# Group 150

Nicholas Flege & Zaoxuan Zhou

## Running the Project

1. Everything has already been deployed on the server so nothing needs to be built prior to running.
2. Connect to the campus Wifi or connect to the VPN.
3. Go to http://proj-319-150.cs.iastate.edu:3001 in Google Chrome. **MAKE SURE YOU ARE ON PORT 3001**

## Languages & Libraries

API: NodeJS and Express
Database: MongoDB
Views: Javascript and EJS

## Checklist

- ✓ Need to Sign Up before you can upload any files
- ✓ Clicking your user name in the navigation bar will navigate you to your documents.
- ✓ Clicking the CyCloud button takes you to the home page
- ✓ Certain file types can be viewed in the browser by clicking the “Open File” button. If the file cannot be opened in the browser, it will be downloaded.
- ✓ Look at Portfolio 3/routes/ to see all the files with the various routes for the application
- ✓ Look at Portfolio 3/views/ to see the EJS template files used to generate our views
- ✓ Look at Portfolio 3/public/js/upload.js to see the client-side Javascript code used to upload a file
- ✓ Look at Portfolio 3/routes/upload.js to see the server-side code used to save the uploaded files
- ✓ Look at Portfolio 3/routes/documents.js to see how a user’s documents are retrieved and send to the documents EJS template
- ✓ Look at Portfolio 3/app.js to see how we setup the authentication with Passport and Express sessions to verify users were signed in before displaying specific routes
- ✓ Look at Portfolio 3/app.js to also see how we refactored route code into different files to better modulate the code and make it easier to read

# Portfolio Three

Nicholas Flege & Zaoxuan Zhou
Group 150

# CyClould

A Node.js, Express, and MongoDB cloud storage application.

# Table of Contents

## Overview

Our project is a cloud storage application created with Node.js, Express, EJS, and MongoDB. Users can upload documents to the cloud, view their documents, and download their documents from the cloud. Users must create an account to upload and view their documents. For this project, we expanded our knowledge of Node.js that we had previously learned during the second portfolio and what we had learned during the last few weeks of class. We also expanded our knowledge of the Node.js Express library, as well as, learning many new technologies as well. During Portfolio Two, we learned how to use Express to make a REST API. For this project, we used Express to create a full web application. Instead of returning JSON data when a user goes to a certain URL, like we did for Portfolio Two, we serve actual HTML pages from Express. We used MongoDB to save our Users information because of how well it integrates with Node.js and Express. We used another Node library called EJS to create our views and HTML pages and we used Bootstrap to style the application. EJS allowed us to do server side rendering of our views, different from the client side rendering we did using React for Portfolio Two. This project almost entirely used new technologies/libraries or expanded on the technologies we used in Portfolio Two.

Unlike Portfolio Two, we did not want to use a token based authentication system. We went with a session based system, similar to the PHP sessions we learned about in class. We found and used an Express plugin library that added sessions to Express. We then used another library called Passport to implement the authentication with the Express sessions. To implement the file upload, we used the libraries fs and formidable, which were completely new to us.

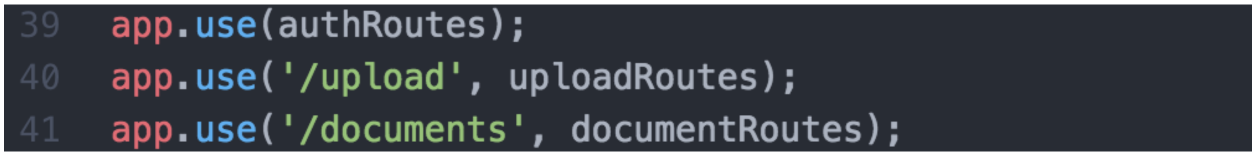
This project had us using Bloom's Taxonomy throughout the whole process of creating the cloud storage application. We were constantly analyzing, evaluating, and creating. After coming up with this idea for the project, we had to decide how we were actually going to make it. We analyzed our idea and researched what different technologies we could use to create it. We experimented other front-end frameworks similar to React, as well as, frameworks for server-side rendering with Express. We weighed the benefits and drawbacks of client-side vs. server-side rendering before making a decision. We also had to evaluate different technologies to decide if they were useful enough to be added to the application. We then started to design the application routes and layouts. We created a skeleton of the application with little to no frontend, only the different routes between pages. Finally, we assembled the routes with the server-side rendered views to create the full app.

