Project: Product Store API - Requirements and Design

---

1. Project Overview

This document outlines the requirements and design standards for developing a **Product Store API**. The initial version of this API will allow users to perform basic product management operations, such as adding products, retrieving product details, updating product information, and deleting products. The **Product Store API** is designed as a backend service for managing products in an online store. In **Version 1**, it focuses solely on basic CRUD (-> Google) operations for products without user authentication or authorization.

Objective: Provide a basic RESTful API (-> Google) for product management to serve as a foundation for future enhancements.

2. Out of Scope for Version 1

- **Authentication or Authorization**:
  - All CRUD operations are publicly accessible.
  - No user authentication and token-based security.
  - No Role-based access control (Admins vs. Users).
- **Persistent data storage** (e.g., databases like SQLite).
  - simplifies the CRUD operations
  - Restart of the application lets you continue from a clean state

---

3. Functional Requirements

- **Product Management (CRUD Operations)**:
  - **Create Products**: Add new products with details such as name, price, category, and stock level.
  - **Read Products**: Retrieve the entire product catalog or details of individual products by ID.
  - **Update Products**: Modify product information such as name, price, category, and stock levels.
  - **Delete Products**: Remove products from the catalog.

---

4. Design Considerations

- **Data Modeling**:

  - Products will be stored in an in-memory Python dictionary with unique IDs as keys.
    - simpler -> helps in rapid prototyping

- **RESTful API Design**:

  - The API follows REST principles, using appropriate HTTP methods for CRUD operations:
    - `POST /products/` → Create a new product.
    - `GET /products/` → Retrieve all products.
    - `GET /products/{id}` → Retrieve a specific product by ID.
    - `PUT /products/{id}` → Update a product by ID.

  - `DELETE /products/{id}` → Delete a product by ID.

- **Error Handling**:
  - The API will return appropriate HTTP status codes and error messages for invalid requests (e.g., 400 Bad Request, 404 Not Found).

---

5. API Endpoints for Version 1

Method	Endpoint	Description	HTTP Status Codes
POST	`/products/`	Create a new product	201 Created, 400 Bad Request
GET	`/products/`	Retrieve all products	200 OK
GET	`/products/{id}`	Retrieve a specific product	200 OK, 404 Not Found
PUT	`/products/{id}`	Update a product	200 OK, 400 Bad Request, 404 Not Found
DELETE	`/products/{id}`	Delete a product	204 No Content, 404 Not Found

---

6. API Examples Version 1

Create a New Product

- **Endpoint**: `POST /products/`
- **Request Body**:

```
{
  "name": "Wireless Headphones",
  "price": 99.99,
  "category": "Electronics",
  "stock": 100
}
```

- **Response (201 Created)**:

```
{
  "id": 1,
  "name": "Wireless Headphones",
  "price": 99.99,
  "category": "Electronics",
  "stock": 100
}
```

Retrieve All Products

- **Endpoint**: `GET /products/`
- **Response (200 OK)**:

```
[
  {
    "id": 1,
    "name": "Wireless Headphones",
    "price": 99.99,
    "category": "Electronics",
    "stock": 100
  }
]
```

Retrieve a Specific Product

- **Endpoint**: `GET /products/1`
- **Response (200 OK)**:

```
{
  "id": 1,
  "name": "Wireless Headphones",
  "price": 99.99,
  "category": "Electronics",
  "stock": 100
}
```

- **Error Response (404 Not Found)**:

```
{
  "error": "Product not found"
}
```

Update a Product

- **Endpoint**: `PUT /products/1`
- **Request Body**:

```
{
  "name": "Wireless Earbuds",
  "price": 89.99
}
```

- **Response (200 OK)**:

```
{
  "id": 1,
  "name": "Wireless Earbuds",
  "price": 89.99,
  "category": "Electronics",
  "stock": 100
}
```

- **Error Response (404 Not Found)**:

```
{
  "error": "Product not found"
}
```

Delete a Product

- **Endpoint**: `DELETE /products/1`
- **Response (204 No Content)**
- **Error Response (404 Not Found)**:

```
{
  "error": "Product not found"
}
```

---

7. Technology Stack

- **Backend Framework**: FastAPI
    - A modern, high-performance web framework for building APIs with Python.
- **In-Memory Storage**:
    - Products will be stored in a Python dictionary for quick prototyping.
- **Testing Framework**: pytest
    - Used for writing automated tests to ensure API functionality.
- **API Testing Tool**: Postman
    - A API testing tool for manual testing.

