Brrow

Complete System Documentation

Generated on October 08, 2025

Table of Contents

📘 Brrow - Complete System Documentation

> ***Version 1.3.4** | Last Updated: October 8, 2025 **Author**: Claude Code (Anthropic) **Owner**: Shalin Ratna*

🎨 Document Theme

Colors: Blue (#007AFF), White (#FFFFFF), Green (#34C759) **Style**: Professional, Clean, Easy to Navigate

📑 Table of Contents

Part 9: Xcode & iOS Build

Part 10: Maintenance & Updates

Part 1: System Overview

Executive Summary

Brrow is a peer-to-peer marketplace iOS application that enables users to buy, sell, and rent items within their local community. The platform features real-time messaging, secure payments via Stripe, identity verification through ID.me, and a sophisticated seller payout system using Stripe Connect.

Platform Status

- **iOS Version**: 1.3.4 (Build 594)
- **Backend Version**: 1.3.4
- **Production URL**: https://brrow-backend-nodejs-production.up.railway.app
- **Platform**: iOS 16+
- **Database**: PostgreSQL (Railway)
- **Users**: 143 active
- **Listings**: 7 active
- **Status**: ✅ Fully Operational

Architecture Overview

```
┌─────────────────────────────────────────────────────────┐
│                    iOS Application                       │
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐  │
│  │ SwiftUI Views│  │ ViewModels   │  │ API Client   │  │
│  └──────────────┘  └──────────────┘  └──────────────┘  │
└────────────────────────┬────────────────────────────────┘
                         │ HTTPS/JSON
                         ▼
┌─────────────────────────────────────────────────────────┐
│              Node.js Backend (Railway)                   │
│  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐  │
│  │ Express.js   │  │ Prisma ORM   │  │ WebSockets   │  │
│  └──────────────┘  └──────────────┘  └──────────────┘  │
└────────────────────────┬────────────────────────────────┘
                         │
         ┌───────────────┼───────────────┐
         ▼               ▼               ▼
┌──────────────┐ ┌──────────────┐ ┌──────────────┐
│ PostgreSQL   │ │   Stripe     │ │ Firebase     │
│  Database    │ │  Payments    │ │  Services    │
└──────────────┘ └──────────────┘ └──────────────┘
```

Data Flow

1. **User Action** → iOS App
2. **API Request** → Backend (via HTTPS)
3. **Authentication** → JWT Token Validation
4. **Database Query** → PostgreSQL via Prisma
5. **Response** → JSON back to iOS
6. **UI Update** → SwiftUI re-renders

Technology Stack

iOS Stack

Component	Technology	Version	Purpose
Language	Swift	5.9+	App Development
UI Framework	SwiftUI	iOS 16+	User Interface
Networking	URLSession	Native	API Communication
Image Loading	SDWebImage	5.19.7	Async Image Loading
Real-time	Socket.IO	16.1.0	Live Messaging
Payments	Stripe iOS SDK	23.29.4	Payment Processing
Auth	Firebase Auth	10.29.0	Authentication
Push	Firebase Messaging	10.29.0	Notifications
Maps	MapKit	Native	Location Services

Backend Stack

Component	Technology	Version	Purpose
Runtime	Node.js	18.20.8	Server Runtime
Framework	Express.js	4.x	HTTP Server
Database ORM	Prisma	6.16.2	Database Access
Database	PostgreSQL	15	Data Storage
Real-time	Socket.IO	Latest	WebSocket Server
Payments	Stripe SDK	Latest	Payment Processing
Images	Cloudinary SDK	Latest	Image Storage
Security	Helmet.js	Latest	Security Headers
Logging	Pino	Latest	Structured Logging

Infrastructure

Component	Provider	Purpose
Backend Hosting	Railway	Node.js Server
Database	Railway (PostgreSQL)	Data Persistence
Image Storage	Cloudinary	User Images
Payment Processing	Stripe	Payments & Payouts
Push Notifications	Firebase	iOS Notifications
Identity Verification	ID.me	User Verification

Key Features

✅ Implemented Features

Core Marketplace

- ☑ Browse listings with infinite scroll
- ☑ Category-based filtering
- ☑ Search functionality
- ☑ Location-based discovery
- ☑ Listing detail views
- ☑ Image galleries
- ☑ Seller profiles

- ✓ Favorites/Saved items

Listing Management

- ✓ Create listings (sale/rent)
- ✓ Upload multiple images
- ✓ Edit listings
- ✓ Delete listings
- ✓ Pricing options
- ✓ Availability status
- ✓ GPS location tagging

Payments

- ✓ Stripe Checkout integration
- ✓ Buy Now (direct purchase)
- ✓ Secure payment processing
- ✓ Transaction history
- ✓ Stripe Connect (seller payouts)
- ✓ Payment holds
- ✓ Automatic releases

Messaging

- ✓ Real-time chat (Socket.IO)
- ✓ Conversation threads
- ✓ Message notifications
- ✓ Unread counts
- ✓ Chat search

User Management

- ✓ Registration (Email/Google/Apple)
- ✓ JWT Authentication

- [x] Profile customization
- [x] Display name & username
- [x] Bio & location
- [x] Profile pictures
- [x] ID.me verification

Social Features

- [x] User profiles
- [x] Seller ratings
- [x] Favorites system
- [x] Follow system
- [x] Activity feed

🚧 Partially Implemented

- [ ] Offers system (backend ready, UI incomplete)
- [ ] Reviews & ratings (backend ready, UI incomplete)
- [ ] Garage sales (backend ready, UI minimal)
- [ ] Analytics dashboard (backend ready, UI incomplete)

❌ Not Implemented

- [ ] In-app video calls
- [ ] Voice messages
- [ ] Product recommendations
- [ ] Referral system

Part 2: Backend Systems

Backend Architecture

File Structure

```
brrow-backend/
├── prisma-server.js          # Main server (9,981 lines)
├── config/
|   ├── production.js          # Production config
|   ├── database.js            # Prisma client factory
|   └── logger.js              # Pino logging
├── middleware/
|   ├── rateLimiter.js         # Rate limiting
|   ├── memoryOptimization.js # Memory management
|   └── authentication.js      # JWT validation
├── services/
|   ├── websocket.service.js   # Socket.IO
|   ├── emailService.js        # Email sending
|   ├── analyticsService.js    # Analytics
|   └── monitoringService.js   # Health monitoring
├── routes/
|   ├── search.js              # Search endpoints
|   ├── admin.js               # Admin panel
|   └── analytics.js           # Analytics routes
├── utils/
|   ├── responseTransformers.js # iOS response format
|   └── urlValidator.js        # URL validation
├── prisma/
|   └── schema.prisma          # Database schema
├── settings-system.js         # User settings
├── stripe-connect-insurance.js # Stripe Connect
├── blocked-users.js          # User blocking
└── google-oauth.js           # Google OAuth
```

Server Startup Process

1. Database Connection

```
// Create optimized Prisma client with connection pooling
const prisma = createOptimizedPrismaClient();

// Explicitly connect to database
await prisma.$connect();
logger.info('Database connected successfully');
```

2. Middleware Setup

```
// Security
app.use(helmet());

// Compression
app.use(compression());

// Rate limiting
app.use(rateLimiter.globalLimiter());
app.use(rateLimiter.burstLimiter());

// Memory optimization
app.use(memoryMonitor(400)); // 400MB alert
app.use(connectionCleanup());
app.use(responseSizeLimiter(5 * 1024 * 1024)); // 5MB max
```

3. Routes Registration

```
// Core routes
app.get('/api/listings', listingsEndpoint);
app.post('/api/auth/register', registerEndpoint);
app.get('/api/users/me', authenticateToken, getUserProfile);

// External modules
settingsSystem(app, prisma, authenticateToken);
stripeConnect(app, prisma, authenticateToken);
blockedUsers(app, prisma, authenticateToken);
```

4. WebSocket Initialization

```
webSocketService.initialize(httpServer);
```

5. Background Tasks

```
// Stats broadcasting
startStatsBroadcasting();

// Garbage collection
scheduleGarbageCollection();

// Connection cleanup
scheduleConnectionCleanup(prisma);

// Memory monitoring
setInterval(() => logMemoryUsage(), 60000);
```

Database Schema

Core Tables

users

```
CREATE TABLE users (
  id TEXT PRIMARY KEY,
  api_id TEXT UNIQUE NOT NULL,
  username TEXT UNIQUE NOT NULL,
  display_name TEXT,
  email TEXT UNIQUE NOT NULL,
  password TEXT, -- bcrypt hashed
  first_name TEXT,
  last_name TEXT,
  phone_number TEXT,
  phone_verified BOOLEAN DEFAULT false,
  profile_picture_url TEXT,
  bio TEXT,
  location JSONB,
  date_of_birth TIMESTAMP,
  is_verified BOOLEAN DEFAULT false,
  average_rating DECIMAL(3,2),
  total_ratings INTEGER DEFAULT 0,
  is_creator BOOLEAN DEFAULT false,
  creator_tier TEXT,
  stripe_account_id TEXT, -- For sellers
  stripe_customer_id TEXT, -- For buyers
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW(),
  last_login_at TIMESTAMP,
  last_username_change TIMESTAMP
);
```

listings

```sql
CREATE TABLE listings (
  id TEXT PRIMARY KEY,
  title TEXT NOT NULL,
  description TEXT NOT NULL,
  price DECIMAL(10,2) NOT NULL,
  daily_rate DECIMAL(10,2),
  pricing_type TEXT, -- 'sale' or 'rent'
  category_id TEXT REFERENCES categories(id),
  user_id TEXT REFERENCES users(id),
  condition TEXT,
  location JSONB,
  tags TEXT[],
  metadata JSONB,
  view_count INTEGER DEFAULT 0,
  favorite_count INTEGER DEFAULT 0,
  is_active BOOLEAN DEFAULT true,
  is_premium BOOLEAN DEFAULT false,
  availability_status TEXT DEFAULT 'AVAILABLE',
  moderation_status TEXT DEFAULT 'PENDING',
  created_at TIMESTAMP DEFAULT NOW(),
  updated_at TIMESTAMP DEFAULT NOW()
);
```

listing_images

```
CREATE TABLE listing_images (
  id TEXT PRIMARY KEY,
  listing_id TEXT REFERENCES listings(id),
  image_url TEXT NOT NULL,
  thumbnail_url TEXT,
  is_primary BOOLEAN DEFAULT false,
  display_order INTEGER DEFAULT 0,
  has_gps_data BOOLEAN DEFAULT false,
  latitude DECIMAL(10,8),
  longitude DECIMAL(11,8),
  uploaded_at TIMESTAMP DEFAULT NOW()
);
```

purchases

```
CREATE TABLE purchases (
  id TEXT PRIMARY KEY,
  buyer_id TEXT REFERENCES users(id),
  seller_id TEXT REFERENCES users(id),
  listing_id TEXT REFERENCES listings(id),
  amount DECIMAL(10,2) NOT NULL,
  payment_status TEXT, -- PENDING, COMPLETED, FAILED, REFUNDED
  verification_status TEXT, -- PENDING, VERIFIED, DISPUTED
  payment_intent_id TEXT, -- Stripe payment intent
  stripe_session_id TEXT, -- Stripe checkout session
  purchase_type TEXT, -- BUY_NOW, RENTAL, OFFER
  created_at TIMESTAMP DEFAULT NOW(),
  deadline TIMESTAMP, -- 7 days for verification
  completed_at TIMESTAMP,
  verified_at TIMESTAMP
);
```

messages

```sql
CREATE TABLE messages (
  id TEXT PRIMARY KEY,
  conversation_id TEXT NOT NULL,
  sender_id TEXT REFERENCES users(id),
  recipient_id TEXT REFERENCES users(id),
  message TEXT NOT NULL,
  message_type TEXT DEFAULT 'text', -- text, image, offer
  is_read BOOLEAN DEFAULT false,
  created_at TIMESTAMP DEFAULT NOW(),
  read_at TIMESTAMP
);
```

favorites

```sql
CREATE TABLE favorites (
  id TEXT PRIMARY KEY,
  user_id TEXT REFERENCES users(id),
  listing_id TEXT REFERENCES listings(id),
  created_at TIMESTAMP DEFAULT NOW(),
  UNIQUE(user_id, listing_id)
);
```

