DWAD Advanced Framework Lab

# 1| Preamble and Prerequisites

In this lab we are going to implement an ecommerce application using Express and Vue and online organic food shop portal. The site is divided into both the frontend and backend respectively:

The frontend of the site is implemented using React, and as such, you are expected to have an understanding of the following React concepts:

* Components
* States
* JSX
* Conditional rendering
* Updating arrays without mutation

The backend is implemented using Express and MySQL. You are expected to be familiar with the following concepts:

* Routes
* Handlebars
* Processing forms
* Redirects

# 2| Setup

We are going to quickly create an Express application. To get started, create a new Github repo base on this template: <https://github.com/kunxin-chor/gitpod-full-template> and launch Gitpod from it.

**Step 1| Initialise a new Node application**

At the terminal, type in:

| npm init |
| --- |

And press <ENTER> all the way, skipping all the prompts.

**Step 2| Copy the boilerplate index.js**

Create a new file named *index.js* and add in the following code. For a full explanation for each setup step, please refer to the basic Express module.

| const express = require("express"); const hbs = require("hbs"); const wax = require("wax-on"); require("dotenv").config();  // create an instance of express app let app = express();  // set the view engine app.set("view engine", "hbs");  // static folder app.use(express.static("public"));  // setup wax-on wax.on(hbs.handlebars); wax.setLayoutPath("./views/layouts");  // enable forms app.use(  express.urlencoded({  extended: false  }) );  async function main() {   }  main();  app.listen(3000, () => {  console.log("Server has started"); }); |
| --- |

**Step 3 | Create folders for views and static files**

In the same directory as the *index.js*, create a folder name public

Also create a new folder named views

**Step 4 | Add a testing route**

Within the main function, add the following test route

| app.get('/', (req,res)=>{  res.send("It's alive!")  }) |
| --- |

**Step 5 | Install nodemon**

At the terminal, type in

| npm install -g nodemon |
| --- |

**Step 6| Install dependencies**

We need to add in all the following dependencies before we can run the application:

| yarn add express yarn add hbs yarn add wax-on yarn add dotenv |
| --- |

**Step 7| Run the app**

Type in in the terminal:

| nodemon start |
| --- |

And Gitpod will prompt you to open port 3000 in the browser.

If you are wondering how Nodemon knows what *start* means, check out *package.json*. Inside there, there's an object and in it a key named *scripts.* Defined within the *scripts* object is a *start* key which we specify what commands to run for the script.

**Step 8| Update .gitignore file**

Once the app is working, add *node\_modules* to the last line of the *.gitignore* file.

Part 1: Creating the Landing page

For this lab, we are going to create the landing page. While doing so, we will examine advanced routing concepts with Express, and how to use *controllers* classes to organise our code.

The landing page will consist of the following routes: home, about us and contact us.

# 3| Route Managements

We are going to look at a system of organising our routes, by grouping them into different files. This allows us to scale our application without having twenty or thirty routes all in just one *index.js* file.

**Step 1| Create a folder name *routes***

Create a folder and give it the name *routes*. Make sure you've create the new file in the same folder as your *index.js* file (that is, in the project root folder).

**Step 2| Create a *landing.js* file in the *routes* folder.**

This file will store all the routes related to the landing page.

**Step 3| Add the following code to *landing.js***

| const express = require("express"); const router = express.Router(); // #1 - Create a new express Router  // #2 Add a new route to the Express router router.get('/', (req,res)=>{  res.send("Welcome") })  module.exports = router; // #3 export out the router |
| --- |

Note there are three important concepts here (listed in the comments, highlighted in yellow):

1. First, we create a **new** Express router object
2. We add a new route to the Express router. In this case, the route maps to the */* url.
3. We export out the router object -- this object will have the routes we create in step 2 (currently, there's only one route. There will be more later).

**Step 4| Import the landing page routes**

Back at *index.js*, add the following lines **just** before where we declare the main function.

| // import in routes const landingRoutes = require('./routes/landing');  async function main() {  app.get('/', (req,res)=>{  res.send("It's alive!")  }) } |
| --- |

The *landingRoutes* object above will refer to the router object which we export out from the *landing.js* file.

**Step 5| Remove the test route, and add in the landing page route**

Update your *main* function to:

| async function main() {  app.use('/', landingRoutes); } |
| --- |

Here we are telling Express -- if a URL begins with a single forward slash, then consult the routes registered in the *landingRoutes* object (which we import in from *landingRoutes.js*).

**Step 6| Test the route**

Now with the server running, go to the root URL (that is, a single forward slash). You should see the text *welcome*.

### Conclusion

In this lab, we examine how we can place routes in their own files. This allows us to group routes into files for better organization.

# Parallel Lab 1: Create a Poster Website

For the Parallel Lab, we are going to create a website that sells posters -- be it of bands, albums, movies or games.

1. Start a new Github repository using the *Gitpod Full Template*.
2. Replicate Lab 1, Lab 2 and Lab 3 above

# 4| Add base layout

We are going to create our base layout.

**Step 1 | Create a *base* template file**

Inside your *views* folder, create another folder named *layouts*. Create a file named *base.hbs* within the folder.

**Step 2| Add the Bootstrap 4 template to the base.hbs**

Add the following Bootstrap 4 template to *base.hbs*

| <!doctype html> <html lang="en">  <head>  <!-- Required meta tags -->  <meta charset="utf-8">  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">   <!-- Bootstrap CSS -->  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css" integrity="sha384-Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm" crossorigin="anonymous">   <title>Hello, world!</title>  </head>  <body>  <div class="container">  {{#block 'content'}}{{/block}}  </div>  <!-- Optional JavaScript -->  <!-- jQuery first, then Popper.js, then Bootstrap JS -->  <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js" integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN" crossorigin="anonymous"></script>  <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js" integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q" crossorigin="anonymous"></script>  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js" integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmYl" crossorigin="anonymous"></script>  </body> </html> |
| --- |
|  |

**Step 3 | Creating a hbs file for the landing page**

We are going to have a folder in the *views* folder for each major section of the site. Those folders will be used to group our hbs files.

In the *views* folder, create a new folder name *landing.* Inside there, create a file with the name *index.hbs*

Fill in the content for *index.hbs* as below:

| {{#extends 'base'}}  {{#block 'content'}} <h1>Welcome!</h1> {{/block}}  {{/extends}} |
| --- |

**Step 4| Change the *landing* route to render a template**

Update the route for the / url to be as below:

| router.get('/', (req,res)=>{  res.render('landing/index') }) |
| --- |

# 5| Adding the other pages

We are going to add in the *About Us* and *Contact Us* pages.

**Step 1| Create the hbs files for the two routes**

Duplicate the *index.hbs* twice, and rename them as *contact-us.hbs* and *about-us.hbs* respectively. Change the header in both files to "Contact Us" and "About Us" respectively too.

**Step 2| Add new routes to *landing.js***

| router.get('/', (req,res)=>{  res.render('landing/index') })  router.get('/about-us', (req,res)=>{  res.render('landing/about-us') })  router.get('/contact-us', (req,res)=>{  res.render('landing/contact-us') }) |
| --- |

# Parallel Lab 2: Add Pages to the Posters Site

Replicate lab 4 and 5 for the Posters portal.

Part 2 : Doing the CRUD

In this part of the lab, we are going to create the functionality for the **create, read, update** and **delete** operations for products. To do this, we will be using the feature

We are going to create a database to store the products, orders and user details. We are going to use the **Bookshelf ORM with db-migrate.** ORM in this case stands for **object relational mapping**, and it allows us to represent tables in a SQL database with objects.

# 6| Setting up the database with db-migrate

A *migration* is a set of JavaScript code that makes change to a database. Why would we want to use JavaScript to configure a database instead of using the SQL data definition languages? The reasons are:

1. To make it easier to use different databases
2. So that changes to the databases can be tracked via Git

**Step 1 | Create a new database user**

Log into your MySQL database using mysql -u root, and then key in the following commands:

| CREATE USER 'foo'@'%' IDENTIFIED WITH mysql\_native\_password BY 'bar';  grant all privileges on \*.\* to 'foo'@'%';  FLUSH PRIVILEGES; |
| --- |

We are creating a new user which we will use for this project. It's a good idea to use a new user instead of *root*, because the latter has permissions to all databases which could be a security risk.

**Step 3 | Create the database**

For the purpose of this walkthrough we are creating an organic food website. In your MySQL terminal, type in:

| create database organic[[1]](#footnote-0) |
| --- |

**Step 4 | Setup db-migrate**

We are going to add db-migrate to our project.

| yarn add db-migrate  yarn add db-migrate-mysql  yarn add mysql |
| --- |

**Step 5 | Configure db-migrate**

We are going to use **MySQL** as our database. Since we are using *db-migrate*, it's easy to switch to Postgres later (which we are using).

In the same folder as your *index.js*, create a new file named *database.json*

| {  "dev": {  "driver": "mysql",  "user": "foo",  "password":"bar",  "database":"organic"[[2]](#footnote-1)  } } |
| --- |

What we have to do next is to create a *bash [[3]](#footnote-2)shell* script

Next, in the same directory, create a new bash shell script named *db-migrate.sh*, and add in the following code:

| node node\_modules/db-migrate/bin/db-migrate "$@" |
| --- |

In the terminal, then type in the following so that we have permission to run the script

| chmod +x db-migrate.sh |
| --- |

**Step 6| Create a new product migration**

Now let's add a new *products* table to our newly created database. This table will store the information about our products.

At the terminal, type in:

| ./db-migrate.sh create products |
| --- |

You will see that a new folder name *migrations* have been created, and there is a single file instead. The *create products* part of the command instruct db-migrate to create a new migration file named "xxxx-products.js", where xxxx is an unique number. When creating other tables with db-migrate, you should replaced the highlighted part with the name of your desired table.

**Step 7| Define the products table**

Change the *exports.up* and *exports.down* functions:

| exports.up = function(db) {  return db.createTable('products',{  id: { type: 'int', primaryKey:true, autoIncrement:true, unsigned: true},  name: { type: 'string', length:100, notNull:false},  cost: 'int',  description:'text'  }) };  exports.down = function(db) {  return db.dropTable('products'); }; |
| --- |

This migration file contains the instructions for db-migrate to create the products table. Change the highlighted part above to the name of your own tables when following this step later for your own project.

Note how we are specifying the data types of each column using the JavaScript object syntax. The key will be the name of the column while the value will be attributes of the column, such as its data type, whether it is null or not null and so on. You can find the whole list of attributes here: [SQL API - db-migrate](https://db-migrate.readthedocs.io/en/latest/API/SQL/)

You can find all the supported datatypes at: <https://github.com/db-migrate/shared/blob/master/data_type.js>

Remember to save the file once you are done.

**Step 8| Perform migrations**

In the terminal, type in:

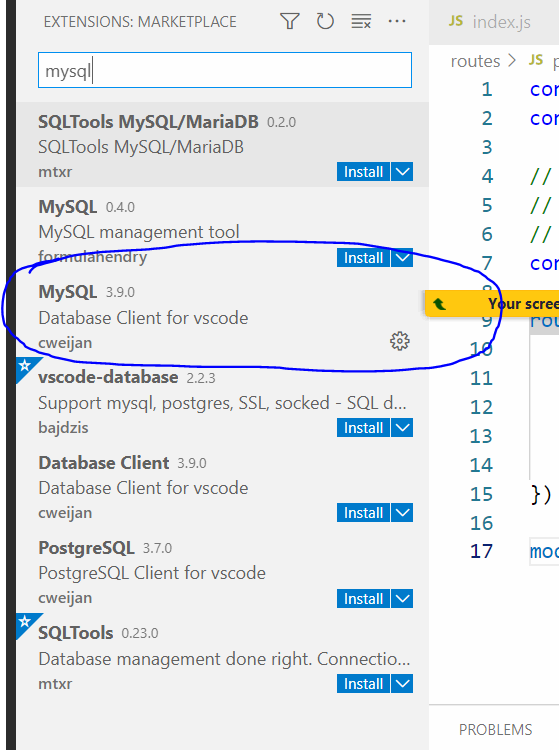
| ./db-migrate.sh up |
| --- |

Check your *organic* database -- it will now have two new tables. One of them is a *migrations* table, the other is your *products* table.

| **Warning:** do not modify the rows in the *migrations* table, and do not delete it. Doing so will cause *db-migrate* to fail. |
| --- |

**Step 9| View the MySql database**

On the left-hand side of your Gitpod, click the *Extension* icon, and install the *MySQL Client* (see below):

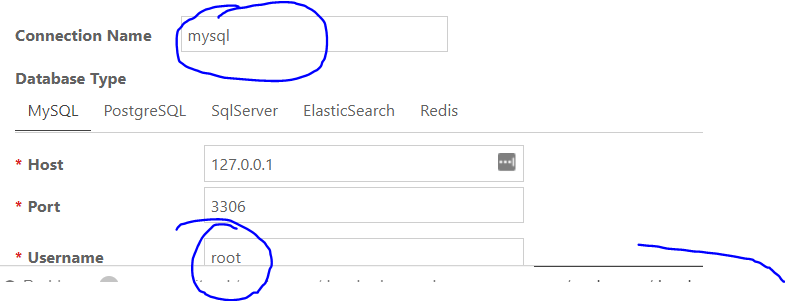


After it has been installed, you will have a new icon at your left hand side bar. Click on this to bring up the MySQL client.

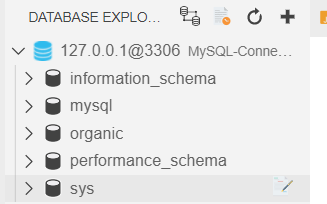
After which, click on the + button, illustrated below:



You will be prompted to enter your MySQL details. Just enter "mysql" for the connection name and "root" for the user, as below:

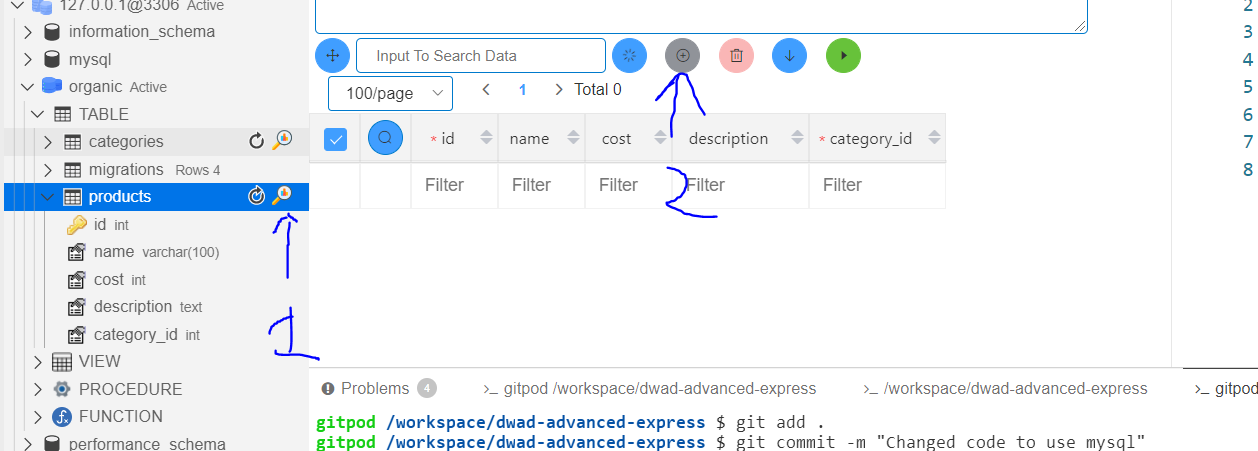


When successfully connected, the client will display all the databases you have in your MySQL database, like below:



**Step 10| Add in a few products**

Click on the magnifying glass next to the *products* table, and you will see the table displayed on the right hand side. Click on the grey + icon to insert new products.



You will see the newly added product on the right-hand side window. The query window can be closed without any issues.

### CONCLUSION

Migration files allow us to create tables in the database without having to use SQL code itself. This allows us to track changes to the database in Git, and also let us use different database technologies seamlessly.

### FURTHER READINGS

* API for creating tables using *db-migrate*: <https://db-migrate.readthedocs.io/en/latest/API/SQL/>
* Video tutorial: <https://www.youtube.com/watch?v=xpfgkIl8oDU>

# Parallel Lab 3: Add in a *posters* table

For this Parallel Lab, you are to set up *db-migrate* and create a *posters* table.

1. Follow the steps as highlighted in Lab 6 for your parallel lab.
2. Name your database *poster-shop*
3. Create a new table named *posters.* The *posters* table should have the following columns

* Title (the title of the poster)
* Cost (cost of the poster, in cents)
* Description (textual description of the poster)
* Date (the date the posted is listed)
* Stock (the quantity of this poster in stock)
* Height (the height of the poster, in CM)
* Width (the width of the poster, in CM)

# 7| Listing all the products

Let's add in the route to create a new product. As usual, we need one route to display the form, and another route to process the form.

We will be using *Bookshelf*, instead of raw SQL queries, to create the product. The reason for using an ORM solution like Bookshelf are:

* SQL syntax independence (our code works with all different kind of SQL databases)
* Security (most ORM solutions have basic database security baked in)

**Step 1| Install Bookshelf**

In the terminal, install the follow dependencies:

| yarn add knex yarn add bookshelf |
| --- |

**Step 2| Create a Bookshelf module**

Create a folder name *bookshelf*, and create an *index.js* file inside it

Add the following code:

| // Setting up the database connection const knex = require('knex')({  client: 'mysql',  connection: {  user: 'foo',  password:'bar',  database:'organic'  } }) const bookshelf = require('bookshelf')(knex)  module.exports = bookshelf; |
| --- |

The user, password and database (highlighted in yellow) should match those in *database.json.* When following this instructions for your parallel lab, be sure to replace the value for the *database* key to be the name of your database.

**Step 3 | Create a *models* module**

A *model* is a JavaScript class that represents one table. An instance of the model (aka, a model instance) represents one **row** in the table.

We are going to create a *models* module which will initialise all the models. Create a folder with the name *models*, and create an *index.js* file inside it.

Add a new model, Product, with the following code:

| const bookshelf = require('../bookshelf')  const Product = bookshelf.model('Product', {  tableName:'products' });  module.exports = { Product }; |
| --- |

The lines in yellow create a new Product model and store it in the *Product* object. Note how we use the *tableName* key in the second argument to set which table the model belongs to. When following this step to create the models for your other tables, be sure to adjust *tableName* accordingly. Also make sure to name the variable and the model appropriately as well.

**Step 4| Create a *products* route**

In the */routes* folder, create a new file named, *products.js*. We are going to register all the routes pertaining to products here.

**Step 5| Add the code to display all products**

Add the following code to *products.js*

| const express = require("express"); const router = express.Router();  // #1 import in the Product model const {Product} = require('../models')  router.get('/', async (req,res)=>{  // #2 - fetch all the products (ie, SELECT \* from products)  let products = await Product.collection().fetch();  res.render('products/index', {  'products': products.toJSON() // #3 - convert collection to JSON  }) })  module.exports = router; |
| --- |

1. We import the Product model from the *models* module. Note that we have to use "../models" to back up one directory.
2. We execute a *SELECT \* from products* using the model.
3. We pass the JSON version of the results to the hbs file.

**Step 6| Create the hbs file**

Create a new folder named *products* in the *views* folder. In the new folder, create a hbs file with the name *index.hbs*.

Add in the following code:

| {{#extends 'base'}}  {{#block 'content'}} <table class="table">  <thead>  <tr>  <th>ID</th>  <th>Name</th>  <th>Cost</th>  <th>Description</th>  </tr>   </thead>  <tbody>  {{#each products}}  <tr>  <td>  {{this.id}}  </td>  <td>  {{this.name}}  </td>  <td>  {{this.cost}}  </td>  <td>  {{this.description}}  </td>  </tr>  {{/each}}  </tbody> </table> {{/block}}  {{/extends}} |
| --- |

**Step 7| Register the route**

Back in *index.js*, register the route before the main function, after the landing routes:

| const landingRoutes = require('./routes/landing') const productRoutes = require('./routes/products') |
| --- |

Inside the *main()* function, register the *productRoutes* router object with the Express app:

| app.use('/products', productRoutes); |
| --- |

Now test the route, and you will be able to display all the products from the product table.

### CONCLUSION

In this lab we installed the Bookshelf ORM, and created a model. Keep in mind that a model represents one table from the database. We also explored how to use a model to a *SELECT \* FROM <table>* query.

Do take note that you just need to repeat step 3 to step 6 for each new table and model that you create in the future.

## Parallel Lab 4

Follow Lab 7 and insert some rows into the *posters* table. Display the posters in an Express route, making use of a *hbs* file.

# 8| Create a new product

We are going to write the code to create a new product. Instead of creating forms manually by hand, we will use a form helper that can help us to automate the process.

**Step 1| Install the *caolan/form-helper***

The form helper that we are going to use is the *Caolan Form Helper.* To begin its setup, at the terminal type in:

| yarn add forms |
| --- |

**Step 2| Create a Form module**

We are going to create a new module for the forms. Create a new folder named *forms* in the same folder as *index.js*, and add its own *index.js* file as well. Then add in the following code:

| // import in caolan forms const forms = require("forms"); // create some shortcuts const fields = forms.fields; const validators = forms.validators;  var bootstrapField = function (name, object) {  if (!Array.isArray(object.widget.classes)) { object.widget.classes = []; }   if (object.widget.classes.indexOf('form-control') === -1) {  object.widget.classes.push('form-control');  }   var validationclass = object.value && !object.error ? 'is-valid' : '';  validationclass = object.error ? 'is-invalid' : validationclass;  if (validationclass) {  object.widget.classes.push(validationclass);  }   var label = object.labelHTML(name);  var error = object.error ? '<div class="invalid-feedback">' + object.error + '</div>' : '';   var widget = object.widget.toHTML(name, object);  return '<div class="form-group">' + label + widget + error + '</div>'; }; |
| --- |

The code above imports in the *forms* module, and also implements the bootstrapFieldhelper function. The function will help us format forms to use CSS classes from Bootstrap4.

