**Introduction to Functional Testing**

In this module, you'll add functional tests to the pipeline. These tests verify an application's behavior.

In the Manage database changes in Azure Pipelines  module, you helped the Tailspin Toys web team connect their web application to Azure SQL Database. You also used a release approval to help the developers and the database administrator manage changes to the database schema.

The stages that you and the team defined provide the overall shape of your pipeline. But you can add more to each stage. For example, in the *Test* stage, Amita still tests the web application manually as she always has. When she's satisfied, she manually promotes the application to *Staging*. In *Staging*, management reviews the new features and decides whether to make the release publicly available.

In the Run quality tests in your build pipeline using Azure Pipelines  module, you incorporated unit and code coverage tests into the build process. These tests help avoid regression bugs and ensure that the code meets the company's standards for quality and style. But what kinds of tests can you run after a service is operational and deployed to an environment?

**Learning objectives**

In this module, you will:

* Define functional tests and identify some popular kinds of tests.
* Map manual testing steps to automated test cases.
* Run automated UI tests locally and in the pipeline by using Selenium.

**Prerequisites**

The modules in this learning path form a progression. To follow the progression from the beginning, complete these learning paths first:

* Evolve your DevOps practices
* Build applications with Azure DevOps

We also recommend that you start at the beginning of the Deploy applications with Azure DevOps  learning path.

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 these assets:

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

You can get started with Azure and Azure DevOps for free. You don't need an Azure subscription to work with Azure DevOps, but here you'll use Azure DevOps to deploy to resources that exist in your Azure subscription.

This environment lets you complete the exercises in this module 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. She's helping the team adopt a more automated process that uses Azure DevOps.

# What is functional testing?

* 10 minutes

In this section, you join the Tailspin team as they define functional tests for their pipeline. Functional tests verify that each function of the software does what it should.

The team first defines what a functional test covers. They explore some types of functional tests. Then they decide on the first test to add to their pipeline.

## Weekly meeting

The team is having their weekly meeting. Andy is demonstrating the release pipeline. The team watches a successful build move through the pipeline, from one stage to another. Finally, the web app is promoted to Staging.

**Amita:** I'm so happy with the pipeline. It makes my life much easier. For one thing, I automatically get a release deployed to the **test** environment. That means I don't have to manually download and install build artifacts on my test servers. That's a significant time saver.

Also, the unit tests that Mara and Andy wrote eliminate all the regression bugs before I get the release. That removes a major source of frustration. I don't spend time finding and documenting regression bugs.

But I'm worried that all of my testing is still manual. The process is slow, and we can't show anything to management until I finish. It's hard because the testing is important. Testing ensures that the users get the right experience. But the pressure is on to deliver faster.

**Andy:** I'm sure we can help you. What kind of tests take up most of your time?

**Amita:** I think the UI tests do. I have to click through every step to make sure I get the correct result. And I have to do that for every browser we support. It's very time consuming. And as the website grows in complexity, UI testing won't be practical in the long run.

**Mara:** UI tests are considered to be functional tests.

**Tim:** As opposed to what, nonfunctional tests?

**Mara:** Exactly. And nonfunctional tests are something that you, in particular, care about.

**Tim:** Okay, I'm confused.

## What are functional and nonfunctional tests?

**Mara:** Functional tests verify that each function of the software does what it should. How the software implements each function isn't important in these tests. What's important is that the software behaves correctly. You provide an input and check that the output is what you expect. That's how Amita tests the UI. For example, if she selects the top player on the leaderboard, she expects to see that player's profile.

Nonfunctional tests check characteristics like performance and reliability. An example of a nonfunctional test is checking to see how many people can sign in to the app simultaneously. Load testing is another example of a nonfunctional test. Those performance and reliability concerns are things you care about, Tim.

**Tim:** They are, indeed. I need to think about this for a bit. I might want to add some automation to the pipeline too, but I'm not sure what I want to do. What kinds of automated tests can I run?

**Mara:** For now, let's focus on functional testing. It's the kind of testing that Amita does. And it sounds like an area where we want to improve.

## What kinds of functional tests can I run?

There are many kinds of functional tests. They vary by the functionality that you need to test and the time or effort that they typically require to run.

The following sections present some commonly used functional tests.

### Smoke testing

Smoke testing verifies the most basic functionality of your application or service. These tests are often run before more complete and exhaustive tests. Smoke tests should run quickly.

