Testing Report

Test Strategy, Execution, Coverage, and Reliability

This document summarizes the test strategy, execution approach, coverage workflow, and key outcomes for the project. It also provides concrete commands to reproduce results and recommendations to keep the test suite reliable as the codebase scales.

Overview

- **Goal:** Keep core business logic deterministic and fast via unit/widget tests, with a thin layer of end-to-end (E2E) coverage for critical flows.
- **Outcome:** All unit/widget tests pass in the standard run; E2E tests are gated or skipped unless backend test doubles are available.
- **Artifacts:** Coverage report at `coverage/lcov.info` when generated; stable test harnesses and mocks are available under `test/`.

Current Situation (for next interns)

- **App status:** In progress; core flows exist but backend integration is not yet stable for automated tests.
- **Test status:** Unit/widget tests are green; integration/E2E tests are skipped unless a mock backend is provided.
- **Known constraints:**
    - A verification-code widget overflows under the widget test harness (viewport constraint).
    - Image/Lottie assets require the provided test asset helper.
- **Practical takeaway:** You can confidently refactor core models/managers and most widgets with test coverage. Plan extra work for backend-dependent flows.

Test Strategy

Pyramid

1. **Unit tests**
    - Validation rules
    - Model serialization (`fromJson` / `toJson`)
    - Manager/provider logic that is pure or easily isolated
2. **Widget tests**
    - Layout and interactions
    - Provider-backed state updates

    - Scoped via `MultiProvider` in test scaffolds
3. **E2E / Integration tests**
    - Selected critical flows (e.g., reporting journeys)
    - Guarded behind a flag or temporarily skipped when external APIs are unstable

