This is an exciting project! You're combining powerful, modern technologies for your backend. Here's a comprehensive step-by-step guide to implement your backend using Java (Spring Boot), Ballerina, Clerk for authentication, and MongoDB as the database, integrated with your existing frontend.

Backend Implementation: Step-by-Step Guide

High-Level Architecture

Your application will follow a microservices-oriented approach:

* Frontend (HTML/CSS/JS): Your existing web pages.
* Clerk: Handles all user authentication (login, signup, user profiles, sessions).
* Java Spring Boot Backend: Your main API service for managing project data (CRUD operations, search, trending, stats). It will also communicate with Clerk for user identification.
* Ballerina Service (Optional but Recommended for Webhooks): A lightweight service to handle webhooks from Clerk (e.g., when a new user signs up) and sync essential user data to your MongoDB.
* MongoDB: Your primary database for storing project details and potentially expanded user profiles.
* Cloud Storage (e.g., AWS S3): For storing project media files (images, videos).

Generated code

+------------+ +--------------------+ +---------------------+

| Frontend | <----->| Java Spring Boot | <----->| MongoDB |

| (HTML/JS) | | (Project APIs) | | (Projects, Users) |

+------------+ +--------------------+ +---------------------+

^ ^ ^ ^

| | (Auth UI/SDK) | | (API Calls/JWT Validation)

| +--------------------+ |

| | (Webhook for User Sync)

| |

+------------------------------>|

+---------------------+ +--------------------+

| Clerk | <----->| Ballerina (Webhooks)|

| (User Auth, Profiles)| | |

+---------------------+ +--------------------+

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).

Phase 1: Setup & Core User Management

This phase focuses on getting your authentication with Clerk set up and connecting your database.

Step 1.1: Clerk Setup (Authentication & User Management)

Clerk simplifies user management significantly. Instead of building your own login/signup forms and logic, you'll typically use Clerk's pre-built UI components or SDKs.

1. Create a Clerk Account: Go to [clerk.com](https://www.google.com/url?sa=E&q=https%3A%2F%2Fclerk.com%2F) and sign up.
2. Create an Application: In your Clerk dashboard, create a new application (e.g., "CODENET").
3. Get API Keys:
   * You'll find your Publishable Key (e.g., pk\_live\_...) for your frontend.
   * You'll find your Secret Key (e.g., sk\_live\_...) for your backend (Spring Boot and Ballerina) to interact with Clerk's APIs securely.
4. Frontend Integration Strategy (Crucial for your existing HTML):  
   Given your existing HTML files (login.html, signup.html, userprofile.html), you have two main options for integrating Clerk:
   * Option A (Recommended for simplicity with Clerk): Use Clerk's Hosted Pages or Components.
     + For login.html and signup.html, you would redirect users to Clerk's hosted login/signup pages, or embed Clerk's pre-built React/Vue/Vanilla JS components into an empty div on those pages. Clerk handles the forms, validation, and session management.
     + For userprofile.html, you'd use Clerk's UserProfile component or fetch user data via the Clerk JS SDK and populate your existing HTML.
     + Benefit: Easiest and most secure way to integrate Clerk.
   * Option B (More Complex, but keeps your existing form structure): Custom UI with Clerk Backend API.
     + Your existing login.html and signup.html forms would submit their data (username/password) to your Spring Boot backend.
     + Your Spring Boot backend would then call Clerk's backend APIs (e.g., sign-in, sign-up) using your Secret Key.
     + This requires careful handling of credentials, sessions, and error messages on your backend.
     + Recommendation: For a seamless experience and leveraging Clerk's full power, Option A is highly recommended. This guide will assume you move towards Option A for authentication, but your Spring Boot backend will still interact with Clerk for authorization checks.
5. Configure Webhooks in Clerk:
   * In your Clerk dashboard, navigate to Webhooks.
   * Add an endpoint for events like user.created, user.updated, user.deleted. These will be crucial if you want to store a subset of user data (like their Clerk ID) in your MongoDB.
   * Set the target URL to an endpoint you'll expose from your Ballerina service (e.g., https://your-domain.com/ballerina-clerk-webhook).
   * Copy the Webhook Secret – you'll need this for verifying the webhook signature in Ballerina.

Step 1.2: MongoDB Setup

1. Choose MongoDB Deployment:
   * MongoDB Atlas (Recommended): Cloud-hosted MongoDB. Easy to set up, scalable, and production-ready.
     + Go to [cloud.mongodb.com](https://www.google.com/url?sa=E&q=https%3A%2F%2Fcloud.mongodb.com%2F) and sign up.
     + Create a new Cluster (M0 Free Tier is sufficient for testing).
     + Configure network access (allow connections from your application's IP addresses).
     + Create a Database User with a strong password.
     + Get the Connection String for your application (e.g., mongodb+srv://<username>:<password>@<cluster-url>/<dbname>?retryWrites=true&w=majority).
   * Local Installation: Download and install MongoDB Community Server for your OS. This is suitable for local development.
2. Database and Collections:
   * You'll use one database (e.g., codenet\_db).
   * Initial collections:
     + users: To store basic user info synced from Clerk (e.g., Clerk userId, email, name, join date).
     + projects: To store project details.

Step 1.3: Java Spring Boot Project Setup

This will be your main API backend.

