**Introduction to Automated Testing**

As you add a feature to your app, how do you *know* whether the feature will work correctly, given all possible interactions? How do you know that the feature works well with and doesn't break other features? How do you know that your code is maintainable and can be easily understood by others?

You could run the app locally and try a few inputs, but that takes time and doesn't cover all cases. Plus, repeatedly testing existing features gets tedious and time consuming as you add new features.

This module demonstrates how to set up automated testing to help ensure that your latest feature works and that you haven't broken anything along the way.

In this module, you will:

* Learn the benefits of automated testing and the kinds of testing you can use.
* Run unit tests locally and then in Azure Pipelines.
* Add dashboard widgets to visualize test runs over time.
* Perform code coverage testing to see how much of your code is covered by unit tests.
* See a test fail in the pipeline and then fix the failure.

**Prerequisites**

The modules in this learning path form a progression.

To follow the progression from the beginning, be sure to first complete the Evolve your DevOps practices  learning path.

We also recommend you start at the beginning of this learning path, Build applications with Azure DevOps .

If you want to go through just this module, you need to set up a development environment on your Windows, macOS, or Linux system. You need:

* An Azure DevOps organization
* A GitHub  account
* Visual Studio Code
* .NET Core 3.1 SDK
* Git

You can get started with Microsoft Azure DevOps for free.

This environment lets you complete the exercises in this and future modules. You can also use it to apply your new skills to your own projects.

**Note**

Keep in mind that you can use Azure DevOps to build and deploy almost any kind of application written in any language. In this module, you'll be working with a .NET Core application written in C#.

You don't need to be an expert in .NET or C# to complete this module. You can apply the patterns you learn here to your own projects that use your favorite programming languages and frameworks.

**Meet the team**

You met the *Space Game* web team at Tailspin Toys in previous modules. As a refresher, here's who you'll work with in this module.



Andy is the development lead.



Amita is in QA.



Tim is in operations.



Mara just joined as a developer and reports to Andy.

Mara has prior experience with DevOps and is helping the team adopt a more streamlined process that uses Azure DevOps.

# What is automated testing?

* 12 minutes

In this unit, you learn about the benefits of automated testing and the kinds of testing you can perform. You also learn what makes a good test and be introduced to the various testing tools that are available to you.

Let's begin by checking in with the Tailspin web team.

## The need for automated testing

Mara, the newest member of the team, is slowly introducing her teammates to DevOps and Microsoft Azure DevOps. In particular, she's using Azure Boards and Azure Pipelines to get everyone started.

To begin the project, the team has identified three key needs to address:

* Stabilize the build server
* Create a Git-based workflow
* Create unit tests

So far, Mara's efforts have been one-sided. She's the only team member who's begun work on these issues, although Andy is starting to get involved. Unfortunately, he seems to spend most of his days in meetings. Amita, the tester, is busy helping other teams. Tim, from operations, hasn't seen any good reason to get involved at all.

Things are about to change. Amita is stressed and frustrated. She's stressed because she's helping other teams while working on Space Game. She's frustrated because she's seeing the same errors with the Space Game website repeatedly, particularly with the leaderboard. The errors are all regression bugs. As Andy and Mara add features to the leaderboard, the existing features break.

Mara has already taken on the first two issues and has started thinking about the third issue, creating unit tests. And it couldn't come at a better time. Amita has already pinged Andy and Mara about the problem. She wants to know if they'll work with her to improve the code quality and eliminate those bugs.

**Amita:** Thanks for meeting with me. I wanted to see if we can stop breaking the filtering feature on the leaderboard. Is there any way we can get more stability? I'm pulled in all kinds of directions right now. Reporting the same broken functionality over and over on something that should work is driving me crazy!

**Mara:** I have an idea. You've seen the automated pipeline. I think we can add unit tests to the automated build. Unit testing is about testing a single piece of functionality, like the filtering feature, in isolation. We'd catch regression errors before they ever get to you.

**Andy:** We tried unit testing a while ago but we had trouble keeping the tests up to date, so we gave up. Part of the problem was that, when the tests failed on the build server, there was no way to know about it. Plus, we're pressured for time. We'd ignore failed tests and nothing was in place to keep us honest.

