "password": state.password,
      }

      // send user data to data base server
      axios.post(API_BASE_URL + '/api/register', payload)
        .then(function (response) {

          // alert(JSON.stringify(response));

          if (response.status === 201) {
            setState(prevState => ({
              ...prevState,
              'successMessage': 'Registration successful. Redirecting to Login'
            }))

            redirectToLogin();

  }
```

RedirectToLogin

We push the '/' login path into our history stored in props. This is possible because we are using **withRouter**.

```
  // redirect to login
  const redirectToLogin = () => {

    props.history.push('/');
  }
```

When the user is redirected back to the login page then the user can login

Here is the complete registration.js file

```
// register.js
// register new user

import React, { useState } from 'react';
import axios from 'axios';
import { withRouter } from "react-router-dom";
const API_BASE_URL = 'http://localhost:4000';

function Register(props) {
  const [state, setState] = useState({
    username: "",
    email: "",
    password: "",
    confirmPassword: "",
    successMessage: null
  })
  const handleChange = (e) => {
    const { id, value } = e.target
    setState(prevState => ({
      ...prevState,
      [id]: value
    }))
  }
```

```
// validate inputs and send user data to database server
const sendDetailsToServer = () => {
   if (state.password.length && state.email.length && state.password.length) {

      const payload = {
         "username": state.username,
         "email": state.email,
         "password": state.password,
      }

      // send user data to data base server
      axios.post(API_BASE_URL + '/api/register', payload)
         .then(function (response) {

            // alert(JSON.stringify(response));

            if (response.status === 201) {
               setState(prevState => ({
                  ...prevState,
                  'successMessage': 'Registration successful. Redirecting to Login'
               }))
               redirectToLogin();
            }

           else {
               setState(prevState => ({
                  ...prevState,
                  'successMessage': 'Registration not successful. Try again'
               }))
            }
         })
         .catch(function (error) {
            console.log(error);
         });
   } else {
      setState(prevState => ({
         ...prevState,
         'successMessage': 'Please enter valid username, email and password'
      }))
   }

}
```

```
// redirect to login
const redirectToLogin = () => {

  props.history.push('/');
}

// submit button
const handleSubmitClick = (e) => {
  e.preventDefault();
  if (state.password === state.confirmPassword) {
    sendDetailsToServer()
  } else {
    setState(prevState => ({
      ...prevState,
      'successMessage': 'Passwords do not match'
    }))
  }
}

// display registration form
return (
  <div className="card col-12 col-lg-4 login-card mt-2 hv-center">
    <form>
      <div className="form-group text-left">
        <label For="username">Username</label>
        <input type="text"
          className="form-control"
          id="username"
          value={state.username}
          onChange={handleChange}
        />
      </div>
      <div className="form-group text-left">
        <label For="email">Email address</label>
        <input type="email"
          className="form-control"
          id="email"
          value={state.email}
          onChange={handleChange}
        />
       </div>
```

```
            <div className="form-group text-left">
              <label For="password">Password</label>
              <input type="password"
                 className="form-control"
                 id="password"
                 value={state.password}
                 onChange={handleChange}
              />
            </div>
            <div className="form-group text-left">
              <label for="confirmPassword">Confirm Password</label>
              <input type="password"
                 className="form-control"
                 id="confirmPassword"
                 value={state.confirmPassword}
                 onChange={handleChange}
              />
            </div>
            <button
              type="submit"
              className="btn btn-primary"
              onClick={handleSubmitClick}>
              Register
            </button>

