# INFSCI 2560 Network & Web Data Technologies

**Dhaval Sonani (dvs15)**
**OM Patel (OMP8)**
**Sandeep Mysore (SKM62)**

# Let's Debate

## Project Report

### Readme

Project folder contains folders such as models, public and views. It also contains individual files like app.js, db.js, passport-config.js and routes.js. db.js file is script to connect with mongo DB database. Passport-config.js file is user credentials authentication script. Routes.js file contains all the routes for login and signup. The app.js is primary node js file to run project with command "node app.js". Models folder contains all mongo DB collection schema each with descriptive name. Public folder has static files like JavaScript, CSS and SVG images. Views folder contains EJS files each described below.

Login.ejs file is login page.

Signup.ejs file is signup page.

Index.ejs file is home page of application.

debate_.ejs file is page for each individual debates.

### Accessibility and Responsive Design

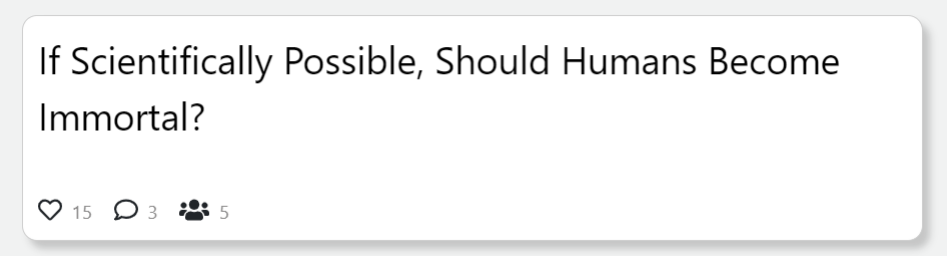
**Perceivable:** We have used white background with black font so it would be easily perceivable for all age group people. We have also used text alternatives for images in this application.

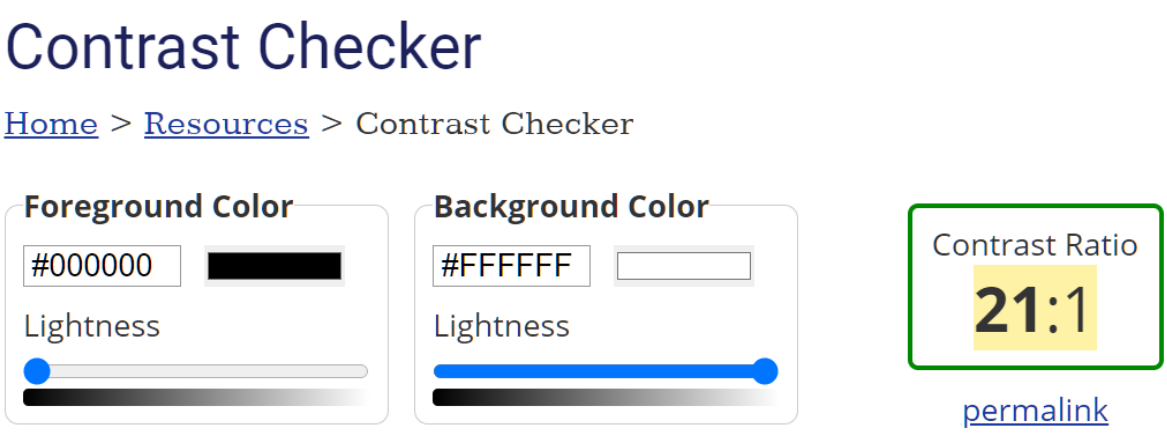
**Operable:** User can mostly navigate thorough keyboard as well apart from mouse. We have given hover effect to buttons and cursor changes to pointer on hover so that user can easily differentiate between clickable and non-clickable component.

**Understandable:** We have used simple language which is easily readable and understandable.

**Robust:** Out application works on all famous browser in market and it is compatible all screen sizes.

**General:** There are appropriate page titles for each page which are easy to understand. We have used simple fonts and our contact ratio for background color and text is 21:1 as shown below. Interaction – only keyboard and mouse needed to interact.



