E-commerce Application on IBM Cloud Foundry

Phase 5: Project Documentation & Submission

In this part you will document your project and prepare it for submission.

Document the e-commerce platform project and prepare it for submission.

Documentation

- Outline the project's objective, design thinking process, and development phases.
- Describe the platform's layout, features, and technical implementation details.
- Include screenshots or images of the platform's user interface.

Submission

- Share the GitHub repository link containing the project's code and files.
- Provide instructions on how to deploy e-commerce platform on IBM Cloud Foundry.
- Write a detailed README file explaining how to navigate the website, update content, and any dependencies.

Documentation

Outline the project's objective, design thinking process, and development phases.

PROJECT: Artisanal E-commerce Application on IBM Cloud Foundry

Problem Definition:

The project is to build an artisanal e-commerce platform using IBM Cloud Foundry. The goal is to connect skilled artisans with a global audience, showcasing their handmade products and providing features like secure shopping carts, payment gateways, and an intuitive checkout process. This involves designing the e-commerce platform, implementing necessary features, and ensuring a seamless user experience.

Project Objective: Building an Artisanal E-Commerce Platform with IBM Cloud Foundry

The primary objective of this project is to create a robust and user-friendly e-commerce platform tailored to the needs of skilled artisans and their global audience. This platform will serve as a bridge, connecting artisans with a wide customer base while showcasing their unique, handcrafted products. Key components of this project include design, implementation, and a focus on delivering a seamless user experience.

1. **Empowering Artisans:** The project aims to empower artisans, offering them a digital marketplace to display their handmade creations. This platform will provide a global stage for artisans, enabling them to reach a broader audience, expand their businesses, and maintain their artistic freedom.

2. **E-Commerce Functionality:** The e-commerce platform will offer essential features such as secure shopping carts, payment gateways, and a straightforward checkout process. This will allow customers to browse, select, and purchase artisanal products with confidence.

3. **User Experience:** Ensuring an intuitive and satisfying user experience is a central objective. The platform will be designed with a user-centric approach, focusing on ease of navigation, clear product presentation, and efficient search and filter options. This will enhance the satisfaction of both artisans and customers.

4. **Global Reach:** The project's global focus aims to connect artisans and buyers from different corners of the world. It will support multiple languages, currencies, and payment methods to facilitate seamless international transactions.

5. **Security and Trust:** Security is a paramount concern. The project will incorporate robust security measures to protect customer data and ensure secure online transactions, fostering trust and confidence among users.

6. **Scalability and Reliability:** The e-commerce platform will be built on IBM Cloud Foundry, which offers scalability and reliability. The project's objective is to create a system that can grow with the increasing demand and remain available and responsive.

7. **Documentation:** A comprehensive set of documentation will be developed to aid in the platform's maintenance, user support, and future enhancements.

8. **Artisan Community:** Beyond being a marketplace, the platform will foster a sense of community among artisans, encouraging collaboration and knowledge sharing.

Design Thinking:

Design Thinking is a creative problem-solving approach that focuses on understanding the needs and preferences of users to design innovative solutions. It involves a series of iterative steps to empathize with users, define the problem, ideate potential solutions, prototype concepts, and test them.

1. Platform Design: Design the platform layout with sections for product categories, individual product pages, shopping cart, checkout, and payment.

Designing the layout of the artisanal e-commerce platform is a crucial step in creating a user-friendly and visually appealing website. Here, I'll outline a basic design for the platform's key sections: product categories, individual product pages, shopping cart, checkout, and payment.

Platform Layout Design

1. Home Page

 Header: The header should contain the platform's logo, navigation menu, and user login/registration options.

 Hero Section: A visually engaging section with high-quality images showcasing featured artisan products.

 Product Categories: Display a grid of product categories that users can explore. Each category should have an image and a brief description.

 Search Bar: Include a prominent search bar for users to find specific products quickly.

 Featured Artisans: Showcase profiles of featured artisans to encourage users to explore their products.

2. Product Category Page

 Category Header: Display the name of the selected category and a breadcrumb navigation trail for easy navigation.

 Product Grid: Arrange product listings in a grid format, including images, titles, prices, and average ratings. Use filters or sorting options for user convenience.

 Filter Sidebar: Provide filter options (e.g., price range, ratings, material type) to narrow down product search results.

 Pagination: If there are many products in a category, implement pagination to load products in batches.

3. Individual Product Page

 Product Details: Display detailed product information, including high-resolution images, product name, artisan details, price, and a product description.

 Add to Cart: Include a prominent "Add to Cart" button.

 Product Reviews: Allow users to read and leave reviews and ratings for the product.

 Related Products: Suggest related products or items frequently bought together.

4. Shopping Cart

 Cart Preview: Show a summary of items in the cart, including images, product names, quantities, and prices.

 Edit Cart: Enable users to adjust quantities, remove items, or continue shopping from the cart page.

 Proceed to Checkout: Include a clear button to take users to the checkout page.

5. Checkout

 Shipping Information: Collect user shipping details, including name, address, and contact information.

 Order Summary: Display a summary of the order, including product names, quantities, prices, and the total amount.

 Payment Options: Provide various payment options (credit/debit card, PayPal, etc.) and ensure a secure payment process.

 Promo Code: If applicable, allow users to enter promo codes for discounts.

 Place Order Button: A prominent button to confirm the order.

6. Payment

 Payment Form: A secure and user-friendly payment form where users can enter payment details.

 Order Confirmation: Display an order confirmation page with details and a confirmation number.

 Email Confirmation: Automatically send an email confirmation to the user with order details and a receipt.

Additional Considerations

 Mobile Responsiveness: Ensure that the platform is fully responsive, providing an optimal user experience on mobile devices.

 Navigation: Implement a consistent and intuitive navigation menu throughout the platform.

 User Accounts: Allow users to create accounts for a personalized shopping experience and order history.

 Security: Prioritize the security of user data and payment information.

Product Showcase:

Create a database to store product information such as images, descriptions, prices, and categories.

To create a database for storing product information, you can use a relational database management system (RDBMS) like PostgreSQL, MySQL, or SQLite. Below, I'll outline the steps to design a database schema for storing product information, including images, descriptions, prices, and categories.

Database Schema Design

1. Products Table

 product_id (Primary Key): A unique identifier for each product.

 product_name: The name or title of the product.

 description: A detailed description of the product.

 price: The price of the product.

 category_id (Foreign Key): A reference to the product's category.

 image_url: A URL or file path to the product's image.

 created_at: Timestamp indicating when the product was added to the database.

 updated_at: Timestamp indicating the last update to the product information.

2. Categories Table

 category_id (Primary Key): A unique identifier for each category.

 category_name: The name of the product category.

 created_at: Timestamp indicating when the category was added to the database.

 updated_at: Timestamp indicating the last update to the category information.

Database Relationships

 The **category_id** in the **Products** table serves as a foreign key that references the **category_id** in the **Categories** table. This establishes a one-to-many relationship, where one category can have multiple products associated with it.

Database Operations

With this database schema in place, you can perform various operations to manage product information, including:

Insertion: Add new products and categories to the database.

Retrieval: Retrieve product information by querying the database based on criteria such as category or product name.

Updates: Modify product or category information as needed.

Deletion: Remove products or categories from the database.

Additionally, you can implement data validation and constraints to ensure data integrity and security within the database.:

User Authentication:

Implement user registration and authentication features to enable artisans and customers to access the platform.

Implementing user registration and authentication features is crucial for the security and functionality of your artisanal e-commerce platform. Below, I'll outline the steps and components involved in setting up user registration and authentication.

User Authentication Components

1. **User Database**: You'll need a database to store user information securely. This can include tables for user profiles, credentials, and roles.

2. **Registration Page**: A web page where users can create an account by providing their details.

3. **Login Page**: A login form for users to enter their credentials and access their accounts.

4. **Session Management**: Mechanisms to create and manage user sessions after successful login.

5. **Password Security**: Safeguard user passwords by hashing and salting them before storage.

6. **User Roles**: Assign roles (e.g., artisan or customer) to users to determine their access and permissions.

Steps to Implement User Authentication

1. Database Setup

Create tables in your database to store user information. At a minimum, you'll typically have a **Users** table with columns such as **user_id**, **username**, **email**, **password_hash**, and **user_role**.

2. Registration Page

 Build a user-friendly registration form with fields like username, email, password, and user role (artisan or customer).

 Validate user inputs on the client-side and server-side to ensure data integrity.

 Implement CAPTCHA or anti-bot measures to prevent spam registrations.

3. Password Management

 Hash and salt user passwords before storing them in the database. Use a strong cryptographic hash function (e.g., bcrypt).

 Implement password recovery and reset mechanisms.

4. Login Page

 Create a login form where users can enter their credentials (username/email and password).

 Implement server-side validation and authentication logic.

 Set up sessions or JWT (JSON Web Tokens) for managing user sessions.

5. User Roles

 Determine the access and permissions for artisans and customers based on their roles.

 Implement role-based access control (RBAC) to restrict certain features or pages to specific user roles.

6. Authentication Middleware

 Implement middleware functions to check if a user is authenticated before allowing access to protected routes.

 Handle scenarios like expired sessions or revoked tokens.

7. User Profile

 Allow users to update their profile information (e.g., change password, update email).

 Artisans may have additional profile details like a bio and product listings.

8. Logging and Monitoring

 Implement logging mechanisms to record authentication and security-related events.

 Set up monitoring to detect suspicious activities (e.g., failed login attempts).

9. Security Considerations

 Protect against common web vulnerabilities like SQL injection and cross-site scripting (XSS).

 Implement rate limiting to prevent brute-force attacks.

 Keep software libraries and dependencies up-to-date to address security vulnerabilities.

10. Testing

 Thoroughly test the registration and authentication flows to ensure they work as expected.

 Conduct security testing (e.g., penetration testing) to identify and address vulnerabilities.

11. User Experience

 Ensure a smooth and user-friendly registration and login process with clear error messages.

 Implement password strength requirements and provide guidance to users.

12. Compliance

 Ensure compliance with data protection and privacy regulations (e.g., GDPR) by handling user data responsibly.

13. Documentation

 Document the authentication and registration process for future reference and maintenance.

Shopping Cart and Checkout:

Design and develop the shopping cart functionality and a smooth checkout process.

Designing and developing the shopping cart functionality and a seamless checkout process is crucial for a successful e-commerce platform. Below, I'll outline the key components and steps involved in creating a shopping cart and checkout system.

Shopping Cart Functionality

1. Shopping Cart Page

 Display Cart Contents: Show a list of items in the cart, including product names, quantities, prices, and a subtotal for each item.

 Allow Quantity Adjustment: Enable users to change the quantity of items or remove items from the cart.

 Update Cart: Include a button to update the cart when users change quantities.

 Calculate Total: Display the total cost of all items in the cart.

2. Add to Cart

 Product Pages: Add an "Add to Cart" button on individual product pages to add products to the cart.

 Confirmation: Provide feedback when an item is successfully added to the cart.

3. Cart Persistence

 User Sessions: Store the cart contents in the user's session to persist it across pages and visits.

 Guest Carts: Allow non-registered users to add items to a temporary cart (cookie-based) that can be converted to a registered user's cart upon login or registration.

Checkout Process

4. User Authentication

 Login or Guest Checkout: Allow users to log in or proceed as guests.

 Cart Association: If a guest user has items in their cart, associate those items with their account upon login or registration.

5. Shipping Information

 Shipping Address: Collect the user's shipping address, including name, street address, city, state, postal code, and country.

 Billing Address: If different from the shipping address, collect billing information.

6. Order Summary

 Review Cart: Show a summary of the cart contents, including product names, quantities, and prices.

 Shipping Costs: Calculate and display shipping costs based on the shipping address.

 Promo Codes: Allow users to enter promo codes for discounts.

7. Payment Information

 Payment Methods: Offer various payment methods (credit/debit card, PayPal, etc.).

 Secure Payment: Implement secure payment processing with encryption.

 Order Total: Display the final order total, including shipping and any discounts.

8. Order Confirmation

 Confirmation Page: Show an order confirmation page with details such as order number, items purchased, shipping address, and payment information.

 Email Confirmation: Send an email confirmation to the user with order details and a receipt.

9. Inventory Management

 Inventory Tracking: Ensure that product quantities are updated upon order placement to prevent overselling.

 Reserve Items: Reserve items in the cart for a limited time to give users a chance to complete the purchase.

10. Error Handling

 Validation: Implement validation to catch errors in user inputs.

 Error Messages: Provide clear error messages and guidance on how to resolve issues.

11. Order History

 User Accounts: If users have accounts, save order history for reference.

12. Security and Compliance

 Data Security: Ensure the security of user data and payment information.

 Compliance: Comply with data protection and privacy regulations (e.g., GDPR).

