My first Ecommerce project

start: 04/10/20

Set things up

Create the ROOT folder somewhere on your PC/mac: call it STORE

Open it using VS code.

1.make sure you have git & node.js latest version

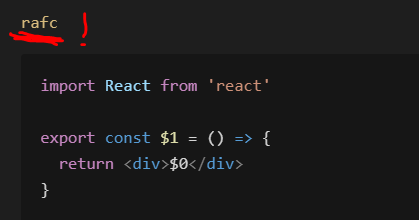
Just download the msi file. Check version by >>node -v

<https://nodejs.org/en/download/current/>

>>npm i -g npm

<https://www.npmjs.com/>

2. VS code extensions:

* ES7 react /redux/graphQL/ react-native snippits: rcc , rafce
* Bracket pair colorizer
* Auto rename tag -changing opening tac with the closing tag in jsx and html
* Javascript ES6 snippits: clg +enter=> console.log()
* Prettier

FRONT END-Start with create-react-app

Open Terminal at root level:

>>npx create-react-app frontend (yes! Npx!) :will create it inside frontend folder.

>>cd frontend

>>npm start //will run our frontend (react) web dev server

// ctrl +c will terminate the front end dev server.

Go to package.json=> check out the scripts and dependencies



React data flow:

react is a single page application

public-> index.html : the browser initially will read it.

It doesn’t have much in it except for a div with an id attribute “root”, this is the react entry point from:

src-> index.js which contains:

import App from './App';

ReactDOM.render(

  <React.StrictMode>

    <App />

  </React.StrictMode>,

  document.getElementById('root')

);

Src -> app.js is a component containing data that will be rendered to the DOM .

Public-> index.html : clean it a little bit:

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="utf-8" />

    <link

      rel="icon"

      href="%PUBLIC\_URL%/Polyverse\_logo\_ico"

      type="image/x-icon"

    />

    <meta name="viewport" content="width=device-width, initial-scale=1" />

    <meta name="theme-color" content="#000000" />

    <meta

      name="description"

      content="POLYVERSE- makes your music feel different"

    />

    <link rel="apple-touch-icon" href="%PUBLIC\_URL%/logo192.png" />

    <link rel="manifest" href="%PUBLIC\_URL%/manifest.json" />

    <title>Polyverse Music</title>

  </head>

  <body>

    <noscript>You need to enable JavaScript to run this app.</noscript>

    <div id="root"></div>

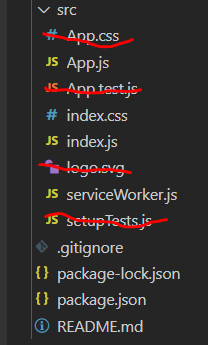
  </body>

</html>

will add a description in the tab and a small logo photo.

**Src->**

Files to clean:



Src-> app.js => lets clean it so the app won’t run what we’ve just deleted:

Did you notice the **JSX** syntax? Same as html with slight different.

Lets clear this code :

 <div className="App">

      <header className="App-header">

        <img src={logo} className="App-logo" alt="logo" />

        <p>

          Edit <code>src/App.js</code> and save to reload.

        </p>

        <a

          className="App-link"

          href="https://reactjs.org"

          target="\_blank"

          rel="noopener noreferrer"

        >

          Learn React

        </a>

      </header>

    </div>

Let’s make it an arrow function component:

import React from 'react'

const App = () => {

  return (

    <div>

      <h1>Hi!</h1>

    </div>

  )

}

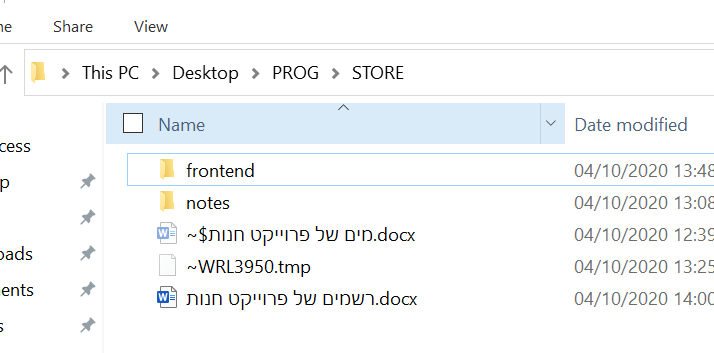
export default App

src -> index.css =>we will also delete all the content in this file

GIT- version control

Make sure you have the latest version of git.

Terminate the front end dev server by ctrl + c.

Go in windows to the frontend folder: right click ->

Bit bash here -> git bash cli opens ->

>> ls -a

//you will see all the hidden folders under the frontend folder and specificly the /.git

We want to delete it!

>> rm -rf .git

Now go to the  **.gitignore**  file and move it so it will be under frontend folder!

This file will let to define which folders/ files you don’t want to move to your github repository:

Basically node\_modules folder (because you may just download the repo and use

>> npm i to install all dependencies.

We will make sure you add =>

# dependencies

node\_modules -> this one is the frontend node modules.

node\_modules/ -> this one is the backend node modules.

# misc

.env

.env ->a file in which we will store some global variables that may contain sensitive information such as pay pal API key , mongoDB URI – we don’t want GitHub users to see.

**Initializing GitHub Repository:**

In the terminal :

We want to be on the root folder -> STORE (right now you should be on frontend).

>> cd ..

>> git init

>> git add . 🡺 stage all files!

>> git status 🡺 check which files got staged

>> git commit -m 'React first setup'

* Go to github and open a new reposetory

>> git remote add origin “repo URL…”

>>git push -u origin master

Next time in order to push new changes:

That’s is!

Now Relaunch the React dev server:

>>cd frontend

>>npm start

React-Working on the Front End.

Src-> create new folder -> compponents

Here all the components files will be.

Create inside components folder -> Header.js

Header.js 🡺 is our first component file! Naming conventions 🡺 upper case first letter + .js.

Now use the 🡺 rafce ->code snippet shortcut

So you will create a react arrow functional component syntax with export expression.

import React from 'react'

const Header = () => {

    return (

        <div>

        </div>

    )

}

export default Header

in the same manner create Footer.js file.

Render some text in both files.

App.js 🡺 import the footer and header components and render them:

import React from 'react'

import Header from './components/Header'

import Footer from './components/Footer'

const App = () => {

  return (

    <div>

      <Header />

      <main>

        <h1>this is some text</h1>

      </main>

      <Footer />

    </div>

  )

}

export default App

Using Bootswatch theme & React Bootstrap

\*you may also use Material UI

\*styled components.

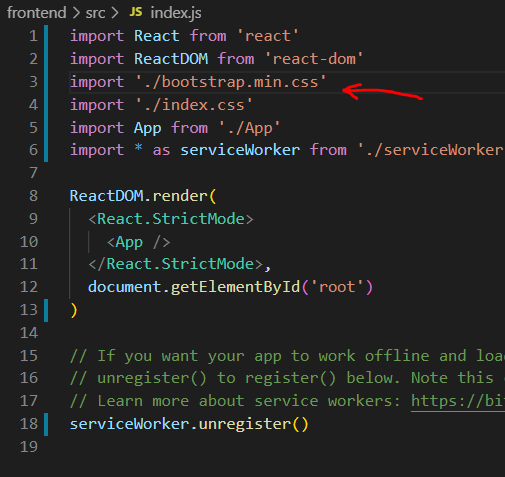
\*CSS

<https://bootswatch.com/>

download a template as you wish 🡺 bootstrap.min.css

drag the file to frontend-> src.

Index.js 🡺



You may see that the font has change.

Now let’s install Bootstrap:

Terminal:

Make sure you on frontend folder.

>> npm I react-bootstrap

Go back to App.js

Import container from bootstrap and use it to render h1, will move the text 20% to the center of the page:

import React from 'react'

import { Container } from 'react-bootstrap'

import Header from './components/Header'

import Footer from './components/Footer'

const App = () => {

  return (

    <div>

      <Header />

      <main>

        <Container>

          <h1> Poly-this is some text</h1>

        </Container>

      </main>

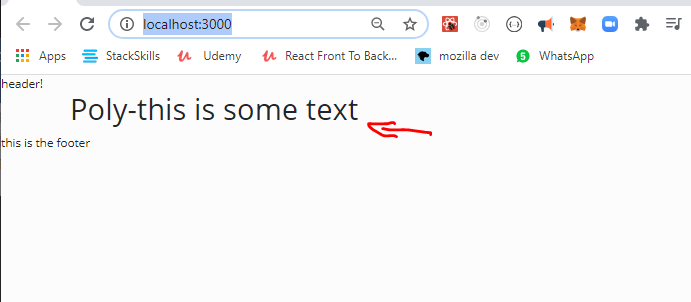
      <Footer />

    </div>

  )

}

export default App



Footer.js 🡺 we want to center the text in the footer and push it to the bottom of the page.

We will use bootstrap and CSS at index.css file

Import some more bootstrap components: Row & Col.

import React from 'react'

import { Container, Row, Col } from 'react-bootstrap'

const Footer = () => {

  return (

    <footer>

      <Container>

        <Row>

          <Col className='text-center py-3'>Copyright &copy; Polyverse </Col>

        </Row>

      </Container>

    </footer>

  )

}

export default Footer

lets explain 🡺

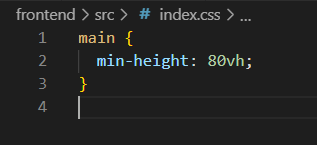
<Col className='text-center py-3'>Copyright &copy; Polyverse </Col>

className='text-center py-3' 🡺 we use className to pass the visual inputs to bootstrap.

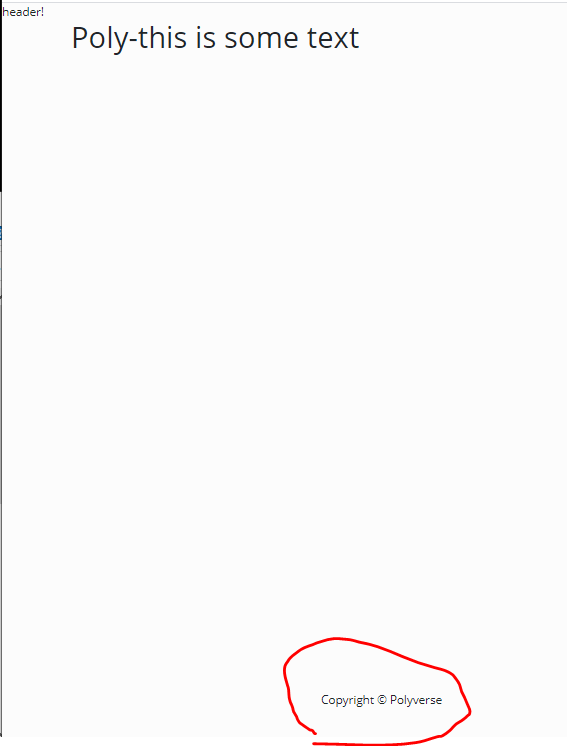
Text center + padding top and bottom.

&copy;  🡺 is the copyright © sign

Index.css 🡺 will push the text down.



And the result :



**Creating a NAV BAR**

Header.js🡺

<https://react-bootstrap.github.io/components/navbar/>

choose from the documentation a navbar code example and add it to the Header component:

import React from 'react'

import { Navbar, Nav, Container } from 'react-bootstrap'

import mainLogo from './mainlogo.png'

const Header = () => {

  return (

    <header>

      <Navbar bg='light' expand='lg' collapseOnSelect>

        <Container>

          <Navbar.Brand href='/'>

            <img

              src={mainLogo}

              className='d-inline-block align-top'

              alt='Polyverse logo'

            />

          </Navbar.Brand>

          <Navbar.Toggle aria-controls='basic-navbar-nav' />

          <Navbar.Collapse id='basic-navbar-nav'>

            <Nav className='ml-auto'>

              <Nav.Link href='/cart' varient='dark'>

                <h2>Cart</h2>

              </Nav.Link>

              <Nav.Link href='/login' varient='dark'>

                <h2>Sign In</h2>

              </Nav.Link>

            </Nav>

          </Navbar.Collapse>

        </Container>

      </Navbar>

    </header>

  )

}

export default Header

* Make sure you add your logo.png to the components folder. And import it to the component.



Adding icons with font awesome CDN:

Go to: <https://cdnjs.com/>

Search for font awesome CDN:



Copy Link Tag </>

And paste in public 🡺 index.html

Above the title tag :

 <link

      rel="stylesheet"

      href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.14.0/css/all.min.css"

      integrity="sha512-1PKOgIY59xJ8Co8+NE6FZ+LOAZKjy+KY8iq0G4B3CyeY6wYHN3yt9PW0XpSriVlkMXe40PTKnXrLnZ9+fkDaog=="

      crossorigin="anonymous"

    />

    <title>Polyverse Music</title>

Lets search for icons for shopping cart & login:

Implement it in the navbar code:

 <Nav className='ml-auto'>

              <Nav.Link href='/cart' varient='dark'>

                <h4>

                  {' '}

                  <i className='fas fa-shopping-cart'></i> Cart

                </h4>

              </Nav.Link>

              <Nav.Link href='/login' varient='dark'>

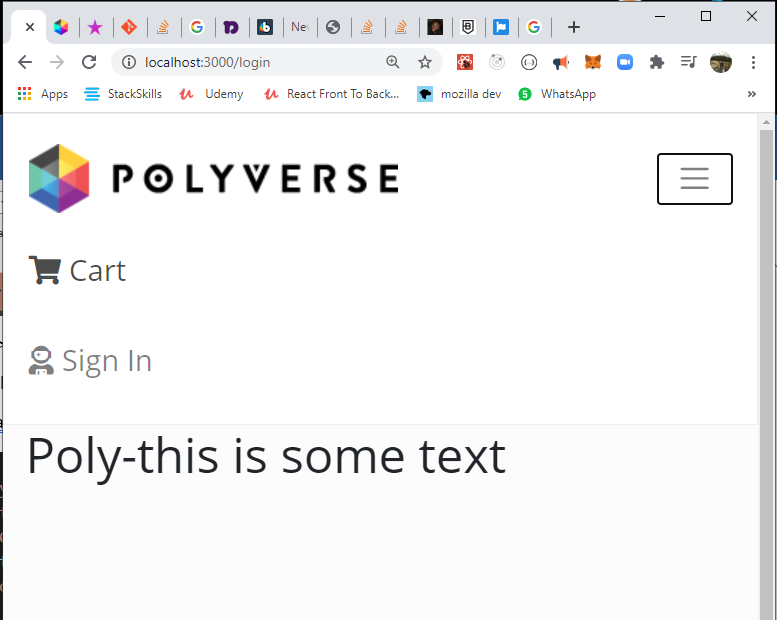
                <h4>

                  <i className='fas fa-user-astronaut'></i> Sign In

                </h4>

              </Nav.Link>

 </Nav>



Website main content area:

Now lets create some space between the web content and the nav bar with bootstrap className=’py-4’

Src->App.js 🡺

const App = () => {

  return (

    <div>

      <Header />

      <main className='py-4'>

        <Container>

          <h1> Poly-this is some text</h1>

        </Container>

      </main>

      <Footer />

    </div>

  )

}

export default App

**Adding products:**

Frontend->public 🡺 create a new folder “images” : add to it all your products photos.

Frontend -> src 🡺 create a new file : products.js .

add to it products: an array of objects.

const products = [

  {

    \_id: '1',

    name: 'Airpods Wireless Bluetooth Headphones',

    image: '/images/airpods.jpg',

    description:

      'Bluetooth technology lets you connect it with compatible devices wirelessly High-quality AAC audio offers immersive listening experience Built-in microphone allows you to take calls while working',

    brand: 'Apple',

    category: 'Electronics',

    price: 89.99,

    countInStock: 10,

    rating: 4.5,

    numReviews: 12,

  }]

Lets start working on our shop:

Src🡺 new folder “screens” 🡺 new file HomeScreen.js

Import products (notice it is lower case import!!!)

and bootstrap Row & Col

Render all product names to the dom.

<Col sm={12} md={6} lg={4} xl={3}>

🡺 a syntax for what will be the column size on different screens (small, medium, large …)

import React from 'react'

import { Row, Col } from 'react-bootstrap'

import products from '../products'

const HomeScreen = () => {

  return (

    <>

      <h1> Latest Products </h1>

      <Row>

        {products.map((product) => (

          <Col sm={12} md={6} lg={4} xl={3}>

            <h3>{product.name}</h3>

          </Col>

        ))}

      </Row>

    </>

  )

}

export default HomeScreen

add the component to the App.js 🡺

import HomeScreen from './screens/HomeScreen'

const App = () => {

  return (

    <div>

      <Header />

      <main className='py-4'>

        <Container>

          <HomeScreen />

        </Container>

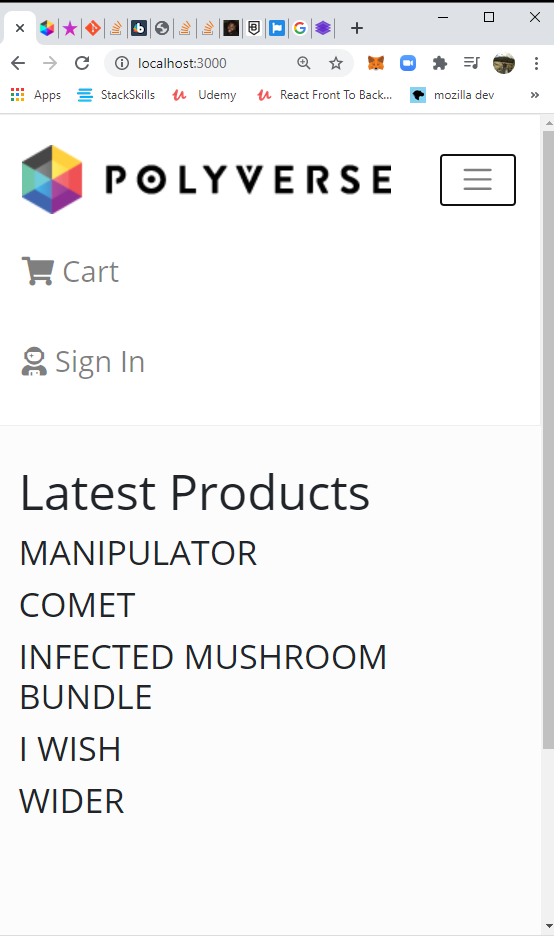
      </main>

      <Footer />

    </div>

  )

}



**Product CARD component:**

**Src -> components 🡺 new file ProductCard.js**

import React from 'react'

import { Card } from 'react-bootstrap'

const ProductCard = ({ product }) => { //destructure product from the prop

  return (

    <Card className='my-3 p-3 rounded zoom'> // margin padding round corners & zoom on hover!!!

      <a href={`/product/${product.\_id}`}> //soon we will change it to react router Link

        <Card.Img src={product.image} variant='top' />

      </a>

    </Card>

  )

}

export default ProductCard

**In src->screens-> HomeScreen.js make sure you :**

import ProductCard from '../components/ProductCard'

and add the card component under the product name and pass it a prop=> of the current product being looped in the map method.

 <h1> Latest Products </h1>

      <Row>

        {productsCat.map((product) => (

          <Col sm={12} md={6} lg={4} xl={3}>

            <ProductCard product={product} />

          </Col>

        ))}

      </Row>

**Zoom on hove Card :** we added a className of zoom to the card in ProductCard.js

Index.css 🡺

.zoom {

  transition: transform 0.2s; /\* Animation \*/

  margin: 0 auto;

}

.zoom:hover {

  transform: scale(1.1); /\* (10% zoom)\*/

  z-index: 1;

}

**Product card.js:**

Lets design the structure of the card: an image, name of product, star ratings, number of reviews and price.

Add an import to Ratings.js – this component renders the star icons ratings.

import React from 'react'

import { Card } from 'react-bootstrap'

import Ratings from './Ratings'

const ProductCard = ({ product }) => {

  //destructure props

  return (

    <Card className='my-3 p-3 rounded zoom'>

      <div>

        <a href={`/product/${product.\_id}`}>

          <Card.Img src={product.image} variant='top' />

        </a>

      </div>

      <Card.Body>

        <a href={`/product/${product.\_id}`} className='text-decoration-none'>

          <Card.Title as='div' className='text-info'>

            <p class='font-weight-bold h4'>{product.name}</p>

          </Card.Title>

        </a>

        <Card.Text as='div' className='my-3 text-dark h5'>

          <Ratings value={product.rating} text={product.numReviews} /> //pass two props: the rating and number of reviews!

        </Card.Text>

        <Card.Text as='h3'>${product.price}</Card.Text>

      </Card.Body>

    </Card>

  )

}

export default ProductCard

components-> Ratings.js 🡺 this component will return stars rating and beneath it a text of number of reviewers.   
CHECK OUT THE STAR ICON AT <https://fontawesome.com/icons?d=gallery&q=star>

FIND HALF STAR ICON ALSO!

import React from 'react'

const Ratings = ({ value, text }) => {

  return (

    <div className='rating'>

      <span>

        <i

          className={

            value >= 1

              ? 'fas fa-star'

              : value >= 0.5

              ? 'fas fa-star-half-alt'

              : 'far fa-star' //empty star

          }

        ></i>

      </span>

      <span>

        <i

          className={

            value >= 2

              ? 'fas fa-star'

              : value >= 1.5

              ? 'fas fa-star-half-alt'

              : 'far fa-star'

          }

        ></i>

      </span>

      <span>

        <i

          className={

            value >= 3

              ? 'fas fa-star'

              : value >= 2.5

              ? 'fas fa-star-half-alt'

              : 'far fa-star'

          }

        ></i>

      </span>

      <span>

        <i

          className={

            value >= 4

              ? 'fas fa-star'

              : value >= 3.5

              ? 'fas fa-star-half-alt'

              : 'far fa-star'

          }

        ></i>

      </span>

      <span>

        <i

          className={

            value >= 5

              ? 'fas fa-star'

              : value >= 4.5

              ? 'fas fa-star-half-alt'

              : 'far fa-star'

          }

        ></i>

      </span>

      <p>{text > 1 ? `${text} Artists reviews` : `${text} Artist review`}</p>

      {/\*may also write {text && text} \*/}

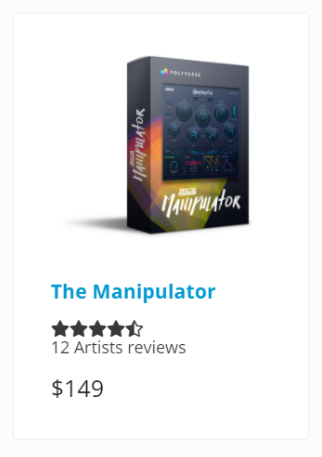
    </div>

  )

}

export default Ratings

**SO THE CARD WILL LOOK LIKE THAT :**



**Ratings.js 🡺**

Lets add more features:

1. Yellow color to the rating stas (as a prop that its color set by default as yellow)
2. Set up default prop method.
3. Set up propTypes to regulate the data type that we receive in the props.

import React from 'react'

import PropTypes from 'prop-types' //impt -shortcut. //import propTypes!

const Ratings = ({ value, text, starColor }) => { //star color prop!

  return (

    <div className='rating'>

      <span>

        <i

          style={{ color: starColor }} //react CSS inline style as star color prop!- make sure you code-> style ={{ }}.

          className={

            value >= 1

              ? 'fas fa-star'

              : value >= 0.5

              ? 'fas fa-star-half-alt'

              : 'far fa-star' //empty star

          }

        ></i>

      </span>

      <span>

        <i

          style={{ color: starColor }}

          className={

            value >= 2

              ? 'fas fa-star'

              : value >= 1.5

              ? 'fas fa-star-half-alt'

              : 'far fa-star'

          }

        ></i>

      </span>

      <span>

        <i

          style={{ color: starColor }}

          className={

            value >= 3

              ? 'fas fa-star'

              : value >= 2.5

              ? 'fas fa-star-half-alt'

              : 'far fa-star'

          }

        ></i>

      </span>

      <span>

        <i

          style={{ color: starColor }}

          className={

            value >= 4

              ? 'fas fa-star'

              : value >= 3.5

              ? 'fas fa-star-half-alt'

              : 'far fa-star'

          }

        ></i>

      </span>

      <span>

        <i

          style={{ color: starColor }}

          className={

            value >= 5

              ? 'fas fa-star'

              : value >= 4.5

              ? 'fas fa-star-half-alt'

              : 'far fa-star'

          }

        ></i>

      </span>

      <p>{text > 1 ? `${text} Artists reviews` : `${text} Artist review`}</p>

      {/\*may also write {text && text} \*/}

    </div>

  )

}

//set default prop

Ratings.defaultProps = {

  //set starColor (it's a component props) as yellow by default

  starColor: '#fada00',

}

//propTypes- setting the data types of the props. if conditions not met, will throw an error in consule.

Ratings.propTypes = {

  value: PropTypes.number.isRequired,

  text: PropTypes.string.isRequired,

  starColor: PropTypes.string,

}

export default Ratings

Implementing React Router!

First- install react router and react router boostrap:

Terminal :

>>cd frontend

>> npm i react- router-dom react-router-bootstrap

App.js 🡺

from this:

import React from 'react'

import { BrowserRouter as Router, Rout } from 'react- router-dom'

import { Container } from 'react-bootstrap'

import Header from './components/Header'

import Footer from './components/Footer'

import HomeScreen from './screens/HomeScreen'

const App = () => {

  return (

    <Router>

      <Header />

      <main className='py-4'>

        <Container>

          <HomeScreen />

        </Container>

      </main>

      <Footer />

    </Router>

  )

}

export default App

to this:

import React from 'react'

import { BrowserRouter as Router, Route } from 'react- router-dom' //import of react router!

import { Container } from 'react-bootstrap'

import Header from './components/Header'

import Footer from './components/Footer'

import HomeScreen from './screens/HomeScreen'

import ProductScreen from './screens/ProductScreen' //import of a new component

const App = () => {

  return (

    <Router> //must wrap the whole jsx code!

      <Header />

      <main className='py-4'>

        <Container>

          <Route path='/' component={HomeScreen} exact /> //exact because it’s the root url! There is nothing after the ‘/’

          <Route path='/product/:id' component={ProductScreen} /> // path’s id value will be a parameter because we wrote it as one -> “/:XYZ” , check the dynamic code in ProductCard.js who’s generating this path.

And ProductScreen.js as the one who needs acceses to this Quary param

        </Container>

      </main>

      <Footer />

    </Router>////must wrap the whole jsx code!

  )

}

export default App

lets make a ProductScreen component to show each product page:

screens -> ProductScreen.js

import React from 'react'

const ProductScreen = () => {

  return <div>product</div>

}

export default ProductScreen

we also now need to change the <a href> tags in the card component to react router Link component! This is the link that generates the dynamic URL with the item’s id.

components-> Productcard.js :

import React from 'react'

import { Link } from 'react-router-dom' //import Link

import { Card } from 'react-bootstrap'

import Ratings from './Ratings'

const ProductCard = ({ product }) => {

  //destructure props

  return (

    <Card className='my-3 p-3 rounded zoom'>

      <div>

        <Link to={`/product/${product.\_id}`}> //will rout us to this url.

          <Card.Img src={product.image} variant='top' />

        </Link>

      </div>

      <Card.Body>

        <Link to={`/product/${product.\_id}`} className='text-decoration-none'>

          <Card.Title as='div' className='text-info'>

            <p class='font-weight-bold h4'>{product.name}</p>

          </Card.Title>

        </Link>

        <Card.Text as='div' className='my-3 text-dark h5'>

          <Ratings value={product.rating} text={product.numReviews} />

        </Card.Text>

        <Card.Text as='h3'>${product.price}</Card.Text>

      </Card.Body>

    </Card>

  )

}

export default ProductCard

lets use react router in the NavBar

we will use react-router-bootstrap for that. Only then we may wrap the react-bootstrap components.

Why we do this anyways? Right now when we click links in the navbar the page will refresh, because react is a single page app, we don’t welcome this kind of behavior.

components-> Header.js 🡺

import React from 'react'

import { Navbar, Nav, Container } from 'react-bootstrap'

import { LinkContainer } from 'react-router-bootstrap'

import mainLogo from './mainlogo.png'

const Header = () => {

  return (

    <header>

      <Navbar bg='light' expand='lg' fixed='top' collapseOnSelect>

        <Container>

          <LinkContainer to='/'>

            <Navbar.Brand>

              <img

                src={mainLogo}

                className='d-inline-block align-top'

                alt='Polyverse logo'

              />

            </Navbar.Brand>

          </LinkContainer>

          <Navbar.Toggle aria-controls='basic-navbar-nav' />

          <Navbar.Collapse id='basic-navbar-nav'>

            <Nav className='ml-auto'>

              <LinkContainer to='/cart'>

                <Nav.Link varient='dark'>

                  <h4>

                    {' '}

                    <i className='fas fa-shopping-cart'></i> Cart

                  </h4>

                </Nav.Link>

              </LinkContainer>

              <LinkContainer to='/login'>

                <Nav.Link varient='dark'>

                  <h4>

                    <i className='fas fa-user-astronaut'></i> Sign In

                  </h4>

                </Nav.Link>

              </LinkContainer>

            </Nav>

          </Navbar.Collapse>

        </Container>

      </Navbar>

    </header>

  )

}

export default Header

working on the product page:

\*after we click the desired product we will be routed to the url/product/:id

In order to show the right content we must get access to the **id URL parameter** from the productScreen.js component!

We do this by using **props.match.params.id** 🡺

import React from 'react'

import { Link } from 'react-router-dom'

import { Row, Col, Image, ListGroup, Card, Buttton } from 'rect-bootstrap'

import Ratings from '../components/Ratings'

import products from '../products' //its not a react component but a js variable

const ProductScreen = (props) => {

    //accessing the URL id param using props.match

    const product = products.find(element => element.\_id === props.match.param s.id ) //this is how you access it! And will later search the |DB for the right user id. Right now it’s a product array.

  return (

    <div>

      <br />

      <h1>product </h1>

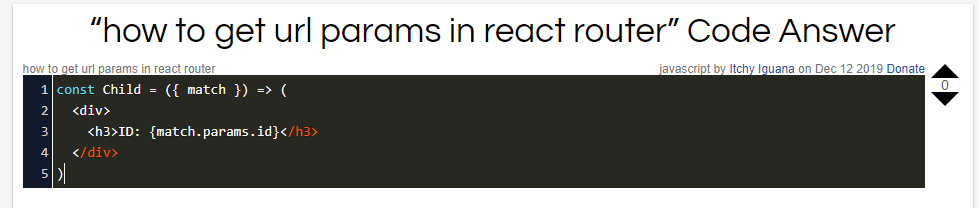
    </div>

  )

}

export default ProductScreen

**Destructure match from props:**



Lets build the productScreen content:

brace yourself , its a long one…

 return (

    <>

      <Link className='btn btn-light my-3' to='/'> //back button

        Back to Products

      </Link>

      <Row>

        <Col>

          <Image src={product.poster} alt={product.name} fluid /> //poster image

        </Col>

      </Row>

      <Row>

        <Col>

          <h1 className='text-center my-5'> //statement centered text

            A NEW KIND OF VOCAL TRANSFORMING PROCESSOR

          </h1>

        </Col>

      </Row>

      <Row>

        <Col>

          <Row className='justify-content-md-center'> //small interface photo centered in its father element (this is the father.)

            <Image src={product.interface} alt='product interface' fluid /> //fluid tag will make it appear as larger as possible

          </Row>

          <Row>

            <Card className='priceCard'> //see index.css for hover effect and pisitioning

              <ListGroup variant='flush'>

                <ListGroup.Item>

                  <Row>

                    <Col className='my-1 text-dark h5'>Price:</Col>

                    <Col className='my-1 text-success h5'>

                      <strong>${product.price}</strong>

                    </Col>

                  </Row>

                </ListGroup.Item>

                <ListGroup.Item>

                  <Row>

                    <Col className='my-1 text-dark h5'>Status:</Col>

                    <Col className='my-1 text-info h5'>

                      {product.status ? ' Available' : 'Coming Soon'}

                    </Col>

                  </Row>

                </ListGroup.Item>

                <ListGroup.Item className='text-center py-0 '>

                  <Button

                    // className='btn-block' //will make it spread all across its div.

                    variant='info'

                    type='button'

                    disabled={!product.status} //if status is false, button is disabled- for coming soon products.

                  >

                    Add to cart

                  </Button>

                </ListGroup.Item>

              </ListGroup>

            </Card>

          </Row>

        </Col>

        <Col>

          <ListGroup variant='flush'> //will take away the border of the list group

            <ListGroup.Item>

              <h2>{product.name}</h2>

            </ListGroup.Item>

            <ListGroup.Item>

              <h4>

                <Ratings value={product.rating} text={product.numReviews} />

              </h4>

            </ListGroup.Item>

            <Card

              style={{

                width: '18rem',

              }}

            >

              <Card.Body>

                <Card.Title>

                  {' '}

                  <strong>Description</strong>

                </Card.Title>

                <Card.Text style={{ fontSize: '1rem' }}>

                  {product.longDescription}

                </Card.Text>

                <Button variant='secondary'>Read more</Button>

              </Card.Body>

            </Card>

          </ListGroup>

        </Col>

      </Row>

      <Row>

        <Col className='text-center my-5'> //text centered

          <h1>FEATURES</h1>

          <ListGroup variant='flush'>

            {product.features.map((feature, i) => ( //don’t forget to use key when iterating a jsx code

              <ListGroup.Item key={i}>

                <h5>{feature}</h5>

              </ListGroup.Item>

            ))}

            <ListGroup.Item>

              <Image src={product.compatibility} alt='compatibility' fluid />

            </ListGroup.Item>

          </ListGroup>

        </Col>

      </Row>

    </>

  )

We will also add code in index.css: for the price card.

