North South University

Department of Electrical and Computer Engineering

CSE327: Software Engineering
Semester: Fall 2025
Section: 06

***CarHub*: Car Reselling Platform**

Submitted To:

AKM Iqtidar Newaz
Professor
NORTH SOUTH UNIVERSITY

Submitted by:

GROUP - 03

NAME	NSU ID
Samina Islam Mim	2311943042
Safwan Ismaun Amin	2311443642
Farhan Karim	2311790642
Safat Uddin	231196642

GitHub Repository: **https://github.com/SafatUddin/CAR_Hub**

1. PROJECT DESCRIPTION:

1.1 OVERVIEW

CarHub is an online platform for buying and selling cars that simplifies and speeds up the process compared to traditional methods like dealerships or newspaper ads. It connects buyers and sellers securely and conveniently, supporting new and used cars. Buyers can find vehicles based on their needs and budget, while sellers can reach a wider audience efficiently.

1.2 PROBLEM STATEMENT

The current car resale market faces major issues for both buyers and sellers.

- **Sellers** struggle to attract enough buyers, manage inquiries, and ensure safe payments.
- **Buyers** face problems verifying car conditions, tracking prices, and finding fair deals.

Existing platforms are limited, they lack advanced search, real-time notifications, secure transactions, and trust mechanisms. Moreover, poor verification often leads to fake listings and unreliable sellers.

1.3 SOLUTION AND KEY FEATURES

CarHub provides a marketplace by integrating advanced features that simplify the buying and selling process while ensuring security, transparency, and user satisfaction.

Advanced Search and Discovery

CarHub offers a powerful search system that allows users to combine multiple filters such as make, model, price range, year, mileage, and vehicle type. Users can refine their searches easily, save preferred criteria, and access tailored results that match their preferences.

Intelligent Price Alerts

Users can subscribe to notifications for specific car listings or price ranges. The system automatically tracks new listings and price changes, sending instant alerts when relevant opportunities arise ensuring buyers never miss the best deals.

Flexible Listing Creation

Sellers can quickly create and customize listings for different types of vehicles, each with relevant details such as warranty, ownership history.

1.4 TECHNICAL ARCHITECTURE AND DESIGN

CarHub is built on an architecture designed for scalability, reliability, and ease of maintenance. The system structure allows different components to work independently while maintaining seamless integration, enabling smooth updates and future feature expansion.

The platform emphasizes clear separation of concerns, efficient resource management, and reusable components. This ensures that new functionalities can be added with minimal disruption to existing systems, supporting long-term growth and adaptability as user needs evolve.

1.5 USER EXPERIENCE AND INTERFACE DESIGN

CarHub features a clean, responsive design for desktop and mobile users.

- Buyers can browse listings, manage alerts.
- Sellers access a dashboard to manage listings, and respond to inquiries.

The interface follows modern design principles—simple navigation, clear call-to-actions, and visual hierarchy for a smooth user journey.

1.6 SECURITY AND TRUST

CarHub prioritizes safety through:

- Secure password hashing.
- Protection against SQL injection.

An administrative moderation system ensures listing verification and user accountability, promoting trust and transparency across the platform.

2. END USERS:

2.1 PRIMARY USERS

Buyers: Individuals looking to purchase vehicles (new or used)

Sellers: Individual car owners looking to sell their vehicles or small-scale car dealers.

2.2 SECONDARY USERS

System Administrators: Admins to manage the data of Sellers and Buyers.

3. PROPOSED FUNCTIONAL REQUIREMENTS:

3.1 USER MANAGEMENT

a) User Registration

- Users can register as buyers or sellers with email and password.
- System validates unique email addresses.
- Users must verify email before full account access.

b) User Authentication

- Users can log in with email and password.
- System maintains secure sessions.
- Users can reset forgotten passwords.

c) User Profile Management

- Users can view and edit their profile information.
- Users can upload profile pictures.

3.2 CAR LISTING MANAGEMENT

a) Create Listings

- Sellers can create new car listings with detailed information (make, model, year, price, mileage, condition).
- System supports multiple car types: new cars or used cars.
- Sellers can upload images per listing.

b) Edit and Delete Listings

- Sellers can edit their active listings.
- Sellers can delete or mark listings as sold.

c) View Listings

- All users can browse available car listings
- Users can view detailed information about each car
- System displays seller contact information on listings

3.3 SEARCH AND FILTER

a) Search

- Users can search cars using specific criteria
- The system returns relevant results based on search query

b) Filter Options

- Users can filter search results by various attributes
- Users can sort results by price, year, mileage, or date posted

3.4 PRICE ALERT SYSTEM

a) Subscribe to get Alerts

- Buyers can subscribe to specific sellers/criteria for alerts.
- System sends alerts to users who are subscribed.
- Users can subscribe or unsubscribe.

b) Alert Notifications

- System automatically checks for matching listings when prices update.
- Users receive notifications when criteria are met

3.5 COMMUNICATION

a) Contact Seller

- Buyers can send inquiries to sellers through the platform.
- Users receive notifications for new messages

b) Notifications

- Users receive notifications for important events (new messages, price alerts, listing updates)
- System sends notifications for all critical actions

3.6 PAYMENT PROCESSING

a) Secure Payments

- Buyers can make payments through integrated payment gateway.
- System supports multiple payment methods (credit card, debit card).
- Platform processes payment and generates invoice.

b) Transaction Management

- System logs all transactions for audit purposes
- Users receive payment confirmation and receipt.
- System maintains transaction history for both buyers and sellers

3.7 ADMINISTRATIVE FUNCTIONS

a) Listing Moderation

- Admins can review and approve new listings
- Admins can flag or remove inappropriate listings

b) User Management

- Admins can view all registered users
- Admins can suspend or ban accounts

4. PROPOSED NON-FUNCTIONAL REQUIREMENTS:

4.1 PERFORMANCE

- System shall load pages within 2 seconds under normal conditions
- Search queries shall return results within 3 seconds
- System shall be scalable.

4.2 SECURITY

a) Authentication and Authorization

- System shall implement secure password hashing
- System shall use JWT tokens for API authentication
- Role-based access control shall restrict unauthorized access

b) Data Protection

- All sensitive data shall be encrypted.
- System shall implement SQL injection prevention.
- User personal information shall not be shared without consent

4.3 RELIABILITY

- System shall handle errors gracefully with user-friendly messages
- Critical errors shall be logged for debugging
- System shall recover from failures without data loss

4.4 USABILITY

- Interface should be intuitive, consistent, and responsive across all devices.
- Error messages should be clear and helpful.

4.5 MAINTAINABILITY

- System architecture should allow easy updates and feature extensions.
- Components shall be loosely coupled and highly cohesive

4.6 COMPATIBILITY

- System shall support latest versions of Chrome, Firefox, Safari, and Edge.
- System shall be compatible with PostgreSQL or whatever database we use.
- System shall integrate with standard payment gateways.

5. SYSTEM CONSTRAINTS:

a) Technical Constraints:

- Development using Python/Django framework
- PostgreSQL as primary database
- Cloud deployment.

b) Project Constraints:

- Project timeline: One month
- Budget limitations for third-party services
- Team size: 4 members

6. IMPLEMENTATION STATUS AND DESIGN VALIDATION

a) Functional Requirements

Requirement ID	Module	Key Features Implemented	Design Patterns Used	Implementation Status
FR 3.1	User Management	Registration with unique email, login via email/username, profile edit	Proxy (access checks)	Fully Implemented *(profile picture & password reset missing)*
FR 3.2	Car Listing Management	Create, edit, delete, mark sold, image upload, approval workflow	Factory, Proxy	Fully Implemented
FR 3.3	Search & Filter	Multi-criteria search, sorting, currency-based price filter	Strategy, Adapter	Fully Implemented
FR 3.4	Price Alert System	Follow cars, notify on price change	Observer	Partially Implemented *(no criteria-based alerts)*
FR 3.5	Communication	Email & WhatsApp contact, in-app notifications	Observer	Fully Implemented
FR 3.6	Payment Processing	Mock payment flow, transaction logs, invoices, order status	—	Half Implemented *(no real gateway)*
FR 3.7	Administrative Functions	Listing approval/rejection, user viewing	Proxy	Fully Implemented *(no ban/suspend)*

b) Non-Functional Requirements

NFR ID	Category	Implementation Summary	Status
NFR 4.1	Performance	Basic ORM queries, no caching or monitoring	Partially Implemented
NFR 4.2	Security	Password hashing, RBAC, ORM-based SQL protection	Partially Implemented *(no JWT, no encryption)*
NFR 4.3	Reliability	Graceful error handling, transactional integrity	Implemented
NFR 4.4	Usability	Responsive UI, consistent navigation, clear messages	Implemented
NFR 4.5	Maintainability	Modular architecture, extensive design pattern usage	Implemented
NFR 4.6	Compatibility	Modern browser support, SQLite/MySQL support	Partially Implemented

c) Design Patterns Summary

Pattern	Purpose	Used In
Factory	Create different car types dynamically	Car creation
Singleton	Single shared configuration instance	Database configuration
Observer	Notify users on price changes	Price alerts, notifications
Strategy	Flexible search logic	Search & filtering
Decorator	Add optional car features with cost	Car feature pricing
Proxy	Role-based access control	Listing & admin actions
Adapter	Currency conversion	Price display & filtering

c) Screenshots of code & UI

i) Functional Requirements

```python
def signup(request):
    # If user is already logged in, redirect to home
    if request.user.is_authenticated:
        return redirect('home')

    if request.method == 'POST':
        form = SignUpForm(request.POST)
        if form.is_valid():
            user = form.save()
            login(request, user)
            messages.success(request, "Registration successful!")
            return redirect('home')
    else:
        form = SignUpForm()
    return render(request, 'registration/signup.html', {'form': form})
```

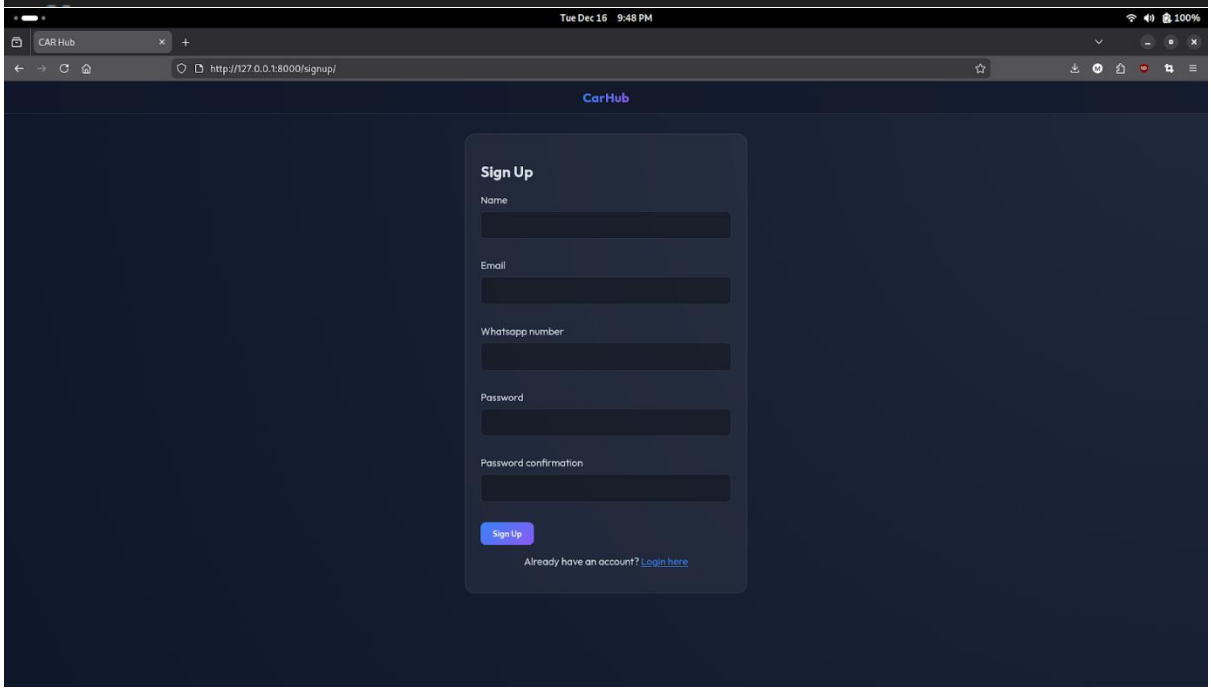
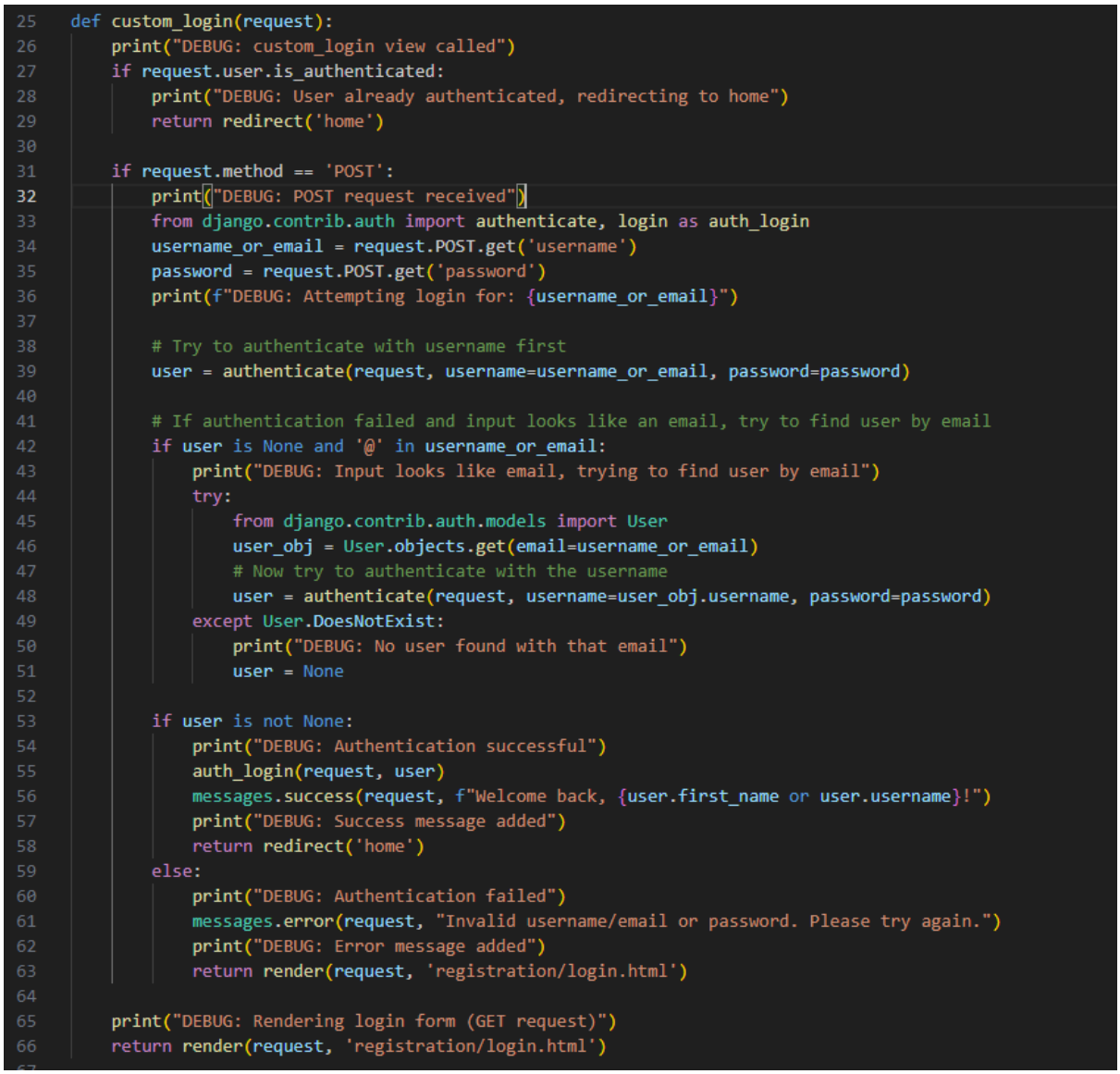


