circleci

Software Testing for DevOps-Driven Teams

BY JUNE JUNG,
ENGINEERING MANAGER
github.com/junejung

If you've been hearing the term DevOps a lot recently but you're not quite sure what it is or whether it's relevant to your organization, you've come to the right place. We're here to break down exactly what DevOps is so you can determine whether it would benefit your engineering team and, ultimately, your end users.

We'll cover this in depth, but for a short introduction, DevOps is an approach to software engineering that brings the domains and skillsets of developers and operators together to facilitate faster, better development infrastructures. It's essentially an iterative process that enables you to test early and often, and to consider the extended lifecycle and operability of code early on, to identify problems and solve them that much faster. The end goal is to increase both productivity and the quality of the product and the end user experience.

As we'll address in this ebook, DevOps isn't right for every organization. If your current workflow functions well, transitioning to DevOps may be an unnecessary undertaking. Depending on your product, it may be the wrong move as well. After all, an iterative process wouldn't serve a company that must do major launches due to privacy and other concerns.

However, DevOps is becoming increasingly popular for a reason. In this ebook, we'll walk you through exactly what DevOps is, when it makes sense to adopt it, and how and when to test throughout your processes.

Table of Contents

Moving to DevOps: What tools do you really need?

DevOps has been the latest in a long succession of problem-solving processes that each come with a digital garage full of tools: CI/CD systems, testing frameworks, monitoring tools, and security audit tools to name a few.

Thinking about DevOps raises a number of questions for organizations. Which of these tools do you need? Which will solve the problems, pains, and slowdowns your organization faces? What organizational structure do you need to have to support it? What tools should you implement?

These are all fine questions to ask, but asked in isolation, they miss the point. When you go straight to these queries, you are thinking about solutions before you've really assessed the problem that you are trying to solve.

Organizations often think that the top-down model ("Use this! Do this!") will get their teams innovating faster. Eager team leads will bring in a new CI/CD tool and get everyone up and running on it. Providing proper tooling for individual contributors to follow an adopted practice is important, but the problem begins when team leads bring in a tool without fully understanding its value or why they are doing it. Oftentimes, even the people who ordered the change will forget why they put it in place to begin with, or what they wanted to get out of it. The sad truth is how easy it is to start misusing tools once the original reasoning is obscured, thus creating lost or even negative value.

We want DevOps!

Many organizations are convinced that DevOps is the solution to all their problems, if only they can get it up and running quickly. If you're in this position, ask yourself, "Why do we want DevOps in our organization? What value do we think it will bring?"

At this point, let's briefly talk about what DevOps is, and isn't.

DevOps is a very specific collaboration between developers and operations teams. In essence, it indicates that you've culturally adopted development practices into your infrastructure and operational practices into your development cycle. What does this look like in practice? It can mean maintaining infrastructure as code, or creating immutable infrastructure by building reusable components so you can tear down or up whenever you want, giving you version control and a history of changes you've made.

It also means getting all product contributors to care more about the end result of what they're working on — how does it function in the world? How are users interacting with it? Getting people to truly care about quality means caring about both business value and usability. When everyone who is building a product cares about both of these aspects, you know you've achieved true DevOps adoption.

In our experience, this kind of widespread buy-in has been particularly difficult for software teams to achieve because it requires a lot of cooperation from people with different skills and domain expertise. Pulling this off depends on both cross-functional team structure and thoughtful communication skills. For example, if an engineer needs to talk to someone on the business side about a database problem, she needs to not just show the data she's working from, but give necessary context to focus that person's attention on what they should care about and why.

New tools can sometimes seem like a quick fix, but they are not a one-size-fits-all solution. In our experience, keeping the following considerations in mind before you bring in a new DevOps tool will increase your chances of success.

1. **Make sure everyone is on the same page with respect to what you're trying to achieve with this transformation.** Everyone should agree on the problem you're trying to solve and should be aligned on the pain points.

2. **Always start small.** Don't try to make your entire organization into a model DevOps team overnight. Instead, start with one team, and see if the process change works within that group. If you see improvements, keep moving incrementally.

3. **Do what works for you.** Know that DevOps might not be the right solution for your organization. Some companies have been successful for a long time without DevOps, and it might not be right for them, given their cultures or their product needs. We've seen waterfall work really well for some very successful organizations. For example, if confidentiality is a big part of your company's product strategy, then shipping incrementally to get feedback would not work for you, as you'd need to keep all product details under lock and key until a big launch. In that environment, it would be very difficult, and counterproductive, to build a DevOps culture.

4. **Always measure.** Before you start any improvement plan, get accurate metrics for where you're currently at (i.e. "our dev cycle takes X time"). Measure before and after you make a change, to see if you've improved. As an example: When Agile transformation was at its peak, a lot of companies adopted standups (brief daily status meetings), without really understanding why, and without measuring whether it had a positive effect on their team. This likely wasted more time than it saved.

5. **Do not try to automate everything.** At least not all at once. One misconception about DevOps is that all the infrastructure provisioning and the configuration management must be done automatically. This is referred to as "infrastructure as code." But some things work better when they are manual; automation is not the solution for everything. Also, think about how many times you're going to run that automation script and how much time it would take you to set up. Will you use it thousands of times or only three? Additionally, sometimes you have to start the manual way to even figure out what would be the best solution for automation.

Still itching to automate? Dockerizing your application is a great way to automate because the work you put in is likely to be re-used. Automating pre-production environment creation is another great way to implement automation. As another example: are you trying to automate firewall setup? That might not be worth it given the lack of API support on a lot of current firewall software. While it's prudent to prepare for disaster, you'd probably be putting so much more value into it than you'll ever get back out of it.

If your org is thinking about DevOps transformation, start thinking about your speed of delivery and quality of product. What's getting in your way today? Knowing the answers to these questions will help everyone in your organization understand what your pain points are, so you'll be in the best position to start improving on them.

The path to production: how and where to segregate test environments

Once you decide to transition to DevOps, bear in mind that bringing a new tool into an organization is no small task. Adopting a CI/CD tool, or any other tool should follow a period of research, analysis, and alignment within your organization.