.priceCard {

  opacity: 0.6;

  margin-top: 2%;

  width: 13rem;

  position: fixed;

  top: 10%;

  right: 2%;

  z-index: 1;

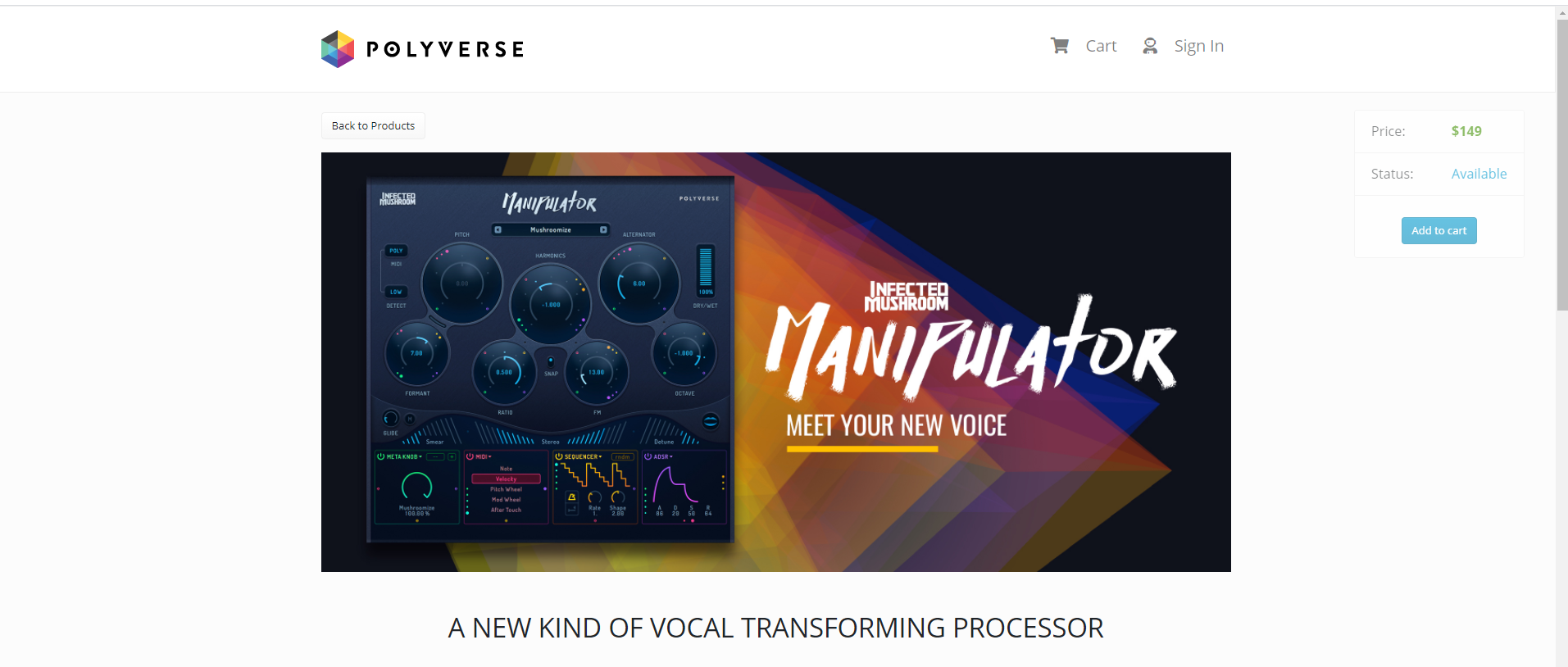
}

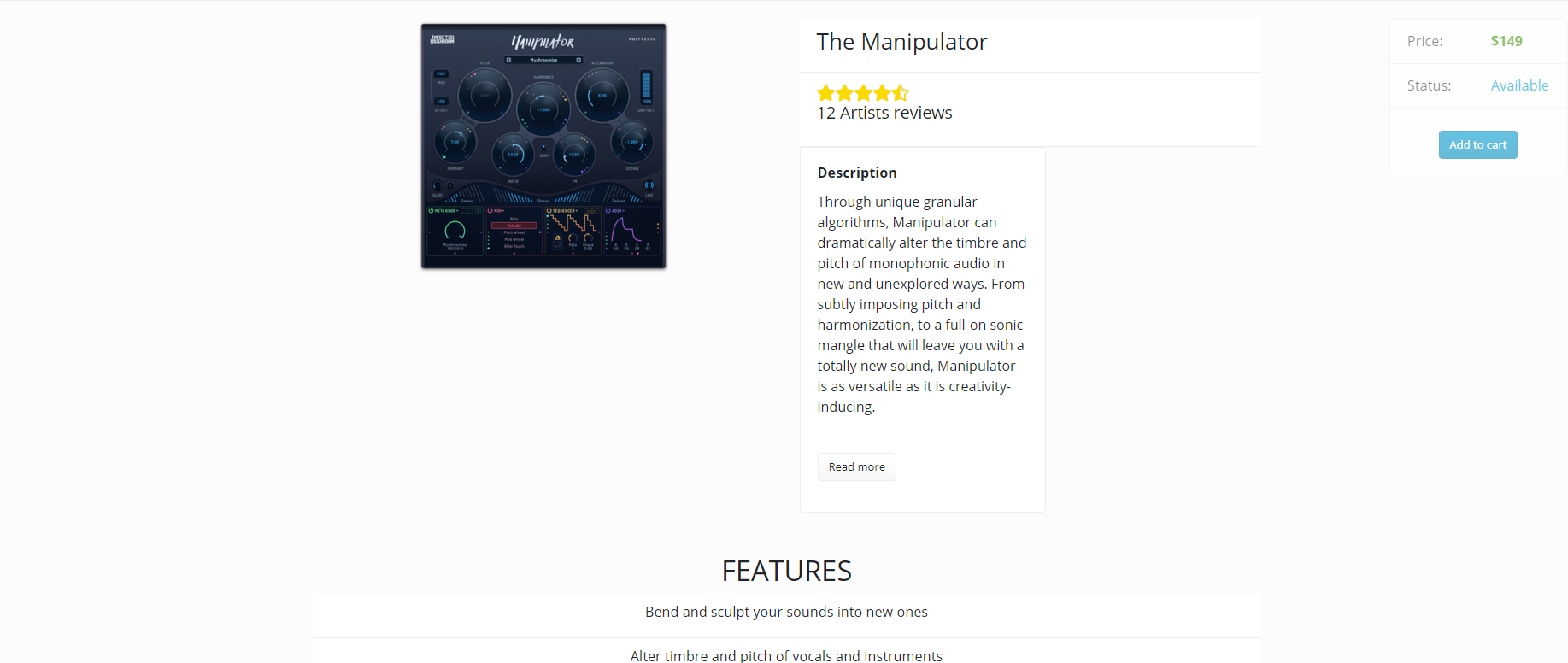
.priceCard:hover {

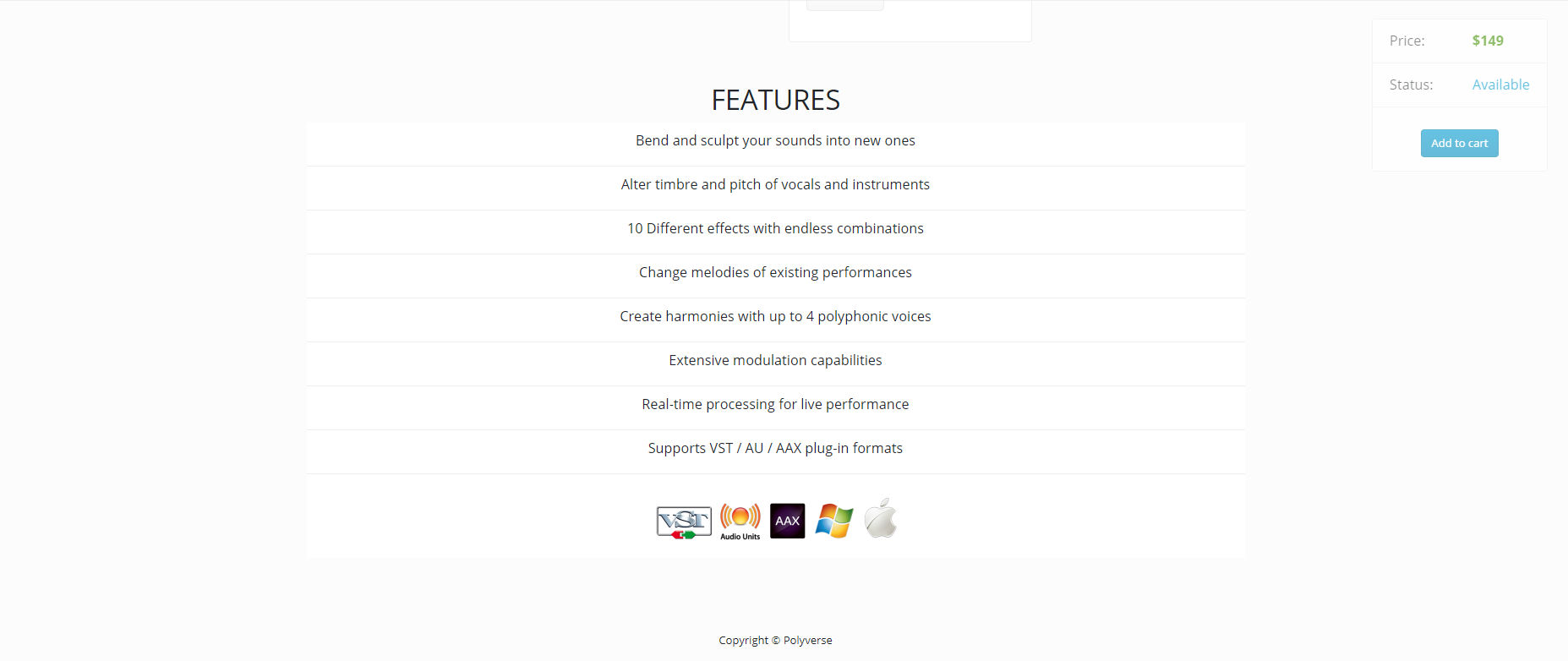
  opacity: 1;

}

This is how it will look:





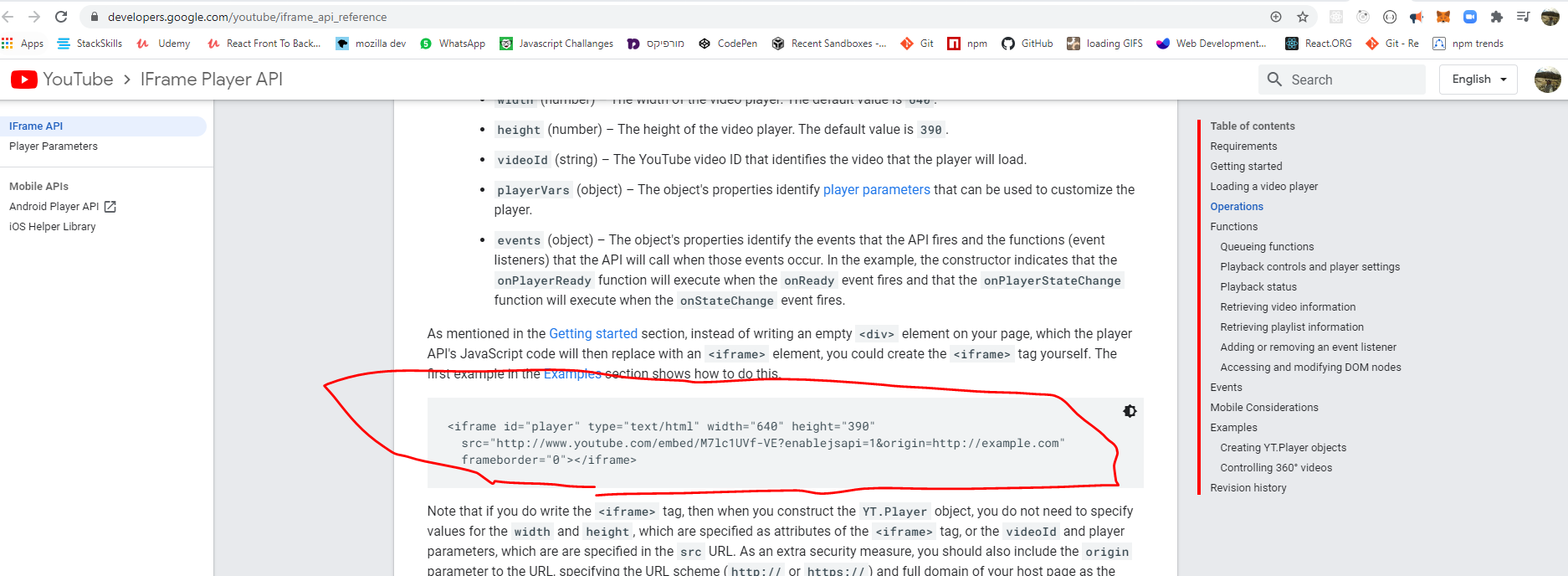


YouTube player component:

<https://www.youtube.com/watch?v=lT4uJI6TXAg&t=607s>

components-> YouTubePlayer.js

YouTube Player API Reference for iframe Embeds:



import React from 'react'

const YouTubePlayer = () => {

  return (

    <div className='youtube-player'>

      {/\*snippit from youtube api\*/}

      <iframe

        id='player'

        type='text/html'

        style={{

          width: '100%',

          height: '100%',

        }}

        src='http://www.youtube.com/embed/M7lc1UVf-VE?'

        frameborder='0'

      ></iframe>

    </div>

  )

}

export default YouTubePlayer

draggable component🡺

<https://medium.com/@edwebber06/react-draggable-components-not-such-a-drag-d4fd6098be38>

<https://www.youtube.com/watch?v=lT4uJI6TXAg&t=607s>

make sure you install:

<https://www.npmjs.com/package/react-resizable>

<https://www.npmjs.com/package/react-draggable>

import React from 'react'

import Draggable from 'react-draggable'

import { Resizable, ResizableBox } from 'react-resizable'

import 'react-resizable/css/styles.css'

const YouTubePlayer = ({ videoLink }) => {

  return (

    <Draggable handle='.handle'> //must pass the className of the object that will allow you to drag the div.

      <div

        className='youtube-player'

        style={{ zIndex: 1, paddingBottom: '5%' }}

      >

        <ResizableBox width={300} height={200}> //initial video box size

          {/\*snippit from youtube api\*/}

          <iframe

            title='iframe'

            id='player'

            type='text/html'

            style={{

              width: '100%',

              height: '100%',

            }}

            src={videoLink}

            frameborder='0'

          ></iframe>

          <div

            style={{

              display: 'flex',

              alignItems: 'flex-start',

              justifyContent: 'space-between',

            }}

          >

            <div className='handle'> //this is the icon to drag the video around

              <i className='fas fa-arrows-alt'></i>

            </div>

            <div className='expand'>

              <i className='fas fa-expand-alt'></i>

            </div>

          </div>

        </ResizableBox>

      </div>

    </Draggable>

  )

}

export default YouTubePlayer

index.css🡺

/\*see YouTubePlayer.js ////////////////////\*/

.youtube-player {

  display: flex;

  flex-direction: column;

  justify-content: center;

  align-items: center;

}

.youtube-player .handle {

  position: absolute;

  bottom: -20%;

  right: 95%;

  color: rgb(7, 7, 7);

  opacity: 0.85;

  font-size: 1.5rem;

  cursor: move;

}

.youtube-player .expand {

  position: absolute;

  bottom: -20%;

  left: 95%;

  font-size: 1.5rem;

  color: rgb(7, 7, 7);

  opacity: 0.85;

  cursor: pointer;

}

**Backend-Node.js and express framework, mongo DB and mongoose -object data mapper**

Beckend first setups:

1. Crate a folder in the root 🡺 beckend
2. Initialize package.json and node modules 🡺 go back to the **root** folder and in the **terminal** :  
   >>npm init
3. Press enter, then

Enter project description(=my shop), entry point(= server.js),author(=that’s all you!)

1. Install express 🡺root folder-> >>npm i express

Root folder -> beckend 🡺 new file : server.js (this is the entry point to our server).

1. beckend 🡺 new folder: data, copy to it the products file with the data of each product (Root->frontend-> products.js).
2. \*\*make sure the root level has now 2 new json files: package-lock.json & package.json

notice! **Importing** modules in the **front end** is different than the **back end**.

We don’t use the ***import X from “y”*** syntax (import ES modules)

Rather a ***const X = require(‘Y’)***  - common JS syntax.

But with since Node.js Version 14.4 you may use the ES import module syntax without using Babbel or experimental flag ***server.mjs*** .

(<https://www.youtube.com/watch?v=teDVlOjOCT0> )

beckend->server-> server.js 🡺

1. import express: with common JS syntax
2. set listener on port 5000

const express = require('express')

const app = express()

app.listen(5000, console.log('Hey! server is running on port 5000'))

To test the listener in the terminal go to the root folder by :

>>cd ..

Run the server.js we just created in the backend folder:

>>node backend/server

STORE> node backend/server

Hey! server is running on port 5000

Great. It’s on!

Now instead of running the server like that we can set a shorter script in the

package.json file🡺

we will edit the script section from this:

 "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1"

  },

 "scripts": {

    "start": "node backend/server"

  },

So now to run the server we will only run:

>> npm start

**Some Express methods : .get(), .use(),.listen(),.delete(), .set(), .post() and more…**

Lets make a get request🡺

app.get(path, callback [, callback ...])

Routes HTTP GET requests to the specified path with the specified callback functions.

app.get('/', function (req, res) {

res.send('GET request to homepage')

})

const express = require('express')

const app = express()

app.listen(5000, console.log('Hey! server is running on port 5000'))

app.get('/', function (req, res) { //request or response

//we will response with a send method= this will send the user some data!

res.send(‘this is what you GET when you request this path (/)’)

})

Now we will run the server :

Terminal: >> npm start

On the browser we will go to :

Localhost:5000/

**And we will see the output at the body of the browser!**

**We may also send with is a status method:**

res.status(404).send('Sorry, we cannot find that!')

**but we’ll get to that later. 😊**

Responding an http get request with a product! :

In “Backend first steps” #5 we added the products file to the data folder.

Now look at its content, anything exceptional?

Well… yes.

Backend->Data->Products.js🡺

We need to change its export syntax to module syntax:

module.exports = products

excellent!

Now lets code the response

Backend-> server.js:

const express = require('express')

const products = require('./data/products') //import the product list!

const app = express()

app.get('/', (req, res) => {

  res.send('this is what you GET when you request this path (/)')

})

app.get('/api/products', (req, res) => {

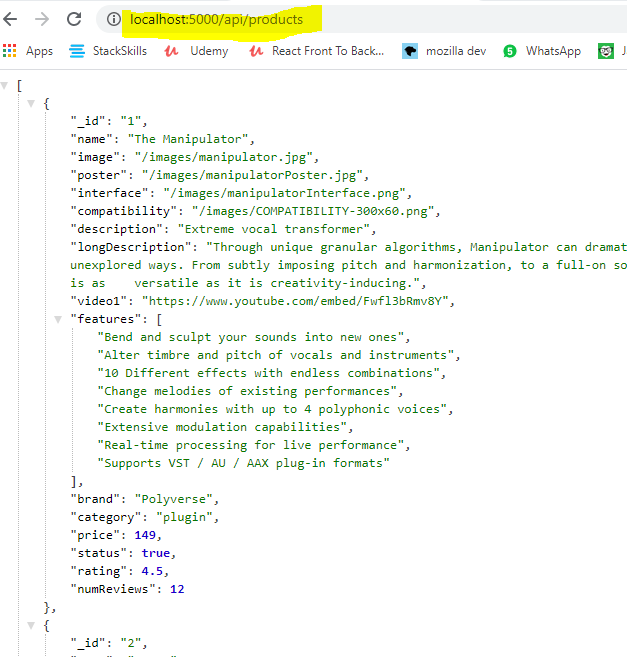
  //lets respond with a product information see backend->data-> products.js

  res.json(products) //.json will sent the data as a JSON format!

})

app.listen(5000, console.log('Hey! server is running on port 5000'))

In the browser:



Gaining access to url parameters:

Now lets serve a GET request from the browser => it is asking, by url parameter, a specific product data!

req.params.id 🡺 it allows us check the input in the url under “/:somthingWeLookingFor”

in this case the user id!

app.get('/api/products/:id', (req, res) => {

  //now lets serve a specific product data by url param, user id!

  const product = products.find((element) => element.\_id === req.params.id)

 //find the product in the products array with the id in the url

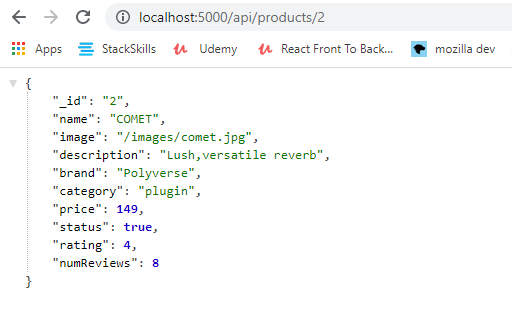
 res.json(product) //.json will sent the data as a JSON format!

})

In the browser:

We will run the server again >>npm start

And in the url ask for user with id === 1



Serve data from the server by requesting it in the frontend!

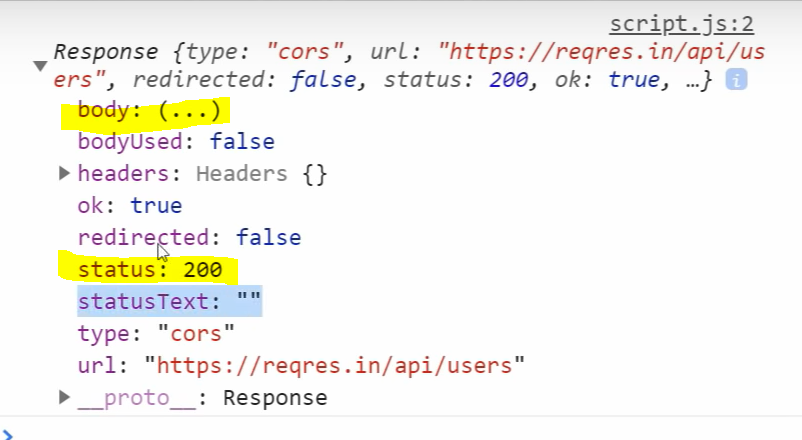
\*\*you should be familiar with:

***the fetch API***

**fetch (‘url’ , {option} ).then(res=> return res.json() ).then(data => console.log(data) ).catch(error)**

**option🡺 {method: ‘POST’ , headers: {‘Content-Type’ : ’application/json’} ,body: JSON.stringify({name : ’User 1’})**

* **you must stringify the body!!**
* **POST** 🡺 **adding data to the server.**
* **fetch** 🡺 **returns a promise. (so we can use async await or then().catch**
* **first then** 🡺 **returns the response object, its containing a body ,headers, status (200,404 and so on…) the body who is actually containing our data is not accessible. We must convert the response to json! => we do it by : res.json()=> which will return the data in the body BUT in another promise!! So we should use again another then.**



* **second then** 🡺 **returns the data itself.**
* **Catch statement: will return an Error only in the case of connection to url failure.**
* **In order to catch a 404 error (when trying to get a data that does not exists in the DB you must add in the first then statement : if (res.ok){}else{ console.log(error)}**

<https://www.pluralsight.com/guides/axios-vs-fetch>

<https://dev.to/shoupn/javascript-fetch-api-and-using-asyncawait-47mp>

async function getUserAsync(name)

{

let response = await fetch(`https://api.github.com/users/${name}`);

let data = await response.json()

return data;

}

getUserAsync('yourUsernameHere')

.then(data => console.log(data));

Lets start!

**Make sure your backend server is running!**

Terminal 1-> cd .. (to root folder)-> >>npm start (on port 5000)

**Open a new terminal and run the front end dev server :**

Terminal 2-> cd .. (to root folder)->>>cd ./frontend

Add **axios** to your project:

its an http requests library to make requests to the server.

You may also use the fetch API

Terminal 2-> >>cd ./frontend -> npm i axios

>> npm start to launch

HomeScreen.js🡺

**Although the products will soon be handled by redux as app level.**

**For now** the products will be a **local state** in the home screen page using **the useState HOOK**!

**useEffect** runs as soon as the component load=> will enable us to request the products from the server via fetch! (or axios, in our case)

syntax : useEffect( ()=>{…}, [dependencied])

dependencied- variables that when changed, will cause the useEffect to run again.

\*notice we didnt set a proxy yet, but wait…

import React, { useState, useEffect } from 'react'

import { Row, Col } from 'react-bootstrap'

import axios from 'axios'

// import products from '../products' at first we brought the data from a js file.

import ProductCard from '../components/ProductCard'

const HomeScreen = () => {

  const [products, setProducts] = useState([]) // initial state is an empty array, because the product list is an array of objects (products).

  useEffect(() => {

    //will run as soon as this component mounts!

    const fetchProductsServer = async () => {

      //the data has been sent as json from  server.js (app.get...)

      const response = await axios.get('/api/products')

      const data = await response.data //no need to use .json() method to pase the data , its already parsed by axios.

      console.log(response)

      console.log(data)

      setProducts(data) //connecting the state to data ==> products will be  equal to data

    }

    fetchProductsServer() //calling the function to fetch the products!

  }, [])

  return (

    <>

      <h1> Our Products </h1>

      <Row>

        {products.map((product) => (

          <Col sm={12} md={5} lg={4} xl={3} key={product.\_id}>

            <ProductCard product={product} />

          </Col>

        ))}

      </Row>

    </>

  )

}

export default HomeScreen

Set a proxy:

When you will will try to run the code you will get an error in the console. the GET methot you ran on the server when you tried to fetch the data === axios.get('/api/products') === failed!

server.js

app.get('/api/products', (req, res) => {

  //lets respond with a product information see backend->data-> products.js

  res.json(products) //.json will sent the data as a JSON format!

})

It’s currently looking for the data in the frontend dev server and not the express server we set on port 5000!

Frontend/Package.json.json🡺

\*notice on deploy it will change!

{

  "name": "frontend",

  "proxy": "http://127.0.0.1:5000", //just write it beneath name …

  "version": "0.1.0",

  "private": true,

  "dependencies": {

    "@testing-library/jest-dom": "^4.2.4",

    "@testing-library/react": "^9.5.0",

    "@testing-library/user-event": "^7.2.1",

    "axios": "^0.20.0",

    "react": "^16.13.1",

    "react-bootstrap": "^1.3.0",

    "react-dom": "^16.13.1",

    "react-draggable": "^4.4.3",

    "react-resizable": "^1.11.0",

    "react-router-bootstrap": "^0.25.0",

    "react-router-dom": "^5.2.0",

    "react-scripts": "3.4.3"

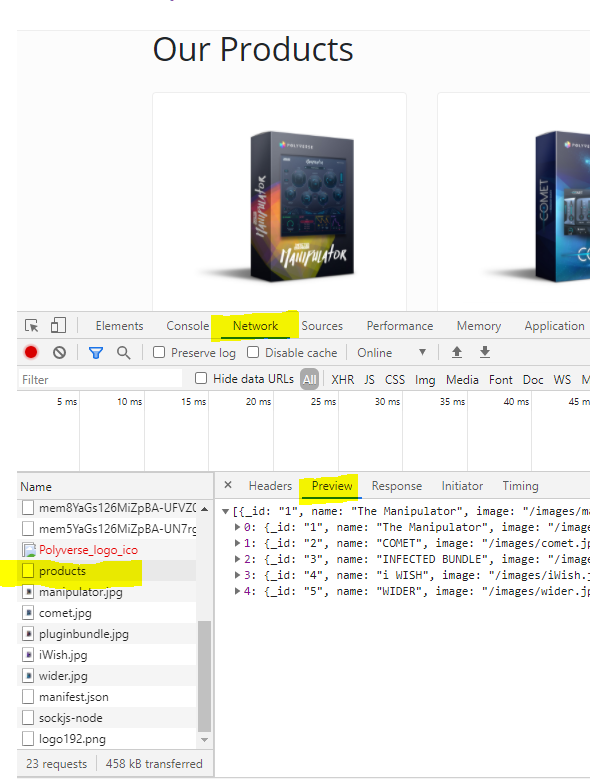
  }, //there is more data..

Re run the frontend wed dev server :

>> npm start

And products should be loaded from the beckend server.

In the browser check the network tab:



Now lets do the same process in the product page:

Server.js 🡺 remember ?

app.get('/api/products/:id', (req, res) => {

  //now lets serve a specific product data by url param, user id!

  const product = products.find((element) => element.\_id === req.params.id)  // matching the user in the DB to the one being asked in the url parameter.

  res.json(product) //.json will sent the data as a JSON format!

producstScreen.js🡺

import React, { useState, useEffect } from 'react'

import { Link } from 'react-router-dom'

import { Row, Col, Image, ListGroup, Card, Button } from 'react-bootstrap'

import Ratings from '../components/Ratings'

//import products from '../products' //its not a react component but a js variable

import YouTubePlayer from '../components/YouTubePlayer'

import axios from 'axios'

const ProductScreen = (props) => {

  //we use props.match!

  const [product, setProduct] = useState({}) // initial state is an empty object, because product is an object.

  useEffect(() => {

    const fetchProdFromDB = async () => {

      //props.match.params.id allow us to get the params passed in the frontend's url !

      const response = await axios.get(`/api/products/${props.match.params.id}`)

      const data = await response.data

      console.log(data)

      setProduct(data) //setting the local state

    }

    fetchProdFromDB()

  }, [])

**That it! Now the product page should run while fetching the product data from the server based on id in the url.**

Install Nodemon & concurrently & running client server (frontend).

Install As dev dependencies.

**Nodemon- restarting automatically the backend server on update.**

**At root folder:**

**>> npm i -D nodemon concurrently**

**Now lets write a script for initializing nodemon:**

**At root folder package.json. json 🡺**

 "scripts": {

    "start": "node backend/server",

    "server": "nodemon backend/server",

    "client": "npm start --prefix frontend",

    "dev": " concurrently \"npm run server\" \"npm run client\""

  },

**"client": "npm start --prefix frontend" 🡺 frontend is the name of the folder that will run.**

**When using >> npm run client .**

**Meaning we can run the frontend dev server now from the root folder.**

**Also : >>npm run server**

**Meaning we can run the backend dev server now from the root folder and it will refresh automatically when backend changes.**

**>>npm run dev //is a concurrently feature which allow us to run both simultaneously**

Environment variables

One of the famous environment variables is the process.env.PORT (also API keys ):

Right now we are using port 5000 manually.

But when we will connect a DB with our code and deploy it we will have to get it using the

process.env.PORT method.

\*\*API keys – whether a paid or free one it is very important as a security measure to set them as ENV variables.!

\*\*secret tokens as well (we’ll get to that later).

Some environment variables are changing from one operating system to another browsers etc.…

dotenv - an npm package that loads environment variables from a .env file into [process.env](https://nodejs.org/docs/latest/api/process.html#process_process_env). Storing configuration in the environment separate from code is based on [The Twelve-Factor App](http://12factor.net/config) methodology.

Lets install it:

At root level-

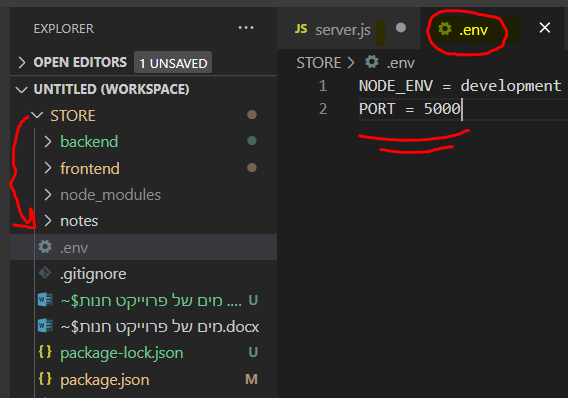
> npm i dotenv

CREATE A NEW .env FILE AT ROOT LEVEL:

IN IT SET TWO ENV VARIABLES, AT THIS EXACT SYNTAX!

ROOT-> .env 🡺

\*\*make sure it’s in the .gitignore file as well!!!



