Portfolio Stage 3 Report

Technical Documentation

1447 (Summer 2025)

UniStay – Students accommodation

NAME
Abdulrahman Al-Fawzan
Abdulelah Al-Shehri
Mohammad Al-Omar
Meshari Al-Abdullah

1- User Stories:

A user story is a short, simple description of a feature or capability written from the perspective of the person who needs it—usually the end-user or another stakeholder.

- **Purpose: It captures functional requirements of the system in plain, non-technical language.**
- **Format: Typically follows the template:**
  "As a <type of user>, I want <goal or action> so that <benefit or reason>."
- **Focus: Describes what the system should do (the *what* and *why*), not how it will be implemented.**
- **Role in development: Guides designers and developers in building features that deliver real value to users, and helps prioritize work for MVPs, sprints, or product backlogs.**

Students (End-Users):

Must Have:

As a student, **I want to** search for available accommodation near my campus, **so that** I can quickly find suitable housing when relocating.

As a student, I want to filter listings by price, distance, female-only, and roommate availability, **so that** I can narrow results to my specific needs.

As a student, **I want to** view detailed listings with photos, price, rules, and map location, **so that** I can evaluate whether a place fits my needs.

As a student, **I want to** message the owner/landlord directly in the app, **so that** I can ask questions and clarify details before deciding.

Should Have:

As a student, I want to bookmark or save favorite listings, so that I can compare and return later.

As a student, **I want to** see a verification badge (edu-email, phone, or ID) on listings, **so that** I can trust the landlord or roommate is legitimate.

As a student, **I want to** opt into roommate matching by filling in preferences, **so that** I can share housing costs with someone compatible.

Could Have/Nice to have:

As a student, I want to rate/report listings, so that I can help improve platform trust and avoid scams.

As a student, I want to see a “student discount” tag on some listings, so that I know which options are more affordable.

Won’t Have (MVP)

As a student, I want to pay deposits or rent directly in-app, so that I can secure housing digitally. (Future feature)

Owners / Landlords:

Must Have

As an owner, **I want to** create a new listing with required fields (price, rules, photos, availability), **so that** students see complete and useful information.

As an owner, **I want to** edit or manage my listings from a dashboard, **so that** I can keep availability and pricing updated.

As an owner, I want to verify my identity via phone/email, **so that** students trust my listings.

Should Have:

As an owner, I want to receive inquiries and messages from students in the app, so that I can communicate easily without sharing my phone number.

As an owner, I want to mark my listing as “student discount”, so that I can attract more student renters.

Could Have:

As an owner, I want to see profile completeness scores of students, so that I can decide whether they are reliable tenants.

Won’t Have (MVP):

As an owner, I want to manage rental payments or contracts in-app, so that I can streamline the process digitally. (Future release)

Hotel / Serviced Apartment Partners

Must Have

As a hotel partner, I want to list student-friendly rooms/apartments with student discounts, so that I can fill off-peak occupancy with student tenants.

Should Have

As a hotel partner, I want to receive inquiries directly via dashboard/chat, so that I can convert more bookings quickly.

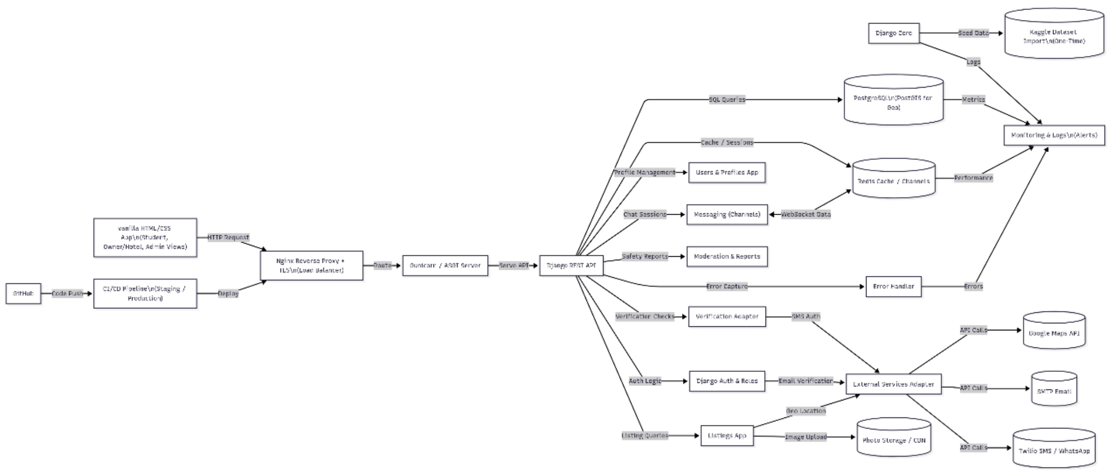
Admin / Moderator

Must Have

As an admin, I want to review and approve owner/hotel listings before they go live, so that the platform maintains quality and trust.

