horizontal line

GoBeyondClassroom:

**15th April 2024**

# Introduction

* **A. Purpose**
  + This document provides a comprehensive technical overview of the GoBeyondClassroom web application.
  + It details the architecture, functionalities, code structure, and deployment process.
* **B. Target Audience**
  + Developers with a basic understanding of Java, Spring Boot, Angular, PostgreSQL, Docker, and Docker Compose.
* **C. Project Significance**
  + GoBeyondClassroom facilitates collaboration between students, faculty, and clients for project-based learning.
  + It manages user registration, project proposals, applications, approvals, and communication.

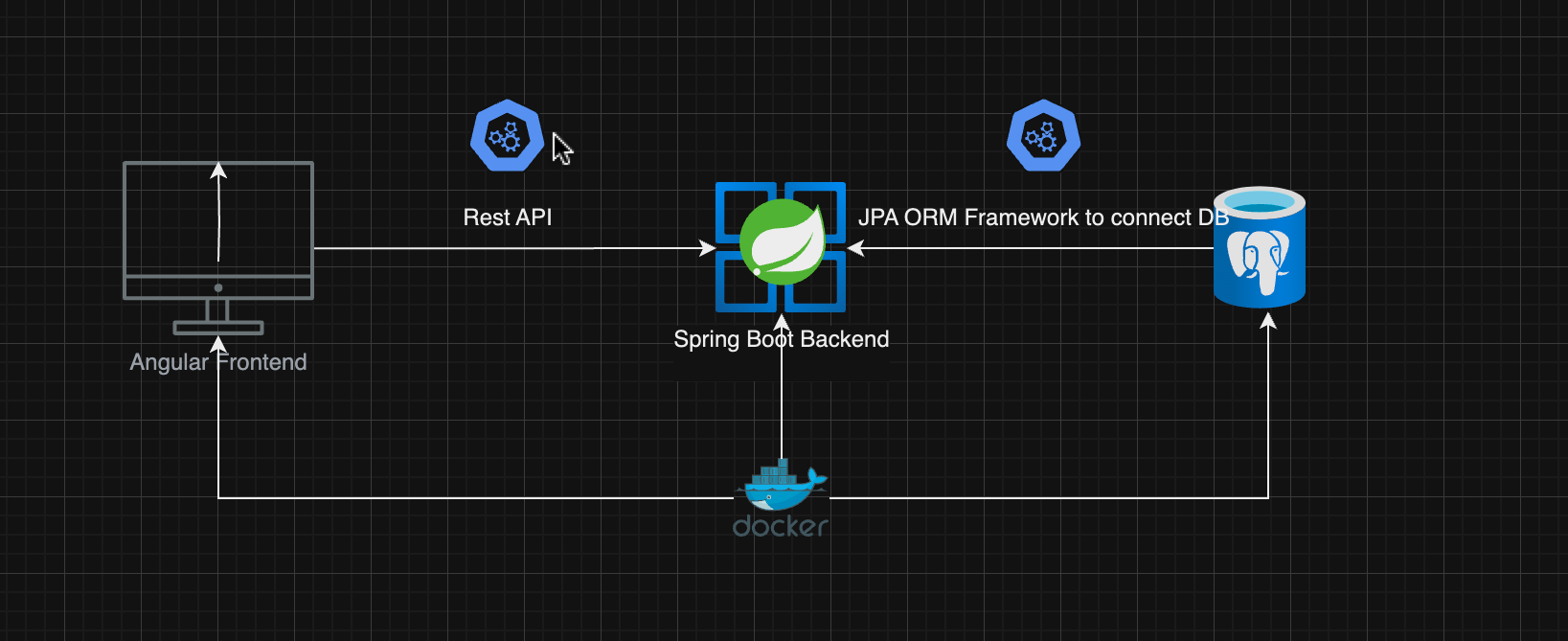
# Technology Stack Justification

* **A. Backend (Java Spring Boot):**
  + **Reason:** Spring Boot offers a robust and efficient framework for building RESTful APIs.
  + **Benefits:**
    - Rapid application development
    - Easy integration with databases and other services
    - Spring Security for user authentication and authorization
* **B. Frontend (Angular):**
  + **Reason:** Angular provides a powerful and scalable framework for building dynamic single-page applications (SPAs).
  + **Benefits:**
    - Improved user experience with rich client-side interactions
    - Modular architecture for maintainability
    - Angular CLI simplifies development and deployment
* **C. Database (PostgreSQL):**
  + **Reason:** PostgreSQL is a powerful and open-source relational database management system (RDBMS).
  + **Benefits:**
    - Scalability to handle growing user base and project data
    - Robust data integrity and security features
    - Cost-effective and well-supported
* **D. Containerization (Docker) and Orchestration (Docker Compose):**
  + **Reason:** Docker and Docker Compose enable a lightweight and portable deployment approach.
  + **Benefits:**
    - Consistent application environment across different machines
    - Simplified deployment and scaling of individual services
    - Efficient resource management

