Master's in Computer Engineering

Curricular Unit: **Tópicos Avançados de Bases de Dados**

Teacher: **Rosa Maria Do Nascimento Da Silva Reis**

**E-Commerce Web Application Development**

1161424 – Elmer Graça

Porto, December 12th 2024

**Summary:** The goal of the project is to create an e-commerce platform for the XPTOISEP firm that is safe, scalable, and easy to use. To effectively handle structured and semi-structured data, the project combines cutting-edge database paradigms, using relational (Oracle) and non-relational (MongoDB) databases. The foundation of the system's architecture is made up of Python technology. Dynamic product catalog browsing, a strong checkout procedure, customer loyalty programs, and smooth payment gateway integrations with MB Way and credit card services are some of the main features. To guarantee operational dependability even during periods of high demand, the platform also places a strong emphasis on non-functional needs including scalability, performance, security, and availability.

**Keywords: Database, Oracle, MongoDB, Flask, Backend, Frontend**.

Index

Contents

[1. Introduction 7](#_Toc184911697)

[2. Theoretical Framework 8](#_Toc184911698)

[3. Methodology 9](#_Toc184911699)

[4. Architecture 10](#_Toc184911700)

[5. Backend 11](#_Toc184911701)

[6. Frontend 14](#_Toc184911702)

[7. Databases 15](#_Toc184911703)

[a. MongoDB 15](#_Toc184911704)

[b. Oracle 15](#_Toc184911705)

[8. Deployment Strategies 16](#_Toc184911706)

[9. Final Considerations 17](#_Toc184911707)

[Bibliography 18](#_Toc184911708)

[10. Appendices 19](#_Toc184911709)

Tables Index

Figure Index

# Introduction

The present work was carried out within the scope of the curricular unit "Advance Database Topics" of the Master’s in Data Engineering, taught at the Instituto Superior de Engenharia do Porto by professors ..., during the academic year 2024/2025.

# Theoretical Framework

# Methodology

# Architecture

# Backend

Flask is a micro-framework for web application development that is built on Python. It follows the ideas of modularity and minimalism, giving developers the freedom to incorporate extra libraries and tools as needed while just offering the essential parts needed to create online applications. Flask's design philosophy makes it extremely versatile for building unique architectures.

Werkzeug serves as the foundational library for request and response handling in Flask, which is based on the WSGI (Web Server Gateway Interface) standard. Developers can easily incorporate backend data into frontend displays thanks to Jinja2, a templating engine for creating dynamic HTML. In order to provide exact control over application endpoints, Flask's routing system uses decorators to bind URLs to certain Python functions.

Flask provided the framework for controlling user authentication, establishing database connections, and enabling front-end interaction in the XPTOISEP e-commerce platform. Flask-Login was used to create user authentication and session management, allowing for the safe handling of user roles and states. Database interactions made use of PyMongo for MongoDB operations and SQLAlchemy for Oracle connectivity, allowing for complicated queries and effective CRUD operations.

**Password Hashing and Security**

To manage passwords, Flask facilitates the integration of secure libraries. Passwords were hashed for this project using the generate\_password\_hash and check\_password\_hash methods of the werkzeug.security package. Sensitive user data is safely saved thanks to these hashing algorithms, which guard against possible breaches. Other security features like encryption, CSRF protection, and secure sessions were implemented as well as they are essential for e-commerce apps.

**User Management**

Flask offers powerful user management tools with extensions such as Flask-Login. User authentication, including login sessions, role-based access, and session persistence, was managed with this plugin. Both consumer and staff models, each with unique characteristics and functions, were incorporated into the implementation. Using secure session cookies to preserve user state across requests and defining user models and role-based behavior were made simple by Flask's modular design.

**Database Connections**

For this project, which uses Oracle for structured data and MongoDB for unstructured data, Flask is a great option because it easily interfaces with both relational and non-relational databases:

* **Oracle**: Oracle database interactions were managed using SQLAlchemy, an ORM (Object-Relational Mapping) framework. By mapping database tables to Python classes, it streamlines SQL queries and makes it possible to effectively get Oracle tables data access.
* **MongoDB**: MongoDB was accessed using the PyMongo library. It offers a straightforward interface for carrying out tasks to access, create and update data on the collections as well running analytics aggregate queries.