13. Testing

 Thorough Testing: Test the entire checkout process, including edge cases and error scenarios.

 Load Testing: Simulate heavy traffic to ensure the platform can handle peak loads.

14. Mobile Responsiveness

 Mobile-Friendly: Ensure that the entire checkout process is optimized for mobile devices.

15. Documentation

 Internal Documentation: Document the codebase and the checkout process for future maintenance.

 User Guidance: Provide user documentation or guidance on how to complete the checkout process.

Payment Integration:

Integrate secure payment gateways to facilitate transactions.

Integrating secure payment gateways is a critical component of any e-commerce platform. Payment gateways facilitate the secure processing of transactions, ensuring that customer data and financial information are protected. Below, I'll outline the steps to integrate secure payment gateways into your artisanal e-commerce platform.

1. Choose a Payment Gateway

Select a reliable and secure payment gateway provider that suits your needs. Common options include Stripe, PayPal, Square, Braintree, and Authorize.Net. Consider factors such as transaction fees, supported payment methods, and ease of integration.

2. Set Up an Account

Create an account with the chosen payment gateway provider. During this process, you'll need to provide business information and bank account details for fund transfers.

3. API Integration

Most payment gateway providers offer APIs and SDKs that allow you to integrate their services into your platform. Here's how to proceed:

 Developer Documentation: Review the provider's developer documentation thoroughly to understand the integration requirements and options.

 Sandbox/Test Environment: Create a sandbox or test environment provided by the gateway to test payment flows without processing real transactions. This is essential for debugging and testing.

 Integration Code: Integrate the payment gateway's API into your platform. This typically involves adding code for initializing the gateway, processing payments, and handling responses.

4. Secure Handling of Payment Data

Ensure the security of payment data by following best practices:

 Encryption: Use SSL/TLS encryption to secure data transmission between the user's browser and your server.

 Tokenization: Implement tokenization to replace sensitive payment data with secure tokens. Store these tokens instead of actual card numbers.

 PCI DSS Compliance: Comply with Payment Card Industry Data Security Standard (PCI DSS) requirements if you handle payment data directly. Many payment gateways offer PCI compliance solutions.

5. Payment Flow

Design a smooth payment flow for your users:

 Checkout Page: On the checkout page, present users with payment options and fields for entering payment details.

 Confirmation Page: After payment processing, display a confirmation page with order details.

6. Error Handling

Implement robust error handling to manage various scenarios:

 Validation Errors: Check for validation errors in user input on the checkout page.

 Payment Errors: Handle payment failures and provide clear error messages to users.

7. Testing

Thoroughly test the payment integration:

 Test Transactions: Use the sandbox environment to simulate test transactions for different scenarios (e.g., successful payments, declined payments, chargebacks).

 Functional Testing: Test all aspects of the payment flow, including validation, error handling, and order confirmation.

8. Go Live

Once you're confident in the payment integration and have thoroughly tested it, switch to the live environment to process real transactions. Ensure that your platform is configured to use the production API credentials provided by the gateway.

9. Monitoring and Maintenance

Regularly monitor payment transactions for issues, and keep your payment gateway integration up to date with any changes or updates from the provider.

10. Documentation

Document the payment integration process and any troubleshooting steps for future reference.

User Experience:

Focus on providing an intuitive and visually appealing user experience for both artisans and customers.

Creating an intuitive and visually appealing user experience (UX) is essential for the success of your artisanal e-commerce platform. A well-designed UX not only attracts and retains users but also enhances their satisfaction. Here are key considerations to achieve this:

1. User-Centered Design

 User Research: Understand the needs, preferences, and pain points of artisans and customers through surveys, interviews, and usability testing.

 User Personas: Create user personas to represent the typical artisans and customers who will use the platform. This helps in designing with specific user needs in mind.

2. Responsive Design

 Ensure that your platform is fully responsive, providing an optimal user experience on various devices and screen sizes (desktops, tablets, smartphones).

3. Clear Navigation

 Implement an intuitive and organized navigation menu that allows users to easily find products, categories, and essential features.

 Use breadcrumb trails to show users their location within the platform and help them navigate back to previous pages.

4. Visually Appealing Design

 Use a clean and visually appealing design with a consistent color scheme, typography, and branding elements.

 Utilize high-quality images to showcase products and artisans' work effectively.

5. Streamlined Registration and Onboarding

 Simplify the registration process for artisans and customers, asking only for necessary information.

 Provide a guided onboarding process that helps users set up their profiles and understand how to use the platform.

6. Product Discovery

 Implement powerful search functionality with filters (e.g., by category, price range, ratings) to help customers find products efficiently.

 Use recommendation algorithms to suggest relevant products to users based on their browsing and purchase history.

7. Product Presentation

 Design visually appealing product listings with high-quality images and concise, informative descriptions.

 Include user reviews and ratings to build trust and help customers make informed decisions.

8. Shopping Cart and Checkout

 Make the shopping cart easily accessible and visible to users throughout their browsing experience.

 Simplify the checkout process, requiring only essential information and steps to complete a purchase.

9. User Feedback and Reviews

 Encourage users to leave feedback and reviews, and provide a platform for artisans to respond to customer inquiries and comments.

 Use feedback to continuously improve the platform's user experience.

10. User Support

 Offer customer support channels, such as live chat, email, or a dedicated support page, to assist users with any issues or questions.

 Provide a comprehensive FAQ section to address common user queries.

11. Performance Optimization

 Optimize page loading times to ensure a smooth and responsive user experience.

 Minimize the use of large media files or unnecessary scripts that could slow down the platform.

12. Accessibility

 Ensure that the platform is accessible to users with disabilities by following web accessibility guidelines (e.g., WCAG).

 Implement alt text for images, keyboard navigation, and other accessibility features.

13. A/B Testing

 Conduct A/B testing to compare different design and user experience elements to determine which ones work best for your audience.

 Use analytics to track user behavior and make data-driven design decisions.

14. Feedback Loops

 Create mechanisms for users to provide feedback directly through the platform.

 Act on user feedback to continuously improve the user experience.

15. Training and Documentation

 Provide clear and concise documentation and tutorials for both artisans and customers to learn how to use the platform effectively.

Development phase:

CODE BASE:

DATABASE.PY

(database managing python file)

```
import sqlite3

#Open database
conn = sqlite3.connect('C:/Users/admin/Desktop/database/E_commerce_database.db')

#Create table
conn.execute('''CREATE TABLE users
        (userId INTEGER PRIMARY KEY,
        password TEXT,
        email TEXT,
        firstName TEXT,
        lastName TEXT,
        address1 TEXT,
        address2 TEXT,
        zipcode TEXT,
        city TEXT,
        state TEXT,
        country TEXT,
        phone TEXT
        )''')
```

```
conn.execute('''CREATE TABLE products
        (productId INTEGER PRIMARY KEY,
        name TEXT,
        price REAL,
        description TEXT,
        image TEXT,
        stock INTEGER,
        categoryId INTEGER,
        FOREIGN KEY(categoryId) REFERENCES categories(categoryId)
        )''')

conn.execute('''CREATE TABLE kart
        (userId INTEGER,
        productId INTEGER,
        FOREIGN KEY(userId) REFERENCES users(userId),
        FOREIGN KEY(productId) REFERENCES products(productId)
        )''')

conn.execute('''CREATE TABLE categories
        (categoryId INTEGER PRIMARY KEY,
        name TEXT
        )''')



conn.close()
```