\*NODE\_ENV will soon be interactive, it could be also be production mode

Backend/server.js 🡺

const express = require('express')

const dotenv = require('dotenv') //

const products = require('./data/products')

const app = express()

dotenv.config() //

const PORT = process.env.PORT || 5000

const mode = process.env.NODE\_ENV

{ . . .}

app.listen(

  PORT,

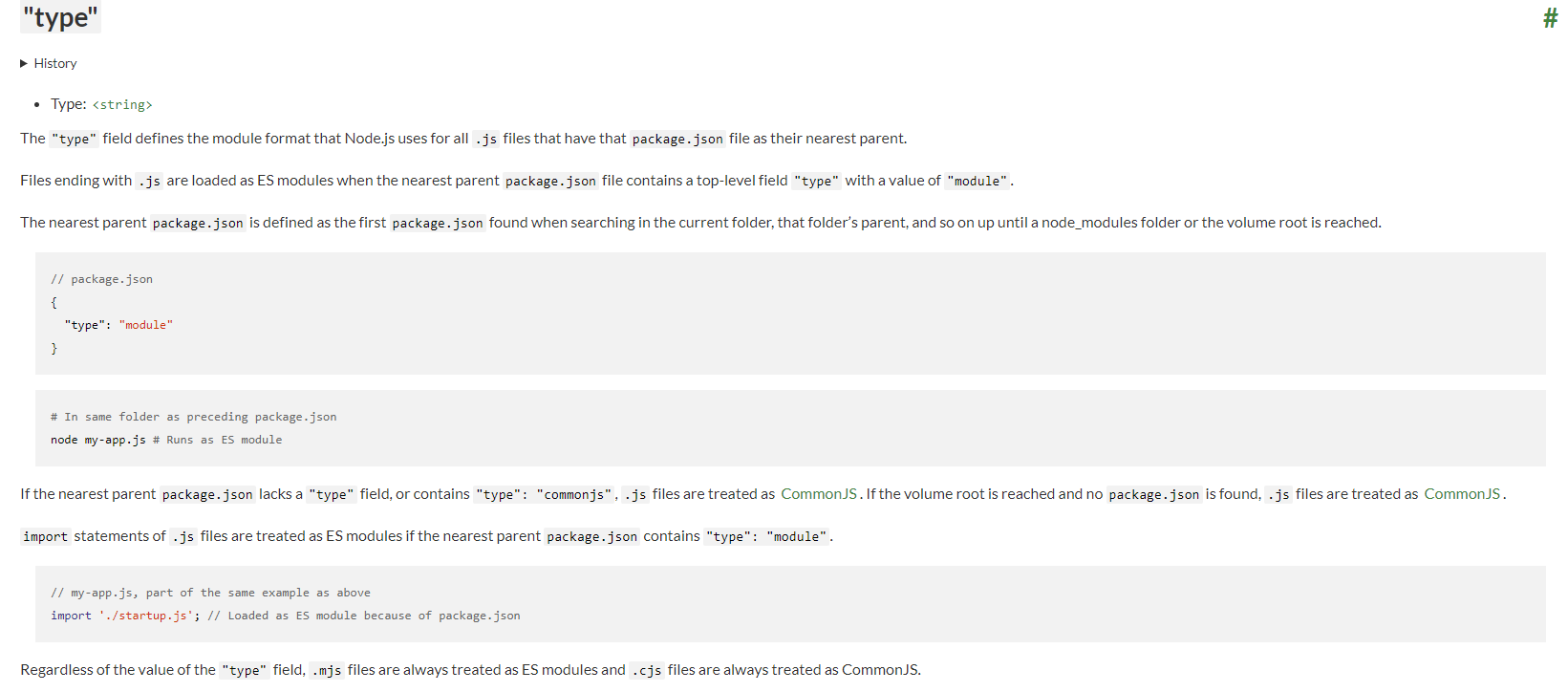
  console.log(`Hey! server is running in ${mode} mode on port: ${PORT}`)

)

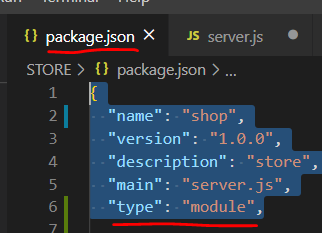
Changing backend import expression to ES modules

Check node.JS documentation: (you should have at least V14.3)

<https://nodejs.org/dist/latest-v14.x/docs/api/packages.html#packages_type>



package.json on root level (backend)



\*The tradeoff of using the ES module syntax is that you must mention the file’s type : **.js**

Backend->data->product.js🡺

Now we must change the export syntax…

export default products

connecting the project to a Database (DB)

**mongoDB: no sql DB**

**Collection and documents:**

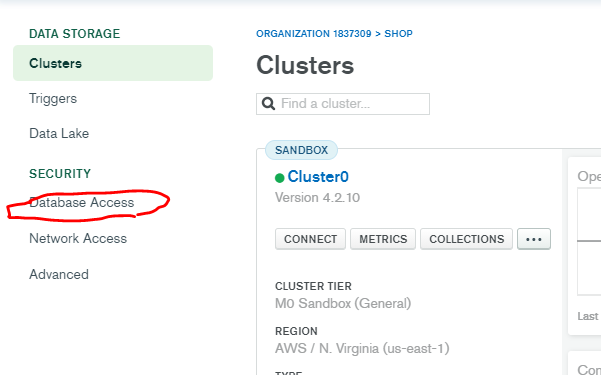
**Each document is basically a json file – format**

**For instance: products == collection , product X == document.**

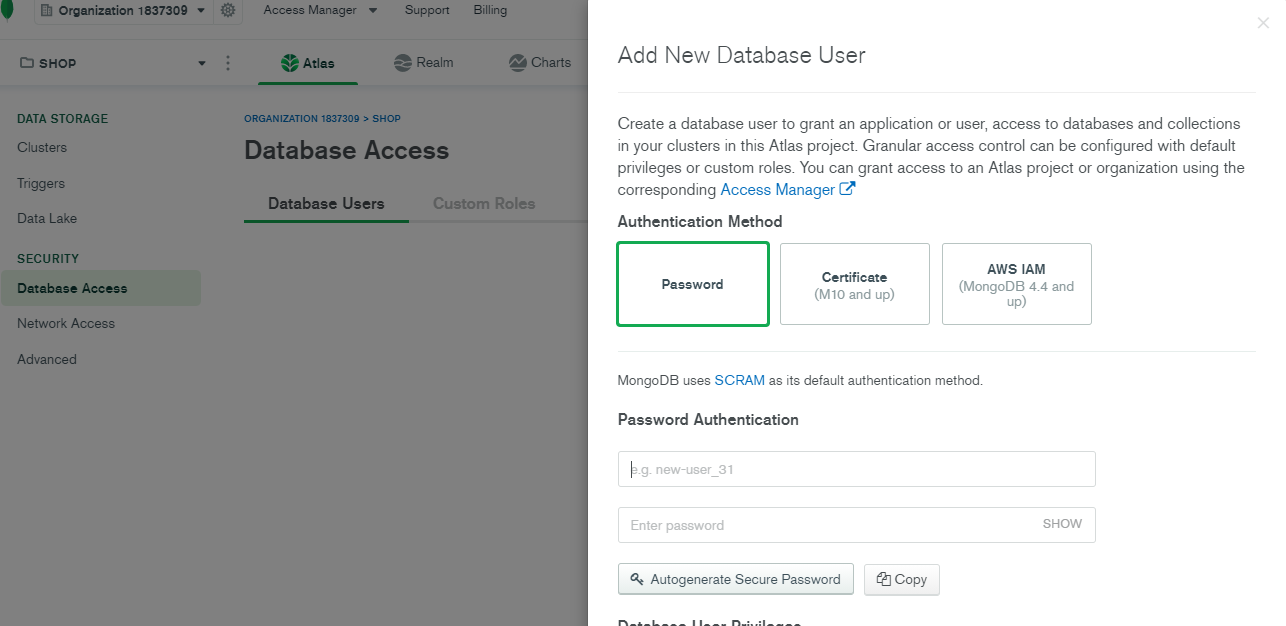
**Users==collection, user X == document.**

**Orders… and so on..**

* **Compass- desktop (Graphical User Interface) GUI for interacting with the data IN THE db. DOWNLOAD IT AND INSTALL ON YOUR PC.**
* **Atlas- cloud DB. GO TO🡺 cloud.mongodb.com, AND START A NEW PROJECT / CLUSTER. FOR FREE!**

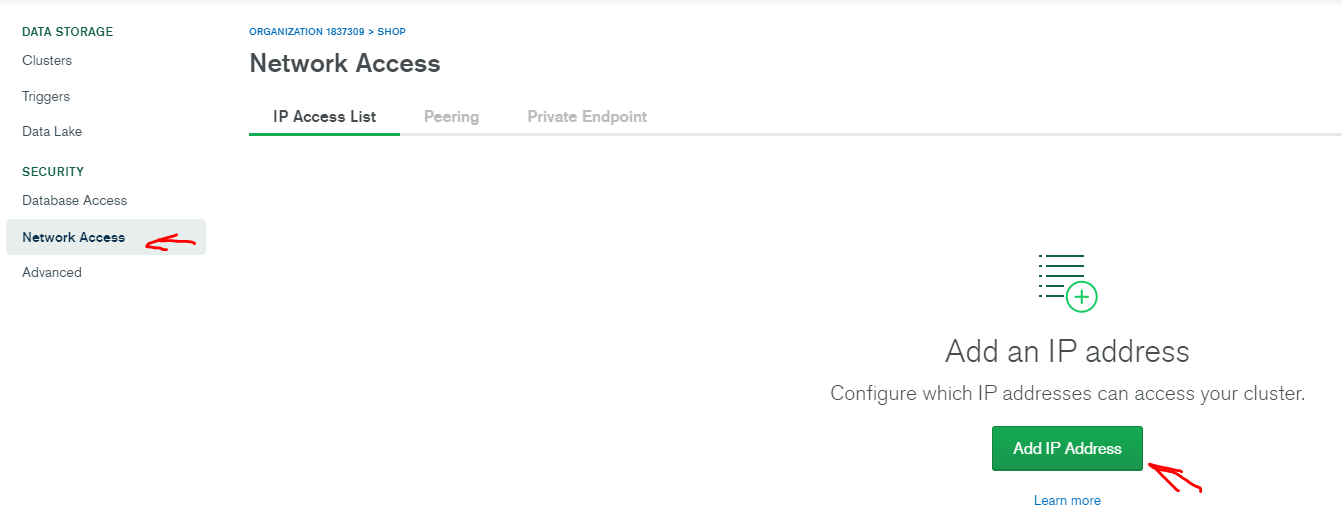
**Security measures in Atlas before connecting the two:**

**Add a new DB user:**

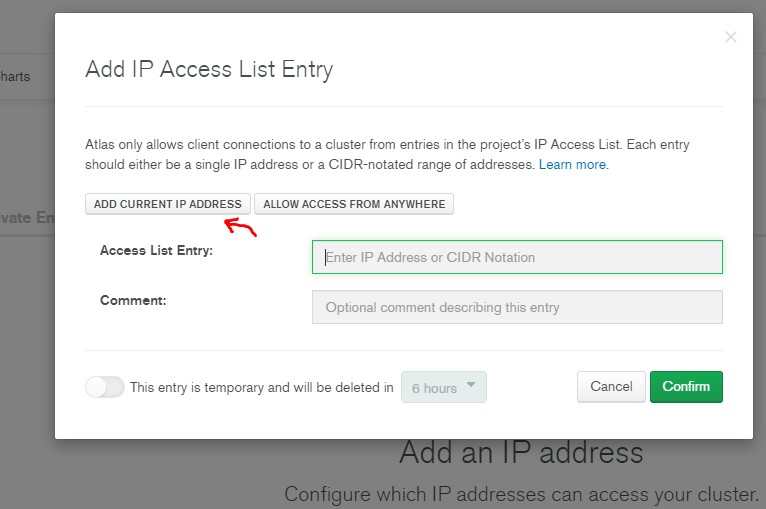


**Choose user name and password.**

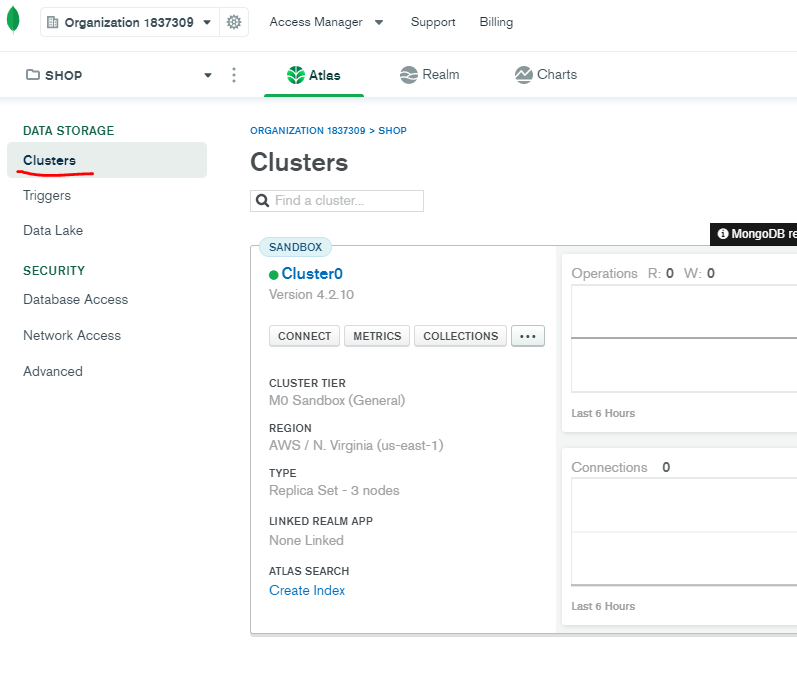
**network access**



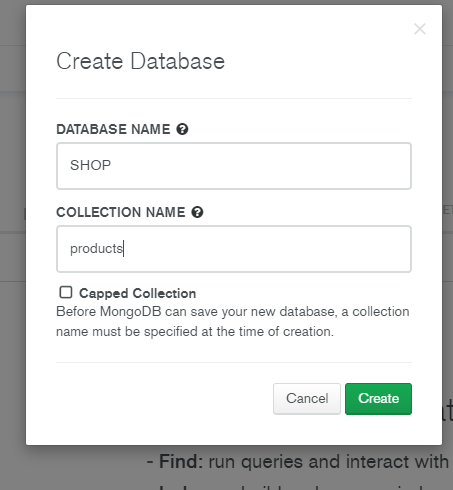
**Add your current IP address**



**Go back to clusters**

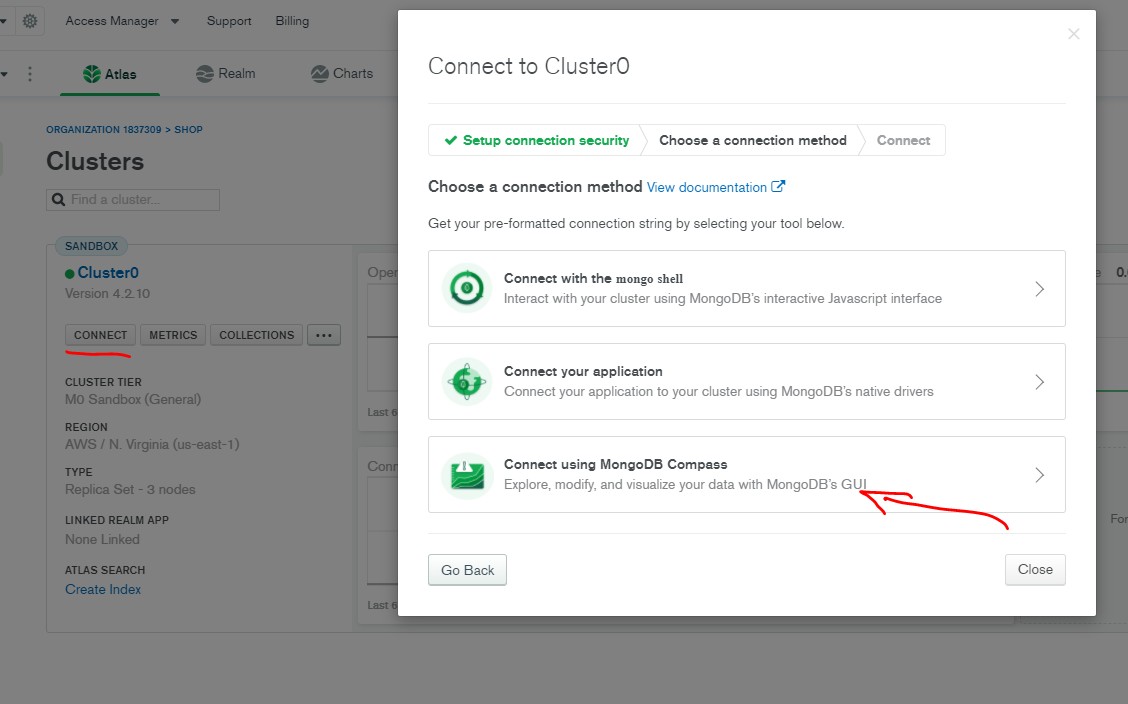


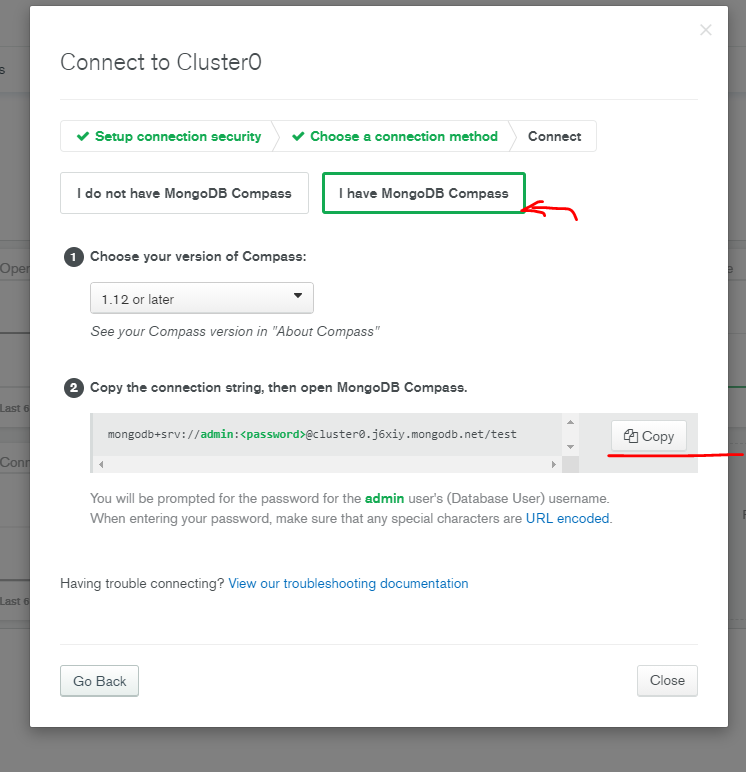
**Click on: collections -> click on: Add my own database->  
choose your DB name and a collection name your app will have (just one for now)**



**Now go back to the clusters main page and click Connect!**

**Then click connect to compass (later we will click connect to application)**

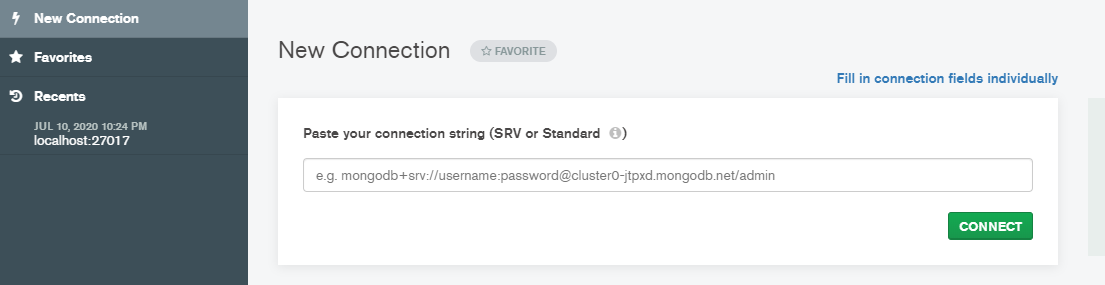




**Copy the connection string**

**Connect compass and alas: we will use a connection string we just copied**

**Compass:**

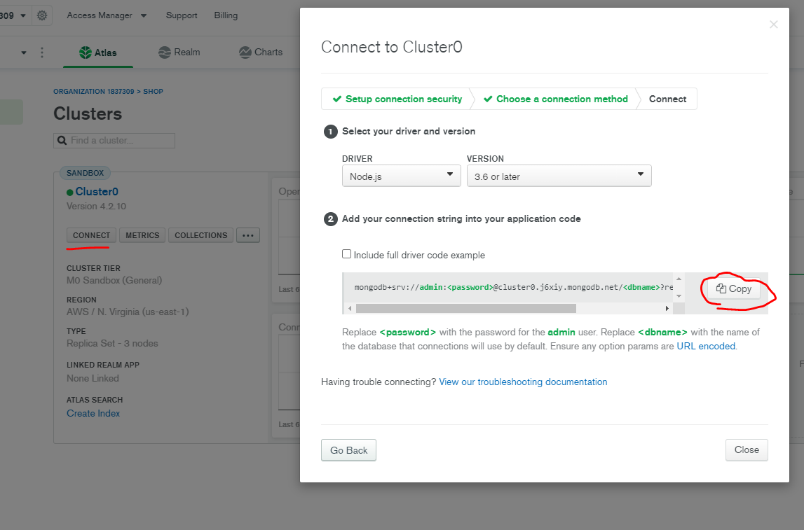




**Make sure you add your password and dich the <>, instead of test write the name of your DB name.**

**Click connect, and that’s it! The two are paired!**

**Now we need to connect the app itself to the DB:**



**The string we just copied will be added to the .env file under MONGO\_URI:**

**MONGO\_URI = mongodb+srv://admin:\*\*\*password here\*\*\*@cluster0.j6xiy.mongodb.net/\*\*DB NAME HERE\*\*?retryWrites=true&w=majority**

**the next step is to use mongoose- an object data modelling for Node.js that allows us to interact with our DB, create & find data.**

**Terminal->**

**Root folder level:**

**>> npm I mongoose**

**Backend-> create a new folder: config -> in it create a file: db.js 🡺**

**This is the file which serve as a database connection. It will look like this:**

import mongoose from 'mongoose';

const connectDB =  async ()=>{

    try{

        const conn = await mongoose.connect(process.env.MONGO\_URI,{

            //must add in order to not get any error masseges:

            useUnifiedTopology:true,

            useNewUrlParser: true,

            useCreateIndex: true

        })

        console.log(`mongo database is connected!!! ${conn.connection.host} `)

    }catch(error){

        console.error(`Error: ${error} `)

        process.exit(1) //passing 1 - will exit the proccess with error

    }

}

export default connectDB

**backend-> server.js 🡺 add two lines**

import express from 'express'

import dotenv from 'dotenv'

import connectDB from './config/db.js' // DB connection

import products from './data/products.js'

dotenv.config()

const PORT = process.env.PORT || 5000

const mode = process.env.NODE\_ENV

const app = express()

connectDB() //this function connects us to the DB!!!

. . . more code…

**Important note!!! Make sure connectDB() is after dotenv.config!!!**

**that’s it! We are connected to the DB.**

**You can check it by root folder terminal**

**>> npm run server**

**Colorize the termina outputs:**

**(\*Not a must – before you run the server install >>npm i colors 🡺 will color the terminal outputs)**

<https://www.npmjs.com/package/colors>

console.log('hello'.green); *// outputs green text*

console.log('i like cake and pies'.underline.red) *// outputs red underlined text*

console.log('inverse the color'.inverse); *// inverses the color*

console.log('OMG Rainbows!'.rainbow); *// rainbow*

console.log('Run the trap'.trap); *// Drops the bass*

**you may add it in db.js & server.js :**

**server.js🡺**

app.listen(

  PORT,

  console.log(`Hey! server is running in ${mode} mode on port: ${PORT}`.yellow.bold)

)

**Terminal🡺**

**Hey! server is running in development mode on port: 5000**

**Data modeling**

Backend-> create a new folder: **models🡺**

**We will create 3 models : userModel.js , productModel.js , orderModels.js**

each file will contain a schema: the structure of each object in the data base.

And a MODEL- constructed from the schema and a name of model.

The model variable name is what being exported from each file.

import mongoose from 'mongoose';

const userSchema = mongoose.Schema({

    name:{

        type: String,

        required: true

    },

    email:{

        type: String,

        required: true,

        unique: true

    },

    password:{

        type: String,

        required: true

    },

    isAdmin:{

        type: Boolean,

        required: true,

        default: false

    }

},{

    //mongoose built in time stamps for "created at"/ "updated at"

    timestamps: true

})

const User = mongoose.model('User', userSchema) //model is constructed from the schema and the name ‘User’

export default User

Product schema contains also the review schema.

import mongoose from 'mongoose';

const reviewSchema = mongoose.Schema({ //a single review schema may also be in its own file ...

    name:{

        type: String,

        required: true

    },

    rating:{

        type: Number,

        required: true

    },

    comment:{

        type: String,

        required: true

    },

},{ //mongoose built in time stamps for "created at"/ "updated at"

timestamps: true })

const productSchema = mongoose.Schema({

    user:{ //who created the product?

        type: mongoose.Schema.Types.ObjectId,

        required: true,

        ref: 'User' //will create relations between the productModel and userModel (the one who added the product)

    },

    name:{

        type: String,

        required: true

    },

    image:{

        type: String,

        required: true,

    },

    brand:{

        type: String,

        required: true

    },

    category:{

        type: String,

        required: true

    },

    description:{

        type: String,

        required: true

    },

    rating:{

        type: Number,

        required: true,

        default:0

    },

    reviews:[

        reviewSchema

    ],

    numReviews:{

        type: Number,

        required: true,

        default:0

    },

    price:{

        type: Number,

        required: true,

        default: 150

    },

    inStock:{

        type: Boolean,

        required: true,

        default: true

    },

},{

    //mongoose built in time stamps for "created at"/ "updated at"

    timestamps: true

})

const Product = mongoose.model('Product', productSchema)

export default Product

import mongoose from 'mongoose';

const orderSchema = mongoose.Schema({

    user:{ //who created the product? //will create relations between the productModel and userModel-> (the one who buys the product)

        type: mongoose.Schema.Types.ObjectId,

        required: true,

        ref: 'User'

    },

    orderItems:[{

        name:{type: String, required:true},

        quantity:{type: Number, required:true},

        image:{type: String, required:true},

        price:{type: Number, required:true},

        product:{//will have relations with the product schema

            type: mongoose.Schema.Types.ObjectId,

            required: true,

            ref: 'Product'

        }

    }],

    shippingAddress:{

        adress:{ type: String, required:true },

        city:{ type: String, required:true },

        postalCode:{ type: String, required:true },

        country:{ type: String, required:true },

    },

    paymentMethod:{

        type: String,

        required: true,

    },

    paymentResult:{ //will be recieved from paypal...

        id: {type: String},

        status: {type: String},

        update\_time: {type: String},

        email\_address: {type: String},

    },

    taxPrice:{

        type: Number,

        required: true,

        default: 0.0

    },

    shippingPrice:{

        type: Number,

        required: true,

        default: 0.0

    },

    totalPrice:{

        type: Number,

        required: true,

        default: 0.0

    },

    isPaid:{

        type: Boolean,

        required: true,

        default: false

    },

    paidAt:{

        type: Date

    },

    isDelivered:{

        type: Boolean,

        required: true,

        default: false

    },

    deliveredAt:{

        type: Date

    },

},{

    //mongoose built in time stamps for "created at"/ "updated at"

    timestamps: true

})

const Order = mongoose.model('Order', orderSchema)

export default Order

adding product data to database

backend->data->products.js 🡺

each product has an \_id: property , we need to delete it because when adding data to monogo it’s automatically generates an id.

backend->data-> add a new file: users.js🡺

in it we will create 3 user ,one of them is the admin:

const users = [

    {

        name: 'Admin',

        email:'orisouchami1@gmail.com',

        password: xxxx, //just a placeholder for now, we need to hash it in

 the DB.

        isAdmin: true

    },

    {

        name: 'joe',

        email:'joe@gmail.com',

        password: xxxx,

    },

    {

        name: 'john',

        email:'john@gmail.com',

        password: xxxx,

    }

]

In order to encrypt we will install bcryptjs:

Terminal at root level:

>>npm i bcryptjs

import bcrypt from 'bcryptjs'

const users = [

    {

        name: 'Admin',

        email:'orisouchami1@gmail.com',

        password: bcrypt.hashSync('Shopadminori1',10), //hash the pasword you entered when you set the admin user in mongo

        isAdmin: true

    },

    {

        name: 'joe',

        email:'joe@gmail.com',

        password: bcrypt.hashSync('123456',10),

    },

    {

        name: 'john',

        email:'john@gmail.com',

        password: bcrypt.hashSync('123456',10),

    }

]

export default users

Database Seeder:

Seeding a database is a process in which an initial set of data is provided to a database when it is being installed. It is especially useful when we want to populate the database with data we want to develop in future.

Backend-> create a new file: seeder.js🡺

A separate script we will run in order to import data to DB (or destroy)

import mongoose from 'mongoose'

import dotenv from 'dotenv'

import colors from 'colors'

import users from './data/users.js'

import products from './data/products.js'

import User from './models/userModel.js'

import Product from './models/productModel.js'

import Order from './models/orderModel.js'

import connectDB from './config/db.js'

dotenv.config()

connectDB()

const importData = async ()=>{ //from the database therefore its asynchronous

    try{

        //we first want to clear all our collections (user, order , product) from data it might have and prepare it to recieve fresh data.

       await Order.deleteMany()

       await Product.deleteMany()

       await User.deleteMany()

       const createdUsers = await User.insertMany(users) //will be an array

       const adminUser = createdUsers[0].\_id // in users.js the first user in the array of users is the admin. we will populate this variable with the admin id!

       const sampleProducts = products.map(product=>{ //we will populate the "user" field in each product with the admin id, meaning he was the one who created it!

           return{...product, user: adminUser}

       })

       await Product.insertMany(sampleProducts) //populate the DB with products containing the admin's id

       console.log('Data imported'.cyan)

       proccess.exit()

    } catch(error){

        console.error(`${error} -Data could not be imported`.red)

        proccess.exit(1) //1=> exit with failure

    }

}

const destroyData = async ()=>{ //from the database therefore its asynchronous

    try{

        //we first want to clear all our collections (user, order , product) from data it might have and prepare it to recieve fresh data.

       await Order.deleteMany()

       await Product.deleteMany()

       await User.deleteMany()

       console.log('All Data Deleted'.orange)

       proccess.exit()

    } catch(error){

        console.error('Data could not be deleted'.red)

        proccess.exit(1) //1=> exit with failure

    }

}

if(process.argv[2] === ‘-d’){ // argv will be an array, if we will pass -d it will populate the third element of the array

    destroyData()

}else{

    importData()

}

In order to run this script via terminal : root level->

>>node backend/seeder (for importing data to DB)

>>node backend/seeder -d (for destroy)

We can also add a script to package.json, under scripts:

"data:import": "node backend/seeder",

    "data:delete": "node backend/seeder -d"

**After you start filling orders and add users don’t use these operations!!**

Fetch data from mongoDB:

Before we get to that lets organize server.js and create routes using express Router.

Cut these lines from server.js 🡺

app.get('/api/products', (req, res) => {

  //lets respond with a product information see backend->data-> products.js

  res.json(products) //.json will sent the data as a JSON format!

})

app.get('/api/products/:id', (req, res) => {

  //now lets serve a specific product data by url param, user id!

  const product = products.find((element) => element.\_id === req.params.id) // matching the user in the DB to the one being asked in the url parameter.

  res.json(product) //.json will sent the data as a JSON format!

})

Backend-> create new folder: routes -> create new file: productRoutes.js🡺

1. And paste theme here! Just don’t forget to change from **app**.xyz -> to **router**.xyz
2. When using router for products we may eliminate the url part of : ‘/api/products’ because we will refer it to this base url from server.js

import express from 'express'

const router = express.Router()

router.get('/', (req, res) => {

    //lets respond with a product information see backend->data-> products.js

    res.json(products) //.json will sent the data as a JSON format!

  })

  router.get('/:id', (req, res) => {

    //now lets serve a specific product data by url param, user id!

    const product = products.find((element) => element.\_id === req.params.id) // matching the user in the DB to the one being asked in the url parameter.

    res.json(product) //.json will sent the data as a JSON format!

  })

  export default router

back to server.js 🡺

add these two lines:

import productRoutes from './routes/productsRoutes.js' //import the routes.

app.use('/api/products', productRoutes) //connect the product url to the

router & routes

it will look like that:

import express from 'express'

import dotenv from 'dotenv'

import colors from 'colors'

import connectDB from './config/db.js' // DB connection

import productRoutes from './routes/productsRoutes.js' //import the routes.

dotenv.config()

const PORT = process.env.PORT || 5000

const mode = process.env.NODE\_ENV

const app = express()

connectDB() //this function connects us to the DB!!! it must be after dotenv.config

app.get('/', (req, res) => {

  res.send('this is what you GET when you request this path (/)')

})

app.use('/api/products', productRoutes) //connect the product url to the router

app.listen(

  PORT,

  console.log(`Hey! server is running in ${mode} mode on port: ${PORT}`.yellow.bold)

)

Backend-> routes -> productRoutes.js🡺

import express from 'express'

const router = express.Router()

import Product from '../models/productModel.js'//lets import the product model

router.get('/', async (req, res) => {

    const products = await Product.find({})

//passing empty object to find method will give us all the elements(products) as a promise!!!.

therefore, we must use async await!

    res.json(products) //.json will sent the data as a JSON format!

  })

  export default router

* When using async await it’s recommended to also use some sort of error handling such as try{}catch(){}. In our case we will use a third party middleware:

An npm package “express-async-handler”

Terminal at root level:

>> npm i express-async-handler

The file will look like that :

import express from 'express'

import asyncErrorhandler from 'express-async-handler'

//an npm pack for handling errors instead of using try catch we will have to wrap the whole function with it.

const router = express.Router()// api/products/...

import Product from '../models/productModel.js'//lets import the po

//fetch all products from DB

router.get('/', asyncErrorhandler( async (req, res) => {

    const products = await Product.find({}) //passing empty object will give us all the elements(products) as a promise!!!.

    res.json(products) //.json will sent the data as a JSON format!

  }))

  //fetch single  product by id from DB

  router.get('/:id', asyncErrorhandler(async (req, res) => {

    //now lets serve a specific product data by url param, user id!

    const product = await Product.findById(req.params.id)

// matching the user in the DB to the one being asked in the url parameter.

    if(product){

        res.json(product) //.json will sent the data as a JSON format!

    }else{

        res.status(404).json({message:'Product not found'})

    }

  }))

  export default router

that’s it !

run the server:

>> npm run server

Go to the browser :

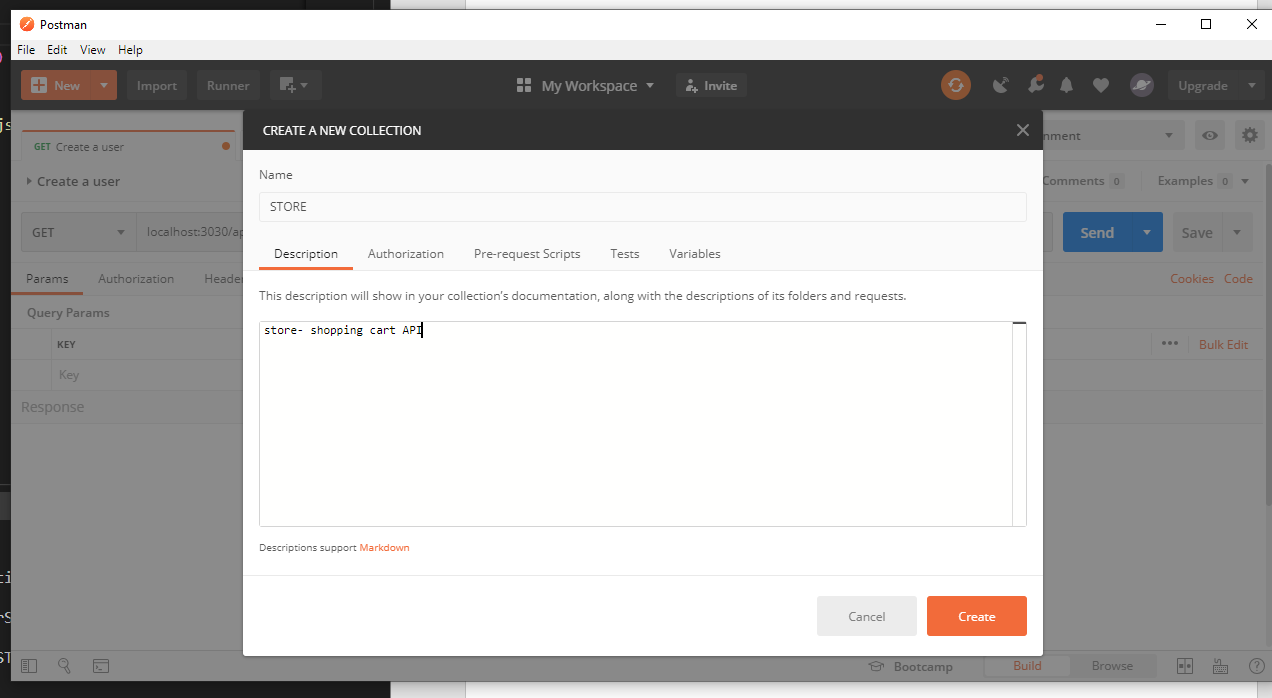
<http://localhost:5000/api/products/> 🡺 you’ll see a json with all users

<http://localhost:5000/api/products/someuser>id 🡺 you’ll see a specific user json but!