For example, say you're developing a website. Your smoke test might use curl to verify that the site is reachable and that fetching the home page produces a 200 (OK) HTTP status. If fetching the home page produces another status code, such as 404 (Not Found) or 500 (Internal Server Error), you know that the website isn't working. You also know that there's no reason to run other tests. Instead, you diagnose the error, fix it, and restart your tests.

### Unit testing

You worked with unit tests in the Run quality tests in your build pipeline using Azure Pipelines  module.

In short, 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 results match the expected results.

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. You also specify edge-case values such as 0 and -1. If you expect a certain input to produce an error or exception, you can verify that the function produces that error.

The UI tests that you'll run later in this module are unit tests.

### Integration testing

Integration testing verifies that multiple software components work together to form a complete system. For example, an e-commerce system might include a website, a products database, and a payment system. You might write an integration test that adds items to the shopping cart and then purchases the items. The test verifies that the web application can connect to the products database and then fulfill the order.

You can combine unit tests and integration tests to create a layered testing strategy. For example, you might run unit tests on each of your components before you run the integration tests. If all unit tests pass, you can move on to the integration tests with greater confidence.

### Regression testing

A regression occurs when existing behavior either changes or breaks after you add or change a feature. Regression testing helps determine whether code, configuration, or other changes affect the software's overall behavior.

Regression testing is important because a change in one component can affect the behavior of another component. For example, say you optimize a database for write performance. The read performance of that database, which is handled by another component, might unexpectedly drop. The drop in read performance is a regression.

You can use various strategies to test for regression. These strategies typically vary by the number of tests you run to verify that a new feature or bug fix doesn't break existing functionality. However, when you automate the tests, regression testing might involve just running all unit tests and integration tests each time the software changes.

### Sanity testing

Sanity testing involves testing each major component of a piece of software to verify that the software appears to be working and can undergo more thorough testing. You can think of sanity tests as being less thorough than regression tests or unit tests. But sanity tests are broader than smoke tests.

Although sanity testing can be automated, it's often done manually in response to a feature change or a bug fix. For example, a software tester who is validating a bug fix might also verify that other features are working by entering some typical values. If the software appears to be working as expected, it can then go through a more thorough test pass.

### User interface testing

User interface (UI) testing verifies the behavior of an application's user interface. UI tests help verify that the sequence, or order, of user interactions leads to the expected result. These tests also help verify that input devices, such as the keyboard or mouse, affect the user interface properly. You can run UI tests to verify the behavior of a native Windows, macOS, or Linux application. Or you can use UI tests to verify that the UI behaves as expected across web browsers.

A unit test or integration test might verify that the UI receives data correctly. But UI testing helps verify that the user interface displays correctly and that the result functions as expected for the user.

For example, a UI test might verify that the correct animation appears in response to a button click. A second test might verify that the same animation appears correctly when the window is resized.

In this module, you work with UI tests that are coded by hand. But you can also use a capture-and-replay system to automatically build your UI tests.

### Usability testing

Usability testing is a form of manual testing that verifies an application's behavior from the user's perspective. Usability testing is typically done by the team that builds the software.

Whereas UI testing focuses on whether a feature behaves as expected, usability testing helps verify that the software is intuitive and meets the user's needs. In other words, usability testing helps verify whether the software is "usable."

For example, say you have a website that includes a link to the user's profile. A UI test can verify that the link is present and that it brings up the user's profile when the link is clicked. However, if humans can't easily locate this link, they might become frustrated when they try to access their profile.

### User acceptance testing

User acceptance testing (UAT), like usability testing, focuses on an application's behavior from the user's perspective. Unlike acceptance testing, UAT is typically done by real end users.

Depending on the software, end users might be asked to complete specific tasks. Or they might be allowed to explore the software without following any specific guidelines. For custom software, UAT typically happens directly with the client. For more general-purpose software, teams might run beta tests. In beta tests, users from different geographic regions or users who have certain interests receive early access to the software.

Feedback from testers can be direct or indirect. Direct feedback might come in the form of verbal comments. Indirect feedback can come in the form of measuring testers' body language, eye movements, or the time they take to complete certain tasks.

We've already covered the importance of writing tests. But just to emphasize it, here's a short video where Abel Wang, Cloud Advocate at Microsoft, explains how to help ensure quality in your DevOps plan.

**What does the team choose?**

**Tim:** All of these tests sound important. Which should we tackle first?

**Andy:** We already have working unit tests. We're not yet ready to perform user acceptance testing. Based on what I hear, I think we should focus on UI testing. Right now, it's the slowest part of our process. Amita, do you agree?