**Step 3| Add in a createProductForm function**

Next, still inside *form/index.js*, we are going to add in the code to define a product form. The code is as below:

| const createProductForm = () => {  return forms.create({  'name': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  'cost': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  'description': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  }) }; |
| --- |

**Step 5| Add the exports**

Be sure to add the following exports to the last line in *forms/index.js*

| module.exports = { createProductForm, bootstrapField }; |
| --- |

**Step 6| Create the route to display the form**

Inside *routes/products.js*, add in the following code to import the *createProductForm* and *bootStrapField* functions:

| // import in the Forms const { bootstrapField, createProductForm } = require('../forms'); |
| --- |

**Step 7| Render the form**

Add the following function to render the form.

| router.get('/create', async (req, res) => {  const productForm = createProductForm();  res.render('products/create',{  'form': productForm.toHTML(bootstrapField)  }) }) |
| --- |

Note the line in yellow when we use the *createProductForm* function to get an instance of the form. The line in green will convert the form to its HTML equivalent, using the *bootstrapField* function to format it using Bootstrap4 styles.

After which, also create *create.hbs* in *views/products*, and add in the following code:

| {{#extends 'base'}}  {{#block 'content'}} <h1>Create New Product</h1> <form method="POST">  {{{form}}}  <input type="submit" value="Add Product" class="btn btn-primary"/> </form>  {{/block}}  {{/extends}} |
| --- |

The {{{form}}} (highlighted in yellow) allows us to render out *raw* HTML. This will produce the content of the form.

**Step 8| Process the submitted form**

Add the following function to *routes/products.js*

| router.post('/create', async(req,res)=>{  const productForm = createProductForm();  productForm.handle(req, {  'success': async (form) => {  const product = new Product();  product.set('name', form.data.name);  product.set('cost', form.data.cost);  product.set('description', form.data.description);  await product.save();  res.redirect('/products');   }  }) }) |
| --- |

In this route, we once more create a productForm object. We use its *handle* function to process the request. The second argument is an object which contains a *success* function, which is run when the form is successfully processed.

The first argument to the *success* function is the form itself, and we can retrieve the form data using *form.data*. We then use the form data to create a new instance of the Product model, and then save it.

**Reminder:** An instance of the model represents one *row* in the table.

**Step 9 | Add a validator**

What if the user enters erroneous inputs? For instance, the user might have entered alphabets or words for the cost, when it should just have been numbers. Dealing with such errors is known as *validation*, and we can quickly add *validators* to each form field.

First, update the ProductForm in *form/index.js* to add a validator to the *cost* field. See the highlighted line below for the change.

| const createProductForm = () => {  return forms.create({  'name': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  'cost': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  },  'validators':[validators.integer()]  }),  'description': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  }) }; |
| --- |

If you try to create a new product, but enter text instead of numbers for the cost, the product will not be created.

For a whole list of available validators, check out the documentation for [caolan/forms](https://github.com/caolan/forms).

**Step 10 | Display errors**

Right now we just block the user from creating a product if one or more inputs is invalid, however the user has no idea *why* they are blocked from creating a product and *what* went wrong. So in this step, we are going to display the error messages.

Back at *routes/products.js*, in the route where we process the form, add the following changes:

| router.post('/create', async (req, res) => {  console.log('post /create');  const productForm = createProductForm();  productForm.handle(req, {  'success': async (form) => {  const product = new Product(form.data);  await product.save();  res.redirect('/products');  },  'error': async (form) => {  res.render('products/create', {  'form': form.toHTML(bootstrapField)  })  }  }) }) |
| --- |

To show the error messages, we render the *products/create* hbs file with the processed form. The *bootstrapField* function in *forms/index.js* will format the error messages properly.

# 9| Update an existing product

We can create a new product, and display it in the product listing. The next feature to do will be to *update* an existing product. We can reuse the *createProductForm* function for this.

**Step 1| Write the route to display the form**

Create a new route in the *routes/products.js* as below:

| router.get('/:product\_id/update', async (req, res) => {  // retrieve the product  const productId = req.params.product\_id  const product = await Product.where({  'id': productId  }).fetch({  require: true  });   const productForm = createProductForm();   // fill in the existing values  productForm.fields.name.value = product.get('name');  productForm.fields.cost.value = product.get('cost');  productForm.fields.description.value = product.get('description');   res.render('products/update', {  'form': productForm.toHTML(bootstrapField),  'product': product.toJSON()  })  }) |
| --- |

There are three distinct parts to the code above.

The first part we read in the *product\_id* URL parameter. This stores the *id* of the product that we want to update. We retrieve the product instance with that specific product id and store it in the product variable.

In the second part of the code (the second block of highlighted code), we once again create a *productForm*. However this time round we assign the *value* of each field from its corresponding key in the *product* model instance.

Finally, we send the form and the product variable to the hbs file for rendering.

**Step 2| Create the hbs file to display the form**

Create a new hbs file at  *views/products* and name it as *update.hbs*. Add in the following code:

| {{#extends 'base'}}  {{#block 'content'}} <h1>Update Product: {{product.name}}</h1> <form method="POST">  {{{form}}}  <input type="submit" value="Update Product" class="btn btn-primary"/> </form>  {{/block}}  {{/extends}} |
| --- |

The hbs file is mostly similar to *views/products/create.hbs*, except for some small tweaks: the header is different, and we change the value for the submit button to say "Update Product"

**Step 3| Process the form**

Let's create the route to process the submitted form.

| router.post('/:product\_id/update', async (req, res) => {   // fetch the product that we want to update  const product = await Product.where({  'id': req.params.product\_id  }).fetch({  require: true  });   // process the form  const productForm = createProductForm();  productForm.handle(req, {  'success': async (form) => {  product.set(form.data);  product.save();  res.redirect('/products');  },  'error': async (form) => {  res.render('products/update, {  'form': form.toHTML(bootstrapField),  'product': product.toJSON()  })  }  })  }) |
| --- |

Once again the code is divided into two parts. In the first part we fetch the product by the product id from the URL parameters.

In the second part, we perform the form processing. In the code highlighted in yellow, the form is successfully processed (i.e, no validation errors) and we use the product.set function to overwrite the original product data with the data from the form. Then we instruct the product to save itself.

If there's an error in the form, we just re-render the form to display the error messages.

# 10 | Delete a product

In this step we will finish the last of the CRUD features -- delete. The strategy is to display a form to confirm if the user wishes to delete the product specified by the *product\_id* parameter in the URL. If the user presses on the submit button, then we will actually implement the delete.

**Step 1| Create a route that displays the confirmation message**

In *routes/products.js*, add the following code:

| router.get('/:product\_id/delete', async(req,res)=>{  // fetch the product that we want to delete  const product = await Product.where({  'id': req.params.product\_id  }).fetch({  require: true  });   res.render('products/delete', {  'product': product.toJSON()  })  }); |
| --- |

We simply fetch the product that the user wishes to delete, and send it to the hbs file *products/delete*.

Now create *views/products/delete.hbs* and add in the following code:

| {{#extends 'base'}}  {{#block 'content'}} <h1>Deleting {{product.name}}</h1>  <form method="POST">  <div class="alert alert-danger">  <p>Are you sure you want to delete {{product.name}}?</p>  <input type="submit" class="btn btn-danger" value="Yes"/>  <a href="/products" class="btn btn-success">No</a>  </div> </form> {{/block}} {{/extends}} |
| --- |

We are now able to display the confirmation for deleting a product via specifying its ID in the URL.

**Step 2| Process the delete**

To process the delete, add the following code to *routes/products.js*:

| router.post('/:product\_id/delete', async(req,res)=>{  // fetch the product that we want to delete  const product = await Product.where({  'id': req.params.product\_id  }).fetch({  require: true  });  await product.destroy();  res.redirect('/products') }) |
| --- |

Again, we fetch the *product* that we want to delete, and call the *destroy* function on it.

# 11 | Add in Links to update and delete

Right now the only way to delete and to update is by specifying the *id* in the URL. This won't work for the end-user, so let's modify the *views/products/index.hbs*, where we show all the products, to display links for updating and deleting:

| {{#extends 'base'}}  {{#block 'content'}} <table class="table">  <thead>  <tr>  <th>ID</th>  <th>Name</th>  <th>Cost</th>  <th>Description</th>  <th></th>  </tr>   </thead>  <tbody>  {{#each products}}  <tr>  <td>  {{this.id}}  </td>  <td>  {{this.name}}  </td>  <td>  {{this.cost}}  </td>  <td>  {{this.description}}  </td>  <td>  <a href="/products/{{this.id}}/update" class="btn btn-primary btn-sm">Update</a>  <a href="/products/{{this.id}}/delete" class="btn btn-danger btn-sm">Delete</a>  </td>   </tr>   {{/each}}  </tbody>   </table> {{/block}}  {{/extends}} |
| --- |

The new code is highlighted in yellow.

# Parallel Lab 5

We have explored how to add *CRUD* features for the Organic Food project. Now it's your turn to do the same for the Posters project. Implement create, read, update and delete for the Poster shop.

Part 3: Relationships

In part 3 of this series, we will be exploring how to implement 1 to 1, 1 to many and many to many relationships with Bookshelf ORM.

# 12 | Create a One to Many Relationship with Category

Let's assume that each product belongs to a category; this implies that there is a 1 to many relationship between Category and Product. Let's begin by creating a Category table, and then the Category model.

**Step 1| Create a Category migration**

At the terminal, type in:

| ./db-migrate.sh create categories |
| --- |

And a new migration file will be created.

**Step 2| Add the code to create the table**

In the new migration file, modify the *up* and *down* functions as below:

| exports.up = function(db) {  return db.createTable('categories',{  id: { type: 'int', unsigned: true, primaryKey:true, autoIncrement:true},  name: { type: 'string', length:100},  }) };  exports.down = function(db) {  return db.dropTable('categories') }; |
| --- |

Be sure to save the file.

**Step 3| Run the migration**

In the terminal, type in:

| ./db-migrate.sh up |
| --- |

**Step 4| Create the Category model**

Inside *models/index.js*, add in the following code

| const Category = bookshelf.model('Category',{  tableName: 'categories' }) |
| --- |

And be sure to export out the new Category model too:

| module.exports = {Product, Category}; |
| --- |

**Step 6| Add relationship between Product and Category**

Now a product belongs to one category, so this means we need to add a foreign key to the *products* table that references the primary key of the *categories* table.

First, let's create the migration.

| ./db-migrate.sh create add\_fk\_to\_products |
| --- |

Second, change the *up* function of the newly created migration file:

| exports.up = function(db) {  return db.addColumn('products', 'category\_id', {  type: 'int',  unsigned:true,  notNull : true,  foreignKey: {  name: 'product\_category\_fk',  table: 'categories',  rules: {  onDelete:'cascade',  onUpdate:'restrict'  },  mapping: 'id'  }  }) } |
| --- |

**Next, delete all existing products first.**

Finally, run the migration, after making sure to save the changes to the migration file:

| ./db-migrate.sh up |
| --- |

**NOTE:** You are likely to run into an error at this point, especially if you have already created some products! This is because we cannot add a compulsory FK to products if there are already existing products, because that would cause a contradiction. There are many ways to solve this problem, but the easiest way is to delete all existing products first. We will explore other ways to circumvent this issue later.

**Step 7 | Create some fake categories**

Using the command line client or the extension, create some fake categories. We are going to create "vegetables", "starch" and "milk".

**Step 8| Add the relationships in the model**

We will add the relationships to the models. Open up *models/index.js*, and add the following changes:

| const bookshelf = require('../bookshelf')  const Product = bookshelf.model('Product', {  tableName:'products',  category() {  return this.belongsTo('Category')  } });  const Category = bookshelf.model('Category',{  tableName: 'categories',  products() {  return this.hasMany('Product');  } })  module.exports = {Product, Category}; |
| --- |

In the added code, we use functions to represent the relationships.

The Product model has a function named *category* which returns a **belongsTo** relationship. The argument is the *name of the model* (not the table) that it belongs to. This indicates that the Product model belongs to one Category model.

The Category model has a function named *products* which returns a **hasMany** relationship The argument is the name of the model (not the table) which it is involved in the relationship.

| **IMPORTANT**  For the Bookshelf ORM to works, the **foreign** key column name must be of the following structure:  *<table\_name\_in\_singular\_form>\_id*  Take the example of adding a foreign key to the categories table. The name of the table is *categories*, so the first part of the foreign key column name should be *category* (i.e, the table name in singular form). Then we should add *\_id* to the back.  Look at more examples below:   | **Table Name** | **Foreign Key Column Names** | | --- | --- | | tags | tag\_id | | contributing\_authors | contributing\_author\_id | | media\_properties | media\_property\_id |   It is still **possible not** to follow the convention, you just need to provide the foreign key column name as the second argument of the *belongsTo* or *belongsToMany* calls etc, like below:  const Category = bookshelf.model('Category',{  tableName: 'categories',  products() {  return this.hasMany('Product', 'category\_id');  } }) |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |

**Step 9| Allow the user to select the category**

Now when we try to create a new product, we get an error message -- because all products now must be given a category. Hence, we must change the form so that the user can select the category of the new product. Make the following changes to the *createProductForm* function in *forms/index.js*

| const createProductForm = (categories) => {  return forms.create({  'name': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  'cost': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  },  'validators':[validators.integer()]  }),  'description': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  'category\_id': fields.string({  label:'Category',  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  },  widget: widgets.select(),  choices: categories  })  }) }; |
| --- |

And we also need to update the *routes/products.js* file to pass all the existing categories to the form:

| router.post('/create', async (req, res) => {  const allCategories = await Category.fetchAll().map((category) => {  return [category.get('id'), category.get('name')];  })   const productForm = createProductForm(allCategories);  // … snipped …  }) |
| --- |

With this update, we will be able to allow the user to select the category which their product belongs to.

**Step 10| Process adding a new Product with Category**

To process the adding of a new product with category, we must:

1. Create a product form with all the possible categories
2. Set the selected category in the *product* instance

| router.post('/create', async (req, res) => {  // 1. Read in all the categories  const allCategories = await Category.fetchAll().map((category) => {  return [category.get('id'), category.get('name')];  })   const productForm = createProductForm(allCategories);  productForm.handle(req, {  'success': async (form) => {  // 2. Save data from form into the new product instance  const product = new Product(form.data);  await product.save();  res.redirect('/products');  },  'error': async (form) => {  res.render('products/create', {  'form': form.toHTML(bootstrapField)  })  }  }) }) |
| --- |

For step 2, we use a shortcut. Instead of manually setting each field, we pass all the data in the form to the product via the constructor. For this to work, the name of fields in the form **must match** the name of all columns in the table.

**Step 11| Set the original category when updating a product**

Now choose to update a product; you will see that the category has not been set properly and will always default to the first one. To solve this problem, we need to set the initial value fo the *category\_id* field of the form:

| router.get('/:product\_id/update', async (req, res) => {  // retrieve the product  const productId = req.params.product\_id  const product = await Product.where({  'id': productId  }).fetch({  require: true  });   // fetch all the categories  const allCategories = await Category.fetchAll().map((category)=>{  return [category.get('id'), category.get('name')];  })   const productForm = createProductForm(allCategories);   // fill in the existing values  productForm.fields.name.value = product.get('name');  productForm.fields.cost.value = product.get('cost');  productForm.fields.description.value = product.get('description');  productForm.fields.category\_id.value = product.get('category\_id');   res.render('products/update', {  'form': productForm.toHTML(bootstrapField),  'product': product  })  }) |
| --- |

First, the lines highlighted in yellow fetch all the categories in the system and use that to populate the forms.

Second, the green line sets the form's category\_id field value to be the same as the category\_id from the product. When the form is displayed, the correct category will be selected by default.

**Step 12| Update the *update* route**

We also need to fill the form with all the existing categories when processing the update form. Make the following changes:

| router.post('/:product\_id/update', async (req, res) => {  // fetch all the categories  const allCategories = await Category.fetchAll().map((category)=>{  return [category.get('id'), category.get('name')];  })   // fetch the product that we want to update  const product = await Product.where({  'id': req.params.product\_id  }).fetch({  required: true  });   // process the form  const productForm = createProductForm(allCategories);  productForm.handle(req, {  'success': async (form) => {  product.set(form.data);  product.save();  res.redirect('/products');  },  'error': async (form) => {  res.render('products/update', {  'form': form.toHTML(bootstrapField)  })  }  }) }) |
| --- |

Note that there is almost no change to the code at all, except that when we create the form, we pass in all the possible categories. The line in green will automatically update the row for as **provided that the fields' names in the form matches the columns' names in the table.**

**Step 13| Display the category of the product in the listing**

We are now able to select a category for a product when we create a new product or when we update an existing one. However we can't see the category of the product right in the products listing.

To show the category, first we need to fetch all the products and load each of their *category* relationship:

| router.get('/', async (req, res) => {  let products = await Product.collection().fetch({  withRelated:['category']  });  res.render('products/index', {  'products': products.toJSON()  }) }) |
| --- |

The *withRelated* key allows us to specify the **name of the relationship** on the model to load. In this case, we want to load the *category* relationship. The name of the relationship is the name of the function that returns a relationship in the model. (Consult step 6 above).

Now let's display the category in the *views/products/index.hbs* file:

| {{#extends 'base'}}  {{#block 'content'}} <table class="table">  <thead>  <tr>  <th>ID</th>  <th>Name</th>  <th>Category</th>  <th>Cost</th>  <th>Description</th>  <th></th>  </tr>   </thead>  <tbody>  {{#each products}}  <tr>  <td>  {{this.id}}  </td>  <td>  {{this.name}}  </td>  <td>  {{this.category.name}}  </td>   <td>  {{this.cost}}  </td>  <td>  {{this.description}}  </td>  <td>  <a href="/products/{{this.id}}/update" class="btn btn-primary btn-sm">Update</a>  <a href="/products/{{this.id}}/delete" class="btn btn-danger btn-sm">Delete</a>  </td>   </tr>   {{/each}}  </tbody>   </table> {{/block}}  {{/extends}} |
| --- |

Once we have loaded a relationship, we can access its table by referring by its name as a property of the model. So in this case, to retrieve the name of a category, we use this.category.name

# 12B | Add in Foreign Keys to an Existing Table

For the previous example, we have to remove the existing rows from the *products* table. We are going to explore how we can add in a new foreign key to *products* without having to delete any products.

What we are going to do now is to add a in a **brands** table. Each product will belong to one brand. To achieve this while preserving all the existing rows, we have to do the following:

1. Create the **brands** table
2. Add in a default brand to serve as placeholder
3. Add a new column *brand\_id* in the **products** table, which is also the foreign key. We will also set its default value to the id of the default brand, which should be 1.

**Step 1| Create the brands table**

Create a new migration file:

| ./db-migrate.sh create brands |
| --- |

And define the columns as such:

| exports.up = function(db) {  return db.createTable('brands',{  "id":{  'type':'int',  'primaryKey': true,  'autoIncrement': true,  'unsigned': true  },  "name": {  "type":"string",  "length": 100,  "notNull": true,  }  }) }; |
| --- |

And run the migration with ./db-migrate.sh up

**Step 2| Add in a default brand**

Let's add in a default brand to act as placeholder.

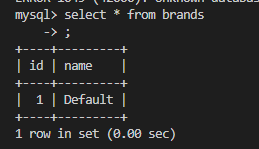
Create a new migration file to add in the new brand:

| ./db-migrate.sh create add\_default\_brand |
| --- |

And in the newly created migration file, we'll use the following function call to add in the new column:

| exports.up = function(db) {  return db.insert("brands", ['name'], ['Default']); }; |
| --- |

Now when we check the tables in the database, we would see that there is a new row in the *brands* table:



Note that the *id* of the default brand is 1.

**Step 3| Add in a new column named *brand\_id* to the products table**

We will add in a new column to the *products* table and set its default value to 1, as well as setting it as a foreign key:

First, create the migration file with: ***./db-migrate.sh create add-brand-fk-to-products***

| exports.up = function(db) {  return db.addColumn('products', 'brand\_id',{  'type':'int',  'unsigned': true,  'notNull':true,  'defaultValue': 1,  'foreignKey': {  'name':'product\_brand\_fk',  'table':'brands',  'mapping':'id',  'rules':{  'onDelete': 'cascade',  'onUpdate': 'restrict'  }  }   }); }; |
| --- |

The important difference is that when we create the *brand\_id* column, we also set its default value to **1,** which is also the **id** of the default brand we have created in the step before. This allows us to satisfy the constraint of the foreign keys **since all the existing rows in the products table already have a brand\_id column with value 1.**

# 13 | Creating Many to Many Relationships with Tags

We have explored how to create 1 to many relationships; the next relationship to explore are *many to many relationships.* For the Organic Food e-shop case, we are going to assign tags to each product. A product can be tagged as *sweet*, *processed*, *home-made* etc.

**Step 1| Create the table**

We are going to begin by creating a *tags* table.

Create a new migration with

| ./db-migrate.sh create tags |
| --- |

And then make the following changes to the *up* and *down* functions to create a *tags* table in the database:

| exports.up = function(db) {  return db.createTable('tags', {  id: { type: 'int', primaryKey:true, autoIncrement:true, unsigned:true},  name: { type: 'string', length:100},  }) };  exports.down = function(db) {  return db.dropTable('tags'); }; |
| --- |

Remember to run the migration with ./db-migrate.sh up

**Step 2| Create the pivot table**

A *pivot table* is the table between two tables which is to store their many to many relationships. To create a pivot table for a relationship that works with the Bookshelf ORM, we must make sure that its name is a combination of the tables on both sides of the relationship, arranged in alphabetical order.

So, if we want to create a pivot table to store the M:N relationship between products and tags, the table name should be *products\_tags*.

The pivot table should contain foreign keys to the two other tables in the relationship. For the Bookshelf ORM to work, use the **singular name of the table with \_id attached to the back** as the foregin key.

So in the case of the *products\_tags* table, the foreign keys should be *product\_id* and *tag\_id*.

We have figured out what to name the pivot table and its foreign keys, so let's create a migration for it.

| ./db-migrate.sh create products\_tags |
| --- |

And in the newly created migration file, add in the following code for the *exports.up* function:

| exports.up = function (db) {  return db.createTable('products\_tags', {  id: { type: 'int', primaryKey: true, autoIncrement: true },  product\_id: {  type: 'int',  notNull: true,  unsigned: true,  foreignKey: {  name: 'products\_tags\_product\_fk',  table: 'products',  rules: {  onDelete: 'CASCADE',  onUpdate: 'RESTRICT'  },  mapping: 'id'  }  },  tag\_id: {  type: 'int',  notNull: true,  unsigned:true,  foreignKey: {  name: 'products\_tags\_tag\_fk',  table: 'tags',  rules: {  onDelete: 'CASCADE',  onUpdate: 'RESTRICT'  },  mapping: 'id'  }  }  }); }; |
| --- |

After saving the file, run the migration with ./db-migrate.sh up

**Step 3| Add the relationship to the model**

Modify the *models/index.js* and add in the Tag model, and also the M:N relationship between Tag and Product:

| const bookshelf = require('../bookshelf')  const Product = bookshelf.model('Product', {  tableName:'products',  category() {  return this.belongsTo('Category')  },  tags() {  return this.belongsToMany('Tag');  }  });  . . .  const Tag = bookshelf.model('Tag',{  tableName: 'tags',  products() {  return this.belongsToMany('Product')  } })  module.exports = {Product, Category, Tag}; |
| --- |

**Step 4| Display all the possible tags when creating a product**

First, use either the MySQL CLI or the extension client to add in some tags. Examples include: processed, local, home-made, contain nuts etc.

