CoBuild

System Design Document

Minjun Kim
Ansh Aneel
Andrew Aucie
Ashwin Malik
Dhruvin Patel
Vikram Nara
Vedat Gotkepe

Contents

Front-end

Class Name: CreatePost	
Parent Class (if any): N/A **Subclasses (if any):** N/A	
Responsibilities: ● Extracts text from fields in create job posting page and runs API call to connect to backend. ● Render a form for creating a job posting ● Handle user input for job details such as title, location, job description, company name, deadline, and skills ● Allow users to upload files for the job posting ● Handle the submission of the job posting form ● Display a grid of job postings from a predefined list	**Collaborators:** ● React ● RxOpenInNewWindow (from 'react-icons/rx') ● Container, Grid, Box, Typography, alpha, Button, TextField, IconButton (from '@mui/material') ● CloudUpload (from '@mui/icons-material') ● './CreatePost.css' (custom CSS file) ● JobGrid component

React Component(s): JobPosting	
Parent Class (if any): React.Component **Subclasses (if any):** N/A	
Responsibilities: ● Render the view for a job posting ● Calculate the time remaining until the application deadline ● Format time values for display ● Fetch job post data from the server using the provided API URL and post ID ● Set the state variables for position name, company name, location, description, tags, and target date ● Display the remaining time until the application deadline ● Display the position name, company name, location, description, and tags of the job posting	**Collaborators:** ● React (useState, useEffect) ● react-router-dom (useParams)

React Component(s): Jobs	
Parent Class (if any): React.Component **Subclasses (if any):** N/A	
Responsibilities: ● Render the view for a list of jobs ● Fetch job IDs from the server using the provided API URL ● Set the state variable for job IDs ● Pass the job IDs as props to the JobPosting component	**Collaborators:** ● React (useState, useEffect) ● react-router-dom (Link) ● JobPosting component

React Component(s): Login
Parent Class (if any): React.Component **Subclasses (if any):** N/A

Responsibilities:	Collaborators:
<ul><li>Render the view for the login page</li><li>Manage state variables for username, password, and login error message</li><li>Handle changes in the username and password fields</li><li>Handle form submission for login</li><li>Make a POST request to the server to authenticate the user</li><li>Set the login error message if authentication fails</li><li>Use the react-auth-kit library to handle user authentication and store the access token</li><li>Redirect the user to the jobs page upon successful login</li></ul>	<ul><li>React (useState)</li><li>@mui/material (AppBar, Avatar, Typography, Button, TextField, BottomNavigation, Toolbar, InputAdornment)</li><li>react-router-dom (Link, useNavigate)</li><li>react-auth-kit (useSignIn</li></ul>

React Component(s): Signup

Parent Class (if any): N/A
Subclasses (if any): N/A

Responsibilities:	Collaborators:
<ul><li>Render a sign-up form with input fields for name, email, username, password, and other information.</li><li>Handle user input changes for name, email, username, password, and other fields.</li><li>Handle form submission and send a sign-up request to the server.</li><li>Reset input fields after successful sign-up or display an error message on failure.</li><li>Apply styles and animations to the form and input fields.</li><li>Navigate to the login page when the "Sign In" button is clicked.</li></ul>	<ul><li>React: Used to create functional components and manage component state.</li><li>react-router-dom: Used for navigation to the login page.</li><li>useState: Hook for managing state variables.</li><li>Fetch API: Used for making HTTP requests to the server</li></ul>

React Component(s): Assessment

Parent Class (if any): N/A
Subclasses (if any): N/A

Responsibilities:	**Collaborators:**
● Assess users' code submission and run their code against test cases. ● Displays a leetcode type problem and a code editor for users to solve the problem. ● Fetch assessment data from the server using fetch API. ● Manage state for assessment properties such as id, title, description, boilerCode, code, testCases, exampleCases, datePosted, jobId. ● Handle user interactions and update state accordingly for selected problem and input code value. ● Manage the timer using useTimer hook and handle timer expiration events. ● Handle user code submissions and communicate with the server for compilation and testing. ● Display problem descriptions, example cases, and submission details based on the selected tab. ● Keep track of user submissions and display them along with the results.	● java: programming language for running test cases. ● React Router (useParams): To get the id from the URL parameters. ● CodeMirror component: To provide a code editor with syntax highlighting and editing capabilities. ● dracula theme from @uiw/codemirror-theme -dracula: To style the code editor. ● codemirrorPython from @codemirror/lang-pytho n: To enable Python language support in CodeMirror. ● react-timer-hook: To manage and display the timer. ● fetch API: To make HTTP requests to the server for fetching assessment data and submitting code.