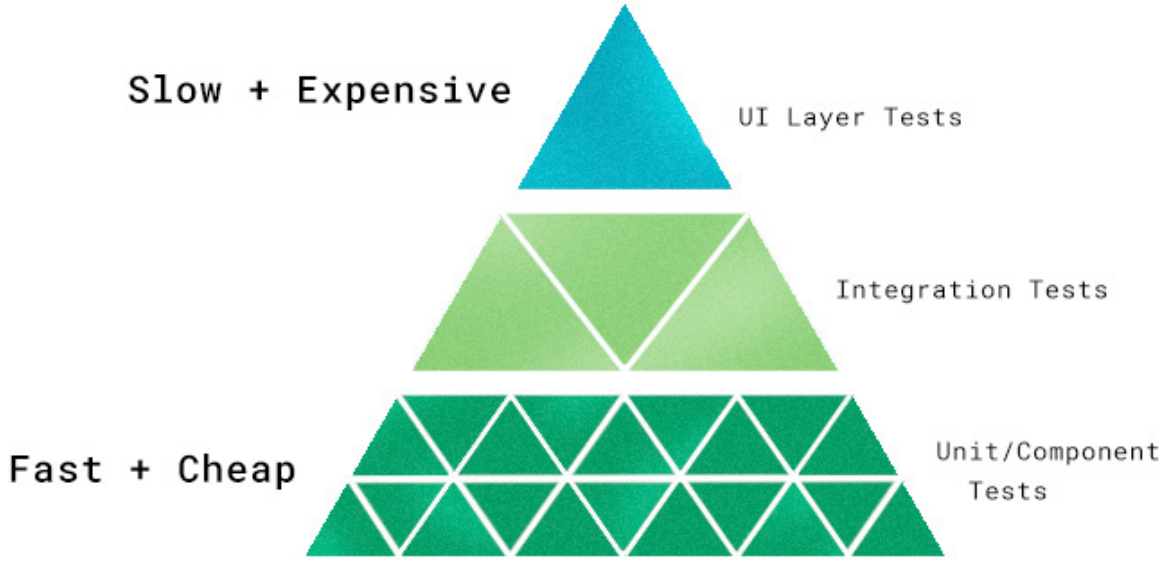
The precursor to any successful tool adoption is about people: alignment on purpose, and setting expectations appropriately, as well as getting some "before" metrics to support your assessment.

The next step is about analysis: deducing exactly what pipeline problem you most need to solve. By examining your development process (which we'll refer to as your "path to production") you can pinpoint where your biggest problems are coming from. Only when you know what problem you're trying to solve can you make a well-reasoned tooling decision.

The goal in examining your path to production is creating clear stages which will serve as checkpoints. In order to pass from one stage to the next, a build has to pass through these quality gates. The gates will be separated using various kinds of tests. Once tests are passed, quality at that level has been assured, and the build can move on.

These pipeline stages, or build environments, are separated by the increasing scope of responsibility that the developer assumes, ranging from their local laptop to the team space to the application's entire codebase. As the scope of responsibility grows, the cost of mistakes is higher. That's why we test incrementally before passing a build to each successive level.

What's more, each stage requires different types of tests. As you move from staging toward production, the tests go from lighter weight to heavy duty. The cost in resources increases with each stage as well. Heavy duty testing can only happen in more production-like environments, involving a full tech stack or external dependencies. In order to do these tests properly, there's more to spin up, and they require more expensive machinery. Therefore, it's beneficial if you can do as much testing as possible in earlier environments, which are much less costly.

