Final Feature List for EffortlessQA (MVP)

1. User Management & Authentication

* User Registration/Login:
  + Signup and login via email/password or Google OAuth (optional GitHub OAuth) for quick and secure access.
  + Basic user profile management (name, email, password reset).
* Role-Based Access:
  + Simple roles: Admin (full access, manage users/projects) and Tester (execute tests, view results).
  + Admins can invite users to projects and assign roles.
* Multi-Project Support:
  + Create, edit, and delete projects as containers for test activities.
  + Fields: Project Name, Description.
  + Users can switch between multiple projects within their account.

2. Requirement Management (Lightweight)

* Add/Edit/Update Requirements:
  + Create and manage basic requirements with fields: Title, Description.
  + Link requirements to test cases for traceability.
* Tagging:
  + Use tags for categorizing and filtering requirements.

3. Test Case Management

* Create/Edit/Delete Test Cases:
  + Fields: Title, Description, Steps (plain text), Expected Results, Priority (High/Medium/Low), Tags.
  + Organize test cases into folders or test suites (single-level hierarchy).
* Copy and Reuse Test Cases:
  + Duplicate test cases within or across projects for efficiency.
* Import/Export:
  + Import test cases from CSV/Excel for quick setup.
  + Export test cases to CSV for sharing or reporting.

4. Test Suite & Test Run Management

* Test Suites:
  + Group test cases into suites (e.g., by module or feature).
  + Create, edit, and delete suites within a project.
* Test Runs:
  + Create test runs by selecting test cases from suites or projects.
  + Fields: Run Name, Description, Assigned Tester (optional).
  + Track execution progress (% complete) in real-time.

5. Test Execution

* Execute Test Cases:
  + View test case details (title, steps, expected results) during execution.
  + Mark status: Pass, Fail, Blocked, Skipped/Not Run.
  + Add notes/comments (plain text) and attachments (e.g., screenshots) for failed/blocked cases.
* Bulk Updates:
  + Allow bulk updates for test case statuses to streamline execution.
* Progress Tracking:
  + Display a real-time progress tracker (e.g., list of test cases with status) in an intuitive UI.

6. Defect Tracking (Minimal)

* Basic Defect Logging:
  + Log defects during test runs with fields: Title, Description, Severity (High/Medium/Low), Screenshot/Attachment.
  + Link defects to test cases or runs for traceability.
  + Track defect statuses: Open, In Progress, Resolved, Closed.
* External Integration Placeholder:
  + Allow manual entry of external defect IDs (e.g., Jira or GitHub Issues ticket numbers) for linking to third-party tools.

7. Reporting & Dashboard

* Real-Time Dashboard:
  + Display project overview: Active test runs, pass/fail rates, defect counts, recent activity.
  + Use basic visuals (e.g., pie charts, bar graphs) via Chart.js for test status and progress.
* Reports:
  + Generate test run summaries (test coverage, pass/fail rates, number of tests completed, status breakdown).
  + Export reports to CSV and PDF for external use.
* Filterable Views:
  + Filter test cases, runs, requirements, and defects by status, priority, tags, or project.

8. Search & Filters

* Global Search:
  + Search test cases, suites, runs, requirements, and defects by title or tags.
* Filters:
  + Filter entities by status, priority, tags, or modules for quick navigation.

9. Simple User Interface & Usability

* Clean, Responsive UI:
  + Minimalistic design with clear navigation (Projects → Suites → Test Cases → Runs → Requirements).
  + Responsive for desktop and basic mobile compatibility (using React.js or Vue.js).
* Drag-and-Drop:
  + Organize test cases into suites using drag-and-drop functionality.
* Quick Onboarding:
  + Guided setup wizard for creating the first project, test case, and run.
  + Tooltips and a basic help section/documentation for key actions.
* Intuitive Execution Interface:
  + Streamlined UI for executing tests with a focus on usability.

10. Audit Trail

* Activity Log:
  + Log key actions (e.g., test case created/updated, test run completed, defect logged) with timestamps and user info.
  + Viewable per project for transparency.

11. Authentication & Security

* Secure Access:
  + Implement HTTPS, data encryption at rest, and secure user authentication (email/password + Google/GitHub OAuth).
* Multi-Tenancy:
  + Support data isolation for different customers in a B2B SaaS model.
* Compliance:
  + Ensure basic GDPR/CCPA compliance (e.g., user consent for data storage, data deletion options).

