Introduction to Automated Tests

Why Automation

/thoughtworks

Why Automation?

Manual testing takes too long

- Agile teams deliver production ready software at the end of each iteration, by having production ready software every day.
  - If regression tests are manual,
    - takes more and more time in testing everyday
    - spend more time trying to reproduce & report simple bugs and less time finding serious system level bugs
  - Only less scenarios can be tested - test data setup for complex scenarios also becomes overwhelming

Why Automation?

Manual processes are error prone

- Scripted tests gets repetitive, boring, overlook even simple bugs
- Skip tests when in tight deadline

Why Automation?

Automation frees people to do their best work

- Write code test first (TDD) helps programmers understand requirements and design code accordingly
- Automatically run all unit tests, functional regression tests = time for exploratory tests
- Automating setup for exploratory tests = more time for exploratory tests
- Automated regression tests - detect changes and provide immediate feedback - projects succeed when people free to do best work

Why Automation?

Automated regression tests provide safety net

- Confidence to make change
    - Feedback time proportional to level of tests we have
- Can make changes lot faster

Why Automation?

Automated tests give feedback early and often

- Running automated tests every time code changes are “checked in” ensures regression bugs are caught quickly - bugs are cheaper to troubleshoot and fix.
- Allows to run regression tests on daily basis.

Why Automation?

Tests and examples that drive coding can do more

- Automated tests become base for very strong regression suite.
- Increased team velocity
- TDD - design code to make tests pass, so testability of application is not an issue
- Provides safety net for constant refactoring

Why Automation?

Tests provide documentation

- Become living documentation of how system actually works.
- System changes - automated tests are not updated, tests fail, we need to fix it make the build process green.
- Allows for team members rotation

Why Automation?

Build once - deploy to many

- Write automation code once and use it to run on many regions

Why Automation?

Return on investment

- Reduced team capacity
- Quicker time to market
- Bugs are easier to fix

Barriers to Automation

- Programmers attitude
- Learning curve
- Initial investment
- Legacy Systems
- Fear

Should we automate everything

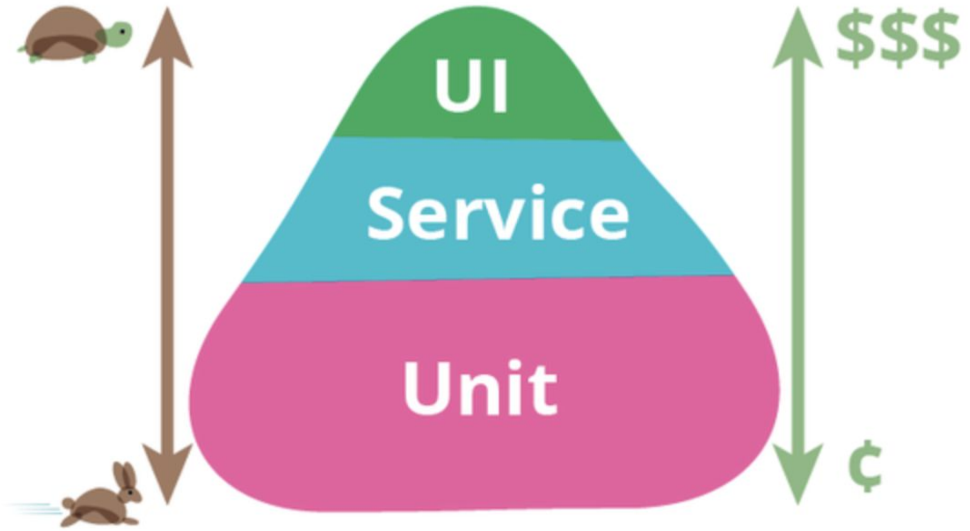
- Usability
- Exploratory
- Tests that will never fail - use risk analysis
- One-Off tests - automation cost vs manual testing time