As an admin, I want to see reports of fraudulent or inappropriate listings, so that I can take action quickly to protect users.

2- System Architecture:



System Architecture Explanation

Clients

- A single **HTML/CSS front-end** serves three roles: Student, Owner/Hotel, and Admin views.
- All traffic goes through **HTTPS** to ensure secure communication.

Edge Layer

- **Nginx** acts as a reverse proxy and TLS terminator, handling load balancing and routing requests.
- Requests are passed to **Gunicorn (ASGI server)**, which serves the Django back-end.

Back-End Layer (Django)

- **Django REST API** is the central entry point for business logic.
- It delegates requests to specialized apps:
    - **Auth** → manages login, OTP verification, roles, and permissions.
    - **Listings** → handles CRUD for properties (rooms, studios, apartments).
    - **Users** → manages student/landlord profiles.
    - **Messaging** → supports in-app chat using Django Channels + Redis.
    - **Reports** → processes moderation actions and reported listings.
    - **Verify** → adapters for verification services (e.g., Twilio).
    - **Error Handler** → central exception capture for monitoring.
- **Services Adapter** standardizes communication with external APIs (Twilio, SMTP, Google Maps).

Data Layer

- **PostgreSQL (with PostGIS)** stores persistent data such as users, listings, reviews, and geolocation info.
- **Redis** manages caching, session storage, and WebSocket channels for chat.

External Integrations

- **Twilio** → sends OTP codes via SMS/WhatsApp for phone verification.
- **SMTP** → sends verification emails and notifications.
- **Google Maps API** → geocoding (address to coordinates) and maps integration.
- **Photo Storage/CDN** → stores and serves uploaded listing images.
- **Kaggle Dataset** → one-time seed data for testing/demo purposes.

DevOps Layer

- **GitHub** → source code management with branching strategy.
- **CI/CD pipeline** → automated testing, staging deployment, and production rollout.
- **Monitoring & Logs** → centralized logging, alerts, and metrics (errors, DB load, Redis performance).

Data Flow

1. A user action in the React front-end (e.g., "Search listings") → sends request to Nginx.
2. Nginx routes to Gunicorn → Django REST API.
3. API validates/authenticates request → calls appropriate Django app.
4. App queries **PostgreSQL** (persistent data) or **Redis** (cache/sessions).

5. If needed, Django calls **external services** (Twilio, Maps, SMTP).
6. Results return through the API → back to the React client.
7. Errors are logged centrally and monitored.

3- Components, Classes, and Database Design:

A) Roles & Core Classes:

1) User (AbstractUser-based)

- role: ENUM { STUDENT, LANDLORD, HOTEL, ADMIN }
- phone: string (unique)
- is_phone_verified: bool
- is_edu_verified: bool (optional badge)
- gender: ENUM { MALE, FEMALE, OTHER } (for filters like female-only)

2) Listing (aka Place)

- owner: FK → User (LANDLORD or HOTEL)
- title, description
- monthly_price, deposit_amount (nullable)
- room_type: ENUM { ROOM, SHARED, STUDIO, APARTMENT }
- female_only: bool
- roommates_allowed: bool
- student_discount: bool
- available_from: date
- address: string
- latitude, longitude (or PostGIS Point)
- campus: FK → Campus (Riyadh campus; helps distance filter)
- status: ENUM { DRAFT, PENDING_REVIEW, APPROVED, REJECTED }
- is_active: bool

3) Review

- author: FK → User

- target_type: ENUM { USER, LISTING }
- target_user: FK → User (nullable; used when target_type=USER)
- target_listing: FK → Listing (nullable; used when target_type=LISTING)
- rating: smallint (1–5)
- comment: text
- created_at
- **Note**: Single table with a target_type enum keeps it simple and matches your “same attributes” idea.