**Frontend Interaction** Flask's seamless frontend integration was made possible by its ability to deliver dynamic templates via the Jinja2 templating engine. Dynamic HTML pages (such as user dashboards, product details, and cart pages) were populated using backend logic, such as retrieving data from the database. Additionally, Flask offers URL routing, which gives you fine-grained control over endpoints and guarantees that resource requests—such as product searches or cart interactions—are effectively sent to the appropriate backend logic.

**Modular Development with Blueprints** The backend could be separated into modules for admin functions, cart management, and user authentication thanks to Flask's blueprints. Because each component functions independently while yet being a part of the broader application, this modularity guarantees scalable and maintainable code.

**Libraries and Extensions Used**

1. **User Management**
   * Flask-Login for session management and role-based access control.
2. **Database Management**
   * SQLAlchemy for Oracle database integration.
   * PyMongo for MongoDB interaction.
3. **Frontend Interaction**
   * Jinja2 for rendering templates dynamically.
   * Flask’s routing mechanism to connect frontend actions with backend logic.
4. **Error Handling**
   * Custom error pages using Flask’s @app.errorhandler.
5. **Modularity**
   * Blueprints to segment features such as authentication, cart management, and admin tools.

To guarantee modularity, maintainability, and scalability, the XPTOISEP e-commerce platform's backend is separated into nine files, each of which is devoted to a certain function. In keeping with the platform's primary functions, this division comprises files for admin functions, database models, user information, cart management, authentication, and more. The development process is streamlined by separating these issues, enabling the independent development, testing, and updating of different areas. In addition to improving code readability, this modular approach makes debugging and integrating new features easier.

**main.py:**

The Flask application is initialized and launched via the main.py file, which acts as the entry point for the XPTOISEP e-commerce platform. To guarantee that all configurations, database connections, and extensions are initialized prior to the application starting, it imports the create\_app method from the \_\_init\_\_.py file. Additionally, the file specifies the execution environment, which allows the application to operate in debug mode while it is being developed for simpler error tracking and testing. Main.py simplifies deployment and guarantees a uniform initialization procedure across many contexts by centralizing the application startup.

**\_\_inti\_\_.py : Application Initialization and Configuration**

The \_\_init\_\_.py file serves as the initialization point for the Flask application, centralizing configurations and establishing the foundational components required for the e-commerce platform. It includes the following functionalities:

1. **Application Factory**:
   * Implements the create\_app() function, adhering to the Flask application factory pattern. This approach enables modular initialization, making it easier to configure and extend the application in different environments (e.g., development, production).
   * Configures the Flask instance with essential settings, such as the SECRET\_KEY for session security, ensuring the application’s cryptographic operations are protected against unauthorized access.
2. **Database Configuration**:
   * Sets up the SQLAlchemy ORM for seamless integration with Oracle databases, using connection credentials (SQLALCHEMY\_DATABASE\_URI) to establish the link to the Oracle server.
   * Configures MongoDB using PyMongo via the mongoengine.connect() function, enabling interaction with non-relational data such as product reviews and session analytics.
3. **Login Management**:
   * Integrates Flask-Login to manage user authentication and sessions.
   * Defines a user loader function to fetch user details from the database during authentication, supporting both customers and employees by querying their respective models.
4. **Error Handling**:
   * Defines a custom error handler for HTTP 404 errors, rendering a 404.html template when users attempt to access non-existent resources.
5. **Extension Initialization**:
   * Initializes all required Flask extensions, including SQLAlchemy for relational database management and Flask-Login for user authentication.
6. **Blueprint Registration**:
   * While not included directly in this file, \_\_init\_\_.py provides the structure for registering modular blueprints, such as those for authentication, cart management, and admin functionalities, ensuring a clean and organized application layout.

This file establishes the backbone of the application by coordinating core configurations and dependencies. Its role as an entry point ensures that database connections, security settings, and other extensions are correctly initialized before any routes or business logic are executed. By adhering to Flask’s factory pattern, the application achieves flexibility and scalability, accommodating future changes or additions with minimal disruption.

**Auth.py: Authentication and Form Management**

The auth.py and forms.py files collectively handle user authentication, form validation, and secure session management, aligning with the role-based user system of the XPTOISEP e-commerce platform. These files implement the logic for user registration, login, and session persistence.

**Authentication Logic (auth.py)**

The auth.py file defines routes and associated logic for user registration (/sign-up), login (/login), and logout (/logout). Utilizing Flask’s Blueprint feature, the authentication module is encapsulated, ensuring modularity and separation from other application functionalities. The Flask-Login extension facilitates user session management by providing decorators such as @login\_required to restrict access to authenticated users and login\_user() for secure session creation. The logout\_user() method ensures secure termination of sessions, preventing unauthorized access to resources.

