Issue 28 | Jan-Feb 2017

DNCMagazine

www.dotnetcurry.com

VISUAL STUDIO CODE Condensed

DockNetFiddle using Docker and .NET Core

Automated CI/CD for your Xamarin Apps

Coding Guidelines

ASP.NET Core with Antiforgery

AUTOMATED TESTING using Composition Root

Agile Development Best Practices Using Visual Studio 2015

Migrating from Bootstrap 3 to 4

CONTENTS

THE TEAM

Editor In Chief
Suprotim Agarwal
suprotimagarwal@dotnetcurry.com

Art Director
Minal Agarwal

Contributing Authors
Benjamin Jakobus
Craig Berntson
Damir Arh
Daniel Jimenez Garcia
Gouri Sohoni
Gerald Versluis
Yacoub Massad

Technical Reviewers
Damir Arh
Daniel Jimenez Garcia
Gil Fink
Subodh Sohoni
Suprotim Agarwal
Swaminathan Vetri

Next Edition
March 2017

www.dotnetcurry.com/magazine

EDITOR'S NOTE

Any developer. Any app. Any platform.

In today's software ecosystem, change is disruptive and occurs more frequently than ever. Microsoft has changed a lot in the past few years, and for better. It is currently a leading open source contributor on GitHub. Surprised? You shouldn't be.

This focus on the cross-platform, collaborative, open source world did not happen overnight. It's been a few years in the making. Microsoft raised a few eyebrows in 2008 when it started contributing to Hadoop, followed by embracing Linux in Azure in 2012. In 2014, Microsoft made .NET open source, and cross platform. Visual Studio Code, Chakra, PowerShell, Docker integration, SQL Server on Linux and close to two hundred other open source initiatives are a testimony to the fact that Microsoft has aligned itself with the changing need of today's ecosystem. Developers, Developers, and Developers - it's time we too reinvent ourselves!

Yes, Pigs can fly!

Suprotim Agarwal
Editor in Chief

Yacoub Massad

AUTOMATED TESTING USING THE COMPOSITION ROOT

This article describes how to do unit and integration testing using the Composition Root as the source of Systems Under Test (SUTs).

Introduction

Software testing is a very important part of any good software development process. As developers, we should test our software to gain confidence that once deployed in production. the software will work as expected,.

Automated tests

An important type of software testing is **Automated Testing**. As opposed to manually running the software to verify its behavior, automated testing is about writing special testing code that runs some part (called the System Under Test (SUT)) of the software, collects results, and compares such results with predefined expected results. This helps reduce the amount of time needed to test the software, as testing needs to be done periodically. It also gives us more confidence that while modifying the software (to refactor or change behavior), we won't introduce regressions.

We can further classify automated testing into two types: unit testing and integration testing.

While the community does not completely agree on the difference between these two, a good definition can be found in the second edition of The Art of Unit Testing book by Roy Osherove.

Basically, a **unit test** tests a single unit of work in isolation. This unit of work can span a single method, a single class, or multiple classes. Moreover, we cannot use databases, web services, the file system, etc. in unit tests. Unit tests also need to be deterministic and has to run fast.

To be able to test a single unit in isolation, we need to replace any dependencies that it might have with fake objects (more on this in the next section).

An **Integration test**, on the other hand, can include databases, web services etc., and therefore is allowed to be slower than unit tests.

There is more to these definitions and to the differencea between these two types of tests. You can find more information at the web site of the book here: http://artofunittesting.com/definition-of-a-unit-test/

Fakes

Fakes are objects with fake behavior that we can use to substitute real dependencies in order to isolate the unit under test.

For example, consider a class that does some financial calculations, say XYZFinancialCalculator, and that has a dependency on a currency exchange rate provider dependency. Such dependency would be injected into this object via an interface called something like ICurrencyExchangeRateProvider. In production code, there would be an implementation of this interface called WebServiceBasedCurrencyExchangeRateProvider that would communicate with a commercial currency exchange rate web service to obtain exchange rates. If we want to test XYZFinancialCalculator, then we wouldn't want to use the real exchange rate provider because of the following reasons:

- The values returned by the real service are unpredictable which makes the behavior of the SUT itself unpredictable.
- It is slow to speak to a real web service which would increase the test execution time.
- It might cost money to communicate with a commercial service.
- We could get a false positive if there is a problem with the commercial service itself.

Instead, we can create a fake object that implements the ICurrencyExchangeRateProvider interface and inject it into an instance of the XYZFinancialCalculator class. This fake object would be predefined with a list of currency exchange rates.

Fake objects can be created manually or they can be created using an isolation framework. Creating a fake manually means that we create a class that implements the interface we want to fake, and that we write code in this class that represents the fake behavior. Here is how a manually created fake of ICurrencyExchangeRateProvider might look like:

```
public class FakeCurrencyExchangeRateProvider : ICurrencyExchangeRateProvider
{
  private double usdToEuroRate;

  public FakeCurrencyExchangeRateProvider(double usdToEuroRate) {
    this.usdToEuroRate = usdToEuroRate;
  }
```

```
  public double? GetExchangeRate(Currency from, Currency to)
  {
    if(from == Currency.USD && to == Currency.EURO)
      return usdToEuroRate;

    return null;
  }
}
```

We can create an instance of this class and provide a value of USD to EURO exchange rate that we want this fake object to return:

```
var fakeExchangeRateService = new FakeCurrencyExchangeRateProvider(0.94);
```

With isolation frameworks, we can create a fake object without defining a new class. Here is an example that uses FakeItEasy to create an equivalent fake object:

```
var fakeExchangeRateService = A.Fake<ICurrencyExchangeRateProvider>();

A.CallTo(() => fakeExchangeRateService.GetExchangeRate(Currency.USD, Currency.
EURO)).Returns(0.94);
```

Two types of fakes

We can classify fakes into two types: stubs and mocks. Again, there does not seem to be a complete agreement in the community on the difference between the two. So, I am going to rely mainly on the definition from The Art of Unit Testing book I mentioned earlier.