**Amita:** Yes, I do agree. We still have some time left in this meeting. Andy or Mara, do you want to help me plan an automated UI test?

**Mara:** Absolutely. But let's get a few preliminaries out of the way. I'd like to discuss what tool we should use and how we'll run the tests.

**What tools can I use to write UI tests?**

**Mara:** When it comes to writing UI tests, what are our options? I know there are many. Some tools are open source. Others offer paid commercial support. Here are a few options that come to mind:

* **Windows Application Driver** (WinAppDriver): WinAppDriver helps you automate UI tests on Windows apps. These apps can be written in Universal Windows Platform (UWP) or Windows Forms (WinForms). We need a solution that works in a browser.
* **Selenium**: Selenium is a portable open-source software-testing framework for web applications. It runs on most operating systems, and it supports all modern browsers. You can write Selenium tests in several programming languages, including C#. In fact, there are NuGet packages that make it easy to run Selenium as NUnit tests. We already use NUnit for our unit tests.
* **SpecFlow**: SpecFlow is for .NET projects. It's inspired by a tool called Cucumber. Both SpecFlow and Cucumber support behavior-driven development (BDD). BDD uses a natural-language parser called Gherkin to help both technical teams and nontechnical teams define business rules and requirements. You can combine either SpecFlow or Cucumber with Selenium to build UI tests.

Andy looks at Amita.

**Andy:** I know these options are new to you, so do you mind if we pick Selenium? I have some experience with it, and it supports languages I already know. Selenium also will give us automatic support for multiple browsers.

**Amita:** Sure. It's better if one of us has some experience.

**How do I run functional tests in the pipeline?**

In Azure Pipelines, you run functional tests just like you run any other process or test. Ask yourself:

* In which stage will the tests run?
* On what system will the tests run? Will they run on the agent or on the infrastructure that hosts the application?

Let's join the team as they answer these questions.

**Mara:** One thing I'm excited about is that now we can test in an environment that's like production, where the app is actually running. Functional tests like UI tests make sense in that context. We can run them in the *Test* stage of our pipeline.

**Amita:** I agree. We can maintain the same workflow if we run automated UI tests in the same stage in which I run manual tests. Automated tests will save us time and enable me to focus on usability.

**Tim:** Amita tests the website from her Windows laptop because that's how most of our users visit the site. But we build on Linux and then deploy Azure App Service on Linux. How do we handle that?

**Mara:** Great question. We also have a choice about where we run the tests. We can run them:

* On the agent: either a Microsoft agent or an agent that we host.
* On test infrastructure: either on-premises or in the cloud.

Our existing *Test* stage includes one job. That job deploys the website to App Service from a Linux agent. We can add a second job that runs the UI tests from a Windows agent. The Windows agent that's hosted by Microsoft is already set up to run Selenium tests.

**Andy:** Again, let's stick with what we know. Let's use a Microsoft-hosted Windows agent. Later, we can run the same tests from agents that run macOS and Linux if we need additional test coverage.

**The plan**

**Mara:** OK. Here's what we're going to do:

* Run Selenium UI tests from a Microsoft-hosted Windows agent.
* Have the tests fetch the web content from the app that's running on App Service, in the *Test* stage.
* Run the tests on all the browsers that we support.

**Andy:** I'll work with Amita on this one. Amita, let's meet tomorrow morning. I'd like to do a bit of research before we meet.

**Amita:** Great! See you then.

**Create a functional test plan**

We've just seen the team decide on how they'll implement their first functional tests. If your team is just starting to incorporate functional tests into their pipeline (or even if you're already doing that), remember that you always need a plan.

Many times, when someone asks team members about their performance testing plan, it's common for them to respond with a list of tools they are going to use. However, a list of tools isn't a plan. You also must work out how the testing environments will be configured, you need to determine the processes to be used, and you need to determine what success or failure looks like.

Make sure your plan:

* Takes the expectations of the business into account.
* Takes the expectations of the target users into account.
* Defines the metrics you will use.
* Defines the KPIs you will use.

Performance testing needs to be part of your planning, right from the start. If you use a story or Kanban board, you might consider having an area near it where you can plan out your testing strategy. As part of the iteration planning, gaps in the testing strategy should be highlighted. It is also important to work out how you will monitor performance once the application has been deployed, and not just measure performance before it's released.