Fig.6.1.1: User Registration



```python
def custom_login(request):
    print("DEBUG: custom_login view called")
    if request.user.is_authenticated:
        print("DEBUG: User already authenticated, redirecting to home")
        return redirect('home')

    if request.method == 'POST':
        print("DEBUG: POST request received")
        from django.contrib.auth import authenticate, login as auth_login
        username_or_email = request.POST.get('username')
        password = request.POST.get('password')
        print(f"DEBUG: Attempting login for: {username_or_email}")

        # Try to authenticate with username first
        user = authenticate(request, username=username_or_email, password=password)

        # If authentication failed and input looks like an email, try to find user by email
        if user is None and '@' in username_or_email:
            print("DEBUG: Input looks like email, trying to find user by email")
            try:
                from django.contrib.auth.models import User
                user_obj = User.objects.get(email=username_or_email)
                # Now try to authenticate with the username
                user = authenticate(request, username=user_obj.username, password=password)
            except User.DoesNotExist:
                print("DEBUG: No user found with that email")
                user = None

        if user is not None:
            print("DEBUG: Authentication successful")
            auth_login(request, user)
            messages.success(request, f"Welcome back, {user.first_name or user.username}!")
            print("DEBUG: Success message added")
            return redirect('home')
        else:
            print("DEBUG: Authentication failed")
            messages.error(request, "Invalid username/email or password. Please try again.")
            print("DEBUG: Error message added")
            return render(request, 'registration/login.html')

    print("DEBUG: Rendering login form (GET request)")
    return render(request, 'registration/login.html')
```

From: views.py

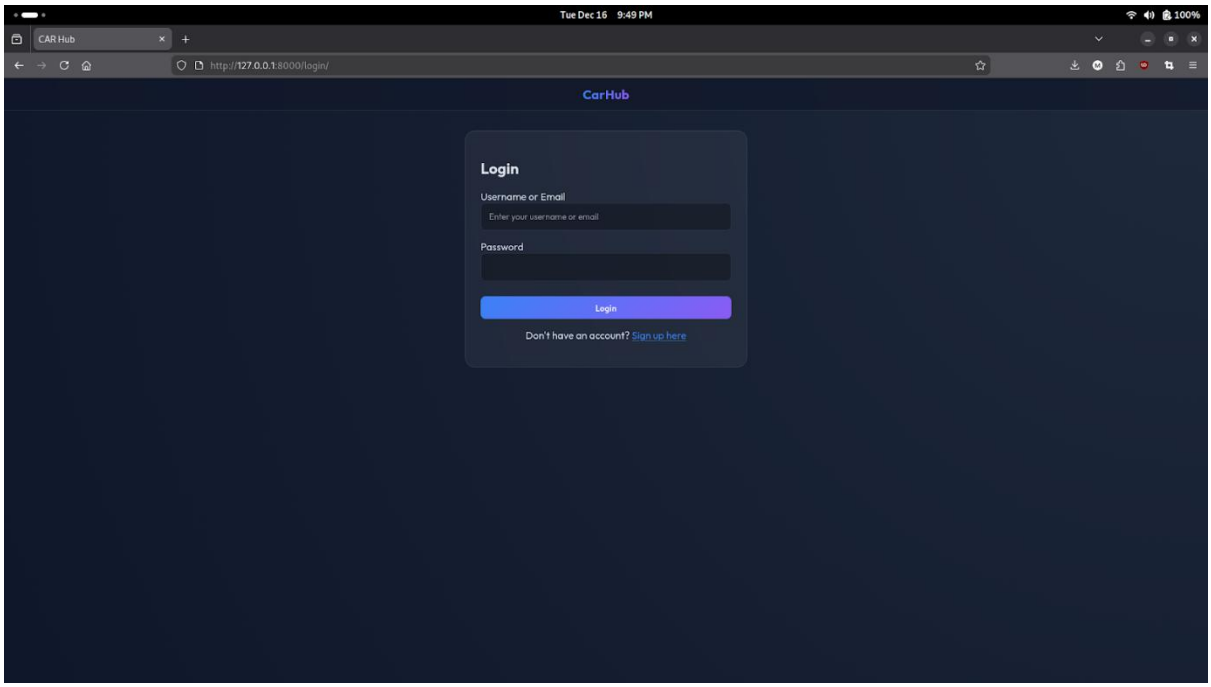


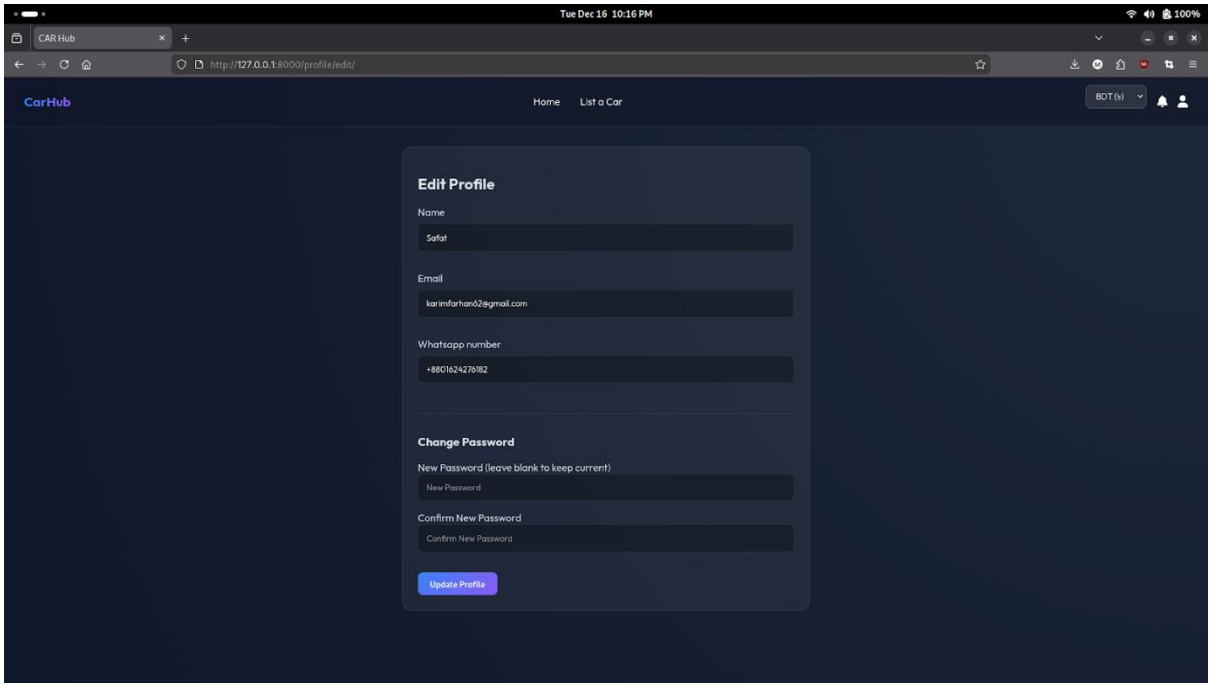
Fig.6.1.2: User Authentication

```python
@login_required
def edit_profile(request):
    if request.method == 'POST':
        form = EditProfileForm(request.POST, instance=request.user)
        if form.is_valid():
            user = form.save()

            # Handle Password Change
            new_password = request.POST.get('new_password')
            confirm_password = request.POST.get('confirm_password')

            if new_password:
                if new_password == confirm_password:
                    user.set_password(new_password)
                    user.save()
                    login(request, user)
                    messages.success(request, "Profile and password updated successfully!")
                else:
                    messages.error(request, "Passwords do not match.")
                    return render(request, 'cars/edit_profile.html', {'form': form})
            else:
                messages.success(request, "Profile updated successfully!")

            return redirect('profile')
    else:
        form = EditProfileForm(instance=request.user)

    return render(request, 'cars/edit_profile.html', {'form': form})
```

From: views.py

```
<div class="card" style="max-width: 800px; margin: 0 auto;">
    <h2>My Profile</h2>
    <div style="margin-bottom: 2rem;">
        <p><strong>Name:</strong> {{ user.first_name }}</p>
        <p><strong>Email:</strong> {{ user.email }}</p>
        {% if user.profile.whatsapp_number %}
        <p><strong>WhatsApp:</strong> {{ user.profile.whatsapp_number }}</p>
        {% endif %}
        <p><strong>Date Joined:</strong> {{ user.date_joined|date:"F j, Y" }}</p>
        <a href="{% url 'edit_profile' %}" class="btn btn-primary" style="margin-top:1rem;">Edit Profile</a>
    </div>
```

From: profile.html



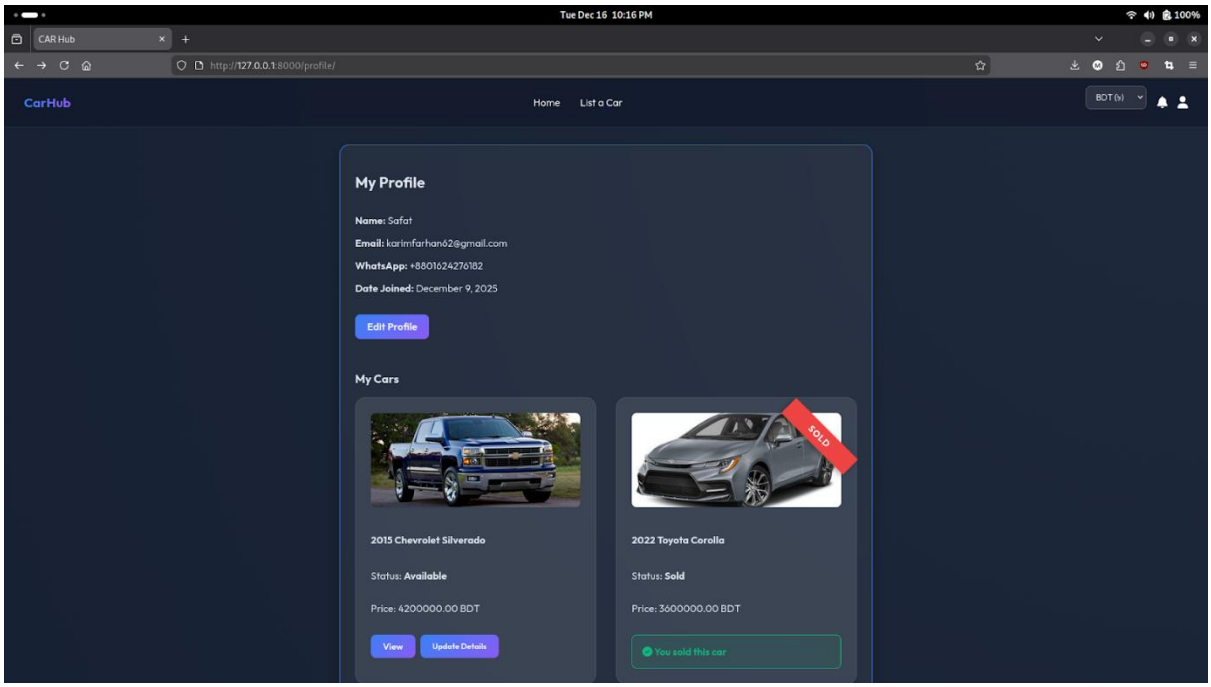


Fig.6.1.3: User Profile Management

```python
@login_required
def create_car(request):
    # Proxy Pattern check
    proxy = CarAccessProxy(request.user)
    allowed, msg = proxy.post_car({})
    if not allowed:
        messages.error(request, msg)
        return redirect('home')

    if request.method == 'POST':
        make = request.POST.get('make')
        model = request.POST.get('model')
        year = request.POST.get('year')
        price_input = float(request.POST.get('price'))
        currency_input = request.POST.get('currency', 'BDT')
        mileage = request.POST.get('mileage')
        car_type = request.POST.get('car_type')
        contact_email = request.POST.get('contact_email', '').strip()
        contact_whatsapp = request.POST.get('contact_whatsapp', '').strip()
        images = request.FILES.getlist('images')
        registration_paper = request.FILES.get('registration_paper')

        # Validate Registration Paper
        if not registration_paper:
            messages.error(request, "Registration paper is required. Please upload the vehicle registration document.")
            return render(request, 'cars/create.html')

        # Validate Contact Info
        if not contact_email and not contact_whatsapp:
            messages.error(request, "You must provide at least one contact method (Email or WhatsApp).")
            return render(request, 'cars/create.html')

        # Validate email format
        if contact_email:
            from django.core.validators import validate_email
            from django.core.exceptions import ValidationError
            try:
                validate_email(contact_email)
            except ValidationError:
                messages.error(request, "Invalid email address format.")
                return render(request, 'cars/create.html')

            # Check if domain has valid MX records
            if not validate_email_domain(contact_email):
                messages.error(request, "This email domain does not exist or cannot receive emails. Please check your email address.")
                return render(request, 'cars/create.html')
```

```
        # Validate WhatsApp format
        if contact_whatsapp:
            is_valid, error_msg = validate_whatsapp_number(contact_whatsapp)
            if not is_valid:
                messages.error(request, error_msg)
                return render(request, 'cars/create.html')

        # Validate registration paper format
        if not registration_paper.name.endswith('.pdf'):
            messages.error(request, "Registration paper must be a PDF file.")
            return render(request, 'cars/create.html')

        # Convert to BDT
        price_bdt = CurrencyAdapter.convert_to_bdt_static(price_input, currency_input)

        if len(images) < 1:
            messages.error(request, "You must upload at least 1 image.")
            return render(request, 'cars/create.html')

        if len(images) > 5:
            messages.error(request, "You can upload a maximum of 5 images.")
            return render(request, 'cars/create.html')

        # Factory Pattern
        factory = None
        if car_type == 'sedan':
            factory = SedanFactory()
        elif car_type == 'suv':
            factory = SUVFactory()
        elif car_type == 'truck':
            factory = TruckFactory()
        elif car_type == 'coupe':
            factory = CoupeFactory()

        if factory:
            car = factory.create_car(make, model, year, price_bdt, mileage, request.user)

            # Save contact info
            car.contact_email = contact_email
            car.contact_whatsapp = contact_whatsapp

            # Save registration paper (required)
            car.registration_paper = registration_paper

            car.save()
```

```
            car.save()

            # Save images
            for image in images:
                CarImage.objects.create(car=car, image=image)

            messages.success(request, "Request for listing Car is sent successfully!")
            return redirect('home')

    import datetime
    current_year = datetime.date.today().year
    year_range = range(current_year, 1939, -1)

    # Get user's contact info from profile
    user_email = request.user.email
    user_whatsapp = ''
    if hasattr(request.user, 'profile'):
        user_whatsapp = request.user.profile.whatsapp_number or ''

    return render(request, 'cars/create.html', {
        'year_range': year_range,
        'user_email': user_email,
        'user_whatsapp': user_whatsapp
    })

