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 in Google Chrome.

Languages & Libraries

API: NodeJS and Express
Database: MongoDB
Front-end: Javascript and ReactJS

Checklist

API

- Look at isupoll-api/models/user.js to see how user's password is encrypted before saving
- Look at isupoll-api/config/passport.js and controllers/authController.js to see how the authentication is handled using JSON Web Tokens
- Look at isupoll-api/controllers/apiController.js to see API routes and how we used middleware to restrict some routes to only authenticated users
- isupoll-api/App.js contains code to setup and run the API server

Client App

- Create a new user by using the ***Sign Up*** button
- Sign out after creating a new user using ***Sign Out*** button
- Sign back in to the newly created user using ***Sign In*** button
- To go back to Home page, refresh or press the ***ISUPoll*** button
- After signing in you can view your polls and create new polls by clicking your username in the navigation bar
- You can click a poll and vote on it without creating an account
- See isupoll/src for the various React components created to the app
- See an example of an API request with an Authorization Header containing the auth token in isupoll/src/UserPollList.js

Portfolio Two

Nicholas Flege & Zaoxuan Zhou
Group 150

ISUPoll

A Node.js, MongoDB, and React voting app

Table of Contents

Overview

Our project is a voting/polling application created with NodeJS, Express and React. Users can create polls and allow other people to vote on them. To create a poll, a user must sign up or sign in, but an account is not required to vote on a poll. This project expanded on our in-class learning of Javascript and had us learn many new technologies as well. The backbone of our voting web app is a Node.js and Express REST API. While building this application, we gained a very solid understanding of the Express library and its usefulness for creating web applications and REST APIs. We also used a different database then what we had previously used in class and the labs. We decided to use MongoDB instead of MySQL because of how well it integrates with Express and Node.js. MongoDB is a NoSQL database, so unlike MySQL, it is not a relational database. Instead, the data is stored in JSON-like objects.

We wanted the authentication method to be as similar to a real-world example as possible. This led us to implement a token-based authentication using JSON web tokens. For the front-end of our voting web app, we used the React framework created by Facebook. This framework allowed us to create reusable view components with Javascript. We decided to use Fetch API instead of AJAX to connect to our Express REST API. With Fetch API, we were able to connect our API to our React components to display the data from the database. This project has used almost all new technologies that we learned outside of class, aside from some basic Javascript knowledge.

This project had us using Bloom's Taxonomy throughout the whole process of creating the voting web app. We had to analyze, evaluate, and create constantly. There were two major components for this project. The back-end REST API created with Node.js and Express and the front-end client built with the React Framework. While analyzing our project idea, we experimented with different technologies and compared and contrasted their usefulness to our idea. We then had to evaluate the various technologies to decided which we would use for the application. Finally, we had to actually create the app. We designed the structure of the API. Then we created a skeleton client frontend before finally assembling the two pieces to create the full voting app.

New & Complex

Node.js

For this project, we wanted to use a different backend technology then what we had already used in class. This led use to decide to use Node.js for the backend REST API. We chose Node because it allowed us to use our existing knowledge of Javascript and also learn something new. Prior to this project, we had no experience using Node. Node allowed us to write Javascript code for the both the front and back-end of our web app. One thing that really stuck out to us while creating the API was how much callback functions and closures are used in Javascript. We had used callbacks occasionally while working on the labs for class, but never to the extent we found we were using them in Node.

We also had to learn what NPM was and how it worked with Node.js. NPM is a package manager for Node used to easily add libraries to a project. We used NPM to add libraries like Express, Body Parser, Passport, and a few others to create our back-end REST API.

Express

Express is the Node.js framework we used to create the REST API for our voting app. We decided to build a REST API instead of web server because an API could be accessed by other platforms (i.e. mobile) if that was something we eventually wanted to create. A web server would have required us to render the html pages on the server and then serve them to the client. With the API approach, we just make a request to the server for the data we need and then display it.

With Express, we were easily able to create and setup the endpoints the client would need for the web app to function. We created routes for user creation, authentication, retrieving polls, creating polls, updating polls, and deleting polls. An interesting feature we found in Express was the ability to add *middleware* to our API routes. Middleware are the functions that are invoked before the final request handler is executed. This was a concept we hadn't seen in any of the labs yet and was completely new. We used a middleware library called Morgan to log every request that came in to our server. This helped us debug errors during development. We also used middleware to implement authentication for our API.

