# **Actors**

## **Admin**

Admin of the website can:

* Add a Lesson
* Edit a Lesson
* Delete a Lesson
* Add a Quiz
* Edit a Quiz
* Delete a Quiz
* Manage the Students or Clients
* Delete a Student or Client

## **Student (Client)**

Students can access, read, and listen the lesson. They are able to write in the ***writing section*** when they want. They also can:

* Send a friend request to another student.
* Accept a friend request from another student.
* Chat only with friends.

They should pass the quiz to continue learning. A student can retake a lesson. Students can chat with a robot (*future*). A student can view and edit his/her profile. When a student want to stop learning, he/she can deactivate his/her account.

## **System**

The system can record the progress of a student. The system should have a chat application. Every sentence should have both the pronunciation and meaning. Every lesson should have a quiz. The quiz should cover all the contents of the lesson.

# **Database UML**

## **In Code**

Table users {

  id integer [primary key]

  username string

  password string

  LastName string

  firstName stirng

  Email string

  role boolean

  created\_at timestamp

  lesson\_id integer

}

Table lessons {

  id integer [primary key]

  title varchar

  deutsch text [note: 'Lessons Content in Deutsch']

  dari text [note: 'Lessons Content in Dari']

}

Table quiz {

  id integer [primary key]

  number integer

  quiz text

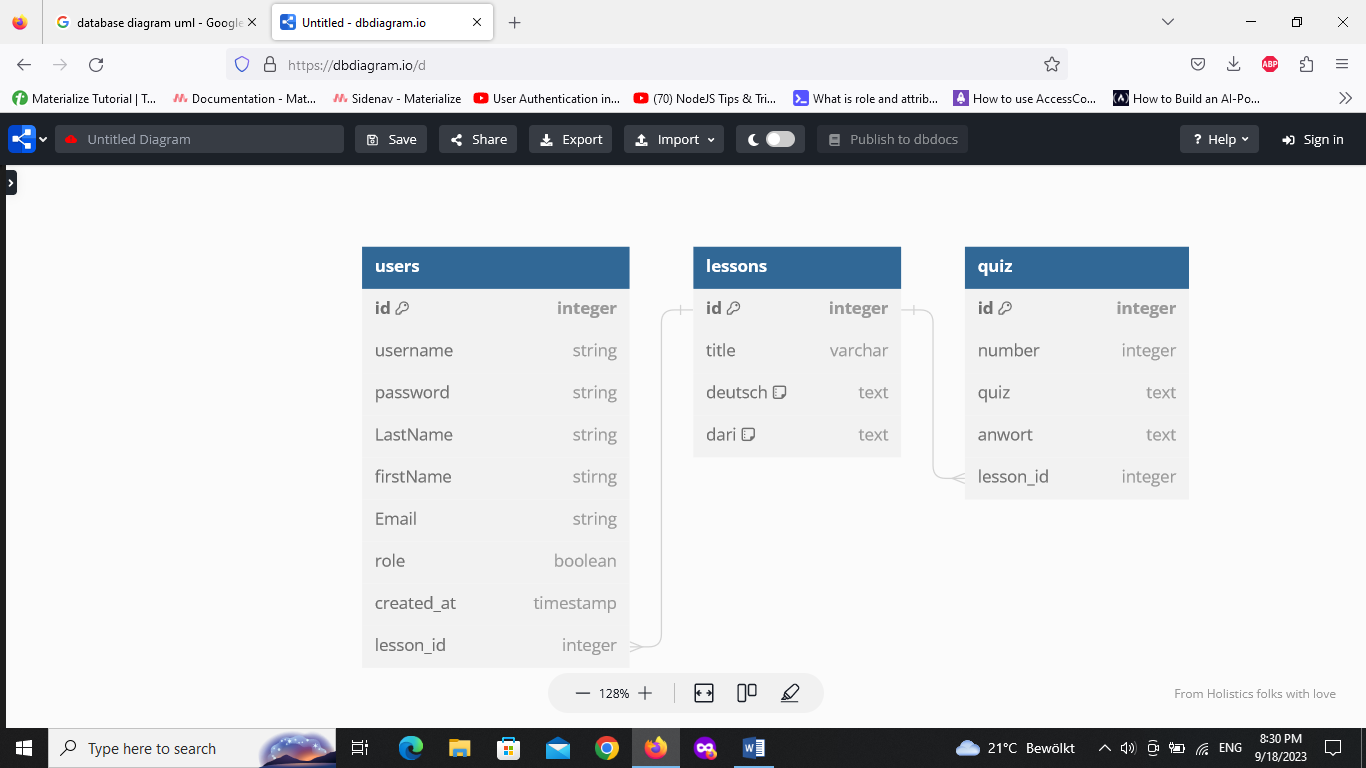
  answer text

  lesson\_id integer

}

Ref: users.lesson\_id > lessons.id // many-to-one

Ref: quiz.lesson\_id > lessons.id // many-to-one



# **Pseudocode**

## **Storing the environmental variables separately from the Project code:**

1. Install ‘dotenv’ library
2. Create a ‘.env’ file
3. Add environmental variables in the .env file
4. Import the .env file in other files that we want to use the environmental vairables

## **Creating a Server:**

1. Install ‘express’ framework
2. Import the ‘express’ framework into the app.js file
3. Set the ‘express’ app
4. Listen for requests on an specific PORT number

## **Database for the Project (MongoDB):**

1. Setup MongoDB locally / on Cloud (MongoDB Atlas). 🡪 We use MongoDB Atlas