Basically, a **stub** is used to replace a dependency of the unit under test, so that the test can pass without problems. When we use stubs, we make no assertions (in the Assert phase of the test) that the stub was invoked, or not invoked, in any way. We can however configure the stub to act in a particular way. For example, we can create a stub of ICurrencyExchangeRateProvider that returns a rate of 0.94 when it is asked for the exchange rate from US dollars to Euros. However, in the Assert phase of the test, we don't check whether the SUT communicated with the stub.

Mocks also act like stubs. They substitute the dependencies of the SUT. However, in the Assert phase of the test, we use mocks to verify that it was invoked in a certain way.

One way to think about stubs and mocks is to think that stubs are used to substitute dependencies that represent queries; and mocks are used to substitute dependencies that represent commands. If we have a dependency that we use to obtain/query data (e.g. exchange rates, or customer information, etc.), we should only care about replacing this dependency so that we control which data it provides - so we use a stub. On the other hand, if we have a dependency that is used to create some side effect (write customer information to the database, writing a message to the queue, etc.), we probably need to verify that such side effect has actually taken place, so we use a mock.

Levels of testing

We can write our unit and integration tests on different levels. For example, the SUT of a unit test can be a single object, an object graph of any size that represents some component in the system, or the whole application (without the real dependencies).

When we do integration testing, we can also test a single class with an external dependency (e.g. database) or a component consisting of an object graph of any size that also has external dependencies, or even the whole application (this is sometimes called end-to-end testing).

Based on this, a test has the *highest level,* if it has the whole application as the SUT; and the *lowest level,* if it tests a single method.

Lower level tests are usually easier to create, and they run faster than higher level tests. However, higher level tests are less likely to break when we refactor the internal structure of our application.

The SOLID principles

The SOLID principles are a set of five principles that aim at creating software that is flexible and maintainable. When we apply these principles, our code base becomes a set of small classes that each has a single responsibility. These small classes are highly composable which makes it easy for us to compose them together to achieve the complex behavior that we want for the application.

Classes become composable by having their dependencies injected into them instead of them creating these dependencies, and by having dependencies on abstract types or interfaces, instead of concrete classes. This is what Dependency Injection (DI) is all about.

For an example on how to compose objects, please refer to the Object Composition with SOLID article.

The Composition Root

The Composition Root is the piece of code in an application that creates instances of the application classes and connects them to each other to make the object graph that formulates the application.

In the article, Clean Composition Roots with Pure DI, I described how we can use Pure DI to create Composition Roots that are maintainable and navigable. Pure DI is simply the application of Dependency Injection without the use of a DI container. If you haven't read that article, please go ahead and read it because this article builds on the ideas explained in that article.

Here are some properties of a clean Composition Root:

1. The Composition Root is partitioned into many "Create" methods, each responsible for creating some component of the application.

2. In order to create a component, a method can new up class instances, call other methods that create subcomponents, and then wire all of these together.

3. The Composition Root is *navigable*. In order to understand the application, a programmer can go to the Composition Root, and in the first method, know what the highest level components of the application are. This method will include method calls to create these components. The programmer can navigate to any of these methods to understand the subcomponents of the corresponding component. From there, the programmer can do the same thing to understand lower-level components. This navigation process can continue until the programmer gets to the lowest-level components.

4. The Composition Root is concrete. Although the individual classes in the application are generic in

nature (i.e., they are reusable, configurable, and context independent), the Composition Root itself is specific. It is not reusable and it sets up the context for the objects that formulate the application's object graph. This means that when we name a method in the Composition Root, we don't use generic names as we do in our classes. We instead use specific names.

For example, I would expect to see a method in the Composition Root called CreateMessageQueueProcessorThatWritesSubscriptionRequestsToTheDatabase. Such a method might create an instance of a general-purpose MessageQueueReader class, an instance of a general purpose MessageDeserializer<T> class, an instance of the SubscriptionRequestRepository class, and so on.

Consider the following figure. It shows six composable classes. Arrows going outwards represent declared dependencies of the class. The arrow going inward represent the interface this class implements. We can basically use these arrows to compose objects together.

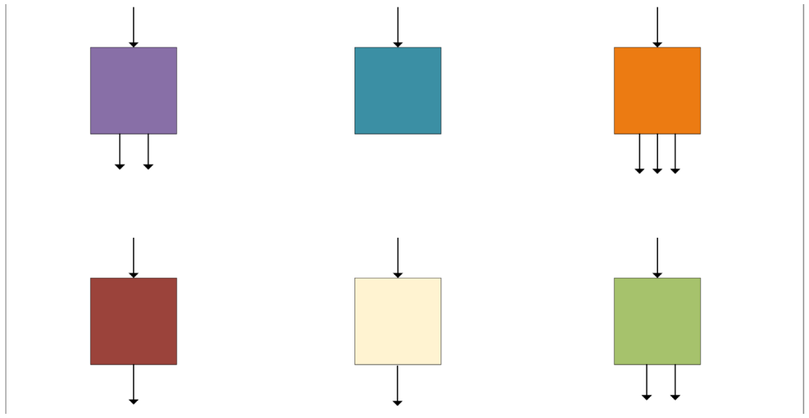


Figure 1: A representation of six composable classes

The next figure shows how a Composition Root is structured to create the application out of instances of these classes. It also shows how the Composition Root is partitioned into multiple "Create" methods.