React Component(s): Leaderboard

Parent Class (if any): N/A
Subclasses (if any): N/A

Responsibilities:	**Collaborators:**
● Displays a leaderboard of users who completed a specific leetcode type problem. ● Ranks you amongst other applicants, displaying statistics like time taken, complexity, and the number of test cases passed.	● JobPosting: Represents a job posting with details like title, description, and requirements. ● User: Represents the list of users in the leaderboard. ● Backend API: Collaborates with the backend to fetch scores of users

React Component(s): SignUpRecruiter	
Parent Class (if any): N/A **Subclasses (if any):** N/A	
Responsibilities: ● Allows recruiters to sign up using their company name, job position, recommended skills, field, etc. ● Upload relevant files or documents	**Collaborators:** ● useState: Hook for managing state variables. ● materialUI

React Component(s): SubmissionForm	
Parent Class (if any): React.Component **Subclasses (if any):** N/A	
Responsibilities: ● Render the form for users to submit their application for a specific job posting ● Validate the input data entered by the user ● Submit the application to the backend for processing	**Collaborators:** \`User\` (model) \`JobPosting\` \`BackendAPI\`

React Component(s): ApplicationDialog	
Parent Class (if any): React.Component **Subclasses (if any):** N/A	
Responsibilities: ● Display dialogues for when you apply to a job posting.	**Collaborators:** \`User\` (model): Provides information about the user receiving the notification \`Recruiter\` (model): Provides information about the recruiter receiving the notification \`BackendAPI\`

React Component(s): BottomNavBar	
Parent Class (if any): N/A **Subclasses (if any):** N/A	
Responsibilities: ● Provides a bottom navigation bar to be used throughout the app.	**Collaborators:** ● @mui/materialUI

React Component(s): JobBox	
Parent Class (if any): React.Component **Subclasses (if any):** N/A	
Responsibilities: ● Provides a card template for job postings, a company image above a job description	**Collaborators:** ● makeStyles ● @mui/material

React Component(s): Step1	
Parent Class (if any): React.Component **Subclasses (if any):** Step2, Step3, Step4	
Responsibilities: ● Manage the state for selectedLanguages and activeStep. ● Render different steps of the multi-step form based on the active step. ● Handle the selection of tools/languages using an autocomplete input with chips. ● Communicate with the server to update the user's skills with selected languages. ● Handle navigation to the next or previous steps in the form. ● Navigate to the final page when the form is complete. ●	**Collaborators:** ● React (for functional components and hooks) ● "react-router-dom" (for navigation) ● "@mui/material" (for Material-UI components) ● "@mui/system" (for styled components) ●

React Component(s): UserProfile	
Parent Class (if any): React.Component **Subclasses (if any):** N/A	
Responsibilities: ● Displays a profile of a user, showing saved and applied applications as well as their user information such as university, courses, etc.	**Collaborators:** ● makeStyles ● JobBox: to incorporate job postings

React Component(s): UploadAssessment	
Parent Class (if any): React.Component **Subclasses (if any):** N/A	
Responsibilities: ● Manage the state for activeStep, category, title, description, solution, testCases, codeEditorState, solutionValue, and testcasesValue. ● Handle changes in category, title, description, solution, test cases. and code editor. ● Highlight the code in the code editor. ● Handle form submission and make an API call to create the assessment..	**Collaborators:** ● React (for functional components and hooks) ● "@uiw/react-codemirror" (for CodeMirror component) ● "@uiw/codemirror-theme-dracula" (for the Dracula theme of CodeMirror) ● "@codemirror/lang-json" (for JSON language support in CodeMirror) ● "react-draft-wysiwyg" (for the rich text editor) ● "draft-js" (for Draft.js support) ● "@mui/material" (for Material-UI components) ● "@mui/icons-material" (for Material-UI icons) ● "react-syntax-highlighter" (for code syntax highlighting) ● "highlight.js" (for Python language support) ● Backend API (for making API

	calls)

React Component(s): Verification	
Parent Class (if any): React.Component **Subclasses (if any):** N/A	
Responsibilities: ● Manage the state for uniqueString, message, and isValid. ● Fetch verification data from the server using uniqueString. ● Update the message and isValid based on the server response. ● Render the UI based on the verification status (valid or not). ● Handle the "Login" button click and navigate to the login page.	**Collaborators:** ● React (for functional components and hooks) ● "@mui/material" (for Material-UI components) ● "react-router-dom" (for handling URL parameters and navigation)

React Component(s): RecruiterSteps1	
Parent Class (if any): React.Component **Subclasses (if any):** RecruiterSteps2, RecruiterSteps3	
Responsibilities: ● Manage the state for selectedLanguages, activeStep, company, description, location, and uploadedFiles. ● Render different steps of the recruitment process based on the active step. ● Handle changes in the form fields for company information (name, description, location). ● Handle the selection of programming languages. ● Handle the file upload for the company logo. ● Communicate with the server to update the user's skills with selected languages. ● Handle navigation to the next or previous steps in the	**Collaborators:** ● React (for functional components and hooks) ● "react-router-dom" (for navigation) ● "@mui/material" (for Material-UI components) ● "@mui/styles" (for custom styles) ● "@mui/system" (for styled components)

recruitment process. ● Navigate to the final page when the process is complete.	