Second, make the following changes to the route that display the form for adding new products:

| router.get('/create', async (req, res) => {   const allCategories = await Category.fetchAll().map((category)=>{  return [category.get('id'), category.get('name')];  })   const allTags = await Tag.fetchAll().map( tag => [tag.get('id'), tag.get('name')]);   const productForm = createProductForm(allCategories, allTags);    res.render('products/create', {  'form': productForm.toHTML(bootstrapField)  }) }) |
| --- |

The new line of code reads in all the tags from the table and for each tag, store their id and name in an array. All the tags are then passed to the *createProductForm* function.

Third, we modify the *createProductForm* function in the *forms/index.js* file:

| const createProductForm = (categories, tags) => {  return forms.create({  'name': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  'cost': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  },  'validators':[validators.integer()]  }),  'description': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  'category\_id': fields.string({  label: 'Category',  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  },  widget: widgets.select(),  choices: categories  }),  'tags': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  },  widget: widgets.multipleSelect(),  choices:tags  })  }) }; |
| --- |

We create a new field in the form named *tags*, and set its widget as a multiple select. We then set the *choices* key to be all the tags from the *tags* table.

Now when we try to create a new product, we should be able to see a multiple select of all the tags we have created in the database.

**Step 5| Save the relationship**

Now let's turn our attention to saving the relationship. Update the *POST products/create* route:

| router.post('/create', async (req, res) => {  const productForm = createProductForm();  productForm.handle(req, {  'success': async (form) => {  // separate out tags from the other product data  // as not to cause an error when we create  // the new product  let {tags, ...productData} = form.data;  const product = new Product(productData);  await product.save();  // save the many to many relationship  if (tags) {  await product.tags().attach(tags.split(","));  }   res.redirect('/products');  },  'error': async (form) => {  res.render('products/create', {  'form': form.toHTML(bootstrapField)  })  }  }) }) |
| --- |

The first change is that we extract out the *tags* from the form data, and assign the rest of the form keys to be in a new object named *productData.* This combines the techniques of ES6 **spread and destructuring**.

The second change is to check if the user has selected any tags; if so, then we attach the ID of those tags to the product. We have to use tags.split(',') because Caolan Forms will return the selected options from a multiple select as a comma delimited string.

**Step 6| Modify the Update Product Form**

In the same vein, let's modify the update product form so that it will display the tags of the model.

| router.get('/:product\_id/update', async (req, res) => {  // retrieve the product  const productId = req.params.product\_id  const product = await Product.where({  'id': parseInt(productId)  }).fetch({  require: true,  withRelated:['tags']  });  // fetch all the tags  const allTags = await Tag.fetchAll().map( tag => [tag.get('id'), tag.get('name')]);   // fetch all the categories  const allCategories = await Category.fetchAll().map((category)=>{  return [category.get('id'), category.get('name')];  })   const productForm = createProductForm(allCategories, allTags);   // fill in the existing values  productForm.fields.name.value = product.get('name');  productForm.fields.cost.value = product.get('cost');  productForm.fields.description.value = product.get('description');  productForm.fields.category\_id.value = product.get('category\_id');   // fill in the multi-select for the tags  let selectedTags = await product.related('tags').pluck('id');  productForm.fields.tags.value= selectedTags;   res.render('products/update', {  'form': productForm.toHTML(bootstrapField),  'product': product.toJSON()  })  }) |
| --- |

There are two main groups of changes here. The lines in orange deal with displaying all the possible tags in the form while the lines in yellow read the current tags of the product and set them as the value of the *tags* field of the form. This will set the default values of the tags multi-select to the current tags of the product.

**Step 7 | Process the updated tags**

Now we can allow the user to change the tags of the product via the form, let's process the updates. We need to **synchronize** the *tags* many to many relationship in two ways:

1. Any tag that belongs to the product that is not in the selected tags will have to be removed
2. Any tags that are in the selected tags but not in the model have to be added to the model.

Let's examine the code for doing the above two:

| router.post('/:product\_id/update', async (req, res) => {   // fetch the product that we want to update  const product = await Product.where({  'id': req.params.product\_id  }).fetch({  require: true,  withRelated:['tags']  });   // process the form  const productForm = createProductForm();  productForm.handle(req, {  'success': async (form) => {  let { tags, ...productData} = form.data;  product.set(productData);  product.save();  // update the tags    let tagIds = tags.split(',');  let existingTagIds = await product.related('tags').pluck('id');   // remove all the tags that aren't selected anymore  let toRemove = existingTagIds.filter( id => tagIds.includes(id) === false);  await product.tags().detach(toRemove);   // add in all the tags selected in the form  await product.tags().attach(tagIds);   res.redirect('/products');  },  'error': async (form) => {  res.render('products/update', {  'form': form.toHTML(bootstrapField)  })  }  }) }) |
| --- |

The line in orange essentially retrieves the selected tags and the product data from the form.

The lines in yellow do two things.

First, it goes through all the existing tags in the product and removes those not in the selected tags.

Second, it adds all the selected tags to the model.

**Step 8 | Display the tags in the product display**Now we want to display the tags associated with the product.

First, let's fetch all the products with tags:

# Parallel Lab 6

Now it's our turn to implement relationships for the Poster shop.

***1. CREATE A MEDIA PROPERTY TABLE***

Read the entire specification for this requirement before beginning work.

* Create a new table named *media\_properties*, and its corresponding model, *MediaProperty*. This describes the media property that the poster belongs to, such as "Star Wars", "BTS", "Metallica" etc. Add the following columns to the *media\_properties* table:
  + Name: the name of the media property
  + Description: a brief description of the media property
  + URL: Website of the media property
* A Poster belongs to one Media Property, a Media Property can have many posters.

***2. ADD TAGS TO POSTERS***Create a tagging system for posters, just like how we have created one for the Organic Portal's *products* model.

Part 4 | Sessions and Authentication

With the skills that we have learned, we can create a functional web application that does CRUD. However, it is lacking in features such as flash messages, user authentication and security. All those require the use of sessions, which we will implement in this part of the walkthrough.

## Preamble: Understanding sessions

In *Express*, once we have set up sessions, we are able to save data and retrieve data from it. A session is essentially a key/value pair data store (think of it as like a dictionary from Python or a JSON file).

# 14 | Add in Flash Messages

A flash message allows us to display information to the user after a redirect. To use flash messaging, we have to set up sessions in Express first.

**Step 1| Setup sessions**

Add the following dependencies:

| yarn add express-session yarn add connect-flash  yarn add session-file-store |
| --- |

And inside *index.js* add to the *requires*:

| const session = require('express-session'); const flash = require('connect-flash');  const FileStore = require('session-file-store')(session); |
| --- |

And before you import your routes, add the following:

| // set up sessions app.use(session({  store: new FileStore(),  secret: 'keyboard cat',  resave: false,  saveUninitialized: true })) |
| --- |

At this point, check your app and make sure that it works. Adding sessions have the tendencies to break your application.

**Step 3| Run nodemon and ignore session files**

We want nodemon to ignore changes to the *sessions* folder, so from now on, we run nodemon like this:

| nodemon --ignore sessions |
| --- |

**Step 4| Add sessions to .gitignore**

In your .gitginore, add in *sessions* on its own line. This will prevent the sessions folder from being tracked by your Git repository.

**Step 5| Setup Flash messages**

Add in the following line right before you import the routes:

| app.use(flash())  // Register Flash middleware app.use(function (req, res, next) {  res.locals.success\_messages = req.flash("success\_messages");  res.locals.error\_messages = req.flash("error\_messages");  next(); }); |
| --- |

*req.flash("success\_messages")* will retrieve from the session store the flashed message and delete it at the same time. Remember that *res.locals* refer to variables that are available to all *hbs* files. So essentially we are transferring the flashed message from the session to the *hbs* file which we are going to render.

Next, in your *base.hbs* file, add the following lines before {{#block 'content'}}

| {{#if success\_messages }}   {{#each success\_messages}}  <div class="alert alert-success">  {{this}}  </div>  {{/each}}   {{/if}}   {{#if error\_messages}}  {{#each error\_messages}}  <div class="alert alert-danger">  {{this}}  </div>  {{/each}}    {{/if}} |
| --- |

**Step 6 | Add flash messages**

We can now experiment with adding a new flash message. Let's do one for when the user creates a product. In *routes/products.js*, add the following code to the *POST products/create* route:

| router.post('/create', async (req, res) => {  const productForm = createProductForm();  productForm.handle(req, {  'success': async (form) => {  // separate out tags from the other product data  // as not to cause an error when we create  // the new product  let {tags, ...productData} = form.data;  const product = new Product(productData);  await product.save();  // save the many to many relationship  if (tags) {  await product.tags().attach(tags.split(","));  }  req.flash("success\_messages", `New Product ${product.get('name')} has been created`)  res.redirect('/products');  },  'error': async (form) => {  res.render('products/create', {  'form': form.toHTML(bootstrapField)  })    }  }) }) |
| --- |

We add in a flash message that adds to the *success\_messages* key in the session if the product is created successfully. Remember that a flash message is only displayed after a *redirect* or when the user goes to the next page.

# Parallel Lab 7

Implement sessions and flash messaging for the Posters application.

# 15 | User Signup

We are going to create a simple user authentication system using sessions and Express.

A session is a temporary store of data that exists between different requests and is only destroyed in one of two cases: when it expires after some time (set by you), or when you manually destroy the session. A user can destroy their own session by clearing their cookies in their browser.

The flow of user login will be:

1. The user attempts to log in with email and password
2. If the email and password is valid, we will store in the user's session that the login has been successfully
3. When the user attempts to access a restricted page that only logged-in users can access, we will check the session to see if they have logged in.

To begin, we will first create a user registration page. This will reuse the concepts from Part 1 and Part 2, so we will go through the steps here briefly.

**Step 1 | Create user migration and user model**

Create a migration to add a user:

| ./db-migrate.sh create users |
| --- |

Modify the code for the *exports.up* and *exports.down* function

| exports.up = function(db) {  return db.createTable('users',{  id: { type: 'int', primaryKey:true, autoIncrement:true},  username: { type: 'string', length:100},  email: {type: 'string', length:320},  password: {type: 'string', length:80}  }) };  exports.down = function(db) {  return db.dropTable('users'); }; |
| --- |

Run the migration:

| ./db-migrate.sh up |
| --- |

Create the model in *model/index.js* and be sure to export it:

| const User = bookshelf.model('User',{  tableName: 'users' })  module.exports = {Product, Category, Tag, User}; |
| --- |

**Step 2| Create the form for register**

Let's create a form for the registration. In *forms/index.js*, add in:

| const createRegistrationForm = () => {  return forms.create({  'username': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  'email': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  'password': fields.password({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  'confirm\_password': fields.password({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  },  validators: [validators.matchField('password')]  })  }) } |
| --- |

Remember to export out the form:

| module.exports = { createProductForm, createRegistrationForm, bootstrapField }; |
| --- |

**Step 3| Create the route to display the form**

In the *routes* folder, create a new file named *users.js*

Inside newly file, add in the following code:

| const express = require("express"); const router = express.Router();  // import in the User model const { User } = require('../models');  const { createRegistrationForm, bootstrapField } = require('../forms');  router.get('/register', (req,res)=>{  // display the registration form  const registerForm = createRegistrationForm();  res.render('users/register', {  'form': registerForm.toHTML(bootstrapField)  }) })  module.exports = router; |
| --- |

Note how we import in *createRegistrationForm*, and the *User* model.

In the *get /register* route, we create an instance of the form and render it with the *views/users/register.hbs* file. This folder and file does not exists yet -- create them, and add the following code to *views/users/register.hbs:*

| {{#extends 'base'}}  {{#block 'content'}} <h1>Register</h1> <form method="POST">  {{{form}}}  <input type="submit" value="Register" class="btn btn-primary"/>  {{/block}} </form>  {{/extends}} |
| --- |

Now test the route; you should be able to see the signup form.

**Step 4| Process the registration**

Add the following to *routes/users.js*:

| router.post('/register', (req, res) => {  const registerForm = createRegistrationForm();  registerForm.handle(req, {  success: async (form) => {  const user = new User({  'username': form.data.username,  'password': form.data.password,  'email': form.data.email  });  await user.save();  req.flash("success\_messages", "User signed up successfully!");  res.redirect('/users/login')  },  'error': (form) => {  res.render('users/register', {  'form': form.toHTML(bootstrapField)  })  }  }) }) |
| --- |

We create a User from the form, using the same methods we create products.

**Step 5| Create a temporary login page**

We redirect the user to a */users/login* after they have registered, which does not exist yet. Add a new route to *routes/users.js*

| router.get('/login', (req,res)=>{  res.render('users/login') }) |
| --- |

And then the following *views/users/login.hbs* file:

| {{#extends 'base'}}  {{#block 'content'}} <h1>Login</h1> <p>Please login with your account below!</p> {{/block}}  {{/extends}} |
| --- |

We will flesh out the login in the next section. Test that you can register as a new user and will be redirected to the login page when done.

# 16 | User Login

Now that we can sign up a new user, let's explore how we do authentication. The flow is as below:

1. When the user submits the login form, we will check if the email and password combination is correct.
2. If the email and password are correct, we store in the user's session the *ID* of the currently logged in user.
3. Whenever the user accesses a route which needs authentication, we will check if the user has logged in.

**Step 1 | Create the login form**

In *forms/index.js*, we will create a new form. Name it as *createLoginForm*, and its code are as below:

| const createLoginForm = () => {  return forms.create({  'email': fields.string({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  'password': fields.password({  required: true,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  }) } |
| --- |

Be sure to export out the form:

| module.exports = { createProductForm,   createRegistrationForm,  createLoginForm,   bootstrapField }; |
| --- |

**Step 2| Create the route to display the form**

Import in *createLoginForm* at the top of your *routes/users.js:*

| const { createRegistrationForm,   createLoginForm,  bootstrapField } = require('../forms'); |
| --- |

Add to *routes/user.js:*

| router.get('/login', (req,res)=>{  const loginForm = createLoginForm();  res.render('users/login',{  'form': loginForm.toHTML(bootstrapField)  }) }) |
| --- |

Create a *login.hbs* at the *views/users/* folder:

| {{#extends 'base'}}  {{#block 'content'}} <h1>Login</h1> <p>Please login with your account below!</p>  <form method="POST"> {{{form}}} <input type="submit" class="btn btn-primary" value="Login"/> </form> {{/block}}  {{/extends}} |
| --- |

**Step 3| Create the route to process the login**

This is where we actually implement the login system.

Add the following code to *routes/users.js* file:

| router.post('/login', async (req, res) => {  const loginForm = createLoginForm();  loginForm.handle(req, {  'success': async (form) => {  // process the login   // ...find the user by email and password  let user = await User.where({  'email': form.data.email  }).fetch({  require:false}  );   if (!user) {  req.flash("error\_messages", "Sorry, the authentication details you provided does not work.")  res.redirect('/users/login');  } else {  // check if the password matches  if (user.get('password') === form.data.password) {  // add to the session that login succeed   // store the user details  req.session.user = {  id: user.get('id'),  username: user.get('username'),  email: user.get('email')  }  req.flash("success\_messages", "Welcome back, " + user.get('username'));  res.redirect('/users/profile');  } else {  req.flash("error\_messages", "Sorry, the authentication details you provided does not work.")  res.redirect('/users/login')  }  }  }, 'error': (form) => {  req.flash("error\_messages", "There are some problems logging you in. Please fill in the form again")  res.render('users/login', {  'form': form.toHTML(bootstrapField)  })  }  }) }) |
| --- |

The first block of highlighted code will attempt to fetch a User model instance by email. Following which, we check if any user has been retrieved -- if the variable is *null*, then it means there is no user with that email. In that case we will just redirect back to the login page with an error message.

If the user is successfully retrieved, we proceed to check if the password matches. In case that it does, we store the user information in the session, and redirects the user to the */users/profile* route.

**Step 4| Add a *users/profile* route**

This route displays the details of a logged in user; we use this to test if the user authentication is working.

Add to your *routes/users.js* file:

| router.get('/profile', (req, res) => {  const user = req.session.user;  if (!user) {  req.flash('error\_messages', 'You do not have permission to view this page');  res.redirect('/users/login');  } else {  res.render('users/profile',{  'user': user  })  }  }) |
| --- |

The line in yellow is important: it retrieves the currently logged in user from the session. If there is none, we will redirect to the login page with an error message.

Create the following *profile.hbs* at the *views/users/* folder:

| {{#extends 'base'}}  {{#block 'content'}} <h1>User Profile</h1> <ul>  <li>Username: {{user.username}}</li>  <li>Email:{{user.email}}</li> </ul>  {{/block}}  {{/extends}} |
| --- |

**NOTE:** By default, all sessions are lost when the server restarts!

**Step 5| Add in a log out**

Let's create a route for logging out. Add the following code to *routes/users.js*

| router.get('/logout', (req, res) => {  req.session.user = null;  req.flash('success\_messages', "Goodbye");  res.redirect('/users/login'); }) |
| --- |

Now if you wish to logout, just go to the URL */users/logout*

# 17| Hashing Passwords

We have a rudimentary user authentication going; however we can improve on the security aspect. Right now, the password is stored in *plaintext* -- that is, it's readable by humans. This is a major security issue which we will fix.

**Step 1 | Create a function that generates a hashed password**

We will be using Nodejs built-in *crypto* module to create the hashed version of the user's password.

Add the following highlighted code, right near the top of your *routes/users.js*

| const express = require("express"); const router = express.Router(); const crypto = require('crypto');  const getHashedPassword = (password) => {  const sha256 = crypto.createHash('sha256');  const hash = sha256.update(password).digest('base64');  return hash; } |
| --- |

**Step 2| Save a hashed version of the user's password**

Now when we register a user, we hash the password before saving to the database:

| router.post('/register', (req, res) => {  const registerForm = createRegistrationForm();  registerForm.handle(req, {  success: async (form) => {  const user = new User({  'username': form.data.username,  'password': getHashedPassword(form.data.password),  'email': form.data.email  });  await user.save();  req.flash("success\_messages", "User signed up successfully!");  res.redirect('/users/login')  },  'error': (form) => {  res.render('users/register', {  'form': form.toHTML(bootstrapField)  })  }  }) }) |
| --- |

**Step 3| When logging in, compare the hashed passwords**

Let's also change the code where the user logins. Right now, when the user provides their password, it's in *plaintext* (that is not hashed or encrypted). If we compare that with the hashed password, the comparison will fail. So we have to hash the password from the form and then compare that with the one in the database.

| router.post('/login', async (req, res) => {  const loginForm = createLoginForm();  loginForm.handle(req, {  'success': async (form) => {  // snipped ...  if (!user) {  req.flash("error\_messages", "Sorry, the authentication details you provided does not work.")  res.redirect('/users/login');  } else {  // check if the password matches  if (user.get('password') === getHashedPassword(form.data.password)) {  // add to the session that login succeed   // store the user details  req.session.user = {  id: user.get('id'),  username: user.get('username'),  email: user.get('email')  }  req.flash("success\_messages", "Welcome back, " + user.get('username'));  res.redirect('/users/profile');  } else {  // ...snipped  }) |
| --- |

Now all new users will have their passwords hashed before saving to the database, and are still able to login.

# 18| Using middleware to protect routes

Right now you can protect a route from public access by using the following code snippet:

| if (req.session.user) {  // if reaches here means the user has logged in } else {  // user is not logged in  req.flash('error\_messages', 'You must be logged in to view the required page')  res.redirect('/users/login') } |
| --- |

However, we will need to repeat the above code for **all** routes that we want to protect. This is tiresome and prone to errors. What we can do is to create a **middleware.**

A middleware happens before the route is accessed, so if there is any code we want to run before the route is actually executed, we place it in a middleware, and assign that middleware to the route.

In this exercise, we will create a middleware that will redirect non logged in users to the login page.

**Step 1 | Create a new module named *middlewares***

The folder should be in the same folder as *packages.json*

**Step 2 | Create the middleware inside *index.js***

| const checkIfAuthenticated = (req, res, next) => {  if (req.session.user) {  next()  } else {  req.flash("error\_messages", "You need to sign in to access this page");  res.redirect('/users/login');  } }  module.exports = {  checkIfAuthenticated } |
| --- |

A middleware looks exactly like the function that handles a request -- it's a function that takes in the *request, response* and a *next* function.

The *next* function is provided by Express and we call it when we want Express to move on to the next middleware[[4]](#footnote-3). If there is no more middleware left, then the route will be executed.

Also, remember to export out the middleware function when we are done.

**Step 3| Add the middleware to routes we want to protect**

Let's protect the two *products/create* route -- we only allow those who have logged in to create new products.

First, import the *checkIfAuthenticated* middleware:

| // import in the CheckIfAuthenticated middleware const { checkIfAuthenticated } = require('../middlewares'); |
| --- |

Second, place the middleware as the second argument:

| router.get('/create', checkIfAuthenticated, async (req, res) => {  //… snipped  })  router.post('/create', checkIfAuthenticated, async (req, res) => {  // … snipped }) |
| --- |

Now before the route is accessed, the *checkIfAuthenticated* middleware will be executed.

**Hint:** If you want to chain multiple middlewares, you can use an array, or just specify as many middlewares you want before the final function.

**Step 4| Test the *products/create* route**

Now attempt to create a new product; you will be prompted to log in.

# 19 | Using global middleware to share session data

We are now able to detect if a user has logged in when coding the routes, but we will also like to display the username and email of the user in the different *.hbs* files. However, we don't want to go through the tedious process of manually passing to every hbs file the user information, like below:

| res.render('somefile.hbs', {  'user': req.session.user } |
| --- |

For this kind of problems, we can use a *global* middleware to inject the logged in user data to all hbs files.

