

Analysis Tooltopia

Client: Tooltopia

Background of the problem

Tooltopia is a small business located in the village of Dallington. Founded in 2023, Tooltopia has quickly established itself as a reputable provider of specialized tools and automotive parts. What sets Tooltopia apart is its unique product line offering their own branded tools and parts that are not available through any other retailers. This exclusivity has fostered a strong local customer base, cementing Tooltopia's position as the premier destination for automotive tools in Dallington.

Despite their local success, Tooltopia faces significant growth limitations due to its exclusive reliance on a physical retail model or local customers. Dallington, while supportive, has a finite population, inherently capping the potential customer base. Additionally, without a website, Tooltopia is unable to tap into the vast market opportunities that exist beyond their local area. Recognizing these constraints, the management team at Tooltopia has identified the need for digital transformation to facilitate business expansion

Interview with CEO of Tooltopia

What is your current system?

CEO: We do not have any website and our company is therefore ran manually including managing sales and inventory.

What are the drawbacks of the current system?

CEO: We do not have a website and therefore our company is dependent on local customers, and as our business grows and more demand is required it is becoming increasingly harder to accurately manage our sales and inventory.

What specific challenges do you face with the current manual system?

CEO: The manual system is incredibly time-consuming and prone to errors. For instance, tracking inventory by hand often leads to discrepancies between actual stock and recorded stock levels. This can result in stockouts or overstocking,

both of which are costly for the business. Additionally, during peak times, it becomes difficult to keep up with customer demands and inquiries. We also lack real-time data, which makes it hard to make informed decisions quickly

How does the lack of a website affect your business?

CEO: Without a website, we are limited to serving local customers who visit our physical store in Dallington. This severely restricts our potential customer base. Many potential customers from nearby towns or even further away have no way of knowing about our unique products. Moreover, we miss out on the growing trend of online shopping, where customers expect the convenience of browsing and purchasing products from their homes.

Can you describe the specific features you would like to see in the new website?

CEO: Certainly. First and foremost, we need a comprehensive product catalog where customers can view all available tools and automotive parts, complete with detailed specifications, prices, and high-quality images. It's important that customers can quickly find what they're looking for, so a robust search bar is essential. This would allow them to search for products by entering keywords or product names.

We also need a user-friendly shopping cart. This feature should let customers add items, view their selections, modify quantities, and proceed to checkout with ease. Additionally, a secure user account system is crucial. Customers should be able to register, log in, and save their shopping carts for future visits. Another critical feature is the stock management system. We need an admin-only interface that allows us to update stock levels in real-time easily.

What security measures are essential for the website?

CEO: Security is a top priority. We need SSL encryption to secure data transmissions, especially during payment processes. User passwords should be hashed and stored securely to prevent unauthorized access. Additionally, we should comply with data protection regulations, ensuring customer information is handled responsibly and securely.

Current System and its drawbacks

Tooltopia doesn't have any digital systems for managing sales or inventory. All transactions and inventory tracking are conducted manually which is time consuming and is prone to errors. This limits the effectiveness of operations, and restricts the ability to broaden and scale their business as demand increases. This also makes it difficult to provide immediate response to customer enquiries regarding product availability.

Technical requirements for the proposed system

The website will be developed using Flask, a python-based web framework. There will be all the products price and where users can learn more about the specifications of that specific product. Each product is stored in an SQL lite database, along with all its attributes. Additionally there will be a search bar to help the user their desired product faster in a more convenient way. There will also be a cart system where users can add the products to the cart, and can access their cart where they can modify the contents of the cart or buy their products. There will also be a login system where users register for a new account or if logged in can save what they put in their cart, and buy it at a later date, which is all stored in an SQL LITE database. Additionally for security purposes the passwords of users are passed through a hashing algorithm that makes the website much more secure. It also uses "sessions" where data is temporarily stored while the user is logged in and when they logout it removes all that data, effectively "clearing the session". Finally there will be a stock management system where the admin or only the company itself can access it and update their stock levels accordingly in the products database.

Feasibility Study

Technical Feasibility: Flask and SQLite are suitable for developing a scalable and secure website. They provide the necessary flexibility and performance for a small to medium-sized business.

Economic Feasibility: Initial development costs are justified by the expected increase in sales and operational efficiency gains.

Operational Feasibility: The new system will streamline operations, reduce manual errors, and improve customer satisfaction.

Documented design

Database design.

The Tooltopia website utilises an SQLite database to store and manage data efficiently. here are the primary table and their functions,

Users Table: Stores user information, including usernames, password hashes, and session data. This table supports the login system and it makes it so that users can save items in their carts for future purchases.

Products Table: Contains details of each product such as name, price, description, image URL, and stock levels. It serves as the backbone for the product listing and stock management functionalities.

Cart Items Table: Manages the shopping cart items for each user. It links users to the products they have added to their carts and tracks quantities.

Each table is designed with primary keys for identification and foreign keys for maintaining relationships between tables. For instance, the Cart Items table references both the Users and Products tables to ensure data consistency.

User Interface Design

The user interface is designed to be intuitive and user-friendly, facilitating easy navigation and interaction:

Home Page: This page shows all the products,including their image and prices respectively and an add to cart button under each product where when clicked and logged in allows the user to add that product to the cart.There is also a navigation bar on top where there are 5 buttons first one being the Tooltopia button that returns the user to the home page. the register button. For new users to register with a new account. A login button where existing users can login. Logout button which when clicked allows the user to logout from their account if logged in.Finally a cart button,which redirects the user to the cart page.

Product Detail Page: This is the page that the user sees when clicking on the products in the home page,it shows the product itself the name of the product

the price and additionally a more detailed description of that product and the stock level of the product. There also is an add to cart button that adds that product to the cart.

Cart Page: This page shows all the products that the user added in the cart. This is also where there is a quantity button to update how many of that product they want, and also a remove button to remove that item completely from the cart. There is also the total price of the cart displayed in the bottom right corner and a “Buy” button that allows the user to buy all the products in the cart, and this accordingly updates the stock levels in the database.

Admin Panel: This page is only Accessible to administrators where they need to enter the admin password to access it and a specific url. This page allows for the management of product listings and stock levels.

Functional design

The functional components of the website are implemented using Flask routes and html templates.

Home Route (/): Displays the homepage with all the products. It integrates search functionality to filter products based on user queries.

Product Detail Route (/product/<product_id>): Fetches and displays detailed information for a specific product.

Cart Route (/cart): Manages and displays the shopping cart's contents. Includes functionalities for updating quantities and removing items.

Add to cart (/add_to_cart): manages all the processes of adding products to the cart.

Checkout Route (/checkout): Handles the checkout process, including order confirmation and stock adjustment.

Admin login route (/admin_login): this prompts the user to enter the admin password with which it redirects to the admin page (admin.html).

Admin Route (/admin): Enables product and inventory management through a secure, authenticated interface

Register route (/register): handles making a new account for new users, where users are asked to input username and password and checks if the username is taken or not, and if it is not taken, the new account is stored in the SQLite database, but the password is not stored in its plain form; it is hashed and then stored.

Login route (/login): handles logging in an existing user where if all the details match the one in the database they are logged in and returned to the homepage.

Logout route (/logout): Logs out user from the website.

Search route (/search-products): this route handles actively checking the input on the search bar of the website and applies a linear search algorithm to actively check if the input matches any of the products in the product database, and shows suggestions to the user dynamically, which when clicked redirects the product detail route, with that specific product's id, so that product's detail is displayed accordingly.

Flow of Interaction

Registration and Login: Users can register for a new account or log in to an existing one. The system securely manages user sessions and cart data.

Adding Products to Cart: Users select products on the product detail page and add them to their cart. The user can also update the quantities of the items in the cart. The system then checks stock levels before updating the cart.

Checkout Process: Upon checkout, the system finalizes the order, adjusts inventory, and provides a confirmation to the user along with the total cost of their transaction.

Admin Management: After inputting the admin password, Admins can add new products, update existing listings, and adjust stock levels through the admin panel.

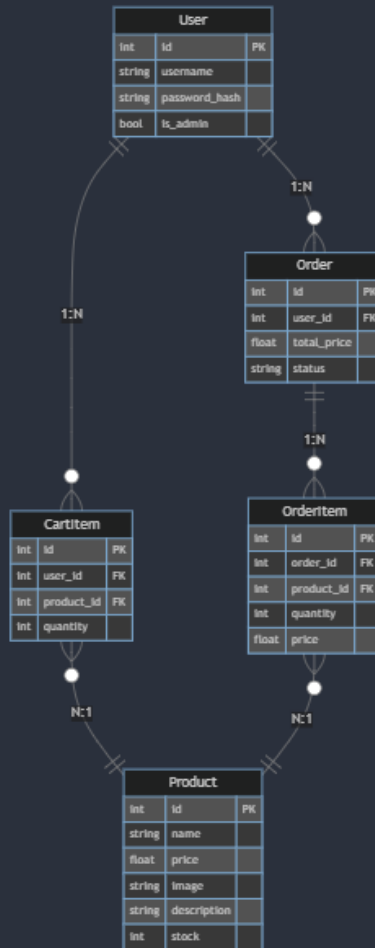
Class diagram

The class diagram will show the relationships between different classes in the system, based on the entities