              Login if have Account  

            <input
              type="button" value="Login" className="btn btn-primary"
              onClick={(e) => {
                 e.preventDefault();
                 window.location.href = '/';
              }}
            />
          </form>
          <div className="alert alert-success mt-2"
                    style={{ display: state.successMessage ? 'block' : 'none' }} role="alert">
            {state.successMessage}
          </div>
        </div>
    )
}
```

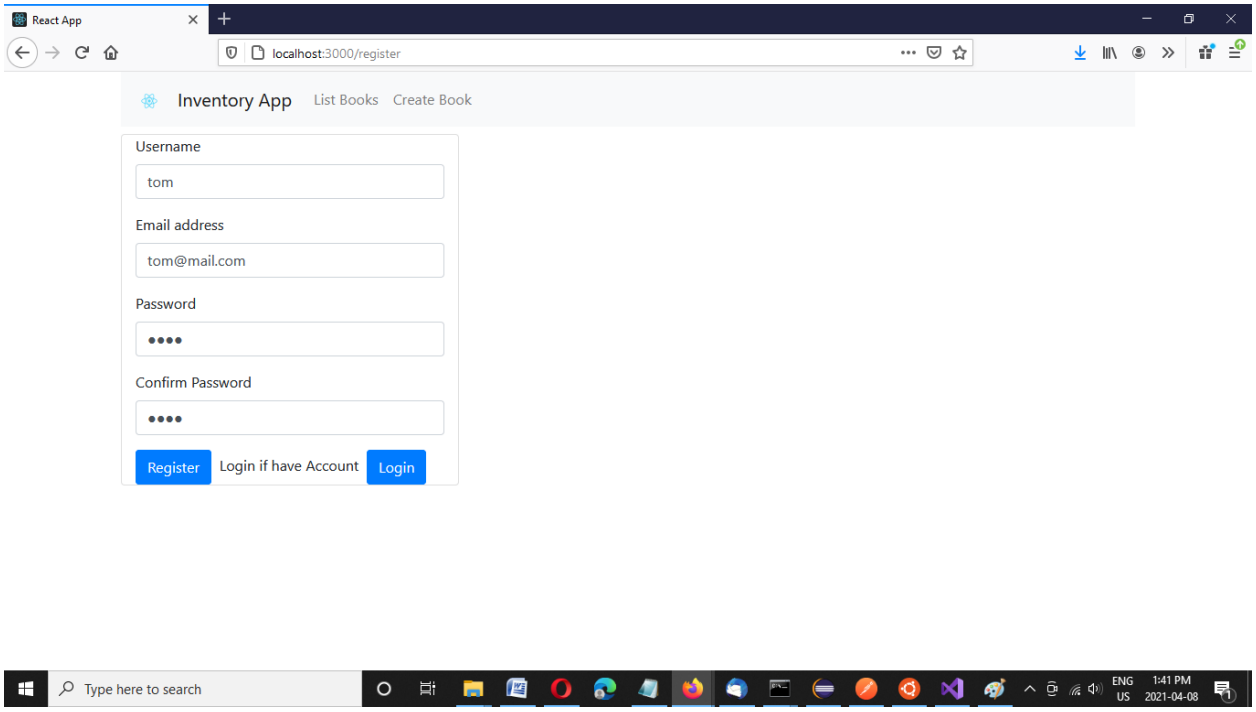
export default withRouter(Register);

Todo:

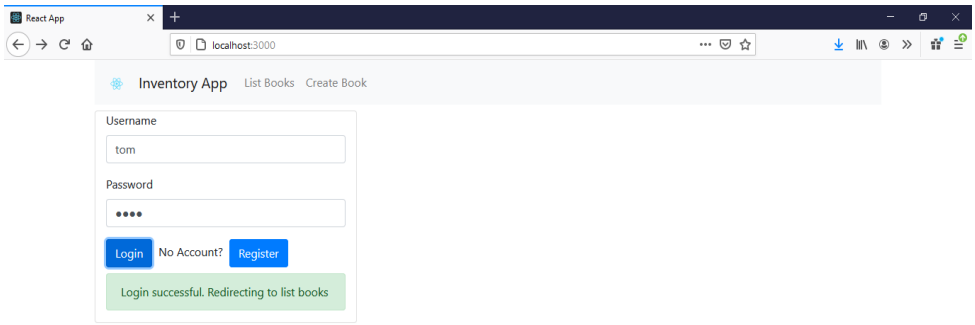
Type in or copy/paste the above registration code and put into file register.js
Run your program

npm start

You should get something like this



Now try to login:



LESSON9 HOMEWORK

Add validation to the registration page so they must enter a username email and password. Print out error messages beside each text box.

LESSON 10
PREVENTING UNAUTHORIZED USERS AND LOGGING OUT

LOGOUT MENU

When the user logs in the dashboard should have a menu item that says they can log out.

In our dashboard component we can display the logout menu when the local storage login token is set.

We use the **localStorage.getItem("token")** value and the && operator to display the logout menu item link.

```
{
  localStorage.getItem("token") != null &&
  <li className="navbar-item">
     <Link to="/logout" className="nav-link">Logout</Link>
  </li>
}
```

Todo: Add the above logic to the dashboard.js component.

The final dsashboard.js file should look like this:

```
// dashboard.js
// provides navigation dashboard

import React, { Component } from 'react';
import { Link } from "react-router-dom";
import "bootstrap/dist/css/bootstrap.min.css";
import logo from "../logo.svg";
```

```
export default class Dashboard extends Component {

  render() {

    return (
      <nav className="navbar navbar-expand-lg navbar-light bg-light">
        <a className="navbar-brand" href="https://onlineprogramminglessons.com" >
          <img src={logo} width="30" height="30" alt="onlineprogramminglessons.com" />
        </a>
        <Link to="/" className="navbar-brand">Inventory App</Link>
        <div className="collpase nav-collapse">
          <ul className="navbar-nav mr-auto">
            <li className="navbar-item">
              <Link to="/list" className="nav-link">List Books</Link>
            </li>
            <li className="navbar-item">
              <Link to="/create" className="nav-link">Create Book</Link>
            </li>
            {
              localStorage.getItem("token") != null &&
              <li className="navbar-item">
                <Link to="/logout" className="nav-link">Logout</Link>
              </li>
            }
          </ul>
        </div>
      </nav>
    )
  }
}
```

PREVENTING UNAUTHORIZED USERS

The next thing we have to do is prevent unauthorized users. We can do this conveniently in our router located in App Component.

We split our routes into two sections. Before login and after login. Before login we have access to login, register. After login we have access to list books, create books and logout.