**Step 1| Add a global middleware to /index.js**

Inside the root *index.js* (that is, in the same folder as *package.json)*, add the following code:

| // Share the user data with hbs files app.use(function(req,res,next){  res.locals.user = req.session.user;  next(); }) |
| --- |

This is an example of a *global* middleware -- it is not attached to any routes, so it applies to **all** routes.

**Step 2| Update *views/layouts/base.hbs***

Let's display a navigation bar that will display a navigation bar with the user's username if they are logged in.

Add the following code right at the top of <section class="container">:

| <nav class="navbar navbar-expand-lg navbar-light bg-light">  <a class="navbar-brand" href="#">Navbar w/ text</a>  <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#navbarText"  aria-controls="navbarText" aria-expanded="false" aria-label="Toggle navigation">  <span class="navbar-toggler-icon"></span>  </button>  <div class="collapse navbar-collapse" id="navbarText">  <ul class="navbar-nav mr-auto">  <li class="nav-item active">  <a class="nav-link" href="#">Home <span class="sr-only">(current)</span></a>  </li>  <li class="nav-item">  <a class="nav-link" href="#">Features</a>  </li>  <li class="nav-item">  <a class="nav-link" href="#">Pricing</a>  </li>  </ul>  <div>  {{#if user}}  Hi, {{user.username}}   <a href="/users/logout" class="ml-3 btn btn-primary btn-sm">Logout</a>   {{else}}  Hi, stranger  <a href="/users/login" class="ml-3 btn btn-primary btn-sm">Login</a>  {{/if}}  </div>  </div>  </nav> |
| --- |

The code highlighted in yellow tests if there is a user logged in. If there is, we display the user's name and a logout button. Otherwise, we display a button that allows the user to sign login.

# Parallel Lab 8

Implement a user registration and authentication using the techniques described above.

# 20| Preventing CSRF attacks

CSRF is short for **cross site request forgery**. It prevents bad actors from reusing your forms to fool others that they are on your website, and to prevent users from being tricked into performing actions which they don't intend to.

* Watch a video on CSRF and its dangers: <https://youtu.be/vRBihr41JTo>

We need to install a new external module to prevent CSRF attacks.

**Step 1| Install *csurf***

The *csurf* module allows us to generate CSRF tokens. A CSRF token is embedded into each form and is unique each time the user submits the form. A token that has been used cannot be reused. This ensures that no hackers or malicious actors are retrying to spoof our forms.

| yarn add csurf |
| --- |

**Step 2| Enable csurf**

Let's setup *csurf* in our *index.js*

Import in the *csrf* module:

| const csrf = require('csurf') |
| --- |

Add in the following code to enable *csrf* for all routes (before the *main* function)

| // enable CSRF app.use(csrf()); |
| --- |

**Step 3| Inject the CSRF token into HBS files**

We need to share the CSRF token with the HBS files. We will create a new global middleware which will be applied to all routes. Add this to your *index.js* file:

| // Share CSRF with hbs files app.use(function(req,res,next){  res.locals.csrfToken = req.csrfToken();  next(); }) |
| --- |

Then for the forms, add in the CSRF token as such (the example below is for *views/register.hbs*)

| {{#extends 'base'}}  {{#block 'content'}} <h1>Register</h1> <form method="POST">  <input type="hidden" name="\_csrf" value="{{csrfToken}}"/>  {{{form}}}  <input type="submit" value="Register" class="btn btn-primary"/>  {{/block}} </form>  {{/extends}} |
| --- |

**IMPORTANT:** You will have to do this with each form before they will work.

**Step 4 | Handle CSRF error**

Right now, the entire app will display an error message if the form has expired. Let's show an error message instead. Add the following middleware **immediately** after the app.use(csrf()).

| app.use(function (err, req, res, next) {  if (err && err.code == "EBADCSRFTOKEN") {  req.flash('error\_messages', 'The form has expired. Please try again');  res.redirect('back');  } else {  next()  } }); |
| --- |

# Parallel Lab 9

Add CSRF protection to all your forms in the Poster application

PART 5 | ADVANCED FEATURES

# 21| UPLOADING IMAGES

A common requirement for web-based applications is to upload images. However uploading images is often difficult and is often harder when using a hosting service, such as Heroku. For this purpose we are going to use Cloudinary to handle images uploading.

**Step 1| Sign up for an account at Cloudinary**

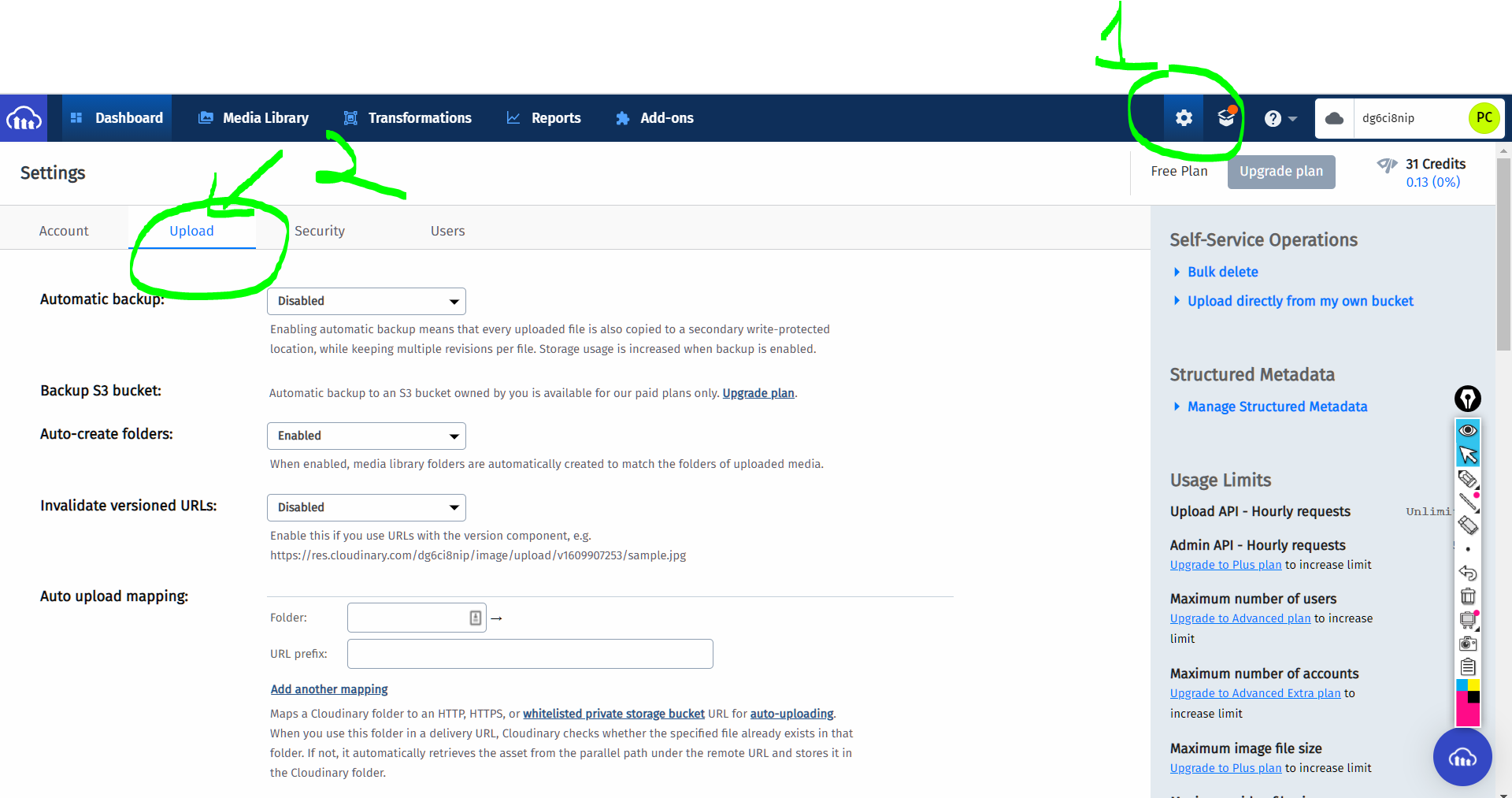
Go to <https://www.cloudinary.com> and sign up for an account.

**Step 2| Install dependencies**

| yarn add cloudinary |
| --- |

**Step 3| Create a upload preset**

In your Cloudinary account, go to **Settings** (the gear at the upper right corner), then the **Upload** tab.



After which, click **Add New Upload Preset**, and name it as *uploads.* Leave the other settings alone and choose to add the new preset. (If you ever change the preset name, or use a new preset, remember to update the .env file too).

Think of an upload preset like policies for uploading, such as whether authentication is needed, which folder the files will be uploaded to etc.

**Step 4| Add Cloudinary Configuration**

Log into your Cloudinary account, and add your cloudinary name, API Key and API Secret to your .env file:

| CLOUDINARY\_NAME=***your cloudinary name*** CLOUDINARY\_API\_KEY=***your api key*** CLOUDINARY\_API\_SECRET=***your api secret***  CLOUDINARY\_UPLOAD\_PRESET=***uploads*** |
| --- |

**Step 5| Add a new column named *image\_url*** **to the products table**

Create a new migration file with

| ./db-migrate.sh create add\_image\_url\_to\_products |
| --- |

And add the following to *exports.up*:

| exports.up = function(db) {  return db.addColumn('products', 'image\_url', {  type:'string',  length:255 });; }; |
| --- |

Finally, run the migration with ./db-migrate.sh up

**Step 6| Add a JavaScript block to *base.hbs***

We want to be able to inject JavaScript just before the closing <body> tag in *base.hbs*. Update your ***base.hbs***file as below

| <html> ...   <!-- Optional JavaScript -->  <!-- jQuery first, then Popper.js, then Bootstrap JS -->  <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"  integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFDMVNA/GpGFF93hXpG5KkN"  crossorigin="anonymous"></script>  <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.12.9/umd/popper.min.js"  integrity="sha384-ApNbgh9B+Y1QKtv3Rn7W3mgPxhU9K/ScQsAP7hUibX39j7fakFPskvXusvfa0b4Q"  crossorigin="anonymous"></script>  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/js/bootstrap.min.js"  integrity="sha384-JZR6Spejh4U02d8jOt6vLEHfe/JQGiRRSQQxSfFWpi1MquVdAyjUar5+76PVCmYl"  crossorigin="anonymous"></script>  {{#block 'js'}}  {{/block}} </body> </html> |
| --- |

**Step 7| Allow uploading of images when creating a new product**

All the preparations are ready, let's add the code to display a widget that we can use to upload the file.

First, we need to send to the hbs file the Cloudinary information:

Inside your *products/create.hbs,* add the highlighted yellow code:

| router.get('/create', async (req, res) => {   const allCategories = await Category.fetchAll().map((category) => {  return [category.get('id'), category.get('name')];  })   const allTags = await Tag.fetchAll().map(tag => [tag.get('id'), tag.get('name')]);   const productForm = createProductForm(allCategories, allTags);   res.render('products/create', {  'form': productForm.toHTML(bootstrapField),  cloudinaryName: process.env.CLOUDINARY\_NAME,  cloudinaryApiKey: process.env.CLOUDINARY\_API\_KEY,  cloudinaryPreset: process.env.CLOUDINARY\_UPLOAD\_PRESET  }) }) |
| --- |

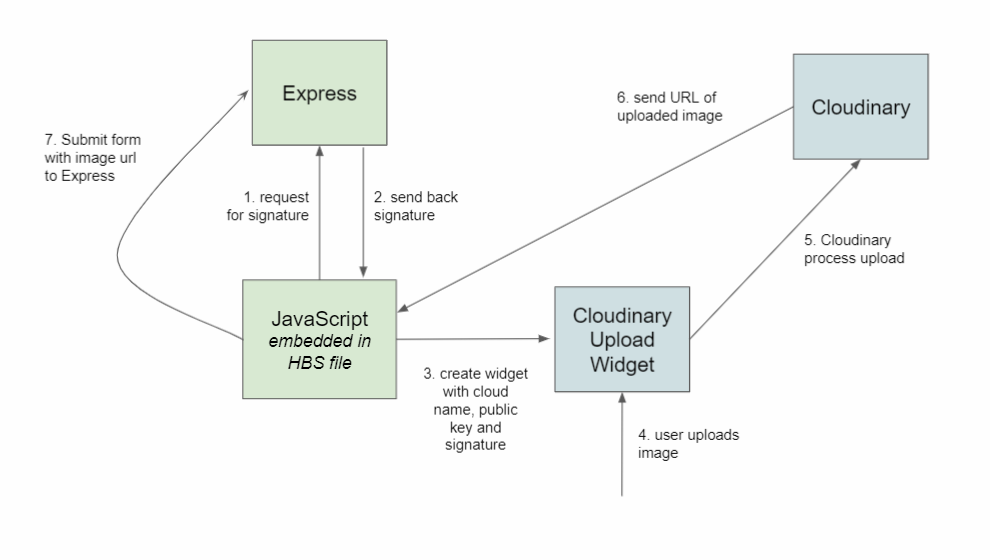
This will allow our *create.hbs* file to have access to the various Cloudinary settings.

**Step 8| Add a hidden form field to the product form**

We also update *createProductForm* function inside *forms/index.js* to add in a hidden field named *image\_url* (it's deliberate that we make sure the field matches the *image\_url* column inside the *products* table)

| const createProductForm = (categories, tags) => {  return forms.create({    'name': fields.string({  // … snipped  }),  'image\_url':fields.string({  widget: widgets.hidden()  })  }) }; |
| --- |

**INTERLUDE: Process of uploading to Cloudinary**



**Step 9| Generate a Cloudinary Signature**

A cloudinary signature is like CSRF for Cloudinary. It prevents others from stealing our API Key to perform unauthorized uploads to our account.

Create a new *cloudinary.js* file inside the *routes* folder, and add in the following code

| const express = require('express'); const router = express.Router();  const cloudinary = require('cloudinary') cloudinary.config({  'api\_key': process.env.CLOUDINARY\_API\_KEY,  'api\_secret': process.env.CLOUDINARY\_SECRET })   router.get('/sign', async (req,res)=>{  // retrieve the parameters we need to send to cloudinary  const params\_to\_sign = JSON.parse(req.query.params\_to\_sign);   // retrieve our cloudinary api secret from the environment  const apiSecret = process.env.CLOUDINARY\_API\_SECRET   // get the signature (aka CSRF)  const signature = cloudinary.utils.api\_sign\_request(params\_to\_sign, apiSecret);    res.send(signature); })  module.exports = router; |
| --- |

Remember to setup the new route file in our project level */index.js*:

| const cloudinaryRoutes = require('./routes/cloudinary.js') ... async function main() {  // if the URL begins exactly with a forward slash  // use the landingRoutes  app.use('/', landingRoutes);  app.use('/products', productRoutes);  app.use('/users', userRoutes);  app.use('/cloudinary', cloudinaryRoutes); } |
| --- |

**Step 10| Add in the JavaScript to upload files**

We need to make some major changes to the *products/create.hbs* file. The updated file is below, and changes are highlighted in yellow:

| {{#extends 'base'}}  {{#block 'content'}} <h1>Create New Product</h1> <form method="POST">  <input type="hidden" name="\_csrf" value="{{csrfToken}}"/>  {{{form}}}  <div>  <a href="#" class="btn btn-primary" id="upload\_widget">Upload</a>  <img src="" style="display:none" id="uploaded\_image"/>  </div>  <input type="submit" value="Add Product" class="btn btn-primary"/>  {{/block}} </form> {{/extends}}  {{#block 'js'}} <!-- initialise cloudinary –-> <script src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.21.1/axios.min.js" integrity="sha512-bZS47S7sPOxkjU/4Bt0zrhEtWx0y0CRkhEp8IckzK+ltifIIE9EMIMTuT/mEzoIMewUINruDBIR/jJnbguonqQ==" crossorigin="anonymous"></script>  <script src="https://widget.cloudinary.com/v2.0/global/all.js" type="text/javascript"></script>  <!-- get signature --> <script> function generateSignature(callback,params\_to\_sign){  axios.get('/cloudinary/sign',{  params:{  params\_to\_sign  }  }).then(function(response){  callback(response.data);  }) }  // create the upload widget  var myWidget = cloudinary.createUploadWidget({  cloudName: '{{cloudinaryName}}',  apiKey: '{{cloudinaryApiKey}}',  uploadPreset: '{{cloudinaryPreset}}',  uploadSignature: generateSignature  }, (error, result) => {  if (!error && result && result.event === "success") {  console.log('Done! Here is the image info: ', result.info);  // hide the upload widget   document.querySelector('#upload\_widget').style.display="none";   // display the image  document.querySelector('#id\_image\_url').value = result.info.url;  document.querySelector('#uploaded\_image').src = result.info.url;  document.querySelector('#uploaded\_image').style.display = 'inline';  }  } )  <!-- add event listener to initalise the widget --> document.getElementById("upload\_widget").addEventListener("click", function(){  myWidget.open();  }, false); </script>  {{/block}} |
| --- |

First, we need to include the *Cloudinary Widget Script*. The widget allows us to display a dialog box for users to drag and drop pictures to upload

Second, we create a function to request a signature from our Express app (it's the route that we create in step 2, above).

Third, we initialise the widget with our cloudinary settings. The second argument of the function is callback which is called by Cloudinary once the upload is successful. When the upload is successful, we hide the upload button and display the uploaded image. We also save the URL of the uploaded image into the *image\_url*, which is a hidden form field.

**Step 11| Updating the image**

Right now we can attach an image when we create a product; the next task is to allow the user to update an image when editing a product.

Note the following changes (in yellow) to the route *get/:product\_id/update*:

| router.get('/:product\_id/update', async (req, res) => {  // retrieve the product  const productId = req.params.product\_id  const product = await Product.where({  'id': parseInt(productId)  }).fetch({  require: true,  withRelated: ['tags', 'category']  });   const allTags = await Tag.fetchAll().map(tag => [tag.get('id'), tag.get('name')]);   // fetch all the categories  const allCategories = await Category.fetchAll().map((category) => {  return [category.get('id'), category.get('name')];  })   const productForm = createProductForm(allCategories, allTags);   // fill in the existing values  productForm.fields.name.value = product.get('name');  productForm.fields.cost.value = product.get('cost');  productForm.fields.description.value = product.get('description');  productForm.fields.category\_id.value = product.get('category\_id');  // 1 - set the image url in the product form  productForm.fields.image\_url.value = product.get('image\_url');   // fill in the multi-select for the tags  let selectedTags = await product.related('tags').pluck('id');  productForm.fields.tags.value = selectedTags;   res.render('products/update', {  'form': productForm.toHTML(bootstrapField),  'product': product.toJSON(),  // 2 - send to the HBS file the cloudinary information  cloudinaryName: process.env.CLOUDINARY\_NAME,  cloudinaryApiKey: process.env.CLOUDINARY\_API\_KEY,  cloudinaryPreset: process.env.CLOUDINARY\_UPLOAD\_PRESET  })  }) |
| --- |

The first new line we fill in the *image\_url* field of the form with the existing image url, if any.

The second set of new lines we are sending to the hbs file our Cloudinary information, which we will need to upload images.

**Step 12| Update the *views/products/update.hbs* file**

We are going to display the upload image widget, with some modifications -- we will show the existing product's image, if it exists:

| {{#extends 'base'}}  {{#block 'content'}} <h1>Update Product: {{product.name}}</h1> <form method="POST">  <input type="hidden" name="\_csrf" value="{{csrfToken}}"/>  {{{form}}}  <!-- 1. shows the image if exists -->  <div>   {{#if product.image\_url}}  <img src="{{product.image\_url}}" id="uploaded\_image" />  {{else}}  <img src="" style="display:none" id="uploaded\_image" />  {{/if}}  </div>  <div>  <a href="#" class="btn btn-primary" id="upload\_widget">Upload</a>  </div>  <input type="submit" value="Update Product" class="btn btn-primary" /> </form> {{/block}} {{/extends}}  {{#block js}}  <script src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.21.1/axios.min.js"  integrity="sha512-bZS47S7sPOxkjU/4Bt0zrhEtWx0y0CRkhEp8IckzK+ltifIIE9EMIMTuT/mEzoIMewUINruDBIR/jJnbguonqQ=="  crossorigin="anonymous"></script> <script src="https://widget.cloudinary.com/v2.0/global/all.js" type="text/javascript"></script> <script>  function generateSignature(callback, params\_to\_sign) {  axios.get('/cloudinary/sign', {  params: {  params\_to\_sign  }  }).then(function (response) {  callback(response.data);  })  }   var myWidget = cloudinary.createUploadWidget({  cloudName: '{{cloudinaryName}}',  apiKey: '{{cloudinaryApiKey}}',  uploadPreset: '{{cloudinaryPreset}}',  uploadSignature: generateSignature  }, (error, result) => {  if (!error && result && result.event === "success") {  console.log('Done! Here is the image info: ', result.info);  // hide the upload widget   document.querySelector('#upload\_widget').style.display = "none";   // display the image  document.querySelector('#id\_image\_url').value = result.info.url;  document.querySelector('#uploaded\_image').src = result.info.url;  document.querySelector('#uploaded\_image').style.display = 'inline';  }  }  )   document.getElementById("upload\_widget").addEventListener("click", function () {  myWidget.open();  }, false); </script>  {{/block}} |
| --- |

We changed how we display the *Upload* button and the image itself. If the product's *image\_url* is set, we will display the image, or else we won't.

## Conclusion

We have examined how to set up image uploads using Cloudinary. There are many other services that allow us to upload images.

# For Further Reading

* Original instructions for the upload widget: <https://cloudinary.com/documentation/upload_widget>
* Additional settings for Cloudinary widget
  + https://cloudinary.com/documentation/upload\_widget\_reference#widget\_parameters
* You may consider using UploadCare. Check out the relevant documentation below:
  + <https://uploadcare.com/docs/>
  + <https://uploadcare.com/docs/security/secure-uploads/>

# Parallel Lab 10

Add the functionality to add images to your poster.

# 22 | SEARCH ENGINE

Let's implement a search engine for our application. To do so, we have to build dynamic queries based on the user's input.