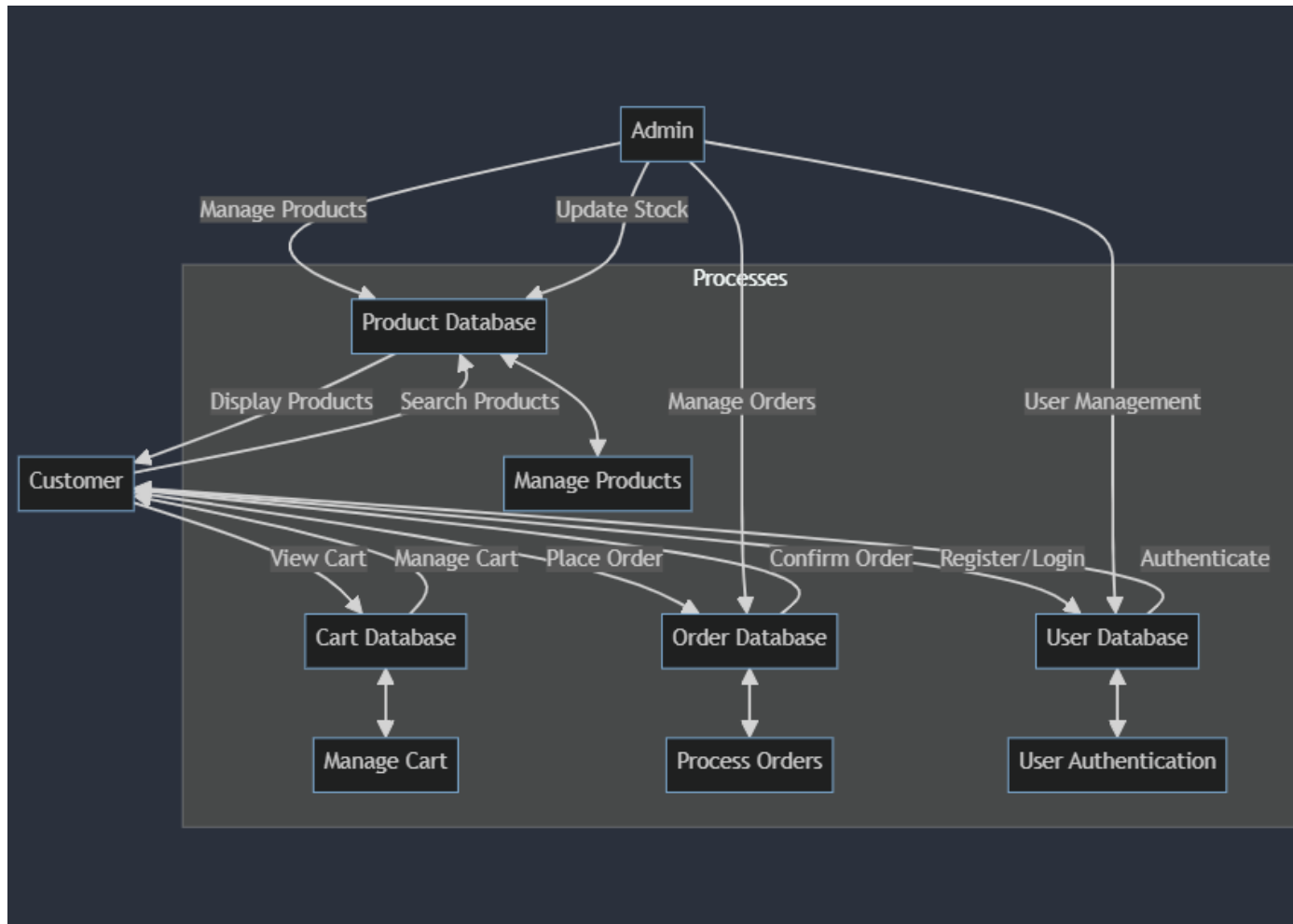


Entity relationship diagram

The ERD will show the relationships between different entities (tables) in the system.

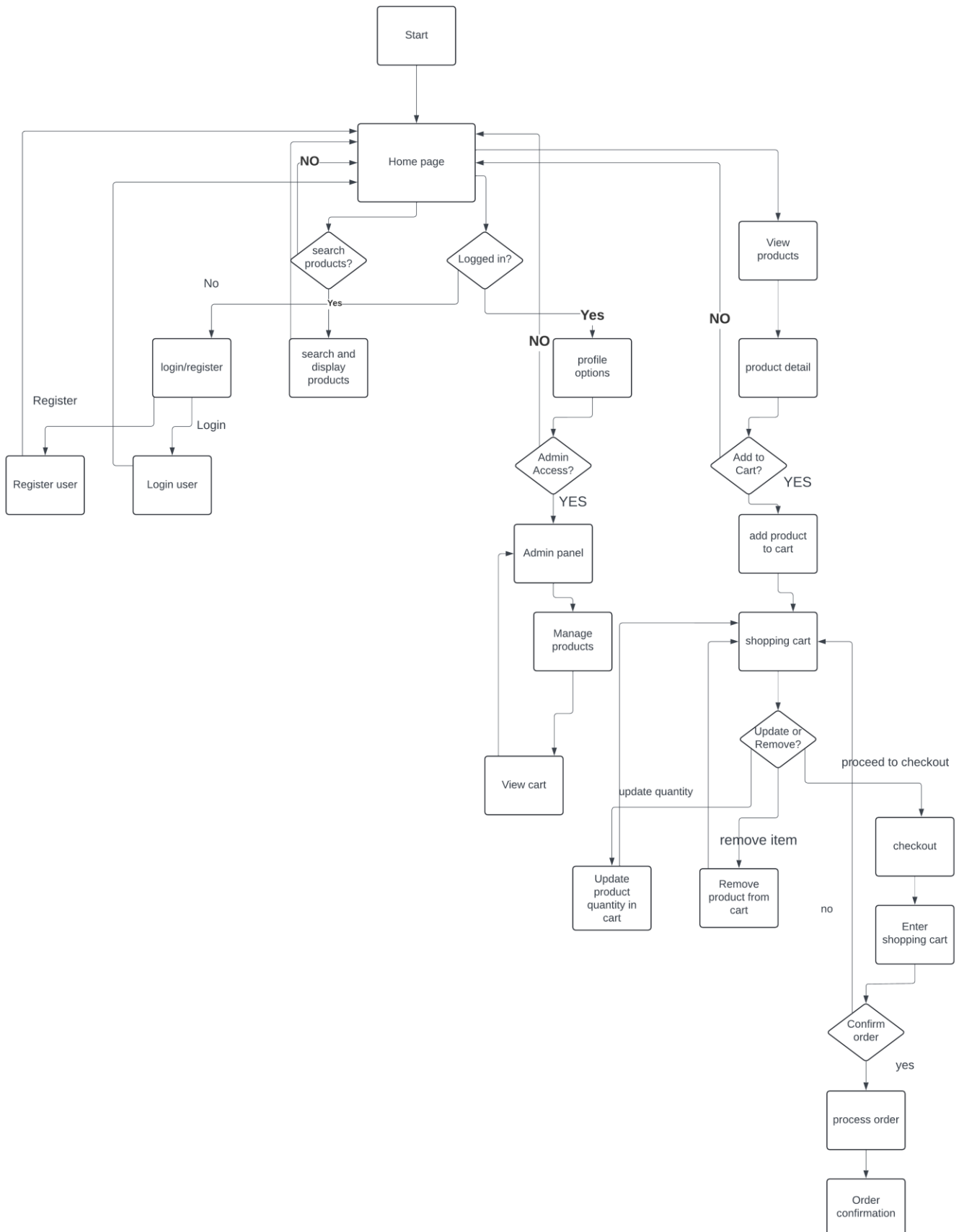


Data flow diagram: The DFD will represent the flow of information within the Tooltopia e-commerce system.



System flowchart

The following chart outlines how the new system operates.



Hashing:

It is also worth noting that passwords are not stored as plain text in the database as this would pose a security risk. It is instead run through a hashing algorithm and a salting algorithm which means they can not be converted to plain text easily.

Technical Solutions

In this section I will explain what each snippet of code does and its role in the website. I will also show how it interacts with other components in the code.

The first part will consist of all the raw code that is used in the website. An explanation will be provided in the second part subtitled “Explanation”.

The app.py file(coded in python,and SQLite and acts as the backend of this website).

```
from flask import Flask, request, render_template, jsonify, flash, redirect, url_for, session
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager, UserMixin, login_user, logout_user, current_user, login_required
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired
from werkzeug.security import generate_password_hash, check_password_hash
import os

class AdminLoginForm(FlaskForm):
    admin_password = PasswordField('Admin Password', validators=[DataRequired()])
    submit = SubmitField('Enter Admin Area')

# Initialize the Flask App
app = Flask(__name__)

# Configure the App
app.config['SECRET_KEY'] = 'your-secret-key-here' # Necessary for session management and form protection
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///products.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# Initialize the Database
db = SQLAlchemy(app)

# Define the User Model
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), unique=True, nullable=False)
    password_hash = db.Column(db.String(128))
    is_admin = db.Column(db.Boolean, default=False)

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)
```

```

def check_password(self, password):
    return check_password_hash(self.password_hash, password)

# Define the Product Model

def __repr__(self):
    return f'<Product {self.name}>'

# Initialize Flask-Login
login_manager = LoginManager(app)
login_manager.login_view = 'login'

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

# Define Forms
class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    remember_me = BooleanField('Remember Me')
    submit = SubmitField('Sign In')

class RegistrationForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
    password = PasswordField('Password', validators=[DataRequired()])
    submit = SubmitField('Register')

class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(150), nullable=False)
    price = db.Column(db.Float, nullable=False)
    image = db.Column(db.String(150), nullable=True)
    description = db.Column(db.Text, nullable=True)
    stock = db.Column(db.Integer, default=0) # Assumes a default stock level of 0 for new
products

class CartItem(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    product_id = db.Column(db.Integer, db.ForeignKey('product.id'), nullable=False)
    quantity = db.Column(db.Integer, default=1, nullable=False)

    user = db.relationship('User', backref=db.backref('cart_items', lazy=True))
    product = db.relationship('Product', backref=db.backref('cart_items', lazy=True))

    def __repr__(self):
        return f'<Product {self.name}>'

cart_items = []

def create_tables():
    db.create_all()

