Welcome to Socon-MKT Documentation

Welcome to the official documentation for **Socon-MKT**, a hybrid social marketplace that rethinks how people buy, sell, and interact online.

At its core, Socon-MKT is built to bridge the gap between community engagement and secure e-commerce. Use the directory below to navigate the system architecture, frontend and backend breakdowns, and deployment guidelines.

Master Table of Contents

PROJECT OVERVIEW

- The Problem & Solution
- Target Audience

SYSTEM ARCHITECTURE

- Frontend Stack
- Backend Stack
- Database & Storage

LOCAL DEVELOPMENT SETUP

- Prerequisites
- Installation Steps
- Environment Variables

FRONTEND PAGES BREAKDOWN

- Feeds
- Marketplace
- Discover Sellers (Shop)
- Cart & Checkout
- Inbox & Negotiations
-

Notifications
- Saved Items
- Create Post
- Upload Product
- Product Detail
- Social Post Detail
- Chat Room
- Seller Profile
- User Profile

BACKEND & ADMIN

- Django Admin & Models

DEPLOYMENT & INFRASTRUCTURE

- Server Setup
- Web Server Configuration
- Application Server
- Security & DNS

The Problem & Solution

The Problem: Trust and Engagement

If you look at traditional online classifieds or peer-to-peer marketplaces today, they typically suffer from two major flaws:

1. **The Trust Deficit (The Scam Problem):** Buying from a stranger online is risky. Buyers are terrified of sending money and never receiving the product. Sellers are tired of shipping items only to be ghosted, or dealing with fake payment receipts. There is zero built-in protection.
2. **The "Static Catalog" Feel:** Most marketplaces are transactional and boring. You log in, search for an item, and leave. There is no community, no reason to stick around, and no way for sellers to build a genuine following beyond their product listings.

The Solution: A Social Marketplace with Built-in Security

Socon-MKT solves these problems by combining the engaging feed of a social media network with the robust transaction security of an escrow service.

Here is how the platform fixes the broken marketplace experience:

- **Community-First Content:** Socon-MKT isn't just for selling. Users can post standard social content, share recommendations, and interact with media. This creates an active, returning user base. Sellers aren't just faceless vendors; they are active members of the community.
- **Transparent Negotiation Tracker:** When a buyer finds a product they like, they don't just send a generic message. They enter a dedicated chat where the seller can officially update the progress of the negotiation right on the platform. Both parties always know exactly where the deal stands.
- **Flexible Payments & Optional Escrow:** When a deal is struck, buyers and sellers have the freedom to choose how they complete the transaction. If they are comfortable, they can proceed with a direct, off-platform payment. However, to completely eliminate risk, they can opt into Socon-MKT's admin-managed escrow service.
- If escrow is used, the buyer funds the platform's secure account instead of paying the seller directly.
- The seller is notified that the funds are safely locked in.
- The seller delivers the product.
- Once the buyer confirms receipt, the admin releases the funds to the seller.
- *Result:* Users get the freedom to transact on their own terms, backed by an on-demand safety

net that guarantees payment and prevents theft.

By merging social engagement with risk-free transactions, Socon-MKT provides a space where users can scroll, chat, and shop with complete peace of mind.

Target Audience

Socon-MKT is built on a simple philosophy: **The market belongs to everyone.**

Because the platform merges social networking with e-commerce, the barrier to entry is virtually zero. Anyone can be a buyer, anyone can be a seller, and anyone can just be a creator posting engaging content.

Who is Socon-MKT Built For?

Rather than targeting a narrow niche of professional e-commerce vendors, Socon-MKT is designed for the everyday person. It specifically targets users who are frustrated by the physical limitations of traditional shopping.

1. The Time-Starved Professional

For the busy worker, time is money. Going to a physical market means taking hours out of a busy schedule. Socon-MKT allows them to browse, negotiate, and secure an item via Escrow during their lunch break or commute, completely bypassing the hassle of physical shopping.

2. The Traffic-Weary Shopper

City traffic jams can turn a simple errand into a half-day nightmare. Users who dread the congestion, noise, and chaos of physical markets can find exactly what they need from the comfort of their couch.

3. The Distant Buyer

Not everyone lives next to a major commercial hub. For users separated by long distances, Socon-MKT brings the market to their fingertips, allowing them to negotiate and arrange deliveries without leaving their neighborhood.

4. The Casual Browser (The "Bored" User)

Traditional marketplaces are boring if you aren't actively looking to buy something. Because anyone can post non-product content (videos, memes, updates), users open Socon-MKT just to see what's happening in their community. This social scrolling naturally

leads to product discovery.

The Data: Why Users Are Shifting Online

Understanding the user's pain points is critical to seeing the value of Socon-MKT. Below is a breakdown of the primary reasons users abandon physical markets in favor of our social marketplace.

--- title: Primary Drivers for Choosing Socon-MKT --- pie "Traffic Jams and Commute" : 45 "Busy Schedules or No Time" : 30 "Long Distances" : 15 "Boredom or Entertainment" : 10

Frontend Architecture

The Socon-MKT user interface is built as a highly responsive, single-page web application (SPA). Currently operating exclusively as a web platform, the frontend is engineered to handle both heavy e-commerce interactions (like the negotiation tracker) and dynamic social media feeds without sacrificing performance.

Core Technology Stack

We chose a modern, type-safe React stack to ensure the codebase remains scalable and maintainable as the community and marketplace features grow.

Technology	Role in Socon-MKT	Why We Use It
React (v18+)	UI Framework	Provides a component-based architecture perfect for building reusable UI elements like product cards, chat bubbles, and feed posts.
TypeScript	Language & Tooling	Adds static typing to JavaScript, catching errors at compile-time and strictly defining our API response payloads (e.g., `Product`, `User`, `Message` interfaces).
Tailwind CSS	Primary Styling	Utility-first CSS framework used for rapid, responsive UI development without leaving the JSX files.
Zustand	Global Client State	A small, fast, and scalable bearbones state-management solution used for UI state (e.g., toggling dark mode, opening modals, managing active chat windows) without the boilerplate of Redux.
React Query	Data Fetching & Server State	Handles fetching, caching, synchronizing, and updating server state. It eliminates the need to store API responses in global state and handles loading/error states automatically.
Standard CSS	Custom Styling	Used sparingly for complex keyframe animations, global CSS variables, or styling exceptions that Tailwind's utility classes cannot easily handle.

❗ Future Mobile Expansion

While Socon-MKT is currently a web-only platform, all API calls and global state mechanisms have been abstracted. This ensures that when the platform eventually

expands to mobile (e.g., via React Native), the core business logic (Zustand stores and React Query hooks) can be easily reused.

Component Architecture Flow

The frontend follows a modern separation of concerns, clearly dividing **Client State** (UI toggles, themes) managed by Zustand, and **Server State** (database records, API responses) managed by React Query.

Below is a high-level view of how a user interaction (like sending a chat in the negotiation tracker) moves through the frontend system.

graph TD A["User Interaction
 e.g., Clicks 'Buy Product'"] --> B["React Component
 ChatBox.tsx"] B --> C{"State Handling"} C -->|"Client UI State
 (e.g., Open Modal)"| D["Zustand Store"] D -->|"Triggers Re-render"| B C -->|"Server Mutation / Fetching"| E["TanStack React Query <br/> useMutation / useQuery"] E -->|"HTTP Request"| F[("Django Backend API")] F -->|"JSON Response"| E E -->|"Updates Cache & Background Sync"| B E -->|"Error/Loading States"| B

Backend Architecture

The backend of Socon-MKT is powered by Python and Django. It acts as the central hub for the platform, securely managing the REST API, complex database interactions, asynchronous background tasks, and third-party integrations like email services.

Core Technology Stack

Technology	Role	Why We Use It
Python	Core Language	Provides a robust, readable, and highly scalable foundation for backend logic.
Django	Web Framework	Offers built-in security, ORM, and admin interfaces, allowing us to focus on business logic rather than reinventing the wheel.
Django REST Framework (DRF)	API Layer	Facilitates the rapid creation of robust API endpoints and handles complex serialization of nested database models.
Celery	Background Operations	Offloads heavy tasks (like processing large media uploads or sending transactional emails) from the main request/response cycle, keeping the API fast.
Resend	Email Service	A modern, developer-friendly API for sending transactional emails (like escrow alerts or welcome messages) with high deliverability.
Tailwind CSS & HTML	Email Templates	We use Tailwind to style our custom HTML email templates, ensuring responsive, modern-looking transactional emails across all email clients.

Modular Project Structure

As platforms grow, standard Django apps that dump all logic into single `views.py` or `models.py` files quickly become unmanageable. Socon-MKT avoids this by using a **modular folder architecture** across its core applications.

By converting components like `models`, `views`, and `serializers` into Python packages (folders with `__init__.py` files), we keep the codebase clean, highly scalable, and easy to navigate.