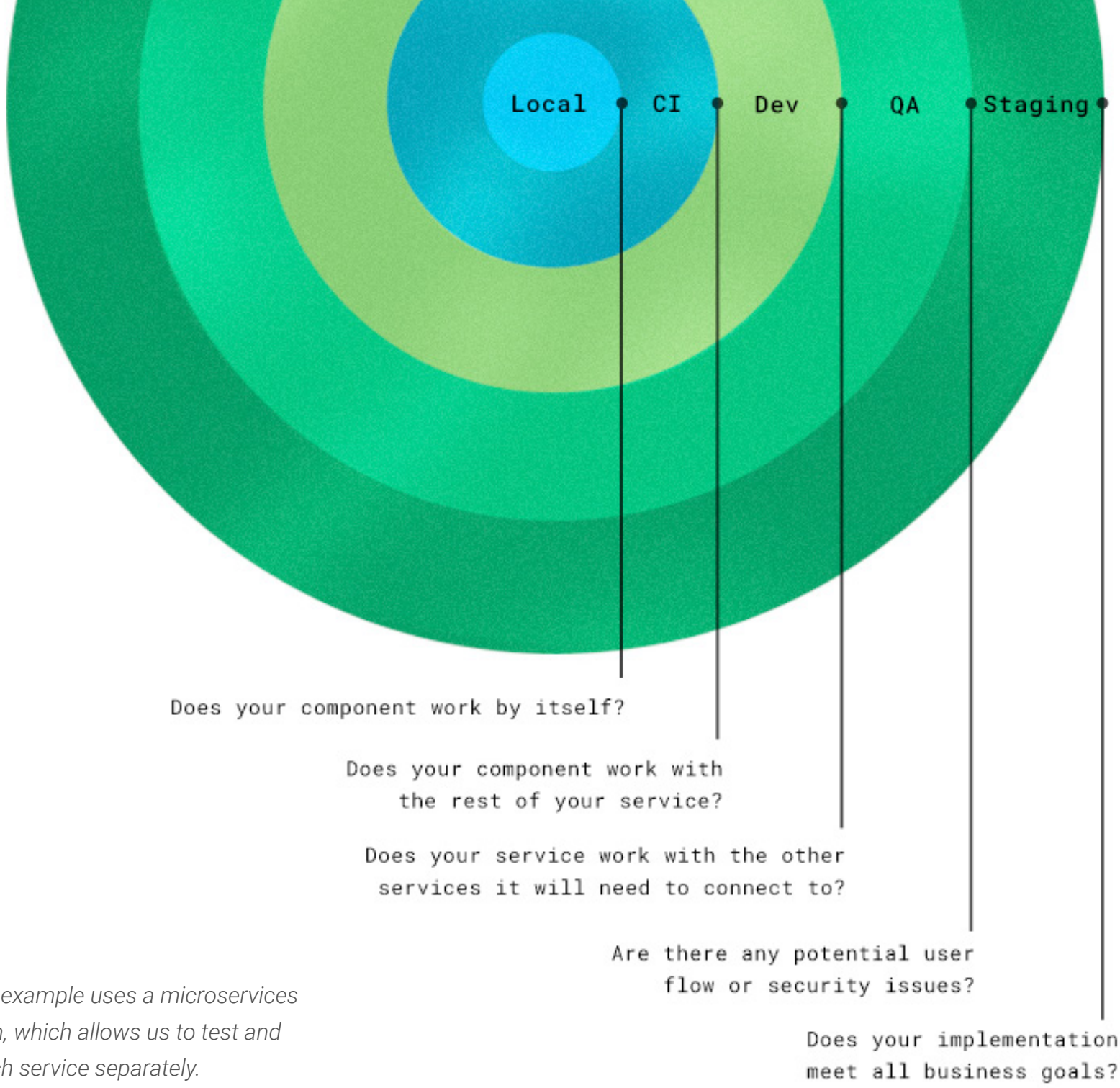


Now let's look at some common types of tests.

Unit/component test: These cover the smallest possible component, unit, or functionality. They're the cheapest and fastest tests to run since they don't require a lot of dependencies or mocking. These should be done early to get them out of the way.

Integration test: These check how well each unit from the previous stage works with the other components, units, and functionalities. In a broader sense, it can test how services (such as APIs) integrate with one another.

UI layer testing: This is automated browser-based testing which tests basic user flow. It is expensive to set up and slow to run, so it should happen later in the pipeline.



Note: This example uses a microservices application, which allows us to test and deploy each service separately.

Next up, we'll talk about how these tests fit into a software development pipeline. Each test has a specific role and place.

Local environment

This environment is private, limited to a single developer and their laptop. It is the easiest in which to make changes and test your own implementation.

The "local" we are talking about here is really just a "personal environment." It could be on your laptop, but you could just as easily use a cloud environment if the application is too big to run on your local machine. The key here is it's a smaller scale instance that's yours alone, in which you can test and debug your implementation while you're developing, without interrupting other developers on your team. Because nothing in your local environment is visible to others, your team can agree to integrate a Git pre-push hook into your repo to ensure the local environment is used and to run automated tests before code gets pushed to a remote, or shared, repository.

Tests: The tests we recommend before moving out of a local environment are unit tests, integration testing with mocked components, and UI testing to the degree that it's possible. The more tests you can do in this environment, the more room you'll have to be successful in the integration environment.

Scope of responsibility: The scope here covers just the implementation or functionality of what you're building. Does your component work by itself? The surface area is relatively small. But this is the last environment before you merge to a shared environment. Therefore, you'll want to test responsibly so that you don't break the shared environment later and block active development by others.

CI environment

This is the shortest-lived environment; it lives with the build. It gets created when a build gets triggered and torn down once the build is done. It's also the most unstable environment. Our developers check code into the CI environment. Since other developers could be deploying at the same time, the CI environment has a lot of deployment activities that are happening concurrently. As a result, the CI can, and often does, break. That's ok–it's meant to break. The key is fixing it when it does.

If you were unable to spin up your entire application in a local environment (which is highly likely), the CI environment will be the first in which you'll be able to do browser-driven testing. You'll also be able to do any UI testing that you weren't able to do in your local environment.

In this environment, you should be using mock external services and databases to keep things running fast.

We suggest automating the lifetime of the CI environment like this: as soon as you merge the code, the CI environment automatically spins up, runs that code, tells you whether it's safe or not, and then tears itself down. Using Docker to automatically spin up the environment will save time, and the whole process of automated builds and environment creation will make it easier for team members to commit more often.

Development environment

The development environment is a shared environment with other developers. In this environment, every service within the application is getting deployed every time. These environments are very unstable because there are constant changes from different teams. It's important to note here that your integration and browser-based tests may now fail, even if they passed in the CI environment. That's because they are now fully integrated with outside services and other services are also currently in development.

Whereas the CI environment is just for your team (or for one service of your product) and can be torn down between builds, this development environment is for your entire product codebase. If you choose to merge into a development environment for just your team, that will have a smaller scope of impact than merging into the full development environment shared by all teams. In the development environment, system health check monitoring is now required since it's a fast moving environment with many different components. Since it sits in the middle of the path to production, when this environment breaks, it has a detrimental effect on those that follow - a failure here blocks all changes and no code can move forward if the environment breaks.

In the development environment, some external services are mocked and some are not, depending on how crucial each service is to what you're testing and also the cost of connecting to that external service. If you are using mocked data here, make sure that a decent amount of test data available.

Everyone in the organization has visibility into this environment; any developer can log in and run it as an application. This environment can be used by all the developers for testing and debugging.

QA environment

This is the first manually deployed environment in this scenario. It is manually deployed because the QA team needs to decide which features are worth testing on their own, based on the structure of the changes. They might take a stacked change (Change A, B and C) and test them each separately. In this scenario, they test Change A, and once they have assurance that it's working as expected, they can move on and test Change B and when it throws an error, they know that the issue is isolated to Change B. Otherwise, in the case of just testing Change C, if it were to throw an error, it would block progress on C, B, and A.

The QA environment is a controlled and integrated environment. Here, the QA team is controlling what change is coming in; in contrast, in the development environment, any change can happen at any time. The build is now integrated with the services it will interact with in the application.

In this environment, we now have the same infrastructure and application as in production. We are using a representative subset of production data–close enough to production data to test. The QA engineers or testers understand what they should focus their tests on. We recommend manual deployment at this stage so that small changes can be tested in an isolated environment to help identify bugs. The closer the QA environment can get to production, the higher confidence you will have in the results of the tests.

Staging environment

This is the last environment before production. The purpose of staging is to have an environment almost exactly the same as production. When you deploy something into staging, and it works, you can be reasonably assured that that version won't fail in production and cause an outage. All environments help you catch potential issues; staging is the final check of confidence. Here it is important to have almost the same amount of data as you would in production. This enables you to do load testing, and test the scalability of the application in production.

Production-ready code should be deployed to this environment; again, almost the same as in production. Infrastructure, databases, and external service integration should be exactly the same as in production. It will be expensive to maintain and build – the only thing more expensive is not doing it, and breaking production. The scale can be smaller, but your setup and your configuration has to be the same.