Relationships

```
users (1) ──< (many) listings
users (1) ──< (many) purchases (as buyer)
users (1) ──< (many) purchases (as seller)
listings (1) ──< (many) listing_images
listings (1) ──< (many) purchases
listings (1) ──< (many) favorites
categories (1) ──< (many) listings
users (1) ──< (many) messages (as sender)
users (1) ──< (many) messages (as recipient)
```

API Endpoints

Authentication Endpoints

POST /api/auth/register

Purpose: Create new user account

Request Body:

```json
{
  "email": "user@example.com",
  "password": "securePassword123",
  "username": "johndoe",
  "firstName": "John",
  "lastName": "Doe"
}
```

Response:

```
{
  "success": true,
  "message": "Registration successful",
  "user": {
    "id": "cmf123...",
    "username": "johndoe",
    "email": "user@example.com",
    "apiId": "cmf123..."
  },
  "token": "eyJhbGciOiJIUzI1NiIs...",
  "refreshToken": "eyJhbGciOiJIUzI1NiIs..."
}
```

Status Codes:

- `201` : User created successfully
- `400` : Invalid input or email already exists
- `500` : Server error

POST /api/auth/login

Purpose: Authenticate user

Request Body:

```
{
  "email": "user@example.com",
  "password": "securePassword123"
}
```

Response:

```json
{
  "success": true,
  "message": "Login successful",
  "user": {
    "id": "cmf123...",
    "username": "johndoe",
    "email": "user@example.com"
  },
  "token": "eyJhbGciOiJIUzI1NiIs...",
  "refreshToken": "eyJhbGciOiJIUzI1NiIs..."
}
```

Status Codes:

- `200` : Login successful
- `401` : Invalid credentials
- `500` : Server error

Listings Endpoints

GET /api/listings

Purpose: Get all active listings

Query Parameters:

- `page` (number, default: 1)
- `limit` (number, default: 20, max: 100)
- `category` (string, optional)
- `minPrice` (number, optional)
- `maxPrice` (number, optional)
- `status` (string, default: 'active')
- `user_id` (string, optional) - Filter by user

Response:

```json
{
  "success": true,
  "data": {
    "listings": [
      {
        "id": "listing-id-123",
        "title": "iPhone 15 Pro",
        "description": "Brand new, sealed",
        "price": 999,
        "categoryId": "electronics",
        "userId": "user-id-456",
        "condition": "NEW",
        "location": {
          "city": "San Francisco",
          "state": "CA",
          "zipCode": "94102"
        },
        "images": [
          {
            "id": "img-1",
            "imageUrl": "https://res.cloudinary.com/...",
            "isPrimary": true,
            "displayOrder": 0
          }
        ],
        "user": {
          "id": "user-id-456",
          "username": "johndoe",
          "profile_picture_url": "https://..."
        },
        "category": {
          "id": "electronics",
          "name": "Electronics",
          "icon_url": "📱"
        }
      }
    ],
    "pagination": {
      "total": 50,
      "page": 1,
      "per_page": 20,
      "total_pages": 3,
      "has_more": true
    }
  }
}
```

GET /api/listings/:id

Purpose: Get single listing details

Response:

```
{
  "id": "listing-id-123",
  "title": "iPhone 15 Pro",
  "description": "Detailed description...",
  "price": 999,
  "dailyRate": null,
  "pricingType": "sale",
  "condition": "NEW",
  "viewCount": 42,
  "favoriteCount": 7,
  "isActive": true,
  "availabilityStatus": "AVAILABLE",
  "images": [...],
  "user": {...},
  "category": {...},
  "createdAt": "2025-10-01T12:00:00Z"
}
```

POST /api/listings

Purpose: Create new listing (requires authentication)

Headers:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIs...
```

Request Body:

```json
{
  "title": "iPhone 15 Pro",
  "description": "Brand new, sealed in box",
  "price": 999,
  "pricingType": "sale",
  "categoryId": "electronics",
  "condition": "NEW",
  "location": {
    "city": "San Francisco",
    "state": "CA",
    "zipCode": "94102",
    "latitude": 37.7749,
    "longitude": -122.4194
  },
  "images": [
    "https://res.cloudinary.com/brrow/image/upload/v.../image1.jpg",
    "https://res.cloudinary.com/brrow/image/upload/v.../image2.jpg"
  ],
  "tags": ["smartphone", "apple", "ios"],
  "deliveryOptions": {
    "pickup": true,
    "delivery": false,
    "shipping": true
  }
}
```

Response:

```
{
  "success": true,
  "message": "Listing created successfully",
  "listing": {
    "id": "new-listing-id",
    "title": "iPhone 15 Pro",
    ...
  }
}
```

Status Codes:

- `201` : Listing created
- `400` : Invalid input
- `401` : Not authenticated
- `500` : Server error

User Endpoints

GET /api/users/me

Purpose: Get current user profile (requires authentication)

Response:

```json
{
  "success": true,
  "user": {
    "id": "cmf123...",
    "email": "user@example.com",
    "username": "johndoe",
    "firstName": "John",
    "lastName": "Doe",
    "displayName": "John D.",
    "profilePictureUrl": "https://...",
    "bio": "Tech enthusiast",
    "phoneNumber": "+1234567890",
    "phoneVerified": true,
    "location": "San Francisco, CA",
    "isVerified": true,
    "averageRating": 4.8,
    "totalRatings": 24,
    "createdAt": "2025-01-15T10:30:00Z"
  }
}
```

PUT /api/users/me

Purpose: Update user profile (requires authentication)

Request Body:

```json
{
  "firstName": "John",
  "lastName": "Doe",
  "displayName": "Johnny",
  "bio": "Updated bio",
  "location": "San Francisco, CA",
  "website": "https://johndoe.com"
}
```

Response:

```json
{
  "id": "cmf123...",
  "username": "johndoe",
  "displayName": "Johnny",
  ...
}
```

Important Notes:

- ✅ **Display name** can be updated anytime
- ⚠️ **Username** can only be changed once every 90 days
- ⚠️ **Phone number** requires SMS verification (use `/api/verify/send-sms`)

Purchase Endpoints

POST /api/purchases

Purpose: Create purchase and Stripe checkout session

Request Body:

```json
{
  "listing_id": "listing-id-123",
  "amount": 999,
  "purchase_type": "BUY_NOW"
}
```

Response:

```json
{
  "success": true,
  "message": "Purchase created - please complete checkout",
  "needsPaymentMethod": true,
  "checkoutUrl": "https://checkout.stripe.com/c/pay/cs_test_...",
  "sessionId": "cs_test_...",
  "purchase": {
    "id": "purchase-id-789",
    "buyer_id": "user-id-456",
    "seller_id": "user-id-123",
    "listing_id": "listing-id-123",
    "amount": 999,
    "payment_status": "PENDING",
    "verification_status": "PENDING",
    "created_at": "2025-10-08T10:30:00Z",
    "deadline": "2025-10-15T10:30:00Z"
  }
}
```

Flow:

1. Backend creates purchase record
2. Backend creates Stripe checkout session
3. Returns checkout URL
4. iOS app opens URL in SafariViewController
5. User completes payment on Stripe
6. Stripe redirects to `brrowapp://stripe/success`

7. Stripe sends webhook to backend
8. Backend updates purchase status to `COMPLETED`

Authentication & Security

JWT Token System

Access Token (1 hour expiry):

```
{
  "userId": "cmf123...",
  "email": "user@example.com",
  "username": "johndoe",
  "iat": 1633024800,
  "exp": 1633028400
}
```

Refresh Token (30 days expiry):

```
{
  "userId": "cmf123...",
  "type": "refresh",
  "iat": 1633024800,
  "exp": 1635616800
}
```

Authentication Flow

```
1. User logs in
   ↓
2. Backend validates credentials
   ↓
3. Backend generates JWT tokens
   ↓
4. iOS stores tokens in Keychain
   ↓
5. iOS includes token in Authorization header
   ↓
6. Backend validates token on each request
   ↓
7. If expired, iOS uses refresh token
```

Token Storage (iOS)

```swift
// Keychain storage
KeychainHelper.save(token, forKey: "authToken")
KeychainHelper.save(refreshToken, forKey: "refreshToken")

// Usage
let token = KeychainHelper.load(forKey: "authToken")
request.setValue("Bearer \(token)", forHTTPHeaderField: "Authorization")
```

Security Measures

Rate Limiting

```
// Global rate limit: 100 requests per 15 minutes
app.use(rateLimiter.globalLimiter());

// Burst protection: 10 requests per second
app.use(rateLimiter.burstLimiter());

// API-specific limits
/api/auth/login: 5 requests per 15 minutes
/api/listings: 60 requests per minute
/api/messages: 120 requests per minute
```

Security Headers (Helmet.js)

```
{
  contentSecurityPolicy: true,
  xssFilter: true,
  noSniff: true,
  ieNoOpen: true,
  hsts: { maxAge: 31536000 }
}
```

Password Security

```
// Bcrypt with 10 salt rounds
const hashedPassword = await bcrypt.hash(password, 10);

// Password requirements
- Minimum 8 characters
- At least 1 uppercase letter
- At least 1 lowercase letter
- At least 1 number
```

Part 3: iOS Application

iOS Architecture

MVVM Pattern

```
┌──────────────────────────────────────────┐
│              Views (SwiftUI)              │
│  - Declarative UI                         │
│  - Reactive to state changes              │
│  - No business logic                      │
└────────────────┬─────────────────────────┘
                 │ Binds to @Published vars
                 ▼
┌──────────────────────────────────────────┐
│         ViewModels (ObservableObject)     │
│  - Business logic                         │
│  - State management                       │
│  - API calls                              │
│  - Data transformation                    │
└────────────────┬─────────────────────────┘
                 │ Uses
                 ▼
┌──────────────────────────────────────────┐
│            Models (Codable)               │
│  - Data structures                        │
│  - JSON decoding/encoding                 │
│  - Immutable data                         │
└──────────────────────────────────────────┘
```

Project Structure