# System Requirements and Setup

* **A. System Requirements**
  + Operating System: Windows 10 (or later), macOS, or Linux
  + Hardware: Minimum 8GB RAM, recommended 16GB or more
  + Software: Java Development Kit (JDK) 11 or later, Node.js and npm (Node Package Manager), Angular CLI, Docker Desktop and PostgreSQL JDBC driver
* **B. Installation and Configuration**
  + Install Docker Engine and Docker Compose.
  + Clone the project repository from GitHub.
  + Navigate to the project directory.
  + Run docker-compose up command to build and start the containers.
  + Access the application via the specified port in the browser.

# Application Architecture



## Explanation:

The GoBeyondClassroom application follows a microservices architecture with containerization using Docker and orchestration by Docker Compose. Here's a breakdown of the components and their interactions:

* **Client:** Users representing departments or clients within ETSU access the application through a web browser.
* **Angular Frontend:** This single-page application (SPA) built with Angular handles user interactions, displays information, and sends requests to the backend API.
* **Spring Boot Backend:** This Java-based service layer exposes RESTful APIs for user management, project management, and communication functionalities. It interacts with the database and sends responses back to the frontend.
* **PostgreSQL Database:** This relational database stores all application data, including user details, project information, and communication records.
* **Docker Containers:** Each service (frontend, backend, database) runs within its own Docker container, providing a lightweight and isolated environment.
* **Docker Compose:** This tool manages the lifecycle of the Docker containers, defining dependencies and automating service startup and shutdown.

**Benefits of this Architecture:**

* **Scalability:** Individual services can be scaled independently by adding more containers to handle increased load.
* **Maintainability:** Modular architecture with clear separation of concerns simplifies development, testing, and deployment.
* **Portability:** Docker containers ensure consistent application behavior across different environments.
* **Resilience:** If one service fails, it won't affect the others due to container isolation.

**Communication Flow:**

1. **User Interaction:** Users interact with the Angular frontend through the web browser.
2. **API Requests:** The frontend sends HTTP requests to the Spring Boot backend APIs.
3. **Backend Processing:** The backend API processes the request, interacts with the PostgreSQL database as needed, and generates a response.
4. **API Response:** The backend API sends the response back to the Angular frontend.
5. **Frontend Update:** The frontend updates the user interface based on the received response.

# Functionalities and Usage

* **A. Main Features**
  + User registration for students, faculty, and clients
  + Project proposal creation by clients
  + Project browsing and filtering based on skills and interests
  + Project application by faculty and students
  + Project approval by clients
* **B. Basic Usage Examples**

Here's a breakdown of the user journey for each role (student, faculty, client) interacting with the GoBeyondClassroom application, along with concrete examples of how they utilize features for project collaboration:

**1. Student Journey:**

* **Sign Up (similar to student):** A new student visits the application and clicks "Sign Up." They enter their details (name, email, password, role as "Student," and skills) and submit the registration.
* **Approval:** The student receives a message that their request has been sent for approval. An admin reviews the request in the dashboard.
* **Login:** Once approved, the student logs in with their credentials and lands on the home page.
* **Project Browsing:** The student sees a list of available projects. Each project card displays details like title, description, skills required, and application status.
* **Project Application:** The student finds a suitable project and clicks "Apply." They can optionally add notes highlighting their relevant skills and interest in the project.
* **Collaboration:** If the client selects the student for the project, they can collaborate through the application's communication features (if implemented) to discuss tasks and progress.
* **My Projects Tab:** The student can access a dedicated "My Projects" tab to view all the projects they have applied to, including their application status (pending, approved, rejected).

**2. Faculty Journey:**

* **Sign Up (similar to student):** A faculty member follows the same steps as a student for registration, selecting their role as "Faculty" and specifying their expertise as skills.
* **Approval:** The admin reviews and approves the faculty member's request.
* **Login:** The faculty member logs in and views the available projects.
* **Project Selection:** They can browse projects based on their expertise and the skills required.
* **Project Application:** The faculty member identifies a project they're interested in and applies. They might include a brief statement outlining their relevant experience and how they can contribute.
* **Collaboration:** If selected for the project, the faculty member collaborates with the student(s) and client to guide and mentor them during project execution.
* **My Projects Tab:** Similar to students, faculty can access a "My Projects" tab to see all projects they have applied to, along with their application status.

