**Lecture 319**

**Adding a file picker to frontend**

* Refer code 01-adding-a-file-picker-to-frontend
* Edit-product.ejs 🡪 adding input file field

**Lecture 320**

**Handling multipart form data**

* Refer code 01-adding-a-file-picker-to-frontend
* Editproduct.ejs 🡪 added enctype="multipart/form-data to form
* Refer notes
* Bodyparser package cannot handle file data in the request, for that we need another package multer

**Lecture 321,322**

**Handling file uploads with multer**

* Refer code 01-handling-file-uploads-with-multer
* Npm install –save multer
* App.js 🡪 adding multer middleware
* Controllers/admin.js 🡪 postAddProduct 🡪 the file will be stored in the req.file attribute which is set by the multer

**Lecture 323**

**Filtering files by mimetype**

* Refer code 02-filtering-files-by-mimetype
* App.js 🡪 adding fileFilter callback
* If we choose a file which doesnot satisfy the mimetype , req.file will be undefined in controller

**Lecture 324**

**Storing File Data in the Database**

* Refer code 03-storing-file-data-in-database
* Controlers/admin.js 🡪 postAddProduct – showing error when req.file is undefined
* Controllers/admin.js 🡪 postEditProduct 🡪 adding logic to update the product with old imageurl , if the user has not selected any file or selected invalid file type during editing.
* Storing files in database is inefficient since files can be too large. So we store only file paths inside database.

**Lecture 326**

**Serving images statically**

* Refer code 03-serving-images-statically
* App.js 🡪 serving images folder also statically
* Shop/index.ejs;product-list.ejs;product-detail.ejs , admin/products.ejs 🡪 adding ‘/’ before image url
* Now we can serve more than one folder statically and remember, statically serving a folder simply means that requests to files in that folder will be handled automatically and the files will be returned, so all the heavy lifting is done behind the scenes by express then. I can duplicate this middleware and now also serve the images folder, just like this and with this little change if I save that and the server therefore restarts, if we reload this page, we still fail. Well if we inspect that failing request, we see that it tried to get the image from admin/images, the reason for that is that I'm on the admin route here, admin products to be precise, so it's only replacing the last part of my path with that image url and the solution for that is pretty straightforward. In our view products.ejs where I do show my image, we simply need to add a slash at the beginning which will turn this into an absolute path, so it will not append it to the current path but rather create a new path with only our domain and then the path which gets rendered here and I do this in products and of course also on my shop pages, so on index.ejs, so basically everywhere where I do render an image, I add a slash at the beginning. The alternative would be to store that path in a database with a slash at the beginning of course, might have been easier but I want to show you both ways
* If we change all of that and I now reload this page, I still fail and now the reason for that is something different. The reason for that is that the path now is correct but my images here in the images folder are served incorrectly. In app.js where I set up this static middleware, keep in mind what I taught you earlier in this course, we basically point to a folder there like public and images and we then tell express serve the files from inside that folder as if they were on the root folder. So we would see that image if we go there and we see the path under which I try to find that, if we open that in a new tab, we can't find it of course but we will see it if I remove images there and the reason for that is that express assumes that the files in the images folder are served as if they were in the root folder, so slash nothing. Of course we want to keep them in the images folder and keep the path like this and for this, we can simply adjust our middleware here and say if we have a request that goes to /images, that starts with /images, then serve these files statically and now /images is the folder we assume for this static serving and now with that if we save that and we reload, we see our image here.

