Turista API — Full Documentation

> Combined single-file version of the documentation suite.

Turista API Documentation

Welcome to the Turista API documentation. This guide covers everything you need to develop against, operate, and contribute to the project.

What is Turista?

Turista is a Laravel 13 API for managing vacation rentals, reservations, payments, and on-arrival guest bookings. It supports multiple roles — super admin, admin, owner, employee, and customer — with role-based access control and a stateless Sanctum authentication layer.

Quick links

- Top-level README — one-liner description, stack, quick start, and verification commands.
- Project Progress — completed features and recent fixes.
- Remaining Fixes — intentionally retained technical debt.
- Security Audit — detailed security review.

Start here

If you are...	Start with...
A new developer joining the team	`development/setup.md`
An API consumer building a client	`api/overview.md`

An owner or employee using the API	`api/roles/owner.md` or `api/roles/employee.md`
A DevOps engineer deploying the app	`operations/deployment.md`
A security reviewer	`security/overview.md`

Documentation map

Architecture

- `architecture/overview.md` — high-level architecture and design principles.
- `architecture/directory-structure.md` — what lives where.
- `architecture/authentication.md` — Sanctum, roles, and permissions.
- `architecture/data-flow.md` — request lifecycle and service layer.
- `architecture/error-handling.md` — exception mapping and response format.

API reference

- `api/overview.md` — base URL, versioning, auth headers, rate limits.
- `api/authentication.md` — login, registration, OTP, password reset.
- `api/roles/admin.md` — admin endpoints.
- `api/roles/owner.md` — owner endpoints.
- `api/roles/employee.md` — employee endpoints.
- `api/roles/customer.md` — customer endpoints.
- `api/auto-generated.md` — Laravel Request Docs setup.

Business domains

- `domains/users-and-profiles.md`
- `domains/buildings-and-units.md`
- `domains/reservations.md`
- `domains/billing.md`
- `domains/promo-codes.md`
- `domains/locations.md`
- `domains/notifications.md`

Development

- `development/setup.md`
- `development/testing.md`
- `development/code-style.md`
- `development/common-tasks.md`

Operations

- `operations/deployment.md`
- `operations/environment-variables.md`
- `operations/scheduled-tasks.md`
- `operations/monitoring.md`

Security

- `security/overview.md`
- `security/audit-report-summary.md`
- `security/checklist.md`

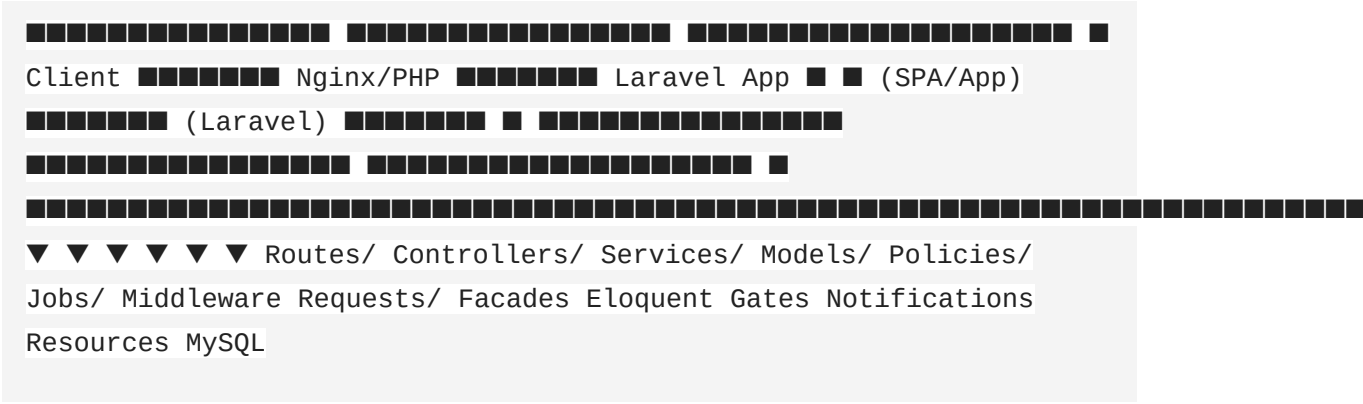
Contributing to docs

- Keep files focused on one concern.
- Prefer links over duplication. API schemas are maintained by Laravel Request Docs; the manual docs explain concepts and workflows.
- Run the verification commands in the top-level README after any code change.

Architecture Overview

Turista is an API-first Laravel 13 application. It exposes a JSON REST API under `/api/v1` and keeps frontend assets minimal.

High-level components



Tech stack

Layer	Technology
Framework	Laravel 13
Language	PHP 8.4
Database	MySQL 8.0 (dev), SQLite in-memory (tests)
API authentication	Laravel Sanctum
Roles & permissions	Spatie Laravel Permission
Media uploads	Spatie Media Library
Auditing	OwenIt Auditing
Frontend build	Vite + Tailwind CSS 4
Testing	Pest PHP 4
Queues	Database (default)

Design principles

Thin controllers, fat services. Controllers validate input, authorize actions, and delegate business logic to service classes in `app/Services/`. Services are exposed through facades in `app/Facades/`.

Policy-based authorization. Every resource has a policy in `app/Policies/`. Route middleware checks roles; policies check ownership and state. `super_admin` bypasses all policy checks via `Gate::before`.

Availability ledger. The `unit_availabilities` table stores one row per unit per night. This makes date-range conflict checks simple and reliable.

Invoice/receipt mirroring. When a reservation is created or modified, the system generates an owner-facing invoice and a customer-facing receipt with matching financial totals.

Scheduled notifications. Reminders are stored as `ScheduledNotification` rows and dispatched by a scheduled command through queue jobs.

Morph relationships. `Reservation` and `PendingReservation` are polymorphically linked to either a `Customer` or a `PendingCustomer`, supporting both online and on-arrival booking flows.

Key entry points

- **HTTP API:** `routes/api.php`
- **Console commands:** `routes/console.php`
- **Exception handling:** `bootstrap/app.php`
- **Service container bindings:** `app/Providers/AppServiceProvider.php`

Directory Structure

This document explains the purpose of each major directory and file group in the Turista project.

Top level

Path	Purpose
`app/`	Application code (controllers, models, services, etc.).

`bootstrap/`	Laravel bootstrapping and exception handling.
`config/`	Configuration files.
`database/`	Migrations, seeders, factories, and location data.
`docs/`	Project documentation (this suite).
`public/`	Web server document root.
`resources/`	CSS, JS, and email views.
`routes/`	Route definitions.
`storage/`	Logs, cache, uploads, and compiled files.
`tests/`	Pest feature and unit tests.
`vendor/`	Composer dependencies.
`node_modules/`	NPM dependencies.

`app/`

Path	Purpose
`Console/Commands/`	Artisan commands (`CreateSuperAdmin`, `DownloadLocations`, `ReleaseExpiredPendingReservations`, `SendDueScheduledNotifications`).
`Exceptions/`	Custom exceptions such as `ReservationUnavailableException`.

`Facades/`	Laravel facades for services (`ReservationService`, `PaymentService`, `FilterService`, etc.).
`Http/Controllers/`	HTTP controllers grouped by area (API auth, admin, owner, customer, core resources).
`Http/Middleware/`	Custom middleware (`EnsureUserIsVerified`, `SecurityHeadersMiddleware`).
`Http/Requests/`	Form request classes for validation.
`Http/Resources/`	API resource transformers.
`Jobs/`	Queue jobs (`DispatchScheduledNotification`).
`Mail/`	Mailable classes (`PasswordResetMail`).
`Models/`	Eloquent models.
`Notifications/`	Notification classes and custom channels (database, WhatsApp).
`Policies/`	Authorization policies for every major model.
`Providers/`	Service providers (`AppServiceProvider`, etc.).
`Rules/`	Custom validation rules (`PhoneNumber`, `WhatsAppNumber`, `PhotoFileRules`).

`Services/`	Business logic services (`ReservationService`, `PaymentService`, `FilterService`, etc.).
`Traits/`	Reusable traits (`HandlesMediaPhotos`).