The image below shows an example of three API routes we created. The first gets all polls for a provided username. The second gets all polls in the database, and the third gets a poll with a specific id. Express allowed us to do some really complex things with relatively little code. As you can see below, it only took 25 lines to create these three routes.

```
app.get('/api/polls/:username', passport.authenticate('jwt', { session: false }), function(req, res) {
  Polls.find({ username: req.params.username }, function(err, polls) {
    if (err) {
      throw err;
    }
    res.send(polls);
  });
});

app.get('/api/polls', function(req, res) {
  Polls.find({}, function(err, polls) {
    if (err) {
      throw err;
    }
    res.send(polls);
  });
});

app.get('/api/poll/:id', passport.authenticate('jwt', { session: false }), function(req, res) {
  Polls.findById({ _id: req.params.id }, function(err, poll) {
    if (err) {
      throw err;
    }
    res.send(poll);
  });
});
```

Token-Based Authentication

The authentication portion of the REST API was the most complex and difficult part for us to understand and implement. We wanted to incorporate an authentication piece to our API because authentication and security are very important in real-world applications. We wanted to gain experience to help prepare us for when we eventually enter the workforce. We decided to use a token-based authentication for the API. We did this using JSON Web Tokens and a Node.js library called PassportJS. After a user signs in, they are given a token that they need to provide with each request in order to access the API. If that token is expired or is not provided, the user will not be able to access most of the API routes. The most complex part was figuring out how to verify a user's username and password and then create and return the tokens when they signed in or signed up with valid information.

MongoDB

We decided also decided to use a database different from MySQL, which we had used in class. We deiced on MongoDB. Mongo is a NoSQL database that stores its data in JSON-like documents. This allows Mongo to interact with Javascript code very easily. The schemas are also dynamic unlike MySQL databases, so the structure of the data can change very easily. We used a Node library called Mongoose to interact with our Mongo database that we setup on our server.

A complex piece we added with Mongo was not saving the users passwords in plaintext. We used another Node library in addition to Mongoose called Bcrypt to encrypt a user's password before saving it to the database. Bcrypt uses a salted hash to encrypt the password. We also

had to be able to decrypt the passwords so we could compare the them when a user attempted to sign in. The code below shows how we do this. The *pre('save')* method is part of Mongoose that allows us to define a method that is ran every time before a User is saved to the database. This is where we encrypt the password before it is saved. The *comparePassword* method is used to decrypt the password and compare it with the provided password from the user.

```
UserSchema.pre('save', function(next) {
  const user = this;
  const SALT_FACTOR = 5;

  if (!user.isModified('password')) {
    return next();
  }

  bcrypt.genSalt(SALT_FACTOR, function(err, salt) {
    if (err) {
      return next(err);
    }
    bcrypt.hash(user.password, salt, null, function(err, hash) {
      if (err) {
        return next(err);
      }
      user.password = hash;
      next();
    });
  });
});

UserSchema.methods.comparePassword = function(providedPassword, callback) {
  bcrypt.compare(providedPassword, this.password, function(err, isMatch) {
    if (err) {
      return callback(err);
    }
    callback(null, isMatch);
  });
}

module.exports = mongoose.model('Users', UserSchema);

```

React Framework

For the front-end of our voting app we wanted to use a framework to organize code and make connecting the view to back-end data easier. We decided to use React, which is a front-end framework created by Facebook. This was completely new to us. We debated using Angular but eventually decided on React. With React we didn't create any HTML files for our front-end. All the HTML code is written in a special form of Javascript created as part of React called JSX. React uses the concept of Components. Every view is built from different components. Using components allowed us to reuse a lot of code. The components also can store state for the component. When the state changes, the component is automatically rendered and displayed on the screen. We discovered an advantage of React is that when it renders changes on the screen, it doesn't re-render the entire DOM. Instead, it maintains a Virtual DOM and then

compares that to the actual DOM whenever a change occurs and only renders the changes, which really improves performance and load time for the app.

Using React we also started to use the some of the new ES6 Javascript syntax. For example, we used *import* instead of *require* when importing other files or libraries. We also used *const* and *let* in most cases instead of *var*. When we created components, we used the new class syntax that allowed our custom component classes to extend the React Component class. Finally, we used the arrow function syntax in some places, mainly when using the Fetch API.

Fetch Framework

Instead of using AJAX to get the data from our API to our client React app, we decided to use the Fetch API. Fetch is in the process of becoming the standard for HTTP requests and responses. We found the syntax to be a little different from AJAX, but the concepts were essentially the same. We had to learn how to provide headers containing the authentication token when making requests to our API.