Back-end

Class Name: JobSchema	
Parent Class (if any): None **Subclasses (if any):** None	
Responsibilities: ● Store and define the structure of a job document in the database. ● Define the fields and their data types for a job document, including: o jobId: String o title: String o location: String o jobDescription: String o deadline: String o companyName: String o datePosted: String o skills: Array of strings	**Collaborators:** mongoose

Class Name: UserLoginSchema	
Parent Class (if any): None **Subclasses (if any):** None	
Responsibilities: ● Store and define the structure of a user login document in the database. ● Define the fields and their data types for a user login document, including: o username: String (required) o password: String (required)	**Collaborators:** mongoose

Class Name: SignUpSchema

Parent Class (if any):
Subclasses (if any):	

Responsibilities: ● Store and define the structure of a sign-up document in the database. ● Define the fields and their data types for a sign-up document, including: o username: String o email: String o name: String o password: String o resume: String o transcript: String	**Collaborators:** mongoose

Class Name: Server.js	

Parent Class (if any):
Subclasses (if any):	

Responsibilities: ● Create and configure an Express application. ● Set up routes for various API endpoints. ● Establish a connection to the MongoDB database. ● Start the server and listen on port 8000. ● Handle incoming HTTP requests and route them to the corresponding API handlers.	**Collaborators:** express: Express framework mongoose: MongoDB library cors: Cross-origin resource sharing library app: Express application instance assesmentApi: "/createassesment" endpoint handler getPost: "/getpost" endpoint handler createPost: "/createpost" endpoint handler userLogin: "/login" endpoint handler signUpRequest: "/signup" endpoint handler

Class Name: Assessment

Parent Class (if any):
Subclasses (if any):

Responsibilities:	Collaborators:
● Create and define the assessment schema in MongoDB ● Handle the POST request for creating a new assessment ● Generate a UUID for the assessment ● Save the new assessment in the database ● Define the schema for the "assessments" collection in MongoDB using Mongoose. ● Define the properties and data types for assessment documents, such as jobId, title, description, exampleCases, and testCases. ● Provide a method to save a new assessment document in the "assessments" collection.	mongoose uuidv4 AssesmentModel

Class Name: getpost.js	

Parent Class (if any):
Subclasses (if any):	

Responsibilities: ● Define the job schema in MongoDB ● Handle the POST request for creating a new job posting ● Generate a unique job ID using short-uuid and uuidv4 ● Convert the deadline to ISO format ● Save the new job posting in the database	**Collaborators:** mongoose uuidv4 short jobschema job

Class Name: Loginuser.js	

Parent Class (if any):
Subclasses (if any):	

Responsibilities: ● Define the user login schema in MongoDB ● Handle the POST request for user login ● Validate the provided username and password ● Generate an access token using JWT ● Return the access token upon successful login	**Collaborators:** userLoginSchema mongoose jwt dotenv

Class Name: Signup.js

Parent Class (if any):
Subclasses (if any):

Responsibilities:	Collaborators:
● Define the user sign-up schema in MongoDB ● Handle the POST request for user sign-up ● Validate the uniqueness of the email and username ● Create a new user instance and save it to the database ●	signUpSchema mongoose

Class Name: Assessments	
Parent Class (if any): None **Subclasses (if any):** None	
Responsibilities: ● Maintain a leaderboard of users based on their assessment scores ● Calculate and update user rankings based on assessment scores ● Store user information, such as username and assessment scores ● Provide methods to add new users and update their assessment scores ● Retrieve and display the leaderboard in descending order of scores	**Collaborators:** User: Represents a user in assessments Assessment: Represents an assessment with score access Database: Stores user info and assessment scores Sorting algorithm: Sorts leaderboard based on scores

Class Name: createAssessment	
Parent Class (if any): None **Subclasses (if any):** None	
Responsibilities: ● Implement two API handlers, createAssessmentApi and getAssessmentApi, for creating and fetching assessments. ● Handle incoming HTTP requests for creating and fetching assessments. ● Extract data from the request body, such as title, description, code, testCases, exampleCases, jobId, and boilerCode, as required for creating a new assessment.	**Collaborators:** mongoose: The class interacts with Mongoose to define the assessment schema and communicate with the MongoDB database.

● Generate a unique assessmentId using short-uuid and store the assessment data into the database using Mongoose. ● Retrieve assessment data from the database based on the provided jobId using the getAssessmentApi handler. ● Update the isAssessment property of the corresponding job document to true using the createAssessmentApi handler.	uuidv4: The class uses uuidv4 from the uuid library to generate a unique assessmentId for each assessment. short-uuid: The class uses short-uuid to generate short and unique IDs for the assessment objects. Assessment: The class requires the Assessment schema to create new assessment objects. jobSchema: The class uses the jobSchema to define the job model, which represents job documents stored in the database. Job: The class interacts with the Job model to find job documents based on the provided jobId.