1. Create Spring Boot Project:
   * Go to [start.spring.io](https://www.google.com/url?sa=E&q=https%3A%2F%2Fstart.spring.io%2F)
   * Project: Maven Project or Gradle Project (Maven is common).
   * Language: Java
   * Spring Boot: Latest Stable version (e.g., 3.2.x)
   * Group: com.codenet
   * Artifact: codenet-backend
   * Packaging: Jar
   * Java: 17 or higher
   * Dependencies:
     + Spring Web (for REST endpoints)
     + Spring Data MongoDB (for MongoDB integration)
     + Lombok (optional, for less boilerplate code like getters/setters)
     + Spring Security (essential for securing your APIs, especially with Clerk JWTs)
     + Spring Boot DevTools (optional, for faster development cycles)
   * Click Generate and download the ZIP. Extract it.
2. Configure application.properties:  
   In src/main/resources/application.properties, add your MongoDB connection details:

Generated properties

spring.data.mongodb.uri=mongodb+srv://YOUR\_MONGO\_USER:YOUR\_MONGO\_PASSWORD@YOUR\_MONGO\_CLUSTER.mongodb.net/codenet\_db?retryWrites=true&w=majority

# Or for local: spring.data.mongodb.host=localhost

# spring.data.mongodb.port=27017

# spring.data.mongodb.database=codenet\_db

# Spring Security & Clerk Integration (add these later, but here for context)

clerk.secret-key=sk\_live\_YOUR\_CLERK\_SECRET\_KEY

clerk.api-url=https://api.clerk.com/v1 # This is the default

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Properties

1. Create User Model (in Spring Boot):  
   Even though Clerk manages users, you'll likely want to store a simplified version or additional app-specific data in your own DB.

Generated java

// src/main/java/com/codenet/codenetbackend/model/AppUser.java

package com.codenet.codenetbackend.model;

import lombok.Data;

import org.springframework.data.annotation.Id;

import org.springframework.data.mongodb.core.mapping.Document;

import java.time.Instant;

@Data // From Lombok, generates getters, setters, equals, hashCode, toString

@Document(collection = "users")

public class AppUser {

@Id

private String id; // This will store Clerk's userId

private String email;

private String username;

private Instant joinedAt;

// Add any other specific fields you need for your app that Clerk doesn't manage

private String role; // e.g., "Developer", "Admin"

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

1. Create User Repository (in Spring Boot):

Generated java

// src/main/java/com/codenet/codenetbackend/repository/AppUserRepository.java

package com.codenet.codenetbackend.repository;

import com.codenet.codenetbackend.model.AppUser;

import org.springframework.data.mongodb.repository.MongoRepository;

import java.util.Optional;

public interface AppUserRepository extends MongoRepository<AppUser, String> {

Optional<AppUser> findByEmail(String email);

Optional<AppUser> findByUsername(String username);

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

1. Create User Service (in Spring Boot):

Generated java

// src/main/java/com/codenet/codenetbackend/service/AppUserService.java

package com.codenet.codenetbackend.service;

import com.codenet.codenetbackend.model.AppUser;

import com.codenet.codenetbackend.repository.AppUserRepository;

import org.springframework.stereotype.Service;

import java.time.Instant;

import java.util.Optional;

@Service

public class AppUserService {

private final AppUserRepository appUserRepository;

public AppUserService(AppUserRepository appUserRepository) {

this.appUserRepository = appUserRepository;

}

public AppUser createUser(String clerkUserId, String email, String username) {

AppUser newUser = new AppUser();

newUser.setId(clerkUserId); // Store Clerk's user ID as our \_id

newUser.setEmail(email);

newUser.setUsername(username);

newUser.setJoinedAt(Instant.now());

newUser.setRole("Developer"); // Default role

return appUserRepository.save(newUser);

}

public Optional<AppUser> getUserById(String id) {

return appUserRepository.findById(id);

}

public AppUser updateUser(AppUser user) {

return appUserRepository.save(user);

}

public void deleteUser(String id) {

appUserRepository.deleteById(id);

}

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

Step 1.4: Ballerina Service for Clerk Webhooks (Optional but Robust)

Ballerina is excellent for integrating different systems and handling event-driven scenarios like webhooks.

1. Install Ballerina: Follow instructions at [ballerina.io/downloads/](https://www.google.com/url?sa=E&q=https%3A%2F%2Fballerina.io%2Fdownloads%2F).
2. Create a Ballerina Project:

Generated bash

bal new clerk\_webhook\_listener

cd clerk\_webhook\_listener

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Bash

1. Add Dependencies: Add http for the service and potentially mongodb if you decide to interact directly (though calling Spring Boot is cleaner for separation of concerns).
   * In Ballerina.toml, under [build-options], ensure stdlib is configured for http.
2. Create Ballerina Service (service.bal):  
   This service will receive webhooks from Clerk, verify their signature, and then call your Spring Boot backend to sync the user.

Generated ballerina

import ballerina/http;

import ballerina/log;

import ballerina/jwt;

import ballerina/crypto;

import ballerina/mime; // For parsing application/json

// Replace with your actual Clerk Webhook Secret (from Clerk Dashboard -> Webhooks)

configurable string CLERK\_WEBHOOK\_SECRET = "whsec\_YOUR\_CLERK\_WEBHOOK\_SECRET";

// Replace with your Spring Boot backend's user sync endpoint

configurable string SPRING\_BOOT\_USER\_SYNC\_URL = "http://localhost:8080/api/users/sync";

service /clerk-webhook on new http:Listener(8080) { // Or a different port like 9090

resource function post .(@http:Header {name: "svix-id"} string svixId,

@http:Header {name: "svix-timestamp"} string svixTimestamp,

@http:Header {name: "svix-signature"} string svixSignature,

http:Request req) returns http:Ok|http:BadRequest {

// 1. Get raw request body (needed for signature verification)

string jsonBody = check req.getTextPayload();

log:printInfo("Received Clerk Webhook", body = jsonBody);

// 2. Verify Webhook Signature (CRUCIAL for security)

// Clerk uses Svix for webhooks. Verification logic is based on Svix docs.

