**Secure E-Commerce Platform for Electronics and Gadgets**

***EECE 503M: Software Security***

**Prepared by:** Taline Slim, Ahmad Dimashkie and Karim Kobeissi

**Prepared for:** Professor Khaled Dassouki

Table of Contents

[I. Introduction: *Project overview and objectives* 3](#_Toc182771497)

[1.1. Project Goals 3](#_Toc182771498)

[1.2. Importance of Security in E-commerce 3](#_Toc182771499)

[1.3. The Rising Threat of Cyber Attacks 3](#_Toc182771500)

[1.4. The Rising Threat of Cyber Attacks 4](#_Toc182771501)

[II. System Architecture: *System structure and components* 5](#_Toc182771502)

[2.1. High-Level Overview 5](#_Toc182771503)

[2.2. Data Flow and Interactions 5](#_Toc182771504)

[III. Modules and Features: *Main modules and their roles* 5](#_Toc182771505)

[3.1. Backend Services 5](#_Toc182771506)

[3.2. Inventory and Product Management 6](#_Toc182771507)

[3.3. Order Management 6](#_Toc182771508)

[3.4. User and Role Management 6](#_Toc182771509)

[IV. Security Features: *Key security mechanisms* 7](#_Toc182771510)

[4.1. Authentication and Authorization 7](#_Toc182771511)

[4.2. Input Validation and Data Integrity 8](#_Toc182771512)

[4.3. Token Security 9](#_Toc182771513)

[4.4. Secure Password Management 11](#_Toc182771514)

[4.5. Secure File Uploads 11](#_Toc182771515)

[4.6. Communication Security 12](#_Toc182771516)

[4.7. Notifications and Alerts 12](#_Toc182771517)

[4.8. Error Handling and Logging 13](#_Toc182771518)

[V. Implementation Details: *Code examples and workflows* 13](#_Toc182771519)

[5.1. Database Schema 13](#_Toc182771520)

[5.2. Workflow Diagrams 15](#_Toc182771521)

[VI. Testing and Validation: *Testing process and results* 17](#_Toc182771522)

[6.1. Functional Testing 17](#_Toc182771523)

[6.2. Security Testing 18](#_Toc182771524)

[6.3. Results and Observations 18](#_Toc182771525)

[Security Testing Results Table 19](#_Toc182771526)

[VII. Conclusion: *Summary and impact* 20](#_Toc182771527)

# Introduction: *Project overview and objectives*

The rise of e-commerce has revolutionized the way businesses and consumers interact, providing convenience and efficiency in transactions. However, this growth also introduces significant security challenges, as sensitive user data, financial information, and business operations are increasingly targeted by malicious actors. This project focuses on the development of a Secure E-commerce Platform with robust security mechanisms to protect both users and the system from vulnerabilities.

The system includes critical components like inventory management, order processing, and user-role management, all designed with security as a foundational element.

## Project Goals

The primary aim of the project is to develop a secure, user-friendly e-commerce platform while implementing robust security measures to mitigate modern cyber threats. The platform will cater to different types of users—administrators, inventory managers, product managers, and order managers—through a role-based access control system.

* Develop a fully functional e-commerce platform with modular components, including inventory and product management, order processing, and user-role management.
* Implement a robust Role-Based Access Control (RBAC) system to ensure appropriate authorization for each role.
* Secure sensitive user and transaction data through mechanisms like JWT-based authentication, secure password hashing, and input validation.
* Mitigate common threats such as SQL injection, Cross-Site Request Forgery (CSRF), and unauthorized access.
* Provide an efficient system for low stock alerts, product promotions, and order return requests, all secured with robust data integrity mechanisms.

## Importance of Security in E-commerce

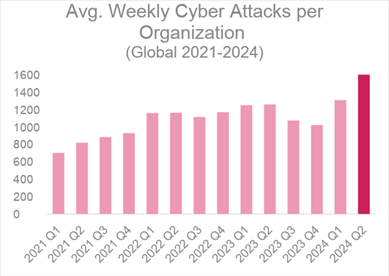
In today’s digital era, e-commerce platforms are prime targets for malicious attacks. Ensuring security is paramount to protect sensitive user data, maintain trust, and comply with regulations. Key reasons why security is critical include:

* **Sensitive Data Protection**: E-commerce systems handle sensitive user data, including personally identifiable information (PII) and payment details, which require high levels of protection.
* **Trust Building**: Secure systems enhance user trust, ensuring repeat business and reputation management.
* **Regulatory Compliance**: Adherence to regulations like GDPR and PCI DSS requires strong data security measures.
* **Mitigation of Financial Losses**: Preventing attacks like SQL injection, data breaches, and unauthorized access reduces potential financial and reputational damage.
* **System Availability**: Security measures ensure uptime and prevent disruptions caused by malicious actors.

## The Rising Threat of Cyber Attacks

The increasing frequency and sophistication of cyberattacks worldwide present a significant challenge for e-commerce platforms. As businesses rely more on digital transactions, attackers exploit vulnerabilities to compromise sensitive user data, disrupt operations, and undermine customer trust.