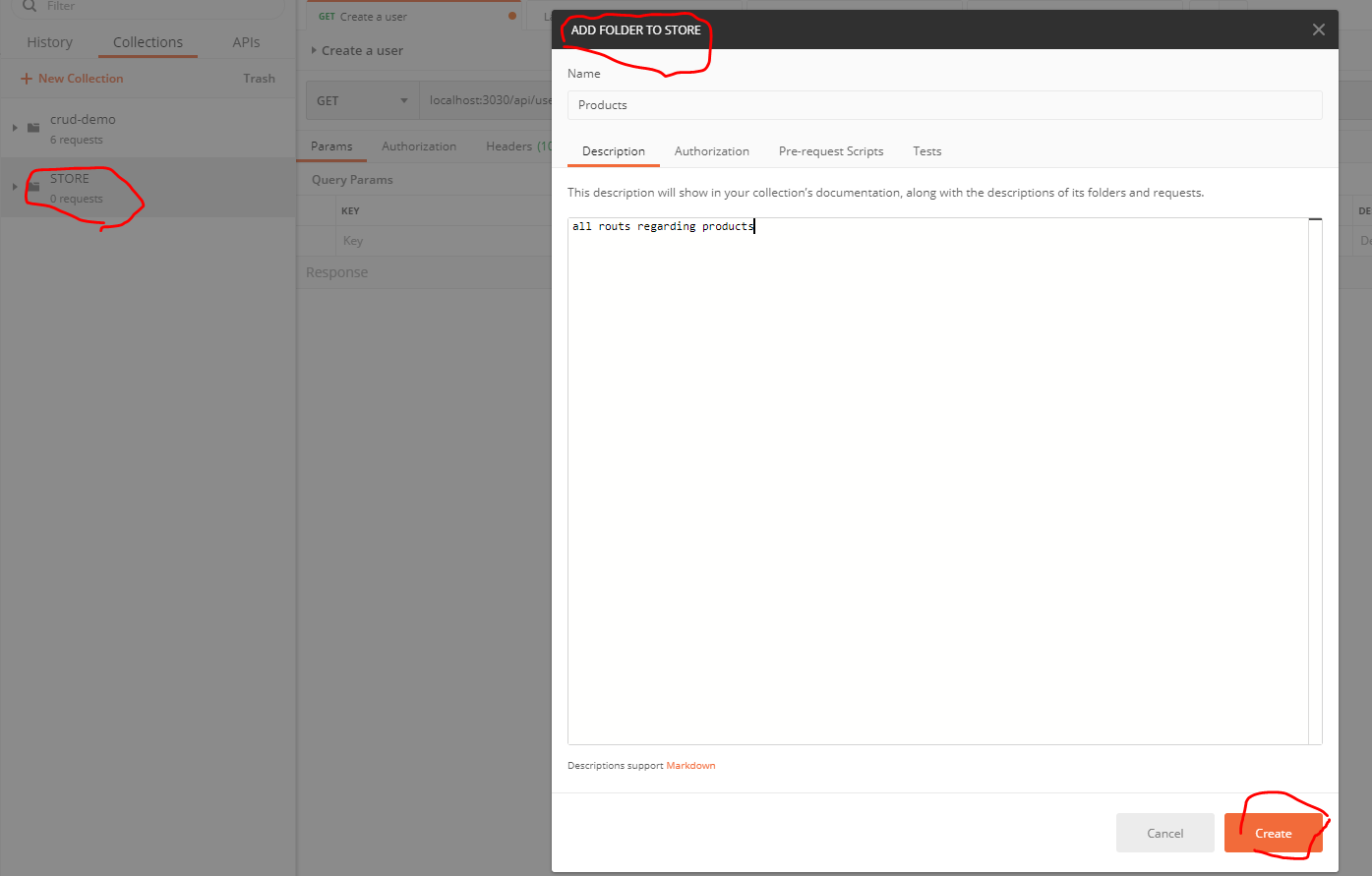
if you provide an id with the same length but one that does not exists, oure 404 error will be shown!

Setting up Postman:

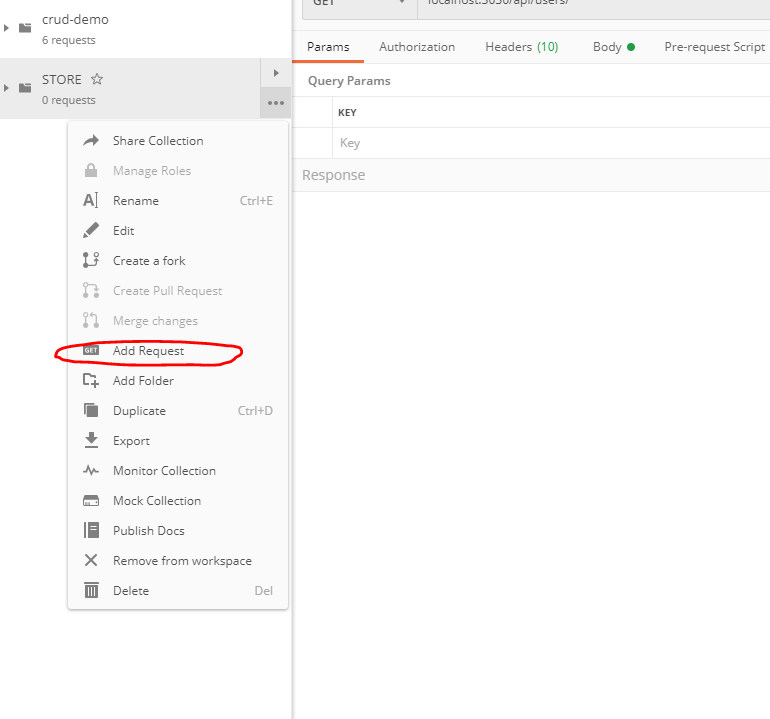
1. Open postman
2. Create new Collection: STORE

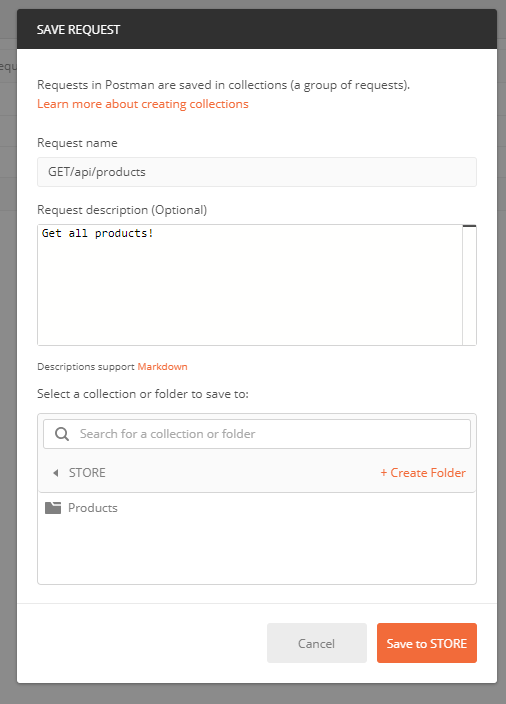


1. Open a new folder in store collection:

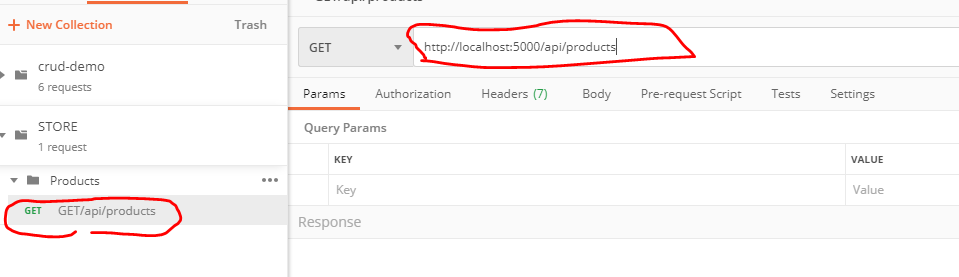


1. Add a new request: GET all products





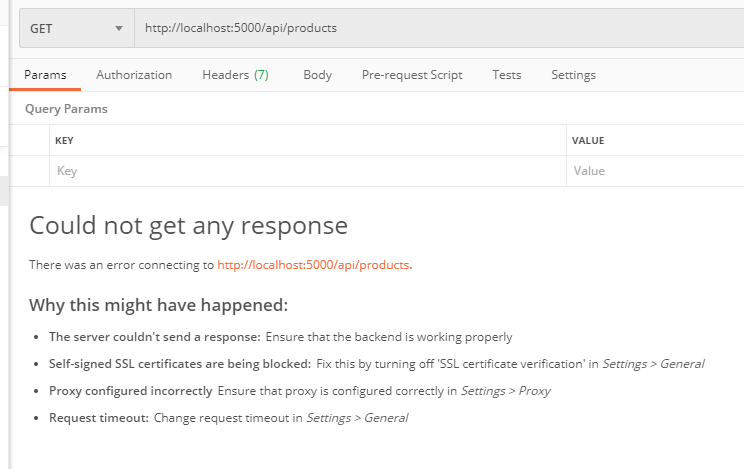
Save it in products folder.

1. Write the **path** to the get request and predd send to get all users ! make sure you run “npm run server” in the terminal

\*\*\*troubleshooting session:

When trying to send the get request I stumbled across those messages:

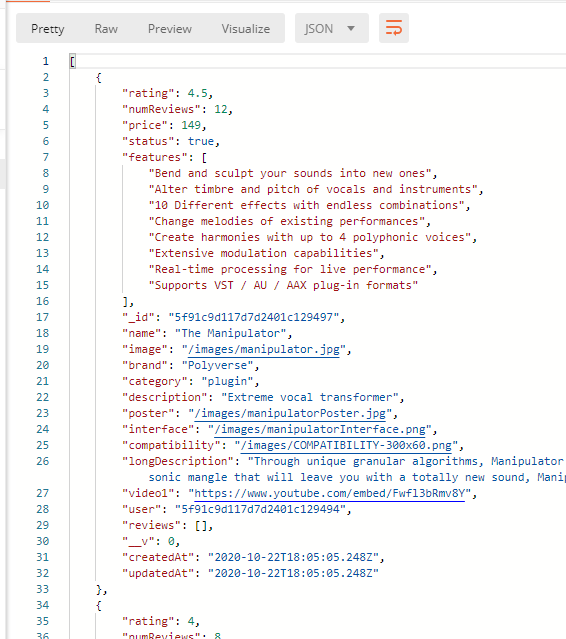




Apparently, the day before, I had an electricity issue which made my router re start, my ip address chanced and therefore I couldn’t connect to the mongoDB Atlas service, I had to update the ip address there and then re connect:



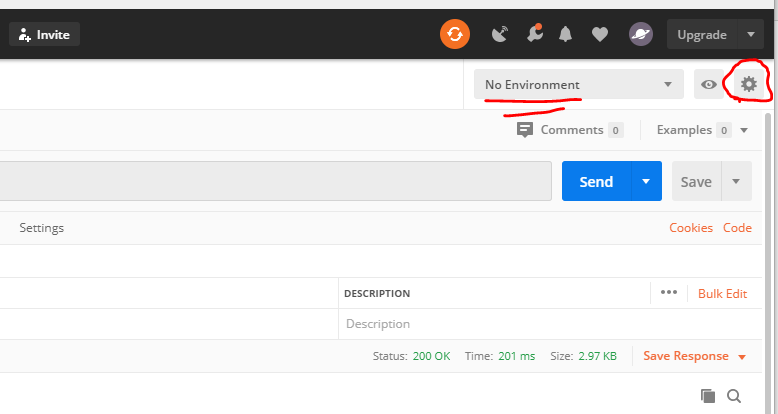
**And vuala you got the products response!!!**

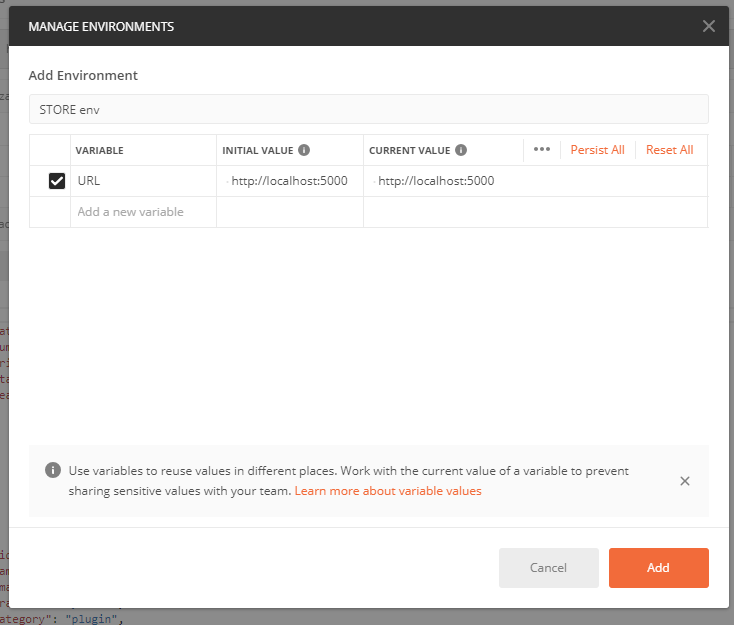


**Creating an environment variable in postman:**

**In order to skip the part of writing the URL path including the** <http://localhost:5000/api/products>

We will create an environment variable in postman: click settings 🡪ADD🡪fill🡪and add agin



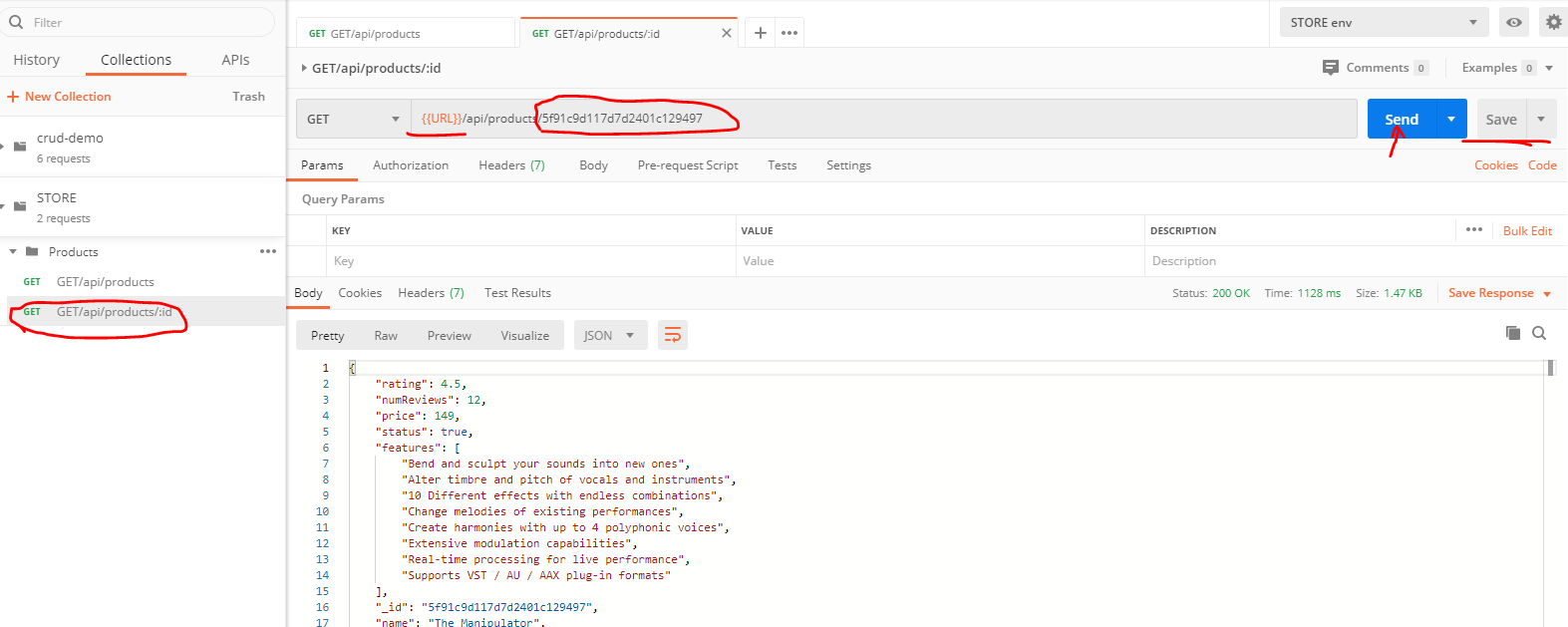




Change the environment 🡪 then write {{URL}}/api/products 🡪 save and send!

Later when you deploy and have a domain name you can change the URL accordingly so you can use postman this way also.

**Create a new GET request for product/:id**



Error handling with custom middleware

in order to receive a custom error message when requesting for a product that do not exists in the DB (api/products/:id) we will use a custom middleware

**http://expressjs.com/en/guide/error-handling.html**

**middleware**- a function that has access to the request response cycle,  
you must use next in order to move to the next piece of middleware (unless you want to stop the cycle).

Backend🡪server.js🡺 we will add this code just above the app.listen and at the end of all the other routes handling in tha app, therefore if **none** of them resolved or one of them throws an error the next code will be fired off :

// 404 error creator:

app.use((req, res, next)=>{ //no specified rout meaning all server requests  will pass through this code!  if the code above was not resolved

  const error = new Error(`Not Found ${req.originalUrl}`) //req.originalUrl=> is the url the user entered

  res.status(404)

  next(error)

  })

//error handling middleware:

app.use((err, req, res, next)=>{ //this code will be fired off only when error object exists in the app.

  //err- catches errors thrown from anyware in our server or errors from the

DB

  console.log('error middleware on')

  //sometimes even errors could have a statuscode of 200 so we need to change them to the 500 server error relm

  //if it's not 200 it will have it's original status code.

  const ststusCode = res.statusCode === 200 ? 500 : res.statusCode

  res.status(ststusCode)

  res.json({

    message: err.message,

    stack: process.env.NODE\_ENV === 'production' ? null : err.stack, //the stack of the error object is it's explanation (we will show it only in dev)

  })

  next()

  })

app.listen(

  PORT,

  console.log(`Hey! server is running in ${mode} mode on port: ${PORT}`.yellow.bold)

)

Now let’s move the error handling code to different files (so the server file will be cleaner):

Backend🡪create a folder: moddleware 🡪 create a new file: errorMiddleware.js 🡺

// 404 error creator:

const notFoundError = (req, res, next)=>{ //no specified rout meaning all server requests will pass through this code! if the code above was not resolved

    console.log('404 middleware on')

    const error = new Error(`Not Found ${req.originalUrl}`) //req.originalUrl=> is the url the user entered

    res.status(404)

    next(error)

    }

  //error handling middleware:

  const allErrorsHandler =(err, req, res, next)=>{ //no specified rout meaning all server requests will pass through this code!was not resolved

    //err- catches errors thrown from anyware in our server or errors from the DB

    console.log('error middleware on')

    //sometimes even errors could have a statuscode of 200 so we need to change them to the 500 server error relm

    //if it's not 200 it will still have it's status code.

    const ststusCode = res.statusCode === 200 ? 500 : res.statusCode

    res.status(ststusCode)

    res.json({

      message: err.message,

      stack: process.env.NODE\_ENV === 'production' ? null : err.stack, //the stack of the error object is it's explanation (we will show it only in dev)

    })

    next()

    }

    export{notFoundError, allErrorsHandler}

**while server.js will now look like that with the error handling file being imported:**

import express from 'express'

import dotenv from 'dotenv'

import colors from 'colors'

import {notFoundError, allErrorsHandler} from './middleware/errorMiddleware.js'

import connectDB from './config/db.js' // DB connection

// import products from './data/products.js'

import productRoutes from './routes/productsRoutes.js' //import the routes.

dotenv.config()

const PORT = process.env.PORT || 5000

const mode = process.env.NODE\_ENV

connectDB() //this function connects us to the DB!!! it must be after dotenv.config

const app = express()

app.get('/', (req, res) => {

  res.send('this is what you GET when you request this path (/)')

})

app.use('/api/products', productRoutes) //connect the product url to the router

// 404 error creator:

app.use(notFoundError)

//error handling middleware:

app.use(allErrorsHandler)

app.listen(

  PORT,

  console.log(`Hey! server is running in ${mode} mode on port: ${PORT}`.yellow.bold)

)

Lets edit the products router: right now it’s throwing a 404 error manually, which don’t wont because he handle it with a custom middleware: backend🡪routes🡪 productsRouter.js🡺 we will set the request status as 404 and will throw a new error with a massage , the 404 error handler middle ware will catch it and handle it as we set it above!

import express from 'express'

import asyncErrorhandler from 'express-async-handler' //an npm pack for handling errors instead of using try catch

const router = express.Router()// api/products/...

import Product from '../models/productModel.js'//lets import the po

//fetch all products from DB

router.get('/', asyncErrorhandler( async (req, res) => {

    const products = await Product.find({}) //passing empty object will give us all the elements(products) as a promise!!!.

    res.json(products) //.json will sent the data as a JSON format!

}))

//fetch single  product by id from DB

router.get('/:id', asyncErrorhandler( async (req, res) => {

    //now lets serve a specific product data by url param, user id!

    const product = await Product.findById(req.params.id) // matching the user in the DB to the one being asked in the url parameter.

    if(product){

        res.json(product) //.json will sent the data as a JSON format!

    }else{

       // res.status(404).json({message:'Ho no! Product not found'})

       res.status(404) //if not set as 404 it will be 500 which is ok but it’s a better practice…

       throw new Error('Product not found in DB')

    }

}))

  export default router

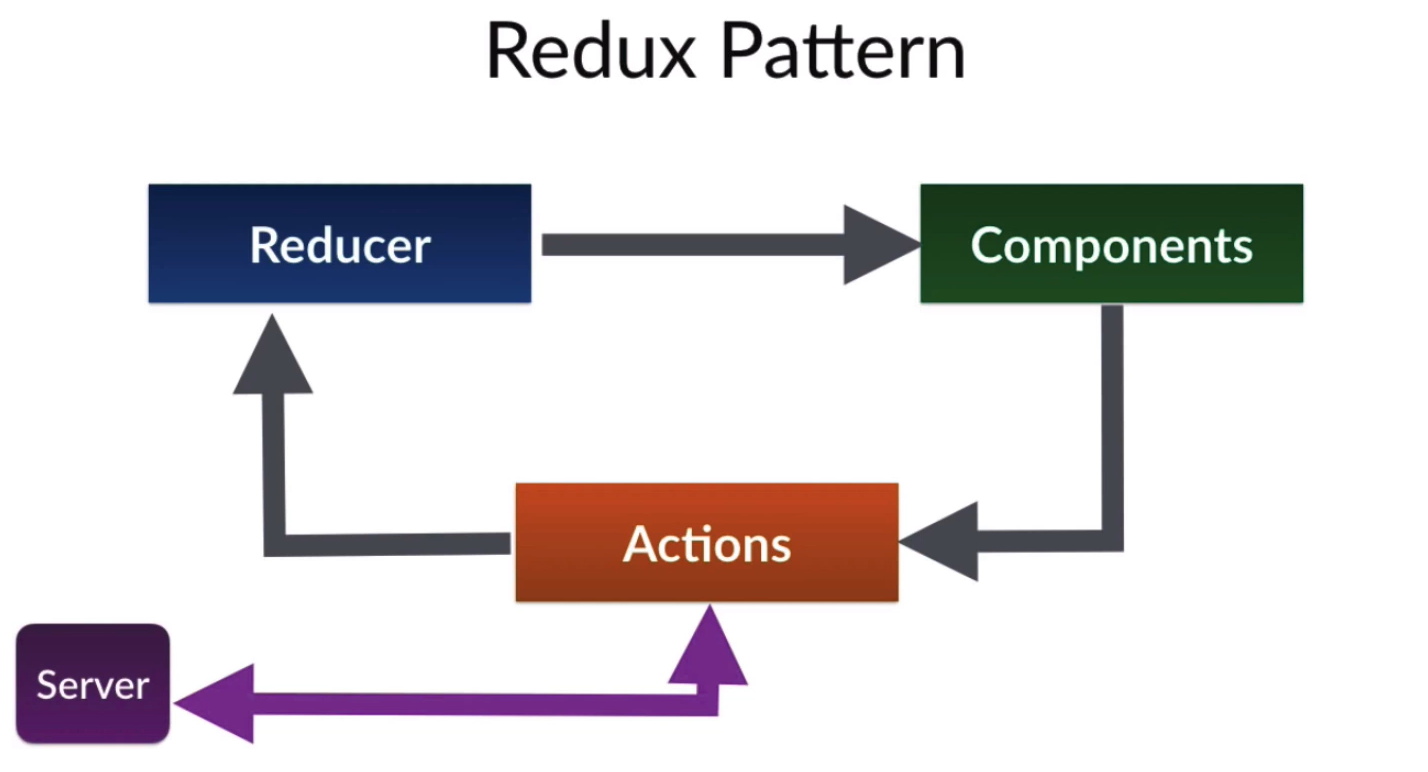
lets run both backend and front end dev servers : terminal : >>npm run dev

**REDUX**

A third-party library which can be used with other frameworks (like Angular) as well, we will use its react redux package.

A global state manager, allow the state to be passed all across the app without the need to pass it as a prop from parent to child component. While in parallel manage a local state in each component.

Simplified diagram of the data flow:



Global state in our example: shopping cart, authenticated user’s data and so on…any data that needs to be accessed all over the app should be at global state level.

* Reducers= functions that manipulate data and passing the state down to components.
* Actions= objects that represent the intension of changing the data in the state.
* Action creators/Dispatchers= functions that will initiate actions at will.

**Redux-thunk middleware:** With a plain basic Redux store, you can only do simple synchronous updates by dispatching an action. Middleware extends the store's abilities, and lets you write asynchronous logic that interacts with the store.

Redux Thunk middleware allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met. The inner function receives the store methods dispatch and getState as parameters.  
**An action creator that returns a function to perform asynchronous dispatch.**

**Redux dev-tools: a chrome browser extension. Make sure you install it. In the browser!!!**

**We will also have to install it as an npm pack plus adding a piece of code in oure project to make it work.**

**Install redux & Thunk & devtools:**

Terminal on Root level -> >>cd frontend

>>npm i redux react-redux redux-thunk redux-devtools-extension

**Creating the redux scaffolding**

<https://github.com/zalmoxisus/redux-devtools-extension#usage>

**Frontend🡪 src 🡪create a new file : store.js🡺**

import {createStore, combineReducers, applyMiddleware } from 'redux'

import thunk from 'redux-thunk'

import {composeWithDevTools} from 'redux-devtools-extension'

const reducer  = combineReducers({})

const initialState = {}

const middleWare = [thunk]

const store = createStore(

reducer,

 initialState,

 composeWithDevTools(applyMiddleware(...middleWare)

))

export default store

**Frontend🡪 src 🡪index.js (our react entry point):**

**We will pass the store data through a Provider that will wrap our reat app:**

import React from 'react'

import ReactDOM from 'react-dom'

import {Provider} from 'react-redux'

import store from './store'

import './bootstrap.min.css'

import './index.css'

import App from './App'

import \* as serviceWorker from './serviceWorker'

ReactDOM.render(

  <Provider store={store}>

    <App />

  </Provider>,

  document.getElementById('root')

)

// If you want your app to work offline and load faster, you can change

// unregister() to register() below. Note this comes with some pitfalls.

// Learn more about service workers: https://bit.ly/CRA-PWA

serviceWorker.unregister()

Redux-actions & reducers:

Create productReducers: plural, because the products going to have plenty of reducers === functions

**Actions-**  EACH REDUCER function:

* have “action” and “state” inputs
* contains a switch-case based on the action’s .payload and .type .

Src🡪 create a folder: reducers🡪 create a file: productReducers.js🡺

export const productListReducer  = ( state = {products: []}, action ) => { //

this is the reducer function ,we also export it, it will be consumed by the store entually!

    switch( action.type ){

        case 'PRODUCT\_LIST\_REQUEST': //A reducer’s case

            return {loading: true, products: [] } // fetching data ...

        case 'PRODUCT\_SUCCESS':

                return {loading: false, products: action.payload} //if fech success...

        case 'PRODUCT\_LIST\_FAIL':

                return {loading: false, error: action.payload} //if fech failed...

        default:

            return state //just pass the state…

    }

}

\*\*\*later the reducer cases names : 'PRODUCT\_LIST\_REQUEST' type of strings will be accessed from: src-> folder: constants

Src-> store.js🡺

Import the reducer file, it will be consumed by the store eventually:

import {createStore, combineReducers, applyMiddleware } from 'redux'

import thunk from 'redux-thunk'

import {composeWithDevTools} from 'redux-devtools-extension'

import {productListReducer} from './reducers/productReducers'

const reducer  = combineReducers({ //reducer is part of the store's inputs

    productList: productListReducer //we will access it (products data) from components in the app by a hook: useSelector(state=> state.productList)

})

const initialState = {}

const middleWare = [thunk]

//store is being consumed all accross the app. see index.js

const store = createStore( reducer, initialState, composeWithDevTools(applyMiddleware(...middleWare)))

export default store

so the product list that will be fetched , will be under the name .productlist .

**assign constants for the reducer’s cases ('PRODUCT\_LIST\_REQUEST'):**

src🡪 create a folder: constants 🡪 create a file: productsConstants.js🡺

Each string will be accessed from a constant variable, so it cannot be manipulated.

export const PRODUCTS\_LIST\_REQUEST = 'PRODUCTS\_LIST\_REQUEST'

export const PRODUCTS\_LIST\_SUCCESS = 'PRODUCTS\_LIST\_SUCCESS'

export const PRODUCTS\_LIST\_FAIL = 'PRODUCTS\_LIST\_FAIL'

back to Src🡪 reducers🡪 productsReducers.js🡺

we will import the constants, and use them in the case:

import {

         PRODUCTS\_LIST\_REQUEST,

         PRODUCTS\_LIST\_SUCCESS,

         PRODUCTS\_LIST\_FAIL

        } from '../constants/productsConstants'

export const productListReducer  = ( state = {products: []}, action ) => { //this is the reducer function we export it!

    switch( action.type ){

        case PRODUCTS\_LIST\_REQUEST:

            return {loading: true, products: [] } // fetching data ...

        case PRODUCTS\_LIST\_SUCCESS:

                return {loading: false, products: action.payload} //if fech success...

        case PRODUCTS\_LIST\_FAIL:

                return {loading: false, error: action.payload} //if fech failed...

        default:

            return state //pass the state as is.

    }

}

**Just a refresher:**

src🡪screens🡪 HomeScreen.js :

**currently we do the actual fetch (by axios) When the home screen component mounts:**

import React, { useState, useEffect } from 'react'

import { Row, Col } from 'react-bootstrap'

import axios from 'axios'

// import products from '../products' at first we brought the data from a js file.

import ProductCard from '../components/ProductCard'

const HomeScreen = () => {

  const [products, setProducts] = useState([]) //no initial state

  useEffect(() => {

    //will run as soon as this component mounts!

    const fetchProductsServer = async () => {

      //the data has been sent as json from  server.js (app.get...)

      const response = await axios.get('/api/products')

      const data = await response.data //no need to use .json() method to pase the data , its already parsed by axios.

      console.log(response)

      console.log(data)

      setProducts(data) //connecting the state to data ==> products will be equal to data

    }

    fetchProductsServer() //calling the function to fetch the products!

  }, [])

  return (

    <>

      <h1> Our Products </h1>

      <Row>

        {products.map((product) => (

          <Col sm={12} md={5} lg={4} xl={3} key={product.\_id}>

            <ProductCard product={product} />

          </Col>

        ))}

      </Row>

    </>

  )

}

export default HomeScreen

Src🡪 actions🡪 productsActions.js🡺

**Now we will fetch** **in this file**, as part of an **action creator** ( **function**) it will also **dispatch** the data to the **reducer** (which is consumed by **store** which is **accessible all across the app**.

In Redux, **action creators** simply return an **action**: such as :

‘type: PRODUCTS\_LIST\_FAIL’

The dispatch() function can be accessed directly from the store as [store.dispatch()](https://redux.js.org/api/store#dispatchaction), but more likely you'll access it using a helperlike react hooks useDispatch() to automatically bind many action creators to a dispatch() function.

**This file will be fired off when the HomeScreen component mounts!!!**

import axios from 'axios'

import {

     PRODUCTS\_LIST\_REQUEST,

    PRODUCTS\_LIST\_SUCCESS,

    PRODUCTS\_LIST\_FAIL

 } from '../constants/productsConstants'

export const listProducts =  ()=>{//this is an action creator function

     //we will do the fetch here! instead of the HomeScreen component (with axios)

     //because it's asynchronous we need redux thunk middleware:

      async (dispatch)=>{

        try{

            dispatch({type:  PRODUCTS\_LIST\_REQUEST}) // only type will be sent to product reducer

            const { data } = await axios.get('/api/products') //here the actual data of products fron the DB will be.

            dispatch({type: PRODUCTS\_LIST\_SUCCESS, payload: data}) //the actual products data will be sent to reducer as payload, and a type will be included!

        }catch(error){  // if the fetch failed we need to catch  en error that was sent by the server's error handler moddleware we built!!

            // type will be sent to product reducer as well as a payload with the propper error message.

            dispatch({type: PRODUCTS\_LIST\_FAIL,

                payload:

                error.response && error.response.data.message

                ?

                 error.response.data.message

                :

                error.message

                 })

        }

      }

}

**Back to**

src🡪screens🡪 HomeScreen.js 🡺

**we will edit the useEffect : right now it is still fetching the data , but we only want it to invoke the action file (productsActions)! We will use react hooks for accessing the dispatch**

// import React, { useState, useEffect } from 'react'

import React, { useEffect } from 'react'

import {useDispatch , useSelector } from 'react-redux' //using hooks for redux dispatch of the action creator

import { Row, Col } from 'react-bootstrap'

//import axios from 'axios'

// import products from '../products' at first we brought the data from a js file.

import ProductCard from '../components/ProductCard'

import { listProducts } from '../actions/productsActions'

const HomeScreen = () => {

  const dispatch = useDispatch()

  useEffect(() => {

    //make a request to the DB to fetch the products: by invoking productsActions.js

    dispatch(listProducts())

  }, [dispatch]) //will be re invoked it dispatch chages

  const products = [] //just for now so the app won’t crash

  return (

    <>

      <h1> Our Products </h1>

      <Row>

        {products.map((product) => (

          <Col sm={12} md={5} lg={4} xl={3} key={product.\_id}>

            <ProductCard product={product} />

          </Col>

        ))}

      </Row>

    </>

  )

}

export default HomeScreen

**final touches: connecting the component to redux’s global state and making conditional rendering of JSX code based on the product actions: success/ fail / loading….**

// import React, { useState, useEffect } from 'react'

import React, { useEffect } from 'react'

import {useDispatch , useSelector } from 'react-redux' //using hooks for redux dispatch of the action creator

import { Row, Col } from 'react-bootstrap'

//import axios from 'axios'

// import products from '../products' at first we brought the data from a js file.

import ProductCard from '../components/ProductCard'

import { listProducts } from '../actions/productsActions'

const HomeScreen = () => {

  const dispatch = useDispatch()

  const productsList = useSelector(state => state.productsList)

  const { loading , error , products } = productsList

  useEffect(() => {

    //make a request to the DB to fetch the products: by invoking productsActions.js

    dispatch(listProducts())

  }, [dispatch]) //will be re invoked it dispatch chages

  return (

    <>

      <h1> Our Products </h1>

          { loading ?

