Hand Me Down Clothing Documentation

Table of Contents

1 Informative Part

1.1 Team

The team is organized into key functional areas with dedicated leads overseeing documentation and requirement completion, authentication, listings, and search and map integration, under the guidance of three project managers.

Managers

- Anthony Martinez
- Alma Piñeiro
- Jahsyel Rojas

Team 1 - Documentation & Requirements

- Joshua Dávila (Lead)
- Ojani Figueroa
- Giovanny García
- Juan Iranzo

Team 2 - Authentication & User Accounts

- Lorenzo Pérez (Lead)
- Jessy Andújar
- Gabriel Marrero
- Luis Marrero
- Ángel Villegas

Team 3 - Listings

- Kevin Gómez (Lead)
- Leanelys González
- Karina López
- Nicolás Rivera
- Jachikasielu Uwakweh

Team 4 - Search & Map Integration

- Jorge De León (Lead)
- Devlin Hahn

- Alejandro Marrero
- Kian Ramos
- Januel Torres

Team 5 - UI/UX & Branding

- Fabiola Torres (Lead)
- Yamilette Alemañy
- Daniella Melero
- Andrea Segarra
- Kenneth Sepúlveda

1.2.1 Current Situation

Landfills and textile waste in Puerto Rico

- High waste volume: Approximately 250 million pounds of clothing and textiles are sent to Puerto Rico's landfills annually.
- High landfill rate: Similar to the global and U.S. trends, a very high percentage of discarded textiles, around 85%, end up in landfills, despite being largely recyclable.
- Low recycling rates: Puerto Rico's overall recycling rate is notably low, with some reports estimating it to be less than 10%. This is significantly lower than the U.S. national average.
- Overwhelmed landfills: Puerto Rico's landfills are facing a serious crisis, with many already at or over capacity. The high volume of textile waste contributes to this problem.

Sources:

- investpr.org
- theenvironmentalblog.org

Poverty

- Overall Poverty: The poverty rate in Puerto Rico is alarmingly high, at around 41.7% as of 2022. This is over three times the U.S. national average.
- Child Poverty: A staggering 54.3% to 57.6% of children under 18 in Puerto Rico live in poverty. This is more than any U.S. state and indicates that a vast number of families cannot afford to consistently provide their children with properly fitting, weather-appropriate clothing and shoes.
- Persistent Poverty: All 78 municipalities in Puerto Rico are classified as "persistent poverty counties," meaning they have maintained a poverty rate of 20% or more for at least 30 years.

Sources:

- centropr.hunter.cuny.edu

- geopoliticaleconomy.com

Homelessness:

- Homeless Population: Recent counts show the homeless population in Puerto Rico to be around 2,096 individuals. For these individuals, clothing is a constant and critical need.

Sources:

- periodismoinvestigativo.com

1.2.2 Need

The purpose of this section is to establish the fundamental needs that motivate the Hand Me Down project, expressed independently of any system-to-be. These needs are grounded in the resale domain and must reflect the concerns of students and families who participate in secondhand exchanges. The articulation of these needs will guide the subsequent development of domain descriptions, requirements, software architecture, and testing activities.

Stakeholders in this domain shall be understood as students and families seeking opportunities for affordable, accessible, and trustworthy secondhand exchanges. Their needs are not for a platform itself, but for solutions to the problems they encounter when attempting to exchange goods in local communities.

The following distinct needs are identified:

- Students and families must have affordable access to secondhand goods that support daily life, education, and well-being.
- Stakeholders must be able to rely on transparent information about the condition and history of pre-owned items.
- Exchanges shall be conducted in a manner that establishes trust, fairness, and safety between participants.
- Opportunities for accessibility and inclusivity must be available so that all families and students, regardless of economic background, will participate in the resale domain without barriers.
- Developers shall have clear requirements, descriptions, and architecture to build upon, since no structured system currently exists to organize this resale context.

These needs form the foundation for further project work. They are deliberately expressed at the domain level, independently of any particular solution, to ensure that subsequent design and implementation activities will remain aligned with the stakeholders underlying motivations.

1.3.1 Scope & Span

Scope

The Hand Me Down project will operate in the broad domain of online resale marketplaces. It will

address the general problem of enabling individuals and communities to exchange secondhand goods in a structured, reliable, and sustainable manner. The scope will cover activities in domain engineering, requirements engineering, and software architecture to ensure a well-founded solution.

The project will emphasize the following areas:

- Domain:: Resale of pre-owned items across categories such as clothing and accesories.
- Requirements:: Identifying user needs related to affordability, sustainability, accessibility, and usability.
- Architecture:: Defining a framework that supports secure and scalable interactions between sellers and buyers.
- Project Activities:: Documentation, validation, and design processes that must accompany implementation.

Span

The span narrows the focus of the Hand Me Down project to **specific concerns and audiences** within the general resale domain. The platform must primarily serve individuals and families who wish to exchange items affordably, students and young adults seeking budget-friendly goods, and community members interested in sustainable consumption.

The span includes the following project-specific aspects:

- User Interaction:: Individuals must be able to list, browse, and search for secondhand items.
- Categorization:: Items will be organized into categories that facilitate discovery.
- Transaction Support:: The system must provide structured means for creating and viewing listings, including optional prices or donation markers. Negotiation and exchange are arranged outside the platform.
- Trust and Transparency:: Item conditions and relevant metadata must be clearly described to support informed decisions.

1.3.2 Synopsis

This Synopsis provides overview of the Hand Me Down project from the perspective of students and families engaged in secondhand exchanges. It articulates the domain, affordability, accessibility, trust, safety and states that stakeholders must be able to discover, evaluate, and exchange pre-owned goods with transparent information about item condition and history. The project shall be conducted through structured domain acquisition to produce a domain description, a requirements prescription that specifies goals, constraints, and quality attributes with traceability to stakeholder needs and a software architecture that evaluates alternatives and justifies decisions, prototyping where necessary to mitigate risk. Component design and iterative implementation will realize prioritized capabilities while preserving traceability. Verification and validation shall include a test plan that covers functional fitness, usability, and trust/safety concerns, supported by versioned documentation, change control, risk tracking, and metrics.

1.4 Other Activities than Just Developing Source Code

This project will not be limited to writing source code. To satisfy the needs identified for affordability, accessibility, trust/safety, and transparency in the Mayagüez/UPRM context, the team shall execute the following activities in addition to implementation. Each activity is mandatory, tied to a stakeholder need and justified by the current situation.

Domain engineering

The team shall elicit and model the domain of donation and resale of clothing and accessories for students and families (actors, workflows, vocabulary, constraints).

- **Need satisfied:** developers must share a precise vocabulary and mental model to preserve accessibility, affordability, and transparency.
- **Current situation:** rough sketches are based on team self-observation; no external interviews conducted yet; stakeholder roles are not enumerated; internal terminology seems consistent but remains unvalidated in the field.
- **Contributions to date:** condition labels and verification practices were researched; the **Sell** vs **Donate** category structure was standardized; initial personas were drafted to ground concepts.
- **Planned outcomes:** domain glossary, context diagram, concrete exchange workflows, and domain verification norms (tag photo, full-item photos, defect call-outs).

Requirements engineering

The team shall prescribe goals, user-level functional requirements (listing, browse/search, donation/resale flows, offer/negotiation) and quality attributes (trust/safety, transparency, usability, accessibility) with explicit traceability to needs and acceptance criteria.

- **Need satisfied:** developers must clearly understand system functionality and nonfunctional expectations that build trust and reduce effort.
- **Current situation:** no consolidated requirements baseline or acceptance criteria exist.
- **Contributions to date:** epics were linked to user stories (buyer/seller perspectives) against stakeholder needs; interface requirements at the system boundary were authored; measurable machine requirements (response time, uptime, user capacity) were drafted.
- **Planned outcomes:** requirements set with SHALL statements and acceptance criteria; traceability matrix (Need → Requirement → Test); nonfunctional thresholds made testable.

Software architecture

The team shall select and justify an architecture addressing security, privacy, modifiability, and campus-scale usage; architectural views and decision records will be maintained.

- **Need satisfied:** stakeholders must receive a reliable, maintainable basis for transparent, safe exchanges.
- **Current situation:** direction is leaning toward Supabase for authentication (Google sign-in compatibility) with a full custom backend; no UPRM hosting/privacy constraints identified; ADRs and C4 views are not yet written.
- **Contributions to date:** search and map integration approaches were evaluated and

geolocation/privacy considerations documented; authentication backends (Firebase vs. Supabase) and session-management implications were analyzed; page-level layouts were produced to inform view composition and navigation flows.

- **Planned outcomes:** C4 views (context/container), Architectural Decision Records (auth, data, map/search), quality-attribute scenarios, and targeted risk spikes where uncertainty is high.

Component design

The team shall define modules, interfaces, and data contracts to preserve testability and changeability (catalog/search, profiles/auth, exchange/offer, reporting/moderation, “items circulated” metrics).

- **Need satisfied:** maintainable, verifiable components are required to deliver transparency (clear item/condition data) and accessibility (predictable flows).
- **Current situation:** boundaries and interfaces are only partially documented.
- **Contributions to date:** the user/auth schema (roles, profile fields, donation/sell history) was initiated and APIs for login/registration/logout were defined; search behavior and map-related data interactions were documented; wireframes and page designs (Homepage, About, Clothes Listing, Individual Item, Favorites, Checkout, Log In/Sign Up, Profile) clarify interface responsibilities and data needs.
- **Planned outcomes:** module responsibilities, interface specs (inputs/outputs/preconditions/postconditions), initial schemas with migration notes, and example queries.

Implementation planning

The team will establish a delivery roadmap, Definition of Done, contribution standards, and a branching/CI strategy suitable for a student-run service.

- **Need satisfied:** predictable, reviewable progress must be ensured to realize stakeholder value without regressions.
- **Current situation:** a branch is created per team with controlled pull requests and merges to main when working; no CI pipeline is in place; review checklist/PR template usage is minimal.
- **Contributions to date:** the AsciiDoc documentation structure and conventions were established; documentation issues with acceptance criteria were created; a step-by-step Node.js/npm installation and verification guide was produced to bootstrap the development environment.
- **Planned outcomes:** roadmap with dates/owners, Definition of Done, CONTRIBUTING guidelines, PR template, and CI workflow for lint/tests on pull requests.

Testing and validation

The team shall produce a test plan spanning unit, integration, end-to-end, and usability/acceptance with students and families; trust/safety validations shall be included (e.g., prohibited items policy, condition/fit etiquette).

- **Need satisfied:** stakeholders must have confidence that behavior and quality attributes match the prescription.
- **Current situation:** no automated tests exist and the test plan is not started;

usability/acceptance testing is deferred to later milestones; recruitment will be informal ("ask around UPRM"); top priority requirements for acceptance scenarios are not selected yet.

- **Contributions to date:** machine requirements were expressed in measurable terms (e.g., support ~100 simultaneous users) to anchor performance testing; trust-building practices (verification methods, condition labels) were documented to translate into validation checks.
- **Planned outcomes:** test plan, seeded test suites, acceptance scenarios (Given–When–Then) mapped to requirements, trust/safety validations, and a defect taxonomy with triage protocol.

Deployment considerations

The team will define dev/staging environments, configuration, seed/reset data, rollback procedures, and a minimal operations runbook for a student-operated service.

- **Need satisfied:** availability and safe adoption are necessary for accessibility and affordability benefits to materialize.
- **Current situation:** no documented path to deploy, recover, or roll back; backend selection work is informing environment and secrets management but is not yet consolidated in docs.
- **Contributions to date:** Docker usage was explored to standardize developer environments and setup was documented; backend evaluations (Supabase/Firebase) inform environment and secret management decisions.
- **Planned outcomes:** environment definitions, secrets/config guidance, release/rollback steps, seed scripts, and a basic ops runbook.

Stakeholder liaison and feedback (cross-cutting)

The team shall schedule and document periodic touchpoints with students and families in Mayagüez to validate assumptions early (quotes/anecdotes shall be recorded in §2.1.1).

- **Need satisfied:** continuous alignment is required to keep requirements correct and trust high.
- **Current situation:** no formal liaison cadence is defined; external interviews are not planned at this time; consent/ethics approach is undefined.
- **Planned outcomes:** contact cadence, feedback and decision logs, and lightweight consent notes for any future interactions.

Documentation & governance (cross-cutting)

The team shall maintain versioned documentation, change control, risk tracking, and metrics to ensure durable traceability across activities.

- **Need satisfied:** traceability is required to justify decisions and onboard contributors without rework.
- **Current situation:** a docs index/navigation page is considered established (initial draft); changelog and risk register are not started; project metrics beyond "items circulated" are not yet defined.
- **Contributions to date:** the docs/ layout and AsciiDoc style were standardized and most documentation issues were created; branding (logo, color palette, typography) was

established to keep artifacts consistent and legible.

- **Planned outcomes:** docs index and navigation (maintained), changelog, risk register (e.g., technology choice risk, schedule slip, data/privacy misconfiguration), and basic metrics (e.g., items circulated as a primary signal).

1.5 Derived Goals

Derived goals provide higher-level motivations that inform the design of epics and features. Each derived goal links to the epics it directly influences.

DG-1: Promote sustainability literacy and circular practices in Mayagüez The project shall normalize reuse, repair, and responsible disposal behaviors among students and families through donation and resale norms. **Supports:** [EPIC-1], [EPIC-4] **Broader impact:** item lifecycles will be extended and textile waste pressure will be reduced.

DG-2: Strengthen community engagement and mutual aid through UPRM-led outreach The project will cultivate equitable sharing practices (donation, fair resale) centered on UPRM as the primary touchpoint. **Supports:** [EPIC-2], [EPIC-5], [EPIC-6] **Broader impact:** social capital will increase and households will respond more effectively to clothing and accessory needs.

DG-3: Raise awareness of affordability and access constraints faced by local households The project shall make visible how structured sharing reduces acquisition cost and effort for students and families. **Supports:** [EPIC-1], [EPIC-3] **Broader impact:** schools and neighborhood groups will make more informed choices about drives and targeted outreach.

These derived goals guide outreach, education, and validation activities alongside the primary objectives.

2.1 Domain Description

The domain of **hand-me-down clothing exchange** in Puerto Rico is shaped by social, environmental, and economic realities. It exists independently of any digital platform or technical system and can be understood through the people, practices, artifacts, and norms that sustain the circulation of clothing and accessories among students, families, and local communities.

At its core, the domain revolves around two primary actors:

- **Sellers**: Individuals or households who post garments they no longer need, offering them for reuse or resale.
- **Buyers**: Individuals who discover these garments through listings and arrange exchanges directly with the seller, outside the system.

The central entity in this domain is the **Piece**, any individual article of clothing or accessory that circulates between actors. Each Piece carries attributes such as:

- **Type**: The category of clothing.
- **Condition Rating**: A measure of quality or usability.

Pieces move between states through events:

- **Listing Published**: Occurs when a Seller makes a garment visible to potential Buyers.
- **Interest Expressed**: Occurs when a Buyer contacts a Seller regarding a listed garment.
- **Listing Closed**: Occurs when a garment has been exchanged offline or removed from visibility.
- **Discard Event**: Occurs when a Piece leaves circulation, feeding into the wider **Textile Waste Stream**.

The domain is sustained by informal practices and behaviors:

- **Discovery Flows**: The recurring sequence in which garments are listed, browsed, and handed over in person.
- **Condition Disclosure Norms**: The expectation that Sellers will show tag photos, highlight defects, and represent garments honestly.
- **Informal Price Bands**: Symbolic or suggested valuations (typically USD $8–$15), distinguishing resale from donation or commercial sale.
- **Dormant Stock**: Clothing stored in homes awaiting redistribution, resale, or disposal.
- **Meetup Spots**: Semi-public locations (e.g., campus benches, apartment lobbies) chosen for exchanges.
- **Ad-hoc Channels**: Informal digital venues (e.g., Facebook Marketplace, WhatsApp groups, Instagram stories).
- **Trust Cues**: Bilingual communication, recognizable names, or clear photos that influence whether an exchange proceeds.
- **Seasonal Demand Pulses**: Cyclical increases in demand, such as back-to-school or weather-driven surges for uniforms or outerwear.

These elements form an interconnected web of events, actions, and behaviors. Sellers and Buyers rely on ad-hoc digital channels for discovery, agree on terms through direct contact, and meet in semi-public spaces to exchange items offline. Exchanges are underpinned by implicit rules of trust and fairness, while also constrained by larger social and environmental forces such as poverty rates, limited recycling infrastructure, and overflowing landfills.