Recent trends reveal an alarming rise in cyberattacks targeting organizations globally. E-commerce platforms are particularly vulnerable due to the high volume of transactions, customer data, and integration with third-party systems. This highlights the urgent need for robust security measures to protect both businesses and customers.

**Visual: Average Weekly Cyber Attacks**

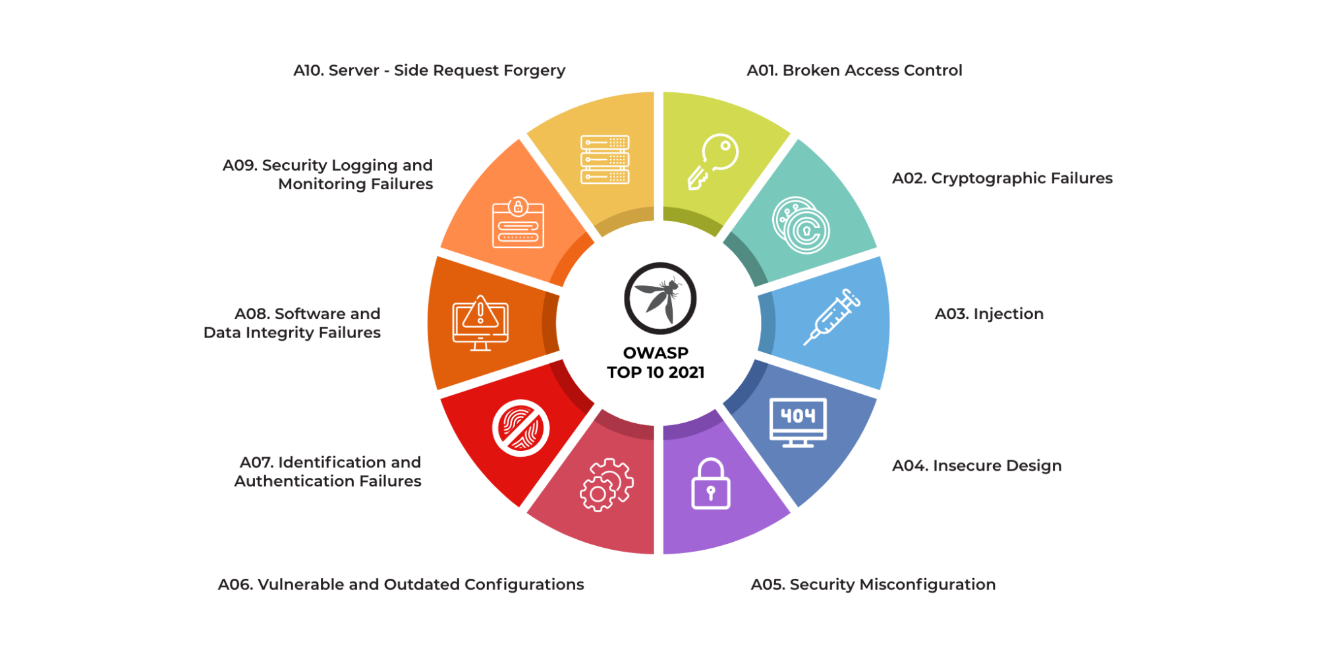
The following chart illustrates the global increase in weekly cyberattacks per organization, underscoring the growing threat landscape for e-commerce and other digital platforms.

## The Rising Threat of Cyber Attacks

The e-commerce platform's security design aligns with the **OWASP Top 10** framework, addressing critical vulnerabilities to enhance overall resilience. These risks, identified as the most prevalent and damaging in web applications, include injection flaws, misconfigurations, and broken access controls.

**Key Measures Taken in the Project:**

* **Mitigating Injection Attacks:** Input validation and secure database queries prevent SQL injection and similar attacks.
* **Enforcing Access Control:** Role-Based Access Control (RBAC) ensures that users access only permitted resources.
* **Implementing Secure Design:** Comprehensive testing and regular audits are incorporated to identify and resolve flaws in the system's architecture.



**Visual: OWASP Top 10**

The diagram below highlights the key vulnerabilities identified by OWASP. Each vulnerability serves as a guiding principle for designing secure systems, particularly in e-commerce.



# System Architecture: *System structure and components*

A well-defined system architecture is crucial for developing a secure and efficient e-commerce platform. This section provides an overview of the system's structure and the interactions between its components.

## High-Level Overview

The e-commerce platform's architecture is modular and structured into three key areas:

* **Backend Services:** Includes APIs for user and role management, inventory and product management, and order processing. The app.py files in each module define endpoints for these services.
* **Database Layer:** Uses SQLAlchemy for object-relational mapping (ORM). Models include User, Role, Product, Order, Invoice, and more, ensuring data integrity and ease of query writing.
* **Security Layer:** Implements features directly observed in the code:
  + **JWT-Based Authentication:** Secures API interactions using JSON Web Tokens.
  + **Role-Based Access Control (RBAC):** Enforces user permissions based on roles such as Admin, Inventory Manager, Product Manager, and Order Manager.
  + **CSRF Protection:** Ensures cookies and sensitive actions are protected against cross-site request forgery.