MAIN.PY

(Main application file)

```
from flask import *
import sqlite3, hashlib, os
from werkzeug.utils import secure_filename

app = Flask(__name__)
app.secret_key = 'random string'
UPLOAD_FOLDER = 'static/uploads'
ALLOWED_EXTENSIONS = set(['jpeg', 'jpg', 'png', 'gif'])
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

def getLoginDetails():
    with sqlite3.connect('database.db') as conn:
```

```
        cur = conn.cursor()
        if 'email' not in session:
            loggedIn = False
            firstName = ''
            noOfItems = 0
        else:
            loggedIn = True
            cur.execute("SELECT userId, firstName FROM users WHERE email = ?",
(session['email'], ))
            userId, firstName = cur.fetchone()
            cur.execute("SELECT count(productId) FROM kart WHERE userId = ?",
(userId, ))
            noOfItems = cur.fetchone()[0]
    conn.close()
    return (loggedIn, firstName, noOfItems)

@app.route("/")
def root():
    loggedIn, firstName, noOfItems = getLoginDetails()
    with sqlite3.connect('database.db') as conn:
        cur = conn.cursor()
        cur.execute('SELECT productId, name, price, description, image, stock
FROM products')
        itemData = cur.fetchall()
        cur.execute('SELECT categoryId, name FROM categories')
        categoryData = cur.fetchall()
    itemData = parse(itemData)
    return render_template('home.html', itemData=itemData, loggedIn=loggedIn,
firstName=firstName, noOfItems=noOfItems, categoryData=categoryData)

@app.route("/add")
def admin():
    with sqlite3.connect('database.db') as conn:
        cur = conn.cursor()
        cur.execute("SELECT categoryId, name FROM categories")
        categories = cur.fetchall()
    conn.close()
    return render_template('add.html', categories=categories)

@app.route("/addItem", methods=["GET", "POST"])
def addItem():
    if request.method == "POST":
        name = request.form['name']
        price = float(request.form['price'])
        description = request.form['description']
```

```
        stock = int(request.form['stock'])
        categoryId = int(request.form['category'])

        #Uploading image procedure
        image = request.files['image']
        if image and allowed_file(image.filename):
            filename = secure_filename(image.filename)
            image.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
        imagename = filename
        with sqlite3.connect('database.db') as conn:
            try:
                cur = conn.cursor()
                cur.execute('''INSERT INTO products (name, price, description,
image, stock, categoryId) VALUES (?, ?, ?, ?, ?, ?)''', (name, price,
description, imagename, stock, categoryId))
                conn.commit()
                msg="added successfully"
            except:
                msg="error occured"
                conn.rollback()
        conn.close()
        print(msg)
        return redirect(url_for('root'))

@app.route("/remove")
def remove():
    with sqlite3.connect('database.db') as conn:
        cur = conn.cursor()
        cur.execute('SELECT productId, name, price, description, image, stock
FROM products')
        data = cur.fetchall()
    conn.close()
    return render_template('remove.html', data=data)

@app.route("/removeItem")
def removeItem():
    productId = request.args.get('productId')
    with sqlite3.connect('database.db') as conn:
        try:
            cur = conn.cursor()
            cur.execute('DELETE FROM products WHERE productID = ?', (productId,
))
            conn.commit()
            msg = "Deleted successsfully"
        except:
```

```
            conn.rollback()
            msg = "Error occured"
    conn.close()
    print(msg)
    return redirect(url_for('root'))

@app.route("/displayCategory")
def displayCategory():
        loggedIn, firstName, noOfItems = getLoginDetails()
        categoryId = request.args.get("categoryId")
        with sqlite3.connect('database.db') as conn:
            cur = conn.cursor()
            cur.execute("SELECT products.productId, products.name,
products.price, products.image, categories.name FROM products, categories WHERE
products.categoryId = categories.categoryId AND categories.categoryId = ?",
(categoryId, ))
            data = cur.fetchall()
        conn.close()
        categoryName = data[0][4]
        data = parse(data)
        return render_template('displayCategory.html', data=data,
loggedIn=loggedIn, firstName=firstName, noOfItems=noOfItems,
categoryName=categoryName)

@app.route("/account/profile")
def profileHome():
    if 'email' not in session:
        return redirect(url_for('root'))
    loggedIn, firstName, noOfItems = getLoginDetails()
    return render_template("profileHome.html", loggedIn=loggedIn,
firstName=firstName, noOfItems=noOfItems)

@app.route("/account/profile/edit")
def editProfile():
    if 'email' not in session:
        return redirect(url_for('root'))
    loggedIn, firstName, noOfItems = getLoginDetails()
    with sqlite3.connect('database.db') as conn:
        cur = conn.cursor()
        cur.execute("SELECT userId, email, firstName, lastName, address1,
address2, zipcode, city, state, country, phone FROM users WHERE email = ?",
(session['email'], ))
        profileData = cur.fetchone()
    conn.close()
```

```
    return render_template("editProfile.html", profileData=profileData,
loggedIn=loggedIn, firstName=firstName, noOfItems=noOfItems)

@app.route("/account/profile/changePassword", methods=["GET", "POST"])
def changePassword():
    if 'email' not in session:
        return redirect(url_for('loginForm'))
    if request.method == "POST":
        oldPassword = request.form['oldpassword']
        oldPassword = hashlib.md5(oldPassword.encode()).hexdigest()
        newPassword = request.form['newpassword']
        newPassword = hashlib.md5(newPassword.encode()).hexdigest()
        with sqlite3.connect('database.db') as conn:
            cur = conn.cursor()
            cur.execute("SELECT userId, password FROM users WHERE email = ?",
(session['email'], ))
            userId, password = cur.fetchone()
            if (password == oldPassword):
                try:
                    cur.execute("UPDATE users SET password = ? WHERE userId = ?",
(newPassword, userId))
                    conn.commit()
                    msg="Changed successfully"
                except:
                    conn.rollback()
                    msg = "Failed"
                return render_template("changePassword.html", msg=msg)
            else:
                msg = "Wrong password"
        conn.close()
        return render_template("changePassword.html", msg=msg)
    else:
        return render_template("changePassword.html")

@app.route("/updateProfile", methods=["GET", "POST"])
def updateProfile():
    if request.method == 'POST':
        email = request.form['email']
        firstName = request.form['firstName']
        lastName = request.form['lastName']
        address1 = request.form['address1']
        address2 = request.form['address2']
        zipcode = request.form['zipcode']
        city = request.form['city']
        state = request.form['state']
```

```
        country = request.form['country']
        phone = request.form['phone']
        with sqlite3.connect('database.db') as con:
                try:
                    cur = con.cursor()
                    cur.execute('UPDATE users SET firstName = ?, lastName = ?,
address1 = ?, address2 = ?, zipcode = ?, city = ?, state = ?, country = ?, phone
= ? WHERE email = ?', (firstName, lastName, address1, address2, zipcode, city,
state, country, phone, email))

                    con.commit()
                    msg = "Saved Successfully"
                except:
                    con.rollback()
                    msg = "Error occured"
        con.close()
        return redirect(url_for('editProfile'))

@app.route("/loginForm")
def loginForm():
    if 'email' in session:
        return redirect(url_for('root'))
    else:
        return render_template('login.html', error='')

@app.route("/login", methods = ['POST', 'GET'])
def login():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        if is_valid(email, password):
            session['email'] = email
            return redirect(url_for('root'))
        else:
            error = 'Invalid UserId / Password'
            return render_template('login.html', error=error)

@app.route("/productDescription")
def productDescription():
    loggedIn, firstName, noOfItems = getLoginDetails()
    productId = request.args.get('productId')
    with sqlite3.connect('database.db') as conn:
        cur = conn.cursor()
        cur.execute('SELECT productId, name, price, description, image, stock
FROM products WHERE productId = ?', (productId, ))
```

```
        productData = cur.fetchone()
    conn.close()
    return render_template("productDescription.html", data=productData, loggedIn
= loggedIn, firstName = firstName, noOfItems = noOfItems)

@app.route("/addToCart")
def addToCart():
    if 'email' not in session:
        return redirect(url_for('loginForm'))
    else:
        productId = int(request.args.get('productId'))
        with sqlite3.connect('database.db') as conn:
            cur = conn.cursor()
            cur.execute("SELECT userId FROM users WHERE email = ?",
(session['email'], ))
            userId = cur.fetchone()[0]
            try:
                cur.execute("INSERT INTO kart (userId, productId) VALUES (?, ?)",
(userId, productId))
                conn.commit()
                msg = "Added successfully"
            except:
                conn.rollback()
                msg = "Error occured"
        conn.close()
        return redirect(url_for('root'))

@app.route("/cart")
def cart():
    if 'email' not in session:
        return redirect(url_for('loginForm'))
    loggedIn, firstName, noOfItems = getLoginDetails()
    email = session['email']
    with sqlite3.connect('database.db') as conn:
        cur = conn.cursor()
        cur.execute("SELECT userId FROM users WHERE email = ?", (email, ))
        userId = cur.fetchone()[0]
        cur.execute("SELECT products.productId, products.name, products.price,
products.image FROM products, kart WHERE products.productId = kart.productId AND
kart.userId = ?", (userId, ))
        products = cur.fetchall()
    totalPrice = 0
    for row in products:
        totalPrice += row[2]
```

```
    return render_template("cart.html", products = products,
totalPrice=totalPrice, loggedIn=loggedIn, firstName=firstName,
noOfItems=noOfItems)

@app.route("/removeFromCart")
def removeFromCart():
    if 'email' not in session:
        return redirect(url_for('loginForm'))
    email = session['email']
    productId = int(request.args.get('productId'))
    with sqlite3.connect('database.db') as conn:
        cur = conn.cursor()
        cur.execute("SELECT userId FROM users WHERE email = ?", (email, ))
        userId = cur.fetchone()[0]
        try:
            cur.execute("DELETE FROM kart WHERE userId = ? AND productId = ?",
(userId, productId))
            conn.commit()
            msg = "removed successfully"
        except:
            conn.rollback()
            msg = "error occured"
    conn.close()
    return redirect(url_for('root'))

@app.route("/logout")
def logout():
    session.pop('email', None)
    return redirect(url_for('root'))

def is_valid(email, password):
    con = sqlite3.connect('database.db')
    cur = con.cursor()
    cur.execute('SELECT email, password FROM users')
    data = cur.fetchall()
    for row in data:
        if row[0] == email and row[1] ==
hashlib.md5(password.encode()).hexdigest():
            return True
    return False

@app.route("/register", methods = ['GET', 'POST'])
def register():
    if request.method == 'POST':
        #Parse form data
```

```
        password = request.form['password']
        email = request.form['email']
        firstName = request.form['firstName']
        lastName = request.form['lastName']
        address1 = request.form['address1']
        address2 = request.form['address2']
        zipcode = request.form['zipcode']
        city = request.form['city']
        state = request.form['state']
        country = request.form['country']
        phone = request.form['phone']

        with sqlite3.connect('database.db') as con:
            try:
                cur = con.cursor()
                cur.execute('INSERT INTO users (password, email, firstName,
lastName, address1, address2, zipcode, city, state, country, phone) VALUES (?, ?,
?, ?, ?, ?, ?, ?, ?, ?, ?)', (hashlib.md5(password.encode()).hexdigest(), email,
firstName, lastName, address1, address2, zipcode, city, state, country, phone))

                con.commit()

                msg = "Registered Successfully"
            except:
                con.rollback()
                msg = "Error occured"
        con.close()
        return render_template("login.html", error=msg)

@app.route("/registerationForm")
def registrationForm():
    return render_template("register.html")

def allowed_file(filename):
    return '.' in filename and \
            filename.rsplit('.', 1)[1] in ALLOWED_EXTENSIONS

def parse(data):
    ans = []
    i = 0
    while i < len(data):
        curr = []
        for j in range(7):
            if i >= len(data):
                break
```

```
            curr.append(data[i])
            i += 1
        ans.append(curr)
    return ans

if __name__ == '__main__':
    app.run(debug=True)
```

HTML TEMPLATES:

HOME.HTML

```
<!DOCTYPE HTML>
<html>
<head>
<title>Made with Passion</title>
<link rel="stylesheet" href={{ url_for('static', filename='css/home.css') }} />
<link rel="stylesheet" href={{ url_for('static', filename='css/topStyle.css') }}
/>
</head>
<body>
<div id="title">
    <a href="#">
        <img id="logo" src= {{ url_for('static', filename='images/logo.png') }}
/>
    </a>
    <form>
        <input id="searchBox" type="text" name="searchQuery">
        <input id="searchButton" type="submit" value="Search">
    </form>

    {% if not loggedIn %}
    <div id="signInButton">
        <a class="link" href="/loginForm">Sign In</a>
    </div>
    {% else %}
    <div class="dropdown">
        <button class="dropbtn">Hello, <br>{{firstName}}</button>
        <div class="dropdown-content">
            <a href="/account/orders">Your orders</a>
            <a href="/account/profile">Your profile</a>
            <hr>
            <a href="/logout">Sign Out</a>
```

```
        </div>
    </div>
    {% endif %}
    <div id="kart">
        <a class="link" href="/cart">
            <img src={{url_for('static', filename='images/shoppingCart.png')}}
id="cartIcon" />
            CART {{noOfItems}}
        </a>
    </div>
</div>
<div class="display">
    <div class="displayCategory">
        <h2>Shop by Category: </h2>
        <ul>
            {% for row in categoryData %}
            <li><a
href="/displayCategory?categoryId={{row[0]}}">{{row[1]}}</a></li>
            {% endfor %}
        </ul>
    </div>
    <div>
        <h2 style="margin-left: 230px;">Items</h2>
        {% for data in itemData %}
        <table>
            <tr id="productName">
                {% for row in data %}
                <td>
                     {{row[1]}}
                </td>
                {% endfor %}
            </tr>
            <tr id="productImage">
                {% for row in data %}
                <td>
                    <a href="/productDescription?productId={{row[0]}}">
                        <img src={{ url_for('static', filename='uploads/' +
row[4]) }} id="itemImage" />
                    </a>
                </td>
                {% endfor %}
            </tr>
            <tr id="productPrice">
                {% for row in data %}
                <td>
```

```
                    ₹ {{row[2]}}
                </td>
                {% endfor %}
            </tr>
        </table>
        {% endfor %}
    </div>
</div>
</body>
</html>
```

LOGIN.HTML

```
<html>
<head>
<title> First flask app </title>
</head>
<body style="background-color: #a6f0eb;">
    <center><p> Login</p><br>
<p> {{error}} </p>
<form action="/login" method="POST">
    <p>Email: <input type="text" name="email"></p>
    <p>Password: <input type="password" name="password"></p>
    <p><input type="submit"></p>
    <a href="/registerationForm">Register here</a>
</form>
</center>
</body>
</html>
```

PRODUCTDESCRIPTION.HTML

```
<!DOCTYPE HTML>
<html>
<head>
<title>Product Description</title>
```

```
<link rel="stylesheet" href={{url_for('static',
filename='css/productDescription.css')}} />
<link rel="stylesheet" href={{ url_for('static', filename='css/topStyle.css')}}
/>
</head>
<body>
<div id="title">
    <a href="/">
        <img id="logo" src= {{ url_for('static', filename='images/logo.png') }}
/>
    </a>
    <form>
        <input id="searchBox" type="text" name="searchQuery">
        <input id="searchButton" type="submit" value="Search">
    </form>

    {% if not loggedIn %}
    <div id="signInButton">
        <a class="link" href="/loginForm">Sign In</a>
    </div>
    {% else %}
    <div class="dropdown">
        <button class="dropbtn">Hello, <br>{{firstName}}</button>
        <div class="dropdown-content">
            <a href="/account/orders">Your orders</a>
            <a href="/account/profile">Your profile</a>
            <hr>
            <a href="/logout">Sign Out</a>
        </div>
    </div>
    {% endif %}
    <div id="kart">
        <a class="link" href="/cart">
            <img src={{url_for('static', filename='images/shoppingCart.png')}}
id="cartIcon" />
            CART {{noOfItems}}
        </a>
    </div>
</div>
<div id="display">
    <div id="productName">
        <h1>{{data[1]}}</h1>
    </div>
    <div>
```

```
        <img src={{url_for('static', filename='uploads/'+data[4]) }}
id="productImage"/>
    </div>

    <div id="productDescription">
        <h2>Details</h2>
        <table id="descriptionTable">
            <tr>
                <td>Name</td>
                <td>{{data[1]}}</td>
            </tr>
            <tr>
                <td>Price</td>
                <td>${{data[2]}}</td>
            </tr>
            <tr>
                <td>Stock</td>
                <td>{{data[5]}}</td>
            </tr>
        </table>
        <h2>Description</h2>
        <p>{{data[3]}}</p>
    </div>
    <div id="addToCart">
        <a href="/addToCart?productId={{request.args.get('productId')}}">Add to
Cart</a>
    </div>
</div>
</body>
</html>
```

PROFILEHOME.HTML

```
<!DOCTYPE HTML>
<html>
<head>
<title>Profile Home</title>
<link rel="stylesheet" href={{ url_for('static', filename='css/profileHome.css')
}} />
<link rel="stylesheet" href={{ url_for('static', filename='css/topStyle.css') }}
/>
```

```
</head>
<body>
<div id="title">
    <a href="/">
        <img id="logo" src= {{ url_for('static', filename='images/logo.png') }}
/>
    </a>
    <form>
        <input id="searchBox" type="text" name="searchQuery">
        <input id="searchButton" type="submit" value="Search">
    </form>

    {% if not loggedIn %}
    <div id="signInButton">
        <a class="link" href="/loginForm">Sign In</a>
    </div>
    {% else %}
    <div class="dropdown">
        <button class="dropbtn">Hello, <br>{{firstName}}</button>
        <div class="dropdown-content">
            <a href="/account/orders">Your orders</a>
            <a href="/account/profile">Your profile</a>
            <hr>
            <a href="/logout">Sign Out</a>
        </div>
    </div>
    {% endif %}
    <div id="kart">
        <a class="link" href="/cart">
            <img src={{url_for('static', filename='images/shoppingCart.png')}}
id="cartIcon" />
            CART {{noOfItems}}
        </a>
    </div>
</div>

<div class="display">
    <a href="/account/profile/view">View Profile</a><br>
    <a href="/account/profile/edit">Edit Profile</a><br>
    <a href="/account/profile/changePassword">Change password</a>
</div>
</body>
</html>
```