Class Name: createPost
Parent Class (if any): None **Subclasses (if any):** None

Responsibilities:	**Collaborators:**
● Implement an API endpoint handler, createPostApi, for creating job postings. ● Handle incoming HTTP requests for creating job postings. ● Extract data from the request body, such as title, location, jobDescription, companyName, deadline, and skills, required for creating a new job posting. ● Generate a unique jobId using short-uuid for each job posting. ● Convert the provided deadline to ISO format using new Date(deadline).toISOString() before storing it in the database. ● Save the job posting data, along with the generated jobId, into the database using Mongoose.	mongoose: The class interacts with Mongoose to define the job schema and communicate with the MongoDB database. uuidv4: The class uses uuidv4 from the uuid library to generate a unique jobId for each job posting. short-uuid: The class uses short-uuid to generate short and unique IDs for the job postings. jobSchema: The class uses the jobSchema to define the job model, which represents job documents stored in the database. Job: The class interacts with the Job model to save new job postings.

Class Name: postBookmarkJob
Parent Class (if any): None **Subclasses (if any):** None

Responsibilities:	**Collaborators:**
● Implement an API endpoint handler, postBookmarkJob, for bookmarking/unbookmarking job postings for a user. ● Handle incoming HTTP requests for bookmarking/unbookmarking job postings. ● Extract data from the request body, such as username and jobId, required for bookmarking/unbookmarking a job posting. ● Find the user with the provided username in the database using Mongoose. ● Find the job with the provided jobId in the database using Mongoose. ● Check if the user and job exist, and return appropriate error responses if not found. ● Maintain the bookmarked job IDs for the user and add or remove the provided jobId accordingly. ● Save the updated user data with the bookmarked job IDs into the database using Mongoose.	mongoose: The class interacts with Mongoose to define the user and job schemas and communicate with the MongoDB database. signUpSchema: The class uses the signUpSchema to define the user model, which represents user documents stored in the database. jobSchema: The class uses the jobSchema to define the job model, which represents job documents stored in the database. User: The class interacts with the User model to find and update user documents. Job: The class interacts with the Job model to find job documents. req: The class handles incoming HTTP requests to the API endpoint.

Class Name: postEmailVerification
Parent Class (if any): None **Subclasses (if any):** None

Responsibilities:	Collaborators:
● Implement an API endpoint handler, verifyEmail, for verifying user emails based on a unique verification token. ● Handle incoming HTTP requests for email verification. ● Extract the uniqueString from the request body, which represents the verification token provided by the user. ● Find the user with the provided uniqueString in the database using Mongoose. ● Check if the user exists and return appropriate error responses if not found. ● Check if the user's email has already been verified and return an error response if it is already verified. ● Check if the verification token has expired and return an error response if it has. ● Update the user's verification.verified field to true to indicate that the email has been verified. ● Save the updated user data with the verified email status into the database using Mongoose.	mongoose: The class interacts with Mongoose to define the user schema and communicate with the MongoDB database. signUpSchema: The class uses the signUpSchema to define the user model, which represents user documents stored in the database. bcrypt: The class uses bcrypt for password hashing and authentication. User: The class interacts with the User model to find and update user documents. req: The class handles incoming HTTP requests to the API endpoint.

Class Name: removeBookmarkJob
Parent Class (if any): None **Subclasses (if any):** None

Responsibilities:	**Collaborators:**
• Implement an API endpoint handler, removeBookmarkJob, for removing a bookmarked job from a user's list of bookmarked jobs. • Handle incoming HTTP requests for removing a bookmarked job. • Extract the username and jobId from the request body. • Find the user with the provided username in the database using Mongoose. • Populate the bookmarkedJobsIds field of the user to include the full job information using Mongoose's populate method. • Check if the user exists and return an error response if not found. • Find the job with the provided jobId in the database using Mongoose. • Check if the job exists and return an error response if not found. • Validate if the jobId is a valid MongoDB ObjectId. • Convert the jobId to a MongoDB ObjectId. • Check if the jobId exists in the bookmarkedJobsIds array of the user. • If the jobId exists, remove the job object from the bookmarkedJobsIds array. • Save the updated user data with the removed bookmarked job in the database using Mongoose.	mongoose: The class interacts with Mongoose to define the user and job schemas and communicate with the MongoDB database. signUpSchema: The class uses the signUpSchema to define the user model, which represents user documents stored in the database. jobSchema: The class uses the jobSchema to define the job model, which represents job documents stored in the database. User: The class interacts with the User model to find and update user documents. Job: The class interacts with the Job model to find job documents. req: The class handles incoming HTTP requests to the API endpoint.