ChartJS

We used the ChartJS library to display the results of each poll in a pie chart. We were able to also find another library that combined ChartJS and React allowing the two libraries to interface very well with each other. With the power of React, we were able to create the pie graphs with only a few lines of code and a very easy to read format.

Bloom's Taxonomy

Analyze

Differentiate

While planning how we were going to build our voting application, we had to differentiate. First, we had to establish a solid understanding of the differences between a front-end and back-end of an application. We had to research and find the different technologies available to use on both ends. Before starting this project, neither of us had a solid understanding of what Node.js was. After researching, we found that Node.js was different from regular Javascript in that it was ran on the server-side rather than the client-side web browser. From there, we were able to research the libraries available to use while also differentiating whether the library was a server-side or a client-side library.

Experiment

Once we had established the difference between the back and front-end technologies, we were able to start doing some experiments. We created some simple "Hello World" apps with both React and Angular to get a feel for both. We later used the experiments from both frameworks to help us compare and contrast before deciding what we would use for the project. We also did some experiments with Express and Hapi before deciding on the back-end framework we were going to use. We went through the "Get Started" tutorials on the online docs for both frameworks to get a feeling for each. When comparing the technologies later, we found it very beneficial to have worked through experiments first.

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. For example, when we were looking for frameworks to use for our back-end API, we found a few Node.js libraries. There was Express, Koa, and Hapi to name a few. We ended up deciding on Express because of how friendly it was for people just learning Node.js. Express used a syntax and coding style similar to what we had already done in class with client-side Javascript. Express also allowed us to setup an API server fairly quickly with relatively little code.

We then had to decide how we were going to organize the front-end Javascript/HTML code. We could have either used a framework or the barebones approach of just regular HTML and JS files. We decided to go with the framework approach so our Javascript code could more easily be connected to the HTML views. We then had to decide which front-end framework to use. We narrowed it down either React or Angular. We were somewhat familiar with Angular from what we had covered in class, but we had virtually no experience with React. With React we

found we could build custom HTML components, which allowed us to reuse a lot of code. Also, when changes are made to the model, React only renders the required changes in the view instead of redrawing the entire view. Using React also gave us the chance to learn another new technology.

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. We evaluated the backend technologies and eventually decided Express would be the best for our needs. Express was straightforward to use and was well-tested, so we wouldn't have to worry about encountering bugs within the framework. Express also provided an easy way to add middleware authentication for our API. We then had to make a decision for the front-end framework. We selected React for its performance and also as the chance to learn a new technology that we hadn't already covered in class.

In retrospect, I think we should have done some more research into React, specifically, the design patterns used when building React apps. React doesn't use the same Model-View-Controller approach used by Angular. Initially, we were creating our app with an MVC approach and were encountering problems because that's not how React is designed. Once we established a solid understanding of how React apps are generally structures, the development when much smoother.

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 REST API. We decided that we needed poll routes for an individual poll, all polls for a username, all polls. We also needed routes to update and delete polls. We also had to design our database structure and schemas. We determined we should only need two MongoDB schemas for the app: Users and Polls. We then planned out the fields for each schema and how they would interface with each other.

By designing the API and database structure first, it made it much easier to plan out the front-end portion of the application and probably saved us a lot of time. With the API designed, we were able to figure out the general structure of each page on the front-end as well as the authentication flow. We also wanted to make the client app look good, so we did some initial design of the look for the app too. Having established some initial design before starting

development helped us make the app look very good and made the creating the design go a lot smoother than if we had just designed it on the fly.

Develop

After having established a what we felt was good design for the API, we started to develop it. Using the knowledge we had learned from our experiments, we were able to get the endpoints for our API working quickly. The next step for us was to add the token-based authentication to the API. This part ended up taking us a lot longer than we planned. We had done a good job researching the technologies we needed to create the API and front-end but had spent very little time doing any research or experiments with authentication. 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 delete everything we had done with the authentication up to that point and start over. It ended up causing us to waste a lot of time because we had not properly planned or researched beforehand.

Assemble

Having first developed the back-end REST API and then front-end client app after, we had to assemble the two pieces in order to make the application whole. Bringing the pieces together went a log smoother than we had originally anticipated. Using the Fetch API to pull the information from the server and display it on the client was really easy. From our previous work with AJAX, making the transition to a different way of retrieving the data was pretty easy. We were able to connect the data pulled from the server to the views without too many problems at all. We attributed this to having taken the time to make sure we had a solid design for our API before starting development. We had all the data we needed readily available and didn't have to add any extra endpoints at a later time.