- The `core` app handles platform-wide logic like API keys, permissions, custom exceptions, paginations, and signals.
- The `post` app divides models, serializers, and views into separate directories for better maintainability.
- We use an `http` directory to store `.http` files for quick and organized API testing.

```
SOCON_MKT/
├── .env                    # Contains all private API keys, database URLs, and sec
├── .env.example
├── .gitignore
├── db.sqlite3
├── manage.py
├── requirements.txt
├── socon_mkt/              # Main project configuration (settings, wsgi, asgi)
├── core/                   # The central app for platform-wide logic
│   ├── migrations/
│   ├── models/             # Modular models directory
│   │   ├── __init__.py
│   │   ├── api_keys_models.py
│   │   └── reposts_models.py
│   ├── permissions/        # Custom DRF permission classes
│   │   ├── __init__.py
│   │   ├── admin_only.py
│   │   └── api_permission.py
│   ├── serializers/        # Modular serializers directory
│   │   ├── __init__.py
│   │   ├── api_key_serializers.py
│   │   └── report_serializers.py
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── exceptions.py       # Custom API exception handlers
│   ├── paginations.py      # Global pagination classes
│   ├── signals.py          # Django signals for cross-app events
│   ├── tests.py
│   ├── urls.py
│   └── views.py
├── post/                   # Feature-specific application for posts
│   ├── management/
│   ├── migrations/
│   ├── models/             # Modular models for posts
│   ├── serializer/         # Modular serializers for posts
│   ├── views/              # Modular views for posts
│   ├── __init__.py
│   ├── admin.py
│   ├── apps.py
│   ├── filter.py           # Custom DRF filtersets
│   ├── pagination.py
│   ├── tests.py
│   └── urls.py
├── chat/                   # Application for real-time or REST chat logic
│   ├── migrations/
│   ├── __init__.py
│   ├── admin.py
```

```
│   ├── apps.py
│   ├── models.py           # Standard models layout
│   ├── pagination.py
│   ├── serializers.py
│   ├── tests.py
│   ├── urls.py
│   └── views.py
├── notifications/          # Feature application for alerts
├── users/                  # Feature application for user management
├── http/                   # REST Client testing requests
│   ├── auth.http
│   ├── chats.http
│   ├── core.http
│   ├── notification.http
│   └── posts.http
├── templates/              # Handles email rendering and Tailwind HTML templates
├── utils/                  # Shared utility scripts and helpers
├── media/                  # User-uploaded files
├── post_profile_pics/      # App-specific media uploads
├── statics/                # Collected static assets
└── staticfiles/            # Static file source directory
```

Database & Storage Architecture

The data layer of Socon-MKT is designed with a strict separation between local development and production environments. This approach ensures that local development remains lightweight, fast, and easy to set up, while the production environment is robust, highly scalable, and secure.

Database Architecture

Socon-MKT relies on a relational database to manage critical platform data, including users, marketplace products, chat histories, and escrow transactions.

Development: SQLite3

For local development, the platform uses **SQLite3**.

- **Why:** It requires zero external configuration, is supported natively by Python, and allows developers to get the project running instantly.
- **Location:** The entire database is stored as a single local file (`db.sqlite3`) at the root of the project. This file is excluded from version control via `.gitignore`.

Production: PostgreSQL

In the live production environment, the platform switches to **PostgreSQL**.

- **Why:** PostgreSQL handles concurrent connections effortlessly, ensures strict data integrity, and supports the complex, high-volume querying required by a social marketplace and active negotiation trackers.
- **Security:** To prevent credential leaks, all database settings are injected dynamically using environment variables.

.env PostgreSQL Configuration

```
DB_ENGINE=django.db.backends.postgresql
DB_NAME=socon_mkt_prod
DB_USER=socon_admin
DB_PASSWORD=your_secure_password
DB_HOST=your_database_host_ip
DB_PORT=5432
```

File Storage Architecture

Socon-MKT handles two types of files:

1. **Static Files:** CSS, JavaScript, fonts, and platform images needed for the site to function.
2. **Media Files:** User-generated content, such as profile pictures (`post_profile_pics/`), post images, and product uploads.

Development Storage (Local File System)

When running locally (`DEBUG = True`), Django serves both static and media files directly from your computer's hard drive.

File Type	Source Directory	Root/Destination	URL Path
Static Files	`statics/`	`staticfiles/`	`/static/`
Media Files	Stored directly in `media/`	`media/`	`/media/`

Production Storage (Cloudflare R2)

Serving static and media files directly from a Django application server in production is highly inefficient. For the live deployment, Socon-MKT offloads all file storage to **Cloudflare R2**.

- **Why Cloudflare R2?** R2 is an S3-compatible object storage solution. It provides lightning-fast content delivery via Cloudflare's global CDN while avoiding the egregious egress fees typically associated with AWS S3.
- **How it works:** When a user uploads a product image, Django processes the file and immediately pushes it to the R2 bucket. When users view the feed, their browsers fetch the images directly from Cloudflare, bypassing the Django server entirely.

Environment Data Flow

Below is a visual representation of how Django routes database queries and file requests based on the active environment.

graph TD A{"Active Environment"} A -->|"DEBUG = True (Local)"| B["Local Django Server"] B -->|"DB Queries"| C[("SQLite3
 (db.sqlite3)")] B -->|"Static Files"| D["Local /staticfiles/"] B -->|"Media Uploads"| E["Local /media/"] A -->|"DEBUG = False (Production)"| F["Production Django Server"] F -->|"DB Queries"| G[("PostgreSQL
 (Managed DB)")] F -->|"Uploads & Collectstatic"| H[("Cloudflare R2
 (Object Storage)")] H -->|"CDN Delivery"| I["End User Browser"]

Prerequisites

Before setting up the Socon-MKT environment on your local machine, ensure you have the following system requirements and software installed.

Required Software

To run both the React/Vite frontend and the Django backend, you will need the following tools installed on your operating system:

- **Python (3.10 or higher):** Required to run the Django backend, Celery workers, and manage the virtual environment.
- **Node.js (v18.0 or higher):** Required to run the Vite frontend development server and manage npm packages.
- **Git:** For version control and cloning the repository.
- **SQLite3:** The default database used for local development (this is typically pre-installed on Linux/macOS and bundled with Python).

❗ Kali Linux / Ubuntu Users

If you are running a Debian-based system, you can verify your installations by running the following in your terminal: `bash`

```
    python3 --version
```

```
    node -v
```

```
    git --version
```

Optional (But Recommended) Software

- **PostgreSQL (v14+):** While SQLite is the default for local testing, installing PostgreSQL locally is recommended if you wish to perfectly mirror the production database environment.
- **Redis:** If you plan on actively developing and testing the Celery background tasks locally, you will need a Redis server running as the message broker.

Local Installation

Socon-MKT utilizes a decoupled architecture. To run the platform locally, developers must initialize both the Django backend and the React frontend from their respective repositories.

Setup Instructions

Choose the environment you wish to configure below. It is required to have the backend running before the frontend can successfully make API calls.

=== "Backend (Django)"

````
The backend handles REST API requests, database migrations, and background Celery ta

**1. Clone the Repository**
```bash
git clone git@github.com:DevNinja254/socon_backend.git
cd socon_backend
```

**2. Initialize Virtual Environment**
```bash
python3 -m venv venv
source venv/bin/activate  # Use venv\Scripts\activate on Windows
pip install -r requirements.txt
```

**3. Environment Configuration**
Before proceeding, you must configure your local environment variables. Create a `.e

**4. Apply Migrations & Run Server**
```bash
python manage.py migrate
python manage.py runserver 0.0.0.0:8000
```
*The API will be available at `http://localhost:8000`.*
````

=== "Frontend (React + Vite)"

````
The frontend is a single-page application built with React, TypeScript, and Tailwinc

**1. Clone the Repository**
```bash
````
````

````
git clone git@github.com:DevNinja254/sokoni-mkt.git
cd sokoni-mkt
```


**2. Install Dependencies**
```bash
npm install
```


**3. Environment Configuration**
Create a `.env` file in the project root. Ensure your API endpoint points to your lo

**4. Start Development Server**
```bash
npm run dev
```
*Vite will launch the local development server at `http://localhost:5173`.*
````
````

Environment Variables

Socon-MKT relies on environment variables to securely manage API keys, database credentials, and storage endpoints. These values differ between local development and production environments.

Security Notice

The `.env` files must never be committed to version control. Ensure `.env` is declared in the `.gitignore` file of both repositories before deployment.

Frontend Variables (`.env`)

The frontend environment configuration is maintained in the root directory of the `sokoni-mkt` repository. Vite requires all custom environment variables to be prefixed with `VITE_`.

Below is the structure of the frontend environment variables:

```
# Local Development URLs
VITE_API_BASE_URL="<local_backend_api_url>"
VITE_API_BASE_DOMAIN="<local_backend_domain>"

# Google Authentication
VITE_GOOGLE_WEB_CLIENT_ID="<google_client_id>.apps.googleusercontent.com"

# Platform API Key
VITE_API_KEY="<frontend_api_key>"
```

Backend Variables (`.env`)

The backend environment configuration is maintained in the root directory of the `sokoni-mkt-backend` repository (Django). These variables control debug mode, security, database connections, and object storage.

Below is the structure of the backend environment variables:

```
# Core Django Settings
```

```
DEBUG=<boolean_value>
SECRET_KEY="<django_secret_key>"ALLOWED_HOSTS=<comma_separated_allowed_hosts>

# External APIsRESEND_API_KEY="<resend_api_key>"# Database Configuration (Developmer
DB_USER=<database_user>
DB_PASSWORD="<database_password>"DB_HOST=<database_host_ip>
DB_PORT=<database_port>

# Cloudflare R2 Settings (Object Storage)AWS_ACCESS_KEY_ID='<cloudflare_r2_access_ke
```

Database Switching

The project uses SQLite for local development (`DATABASE_URL`) and PostgreSQL in production (`DB_*` variables). Ensure only the relevant block is active depending on the environment, or that your settings logic switches between them based on `DEBUG`.

Main Feed (Posts & Products)

The `FeedPage` serves as the primary dashboard and landing interface for authenticated users on Socon-MKT. It features a continuous, infinitely scrolling feed that intelligently merges standard social media posts with e-commerce product listings.

The page is built using a responsive CSS Grid layout, splitting into a central content feed and a sticky right-rail for desktop users.

Component Architecture

The feed is highly modular, breaking down complex data rendering into specialized React components.

1. ComposerComponent

Located at the very top of the feed, this component acts as the entry point for content creation. It reads the authenticated user's profile data (`ProfileResponseType`) to display their avatar and offers two distinct routing actions:

- **Sell a product:** Routes the user to the `/upload` marketplace flow.
- **Write a post:** Routes the user to the standard `/create-post` social flow.

2. PostComponentItems

The core rendering engine for individual feed items. It dynamically adjusts its UI based on the `is_product` boolean flag within the data payload.

- **Media Grid Engine:** Handles complex rendering for media arrays. It conditionally renders single images, video players, or multi-item grids (up to 4+ items with an overlay counter).
- **Product Badges:** If `is_product` is true, it automatically injects a floating price tag, condition pills (e.g., "New", "Used"), and a "Seller" verification badge.
- **Interactions:** Handles internal mutations for liking (`usePostLike`), saving (`usePostSave`), and sharing (`usePostShare`) via the Web Share API or clipboard fallback.

3. Seller Discovery Injections

To drive marketplace engagement, seller recommendations are seamlessly injected into

the user's feed.

