# Behavior Driven Development (BDD) and Functional Testing

Eric Elliott Follow
May 25 · 14 min read

```
import { Selector, ClientFunction } from 'testcafe';

fixture `TDD Day Homepage`
  .page('https://tddday.com');

test('Page should load and display the correct title', async t => {
  const actual = Selector('h1').innerText;
  const expected = 'TDD DAY 2019';
  await t.expect(actual).eql(expected);
});
```

Unit testing is a methodology where units of code are tested in isolation from the rest of the application. A unit test might test a particular function, object, class, or module. Unit tests are great to learn whether or not individual parts of an application work. NASA had better know whether or not a heat shield will work before they launch the rocket into space.

But unit tests don't test whether or not units work together when they're composed to form a whole application. For that, you need integration tests, which can be collaboration tests between two or more units, or full end-to-end functional tests of the whole running application (aka system testing). Eventually, you need to launch the rocket and see what happens when all the parts are put together.

There are multiple schools of thought when it comes to system testing, including Behavior Driven Development (BDD), and functional testing.

## What is Behavior Driven Development?

**Behavior Driven Development** (BDD) is a branch of Test Driven Development (TDD). BDD uses human-readable descriptions of software user requirements as the basis for software tests. Like Domain Driven Design (DDD), an early step in BDD is the definition of a shared vocabulary between stakeholders, domain experts, and engineers. This process involves the definition of entities, events, and outputs that the users care about, and giving them names that everybody can agree on.

BDD practitioners then use that vocabulary to create a domain specific language they can use to encode system tests such as User Acceptance Tests (UAT).

Each test is based on a user story written in the formally specified ubiquitous language based on English. (A ubiquitous language is a vocabulary shared by all stakeholders.)

A test for a transfer in a cryptocurrency wallet might look like this:

```
Story: Transfers change balances

As a wallet user
In order to send money
I want wallet balances to update

Given that I have $40 in my balance
And my friend has $10 is their balance
When I transfer $20 to my friend
Then I should have $20 in my balance
And my friend should have $30 in their balance.
```

Notice that this language is focused exclusively on the business value that a customer should get from the software rather than describing the user interface of the software, or how the software should accomplish the goals. This is the kind of language you could use as input for the UX design process. Designing these kinds of user requirements up front can save a lot of rework later in the process by helping the team and customers get on the same page about what product you're building.

From this stage, there are two paths you can venture down:

1. Give the test a concrete technical meaning by turning the description into a domain specific language (DSL) so that the human-readable description doubles as machine-readable code, (continue on the BDD path) or
2. Translate the user stories into automated tests in a general-purpose language, such as JavaScript, Rust, or Haskell.

Either way, it's generally a good idea to treat your tests as black box tests, meaning that the test code should not care about the implementation details of the feature you're testing. Black box tests are less brittle than white box tests because, unlike white box tests, black box tests won't be coupled to the implementation details, which are likely to change as requirements get added or adjusted, or code gets refactored.

Proponents of BDD use custom tools such as Cucumber to create and maintain their custom DSLs.

For contrast, proponents of functional tests generally test functionality by simulating user interactions with the interface and comparing the actual output to the expected output. In web software, that typically means using a test framework which interfaces with the web browser to simulate typing, button presses, scrolling, zooming, dragging, etc, and then selecting the output from the view.

I typically translate user requirements into functional tests rather than keep up BDD tests, mostly because of the complexity of integrating BDD frameworks with modern applications, and the cost of maintaining custom, English-like DSL whose definitions may end up spanning several systems, and even several implementation languages.

I find the layman-readable DSL useful for very high-level specifications as a communications tool between stakeholders, but a typical software system will require orders of magnitude more low-level tests in order to produce adequate code and case coverage to prevent show-stopping bugs from reaching production.

In practice, you have to translate *"I transfer $20 to my friend"* into something like:

1. Open wallet
2. Click transfer
3. Fill in the amount
4. Fill in the receiver wallet address
5. Click `[Send money]`
6. Wait for a confirmation dialog
7. Click "Confirm transaction"

A layer below that, you're maintaining state for the "transfer money" workflow, and you'll want unit tests that ensure that the correct amount is being transferred to the correct wallet address, and a layer below that, you'll want to hit the blockchain APIs to ensure that the wallet balances were actually adjusted appropriately (something that the client may not even have a view for).

