**Test Plan for Student Registration System**

1. Introduction

Project Overview

The Student Registration System is a web-based application designed to facilitate the registration of students for various courses and manage payment processing efficiently. It comprises several interconnected components, including user management, course management, registration, and payment functionalities. The system allows students to browse available courses, register for their chosen classes, and process payments securely via the Stripe payment gateway. The application is built using Django, leveraging its robust framework to ensure a seamless user experience and maintain data integrity.

Testing Goals

The primary goals of testing for the Student Registration System are as follows:

Validate Functionality: Ensure that all features, including user registration, course selection, registration management, and payment processing, work as intended without errors.

Enhance Usability: Assess the user interface for intuitiveness and accessibility, ensuring that users can navigate the system with ease.

Ensure Performance: Evaluate the system's performance under various load conditions to confirm it can handle peak usage scenarios without degradation.

Identify and Resolve Defects: Detect any defects or vulnerabilities within the application and ensure they are addressed before deployment to maintain software quality.

Verify Integration: Confirm that the interactions between different modules (registration and payment) function correctly and data flows seamlessly throughout the system.

Compliance and Security: Ensure that the system adheres to relevant security standards and regulations, protecting user data and payment information.

Stakeholders

The successful development and deployment of the Student Registration System involve several key stakeholders, each playing a critical role in the project:

Project Manager: Oversees the entire project, coordinating activities between teams, managing timelines, and ensuring stakeholder requirements are met.

QA Lead: Responsible for developing the testing strategy, defining test objectives, and managing the quality assurance team to ensure thorough testing.

Test Engineers: Execute test cases, report defects, and validate fixes. They are responsible for manual and automated testing efforts.

Developers: Implement the system's functionalities and address any defects identified during testing. They collaborate closely with QA to ensure a high-quality product.

Product Owner: Represents the end-users, defining requirements and expectations for the system. They are involved in acceptance testing and provide feedback on usability.

End Users (Students): The primary users of the system who will register for courses and make payments. Their feedback is crucial for assessing usability and overall satisfaction with the application.

2. Scope of Testing

Features to be Tested:

User registration and role validation (student vs. non-student).

Course selection and registration process.

Payment processing through the Stripe API.

Handling of successful and failed payments.

User interface elements and navigation across the application.

Features Not to be Tested:

Third-party integrations outside of payment processing (e.g., other external APIs).

Non-functional aspects such as performance under extreme load (initially).

3. Test Objectives

Functional Validation:

Ensure that all functionalities of the student registration system operate as intended. This includes:

User registration and authentication processes.

Course creation, management, and availability checks.

Registration for courses by students.

Payment processing via Stripe API, including handling both successful and failed transactions.

Defect Detection:

Identify and document defects in the system, including:

Functional defects (e.g., incorrect registration or payment processing).

Usability defects (e.g., confusing user interfaces or navigation issues).

Security vulnerabilities (e.g., unauthorized access to registration or payment features).

Performance Assessment:

Validate the system’s performance under normal and peak load conditions by:

Measuring response times for user interactions (registration and payment).

Assessing the system’s stability during concurrent user registrations and payments.

User Experience Evaluation:

Ensure that the user interface is intuitive and user-friendly by:

Conducting usability testing with real users to gather feedback.

Ensuring accessibility compliance for users with disabilities.

Integration Verification:

Test the integration between different modules of the system, specifically:

The interaction between the registration app and the payment app.

The communication with the Stripe API for payment processing.

The correct flow of data between user registration, course registration, and payment records.

Data Integrity and Validation:

Ensure that data is accurately captured and stored in the database by:

Verifying that registration records are correctly linked to user and course data.

Ensuring that payment records reflect accurate amounts and statuses.

Error Handling and Recovery:

Test the application’s ability to handle errors gracefully by:

Simulating failure scenarios (e.g., payment failures, invalid course registrations).