```

From: views.py

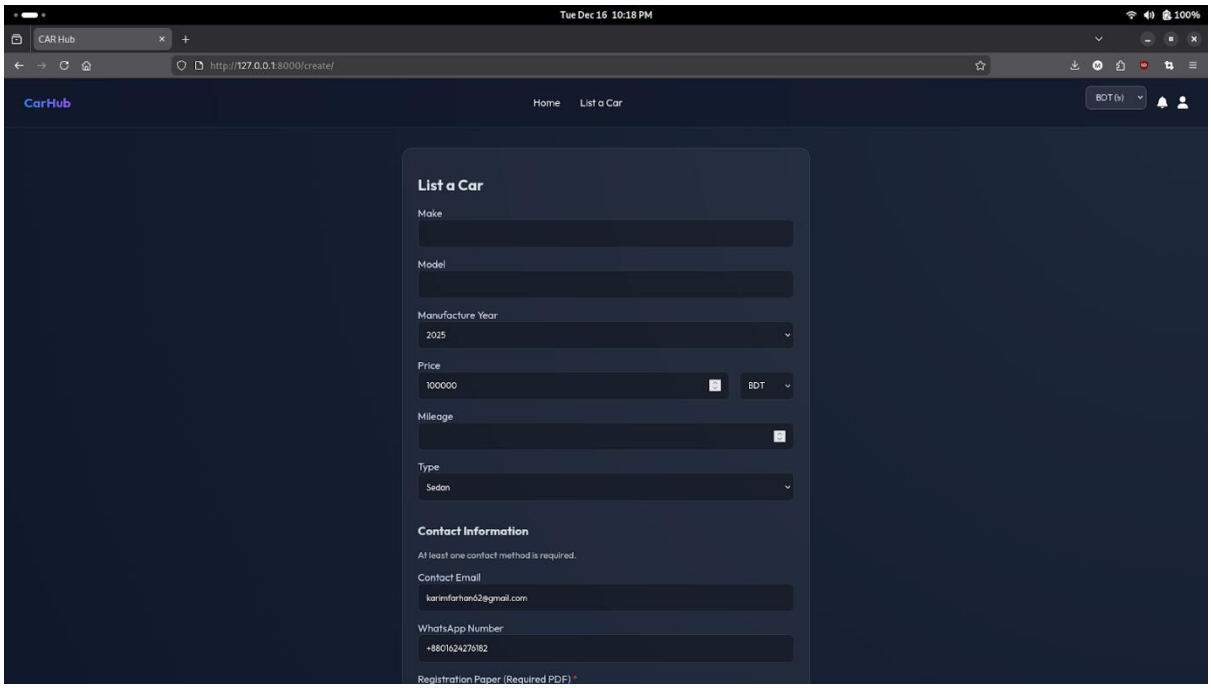


Fig.6.2.1 Create Listings

```python
@login_required
def delete_car(request, car_id):
    # Proxy Pattern
    proxy = CarAccessProxy(request.user)
    success, msg = proxy.delete_car(car_id)

    if success:
        messages.success(request, msg)
    else:
        messages.error(request, msg)

    return redirect('home')
```

```python
@login_required
def update_car(request, car_id):
    car = get_object_or_404(Car, id=car_id)

    if request.user != car.owner:
        messages.error(request, "You are not authorized to edit this car.")
        return redirect('car_detail', car_id=car.id)

    # Prevent editing if car is sold
    if car.status == 'sold':
        messages.error(request, "You cannot edit a car that has been sold.")
        return redirect('profile')

    if request.method == 'POST':
        # Get fields
        make = request.POST.get('make')
        model = request.POST.get('model')
        year = request.POST.get('year')
        price_input = float(request.POST.get('price'))
        currency_input = request.POST.get('currency', 'BDT')
        mileage = request.POST.get('mileage')
        car_type = request.POST.get('car_type')
        contact_email = request.POST.get('contact_email', '').strip()
        contact_whatsapp = request.POST.get('contact_whatsapp', '').strip()
        images = request.FILES.getlist('images')
        registration_paper = request.FILES.get('registration_paper')

        # Validate Contact Info
        if not contact_email and not contact_whatsapp:
            messages.error(request, "You must provide at least one contact method (Email or WhatsApp).")
            import datetime
            current_year = datetime.date.today().year
            year_range = range(current_year, 1939, -1)
            return render(request, 'cars/update.html', {'car': car, 'year_range': year_range})

        # Validate email format
        if contact_email:
            from django.core.validators import validate_email
            from django.core.exceptions import ValidationError
            try:
                validate_email(contact_email)
            except ValidationError:
                messages.error(request, "Invalid email address format.")
                import datetime
                current_year = datetime.date.today().year
                year_range = range(current_year, 1939, -1)
                return render(request, 'cars/update.html', {'car': car, 'year_range': year_range})
```

```
            # Check if domain has valid MX records
            if not validate_email_domain(contact_email):
                messages.error(request, "This email domain does not exist or cannot receive emails. Please check your email address.")
                import datetime
                current_year = datetime.date.today().year
                year_range = range(current_year, 1939, -1)
                return render(request, 'cars/update.html', {'car': car, 'year_range': year_range})

        # Validate WhatsApp format
        if contact_whatsapp:
            is_valid, error_msg = validate_whatsapp_number(contact_whatsapp)
            if not is_valid:
                messages.error(request, error_msg)
                import datetime
                current_year = datetime.date.today().year
                year_range = range(current_year, 1939, -1)
                return render(request, 'cars/update.html', {'car': car, 'year_range': year_range})

        # Convert to BDT
        new_price_bdt = CurrencyAdapter.convert_to_bdt_static(price_input, currency_input)

        # Check if price changed for notification
        old_price = float(car.price)
        price_changed = abs(old_price - new_price_bdt) > 1.0 # Float comparison

        # Update fields
        car.make = make
        car.model = model
        car.year = year
        car.mileage = mileage
        car.car_type = car_type
        car.contact_email = contact_email
        car.contact_whatsapp = contact_whatsapp

        # Note: Registration paper cannot be changed after listing

        # Save all fields except handle price change through Observer pattern
        if not price_changed:
            car.price = new_price_bdt

        car.save()
```

```
            car.save()

            # Add new images
            for image in images:
                CarImage.objects.create(car=car, image=image)

            # Notify followers if price changed (BEFORE updating price in DB)
            if price_changed:
                # Reload car from database to get the current saved price
                car.refresh_from_db()

                # Create subject with the car (car has old price from DB)
                subject = CarPriceSubject(car)

                # Attach all followers as observers (proper Observer pattern)
                for follower in car.followers.all():
                    observer = UserObserver(follower)
                    subject.attach(observer)

                # Change price and notify all attached observers
                subject.change_price(new_price_bdt)

            messages.success(request, "Car details updated successfully!")
            return redirect('car_detail', car_id=car.id)

        import datetime
        current_year = datetime.date.today().year
        year_range = range(current_year, 1939, -1)
        return render(request, 'cars/update.html', {'car': car, 'year_range': year_range})
```

```python
@login_required
def update_car_status(request, car_id):
    car = get_object_or_404(Car, id=car_id)

    if request.user != car.owner:
        messages.error(request, "You are not authorized to perform this action.")
        return redirect('home')

    if request.method == 'POST':
        new_status = request.POST.get('status')
        if new_status in ['available', 'sold']:
            car.status = new_status
            car.save()
            messages.success(request, f"Car status updated to {new_status}.")

    return redirect('car_detail', car_id=car.id)
```

From: views.py

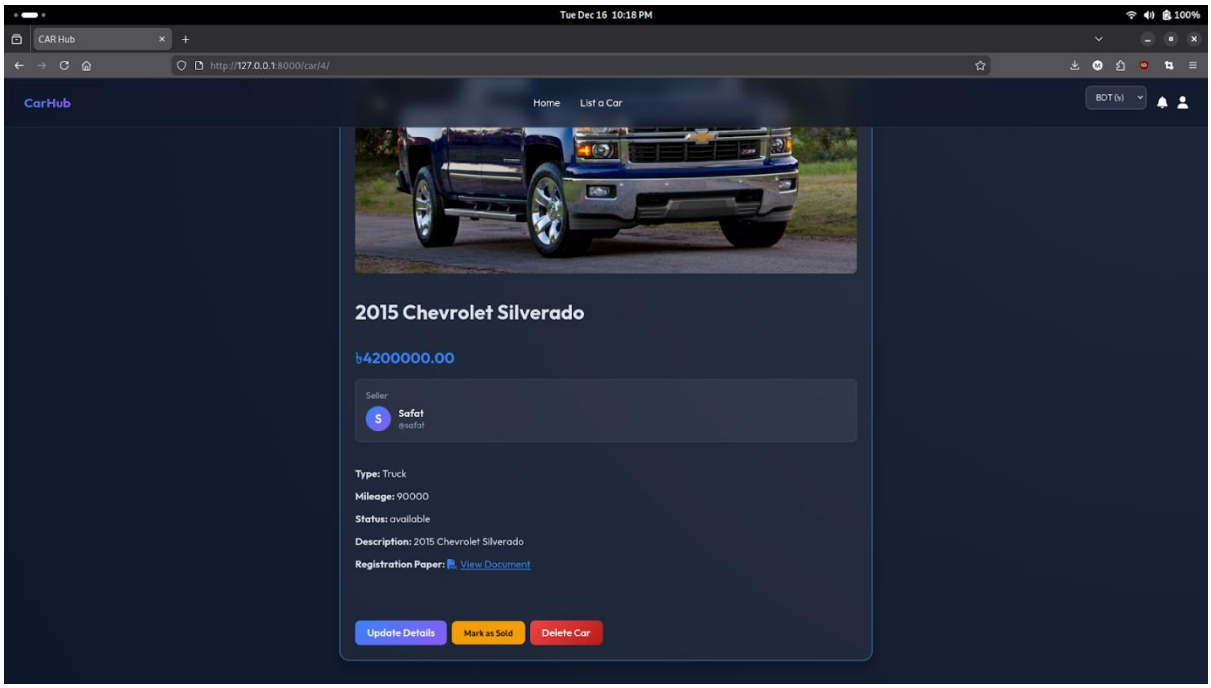


Fig.6.2.2: Edit and Delete Listings

```python
@login_required
def home(request):
    # Require authentication for home page
    if not request.user.is_authenticated:
        return redirect('welcome')

    # Redirect admin to dashboard
    if request.user.is_superuser:
        return redirect('admin_dashboard')

    # Singleton usage (just for demo)
    db_config = DatabaseConfigManager().get_config()

    # Only show approved cars
    cars = Car.objects.filter(approval_status='approved')

    # Strategy Pattern
    search_type = request.GET.get('search_type')
    query = request.GET.get('query')

    from .patterns.strategy import ModelSearchStrategy

    if search_type:
        context = None
        if search_type == 'price':
            min_price = request.GET.get('min_price')
            max_price = request.GET.get('max_price')
            if min_price and max_price:
                try:
                    # Get current currency
                    currency = request.GET.get('currency', request.session.get('currency', 'BDT'))

                    # Convert input prices (in selected currency) to BDT
                    min_price_bdt = CurrencyAdapter.convert_to_bdt_static(float(min_price), currency)
                    max_price_bdt = CurrencyAdapter.convert_to_bdt_static(float(max_price), currency)

                    context = CarSearchContext(PriceSearchStrategy())
                    cars = context.execute_search([min_price_bdt, max_price_bdt])
                    # Filter to only approved cars
                    cars = cars.filter(approval_status='approved')
                except ValueError:
                    pass
        elif search_type == 'brand':
            query = request.GET.get('query')
            if query:
                context = CarSearchContext(BrandSearchStrategy())
```

```
                context = CarSearchContext(BrandSearchStrategy())
                cars = context.execute_search(query)
                # Filter to only approved cars
                cars = cars.filter(approval_status='approved')
        elif search_type == 'model':
            query = request.GET.get('query')
            if query:
                context = CarSearchContext(ModelSearchStrategy())
                cars = context.execute_search(query)
                # Filter to only approved cars
                cars = cars.filter(approval_status='approved')
        elif search_type == 'mileage':
            min_mileage = request.GET.get('min_mileage')
            max_mileage = request.GET.get('max_mileage')
            if min_mileage and max_mileage:
                try:
                    context = CarSearchContext(MileageSearchStrategy())
                    cars = context.execute_search([int(min_mileage), int(max_mileage)])
                    # Filter to only approved cars
                    cars = cars.filter(approval_status='approved')
                except ValueError:
                    pass
        elif search_type == 'type':
            query = request.GET.get('query')
            if query:
                context = CarSearchContext(TypeSearchStrategy())
                cars = context.execute_search(query)
                # Filter to only approved cars
                cars = cars.filter(approval_status='approved')
        elif search_type == 'year':
            query = request.GET.get('query')
            if query:
                try:
                    # Single year selection
                    context = CarSearchContext(YearSearchStrategy())
                    cars = context.execute_search([int(query), int(query)])
                    # Filter to only approved cars
                    cars = cars.filter(approval_status='approved')
                except ValueError:
                    pass
```

From: views.py

```
<div style="margin-bottom: 1.5rem; padding: 1rem; background: rgba(255,255,255,0.05); border-radius: 0.5rem; border: 1px solid var(--glass-border);">
    <p style="margin: 0; color: #94a3b8; font-size: 0.9rem;">Seller</p>
    <a href="{% url 'seller_profile' car.owner.id %}" style="display: flex; align-items: center; gap: 0.75rem; text-decoration: none; color: white; margin-top: 0.5rem;">
        <div style="width: 40px; height: 40px; border-radius: 50%; background: linear-gradient(135deg, var(--primary-color), var(--secondary-color)); display: flex; align-items: cent
            {{ car.owner.first_name.0|upper }}
        </div>
        <div>
            <p style="margin: 0; font-weight: 600; font-size: 1rem;">{{ car.owner.first_name }}</p>
            <p style="margin: 0; color: #94a3b8; font-size: 0.85rem;">@{{ car.owner.username }}</p>
        </div>
    </a>
</div>

<div style="margin-bottom: 2rem;">
    <p><strong>Type:</strong> {{ car.get_car_type_display }}</p>
    <p><strong>Mileage:</strong> {{ car.mileage }}</p>
    <p><strong>Status:</strong> {{ car.status }}</p>
    <p><strong>Description:</strong> {{ description }}</p>
    {% if car.registration_paper %}
    <p><strong>Registration Paper:</strong>
        <a href="{{ car.registration_paper.url }}" target="_blank"
           style="color: var(--primary-color); text-decoration: underline; display: inline-flex; align-items: center; gap: 0.3rem;">
            <i class="fa-solid fa-file-pdf"></i> View Document
        </a>
    </p>
    {% endif %}
</div>
        {% else %}
        <div style="width:100%; margin-bottom:1rem;">
            <h3 style="margin-bottom: 0.5rem; font-size: 1.2rem;">Seller Contact</h3>
            <div style="display:flex; gap:1rem;">
                {% if car.contact_email %}
                <a href="mailto:{{ car.contact_email }}" class="btn"
                    style="background:white; color:black; display:flex; align-items:center; gap:0.5rem; font-weight:600;">
                    <i class="fa-solid fa-envelope"></i> Email Seller
                </a>
                {% endif %}
                {% if car.contact_whatsapp %}
                <a href="https://wa.me/{{ car.contact_whatsapp }}" target="_blank" class="btn"
                    style="background:white; color:black; display:flex; align-items:center; gap:0.5rem; font-weight:600;">
                    <i class="fa-brands fa-whatsapp"></i> WhatsApp
                </a>
                {% endif %}
            </div>
        </div>
```

