UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Faculty of Engineering.

Computing Engineering

Subject: Software Modeling

Teacher: Carlos Andres Sierra

Project: Forms

Jaider Santiago Avila Robles **20231020200**

Daniel Santiago Pérez **20231020203**

Bogotá, D.C.

14 february 2025.

Introduction:

This project is an online form platform that allows users to create, customize, and share forms for data collection. The application is designed to be accessible from any device and has a robust backend implemented in Java and Python. Users can register, log in, create different types of forms (open-ended questions, multiple-choice, etc.), and view the responses in a clear and organized manner.

The primary goal of the application is to provide a simple and customizable solution for collecting and analyzing structured data, similar to tools like Google Forms. This technical report describes the architecture, design, implementation, and best practices used in the development of the application, as well as the design patterns and SOLID principles applied.

User stories

· As a user, I want to register on the platform to create and manage forms.

· As a user , I want to log in to access my forms and responses.

· As a user , I want to create a customized form to collect specific data.

· As a user , I want to edit the forms I have created previously to adjust them according to my needs.

· As a user, I want personalize type of the question

· As user , I want I want to answer forms

· As a user , I want to share my forms via link or email to collect responses.

· As a user , I want to view the responses to my forms clearly and download them if necessary.

· As a user , I want to see a history of the forms I've created and manage them as needed.

3. Functional and Non-Functional Requirements

Functional Requirements

- User Authentication:
  Users must be able to register and log in. Only authenticated users can create, edit, and view forms and responses.
- Form Creation:
  Users can create customized forms with different types of questions (multiple choice, short text, long text, etc.). Each form can contain up to 100 questions.
- Form Sharing:
  Forms can be shared via a link or email. Any user can respond to a form without authentication.

- Response Viewing:
  Users can view responses to their forms in a clear interface. Responses can be exported in formats like PDF and CSV.
- Form History:
  Users have access to a history of created and answered forms.

Non-Functional Requirements

- Scalability:
  The application must handle a large number of users and forms simultaneously.
- Security:
  User data and responses must be protected through encryption. The application must comply with privacy regulations.
- Performance:
  The application must respond in less than 2 seconds for most operations.
- Availability:
  The application must have an uptime of 99.9%.
- Compatibility:
  The application must be accessible from any device with an internet connection.

4. Technical Considerations (Updated)

Technologies Used

- Frontend:
  Console Interface (Java): Allows users to interact with the system via the command line. It runs locally (e.g., on localhost).
- Backend:
  Java: For business logic and managing users, forms, and responses.
  Python: For report generation and response processing.
- Database:
  JSON Files: Used for data persistence.
  users.json: Stores user information.
  forms.json: Stores created forms.
  responses.json: Stores form responses.
- Deployment Tools:
  Docker: For containerization of the application.
  Docker Compose: For service orchestration.

Diagramas UML:

Diagrama de Secuencia:

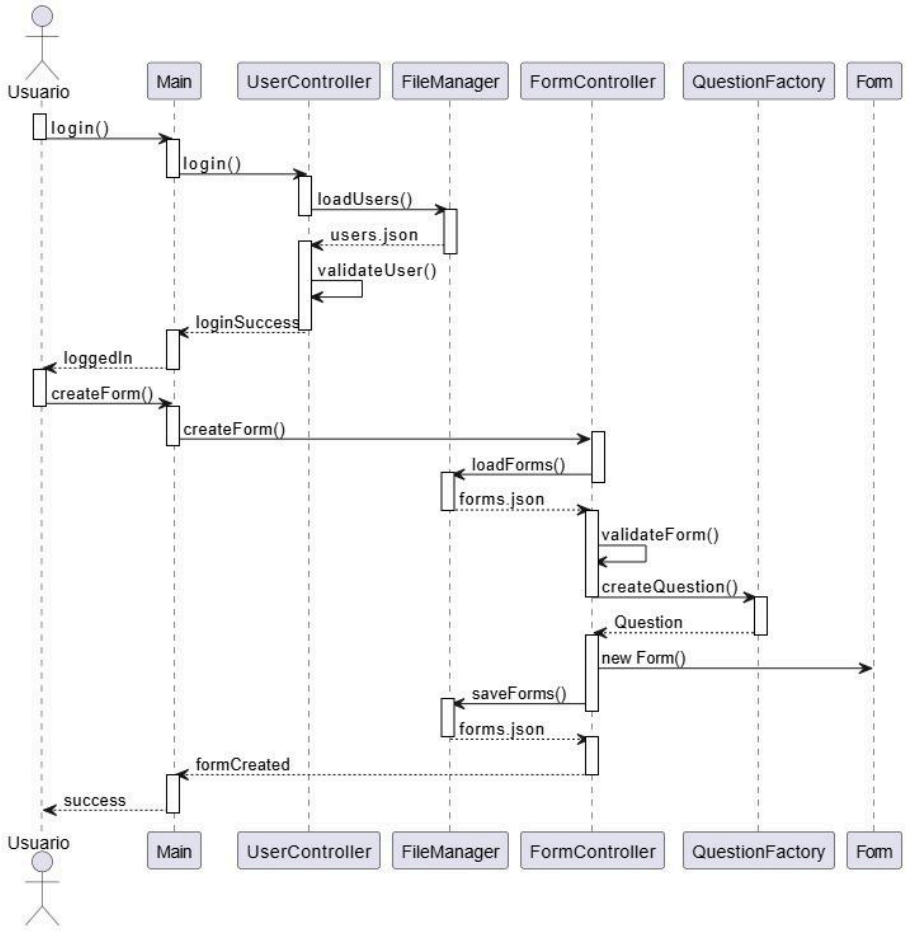
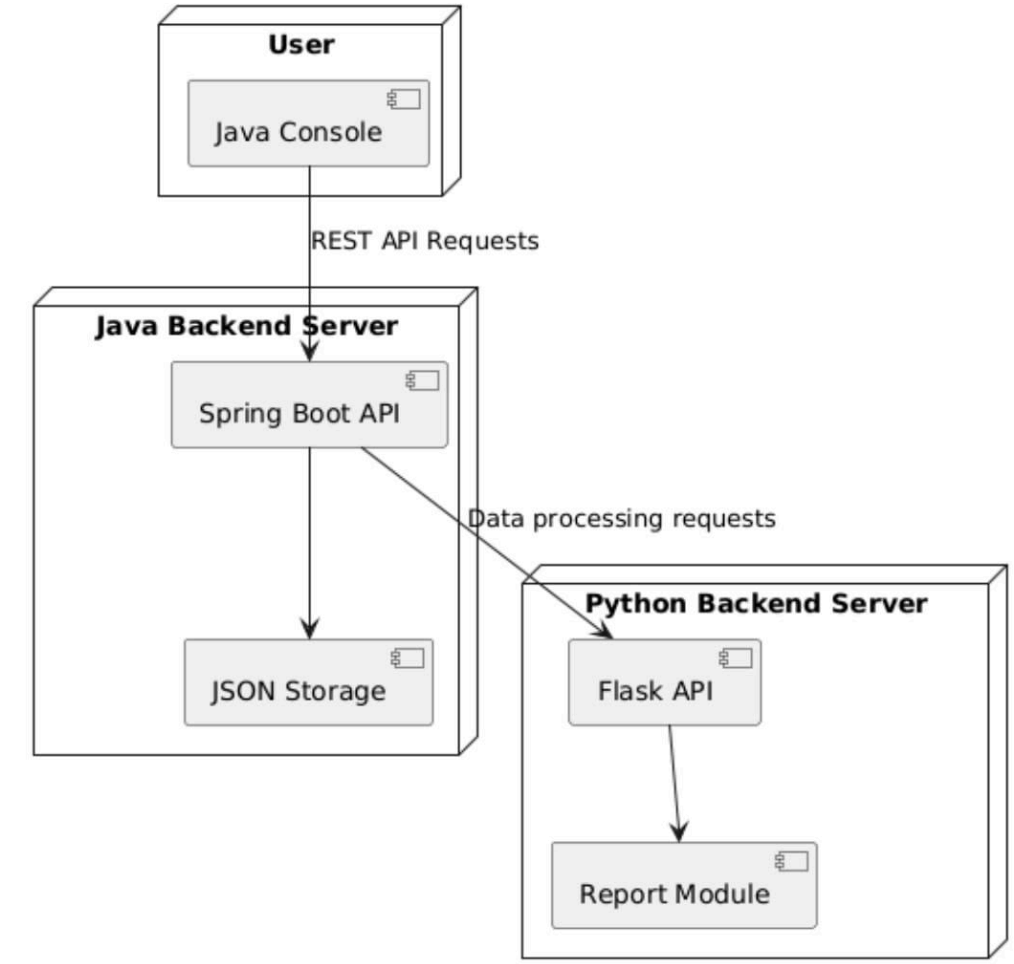