These different testing needs are best served by different layers of tests:

1. **Unit tests** can test that local client state is updated correctly and presented correctly in the client view.
2. **Functional tests** can test UI interactions and ensure that user requirements are met at the UI layer. This also ensures that UI elements are wired up appropriately.
3. **Integration tests** can test that API communications happen appropriately and that the user wallet amounts were actually updated correctly on the blockchain.

I have never met a layman stakeholder who is remotely aware of all of the functional tests verifying even the top-most level UI behavior, let alone one who cares about all of the lower level behaviors. Since laymen are not interested, why pay the cost of maintaining a DSL to translate for them?

Regardless of whether or not you practice the full BDD process, it has a lot of great ideas and practices we should not lose sight of. Specifically:

- The formation of a **shared vocabulary** that engineers and stakeholders can use to communicate effectively about the user needs and software solutions.
- The creation of **user stories** and scenarios that help formulate acceptance criteria and a *definition of done* for a particular feature of the software.
- The practice of **collaboration** between users, the quality team, product team, and engineers to reach consensus on what the team is building.

Another approach to system testing is functional testing.

## What is Functional Testing?

The term "functional testing" can be confusing because it has had several meanings in software literature.

IEEE 24765 gives two definitions:

> *1. testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions [i.e., black box testing]*
>
> *2. testing conducted to evaluate the compliance of a system or component with specified functional requirements*

The first definition is general enough to apply to almost all popular forms of testing, and already has a perfectly suitable name that is well understood by software testers: "black box testing". When I'm talking about black box testing, I'll use that term, instead.

The second definition is usually used in contrast to testing that is not directly related to the features and functionality of the app, but instead concentrates on other characteristics of the app, such as load times, UI response times, server load testing, security penetration testing, and so on. Again, this definition is too vague to be very useful on its own. Usually, we want to get more specific about what kind of testing we're doing, e.g., unit testing, smoke testing, user acceptance testing?

For those reasons, I prefer another definition that has been popular recently. IBM's Developer Works says:

> *Functional tests are written from the user's perspective and focus on system behavior that users are interested in.*

That's a lot closer to the mark, but if we're going to automate tests, and those tests are going to test from the user's perspective, that means we'll need to write tests which interact with the UI.

Such tests can also go by the names "UI testing" or "E2E testing", but those names don't replace the need for the term "functional tests" because there is a class of UI tests which test things like styles and colors, which are not directly related to user requirements like "I should be able to transfer money to my friend".

Using "functional testing" to refer to testing the user interface to ensure that it fulfills the specified user requirements is usually used in contrast to unit testing, which is defined as:

> *the testing of individual units of code (such as functions or modules) in isolation from the rest of the application*

In other words, while a unit test is for testing individual units of code (functions, objects, classes, modules) in isolation from the application, a functional test is for testing the

units in integration with the rest of the app, from the perspective of the user interacting with the UI.

I like the classification of "unit tests" for developer-perspective code units, and "functional tests" for user-perspective UI tests.

## Unit Tests vs Functional Tests

Unit tests are typically written by the implementing programmer, and test from the programmer's perspective.

Functional tests are informed by the user acceptance criteria and should test the application from the user's perspective to ensure that the user's requirements are met. On many teams, functional tests may be written or expanded on by quality engineers, but every software engineer should be aware of how functional tests are written for the project, and what functional tests are required to complete the "definition of done" for a particular feature set.

Unit tests are written to test individual units in isolation from the rest of the code. There are two major benefits to this approach:

1. **Unit tests run very fast** because they're not dependent on other parts of the system, and as such, typically have no asynchronous I/O to wait on. It's much faster and less expensive to find and fix a flaw with unit tests than it is to wait for a complete integration suite to run. Unit tests typically complete in milliseconds, as opposed to minutes or hours.
2. **Units must be modular** in order to make it easy to test them in isolation from other units. This has the added benefit of being very good for the architecture of the application. Modular code is easier to extend, maintain, or replace because the effects of changing it are generally limited to the module unit under test. Modular applications are more flexible and easier for developers to work with over time.