Verifying that appropriate error messages are displayed to users and that the system can recover from errors without data loss.

Compliance and Security:

Validate compliance with relevant regulations and standards by:

Ensuring that sensitive data (e.g., payment information, user credentials) is handled securely.

Testing the application for vulnerabilities that could lead to data breaches or unauthorized access.

Documentation and Reporting:

Ensure that all testing activities are documented thoroughly by:

Creating detailed test cases and scripts that cover all functional requirements.

Compiling a comprehensive test report that summarizes findings, defects, and recommendations for improvements.

4. Test Strategy

Levels of Testing

Unit Testing:

Objective: Validate individual components of the application to ensure they function correctly in isolation.

Focus Areas:

Test models in the models.py file to confirm that data relationships and validations work as intended (e.g., Registration, Course, Payment).

Test form validations in the registration and payment forms to ensure they enforce correct data entry.

Integration Testing:

Objective: Verify the interactions between different modules and ensure that they work together seamlessly.

Focus Areas:

Test the integration between the registration and payment systems, ensuring that registrations correctly create payment records.

Validate the functionality of the Stripe API integration to ensure payments are processed accurately based on course selections.

System Testing:

Objective: Conduct end-to-end testing of the complete application to ensure it meets the specified requirements.

Focus Areas:

Test the entire workflow from user registration to course registration and payment processing.

Verify that user interfaces respond correctly to inputs and that all functionalities are accessible and work together as expected.

Acceptance Testing:

Objective: Ensure the application meets business requirements and is ready for deployment.

Focus Areas:

Conduct user acceptance testing (UAT) with real users (students) to gather feedback on usability and functionality.

Validate that all acceptance criteria defined by stakeholders are met prior to going live.

Types of Testing

Manual Testing:

Perform exploratory testing and usability assessments to evaluate user experience.

Execute test cases for functional requirements to ensure correctness.

Automated Testing:

Develop automated test scripts for regression testing of critical functionalities, such as course registration and payment processing.

Use tools like Selenium for UI testing to ensure that the application behaves correctly across different browsers.

Regression Testing:

Conduct regression tests after any code changes to ensure that existing functionalities remain unaffected by new features or fixes.

Automate regression test cases to streamline the testing process after each release.

Load Testing:

Perform load testing to assess system performance under high user loads, simulating multiple users registering and processing payments simultaneously.

Utilize tools like JMeter or LoadRunner to identify performance bottlenecks and ensure the system can handle peak usage.

Security Testing:

Conduct security assessments to identify vulnerabilities, including penetration testing and code reviews for security best practices.

Ensure compliance with data protection regulations regarding sensitive information handling.

Test Environment Requirements

Hardware Requirements:

A dedicated server for hosting the application during testing, equipped with sufficient CPU and RAM to handle concurrent users during load testing.

Local development machines for test engineers to run automated tests and perform manual testing.

Software Requirements:

Operating System: Any OS that supports Django (e.g., Ubuntu, Windows, macOS).

Web Server: A web server (e.g., Gunicorn, Nginx) for serving the Django application.

Database: PostgreSQL or SQLite for database management to store user, course, registration, and payment data.

Django Environment: Django framework installed along with necessary packages and libraries (e.g., Stripe Python library).

Testing Tools:

Automation: Selenium for automated UI testing.

Load Testing: Apache JMeter or LoadRunner for performance testing.

Test Management: Jira or TestRail for tracking test cases and defects.

Network Requirements:

Stable internet connection for testing payment processing through the Stripe API.

Configuration to allow access to the testing server from testing environments.

5. Test Deliverables

Test Cases/Scripts: Detailed test cases covering all functional and non-functional requirements.

Test Data: Sample data for registration and payment scenarios.

Test Reports and Defect Logs: Documentation of test results, including any identified defects and their status.

6. Entry and Exit Criteria

Entry Criteria for Testing:

Development Completion:

The development team has completed coding for all planned features and fixes in the release.