From: detail.html

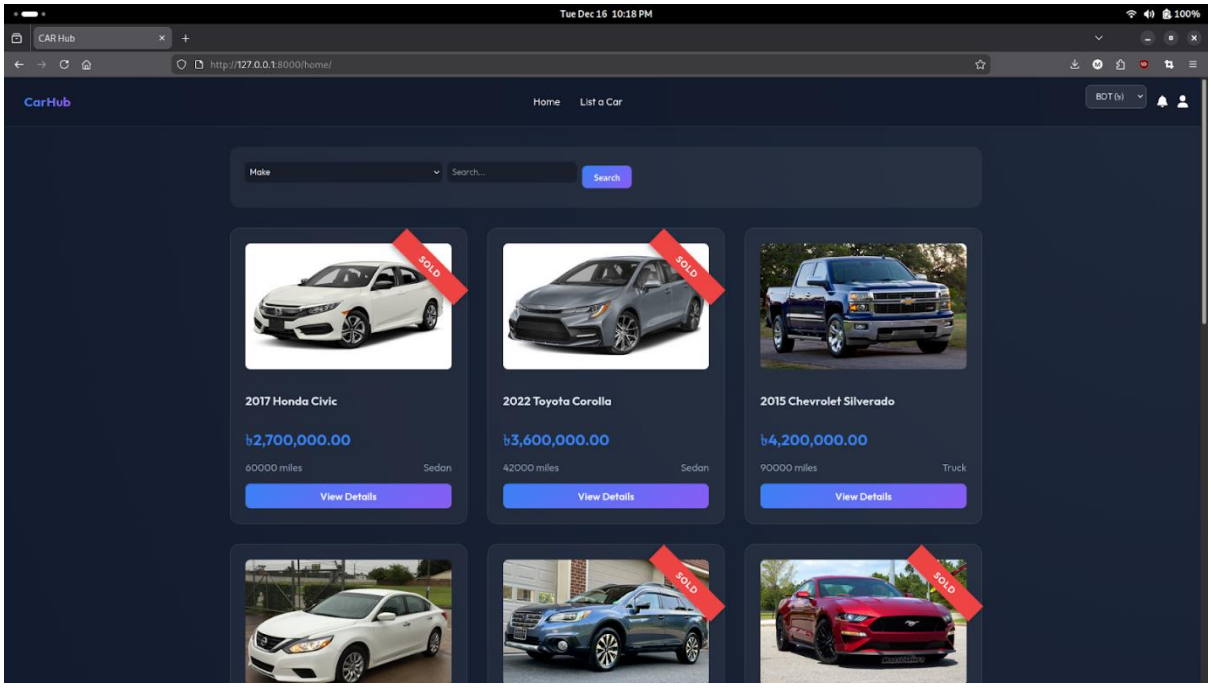


Fig:.6.2.3: View Listings

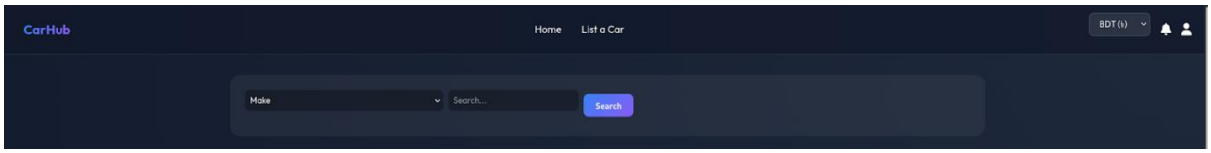


Fig.6.3.1: Search, Filter (code in views.py234-303, 246-257, 232)

```python
@login_required
def follow_car(request, car_id):
    car = get_object_or_404(Car, id=car_id)
    if request.user in car.followers.all():
        car.followers.remove(request.user)
        messages.success(request, f"You have unfollowed this {car.make} {car.model}.")
    else:
        car.followers.add(request.user)
        messages.success(request, f"You are now following this {car.make} {car.model}. You will be notified of updates.")
    return redirect('car_detail', car_id=car.id)
```

From: views.py

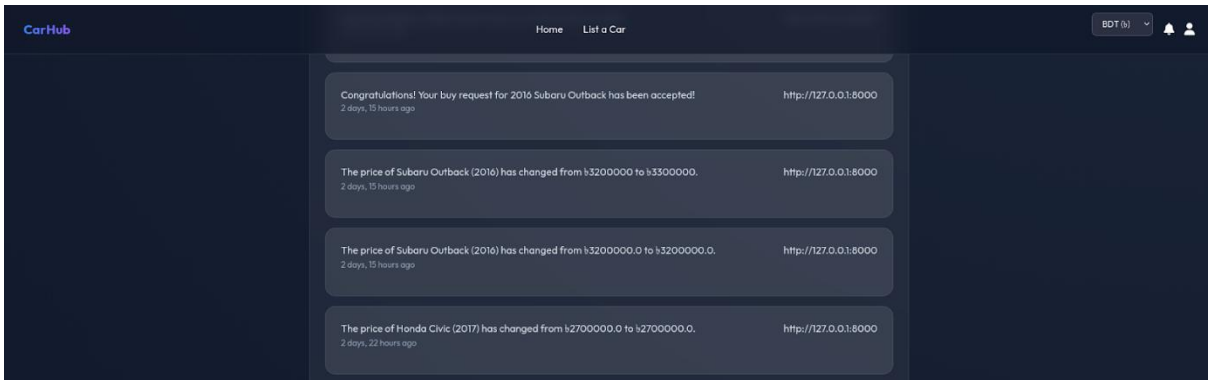
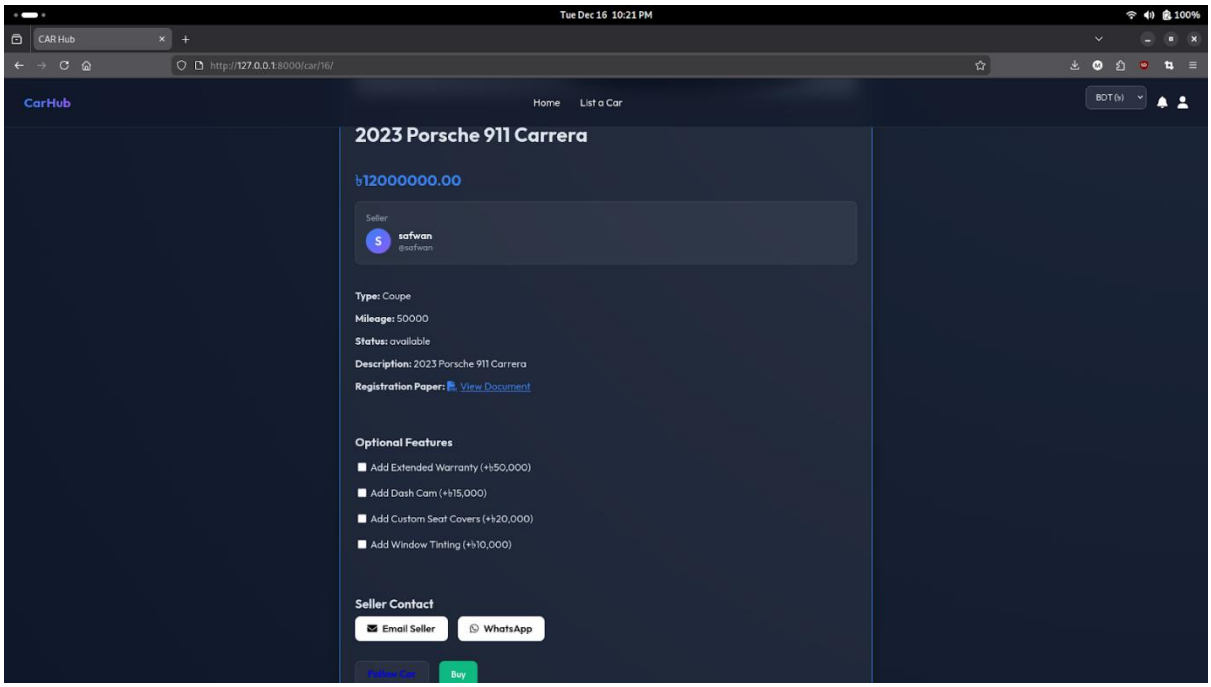


Fig: 6.4.1: Subscribe to Alerts

```
        <div style="width:100%; margin-bottom:1rem;">
            <h3 style="margin-bottom: 0.5rem; font-size: 1.2rem;">Seller Contact</h3>
            <div style="display:flex; gap:1rem;">
                {% if car.contact_email %}
                <a href="mailto:{{ car.contact_email }}" class="btn"
                    style="background:white; color:black; display:flex; align-items:center; gap:0.5rem; font-weight:600;">
                    <i class="fa-solid fa-envelope"></i> Email Seller
                </a>
                {% endif %}
                {% if car.contact_whatsapp %}
                <a href="https://wa.me/{{ car.contact_whatsapp }}" target="_blank" class="btn"
                    style="background:white; color:black; display:flex; align-items:center; gap:0.5rem; font-weight:600;">
                    <i class="fa-brands fa-whatsapp"></i> WhatsApp
                </a>
                {% endif %}
            </div>
        </div>
```

From: detail.html




```python
class Notification(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    message = models.TextField()
    is_read = models.BooleanField(default=False)
    created_at = models.DateTimeField(auto_now_add=True)
```

```python
    # Notify Owner
    Notification.objects.create(
        user=car.owner,
        message=f"New Buy Request: {request.user.username} wants to buy your {car.year} {car.make} {car.model}."
    )

    messages.success(request, "Buy request sent!  Proceed to payment.")

    # Redirect to profile page
    return redirect('profile')
...
def notifications(request):
    if not request.user.is_authenticated:
        return redirect('login')
    notifs = Notification.objects.filter(user=request.user).order_by('-created_at')
    unread_count = notifs.filter(is_read=False).count()

    # Get admin contact info
    admin_user = User.objects.filter(is_superuser=True).first()
    admin_email = admin_user.email if admin_user else None
    admin_whatsapp = admin_user.profile.whatsapp_number if admin_user and hasattr(admin_user, 'profile') else None

    return render(request, 'cars/notifications.html', {
        'notifications': notifs,
        'unread_count': unread_count,
        'admin_email': admin_email,
        'admin_whatsapp': admin_whatsapp
    })
```