From a functional perspective, the domain can be represented through abstract operations:

- `publishListing(Piece, Seller, Locale) → ListingPublished`
- `expressInterest(Listing, Buyer) → InterestExpressed`
- `rate(Piece, ConditionRating) → ConditionRating`
- `review(Seller, Buyer, Review) → ReviewSubmitted`
- `categorize(Piece, Type) → Piece`
- `closeListing(Listing) → ListingClosed`

Each function captures an action that transforms the state of a **Listing** or **Piece** within the domain, producing observable changes such as entering circulation, connecting interested parties, or leaving visibility once the exchange occurs offline.

In sum, the domain of secondhand clothing discovery in Puerto Rico is defined by the **visibility of Pieces**, the **actors who list and find them**, the **events that mark transitions**, and the **behaviors and norms** that make these exchanges trustworthy, affordable, and sustainable. This description provides a foundation for later requirements and design work, while remaining independent of any specific system or implementation.

2.1.1 Domain Rough Sketches

This section documents raw, observable examples of clothing exchange and discovery as they occur in daily life. These anecdotes are early insights and do not interpret or generalize; they simply record what was seen or described.

Example 1: Adriana's Search for Affordable Outfits

Adriana, a 20-year-old UPRM student, needs new outfits for an upcoming presentation but cannot afford retail prices. She opens a local Facebook group called "UPRM Ropa y Accesorios" and scrolls through the feed. She spots a post titled **"Blazer, lightly used, $10"** with photos showing the tag and sleeves. Adriana messages the seller in Spanish: **"¿Todavía lo tienes disponible?"** The seller responds quickly and confirms that the blazer is still available. They agree to meet at the main campus entrance the next afternoon. Adriana checks the garment in person, pays in cash, and the seller later marks the post as "Sold." The entire flow—scrolling, messaging, coordinating, and closing—happened outside any dedicated app, relying solely on informal trust and visible cues like clear photos and quick replies.

Example 2: Manuel's Donation After Semester End

Manuel, a middle school teacher and UPRM alumnus, sorts his closet at the end of the semester. He finds several shirts in good condition but no longer wears them. He takes photos, labels them **"Free for pickup near UPRM apartments"**, and posts them in a WhatsApp group where local students trade items. Within hours, a student replies: **"Can I pick up tomorrow morning?"** Manuel agrees, places the items in a bag labeled "Free clothes," and leaves them at his apartment lobby. By afternoon, the bag is gone. No money exchanged hands, and no platform intervention was needed — only quick, direct communication and mutual trust.

Observed Raw Facts

- Listings often use informal phrases like "lightly used," "like new," and "free."
- Meetups happen in semi-public, familiar places.
- Sellers use photos as proof of honesty (tags, condition, size).
- Prices are symbolic and flexible ($8–$15 typical range).
- Listings and messages are bilingual; Spanish often dominates the first contact.
- Exchanges rely on visibility and responsiveness rather than any formal guarantee.

Observed Item Lifecycle

Real-world clothing circulation in the Mayagüez and UPRM community is **not linear**. The same

item may move through several stages repeatedly as different people use it or store it.

Below is a raw-domain sketch of this recurring cycle:

```
      Active Use
           |
           v
       Storage
  (Dormant Period)
           |
           v
Resale or Donation
   (Relisting)
           |
           v
      Exchange
   (In-person)
           |
           v
→ Returns to Active Use
  by a new owner
  (cycle repeats)
```

Items often circulate through several owners. A shirt donated one semester may reappear in a resale group months later, reenter storage, and be listed again by a new owner. This repeating loop—use → store → relist → exchange → new use—continues until the garment is no longer wearable and is ultimately discarded or repurposed.

2.1.2 - Terminology

The following terminology consolidates entities, events, functions, and behaviors in the domain. Each entry specifies the type of concept it represents and the phase in which it is introduced (domain, requirements, design, implementation). This approach avoids circular definitions and ensures alignment with both domain knowledge and system concerns.

Term	Concept Type	Phase Introduced	Definition / Notes
Seller	Role	Domain	Social role representing an actor who offers Pieces for reuse or resale.
Buyer	Role	Domain	Social role representing an actor who browses Listings and initiates contact.
Piece	Entity	Domain	A physical clothing item, existing independently of the system's representation.
Listing	Entity	Design	A system-created representation of a Piece made visible for discovery.

Term	Concept Type	Phase Introduced	Definition / Notes
Listing Published	Event	Domain	Instantaneous domain event marking when a Listing becomes publicly visible.
Interest Expressed	Event	Domain	Event occurring when a Buyer initiates contact regarding a Listing.
Listing Closed	Event	Domain	Event reflecting that a Listing is no longer available, either due to exchange or withdrawal.
Condition Rating	Value Object	Domain	Immutable value representing the assessed quality of a Piece (e.g., scale 1–10).
Review	Value Object	Domain	User-generated evaluative artifact created after an offline interaction; not an event itself.
Locale	Setting	Domain	Physical or institutional environment where exchanges occur (campus areas, public spaces).
Type	Value Object	Domain	Taxonomic clothing classification used to categorize Pieces (e.g., shirts, pants, shoes).
rate(Piece, ConditionRating) → ConditionRating	Action	Design	Operation that evaluates or updates the condition rating of a Piece (pure transformation).
publishListing(Piece, Seller, Locale) → ListingPublished	Action	Design	Action that causes the Listing Published event and results in a new system Listing.
expressInterest(Listing, Buyer) → InterestExpressed	Action	Design	Action that triggers the Interest Expressed event by creating an Interest that references the Buyer and Listing (the Seller is obtained through the Listing, not through a direct structural link).
Interest	Entity	Domain	A persistent association created when a buyer shows interest in a listing; the Interest references the Buyer and the Listing involved, with the Seller derived through the Listing.
closeListing(Listing, Seller) → ListingClosed	Action	Design	Action that ends Listing availability and triggers the Listing Closed event.

2.1.3 Domain Terminology in Relation to Domain Rough

This section explains how several of the terms defined in **2.1.2 – Terminology** were derived through the analysis of the material captured in **2.1.1 – Domain Rough Sketch**. Its purpose is to make explicit the reasoning that transformed informal notes, phrases, and anecdotes into the concepts that organise our description of the hand-me-down clothing domain. It does not repeat the glossary presented in 2.1.2; instead, it narrates the steps that led from raw wording to stable vocabulary.

The rough sketch included an observation that students commonly **"look up an item, contact the seller, agree on price, size, and place, then meet to exchange the item."** This line was analysed as evidence of a recurring, recognisable structure in how people organise exchanges. Rather than a random set of moves, it revealed a social routine for arranging garment handoffs: identify an article, reach out to whoever offers it, discuss its details, and complete the handoff face-to-face. From this analysis emerged the term **Exchange Flow**, which designates the sequence of actions that shapes informal clothing listings and exchanges in the Mayagüez/UPRM setting.

Several fragments in the sketch referred to sellers showing **a photograph of the size tag and full images of the garment, including any flaws**. These snippets suggested an implicit rule: before an exchange, the seller is expected to disclose enough detail to reassure the buyer about size, hygiene, and hidden damage. By organising these remarks, we formulated the term **Condition Disclosure Norm**, a name for the informal convention that garments be represented honestly, with visual evidence, before a meeting occurs. This term encapsulates how participants try to manage uncertainty and build trust.

Price information appeared in multiple places. Notes such as **"symbolic prices between eight and fifteen dollars are common reference points"** hinted at an unwritten valuation practice. The figures were neither arbitrary nor fixed by any authority; they functioned as a social guideline for what counts as reasonable in low-cost exchanges among students. Processing this evidence produced **Student Resale Price Band**, a phrase describing the monetary interval that frames conversations and distinguishes a low-cost resale from a donation.

Other material dealt with what happens when garments do not re-enter circulation. Mentions of **discarded clothes left in open areas or delivered directly to landfill, with only rare opportunities for recycling** highlighted a backdrop of disposal routes. These lines were interpreted as documenting the "exit channels" of clothing once it ceases to be worn. We named this phenomenon **Textile Waste Stream**, referring to the set of pathways — municipal collection, careless dumping, or limited recycling — through which clothing leaves everyday use. This term anchors the environmental dimension of the domain.

A different cluster of notes described **bags of outgrown clothing kept at home while owners decided whether to give them away or sell them**. Here the sketch captured a liminal state: garments were no longer needed, yet had not been reassigned. Analysing this condition produced **Dormant Stock**, a concept for clothing retained in domestic spaces after its initial life, awaiting a new role or final disposal.

The sketch also documented preferred locations for handing over items: students mentioned

campus benches, apartment lobbies, and other familiar public corners. Rather than isolated remarks, these examples pointed to a shared concern for safety and practicality. We consolidated them under the term **Meetup Spot**, which denotes semi-public environments chosen because they balance accessibility, visibility, and comfort during an exchange.

Another strand involved the digital places where clothing is discovered. Notes cited **Facebook Marketplace, WhatsApp groups, and Instagram stories** as typical sources. Instead of treating each separately, we recognised a broader category — lightly moderated online venues where offers, requests, and quick negotiations happen. From this reasoning came the expression **Ad-hoc Channel**, describing the informal communication spaces that enable the visibility of available garments.

Scattered remarks highlighted how participants assess reliability. Seeing clear pictures, recognising a name, or receiving bilingual messages were all said to make people more comfortable proceeding with an exchange. Bringing these hints together led to the concept **Trust Cue**, a label for the small but influential signals that reduce perceived risk in peer-to-peer exchanges.

Finally, the rough sketch posed questions about changes in activity during **back-to-school periods, semester starts, or seasonal weather**. Even without full data, the presence of these queries suggested that demand for particular items is not constant. To represent this dynamic aspect we coined **Seasonal Demand Pulse**, a term for the predictable fluctuations in which garments are offered or sought as academic and climatic cycles progress.

By articulating these derivations, this section clarifies the analytical bridge between the exploratory material of the rough sketch and the structured vocabulary presented in the Terminology section. Understanding this path is essential for tracing how domain knowledge was built and for ensuring that later requirements remain anchored in the observed environment.

2.1.4 Domain Narrative

In Puerto Rico, especially around university communities like Mayagüez, informal clothing exchange is woven into everyday life. Students, families, and neighbors frequently pass along garments that no longer fit, are no longer needed, or might be useful to someone else. Most of these exchanges begin casually: a photo shared through a group chat, a quick post on social media, or a message asking whether anyone nearby has a specific item available.

People commonly describe the condition of an item, mention its size, and include a picture taken at home or on campus. Those looking for clothing browse these posts whenever they have a free moment—between classes, during breaks at work, or at home in the evening. When something catches their attention, they reach out directly to the person offering it and ask whether it is still available.

If both parties agree, they arrange to meet somewhere familiar and convenient, usually on or near the UPRM campus, at a bus stop, in a parking lot, or outside a classroom building. The handoff is quick and direct. Some exchanges involve a symbolic price, while others are simple gifts. Afterward, people often let their friends or peers know the item has been taken so others do not continue asking about it. In many cases, those who have had a positive experience mention it to others or comment on it in the group where the item was originally shared.

Alongside these exchanges, people sometimes keep track of items they find interesting, ask friends to check certain posts for them, or look at previous interactions whendeciding whether to engage with someone new. These habits—sharing, searching, meeting, and informal reputation—form the everyday flow of secondhand clothing circulation in the community.

2.1.5 Events, Actions, and Behaviors

This section describes only domain **actions**, **events**, and the **behaviors** they form. Entities and structural concepts appear elsewhere and are not repeated here.

Events

Listing Published

An instantaneous occurrence when a seller makes a clothing item publicly visible to others.

Interest Expressed

An instantaneous occurrence when a buyer signals desire to acquire or inquire about a listing. This event marks the **creation** of an **Interest** entity connecting the buyer and seller.

Listing Closed

A moment in which the seller marks that the item is no longer available after an offline exchange or withdrawal.

Listing Updated

An instantaneous occurrence when a seller changes visible details (title, condition, images, category, etc.).

Actions

Publish Item

A seller provides the information necessary to make an item visible (photos, condition, category). *Triggers:* **Listing Published**

Browse

A buyer looks through available items, optionally using categories, keywords, or filters. *Does not trigger an event by itself.*

Express Interest

A buyer sends a message or contact request asking about availability or arranging a meetup. *Triggers:* **Interest Expressed**

Update Listing

A seller modifies previously posted information. *Triggers:* **Listing Updated**

Close Listing

A seller marks the item as no longer available after an offline exchange or decision to withdraw it. *Triggers:* **Listing Closed**

Behaviors

A behavior is a recurring domain pattern composed of actions + events that appear across many exchanges.

1. Listing Lifecycle Behavior

A recurring cycle in which a seller manages the visibility of an item.

Trigger	Seller decides to make an item available.
Actors	Seller (primary)
Actions + Events	- Publish Item → **Listing Published** - Update Listing (optional) → **Listing Updated** - Close Listing → **Listing Closed**
Observed Pattern	Items enter visibility, may be refined, and eventually leave circulation.

2. Discovery Behavior

The pattern by which buyers locate items of interest.

Trigger	Buyer searches for clothing or browses casually.
Actors	Buyer (primary), Seller (indirect)
Actions + Events	- Browse (buyer) - (optional) filtering or keyword search *No event is triggered until buyer expresses interest.*
Observed Pattern	Buyers scan available offerings opportunistically, often during brief windows of free time.

3. Interest and Contact Behavior

A recurring interaction when a buyer wants more information or wishes to acquire the item.

Trigger	Buyer identifies an interesting item during discovery.
Actors	Buyer (primary), Seller (respondent)
Actions + Events	- Express Interest → **Interest Expressed** (this event initiates the period during which buyer and seller coordinate) - Seller responds (clarifications, availability confirmation)
Observed Pattern	Short informal message chains establish availability and next steps.

4. Offline Exchange Behavior

The final handoff of the item, outside the platform.

Trigger	Buyer and seller agree to meet after interest is established.
Actors	Buyer and Seller

Actions + Events	- Coordinate meetup (time, place) - Physical handoff - Seller closes listing → **Listing Closed**
Observed Pattern	Exchanges occur face-to-face in familiar public areas; value transfer is informal.

5. Post-Exchange Feedback Behavior

Reputation-related actions occurring after the physical exchange.

Trigger	Exchange has concluded and listing has been closed.
Actors	Buyer (primary), Community (indirect)
Actions + Events	- Buyer leaves informal comments or reviews through social channels (No required event; this varies by community norms)
Observed Pattern	Positive experiences are shared informally; negative ones are quietly avoided.

2.1.6 - Function Signatures

Objective

Define domain-level function signatures that describe how actions are carried out, including inputs, outputs, and possible changes in the domain.

Description

Function signatures specify how actors in the system interact through domain actions. They define the logical relationships between inputs, outputs, and resulting state changes:

- **The name of the function**: The action being performed.
- **The input parameters**: The information required by the action.
- **Output**: Type of data the function produces.
- **State changes**: How the action affects the domain.

The general format of a function signature is:

- `FunctionName : Input1 >< Input2 >< ⋯ → OutputType`
- **Description** – what the function does.
- **Preconditions** – what must be true or available for the function to execute.
- **Postconditions** – what becomes true afterwards.
- **Notes** – clarifications about scope, limits, or non-covered cases.

In addition to individual entities (Piece, Listing, Seller, Buyer), we use:

- `ListingRepository` as the **domain-level collection** that conceptually stores all listings in the

system. From this repository we can derive:

- **all listings** (including closed/archived),
- **available listings** (listings that are still open and visible to buyers).

Browsing and searching operations always work **against** this conceptual repository.

Filter Value Object

A **Filter** is a **value object** that represents buyer-defined search constraints. It is immutable and does not depend on `Piece`, `Page`, or UI-layer concepts.

The Filter has the following fields:

Field	Description	Type / Concept
`category`	Desired category of the item. Optional.	Value: Category
`size`	Requested size. Optional.	Value: Size
`gender`	Intended gender classification. Optional.	Value: Gender
`minCondition`	Minimal acceptable condition rating. Optional.	Value: ConditionRating
`maxPrice`	Maximum price the buyer is willing to pay. Optional.	Value: Money
`locale`	Campus/location where the buyer is willing to meet. Optional.	Value: Locale
`textQuery`	Free-text query for title/description matches. Optional.	Value: String
`onlyActive`	Whether to restrict results to active listings. Defaults to true.	Boolean

The Filter induces a matching predicate:

`matches(f: Filter, p: Listing)` holds iff all present fields in `f` are satisfied by `p`.

Function Signatures

publishListing : Piece >< Seller >< Locale → Either[ValidationError, Listing]

Description

A Seller publishes a Piece in a given Locale, producing a new Listing.