4) Message (Twilio-friendly)

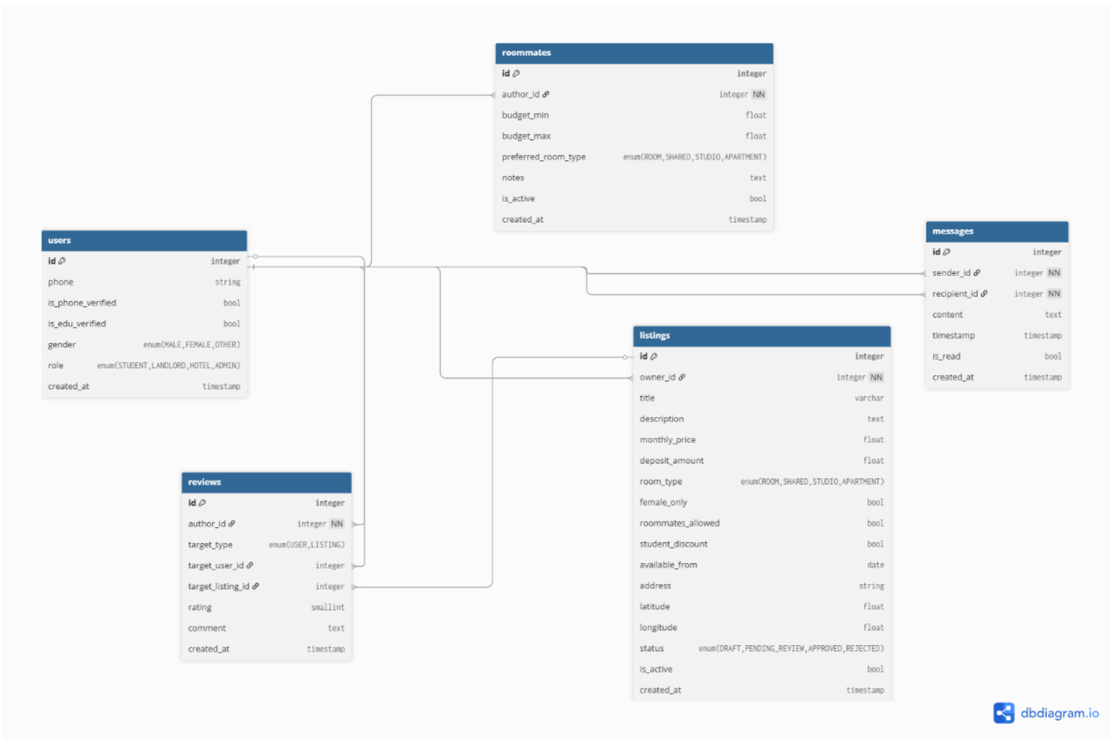
- listing: FK → Listing (context)
- from_user, to_user: FK → User
- direction: ENUM { OUTBOUND, INBOUND }
- body: text
- twilio_sid, status (queued/sent/delivered/failed)
- sent_at

5) Roommate

Keep it as a **post** that students can publish for discovery:

- author: FK → User (STUDENT)
- campus: FK → Campus (Riyadh campus)
- budget_min, budget_max
- preferred_room_type: ENUM (same as listing)
- notes: text
- is_active: bool
- created_at

B) Database Schema:



roommates

id	integer	
author_id	integer	NN
budget_min	float	
budget_max	float	
preferred_room_type	enum(ROOM,SHARED,STUDIO,APARTMENT)	
notes	text	
is_active	bool	
created_at	timestamp	

users

id	integer	
phone	string	
is_phone_verified	bool	
is_edu_verified	bool	
gender	enum(MALE,FEMALE,OTHER)	
role	enum(STUDENT,LANDLORD,HOTEL,ADMIN)	
created_at	timestamp	

messages

id	integer	
sender_id	integer	NN
recipient_id	integer	NN
content	text	
timestamp	timestamp	
is_read	bool	
created_at	timestamp	

listings

id	integer	
owner_id	integer	NN
title	varchar	
description	text	
monthly_price	float	
deposit_amount	float	
room_type	enum(ROOM,SHARED,STUDIO,APARTMENT)	
female_only	bool	
roommates_allowed	bool	
student_discount	bool	
available_from	date	
address	string	
latitude	float	
longitude	float	
status	enum(DRAFT,PENDING_REVIEW,APPROVED,REJECTED)	
is_active	bool	
created_at	timestamp	

reviews

id	integer	
author_id	integer	NN
target_type	enum(USER,LISTING)	
target_user_id	integer	
target_listing_id	integer	
rating	smallint	
comment	text	
created_at	timestamp	