2. Create an Account in the MongoDB Atlas
3. Built a Database
4. Create a User for the Database
5. Setup the network setting (Add your IP Address)

## **Connect to the Database:**

1. Save the Database link in the ‘.env’ File
2. Install ‘Mongoose’
3. Import ‘Mongoose’ in the Project file
4. Connect to the MongoDB database using Mongoose and Database link

## **View-Engine for the Project: (ejs)**

1. Install ‘ejs’ view engine
2. Register the ‘ejs’ view engine as the views for the Project 🡪 app.set(…);

## **View Files:**

1. Create different view files for different routes
2. Save the views files with the ‘.ejs’ extension
3. Render a view using a views file instead of HTML file
4. Pass data from handler into a view create dynamic Content

## **Reusing code in the View:**

Because all views should have navigation-bar, header, footer and flash-message.

1. Create a partial file in the view for the navigation-bar
2. Create a partial file for the header of views
3. Create a partial file for the footer of the views
4. Import the above files into all views files using ‘ejs’
5. Create a partial file to display flash-message
   1. It accept an object of messages
   2. Iterate inside the flash-messages
   3. Display the flash-messages to the end\_user
6. Style the created partial using CSS

## **Serving static/public files:**

1. Create a new folder in the project folder
2. Call the ‘express.static’ and pass the name of created folder, to make the created folder public.
3. Put public/static files into created folder. (CSS, img, …)

## **Database Schema and Model:**

User Schema and Model

1. Import Mongoose
2. Create the Database Schema for User
3. Create Database Model based on the Schema
4. Export the Model for using

Lesson Schema and Model

1. Import Mongoose
2. Create the Database Schema for Lesson
3. Create Database Model based on the Schema
4. Export the Model for using

## **User-Register Form:**

1. Create an ‘ejs’ file in the views folder
2. Insert a HTML Form with required fields and buttons for a User in the created file
3. Style the Form using CSS
4. Generate a route to the Users routes (from app.js)
5. Render the ‘ejs’ created file

When the User filled out the Form and sent to the server (Post Request):