REGISTER.HTML

```
<html>
<head>
<title>Registration</title>
<link rel="stylesheet" href={{ url_for('static', filename='css/remove.css') }} />
<script type="text/javascript" src="{{ url_for('static', filename =
'js/validateForm.js') }}">
</script>
</head>
<body style="background-color: #a6f0eb;">
    <center>
    <p style="font-size: 60 px;">SIGNIN</p><br>
    <form action="/register" method="POST" onsubmit="return validate()">
    <p>Email: <input type="email" name="email"></p>

    <P>Password: <input type="password" name="password" id="password"
required></p>

    <p>Confirm Password: <input type="password" name="cpassword"
id="cpassword"></p>

    <p>First Name: <input type="text" name="firstName"></p>
    <p>Last Name: <input type="text" name="lastName"></p>
    <p>Address Line 1: <input type="text" name="address1"></p>
    <p>Address Line 2: <input type="text" name="address2"></p>
    <p>Zipcode: <input type="text" name="zipcode"></p>
    <p>City: <input type="text" name="city"></p>
    <p>State: <input type="text" name="state"></p>
    <p>Country: <input type="text" name="country"></p>
    <p>Phone Number: <input type="text" name="phone"></p>

    <p><input type="submit" value="Register"></p>

    <a href="/loginForm">Login here</a>
</form>
</center>
</body>
</html>
```

REMOVE.HTML

```
<!DOCTYPE HTML>
<html>
<head>
<title>Remove</title>
<link rel="stylesheet" href={{url_for('static', filename='css/remove.css')}}>
</link>
</head>
<body>
<table>
    {% for i in range(6) %}
    <tr>
        {% for row in data %}
        <td>
            <a href="/removeItem?productId={{row[0]}}">
            {% if i == 4 %}
                <img src={{ url_for('static', filename='uploads/' + row[i]) }}
id="itemImage" />
            {% else %}
                {{row[i]}}
            {% endif %}
            </a>
        </td>
        {% endfor %}
    </tr>
    {% endfor %}
</table>
</body>
</html>
```

EDITPROFILE.HTML

```
<!DOCTYPE HTML>
<html>
<head>
<title>Edit Profile </title>
<link rel="stylesheet" href={{ url_for('static', filename='css/editProfile.css')
}} />
<link rel="stylesheet" href={{ url_for('static', filename='css/topStyle.css') }}
/>
```

```
</head>
<body>
<div id="title">
    <a href="/">
        <img id="logo" src= {{ url_for('static', filename='images/logo.png') }}
/>
    </a>
    <form>
        <input id="searchBox" type="text" name="searchQuery">
        <input id="searchButton" type="submit" value="Search">
    </form>

    {% if not loggedIn %}
    <div id="signInButton">
        <a class="link" href="/loginForm">Sign In</a>
    </div>
    {% else %}
    <div class="dropdown">
        <button class="dropbtn">Hello, <br>{{firstName}}</button>
        <div class="dropdown-content">
            <a href="/account/orders">Your orders</a>
            <a href="/account/profile">Your profile</a>
            <hr>
            <a href="/logout">Sign Out</a>
        </div>
    </div>
    {% endif %}
    <div id="kart">
        <a class="link" href="/cart">
            <img src={{url_for('static', filename='images/shoppingCart.png')}}
id="cartIcon" />
            CART {{noOfItems}}
        </a>
    </div>
</div>

<div class="display">
    <h2>Edit profile</h2>
    <form action="/updateProfile" method="POST">
        <p>Email:<input type="email" name="email" value={{profileData[1]}}
readonly="readonly"></p>
        <p>First Name:<input type="text" name="firstName"
value={{profileData[2]}}></p>
        <p>Last Name: <input type="text" name="lastName"
value={{profileData[3]}}></p>
```

```
        <p>Address 1: <input type="text" name="address1"
value={{profileData[4]}}></p>
        <p>Address 2: <input type="text" name="address2"
value={{profileData[5]}}></p>
        <p>Zip Code: <input type="text" name="zipcode"
value={{profileData[6]}}></p>
        <p>City: <input type="text" name="city" value={{profileData[7]}}></p>
        <p>State: <input type="text" name="state" value={{profileData[8]}}></p>
        <p>Country: <input type="text" name="country"
value={{profileData[9]}}></p>
        <p>Phone Number: <input type="text" name="phone"
value={{profileData[10]}}></p>
        <input type="submit" value="Save">
    </form>
</div>
</body>
</html>
```

DISPLAYCATRGORY.HTML

```
<!DOCTYPE HTML>
<html>
<head>
<title>Category: {{categoryName}}</title>
<link rel="stylesheet" href={{ url_for('static', filename='css/home.css') }} />
<link rel="stylesheet" href={{ url_for('static', filename='css/topStyle.css') }}
/>
</head>
<body>
<div id="title">
    <a href="/">
        <img id="logo" src= {{ url_for('static', filename='images/logo.png') }}
/>
    </a>
    <form>
        <input id="searchBox" type="text" name="searchQuery">
        <input id="searchButton" type="submit" value="Search">
    </form>

    {% if not loggedIn %}
    <div id="signInButton">
```

```
        <a class="link" href="/loginForm">Sign In</a>
    </div>
    {% else %}
    <div class="dropdown">
        <button class="dropbtn">Hello, <br>{{firstName}}</button>
        <div class="dropdown-content">
            <a href="/account/orders">Your orders</a>
            <a href="/account/profile">Your profile</a>
            <hr>
            <a href="/logout">Sign Out</a>
        </div>
    </div>
    {% endif %}
    <div id="kart">
        <a class="link" href="/cart">
            <img src={{url_for('static', filename='images/shoppingCart.png')}}
id="cartIcon" />
            CART {{noOfItems}}
        </a>
    </div>
</div>

<div>
    <h2>Showing all products of Category {{categoryName}}:</h2>
    {% for itemData in data %}
    <table>
        <tr id="productName">
            {% for row in itemData %}
            <td>
                {{row[1]}}
            </td>
            {% endfor %}
        </tr>
        <tr id="productImage">
            {% for row in itemData %}
            <td>
                <a href="/productDescription?productId={{row[0]}}">
                    <img src={{ url_for('static', filename='uploads/' + row[3])
}} id="itemImage" />
                </a>
            </td>
            {% endfor %}
        </tr>
        <tr id="productPrice">
            {% for row in itemData %}
```

```
            <td>
                ${{row[2]}}
            </td>
            {% endfor %}
        </tr>
    </table>
    {% endfor %}
</div>
</body>
</html>
```

CHANGEPASSWORD.HTML

```
<html>
<head>
<title>Change Password</title>
<script src={{ url_for('static', filename='js/changePassword.js') }}></script>
</head>
<body>
<h1>Change password</h1>
<p>{{ msg }}</p>
<form action={{ url_for('changePassword') }} method="POST" onsubmit="return
validate()">
    <p>Old Password: <input type="password" name="oldpassword"></p>
    <p>New Password: <input type="password" name="newpassword"
id="newpassword"></p>
    <p>Confirm Password: <input type="password" name="cpassword"
id="cpassword"></p>
    <input type="submit" value="Save">
</form>
<a href="{{ url_for('profileHome') }}">Go to Profile</a>
</body>
</html>
```

ADD.HTML

```
<!DOCTYPE HTML>
```

```
<html>
<head>
<title>Admin</title>
</head>
<body>
<h2>Add items</h2>
<form action="/addItem" method="POST" enctype="multipart/form-data">
    Name: <input type="text" name="name"><br>
    Price: <input type="text" name="price"><br>
    Description: <textarea name="description" rows=3 cols="40"></textarea><br>
    Image: <input type="file" name="image"><br>
    Stock: <input type="text" name="stock"><br>
    Category: <select name="category">
        {% for row in categories %}
            <option value="{{row[0]}}">{{row[1]}}</option>
        {% endfor %}
    </select><br>
    <input type="submit">
</form>
</body>
</html>
```

STATIC FILES:

CSS FILES:

HOME.CSS

```
#itemImage {
    height: 200px;
    width: 150px;
}

.display {
    margin-top: 20px;
    margin-left: 20px;
    margin-right: 20px;
    margin-bottom: 20px;
}

table {
    margin-left: 200px;
    border-spacing: 20px;
}
```

```css
#productName {
    text-align: center;
    font-weight: bold;
}

#productPrice {
    text-align: center;
}




.displayCategory {
    list-style-type: none;
    margin: 0;
    padding: 0;
    width: 200px;
    background-color: #f1f1f1;
  }


body {
    margin: 0;
}

ul {
    list-style-type: none;
    margin: 0;
    padding: 0;
    width: 15%;
    background-color: #f1f1f1;
    position: fixed;
    height: 100%;
    overflow: auto;
}

li a {
    display: block;
    color: rgb(19, 131, 139);
    padding: 8px 16px;
    text-decoration: none;
}

li a.active {
```

```
    background-color: #09bdb1;
    color: white;
}

li a:hover:not(.active) {
    background-color: #09bdb1;
    color: white;
}
```

PRODUCTDESCRIPTION.CSS

```
#display {
    margin-top: 20px;
    margin-left: 20px;
    margin-right: 20px;
    margin-bottom: 20px;
}

#productImage {
    height: 250px;
    width: 200px;
    margin-left: 20px;
    margin-right: 20px;
    margin-top: 20px;
    margin-bottom: 20px;
    display: inline-block;
    float: left;
}

#productDescription {
    margin-left: 20px;
    margin-right: 20px;
    margin-top: 20px;
    margin-bottom: 20px;
    display: inline-block;
    font-size: 19px;
}

#descriptionTable td {
    width: 150px;
}
```

```
#addToCart {
    font-size: 20px;
}
```

JAVASCRIPT FILES:

FORMVALIDATION.JS

```
function validate() {
    var pass = document.getElementById("password").value;
    var cpass = document.getElementById("cpassword").value;
    if (pass == cpass) {
        return true;
    } else {
        alert("Passwords do not match");
        return false;
    }
}
```

CHANGEPASSWORD.JS

```
function validate() {
    var pass = document.getElementById("newpassword").value;
    var cpass = document.getElementById("cpassword").value;
    if (pass == cpass) {
        return true;
    } else {
        alert("Passwords do not match!");
        return false;
    }
}
```

Describe the platform's layout, features, and technical implementation details.

1. **DB2**
2. **FLASK**
3. **DOCKER**
4. **KUBERNETES**

DB2 :

Designing an artisanal e-commerce platform using IBM Cloud Foundry and DB2 requires careful consideration of layout, features, and technical implementation details. Below is an overview of how the platform's layout, key features, and the technical implementation using DB2 might look:

Platform Layout:

Homepage:

- The homepage should feature a clean, visually appealing design that highlights featured artisans and their products.
- Sections for various product categories or artisan specialties will make it easy for customers to browse.
- A search bar at the top for quick product and artisan searches.

Product Listings:

- Product listings should include high-quality images, product descriptions, prices, and artisan details.
- Customers can filter products by category, price range, and popularity.
- An intuitive and responsive layout ensures a seamless browsing experience on various devices.

Artisan Profiles:

- Each artisan has a dedicated profile page showcasing their work, a brief biography, and contact information.
- Customer reviews and ratings to build trust and credibility.

- Options to follow or favorite artisans for personalized recommendations.

Shopping Cart and Checkout:

- A user-friendly shopping cart where customers can review their selections and make modifications.
- Secure and straightforward checkout process with options for multiple payment gateways.
- Order tracking and confirmation emails for customers.

Key Features:

1. Secure Shopping Cart and Payment Gateway:

- Implement secure payment processing using DB2 to handle financial transactions.
- Ensure encryption and security protocols are in place to protect customer information.

2. User Accounts and Profiles:

- Allow customers to create accounts and save their information for future purchases.
- Artisans can create and manage profiles, including product listings and order management.

3. Product Reviews and Ratings:

- Enable customers to leave reviews and rate products and artisans.
- Implement a system for monitoring and managing reviews to maintain quality and credibility.

4. Multilingual and Multi-Currency Support:

- Offer support for multiple languages and currencies to cater to a global audience.

5. Search and Filter Options:

- Implement an advanced search feature with filters for narrowing down product choices.

6. Notifications and Messaging:

- Set up a notification system for order updates, promotions, and messages between customers and artisans.

7. Responsive Design:

- Ensure the platform is responsive and accessible on various devices, including mobile phones and tablets.

Technical Implementation Details with DB2:

1. Database Schema:

- Design the database schema to store information about products, artisans, customers, orders, and reviews.
- Define tables, relationships, and indexes for efficient data retrieval.