**Lecture 327,328**

**Downloading files with authentication**

* Refer code 04-downloading-files-with-authentication
* Orders.ejs 🡪 adding link for Invoice
* Routes/shop.js 🡪 adding route for invoice
* Controllers/shop.js 🡪 getInvoice method

**Lecture 329**

**Restricting file access**

* Refer code 04-restricting-file-access
* Controllers/shop.js 🡪 getInvoice method 🡪 adding extra protection to allow only the user who created the order to download the invoice. Currently a user can download any invoice by typing in the url

**Lecture 330**

**Streaming Data vs Preloading Data**

* Refer code 05-streaming-data-vs-preloading-data
* Controlers/shop.js 🡪 getInvoice method 🡪 using read stream
* Now that's not all we can improve, we can also improve the way we are serving that file because right now, I'm simply reading that file and once I read it, I return it. Now for small files this is probably fine but you should be aware of one important fact, if you read a file like this, node will first of all access that file, read the entire content into memory and then return it with the response. This means that for bigger files, this will take very long before a response is sent and your memory on the server might actually overflow at some point for many incoming requests because it has to read all the data into memory which of course is limited. So reading file data into memory to serve it as a response is not really a good practice, for tiny files it might be ok but for bigger files, it certainly is not, instead you should be streaming your response data and that is what I will do now.
* For that I'll create a new constant, I'll name it file and I will use the file system and create a read stream because I want to read some data in. Now I want to read in data at a specific path and the path hasn't changed, it's the invoice path, so now I have to read read stream and node will be able to use that to read in the file step by step in different chunks. I will then take my response code here where I set the headers, I still do that on the response object and then here, I will use that file read stream and call the pipe method to forward the data that is read in with that stream to my response because the response object is a writable stream actually and you can use readable streams to pipe their output into a writable stream, not every object is a writable stream but the response happens to be one. So we can pipe our readable stream, the file stream into the response and that means that the response will be streamed to the browser and will contain the data and the data will basically be downloaded by the browser step by step and for large files, this is a huge advantage because node never has to pre-load all the data into memory but just streams it to the client on the fly and the most it has to store is one chunk of data. Again we're back to the streams and the buffers, the chunks are what we work with, the buffers basically gives us access to these chunks and here we don't wait for all the chunks to come together and concatenate them into one object, instead we forward them to the browser which then is also able to concatenate the incoming data pieces into the final file.

**Lecture 331**

**Using PDFKit for pdf generation**

* Refer code 06-using-pdf-kit-for-pdf-generation
* Using pdf kit to generate pdf on the fly.
* npm install --save pdfkit
* http://pdfkit.org/docs/text.html
* So let's create a new pdf doc, whatever you want to name it by calling new pdf document and that's what I meant, you need to add normal parentheses as we always did for all constructors. So now we have a new pdf document. This actually also turns out to be a readable stream, therefore what we can do here is we can use the pdf document and we can do two things. First of all we pipe this output into a writable file stream, so create write stream is a function we can call on the file system package and to that we pass a path where we want to write it to, the invoice path in my case and this ensures that the pdf we generate here also gets stored on the server and not just serve to the client. So we create that and of course I also want to return it to the client, so I also pipe the output into my response, just as before, the response is a writable read stream, pdf doc is a readable one so we can do that. Now we have this set up and now whatever we add to the document will be forwarded into this file which gets generated on the fly and into our response. Now let's start simple and let's now use pdf doc and let's call the text method which exists there, this allows us to add a single line of text into the pdf document. So here let's add hello world maybe and then you have to call pdf doc to tell node when you're done writing to that stream because you have to be done at some point, right. So here you simply call end and when you call end, these writable streams for creating the file and for sending the response will be closed so to say or will know that you are done writing and therefore the file will be saved and the response will be sent. And now with that saved, just make sure you also change your setup down there at the bottom. I will keep the headers but I will not pipe any file because we now, well pipe the file up there when we create it and I just need to make sure that I set my response headers accordingly.

**Lecture 332**

**Generating pdf files with order data**

* Refer code 06-generating-pdf-files-with-order-data
* Controllers/shop.js 🡪 getInvoice method

**Lecture 333,334**

**Deleting Files**

* Refer code 07-deleting-files
* Controllers/admin.js 🡪 postEditProduct , postDeleteProduct
* Util/file.js 🡪 adding deleteFile method
* When we delete a product , we should also delete the image of that file from the filesystem.
* Also when we update a product with new image , old one should be deleted.