Functional tests on the other hand:

1. **Take longer to run,** because they must test the system end-to-end, integrating with all the various parts and subsystems the application relies on to enable the user workflow being tested. Large integration suites sometimes take hours to run. I've heard stories of integration suites that took days to run. I recommend hyper-optimizing your integration pipeline to run in parallel so that it can complete in under 10 minutes — but that's still too long for developers to wait on every change.
2. **Ensure that the units work together as a whole system.** Even if you have excellent unit test code coverage, you still need to test your units integrated with the rest of the application. It doesn't matter if NASA's heat shields work if they don't stay attached to the rocket on reentry. Functional tests are a form of system tests which ensure that the system as a whole behaves as expected when it's fully integrated.

Functional tests without unit tests can never provide deep enough code coverage to be confident that you have an adequate regression safety net for continuous delivery. Unit

tests provide code coverage depth. Functional tests provide user requirement test case coverage breadth.

> *Functional tests help us build the right product. (Validation)*
> *Unit tests help us build the product right. (Verification)*

You need both.

> *Note: See Validation vs Verification. Build the right product vs build the product right distinction was succinctly described by Barry Boehm.*

## How to Write Functional Tests for Web Applications

There are lots of frameworks that allow you to create functional tests for web applications. Many of them use an interface called Selenium. Selenium is a cross-platform, cross-browser automation solution created in 2004 which allows you to automate interactions with the web browser. The trouble with Selenium is that it is an engine external to the browsers which relies on Java, and getting it to work together with your browsers can be harder than it needs to be.

More recently, a new family of products has popped up which integrate far more smoothly with browsers with fewer pieces to worry about installing and configuring. One of those solutions is called TestCafe. It's the one that I currently use and recommend.

Let's write a functional test for the TDD Day website. First, you'll want to create a project for it. In a terminal:

```
mkdir tddday
cd tddday
npm init -y # initialize a package.json
npm install --save-dev testcafe
```

Now we'll need to add a `"testui"` script to our `package.json` in the `scripts` block:

```
{
  "scripts": {
    "testui": "testcafe chrome src/functional-tests/"
  }
  // other stuff...
}
```

You can run the tests by typing `npm run testui`, but there aren't any tests to run yet.

Create a new file at `src/functional-tests/index-test.js`:

```
import { Selector } from 'testcafe';
```

TestCafe automatically makes the `fixture` and `test` functions available. You can use `fixture` with the tagged template literal syntax to create titles for groups of tests:

```
fixture `TDD Day Homepage`
  .page('https://tddday.com');
```

Now you can select from the page and make assertions using the `test` and `Select` functions. When you put it all together, it looks like this:

```
import { Selector } from 'testcafe';

fixture `TDD Day Homepage`
  .page('https://tddday.com');

test('Page should load and display the correct title', async t => {
  const actual = Selector('h1').innerText;
  const expected = 'TDD DAY 2019';
  await t.expect(actual).eql(expected);
});
```

TestCafe will launch the Chrome browser, load the page, wait for the page to load, and wait for your selector to match a selection. If it doesn't match anything, the test will eventually time out and fail. If it does match something, it will check the actual selected value against the expected value, and the test will fail if they don't match.

TestCafe provides methods to test all sorts of UI interactions, including clicking, dragging, typing text, and so on.

TestCafe also supplies a rich selector API to make DOM selections painless.

Let's test the registration button to ensure that it navigates to the correct page on click. First, we'll need a way to check the current page location. Our TestCafe code is running in Node, but we need it to run in the client. TestCafe supplies a way for us to run code in the client. First, we'll need to add `ClientFunction` to our import line:

```
import { Selector, ClientFunction } from 'testcafe';
```

Now we can use it to test the window location:

```
const getLocation = ClientFunction(() => window.location.href);

test('Register button should navigate to registration page',
async t => {
  // Flexible selectors let us select arbitrary things on the page,
  // regardless of how the page was marked up.
  const registerButton = Selector('span').withText('REGISTER NOW');
  const expected =
    'https://zoom.us/webinar/register/WN_rYdjYdXFTPiHCsiWsnq0jA';

  // Wait for the button click navigation
  await t.click(registerButton);
```

```
  // Now check the location.
  await t.expect(getLocation())
    .eql(expected);
});
```

If you're not sure how to do what you're trying to do, TestCafe Studio lets you record and replay tests. TestCafe Studio is a visual IDE for interactively recording and editing functional tests. It's designed so that a test engineer who may not know JavaScript can build a suite of functional tests. The tests it generates automatically await asynchronous jobs like page loads. Like the TestCafe engine, TestCafe Studio can produce tests which can be run concurrently across many browsers, and even remote devices.

