
# Bil482 - Software Design Document: Wi$h Li$t

## Contributors:

Akif Emre Reis
Alaaddin Gürsoy
Elif Serra Öncü
Gizem Elif Bayar

Department of Computer Engineering
TOBB University of Economics & Technology

Spring 2026

# 1. System Overview

The Category Creation and Management module operates within the Wishlist and Price Tracking Application as a foundational organizational layer. It enables users to create, edit, delete, and retrieve product categories for organizing wishlist items. This module ensures that products are grouped logically, improving usability and data management.

The module is implemented as a web-based client–server interaction where:

- The frontend (React) provides user interaction.

- The backend (FastAPI with Python) handles business logic.
- Supabase (PostgreSQL) stores category data.

# 2. System Context

**Primary actor:** User

**Supporting Systems:**
- Supabase
- FastAPI Backend
- React Frontend

**Context Description:**
The User interacts with the React UI → React sends HTTP requests to FastAPI → FastAPI validates and processes data → Supabase stores/retrieves category data.

# 3. Key Features and Functionality

The system shall:

1. Create a new custom category.
2. Create default categories (Cosmetics, Clothing, Technology and Kitchen).
3. Prevent duplicate category names per user.
4. Edit existing categories.
5. Delete custom categories.
6. Retrieve and display categories grouped by user.
7. Associate products with selected categories (future integration).

# 4. Assumptions and Dependencies

**Assumptions:**
- Each category belongs to a single user.
- Category names must be unique per user.
- User authentication is not yet implemented (user_id passed manually via query variables).

**Dependencies:**
- Supabase PostgreSQL availability.
- Backend API availability.
- Internet connection.

# 5. Architectural Design

### 5.1 System Architecture Diagram (High-Level)

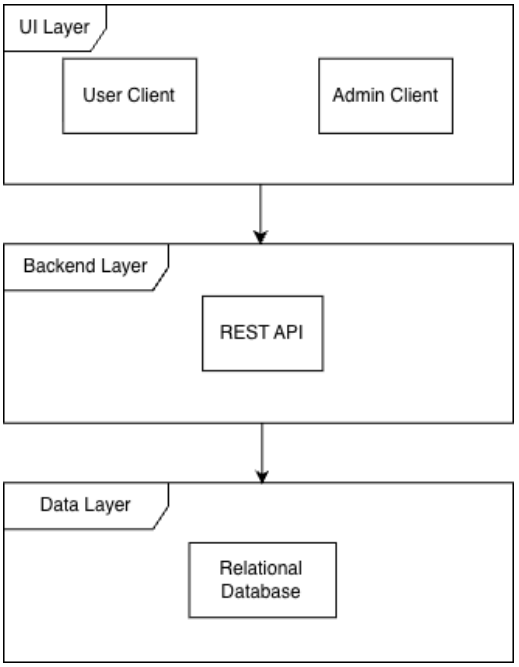



*Figure 1. High Level Design*

**Frontend:** React
**Backend:** FastAPI (Python)
**Data Layer:** Supabase PostgreSQL

### 5.2 Architectural Patterns and Styles

The system follows:

#### 1. Client–Server Architecture

Frontend and backend are separated.

#### 2. Layered Architecture (Backend)

The backend is structured into layers:

- Presentation Layer (API Routers)
- Service Layer (Business Logic)
- Data Access Layer (Repository Layer and Supabase Client)

#### 3. RESTful API Design

Stateless communication using HTTP methods (implemented using FastAPI).

### 5.3 Rationale for Architectural Decisions

- The layered design improves maintainability.
- REST API allows easy future mobile integration.
- Supabase reduces infrastructure complexity.

# 6. Component Design

## 6.1 Subsystem and Modules

**Backend Components:**

1. Routers (APIs)
2. Services
3. Repositories
4. Supabase Client Module

**Frontend Components:**

1. Category List Component
2. Category Form Component

## 6.2 Responsibilities of Each Component

**Routers**

- Defines API endpoints.
- Handles HTTP requests.
- Returns HTTP responses.
- Validates incoming request data.
- Forwards request to service layer.

**Services**

- Contains business logic.
- Applies application rules.
- Coordinates between repositories and other components.

**Repositories**

- Handles database operations.
- Communicates directly with the Supabase Client.

**Supabase Client**

- Provides connection to Supabase

## 6.3 Interfaces Between Components

Internal backend components communicate via direct method invocations following a layered architecture approach.