```
Brrow/
├── Models/
|   ├── Listing.swift
|   ├── User.swift
|   ├── Message.swift
|   └── Purchase.swift
├── ViewModels/
|   ├── MarketplaceViewModel.swift
|   ├── ListingDetailViewModel.swift
|   ├── MyPostsViewModel.swift
|   └── ChatViewModel.swift
├── Views/
|   ├── ProfessionalMarketplaceView.swift
|   ├── ProfessionalListingDetailView.swift
|   ├── ModernCreateListingView.swift
|   ├── EnhancedChatDetailView.swift
|   └── SimpleProfessionalProfileView.swift
├── Services/
|   ├── APIClient.swift
|   ├── AuthenticationService.swift
|   ├── ListingNavigationManager.swift
|   ├── ImageCacheService.swift
|   └── AppOptimizationService.swift
├── Components/
|   ├── BrrowAsyncImage.swift
|   ├── ListingGridCard.swift
|   └── ModernTextFieldComponents.swift
└── BrrowApp.swift (Entry point)
```

Navigation System

ListingNavigationManager

Purpose: Centralized navigation for listing detail views

Key Features:

- ✅ Single source of truth for listing navigation
- ✅ Supports both Listing object and ID-based navigation
- ✅ Handles loading states
- ✅ Works from any view in the app
- ✅ Prevents duplicate sheets

Usage:

```
// Navigate with Listing object
ListingNavigationManager.shared.showListing(listing)

// Navigate with ID (will load listing)
ListingNavigationManager.shared.showListingById("listing-id-123")

// Clear navigation
ListingNavigationManager.shared.clearListing()
```

Implementation:

```swift
class ListingNavigationManager: ObservableObject {
    static let shared = ListingNavigationManager()

    @Published var selectedListing: Listing?
    @Published var showingListingDetail = false
    @Published var pendingListingId: String?
    @Published var isLoadingListing = false

    func showListing(_ listing: Listing) {
        pendingListingId = nil
        selectedListing = listing
        showingListingDetail = true
    }

    func showListingById(_ listingId: String) {
        pendingListingId = listingId
        showingListingDetail = true
        loadListing(id: listingId)
    }
}
```

View Modifier:

```swift
extension View {
    func withUniversalListingDetail() -> some View {
        modifier(UniversalListingDetailModifier())
    }
}

// Usage in root view
var body: some View {
    TabView {
        ...
    }
    .withUniversalListingDetail()
}
```

Important Fix: SwiftUI reuses views by default. To force recreation when listing changes:

```
.sheet(isPresented: $navigationManager.showingListingDetail) {
    if let listing = navigationManager.selectedListing {
        NavigationView {
            ProfessionalListingDetailView(listing: listing)
        }
        .id(listing.listingId) // ← Forces view recreation
    }
}
```

State Management

@StateObject vs @ObservedObject

@StateObject: View owns the object (creates it)

```
struct MarketplaceView: View {
    @StateObject private var viewModel = MarketplaceViewModel()
}
```

@ObservedObject: View observes external object

```
struct ListingCard: View {
    @ObservedObject var viewModel: MarketplaceViewModel
}
```

@Published Properties

ViewModel Example:

```
class MarketplaceViewModel: ObservableObject {
    @Published var listings: [Listing] = []
    @Published var isLoading = false
    @Published var errorMessage: String?
    @Published var selectedCategory: Category?

    func fetchListings() async {
        isLoading = true
        do {
            let response = try await APIClient.shared.fetchListings()
            await MainActor.run {
                self.listings = response.listings
                self.isLoading = false
            }
        } catch {
            await MainActor.run {
                self.errorMessage = error.localizedDescription
                self.isLoading = false
            }
        }
    }
}
```

UI Components

BrrowAsyncImage

Purpose: Optimized async image loading with caching

Features:

- ✅ Automatic Cloudinary URL handling
- ✅ Memory caching via SDWebImage
- ✅ Placeholder support
- ✅ Error handling
- ✅ Retry logic

Usage:

```
BrrowAsyncImage(
    url: listing.primaryImageUrl,
    width: 300,
    height: 300
)
.cornerRadius(12)
```

Implementation:

```swift
struct BrrowAsyncImage: View {
    let url: String?
    let width: CGFloat
    let height: CGFloat

    var body: some View {
        if let imageUrl = processedURL {
            WebImage(url: URL(string: imageUrl))
                .resizable()
                .placeholder { Color.gray.opacity(0.2) }
                .indicator(.activity)
                .transition(.fade(duration: 0.3))
                .scaledToFill()
                .frame(width: width, height: height)
                .clipped()
        } else {
            Color.gray.opacity(0.2)
                .frame(width: width, height: height)
        }
    }
}
```

Part 4: Third-Party Integrations

Stripe Payment System

Setup

Dashboard: https://dashboard.stripe.com

API Keys:

- **Test Mode**:
  - `pk_test_...` (Publishable)
  - `sk_test_...` (Secret)
- **Live Mode**:
  - `pk_live_...` (Publishable)
  - `sk_live_...` (Secret)

Environment Variables (Railway):

```
STRIPE_SECRET_KEY=sk_test_...  # or sk_live_...
STRIPE_PUBLISHABLE_KEY=pk_test_...  # or pk_live_...
STRIPE_WEBHOOK_SECRET=whsec_...
```

Test vs Live Mode

Switching Between Modes:

1. Update Railway variables with test keys → Test mode
2. Update Railway variables with live keys → Live mode
3. Railway auto-restarts (2 minutes)
4. All new transactions use the new mode

⚠️ IMPORTANT: Your System is Currently in LIVE Mode

Based on transaction logs showing `cs_live_...` session IDs, your production system is using **live Stripe keys**. This means:

- ✅ Real payments are being processed
- ✅ Real money is being charged
- ⚠️ You should switch to test mode for development/testing

How to Switch to Test Mode (Step-by-Step):

1. **Get Your Test Keys** from Stripe Dashboard:

- Go to: https://dashboard.stripe.com/test/apikeys
- Copy **Publishable key** (starts with `pk_test_...`)
- Copy **Secret key** (starts with `sk_test_...`)

2. **Update Railway Environment Variables**:
   - Go to: https://railway.app
   - Select service: `brrow-backend-nodejs-production`
   - Click "Variables" tab
   - Find `STRIPE_SECRET_KEY` → Click to edit
   - Paste your test secret key: `sk_test_...`
   - Find `STRIPE_PUBLISHABLE_KEY` → Click to edit
   - Paste your test publishable key: `pk_test_...`
   - Railway will auto-restart (wait 2 minutes)
3. **Verify Test Mode is Active**:
   - Make a test purchase in iOS app
   - Check checkout URL contains: `cs_test_...` (not `cs_live_...`)
   - View transaction in Stripe Test Dashboard (not Live Dashboard)

What Changes in Test Mode:

- ✅ All new checkout sessions use `cs_test_...` prefix
- ✅ All new payments are test transactions (no real charges)
- ✅ All new webhooks are test events
- ✅ Database stores test purchase records
- ✅ Stripe dashboard shows test data only
- ✅ Backend code unchanged (same code works for both modes)
- ✅ iOS app unchanged (just uses whatever backend provides)

What DOESN'T Change:

- ❌ Old live purchase records in database (they stay)
- ❌ Old live transactions in Stripe (they stay)
- ❌ Any code (backend or iOS)

Test Mode Characteristics:

- ✅ No real money is charged - EVER
- ✅ Use test card: `4242 4242 4242 4242`
- ✅ Checkout URLs have `cs_test_...` prefix
- ✅ Transactions visible in Test Dashboard only
- ✅ Database records are test data (safe to delete)
- ✅ Unlimited free testing

Test Cards:

Card Number	Scenario
4242 4242 4242 4242	✅ Success
4000 0000 0000 9995	❌ Insufficient funds
4000 0000 0000 0002	❌ Card declined
4000 0000 0000 0341	⚠️ Requires 3D Secure

Test Card Details (use with any test card):

- Expiry: Any future date (e.g., `12/34`)
- CVC: Any 3 digits (e.g., `123`)
- ZIP: Any 5 digits (e.g., `12345`)
- Name: Any name

Buy Now Flow

Complete Payment Flow:

```
1. User taps "Buy Now" in iOS app
   ↓
2. iOS calls POST /api/purchases
   {
     "listing_id": "...",
     "amount": 999,
     "purchase_type": "BUY_NOW"
   }
   ↓
3. Backend creates purchase record (status: PENDING)
   ↓
4. Backend creates Stripe Checkout Session
   stripe.checkout.sessions.create({
     mode: 'payment',
     line_items: [{
       price_data: {
         currency: 'usd',
         product_data: { name: listing.title },
         unit_amount: amount * 100
       },
       quantity: 1
     }],
     success_url: 'brrowapp://stripe/success?session_id={CHECKOUT_SESSION_ID}',
     cancel_url: 'brrowapp://stripe/cancel'
   })
   ↓
5. Backend returns checkout URL
   {
     "checkoutUrl": "https://checkout.stripe.com/c/pay/cs_test_...",
     "sessionId": "cs_test_...",
     "purchase": { "id": "...", "payment_status": "PENDING" }
   }
   ↓
6. iOS opens checkout URL in SafariViewController
   ↓
7. User enters card info and completes payment on Stripe
   ↓
8. Stripe redirects to: brrowapp://stripe/success?session_id=cs_test_...
```

```
   ↓
9. iOS app handles deep link
   ↓
10. Stripe sends webhook to backend
    POST /api/stripe/webhook
    Event: checkout.session.completed
   ↓
11. Backend verifies webhook signature
   ↓
12. Backend updates purchase:
    payment_status = 'COMPLETED'
    payment_intent_id = pi_...
    stripe_session_id = cs_test_...
   ↓
13. Backend holds funds for 7 days (verification period)
   ↓
14. After 7 days OR buyer confirms:
    - Backend releases payment to seller
    - Updates verification_status = 'VERIFIED'
```

Backend Code (`prisma-server.js`):

```
app.post('/api/purchases', authenticateToken, async (req, res) => {
  const { listing_id, amount, purchase_type } = req.body;

  // Create purchase record
  const purchase = await prisma.purchases.create({
    data: {
      buyer_id: req.userId,
      seller_id: listing.user_id,
      listing_id,
      amount,
      payment_status: 'PENDING',
      verification_status: 'PENDING',
      purchase_type,
      deadline: new Date(Date.now() + 7 * 24 * 60 * 60 * 1000) // 7 days
    }
  });

  // Create Stripe checkout session
  const session = await stripe.checkout.sessions.create({
    mode: 'payment',
    payment_method_types: ['card'],
    line_items: [{
      price_data: {
        currency: 'usd',
        product_data: { name: listing.title },
        unit_amount: Math.round(amount * 100) // Convert to cents
      },
      quantity: 1
    }],
    metadata: {
      purchase_id: purchase.id,
      buyer_id: req.userId,
      seller_id: listing.user_id,
      listing_id
    },
    success_url: 'brrowapp://stripe/success?session_id={CHECKOUT_SESSION_ID}',
    cancel_url: 'brrowapp://stripe/cancel'
  });
  res.status(201).json({
    success: true,
    needsPaymentMethod: true,
    checkoutUrl: session.url,
    sessionId: session.id,
    purchase
  });
});
```

iOS Code (BuyNowHandler.swift):