Preconditions

Seller and Piece exist and are active; Piece has required metadata (title, condition, images).

Postconditions

A new Listing is created and stored in the ListingRepository as **available**.

Notes

Discovery-only; no in-platform payments.

expressInterest : Listing >< Buyer → Either[ContactError, InterestExpressed]

Description

A Buyer signals interest in a Listing, creating an Interest entity that references the Buyer and the Listing (the Seller is determined from the Listing).

Preconditions

Listing is available (open); Buyer is authenticated.

Postconditions

InterestExpressed is recorded as an Interest referencing the Buyer and the Listing; the Seller is indirectly determined through the Listing.

rate : Piece >< ConditionRating → Piece

Description

Assigns or updates a Piece's condition rating.

Preconditions

Piece exists; rating within valid scale.

Postconditions

Returns the updated Piece.

review : Seller >< Buyer >< Review → Either[ReviewError, ReviewSubmitted]

Description

Records a review after an offline exchange.

Preconditions

A prior successful exchange exists.

Postconditions

ReviewSubmitted maintains references to the Listing and the User being reviewed; Listings do not store or contain Reviews structurally.

closeListing : Listing >< Seller → Either[ClosedError, Listing]

Description

Seller closes an active Listing after an exchange or withdrawal.

Preconditions

Seller owns the Listing; Listing is active.

Postconditions

Listing state becomes terminal (sold/donated/retracted) and the updated Listing is returned.

categorize : Piece >< Type → Option[Piece]

Description

Assigns a category type to a Piece.

discard : Piece → Option[Void]

Description

Marks a Piece as inactive and removes it from circulation.

browseAvailableItems : ListingRepository → Set<Listing>

Description

Returns the set of Listings that are currently **available**.

Preconditions

ListingRepository is defined.

Postconditions

Returns a (possibly empty) set of available Listings.

Notes

Backend returns only active listings.

findAvailableByFilter : ListingRepository >< Filter → Set<Listing>

Description

Returns the set of available Listings that satisfy the Filter predicate.

Preconditions

ListingRepository is defined; Filter is valid.

Postconditions

Returns all Listings that are:

- available, and
- matching the Filter.

Notes

Distinguishes “all listings” from the subset that is currently available and relevant to buyers.

findListingById : ListingRepository >< ListingID → Option[Listing]

Description

Retrieves a Listing by identifier from the repository.

Example Scenario: From Listing to Offline Exchange

1. Seller publishes a listing using `publishListing(Piece, Seller, Locale) → Listing`, stored in the

ListingRepository as **available**.

2. Buyer browses available items using `browseAvailableItems(ListingRepository) → Set<Listing>` or refines results using `findAvailableByFilter(ListingRepository, Filter) → Set<Listing>`.
3. Buyer expresses interest using `expressInterest(Listing, Buyer) → Either[ContactError, InterestExpressed]`.
4. The exchange happens offline.
5. Seller closes the listing using `closeListing(Listing, Seller) → Either[ClosedError, Listing]`.
6. Buyer leaves a review using `review(Seller, Buyer, Review) → Either[ReviewError, ReviewSubmitted]`.

2.1.7 Closure Under Operations

Closure Under Operations ensures that every domain action returns an object of the **same conceptual type** as the entity being transformed. A function that takes a Listing must return a Listing; a function that updates a Piece must return a Piece. This keeps all transformations inside the domain vocabulary and prevents leaking UI objects, storage formats, or event structures into the domain layer.

Motivation

Earlier versions of the model returned heterogeneous outputs: some functions returned events, others returned partial data, and several returned structures tied to implementation details. These breaks in closure made it difficult to chain operations and prevented the domain from being expressed as a stable algebra of transformations.

This was corrected by ensuring that all operations return either:

- the updated domain entity, or
- a domain-defined error wrapped around the same entity type.

Updated Function Shapes

The following operations now preserve closure and match the function definitions in Section 2.1.6 of the documentation.

- publishListing : Piece >< Seller >< Locale → Either[ValidationError, Listing]
- closeListing : Listing >< Seller → Either[ClosedError, Listing]
- rate : Piece >< ConditionRating → Piece
- browseAvailableItems : ListingRepository → Set<Listing>
- findAvailableByFilter : ListingRepository >< Filter → Set<Listing]

Listing-related transformations return Listings, and piece-related updates return Pieces.

Cross-Team Evidence of Closure

- Authentication operations always return the canonical AuthContextValue shape or an AuthError.
- Listing and piece operations always return Piece or Listing entities, never DTOs.
- Location operations always return Location or List<Location>.

This cross-team consistency ensures that closure is preserved throughout the system.

Unified Example of Closure

A typical Listing lifecycle demonstrates closure under operations:

1. A seller publishes an item, returning a Listing.
2. A buyer expresses interest, producing an InterestExpressed event referencing the same Listing.
3. The seller closes the listing, returning the updated Listing.

Events annotate transitions, but the entity remains stable throughout.

Canonical Flow Diagram

```
Piece + Seller + Locale
          |
          v
       Listing (available)

Listing + Buyer
          |
          v
   InterestExpressed (event)
          |
          v
       Listing (same structure)

Listing + Seller
          |
          v
       Listing (closed)
```

Benefits

Enforcing closure under operations provides:

- predictable chaining of domain functions,
- compatibility with Command–Query Separation,
- composability across repositories and domain collections,

- a stable set of shapes for testing, validation, and requirements traceability.

2.2.1 Epics, Features, and User Stories

Epics

Epics are a higher-level overview of goals that are large enough in scope that they can be broken down into multiple sprints. They provide direction and group related work together.

Buyer Epics

1. Listing Discovery
    a. As a buyer, I want to search for listings by category, filters, and keywords, so that I can quickly find clothing items that meet my needs. **Implemented by:** [FEATURE-1.1], [FEATURE-1.2], [FEATURE-1.3]

1. Saved Listings
    a. As a buyer, I want to save or bookmark listings I am interested in, so that I can revisit them later when deciding whether to contact the seller. **Implemented by:** [FEATURE-2.1], [FEATURE-2.2]

1. Trust and Transparency
    a. As a buyer, I want to view detailed seller profiles and leave reviews after exchanges, so that I can make informed and confident decisions about future interactions. **Implemented by:** [FEATURE-3.1], [FEATURE-3.2], [FEATURE-3.3], [FEATURE-3.4]

Seller Epics

1. Listing Management
    a. As a seller, I want to create, edit, and close listings with detailed information, so that I can effectively manage the items I am offering for reuse or resale. **Implemented by:** [FEATURE-4.1], [FEATURE-4.2], [FEATURE-4.3], [FEATURE-4.4]

1. Interest Notifications
    a. As a seller, I want to receive notifications when someone shows interest in my item, so that I can promptly respond and arrange the exchange offline. **Implemented by:** [FEATURE-5.1], [FEATURE-5.2]

1. Seller Profile and Trust
    a. As a seller, I want to maintain a profile with personal and location details, so that I can establish credibility and attract buyers interested in my listings. **Implemented by:** [FEATURE-6.1], [FEATURE-6.2]

Features

Features specify the functionality required to deliver the goals described in the epics to the user.

They serve to provide more concrete goals related to development.

Buyer Features

1. Listing Discovery
    a. Filtering (clothing type, size, color, condition, category, price/free marker, etc.). **Supports:** [EPIC-1] **Requires:** [REQ-DR1], [REQ-DR2], [REQ-DR4]

a. Keyword search. **Supports:** [EPIC-1] **Requires:** [REQ-DR1], [REQ-DR3]

a. Sorting options (newest first, alphabetical, etc.). **Supports:** [EPIC-1] **Requires:** [REQ-DR1]

1. Saved Listings

b. Ability to bookmark or save listings for later consideration. **Supports:** [EPIC-2] **Requires:** [REQ-DR1]

a. Saved listings persist across sessions and devices. **Supports:** [EPIC-2] **Requires:** [REQ-DR1]

1. Trust and Transparency

b. Seller profile page (location, account age, bio). **Supports:** [EPIC-3] **Requires:** [REQ-DR3]

a. Seller ratings & reviews system. **Supports:** [EPIC-3] **Requires:** [REQ-DR3]

a. Buyer-to-seller review submission flow. **Supports:** [EPIC-3] **Requires:** [REQ-DR3]

a. Reporting mechanism for problematic listings or suspicious behavior. **Supports:** [EPIC-3] **Requires:** [REQ-DR1]

Seller Features

1. Listing Management
    a. Create listing form (title, description, tags, price/free marker, category). **Supports:** [EPIC-4] **Requires:** [REQ-DR1], [REQ-DR4]

a. Upload multiple images per listing. **Supports:** [EPIC-4] **Requires:** [REQ-DR1]

a. Edit listing details (update description, condition, images, price). **Supports:** [EPIC-4] **Requires:** [REQ-DR1], [REQ-DR5]

a. Close or deactivate a listing once the exchange is complete offline. **Supports:** [EPIC-4] **Requires:** [REQ-DR3]

1. Interest Notifications

b. Push/email/in-app notifications when a buyer expresses interest in a listing. **Supports:** [EPIC-5] **Requires:** [REQ-DR3]

a. Notification center showing recent messages or contact attempts. **Supports:** [EPIC-5] **Requires:** [REQ-DR3]

1. Seller Profile and Trust

b. Editable seller profile (profile picture, name, location, short bio). **Supports:** [EPIC-6] **Requires:**

[REQ-DR3]

a. Seller dashboard displaying active and closed listings, as well as reviews and ratings. **Supports:** [EPIC-6] **Requires:** [REQ-DR1], [REQ-DR3]

User Stories

User stories are derived from Features, breaking them down into smaller, individual tasks to be added to the backlog. These stories focus on user needs and help make development more user-focused.

Buyer User Stories

1. As a buyer, I want to browse listings by category to find a specific type of item I want.
2. As a buyer, I want to filter my search by size, condition, and category to tailor my search to my preferences.
3. As a buyer, I want to search for listings using keywords to find specific items that may not fit predefined categories.
4. As a buyer, I want to save interesting listings so I can revisit them later.
5. As a buyer, I want to view a seller's information such as location, reviews, and account details to feel confident before contacting them.
6. As a buyer, I want to leave reviews for sellers after an offline exchange so others can trust the process.

Seller User Stories

1. As a seller, I want to create listings for my items with options such as adding multiple pictures, a description, and a condition rating so interested buyers can find me easily.
2. As a seller, I want to edit my listings so I can update details whenever necessary.
3. As a seller, I want to close listings once the item is exchanged offline so the system stays accurate.
4. As a seller, I want to receive notifications when someone contacts me about a listing so I can respond quickly.
5. As a seller, I want to provide information on my profile such as my name and location to increase trust with buyers.

2.2.2 - Personas

A persona is a fictional yet plausible representation of the goals, needs, and expectations of any person within the project's domain. These don't have to just be direct users of the platform; anyone who benefits from the project's goals in one way or another can be a good persona. These personas serve to guide development in such a way where users are kept in mind when creating, developing, and refining features.

The following personas represent direct users of the platform:

Adriana Gómez

- **Age**: 20 years old
- **Occupation**: University student on financial aid working part-time on campus
- **Build / Appearance**: Dark and straight hair, low to average height, dresses in vintage clothes and kitschy accessories
- **Personality**: Creative, unique, community-driven

Adriana loves expressing herself through her fashion, yet she struggles finding clothes that feel like her. She has an affinity for older clothing, so thrift stores are a common spot for her. However, she struggles finding thrift stores in her area given that most of them don't have an online presence and, between her studies and her job, she doesn't have a lot of time to go out and look for them.

- **Goals:** Build a wardrobe that reflects her individuality without overspending
- **Pain Points**: Difficulty finding unique clothing items and accessories that don't break the bank, difficulty finding local thrift stores
- **Needs**: Access to trustworthy online listings, a way to find local thrift stores
- **Platform Interaction**: Browses listings from her phone during her breaks between classes
- **Keep Adriana in mind when:** Developing the general listings layout

Manuel Torres

- **Age**: 35 years old
- **Occupation**: Middle school teacher
- **Build / Appearance**: Brown and curly hair, average height, frequently seen in polo shirts or t-shirts with witty quotes on them.
- **Personality**: Patient, funny, good with kids, adventurous

Manuel is moving to a new location and wants a fresh start. Even if he doesn't want a lot of his belongings anymore, he'd rather not throw them away. He wants to sell whatever he has that's still in good condition to help out with the expenses involved in the move but can't find a way to do so.

- **Goals:** Simplify his moving process while giving his clothes a second life
- **Pain Points**: No low-hassle way to sell his belongings
- **Needs**: Access to a way to sell his belongings on his own terms
- **Platform Interaction**: Creates listings from his work computer for his pre-loved clothes
- **Keep Manuel in mind when:** Developing the listing creation interface and functionality

Daniela López

- **Age**: 27 years old
- **Occupation**: Nurse practitioner
- **Build / Appearance**: Long and wavy brunette hair, tall, dresses in comfortable clothes like baggy jeans or athleisure

- **Personality**: Outspoken, understanding, eco-conscious

Daniela is the type of person to own something until it falls apart. She is vehemently against fast fashion and tries to not contribute to it to the best of her ability. She believes older clothing was made to last longer and buys second-hand whenever she can, but her favorite local thrift store has just closed down and the closest is now too far away for her to visit frequently.

- **Goals:** Achieve convenient access to good-quality clothing
- **Pain Points**: Doesn't want to contribute to fast fashion, tired of the low quality of modern clothing items
- **Needs**: A way to purchase good quality vintage clothing items
- **Platform Interaction**: Browses listings using search filters to narrow down her results to her needs
- **Keep Daniela in mind when:** Developing the search filters

As mentioned, it's not just direct users of the platform that benefit from it; other people within the domain can also benefit from the platform's goals. The following personas pertain to the project's domain:

Isidra Montoya

- **Age**: 61 years old
- **Occupation**: Thrift store owner
- **Build / Appearance**: White and straight hair in a pixie cut, short in stature, wears comfortable outfits like jeans and a t-shirt
- **Personality**: Friendly, welcoming, hard-working

Isidra runs a small thrift store that has been struggling to pull in customers as of late. Her store isn't located very close to the main highway and lacks an online presence due to Isidra's limited technical knowledge and reluctance to learn, making her rely mainly on word of mouth and the occasional curious passerby.

- **Pain Points**: Limited digital presence, lack of technical skills, poor reach
- **Needs**: An alternate way to expand her shop's reach to attract more customers
- **Platform Interaction**: Despite not interacting directly with the platform, a user uploaded her store's location to the thrift store map, expanding her store's reach and allowing her to gain the additional customers she needed.
- **Keep Isidra in mind when:** Developing the map feature and designing user friendly interfaces so intuitive that even those with poor technical knowledge can use them

Fernando Ruiz

- **Age**: 29 years old
- **Occupation**: Event organizer and nonprofit volunteer
- **Build / Appearance**: Short hair that has been dyed blue at the tips, medium build, tall, usually

seen wearing light colors like whites and pastels

- **Personality**: Proactive, charismatic, enthusiastic

Fernando is very involved in volunteer work and helps host monthly clothing drives in his community to help those in need. However, getting sufficient clothes for the drives has been a recurring issue, as they don't receive enough donations to help as many people as they'd like.

- **Pain Points**: Lack of clothing donations due to a lack of visibility
- **Needs**: More contributors/donors
- **Platform Interaction**: Despite not interacting directly with the platform, a user of the platform added the recurring clothing drive event to the map, increasing its visibility and contributing to getting more clothing donations for the drive.
- **Keep Fernando in mind when:** Developing the map feature

2.2.3 Domain Requirements

DR1: The system must classify every listing under exactly one primary category. **Supported by:** [FEATURE-1.1], [FEATURE-1.2], [FEATURE-1.3], [FEATURE-2.1], [FEATURE-2.2], [FEATURE-3.4], [FEATURE-4.1], [FEATURE-4.2], [FEATURE-4.3], [FEATURE-6.2]

DR2: Categories may have hierarchical subcategories. **Supported by:** [FEATURE-1.1]

DR3: The system must distinguish between **Resale** and **Free/Donation** listings. **Supported by:** [FEATURE-1.2], [FEATURE-3.1], [FEATURE-3.2], [FEATURE-3.3], [FEATURE-4.4], [FEATURE-5.1], [FEATURE-5.2], [FEATURE-6.1], [FEATURE-6.2]

Justification: Why Donation Is Not Equivalent to a Zero-Price Sale

While both resale and donation result in a transfer of an item, they represent **fundamentally different domain phenomena** that must remain distinct in the model. Treating donation as simply "a sale at price = 0" erases critical behavioral, motivational, and social differences observed in the field:

Behavioral Motivation - **Resale** is economically motivated: sellers intend to recover value. - **Donation** is altruistic: donors intend to help others, reduce waste, or give items a second life. These intentions affect how users browse, filter, and evaluate listings.