- **SellerCarousel:** A horizontal, scrollable list of suggested sellers. Rather than sitting at the top of the feed, it calculates random index positions (`carouselPositions`) to organically inject itself *between* feed posts as the user scrolls.
- **SellersToFollowLg:** A sticky right-rail component visible only on large screens. It utilizes the same cached seller query as the carousel, alongside a localized "Trending Nearby" subset of posts filtered strictly for media-rich content.

4. CarouselModal (Lightbox)

A full-screen, high-z-index overlay triggered when a user clicks on any post media. It supports infinite looping through media arrays and natively handles both `<img />` and `<video controls />` rendering based on the `media_type`.

State Management & Infinite Scrolling

The feed relies heavily on `@tanstack/react-query` to manage server state and caching, completely eliminating the need for global Redux stores for post data.

- **Intersection Observer:** The `react-intersection-observer` package monitors a hidden sentinel `<div>` at the bottom of the feed list.
- **Cursor Pagination:** When the sentinel enters the viewport, the `useInfiniteFetchPosts` hook triggers the `fetchNextPage()` function. The application extracts the `c=` (cursor) parameter from the backend's `next` URL string and passes it into the subsequent API call, seamlessly appending new `PostType` items to the flattened data array.

API Endpoints & Data Structures

The `FeedPage` interacts with three primary API endpoints via the globally configured Axios `authApi` instance. This instance automatically intercepts requests to attach the Bearer token and proactively refreshes expired sessions.

1. User Statistics

Fetches the current user's profile details and notification badges for the Composer UI.

- **Endpoint:** `GET /users/stat/`
- **Authorization:** Required (Bearer Token)
- **Response Structure (`ProfileResponseType`):**

```
{
  "profile": {
    "user": {
      "id": "string",
      "username": "string",
      "email": "string"
    },
    "profile_id": "string",
    "profile_pic": "string | null",
    "created_at": "ISO 8601 string",
    "full_name": "string"
  },
  "total_notification": 0,
  "total_message": 0
}
```

2. Main Feed (Infinite Posts)

Retrieves the paginated list of social posts and product listings.

- **Endpoint:** `GET /posts/`
- **Query Parameters:** `c` (Cursor string for pagination)
- **Authorization:** Required (Bearer Token)
- **Response Structure (`PostResponseType`):**

```
{
  "next": "url_string | null",
  "previous": "url_string | null",
  "results": [
    {
      "id": "string",
      "author": { /* AuthorType */ },
      "caption": "string | null",
      "is_product": true,
      "title": "string | null",
      "price": "string | null",
      "category": ["string"],
      "quatity": 1,
      "location": "string | null",
      "listing_conditions_display": ["string"],
      "media": [
        {
          "id": "string",
          "file": "string_url",
```

```
            "media_type": "image | video",          "thumbnail": "string_url | nul
```

3. Suggested Sellers (Infinite List)

Retrieves a paginated list of verified marketplace sellers to populate the discovery carousels.

- **Endpoint:** `GET /users/sellers/`
- **Query Parameters:** `c` (Cursor string for pagination)
- **Authorization:** Required (Bearer Token)
- **Response Structure (`SellerListResponse`):**

```
{
  "next": "url_string | null",
  "previous": "url_string | null",
  "results": [
    {
      "id": "string",
      "profile": { /* BaseProfile */ },
      "business_title": "string",
      "is_verified": true,
      "business_logo": "string_url | null",
      "is_following": false
    }
  ]
}
```

Marketplace Page

The `MarketplacePage` serves as the dedicated e-commerce hub for Socon-MKT. Unlike the Main Feed, which blends social content with commerce, this page is strictly focused on product discovery, categorization, and browsing.

The layout is constrained to a maximum width of 1100px for optimal viewing across devices and is divided into three distinct vertical sections.

Component Architecture

The page aggregates three modular components, each responsible for fetching and rendering a specific slice of the marketplace data.

1. MarketCategories

Located at the top of the page, this component renders a responsive grid of available product categories (e.g., Electronics, Fashion).

- **Visuals:** Displays a category icon/image with a fallback emoji (📦) if no image is provided.
- **Routing:** Clicking a category routes the user to a filtered view (`/category/:id`).
- **UX:** Implements a localized pulse skeleton loading state while fetching data.

2. RecentFeaturedMarket

A horizontal, snap-scroll carousel highlighting newly added or featured products.

- **Data Fetching:** Fetches a hard-limited list of the 10 most recent products by applying the `-created_at` ordering parameter.
- **Fail-Safe:** If this request fails, the component silently fails (returns `null`) rather than breaking the page, ensuring the user can still browse the main market.

3. MainMarketList

The primary product grid featuring infinite scrolling.

- **Product Cards:** Renders responsive square cards for each item. If a product has a price, it overlays a custom, rotated "Soko Price Tag" styling.

- **Infinite Scroll:** Utilizes a bottom sentinel `div` to automatically fetch the next cursor-based page of products as the user scrolls down.
- **Empty States:** Provides a styled empty state UI if no products exist in the marketplace yet.

State Management & Infinite Scrolling

Data fetching is handled exclusively through `@tanstack/react-query` to ensure fast navigation and cached responses.

- **Standard Queries:** The Categories and Featured sections use standard `useQuery` hooks.
- **Infinite Queries:** The Main Market uses `useInfiniteQuery`. It intercepts the `next` URL string returned by the Django backend, extracts the `c=` (cursor) parameter, and passes it to the next API call.
- **Scroll Detection:** The `react-intersection-observer` package monitors the `loadMoreRef` sentinel. When it enters the viewport (`inView === true`), it triggers the `fetchNextPage()` function.

API Endpoints & Data Structures

The Marketplace interacts with two primary API endpoints via the globally configured Axios `authApi` instance.

1. Market Categories

Retrieves the full list of active product categories.

- **Endpoint:** `GET /posts/market_category/`
- **Authorization:** Required (Bearer Token)
- **Response Structure (`MarketCategoryType[]`):**

```
[
  {
    id: "string",
    image: "string_url | null",
    title: "string",
  },
];
```

2. Recent Featured Products

Retrieves the 10 most recently added products for the top carousel.

- **Endpoint:** `GET /posts/marketplace/`
- **Query Parameters:**
- `page_size`: 10
- `o`: "-created_at"
- **Authorization:** Required (Bearer Token)
- **Response Structure (`PaginatedMarketplaceResponse`):** *(See structure below)*

3. Main Market List (Infinite)

Retrieves the paginated list of all marketplace products.

- **Endpoint:** `GET /posts/marketplace/`
- **Query Parameters:** `c` (Cursor string for pagination)
- **Authorization:** Required (Bearer Token)
- **Response Structure (`PaginatedMarketplaceResponse`):**

```
{
  "next": "url_string | null",
  "previous": "url_string | null",
  "results": [
    {
      "id": "string",
      "author": {
        "profile": {
          "seller_profile": "string | null"
        }
      },
      "title": "string",
      "price": "string | null",
      "unit": "string | null",
      "media": [
        {
          "file": "string_url",
          "media_type": "image | video",
          "thumbnail": "string_url | null"
        }
      ]
    }
  ]
}
```

Discover Sellers (Shop Page)

The `SellersPage` serves as the centralized directory for users to discover, browse, and connect with merchants on Socon-MKT. The interface provides a streamlined, infinitely scrolling feed of seller profiles, allowing users to quickly read business bios and follow stores to customize their marketplace experience.

Component Architecture

The page is built using a clean, list-based architecture focused on performance and seamless user interaction.

1. SellersPage

The main layout wrapper component constrained to a maximum width of 800px to maintain readability.

- **Initialization:** Forces the window to scroll to the top immediately upon mounting (`window.scrollTo`).
- **Loading States:** Implements a full-page `LoadingSkeleton` for the initial fetch, and a smaller inline spinner for subsequent infinite scroll fetches.
- **Empty & Error States:** Provides stylized fallback UI components (`ErrorUi`) if the network fails or if the platform currently has no registered sellers.

2. SellerItem

The individual row component representing a single merchant.

- **Visuals:** Extracts and prioritizes the `business_logo` and `business_title`, falling back to the user's standard profile picture and username if specific business details are missing. Includes an absolute-positioned verification badge (✓) for trusted sellers.
- **Routing:** Clicking the card routes the user to the specific seller's store view (`/sellers/:id`).
- **Action Button:** Features an interactive Follow/Following toggle button that stops event propagation to prevent accidental navigation when clicked.

State Management & Optimistic UI

This page heavily leverages `@tanstack/react-query` to handle both the data ingestion and complex user interactions seamlessly.

- **Infinite Scrolling (`useFetchSellers`):** Similar to the feed and marketplace, it uses a sentinel `div` monitored by `react-intersection-observer`. When the sentinel is in view, it extracts the `c` cursor parameter and triggers `fetchNextPage()`.
- **Optimistic Follow Updates (`useToggleFollow`):** To make the app feel instantaneously fast, the Follow button does not wait for the server to respond.
- When clicked, the mutation cancels any outgoing seller fetches.
- It immediately alters the React Query cache, flipping the `is_following` boolean for that specific seller.
- If the backend API call fails, it rolls the cache back to its previous state to prevent data desynchronization.

API Endpoints & Data Structures

The Discover Sellers page interacts with two primary API endpoints via the globally configured Axios `authApi` instance.

1. Fetch Sellers List (Infinite)

Retrieves the paginated list of all active marketplace sellers.

- **Endpoint:** `GET /users/sellers/`
- **Query Parameters:** `c` (Cursor string for pagination)
- **Authorization:** Required (Bearer Token)
- **Response Structure (`SellerListResponse`):**

```
{
  "next": "url_string | null",
  "previous": "url_string | null",
  "results": [
    {
      "id": "string",
      "profile": {
        "user": {
          "id": "string",
          "username": "string",
```

```
            "email": "string"        },        "profile_id": "string",        "pro
```


2. Toggle Follow Status

Sends a mutation to follow or unfollow a specific seller's profile.