// Simplified example, production code should use a robust library if available

// or implement the full verification detailed in Svix documentation.

// Basic idea: Concatenate timestamp, ".", and payload, sign with secret.

string signedContent = svixId + "." + svixTimestamp + "." + jsonBody;

string key = CLERK\_WEBHOOK\_SECRET;

// Simplified verification for demonstration. In production, use a library

// that implements the full Svix verification (HMAC-SHA256, compare signatures securely).

// A common approach is to use the `webhook-verifier` library if available for Ballerina,

// or implement it as per Svix documentation.

// For now, we'll assume a successful verification for flow purposes.

// For actual verification: https://docs.svix.com/receiving/verifying-payloads/how

// Example of a basic, \*incomplete\* verification idea. DO NOT USE IN PRODUCTION AS IS.

// You need to parse svixSignature to get the version and actual signature,

// then compute HMAC-SHA256 of signedContent with your secret, and compare.

// Proper handling involves splitting `svixSignature` at ',', then splitting parts at '=',

// finding `v1` signature, and comparing it securely with your computed hash.

// If verification fails, return http:BadRequest.

// if (!verifySvixSignature(svixId, svixTimestamp, svixSignature, jsonBody, CLERK\_WEBHOOK\_SECRET)) {

// log:printError("Webhook signature verification failed.");

// return http:BadRequest();

// }

log:printInfo("Webhook signature assumed to be verified (implement full check!).");

// 3. Parse Payload

json payload = check json.parse(jsonBody);

string eventType = payload.eventType.toString();

if (eventType == "user.created" || eventType == "user.updated") {

json userData = payload.data;

string clerkUserId = userData.id.toString();

string email = userData.email\_addresses[0].email\_address.toString(); // Assuming primary email

string? username = userData.username?.toString(); // Username might be null

log:printInfo("Clerk User Event", eventType = eventType, userId = clerkUserId, email = email);

// 4. Call Spring Boot User Sync Endpoint

http:Client backendClient = new (SPRING\_BOOT\_USER\_SYNC\_URL, {

// This assumes your Spring Boot endpoint handles the full path, e.g. http://localhost:8080/api/users/sync

// You might need to adjust based on your Spring Boot Controller design.

// If you have a specific endpoint like /api/users/sync-clerk-user

});

json userSyncPayload = {

"id": clerkUserId,

"email": email,

"username": username

};

// The Spring Boot service should have an endpoint like POST /api/users/sync-clerk-user

// and accept a DTO with id, email, username

// If it's `POST /api/users` for general creation, you might call that.

// For simplicity, let's assume `POST /api/users/sync` and send a simple JSON.

http:Response|error response = backendClient->post("", userSyncPayload); // Path "" as base URL is already set

if (response is http:Response) {

log:printInfo("Spring Boot sync call successful", status = response.statusCode);

} else {

log:printError("Error calling Spring Boot for user sync", err = response.toString());

return http:BadRequest(); // Indicate failure

}

} else if (eventType == "user.deleted") {

json userData = payload.data;

string clerkUserId = userData.id.toString();

log:printInfo("Clerk User Deleted Event", userId = clerkUserId);

// Call Spring Boot to delete user

http:Client backendClient = new (SPRING\_BOOT\_USER\_SYNC\_URL + "/" + clerkUserId);

http:Response|error response = backendClient->delete("");

if (response is http:Response) {

log:printInfo("Spring Boot delete call successful", status = response.statusCode);

} else {

log:printError("Error calling Spring Boot for user deletion", err = response.toString());

return http:BadRequest();

}

}

return http:Ok();

}

}

// Helper function for Svix signature verification - this is a placeholder!

// You MUST implement the full Svix verification as per their documentation for production.

// This is a complex cryptographic operation.

function verifySvixSignature(string svixId, string svixTimestamp, string svixSignature, string payload, string secret) returns boolean {

// Implement full Svix webhook signature verification here

// This involves splitting svixSignature, calculating HMAC-SHA256, and comparing.

// For production, you might look for a Ballerina Svix library or carefully implement it.

return true; // DANGER: Placeholder - always return false if verification fails

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Ballerina

* + Note: The verifySvixSignature function is a critical security component. The placeholder return true is DANGEROUS for production. You must implement the full HMAC-SHA256 verification as described in the Svix documentation (Clerk uses Svix for webhooks).
  + To run Ballerina: bal run

Step 1.5: Spring Boot Endpoint for User Sync

Now, create the corresponding endpoint in your Spring Boot app that Ballerina will call.

1. Create a DTO for User Sync:

Generated java

// src/main/java/com/codenet/codenetbackend/dto/ClerkUserSyncDTO.java

package com.codenet.codenetbackend.dto;

import lombok.Data;

@Data

public class ClerkUserSyncDTO {

private String id; // Clerk's user ID

private String email;

private String username;

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

1. Add AppUserController:

Generated java

// src/main/java/com/codenet/codenetbackend/controller/AppUserController.java

package com.codenet.codenetbackend.controller;

import com.codenet.codenetbackend.dto.ClerkUserSyncDTO;

import com.codenet.codenetbackend.model.AppUser;

import com.codenet.codenetbackend.service.AppUserService;

import lombok.extern.slf4j.Slf4j;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/api/users")

@Slf4j // For logging

public class AppUserController {

private final AppUserService appUserService;

public AppUserController(AppUserService appUserService) {

this.appUserService = appUserService;