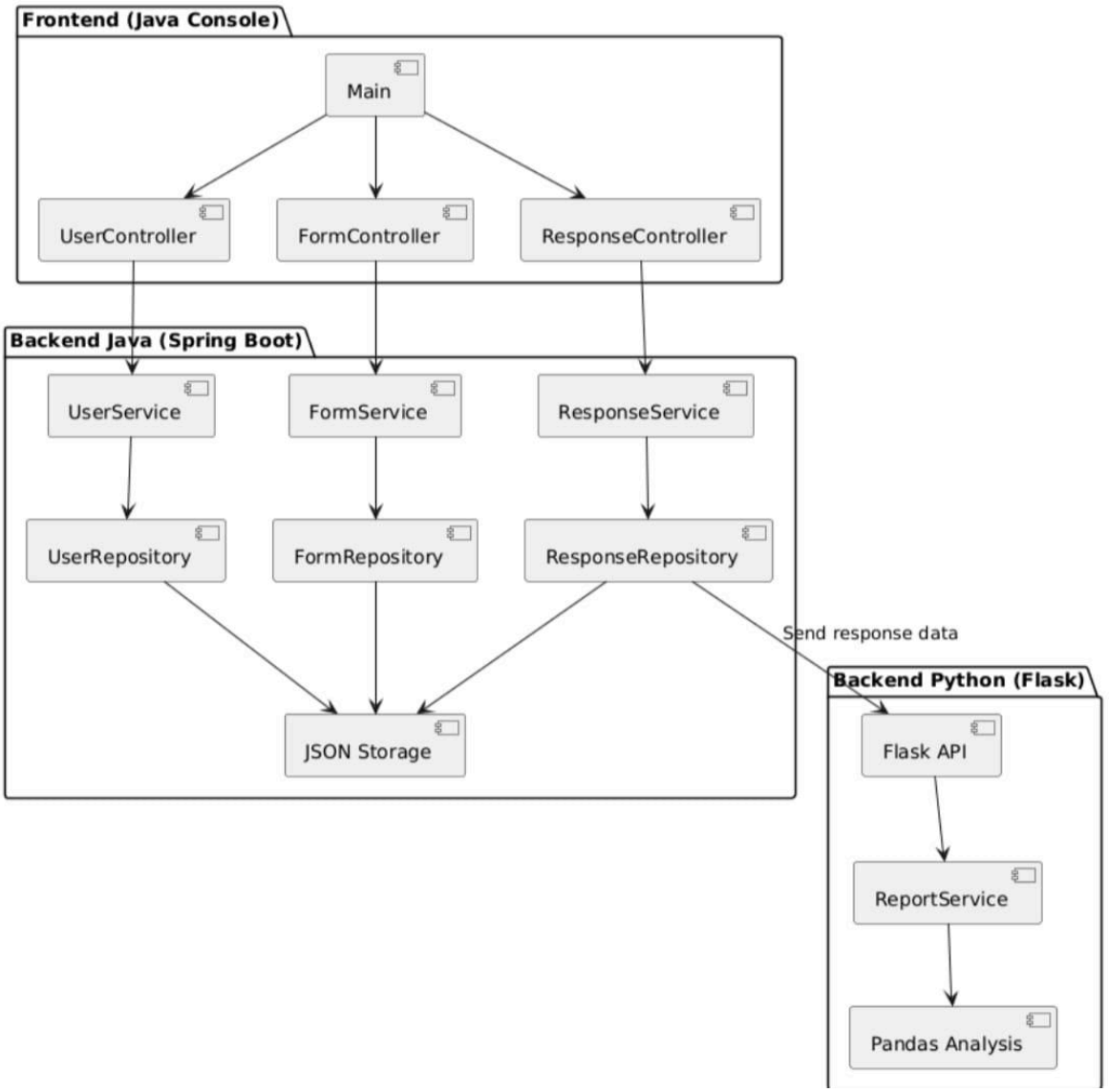
Diagrama de Actividades:

Architecture:

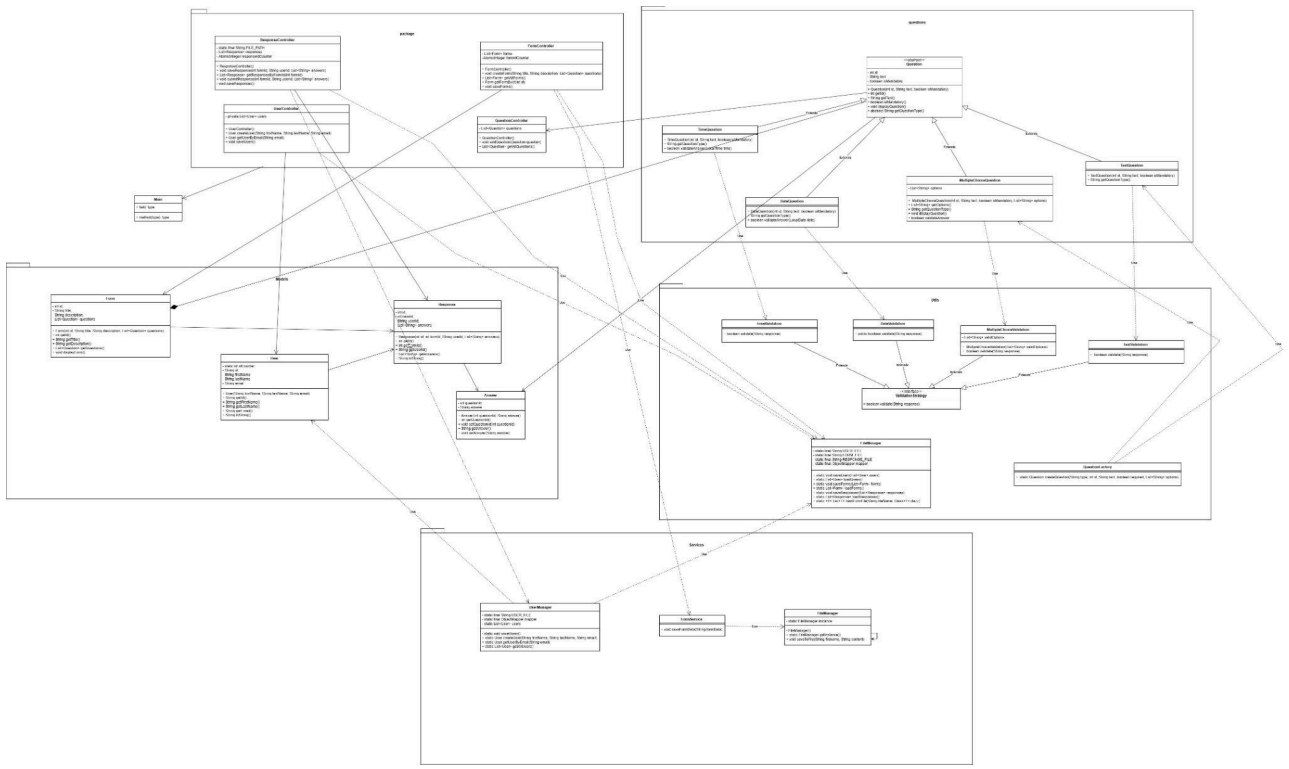
deployment diagram:

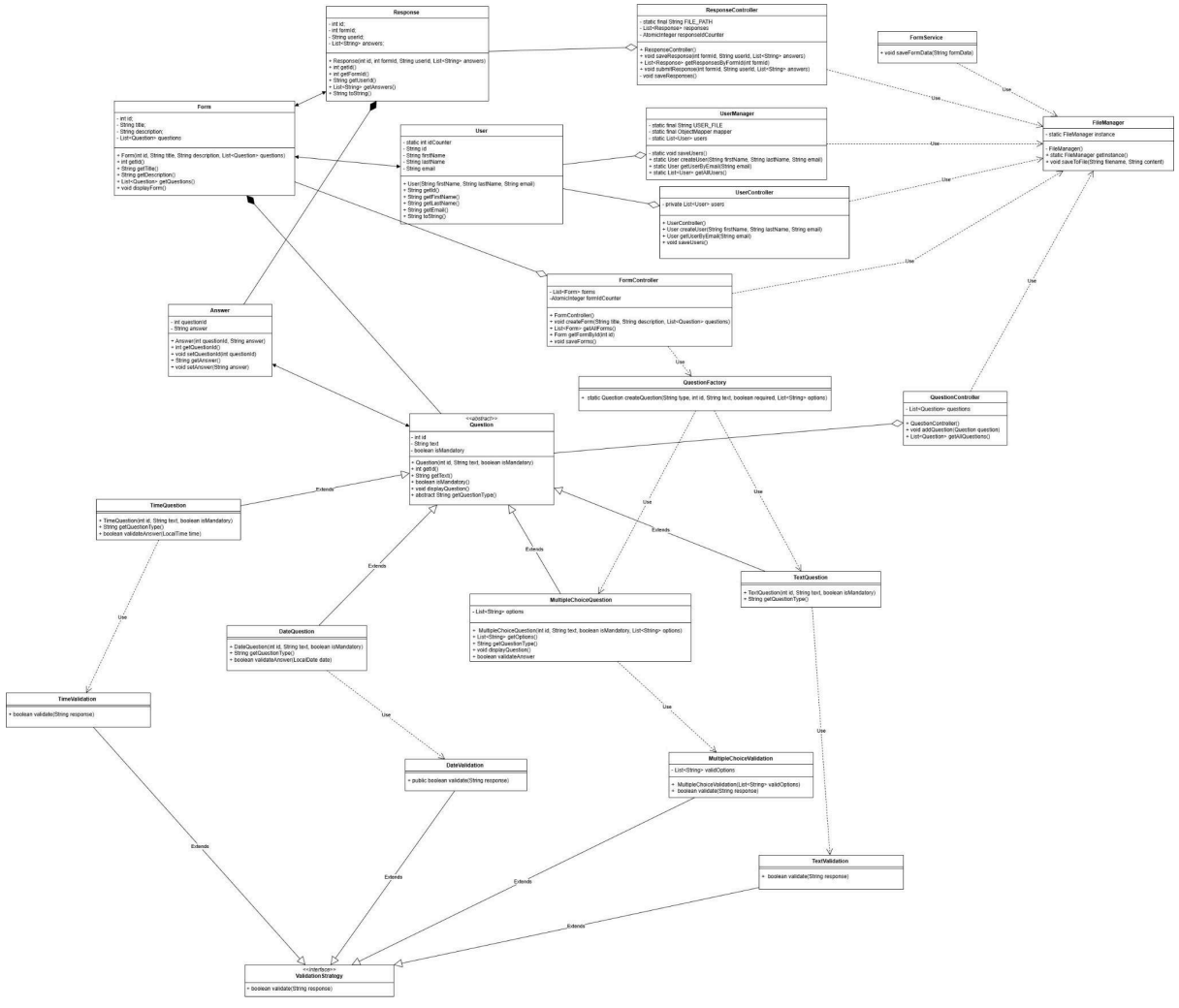


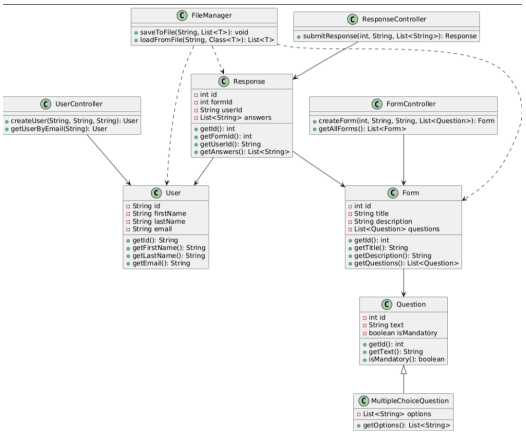
component diagram:



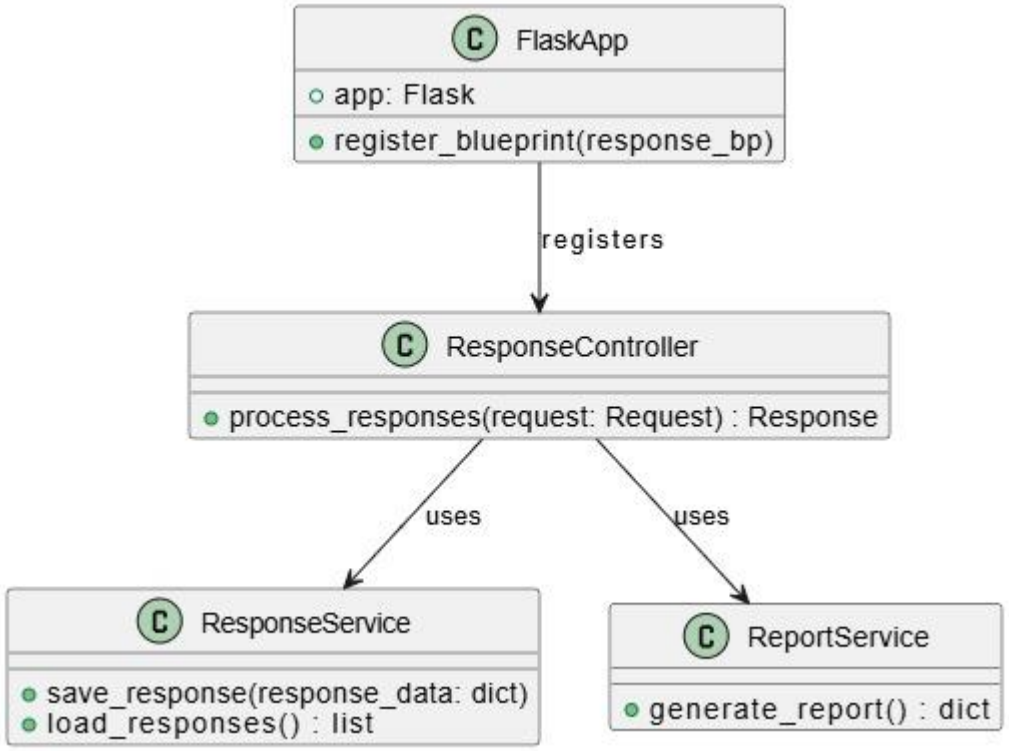
class diagram java:







class diagram python:



SOLID Principles Analysis in the Architecture The application architecture follows SOLID principles in its design, ensuring the code is maintainable, flexible, and scalable.

1. **Single Responsibility Principle (SRP):**
   Each class and module has a clear responsibility:
   - **Java:** UserController, FormController, and ResponseController manage users, forms, and responses, respectively, avoiding any class from handling too many responsibilities.
   - **Python:** response_service.py handles response storage, and report_service.py handles report generation.