Introduction to Automated Tests

Test Pyramid

/thoughtworks

Test Pyramid - Best practice for Automated testing

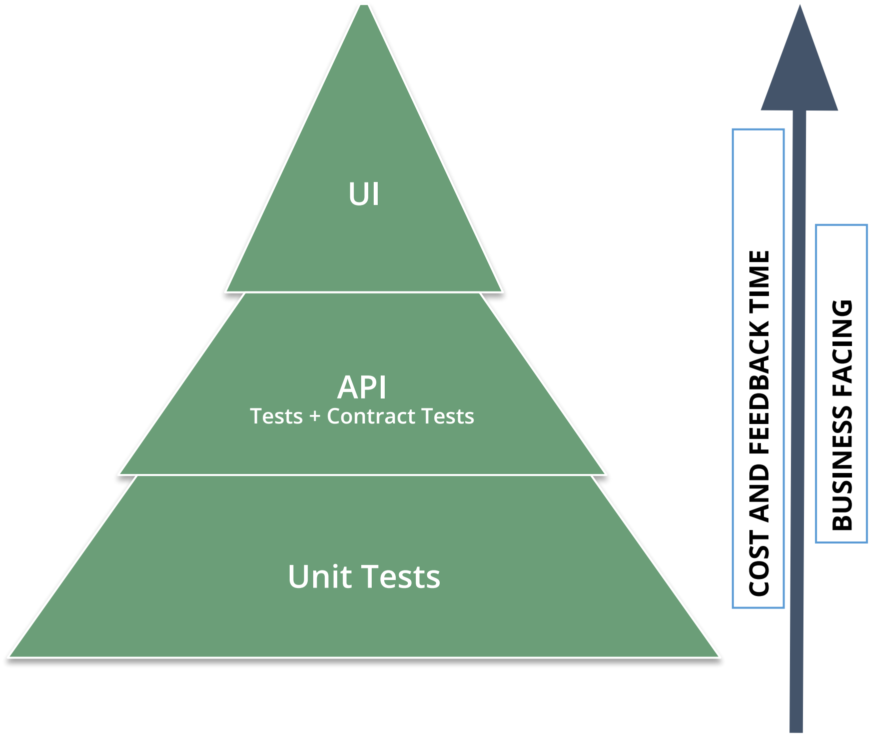


Test Pyramid - Best practice for Automated testing

- End-to-end user journeys
- UI interactions
- Tests from a User point of view
- Expensive to Maintain

- Integration between APIs
- Regressive end-to-end tests
- Quick feedback on interactions among APIs

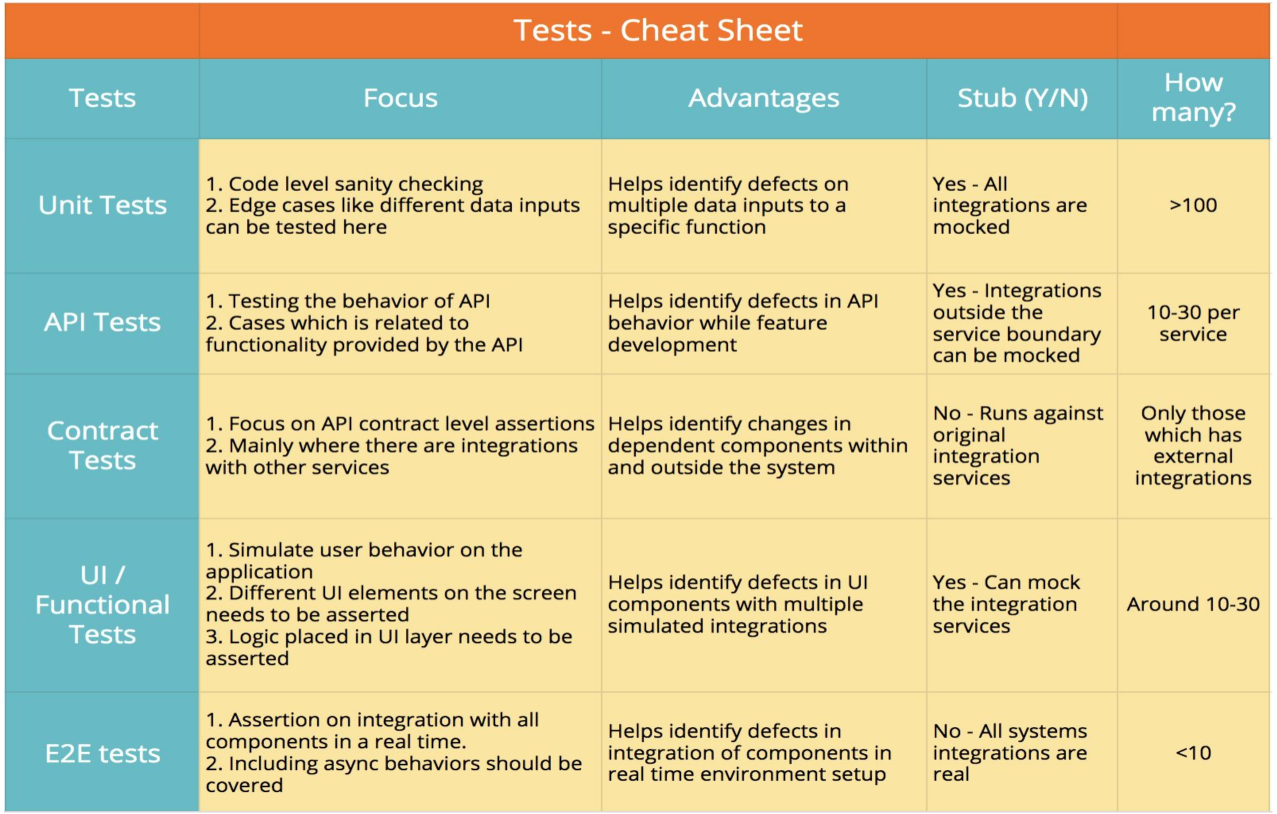
- Unit tests for both functional and UI classes
- Boundary values and edge cases
- Immediate feedback
- Safety net
- Lowest cost of implementation