              ( <h2> Loading...</h2> ) : error ?

              (<h3> {error /\*this error is provided message from our server! \*/} </h3>)

              :

              ( //if fetching products action: success

                  <Row>

                        {products.map((product) => (

                          <Col sm={12} md={5} lg={4} xl={3} key={product.\_id}>

                            <ProductCard product={product} />

                          </Col>

                        ))}

                  </Row>

              )

          }

    </>

  )

}

export default HomeScreen

**and that’s it. Its running!**

**Creating a spinner for loading a component**

**Src🡪components🡪 create 2 files: Loader.js, Message.js :**

**Loader.js 🡺**

**Generate with abbreviation a react component: rafce  
we will also use react bootstrap spinner component:** [**https://react-bootstrap.github.io/components/spinners/**](https://react-bootstrap.github.io/components/spinners/)

import React from 'react'

import {Spinner} from 'react-bootstrap'

const Loader = () => {

    return (

        <Spinner animation= 'border' role='status' style={{width: '100px', margin: 'auto', height:'100px', display:'block' }}>

            <span class='sr-only'>Loading Products...</span>

        </Spinner>

    )

}

export default Loader

**Message.js🡺**

import React from 'react'

import {Alert}  from 'react-bootstrap'

const Message = ({variant, children}) => { //varient and children are set as props.

    return (

        <Alert varient={variant}>

            {children}

        </Alert>

    )

}

Message.defaultProps = { variant: 'dark'} //set default prop

export default Message

src🡪screens🡪 HomeScreen.js 🡺 we will import these two components:

// import React, { useState, useEffect } from 'react'

import React, { useEffect } from 'react'

import {useDispatch , useSelector } from 'react-redux' //using hooks for redux dispatch of the action creator

import { Row, Col } from 'react-bootstrap'

//import axios from 'axios'

// import products from '../products' at first we brought the data from a js file.

import ProductCard from '../components/ProductCard'

import { listProducts } from '../actions/productsActions'

import Message from '../components/Message'

import Loader from '../components/Loader'

const HomeScreen = () => {

  const dispatch = useDispatch()

  const productsList = useSelector(state => state.productsList)

  const { loading , error , products } = productsList

  useEffect(() => {

    //make a request to the DB to fetch the products: by invoking productsActions.js

    dispatch(listProducts())

  }, [dispatch]) //will be re invoked it dispatch chages

  return (

    <>

      <h1> Our Products </h1>

          { loading ? ( <Loader/> ) : error ?

              (<Message variant='info'>{error}</Message>)

              :

              ( //if fetching products action: success

                  <Row>

                        {products.map((product) => (

                          <Col sm={12} md={5} lg={4} xl={3} key={product.\_id}>

                            <ProductCard product={product} />

                          </Col>

                        ))}

                  </Row>

              )

          }

    </>

  )

}

export default HomeScreen

**right now we are passing each product data via props, we will fetch it from the global state using redux- constants, actions and reducers…**

**Product card with redux:**

**right now we are passing each product data via props, we will fetch it from the global state using redux- constants, actions and reducers… see frontend🡪src🡪screens🡪ProductScreen.js**

**Constants:**

Src🡪 constants🡪 productsConstants.js 🡺

Add the products card constants:

export const PRODUCTS\_LIST\_REQUEST = 'PRODUCTS\_LIST\_REQUEST'

export const PRODUCTS\_LIST\_SUCCESS = 'PRODUCTS\_LIST\_SUCCESS'

export const PRODUCTS\_LIST\_FAIL = 'PRODUCTS\_LIST\_FAIL'

export const PRODUCT\_CARD\_REQUEST = 'PRODUCT\_CARD\_REQUEST'

export const PRODUCT\_CARD\_SUCCESS = 'PRODUCT\_CARD\_SUCCESS'

export const PRODUCT\_CARD\_FAIL = 'PRODUCT\_CARD\_FAIL'

**REDUCERS**

Src🡪 constants🡪 productsConstants.js 🡺 add a new reducer function:

import {

         PRODUCTS\_LIST\_REQUEST,

         PRODUCTS\_LIST\_SUCCESS,

         PRODUCTS\_LIST\_FAIL,

         PRODUCT\_CARD\_REQUEST,

         PRODUCT\_CARD\_SUCCESS,

         PRODUCT\_CARD\_FAIL

        } from '../constants/productsConstants'

export const productsListReducer  = ...

export const productCardReducer  = ( state = {product: {reviews:[]}}, action ) => { //this is the reducer function we export it!

    switch( action.type ){

        case PRODUCT\_CARD\_REQUEST:

            return {loading: true, ...state } // fetching data ...

        case PRODUCT\_CARD\_SUCCESS:

                return {loading: false, product: action.payload} //if fech success...

        case PRODUCT\_CARD\_FAIL:

                return {loading: false, error: action.payload} //if fech failed...

        default:

            return state //pass the state as is.

    }

}

**Now each time you add a new reducer to your app, you also need to update the store.js file:**

**Src🡪store.js:**

import {createStore, combineReducers, applyMiddleware } from 'redux'

import thunk from 'redux-thunk'

import {composeWithDevTools} from 'redux-devtools-extension'

import {productsListReducer, productCardReducer } from './reducers/productsReducers'

const reducer  = combineReducers({ //reducer is part of the store's inputs

    productsList: productsListReducer,

    productDetails: productCardReducer

})

const initialState = {}

const middleWare = [thunk]

//store is being consumed all accross the app. see index.js

const store = createStore( reducer, initialState, composeWithDevTools(applyMiddleware(...middleWare)))

export default store

**ACTIONS**

src🡪actions🡪productsActions.js🡺

import the constants!

add another action function, notice it must get an id as an argument!

export const detailsProduct =  (id)=> async (dispatch)=>{ // we will call this function at ProductScreen.js and will pass it the id from the url params.

         try{

             dispatch({type: PRODUCT\_CARD\_REQUEST})

             const { product } = await axios.get(`/api/products/:${id}`)

             dispatch({type: PRODUCT\_CARD\_SUCCESS, payload: product})

         }catch(error){

             dispatch({type: PRODUCT\_CARD\_FAIL,

                 payload:

                 error.response && error.response.product.message

                 ?

                  error.response.product.message

                 :

                 error.message

                  })

         }

       }

**src🡪screens🡪ProductScreen.js**

Now we need to re arrange some things here.

We will fetch the data from the state and not from the component itself:

From this:

import React, { useState, useEffect } from 'react'

import { Link } from 'react-router-dom'

import { Row, Col, Image, ListGroup, Card, Button } from 'react-bootstrap'

import Ratings from '../components/Ratings'

//import products from '../products' //its not a react component but a js variable

import YouTubePlayer from '../components/YouTubePlayer'

import axios from 'axios'

const ProductScreen = (props) => {

  //we use props.match!

  //accessing the URL id param using props.match

  // const product = products.find(

  //   //this was used at the beginning when we fetch producs from ../products.js

  //   (element) => element.\_id === props.match.params.id

  // )

  const [product, setProduct] = useState({}) // initial state is an empty object, because product is an object.

  useEffect(() => {

    const fetchProdFromDB = async () => {

      //props.match.params.id allow us to get the params passed in the frontend's url !

      const response = await axios.get(`/api/products/${props.match.params.id}`)

      const data = await response.data

      console.log(data)

      setProduct(data) //setting the local state

    }

    fetchProdFromDB()

  }, [props.match])

  return (

    <>

      <Link className='btn btn-light my-5 ' to='/'>

        Back to Products

      </Link>

      <Row>

        <Col>

          <Image src={product.poster} alt={product.name} fluid />

        </Col>

      </Row>

      {/\*/////////////////////// \*/}

      <Row>

        <Col>

          <h1 className='text-center my-5'>

            A NEW KIND OF VOCAL TRANSFORMING PROCESSOR

          </h1>

        </Col>

      </Row>

      {/\*/////////////////////// \*/}

      <Row>

        <Col>

          <Row className='justify-content-center'>

            <Image src={product.interface} alt='product interface' fluid />

          </Row>

          <Row className='justify-content-center m-3'>

            <YouTubePlayer videoLink={product.video1} />{' '}

            {/\*youtube player is here! \*/}

          </Row>

          <Row>

            <Card className='priceCard'>

              <ListGroup variant='flush'>

                <ListGroup.Item>

                  <Row>

                    <Col className='my-1 text-dark h5'>Price:</Col>

                    <Col className='my-1 text-success h5'>

                      <strong>${product.price}</strong>

                    </Col>

                  </Row>

                </ListGroup.Item>

                <ListGroup.Item>

                  <Row>

                    <Col className='my-1 text-dark h5'>Status:</Col>

                    <Col className='my-1 text-info h5'>

                      {product.status ? ' Available' : 'Coming Soon'}

                    </Col>

                  </Row>

                </ListGroup.Item>

                <ListGroup.Item className='text-center py-0 '>

                  <Button

                    // className='btn-block' //will make it spread all across its div.

                    variant='info'

                    type='button'

                    disabled={!product.status}

                  >

                    Add to cart

                  </Button>

                </ListGroup.Item>

              </ListGroup>

            </Card>

          </Row>

        </Col>

        <Col>

          <ListGroup variant='flush'>

            <ListGroup.Item>

              <h2>{product.name}</h2>

            </ListGroup.Item>

            <ListGroup.Item>

              <h4>

                <Ratings value={product.rating} text={product.numReviews} />

              </h4>

            </ListGroup.Item>

            <Card

              style={{

                width: '100%',

              }}

            >

              <Card.Body>

                <Card.Title>

                  {' '}

                  <strong>Description</strong>

                </Card.Title>

                <Card.Text style={{ fontSize: '1rem' }}>

                  {product.longDescription}

                </Card.Text>

                <Button variant='secondary'>Read more</Button>

              </Card.Body>

            </Card>

          </ListGroup>

        </Col>

      </Row>

      {/\*/////////////////////// \*/}

      <Row>

        <Col className='text-center my-5'>

          <h1>FEATURES</h1>

          <ListGroup variant='flush'>

            {!product.features ? (

              <h1> ...loading </h1>

            ) : (

              product.features.map((feature) => (

                <ListGroup.Item>

                  <h5>{feature}</h5>

                </ListGroup.Item>

              ))

            )}

            <ListGroup.Item>

              <Image src={product.compatibility} alt='compatibility' fluid />

            </ListGroup.Item>

          </ListGroup>

        </Col>

      </Row>

    </>

  )

}

export default ProductScreen

to this:

import React, { useState, useEffect } from 'react'

import { Link } from 'react-router-dom'

import {useDispatch, useSelector} from 'react-redux'

import { productCardReducer } from '../reducers/productsReducers';

import { detailsProduct } from '../actions/productsActions';

import { Row, Col, Image, ListGroup, Card, Button } from 'react-bootstrap'

import Ratings from '../components/Ratings'

//import products from '../products' //its not a react component but a js variable

import YouTubePlayer from '../components/YouTubePlayer'

import Message from '../components/Message'

import Loader from '../components/Loader'

// import axios from 'axios'

const ProductScreen = (props) => {

    const dispatch = useDispatch()

    const productRes = useSelector(state=> state.productDetails) // initial state is an empty object, because product is an object.

    const {product, loading, error} = productRes

  useEffect(() => {

      dispatch(detailsProduct(props.match.params.id))

  }, [dispatch,props.match])

  return (

    <>

        <Link className='btn btn-light my-5 ' to='/'>

          Back to Products

        </Link>

    {loading ? <Loader/> : error ? <Message varient = 'dark'> {error} </Message>:(

        <>

         <Row>

         <Col>

           <Image src={product.poster} alt={product.name} fluid />

         </Col>

       </Row>

       <Row>

         <Col>

           <h1 className='text-center my-5'>

             A NEW KIND OF VOCAL TRANSFORMING PROCESSOR

           </h1>

         </Col>

       </Row>

       <Row>

         <Col>

           <Row className='justify-content-center'>

             <Image src={product.interface} alt='product interface' fluid />

           </Row>

           <Row className='justify-content-center m-3'>

             <YouTubePlayer videoLink={product.video1} />{' '}

             {/\*youtube player is here! \*/}

           </Row>

           <Row>

             <Card className='priceCard'>

               <ListGroup variant='flush'>

                 <ListGroup.Item>

                   <Row>

                     <Col className='my-1 text-dark h5'>Price:</Col>

                     <Col className='my-1 text-success h5'>

                       <strong>${product.price}</strong>

                     </Col>

                   </Row>

                 </ListGroup.Item>

                 <ListGroup.Item>

                   <Row>

                     <Col className='my-1 text-dark h5'>Status:</Col>

                     <Col className='my-1 text-info h5'>

                       {product.status ? ' Available' : 'Coming Soon'}

                     </Col>

                   </Row>

                 </ListGroup.Item>

                 <ListGroup.Item className='text-center py-0 '>

                   <Button

                     // className='btn-block' //will make it spread all across its div.

                     variant='info'

                     type='button'

                     disabled={!product.status}

                   >

                     Add to cart

                   </Button>

                 </ListGroup.Item>

               </ListGroup>

             </Card>

           </Row>

         </Col>

         <Col>

           <ListGroup variant='flush'>

             <ListGroup.Item>

               <h2>{product.name}</h2>

             </ListGroup.Item>

             <ListGroup.Item>

               <h4>

                 <Ratings value={product.rating} text={product.numReviews} />

               </h4>

             </ListGroup.Item>

             <Card

               style={{

                 width: '100%',

               }}

             >

               <Card.Body>

                 <Card.Title>

                   {' '}

                   <strong>Description</strong>

                 </Card.Title>

                 <Card.Text style={{ fontSize: '1rem' }}>

                   {product.longDescription}

                 </Card.Text>

                 <Button variant='secondary'>Read more</Button>

               </Card.Body>

             </Card>

           </ListGroup>

         </Col>

       </Row>

       <Row>

         <Col className='text-center my-5'>

           <h1>FEATURES</h1>

           <ListGroup variant='flush'>

             {!product.features ? (

               <h1> ...loading </h1>

             ) : (

               product.features.map((feature) => (

                 <ListGroup.Item>

                   <h5>{feature}</h5>

                 </ListGroup.Item>

               ))

             )}

             <ListGroup.Item>

               <Image src={product.compatibility} alt='compatibility' fluid />

             </ListGroup.Item>

           </ListGroup>

         </Col>

       </Row>

        </>

    )}

    </>

  )

}

export default ProductScreen

SHOPPING CART

Just before we handle the shopping cart we need to adjust some things in the product screen:

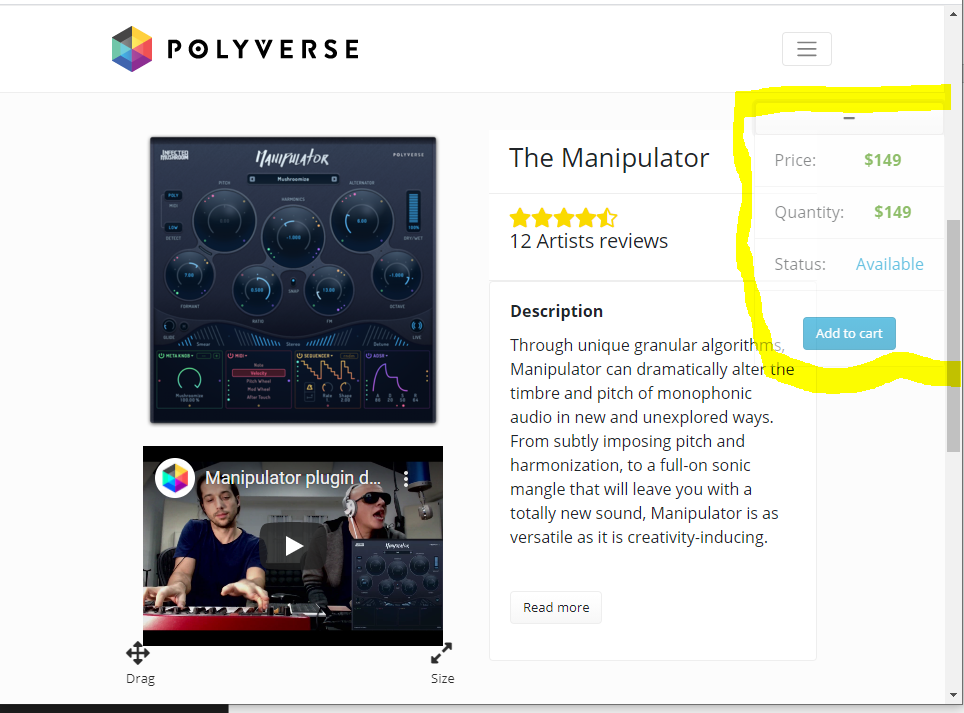
**src🡪screens🡪ProductScreen.js**

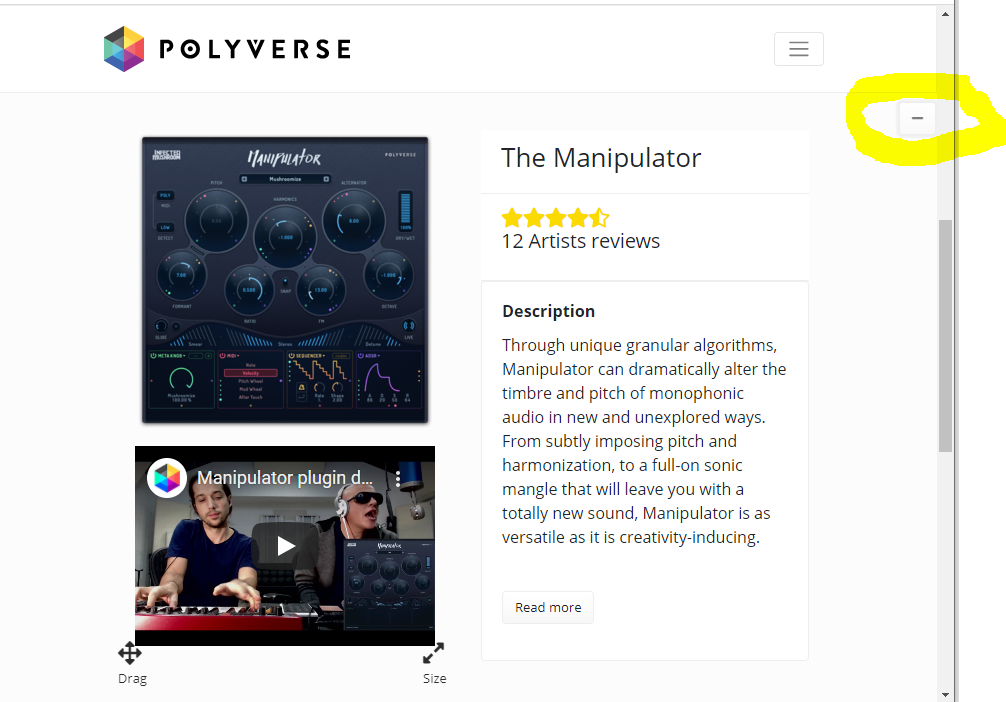
In the product screen we wand the user to have the ability to purchase as much as 4 items of the same kind.

Plus I want the price card to collapse to it won’t bother the user too much, although it will follow hom all across the screen.

we will have to use the useState Hook

let’s handle the collapsed price first





import { Row, Col, Image, ListGroup, Card, Button,Collapse  } from 'react-bootstrap'

const ProductScreen = (props) => {

    const [minimPrice, setMinimPrice] = useState(true) // this state will control the size of the price card

<Row>

               {/\*this is the price & buy card \*/}

               <Card className='priceCard'>

                    <Button variant='light' type='button'  onClick ={()=> setMinimPrice(!minimPrice)} aria-controls="example-collapse-text" aria-expanded={minimPrice}>

                            <i className='fas fa-minus'></i>

                    </Button>

                    <Collapse in={minimPrice}>

                        <div id="example-collapse-text">

                            <ListGroup variant='flush'>

                                <ListGroup.Item>

                                <Row>

                                    <Col className='my-1 text-dark h5'>Price:</Col>

                                    <Col className='my-1 text-success h5'>

                                    <strong>${product.price}</strong>

                                    </Col>

                                </Row>

                                </ListGroup.Item>

                                <ListGroup.Item>

                                <Row>

                                    <Col className='my-1 text-dark h5'>Quantity:</Col>

                                    <Col className='my-1 text-success h5'>

                                    <strong>${product.QTY}</strong>

                                    </Col>

                                </Row>

                                </ListGroup.Item>

                                <ListGroup.Item>

                                <Row>

                                    <Col className='my-1 text-dark h5'>Status:</Col>

                                    <Col className='my-1 text-info h5'>

                                    {product.status ? ' Available' : 'Coming Soon'}

                                    </Col>

                                </Row>

                                </ListGroup.Item>

                                <ListGroup.Item className='text-center py-0 '>

                                <Button variant='info'type='button' disabled={!product.status}>

                                    Add to cart

                                </Button>

                                </ListGroup.Item>

                            </ListGroup>

                        </div>

                    </Collapse>

             </Card>

           </Row>

Now lets add to the product Card a Quantity handler:

const [prodQuant, setProdQuant] = useState(1) // initial Quantity of 1 product

as you can see if the product status is false. Than the quantity component wont be rendered!(conditional rendereing) plus, a user can buy a max of 5 products of the same type.

On click each button will manipulate the local state of product quantity.

{product.status ? (

                                <ListGroup.Item>

                                <Row>

                                    <Col className='my-1 text-dark h5'>Quantity:</Col>

                                    <Button variant='light' type='button'   disabled={prodQuant <= 0} onClick ={()=> setProdQuant(prodQuant -1)} >

                                            <i className='fas fa-minus'></i>

                                    </Button>

                                    <Col className='my-1 h4'>

                                    <strong>{prodQuant}</strong>

                                    </Col>

                                    <Button variant='light' type='button'  disabled={prodQuant > 4} onClick ={()=> setProdQuant(prodQuant +1)} >

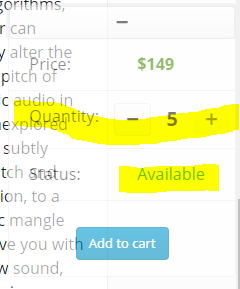
                                            <i className='fas fa-plus'></i>

                                    </Button>

                                </Row>

                                </ListGroup.Item>

                                ) : '' }



ADD TO CART BUTTON

**src🡪screens🡪ProductScreen.js**

<https://reactrouter.com/web/api/history>

We will add below the useEffect function our add to cart handler which will be fired off when the ADD TO CART button wil be clicked!

We will use props.match & props.history.push() (history-part of react router) in order to pass the Quantity and other data about the purchase using the url’s params and Quary string.

  const addToCartHandler = ()=>{

      props.history.push(`/cart/${props.match.params.id}?quant=${prodQuant}`) //now when we will push the button we will see this change in the url of the browser.

  }

Part of the product priceCard:

<ListGroup.Item className='text-center py-0 '>

                                <Button variant='info'type='button' disabled={!product.status} onClick={ addToCartHandler }>

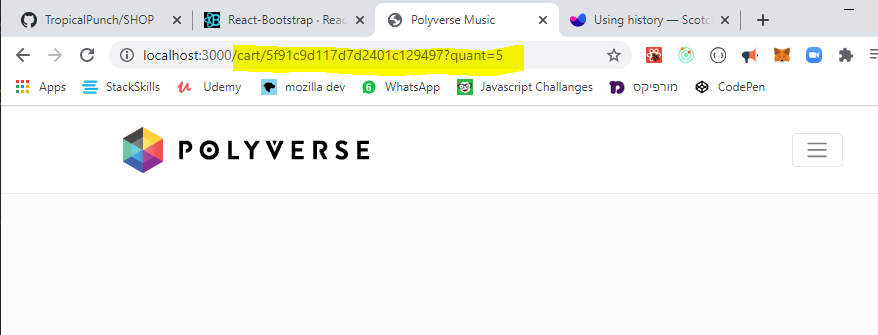
                                    Add to cart

                                </Button>

                                </ListGroup.Item>

The react router history object allows us to gain access to the browsers history api!

So When clicked:



SHOPPING CART

Finally!

Let’s set the shopping cart screen!!

frontend🡪src🡪screens 🡪 create a new file: CartScreen.js 🡺

generate a function component: (you may use **rafce)**

import React from 'react'

const CartScreen = () => {

    return (

        <div>

            this is the cart

        </div>

    )

}

export default CartScreen

go to frontend🡪src🡪App.js 🡺

we will add the rout to the cart component:

notice:

<Route path='/cart/:id?' component={CartScreen} />

/:id? 🡪 ?-at the end of the param means that the id param is optional! Because if we will access the cart from the products screen and not from the “Add to cart “ button then we will have no id reference!

import React from 'react'

import { BrowserRouter as Router, Route } from 'react-router-dom'

import { Container } from 'react-bootstrap'

import Header from './components/Header'

import Footer from './components/Footer'

import HomeScreen from './screens/HomeScreen'

import ProductScreen from './screens/ProductScreen'

import CartScreen from './screens/CartScreen'

const App = () => {

  return (

    <Router>

      <div>

        <Header style={{ position: 'relative' }} />

        <main className='py-4 '>

          <Container>

            <Route path='/' component={HomeScreen} exact />

            <Route path='/products/:id' component={ProductScreen} />

            <Route path='/cart/:id?' component={CartScreen} />

          </Container>

        </main>

        <Footer />

      </div>

    </Router>

  )

}

export default App

**CART FUNCTIONALITY USING REDUX:**

**CONSTANTS**

go to frontend🡪src🡪constants 🡪 create a new file : cartConstants.js🡺

export const CART\_ADD\_ITEM = 'CART\_ADD\_ITEM'

export const CART\_REMOVE\_ITEM = 'CART\_REMOVE\_ITEM'

**REDUCERS**

go to frontend🡪src🡪reducers 🡪 create a new file : cartReducers.js 🡺

\*\*\*we will have the id of the product named==>productId in both the payload & in the state!

import { CARD\_ADD\_ITEM } from '../constants/cartConstants'

export const cartReducer = (state = { cartItems: []}, action) =>{ //the initial state object is going to have an empty array of items(products).

    switch(action.type){

        case CARD\_ADD\_ITEM :

            const payloadProduct = action.payload

            //we will have the id of the product named==>productId in both the payload & in the state!

            const itemExists = state.cartItems.find(currentProduct =>  payloadProduct.productId === currentProduct.productId )

            if(itemExists){

                //the tricky line:

                return{...state, cartItems: state.cartItems.map(currentProduct => currentProduct.productId === payloadProduct.productId  ? payloadProduct : currentProduct )}

            }else{

                //when it does not exists, we will just push it to the state.

                return { ...state , cartItems: [...state.cartItems, payloadProduct] }

            }

        default :

        return state

    }

}

                //the tricky line:

                return{...state, cartItems: state.cartItems.map(currentProduct => currentProduct.productId === payloadProduct.productId  ? payloadProduct : currentProduct )}

so.. if the item we just sent to the cart already exists , we will map all over the cartItems array once we encounter the similar product we will overwrite it (so lets say we had already added product X to the cart with quantity of 1, and then added again product X with quantity of 3 the first addition will be deleted and replaced with the new quantity.)

**STORE**

go to frontend🡪src🡪store.js

Lets add the cart to the store

import {createStore, combineReducers, applyMiddleware } from 'redux'

import thunk from 'redux-thunk'

import {composeWithDevTools} from 'redux-devtools-extension'

import {productsListReducer, productCardReducer } from './reducers/productsReducers'

import {cartReducer} from './reducers/cartReducers'

const reducer  = combineReducers({ //reducer is part of the store's inputs

    productsList: productsListReducer,

    productDetails: productCardReducer,

    cart: cartReducer

})

const initialState = {}

const middleWare = [thunk]

//store is being consumed all accross the app. see index.js

const store = createStore( reducer, initialState, composeWithDevTools(applyMiddleware(...middleWare)))

export default store

**ACTIONS**

frontend🡪src🡪actions🡪create a new file : cartActions

in the cart action we will fetch the products id from the DB , therefore it’s an asynchronous operation in order to implement this we will use axios.

Also when the cart action will be fired off it will automatically be saved in the user’s local storage.

import axios from 'axios'

import {CART\_ADD\_ITEM} from '../constants/cartConstants'

//here we are using thunk...

//we also gonna save the entire cart in the browsers local storage.

//getSate will allow us to interact with the app state as written in the store

export const addToCartAction = (id, quantity ) => async (dispatch,getState)=>{

    const {data} = await axios.get(`/api/products/${id}`) //we will destructure the data from the request.

    dispatch({

        type: CART\_ADD\_ITEM,

payload:{

         productId: data.\_id,

         name: data.name,

         image:data.image,

         price: data.price,

         quantity //users input

}

    })

    //lets save it in loca storage : we may save only strings therefore we need JSON.stringify

    ///once we take it oute well use JSON.parse

    localStorage.setItem('cartItems',JSON.stringify(getState().cart.cartItems))

}

Now in order to have access to the user’s local storage in our APP’s STATE we must go back to the store file:

**STORE**

go to frontend🡪src🡪store.js

Now in order to have access to the user’s local storage in our APP’s STATE we must go back to the store file so if it exists it will immediately be part of the initial state! (and accessible from the store)

import {createStore, combineReducers, applyMiddleware } from 'redux'

import thunk from 'redux-thunk'

import {composeWithDevTools} from 'redux-devtools-extension'

import {productsListReducer, productCardReducer } from './reducers/productsReducers'

import {cartReducer} from './reducers/cartReducers'

const reducer  = combineReducers({ //reducer is part of the store's inputs

    productsList: productsListReducer,

    productDetails: productCardReducer,

    cart: cartReducer

})

const cartLocalStorage = localStorage.getItem('cartItems') ? JSON.parse( localStorage.getItem('cartItems')) : []

const initialState = {

    cart:{cartItems: cartLocalStorage}

}

const middleWare = [thunk]

//store is being consumed all accross the app. see index.js

const store = createStore( reducer, initialState, composeWithDevTools(applyMiddleware(...middleWare)))

export default store

**Back to the Cart Component**

frontend🡪src🡪screens 🡪 create a new file: CartScreen.js 🡺

props.location: <https://reactrouter.com/web/api/location>

import React ,{useEffect} from 'react'

import{Link} from 'react-router-dom'

import{useDispatch, useSelector} from 'react-redux'

import {Row,Col,ListGroup, Image, From, Button, Card} from 'react-bootstrap'

import Message from '../components/Message'

import {addToCartAction} from '../actions/cartActions'

const CartScreen = (props) => {

    const {match, location, history} = props

    const productId = match.params.id //if we click the cart icon before we addded any product there wont be any id!

    const quantity = location.search ? Number(location.search.split('=')[1]) : 1

     //location.search -> will get the url's quary param meaning the string that is after the id in the url.

    //in our case:   /cart/5f91c9d117d7d2401c129497?quant=1 ==> ?quant=1

    //inorder to gain access to the number we will split the string in the '=' mark and it will turn it into an array.

    /// and then we'll grab the number by the [1] index. we will still need to turn it into a number form by  Number() method

    //console.log(quantity)

    const dispatch = useDispatch()

    const cart = useSelector(state=> state.cart) //gain access to the cartItems from the state.

    const {cartItems} = cart

    useEffect(() => {

        if(productId){

            dispatch(addToCartAction(productId,quantity))

        }

    }, [dispatch,productId,quantity])

    return (

        <div>

           <h1>this is the cart</h1>

        </div>

    )

}

export default CartScreen

now lets add some front-end functionality:

import React ,{useEffect} from 'react'

import{Link} from 'react-router-dom'

import{useDispatch, useSelector} from 'react-redux'

import {Row,Col,ListGroup, Image, From, Button, Card} from 'react-bootstrap'

import Message from '../components/Message'

import {addToCartAction} from '../actions/cartActions'

const CartScreen = (props) => {

    const {match, location, history} = props

    const productId = match.params.id //if we click the cart icon before we addded any product there wont be any id!

    const quantity = location.search ? Number(location.search.split('=')[1]) : 1

     //location.search -> will get the url's quary param meaning the string that is after the id in the url.

    //in our case:   /cart/5f91c9d117d7d2401c129497?quant=1 ==> ?quant=1

    //inorder to gain access to the number we will split the string in the '=' mark and it will turn it into an array.

    /// and then we'll grab the number by the [1] index. we will still need to turn it into a number form by  Number() method

    //console.log(quantity)

    const dispatch = useDispatch()

    const cart = useSelector(state=> state.cart) //gain access to the cartItems from the state.

    const {cartItems} = cart

    useEffect(() => {

        if(productId){

            dispatch(addToCartAction(productId,quantity))

        }

    }, [dispatch,productId,quantity])

    const removeItemFromCart = (id) =>{ console.log('removed')} //see page 124

    return (

        <div>

           <Row className='py-5' >

               <Col md={8}>

                   <h1>Shopping Cart</h1>

                   {cartItems.length === 0 ?