**3. Client Journey:**

* **Sign Up (similar to student):** A client representing a department or company registers with the application, selecting their role as "Client."
* **Approval:** The admin reviews and approves the client's request.
* **Login:** The client logs in and accesses the project management features.
* **Project Creation:** The client creates a new project by providing details like the project title, description, required skills, expected timeline, and any relevant documents.
* **Project Management:** The client views the list of submitted applications for their projects.
* **Approval of Applicants:** The client reviews applications from students and faculty, selecting suitable candidates based on their skills and experience. This includes the ability to approve or deny applications.
* **Collaboration:** The client interacts with the selected students and faculty member(s) to discuss project details, provide feedback, and monitor progress.
* **My Projects Tab:** Clients have a dedicated "My Projects" tab that displays a list of all projects they created. This view allows them to manage their projects, including:
  + Viewing project details and applicant information.
  + Approving/denying student and faculty applications.
  + Deleting a project once it's completed or no longer relevant.

These enhancements ensure a more complete and accurate representation of the user experience within GoBeyondClassroom. Each role has functionalities tailored to their needs, enabling efficient project collaboration.

# Code Structure:

**A. Directory Structure**

1. **backend (Spring Boot project):**
   * Contains the core Java code for the application logic, including:
     + **src/main/java:** Packages containing Spring Boot application classes, controllers, services, repositories, and models representing your data entities (users, projects, applications, etc.).
     + **src/main/resources:** Configuration files (application.properties), database schema definition files, and static resources.
     + **pom.xml:** The project object model file defining project dependencies and build configuration.
2. **frontend (Angular project):**
   * Contains the Angular application code for the user interface:
     + **src/app:** The root directory for Angular components, services, modules, and routing configurations.
     + **assets:** Static assets like images, fonts, or stylesheets used in the application.
     + **environments:** Environment-specific configuration files (e.g., development, production).
     + **angular.json:** The Angular project configuration file.
3. **docker (Docker configuration files):**
   * Contains files for defining Docker images for each service:
     + **Dockerfile:** Specifies instructions for building Docker images for the backend, frontend, and potentially the database server.
     + **docker-compose.yml:** This file defines the services, their configurations, dependencies, and how they are orchestrated together. You can find this file under the root path of backend service.

**B. Component Descriptions**

**Backend (Spring Boot):**

* **Spring Boot Application Class:** The main entry point for the application, responsible for initializing Spring components and starting the server.
* **Controllers:** Handle incoming API requests from the frontend, interact with services to process data, and prepare responses.
* **Services:** Encapsulate business logic, interact with repositories to access and manipulate data, and return processed information to controllers.
* **Repositories:** Provide an abstraction layer for interacting with the database, performing CRUD (Create, Read, Update, Delete) operations on data entities.
* **Models:** Represent data entities (users, projects, applications) with their attributes and relationships.

**Frontend (Angular):**

* **Components:** Reusable UI building blocks that define the visual elements and functionalities of the application pages.
* **Services:** Provide functionalities for data fetching, manipulation, and communication with the backend APIs.
* **Modules:** Organize components, services, and other building blocks into logical units based on features or functionalities.
* **Routing:** Defines how the application navigates between different views (pages) based on user actions or URL changes.

**C. Code Interdependencies**

1. **Frontend-Backend Interaction:**
   * The Angular frontend makes HTTP requests to Spring Boot backend API endpoints defined in controllers.
   * Requests typically include data in the request body (e.g., user login credentials, project details) or query parameters (e.g., filtering projects by skills).
   * Backend services process the request data, interact with repositories to retrieve or manipulate data, and generate a response.
   * The response data (e.g., list of projects, user information) is sent back to the frontend in JSON format.
   * The Angular frontend components receive the response data, update the user interface accordingly, and potentially make further API calls based on user interactions.
2. **Database Interaction:**
   * Spring Boot repositories use JPA (Java Persistence API) to interact with the PostgreSQL database.
   * Repositories provide methods for performing CRUD operations on data entities mapped to database tables.
   * When the backend services need to access or modify data, they call repository methods, which translate these calls into SQL queries for the database.
   * The frontend doesn't directly interact with the database. All data access and manipulation are handled by the backend services.