```
func handleBuyNow(listing: Listing, amount: Double) async throws {
    // 1. Create purchase
    let response = try await APIClient.shared.createPurchase(
        listingId: listing.listingId,
        amount: amount,
        purchaseType: "BUY_NOW"
    )

    // 2. Open Stripe checkout
    if let checkoutUrl = response.checkoutUrl {
        await MainActor.run {
            self.showingCheckout = true
            self.checkoutURL = URL(string: checkoutUrl)
        }
    }
}
```

Webhook Handler:

```
app.post('/api/stripe/webhook', express.raw({type: 'application/json'}), async (req, r
  const sig = req.headers['stripe-signature'];

  try {
    const event = stripe.webhooks.constructEvent(
      req.body,
      sig,
      process.env.STRIPE_WEBHOOK_SECRET
    );

    if (event.type === 'checkout.session.completed') {
      const session = event.data.object;

      // Update purchase
      await prisma.purchases.update({
        where: { id: session.metadata.purchase_id },
        data: {
          payment_status: 'COMPLETED',
          payment_intent_id: session.payment_intent,
          stripe_session_id: session.id
        }
      });

      // Mark listing as pending
      await prisma.listings.update({
        where: { id: session.metadata.listing_id },
        data: { availability_status: 'PENDING' }
      });
    }

    res.json({received: true});
  } catch (err) {
    res.status(400).send(`Webhook Error: ${err.message}`);
  }
});
```

Stripe Connect (Seller Payments)

Purpose

Allows sellers to receive payments directly to their bank accounts.

Setup Flow

1. Seller Clicks "Connect Stripe"

```
// iOS calls backend
try await APIClient.shared.createStripeConnectAccount()
```

2. Backend Creates Connected Account

```
app.post('/api/stripe/connect/create', authenticateToken, async (req, res) => {
  // Create Stripe Connect account
  const account = await stripe.accounts.create({
    type: 'express',
    country: 'US',
    email: user.email,
    capabilities: {
      card_payments: {requested: true},
      transfers: {requested: true}
    },
    metadata: {
      user_id: req.userId,
      username: user.username
    }
  });

  // Save to database
  await prisma.users.update({
    where: { id: req.userId },
    data: { stripe_account_id: account.id }
  });

  // Create onboarding link
  const accountLink = await stripe.accountLinks.create({
    account: account.id,
    refresh_url: 'brrowapp://stripe-connect/refresh',
    return_url: 'brrowapp://stripe-connect/success',
    type: 'account_onboarding'
  });

  res.json({
    success: true,
    onboardingUrl: accountLink.url,
    accountId: account.id
  });
});
```

3. iOS Opens Onboarding URL

```
// Stripe hosts the onboarding form
// User enters bank details, tax info, etc.
// Stripe handles all compliance
```

4. User Completes Onboarding

```
// Stripe redirects to: brrowapp://stripe-connect/success
// iOS handles deep link
// Backend verifies account is fully onboarded
```

Payout Flow

When buyer verifies purchase (after 7 days OR manual confirmation):

```
// Backend automatically transfers funds to seller
const transfer = await stripe.transfers.create({
  amount: Math.round(purchase.amount * 100 * 0.90), // 90% to seller (10% platform fee
  currency: 'usd',
  destination: seller.stripe_account_id,
  transfer_group: purchase.id,
  metadata: {
    purchase_id: purchase.id,
    listing_id: purchase.listing_id,
    buyer_id: purchase.buyer_id
  }
});

await prisma.purchases.update({
  where: { id: purchase.id },
  data: {
    verification_status: 'VERIFIED',
    verified_at: new Date()
  }
});
```

Payout Schedule:

- Stripe automatically pays out to seller's bank
- Default: 2 business days
- Seller can view payouts in Stripe Dashboard

Firebase Services

Firebase Authentication

Providers:

- Email/Password
- Google Sign-In
- Apple Sign-In

iOS Setup:

```swift
// GoogleService-Info.plist in project
// Firebase SDK initialization in BrrowApp.swift
import Firebase

@main
struct BrrowApp: App {
    init() {
        FirebaseApp.configure()
    }
}
```

Google Sign-In Flow:

```swift
func signInWithGoogle() async throws {
    // 1. Get ID token from Google
    guard let idToken = googleUser.authentication.idToken else {
        throw AuthError.noIDToken
    }

    // 2. Send to backend
    let response = try await APIClient.shared.googleSignIn(idToken: idToken)

    // 3. Save tokens
    try KeychainHelper.save(response.token, forKey: "authToken")

    // 4. Update app state
    await MainActor.run {
        self.isAuthenticated = true
    }
}
```

Backend Integration:

```
app.post('/api/auth/google-signin', async (req, res) => {
  const { idToken } = req.body;

  // Verify token with Google
  const ticket = await googleClient.verifyIdToken({
    idToken,
    audience: GOOGLE_CLIENT_ID
  });

  const payload = ticket.getPayload();
  const email = payload.email;

  // Find or create user
  let user = await prisma.users.findUnique({
    where: { email }
  });

  if (!user) {
    user = await prisma.users.create({
      data: {
        email,
        username: generateUsername(payload.name),
        first_name: payload.given_name,
        last_name: payload.family_name,
        profile_picture_url: payload.picture,
        is_verified: payload.email_verified
      }
    });
  }

  // Generate JWT
  const token = jwt.sign(
    { userId: user.id, email: user.email },
    JWT_SECRET,
    { expiresIn: '1h' }
  );
  res.json({ token, user });
});
```

Firebase Cloud Messaging (Push Notifications)

iOS Setup:

```
// Request permission
UNUserNotificationCenter.current().requestAuthorization(options: [.alert, .badge, .sou
    if granted {
        DispatchQueue.main.async {
            UIApplication.shared.registerForRemoteNotifications()
        }
    }
}

// Handle token
func application(_ application: UIApplication, didRegisterForRemoteNotificationsWithDe
    // Convert to string
    let token = deviceToken.map { String(format: "%02.2hhx", $0) }.joined()

    // Send to backend
    Task {
        try await APIClient.shared.updateFCMToken(token)
    }
}
```

Backend Storage:

```
app.put('/api/users/me/fcm-token', authenticateToken, async (req, res) => {
  const { fcmToken } = req.body;

  await prisma.users.update({
    where: { id: req.userId },
    data: { fcm_token: fcmToken }
  });

  res.json({ success: true });
});
```

Sending Notifications:

```
const admin = require('firebase-admin');

async function sendNotification(userId, title, body, data = {}) {
  const user = await prisma.users.findUnique({
    where: { id: userId },
    select: { fcm_token: true }
  });

  if (!user.fcm_token) return;

  const message = {
    notification: { title, body },
    data,
    token: user.fcm_token
  };

  await admin.messaging().send(message);
}

// Example: New message notification
await sendNotification(
  recipientId,
  'New Message',
  `${senderUsername}: ${messagePreview}`,
  {
    type: 'new_message',
    conversation_id: conversationId,
    sender_id: senderId
  }
);
```

Cloudinary (Image Storage)

Configuration

Environment Variables:

```
CLOUDINARY_CLOUD_NAME=brrow
CLOUDINARY_API_KEY=918121214196197
CLOUDINARY_API_SECRET=_uv_x8ku7vRhFN7Z0Ko61xibqYY
```

Backend Setup:

```
const cloudinary = require('cloudinary').v2;

cloudinary.config({
  cloud_name: process.env.CLOUDINARY_CLOUD_NAME,
  api_key: process.env.CLOUDINARY_API_KEY,
  api_secret: process.env.CLOUDINARY_API_SECRET
});
```

Upload Flow

iOS → Backend:

```swift
// 1. User picks image
let image = UIImage(...)

// 2. Convert to base64
guard let imageData = image.jpegData(compressionQuality: 0.8) else { return }
let base64String = imageData.base64EncodedString()

// 3. Send to backend
let response = try await APIClient.shared.uploadImage(base64: base64String)
// Returns: { "url": "https://res.cloudinary.com/brrow/image/upload/v.../image.jpg" }
```

Backend Upload:

```javascript
async function uploadToCloudinary(base64Data, folder = 'brrow') {
  // Add data URI prefix if missing
  if (!base64Data.startsWith('data:')) {
    base64Data = `data:image/jpeg;base64,${base64Data}`;
  }

  const result = await cloudinary.uploader.upload(base64Data, {
    folder: folder,
    resource_type: 'image',
    format: 'jpg',
    quality: 'auto',
    transformation: [
      { width: 1000, height: 1000, crop: 'limit' },
      { quality: 'auto:good' }
    ]
  });

  return {
    success: true,
    url: result.secure_url,
    public_id: result.public_id
  };
}
```

URL Format:

```
https://res.cloudinary.com/brrow/image/upload/v1759731593/brrow/uploads/1759731593301-
                              └─ cloud     └─ folder     └─ timestamp   └─ filename
```

Image Transformations

Cloudinary automatically optimizes:

- Format: WebP (for browsers that support it)
- Quality: Auto-adjusted based on content
- Size: Responsive sizes
- Compression: Intelligent compression

Manual transformations (via URL):

```
// Original
https://res.cloudinary.com/brrow/image/upload/v.../image.jpg

// 300x300 thumbnail
https://res.cloudinary.com/brrow/image/upload/w_300,h_300,c_fill/v.../image.jpg

// Blur effect
https://res.cloudinary.com/brrow/image/upload/e_blur:400/v.../image.jpg

// Grayscale
https://res.cloudinary.com/brrow/image/upload/e_grayscale/v.../image.jpg
```

ID.me Verification

Purpose

Government-issued ID verification for user trust and safety.

Configuration

Environment Variables:

```
IDME_CLIENT_ID=02ef5aa6d4b40536a8cb82b7b902aba4
IDME_CLIENT_SECRET=d79736fd19dd7960b40d4a342fd56876
IDME_REDIRECT_URI=https://brrow-backend-nodejs-production.up.railway.app/brrow/idme/ca
```

Verification Flow

1. User Initiates Verification

```
// iOS opens URL in Safari
let authURL = "https://brrow-backend-nodejs-production.up.railway.app/brrow/idme/auth"
UIApplication.shared.open(URL(string: authURL)!)
```

2. Backend Redirects to ID.me

```
app.get('/brrow/idme/auth', (req, res) => {
  const authUrl = `https://api.id.me/oauth/authorize?` +
    `client_id=${IDME_CLIENT_ID}` +
    `&redirect_uri=${IDME_REDIRECT_URI}` +
    `&response_type=code` +
    `&scope=military`;

  res.redirect(authUrl);
});
```

3. User Completes ID.me Process

- Uploads government ID photo
- Takes selfie
- ID.me verifies identity
- Redirects back to backend

4. Backend Receives Verification

```
app.get('/brrow/idme/callback', async (req, res) => {
  const { code } = req.query;

  // Exchange code for access token
  const tokenResponse = await fetch('https://api.id.me/oauth/token', {
    method: 'POST',
    body: new URLSearchParams({
      client_id: IDME_CLIENT_ID,
      client_secret: IDME_CLIENT_SECRET,
      redirect_uri: IDME_REDIRECT_URI,
      grant_type: 'authorization_code',
      code
    })
  });

  const tokenData = await tokenResponse.json();

  // Get user profile
  const userResponse = await fetch('https://api.id.me/api/public/v3/attributes.json',
    headers: {
      'Authorization': `Bearer ${tokenData.access_token}`
    }
  });

  const userProfile = await userResponse.json();