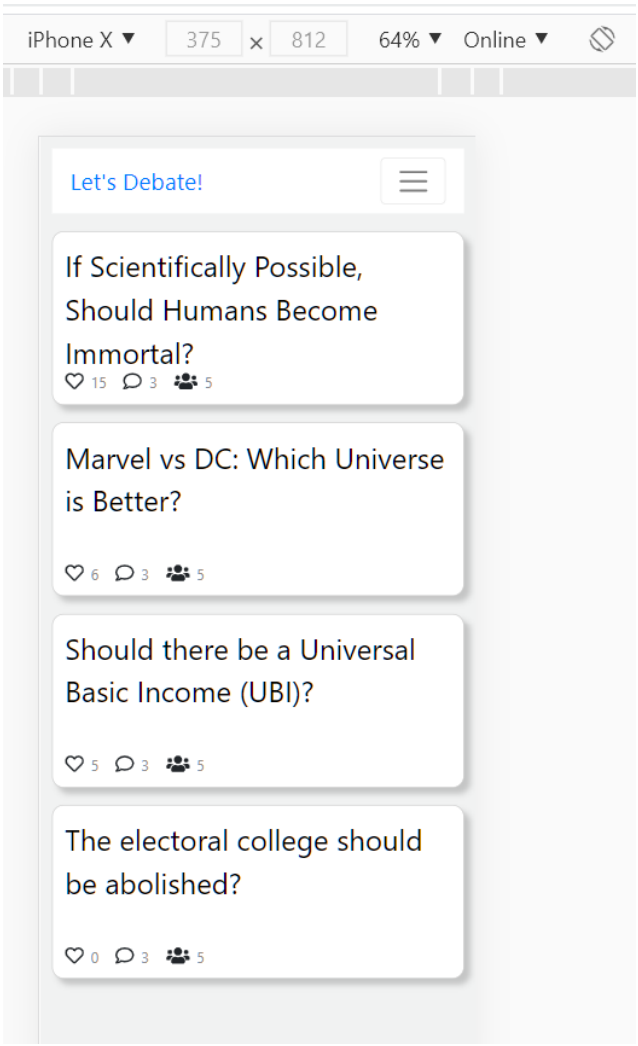




**Limitations in terms of Accessibility**

We have not implemented color scheme for colorblind people.

**Responsiveness**

Application layout perfectly fits devices of all sizes and navigation bar converts into toggle button for smaller display devices as shown below. We have used min-width and max-width CSS properties of various elements to achieve responsiveness. Navigation bar is developed using CSS framework bootstrap which automatically changes navigation bar in toggle button for small screen.




## Authentication

In this project we have used passport authentication middleware for Node.js to authenticate user and we have also used flash to store error messages.

As shown in below code, we have used passport local strategy to authenticate user with their email address and password. Here, username represents user's email address. In this strategy, we are first looking for potential error during authentication and then we are verifying whether user exists or not. If user has does not exists, we are passing error message "No user with the username". Then it verifies if password is correct or not. If it's not correct, then it will pass error message "Invalid password". If everything is good, passport will log in the user and it will render index.ejs page.

We are serializing and deserializing user using _id field of user as shown in below code. Passport creates user session and stores all user details in request object of Node.js routes. It destroys session when user logs out.

```
// serializing and deserializing users from the mongodb
module.exports = function() {
  passport.serializeUser(function(user, done) {
    done(null, user._id);
  });

  passport.deserializeUser(function(id, done) {
    User.findById(id, function(err, user) {
      done(err, user);
    });
  });
};

// setup the authentication strategy
passport.use(new LocalStrategy(function(username, password, done) {
  User.findOne({email: username}, function(err, user){
    if (err) { return done(err)};
    if (!user) {
      return done(null, false, {message: "No user with that username"});
    }
    if(password === user.password){
        return done(null, user);
    }
    return done(null,  false, { message : 'Invalid Password'});
  });
}));
```

## Front End

We have used EJS template for web pages. Apart from that, We used CSS3, little bit of bootstrap and font-awesome icons for web page design. In this project, we created 4 EJS templates – index.ejs, login.ejs, signup.ejs and debate_.ejs in view folders.

