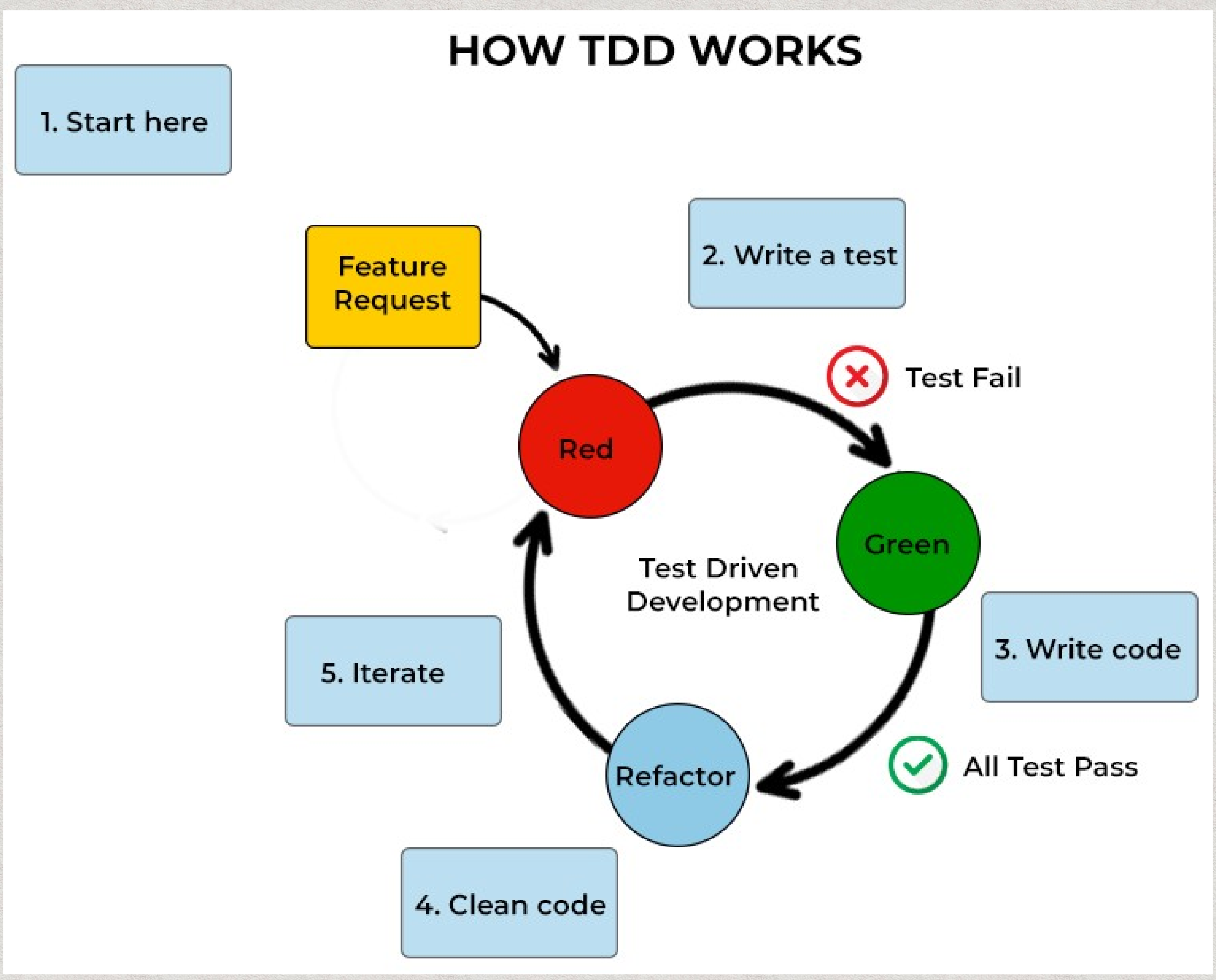
Weather App – TDD.