All features intended for the current testing phase are implemented and available in the testing environment.

Stability of the Build:

The application build has been tested by the development team, with no high-severity defects remaining.

The application is stable enough for testing, with all critical functionalities operational.

Test Environment Setup:

The testing environment is configured correctly and mirrors the production environment as closely as possible.

All necessary software (e.g., web server, database, testing tools) is installed and functioning properly.

Test Cases Prepared:

Test cases and scripts have been designed, reviewed, and approved by the QA team.

Test data required for executing test cases is available and prepared.

Documentation Accessibility:

All relevant documentation (requirements, design documents, user stories) is available for reference.

Previous testing reports, if applicable, are accessible to understand prior test results and any unresolved issues.

Stakeholder Approval:

The project manager and key stakeholders have approved the testing plan and test cases.

All necessary resources (testers, tools, etc.) are allocated and ready for use.

Exit Criteria for Testing:

Completion of Test Cases:

All planned test cases have been executed, including functional, integration, system, and acceptance tests.

Test coverage meets the predefined requirements, ensuring all critical functionalities are tested.

Defect Resolution:

All critical and high-severity defects have been addressed and resolved.

Any medium and low-severity defects are documented, and a decision has been made regarding their resolution (e.g., deferred for future releases).

Pass Rate:

A minimum pass rate is achieved for all executed test cases (e.g., 95% or higher).

All essential functionalities must pass their respective tests to ensure reliability.

Performance Benchmarks:

The application meets predefined performance benchmarks under expected load conditions (e.g., response times, throughput).

Load testing results indicate that the system can handle the anticipated user load without degradation.

User Acceptance Testing (UAT):

UAT has been conducted with real users (students), and feedback has been collected.

All critical feedback from UAT has been addressed, with necessary adjustments made.

Test Documentation:

Test reports summarizing results, including defect logs and test execution status, have been compiled and reviewed.

All test documentation is complete, including test cases, test data, and any modifications made during testing.

Stakeholder Sign-off:

Formal sign-off has been obtained from stakeholders, including the project manager, product owner, and key team members, indicating that the application meets the acceptance criteria for release.

All stakeholders are informed of the testing outcomes and any unresolved issues.

7. Resources

1.Team Members:

1 QA Lead.

2 Test Engineers.

2. Testing Tools

Test Management Tools:

Jira: For tracking test cases, defects, and overall project management.

TestRail: For organizing test cases, managing test runs, and reporting on test results.

Automated Testing Tools:

Selenium: For automated UI testing across different browsers and platforms.

Postman: For testing API endpoints and validating responses during integration testing.

Performance Testing Tools:

Apache JMeter: For load testing and performance assessment of the application under various user loads.

LoadRunner: (Optional) For more advanced load testing capabilities and detailed performance analysis.

Security Testing Tools:

OWASP ZAP: For identifying security vulnerabilities and conducting penetration testing on the application.

Burp Suite: For web application security testing to find vulnerabilities like XSS and SQL injection.

Continuous Integration/Continuous Deployment (CI/CD) Tools:

Jenkins: For automating the testing process and integrating automated tests into the development pipeline.

GitHub Actions: For automating workflows related to testing and deployment.

Version Control:

Git: For source code version control, allowing multiple team members to collaborate effectively.

3. Hardware Requirements

Testing Environment Server:

Specifications:

CPU: Minimum 4 cores (e.g., Intel i5 or equivalent).

RAM: At least 16 GB to support multiple concurrent tests.

Storage: 500 GB SSD for faster read/write speeds and sufficient storage for the application and database.

Local Development Machines:

Individual machines for test engineers with the following minimum specifications:

CPU: Dual-core processor (e.g., Intel i3 or equivalent).

RAM: Minimum 8 GB.

Storage: 256 GB SSD or larger.

Network Requirements:

Reliable internet connection for accessing cloud services (e.g., Stripe API) and for collaboration tools (e.g., Jira, GitHub).