## Data Flow and Interactions

The data flow observed in the code involves the following steps:

1. **Authentication and Authorization:**
   * Users authenticate via endpoints like /login and /register.
   * Tokens are generated with role and user information, enforcing RBAC when accessing protected routes.
   * CSRF tokens are included in cookies for added security.
2. **Request Handling:**
   * User requests are routed through endpoints in app.py, where they interact with CRUD operations (e.g., create\_product, update\_inventory\_item).
   * Input validation is enforced in several modules, such as crud.py and utils.py.
3. **Database Operations:**
   * Database models are defined in models.py for entities like User, Role, Order, and Product.
   * Secure ORM queries are used to interact with these models (e.g., creating roles, adding inventory, processing orders).
4. **Security Enforcement:**
   * JWT tokens are validated with middleware like @jwt\_required().
   * Input validation is explicitly implemented (e.g., validate\_inventory\_fields in models.py and validate\_input in utils.py).
   * Alerts for stock levels are triggered in the inventory\_and\_product\_management module.

# Modules and Features: *Main modules and their roles*

## Backend Services

The **Backend Services** module is the foundation of the e-commerce platform, providing a set of APIs to handle critical business logic and facilitate interactions between users and the database. Based on the code, its key features include:

* **Authentication and Authorization:** Implements login, registration, and JWT-based token management to secure user access.
* **Data Validation:** Enforces strict input validation in utils.py and CRUD operations to ensure data integrity and prevent malicious inputs.
* **Error Handling:** Global exception handlers ensure unhandled errors are logged and provide secure responses to users.
* **Cross-Origin Resource Sharing (CORS):** Configured to allow secure communication between the frontend and backend.

## Inventory and Product Management

The **Inventory and Product Management** module manages product details, inventory levels, and promotional activities. Key features include:

* **Inventory Management:** Handles CRUD operations for inventory (e.g., create\_inventory\_item, update\_inventory\_item, get\_low\_stock\_items).
* **Product Management:** Manages product lifecycle, including creation, updates, and deletions (create\_product, update\_product, delete\_product).
* **Alerts and Monitoring:** Implements low-stock alerts via the send\_low\_stock\_alert utility function.
* **CSV Upload:** Provides functionality to bulk upload products via a CSV file (process\_csv).
* **Promotions and Coupons:** Manages promotions and discounts, ensuring active promotions are applied to products dynamically.

## Order Management

The **Order Management** module oversees customer orders, returns, and invoices. Key features include:

* **Order Processing:** Creates and updates orders with validation for data integrity (create\_order, update\_order\_status).
* **Return Requests:** Manages customer return requests with options for approval or denial (create\_return\_request, process\_return\_request).
* **Invoice Generation:** Automatically generates invoices based on order details (generate\_invoice).
* **Notifications:** Simulates customer notifications for status updates (notify\_customer).

## User and Role Management

The **User and Role Management** module secures the platform by managing users, their roles, and permissions. Key features include:

* **User Registration and Login:** Enables secure user registration and authentication (register\_user, login).
* **Role-Based Access Control (RBAC):** Assigns roles (Admin, Inventory Manager, Product Manager, Order Manager) and enforces access permissions.
* **Permissions Management:** Allows assignment of specific permissions to roles (assign\_permission\_to\_role).
* **Token Management:** Secures APIs using JWT tokens with role-based claims (create\_access\_token).

# Security Features: *Key security mechanisms*

The e-commerce platform incorporates several robust security mechanisms to safeguard data, ensure user privacy, and defend against common vulnerabilities.

## Authentication and Authorization

The system implements **JWT-based Authentication** and **Role-Based Access Control (RBAC)** to ensure secure access to resources.

* **Authentication:**
  + Users authenticate using their credentials (username and password). Upon successful login, a **JWT access token** is generated and stored securely in HTTP-only cookies. The token includes claims such as role\_id, user\_id, and username, which are used for subsequent authorization checks.
  + Refresh tokens ensure extended access without requiring frequent logins.

***Code Highlight:***

@app.route("/login", methods=["POST"])

def login():

user = User.query.filter\_by(username=data["username"]).first()

if user and check\_password\_hash(user.password\_hash, data["password"]):

access\_token = create\_access\_token(identity=user.id, additional\_claims={

"role\_id": user.role\_id,

"username": user.username,

})

refresh\_token = create\_refresh\_token(identity=user.id)

csrf\_access\_token = decode\_token(access\_token)["csrf"]

csrf\_refresh\_token = decode\_token(refresh\_token)["csrf"]

response = make\_response(jsonify({"message": "Login successful"}))

set\_cookie(response, "access\_token", access\_token)

set\_cookie(response, "refresh\_token", refresh\_token)

set\_cookie(response, "csrf\_access\_token", csrf\_access\_token)

set\_cookie(response, "csrf\_refresh\_token", csrf\_refresh\_token)

logging.info(f"User '{user.username}' logged in")

return response, 200

return jsonify({"error": "Invalid credentials"}), 401

* **Role-Based Access Control (RBAC):**  
  Specific endpoints enforce role-based restrictions using the authorize decorator. For instance, only users with Admin or Inventory Manager roles can access inventory-related endpoints.

