Instagram social network clone: A Monolithic approach

Andrés Cerdas Padilla, Fabián Yesith Aguilar Jimenez

Facultad de ingeniería, Universidad Distrital Francisco José de Caldas, Bogotá, Colombia

acerdasp@udistrital.edu.co
fyaguilarj@udistrital.edu.co

ABSTRACT— This article presents the development of a simplified Instagram clone using a monolithic object-based backend with Python, FastAPI and Django. An argument is made in favor of object-oriented design to overcome the scalability and maintainability limitations associated with traditional monolithic architectures. The document describes the architecture of the system, detailing the layers, components and technical decisions made. Finally, the results obtained are discussed, including the implemented functionality and the advantages of the object-oriented approach in this project.

I. INTRODUCCIÓN

Visual social media has revolutionized the way we communicate and interact online. Platforms like Instagram allow users to share photos and videos, apply creative filters, follow other users, and build communities around common interests. While there are several previous solutions for sharing photos, replicating the basic functionality of Instagram with a rapid and efficient development approach presents an interesting challenge.

In this context, monolithic architectures offer simplicity and ease of initial development, making them an attractive option for small-scale projects or those with tight deadlines. However, as the application grows in complexity and the user base expands, monolithic architectures can present scalability and maintainability limitations.

Object-oriented design (OOD) emerges as a strategy to overcome these limitations. This programming paradigm promotes modularity, code reuse and the encapsulation of business logic, which allows building more robust systems that are adaptable to future changes [1].

In this work, we propose the development of a simplified Instagram clone using a monolithic object-based backend with Python, FastAPI and Django technologies. The OOD will be the cornerstone of the system architecture, allowing the code to be structured in a modular way, encapsulating the business logic and facilitating long-term maintenance.

The choice of Python as a programming language is based on its versatility, ease of learning, and broad community of developers. FastAPI, a lightweight and performant framework for creating RESTful APIs, will be used to expose backend services to the web interface. Django, a robust framework for high-level web development, will provide the basis for the implementation of business logic and interaction with the database.

The OOD will be applied in the backend business logic layer, where classes will be defined to represent the domain entities (users, posts, followers, etc.). These classes will encapsulate the logic associated with each entity, such as registering users, uploading posts, and managing interactions between users.

The modularity inherent in OOD will allow the system to be divided into independent and reusable components, facilitating the development and maintenance of the code as the application grows in complexity. Encapsulating business logic within classes will allow better organization of the code and hide implementation details, improving its readability and maintainability.

Inheritance, a fundamental principle of OOD, will allow the creation of specialized classes that inherit attributes and behavior from base classes, promoting code reuse and reducing duplication of effort. Polymorphism, another pillar of the OOD, will allow different classes to respond to the same message in diverse ways, providing flexibility to the system and facilitating adaptation to new requirements.

The implementation of the OOD in this project will not only allow the code to be structured in a modular and reusable way, but will also lay the foundations for the future scalability of the system. The modularity inherent to the OOD will make it easier to incorporate new functionality and adapt to a more complex environment without the need to rewrite large portions of the code.

In later stages, migration of the backend to a microservices architecture could be considered, taking advantage of the modularity already implemented with classes and objects. Microservices will allow the workload to be distributed between different independent services, improving the scalability and responsiveness of the system.

II. METHOD AND MATERIALS:

The system is made up of two main components:

1. Frontend: A simple web frontend developed with HTML, CSS and Javascript. This frontend allows users to register, log in, upload photos, follow other users, and like posts. A CSS framework such as Bootstrap was used to facilitate the design and organization of the elements in the interface. In addition, JavaScript libraries such as jQuery were implemented to improve the interactivity and dynamism of the page.

The frontend is structured into different pages and sections, each with a specific function. The home page shows the most recent posts of followed users, while the profile section allows users to edit their details, upload photos, and view their posts. Simple navigation mechanisms were implemented so that users can easily move between different sections of the site.

When applying the OOD, classes derived from Post can be created to represent specific types of posts, such as posts with filters or collaborative posts. This inheritance allows you to reuse the base class code and extend it to meet the specific needs of each publication type. The frontend stands out in these 3 sections:

- *Design:* A CSS framework such as Bootstrap was used to facilitate the design and organization of the elements in the interface. This allowed us to create a visually attractive and consistent interface on different devices.
- *Interactivity:* JavaScript libraries such as jQuery were implemented to improve the interactivity and dynamism of the page. This made it possible to create dynamic elements, such as drop-down menus and animations, that enrich the user experience.
- *Navigation:* Simple navigation mechanisms were implemented so that users can easily move between the different sections of the site. Menus, links and navigation bars were used so that users can quickly find the information they are looking for.

2. Backend: A monolithic Python-based backend that implements business logic and interaction with the database. This backend is divided into two layers:

- API Layer: Implemented with FastAPI, this layer exposes RESTful services for managing users, posts, followers, and likes. FastAPI makes it easy to create robust, documented APIs automatically.