`database/`

Path	Purpose
`data/`	JSON/PHP location data used by seeders.
`factories/`	Model factories for tests and seeding.
`migrations/`	All database migrations, ordered by timestamp.
`seeders/`	`DatabaseSeeder`, `RolesAndPermissionsSeeder`, `LocationSeeder`.

`routes/`

File	Purpose
`api.php`	All API routes (prefixed with `/api/v1`).
`console.php`	Scheduled console commands.
`web.php`	Minimal web routes (health check, password-reset email view).

`resources/`

Path	Purpose
`css/app.css`	Tailwind CSS import.
`js/app.js`	Minimal Vite entry point.
`views/emails/password-reset.blade.php`	Password reset email template.

`tests/`

Path	Purpose
`Feature/`	High-level HTTP/feature tests.
`Unit/`	Isolated unit tests for services, policies, and models.
`Pest.php`	Pest configuration; applies `RefreshDatabase` to feature/unit tests.
`TestCase.php`	Base test case with shared helpers and CoreVerde fakes.

`config/`

Key files include:

- `app.php` — application name, environment, timezone, locale.
- `auth.php` — default `api` guard and password reset settings.
- `sanctum.php` — token expiration and stateful domains.
- `audit.php` — OwenIt Auditing configuration.
- `request-docs.php` — Laravel Request Docs settings.

- `locations.php` — countries to seed.

Authentication & Authorization

Turista uses a stateless token architecture built on Laravel Sanctum. Roles and permissions are managed by Spatie Laravel Permission.

Authentication flow

1. **Registration.** A user creates an account as an owner or customer. The system creates a `User` record plus the matching profile (`Owner` or `Customer`) and sends an OTP.
2. **Verification.** The user submits the OTP. On success, `is_verified` is set to `true`.
3. **Login.** The user sends email/phone and password. If credentials are valid and the account is verified, a Sanctum personal access token is returned.
4. **Authenticated requests.** The client sends `Authorization: Bearer {token}` on subsequent requests.
5. **Logout.** The current token is deleted.

Sanctum configuration

- Default guard: `api`
- Token expiration: 1 week (`60 * 24 * 7` minutes in `config/sanctum.php`)
- Stateful domains are read from `SANCTUM_STATEFUL_DOMAINS`.

Roles

`RolesAndPermissionsSeeder` creates the following roles against the `api` guard:

- `super_admin`
- `admin`
- `owner`
- `employee`
- `customer`

Permissions

Key permissions include:

- `manage_platform`
- `approve_buildings`
- `manage_employees`
- `manage_buildings`
- `manage_units`
- `view_reservations`
- `manage_reservations`
- `manage_customers`
- `apply_promo_codes`

Roles map to permissions in the seeder. Controllers and routes use role middleware (`role:owner`, `role:owner|employee`, etc.) for coarse access and policies for fine-grained authorization.

Middleware

Middleware	Alias	Purpose
`EnsureUserIsVerified`	`verified`	Returns `403` if the authenticated user is not verified.
Spatie `RoleMiddleware`	`role`	Restricts routes by role.
Spatie `PermissionMiddleware`	`permission`	Restricts routes by permission.
Spatie `RoleOrPermissionMiddleware`	`role_or_permission`	Restricts by role or permission.

Super-admin bypass

`AppServiceProvider` registers a `Gate::before` callback that grants `super_admin` users access to every policy check. This is the only role that bypasses policies.

Verified accounts

Several routes use both `auth:api` and `verified` middleware. Unverified users can register, verify OTP, and resend OTP, but cannot access protected resources.

Owner approval

Owners have an additional `status` field (`pending`, `active`, `suspended`) and an `is_verified` flag on the `Owner` model. Admin users can verify owners via `POST /api/v1/owners/{owner}/verify`. Many owner actions require the owner to be approved (`status = active`).

Employees

Employees belong to an `Owner` and optionally to a `Building`. Their `status` can be `active` or `suspended`. Active employees can perform owner-delegated actions such as managing reservations.

Data Flow

This document describes how a typical HTTP request moves through the Turista application.

Standard request lifecycle

```
Request ▼ Route (routes/api.php) ▼ Middleware (auth, verified, role, throttle) ▼ Form Request validation (app/Http/Requests/) ▼ Controller action (app/Http/Controllers/) ▼ Policy authorization ($this->authorize('view', $model)) ▼ Service layer (app/Services/ via Facades) ▼ Eloquent models & database ▼ API Resource (app/Http/Resources/) ▼ JSON response
```

Example: creating a reservation

1. **Route.** `POST /api/v1/customer/reservations` hits `ReservationController@store`.
2. **Middleware.** `auth:api` and `role:customer` ensure an authenticated customer.
3. **Validation.** `ReservationRequest` validates dates, guest counts, `unit_id`, and promo code.
4. **Controller.** The controller calls `ReservationService::createReservationForCustomer()`.
5. **Service logic.**
6. Looks up the unit and checks availability via `UnitAvailabilityService`.
7. Validates the promo code via `PromoCodeService`.
8. Calculates the total price.
9. Creates the `Reservation` record.
10. Books availability dates.
11. Generates an `Invoice` for the owner and a `Receipt` for the customer.
12. Schedules reminder notifications via `ReservationReminderScheduler`.
13. **Response.** A `ReservationResource` is returned with the new reservation, invoice, and receipt.

Service layer and facades

Business logic lives in `app/Services/` and is consumed through facades in `app/Facades/`. This keeps controllers thin and makes the logic testable in isolation.

Service	Responsibility
`ReservationService`	Reservation lifecycle, pricing, availability booking, documents.
`PaymentService`	Payments, refunds, wallet updates.
`UnitAvailabilityService`	Date ledger operations (book, hold, release, block).
`PromoCodeService`	Bulk generation and redemption validation.

`OtpService`	OTP generation, caching, and verification.
`WhatsAppService`	CoreVerde WhatsApp HTTP integration.
`DashboardMetrics`	Revenue, occupancy, and user-growth aggregations.
`FilterService`	Query filtering for list endpoints.
`SequenceService`	Atomic document number generation.
`ReservationReminderScheduler`	Schedules reminder notifications.

Availability ledger

`unit_availabilities` has one row per unit per night with a `status` of `available`, `blocked`, or `booked`. When a reservation is created, the service changes the matching rows to `booked` and links them to the reservation. Cancellations reverse this process.

Scheduled notifications

When a reservation is confirmed, `ReservationReminderScheduler` creates `ScheduledNotification` rows for check-in, check-out, and evacuate reminders. The `notifications:send-due` scheduler command runs every minute and dispatches `DispatchScheduledNotification` jobs for due rows. The job sends the notification through the database channel (in-app notification) and WhatsApp channel.

Sequence numbering

`SequenceService` increments counters stored in the `sequences` table to produce zero-padded document numbers such as `INV-00001`, `REC-00001`, and `RES-00001`.

Error propagation

Controllers catch domain exceptions (e.g., `ReservationUnavailableException`) and return structured JSON errors. Unexpected exceptions are rendered as JSON by the handler in `bootstrap/app.php`.

Error Handling

Turista is an API-first application, so all errors are returned as JSON. Exception rendering is configured in `bootstrap/app.php`.

Response envelope

A typical error response follows this shape:

```
{ "message": "The given data was invalid.", "errors": { "email": ["The email field is required."] } }
```

For non-validation errors the response may contain only `message`:

```
{ "message": "Reservation dates are not available." }
```

Mapped exceptions

Exception type	HTTP status	Notes
ValidationException	422	Returned when Form Request validation fails.
AuthenticationException	401	Returned by `auth:api` middleware.

AuthorizationException	403	Returned by policies or `verified` middleware.
ModelNotFoundException	404	Returned when a route-bound model is missing.
ReservationUnavailableException	422	Returned when requested dates cannot be booked or blocked.
HttpException	as set	Generic HTTP exceptions.