**Step 1| Create a search form**

Let's create a new form for searching inside *forms/index.js*

| const createSearchForm = (categories, tags) => {  return forms.create({  'name': fields.string({  required: false,  errorAfterField: true,  cssClasses: {  label: ['form-label']  }  }),  'min\_cost': fields.string({  required: false,  errorAfterField: true,  cssClasses: {  label: ['form-label']  },  'validators': [validators.integer()]  }),  'max\_cost': fields.string({  required: false,  errorAfterField: true,  cssClasses: {  label: ['form-label']  },  'validators': [validators.integer()]  }),  'category\_id': fields.string({  label: 'Category',  required: false,  errorAfterField: true,  cssClasses: {  label: ['form-label']  },  widget: widgets.select(),  choices: categories  }),  'tags': fields.string({  required:false,  errorAfterField: true,  cssClasses: {  label: ['form-label']  },  widget: widgets.multipleSelect(),  choices: tags  }),  }) } |
| --- |

This is a clone of the create product form, however we set *require* to false for all the fields (as searching by those fields as optional).

Be sure to export the form.

**Step 2 | Add the form to *get /products/* route**

Comment out the old *get /* route in *routes/products.js*.

At *routes/products.js*, add the form to the route like so:

| router.get('/', async (req, res) => {  // 1. get all the categories  const allCategories = await Category.fetchAll().map((category) => {  return [category.get('id'), category.get('name')];  })  allCategories.unshift([0, '----']);  // 2. Get all the tags  const allTags = await Tag.fetchAll().map(tag => [tag.get('id'), tag.get('name')]);     // 3. Create search form   let searchForm = createSearchForm(allCategories, allTags);   searchForm.handle(req, {  'empty': async (form) => {  },  'error': async (form) => {  },  'success': async (form) => {    }  }) }) |
| --- |

Right now the route is not ready yet. But let's take this slowly.

The first set of highlighted code simply fetches all the categories. This is similar to when we create a new product, however there is a twist. We manually add in a new category, '----', which simply represents no category selected. The value for this option is 0.

The second set of highlighted code fetches all the tags.

The last set of code creates the search form and handles it when it is submitted.

We also display in the .hbs file (*views/products/index.hbs*)

| <form method="GET">  {{{form}}}  <input type="submit" class="mb-3 mt-3 btn btn-primary" value="Search"/> </form> |
| --- |

Do note that we are processing the form via the HTTP GET method, so there's no need for a CSRF token.

**Step 3| Display all results if the form is empty**

If the search form is not filled in, we simply display all the results. Add in the following highlighted code:

| router.get('/', async (req, res) => {     const allCategories = await Category.fetchAll().map((category) => {  return [category.get('id'), category.get('name')];  })  allCategories.unshift([0, '----']);    const allTags = await Tag.fetchAll().map(tag => [tag.get('id'), tag.get('name')]);   let searchForm = createSearchForm(allCategories, allTags);  let q = Product.collection();   searchForm.handle(req, {  'empty': async (form) => {  let products = await q.fetch({  withRelated: ['category']  })  res.render('products/index', {  'products': products.toJSON(),  'form': form.toHTML(bootstrapField)  })  },  'error': async (form) => {  },  'success': async (form) => {  }  }) }) |
| --- |

The first highlighted line, let q = Product.collection(), creates a *query builder* that simply means "SELECT \* from products". We can continue to add clauses to a query builder until we execute it with a *fetch* function call.

The second set of highlighted lines basically fetches all the products, and renders the hbs file and passes it the *products* results and the form. Those lines execute when all the fields in the form are blank.

Now test this and we will see all the products.

**Step 4| Display all results if there are errors**

We will duplicate the same code for the *error* case as well:

| router.get('/', async (req, res) => {  const allCategories = await Category.fetchAll().map((category) => {  return [category.get('id'), category.get('name')];  })  allCategories.unshift([0, '----']);    const allTags = await Tag.fetchAll().map(tag => [tag.get('id'), tag.get('name')]);   let searchForm = createSearchForm(allCategories, allTags);  let q = Product.collection();   searchForm.handle(req, {  'empty': async (form) => {  let products = await q.fetch({  withRelated: ['category']  })  res.render('products/index', {  'products': products.toJSON(),  'form': form.toHTML(bootstrapField)  })  },  'error': async (form) => {  let products = await q.fetch({  withRelated: ['category']  })  res.render('products/index', {  'products': products.toJSON(),  'form': form.toHTML(bootstrapField)  })   },  'success': async (form) => {  }  }) }) |
| --- |

**Step 5| Perform the search if any of the search form fields are filled in**

Now we will add more clauses to the query depending on what the user has filled in.

All the code below here goes in the *success* case:

| searchForm.handle(req, {  'empty': async (form) => {  // snipped   },  'error': async (form) => {  // snipped  },  'success': async (form) => {  if (form.data.name) {  q = q.where('name', 'like', '%' + req.query.name + '%')  }   if (form.data.category\_id && form.data.category\_id != "0"  ) {  q = q.query('join', 'categories', 'category\_id', 'categories.id')  .where('categories.name', 'like', '%' + req.query.category + '%')  }   if (form.data.min\_cost) {  q = q.where('cost', '>=', req.query.min\_cost)  }   if (form.data.max\_cost) {  q = q.where('cost', '<=', req.query.max\_cost);  }   if (form.data.tags) {  q.query('join', 'products\_tags', 'products.id', 'product\_id')  .where('tag\_id', 'in', form.data.tags.split(','))  }    let products = await q.fetch({  withRelated: ['category']  })  res.render('products/index', {  'products': products.toJSON(),  'form': form.toHTML(bootstrapField)  })  }  }) |
| --- |

The added code checks which field has been added in, and adds more *where* clauses based on that. For instance, if the user fills in the *name* field, we will add ***AND WHERE name LIKE "%<name>%"*** to the base query.

Note that to search by categories, we have to perform a join with the categories table first.

# Parallel Lab 11

Create a search engine for your Posters application, making it possible to search by the name, media properties, tags, price and size.

* For price, allow the user to specify a range to search by (e.g, search for posters costing between 50 to 100, or lesser than 200)
* For size, allow the user to specify the maximum or minimum width/height

# Further Reading

* Knex commands: https://knexjs.org/

# 

# 23 | PLACING DATABASE INFO AND KEYS IN .ENV FILES

Let's place our database credentials into login files for easier updates and better security.

**Step 1| Add database credentials to .env file**

Add your database server's host, user name, password and the name of your database to your .env file:

| CLOUDINARY\_NAME=dg6ci8nip CLOUDINARY\_API\_KEY=239117997147176 CLOUDINARY\_API\_SECRET=TaiI5E4pu2YM2v0uAG9k10y\_qd4 CLOUDINARY\_UPLOAD\_PRESET=uploads DB\_DRIVER=mysql DB\_USER=foo DB\_PASSWORD=bar DB\_DATABASE=organic DB\_HOST=localhost |
| --- |

**Step 2| Change *database.json* to use environment variables**

Update your *database.json*, which is used by db-migrate:

| {  "dev": {  "driver": {"ENV" :"DB\_DRIVER"},  "user": {"ENV": "DB\_USER" },  "password": {"ENV":"DB\_PASSWORD"},  "database": {"ENV":"DB\_DATABASE"},  "host": {"ENV":"DB\_HOST"}  } } |
| --- |

**Step 3| Update your *bookshelf/index.js* file**

| // Setting up the database connection const knex = require('knex')({  client: process.env.DB\_DRIVER,  connection: {  user: process.env.DB\_USER,  password: process.env.DB\_PASSWORD,  database: process.env.DB\_DATABASE,  host: process.env.DB\_HOST  } }) const bookshelf = require('bookshelf')(knex)  module.exports = bookshelf; |
| --- |

**Step 4| Test your database**

After making the changes, test if your database still works. You can temporarily change the DB\_DATABASE variable in your .env to a different, new database and see if you can run ./db-migrate.sh up

If it works, then this means the new changes are valid. Switch back to your original database afterwards.

**Step 5| Add the secret key for sessions inside your .env file**

Now add a new environment variable in your .env file named SESSION\_SECRET\_KEY. Set it to a random string of characters (you can use <https://randomkeygen.com/> and select one key from the CodeIgniter keys).

For our example, we are going to use 'quick brown fox'

| CLOUDINARY\_NAME=dg6ci8nip CLOUDINARY\_API\_KEY=239117997147176 CLOUDINARY\_API\_SECRET=TaiI5E4pu2YM2v0uAG9k10y\_qd4 CLOUDINARY\_UPLOAD\_PRESET=uploads DB\_DRIVER=mysql DB\_USER=foo DB\_PASSWORD=bar DB\_DATABASE=organic DB\_HOST=localhost SESSION\_SECRET\_KEY=quick brown fox |
| --- |

Next, update your project-level *index.js*, and when initializing the session, we use the environment variable instead.

| app.use(session({  secret: process.env.SESSION\_SECRET\_KEY,  resave: false,  saveUninitialized: true })) |
| --- |

Now, restart your app and test that you are able to login and register new users.

## Parallel Lab 12

Move your database credentials to .env file instead of coding it within your application. Also generate a secret key for your session.

# 24 | Creating Data Access Layer

A data access layer (DAL for short), in software-engineering terms, is a collection of functions that deal with a **single domain.** A domain is a group of related user requirements. So for instance, the updating and adding of products, their categories and their tags are considered to be in one domain (product management, for example), while the requirements of adding products to shopping cart, checking out and payment belongs to the *checkout* domain.

In Express, we represent each data access layer as a class in the *dal* module.

**Step 1| Create a new *product* repos**

Let's create a new module named *dal*. First, create a new folder by the name of *dal*in the root directory, and create a file named *products.js* there. There will be other repositories in there, so we won't use *index.js*.

**Step 2 | Add commonly used functions to the *product* repository**

The purpose of a DAL is to store commonly used code that accesses the database in a centralized location, so that there will be code reuse and lesser duplication of code.

Looking at our *routes/products.js* file we see there are some code which are repeated:

* Fetching of all categories
* Fetching of all tags
* Fetching of a single product by its id

Let's add in the following code:

| // import in the Product model const { Product, Category, Tag } = require('../models');  const getAllCategories = async () => {  return await Category.fetchAll().map((category) => {  return [category.get('id'), category.get('name')];  }) }  const getAllTags = async () => {  return await Tag.fetchAll().map(tag => [tag.get('id'), tag.get('name')]); }  const getProductByID = async (productId) => {  return await Product.where({  'id': parseInt(productId)  }).fetch({  require: true,  withRelated: ['tags', 'category']  }); }  module.exports = {  getAllCategories, getAllTags, getProductByID } |
| --- |

The functions are essentially based on the code that we have already written.

**Step 3| Make use of the data access layer**

We have numerous lines in *routes/products.js* that fetch all categories and tags. Let's replace them with our DAL functions.

First, make sure to import our DAL:

| // import in the DAL const dataLayer = require('../dal/products') |
| --- |

As it is very likely that we will use all the functions from the DAL, we import all the functions into the *dataLayer* object instead of specifying them one by one like usual.

Next in the same file, we will update the *get /* route with the functions from our DAL:

| router.get('/', async (req, res) => {  const allCategories = await dataLayer.getAllCategories();  allCategories.unshift([0, '----']);   const allTags = await dataLayer.getAllTags();   let q = Product.collection();  let searchForm = createSearchForm(allCategories, allTags);  . . . }) |
| --- |

We can also update *update /:product\_id/* with a call to our DAL functions:

| router.get('/:product\_id/update', async (req, res) => {  // retrieve the product  const productId = req.params.product\_id  const product = await dataLayer.getProductByID(productId); . . . }) |
| --- |

## CONCLUSION

The role of a data layer is to simplify access to our models. When you find yourself writing the same lines of code to retrieve or to modify models, you can generalize those code into a function and place them in a data layer. This cuts down on duplication code and increases the robustness of your program.

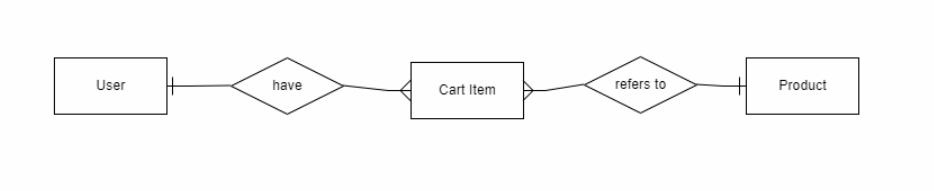
## Parallel Lab 13

Create a data layer class which has functions to store duplicated code. Name the class *PosterServices* which has the following functions:

* getAll : get all the posters
* getMediaProperties: retrieve all the media properties
* addPoster : add a new poster to the database
* findPoster : retrieve a poster by id

# 25 | SHOPPING CART

We are going to implement a shopping cart, using the database to store the cart items. A user can have many items in the cart (each of them will be referred to as a cart item) but a cart item only belongs to one user. See the ERD diagram below to see the relationship between products, users and cart items:



Each cart item represents one product in a user's shopping cart. Since a user can add the same product to the cart multiple times, a cart item will also keep track of its quantity as well.

**Note:** We are going to treat the concept of cart items as a model for easier processing. It is entirely possible to consider a cart item a pivot table between products and users, but that route is just messier.

## Creating a service layer

A service encapsulates **complex logic** which we may want to reuse. The difference between a data access layer and a service is that the latter usually encompasses *business rules/logic* while the former is usually simple access to the tables.

For example, *where* do we implement the following functionality?

* Discount (per item and for the entire shopping cart)
* Promotions
* Limiting the quantity of certain coveted products (such as customized shoes or limited t-shirts)

Programming is concerned with **how** to implement features whereas software engineering is concerned with **where** to place those code.

Here are a few places to implement the above features

* **Controller:** Those logic can definitely go into the controller, but right now our controllers are for a web application. What if we want to support an API later?
* **Model:** A *CartItem* model represents only one item in a shopping cart. We cannot place code that requires the entire shopping cart.
* **Data Access:** While it can arguably be placed there (meaning, it's up for debate), we don't usually place business logic there.

Hence the ideal will be in a service.

In this case, our cart will have the following functionalities:

* Add item to cart
* Remove item from cart
* Create Stripe payment
* Empty cart

A service layer will in turn rely on a data layer to get those tasks done. The diagram below shows an overview of the methods inside the service layer and the data layer it will use.



So when a route wishes to add an item to the cart or update an existing cart item's quantity, it will call one of the methods in the Cart Services Layer, which will in turn call the corresponding functions in the Cart Data Layer. Look at the diagram below:



1. This begins when the client sends a request to the Express server.
2. After the request reaches the route, it will be sent to the service layer
3. The service layer will call the relevant functions in the data layer
4. The data layer will access the database
5. The database will return the requested information or perform the action as required.
6. A status message will be returned from the database to the data layer, which either contains the data requested or a status code which indicates whether the request is successful or not.
7. The status message or requested data will be returned to the data layer
8. The status message or requested data will be returned to the services layer
9. The services layer will return the status messages or requested data to the route.

This can be applied to other features that we have implemented so far, such as CRUD for products and user registration.

**Step 1| Create a new migration for the table *cart\_items***

We will need a table to store all the *cart\_items* first. Let's follow the series of steps below to create the *cart\_items* table. First, begin with:

| ./db-migrate.sh create cart\_items |
| --- |

Next, we add the table creation code to the newly created migration file:

| exports.up = function (db) {  return db.createTable('cart\_items', {  id: { type: 'int', unsigned: true, primaryKey: true, autoIncrement: true },  quantity: {type: 'int', unsigned:true},  user\_id: {  type: 'int',  unsigned: true,  notNull: true,  foreignKey: {  name: 'cart\_items\_user\_fk',  table: 'users',  mapping: 'id',  rules: {  onDelete: 'CASCADE',  onUpdate: 'CASCADE'  }  }  },  product\_id: {  type: 'int',  notNull: true,  unsigned: true  foreignKey: {  name: 'cart\_items\_product\_fk',  table: 'products',  rules: {  onDelete: 'CASCADE',  onUpdate: 'RESTRICT'  },  mapping: 'id'  }  }  }); }; |
| --- |

After adding in the code, run the migration with ./db-migrate.sh up and the new cart items table will be created in our database.

Note that a row in the *CartItem* table represents one item in a user's shopping cart. Which is why the row includes a foreign key to the product and a foreign key to the user who owes the cart.

Finally, let's create a model for the *cart\_items* table (and remember to export it out)

| const CartItem = bookshelf.model('CartItem', {  tableName: 'cart\_items',  product() {  return this.belongsTo('Product')  }  })  module.exports = {Product, Category, Tag, User, CartItem}; |
| --- |

**Step 2 | Create the Cart Data Layer**

The data layer will contain all the functions that allows us to:

* Get all the cart items of a user
* Add a product to a user's shopping cart
* Remove a product from the user's shopping cart
* Change the quantity of a item in the user's shopping cart

Inside the *dal* folder, create a new file named *cart\_items.js*. We begin with a function that allows us to get the items of a user's shopping cart. The function simply takes in one argument, the id of a user, and returns all the items in the cart.

| const { CartItem } = require('../models');  const getCart = async (userId) => {  return await CartItem.collection()  .where({  'user\_id': userId  }).fetch({  require: false,  withRelated: ['product', 'product.category']  }); }  module.exports = { getCart } |
| --- |

There isn't any new concept taught here; we are simply reorganizing existing code inside a class.

We will also add in a function that will check if a specific product exists in a user's shopping cart. If it does, it will return the cart item. Add the following function, and also remember to update the exports:

| const getCartItemByUserAndProduct = async (userId, productId) => {  return await CartItem.where({  'user\_id': userId,  'product\_id': productId  }).fetch({  require: false  }); }  module.exports = { getCart, getCartItemByUserAndProduct } |
| --- |

The next function is to add an item to the cart. The function receives three arguments -- the id of the user, the id of the product and the quantity. The function takes those arguments and creates a new instance of the CartItem model before saving it. (Remember, an **instance** of a model class represents one row in its corresponding MySQL table). Remember to export the function.

| async function createCartItem(userId, productId, quantity) {   let cartItem = new CartItem({  'user\_id': userId,  'product\_id': productId,  'quantity': quantity  })  await cartItem.save();  return cartItem; }  module.exports = {getCart, getCartItemByUserAndProduct, createCartItem} |
| --- |

Likewise, we will add in a function to remove an item from cart. This takes in a user (to reflect which user we the item from) and the id of the product which we want to remove from the cart. We'll check if the cart item exists first before attempting to destroy it. Note how we reuse the *getCartItemByUserAndProduct* in the data layer to achieve that.

| async function removeFromCart(userId, productId) {  let cartItem = await getCartItemByUserAndProduct(userId, productId);  if (cartItem) {  await cartItem.destroy();  return true;  }  return false; } |
| --- |

In the same vein, let's add a function that allows us to update the quantity of a given cart item, identified by the product id and user id. We will use the *getCartItemByUserAndProduct* to get the cart item that we want to update. If it exists, we will update its quantity and then save it.

| async function updateQuantity(userId, productId, newQuantity) {  let cartItem = await getCartItemByUserAndProduct(userId, productId);  if (cartItem) {  cartItem.set('quantity', newQuantity);  cartItem.save();  return true;  }  return false; } |
| --- |

Here's the source code for the cart data layer: <https://github.com/kunxin-chor/dwad-advanced-express/blob/32-updated-shopping-cart/dal/cart_items.js>

**Step 3 | Create a Cart Service**

After creating the cart data layer, let's move on to the cart services. Create a new folder named *services* in the project folder (the folder with the *package.json* file) and create a new file named *cart\_services.js.*

Add in the following code to create the CartServices class and export it:

| const cartDataLayer = require('../dal/cart\_items')  class CartServices {  constructor(user\_id) {  this.user\_id = user\_id;  } } module.exports = CartServices; |
| --- |

**Step 4| Add to cart**

In the *CartServices* class, let's add a function to an item to the cart. This will be done by using the functions from the *cart\_items.js* file in the *dal* folder. Here we check if the product being added is already in the cart. If it already exists, we simply update the cart item by 1, otherwise we will create a new cart item for that product and user.

| async addToCart(productId, quantity) {  // check if the user has added the product to the shopping cart before  let cartItem = await cartDataLayer  .getCartItemByUserAndProduct(this.user\_id, productId);  if (cartItem) {  return await cartDataLayer  .updateQuantity(this.user\_id, productId, cartItem.get('quantity') + 1);  } else {  let newCartItem = cartDataLayer.  createCartItem(this.user\_id, productId, quantity);  return newCartItem;  }  } |
| --- |

**Step 5| Remove from cart**

Next, we add in the function to remove a cart item from the cart. There isn't much to the function -- we are simply relying on the *removeFromCart* function defined in the cart data layer.

| async remove(productId) {  return await cartDataLayer  .removeFromCart(this.user\_id, productId);  } |
| --- |

**Step 6| Update cart item quantity**

| async setQuantity(productId, quantity) {  return await cartDataLayer  .updateQuantity(this.user\_id, productId, quantity);  } |
| --- |

Again we are just reusing the function from the cart data layer.

**Step 6| Get all cart items for a user**

The final function we will add is to retrieve all the cart items for a particular user:

| async getCart() {  return await cartDataLayer.getCart(this.user\_id);  } |
| --- |

You can find all the code for the Cart Services so far: <https://github.com/kunxin-chor/dwad-advanced-express/blob/32-updated-shopping-cart/services/cart_services.js>

**Step 7| Display the content of the shopping cart**

Create a new router file named *shoppingCart.js* inside the *routes* folder, and in the following content:

| const express = require("express"); const router = express.Router();  const CartServices = require('../services/cart\_services');  router.get('/', async(req,res)=>{  let cart = new CartServices(req.session.user.id);  res.render('cart/index', {  'shoppingCart': (await cart.getCart()).toJSON()  }) })  router.get('/:product\_id/add', async (req,res)=>{  let cart = new CartServices(req.session.user.id);  await cart.addToCart(req.params.product\_id, 1);  req.flash('success\_messages', 'Yay! Successfully added to cart')  res.redirect('/products') }) |
| --- |

We make use of the *CartServices* to return the content of the shopping cart, and also to add a product to the cart.

Next, let's create the *index.hbs* file for the cart view (place it in *views/carts*)

| {{#extends 'base'}}  {{#block 'content'}} <h1>My shopping cart</h1>  <ul class="list-group">  {{#each shoppingCart}}  <li class="list-group-item my-3">  <h4>{{this.product.name}}</h3>  <h5>Category: {{this.product.category.name}}</h5>  <h5>Cost: {{this.product.cost}} </h5>  <h5>Quantity{{this.quantity}}</h5>   </li>  {{/each}} </ul>   {{/block}}  {{/extends}} |
| --- |

**Step 8| Implement an add to cart button**

Go back to *products/index.hbs* and add in the following code to display a "Add to Cart" item:

| {{#each products}}  <tr>  <td>  {{this.id}}  </td>  <td>  {{this.name}}  </td>  <td>  {{this.category.name}}  </td>  <td>  {{this.cost}}  </td>  <td>  {{this.description}}  </td>  <td>  <a href="/cart/{{this.id}}/add" class="btn btn-success btn-sm">Add to cart</a>  <a href="/products/{{this.id}}/update" class="btn btn-primary btn-sm">Update</a>  <a href="/products/{{this.id}}/delete" class="btn btn-danger btn-sm">Delete</a>  </td>  </tr>  {{/each}} |
| --- |

The line in yellow will add a button that allows us to add the current product to the cart. Now add a few products to the cart and test that the */cart* route works.