}

@PostMapping("/sync")

public ResponseEntity<AppUser> syncClerkUser(@RequestBody ClerkUserSyncDTO userSyncDTO) {

log.info("Received user sync request for Clerk ID: {}", userSyncDTO.getId());

// Check if user already exists

return appUserService.getUserById(userSyncDTO.getId())

.map(existingUser -> {

// Update existing user

existingUser.setEmail(userSyncDTO.getEmail());

existingUser.setUsername(userSyncDTO.getUsername());

log.info("Updating existing user: {}", existingUser.getId());

return ResponseEntity.ok(appUserService.updateUser(existingUser));

})

.orElseGet(() -> {

// Create new user

log.info("Creating new user with Clerk ID: {}", userSyncDTO.getId());

AppUser newUser = appUserService.createUser(userSyncDTO.getId(), userSyncDTO.getEmail(), userSyncDTO.getUsername());

return ResponseEntity.ok(newUser);

});

}

@DeleteMapping("/sync/{clerkUserId}")

public ResponseEntity<Void> deleteClerkUser(@PathVariable String clerkUserId) {

log.info("Received user delete request for Clerk ID: {}", clerkUserId);

appUserService.deleteUser(clerkUserId);

return ResponseEntity.noContent().build();

}

// Endpoint to get current user details from Clerk (via backend proxy)

// This requires Spring Security and Clerk JWT verification setup

// This will be added in a later step related to Spring Security

@GetMapping("/me")

public ResponseEntity<AppUser> getCurrentUserDetails(@RequestAttribute("clerkUserId") String clerkUserId) {

return appUserService.getUserById(clerkUserId)

.map(ResponseEntity::ok)

.orElseGet(() -> ResponseEntity.notFound().build());

}

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

Phase 2: Project Management (Spring Boot & MongoDB)

Now we build the core functionality for projects.

Step 2.1: Project Model Definition

Generated java

// src/main/java/com/codenet/codenetbackend/model/Project.java

package com.codenet.codenetbackend.model;

import lombok.Data;

import org.springframework.data.annotation.Id;

import org.springframework.data.mongodb.core.mapping.Document;

import java.time.Instant;

import java.util.ArrayList;

import java.util.List;

@Data

@Document(collection = "projects")

public class Project {

@Id

private String id;

private String title;

private String subtitle;

private String description;

private String ownerId; // Clerk's userId of the project owner

private List<String> mediaUrls = new ArrayList<>(); // URLs to uploaded images/videos

private long likes = 0;

private Instant uploadDate;

private String status = "PENDING"; // E.g., PENDING, APPROVED, REJECTED

// Add more fields as needed: categories, tags, GitHub link, live demo link, collaborators etc.

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

Step 2.2: Project Repository

Generated java

// src/main/java/com/codenet/codenetbackend/repository/ProjectRepository.java

package com.codenet.codenetbackend.repository;

import com.codenet.codenetbackend.model.Project;

import org.springframework.data.mongodb.repository.MongoRepository;

import org.springframework.data.mongodb.repository.Query;

import java.util.List;

public interface ProjectRepository extends MongoRepository<Project, String> {

List<Project> findByOwnerId(String ownerId);

// Custom query for search

@Query("{ $or: [ { 'title': { $regex: ?0, $options: 'i' } }, { 'description': { $regex: ?0, $options: 'i' } } ] }")

List<Project> searchProjects(String searchTerm);

// Find top N projects by likes (example for trending)

List<Project> findTop10ByOrderByLikesDesc(); // Adjust N and criteria as needed

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

Step 2.3: Project Service

Generated java

// src/main/java/com/codenet/codenetbackend/service/ProjectService.java

package com.codenet.codenetbackend.service;

import com.codenet.codenetbackend.model.Project;

import com.codenet.codenetbackend.repository.ProjectRepository;

import org.springframework.stereotype.Service;

import java.time.Instant;

import java.util.List;

import java.util.Optional;

@Service

public class ProjectService {

private final ProjectRepository projectRepository;

public ProjectService(ProjectRepository projectRepository) {

this.projectRepository = projectRepository;

}

public Project createProject(Project project, String ownerId) {

project.setOwnerId(ownerId);

project.setUploadDate(Instant.now());

project.setStatus("PENDING"); // New projects are pending approval

project.setLikes(0);

return projectRepository.save(project);

}

public Optional<Project> getProjectById(String id) {

return projectRepository.findById(id);

}

public List<Project> getAllApprovedProjects() {

// Only return approved projects for public viewing

return projectRepository.findByStatus("APPROVED");

}

public List<Project> getProjectsByOwner(String ownerId) {

return projectRepository.findByOwnerId(ownerId);

}

public List<Project> searchProjects(String query) {

return projectRepository.searchProjects(query);

}

public Optional<Project> updateProject(String id, Project updatedProject) {

return projectRepository.findById(id)

.map(project -> {

project.setTitle(updatedProject.getTitle());

project.setSubtitle(updatedProject.getSubtitle());

project.setDescription(updatedProject.getDescription());

project.setMediaUrls(updatedProject.getMediaUrls());

// Admin can change status, user cannot directly

// project.setStatus(updatedProject.getStatus());

return projectRepository.save(project);

});

}

public boolean deleteProject(String id) {

if (projectRepository.existsById(id)) {

projectRepository.deleteById(id);

return true;

}

return false;

}

public Optional<Project> likeProject(String id) {

return projectRepository.findById(id)

.map(project -> {

project.setLikes(project.getLikes() + 1);

return projectRepository.save(project);

});

}

public List<Project> getTrendingProjects() {

// Example: top 10 most liked projects

return projectRepository.findTop10ByOrderByLikesDesc();