Decorators were used to define the API endpoints and the marshmallow library was integrated for input and output data validation. For example, the /posts endpoint allows you to create new posts, validating the request data before storing it in the database.

- Business logic layer: Developed with Django, this layer implements the main functionality of the system. Django models are used to represent domain entities (users, posts, etc.) and views to handle API requests. The OOD is applied at this layer to define classes that encapsulate the business logic and interaction with the PostgreSQL database.

- Data Models: Models were created to represent the domain entities, such as User, Post, Follower and Like. These models define the attributes of each entity and the relationships between them. For example, the User model includes attributes such as name, email, password, and registration_date, while the Publication model includes attributes such as title, description, image, and publication_date.

- Django Views: Django views are used to handle API requests. Views are defined for each API endpoint, which are responsible for receiving the request, validating the input data, executing the corresponding business logic and sending the response to the client. For example, the create_post view receives data from a new post, validates the information, stores the post in the database, and generates a response with the identifier of the created post.

- Logic Encapsulation: OOD principles were applied to encapsulate business logic within classes. For example, the User class encapsulates methods for registering new users, updating profile data, and authenticating to the system. These methods access user information stored in the database, but hide connection details and SQL queries.

❖ OBJECT ORIENTED DESIGN (OOD)

Modularity: The system is divided into independent and reusable components, facilitating development and maintenance. For example, the Publication class is defined as an independent component that can be used to manage publications in different parts of the system. This allows code to be reused and avoids duplication of functionality. This reduces development time and makes it easier to add new functionality in the future.

Inheritance: Specialized classes are created that inherit attributes and behavior from base classes, promoting code reuse. For example, the SharedPost class inherits from the Post class and adds the logic to share posts with other users. This allows you to create new classes with specific functionality without having to rewrite existing code. This makes it easy to create class hierarchies that reflect the relationships between domain entities.

Polymorphism: Different classes can respond to the same message differently, providing flexibility to the system. For example, the Notification class can be implemented for different types of events, such as posting a new photo or receiving a like. This allows you to send notifications to users in a personalized way based on the type of event that has occurred. This makes the system more adaptable to changes in

requirements and facilitates the incorporation of new functionalities.

❖ IMPLEMENTATION WITH DJANGO AND FASTAPI

The business logic layer was implemented with Django, taking advantage of its data models, views and content management system (CMS). Models were created to represent the domain entities, such as User, Post, Follower, and Like. These models define the attributes of each entity and the relationships between them. For example, the User model includes attributes such as name, email, password, and registration_date, while the Publication model includes attributes such as title, description, image, and publication_date.

Django views are used to handle API requests. Views are defined for each API endpoint, which are responsible for receiving the request, validating the input data, executing the corresponding business logic and sending the response to the client. For example, the create_post view receives data from a new post, validates the information, stores the post in the database, and generates a response with the identifier of the created post.

OOD principles were applied to encapsulate business logic within classes. For example, the User class encapsulates methods for registering new users, updating profile data, and authenticating to the system. These methods access user information stored in the database, but hide connection details and SQL queries. This makes the code easier to understand and maintain, as developers do not need to know the implementation details of each entity.

The API layer was implemented with FastAPI, taking advantage of its ease of defining endpoints and automatically documenting the API. Decorators were used to define the API endpoints, assigning a friendly name and route for each action.

❖ Database

A PostgreSQL database was used to store system information. PostgreSQL is a robust and scalable relational database that offers features such as ACID transactions, complex SQL queries, and support for different data types.

Tables were created to represent domain entities such as users, posts, followers, and likes. The relationships between the entities were implemented using foreign keys. Used SQL queries to retrieve, insert, update and delete data from the database.

III. RESULTS:

The simplified Instagram clone can implement the following basic functionalities:

User Registration:

- Users can create new accounts by providing their full name, valid email address, and a secure password.
- Registration information is validated to ensure that the name is alphanumeric, the email address is correctly formatted, and the password meets minimum security requirements (length, complexity, etc.).
- An email confirmation system is implemented to verify the address provided and prevent fraudulent registrations.

Photo upload:

- Users can upload photos from their device to the platform, whether from a computer, mobile phone or tablet.
- A variety of common image formats are supported, such as JPEG, PNG, and GIF.
- Photos are resized and optimized to reduce their size and improve app performance.
- A unique file name is stored for each photo to avoid conflicts and facilitate retrieval.
- A unique identifier is associated with each photo with the account of the user who uploaded it.

Profile Edit:

- Users can edit their profile, including their full name, profile photo and personal description.
- Editing of the full name is subject to restrictions to avoid inappropriate or abusive names.
- It is allowed to upload a new profile photo, resizing and optimizing it as in the case of publications.
- The personal description can have a maximum length to avoid texts that are too long or irrelevant.
- Changes are saved to the database and reflected in the user's profile UI.

Viewing posts:

- Users can see posts from users they follow in their main feed.
- Posts are displayed in reverse chronological order, with the most recent post first.
- Each post displays the uploaded photo, the name of the user who uploaded it, the date it was posted, the number of likes, and a short description (optional).
- A pagination system is implemented to load posts incrementally as the user scrolls through the feed.

interact:

Users can like and comment on other users' posts to indicate that they liked them.