The user registration logic incorporates a mechanism for generating unique account identifiers using user data (e.g., partial NIF and username) and assigns default statuses (e.g., "new"). Passwords are securely hashed using the generate\_password\_hash method from werkzeug.security, and user details are stored in the Oracle database using SQLAlchemy. Validation is performed to ensure data integrity, such as matching passwords and avoiding duplicate emails.

**Form Validation (forms.py)**

The forms.py file defines the form structures and validation rules for the registration and login processes using the Flask-WTF extension. Forms such as SignUpForm and LoginForm are implemented, employing fields like StringField and PasswordField to collect user inputs. Validation criteria, such as data presence (DataRequired), email format (Email), and length constraints (Length), are applied to ensure robust input handling. For example, the SignUpForm includes additional fields for address, postal code, and GDPR compliance acknowledgment, reflecting the e-commerce domain’s regulatory requirements.

These forms are rendered dynamically in the associated HTML templates, ensuring a seamless user experience. Errors arising from invalid submissions are handled gracefully, with feedback provided to users for corrective actions.

**Integration and Functionality**

The integration of auth.py and forms.py allows for a cohesive authentication system where user inputs are validated against predefined criteria, securely processed, and stored in the database. The modular design supports future enhancements, such as adding two-factor authentication or extending user roles, without affecting the broader application structure. Together, these files provide a robust foundation for managing user accounts and sessions within the platform.

**Models.py: Database Schema and ORM Integration**

The models.py file defines the database schema and provides an interface for managing the relational and non-relational data used in the XPTOISEP e-commerce platform. It utilizes SQLAlchemy for interacting with the Oracle database and MongoEngine for MongoDB integration, encapsulating data definitions and their associated behaviors.

**Integration of SQLAlchemy and MongoEngine**

By combining SQLAlchemy and MongoEngine, the models.py file supports a hybrid database architecture, leveraging the strengths of both relational and non-relational databases. Relationships between models are explicitly defined in Oracle, ensuring referential integrity, while MongoDB enables flexible storage for unstructured or dynamic data.

**Modularity and Scalability**

Each model is encapsulated with methods and constraints specific to its functionality, enabling clean and maintainable code. The use of ORM abstractions allows for database queries and operations to be expressed in Python, reducing the complexity of SQL statements and enabling seamless database interaction.

The structure defined in models.py ensures the platform can scale efficiently, handling complex e-commerce operations while maintaining data integrity and flexibility.

**Views.py**

The views.py file encapsulates the core application logic and URL routing for the XPTOISEP e-commerce platform. It manages the interaction between the frontend and backend, providing endpoints to serve dynamic content and handle user requests.

**Functionality**

1. **Home Page and Navigation**:
   * The home route (/) fetches data such as featured products from the database and renders the main homepage using the home.html template. It uses a randomized selection of products to enhance user engagement.
2. **Product Search**:
   * The search route handles product search queries submitted by users, retrieving matching items from the MongoDB collection. Results are passed to a search results page, dynamically rendered using the Jinja2 templating engine.
3. **Product Details**:
   * The product/<int:item\_id> route retrieves detailed information about a specific product, such as technical specifications and pricing history. Data is fetched from MongoDB and Oracle, demonstrating the hybrid database architecture.
4. **Terms and Conditions**:
   * The termos route dynamically fetches the latest version of the platform’s terms and conditions from MongoDB and renders it to the user.
5. **Modularity and Extensibility**:
   * The views.py file employs Flask's routing system to cleanly map URL endpoints to Python functions. Each route is self-contained and interacts with specific models or services, ensuring modularity and maintainability.

**Integration**

The views.py file connects the backend logic to the frontend templates, enabling seamless user interactions. By managing data retrieval, processing, and presentation, it acts as the bridge between user requests and the platform’s databases. This file ensures a cohesive user experience by dynamically serving content based on real-time interactions.

**Admin.py: Administrative Functionality and Analytics**

The admin.py file is dedicated to managing administrative operations and providing insights into the XPTOISEP e-commerce platform's performance. It uses Flask’s Blueprint to encapsulate administrative routes, ensuring modularity and separation of concerns from user-facing features.