## 6.4 Component Diagrams

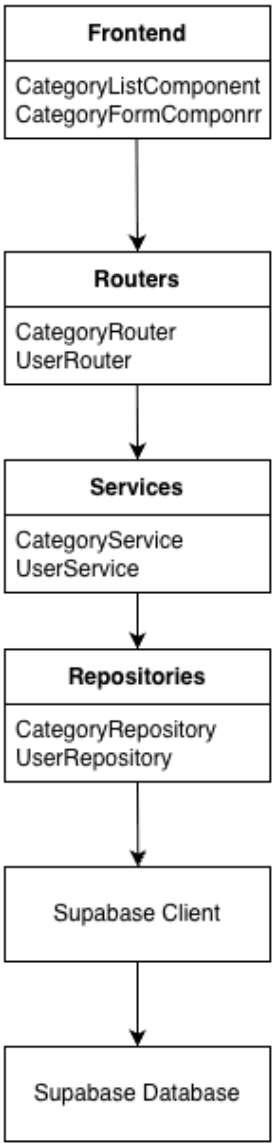



*Figure 2. Component Diagram*

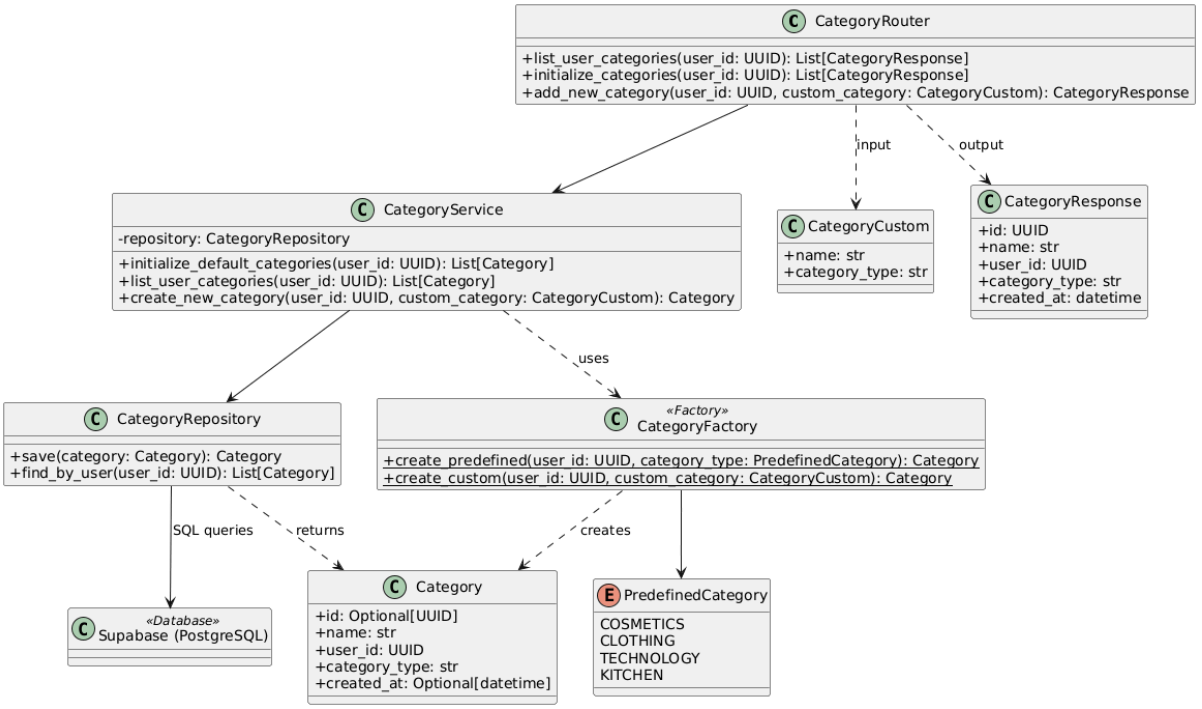



*Figure3. Class Diagram - Category Creation and Management*

# 7. Data Design

### 7.1 Data Model / ER Diagram

**Data Model:**

**User**
Represents a user of the wishlist system.

Attributes:

- `id` (primary_key, UUID)
- `first_name`
- `last_name`
- `email` (unique)
- `created_at`

Constraints:

- `email` must be unique
- `id` is automatically generated (`uuid_generate_v4()`)

**Category**

Represents a category belonging to a specific user.

Attributes:

- `id` (primary_key, UUID)
- `name`
- `category_type`
- `user_id` (foreign_key)
- `created_at`

Constraints:

- `(user_id, category_type)` must be unique
- `user_id` references `users.id`
- Cascade delete: deleting a user deletes their categories

- One User can have multiple Categories
- Each Category belongs to exactly one User

**ER Diagram**


*Figure 3. ER Diagram*

Constraints:

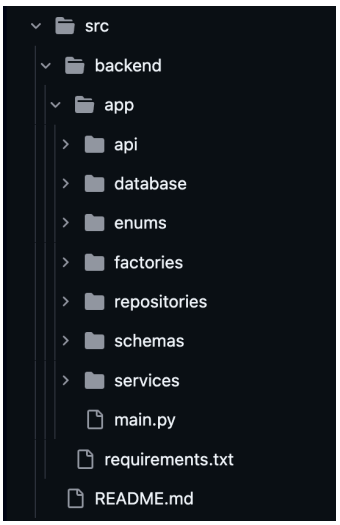
- UNIQUE(user_id, category_type)
- ON DELETE CASCADE