1. Import User Model
2. Make a Post route
3. Get Inserted information
4. Validated the Inserted data
   1. Install ‘joi’
   2. Import “joi”
   3. Generate joi schema to validate a Form
5. Generate different flash-messages
   1. Install ‘connect-flash’ package
   2. Import ‘connect-flash’ package in the ‘app.js’
   3. Initializing Connect-Flash in the ‘app.js’
   4. Create a Middleware for Connect-Flash Messages
      1. To display messages into different pages
6. If everything is okay, Save it into an Object
7. Check that the user is not exists in the System
8. Encrypt the Password
   1. Install encrypt Package
   2. Import encrypt Package in the User Model File
9. Save the inserted data into the Database using User Model
10. Generate and display a Flash-Message
11. Go to the login page

## **User-Login Form:**

1. Create an ‘ejs’ file in the views folder
2. Insert a HTML Form with required fields and buttons in the created file
3. Style the Form using CSS
4. Generate a route to the Users routes (from app.js)
5. Render the ‘ejs’ created file

When the User filled out the login Form and sent login data to the server (Post Request):

1. Import User Model
2. Make a Post route
3. Get Inserted login data
4. Validate the Inserted data
5. Generate different flash-message
6. Everything is okay, Save it into an Object
7. Find the user by Email Address
8. Check if the user exists, then Compare the Password entered by User and Password in the database (Hashed Password)
9. If everything is okay, Generate flach-message
10. Route the user to the Home page, otherwise reload the Login page

## **Create Lesson Page:**

* When the Users click on the ‘Create Lessons’ menu, they should see ‘Creating a new Lesson’ page which has a Form for inserting Lesson data with a ‘Create Lesson’ button
* When the user filled out the Form and click ‘Create Lesson’ button
* The data should be saved in the Database
* The system should display a success message to the User

1. Create an ‘ejs’ file in the views folder
2. Insert a HTML Form with required fields and buttons for inserting a Lesson in the created file
3. Style the Form using CSS
4. Generate a route to the Lesson routes (from app.js)
5. Render the ‘ejs’ created file

When the User filled out the Form and sent data to the server (Post Request):

1. Make a Post route
2. Import Lesson Model
3. Get Inserted information and save it into an Object
4. Check that the Lesson number is not exists in the System
5. Save the inserted data into the Database using Lesson Model
6. Generate a flash-message
7. Go to the ‘create-quiz’ page

## **Create a Quiz Page:**

* When the Users click on the ‘Create Quizes’ menu, they should see ‘Creating A Quiz For A Lesson’ page which has a Form for inserting Quiz data with a ‘Create Quiz’ button.
* When the user filled out the Form and click ‘Create Quiz’ button.
* The data should be saved in the Database
* The system should send a success /error message to the user

1. Create an ‘ejs’ file in the views folder
2. Insert a HTML Form with required fields and buttons for inserting a Quiz in the created file
3. Style the Form using CSS
4. Generate a route to the Lesson routes (from app.js)
5. Render the ‘ejs’ created file

When the User filled out the Form and sent data to the server (Post Request):

1. Make a Post route
2. Import Lesson Model
3. Get Inserted information and save it into an Object
4. Query the Lesson form the Database using ‘Lesson No’
5. Update the Lesson
6. Generate a flash-message
7. Route the User to the Lessons page

## **Lesson Page:**

* When the Users click on the ‘Lessons’ menu, they should see ‘All Lessons’ page which has a list of Lessons.
* Each Lesson is displayed only with its Number and its Title.
* When the user click on a Lesson, that Lesson will be opened and this page will have all details of the Lesson.
* In the end of Each Lesson, There is a ‘Quiz’ button that the user can take a Quiz when the Lesson is finished.

1. Create an ‘ejs’ file in the views folder
2. Import the ‘Partial’ files
3. Create a loop for render object (Lessons) to display all of the lessons
4. Style the Page and Lessons using CSS
5. Generate a route to the Lesson routes (from app.js)
6. In the route file, import the Lesson Model
7. Query all Lessons from the Database using Lesson Model
8. Save the Lessons/query into an Object
9. Render the ‘ejs’ created file and the Object that have all the Lessons

## **Quiz Page:**

* When the Users click on the ‘Quiz’ button, they should see ‘Lessons number Quizes’ page which display all Quizes of a Lesson.
* User can Answer the Quizes and submit the Answers
* The Server calculate the result
* Send the Result as a Flash Message
* If the User passed the Quizes, the User can continue learning and learn next lesson.