2. **Open/Closed Principle (OCP):**
   The code is designed to allow extensions without modifying existing classes:
   - **Java:** Factory Method allows adding new question types without modifying the main logic.
   - **Python:** Strategy in report_service.py enables new analysis methods without altering the base code.
3. **Liskov Substitution Principle (LSP):**
   Subclasses can replace their base classes without affecting functionality:
   - **Java:** TextQuestion, DateQuestion, and others inherit from a base class Question, ensuring any question type can be used interchangeably.
4. **Interface Segregation Principle (ISP):**
   Each class only implements the interfaces it needs:
   - **Java:** UserService, FormService, and ResponseService avoid monolithic interfaces.
   - **Python:** Response and report services are separated to avoid unnecessary dependencies.
5. **Dependency Inversion Principle (DIP):**
   Abstractions are used instead of direct dependencies:
   - **Java:** Dependency Injection allows controllers to receive services without creating them internally.
   - **Python:** Flask Blueprint separates routes from business logic.

Design Pattern Implementation Analysis

Patterns Used

1. **Singleton (JSON File Manager in Java)**
   - Goal: Ensure there is only one instance of FileManager to avoid concurrency issues.
   - Implementation: The class has a static method that returns the same instance.
2. **Factory Method (Creating Questions in Java)**
   - Goal: Facilitate system extensibility by allowing the creation of different question types without modifying the base code.
   - Example: A method in QuestionFactory returns instances of TextQuestion, DateQuestion, etc.
3. **Strategy (Report Generation in Python)**
   - Goal: Allow different analysis strategies to be applied without modifying the base structure.
   - Example: generate_report() can use various methods depending on the desired type of analysis.

Patterns that were not used:

1. **Observer**
   Reason for non-use: This pattern is used when multiple components need to react to changes in an object (e.g., a real-time event), but in this project, there were no live notifications or dynamic events.

2. **Command**
   Reason for non-use: There was no need to encapsulate actions as independent objects, as the application flow was linear and based on console interactions.
3. **Decorator**
   Reason for non-use: There was no need to dynamically add functionality to the question or answer objects.
4. **Proxy**
   Reason for non-use: There were no remote accesses or complex access control mechanisms that would justify its implementation.

Best Programming Practices and Anti-Pattern Prevention

Best Practices Implemented

- Modularity: Clear separation between controllers, services, and storage.
- Dependency Injection: Prevents tight coupling and facilitates unit testing.
- Error Handling: Exception management in communication between Java and Python.
- Separation of Concerns: Each component serves a specific function.

Anti-Patterns Avoided

1. **God Object:**
   No excessive functionality centralized in one class. Responsibilities are split between controllers and services.
2. **Hardcoding:**
   Avoided by using configuration files instead of fixed values. JSON used for persistence instead of hardcoded strings.
3. **Spaghetti Code:**
   Prevented with a modular structure and SOLID principles. MVC used in Java and Blueprint in Flask to separate presentation logic.

Conclusions

The development of this online form platform allowed the exploration and application of various technologies and design principles that ensure a robust, modular, and scalable system. Below are the most relevant conclusions:

- **Application of SOLID Principles:**
  The system architecture was designed following SOLID principles, allowing for more maintainable and flexible code. Responsibility segregation, extension without modifying the base code, and dependency injection ensured a clean structure that is easy to scale.
- **Effective Use of Design Patterns:**
  The implementation of patterns like Singleton, Factory Method, and Strategy improved the system's modularity and extensibility. Singleton ensured efficient management of JSON files, Factory Method enabled flexible question creation, and Strategy facilitated report generation in Python without modifying the base logic.

- **Interoperability Between Technologies:**
  The combination of Java and Python leveraged the strengths of both languages: Java for structured data management and REST APIs, and Python for data processing and report generation. Communication between both services was designed to be efficient and scalable.
- **Best Programming Practices and Anti-Pattern Prevention:**
  Strategies like dependency injection, responsibility separation, and using JSON for data persistence were implemented, avoiding anti-patterns like spaghetti code and god objects. This resulted in more modular, maintainable code.

In conclusion, the project successfully met the initial goals, providing a functional, well-structured platform with a design based on best practices and software patterns. Its architecture allows for future improvements and ensures efficient maintenance over time.