2. Data Security:

- Implement robust data security measures to protect sensitive customer and financial information, including encryption of data in transit and at rest.

3. Scalability:

- Utilize DB2's scalability features to accommodate a growing number of users, products, and transactions.

4. Backup and Recovery:

- Set up regular backups and a disaster recovery plan to ensure data integrity.

5. Query Optimization:

- Optimize database queries to enhance the platform's performance and response times.

6. Compliance and Regulations:

- Ensure compliance with data protection regulations, such as GDPR, by properly handling and securing user data.

7. Integration:

- Integrate the DB2 database with the e-commerce platform's backend and frontend components using appropriate APIs and libraries.

8. Monitoring and Maintenance:

- Implement monitoring tools to track database performance and health, and establish a maintenance schedule for updates and optimizations.

FLASK:

Platform Layout:

1. **Homepage:**
    - Showcase featured artisan products.
    - Highlight artisan profiles and stories.
    - Provide category navigation.
2. **Product Listings:**
    - Categorized product listings for easy browsing.
    - Product images, descriptions, and pricing.
    - Filters and search functionality for finding specific products.
3. **Artisan Profiles:**
    - Individual profiles for artisans with their bio, portfolio, and contact information.
    - Links to products by the artisan.
4. **User Accounts:**
    - User registration and login.
    - User profile management, including order history.
    - Cart management for customers.
5. **Shopping Cart:**
    - Cart page for adding/removing products.
    - Calculate total order cost.
    - Proceed to checkout option.
6. **Checkout:**
    - Multiple payment gateways (e.g., credit card, PayPal).
    - Shipping information collection.

- Confirmation and order summary.

7. **Admin Dashboard:**
    - Artisan and product management.
    - Order management and reporting.
    - User management and support features.

Platform Features:

1. **User Authentication:** Implement user registration and login features using Flask's built-in security tools or popular extensions like Flask-Login.
2. **Database Integration:** Use a database system (e.g., PostgreSQL, MySQL) to store product information, user data, and order history. Flask-SQLAlchemy is a common choice for database integration in Flask applications.
3. **Session Management:** Manage user sessions to keep track of cart items and user authentication status.
4. **Security:** Implement security best practices to protect against common web vulnerabilities (e.g., SQL injection, cross-site scripting). Flask-WTF can be used for form handling and validation.
5. **Payment Integration:** Integrate payment gateways (e.g., Stripe, PayPal) for secure online transactions. Many Flask extensions are available to simplify this process.
6. **Front-End Framework:** Use a front-end framework like Bootstrap to create a responsive and visually appealing user interface.
7. **File Upload:** Allow artisans to upload product images and descriptions. Flask-Uploads can help manage file uploads.
8. **Search and Filtering:** Implement search and filtering functionality for product discovery. Flask-Admin or custom Flask views can help achieve this.

Technical Implementation Details:

1. **Routing:** Define routes for different parts of your application using Flask's routing system.
2. **Templates:** Create HTML templates for various pages, utilizing Flask's Jinja2 template engine for dynamic content rendering.
3. **View Functions:** Write view functions for each page or feature, handling HTTP requests and rendering templates.
4. **Models:** Define data models using Flask-SQLAlchemy to interact with the database.

5. **Forms:** Create forms for user registration, login, product uploads, and more using Flask-WTF or Flask-Form.
6. **User Authentication:** Implement user authentication and authorization using Flask-Login.
7. **Session Management:** Use Flask's session handling to manage user sessions.
8. **Error Handling:** Develop error pages and implement error handling for a smooth user experience.
9. **API Integration:** Integrate third-party APIs for features like payment processing and geolocation services.
10. **Deployment:** Deploy your Flask application using web servers like Gunicorn or uWSGI and reverse proxies like Nginx.
11. **Scalability:** Consider options for scaling the platform as your user base grows, including load balancing and caching strategies.

DOCKER:

Layout: The platform's layout includes various components that interact to provide a seamless shopping experience. These components can be organized as microservices, each running in a Docker container for scalability and maintainability. Here are some of the key components:

1. **Frontend:**
    - The customer and artisan-facing user interface for browsing products, managing accounts, and making purchases.
    - Developed using modern web technologies, such as HTML, CSS, and JavaScript.
    - Running in a Docker container with a web server like Nginx or Apache.
2. **Backend Services:**
    - Handling essential functionalities like product management, user authentication, and order processing.
    - Written in a programming language like Python, Node.js, or Java.
    - Containerized using Docker to ensure portability and scalability.
3. **Database:**

- A relational or NoSQL database to store product information, customer data, order history, and more.
- Common choices include PostgreSQL, MySQL, or MongoDB, which can also be containerized with Docker.

4. **Payment Gateway Integration:**
    - Integration with payment gateways like Stripe, PayPal, or Square to facilitate secure transactions.
    - Utilizing Docker for payment gateway service integration.
5. **Search and Filter Engine:**
    - A component for efficiently searching and filtering products based on various criteria.
    - Containerized to ensure high availability and scalability.
6. **Authentication and Authorization:**
    - Managing user authentication and authorization, ensuring secure access to the platform.
    - Docker containers can house authentication services.
7. **Cache and Load Balancer:**
    - Implementing caching for frequently accessed data and load balancing to distribute traffic evenly.
    - Containerized for improved performance and redundancy.

Features: The e-commerce platform should offer a range of features to meet the needs of artisans and customers:

1. **Product Listings:**
    - Artisans can create and manage product listings with images, descriptions, and pricing.
    - Customers can search and browse products by category, price, or artisan.
2. **Shopping Cart:**
    - Customers can add products to their cart, review items, and proceed to checkout.
    - Cart data should be stored securely and persistently.
3. **User Profiles:**

- Artisans and customers can create and manage their profiles, including contact information and order history.

4. **Secure Checkout:**
    - A secure payment process with integrated payment gateways.
    - Customers can choose their preferred payment method, and the platform ensures data encryption.
5. **Order Management:**
    - Artisans and customers can track orders, view order history, and receive order status updates.
6. **Customer Reviews:**
    - Allow customers to leave reviews and ratings for products and artisans.

Technical Implementation with Docker: Docker containers will be used to encapsulate various components of the platform for deployment and scalability:

1. **Containerization:** Each component (frontend, backend services, databases, payment gateways, etc.) is containerized using Docker.
2. **Orchestration:** Use Docker Compose or Kubernetes for managing the deployment of multiple containers, ensuring they work together seamlessly.
3. **Scalability:** Employ Docker Swarm or Kubernetes to scale containers horizontally based on demand.
4. **Security:** Implement container security best practices, such as image scanning, network isolation, and role-based access control (RBAC).
5. **Continuous Integration/Continuous Deployment (CI/CD):** Set up CI/CD pipelines to automate testing, building, and deploying Docker containers.
6. **Monitoring and Logging:** Use Docker monitoring tools and log aggregation systems to track container performance and troubleshoot issues.
7. **Backups:** Implement regular data backups for the database containers to prevent data loss.

KUBERNETES:

Kubernetes is an open-source container orchestration platform designed to automate the deployment, scaling, and management of containerized applications. Below, I'll describe the platform's layout, key features, and some technical implementation details:

Platform Layout:

Kubernetes follows a master-worker architecture. The primary components are:

1. **Master Node:**
    - **Kube API Server:** Exposes the Kubernetes API and is the entry point for all commands.
    - **etcd:** A distributed key-value store for maintaining cluster state.
    - **Kube Controller Manager:** Ensures the desired state of the system and manages controller processes.
    - **Kube Scheduler:** Assigns work to nodes, based on available resources.
2. **Worker Nodes (Minions):**
    - **Kubelet:** Ensures that containers are running in a Pod (the smallest deployable unit in Kubernetes).
    - **Kube Proxy:** Maintains network rules on the host and enables network communication to Pods.

Key Features:

1. **Container Orchestration:** Kubernetes automates the deployment, scaling, and management of containers (usually Docker containers).
2. **Automated Load Balancing:** Services in Kubernetes are automatically load-balanced, ensuring reliable application accessibility.
3. **Self-Healing:** Kubernetes monitors the health of containers and Pods and can automatically restart or reschedule them if they fail.
4. **Horizontal Scaling:** Easily scale applications up or down based on resource utilization, ensuring optimal performance.

5. **Rolling Updates and Rollbacks:** Kubernetes supports rolling updates to applications, making it easy to change application versions without downtime. If a new version fails, rollbacks can be performed.

6. **Configuration Management:** Kubernetes allows you to define configuration as code and manage it alongside your application.

7. **Storage Orchestration:** Persistent storage can be managed and allocated to containers, ensuring data persistence.

8. **Secrets and Configuration Management:** Kubernetes provides a way to manage sensitive information, like API keys or database passwords, securely.

Technical Implementation Details:

1. **Containers:** Kubernetes primarily works with container runtimes like Docker or containerd. Containers package applications and their dependencies into a single, portable unit.

2. **Pods:** A Pod is the smallest deployable unit in Kubernetes. It can contain one or more containers sharing the same network and storage namespace. Containers within a Pod can communicate with each other using **localhost**.

3. **Deployments:** Deployments are used to define and manage application lifecycles, including scaling and rolling updates.

4. **Services:** Services abstract the network communication between Pods. They ensure that requests to a particular service (e.g., a web application) are correctly routed to the appropriate Pods.

5. **Kubectl:** Kubernetes provides a command-line tool called **kubectl** for interacting with the cluster. You can use **kubectl** to create, inspect, update, and delete resources.

6. **YAML Configuration:** Kubernetes configuration is often defined using YAML files. These files describe the desired state of resources like Pods, Services, Deployments, and more.

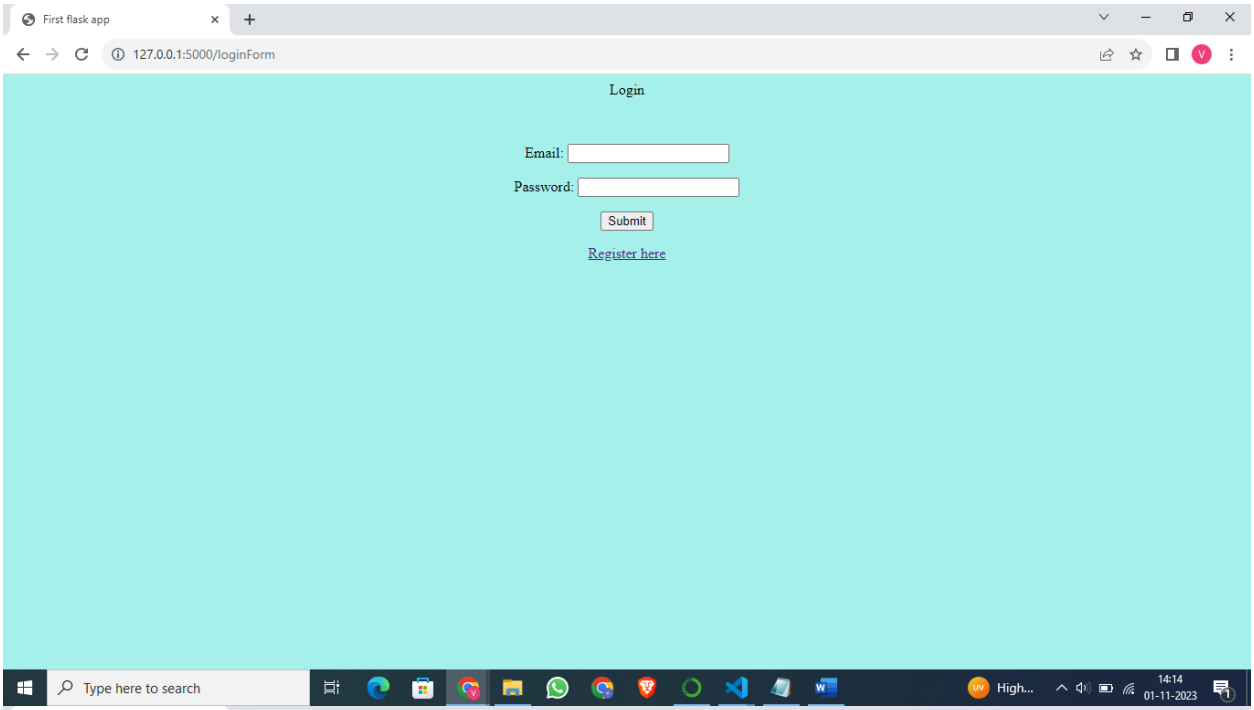
7. **Ingress Controllers:** For managing external access to services, Ingress controllers are used. They can configure rules for routing traffic to different services based on HTTP or HTTPS.