# New & Complex

## *Express*

We had prior experience using the Express Node.js framework from Portfolio Two when we created a REST API. For this project, we are serving our entire web application from the server, instead of generating the views on the client side. Now, instead of sending a JSON response when a user accesses some endpoint, we send an HTML page. Express makes it very easy to create routes and can be done with only a few lines of code.

We also used a new Express feature called Routers for this app. Using routers allowed us to better separate our code into different files, making the code cleaner and modulated. Using the router, we separated the authentication, documents, and upload routes all into separate files. We were able to add routes to a router and then export that router using Node.js module exports. Then, in the app.js we import the routers and add them to the express variable. When you add the router to the express app, you can setup the base route for all the routes on the router as you can see in the code below for the upload and document routes.



```
app.use(authRoutes);
app.use('/upload', uploadRoutes);
app.use('/documents', documentRoutes);
```

## *EJS*

Since we decided not to do client-side view rendering for this application, we needed a way to create and serve the views from the server-side. There were quite a few options we had for libraries that allowed us to do this. Some of the options other than EJS are Jade, Underscore, and Handlebars, to name a few. We eventually decided on EJS. With EJS, we can create view templates that can then be sent from the server to the client to render.

With EJS, you can write normal HTML in an ejs file, that is then converted to an HTML file before being sent to the client. You can also write normal Javascript code in the ejs file if you use the special ejs tag <%  %>. When you render an ejs file, you can an object to the file with data to populate the template. For example, when we render the documents table, we pass an object with an array of file data objects. That object can be referenced and output to the HTML using another special ejs tag <%=  %>.

EJS also allows you to create partial templates where you can put view components that appear on multiple pages. This was helpful for us because we could put the header and navigation bar code in a partial and then use it on every view. The partials can be included in any view with a single line of code. This allowed us to avoid a lot of code duplication.

## *Express Sessions and Authentication*

Since your files should only be visible to you, we needed an authentication system to sign users in and out. We used the Passport library for a token based authentication system on Portfolio Two, but we couldn't use that same approach again since we weren't creating only an API. To solve this, we used the Express Sessions library. The library adds sessions to the default Express app that function similar to the sessions we learned about for PHP in class. Setting up Express to use sessions was fairly simple and only took a few lines of code.

To handle the user authentication, we used the Passport library and connected it to our MongoDB User collection using a Passport-Mongoose plugin. This plugin connected our User model in MongoDB directly to the Passport library. Using the plugin, we converted the user's password to a hash, so we weren't storing their password in plain-text on the database. There is also a compare password method which checks if the stored hash matches the user's input. We used this to sign users in.

To make sure only authenticated users could access their documents list and upload files, we used passport and express sessions as a middleware on our routes. If a user isn't signed in, and they attempt to access the upload or documents pages, they will be redirected to the sign in page.

## *Uploading Files*

We used new and complex libraries to implement uploading files. Users can upload files to the cloud to be viewed or downloaded at any time, and the user must be signed in to view their documents. When a user uploads a file to the server, we save the file to the server itself. To do this, we used three main libraries: path, fs, and formidable. The path and fs libraries come with the default installation of Node.js, so we only had to add formidable using NPM.

