**Client Tracking System Documentation**

**Step-by-Step Guide for Beginners**

This documentation explains how the Client Tracking System works in simple steps. It will help a beginner understand your project easily.

**1.What is This Project?**

The Client Tracking System is a web application that helps manage clients, projects, resources (like employees), and work timetables. It provides APIs (a way to communicate with the system) to add, view, update, or delete data related to clients, projects, and resources. The system uses MySQL as the database and Swagger for API documentation.

**2. Step-by-Step Explanation**

**Step 1:** **Configure the Application**

SwaggerConfig.java : Configures Swagger for API documentation.

Provides details like API title, description, and version.

Accessible at /swagger-ui.html.

**Step 2: Define Models**

Models represent the database tables. Each model corresponds to an entity with attributes and relationships.

**1) Client.java**

Represents a client with fields like clientId, clientName, clientEmail, and clientProject.

Relationships: A client can have multiple projects (@ManyToOne).

**2) ClientProject.java**

Links clients and projects.

Fields: clientProjectId, client, and projectId.

**3) ProjectTable.java**

Represents a project with fields like projectId, projectName, and description.

**4) Resource.java**

Represents a resource (e.g., an employee) with fields like resourceId, resourceName, and resourceRole.

**5) WorkTimetable.java**

Tracks work hours for resources on specific projects.

Fields: workId, client, resource, workDate, and hoursWorked.

**Step 3: Create Repositories**

Repositories interact with the database using Spring Data JPA.

ClientRepository.java : Handles database operations for the Client entity.

ClientProjectRepository.java : Handles database operations for the ClientProject entity.

ProjectTableRepository.java : Handles database operations for the ProjectTable entity.

ResourceRepository.java : Handles database operations for the Resource entity.

WorkTimetableRepository.java : Handles database operations for the WorkTimetable entity.

**Step 4: Build Controllers**

Controllers handle HTTP requests and responses.

ClientController.java

Endpoints:

GET /clients: Get all clients.

GET /clients/{id}: Get a client by ID.

POST /clients: Add a new client.

PUT /clients/{id}: Update a client.

DELETE /clients/{id}: Delete a client.

**ClientProjectController.java**

Similar endpoints for managing client-project relationships.

**ProjectTableController.java**

Similar endpoints for managing projects.

**ResourceController.java**

Similar endpoints for managing resources.

**WorkTimetableController.java**

Similar endpoints for managing work timetables.

**Step 5: Write Unit Tests**

Unit tests ensure the application works as expected.

ClientControllerTest.java : Tests all endpoints of the ClientController.

ClientProjectControllerTest.java : Tests endpoints for ClientProjectController.

ResourceControllerTest.java : Tests endpoints for ResourceController.

**Step 6: Configure the Database**

The application connects to a MySQL database using the configuration in application.properties.

Hibernate automatically creates or updates the database schema based on the models.

**Step 7: Run the Application**

Ensure MySQL is installed and running.

Create a database named client\_tracking or let Hibernate create it automatically.

**Run the application using :** mvn spring-boot:run

**Access Swagger UI at :** <http://localhost:8080/swagger-ui.html>

**Step 8: Test the APIs**

Use tools like Postman or Swagger UI to interact with the APIs.

**Example:**

To get all clients: GET http://localhost:8080/clients.

To add a new client: POST http://localhost:8080/clients with JSON body

{"clientName": "ABC Corp",

"clientEmail": [abc@example.com](mailto:abc@example.com)”}

**Tools and Technologies**

**Spring Boot :** Framework for building the application.

**Spring Data JPA :** Simplifies database interactions.

**MySQL :** Database for storing data**.**

**Swagger :** Tool to document and test APIs**.**

**JUnit & Mockito :** For testing the application.

**File Structure**

client-tracking/

├── src/

│ ├── main/

│ │ ├── java/com/example/clienttracking/

│ │ │ ├── ClientTrackingApplication.java

│ │ │ ├── config/

│ │ │ │ └── SwaggerConfig.java

│ │ │ ├── controller/

│ │ │ │ ├── ClientController.java

│ │ │ │ ├── ClientProjectController.java

│ │ │ │ ├── ProjectTableController.java

│ │ │ │ ├── ResourceController.java

│ │ │ │ └── WorkTimetableController.java

│ │ │ ├── model/

│ │ │ │ ├── Client.java

│ │ │ │ ├── ClientProject.java

│ │ │ │ ├── ProjectTable.java

│ │ │ │ ├── Resource.java

│ │ │ │ └── WorkTimetable.java

│ │ │ ├── repository/

│ │ │ │ ├── ClientRepository.java

│ │ │ │ ├── ClientProjectRepository.java

│ │ │ │ ├── ProjectTableRepository.java

│ │ │ │ ├── ResourceRepository.java

│ │ │ │ └── WorkTimetableRepository.java

│ │ └── resources/

│ │ └── application.properties

│ └── test/

│ └── java/com/example/clienttracking/

│ ├── ClientControllerTest.java

│ ├── ClientProjectControllerTest.java

│ └── ResourceControllerTest.java

├── pom.xml

└── README.md (Optional)

**Conclusion**

This project is a fully functional RESTful API-based system for managing clients, projects, resources, and work timetables. By following the steps above, you can easily understand how the application is structured and how it works.

**Annotation used in the project** :

@**Configuration** : Marks a class as a configuration file for Spring (like settings for the app).

@**Bean** : Tells Spring to create and manage an object (like Swagger setup).

@**RestController** : Makes a class handle API requests and return data directly.

@**RequestMapping** : Maps a URL path to a method or controller (e.g., /clients).

@**Operation** : Adds details about an API operation for documentation (Swagger).

@**Autowired** : Automatically injects dependencies (like services) into a class.

@**GetMapping** : Handles GET requests to fetch data from the server.

@**PostMapping** : Handles POST requests to add new data to the server.

@**PutMapping** : Handles PUT requests to update existing data on the server.

@**DeleteMapping** : Handles DELETE requests to remove data from the server.

@**PathVariable** : Extracts values from the URL (e.g., /clients/{id} → id).

@**RequestBody** : Converts JSON data from the request body into a Java object.

@**Entity** : Marks a class as a database table (each object is a row in the table).

@**Id** : Marks a field as the primary key in a database table.

@**GeneratedValue** : Automatically generates values for the primary key (e.g., 1, 2, 3…).

@**Column** : Maps a field to a column in the database table (with optional rules like "not null").

@**ManyToOne** : Defines a relationship where many items belong to one item (e.g., many projects for one client).

@**JoinColumn** : Specifies the foreign key column for a relationship.

@**Temporal** : Handles date/time fields in the database with precision (e.g., only date or date + time).

@**Repository** : Marks an interface as a place to interact with the database.

@**SpringBootApplication** : Combines three annotations to start a Spring Boot app.

@**Test** : Marks a method as a test case to check if the code works correctly.

@**Mock** : Creates a fake version of a dependency for testing.

@**InjectMocks** : Injects mocked dependencies into the class being tested.

@**ExtendWith** : Registers extensions for JUnit tests (e.g., MockitoExtension).

@**BeforeEach** : Runs setup code before each test method in JUnit.

@**CrossOrigin** : Allows requests from different websites (for APIs).