1. Create an ‘ejs’ file in the views folder
2. Import the ‘Partial’ files
3. Create a loop for render object (Quizes) to display all of the Quizes of a Lesson
4. Style the Page using CSS
5. Generate a route to the Lesson routes (from app.js)
6. In the route, import Lesson Model
7. Query all Quizes of a Lesson from the Database
8. Save the Quizes into an Object
9. Sort Quizes based on the QuizNumber
10. Render the ‘ejs’ created file and the Object that have all the Quizes of a Lesson

## **Sessions and Cookies:**

* Configure the Session and Cookies in the Project. Use Session to maintain the state.

1. Install ‘express-session’ Package
2. Import ‘express-session’ into the main file of the Project
3. Make Session and Cookies 🡪 Using app.use()

## **Passport Local Strategy for the Project:**

* Users must be authenticated using email and password by the System.

1. Install ‘passport’ Package
2. Install ‘passport-local’ Package
3. Import ‘passport’
4. Import ‘passport-local’
5. Create a separate file (.js) for Passport Local Strategy
   1. Configure Passport Local Strategy for the Project
   2. Use Passport serializes and deserializes user information to and from the session to maintain a login session,
6. In the main file of the application
   1. Initialize the Passport
   2. Establish Session by using built-in Passport session strategy
   3. Authenticate the Session
7. In the User routes file, Use **passport.authenticate()**, specifying the 'local' strategy, to authenticate User requests
   1. It is used as middleware in the Login raoute

## **Logout:**

* Users must be logout from the system when they click on the ‘Logout’ Menu.

1. Add a Logout menu in the Navigation bar
2. Create a logout route in the Users Route
3. Use request.logout() to logout users from the System
   1. It is a middleware from the Passport.
4. Generate a Flash-message for success or error
5. Route User to the main Page

## **Route Protection:**

We should have different web-pages for different User states:

* When the User state is before the login or after the logout. He/She can visit Home, About, Login, and Register pages.
* When the User is logged in the System and authorized successfully. He/She can visit Home, About, User Profile, logout, Lessons, Quizes, Create Lessons, and Create Quizes pages.

1. Create Two Middleware using **isAuthenticated()** middleware which is provided by the Passport.
   1. One Middleware for Unauthorized Users 🡪 Users before login and after logout
   2. One Middleware for Authorized Users
2. Now use these middleware to different routes as a Middleware based on their requirements
   1. For example for Register Page 🡪 Use Unauthorized Middleware, because logged in / authorized users do not need to re-register.

## **Admin Role:**

* Only Admin User can Create Lessons, and Quizes.

1. Create a Middleware for checking if the User is an admin or not
   1. If Not, route User to the previous Page
   2. If Yes, route the User to the selected Page
2. Now Use the Middleware into Project Routes

## **Dynamic Navigation Bar:**

* Different Menu should be display for different User states.
* Unauthorized User
  + Can see Home, About, Login, and Register pages.
* Authorized User and Not Admin
  + Can see Home, User Profile, About, logout, Lessons, and Quizes pages.
* Authorized User and Admin
  + Can see Home, About, User Profile, logout, Lessons, Quizes, Create Lessons, and Create Quizes pages.

1. Use If…else statement in the navigation bar partial file in the Views folder.

## **User Profile:**

* When a User click on ‘User Profile’ Menu
* The User Should see her/his information

1. Create a ‘ejs’ File in the Views
2. Import partials view files in the created file
3. Display User data from the Local User
4. Style the page using CSS
5. Create a route for the Profile Page in the User Route
6. Check that the user is Authenticated
7. If the User is Authenticated, Render the created ‘ejs’ file for the User
8. Otherwise route the User to the login page