---

8. Steps for Development

1. Set Up Project Structure

Create the following directory and file structure:

```
product_store_api/
├── src/
│   ├── __init__.py
│   ├── main.py
│   ├── models.py
│   ├── schemas.py
│   ├── crud.py
│   └── routes.py
├── tests/
├── requirements.txt
├── .gitignore
└── README.md
```

- `app/`: Contains all the application code.
- `__init__.py`: Indicates that `app/` is a Python package.
- `main.py`: Entry point of the application where the FastAPI instance is created.
- `models.py`: Defines data models (in-memory structures for Version 1).
- `schemas.py`: Contains Pydantic models for data validation and serialization.
- `crud.py`: Implements functions for CRUD operations.
- `routes.py`: Defines the API endpoints.
- `requirements.txt`: Lists project dependencies.
- `.gitignore`: Specifies files and directories for Git to ignore.
- `README.md`: Provides project documentation.

This structure is simply a recommendation. It might be useful to change files like `models.py` to a module `models/`.

2. Set Up Virtual Environment

- Ensure you have Python 3.11+ installed.
- Create a virtual environment:

  ```
  python -m venv venv
  ```

- Activate the virtual environment:
  - **On Linux**:

    ```
    source venv/bin/activate
    ```

- Install dependencies:

  ```
  pip install fastapi uvicorn pytest
  ```

- Freeze the installed packages:

```
pip freeze > requirements.txt
```

3. Initialize Git Repository

- Your Project should be under `https://gitlab.plri.de/<YOUR-NAME>/product-store-api.git`
- More information Here

4. Develop the In-Memory Database

- Implement product storage using a Python dictionary in `models.py` or `crud.py`.
- Example in `models.py`:

```
products_db = {}
```

5. Implement API Endpoints

- Define the API endpoints in `routes.py` according to the specifications in Section 5.
- Include proper error handling and data validation (Pydantic).

6. Testing

- Testing FastAPI Applications

- **Unit Tests**:

  - Test individual functions in `crud.py` for CRUD operations.
  - Ensure proper error handling and data validation.

- **Integration Tests**:

  - Test API endpoints in `routes.py` using FastAPI's test client.

- **Manual Testing**:

  - Use Thunder Client to manually test each endpoint with various inputs, including edge cases.

7. Documentation

- Utilize FastAPI's automatic documentation accessible at `http://localhost:8000/docs`.
- inform yourself about python documentation style and docstrings
- document code that is more difficult to understand
- describe your project in the `README.md` in your root directory
  - what is the project about?
  - what is your project structure?

-> the goal is to give a coworker an easy time to start developing in your project

---

9. Version 2: Persistent Storage and Authentication

Before starting version 2 of the application, get approval from Paulo/Luca about version 1 of your application.

- **Persistent Storage with SQLite and SQLAlchemy ORM**:
  - Replace in-memory storage with a SQLite database managed by SQLAlchemy ORM.
- **User Authentication with JWT**:
  - Implement user authentication using JWT for secure access to the API. FastAPI Security
- **Role-Based Access Control**:
  - Introduce user roles (Admins and Users) with different permissions.

Planned API Endpoints for Version 2

Method	Endpoint	Description
POST	`/auth/register`	Register a new user
POST	`/auth/login`	User login (returns a JWT token)
GET	`/products/`	Retrieve all products (authenticated users)
POST	`/products/`	Create a new product (admins only)
GET	`/products/{id}`	Retrieve a specific product (authenticated users)
PUT	`/products/{id}`	Update a product (admins only)
DELETE	`/products/{id}`	Delete a product (admins only)

API Examples for Version 2

In **Version 2**, the **Product Store API** introduces user authentication using JWT tokens and role-based access control, as well as persistent storage using SQLite with SQLAlchemy ORM. Below are detailed API examples demonstrating how to interact with the updated endpoints.

User Registration

- How do you make sure that not everybody with access to your application can register as an Store Admin?
- **Endpoint**: `POST /auth/register`
- **Description**: Registers a new user (either a Store User or Store Admin).

Request Body

```
{
  "username": "john_doe",
  "password": "securepassword123",
  "is_admin": false
}
```

- **Fields**:
  - `username` (string): Desired username (must be unique).
  - `password` (string): User's password (will be securely hashed).
  - `is_admin` (boolean, optional): Indicates if the user is an admin. Default is `false`.