A simple project in my journey of learning Test-driven development (TDD) and Tests in Flutter.

- Weather
- Bloc State Management.
- Unit Testing.
- Widget Testing.
- Clean Architecture.
- TDD.

Test-driven development (TDD).



Test-driven development (TDD) is a software development process relying on software requirements being converted to test cases before software is fully developed, and tracking all software development by repeatedly testing the software against all test cases. This is as opposed to software being developed first and test cases created later.

Software engineer Kent Beck, who is credited with having developed or "rediscovered" the technique, stated in 2003 that
TDD encourages simple designs and inspires confidence.

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999, but more recently has created more general interest in its own right.

Test-driven development cycle.

1. Add a test.

   The adding of a new feature begins by writing a test that passes if the feature's specifications are met. The developer can discover these specifications by asking about use cases and user stories. A key benefit of test-driven development is that it makes the developer focus on requirements before writing code. This is in contrast with the usual practice, where unit tests are only written after code.

2. Run all tests.

The new test should fail for expected reasons. This shows that new code is actually needed for the desired feature. It validates that the test harness is working correctly. It rules out the possibility that the new test is flawed and will always pass.

3. Write the simplest code that passes the new test.

   Inelegant or hard code is acceptable, as long as it passes the test. The code will be honed anyway in Step 5. No code should be added beyond the tested functionality.

4. All tests should now pass.

   If any fail, the new code must be revised until they pass. This ensures the new code meets the test requirements and does not break existing features.

5. Refactor as needed, using tests after each refactor to ensure that functionality is preserved.

Code is refactored for readability and maintainability. In particular, hard-coded test data should be removed. Running the test suite after each refactor helps ensure that no existing functionality is broken.

Examples of refactoring: 1- Moving code to where it most logically belongs. 2- removing duplicate code. 3- making names self-documenting. 4- splitting methods into smaller pieces. 5- re-arranging inheritance hierarchies.

Other Related Concepts.

1- Unit Testing.

How can you ensure that your app continues to work as you add more features or change existing functionality? By writing tests.

Unit tests are handy for verifying the behavior of a single function, method, or class. The test package provides the core framework for writing unit tests, and the flutter_test package provides additional utilities for testing widgets.

[Continue Reading (https://docs.flutter.dev/cookbook/testing/unit/introduction)](https://docs.flutter.dev/cookbook/testing/unit/introduction)

2- Mock dependencies using Mockito.

Sometimes, unit tests might depend on classes that fetch data from live web services or databases. This is inconvenient for a few reasons:

- Calling live services or databases slows down test execution.
- A passing test might start failing if a web service or database returns unexpected results. This is known as a “flaky test.”
- It is difficult to test all possible success and failure scenarios by using a live web service or database.
- Therefore, rather than relying on a live web service or database, you can “mock” these dependencies.
- Mocks allow emulating a live web service or database and return specific results depending on the situation.

[Continue Reading (https://docs.flutter.dev/cookbook/testing/unit/mocking)](https://docs.flutter.dev/cookbook/testing/unit/mocking)

3- Widget testing.

To test widget classes, you need a few additional tools provided by the flutter_test package, which ships with the Flutter SDK.

The flutter_test package provides the following tools for testing widgets:

1- The WidgetTester allows building and interacting with widgets in a test environment.

2- The testWidgets() function automatically creates a new WidgetTester for each test case, and is used in place of the normal test() function.

3 - The Finder classes allow searching for widgets in the test environment.

4- Widget-specific Matcher constants help verify whether a Finder locates a widget or multiple widgets in the test environment.

Continue Reading (https://docs.flutter.dev/cookbook/testing/widget/introduction)

4- get_it.

This is a simple Service Locator for Dart and Flutter projects with some additional goodies highly inspired by Splat. It can be used instead of InheritedWidget or Provider to access objects e.g. from your UI.

Continue Reading (https://pub.dev/packages/get_it)