- **Endpoint:** `POST /users/follow/`
- **Payload:**

```
{
  "following": "profile_id_string"
}
```

- **Authorization:** Required (Bearer Token)

Cart Page

The `CartPage` acts as the user's staging area for purchasing products. Because Socon-MKT operates as a peer-to-peer social marketplace, the cart behaves differently than traditional e-commerce platforms. Instead of a single checkout gateway, items are grouped by individual sellers, and "checkout" is initiated via direct chat negotiations.

The page is constrained to a readable maximum width of 760px and is designed for a highly responsive mobile and desktop experience.

Component Architecture

The cart view breaks down complex, paginated data into logical, seller-based groupings.

1. CartPage (Main Wrapper)

This component fetches the raw, paginated cart data and performs heavy data transformation on the client side.

- **Data Grouping (`groupedCart`):** Uses a `useMemo` hook to iterate over the flattened infinite list of cart items and groups them by `sellerId`. This ensures that even if a user adds items from the same seller at different times, they appear together in the UI.
- **Total Estimation (`grandTotal`):** Calculates the total cost of all items in the cart.
- **Summary Card:** A sticky right-rail (on desktop) or bottom block (on mobile) that displays the total estimate, reminding the user that shipping fees and final negotiations happen per seller.
- **Empty & Error States:** Provides fallback UI if the cart is empty or if the network request fails.

2. CartItem

Renders a specific group of products belonging to a single seller.

- **Seller Header:** Displays the seller's avatar, business title, and verification badge, linking directly to their store profile.
- **Product List:** Maps through the `items` array for that specific seller, rendering the product image, title, and a customized "Soko Price Tag".
- **Action - Remove Item:** Allows users to remove a product from their cart via the `useDeleteCart` mutation.
- **Action - Message Seller:** Acts as the "checkout" button for this specific seller. It triggers the

`useFetchChatId` hook to initiate a chat room using the first product in the group as the context, then dynamically redirects the user to the `inbox/:chatId` route.

State Management & Optimistic UI

The cart relies on `@tanstack/react-query` to handle pagination, state synchronization, and immediate UI feedback.

- **Infinite Scrolling (`useInfiniteFetchCart`):** Uses a sentinel `div` monitored by `react-intersection-observer` to seamlessly fetch the next cursor-based page of cart items as the user scrolls.
- **Optimistic Deletions (`useDeleteCart`):** When a user removes an item, the UI updates instantly. The mutation intercepts the active query cache, manually filters out the deleted `productId` from the deeply nested infinite pages array, and updates the screen before the server even responds. If the backend request fails, it automatically rolls back to the previous state.

API Endpoints & Data Structures

The cart leverages four specific endpoints via the globally configured Axios `authApi` instance.

(Note: Deletion uses a `POST` request to a specific action endpoint rather than a standard `DELETE` method).

1. Fetch Cart (Infinite)

Retrieves the paginated list of items currently in the user's cart.

- **Endpoint:** `GET /posts/cart/`
- **Query Parameters:** `c` (Cursor string for pagination)
- **Authorization:** Required (Bearer Token)
- **Response Structure (`CartResponse`):**

```
{
  "next": "url_string | null",
  "previous": "url_string | null",
```

```
  "results": [    {      "id": "string",      "created_at": "ISO 8601 string",
```

2. Add Item to Cart

Adds a new product to the user's cart.

- **Endpoint:** `POST /posts/cart/`
- **Payload:**

```
{
  "product": "product_id_string"
}
```

- **Authorization:** Required (Bearer Token)

3. Remove Item from Cart

Removes a specific product from the cart.

- **Endpoint:** `POST /posts/cart/delete_cart/`
- **Payload:**

```
{
  "productId": "product_id_string"
}
```

- **Authorization:** Required (Bearer Token)

4. Initiate Checkout Chat

Creates or retrieves an existing chat session with the seller regarding a specific product.

- **Endpoint:** `POST /chat/confirm_chat/`
- **Payload:**

```
{
  "productId": "product_id_string"
}
```

- **Authorization:** Required (Bearer Token)
- **Response Structure (`ChatIdReponse`):**

```
{
  "chatId": "string"
}
```

Inbox Page (Messages & Negotiations)

The `InboxPage` acts as the central communication hub for Socon-MKT. Because the platform relies on peer-to-peer negotiations rather than a standard checkout cart, this page tracks all active and past conversations between buyers and sellers, displaying real-time statuses like "Negotiating" or "Delivering."

The interface is constrained to a 680px maximum width, providing a focused, chat-app-like experience.

Component Architecture

The inbox is designed for quick scanning and easy filtering of active negotiations.

1. InboxPage (Main Wrapper)

This component fetches the paginated list of all active chat threads and provides a unified view for the user.

- **Local Search Filtering:** Features a text input that filters the loaded chats *locally* (client-side) rather than making new server requests. It safely checks the search query against the seller's business title, username, product title, and the last message preview.
- **Empty & Error States:** Renders contextual empty states (e.g., if the user has no chats globally vs. if a specific search query yields no results).
- **Infinite Scroll:** Utilizes the standard `react-intersection-observer` sentinel at the bottom of the list to automatically fetch older chat threads.

2. MessageItem

Renders an individual chat row summarizing the negotiation.

- **Contextual Badges:**
- `PriceTag`**:** Conditionally changes styling based on the negotiation status (e.g., gray if currently negotiating, gold otherwise).
- `StatusPill`**:** Dynamically renders color-coded pills based on the `order.status` ("Negotiating" [Gray], "Agreed" [Teal], "Delivering" [Violet], or "Completed" [Green]).
- **Visual Hierarchy:** Highlights unread threads using a tinted background (`#FFF8E8`), a bolder preview text, and a distinct red dot indicator. It also displays a miniature thumbnail of the

product being discussed to provide immediate visual context.

State Management & Real-Time Updates

The Inbox leverages `@tanstack/react-query` to handle both historical data pagination and near real-time updates.

- **Background Polling:** The `useFetchAllChats` hook implements a `refetchInterval: 10000` (10 seconds). This acts as a lightweight polling mechanism, automatically pinging the server in the background to fetch new messages and update the "last message preview" and "unread" statuses without requiring WebSockets.
- **Infinite Scrolling:** Uses the `c` (cursor) parameter from the backend's `next` pagination link to continuously append older chat threads as the user scrolls down the list.

API Endpoints & Data Structures

The inbox interacts with the backend via a primary fetch endpoint using the globally configured Axios `authApi` instance.

1. Fetch All Chats (Infinite)

Retrieves the paginated list of chat threads the current user is participating in.

- **Endpoint:** `GET /chat/all_chats/`
- **Query Parameters:** `c` (Cursor string for pagination)
- **Authorization:** Required (Bearer Token)
- **Response Structure (`ChatListResponse`):**

```
{
  "next": "url_string | null",
  "previous": "url_string | null",
  "results": [
    {
      "chat_id": "string",
      "created_at": "ISO 8601 string",
      "updated_at": "ISO 8601 string",
      "last_message_preview": "string",
      "is_own": false,
      "is_unread": true,
      "order": {
```

```
        "id": "string",        "status": "NEGOTIATING | COMPLETED | AGREED | DEL
```

Notifications Page

The `NotificationsPage` is the centralized activity feed for users on Socon-MKT. It tracks and displays all inbound interactions, including social engagements (likes, comments, follows) and marketplace activities (deal updates, reminders).

The layout is constrained to a narrow 620px maximum width, providing a highly readable, focused timeline experience similar to standard social networking platforms.

Component Architecture

The notification feed relies on client-side data transformation to categorize and prioritize new alerts for the user.

1. NotificationsPage (Main Wrapper)

This core component handles the fetching, sorting, and display logic of the activity feed.

- **Data Grouping (`groupedNotifications`):** Uses a `useMemo` hook to iterate over the flattened infinite list of notifications and dynamically split them into two distinct arrays: `unread` and `read`.
- **Visual Hierarchy:** The UI maps through these groups sequentially, rendering the "Unread" block at the top of the feed and the historical "Read" block directly beneath it, separated by distinct section headers.
- **Empty & Error States:** Renders a stylized fallback UI ("You're all caught up!") if the user has no historical notifications.

2. NotificationItem

Renders the individual notification row based on the specific event type.

- **Event Types:** Handles diverse notification triggers dictated by the `NotificationType` enum (`DEAL`, `COMMENT`, `LIKE`, `FOLLOW`, `REMINDER`).
- **Contextual Routing:** Links the user to the relevant destination (e.g., routing to a specific post ID for a `COMMENT`, or a user profile for a `FOLLOW`).

State Management & Automated Actions

The page utilizes `@tanstack/react-query` for data ingestion and incorporates a silent background mutation to manage the read state without requiring explicit user interaction.

- **Silent Auto-Read Logic:** The component utilizes a `useEffect` hook that monitors the `allNotifications` array. When new unread notifications are loaded into the DOM, the hook automatically extracts their IDs and fires the `useReadNotifications` mutation in the background. This ensures the global notification badge clears automatically just by viewing the page.
- **Infinite Scrolling:** Uses the standard `react-intersection-observer` sentinel to extract the `c` (cursor) parameter and continually fetch older notifications as the user scrolls down.

API Endpoints & Data Structures

The notifications system interacts with the backend via two primary endpoints through the `authApi` Axios instance.

1. Fetch Notifications (Infinite)

Retrieves the paginated list of all notifications targeted at the current user.

- **Endpoint:** `GET /notification/`
- **Query Parameters:** `c` (Cursor string for pagination)
- **Authorization:** Required (Bearer Token)
- **Response Structure (`NotificationListResponse`):**

```
{
  "next": "url_string | null",
  "previous": "url_string | null",
  "results": [
    {
      "id": "string",
      "content": "string",
      "notified_at": "ISO 8601 string",
      "to_user": "string",
      "type": "DEAL | COMMENT | LIKE | FOLLOW | REMINDER",
      "post": "string | null",
      "is_read": false,
      "from_user": {
        "id": "string",
        "username": "string",
        "email": "string",
```

```
        "profile": {          "profile_id": "string",          "profile_pic": "s
```

2. Mark Notifications as Read

Sends a batch array of notification IDs to the server to update their `is_read` boolean status to `true`.