  // Store verification data
  await prisma.users.update({
    where: { email: userProfile.attributes.email },
    data: {
      is_verified: true,
      idme_uuid: userProfile.attributes.uuid,
      idme_verified_at: new Date()
    }
  });

  // Redirect to app
  res.redirect('brrowapp://verification/success');
});
```

5. iOS Handles Success

```
// App receives deep link: brrowapp://verification/success
func scene(_ scene: UIScene, openURLContexts URLContexts: Set<UIOpenURLContext>) {
    guard let url = URLContexts.first?.url else { return }

    if url.scheme == "brrowapp" && url.host == "verification" {
        if url.path == "/success" {
            // Show success message
            // Refresh user profile
        }
    }
}
```

Verification Badge

Display in UI:

```
if user.isVerified {
    Image(systemName: "checkmark.seal.fill")
        .foregroundColor(.blue)
}
```

Part 5: Core Features

Marketplace System

Grid Layout

ProfessionalMarketplaceView.swift:

```swift
LazyVGrid(columns: [
    GridItem(.flexible(), spacing: 12),
    GridItem(.flexible(), spacing: 12)
], spacing: 12) {
    ForEach(viewModel.listings) { listing in
        ListingGridCard(listing: listing)
            .onTapGesture {
                // IMPORTANT: Use ID-based navigation to force view recreation
                ListingNavigationManager.shared.showListing(listing)
            }
    }
}
```

Important: The marketplace uses `.id(listing.listingId)` on the detail sheet to prevent SwiftUI from reusing the same view instance when different listings are tapped.

Infinite Scroll

```
ScrollView {
    LazyVGrid(...) {
        ForEach(viewModel.listings) { listing in
            ListingGridCard(listing: listing)
                .onAppear {
                    // Load more when last item appears
                    if listing == viewModel.listings.last {
                        Task {
                            await viewModel.loadNextPage()
                        }
                    }
                }
        }
    }
}
```

Category Filtering

```
ScrollView(.horizontal) {
    HStack(spacing: 12) {
        ForEach(viewModel.categories) { category in
            CategoryPill(
                category: category,
                isSelected: viewModel.selectedCategory?.id == category.id
            )
            .onTapGesture {
                viewModel.selectCategory(category)
            }
        }
    }
}
```

Messaging System

Real-Time Chat (Socket.IO)

Connection Setup:

```swift
class ChatViewModel: ObservableObject {
    private var socketManager: SocketManager?
    private var socket: SocketIOClient?

    func connect() {
        guard let token = AuthenticationService.shared.authToken else { return }

        socketManager = SocketManager(
            socketURL: URL(string: "https://brrow-backend-nodejs-production.up.railway
            config: [
                .log(false),
                .compress,
                .extraHeaders(["Authorization": "Bearer \(token)"])
            ]
        )

        socket = socketManager?.defaultSocket

        // Listen for events
        socket?.on("connect") { [weak self] data, ack in
            print("✅ Socket connected")
            self?.joinConversation()
        }

        socket?.on("new_message") { [weak self] data, ack in
            guard let messageData = data.first as? [String: Any] else { return }
            let message = try? Message(from: messageData)
            self?.receiveMessage(message)
        }

        socket?.connect()
    }

    func sendMessage(_ text: String) {
        socket?.emit("send_message", [
            "conversation_id": conversationId,
            "message": text,
            "recipient_id": recipientId
        ])
      }
  }
```

Backend WebSocket Handler:

```
// services/websocket.service.js
io.on('connection', (socket) => {
  const userId = socket.handshake.auth.userId;

  socket.on('send_message', async (data) => {
    const { conversation_id, message, recipient_id } = data;

    // Save to database
    const newMessage = await prisma.messages.create({
      data: {
        conversation_id,
        sender_id: userId,
        recipient_id,
        message,
        message_type: 'text'
      }
    });

    // Emit to recipient
    io.to(recipient_id).emit('new_message', {
      ...newMessage,
      sender: { username: socket.username }
    });

    // Send push notification
    await sendPushNotification(recipient_id, 'New Message', message);
  });
});
```

Message Persistence

Database Table:

```
CREATE TABLE messages (
  id TEXT PRIMARY KEY,
  conversation_id TEXT NOT NULL,
  sender_id TEXT REFERENCES users(id),
  recipient_id TEXT REFERENCES users(id),
  message TEXT NOT NULL,
  message_type TEXT DEFAULT 'text',
  is_read BOOLEAN DEFAULT false,
  created_at TIMESTAMP DEFAULT NOW(),
  read_at TIMESTAMP,
  INDEX idx_conversation (conversation_id),
  INDEX idx_unread (recipient_id, is_read)
);
```

Load Chat History:

```
app.get('/api/messages/chats/:conversationId', authenticateToken, async (req, res) =>
  const { conversationId } = req.params;
  const { page = 1, limit = 50 } = req.query;

  const messages = await prisma.messages.findMany({
    where: { conversation_id: conversationId },
    include: {
      sender: {
        select: {
          id: true,
          username: true,
          profile_picture_url: true
        }
      }
    },
    orderBy: { created_at: 'desc' },
    skip: (page - 1) * limit,
    take: parseInt(limit)
  });

  res.json({
    success: true,
    messages: messages.reverse(), // Oldest first for chat UI
    pagination: {
      page: parseInt(page),
      limit: parseInt(limit),
      total: await prisma.messages.count({
        where: { conversation_id: conversationId }
      })
    }
  });
});
```

Listing Management

Create Listing Flow

1. User Fills Form

```swift
struct ModernCreateListingView: View {
    @State private var title = ""
    @State private var description = ""
    @State private var price = ""
    @State private var selectedCategory: Category?
    @State private var selectedImages: [UIImage] = []
    @State private var pricingType: PricingType = .sale

    var body: some View {
        Form {
            TextField("Title", text: $title)
            TextEditor(text: $description)
            TextField("Price", text: $price)

            Picker("Pricing Type", selection: $pricingType) {
                Text("Sale").tag(PricingType.sale)
                Text("Rent").tag(PricingType.rent)
            }

            CategoryPicker(selection: $selectedCategory)

            ImagePicker(images: $selectedImages)

            Button("Create Listing") {
                Task {
                    await createListing()
                }
            }
        }
    }
}
```

2. Upload Images

```
func createListing() async {
    // Upload images first
    var imageURLs: [String] = []

    for image in selectedImages {
        guard let imageData = image.jpegData(compressionQuality: 0.8) else { continue
        let base64 = imageData.base64EncodedString()

        let response = try await APIClient.shared.uploadImage(base64: base64)
        imageURLs.append(response.url)
    }

    // Create listing
    let listing = try await APIClient.shared.createListing(
        title: title,
        description: description,
        price: Double(price) ?? 0,
        pricingType: pricingType.rawValue,
        categoryId: selectedCategory?.id ?? "",
        images: imageURLs
    )

    // Navigate to listing detail
    ListingNavigationManager.shared.showListing(listing)
}
```

3. Backend Creates Listing

```
app.post('/api/listings', authenticateToken, async (req, res) => {
  const {
    title,
    description,
    price,
    pricingType,
    categoryId,
    images,
    location,
    condition,
    tags,
    deliveryOptions
  } = req.body;

  // Create listing
  const listing = await prisma.listings.create({
    data: {
      title,
      description,
      price: parseFloat(price),
      daily_rate: pricingType === 'rent' ? parseFloat(price) : null,
      pricing_type: pricingType,
      category_id: categoryId,
      user_id: req.userId,
      condition,
      location,
      tags,
      delivery_options: deliveryOptions,
      is_active: true,
      availability_status: 'AVAILABLE',
      moderation_status: 'PENDING'
    }
  });

  // Create image records
  for (let i = 0; i < images.length; i++) {
    await prisma.listing_images.create({
      data: {
          listing_id: listing.id,
          image_url: images[i],
          is_primary: i === 0,
          display_order: i
        }
      });
    }

    res.status(201).json({
      success: true,
      listing: await prisma.listings.findUnique({
        where: { id: listing.id },
        include: {
          listing_images: true,
          users: true,
          categories: true
        }
      })
    });
  });
```

User Profiles

Profile View

SimpleProfessionalProfileView.swift:

```
struct SimpleProfessionalProfileView: View {
    @StateObject private var viewModel = ProfileViewModel()

    var body: some View {
        ScrollView {
            VStack(spacing: 24) {
                // Header
                profileHeader

                // Stats
                statsRow

                // Bio
                if let bio = viewModel.user?.bio {
                    Text(bio)
                        .font(.body)
                }

                // Listings Grid
                LazyVGrid(columns: [
                    GridItem(.flexible()),
                    GridItem(.flexible())
                ]) {
                    ForEach(viewModel.userListings) { listing in
                        ListingGridCard(listing: listing)
                    }
                }
            }
        }
        .task {
            await viewModel.loadProfile()
        }
    }

    var profileHeader: some View {
        VStack {
            // Profile picture
            AsyncImage(url: URL(string: viewModel.user?.profilePictureUrl ?? "")) { im
```

```
                image
                    .resizable()
                    .scaledToFill()
            } placeholder: {
                Color.gray
            }
            .frame(width: 100, height: 100)
            .clipShape(Circle())

            // Name and verification
            HStack {
                Text(viewModel.user?.displayName ?? viewModel.user?.username ?? "")
                    .font(.title2)
                    .bold()

                if viewModel.user?.isVerified == true {
                    Image(systemName: "checkmark.seal.fill")
                        .foregroundColor(.blue)
                }
            }

            Text("@\(viewModel.user?.username ?? "")")
                .font(.subheadline)
                .foregroundColor(.secondary)
        }
    }

    var statsRow: some View {
        HStack(spacing: 40) {
            StatView(
                value: "\(viewModel.userListings.count)",
                label: "Listings"
            )

            StatView(
                value: String(format: "%.1f", viewModel.user?.averageRating ?? 0),
                label: "Rating"
            )
```

```
                StatView(
                    value: "\(viewModel.user?.totalRatings ?? 0)",
                    label: "Reviews"
                )
            }
        }
    }
```

Edit Profile

Important Fix: Display name vs username validation

Backend (settings-system.js:107-140):

```
if (username !== undefined && username.trim() !== '') {
  const currentUser = await prisma.users.findUnique({
    where: { id: userId },
    select: { username: true }
  });

  const newUsername = username.trim().toLowerCase();
  const currentUsername = (currentUser.username || '').toLowerCase();

  // Only validate if username is actually different
  if (currentUsername !== newUsername) {
    const existingUser = await prisma.users.findFirst({
      where: {
        username: newUsername,
        id: { not: userId }
      }
    });

    if (existingUser) {
      return res.status(400).json({ error: 'Username is already taken' });
    }

    updateData.username = newUsername;
  }
}
```

This fix prevents:

- ❌ "Username is already taken" error when updating display name
- ❌ False validation when username hasn't changed
- ✅ Allows display name updates without username conflicts

Search & Discovery

Search Endpoint

```
app.get('/api/listings/search', authenticateToken, async (req, res) => {
  const { q, category, minPrice, maxPrice, condition, page = 1, limit = 20 } = req.que

  const where = {
    is_active: true,
    OR: []
  };