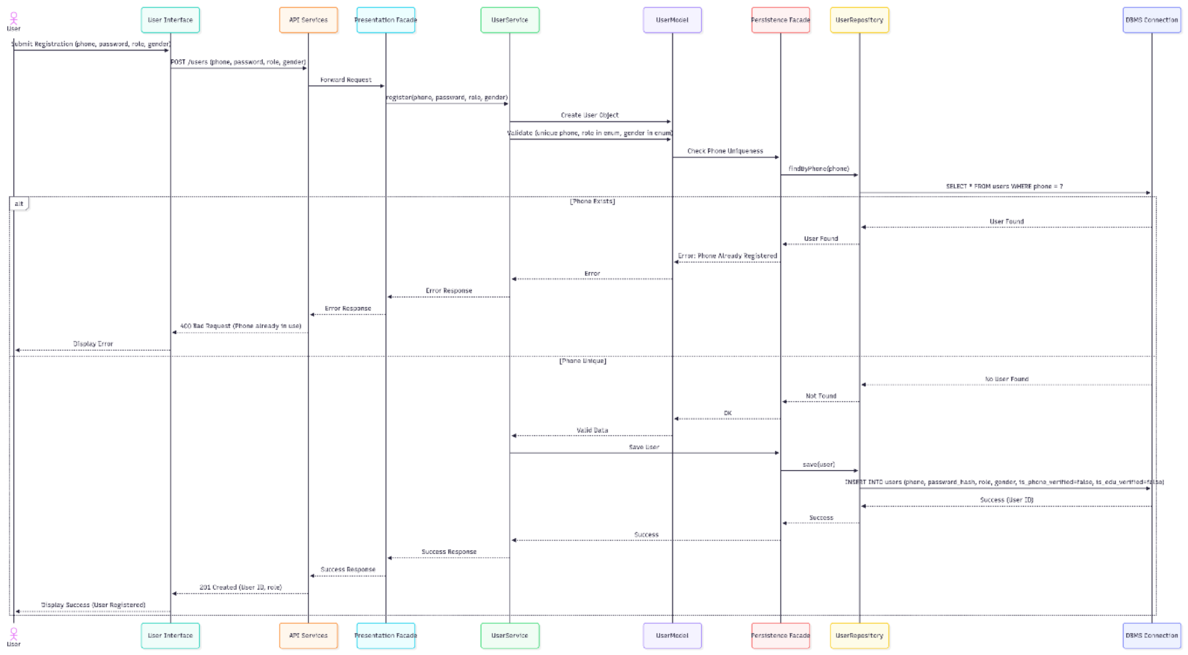
C) Front-end: Main UI Components & Interactions (Django templates)

- **SearchComponent:**
    - **Interaction: Handles user input for filters (price, room type, female-only, roommates allowed, student discount) and fetches listings from Firebase based on Riyadh campus distances. Triggers ListingDetailComponent on selection.**
    - **Sub-Components:**
        - **FilterDropdown: Updates filter state and queries listings.**
        - **ListingCard: Displays title, price, and thumbnail, clickable to details.**
- **ListingDetailComponent:**
    - **Interaction: Fetches and displays full listing data, allows bookmarking (local state), and triggers MessagingComponent for landlord communication. Updates status via submitForReview() if owner edits.**
    - **Sub-Components:**
        - **PhotoGallery: Cycles through listing photos.**
        - **MessageButton: Initiates message with landlord.**
        - **BookmarkButton: Toggles bookmark state.**
- **RoommateMatchComponent:**
    - **Interaction: Displays active Roommate posts, applies filters (budget, room type, gender), and allows students to publishPost(). Triggers MessagingComponent for contact.**
    - **Sub-Components:**
        - **RoommateCard: Shows author details and notes.**
        - **AdForm: Collects post data and calls publishPost().**
- **LandlordDashboardComponent:**
    - **Interaction: Lists owner’s listings, allows editing (updates attributes), and submits for review. Shows messages and reviews tied to listings.**
    - **Sub-Components:**
        - **ListingManager: Edits and calls updateAvailability() or submitForReview().**
        - **MessageList: Displays inbox with updateStatus() integration.**
        - **ReviewDisplay: Shows rating and comment.**
- **MessagingComponent:**