Community Norms In observed platforms and local UPRM reuse culture, donation is framed as part of **mutual aid** and community support. A zero-price sale does not communicate the same norm of generosity or accessibility.

Visibility & Discovery Differences Users frequently search **only** for free items—especially students with financial constraints. Treating donation as a sale at price 0 would require special-case logic to recreate a distinction that exists naturally in the domain.

Trust & Safety Implications Donation interactions often carry lighter expectations: - no negotiation - no price comparison - less pressure to "justify" quality Resale carries transactional expectations around condition, fairness, and accuracy.

Lifecycle & Data Semantics Donation signals a **different lifecycle path**, aligning with sustainability and circular-economy practices. Platforms and institutions (e.g., sustainability coordinators, thrift partners) often track donation volumes separately from resale activity.

Legal & Institutional Context Some campus programs, community drives, or thrift partners only accept donation flows—not resales. Modeling donation explicitly allows these stakeholders (e.g., Fernando, Isidra, Dr. Rivera) to interpret the platform's impact correctly.

For these reasons, the system must treat donation as its own domain concept rather than encoding it as a numeric edge case. This preserves clear intent, aligns with community expectations, and supports accurate categorization, filtering, reporting, and future integrations with sustainability efforts.

DR4: The system shall **support** item-to-category compatibility by helping users select an appropriate category for their listing. The system should provide suggestions or guidance, but shall not attempt to strictly enforce correctness.

To assist users, the system may: - suggest likely categories based on the title, description, or user-selected attributes - highlight common category choices for similar items - warn when a category seems unusual (e.g., "Did you mean: Footwear?") - optionally use image recognition or LLM-assisted suggestions to improve accuracy

This requirement does **not** imply strict enforcement or prevention of mismatches. Its purpose is to reduce accidental misclassification while respecting user autonomy.

Supported by: [FEATURE-1.1], [FEATURE-4.1]

DR5: The system must support category evolution. **Supported by:** [FEATURE-4.3]

2.2.4 Interface Requirements

Objective

Define how users interact with the system through visible UI elements, controls, states, and feedback. These **Interface Requirements** specify the behavior of forms, buttons, validation, navigation, and visibility rules. The platform facilitates listing and discovery but does not process payments or host transactions; all exchanges occur directly between buyers and sellers outside the system.

Interface Requirements

Create Listing Form - The authenticated user is implicitly the seller; **no seller field is displayed**. - Required fields: **Title**, **at least one Image**, and **Category**. Optional fields: **Price** (if absent, listing is marked **Free**), **Description**, **Condition**, and **Size**. - When selecting a **Category**, the system may show suggested categories based on the item's title, description, or images. These suggestions help reduce misclassification but do **not** restrict the user's choice. - The **Publish** button is **disabled** until all required fields are valid. - On submit with missing or invalid fields, show **inline errors**; **focus moves** to the first invalid field. - On success, navigate to the **Listing Details** page and show a **toast**: "Listing published."

Listing Details (Seller View) - Shows editable fields (Title, Description, Images, Price, Condition) while **Status = Active**. - Includes a **Status** dropdown with transitions: **Active → Reserved → Closed** (no skipping). - When **Status = Closed**, all edit controls are **disabled**, and a **'Closed' badge** is displayed. - Displays **Contact Requests** received for that listing, sorted by most recent.

Browse & Search (Buyer View) - Buyers can **search by keyword** and **filter by Category, Condition, Size, and Price/Free marker**. - Sorting options include **Newest first**, **Lowest price**, and **Condition rating**. - Each listing card shows **Title, Thumbnail, Price/Free marker, Condition, and Seller rating (if available)**. - Clicking a card opens **Listing Details**, where the **Contact Seller** button is visible if the viewer is not the owner.

Saved Listings (Buyer View) - Buyers can **Save/Unsave** a listing from both the card and details views. - Saved items persist across sessions and appear in the **Saved Listings** section under the user menu. - Saved icons visually change state (e.g., outlined vs. filled heart).

Contact Seller / Messaging - **Contact Seller** opens a message composer prefilled with the listing title. - On send, show confirmation ("Message sent") and add it to **Message Threads**. - Seller receives a **notification badge** in the header or inbox. - Messages are displayed chronologically, grouped by listing.

Profile & Reviews - Sellers and buyers each have a **profile page** displaying their listings, ratings, and reviews. - Users can **edit their own profile** (profile picture, bio, location). - Reviews are **read-only** for the profile owner but can be submitted by others after an offline exchange. - The **average rating** and total reviews are displayed prominently at the top of the profile.

Accessibility & Feedback - All interactive elements are **keyboard-accessible** (Tab/Shift+Tab) with visible **focus outlines**. - Inline validation messages are **ARIA-live announced** to screen readers. - Toasts and banners are **non-blocking** and dismissible via keyboard. - Forms include descriptive labels and placeholder text for clarity.

Examples

1. **Create Listing**: A seller opens the **Create Listing** form. Title is empty and there are no images, so **Publish** is disabled. The user enters a title, selects a category, and uploads a photo. **Publish** becomes active. On submit, a success toast appears and the app redirects to the Listing Details page.
2. **Reserve Then Close**: A seller opens their Listing (Status = Active), selects **Reserved** from the dropdown. After completing the exchange offline, they select **Closed**, disabling all edit controls.
3. **Contact Seller**: A buyer opens a listing and clicks **Contact Seller**, writes a message, and sends it. A confirmation appears, and the seller's header shows a **1** notification badge.

Relation to Domain Requirements

- Domain Requirements define what must exist (e.g., Listing entity, User, Saved association).
- Interface Requirements define how users **create, view, modify, and interact** with those entities through UI components.
- Each interface rule directly supports a domain rule by providing a visible, testable interaction.

Justification

This section ensures that UI behaviors are intuitive, consistent, and verifiable. It separates internal representation (domain) from what users experience, providing a clear mapping between actions and visible states. These behaviors reinforce trust and usability while aligning with accessibility and real-world discovery practices.

Testing Plan

- Test each form and button for proper enable/disable states and validation.
- Confirm that “Publish” and “Close” actions trigger correct navigation and feedback messages.
- Verify that saved listings persist across sessions.
- Ensure that editing is disabled once a listing is closed.
- Check accessibility by tab-navigating forms and validating screen reader output.
- Confirm that notifications appear when a buyer sends a contact message.

2.2.5 Machine Requirements

Objective

The web server must support reliable and efficient operation of a React + JavaScript clothing discovery platform, enabling users to create, browse, filter, and save listings. Requirements are defined in measurable, testable terms to ensure responsiveness, reliability, and scalability. The platform does not handle payments, carts, or checkout operations; all exchanges between buyers and sellers occur offline after contact through the system.

To avoid ambiguity, terms such as **simultaneous users**, **average load**, **peak load**, and **uptime window** are defined in advance. (See the “Clarifications,” “Load Definitions,” and “User Mix Scenarios” sections below for detailed breakdowns.)

Requirements

Performance - The system shall return **search results within 2 seconds on average**, where “average” refers to the mean response time over a **5-minute rolling window** during normal operating hours. - The system shall maintain a maximum response time of **4 seconds** under peak load, defined as **150 concurrent active users** performing real-time actions (see Clarifications Clarifications). - The system shall support **200 simultaneous browsing users**, where “simultaneous” refers to users performing actions within the same **10-second activity window** (e.g., searching, filtering, saving listings, viewing details). - Listing creation and updates (e.g., uploading images, editing descriptions, updating status) shall complete within **3 seconds on average**, excluding client-side image compression delays.

Reliability - The web server shall maintain an uptime of **99.7% measured monthly**, allowing no more than **2.1 hours** of unscheduled downtime **per calendar month**. - Core operations — creating, editing, saving, and closing listings — must be processed reliably, with **no more than 0.1% of requests** failing due to server errors. - Scheduled maintenance shall be limited to **3 hours per**

month, with announcements given at least **48 hours in advance**.

Scalability - The system shall scale to support **500 concurrent users** performing mixed actions (browsing, searching, creating listings, saving favorites, sending messages) while maintaining average response times ≤ **3.5 seconds**. - The system shall handle a database of up to **50,000 active listings** without significant degradation (response time increase ≤ **20%** compared to baseline). - The system shall support both **vertical scaling** (increasing server resources) and **horizontal scaling** (adding more servers or instances) without major architectural changes.

Clarifications

- "Peak load" is defined as ≥ **150 concurrent active users** within a **10-second action window**, with at least **10% performing listing creation or updates**.
- "Average load" refers to typical daytime usage of **200 concurrent users**, where actions occur within a **1-minute window**.
- "Minimal outages" means ≤ **2.1 hours of unplanned downtime per month**.
- "Acceptable performance" means ≤ **3.5 seconds response time for 95% of requests** measured over a **rolling daily interval**.
- Terms are defined before more detailed breakdowns in the **Load Definitions** section (see below).

Areas for Refinement

- Stress tolerance for **extreme traffic spikes** (≥ 1000 users during semester openings or large donation drives) remains under evaluation.
- Performance on **mobile devices under weak network conditions** requires additional benchmarking.
- Further testing is required for **concurrent image upload** operations and cache performance during peak activity.

Justification

The clothing discovery platform requires fast, consistent responses to maintain usability and trust. Setting explicit definitions for "simultaneous," "average," "peak," and "uptime windows" ensures the requirements are measurable and testable. These benchmarks align with lightweight listing platforms similar to Facebook Marketplace or OfferUp and support predictable behavior during high-demand periods such as semester starts or community donation events.

Load Definitions

Refined load definitions include: - **Peak Concurrent Users**: 500 users performing actions within the same **10-second window**. - **Average Concurrent Users**: 200 users active within a **1-minute** usage window. - **Light Load**: < 80 active users interacting within a **1-minute window**.

User Mix Scenarios

1. **Scenario 1: Browsing Listings**
    - 60% of users are browsing listings.
    - Actions: Filtering, searching, viewing item details.
2. **Scenario 2: Creating Listings**
    - 20% of users are creating new listings.
    - Actions: Uploading photos, entering item details, submitting listings.
3. **Scenario 3: Messaging**
    - 15% of users are messaging sellers or buyers.
    - Actions: Sending and receiving messages.
4. **Scenario 4: Administrative Actions**
    - 5% of users are performing administrative tasks.
    - Actions: Moderating listings, resolving disputes.

2.3 Implementation

Objective

Describe the overall architecture and design of the clothing repurposing application, showing how the React frontend, JavaScript logic, and Supabase database/storage/auth components work together. Include mockups and selected fragments that clarify implementation decisions. This chapter distinguishes between what has already been implemented and what remains part of the planned architecture.

Description

The application is structured as follows. Where relevant, we clarify which parts are already implemented in the current system versus planned extensions that have architectural support but no functional code yet.

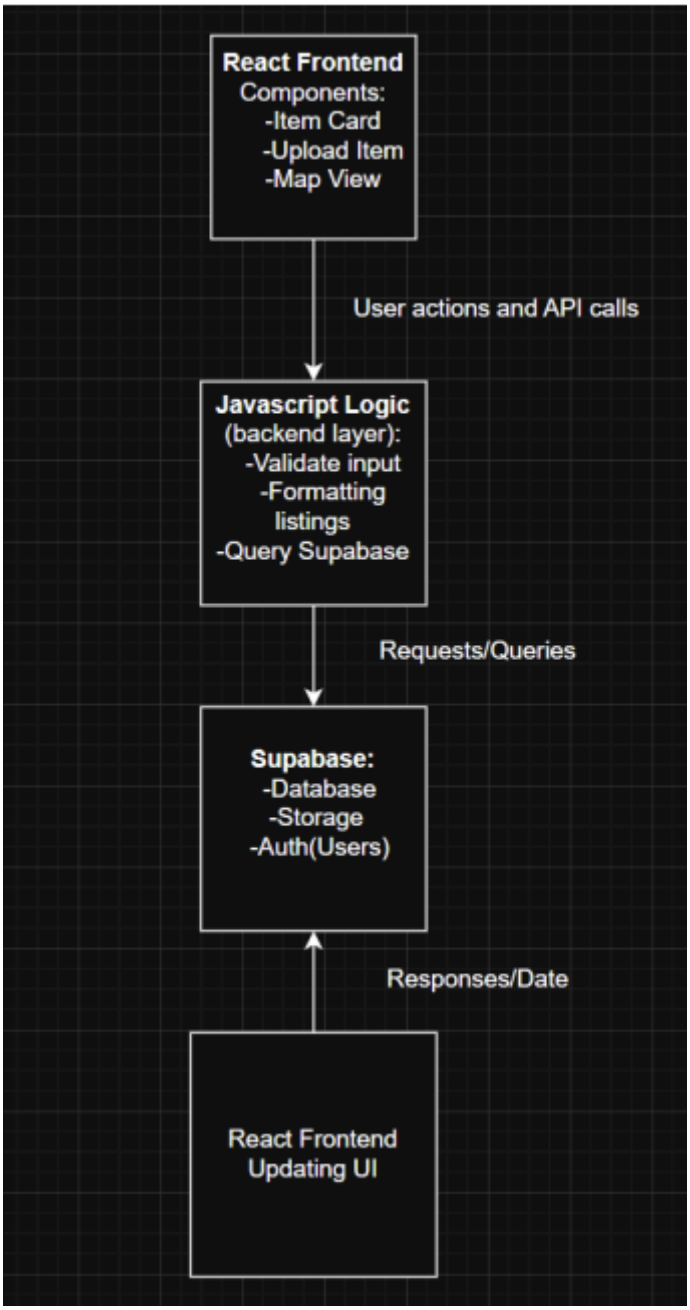
- Frontend (React)
    - The implemented portions of the frontend handle the user interface for browsing active listings, saving favorites, uploading new listings, and loading donation centers. These components are fully connected to Supabase and reflect real data.
    - Some UI pages are **planned but not yet implemented**; these exist only as prototypes and their purpose here is to show intended architectural integration, not to behave as a user manual.
    - Planned pages/screens include:
        - Home Page: shows recent listings and navigation options. **(Prototype implemented; data binding functional.)**
        - Clothing Listing Page: displays a scrollable list of available items with filters and sorting.

(Core listing feed implemented; extended filters planned.)

        - Upload Page: supports publishing items as listings. **(Partially implemented: image upload + required-field validation completed; extended editing flows planned.)**
        - Saved Listings Page: lets buyers revisit bookmarked items. **(Basic saving implemented; full saved-list dashboard planned.)**
        - Map Page: integrates with Leaflet to display donation centers. **(Basic map rendering implemented; advanced metadata display planned.)**
    - Styling is implemented with CSS modules or CSS-in-JS for maintainability.
- Backend Logic (JavaScript functions)
    - The backend logic includes a mixture of implemented modules and planned modules. Implemented logic includes:
        - Input validation for listing creation.
        - Image uploads to Supabase Storage.
        - Authentication integration via Supabase Auth.
        - Basic CRUD operations on listings and user profiles.
        - Enforcement of invariants such as required fields and mandatory seller identity.
    - Planned logic includes:
        - Messaging and contact flows linked to the `InterestExpressed` event.
        - Review submission rules (requiring a closed listing before review).
        - Advanced filtering through a `Filter` value object tied to domain requirements.
        - Moderation and reporting workflows.
- Database & Storage (Supabase)
    - The database schema contains several **fully implemented** structures:
        - User profiles (name, email, role, reputation score).
        - Listings (seller ID, title, description, category, size, condition, price/free marker, status).
        - Saved listings associations.
        - Donation center metadata used by the Map page.
    - Reviews and ratings tables exist in the schema but **full review logic is planned**, not yet implemented.
    - Supabase Storage manages image uploads and retrieval in production.
    - Supabase Auth is fully functional for authentication and secure access.
- External Services
    - Leaflet.js provides the implemented base map functionality for donation center locations.
    - Optional geocoding APIs are part of the **planned** extension for address-to-coordinate conversion.

Architecture Diagram

The architecture of the application is designed to separate concerns between the frontend, backend logic, and Supabase services. This diagram is included because it clarifies how implemented and planned components fit into the system's layered design and how data moves between layers.