Login.ejs contains form for login with username and password field. On submission, it will be redirected to login post route in back-end.

```
<form action="/login" method="POST">
    <label for="email">Email: </label><br>
    <input type="email" id="email" name="username" placeholder="Enter email..." pattern="^[\w-\.]
    +@([\w-]+\.)+[\w-]{2,4}$" required><br>
    <label for="password">Password:</label><br>
    <input type="password" id="password" name="password" required><br>
    <input type="submit" id="submit" value="Login"><br>
    <a href="/signup" id="signup">Sign Up</a>
</form>
```

Signup.js contains signup form which contains user fields like firstname, lastname, email, password and preferences. On submission, it will be redirected to signup post route in back-end.

```
<form action="/register" method="POST">
    <label for="first_name">First Name:</label><br>
    <input type="text" id="first_name" name="first_name" required><br>
    <label for="last_name">Last Name:</label><br>
    <input type="text" id="last_name" name="last_name" required><br>
    <label for="email">Email:</label><br>
    <input type="email" id="email" name="username" pattern="^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$"
    required><br>
    <label for="password">Password:</label><br>
    <input type="password" id="password" name="password" required><br>
    <label for="preferences">Preferences:</label><br>
    <input type="text" id="preferences" name="preferences"><br>
    <input type="submit" id="submit" value="Sign Up">
    <a href="/login" id="login">Login</a>
</form>
```

Index.ejs contains bootstrap navigation bar which has link for home page, greeting message for user, link to create new debate and logout button. User can click on create debate button and it will take user to create debate window on which user can enter debate title and related tags to create new debate. Below navigation bar, it shows list of ongoing debates. User can click on any debates to go debate page of corresponding debate. If user is admin, user gets additional rights to delete any debates. On this page we have used font-awesome icons for likes, user, comment and delete.

Debate_.ejs page also contains bootstrap navigation bar and it displays all comments received for selected debates. On this page user can add comment, like/unlike debate and like/unlike any comment. If user is admin, then admin gets additional functionality to delete any comment permanently. On this page also we have used font-awesome icons for delete.

## Back-end

To implement back-end we used Node.js and Express.js library for Node.js.

In our project app.js is primary js file to run application. It runs application on port 3000 and it can be accessed with http://localhost:3000 link in browser. In this file, we have included all the packages such as express, passport, express-session etc., which we use in this web application. It also included all other internal js files such as route.js, passport-config.js and db.js.

```
const express = require('express');
const path = require('path');
const passport = require('passport');
const session = require('express-session');
const flash = require('express-flash');
const bodyParser = require("body-parser");
const routes = require('./routes');
const App = express();
const connectDB = require('./db');
const initializePassport = require('./passport-config');
initializePassport();
```

```
App.set('views', __dirname + '/views/');
App.set('view engine', 'ejs');
App.engine('ejs', require('ejs').__express);
```

It sets EJS as template engine and it uses some middleware functions such as flash, session, passport functions and internal file routes as shown below.

```
App.use(bodyParser.json());
App.use(flash());
App.use(session({
    secret: 'sdfvfvf',
    resave: false,
    saveUninitialized: false,
}));
App.use(passport.initialize());
App.use(passport.session());
App.use(routes);
```

All login and signup routes are implemented in route.js file which has been added as middleware in primary app.js file.