**7.2 Data Storage**

**Database:**

- Supabase (PostgreSQL)
- Relational model
- Migrations handle schema evolution

**File Structure (not finalized):**




### 7.3 Data Flow Diagrams

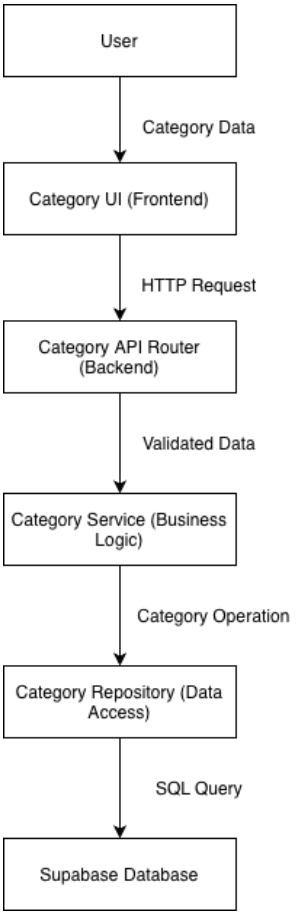



*Figure 4. Data Flow Diagram*

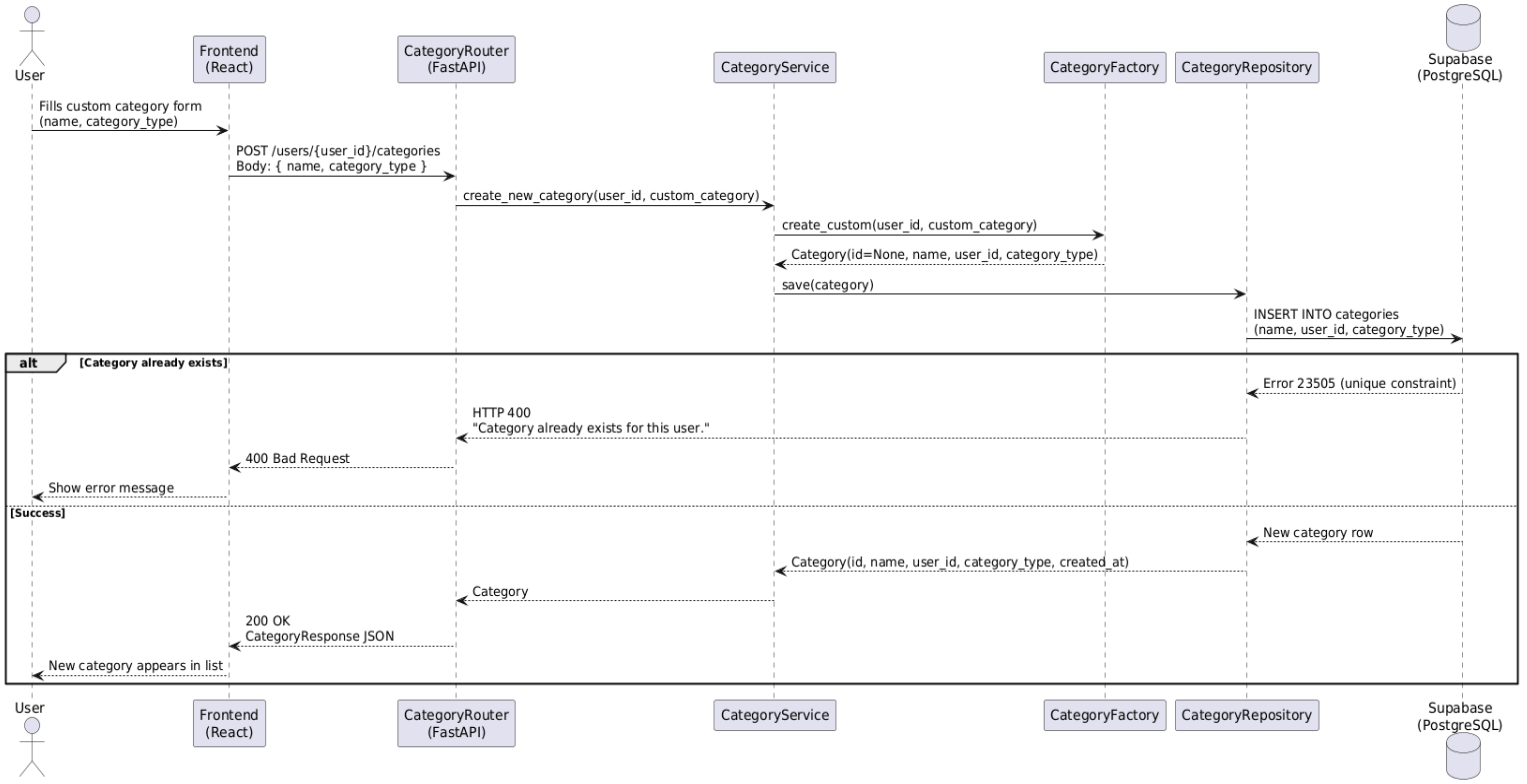



*Figure 5. Sequence Diagram – Create Custom Category*

### 7.4 Data Validation Rules

Input validation is performed using *Pydantic schemas*. Incoming request bodies are validated for type correctness and format constraints before being processed by the service layer. Invalid requests shall automatically return HTTP errors.

For the predefined categories, the category types are restricted via *enumeration definitions*. These definitions are used when initializing default categories for new users, ensuring consistency.

Certain validation rules are enforced directly at the database level using relational constraints :

- Unique Constraint: Each user cannot have multiple categories with the same name ensuring per-user uniqueness.
- Foreign Key Constraint: Each *category* references a valid user (*user_id* is a foreign key to the *users* table).
- Cascade Delete: When a user is deleted, all associated categories are automatically deleted, maintaining referential integrity.

# 8. Design Patterns

This section describes the design pattern applied within the Category Creation and Management use case. The Factory pattern was selected to handle the distinction between predefined (default) categories and user-defined custom categories in a clean and extensible way.

## 8.1 Applied Design Pattern

The Factory pattern is applied in the Category Creation and Management module. When a user initializes their account, the system automatically creates a set of default categories (Cosmetics, Clothing, Technology, Kitchen). When a user creates a custom category, a different type of category object is instantiated. The CategoryFactory class encapsulates this decision so that the service layer never needs to distinguish between the two types directly.

## 8.2 Context and Justification

### Context

The system must support two kinds of categories (SRS Section 5.1): predefined categories that are created automatically for every new user, and custom categories that are created on demand. These two types share the same data structure but differ in how they are initialized — predefined categories have a fixed name and an is_default flag set to true, while custom categories take a user-supplied name and have is_default set to false. Without a factory, this branching logic would need to be duplicated across the service layer every time a category object is created.

### How it is applied

A CategoryFactory class exposes two static methods: create_default() and create_custom(). Both return a Category object ready to be passed to the repository layer. The service layer calls the appropriate factory method and never constructs Category objects directly.

### Class structure:

```
class Category:
       def __init__(self, user_id: UUID, name: str, is_default: bool):
       self.user_id = user_id
       self.name = name
       self.is_default = is_default
```

```
class CategoryFactory:
	DEFAULT_CATEGORIES = ['Cosmetics', 'Clothing', 'Technology',
'Kitchen']

	@staticmethod
	def create_default(user_id: UUID) -> list[Category]:
	return [
		Category(user_id=user_id, name=name, is_default=True)
		for name in CategoryFactory.DEFAULT_CATEGORIES
	]

	@staticmethod
	def create_custom(user_id: UUID, name: str) -> Category:
	return Category(user_id=user_id, name=name, is_default=False)
```

**Usage in the service layer:**

```
class CategoryService:
	def initialize_defaults(self, user_id: UUID) -> None:
	categories = CategoryFactory.create_default(user_id)
	for cat in categories:
		self.repository.save(cat)

	def create_custom(self, user_id: UUID, name: str) -> Category:
	category = CategoryFactory.create_custom(user_id, name)
	return self.repository.save(category)
```

**Justification**

- The service layer is decoupled from object construction — it never sets is_default manually or hard-codes category names.
- Adding a new default category in the future requires changing only the DEFAULT_CATEGORIES list inside the factory, with no impact on the service or repository layers.
- Directly supports SRS Section 5.1: 'The system shall allow users to create a new custom category' and 'The system shall create default categories (Cosmetics, Clothing, Technology, Kitchen).'
- Custom categories and default categories are prevented from being confused — the factory is the single place where the is_default flag is set.

