Main goal: In order to facilitate efficient communication between users and a movie information service, I have developed a robust REST API application. This application serves as the backbone for seamless interactions, ensuring that users can easily retrieve movie information based on their preferences.

Key Technologies:

Flask: Flask is a lightweight and flexible Python web framework that simplifies the process of building web applications. It provides the foundation for creating the backend of our application.

Elasticsearch: I've utilized Elasticsearch, a distributed search and analytics engine, to efficiently store and retrieve movie data. This allows for quick and flexible searches based on user queries.

Pandas: Pandas is a powerful data manipulation library in Python. I've used it to handle and transform data, particularly to convert Elasticsearch search results into a user-friendly HTML table.

Development Process:

Setting Up the Backend (Flask and Elasticsearch): The development begins with creating the backend using Flask. This involves setting up the server, defining routes, and establishing a connection to the Elasticsearch cluster.

Indexing Movie Data: To populate Elasticsearch with movie information, I read data from a CSV file and index it into Elasticsearch. This ensures that the movie data is organized and easily searchable.

Implementing RESTful Endpoints: Flask is used to create RESTful endpoints that handle various operations, such as retrieving movie details. The **/load\_data** endpoint, for example, fetches movie data from Elasticsearch based on user-specified criteria.

Testing with Postman: Rigorous testing with Postman is performed to ensure that the API functions as expected. This step helps identify and address any issues in the backend logic.

Benefits:

Efficient Data Retrieval: Users can easily access movie information through a user-friendly interface, with Elasticsearch enabling quick and flexible searches.

Structured Data: Elasticsearch, acting as the database, ensures that movie data is stored in a structured manner, facilitating easy retrieval and management.

Rigorous Testing: Postman is utilized for testing, guaranteeing that the API operates reliably and minimizing the chances of errors.

User-Friendly Interface: The integration of Flask, Elasticsearch, and Pandas provides a modern and intuitive interface for users, enhancing their overall experience.

SECOND PART

Entities: In the backend, Python classes representing movie entities are created, such as Movie. These classes contain fields corresponding to properties like title, release year, and genre.

Repositories: Although not explicitly defined in the Flask application (as it follows a more lightweight approach), the indexing process into Elasticsearch serves the purpose of a repository. Elasticsearch handles the storage and retrieval of movie data.

Service Layer: The logic to retrieve and transform data from Elasticsearch is encapsulated in the Flask application. This corresponds to the service layer, ensuring separation of concerns and maintainability.

Controller Layer: Flask routes, such as **/load\_data**, act as the controller layer, handling HTTP requests and calling the appropriate logic to fetch movie data.

Request and Response DTOs: While not explicitly implemented in this context, the transformation of data from Elasticsearch into a Pandas DataFrame serves a similar purpose. This step ensures smooth data exchange between the client and the server.

Exception Handling: Flask inherently handles exceptions such as 404 (resource not found) when Elasticsearch queries do not yield results. This provides a graceful way to manage exceptional scenarios.

Security: Given the nature of the application and its focus on retrieving public movie information, advanced security features are not explicitly implemented. Flask provides basic security, and Elasticsearch access can be controlled through its settings.

Documentation: Flask's simplicity and self-descriptive nature make it easy to understand, but additional documentation using tools like Swagger can be considered for more extensive projects.

Dependency Management: Flask being a lightweight framework does not require extensive dependency management, but dependencies like Elasticsearch and Pandas are included based on project requirements.

Configuration: Configuration settings, such as Elasticsearch connection details, are set up in alignment with Flask conventions, ensuring easy configuration management.