**Mara:** If we add tests to the Azure Pipelines build, the build will fail if any tests fail. That should certainly keep us honest! Andy and I could do this and it will make life better for you, Amita.

**Amita:** I'm all for an easier life, but you're not going to put me out of a job, are you?

**Mara:** There's still plenty of manual testing to do. This is just one piece of the puzzle.

**Andy:** I agree. It's our job to make sure we're handing off quality code to you. I say we at least try it. Unit tests run fast. They should add only a minute to the build process, at most. Plus, there are other kinds of tests we can run as well, such as lint and code coverage tests.

Honestly, I've wanted to do this for a long time. Mara's given us a good start with the automated build pipeline. I think it's time for me to get more involved.

**Amita:** Thank you both! I can't wait to see the results.

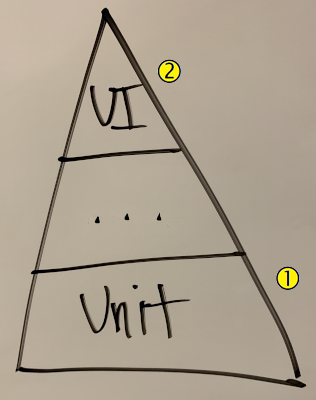
## What is automated testing?

Automated testing uses software to execute your code and compare the actual results with the results you expect. Compare this with exploratory or manual testing, where a human typically follows instructions in a test plan to verify that software functions as expected.

Manual testing has its benefits. But as your code base grows in size, testing all features manually (including edge cases) can become repetitive, tedious, and error prone. Automated testing can help eliminate some of this burden and enable manual testers to focus on what they do best: ensuring that your users will have a positive experience with your software.

### The test pyramid

When we think about automated testing, it's common to separate tests into layers. Mike Cohn proposes this concept, known as the test pyramid, in his book Succeeding with Agile.



Although this is a simplistic version of Cohn's model, the concept illustrates that you focus most of your effort on writing tests that verify the foundational levels of your software (callout 1 in the pyramid), such as functions, classes, and methods. You focus progressively less effort as features are combined, such as at the user interface (UI) layer (callout 2 in the pyramid). The idea is that if you can verify that each lower-level component works as expected in isolation, tests at the higher levels need only verify that multiple components work together to get the expected result.

### When should I write tests?

The answer mainly depends on your needs and experience in writing tests.

It's never too late to start adding tests for code you've already written and deployed. This is especially true for features that often break or require the most effort from your test team.

When you relate testing to continuous integration and continuous delivery pipelines, two concepts you'll hear about are continuous testing and shifting left.

Continuous testing means tests are run early in the development process and as every change moves through the pipeline. Shifting left means considering software quality and testing earlier in the development process.

As an example, developers often add test cases as they develop their feature and run the entire suite of tests before they submit the change to the pipeline. This approach helps ensure both that the feature they're building behaves as expected and that it doesn't break existing features.

Here's a short video where Abel Wang, Cloud Advocate at Microsoft, explains how to ensure you maintain quality in your DevOps plan.

**Ask Abel**

Shifting left often requires testers to get involved in the design process, even before any code for the feature is written. Compare this to the "handoff" model, where the test team is presented with new features to test only after the software is designed and written. A bug discovered late in the process can affect the team's delivery schedule. And bugs might be discovered weeks or even months after the developer originally built the feature.

### The tradeoff

With automated testing, there comes a tradeoff. Although automated testing enables testers to focus their time verifying the end-user experience, developers might need to dedicate more time to writing and maintaining their test code.

However, the point of automated testing is to help ensure that testers receive only the highest quality code, code that's been proven to function as expected. Therefore, developers can reclaim some of their time by having to handle fewer bugs or avoiding code rewrites because of an edge case they hadn't originally considered.

### Added benefits

Documentation and the ability to more easily refactor your code are two added benefits of automated testing.

#### Documentation

Manual test plans can serve as a type of documentation as to how software should behave and why certain features exist.

Automated tests can serve the same purpose. Automated test code often uses a human-readable format. The set of inputs you provide represent values your users might enter. Each associated output specifies the result your users should expect.