***Code Highlight:***

@app.route('/inventory', methods=['GET'])

@jwt\_required()

@authorize(required\_roles=[1, 4]) # Admin and Inventory Manager

def get\_inventory\_route():

inventory\_data = [item.to\_dict() for item in get\_inventory()]

return jsonify(inventory\_data), 200

A diagram of a company

Description automatically generated

*A chart displaying user roles (e.g., Admin, Product Manager) and their respective permissions mapped to specific API endpoints.*

## Input Validation and Data Integrity

Robust validation ensures only valid and safe inputs are processed, preventing injection attacks and maintaining data integrity.

* **API-Level Validation:**  
  Input validation occurs at the API level. The validate\_input utility checks for the presence of required fields in requests.

***Code Highlight:***

@app.route('/create\_order', methods=['POST'])

def create\_order\_route():

    data = request.get\_json()

    if not validate\_input(data, ['customer\_id', 'customer\_email', 'items']):

        return jsonify({"error": "Invalid data provided"}), 400

    # Creating order with basic validation on items

    order = create\_order(data['customer\_id'], data['items'], data['customer\_email'])

    # Check if create\_order returned an error dictionary

    if isinstance(order, dict) and 'error' in order:

        return jsonify(order), 400

    # Otherwise, assume order is an Order object and serialize it to JSON

    return jsonify(order.to\_dict()), 201

* **SQL Injection Protection:**  
  All database queries use parameterized inputs via SQLAlchemy, ensuring raw user input is not directly executed.

***Code Highlight: Parametrized Query***

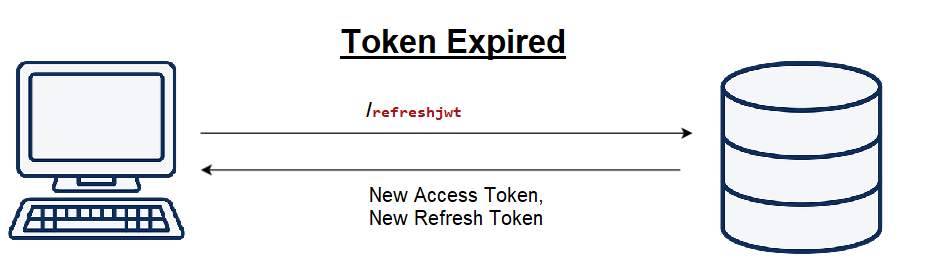
user = User.query.filter\_by(username=data["username"]).first()

|  |  |  |
| --- | --- | --- |
| **Query Type** | **Description** | **Risk** |
| **Parametrized** | SQL query uses placeholders to separate code from input | Protects against SQL injection |
| **Raw Input** | User input directly interpolated into SQL statement | Vulnerable to SQL injection attacks |

## Token Security

The system uses **JSON Web Tokens (JWT)** to manage session-based authentication securely.

* **Access and Refresh Tokens:**
  + Access tokens are stored in secure, HTTP-only cookies with a short expiration (15 minutes) to minimize risk.
  + Refresh tokens allow users to obtain a new access token without re-authentication, stored in similar secure cookies.



* **Cross-Site Request Forgery (CSRF) Protection:**
  + CSRF tokens are embedded within the access and refresh tokens and validated during critical operations.

***Code Highlight: Secure Token Storage with CSRF Protection***

@app.route("/login", methods=["POST"])

def login():

    data = request.get\_json()

    username = data.get("username", "").strip()

    password = data.get("password", "").strip()

    if not username or not password:

        return jsonify({"error": "Invalid credentials"}), 401

    user = User.query.filter(User.username == username).first()

    if user and check\_password\_hash(user.password\_hash, password):

        access\_token = create\_access\_token(identity=user.id, additional\_claims={

            "role\_id": user.role\_id,

            "username": user.username,

        })

        refresh\_token = create\_refresh\_token(identity=user.id)

        # Decode tokens to extract CSRF tokens

        csrf\_access\_token = decode\_token(access\_token)["csrf"]

        csrf\_refresh\_token = decode\_token(refresh\_token)["csrf"]

        response = make\_response(jsonify({"message": "Login successful"}))

        set\_cookie(response, "access\_token", access\_token)

        set\_cookie(response, "refresh\_token", refresh\_token)

***Code Highlight: Validating CSRF Tokens***

@app.route('/refresh', methods=['POST'])

@jwt\_required(refresh=True)

def refresh():

incoming\_csrf = request.headers.get("X-CSRF-TOKEN")

if not incoming\_csrf:

return jsonify({"error": "CSRF token missing"}), 400

# Further processing...

    # Otherwise, assume order is an Order object and serialize it to JSON

    return jsonify(order.to\_dict()), 201

## Secure Password Management

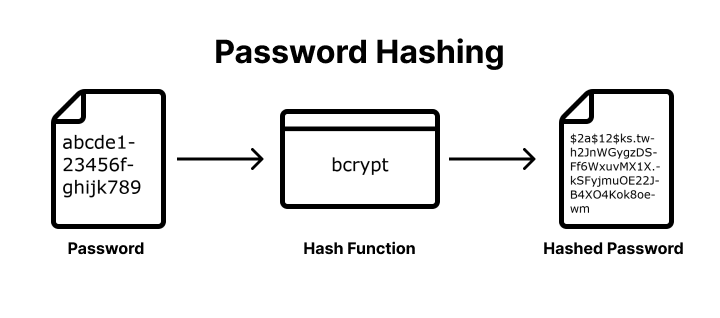
Passwords are hashed using **PBKDF2 with SHA-256** (via Werkzeug's generate\_password\_hash) before storage, preventing raw passwords from being exposed. During authentication, the system verifies passwords with check\_password\_hash.