```
{
localStorage.getItem("token") != null ? (
        <Switch>
          <Route path="/" exact component={Login} />
          <Route path="/register" exact component={Register} />
          <Route path="/logout" exact component={Logout} />
          <Route path="/list" exact component={ListBooks} />
          <Route path="/edit/:id" component={EditBook} />
          <Route path="/delete/:id" component={DeleteBook} />
          <Route path="/create" component={CreateBook} />
          <Redirect to="/" />
        </Switch>
    )
        :
        (
          <Switch>
            <Route path="/" exact component={Login} />
            <Route path="/register" exact component={Register} />
            <Route path="/logout" exact component={Logout} />
            <Redirect to="/" />
          </Switch>
        )
  }
```

We are using a react **conditional if** statement.

operator `condition ? true : false`.

To do:

Add the conditional if statement to the App component render method. Your final App.js file should look like this: (make sure you have the { and })

```
// App.js
// inventory app
// create, list, update and delete books

import React, { Component } from 'react';
import { BrowserRouter as Router, Route,Switch,Redirect} from "react-router-dom";
import "bootstrap/dist/css/bootstrap.min.css";
```

```
import CreateBook from "./components/createbook";
import EditBook from "./components/editbook";
import ListBooks from "./components/listbooks";
import DeleteBook from "./components/deletebook";
import Login from "./components/login";
import Register from "./components/register";
import Logout from "./components/logout";
import Dashboard from "./components/dashboard";

class App extends Component {

  // show navigation
  // handle routing
  render() {

    return (
      <Router>
        <div className="container">

          <Dashboard />
          {
            localStorage.getItem("token") != null ? (
              <Switch>
                <Route path="/" exact component={Login} />
                <Route path="/register" exact component={Register} />
                <Route path="/logout" exact component={Logout} />
                <Route path="/list" exact component={ListBooks} />
                <Route path="/edit/:id" component={EditBook} />
                <Route path="/delete/:id" component={DeleteBook} />
                <Route path="/create" component={CreateBook} />
                <Redirect to="/" />
              </Switch>
            )
              :
                (
                <Switch>
                  <Route path="/" exact component={Login} />
                  <Route path="/register" exact component={Register} />
                  <Route path="/logout" exact component={Logout} />
                  <Redirect to="/" />
                </Switch>
              )
```

```
            }
          </div>
        </Router>
      );
    }
}

export default App;
```

LOGGING OUT

Logout Component logout.js

Logging out is quite easy to do.

The logout.js file is complete and simply logs the user out and then redirects the user back to the login page.

If first clears the authorization token store in local storage.

```
        // clear auth token
        localStorage.removeItem("token");
```

Local storage acts like a session, so that when a user logs in we know that someone has logged in.

Then we refresh the app

```
            // reload navigation
            window.location.reload();
```

and finally redirect back the login page

```
    // redirect to login
     window.location.href = '/';
```

Todo:

Update your logout.js file and type in or copy/paste the following code.

```
// logout.js
// handle logout
// remove token
// redirect back to login page

function Logout(props) {

    // clear auth token
    localStorage.removeItem("token");

    // reload navigation
    window.location.reload();

    // redirect to login
    window.location.href = '/';

    return "You are logged out";
}

export default Logout;
```

LESSON 11 Create Books

The Create Book Component is used to create books to be stored on the node.js server.

A state is used to store the book details

```
this.state = {
        isbn: '',
        title: '',
        author: '',
        price: 0,
        quantity: 0,
        image: ''
      }
```

A form of text boxes is used to gather information for book details. Each text box is used to send data to the state variables.

We have made six onChange event handler's to get the data from each text box. Here is one of them:

```
onChangeIsbn(e) {
      this.setState({
        isbn: e.target.value
      });
   }
```

Each text box has a **onChange** event attribute that calls a corresponding event handler.

```
   <input type="text"
                      className="form-control"
                      value={this.state.isbn}
                      onChange={this.onChangeIsbn}
                    />
```

We need to bind our **this** pointer to the onChangeIsbn event handler. The this pointer refers to our CreateBook instance. An **instance** is the memory allocated for the component that we are using. The memory allocated is the state variables. Bind actually means the reference to our CreateBook instance the **this** pointer is given to **onChangeIsbn** event handler function so the data being transferred refers to the same instance. Once the **onChangeIsbn** has a reference to our **this** pointer it can store data to the state variable, else the data would not get stored in the state variable, it would be **undefined**.

```
this.onChangeIsbn = this.onChangeIsbn.bind(this);
```

Submit button

When the submit button is pressed the onSubmit event handler is called. The form tag has the onSubmit event attribute equated to the onSubmit event handler.