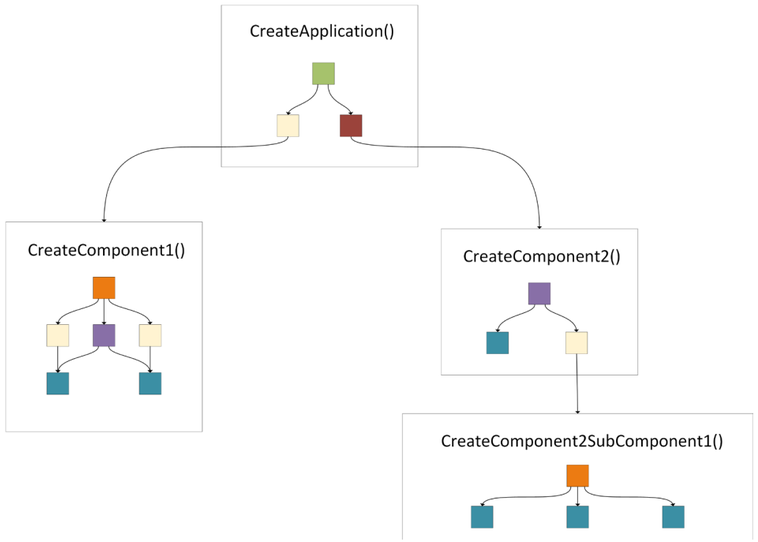


Figure 2: A representation of the "Create" methods of some Composition Root.

This is a simple example that involves six classes only, and four methods in the Composition Root. A large application could have much larger number of classes, and a much larger number of methods in the Composition Root.

The Composition Root as a source of SUTs

In many cases, a programmer writes unit tests that have only a single object as the SUT. This can be done easily by creating an instance of a class, passing fake objects for any dependencies that it might have, executing some public method on the object, and then verifying that the output is as expected or that a certain side effect has occurred (usually via a mock).

Some classes are a good fit for single-object testing. Classes that contain a good amount of logic, and none or a small amount of interaction with other objects, are an example of such classes. For example, a class that generates prime numbers is a good candidate as a SUT of a single-object unit test.

However in other cases, a group of objects in an object-graph that represents some component, is better tested as a single unit instead of testing each individual object in the graph.

For example, you might want to test that subscription requests written to a message queue will find their way to the database. There might be many objects in the application involved in making this happen. In this case, it makes sense to test all these objects as a single unit.

Using "Create" methods to create the SUT

The methods in the Composition Root create object-graphs that represent concrete application components at different levels. By having these methods, we have actually sliced the application into many cohesive pieces. Many of these pieces represent units that are attractive to test.

Instead of creating such an object graph manually in an automated test, we can invoke a "Create" method in the Composition Root to create the object graph for us. Here is how such an automated test would look like:

```
[TestMethod]
public void SubscriptionRequestsWrittenToTheQueueWillFindTheirWayToTheDatabase()
{
  //Arrange
  var sut =
    CompositionRoot
    .CreateMessageQueueProcessorThatWritesSubscriptionRequestsToTheDatabase();

  AddSubscriptionMessagesToTheQueue();

  //Act
  sut.ProcessQueueMessages();

  //Assert
  AssertThatSubscriptionMessagesWereWrittenToTheDatabase();
}
```

There are some advantages of this approach:

- It helps us test the application behavior accurately. If we create the object-graph manually in the test, it is possible that with time, the real object graph in the Composition Root will divert from the one manually

created in the test itself. For example, this can happen if the programmer decided to modify the real object graph in the Composition Root to extend the behavior of the component. The programmer might add a decorator to introduce caching. The test with the manually created object-graph would still test the component without caching, and there might be a bug in the caching logic or the way it was composed into the component under test, and such test would not catch the bug.

- It helps us follow the Don't Repeat Yourself (DRY) principle. Instead of duplicating the code that creates application components in the tests, we reuse the existing code in the Composition Root.

- It helps us create tests in the concrete language of the application instead of the generic language of individual classes.

Isolation

When creating the SUT object-graph manually in the test, we can easily isolate it by injecting fake objects into the right locations when we create the object-graph. On the other hand, when we call a "Create" method in the Composition Root, it will create real dependencies directly or indirectly via sub "Create" methods. **How do we isolate the SUT from dependencies in this case?**

One solution would be to make the "Create" methods take their dependencies as parameters instead of creating them via sub "Create" methods. This however, will sacrifice the navigability and readability of the Composition Root which are much more important than testability. I recommend against this approach.

As described in the "Code sharing versus Resource/state sharing" section of the Clean Composition Roots with Pure DI article, one exceptional case where it is OK to inject dependencies into "Create" methods is when injecting stateful objects, i.e., objects that hold state.

So, if in general we shouldn't inject dependencies into "Create" methods, what is the solution then?

Using "Create" methods as isolation boundaries

Like we used "Create" methods to create SUTs, we can use them as isolation boundaries. For example, if we are using the "CreateComponent2" method (see Figure 2) to create the SUT, we can use isolation frameworks to make the "CreateComponent2SubComponent1" method return a fake object instead of the real sub component. This way, we can test Component2 in isolation of its SubComponent1.
To make this work, we first need to make sure that the Composition Root is a normal (non-static) class. In this case, all "Create" methods are instance methods.

Next, we need to make the "Create" methods that we would like to replace, e.g. "CreateComponent2SubComponent1", both *public* and *virtual*. The virtual modifier is needed so that isolation frameworks can override these method. The public modifier is needed to make it convenient to use these methods in a lambda expression when we tell the isolation framework which method we want to configure.