Response (201 Created)

```
{
  "id": 1,
  "username": "john_doe",
  "is_admin": false
}
```

- **Fields**:
  - `id` (integer): Unique identifier for the user.
  - `username` (string): The registered username.
  - `is_admin` (boolean): Indicates the user's role.

Error Responses

- **409 Conflict**: Username already exists.

  ```
  {
    "error": "Username already registered"
  }
  ```

User Login

- **Endpoint**: `POST /auth/login`
- **Description**: Authenticates a user and returns a JWT access token.

Request Body

```
{
  "username": "john_doe",
  "password": "securepassword123"
}
```

Response (200 OK)

```
{
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9...",
  "token_type": "bearer"
}
```

- **Fields**:
    - `access_token` (string): The JWT token to be used for authenticated requests.
    - `token_type` (string): The token type, typically `bearer`.

Error Responses

- **401 Unauthorized**: Invalid credentials.

  ```
  {
    "error": "Incorrect username or password"
  }
  ```

Create a New Product (Admin Only)

- **Endpoint**: `POST /products/`
- **Description**: Allows an admin to add a new product to the store.
- **Authentication Required**: Yes (Admin role)

Request Header

```
Authorization: Bearer <access_token>
```

Request Body

```
{
  "name": "Wireless Earbuds",
  "price": 89.99,
  "category": "Electronics",
  "stock": 200
}
```

Response (201 Created)

```
{
  "id": 2,
  "name": "Wireless Earbuds",
  "price": 89.99,
  "category": "Electronics",
  "stock": 200
}
```

Error Responses

- **401 Unauthorized**: Missing or invalid JWT token.

  ```
  {
    "error": "Not authenticated"
  }
  ```

- **403 Forbidden**: Authenticated user is not an admin.

  ```
  {
    "error": "Not enough permissions"
  }
  ```

Retrieve All Products (Authenticated Users)

- **Endpoint**: `GET /products/`
- **Description**: Retrieves all available products.
- **Authentication Required**: Yes (User or Admin)

Request Header

```
Authorization: Bearer <access_token>
```

Response (200 OK)

```
[
  {
    "id": 1,
    "name": "Wireless Headphones",
    "price": 99.99,
    "category": "Electronics",
    "stock": 100
  },
```

```
  {
    "id": 2,
    "name": "Wireless Earbuds",
    "price": 89.99,
    "category": "Electronics",
    "stock": 200
  }
]
```

Error Responses

- **401 Unauthorized**: Missing or invalid JWT token.

  ```
  {
    "error": "Not authenticated"
  }
  ```

Retrieve a Specific Product (Authenticated Users)

- **Endpoint**: `GET /products/{id}`
- **Description**: Retrieves details of a specific product by ID.
- **Authentication Required**: Yes (User or Admin)

Request Header

```
Authorization: Bearer <access_token>
```

Response (200 OK)

```
{
  "id": 1,
  "name": "Wireless Headphones",
  "price": 99.99,
  "category": "Electronics",
  "stock": 100
}
```

Error Responses

- **401 Unauthorized**: Missing or invalid JWT token.

```json
{
  "error": "Not authenticated"
}
```

- **404 Not Found**: Product does not exist.

```json
{
  "error": "Product not found"
}
```

Update a Product (Admin Only)

- **Endpoint**: `PUT /products/{id}`
- **Description**: Allows an admin to update product details.
- **Authentication Required**: Yes (Admin role)

Request Header

```
Authorization: Bearer <access_token>
```

Request Body

```json
{
  "name": "Wireless Earbuds Pro",
  "price": 99.99,
  "stock": 180
}
```

- **Note**: Only the fields to be updated need to be included.

Response (200 OK)

```json
{
  "id": 2,
  "name": "Wireless Earbuds Pro",
  "price": 99.99,
  "category": "Electronics",
  "stock": 180
}
```

Error Responses

- **401 Unauthorized**: Missing or invalid JWT token.

  ```
  {
    "error": "Not authenticated"
  }
  ```

- **403 Forbidden**: Authenticated user is not an admin.

  ```
  {
    "error": "Not enough permissions"
  }
  ```

- **404 Not Found**: Product does not exist.

  ```
  {
    "error": "Product not found"
  }
  ```

Delete a Product (Admin Only)

- **Endpoint**: `DELETE /products/{id}`
- **Description**: Allows an admin to delete a product.
- **Authentication Required**: Yes (Admin role)