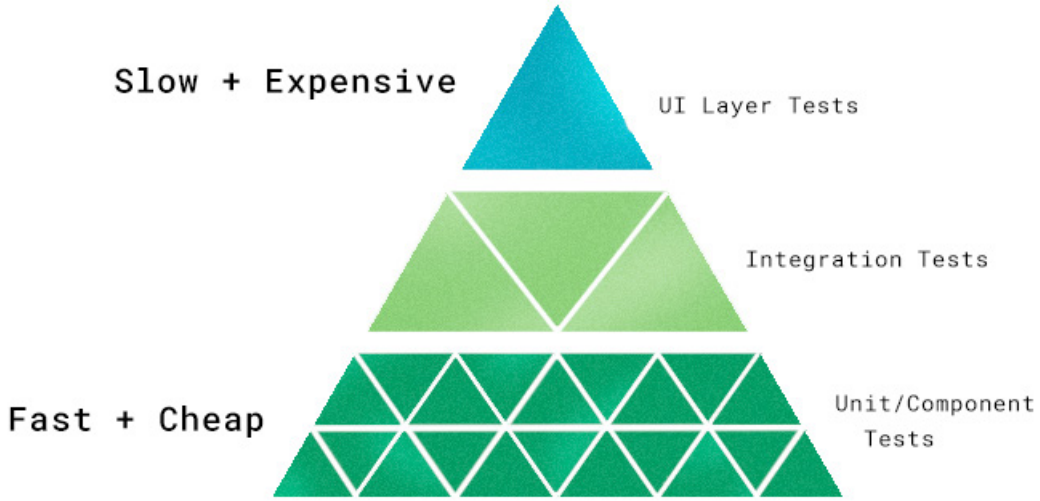
This is the last gate before production to test the implementation, migration, configuration and business requirements, so we strongly encourage you to get business signoff before you get to staging. Otherwise, it will be very expensive to make a change. While you're doing development, ensure that whoever owns the product or requested feature thoroughly understands what it is you're building. Be on the same page the whole way through.

Every developer who is implementing code in their local environment should be able to have visibility into how it will be deployed into production as well. Only by being aware of how their code is being deployed can developers make the greatest impact in reducing errors before they move onto the later stages.

How to Test Software, Part I: Mocking, Stubbing, and Contract Testing

In this section, we'll cover the techniques of **mocking and stubbing,** and **contract testing** to help each testing layer. Refer back to the test pyramid above, which helps illustrate the difference between types of tests and when it's advantageous to do each.

As we mentioned, unit or component tests (shown at the bottom of our pyramid) are inexpensive and fast to perform. Rely heavily on these. Only once you've exhausted what these tests can do, move on to more time- and resource-intensive tests, such as integration and UI layer tests.



Mocking and stubbing

A lot of people think that mocking and stubbing are used just for unit and component tests. However, we want to show you how mock objects or stubs can be used in other layers of testing as well.

Let's start with some definitions.

Mocking means creating a fake version of an external or internal service that can stand in for the real one, helping your tests run more quickly and reliably. When your implementation interacts with an object's properties, rather than its function or behavior, a mock can be used.

Stubbing, like mocking, means creating a stand-in, but a stub only mocks the behavior, not the entire object. This is used when your implementation only interacts with a certain behavior of the object. For a more in depth look at the differences between mocking and stubbing, check out Martin Fowler's post "**Mocks Aren't Stubs**." Let's discuss how we can apply these methods to improve our testing in all levels of the pyramid above.

Mocking and Stubbing in unit + component tests

Mocking of external functionality

We recommend mocking or stubbing when your code uses external dependencies like system calls, or accessing a database. For example, whenever you run a test, you're exercising the implementation. So when a delete or create function happens, you're letting it create a file, or delete a file. This work is not efficient, and the data it creates and deletes is not actually useful. Furthermore, it's expensive to clean up, because now you have to manually delete something every time. This is a case where mocking/stubbing can help a lot.

Using mocks and stubs to fake the external functionality help you create tests that are independent. For instance, say that the test writes a file to /tmp/test_file.txt and then the system under the test deletes it. The problem then is not that the

test is not independent; it is that the system calls take a lot of time. In this instance, you can stub the file system call's response, which will take a lot less time because it immediately returns.

Another benefit is that you can reproduce complex scenarios more easily. For instance, it is much easier to test the many error responses you might get from the filesystem then to actually create the condition. Say that you only wanted to delete corrupt files. Writing a corrupt file can be difficult programmatically, but returning the error code associated with a corrupt file is a matter of just changing what a stub returns.

See this example code:

```
def read_and_trim(file_path)
    return os.open(file_path).rstrip("\n")
```

This method will call system call to look for the file from the given file path and read the content from them and removing new line terminator.

The code above interacts with Python's built-in open function which interacts with a system call to actually look for the file from the given file path. This means that wherever and whenever you run the test for that function:

1. You will need to ensure that the file that the test will be looking for exists; when it does not exist, the test fails.

2. The test will need to wait for the system call's response; if the system call times out, the test fails.

Neither case of failure means your implementation failed to do its job. These tests are now neither isolated (since they're dependent on the system call's response) nor efficient (since the system call connection will take time to deliver the request and response).

The test code for the implementation above looks like this:

```
from unittest.mock import patch


content = "fake file content\n"
trimed_content = content.rstrip("\n")
@patch("builtins.open", new_callable=mock_open, read_data=content)
def test_read_trim_content(self, mock_object):
    file_path = "/fake/file/path"
    self.assertEqual(read_and_trim(file_path), trimed_content)
    mock_object.assert_called_with(file_path)
```

We are using a Python mock patch to mock the built-in open call. In this way, we are only testing what we actually built.

Another good example of using mocks and stubs in unit testing is faking database calls. For example, let's say you are testing whether your function deletes the entity from a database. For the first test, you manually create a file so that there's one to be deleted. The test passes. But then, the second time, someone else (who isn't you) doesn't know that they have to manually create the entity. Now the test fails. There was no file to delete since they didn't know they had to create the entity, so this is not an independent test.

In cases like these, you'll want to prevent modifying the data or making operating system calls to remove the file. This will prevent tests from being flaky whenever someone accidentally fails to create test data.

Mocking and stubbing of internal functions

Mocks and stubs are very handy for unit tests. They help you to test a functionality or implementation independently, while also allowing unit tests to remain efficient and cheap.