Class Name: getJobPost
Parent Class (if any): None **Subclasses (if any):** None

Responsibilities:	Collaborators:
● Implement an API endpoint handler, getJobPost, for retrieving job posts based on different criteria such as id, requestType, etc. ● Handle incoming HTTP requests for fetching job posts. ● Extract relevant parameters from the request body, such as id and requestType. ● Query the database using Mongoose based on the specified criteria to retrieve job posts. ● Return the retrieved job posts as JSON responses.	mongoose: The class interacts with Mongoose to define the job post schema and communicate with the MongoDB database. jobSchema: The class uses the jobSchema to define the job post model, which represents job post documents stored in the database. Job: The class interacts with the Job model to query the database and retrieve job post documents. req: The class handles incoming HTTP requests to the API endpoint. res: The class sends HTTP responses to the client.

Class Name: getUser
Parent Class (if any): None **Subclasses (if any):** None

Responsibilities:	**Collaborators:**
● Implement an API endpoint handler, getUser, for retrieving user information based on the provided username. ● Handle incoming HTTP requests for fetching user information. ● Extract the username parameter from the request body. ● Query the database using Mongoose to find the user with the given username. ● Populate the user's bookmarkedJobsIds and appliedJobsIds fields from their references. ● Return the user information, including the populated bookmarked and applied jobs, as a JSON response.	mongoose: The class interacts with Mongoose to define the user schema and communicate with the MongoDB database. signUpSchema: The class uses the signUpSchema to define the user model, which represents user documents stored in the database. User: The class interacts with the User model to query the database and retrieve user documents. req: The class handles incoming HTTP requests to the API endpoint. res: The class sends HTTP responses to the client.

Class Name: loginUser
Parent Class (if any): None **Subclasses (if any):** None

Responsibilities:	Collaborators:
• Implement an API endpoint handler, userLogin, for user authentication during login. • Handle incoming HTTP requests for user login. • Extract the username and password parameters from the request body. • Query the database using Mongoose to find the user with the provided username. • Compare the entered password with the hashed password stored in the database using bcrypt. • Check if the user's email is verified before allowing login. • If login is successful, generate a JWT access token for the user. • Return appropriate JSON responses for different scenarios during the login process.	mongoose: The class interacts with Mongoose to define the user schema and communicate with the MongoDB database. signUpSchema: The class uses the signUpSchema to define the user model, which represents user documents stored in the database. bcrypt: The class uses bcrypt for comparing the entered password with the hashed password in the database. jwt: The class uses JWT to generate an access token for authenticated users. Login: The class interacts with the Login model to query the database and find user documents. req: The class handles incoming HTTP requests to the API endpoint. res: The class sends HTTP responses to the client.

Class Name: getLeaderboard
Parent Class (if any): None **Subclasses (if any):** None

Responsibilities:	**Collaborators:**
● Implement an API endpoint handler, getLeaderboard, for retrieving a leaderboard of applications based on the given jobID and days. ● Handle incoming HTTP requests for fetching the leaderboard. ● Extract the jobID and days parameters from the request body. ● Query the database using Mongoose to find the job document with the given jobID. ● Calculate the date that is days ago from the current date. ● Retrieve applications for the specified job with submissionTime greater than or equal to the calculated date and with codingQuestionStatus set to "done". ● Sort the applications based on the score, codingQuestionResult.complexity, and codingQuestionResult.time. ● Return the applications as a JSON response.	mongoose: The class interacts with Mongoose to define the application schema and communicate with the MongoDB database. applicationSchema: The class uses the applicationSchema to define the application model, which represents application documents stored in the database. jobSchema: The class uses the jobSchema to define the job model, which represents job documents stored in the database. Application: The class interacts with the Application model to query the database and retrieve application documents. Job: The class interacts with the Job model to query the database and retrieve job documents. req: The class handles incoming HTTP requests to the API endpoint. res: The class sends HTTP responses to the client.

Class Name: createAssessment
Parent Class (if any): None **Subclasses (if any):** None

Responsibilities:	Collaborators:
● Implement an API endpoint handler, createAssessmentApi, for creating a new assessment based on the provided data. ● Handle incoming HTTP requests for creating assessments. ● Extract the assessment data, such as title, description, code, testCases, exampleCases, jobId, and boilerCode, from the request body. ● Create a new assessment object using the Assessment model and the extracted data. ● Retrieve the job document based on the provided jobId. ● If the job document is not found, return an error response. ● Set the isAssessment property of the job document to true or update it if already present. ● Save the updated job document to the database. ● Save the new assessment object to the database. ● Return the saved assessment as a JSON response.	mongoose: The class interacts with Mongoose to define the assessment schema and communicate with the MongoDB database. assessmentSchema: The class uses the assessmentSchema to define the assessment model, which represents assessment documents stored in the database. jobSchema: The class uses the jobSchema to define the job model, which represents job documents stored in the database. Assessment: The class interacts with the Assessment model to create and save assessment documents to the database. Job: The class interacts with the Job model to query the database and retrieve job documents. req: The class handles incoming HTTP requests to the API endpoint. res: The class sends HTTP responses to the client.