**Functionality**

1. **Administrative Dashboard**:
   * The admin\_manager route provides an interface for administrators to access high-level operations and analytics via the adminManager.html template.
2. **Data Insights and Queries**:
   * The file defines specialized routes for extracting business insights, such as:
     + **Voucher Analytics** (query\_us13): Uses MongoDB's aggregation framework to identify top customers based on voucher usage. Results are enriched with customer details retrieved from Oracle.
     + **Order Efficiency Analysis** (query\_us14): Analyzes orders with expedited preparation times within a specific date range, using MongoDB for filtering and Oracle for customer enrichment.
3. **Delivery Order Management**:
   * The deliveryOrderManager route allows administrative staff to track and manage delivery orders. It integrates data from MongoDB to provide real-time updates on shipping statuses.
4. **Error Handling and Logging**:
   * Administrative routes are designed to handle edge cases gracefully, such as missing data or invalid inputs. Logging is implemented to aid in debugging and monitoring.

**Cart.py: Shopping Cart and Checkout Management**

The cart.py file manages the shopping cart and checkout functionalities of the XPTOISEP e-commerce platform. It employs Flask’s Blueprint to encapsulate routes specific to cart-related operations, ensuring modular and maintainable code.

**Functionality**

1. **Cart View**:
   * The view\_cart route retrieves the current user's cart from MongoDB. If the cart exists, it calculates the total price by aggregating item prices, discounts, and quantities. The cart details are then rendered using the cart.html template.
2. **Adding Items to Cart**:
   * The add\_to\_cart route allows users to add items to their cart by providing the product ID and customer ID. Items are stored in MongoDB for flexibility in managing hierarchical or nested data, such as product attributes and quantities.
3. **Checkout**:
   * The checkout\_cart route manages the checkout process. It retrieves the cart’s contents, calculates totals (including taxes and shipping), and processes voucher codes if submitted. The logic applies valid vouchers to provide discounts and dynamically updates the cart total.
4. **Utility Functions**:
   * **calculate\_totals**: Computes the total price of items in the cart, including taxes (fixed at 23%) and shipping costs (calculated randomly for simulation purposes).
   * **apply\_voucher**: Validates and applies discount vouchers by querying Oracle’s database for voucher details. It adjusts the final total to reflect the applied discount.
5. **Error Handling**:
   * The routes handle scenarios such as empty carts or invalid voucher codes gracefully, providing user feedback through templates or JSON responses.

**Integration**

The cart.py file bridges MongoDB’s flexible data storage for dynamic cart management with Oracle’s relational database for validating vouchers and retrieving product details. This dual-database approach ensures efficient handling of cart operations and checkout processing while maintaining data integrity and performance. By modularizing cart-specific logic, this file supports scalability and easy integration of additional features, such as loyalty points or advanced discount rules.

**Customer.py: Customer Profile and Order Management**

The customer.py file focuses on managing customer-specific features, including profile views, active orders, and order reviews. It uses Flask’s Blueprint for modular route management, separating customer-related functionalities from other application components.

**Functionality**

1. **Customer Profile**:
   * The profile\_view route retrieves and displays the customer’s profile details using the userDetails.html template. This view is secured with the @login\_required decorator, ensuring only authenticated users can access their profile.
2. **Active Orders**:
   * The active\_orders route retrieves orders associated with the current customer where the shipping status is not marked as "Delivered." It queries the MongoDB Orders collection for real-time order data and integrates product details from Oracle. The results are displayed using the activeOrders.html template, enriched with item names and shipping information.
3. **Order Reviews**:
   * The orders\_reviews route fetches orders marked as "Delivered" to allow customers to leave feedback. MongoDB stores hierarchical order details, and the logic incorporates Oracle queries to provide product names and other structured information. Customers can rate items and leave comments through the interface, supporting dynamic review submission.
4. **Authorization and Security**:
   * All routes ensure that the customer ID in the request matches the currently authenticated user (current\_user.customer\_id), preventing unauthorized access to other customers’ data. Errors resulting from mismatches are handled with appropriate HTTP status codes (e.g., 403 Unauthorized).

**Integration**

The customer.py file integrates MongoDB for dynamic, session-based data such as order details and Oracle for structured data like customer and product information. This hybrid approach ensures that the customer experience is both dynamic and reliable, accommodating real-time updates for active orders while maintaining data integrity for profile and review functionalities. The modular design supports scalability and easy integration of new customer-related features.

**Features Not implemented**

**Improvements**

**Implement Advanced Logging and Monitoring**

Currently, basic logging is in place, but integrating advanced logging frameworks like Flask-Logging or external monitoring services such as Sentry can significantly enhance error tracking and debugging. By capturing critical application metrics, request logs, and database query performance, developers can proactively identify and resolve issues. This improvement ensures better visibility into the system’s health and supports scaling with minimal disruptions.