```

```

    if not Product.query.first():
        db.session.add(Product(name="Star screwdriver", price=2.99, image="product1.jpg",
description="Star screwdriver DB,this screwdriver is made from steel and has a magnetic
end to make picking up screwdrivers easy!", stock=100))
        db.session.add(Product(name="3/4 ratchet", price=15.34, image="product3.jpg",
description="Xtool 3/4 ratchet with 120 degrees rotation, anticlockwise rotation, and
clockwise", stock=64))
        db.session.add(Product(name="Flathead screwdriver", price=2.99,
image="product2.jpeg", description="Steudt Flathead screwdriver 100% steel with magnetic
tip", stock=24))
        db.session.add(Product(name="1/2 ratchet", price=20.34, image="product4.jpg",
description="Xtool 1/2 ratchet with 150 degrees rotation, no included sockets bit",
stock=59))
        db.session.add(Product(name="Air compressed spray 500ml", price=15.34,
image="product5.jpg", description="liquimoley compressed air spray 500ml, invertable and
easy to use", stock=23))
        db.session.add(Product(name="Denso k20hr-u11", price=59.99, image="product6.jpeg",
description="Denso spark plug copper tip for great combustion performance", stock=4))
        db.session.add(Product(name="ignition coil denso type2", price=49.99,
image="product7.jpg", description="Denso ignition coil for Toyota Aygo, Citroen C1 type 2
ignition coil", stock=15))
        db.session.add(Product(name="Denso 1KW starter", price=149.99,
image="product8.jpeg",description="starter for smaller car,specifically for toyota aygo
rated power:1KW",stock=3))

    db.session.commit()

with app.app_context():
    create_tables()
# Route Definitions
@app.route('/')
def home_page():
    products = Product.query.all()
    return render_template('main.html', products=products)

@app.route('/add_to_cart', methods=['POST'])
@login_required
def add_to_cart():
    product_id = request.form.get('product_id')
    product = Product.query.get(product_id)
    if product and product.stock > 0: # Check if product is in stock
        existing_item = CartItem.query.filter_by(user_id=current_user.id,
product_id=product_id).first()
        if existing_item:
            if existing_item.quantity < product.stock:
                existing_item.quantity += 1
            else:
                flash('Not enough stock.')
        else:
            db.session.add(CartItem(user_id=current_user.id, product_id=product_id))
            db.session.commit()
            return jsonify({'message': 'Product added to cart'})
    return jsonify({'message': 'Product not found or out of stock'}), 404

```

```

@app.route("/cart")
@login_required
def show_cart():
    cart_items = CartItem.query.filter_by(user_id=current_user.id).all()
    total_price = sum(item.product.price * item.quantity for item in cart_items)
    return render_template('results.html', cart_items=cart_items, total_price=total_price)

@app.route('/remove_from_cart/<int:item_id>', methods=['POST'])
@login_required
def remove_from_cart(item_id):
    item = CartItem.query.get(item_id)
    if item and item.user_id == current_user.id: # Ensures the item exists and belongs to
the current user
        db.session.delete(item)
        db.session.commit()
        flash('Item removed from cart.')
    else:
        flash('Item could not be found.')
    return redirect(url_for('show_cart'))

@app.route('/update_quantity/<int:item_id>', methods=['POST'])
@login_required
def update_quantity(item_id):
    item = CartItem.query.get(item_id)
    if item and item.user_id == current_user.id: # Ensures the item exists and belongs to
the current user
        try:
            new_quantity = int(request.form.get('quantity'))
            if new_quantity > 0:
                item.quantity = new_quantity
                db.session.commit()
                flash('Quantity updated.')
            else:
                flash('Invalid quantity.')
        except ValueError:
            flash('Invalid input for quantity.')
    else:
        flash('Item not found.')
    return redirect(url_for('show_cart'))

@app.route("/result.html")
def nextpage():
    # This creates a dictionary where each key is a product ID
    products = {str(product.id): product for product in Product.query.all()}
    return render_template('results.html', cart_items=cart_items, products=products)

@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        existing_user = User.query.filter_by(username=form.username.data).first()
        if existing_user is not None:
            flash('Username already taken. Please choose a different one.')
            return redirect(url_for('register'))

```

```

        user = User(username=form.username.data)
        user.set_password(form.password.data)
        db.session.add(user)
        db.session.commit()
        flash('Congratulations, you are now a registered user!')
        return redirect(url_for('login'))

    return render_template('register.html', title='Register', form=form)

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()
        if user is None or not user.check_password(form.password.data):
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        # Redirects to the home page
        return redirect(url_for('home_page'))
    return render_template('login.html', title='Sign In', form=form)

@app.route('/logout')
def logout():
    logout_user()
    return redirect(url_for('home_page'))

@app.route('/search')
def search():
    query = request.args.get('q')
    if query:
        products = Product.query.filter(Product.name.ilike(f'%{query}%')).all()
        results = [{'id': p.id, 'name': p.name} for p in products]
        return jsonify(results)
    return jsonify([])

@app.route('/product/<int:product_id>')
def product_detail(product_id):
    product = Product.query.get_or_404(product_id)
    return render_template('product_detail.html', product=product)

@app.route('/checkout', methods=['POST'])
@login_required
def checkout():
    cart_items = CartItem.query.filter_by(user_id=current_user.id).all()
    total_price = 0
    out_of_stock_items = []

    for item in cart_items:
        product = Product.query.get(item.product_id)
        if product and product.stock >= item.quantity:
            total_price += item.product.price * item.quantity
            product.stock -= item.quantity

```

```

        db.session.delete(item)
    else:
        out_of_stock_items.append(product.name)

    if out_of_stock_items:
        flash(f"Not enough stock for {'', '.join(out_of_stock_items)}.", 'error')
        return redirect(url_for('show_cart'))

    db.session.commit()
    flash(f'Thank you for your purchase! Total amount: £{total_price}', 'success')
    return redirect(url_for('home_page'))

@app.route('/admin-login', methods=['GET', 'POST'])
@login_required
def admin_login():
    form = AdminLoginForm()
    if form.validate_on_submit():
        if form.admin_password.data == '112911':
            session['admin_access'] = True
            return redirect(url_for('admin'))
        else:
            flash('Incorrect admin password.')
            session.pop('admin_access', None)
    # If GET request or the password is incorrect, render the login template
    return render_template('admin_login.html', form=form)

@app.route('/admin', methods=['GET', 'POST'])
@login_required
def admin():
    print("Admin access session variable:", session.get('admin_access')) # For debugging
    if 'admin_access' not in session:
        return redirect(url_for('admin_login'))

    products = Product.query.all()
    return render_template('admin.html', products=products)

    products = Product.query.all()
    return render_template('admin.html', products=products)

@app.route('/update_stock/<int:product_id>', methods=['POST'])
@login_required
def update_stock(product_id):
    # Checks if the 'admin_access' session variable is set, implying they've passed the
    admin login.
    if not session.get('admin_access'):
        flash('You do not have access to this page.')
        return redirect(url_for('admin_login'))

    product = Product.query.get_or_404(product_id)
    try:
        new_stock = int(request.form['stock'])
        product.stock = new_stock
        db.session.commit()

```



```

        flash('Stock updated successfully!')
    except ValueError:
        flash('Invalid input for stock.')

    return redirect(url_for('admin'))

@app.route('/admin-logout')
@login_required
def admin_logout():
    session.pop('admin_access', None)
    flash('You have been logged out of the admin area.')
    return redirect(url_for('home_page'))

@app.route('/search-products')
@login_required
def search_products():
    query = request.args.get('q', '')
    if query:
        products = Product.query.filter(Product.name.ilike(f'%{query}%')).all()
        return jsonify([{'id': p.id, 'name': p.name} for p in products])
    return jsonify([])

if __name__ == '__main__':
    with app.app_context():
        db.create_all() # This will create all tables initially if they don't exist.
    app.run(debug=True)

```

Then we have all the html files. Which acts as our frontend. Our first file is the base.html, here's the code for it.

```

<!DOCTYPE html>

<html>
<head>
    <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">

    <title>{%block title%}{%endblock%}</title><!--created a template which another html
page can use and modify it as that page wishes-->
    <link rel="stylesheet" type="text/css" href="{{ url_for('static',
filename='styles.css') }}">
    <script
src="https://code.jquery.com/jquery-3.6.0.min.js"
integrity="sha256-/xUj+30JU5yEx1q6GSYGSHk7tPXikynS7ogEvDej/m4="
crossorigin="anonymous"></script>

</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-light bg-light">
        <a class="navbar-brand" href="/">Tooltopia</a>
        <div class="collapse navbar-collapse" id="navbarNavAltMarkup">

```

```

    <div class="navbar-nav">
      <a class="nav-item nav-link" id="login" href="/login">Login</a>
      <a class="nav-item nav-link" id="register" href="/register">Register</a>
      <a class="nav-item nav-link" id="logout" href="/logout">Log out</a>
    </div>
  </div>
  <form class="form-inline my-2 my-lg-0 ml-auto"> <!-- ml-auto will push the search bar
to the left -->
    <input class="form-control mr-sm-2" type="search" placeholder="Search" aria-
label="Search" id="search-input">
    <button class="btn btn-outline-success my-2 my-sm-0" type="submit">Search</button>
    <div id="search-results" style="position: absolute; background: white; z-index:
1000;"></div>
  </form>
  <ul class="navbar-nav ml-3"> <!-- ml-3 will give some margin to the left of the cart -
->
    <li class="nav-item cart-logo">
      <a class="nav-link" href="/cart">
        
      </a>
    </li>
  </ul>
</nav>

<script>
  document.addEventListener('DOMContentLoaded', function() {
    const searchInput = document.getElementById('search-input');
    const resultsDiv = document.getElementById('search-results');

    searchInput.addEventListener('input', function() {
      const query = searchInput.value;
      if (query.length > 2) { // Only searches if there are more than 2 characters
        fetch(`/search?q=${encodeURIComponent(query)}`)
          .then(response => response.json())
          .then(data => {
            resultsDiv.innerHTML = ''; // Clears previous results
            data.forEach(item => {
              const div = document.createElement('div');
              div.innerHTML = `<a
href="/product/${item.id}">${item.name}</a>`;
              resultsDiv.appendChild(div);
            });
          });
      } else {
        resultsDiv.innerHTML = '';
      }
    });
  });
</script>