***Code Highlight:*** ***Password Hashing***

password\_hash = generate\_password\_hash(password)

user = User(username=username, password\_hash=password\_hash, role\_id=role\_id) db.session.add(user)

db.session.commit()



***Code Highlight:*** ***Password Verification***

if user and check\_password\_hash(user.password\_hash, data["password"]):

# Generate tokens and log in user

## Secure File Uploads

To prevent malicious file uploads, the system enforces the following checks:

* Only files with .csv extensions are accepted.
* File names are sanitized using secure\_filename to avoid directory traversal attacks.
* Files exceeding 5MB are automatically rejected.

***Code Highlight:*** ***File validation***

if 'file' not in request.files or not request.files['file'].filename.endswith('.csv'):

return jsonify({"error": "Invalid file type"}), 400

file\_path = secure\_filename(file.filename)

if os.path.getsize(file\_path) > 5 \* 1024 \* 1024:

return jsonify({"error": "File size exceeds limit"}), 400

|  |  |  |
| --- | --- | --- |
| **File Type** | **Action** | **Reason** |
| .csv | Accepted | Allowed extension |
| .exe, .js | Rejected | Disallowed extensions |
| .csv (6MB) | Rejected | Exceeds file size limit (5MB) |
| Corrupted .csv file | Rejected | Fails content validation checks |

## Communication Security

* **Cross-Origin Resource Sharing (CORS):** The system restricts access to trusted origins (http://localhost:3000 during development) and ensures that cookies are sent securely during cross-origin requests.

***Code Highlight:*** ***CORS Configuration***

CORS(app, resources={r"/\*": {"origins": "http://localhost:3000"}}, supports\_credentials=True)

* **Whitelisting Trusted Domains:** URLs are validated against a whitelist of trusted domains to prevent **Server-Side Request Forgery (SSRF)** attacks. This ensures unsafe URLS are rejected

***Code Highlight:*** ***URL Whitelisting***

ALLOWED\_DOMAINS = ['example.com', 'trusted-cdn.com', 'localhost']

def is\_valid\_url(url):

parsed\_url = urlparse(url)

if parsed\_url.hostname not in ALLOWED\_DOMAINS:

return False

return True

## Notifications and Alerts

Proactive notifications ensure critical events are brought to attention:

* **Low Stock Alerts:**  
  If inventory drops below the threshold, an alert is sent to administrators via email.

***Code Highlight:*** ***Low stock alerts***

if item.stock\_level < item.threshold:

send\_low\_stock\_alert(item.product\_id, item.warehouse\_id)

## Error Handling and Logging

Errors are managed centrally to ensure that sensitive information is not leaked while detailed logs are retained for debugging.

* **Centralized Error Responses:** Users see generic error messages while logs capture specific error details.

***Code Highlight:*** ***Centralized Error Handling***

@app.errorhandler(Exception)

def handle\_exceptions(e):

logging.exception(f"Unhandled exception: {e}")

return jsonify({"error": "An internal server error occurred"}), 500

* **Detailed Logging:** Activities like user login, failed login attempts, and token validation are logged for auditing.

***Code Highlight:*** ***Login Logging***

logging.info(f"User '{user.username}' logged in")

logging.warning(f"Invalid login attempt for user '{data['username']}'")

# Implementation Details: *Code examples and workflows*

## Database Schema

The database schema forms the backbone of the e-commerce platform, enabling secure data storage and seamless operations. It is structured to support role-based access control (RBAC), inventory management, and order processing.

**Users and Roles**

The **users** table stores user information, including usernames, securely hashed passwords, and assigned roles. The **roles** table defines the different roles available on the platform (e.g., Admin, Product Manager, Inventory Manager). Each user is associated with a role via the role\_id field.

* **Schema Details:**
  + The username field ensures unique identification for each user.
  + The password\_hash field securely stores hashed passwords, preventing unauthorized access.
  + The role\_id field acts as a foreign key, linking users to roles in the **roles** table for RBAC.

For example, the **User** model is defined as follows:

*"class User(db.Model):****tablename*** *= 'users'  
id = db.Column(db.Integer, primary\_key=True)  
username = db.Column(db.String(50), unique=True, nullable=False)  
password\_hash = db.Column(db.String(256), nullable=False)  
role\_id = db.Column(db.Integer, db.ForeignKey('roles.id'), nullable=False)"*

**Inventory and Products**

The platform uses the **products** table and the **inventory** table to manage stock levels and product information. These tables are critical for maintaining data integrity and supporting secure inventory management.

* **Schema Details:**
  + The **products** table includes fields like name, subcategory\_id, price, stock\_level, and image\_url.
  + The **inventory** table tracks stock levels, warehouses, and includes a threshold field for triggering low-stock alerts.