# 9. Implementation Notes

This section documents the key implementation decisions, project structure, coding conventions, and development guidelines for the Wi$h Li$t application. It serves as a reference for developers working on any part of the system.

### 9.1 Technology Stack

| Layer | Technology / Tool |
|---|---|
| Frontend | React (JavaScript) |
| Backend | FastAPI (Python) |
| Database & BaaS | Supabase (PostgreSQL) |
| Web Scraping | BeautifulSoup, httpx / requests, w |
| Task Scheduling | APScheduler (background price checks) |
| Version Control | Git |
| Development IDE | VS Code |
| API Communication | REST over HTTPS, JSON payloads |

### 9.2 Project Directory Structure

The repository is divided into two top-level directories: frontend/ for the React application and backend/ for the FastAPI service.

**Backend (FastAPI):**

```
backend/
  app/
       main.py             # FastAPI app entry point
       routers/            # API route definitions (categories, products,
notifications)
       services/           # Business logic layer
       repositories/       # Database access layer (Supabase queries)
       models/             # Pydantic request/response schemas
       scrapers/           # Scraper classes + ScraperFactory
       strategies/         # ExtractionStrategy implementations
       scheduler/          # APScheduler setup and price-check jobs
       core/
       config.py           # Environment variables (Supabase URL, keys)
       supabase_client.py # Supabase client singleton
  requirements.txt
  .env                     # Local environment variables (not committed)
```

**Frontend (React):**

```
frontend/
  src/
       components/         # Reusable UI components
       CategoryList/
       CategoryForm/
       ProductCard/
       ProductForm/
       PriceHistoryChart/
       NotificationPanel/
       pages/              # Top-level route pages
       Dashboard.jsx
       ProductDetail.jsx
       Archive.jsx
       services/           # API call functions (axios / fetch wrappers)
       categoryService.js
       productService.js
       notificationService.js
       context/            # React Context for global state (auth,
categories)
       App.jsx
       index.jsx
  public/
  package.json
```

## 9.3 Backend Implementation Guidelines

### 9.3.1 Layered Architecture

The backend strictly follows a three-layer architecture. No layer should skip the one above or below it:

- Router layer — handles HTTP request/response only. No business logic.
- Service layer — contains all business rules. Calls repositories; never calls Supabase directly.
- Repository layer — contains all Supabase/PostgreSQL queries. Returns plain Python objects or dicts.

### 9.3.2 Pydantic Schemas

All incoming request bodies and outgoing responses are validated using Pydantic models defined in app/schemas/. This ensures type safety and produces automatic OpenAPI (Swagger) documentation. Validation errors are automatically converted to HTTP 422 responses by FastAPI.

**Example — Category schema:**

```
class Category(BaseModel):
    name: str
    user_id: UUID
    category_type: str
    created_at: datetime | None = None

class CategoryCustom(BaseModel):
    name:str
    category_type: str
```

### 9.3.3 Scraper and Strategy Implementation

All scraper classes reside in app/scrapers/. Each scraper accepts an ExtractionStrategy at construction time (see Section 8.2.1 and 8.2.2). The ScraperFactory.create(url) method is the single entry point for obtaining a scraper instance — it should never be bypassed.

- Scrapers must handle network errors (timeouts, connection refused) and raise a ScrapingError.
- The scheduler catches ScrapingError, logs the failure, and skips updating the price for that cycle.
- Manual price entry is always available as a fallback (SRS Section 5.2).

#### 9.3.4 Background Scheduler

APScheduler runs inside the FastAPI process and triggers price-check jobs based on each product's configured frequency (hourly / daily / weekly). Key implementation notes:

- Jobs are stored in memory; on restart, active products are re-scheduled from the database.
- Each job calls PriceMonitor.check_price(product) for a single product to keep jobs independent.
- Job execution must not block the FastAPI event loop — use BackgroundTasks or run_in_executor for long-running scraping calls.

### 9.4 Frontend Implementation Guidelines

#### 9.4.1 Component Structure

The React frontend follows a component-based architecture. Components are split into two categories:

- Page components (src/pages/) — correspond to routes; responsible for fetching data and passing it to child components.
- UI components (src/components/) — stateless or locally-stateful; receive data via props; handle only presentation logic.

#### 9.4.2 API Communication

All HTTP calls to the FastAPI backend are centralized in src/services/. Components never call fetch() or axios directly — they import from the service module. This makes it straightforward to swap the base URL between development and production environments.

#### 9.4.3 State Management

React Context is used for lightweight global state (authenticated user, category list). Component-level state (useState / useReducer) is preferred for local UI state such as form inputs and modal visibility. A dedicated state management library (Redux, Zustand) is not required for the current scope.

#### 9.4.4 Price History Chart

The price history visualization on the Product Detail screen is implemented using a charting library (e.g., Recharts or Chart.js). Price history data is fetched from the backend as a list of { timestamp, price } records and passed directly to the chart component. The chart renders a line graph showing price over time.

### 9.5 Database Conventions

- All primary keys use UUID type generated by Supabase (gen_random_uuid()).
- Timestamps (created_at, updated_at) use TIMESTAMPTZ and default to now().
- Foreign key constraints and cascade deletes are enforced at the database level.
- The unique constraint on (user_id, category_name) is enforced at the database level — not only in application code.
- Database migrations are managed as versioned SQL files; no schema changes are applied manually via the Supabase UI in production.