Proper network configuration to allow secure access to the testing environment from different locations.

4. Software Resources

Operating System:

Ubuntu or Windows: For hosting the application and supporting the testing environment.

1. Testing Tools

Test Management Tools:

Jira: For tracking test cases, defects, and overall project management.

TestRail: For organizing test cases, managing test runs, and reporting on test results.

Automated Testing Tools:

Selenium: For automated UI testing across different browsers and platforms.

Postman: For testing API endpoints and validating responses during integration testing.

Performance Testing Tools:

Apache JMeter: For load testing and performance assessment of the application under various user loads.

LoadRunner: (Optional) For more advanced load testing capabilities and detailed performance analysis.

Security Testing Tools:

OWASP ZAP: For identifying security vulnerabilities and conducting penetration testing on the application.

Burp Suite: For web application security testing to find vulnerabilities like XSS and SQL injection.

Continuous Integration/Continuous Deployment (CI/CD) Tools:

Jenkins: For automating the testing process and integrating automated tests into the development pipeline.

GitHub Actions: For automating workflows related to testing and deployment.

Version Control:

Git: For source code version control, allowing multiple team members to collaborate effectively.

2. Hardware Requirements

Testing Environment Server:

Specifications:

CPU: Minimum 4 cores (e.g., Intel i5 or equivalent).

RAM: At least 16 GB to support multiple concurrent tests.

Storage: 500 GB SSD for faster read/write speeds and sufficient storage for the application and database.

Local Development Machines:

Individual machines for test engineers with the following minimum specifications:

CPU: Dual-core processor (e.g., Intel i3 or equivalent).

RAM: Minimum 8 GB.

Storage: 256 GB SSD or larger.

Network Requirements:

Reliable internet connection for accessing cloud services (e.g., Stripe API) and for collaboration tools (e.g., Jira, GitHub).

Proper network configuration to allow secure access to the testing environment from different locations.

3. Software Resources

Operating System:

Ubuntu or Windows: For hosting the application and supporting the testing environment.

Docker: For containerizing the application and testing environment to ensure consistency across different setups.

Web Server:

Gunicorn or Nginx: For serving the Django application during testing.

Database Management System:

PostgreSQL or SQLite: For managing application data, including user registrations, courses, and payment records.

Django Framework:

The latest stable version of Django to support application development and testing.

API Integration:

Stripe Python SDK: For integrating payment processing features within the application.

Testing Frameworks:

pytest: For writing and executing automated tests for the application.

unittest: Native Python unit testing framework for testing individual components.

Documentation Tools:

Confluence: (Optional) For maintaining project documentation and test-related information.

Markdown Editors: For drafting test plans and reports.

Web Server:

Gunicorn or Nginx: For serving the Django application during testing.

Database Management System:

PostgreSQL or SQLite: For managing application data, including user registrations, courses, and payment records.

Django Framework:

The latest stable version of Django to support application development and testing.

API Integration:

Stripe Python SDK: For integrating payment processing features within the application.

Testing Frameworks:

pytest: For writing and executing automated tests for the application.

unittest: Native Python unit testing framework for testing individual components.

Documentation Tools:

Confluence: (Optional) For maintaining project documentation and test-related information.

Markdown Editors: For drafting test plans and reports.

8. Schedule

Timeline:

Test Planning: 1 week.

Test Case Development: 2 weeks.

Test Execution: 1 weeks.

Defect Triage and Retesting: 1 weeks.

Final Reporting: 1 week.

9. Risk Management

Potential Risks:

Delays in code deliveries may impact testing timelines.

Unforeseen defects may require additional testing cycles.

Resource availability may fluctuate.

Mitigation Strategies:

Maintain regular communication with developers to anticipate changes.

Prioritize critical test cases to ensure essential functionalities are validated first.

Cross-train team members to cover for absences.