Next, we use an isolation framework to create an instance of the Composition Root class in a way that allows us to override the "Create" methods that we want to return fake objects (e.g. "CreateComponent2SubComponent1"), while keeping the other methods (e.g. "CreateComponent2") intact.

The following example shows how the Composition Root for a Console Application would look like:

```
class Program
{
  static void Main(string[] args)
  {
    new CompositionRoot().Run();
  }
}

public class CompositionRoot
{
  public void Run()
  {
    CreateApplication().Run();
  }

  public virtual ISomeAppInterface CreateApplication()
  {
    //..
  }

  //Other "Create" methods here
}
```

Here is how an automated test would look like (using FakeItEasy):

```
[TestMethod]
public void SomeTestMethod()
{
  //Arrange
  var compositionRoot = A.Fake<CompositionRoot>(options => options.
  CallsBaseMethods()); //By default, use the real methods of the Composition Root

  var component2SubComponent1Stub = A.Fake<ISomeInterface>(); //ISomeInterface is
  the return type of CreateComponent2SubComponent1

  A.CallTo(() => component2SubComponent1Stub.QuerySomeSubComponentData())
    .Returns("Some fake data");

    //Make CreateComponent2SubComponent1 return a fake sub-component
    //The CreateComponent2SubComponent1 method is public
    //which makes it convenient to use it to tell FakeItEasy
    //which method we want to replace via a lambda expression
    A.CallTo(() => compositionRoot.CreateComponent2SubComponent1())
      .Returns(component2SubComponent1Stub);

    var sut = compositionRoot.CreateComponent2();

    //Act
    var result = sut.QuerySomeComponent2Data();

    //Assert
    Assert.AreEqual("Some expected data", result);
}
```

The call to "CreateComponent2" would call the real "CreateComponent2" method. But when the "CreateComponent2SubComponent1" method is called from within the "CreateComponent2" method (see Figure 2), FakeItEasy kicks in and returns the fake *component2SubComponent1Stub* object.

To make this work, we need to make sure that the return type of the "CreateComponent2SubComponent1"

method (and all methods that we want to replace), is abstract (e.g. an interface type). This is needed because standard isolation frameworks can fake abstract types only.

Furthermore, the CreateComponent2 method has to be made public so that we can use it from inside the test to create the SUT. So in general, all methods that we need to use to create the SUTs, must be public.

Please note that the example above doesn't really look nice because I wanted to show all the steps needed. We can easily extract the code that creates the SUT into its own method to make the test more readable.

There is another type of isolation frameworks that allows us to do the same thing we did earlier, but without the requirement of making the "Create" methods that we want to replace non-static, virtual, and public. I don't see this requirement as a problem, but I am going to show this alternative for completeness sake. Also, it is nice to learn about these frameworks since they can help with unit testing existing applications that are not designed with unit testing in mind.

.NET profiling API based isolation frameworks

Standard isolation frameworks, e.g. FakeItEasy, create fakes by creating new classes at runtime that implement the desired interfaces, and that behave in the configured way. The unit to be isolated should be designed to accept a fake object, e.g. by receiving it as a dependency via the constructor, or as we did earlier, by having a virtual method that creates the dependency (so that we can override it and make it return a fake object). In other words, the ability for isolation should be part of the design.

There exists another set of isolation frameworks that allows us to substitute the behavior of methods without the need for designing for isolation required by standard isolation frameworks. These frameworks can be used to modify the behavior of any method, even if it is private or static.

How do these frameworks work?

In the .NET framework SDK, there exists an API called the profiling API. This API is designed to allow us to create unmanaged programs that monitor CLR-based programs. One thing that can be done using this API is to replace the IL code of a method before it is JIT compiled. This allows us to create isolation frameworks that can change the behavior of any method.

There already exists some isolation frameworks that use the profiling API to give us this capability. Examples of such frameworks are Microsoft Fakes, TypeMock Isolator and Telerik JustMock. Please note that these frameworks also support the standard isolation model.

Microsoft Fakes is included in Visual Studio 2015 Enterprise, but not in the Professional or Community editions. The other two frameworks are commercial.

If we wanted, we can keep the Composition Root methods non-virtual, static and private, and use such frameworks to replace dependencies.

Here is how a test using such frameworks looks like:

```
[TestMethod]
public void SomeTestMethod()
{
  //Arrange
  var component2SubComponent1Stub = A.Fake<ISomeInterface>(); //ISomeInterface is
  the return type of CreateComponent2SubComponent1

  A.CallTo(() => component2SubComponent1Stub.QuerySomeSubComponentData())
    .Returns("Some fake data");

  IComponent2 sut;

  using (ShimsContext.Create())
  {
    ShimCompositionRoot.CreateComponent2SubComponent1 = () =>
    component2SubComponent1Stub;

    sut = CompositionRoot.CreateComponent2();
  }

  //Act
  var result = sut.QuerySomeComponent2Data();

  //Assert
  Assert.AreEqual("Some expected data", result);
}
```

This test uses Shims from Microsoft Fakes to replace the behavior of the CreateComponent2SubComponent1 method. For more information about Shims, take a look at this reference from MSDN: https://msdn.microsoft.com/en-us/library/hh549176.aspx

It also uses FakeItEasy to create a stub for the component we want to replace.

External Dependencies

One particular type of dependency that we should care about replacing in unit tests, are external dependencies. Examples of such dependencies are databases, the file system, the system timer, and web services.

It makes a lot of sense to make sure that there exists special "Create" methods in the Composition Root that are responsible for creating these dependencies. This way, replacing such dependencies in tests would be very easy.

Application Configurations

Most applications, if not all, are customizable via configuration. Usually, there is a configuration file beside the application executable that contains all kinds of settings that can be used to customize the application behavior.