### 9.6 Environment Variables

Sensitive configuration is stored in .env files and never committed to the repository. The following variables are required:

### 9.7 Coding Conventions

**Backend (Python):**

- Follow PEP 8. Use type hints on all function signatures.
- Use snake_case for variables and function names; PascalCase for class names.
- All public functions in the service and repository layers must have a docstring.
- Raise domain-specific exceptions (e.g., CategoryNotFoundError, DuplicateCategoryError) rather than generic exceptions.

**Frontend (React / JavaScript):**

- Use functional components with hooks exclusively — no class components.
- Use camelCase for variables and function names; PascalCase for component names.
- Each component file exports a single default component.

- Avoid inline styles; use CSS modules or a utility-first CSS framework.

### 9.8 Known Limitations and Constraints

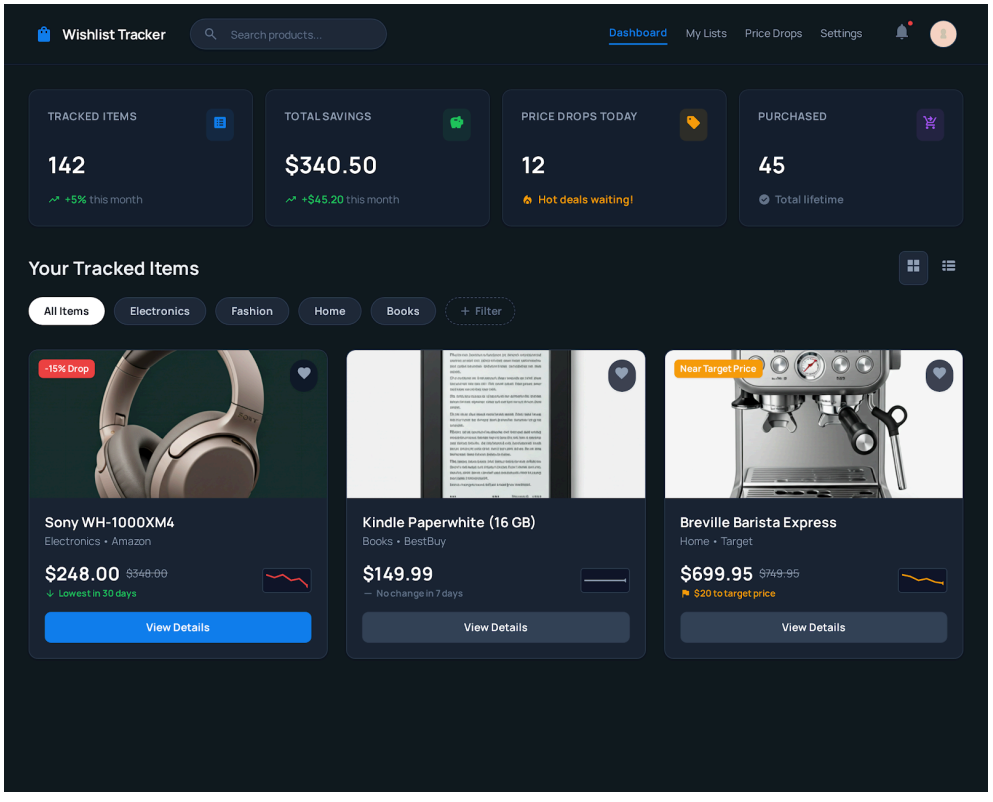
- User authentication is not yet fully implemented; user_id is currently passed manually via query parameters during development.
- Scraping availability cannot be guaranteed for all e-commerce websites due to anti-bot mechanisms and DOM structure changes (SRS Section 4.3).
- The scheduler runs in-process with FastAPI; for production scale, a dedicated task queue (e.g., Celery with Redis) would be preferable.
- Price checks are limited by Supabase free-tier rate limits and external website anti-scraping protections.

# 10. User Interface Design

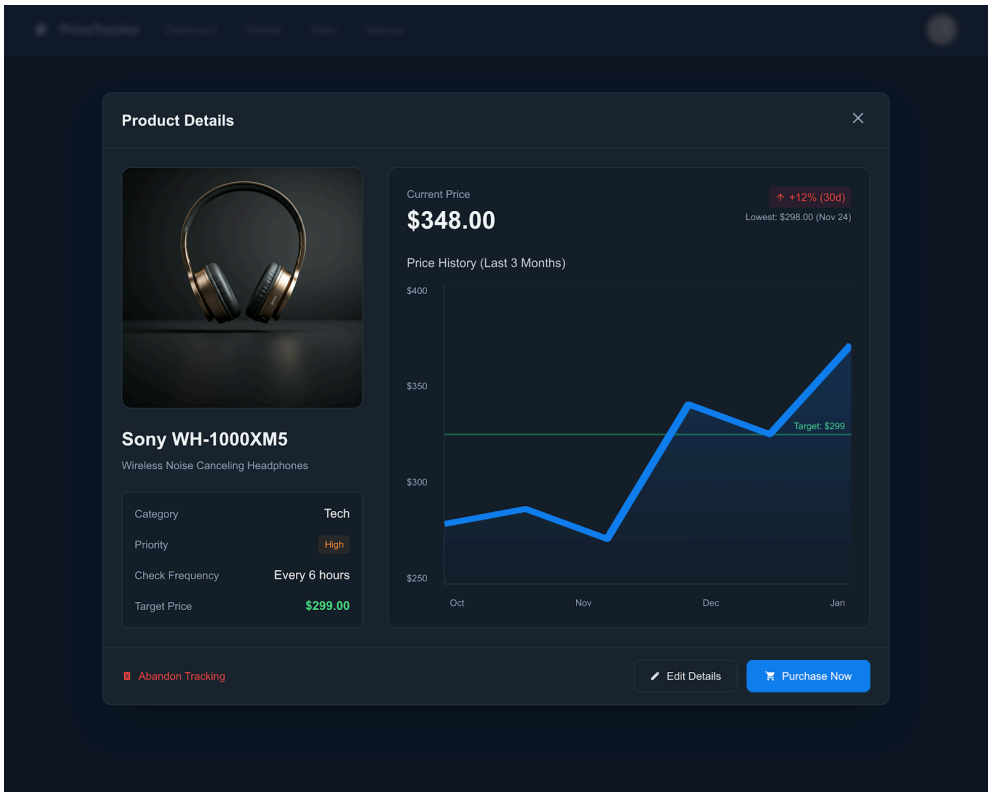
### 10.1 UI Mockups or Wireframes

The following visuals represent draft concepts and are not finalized.