```
<form onSubmit={this.onSubmit}>
```

The onSubmit event handler create a new book object the calls the **axios** post function that sends the book data object to the node.js server. If everything is successful the state variables are cleared, Here is the onSubmit event handler:

```
// send book data to  database server
 onSubmit(e) {
   e.preventDefault();

   // make new book
   const newBook = {
     isbn: this.state.isbn,
     title: this.state.title,
     author: this.state.author,
     price: this.state.price,
     quantity: this.state.quantity,
     image: this.state.image
   }

   // send book to server
   axios.post(API_BASE_URL + '/api/book', newBook)
     .then(res => console.log(res.data));
```

```
      // clear state
      this.setState({
        isbn: '',
        title: '',
        author: '',
        price: 0,
        quantity: 0,
        image: ''
      })
    }
```

The render method displays all the book data in text boxes.
Here is the updated CreateBook component.

```
// createbook.js
// create info for  new book and send details to database server

import React, { Component } from 'react';
import axios from 'axios';
const API_BASE_URL = 'http://localhost:4000';

export default class CreateBook extends Component {

  constructor(props) {
    super(props);
    this.onChangeIsbn = this.onChangeIsbn.bind(this);
    this.onChangeTitle = this.onChangeTitle.bind(this);
    this.onChangeAuthor = this.onChangeAuthor.bind(this);
    this.onChangePrice = this.onChangePrice.bind(this);
    this.onChangeQuantity = this.onChangeQuantity.bind(this);
    this.onChangeImage = this.onChangeImage.bind(this);
    this.onSubmit = this.onSubmit.bind(this);

    this.state = {
      isbn: '',
      title: '',
      author: '',
      price: 0,
      quantity: 0,
      image: ''
    }
  }
```

```
onChangeIsbn(e) {
   this.setState({
      isbn: e.target.value
   });
}

onChangeTitle(e) {
   this.setState({
      title: e.target.value
   });
}

onChangeAuthor(e) {
   this.setState({
      author: e.target.value
   });
}

onChangePrice(e) {
   this.setState({
      price: e.target.value
   });
}

onChangeQuantity(e) {
   this.setState({
      quantity: e.target.value
   });
}

onChangeImage(e) {
   this.setState({
      image: e.target.value
   });
}

// send book data to  database server
onSubmit(e) {
   e.preventDefault();
```

```
  // make new book
   const newBook = {
      isbn: this.state.isbn,
      title: this.state.title,
      author: this.state.author,
      price: this.state.price,
      quantity: this.state.quantity,
      image: this.state.image
   }

   // send book to server
   axios.post(API_BASE_URL + '/api/book', newBook)
      .then(res => console.log(res.data));

   // clear state
   this.setState({
      isbn: '',
      title: '',
      author: '',
      price: 0,
      quantity: 0,
      image: ''
   })
}

// show form
render() {

   return (
      <div style={{ marginTop: 20 }}>
         <h3>Create Book</h3>

         <form onSubmit={this.onSubmit}>
            <div className="form-group">
               <label>ISBN: </label>
               <input type="text"
                  className="form-control"
                  value={this.state.isbn}
                  onChange={this.onChangeIsbn}
               />
            </div>
            <div className="form-group">
```

```
    <label>Title: </label>
    <input type="text"
      className="form-control"
      value={this.state.title}
      onChange={this.onChangeTitle}
    />
  </div>
  <div className="form-group">
    <label>Author: </label>
    <input type="text"
      className="form-control"
      value={this.state.author}
      onChange={this.onChangeAuthor}
    />
  </div>
  <div className="form-group">
    <label>Price: </label>
    <input type="number"
      min="0" max="1000000" step=".01"
      className="form-control"
      value={this.state.price}
      onChange={this.onChangePrice}
    />
  </div>
  <div className="form-group">
    <label>Quantity: </label>
    <input type="number"
      min="0" max="1000000" step="1"
      className="form-control"
      value={this.state.quantity}
      onChange={this.onChangeQuantity}
    />
  </div>
  <div className="form-group">
    <label>Image: </label>
    <input type="text"
      className="form-control"
      value={this.state.image}
      onChange={this.onChangeImage}
    />
  </div>
```

```
            <div className="form-group">
              <input type="submit" value="Create Book" className="btn btn-primary" />
              <input
                type="button" value="List" className="btn btn-danger"
                onClick={(e) => {
                  e.preventDefault();
                  window.location.href = '/list';
                }}
              />
              <input
                type="button" value="Cancel" className="btn btn-danger"
                onClick={(e) => {
                  e.preventDefault();
                  window.location.href = '/list';
                }}
              />
            </div>
          </form>

        </div>
      )
    }
}
```

Todo:

Update your CreateBook App with the above code either type in or copy/paste.

Lesson11 Homework

Add validation and include error messages.

LESSON 12 List, Edit and Delete Books

The List Component stores a list of books in its state.

```
constructor(props) {
      super(props);
      this.state = { books: [] };
   }
```

When a component is first ran the **componentDidMount** is called. That obtain a list of books from the node.js server using axios with a GET request.