Below is implementation of login get and post request. As we can see, it renders login page on get login request and it authenticates user using passport when login post request is called.

```
router.get("/login", checkNotAuthenticated, function(req, res){
  res.render("login");
});

router.post("/login", passport.authenticate("local", {
  successRedirect: "/",
  failureRedirect: "/login",
  failureFlash: true
}));
```

Below is signup get and post request. It renders signup page upon signup get request. When signup post request is called, it first checks whether user already exist. If user does not exist it adds new user in database and authenticates user using passport.

```
// Route to signup page
router.get("/signup", function(req, res){
  res.render("signup");
});

router.post("/register", function(req, res, next){
  var username = req.body.username;
  let preferences = req.body.preferences.split(',');
  preferences.forEach(function(preference, index){
      preferences[index] = preference.trim();
  });

  User.findOne({email: username }, function(err, user){

    if (err) {
      return next(err);
    }
    if(user) {
      console.log(user)
      req.flash("error", "User already exists");
      return res.redirect("/signup");
    }
    var new_user = new User({
      first_name: req.body.first_name,
      last_name: req.body.last_name,
      email: req.body.username,
      password: req.body.password,
      preferences: preferences,
      debate_liked: [],
      comment_liked: [],
    });
    console.log('here')
    new_user.save(next);
  });
}, passport.authenticate("local", {
    successRedirect: "/",
    failureRedirect: "/signup",
    failureFlash: true
}));
```

Below is route for home page. It always checks whether user is already authenticated or not before rendering home page. If user is not authenticated, it will redirect user to login page. If user is already authenticated, then it will load available debates from database according to their popularity. If user is admin it will give debate delete option to admin.

```
App.get('/', checkAuthenticated, (req, res) => {
    let num_comment = []
    Debate.find({}, function(err, debate) {
        let is_admin = false;
        Admin.exists( {email_id: req.user.email}, (err, result) => {
            if(err){
                throw err;
            }
            res.render('index', { user : req.user.first_name, debates: debate, is_admin: result});

        })

    }).sort({ likes : -1 });
})
```

Below routes renders individual debates. Like above route, it will also verify whether user is logged in or not. This route fetches all comments received by given debate and pass them on to debate_.ejs page. It also sorts comments according to their popularity.

```
App.get('/debate_', checkAuthenticated, (req, res) => {

    Debate.find( {_id : req.query.debate_id}, function(err, debate){

        Comment.find({debate_id: req.query.debate_id}, function(err, comment) {
            let is_admin = false;
            if(admin.includes(req.user.email)){
                is_admin = true;
            }
            res.render('debate_', { "debate" : debate, "comments": comment, "user": req.user,
            "is_admin": is_admin});
        }).sort({ likes : -1 });
    })
})
```

We have created below route for back-end database update, retrieve and delete requests.

This route is used to create new debate.

```
App.post('/createdebate', (req, res) =>
```

This route is used to fetch a debate data.

```
App.get("/debate_data", (req, res) =>
```

This route is used to fetch a user data.

```
App.get("/user_data", (req, res) =>
```

This route is used to delete a debate.

```
App.get('/deletedebate', (req, res) =>
```

This route is used to delete a comments.

```
App.get('/deletecomment', (req, res) =>
```

This route is used to update like count for a comment when user likes it.

```
App.post("/updatelikes", (req, res) =>
```

This route is used to update comment like list in user collection when user likes a comment.

```
App.post("/updatelikeslist", (req, res) =>
```

This route is used to update like count for a comment when user unlikes it.

```
App.post("/updatelikes_unlike", (req, res) =>
```

This route is used to update comment like list in user collection when user unlikes a comment.

```
App.post("/updatelikeslist_unlike", (req, res) =>
```

This route is used to update likes when user likes a debate.

```
App.post("/updatelikes_debate", (req, res) =>
```

This route is used when to update debate like list in user collection when user likes debate.

```
App.post("/updatelikeslist_debate", (req, res) =>
```

This route is used to update likes when user unlikes a debate.

```
App.post("/updatelikes_debate_unlike", (req, res) =>
```

This route is used when to update debate like list in user collection when user unlikes debate.

```
App.post("/updatelikeslist_debate_unlike", (req, res) =>
```

This route is used to find all comments of a debate.

```
App.get("/comments", (req, res) =>
```

This route is used to add pro comment.

```
App.post("/pro", (req, res) =>
```

This route is used to update con comment.

```
App.post("/con", (req, res) =>
```

## Database and Database Access

We have used mongo DB database for our project. To implement mongo DB we have used mongoose JavaScript library. Our database is stored on mongo DB atlas cloud. Four schemas have been generated for four mongo DB collections - admin, comment, debate and user as shown below.