Automated Tests at different layers

- Unit Tests
- API Tests
- Contract tests
- UI Tests
- End to end Tests

Tests Cheat sheet



Tests - Cheat Sheet

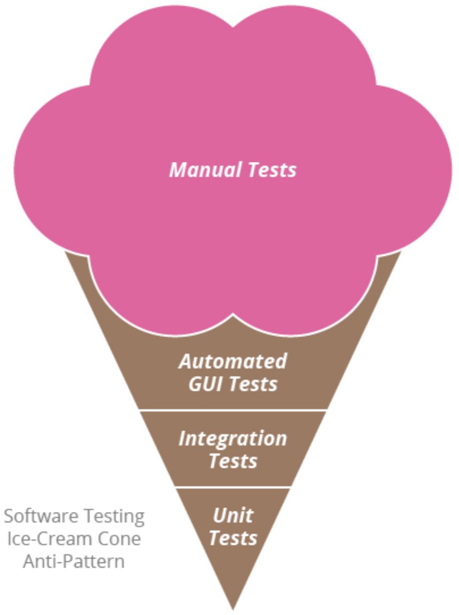
Tests	Focus	Advantages	Stub (Y/N)	How many?
Unit Tests	1. Code level sanity checking 2. Edge cases like different data inputs can be tested here	Helps identify defects on multiple data inputs to a specific function	Yes - All integrations are mocked	>100
API Tests	1. Testing the behavior of API 2. Cases which is related to functionality provided by the API	Helps identify defects in API behavior while feature development	Yes - Integrations outside the service boundary can be mocked	10-30 per service
Contract Tests	1. Focus on API contract level assertions 2. Mainly where there are integrations with other services	Helps identify changes in dependent components within and outside the system	No - Runs against original integration services	Only those which has external integrations
UI / Functional Tests	1. Simulate user behavior on the application 2. Different UI elements on the screen needs to be asserted 3. Logic placed in UI layer needs to be asserted	Helps identify defects in UI components with multiple simulated integrations	Yes - Can mock the integration services	Around 10-30
E2E tests	1. Assertion on integration with all components in a real time. 2. Including async behaviors should be covered	Helps identify defects in integration of components in real time environment setup	No - All systems integrations are real	<10

Test Data Management for Automated tests

- All kinds of tests needs sample test data for ensuring the functionality behaves as expected. There are several ways to manage test data:
    - Directly inserting data via the automation code into the DB for testing end to end flows.
    - Using APIs to insert necessary data and then assert behaviour
    - Using stubs or mocks for downstream integrations to mimic various responses from the system
    - One rule is to always make sure each test creates its own data and erases the data at the end of the test. So that the tests can be run independent.
    - Test Data have to be managed properly anticipating parallel execution of tests - such as new users for different tests