```
// get initial books
   componentDidMount() {

      //alert("mount")

      // get books from data base server
      axios.get(API_BASE_URL + '/api/books/')
            .then(response => {
               this.setState({ books: response.data.data });
            })
            .catch(function (error) {
               console.log(error);
            })
      }
```

We also have the **componentDidUpdate()** function that is called when a state changes. This is necessary because we have to get a new list of books from the nofde.js server when a edit or delete is performed. The **componentDidUpdate()** function is little dangerous to use because it will be called again many times when the state is changed. The work around is to don't update the state variable if the book data is the same as the saved book list from the node.js server.

```
// get book list again after edit
   componentDidUpdate() {

         //alert("update")
         axios.get(API_BASE_URL + '/api/books/')
            .then(response => {
```

```
        // update only fresh data
        if (JSON.stringify(response.data.data) !== JSON.stringify(this.state.books))
        {
        this.setState({ books: response.data.data });
        }
        })
        .catch(function (error) {
           console.log(error);
        })
    }
```

Render book list

We use a table to list the books. We use a map to display information for each book.

```
        this.state.books.map((item, i) => {
                    return (
                      <tr>
                        <td>{item.isbn}</td>
                        <td>{item.title}</td>
                        <td>${item.price.toFixed(2)}</td>

                        <td>{item.quantity}</td>
                        <img src={item.image} height='50' width='50' alt={item.image} />

                        <td>
                          <Link to={"/edit/" + item._id}>Edit</Link>
                        </td>
                        <td>
                          <Link to={"/delete/" + item._id}>Delete</Link>
                        </td>
                      </tr>
                    );
                  })
```

We have to links shown above to route to the edit page and one to the delete page that also contains the item.sd to identify the book to edit and delete

Here is the complete BookList component

```
// listbook.js
// list all books on books inventory from database server

import React, { Component } from 'react';
import { Link } from 'react-router-dom';
import axios from 'axios';
const API_BASE_URL = 'http://localhost:4000';

export default class ListBooks extends Component {

  constructor(props) {
    super(props);
    this.state = { books: [] };
  }

  // get initial books
  componentDidMount() {

    //alert("mount")

    // get books from data base server
    axios.get(API_BASE_URL + '/api/books/')
        .then(response => {
          this.setState({ books: response.data.data });
        })
        .catch(function (error) {
          console.log(error);
        })
    }

  // get book list again after edit
  componentDidUpdate() {

      //alert("update")
      axios.get(API_BASE_URL + '/api/books/')
        .then(response => {

        // update only fresh data
        if (JSON.stringify(response.data.data) !== JSON.stringify(this.state.books))
        {
        this.setState({ books: response.data.data });
        }
```

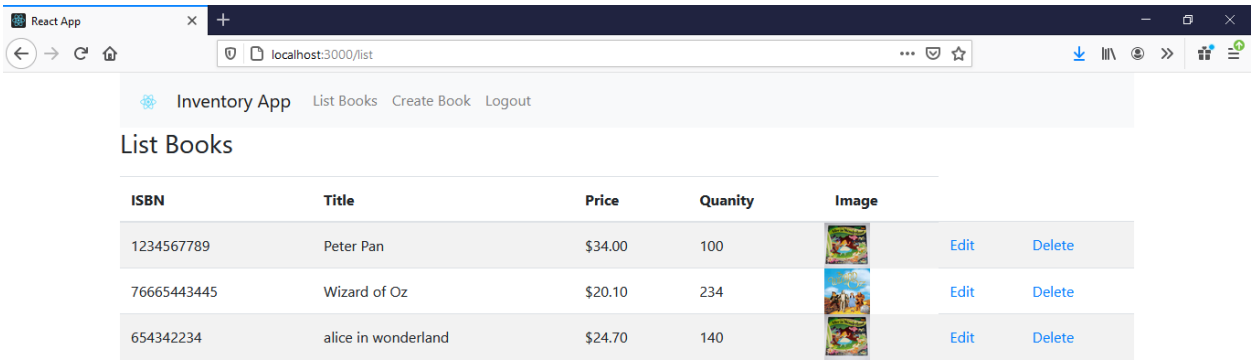
```
      })
      .catch(function (error) {
        console.log(error);
      })
}

// show books
render() {

  return (
    <div>

      <h3>List Books</h3>

      <table className="table table-striped" style={{ marginTop: 20 }}>

        <thead>
          <th>ISBN</th>
          <th>Title</th>
          <th>Price</th>
          <th>Quantity</th>
          <th>Image</th>
        </thead>

        <tbody>
          {
            this.state.books.map((item, i) => {
              return (
                <tr>
                  <td>{item.isbn}</td>
                  <td>{item.title}</td>
                  <td>${item.price.toFixed(2)}</td>

                  <td>{item.quantity}</td>
                  <img src={item.image} height='50' width='50' alt={item.image} />

                  <td>
                    <Link to={"/edit/" + item._id}>Edit</Link>
                  </td>
                  <td>
                    <Link to={"/delete/" + item._id}>Delete</Link>
                  </td>
```

```
                    </tr>
                );
            })
        }
    </tbody>

</table>

</div>
)

}
}
```

To do:

Update your BookList App with the above code either type in or copy/paste.

You should get something like this:



EDITING BOOKS

The Edit component will receive the book id to delete, stored in the **prods.match.param_id**

The EditComponent page and is very similar to the Create Book code page. The only difference is when we submit we are updating an existing book rather than creating a new book.