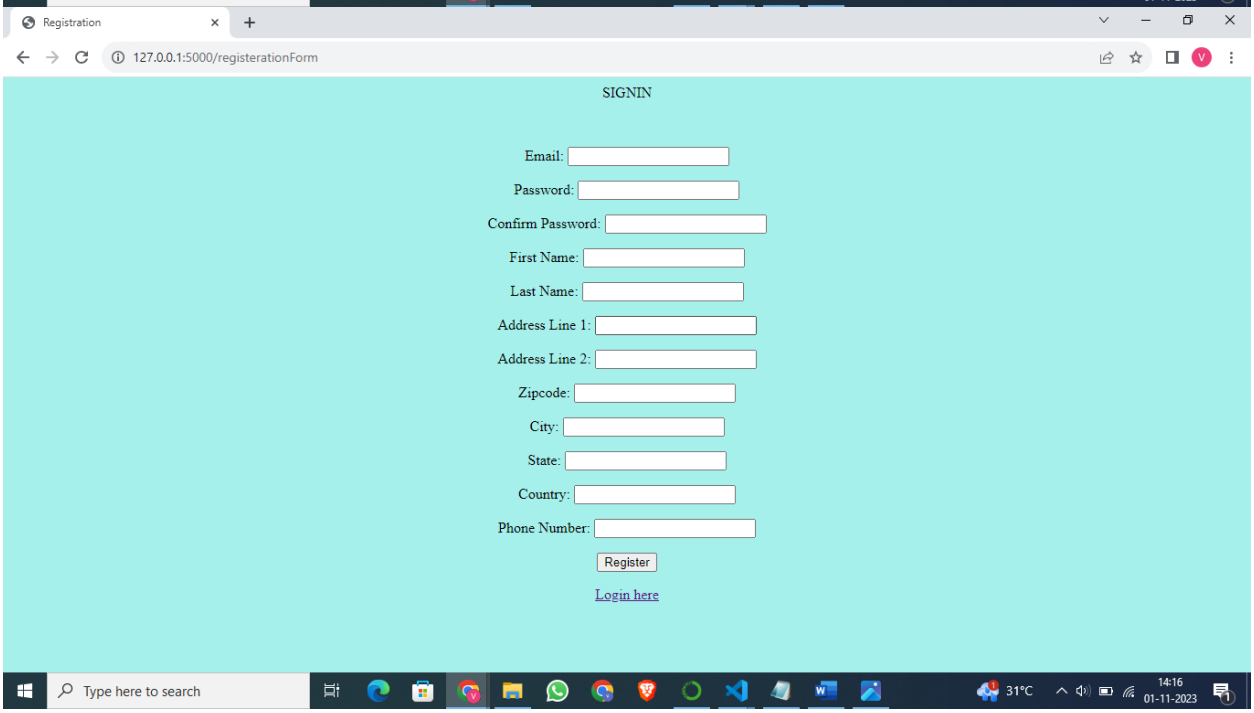
8. **Helm:** Helm is a package manager for Kubernetes that simplifies the installation and management of applications and their dependencies.

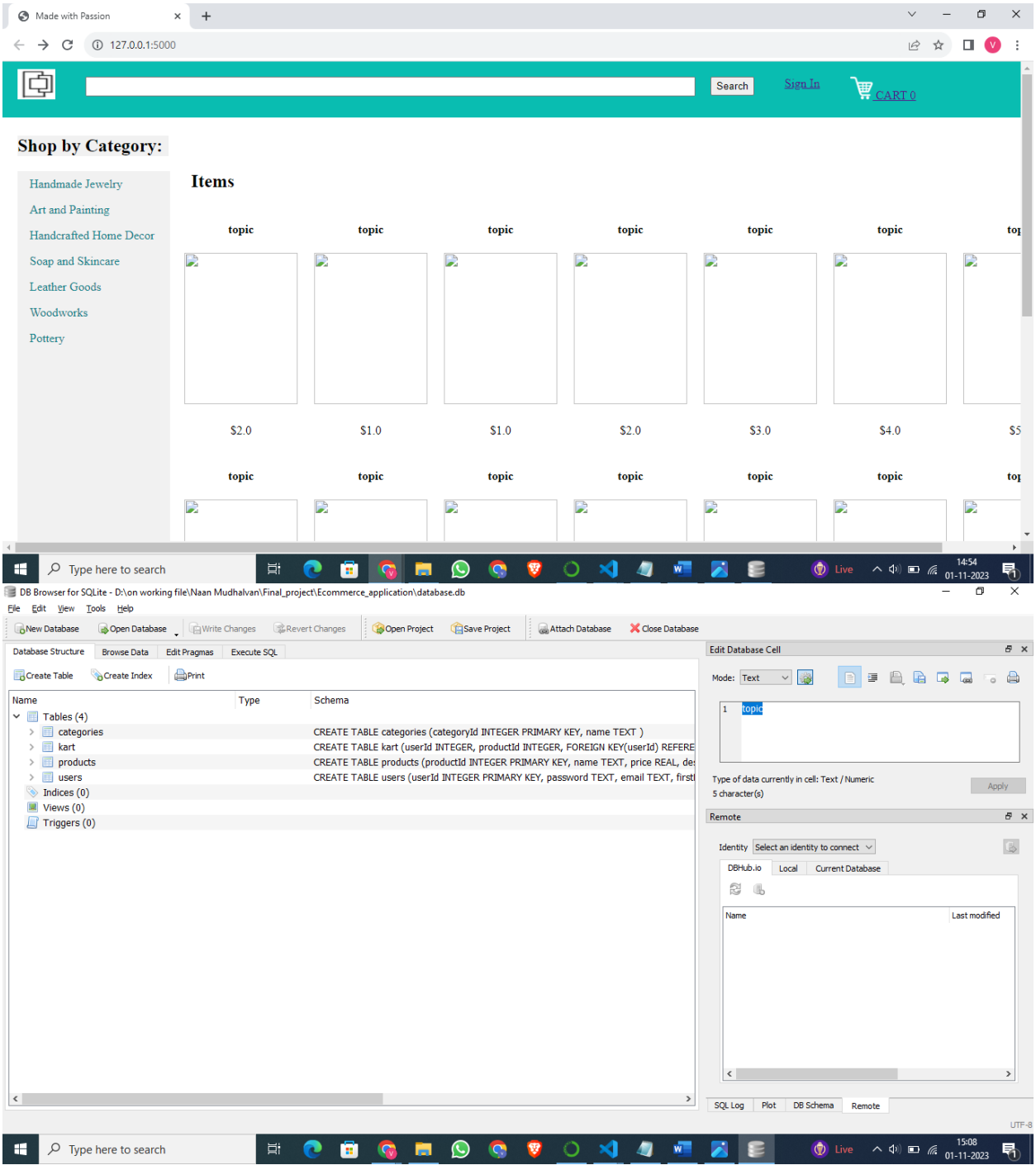
9. **Monitoring and Logging:** Various tools and solutions can be integrated with Kubernetes for monitoring and logging, such as Prometheus and Grafana.

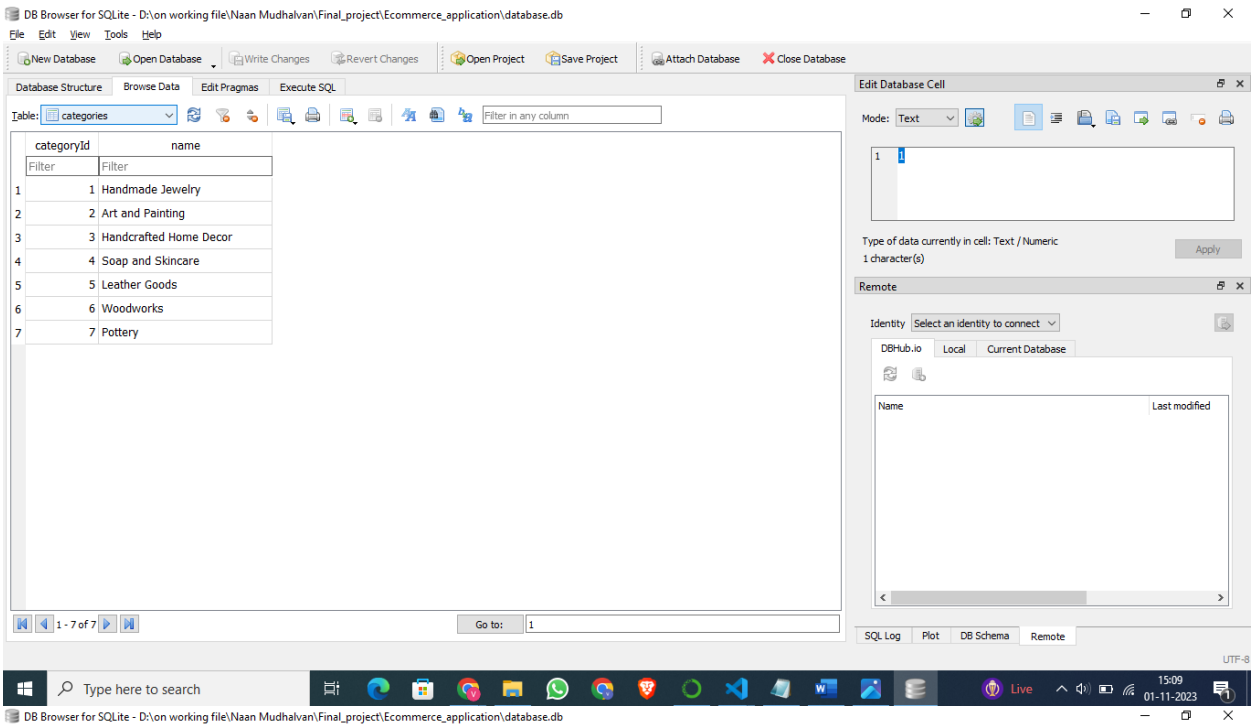
10. **Security:** Kubernetes provides mechanisms for securing the cluster, including RBAC (Role-Based Access Control) and Network Policies.

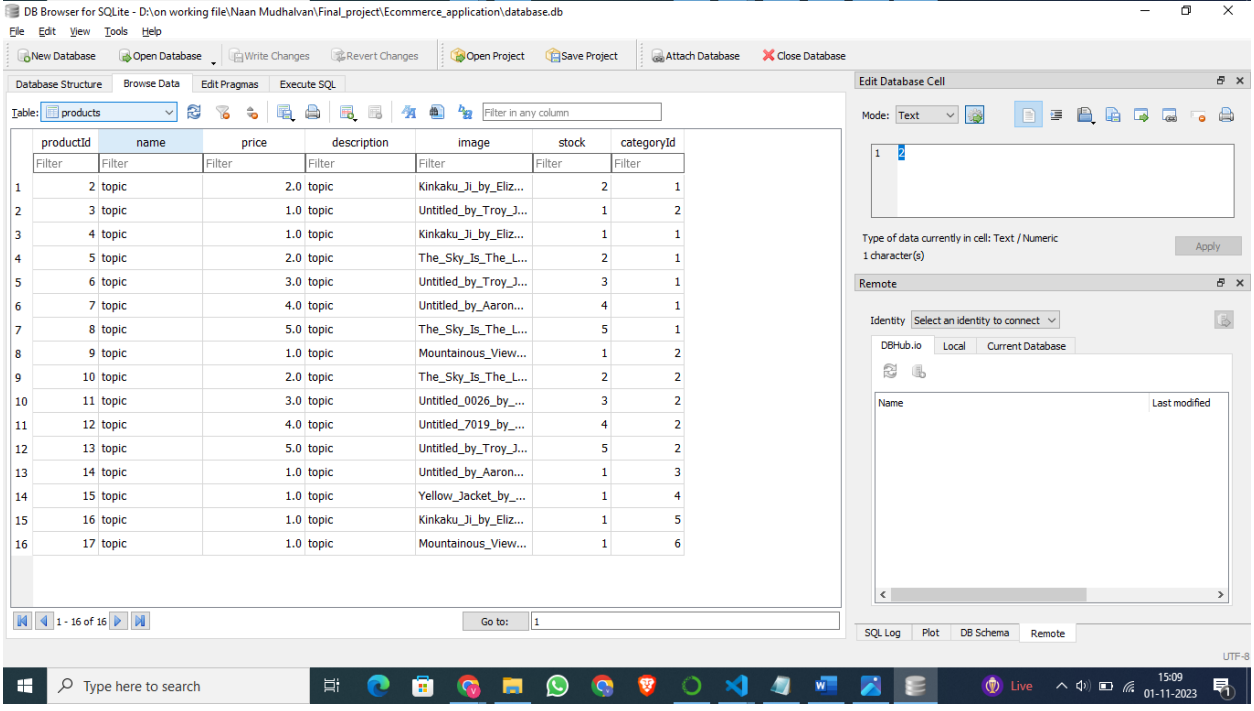
Include screenshots or images of the platform's user interface.