In most cases, individual classes should not talk to this configuration file directly. Instead, such settings should be read from the configuration file in the Composition Root, and then individual settings should be injected into objects that need them. This way, the application objects are not tightly coupled to the configuration file.

When writing high-level tests, we should be able to vary the configurations in the *arrange* phase of the test.

One way to do this is to create a method in the Composition Root called something like "GetConfigurations" that would read the configurations from the configuration file or whatever source, and then in the test we can simply change the behavior of this method to return some settings that are appropriate for the test.

Any "Create" method in the Composition Root that needs the configurations would simply call this method to obtain the configurations. Of course, the configurations themselves can be cached inside the "GetConfigurations" method when it is called for the first time.

Automated tests for class libraries

Class libraries shouldn't contain Composition Roots, only applications should. This is true because we need to be able to compose individual objects from the class library inside the application in a certain way. For example, the application might choose to create some caching or logging decorators, and compose them with the objects from the class library.

Nevertheless, in a class library, it makes a lot of sense to create convenient factories that a simple user of the library can use to create the default composition of objects. This way, the user of the library can use the library without understanding a lot about the internal structure of the library, and the advanced user of the library has the ability to customize it in many ways.

Such convenient factories look a lot like Composition Roots and can be used as sources of SUTs also in the same way that Composition Roots can be.

Conclusion:

In this article, I discussed automated testing in general and presented an approach of automated testing that uses the Composition Root as a source or factory of SUTs. This approach can be used when creating tests that verify the behavior of components that are composed of a few or many objects. In this approach we use the "Create" methods in the Composition Root in the *arrange* phase of our tests to create the components that we want to test. This has the advantage of making sure that the system under test, matches the actual component as defined in the Composition Root. It also helps us follow the DRY principle and write tests in the same concrete language we use in the Composition Root. I also demonstrated how we can isolate the SUT by using "Create" methods as isolation boundaries ■

Yacoub Massad
Author

Yacoub Massad is a software developer who works mainly with Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable software. You can view his blog posts at criticalsoftwareblog.com.

Thanks to Damir Arh for reviewing this article.

Daniel Jimenez Garcia

BUILDING DOCKNETFIDDLE USING DOCKER AND .NET CORE

.Net Core is a new lightweight modular platform maintained by Microsoft and the .NET Community on GitHub. .NET Core is open source and cross platform, and is used to create applications and services that run on Windows, Linux and Mac.

WHAT IS DOCKER ?

Docker is a software containerization platform. Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools, system libraries. This guarantees that the software will always run the same, regardless of its environment.

Thanks to Swaminathan Vetri and Suprotim Agarwal for reviewing this article.

This article explores Docker, and how .NET Core applications can be run inside Docker containers by building your own version of dotNetFiddle.

dotNetFiddle is an online environment where users can write and execute simple .Net applications, in a very similar way to the more popular jsfiddle.

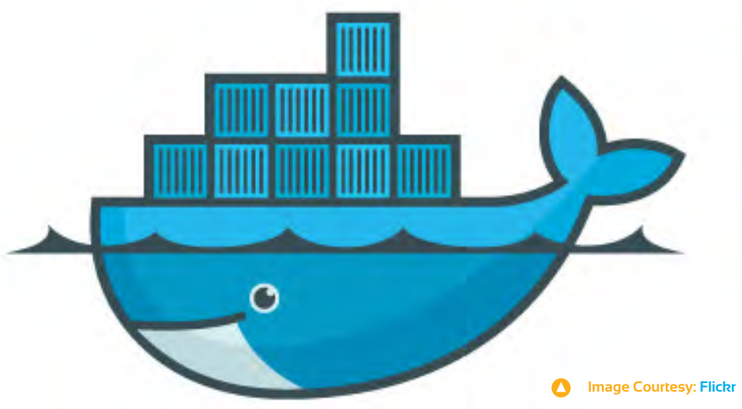


Image Courtesy: Flickr

.Net Core inside Docker containers - Introduction

If you have never used docker before, I would recommend you to check www.docker.com. Download and install docker for your OS, and make sure you run through some of the examples in the Getting Started Tutorial.

The docker website explains What is Docker? In simple words:

Docker allows you to build an image for your application with its necessary dependencies (frameworks, runtimes etc). These images can be distributed, deployed and run within any Docker containers.

Docker also changes the way applications are deployed. The apps as such are no longer deployed to different environments, but it's the images – app with its dependencies, that gets deployed. This way, the application is guaranteed to work with any environment as the image contains everything that's required to run the application.

These containers are different from virtual machines as they share the OS kernel with the host machine. That means they are lighter and easier to start/stop. But they still provide many of the benefits of virtual machines like isolating your processes and files.

Assuming that you are up and running with docker on your machine, let us get started with .Net Core and Docker.

Hello World app inside Docker

Microsoft has conveniently created and published .Net Core Docker images that we can use as a starting point. You can check these images in their GitHub or Docker Hub sites.

Run the following command to start a new container in an interactive mode, attaching the terminal, so we can run commands inside the container:

```
>docker run --rm -it Microsoft/dotnet:latest
```

A few things will happen when running that command:

1. Docker will look for the image **Microsoft/dotnet** with the tag name **latest** in the host machine's local cache.
2. If the image is not found, it searches and pulls it from the docker registry. If the image is found locally, it doesn't download it again.
3. Docker instantiates and starts a new container with the downloaded image. The entry point of that image is executed, which in this case is a linux bash.
4. The container's terminal is exposed and attached to your host terminal so you can interact with it.