                    <Message><h3>Cart is currently empty  <Link to='/'>Back To Products</Link></h3></Message>

                    :

                    (<ListGroup variant='flush'>

                        {cartItems.map(productProp => (

                            <ListGroup.Item key={productProp.productId}  className= ''>

                                 <Row className="float-right">

                                    <Button  type='button' variant='light'  onClick={()=> removeItemFromCart(productProp.productId)}>

                                        <i className='fas fa-times'></i>

                                    </Button>

                                 </Row>

                                <Row>

                                    <Col className="float-left" >

                                        <Image className="float-left" src={productProp.image} alt={productProp.name}  fluid ></Image>

                                    </Col>

                                    <Col className='d-flex flex-column pt-4'>

                                        <Row>

                                            <Link className='font-weight-bold h3 ' to={`/product/${productProp.productId}`}>{productProp.name}</Link>

                                        </Row>

                                        <Row className='font-weight-bold h3 p-2' >

                                           ${productProp.price}

                                        </Row>

                                        <Row className="d-flex flex-nowrap justify-content-start ">

                                            <p className='text-dark h4 font-weight-bold p-2 '>Quantity:</p>

                                            <Button variant="outline-light" type='button'  disabled={productProp.quantity <= 1} onClick ={()=> dispatch(addToCartAction(productProp.productId, productProp.quantity - 1))} >

                                                    <i className='fas fa-minus'></i>

                                            </Button>

                                            <div className='h4 d-flex justify-content-center p-2'>

                                                    <strong>{productProp.quantity}</strong>

                                            </div>

                                            <Button   variant="outline-light" type='button' disabled={productProp.quantity > 4} onClick ={()=> dispatch(addToCartAction(productProp.productId, productProp.quantity +1))} >

                                                    <i className='fas fa-plus'></i>

                                            </Button>

                                        </Row>

                                    </Col>

                                </Row>

                            </ListGroup.Item>

                        ))}

                    </ListGroup>)

                    }

               </Col>

               <Col md={4}>

//Next- we will add a card with the total price count and items in the cart

               </Col>

           </Row>

        </div>

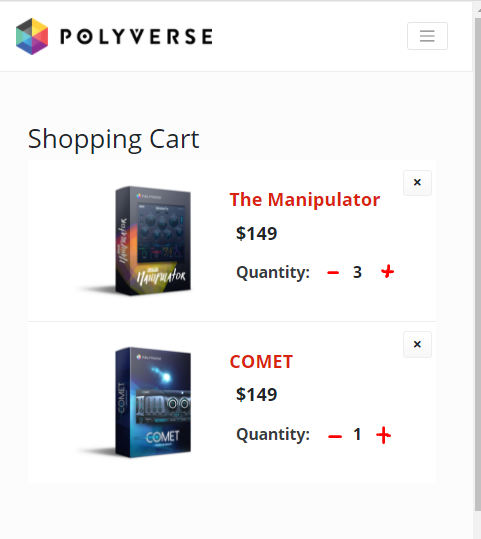
    )

}

export default CartScreen

see explanation and visual 🡺

**so the cart will look like this**



**Summary for the cart functionality so far:**

1. So we used useSelector to gain access to the store -which contains the items we added to the cart ,whether it’s in the local storage or from current session, under state .cart.cartItems (seecaretActions file)
2. we used useEffect- so each time product id or quantity or/ and the amount of items the user want has changed the cart component will be re rendered and the state will be changed by firing the “addToCartAction” function.
3. Conditional rendering- If the cartItems array is empty a relevant message will screen by using the message component.
4. Conditional rendering- If the cartItems array is full- a list of all the items in the cart will be shown, plus a button for deleting an item and buttons for adding or subtracting more items of the same product.
5. Each item in the product cart array was maped and named : “productProp” 🡺 productProp.ProductId / .image / . name / .quantity / .price (these names derive straight from the “cartActions” file.

**Total price ant item calc**

**Still in:** frontend🡪src🡪screens 🡪 create a new file: CartScreen.js 🡺

      ))}

                    </ListGroup>)

                    }

               </Col>

               <Col md={4}>

//Next- we will add a card with the total price count and items in the cart

               </Col>

           </Row>

        </div>

    )

}

export default CartScreen

we will place this code:

<Card className='mt-5' >

  <ListGroup variant='flush' className="d-flex align-items-center" >

       <ListGroup.Item >

           <h2>Subtotal ({cartItems.reduce((acc,currItem)=> acc +  acurrItem.quantity,0)} items): ${cartItems.reduce((acc,currItem)=> acc +  currItem.quantity\*currItem.price,0).toFixed(2)} </h2>

         <Button onClick={checkOutHandler} disabled={cartItems.length === 0}   variant="warning" block>

           <h3> Proceed to checkout</h3>

         </Button>

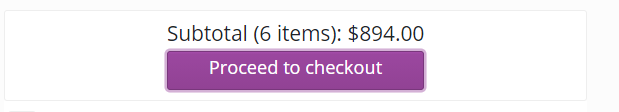
      </ListGroup.Item>

     </ListGroup>

                   </Card>

We used reducer array method to calculate the sum of items in the cart and the total of dollars for all items. toFixed-> to show two digits after the decimal point.

It will look like that:



delete from cart handler &

Procced to checkout Handler first steps

When clicking the button we want to re-direct the user to the login page in order to complete the purchase, if he is already logged , he will be re-directed to the checkout (billing & shipping).

How we do it ? 🡺 history.push('/login?redirect=shipping')

In order to implement this we will need to build the user authentication both backend and frontEnd.

First we will implement the delete from cart.

**delete from cart handler: with redux**

**CONSTANTS:**

go to frontend🡪src🡪constants 🡪 create a new file : cartConstants.js🡺

export const CART\_ADD\_ITEM = 'CART\_ADD\_ITEM'

export const CART\_REMOVE\_ITEM = 'CART\_REMOVE\_ITEM'

**REDUCERS:**

go to frontend🡪src🡪reducers 🡪 create a new file : cartReducers.js 🡺

we’ll add this code:

import { CART\_ADD\_ITEM } from '../constants/cartConstants'

import { CART\_REMOVE\_ITEM } from '../constants/cartConstants'

export const cartReducer = (state = { cartItems: []}, action) =>{ //the initial state object is going to have an empty array of items(products).

    switch(action.type){

        case CART\_ADD\_ITEM :

            const payloadProduct = action.payload

            //we will have the id of the product named==>productId in both the payload & in the state!

            const itemExists = state.cartItems.find(currentProduct => currentProduct.productId === payloadProduct.productId  )

            if(itemExists){

                //the tricky line:

                return{...state, cartItems: state.cartItems.map(currentProduct => currentProduct.productId === payloadProduct.productId  ? payloadProduct : currentProduct )}

            }else{

                //when it does not exists, we will just push it to the state.

                return { ...state , cartItems: [...state.cartItems, payloadProduct] }

            }

        case CART\_REMOVE\_ITEM :

            return{...state, cartItems: state.cartItems.filter(currentProduct => currentProduct !== action.payload )} //we’ll filter out all products that don’t equal to the product in the payload (we will set a payload in the action file.

        default:

        return state

    }

}

**ACTIONS:**

frontend🡪src🡪actions🡪create a new file : cartActions

import axios from 'axios'

import {CART\_ADD\_ITEM, CART\_REMOVE\_ITEM} from '../constants/cartConstants'

//here we are using thunk...

//we also gonna save the entire cart in the browsers local storage.

//getSate will allow us to interact with the app state as written in the store

export const addToCartAction = (id, quantity ) => async (dispatch,getState)=>{

    const {data} = await axios.get(`/api/products/${id}`) //we will destructure the data from the request.

    dispatch({

        type: CART\_ADD\_ITEM,

        payload: {

            productId: data.\_id,

            name: data.name,

            image:data.image,

            price: data.price,

            quantity //users input= action argument

        },

    })

    //lets save it in loca storage : we may save only strings therefore we need JSON.stringify

    ///once we take it out (in the store) well use JSON.parse()

    localStorage.setItem('cartItems',JSON.stringify(getState().cart.cartItems))

}

export const removeFromCartAction = (id) => async (dispatch,getState)=>{

    //the action argument is the product ID!!

    dispatch({

        type: CART\_REMOVE\_ITEM,

        payload: id, //the paload is the ID that we got as an argument!!

    })

    //lets save it in loca storage : we may save only strings therefore we need JSON.stringify

    ///once we take it out (in the store) well use JSON.parse()

    localStorage.setItem('cartItems',JSON.stringify(getState().cart.cartItems))

}

**cartScreen.js-**

frontend🡪src🡪screens 🡪 create a new file: CartScreen.js 🡺

back to cartScreen we’ll code the delete removeItemFromCart handler that will be fired when clicing the delete button: the habdler will fire off the action while passing it the id od the product!

The button code:

<Button  type='button' variant='light' onClick={()=> removeItemFromCart(productProp.productId)}>

                                        <i className='fas fa-times'></i>

                                    </Button>

The action import statement:

import {addToCartAction, removeFromCartAction} from '../actions/cartActions'

The handler code:

const removeItemFromCart = (id) =>{

        dispatch(removeFromCartAction(id))

}

That’s it! The item will be removed when clicked the button.

**Backend user authentication**

**SETUP & CLENUP PRE WORK**

Lets clean up the code in our server:  
Root->backend🡪 routes🡪productsRoutes.js

BEFORE clenup: copy the lines highlighted

import express from 'express'

import asyncErrorhandler from 'express-async-handler' //an npm pack for handling errors instead of using try catch

const router = express.Router()// api/products/...

import Product from '../models/productModel.js'//lets import the po

//fetch all products from DB

router.get('/', asyncErrorhandler( async (req, res) => {

    const products = await Product.find({}) //passing empty object will give us all the elements(products) as a promise FROM THE DB!!!.

    // throw new Error('throw error at will for fun:)')

    res.json(products) //.json will sent the data as a JSON format! sowhen fatching it we will nedd to JSON.parse()

}))

//fetch single  product by id from DB

router.get('/:id', asyncErrorhandler( async (req, res) => {

    //now lets serve a specific product data by url param, user id!

    const product = await Product.findById(req.params.id) // matching the user in the DB to the one being asked in the url parameter.

    if(product){

        res.json(product) //.json will sent the data as a JSON format!

    }else{

       // res.status(404).json({message:'Ho no! Product not found'})

       res.status(404)

       throw new Error('Product not found in DB')

    }

}))

  export default router

by creating controllers we will make the two callbacks functions, we just highlighted, variables being called from a separate file. (you might also call it: separation of concerns)

Root->backend🡪 create a new folder: controllers 🡪create a new file: productControllers.js🡺

Create the controllers for fetching all products from DB & fetching by product ID:

import Product from '../models/productModel.js'

import asyncErrorhandler from 'express-async-handler' //an npm pack for handling errors instead of using try catch

const getProducts = asyncErrorhandler(async (req,res)=>{

    const products = await Product.find({}) //get all products

 res.json(products)//send all products currently in db!

})

const getProductById =  asyncErrorhandler( async (req, res) => {

    //now lets serve a specific product data by url param, user id!

    const product = await Product.findById(req.params.id) // matching the user in the DB to the one being asked in the url parameter.

    if(product){

        res.json(product) //.json will sent the data as a JSON format!

    }else{

       // res.status(404).json({message:'Ho no! Product not found'})

       res.status(404)

       throw new Error('Product not found in DB')

    }

})

export {getProductById, getProducts} //don’t forget to export the functions!

Root->backend🡪 routes🡪productsRoutes.js

AFTER clenup:

import express from 'express'

//import asyncErrorhandler from 'express-async-handler' //an npm pack for handling errors instead of using try catch

const router = express.Router()// api/products/...

//import Product from '../models/productModel.js'//lets import the po

import {getProductById, getProducts} from

'../controllers/productControllers.js' //import the controllers

//fetch all products from DB

router.get('/', getProducts )

//fetch single  product by id from DB

router.get('/:id',getProductById )

  export default router

we may also write this syntax this way:

//fetch all products from DB

// syntax B-> router.get('/', getProducts )

router.route('/').get(getProducts )

//fetch single  product by id from DB

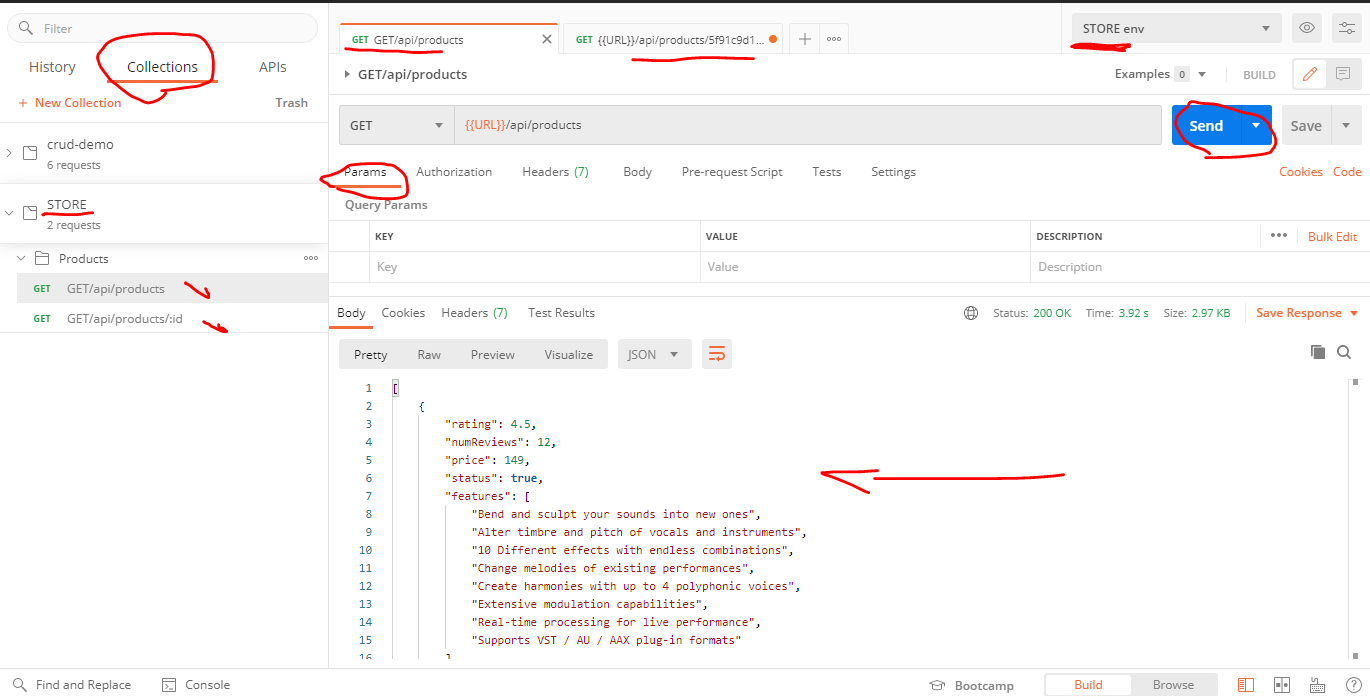
//syntax B-> router.get('/:id',getProductById )

router.route('/:id').get(getProductById)

in the terminal at root level run the server only:

>> npm run server

Open POSTMAN and send a get request for the api/products and products/--a product id--:



**So after the controller setup Lets start with the coding the Authentication endpoint:**

Remember on p 67 when we seeded the DB with users and produts?

At that point we added fictional users with a password of 1-6 (including an admin user),

you can also check it on: backend🡪 data 🡪 users.js.

1. backend🡪routes🡪 create a new file: userRoutes.js 🡺
2. **BODY PARSER:**

backend🡪server.js

Because we send data using the body of the HTML we need to add to server.js this code:

Import the user router: import userRoutes from './routes/userRoutes.js'//import the routes of users (login).

Add the router of the users/login:

app.use('/api/products', productRoutes) //connect the product url to the router

app.use('/api/users/login', userRoutes)//connect the users/login url to the router

const app = express()

// Just beneath the app variable add:

app.use(express.json()) //will allow us to parse json data that arrives  through the body

1. Backend🡪 routes🡪 create a file : userRoutes.js🡺

import express from 'express'

const router = express.Router()// api/users/login

import {authUser} from '../controllers/userControllers.js' //this is the controller!

//fetch users email and password from the DOM(body)

router.post('/login',authUser)

export default router

1. backend🡪controllers🡪 create a new file: userControllers.js 🡺

this file is going to handle the user authentication & get token (we’ll get to tokens later)

this is a post request to route: api/users/login

import User from '../models/userModel.js'

import asyncErrorhandler from 'express-async-handler' //an npm pack for handling errors instead of using try catch

//this is a post request to the route: api/users/login

const authUser = asyncErrorhandler(async (req,res)=>{

   const {email,password} =req.body  //extract the email & password from the body!

   res.send({email,password})

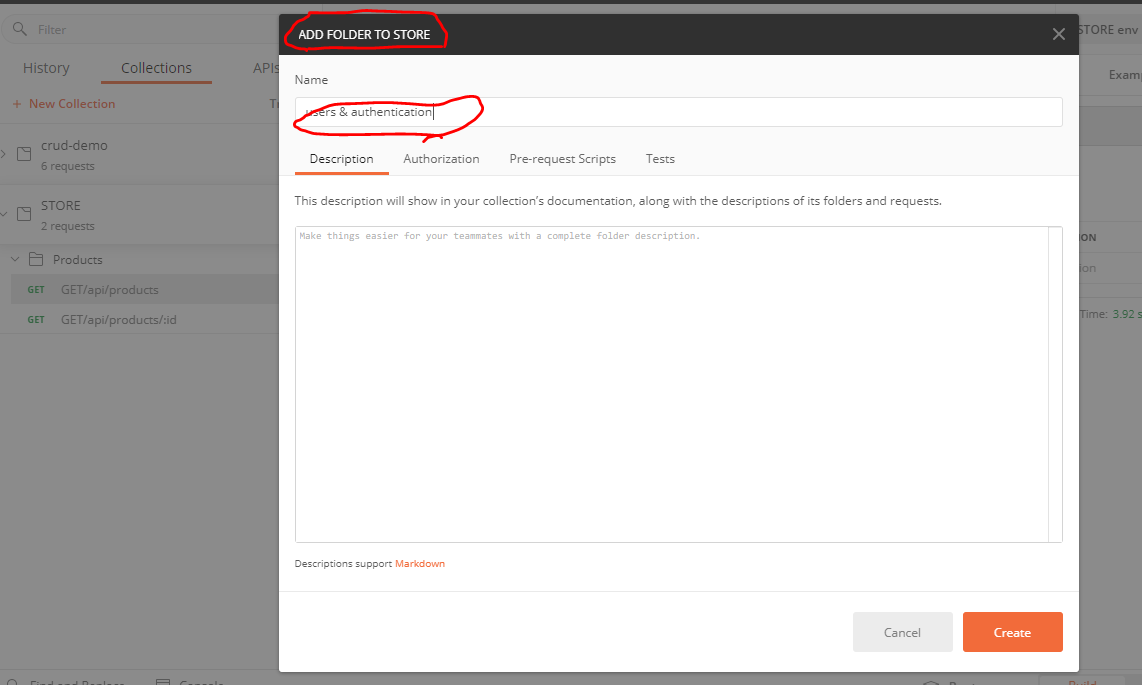
})

export {authUser}

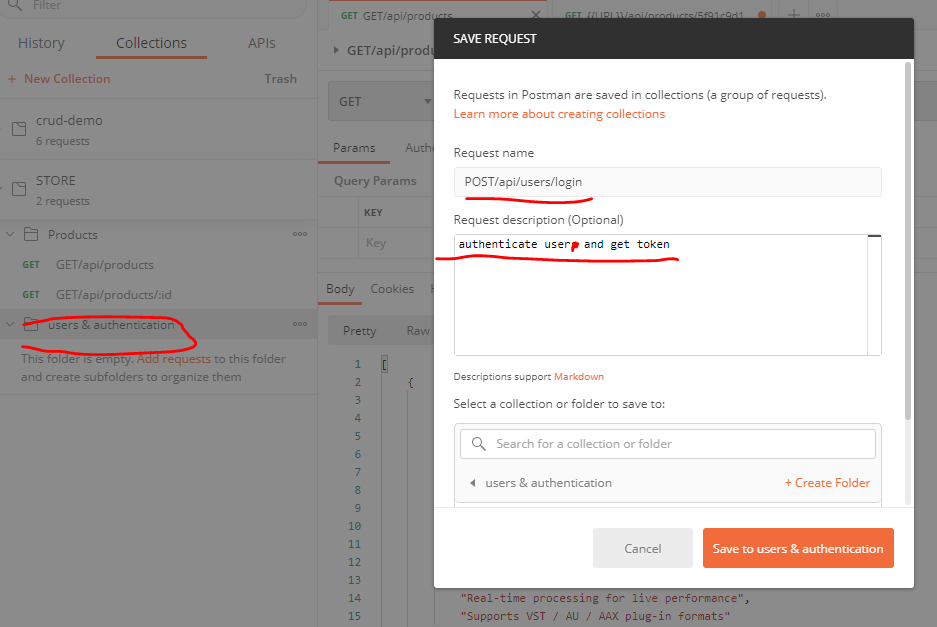
1. before we start the frontend work, we’ll use postman

postman:

create a new folder for users routes and auth



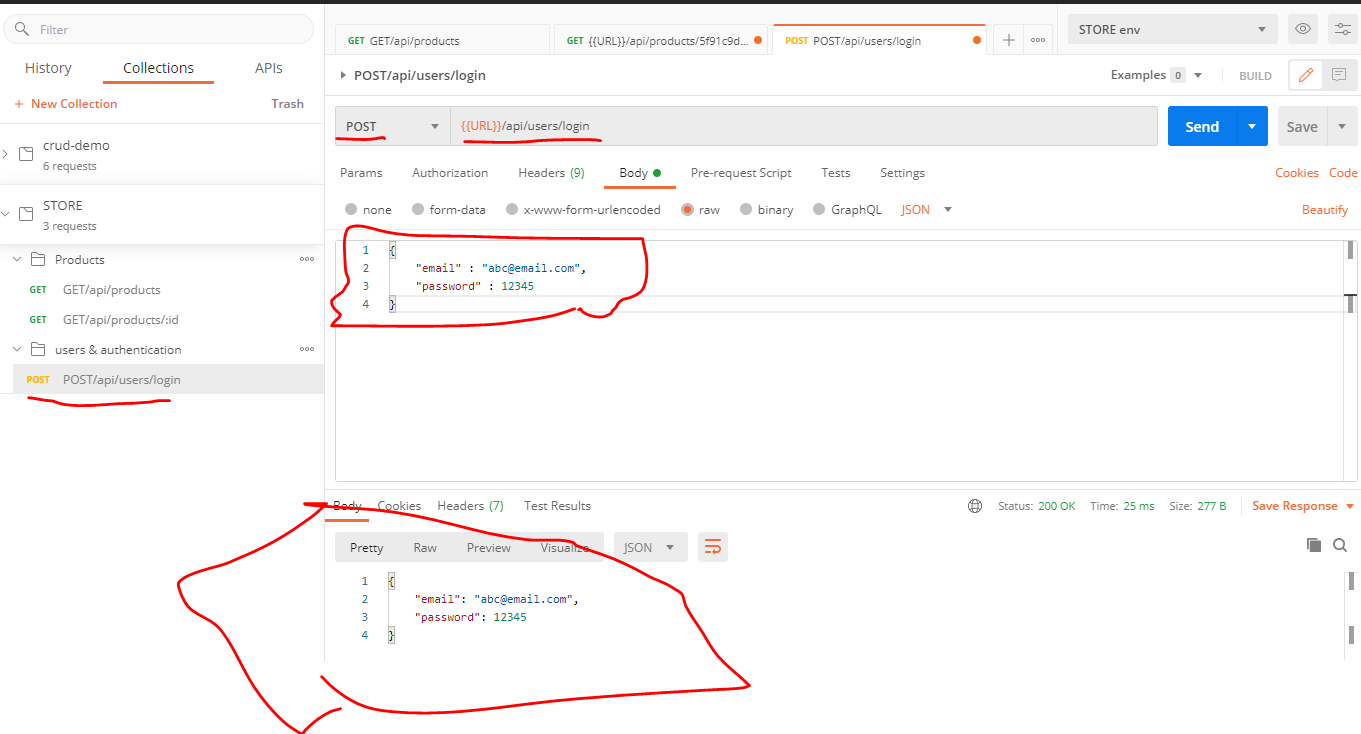
Create a new POST request



Edit the request:



* POST, {{URL}}/api/users/login, Body-the data will be passed through the body, raw, JSON, and add dummy data we want to send to the server (email and password in the user case).



As you can see the response in post man will be exactly what we wrote in the “body” so now we have access to data that’s being sent from the body!

**Ok so now we need to match the inputs from the body and see if they exists in the DB !!!**

backend🡪controllers🡪 userControllers.js 🡺

import User from '../models/userModel.js'

import asyncErrorhandler from 'express-async-handler' //an npm pack for handling errors instead of using try catch

//this is a post request to route: api/users/login

const authUser = asyncErrorhandler(async (req,res)=>{

   const {email,password} = req.body //extract the email & password from the body!

   //res.send({email,password}) //we have access to inputs from the body

    const user = await User.findOne({email: email})

    //we need to use bcypt beacuse we store the passwords in the DB in an encrypted from

   //we also constructed a method for decrypting the password "matchPassword", you can it in userModel.js

    if(user && await user.matchPassword(password)){

        res.json({

            \_id: user.\_id,

            name: user.name,

            email: user.email,

            isAdmin: user.isAdmin,

            token: null //soon we'll deal with tokens, for now null

        })

    }else{

        res.status(401)

        throw new Error('Invalid Email / Password')

    }

})

export {authUser}

backend🡪models🡪 userModel.js 🡺

\*here we are coding the method we use inorder to check if the password enterd is equal to the one the user entered by matching them

import mongoose from 'mongoose';

import bcrypt from 'bcryptjs'

const userSchema = mongoose.Schema({

    name:{

        type: String,

        required: true

    },

    email:{

        type: String,

        required: true,

        unique: true

    },

    password:{

        type: String,

        required: true

    },

    isAdmin:{

        type: Boolean,

        required: true,

        default: false

    }

},{

    //mongoose built in time stamps for "created at"/ "updated at"

    timestamps: true

})

userSchema.methods.matchPassword = async function(enteredPassword){ //we assign a method manualy to the user schema

 return await bcrypt.compare(enteredPassword, this.password)

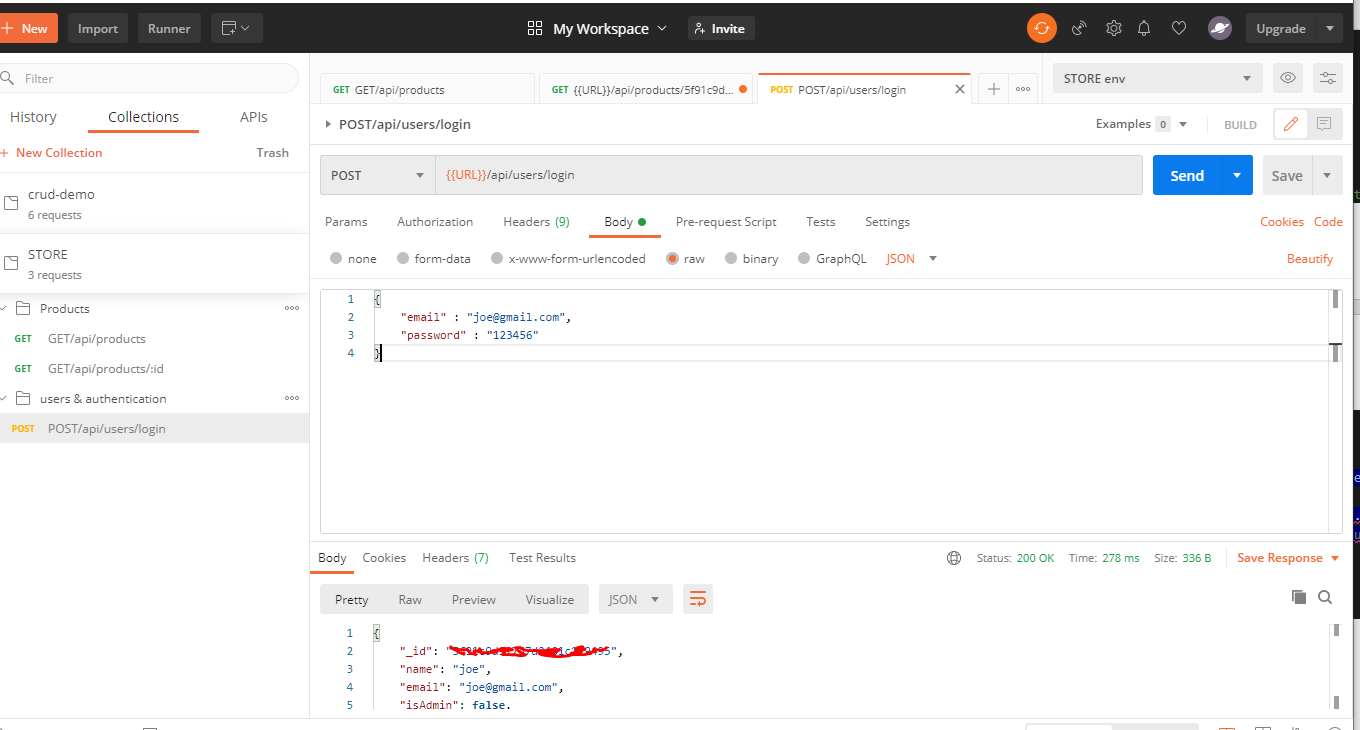
}//we will use this method "matchPassword" in the userController

const User = mongoose.model('User', userSchema)

export default User

**let’s check the code we wrote so far using postman:**  what happens when we write an email that does no exists in the DB or pass word and what happen when we use valid data.

Donst forget to ass password as a string! “12345…”



Generating TOKENS using: JWT

So far we conducted **Authentication** to the user, by searching if the email and passwords he provided in the login are correct and exists it the DB.

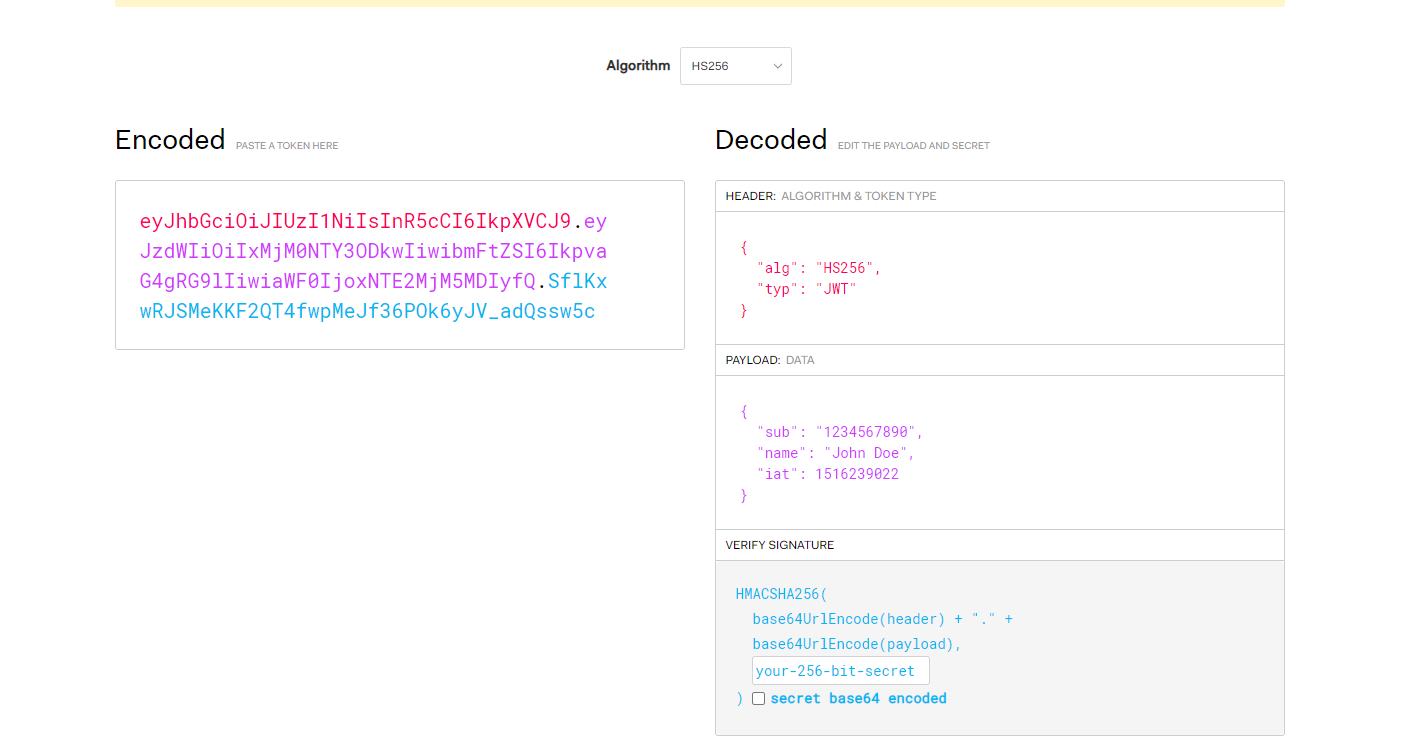
Now we handle the **Authorization** process using JWT – JSON WEB TOKENS, we will give the authenticate user a token which will allow him access in to various pages web (protected routes).

We will create protected routes soon using a middleware.

**We will use an npm package to generate the tokens- >>npm install jsonwebtoken**

[**https://www.npmjs.com/package/jsonwebtoken**](https://www.npmjs.com/package/jsonwebtoken)

<https://jwt.io/>



Header-> name the type of token + the hash algorithem name in use

{

"alg": "HS256",

"typ": "JWT"

}

Payload-> will usually contain user id or session id – not a sensitive data which we can pass to the server and get in return data.

{

"sub": "1234567890",

"name": "John Doe",

"iat": 1516239022 //when it was issued

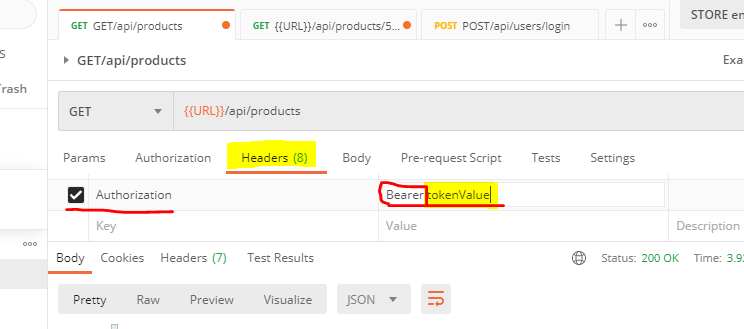
“exp” : 2342342 //when the token will expire

}

Signature-> used to verify nothing has changed

**Using postman in the authorization process:**

\*lets say the current rout is protected

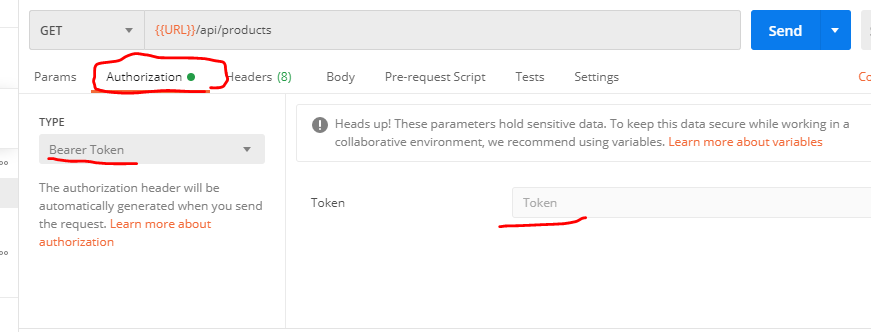


Notice you must use **Bearer** and then write the token value

Or, you can use the Authorization tab (between Params and Headers):

Type = Bearer Token

Token = token value



Generating TOKENS

**>>npm install jsonwebtoken**

**Generally- in order to generate the token we will have to use the jwt.sign() method**

**The pattern according the documentation**

var jwt **=** require('jsonwebtoken');

var token **=** jwt.sign({ foo**:** 'bar' }, 'shhhhh', {option});

* foo : user id
* ‘shhh’ =the secret string , we will keep it in the .env file
* Option= we’ll add an expiry code- 30d == 30 days

Backend🡪.env 🡺 we will add this line

JWT\_SECRET = justasecretsentence

Backend🡪 create a new folder: utills 🡪 create a new file: generateToken.js🡺

import jwt from 'jsonwebtoken'

const generateToken = (id) =>{

    return jwt.sign({id}, process.env.JWT\_SECRET,{expiresIn:'30d'})

}

export default generateToken

backend🡪controllers 🡪usercontrollers.js🡺

we will generate the token after the user authenticates!

import User from '../models/userModel.js'

import generateToken from '../utills/generateToken.js'

import asyncErrorhandler from 'express-async-handler' //an npm pack for handling errors instead of using try catch

//this is a post request to route: api/users/login

const authUser = asyncErrorhandler(async (req,res)=>{

   const {email,password} = req.body //extract the email & password from the body!