Exception handler

`bootstrap/app.php` configures the exception handler to render API exceptions as JSON. Validation, authentication, authorization, and model-not-found exceptions are mapped to clean responses. In production, detailed stack traces are hidden (`APP_DEBUG=false`).

Domain exceptions

`App\Exceptions\ReservationUnavailableException` is thrown by the reservation and availability services when a unit cannot be booked or blocked for the requested dates. Controllers catch this and return a `422` response with a clear message.

Validation errors

Form Request classes in `app/Http/Requests/` centralize validation rules. When validation fails, Laravel returns a 422 response with the `errors` object keyed by field name.

Logging

Unexpected exceptions are logged to `storage/logs/laravel.log`. Operators can tail these logs or forward them to a centralized logging service.

Client guidance

API clients should:

- Check the HTTP status code first.
- Read `message` for a human-readable description.
- Parse `errors` for field-level validation feedback.
- Not rely on the exact text of exception messages in production.

API Overview

Turista exposes a JSON REST API under the base URL `/api/v1`.

Base URL

```
https://{your-domain}/api/v1
```

Versioning

The current API version is `v1`. Versioning is path-based. Future versions will use a new path prefix (e.g., `/api/v2`).

Headers

All requests should include:

```
Accept: application/json Content-Type: application/json
```

Authenticated requests must also include:

```
Authorization: Bearer {sanctum_token}
```

Authentication

See `authentication.md` for details on login, registration, OTP verification, password reset, and logout.

Rate limiting

Public authentication endpoints (login, register, OTP) are rate-limited by IP and contact information. Authenticated endpoints generally use the default Laravel throttle. Specific limits are configured in `RouteServiceProvider` or route middleware.

Response format

Successful responses return a 2xx status and a JSON body. The shape depends on the endpoint; see `auto-generated.md` for full schemas.

List endpoints typically return paginated data:

```
{ "data": [...], "links": {...}, "meta": {...} }
```

Single-resource endpoints return a resource object:

```
{ "data": {...} }
```

Error format

Errors are returned as JSON with an HTTP 4xx/5xx status. See `architecture/error-handling.md` for details.

Filtering and sorting

List endpoints accept query parameters for filtering, sorting, and pagination. Common patterns:

- `?page=2`
- `?per_page=20`
- `?status=active`
- `?sort=-created_at`

Exact parameter names vary by endpoint. Use Laravel Request Docs for the full list.

Role-specific endpoints

The API is organized by actor:

- `Admin endpoints`
- `Owner endpoints`
- `Employee endpoints`
- `Customer endpoints`

Auto-generated reference

For exhaustive request/response schemas, see `auto-generated.md` on how to use Laravel Request Docs.

Authentication Endpoints

This document covers the public and authenticated auth endpoints.

Public endpoints

Login

```
POST /api/v1/login
```

Request:

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

Response includes a Sanctum token:

```
{ "data": { "user": {...}, "token": "{sanctum_token}" } }
```

Unverified accounts receive a `403` response.

Register owner

```
POST /api/v1/register/owner
```

Creates a `User` and an `Owner` profile. Sends an OTP.

Register customer

```
POST /api/v1/register/customer
```

Creates a `User` and a `Customer` profile. Links any prior `PendingCustomer` reservations.

Verify OTP

```
POST /api/v1/verify-otp
```

Validates the OTP and marks the user as verified.

Verify account

```
POST /api/v1/verify-account
```

Alternative verification endpoint.

Resend OTP

```
POST /api/v1/resend-otp
```

Resends the OTP if the user exists and is unverified.

Forgot password

```
POST /api/v1/forgot-password
```

Creates a password-reset token and sends a reset email.

Reset password

```
POST /api/v1/reset-password
```

Validates the reset token and updates the password.

Authenticated endpoints

These require a valid bearer token.

Logout

```
POST /api/v1/logout
```

Deletes the current access token.

Update password

```
PUT /api/v1/update-password
```

Changes the authenticated user's password after verifying the current password.

Environment note on OTPs

OTP codes are exposed in API responses only in `local` and `testing` environments for development convenience. In production, OTPs are sent through the configured channel only.

Token usage

After login, include the token in all subsequent requests:

```
Authorization: Bearer {sanctum_token}
```

Tokens expire after one week by default.

Admin API

Admins manage the platform, locations, and owner verification.

Role requirements

- Most admin routes require the `admin` or `super_admin` role.
- Creating another admin requires the `super_admin` role.

Dashboard

Method	Path	Description
GET	`/api/v1/admin/dashboard`	Revenue and user-growth summary with filters.

Reservations & buildings

Method	Path	Description
GET	`/api/v1/admin/reservations`	List all reservations.
GET	`/api/v1/admin/buildings`	List all buildings.

Locations

Admins have full CRUD over locations. Public read-only endpoints are documented in `customer.md`.

Method	Path	Description
POST	`/api/v1/admin/cities`	Create a city.
PUT	`/api/v1/admin/cities/{city}`	Update a city.
DELETE	`/api/v1/admin/cities/{city}`	Delete a city.
POST	`/api/v1/admin/countries`	Create a country.
PUT	`/api/v1/admin/countries/{country}`	Update a country.
DELETE	`/api/v1/admin/countries/{country}`	Delete a country.
POST	`/api/v1/admin/currencies`	Create a currency.
PUT	`/api/v1/admin/currencies/{currency}`	Update a currency.
DELETE	`/api/v1/admin/currencies/{currency}`	Delete a currency.
POST	`/api/v1/admin/regions`	Create a region.
PUT	`/api/v1/admin/regions/{region}`	Update a region.
DELETE	`/api/v1/admin/regions/{region}`	Delete a region.

Users & owners

Method	Path	Description
GET	`/api/v1/owners`	List owners.
POST	`/api/v1/owners/{owner}/verify`	Verify an owner profile.
GET	`/api/v1/admin/employees`	List all employees.
GET	`/api/v1/admin/notifications`	List all notifications.
POST	`/api/v1/admins`	Create a new admin (`super_admin` only).
GET	`/api/v1/customers`	List customers.
DELETE	`/api/v1/customers/{customer}`	Delete a customer.

For detailed request/response schemas, generate Laravel Request Docs (see `auto-generated.md`).

Owner API

Owners manage buildings, units, employees, reservations, billing, and promo codes.

Role requirements

- Routes require the `owner` role.
- Many actions also require the owner to be approved (`status = active`).

Dashboard & profile

Method	Path	Description
GET	`/api/v1/owner/dashboard`	Revenue and occupancy dashboard.
GET	`/api/v1/owner/profile`	Get owner profile.
PUT	`/api/v1/owner/profile`	Update owner profile.
GET	`/api/v1/owner/calendar`	Unit availability calendar.

Buildings

Method	Path	Description
GET	`/api/v1/owner/buildings`	List owner's buildings.
POST	`/api/v1/owner/buildings`	Create a building.
GET	`/api/v1/owner/buildings/{building}`	Show a building.
PUT	`/api/v1/owner/buildings/{building}`	Update a building.
DELETE	`/api/v1/owner/buildings/{building}`	Delete a building.

POST	`/api/v1/owner/buildings/{building}/photos`	Upload building photos.
POST	`/api/v1/owner/buildings/{building}/facilities`	Assign facilities.

Units

Method	Path	Description
GET	`/api/v1/owner/units`	List owner's units.
POST	`/api/v1/owner/units`	Create a unit.
GET	`/api/v1/owner/units/{unit}`	Show a unit.
PUT	`/api/v1/owner/units/{unit}`	Update a unit.
DELETE	`/api/v1/owner/units/{unit}`	Delete a unit.
POST	`/api/v1/owner/buildings/{building}/units/bulk`	Bulk-create units.
POST	`/api/v1/owner/units/{unit}/photos`	Upload unit photos.
POST	`/api/v1/owner/units/{unit}/facilities`	Assign facilities.

Employees

Method	Path	Description
GET	`/api/v1/owner/employees`	List employees.
POST	`/api/v1/owner/employees`	Create an employee.
GET	`/api/v1/owner/employees/{employee}`	Show an employee.
DELETE	`/api/v1/owner/employees/{employee}`	Delete an employee.