A great application of mocks and stubs in a unit/component test is when your implementation interacts with another method or class. You can mock the class object or stub the method behavior that your implementation is interacting with. Mocking or stubbing the other functionality or class, and therefore only testing your implementation logic, is the key benefit of unit tests, and the way to reap the biggest benefit from performing them.

Mocking in integration testing

With integration tests, you are testing relationships between services. One approach might be to get all the dependent services up and running for the testing environment. But this is unnecessary. It can create a lot of potential failure points from services you do not control, adding time and complexity to your testing. Try narrowing it down by writing a few service integration tests using mocks and stubs, which will make your test suite more reliable.

In integration testing, the rules are different from unit tests. Here, you should only test the implementation and functionality that you have the control to edit. Mocks and stubs can be used for this purpose. First, identify which integrations are important. Then, you can decide which external or internal services can be mocked.

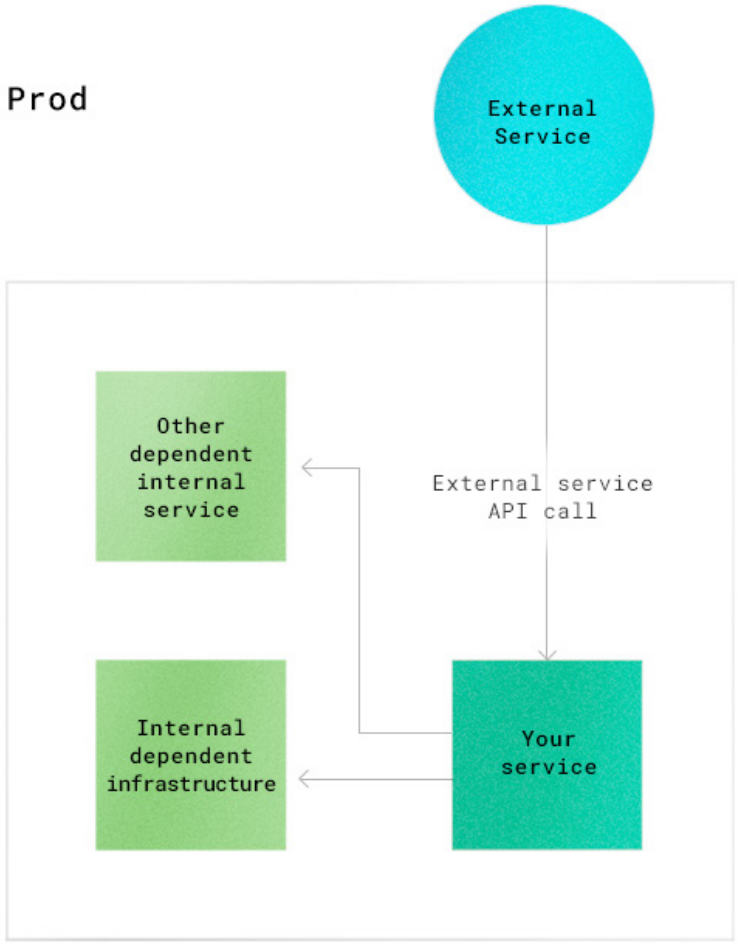
Let's say your code interacts with the GitHub API, like in the example below. Since you personally can't change how the GitHub API is responding from your request call, you don't have to test it. Mocking the expected GitHub API's response lets you focus more on testing the interactions within your internal code base.

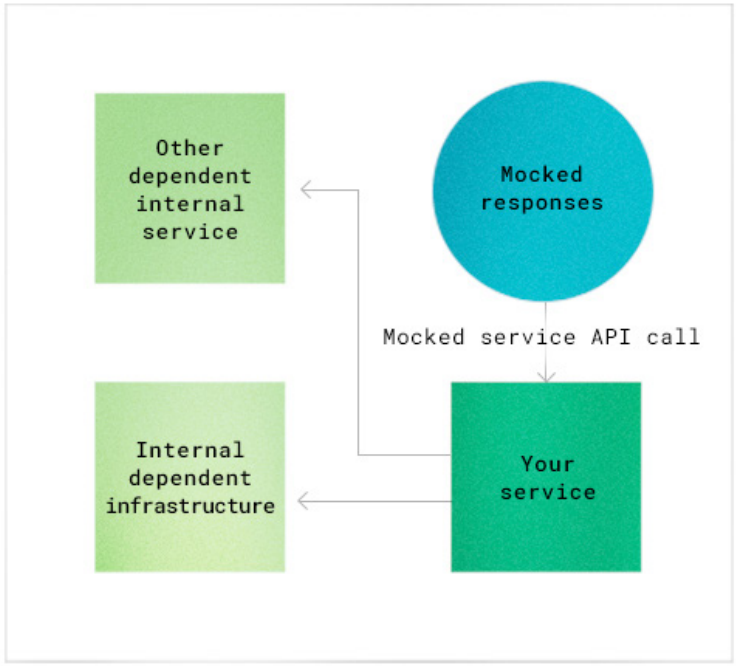
```
@unittest.mock.patch('Github')
def test_parsed_content_from_git(self,
mocked_git):
   expected_decoded_content = "b'# Sample Hello World\n\n> How to run this app\n\n- installation\n\n dependencies\n"
   mocked_git.get_repo.return_value =
expected_decoded_content
   parsed_content = read_parse_from content(repo='my/repo',
file_to_read='README.md')
   self.assertEqual(parsed_content['titles'], ['Sample Hello World'])
```

In the test code above, the **`read_parse_from_content`** method is integrated with the class that parses the JSON object from the GitHub API call. In this test, we are testing the integration between two classes.

Since we are using a mock in the test above, your test will be faster and less dependent by avoiding making the call to the GitHub API. This will also save time and effort by not needing internet access for the environment that will run the test. However, in order for you to have reliable testing while mocking the dependent external services, it's extremely important for you to understand how external dependencies will behave in the real world.

For example, if the **`expected_decoded_content`** in the code example above is not how GitHub returns the repo file content, incorrect assumptions from the mocked test can lead to unexpected breakage. Before writing the test that will have the mocked response, it's best to make the actual snapshot of the external dependency call and use it as a mocked response. Once you have created the mocked response with the snapshot, that should not change often since the Application Programming Interface should almost always be backward compatible. However, it is important to validate the API regularly for the occasional unexpected change.