TestCafe Studio is a commercial product with a free trial. You do not need to purchase TestCafe studio to use the open source TestCafe engine, but the visual editor with built-in recording features is definitely a tool worth exploring to see if it's right for your team.

TestCafe has set a new bar for cross-browser functional testing. Having endured many years of trying to automate cross-platform tests, I'm happy to say that there is finally a fairly painless way to create functional tests, and there is now no good excuse to neglect your functional tests, even if you don't have dedicated quality engineers to help you build your functional test suite.

## Dos and Don'ts of Functional Tests

- **Don't alter the DOM.** If you do, your test runner (e.g., TestCafe) may not be able to understand how the DOM changed, and that DOM alteration could impact the other assertions which may be relying on the DOM output.
- **Don't share mutable state between tests.** Because they're so slow, it's incredibly important that functional tests can be run in parallel, and they can't do that deterministically if they're competing for the same shared mutable state, which could cause nondeterminism due to race conditions. Because you're running system tests, keep in mind that if you're modifying user data, you should have different test user data in the database for different tests so they don't randomly fail due to race conditions.
- **Don't mix functional tests in with unit tests.** Unit tests and functional tests should be written from different perspectives, and run at different times. Unit tests should be written from the developer's perspective and run every time the developer makes a change, and should complete in less than 3 seconds. Functional tests should be written from the user's perspective, and involve asynchronous I/O which can make test runs too slow for a developer's immediate feedback on every code change. It should be easy to run the unit tests without triggering functional test runs.
- **Do run tests in headless mode,** if you can, which means that the browser UI doesn't actually need to be launched, and tests can run faster. Headless mode is a great way to speed up most functional tests, but there is a small subset of tests which can't be run in headless mode, simply because the functionality they rely on doesn't work in headless mode. Some CI/CD pipelines will require you to run functional tests in headless mode, so if you have some tests which can't run in headless mode, you may

need to exclude them from the CI/CD run. Be sure the quality team is on the lookout for that scenario.

- **Do run tests on multiple devices.** Do your tests still pass on mobile devices? TestCafe can run on remote browsers without installing TestCafe to the remote devices. However, screenshot functionality does not work on remote browsers.
- **Do grab screenshots on test failures.** It can be useful to take a screenshot if your tests fail to help diagnose what went wrong. TestCafe studio has a run configuration option for that.
- **Do keep your functional test runs under 10 minutes.** Any longer will create too much lag between the developer working on a feature and fixing something that went wrong. 10 minutes is long enough for a developer to get busy working on the next feature, and if the test fails after longer than 10 minutes, it will likely interrupt the developer who has moved on to the next task. An interrupted task takes an average of twice as long to complete and contains roughly twice as many errors. TestCafe allows you to run many tests concurrently, and the remote browser option can do so across a fleet of test servers. I recommend taking advantage of those features to keep your test runs as short in duration as possible.
- **Do halt the continuous delivery pipeline when tests fail.** One of the great benefits of automated tests is the ability to protect your customers against regressions — bugs in features that used to work. This safety net process can be automated so that you have good confidence that your release is relatively free of bugs. Tests in the CI/CD pipeline effectively eliminate a development team's fear of change, which can be a serious drain on developer productivity.

## Next Steps

Join TDD Day.com — an all-day TDD curriculum featuring 5 hours of recorded video content, projects to learn unit testing and functional testing, how to test React components, and an interactive quiz to make sure you've mastered the material.

. . .

***Eric Elliott*** *is a distributed systems expert and author of the books, "Composing Software" and "Programming JavaScript Applications". As co-founder of DevAnywhere.io, he teaches developers the skills they need to work remotely and embrace work/life balance. He builds and advises development teams for crypto projects, and has contributed to software experiences for* ***Adobe Systems, Zumba Fitness, The Wall Street Journal, ESPN, BBC,*** *and top recording artists including* ***Usher, Frank Ocean, Metallica,*** *and many more.*

*He enjoys a remote lifestyle with the most beautiful woman in the world.*

Thanks to JS_Cheerleader.

JavaScript Software Engineering Unit Testing Technology Tdd