}

// You'll need other methods for stats, e.g., getTotalProjects(), getTotalUsers()

public long getTotalProjects() {

return projectRepository.count();

}

public long getTotalUsers() {

// This count should come from your AppUser repository, not projects

return 0; // Placeholder, implement in AppUserService or a dedicated StatService

}

public long getSoldProjects() {

// This assumes a 'sold' flag or status in Project model

return 0; // Placeholder

}

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

Step 2.4: Project Controller

Generated java

// src/main/java/com/codenet/codenetbackend/controller/ProjectController.java

package com.codenet.codenetbackend.controller;

import com.codenet.codenetbackend.model.Project;

import com.codenet.codenetbackend.service.ProjectService;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.util.List;

@RestController

@RequestMapping("/api/projects")

public class ProjectController {

private final ProjectService projectService;

public ProjectController(ProjectService projectService) {

this.projectService = projectService;

}

@PostMapping

public ResponseEntity<Project> createProject(@RequestBody Project project,

@RequestAttribute("clerkUserId") String clerkUserId) {

// clerkUserId will be injected by our Spring Security config (see Phase 3)

Project createdProject = projectService.createProject(project, clerkUserId);

return new ResponseEntity<>(createdProject, HttpStatus.CREATED);

}

@GetMapping("/{id}")

public ResponseEntity<Project> getProjectById(@PathVariable String id) {

return projectService.getProjectById(id)

.map(ResponseEntity::ok)

.orElseGet(() -> ResponseEntity.notFound().build());

}

@GetMapping

public List<Project> getAllProjects(@RequestParam(required = false) String search) {

if (search != null && !search.isEmpty()) {

return projectService.searchProjects(search);

}

return projectService.getAllApprovedProjects(); // Only show approved projects

}

@PutMapping("/{id}")

public ResponseEntity<Project> updateProject(@PathVariable String id,

@RequestBody Project project,

@RequestAttribute("clerkUserId") String clerkUserId) {

// Add authorization check: Only owner can update

return projectService.getProjectById(id)

.filter(existingProject -> existingProject.getOwnerId().equals(clerkUserId))

.flatMap(existingProject -> projectService.updateProject(id, project))

.map(ResponseEntity::ok)

.orElseGet(() -> ResponseEntity.status(HttpStatus.FORBIDDEN).build()); // Or NOT\_FOUND if ID doesn't exist

}

@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteProject(@PathVariable String id,

@RequestAttribute("clerkUserId") String clerkUserId) {

// Add authorization check: Only owner can delete

return projectService.getProjectById(id)

.filter(existingProject -> existingProject.getOwnerId().equals(clerkUserId))

.map(existingProject -> {

projectService.deleteProject(id);

return ResponseEntity.noContent().<Void>build();

})

.orElseGet(() -> ResponseEntity.status(HttpStatus.FORBIDDEN).build());

}

@PutMapping("/{id}/like")

public ResponseEntity<Project> likeProject(@PathVariable String id) {

return projectService.likeProject(id)

.map(ResponseEntity::ok)

.orElseGet(() -> ResponseEntity.notFound().build());

}

@GetMapping("/trending")

public List<Project> getTrendingProjects() {

return projectService.getTrendingProjects();

}

// You'll need a separate controller or method for overall stats

@GetMapping("/stats")

public ResponseEntity<Object> getStats() {

// This will combine counts from ProjectService and AppUserService

long totalProjects = projectService.getTotalProjects();

long totalUsers = appUserService.getTotalUsers(); // Assuming appUserService is injected here

long soldProjects = projectService.getSoldProjects(); // Placeholder

return ResponseEntity.ok(

new StatsDTO(totalProjects, totalUsers, soldProjects, 0) // Collaborators placeholder

);

}

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

Generated java

// src/main/java/com/codenet/codenetbackend/dto/StatsDTO.java

package com.codenet.codenetbackend.dto;

import lombok.AllArgsConstructor;

import lombok.Data;

@Data

@AllArgsConstructor

public class StatsDTO {

private long projectsUploaded;

private long users;

private long soldProjects;

private long collaborators;

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

Phase 3: Authentication & Authorization with Clerk and Spring Security

This is crucial for securing your Spring Boot APIs.

1. Add spring-boot-starter-security to pom.xml:

Generated xml

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</token>

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Xml

1. Create a Security Configuration:  
   This configures Spring Security to validate JWTs issued by Clerk.

Generated java

// src/main/java/com/codenet/codenetbackend/config/SecurityConfig.java

package com.codenet.codenetbackend.config;

import com.auth0.jwk.JwkProvider;

import com.auth0.jwk.JwkProviderBuilder;

import com.auth0.jwt.JWT;

import com.auth0.jwt.algorithms.Algorithm;

import com.auth0.jwt.interfaces.DecodedJWT;

import jakarta.servlet.FilterChain;

import jakarta.servlet.ServletException;

import jakarta.servlet.http.HttpServletRequest;

import jakarta.servlet.http.HttpServletResponse;

import lombok.extern.slf4j.Slf4j;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.http.HttpMethod;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.config.http.SessionCreationPolicy;

import org.springframework.security.core.authority.SimpleGrantedAuthority;

import org.springframework.security.core.context.SecurityContextHolder;

import org.springframework.security.web.SecurityFilterChain;

import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;

import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

import java.net.URL;

import java.util.Collections;

import java.util.Date;

import java.util.concurrent.TimeUnit;

@Configuration

@EnableWebSecurity

@Slf4j

public class SecurityConfig {

@Value("${clerk.secret-key}")

private String clerkSecretKey;