10. Approval

Approval Process:

The test plan and all deliverables will be reviewed and approved by:

Project Manager.

QA Lead.

Key stakeholders (e.g., product owner, developers).

**2. Test Analysis and Design**

**2.1 Test Analysis**

**2.1.1 Requirement Analysis**

#### **1. Functional Requirements**

**User Management**

* **Registration**: Users must be able to register as students, providing necessary information (username, email, password).
* **Authentication**: Users must be able to log in and log out securely.
* **Role Management**: The system should differentiate between students and other types of users (e.g., administrators).

**Course Management**

* **Course Creation**: Administrators should be able to create, update, and delete courses.
* **Course Details**: Each course must have a name, description, available slots, and price.
* **Course Availability**: Courses should reflect the correct number of available slots based on registrations.

**Registration for Courses**

* **Course Registration**: Students should be able to register for available courses.
* **Registration Confirmation**: Upon successful registration, students should receive confirmation of their enrollment.

**Payment Processing**

* **Payment Integration**: The system must integrate with Stripe for processing payments.
* **Payment Creation**: A payment record should be created upon initiating a payment for a registered course.
* **Payment Status Tracking**: The system should track payment status (Pending, Completed, Failed) and update accordingly.
* **Success and Failure Handling**: Upon payment completion, users should be redirected to a success page; if payment fails, they should be redirected to a failure page.

**User Interface**

* **Registration Form**: A form should be available for students to register for courses, including CSRF protection.
* **Payment Form**: A form should be available for users to make payments for their selected courses.

#### **2. Non-Functional Requirements**

**Performance**

* **Response Time**: The system should respond to user actions (e.g., course registration, payment processing) within 2 seconds under normal load.
* **Load Handling**: The system should support at least 100 concurrent users without performance degradation.

**Usability**

* **User Experience**: The registration and payment interfaces should be intuitive and user-friendly, with clear instructions and error messages.
* **Accessibility**: The application should comply with accessibility standards (e.g., WCAG 2.1) to ensure it is usable for all students, including those with disabilities.

**Security**

* **Data Encryption**: All sensitive data (e.g., user credentials, payment information) must be securely encrypted during transmission and storage.
* **Input Validation**: The system should validate all user inputs to prevent SQL injection, cross-site scripting (XSS), and other common vulnerabilities.

**Reliability**

* **Error Handling**: The system should gracefully handle errors and provide meaningful feedback to users without crashing.
* **Backup and Recovery**: Regular backups of user data, course information, and payment records should be maintained to ensure recovery in the event of data loss.

**Scalability**

* **Future Growth**: The architecture should support scaling to accommodate an increasing number of users and courses without significant rework.

#### **3. Technical Requirements**

**Architecture**

* **Framework**: The application should be built using the Django framework, following MVC architecture principles.
* **Database**: A relational database (e.g., PostgreSQL or SQLite) should be used to manage user, course, registration, and payment data.

**Development Environment**

* **Version Control**: Git should be used for version control to manage code changes and collaboration among developers.
* **Testing Framework**: The application should include unit tests and integration tests using frameworks like pytest or Django's built-in testing tools.

**Integration**

* **Payment Gateway**: Integration with the Stripe API for payment processing, including handling payment intents and sessions.
* **API Endpoints**: The system should provide API endpoints for frontend interactions (e.g., fetching course prices).

**Deployment**

* **Web Server**: The application should be hosted on a production-ready web server (e.g., Gunicorn, Nginx).
* **Containerization**: Optionally, Docker may be used for containerizing the application to ensure consistency across development, testing, and production environments.