**Admin schema**

```
const AdminSchema = new mongoose.Schema({
    email_id:{type:String},
})
```

**Comment Schema**

```
const CommentSchema = new mongoose.Schema({
    pro_con:{type:Boolean},
    comment:{type:String},
    debate_id:{type:String},
    user_id:{type:String},
    likes:{type:Number}
})
```

**Debate Schema**

```
const DebateSchema = new mongoose.Schema({

        title:{type:String},
        tags:{type:Array},
        posting_date:{type:Date},
        user_id:{type:String},
        likes:{type:Number}
})
```

**User Schema**

```
const UserSchema = new mongoose.Schema({
    first_name:{type:String},
    last_name:{type:String},
    password:{type:String},
    email:{type:String},
    preferences:{type:Array},
    debate_liked:{type:Array},
    comment_liked:{type:Array}
});
```

We are using below script to connect with mongo DB atlas cloud. It will show message with host once it is connected to mongo DB database, otherwise it will show error.

```
const connectDB = async () => {
    try {
        const conn = await mongoose.connect('mongodb+srv://dhavalsonani:xyz123@cluster0.el8ap.mongodb.
        net/lets_debate', {
            useNewUrlParser: true,
            retryWrites:true,
            useUnifiedTopology: true
        });
        console.log(`mongoDB connected: ${conn.connection.host}`);
    }
    catch(err){
        console.error(err);
    }
}
```

To add and update database we just need to include various schema mentioned above in our route file and then we can use mongoose inbuilt functions like findOne(), updateMany() etc to fetch, update and delete data.

## Error Handling

For error handling, we tried to use try catch block and if block as much as possible so that we can know what exact error is, as shown below. Moreover, we are also implementing user validation at front-end to minimize exceptions at the backend.

Here we are using if block before rendering page so that execution does not continue in case of any error.

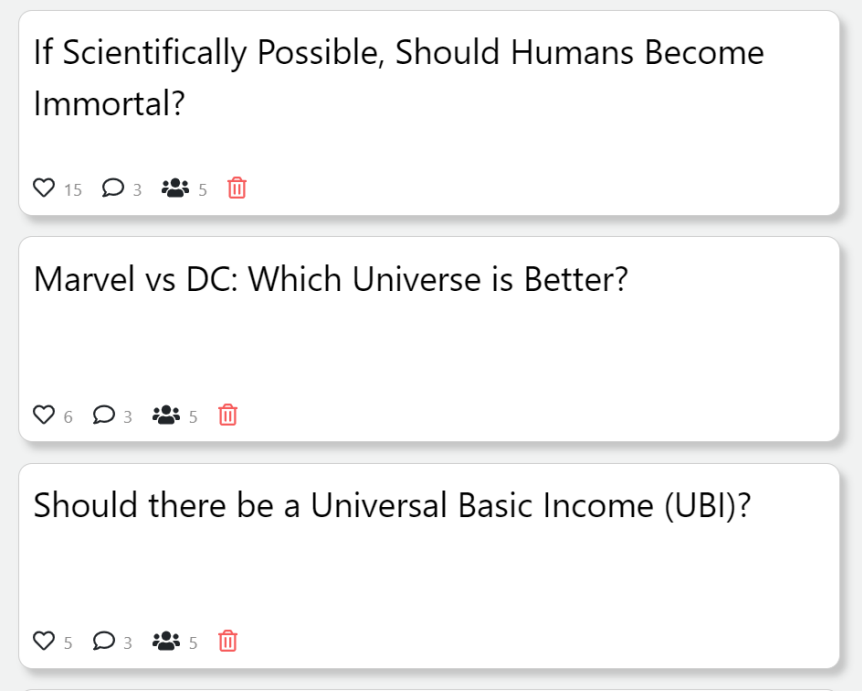
```
App.get('/', checkAuthenticated, (req, res) => {
    let num_comment = []
    Debate.find({}, function(err, debate) {
        let is_admin = false;
        Admin.exists( {email_id: req.user.email}, (err, result) => {
            if(err){
                throw err;
            }
            res.render('index', { user : req.user.first_name, debates: debate, is_admin: result});

        })

    }).sort({ likes : -1 });
})
```