**D. Code Deep Dive**

Backend:

**1. GoBeyondClassroomApplication.java**

* **Package:** com.etsu.gobeyondclassroom - This is the base package for your application code.
* **@SpringBootApplication Annotation:** This annotation marks this class as the main Spring Boot application class. It automatically enables features like component scanning, auto-configuration, and property binding. This essentially bootstraps the Spring application and serves as the entry point.
* **public class GoBeyondClassroomApplication:** This line defines the main class for the application.
* **public static void main(String[] args):** This is the main method where the application execution starts. Inside this method:
  + SpringApplication.run(GoBeyondClassroomApplication.class, args); - This line starts the Spring Boot application. It takes two arguments:
    - The current class (GoBeyondClassroomApplication.class) - This tells Spring Boot which class contains the main entry point.
    - The command-line arguments (args) - These arguments can be used to configure the application at runtime (not used in this example).

In summary, this class is the starting point for your Spring Boot application. It leverages Spring Boot's functionalities to launch the application.

**2. CorsConfig.java**

* **Package:** com.etsu.gobeyondclassroom.config - This package stores configuration classes for your application.
* **@Configuration Annotation:** This annotation marks the class as a Spring configuration class. Spring will scan this package for classes with this annotation and use them to configure the application context.
* **public class CorsConfig:** This defines the configuration class for CORS (Cross-Origin Resource Sharing).
* **@Bean Annotation:** This annotation is used on methods within a configuration class to define a bean. A bean is an object managed by the Spring container.

Within the CorsConfig class:

* **CorsFilter corsFilter():** This is a bean method that creates and returns a CorsFilter object. This filter handles CORS requests (requests from a different domain than the one serving the application).
  + UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource(); - This line creates a new UrlBasedCorsConfigurationSource object. This object is used to map CORS configurations to specific URLs.
  + CorsConfiguration config = new CorsConfiguration(); - This line creates a new CorsConfiguration object. This object defines the allowed origins, methods, headers, etc. for CORS requests.
  + config.addAllowedOrigin("\*"); - This line allows all origins to make CORS requests to your application. In a production environment, you'll typically want to restrict allowed origins for security reasons. (Currently allowing all origins).
  + config.addAllowedMethod("\*"); - This line allows all HTTP methods (GET, POST, PUT, DELETE, etc.) for CORS requests.
  + config.addAllowedHeader("\*"); - This line allows all headers in the request.
  + source.registerCorsConfiguration("/\*\*", config); - This line registers the CorsConfiguration object for all URLs (/\*\*) within the application.
  + return new CorsFilter(source); - This line creates a new CorsFilter object with the configured UrlBasedCorsConfigurationSource and returns it. This filter will be applied to all incoming requests and handle CORS preflight checks and actual requests based on the configuration.

Overall, this class configures CORS for your application. Currently, it allows CORS requests from any origin, method, and header. This is a common configuration for development, but **in a production environment, you'll want to restrict CORS settings for enhanced security.**

**3. SecurityConfig.java**

* **Package:** com.etsu.gobeyondclassroom.config - As mentioned before, this package stores configuration classes.
* **@Configuration Annotation:** Same functionality as in CorsConfig.java.
* **@EnableWebSecurity Annotation:** This annotation enables Spring Security in your application. Spring Security is a framework that provides authentication, authorization, and other security features.
* **public class SecurityConfig:** This defines the configuration class for Spring Security.

Within the SecurityConfig class:

* **@Bean** annotation is used on two methods in this class:
  + SecurityFilterChain filterChain(HttpSecurity http) throws Exception { ... }: This method defines the security configuration for your application. It uses the HttpSecurity object to configure various aspects of security, such as authorization rules and CSRF protection.
    - http.authorizeHttpRequests(requests -> requests.anyRequest().permitAll()) - This line allows any request to access any resource in your application. This essentially disables Spring Security's

**1. ProjectApplicationController.java**

* **Package:** com.etsu.gobeyondclassroom.controller - This package stores controllers, which handle incoming API requests and interactions with services.
* **@RestController Annotation:** This annotation marks this class as a RESTful web service controller.
* **@RequestMapping("/projectApplications")**: This annotation maps all the request paths handled by this controller to start with "/projectApplications".

This controller class defines methods for CRUD (Create, Read, Update, Delete) operations on project applications:

* **createProjectApplication(ProjectApplication projectApplication):** This method creates a new project application by calling the ProjectApplicationService to save the provided projectApplication object.
* **getAllProjectApplications():** This method retrieves all project applications from the database using the ProjectApplicationService.
* **getProjectApplicationById(Long id):** This method retrieves a specific project application by its ID using the ProjectApplicationService.
* **updateProjectApplication(Long id, ProjectApplication projectApplicationDetails):** This method updates a project application by ID. It first fetches the existing application and then updates its attributes based on the provided projectApplicationDetails object. Finally, it calls the ProjectApplicationService to save the updated application.
* **deleteProjectApplication(Long id):** This method deletes a project application by ID using the ProjectApplicationService.