```

```

{%endblock%}

{% for message in get_flashed_messages() %}
    <div>{{ message }}</div>
{% endfor %}
<script>
    window.setTimeout(function() {
        $(".flash").hide();
    }, 4000);

    // Adjusts the timing of the messages being flashed
</script>

    <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-
q8i/X+965Dz00rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo"
crossorigin="anonymous"></script>
    <script src="https://cdn.jsdelivr.net/npm/popper.js@1.14.7/dist/umd/popper.min.js"
integrity="sha384-U02eT0CpHqdsSJQ6hJty5KVphtPhzWj9W01clHTMGa3JDZwrnQq4sF86dIHNDz0W1"
crossorigin="anonymous"></script>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/js/bootstrap.min.js"
integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy60rQ6VrjIEaFf/njGzIxFDsf4x0xIM+B07jRM"
crossorigin="anonymous"></script>
</body>
</html>

```

Main.html:

```

{% extends "base.html" %}
{% block title %}Homepage{% endblock %}
{% block content %}
<h1>Products</h1>
{% with messages = get_flashed_messages() %}
    {% if messages %}
        <ul>
            {% for message in messages %}
                <li>{{ message }}</li>
            {% endfor %}
        </ul>
    {% endif %}
{% endwith %}
<div class="product-grid">
    {% for product in products %}
        <div class="product">
            <a href="{{ url_for('product_detail', product_id=product.id) }}">
                
            </a>
            <h3>{{ product.name }}</h3>
            <p>£{{ product.price }}</p>
            <button class="add-to-cart" data-product-id="{{ product.id }}">Add to
Cart</button>

```

```

    </div>
    {% endfor %}
</div>
<div id="cart-notification" style="display: none;">Product added to cart!</div>
<script>
    document.addEventListener('DOMContentLoaded', function() {
        var addToCartButtons = document.querySelectorAll('.add-to-cart');
        addToCartButtons.forEach(function(button) {
            button.addEventListener('click', function() {
                var productId = button.dataset.productId;
                var xhr = new XMLHttpRequest();
                xhr.open('POST', '/add_to_cart');
                xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
                xhr.onload = function() {
                    if (xhr.status === 200) {
                        var notification = document.getElementById('cart-notification');
                        notification.style.display = 'block';
                        setTimeout(function() {
                            notification.style.display = 'none';
                        }, 3000);
                    }
                };
                xhr.send('product_id=' + productId);
            });
        });
    });
</script>

{% endblock %}

```

Results.html:

```

{% extends "base.html" %}
{% block title %}Cart{% endblock %}

{% block content %}
<h1>Your Cart</h1>
{% for item in cart_items %}
<div class="cart-item">
    
    <div class="cart-item-details">
        <h3>{{ item.product.name }}</h3>
        <p>£{{ item.product.price }}</p>
        <form action="{{ url_for('update_quantity', item_id=item.id) }}" method="post">
            <input type="number" name="quantity" value="{{ item.quantity }}" min="1">
            <input type="submit" value="Update" class="btn btn-info">
        </form>
        <form action="{{ url_for('remove_from_cart', item_id=item.id) }}" method="post">
            <input type="submit" value="Remove" class="btn btn-danger">
        </form>
    </div>
</div>

```

```

</div>
{% endfor %}
<div class="cart-summary">
  <div class="total-price">
    <h3>Total Price: £{{ total_price }}</h3>
  </div>
  <form action="{{ url_for('checkout') }}" method="post">
    <button type="submit" class="buy-btn">Buy</button>
  </form>

{% endblock %}

```

Login.html:

```

<!-- login.html -->
{% extends "base.html" %}

{% block title %}
Login
{% endblock %}

{% block content %}
<h1>Login</h1>
<form action="{{ url_for('login') }}" method="post">
  {{ form.hidden_tag() }}
  <div>
    {{ form.username.label }}
    {{ form.username(size=32) }}
    {% if form.username.errors %}
      {% for error in form.username.errors %}
        <div>{{ error }}</div>
      {% endfor %}
    {% endif %}
  </div>
  <div>
    {{ form.password.label }}
    {{ form.password(size=32) }}
    {% if form.password.errors %}
      {% for error in form.password.errors %}
        <div>{{ error }}</div>
      {% endfor %}
    {% endif %}
  </div>
  <div>
    {{ form.remember_me() }} Remember Me
  </div>
  <div>
    {{ form.submit() }}
  </div>
</form>
{% endblock %}

```

Register.html:

```
<!-- register.html -->
{% extends "base.html" %}

{% block title %}
Register
{% endblock %}

{% block content %}
<h1>Register</h1>
<form action="{{ url_for('register') }}" method="post">
    {{ form.hidden_tag() }}
    <div>
        {{ form.username.label }}
        {{ form.username(size=32) }}
        {% if form.username.errors %}
            {% for error in form.username.errors %}
                <div>{{ error }}</div>
            {% endfor %}
        {% endif %}
    </div>
    <div>
        {{ form.password.label }}
        {{ form.password(size=32) }}
        {% if form.password.errors %}
            {% for error in form.password.errors %}
                <div>{{ error }}</div>
            {% endfor %}
        {% endif %}
    </div>
    <div>
        {{ form.submit() }}
    </div>
</form>
{% endblock %}
```

Product_detail.html: {% extends "base.html" %}

```
{% block content %}
<div class="product-detail-container">
    <div class="product-image">
        
    </div>
    <div class="product-info">
        <h1>{{ product.name }}</h1>
        <p class="product-price">£{{ product.price }}</p>
        <p class="product-description">{{ product.description }}</p>
        <p class="in-stock">In stock: {{ product.stock }}</p>
    </div>
</div>
{% endblock %}
```

```

        <button class="add-to-cart" data-product-id="{{ product.id }}">Add to
Cart</button>
    </div>

</div>
<div id="cart-notification" style="display: none;">Product added to cart!</div>
<script>
    document.addEventListener('DOMContentLoaded', function() {
        var addToCartButtons = document.querySelectorAll('.add-to-cart');
        addToCartButtons.forEach(function(button) {
            button.addEventListener('click', function() {
                var productId = button.dataset.productId;
                var xhr = new XMLHttpRequest();
                xhr.open('POST', '/add_to_cart');
                xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
                xhr.onload = function() {
                    if (xhr.status === 200) {
                        var notification = document.getElementById('cart-notification');
                        notification.style.display = 'block';
                        setTimeout(function() {
                            notification.style.display = 'none';
                        }, 3000);
                    }
                };
                xhr.send('product_id=' + productId);
            });
        });
    });
</script>
</div>
{% endblock %}

```

Admin_login.html:

```

{% extends "base.html" %}

{% block content %}
<h1>Admin Login</h1>
<form action="{{ url_for('admin_login') }}" method="post">
    {{ form.hidden_tag() }}
    <div>
        {{ form.admin_password.label }}
        {{ form.admin_password(size=32) }}
        {% if form.admin_password.errors %}
            {% for error in form.admin_password.errors %}
                <div>{{ error }}</div>
            {% endfor %}
        {% endif %}
    </div>
    <div>
        {{ form.submit() }}
    </div>
</form>

```

```
</form>
{% endblock %}
```

admin.html:

```
{% extends "base.html" %}

{% block content %}
<h1>Admin Page</h1>
<div class="product-list">
    {% for product in products %}
        <div class="product">
            
            <h3>{{ product.name }}</h3>
            <p>£{{ product.price }}</p>
            <form action="{{ url_for('update_stock', product_id=product.id) }}"
method="post">
                <input type="number" name="stock" value="{{ product.stock }}">
                <button type="submit">Update</button>
            </form>
        </div>
    {% endfor %}
</div>
{% endblock %}
```

Explanation:

1.The first file is app.py.This is the Backend of the website or in other words where every complex algorithms, processes and where it its managed how the data is moving in different part of the code. There is also “Front end”this is regarded as being the actual part of the code that the user sees such as the actual website, these are often coded in .html files which I will discuss later.

```
from flask import Flask, request, render_template, jsonify, flash, redirect,
url_for,session
from flask_sqlalchemy import SQLAlchemy
from flask_login import LoginManager, UserMixin, login_user, logout_user,
current_user,login_required
from flask_wtf import FlaskForm
from wtforms import StringField, PasswordField, BooleanField, SubmitField
from wtforms.validators import DataRequired
from werkzeug.security import generate_password_hash, check_password_hash
import os
```


this is written in the top of the app.py file and these codes import all the necessary libraries in python, specifically for a web application built with flask.

Flask: The core framework used to create instances of the web application.

request: Used to handle data sent from a client (such as web browsers) to the server.

render_template: This allows to render HTML templates

jsonify: Converts data such as dictionaries in Python into JSON format to send responses back to the client.(front end)

flash: Used to send one-time messages to templates(front end), typically for alerts like errors or notifications.

redirect: Redirects the user to another endpoint within the application.

url_for: Generates URLs to specific functions or endpoints in the app.

session: Manages sessions allows to store information specific to a user from one request to the next.

Flask-SQLAlchemy:

```
from flask_sqlalchemy import SQLAlchemy:
```

SQLAlchemy: This is an ORM (Object Relational Mapper) that allows to work with databases using Python classes and objects instead of writing SQL queries directly.

Flask-Login:

```
from flask_login import LoginManager, UserMixin, login_user, logout_user, current_user, login_required:
```

LoginManager: Manages the user login process.

UserMixin: Provides default implementations for methods that Flask-Login expects user objects to have.

login_user: Function that logs a user in.

logout_user: Function that logs a user out.

`current_user`: Represents the logged-in user and can be used in the application to access user data.

`login_required`: A decorator to protect certain routes, making them accessible only to authenticated users.