- **Endpoint:** `POST /notification/mark_all_read/`
- **Payload (`ReadType`):**

```
{
  "readId": ["notification_id_1", "notification_id_2"]
}
```

- **Authorization:** Required (Bearer Token)

Saved Items Page

The `SavedPage` serves as a personalized bookmarking hub for users on Socon-MKT. Because the platform combines social networking with e-commerce, users can save both standard social posts and marketplace products. This page cleanly separates the two content types into distinct, easily navigable tabs.

The interface utilizes a wide 1100px maximum container to accommodate the dense product grids alongside standard vertical social feeds.

Component Architecture

The page relies on a single backend query, utilizing client-side logic to route the data into specialized rendering components based on the content type.

1. SavedPage (Main Wrapper)

This component handles the primary data ingestion, infinite scrolling, and tab state management.

- **Data Separation (`savedProducts` & `savedPosts`):** Rather than making two separate API calls, it uses a `useMemo` hook to iterate over the flattened `allSavedItems` array. It splits the array into two separate lists based on the `is_product` boolean attached to each post.
- **Tab Navigation:** Manages an `activeTab` state ("products" or "posts"), conditionally rendering either the `ProductTab` or `PostTab` component while displaying the precise count of items in each tab header.
- **Infinite Scroll:** Utilizes the standard `react-intersection-observer` sentinel to fetch older saved items dynamically as the user scrolls to the bottom of either tab.

2. ProductTab

Renders saved e-commerce listings in a responsive grid layout (2 columns on mobile, up to 4 on desktop).

- **Visual Logic:** Prioritizes media. If the product has a video, it renders a play button overlay. If it lacks media entirely, it dynamically renders a styled text block displaying the caption or title.
- **Price Tag:** Displays the rotated "Soko Price Tag" overlay if the item has an associated price.
- **Action Routing:** Clicking anywhere on the card routes the user to the dedicated

`/marketplace/:id` view, while clicking the "Unsave" button strictly fires a mutation and stops event propagation.

3. PostTab

Renders saved social media posts in a vertical, single-column feed.

- **Layout:** Emulates the Main Feed layout, displaying the author's avatar, business title, and the full text caption.
- **Media Constraints:** Restricts attached media (images or videos) to a maximum width of 400px to maintain readability and prevent massive images from dominating the screen.
- **Action Routing:** Clicking the post routes the user to the specific `/feeds/:id` view.

State Management & Optimistic UI

The Saved Items page leverages `@tanstack/react-query` to ensure the UI feels instantaneous, even when removing items from the list.

- **Optimistic Unsave (`useUnsavedPost`):** When a user clicks "Unsave" on either a product or a post, the application does not wait for a server response. The mutation instantly cancels outgoing fetches, manually filters out the target `postId` from the cached infinite query pages, and updates the DOM immediately. If the server request fails, it rolls the cache back to its previous state.
- **Infinite Scrolling (`useFetchPostSaved`):** Continuously appends older saved items by extracting the `c` (cursor) parameter from the backend's `next` pagination link.

API Endpoints & Data Structures

The page interacts with the backend via two main endpoints using the globally configured Axios `authApi` instance.

1. Fetch Saved Items (Infinite)

Retrieves a paginated list of all items (both posts and products) saved by the current user.

- **Endpoint:** `GET /posts/post_saved/`
- **Query Parameters:** `c` (Cursor string for pagination)

- **Authorization:** Required (Bearer Token)
- **Response Structure (`SavedPostListResponse`):**

```
{
  "next": "url_string | null",
  "previous": "url_string | null",
  "results": [
    {
      "id": "string",
      "post": {
        "id": "string",
        "title": "string | null",
        "price": "string | null",
        "unit": "string | null",
        "is_product": true,
        "caption": "string",
        "author": {
          "profile": {
            "seller_profile": "string | null"
          }
        },
        "media": [
          {
            "file": "string_url",
            "media_type": "image | video",
            "thumbnail": "string_url | null"
          }
        ]
      }
    }
  ]
}
```

2. Toggle Save Status (Unsave)

Toggles the saved status of a specific post. In the context of this page, it acts as a delete/unsave action.

- **Endpoint:** `POST /posts/toggle_post_save/`
- **Payload:**

```
{
  "postId": "post_id_string"
}
```

- **Authorization:** Required (Bearer Token)

Create Post Page

The `CreatePostPage` provides a streamlined interface for users to share standard social updates, photos, or videos with their community. It acts as the social counterpart to the marketplace, keeping users engaged with non-transactional content.

The page is constrained to a 640px maximum width to maintain a focused, distraction-free authoring environment.

Component Architecture

The interface is built to handle local media previews smoothly before committing any data to the server.

1. Media Handling & Previews

- **Constraints:** Users can upload a maximum of 10 photos **OR** 1 video (capped at 100MB). The component actively prevents mixing media types.
- **Memory Management:** When files are selected, the app generates local `URL.createObjectURL()` strings for instantaneous preview rendering. A `useEffect` cleanup function ensures `URL.revokeObjectURL()` is called when items are removed or the component unmounts, strictly preventing browser memory leaks.
- **Hidden Inputs:** Uses hidden native `<input type="file" />` elements triggered programmatically by styled UI buttons to maintain a clean aesthetic.

2. Upload UI & Error States

- **Dynamic Overlay:** Utilizes a `PendingSkeleton` component to block the UI and display a real-time progress bar during the upload phase.
- **Partial Failure Banner:** If the text post succeeds but the network drops during media upload, a specific error banner allows the user to retry *only* the failed media uploads without creating a duplicate text post.

State Management & Sequential Upload Logic

Content creation utilizes a custom `@tanstack/react-query` mutation hook (

`useCreateProductPost`) designed for robust, sequential file handling.

- **Caching Strategy:** The hook maintains a local `createdPostCache` state.
- **Step 1 (Text Payload):** The application first sends the text caption with `is_product: false`. The backend responds with the newly created Post ID.
- **Step 2 (Media Loop):** If media exists, the hook iterates through the `mediaFiles` array. It uses `FormData` (`multipart/form-data`) to upload each file sequentially, attaching the cached Post ID to each request.
- **Global Progress Tracking:** The hook intercepts Axios `onUploadProgress` events, mathematically combining the progress of individual files into a single, global percentage (0-100%) for the UI loader.

API Endpoints & Data Structures

The page interacts with three endpoints via the globally configured Axios `authApi` instance.

1. Fetch User Stats (Header UI)

Retrieves the user's basic profile details to display their avatar and name in the composer.

- **Endpoint:** `GET /users/stat/`

2. Create Post Payload

Creates the base post object in the database.

- **Endpoint:** `POST /posts/create_post/`
- **Payload (`PostProductRequestType` context):**

```
{
  "is_product": false,
  "caption": "string | null"
}
```

- **Authorization:** Required (Bearer Token)

3. Upload Media (Sequential)

Attaches a file to the previously created post. Called multiple times if multiple photos are selected.

- **Endpoint:** `POST /posts/create_post_media/`
- **Headers:** `Content-Type: multipart/form-data`
- **FormData Payload:**
- `post`: "post_id_string"
- `file`: [Binary File Data]
- **Authorization:** Required (Bearer Token)

Product Upload Page

The `ProductUploadPage` is the comprehensive merchant interface for listing items on the Socon-MKT marketplace. It handles complex form state, dynamic tagging, media uploads, and security validations to ensure only authorized sellers can create listings.

Component Architecture

The interface is broken down into modular cards, guiding the merchant through the listing process logically.

1. Security & Route Protection

- `useHasSellerProfile` **Check:** Upon mounting, the component verifies if the authenticated user has an active seller profile. If `is_seller` returns `false`, the component instantly redirects the user to the `/onboard` flow, preventing unauthorized marketplace listings.

2. Dynamic Tagging (`ToggleChip`)

- **Product Conditions:** Provides default presets (e.g., "Brand New", "Refurbished") while allowing sellers to type and append custom tags (e.g., "Missing Box", "Blue").
- **Listing Rules:** A boolean toggle system for store policies (e.g., "Payment on delivery", "No refunds").
- **Component:** Both utilize a shared `ToggleChip` component for a uniform, interactive pill-based UI.

3. Media Validation

- Shares the exact same robust media logic as the social post creator: 10 photos **OR** 1 video (100MB limit), utilizing `URL.createObjectURL()` for safe, instantaneous local previews.

State Management & Sequential Upload Logic

Because product listings often contain multiple high-resolution images, the page uses the exact same `useCreateProductPost` custom hook as the social feed to guarantee upload stability.

- **Form Compilation:** Before submission, the component validates required fields (Title, Price, Media). It then compiles the active boolean `listingConditions` and the string array of `productConditions` into arrays suitable for the backend payload.
- **Sequential Upload:**
- The complete JSON payload is sent to create the product entry.
- The resulting Product ID is cached.
- A loop uploads the media files one by one using `FormData`. If the internet connection drops during file 3 of 5, the merchant can click "Retry" to resume the media upload without recreating the base product.

API Endpoints & Data Structures

The page interacts with multiple endpoints to authorize the user, fetch form options, and process the upload.

1. Seller Authorization Check

Validates if the user is permitted to view this page.

- **Endpoint:** `GET /users/has_seller/`
- **Response (`IsSellerResponseType`):**

```
{
  "is_seller": true
}
```

2. Fetch Market Categories

Populates the category dropdown `<select>` menu.

- **Endpoint:** `GET /posts/market_category/`

3. Create Product Payload

Creates the base product listing in the database.

- **Endpoint:** `POST /posts/create_post/`
-

Payload (`PostProductRequestType`):

```
{
  "is_product": true,
  "title": "string",
  "price": 2800,
  "unit": "string | null",
  "category": "string",
  "caption": "string | null",
  "quatity": 1,
  "location": "string",
  "listing_conditions_text": ["Payment on delivery", "No refunds"],
  "product_conditions_text": ["Brand New", "Custom Tag"]
}
```

- **Authorization:** Required (Bearer Token)

4. Upload Product Media (Sequential)

Attaches files to the product.

- **Endpoint:** `POST /posts/create_post_media/`
- **Headers:** `Content-Type: multipart/form-data`
- **FormData Payload:**
- `post`: "product_id_string"
- `file`: [Binary File Data]
- **Authorization:** Required (Bearer Token)

Product Page

The `ProductPage` is the detailed view for a specific marketplace listing. It provides buyers with a comprehensive breakdown of the item, an interactive media gallery, and direct actions to either add the item to their cart or immediately initiate a purchase negotiation.

