NodeJS BookStore Last Update: June 1, 2021

Build a bookstore using Javascript,NodejS and MongoDB

Conventions used in these lessons:
bold - headings, keywords, code

italics - code syntax

underline - important words

These lessons are provide free of charge as a courtesy to you.
If you need additional help we charge $15 per hour.
You will need to send a request to students@cstutoring.com to book an available day and time.

Lesson1 BookStore Overview

The Nodejs Bookstore allows someone to select and order from a book store using **JavaScript**, **nodejs** and **mongoDB** data base.

Nodejs is a JavaScript server that allows a web browser to communicate with it. Mongo db is a data base the works well with nodejs . We build our book store using nodejs 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. Download 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 install with no problems.

If you allready 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 bookstore-app folder

Make a top level folder called **bookstore-app** in your C drive.

Open up a msdos prompt and type in:

```
C:\Users\ADMIN>cd ..

C:\Users>cd ..

C:\>mkdir bookstore-app

C:\>cd bookstore-app
```

Step 3 initialize the bookstore-app folder to run node.js

You use the **npm init** command to initialize the bookstore-app folder/

We use the –y option when means accept all default values.

Make sure you are in the **bookstore-app** before proceeding.

```
C:\bookdtore-app>npm init –y

Wrote to C:\inventory-app\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 Install express:

D:\bookstore-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

Express is installed so a web browser can talk easier to the node.js server.

Step 5 Write the testnodejs.js program

Open notepad and copy/paste or type in the following program **testnodejs.js** and place in the bookstore-app folder.

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

```
// testnodejs.js
// test connection to nodejs server

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

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

// use port 5000
const apiPort = 5000;

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

// start the server, listen for requests
app.listen(apiPort, () => console.log(`Bookstore App 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 testnodejs.js**

You should get something like this:

Bookstore App running on port 5000

Open up a web browser at

http://localhost:5000/

You should get something like this:

INSTALLING MONGODB

We use the MongoDB as our 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. A Service runs automatically on your computer and can be easily started and stopped in your services control panel.

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 4.4
C:\Program Files\MongoDB\Server\4.4>cd bin
C:\Program Files\MongoDB\Server\4.4\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.

Step 4: install mongoose

```
D:\bookstore-app>npm install mongoose
npm WARN bookstore-app@1.0.0 No description
npm WARN bookstore-app@1.0.0 No repository field.

+ mongoose@5.12.11
added 29 packages from 92 contributors and audited 79 packages in 4.46s
found 0 vulnerabilities
```

mongoose makes it easier for nodejs to tal to the mongodb database.

Step 5:

Put the following program in a **`testmongodb.js`** in the bookstore-app folder.

```
// testmongodb.js
// test mongodb connection

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

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

```
//Set up mongoose connection
var mongoDBURL = 'mongodb://127.0.0.1/shopping';

// connect to mongo db
mongoose.connect(mongoDBURL, {
    useNewUrlParser: true, useUnifiedTopology: true
}).then(function (db) {
    console.log("You are connected to the shopping database");
}, function (err) {
        console.log('Cannot connected to the shopping database', err)
})

// open connection to shopping database
mongoose.Promise = global.Promise;
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'MongoDB connection error:'));

// use port 5000
const apiPort = 5000;

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

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

step 6: run mongodb test program

From your ms-dos prompt type in the following to start the mongodb test program:

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

You should get something like this:

Bookstore App running on port 5000
You are connected to the shopping database

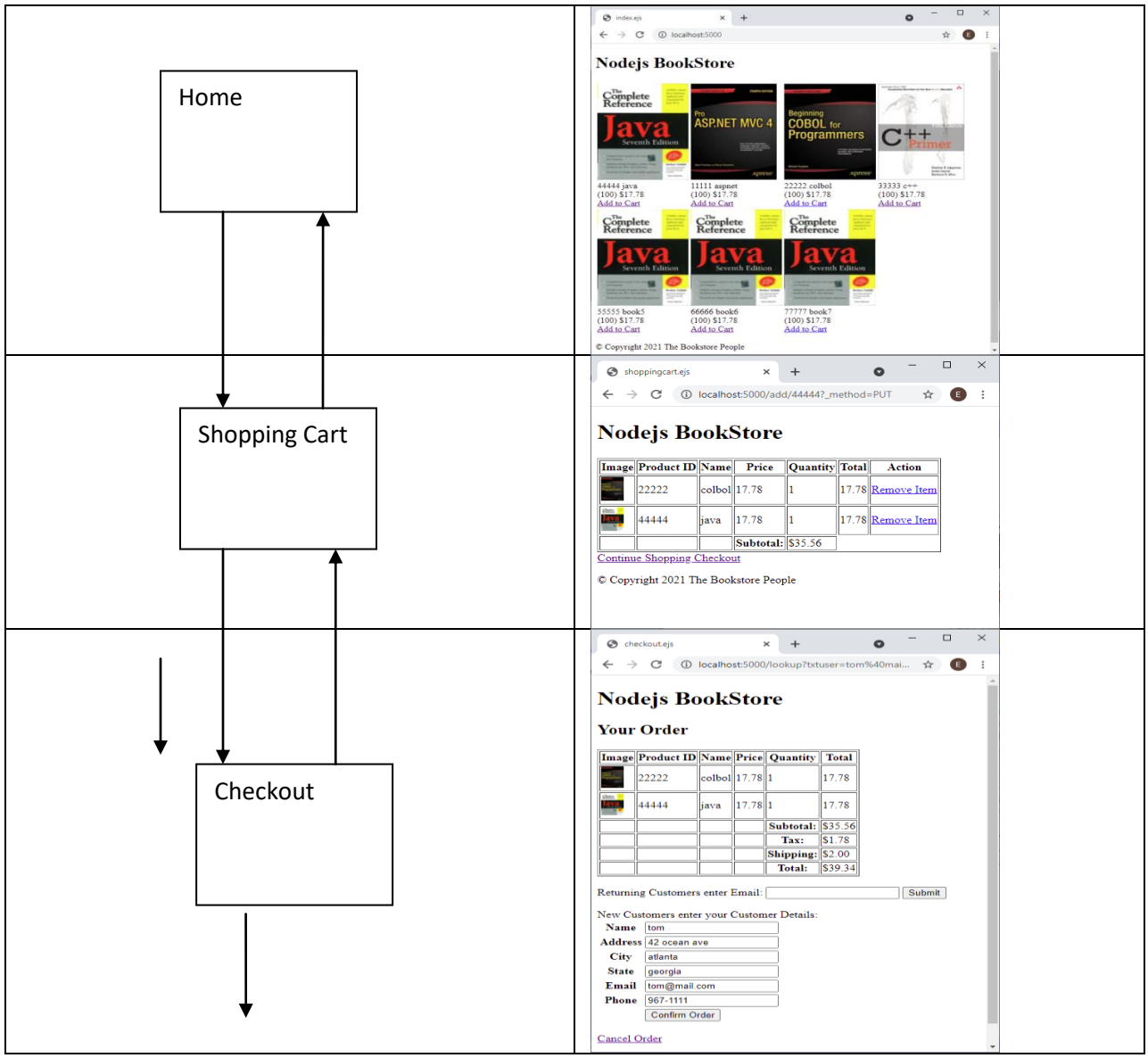
If this is not working then you need to trouble shoot on your own.

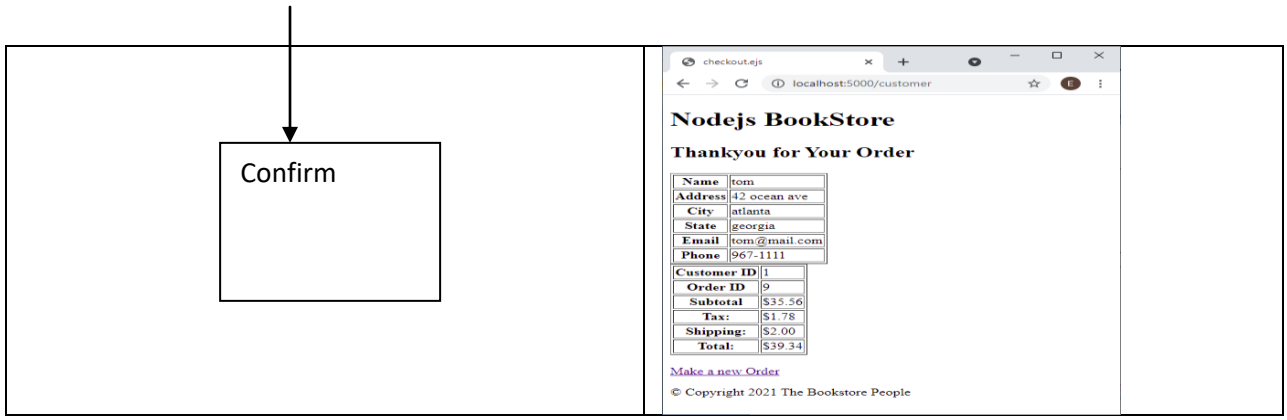
If you cannot resolve this issue then you can book an appointment with one of our technical people at $25 per hour or part of. Send us an email at students@cstutoring.com

BOOKSTORE WEBPAGES

We have 4 web pages as follows:

Webpage	Purpose
Home	Display books for sale
Shopping Cart	Displays books that have been ordered
Checkout	Checkout ordered , show books ordered and totals, enter customer details or returning customer email
Confirm	Place order





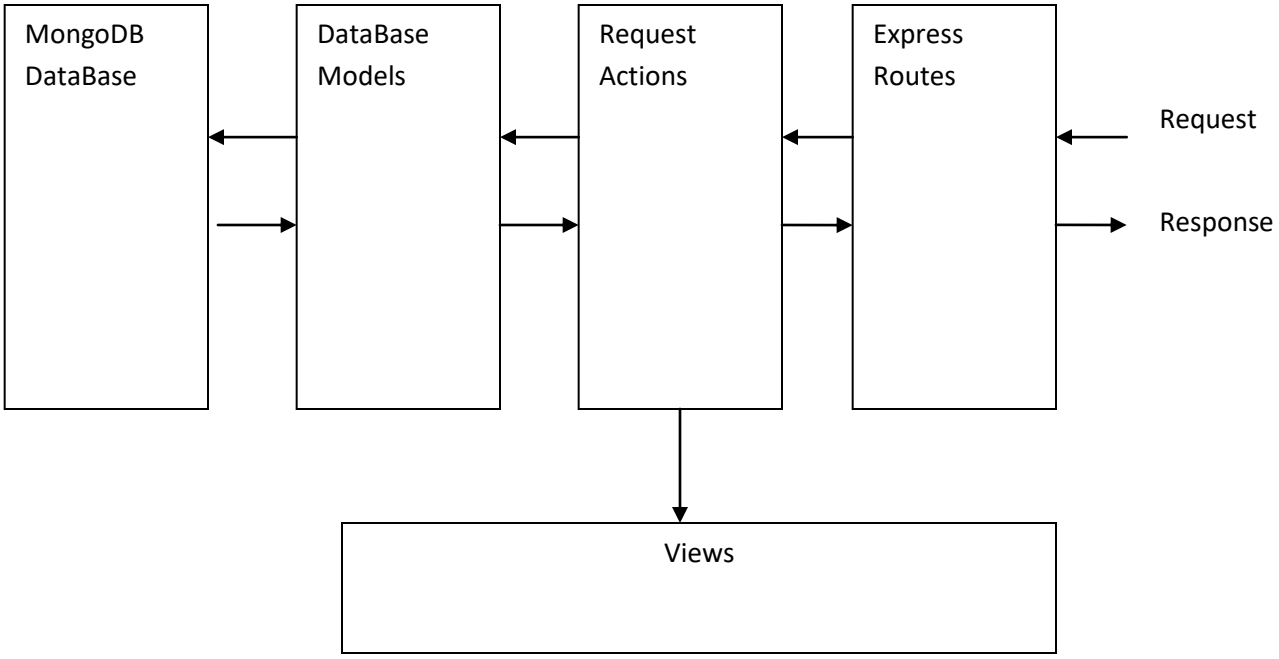
BOOKSTORE COMPONENTS

Our bookstore is using many components supplied by nodejs. We have listed then for you in the following chart:

App Components:

Component	Purpose
Express	Handle incoming requests and provide routing
BodyParser	Parses incoming messages and extracts data objects
Routes	Executes code to process request task
Actions	Execute required task
Views	Present the data to the user as html sent to the web browser
Models	Data base to read and store data in the data base
MongoDB Database	Stores customer, orders, ordertitem,and products

Our block bookstore app diagram would look like this



Action Requests

Each url path request received, must be executed. The express router directs the route to the code section to execute the request. The result of the request (the response) is sent as an data object to one of the webpages (using views) that displays the data. Here are the url paths that the express router must handle:

Path	Purpose
/ /index	Display books for sale
`/add/:productid`	Add product id to shopping cart
`/remove/:productid`	remove product by id from shopping card
`/checkout/`	Checkout order
`/lookup/`	Lookup customer in data base
`/customer/`	Add customer to data base

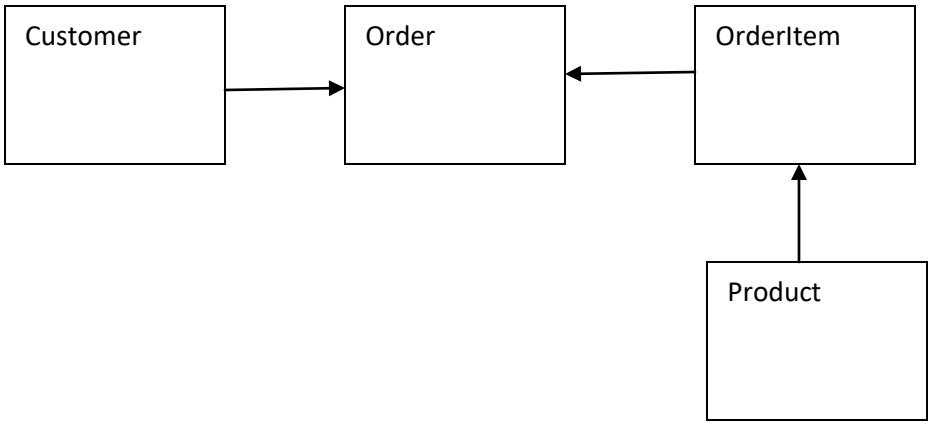
LESSON 2 MONGODB SCHEMA MODELS

We use the MongoDB as our DataBase, it is a Non SQL DataBase and reads and writes JSON.

We will have the following database tables:

Table	Purpose
Customer	Store customer info
Order	Store customer order
OrderItem	Store customer orders items
Product	Store books to be ordered

Our table relationships is as follows:



Mongo Schema Models

A Schema model is needed to describe and create the data base table. A schema is similar to a SQL create statement.

Mongoose is used to make MongoDB schema modes like this:

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema;
```

To make a schema mode we specify the column name , data type and if required.

```
var SchemaModelName = new Schema(
    {
        Column_name: { type: String , required: true },
    }
    }
);
```

You then export the mode specifying the table name

```
module.exports = mongoose.model('table_name',SchemaModel);
```

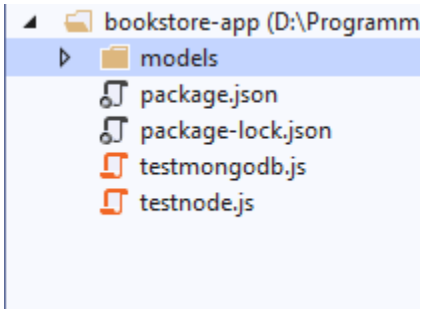
MongoDB available column data types:

Double
String
Object
Array
Binary data
Undefined
Object Id
Boolean
Date
Null

We will now make all the Database Model Schemas:

Todo:

Make a folder called **models** if the bookstore-app folder.



Customer Table

The customer table is used to store the customer for orders

Name	data type	Key type	required
Customeid	String	Primary key	true
Name	String		true
Address	String		true
City	String		true
State	String		true
Email	String		true

Here is the mongodb schema model to create the mongo data base customer table:

```
// customer.js
var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var CustomerSchema = new Schema(
   {
      customerid: { type: String , required: true },
      name: {type: String, required: true},
      address: {type: String, required: true},
      city: {type: String, required: true},
      state: {type: String, required: true},
      email: {type: String, required: true},
      phone: {type: String, required: true},
   }
);

//Export model
module.exports = mongoose.model('Customer',CustomerSchema);
```

Todo:

Put the customer model in a file called customer.js and place in the models folder.

Order Table

The order table contains the customer id, subtotal, tax, shipping and total of the order. Each order get's a unique id.

Name	data type	Key type	required
Orderid	String	primary key	true
customerid	String	foreign key	true
Subtotal	Number		true
Tax	Number		true
Shipping	Number		true
Total	Number		true

Here is the mongodb schema model to create the mongodb data base order table:

```
// order.js
var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var OrderSchema = new Schema(
  {
    orderid: { type: String , required: true },
    customerid: {type: String, required: true},
    subtotal: {type: Number, required: true},
    tax: {type: Number, required: true},
    shipping: {type: Number, required: true},
    total: {type: Number, required: true},
  }
);

//Export model
module.exports = mongoose.model('Order',OrderSchema);
```

Todo:

Put the order model in a file called order.js and place in the models folder.

OrderItem Table

The orderItem table contains the customer id, subtotal, tax, shipping and total of the order. each order gets a unique id.

Name	data type	Key type	required
orderitemid	String	primary key	true
Orderid	String	foreign key	true
Productid	Number	Foreignkey	true
Price	Number		true
Quantity	Number		true
Total	Number		true

Here is the mongodb schema model to create the mongodb data base orderItem table:

```
// orderitem.js
var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var OrderItemSchema = new Schema(
   {
      orderitemid: { type: String , required: true },
      orderid: {type: String, required: true},
      productid: {type: String, required: true},
      price: {type: Number, required: true},
      quantity: {type: Number, required: true},
      total: {type: Number, required: true},
   }
);

//Export model
module.exports = mongoose.model('OrderItem',OrderItemSchema);
```

Todo:

Put the orderitem model in a file called orderitem.js and place in the models folder.

Product Table

The product table store the productid, name, price quantity and image file name

Name	data type	Key type	Required
productid	String	primary key	True
Name	String		True
Price	Number		True
quantity	Number		True
Image	String		True

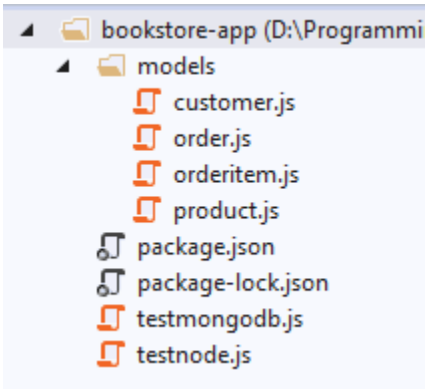
Here is the mongodb product schema model to create the mongodb data base product table:

```
// product.js
var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var ProductSchema = new Schema(
   {
      productid: { type: String , required: true },
      name: {type: String, required: true},
      price: {type: Number, required: true},
      quantity: {type: Number, required: true},
      image: {type: String, required: true},
   }
);

//Export model
module.exports = mongoose.model('Product', ProductSchema);
```

Todo:

Put the product model in a file called product.js and place in the models folder.

Your bookstore-app folder should now look like this:



Lesson2 Homework

Write code to read and write data to each table. Call your homework testTableModels.js

Lesson 3 Populating the MongoDB Database

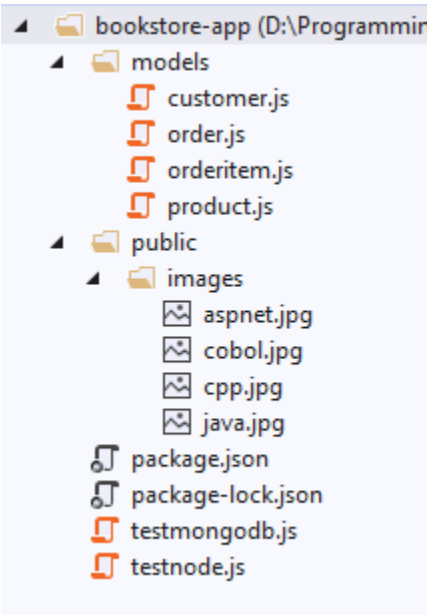
The next thing we need to do is get some images.

You need to get some book images

```
aspnet.jpg
cobol.jpg
cpp.jpg
java.jpg
```

You can find these and download from Google images.

Make a folder called public in the bookstore-app folder and then make a sub folder called images and put the images in it. Your folder should now look something like this:



The next thing we need to do is populate the data base with books to sell.
For convenience we hard code the products with a separate program.
This program lets you populate all four tables with data in out shopping data base:

product
customer
order
orderitem

We will just populate the products table.
The code is quite straight forward.

We first import all the required modules.

```
var async = require('async')
var Product = require('./models/product')
var Customer = require('./models/customer')
var Order = require('./models/order')
var OrderItem = require('./models/orderitem')
```

Then we connect to the **mongodb** using the **mongoose** module.

```
// connect to data base
var mongoose = require('mongoose');
var mongoDB = 'mongodb://127.0.0.1/shopping';
mongoose.connect(mongoDB, { useNewUrlParser: true });
mongoose.Promise = global.Promise;
var db = mongoose.connection;

mongoose.connection.on('error', console.error.bind(console,
    'MongoDB connection error:'));
```

Next we drop the previous shopping data base so we can start fresh.

```
db.once('open', function () {

    console.log("db connect");

    // drop products table
    db.dropCollection("products", function (err, result) {

        if (err) {

            console.log("error delete collection" + err);

        } else {

            console.log("delete collection success");

        }

    });

});
```

We then make 4 arrays to store our created data so we can display the data, once the data is written to the data base.

```
var products = []
var customers = []
var orders = []
var orderitems = []
```

Then we make 4 functions for saving data to the data base:

```
// save product in products table
function productCreate(productid, name, price, quantity,image,cb)

// save customer in customers table
function customerCreate(customerid, name, address, city, state, email, phone,cb)

// store order in orders table
function orderCreate(orderid, customerid, subtotal, tax, shipping, total, cb)

// store orderItem in orderItem table
function orderItemCreate(orderitemid, orderid, customerid, price, quantity, total, cb)
```

Each function works the same inside like this:

```
    // create product detail object
    productdetail =
    { productid: productid, name: name, price: price, quantity: quantity, image: image }

    // create product object
    var product = new Product(productdetail);

    // save product in data base
    product.save(function (err) {
        if (err) {
            cb(err, null)
            return
        }
        console.log('New Product: ' + product);
        products.push(product)
        cb(null, product)
    }  );
}
```

Lastly we have a create products function to create all our products at once.

```
// create product
function createProducts(cb) {
    async.parallel([

            function(callback) {
                productCreate('11111', 'aspnet',17.78, 100, 'aspnet.jpg', callback);
            },
            function(callback) {
                productCreate('22222', 'colbol',17.78, 100, 'cobol.jpg', callback);
            },
            function(callback) {
                productCreate('33333', 'c++',17.78, 100, 'cpp.jpg', callback);
            },
            function(callback) {
                productCreate('44444',  'java',17.78, 100, 'java.jpg', callback);
            },
            function(callback) {
                productCreate('55555',   'book5',17.78, 100, 'java.jpg', callback);
            },
            function(callback) {
                productCreate('66666', 'book6',17.78, 100, 'java.jpg', callback);
            },
            function(callback) {
                productCreate('77777', 'book7',17.78, 100, 'java.jpg', callback);
            }
        ],
        // optional callback
        cb);
}
```

We execute all our productCreate function calls in parallel so we do not need to wait for each one to finish one by one.

async.parallel means to execute operations that must be performed in parallel.

We must call the createProducts function inside the delete data base callback. A callback is a function called from another function. The call back function is when some operation is complete at a later time. When the data base has finished the delete operation it calls our call back function informing that the delete operation has completed. Therefore we can only create Products after the delete operation has completed or else some created products could be deleted prematurely.