Flask-WTF and WTForms:

```
from flask_wtf import FlaskForm:
```

`FlaskForm`: A class that all your forms inherit from to get CSRF protection and other form handling functionalities.

```
from wtforms import StringField, PasswordField, BooleanField, SubmitField:
```

`StringField`, `PasswordField`, `BooleanField`, `SubmitField`: Various form field types used to capture data in forms, e.g., text, passwords, checkboxes, and buttons.

```
from wtforms.validators import DataRequired:
```

`DataRequired`: A validator to ensure that a field is not submitted empty.

Werkzeug Security:

```
from werkzeug.security import generate_password_hash, check_password_hash:
```

`generate_password_hash`: Creates a hashed (encrypted) version of a password.

`check_password_hash`: Validates a password against the hashed version to see if they match, ensuring security during authentication.

OS Module:

```
import os:
```

`os`: Provides a way of using operating system dependent functionality like reading or writing to the filesystem, manipulating paths.

The website is initiated with these codes

```
app = Flask(__name__)  
if __name__ == '__main__':
```

these codes gets the website up and running. Nothing can be seen yet by the user at this stage as this only allowed for the server or backend to be activated.

The next part is securing the client-side sessions secure, for this, the following code has been used

```
app.config['SECRET_KEY'] = 'your-secret-key-here' # Necessary for session management and form protection
```

We then have to set up a database as previously mentioned to store product info user data price stock levels etc.

```
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///products.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

# Initialize the Database
db = SQLAlchemy(app)
```

the following code is used to configure and set up the database.

The first line, `app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///products.db'`

Is used to configures the URI (Uniform Resource Identifier) for the database. It tells SQLAlchemy which database engine to use and how to connect to it. Here, it's set to use SQLite and to store the database in a file named `products.db`.

The second line `app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False`:

Is set to avoid signaling the application every time a change is about to be made in the database. It's typically set to False to disable this feature and improve performance.

feature and improve performance, as it consumes extra memory and should be turned off unless specifically needed

`db=SQLAlchemy(app)` is used to initialise and set the database to be running.

```
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(64), unique=True, nullable=False)
    password_hash = db.Column(db.String(128))
    is_admin = db.Column(db.Boolean, default=False)
```

`class User(db.Model, UserMixin)::`

This defines a User model as a Python class that inherits from `db.Model` (a base class for all models from SQLAlchemy) and `UserMixin` (provides default

implementations for methods that Flask-Login expects the user objects to have).

id: A column in the database table that represents the unique identifier for each user. Which is our primary key in this case.

username: A column defined to store the username. It is unique (no two users can have the same username), and it cannot be null (every user must have a username).

password_hash: This column stores the hashed password of the user, enhancing security by not storing plain passwords.

is_admin: A boolean column that indicates whether the user has admin privileges. It defaults to False, meaning users are not admins unless explicitly specified.

```
class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(150), nullable=False)
    price = db.Column(db.Float, nullable=False)
    image = db.Column(db.String(150), nullable=True)
    description = db.Column(db.Text, nullable=True)
    stock = db.Column(db.Integer, default=0)
```

this creates columns within the “product” database, which has a primary key the “id” number along with the other attributes.

```
@app.route('/')
def home_page():
    products = Product.query.all()
    return render_template('main.html', products=products)
```

this is the function that is run when the website is opened by default the website will have a / at the end so this code says that if there is / at the end run the “home page” function. then there is a variable products that stores everything from the database with class “product”. then the “return render template” function on the 4th line is saying to open or render the main.html page which is what web browsers can understand. and it also passes the variable to the front end. Simply this is the function that allows for the home page to be rendered.

Introduction of html files

Html files are the files which can be read by the web browser itself this is often referred to as the front end. In this website we have a base html file. This file is the file which all the other html file will copy from.

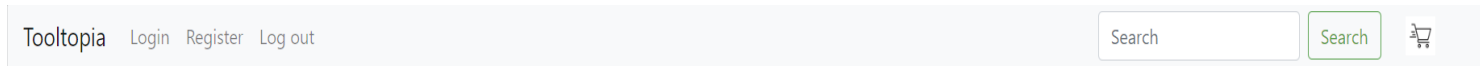
```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bootstrap@4.3.1/dist/css/bootstrap.min.css"
integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">

  <title>{%block title%}{%endblock%}</title><!--created a template which another html
page can use and modify it as that page wishes-->
  <link rel="stylesheet" type="text/css" href="{% url_for('static',
filename='styles.css') %}">
  <script
src="https://code.jquery.com/jquery-3.6.0.min.js"
integrity="sha256-/xUj+30JU5yEx1q6GSYGS7tPXikynS7ogEvDej/m4="
crossorigin="anonymous"></script>

</head>
<body>
  <nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand" href="/">Tooltopia</a>
    <div class="collapse navbar-collapse" id="navbarNavAltMarkup">
      <div class="navbar-nav">
        <a class="nav-item nav-link" id="login" href="/login">Login</a>
        <a class="nav-item nav-link" id="register" href="/register">Register</a>
        <a class="nav-item nav-link" id="logout" href="/logout">Log out</a>
      </div>
    </div>
    <form class="form-inline my-2 my-lg-0 ml-auto"> <!-- ml-auto will push the search bar
to the left -->
      <input class="form-control mr-sm-2" type="search" placeholder="Search" aria-
Label="Search" id="search-input">
      <button class="btn btn-outline-success my-2 my-sm-0" type="submit">Search</button>
      <div id="search-results" style="position: absolute; background: white; z-index:
1000;"></div>
    </form>
    <ul class="navbar-nav ml-3"> <!-- ml-3 will give some margin to the left of the cart -
->
      <li class="nav-item cart-logo">
        <a class="nav-link" href="/cart">
          
        </a>
      </li>
```

```
</ul>
</nav>
```

This is part of the base.html page. In this page anything between the <head> tags are what's displayed on the webpage itself. These codes are taken from "bootstrap" which are premade CSS files to style a website and can be modified according to the programmer's wishes. In this case these codes generate this "nav bar" as seen below.



```
<div class="navbar-nav">
  <a class="nav-item nav-link" id="login" href="/login">Login</a>
  <a class="nav-item nav-link" id="register" href="/register">Register</a>
  <a class="nav-item nav-link" id="logout" href="/logout">Log out</a>
```

These codes are what create the "Tooltopia" button, the login, register, and Logout button respectively.

```
<form class="form-inline my-2 my-lg-0 ml-auto"> <!-- ml-auto will push the search bar
to the left -->
  <input class="form-control mr-sm-2" type="search" placeholder="Search" aria-
Label="Search" id="search-input">
  <button class="btn btn-outline-success my-2 my-sm-0" type="submit">Search</button>
  <div id="search-results" style="position: absolute; background: white; z-index:
1000;"></div>
```

And this code is what creates the search bar, which I will go more into depth later. When I will explain how the search bar functions.

```
<ul class="navbar-nav ml-3"> <!-- ml-3 will give some margin to the left of the cart -
->
  <li class="nav-item cart-logo">
    <a class="nav-link" href="/cart">
      
```

This code creates the shopping cart button on the right side of the navbar.

Back to the home page of the website.

As I said previously initially when the website is ran this function is ran

```
@app.route('/')
def home_page():
    products = Product.query.all()
    return render_template('main.html', products=products)
```

It renders or opens the main.html page which is this code

```
{% extends "base.html" %}
{% block title %}Homepage{% endblock %}
{% block content %}
```

```

<h1>Products</h1>
{% with messages = get_flashed_messages() %}
    {% if messages %}
        <ul>
            {% for message in messages %}
                <li>{{ message }}</li>
            {% endfor %}
        </ul>
    {% endif %}
{% endwith %}
<div class="product-grid">
    {% for product in products %}
        <div class="product">
            <a href="{{ url_for('product_detail', product_id=product.id) }}">
                
            </a>
            <h3>{{ product.name }}</h3>
            <p>£{{ product.price }}</p>
            <button class="add-to-cart" data-product-id="{{ product.id }}">Add to
Cart</button>
        </div>
    {% endfor %}
</div>
<div id="cart-notification" style="display: none;">Product added to cart!</div>
<script>
    document.addEventListener('DOMContentLoaded', function() {
        var addToCartButtons = document.querySelectorAll('.add-to-cart');
        addToCartButtons.forEach(function(button) {
            button.addEventListener('click', function() {
                var productId = button.dataset.productId;
                var xhr = new XMLHttpRequest();
                xhr.open('POST', '/add_to_cart');
                xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
                xhr.onload = function() {
                    if (xhr.status === 200) {
                        var notification = document.getElementById('cart-notification');
                        notification.style.display = 'block';
                        setTimeout(function() {
                            notification.style.display = 'none';
                        }, 3000);
                    }
                };
                xhr.send('product_id=' + productId);
            });
        });
    });
</script>

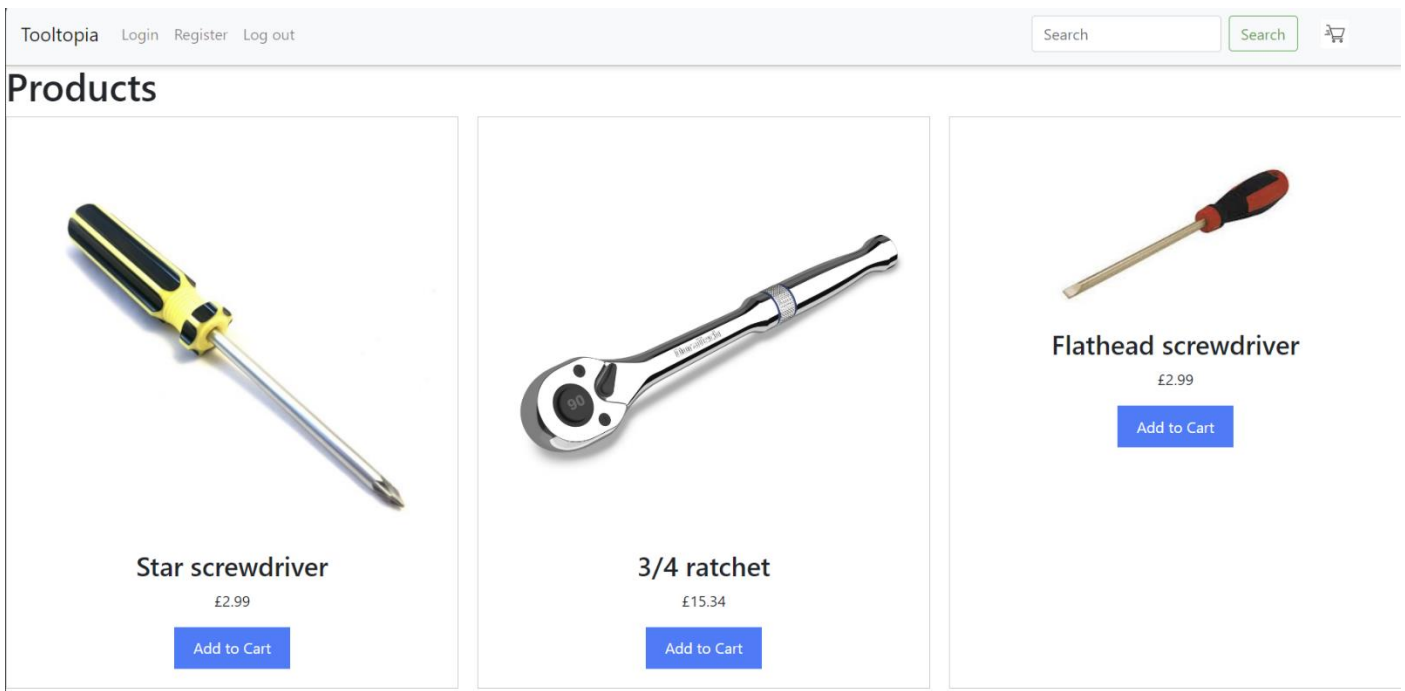
{% endblock %}