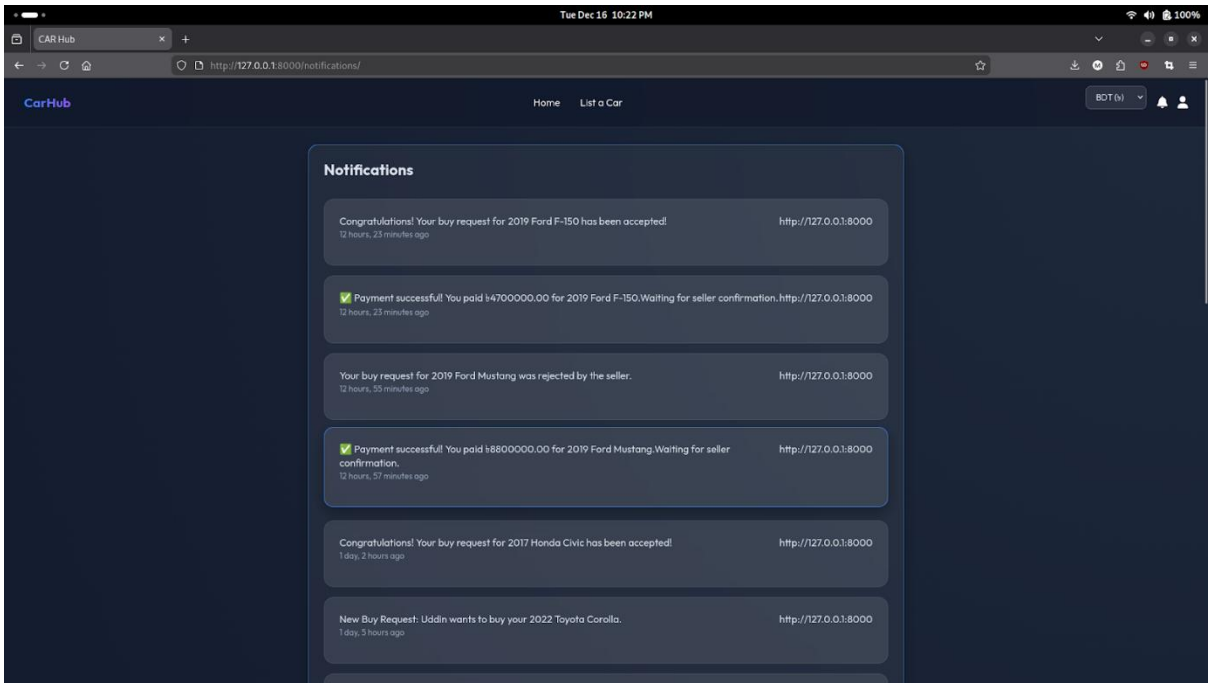


Fig.6.5.1: Contact Seller, Notifications

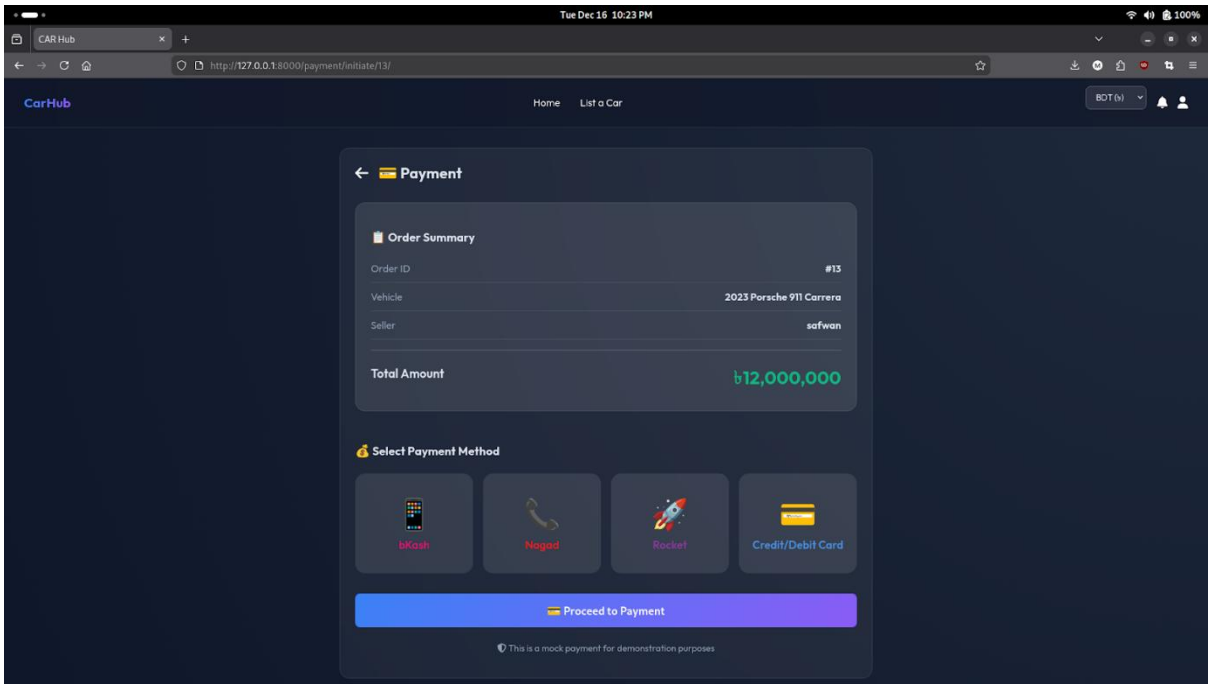


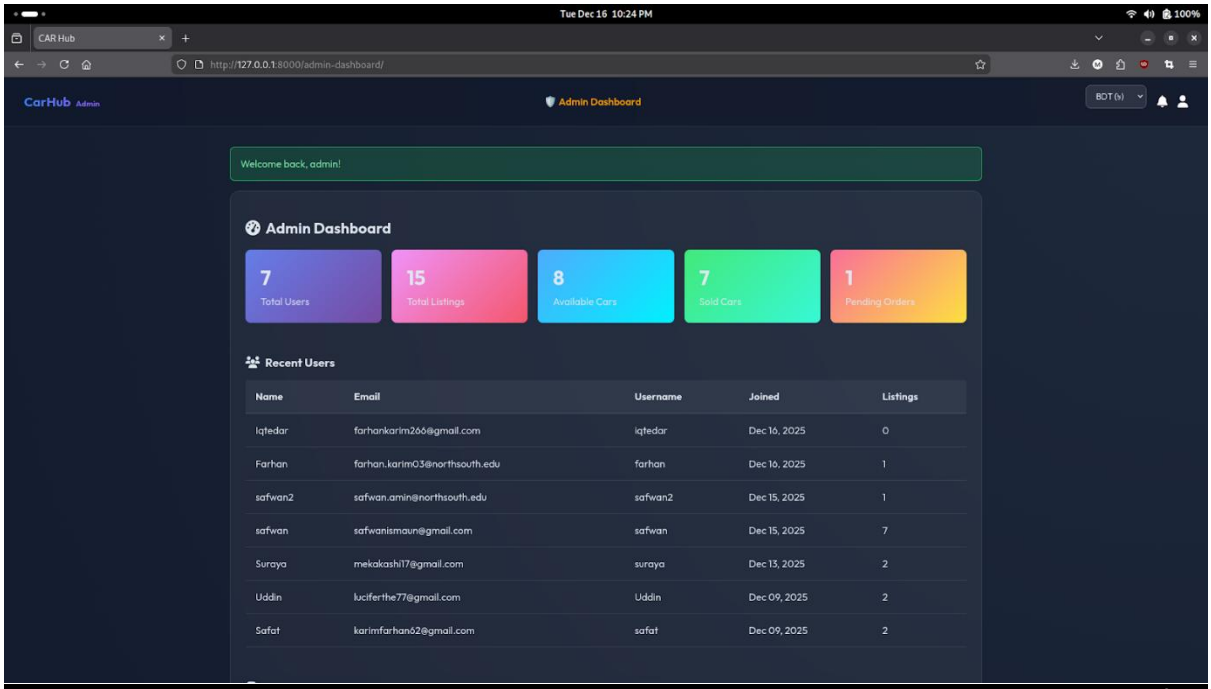
Fig.6.6.1: Payment Procesing

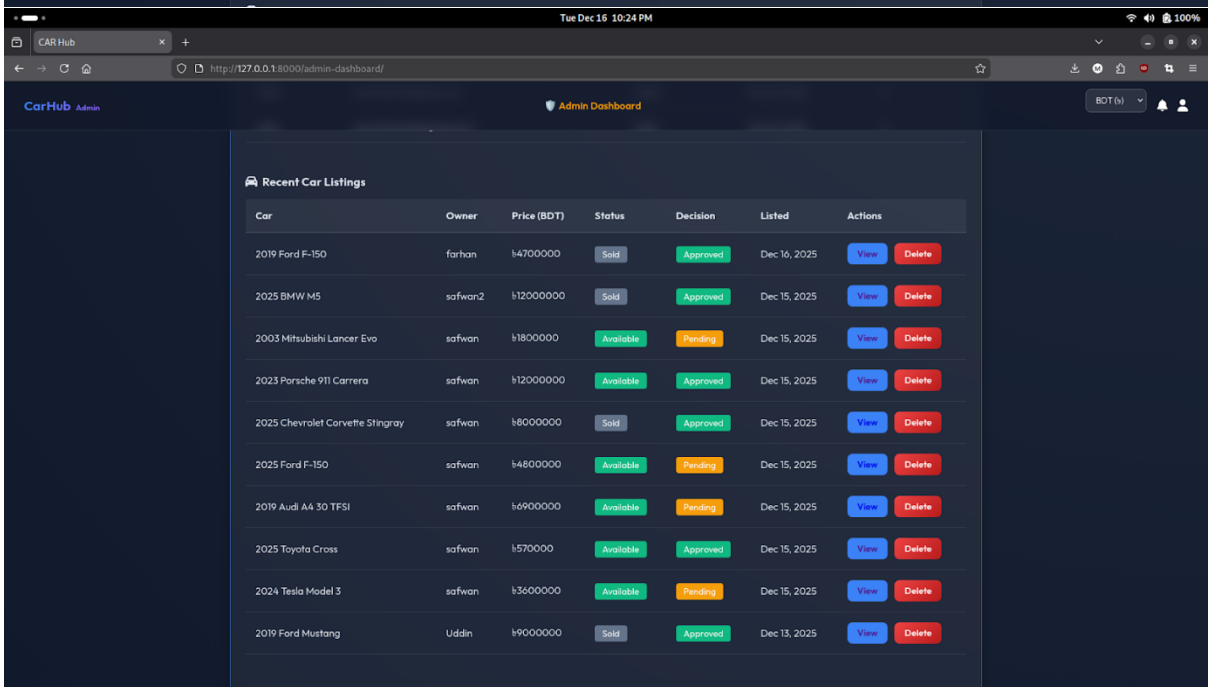
```python
@login_required
def admin_dashboard(request):
    # Only allow superusers to access
    if not request.user.is_superuser:
        messages.error(request, "You do not have permission to access the admin dashboard.")
        return redirect('home')

    # Get statistics
    from django.db.models import Count
    total_users = User.objects.filter(is_superuser=False).count()
    total_cars = Car.objects.count()
    available_cars = Car.objects.filter(status='available').count()
    sold_cars = Car.objects.filter(status='sold').count()
    pending_orders = Order.objects.filter(status='pending').count()

    # Get recent users with their car count (exclude superusers)
    recent_users = User.objects.filter(is_superuser=False).annotate(car_count=Count('car')).order_by('-date_joined')[:10]
    recent_users_data = [{'user': user, 'car_count': user.car_count} for user in recent_users]

    # Get recent cars
    recent_cars = Car.objects.all().order_by('-created_at')[:10]

    # Get recent orders
    recent_orders = Order.objects.all().order_by('-created_at')[:10]

    return render(request, 'cars/admin_dashboard.html', {
        'total_users': total_users,
        'total_cars': total_cars,
        'available_cars': available_cars,
        'sold_cars': sold_cars,
        'pending_orders': pending_orders,
        'recent_users': recent_users_data,
        'recent_cars': recent_cars,
        'recent_orders': recent_orders,
    })
```

```python
@login_required
def approve_car(request, car_id):
    proxy = CarAccessProxy(request.user)
    success, msg = proxy.approve_car(car_id)

    if success:
        messages.success(request, msg)
    else:
        messages.error(request, msg)

    return redirect('car_detail', car_id=car_id)

@login_required
def reject_car(request, car_id):
    proxy = CarAccessProxy(request.user)
    reason = request.POST.get('reason', '') if request.method == 'POST' else ''
    success, msg = proxy.reject_car(car_id, reason)

    if success:
        messages.success(request, msg)
    else:
        messages.error(request, msg)

    return redirect('admin_dashboard')

```

```python
    description = models.TextField(blank=True)
    status = models.CharField(max_length=20, default='available') # available, sold
    approval_status = models.CharField(max_length=20, default='pending') # pending, approved, rejected
    owner = models.ForeignKey(User, on_delete=models.CASCADE)
```

```
770
771     # Get statistics
772     from django.db.models import Count
773     total_users = User.objects.filter(is_superuser=False).count()
774     total_cars = Car.objects.count()
775     available_cars = Car.objects.filter(status='available').count()
776     sold_cars = Car.objects.filter(status='sold').count()
777     pending_orders = Order.objects.filter(status='pending').count()
778
```





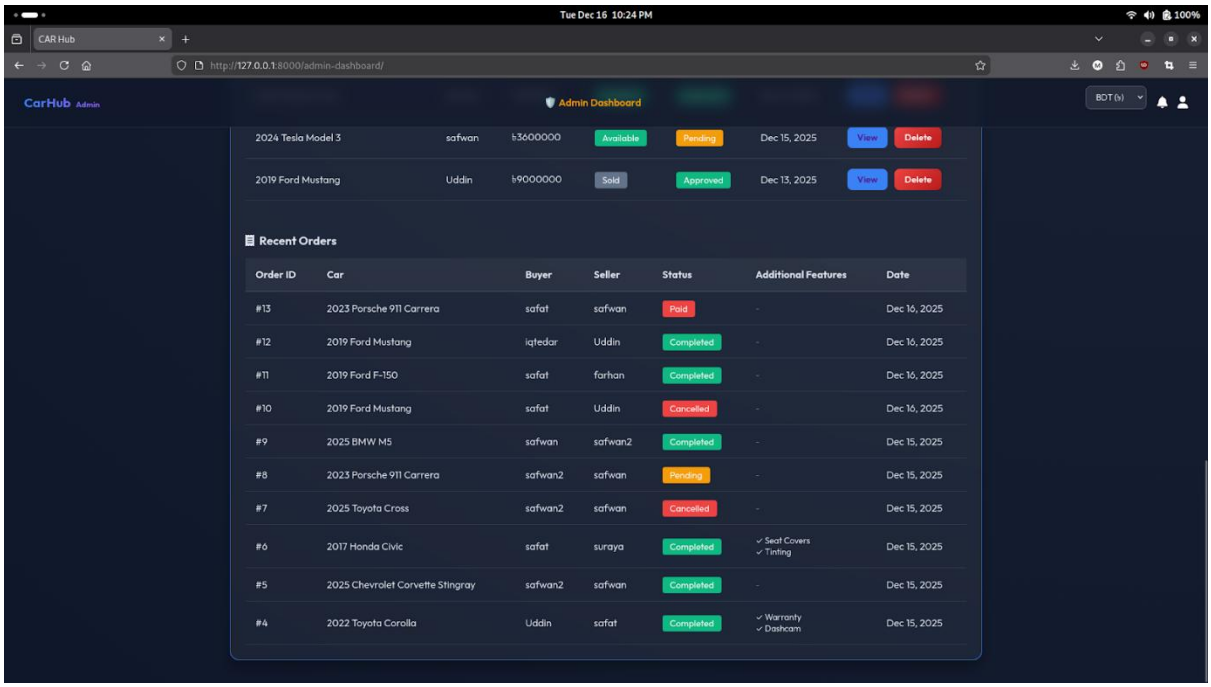


Fig..6.7.1: Admin Dashboard

ii) Non-Functional Requirements

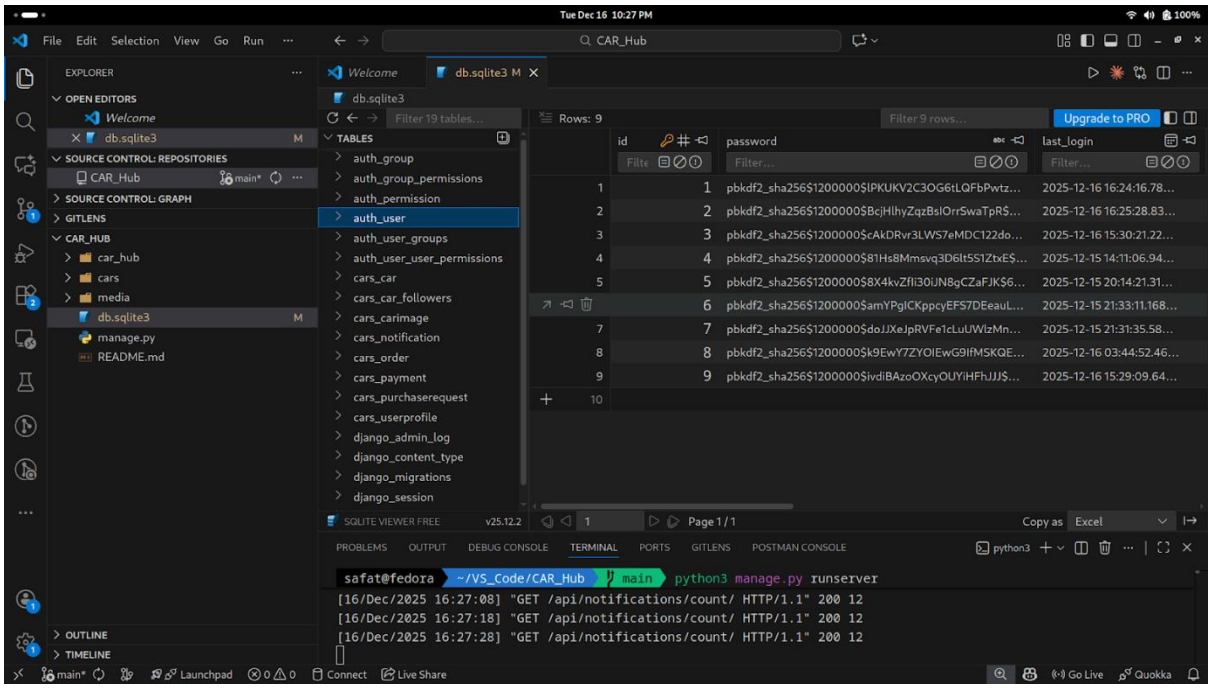


Fig..6.8.1: Security

iii) Design Patterns

a) Factory Pattern

```
from abc import ABC, abstractmethod

# Car -> Product
from cars.models import Car

# Concrete Products inside models.py
#  CAR_TYPES = (
#      ('sedan', 'Sedan'),
#      ('suv', 'SUV'),
#      ('truck', 'Truck'),
#      ('coupe', 'Coupe'),
#  )

# Creator
class CarFactory(ABC):
   @abstractmethod
   def create_car(self, make, model, year, price, mileage,
owner):
       pass

# Concrete Creators
class SedanFactory(CarFactory):
   def create_car(self, make, model, year, price, mileage,
owner):
       return Car.objects.create(
           make=make, model=model, year=year, price=price,
           mileage=mileage, car_type='sedan', owner=owner
       )

class SUVFactory(CarFactory):
   def create_car(self, make, model, year, price, mileage,
owner):
       return Car.objects.create(
           make=make, model=model, year=year, price=price,
           mileage=mileage, car_type='suv', owner=owner
       )

class TruckFactory(CarFactory):
   def create_car(self, make, model, year, price, mileage,
owner):
```

```
        return Car.objects.create(
            make=make, model=model, year=year, price=price,
            mileage=mileage, car_type='truck', owner=owner
        )

class CoupeFactory(CarFactory):
    def create_car(self, make, model, year, price, mileage,
owner):
        return Car.objects.create(
            make=make, model=model, year=year, price=price,
            mileage=mileage, car_type='coupe', owner=owner
        )
```

b) Singleton Pattern

```
class DatabaseConfigManager:

    # Private static instance variable
    _instance = None

    def __new__(cls):
        """
        Private constructor - controls object creation.
        Only creates a new instance if one doesn't exist.
        """
        if cls._instance is None:
            cls._instance = super(DatabaseConfigManager,
cls).__new__(cls)
            # Initialize configuration only once
            cls._instance._initialize_config()
        return cls._instance

    def _initialize_config(self):
        """
        Private method to initialize configuration.
        Called only once when the instance is first created.
        """
        self.db_connection_string =
"mysql://root:password@localhost:3306/car_hub_db"
        self.max_connections = 100
        self.pool_size = 10
        self.timeout = 30

    @classmethod
    def getInstance(cls):
        """
        Public static method to get the singleton instance.
```

```
        This is the standard way to access the singleton.
        """
        if cls._instance is None:
            cls._instance = cls()
        return cls._instance

    def get_connection_string(self):
        """Get the database connection string"""
        return self.db_connection_string

    def get_config(self):
        """
        Get the complete database configuration.
        """
        return {
            "connection": self.db_connection_string,
            "max_connections": self.max_connections,
            "pool_size": self.pool_size,
            "timeout": self.timeout
        }

    def update_max_connections(self, new_max):
        """Update maximum connections (affects all references)"""
        self.max_connections = new_max
```