```
    // drop products table
    db.dropCollection("products", function (err, result) {

        if (err) {
            console.log("error delete collection" + err);

        } else {

            console.log("delete collection success");

        }

     // call createProducts
      async.series([
        createProducts,
    ],
    // Optional callback
    function(err, results) {
        if (err) {
            console.log('final Creation error: '+err);
        }
        else {
            console.log('products: '+ products);

        }
        // All done, disconnect from database
        mongoose.connection.close();
    });

    });

});
```

Call back functions are inline anonymous functions that do not need a name, and are placed directly in a argument of a function call.

Here is a call back function that reports success or failure.

```
// Optional callback
    function(err, results) {
        if (err) {
            console.log('final Creation error: '+err);
        }
        else {
            console.log('products: '+ products);

        }
        // All done, disconnect from database
        mongoose.connection.close();
    });
```

Here is the complete code for the populatedb.js file:

```
// populatedb.js
// populate data base
console.log('This script populates shopping db');

var async = require('async')
var Product = require('./models/product')
var Customer = require('./models/customer')
var Order = require('./models/order')
var OrderItem = require('./models/orderitem')

// connect to data base
var mongoose = require('mongoose');
var mongoDB = 'mongodb://127.0.0.1/shopping';
mongoose.connect(mongoDB, { useNewUrlParser: true });
mongoose.Promise = global.Promise;
var db = mongoose.connection;

mongoose.connection.on('error', console.error.bind(console,
    'MongoDB connection error:'));

db.once('open', function () {

    console.log("db connect");

    // drop products table
    db.dropCollection("products", function (err, result) {

        if (err) {

            console.log("error delete collection" + err);

        } else {

            console.log("delete collection success");

        }

     // call createProducts
      async.series([
        createProducts,
    ],
    // Optional callback
    function(err, results) {
        if (err) {
            console.log('FINAL ERR: '+err);
        }
        else {
            console.log('products: '+ products);

        }
```

```
        // All done, disconnect from database
        mongoose.connection.close();
    });

    });

});


// arrays to store created data
var products = []
var customers = []
var orders = []
var orderitems = []


// save product in products table
function productCreate(productid, name, price, quantity,image,cb) {

    // create product detail object
    productdetail =
  { productid: productid, name: name, price: price, quantity: quantity, image: image }

    // create product object
    var product = new Product(productdetail);

    // save product in data base
    product.save(function (err) {
        if (err) {
            cb(err, null)
            return
        }
        console.log('New Product: ' + product);
        products.push(product)
        cb(null, product)
    }  );
}


// save customer in customers table
function customerCreate(customerid, name, address, city, state, email, phone,cb) {

    // create customer details object
    customerdetail = {
        customerid: customerid,
        name: name,
        address: address,
        city: city,
        state: state,
        email: email,
        phone: phone
    }
```

```
    // create customer object
    var customer = new Customer(customerdetail);

    // save customer object in customers table
    customer.save(function (err) {
        if (err) {
            cb(err, null)
            return
        }
        console.log('New Customer: ' + customer);
        customers.push(customer)
        cb(null, customer)
    }  );
}


// store order in orders table
function orderCreate(orderid, customerid, subtotal, tax, shipping, total, cb) {

    // make order detail object
    orderdetail = {
        orderid: orderid, customerid: customerid, subtotal: subtotal, tax: tax,
        shipping: shipping, total:total}

    // make order object
    var order = new Order(orderdetail);

    // save order in order table
    order.save(function (err) {
        if (err) {
            cb(err, null)
            return
        }
        console.log('New Order: ' + order);
        orders.push(order)
        cb(null, order)
    }  );
}

// store orderItem in orderItem table
function orderItemCreate(orderitemid, orderid, customerid, price, quantity, total, cb)
{

    // make orderitemdetail object
    orderitemdetail = {
        orderitemid: orderitemid, orderid: orderid, customerid: customerid,
              price:price, quantity: quantity , total: total}


    // create order item object from schema model
    var orderitem = new OrderItem(orderitemdetail);
```

```
    // save order item in data base
    orderitem.save(function (err) {
        if (err) {
            cb(err, null)
            return
        }
        console.log('New OrderItem: ' + orderitem);
        orderitems.push(orderitem)
        cb(null, orderitem)
    }  );
}

// create product
function createProducts(cb) {
    async.parallel([

            function(callback) {
                productCreate('11111', 'aspnet',17.78, 100, 'aspnet.jpg', callback);
            },
            function(callback) {
                productCreate('22222', 'colbol',17.78, 100, 'cobol.jpg', callback);
            },
            function(callback) {
                productCreate('33333', 'c++',17.78, 100, 'cpp.jpg', callback);
            },
            function(callback) {
                productCreate('44444',  'java',17.78, 100, 'java.jpg', callback);
            },
            function(callback) {
                productCreate('55555',   'book5',17.78, 100, 'java.jpg', callback);
            },
            function(callback) {
                productCreate('66666', 'book6',17.78, 100, 'java.jpg', callback);
            },
            function(callback) {
                productCreate('77777', 'book7',17.78, 100, 'java.jpg', callback);
            }
        ],
        // optional callback
        cb);
    }
```

Todo:

Type in or copy and paste the above code and out into a js file called populatedb.js and put into the bookstore-app folder.

Install async module

D:\ bookstore-app>npm install async
npm WARN bookstore-app@1.0.0 No description
npm WARN bookstore-app@1.0.0 No repository field.

+ async@3.2.0
added 1 package from 1 contributor and audited 80 packages in 1.536s
found 0 vulnerabilities

run populatedb.js:

```
node populatedb.js
```

You should get something like this:

```
This script populates shopping db

db connect
delete collection success
New Product: { _id: 608de5db2b8c2d508842c29a,
  productid: '11111',
  name: 'aspnet',
  price: 17.78,
  quantity: 100,
  image: 'aspnet.jpg',
  __v: 0 }
New Product: { _id: 608de5db2b8c2d508842c29e,
  productid: '55555',
  name: 'book5',
  price: 17.78,
  quantity: 100,
  image: 'java.jpg',
  __v: 0 }
New Product: { _id: 608de5db2b8c2d508842c29d,
  productid: '44444',
  name: 'java',
  price: 17.78,
  quantity: 100,
  image: 'java.jpg',
  __v: 0 }
```

```
New Product: { _id: 608de5db2b8c2d508842c29f,
  productid: '66666',
  name: 'book6',
  price: 17.78,
  quantity: 100,
  image: 'java.jpg',
  __v: 0 }
New Product: { _id: 608de5db2b8c2d508842c29c,
  productid: '33333',
  name: 'c++',
  price: 17.78,
  quantity: 100,
  image: 'cpp.jpg',
  __v: 0 }
New Product: { _id: 608de5db2b8c2d508842c29b,
  productid: '22222',
  name: 'colbol',
  price: 17.78,
  quantity: 100,
  image: 'cobol.jpg',
  __v: 0 }
New Product: { _id: 608de5db2b8c2d508842c2a0,
  productid: '77777',
  name: 'book7',
  price: 17.78,
  quantity: 100,
  image: 'java.jpg',
  __v: 0 }
products: { _id: 608de5db2b8c2d508842c29a,
  productid: '11111',
  name: 'aspnet',
  price: 17.78,
  quantity: 100,
  image: 'aspnet.jpg',
  __v: 0 },{ _id: 608de5db2b8c2d508842c29e,
  productid: '55555',
  name: 'book5',
  price: 17.78,
  quantity: 100,
  image: 'java.jpg',
  __v: 0 },{ _id: 608de5db2b8c2d508842c29d,
  productid: '44444',
  name: 'java',
  price: 17.78,
  quantity: 100,
  image: 'java.jpg',
  __v: 0 },{ _id: 608de5db2b8c2d508842c29f,
```

productid: '66666',
name: 'book6',
price: 17.78,
quantity: 100,
image: 'java.jpg',
__v: 0 },{ _id: 608de5db2b8c2d508842c29c,
productid: '33333',
name: 'c++',
price: 17.78,
quantity: 100,
image: 'cpp.jpg',
__v: 0 },{ _id: 608de5db2b8c2d508842c29b,
productid: '22222',
name: 'colbol',
price: 17.78,
quantity: 100,
image: 'cobol.jpg',
__v: 0 },{ _id: 608de5db2b8c2d508842c2a0,
productid: '77777',
name: 'book7',
price: 17.78,
quantity: 100,
image: 'java.jpg',
__v: 0 }

Make test populatedb file testpopulatedb.js

We can now make a populatedb test program to read the products that were written to the mongodb database using the populated.js program. The products read from the mongodb database will be sent to the web browser for viewing.

todo

Put the following code into a javascript file called the testpopulatedb.js:

```
// testpopulatedb.js
// test products written to mongodb using the populated.js program

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

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

```
//Set up mongoose connection
var mongoDBURL = 'mongodb://127.0.0.1/shopping';
// connect to mongo db
mongoose.connect(mongoDBURL, {
    useNewUrlParser: true, useUnifiedTopology: true
}).then(function (db) {
    console.log("You are connected to the shopping database");
}, function (err) {
    console.log('Cannot connected to the shopping database', err)
})

// open connection to shopping database
mongoose.Promise = global.Promise;
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'MongoDB connection error:'));

// Load product table from data base
require('./models/product');
const products = mongoose.model('Product');

// Load customers table from data base
require('./models/customer');
const customers = mongoose.model('Customer');

// Load orders table from data base
require('./models/order');
const orders = mongoose.model('Order');

// Load customers table from data base
require('./models/orderitem');
const orderitems = mongoose.model('OrderItem');


// use port 5000
const apiPort = 5000;

// return server info as a default get requests
app.get('/', (req, res) => {

    // get array of products from data base and then call index view
    products.find(function (err, products) {
        console.log(products);
        res.send({ products: products });

    });

});

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

todo: run testpopulatedb.js

node testpopulatedb.js

You should get something like this:

Lesson 3 Homework

Update populated.js with some some customers and then update testmongodb.js to display the products and customers on the web browser.

Lesson4 Views

Views render HTML to the web browser. We need views in the app end to display the bookstore information stored in the data base. There are many view engines available to nodejs. We are using EJS one of most easiest templates to use. Each view file ends with the ejs extension. Views are called Embedded JavaScript templates that allow you incorporate JavaScript code with html. The EJS Views work like this:

You put the JavaScript code inside scriplets % JavaScript code %

Example:

```
<% var k=0; %>
```

You can also include other ejs file that have headers and footers code

```
<%- include ('header'); -%>
```

The views are called from a response render function, passing it the view file name and a data object to be used in the view

```
res.render('view_name', { name : value });
```

Our bookstore app has the following views that represent each of our webpage's.

View	**Purpose**
Index	display books for sale
shoppingcart	Display bolds ordered
Checkout	Checkout order
Confirm	Confirm
Header	Display header info
Footer	Display footer info

To use the ejs views you need to install the ejs module:

install ejs

```
D:\bookstore-app>npm install ejs
npm WARN bookstore-app@1.0.0 No description
npm WARN bookstore-app@1.0.0 No repository field.

+ ejs@3.1.6
added 15 packages from 8 contributors and audited 102 packages in 2.248s
found 0 vulnerabilities
```

Todo:

Make a folder called views in your bookstore-app folder

header view

The header view just displays the bookstore title.

```
<!-- views/header.ejs -->
<h1>Nodejs BookStore</h1>
```

Todo:

Type in or copy and paste the above code and put in a file called **header.ejs** and place in the **views** folder.

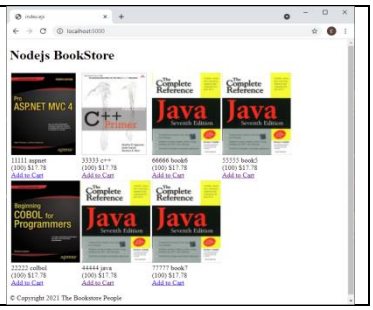
footer view

The footer view just displays the name of the bookstore title.