**2. ProjectApplication.java**

* **Package:** com.etsu.gobeyondclassroom.model - This package stores model classes that represent data entities in your application.
* **@Entity Annotation:** This annotation marks this class as a JPA (Java Persistence API) entity, indicating that it maps to a database table.
* **@Data Annotation (from Lombok):** This annotation (from a Lombok library) generates boilerplate code for getters, setters, toString, equals, and hashCode methods.
* **@NoArgsConstructor Annotation (from Lombok):** This annotation generates a no-argument constructor for the class.
* **@AllArgsConstructor Annotation (from Lombok):** This annotation generates a constructor with arguments for all fields of the class.

This class defines the model for a project application. It has the following attributes:

* **id (Long):** Unique identifier for the project application (auto-generated).
* **userId (Long):** ID of the user who submitted the application.
* **email (String):** Email address of the user (optional).
* **projectId (Long):** ID of the project the application is for.
* **status (String):** Status of the application (e.g., "pending", "approved", "denied").
* **comments (String):** Any comments or notes associated with the application.

**3. ProjectApplicationService.java**

* **Package:** com.etsu.gobeyondclassroom.service - This package stores service classes that handle business logic and interact with repositories.
* **@Service Annotation:** This annotation marks this class as a Spring service.

This interface defines the functionalities for managing project applications. It has methods for CRUD operations:

* createProjectApplication(ProjectApplication projectApplication) (implemented in ProjectApplicationServiceImpl.java)
* getAllProjectApplications() (implemented in ProjectApplicationServiceImpl.java)
* getProjectApplicationById(Long id) (implemented in ProjectApplicationServiceImpl.java)
* updateProjectApplication(Long id, ProjectApplication projectApplicationDetails) (implemented in ProjectApplicationServiceImpl.java)
* deleteProjectApplication(Long id) (implemented in ProjectApplicationServiceImpl.java)

**4. ProjectApplicationServiceImpl.java**

* **Package:** com.etsu.gobeyondclassroom.service - Same as ProjectApplicationService.java.
* **@Service Annotation:** Same as ProjectApplicationService.java.

This class implements the ProjectApplicationService interface. It provides the actual logic for CRUD operations on project applications:

* **createProjectApplication(ProjectApplication projectApplication):** This method delegates the task of saving the project application to the projectApplicationRepository.
* **getAllProjectApplications():** This method retrieves all project applications from the projectApplicationRepository.
* **getProjectApplicationById(Long id):** This method retrieves a project application by ID from the projectApplicationRepository.
* **updateProjectApplication(Long id, ProjectApplication projectApplicationDetails):** This method first fetches the existing project application by ID. Then, it updates its attributes based on the provided projectApplicationDetails object. Finally, it saves the updated application using the projectApplicationRepository.
* **deleteProjectApplication(Long id):** This method deletes a project application by ID

The concepts demonstrated here for CRUD operations on Technology and User entities apply similarly to other entities in the system. While there might be minor variations, the core logic and annotations used for data access and manipulation remain consistent across these entities.

Frontend:

**1. app.routing.module.ts:**

This file defines the routing configuration for the application. It imports the RouterModule from @angular/router and uses it to create an array of Routes representing the different paths in the application.

* Each Route object specifies a path, a component to load for that path, and optionally an AuthGuard to restrict access to certain routes.
* The redirectTo and \*\* (wildcard) routes handle default and unmatched routes.

**2. auth.guard.ts:**

This file implements the AuthGuard, which is a service used to restrict access to certain routes based on authentication status.

* It injects the AuthService and Router services.
* The canActivate method checks if the user is authenticated using the AuthService.
* If not authenticated, it redirects the user to the login page.

**3. login.component.ts:**

This file defines the login component (app-login).

* It imports necessary modules like MatSnackBar for material design notifications and FormsModule or ReactiveFormsModule for form handling (based on your approach).
* It injects the AuthService, Router, MatSnackBar, and optionally UserService services.
* It defines a loginForm using either template-driven or reactive forms approach.
* It handles form submission logic, calling the AuthService to login and potentially retrieving user data using the UserService.
* It uses MatSnackBar to display success or error messages.

**4. login.component.html:**

This file defines the login component's template (app-login).

* It uses Angular Material components like mat-form-field, mat-label, mat-input, etc. to create the login form.
* It binds the form to the loginForm defined in the component class.
* It includes buttons for login and potentially a link to signup.