Note: employees update their own profiles through the shared employee endpoint.

Reservations

Method	Path	Description
GET	`/api/v1/owner/reservations`	List reservations.
POST	`/api/v1/owner/reservations`	Create a reservation.
GET	`/api/v1/owner/reservations/{reservation}`	Show a reservation.
PUT	`/api/v1/owner/reservations/{reservation}`	Update a reservation.
DELETE	`/api/v1/owner/reservations/{reservation}`	Cancel a reservation.
POST	`/api/v1/owner/reservations/{reservation}/check-in`	Check in a guest.

POST	`/api/v1/owner/reservations/{reservation}/check-out`	Check out a guest.
GET	`/api/v1/owner/about-to-end`	Reservations about to end.

Billing

Method	Path	Description
GET	`/api/v1/owner/invoices`	List invoices.
POST	`/api/v1/owner/invoices`	Create an invoice.
GET	`/api/v1/owner/invoices/{invoice}`	Show an invoice.
DELETE	`/api/v1/owner/invoices/{invoice}`	Delete an invoice (guarded by transactions).
GET	`/api/v1/owner/transactions`	List transactions.
POST	`/api/v1/owner/transactions`	Record a payment or refund.
GET	`/api/v1/owner/receipts`	List receipts.
POST	`/api/v1/owner/receipts`	Generate a receipt.

Promo codes

Method	Path	Description
GET	`/api/v1/owner/promo-codes`	List promo codes.
POST	`/api/v1/owner/promo-codes`	Bulk-generate promo codes.
GET	`/api/v1/owner/promo-codes/{promo_code}`	Show a promo code.
PUT	`/api/v1/owner/promo-codes/{promo_code}`	Update a promo code.
DELETE	`/api/v1/owner/promo-codes/{promo_code}`	Delete a promo code.

Unit availability

Method	Path	Description
GET	`/api/v1/owner/unit-availabilities`	List availability.
POST	`/api/v1/owner/unit-availabilities/block`	Block dates.
POST	`/api/v1/owner/unit-availabilities/unblock`	Unblock dates.

Facilities

Method	Path	Description
GET	`/api/v1/owner/facilities`	List facilities.
POST	`/api/v1/owner/facilities`	Create a facility.
GET	`/api/v1/owner/facilities/{facility}`	Show a facility.
PUT	`/api/v1/owner/facilities/{facility}`	Update a facility.
DELETE	`/api/v1/owner/facilities/{facility}`	Delete a facility.

For detailed schemas, see `auto-generated.md`.

Employee API

Employees act on behalf of an owner. They can manage reservations and view related data.

Role requirements

- Routes require the `employee` role.
- The employee must be `active`.
- Employees belong to an `Owner` and optionally to a specific `Building`.

Shared owner/employee routes

The following routes accept both `owner` and `employee` roles:

Method	Path	Description

GET	`/api/v1/reservations`	List reservations scoped to the owner/employee.
POST	`/api/v1/reservations/on-arrival/prepare`	Start an on-arrival booking.
POST	`/api/v1/reservations/on-arrival/validate`	Validate the on-arrival OTP.
POST	`/api/v1/reservations/{reservationId}/check-in`	Check in a guest.
POST	`/api/v1/reservations/{reservationId}/check-out`	Check out a guest.

Employee profile

Method	Path	Description
GET	`/api/v1/employee/profile`	Get own employee profile.
PUT	`/api/v1/employee/profile`	Update own employee profile.

Scoped data access

Employees see:

- Reservations related to their owner (or building, if assigned).
- Invoices, receipts, and transactions for those reservations.

- Buildings and units owned by their owner.

They cannot create or delete buildings, units, or employees.

For detailed schemas, see `auto-generated.md`.

Customer API

Customers browse the public catalog and manage their own reservations and profile.

Role requirements

- Routes require the `customer` role.
- The customer account must be verified.

Public catalog (no authentication)

Method	Path	Description
GET	`/api/v1/units`	List available units with filters.
GET	`/api/v1/units/{unit}`	Show a unit.
POST	`/api/v1/promo-codes/preview`	Preview promo-code discount.
GET	`/api/v1/countries`	List countries.
GET	`/api/v1/countries/{country}`	Show a country.

GET	`/api/v1/cities`	List cities.
GET	`/api/v1/cities/{city}`	Show a city.
GET	`/api/v1/regions`	List regions.
GET	`/api/v1/regions/{region}`	Show a region.
GET	`/api/v1/currencies`	List currencies.
GET	`/api/v1/currencies/{currency}`	Show a currency.

Customer profile

Method	Path	Description
GET	`/api/v1/customer/profile`	Get own profile.
PUT	`/api/v1/customer/profile`	Update own profile.

Reservations

Method	Path	Description
GET	`/api/v1/customer/reservations`	List customer reservations.
POST	`/api/v1/customer/reservations`	Create a reservation.

GET	`/api/v1/customer/reservations/{reservation}`	Show a reservation.
PUT	`/api/v1/customer/reservations/{reservation}`	Update a reservation.
DELETE	`/api/v1/customer/reservations/{reservation}`	Cancel a reservation.

Billing

Method	Path	Description
GET	`/api/v1/customer/invoices`	List customer invoices.
GET	`/api/v1/customer/receipts`	List customer receipts.
GET	`/api/v1/customer/transactions`	List customer transactions.

Notifications

Method	Path	Description
GET	`/api/v1/notifications`	List own notifications.
POST	`/api/v1/notifications/{notification}/mark-as-read`	Mark a notification as read.
DELETE	`/api/v1/notifications/{notification}`	Delete a notification.

For detailed schemas, see `auto-generated.md`.

Auto-Generated API Docs

Turista includes Laravel Request Docs, which generates interactive API documentation from your routes and Form Request classes.

What it covers

Laravel Request Docs can display:

- All registered routes.
- HTTP methods and paths.
- Request validation rules derived from Form Requests.
- Middleware applied to each route.
- Bearer token authentication support for testing endpoints.

Enabling the docs

Set the environment variable:

```
REQUEST_DOCS_ENABLED=true
```

Then clear the config cache if running in production:

```
php artisan config:clear
```

Accessing the docs

When enabled, visit:

```
https://{your-domain}/request-docs
```

The UI allows you to authenticate with a bearer token and make test requests against the API.

Generating static files

Laravel Request Docs can generate `api.json` and `routes.json` for external consumers. To regenerate them:

```
php artisan route:docs
```

> Check the package documentation for the exact command if this differs.

Security

Keep `REQUEST_DOCS_ENABLED=false` in production. The generated `api.json` and `routes.json` files reveal endpoint structure, middleware, and controllers. Consider adding them to `.gitignore` and generating them only in build pipelines.

Manual docs vs. auto-generated docs

The documentation in this `docs/api/` directory explains concepts, roles, and workflows. Laravel Request Docs provides the exhaustive endpoint-level schemas. Use both together: start here for context, then use Request Docs for precise payloads.

Users & Profiles

Turista uses a single `users` table as the shared identity for all actors. Role-specific data lives in profile tables.

User model

`App\Models\User` stores:

- `name`
- `email` (unique)
- `phone` (unique)
- `password` (hashed)
- `is_verified`
- timestamps and soft deletes

A user has one of the following profiles:

- `Owner`
- `Employee`
- `Customer`

Roles

Spatie Laravel Permission assigns one role per user:

- `super_admin`
- `admin`
- `owner`
- `employee`
- `customer`

Owner profile

`App\Models\Owner` stores:

- `status` — `pending`, `active`, or `suspended`
- `whatsapp_number`
- `is_verified`

The primary key `id` is also a foreign key to `users.id`. An owner has many `Buildings`, `Employees`, and `Units` through buildings.

Owners must be approved (`status = active`) before they can list properties or accept reservations.

Employee profile

`App\Models\Employee` stores:

- `owner_id` — the owner they work for
- `building_id` — optional building assignment
- `status` — `active` or `suspended`

Employees act on behalf of an owner and can manage reservations and related data. Their access is scoped to their owner (and optionally building).