Class Name: createPost
Parent Class (if any): None **Subclasses (if any):** None

Responsibilities:	Collaborators:
● Implement an API endpoint handler, createPostApi, for creating a new job post based on the provided data. ● Handle incoming HTTP requests for creating job posts. ● Extract the job post data, such as title, location, jobDescription, companyName, deadline, and skills, from the request body. ● Generate a new jobId using short.generate() from the short-uuid library. ● Convert the deadline date to ISO format and get the current date for the `datePosted.	mongoose: The class interacts with Mongoose to define the assessment schema and communicate with the MongoDB database. assessmentSchema: The class uses the assessmentSchema to define the assessment model, which represents assessment documents stored in the database. jobSchema: The class uses the jobSchema to define the job model, which represents job documents stored in the database. Job: The class interacts with the Job model to query the database and retrieve job documents. req: The class handles incoming HTTP requests to the API endpoint. res: The class sends HTTP responses to the client.

Class Name: applicationSchema

Parent Class (if any): None
Subclasses (if any): None

Responsibilities:	Collaborators:
● Define the applicationSchema using Mongoose to represent application documents stored in the MongoDB database. ● Specify the fields and their types for the applicationSchema, including username, job, jobId, submissionTime, codingQuestionStatus, and codingQuestionResult. ● Set validation rules for the required fields (username, job), default values (submissionTime, codingQuestionStatus, codingQuestionResult.score, codingQuestionResult.complexity, codingQuestionResult.time), and allowed enum values (codingQuestionStatus).	mongoose: The class interacts with Mongoose to define the assessment schema and communicate with the MongoDB database.

Class Name: assessmentSchema	
Parent Class (if any): None **Subclasses (if any):** None	
Responsibilities: ● Define the assessmentSchema using Mongoose to represent assessment documents stored in the MongoDB database. ● Specify the fields and their types for the assessmentSchema, including assessmentId, title, description, boilerCode, code, testCases, exampleCases, datePosted, and jobId.	**Collaborators:** mongoose: The class interacts with Mongoose to define the assessment schema and communicate with the MongoDB database.

Class Name: recruiterSchema	
Parent Class (if any): None **Subclasses (if any):** None	
Responsibilities: ● Define the recruiterSchema using Mongoose to represent recruiter documents stored in the MongoDB database. ● Specify the fields and their types for the recruiterSchema, including username, password, email, name, links, jobCategories, and positionList. ● Set validation rules for the required fields (username, password, email, name).	**Collaborators:** mongoose: The class interacts with Mongoose to define the assessment schema and communicate with the MongoDB database.

Class Name: codeExecutionStrategy
Parent Class (if any): None **Subclasses (if any):** None

Responsibilities: ● Define the CodeExecutionStrategy interface with two abstract methods: executeCode(code, tests) and evaluateCode(code, memoryUsage, timeUsage, tests). ● The executeCode(code, tests) method should be implemented by concrete strategy classes to execute the provided code using a specific programming language (e.g., Python, JavaScript, etc.) and return the results of the execution (e.g., passing test cases, runtime, memory, etc.). ● The evaluateCode(code, memoryUsage, timeUsage, tests) method should be implemented by concrete strategy classes to evaluate the execution of the provided code based on various factors, such as test cases passed, runtime, memory usage, clean code, time taken to write the code, etc.	**Collaborators:** PythonExecutionStrategy: A specific concrete implementation of the CodeExecutionStrategy interface for executing Python code.

Class Name: PythonExecutionStrategy	
Parent Class (if any): None **Subclasses (if any):** None	
Responsibilities: ● Define the PythonExecutionStrategy class that is a concrete implementation of the CodeExecutionStrategy interface for executing Python code. ● The class implements the executeCode(code, tests) method to execute the provided Python code for each test case in the tests array. ● The execution of Python code is done using the PythonShell library, which provides a Python interpreter in Node.js. ● The executeCode(code, tests) method returns the percentage of test cases passed by the Python code.	**Collaborators:** CodeExecutionStrategy: The class is a specific concrete implementation of the CodeExecutionStrategy interface.