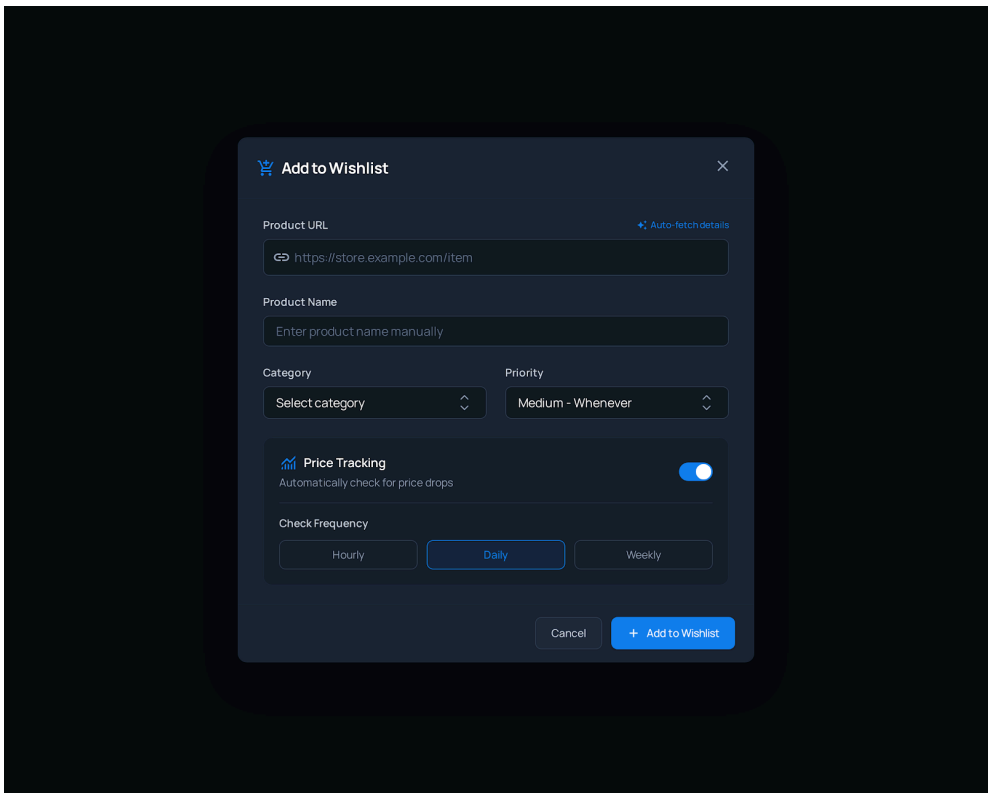
- Wishlist Dashboard — stat cards (tracked items, savings, price drops, purchased), category filter pills, and a 3-column product grid with sparkline charts



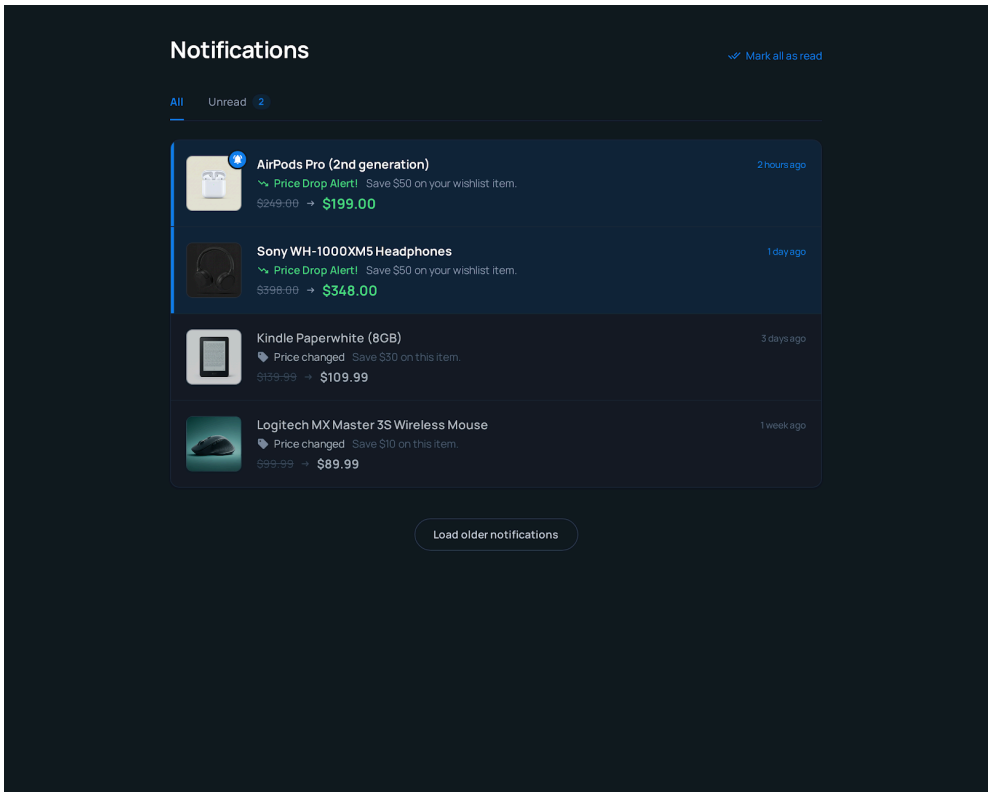
- Product Detail Modal — click any product card to see a full price history chart, metadata (category, priority, check frequency), and action buttons (purchase / edit / abandon)