```

This html file,"extends base.html"as seen in the first part of the code.This means that the website will inherit whatever is in the base.html file,which in

this case is responsible for showing a navbar as previously mentioned, and additionally add whatever is written between the `{% block content %}` `{% endblock %}`.

The code between the block content tags add all the products in the main page along with all the attributes. These informations are pulled from the database itself. This html file is responsible for displaying this main page as shown below:



If the Tooltopia button is pressed in this page this code is ran `Tooltopia` #this sets the url to / and as we discussed earlier when / this url is present it runs the home page therefore returning us back to this page. This button serves as a home page button in simpler terms.

When the login button is pressed it runs this code `Login`


This code sets the url to /login. this runs a different function in the backend

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(username=form.username.data).first()
        if user is None or not user.check_password(form.password.data):
            flash('Invalid username or password')
            return redirect(url_for('login'))
        login_user(user, remember=form.remember_me.data)
        # Redirects to the home page
        return redirect(url_for('home_page'))
    return render_template('login.html', title='Sign In', form=form)
```


It runs this function. Where `loginForm()` object captures the user's input. then the following if statement is for user authentication, where it queries the database for the user with the given username. Dates the password using the hash comparison. If credentials are incorrect it flashes a message and re directs back to the login page. If correct it logs the user in and redirects to the home page. Finally when the GET request is made to `/login` the server renders the login page or `login.html`

```
return render_template('login.html', title='Sign In', form=form)
```

This is why we see this page being rendered when going to the login page

[Tooltopia](#) [Login](#) [Register](#) [Log out](#) 

Login

Username

Password

☐ Remember Me

This is the code responsible for the login page named “login.html”

```
<!-- login.html -->
{% extends "base.html" %}

{% block title %}
Login
{% endblock %}

{% block content %}
<h1>Login</h1>
<form action="{{ url_for('login') }}" method="post">
  {{ form.hidden_tag() }}
  <div>
    {{ form.username.label }}
    {{ form.username(size=32) }}
    {% if form.username.errors %}
      {% for error in form.username.errors %}
        <div>{{ error }}</div>
      {% endfor %}
    {% endif %}
  </div>
  <div>
    {{ form.password.label }}
    {{ form.password(size=32) }}
    {% if form.password.errors %}
      {% for error in form.password.errors %}
        <div>{{ error }}</div>
      {% endfor %}
    {% endif %}
  </div>
</form>
</div>
```

```

</div>
<div>
    {{ form.remember_me() }} Remember Me
</div>
<div>
    {{ form.submit() }}
</div>
</form>
{% endblock %}

```

Form Tag: The form uses the POST method to submit data securely to the server, with the action set to the login route, which handles form submission.

CSRF Protection: {{ form.hidden_tag() }} includes a hidden field that protects from CSRF attacks by embedding a token in the form.

Input Fields:

Username and Password Fields: Include input fields for the username and password. The size attribute specifies the width of the input field.

Validation Errors: If there are any errors from previous submission attempts (like incorrect username or password), they are displayed under the respective fields.

Remember Me Checkbox: Provides an option for users to remain logged in even after closing the browser.

Submit Button: Triggers the form submission.

Register:

The other button in the home page is the register button. When pressed,

```

<a class="nav-item nav-link" id="register" href="/register">Register</a>

```

This code is ran which sets the url to /register. This causes the /register route to be ran.

/register route's code

```

@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
    if form.validate_on_submit():
        existing_user = User.query.filter_by(username=form.username.data).first()
        if existing_user is not None:
            flash('Username already taken. Please choose a different one.')

```

```

    return redirect(url_for('register'))

    user = User(username=form.username.data)
    user.set_password(form.password.data)
    db.session.add(user)
    db.session.commit()
    flash('Congratulations, you are now a registered user!')
    return redirect(url_for('login'))

```

Form Instantiation and Validation: A “RegistrationForm” instance is created, and if the form is submitted, the server validates the input.

User Creation:

Username and Password: The username and password are extracted from the form data. It is also compared with the existing users in the database and if it matches with an existing user it flashes an error message

Password Hashing: The password is hashed using a method in the User model for secure storage.

Database Interaction: The new User object is added to the database.

Success Feedback: A flash message confirms successful registration.

Redirection: After successful registration, the user is redirected to the login page to sign in with their new credentials.

There also is again an html page for the registering process here is the code for register.html

```

<!-- register.html -->
{% extends "base.html" %}

{% block title %}
Register
{% endblock %}

{% block content %}
<h1>Register</h1>
<form action="{{ url_for('register') }}" method="post">
    {{ form.hidden_tag() }}
    <div>
        {{ form.username.label }}
        {{ form.username(size=32) }}
        {% if form.username.errors %}
            {% for error in form.username.errors %}

```

```

        <div>{{ error }}</div>
    {% endfor %}
{% endif %}
</div>
<div>
    {{ form.password.label }}
    {{ form.password(size=32) }}
    {% if form.password.errors %}
        {% for error in form.password.errors %}
            <div>{{ error }}</div>
        {% endfor %}
    {% endif %}
</div>
<div>
    {{ form.submit() }}
</div>
</form>
{% endblock %}

```

Action and Method: The form submits data to the /register route using a POST method to ensure data privacy.

CSRF Protection: {{ form.hidden_tag() }} is used to include a CSRF token in the form, which protects against CSRF attacks.

User Fields: Input fields for username and password. Each field uses Flask-WTF to render and validate input on the server side.

Submit Button: Users click this button to submit their registration details.

Hashing

It is also worth noting that passwords are not stored as plain text in the database as this would pose a security risk. It is instead run through a hashing algorithm and a salting algorithm which means they can not be converted to plain text easily.

This hashing algorithm is implemented in the following way.

```
def set_password(self, password):
    self.password_hash = generate_password_hash(password)
```

when we set up the database with the user model in the beginning we defined a method named set_password. This method is responsible for taking plain text password and converting it into hashed version.

this is also used in the registration process, When a new user registers, the `set_password` method is called with the plain text password they provided. This occurs within the route handling the registration form submission

```
if form.validate_on_submit():
    user = User(username=form.username.data)
    user.set_password(form.password.data) # Hashing the password
    db.session.add(user)
    db.session.commit()
    flash('Congratulations, you are now a registered user!')
    return redirect(url_for('login'))
```

When the form is submitted and validated, a new User instance is created.

The `set_password` method is called with the user's plain text password.

The method hashes the password and stores the hash in the `password_hash` attribute.

The user object (with the hashed password) is then saved to the database.

Logout button

When the logout button is pressed it sets the url to `/logout`

Which again runs the `/logout` route. which is the following

```
@app.route('/Logout')
def logout():
    logout_user()
    return redirect(url_for('home_page'))
```

Route Definition: The `@app.route('/logout')` decorator defines the URL endpoint for the logout process. When a user navigates to `/logout`, the `logout()` function is triggered.

Function Definition: The `logout()` function handles the actual process of logging out.

logout_user() Function: This is a function provided by Flask-Login that removes the user from the session. It effectively "logs out" the user by clearing any user-related data stored in the session. It ensures that the session is immediately invalidated. This means that any further requests made using the same session will not be recognized as authenticated, thereby preventing unauthorized access.

Redirection: After logging out the user, the function redirects the user to the `home_page`. This is a typical practice to return users to a public area of the site

where no authentication is required, ensuring that access to user-specific areas is securely restricted post-logout.

The `logout_user()` function ensures that the session is immediately invalidated. This means that any further requests made using the same session will not be recognized as authenticated, thereby preventing unauthorized access.

Product display in the main/home page.

Flask Route for Home Page

The process begins with the Flask route that serves the home page. This route queries the database for all products and passes them to the `main.html` template.