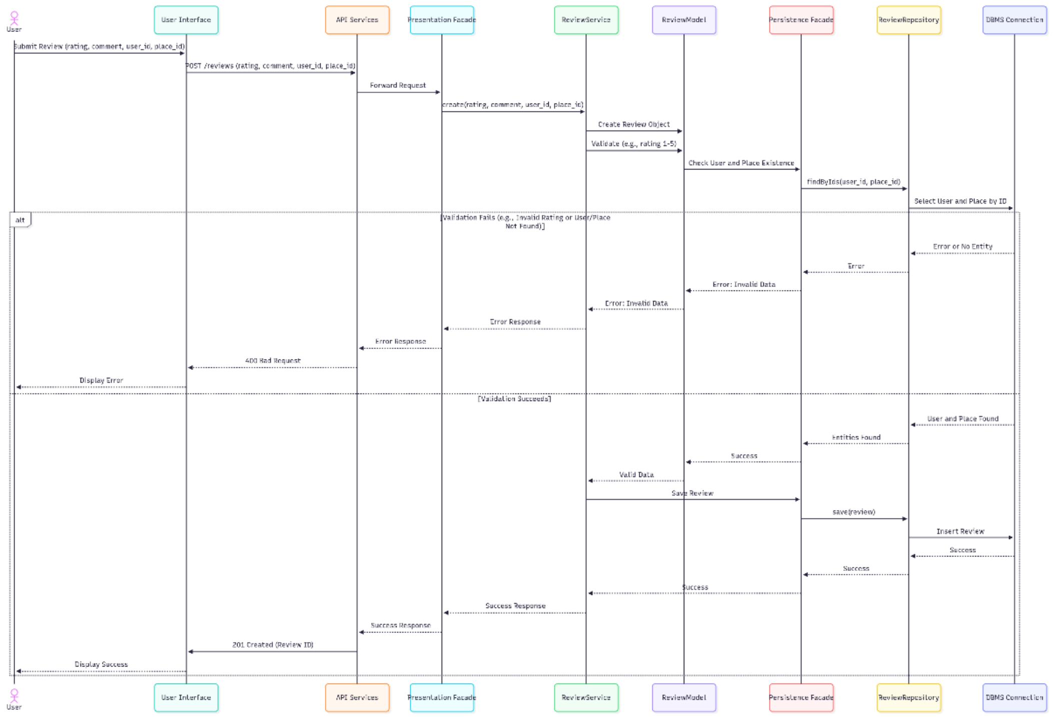
- **Interaction: Manages chat threads between users, sends messages via sendMessage(), and updates is_read on view. Supports reporting.**
- **Sub-Components:**
    - **ChatThread: Lists messages with timestamps.**

4- Sequence Diagrams:

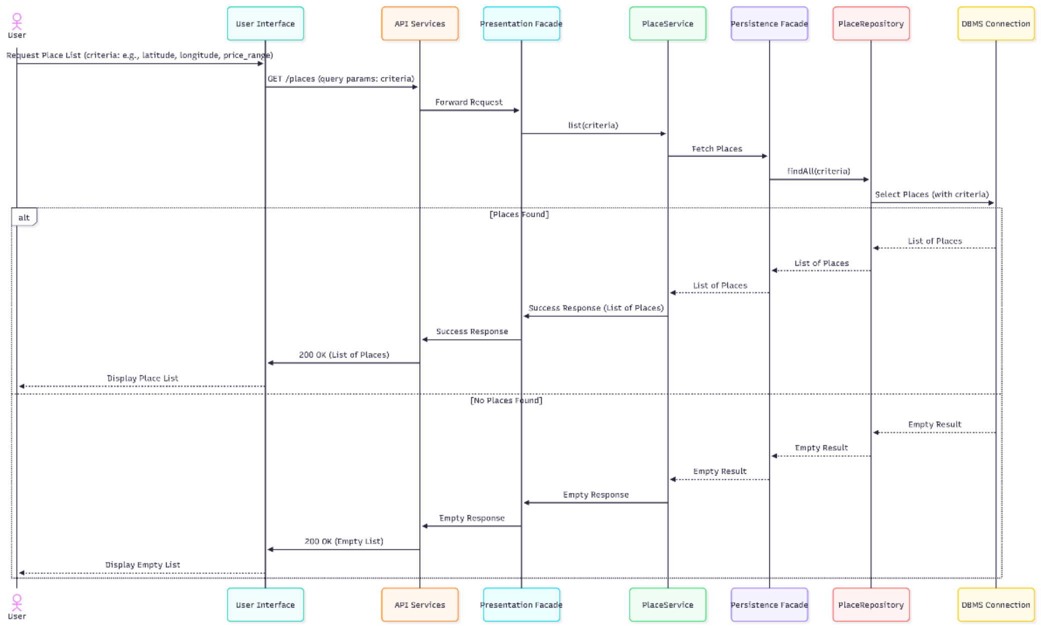
1- User Registration:



2-Review submissions:



3-Fetch places:



5- API Documentation:

List of External APIs

- **Google Maps API**
    - **Purpose: Used for geolocation services, including calculating distances to campuses, validating addresses, and displaying maps for listings in Riyadh. Chosen for its robust geocoding, distance matrix, and mapping capabilities, which are essential for the location-based filtering in the UniStay KSA MVP.**
    - **Integration: Will be queried via HTTP requests (e.g., GET /maps/api/geocode/json) with parameters like address and API key, returning JSON with latitude/longitude data.**
- **Twilio API**
    - **Purpose: Implements messaging functionality for in-app communication between users (e.g., students and landlords). Chosen for its reliable SMS and voice capabilities, supporting the MVP's requirement for secure, real-time messaging with local compliance (e.g., Saudi regulations).**
    - **Integration: Uses HTTP POST requests (e.g., POST /2010-04-01/Accounts/{AccountSid}/Messages) with JSON payload containing sender, recipient, and message body, returning a JSON response with a SID and status.**
- **Kaggle Dataset**
    - **Purpose: Provides initial data for listings, user preferences, or roommate matching (e.g., housing data or student demographics). Chosen as a cost-effective way to populate the MVP with sample data given the $0 budget constraint.**
    - **Integration: Data is downloaded as CSV or JSON files and imported into Firebase, not a live API. This is a one-time or periodic data source rather than a real-time API, so it's a static resource rather than an interactive service.**

Table of Internal API Endpoints

Endpoint	HTTP Method	Input Format	Output Format	Description
/users	**POST**	**JSON { "phone": "string", "role": "enum", "gender": "enum", ... }**	**JSON { "id": "integer", "status": "string", ... }**	**Creates a new user with validation (e.g., unique phone).**

				Returns user ID on success.
/users/{id}	GET	Query params (e.g., ?fields=phone)	JSON { "id": "integer", "phone": "string", ... }	Retrieves user details by ID. Returns 404 if not found.
/users/{id}	PUT	JSON { "phone": "string", "gender": "enum", ... }	JSON { "id": "integer", "status": "string", ... }	Updates user attributes. Returns updated user data or 400 for invalid data.
/users/{id}	DELETE	None	JSON { "message": "string", "status": "string" }	Deletes a user. Returns success message or 404 if not found.
/listings	POST	JSON { "owner_id": "integer", "title": "string", "monthly_price": "float", ... }	JSON { "id": "integer", "status": "string", ... }	Creates a new listing. Returns listing ID on success.
/listings/{id}	GET	Query params (e.g., ?include=amenities)	JSON { "id": "integer", "title": "string", ... }	Retrieves listing details by ID. Returns 404 if not found.
/listings/{id}	PUT	JSON { "title": "string", "monthly_price": "float", ... }	JSON { "id": "integer", "status": "string", ... }	Updates listing attributes. Returns updated data or 400 for invalid data.
/listings/{id}	DELETE	None	JSON { "message": "string", "status": "string" }	Deletes a listing. Returns success message or 404 if not found.
/reviews	POST	JSON { "author_id": "integer", "target_type": "enum", "target_id": "integer", "rating":	JSON { "id": "integer", "status": "string", ... }	Submits a new review. Returns review ID on success.

		"smallint", "comment": "text" }		
/reviews/{id}	GET	None	JSON { "id": "integer", "rating": "smallint", ... }	Retrieves review details by ID. Returns 404 if not found.
/reviews/{id}	PUT	JSON { "rating": "smallint", "comment": "text" }	JSON { "id": "integer", "status": "string", ... }	Updates review content. Returns updated data or 400 for invalid data.
/reviews/{id}	DELETE	None	JSON { "message": "string", "status": "string" }	Deletes a review. Returns success message or 404 if not found.
/messages	POST	JSON { "sender_id": "integer", "recipient_id": "integer", "body": "text" }	JSON { "id": "integer", "twilio_sid": "string", "status": "string", ... }	Sends a new message via Twilio. Returns message ID and status.
/messages/{id}	GET	None	JSON { "id": "integer", "body": "text", ... }	Retrieves message details by ID. Returns 404 if not found.
/messages/{id}	PUT	JSON { "status": "enum" }	JSON { "id": "integer", "status": "string", ... }	Updates message status (e.g., read). Returns updated data.
/messages/{id}	DELETE	None	JSON { "message": "string", "status": "string" }	Deletes a message. Returns success message or 404 if not found.
/roommates	POST	JSON { "author_id": "integer", "budget_min": "float",	JSON { "id": "integer",	Publishes a new roommate post.