The **Inventory** model is defined as:

*"class Inventory(db.Model):****tablename*** *= 'inventory'  
id = db.Column(db.Integer, primary\_key=True)  
product\_id = db.Column(db.Integer, nullable=False)  
warehouse\_id = db.Column(db.Integer, db.ForeignKey('warehouse.id'), nullable=False)  
stock\_level = db.Column(db.Integer, default=0)  
threshold = db.Column(db.Integer, default=10)"*

**Orders and Order Items**

To handle customer transactions, the platform utilizes the **orders** and **order\_items** tables. These tables record the order details and associated products.

* **Schema Details:**
  + The **orders** table stores fields such as customer\_id, status, total\_amount, and customer\_email.
  + The **order\_items** table includes fields for product\_id, quantity, and price\_per\_unit.

For example, the **Order** model is implemented as:

*"class Order(db.Model):****tablename*** *= 'orders'  
id = db.Column(db.Integer, primary\_key=True)  
customer\_id = db.Column(db.Integer, nullable=False)  
status = db.Column(db.String(50), nullable=False, default='pending')  
total\_amount = db.Column(db.Float, nullable=False, default=0.0)"*

**Role-Based Access Control Integration**

RBAC is implemented using the **roles** table and the **permissions** table. Permissions are linked to roles to ensure that users can only perform actions allowed by their roles.

* **Schema Details:**
  + The name field in the **permissions** table specifies the action (e.g., "Manage Inventory").
  + The role\_id field links permissions to roles.

For example, the **Permission** model is structured as:

*"class Permission(db.Model):****tablename*** *= 'permissions'  
id = db.Column(db.Integer, primary\_key=True)  
name = db.Column(db.String(50), unique=True, nullable=False)  
role\_id = db.Column(db.Integer, db.ForeignKey('roles.id'), nullable=False)"*

## Workflow Diagrams

**1) User Login Workflow**

The user login workflow ensures secure authentication and protects against unauthorized access. It employs hashed password verification and JSON Web Tokens (JWT) to manage user sessions securely.

When a user submits their credentials through a login form, the system first validates the input structure. The user record is retrieved from the database using a parameterized query to safeguard against SQL injection attacks. The code for this is:

*“user = User.query.filter\_by(username=data["username"]).first()”*

This ensures only valid, sanitized input is used to query the database. Next, the submitted password is hashed and compared against the stored hash using a secure hashing algorithm. If the credentials match, the user is authenticated. For verified users, the system generates an access token with the following code:

“access\_token = create\_access\_token(identity=user.id, additional\_claims={

"role\_id": user.role\_id,

"username": user.username

})”

This token contains the user’s identity and role claims, enabling secure session management and access to resources based on roles.

**2) Role-Based Page Access Workflow**

This workflow governs access to specific pages or resources based on the user's role and enforces Role-Based Access Control (RBAC) using JWT claims and decorators.

When a user requests a protected page, such as /admin/inventory, the system validates their JWT token to ensure its authenticity. From the token, the user’s role\_id is extracted and compared against the roles authorized for that route. For example, in the route:

“*@jwt\_required()*

*@authorize(required\_roles=[1, 2])*

*def admin\_inventory():*

*return jsonify({"message": "Inventory data"})”*

Only users with roles matching 1 (Admin) or 2 (Product Manager) can access the route. Unauthorized users receive a 403 Unauthorized error, effectively restricting their access to sensitive operations and enforcing role-based security.

**3) Inventory Management Workflow**

This workflow handles stock level updates, input validation, and alerting for low inventory. It ensures secure updates and proactive notifications.

When an Inventory Manager submits a stock update request, the system first validates the inputs to ensure the stock quantity is a valid integer and within acceptable limits. The system uses parameterized queries to process the update securely, reducing vulnerabilities to SQL injection attacks.

After validation, the stock levels in the database are updated. If the new stock level falls below a predefined threshold, the system triggers a low-stock alert with the following code:

*“if item.stock\_level < item.threshold:*

*send\_low\_stock\_alert(item.product\_id, item.warehouse\_id)”*

This proactive alert mechanism helps prevent stockouts by notifying relevant personnel. Additionally, all changes to stock levels are logged, ensuring auditability and compliance with inventory management standards.

**4) File Upload Workflow**

This workflow manages the secure upload and processing of files, specifically CSV files for bulk product additions. The system enforces strict validation to ensure that only permitted file types and sizes are processed.

* **File Submission:**  
  Users upload a file through an interface. The system verifies the file’s presence in the request. Missing files are immediately rejected with an error response.
* **File Validation:**  
  The uploaded file is checked to ensure it is a valid CSV file. The system rejects invalid file types by verifying the file extension and content type. Additionally, file size limits are enforced to prevent oversized uploads, as shown in:

*“if not filename.endswith('.csv'):*

*return jsonify({"error": "Invalid file type, only CSV files are allowed"}), 400”*

* **File Storage:**  
  The validated file is securely stored in the designated UPLOAD\_FOLDER. The file path is sanitized to prevent directory traversal attacks.
* **File Processing:**  
  The system reads the CSV data, validates each row for required fields, and ensures that data types and values meet predefined constraints. For instance, product descriptions and prices must adhere to set formats.
* **Database Updates:**  
  After validation, the system adds valid products to the database. Errors in the file are logged, and the user is notified of any invalid entries.