```
@app.route('/')
def home_page():
    products = Product.query.all() # Retrieve all products from the database
    return render_template('main.html', products=products)
```

Querying Products: The `Product.query.all()` statement retrieves all entries in the `Product` table from the database, effectively collecting all product information available.

Rendering Template: The `render_template` function is called with `main.html` and a list of products. This function dynamically populates the HTML template with data from the product list.

HTML Template (main.html)

The HTML template uses Jinja2 templating engine (which is integrated into Flask) to loop through the list of products and display them on the page.

```
{% extends "base.html" %}

{% block content %}
<h1>Products</h1>
<div class="product-grid">
    {% for product in products %}
    <div class="product">
        <a href="{{ url_for('product_detail', product_id=product.id) }}">
            
        </a>
        <h3>{{ product.name }}</h3>
        <p>£{{ product.price }}</p>
        <button class="add-to-cart" data-product-id="{{ product.id }}">Add to Cart</button>
    </div>
    {% endfor %}
</div>
{% endblock %}
```

Template Inheritance: The template extends base.html, which might include common elements like headers, navigation bars, and footers.

Product Loop: Inside the product-grid div, a Jinja2 loop iterates over each product passed to the template. For each product, a block of HTML is used to display its details.

Product Link and Image: Each product image is wrapped in an anchor tag (<a>) that links to a detailed view (product_detail route) for that product. The url_for function generates these URLs dynamically.

Product Information: The product name and price are displayed. Jinja2 expressions ({{ }}) are used to insert data into the HTML.

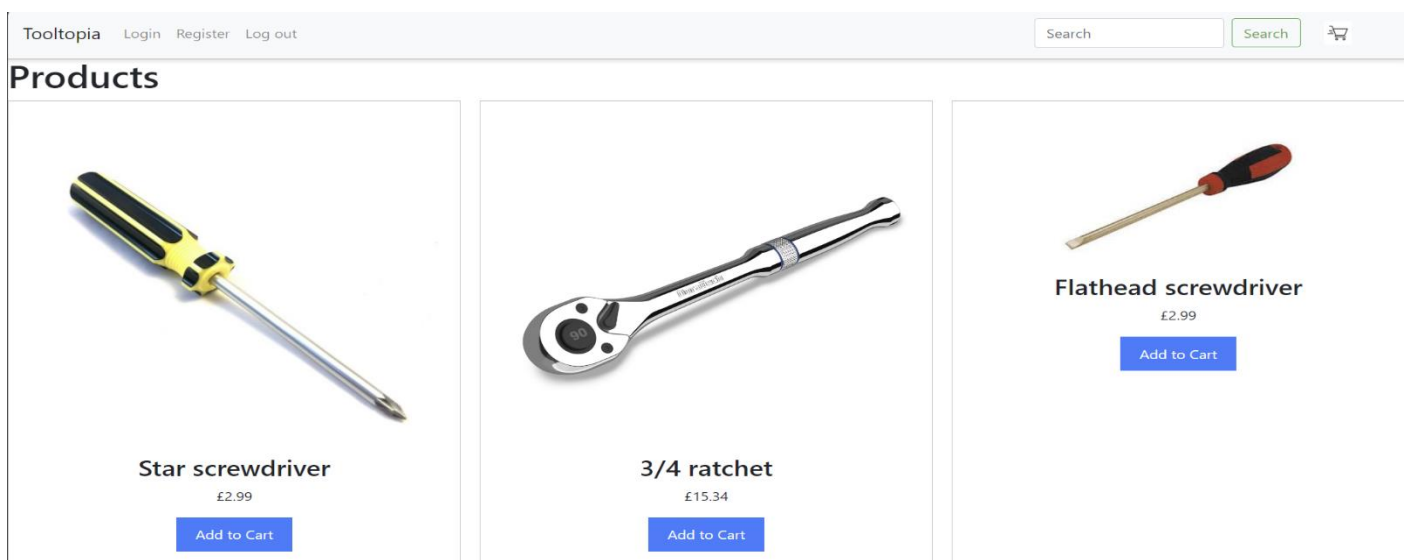
Add to Cart Button: Includes a button that might trigger a JavaScript function or another route to add the product to the user's shopping cart.

CSS and javascript integration

While the HTML code structures the content, CSS would typically be used to style the layout, and JavaScript might handle interactions like the "Add to Cart" button. Here's the css code used for this page and the product's styling and layout.

```
.product-grid {  
    display: grid;  
    grid-template-columns: repeat(3, 1fr);  
    gap: 20px;  
}  
.product img {  
    width: 100%;  
    height: auto;  
}
```

This code defines the layout the products should be in, so in this case a 3 , 1 layout like so:



```

<script>
  document.addEventListener('DOMContentLoaded', function() {
    var addToCartButtons = document.querySelectorAll('.add-to-cart');
    addToCartButtons.forEach(function(button) {
      button.addEventListener('click', function() {
        var productId = button.dataset.productId;
        var xhr = new XMLHttpRequest();
        xhr.open('POST', '/add_to_cart');
        xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
        xhr.onload = function() {
          if (xhr.status === 200) {
            var notification = document.getElementById('cart-notification');
            notification.style.display = 'block';
            setTimeout(function() {
              notification.style.display = 'none';
            }, 3000);
          }
        };
        xhr.send('product_id=' + productId);
      });
    });
  });
</script>

```

```

document.addEventListener('DOMContentLoaded', function() {
  ...
});

```

This event listener waits for the entire HTML document (DOM) to be fully loaded and parsed before executing the enclosed script. This ensures that all HTML elements, especially those you reference in your script, are available for manipulation.

Functionality: Encapsulates your script to ensure it runs only after the page is fully ready, preventing errors that might occur if the script tries to access elements that haven't yet been loaded.

Selecting Elements

```

var addToCartButtons = document.querySelectorAll('.add-to-cart');

```

Purpose: This line selects all elements with the class name 'add-to-cart' and stores them in the `addToCartButtons` variable.

- Functionality: `document.querySelectorAll` returns a NodeList containing all the elements that match the specified CSS class. In this case, it targets all "Add to Cart" buttons on the page.

```

addToCartButtons.forEach(function(button) {

```



```
button.addEventListener('click', function() {
    ...
});
})
```

- Purpose: Iterates over each button in the `addToCartButtons` NodeList and attaches an event listener to each one.

-Functionality: The event listener triggers when a button is clicked. It defines the behavior that occurs on each "Add to Cart" button click, making the buttons interactive.

Handling the Button Click

```
var productId = button.dataset.productId;
var xhr = new XMLHttpRequest();
xhr.open('POST', '/add_to_cart');
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
...
xhr.send('product_id=' + productId);
```

- Data Extraction*: `button.dataset.productId` retrieves the 'productId' data attribute from the button, which should store the unique identifier for the product associated with the button.

AJAX Request Setup:

- XHR Object: `new XMLHttpRequest()` creates a new AJAX request object.

Open: `xhr.open('POST', '/add_to_cart')` initializes a new request using the POST method to the '/add_to_cart' route on your server.

Set Request Header: `xhr.setRequestHeader(...)` sets the content type of the request body to URL-encoded form data, which is necessary for sending data in the POST body.

- Sending the Request: `xhr.send('product_id=' + productId)` sends the AJAX request to the server with the product ID included in the request body.

Handling the Server Response,