c) Observer Pattern

```
from abc import ABC, abstractmethod
from cars.models import Notification

# Observer
class Observer(ABC):
    @abstractmethod
    def update(self, message):
        pass

# Concrete Observer
class UserObserver(Observer):
    def __init__(self, user):
        self.user = user

    def update(self, message):
        # Create a notification in DB
```

```
        Notification.objects.create(user=self.user,
message=message)

# Subject
class Subject(ABC):
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        if observer not in self._observers:
            self._observers.append(observer)

    def detach(self, observer):
        try:
            self._observers.remove(observer)
        except ValueError:
            pass

    def notify(self, message):
        for observer in self._observers:
            observer.update(message)

class CarPriceSubject(Subject):
    def __init__(self, car):
        super().__init__()
        self.car = car

    def change_price(self, new_price):
        # Capture old price BEFORE saving
        old_price = self.car.price

        # Update and save the new price
        self.car.price = new_price
        self.car.save()

        # Notify all observers through the proper Observer
pattern mechanism
        message = f"The price of {self.car.make} {self.car.model}
({self.car.year}) has changed from ৳{old_price:.0f} to
৳{new_price:.0f}."
        self.notify(message)
```

d) Strategy Pattern

```
from abc import ABC, abstractmethod
from cars.models import Car

class SearchStrategy(ABC):
    @abstractmethod
    def search(self, query):
        pass

class PriceSearchStrategy(SearchStrategy):
    def search(self, price_range):
        # price_range is expected to be a tuple or list:
[min_price, max_price]
        min_price, max_price = price_range
        return Car.objects.filter(price__gte=min_price,
price__lte=max_price)

class BrandSearchStrategy(SearchStrategy):
    def search(self, brand_name):
        return Car.objects.filter(make__icontains=brand_name)

class ModelSearchStrategy(SearchStrategy):
    def search(self, model_name):
        return Car.objects.filter(model__icontains=model_name)

class MileageSearchStrategy(SearchStrategy):
    def search(self, mileage_range):
        # mileage_range is expected to be a tuple or list:
[min_mileage, max_mileage]
        min_mileage, max_mileage = mileage_range
        return Car.objects.filter(mileage__gte=min_mileage,
mileage__lte=max_mileage)

class TypeSearchStrategy(SearchStrategy):
    def search(self, car_type):
        return Car.objects.filter(car_type__iexact=car_type)

class YearSearchStrategy(SearchStrategy):
    def search(self, year_range):
        # year_range is expected to be a tuple or list:
[min_year, max_year]
        min_year, max_year = year_range
        return Car.objects.filter(year__gte=min_year,
year__lte=max_year)

class CarSearchContext:
```

```
    def __init__(self, strategy: SearchStrategy):
        self.strategy = strategy

    def set_strategy(self, strategy: SearchStrategy):
        self.strategy = strategy

    def execute_search(self, query):
        return self.strategy.search(query)
```

e) Decorator Pattern

```
from abc import ABC, abstractmethod

# Component Interface
class CarComponent(ABC):
    @abstractmethod
    def get_price(self):
        pass

    @abstractmethod
    def get_description(self):
        pass

# Concrete Component
class BasicCar(CarComponent):
    def __init__(self, car_model):
        self.car = car_model

    def get_price(self):
        return float(self.car.price)

    def get_description(self):
        return f"{self.car.year} {self.car.make}
{self.car.model}"

# Decorator
class CarDecorator(CarComponent):
    def __init__(self, car_component: CarComponent):
        self.car_component = car_component

    @abstractmethod
    def get_price(self):
        pass

    @abstractmethod
    def get_description(self):
        pass
```

```
# Concrete Decorators
class WarrantyDecorator(CarDecorator):
    def get_price(self):
        return self.car_component.get_price() + 50000.00

    def get_description(self):
        return self.car_component.get_description() + " +
Extended Warranty"

class DashCamDecorator(CarDecorator):
    def get_price(self):
        return self.car_component.get_price() + 15000.00

    def get_description(self):
        return self.car_component.get_description() + " + Dash
Cam"

class SeatCoversDecorator(CarDecorator):
    def get_price(self):
        return self.car_component.get_price() + 20000.00

    def get_description(self):
        return self.car_component.get_description() + " + Custom
Seat Covers"

class WindowTintingDecorator(CarDecorator):
    def get_price(self):
        return self.car_component.get_price() + 10000.00

    def get_description(self):
        return self.car_component.get_description() + " + Window
Tinting"
```

f) Proxy Pattern

```
from abc import ABC, abstractmethod
from cars.models import Car

class CarAccessInterface(ABC):
    @abstractmethod
    def delete_car(self, car_id):
        pass

    @abstractmethod
    def post_car(self, car_data):
```

```
        pass

    @abstractmethod
    def approve_car(self, car_id):
        pass

    @abstractmethod
    def reject_car(self, car_id, reason):
        pass

class RealCarService(CarAccessInterface):
    def delete_car(self, car_id):
        try:
            car = Car.objects.get(id=car_id)
            car.delete()
            return True, "Car deleted successfully."
        except Car.DoesNotExist:
            return False, "Car not found."

    def post_car(self, car_data):
        # Assuming car_data is a dictionary and we use the
factory or direct create
        # For simplicity, just creating here
        return True, "Car posted successfully."

    def approve_car(self, car_id):
        try:
            car = Car.objects.get(id=car_id)
            car.approval_status = 'approved'
            car.save()

            # Notify owner
            from cars.models import Notification
            Notification.objects.create(
                user=car.owner,
                message=f"Your listing '{car.year} {car.make}
{car.model}' has been approved by admin and is now visible to
buyers."
            )
            return True, "Car listing approved successfully."
        except Car.DoesNotExist:
            return False, "Car not found."

    def reject_car(self, car_id, reason=""):
        try:
            car = Car.objects.get(id=car_id)
```

```
            # Store car details before deletion
            car_year = car.year
            car_make = car.make
            car_model = car.model
            car_owner = car.owner

            # Build notification message
            from cars.models import Notification
            message = f"Your car listing '{car_year} {car_make}
{car_model}' has been rejected and removed by admin.\n\nReason:
{reason}"

            Notification.objects.create(
                user=car_owner,
                message=message
            )

            # Delete the car
            car.delete()
            return True, "Car listing rejected and seller
notified."
        except Car.DoesNotExist:
            return False, "Car not found."

class CarAccessProxy(CarAccessInterface):
    def __init__(self, user):
        self.user = user
        self.real_service = RealCarService()

    def delete_car(self, car_id):
        if self.user.is_superuser:
            return self.real_service.delete_car(car_id)
        # Allow owner to delete their own car
        try:
            car = Car.objects.get(id=car_id)
            if self.user == car.owner:
                return self.real_service.delete_car(car_id)
        except Car.DoesNotExist:
            return False, "Car not found."
        return False, "Permission denied: You are not authorized
to delete this car."

    def post_car(self, car_data):
        if not self.user.is_authenticated:
            return False, "Permission denied: You must be logged
in to post a car."
        if self.user.is_superuser:
```

```
            return False, "Permission denied: Admin users cannot
list cars for sale."
        return self.real_service.post_car(car_data)

    def approve_car(self, car_id):
        if not self.user.is_superuser:
            return False, "Permission denied: Only admin can
approve car listings."
        return self.real_service.approve_car(car_id)

    def reject_car(self, car_id, reason=""):
        if not self.user.is_superuser:
            return False, "Permission denied: Only admin can
reject car listings."
        return self.real_service.reject_car(car_id, reason)
```

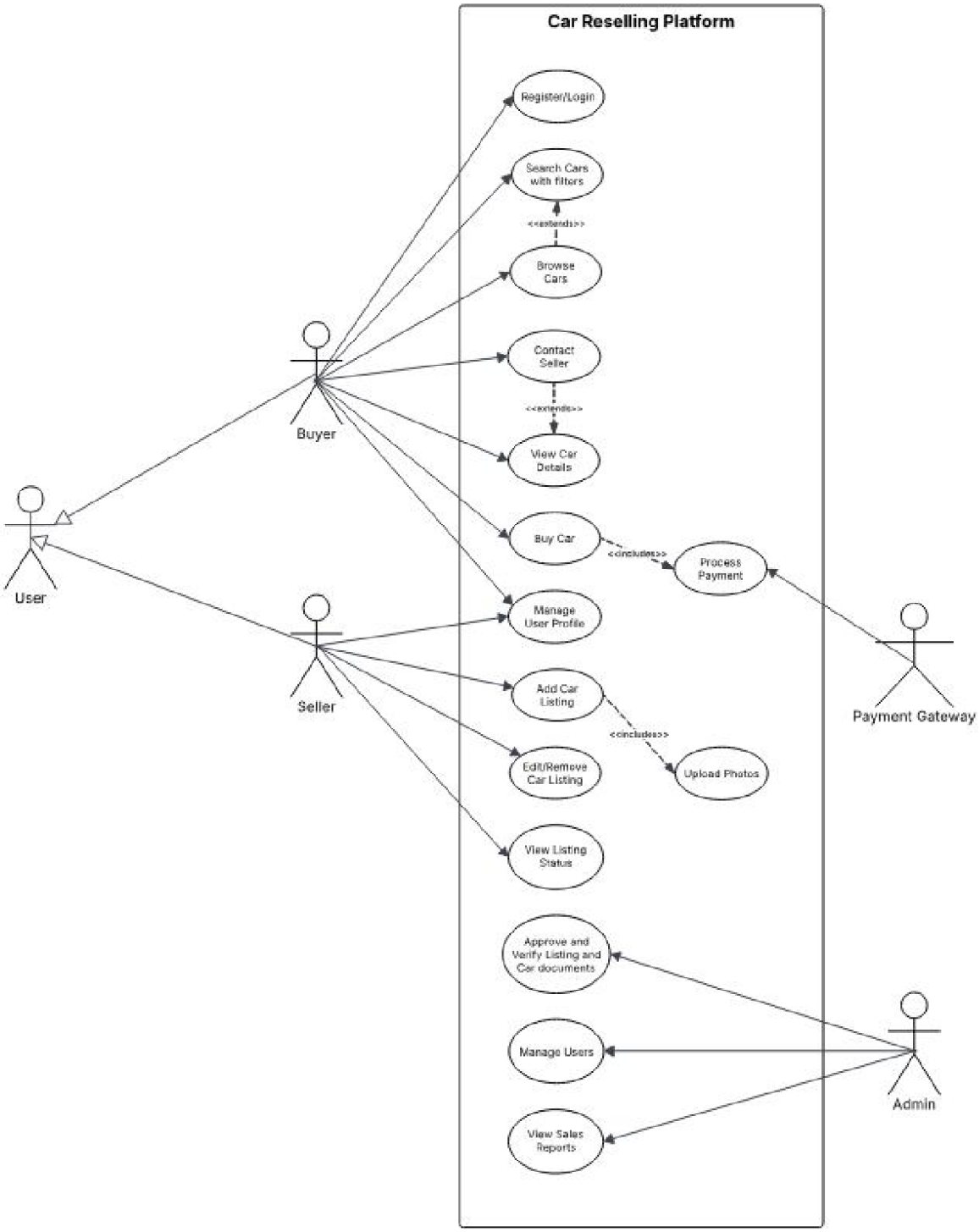
g) Adapter Pattern

```
# target interface
class CurrencyConverter:

    def convert_to_bdt(self, amount, currency): #to be
implemented by adapter
        """Convert any currency to BDT"""
        raise NotImplementedError #make sure derived class
implements this method

    def convert_from_bdt(self, amount_bdt, target_currency): #to
be implemented by adapter
        """Convert BDT to any currency"""
        raise NotImplementedError #make sure derived class
implements this method

    def get_currency_symbol(self, currency): #to be implemented
by adapter
        """Get the symbol for a currency"""
        raise NotImplementedError #make sure derived class
implements this method


# Adaptee - Third-party service with incompatible interface
class ThirdPartyCurrencyAPI:


    # Exchange rates: 1 unit of currency = X BDT
    EXCHANGE_RATES = {
        'BDT': 1.0,
```

```
        'USD': 120.0,   # 1 USD = 120 BDT
        'GBP': 150.0,   # 1 GBP = 150 BDT
        'EUR': 130.0,   # 1 EUR = 130 BDT
        'INR': 1.45     # 1 INR = 1.45 BDT
    }

    CURRENCY_SYMBOLS = {
        'BDT': '৳',
        'USD': '$',
        'GBP': '£',
        'EUR': '€',
        'INR': '₹'
    }

    def get_exchange_rate(self, currency_code):
        """Third-party API method to get exchange rate"""
        return self.EXCHANGE_RATES.get(currency_code, 1.0)

    def get_symbol_for_currency(self, currency_code):
        """Third-party API method to get currency symbol"""
        return self.CURRENCY_SYMBOLS.get(currency_code, '৳')

    def multiply_by_rate(self, amount, rate):
        """Third-party API helper method"""
        return float(amount) * rate

    def divide_by_rate(self, amount, rate):
        """Third-party API helper method"""
        return float(amount) / rate


# Adapter - Makes the adaptee compatible with the target
interface
class CurrencyAdapter(CurrencyConverter):

    def __init__(self):
        # Composition: Adapter contains an instance of the
adaptee
        self._api = ThirdPartyCurrencyAPI()

    def convert_to_bdt(self, amount, currency):
        rate = self._api.get_exchange_rate(currency)
        return self._api.multiply_by_rate(amount, rate)

    def convert_from_bdt(self, amount_bdt, target_currency):
        rate = self._api.get_exchange_rate(target_currency)
        return self._api.divide_by_rate(amount_bdt, rate)
```

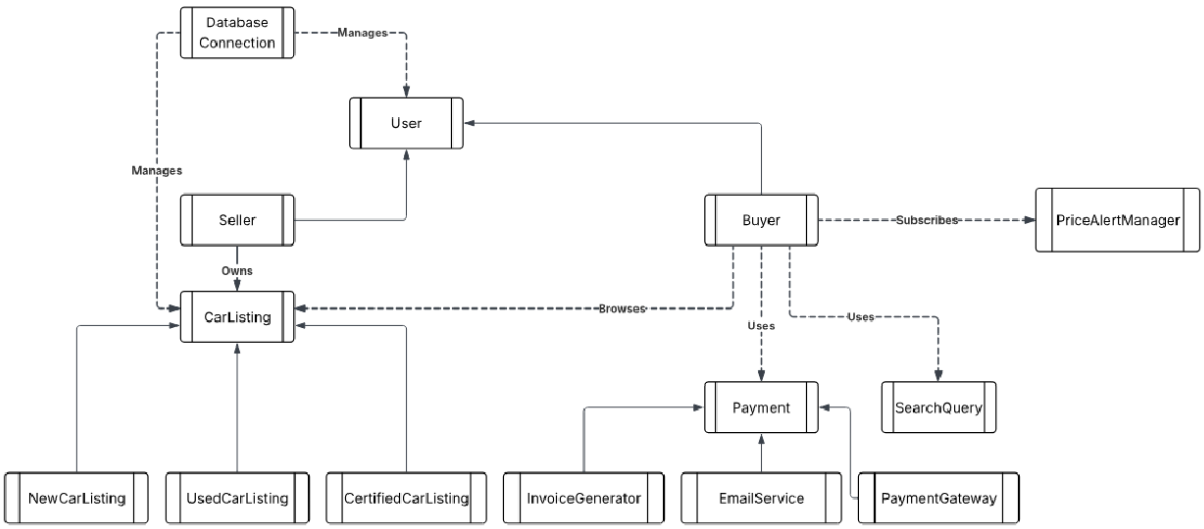
```
    def get_currency_symbol(self, currency):
        return self._api.get_symbol_for_currency(currency)

    # Convenience class methods for backward compatibility
    @classmethod
    def convert_to_bdt_static(cls, amount, from_currency):
        """Static helper method for quick conversions"""
        adapter = cls()
        return adapter.convert_to_bdt(amount, from_currency)

    @classmethod
    def get_symbol(cls, currency):
        """Static helper method to get currency symbol"""
        adapter = cls()
        return adapter.get_currency_symbol(currency)
```