The interface is structured using a responsive CSS Grid with a maximum width of 1500px, splitting the layout into a sticky media gallery on the left and scrollable product details on the right for desktop users.

Component Architecture

The page aggregates several sub-components to organize the dense product data into digestible sections.

1. Media Gallery & Lightbox

- **Main Viewer:** Displays the currently selected image or video. Videos feature an absolute-positioned play button overlay and fall back to thumbnails when available.
- **Thumbnails:** A horizontally scrollable list of all attached media. Clicking a thumbnail updates the `mainMediaIndex` state to swap the main viewer's content.
- **CarouselModal:** Clicking the main image triggers a high-z-index, full-screen lightbox for immersive viewing.

2. Action Buttons

- **Buy Now:** Triggers the `useFetchChatId` hook. This acts as the direct checkout flow by generating (or retrieving) a dedicated chat room with the seller for this specific product, instantly redirecting the user to `/inbox/:chatId`.
- **Add to Cart:** Triggers the `useCreateCart` mutation. If the product is already in the cart (`isInCart`), the button dynamically changes its label and routes the user to the `/cart` page instead.

3. Tabbed Content Interface

To prevent vertical UI clutter, secondary information is organized into three toggleable tabs:

- **Details Tab:** Displays the product specifications (category, location, quantity), listing rules (delivery/payment terms), and specific product conditions.
- **Comments Tab:** Reuses the social comment thread component (`CommentsTab`).
- **Reviews Tab:** Uses `useFetchReview` to display an infinitely scrolling list of verified buyer reviews, including their star ratings and feedback.

State Management & Cache Seeding

This page uses `@tanstack/react-query` to ensure instant load times by intelligently reusing existing data.

- **Initial Data Seeding:** The `useFetchPost` hook is configured with an `initialData` function. Before making a network request, it scans the deeply nested `["posts"]` infinite query cache (from the Main Feed or Marketplace). If it finds a matching product ID, it instantly paints the UI with that cached data while fetching fresh data in the background.
- **Infinite Reviews:** The `ReviewsTab` uses `react-intersection-observer` to fetch the next cursor-based page of product reviews automatically when the user reaches the bottom of the tab.

API Endpoints & Data Structures

The product page interacts with the backend via several endpoints through the `authApi` Axios instance.

1. Fetch Product Details

Retrieves the complete data object for a specific product post.

- **Endpoint:** `GET /posts/post_retrieve/:postId`
- **Authorization:** Required (Bearer Token)
- **Response Structure (`ExtendPostType`):** Returns the standard `PostType` combined with an `is_cart` boolean to indicate if the user already has this item staged.

2. Fetch Product Reviews (Infinite)

Retrieves the paginated list of reviews associated with this specific product.

- **Endpoint:** `GET /posts/review/`

- **Query Parameters:**
- `productId`: The ID of the current product.
- `c`: Cursor string for pagination.
- **Authorization:** Required (Bearer Token)
- **Response Structure (`PaginatedReviews`):**

```
{
  "next": "url_string | null",
  "previous": "url_string | null",
  "results": [
    {
      "review_id": "string",
      "product": "string",
      "text": "string",
      "created_at": "ISO 8601 string",
      "rating": 5,
      "author": {
        "id": "string",
        "username": "string",
        "email": "string",
        "profile": { /* ReviewUserProfile */ }
      }
    }
  ]
}
```

3. Action Endpoints

- **Buy Now (Chat):** `POST /chat/confirm_chat/`
- **Add to Cart:** `POST /posts/cart/`

Post Page (Social Detail)

The `PostPage` is the detailed view for a standard social post. It focuses primarily on community engagement, displaying the original post body alongside a deeply nested, infinitely scrolling comment thread.

The layout is constrained to a 680px maximum width, mirroring the Main Feed's central column to maintain visual consistency.

Component Architecture

The interface handles complex, recursive data structures to render the conversation accurately.

1. PostPage (Main Wrapper)

- **Post Body:** Reuses the `PostComponentItems` component from the Main Feed, passing a `hideAction=true` prop to disable redundant navigation buttons.
- **Top-Level Comment Input:** Provides a sticky or embedded text input for users to reply directly to the main post. It utilizes a toast notification system (`showToast`) to provide immediate success/error feedback upon submission.

2. CommentItem & SingleComment (Recursive Tree)

Because comments can have replies, and those replies can have replies, the UI uses a recursive component pattern.

- **`CommentItem`:** The master container. It maps over the top-level comments and passes them to `SingleComment` with an initial `depth=0`.
- **`SingleComment`:** Renders the individual comment data.
- **Indentation:** Dynamically applies left margins based on its `depth` prop to visually represent the conversation thread.
- **Depth Limiting:** To prevent the UI from becoming unreadable on mobile devices, the "Reply" button is hidden if `depth >= 2`.
- **Inline Replies:** Toggling "Reply" opens an inline text input specifically tied to that comment's ID as the `parent`.
- **Recursion:** If a comment contains a `children` array, it maps over them and calls

`<SingleComment />` inside itself, incrementing the `depth` by 1.

State Management & Deep Optimistic Updates

Managing server state for deeply nested trees requires complex cache manipulation using `@tanstack/react-query`.

- **Optimistic Nested Likes (`useToggleCommentLike`):** When a user likes a comment, the mutation intercepts the cache. Because the target comment could be hidden three levels deep, the code executes a recursive `map` function over the `oldData.pages`, drilling down through `c1.children` -> `c2.children` -> `c3.children` to find the exact ID, flip its `liked` boolean, increment/decrement the `likes` integer, and update the UI instantly.
- **Cache Injection (`useCreateComment`):**
- **Scenario 1 (Top-Level):** If a user replies to the post, the new comment is injected into `newPages[0].results` at the very top.
- **Scenario 2 (Nested Reply):** If a user replies to another comment, a custom recursive function (`addCommentToTree`) scans every page and every child array to find the parent ID, injecting the new reply directly into the parent's `children` array so it appears immediately without a page refresh.

API Endpoints & Data Structures

The social post detail view interacts with the backend via the `authApi` Axios instance.

1. Fetch Comments (Infinite)

Retrieves the paginated list of top-level comments. Nested replies are included within the `children` array of each top-level comment.

- **Endpoint:** `GET /posts/comments/`
- **Query Parameters:**
- `postId`: The ID of the target post.
- `c`: Cursor string for pagination.
- **Authorization:** Required (Bearer Token)
- **Response Structure (`CommentItemReponseType`):**

```
```

```
{  "next": "url_string | null",  "previous": "url_string | null",  "results": [
```

2. Create Comment (Top-Level or Reply)

Posts a new comment.

- **Endpoint:** `POST /posts/comments/`
- **Payload (`PostCommentRequestType`):**

```
{
  "post": "post_id_string",
  "text": "string",
  "parent": "comment_id_string | null" // Null for top-level, ID for replies
}
```

- **Authorization:** Required (Bearer Token)

3. Toggle Comment Like

Optimistically toggles the like status of a specific comment.

- **Endpoint:** `POST /posts/comments/:commentId/toggle_comment_like/`
- **Authorization:** Required (Bearer Token)

Chat & Negotiation Page

The `ChatPage` is the core transactional interface of Socon-MKT. Because the platform relies on peer-to-peer agreements rather than automated checkout gateways, this page facilitates direct communication between buyers and sellers, tracks deal progress, and provides an off-platform Escrow safety net.

The page uses a maximum width of 1000px and features a responsive split-pane layout: a sticky context sidebar on desktop (collapsing to the top on mobile) and the primary messaging interface.

Component Architecture

The chat interface is divided into contextual metadata and the active message stream.

1. ChatPage (Layout & Context)

This wrapper component manages the layout and fetches the high-level metadata for the current chat session.

- **Product Context Card:** Displays the product being negotiated, its price, and a link to view the full listing.
- **Deal Progress Tracker:** A visual step-indicator mapped to the backend `order.status` (`NEGOTIATING`, `AGREED`, `DELIVERING`, `COMPLETED`).
- **Admin Escrow Integration:** A premium UI card that dynamically generates a pre-filled WhatsApp message. It encodes the product name, price, seller, and a direct URL, allowing users to quickly request Socon-MKT admin intervention to hold funds securely.

2. ChatMessages

A dedicated component handling the actual message feed to isolate intensive re-renders from the main layout.

- **Reverse Infinite Scroll:** Unlike feeds that scroll down for *older* content, chats load newest messages at the bottom. The `react-intersection-observer` sentinel is placed at the *top* of the container to fetch older pages when the user scrolls up.
- **Auto-Scroll to Bottom:** Uses a `useRef` dummy `div` at the bottom of the message list. On initial load, it forces the container to scroll down to the most recent message.

- **Read Receipts:** Mimics standard chat applications. Outgoing messages display a single gray tick (sent/unread) or a double blue tick (read by the recipient).

State Management & Optimistic UI

To provide a real-time chat experience without the overhead of WebSockets, the application leverages advanced `@tanstack/react-query` polling and optimistic updates.

- **Background Polling:**
- `useFetchChat`: Polls the chat metadata (like order status updates) every 10 seconds.
- `useFetchChatMessages`: Polls the message array every 5 seconds to instantly pull in new replies.
- **Optimistic Message Sending (`useCreateMessages`):** When a user hits send, the UI does not wait for a server response. The mutation cancels active fetches, snapshots the cache, and injects a mock `MessageItem` with a `temp_` ID and `is_own: true` into the top of the paginated array. The message appears instantly on screen. If the API call fails, the cache rolls back to the snapshot.
- **Auto-Read Mechanism (`useReadMessage`):** A `useEffect` hook actively monitors the incoming messages array. If it detects unread messages where `is_own` is false, it automatically fires a mutation in the background to mark those specific message IDs as read on the server.

API Endpoints & Data Structures

The chat interface relies on four primary endpoints via the `authApi` Axios instance.

1. Fetch Chat Metadata (Polling)

Retrieves the high-level details of the conversation, including the order status and associated product.