// Clerk's JWT issuer URL

@Value("${clerk.issuer-url:https://clerk.YOUR\_PUBLISHABLE\_KEY\_DOMAIN/.well-known/jwks.json}")

// Example: https://clerk.YOUR\_PUBLISHABLE\_KEY\_DOMAIN/.well-known/jwks.json

// You can find this in your Clerk dashboard under JWT Templates -> Public JWK URL

private String clerkJwksUrl;

@Bean

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http

.csrf(csrf -> csrf.disable()) // Disable CSRF for stateless APIs

.cors(cors -> {}) // Enable CORS (configure CorsConfigurationSource bean)

.authorizeHttpRequests(authorize -> authorize

// Public endpoints (no authentication needed)

.requestMatchers(HttpMethod.GET, "/api/projects/\*\*", "/api/trending", "/api/stats").permitAll()

.requestMatchers(HttpMethod.POST, "/api/users/sync").permitAll() // For Ballerina webhook

.requestMatchers(HttpMethod.DELETE, "/api/users/sync/\*\*").permitAll() // For Ballerina webhook

.requestMatchers(HttpMethod.POST, "/api/public/login", "/api/public/signup").permitAll() // If you decide to proxy Clerk auth

// All other API endpoints require authentication

.requestMatchers("/api/\*\*").authenticated()

.anyRequest().permitAll() // Allow access to static files (frontend)

)

.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)) // Use stateless sessions (JWT)

.addFilterBefore(clerkJwtAuthenticationFilter(), UsernamePasswordAuthenticationFilter.class); // Add custom JWT filter

return http.build();

}

@Bean

public OncePerRequestFilter clerkJwtAuthenticationFilter() {

return new OncePerRequestFilter() {

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {

String authHeader = request.getHeader("Authorization");

if (authHeader != null && authHeader.startsWith("Bearer ")) {

String token = authHeader.substring(7);

try {

// Clerk uses its own JWT signing keys. We need to fetch them.

JwkProvider jwkProvider = new JwkProviderBuilder(new URL(clerkJwksUrl))

.cached(10, 24, TimeUnit.HOURS) // Cache JWKs for 24 hours

.rateLimited(10, 1, TimeUnit.MINUTES) // 10 requests per minute

.build();

DecodedJWT jwt = JWT.decode(token);

Algorithm algorithm = Algorithm.RSA256(jwkProvider.get(jwt.getKeyId()).getPublicKey(), null); // Use public key from JWKS

algorithm.verify(jwt); // Verify signature

// Check token validity (expiration, issuer, audience)

if (jwt.getExpiresAt().before(new Date())) {

throw new RuntimeException("JWT token has expired.");

}

// Further checks: issuer (iss), audience (aud) claims if needed.

// For Clerk, 'iss' claim typically looks like 'https://clerk.YOUR\_PUBLISHABLE\_KEY\_DOMAIN/'

// And 'aud' claims should match your application's expected audience (often your frontend domain).

// Get Clerk User ID from token

// Clerk uses 'sub' (subject) claim for userId or a custom 'sid'

// From Clerk docs, 'sub' is the user ID.

String clerkUserId = jwt.getSubject();

// Store Clerk user ID in request attributes for controllers to use

request.setAttribute("clerkUserId", clerkUserId);

// You can also load user details from your MongoDB if needed for roles/permissions

// For simplicity, we're just authenticating based on Clerk JWT and providing userId

UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(

clerkUserId, null, Collections.singletonList(new SimpleGrantedAuthority("USER"))); // Default role

SecurityContextHolder.getContext().setAuthentication(authentication);

} catch (Exception e) {

log.error("JWT authentication failed: {}", e.getMessage());

response.setStatus(HttpServletResponse.SC\_UNAUTHORIZED);

return;

}

}

filterChain.doFilter(request, response);

}

};

}

// Configure CORS for frontend access

@Bean

public CorsConfigurationSource corsConfigurationSource() {

CorsConfiguration configuration = new CorsConfiguration();

configuration.setAllowedOrigins(List.of("http://127.0.0.1:5500", "http://localhost:8080")); // Your frontend origins

configuration.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));

configuration.setAllowedHeaders(List.of("\*")); // Allow all headers

configuration.setAllowCredentials(true); // Allow sending cookies/auth headers

UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

source.registerCorsConfiguration("/\*\*", configuration);

return source;

}

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

* + IMPORTANT: Replace clerk.issuer-url with your actual Clerk JWKS URL. This URL provides the public keys used by Clerk to sign JWTs. You can find it in your Clerk dashboard under "JWT Templates" -> "Public JWK URL". It usually looks like https://clerk.YOUR\_PUBLISHABLE\_KEY\_DOMAIN/.well-known/jwks.json.
  + CORS: Ensure http://127.0.0.1:5500 (or whatever Live Server uses) and any future deployment URLs for your frontend are in setAllowedOrigins.

Phase 4: File Storage (Media Uploads)

You'll need a place to store images/videos for projects. Local storage is only for development; cloud storage is essential for production.

1. Choose a Cloud Storage Provider (Recommended):
   * AWS S3: Very common, scalable, reliable.
   * Google Cloud Storage: Similar to S3.
   * Azure Blob Storage: Microsoft's offering.
2. Spring Boot Integration (Example with a placeholder for local/cloud):

Generated java

// src/main/java/com/codenet/codenetbackend/service/FileStorageService.java

package com.codenet.codenetbackend.service;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.stereotype.Service;

import org.springframework.web.multipart.MultipartFile;

import java.io.File;

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.util.UUID;

@Service

public class FileStorageService {

// For local storage (DEVELOPMENT ONLY!)