   //res.send({email,password}) //we have access to inputs from the body

    const user = await User.findOne({email: email})

    //we need to use bcypt beacuse we store the passwords in the DB in an encrypted from

   //we also constructed a method for decrypting the password "matchPassword", you can it in userModel.js

    if(user && await user.matchPassword(password)){

        res.json({

            \_id: user.\_id,

            name: user.name,

            email: user.email,

            isAdmin: user.isAdmin,

            token: generateToken(user.\_id)

            // token: null //soon we'll deal with tokens, for now null

        })

    }else{

        res.status(401)

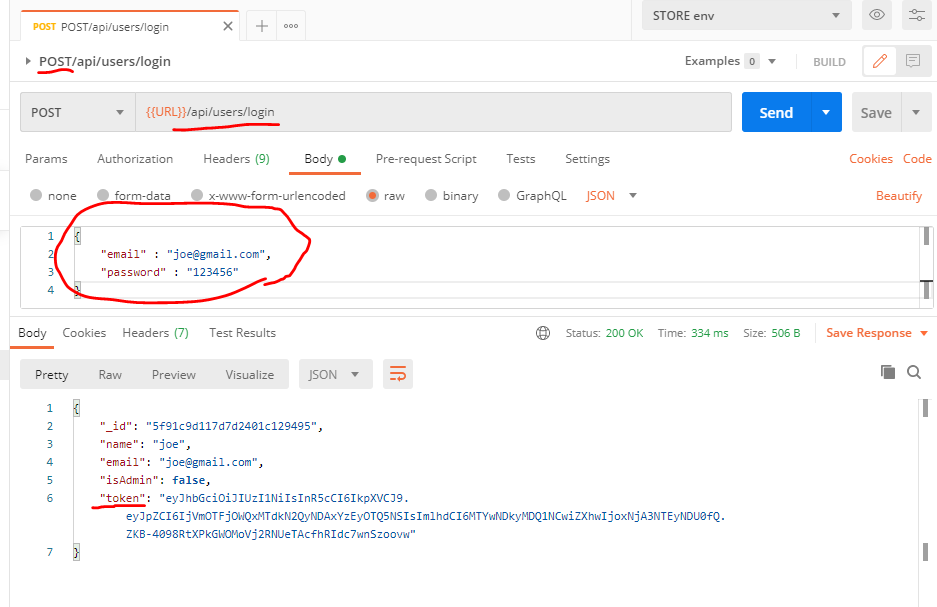
        throw new Error('Invalid Email / Password')

    }

})

export {authUser}

**postman-generate a token**



Now we will try to log in with an existing user using the post method we constructed: we will see that the response consists a token string!

We can copy the token to the JWT site’s Encoder and decipher the token’s payload :

{

"id": "5f91c9d117d7d2401c129495",// the actual user ID in the DB

"iat": 1604920454,// generated at …

"exp": 1607512454 //30 days

}

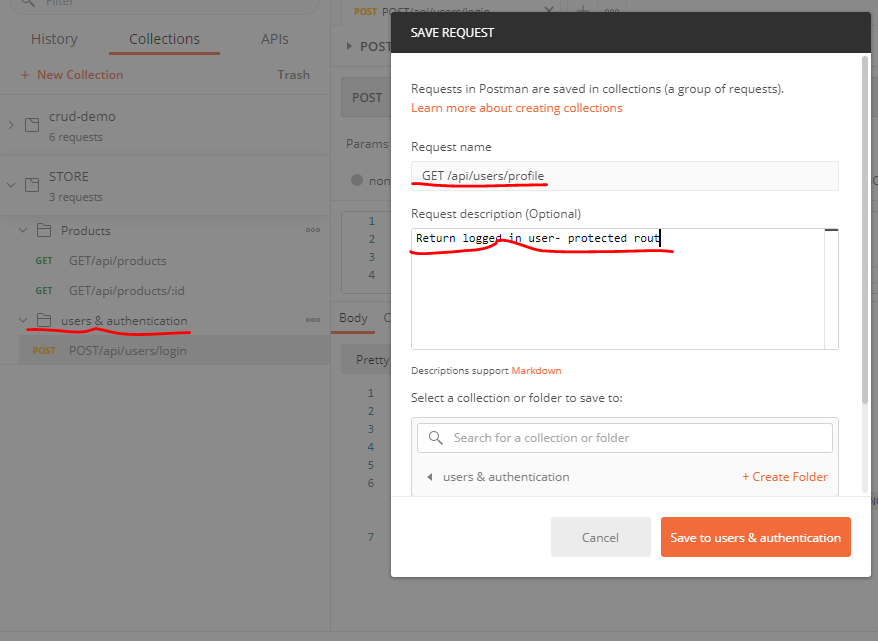
**Now we can use this token to access protected routes- copy it !**

BUT how we do it ?

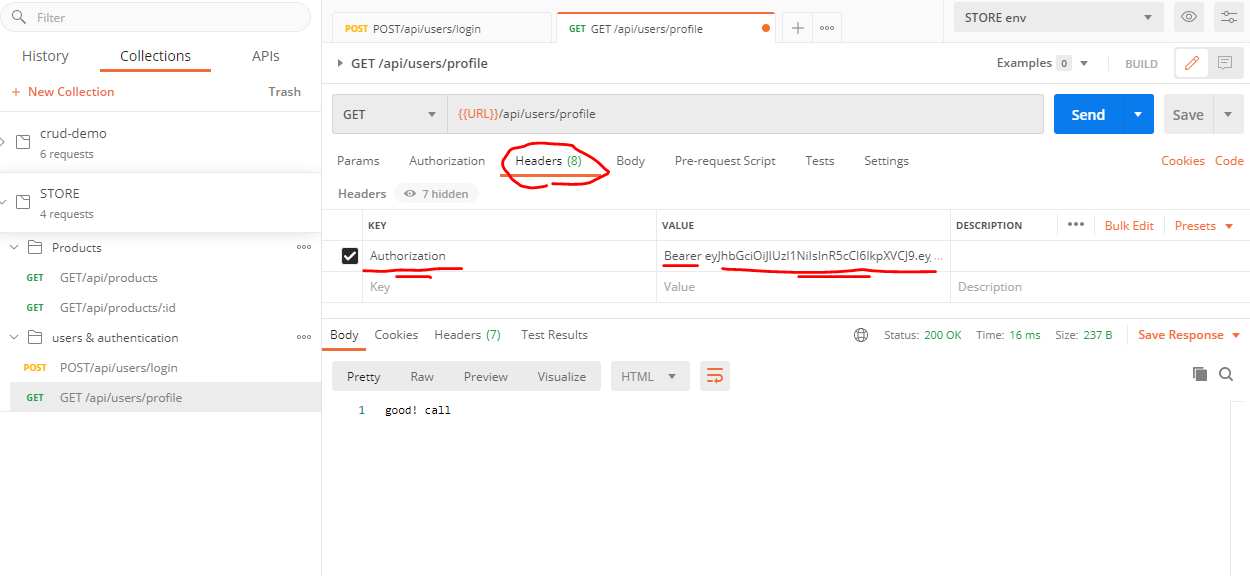
PROTECTED ROUTES & TOKENS

**POSTMAN**

Before we start with coding , first lets construct a get request in postman for the logged user profile, it will be a protected route.



The request: don’t forget to write Bearer before the token. (don’t send it yet)



backend🡪routes 🡪userRoutes.js🡺

import express from 'express'

const router = express.Router()// api/users/login

import {authUser,getUserProfile} from '../controllers/userControllers.js'

//fetch users email and password from the DOM(body)

router.post('/login',authUser)

router.route('/profile').get(getUserProfile)

export default router

backend🡪controllers 🡪usercontrollers.js🡺

import User from '../models/userModel.js'

import generateToken from '../utills/generateToken.js'

import asyncErrorhandler from 'express-async-handler' //an npm pack for handling errors instead of using try catch

//this is a post request to route: api/users/login

const authUser = asyncErrorhandler(async (req,res)=>{

   const {email,password} = req.body //extract the email & password from the body!

   //res.send({email,password}) //we have access to inputs from the body

    const user = await User.findOne({email: email})

    //we need to use bcypt beacuse we store the passwords in the DB in an encrypted from

   //we also constructed a method for decrypting the password "matchPassword", you can it in userModel.js

    if(user && await user.matchPassword(password)){

        res.json({

            \_id: user.\_id,

            name: user.name,

            email: user.email,

            isAdmin: user.isAdmin,

            token: generateToken(user.\_id)

            // token: null //soon we'll deal with tokens, for now null

        })

    }else{

        res.status(401)

        throw new Error('Invalid Email / Password')

    }

})

//this is a GET request to the protected route: api/users/profile

const getUserProfile = asyncErrorhandler(async (req,res)=>{

 // const user = await User.findById({req.user.\_id})

    res.send('good! call')

 })

export {authUser,getUserProfile}

req.user.\_id -> will not work yet.

**Create the protected routes middleware**

Backend🡪middleware🡪create a new file: authMiddleware.js

Remember it’s a middleware so it must be a functions with **(req, res,next) =>{}**

req.header.authorization-> will contain the token, we provided in p.142 with postman GET request to route: /api/users/profile , in the header!

import jwt from 'jsonwebtoken'

import User from '../models/userModel.js'

const protect  = async(req, res,next) =>{

    let token

    console.log(req.headers.authorization) //will log in our console the exact token we sent through postman headers

    next()

}

export{ protect }

**now let’s implement the “protect” middleware on the route**

backend🡪routes🡪userRoutes.js :

import express from 'express'

const router = express.Router()// api/users/login

import {authUser,getUserProfile} from '../controllers/userControllers.js'

import {protect} from '../middleware/authMiddleware.js'

//routes protecting middleware

//fetch users email and password from the DOM(body)

router.post('/login',authUser)

//routes protecting middleware will protect the /profile route by adding it to the get request as its first argument!

router.route('/profile').get(protect, getUserProfile)

export default router

so now, when we send the get request with postman we will

see the response as written in p.144

    res.send('good! call')

while in the terminal we will see the token as passed in postman headers:

Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9 .eyJpZCI6IjVmOTFjOWQxMTdkN2QyNDAxYzEyOTQ5NSIsImlhdCI6MTYwNDkyMDQ1NCwiZXhwIjoxNjA3NTEyNDU0fQ.ZKB-4098RtXPkGWOMoVj2RNUeTAcfhRIdc7wnSzoovw

Don’t forget! the structure is: Bearer -space- tokenstring

**Now we will add more functionality to the authMiddleware:**

Note: this one is complicated so make sure you read all the notes in the code.

import jwt from 'jsonwebtoken'

import asyncErrorHandler from 'express-async-handler'

import User from '../models/userModel.js'

const protect = asyncErrorHandler(async(req, res,next) =>{ //we wrap the midleware with express async error handler

    let token

    //console.log(req.headers.authorization)  //the token passed through the headers- Bearer \*space\* tokenString.

    if(req.headers.authorization &&

        req.headers.authorization.startsWith('Bearer')){

        //only if there is a token and it starts with Bearer we continue.

         try{

             //we need to extract from the headers only the token string excluding the space and 'Bearer' so we will split the string at the space between the token and Bearer.

             token = req.headers.authorization.split(' ')[1]

             const decoded = jwt.verify(token, process.env.JWT\_SECRET)

             //console.log(decoded) //will log: { id: '5f91c9d117d7d01c129495', iat: 1604920454, exp: 1607512454 }

             //so we can gain access to this users id as it appears in the DB

             //assigning user data to req.user!!!

req.user = await  User.findById(decoded.id).select('- password')

//.select('-password') so we will

 store all users data from the DB in this variable except for his password

             next()

            }catch(error){

                console.error(error)

                res.status(401)

                throw new Error('Not authrized, token faild verification')

         }

    }

    if(!token){

        res.status(401)

        throw new Error('Not authrized, no token in headers')

    }

})

export{ protect }

now we can use protect for all the routes we want!

The user’s profile data will be accessed in the user controller via req.user

**Final touch in user controller:**

backend🡪controllers 🡪usercontrollers.js🡺

import User from '../models/userModel.js'

import generateToken from '../utills/generateToken.js'

import asyncErrorhandler from 'express-async-handler' //an npm pack for handling errors instead of using try catch

//this is a post request to route: api/users/login

const authUser = asyncErrorhandler(async (req,res)=>{

   const {email,password} = req.body //extract the email & password from the body!

   //res.send({email,password}) //we have access to inputs from the body

    const user = await User.findOne({email: email})

    //we need to use bcypt beacuse we store the passwords in the DB in an encrypted from

   //we also constructed a method for decrypting the password "matchPassword", you can it in userModel.js

    if(user && await user.matchPassword(password)){

        res.json({

            \_id: user.\_id,

            name: user.name,

            email: user.email,

            isAdmin: user.isAdmin,

            token: generateToken(user.\_id)

            // token: null //soon we'll deal with tokens, for now null

        })

    }else{

        res.status(401)

        throw new Error('Invalid Email / Password')

    }

})

//this is a GET request to the protected route: api/users/profile

const getUserProfile = asyncErrorhandler(async (req,res)=>{

    //in authMiddleware.js we store all user data that's passed the authentication and authorization proccess in req.user (excluding his password)

    const user = await User.findById(req.user.\_id)

   // res.send('good! call')

   if(user){

     res.json({

        \_id: user.\_id,

        name: user.name,

        email: user.email,

        isAdmin: user.isAdmin,

     })

   }else{

       res.status(404)

       throw new Error('user not found')

   }

 })

export {authUser,getUserProfile}

**summary JWT-TOKEN-authentication-authorization-rout protecting middleware**

now when we will run the GET request /api/users/profile on postman , we will be given access to the users data.

all we had to do is pass a token in the headers. (never forget to write Bearer )

The token must be correlated to a user’s id in the DataBase.

The token, in the first place was created using a hash function of JWT on the users ID

So user will have access to this rout only if he can pass the token cuz this rout is protected via the authmiddleware!!!

In the middleware, after authorization granted via token verification 🡺

We assign the specific user data to req.user

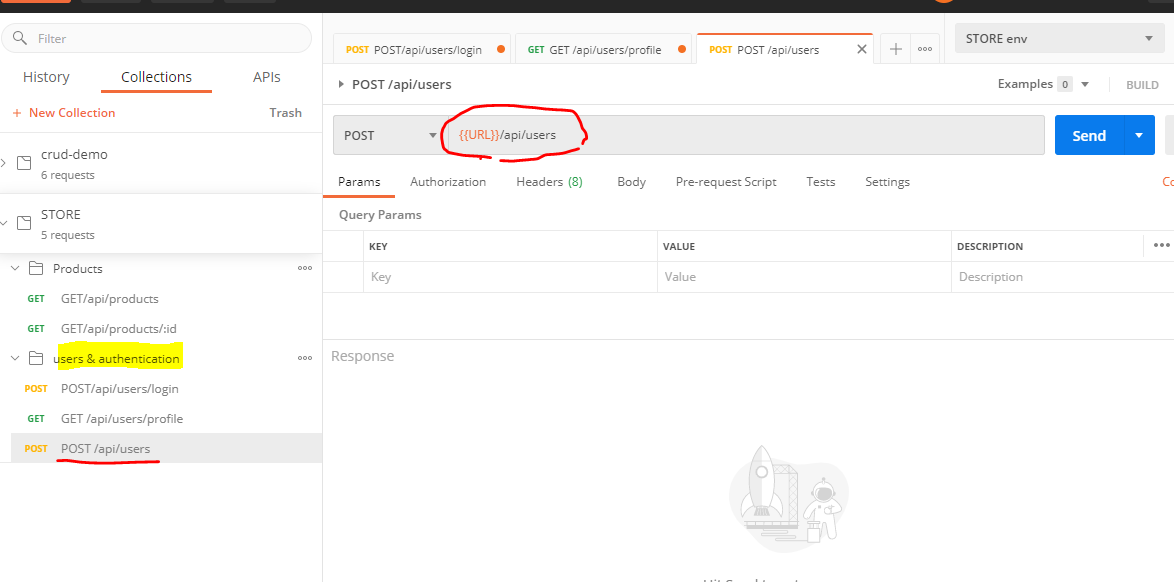
**USER REGISTRATION & PASSWORD ENCRYPTION**

BASICALLY, NOW WE WANT TO CREATE A NEW USER IN THE DB:

**POSTMAN**

Let’s construct the postman request:

It will be a POST request to /api/users



backend🡪controllers 🡪usercontrollers.js🡺

add this code:

 //this is a post request to route: api/users

 //in order to create a new user in the DB, this is a public route!

const registerUser = asyncErrorhandler(async (req,res)=>{

    const {email,password, name} = req.body //extract the name email and password from the body!

    //if the email exists in the DB user wont be able to register to our system

     const userExists = await User.findOne({email: email})

     if(userExists){

         res.status(400)

         throw new Error('Cannot register, email exists in DB')

     }

     const user = await User.create({

         name,

         email,

         password //soon we will encrypt it see p. 154

     })

     if(user){

         res.status(201).json({

             //as user is created we immediatly send his data in a response including the token so he will be able to get authorization

//you will get access to this data if email and password authorized when logging in as well:

            \_id: user.\_id,

            name: user.name,

            email: user.email,

            isAdmin: user.isAdmin,

            token: generateToken(user.\_id)

         })

     }else{

         res.status(400)

         throw new Error('User could not be created- invalid data')

     }

 })

export {authUser,getUserProfile, registerUser} //don’t forget to export it!

backend🡪routes🡪userRoutes.js :

import express from 'express'

const router = express.Router()// api/users/

import {authUser,getUserProfile, registerUser} from '../controllers/userControllers.js'

import {protect} from '../middleware/authMiddleware.js' //routes protecting middleware

//fetch users email and password from the DOM(body)

router.post('/login',authUser) // api/users/login

//routes protecting middleware will protect the /profile route by adding it to the get request as its first argument!

router.route('/profile').get(protect, getUserProfile) // api/users/profile

//register a new user route

router.route('/').post(registerUser) // api/users/

export default router

one last step before checking the results in postman!

We need to encrypt the password! BUT, we need to do so on the schema in user model, just before we save the new user data in the DB plus we will use a middleware for that!

Backend🡪 models 🡪userModel.js🡺

import mongoose from 'mongoose';

import bcrypt from 'bcryptjs'

const userSchema = mongoose.Schema({

    name:{

        type: String,

        required: true

    },

    email:{

        type: String,

        required: true,

        unique: true

    },

    password:{

        type: String,

        required: true

    },

    isAdmin:{

        type: Boolean,

        required: true,

        default: false

    }

},{

    //mongoose built in time stamps for "created at"/ "updated at"

    timestamps: true

})

userSchema.methods.matchPassword = async function(enteredPassword){ //we assign a method manualy to the user schema

 return await bcrypt.compare(enteredPassword, this.password)

}//we will use this method "matchPassword" in the userController

userSchema.pre('save', async function(next){

    if(!this.isModified('password')){

// this part of the code is a preparation to edit user profile.

        next()

    }

    const salt = await bcrypt.genSalt(10) //we use await cuz it returns a  promise

    this.password = await bcrypt.hash(this.password, salt)

    //this refers to the user we are just creating! meaning we store in the password key a new password value, which is the hashed one !

})

const User = mongoose.model('User', userSchema)

export default User

**Edit user profile** – in the future we will have an “edit profile” option which will enable the user to edit his name email or even password , this part of the code it a preparation due to the fact that we are checking🡪 if the user updated his profile but didn’t update his password !this.isModified('password')- this is a mongoose method

Than there is no need to hash the password again therefore we call next() !

But if it has changed or this profile just been created it will be hashed!

Back to  
 backend🡪controllers 🡪usercontrollers.js🡺

     const user = await User.create({

         name,

         email,

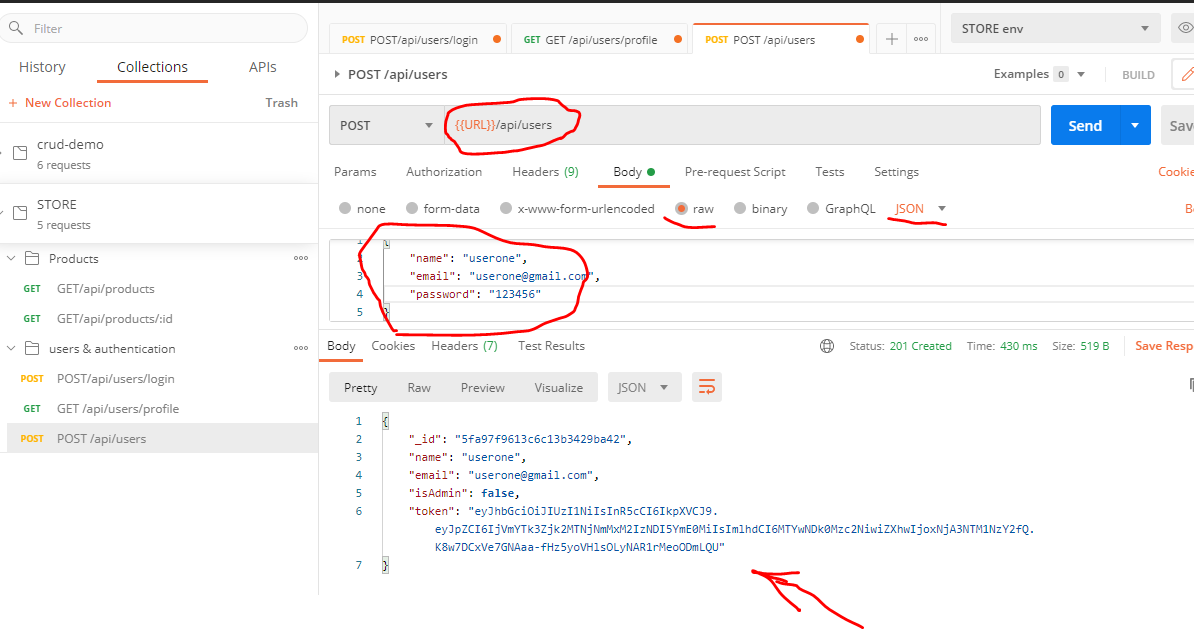
         password

     })

So as the User.create runs, it will run the schema.pre (see p.152) function which will encrypt it and then save it to the DB!

postman:

lets run the request (creating a new user…)



We will get all the parameters except the password, but if you visit your DB (using compass) you will see the password is hashed!!!

Login and signup- frontend

After we implemented the backend of the authentication and authorization we now go back to assemble the login and register pages, we will also use redux.

**Login page**

**Redux setup:**

**Constants:**

Frontend🡪src🡪constatnts🡪create a new file: userConstants.js🡺

export const USER\_LOGIN\_REQUEST = 'USER\_LOGIN\_REQUEST'

export const USER\_LOGIN\_SUCCESS = 'USER\_LOGIN\_SUCCESS'

export const USER\_LOGIN\_FAIL = 'USER\_LOGIN\_FAIL'

export const USER\_LOGOUT = 'USER\_LOGOUT'

**Reducers:**

Frontend🡪src🡪reducers🡪create a new file: userReducers.js🡺

import {

      USER\_LOGIN\_REQUEST,

  USER\_LOGIN\_SUCCESS ,

  USER\_LOGIN\_FAIL ,

  USER\_LOGOUT ,

   } from '../constants/userConstants'

export const userLoginReducer  = ( state = {}, action ) => {

 //this is the reducer function we export it!

//the initial state will be an empty object.

switch( action.type ){

   case  USER\_LOGIN\_REQUEST:

       return {loading: true } // fetching data ...

   case USER\_LOGIN\_SUCCESS:

           return {loading: false, userInfo: action.payload} //if fech success...

   case USER\_LOGIN\_FAIL:

           return {loading: false, error: action.payload} //if fech failed...

   case USER\_LOGOUT:

           return {} //we will deal with that later

     default:

       return state //pass the state as is.

}

**store:**

Frontend🡪src🡪store.js 🡺

Now we will update the store’s state to include the user login as well:

import {createStore, combineReducers, applyMiddleware } from 'redux'

import thunk from 'redux-thunk'

import {composeWithDevTools} from 'redux-devtools-extension'

import {productsListReducer, productCardReducer } from './reducers/productsReducers'

import {cartReducer} from './reducers/cartReducers'

import {userLoginReducer} from './reducers/userReducers'

const reducer  = combineReducers({ //reducer is part of the store's inputs

    productsList: productsListReducer,

    productDetails: productCardReducer,

    cart: cartReducer,

    userLogin: userLoginReducer

})

const cartLocalStorage = localStorage.getItem('cartItems') ? JSON.parse( localStorage.getItem('cartItems')) : []

const initialState = {

    cart:{cartItems: cartLocalStorage}

}

const middleWare = [thunk]

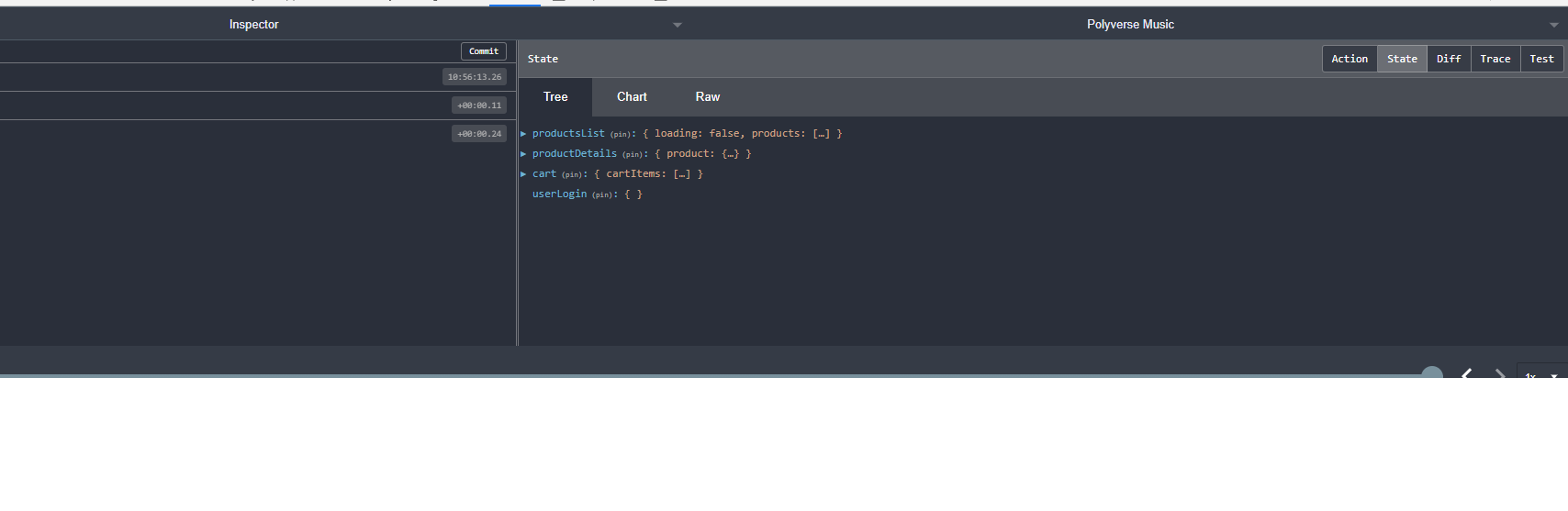
//store is being consumed all accross the app. see index.js

const store = createStore( reducer, initialState, composeWithDevTools(applyMiddleware(...middleWare)))

export default store

now when we will re render the project (>> npm run dev)

we can see in the redux devtools the state now includes the user login, and it’s for now an empty object.



**actions:**

Frontend🡪src🡪actions🡪create a new file: userActions.js🡺

Here we are going to make a request from the server for the user’s token- that’s will enable us to gain access to the user info. In order to get it we must provide an email and a password- if valid you will get access to the user’s data!

You may see the data you will access via server's userController.js file.

import axios from 'axios'

import {

    USER\_LOGIN\_REQUEST,

USER\_LOGIN\_SUCCESS ,

USER\_LOGIN\_FAIL ,

USER\_LOGOUT ,

 } from '../constants/userConstants'

export const login = (email,password) => async (dispatch)=>{

 //here we also use the redux-thunk to make an asynchronous request.

 try{

    dispatch({

        type: USER\_LOGIN\_REQUEST

    })

    const config = { //setup – headers for the post request

        headers:{

            'Content-type':'application/json'

        }

    }

    const {data} = await axios.post('/api/users/login',{email,password}, config)

 //the data we are going to get back may be found on the server's user controller file;

 //it's the user name email id and his token

    dispatch({

     type: USER\_LOGIN\_SUCCESS,

     payload: data,

    })

//we are going to save the users data on local storage as userinfo, we will also have to update store.js with this feature!

    localStorage.setItem('userInfo',JSON.stringify(data))

}catch (error){

    dispatch({type: USER\_LOGIN\_FAIL,

        payload:

        error.response && error.response.data.message

        ?

         error.response.data.message

        :

        error.message

         })

 }

}

**store:**

Frontend🡪src🡪store.js 🡺

Now we will update the store’s state to include the user information if stored on local storage:

import {createStore, combineReducers, applyMiddleware } from 'redux'

import thunk from 'redux-thunk'

import {composeWithDevTools} from 'redux-devtools-extension'

import {productsListReducer, productCardReducer } from './reducers/productsReducers'

import {cartReducer} from './reducers/cartReducers'

import {userLoginReducer} from './reducers/userReducers'

const reducer  = combineReducers({ //reducer is part of the store's inputs

    productsList: productsListReducer,

    productDetails: productCardReducer,

    cart: cartReducer,

    userLogin: userLoginReducer

})

const cartLocalStorage = localStorage.getItem('cartItems') ? JSON.parse( localStorage.getItem('cartItems')) : []

const userInfoLocalStorage = localStorage.getItem('userInfo') ? JSON.parse( localStorage.getItem('userInfo')) : null

const initialState = {

    cart:{cartItems: cartLocalStorage},

    userLogin:{ userInfo: userInfoLocalStorage}

}

const middleWare = [thunk]

//store is being consumed all accross the app. see index.js

const store = createStore( reducer, initialState, composeWithDevTools(applyMiddleware(...middleWare)))

export default store

**Login Screen frontend :**

Frontend🡪 src🡪screens🡪create a new file: LoginScreen.js🡺

Create a new function component (“rafce” short writing):

import React,{useState,useEffect} from 'react'

import {useDispatch, useSelector} from 'react-redux' //so we can access the redux app level state

import {Link} from 'react-router-dom'

import {Form, Button, Row, Col} from 'react-bootstrap'

import Message from '../components/Message'

import Loader from '../components/Loader'

import {login} from '../actions/userActions'

const LoginScreen = () => {

    const [email,setEmail] = useState('')  //local state

    const [password,setPassword] = useState('') //local state

    return (

        <div>

        </div>

    )

}

export default LoginScreen

login form = in order to save time, we will now create another component which will be used couple of times across the app, it will be a simple one, don’t worry.

Frontend🡪 src🡪components🡪create new file formContainer.js🡺

import React from 'react'

import {Container, Row, Col} from 'react-bootstrap'

const FormContainer = ({children}) => { //we destructure from props the children object

    return (

        <Container>

            <Row clasName='justify-content-md-center'>

                <Col xs={12} md={6}>

                    {children}

                </Col>

            </Row>

        </Container>

    )

}

export default FormContainer

Back to:

Frontend🡪 src🡪screens🡪 LoginScreen.js🡺

Let’s create the login form

import React,{useState,useEffect} from 'react'

import {useDispatch, useSelector} from 'react-redux' //so we can access the redux app level state

import {Link} from 'react-router-dom'

import {Form, Button, Row, Col} from 'react-bootstrap'

import Message from '../components/Message'

import Loader from '../components/Loader'

import FormContainer from'../components/FormContainer'

import {login} from '../actions/userActions'

const LoginScreen = () => {

    const [email,setEmail] = useState('')  //local state

    const [password,setPassword] = useState('') //local state

    return (

        //let's render the login from

        <FormContainer >

            <h1>Sign In</h1>

            <Form onSubmit = {submitHandler}>

                <Form.Group controlId='email'>

                    <Form.Label>Email</Form.Label>

                    <Form.Control

                        type='email'

                        placeholder='Enter Email'

                        value={email}

                        onChange={(event)=> setEmail(event.target.value)}

                        /\*whatever we typein will constantly be updated to the local state \*/

                        >

                    </Form.Control>

                </Form.Group>

                <Form.Group controlId='password'>

                    <Form.Label>Password</Form.Label>

                    <Form.Control

                        type='password'

                        placeholder='Enter password'

                        value={password}

                        onChange={(event)=>setPassword(event.target.value)}/\*whatever we typein will constantly be updated to the local state \*/

                        >

                    </Form.Control>

                </Form.Group>

                <Button type='submit' vatriant='primary'>Sign In</Button>

            </Form>

            <Row className='py-3'>

                <Col>

                 New Here?

                </Col>

                <Col>

                    Sign Up, it's totally free!

                    <Link to= { redirect ? `/register?redirect=${redirect}` : '/register' }>

                      Register

                    </Link>

                </Col>

            </Row>

        </FormContainer>

    )

}

export default LoginScreen

<Link to={redirect ? `/register?redirect=${redirect}` : '/register' }>

                    Registre

                </Link>

//if we have a redirect value we will go to that one , else we will go to the /register URL

//soon he will write the code for submitHandler & redirect variable

Now in order to see the form we must connect this component to a route!