Prod
External Service
Other dependent internal service
External service API call
Internal dependent infrastructure
Your service


Test Environment
Other dependent internal service
Mocked responses
Mocked service API call
Internal dependent infrastructure
Your service

Mocks and stubs in contract-based testing (in a microservices architecture)

When two different services integrate with each other, they each have "expectations," i.e. standards about what they're giving and what they expect to get in return. We can think of these as contracts between integrated endpoints. Because of this standardization, contract tests can be used to test integrations.

Let's walk through an example. The version-tagged API should not change often, possibly not ever. For any API you choose, you will generally be able to find documentation about that API, including what to expect from it. When you decide to use a certain version of an API, you can rely on the return of that API call. This is the presumed contract between the engineers who provide the API and the engineers who will use its data.

You can use the idea of contracts to test internal services as well. When testing a large scale application using microservices architecture it could be costly to install the entire system and infrastructure. Such applications can benefit greatly from using contract testing. In the testing pyramid, contract testing sits in between the unit/component testing and integration testing layers, depending on the coverage of the contract testing in your system. Some organizations utilize contract testing to completely replace end-to-end or functional testing.

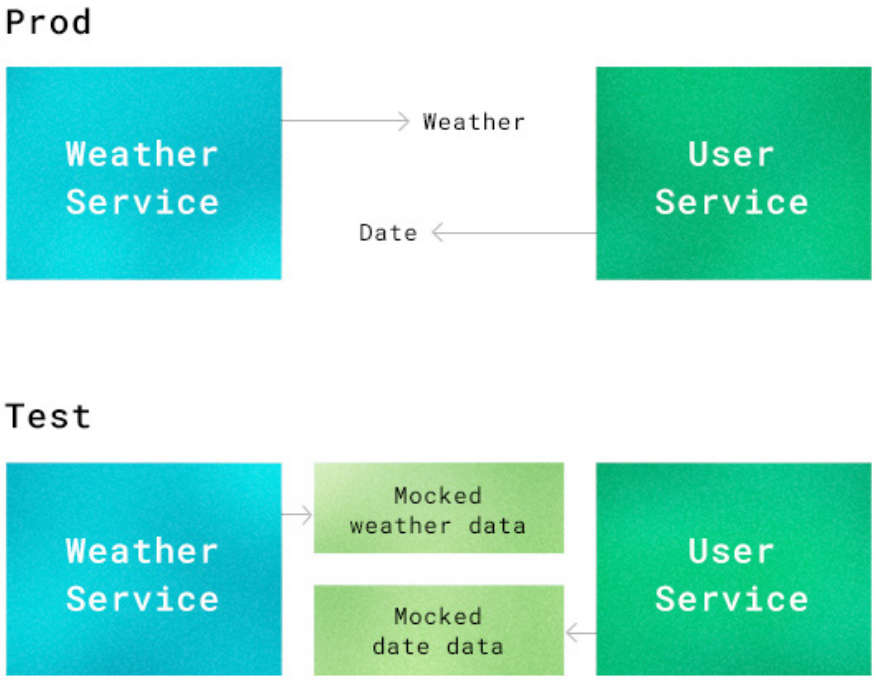
Contract-based testing can cover two important things:

1. Checking the connectivity of endpoint that has been agreed upon
2. Checking the response from the endpoint with a given argument

As an example, let's imagine a weather-reporting application involving a weather service interacting with a user service. When the user service connects to the endpoint of the weather service with the date (the request), the user service processes the date data to get the weather for that date. These two services have a contract: the weather service will maintain the endpoint to be always accessible by the user service and provide the valid data that the user service is requesting, and in the same format.

Now, let's take a look at how we can utilize mocks and stubs in the contract test. Instead of the user service making the actual request call to the weather service in the test, you can create a mocked response. Since there is a contract between two services, the endpoint and response should not change. This will free both services from depending on each other during tests, allowing tests to be faster and more reliable.

It can be useful to run the same test in a different environment with a different configuration. Contract tests are one of the great examples of the latter case. We can achieve different goals when running contract tests in different environments with different configurations. When it's a lower layer environment such as Dev or CI, running the test with a mocked contract would serve the purpose of testing our internal implementation within the constraints of the environment. However, when it goes to an upper layer environment such as QA or Staging, the same test can be used without a mocked contract but with the actual external dependency connection. **Mbtest** is one tool that can help with the kind of contract testing and mocking response explained above.



We've taken a look at examples of different layers of testing using mocks and stubs. Now let's recap why they are useful:

1. Tests with mocks and stubs go faster because you don't have to connect with external services. There's no delay waiting for them to respond.
2. You have the flexibility to scope the test to cover just the parts you can control and change. With external services, you are powerless in the case that they're wrong or the test fails. Mocking ensures you are scoping tests for work that you can actually do and not giving yourself problems you can't fix.
3. Mocking external API calls helps your test to be more reliable
4. Contract testing empowers service teams to be more autonomous in development

Next up, we'll explore the principles of test-driven development (TDD) and behavior-driven development (BDD), and see how they can improve outcomes for everything from functional testing to unit testing.

How to Test Software, Part II: TDD and BDD

So far, we've discussed what mocks and stubs are, as well as how to use them in various testing scenarios to give yourself more flexibility, speed up your tests, and get more determinism out of your test suite.

Now we're going to cover two methods for software development that take testing into consideration at the outset: test-driven development (TDD) and behavior-driven development (BDD). Using these methodologies will improve the way you think about software development, and greatly enhance the efficacy of your tests. Let's dive in:

TDD: test-driven development

TDD (test-driven development) is known as a method for writing unit tests. We are going to talk about using TDD principles for everything from functional testing to unit testing.