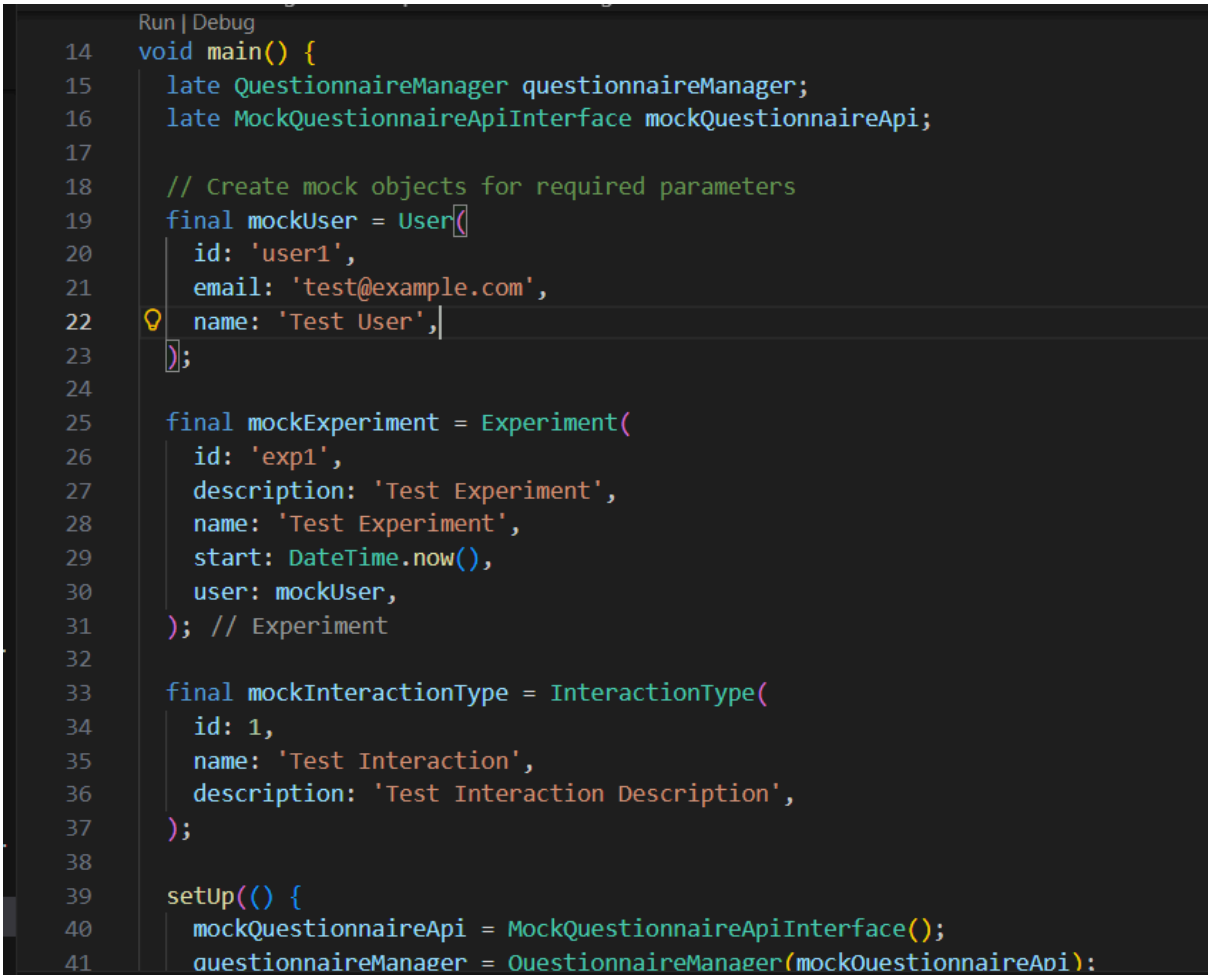
---

Tooling & Harness

- **Frameworks:** `flutter_test`, Mockito
- **State:** Provider (`MultiProvider`) in test scaffolds
- **Asset mocking:** `registerTestAssets()` for images/Lottie in widget tests

Example files

- Unit/business logic: `test/business/managers/questionnaire_manager_test.dart`
    - Link: test/business/managers/questionnaire_manager_test.dart
- Asset utilities: `test/widgets/test_asset_utils.dart`
    - Link: test/widgets/test_asset_utils.dart
- E2E entry: `integration_test/app_test.dart`
    - Link: integration_test/app_test.dart



```
Run | Debug
void main() {
  late QuestionnaireManager questionnaireManager;
  late MockQuestionnaireApiInterface mockQuestionnaireApi;

  // Create mock objects for required parameters
  final mockUser = User(
    id: 'user1',
    email: 'test@example.com',
    name: 'Test User',
  );

  final mockExperiment = Experiment(
    id: 'exp1',
    description: 'Test Experiment',
    name: 'Test Experiment',
    start: DateTime.now(),
    user: mockUser,
  ); // Experiment

  final mockInteractionType = InteractionType(
    id: 1,
    name: 'Test Interaction',
    description: 'Test Interaction Description',
  );

  setUp(() {
    mockQuestionnaireApi = MockQuestionnaireApiInterface();
    questionnaireManager = QuestionnaireManager(mockQuestionnaireApi);
```

Execution Results (Summary)

- **Standard run:** All unit and widget tests are passing.
- **Integration/E2E:** Tests depending on live backend responses are skipped by default to keep the suite deterministic. The underlying domain logic is still covered by unit/widget tests.

What's Been Done (testing so far)

- Fixed and aligned tests for:
  - Model `toJson` / `fromJson`
  - Validation (including numeric-only rules)
  - UI components (animals grid/tile/table)
- Added asset mocking for images/Lottie to keep widget tests deterministic.
- Stabilized provider/state interactions and simplified brittle widget expectations.

Coverage Workflow

Generate coverage

Produces `coverage/lcov.info`:

flutter test --coverage

Notes

- Coverage focuses on business logic and core widgets.
- E2E coverage is deliberately light to avoid coupling to external systems.


docs \ testing
5.4-Testing-and-Evaluation.md U
integration_test
ios
lib 28%
config 0%
constants 80%
data_managers 0%
exceptions 33%
interfaces 91%
managers 64%
api_managers 61%
belonging_damage_report_flow
filtering_system 87%
map 13%
other 88%
permission 48%
state_managers 87%
waarneming_flow 72%
readmea_manager_folder.txt
models 54%
animal_waarneming_models 88%
api_models 42%
beta_models 71%
enums 19%
factories 50%
reports
ui_models 20%
readme_model_folder.txt
providers 22%
screens 7%
utils 56%
widgets 29%
animals 66%
belonging
location 3%
login 0%
map 0%
overlay 41%
overzicht 75%
questionnaire 26%
shared_ui_widgets 72%
toasts 0%
readme_widget_folder.txt

Coverage Analysis

High coverage (business logic, models)

- Model (de)serialization and validation in `lib/models/beta_models/`, including:
    - lib/models/beta_models/belonging_damage_report_model.dart
- Filtering/transformation utilities and small managers where logic is deterministic.