**5. signup.component.ts:**

This file defines the signup component (app-signup).

* It imports necessary modules similar to login.component.ts.
* It injects similar services like AuthService, Router, MatSnackBar, and optionally HttpClient for making API calls.
* It defines a signupForm using either template-driven or reactive forms approach.
* It implements form validation using custom validators for email and password patterns, and a password match validator.
* It handles form submission logic, potentially making an HTTP POST request to a backend API for user registration using HttpClient.
* It displays success or error messages using MatSnackBar.

**6. signup.component.html:**

This file defines the signup component's template (app-signup).

* It uses Angular Material components similar to login.component.html to create the signup form.
* It binds the form to the signupForm defined in the component class.
* It includes fields for user details, skills selection, and buttons for signup and potentially a link to login.

**Deep Dive into app-routing.module.ts and Authentication**

**Route Mappings and Components:**

The app-routing.module.ts file defines how URLs in your application map to specific components. Here's a breakdown of the provided routes:

* /login: This route maps to the LoginComponent. This component likely handles the login form and logic.
* /signup: This route maps to the SignupComponent. This component likely handles the user registration form and logic.
* /home: This route maps to the ProjectListComponent. This component might display a list of projects. However, it requires the AuthGuard for access, indicating it's a protected route.
* /approval-dashboard: This route maps to the ApprovalDashboardComponent. Similar to /home, it requires AuthGuard for access.
* /projects: This route likely also maps to the ProjectListComponent for displaying projects, requiring AuthGuard.
* /projects/new: This route maps to a component (potentially ProjectFormComponent) for creating new projects, requiring AuthGuard.
* { path: "", redirectTo: "/home", pathMatch: "full" }: This is a default route. If a user enters a URL that doesn't match any defined route, they will be redirected to /home.
* { path: "\*\*", redirectTo: "/home" }: This is a wildcard route. It catches any URL that doesn't match a specific route and redirects the user to /home.

**Components and View/API Interaction:**

* **Components:** Components are the building blocks of an Angular application. They define a specific view with HTML templates and handle user interactions using TypeScript code. Each component manages its own data and logic.
* **View:** The component's HTML template defines the visual representation of the component. It uses Angular directives and data binding to display information and interact with the user.
* **APIs:** Components can call backend APIs using the HttpClient service provided by Angular. This allows components to fetch data, perform actions, and interact with a server.

**Auth Guard Implementation:**

The AuthGuard service restricts access to certain routes based on the user's authentication status. Here's how it works:

1. The AuthGuard injects the AuthService (likely responsible for authentication logic) and the Router (for navigation).
2. The canActivate method is called whenever a user tries to access a route protected by the AuthGuard.
3. Inside canActivate, the service calls the AuthService to check if the user is authenticated.
   * If authenticated, the method returns true, allowing access to the route.
   * If not authenticated, the method uses the Router to redirect the user to the login page (/login).

**First-time User and Signup:**

When a user first visits the application, they will land on the default route (/home). However, since it's protected by the AuthGuard, they will be redirected to the login page (/login) if not authenticated.

This ensures unauthorized users cannot access secure areas of the application. They must complete the login process or signup (if available) to gain access.

## Deep Dive into approval-dashboard.component.ts

This component (ApprovalDashboardComponent) handles the functionality for managing user approvals within the application.

**Functionality Breakdown:**

1. **Imports:**
   * Component and OnInit from @angular/core: Decorators for Angular components and lifecycle hooks.
   * UserService from ../user.service: Service likely responsible for user data access and manipulation.
   * User from ../models/user: Interface or class defining the user data structure (ID, username, email, etc.).
   * MatTableDataSource from @angular/material/table: Material Design component for displaying tabular data.
2. **Component Definition:**
   * Decorated with @Component: Defines the component's metadata like selector, template URL, and stylesheet.
   * selector: app-approval-dashboard - Selector used in HTML templates to reference this component.
   * templateUrl: Path to the component's HTML template.
   * styleUrls: Paths to component-specific CSS styles.
3. **Properties:**
   * usersWithPendingStatus: Array of User objects. Stores users with a pending approval status.
   * dataSource: Instance of MatTableDataSource<User>. Used to bind user data to the Material table.
   * displayedColumns: Array of strings. Defines column names for the user table.
4. **Constructor:**
   * Injects the UserService to interact with user data.
5. **ngOnInit Lifecycle Hook:**
   * Called once after the component is initialized.
   * Subscribes to the userService.getUsersWithPendingStatus() method.
   * Fetches users with pending status and assigns them to usersWithPendingStatus.
   * Creates a new MatTableDataSource instance using usersWithPendingStatus.
6. **approveUser Method:**
   * Takes a userId as input (number).
   * Calls userService.updateUserStatus(userId, 'approved') to update the user's status to "approved".
   * Subscribes to the update observable and refreshes the user list upon successful update using refreshUsersWithPendingStatus.
7. **denyUser Method:**
   * Similar to approveUser, but updates the user's status to "denied".
8. **refreshUsersWithPendingStatus Method:**
   * Calls userService.getUsersWithPendingStatus() to fetch the latest list of users with pending status.
   * Updates usersWithPendingStatus and the dataSource with the new data.