**Step 9| Display the remove from cart button**

Next, update the *shoppingCart/index.hbs* route to include the remove from cart button.

| <li class="list-group-item">  <h4>{{this.product.name}}</h3>  <h5>Category: {{this.product.category.name}}</h5>  <h5>Cost: {{this.product.cost}} </h5>  <h5>Quantity{{this.quantity}}</h5>  </form>  <div>  <a href="/cart/{{this.product.id}}/remove" class="btn btn-danger">Remove</a>  </div>   </li> |
| --- |

And update *routes/shoppingCart.js* by adding in this new function:

| router.get('/:product\_id/remove', async(req,res)=>{  let cart = new CartServices(req.session.user.id);  await cart.remove(req.params.product\_id);  req.flash("success\_messages", "Item has been removed");  res.redirect('/cart/'); }) |
| --- |

Test removing a few products from the shopping cart.

**Step 10| Test updating quantity of products in the cart**

We will begin by adding the following route in *routes/shoppingCart.js*. We simply use the *setQuantity* from the CartServices class.

| router.post('/:product\_id/quantity/update', async(req,res)=>{  let cart = new CartServices(req.session.user.id);  await cart.setQuantity(req.params.product\_id, req.body.newQuantity);  req.flash("success\_messages", "Quantity updated")  res.redirect('/cart/');  }) |
| --- |

FInally, we will also add in a form for each item to be updated in the shopping cart. Make the following changes to *views/cart/index.hbs*

| {{#extends 'base'}}  {{#block 'content'}} <h1>My shopping cart</h1>  <ul class="list-group">  {{#each shoppingCart}}  <li class="list-group-item">  <h4>{{this.product.name}}</h3>  <h5>Category: {{this.product.category.name}}</h5>  <h5>Cost: {{this.product.cost}} </h5>  <form method="POST" action="/cart/{{this.id}}/quantity/update">  <input type="hidden" value="{{../csrfToken}}" name="\_csrf"/>   <h5>  Quantity: <input type="text" name="newQuantity" value="{{this.quantity}}" style="width:30px" />   <input type="submit" value="Update Quantity" class="btn btn-success btn-sm" />  </h5>  </form>  <div>  <a href="/cart/{{this.product.id}}/remove" class="btn btn-danger">Remove</a>  </div>   </li>  {{/each}} </ul>   {{/block}}  {{/extends}} |
| --- |

Note how we specify the action of the form to be the URL for updating the quantity of a cart item, and how we have to specify the CSRF token.

## Parallel Lab 14

Add a shopping cart to the poster shop.

# 26 | STRIPE CHECKOUT

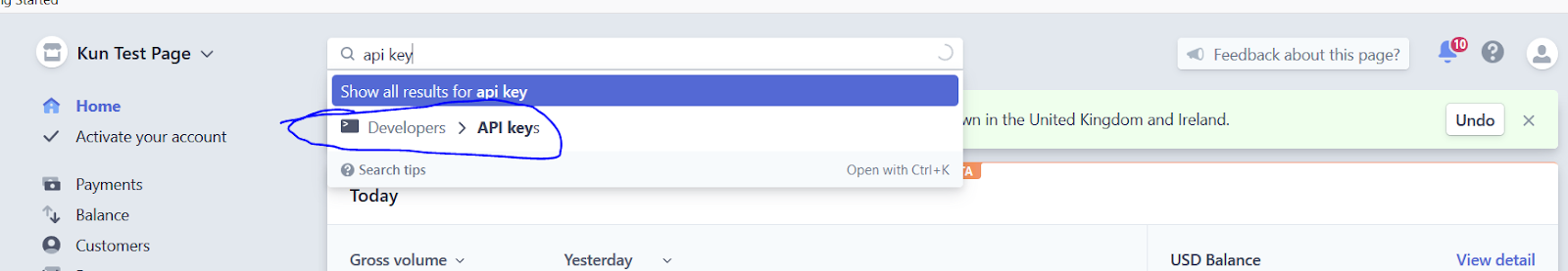
## Step 1| Register an account with Stripes

Go to <https://stripe.com/> and sign up for an account.

Be sure to set an account name for your Stripe account.

## Step 2| Obtain the keys

Sign into your Stripe Account, and inside the search box, type in "API Keys", and click on "Developers > API Keys"



## Step 3| Add the keys to your .env file

In your .env file, add the keys you have just copied from Stripes:

| STRIPE\_PUBLISHABLE\_KEY*=your publishable key here* STRIPE\_SECRET\_KEY*=your secret key here* |
| --- |

## Step 4| Install the stripes package

At the terminal, install the *stripes* package with:

| yarn add stripe |
| --- |

## Step 5| Create a *checkout.js* route

For the next step, we are going to create a *routes/checkout.js* file. Add the following code:

| const express = require('express'); const router = express.Router();  const CartServices = require('../services/cart\_services') const Stripe = require('stripe')(process.env.STRIPE\_SECRET\_KEY)  module.exports = router; |
| --- |

Note how when we require in *Stripe*, we also pass it our Stripe secret key. Make sure *process.env.STRIPE\_SECRET\_KEY* is defined if your Stripe payment doesn't work.

## Step 6| Create a route to obtain a session id

When we do checkout via Stripes, we perform the following steps:

1. Create line items (an array of all the products the user is purchasing)
2. Create a stripe payment session
3. Register the session with the stripe
4. Get the session id of the session

The session id is important because we will redirect the user to the Stripe checkout page, where they will enter their credit card information.

Add the following route to the file *routes/checkout.js*

| const Stripe = **require**('stripe')(process.env.STRIPE\_SECRET\_KEY)  router.get('/', async (req, res) => {  const cart = new CartServices(req.session.user.id);   // get all the items from the cart  let items = await cart.getCart();   // step 1 - create line items  let lineItems = [];  let meta = [];  for (let item of items) {  const lineItem = {  'name': item.related('product').get('name'),  'amount': item.related('product').get('cost'),  'quantity': item.get('quantity'),  'currency': 'SGD'  }  if (item.related('product').get('image\_url')) {  lineItem['images'] = [item.related('product').get('image\_url')]  }  lineItems.push(lineItem);  // save the quantity data along with the product id  meta.push({  'product\_id' : item.get('product\_id'),  'quantity': item.get('quantity')  })  }   // step 2 - create stripe payment  let metaData = JSON.stringify(meta);  const payment = {  payment\_method\_types: ['card'],  line\_items: lineItems,  success\_url: process.env.STRIPE\_SUCCESS\_URL + '?sessionId={CHECKOUT\_SESSION\_ID}',  cancel\_url: process.env.STRIPE\_ERROR\_URL,  metadata: {  'orders': metaData  }  }   // step 3: register the session  let stripeSession = await Stripe.checkout.sessions.create(payment)  res.render('checkout/checkout', {  'sessionId': stripeSession.id, // 4. Get the ID of the session  'publishableKey': process.env.STRIPE\_PUBLISHABLE\_KEY  })   }) |
| --- |

In step 1, we create line items. Each line item is an object with very specific keys as defined by Stripe. We cannot use our own keys, we must follow the keys as set in the Stripe documentation ([Stripe API Reference - Create a Session](https://stripe.com/docs/api/checkout/sessions/create)). Note that we also save an array of objects (into the *meta* variable), with each object storing the product the user is buying and the quantity.

Do take note that when specifying the *cost* of an item, it is in cents.

In step 2, we create the payment. We specify the line items and the met data while creating the payment.

In step 3, we send the payment to Stripes and get back a session.

In step 4, we send the session's id to a HBS file.

Before testing, we have to make add two more settings to our .env file:

| STRIPE\_SUCCESS\_URL=<domain name>/checkout/success STRIPE\_ERROR\_URL=https:<domain name>/checkout/error |
| --- |

Replace the highlighted section (in *yellow*) with the domain name of your Gitpod instance (or Heroku's app name, once deployed).

## Step 7| Create the *checkout.hbs* file

Create a new file as */views/checkout/checkout.hbs*, and add in the following code:

| <script src="https://js.stripe.com/v3/"></script>  <script type="text/javascript">  var stripe = Stripe('{{ publishableKey }}');  stripe.redirectToCheckout({  sessionId: '{{ sessionId }}'  }) </script> |
| --- |

The code does a redirect to the Stripe checkout page. We fill in the publishable key and the session id for the *redirectToCheckout* function to work.

## Step 8| Testing

At the Stripe checkout page, you can enter the fake credit card number "4242-4242-4242-4242" for a valid credit card. Enter any numbers for the CSV and a date in the future for the expiry date. Key in any email addresses and name you wish.

After doing so, you will be able to click on the button to submit payment, and it should appear on your Stripe account.

## Parallel Lab 15: Stripes Checkout

Add stripes payment to your poster shop.

# 27| BEING NOTIFIED ABOUT PURCHASES VIA WEBHOOKS

Webhooks are *reverse endpoints* in a sense; those are endpoints in our app that we provide to other platforms, such as Stripe, to call when a certain event happens. In this case, we will like to be notified when a payment is completed.

## Step 1| Include *bodyParser* as a module (might be depreciated)

We need to use *bodyParser* to extract out the raw HTTP request information. Add the following *require* statement at the top of *routers/checkout.js*

| const bodyParser = require('body-parser'); |
| --- |

Remember to *yarn add body-parser* in the terminal

## Step 2| Create the webhook

In your *routes/checkout.js,* create a new route and the following code:

| router.post('/process\_payment', bodyParser.raw({type: 'application/json'}), async (req, res) => {  let payload = req.body;  let endpointSecret = process.env.STRIPE\_ENDPOINT\_SECRET;  let sigHeader = req.headers["stripe-signature"];  let event;  try {  event = Stripe.webhooks.constructEvent(payload, sigHeader, endpointSecret);   } catch (e) {  res.send({  'error': e.message  })  console.log(e.message)  }  if (event.type == 'checkout.session.completed') {  let stripeSession = event.data.object;  console.log(stripeSession);  // process stripeSession  }  res.send({ received: true }); }) |
| --- |

The code is **called by Stripe**, not by us. It will extract out an *event* object from the HTTP request, and attempt to verify that it comes from Stripe and not other fraudulent sources. If the event is validated successfully, we then check its event. If it is a *checkout.session.completed* event, then the payment has been made. The data of the transaction is stored inside *stripeSession*. At this point we can proceed to store the session in the database, and trigger any other processes, like sending an invoice to the user.

We need one more step before this can work; we must inform Stripe about this endpoint.

## Step 3| Exclude from CSRF

In your *index.js*, replace *app.use(csurf())* with the following:

| // note: replaced app.use(csrf()) with the following: const csurfInstance = csurf(); app.use(function(req,res,next){  console.log("checking for csrf exclusion")  // exclude whatever url we want from CSRF protection  if (req.url === "/checkout/process\_payment") {  return next();  }  csurfInstance(req,res,next); }) |
| --- |

We also need to modify the global middleware where we add the token to the HBS file, because the the csrfToken function may not be available:

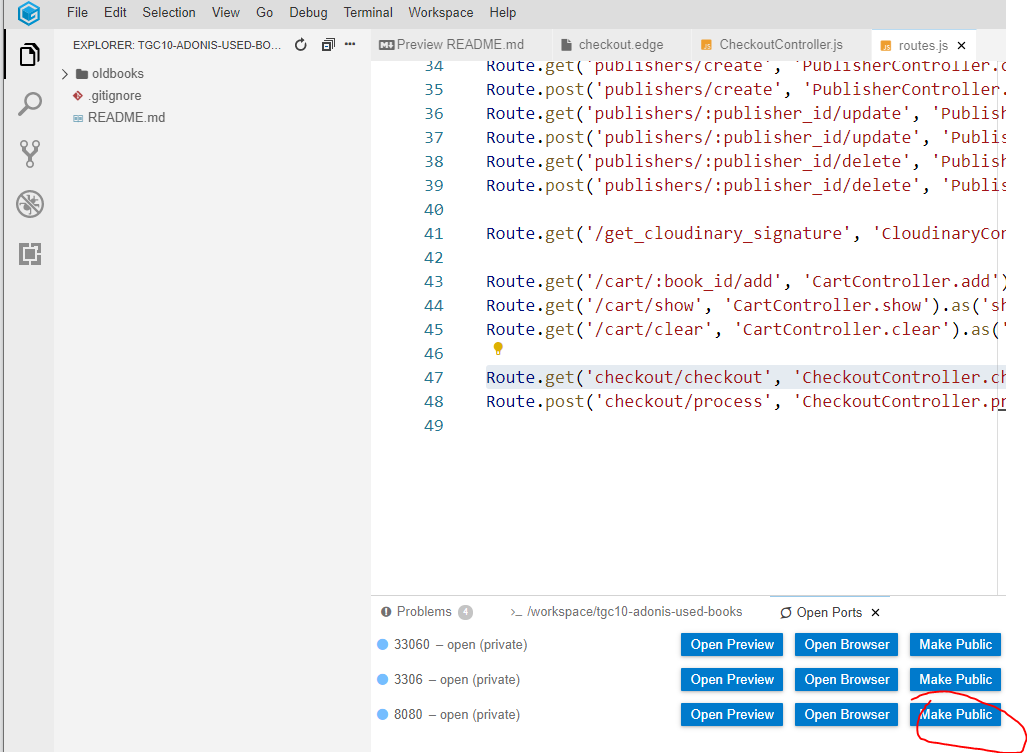
| app.use(function(req,res,next){  if (req.csrfToken) {  res.locals.csrfToken = req.csrfToken();  }    next(); }) |
| --- |

## Step 4| Create the endpoint on Stripe

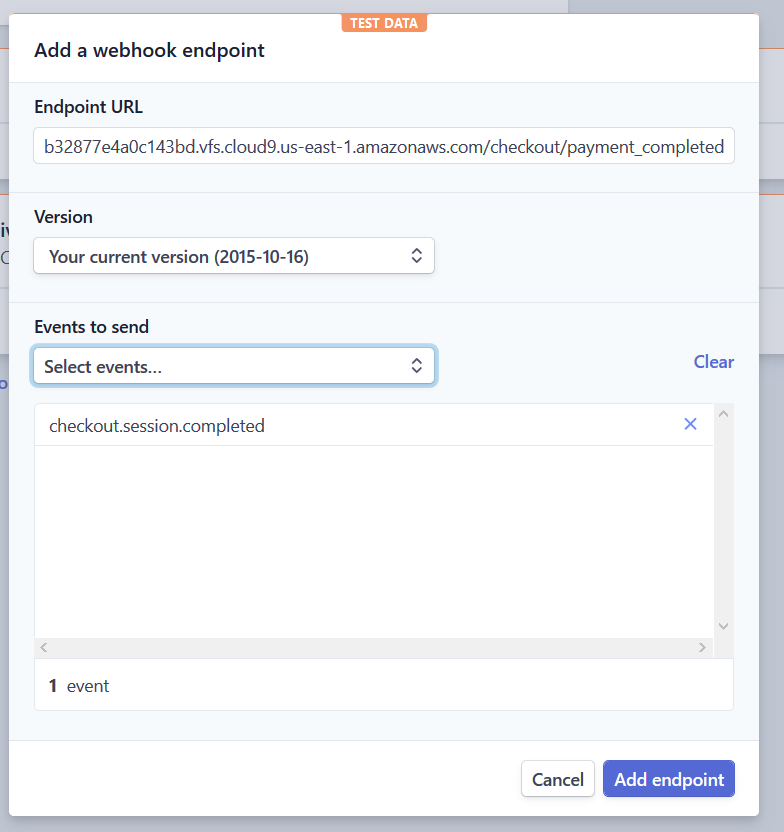
Go to your Dashboard in Stripe, and search for "webhook" and click on the "Developers > Webhooks"

Click on "New Endpoint" button

Create the endpoint URL by appending 'checkout/process\_payment' to the URL to your Gitpod instance. Also make sure that your server has been made public. (see below as to where to find the button)



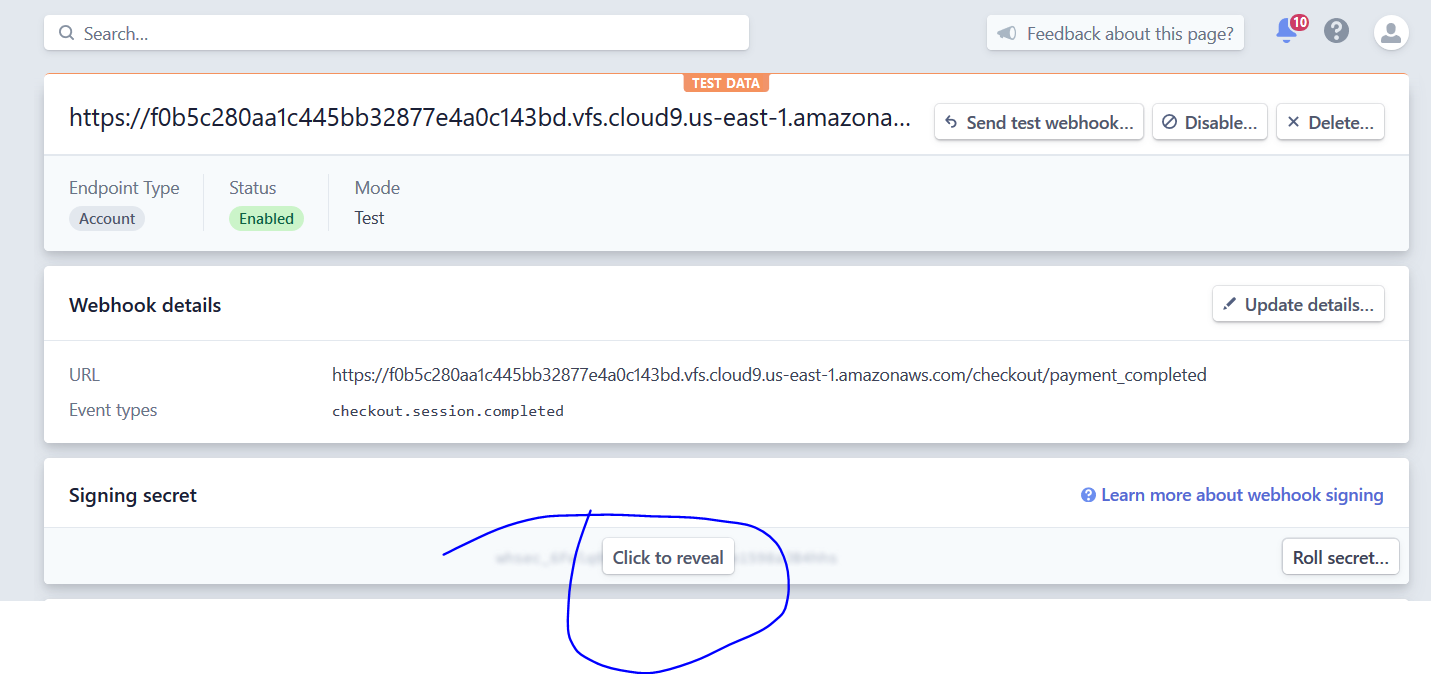
For events, type in "checkout" and select "checkout.session.completed". What you have should look similar to the screenshot below (the endpoint URL will be different from what you have)



Make sure to enable the endpoint before leaving the page.

## Step 5 | Add the endpoint secret to your .env file

First, retrieve the endpoint secret by clicking on the "Click to reveal" button (see below)



Then add it to your *.env* file:

| STRIPE\_PUBLISHABLE\_KEY=pk\_test\_cjZ0FuGOglBmUeCFc6ToHLr8 STRIPE\_SECRET\_KEY=sk\_test\_vgMxhrVz9VcFwa9SXemYtPZl STRIPE\_SUCCESS\_URL=https://3000-fuchsia-takin-f78if4p3.ws-us03.gitpod.io/checkout/success STRIPE\_ERROR\_URL=https://30000-fuchsia-takin-f78if4p3.ws-us03.gitpod.io/checkout/error STRIPE\_ENDPOINT\_SECRET=whsec\_SkIuLzJmZAqjLMNHYbLon3LjnxFZzdMM |
| --- |

Now when you test by checking out your shopping cart, you should be able to see in your console a dump of the session data.

## PARALLEL LAB 16

Implement webhooks for your Poster application.

Part 4 | API DEVELOPMENT

Very often, an Express application hosts both routes for browsers and for API endpoints. We will see how we can add endpoints for our current Express application, and how to implement JSON web tokens for user authentication over API.

# 28 | Create an API route

There are some considerations to keep in mind when creating an API route:

1. API routes are excluded from CSRF, since they don't use sessions
2. Data sent to API routes are in JSON format.

We will create an API route that will return all the products for this example.

**Step 1| Create an *api* folder to store *product.js***

All routes which will be endpoints for the API will be stored in the *api* folder. Create a new *product.js* file at */routes/api*

**Step 2| Add in the code to fetch all the products**

At *dal/product.js*, add the following function and be sure to export it:

| const getAllProducts = async () => {  return await Product.fetchAll(); }  . . . module.exports = {  getAllCategories, getAllTags, getProductByID, getAllProducts } |
| --- |

**Step 3| Return all products at *routes/api/products.js***

Add in the following route function:

| const express = require('express') const router = express.Router();  const productDataLayer = require('../../dal/products')  router.get('/', async(req,res)=>{  res.send(await productDataLayer.getAllProducts()) })  module.exports = router; |
| --- |

Note how we have to use *../../* to go up two directories to access the *dals* folder.

This code essentially just returns all the products in the database.

**Step 4| Register the route**

Let's register the newly created *api/products* route in the project level *index.js*

Add in the following highlighted code:

| const api = {  products: require('./routes/api/products') }  async function main() {  app.use('/', landingRoutes);  app.use('/products', productRoutes);  app.use('/users', userRoutes);  app.use('/cloudinary', cloudinaryRoutes);  app.use('/cart', shoppingCartRoutes);  app.use('/checkout', checkoutRoutes);  app.use('/api/products', api.products); } |
| --- |

Note how we place the *products* router as an key/value pair in the *api* object. We do so that so we can easily differentiate between the browser routes and the API routes.