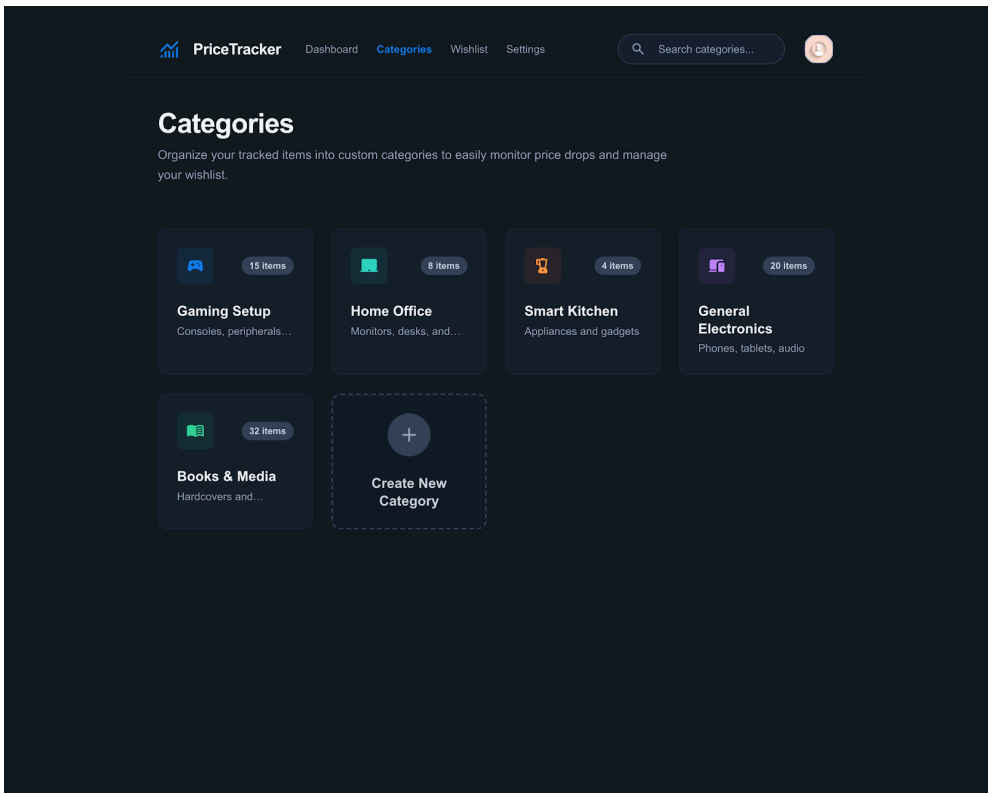

- Add Product Modal — full form with URL input, category & priority dropdowns, frequency selector, and an auto-tracking toggle




- Notifications — unread/read states with product name, old→new price, and timestamps




- Categories — visual category management grid




# 11. External Interfaces

This section explains how external systems interact with the module.

## 11.1 APIs

GET /users/{user_id}/categories — Kategorileri Listele

| Parametre | Yer | Tip | Zorunlu |
|---|---|---|---|
| user_id | Path | UUID | ✓ |

Response 200:

```
[
  {
    "id": "550e8400-e29b-41d4-a716-446655440000",
    "name": "Cosmetics",
    "user_id": "123e4567-e89b-12d3-a456-426614174000",
    "category_type": "COSMETICS",
    "created_at": "2026-03-29T10:00:00Z"
  },
  ...
]
```

> Not: Kullanıcının hiç kategorisi yoksa 4 predefined kategori otomatik oluşturulur ve döner.

---
POST /users/{user_id}/categories/init — Varsayılan Kategorileri Başlat

| Parametre | Yer | Tip | Zorunlu |
|---|---|---|---|
| user_id | Path | UUID | ✓ |

Response 200: List[CategoryResponse] — 4 predefined kategori (Cosmetics, Clothing, Technology, Kitchen Utensils)

---
POST /users/{user_id}/categories — Özel Kategori Oluştur

| Parametre | Yer | Tip | Zorunlu |
|---|---|---|---|
| user_id | Path | UUID | ✓ |
| name | Body | string | ✓ |
| category_type | Body | string | ✓ |

Request Body:

```
{
  "name": "Sports",
  "category_type": "SPORTS"
}
```


Response 200:

```
{
  "id": "7c9e6679-7425-40de-944b-e07fc1f90ae7",
  "name": "Sports",
  "user_id": "123e4567-e89b-12d3-a456-426614174000",
  "category_type": "SPORTS",
  "created_at": "2026-03-29T10:15:00Z"
}
```


Response 400:

```
{
  "detail": "Category already exists for this user."
}
```

### 11.2 Third-party Systems

The category module interacts with Supabase:

Supabase provides the PostgreSQL database used to store category and user data. The backend communicates with Supabase through the official Python client library. Supabase handles data persistence, authentication keys, and database connection management.

## 12. Performance Considerations

This section defines the performance targets of the system and describes the strategies used to meet them under the constraints of an academic project deployed on free-tier infrastructure.

### 12.1 Performance Requirements

| Requirement | Target | Notes |
|---|---|---|
| API response time | < 2 seconds (p95) | Under normal single-user load; defined in SRS Section 7.2 |
| Price-check job execution | < 10 seconds per product | Includes HTTP fetch + HTML parse + DB write |
| Frontend initial load | < 3 seconds | Dashboard with up to 50 products |
| Notification delivery latency | < 30 seconds after price change detected | In-app notification; depends on scheduler interval |
| Database query time | < 500 ms | All Supabase queries; enforced via indexed columns |

| | | |
|---|---|---|
| Concurrent users (MVP) | Up to 10 simultaneous users | Academic/demo scope; Supabase free tier limit |

## 12.2 Scalability and Optimization Strategies

### 12.2.1 Database Indexing

The following columns are indexed in the PostgreSQL schema to ensure fast queries as data grows:

- categories.user_id — queries filtering categories by user.
- products.user_id, products.category_id, products.state — filtering wishlist/archive views.
- price_history.product_id — fetching price history for a given product.
- notifications.user_id, notifications.is_read — unread notification queries.