Request Header

```
Authorization: Bearer <access_token>
```

Response (204 No Content)

- **No response body**

Error Responses

- **401 Unauthorized**: Missing or invalid JWT token.

  ```
  {
    "error": "Not authenticated"
  }
  ```

- **403 Forbidden**: Authenticated user is not an admin.

```
{
  "error": "Not enough permissions"
}
```

- **404 Not Found**: Product does not exist.

```
{
  "error": "Product not found"
}
```

Attempting Unauthorized Access

Example: Non-Admin User Tries to Create a Product

- **Endpoint**: `POST /products/`
- **Request Header**:

```
Authorization: Bearer <access_token_of_non_admin_user>
```

- **Request Body**:

```
{
  "name": "Smart Watch",
  "price": 149.99,
  "category": "Electronics",
  "stock": 50
}
```

- **Error Response (403 Forbidden)**:

```
{
  "error": "Not enough permissions"
}
```

Example: Unauthenticated User Tries to Access Protected Route

- **Endpoint**: `GET /products/`

- **Request Header**:
  - **No Authorization header provided**
- **Error Response (401 Unauthorized)**:

  ```
  {
    "error": "Not authenticated"
  }
  ```

Token Expiration Handling

- **Scenario**: The user tries to access a protected route with an expired token.

Error Response (401 Unauthorized)

```
{
  "error": "Token has expired"
}
```

Refresh Token Mechanism (If Implemented)

- **Note**: If your application implements token refresh functionality, include the relevant endpoints and examples.

Invalid Token Handling

- **Scenario**: The user provides a malformed or invalid JWT token.

Error Response (401 Unauthorized)

```
{
  "error": "Could not validate credentials"
}
```

Input Validation Errors

- **Example**: Creating a product with missing required fields.

Request Body

```
{
  "price": 149.99,
  "category": "Electronics"
}
```

- **Error Response (422 Unprocessable Entity)**:

  ```
  {
    "error": [
      {
        "loc": ["body", "name"],
        "msg": "field required",
        "type": "value_error.missing"
      },
      {
        "loc": ["body", "stock"],
        "msg": "field required",
        "type": "value_error.missing"
      }
    ]
  }
  ```

Additional Notes

- **Authentication Workflow**:

  1. **Register**: A new user registers via `POST /auth/register`.
  2. **Login**: The user logs in via `POST /auth/login` to receive an `access_token`.
  3. **Authenticated Requests**: The user includes the `Authorization` header with the `Bearer` token in subsequent requests to protected endpoints.

- **Password Handling**:

  - Passwords are securely hashed using a hashing algorithm (e.g., bcrypt) before storing in the database.
  - Plain-text passwords are never stored or transmitted.

- **Role-Based Access Control**:

  - The user's role (`is_admin` flag) is encoded in the JWT token or retrieved from the database upon authentication.
  - Access to certain endpoints is restricted based on this role.

Summary of HTTP Status Codes

- **200 OK**: Successful retrieval or update of resources.
- **201 Created**: Successful creation of a resource.

- **204 No Content**: Successful deletion of a resource.
- **400 Bad Request**: Invalid request syntax or parameters.
- **401 Unauthorized**: Authentication is required and has failed or not been provided.
- **403 Forbidden**: The user does not have the necessary permissions.
- **404 Not Found**: The requested resource does not exist.
- **409 Conflict**: Conflict with the current state of the resource (e.g., duplicate username).
- **422 Unprocessable Entity**: The request was well-formed but was unable to be followed due to semantic errors (validation errors).

> Read more at https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

---

10. Test Plan for Version 2

This section outlines a comprehensive testing strategy for **Version 2** of the **Product Store API**, focusing on the new features introduced:

- User authentication with JWT tokens.
- Role-based access control (Store Users and Store Admins).
- Persistent storage using SQLite with SQLAlchemy ORM.
- Enhanced error handling and security measures.

The testing plan includes unit tests, integration tests, functional tests and security tests to ensure the application operates reliably and securely.

10.1 Testing Objectives

- **Authentication Verification**: Ensure user registration and login functionalities work correctly, and JWT tokens are properly issued, validated, and expired.
- **Role-Based Access Control**: Confirm that permissions are correctly enforced based on user roles (Admin vs. User).
- **Database Operations**: Validate that CRUD operations interact correctly with the SQLite database via SQLAlchemy ORM.
- **Security Assurance**: Test password hashing, token security, and protection against vulnerabilities like SQL injection and cross-site scripting (XSS).
- **Error Handling**: Verify that the API handles errors gracefully and returns appropriate HTTP status codes and messages.