1. Architecture Overview
    a. React Frontend
        - Implements the primary user-facing mechanisms for browsing, filtering, and saving listings.
        - User actions such as creating a listing or saving an item trigger calls to implemented backend functions.
        - Planned UI components (e.g., messaging, enhanced search tools) will integrate with the same flow.

b. JavaScript Logic (Backend Logic)
   - Implements input validation, storage coordination, and basic CRUD operations.
   - Formats data before insertion into the database and handles error responses.
   - Planned logic will extend this layer to handle events such as expressing interest, review submission, and improved filtering.

c. Supabase Services
   - Implemented database tables store active listings, profiles, saved relationships, and donation centers.
   - Cloud Storage handles implemented image-upload flows.
   - Auth enforces authentication and authorization, ensuring proper access control.
   - Planned enhancements include RLS-based constraints for messaging and review linking.

d. Data Flow
   - Requests flow from the React frontend through JavaScript logic to Supabase.
   - Responses update React state so users see the most recent data.
   - The platform **does not** process payments or facilitate mediated transactions; offline coordination is intentional and domain-driven.
   - Separating UI, logic, and persistence ensures maintainability and future extensibility.

Layer Mapping (Clean Architecture / DDD Alignment)

- **Domain Layer**: Core concepts (Listing, Piece, User, Review), invariants (required fields, category assignment), and event definitions.
- **Application Layer**: Implemented use cases (publishListing, saveListing), and planned use cases (expressInterest, closeListing, review).
- **Interface/Adapters Layer**: React components for forms, lists, profile, and prototype messaging UIs; adapters for calling application services.
- **Frameworks/Drivers Layer**: Supabase services (PostgreSQL, Storage, Auth), Leaflet maps, geocoding APIs, and HTTP clients.

2.3.1 Selected Fragments of the Implementation

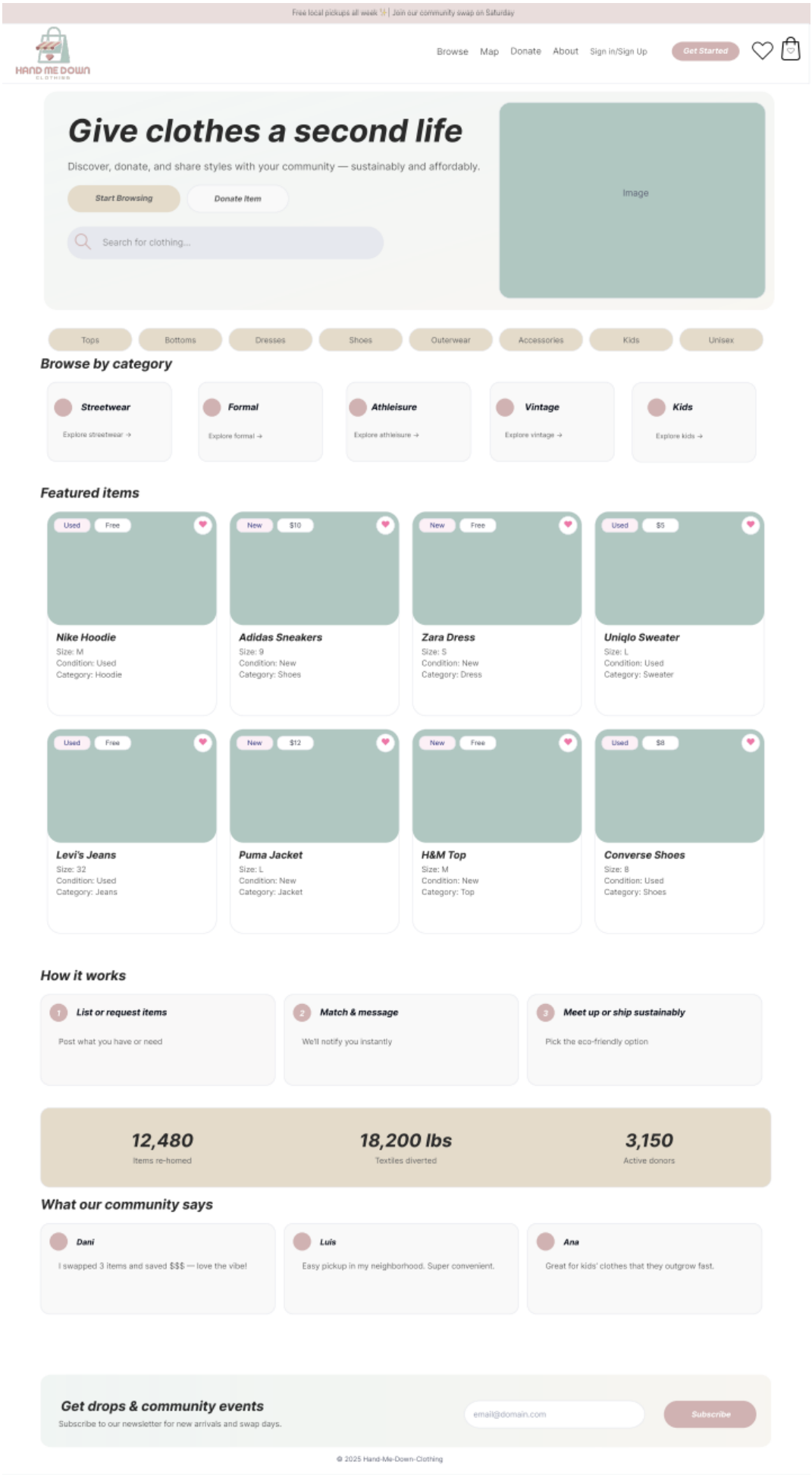
This section provides selected fragments of the implementation that complement the architecture described in Section 2.3. To comply with project guidelines, the fragments are organized into:

1. **Implemented Fragments** – Functionality already built and connected to Supabase.
2. **Planned Fragments** – UI prototypes that illustrate intended behavior or domain mappings but are not yet implemented.

Screenshots are used **only when they illuminate architectural decisions, domain flows, or state transitions**, not as user-manual illustrations.

Implemented Fragments

Home Page (Implemented UI Fragment)



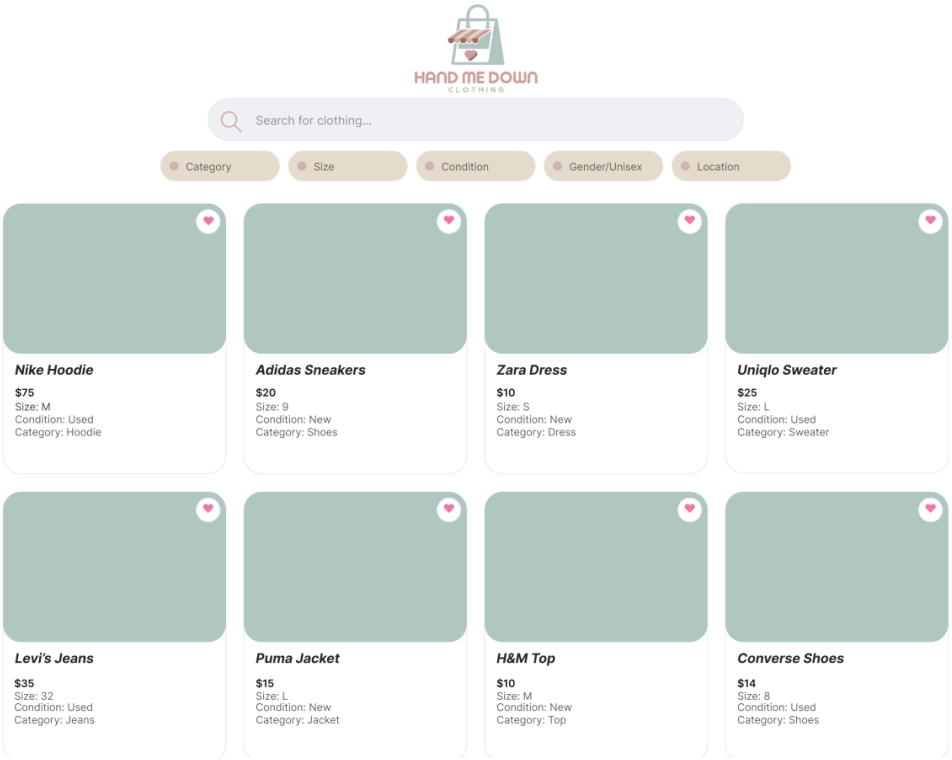
Caption: This screenshot demonstrates the implemented binding between React state and

Supabase listing data. It is included because it shows how the Home Page retrieves the most recent active listings and displays them as part of the implemented browsing flow.

The implemented Home Page fragment: - fetches active listings from Supabase, - displays them in a scrollable feed, and - provides navigation to other implemented features (e.g., listing creation, browsing feed).

This fragment demonstrates how the frontend consumes application-level domain functions such as `browseAvailableItems`.

Clothing Listing Page (Implemented UI Fragment)

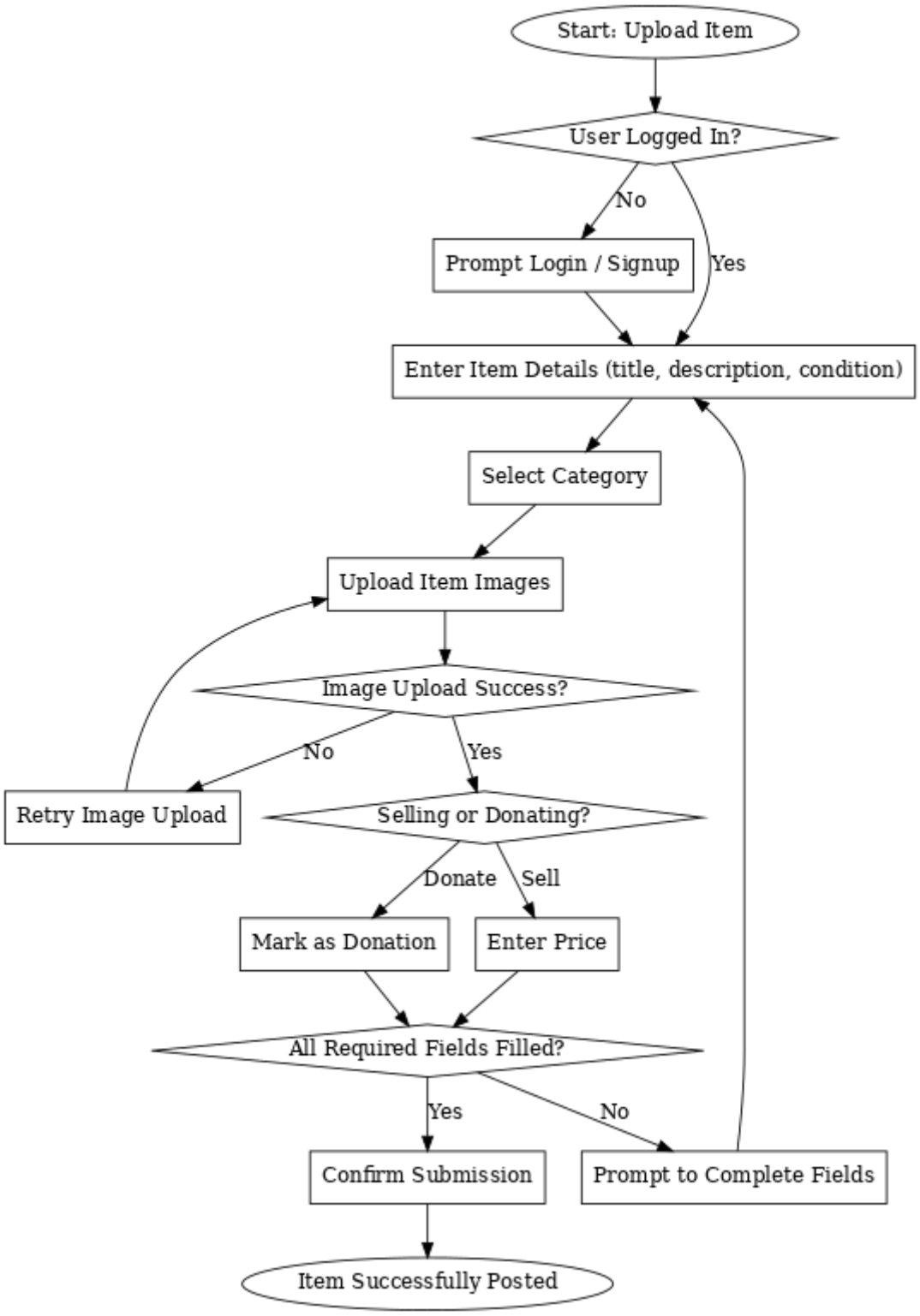


Caption: Included to show the operational flow from a Supabase "active listings" query into a React-rendered list component. This directly reflects the domain's browsing functions.

This implemented component: - loads available listings using Supabase filters, - displays listing metadata (title, image, free/resale marker), and - supports client-side sorting.

These elements correspond directly to domain actions such as filtering and browsing available items.

Item Upload Flow – Implemented Logic



Caption: Flowchart included to illustrate the implemented control flow from the React form to Supabase Storage and Database. This visualization explains the domain-level `publishListing` action and its mapping to the current backend logic.

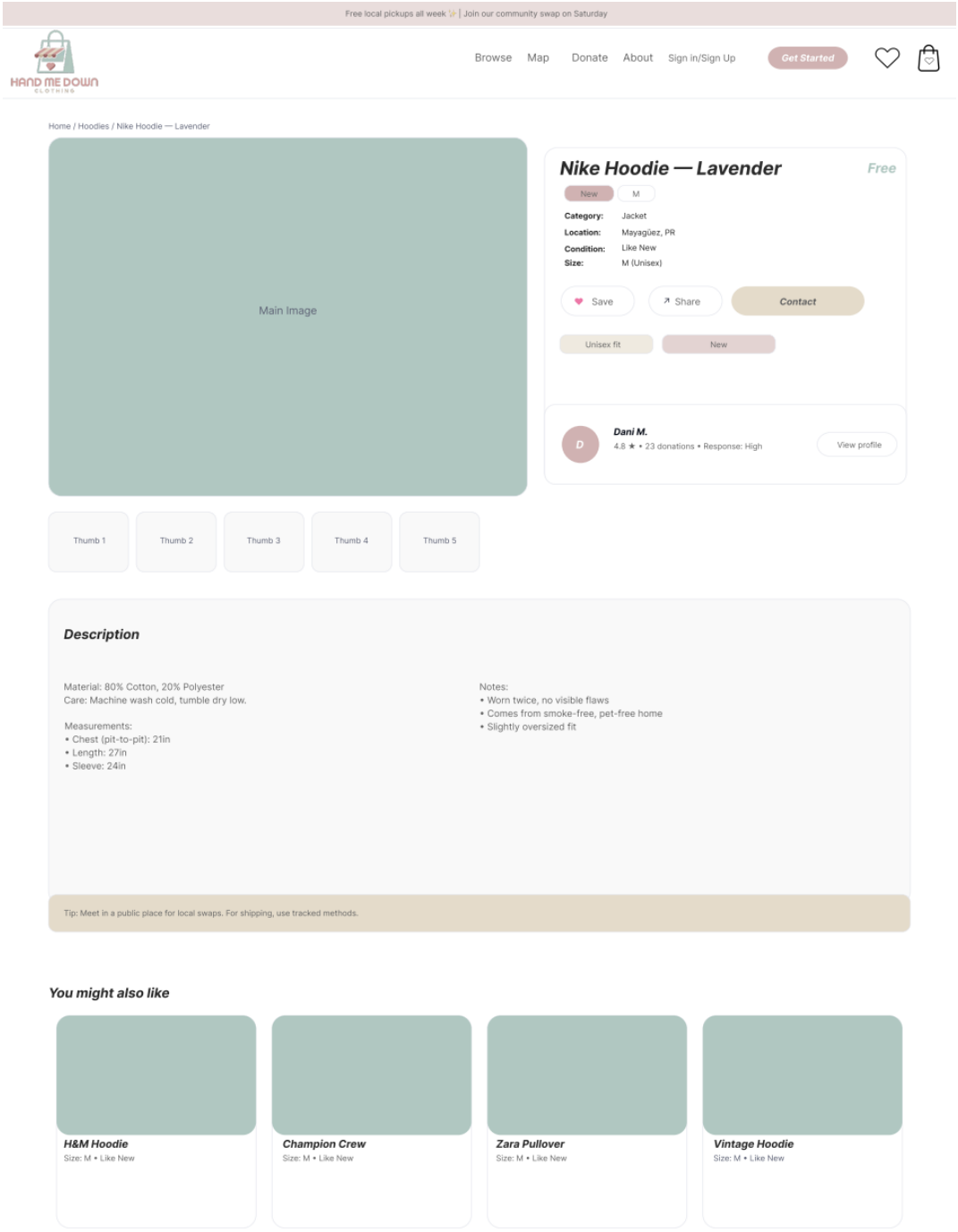
Implemented behaviors include: 1. Authenticated user opens the listing creation form. 2. Required fields validated client-side. 3. Images uploaded to Supabase Storage. 4. Listing inserted into the database with `status = active`.