Red-Green Refactor: test-driven development principles

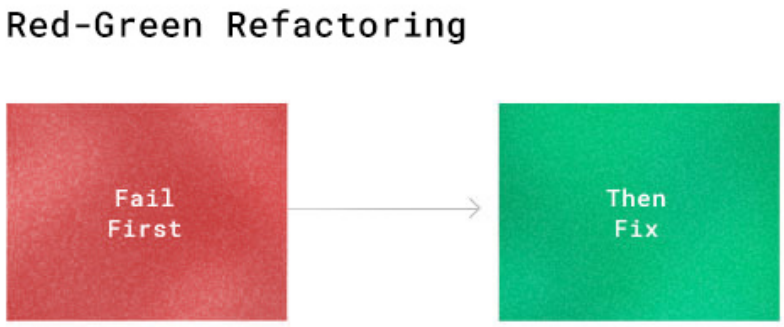
With TDD, you design your code before you implement it. Therefore, TDD forces you to think about your components' behavior before you write it. It's also a great way to keep you focused on what you are trying to deliver. In TDD, you write tests for your method or implementation to test what that implementation should do.

Remember, with TDD, your test will always fail first. You haven't written the code yet so there is no functionality. This is a good thing! It proves that your test won't just pass any old implementation!

Next, it's time for you to prove that your test will pass when the implementation is valid and it serves its purpose. Once you check that your test fails when the implementation doesn't work correctly and passes when implementation does function correctly, you can refactor your code to be better and clearer. Since you already have the test right there, refactoring will be much easier and you can do it with the assurance that your tests will tell you whether you changed the code's behavior successfully. This is called **red-green refactoring**.

Red: First, you make your test fail. **Green:** Then, make it pass.

Test first, refactor after. This ensures that your code is clean and production-ready. It's important to actually go through red and green before you make your code perfect. The red stage will verify that your test is not just going to pass all the time. You



can feel confident it's deterministic later on when things get more complex. Later, in the green stage, your focus isn't "How can I write the best code?" but rather "How can I write code that meets the requirements?" Think of this as the stage to prove that your test is passing for the valid use cases.

In the next stage, refactoring, you will have a change to revisit your code. The refactoring stage is when you can write cleaner, more intelligent code, and make improvements. Often, engineers start writing in the beginning and lose focus of what they were supposed to deliver. Other times, engineers will write tests at the same time as they are creating the implementation and create unexpected bugs. TDD helps avoid those mistakes.

The test you write might look like this:

```
describe('sum()', function () {

    it('should return the sum of given numbers', function () {
      expect(simpleCalculator.sum(1,2)).to.equal(3);
      expect(simpleCalculator.sum(5,5)).to.equal(10);
    });
  })
```

1. **Red:** Your implementation is currently empty. You haven't implemented yet, so the tests will fail. You want to verify that your test is deterministic: it will tell you when it should fail or pass.

```
var Calculator = function () {
    return true // implementation goes here
}
```

2. **Green:** Implement it. Make the test pass. Here we'll write the code that will make the test pass.

```
var Calculator = function () {
    return{
          sum: function(number1, number2){
                return number1 + number2;
          }
    };
}
```

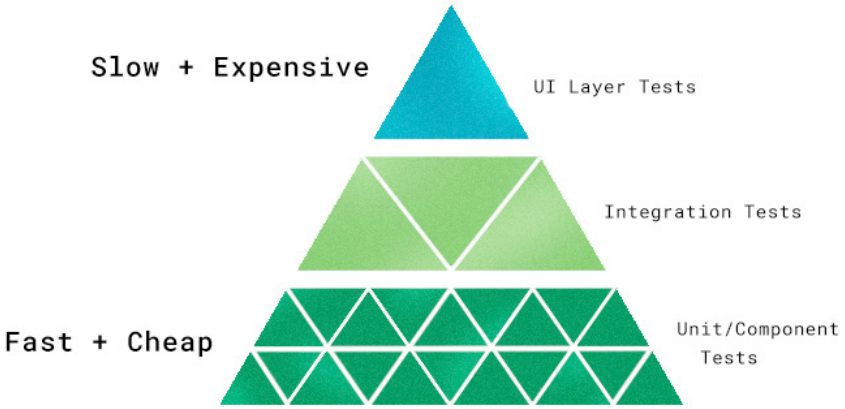
Now your test will be satisfied, because we've added the function.

Refactoring code: Now, refactor your code to be clearer and more readable. The first two steps made it so that your test is reliable and you don't have to worry about modifying your code's behavior accidentally. Remember, you verified the functionality of your code with the test that went through the red and green stages. Once the code is in the refactoring stage, the tests should not be changed. If you make a change now, you increase the likelihood of a functionality failure in your source code. If you need to change the tests, make sure to do so in the red/green stage.

Once you have a valid test, you can refactor the code to be cleaner and more aligned with the style or overall class.

Putting it all together

The example above is for a unit test. But how can we use TDD on other layers of the test pyramid?



When we make an implementation, we'll start with UI layer testing first (using BDD, which we'll explain in the next section). Even though these UI tests will not pass for a long time, starting from the tests helps us focus on what we are actually trying to build and how the backend code will interact with the frontend layer. This approach allows developers to design their implementation before they write it.

From there, we start working on unit/component testing or integration testing, depending on the work itself. If the architectural design is clear before we dig into the code base, we start writing integration tests. These will also fail for a while. While they may not be complete at the moment you write them, they still serve the function of helping you think about what you are trying to build and what the initial design is.

When we move onto the unit/component test, we finally start red-green refactoring in the unit test layer, leaving UI and integration layers in 'red' stage and completing the unit tests to the refactoring stage. Then we move back to the integration tests and make the text green, then refactor. Afterward, the same step applies to the UI testing

As you can see, we are applying the TDD principles throughout all the layers of testing. The principle is the same, the only difference is the scale.

BDD: behavior-driven development

User Journey Story and Given, When and Then

Any time there is a new feature request, people from the product side of the business write story-level tasks for engineers, including user story and user acceptance criteria. This way, you as an engineer can understand the value to the business and think from the user's perspective about the functionality that they will implement. By seeing user stories, engineers can also better understand the scope of the work.

User-acceptance testing (UI-driven testing) builds on this user acceptance criteria and user story. UI-driven testing usually uses tools like **Selenium** or **Cucumber** which help test against the user's journey on a site that's up and running.

The user journey story represents a user's behavior. Using the business requirements provided, the developer can think about scenarios of how a user will use this new functionality. And those scenarios can be used to write the tests. This is called behavior-driven development (BDD).

BDD is a widely-used method in UI-driven testing. It is written in a structure known as "Given, When and Then."