Class Name: s3
Parent Class (if any): None **Subclasses (if any):** None

Responsibilities:	**Collaborators:**
• Define the generateUploadURL function to generate a pre-signed URL for uploading files to an AWS S3 bucket. • The function takes the username, type, and extension of the file as input and returns a pre-signed URL that allows uploading the file to the specified S3 bucket. • The function uses the randomBytes function from the crypto module to generate a random name for the file. • The generated URL is valid for 60 seconds (Expires: 60) to enable secure file uploads.	dotenv: The function uses dotenv to load environment variables from a .env file. aws-sdk: The function uses the aws-sdk library to interact with AWS services. crypto: The function uses the crypto module to generate random bytes for creating a unique file name. randomBytes: The function uses the randomBytes function from the crypto module to generate random bytes.

Dependencies:

1. "@codemirror/lang-cpp": "^6.0.2",
2. "@codemirror/lang-java": "^6.0.1",
3. "@codemirror/lang-javascript": "^6.1.9",
4. "@codemirror/lang-python": "^6.1.3",
5. "@emotion/react": "^11.11.1",
6. "@emotion/styled": "^11.11.0",
7. "@fontsource/work-sans": "^5.0.3",
8. "@monaco-editor/react": "^4.5.1",
9. "@mui/icons-material": "^5.11.16",
10. "@mui/material": "^5.13.6",
11. "@mui/styles": "^5.13.2",
12. "@testing-library/jest-dom": "^5.16.5",
13. "@testing-library/react": "^13.4.0",
14. "@testing-library/user-event": "^13.5.0",
15. "@uiw/codemirror-theme-dracula": "^4.21.7",
16. "@uiw/react-codemirror": "^4.21.7",
17. "monaco-editor": "^0.39.0",
18. "react": "^18.2.0",
19. "react-auth-kit": "^2.12.2",

```
    "react-dom": "^18.2.0",
    "react-icons": "^4.9.0",
    "react-router-dom": "^6.12.1",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
    "aws-sdk": "^2.1412.0"
    "bcrypt": "^5.1.0"
    "cors": "^2.8.5"
    "dotenv": "^16.2.0"
    "express": "^4.18.2"
    "jsonwebtoken": "^9.0.0"
    "lodash": "^4.17.21"
    "mailgen": "^2.0.27"
    "mongodb": "^5.6.0"
    "mongoose": "^7.2.2"
    "multer": "^1.4.4"
    "multer-gridfs-storage": "^5.0.2"
    "nodemailer": "^6.9.3"
    "nodemon": "^2.0.22"
    "openai": "^3.3.0"
    "python-shell": "^5.0.0"
    "short-uuid": "^4.2.2"
    "uuid": "^9.0.0"
```

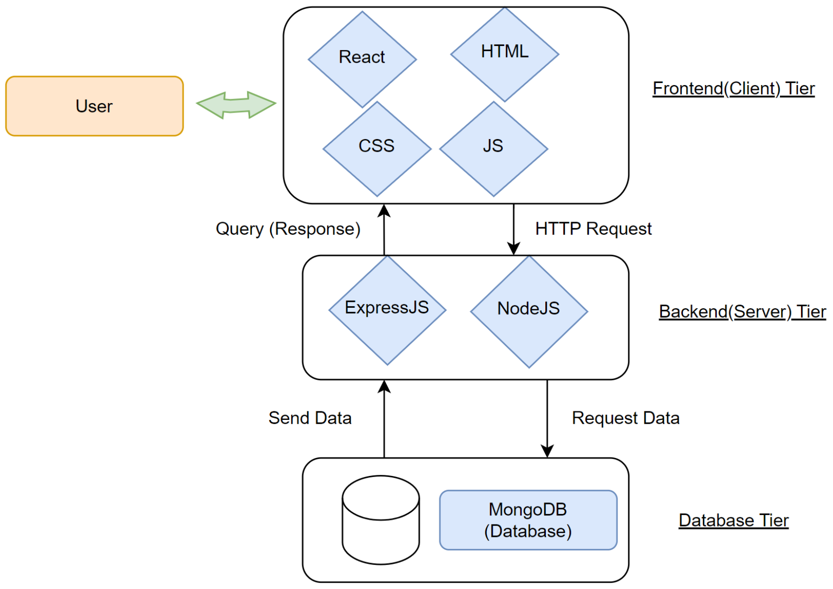
Project Configuration Summary:

Operating System: Any
Languages: JS, JSX
Compiler: BABEL
Virtual Machine: None
Databases: MongoDB, S3 Blob Storage
Server port: 8000
Frontend port: 3000