This flow enforces domain invariants but **does not** create or manage transactions.

Planned Fragments

(These pages are not yet implemented; they serve to document intended architectural fit. No user-manual behavior is described.)

Item Detail Page (Planned UI Fragment)

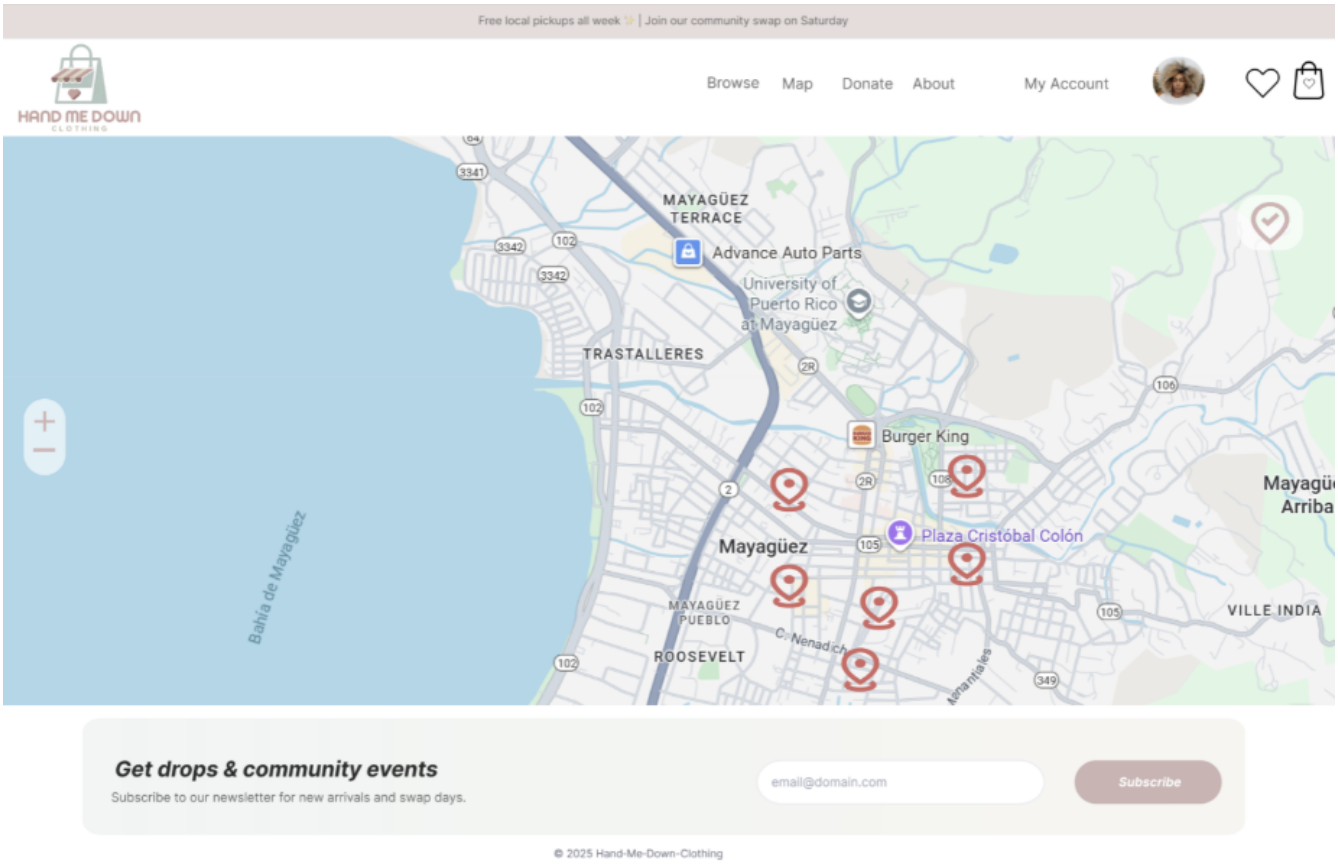


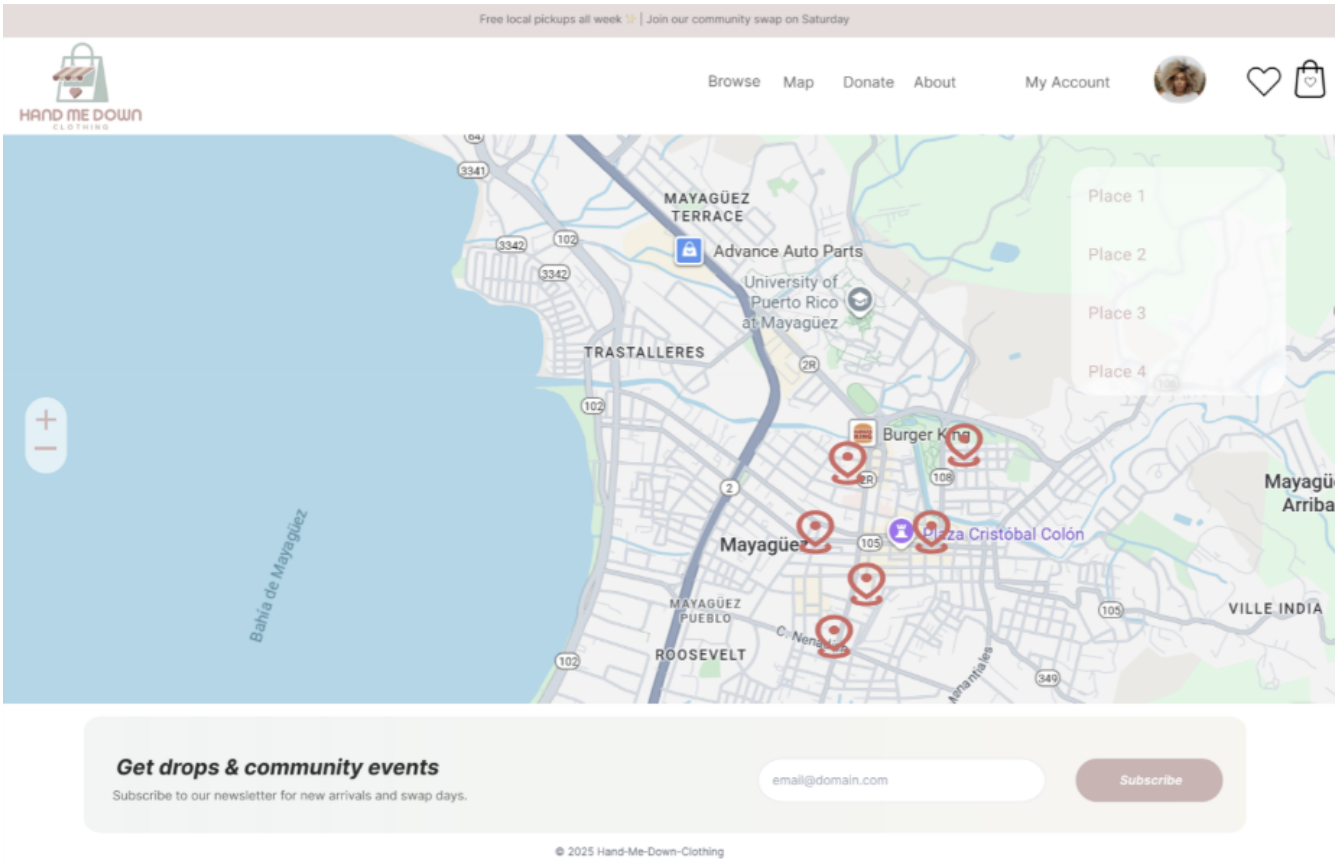
Caption: This prototype is included because it illustrates how domain entities (Listing, Seller, Review summary) will map to a unified detail view once fully implemented.

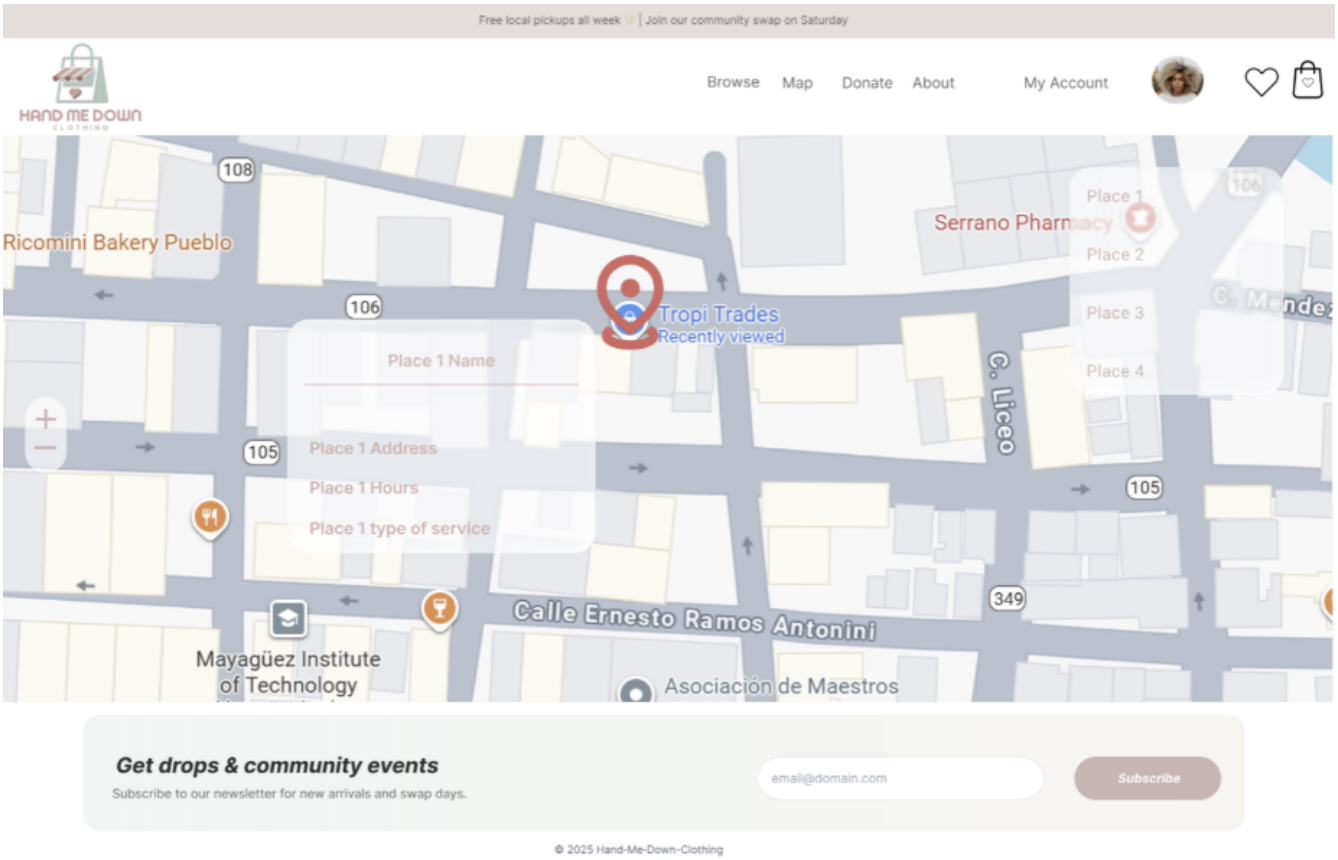
This planned view will integrate: - listing metadata, - seller information, - interest/contact actions tied to the domain event `InterestExpressed`.

No backend logic exists yet.

Map Page (Planned UI Fragment)







Caption: These prototypes illustrate the architectural role of the map: integrating Supabase-stored locations with Leaflet. They are included only to show planned frontend–database interactions, not to document UI behavior.

The map will integrate: - Leaflet for geospatial rendering, - Supabase for donation-center metadata, - optional geocoding services.

This fragment is architectural only; full functionality is not implemented.

Summary

This section now distinguishes implemented behavior from planned prototypes and uses screenshots strictly to illustrate architectural or domain-relevant flows. All user-manual descriptions have been removed, aligning the documentation with Milestone 3 expectations.

2.3.2 Application of Techniques

This subsection presents the concrete techniques applied by the team during Milestone 3, demonstrating how the project evolved from scattered, duplicated logic to modular, intention-revealing, and maintainable structures. The examples below come from real improvements implemented by the Map, Auth, and Backend teams. Each example includes a before/after transformation aligned with Domain-Driven Design principles and the architectural rules described in Section 2.2 Architecture & Layering.

Specification Pattern: From Scattered Conditionals to Composable Rules

Before

Filtering logic in the Map and Listing domains was originally written using inline conditionals inside controllers, leading to duplication and tight coupling. Each filter—such as “Near Me,” “Open Now,” or “Category”—was evaluated manually:

```
function filterResults(listings, filters) {
  return listings.filter(listing => {
    let ok = true;

    if (filters.nearMe) {
      const dist = calculateDistance(listing.location, filters.userLocation);
      if (dist > filters.radiusMeters) ok = false;
    }

    if (filters.openNow) {
      const hour = new Date().getHours();
      if (!(listing.openingHour <= hour && hour <= listing.closingHour)) {
        ok = false;
      }
    }

    if (filters.category) {
      if (listing.category !== filters.category) ok = false;
    }

    return ok;
  });
}
```

Problems:

- Controller-level business logic.
- Repeated conditions in multiple files.
- Adding a new filter required editing several modules.
- Hard to test filtering rules independently.
- No clear mapping to the Filter Value Object documented in the domain model.

After

Filters were refactored into **Specification** objects, each exposing a single, meaningful rule through a shared interface:

```
export interface Specification<T> {
  isSatisfiedBy(candidate: T): boolean;
```

```
}
```

Examples of concrete Specifications:

```
export class NearMeSpecification implements Specification<Listing> {
  constructor(private userLocation, private radiusMeters: number) {}

  isSatisfiedBy(listing: Listing): boolean {
    const dist = calculateDistance(listing.location, this.userLocation);
    return dist <= this.radiusMeters;
  }
}

export class OpenNowSpecification implements Specification<Listing> {
  constructor(private currentHour: number) {}

  isSatisfiedBy(listing: Listing): boolean {
    return (
      listing.openingHour <= this.currentHour &&
      this.currentHour <= listing.closingHour
    );
  }
}
```

Specifications can be composed:

```
export class AndSpecification<T> implements Specification<T> {
  constructor(
    private left: Specification<T>,
    private right: Specification<T>
  ) {}

  isSatisfiedBy(candidate: T): boolean {
    return (
      this.left.isSatisfiedBy(candidate) &&
      this.right.isSatisfiedBy(candidate)
    );
  }
}
```

The call site becomes clean and intention-revealing:

```
const specs: Specification<Listing>[] = [];

if (filterVO.nearMe) {
  specs.push(new NearMeSpecification(userLoc, filterVO.radius));
}
```

```
if (filterVO.openNow) {
  specs.push(new OpenNowSpecification(currentHour));
}

const result = listings.filter(listing =>
  specs.every(spec => spec.isSatisfiedBy(listing))
);
```

Benefits:

- Encapsulated filtering rules instead of scattered conditionals.
- Easy to add new filters without modifying controllers.
- Clear mapping to the Filter VO.
- Fully testable, composable domain rules.

Strategy Pattern: Interchangeable Routing Providers

Before

Routing was tightly coupled to Leaflet's OSRMv1 provider. Controllers directly instantiated and configured the OSRM router:

```
async function getRoute(origin, destination) {
  const router = L.Routing.osrmv1({
    serviceUrl: "https://router.project-osrm.org/route/v1",
  });

  return new Promise((resolve, reject) => {
    router.route([origin, destination], (err, routes) => {
      if (err) reject(err);
      else resolve(routes[0]);
    });
  });
}
```

Problems:

- Hard-coded provider (OSRMv1).
- No ability to switch to other routing engines (e.g., Google Maps, Mapbox, custom).
- Difficult to test routing behavior in isolation from Leaflet.
- Violated the Open–Closed Principle: adding a new routing policy required changing existing controller code.