@Value("${file.upload-dir:./uploads}")

private String uploadDir;

public String storeFile(MultipartFile file) throws IOException {

// \*\* FOR PRODUCTION: INTEGRATE WITH AWS S3, GCP CLOUD STORAGE, OR AZURE BLOB STORAGE \*\*

// Example: using AWS S3 SDK (you'd add aws-java-sdk-s3 dependency)

// S3Client s3Client = S3Client.builder().build();

// String key = UUID.randomUUID().toString() + "\_" + file.getOriginalFilename();

// s3Client.putObject(PutObjectRequest.builder().bucket("your-s3-bucket").key(key).build(),

// RequestBody.fromInputStream(file.getInputStream(), file.getSize()));

// return "https://your-s3-bucket.s3.amazonaws.com/" + key;

// \*\* For local development (REMOVE FOR PRODUCTION DEPLOYMENT!) \*\*

Path uploadPath = Paths.get(uploadDir).toAbsolutePath().normalize();

Files.createDirectories(uploadPath); // Create directory if it doesn't exist

String fileName = UUID.randomUUID().toString() + "\_" + file.getOriginalFilename();

Path filePath = uploadPath.resolve(fileName);

file.transferTo(filePath);

return "/uploads/" + fileName; // Return a URL path

}

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

* + NOTE: For local storage, you'll need to configure Spring Boot to serve static content from the uploads directory. Add this to SecurityConfig or a WebMvcConfig:

Generated java

// In SecurityConfig.java, inside securityFilterChain method

// ...

.authorizeHttpRequests(authorize -> authorize

// ... existing rules ...

.requestMatchers("/uploads/\*\*").permitAll() // Allow access to uploaded files

.anyRequest().authenticated()

)

// ...

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

1. Update ProjectController for File Uploads:  
   Modify the createProject and updateProject methods to handle MultipartFile.

Generated java

// In ProjectController.java

import com.codenet.codenetbackend.service.FileStorageService; // Import this

// Add FileStorageService to constructor

private final FileStorageService fileStorageService;

public ProjectController(ProjectService projectService, FileStorageService fileStorageService) {

this.projectService = projectService;

this.fileStorageService = fileStorageService;

}

@PostMapping(consumes = {"multipart/form-data"}) // Change consume type

public ResponseEntity<Project> createProject(@RequestPart("project") Project project, // JSON part

@RequestPart(value = "media", required = false) List<MultipartFile> mediaFiles, // Files part

@RequestAttribute("clerkUserId") String clerkUserId) throws IOException {

if (mediaFiles != null && !mediaFiles.isEmpty()) {

List<String> mediaUrls = new ArrayList<>();

for (MultipartFile file : mediaFiles) {

mediaUrls.add(fileStorageService.storeFile(file));

}

project.setMediaUrls(mediaUrls);

}

Project createdProject = projectService.createProject(project, clerkUserId);

return new ResponseEntity<>(createdProject, HttpStatus.CREATED);

}

// Similar adjustments for updateProject if you allow media updates

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

* + Note: @RequestPart is used for multipart/form-data. Your frontend form for submitproject.html will need to have enctype="multipart/form-data". The project data will need to be sent as a JSON string within a form-data part.

Phase 5: Frontend Integration & Refinements

This is where your HTML/JS will start talking to the new backend.

Step 5.1: Frontend Authentication (Revisit login.html, signup.html, userprofile.html)

* For login.html and signup.html:
  + If using Clerk's pre-built UI (recommended):
    - Remove your existing forms.
    - Add a div element: <div id="auth-container"></div>.
    - In a <script> tag, initialize Clerk's SDK and mount the components:

Generated javascript

// Add this to your HTML <head> or at the end of <body>

<script async defer src="https://YOUR\_CLERK\_PUBLISHABLE\_KEY\_DOMAIN/\_\_clerk/js/clerk.browser.js" data-clerk-publishable-key="pk\_live\_YOUR\_CLERK\_PUBLISHABLE\_KEY"></script>

<script>

window.onload = function() {

if (window.Clerk && window.Clerk.isReady()) {

// For login page

Clerk.mountSignIn(document.getElementById('auth-container'));

// For signup page

// Clerk.mountSignUp(document.getElementById('auth-container'));

}

};

</script>

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).JavaScript

You'll need separate files or conditional logic for login vs. signup.  
After successful login/signup, Clerk redirects the user or updates the session. You'd likely redirect to codenet.html.

* + For userprofile.html:
    - Again, use Clerk's UserProfile component for a fully managed profile UI:

Generated html

<div id="user-profile-container"></div>

<script>

window.onload = function() {

if (window.Clerk && window.Clerk.isReady()) {

Clerk.mountUserProfile(document.getElementById('user-profile-container'));

}

};

</script>

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Html

* + - Or, if you want to keep your custom UI, use Clerk's user object from the JS SDK to populate your HTML elements:

Generated javascript

window.onload = async function() {

if (window.Clerk && window.Clerk.isReady()) {

const user = Clerk.user;

if (user) {

document.getElementById('profile-name').textContent = user.fullName || user.username;

document.getElementById('profile-email').textContent = user.primaryEmailAddress?.emailAddress;

// ... populate other fields ...

} else {

// User not logged in, redirect to login

window.location.href = 'login.html';

}

}

};

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).JavaScript

* Getting JWT for Backend Calls:  
  Once a user is logged in via Clerk, you can get their JWT token like this:

Generated javascript

async function getAuthToken() {

if (window.Clerk && window.Clerk.session) {

return await Clerk.session.getToken();

}

return null;

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).JavaScript

You'll include this token in the Authorization header (Bearer <token>) of all authenticated requests to your Spring Boot backend.