**Component Template (approval-dashboard.component.html):**

* Uses Angular Material table component (mat-table) to display user data.
* Defines columns using matColumnDef directives with properties for header and cell content.
* The dataSource property is bound to the MatTableDataSource instance.
* The displayedColumns property defines the order and visibility of columns.
* Action buttons for approving and denying users are included in the "actions" column.

**Overall, this component demonstrates how to:**

* Fetch data from a service (UserService).
* Display data in a Material Design table.
* Perform actions on data (updating user status) and refresh the view accordingly.

**Deep Dive into Project Components: project-form.component.ts and project-list.component.ts**

These components manage functionalities related to projects: creating, editing, listing, and potentially applying (based on user role).

**project-form.component.ts:**

1. **Imports:**
   * Necessary modules from @angular/core, @angular/forms, @angular/router, and project related services.
2. **Component Definition:**
   * Handles project form creation and submission.
3. **Properties:**
   * projectForm: FormGroup for managing form data.
   * isEditing: Boolean flag indicating edit mode.
   * projectId: Number representing the project ID (if editing).
   * technologyOptions: Array of Technology objects for the technology selection dropdown.
4. **Constructor:**
   * Injects required services for project and technology data access.
5. **ngOnInit Lifecycle Hook:**
   * Fetches technology options using projectService.getAllTechnologies().
   * Initializes the project form using initForm().
   * Subscribes to route params to handle editing an existing project:
     + If an "id" parameter exists, sets isEditing to true, retrieves the project data using loadProject(), and pre-populates the form.
6. **initForm Method:**
   * Creates a FormGroup with form controls for project details and an empty array for technologies (multi-select).
7. **loadProject Method:**
   * Takes a project ID as input.
   * Retrieves the project details using projectService.getProjectById(id).
   * Populates the form with retrieved project data (implementation details omitted).
8. **onSubmit Method:**
   * Checks if the form is valid.
   * If valid, processes the form data based on editing or creation:
     + **Editing:**
       - Updates the existing project using projectService.updateProject(projectId, formData).
       - Redirects to the projects list on success.
     + **Creation:**
       - Creates a new Project object with form data and sets default values like owner and status.
       - Calls projectService.createProject(project) to create the project.
       - Redirects to the projects list on success.
   * If the form is invalid, logs the form data for debugging and potentially displays validation errors (not implemented here).
9. **loadTechnologies Method:**
   * Fetches all technologies using projectService.getAllTechnologies().
   * Subscribes to the observable and populates the technologyOptions array.

**project-list.component.ts:**

1. **Imports:**
   * Necessary modules from @angular/core, project related services, MatDialog, and components for project card and dialogs.
2. **Component Definition:**
   * Handles displaying a list of projects, user role checks, and opening dialogs for applying and deleting projects.
3. **Properties:**
   * projects: Array of Project objects representing the project list.
   * yourProjects: Array to potentially store user's projects (not implemented in the provided code).
   * canCreateProject: Boolean flag indicating if the user can create projects based on role.
   * storedUserObject: User object retrieved from local storage (if user is not logged in).
4. **Constructor:**
   * Injects project, user, dialog, and project application services.
5. **ngOnInit Lifecycle Hook:**
   * Loads projects using loadProjects().
   * Calls setLoggedInUser() to check user login status and determine project creation permission.
6. **setLoggedInUser Method:**
   * Checks if a user is logged in (userService.loggedInUser).
   * If not logged in, retrieves the user object from local storage and parses it back into a User object.
   * Sets userService.loggedInUser with the retrieved user.
   * Calls userRoleCheck to determine project creation permission based on user role.
7. **userRoleCheck Method:**
   * Checks the user role and sets canCreateProject accordingly (client or admin can create projects).
8. **loadProjects Method:**
   * Fetches all projects using projectService.getAllProjects().
   * Subscribes to the observable and populates the projects array.
9. **openApplyDialog Method:**
   * Opens the ApplyDialogComponent using MatDialog for applying to a project.
   * Subscribes to the submitData observable from the dialog component to capture application comments.
   * Creates a ProjectApplication object with user ID, email, project ID, comments, and a pending status.
   * Calls `projectApplicationService.create

**AuthService:**

* Handles user login and logout functionality.
* Maintains the login state and user email using BehaviorSubject observables.
* Checks for existing login information in localStorage on initialization.
* Provides methods for login, logout, checking authentication state, and retrieving user email.