In fact, many developers follow the test-driven development, or TDD, method by writing their test code before implementing a new feature. The idea is to write a set of tests, often called specs, that initially fail. Then, the developer incrementally writes code to implement the feature until all tests pass. Not only do the specs document the requirements, but the TDD process helps ensure that only the necessary amount of code is written to implement the feature.

#### Refactoring

Say you have a large code base that you want to refactor to make certain parts run faster. How do you know that your refactoring efforts won't cause parts of your application to break?

Automated tests serve as a type of contract. That is, you specify the inputs and the expected results. When you have a set of passing tests, you're better able to experiment and refactor your code. When you make a change, all you need to do is run your tests and verify that they continue to pass. After you've met your refactoring goals, you can submit your change to the build pipeline so that everyone can benefit, but with a lower risk of something breaking.

## What types of automated testing are there?

There are many types of automated testing. Each test serves a separate purpose. For example, you might run security tests to help verify that only authorized users can access a piece of software or one of its features.

When we mention continuous integration and the build pipeline, we're typically referring to development testing. Development testing refers to tests you can run before you deploy the application to a test or production environment.

For example, lint testing, a form of static code analysis, checks your source code to determine whether it conforms to your team's style guide. Code that's formatted consistently is usually easier for everyone to read and maintain.

In this module, you work with unit testing and code coverage testing.

Unit testing verifies the most fundamental components of your program or library, such as an individual function or method. You specify one or more inputs along with the expected results. The test runner performs each test and checks to see whether the actual and expected results match.

As an example, let's say you have a function that performs an arithmetic operation that includes division. You might specify a few values that you expect your users to enter along with edge-case values such as 0 and -1. If a certain input produces an error or exception, you can verify that the function produces the same error.

Code coverage testing computes the percentage of your code that's covered by your unit tests. Code coverage testing can include conditional branches in your code to ensure that a function is completely covered.

The greater your code coverage percentage, the more confident you can be that you won't later discover a bug in code that wasn't fully tested. You don't need to reach 100 percent code coverage. In fact, when you start, you'll likely find that you have a low percentage. But that gives you a starting point from which you can add additional tests that cover problematic or frequently used code.

### Keep unit tests isolated

When you're learning about unit testing, you might hear terms such as mocks, stubs, and dependency injection.

Recall that a unit test should verify an individual function or method, and not how multiple components interact. But if you have a function that calls a database or web server, how do you handle that?

Not only does a call to an external service break isolation, but it can slow things down. If the database or web server goes down or is otherwise unavailable, the call can also disrupt your test run.

By using techniques such as mocking and dependency injection, you can create components that mimic this external functionality. You'll see an example later in this module.

Later, you can run integration tests to verify that your application works correctly with a real database or web server.

## What makes a good test?

You'll be better able to identify a good test as you gain experience writing your own tests and reading tests written by others. Here are some guidelines for getting started:

* **Don't test for the sake of testing**: Your tests should serve a purpose beyond being a checklist item to cross off. Write tests that verify that your critical code works as intended and doesn't break existing functionality.
* **Keep your tests short**: Tests should finish as quickly as possible, especially those that happen during the development and build phases. When tests are run as each change moves through the pipeline, you don't want them to be the bottleneck.
* **Ensure that your tests are repeatable**: Test runs should produce the same results each time, whether you run them on your computer, a coworker's computer, or in the build pipeline.
* **Keep your tests focused**: A common misconception is that tests are meant to cover code written by others. Ordinarily, your tests should cover only your code. For example, if you're using an open-source graphics library in your project, you don't need to test that library.
* **Choose the right granularity**: For example, if you're performing unit testing, an individual test shouldn't combine or test multiple functions or methods. Test each function separately and later write integration tests that verify that multiple components interact properly.

### What types of testing tools are available?

The testing tools you use depend on the type of application you're building and the type of testing you want to perform. For example, you can use Selenium to perform UI testing on many types of web browsers and operating systems.

No matter what language your application is written in, there's likely a number of test tools available to you.

For example, for Java applications, you might choose Checkstyle to perform lint testing and JUnit to perform unit testing.

For the Space Game website, which is a .NET Core application written in C#, Mara chooses NUnit for unit testing because it's popular in the .NET community and she and Andy have prior experience with it.