## **Background**

A **Risk Register** is a tool used in project management to identify, document, and track potential risks associated with a project. Risks can be anything that may negatively impact the project's scope, timeline, budget, or quality. By maintaining a risk register, project managers and teams can assess risks, define mitigation strategies, and monitor their impact over time.

### **What is a Risk?**

A risk is any uncertain event or condition that, if it occurs, can have a positive or negative effect on a project's objectives. Risks can be categorized as:

* **Strategic Risks:** Related to market changes, competition, or regulatory changes.
* **Operational Risks:** Related to internal processes, system failures, or resource constraints.
* **Financial Risks:** Related to cost overruns, funding issues, or economic downturns.
* **Technical Risks:** Related to technology adoption, software failures, or cybersecurity threats.

### **How a Risk Register is Used**

A risk register helps project teams:

* Identify potential risks early in the project lifecycle.
* Assess the likelihood and impact of risks.
* Define mitigation and contingency plans.
* Assign risk ownership to team members.
* Monitor and update risks throughout the project.

### **Example Risks in a Project**

| **ID** | **Risk Description** | **Likelihood** | **Impact** | **Mitigation Plan** | **Owner** |
| --- | --- | --- | --- | --- | --- |
| R1 | Server failure may cause downtime | High | Critical | Implement load balancing & backups | DevOps |
| R2 | Budget cuts may affect resources | Medium | High | Prioritize essential tasks & secure alternative funding | PM |
| R3 | API integration failure | Low | Medium | Test integrations early & provide fallbacks | Dev Team |

## **Assignment Task**

Your task is to design and develop a **Risk Register API** that will allow users to manage risks associated with a project. The application will not have a UI but will expose RESTful endpoints for risk management operations.

### **Functional Requirements**

**Estimated time: 12-15 hours**

Your API must support the following features:

1. **Risk Management**
   * Add a new risk (POST /projects/{projectId}/risks)
   * Retrieve a list of all risks for a project (GET /projects/{projectId}/risks)
   * Retrieve a specific risk by ID (GET /projects/{projectId}/risks/{id})
   * Update an existing risk (PUT /projects/{projectId}/risks/{id})
   * Delete a risk (DELETE /projects/{projectId}/risks/{id})
2. **Risk Attributes** Each risk should have the following attributes:  
   * **Risk ID** (Unique Identifier)
   * **Project ID** (Identifier for the associated project)
   * **Title** (Short description of the risk)
   * **Description** (Detailed explanation of the risk)
   * **Likelihood** (Low, Medium, High)
   * **Impact** (Low, Medium, High, Critical)
   * **Mitigation Plan** (Steps to reduce risk impact)
   * **Owner** (Person responsible for managing the risk)
   * **Status** (Open, Mitigated, Closed)
3. **Filtering & Sorting** **Estimated time: 3-5 hours**
   * Filter risks by **likelihood**, **impact**, or **status**.
   * Sort risks by **likelihood** or **impact** in ascending/descending order.
4. **Logging & Error Handling (Optional)** **Estimated time: 4-6 hours**
   * Implement proper logging for API requests and responses.
   * Return meaningful error messages for invalid inputs.

### **Technical Requirements**

**Estimated time: 6-8 hours**

* Use **Java** (Spring Boot) or **Python** (Flask/Django) for API development.
* Store risks in a **relational database** (PostgreSQL, MySQL, or SQLite).
* Use **JSON format** for API request and response.
* Write **unit tests** to verify API functionality.
* Document API endpoints using **Swagger or OpenAPI**.

### **Software Development Concepts Demonstrated**

**Estimated time: 5-7 hours**

This assignment will help you demonstrate proficiency in the following software development concepts:

* **RESTful API Design** ([Free Course: REST API Concepts - Udacity](https://www.udacity.com/course/intro-to-apis--ud117))
* **CRUD Operations** (Create, Read, Update, Delete) ([Free Course: SQL for Developers - W3Schools](https://www.w3schools.com/sql/))
* **Database Design & ORM (Object-Relational Mapping)** ([Free Course: Database Design - Khan Academy](https://www.khanacademy.org/computing/computer-programming/sql))
* **Filtering and Sorting Data** ([Free Resource: SQL Order By - W3Schools](https://www.w3schools.com/sql/sql_orderby.asp))
* **Logging & Error Handling (Optional)** ([Free Course: Logging Basics - Coursera](https://www.coursera.org/learn/logging-monitoring))
* **Unit Testing & Test-Driven Development (TDD)** ([Free Course: Test-Driven Development - Pluralsight](https://www.pluralsight.com/courses/test-driven-development))
* **API Documentation with Swagger/OpenAPI** ([Free Guide: Swagger Docs](https://swagger.io/docs/))
* **Authentication & Authorization (Bonus Challenge)** ([Free Course: OAuth 2.0 Basics - Auth0](https://auth0.com/learn/oauth/))
* **Software Development Best Practices** ([Free Guide: Clean Code - Robert C. Martin](https://github.com/ryanmcdermott/clean-code-javascript))

### **Submission Guidelines**

**Estimated time: 2-3 hours**

Submit the following items:

1. **Source code** (hosted on GitHub, Bitbucket, or ZIP file).
2. **Database schema** (SQL scripts or ORM models).
3. **API documentation** (Swagger/OpenAPI or markdown file).
4. **Unit tests** (demonstrating API correctness).
5. **README file** with instructions to set up and run the API.