12. Foundation for Future AI Features

* Structured Data Storage:
  + Store test case and requirement metadata (e.g., steps, expected results) in JSON format in PostgreSQL or MongoDB to support future AI processing.
  + Use tags for pattern analysis to enable AI-driven test case generation.
* API-First Design:
  + Provide a basic REST API to submit test results programmatically (e.g., POST /api/projects/{id}/runs/{id}/results).
  + Build a modular backend (e.g., microservices or RESTful APIs) for future AI integrations (e.g., OpenAI API).
* Logging for AI:
  + Capture user interactions (e.g., test case edits, run executions) in logs for training future AI models.

1. Defect

Purpose

The Defect entity is used to log and track issues identified during test execution, ensuring that quality issues in the software being tested are documented, prioritized, and resolved. It supports the "Defect Tracking (Minimal)" feature, enabling testers to record defects, link them to test cases or runs, and track their status (e.g., Open, Resolved). This entity is critical for maintaining traceability and facilitating communication between testers and developers.

Key Fields

* Id (Guid): Unique identifier for the defect.
* Title (string, required, max 200 chars): A brief summary of the defect (e.g., "Login button not working").
* Description (string, optional, max 1000 chars): Detailed explanation of the issue, including steps to reproduce.
* Severity (SeverityLevel enum: High, Medium, Low): Indicates the impact of the defect (e.g., High for critical bugs).
* Status (DefectStatus enum: Open, InProgress, Resolved, Closed): Tracks the defect’s lifecycle.
* Attachments (JsonDocument, optional): JSONB field for storing screenshots or other files (e.g., { "file1": "url", "file2": "url" }).
* ExternalId (string, optional, max 100 chars): Links to external tools like Jira or GitHub (e.g., "JIRA-123").
* AssignedUserId (Guid, optional): The user responsible for resolving the defect.
* ResolutionNotes (string, optional, max 1000 chars): Details on how the defect was resolved.
* TestRunResultId (Guid, optional): Links to the test run result where the defect was found.
* TestCaseId (Guid, optional): Links to the test case associated with the defect.
* TenantId (string, required): Ensures multi-tenant data isolation.
* Inherited from EntityBase: CreatedAt, ModifiedAt, CreatedBy, ModifiedBy for audit tracking.
* Navigation Properties:
  + AssignedUser (User): The user assigned to fix the defect.
  + TestRunResult (TestRunResult): The test execution result linked to the defect.
  + TestCase (TestCase): The test case where the defect was identified.
  + DefectHistories (List<DefectHistory>): Tracks status changes and comments.

Role in EffortlessQA

* Defect Logging: Testers log defects during test execution (e.g., marking a test as "Fail" in a TestRunResult) with details like title, description, severity, and screenshots. For example, a tester might log a defect titled "Login fails with invalid credentials" with a screenshot attached.
* Traceability: Linking defects to TestRunResult or TestCase ensures testers can trace issues back to specific tests, supporting the requirement for "Link defects to test cases or runs for traceability."
* Status Tracking: The Status field allows tracking of defect resolution (e.g., from Open to Resolved), with DefectHistory capturing changes (e.g., "Status changed to Resolved by User X").
* External Integration: The ExternalId field allows manual entry of external ticket IDs (e.g., Jira), fulfilling the "External Integration Placeholder" requirement.
* Reporting: Defects contribute to the real-time dashboard and reports, showing defect counts and severity distribution (e.g., pie charts via Chart.js).
* Multi-Tenancy: TenantId ensures defects are isolated per tenant, critical for the SaaS model.

Example Workflow

* A tester executes a test case in a TestRun and marks it as "Fail" due to a bug.
* They create a Defect with Title = "Checkout button unresponsive," Severity = High, and attach a screenshot via Attachments.
* The defect is linked to the TestRunResult and TestCase for traceability.
* An admin assigns the defect to a developer (AssignedUserId), who updates the Status to Resolved and adds ResolutionNotes.
* DefectHistory logs the status change, and an AuditLog (if [Auditable]) records the update.

2. Requirement

Purpose

The Requirement entity supports lightweight requirement management, allowing users to define and manage functional or non-functional requirements for a project. It enables traceability by linking requirements to test cases, fulfilling the "Requirement Management (Lightweight)" feature. Requirements act as the foundation for testing, ensuring that test cases cover all specified needs.