```
<!-- views/footer.ejs -->
<p>(c) Copyright 2021 The Bookstore People</p>
```

Todo:

Type in or copy and paste the above code and put in a file called **footer.ejs** and place in the **views** folder.

index view	

The index view basically display the images of the books for sale in a grid. It receives an array of book products retrieved from the data base. The image tag is used to display the book image as follows:

```
<img src="images/<%= products[k].image %>" height="200"  /> <br />
```

Below the image we display the product name, quantity and price are displayed on separate lines using the
 tag

```
<%= products[k].productid %> <%= products[k].name %> <br />
(<%= products[k].quantity %>) $<%= products[k].price %> <br />
```

An add to card button is the displayed next and when clicked the add path and productid is send back to the bookstore app.

```
<a href="/add/<%= products[k].productid %>?_method=PUT">Add to Cart</a>
```

Here is the complete index.ejs file:

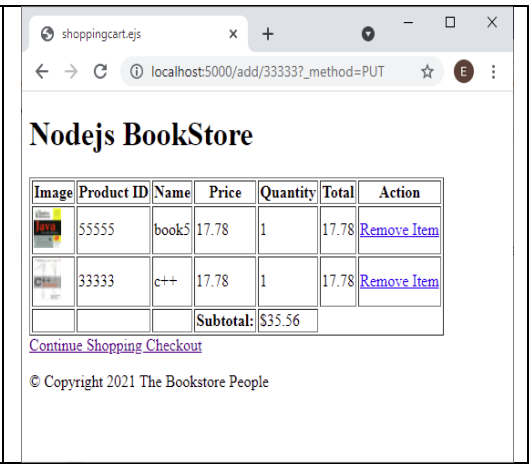
```
<!-- views/index.ejs -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>index.ejs</title>

</head>
<body>

<header>
    <%- include ('header'); -%>
</header>
```

```
<main>
    <div>

        <table>
            <% var k=0; %>
            <% for(var i=0; i < products.length/4; i++) { %>
            <tr>
            <% for(var j=0;k < products.length && j < 4; j++) { %>
                <td>
                    <img src="images/<%= products[k].image %>" height="200"  /> <br />
                    <%= products[k].productid %> <%= products[k].name %> <br />
                    (<%= products[k].quantity %>) $<%= products[k].price %> <br />
                    <a href="/add/<%= products[k].productid %>?_method=PUT">
                       Add to Cart</a>
                    <% k=k+1 %>
                </td>
            <% } %>
            </tr>
            <% } %>
        </table>
    </div>
</main>


<footer>
    <%- include ('footer') -%>
</footer>

</body>
</html>
```

Todo:

Type in or copy and paste the above code and put in a file called **index.ejs** and place in the **views** folder.

Shopping cart view



The shopping cart just displays the products ordered in a table. It receives an array of orders. A remove link lets you remove items by product id.

```
<a href="/remove/<%= cart.items[i].productid %>?_method=PUT">Remove Item</a>
```

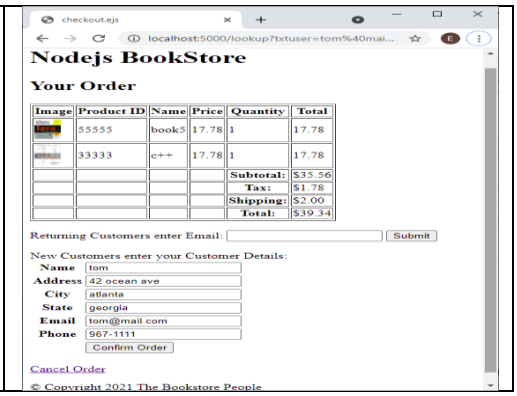
Here is the complete shoppingcart.ejs file.

```
<!-- views/shoppingcart.ejs -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>shoppingcart.ejs</title>
</head>
<body>

<header>
    <%- include ('header'); -%>
</header>
<main>
    <div>
        <table border=="1">
            <tr>
                <th>Image</th>
                <th>Product ID</th>
                <th>Name</th>
                <th>Price</th>
                <th>Quantity</th>
                <th>Total</th>
                <th>Action</th>
            </tr>
```

```
            <% for(var i=0; i < cart.items.length; i++) { %>
            <tr>
                <td>
                    <img src="../images/<%= cart.items[i].image %>" width="30"  />
                </td>
                <td>
                    <%= cart.items[i].productid %>
                </td>
                <td>
                    <%= cart.items[i].name %>
                </td>

                <td>
                    <%= cart.items[i].price.toFixed(2) %>
                </td>

                <td>
                    <%= cart.items[i].quantity %>
                </td>

                <td>
                    <%= cart.items[i].total.toFixed(2) %>
                </td>

                <td>
                    <a href="/remove/<%= cart.items[i].productid %>?_method=PUT">
                            Remove Item</a>
                </td>

            </tr>
            <% } %>

            <tr>
                <th></th>
                <th></th>
                <th></th>
                <th>Subtotal:</th>
               <td>$<%= cart.subtotal.toFixed(2) %></td>
            </tr>
        </table>
    </div>
    <a href="/index" /> Continue Shopping </a>
    <a href="/checkout" /> Checkout </a>
</main>

<footer>
    <%- include ('footer') -%>
</footer>
</body>
</html>
```

Todo:

Type in or copy and paste the above code and put in a file called **shoppingcart.ejs** and place in the **views** folder.

checkout view	

The checkout view displays the productid, name, price and quantity and images of the books for ordered in a table. It receives an array of book products ordered retrieved from the data base. A for loop lets you loop through all the books ordered for display:

```
<% for(var i=0; i < cart.items.length; i++) { %>
    <% for(var i=0; i < cart.items.length; i++) { %>
    <tr>
      <td>
        <img src="../images/<%= cart.items[i].image %>" width="30"  />
      </td>
      <td>
        <%= cart.items[i].productid %>
      </td>
      <td>
        <%= cart.items[i].name %>
      </td>
      <td>
        <%= cart.items[i].price.toFixed(2) %>
      </td>
      <td>
        <%= cart.items[i].quantity %>
      </td>

      <td>
        <%= cart.items[i].total.toFixed(2) %>
      </td>
    </tr>
    <% } %>
```

Next the subtotal, tax, shipping and total are displayed.

A previous customer may look up in the data base by email to supply all the customers details name, address, city, state, email and phone.

```
    <form action="/lookup">

        Returning Customers enter Email: <input type="text" name="txtuser" />
        <input type="submit" /> <br />

    </form>
```

Others wise the customer will enter the details in a form.

Here is the complete checkout.ejs file:

```
<!-- views/checkout.ejs -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>checkout.ejs</title>

</head>
<body>

<header>
    <%- include ('header'); -%>
</header>
<main>
<h2>Your Order</h2>

    <div>

        <table border=="1">

            <tr>
                <th>Image</th>
                <th>Product ID</th>
                <th>Name</th>
                <th>Price</th>
                <th>Quantity</th>
                <th>Total</th>

            </tr>
```

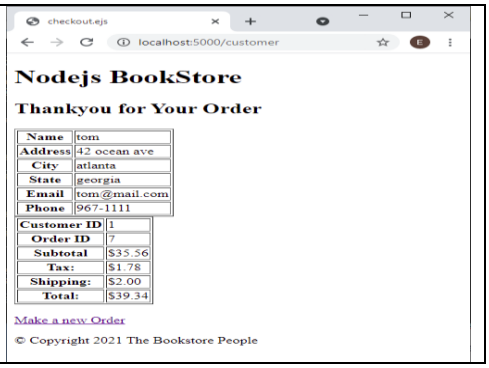
```
<% for(var i=0; i < cart.items.length; i++) { %>
<tr>
    <td>
        <img src="../images/<%= cart.items[i].image %>" width="30"  />
    </td>
    <td>
        <%= cart.items[i].productid %>
    </td>
    <td>
        <%= cart.items[i].name %>
    </td>
    <td>
        <%= cart.items[i].price.toFixed(2) %>
    </td>
    <td>
        <%= cart.items[i].quantity %>
    </td>
    <td>
        <%= cart.items[i].total.toFixed(2) %>
    </td>

</tr>
<% } %>

<tr>

    <th></th>
    <th></th>
    <th></th>
    <th></th>
    <th>Subtotal:</th>
    <td>$<%= cart.subtotal.toFixed(2) %></td>

</tr>

<tr>

    <th></th>
    <th></th>
    <th></th>
    <th></th>
    <th>Tax:</th>
    <td>$<%= cart.tax.toFixed(2) %></td>

</tr>

<tr>

    <th></th>
    <th></th>
    <th></th>
    <th></th>
    <th>Shipping:</th>
    <td>$<%= cart.shipping.toFixed(2) %></td>

</tr>
```

```
            <tr>

                <th></th>
                <th></th>
                <th></th>
                <th></th>
                <th>Total:</th>
                <td>$<%= cart.total.toFixed(2) %></td>

            </tr>

        </table>

    </div>
<br />

    <form action="/lookup">

        Returning Customers enter Email: <input type="text" name="txtuser" />
        <input type="submit" /> <br />


    </form>
<br />

New Customers enter your Customer Details: <br />


    <form action="/customer" method="post" >


        <table>

            <tr><th>Name</th><td><input type="text" name="txtname"
                 value="<%= customer.name %>"/></td></tr>
            <tr><th>Address</th><td><input type="text" name="txtaddress"
                 value="<%= customer.address %>" /></td></tr>
            <tr><th>City</th><td><input type="text" name="txtcity"
                 value="<%= customer.city %>"/></td></tr>
            <tr><th>State</th><td><input type="text" name="txtstate"
                 value="<%= customer.state %>"/></td></tr>
            <tr><th>Email</th><td><input type="text" name="txtemail"
                 value="<%= customer.email %>"/></td></tr>
            <tr><th>Phone</th><td><input type="text" name="txtphone"
                 value="<%= customer.phone %>"/></td></tr>

            <tr><th></th><td><input type="submit" value="Confirm Order" /></td></tr>

        </table>

    </form>
<br />

    <a href="/index" /> Cancel Order </a>

</main>
```

```
<footer>
    <%- include ('footer') -%>
</footer>

</body>
</html>
```

Todo:

Type in or copy and paste the above code and put in a file called **checkout.ejs** and place in the **views** folder.

confirm view	

The confirm view just displays the customer and order details

Here is the complete confirm.ejs file.

```
<!-- views/confirm.ejs -->

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>checkout.ejs</title>

</head>
<body>

<header>
    <%- include ('header'); -%>
</header>

<main>
<h2>Thankyou for Your Order</h2>
```

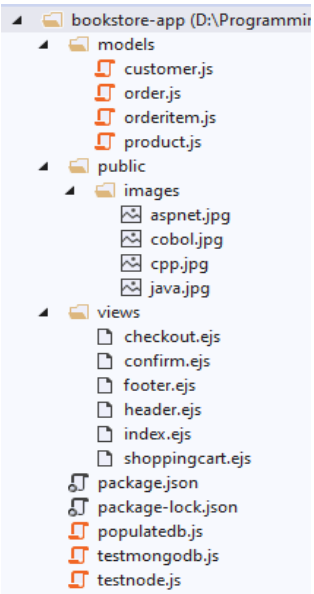
```
<div>

    <table border=="1">

        <tr>
            <th>Name</th>
            <td><%= customer.name %></td>
        </tr>

        <tr>
            <th>Address</th>
            <td><%= customer.address %></td>
        </tr>

        <tr>
            <th>City</th>
            <td><%= customer.city %></td>
        </tr>

        <tr>
            <th>State</th>
            <td><%= customer.state %></td>
        </tr>

        <tr>
            <th>Email</th>
            <td><%= customer.email %></td>
        </tr>

        <tr>
            <th>Phone</th>
            <td><%= customer.phone %></td>
        </tr>

    </table>

    <table border=="1">

        <tr>
            <th>Customer ID</th>
            <td><%= order.customerid %></td>
        </tr>

        <tr>
            <th>Order ID</th>
            <td><%= order.orderid %></td>
        </tr>

        <tr>
            <th>Subtotal</th>
            <td>$<%= order.subtotal.toFixed(2) %></td>
        </tr>

        <tr>
            <th>Tax:</th>
            <td>$<%= order.tax.toFixed(2) %></td>
        </tr>
```

```
            <tr>
                <th>Shipping:</th>
                <td>$<%= order.shipping.toFixed(2) %></td>
            </tr>