After executing the command, you will see something like the following:

```
root@888e80f25148:/#
```

What you see here is the Linux shell which is running inside the container, waiting for you to enter a command! Let's create and run a new .Net Core app:

```
>mkdir helloworld
>cd helloworld
>dotnet new
>dotnet restore
>dotnet run
```

If everything is configured correctly, you will see the following message after the last command:

```
Project helloworld (.NETCoreApp,Version=v1.0) will be compiled because expected
outputs are missing
Compiling helloworld for .NETCoreApp,Version=v1.0
Compilation succeeded.
  0 Warning(s)
  0 Error(s)
Time elapsed 00:00:01.0110247
Hello World!
```

Congratulations, you have run your first .Net Core app inside docker.

What just happened is interesting. We started a Linux container with its own isolated file system. We then initialized an entirely new .Net Core application which was compiled and run. Everything happened inside the container, leaving the host machine clean. And if you are on Windows, it is even more interesting as the docker host is running on a Linux virtual machine!

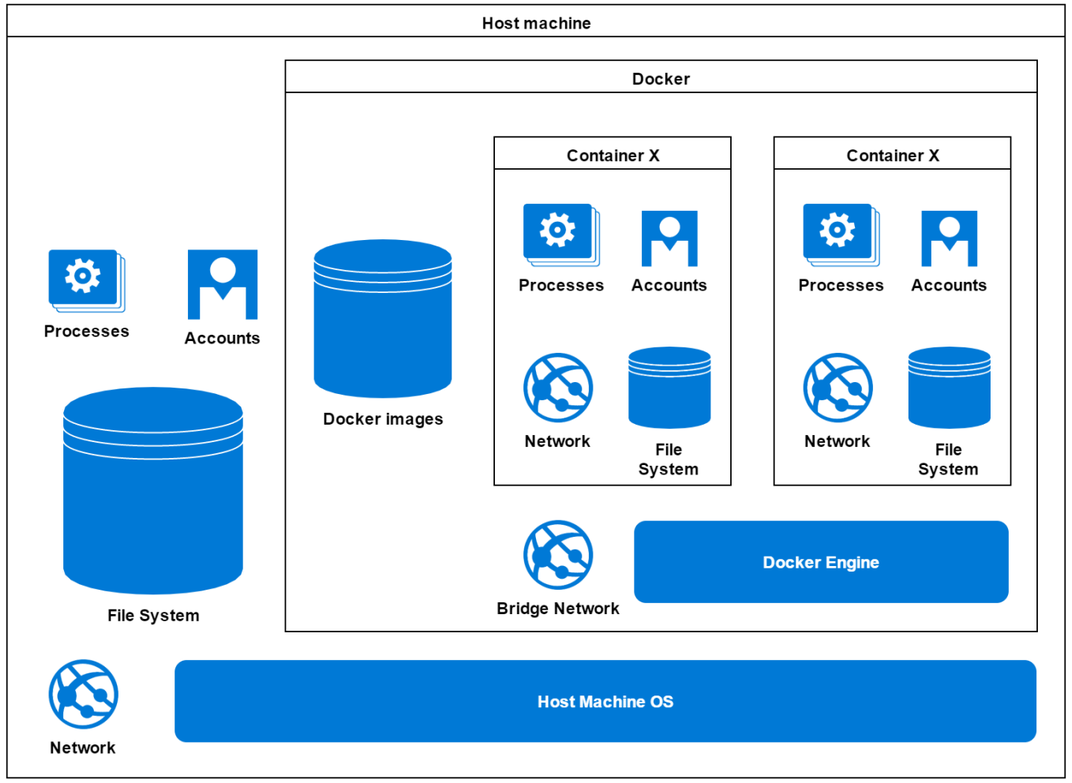


Figure 1: simplified vision of Docker

On the command line, type `exit` to terminate the container shell. This command will also stop the container and return back to the OS shell. Since the container was started with the `--rm` option, it will also be automatically removed as soon as it was stopped. (You can check that by listing all the containers with `docker ps -a`)

File System and Volumes

I hope you found the hello world demo interesting. However docker won't be too exciting if that was all you could do with it. The hello world application was created inside the container, and was lost as soon as the container was removed.

Let's explore some options you have to deal with the file system isolation.

Mounting host folders as volumes

Docker allows you to mount any folder of your host machine at any point of the file system inside the container. So let's revisit the hello world application, this time making sure the application files are created in your host machine.

Start a new command line in your host machine and create a new .Net Core application:

```
>mkdir helloworld
>cd helloworld
>dotnet new
```

You should see the new folder in your machine, which should contain 2 files: **Program.cs** and **project.json**.

Let's now start a new container mounting this folder so that it is available as /app inside the container:

```
>docker run --rm -it -v /c/Users/Daniel.Garcia/helloworld:/app microsoft/
dotnet:latest
```

If you are on windows, the format you need to use to specify the folder depends on whether you are on Windows 10 or not. On Windows 10, docker is accessible from your default command line and you just need to use forward slashes. In previous versions of Windows, you will be using the docker toolbox and will likely be using the git bash as your command line, so apart from using forward slashes, you have to specify your drives as ***/c*** *instead of* ***c:****. In case of any doubts, please check the docker documentation online.*

The volume option (**-v**) maps a folder from your host machine (**C:\Users\Daniel.Garcia\helloworld**) to a folder inside the container (**/app**).

- If you use Docker for Windows, you will need to enable volume mapping in the settings, as explained in this article.

- If you use Docker Toolbox for Windows 8.1 and earlier, you have to check the shared folder settings of the VM created in the Virtual Box. By default, only folders inside C:\Users can be shared, but you can include other folders here.

You can now run `ls /app` (remember your command line is now attached to the shell inside the container) and you will see the two files (program.cs and project.json) of the application.