- **Endpoint:** `GET /chat/all_chats/:chatId/`
- **Authorization:** Required (Bearer Token)
- **Response Structure (`ChatItem`):**

```
{
  "chat_id": "string",
  "created_at": "ISO 8601 string",
```

```
  "updated_at": "ISO 8601 string",  "last_message_preview": "string",  "is_own":
```

2. Fetch Chat Messages (Infinite & Polling)

Retrieves the paginated list of messages for the active chat room.

- **Endpoint:** `GET /chat/messages/`
- **Query Parameters:**
- `chatId`: The ID of the current chat.
- `c`: Cursor string for pagination.
- **Authorization:** Required (Bearer Token)
- **Response Structure (`MessageListResponse`):**

```
{
  "next": "url_string | null",
  "previous": "url_string | null",
  "results": [
    {
      "message_id": "string",
      "text": "string",
      "is_read": false,
      "is_own": true
    }
  ]
}
```

3. Send Message

Posts a new message to the chat room.

- **Endpoint:** `POST /chat/messages/`
- **Payload (`SendMessagePayload`):**

```
{
  "text": "string",
  "is_read": false,
  "chat_id": "string | null",
  "order_id": "string"
}
```

- **Authorization:** Required (Bearer Token)

4. Mark Messages as Read

Sends an array of message IDs to the server to update their read status.

- **Endpoint:** `POST /chat/messages/read_message/`
- **Payload:**

```
{
  "messageId": ["message_id_1", "message_id_2"]
}
```

- **Authorization:** Required (Bearer Token)

Seller Profile Page

The `SellerProfilePage` is a dual-purpose interface on Socon-MKT. For regular users, it acts as a public storefront displaying a merchant's products, reviews, and business policies. For the merchant themselves, it conditionally unlocks private tabs to manage active orders and edit their business details.

The page utilizes a maximum width of 1280px and features a responsive grid layout, prioritizing a tabbed interface alongside a sticky metadata sidebar for desktop users.

Component Architecture

The profile aggregates several highly specialized tabs to organize merchant data effectively.

1. SellerProfileHeader

Renders the visual identity of the storefront.

- **Visuals:** Displays the cover image, business logo (which can be clicked to view in a full-screen lightbox), and verification badges.
- **Dynamic Actions:** Analyzes the `seller.is_own` boolean. If `false`, it displays the standard "Follow" button. If `true`, it displays an "Edit Business" button, which triggers the `EditSellerProfile` modal overlay.
- **Metrics:** Displays aggregated statistics including total followers, total sales, and the average star rating.

2. Tabbed Interface

The layout conditionally renders specific sub-components based on the active tab and the user's permissions:

- **`ProductsProfileTab`:** An infinitely scrolling grid of the seller's active product listings. Displays thumbnail previews, title, and formatted pricing.
- **`ReviewsTab`:** An infinitely scrolling list of verified buyer reviews. Includes a visually distinct summary card at the top displaying the average rating and total review count.
- **`SellerOrdersTab` (Private):** Rendered *only* if `seller.is_own` is true. This is the merchant's operational dashboard, displaying active negotiations and purchases.

- `SellersAbout`: A text-heavy tab detailing the merchant's biography, structured delivery zones/fees, payment methods, and standard terms and conditions.

3. Order Management & Alerts (SellerOrdersTab)

When the merchant is viewing their own orders, they are provided with administrative controls.

- **Status Updates:** Features a `<select>` dropdown mapped to the platform's standard Deal Steps (`NEGOTIATING`, `AGREED`, `DELIVERING`, `COMPLETED`). Changing this triggers a backend status update.
- **OrderAlertModal:** If a transaction goes sour (e.g., a buyer becomes unresponsive), the merchant can click the "🚨 Alert Admin" button. This opens a modal where they can submit a textual report tied directly to the `orderId` for the Socon-MKT moderation team to review.

State Management & Optimistic UI

Because merchants need their dashboard to feel instantaneous, the order management tab heavily utilizes cache manipulation via `@tanstack/react-query`.

- **Optimistic Order Updates (`useUpdateOrderStatus`):** When a merchant changes an order's status from the dropdown, the mutation immediately intercepts the cache. It iterates through the paginated `["seller_order"]` array, locates the specific `orderId`, and updates its status locally before the server responds. If the request fails, the cache rolls back to its previous state.
- **Infinite Scrolling:** The Products, Reviews, and Orders tabs all independently utilize the `react-intersection-observer` sentinel to fetch their respective cursor-based pages, keeping the initial payload lightweight.

API Endpoints & Data Structures

The profile utilizes multiple specific endpoints via the `authApi` Axios instance to segregate public data from private operational data.

1. Retrieve & Update Profile

Fetches the high-level storefront details and determines ownership permissions.

- **Endpoint:** `GET /users/retrieve_seller/:sellerId/`
- **Update Endpoint:** `PATCH /users/retrieve_seller/:sellerId/`
- **Authorization:** Required (Bearer Token)
- **Response Structure (`DetailedSellerProfile`):**

```
{
  "id": "string",
  "business_title": "string",
  "is_verified": true,
  "is_activated": true,
  "created_at": "ISO 8601 string",
  "is_following": false,
  "business_logo": "string_url | null",
  "payment": "string | null",
  "category": ["string"],
  "delivery": [
    {
      "area": "string",
      "fees": "string"
    }
  ],
  "about": "string | null",
  "terms": "string | null",
  "location": "string | null",
  "total_followers": 150,
  "total_sales": 45,
  "rating": "4.8",
  "is_own": false,
  "profile": { /* BaseProfile Object */ }
}
```

2. Fetch Seller Products (Infinite)

- **Endpoint:** `GET /posts/seller_product/`
- **Query Parameters:** `sellerId`, `c` (cursor)

3. Fetch Seller Reviews (Infinite)

- **Endpoint:** `GET /posts/review/`
- **Query Parameters:** `sellerId`, `c` (cursor)

4. Fetch & Update Seller Orders (Infinite - Private)

Retrieves the list of active orders. Only succeeds if the authenticated user is the merchant.

- **Endpoint:**

`GET /posts/seller_order/`

- **Update Endpoint:** `PATCH /posts/seller_order/:orderId/`
- **Payload (Patch):**

```
{
  "status": "NEGOTIATING | AGREED | DELIVERING | COMPLETED"
}
```

5. Alert Admin

Submits a moderation request for a specific order.

- **Endpoint:** `POST /posts/order_alert/`
- **Payload (`OrderAlertRequest`):**

```
{
  "order": "order_id_string",
  "text": "string"
}
```

User & Buyer Profile Page

The `BuyerProfilePage` acts as a dual-purpose interface within the Socon-MKT platform. When a user views their own profile, it serves as a private dashboard for managing orders, reviewing followed stores, and configuring account settings. When viewed by other members of the community, it acts as a public social footprint, displaying the user's personal posts, reviews, and profile details.

The page is constrained to an 800px maximum width, providing a focused, centralized view of the user's activity.

Component Architecture

The profile relies on a dynamic tabbed navigation system that conditionally renders management tools based on the `profile.is_own` boolean returned by the backend.

1. BuyerProfileHeader

Renders the user's public identity and context-sensitive administrative controls.

- **Identity Details:** Displays the user's avatar, `@username`, location (combining city and address), biography, and join date.
- **Share Functionality:** Integrates the browser's native Web Share API with a clipboard copy fallback (`navigator.clipboard.writeText`) to share profile URLs.
- **Action Buttons:**
- If `is_own` is true, renders an "Edit Profile" button that opens the `EditProfileModal` overlay for updating personal details, locations, and profile photos.
- If the user also holds a registered merchant account (`profile.seller_profile`), a "Business Profile" button routes directly to their storefront (`/sellers/:id`).

2. UserPostsTab

Displays an infinitely scrolling feed of the user's standard social posts (excluding marketplace listings).

- **Post Deletion Flow:** If the logged-in user owns the profile (`is_own`), each post card reveals a delete button (`FiTrash2`).
- **5-Second Undo Banner:** Clicking delete triggers a soft-deletion state. The post is immediately

hidden from view and a sticky top notification initiates a 5-second countdown timer. If the user clicks "Undo", the timer is cleared and the post is restored. If the countdown reaches zero (or the component unmounts), the `deletePostApi` mutation executes permanently.

3. OrdersTab (Private)

Visible strictly when `profile.is_own` is `true`.

- **Order History:** Lists all items the user has negotiated and purchased across the platform, complete with current negotiation and delivery status pills (`NEGOTIATING`, `AGREED`, `DELIVERING`, `COMPLETED`).
- **Review Submission:** Once an order reaches the `COMPLETED` state, a "Write Review" button triggers an interactive review modal allowing the buyer to rate the item (1 to 5 stars) and leave written feedback.

4. FollowingTab

Displays an infinitely scrolling list of all verified seller storefronts the user currently follows.

- **Soft-Unfollow with Undo:** Clicking "Following" enters an optimistic unlinking state. Rather than immediately removing the store from the database, the card displays an inline "Undo" prompt. A 5-second timeout delays the actual `toggleFollow` API call, allowing users to reverse accidental clicks seamlessly.

5. MyReviewsTab

Fetches and displays all reviews written by the profile owner across different marketplace products, featuring star ratings and direct links back to the reviewed listings.

State Management & Optimistic UI

The profile page heavily leverages `@tanstack/react-query` to maintain high interface responsiveness across nested tabs.

- **Optimistic Post Deletion (`useDeletePost`):** When a post deletion countdown completes, the mutation cancels active queries and snapshots both the global `["posts"]` feed and user-specific `["user_posts"]` query caches. It removes the post item from the local cache arrays immediately, only rolling back if the server returns an error.

- **Delayed Timeout Mutators:** Both post deletions and unfollow actions employ a timeout-deferred mutation pattern. Local component state immediately alters the interface, while background timeouts (`setTimeout`) manage the eventual server execution, preventing unnecessary network traffic if the action is undone.
- **Infinite Scrolling:** `UserPostsTab`, `OrdersTab`, `FollowingTab`, and `MyReviewsTab` each independently leverage `react-intersection-observer` sentinels to request the next page of results via cursor pagination (`c`).