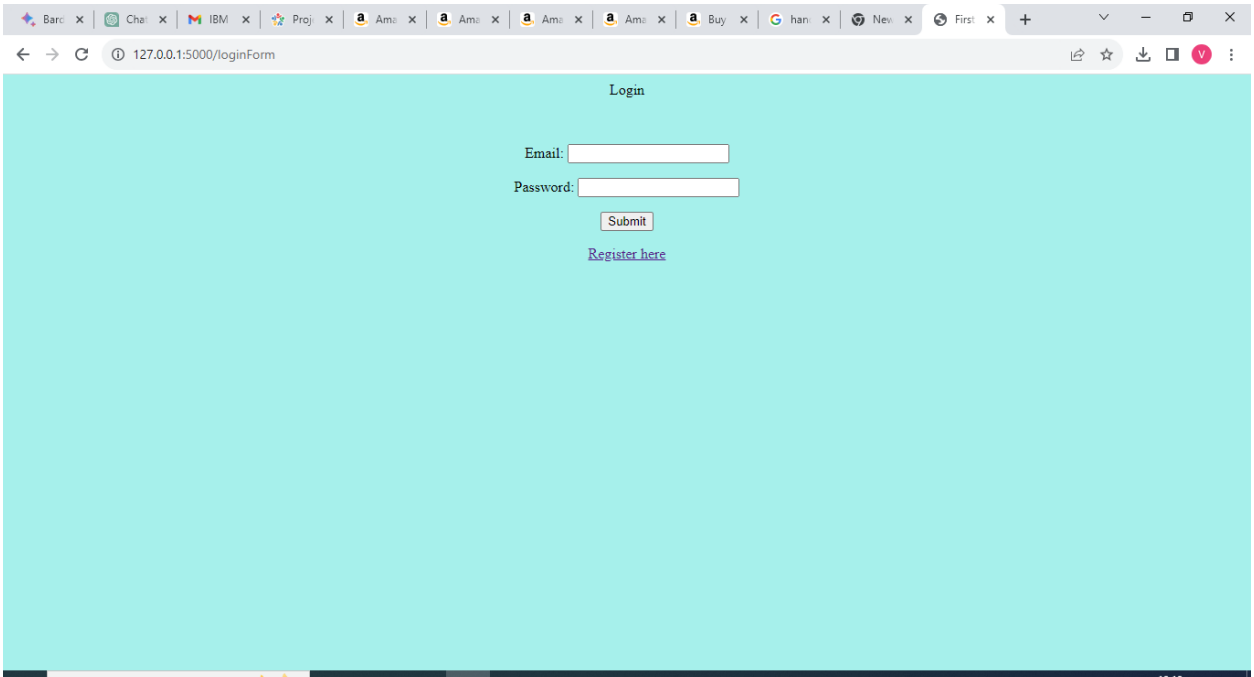


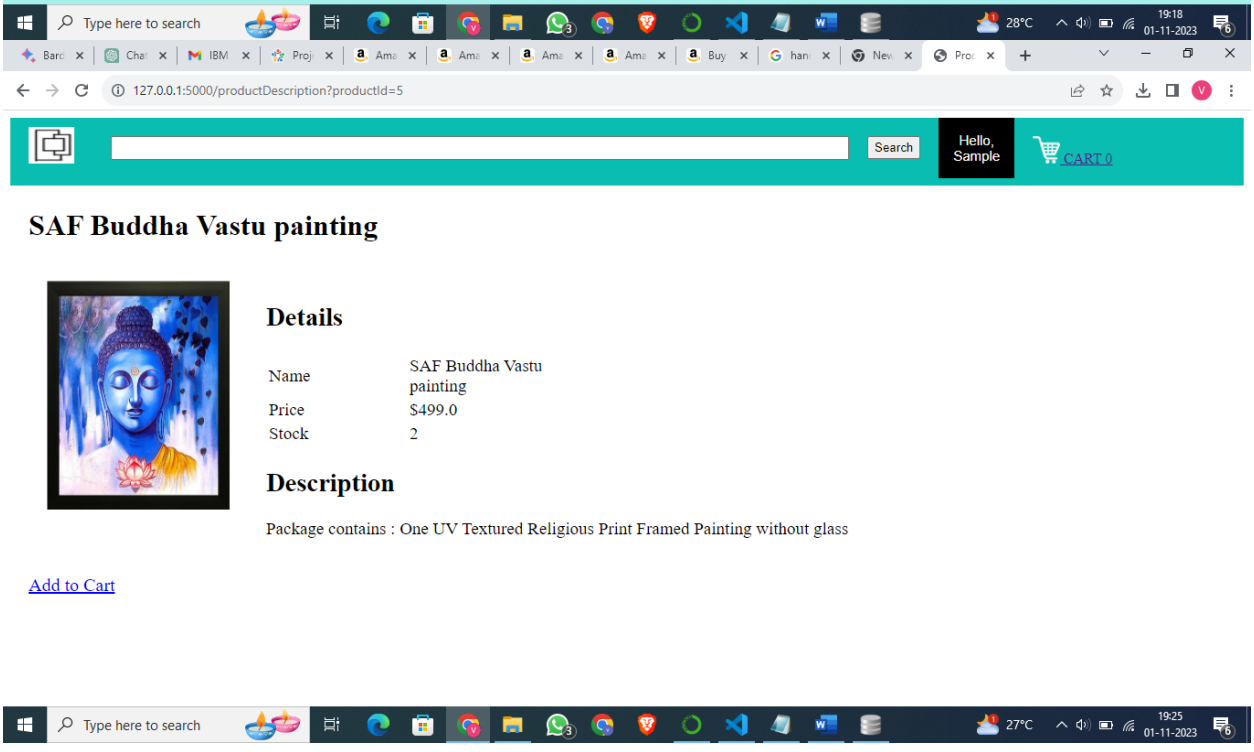


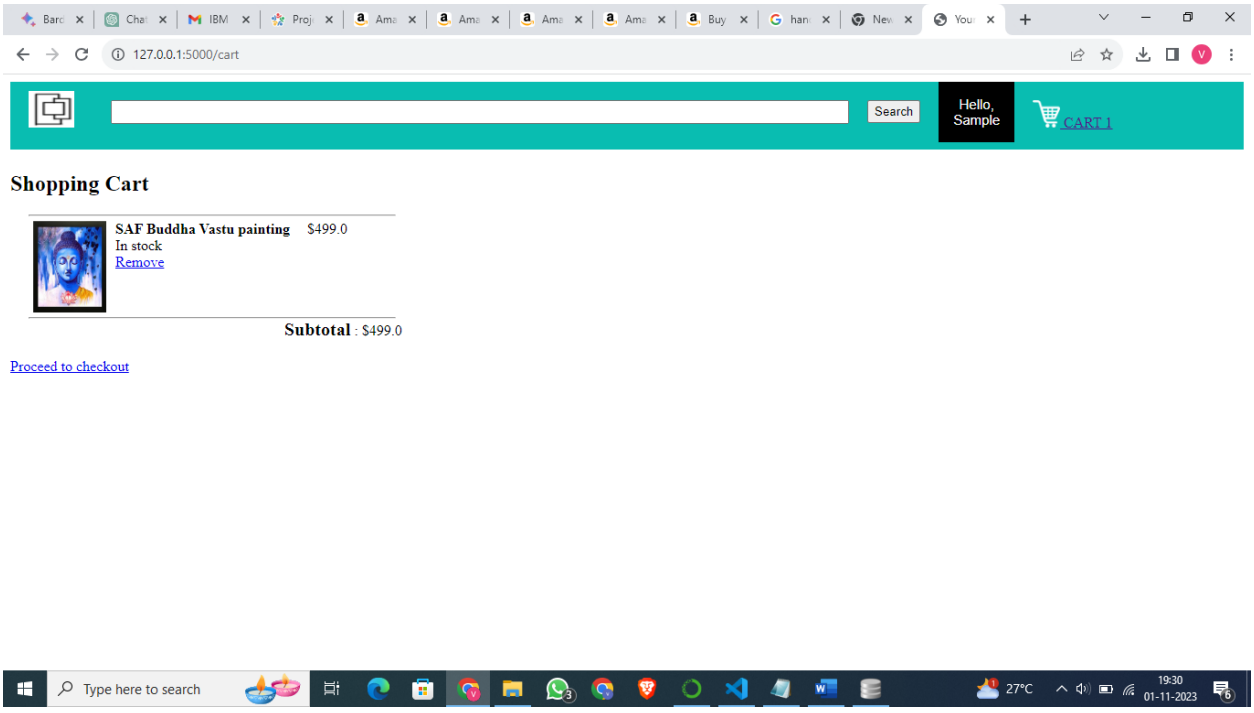

Made with Passion
127.0.0.1:5000
Search
Sign In
CART 0
Shop by Category:
Handmade Jewelry
Art and Painting
Handcrafted Home Decor
Soap and Skincare
Leather Goods
Woodworks
Pottery
Items
topic
$2.0
$1.0
$1.0
$2.0
$3.0
$4.0
Type here to search
DB Browser for SQLite - D:\on working file\Naan Mudhalvan\Final_project\Ecommerce_application\database.db
File Edit View Tools Help
New Database
Open Database
Write Changes
Revert Changes
Open Project
Save Project
Attach Database
Close Database
Database Structure
Browse Data
Edit Pragmas
Execute SQL
Create Table
Create Index
Print
Name
Type
Schema
Tables (4)
categories
CREATE TABLE categories (categoryId INTEGER PRIMARY KEY, name TEXT)
kart
CREATE TABLE kart (userId INTEGER, productId INTEGER, FOREIGN KEY(userId) REFERE
products
CREATE TABLE products (productId INTEGER PRIMARY KEY, name TEXT, price REAL, de
users
CREATE TABLE users (userId INTEGER PRIMARY KEY, password TEXT, email TEXT, first
Indices (0)
Views (0)
Triggers (0)
Edit Database Cell
Mode: Text
Type of data currently in cell: Text / Numeric
5 character(s)
Apply
Remote
Identity
Select an identity to connect
DBHub.io
Local
Current Database
Name
Last modified
SQL Log
Plot
DB Schema
Remote
UTF-8
Type here to search


DB Browser for SQLite - D:\on working file\Naan Mudhalvan\Final_project\Ecommerce_application\database.db
File Edit View Tools Help
New Database Open Database Write Changes Revert Changes Open Project Save Project Attach Database Close Database
Database Structure Browse Data Edit Pragmas Execute SQL
Table: categories
Filter in any column
categoryId name
Filter Filter
1 1 Handmade Jewelry
2 2 Art and Painting
3 3 Handcrafted Home Decor
4 4 Soap and Skincare
5 5 Leather Goods
6 6 Woodworks
7 7 Pottery
1 - 7 of 7
Go to: 1
Edit Database Cell
Mode: Text
1 1
Type of data currently in cell: Text / Numeric
1 character(s)
Apply
Remote
Identity Select an identity to connect
DBHub.io Local Current Database
Name Last modified
SQL Log Plot DB Schema Remote
UTF-8
Type here to search
Live
15:09
01-11-2023

