9-1. Using Testing to Improve the Quality of Your ASP.NET MVC Application

Problem

You are part of an organization that does not follow a formal strategy for testing your applications. In most cases, testing is performed by a developer clicking through the completed application and verifying that it works as expected. In some cases, you may have a dedicated QA that does this basic click-through testing for you.

Unfortunately, you have found that this strategy fails on several levels. First of all, it relies on a completely manual process that is very time-consuming and may only cover a small percentage of your application’s code. This process leaves many potential bugs to be discovered by your end users. Second, in most cases, you are finding problems late in the development cycle, which makes them more difficult and expensive to diagnose, correct, and deploy. Finally, this strategy significantly increases the risk and complexity of implementing subsequent releases because of the requirement for an end-to-end manual regression testing.

Solution

The cost of bugs that make their way into production code is difficult to measure. This is especially true if a bug (or a combination of them) results in lost customers and decreased sales. While it may not be possible to prevent all defects from reaching production, you can significantly improve the quality of your application by implementing a comprehensive quality control strategy. This strategy should consist of unit tests, static code analysis, peer code reviews, integration tests, and performance tests, topped off with manual testing and a customer feedback program.

How It Works

Successful quality assurance (QA) strategies should focus on finding the problems in your application as early in the development process as possible. Some even argue that QA should begin in the requirements gathering phase, where poorly thought through and even contradictory requirements can be corrected or eliminated before making their way to the architects canvas. The architect and technical lead are the next line of defense. They should run each of the use cases defined in the requirements documentation through their design, and ensure that the design can meet the needs of the application. They are also responsible for ensuring that the application is designed in such a way that it can be easily tested.
Assuming that the business analysts, architects, and technical leads have done their jobs properly, most of the responsibility for application quality falls squarely on the shoulders of the developers. The developer is responsible for writing quality code that meets the standards of the organization, writing unit tests that cover as much of the code as possible, and alerting the technical lead to possible flaws in the design. The developer is also responsible for identifying parts of the application that may throw exceptions, and catching and logging the exception appropriately.

Unit Tests

A unit test is a simple program that allows you to test specific sections of code independently. Unit tests should be isolated, fast, and test a very specific test case. They should be executed at the click of a button without any additional input from the tester. They should also support being run automatically, either alone or as part of a set of tests. These automatable unit tests can then be used as part of a nightly build or continuous integration process. If done properly, unit tests should provide coverage of the majority of your production code. The amount of your production code covered by unit tests is known as its code coverage percentage.

Unit testing is not a new concept. It has been around as long as people have been developing component-based applications. Before the popularization of unit testing frameworks, most unit tests were either created as either command-line applications or simple user interfaces. Today most unit tests are written using frameworks that reduce the amount of code that needs to be written for each test and provide APIs to simplify automation of test execution and results aggregation. These frameworks are often bundled with tools called test runners that aid in automating test execution and examining the test results.

Popular unit testing frameworks include the following:

- **MS Test**: The testing framework included as part of Microsoft’s Visual Studio Quality Tools framework. It is included with all versions of Visual Studio 2012.
- **NUnit**: Originally created as a port of the Java-based JUnit framework, NUnit has become extremely popular and is now used by most .NET developers who employ test-first development. It also has a rich ecosystem of add-ons and is compatible with many third-party test runners.
- **xUnit.net**: Created by the original developer of NUnit, xUnit was specifically designed around the concept of test-driven development (TDD) and works with several test runner frameworks, including ReSharper, CodeRush, and TestDriven.NET.
- **MbUnit**: A unit-testing framework for .NET built on top of the Gallio Automation Platform.

All these tools offer similar sets of core capabilities but differ on implementation details and style. If you would like to learn more about each of these tools, you can refer to Recipe 9-5, “Selecting a Unit Test Framework.”

Visual Studio offers an integrated test runner that can be used with many of the aforementioned frameworks.

Code Coverage Tools

A code coverage tool is a software component that can examine a set of unit tests and determine how much of the code under test (CUT) is executed by the tests. Some tools, such as the one built into Visual Studio, can identify production code that is not covered by unit tests and provide visual feedback to the developer by highlighting the uncovered code in the editor.

Microsoft includes an integrated code coverage tool with the Visual Studio Premium and Ultimate editions. If you are not using these top-tier editions of Visual Studio, several third-party code coverage tools are available for .NET developers. They include JetBrains dotCover, NCrunch, and NCover.