10.2 Testing Scope

- **Functional Testing**: Testing individual functions and features for correctness.
- **Integration Testing**: Testing combined parts of the application to ensure they work together.
- **Security Testing**: Checking the application against security threats and vulnerabilities.

10.3 Testing Tools and Environment

- **Testing Framework**: `pytest`.

- **API Testing Tools**: **Thunder Client** or **Postman** for manual testing.
- **Database Inspection**: `Database` Plugin in VSCode or `sqlite3` command-line tool.
- **Mocking**: `unittest.mock` for mocking dependencies.
- **Continuous Integration**: GitLab CI/CD for automated testing.

Environment Setup:

- Python 3.11+
- FastAPI application running on localhost.
- SQLite database for data persistence.

10.4 Test Cases

10.4.1 Authentication Tests

Test Case 1: User Registration

- **Objective**: Verify that a new user can register successfully.
- **Steps**:
  1. Send a `POST` request to `/auth/register` with valid `username`, `password`, and `is_admin` fields.
- **Expected Result**:
  - Status code `201 Created`.
  - Response body contains user `id`, `username`, and `is_admin`.
- **Negative Tests**:
  - Register with an existing username (expect `409 Conflict`).
  - Register with missing fields (expect `422 Unprocessable Entity`).

Test Case 2: User Login

- **Objective**: Verify that a registered user can log in and receive a JWT token.
- **Steps**:
  1. Send a `POST` request to `/auth/login` with valid credentials.
- **Expected Result**:
  - Status code `200 OK`.
  - Response body contains `access_token` and `token_type`.
- **Negative Tests**:
  - Incorrect password (expect `401 Unauthorized`).
  - Non-existent username (expect `401 Unauthorized`).

Test Case 3: Token Validation

- **Objective**: Ensure protected endpoints require a valid JWT token.
- **Steps**:
  1. Access a protected endpoint without an `Authorization` header.
- **Expected Result**:
  - Status code `401 Unauthorized`.

- Error message indicating authentication is required.

10.4.2 Role-Based Access Control Tests

Test Case 4: Admin Access to Create Product

- **Objective**: Confirm that only admins can create products.
- **Steps**:
  1. Log in as an admin and obtain an `access_token`.
  2. Send a `POST` request to `/products/` with product data.
- **Expected Result**:
  - Status code `201 Created`.
  - Product is successfully created.
- **Negative Tests**:
  - Attempt as a non-admin (expect `403 Forbidden`).
  - Missing token (expect `401 Unauthorized`).

Test Case 5: User Access to View Products

- **Objective**: Ensure users can view products.
- **Steps**:
  1. Log in as a regular user.
  2. Send a `GET` request to `/products/`.
- **Expected Result**:
  - Status code `200 OK`.
  - Products are listed.

Test Case 6: Access Control Enforcement

- **Objective**: Verify that users cannot perform admin-only actions.
- **Steps**:
  1. Attempt to update or delete a product as a regular user.
- **Expected Result**:
  - Status code `403 Forbidden`.
  - Error message indicating insufficient permissions.

10.4.3 Database Operation Tests

Test Case 7: CRUD Operations

- **Objective**: Ensure database operations work correctly.
- **Steps**:
  1. **Create**: Add a new product.
  2. **Read**: Retrieve the product by ID.
  3. **Update**: Modify product details.
  4. **Delete**: Remove the product.
- **Expected Result**:
  - All operations succeed with appropriate status codes.

  - Database reflects changes accurately.

Test Case 8: Data Persistence

- **Objective**: Confirm data persists after application restarts.
- **Steps**:
  1. Create products.
  2. Restart the application.
  3. Verify products still exist.
- **Expected Result**:
  - Products are retained in the database.

10.4.4 Security Tests

Test Case 9: Password Hashing

- **Objective**: Ensure passwords are securely hashed.
- **Steps**:
  1. Register a new user.
  2. Inspect the database to verify password is hashed.
- **Expected Result**:
  - Passwords are stored as hashes, not plain text.

Test Case 10: Token Tampering

- **Objective**: Verify the system rejects tampered tokens.
- **Steps**:
  1. Alter a valid JWT token.
  2. Attempt to access a protected endpoint.
- **Expected Result**:
  - Status code `401 Unauthorized`.
  - Error message indicating token validation failure.