  // Text search
  if (q) {
    where.OR.push(
      { title: { contains: q, mode: 'insensitive' } },
      { description: { contains: q, mode: 'insensitive' } },
      { tags: { has: q.toLowerCase() } }
    );
  }

  // Filters
  if (category) where.category_id = category;
  if (condition) where.condition = condition;
  if (minPrice || maxPrice) {
    where.price = {};
    if (minPrice) where.price.gte = parseFloat(minPrice);
    if (maxPrice) where.price.lte = parseFloat(maxPrice);
  }

  const listings = await prisma.listings.findMany({
    where,
    include: {
      listing_images: true,
      users: { select: { id: true, username: true, profile_picture_url: true } },
      categories: true
    },
    skip: (page - 1) * limit,
    take: parseInt(limit),
    orderBy: { created_at: 'desc' }
  });

  res.json({
    success: true,
    results: transformListingsForIOS(listings),
    query: q,
    pagination: {
      page: parseInt(page),
      limit: parseInt(limit),
      total: await prisma.listings.count({ where })
    }
  });
});
```

Part 6: Payment Systems

Buy Now (Direct Purchase)

Complete Flow Documentation

Already covered in Part 4 → Stripe Payment System → Buy Now Flow

Key points:

- ✅ Creates purchase record (PENDING)
- ✅ Creates Stripe checkout session

- ✅ iOS opens checkout in SafariViewController
- ✅ Webhook updates status to COMPLETED
- ✅ Funds held for 7 days
- ✅ Auto-release or manual verification

Seller Payouts

Payment Hold System

When purchase completes:

```
// Purchase created with 7-day deadline
deadline: new Date(Date.now() + 7 * 24 * 60 * 60 * 1000)
```

Automatic Release (after 7 days):

```
// Cron job runs daily
setInterval(async () => {
  const expiredPurchases = await prisma.purchases.findMany({
    where: {
      payment_status: 'COMPLETED',
      verification_status: 'PENDING',
      deadline: { lte: new Date() }
    },
    include: {
      seller: true,
      listing: true
    }
  });

  for (const purchase of expiredPurchases) {
    // Release payment to seller
    await releaseFundsToSeller(purchase);
  }
}, 24 * 60 * 60 * 1000); // Daily
```

Manual Verification (buyer confirms):

```
app.post('/api/purchases/:id/verify', authenticateToken, async (req, res) => {
  const purchase = await prisma.purchases.findUnique({
    where: { id: req.params.id }
  });

  // Verify buyer owns this purchase
  if (purchase.buyer_id !== req.userId) {
    return res.status(403).json({ error: 'Not authorized' });
  }

  // Release funds
  await releaseFundsToSeller(purchase);

  res.json({ success: true });
});
```

Release to Seller

```
async function releaseFundsToSeller(purchase) {
  const seller = await prisma.users.findUnique({
    where: { id: purchase.seller_id }
  });

  if (!seller.stripe_account_id) {
    throw new Error('Seller has not set up Stripe Connect');
  }

  // Calculate platform fee (10%)
  const platformFee = purchase.amount * 0.10;
  const sellerAmount = purchase.amount * 0.90;

  // Transfer to seller
  const transfer = await stripe.transfers.create({
    amount: Math.round(sellerAmount * 100), // Convert to cents
    currency: 'usd',
    destination: seller.stripe_account_id,
    transfer_group: purchase.id,
    metadata: {
      purchase_id: purchase.id,
      listing_id: purchase.listing_id,
      buyer_id: purchase.buyer_id
    }
  });

  // Update purchase
  await prisma.purchases.update({
    where: { id: purchase.id },
    data: {
      verification_status: 'VERIFIED',
      verified_at: new Date()
    }
  });

  // Update listing
```

```
    await prisma.listings.update({
      where: { id: purchase.listing_id },
      data: {
        availability_status: 'SOLD',
        is_active: false
      }
    });

    // Send notification to seller
    await sendPushNotification(
      purchase.seller_id,
      'Payment Released',
      `$${sellerAmount.toFixed(2)} has been transferred to your account`
    );
  }
```

Transaction History

Endpoint

```
app.get('/api/purchases/my-purchases', authenticateToken, async (req, res) => {
  const { type = 'all' } = req.query; // 'buyer', 'seller', 'all'

  let where = {};

  if (type === 'buyer') {
    where.buyer_id = req.userId;
  } else if (type === 'seller') {
    where.seller_id = req.userId;
  } else {
    where.OR = [
      { buyer_id: req.userId },
      { seller_id: req.userId }
    ];
  }

  const purchases = await prisma.purchases.findMany({
    where,
    include: {
      listing: {
        include: {
          listing_images: {
            where: { is_primary: true },
            take: 1
          }
        }
      },
      buyer: {
        select: { id: true, username: true, profile_picture_url: true }
      },
      seller: {
        select: { id: true, username: true, profile_picture_url: true }
      }
    },
    orderBy: { created_at: 'desc' }
  });

  res.json({
    success: true,
    purchases: purchases.map(p => ({
      id: p.id,
      amount: p.amount,
      paymentStatus: p.payment_status,
      verificationStatus: p.verification_status,
      purchaseType: p.purchase_type,
      createdAt: p.created_at,
      deadline: p.deadline,
      listing: {
        id: p.listing.id,
        title: p.listing.title,
        image: p.listing.listing_images[0]?.image_url
      },
      buyer: p.buyer,
      seller: p.seller,
      role: p.buyer_id === req.userId ? 'buyer' : 'seller'
    }))
  });
});
```

Refunds & Disputes

Refund Process

Buyer Requests Refund:

```
app.post('/api/purchases/:id/request-refund', authenticateToken, async (req, res) => {
  const { reason } = req.body;
  const purchase = await prisma.purchases.findUnique({
    where: { id: req.params.id },
    include: { listing: true }
  });

  // Verify buyer
  if (purchase.buyer_id !== req.userId) {
    return res.status(403).json({ error: 'Not authorized' });
  }

  // Check deadline hasn't passed
  if (new Date() > purchase.deadline) {
    return res.status(400).json({
      error: 'Verification period has ended. Funds have been released.'
    });
  }

  // Create dispute
  const dispute = await prisma.disputes.create({
    data: {
      purchase_id: purchase.id,
      buyer_id: purchase.buyer_id,
      seller_id: purchase.seller_id,
      reason,
      status: 'PENDING'
    }
  });

  // Notify admin
  await logToDiscord(
    '⚠️ Refund Request',
    `Purchase ${purchase.id} - ${purchase.listing.title}`,
    [
      { name: 'Amount', value: `$${purchase.amount}`, inline: true },
      { name: 'Reason', value: reason }
    ],
      0xFFA500
    );

    res.json({
      success: true,
      dispute
    });
  });
```

Admin Reviews Dispute:

```
app.put('/api/admin/disputes/:id/resolve', adminAuth, async (req, res) => {
  const { resolution, refundAmount } = req.body;
  const dispute = await prisma.disputes.findUnique({
    where: { id: req.params.id },
    include: { purchase: true }
  });

  if (resolution === 'REFUND') {
    // Issue refund via Stripe
    const refund = await stripe.refunds.create({
      payment_intent: dispute.purchase.payment_intent_id,
      amount: Math.round(refundAmount * 100)
    });

    // Update purchase
    await prisma.purchases.update({
      where: { id: dispute.purchase_id },
      data: {
        payment_status: 'REFUNDED',
        verification_status: 'DISPUTED'
      }
    });
  }

  // Update dispute
  await prisma.disputes.update({
    where: { id: dispute.id },
    data: {
      status: 'RESOLVED',
      resolution,
      resolved_at: new Date(),
      resolved_by: req.userId
    }
  });

  res.json({ success: true });
});
```

Part 7: Deployment & Infrastructure

Railway Deployment

Project Info

Service: `brrow-backend-nodejs-production` **URL**: https://brrow-backend-nodejs-production.up.railway.app **Region**: `us-west1` (Oregon, USA) **Plan**: Hobby ($5/month)

Deployment Process

1. Auto-Deploy from GitHub:

```
Local → git push origin master → GitHub → Railway detects push → Auto-deploys
```

2. Build Process:

```
# Railway runs these automatically:
npm install
npx prisma generate
node prisma-server.js
```

3. Health Check:

```
Railway pings: /health
Expected: 200 OK
If fails: Restarts container
```

4. Deployment Time:

- Build: ~1 minute
- Deploy: ~30 seconds
- Total: ~2 minutes

Manual Deploy

Via Railway Dashboard:

1. Go to https://railway.app
2. Select `brrow-backend-nodejs-production`
3. Click "Deployments" tab
4. Click "..." on latest commit → "Redeploy"

Via CLI (if Railway CLI installed):

```
railway up
```

Environment Variables

Complete List (Railway)

Backend Server:

```
# Database
DATABASE_URL="postgresql://postgres:PASSWORD@host:port/railway"

# JWT
JWT_SECRET="your-secret-key-here"
JWT_REFRESH_SECRET="your-refresh-secret-here"

# Stripe
STRIPE_SECRET_KEY="sk_live_..." # or sk_test_...
STRIPE_PUBLISHABLE_KEY="pk_live_..." # or pk_test_...
STRIPE_WEBHOOK_SECRET="whsec_..."

# Cloudinary
CLOUDINARY_CLOUD_NAME="brrow"
CLOUDINARY_API_KEY="918121214196197"
CLOUDINARY_API_SECRET="_uv_x8ku7vRhFN7Z0Ko61xibqYY"

# Firebase
FIREBASE_PROJECT_ID="brrow-app-firebase"
FIREBASE_PRIVATE_KEY="-----BEGIN PRIVATE KEY-----\n...\n-----END PRIVATE KEY-----\n"
FIREBASE_CLIENT_EMAIL="firebase-adminsdk-xxxxx@brrow-app-firebase.iam.gserviceaccount.