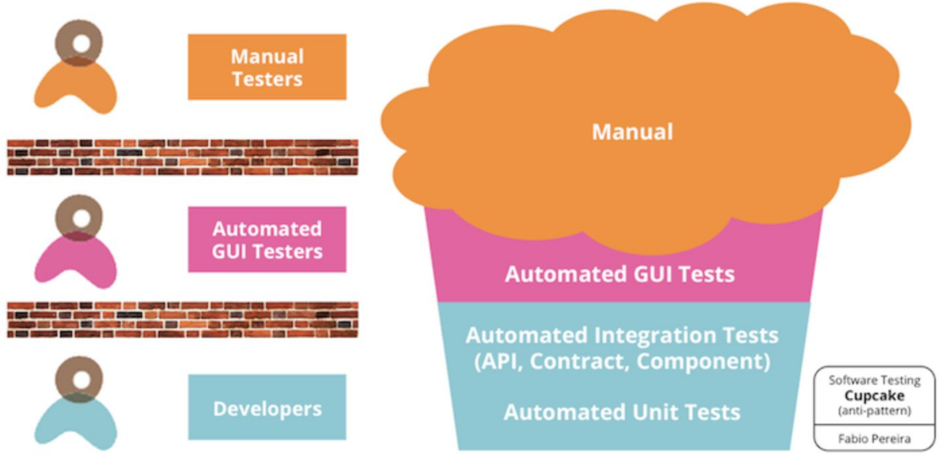
Testing Antipatterns

Ice Cream Cone



Software Testing
Ice-Cream Cone
Anti-Pattern

Cup Cake



Quick intro to Unit tests

- Unit Tests are written to validate the smallest unit of code.
- Usually Test Driven Development (TDD) practice calls out that the devs start writing a ‘failing’ unit test before writing functional code and aim to fix the test.
- This practice helps to code for functionality which is just necessary and also to cover all possible edge cases for a small unit.
- Usually the build fails if the unit tests coverage is less than an agreed percentage (~85%)
- These are the fastest and easiest tests to run and write. Hence this layer has to be bulk

For example, let’s say the order total is expected to be a decimal value such as €64.5. While writing the functionality to calculate total, the following scenarios should be covered in unit tests level -

- Total ending up in min and max digits
- Rounding of decimal values to 2
- Total comes to a negative value, hence assert error
- Total throwing ‘null’ due to input being non digits (characters, symbols etc), hence assert error.

Quick intro to API / Service tests

- API Tests are written to validate the business logic abstracted as services or APIs
- API tests should focus more on validating business use cases and outcomes and not focus on tiny details like how the unit tests were covering.
- API tests are also easier to write and faster to run.
- These are integrated into CI. Only when they pass the build is ready for QA.
- API tests can also be utilised to be '*contract tests*' to assert whether the agreed contract between APIs are consistent (not necessarily focussing on business use cases)

For example, let's say we have a API endpoint for checking final availability for the order items, the order items have to be split by lines and separate stock check have to be made and returned status for each item

- The API tests will have to cover items from different categories with different rules of availability check
- API will have to cover large line items
- API will cover error statuses when item not found, item unavailable

Quick intro to UI tests

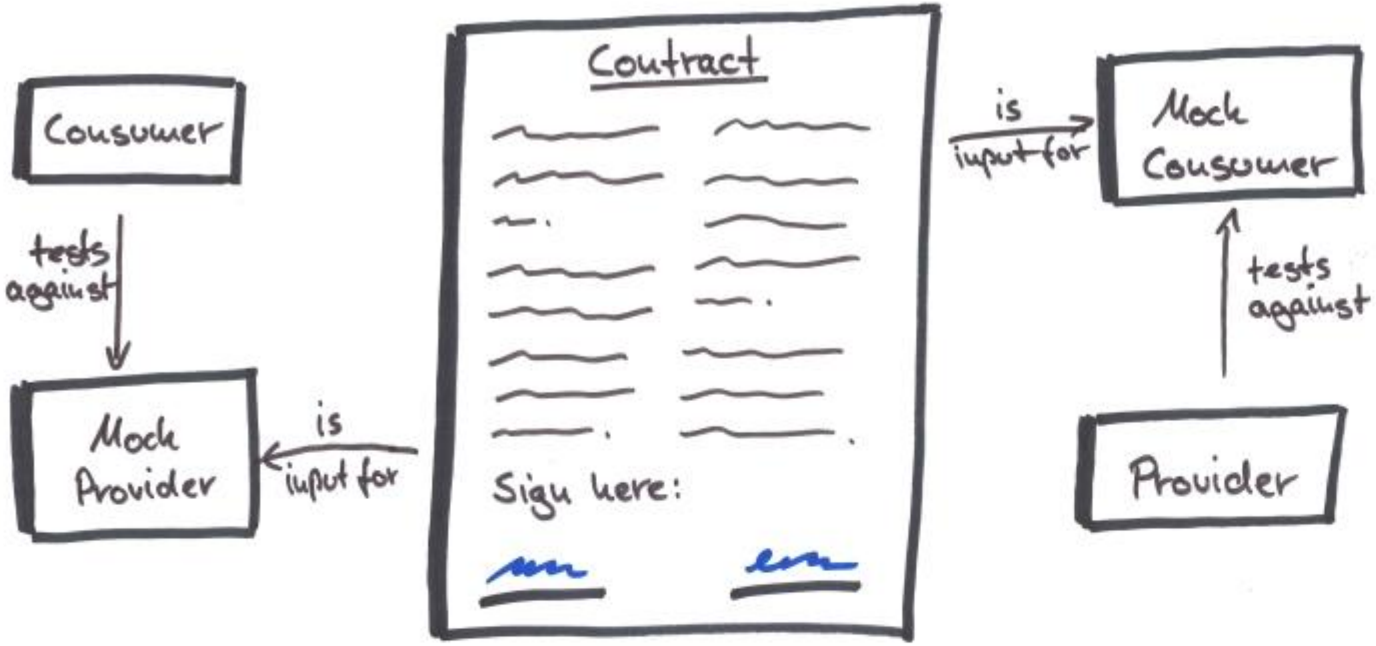
- UI Tests are at a user flow level such as Add item to cart, purchase etc
- The tests will enact exact flows by opening browser, clicking on relevant steps to perform a functionality.
- Typically these tests take much more longer time to write as well as to run in CI. Hence these tests have to be kept minimal.
- These are either run as ‘Smoke tests’ (a selective set of tests to cover the important flows) in the CI pipeline and the rest is covered as nightly regression tests.
- Or if the number of tests are less, then entire suite of tests can run as part of Smoke tests.