		"budget_max": "float", ... }	"status": "string", ... }	Returns post ID on success.
/roommates /{id}	GET	None	JSON { "id": "integer", "budget_min": "float", ... }	Retrieves roommate post details by ID. Returns 404 if not found.
/roommates /{id}	PUT	JSON { "budget_min": "float", "notes": "text", ... }	JSON { "id": "integer", "status": "string", ... }	Updates roommate post attributes. Returns updated data or 400 for invalid data.
/roommates /{id}	DELETE	None	JSON { "message": "string", "status": "string" }	Deletes a roommate post. Returns success message or 404 if not found.

6- SCM and QA Strategies:

SCM Strategy

- **Version Control Tool: Git, hosted on GitHub or GitLab, will be used for version control to track changes, collaborate, and manage the codebase.**
- **Branching Strategy:**
    - **Main Branch: Represents the production-ready code. Only stable, tested releases are merged here.**
    - **Development Branch: Serves as the integration branch for ongoing work. All feature branches are merged here after review, and it's deployed to a staging environment for testing.**
    - **Feature Branches: Created for each task or feature (e.g., feature/user-auth, feature/listing-creation). Named descriptively and branched off development. Developers work on these independently.**
    - **Hotfix Branches: Branched from main for urgent production fixes (e.g., hotfix/bug-123). Merged back into both main and development after validation.**
- **Processes:**
    - **Regular Commits: Developers commit changes frequently with clear, concise messages (e.g., "Add user validation logic") to maintain a traceable history.**
    - **Code Reviews: Every pull request (PR) from a feature branch to development requires at least one peer review. Reviewers check for code quality, adherence to standards, and bug fixes.**
    - **Pull Requests: Used to merge feature branches into development. PRs include a description of changes, test results, and must pass CI/CD checks before merging.**
    - **Tagging: Major releases (e.g., v1.0.0) are tagged on the main branch for version tracking.**

QA Strategy

- **Testing Strategy:**
    - **Unit Tests: Written for individual components (e.g., user authentication logic, listing validation). Ensures functions work as expected in isolation.**
    - **Integration Tests: Verify interactions between components (e.g., API endpoints with Firebase database). Ensures data flow and business logic are correct.**
    - **Manual Testing: Conducted for critical user flows (e.g., listing creation, messaging) to validate UI/UX and edge cases, especially given the bilingual (AR/EN) requirement.**

o End-to-End (E2E) Testing: Simulated user journeys (e.g., login to listing search) using automated tools to ensure the system works holistically.

- **Testing Tools:**
  - **Jest: For unit and integration tests, particularly for JavaScript/React components and Firebase functions.**
  - **Postman: For manual and automated API testing, validating endpoints (e.g., POST /users, GET /listings) with mock data.**
  - **Cypress: For E2E testing, automating browser-based scenarios like form submissions and navigation.**
- **Deployment Pipeline:**
  - **Staging Environment: Deployed from the development branch using Firebase Hosting and Functions. Updated after every PR merge, allowing QA to test new features with sample data (e.g., from Kaggle).**
  - **Production Environment: Deployed from the main branch after thorough testing in staging. Uses Firebase's CI/CD to automate builds and deployments, triggered by tagged releases.**
  - **Pipeline Steps:**
    1. **Build: Compile code and run unit tests.**
    2. **Test: Execute integration and E2E tests.**
    3. **Deploy: Push to staging or production based on branch.**
    4. **Monitor: Log errors and user feedback post-deployment.**
  - **Rollback Plan: If issues arise in production, revert to the previous main tag and investigate the failed deployment.**

7- Technical Justifications

Constraints & Goals (what drives our choices)

- **Timeline: 3 months, pilot in Riyadh.**
- **Team skills: Stronger in Python than JavaScript/PHP.**
- **MVP features: Listings + filters (geo distance), simple in-app chat, phone OTP, AR/EN (RTL), server-rendered pages OK.**
- **Ops: Low cost, low complexity, fast to ship; future path to scale.**

Backend Frameworks

Django (chosen)

