

Chapter 5: System Implementation




5.1 Introduction






In this chapter, we discuss the process where the system design was converted into a system implementation. Section 5.2 introduces the tools and languages that have been used to complete the project. Next, section 5.3 explains how the system design from Chapter 4 connects to the implementation phase. Section 5.4 provides an overview of our project's main algorithm, breaking it down with explanations. Section 5.5 outlines the tests conducted to ensure the system functions correctly and meets user expectations. Finally, Section 5.6 analyses the test result and connects them to the project's goal.

5.2 Tools and Languages

In this section, the list of tools and languages used for implementing the system are shown in table 5-1

Table 5- 1 Tools and Languages

Icon	Tool/Language	Description	Use
	JAVA	Java is a powerful widely used programming language known for its portability and security.	Used for the backend services and controllers, managing business logic and handling requests from the front end.
	Spring Boot	A framework for Java that simplifies backend development, providing built-in configurations for rapid deployment.	was used to set up RESTful APIs, handling requests and responses between the backend and the database.
	MongoDB	A NoSQL database designed for high performance and flexibility, storing data in JSON-like documents instead of relational tables	was used to store and retrieve data efficiently, allowing us to manage and query information dynamically.

	Visual Studio	An integrated development environment (IDE) used for coding, debugging, and building applications	was the main development tool where we wrote and tested our code for both frontend and backend components.
	React	A JavaScript library for building user interfaces, allowing components to update dynamically	React was used to develop the front-end of our website, ensuring interactive and responsive user experience.
	Tailwind CSS	A utility-first CSS framework that simplifies styling by allowing developers to apply predefined classes directly to HTML elements	Tailwind CSS was used to design and style the front-end
	HTML	The standard markup language used to structure web pages by defining elements and content	was used to build the structure of our front-end,
	JavaScript	A programming language that enables interactivity and dynamic elements on web pages	was used to add interactivity, enhance user experience, and integrate front-end functionality with backend services.

5.3 Mapping Design to Implementation

In the previous chapter, the system design was discussed. However, at the implementation phase, some features were changed. We changed to using a NoSQL database, and this affected how data was stored and accessed, as relationships between entities were no longer managed through foreign keys but with document references instead. Additionally, for the class diagram, methods placement was changed from being inside the model to the controller and service classes. These changes improved system scalability and guaranteed flexibility in dealing with data without complex joins. The revised architectural diagram and

class structures reflect these changes and align the implementation with the updated database structure while maintaining the core principles of our initial design.

5.4 Main/Most Important Codes

The system was developed as a full-stack web application using Spring Boot for the backend and React for the frontend. MongoDB served as the primary database, and the system was architected to support multiple user roles, including students, HR managers, and supervisors. In this section we provide the main code part and functions used to build the app.

5.4.1 Internships Recommendation

In this project, rather than training a custom model from scratch, a pre-trained transformer-based language model was employed to perform semantic understanding and recommendation. The model can generate contextual embeddings of text inputs, such as internship descriptions and student profiles. These embeddings capture deeper semantic relationships, making them highly suitable for content-based recommendation systems.

Since no supervised training was involved, there was no need to collect or label a dataset. Instead, the model operated on dynamic input data of student profile attributes such as major, skills, and interests. And Internship descriptions include job titles, required major, and job descriptions.

The feature extraction step involved encoding each student profile and each internship description into a fixed-size vector embedding using BERT's [CLS] token representation as shown in figure 5-1. These embeddings served as high-dimensional (384-dim) feature vectors used in the similarity comparison.

```
@Service
public class EmbeddingService {

    private final BertModel bertModel;
    private static final int EXPECTED_EMBEDDING_SIZE = 384; // Expected size of the embedding vector

    // Constructor to initialize BertModel
    public EmbeddingService(BertModel bertModel) {
        this.bertModel = bertModel;
    }

    // Generates an embedding vector for the given text input
    public List<Float> generateEmbedding(String text) throws TranslateException {
        List<Float> embedding = bertModel.getEmbedding(text);

        // Validates that the generated embedding has the expected size
        if (embedding.size() != EXPECTED_EMBEDDING_SIZE) {
            throw new IllegalStateException("Embedding size mismatch! Expected: " + EXPECTED_EMBEDDING_SIZE +
                ", but got: " + embedding.size());
        }

        return embedding;
    }
}
```

Figure 5- 1 Feature extraction

The method generateEmbedding is used in internship service as well as user controller as shown in figure 5-2,5-3 and passed a normalized text to generate embeddings for both student and internships.

```
// Generating embeddings for internship
String combinedText = title + " " + description + " " +
    String.join(" ", internship.getRequiredSkills()) + " " +
    String.join(" ", internship.getMajors());
String normalizedText = combinedText.trim().toLowerCase().replaceAll("\\s+", " ");
List<Float> embedding = embeddingService.generateEmbedding(normalizedText);
internship.setEmbedding(embedding);
```

Figure 5- 2 Internship Embeddings

```

// Generate and save embedding in the first login of students
if (student != null && (student.getEmbedding() == null || student.getEmbedding().isEmpty())) {
    try {
        String text = student.getMajor() + " " + student.getLocation() + " " + student.getSkills();
        String normalizedText = text.trim().toLowerCase().replaceAll("\\s+", " ");
        List<Float> embedding = embeddingService.generateEmbedding(normalizedText);
        student.setEmbedding(embedding);
        mongoTemplate.save(student, "students");
    } catch (TranslateException e) {
        System.out.println("Embedding generation failed: " + e.getMessage());
    }
}

```

Figure 5- 3 Student Embeddings

The `getRecommendedInternships` method shown in figure 5-4 implements the core functionality of the recommendation system by retrieving a list of internship opportunities relevant for a particular student. The method uses the student's embedding. In order to make sure the input embedding is not null or empty, it first validates it. If the validation is unsuccessful, an exception is raised. After validation, the method builds a vector search query that uses cosine similarity and MongoDB's vector indexing with k-nearest neighbors (k-NN) to identify embeddings most similar to the student's profile. Specifically, it searches using the student's embedding against the vector index, retrieving up to 300 candidate matches and selecting the top 15 based on cosine similarity, limited to 12 results for display.

```

public List<Internship> getRecommendedInternships(List<Float> studentEmbedding, String studentMajor) {
    // Validate that the embedding is not null or empty
    if (studentEmbedding == null || studentEmbedding.isEmpty()) {
        System.err.println("Invalid student embedding: " + studentEmbedding);
        throw new IllegalArgumentException("Invalid student embedding!");
    }

    // Build the vector search query to find similar internships
    Document vectorSearchQuery = new Document("$vectorSearch",
        new Document("index", "internship_index2")
            .append("queryVector", studentEmbedding)
            .append("path", "embedding")
            .append("numCandidates", 300)
            .append("k", 15)
            .append("limit", 12));

    // Filter results to only include active internships matching the student's
    // major
    Document matchStage = new Document("$match", new Document("status", "active")
        .append("majors", studentMajor));

    // Exclude the embedding field from the final result
    Document projectStage = new Document("$project",
        new Document("embedding", 0));

    // Run the aggregation pipeline on the internships collection
    List<Document> results = mongoTemplate.getCollection("internships")
        .aggregate(List.of(vectorSearchQuery, matchStage, projectStage))
        .into(new ArrayList<>());

    // Convert MongoDB documents into Internship objects
    List<Internship> internships = results.stream()
        .map(doc -> mongoTemplate.getConverter().read(Internship.class, doc))
        .toList();

    return internships;
}

```

Figure 5- 4 Recommended Internships method

5.4.3 Authentication and Login

The AuthService class is responsible for retrieving information about the currently logged-in user as depicted in figure 5-3. It works with Spring Security to access the authentication details stored in the security context. This service includes two main methods. The first method, `getAuthenticatedUserId()`, gets the email of the authenticated user and then uses it to find their user ID from the database.

The second method, `getAuthenticatedUserRole()`, retrieves the user's role (such as student, supervisor, or admin). This service is useful for identifying the current user and managing role-based access in the system.

```

@Service
public class AuthService {

    private final UserService userService;

    public AuthService(UserService userService) {
        this.userService = userService;
    }

    // Retrieves the authenticated user's ID from the security context
    public String getAuthenticatedUserId() {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

        // Get the user's email from the authentication object
        String email = authentication.getName();

        // Find the user by their email in the database
        User user = userService.findByEmail(email);

        // Return user ID if found, otherwise return null
        return (user != null) ? user.getId() : null;
    }

    // Retrieves the authenticated user's role from the security context
    public String getAuthenticatedUserRole() {
        Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

        // Get the first authority (role) assigned to the user
        return authentication.getAuthorities().iterator().next().getAuthority();
    }
}

```

Figure 5- 5 Authentication service

The login endpoint in figure 5-6 verifies the provided email and password using the previously defined Authentication Manager. If successful, the authentication object is stored in the Security Context, and in the HTTP session for subsequent requests. The authenticated User object is retrieved and used to determine the role of the logged-in user. On a student's first login, if they do not already have an embedding, one is generated based on their major, location, and skills. This embedding is then stored in the database for future use.


```

@PostMapping("/login")
public ResponseEntity<> login(@RequestBody LoginRequest loginRequest, HttpServletRequest request) {}

try {
    // Authenticate user using email and password
    Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(loginRequest.getEmail(), loginRequest.getPassword()));

    // Store the authentication in the security context and session
    SecurityContextHolder.getContext().setAuthentication(authentication);
    request.getSession().setAttribute("SPRING_SECURITY_CONTEXT", SecurityContextHolder.getContext());

    User user = (User) authentication.getPrincipal();

    // If the logged-in user is a student, check if their embedding exists
    if ("STUDENT".equalsIgnoreCase(user.getUserRole())) {
        Query studentQuery = new Query(Criteria.where("email").is(loginRequest.getEmail()));
        Student student = mongoTemplate.findOne(studentQuery, Student.class, "students");

        // Generate and save embedding in the first login of students
        if (student != null && (student.getEmbedding() == null || student.getEmbedding().isEmpty())) {
            try {
                String text = student.getMajor() + " " + student.getLocation() + " " + student.getSkills();
                String normalizedText = Text.trim().toLowerCase().replaceAll("\\s+", " ");
                List<Float> embedding = embeddingService.generateEmbedding(normalizedText);
                student.setEmbedding(embedding);
                mongoTemplate.save(student, "students");
            } catch (TranslateException e) {
                System.out.println("Embedding generation failed: " + e.getMessage());
            }
        }
    }

    return ResponseEntity.ok().build();
} catch (BadCredentialsException ex) {
    System.out.println("Invalid credentials: " + ex.getMessage());
    return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("Invalid email or password");
} catch (Exception ex) {
    System.out.println("Unexpected error: " + ex.getMessage());
    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("An unexpected error occurred");
}

```

Figure 5- 6 Login method

5.4.4 Conversation service:

The Conversation service class as depicted in figure 5-7 handles the creation and retrieval of one-on-one conversations between users within the system. The `getOrCreateConversation` method is responsible for ensuring a unique conversation exists between two distinct users. It begins by preventing self-conversations and verifying that both users exist via the `UserService`. If a conversation already exists between the two user IDs, it retrieves and returns it using a MongoDB query that checks for both participants and ensures the conversation includes only the two users. If no such conversation exists, a new `Conversation` object is created and saved to the conversations collection.

The `getUserConversations` method retrieves all conversations in which a given user is a participant. It performs a simple query on the conversations collection to find and return all matching documents. These methods together support the chat functionality by maintaining

clean, non-duplicated conversation threads and ensuring data integrity when initiating new chats.

```
public ConversationService(MongoTemplate mongoTemplate, UserService userService) {
    this.mongoTemplate = mongoTemplate;
    this.userService = userService;
}

public Conversation getOrCreateConversation(String user1Id, String user2Id) {
    // Prevent users from starting a conversation with themselves
    if (user1Id.equals(user2Id)) {
        throw new IllegalArgumentException("Cannot start a conversation with yourself.");
    }

    // Ensure both users exist in the system
    if (!userService.userExistsByEmail(user1Id) || !userService.userExistsByEmail(user2Id)) {
        throw new RuntimeException("One or both users do not exist.");
    }

    // Query to check if a conversation between these users already exists
    Query query = new Query();
    query.addCriteria(new Criteria().andOperator(
        Criteria.where("participantIds").all(user1Id, user2Id),
        Criteria.where("participantIds").size(2)));

    Conversation existing = mongoTemplate.findOne(query, Conversation.class, "conversations");

    // Return existing conversation if found, otherwise create a new one
    if (existing != null) {
        return existing;
    }

    Conversation newConversation = new Conversation(List.of(user1Id, user2Id));
    mongoTemplate.save(newConversation, "conversations");
    return newConversation;
}

public List<Conversation> getUserConversations(String userId) {
    // Retrieve all conversations where the user is a participant
    Query query = new Query();
    query.addCriteria(Criteria.where("participantIds").in(userId));
    return mongoTemplate.find(query, Conversation.class, "conversations");
}
```

Figure 5- 7 Conversation Service

To enable real-time communication in the system we included a WebSocket configuration class shown in figure 5-8 . This class sets up the necessary infrastructure for STOMP-over-WebSocket communication. Within the `configureMessageBroker` method, a simple in-memory message broker is enabled with the `/topic` prefix, allowing clients to subscribe and receive messages. The `/app` prefix is designated for application-level destinations that clients use when sending messages to the server. The `registerStompEndpoints` method registers the WebSocket endpoint `/ws`, which clients connect to using libraries like SockJS and stomp.js.

```

@Configuration
@EnableWebSocketMessageBroker // Enables WebSocket message handling
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker("/topic"); // Defines a topic-based message broker for subscribers
        config.setApplicationDestinationPrefixes("/app"); // Prefix for messages sent by clients
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/ws") // Defines WebSocket endpoint
            .setAllowedOrigins("http://localhost:5173") // Allows requests from the system's frontend
            .withSockJS(); // Enables SockJS fallback for clients without native WebSocket support
    }
}

```

Figure 5- 8 WebSocket configuration

5.4.5 Task progress :

To provide supervisors with a quick overview of task statuses, the system includes a method called `getTaskProgressSummary()` shown in figure 5-9 . This method calculates the percentage of tasks that are completed, pending, or overdue for a given supervisor. It begins by querying the database for all tasks associated with the specified supervisor ID. Then, it categorizes these tasks based on their status: completed tasks, pending tasks (not completed and with a future due date), and overdue tasks (not completed and past the due date). After counting each category, it calculates their respective percentages relative to the total number of tasks, ensuring to handle cases with no tasks to avoid division by zero. The method returns a map containing the percentage breakdown, allowing the frontend to visualize the supervisor's overall task management performance in real time.

```

public Map<String, Double> getTaskProgressSummary(String supervisorId) {
    // Fetch tasks for the given supervisor
    Query query = new Query();
    query.addCriteria(Criteria.where("supervisorId").is(supervisorId));

    List<Task> tasks = mongoTemplate.find(query, Task.class);

    // Count tasks by category
    long completed = tasks.stream().filter(Task::isCompleted).count();
    long pending = tasks.stream().filter(t -> !t.isCompleted() && t.getDueDate().isAfter(LocalDate.now())).count();
    long overdue = tasks.stream().filter(t -> !t.isCompleted() && t.getDueDate().isBefore(LocalDate.now())).count();

    // Calculate the total number of tasks
    long totalTasks = completed + pending + overdue;

    // Avoid division by zero
    if (totalTasks == 0) {
        return Map.of(
            "completed", 0.0,
            "pending", 0.0,
            "overdue", 0.0);
    }

    // Calculate percentages
    double completedPercentage = (double) completed / totalTasks * 100;
    double pendingPercentage = (double) pending / totalTasks * 100;
    double overduePercentage = (double) overdue / totalTasks * 100;

    // Return percentages in a map
    return Map.of(
        "completed", completedPercentage,
        "pending", pendingPercentage,
        "overdue", overduePercentage);
}

```

Figure 5- 9 Task Progress

5.5 System Testing:

System testing is done to ensure everything in the system works as it should and meets the needed requirements. It checks all the main features to confirm they function properly without any errors or issues.

Table 5-2 demonstrates the main functionalities of the system and its response to each one of them.

Table 5- 2 Assign supervisor test case

Test Case: Assign supervisor	
Test Description:	To allow HR manager to assign a supervisor to a list of students.
Authorized User:	HR manager.

Test Condition:	The HR manager is logged in, and students have been accepted.		
Test Steps:	Test Input	Expected Result	Test Result
1. The HR manager opens the Assign Supervisor page.	HR manager clicks on the "Assign Supervisor" option on the page.	System navigates to the "Assign Supervisor" page.	Pass
2. System displays students that have been accepted.	System loads the page	A list of accepted students is displayed.	Pass
3. The HR manager selects a list of students.	HR manager checks the boxes to students.	The "Assign" button becomes enabled.	Pass
4. System displays supervisors' names.	System processes to fetch supervisor data.	The list of supervisors is shown.	Pass
5. The HR manager selects a supervisor.	HR selected one supervisor from the list.	A selected supervisor is assigned to the selected students.	Pass
6. The HR clicks assign a supervisor button.	HR clicks the "Assign" button after selecting supervisor.	The system display Supervisor assigned successfully! message.	Pass

5.6 Results and discussion:

Our system achieved its primary objectives by automating and streamlining several critical aspects of the internship process. Through its modular design and integration of modern technologies, the system facilitated smooth interaction between students, faculty supervisors, and company representatives.

Key features were developed and tested successfully. Students were able to view available internships, apply to them, track the status of their applications, and communicate directly with supervisors through a chat module. This chat functionality was implemented using WebSocket and STOMP protocols, allowing real-time messaging between users within an active conversation session.

Supervisors, on the other hand, had access to tools for assigning tasks to students, monitoring task completion, and viewing a progress summary that included completed, pending, and overdue tasks. This allowed for a more organized and data-driven supervision process. The task progress summary provided insights into student performance and time management based on real deadlines and task statuses.

Each module was designed to work cohesively using a Spring Boot backend and MongoDB for data persistence. The system was able to respond correctly to various input scenarios, including data validation, user-specific access, and interaction constraints, such as preventing users from starting conversations with themselves.

Overall, the results demonstrated that the system is functional, user-aware, and logically structured to support the internship process. The implementation relied on sound software engineering principles and offers a solid foundation for future enhancements, such as performance optimization, improved user interfaces, and analytics dashboards.

5.6 Summary:

In this chapter, we explained how the system design was turned into a working system. First, we listed the tools, programming languages, and technologies used in the project. Then, we showed how the design from the previous chapter was used during development. We also described the main algorithm used in the system and explained how it works. After that, we gave an overview of the testing process. Finally, we discussed the results of these tests and how they relate to the system's goals.