# ID.me
IDME_CLIENT_ID="02ef5aa6d4b40536a8cb82b7b902aba4"
IDME_CLIENT_SECRET="d79736fd19dd7960b40d4a342fd56876"
IDME_REDIRECT_URI="https://brrow-backend-nodejs-production.up.railway.app/brrow/idme/c

# Google OAuth
GOOGLE_CLIENT_ID="your-google-client-id"
GOOGLE_CLIENT_SECRET="your-google-client-secret"

# Discord (for error logging)
DISCORD_WEBHOOK_URL="https://discord.com/api/webhooks/..."

# Server Config
NODE_ENV="production"
PORT="3001"
```

How to Update

Railway Dashboard:

1. Go to service
2. Click "Variables" tab
3. Click variable to edit OR "New Variable"
4. Enter value
5. Click outside to save
6. Service auto-restarts

⚠️ **Important**:

- Railway encrypts secrets automatically
- Variables are available as `process.env.VARIABLE_NAME`
- Restart required for changes to take effect
- Test mode Stripe keys: Change `STRIPE_SECRET_KEY` and `STRIPE_PUBLISHABLE_KEY` to `sk_test_...` and `pk_test_...`

Database Management

Access Database

Via Railway CLI:

```
# Install Railway CLI
npm install -g railway

# Login
railway login

# Link project
railway link

# Connect to PostgreSQL
railway run psql $DATABASE_URL
```

Via pgAdmin / TablePlus:

```
Host: yamanote.proxy.rlwy.net
Port: 10740
Database: railway
Username: postgres
Password: kciFfaaVBLcfEAlHomvyNFnbMjIxGdOE
```

Prisma Migrations

Local Development:

```
cd brrow-backend

# Create migration
npx prisma migrate dev --name add_new_field

# Push to database (skip migration)
npx prisma db push

# Generate Prisma Client
npx prisma generate
```

Production (Railway auto-runs):

```
# On deploy, Railway runs:
npx prisma generate

# Manual migration:
railway run npx prisma migrate deploy
```

Common Queries

Check table structure:

```
\d users
\d listings
\d purchases
```

Count records:

```sql
SELECT COUNT(*) FROM users;
SELECT COUNT(*) FROM listings WHERE is_active = true;
SELECT COUNT(*) FROM purchases WHERE payment_status = 'COMPLETED';
```

View recent purchases:

```sql
SELECT
  p.id,
  p.amount,
  p.payment_status,
  l.title AS listing_title,
  b.username AS buyer,
  s.username AS seller
FROM purchases p
JOIN listings l ON p.listing_id = l.id
JOIN users b ON p.buyer_id = b.id
JOIN users s ON p.seller_id = s.id
ORDER BY p.created_at DESC
LIMIT 10;
```

Monitoring & Logging

Railway Logs

View Logs:

1. Railway Dashboard
2. Select service
3. Click "Deployments" tab
4. Click on deployment
5. See real-time logs

Log Levels:

```
logger.info('Server started');     // ✅ Info
logger.warn('Slow query');         // ⚠️ Warning
logger.error('Database failed');  // ❌ Error
logger.debug('Request details');  // 🔍 Debug (production: disabled)
```

Health Monitoring

Health Endpoint (`/health`):

```
{
  "status": "healthy",
  "service": "brrow-backend",
  "version": "1.3.4",
  "database": "connected",
  "uptime": 12345.67,
  "memory": {
    "rss": 197685248,
    "heapTotal": 45010944,
    "heapUsed": 41878464
  }
}
```

Railway checks:

- Every 30 seconds
- If 503 → Restarts container
- If 200 → Healthy

Discord Alerts

Error notifications sent to Discord:

```
await logToDiscord(
  'Payment Failed',
  error.message,
  [
    { name: 'User', value: user.username, inline: true },
    { name: 'Amount', value: `$${amount}`, inline: true }
  ],
  0xFF0000, // Red color
  true      // isError
);
```

Memory Monitoring

Auto-monitoring:

```
// Logs every minute
setInterval(() => logMemoryUsage(), 60000);

// Alerts if > 400MB
app.use(memoryMonitor(400));

// Garbage collection
scheduleGarbageCollection();
```

Part 8: Development Guides

Local Development Setup

Prerequisites

Required Software:

- Node.js 18+
- PostgreSQL 15+
- Git
- Xcode 15+ (for iOS)
- CocoaPods

Backend Setup

1. Clone Repository:

```
cd ~/Documents/Projects
git clone https://github.com/shalinratna/brrow-backend-nodejs.git
cd brrow-backend-nodejs
```

2. Install Dependencies:

```
npm install
```

3. Setup Environment:

```
cp .env.example .env
# Edit .env with your credentials
```

4. Setup Database:

```
# Start PostgreSQL (if local)
pg_ctl start

# Create database
createdb brrow_dev

# Update DATABASE_URL in .env
DATABASE_URL="postgresql://localhost:5432/brrow_dev"

# Run migrations
npx prisma migrate dev

# Generate Prisma Client
npx prisma generate
```

5. Start Server:

```
node prisma-server.js
# Server runs on http://localhost:3001
```

6. Test:

```
curl http://localhost:3001/health
# Should return: {"status":"healthy",...}
```

iOS Setup

1. Install CocoaPods:

```
sudo gem install cocoapods
```

2. Install Pods:

```
cd ~/Documents/Projects/Xcode/Brrow
pod install
```

3. Open Workspace:

```
open Brrow.xcworkspace
```

4. Configure Signing:

- Xcode → Brrow target → Signing & Capabilities
- Select your Apple Developer account
- Choose development team

5. Run on Simulator:

- Select iPhone 16 Pro simulator
- Click Run (⌘R)

Testing Strategies

Manual Testing Checklist

Authentication:

- [ ] Register new user
- [ ] Login with email

- [ ] Login with Google
- [ ] Login with Apple
- [ ] Logout
- [ ] Token refresh

Listings:

- [ ] Browse marketplace
- [ ] Filter by category
- [ ] Search listings
- [ ] View listing detail
- [ ] Create listing (sale)
- [ ] Create listing (rent)
- [ ] Edit listing
- [ ] Delete listing
- [ ] Upload images

Messaging:

- [ ] Send message
- [ ] Receive message (real-time)
- [ ] Load chat history
- [ ] Mark as read
- [ ] Unread count badge

Payments:

- [ ] Buy Now flow
- [ ] Stripe checkout
- [ ] Payment success
- [ ] Payment cancel
- [ ] View transaction history

Profile:

- [ ] View own profile
- [ ] Edit profile

- [ ] Change display name
- [ ] Upload profile picture
- [ ] View other user profile

Test Accounts

Backend (Local):

```
# Create test user via API
curl -X POST http://localhost:3001/api/auth/register \
  -H "Content-Type: application/json" \
  -d '{
    "email": "test@example.com",
    "password": "TestPass123",
    "username": "testuser",
    "firstName": "Test",
    "lastName": "User"
  }'
```

Stripe Test Mode:

- Card: 4242 4242 4242 4242
- Expiry: Any future date
- CVC: Any 3 digits
- ZIP: Any 5 digits

Debugging Tools

iOS Debugging

Console Logging:

```
// Custom emoji logging
print("🔵 [DEBUG] Variable value: \(value)")
print("✅ [SUCCESS] Operation completed")
print("❌ [ERROR] Failed: \(error)")
```

Network Debugging:

```
// APIClient logs all requests
🐛 [Brrow API Debug] Creating request
📊 Data: ["endpoint": "api/listings", "method": "GET"]
📡 Response: 200 for /api/listings
```

Breakpoints:

- Set breakpoint in Xcode (click line number)
- Click "Add Exception Breakpoint" for crashes
- Use `po variable` in console to inspect

Memory Leaks:

- Xcode → Product → Profile
- Select "Leaks" instrument
- Run app and look for red bars

Backend Debugging

Pino Logging:

```
logger.info({ userId: '123' }, 'User logged in');
logger.warn({ query: 'SELECT...' }, 'Slow query');
logger.error({ error: err.message }, 'Database failed');
```

Railway Logs:

- Real-time logs in Railway dashboard

- Filter by level (info, warn, error)
- Search by keyword

Database Queries:

```
# Log all Prisma queries (add to .env)
DEBUG="prisma:query"

# Restart server, see all SQL queries in console
```

Common Issues & Fixes

Issue: "Username is already taken" when updating display name

Symptom: Error when changing display name, even though username isn't changing.

Fix: Already deployed to Railway (2025-10-08)

Verification:

```
# Check Railway logs for:
✅ Username unchanged, skipping validation
```

Issue: Wrong listing shows when tapped

Symptom: Tap listing A, but listing B appears.

Fix: Already implemented (ListingNavigationManager.swift:107, 114)

Code:

```
.sheet(isPresented: $navigationManager.showingListingDetail) {
    if let listing = navigationManager.selectedListing {
        NavigationView {
            ProfessionalListingDetailView(listing: listing)
        }
        .id(listing.listingId) // ← Forces view recreation
    }
}
```

Issue: Archive shows as "Generic Xcode Archive"

Symptom: Xcode Organizer shows archive as Generic instead of iOS App.

Solution: Post-action script adds ApplicationProperties.

Verification:

```
defaults read ~/Library/Developer/Xcode/Archives/*/Brrow*.xcarchive/Info.plist Applica
# Should show CFBundleIdentifier, CFBundleVersion, etc.
```

If still broken:

```
# Clear Xcode cache
rm ~/Library/Developer/Xcode/UserData/IDEArchiveDatabase.db*
```

Issue: Backend returning 500 errors

Symptoms:

```
📡 Response: 500 for /api/listings
📡 Response: 500 for /api/users/me
```

Diagnosis:

```
# Check Railway logs for:
Engine is not yet connected
PrismaClientUnknownRequestError
```

Fix: Already deployed (2025-10-08)

Code (prisma-server.js:9899):

```
async function startServer() {
  try {
    // Explicitly connect to database
    await prisma.$connect();
    logger.info('Database connected successfully');

    // Start server
    const server = httpServer.listen(PORT, ...);
  }
}
```

Issue: Images not loading in iOS app

Symptoms:

- Gray placeholders
- `BrrowAsyncImage: Empty/null URL input`

Diagnosis:

```
// Check logs for:
🖼️ BrrowAsyncImage: Input URL = 'null'
```

Fix: Verify Cloudinary URLs in database

Query:

```sql
SELECT id, title,
  (SELECT COUNT(*) FROM listing_images WHERE listing_id = listings.id) as image_count
FROM listings
WHERE id = 'your-listing-id';

SELECT image_url FROM listing_images WHERE listing_id = 'your-listing-id';
```

Part 9: Xcode & iOS Build

Xcode Project Structure

Workspace vs Project

Brrow.xcworkspace (use this):

- Includes Brrow.xcodeproj
- Includes Pods project (34 dependencies)
- Required for CocoaPods

Brrow.xcodeproj (don't use directly):

- Main app project
- Won't build without workspace (missing Pods)

Targets

1. Brrow (Main App):

- Bundle ID: `com.shaiitech.com.brrow`
- Platform: iOS 16+
- Architectures: arm64 (iPhone)
- Build: 594
- Version: 1.3.4

2. BrrowWidgets (Widget Extension):

- Bundle ID: `com.shaiitech.com.brrow.BrrowWidgets`
- Platform: iOS 16+
- Embedded in main app

3. BrrowTests (Unit Tests):

- Test bundle

4. BrrowUITests (UI Tests):

- UI test bundle

Build Settings (Important)

Release Configuration (Brrow target):

```
ARCHS = arm64
CODE_SIGN_STYLE = Automatic
DEVELOPMENT_TEAM = UXM5W873X3
PRODUCT_BUNDLE_IDENTIFIER = com.shaiitech.com.brrow
CURRENT_PROJECT_VERSION = 594
MARKETING_VERSION = 1.3.4
SKIP_INSTALL = NO
DEPLOYMENT_LOCATION = NO
INFOPLIST_KEY_CFBundleDisplayName = Brrow
```

Archive & Distribution

Why Post-Action Script is Necessary

The Problem:

- Xcode's auto-archiving fails with CocoaPods + App Extensions
- Archive's `Info.plist` missing `ApplicationProperties` dictionary
- Organizer shows "Generic Xcode Archive" instead of "iOS App"
- Cannot distribute to App Store

The Solution:

- Post-action script runs after archive completes
- Adds `ApplicationProperties` to archive's `Info.plist`
- Archive now recognized as iOS App
- Can export IPA and upload to App Store

Post-Action Location: `Brrow.xcworkspace/xcshareddata/xcschemes/Brrow.xcscheme`

XML:

```xml
<ArchiveAction buildConfiguration="Release">
  <PostActions>
    <ExecutionAction ActionType="ShellScriptAction">
      <ActionContent title="Add ApplicationProperties">
        #!/bin/bash
        ${SRCROOT}/fix-archive-proper.sh
      </ActionContent>
    </ExecutionAction>
  </PostActions>
</ArchiveAction>
```

Script (`fix-archive-proper.sh`):

```
#!/bin/bash
ARCHIVE_PATH="$ARCHIVE_PATH"
INFO_PLIST="$ARCHIVE_PATH/Info.plist"

# Add ApplicationProperties if missing
/usr/libexec/PlistBuddy -c "Add :ApplicationProperties dict" "$INFO_PLIST" 2>/dev/null
/usr/libexec/PlistBuddy -c "Add :ApplicationProperties:ApplicationPath string 'Applica
/usr/libexec/PlistBuddy -c "Add :ApplicationProperties:CFBundleIdentifier string 'com.
/usr/libexec/PlistBuddy -c "Add :ApplicationProperties:CFBundleShortVersionString stri
/usr/libexec/PlistBuddy -c "Add :ApplicationProperties:CFBundleVersion string '594'" "
```

Archive Process

1. Clean Build (recommended):

```
Xcode → Product → Clean Build Folder (⌘⇧K)
```

2. Archive:

```
Xcode → Product → Archive (⌘⇧B for build, then Archive)
```

3. Wait:

- Build time: ~2-3 minutes
- Archive creation: ~10 seconds
- Post-action script: <1 second
- Total: ~3 minutes

4. Verify:

```
Window → Organizer → Archives
Should show: "Brrow" as "iOS App"
Build number: 594
```

Export IPA

1. Select Archive:

- Window → Organizer → Archives
- Select latest archive
- Click "Distribute App"

2. Distribution Method:

- **App Store Connect**: Upload to TestFlight/App Store
- **Ad Hoc**: Install on registered devices
- **Enterprise**: Company distribution
- **Development**: Testing on devices

3. Options:

- ✅ Upload your app's symbols (for crash reports)
- ✅ Include bitcode (if required)
- ✅ Manage Version and Build Number

4. Signing:

- Automatic signing (recommended)
- Or: Manual signing with provisioning profile

5. Export:

- Click "Export"
- Choose location
- Xcode creates `.ipa` file

Upload to App Store

Via Xcode:

1. Archive → Distribute App
2. Select "App Store Connect"
3. Select "Upload"
4. Wait for processing (~5 minutes)
5. Check App Store Connect

Via Transporter (alternative):

1. Export IPA (Distribution method: "App Store Connect")
2. Open Transporter app
3. Drag IPA file
4. Click "Deliver"

Verification:

1. Go to https://appstoreconnect.apple.com
2. My Apps → Brrow
3. TestFlight tab
4. See build appear (~5-15 minutes)
5. Status: "Processing" → "Testing"

CocoaPods Dependencies

Current Pods (34 total)

Networking:

- Alamofire (HTTP)
- Socket.IO-Client-Swift (WebSocket)

Images:

- SDWebImage (Async loading & caching)
- SDWebImageSwiftUI (SwiftUI integration)

Payments:

- StripePaymentSheet
- StripePayments
- StripePaymentsUI
- StripeUICore
- StripeCore
- StripeApplePay

Firebase (13 pods):

- Firebase
- FirebaseAnalytics
- FirebaseAuth
- FirebaseCore
- FirebaseCoreInternal
- FirebaseMessaging
- FirebaseInstallations
- GoogleDataTransport
- GoogleUtilities
- nanopb
- PromisesObjC

Social Login:

- GoogleSignIn
- FBSDKLoginKit (Facebook)

UI:

- SwiftyGif (GIF support)
- Starscream (WebSocket)

Managing Pods

Update Pods:

```
cd ~/Documents/Projects/Xcode/Brrow
pod update
```

Install New Pod:

```
# Edit Podfile
pod 'NewPodName', '~> 1.0'

# Install
pod install
```

Remove Pod:

```
# Remove from Podfile
# Then:
pod install
```

Clean:

```
pod deintegrate
pod install
```

App Store Submission

Pre-Submission Checklist

Code:

- [ ] No test/debug code in release
- [ ] No hardcoded credentials
- [ ] All API keys in environment
- [ ] Error handling on all network calls
- [ ] Proper loading states

Assets:

- [ ] App icon (all sizes)
- [ ] Launch screen
- [ ] Screenshots (all required sizes)
- [ ] Preview video (optional)

Metadata:

- [ ] App name
- [ ] Subtitle
- [ ] Description
- [ ] Keywords
- [ ] Support URL
- [ ] Privacy policy URL

Compliance:

- [ ] Privacy nutrition label
- [ ] Export compliance (encryption)
- [ ] Content rights
- [ ] Age rating

App Store Connect Setup

1. Create App:

- Go to https://appstoreconnect.apple.com
- My Apps → "+" → New App
- Platform: iOS
- Name: Brrow
- Primary Language: English (U.S.)
- Bundle ID: com.shaiitech.com.brrow
- SKU: brrow-ios-app

2. App Information:

- Category: Lifestyle / Shopping
- Content Rights: Check box
- Age Rating: Fill questionnaire

3. Pricing:

- Free
- Available in all territories

4. Build:

- Upload via Xcode or Transporter
- Select build in App Store Connect
- Add export compliance info

5. Submit for Review:

- Fill all required fields
- Add screenshots
- Click "Submit for Review"
- Wait for Apple review (~24-48 hours)

Part 10: Maintenance & Updates

Version Management

Versioning Strategy

Semantic Versioning (MAJOR.MINOR.PATCH):

```
1.3.4
| | |
| | └─ Patch (bug fixes)
| └─── Minor (new features, backward compatible)
└───── Major (breaking changes)
```

Examples:

- `1.3.4` → `1.3.5`: Bug fix
- `1.3.5` → `1.4.0`: New feature (e.g., video calls)
- `1.4.0` → `2.0.0`: Major redesign

Incrementing Version

iOS (Xcode):

```
1. Brrow target → General
2. Version: 1.3.4 → 1.3.5 (MARKETING_VERSION)
3. Build: 594 → 595 (CURRENT_PROJECT_VERSION)
4. Archive with new build number
```

Backend:

```
// prisma-server.js:2
const VERSION = '1.3.5';

// Also update package.json:
"version": "1.3.5"
```

Sync Required: ✅ iOS and backend versions should match for clarity

Update Procedures

Backend Update (Railway)

1. Make Changes:

```
cd ~/Documents/Projects/Xcode/Brrow/brrow-backend
# Edit files
```

2. Test Locally:

```
node prisma-server.js
# Test on http://localhost:3001
```

3. Commit & Push:

```
git add .
git commit -m "Fix: Description of change

- Detail 1
- Detail 2

🤖 Generated with [Claude Code](https://claude.com/claude-code)

Co-Authored-By: Claude <noreply@anthropic.com>"

git push origin master
```

4. Railway Auto-Deploys:

- Detects push
- Builds (~1 min)
- Deploys (~30 sec)
- Total: ~2 minutes

5. Verify:

```
curl https://brrow-backend-nodejs-production.up.railway.app/health
# Check version number
```

iOS Update

1. Make Changes:

```
# Edit Swift files in Xcode
```

2. Test:

```
Run on simulator (⌘R)
Run on physical device
Test all affected features
```

3. Increment Build:

```
Brrow target → General → Build: 594 → 595
```

4. Archive:

```
Product → Archive
Verify archive shows as "iOS App"
```

5. Distribute:

```
Organizer → Distribute App → App Store Connect
Wait for processing
```

6. Submit:

```
App Store Connect → Select build → Submit for Review
```

Rollback Strategies

Backend Rollback (Railway)

Option 1: Redeploy Previous Commit:

1. Railway Dashboard
2. Deployments tab
3. Find working deployment
4. Click "..." → "Redeploy"

Option 2: Git Revert:

```
# Find bad commit
git log --oneline -10

# Revert it
git revert <commit-hash>
git push origin master

# Railway auto-deploys reverted code
```

Option 3: Manual Rollback:

```
# Reset to previous commit
git reset --hard <working-commit-hash>
git push --force origin master

# ⚠️ Use with caution - rewrites history
```

iOS Rollback

Can't rollback after approval, but can:

Option 1: Remove from Sale:

- App Store Connect → Brrow
- Pricing and Availability
- "Remove from Sale"

Option 2: Submit Hotfix:

- Fix critical bug
- Increment build: 595 → 596
- Archive & upload
- Submit with "Expedited Review" request

Option 3: Revert Build (before release):

- App Store Connect → TestFlight
- Remove problematic build
- Select previous working build
- Submit for review

Security Updates

Regular Maintenance

Monthly Checklist:

- [ ] Update Node.js dependencies (`npm update`)
- [ ] Update CocoaPods (`pod update`)
- [ ] Review Railway logs for errors
- [ ] Check Stripe Dashboard for issues
- [ ] Monitor Firebase usage
- [ ] Review App Store ratings/reviews

Security Patches:

- [ ] Update critical dependencies
- [ ] Test thoroughly
- [ ] Deploy immediately if security-related

Dependency Updates

Backend:

```
# Check for updates
npm outdated

# Update specific package
npm update express

# Update all (careful!)
npm update

# Audit for vulnerabilities
npm audit
npm audit fix
```

iOS:

```
# Check for updates
pod outdated

# Update specific pod
pod update StripePaymentSheet

# Update all
pod update
```

⚠️ **Always test after updates!**

📌 Quick Reference

Essential URLs

Service	URL
Production Backend	https://brrow-backend-nodejs-production.up.railway.app
Railway Dashboard	https://railway.app
Stripe Dashboard	https://dashboard.stripe.com
App Store Connect	https://appstoreconnect.apple.com
Firebase Console	https://console.firebase.google.com
Cloudinary Dashboard	https://cloudinary.com/console

Essential Commands

Backend:

```
# Start locally
node prisma-server.js

# Database migrations
npx prisma migrate dev
npx prisma generate
npx prisma db push

# Deploy (auto via git push)
git push origin master
```

iOS:

```
# Install dependencies
pod install

# Clean build
⌘⇧K in Xcode

# Archive
Product → Archive

# Run
⌘R
```

Essential Environment Variables

Switch to Test Mode:

```
STRIPE_SECRET_KEY=sk_test_...
STRIPE_PUBLISHABLE_KEY=pk_test_...
```

Switch to Live Mode:

```
STRIPE_SECRET_KEY=sk_live_...
STRIPE_PUBLISHABLE_KEY=pk_live_...
```

Support Contacts

Claude Code: This documentation **GitHub**: https://github.com/shalinratna/brrow-backend-nodejs **Owner**: Shalin Ratna (shalinratna@gmail.com)

📝 Document Update Log

Date	Version	Changes	Updated By
2025-10-08	1.0.0	Initial comprehensive documentation created	Claude Code

🎯 **This is a living document. All future updates will be added here instead of creating new .md files.**

📘 **Bookmark this file**: `/Users/shalin/Documents/Projects/Xcode/Brrow/BRROW_COMPLETE_SYSTEM_DOCUMENTATION.md`