Pros

- **Batteries-included: auth, ORM, admin, i18n/RTL, forms, CSRF, security middleware.**
- **Django Channels for in-app chat; easy sessions/caching with Redis.**
- **Perfect fit with PostgreSQL + PostGIS for distance-to-campus filters.**
- **Rapid CRUD scaffolding and Django Admin for moderators.**
- **Mature ecosystem (DRF, django-storages, allauth, etc.).**

Cons

- **Monolith by default (fine for MVP, but needs modularization later).**
- **ASGI/WebSockets setup is slightly more to configure (still straightforward).**

Why Django wins for us

- **Matches team's Python strength, ships fastest with the least glue code, and covers security/i18n/admin out-of-the-box.**

Flask (considered)

Pros

- **Lightweight, flexible; minimal boilerplate.**
- **Easy to learn; we've used it before.**

Cons

- **You assemble everything (auth, admin, forms, i18n, security) → time sink.**
- **WebSockets/chat, role-based admin, and moderation require extra libraries & plumbing.**

Why not Flask now

- We want new learning beyond past Flask work, and we need speed: Django saves weeks on auth/admin/i18n/security.

PHP (Laravel or raw PHP; considered)

Pros

- Huge hosting ecosystem; Laravel offers great scaffolding (auth, queues, mail).
- Many developers/tools; can be very productive.

Cons

- Team not currently PHP-focused → ramp-up time we don't have.
- Switching stacks mid-semester increases risk; learning curve > MVP window.

Why not PHP now

- Even if strong for web, our time + skill profile favors Python/Django. PHP may be "best" for some teams; it isn't for this team/semester.

Frontend Approach

Django Templates (vanilla HTML/CSS + light JS) — chosen

Pros

- Fastest path: server-rendered pages; minimal state management.
- No separate SPA build/deploy; SEO + i18n + RTL are trivial.
- Lower cognitive load for the team; great for forms (listing create/edit), moderation.
- Can progressively enhance with small JS (Leaflet map, AJAX filters).

Cons

- Not a SPA; complex client-side interactions are more manual.
- If we later need rich real-time UX, we may refactor specific screens.

Why templates win

- Fits MVP scope (search, details, chat, dashboards) with minimal overhead; we can add DRF endpoints incrementally where needed.

React SPA (considered)

Pros

- Excellent for rich interactive UIs; component ecosystem.

Cons

- Extra stack (Node build, routing, state, auth flows) → more time.
- Team is stronger in Python; JS/React ramp-up would cut into delivery.

Why not React now

- Overkill for an MVP that's largely form/list/detail; we'll revisit if/when UX needs demand it.

Data & Infra Choices

PostgreSQL (+ PostGIS) — chosen

Pros

- Native geo queries (distance, bounding box), robust indexing.
- Reliable transactions; well-supported by Django ORM.

Cons

- Slightly more setup than SQLite (but we need the geo).

Why

- "Distance to campus" is a core filter. PostGIS makes it trivial and fast.

Redis — chosen

Pros

- Sessions/cache; Channels backend for chat; rate-limit OTP endpoints.
  Cons
- Additional service to run.
  Why
- Enables in-app messaging and keeps responses snappy.

External services

- Twilio Verify (SMS/WhatsApp): Fast, reliable OTP → sets is_phone_verified.
- Google Maps (Geocoding + Maps JS): Geocode addresses; render pins on listings.
- Photo Storage (local now, CDN later): Keep MVP free/lean; swap to S3/CDN when needed.

Architecture Rationale

- Django monolith with modular apps (auth, listings, messaging, reports) → fastest to market; easy to moderate via Django Admin.
- Server-rendered UI reduces moving parts; we still expose JSON endpoints (DRF) selectively (search filters, chat polling).

- **Channels + Redis gives basic real-time chat without a separate microservice.**
- **PostGIS future-proofs the core “near campus” experience.**

Risks & Mitigations

- **Risk: Server-rendered UX could feel basic.**
  Mitigation: Add progressive enhancement (AJAX filters, infinite scroll) and DRF endpoints as needed.
- **Risk: WebSockets complexity.**
  Mitigation: Start with polling; graduate to Channels when stable.
- **Risk: Vendor lock-in (Twilio/Maps).**
  Mitigation: Keep a Services Adapter layer to swap providers later.

Final Decision

- **Backend: Django (Python) for fastest, secure MVP with built-ins (auth/admin/i18n).**
- **Frontend: Django templates (vanilla HTML/CSS + light JS) to minimize complexity and ship within 3 months.**
- **Data/Infra: PostgreSQL + PostGIS, Redis, Twilio, Google Maps, local storage now/CDN later.**