**5) Order Management Workflow**

This workflow details how orders are created, managed, and updated to ensure accuracy and security.

* **Order Creation:**  
  When a customer places an order, the system validates the provided items and quantities. Using parameterized queries, it calculates the total cost based on the items' prices and quantities.
* **Order Status Updates:**  
  The system supports specific status transitions, such as pending → processing → shipped → delivered. Invalid transitions, like skipping from pending to delivered, are rejected with the following logic:

“valid\_transitions = {

"pending": ["processing", "shipped"],

"processing": ["shipped"],

"shipped": ["delivered"]

}

if new\_status not in valid\_transitions.get(order.status, []):

return {"error": f"Invalid status transition from '{order.status}' to '{new\_status}'"}”

* **Notification Trigger:**  
  For every status change, the system sends a notification to the customer, ensuring transparency and keeping the customer informed about their order’s progress.
* **Invoice Generation:**  
  Upon order completion, the system generates an invoice detailing the order’s total cost, items, and other relevant information. This invoice is securely stored and shared with the customer.

# Testing and Validation: *Testing process and results*

## Functional Testing

Functional testing focuses on verifying that the e-commerce platform's core features perform as intended. The tests were carefully aligned with the functionalities implemented in the code, ensuring that all modules, endpoints, and workflows worked correctly.

**Authentication and Role-Based Access Control (RBAC)** were key features tested to validate secure user management and differentiated access. During testing, user registration was validated to ensure users could successfully sign up with a valid username, password, and role ID. Login functionality was tested to confirm that users could authenticate using valid credentials, which resulted in the correct issuance of access and refresh tokens. Invalid login attempts, such as those with incorrect passwords or non-existent usernames, were rejected, with clear error messages returned. Role-based access control was verified through the @authorize decorator in the code. Tests confirmed that Admins could access all sensitive routes such as /roles and /admin/inventory, while Inventory Managers were restricted to inventory-specific endpoints. Unauthorized access attempts consistently returned a 403 Unauthorized error.

**Inventory Management** functionalities were extensively tested to ensure efficient handling of stock. Adding new inventory items required valid inputs, including category ID, capacity, and threshold values. Tests ensured that valid inputs successfully created inventory records, while invalid inputs, such as missing fields or incorrect data types, returned detailed validation error messages. Updating inventory levels was another critical area of testing. The /inventory/<int:inv\_id> endpoint was used to modify stock levels, and scenarios were tested where stock levels dropped below defined thresholds to trigger low-stock alerts. These alerts were confirmed to work as expected. Inventory deletion functionality was tested by removing items from the database using their unique IDs. Attempts to delete non-existent inventory items returned a 404 error, confirming robust error handling.

**Product Management** workflows were thoroughly evaluated. Creating products required details such as product name, category ID, price, and stock levels. The system accurately processed valid inputs and rejected invalid ones, such as negative prices or missing required fields, with error messages explaining the issue. Updating product details, including price and stock levels, was tested to ensure modifications were correctly saved in the database. Additionally, tests confirmed that attempts to update or delete non-existent products returned appropriate error responses.

**Order Management** was another focus of functional testing. Orders were successfully created by providing customer details, item lists, and prices. The system calculated total amounts correctly and linked each order to its respective customer. The code’s workflow for order status updates was validated to ensure transitions followed predefined rules, such as moving from "pending" to "processing" and then to "shipped." Invalid transitions were blocked with appropriate error messages. Return requests were tested for functionality, including creation and status updates such as "approved" or "denied."

**File Uploads** functionality was tested for processing CSV files for bulk product uploads. The platform accepted valid CSV files with correct formats and rejected unsupported file types or oversized files. This ensured compliance with the size and type constraints implemented in the code.

## Security Testing

Security testing focused on ensuring that the platform's implemented defenses protected against common threats and vulnerabilities. Tests were aligned with the security features explicitly coded in the system, such as input validation, CSRF protection, and secure authentication workflows.

The SQL Injection defenses were tested by attempting malicious payloads in login and query endpoints. Since all database interactions used parameterized queries (e.g., User.query.filter(User.username == username)), the system effectively blocked attempts to manipulate the SQL query structure. For instance, injecting "' OR 1=1;--" in login requests returned a clean error message without compromising the database.

Cross-Site Scripting (XSS) was addressed by input validation. Testing fields such as product descriptions and inventory names with <script>alert('XSS')</script> confirmed that the system sanitized inputs, preventing script execution. No payloads were rendered, ensuring the platform resisted client-side attacks.

To mitigate Cross-Site Request Forgery (CSRF), the system enforced CSRF tokens in sensitive routes. Requests lacking the X-CSRF-TOKEN header were correctly denied access, ensuring that malicious actors could not perform unauthorized actions on behalf of authenticated users.