Build and run the application:

```
>cd /app && dotnet restore && dotnet run
```

Running this command should show the same output as before, but if you check the helloworld folder in your host machine, you will see the generated bin and obj folders! Try deleting all the files inside the container and see what happens to the files in your host machine.

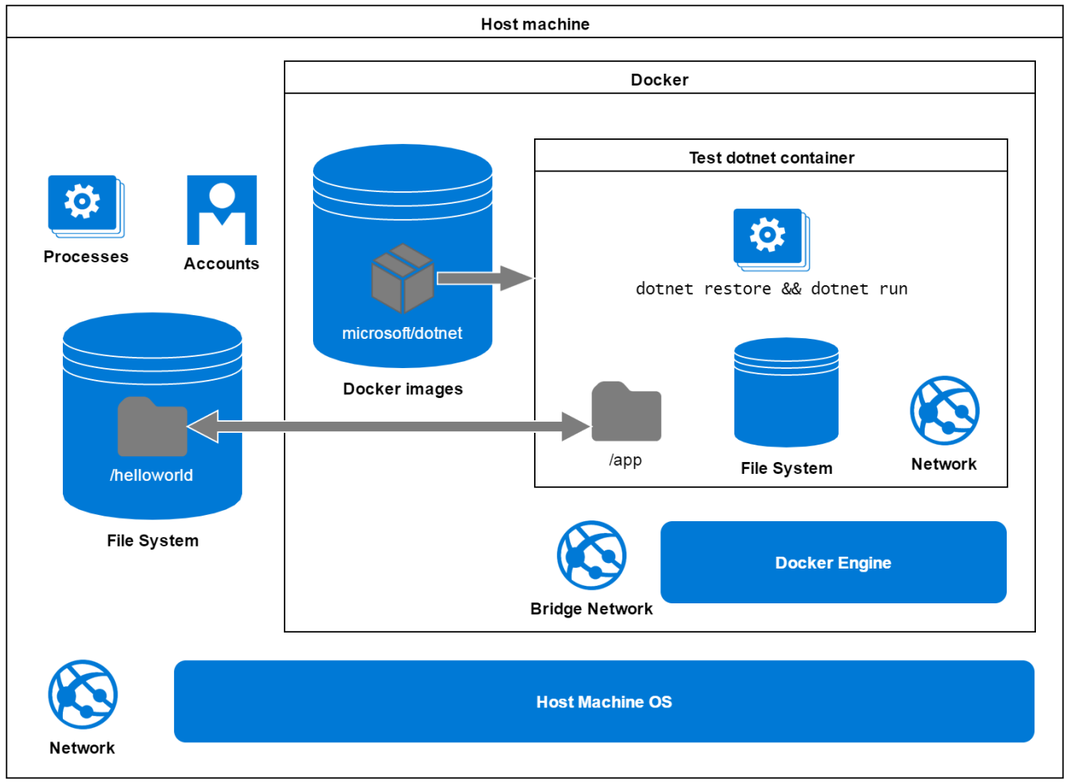


Figure 2: Sharing host folders as a mounted volume

Although this way of sharing data might be useful during development, it is host dependent, which means you cannot easily redeploy your containers on a different host machine.

Building your own images

Reset the files of your hello world application. You are back to a folder in your host machine that contains only **Program.cs** and **project.json**.

Let's now create our own image by using Microsoft's image as the starting point, and copying the application files to the file system inside the container. You will then be able to create containers using this image, and the files will be available inside the container.

Create a new file helloworld.dockerfile inside that folder, and write the following lines of code inside it:

```
FROM microsoft/dotnet

COPY . /app
```

Now build that file in order to create a docker image named **helloworld** (Observe the dot at the end of the command which is intentional!):

```
>docker build -t helloworld -f helloworld.dockerfile .
```

Finally, let's start a new container from our recently created **helloworld** image, instead of the default one from Microsoft:

```
>docker run --rm -it helloworld
```

Once you are inside the container shell, list the files of the **/app** folder. You will see that both Program.cs and project.json are there in the folder. Try compiling and running the application again:

```
>cd /app && dotnet restore && dotnet run
```

Did you notice how nothing gets added this time to the folder in your host machine? Try removing the files inside the container with `cd / && rm -r /app` and notice how the files of your host machine are still not affected.

Once you exit the container, try starting a new container using the same **helloworld** image. What you did previously had no effect, everything is in the expected state, and you can compile and run the application again!

As you can see, you can easily start a new container using your image, and the state will always be the same. This makes it very easy to start containers on different hosts, replacing containers when something goes wrong, or horizontally scaling applications by running more containers.

The way docker builds images using layers is very interesting and understanding the way it works is critical for optimizing your images. It's worth taking a look!

