10/11/2023

Online E-Store

Software specification

Bernard Katamanso
PLS PUT A NAME HERE

Table of Contents

Software Specifications Document:

1. Introduction:

- Project Overview:
  - Provide a brief introduction to the e-commerce site project.
  - Explain the primary purpose and objectives of the site.

- Target Audience:
  - Describe the intended user base for the e-commerce site.
  - Highlight the primary user groups, such as customers, sellers, and administrators.

- Expected Benefits:
  - Enumerate the expected benefits and outcomes of the e-commerce site.
  - Include any goals related to revenue, market expansion, or user satisfaction.

2. Scope:

- Project Boundaries:
  - Define the scope and limitations of the project.
  - Specify what the e-commerce site will include and any excluded features.

- Functional Modules:
  - List the primary functional modules of the e-commerce site, such as product listings, user profiles, shopping cart, checkout, and more.

3. Functional Requirements:

- Use Cases:
  - Provide detailed use cases that describe interactions between users and the system.
  - Include scenarios for common user actions like product purchase, registration, and review submissions.

- User Stories:
  - Present user stories that capture specific user needs, goals, and tasks.
  - Each user story should follow the "As a [user type], I want [an action] so that [a benefit or goal]."

4. Non-Functional Requirements:

- System Performance:
  - Define the expected response times and system performance metrics.
  - Specify acceptable levels of concurrent users and server load.

- Security:
  - Describe the security measures in place, including authentication, authorization, and data encryption.
  - Specify how user data and payment information are protected.

- Scalability:
  - Detail the ability of the system to scale horizontally or vertically as needed.
  - Include plans for accommodating future growth in user base and data volume.

- Usability:
  - Highlight usability requirements, including user interface design, navigation, and accessibility.

- Regulatory and Compliance:
  - Mention any legal or industry-specific compliance requirements.
  - Describe how the system adheres to relevant regulations.

5. User Interfaces:

- Wireframes and Mock-ups:
  - Include wireframes or mock-ups of key user interfaces.
  - Provide visual representations of the main pages and elements.

- User Journey:
  - Explain the typical user journey through the e-commerce site.
  - Detail how different components interact to fulfill user actions.

6. Data Model:

- Database Structure:
  - Define the structure of the database, including tables, fields, and relationships.
  - Specify primary and foreign keys.

- Data Entities:
  - List the data entities that the system will handle.
  - Include their attributes, data types, and constraints.

7. Technology Stack:

- Technologies:
  - Enumerate the technologies, programming languages, and frameworks that will be used in the development of the e-commerce site.
  - Specify the database management system.

8. Testing Plan:

- Testing Methodologies:
  - Describe the testing methodologies that will be employed, such as unit testing, integration testing, and user acceptance testing.

9. Deployment and Hosting:

- Deployment Plan:
  - Explain how the application will be deployed to servers or cloud platforms.
  - Include details about server configurations and hosting providers.

10. Maintenance and Support:

- Post-launch Support:
  - Outline the plans for post-launch support and maintenance.
  - Include information on bug fixes, updates, and response times for user support.

Business Logic Rules:

User Authentication and Authorization:

- Describe the procedures for user registration, login, and password reset.
- Specify user roles and permissions, and which actions are restricted to certain roles.

Product Management:

- Define the processes for adding, editing, and deleting products.
- Detail how products are categorized and what attributes they have.

Shopping Cart:

- Explain how items are added to and removed from the shopping cart.
- Specify how order totals are calculated, including any taxes or discounts.

Order Processing:

- Specify the order fulfillment process, including payment processing, order confirmation, and shipping.

Search and Filtering:

- Define how users can search for products and apply filters to narrow down product selections.

Reviews and Ratings:

- Describe the rules for users to leave reviews and ratings for products.
- Explain how reviews are moderated, if applicable.

Shipping and Delivery:

- Specify how shipping options are provided and how order tracking is managed.

Returns and Refunds:

- Detail the policy and process for returns and refunds, including timelines and conditions.

Customer Support:

- Describe how users can contact customer support, including available communication channels and response timeframes.

Promotions and Discounts:

- Specify how discounts and promotional offers are created, managed, and applied to orders.

API Documentation

1. Create Django Models:

In your Django project, create a Python class for each database table (model) that corresponds to your SQL schema.

Define fields for each model class to match the columns in your SQL tables.

Use Django's field types (e.g., CharField, IntegerField, ForeignKey) to represent different data types and relationships.

Establish foreign key relationships by referencing other model classes.

Ensure that you provide correct options and attributes for each field, such as specifying primary keys, unique constraints, and maximum lengths.