Key Fields

* Id (Guid): Unique identifier for the requirement.
* Title (string, required, max 200 chars): A concise description of the requirement (e.g., "User must be able to reset password").
* Description (string, optional, max 1000 chars): Detailed explanation of the requirement.
* Tags (string[], optional): Array for categorizing requirements (e.g., ["Security", "UI"]).
* ProjectId (Guid, required): Links to the project containing the requirement.
* TenantId (string, required): Ensures multi-tenant isolation.
* Inherited from EntityBase: CreatedAt, ModifiedAt, CreatedBy, ModifiedBy.
* Navigation Properties:
  + Project (Project): The project the requirement belongs to.
  + RequirementTestCases (List<RequirementTestCase>): Links to associated test cases.

Role in EffortlessQA

* Requirement Definition: Admins or testers create requirements to document what the software must do (e.g., "Support multi-factor authentication"). The Title and Description provide clarity, while Tags enable filtering (e.g., all "Security" requirements).
* Traceability: The RequirementTestCase entity links requirements to test cases, ensuring test coverage. For example, a requirement "Support payment via credit card" is linked to test cases verifying payment functionality.
* Project Organization: Requirements are scoped to a Project, allowing teams to manage requirements per project (e.g., "E-commerce Platform" project).
* Search and Filtering: The Tags field supports the "Tagging" and "Search & Filters" requirements, enabling users to filter requirements by category (e.g., via the global search UI).
* Reporting: Requirements contribute to test coverage reports, showing which requirements are tested (linked to test cases) and their status.
* Multi-Tenancy: TenantId isolates requirements per tenant, ensuring data privacy in the SaaS model.

Example Workflow

* An admin creates a Requirement titled "User profile update" with Tags = ["Profile", "UI"] in a project.
* Testers link it to test cases (e.g., "Verify profile picture upload") via RequirementTestCase.
* The dashboard shows test coverage, indicating that 80% of requirements in the project have associated test cases.
* Testers filter requirements by "UI" tag to focus on UI-related testing.

3. RequirementTestCase

Purpose

The RequirementTestCase entity is a join table that establishes a many-to-many relationship between Requirement and TestCase. It supports the traceability requirement by linking requirements to the test cases that verify them, ensuring that all requirements are covered by tests.

Key Fields

* RequirementId (Guid, required): References the Requirement.
* TestCaseId (Guid, required): References the TestCase.
* Navigation Properties:
  + Requirement (Requirement): The requirement being tested.
  + TestCase (TestCase): The test case verifying the requirement.

Role in EffortlessQA

* Traceability: Enables mapping of requirements to test cases, ensuring that each requirement is validated by one or more tests. For example, a requirement "Support offline mode" might be linked to multiple test cases checking offline functionality.
* Test Coverage: Supports reporting on which requirements are covered by test cases, fulfilling the "Link requirements to test cases for traceability" requirement.
* Flexibility: The many-to-many relationship allows a requirement to be tested by multiple test cases and a test case to cover multiple requirements.
* Multi-Tenancy: Inherits TenantId indirectly through Requirement and TestCase, ensuring data isolation.

Example Workflow

* A tester creates a Requirement for "Secure password storage."
* They create test cases like "Verify password hashing" and "Verify password strength validation."
* Using RequirementTestCase, they link the requirement to both test cases.
* A report shows that the requirement is fully covered, as both test cases passed in a TestRun.

4. TestRun

Purpose

The TestRun entity represents a specific instance of test execution within a project, grouping test cases to be executed together (e.g., for a sprint or release). It supports the "Test Suite & Test Run Management" feature, allowing users to create, manage, and track test runs, including assigning testers and monitoring progress.

Key Fields

* Id (Guid): Unique identifier for the test run.
* Name (string, required, max 100 chars): Descriptive name (e.g., "Sprint 5 Regression").
* Description (string, optional, max 500 chars): Additional details about the test run.
* AssignedTesterId (Guid, optional): The user assigned to execute the test run.
* ProjectId (Guid, required): Links to the project containing the test run.
* TenantId (string, required): Ensures multi-tenant isolation.
* Inherited from EntityBase: CreatedAt, ModifiedAt, CreatedBy, ModifiedBy.
* Navigation Properties:
  + AssignedTester (User): The tester assigned to the run.
  + Project (Project): The project the test run belongs to.
  + TestRunResults (List<TestRunResult>): The results of test cases executed in this run.