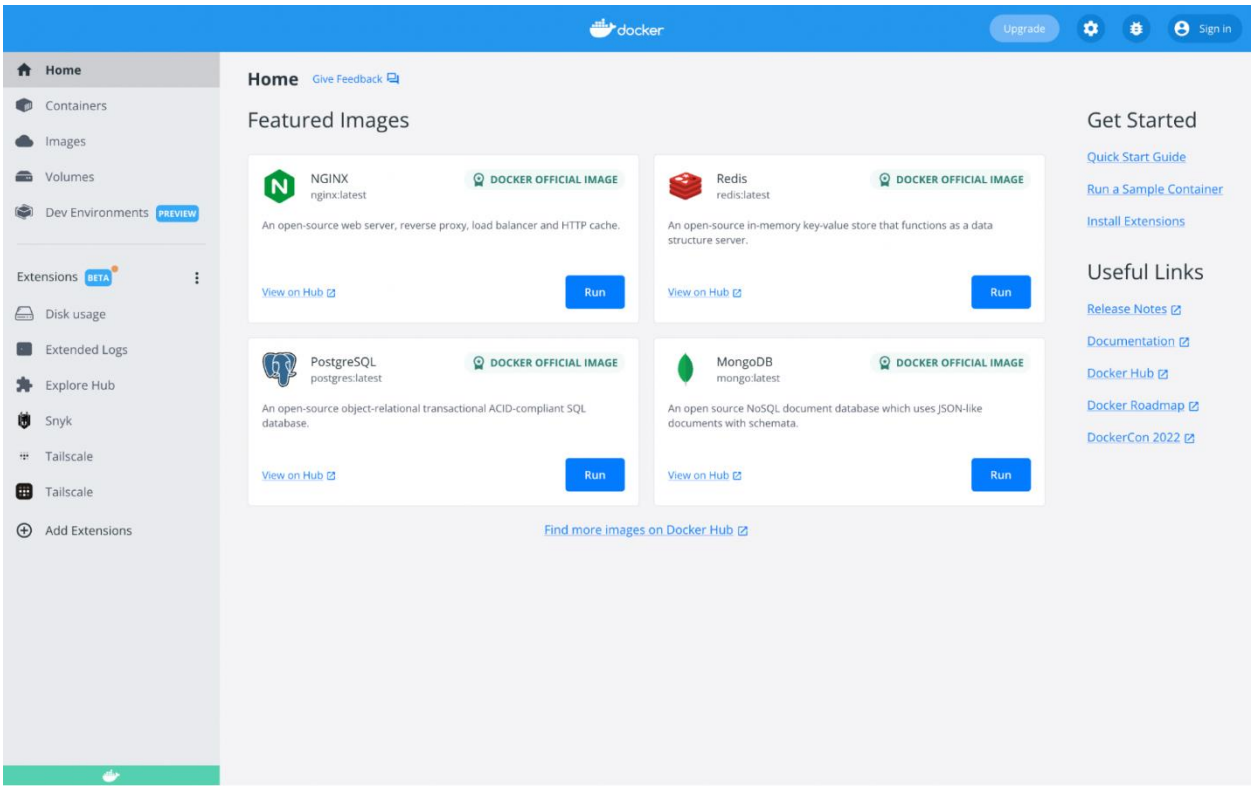

DB Browser for SQLite - D:\on working file\Naan Mudhalvan\Final_project\Ecommerce_application\database.db
File Edit View Tools Help
New Database Open Database Write Changes Revert Changes Open Project Save Project Attach Database Close Database
Database Structure Browse Data Edit Pragmas Execute SQL
Table: products
Filter in any column
productId name price description image stock categoryId
Filter Filter Filter Filter Filter Filter Filter
1 2 topic 2.0 topic Kinkaku_Ji_by_Eliz... 2 1
2 3 topic 1.0 topic Untitled_by_Troy_J... 1 2
3 4 topic 1.0 topic Kinkaku_Ji_by_Eliz... 1 1
4 5 topic 2.0 topic The_Sky_Is_The_L... 2 1
5 6 topic 3.0 topic Untitled_by_Troy_J... 3 1
6 7 topic 4.0 topic Untitled_by_Aaron... 4 1
7 8 topic 5.0 topic The_Sky_Is_The_L... 5 1
8 9 topic 1.0 topic Mountainous_View... 1 2
9 10 topic 2.0 topic The_Sky_Is_The_L... 2 2
10 11 topic 3.0 topic Untitled_0026_by_... 3 2
11 12 topic 4.0 topic Untitled_7019_by_... 4 2
12 13 topic 5.0 topic Untitled_by_Troy_J... 5 2
13 14 topic 1.0 topic Untitled_by_Aaron... 1 3
14 15 topic 1.0 topic Yellow_Jacket_by_... 1 4
15 16 topic 1.0 topic Kinkaku_Ji_by_Eliz... 1 5
16 17 topic 1.0 topic Mountainous_View... 1 6
1 - 16 of 16
Go to: 1
Edit Database Cell
Mode: Text
1 2
Type of data currently in cell: Text / Numeric
1 character(s)
Apply
Remote
Identity Select an identity to connect
DBHub.io Local Current Database
Name Last modified
SQL Log Plot DB Schema Remote
UTF-8
Type here to search
Live
15:09
01-11-2023

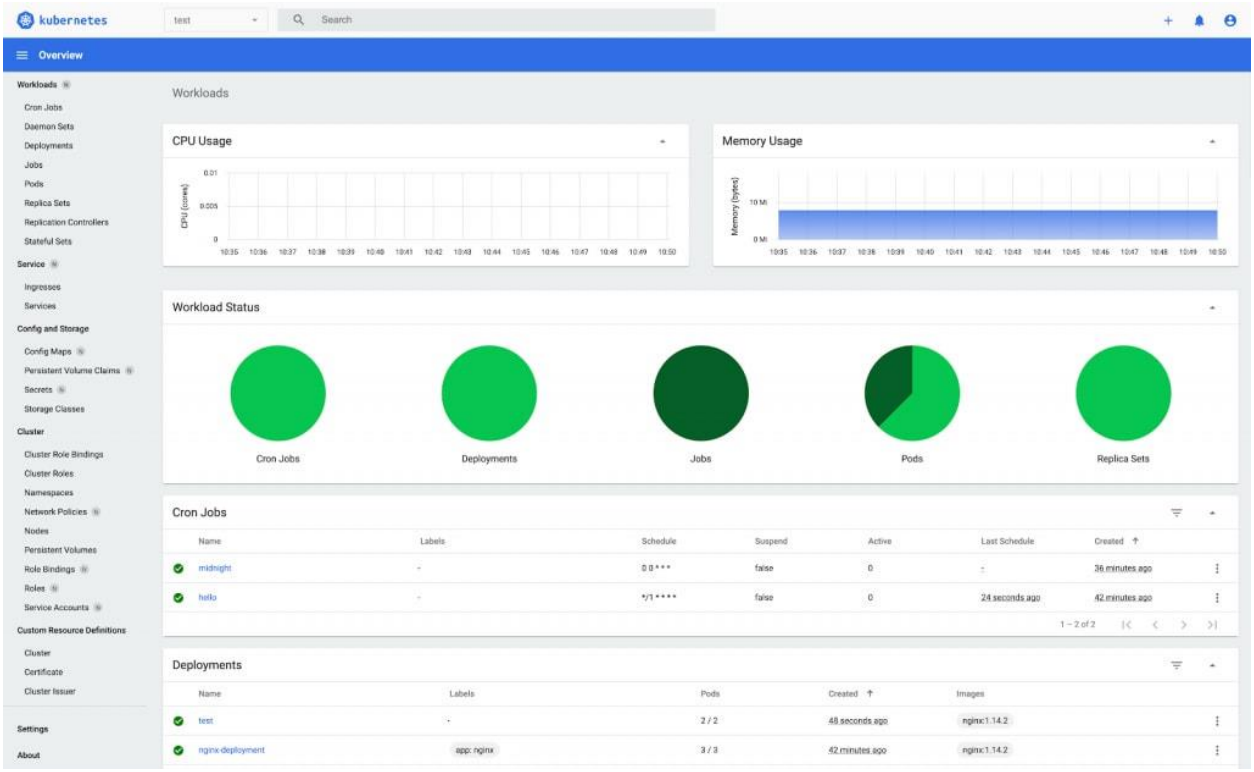

127.0.0.1:5000/loginForm
Login
Email:
Password:
Submit
Register here
Type here to search
28°C
19:18
01-11-2023

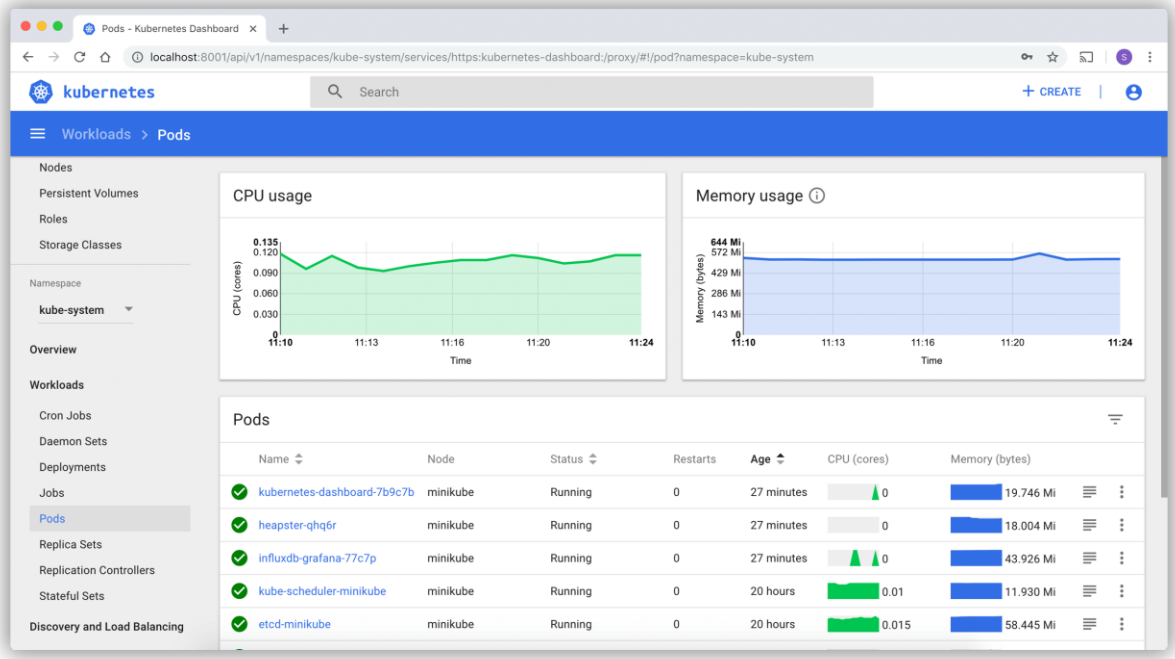

127.0.0.1:5000/productDescription?productId=5
Search
Hello, Sample
CART 0
SAF Buddha Vastu painting
Details
Name
SAF Buddha Vastu painting
Price
$499.0
Stock
2
Description
Package contains : One UV Textured Religious Print Framed Painting without glass
Add to Cart
Type here to search
27°C
19:25
01-11-2023


127.0.0.1:5000/cart
Search
Hello, Sample
CART 1
Shopping Cart
SAF Buddha Vastu painting $499.0
In stock
Remove
Subtotal : $499.0
Proceed to checkout

Type here to search
27°C
19:30
01-11-2023


docker
Upgrade
Sign in
Home
Containers
Images
Volumes
Dev Environments
PREVIEW
Extensions
BETA
Disk usage
Extended Logs
Explore Hub
Snyk
Tailscale
Tailscale
Add Extensions
Home
Give Feedback
Featured Images
NGINX
nginx:latest
DOCKER OFFICIAL IMAGE
An open-source web server, reverse proxy, load balancer and HTTP cache.
View on Hub
Run
Redis
redis:latest
DOCKER OFFICIAL IMAGE
An open-source in-memory key-value store that functions as a data structure server.
View on Hub
Run
PostgreSQL
postgres:latest
DOCKER OFFICIAL IMAGE
An open-source object-relational transactional ACID-compliant SQL database.
View on Hub
Run
MongoDB
mongo:latest
DOCKER OFFICIAL IMAGE
An open source NoSQL document database which uses JSON-like documents with schemata.
View on Hub
Run
Find more images on Docker Hub
Get Started
Quick Start Guide
Run a Sample Container
Install Extensions
Useful Links
Release Notes
Documentation
Docker Hub
Docker Roadmap
DockerCon 2022


kubernetes
test
Search
Overview
Workloads
Cron Jobs
Daemon Sets
Deployments
Jobs
Pods
Replica Sets
Replication Controllers
Stateful Sets
Service
Ingresses
Services
Config and Storage
Config Maps
Persistent Volume Claims
Secrets
Storage Classes
Cluster
Cluster Role Bindings
Cluster Roles
Namespaces
Network Policies
Nodes
Persistent Volumes
Role Bindings
Roles
Service Accounts
Custom Resource Definitions
Cluster
Certificate
Cluster Issuer
Settings
About
Workloads
CPU Usage
CPU (cores)
Memory Usage
Memory (bytes)
Workload Status
Cron Jobs
Deployments
Jobs
Pods
Replica Sets
Cron Jobs
Name
Labels
Schedule
Suspend
Active
Last Schedule
Created
midnight
hello
Deployments
Name
Labels
Pods
Created
Images
test
nginx-deployment


Pods - Kubernetes Dashboard
localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/#!/pod?namespace=kube-system
kubernetes
Search
+ CREATE
Workloads > Pods
Nodes
Persistent Volumes
Roles
Storage Classes
Namespace
kube-system
Overview
Workloads
Cron Jobs
Daemon Sets
Deployments
Jobs
Pods
Replica Sets
Replication Controllers
Stateful Sets
Discovery and Load Balancing
CPU usage
CPU (cores)
Time
Memory usage
Memory (bytes)
Time
Pods
Name
Node
Status
Restarts
Age
CPU (cores)
Memory (bytes)
kubernetes-dashboard-7b9c7b minikube Running 0 27 minutes 0 19.746 Mi
heapster-qhq6r minikube Running 0 27 minutes 0 18.004 Mi
influxdb-grafana-77c7p minikube Running 0 27 minutes 0 43.926 Mi
kube-scheduler-minikube minikube Running 0 20 hours 0.01 11.930 Mi
etcd-minikube minikube Running 0 20 hours 0.015 58.445 Mi