API Endpoints & Data Structures

The profile system interacts with the following endpoints through the configured Axios `authApi` instance:

1. Fetch User Profile

Retrieves the target profile details, nested user record, and seller profile references.

- **Endpoint:** Configured via `useProfileFetch(profileId)`
- **Authorization:** Required (Bearer Token)
- **Response Structure (`UserProfileResponse`):**

```
{
  "user": {
    "id": "string",
    "username": "string",
    "email": "string"
  },
  "profile_id": "string",
  "profile_pic": "string_url | null",
  "created_at": "ISO 8601 string",
  "full_name": "string",
  "city": "string | null",
  "address": "string | null",
  "bio": "string | null",
  "website": "string_url | null",
  "is_own": true,
  "seller_profile": {
    "id": "string",
    "business_title": "string",
    "is_verified": true,
    "business_logo": "string_url | null"
  }
}
```

2. Fetch User Social Posts (Infinite)

Retrieves the paginated list of social posts created by a specific user.

- **Endpoint:** `GET /posts/user_posts/`
- **Query Parameters:**
- `userId`: The target profile's user ID.
- `is_product`: `false` (limits results to non-marketplace content).
- `c`: Cursor string for pagination.
- **Response Structure (`PaginatedUserPostsResponse`):**

```
{
  "next": "url_string | null",
  "previous": "url_string | null",
  "results": [
    {
      "id": "string",
      "caption": "string",
      "is_product": false,
      "is_own": true,
      "created_at": "ISO 8601 string",
      "media": [
        {
          "id": "string",
          "file": "string_url",
          "media_type": "photo | video",
          "thumbnail": "string_url | null"
        }
      ]
    }
  ]
}
```

3. Delete Post

Permanently removes a user post from the platform.

- **Endpoint:** `DELETE /posts/post_delete/:postId/`
- **Authorization:** Required (Bearer Token)

4. Fetch Buyer Orders (Infinite - Private)

Retrieves the purchasing history for the authenticated user.

-

Endpoint: `GET /posts/order/`

- **Query Parameters:** `c` (Cursor string for pagination)
- **Authorization:** Required (Bearer Token)
- **Response Structure (`PaginatedOrders`):**

```
{
  "next": "url_string | null",
  "previous": "url_string | null",
  "results": [
    {
      "id": "string",
      "status": "NEGOTIATING | AGREED | DELIVERING | COMPLETED",
      "created_at": "ISO 8601 string",
      "product_obj": {
        "id": "string",
        "title": "string",
        "price": "string",
        "media": [
          {
            "file": "string_url",
            "thumbnail": "string_url | null"
          }
        ],
        "author": {
          "username": "string",
          "profile": {
            "seller_profile": {
              "id": "string",
              "business_title": "string"
            }
          }
        }
      }
    }
  ]
}
```

5. Submit Product Review

Allows buyers to publish ratings and feedback for purchased products.

- **Endpoint:** `POST /posts/review/`
- **Payload (`SellerReviewRequest`):**

```
{
  "product": "product_id_string",
  "text": "Review text content",
  "rating": 5
```

```
}
```

- **Authorization:** Required (Bearer Token)

6. Fetch Following Storefronts (Infinite)

Retrieves the merchants followed by the active user.

- **Endpoint:** `GET /users/sellers_following/`
- **Query Parameters:** `c` or pagination link string
- **Authorization:** Required (Bearer Token)

Backend Administration

The Socon-MKT administration panel provides a secure, centralized dashboard for managing platform data, moderating content, and resolving buyer-seller disputes. Built on top of the Django Admin interface and customized with a modern dark theme, it is strictly restricted to authorized personnel.

Below is a breakdown of the available modules and their specific responsibilities.

---

1. System & Identity Management

This section handles the foundational access and user records of the platform.

Core The Core application manages overarching platform security and user moderation.

- **Api keys:** Manages the active cryptographic keys required for the frontend React application to communicate securely with the Django backend.
- **Reports:** Acts as the moderation queue. When users click the "🚩 Report" button on a profile or product, the data is aggregated here for admin review.

Users Handles authentication, permissions, and account states. It provides access to base user credentials, standard consumer profiles, and verified seller profiles, allowing admins to manage verifications and account suspensions.

---

2. Content & Commerce (Post App)

The Post module is the backbone of the marketplace, containing all user-generated content and transactional data.

Content & Engagement

- **Posts & Post Medias:** Admins can oversee all social posts and product listings, alongside their associated image and video files.
- **Interactions:** Stores all community engagement data, allowing admins to track or moderate **Post comments**, **Comment likes**, **Post likes**, **Post saves**, and **Post reviews**.
- **Listing conditions:** Manages the global taxonomy of terms available to sellers (e.g., "Payment on delivery", "No refunds").

Commerce & Moderation

- **Carts & Orders:** Tracks active user shopping carts and the statuses of ongoing negotiations between buyers and sellers.
- **Order alerts:** The primary dispute resolution center. When a seller requests Admin Escrow or flags an unresponsive buyer, the alert is logged here for immediate admin intervention.

---

3. Communication & Alerts

This section manages the peer-to-peer and system-to-user communication pipelines.

Chat Contains the **Chats** (individual negotiation rooms) and the raw **Chat_messages**. To maintain user privacy, administrators typically only access these tables when investigating an active Order Alert or a reported scam.

Notifications A centralized log of all system-generated and user-triggered alerts (such as likes, follows, or new messages) delivered across the platform.

Server Setup

The infrastructure for Socon-MKT is split between traditional server hosting for the backend and modern serverless delivery for the frontend.

The backend is hosted on a **DigitalOcean** droplet, utilizing a custom subdomain managed via Cloudflare DNS. The frontend SPA (React/Vite) is hosted on **Cloudflare Pages**, which automatically builds and deploys directly from the GitHub repository.

---

1. Frontend Deployment (Cloudflare Pages)

The frontend requires zero server maintenance. Deployment is handled automatically via Cloudflare Pages:

1. Connect the Cloudflare dashboard to the `sokoni-mkt` GitHub repository.
2. Set the build command to `npm run build` and the output directory to `dist`.
3. Cloudflare automatically provisions the SSL certificate and deploys the build globally to their CDN edge network.

---

2. Backend Server Initialization (DigitalOcean)

When provisioning a fresh DigitalOcean droplet, the first step is to create a dedicated user (e.g., `socon`) and update the system packages.

Update and install system dependencies:

```
sudo apt update && sudo apt upgrade -y
sudo apt install python3-pip python3-venv python3-dev libpq-dev postgresql postgresc
```

Web Server Configuration (Nginx)

Nginx acts as the primary web server and reverse proxy for the backend. It sits in front of Gunicorn, fielding HTTP requests, serving static assets, routing traffic to the Python application via a Unix socket, and managing communication with Cloudflare.

Nginx Server Block

Because the main domain is purchased and managed via Cloudflare, the backend is routed through a specific subdomain (`mkt-api.socon-mkt.com`). Cloudflare handles the edge SSL termination, so Nginx is configured to correctly parse the forwarded headers.

Create the Nginx configuration file:

```
sudo nano /etc/nginx/sites-available/socon_mkt
```

Application Server Infrastructure

To ensure Socon-MKT remains highly available, secure, and capable of handling concurrent user traffic, the production environment utilizes a robust, multi-layered application server architecture.

While local development environments use lightweight built-in servers, the live production platform relies on an enterprise-grade combination of **Gunicorn** and **Nginx**.

---

1. The Role of Gunicorn

Socon-MKT uses Gunicorn as its primary WSGI (Web Server Gateway Interface) HTTP server to power the Python backend.

- **Concurrency:** Gunicorn acts as the robust interface between the web server and the Django application. It manages dedicated worker processes to handle simultaneous client API requests efficiently without bottlenecking.
- **Process Management:** The Gunicorn service is strictly managed by the host operating system's internal daemon (systemd). This ensures that the application server runs continuously in the background and automatically recovers in the rare event of a system reboot or failure.

2. Web Server & Reverse Proxy (Nginx)

Operating directly in front of Gunicorn is **Nginx**, an industry-standard, high-performance web server acting as a reverse proxy.

- **Traffic Routing:** Nginx receives all incoming HTTP/HTTPS requests from the Cloudflare edge network. It smartly routes dynamic API calls to the Gunicorn workers while independently handling other tasks to maximize performance.
- **Security Isolation:** By acting as a proxy, Nginx masks the internal application ports. The Gunicorn application server is never directly exposed to the public internet, adding a critical layer of infrastructure security.

3. Application State & Deployment

During the deployment cycle, the backend undergoes automated preparation to ensure the platform operates smoothly before interacting with user traffic:

- **Database Synchronization:** The application's data models (migrations) are securely synced with the PostgreSQL database to ensure data integrity for all active features.
- **Static Asset Delivery:** All foundational platform assets (stylesheets, scripts, and interface elements) are securely collected and compressed. This allows Nginx and the Cloudflare CDN to deliver the user interface globally with minimal latency.

Security & DNS

Securing Socon-MKT requires a multi-layered approach that protects both the infrastructure layer at DigitalOcean/Cloudflare and the application layer within Django.

---

1. Application-Level Security

- **API Key Protection:** Every REST API endpoint on the backend is strictly protected. Client requests must include a valid API key in the headers to interact with the database. Unauthorized requests are immediately rejected by DRF permission classes before hitting the core logic.
- **Admin Panel Isolation:** The default Django Admin panel (`/admin`) is locked down and restricted to authorized superusers only. Standard users cannot access this portal under any circumstances.

2. Infrastructure Security

- **Cloudflare DNS Proxy:** The backend subdomain (`mkt-api.socon-mkt.com`) is proxied through Cloudflare. This hides the actual IP address of the DigitalOcean droplet, preventing direct DDoS attacks on the origin server.
- **Header Sanitization:** As seen in the Nginx configuration, we explicitly force `X-Forwarded-Proto https` and clear out contradictory SSL headers. This ensures Gunicorn processes requests securely and prevents header spoofing attacks.
- **UFW Firewall (DigitalOcean):** The server's internal firewall is configured to block all ports except those explicitly required by Nginx and SSH.

```
sudo ufw allow 'Nginx Full'
sudo ufw allow OpenSSH
sudo ufw enable
```