Role in EffortlessQA

* Test Execution Planning: Users create a TestRun to execute a set of test cases, either from a TestSuite or directly from the project, fulfilling the "Create test runs by selecting test cases" requirement.
* Assignment: The AssignedTesterId allows admins to assign specific testers, supporting collaborative testing.
* Progress Tracking: The TestRunResults collection tracks execution progress (e.g., % complete), displayed in the real-time dashboard as a progress tracker.
* Reporting: Test run summaries (pass/fail rates, number of tests completed) are generated from TestRunResults, supporting the "Reports" requirement.
* Multi-Tenancy: TenantId ensures test runs are isolated per tenant.

Example Workflow

* An admin creates a TestRun named "Release 1.2 Smoke Test" in a project, selecting test cases from a TestSuite.
* They assign it to a tester (AssignedTesterId).
* The tester executes the test cases, creating TestRunResult entries for each.
* The dashboard shows 75% completion with 80% pass rate, updated in real-time.
* A PDF report is exported, summarizing the test run’s results.

5. TestRunResult

Purpose

The TestRunResult entity records the outcome of executing a specific test case within a TestRun. It supports the "Test Execution" feature, allowing testers to mark test outcomes (Pass, Fail, Blocked, Skipped), add comments, and attach evidence (e.g., screenshots), while tracking progress and linking to defects.

Key Fields

* Id (Guid): Unique identifier for the test run result.
* TestCaseId (Guid, required): The test case being executed.
* TestRunId (Guid, required): The test run containing this result.
* Status (TestExecutionStatus enum: Pass, Fail, Blocked, Skipped): The outcome of the test.
* Comments (string, optional, max 1000 chars): Notes on the test execution (e.g., "Failed due to timeout").
* Attachments (JsonDocument, optional): JSONB field for screenshots or logs.
* TenantId (string, required): Ensures multi-tenant isolation.
* Inherited from EntityBase: CreatedAt, ModifiedAt, CreatedBy, ModifiedBy.
* Navigation Properties:
  + TestCase (TestCase): The test case executed.
  + TestRun (TestRun): The test run this result belongs to.
  + Defects (List<Defect>): Defects logged during this test execution.

Role in EffortlessQA

* Test Execution: Testers mark the Status of each test case during execution, fulfilling the "Execute Test Cases" requirement. For example, a test case might be marked "Fail" with a comment explaining the issue.
* Evidence Collection: Attachments stores screenshots or logs for failed/blocked tests, supporting the "Add notes/comments and attachments" requirement.
* Bulk Updates: The TestRunResultBulkUpdateDto allows testers to update multiple test results at once, streamlining execution (e.g., marking all tests in a suite as "Skipped").
* Progress Tracking: Aggregated Status values drive the real-time progress tracker in the UI, showing pass/fail rates and completion percentage.
* Defect Linking: Defects can be linked to a TestRunResult, ensuring traceability (e.g., a "Fail" result links to a Defect describing the issue).
* Reporting: Contributes to test run summaries and dashboard visuals (e.g., bar graphs of pass/fail rates).
* Multi-Tenancy: TenantId isolates results per tenant.

Example Workflow

* During a TestRun, a tester executes a test case and marks it as "Fail" in the TestRunResult, adding a comment ("Server error 500") and a screenshot via Attachments.
* They create a Defect linked to this TestRunResult.
* The dashboard updates to show 60% pass rate for the TestRun.
* The tester uses bulk update to mark remaining tests as "Skipped" due to a blocking issue.
* A report exported to CSV shows the test case’s failure and linked defect.

6. TestSuite

Purpose

The TestSuite entity organizes test cases into logical groups (e.g., by module or feature) within a project, supporting the "Test Suite & Test Run Management" feature. It provides a single-level hierarchy for structuring test cases, making it easier to manage and execute related tests together.

Key Fields

* Id (Guid): Unique identifier for the test suite.
* Name (string, required, max 100 chars): Descriptive name (e.g., "Authentication Tests").
* Description (string, optional, max 500 chars): Details about the suite’s purpose.
* ProjectId (Guid, required): Links to the project containing the suite.
* TenantId (string, required): Ensures multi-tenant isolation.
* Inherited from EntityBase: CreatedAt, ModifiedAt, CreatedBy, ModifiedBy.
* Navigation Properties:
  + Project (Project): The project the suite belongs to.
  + TestCases (List<TestCase>): The test cases in this suite.