When you "like" a post, the "like" counter associated with it is incremented, just like when commenting.

The action of "liking" and commenting is recorded in the database, associating it with the user and the corresponding publication.

When viewing a post, the current number of likes is displayed and a button to like or remove it, depending on the user's current status with respect to that post.

User tracking:

- Users can follow other users to see their posts in their main feed.

- By following a user, a following relationship is created between the following user and the followed user.
- A user suggestion system is implemented to recommend users to follow other users with similar interests.

- User Search:
-
- Users can search for other users by full name or email address.
- The search is performed incrementally, displaying results in real time as the user types.
- Search results are displayed in a list sorted by relevance, considering factors such as name matching, number of followers, and recent user activity.
- The profile of each user found is allowed to be accessed from the search results.

❖ Performance and Scalability: In-Depth Analysis

The system was tested with a moderate number of users and demonstrated good performance for the basic functionalities implemented. Requests were processed quickly and the user interface responded smoothly.

However, it is important to recognize that monolithic architecture has long-term scalability limitations. As the number of users and the amount of data grow, system performance could be affected in the following ways:

- Increased response time: As the load on the server increases, the time required to process requests could also increase. This could lead to slow UI and a frustrating experience for users.
- Concurrency Limitations: Monolithic architecture may have difficulty handling large numbers of simultaneous users, especially if multiple requests are made at the same time. This could cause bottlenecks and affect system availability.
- Maintenance Complexity: As code grows and complexity increases, system maintenance becomes more challenging. Introducing new changes or fixing bugs can be a slower and error-prone process.

❖ Strategies to improve scalability:

While the monolithic architecture has scalability limitations, there are some strategies that can be implemented to improve the performance and capacity of the system as it grows:

- Code optimization: Review and optimize existing code to eliminate bottlenecks and improve processing efficiency.
- Cache implementation: Implement caching mechanisms to store frequently used data and reduce the number of database queries.
- Horizontal scaling: Add more servers to the system to distribute the load and increase processing capacity.
- Consider a microservices architecture: In the long term, migrating the system to a microservices architecture could be considered to break the application into smaller, independent, and scalable components.

❖ OOD Limitations:

While OOD offers benefits for code organization and maintenance, it also has some limitations:

Increased complexity: Implementing a well-structured object-oriented design can increase the initial complexity of the code, especially for developers less familiar with OOD principles.
Learning Curve: The learning curve for mastering OOD principles may be a little steeper than for more traditional programming approaches.

In general, the OOD proved to be a suitable choice for the development of the backend of the simplified Instagram clone, since it allowed the code to be structured in an organized, reusable and adaptable way to future changes. The benefits of encapsulation, modularity, inheritance, and polymorphism outweighed the initial limitations of complexity and learning curve.

IV. CONCLUSIONS:

The development of this simplified Instagram clone has been a journey of continuous learning about the benefits and challenges of DOO in the context of a complex web application. The successful implementation of the basic functionalities of a social network has demonstrated the viability of this approach, while the analysis of scalability limitations has opened the door to exploring strategies for long-term sustainable growth.

In short, the DOO has proven to be a valuable tool for structuring code in a modular, reusable and adaptable way, laying the foundation for efficient development and simplified maintenance. However, it is important to recognize the limitations of the monolithic architecture and consider scalability strategies as the system grows in complexity. Migrating to a microservices architecture could be a viable option in the future to ensure long-term scalability and performance.

This project has served as a springboard to explore the possibilities of OD in the development of complex web applications, providing valuable lessons that can be applied to future projects. The combination of a well-structured architecture with appropriate scalability strategies will allow the creation of robust, scalable and adaptable systems to the changing needs of the digital world.

BIBLIOGRAPHY

[1] Booch, G. (2010). Object-oriented analysis and design with applications (3rd ed.). Addison-Wesley.

SILVA MORAN, Darío; MERCERAT, Bárbara. Construyendo aplicaciones web con una metodología de diseño orientada a objetos. Revista Colombiana de Computación, 2001, vol. 2.

RODRÍGUEZ, JIPJIP Ángela Indira; PADILLA, Jackelyne Ivone; PARRA, Hugo Alexander. Arquitectura basada en micro-servicios para aplicaciones web. Tecnología Investigación y Academia Universidad Distrital, 2020, vol. 7, no 2, p. 10.

VELASCO, Jackelyne Ivone Padilla; RUIZ, Angela Indira Rodríguez; ALVIRA, Hugo Alexander Parra. Arquitectura basada en microservicios para aplicaciones web. Tecnología Investigación y Academia, 2019, vol. 7, no 2, p. 12-20.

VIDAL-SILVA, Cristian L., et al. Experiencia académica en desarrollo rápido de sistemas de información web con Python y Django. Formación universitaria, 2021, vol. 14, no 5, p. 85-94.