```
// send update book to data base server
  onSubmit(e) {
    e.preventDefault();
    const book = {
      isbn: this.state.isbn,
      title: this.state.title,
      author: this.state.author,
      price: this.state.price,
      quantity: this.state.quantity,
      image: this.state.image
    };

    axios.put(API_BASE_URL + '/api/book/' + this.props.match.params.id, book)
      .then(res => console.log(res.data));

    this.props.history.push('/list');
  }
```

The edit page originates from the listbook page and then returns back to the booklist page when the book is successfully deleted.

Here is the complete editBook component

```
// editbook.js
// update an existing book in data base

import React, { Component } from 'react';
import axios from 'axios';
const API_BASE_URL = 'http://localhost:4000';
```

```
export default class EditBook extends Component {

  constructor(props) {
    super(props);
    this.onChangeIsbn = this.onChangeIsbn.bind(this);
    this.onChangeTitle = this.onChangeTitle.bind(this);
    this.onChangeAuthor = this.onChangeAuthor.bind(this);
    this.onChangePrice = this.onChangePrice.bind(this);
    this.onChangeQuantity = this.onChangeQuantity.bind(this);
    this.onChangeImage = this.onChangeImage.bind(this);
    this.onSubmit = this.onSubmit.bind(this);

    this.state = {
      isbn: '',
      title: '',
      author: '',
      price: 0,
      quantity: 0,
      image: ''
    }
  }

  // get book from data base
  componentDidMount() {
    //alert(this.props.match.params.id);
    axios.get(API_BASE_URL + '/api/book/' + this.props.match.params.id)
      .then(response => {

        // alert(response);
        this.setState({
          isbn: response.data.data.isbn,
          title: response.data.data.title,
          author: response.data.data.author,
          price: response.data.data.price,
          quantity: response.data.data.quantity,
          image: response.data.data.image
        })
      })
      .catch(function (error) {
        console.log(error)
      })
  }
```

```
// store isbn number
onChangeIsbn(e) {
   this.setState({
      isbn: e.target.value
   });
}

// store title
onChangeTitle(e) {
   this.setState({
      title: e.target.value
   });
}

// store author
onChangeAuthor(e) {
   this.setState({
      author: e.target.value
   });
}

// store price
onChangePrice(e) {
   this.setState({
      price: e.target.value
   });
}

// store quantity
onChangeQuantity(e) {
   this.setState({
      quantity: e.target.value
   });
}

// store image
onChangeImage(e) {
   this.setState({
      image: e.target.value
   });
}
```

```
// send update book to data base server
onSubmit(e) {
   e.preventDefault();
   const book = {
      isbn: this.state.isbn,
      title: this.state.title,
      author: this.state.author,
      price: this.state.price,
      quantity: this.state.quantity,
      image: this.state.image
   };
   axios.put(API_BASE_URL + '/api/book/' + this.props.match.params.id, book)
      .then(res => console.log(res.data));