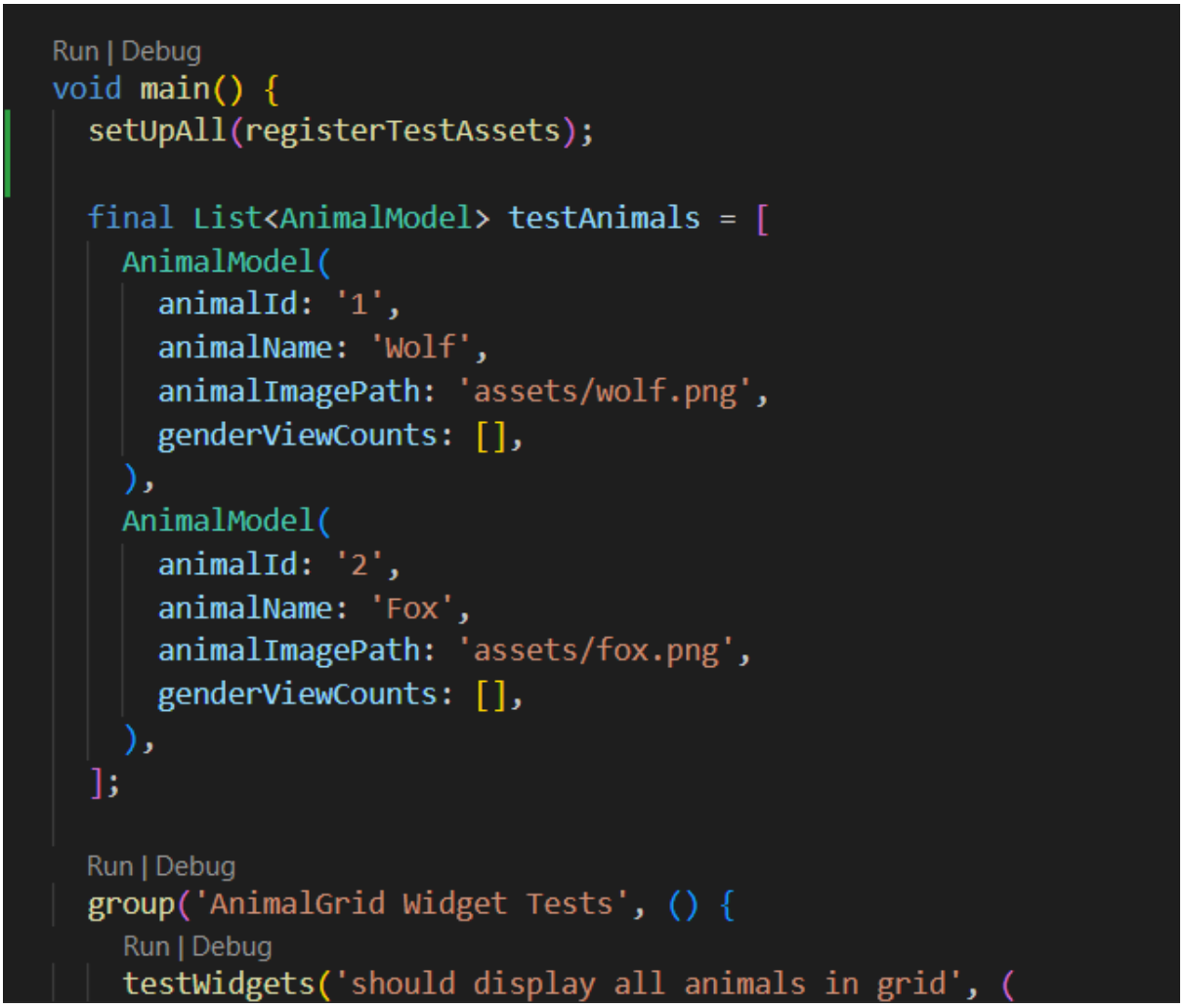
```dart
Run | Debug
void main() {
  Run | Debug
  group('BelongingDamageReport Model', () {
    Run | Debug
    test('should have correct properties', () {
      // Arrange
      final possesion = Possesion(
        possesionID: 'belonging-123',
        possesionName: 'Test Belonging',
        category: 'Test Category',
      );

      final report = BelongingDamageReport(
        possesionDamageReportID: 'report-123',
        possesion: possesion,
        impactedAreaType: 'square-meters',
        impactedArea: 100.0,
        currentImpactDamages: 500.0,
        estimatedTotalDamages: 1000.0,
        description: 'Test damage',
        systemDateTime: DateTime(2023, 5, 15),
      ); // BelongingDamageReport

```

Moderate coverage (state + UI composition)

- Provider-backed state interactions for key flows, e.g.:
    - lib/providers/map_provider.dart
- Core widgets and interactions, e.g.:
    - lib/widgets/animals/animal_grid.dart



```
Run | Debug
void main() {
  setUpAll(registerTestAssets);

  final List<AnimalModel> testAnimals = [
    AnimalModel(
      animalId: '1',
      animalName: 'Wolf',
      animalImagePath: 'assets/wolf.png',
      genderViewCounts: [],
    ),
    AnimalModel(
      animalId: '2',
      animalName: 'Fox',
      animalImagePath: 'assets/fox.png',
      genderViewCounts: [],
    ),
  ];

  Run | Debug
  group('AnimalGrid Widget Tests', () {
    Run | Debug
    testWidgets('should display all animals in grid', (
```

Low / excluded (integration surfaces)

- Live API adapters and network-bound flows, e.g.:
  - lib/managers/api_managers/interaction_manager.dart
  - Covered indirectly via unit tests on their consumers; full-path coverage deferred to integration tests with mocks/fakes.
- Platform integrations (permissions, geolocation, notifications, map rendering specifics)
  - Validated via seam-level tests and manual verification rather than headless automation.
- Known harness-specific skip:
  - The verification-code subview of the login flow is intentionally skipped under the default widget harness due to a layout overflow specific to test constraints; the parent login logic remains covered.

Examples

What's covered

- Serialization/validation invariants for reporting models (inputs, numeric ranges, required fields).
- Map provider behavior for updating address on position change.
- Animals UI components: render, selection/tap callbacks, and table presence checks.
- Questionnaire open-response validation (numeric-only patterns vs. minimum length).

What's not covered (or intentionally minimal)

- Real network calls and endpoint-by-endpoint API error surfaces.
- Platform channel behaviors (sensors, notifications) and device-specific rendering issues.
- Golden image diffs for complex layouts (kept minimal to reduce flakiness).

Rationale

- Prioritize fast, deterministic feedback on core domain logic and essential UI paths.
- Avoid brittle tests against external systems or harness-specific layout quirks.
- Keep E2E behind a toggle until a stable mock backend is provided.

Improving Coverage Next

- Add a small fake API client for integration tests to exercise:
    - lib/managers/api_managers/response_manager.dart
    - Related flows without live endpoints
- Introduce a shared widget test harness with standardized viewport sizes for complex screens (reduces false overflows).
- Add targeted golden tests for stable components (buttons, tiles), avoiding dynamic maps/animations.
- Track coverage deltas per PR and gradually raise thresholds for business-logic packages.

Guidance for Next Interns

How to run

- Unit/widget tests:

flutter test

- With coverage:

flutter test --coverage

If you need E2E

- Prefer a mock/fake API client; avoid hitting live endpoints in CI.
- Add an opt-in flag before enabling integration tests, for example:
    - `RUN_E2E=true`

Reliability & Flakiness Management

- Replace external dependencies with mocks/fakes (network, permissions, location, storage).
- Mock asset decoding to prevent file/manifest flakiness.
- Contain viewport/layout issues by skipping specific widget tests known to fail only in the test harness (parent flows remain validated).
- Guard E2E behind an env flag or `skip` annotations.

What's in scope

- Validation and serialization guarantees
- Manager/provider state transitions
- Primary widget rendering and user interactions

Out of scope (for routine CI runs)

- Live backend availability and contract testing
- Device-specific rendering nuances not reproducible in headless test environments

---

Recommendations

- Gate E2E runs with an env flag (e.g., `RUN_E2E=true`) so CI remains green by default.
- Add a fake API adapter or mock server for integration tests to remove reliance on live endpoints.
- Provide a standard test scaffold with known viewport sizes for complex layout screens.
- Track and enforce a pragmatic coverage threshold (e.g., 70–80% for business-logic packages) and report deltas on PRs.

Quick Commands

- Run all tests:

flutter test

- Run with coverage:

flutter test --coverage