Role in EffortlessQA

* Test Organization: Groups related test cases (e.g., all tests for the "Payment Module" in one suite), fulfilling the "Group test cases into suites" requirement.
* Test Run Creation: Users can select test cases from a TestSuite to create a TestRun, streamlining execution (e.g., running all "Regression Tests" in a suite).
* Drag-and-Drop: Supports the "Drag-and-Drop" requirement, allowing testers to organize test cases into suites via the UI.
* Search and Filtering: Suites can be searched or filtered by name or project, supporting the "Search & Filters" feature.
* Reporting: Contributes to test coverage reports by showing which suites are executed in a TestRun.
* Multi-Tenancy: TenantId isolates suites per tenant.

Example Workflow

* An admin creates a TestSuite named "Login Tests" in a project.
* Testers add test cases like "Verify successful login" and "Verify failed login" to the suite via drag-and-drop in the UI.
* A TestRun is created, pulling all test cases from the "Login Tests" suite.
* The dashboard shows the suite’s test results, with 90% pass rate.
* Testers filter suites by project to focus on specific testing areas.

Interrelationships and Workflow Integration

These entities work together to form a cohesive test management workflow in EffortlessQA:

* Project Setup: An admin creates a Project and defines Requirements to specify what needs to be tested.
* Test Organization: Testers create TestSuites to group related TestCases, linking them to Requirements via RequirementTestCase for traceability.
* Test Execution: A TestRun is created, selecting test cases from a TestSuite or project. Testers execute tests, creating TestRunResults to record outcomes (e.g., Pass, Fail).
* Defect Management: If a test fails, a Defect is logged, linked to the TestRunResult or TestCase. Defects are tracked through their lifecycle, with DefectHistory logging changes.
* Reporting and Dashboard: The dashboard displays real-time metrics (e.g., test run progress, defect counts) based on TestRunResults and Defects. Reports show test coverage by linking Requirements to TestCases.
* Auditing: Changes to [Auditable] entities (e.g., TestCase, Defect) are logged in AuditLog, ensuring transparency.

Example End-to-End Workflow

* Requirement: An admin creates a Requirement titled "Support single sign-on (SSO)" in a project.
* Test Suite: A tester creates a TestSuite named "SSO Tests" and adds test cases like "Verify SSO login with Google."
* RequirementTestCase: The requirement is linked to the test case via RequirementTestCase.
* Test Run: A TestRun named "SSO Feature Test" is created, including the "SSO Tests" suite, assigned to a tester.
* TestRunResult: The tester executes the test case, marking it as "Fail" in a TestRunResult with a comment ("SSO redirects to error page") and a screenshot.
* Defect: A Defect titled "SSO redirect error" is logged, linked to the TestRunResult, with Severity = High.
* Resolution: A developer resolves the defect, updating its Status to Resolved and adding ResolutionNotes. DefectHistory logs the change.
* Reporting: The dashboard shows the test run’s 50% pass rate and one open defect. A report confirms the requirement is partially covered due to the failed test.

Technical Integration

* Multi-Tenancy: Each entity includes TenantId (except RequirementTestCase, which inherits it indirectly), enforced by global query filters in EffortlessQAContext. This ensures data isolation across tenants in the SaaS model.
* Auditing: The [Auditable] attribute on TestCase and Defect (and their key properties) triggers AuditLog entries for create, update, and delete operations, stored in JSONB for AI-driven analysis.
* Minimal API: Endpoints like /api/testruns, /api/defects, and /api/testsuites use DTOs (e.g., TestRunCreateDto, DefectDto) to create and retrieve data, with audit fields set server-side.
* Blazor UI: The responsive UI (using MudBlazor) displays test suites, test runs, and defects, with drag-and-drop for organizing test cases into suites and real-time progress tracking for test runs.
* Extensibility: JSONB fields (Defect.Attachments, TestRunResult.Attachments, TestCase.Steps) support future AI features, such as analyzing test steps or defect screenshots for automated test generation.

Flexibility and Extensibility

* Defect: Attachments and ExternalId allow integration with external tools and AI-driven defect analysis (e.g., image recognition for screenshots).
* Requirement: Tags and JSONB-ready Description support pattern analysis for AI-driven requirement generation.
* RequirementTestCase: Enables flexible test coverage strategies, extensible for weighted traceability.
* TestRun: TestRunResults supports bulk updates and progress tracking, extensible for automated test execution.
* TestRunResult: JSONB Attachments allows AI to analyze test evidence, improving failure analysis.
* TestSuite: Single-level hierarchy is extensible to multi-level suites if needed.