**UserService:**

* Interacts with the backend API (http://localhost:8080/api/users) for user data access.
* Provides methods to:
  + Get all users.
  + Get a user by email address.
  + Update a user's status.
  + Get users with pending status (potentially for admin purposes).

**Calling Backend APIs with Services:**

Services are the preferred way to interact with backend APIs in Angular applications. They encapsulate the data access logic and provide a clean interface for components to consume. Here's how it works:

* **Service Definition:**
  + Create a service class using @Injectable decorator.
  + Inject HttpClient from @angular/common/http for making HTTP requests.
  + Define methods for API calls (e.g., getAllProjects(), getUserByEmail(email: string)) that return Observables representing the asynchronous data streams.
* **Making API Calls:**
  + In your component, inject the service using the constructor.
  + Call the service methods within your component logic (e.g., ngOnInit).
  + Subscribe to the Observables returned by the service methods using subscribe().
  + The subscribe callback receives the response data or error information from the API.
* **Example (ProjectService):**
* TypeScript

@Injectable({ providedIn: 'root' })

export class ProjectService {

private apiUrl = "http://localhost:8080/api/projects";

constructor(private http: HttpClient) {}

getAllProjects(): Observable<Project[]> {

return this.http.get<Project[]>(this.apiUrl);

}

// Other API methods...

}

// In a component (e.g., project-list.component.ts)

constructor(private projectService: ProjectService) {}

ngOnInit() {

this.projectService.getAllProjects().subscribe(

(projects) => {

this.projects = projects; // Update component data

},

(error) => {

console.error("Error loading projects:", error);

}

);

}

1. **Component Communication with Services:**

Services can also act as intermediaries for communication between components that don't have a direct parent-child relationship. Here's a common pattern:

* **Data Sharing Service:**
  + Create a service to hold the shared data (e.g., selectedProject, userData).
  + Use BehaviorSubject or Subject from RxJS to manage the data updates.
* **Component Access:**
  + Inject the service into both components that need to communicate.
  + In one component, update the service's data using the BehaviorSubject/Subject methods (e.g., next(), emit()).
  + In the other component, subscribe to the service's observable to listen for data changes and update the component UI accordingly.
* **Example (Project Selection and Details):**
* TypeScript

// project-selection.component.ts (selecting a project)

constructor(private projectSelectionService: ProjectSelectionService) {}

onProjectSelected(projectId: number) {

this.projectSelectionService.setSelectedProject(projectId);

}

// project-details.component.ts (displaying details)

constructor(private projectSelectionService: ProjectSelectionService) {}

ngOnInit() {

this.projectSelectionService.selectedProject$.subscribe((projectId) => {

if (projectId) {

// Load project details using projectId

}

});

}

By using services for both API calls and component communication, you promote code reusability, maintainability, and separation of concerns in your Angular application.

# Future Enhancements and Limitations

**A. Limitations:**

* **Limited User Management:** The provided code doesn't show extensive user management functionalities. It might lack features like detailed user profiles, or role-based access control.
* **Data Validation:** The current implementation might not have comprehensive data validation on both client and server sides. This could lead to inconsistencies or errors in project data.
* **Security Considerations:** Security best practices like proper authentication, authorization, and input sanitization might not be fully implemented, leaving the application vulnerable to potential security risks.

**B. Future Enhancements:**

* **Advanced Search:** Implement search functionalities to filter projects based on title, technology, owner, or other relevant criteria.
* **Real-time Collaboration:** Integrate tools for real-time collaboration on projects, such as shared editing documents or communication channels.
* **Project Updates & Notifications:** Allow project owners to update project details and send notifications to team members.
* **User Management:** Implement comprehensive user registration, profile management, and role-based access control for different user types (clients, developers, admins).
* **Integration with External Services:** Integrate with third-party services like project management tools, Git repositories, or cloud storage for enhanced functionalities.

## Conclusion

**GoBeyondClassroom** has the potential to be a valuable platform for project-based learning and collaboration. It can provide students with a structured environment to manage projects, collaborate with peers, and showcase their work. By implementing the suggested enhancements, GoBeyondClassroom can become a more robust and feature-rich tool for educational purposes.

# Flow Diagram:

