Presentation Script: Healthcare CRM

Slide 1: Title Slide

- **(All Team / Team Lead):** "Good morning/afternoon everyone. We are [Team Name, list members: John Shaize, Darshan Kiran Upadhyay, Gaurang Hareshkumar Dhameliya, Ramees Karolil Rasheed]. Today we'll be presenting our J2EE Business Components project: the Healthcare CRM."
- "This presentation will cover our project goals, the technologies we used, the features we implemented, our deployment strategy, and a live demonstration."

Slide 2: Introduction & Project Goal

- **(Presenter: Darshan):** "Thank you. The primary goal of this project was to develop a functional Minimum Viable Product, or MVP, for a Healthcare CRM system using the Spring Framework and adhering to J2EE principles."
- "The core purpose is to provide healthcare centers with a tool to manage essential information: patient or customer records, their insurance details, and the assignment and tracking of tasks for clinic employees."
- "Our key objectives included implementing secure, role-based access, providing core CRUD functionalities for the main data entities, enabling basic task management, and importantly, demonstrating that the application can be deployed reliably using containerization."

Slide 3: Technology Stack

- **(Presenter: Gaurang):** "To achieve these goals, we utilized a modern technology stack based primarily on the Spring ecosystem."
- "For the backend, we used Java 21 with Spring Boot 3.4.2, leveraging Spring MVC for handling web requests, Spring Data JPA with Hibernate for object-relational mapping to our database, and Spring Security for robust authentication and authorization."
- "The frontend web interface was built using Thymeleaf for server-side template rendering, styled with Bootstrap 5, HTML5, and CSS3."
- "Our database is MySQL version 8."
- "We also implemented REST APIs using Spring Web and documented them using Springdoc OpenAPI, which provides the interactive Swagger UI."
- "For deployment, we used Docker and Docker Compose to containerize both the application and the database."
- "And finally, the project was built using Apache Maven."

Slide 4: System Architecture

- **(Presenter: Gaurang):** "Architecturally, the application follows a standard layered approach."
- *(Refer to diagram if included)* "User interactions, whether through the browser or API calls, hit our Presentation Layer. This consists of Thymeleaf views rendered by Spring MVC controllers for the web UI, and separate REST Controllers handling JSON requests for the API."

- "Requests are then processed by the Service Layer, which contains the core business logic for operations like creating customers or updating task statuses. These services coordinate interactions with the Data Access Layer."
- "The Data Access Layer uses Spring Data JPA repositories – interfaces that abstract away much of the boilerplate database code – to interact with our MySQL database via Hibernate."
- "The entire application, along with the MySQL database, is designed to run within Docker containers orchestrated by Docker Compose."

Slide 5: Security Implementation

- **(Presenter: Darshan):** "Security was a key focus, implemented using Spring Security."
- "For Authentication, we have a standard form-based login page. User credentials (email/password) are verified against data stored in our database using a custom `UserDetailsService` that checks both Admin and Employee users. Passwords are never stored in plain text; we use BCrypt hashing for secure storage."
- "Authorization is role-based. We defined two main roles: `ROLE_ADMIN` and `ROLE_EMPLOYEE`. Access to different parts of the application, like the admin sections or employee-specific pages, is restricted based on these roles using rules defined in our `SecurityConfig` file."
- "The REST APIs also leverage this same security mechanism, requiring users to be authenticated via the web session. For easier testing with tools like Swagger UI, we specifically disabled CSRF protection for the API paths, while keeping it enabled for the main web application."
- "Lastly, the system ensures a default admin user is created if the database is initially empty, providing a secure starting point."

Slide 6: Backend Features & Database

- **(Presenter: Gaurang):** "Moving to the core backend logic and data persistence..."
- "We modeled our main entities – Customer, Employee, Task, Insurance, and Admin – using JPA annotations. This allowed Hibernate to manage the relationship between our Java objects and the database tables. We used annotations like `@Entity`, `@OneToOne`, and `@ManyToOne` to define the structure and relationships, such as a Customer having one Insurance record, or a Task being assigned to one Customer and one Employee. Lombok was used to reduce repetitive code like getters and setters."
- "The Service layer implements the main business operations. For example, when creating or updating a Customer, the service ensures the associated Insurance details are also saved correctly, leveraging JPA's cascading features. The repositories provide methods like `save`, `findById`, and even custom queries, like the one we use in the Task service to find overdue tasks for the admin follow-up feature."
- "The database schema itself is managed by Hibernate using the `ddl-auto=update` setting during development, which automatically creates or updates tables based on our entity definitions. The database name is `healthcarecrm`."
- "To make the application usable immediately, a `DataInitializer` component runs on startup to populate the database with a default admin and various mock employees, customers, and tasks if the database is detected to be empty."

Slide 7: Frontend UI/UX & Validation

- **(Presenter: John):** "Now let's look at the user-facing web interface."
- "We used Thymeleaf as our template engine, allowing us to generate dynamic HTML on the server by integrating directly with the data provided by our Spring MVC controllers."
- "To ensure a consistent user experience, we implemented shared layouts for the Admin and Employee sections using the Thymeleaf Layout Dialect. This means common elements like the navigation bar and sidebar are defined once in a layout template, and content pages decorate this layout. We encountered and resolved a path resolution issue here when running from the JAR."
- "Styling was handled using Bootstrap 5 for responsive design and pre-built components, supplemented by our own custom styles centralized in `custom.css`. This includes standardized navigation elements and our chosen color scheme."
- *(Briefly reference key screenshots shown on slide)* "Here you can see the login page, the admin dashboard, examples of the management tables for customers and tasks, and the employee task list."
- "Finally, robust data validation is crucial. We implemented server-side validation using Jakarta Bean Validation annotations directly on our model classes (like `@NotEmpty`, `@Size`, `@Email`). When validation fails, errors are passed back to Thymeleaf and displayed clearly to the user on the form, preventing invalid data submission. We also used Validation Groups for scenarios like requiring a password only when creating a new employee, not when updating one."

Slide 8: REST API Implementation

- **(Presenter: Ramees):** "In addition to the web UI, we implemented a RESTful API to allow programmatic access to the CRM data."
- "This involved creating Spring MVC `RestController` classes for Customer, Employee, and Task resources, mapped under the `/api` path prefix."
- "These controllers handle standard HTTP methods – GET for retrieving data, POST for creating, PUT for updating, and DELETE for removing resources. They primarily function by calling the same underlying Service layer methods used by the web controllers, ensuring consistent business logic."
- "The APIs communicate using JSON for request and response bodies."
- "To make the API discoverable and testable, we integrated the `springdoc-openapi` library. By adding annotations like `@Tag` and `@Operation` to our API controllers, we automatically generate OpenAPI specification. This is rendered as an interactive Swagger UI, accessible at the `/swagger-ui.html` endpoint when the application is running. This allows developers (or testers) to easily see available endpoints, parameters, and even try out API calls directly from the browser." *(Show screenshot)*.

Slide 9: Deployment with Docker

- **(Presenter: Ramees):** "A key requirement was demonstrating containerized deployment. We achieved this using Docker and Docker Compose."
- "The goal was to package the application and its MySQL database dependency so they can run consistently in any environment where Docker is installed."
- "First, we created a `Dockerfile`. This defines the steps to build our application's image. It starts from a base Java 21 image, copies the compiled JAR file (created by `mvn clean package`), exposes the application port 8080, and sets the command to run the application using `java -jar`."
- "Next, we created a `docker-compose.yml` file to orchestrate the containers. This file defines two services: `app` for our Spring Boot application (which uses the `Dockerfile` to build its image) and `db` which uses the official MySQL 8 image."

- "Docker Compose automatically handles networking between these containers. We configured environment variables within the compose file so that the `app` container knows how to connect to the `db` container using the service name 'db' as the hostname. We also configured the necessary MySQL credentials."
- "A crucial addition was the `healthcheck` within the `db` service definition. This allows Docker Compose to verify that the MySQL server is actually ready to accept connections before starting the `app` service, which solved an initial startup timing issue we encountered. We also used a named volume to ensure MySQL data persists even if the container is stopped and restarted."
- "This containerization provides significant benefits, including environment consistency, simplified setup for new developers or deployment targets, and better management of application dependencies."

Slide 10: Live Demonstration

- **(Presenters: Coordinate beforehand who shows which part - e.g., Ramees starts Docker, Gaurang shows Admin, John shows Employee, Darshan shows API/Swagger)**:
- "Now, we'll demonstrate the live application running via Docker Compose."
- *(Perform the demo steps outlined in the content slide, narrating the actions and highlighting features)*
    - "First, we start the application using `docker compose up`..." *(Show terminal briefly if possible)*
    - "Here is the login page accessible at localhost:8080..."
    - "Logging in as admin..."
    - "This is the admin dashboard showing key counts..."
    - "Navigating to Customer management... we can see the list, expand details including insurance..." *(Maybe quickly show add/edit)*
    - "Similarly, we have Employee and Task management..."
    - "And the Follow-up center showing overdue/due soon tasks..."
    - "Now, logging out and logging in as an employee..." *(Use a mock employee login)*
    - "Here's the employee's task list... Let's update this task status from Pending to In Progress..." *(Show update)*
    - "Finally, logging out again... We can also briefly show the Swagger UI which documents our REST API..." *(Navigate to Swagger UI)*

Slide 11: Challenges & Limitations

- **(Presenter: John):** "During the project, we encountered a few challenges and acknowledge some limitations."
- "The main challenge was managing scope within the timeframe, which led us to descope comprehensive automated testing (like unit and API tests) to ensure delivery of the core functional MVP and the containerization requirement."
- "On the technical side, setting up Docker Compose required troubleshooting the database startup dependency – the app tried to connect before the database was ready. We resolved this using Docker Compose healthchecks. We also had to resolve a Thymeleaf path issue specific to running from a JAR file, and navigated some Docker command differences between environments."
- "Key limitations of the final product include minimal automated test coverage, relying primarily on manual verification. Also, advanced features mentioned in initial requirements, like microservices or asynchronous processing, were not implemented as part of this MVP."

Slide 12: Conclusion & Future Work

- **(Presenter: Any):** "In conclusion, we have successfully developed and delivered a functional Minimum Viable Product for a Healthcare CRM."
- "The application demonstrates key features including role-based security, data management via both a web UI and REST APIs, task tracking, and containerized deployment using Docker."
- "This project provided valuable practical experience with the Spring Boot ecosystem, including Security, Data JPA, and MVC, as well as frontend development with Thymeleaf, REST API design with OpenAPI, and modern deployment practices using Docker Compose. We also learned significant lessons in troubleshooting deployment and configuration issues."
- "For future work, the most obvious next step would be implementing comprehensive automated tests. Other enhancements could include adding advanced search and filtering capabilities to the UI, implementing more granular user permissions, hardening the application for production environments, and potentially exploring some of the advanced features initially considered."

Slide 13: Q & A

- **(All Team / Team Lead):** "That concludes our presentation. Thank you for your time. We are now open for any questions you may have."
- *(Be prepared to answer questions about specific implementation details, design choices, or challenges)*