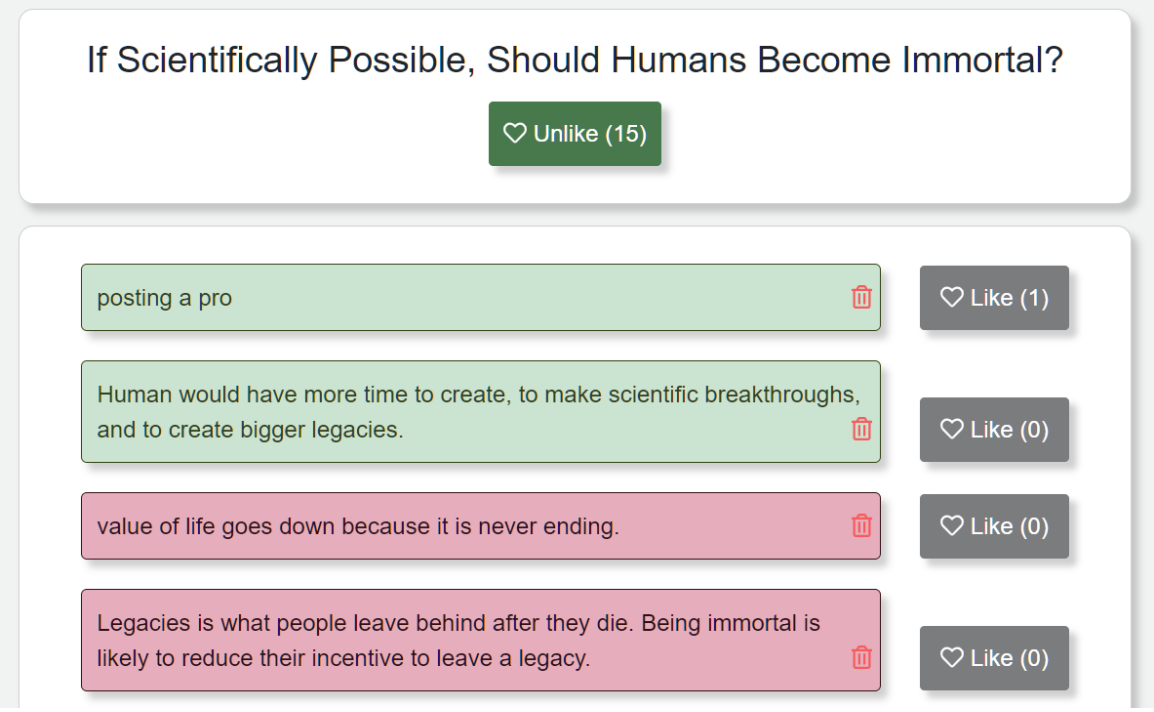
Here we have used try and catch to catch error and show it on console.

```
const connectDB = async () => {
    try {
        const conn = await mongoose.connect('mongodb+srv://dhavalsonani:xyz123@cluster0.el8ap.mongodb.
        net/lets_debate', {
            useNewUrlParser: true,
            retryWrites:true,
            useUnifiedTopology: true
        });
        console.log(`mongoDB connected: ${conn.connection.host}`);
    }
    catch(err){
        console.error(err);
    }
}
```

## Access Control

We have created 2 access control – User and Admin. User gets normal rights where one can signup for application, login in application, create-comment-see debates. While admin also has to signup, but admin's email address is already added to admin collection of database. So, whenever user logs in, application checks for admin email address and if admin is logged in, it gives rights to delete debates and comments to admin as shown in below pictures.




## Framework and External Code

We have not used any external framework except mentioned above. We have used some code from activities carried out throughout semester.