**Step 5| Test**

Now when we go to */api/products* with our browser, we will be able to see all the products in JSON format.

# 29 | Sending a POST, PATCH or DESTROY request

Now let's try implementing a feature to create a product via the API. There are two considerations when we send a HTTP POST, PATCH or DESTROY request:

1. We have to exclude the route from CSRF token checks
2. We have to ensure that the request body is parsed as JSON.

To achieve the above, we need to make some alterations to our project-level index.js

**Step 1 | Disable CSRF for all urls beginning with '/api/'**

Just like how we disable CSRF for our Stripe webhook, we will do the same for any URL that begins with "/api/". Find the code that implements CSRF, and make the following changes highlighted in yellow:

| const csrfInstance = csrf(); app.use(function (req, res, next) {  // exclude /checkout/process\_payment for CSRF  if (req.url === '/checkout/process\_payment' || req.url.slice(0,5)=="/api/") {  return next()  }  csrfInstance(req, res, next) }) |
| --- |

Right now, if the request url is "/checkout/process\_payment" or anything that begins with "/api/", we won't activate the CSRF middleware.

**Step 2 | Parse req.body as JSON for API routes**

Next, at where we add in the API routes to the Express app, specify that we are going to use the *express.json* middleware:

| async function main() {  app.use('/', landingRoutes);  app.use('/products', productRoutes);  app.use('/users', userRoutes);  app.use('/cloudinary', cloudinaryRoutes);  app.use('/cart', shoppingCartRoutes);  app.use('/checkout', checkoutRoutes);  app.use('/api/products', express.json(), api.products) } |
| --- |

When we do this, all content inside *req.body* will be converted to JSON before reaching any route functions in *routes/api/products*

**Step 3| Create a route to create a new product**

In */routes/api/products.js*, add in a new route to create a new product. Add in the code below:

| const { Product } = require('../../models'); const { createProductForm } = require('../../forms');  // ...  router.post('/', async (req, res) => {  const allCategories = await productDataLayer.getAllCategories();  const allTags = await productDataLayer.getAllTags();  const productForm = createProductForm(allCategories, allTags);   productForm.handle(req, {  'success': async (form) => {   let { tags, ...productData } = form.data;  const product = new Product(productData);  await product.save();    // save the many to many relationship  if (tags) {  await product.tags().attach(tags.split(","));  }  res.send(product);  },  'error': async (form) => {  let errors = {};  for (let key in form.fields) {  if (form.fields[key].error) {  errors[key] = form.fields[key].error;  }  }  res.send(JSON.stringify(errors));  }  })  }) |
| --- |

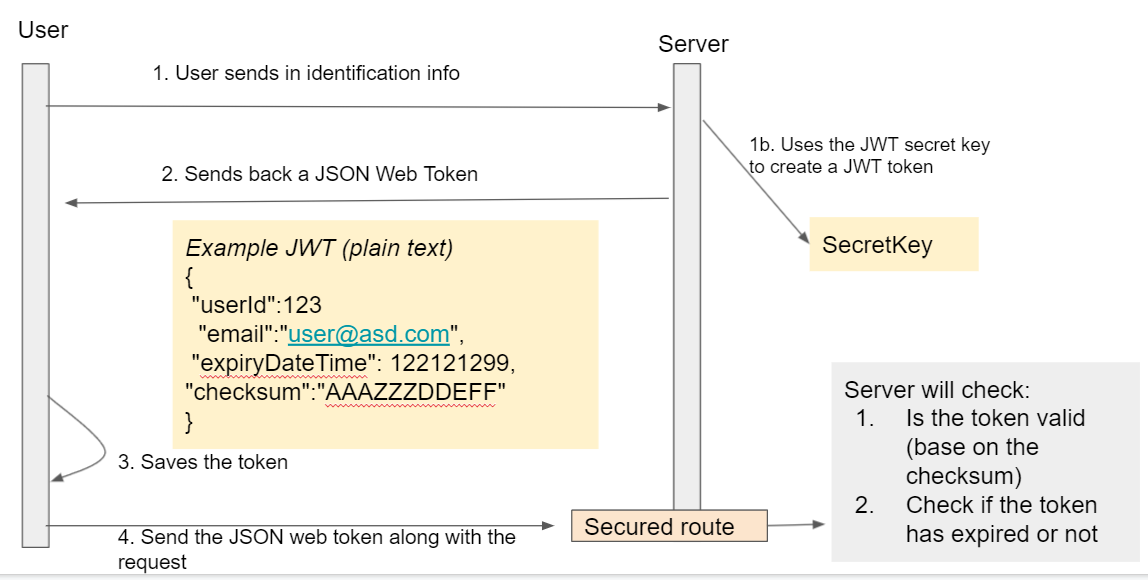
The code is very similar to the one we have written before, except for the parts where to handle the errors. We go through each field in the form and extract each error from the field, if there are any (though there should be at least one. Then we send back the errors to the client calling the API endpoint.

## CONCLUSION

We can use the same technique for processing DELETE, PATCH and PUT requests. Make sure you use the *express.json()* middleware for those routes and that they are not affected by CSRF.

# 30 | API AUTHENTICATION USING JSON WEB TOKENS

We need a method to protect our endpoints from unauthorized access. One technique for doing this is by using *JSON web tokens.*  A web token is generated using a mathematical algorithm and is sent to the client; the client has to send the token back to our API for authentication. Look at the diagram below to understand how an API client can authenticate itself with JWT:



Let's see how we can use JSON web tokens in our project.

**Step 1| Install JSON web token to your dependencies**

In the terminal, type in:

| yarn add jsonwebtoken |
| --- |

**Step 2| Add in a TOKEN\_SECRET your .env file**

In your .env file, add in a new environmental variable named *TOKEN\_SECRET.* Go to the Random Keygen website and use the 504-bit WPA key as its value. See the example below:

| TOKEN\_SECRET=|U7#ee<>dr3UuA%wZ]p;y9[5Ut>rb}Xa4Ob8FU/{43J0[,m3EIa1TqadeWQ"WXc |
| --- |

**Step 3| Create a login route for API**

Let's create a new file *users.js* in */routes/api*. We'll place inside there a function for the user to log in. Once the user has logged in, we will create a JSON web token and send it back to them.

| const express = require('express') const router = express.Router(); const crypto = require('crypto'); const jwt = require('jsonwebtoken');  const generateAccessToken = (user) => {  return jwt.sign({  'username': user.get('username'),  'id': user.get('id'),  'email': user.get('email')  }, process.env.TOKEN\_SECRET, {  expiresIn: "1h"  }); }  const getHashedPassword = (password) => {  const sha256 = crypto.createHash('sha256');  const hash = sha256.update(password).digest('base64');  return hash; }  const { User } = require('../../models');  router.post('/login', async (req, res) => {  let user = await User.where({  'email': req.body.email  }).fetch({  require: false  });   if (user && user.get('password') == getHashedPassword(req.body.password)) {  let accessToken = generateAccessToken(user);  res.send({  accessToken  })  } else {  res.send({  'error':'Wrong email or password'  })  } })  module.exports = router; |
| --- |

First, we use the require function to import in *jwt*. We also create a function to generate a web token. We set its expiry in an hour.

Second, when the user sends their email and password to the *api/users/login* route, we get the user object by their email. If the user object exists in the database, then we check if their hashed password matches. If so, then we generate an access token and send it back.

**Step 4| Add authentication middleware**

Just like we have a middleware to protect routes for session-based authentication, we also have to create one that uses JWT.

Add the following code to *middlewares/index.js*

| const jwt = require('jsonwebtoken'); // ..snipped const checkIfAuthenticatedJWT = (req, res, next) => {  const authHeader = req.headers.authorization;   if (authHeader) {  const token = authHeader.split(' ')[1];   jwt.verify(token, process.env.TOKEN\_SECRET, (err, user) => {  if (err) {  return res.sendStatus(403);  }   req.user = user;  next();  });  } else {  res.sendStatus(401);  } }; |
| --- |

**Step 5| Create a *profile* route in */routes/api/users.js***

In the */routes/api.users.js,* add in a function for the */profile* route:

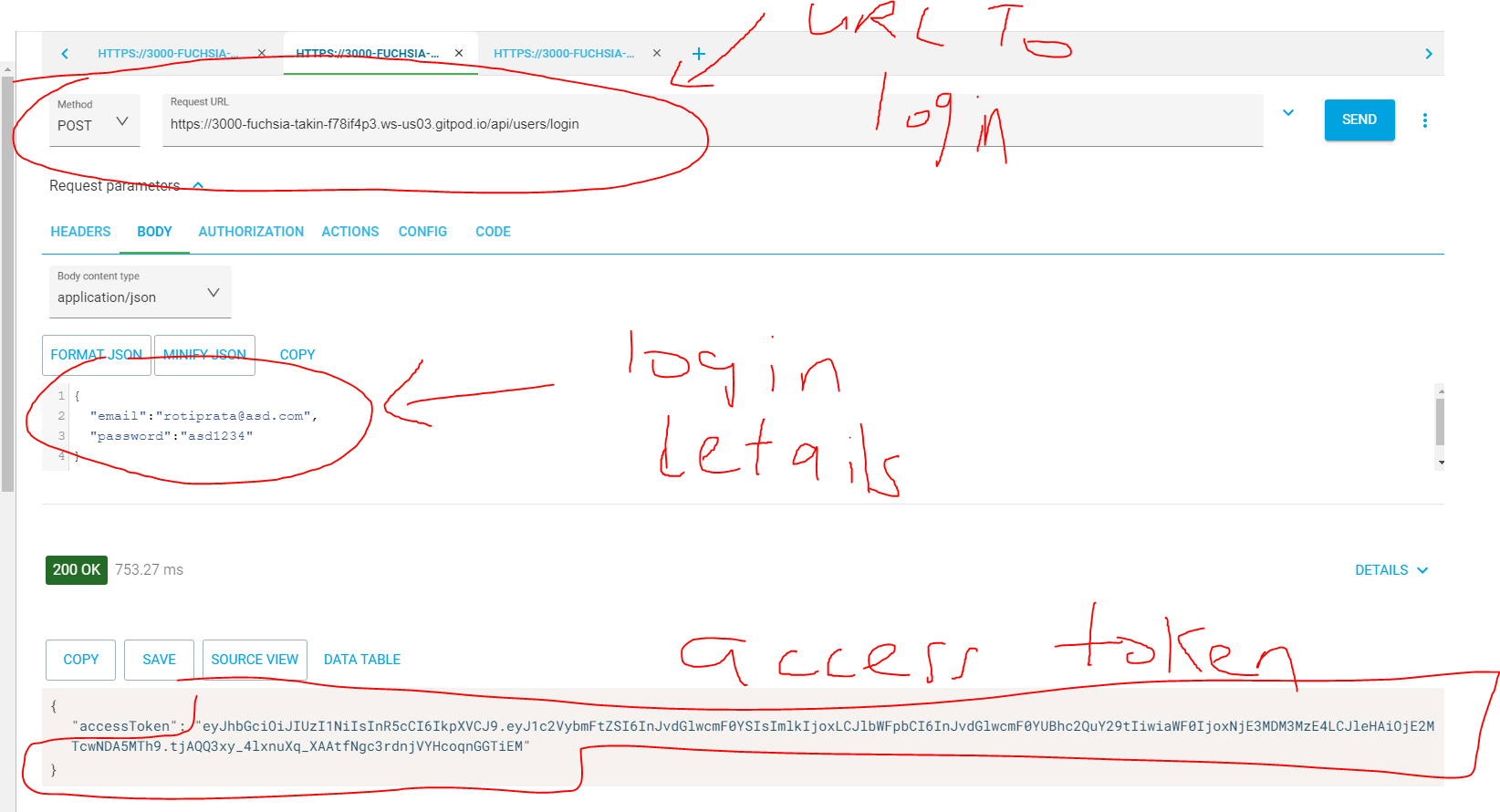
| const {checkIfAuthenticatedJWT} = require('../../middlewares') // . . . snipped router.get('/profile', checkIfAuthenticatedJWT, async(req,res)=>{  const user = req.user;  res.send(user); }) |
| --- |

Note how we add in the *checkIfAuthenticatedJWT* middleware as part of the route.

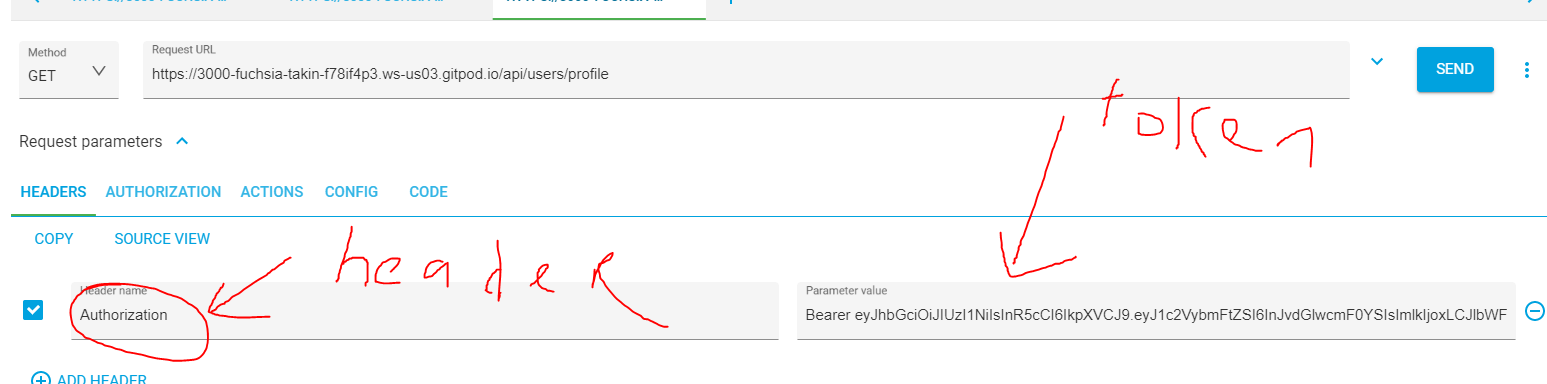
**Step 6|Test the login and profile routes**

Now if you test going to this route, we will have to go to *api/users/login* to get the JWT token, and embed it in the request to *api/users/profile.*

1. Be sure to make your Gitpod instance public
2. Using ARC, make a **POST** request to *api/users/login*



1. Copy the access token (see the screenshot above)
2. Use ARC to access the profile route and be sure to set the Authorization header to *Bearer <AccessToken>*



1. You should be able to see the user information in the response.

**Step 7| Decoding the token in the frontend**

The JWT token contains information about the currently logged in user. We can use the *jwt-decode* library (available from CDN or as an import for React) to decode the token. Look at the [sample](https://replit.com/@ckunx/BountifulBoringSearchservice#script.js) here to see how to decode the token for your frontend.

**HINT:** For your frontend (Vanilla Javascript or React, consider storing the access token in localstorage (or session storage). When the user logs out, remove the access token from storage. There is no way to destroy a token on the Express app directly, but see below for a solution.

## PARALLEL LAB 17

Create an API for logging in and protecting routes using Middleware

# 31 | REFRESH TOKENS

There is an issue with using JWT that has been hotly debated online -- how to *expire* a token after a time of inactivity, or if the user logs out. As it stands, our token expires automatically in an hour's time, meaning that the user will have to login again after an hour. Some applications omits the *exipresIn* parameter when creating token, allowing the token to last indefinitely.

One strategy is to employ the use of a **refresh token**, which lasts longer (say a week or so), while the **access token** expires frequently (say for about 15 minutes). On your frontend (be site vanilla JavaScript or React), when you detect that the access token is close to expiry, you send a request using the *refresh token* to get a new token. If that fails to get a new token, then it means the user must log in again.

When the user logs out, we add the *refresh token* to a black list so that it cannot be reused.

## Step 1 | Create a black list of tokens

This is a standalone table with no relationships. It is just to store a list of used JWT refresh tokens.

Create the following migration with *./db-migrate.sh create token\_black\_lists*

| exports.up = function(db) {  return db.createTable('blacklisted\_tokens', {  id: {type:'bigint', primaryKey:true, autoIncrement:true},  token: { type: 'string', length:5000},  date\_created : {type:'date'}  }) }; |
| --- |

We set the length to 5000 characters because encrypted tokens can be rather long.

Next, we also create a model to represent a blacklisted token. Add the model to your *models/index.js*

| const BlacklistedToken = bookshelf.model('BlacklistedToken',{  tableName: 'blacklisted\_tokens' })  module.exports = {Product, Category, Tag, User, CartItem, BlacklistedToken}; |
| --- |

## Step 2 | Add a REFRESH\_TOKEN\_SECRET to your .env file

We will use this secret key to create refresh tokens. Add the following to your .env file.

| REFRESH\_TOKEN\_SECRET=<token secret> |
| --- |

Of course, replace <token secret> with one you get from a random key generator (256 or 512 bit will do).

## **Step 3 | Update the *generateAccessToken*** **function in *routes/api/users.js***

Let generalize the function so that it will take in any token secret and a variable expiry duration:

| const generateAccessToken = (user, secret, expiresIn) => {  return jwt.sign({  'username': user.get('username'),  'id': user.get('id'),  'email': user.get('email')  }, secret, {  'expiresIn': expiresIn  }); } |
| --- |

## Step 4 | When logging in, create a refresh token and send it back too

In your *routes/api/users.js* file, modify the *POST /login* route so that it will use the new function and return a refresh token as well:

| router.post('/login', async (req, res) => {  let user = await User.where({  'email': req.body.email  }).fetch({  require: false  });   if (user && user.get('password') == getHashedPassword(req.body.password)) {  const userObject = {  'username': user.get('user'),  'email': user.get('email'),  'id': user.get('id')  }  let accessToken = generateAccessToken(userObject, process.env.TOKEN\_SECRET, '15m');  let refreshToken = generateAccessToken(userObject, process.env.REFRESH\_TOKEN\_SECRET, '7d');  res.send({  accessToken, refreshToken  })  } else {  res.send({  'error': 'Wrong email or password'  })  } }) |
| --- |

Note how we specify a different token and expiry duration for the access token and refresh token. We also send both tokens back to the client.

The client-side must store both the access tokens and the refresh tokens.

## Step 5 | Allow the client to get a new access token

As our access token expires every 15 minutes, the client must send a request to the server to get a new access token before it expires. Let's create a route for the user to send the refresh token.

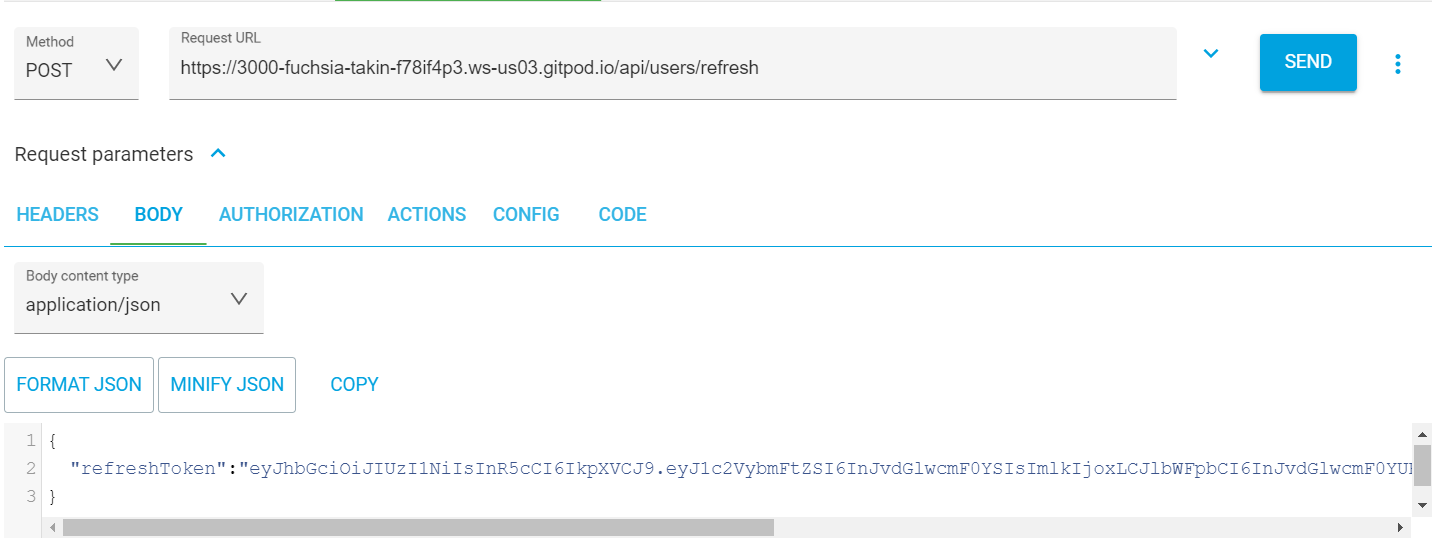
In *routes/api/users.js*, add in the following route:

| router.post('/refresh', async(req,res)=>{  let refreshToken = req.body.refreshToken;  if (!refreshToken) {  res.sendStatus(401);  }  jwt.verify(refreshToken, process.env.REFRESH\_TOKEN\_SECRET, (err, user)=>{  if (err) {  return res.sendStatus(403);  }   let accessToken = generateAccessToken(user, process.env.TOKEN\_SECRET, '15m');  res.send({  accessToken  });  }) }) |
| --- |

## Step 6 | Test getting a refresh token

Now try out the test route from 30.6 above to get both an access token and a refresh token.

In ARC, using the POST request, visit the route and specify a valid refresh token in the body. See below for example:



**HINT:** In your client, you have to set a timer that uses axios to send a POST request to */api/users/refresh* to get a new access token every 10 minutes or so (because the token expires in 15 minutes). If the attempt to get a new access token fails, then it means the user's session has expired).

## Step 7 | Adding a logout

Now we want to add a logout functionality to the API. As stated in the previous lab, there is no way to destroy (or to invalidate) a created token. However, we can add the user's refresh token to a black list, and prevent the user from getting a new access token.

**Note:** This means if a user's access token is accidentally leaked, the attacker has 15 minutes of unsecured access to the user's account. For high security applications, don't use JWT or set the expiry to 5 minutes.