2. Create Django Views:

Django views are Python functions that handle HTTP requests and return HTTP responses. These functions define how your API interacts with the database models.

Create views for different operations like creating, reading, updating, and deleting records.

Use Django's APIView or ModelViewSet classes to simplify the creation of these views.

Write view functions that use Django ORM to perform CRUD operations on your models.

Add logic to handle different HTTP methods (GET, POST, PUT, DELETE) for each view.

3. Define URL Patterns:

In your Django app's urls.py file, define URL patterns that map specific URLs to your view functions.

Use Django's path() or re_path() functions to define URL patterns, and link them to view functions.

Include the API version number in your URL patterns if you are following versioning.

4. Configure Django Rest Framework (DRF):

Install Django Rest Framework (DRF) into your Django project using pip.

Configure DRF in your project's settings by adding it to the INSTALLED_APPS and defining settings such as authentication classes and permission classes.

DRF provides powerful tools for building APIs, including serializers for transforming data to and from JSON, viewsets for simplifying view creation, and built-in authentication classes like Token Authentication or OAuth.

5. Authentication and Permissions:

Determine the authentication method that best suits your project (e.g., Token Authentication, JWT, OAuth).

Configure authentication classes in your project's settings to enforce authentication.

Define permission classes to control access to specific API endpoints. You can set permissions based on user roles and data ownership.

6. Testing:

Write unit tests for your views and serializers to ensure that your API functions correctly.

Django provides a testing framework that allows you to create test cases and execute them with the manage.py test command.

Test different scenarios, including valid and invalid requests, to verify the behavior of your API.

7. Documentation:

Use tools like Django Rest Swagger, drf-yasg, or the built-in DRF documentation generator to create API documentation.

Document your API endpoints, including request parameters, response structures, and sample requests and responses.

Include information about authentication, permissions, and any other relevant details.

8. Deployment:

Deploy your Django API on a web server or a cloud platform like Heroku, AWS, or Google Cloud.

Ensure that your deployment environment is properly configured with the necessary dependencies.

Consider using tools like Gunicorn or uWSGI to serve your Django application in production.

Creating APIs:

Keep It Simple and Consistent:

Strive for simplicity in your API design. Use clear and intuitive endpoints and naming conventions.

Maintain consistency in the structure of your API across different endpoints.

Use HTTP Verbs Appropriately:

Follow RESTful principles. Use HTTP verbs like GET (for retrieving data), POST (for creating data), PUT/PATCH (for updating data), and DELETE (for removing data) appropriately.

Versioning:

Implement versioning from the beginning to ensure backward compatibility as your API evolves.

Error Handling:

Define clear and informative error responses with appropriate HTTP status codes and error messages.

Include error details in the response to assist developers in troubleshooting.

Authentication and Authorization:

Implement secure authentication mechanisms such as API keys, OAuth, or JWT tokens.

Enforce proper authorization to control access to specific endpoints.

Request and Response Format:

Use a consistent data format, such as JSON, for request and response bodies.

Validate and sanitize user inputs to prevent security vulnerabilities.

Implementing Versioning:

URL Versioning:
Include the version number in the URL, such as api/v1/endpoint. This makes it clear which version of the API a client is using.

Header Versioning:
Use the Accept header to specify the API version. For example, Accept: application/vnd.myapi.v1+json.

Namespace Versioning:
Place the version in the namespace or package structure of the code, allowing different versions to coexist.

Deprecation Policy:
Define a clear deprecation policy for older API versions, including a timeline for when they will be phased out.

Backward Compatibility:
Ensure that changes in newer API versions are backward compatible, so existing clients are not broken.

API Documentation:

Clear and Comprehensive Documentation:
Provide thorough documentation covering all endpoints, request parameters, and response structures.

Interactive Documentation:
Use tools like Swagger, Postman, or tools built into API frameworks like Django Rest Framework's built-in documentation to create interactive API documentation.

Example Usage:
Include examples of requests and responses to help developers understand how to use the API.

Authentication and Authorization:
Clearly explain the authentication methods required to access the API and how to obtain necessary credentials.

Rate Limiting and Quotas:
Inform users about rate limits and quotas, if applicable, to prevent abuse of the API.

Change Logs:
Maintain a change log to keep users informed of updates, new features, and deprecated endpoints.

Feedback Mechanism:
Provide a way for developers to give feedback or report issues with the API documentation.

Testing Environment:
Offer a sandbox or testing environment for developers to experiment with the API without affecting production data.

Version Switching:
Clearly display API version information and offer a way to switch between different versions when viewing the documentation.