7.USE CASE DIAGRAM:

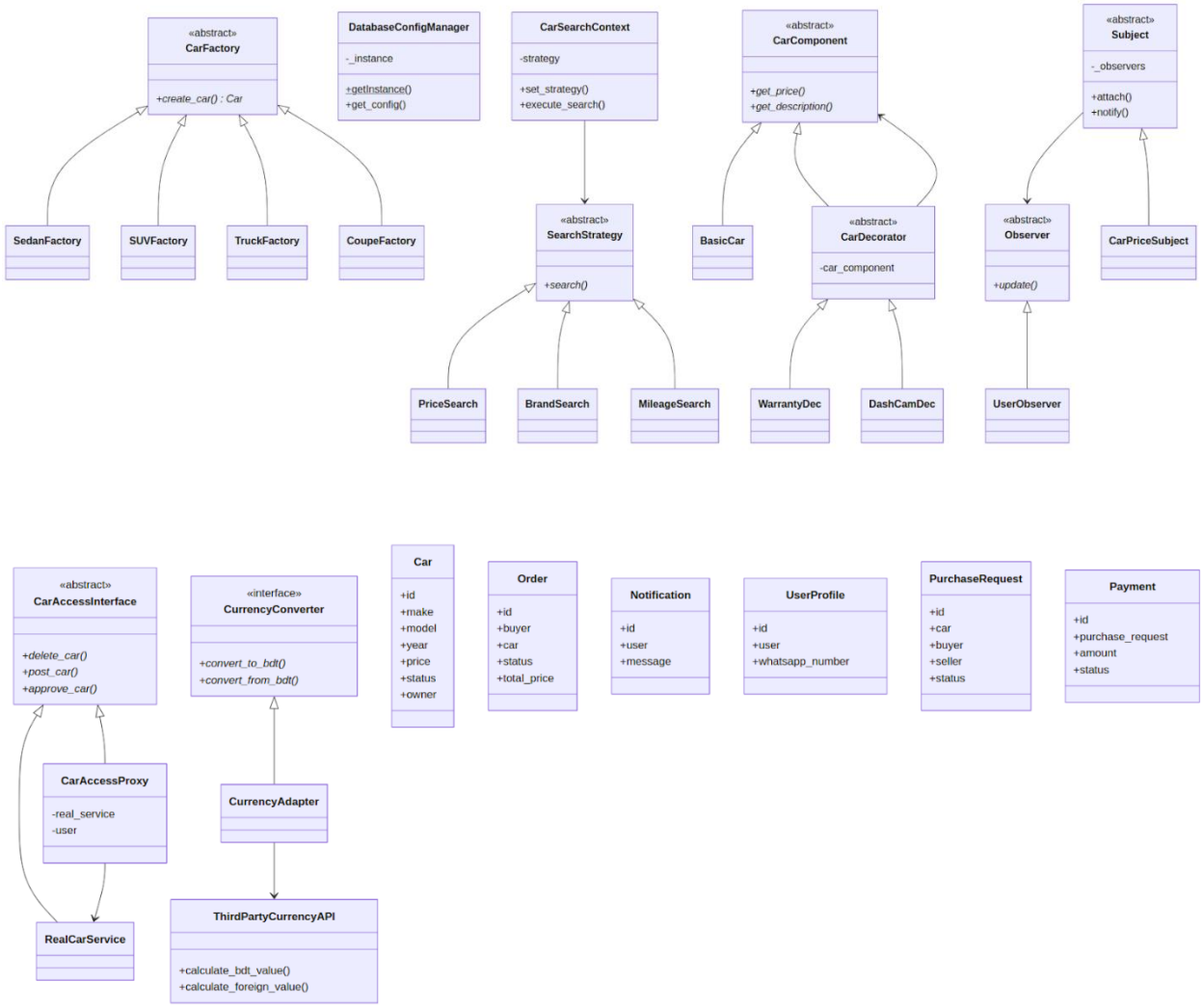


8. CLASS DIAGRAMS

a) Proposed class diagram:

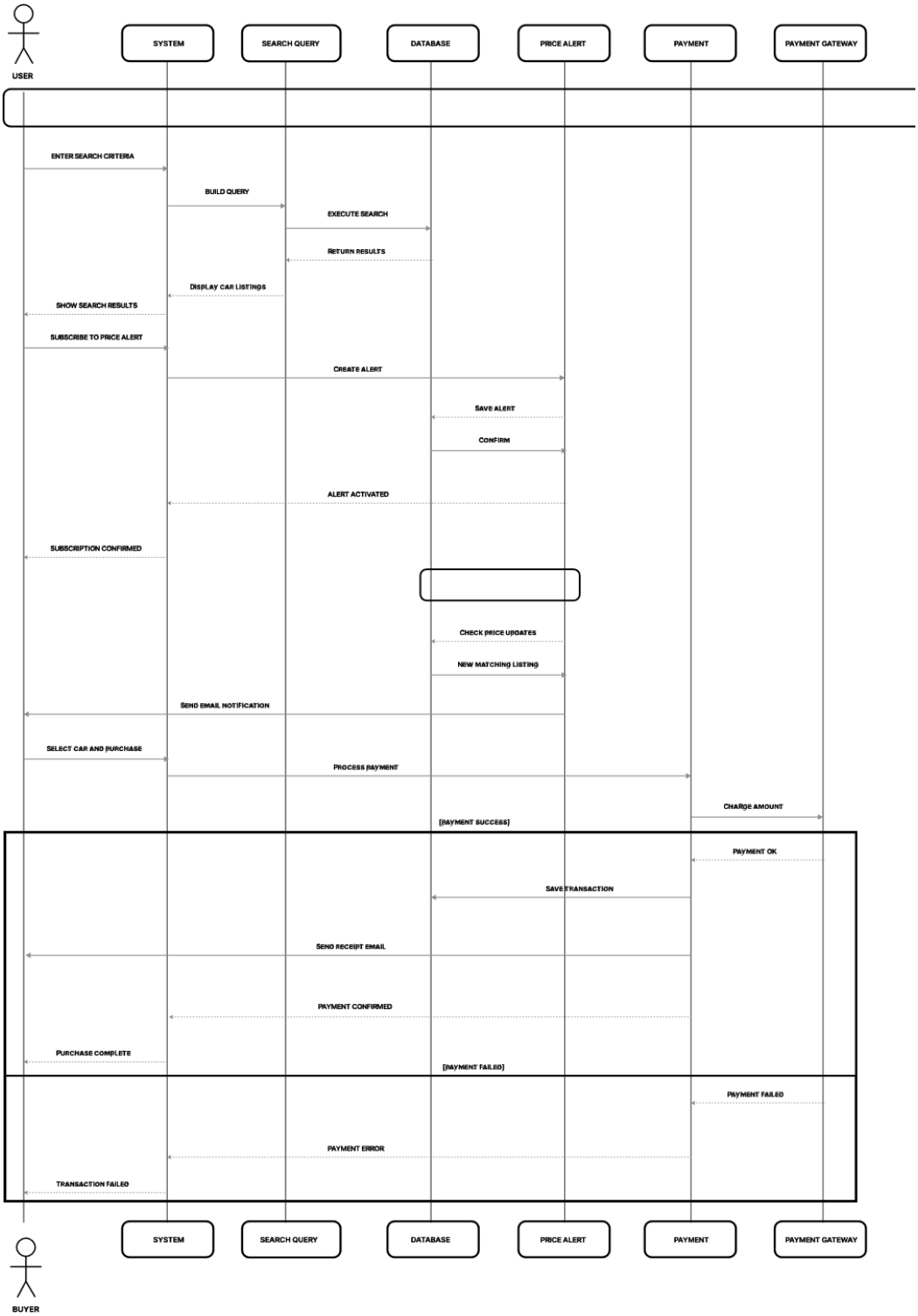


b) Final Class Diagram



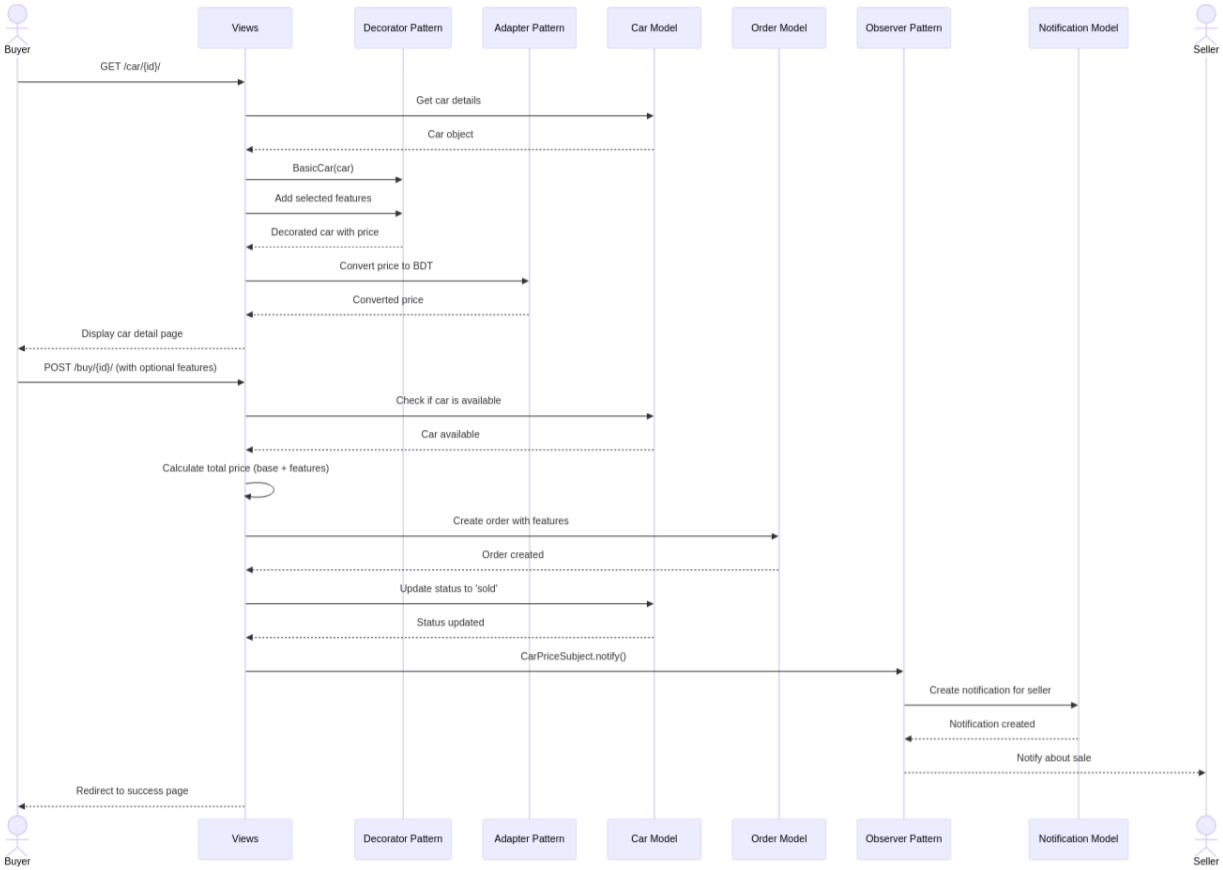
9. SEQUENCE DIAGRAM:

a) Proposed Sequence Diagram

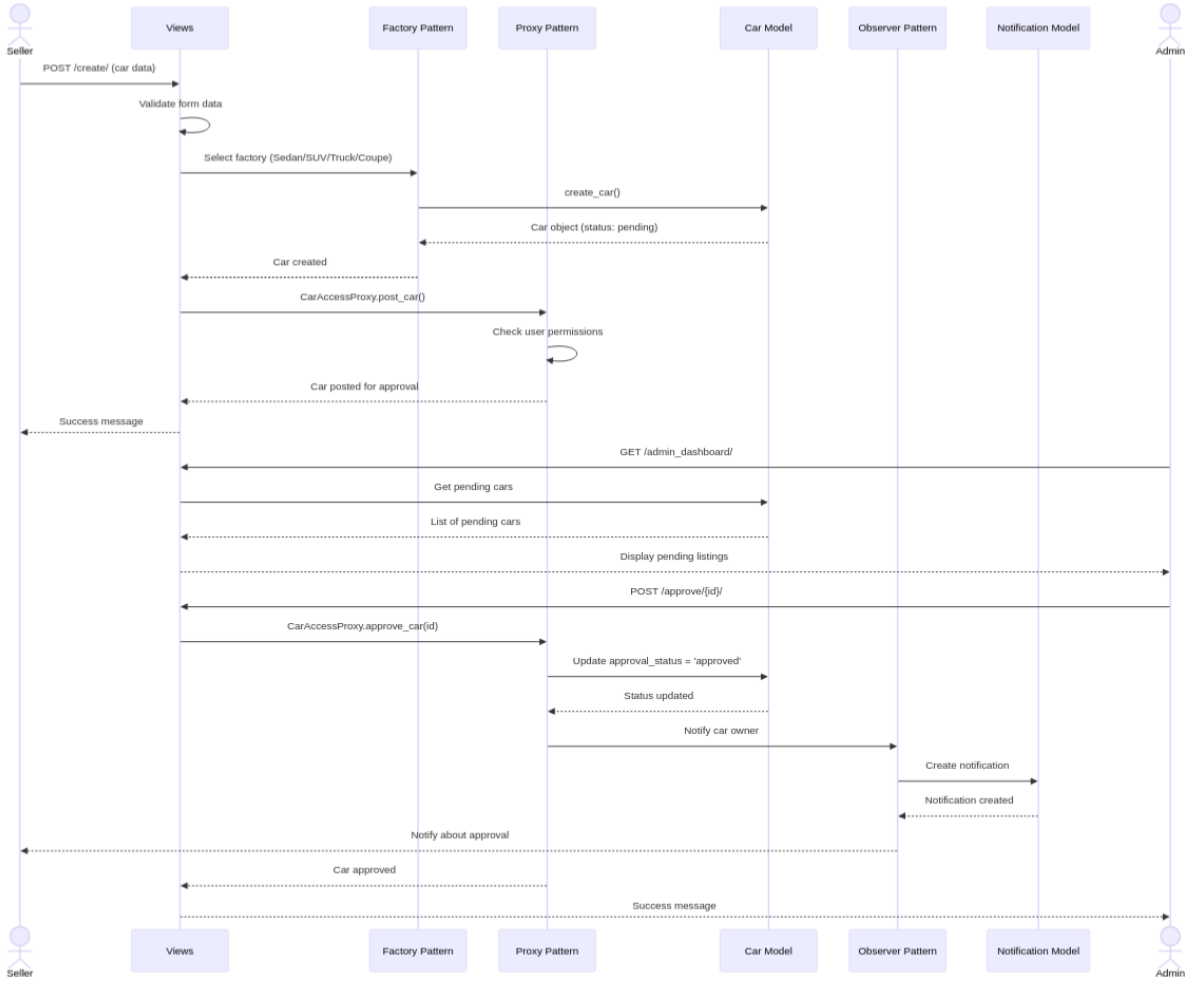


b) Final Sequence Diagram

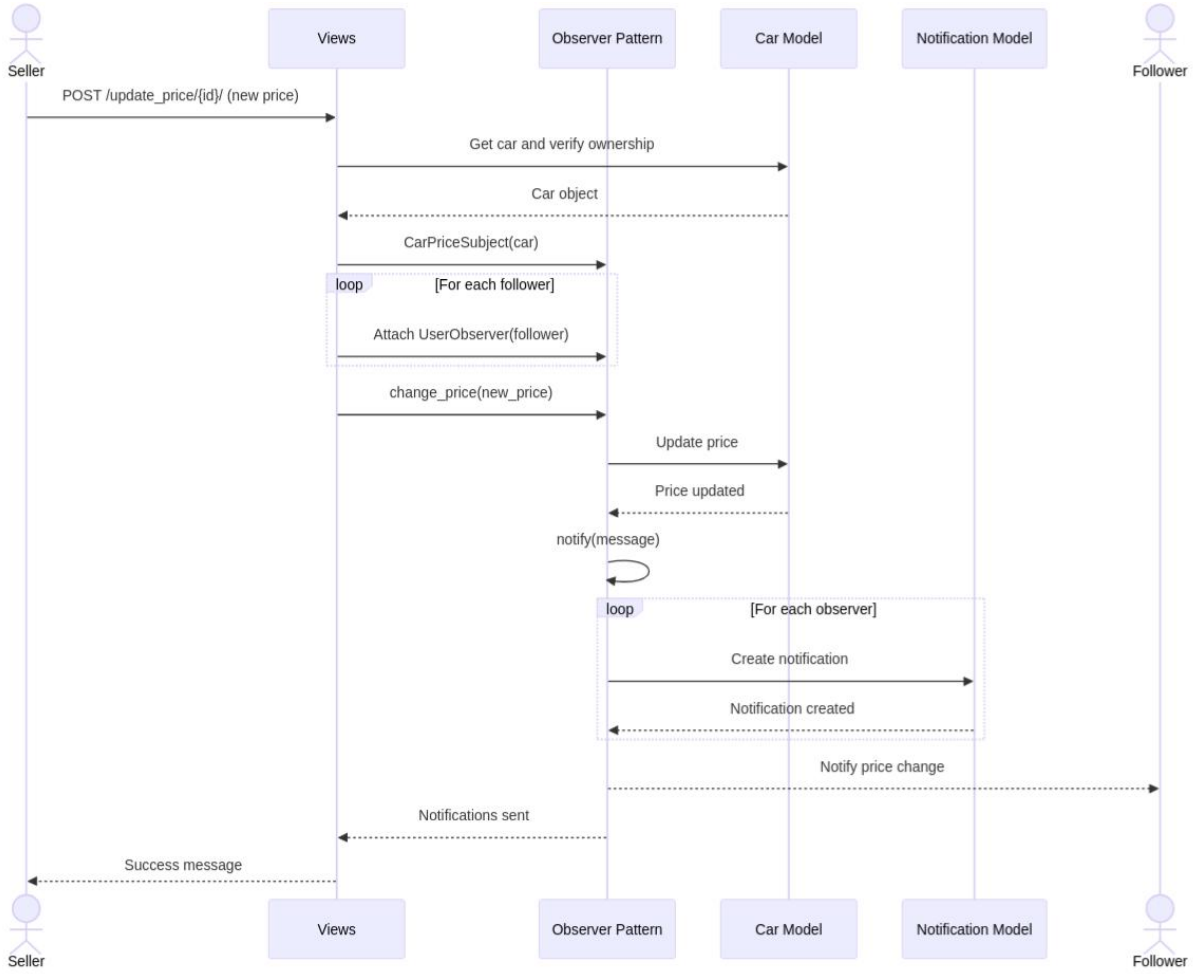
i) Car Purchase Flow:



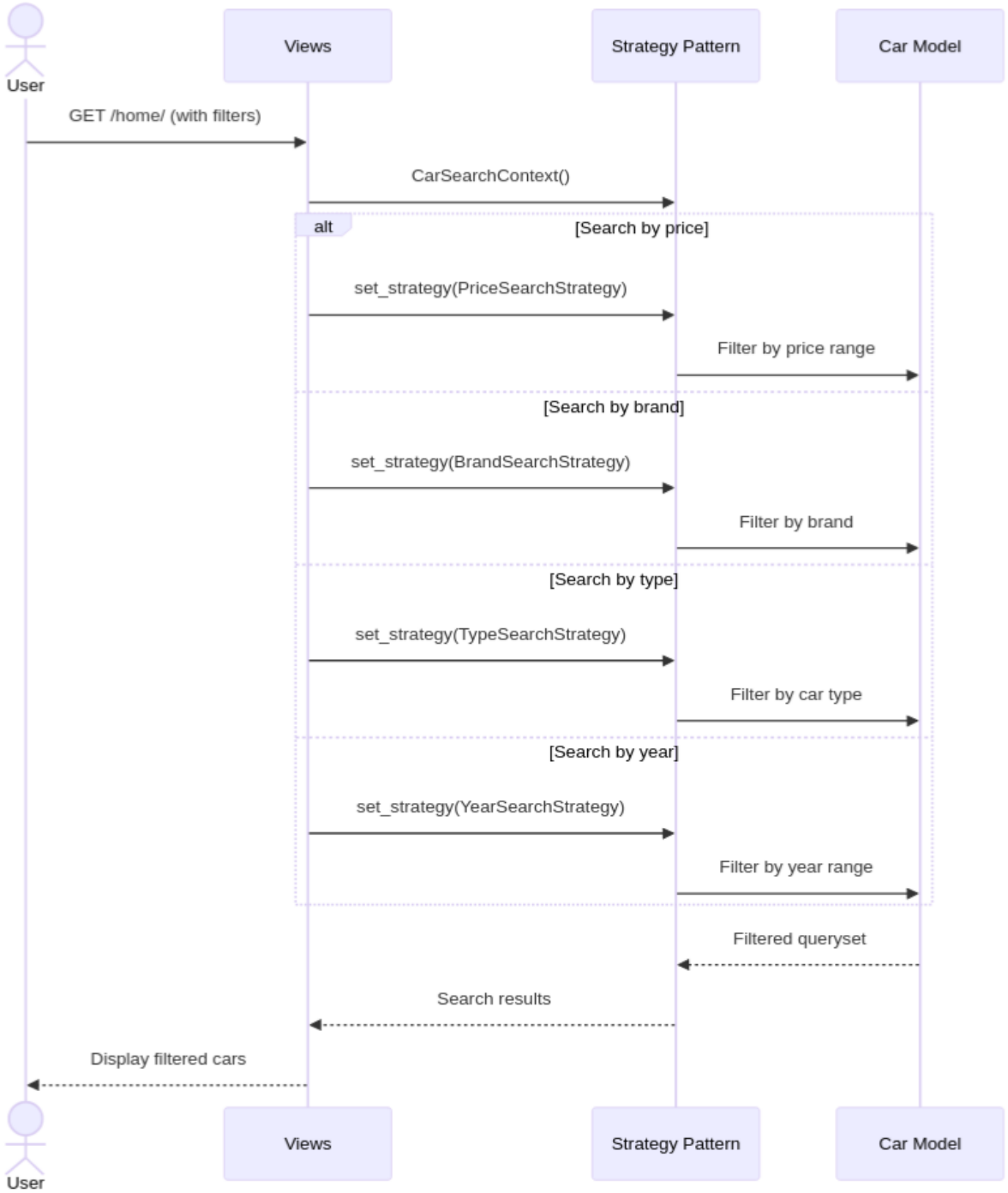
ii) **Car Listing Creation Flow:**



iii) **Price Update and Notification Flow:**

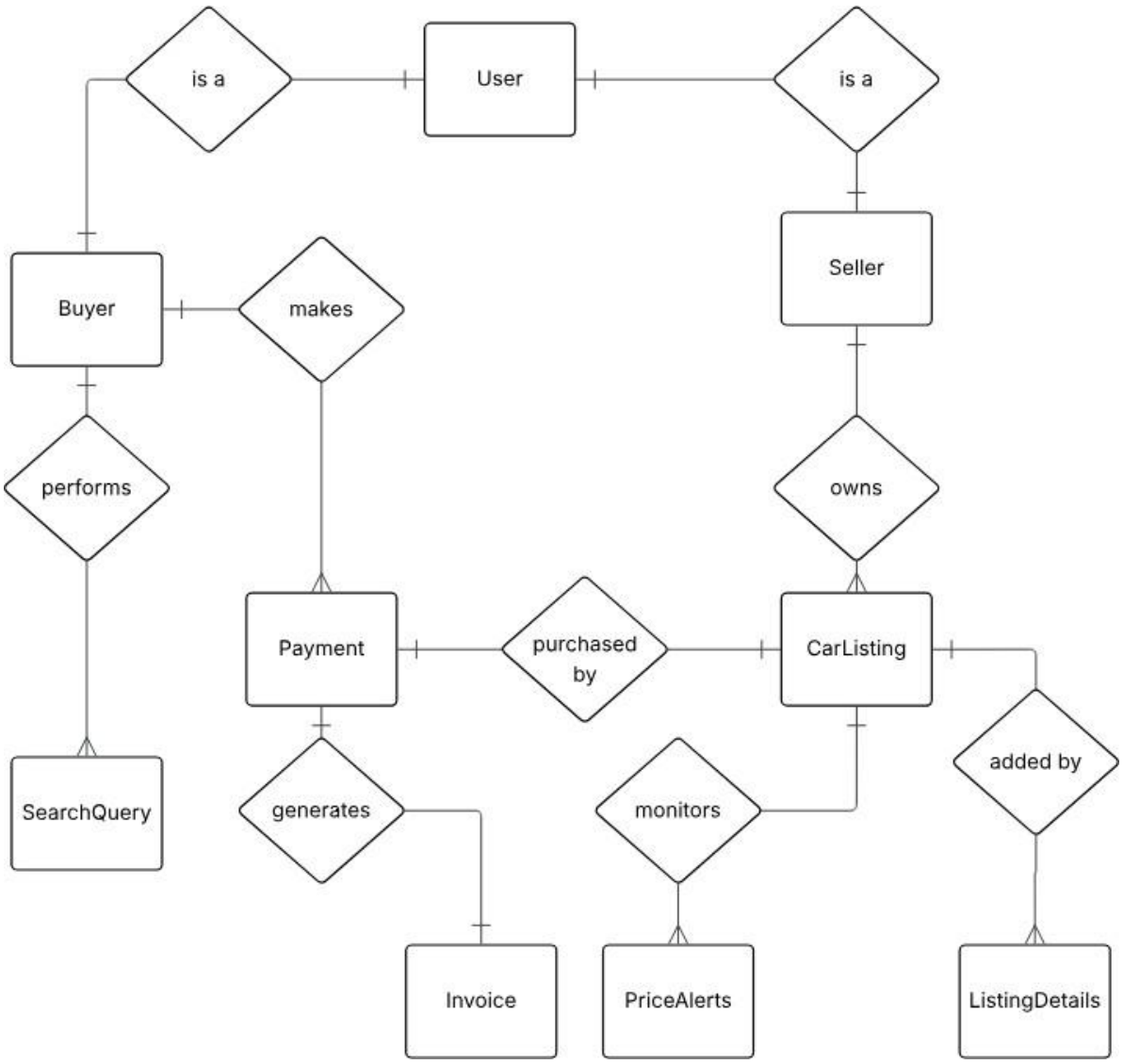


iv) Search and Filter Flow:

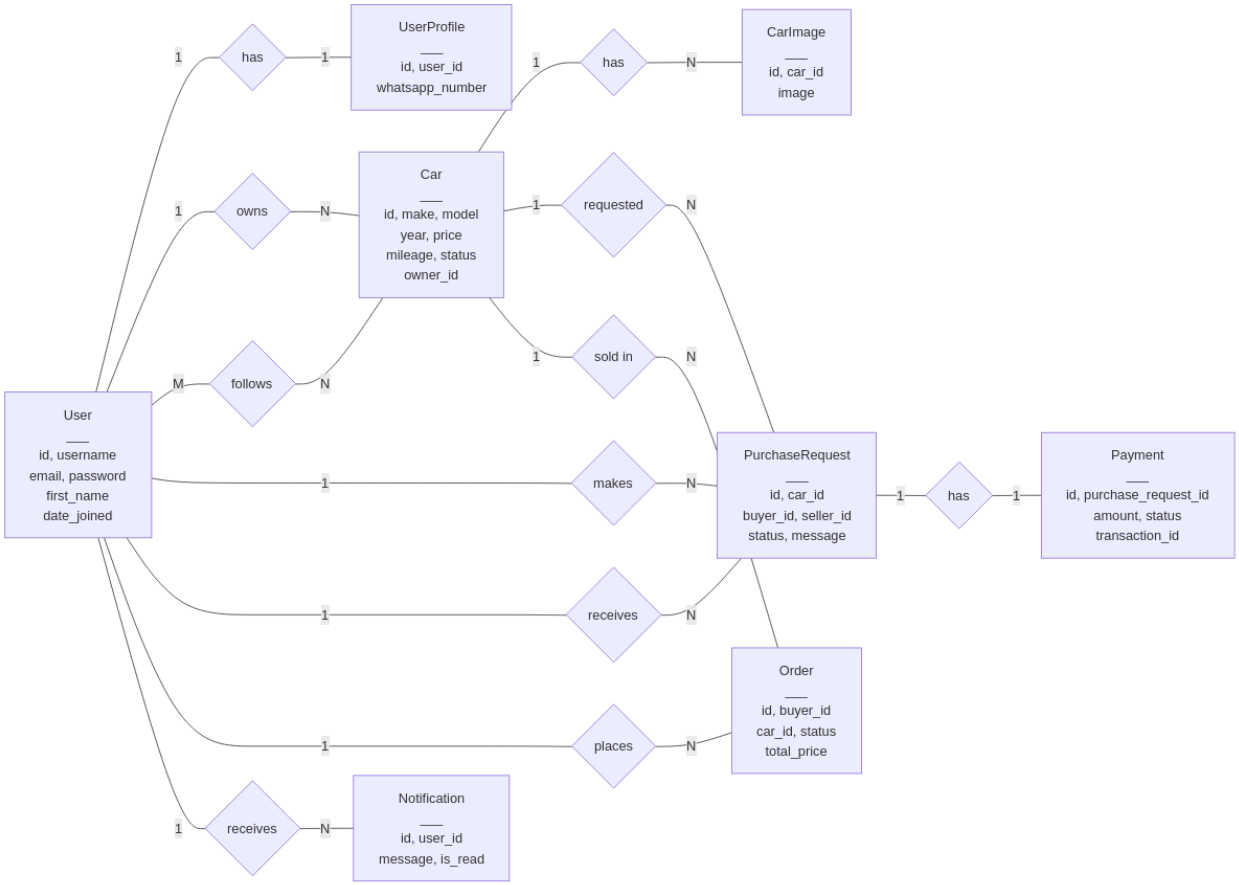


10. ENTITY RELATION DIAGRAM (ERD):

a) Proposed ERD



b) Final ERD:



END of the REPORT