   this.props.history.push('/list');
}

// show book to edit
render() {

   return (
      <div>
         <h3>Update Book</h3>

         <form onSubmit={this.onSubmit}>
            <div className="form-group">
               <label>ISBN: </label>
               <input type="text"
                  className="form-control"
                  value={this.state.isbn}
                  onChange={this.onChangeIsbn}
               />
            </div>
            <div className="form-group">
               <label>Title: </label>
               <input type="text"
                  className="form-control"
                  value={this.state.title}
                  onChange={this.onChangeTitle}
               />
            </div>
```

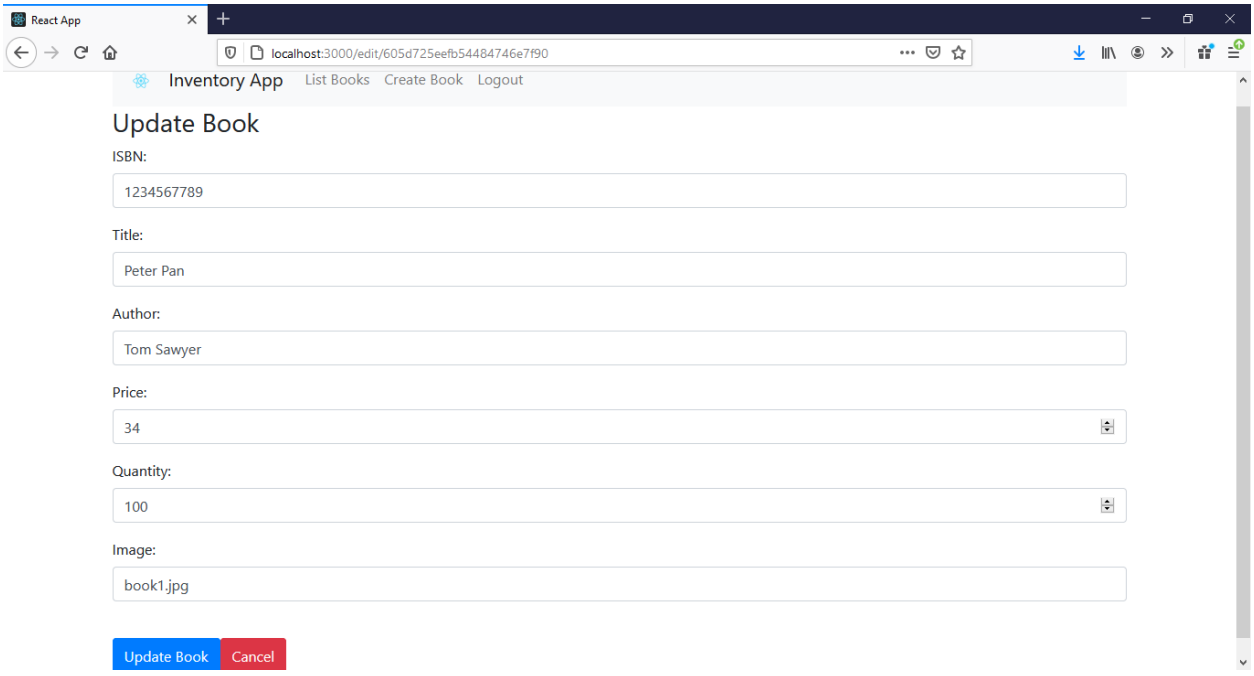
```
<div className="form-group">
  <label>Author: </label>
  <input type="text"
    className="form-control"
    value={this.state.author}
    onChange={this.onChangeAuthor}
  />
</div>
<div className="form-group">
  <label>Price: </label>
  <input type="number"
    min="0" max="1000000" step=".01"
    className="form-control"
    value={this.state.price}
    onChange={this.onChangePrice}
  />
</div>
<div className="form-group">
  <label>Quantity: </label>
  <input type="number"
    min="0" max="1000000"
    className="form-control"
    value={this.state.quantity}
    onChange={this.onChangeQuantity}
  />
</div>
<div className="form-group">
  <label>Image: </label>
  <input type="text"
    className="form-control"
    value={this.state.image}
    onChange={this.onChangeImage}
  />
</div>
<br />
<div className="form-group">
  <input type="submit" value="Update Book" className="btn btn-primary" />
  <input
    type="button" value="Cancel" className="btn btn-danger"
    onClick={(e) => {
      e.preventDefault();
      window.location.href = '/list';
```

```
                }}
              />
            </div>
          </form>
        </div>
    )

  }

}
```

Todo:

Type in or copy and paste in the above code into the editBook.js file,
Run the program you should get something like this:

You should get something like this:



DELETING BOOKS

Books to delete originate from the listBook page delete button. Again the code is very similar to the ListBook and EditBook Components.

When the submit button is pressed the onsubmit button is pressed the onSubmit event handler is called. Using axios we send the delete command to the node.jds server the delete the selected book id is obtained from this.props.match.params.id sent from the listBook component in the route path**.**

```
// delete book from data base
  onSubmit(e) {
    e.preventDefault();

    axios.delete(API_BASE_URL + '/api/book/' + this.props.match.params.id)
      .then(res => {
        console.log(res.data);
        this.props.history.push('/list');
      })
      .catch(function (error) {
        console.log(error)
        //alert(JSON.stringify(error));
      });

    this.setState(prevState => ({
      ...prevState,
      'successMessage': 'Book not Deleted. Try again'
    }))
  }
```

Here is the updated DeleteBook component.

```
// deletebook.js
// delete book from data base

import React, { Component } from 'react';
import axios from 'axios';
const API_BASE_URL = 'http://localhost:4000';
```

```
export default class DeleteBook extends Component {

  constructor(props) {
    super(props);

    this.onSubmit = this.onSubmit.bind(this);

    this.state = {
      isbn: '',
      title: '',
      author: '',
      price: 0,
      quantity: 0,
      image: '',
      successMessage: ''
    }
  }

  // get book from data base
  componentDidMount() {
    //alert(this.props.match.params.id);
    axios.get(API_BASE_URL + '/api/book/' + this.props.match.params.id)
      .then(response => {

        //alert(response);
        this.setState({
          isbn: response.data.data.isbn,
          title: response.data.data.title,
          author: response.data.data.author,
          price: response.data.data.price,
          quantity: response.data.data.quantity,
          image: response.data.data.image
        })
      })
      .catch(function (error) {
        console.log(error)

      })
  }
```

```
// delete book from data base
onSubmit(e) {
  e.preventDefault();

  axios.delete(API_BASE_URL + '/api/book/' + this.props.match.params.id)
    .then(res => {
      console.log(res.data);
      this.props.history.push('/list');
    })
    .catch(function (error) {
      console.log(error)
      //alert(JSON.stringify(error));
    });

  this.setState(prevState => ({
    ...prevState,
    'successMessage': 'Book not Deleted. Try again'
  }))
}

// show book to delete
render() {

  return (
    <div>
      <h3>Delete Book</h3>

      <form onSubmit={this.onSubmit}>
          <div className="form-group">
            <label>ISBN: </label>
            <input type="text"
              className="form-control"
              value={this.state.isbn}
            />
          </div>
          <div className="form-group">
            <label>Title: </label>
            <input type="text"
              className="form-control"
              value={this.state.title}
            />
          </div>
```

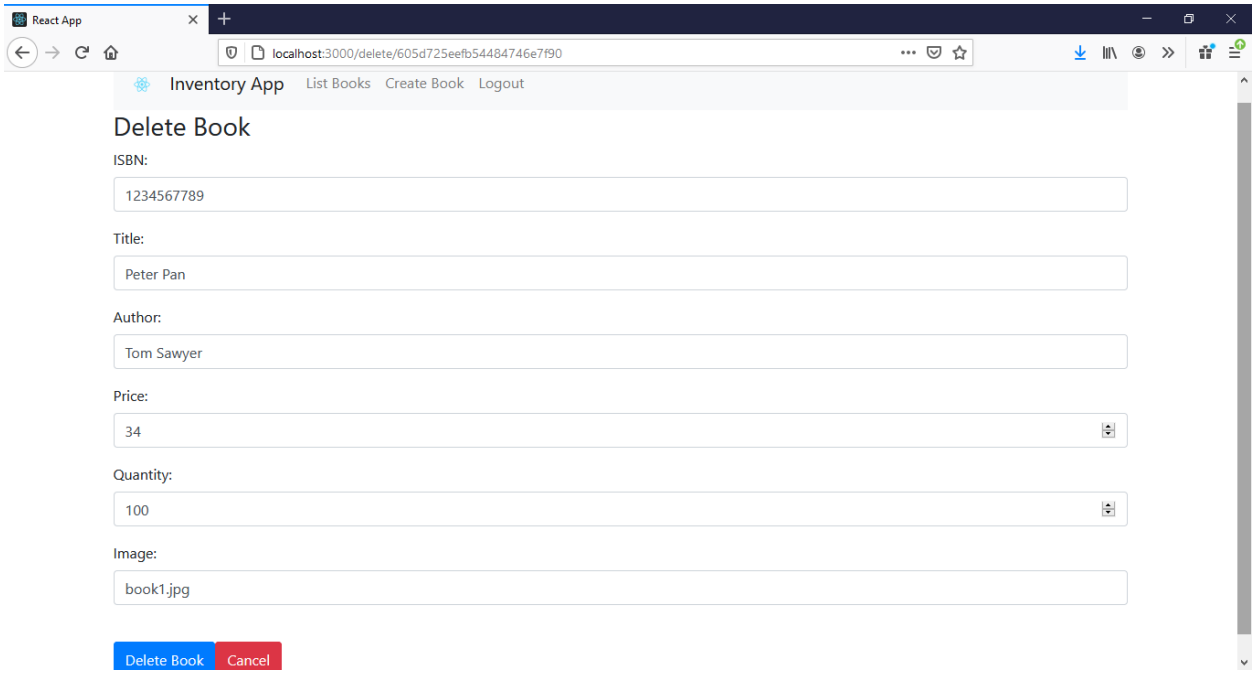
```
    <div className="form-group">
      <label>Author: </label>
      <input type="text"
        className="form-control"
        value={this.state.author}
      />
    </div>
    <div className="form-group">
      <label>Price: </label>
      <input type="number"
        min="0" max="1000000" step=".01"
        className="form-control"
        value={this.state.price}
        onChange={this.onChangePrice}
      />
    </div>
    <div className="form-group">
      <label>Quantity: </label>
      <input type="number"
        min="0" max="1000000" step="1"
        className="form-control"
        value={this.state.quantity}
      />
    </div>
    <div className="form-group">
      <label>Image: </label>
      <input type="text"
        className="form-control"
        value={this.state.image}
      />
    </div>
    <br />
    <div className="form-group">
      <input type="submit" value="Delete Book" className="btn btn-primary" />
    <input
      type="button" value="Cancel" className="btn btn-danger"
      onClick={(e) => {
        e.preventDefault();
        window.location.href = '/list';
      }}
    />
  </div>
```

```
            <div className="alert alert-success mt-2"
                 style={{ display: this.state.successMessage ? 'block' : 'none' }} role="alert">
                {this.state.successMessage}
            </div>
          </form>
        </div>
      )
    }
}
```

Todo:

Type in or copy and paste in the above code into the editBook.js file,
Run the program you should get something like this:

You should get something like this:



MERN LESSON 12 HOMEWORK

Combine the ListBook and EditBook Components together into one app so that when the Edit button is pressed the row of values turn into a textbox of values that you can edit. The edit button text turns to update. When the update button is pressed the book row data gets send to the node.js server for update. The update button return backs and says edit and the text box disappear and the update data id displayed. Add validating to when entering data into the edit text boxes. Change the **DeleteComponent** int a popup.

LESSON 13 Mern Project

Add additional page Edit User that will contain the features of lesson 12 homework.

END