### **Evaluation Criteria**

| **Criteria** | **Weight** |
| --- | --- |
| Code Quality & Best Practices | 25% |
| Functional Completeness | 30% |
| API Documentation | 15% |
| Unit Tests & Error Handling | 20% |
| Project Structure & Readability | 10% |

**Bonus Challenge** (Optional)  
 Enhance your API by implementing **OAuth 2.0 authentication** to secure risk management endpoints.

## **Using Layers in Code**

The following provides guidance on defining different layers within your code.

### **Benefits of Using a Layered Architecture with REST API**

1. **Separation of Concerns:** Each layer has a clear responsibility, making the code easier to understand and modify.
2. **Maintainability:** You can update one layer without affecting others.
3. **Reusability:** Service and DAO layers can be reused if you change the presentation layer (e.g., from console to web).
4. **Interoperability:** REST API allows external systems to interact with the application easily.
5. **Scalability:** Makes it easy to scale different parts of the application independently.
6. **Testability:** Each layer can be tested independently, making it easier to write unit tests.

### **1. Entity Layer (Risk class)**

**Purpose:**

* **The entity layer is responsible for defining the data model used in the application. It represents real-world objects as Java classes with attributes and methods.**

**Why it’s useful:**

* **Having a dedicated entity layer makes it easier to understand what kind of data your application deals with and ensures a consistent way to represent that data.**

### **2. Data Access Object (DAO) Layer (RiskDAO class)**

**Purpose:**

* **The DAO layer handles all interactions with the database. It’s responsible for performing CRUD (Create, Read, Update, Delete) operations using SQL queries.**
* **It separates database-specific logic from the rest of the application, which makes it easier to switch databases if needed.**

**Why it’s useful:**

* **Keeps database code separate from business logic, making code cleaner and easier to maintain.**
* **Makes it easier to handle database changes or switch databases without affecting other parts of the application.**

### **3. Service Layer (RiskService class)**

**Purpose:**

* **The service layer contains the business logic of the application. It acts as a bridge between the DAO layer and other parts of the application, such as the REST API layer.**
* **It decides how data should be processed before storing it in the database or presenting it to the user.**

**Why it’s useful:**

* **Keeps the main application code simple by moving complex logic into the service layer.**
* **Makes it easier to add new business rules or change existing ones without modifying database or UI code.**

### **4. REST API Layer (RiskController class)**

**Purpose:**

* **The REST API layer exposes the application’s functionality to external clients through HTTP endpoints.**
* **It allows external systems (like a web app or mobile app) to create, read, update, and delete risks by sending HTTP requests.**
* **It acts as an interface between the outside world and the service layer.**

**Why it’s useful:**

* **Allows different types of clients (web, mobile, other systems) to interact with the application in a standardized way using HTTP methods.**
* **Makes the application scalable and easy to integrate with other systems.**

### **5. Presentation Layer (Main class)**

**Purpose:**

* **The presentation layer is responsible for interacting with the user. It handles user input and displays output.**
* **In our example, it’s a console-based user interface for testing purposes.**

**Why it’s useful:**

* **Allows you to change the user interface (for example, switch to a web or desktop UI) without changing business or database logic.**
* **Keeps the UI code separate from data management and business logic.**

### **Summary of Layer Responsibilities**

| **Layer** | **Responsibilities** |
| --- | --- |
| **Entity Layer** | **Define data models as Java classes** |
| **DAO Layer** | **Handle database operations using SQL and JDBC** |
| **Service Layer** | **Manage business logic and coordinate between DAO and API** |
| **REST API Layer** | **Expose functionality to external systems via HTTP endpoints** |
| **Presentation Layer** | **Interact with the user and display data** |

# Collaboration Below Here

**Free Charting Tool**

[**https://www.mermaidchart.com/**](https://www.mermaidchart.com/)

**Pseudocode**

**Add a Risk Operation**

1. **Validate that risk info is valid (risk ID is not null, etc)**
2. **Check whether Risk already exists**
3. **If Risk exists**
   1. **Return error**
   2. **Abort**
4. **Else Insert risk**
   1. **Check result of insert**
   2. **If insert error**
      1. **Return error**
      2. **Abort**
   3. **Else**
      1. **Return success**

**Code Structure / Layers**

1. **Entity Layer**
   1. **This is the layer where you define your entities.**
   2. **For example, you would define your Risk class in here.**
   3. **The Risk class defines a risk entity. It does not contain any logic for saving/reading a risk from the database. This just allows you to define a valid risk entity.**
2. **Persistence Layer**
   1. **This is the layer that reads/writes from the database**
   2. **All your queries will be in here**
   3. **This layer will have create, read, update, and delete methods.**
   4. **This layer is very simple. For example, the create method just performs a create database operation (insert a new risk record with the risk info that’s passed to it).**
3. **Service Layer**
   1. **This is the layer where all of the business rules are, and is responsible for calling the persistence layer.**
   2. **It will also have create, read, update and delete methods. These methods will include additional logic. For example, the create method might check whether a risk exists first before performing the create**
4. **API (controller) Layer**
   1. **This is the layer where all of the APIs are.**
   2. **An API will call the appropriate Service layer code.**
   3. **There will be validation in here to validate API parameters, etc, and perform other relevant logic before calling the service layer**