Input Validation and Whitelisting was tested extensively. Functions such as validate\_input ensured fields were present and correctly formatted in API requests. Additionally, URL validation with is\_valid\_url employed a whitelist approach, permitting only trusted domains and rejecting potentially harmful URLs. Test cases with malicious links, such as http://malicious-site.com, were rejected, while valid domains were accepted.

File upload functionality was evaluated for adherence to type and size constraints. Files larger than 5MB or with unsupported formats were rejected, confirming that oversized and non-CSV files posed no risk to the system.

The system's Error Handling and Logging mechanisms were tested to ensure no sensitive information was leaked in error messages. Attempts to trigger errors, such as accessing invalid routes or submitting malformed requests, returned generic error messages while logging detailed information for developers.

## Results and Observations

1. **Functional Testing Results Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Feature** | **Test Case** | **Expected Outcome** | **Actual Outcome** | **Status** |
| Login and Registration | Register and log in with valid credentials. | User successfully logged in, tokens issued. | Success. Tokens issued. | ✅ Passed |
| Login and Registration | Attempt login with invalid credentials. | Login attempt rejected with an error message. | Rejected with 'Invalid credentials' error. | ✅ Passed |
| RBAC Enforcement | Access /roles as Admin. | Admin successfully accesses the route. | Route accessed successfully. | ✅ Passed |
| RBAC Enforcement | Access /inventory as non-Inventory Manager. | Access denied with 403 error. | 403 Unauthorized error returned. | ✅ Passed |
| Inventory Management | Create new inventory item with valid inputs. | Inventory item added successfully. | Successfully added inventory. | ✅ Passed |
| Inventory Management | Delete non-existent inventory item. | 404 Inventory not found error returned. | Error returned as expected. | ✅ Passed |
| Product Management | Create a product with valid details. | Product created successfully. | Product added to the database. | ✅ Passed |
| Product Management | Update product details. | Changes saved correctly in the database. | Successfully updated product details. | ✅ Passed |
| Product Management | Delete non-existent product. | 404 Product not found error returned. | Error returned as expected. | ✅ Passed |
| Order Management | Create order with valid items and customer details. | Order created successfully, total calculated. | Order added to the database. | ✅ Passed |
| Order Management | Attempt invalid status transition (e.g., pending → delivered). | Transition rejected with error. | Error message returned. | ✅ Passed |
| File Uploads | Upload valid CSV file. | File processed successfully. | Products added successfully. | ✅ Passed |
| File Uploads | Upload file exceeding size limit. | File rejected with size error. | File rejected. | ✅ Passed |

1. **Security Testing Results Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test** | **Description** | **Expected Outcome** | **Actual Outcome** | **Status** |
| SQL Injection | Attempt malicious payloads in login endpoint. | Payloads rejected; no database compromise. | Malicious payloads rejected with generic error messages. | ✅ Passed |
| Cross-Site Scripting (XSS) | Submit <script> tags in input fields like product description. | Scripts blocked; input sanitized. | Inputs sanitized; no scripts executed. | ✅ Passed |
| CSRF Protection | Perform sensitive actions without valid CSRF token. | Access denied without token. | Requests denied with CSRF error. | ✅ Passed |
| File Upload Validation | Upload oversized or non-CSV files. | Files rejected with appropriate error messages. | Rejected files exceeding limits or invalid format. | ✅ Passed |
| Input Validation | Submit requests with missing or malformed fields. | Requests rejected with validation error messages. | Rejected invalid inputs, descriptive error returned. | ✅ Passed |
| URL Validation | Test is\_valid\_url with malicious and safe domains. | Malicious URLs rejected, safe domains accepted. | Malicious links rejected, valid links accepted. | ✅ Passed |

# Conclusion: *Summary and impact*

This e-commerce platform successfully integrates essential functionalities with strong security measures, offering a reliable solution for managing inventory, products, orders, and user roles. Features like **Role-Based Access Control (RBAC)** ensure that access to sensitive operations is limited based on roles, enhancing operational efficiency and security.

The system implements modern security practices to mitigate critical threats such as **SQL Injection**, **Cross-Site Scripting (XSS)**, and **Cross-Site Request Forgery (CSRF)**. Measures like token-based authentication, input validation, and secure file handling reinforce data protection and system integrity.

The platform’s modular and scalable architecture allows for future enhancements and adaptability to evolving business requirements. Its secure design and functionality provide businesses with a dependable framework to streamline operations while safeguarding sensitive information, promoting trust and reliability in digital commerce.

This e-commerce platform successfully integrates essential functionalities with strong security measures, offering a reliable solution for managing inventory, products, orders, and user roles. Features like **Role-Based Access Control (RBAC)** ensure that access to sensitive operations is limited based on roles, enhancing operational efficiency and security.

The system implements modern security practices to mitigate critical threats such as **SQL Injection**, **Cross-Site Scripting (XSS)**, and **Cross-Site Request Forgery (CSRF)**. Measures like token-based authentication, input validation, and secure file handling reinforce data protection and system integrity.

The platform’s modular and scalable architecture allows for future enhancements and adaptability to evolving business requirements. Its secure design and functionality provide businesses with a dependable framework to streamline operations while safeguarding sensitive information, promoting trust and reliability in digital commerce.

**Thank you!**