We had no prior experience working with these libraries prior to this project. We used path to make sure we were always getting the correct path to our files. Path has methods to get the file path to the current directory, and then from there we can append the appropriate path to the files we need. We used the file system (fs) library to handle all of our I/O for the files. When we load a user's files, we use fs to get the file data for each file in the folder for the signed in user. For example, the documents list page displays all the uploaded files and the size of each file. We can get the size of a file using the fs.stat() method. This gives use all the stats about a file at a specific path.

Now to actually upload a file to the server, we used the formidable library. The user first selects a file to upload, then we add that file to a FormData object on the client side and make a POST request to the server using AJAX. On the server, we have an Express POST route waiting to save the file to the server. We use formidable to read the FormData object from the client and then use fs to save the file to the user's folder on the server.

## *Progress Bar*

We also added a nice semi-complex feature to show the progress of a file upload. We wanted the client to have an idea of the time it was taking to upload the file to the cloud server, so we created a progress bar on the upload view to show this. We implemented this by updating the progress bar from within the AJAX POST call. We added an event listener to monitor the progress. Every time the listener was called, we calculate the percentage of the upload that has completed and we update the progress bar.

## *Method Override*

Since we created a full web app instead of just an API, there wasn't an easy was for us to send DELETE requests to the endpoints. We have a feature that allows the users to delete files they have uploaded to the server. To delete a file, we have a delete Express route. The problem was you can only have supply POST and GET methods on a form. Therefore, we didn't have a way to access the DELETE route. To solve this, we used the method-override Express plugin. This plugin allowed us to access this DELETE route. We created a form with its method set as POST and its action set as the name of the file to be deleted. Since we used this plugin, we could append "?_method=DELETE" to the end of the action URL and the plugin would override the POST method and send the request to the DELETE route. We debated using some other approaches such as making an AJAX or FetchAPI call, but we found this approach to be much more elegant and made more sense.

# Bloom’s Taxonomy

## *Analyze*

### Experiment

When deciding how we were going to render our views on the server-side before serving them to the client, we needed to decide which technologies or libraries we were going to use to do this. We did some basic research to figure out what our options were. We decided to experiment with EJS, Underscore, Jade, and Handlebars. We created some very basic HTML “Hello World” pages with the different templating/rendering libraries to get a feel for each. If the libraries had a “Get Started” section in their documentation, we went through that too. We used these experiments later to help us compare and contrast the different libraries. While comparing and contrasting them, we found the experiments to be very helpful in giving us a better understanding of each library.

### Compare/Contrast

After researching various libraries and frameworks to use for our project, we had to compare and contrast similar libraries to one-another before deciding which would be the best for our use cases. We needed to compare and contrast the different libraries for our server-side view template/rendering. We needed to choose from the four libraries we had previously experimented with: EJS, Underscore, Jade, and Handlebars. All four libraries were fairly similar in that they allowed us to write regular HTML code in the templates. The differences came in how well each library integrated with Express and with how Javascript could be written within the template files. All the libraries integrated well with Express, but it was simpler for some than others. It only required one line of code in the app.js file to setup Express to use EJS, while the others took 3-6 lines of code. We ended up narrowing it down to EJS or Handlebars. The Handlebars syntax was easier to read than the EJS, mainly because of the use of the <% %> tags required in the EJS templates. The documentation for EJS was much better than the docs for Handlebars. Partials were also much easier to use with EJS than with Handlebars. EJS also allowed us to write complete Javascript syntax within the template files, while Handlebars did not.

### Relate

While creating this cloud storage application, we had to relate what we had previously learned about Node.js and mainly Express while building a REST API to using Node.js and Express to build a full web application. We had to figure out which components and principles we had learned about Express that pertained to both REST APIs and web apps. We figured out that most of the route related principles and setup was generally the same for both types. The main differences where how the authentication should be handled and what is actually returned to the client that is accessing the specified routes. For the API, we only needed to return some JSON data that the client would then decide what it should do with it. For this web app, we needed to return actual HTML pages that could be rendered by the client's browser. For the authentication piece, we didn't need to use a token based system like we had used for the API. Instead we could use a session based approach similar to the PHP sessions we learned about in class.