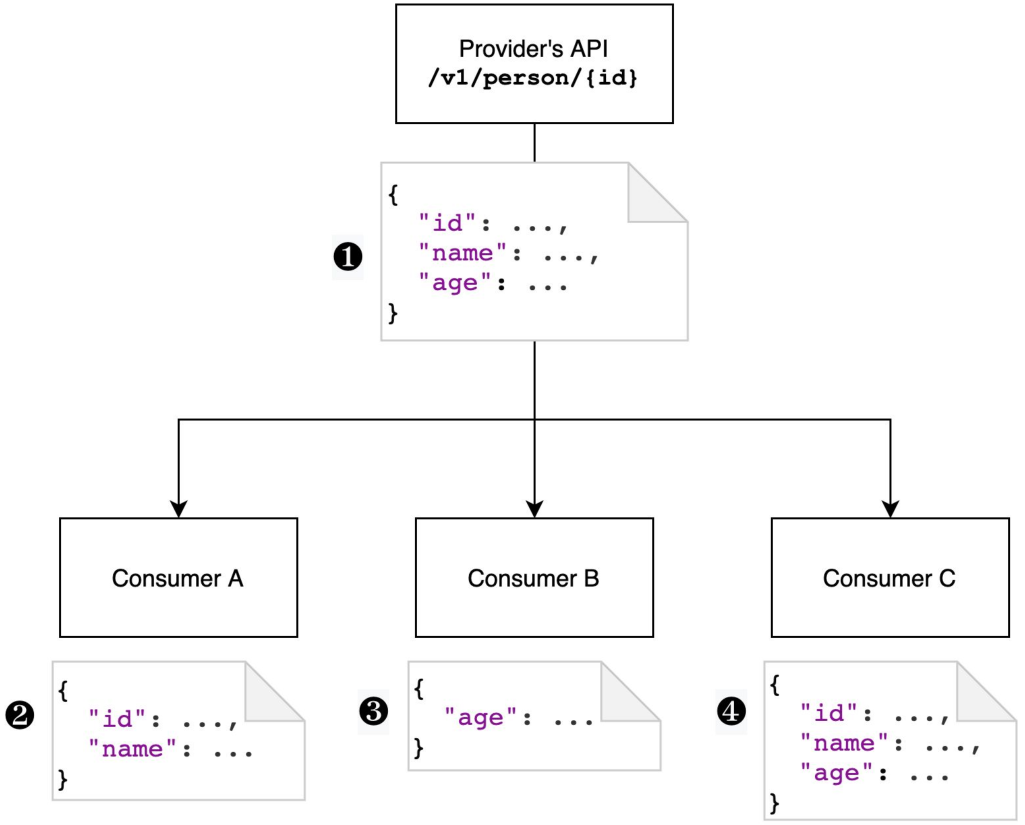
Quick intro to E2E tests

- E2E tests are kept minimal to mainly validate the integrations of systems end to end with actual flow of data.

Quick intro to Contract tests

- Contract tests are only meant to verify contracts of integration systems so that the current owner of the system is notified of any contract changes and can make the current system compatible to new contracts


Consumer
tests against
Mock Provider
is input for
Contract
Sign here:
is input for
Mock Consumer
tests against
Provider



```
{
  "id": ...,
  "name": ...,
  "age": ...
}
```

```
{
  "id": ...,
  "name": ...
}
```

```
{
  "age": ...
}
```

```
{
  "id": ...,
  "name": ...,
  "age": ...
}
```

Thank You

/thoughtworks