Test Case 11: SQL Injection Prevention

- **Objective**: Ensure protection against SQL injection.
- **Steps**:
  1. Attempt to inject SQL code via input fields (e.g., username).
- **Expected Result**:
  - Application handles input safely.
  - No unauthorized actions occur.

Test Case 12: XSS Protection

- **Objective**: Verify inputs are sanitized to prevent XSS attacks.
- **Steps**:
  1. Submit product data containing scripts or malicious code.
- **Expected Result**:

- Application sanitizes input.
- Malicious code is not executed.

10.4.5 Error Handling Tests

Test Case 13: Invalid Input Data

- **Objective**: Test API response to invalid data.
- **Steps**:
  1. Send requests with invalid data types or missing fields.
- **Expected Result**:
  - Status code `422 Unprocessable Entity`.
  - Detailed validation errors.

Test Case 14: Non-Existent Resources

- **Objective**: Ensure proper handling of missing resources.
- **Steps**:
  1. Access a product ID that doesn't exist.
- **Expected Result**:
  - Status code `404 Not Found`.
  - Error message indicating resource not found.

10.4.6 Additional Tests

Test Case 16: Token Expiration

- **Objective**: Verify expired tokens are rejected.
- **Steps**:
  1. Use a token after its expiration time.
- **Expected Result**:
  - Status code `401 Unauthorized`.
  - Error message indicating token expiration.

Test Case 17: Refresh Token (If Implemented)

- **Objective**: Test token refresh functionality.
- **Steps**:
  1. Obtain refresh token upon login.
  2. Use refresh token to get a new access token after expiration.
- **Expected Result**:
  - New access token is issued.
  - Old refresh token is invalidated.

10.5 Test Data

- **Users**:

  - **Admin User**:
    - `username`: "admin_user"
    - `password`: "AdminPass123"
    - `is_admin`: `true`
  - **Regular User**:
    - `username`: "regular_user"
    - `password`: "UserPass123"
    - `is_admin`: `false`
- **Products**:
  - **Product A**:
    - `name`: "Gaming Console"
    - `price`: 299.99
    - `category`: "Electronics"
    - `stock`: 150
  - **Product B**:
    - `name`: "Bluetooth Speaker"
    - `price`: 49.99
    - `category`: "Audio"
    - `stock`: 300

10.6 Testing Procedure

1. **Setup**:
   - Initialize the database.
   - Run the application in a testing environment.
2. **Execution**:
   - Run automated tests using `pytest`.
   - Perform manual tests using API testing tools.
3. **Verification**:
   - Verify responses match expected results.
   - Check database state after operations.
4. **Documentation**:
   - Record test results.
   - Log any defects or issues found.
5. **Cleanup**:
   - Reset the database to a clean state if necessary.

10.7 Test Automation

- **Automated Test Scripts**:
  - Write test cases in `tests/` directory using `pytest`.
- **Continuous Integration**:
  - Set up CI/CD pipelines in GitLab to run tests on code commits.
- **Code Coverage**:
  - Use `pytest-cov` to measure test coverage.

Note: Regular updates to the test plan and cases are essential as the application evolves. Ensure that new features are accompanied by corresponding tests to maintain the integrity and quality of the application.

---

11. Deploy

In Progress

- With Docker on our servers
- use PostgreSQL container

12. References

- **FastAPI Documentation**: https://fastapi.tiangolo.com/
- **pytest Documentation**: https://docs.pytest.org/en/stable/
- **Thunder Client Extension**: https://marketplace.visualstudio.com/items?itemName=rangav.vscode-thunder-client
- **GitHub .gitignore Templates**: https://github.com/github/gitignore
- **HTTP Response Status Codes**: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

---

Appendix A: Git Commands

1. **Initialize Git Repository**:

```
git init
```

2. **Create `.gitignore` File**:

```
touch .gitignore
```

3. **Create and Switch to New Branch**:

```
git switch -c version-1
```

4. **Add and Commit Changes**:

```
git add .
git commit -m "Initial commit for Version 1 - Project Skeleton"
```

5. **Set Up Remote Repository and Push**:

```
git remote add origin https://gitlab.plri.de/your-repo.git
git push -u origin version-1
```

6. **Switch Between Branches**:

```
git switch main
```

7. **Merge Branches**:

```
git merge version-1
```

> If you have any questions or need further clarification on any section, please feel free to reach out.