Security Best Practices:
Include security recommendations and best practices to help developers protect their applications when using the API.

Remember that well-documented and versioned APIs enhance the developer experience and increase the likelihood of adoption and success for your API. Regularly update and maintain your documentation to keep it relevant and user-friendly.

This template provides a comprehensive structure for your software specifications document and business logic rules, ensuring that your team has a clear and organized roadmap for developing your e-commerce site. Tailor this template to your project's specific needs and objectives.

Here are some examples of how to use joins and views to improve the performance and usability of an e-commerce database:

Joins

- Inner Join:

SQL
```sql
SELECT p.productId, p.productName, b.brandName, c.categoryName,
s.subCategoryName
FROM tblProduct p
INNER JOIN tblProductBrand b ON p.prdBrandId = b.prdBrandId
INNER JOIN tblProductCategory c ON p.prdCategoryId = c.prdCategoryId
INNER JOIN tblProductSubCategory s ON p.prdSubCategoryId =
s.prdSubCategoryId;
```

This query will retrieve all product details along with their brand, category, and subcategory information in a single result set. This can improve performance by avoiding the need to make multiple queries to retrieve the related data.

- Left Join:

SQL
```sql
SELECT p.productId, p.productName, r.rwvReveiewContent, r.rwvRatings
FROM tblProduct p
LEFT JOIN tblReviews r ON p.productId = r.rwvProductIdfk;
```

This query will retrieve a list of all products, along with their associated reviews (if any). This can be useful for displaying product reviews on product pages.

- Self-Join:

SQL
```sql
SELECT m1.usrId, m1.usrName, m2.usrName AS managerName
FROM tblUser m1
LEFT JOIN tblUser m2 ON m1.usrIsManager = m2.usrId;
```

This query will create a self-join on the tblUser table to establish a hierarchy between managers and employees. This can be useful for displaying organizational charts and managing employee permissions.

Views

- Product Information View:

SQL
```
CREATE VIEW vwProductInfo AS
SELECT p.productId, p.productName, b.brandName, c.categoryName,
s.subCategoryName
FROM tblProduct p
INNER JOIN tblProductBrand b ON p.prdBrandId = b.prdBrandId
INNER JOIN tblProductCategory c ON p.prdCategoryId = c.prdCategoryId
INNER JOIN tblProductSubCategory s ON p.prdSubCategoryId =
s.prdSubCategoryId;
```
This view combines product details with brand, category, and subcategory information. This can make it easier to browse products by category or brand, and to display product information in a consistent format.

- Customer Details View:

SQL
```
CREATE VIEW vwCustomerDetails AS
SELECT c.customerId, c.custFirstName, c.custLastName, c.custOtherName,
c.custEmail, a.custStreetNumber, a.custHouseNumber, a.custGpsAddress,
co.custCountryName
FROM tblCustomer c
INNER JOIN tblCustomerAddress a ON c.custAddressId = a.custAddressId
INNER JOIN tblCustomerCountry co ON a.custCountryId = co.custCountryId;
```
This view provides comprehensive customer information, including their address and country details. This can be useful for customer management tasks such as creating and editing customer records, and generating customer reports.

- Order Summary View:

SQL
```
CREATE VIEW vwOrderSummary AS
SELECT o.ordId, o.ordDate, o.ordTotal, ps.paymentStatus
FROM tblOrder o
INNER JOIN tblPaymentStatus ps ON o.ordPaymentStatusID =
ps.paymentStatusId;
```
This view presents summarized order information, including customer name, order date, total amount, and payment status. This can be useful for displaying order history to customers, and for generating sales reports.

User Hierarchy View (if applicable)

SQL
```
CREATE VIEW vwUserHierarchy AS
WITH RECURSIVE manager_tree AS (
    SELECT usrId, usrName, usrIsManager
    FROM tblUser
    WHERE usrIsManager IS NULL
    UNION ALL
    SELECT u.usrId, u.usrName, u.usrIsManager
    FROM tblUser u
    JOIN manager_tree m ON u.usrIsManager = m.usrId
)
SELECT *
FROM manager_tree;
```
This view displays the organizational hierarchy, showing managers and their employees. This can be useful for managing employee permissions and reporting.

Inventory Status View:

SQL
```
CREATE VIEW vwInventoryStatus AS
SELECT p.productId, p.productName, i.invItemQuantity,
i.invItemReorderPoint, i.invItemReorderQuantity
FROM tblProduct p
INNER JOIN tblInventory i ON p.productId = i.invItemPrdIdfk;
```
This view shows current inventory status, including product name, quantity, and reorder information. This can be useful for monitoring inventory levels and ensuring that products are