Customer profile

`App\Models\Customer` stores:

- `whatsapp_number`
- `wallet` (decimal)

Customers make online bookings and receive receipts. The wallet can be used for payments or receive refunds.

Pending customer

`App\Models\PendingCustomer` supports the on-arrival booking flow. When a guest arrives without a prior account, the owner/employee can create a pending reservation linked to a pending customer. The guest verifies their phone number via OTP, at which point the pending customer is converted or linked to a real customer.

Key flows

- **Owner registration:** `User` + `Owner` created; OTP sent; admin verifies owner after activation.
- **Customer registration:** `User` + `Customer` created; OTP sent.
- **On-arrival booking:** `PendingCustomer` created with phone; OTP verified; reservation confirmed.
- **Password reset:** token generated and emailed; user resets password.

Buildings & Units

Buildings and units are the core inventory of the platform.

Building model

`App\Models\Building` represents a physical property owned by an owner.

Key fields:

- `owner_id`
- `region_id`
- `currency_id`
- `name`
- `check_in_time` / `check_out_time`
- `map_lat` / `map_lng`
- `status` — e.g., `active`, `inactive`
- `slug` (unique)
- `payment_methods` (JSON)
- `booking_conditions` (JSON)

Relationships:

- `owner`
- `region`
- `currency`
- `units`
- `facilities` (BelongsToMany)
- media photos (Spatie MediaLibrary)

Unit model

`App\Models\Unit` represents a rentable space within a building.

Key fields:

- `building_id`
- `floor`
- `name_or_number`
- `rooms`
- `base_price`
- `offer_price`
- `guest_type`
- `max_adults` / `max_children` / `max_child_age`
- `status`
- `slug` (unique)

- `booking_conditions` (JSON)

Relationships:

- `building`
- `reservations`
- `availabilities`
- `facilities` (BelongsToMany)
- media photos

Facilities

`App\Models\Facility` represents amenities such as Wi-Fi, parking, or a pool. Facilities can be attached to buildings and units.

Availability ledger

`App\Models\UnitAvailability` stores one row per unit per night.

Fields:

- `unit_id`
- `date`
- `status` — `available`, `blocked`, or `booked`
- `reservation_id`
- `pending_reservation_id`

This ledger enables reliable date-range conflict detection. When a reservation is confirmed, the matching rows become `booked`. When canceled, they revert to `available`. Owners can `block` dates to take units off the market.

Public catalog

Unauthenticated users can list and view units. The catalog supports filters for:

- Country, city, region
- Price range
- Guest counts
- Facilities

- Dates (availability check)

Bulk unit creation

Owners can bulk-create units under a building. This is useful for hotels or apartment blocks with many similar units.

Photo uploads

Buildings and units support photo uploads via Spatie MediaLibrary. The `HandlesMediaPhotos` trait centralizes upload and replacement logic. Allowed formats are typically JPG, JPEG, PNG, and WebP with size and dimension limits.

Reservations

Reservations are the central transaction in Turista. They link a customer (or pending customer) to a unit for a date range.

Reservation model

`App\Models\Reservation` stores:

- `reservation_number` (unique)
- `customer_type` / `customer_id` (morph to `Customer` or `PendingCustomer`)
- `unit_id`
- `promo_code_id`
- `check_in_date` / `check_out_date`
- `adults_count` / `children_count`
- `payment_status`
- `status` — lifecycle state
- `source` — online or on-arrival
- `total_price`
- `notes`

Status lifecycle

```
pending → confirmed → checked_in → checked_out ↓ canceled
```

- **pending** — held temporarily while awaiting OTP or confirmation.
- **confirmed** — availability is booked and invoice/receipt generated.
- **checked_in** — guest has arrived.
- **checked_out** — stay is complete; no further edits allowed.
- **canceled** — reservation cancelled; availability released.

Booking channels

Customer online booking

A verified customer selects a unit and dates. The system checks availability, applies any promo code, calculates the total, and creates a confirmed reservation.

Owner/employee on-arrival booking

When a guest arrives without a prior booking:

1. Owner/employee prepares an on-arrival reservation (`POST /api/v1/reservations/on-arrival/prepare`).
2. The system creates a `PendingReservation` linked to a `PendingCustomer` and sends an OTP.
3. The guest verifies the OTP (`POST /api/v1/reservations/on-arrival/validate`).
4. The pending reservation is converted to a confirmed `Reservation`.

Date changes

Customers and owners can update reservation dates. The service:

1. Checks availability for the new range.
2. Recalculates the total price.
3. Adjusts the invoice and receipt.
4. Handles automatic refunds if the new price is lower.

Check-in / check-out

Owner/employee users can transition a confirmed reservation to `checked_in` and later to `checked_out`. Checked-out and canceled reservations cannot be edited or canceled again.

Companions

The `reservation_companions` table was removed; companion information is stored in `notes` or handled by the client application.

Key services

- `ReservationService` — create, update, confirm, cancel, recalculate.
- `UnitAvailabilityService` — book, release, block availability dates.
- `ReservationReminderScheduler` — schedule reminder notifications.

Billing

Turista's billing model separates owner-facing invoices from customer-facing receipts and tracks all money movement through transactions.

Invoice

`App\Models\Invoice` is generated for the owner when a reservation is confirmed.

Key fields:

- `reservation_id`
- `received` enum (invoice type/role)
- `document_number`
- `price`, `total_price`, `discount`, `net_price`
- `paid_amount`, `remaining_amount`
- `paid_at`, `due_at`

Invoices cannot be deleted if transactions exist against them.

Receipt

`App\Models\Receipt` mirrors the invoice for the customer.

Key fields:

- `reservation_id`
- `invoice_id` (nullable)
- same financial fields as invoice

Receipts give customers a record of what they owe or have paid.

Transaction

`App\Models\Transaction` records each payment or refund.

Key fields:

- `invoice_id`
- `type` — `payment` or `refund`
- `payment_method` — `cash`, `card`, or `wallet`
- `amount`
- `reason`

The `PaymentService` processes payments and refunds atomically to avoid double-credits or race conditions.

Wallet

Customers have a `wallet` balance on their profile. Refunds can be credited to the wallet, and wallet balance can be used as a payment method.

Sequence numbering

`SequenceService` maintains atomic counters in the `sequences` table. Each invoice, receipt, and reservation gets a zero-padded number such as:

- `RES-00001`
- `INV-00001`
- `REC-00001`

Recalculation

When a reservation is updated (e.g., date change), `ReservationService::recalculateReservation` updates the invoice and receipt totals and triggers any required refund.

Security note

Financial operations use row-level locking and atomic updates to prevent race conditions. See the security audit for additional hardening notes.

Promo Codes

Promo codes let owners offer discounts on unit bookings.

PromoCode model

`App\Models\PromoCode` stores:

- `owner_id` (nullable; null means platform-wide)
- `code` (encrypted)
- `code_hash` (unique, used for lookups)
- `usage_limit`
- `per_customer_limit`
- `uses_count`
- `discount_value`
- `discount_type` — e.g., fixed or percentage
- `starts_at`, `expires_at`, `used_at`

The actual code is encrypted; a hash is used for validation.

Bulk generation

Owners can generate many promo codes at once. The `PromoCodeService::generateBulk` method:

1. Creates the requested number of unique codes.
2. Hashes each code.
3. Persists them with the configured limits and discount.

Validation

A promo code is valid when:

- It has started and not expired.
- It has not exceeded `usage_limit`.
- The customer has not exceeded `per_customer_limit`.
- It applies to the selected unit/owner.

Redemption

During reservation creation or recalculation, the system:

1. Looks up the code by hash.
2. Validates it against the unit and customer.
3. Applies the discount to the total price.
4. Increments `uses_count`.

Public preview

Unauthenticated users can preview the discount a promo code would apply to a unit without actually redeeming it.

Owner management

Owners can list, update, and delete their own promo codes. Admins can manage platform-wide codes (`owner_id = null`).

Locations

The location catalog provides country, city, region, and currency data used by buildings and the public search.

Models

Model	Purpose
`Country`	Top-level country.
`City`	City within a country.
`Region`	Region/neighborhood within a city.
`Currency`	Currency used by buildings.

Relationships

- `Country` has many `City`
- `City` belongs to `Country` and has many `Region`
- `Region` belongs to `City` and has many `Building`
- `Country` belongs to `Currency`
- `Building` belongs to `Region` and `Currency`

Admin management

Admins can create, update, and delete locations. The admin endpoints are prefixed with `/api/v1/admin/`.

Public read-only access

Unauthenticated users can list and show countries, cities, regions, and currencies. These endpoints power the public catalog filters.

Seeding

The `LocationSeeder` seeds the configured countries (default: Libya) from `database/data/locations.json`. To download fresh location data:

```
php artisan locations:download
```

This command fetches an external dataset, filters it by `config('locations.seed_countries')`, and writes `database/data/locations.json`.

Configuration

`config/locations.php` contains:

```
'seed_countries' => explode(',', env('SEED_COUNTRIES', 'Libya')),
```

Use the `SEED_COUNTRIES` environment variable to control which countries are seeded.

Notifications

Turista sends reservation reminders through in-app database notifications and WhatsApp messages.

ScheduledNotification model

`App\Models\ScheduledNotification` represents a reminder that should be sent at a specific time.

Key fields:

- `notifiable_type` / `notifiable_id` (morph)
- `reservation_id`
- `type` — e.g., `check_in`, `check_out`, `evacuate`
- `send_at`
- `queued_at`, `sent_at`, `cancelled_at`
- `payload` (array)

Scopes:

- `due` — notifications whose `send_at` has passed and are not yet sent/cancelled.
- `pendingForReservation` — unsent notifications for a given reservation.

Reminder types

Type	When it fires
`check_in`	Before the guest arrives.
`check_out`	Before the guest departs.
`evacuate`	When the stay should end.

Scheduler command

The `notifications:send-due` command runs every minute via `routes/console.php`:

```
Schedule::command('notifications:send-due')->everyMinute();
```

It claims due notifications and dispatches `DispatchScheduledNotification` jobs.

Dispatch job

`App\Jobs\DispatchScheduledNotification`:

1. Loads the scheduled notification and reservation.
2. Skips if the reservation is canceled or checked out.
3. Sends the `ReservationReminder` notification.
4. Marks the scheduled notification as sent.

Notification channels

`App\Notifications\ReservationReminder` is sent through two channels:

1. **`AppDatabaseChannel`** — writes to the `notifications` table for in-app display.
2. **`WhatsAppChannel`** — sends a WhatsApp message via `WhatsAppService` / CoreVerde.

User notifications

Authenticated users can list their notifications, mark them as read, and delete them.

Cancellation

When a reservation is canceled or checked out, pending scheduled notifications for that reservation are canceled to avoid sending irrelevant reminders.

Local Setup

This guide walks you through running Turista on your local machine.

Prerequisites

- PHP 8.4+
- Composer 2.9+
- MySQL 8.0+
- Node.js 20+
- NPM 10+

Step-by-step

1. **Install PHP dependencies**

```
bash composer install
```

1. **Install JavaScript dependencies**

```
bash npm install
```

1. **Create environment file**

```
bash cp .env.example .env
```

Update `.env` with your database credentials and any required third-party API keys (CoreVerde WhatsApp, mail, etc.).

1. **Generate application key**

```
bash php artisan key:generate
```

1. **Create the storage link**

```
bash php artisan storage:link
```

1. **Run migrations and seeders**

```
bash php artisan migrate --seed
```

This creates roles, permissions, and the default location data (Libya by default).

1. **Build frontend assets**

```
bash npm run build
```

Running the development stack

```
composer run dev
```

This starts the Laravel development server, a queue worker, and the Vite dev server concurrently.

Creating the first super admin

After seeding, create a super admin via CLI:

```
php artisan admin:create-super admin@example.com
```

You will be prompted for a secure password.

Downloading location data

If you need to seed additional countries:

1. Update `SEED_COUNTRIES` in `.env`.
2. Run:

```
bash php artisan locations:download php artisan db:seed
--class=LocationSeeder
```

Verifying the setup

Run the verification commands from the top-level README:

```
php artisan test vendor/bin/pint --test npm run build php artisan
route:cache php artisan optimize
```

All tests should pass and the build should complete without errors.

Testing

Turista uses Pest PHP 4 for testing. Tests run against an in-memory SQLite database.

Running tests

```
php artisan test
```

To run a specific test file:

```
php artisan test tests/Feature/Auth/LoginTest.php
```

To run with verbose output:

```
php artisan test --verbose
```

Test database

`phpunit.xml` configures SQLite in-memory mode:

```
<env name="DB_CONNECTION" value="sqlite"/> <env name="DB_DATABASE" value=":memory:"/>
```

The `RefreshDatabase` trait is applied to all feature and unit tests via `tests/Pest.php`.

Test structure

Directory	Purpose
`tests/Feature/`	End-to-end HTTP and feature tests.
`tests/Unit/`	Isolated tests for services, policies, and models.

Base test case

`tests/TestCase.php` provides shared helpers and sets up:

- Cache and permission cache flushing.
- CoreVerde HTTP faking.
- Default CoreVerde config for tests.

Writing feature tests

A typical feature test:

```
it('allows a verified customer to create a reservation', function () { $customer = User::factory() ->has(Customer::factory()) ->create() ->assignRole('customer'); $unit = Unit::factory()->create(); actingAs($customer, 'api') ->postJson('/api/v1/customer/reservations', [ 'unit_id' => $unit->id, 'check_in_date' => now()->addDay()->toDateString(), 'check_out_date' => now()->addDays(3)->toDateString(), 'adults_count' => 2, ]) ->assertCreated(); });
```

Writing unit tests

Unit tests focus on a single class or method without HTTP:

```
it('blocks dates in the availability ledger', function () { $unit = Unit::factory()->create(); UnitAvailabilityService::blockDates($unit, ['2026-07-01', '2026-07-02']); expect($unit->availabilities()->where('status', 'blocked')->count())->toBe(2); });
```

Coverage

As of the latest project progress report, the suite has 297 tests and 1111 assertions. Aim to maintain or improve this coverage when adding features.

Code Style

Turista follows Laravel conventions and uses Laravel Pint for code-style enforcement.

Laravel Pint

Pint is included as a dev dependency. To check style without making changes:

```
vendor/bin/pint --test
```

To apply fixes:

```
vendor/bin/pint
```

CI / pre-commit

Run Pint before committing:

```
vendor/bin/pint --test
```

If it reports issues, run `vendor/bin/pint` and review the changes.

Conventions

Naming

- Controllers: `PascalCase`, singular where possible (`BuildingController`).
- Models: `PascalCase`, singular (`Building`, `Unit`).
- Form requests: `PascalCase` + `Request` suffix (`BuildingRequest`).
- Resources: `PascalCase` + `Resource` suffix (`BuildingResource`).
- Services: `PascalCase` + `Service` suffix (`ReservationService`).
- Policies: `PascalCase` + `Policy` suffix (`BuildingPolicy`).
- Database tables: plural, snake_case (`buildings`, `unit_availabilities`).
- Columns: snake_case.

Controllers

Keep controllers thin:

- Validate input via Form Requests.
- Authorize via policies (`$this->authorize(...)`).
- Delegate business logic to services.
- Return resources or responses.

Services

Services live in `app/Services/` and are exposed through facades in `app/Facades/`. Public methods should have clear, single responsibilities.

Models

- Define relationships explicitly.
- Use casts for dates, enums, and JSON fields.
- Avoid business logic in models; use services.

Mass assignment

`Model::unguard()` is enabled globally by project decision. This means every model accepts any database column during mass assignment. Be extremely careful to whitelist input in controllers and Form Requests.

See `security/overview.md` for the rationale and risks.

Common Development Tasks

This document collects frequently used commands and workflows.

Artisan commands

Create a super admin

```
php artisan admin:create-super admin@example.com
```

You will be prompted for a password.

Download location data

```
php artisan locations:download
```

Downloads and filters location data into `database/data/locations.json`.

Release expired pending reservations

```
php artisan pending-reservations:release-expired
```

Finds expired pending reservations and releases held availability. This also runs every minute via the scheduler.

Send due notifications

```
php artisan notifications:send-due
```

Dispatches reminder notifications that are due. Runs every minute via the scheduler.

Database

Fresh migrate and seed

```
php artisan migrate:fresh --seed
```

Run a specific seeder

```
php artisan db:seed --class=LocationSeeder
```

Reset test database

Tests use an in-memory SQLite database, so no manual reset is needed.

Caching

Clear all caches

```
php artisan optimize:clear
```

> In some local environments this may report a missing MySQL `cache` table. This is an environment quirk and does not affect the test suite.

Cache routes and config for production-like testing

```
php artisan config:cache php artisan route:cache php artisan optimize
```

Queue worker

For local development, the queue worker is started by:

```
composer run dev
```

To run it manually:

```
php artisan queue:listen --tries=1
```

Troubleshooting

Tests fail with database errors

Ensure `phpunit.xml` uses SQLite in-memory and that the SQLite extension is enabled.

`npm run build` fails

Make sure `resources/js/app.js` and `resources/css/app.css` exist. They are minimal but required for Vite.

Media uploads return 404

Ensure the storage link exists:

```
php artisan storage:link
```

OTP not sent

Verify CoreVerde configuration in `.env` and that `COREVERDE_ENABLED` is `true`.

Deployment

This checklist covers deploying Turista to a production environment.

Pre-deployment

- [ ] Set `APP_ENV=production`.
- [ ] Set `APP_DEBUG=false`.
- [ ] Set a strong `APP_KEY` (`php artisan key:generate`).
- [ ] Configure MySQL credentials.
- [ ] Set `REQUEST_DOCS_ENABLED=false`.
- [ ] Restrict `CORS_ALLOWED_ORIGINS` to known frontend domains.
- [ ] Configure Sanctum stateful domains if using cookie-based SPA auth.
- [ ] Configure mail/SMS/WhatsApp providers (CoreVerde).
- [ ] Set up queue connection (database or Redis).
- [ ] Set up scheduler cron entry.

Build steps

1. **Install dependencies**

`bash composer install --no-dev --optimize-autoloader npm ci npm run build`

1. **Run migrations**

`bash php artisan migrate --force`

1. **Cache Laravel optimizations**

`bash php artisan config:cache php artisan route:cache php artisan view:cache php artisan optimize`

1. **Create storage link**

`bash php artisan storage:link`

1. **Set directory permissions**

Ensure `storage/` and `bootstrap/cache/` are writable by the web server.

Scheduler and queues

Add the scheduler cron entry:

```
* * * * * cd /path/to/turista && php artisan schedule:run >> /dev/null 2>&1
```

Run a queue worker using Supervisor or systemd:

```
php artisan queue:work --sleep=3 --tries=3 --max-time=3600
```

Web server

Point the document root to `public/`. Use HTTPS in production.

Post-deployment verification

- [ ] Health check returns `200 OK` on `/`.
- [ ] Login endpoint returns a token for valid credentials.
- [ ] Migrations completed without errors.

- [ ] Queue worker is processing jobs.
- [ ] Scheduler is running every minute.
- [ ] Logs are being written to `storage/logs/`.

Rollback

If a deployment fails:

1. Restore the previous code version.
2. Run `php artisan migrate:rollback` if migrations were applied.
3. Clear caches: `php artisan optimize:clear`.
4. Re-run `php artisan optimize`.

Environment Variables

This reference describes the key environment variables used by Turista.

Application

Variable	Default	Description
`APP_NAME`	Laravel	Application name.
`APP_ENV`	local	Environment: `local`, `testing`, `production`.
`APP_DEBUG`	false	Enables debug responses. Must be `false` in production.
`APP_KEY`		Encryption key. Generate with `php artisan key:generate`.
`APP_URL`	http://localhost	Public URL of the application.

`APP_TIMEZONE`	UTC	Application timezone.
`APP_LOCALE`	en	Application locale.

Database

Variable	D	Description
`DB_CONNECTION`	mysql	Database driver. Tests use SQLite.
`DB_HOST`	127.0.0.1	Database host.
`DB_PORT`	3306	Database port.
`DB_DATABASE`	tuna	Database name.
`DB_USERNAME`	root	Database user.
`DB_PASSWORD`		Database password.

Sanctum

Variable	D	Description
`SANCTUM_TOKEN_EXPIRATION`	1008	Token lifetime in minutes (1 week).
`SANCTUM_STATEFUL_DOMAINS`		Comma-separated list of stateful domains.

Mail

Variable	Description

`MAIL_MAILER`	Mail driver (`smtp`, `log`, etc.).
`MAIL_HOST`	SMTP host.
`MAIL_PORT`	SMTP port.
`MAIL_USERNAME`	SMTP username.
`MAIL_PASSWORD`	SMTP password.
`MAIL_ENCRYPTION`	TLS/SSL.
`MAIL_FROM_ADDRESS`	Default from address.
`MAIL_FROM_NAME`	Default from name.

CoreVerde WhatsApp

Variable	Description
`COREVERDE_BASE_URL`	CoreVerde API base URL.
`COREVERDE_API_TOKEN`	API token.
`COREVERDE_DEVICE_ID`	Device ID.
`COREVERDE_ENABLED`	Set to `true` to enable WhatsApp sending.

Request Docs

Variable	Default	Description

`REQUEST_DOCS_ENABLED`	false	Enables `/request-docs` UI. Disable in production.

CORS

Variable	Description
`CORS_ALLOWED_ORIGINS`	Comma-separated allowed origins.

Locations

Variable	Default	Description
`SEED_COUNTRIES`	Libya	Comma-separated countries to seed.

Queue

Variable	Default	Description
`QUEUE_CONNECTION`	database	Queue driver.

Cache

Variable	Default	Description
`CACHE_STORE`	database	Cache driver.

Logging

Variable	Default	Description
`LOG_CHANNEL`	stack	Log channel.
`LOG_LEVEL`	debug	Minimum log level.

Scheduled Tasks

Turista relies on Laravel's task scheduler for background jobs. A cron entry must call `schedule:run` every minute.

Cron entry

```
* * * * * cd /path/to/turista && php artisan schedule:run >> /dev/null 2>&1
```

Registered commands

Commands are defined in `routes/console.php`.

Release expired pending reservations

```
Schedule::command('pending-reservations:release-expired')->everyMinute();
```

Finds pending reservations whose `expires_at` has passed and releases their held availability so the units can be booked again.

Send due notifications

```
Schedule::command('notifications:send-due')->everyMinute();
```

Claims `ScheduledNotification` rows whose `send_at` has passed and dispatches `DispatchScheduledNotification` jobs to the queue.

Queue workers

Scheduled commands dispatch jobs to the queue. Ensure a queue worker is running:

```
php artisan queue:work --sleep=3 --tries=3
```

For production, use Supervisor or systemd to keep the worker alive.

Monitoring

- Check `storage/logs/laravel.log` for scheduler or worker errors.
- Verify the cron daemon is running.
- Verify the queue worker process is running.
- Use `php artisan queue:monitor` or a monitoring service to alert on queue length.

Monitoring

This document covers observability points for operating Turista in production.

Logs

Laravel logs to `storage/logs/laravel.log` by default. Tail logs in real time:

```
tail -f storage/logs/laravel.log
```

For production, forward logs to a centralized system such as ELK, Datadog, or CloudWatch.

Audit log

OwenIt Auditing writes model changes to the `audits` table. You can query it to trace:

- Who created or updated a record.
- What values changed.
- When the change occurred.

Example:

```
SELECT * FROM audits WHERE auditable_type =
'App\\Models\\Reservation' AND auditable_id = 123 ORDER BY
created_at DESC;
```

Note: password hashes are excluded from audit logs via `User::$auditExclude`.