**2.1.2 Test scenarios and Test conditions**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Scenario | Test Condition Description | Requirement Mapping | Risk Level | Importance Level | Complexity Level |
| User Authentication | Verify user can register with valid data | User Management: Registration | High | High | Medium |
| User Authentication | Verify user cannot register with invalid data | User Management: Registration | High | High | Medium |
| User Authentication | Verify user can log in with valid credentials | User Management: Authentication | High | High | Medium |
| User Authentication | Verify user cannot log in with invalid credentials | User Management: Authentication | High | High | Medium |
| Course Management | Verify admin can create a new course | Course Management: Course Creation | Medium | High | Medium |
| Course Management | Verify admin can update an existing course | Course Management: Course Update | Medium | High | Medium |
| Course Management | Verify course details are displayed correctly | Course Management: Course Details | Medium | High | Low |
| Payment Processing | Verify payment is processed successfully | Payment Processing: Payment Creation | High | High | High |
| Payment Processing | Verify payment fails with invalid payment details | Payment Processing: Payment Handling | High | High | High |
| Payment Processing | Verify payment status updates correctly | Payment Processing: Payment Status Tracking | High | High | Medium |
| User Interface | Verify user interface is user-friendly | Usability: User Experience | Medium | High | Medium |
| Security Testing | Verify data is encrypted during transmission | Security: Data Encryption | High | High | High |
| Security Testing | Verify input validation to prevent SQL injection | Security: Input Validation | High | High | High |

**2.2 Test Design**

**2.2.1 Test Cases**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Case ID | Description | Preconditions | Test Steps | Expected Result | Priority |
| TC-001 | Verify user can register with valid data | User is on the registration page | 1. Fill in the registration form with valid data  2. Click the "Register" button. | User is successfully registered and redirected to the login page. | High |
| TC-002 | Verify user cannot register with invalid data | User is on the registration page | 1. Fill in the registration form with invalid data (e.g., empty username).  2. Click the "Register" button. | User sees an error message indicating the invalid input. | High |
| TC-003 | Verify user can log in with valid credentials | User has a registered account | 1. Navigate to the login page.  2. Enter valid username and password.  3. Click the "Login" button. | User is successfully logged in and redirected to the dashboard. | High |
| TC-004 | Verify user cannot log in with invalid credentials | User has a registered account | 1. Navigate to the login page.  2. Enter an invalid username/password.  3. Click the "Login" button. | User sees an error message indicating incorrect credentials. | High |
| TC-005 | Verify admin can create a new course | Admin is logged in | 1. Navigate to the course creation page  2. Fill in the course details.  3. Click the "Create Course" button. | Course is created successfully, and the admin sees a confirmation message. | Medium |
| TC-006 | Verify student can register for an available course | Student is logged in, course is available | 1. Navigate to the course list.  2. Select an available courses  3. Click the "Register" button | Student is successfully registered for the course and sees a confirmation. | High |
| TC-008 | Verify payment is processed successfully | Student has registered for a course | 1. Navigate to the payment page.  2. Fill in payment details.  3. Click the "make Payment" button. | Payment is processed successfully, and the student is redirected to the payment success page | High |
| TC-009 | Verify payment fails with invalid payment details | Student has registered for a course | 1. Navigate to the payment page.  2. Fill in invalid payment details (e.g., expired card).  3. Click the "Proceed to Payment" button. | User sees an error message indicating payment failure. | High |
| TC-010 | Verify payment status updates correctly | Student has made a payment | 1. Make a payment for a course.  Check payment status in the database. | Payment status should be updated to "Completed" after successful payment. | Medium |
| TC-011 | Verify user interface is user-friendly | User is on the registration page | 1. Navigate to the registration page.  2. Evaluate layout, buttons, and error messages. | User finds the interface easy to navigate and understand. | Medium |
| TC-012 | Verify data is encrypted during transmission | User is on the payment page | 1. Make a payment.  2. Analyze network traffic. | Sensitive data should be encrypted (e.g., using HTTPS). | High |
| TC-013 | Verify input validation to prevent SQL injection | User is on the registration page | 1. Fill in the registration form with SQL injection code.  2. Click the "Register" button | System should reject the input without executing any SQL commands. | High |

**2.2.2 Test data**