Step 5.2: Frontend API Calls (Update codelibrary.html, submitproject.html, codenet.html, userprofile.html)

You'll replace placeholder data with actual API calls.

* submitproject.html (Submit Project):
  + Modify your form to have enctype="multipart/form-data".
  + When submitting, gather form data. Create a FormData object.
  + For the project JSON part, you'll need to append it as a Blob or string to FormData.
  + Example fetch call:

Generated javascript

document.querySelector('.submit-form').addEventListener('submit', async function(e) {

e.preventDefault();

const authToken = await getAuthToken();

if (!authToken) {

alert('Please log in to submit a project.');

window.location.href = 'login.html';

return;

}

const title = document.getElementById('title').value;

const subtitle = document.getElementById('subtitle').value;

const description = document.getElementById('description').value;

const mediaFiles = document.getElementById('media').files;

const formData = new FormData();

// Append project details as a JSON string

const projectData = { title, subtitle, description };

formData.append('project', new Blob([JSON.stringify(projectData)], { type: 'application/json' }));

// Append media files

for (let i = 0; i < mediaFiles.length; i++) {

formData.append('media', mediaFiles[i]);

}

try {

const response = await fetch('http://localhost:8080/api/projects', { // Replace with your backend URL

method: 'POST',

headers: {

'Authorization': `Bearer ${authToken}`

},

body: formData // No 'Content-Type' header needed for FormData; browser sets it

});

if (response.ok) {

alert('Project submitted successfully!');

// Clear form or redirect

this.reset();

} else {

const error = await response.json();

alert(`Failed to submit project: ${error.message || response.statusText}`);

}

} catch (error) {

console.error('Error submitting project:', error);

alert('An error occurred while submitting the project.');

}

});

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).JavaScript

* codelibrary.html (Code Library, Search):
  + Fetch projects from GET /api/projects.
  + Handle search by appending ?search=query to the URL.
  + Populate project-list dynamically.
* codenet.html (Home Page - Trending Projects, Stats):
  + Fetch trending projects from GET /api/trending.
  + Fetch stats from GET /api/stats.
  + Update your stats counters and project carousels.
* userprofile.html (My Projects):
  + If using Clerk's UserProfile component, it will handle most data.
  + To display "My Projects", you'll need to fetch them from your backend: GET /api/projects/my (or GET /api/projects?ownerId=clerkUserId, but my is cleaner). You'll need to add this endpoint to your ProjectController in Spring Boot, secured by the JWT.

Generated java

// In ProjectController.java

@GetMapping("/my")

public List<Project> getMyProjects(@RequestAttribute("clerkUserId") String clerkUserId) {

return projectService.getProjectsByOwner(clerkUserId);

}

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).Java

Then, from frontend, after getting clerkUserId from Clerk JS SDK:

Generated javascript

async function fetchMyProjects() {

const authToken = await getAuthToken();

if (!authToken) return; // Not logged in

try {

const response = await fetch('http://localhost:8080/api/projects/my', {

headers: {

'Authorization': `Bearer ${authToken}`

}

});

if (response.ok) {

const projects = await response.json();

// Render projects in your userprofile.html

} else {

console.error('Failed to fetch my projects:', await response.text());

}

} catch (error) {

console.error('Error fetching my projects:', error);

}

}

// Call this when userprofile.html loads and user is logged in

Use code [with caution](https://support.google.com/legal/answer/13505487" \t "_blank).JavaScript

Step 5.3: Error Handling & UI Feedback

* Frontend: Add try-catch blocks around fetch calls. Show loading spinners, success messages, and error alerts to the user.
* Backend: Spring Boot automatically converts exceptions to HTTP error codes. Use @ExceptionHandler for custom error handling or ResponseEntity to return specific error messages.

Phase 6: Deployment Considerations

* Spring Boot:
  + Build JAR: mvn clean install or gradlew bootJar.
  + Deploy: Run on a VM (java -jar your-app.jar), Dockerize, or deploy to PaaS (e.g., Heroku, Railway, AWS Elastic Beanstalk).
* Ballerina:
  + Build Executable: bal build
  + Run: bal run
  + Deploy: Similar to Spring Boot, often as a separate microservice.
* MongoDB:
  + Use MongoDB Atlas for production.
* Clerk: SaaS, no deployment needed from your side.
* Frontend:
  + Serve statically from your Spring Boot app (src/main/resources/static folder) or use a static web server like Nginx, Apache, or a service like Netlify/Vercel.

Key Considerations & Best Practices

* Environment Variables: Never hardcode sensitive information (API keys, database credentials) directly in code. Use environment variables (Spring Boot's @Value, Ballerina's configurable).
* Input Validation: Always validate input on the backend (e.g., using jakarta.validation annotations in Spring Boot) to prevent malformed data or attacks.
* Logging: Use Slf4j in Spring Boot and ballerina/log in Ballerina to log important events, errors, and debugging information.
* Testing: Write unit and integration tests for your Spring Boot and Ballerina services.
* Version Control: Use Git (you seem to be using it!) and host your code on GitHub/GitLab/Bitbucket.
* CI/CD: Automate your build, test, and deployment processes (e.g., GitHub Actions, GitLab CI/CD).
* Security Headers: Configure Spring Security to add appropriate security headers (e.g., XSS protection, content security policy).
* Rate Limiting: Implement rate limiting on your API endpoints to prevent abuse.
* Admin Panel: Consider building a simple admin panel (another frontend app or within your existing one) for approving/rejecting projects, managing users, etc.

This detailed plan should give you a solid roadmap to build out your CODENET backend! Good luck!