            <tr>
                <th>Total:</th>
                <td>$<%= order.total.toFixed(2) %></td>
            </tr>

        </table>

    </div>
<br />

     <a href="/index" /> Make a new Order</a>
</main>


<footer>
    <%- include ('footer') -%>
</footer>

</body>
</html>
```

Todo:

Type in or copy and paste the above code and put in a file called **confirm.ejs** and place in the **views** folder.

Your bookstore-app folder should now look like this:



Test index.ejs view

We now test the index.ejs view to check if it can render some book images. It is basically skeleton code that connects to the data base. Reads the products from the data base and calls index.js to render the book images.

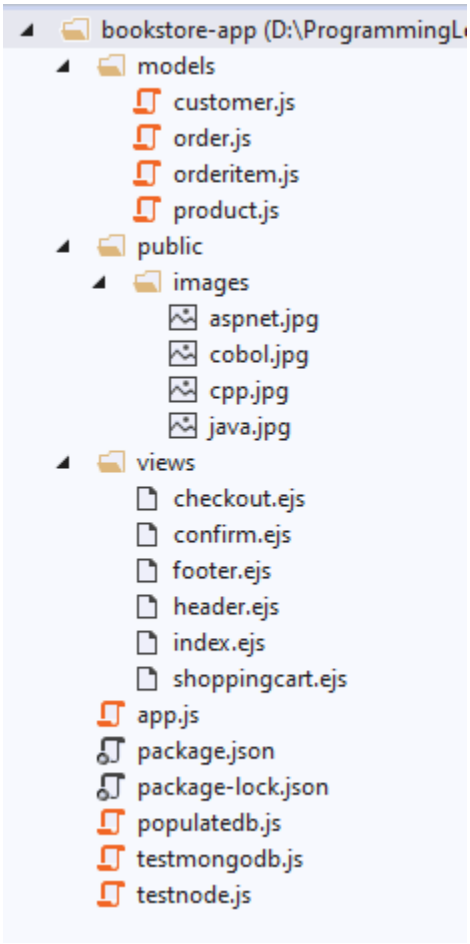
Todo

Make a JavaScript file called testindexview.js and put into your bookstore-app folder and type in or copy/past in the following code:

```
// testindexview.js

// test index.js view

//Import modules
const express = require('express');
const bodyParser = require('body-parser');
var mongoose = require('mongoose');

// make express
const app = express();

// set the view engine to ejs
app.set('view engine', 'ejs');

//Set up mongoose connection
var mongoose = require('mongoose');
var mongoDB = 'mongodb://127.0.0.1/shopping';
mongoose.connect(mongoDB, { useNewUrlParser: true, useUnifiedTopology: true });
mongoose.Promise = global.Promise;
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'MongoDB connection error:'));

// Load product table from data base
require('./models/product');
const products = mongoose.model('Product');

// Load customers table from data base
require('./models/customer');
const customers = mongoose.model('Customer');

// Load orders table from data base
require('./models/order');
const orders = mongoose.model('Order');

// Load customers table from data base
require('./models/orderitem');
const orderitems = mongoose.model('OrderItem');
```

```
// for post data
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

// use to display images and css
app.use(express.static('public'));

// Route's

// main page index.ejs
app.get('/', (req, res) => {

    // get array of products from data base and then call index view
    products.find(function (err, products) {
        console.log(products);
        res.render('index', { products: products });

    });

});

// main page index.ejs
app.get('/index', (req, res) => {

    // get array of products from data base and then call index view
    products.find(function (err, products) {
        console.log(products);
        res.render('index', { products: products });

    });

});


// set port and listen to connections
const port = 5000;

app.listen(port, () => {
    console.log(`Bookstore App started on port ${port}`);
});
```

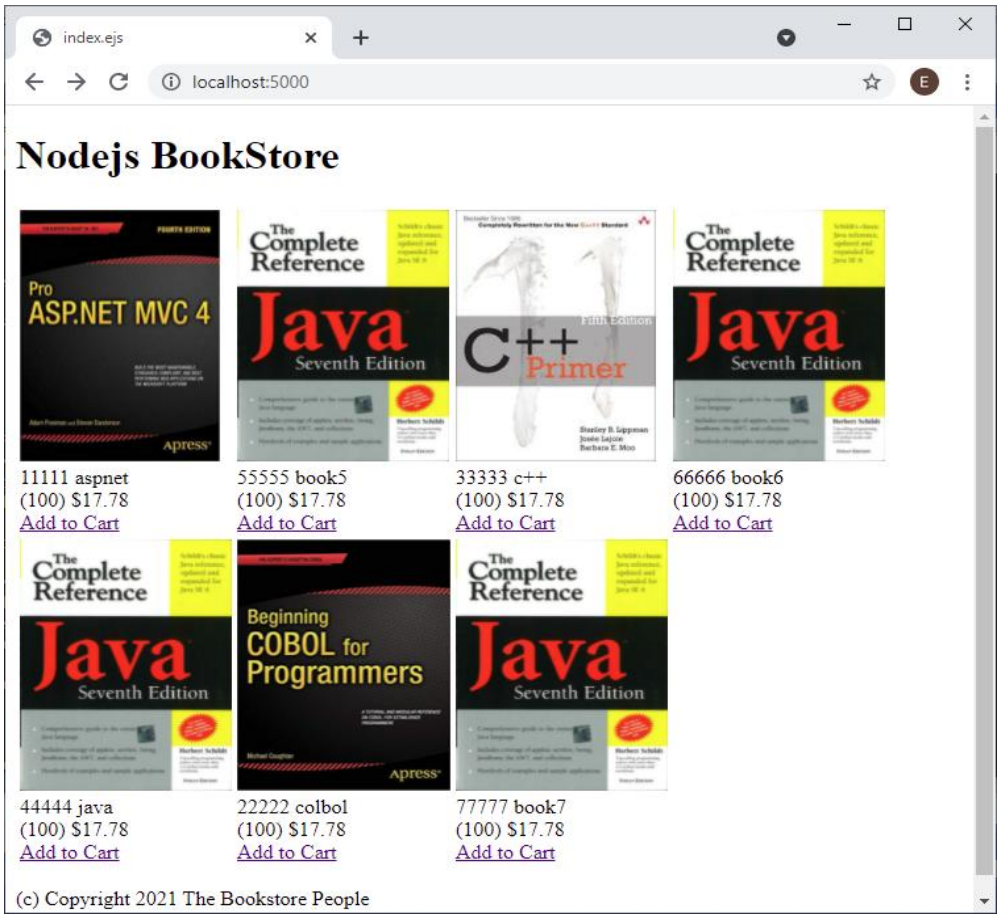
Your bookstore-app folder should now look like this:



You can run like this:

node testindexview.js

You should get something like this:



LESSON 4 Views Homework

Add styling to the view either inline or style sheet.

If you use a style sheet put the style sheet oif a css folder in your public folder

```
./app.js
./public
    /css
        /style.css
```

In your ejs file add the link to your css file

```
<link href="/css/style.css" rel="stylesheet" type="text/css">
```

LESSON5 Shopping Cart class

Our shopping cart class will store the selected ordered books. The shopping cart is a JavaScript class called Cart. All functions of the class are static. The Cart class has no instance variables, so all methods will be static meaning stand alone. The cart items are stores in a Session object. A Session is temporary storage in a server identified by a key. A Session object omly lasts as long as the user is connected to the web server.

Class Cart methods:

```
// add item to cart
addToCart(product, cart);

// remove quantity from cart
// if quantity is 0 remove complete item
removeFromCart(productid = 0, cart);

// return true if item in cart
static inCart(productid = 0, cart);

// calculate total of all items in cart
static calculateTotals(cart);

// clear cart
static emptyCart(cart);

// new cart
static newCart();
```

Here are the methods details:

addToCart(product, cart)

```
purpose: add a product to the cart
inputs:
        product – product to add to cart
        cart – cart object

 description:

        add the product to th cart
        if the product is already in the cart
        then just increments the quantity
```

removeFromCart(product, cart)

```
purpose: remove product from cart
inputs:
        product – product to add to cart
        cart – cart object

 description:

        remove the product the cart
        decrement the quantity
        of decrement is 0 then remove product from cart
```

inCart(productid = 0, cart);

```
purpose: check if product in the cart
inputs:
        product – product to add to cart
        cart – cart object

 description:

        return true if product in cart
```

calculateTotals(cart)

```
purpose:  calculate total of all items in cart

inputs:
     cart – cart object

description:

     calculate subtotal, tax, shipping and total
     of all items in the shopping cart
```

emptyCart(cart);

```
purpose:  remove items in the cars

inputs:
     cart – cart object

description:

     clear all items in the cart
```

```
newCart();

        purpose:  make a new shopping card

        inputs:
             none

        description:

               return a new shopping cart
```

Here is the complete Shopping Cart code:

```
// cart.js
class Cart {

    // add item to cart
    static addToCart(product, cart) {
        //  new item
        if (!this.inCart(product.productid, cart)) {
            cart.items.push({productid:product.productid,name:product.name,
                 price:product.price,quantity:1,
                total:product.price,image:product.image});
        }
        else
           {
            cart.items.forEach(item => {
                if (item.productid === product.productid) {
                    item.quantity+=1;
                    item.total = item.price * item.quantity;
                }
            });
        }

        // re calculate totals
        this.calculateTotals(cart);
    }

    // remove quantity from cart
    // if quantity is 0 remove complete item
    static removeFromCart(productid = 0, cart) {
        for (let i = 0; i < cart.items.length; i++) {
            let item = cart.items[i];
            if (item.productid === productid) {
                if(item.quantity <= 1)
                    cart.items.splice(i, 1);
                else {
                    item.quantity -= 1
                    item.total = item.price * item.quantity;
                }
                this.calculateTotals(cart);
                return
            }
        }

    }
```

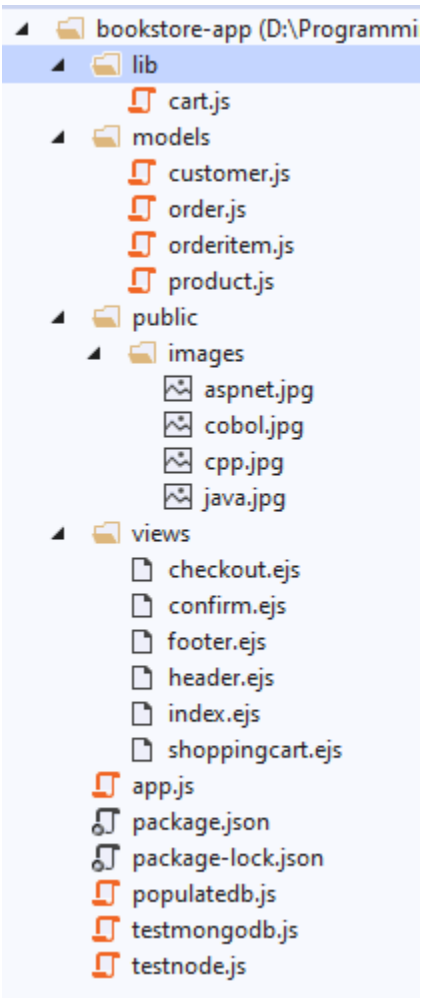
```
    // return true if item in cart
    static inCart(productid = 0, cart) {
        let found = false;
        cart.items.forEach(item => {
            if (item.productid === productid) {
                found = true;
            }
        });
        return found;
    }

    // calculate total of all items in cart
    static calculateTotals(cart) {
        let quanities = 0
        cart.subtotal = 0.00;
        cart.items.forEach(item => {
            cart.subtotal += item.total;
            quanities += item.quantity;
        });

        cart.tax = cart.subtotal * .05;
        cart.shipping = quanities* 1.00;
        cart.total = cart.subtotal + cart.tax + cart.shipping;
    }

    // clear cart
    static emptyCart(cart) {

            cart.items = [];
            cart.subtotal = 0.00;
        cart.tax = 0.00;
            cart.shipping = 0.00;
            cart.total = 0.00;
    }

    // new cart
    static newCart() {

        return(
        {
        items: [],
        subtotal: 0.00,
            tax:0.00,
            shipping:0.00,
            total:0.00

        })
    }
}


module.exports = Cart;
```

todo:

Make a folder called **lib** in the **bookstore-app** folder.

Type in or copy and paste the above Cart code and put in a file called **cart.js** and place in the **lib** folder.

Your bookstore-app folder should now look like this:



Lesson5 Homework

Write test code to test out the Shopping Cart class**.** Call your js file testShoppingCartClass.js.

LESSON 6 Route and Actions

Every request has an action. Each action request has an associated route path and a method. For a **get** method the data is contained in the url request as a name and value pair.

localhost:5000/lookup?txtuser=tom%40mail.com

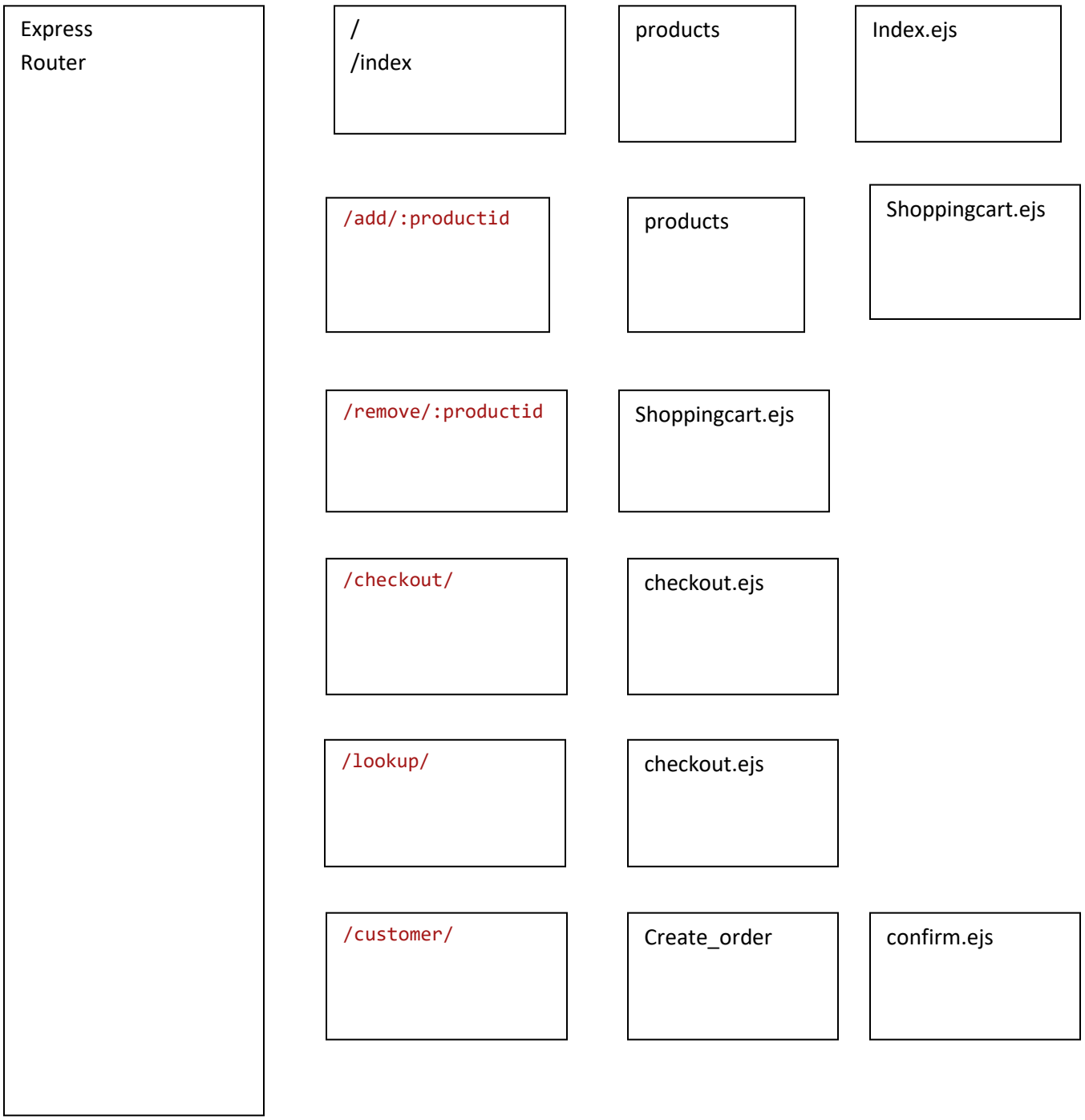
where txtuser is the name and tom@mail.com is the value

For a **post** method the data is contained in the body of the message and is hidden.

Routes are directed by their request path and method either get or post.
Here are the request path and desired purpose:

Method	Path	Purpose
Get	/ /index	Display books for sale
Post	`/add/:productid`	Add product id to shopping cart
Post	`/checkout/`	Checkout order
Get	`/lookup/`	Lookup customer in data base
Post	`/remove/:productid`	remove product by id from shopping card
Post	`/customer/`	Add customer to data base

You could visualize app action as follows, Every route has a associated code to execute , needs database table and view for display.



We now will present the code and description for each route. The app.js file will be quite large but breaking it up into smaller pieces did not work out well. Having everything in one file made things much easier for us.

/ **Display books for sale**

/index

This routes requests the book products to be displayed as the main page. A shopping cart object is created if one does not exists and then stored in a session. A session is use to store objects until destroyed, and is temporary storage mechanism independent from a data base. data base is for long term storage.

Session are similar cookies placed on the web browser to identify the user or store a temporary value to be later retrieved when the uses visits the web site again.

All the products are then read from the data base, the index view is called with the products array to display the books for sale.

```
// main page index.ejs
app.get('/', (req, res) => {

    // create shopping cart and store as a session if not exists
    if (!req.session.cart) {
        req.session.cart = Cart.newCart();
    }

    // get array of products from data base and then call index view
    products.find(function (err, products) {
        console.log(products);
        res.render('index', { products: products });

    });

})


// main page index.ejs
app.get('/index', (req, res) => {

    // create shopping cart and store as a session if not exists
    if (!req.session.cart) {
        req.session.cart = Cart.newCart();
    }
```

```
    // get array of products from data base and then call index view
    products.find(function (err, products) {
        console.log(products);
        res.render('index', { products: products });

    });

}); /add/:productid       Add product id to shopping cart
```

/add/:productid Add product id to shopping cart

The productid is read from the request parameter and then the product is looked up in the data base using the productid. If successful a shopping cart is created if it does not exist., the product is then added to the shopping cart. The shoppingcart view is called with the shopping cart to display the shopping cart books ordered.

```
// add item to a shopping cart
// calls shoppingcart view
app.get('/add/:productid', (req, res) => {

    // get productid from request paramater
    var productid = req.params.productid;

    // get product with product id from data base
    // store product in shopping cart
    products.findOne({
        productid
    })
        .then(product => {
            console.log(product);

            if (!req.session.cart) {
                req.session.cart = Cart.newCart();

            }

            //console.log("cart",req.session.cart);

            // add product to shopping cart
            Cart.addToCart(product, req.session.cart);
            //console.log("cart==>",req.session.cart);

            // call shopping cart view with cart object
            res.render('shoppingcart', { cart: req.session.cart });
        });
});
```

`/remove/:productid` remove product by id from shopping card

The remove action will remove a product from the shopping cart by productid obtained from the request parameter.

```
// remove item from shopping cart
// send updated shopping card to shoppingcart view
app.get('/remove/:productid', (req, res) => {

    //console.log("cart",req.session.cart.items);

    Cart.removeFromCart(req.params.productid, req.session.cart);

    res.render('shoppingcart', { cart: req.session.cart });

});
```

`/checkout/` Checkout order

The checkout route creates a empty customer object and then calls the checkout view with the empty customer object

```
// customer checks out order
// calls checkout view with empty customer object
app.get('/checkout/', (req, res) => {

    // return if no shopping cart
    if (!req.session.cart) return;

    // make empty customer object
    customerdetail = {
        customerid: "",
        name: "",
        address: "",
        city: "",
        state: "",
        email: "",
        phone: ""
    }

    // send shopping cart and empty customer object to checkout view
    res.render('checkout', {
        cart: req.session.cart, customer: new customers(
            customerdetail
        )
    });

});
```

/lookup/ **Lookup customer in data base**

The lookup route gets the email from the customer and looks up the customer in the data base. If the customer is not found then the checkout view is called with the shopping cart and the empty customer object.
If the customer is found in the data base then the checkout view is called with the shopping cart and the filled customer object.

```
// lookup customer in data base
// if customer found call checkout view with customer
// else call checkout view with empty customer
app.get('/lookup', (req, res) => {

    // get email from query
    var email = req.query.txtuser;
    console.log("lookup: " + email);

    // lookup email in customer table
    customers.findOne({
        email
    })
        .then(customer => {
            console.log("customer: " + customer);

            // customer not found, clear customer
            if (customer == null) {

                customerdetail = {
                    customerid: "",
                    name: "",
                    address: "",
                    city: "",
                    state: "",
                    email: "",
                    phone: ""
                }

                // send shopping cart and empty customer to checkout view
                res.render('checkout', {
                    cart: req.session.cart, customer: new customers(
                        customerdetail
                    )
                });

            }

            // send shopping cart and found customer to checkout view
            else {
                res.render('checkout', { cart: req.session.cart, customer: customer });
            }
        });
});
```

/customer/ Add customer to data base

In the customer route the customer details are read from the request body object. The customer is looked up by email. If found then the createorder function is called. If not found then a customer object is created from the customer details and inserted into the data base. Then the order create function is called.

```
// create customer order
// with customer details retrieved from the form
app.post('/customer', (req, res) => {

    // get customer details from body parser
    let name = req.body.txtname;
    let address = req.body.txtaddress;
    let city = req.body.txtcity;
    let state = req.body.txtstate;
    let email = req.body.txtemail;
    let phone = req.body.txtphone;

    let customerid = 0;

    // look for customer in data base
    customers.findOne({
        email
    }).then(customer => {
        console.log("customer: " + customer);

        // existing customer found, so create order
        if (customer != null) {
            create_order(req, res, customer);
        }

        // new customer
        else {

            // make customer id using customer count
            customers.count({}, function (err, count) {
                console.log("Number of users:", count);
                customerid = count + 1;

                // make customer
                customerdetail = {
                    customerid: customerid,
                    name: name,
                    address: address,
                    city: city,
                    state: state,
                    email: email,
                    phone: phone
                }

                // make new customer and store in data base
                var customer = new customers(customerdetail);
                customer.save(function (err) {
```

```
                    if (err) {
                        cb(err, null)
                        return
                    }
                    console.log('New Customer: ' + customer);

                    // create customer order
                    create_order(req, res, customer);
                });

            });
        }
    });

});
```

Create order function

The create order function makes a new orderid from the number of orders. It then gets the shopping cart from the session. For each item in the cart it makes a orderitemdetail object that contains the orderid, customer id, item price quantity and total. From the orderitewmdetail object we make a orderitems object that can be saved to the data base orderitem table. If everything successful then the shopping card is destroyed along with the session.

Here is the function to create the customer order:

```
// create order
// then call confirm.ejs
function create_order(req, res, customer) {

    // get order id
    orders.count({}, function (err, count) {
        console.log("Number of orders:", count);

        // set order id
        let orderid = count + 1;

        // make order
        let cart = req.session.cart;
        let subtotal = cart.subtotal;
        let tax = cart.tax;
        let shipping = cart.shipping;
        let total = cart.total
```

```
// store order items in data base
let i = 0;
async.each(cart.items, function (item, callback) {

    i = i + 1;
    orderitemdetail = {
        orderitemid: i, orderid: orderid, customerid: customer.customerid,
        price: item.price, quantity: item.quantity, total: item.total
    }

    // make order item
    var orderitem = new orderitems(orderitemdetail);

    console.log('New OrderItem: ' + orderitem);

    // save order item in data base
    orderitem.save(function (err) {
        if (err) {
            callback(err)
        }

        else {

            callback()
        }
    });
}, function (err) {
    if (err) {
        // skip errors
    }
    else {
        // all order items should be in db
    }
});

// make order detail object
orderdetail = {
    orderid: orderid,
    customerid: customer.customerid,
    subtotal: subtotal,
    tax: tax,
    shipping: shipping,
    total: total
};

// make order object
var order = new orders(orderdetail);

// save order
order.save(function (err) {
    if (err) {
        cb(err, null)
        return;
    }
    console.log('New Order: ' + order);
```

```
            // delete session
            Cart.emptyCart(req.session.cart)
            req.session.cart = null;
            req.session.destroy();

            // send order and customer object to confirm view
            res.render('confirm', { order: order, customer: customer });
        });
    });
}
```

LESSON 7 RUNNING THE BOOKSTORE APP

The app.js first imports all the required modules.

We first make the app express object that will be used to handling requests and routing.

```
// make express
const app = express();
```

We will be using the ejs view engine.

```
// set the view engine to ejs
app.set('view engine', 'ejs');
```

Next we connect to the mongodb using mongoose.

```
mongoose.connect(mongoDB, { useNewUrlParser: true });
```

Then we load in all the mongo db schema models using mongoose.

```
// Load product table from data base
require('./models/product');
const products = mongoose.model('Product');

// Load customers table from data base
require('./models/customer');
const customers = mongoose.model('Customer');

// Load orders table from data base
require('./models/order');
const orders = mongoose.model('Order');


// Load customers table from data base
require('./models/orderitem');
const orderitems = mongoose.model('OrderItem');
```

Then we load the Cart class from the lib folder.

```
// load cart function modules
const Cart = require('./lib/cart');
```

We use a session to store the shipping cats, out session needs a key to identify it, we are using the key “secret”.

```
// Express session middleware
app.use(session({
    secret: 'secret',
    resave: true,
    saveUninitialized: true
}));
```

The body parser module parses the incoming body messages from the post method. he extended option allows to choose between parsing the URL-encoded data with the querystring library (when false) or the qs library (when true). We are using querystring.

```
// for post data
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
```

We have images to display and css files to display in the public folder, so we tell the app we have items in the public folder to use.

```
// use to display images and css
app.use(express.static('public'));
```

The app.js listens on port 5000

```
// set port and listen to connections
const port = 5000;

app.listen(port, () => {
    console.log(`Shopping cart started on port ${port}`);
});
```

Here is the complete App.js code:

```
// app.js
// bookstore-app

//Import modules
const express = require('express');
const bodyParser = require('body-parser');
var mongoose = require('mongoose');
const session = require('express-session');
var async = require('async');
```

```
// make exress
const app = express();

// set the view engine to ejs
app.set('view engine', 'ejs');

//Set up mongoose connection
var mongoose = require('mongoose');
var mongoDB = 'mongodb://127.0.0.1/shopping';
mongoose.connect(mongoDB, { useNewUrlParser: true,  useUnifiedTopology: true  });
mongoose.Promise = global.Promise;
var db = mongoose.connection;
db.on('error', console.error.bind(console, 'MongoDB connection error:'));

// Load product table from data base
require('./models/product');
const products = mongoose.model('Product');

// Load customers table from data base
require('./models/customer');
const customers = mongoose.model('Customer');

// Load orders table from data base
require('./models/order');
const orders = mongoose.model('Order');

// Load customers table from data base
require('./models/orderitem');
const orderitems = mongoose.model('OrderItem');

// load card function modules
const Cart = require('./lib/cart');

// Express session middleware
app.use(session({
   secret: 'secret',
   resave: true,
   saveUninitialized: true
}));

// for post data
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
```

```
// use to display images and css
app.use(express.static('public'));

// Route's

// main page index.ejs
app.get('/', (req, res) => {

   // create shopping cart and store as a session if not exists
   if (!req.session.cart) {
      req.session.cart = Cart.newCart();
   }

   // get array of products from data base and then call index view
   products.find(function (err, products) {
      console.log(products);
      res.render('index', { products: products });

   });

});

// main page index.ejs
app.get('/index', (req, res) => {

   // create shopping cart and store as a session if not exists
   if (!req.session.cart) {
      req.session.cart = Cart.newCart();
   }

   // get array of products from data base and then call index view
   products.find(function (err, products) {
      console.log(products);
      res.render('index', { products: products });

   });

});

// add item to a shopping cart
// calls shoppingcart view
app.get('/add/:productid', (req, res) => {

   // get productid from request parameter
   var productid = req.params.productid;
```

```
    // get product with product id from data base
    // store product in shopping cart
    products.findOne({
        productid
    })
        .then(product => {
            console.log(product);

            if (!req.session.cart) {
                req.session.cart = Cart.newCart();
            }

            //console.log("cart",req.session.cart);

            // add product to shopping cart
            Cart.addToCart(product, req.session.cart);
            //console.log("cart==>",req.session.cart);

            // call shopping cart view with cart object
            res.render('shoppingcart', { cart: req.session.cart });
        });
});

// remove item from shopping cart
// send updated shopping card to shoppingcart view
app.get('/remove/:productid', (req, res) => {

    //console.log("cart",req.session.cart.items);

    Cart.removeFromCart(req.params.productid, req.session.cart);

    res.render('shoppingcart', { cart: req.session.cart });

});


// customer checks out order
// calls checkout view with empty customer object
app.get('/checkout/', (req, res) => {

    // return if no shopping cart
    if (!req.session.cart) return;
```

```
    // make empty customer object
    customerdetail = {
        customerid: "",
        name: "",
        address: "",
        city: "",
        state: "",
        email: "",
        phone: ""
    }

    // send shopping cart and empty customer object to checkout view
    res.render('checkout', {
        cart: req.session.cart, customer: new customers(
            customerdetail
        )
    });

});

// lookup customer in data base
// if customer found call checkout view with customer
// else call checkout view with empty customer
app.get('/lookup', (req, res) => {

    // get email from query
    var email = req.query.txtuser;
    console.log("lookup: " + email);

    // lookup email in customer table
    customers.findOne({
        email
    })
        .then(customer => {
            console.log("customer: " + customer);

  // customer not found, clear customer
            if (customer == null) {
                customerdetail = {
                    customerid: "",
                    name: "",
                    address: "",
                    city: "",
                    state: "",
                    email: "",
                    phone: ""
                }
```

```
            // send shopping cart and empty customer to checkout view
            res.render('checkout', {
               cart: req.session.cart, customer: new customers(
                  customerdetail
               )
            });
         }

         // send shopping cart and found customer to checkout view
         else {
            res.render('checkout', { cart: req.session.cart, customer: customer });
         }
      });
});

// create customer order
// with customer details retrieved from the form
app.post('/customer', (req, res) => {

   // get customer details from body parser
   let name = req.body.txtname;
   let address = req.body.txtaddress;
   let city = req.body.txtcity;
   let state = req.body.txtstate;
   let email = req.body.txtemail;
   let phone = req.body.txtphone;

   let customerid = 0;

  // look for customer in data base
   customers.findOne({
      email
   }).then(customer => {
      console.log("customer: " + customer);

      // existing customer found, so create order
      if (customer != null) {
         create_order(req, res, customer);
      }

      // new customer
      else {
         // make customer id using customer count
         customers.count({}, function (err, count) {
            console.log("Number of users:", count);
            customerid = count + 1;
```

```
            // make customer
            customerdetail = {
               customerid: customerid,
               name: name,
               address: address,
               city: city,
               state: state,
               email: email,
               phone: phone
            }

            // make new customer and store in data base
            var customer = new customers(customerdetail);
            customer.save(function (err) {
               if (err) {
                  cb(err, null)
                  return
               }
               console.log('New Customer: ' + customer);

               // create customer order
               create_order(req, res, customer);
            });

         });
      }
   });

});
// create order
// then call confirm.ejs
function create_order(req, res, customer) {

   // get order id
   orders.count({}, function (err, count) {
      console.log("Number of orders:", count);

      // set order id
      let orderid = count + 1;

      // make order
      let cart = req.session.cart;
      let subtotal = cart.subtotal;
      let tax = cart.tax;
      let shipping = cart.shipping;
      let total = cart.total
```

```
    // store order items in data base
    let i = 0;
    async.each(cart.items, function (item, callback) {

      i = i + 1;
      orderitemdetail = {
        orderitemid: i, orderid: orderid, customerid: customer.customerid, price: item.price,
        quantity: item.quantity, total: item.total
      }

      // make order item
      var orderitem = new orderitems(orderitemdetail);

      console.log('New OrderItem: ' + orderitem);

      // save order item in data base
      orderitem.save(function (err) {
        if (err) {
          callback(err)
        }
        else {

          callback()
        }
      });
    }, function (err) {
      if (err) {
        // skip errors
      }
  else {
        // all order items should be in db
      }
    });

    // make order detail object
    orderdetail = {
      orderid: orderid,
      customerid: customer.customerid,
      subtotal: subtotal,
      tax: tax,
      shipping: shipping,
      total: total
    };
```

```
      // make order object
      var order = new orders(orderdetail);

      // save order
      order.save(function (err) {
         if (err) {
            cb(err, null)
            return;
         }
         console.log('New Order: ' + order);

         // delete session
         Cart.emptyCart(req.session.cart)
         req.session.cart = null;
         req.session.destroy();

         // send order and customer object to confirm view
         res.render('confirm', { order: order, customer: customer });
      });
   });
}

// set port and listen to connections
const port = 5000;

app.listen(port, () => {
   console.log(`Bookstore App started on port ${port}`);
});
```

Todo:

Type in or copy paste the above code and place in the app.js file located in the bookstore-app folder

install expression-session

```
D:\bookstore-app>npm install express-session
npm WARN bookstore-app@1.0.0 No description
npm WARN bookstore-app@1.0.0 No repository field.