Software Architecture

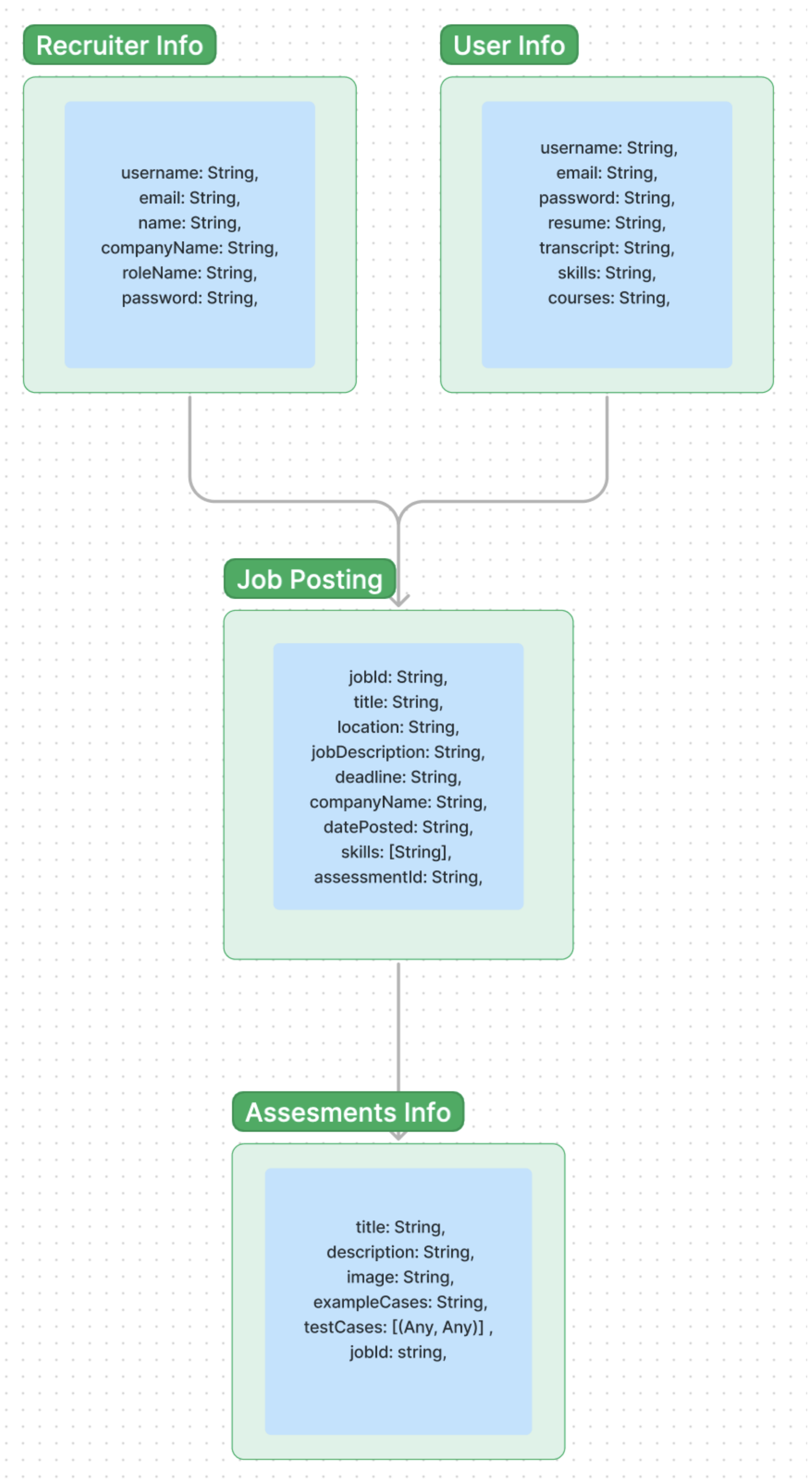
Three-Tier Architecture

This project utilized the Three-Tier Architecture (MERN Stack). The user interacts with React (the client) frontend of the web application. Any user requests will enable React to send a HTTP request to the application server which is written using NodeJS and ExpressJS. The server then serves the client by accessing the MongoDB database.



System Decomposition

The database, the front end, and the back end are the three distinct components of the system architecture. The users will be engaging with the frontend. React and its components are used in the front-end development. For universal styling and the development of components with smoother operation, we also employed Material UI. Using the fetch call, we can also access the API endpoints that the backend offers. In this manner, we can enable the introduction of CRUD properties in our application. The app's backend was developed using Node.js and Express.js. Express is a well-liked node framework with several advantages. Writing handlers for HTTP requests at various URL paths (routes) is one of the ones we used. This is how we configure the API endpoints that communicate with our MongoDB database. We have created distinct collections in our database, including job postings, assessments, and user profiles. To prevent receiving invalid inputs, we have placed input sanitation code in place.


Recruiter Info
username: String,
email: String,
name: String,
companyName: String,
roleName: String,
password: String,
User Info
username: String,
email: String,
password: String,
resume: String,
transcript: String,
skills: String,
courses: String,
Job Posting
jobId: String,
title: String,
location: String,
jobDescription: String,
deadline: String,
companyName: String,
datePosted: String,
skills: [String],
assessmentId: String,
Assesments Info
title: String,
description: String,
image: String,
exampleCases: String,
testCases: [(Any, Any)] ,
jobId: string,