First, let's add a logout route to */routes/api/users.js*

| router.post('/logout', async (req, res) => {  let refreshToken = req.body.refreshToken;  if (!refreshToken) {  res.sendStatus(401);  } else {  jwt.verify(refreshToken, process.env.REFRESH\_TOKEN\_SECRET,async (err, user) => {  if (err) {  return res.sendStatus(403);  }   const token = new BlacklistedToken();  token.set('token', refreshToken);  token.set('date\_created', new Date()); // use current date  await token.save();  res.send({  'message': 'logged out'  })  })   }  }) |
| --- |

Second, we need to check the black list when a user attempts to refresh a token. If the refresh token is already in the black list, we will block the attempt. Let's modify the *api/users/refresh* route:



| router.post('/refresh', async (req, res) => {  let refreshToken = req.body.refreshToken;  if (!refreshToken) {  res.sendStatus(401);  }   // check if the refresh token has been black listed  let blacklistedToken = await BlacklistedToken.where({  'token': refreshToken  }).fetch({  require: false  })    // if the refresh token has already been blacklisted  if (blacklistedToken) {   res.status(401);  return res.send('The refresh token has already expired')  }   jwt.verify(refreshToken, process.env.REFRESH\_TOKEN\_SECRET, (err, user) => {  if (err) {  return res.sendStatus(403);  }   let accessToken = generateAccessToken(user, process.env.TOKEN\_SECRET, '15m');  res.send({  accessToken  });  }) }) |
| --- |

Here we use the BlacklistedToken model to check if the given token has already been blacklisted. If it is, we will block the refresh attempt.

# 33 | Enabling CORS

If we want to use *axios* or any other HTTP request library to access our API, we need to enable cross origin resource sharing (or CORS for short).

**Step 1| Install CORS as dependency**

In the terminal at the project directory, type in:

| yarn add cors |
| --- |

**Step 2| Require in *cors***

At the top of your project level's *index.js*, add the following line

| const cors = require('cors') |
| --- |

**Step 3| Add cors to the middleware**

Use the *cors* middleware. Make sure you do it before you enable sessions!

| app.use(cors()); |
| --- |

PART 5 | DEPLOYMENT

# 34 | DEPLOY TO HEROKU

Heroku is a *service as a platform* which allows us to deploy web-based applications with Github, and take away the hassle of sourcing and setting up a virtual server. Let's see how we can quickly get our NodeJS server up on Heroku quickly.

**Note:**

If you do not have a project ready, you can fork <https://github.com/kunxin-chor/tgc11-advanced-express-deploy> and deploy the project. However first you must set up the database again

**Step 1| Log into Heroku**

At the terminal, log in to heroku with:

| heroku login - i |
| --- |

Enter your username and password.

**Step 2| Create the Heroku App**

Once you have logged in, create a new Heroku app with the following commands at the terminal:

| heroku create <app-name> |
| --- |

Replace <app-name> with a name of your choice. Do not use underscore. As the app name has to be unique, make sure the name you use is distinctive. You can use your initials as part of the app name, for instance.

**Step 3| Define Procfile**

The *Procfile* executes a command when Heroku needs to run our server. Create one in the same directory as *index.js* (project's index.js) and name it as **Procfile** (the first alphabet must be capitalized, and there is no extension).

Add the following line to the Procfile:

| web: node index.js |
| --- |

Make sure to save the Procfile

**Step 4| Add a *start* script to package.json**

| {  "name": "06-api-auth",  "version": "1.0.0",  "description": "",  "main": "index.js",  "scripts": {  "test": "echo \"Error: no test specified\" && exit 1",  "start": "node index.js"  }, . . . } |
| --- |

**Step 5| Change the port that we are using**

In *index.js*, change the *3000* in *app.listen* to *process.env.PORT.*

| app.listen(process.env.PORT, () => {  console.log("Server has started"); }); |
| --- |

**Step 6| Push to Heroku**

Make sure you have a .gitignore file, and it must have *node\_modules, sessions/* and *.env* included,

| git add .  git commit -m "Deploy to Heroku"  git push heroku *master* |
| --- |

Note: If *git push heroku master* doesn't work, try: *git push heroku:main*

| If you get an error message, when pushing, regarding *pushing a shallow clone*, run the following command in the terminal first:  git fetch --unshallow |
| --- |

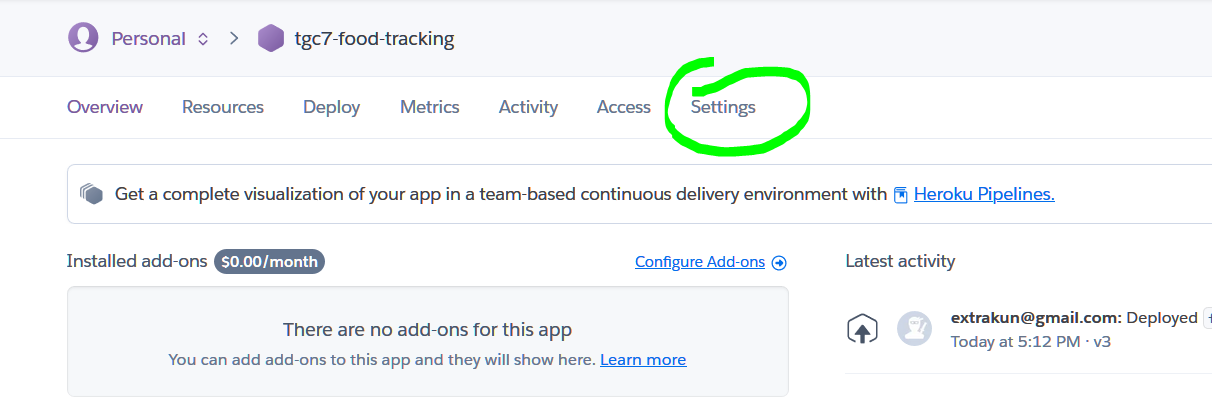
**Step 7| Setup the Database**

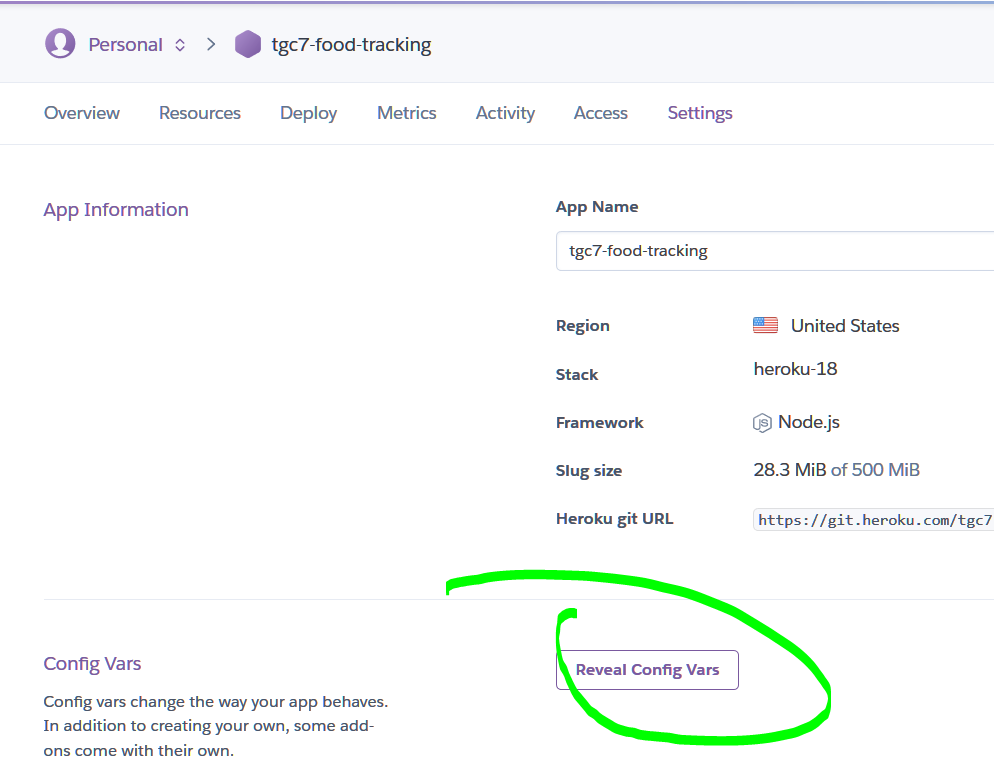
We need to use an external database, hosted on an external server, for our project. Heroku itself offers some database hosting services, such as Postgres and ClearDB. We are going with the former for this lab.

In the terminal, type in:

| heroku addons:create heroku-postgresql |
| --- |

When you are done, go to Heroku, and open your newly created application. Click on settings, and then "Reveal Config" (see the diagrams below):





You should be able to see a *DATABASE\_URL* setting.



**Step 8| Add database information to your .env file**

Make a copy of your .env file.

Open up Notepad, and paste in the *DATABASE\_URL* obtained from the previous step.

Follow the steps below to obtain the host, user, password and database name:

| The syntax is postgres://<user>:<password>@<host>/<database\_name>?reconnect = true  Example: postgres://b80f8d428xxxxx:f48exxxx@us-cdbr-iron-east-02.cleardb.net/heroku\_58632fb6debxxxx?reconnect=true  # host will be: us-cdbr-iron-east-02.cleardb.net  # user will be: B80f8d428xxxxx  # password will be: F48exxxx  # database\_name will be: heroku\_58632fb6debxxxx |
| --- |

In your .env file, change the setting *DB\_DRIVER* to *postgres*

Update your .env file with the host, user, password and database name obtained from parsing the syntax above:

| DB\_DRIVER=postgres DB\_USER=nzabcdefghkah DB\_PASSWORD=84f1d63eb61938670f2efa4aaaaaaaf6b725eeed2e19356e11db92a1 DB\_DATABASE=d1ldaaaa275 DB\_HOST=ec2-54-196-33-23.compute-1.amazonaws.com |
| --- |

Finally, install postgres with:

| yarn add pg  yarn add db-migrate-pg |
| --- |

We also have to push to *heroku* again:

| git add . git commit -m "Added postgres" git push heroku main |
| --- |

**Step 9| Setup tables with migrations**

Change your *database.json* to read as below:

| {  "dev": {  "driver": {"ENV" :"DB\_DRIVER"},  "user": {"ENV": "DB\_USER" },  "password": {"ENV":"DB\_PASSWORD"},  "database": {"ENV":"DB\_DATABASE"},  "host": {"ENV":"DB\_HOST"},  "ssl": {  "rejectUnauthorized": false  }  } } |
| --- |

Do the same for *bookshelf/index.js*

const knex = require('knex')({

'client': process.env.DB\_DRIVER,

'connection': {

'user': process.env.DB\_USER,

'password': process.env.DB\_PASSWORD,

'database': process.env.DB\_DATABASE,

'host':process.env.DB\_HOST,

'ssl': {

'rejectUnauthorized': false

}

}

})

| NOTE: Since we have switched to a different database, it is pristine and does not have any tables. Your original data are still on your local host, and you can see them back once you switch the DB settings back to MySQL |
| --- |

In the terminal, type in:

| ./db-migrate.sh up |
| --- |

The migration takes a longer time to run now because it is happening on a remote server.

**Step 10 | Copy all settings from the .ENV file to Heroku**

Once more, go to your application in Heroku and copy over the various settings from your .env file over.

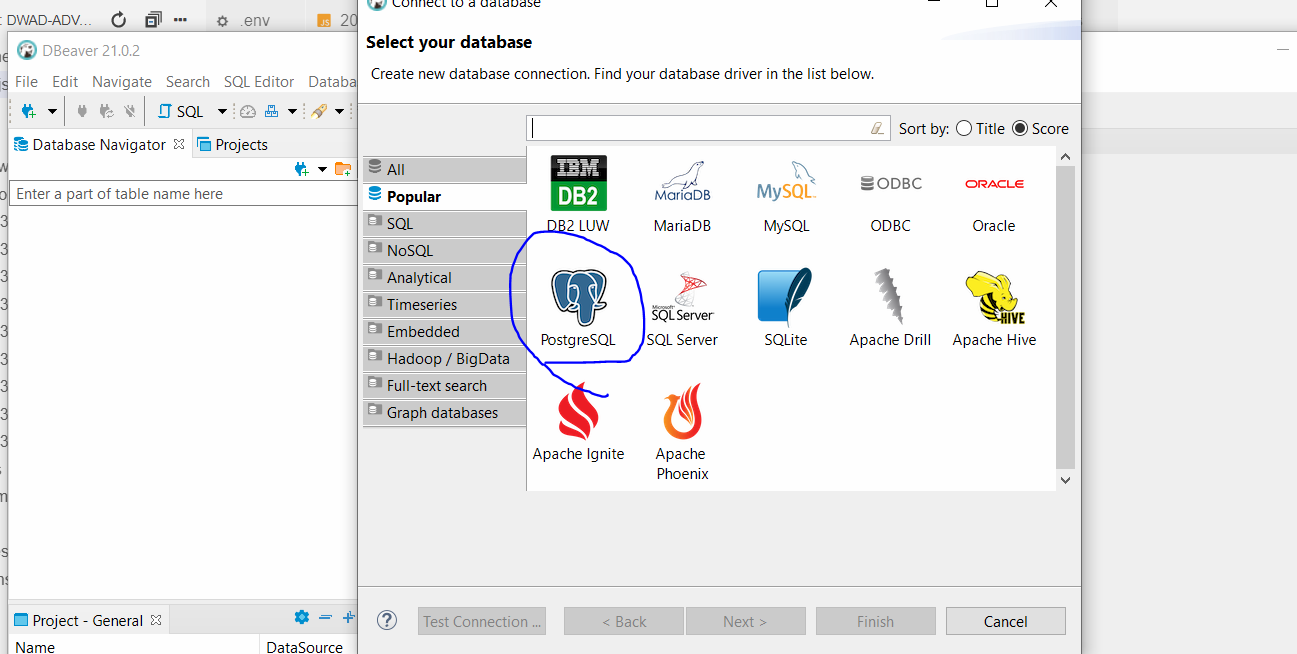
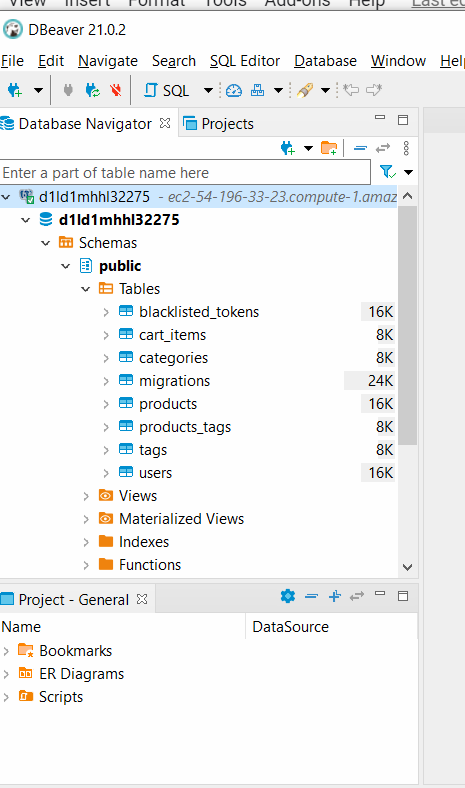
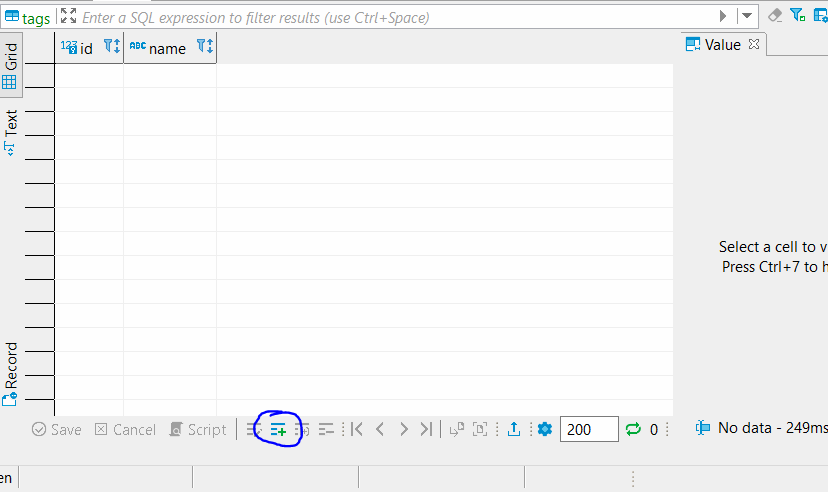
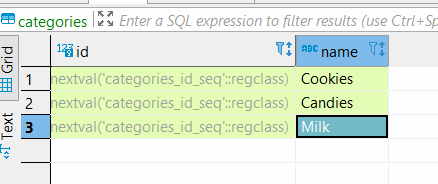
**Step 11 Do a commit and then push to Heroku**

We have made some changes to our code, so be sure to commit and push.

**Step 12| Install DB Beaver**

We need a software that allows us to add in new categories, tags and etc, and one way of connecting to the new Postgres database that we have is *Dbeaver.* Head off to <https://dbeaver.io/> to download the community version of Dbeaver and install that on your computer.

After this, launch the software.

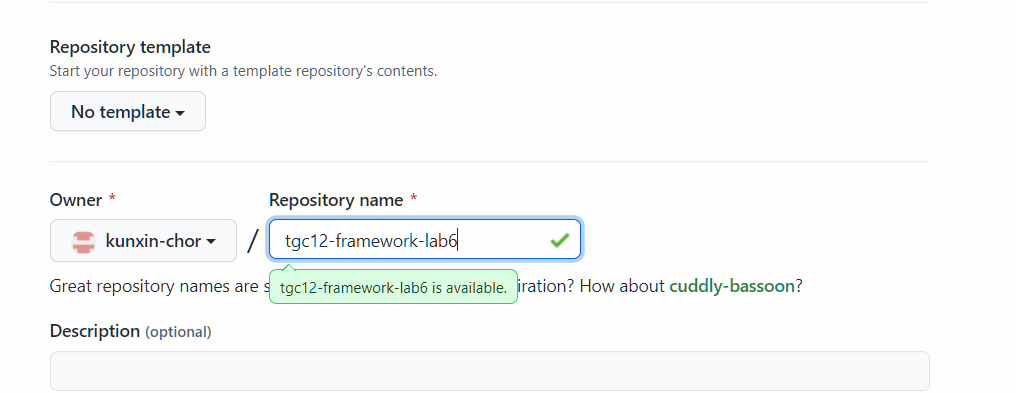
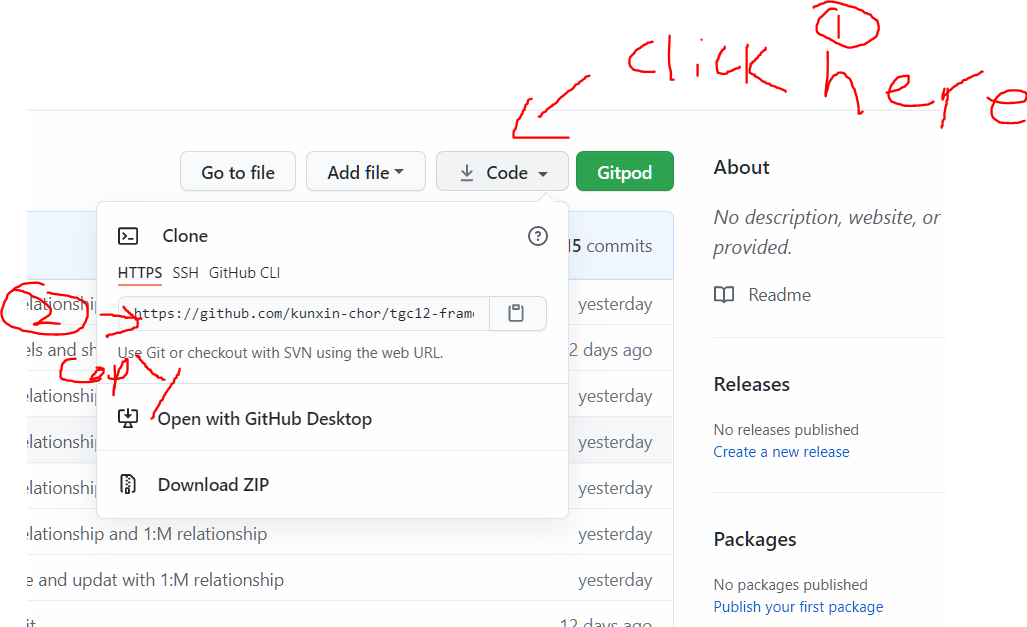
1. From the pop up window, select Postgres:  
     
   
2. It will then request to download some necessary files. Allow the operation.
3. In the window that shows up next, fill in the Postgres database you obtained from step 8. Once finished, click on the *Finish* button.
4. The new connection will appear on the left hand side window. Double click on it. You will be able to see all your tables once you collapse the *schemas* then *publics* folder:  
     
   
5. Select the categories table.
6. Click on the *Insert Row* button at the bottom of the table (or just press ALT + INSERT) to add a new row. (See diagram below):  
     
   
7. Add as many categories as you like. Do not fill in the *id* column (leave it as the text in grey).  
     
   
8. Click on the *Save* button at the bottom of the table when done
9. Do the same for tags.

**Step 13| Generate a new endpoint secret for your Heroku checkout**

Go to Stripe, and add in a new endpoint for *https::<heroku url>/checkout*/*process\_payment*, and replace the old endpoint secret with the new one in your Heroku settings.

# 

# APPENDIX A - STARTING FROM A CHECKPOINT

1. You can resume the walkthrough at any branch. Select that branch in Github and click on the [Gitpod] button
2. Follow the instructions in the readme.md file to set up the project.
3. If you wish to push to your own Github repository, follow the steps below:
   1. Create a new Github repository with default settings. There is no need to select a template.  
        
      
   2. Get the URL of your Github repository -- click on the [Code] button and then copy the URL displayed. Make sure tab reads HTTPS:  
        
      
   3. Type in the following commands at your terminal:

| git remote set-url origin <your github url>.git  git push origin HEAD |
| --- |

# APPENDIX B - GENERATING YOUR OWN HASH

https://newbedev.com/javascript-how-to-generate-hash-key-in-nodejs-api-code-example

1. Of course, when creating your own project, you should substitute *organic* with the name of the database you want to create. [↑](#footnote-ref-0)
2. Of course, substitute this with the name of the database you have created for your own project [↑](#footnote-ref-1)
3. A *shell script* allows us to store commands such as *ls* etc. inside a file and run it as if it is a program [↑](#footnote-ref-2)
4. It is possible to attach more than one middleware to a route. [↑](#footnote-ref-3)