Queue monitoring

Monitor queue health by:

- Checking worker processes are running.
- Watching queue table or Redis list length.
- Setting up alerts for failed jobs in `failed_jobs`.

Health check

`routes/web.php` exposes a simple health-check endpoint at `/`:

```
GET /
```

A `200 OK` response indicates the application is reachable.

Scheduler monitoring

Ensure the cron entry is active:

```
crontab -l
```

Verify scheduler activity in logs.

Error tracking

Consider integrating an error-tracking service such as Sentry, Bugsnag, or Flare to capture production exceptions.

Performance

- Keep `config`, `route`, and `view` caches enabled in production.
- Monitor slow queries on the `unit_availabilities` and `reservations` tables.
- Review the performance indexes added by recent migrations.

Security Overview

Turista's security model combines Laravel's built-in protections, Sanctum tokens, Spatie roles/permissions, policy-based authorization, and additional hardening.

Authentication

- API clients authenticate with Sanctum bearer tokens.
- Tokens expire after one week by default.
- Login requires a verified account.
- OTP verification is required after registration.

Authorization

- Routes are protected by role middleware (`super_admin`, `admin`, `owner`, `employee`, `customer`).
- Policies enforce ownership and state checks.
- `super_admin` bypasses all policies via `Gate::before`.
- Owner actions require the owner to be approved (`status = active`).
- Employee actions require the employee to be `active` and scoped to their owner/building.

Mass assignment

`Model::unguard()` is enabled globally in `AppServiceProvider`. This is an intentional project decision that removes Laravel's default mass-assignment protection. All input must be explicitly whitelisted in Form Requests and controllers.

Removing `Model::unguard()` requires adding `$fillable` or `$guarded` to every model first.

OTP handling

OTP codes are returned in API responses only in `local` and `testing` environments. In production, OTPs are sent via the configured channel (CoreVerde WhatsApp).

File uploads

Photo uploads are validated by custom rules (`PhotoFileRules`). Allowed types typically include JPG, JPEG, PNG, and WebP with size and dimension limits.

Output encoding

API resources return user-supplied strings as-is. Clients must HTML-escape all strings before inserting them into the DOM to prevent stored XSS.

Security headers

`SecurityHeadersMiddleware` is applied globally and sets:

- `X-Frame-Options: DENY`
- `X-Content-Type-Options: nosniff`
- `Referrer-Policy: strict-origin-when-cross-origin`
- `Strict-Transport-Security`
- `Content-Security-Policy`

Rate limiting

Public auth endpoints are rate-limited. Additional rate limiting can be configured per route.

Dependency updates

Run `composer audit` regularly and update dependencies with known vulnerabilities.

Reporting security issues

Document any new vulnerabilities in the security audit file and prioritize fixes before the next release.

Security Audit Report Summary

This document summarizes the findings from `docs/security-audit-2026-06-22.md`.

Audit scope

- **Date:** 2026-06-22
- **Framework:** Laravel 13.8
- **Methodology:** Read-only source review + targeted runtime checks
- **Scope:** `app/`, `routes/`, `config/`, `database/`, `resources/views/`

Overall posture

The application has a **moderate** defensive posture with proper use of Form Requests, Eloquent parameter binding, role-based middleware, and policies on most resources.

Critical finding

Global `Model::unguard()`

- **Location:** `app/Providers/AppServiceProvider.php`
- **Impact:** Disables Laravel's mass-assignment protection globally.
- **Status:** Intentionally retained per project decision. Removing it requires adding `$fillable`/`$guarded` to every model.

High findings

Issue	Status
Default super-admin password in seeder	Addressed — no default admin created by `DatabaseSeeder`; use `admin:create-super` command.
IDOR in bulk unit creation	Fixed — ownership verified.
IDOR in unit listing	Fixed — building authorization enforced.
Cross-owner `unit_id` changes in reservations	Fixed — owner boundary checks added.
Empty collection returns all availabilities	Fixed — `whereIn` applied even for empty sets.

Medium findings

Issue	Status
Dependency CVEs in `guzzlehttp/guzzle` and `guzzlehttp/psr7`	Monitor and update via `composer audit`.
CORS defaults	Addressed — `.env.example` restricts origins; publish `config/cors.php` for production.
Missing security headers	Addressed — `SecurityHeadersMiddleware` added globally.
Debug mode and request docs in local config	Addressed — `.env.example` sets `APP_DEBUG=false` and `REQUEST_DOCS_ENABLED=false`.

Sanctum token expiration	Addressed — token TTL set to 1 week in `config/sanctum.php`.
OTP returned in responses in local/testing	Intentional for development; not returned in production.
Login enumeration	Reviewed; consider generic failure messages in future hardening.
Stored content output encoding	Documented — clients must HTML-escape API strings.

Low findings

- Generated `api.json` / `routes.json` files should be added to `.gitignore`.
- Password-reset email notification implemented.
- Password hashes excluded from audit logs via `User::$auditExclude`.
- Exception messages should be reviewed for information disclosure.
- Upload validation should be standardized across endpoints.
- Employee creation flow should include verification.

Positive findings

- No classic SQL injection (Eloquent/Query Builder with parameter binding).
- No server-side Blade XSS (`resources/views/` contains only the password-reset email view).
- Rate limiting on public auth routes.
- Role/permission middleware on admin, owner, employee, and customer routes.
- Policy usage on most resources.
- `.env` is gitignored.

Current status

As of the latest project progress report, all tests pass and the fixable audit items have been implemented. The remaining accepted risk is the global `Model::unguard()` decision.

Pre-Launch Security Checklist

Use this checklist before deploying Turista to production or making it publicly available.

Environment

- [ ] `APP_ENV=production`
- [ ] `APP_DEBUG=false`
- [ ] `APP_KEY` is strong and unique
- [ ] `.env` file is not readable by the web server (outside document root or denied by server config)

Authentication & authorization

- [ ] `SANCTUM_TOKEN_EXPIRATION` is set to an appropriate value (default 1 week)
- [ ] `SANCTUM_STATEFUL_DOMAINS` is configured if using cookie-based SPA auth
- [ ] No default users seeded in production
- [ ] First super admin created via `php artisan admin:create-super`

API docs & debug exposure

- [ ] `REQUEST_DOCS_ENABLED=false`
- [ ] `NotFoundWhenProduction` middleware enabled in `config/request-docs.php`
- [ ] Telescope, Horizon, or other dev tools are not exposed

CORS & headers

- [ ] `CORS_ALLOWED_ORIGINS` restricted to known frontend domains
- [ ] `config/cors.php` published and configured

- [ ] `SecurityHeadersMiddleware` active and sets CSP, HSTS, X-Frame-Options, etc.

Dependencies

- [ ] `composer audit` reports no high/critical vulnerabilities
- [ ] `npm audit` reports no high/critical vulnerabilities
- [ ] Production dependencies installed with `composer install --no-dev`

Database

- [ ] MySQL user has minimal required privileges
- [ ] Backups configured
- [ ] Migrations run successfully

File uploads

- [ ] Upload size limits configured in web server and PHP
- [ ] MIME type validation active
- [ ] Image dimension limits configured
- [ ] Uploaded files are not executable

Logging & monitoring

- [ ] Production logs are forwarded to a centralized system
- [ ] Audit logs (`audits` table) monitored
- [ ] Failed queue jobs reviewed
- [ ] Error tracking service integrated (recommended)

Scheduler & queues

- [ ] Cron entry for `schedule:run` active
- [ ] Queue worker running under Supervisor/systemd

- [ ] `pending-reservations:release-expired` and `notifications:send-due` execute on schedule

Communication

- [ ] Mail provider configured and tested
- [ ] CoreVerde WhatsApp credentials configured and tested
- [ ] OTP not returned in API responses

Post-launch

- [ ] Health check endpoint returns `200 OK`
- [ ] Login endpoint returns tokens for valid credentials
- [ ] A sample reservation flow completes end-to-end
- [ ] Security headers verified with an external scanner