+ express-session@1.17.2
added 7 packages from 6 contributors and audited 87 packages in 1.714s
found 0 vulnerabilities
```

Running the bookstore-app

Now that we got some books into the data base we can run the bookstore app.

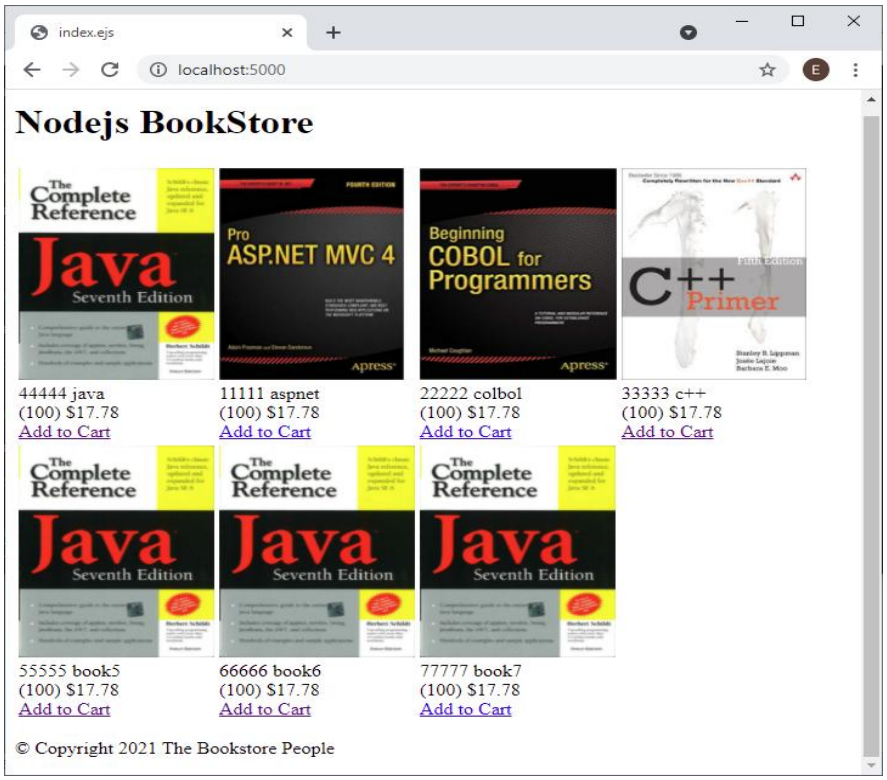
You should get something like this

shopping cart started on port 5000

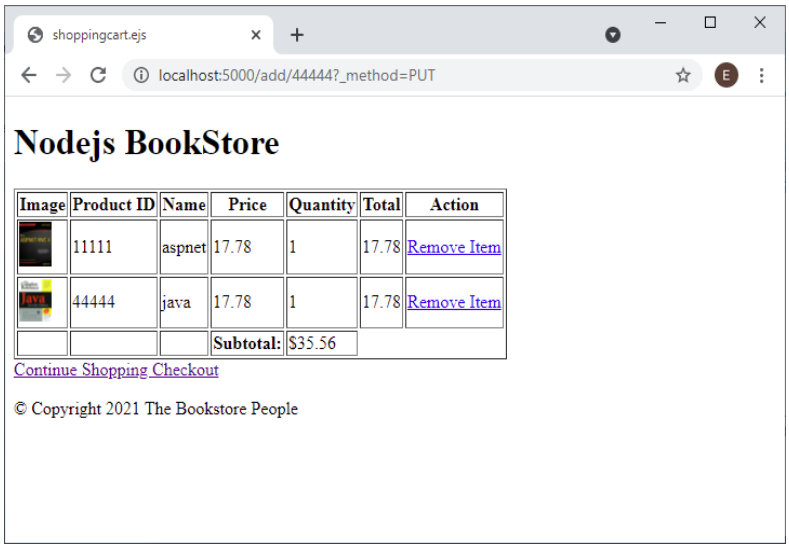
point your web browser at

http://localhost:5000/

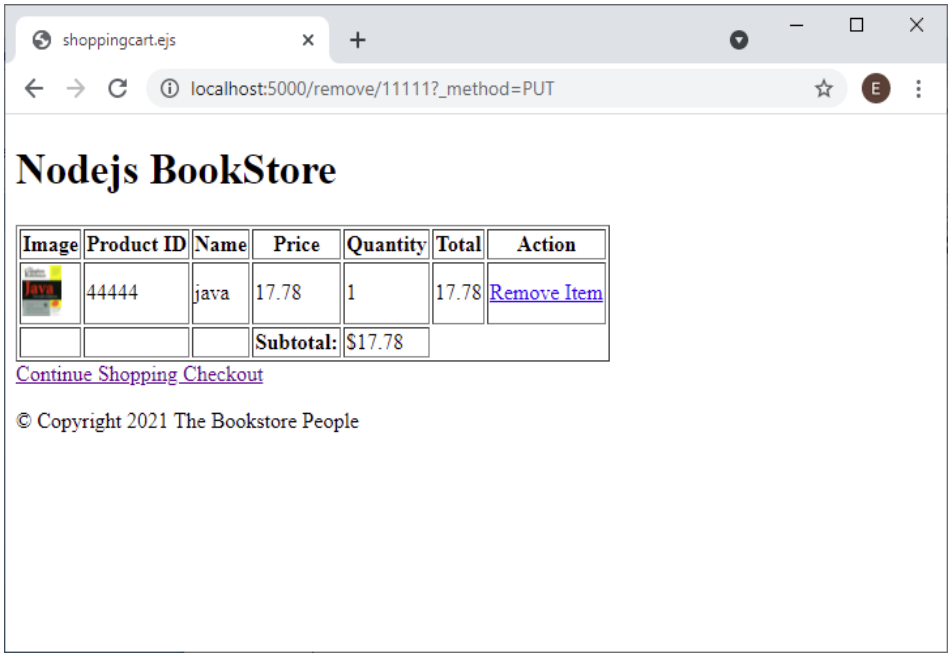
You should get something like this:



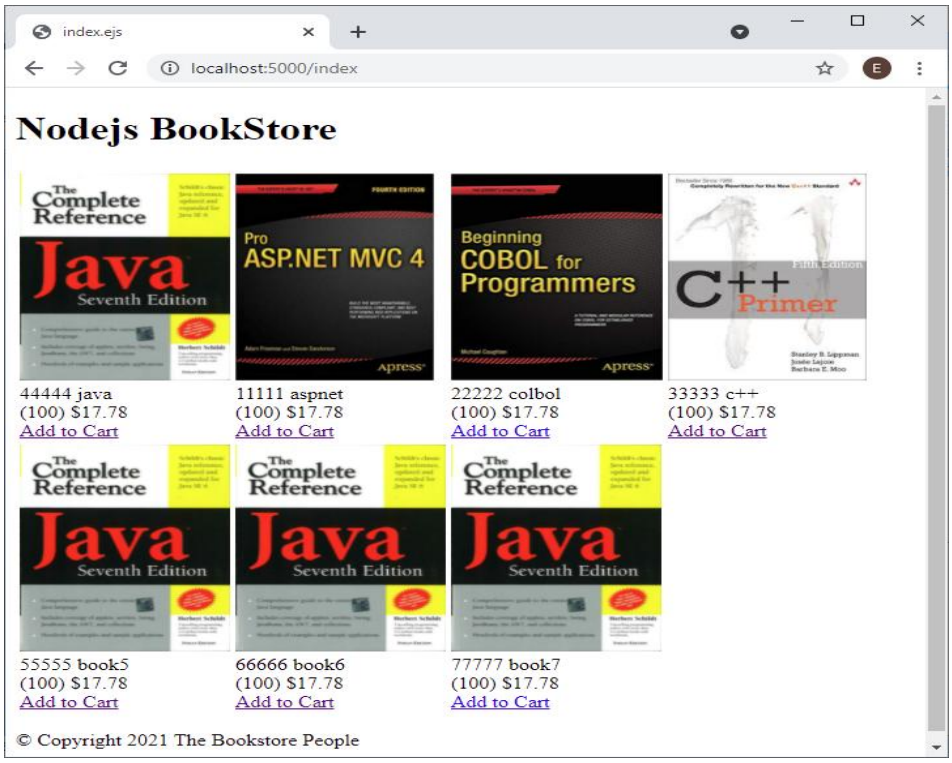
Try to buy some books:



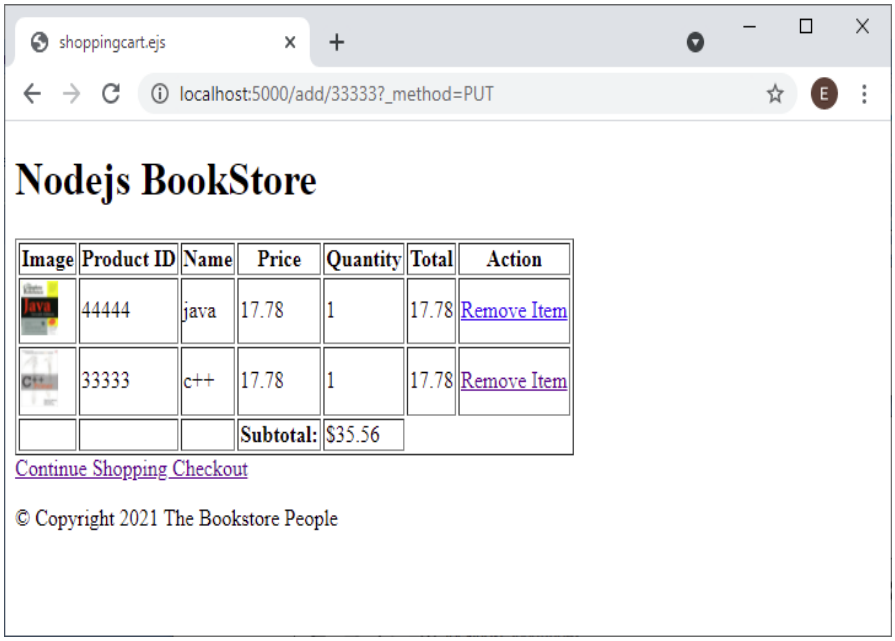
Remove a book:



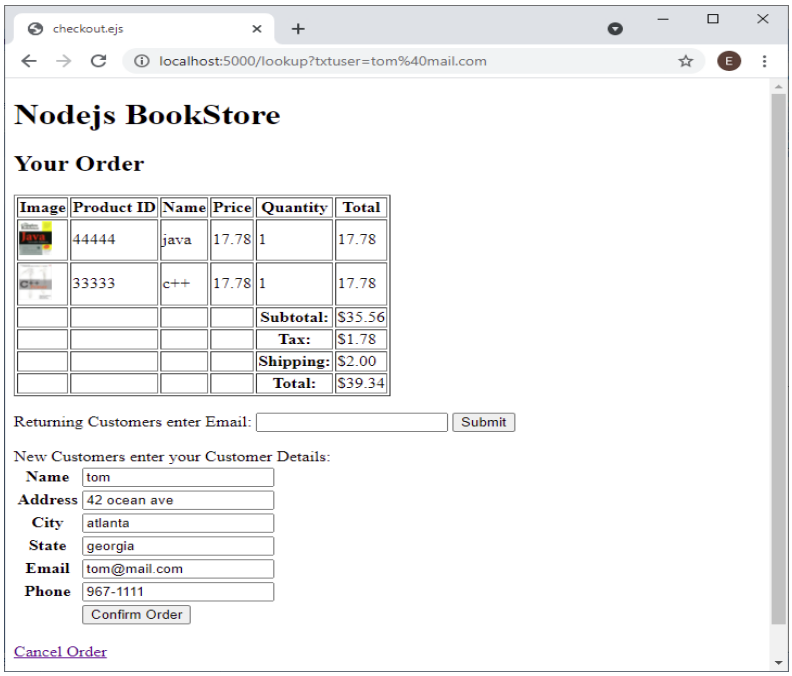
continue shopping



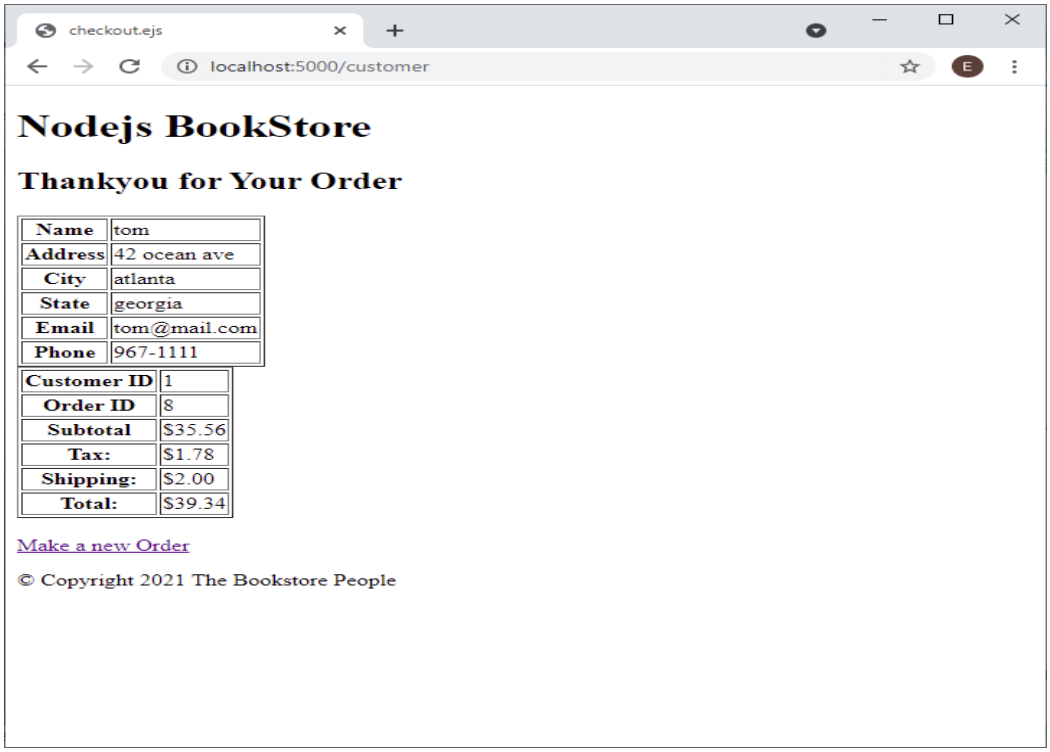
Buy another book



Check out



Confirm order



END