**Enhance Security Practices**

While password hashing and secure session management are implemented, incorporating additional security layers, such as rate-limiting for API endpoints (Flask-Limiter) and automated detection of suspicious activities, can improve the system's resilience. Moreover, enabling HTTPS for secure data transmission and integrating libraries like Flask-Talisman for content security policies will bolster the platform’s overall security posture.

**Optimize Database Queries**

The current database interactions can benefit from query optimization strategies, such as caching frequently accessed data (e.g., product catalogs) using tools like Flask-Caching. Indexing critical fields in Oracle and MongoDB can significantly reduce query execution times. This improvement will enhance the performance of user-facing features like product search and order management, especially under high traffic.

**Improve Modularization and Code Organization**

While the backend is divided into multiple files, further modularization can be achieved by grouping related functionality into packages. For instance, splitting auth.py into separate files for registration and login logic enhances maintainability. Introducing a clear folder hierarchy, such as services for database operations and utils for shared functionality, improves code readability and scalability.

**Implement a Unit and Integration Testing Framework**

Introducing comprehensive testing with frameworks like Pytest or Unittest ensures reliability and reduces the likelihood of bugs in production. Unit tests for individual functions, such as voucher validation, and integration tests for critical workflows, such as checkout, will enhance the platform's robustness. Automated testing in the CI/CD pipeline ensures smooth deployment and faster feedback during development.

# Frontend

# Databases

# MongoDB

# Oracle

# Deployment Strategies

To guarantee security, scalability, and dependability, a systematic method is needed while deploying the XPTOISEP e-commerce to production. The following strategies cover a range of deployment topics, from server configuration to monitoring.

1. **Web Server and Application Server Setup**: In order to deploy the platform and effectively handle concurrent requests, a production-ready WSGI server, such Gunicorn or uWSGI, is needed. Request forwarding, HTTPS termination, and static file delivery are all managed by a reverse proxy, like Nginx or Apache. This setup facilitates load balancing and enhances performance.
2. **Environment Configuration**: Private data, like API keys and database credentials, should be safely maintained via services like AWS Secrets Manager or stored in environment variables. To ensure security and consistency in functioning, the application should use different configuration files for the development, staging, and production environments.
3. **Database Management**: While MongoDB can use MongoDB Atlas, Oracle databases can be scaled using services like AWS RDS. To ensure database consistency when updating applications, schema modifications should be implemented using tools such as Flask-Migrate. Replication and failover techniques should be used while configuring high availability.
4. **Security Improvements**: Secure data transmission is ensured by deploying the platform via HTTPS with SSL/TLS encryption. Only the application server should have network access to databases, and Flask-Talisman should be used to enforce security headers like HSTS and Content Security Policy. Rate-limiting for APIs can further prevent potential exploitation.
5. **Scaling and Load Balancing**: Docker and Kubernetes enable horizontal scaling, which enables the deployment of numerous application instances. Traffic is then distributed via load balancers such as AWS ELB. The platform's performance can be further improved during periods of high demand by vertical scaling, which increases server resources in accordance with workload.
6. **Continuous Integration and Deployment**: By automating testing and deployment with GitHub Actions or comparable solutions, CI/CD pipelines guarantee the application's dependability throughout time. During updates, user access is maintained without interruption thanks to zero-downtime deployment techniques including blue-green deployments and rolling updates.
7. **Scaling and load balancing**: The deployment of many application instances is made possible via horizontal scaling, which is made possible by Docker and Kubernetes. After that, load balancers like AWS ELB are used to disperse the traffic. Vertical scaling, which raises server resources in line with workload, can further enhance the platform's performance during times of heavy demand.
8. **Continuous Integration and Deploymen**t: CI/CD pipelines ensure the application's reliability over time by automating testing and deployment using GitHub Actions or similar tools. Zero-downtime deployment strategies, such as rolling updates and blue-green deployments, ensure uninterrupted user access throughout changes.
9. **Backup and Disaster Recovery**: To guarantee data recovery in the event of a system failure, regular database backups and server snapshots must be planned. These steps save downtime and protect important data.
10. **Testing and Validation**: Performance bottlenecks under heavy traffic are found by load testing with tools like JMeter or Locust. By proactively detecting and addressing vulnerabilities, penetration testing guarantees the security of the production environment.

# Final Considerations

# 

# Appendices