```
xhr.onload = function() {
    if (xhr.status === 200) {
        var notification = document.getElementById('cart-notification');
        notification.style.display = 'block';
        setTimeout(function() {
            notification.style.display = 'none';
        }, 3000);
    }
};
...

```

-Response Handling: This function runs when the AJAX request completes.

-Check Status: ``xhr.status === 200`` checks if the HTTP status code is 200, indicating a successful response.

Notification Display:

Find Notification Element: ``document.getElementById('cart-notification')`` gets the DOM element for displaying cart notifications.

Display Notification: Sets the display style to 'block', making it visible.

Auto-hide Notification: ``setTimeout(..., 3000)`` sets a timer to change the display style back to 'none' after 3 seconds (3000 milliseconds), hiding the notification again.

TESTING

Test case: Checking if the products are taken correctly from the database and are displayed properly. Expected result: 8 products should be displayed along with their images, price, name and an add to cart button.

The actual Result:

Products



Star screwdriver

£2.99

Add to Cart



3/4 ratchet

£15.34

Add to Cart



Flathead screwdriver

£2.99

Add to Cart



ignition coil denso type2

£49.99

Add to Cart



Denso 1KW starter

£149.99

Add to Cart

The result shows that the products are being displayed as expected in a row of 3 products, where each product has an add to cart button, the image, the name, the price.

TEST case 2: Check the registration system

Expected result: There should be a page where there is registration written in top left and two fields one for a username and password. Once entered valid credentials it should give a confirmation message and redirect to the login page. If a known username is used it will throw an error message alerting the user that it is taken.

Actual result:

Register


Username

Password

Register

Register

Username

Password 

Syed was used as a username and password as 1234

Which is a new account with an unused username and password.

Login

Jsername

assword

☐ Remember Me

Congratulations, you are now a registered user!

We can see that the first part of our test aligns with our expected result. Now it will be tested what happens if a known username is used.

We will register again using the username syed again.

Register

Jsername

assword

Jsername already taken. Please choose a different one.

the result again aligns with the expected result, the website does indeed throw an error and denies the creation of that account.


Test case 3:Testing the login system.

Expected result:Login should be written in top left and two input fields should appear username and password.if valid credentials are entered and it matches those stored in the database,it should let the user login and return them to the homepage,otherwise advise the user of an error.

Actual result:

Login


Username

Password 

☐ Remember Me


the username syed and password 1234 is entered which are valid credentials.

Products



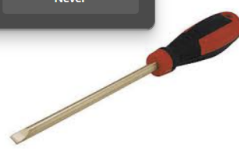
Star screwdriver

£2.99



3/4 ratchet

£15.34



Flathead screwdriver

£2.99

No need to remember your passwords any more

We can see that they have indeed been redirected to the homepage.

In the following image we used an unknown user name ab and 5 as password.

Login

Username

Password

☐ Remember Me

Sign In

Invalid username or password

Login

Username

Password



☐ Remember Me

Sign In

we can see that they have been notified that either the username or password is incorrect.

Test case 4:product detail page.

Expected result:When clicking on each product in the main/home page it should redirect us to a page where there is all the info of that product(image,price,description) and also the stock levels,and should also include an add to cart button.

Actual result:In this example we will click the Denso k20hr-u11 product,



Denso k20hr-u11

£59.99

Add to Cart

Denso k20hr-u11

£59.99

Denso spark plug copper tip for great combustion performance

In stock: 4

Add to Cart



We can see that it all aligns with the expected result.

Test case 5:the add to cart button.

Expected result:The add to cart button should add that product to the cart either from the main page or from the product detail page,it should both add it to the cart.

Actual result:



1/2 ratchet

£20.34

Xtool 1/2 ratchet with 150 degrees rotation, no included sockets bit

In stock: 59

Add to Cart

Both these product have been added to the cart and should appear in the cart.



Denso k20hr-u11

£59.99

Add to Cart

Your Cart



Denso k20hr-u11

£59.99

1

Update

Remove



1/2 ratchet

£20.34

1

Update

Remove

Total Price: £80.33

Buy

We can see that both products have indeed been added to the cart.

Test case number 6:testing the cart's functionality

Expected result:The cart should only be accessible to logged in user,and where the users cart content is also saved to their account and so should be accessible when logging back in.Also when products are added it should display total price a buy button and additionally, quantity function and a remove button that removes that item from the cart.both functionality should also update the cart's total price in real time and

display a confirmation message of each action taking place. Actual result:

Login

Username

Password

....



☐ Remember Me

Please log in to access this page.

Login


Username

Password

☐ Remember Me

Please log in to access this page.

Your Cart




Denso k20hr-u11

£59.99

Update

Remove



1/2 ratchet

£20.34

Update


Remove

Total Price: £80.33

Buy

Remove function.along with the message on bottom right.

Your Cart



Denso k20hr-u11

£59.99

Update

Remove

Total Price: £59.99

Buy

Item removed from cart.

And quantity function along with its confirmation message.

Your Cart



Denso k20hr-u11

£59.99

4

Update

Remove

Total Price: £239.96

Buy

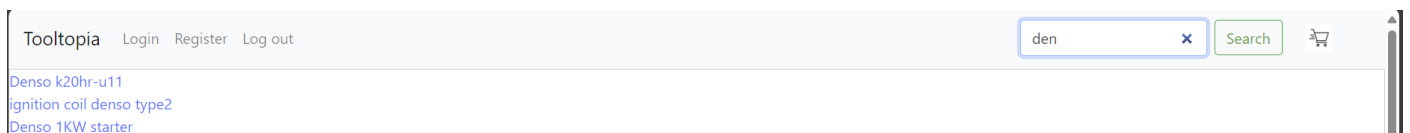
Quantity updated.

Everything is working as expected and the cart items are also saved to the cart of the account with username “abhir”.

Test case 6:search function

Expected result:The search bar should give real time suggestions based on what’s being written in it, and when clicked redirects to that products page.

Actual result:



In this example due to the nature of screenshot in windows it does not show the mouse in the screenshot. But I am going to press on the third option.

[Denso k20hr-u11](#)

1220000



Denso 1KW starter

£149.99

starter for smaller car,specifically for toyota aygo rated power:1KW

In stock: 3

Add to Cart

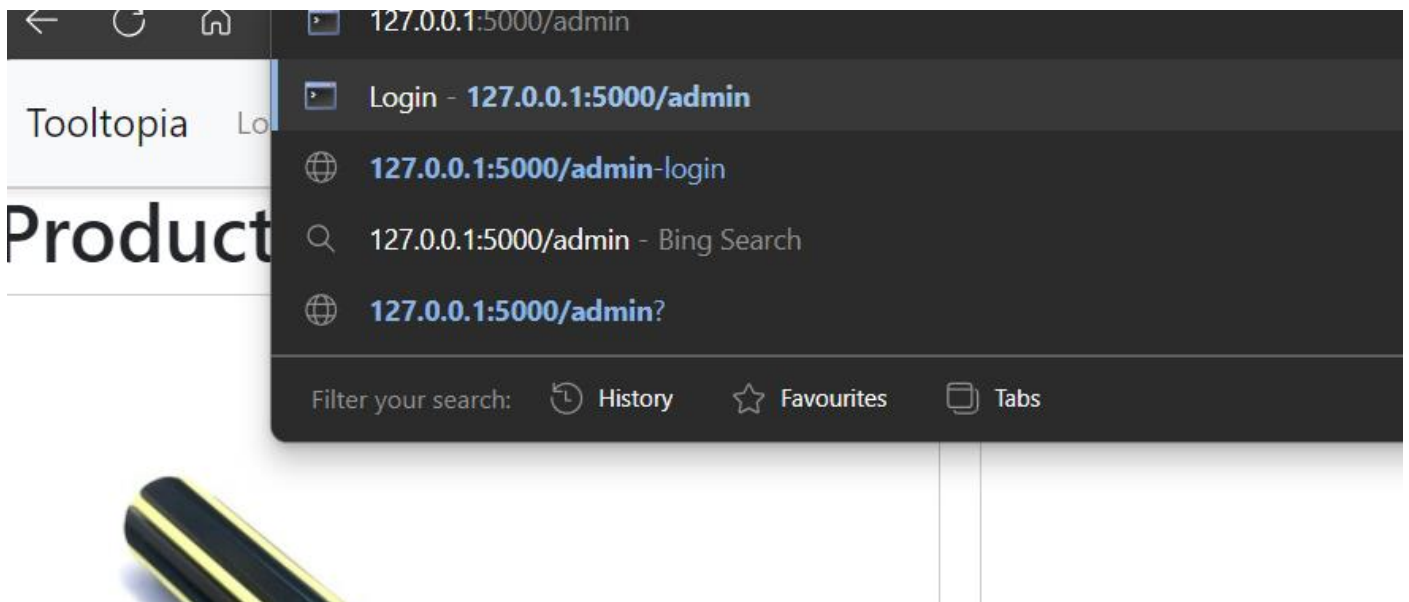
The actual result matches our expected result.

Test case 7:admin page

Expected result: when entering /admin it should redirect to a page where an admin password is asked.if 112911 is entered it gives access to the admin page where there all the products and where we can update the quantities of each product.

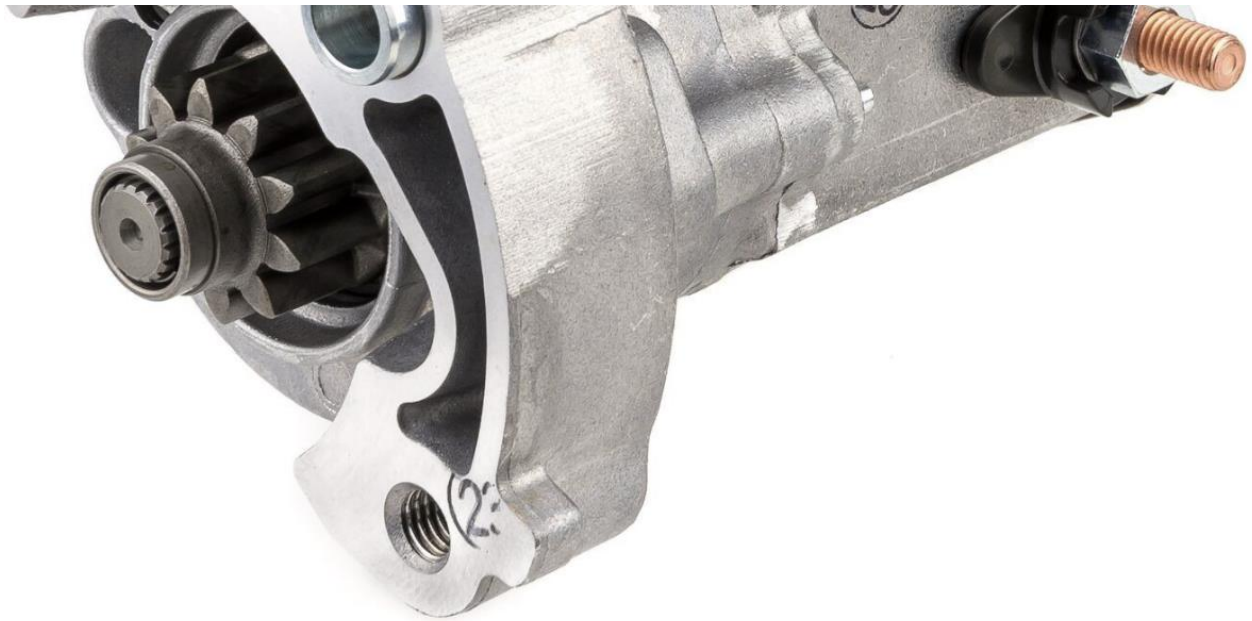
Actual result:

In this test,the denso 1KW starter which is provided in the image above,has a stock level of 3.It will be set to 53.



Admin Login

Admin Password



Denso 1KW starter

£149.99

53

[Tooltopia](#) [Login](#) [Register](#) [Log out](#)

DENSO



Denso 1KW starter

£149.99

starter for smaller car,specifically for toyota aygo rated power:1KW

In stock: 53

[Add to Cart](#)

The results align with the expected result of this test case.

Evaluation

Throughout the development of the Tooltopia website, I successfully met the core objectives, delivering a fully functional e-commerce platform that allows users to browse, add to cart, and purchase tools online. User feedback from both customers and the Company itself highlighted the clean and intuitive interface, and the ease of navigation through it. Though some noted occasional slowdowns during peak usage times, suggesting improvements for performance optimization. Technical challenges included implementing the AJAX-based cart system, which was resolved by refining the JavaScript code. Going forward, adding a recommendation engine and subcategories for each product could significantly enhance user engagement. Additionally, an inventory management system that automatically orders new items would be very beneficial also.

END OF PROJECT