After

A routing abstraction was introduced following the Strategy Pattern taught in lecture:

```
export interface RoutingStrategy {
  getRoute(origin: LatLng, destination: LatLng): Promise<RouteResult>;
}
```

Current concrete strategy:

```
export class OSRMRoutingStrategy implements RoutingStrategy {
  private router = L.Routing.osrmv1({
    serviceUrl: "https://router.project-osrm.org/route/v1",
  });

  async getRoute(origin: LatLng, destination: LatLng): Promise<RouteResult> {
    return new Promise((resolve, reject) => {
      this.router.route([origin, destination], (err, routes) => {
        if (err) reject(err);
        else resolve(routes[0]);
      });
    });
  }
}
```

Planned strategies (documented as extension points, not yet implemented):

- `GoogleMapsRoutingStrategy`
- `MapboxRoutingStrategy`
- `CustomShortestPathRoutingStrategy`

Controller usage becomes independent of the concrete routing engine:

```
const routingStrategy: RoutingStrategy = new OSRMRoutingStrategy();
const route = await routingStrategy.getRoute(origin, destination);
```

Benefits:

- Decouples controllers from a specific routing provider.
- Enables future routing engines by implementing the same interface.
- Simplifies testing by allowing mock strategies.
- Aligns with the Strategy Pattern as presented in the course and with the project's extensibility goals.

Intention-Revealing Interfaces: Centralizing Authentication Logic

Before

Each protected page manually performed Supabase authentication and profile retrieval, mixing concerns and duplicating logic:

```
import { supabase } from "@/lib/supabaseClient";

export async function load({ redirect }) {
  const { data: { user }, error: userError } = await supabase.auth.getUser();
  if (!user || userError) {
    throw redirect(302, "/login");
  }

  const { data: profile, error: profileError } = await supabase
    .from("profiles")
    .select("*")
    .eq("id", user.id)
    .single();

  if (profileError) {
    throw redirect(302, "/login");
  }

  return { user, profile };
}
```

Problems:

- Authentication and profile logic duplicated in many routes.
- Slight variations in profile queries between pages.
- Harder to maintain consistent behavior when auth rules change.
- The main intention (“get authenticated user with profile or redirect”) was obscured by boilerplate.

After

The Auth team introduced intention-revealing helpers, matching the lecture’s emphasis on services with clear intent and side-effect-free functions where possible:

```
export async function getAuthUserWithProfile(supabase) {
  const { data: { user }, error: userError } = await supabase.auth.getUser();
  if (!user || userError) {
    return { user: null, profile: null };
  }

  const { data: profile, error: profileError } = await supabase
```

```
    .from("profiles")
    .select("*")
    .eq("id", user.id)
    .single();

  if (profileError) {
    return { user: null, profile: null };
  }

  return { user, profile };
}
```

At the call site, the logic becomes shorter and intention-revealing:

```
const { user, profile } = await getAuthUserWithProfile(supabase);
if (!user) {
  throw redirect(302, "/login");
}

return { user, profile };
```

For reactive components, a hook centralizes subscription to auth state:

```
export function useSupabaseAuth() {
  const supabase = useSupabase();
  const auth = ref({ user: null, loading: true });

  supabase.auth.onAuthStateChange((_event, session) => {
    auth.value = {
      user: session?.user ?? null,
      loading: false,
    };
  });

  return auth;
}
```

Typical usage:

```
const auth = useSupabaseAuth();

if (auth.value.loading) {
  return "Loading...";
}

if (!auth.value.user) {
  router.push("/login");
```

```
}
```

Benefits:

- **Intention-revealing**: the purpose of each helper is clear at the call site.
- **Centralized logic**: changes to authentication or profile retrieval occur in one place.
- **Consistency**: all pages rely on the same profile shape and redirect behavior.
- **Testability**: helpers can be tested separately from page code.
- **Alignment with lectures**: matches the emphasis on side-effect-free, intention-revealing services at boundaries.

Assertions and Domain Invariants

As discussed in the **Supple Design** lecture (slides 25–31), assertions help document and enforce the invariants that must always hold before and after domain operations. They make assumptions explicit, reduce ambiguity, and support the correctness of aggregates and services. In this project, assertions are incorporated into key operations across the Authentication, Backend, and Map subsystems to guarantee stability and prevent invalid states.

Authentication Operation: signUp(firstname, lastname, email, password)

The `signUp` flow creates a new user within the authentication subsystem. Assertions clarify the conditions under which the operation is valid and the guarantees it provides afterward.

Preconditions - `email` is syntactically valid. - `password` meets the security policy (≥ 8 characters, includes number or symbol). - `email` is not already associated with an existing account. - `firstname` and `lastname` are non-empty strings.

Postconditions - A new User account exists with the provided email. - The password is securely hashed (never stored in plain text). - A valid session or authentication credential is created. - The new user is considered authenticated immediately after sign-up.

This mirrors the lecture guidance that authentication workflows benefit from explicit pre- and post-conditions to avoid ambiguous states (e.g., “user created but not authenticated”).

Backend Operation: createPiece(pieceData)

The `createPiece` operation instantiates a new `Piece` entity and enforces domain rules regarding categories, conditions, and ownership.

Preconditions - The seller initiating the operation is authenticated and valid. - `category` must match one of the allowed domain categories. - Provided attributes must be compatible with the category (e.g., shoes must include `shoeSize`; accessories must not include a size field). - `condition` □ {new, like-new, lightly-used, used, heavily-used}. - `description` length does not exceed the domain’s maximum.

Postconditions - A new Piece object exists and is associated with the seller. - The Piece receives a unique `pieceID`. - All fields (`category`, `condition`, `attributes`, `description`) match the validated input. - The Piece enters the `unlisted` state and is not part of any Listing yet.

These assertions document the invariant that a Piece must always enter the system in a valid, unlisted, domain-compliant state.

Map Operation: nearMe(listings, userLocation)

The `nearMe` operation returns all listings within a fixed radius (8km) of the specified user location or map-selected point. Assertions ensure geospatial correctness and consistent results.

Preconditions - `userLocation` contains valid latitude and longitude values. - `listings` is a non-null, iterable collection. - Each listing must include a valid meetup location or seller coordinate.

Postconditions - Returned listings are only those within 8km of `userLocation`. - No listing outside the radius appears in the result. - Ordering of listings is preserved unless explicitly sorted afterward. - The function has no side effects (does not modify listings or persist data).

By capturing the geospatial invariants explicitly, the Map subsystem aligns with the lecture principle that system behavior should never rely on hidden assumptions.

Summary

These assertions clarify the system's behavioral expectations and match the lecture recommendations for maintaining model integrity through explicit invariants. Each subsystem—Auth, Backend, and Map—uses assertions to ensure operations transition between valid states, prevent malformed inputs, and reinforce domain rules. Together, they contribute to a more predictable, verifiable, and maintainable design.

Command–Query Separation (CQS) and Side-Effect-Free Validators

Command–Query Separation (CQS) was applied in the Auth subsystem to distinguish **pure, side-effect-free validation** from **mutating commands** that change authentication state or persist data. This refactor simplifies reasoning about behavior, improves testability, and aligns with the course emphasis on supple design.

Before: Mixed Validation and Side Effects in a Single Flow

Originally, several Auth-related flows combined **input validation**, **Supabase calls**, and **navigation/redirects** into a single function. A typical "sign up" handler both validated inputs and performed mutations:

```
export async function handleSignUp(formData, supabase, router) {
  const email = formData.get("email") as string;
  const password = formData.get("password") as string;
  const firstname = formData.get("firstname") as string;
  const lastname = formData.get("lastname") as string;

  // Inline validation
  if (!email || !email.includes("@")) {
    throw new Error("Invalid email.");
  }
  if (!password || password.length < 8) {
```

```
    throw new Error("Password too short.");
  }

  // Auth + profile creation (side effects)
  const { data, error } = await supabase.auth.signUp({
    email,
    password,
    options: {
      data: { firstname, lastname },
    },
  });

  if (error) {
    throw error;
  }

  router.push("/dashboard");
}
```

Problems:

- Queries (validation) and commands (mutations) were mixed together.
- The function had multiple responsibilities: validation, auth side effects, and navigation.
- Unit testing required mocking Supabase and routing even when only validation was under test.
- Intent was blurred: the **what** (validate vs. create) and the **how** (Supabase, router) were tangled.

After: Pure Validators + Explicit Command Handlers

The Auth Team refactored validation concerns into **pure functions** that take data and return validation results without side effects. Mutating logic was isolated in dedicated command handlers.

Pure validators (queries):

```
export function validateEmail(email: string): string[] {
  const errors: string[] = [];
  if (!email) {
    errors.push("Email is required.");
  } else if (!email.includes("@")) {
    errors.push("Email must contain '@'.");
  }
  return errors;
}

export function validatePassword(password: string): string[] {
  const errors: string[] = [];
  if (!password || password.length < 8) {
    errors.push("Password must be at least 8 characters.");
  }
  if (!/\d|\W/.test(password)) {
```

```
    errors.push("Password must include a number or symbol.");
  }
  return errors;
}

export function validateProfileFields(firstname: string, lastname: string): string[] {
  const errors: string[] = [];
  if (!firstname.trim()) {
    errors.push("First name is required.");
  }
  if (!lastname.trim()) {
    errors.push("Last name is required.");
  }
  return errors;
}
```

These functions:

- Have **no side effects** (no Supabase calls, no router calls).
- Always return the same output for the same input.
- Are easy to unit-test in isolation.

Command handler (mutation):

```
export async function handleCreateAccount(formData, supabase, router) {
  const email = formData.get("email") as string;
  const password = formData.get("password") as string;
  const firstname = formData.get("firstname") as string;
  const lastname = formData.get("lastname") as string;

  const errors = [
    ...validateEmail(email),
    ...validatePassword(password),
    ...validateProfileFields(firstname, lastname),
  ];

  if (errors.length > 0) {
    return { ok: false, errors };
  }

  const { data, error } = await supabase.auth.signUp({
    email,
    password,
    options: {
      data: { firstname, lastname },
    },
  });

  if (error) {
```

```
    return { ok: false, errors: [error.message] };
  }

  router.push("/dashboard");
  return { ok: true, errors: [] };
}
```

Here:

- The **command** (`handleCreateAccount`) is clearly responsible for **doing things**:
- calling Supabase (auth side effect),
- triggering navigation.
- All **query-style logic** (validation) is delegated to pure functions.
- The handler's return structure (`{ ok, errors }`) makes behavior explicit and testable.

Benefits of the CQS Refactor

- **Clear Separation of Responsibilities** Validators answer questions ("Is this input valid?") without changing state, while commands modify state or trigger navigation.
- **Improved Testability** Pure validators can be tested without mocking Supabase or routing. Command handlers can be tested with targeted integration or mocking.
- **Reduced Side Effects** Fewer functions can unexpectedly cause persistence or navigation; side effects are localized to command functions.
- **Easier Reasoning and Maintenance** Developers can quickly see where mutations happen and reuse validators across multiple forms or flows.
- **Alignment with Course Guidance** The refactor demonstrates Command–Query Separation by drawing a firm line between "asking" (validators) and "doing" (commands), while keeping domain rules explicit and intention-revealing.

Additional CQS Example: signIn (Command) vs getUser (Query)

The Auth subsystem also illustrates Command–Query Separation through a clear distinction between **commands** that change authentication state and **queries** that only read it. Here we document the evolution of the login flow from a mixed command/query function into a clean split between `signIn` (command) and `getUser` (query).

======= Before: Login Flow Mixing Command and Query Responsibilities

Initially, a single function handled both:

- reading current authentication state
- performing a login attempt (mutation)
- deciding navigation behavior

```
export async function handleSignIn(formData, supabase, router) {
```

```
  const email = formData.get("email") as string;
  const password = formData.get("password") as string;

  const { data: { user: existingUser } } = await supabase.auth.getUser();
  if (existingUser) {
    router.push("/dashboard");
    return;
  }

  const { data, error } = await supabase.auth.signInWithPassword({
    email,
    password,
  });

  if (error) {
    throw new Error(error.message);
  }

  router.push("/dashboard");
}
```

Problems:

- **Mixed responsibilities**: querying current user and performing sign-in in the same function.
- **Harder to reuse**: any flow that only needed current user information still depended on login logic.
- **More complex tests**: tests had to mock both the query (`getUser`) and the command (`signInWithPassword`) for this one function.
- **Weaker intent**: it was not obvious from the API which parts **read** state and which parts **mutated** it.

======= After: Explicit Query (`getAuthUser`) and Command (`signInCommand`)

The logic was refactored into:

- a **query** function `getAuthUser` that only reads current authentication state
- a **command** function `signInCommand` that performs sign-in and produces side effects

Query (no side effects):

```
export async function getAuthUser(supabase) {
  const { data: { user }, error } = await supabase.auth.getUser();

  if (error) {
    return { user: null, error };
  }

  return { user, error: null };
```

```
}
```

Command (mutation + navigation):

```
export async function signInCommand(
  email: string,
  password: string,
  supabase,
  router
) {
  const { data, error } = await supabase.auth.signInWithPassword({
    email,
    password,
  });

  if (error) {
    return { ok: false, error: error.message };
  }

  router.push("/dashboard");
  return { ok: true, error: null };
}
```

The form handler then uses the query and command explicitly:

```
export async function handleSignIn(formData, supabase, router) {
  const email = formData.get("email") as string;
  const password = formData.get("password") as string;

  const { user } = await getAuthUser(supabase);
  if (user) {
    router.push("/dashboard");
    return;
  }

  const result = await signInCommand(email, password, supabase, router);
  if (!result.ok) {
    throw new Error(result.error ?? "Unable to sign in.");
  }
}
```

Benefits of this CQS refactor:

- **Explicit intent**: `getAuthUser` clearly communicates that it only **reads** the current user, while `signInCommand` clearly **acts** to change authentication state.
- **Improved reuse**: `getAuthUser` can be reused anywhere current user information is needed without pulling in sign-in logic.

- **Easier testing**: query and command logic can be unit-tested independently with focused mocks.
- **Consistent CQS story**: together with the side-effect-free validators and command handlers documented earlier, this example strengthens the project's adherence to Command–Query Separation in the Auth subsystem.

Stand-Alone Auth Provider: Centralizing Supabase SDK Dependency

To reduce coupling and follow the "stand-alone class" guidance from the Supple Design lecture, the Auth Team refactored how components access authentication. Instead of each component depending directly on the Supabase Auth SDK, all SDK usage is now centralized in a stand-alone provider (`SupabaseAuthProvider`) and a stable hook (`useSupabaseAuth`). This section documents the before/after transformation and how it reduces dependencies.

Before: Components Depending Directly on Supabase Auth SDK

Originally, multiple components imported and used the Supabase client directly. Each component was responsible for calling the SDK, handling session changes, and interpreting authentication state.

Example (simplified):

```
import { supabase } from "@/lib/supabaseClient";
import { onMounted, ref } from "vue";

export default {
  setup() {
    const user = ref(null);
    const loading = ref(true);

    onMounted(async () => {
      const { data: { user: currentUser } } = await supabase.auth.getUser();
      user.value = currentUser ?? null;
      loading.value = false;

      supabase.auth.onAuthStateChange((_event, session) => {
        user.value = session?.user ?? null;
      });
    });

    async function handleSignOut() {
      await supabase.auth.signOut();
      user.value = null;
    }

    return { user, loading, handleSignOut };
  },
};
```

Problems:

- Each component depended directly on the Supabase Auth SDK.
- Auth state subscription logic was duplicated across components.
- Changing auth provider (or how sessions are handled) required editing many files.
- Testing components required mocking the Supabase client in multiple places.
- The intent (“I need auth state”) was obscured by low-level SDK calls.

After: SupabaseAuthProvider and useSupabaseAuth as Stand-Alone Abstractions

The team introduced a stand-alone provider component and a shared hook to encapsulate SDK calls. Components no longer depend on the Supabase SDK; instead, they depend only on a stable interface defined by `useSupabaseAuth`.

Provider component (stand-alone class/component):

```
import { supabase } from "@/lib/supabaseClient";
import { defineComponent, provide, ref, onMounted } from "vue";

const AuthSymbol = Symbol("AuthContext");

export const SupabaseAuthProvider = defineComponent({
  name: "SupabaseAuthProvider",
  setup(_, { slots }) {
    const user = ref(null);
    const loading = ref(true);

    onMounted(async () => {
      const { data: { user: currentUser } } = await supabase.auth.getUser();
      user.value = currentUser ?? null;
      loading.value = false;

      supabase.auth.onAuthStateChange((_event, session) => {
        user.value = session?.user ?? null;
      });
    });

    async function signOut() {
      await supabase.auth.signOut();
      user.value = null;
    }

    const value = { user, loading, signOut };

    provide(AuthSymbol, value);

    return () => slots.default ? slots.default() : null;
  },
});

export function useSupabaseAuth() {
```

```
  const ctx = inject(AuthSymbol);
  if (!ctx) {
    throw new Error("useSupabaseAuth must be used within SupabaseAuthProvider.");
  }
  return ctx;
}
```

Component usage after refactor:

```
import { useSupabaseAuth } from "@/auth/SupabaseAuthProvider";

export default {
  setup() {
    const { user, loading, signOut } = useSupabaseAuth();

    return {
      user,
      loading,
      handleSignOut: signOut,
    };
  },
};
```

Key changes:

- All Supabase Auth SDK calls (getUser, onAuthStateChange, signOut) are centralized in `SupabaseAuthProvider`.
- Components depend only on the stable stand-alone abstraction `useSupabaseAuth`.
- If the project changes auth providers or session handling, only the provider implementation must be updated.