Frontend🡪 src🡪App.js🡺

import React from 'react'

import { BrowserRouter as Router, Route } from 'react-router-dom'

import { Container } from 'react-bootstrap'

import Header from './components/Header'

import Footer from './components/Footer'

import HomeScreen from './screens/HomeScreen'

import ProductScreen from './screens/ProductScreen'

import CartScreen from './screens/CartScreen'

import LoginScreen from './screens/LoginScreen'

const App = () => {

  return (

    <Router>

      <div>

        <Header style={{ position: 'relative' }} />

        <main className='py-4 '>

          <Container>

            <Route path='/login' component={LoginScreen} />

            <Route path='/' component={HomeScreen} exact />

            <Route path='/products/:id' component={ProductScreen} />

            <Route path='/cart/:id?' component={CartScreen} />

          </Container>

        </main>

        <Footer />

      </div>

    </Router>

  )

}

export default App

Back to:

Frontend🡪 src🡪screens🡪 LoginScreen.js🡺

Let’s fix the login form’s submitHandler & redirect variable

User got to login page from product’s add to cart Button-> cart-procced to checkout Button-If the user is already logged in we will want to redirect him to the shipping method.

User got to login page from navBarafter he finishes login process we want him to be re directed to products page.

import React,{useState,useEffect} from 'react'

import {useDispatch, useSelector} from 'react-redux' //so we can access the redux app level state

import {Link} from 'react-router-dom'

import {Form, Button, Row, Col} from 'react-bootstrap'

import Message from '../components/Message'

import Loader from '../components/Loader'

import FormContainer from'../components/FormContainer'

import {login} from '../actions/userActions'

const LoginScreen = ({location, history}) => { //destructure location & history out of props

    const [email,setEmail] = useState('')  //local state

    const [password,setPassword] = useState('') //local state

    const dispatch = useDispatch()

    const userLogin = useSelector(state=>state.userLogin) //from the store's state

    const {loading,error, userInfo} = userLogin //destructre

    const redirect = location.search ? location.search.split('=')[1] : '/'

    useEffect(()=>{

        if(userInfo){

            history.push(redirect) //if userInfo exists we will be redirected

        }

    },[history,userInfo, redirect])

    const submitHandler = (event)=>{

        event.preventDefault() //so the page wont be refreshed

        //dispatch login

        dispatch(login(email,password)) //we execute the login action with the inputs the user gave us!

    }

    return (

        //let's render the login from

        <FormContainer className='p-4' >

            <h1>Sign In</h1>

            {

            error && <Message>{error}</Message>//if an error exists render a mesage

            }

            {

            loading && <Loader/>//if  loading the data ...

            }

            <Form onSubmit = {submitHandler}> //when submitting form

                <Form.Group controlId='email'>

                    <Form.Label>Email</Form.Label>

                    <Form.Control

                        type='email'

                        placeholder='Enter Email'

                        value={email}

                        onChange={(event)=> setEmail(event.target.value)}

                        /\*whatever we typein will constantly be updated to the local state \*/

                        >

                    </Form.Control>

                </Form.Group>

                <Form.Group controlId='password'>

                    <Form.Label>Password</Form.Label>

                    <Form.Control

                        type='password'

                        placeholder='Enter password'

                        value={password}

                        onChange={(event)=>setPassword(event.target.value)}/\*whatever we typein will constantly be updated to the local state \*/

                        >

                    </Form.Control>

                </Form.Group>

                <Button type='submit' vatriant='success'>Sign In</Button>

            </Form>

            <Row className='py-3'>

                <Col>

                 New Here?

                </Col>

                <Col>

                    Sign Up, it's totally free!

                    <Link to= { redirect ? `/register?redirect=${redirect}` : '/register' }>

                      Register

                    </Link>

                </Col>

            </Row>

        </FormContainer>

    )

}

export default LoginScreen

**Changing the navbar sign-in link when the user is logged in**

Frontend🡪src🡪components🡪Header.js🡺

In order to do that we must check the state of the userLogin object in redux state and according to that, render the sign in link or log out. We will use the useSelector and useDispatch HOOKS.

We will make a dropdown menu with the users data.

import React from 'react'

import{useDispatch, useSelector} from 'react-redux'

import { Navbar, Nav, Container, NavDropdown } from 'react-bootstrap'

import { LinkContainer } from 'react-router-bootstrap'

import mainLogo from './mainlogo.png'

import {logout} from '../actions/userActions' //soon we will code it

const Header = () => {

  const dispatch = useDispatch()

  const userLogin =  useSelector(state => state.userLogin)

  const {userInfo} = userLogin

  const logoutHandler = ()=>{console.log('logout')}

  return (

    <header>

      <Navbar bg='light' expand='lg' fixed='top' collapseOnSelect>

        <Container >

          <LinkContainer to='/'>

            <Navbar.Brand>

              <img

                src={mainLogo}

                className='d-inline-block align-top'

                alt='Polyverse logo'

              />

            </Navbar.Brand>

          </LinkContainer>

          <Navbar.Toggle aria-controls='basic-navbar-nav' />

          <Navbar.Collapse id='basic-navbar-nav'>

            <Nav className='ml-auto'>

              <LinkContainer to='/cart'>

                <Nav.Link varient='dark'>

                  <h4>

                    <i className='fas fa-shopping-cart p-3'></i> Cart

                  </h4>

                </Nav.Link>

              </LinkContainer>

              {userInfo ? (

                <div style={{display:'flex', alignItems:'center'}}>

               <NavDropdown title={userInfo.name} id='username' className='h4'>

                  <NavDropdown.Item className='navdrop' href="/profile">Profile</NavDropdown.Item>

                  <NavDropdown.Divider />

                  <NavDropdown.Item className='navdrop' onClick={logoutHandler}>Log Out</NavDropdown.Item>

               </NavDropdown>

                </div>

              ) :

              <LinkContainer to='/login'>

                <Nav.Link varient='dark'>

                  <h4>

                    <i className='fas fa-user-astronaut p-3'></i> Sign In

                  </h4>

                </Nav.Link>

              </LinkContainer>

              }

            </Nav>

          </Navbar.Collapse>

        </Container>

      </Navbar>

    </header>

  )

}

export default Header

Frontend🡪src🡪index.css

Had to make some adjusments…

.navdrop {

  background-color: transparent;

  font-size: medium;

}

.navdrop:hover {

  background-color: #00b1ff;

}

**logoutHandler**

in order to complete the log out process, we must setup redux for it.

Frontend🡪src🡪actions🡪userActions.js🡺

import {

    USER\_LOGIN\_REQUEST,

USER\_LOGIN\_SUCCESS ,

USER\_LOGIN\_FAIL ,

USER\_LOGOUT ,

 } from '../constants/userConstants'

At the end of the page we will add this code:

export const logout = ()=> (dispatch)=>{

    localStorage.removeItem('userInfo')

    dispatch({type: USER\_LOGOUT})

document.location.href = '/login' //redirecting the user to login page after logout

}

We don’t need to setup a reducer but make sure your user reducer looks like that!

import {

      USER\_LOGIN\_REQUEST,

  USER\_LOGIN\_SUCCESS ,

  USER\_LOGIN\_FAIL ,

  USER\_LOGOUT

   } from '../constants/userConstants'

export const userLoginReducer  = ( state = {}, action ) => { //this is the reducer function we export it!

switch( action.type ){

   case  USER\_LOGIN\_REQUEST:

       return {loading: true } // fetching data ...

   case USER\_LOGIN\_SUCCESS:

           return {loading: false, userInfo: action.payload} //if fech success...

   case USER\_LOGIN\_FAIL:

           return {loading: false, error: action.payload} //if fech failed...

   case USER\_LOGOUT:

                return {}

     default:

       return state //pass the state as is.

}}

Frontend🡪src🡪components🡪Header.js🡺

Now let’s edit the logout handler:

 const logoutHandler = ()=>{dispatch(logout())} //the log out action fired off and the user is logged out by removing the userinfo fron the local storage

and let’s add some css magic to LoginScreen:

import React,{useState,useEffect} from 'react'

import {useDispatch, useSelector} from 'react-redux' //so we can access the redux app level state

import {Link} from 'react-router-dom'

import {Form, Button, Row, Col} from 'react-bootstrap'

import Message from '../components/Message'

import Loader from '../components/Loader'

import FormContainer from'../components/FormContainer'

import {login} from '../actions/userActions'

const LoginScreen = ({location, history}) => { //destructure location & history out of props

    const [email, setEmail] = useState('')  //local state

    const [password, setPassword] = useState('') //local state

    const dispatch = useDispatch()

    const userLogin = useSelector(state=> state.userLogin) //from the store's state

    const {loading, error, userInfo} = userLogin //destructre

    const redirect = location.search ? location.search.split('=')[1] : '/';

    useEffect(() => {

        if(userInfo){

           history.push(redirect) //if userInfo exists we will be redirected

        }

    }, [history , userInfo , redirect])

    const submitHandler = (event)=>{

        event.preventDefault() //so the page wont be refreshed

        //dispatch login

        dispatch(login(email, password)) //we execute the login action with the inputs the user gave us!

    }

    return (

        //let's render the login from

        <FormContainer>

            <div style={{paddingTop:"5%"}}><h1>Enter The Polyverse</h1></div>

            {

            loading && <Loader/>//if  loading the data ...

            }

            <Form onSubmit = {submitHandler}>

                <Form.Group controlId='email'>

                    <Form.Control

                        type='email'

                        placeholder='Email'

                        value={email}

                        onChange={(event)=> setEmail(event.target.value)}

                        /\*whatever we typein will constantly be updated to the local state \*/

                        >

                    </Form.Control>

                </Form.Group>

                <Form.Group controlId='password'>

                    <Form.Control

                        type='password'

                        placeholder='Password'

                        value={password}

                        onChange={(event)=>setPassword(event.target.value)}/\*whatever we typein will constantly be updated to the local state \*/

                        >

                    </Form.Control>

                </Form.Group>

                {

                   error &&  <Message>{error}</Message>//if an error exists render a mesage

                 }

                <Button type='submit' variant="warning" block>Sign In</Button>

            </Form>

            <Row className='py-3 d-flex justify-content-center'>

              <h3> New Here?&nbsp;

              <Link to= { redirect ? `/register?redirect=${redirect}` : '/register' }>

                 Register

               </Link>

               </h3>

            </Row>

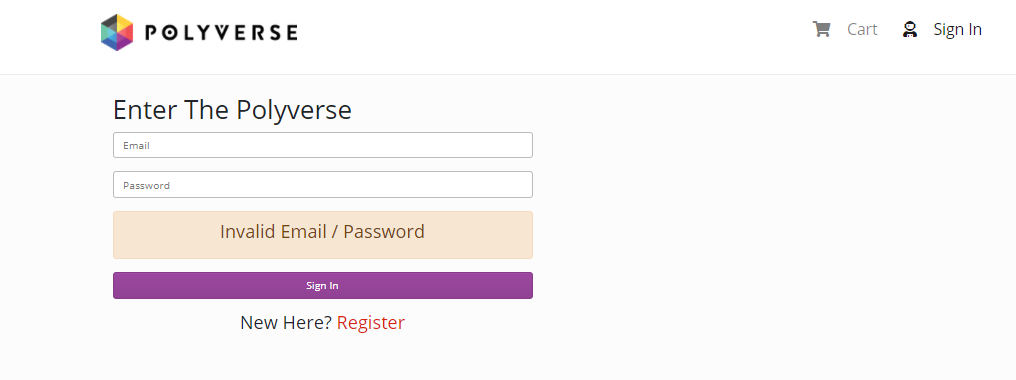
        </FormContainer>

    )

}

export default LoginScreen

so it will look more like that:



**The Register form**

If a user is not a member, he will get to the register form / screen from the login

But before all that we must setup redux for that.

**Constants**

Frontend🡪src🡪constants🡪userConstatns.js🡺

Add register related constants:

export const USER\_REGISTER\_REQUEST = 'USER\_REGISTER\_REQUEST'

export const USER\_REGISTER\_SUCCESS = 'USER\_REGISTER\_SUCCESS'

export const USER\_REGISTER\_FAIL = 'USER\_REGISTER\_FAIL'

**Reducers**

Frontend🡪src🡪reducers🡪 userReducers.js🡺

Add this code, it’s very similar to the login reducer, except the log out case.

(plus don’t forget to import the constatnts)

export const userRegisterReducer  = ( state = {}, action ) => { //this is the reducer function we export it!

        switch( action.type ){

           case  USER\_REGISTER\_REQUEST:

               return {loading: true } // fetching data ...

           case USER\_REGISTER\_SUCCESS:

                   return {loading: false, userInfo: action.payload} //if fech success...

           case USER\_REGISTER\_FAIL:

                   return {loading: false, error: action.payload} //if fech failed...

             default:

               return state //pass the state as is.

        }}

**Store**

Frontend🡪src🡪store.js🡺

Update the combined reducer to contain the user register reducer dependency.

Add this code:

import {userLoginReducer,userRegisterReducer} from './reducers/userReducers'

const reducer  = combineReducers({ //reducer is part of the store's inputs

    productsList: productsListReducer,

    productDetails: productCardReducer,

    cart: cartReducer,

    userLogin: userLoginReducer,

    userRegister: userRegisterReducer,

})

**Actions**

Frontend🡪src🡪actions🡪userActions.js🡺

(plus don’t forget to import the constatnts)

export const register = (name,email,password) => async (dispatch)=>{

    //here we also use the redux-thunk to make an asynchronous request.

    try{

       dispatch({

           type: USER\_REGISTER\_REQUEST

       })

       const config = {

           headers:{

               'Content-type':'application/json'

               //for posting data

           }

       }

       const {data} = await axios.post('/api/users',{name,email,password}, config)

    //we are going to post(!) data to the server, the route is /api/users.

    //if the data is valid we will get the user data (especiallly the token)

       dispatch({

        type: USER\_REGISTER\_SUCCESS,

        payload: data,

       })

 //after we registered successfully we must also dispatch a successful log in!

       //it will enable the change in the navBar (from sign in to the user’s name)

       dispatch({//this one is important!

        type: USER\_LOGIN\_SUCCESS,

        payload: data, //only if the email and pass, are valid

       })

   //we are going to save the users data on local storage as userinfo, we will also have to update store.js with this feature!

       localStorage.setItem('userInfo',JSON.stringify(data))

   }catch (error){

       dispatch({type: USER\_REGISTER\_FAIL,

           payload:

           error.response && error.response.data.message

           ?

            error.response.data.message

           :

           error.message

            })

    }

   }

**The register Screen**

Frontend🡪src🡪screens🡪create a new file: RegisterScreen.js🡺

Copy the data in LoginScreen and edit it:

import React,{useState,useEffect} from 'react'

import {useDispatch, useSelector} from 'react-redux' //so we can access the redux app level state

import {Link} from 'react-router-dom'

import {Form, Button, Row, Col} from 'react-bootstrap'

import Message from '../components/Message'

import Loader from '../components/Loader'

import FormContainer from'../components/FormContainer'

import {register} from '../actions/userActions'

const RegisterScreen = ({location, history}) => { //destructure location & history out of props

    const [email, setEmail] = useState('')  //local state

    const [password, setPassword] = useState('') //local state

    const [confirmPassword, setConfirmPassword] = useState('') //local state

    const [name, setName] = useState('') //local state

    const [message, setMessage] = useState(null) //local state

    const dispatch = useDispatch()

    const userRegister = useSelector(state=> state.userRegister) //from the store's state

    const {loading, error, userInfo} = userRegister //destructre

    const redirect = location.search ? location.search.split('=')[1] : '/';

    useEffect(() => {

        if(userInfo){

           history.push(redirect) //if userInfo exists we will be redirected(form submitted successfully)

        }

    }, [history , userInfo , redirect])

    const submitHandler = (event)=>{

        event.preventDefault() //so the page wont be refreshed

        if(password !== confirmPassword){

            setMessage('Passwords do not match')

        }else{

            //dispatch register

            dispatch(register(name,email, password))

        }

    }

    return (

        //let's render the login from

        <FormContainer>

            <div style={{paddingTop:"5%"}}><h1>Sign Up to become part of thousends of artists who makes magic with our products.</h1></div>

            {

            loading && <Loader/>//if  loading the data ...

            }

            <Form onSubmit = {submitHandler}>

            <Form.Group controlId='name'>

                   <Form.Control

                       type='name'

                       placeholder='Full Name'

                       value={name}

                       onChange={(event)=> setName(event.target.value)}

                       /\*whatever we typein will constantly be updated to the local state \*/

                       >

                   </Form.Control>

               </Form.Group>

                <Form.Group controlId='email'>

                    <Form.Control

                        type='email'

                        placeholder='Email'

                        value={email}

                        onChange={(event)=> setEmail(event.target.value)}

                        /\*whatever we typein will constantly be updated to the local state \*/

                        >

                    </Form.Control>

                </Form.Group>

                <Form.Group controlId='password'>

                    <Form.Control

                        type='password'

                        placeholder='Password'

                        value={password}

                        onChange={(event)=>setPassword(event.target.value)}/\*whatever we typein will constantly be updated to the local state \*/

                        >

                    </Form.Control>

                </Form.Group>

                <Form.Group controlId='confirmPassword'>

                    <Form.Control

                        type='password'

                        placeholder='Confirm Password'

                        value={confirmPassword}

                        onChange={(event)=>setConfirmPassword(event.target.value)}/\*whatever we typein will constantly be updated to the local state \*/

                        >

                    </Form.Control>

                </Form.Group>

                {

                   error &&  <Message>{message}</Message>//if an error exists render a mesage

                 }

                <Button type='submit' variant="warning" block>Sign In</Button>

            </Form>

            <Row className='py-3 d-flex justify-content-center text-center '>

              <h3> Have an Account? Good to have you back&nbsp;

              <Link to= { redirect ? `/login?redirect=${redirect}` : '/login' }>

                 Login

               </Link>

               </h3>

            </Row>

        </FormContainer>

    )

}

export default RegisterScreen

and before we check how it looks we need to add the route in App.js🡺

import React from 'react'

import { BrowserRouter as Router, Route } from 'react-router-dom'

import { Container } from 'react-bootstrap'

import Header from './components/Header'

import Footer from './components/Footer'

import HomeScreen from './screens/HomeScreen'

import ProductScreen from './screens/ProductScreen'

import CartScreen from './screens/CartScreen'

import LoginScreen from './screens/LoginScreen'

import RegisterScreen from './screens/RegisterScreen'

const App = () => {

  return (

    <Router>

      <div>

        <Header style={{ position: 'relative' }} />

        <main className='py-4 '>

          <Container>

          <Route path='/register' component={RegisterScreen} />

            <Route path='/login' component={LoginScreen} />

            <Route path='/products/:id' component={ProductScreen} />

            <Route path='/cart/:id?' component={CartScreen} />

            <Route path='/' component={HomeScreen} exact />

          </Container>

        </main>

        <Footer />

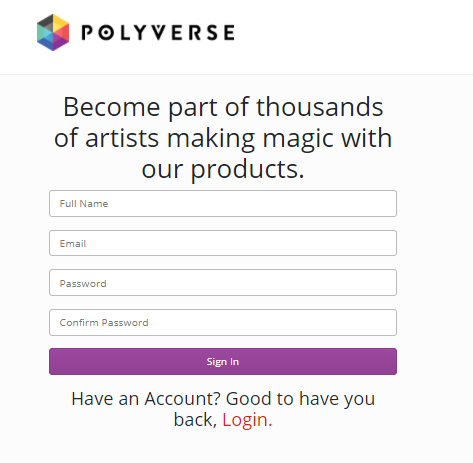
      </div>

    </Router>

  )

}

export default App



**User profile screen**

**Implement the user profile Screen in the backend**

Backend🡪 controllers🡪userControllers.js🡺

After the user in logged in he may get access to his profile data from the navbar.- we already managed this functionality in the backend. api/users/profile.

//this is a GET request to the protected route: api/users/profile

//return a specific user data.

const getUserProfile = asyncErrorhandler(async (req,res)=>{

    //in authMiddleware.js we store all user data that's passed the authentication and authorization proccess in req.user (excluding his password)

    const user = await User.findById(req.user.\_id)

   // res.send('good! call')

   if(user){

     res.json({

        \_id: user.\_id,

        name: user.name,

        email: user.email,

        isAdmin: user.isAdmin,

     })

   }else{

       res.status(404)

       throw new Error('user not found')

   }

 })

We want him to also be able to edit his details. Which we need to handle first in the backend:

(put request)

Add this code:

 //this is a PUT request to the protected route: api/users/profile

//enables edit the specific user's data.

const updateUserProfile = asyncErrorhandler(async (req,res)=>{

    //in authMiddleware.js we store all user data that's passed the authentication and authorization proccess in req.user (excluding his password)

    const user = await User.findById(req.user.\_id)

   if(user){

         user.name = req.body.name || user.name

         user.email = req.body.email || user.email

         if(req.body.password){

            user.password = req.body.password //will be encrypted automatically due to userModel.js encryption method (includs when password is updated)

         }

         const updatedUser = await user.save()

         res.json({

            \_id: updatedUser.\_id,

            name: updatedUser.name,

            email: updatedUser.email,

            isAdmin: updatedUser.isAdmin,

            token: generateToken(updatedUser.\_id)

         })

   }else{

       res.status(404)

       throw new Error('user not found')

   }

 })

export {authUser,getUserProfile,updateUserProfile  ,registerUser}

//don’t forget to export it as well!

Now lets update the user routes:

Backend🡪 routes🡪 userRoutes.js🡺

import express from 'express'

const router = express.Router()// api/users/

import {authUser,getUserProfile, registerUser,updateUserProfile} from '../controllers/userControllers.js'

import {protect} from '../middleware/authMiddleware.js' //routes protecting middleware

//fetch users email and password from the DOM(body)

router.post('/login',authUser) // api/users/login

//routes protecting middleware will protect the /profile route by adding it to the get request as its first argument!

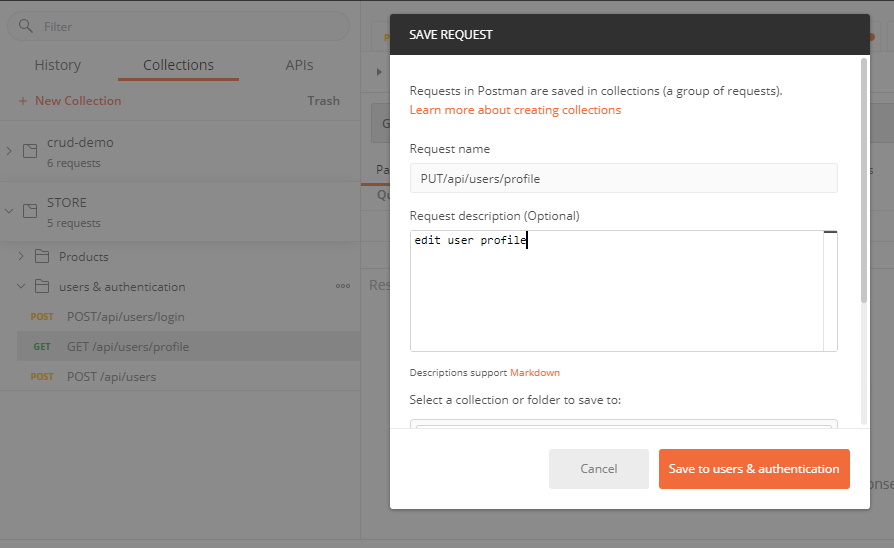
router.route('/profile').get(protect, getUserProfile).put(protect,updateUserProfile) // api/users/profile //put is for updating the profile

//register a new user route

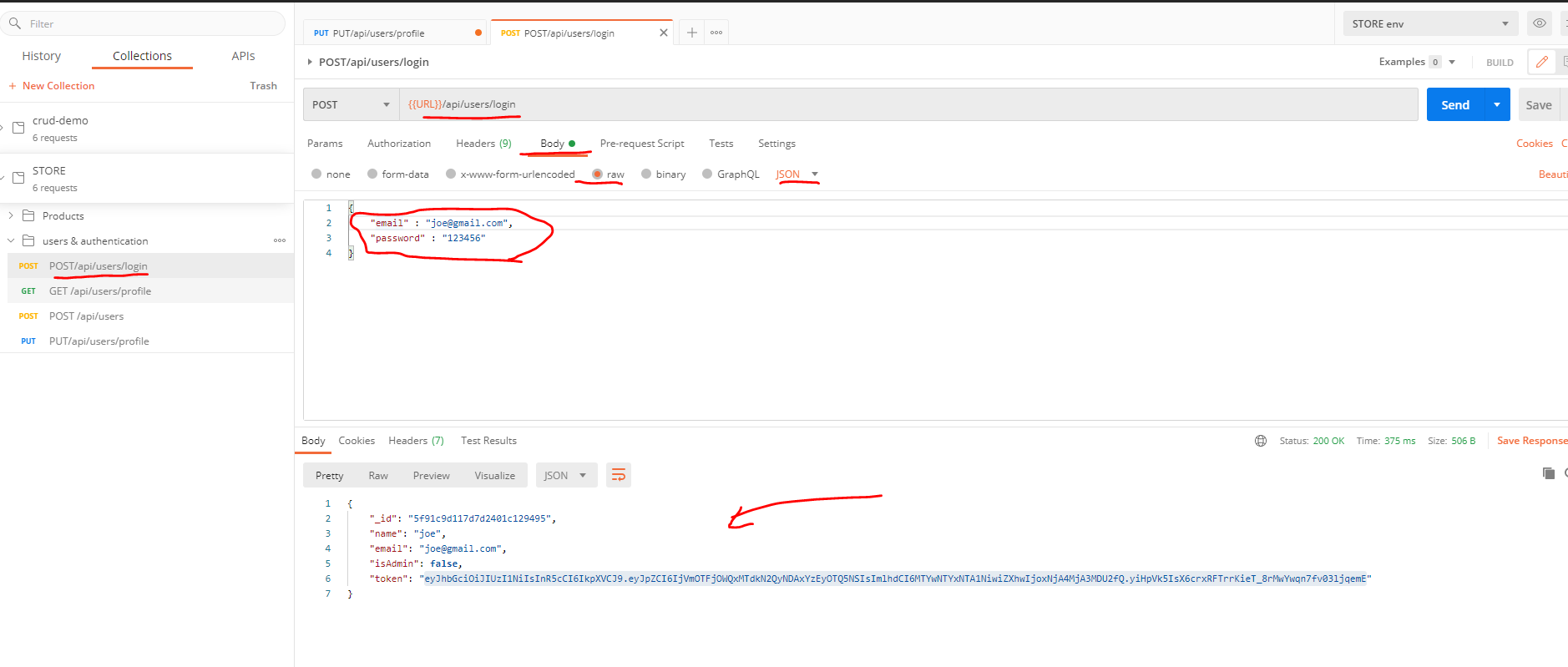
router.route('/').post(registerUser) // api/users/

export default router

let’s check if the /profile route functioning with postman:



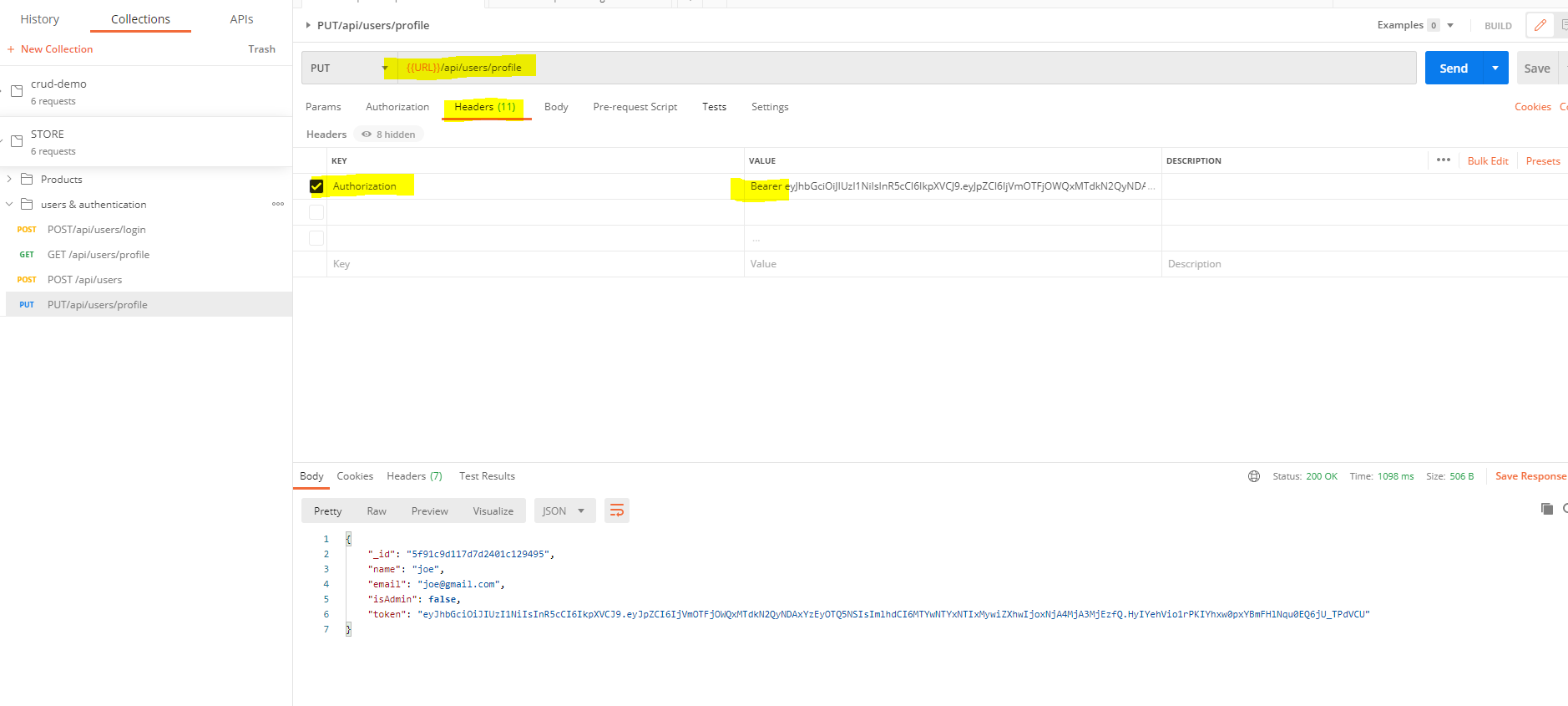
Now before we check the put request, we must perform a log to get a token:

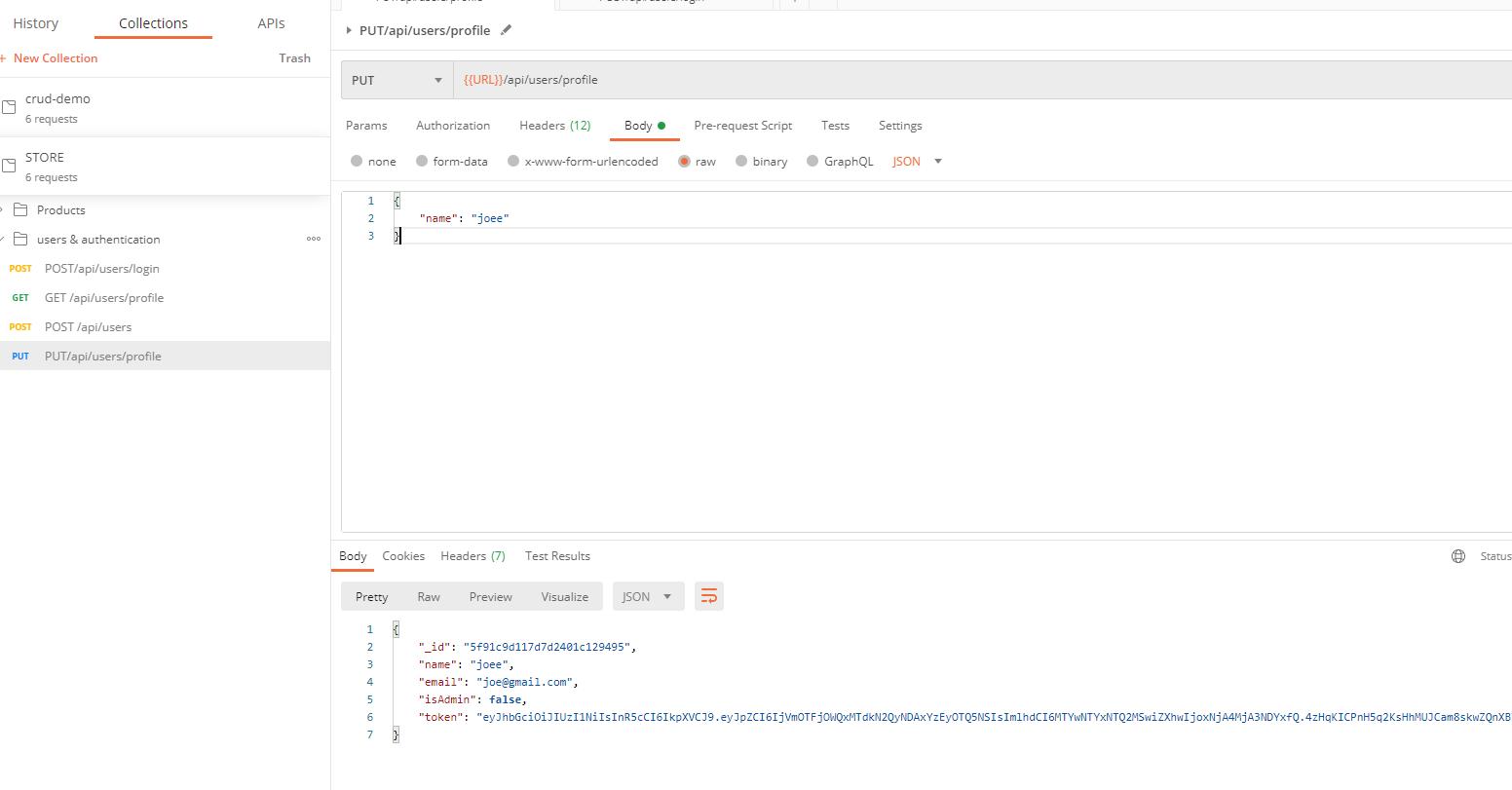


copy the token, and then go to the put request:

don’t forget to fill the headers with Authorization key, and in value write Bearer space token.

In the body write in JSON format a new name for your current user and check the change in the response ! :





**Implement the user profile Screen in the frontend**

So in order to do so again we will use redux: actions, reducers, constants and store

**Actions**

**Reducers**

**Constants**

**Store**