### 12.2.2 Scheduler Design for Efficiency

Rather than running a single job that checks all products at once, each product has its own independent scheduled job based on its configured frequency. This distributes the scraping load over time and prevents a single large batch from overwhelming external websites or the Supabase write quota.

- Hourly products: checked every 60 minutes, staggered by product creation time.
- Daily products: checked once every 24 hours.
- Weekly products: checked once every 7 days.

### 12.2.3 Scraping Request Throttling

To avoid triggering anti-bot protections on external websites, the scraping layer applies the following constraints:

- Minimum 2-second delay between consecutive requests to the same domain.
- HTTP connection timeouts set to 10 seconds to prevent jobs hanging indefinitely.
- Failed requests are retried at the next scheduled interval — not immediately — to avoid hammering unavailable endpoints.

### 12.2.4 Frontend Performance

- The price history chart only fetches data when the Product Detail page is opened — not on the main dashboard.
- The dashboard loads products in a paginated or category-filtered view to avoid rendering a large list at once.
- API calls in the React frontend are debounced where relevant (e.g., search/filter inputs).

#### 12.2.5 Supabase Free-Tier Constraints

The system is designed to operate within Supabase's free-tier limits during the academic demo period. Key limits to be aware of:

- 500 MB database storage limit — price history records older than 90 days may be archived or deleted to stay within quota.
- API rate limiting — the backend batches database writes where possible (e.g., writing price history and updating current price in a single transaction).

# 13. Error Handling and Logging

## 13.1 Exception Management

The system implements structured exception handling in the service and repository layers to ensure that meaningful error messages are returned to the client.

Typical exceptions:

DuplicateCategoryError is raised when a user attempts to create a category with a name that already exists for that user.

CategoryNotFoundError is raised when a category ID does not exist during update or deletion operations.

DatabaseError is raised when a Supabase operation fails.

FastAPI automatically converts validation failures into HTTP 422 responses.

Service-layer exceptions are translated into appropriate HTTP responses.

## 13.2 Logging Mechanisms

Logging is implemented using the structured logging library Loguru. The system records important events such as category creation, duplicate category attempts, database errors, and unexpected exceptions. Logs include timestamps and relevant identifiers to facilitate debugging and monitoring.

# 14. Design for Testability

The category module is designed to be easily testable through clear separation of concerns.

Key testability features include:

**Layer isolation**
Routers, services, factories, and repositories are implemented as independent components that can be tested individually.

**Dependency injection**
Repositories can be replaced with mock implementations during unit testing to avoid real database interactions.

**Factory testing**
The CategoryFactory can be unit-tested independently to verify that predefined and custom categories are constructed correctly.

**API testing**
FastAPI provides a TestClient that allows automated testing of HTTP endpoints without running the full server.

These design choices enable unit tests, integration tests, and API tests to be written for the category module.

# 15. Deployment and Installation Design

## 15.1 Environment Configuration

The application requires several environment variables to connect to Supabase and configure runtime behavior.

The required variables are SUPABASE_URL (url for Supabase project API) and SUPABASE_KEY (Supabase service key) which are stored in the .env file during development and are loaded using the python-dotenv library.

## 15.2 Packaging and Dependencies

Backend dependencies are defined in `requirements.txt`

The backend server can be started using:
```
uvicorn app.main:app --reload
```

The frontend React application is installed and run using:
```
npm install
npm start
```

# 16. Change Log

The change log of the project can be found under the main repository of the project in the `CHANGELOG.md` file.

# 17. Project Plan

## 17.1 Product Backlog

PB1 – User can create a custom category
PB2 – System creates default categories for a new user
PB3 – User can edit a category
PB4 – User can delete a category
PB5 – User can view categories

PB6 – User can add a product
PB7 – User can assign category to product
PB8 – User can set priority for product
PB9 – User can update product

PB10 – System retrieves product price automatically
PB11 – System stores price history
PB12 – System updates prices periodically

PB13 – User can set price check frequency
PB14 – System sends notification when price changes
PB15 – User can view notifications

PB16 – System handles scraping errors
PB17 – System logs system events

## 17.2 Sprint Plan

Sprint 1 (Week 1-2)
- Category management (PB1–PB5)
- Basic API endpoints
- Database setup

Sprint 2 (Week 3-4)
- Product management (PB6–PB9)
- UI for product operations

Sprint 3 (Week 5-6)
- Price monitoring system (PB10–PB12)
- Scraper implementation
- Scheduler setup

Sprint 4 (Week 7-8)
- Notification system (PB13–PB15)

- Error handling & logging (PB16–PB17)
- Testing and bug fixing

# 18. Future Work and Open Issues

Several improvements are planned for category creation and management:

User Authentication Integration
The system currently receives the user_id manually. Future versions will integrate authentication so that the user identity is extracted from access tokens.

Category Deletion
The current implementation does not contain the category deletion feature yet. It will be implemented along with all other improvements.

Notification Service
The repository will be further updated to include the regular price-check/fetch and the notification service when there are updates.

# Task Matrix

| Tasks | Akif Emre Reis | Alaaddin Gürsoy | Elif Serra Öncü | Gizem Elif Bayar |
|---|---|---|---|---|
| **Project Perspective** | | |  |  |
| **Product Functions** |  |  | | |

| | | | | |
|---|---|---|---|---|
| **User Characteristics** |  |  | | |
| **Constraints** | | |  | |
| **System Features (Use Case Based)** | | |  |  |
| **Use Case Diagram** |  | | |  |
| **Non-Functional Requirements** | |  |  | |
| **External Interface Requirements** |  |  | | |
| **User Interfaces** | |  | | |

| | | | | |
|---|---|---|---|---|
| **End-to-end document review and conflict resolution** |  |  |  |  |