Diagram: Stand-Alone Auth Provider Reducing Dependencies

```
@startuml
interface AuthInterface {
  +user
  +loading
  +signOut()
}

class SupabaseAuthProvider {
  -supabaseClient
  +provide(AuthInterface)
}

class ComponentA
class ComponentB
```

```
SupabaseAuthProvider ..> AuthInterface
ComponentA --> AuthInterface : uses via useSupabaseAuth()
ComponentB --> AuthInterface : uses via useSupabaseAuth()

AuthInterface <|.. SupabaseAuthProvider
@enduml
```

Dependency Reduction and Benefits

- **Reduced Dependencies:** Components no longer import or configure the Supabase client directly; they depend only on `useSupabaseAuth`, a stand-alone abstraction.
- **Centralized SDK Usage:** All low-level SDK calls are located in a single provider file, making changes to the auth backend or session behavior localized.
- **Improved Testability:** Components can be tested by mocking `useSupabaseAuth` instead of mocking the entire Supabase client. This yields simpler, more targeted tests.
- **Clearer Intent:** At the component level, the code now expresses intent ("I need auth state and signOut") rather than implementation details ("call supabase.auth.getUser() and manage listeners").
- **Alignment with Milestone 3 and Lectures:** The `SupabaseAuthProvider` behaves as a stand-alone class that shields the rest of the system from low-level dependencies, satisfying the Milestone 3 requirement for dependency reduction and reflecting the stand-alone class guidance from the Supple Design lecture.

Unresolved directive in sections/section2/section.adoc - include::subsections/2.3.3.adoc[]

Unresolved directive in sections/section2/section.adoc - include::subsections/2.3.4.adoc[]

Unresolved directive in sections/section2/section.adoc - include::subsections/2.4.adoc[]

Unresolved directive in sections/section2/section.adoc - include::subsections/2.5.adoc[]

Unresolved directive in sections/section2/section.adoc - include::subsections/2.5.1.adoc[]

Unresolved directive in sections/section2/section.adoc - include::subsections/2.5.2.adoc[]

3.1 Concept Formation and Analysis

This section documents how the domain concepts were derived from the raw observations in **2.1.1 – Domain Rough Sketch** and the everyday practices described in **2.1.4 – Domain Narrative**. It explains how concrete, context-specific details were abstracted into entities, events, roles, value objects, actions, and behaviors. The goal is to show a clear and verifiable path from observation to domain model.

From Observation to Abstraction

The rough sketch stories revealed a recurring structure across different individuals: someone makes an item visible to others, potential takers discover it, a brief conversation follows, and the

parties meet in person to complete the exchange. Although the exact tools vary (group chats, social media posts, direct messaging), the underlying phenomena remain consistent.

Several recurring patterns emerged:

- People show items using posts or photos that function as temporary “public signals.”
- Interested parties initiate contact with short, informal messages asking if the item is still available.
- Agreement is reached by clarifying small details (size, condition, pickup spot).
- The exchange occurs at a familiar physical location.
- After the handoff, the item is no longer advertised, and the original post is often updated or removed.
- Positive or negative experiences influence future interactions through informal reputation.

Because these behaviors appeared across multiple narratives, the team abstracted them into stable domain concepts rather than anecdotal exceptions.

Consolidating Entities

The physical objects (“shirts,” “blazer,” “backpack,” etc.) were generalized into the entity **Piece**, representing any clothing item circulating in the community.

In contrast, the posts, messages, or shared images that made these items visible were abstracted into **Listings**—representational artifacts distinct from the Pieces they describe. This distinction was necessary because Listings have their own lifecycle: they can appear, be updated, be ignored, or be removed independently of the Piece.

This separation mirrors what happens in real exchanges: the representation moves through visibility states while the underlying item simply exists.

Identifying Events and Actions

Certain moments in the narratives corresponded to discrete transitions:

- when an item becomes publicly visible
- when someone expresses interest
- when the availability of the item changes

These transitions were abstracted into **Events** such as **Listing Published**, **Interest Expressed**, and **Listing Closed**. Each event marks an instantaneous change observed in participant behavior.

The human activities that lead to these events—posting an item, contacting someone, removing a post—became **Actions**. This separation allowed the model to differentiate between **what people do** and **what changes occur in the domain** as a result.

Behavioral Norms and Implicit Practices

The rough sketches contained implicit social rules that shape exchanges even though they are rarely stated explicitly.

Several of these norms recurred across different stories:

- **Condition Disclosure Norm** — participants often show wear, defects, or tags before exchanging items.
- **Meetup Practice** — exchanges consistently occur in familiar, visible locations, often near the UPRM campus.
- **Trust Cues** — people rely on conversational tone, mutual acquaintances, bilingual communication, or recognizable names.
- **Student Resale Price Band** — symbolic or low-cost pricing appeared repeatedly in student-to-student transactions.
- **Exchange Flow** — the behavioral pattern "share → discover → contact → meet" formed a recognizable structure across all accounts.
- **Reuse Loop** — items sometimes re-enter circulation after a period of use or storage, showing that clothing may undergo multiple listing cycles.

These implicit behaviors were made explicit to accurately reflect the lived domain and to ensure future requirements align with local social practices rather than imposing new ones.

Roles, Settings, and Value Objects

Observations showed that participants shift roles depending on context. Someone who gives away an item one week may seek another the next. These shifting positions were abstracted as the roles **Buyer** and **Seller**, not as fixed categories of people.

Exchanges consistently took place at physical locations such as campus entries, bus stops, or parking lots. These stable spatial anchors were captured as **Settings** under the term **Locale**.

Repeated references to size, type, and condition formed the basis for **Value Objects** such as **Type**, **Condition Rating**, and other descriptive attributes that help people assess suitability quickly.

Iterative Refinement

As concepts solidified, the team rechecked them against the narratives to ensure that each abstraction described something observable without assuming any future technical system. Concepts related to visibility, interest, trust, representations, and reuse were reshaped until they represented domain truths rather than system intentions.

This iterative process ensured that the domain model remained faithful to real-world behaviors while providing a structured foundation for the requirements and design work in Section 2.2.

In summary, the concept formation process bridges concrete stories and abstract structures. It documents how repeated behaviors, roles, events, and norms were distilled into a coherent domain model that guides subsequent architectural and design decisions.

3.2 Validation and Verification

Validation and verification ensure that the Hand Me Down platform meets stakeholder needs, maintains internal consistency across modules, and satisfies measurable quality standards defined in the requirements. These activities are carried out continuously and collaboratively across all sub-teams—Documentation & Requirements, Authentication, Listings, Map/Search, and UI/UX— to preserve traceability from stakeholder needs through implementation and testing.

Validation

Domain and Requirements Validation

Validation of terminology and requirements was performed iteratively through internal documentation reviews. Each update to §§ 2.1 – 2.3 was examined to confirm that domain concepts such as **Piece**, **Listing**, **Condition Disclosure Norm**, and **Listing Closed** remained coherent and aligned with stakeholder needs (§ 1.2.2). All requirements now employ definitive "shall/must" statements instead of uncertain language ("may," "aims") to ensure they are directly testable. No external stakeholder validation sessions have yet occurred; these will take place in later milestones to confirm usability and trust cues with students and families.

Scenario Walkthroughs

Walkthroughs were conducted for the **publishing** process, tracing data flow from form submission to Supabase storage. No inconsistencies were found. Editing, closing, and saving workflows are planned for final-phase validation once their implementations are complete.

Category and Condition Refinement

Internal validation led to refinement of the category taxonomy and condition-rating scales, resolving ambiguities from the initial model. Terminology was standardized to **Donated Piece** and **Sold Piece** to ensure semantic consistency across documentation and database layers.

Search and Map Validation

The Map & Search module was validated by cross-checking UI results against the actual data stored in Supabase. Each search query correctly displayed only listings matching its attributes, and filter walkthroughs confirmed accurate behavior for "Tops," "Bottoms," and "All." Usability was validated through manual inspection: map markers are accurately pinned, display complete popup information (name, address, hours), and maintain expected cluster and static behaviors. Search relevance and marker accuracy confirm correct data binding between UI and repository functions.

Authentication Validation

Supabase Auth integration was validated by confirming that users can **sign up, log in, and log out** successfully without confusion. The authentication flow follows Supabase and OWASP recommendations for secure web login. Manual tests were executed in Google Chrome, Firefox, and Brave on desktop devices. Error messages were displayed properly when login, signup, or logout failed. Mobile testing is scheduled for the next milestone.

UI/UX Validation

Informal validation was performed through manual walkthroughs and internal demos. Team members and classmates interacted with the interface to identify confusing or redundant

elements and provided feedback on button labeling and visibility of listing-creation steps. Adjustments were applied iteratively, improving label clarity and layout alignment. Accessibility was validated manually through color-contrast and tab-navigation checks. Primary text and buttons met readability standards, and form elements followed a logical tab order. Semantic HTML was verified for buttons and labels, while full ARIA labeling and automated accessibility audits are planned for the next milestone.

Planned Stakeholder Validation

A short validation session with student volunteers will be scheduled before the final milestone to evaluate clarity of interface terminology, filter usability, and trust indicators such as condition labels and safety guidance.

Verification

Traceability and Acceptance Criteria

A preliminary **Need → Requirement → Test** mapping exists conceptually and will be formalized in `/docs/tests/traceability.adoc` for milestone 3. Each requirement includes measurable acceptance conditions: – Interface Requirements define button-state validation, inline error messaging, and toast feedback. – Machine Requirements specify response-time thresholds (≤ 2 s average; ≤ 4 s peak) and uptime ≥ 99.7 %. – Domain Requirements link directly to test cases verifying classification and visibility logic.

Unit Testing

Manual unit tests were completed for **publishListing()**, validating data verification, database insertion, and frontend feedback. All manual tests passed successfully. Additional automated unit tests for **closeListing()** and **editListing()** are scheduled for the next milestone. Authentication unit tests validated correct sign-up, login, and logout behavior, while error handling displayed expected feedback messages when failures occurred.

Integration Testing

End-to-end tests confirmed correct UI-to-database behavior: information entered in the listing form propagates through backend validation and is stored in Supabase as expected. Integration with Authentication (user ID linkage) will be completed in the final milestone. The Search module's repository functions (`PieceRepository.getPieces()` and `filterPieces()`) were validated indirectly through accurate data synchronization between Supabase queries and the rendered results. Authentication privacy constraints were verified through **role-based access control** using Supabase **Row Level Security (RLS)** policies implemented in issue #301. Interface behaviors—such as disabled Publish buttons until form validation passes, visible toast messages, and navigation flow correctness—were manually verified against the Interface Requirements. Visual consistency was confirmed across browsers.

Load and Performance Testing

Preliminary manual observations show average search responses in ≈ **1 second** and listing creation times under **0.5 seconds**—both within the defined machine-requirement limits. Formal automated load testing using **k6** will be added to simulate concurrent usage (150 – 500 users) and confirm scalability benchmarks. Authentication latency and API response times will also be measured in the next milestone.

Data Validation and Security Checks

Map-coordinate rendering logic filters out invalid or non-finite latitude/longitude values, preventing off-map markers. All markers are non-draggable, ensuring location data remains immutable in the UI. RLS policies in Supabase protect user records by restricting read/write access based on authentication state and role. UI components were visually validated across major browsers (Chrome, Edge) to ensure consistent layout, iconography, and branding defined in the global style guide.

Continuous Verification

A GitHub Actions workflow will execute linting and unit-test jobs on pull requests to maintain consistent quality and prevent regressions once automated tests are in place.

End-to-End Feature Verification

The following table summarizes the implemented features across subsystems.

Subsystem	Implemented Features	Source
Authentication (Team 2)	Signup, login, logout, persistent sessions, profile editing, protected routes	Team 2 Auth Summary
Backend (Team 3)	Create Piece, Create Listing, Publish Listing, Close Listing (transactional), Saved Listings, Filtered Listing Retrieval	Team 3 Backend Summary
Map (Team 4)	Map rendering, marker clustering, routing providers, modals, location info, route preview	Team 4 Map Summary
UI (Team 5)	Home → Browse → Listing → Interest flow, Auth screens, Profile screens, Map screens, Saved Listings UI, loading/error states	Team 5 UI Summary

The following acceptance tests demonstrate end-to-end behavior across multiple subsystems. All tests use the recommend Given–When–Then (GWT).

Authentication Flow

Test A1 – Successful Login

Given the user has a valid email and password **When** they submit the login form **Then** the system returns an authenticated session **And** the user is redirected to the Home screen **And** protected content becomes accessible

Test A2 – Session Restoration

Given a previously authenticated session exists **When** the app is reopened **Then** the Auth subsystem restores the session **And** the user is taken directly to Home without re-entering credentials

Test A3 – Profile Update

Given the user is authenticated **When** they update their profile information **Then** the backend persists the new profile **And** the UI reflects the updated data

Listing Lifecycle

Test L1 – Create → Publish Listing

Given the seller is authenticated **And** they have created a Piece **When** they create a Listing and publish it **Then** the backend stores the Listing **And** it appears in the browse screen **And** Map markers update accordingly

Test L2 – Close Listing (Transactional)

Given a Listing is active **When** the seller closes the Listing **Then** the backend transitions the Listing atomically to a closed state **And** it no longer appears in active browsing **And** the UI shows the closed status on the seller's Listing screen

Test L3 – Save a Listing

Given a buyer is authenticated **When** they save a listing **Then** it appears under Saved Listings **And** the backend reflects the saved state

Map Interaction

Test M1 – Load Map With Markers

Given the user opens the Map screen **When** the Map provider initializes **Then** all active Listings are shown as markers **And** marker clustering activates for dense areas

Test M2 – View Location Info Modal

Given markers are visible **When** the user taps a marker **Then** a modal appears with Listing details **And** route options become available

Test M3 – Start Route Preview

Given a Listing location is selected **When** the user chooses a routing mode **Then** the routing provider builds a path **And** the route preview renders on the Map

UI Navigation Flows

Test U1 – Home → Browse → Listing

Given the user is authenticated **When** they navigate to Browse **And** select a Listing **Then** the Listing detail screen appears **And** the UI renders all Listing data returned by the backend

Test U2 – Browse → Filters → Results

Given the Browse screen is open **When** the buyer applies valid filters **Then** the backend returns matching Listings **And** the screen updates with the filtered results

Test U3 – Saved Listings Screen

Given the user has saved Listings **When** they navigate to Saved Listings **Then** all saved items appear with correct metadata **And** tapping one navigates to its detail page

Outcomes

– Documentation, domain model, and requirements were aligned and validated through internal review cycles. – Listing-publication backend passed all manual unit tests, achieving < 0.5 s creation time. – Search and map functionalities were validated against Supabase data, loading results in ≈ 1 s on average. – Map markers were verified for nine sample donation centers. – Authentication features (sign up, login, logout) were validated across major desktop browsers with secure RLS policies. – UI elements and flows passed internal usability and accessibility checks; no critical issues were reported. – Category and condition-rating systems were refined for accuracy and uniformity. – Traceability structure and automated CI testing are established for completion in the final milestone.

Together, these validation and verification activities confirm that the system concepts are sound, the current implementation behaves as specified, and measurable criteria are in place to ensure the platform remains reliable, scalable, and aligned with stakeholder expectations as development continues.

Logbook

Person	Sections worked on
1uismar33r0	2.3.1 (#148)
Alma-pineiro	2.3.1 (#148)
JoshDG03	1.3.1 (#190), 2.1 (#190, #214), 2.1.1 (#190, #214), 2.1.2 (#214), 2.1.3 (#214), 2.1.4 (#214, #216), 2.1.5 (#182, #190, #214, #217), 2.1.6 (#190, #214), 2.2.1 (#190, #214), 2.2.2 (#214), 2.2.3 (#214), 2.2.4 (#190, #214), 2.2.5 (#190, #214, #242), 2.3 (#190, #214), 2.3.1 (#214), 3.2 (#242)
JuanIranzo	2.1.6 (#218)
Ojani	2.3 (#247), 2.3.1 (#247), 3.2 (#246)
angelvillegas1	2.2.5 (#242)