## *Evaluate*

### Select

Ultimately, after our experiments with different technologies and then comparing and contrasting the technologies, we had to select which were best for our goals and the project itself. After weighing the pros and cons for the various templating libraries, we eventually ended up going with EJS as our template engine. EJS allowed us to write the normal Javascript syntax we are accustomed to within the templates themselves. The process to setup and connect EJS with Express was so simple, it could be done with just one line of code. EJS allowed us to create partials very easily and importing them into other templates was very simple too. Finally, while EJS may not have been as easy to real as Handlebars, it was still readable enough for us since it allowed regular HTML and Javascript syntax.

Looking back, after having completed the application, I think we definitely made the right decision by selecting EJS. We encountered next to no problems creating our main views or partial views. It was very easy to create incorporate Javascript code into the templates using the special EJS tags, and the tags had little effect on the readability of the code as a whole. Sending the rendered templates to the client browser was very clean too since Express handled all of the conversion from EJS to HTML files for us.

### Evaluate

After implementing how we uploaded files for our application, we evaluated our solution. At first, we just had a button the user clicked and they picked the file to be uploaded. There was no progress bar to let the user know, the status of their upload or any other type of notification method. We decided that our solution did not have a good user experience. The user should have some idea of the status of their upload. They should not be left wondering if their file was actually uploaded or not. Therefore, we decided to add a progress bar to our upload screen. After the user selected the file to be uploaded, we used an event handler function on the AJAX call that posted the file data to the server that allowed us to monitor the progress of the upload. Using this callback, we created the progress bar and displayed it to the user making the user experience much better.

## *Create*

### Design

The next step for creating this application after deciding on the technologies we were going to use was to design the structure. We started with designing the routes we were going to need for our web app. We knew we needed the basic authentication routes for sign in and sign up. We needed routes for viewing the upload page as well as a POST route for the client to send the upload data to. We also needed our default route for the home page. Finally, we needed document routes to view the documents list, individual documents, and a DELETE route for deleting files for a user's documents. We also determined we only needed one MongoDB schema, which would be for the Users.

We then had to design how we were going to store user's files on the server. We researched some various methods and eventually decided to just store the files directly on the server instead of in a database. We decided to create a different directory for each user within an overall uploads directory. Each user's files would be saved to the directory created for each user. Since each username needs to be unique, we did not have to worry about conflicts.

### Develop

After having established what we felt was good design and structure for the web app, we started to develop it. Using the Express knowledge learned from Portfolio Two, we were able to get the routes setup pretty quickly. Next, we used what we learned from our experiments with various server-side templating libraries to create our basic views. After the routes and basic views were created, we started to add the Express Sessions authentication to restrict access to certain routes unless a user was signed in. While this part ended up not requiring a huge amount of code (thanks to the integration of the Passport and Express Sessions libraries), it still took us longer than expected to setup. We attributed this to not fully understanding the libraries enough to know the correct order required to set things up. Once we figured out those small issues, we had the authentication piece setup.

Next, we needed to implement the file upload to the server. This took us quite a while to figure out. We did experiments with various templating libraries, but we did not experiment with how to implement the upload at all. Having not taken the time to experiment before starting development, we ended up doing experimenting with one approach, only to find that it didn't work for our situation. We then had to pretty much start over with a completely new approach. It ended up causing us to waste a lot of time because we had not properly planned or researched beforehand. The final piece required for the development was to add some styles to our basic views. We used Bootstrap to make the views look better and more appealing to the users.

## Assemble

The upload component of our web app required us to assemble the client-side and server-side components. On the client-side we needed to use an API call with AJAX to make a post request to our application with the file data. The server-side Express route needed to then process the POST data and save it to the server. Bringing these two pieces together did not go as smoothly as anticipated. We had to do a lot of debugging and sending of fake data to make sure everything was working together properly before we finally got it working. Once the two upload components were assembled, our app was finished.