Given: the state of the system that will receive the behavior/action

When: the behavior/action that happens and causes the result in the end

Then: the result caused by the behavior in the state

It's a good idea to think about the user journey and user's behavior first, so that when you implement your feature, it is with consideration of how the user will interact with it.

Here is an example:

Scenario: the user signs up to the site

Given: the user visited the site

When: the user clicked the signup button

Then: ensure the user can access the signup page

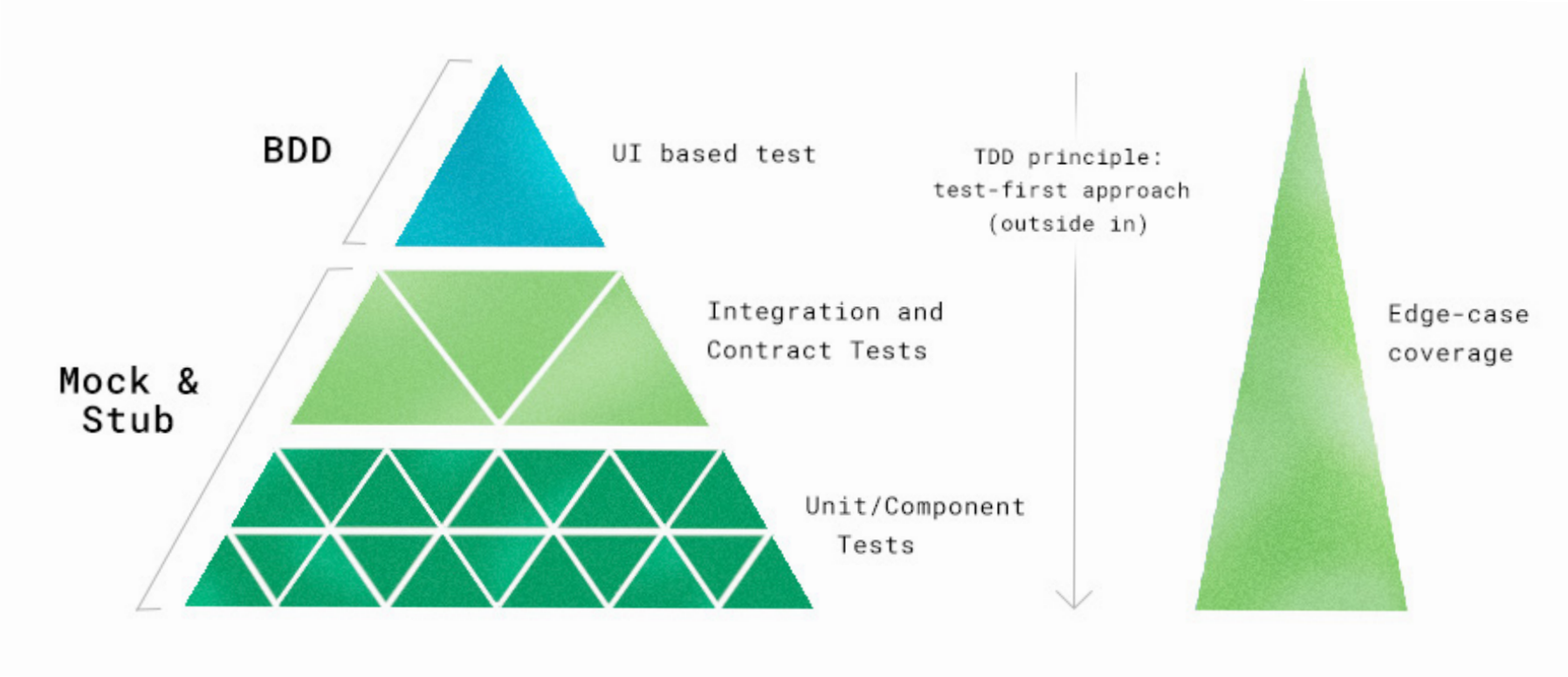
Here are some simple test code examples with **Cypress** (for an easy integration with this tool, explore the **Cypress orb**):

```
describe('User can signup to the test-example site', function() {

  it('clicking "signup" navigate to a signup url', function() {
    // Given
    cy.visit('https://test-example.com/')
    // When
    cy.contains('signup').click()
    //Then
    cy.url().should('include', '/signup')
  })
})
```

Using BDD in UI layer testing make sense since it involves the part of the application that the user will interact with. Other layers of testing won't be as well-suited to using BDD. While UI layer testing with BDD is invaluable to the process of building quality software, it's very expensive and inefficient.

As we mentioned earlier, utilizing different layers and kinds of testing means that when something goes wrong, it will be way faster to pinpoint exactly what is failing and be able to fix it more quickly. This reduces debugging time and enables you to detect low-level problems much more cheaply and quickly.



Conclusion: happy path and edge cases

When you write tests, it's easy to think about what will happen when everything goes well. It can be a challenge for engineers to think about edge cases. That is expected.

When we think about UI layer testing, we are assuming that (almost) the entire site is already up and running and your tests are running against them. Now, it's hard to imagine every single pathway that a user may take, and it would be very expensive to test every single pathway that could possibly occur. Therefore, it's a good practice to **focus on the happy path and major failure path: these will cover both the main behaviors and the worst-case scenario.** Often the edge case bugs are discovered from QA exploratory testing in a QA-like environment. In this process, QA will analyze the business risk and communicate the edge case scenario to the engineers to ensure that they fix the bugs before the code goes to production, and write new tests to cover the edge case scenarios.

When engineers understand the system and circumstances better, it becomes easier to think about edge cases. This results in the tests in the edge cases being covered better by the lower layer of the test pyramid - which is always preferable, as they're more efficient. That said, maintaining tests are also part of an engineer's job. When your code evolves, your tests need to be changed as well. Having meaningful and behavior-driven tests is more important than the number of tests that you have. The purpose of testing, after all, is to deliver quality software to production.

Making your code base more testable is a worthwhile investment, and it will help you scale your business and software in the long term. This fundamental work will allow you to optimize your software's path to production, giving you more confidence every time you deploy.

Conclusion

Learning when and how to use DevOps, and mastering the ins and outs of testing, goes a long way toward ensuring that you're delivering the highest quality software possible. It also reduces wasted time and resources, maximizing both efficiency and output. To learn more about how to make your software delivery faster and more robust, visit **circleci.com**.