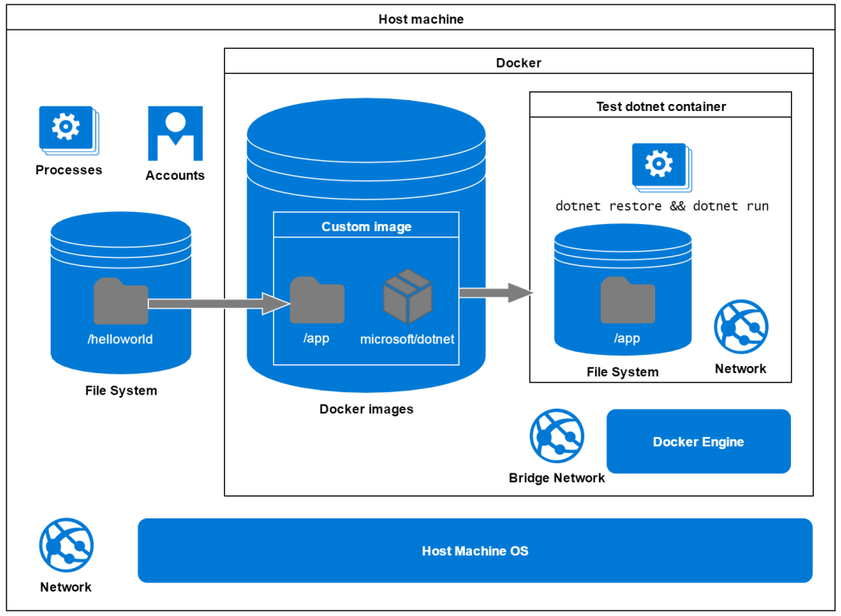


Figure 3: Build an image including folders from host

Creating DockNetFiddle using Docker and .NET Core!

Hopefully you have found this brief introduction interesting. I hope it has triggered multiple use cases for docker (and lots of questions too!) in your brain.

In the rest of the article, we will create DockNetFiddle, a simple clone of dotNetFiddle that will use docker as a sandbox for running programs written online by its users.

As seen during the previous sections, a valid .Net Core application just needs a couple of files, **Program.cs** and **project.json**. The site will allow users to enter the code they want to run, as well as its dependencies. We will then create the corresponding **Program.cs** and **project.json** files and use docker to run the application, sending the output back to the user!

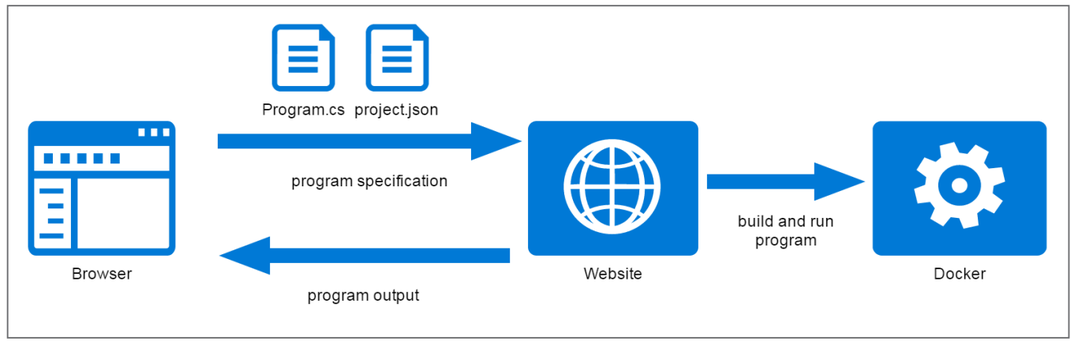


Figure 4: DockNetFiddle high level view

Setting up the project

Although we are going to use ASP.Net Core to create DockNetFiddle, the focus of the article will remain on docker. This means I might speed things up a bit on this section, but don't worry, we just need a very simple website. At its core, it's just a couple of text area fields and a bit of JavaScript to get their values, send a request to the server, and display the results.

Let's start by creating a new *ASP .Net Core Web Application*. Use the *Web Application* template and select *No Authentication*.

Once the project has been generated, delete the `About` and `Content` views (and their actions), leaving just the `Index` view.

Then create a new model class **ProgramSpecification**:

```
public class ProgramSpecification
{
  [Required]
  public string Program { get; set; }
  [Required]
  public string ProjectJSON { get; set; }
}
```

In order to make it easier for users to enter the code they want to run, let's display the default hello world program generated by `dotnet new`. For example, add a static field `ProgramSpecification.Default` hardcoding the `Program` and `ProjectJSON` fields. (By all means, feel free to use a more sophisticated approach using configuration or by getting the values after running `dotnet new`).

Now update the `Index` view so it uses the `ProgramSpecification` as its model and contains a form with textarea inputs for `Program` and `ProjectJSON`, and a submit button. There should also be an element to display the results of running that code.

Go as simple as you want. If you need some inspiration, check the components available on the bootstrap framework, or just take a look at the article code in GitHub! Once you are done, you should have something like this:

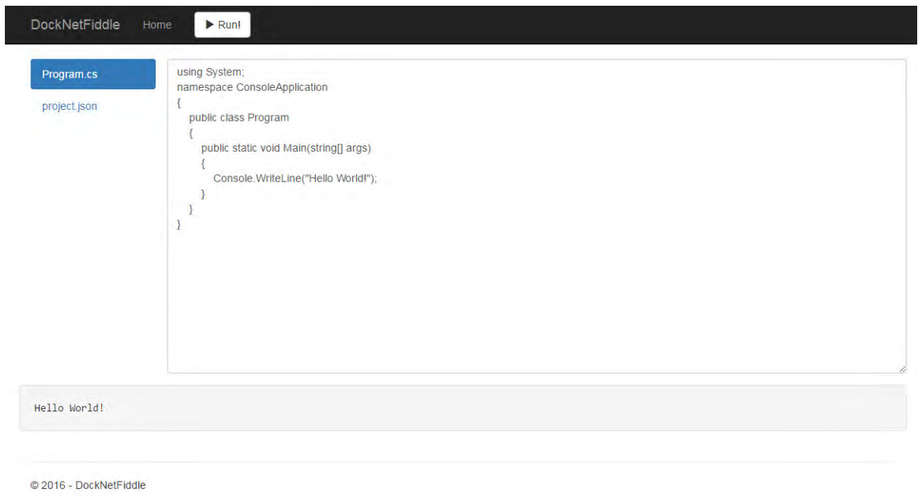


Figure 5: DockNetFiddle design sample

If you feel like giving it a real frontend, feel free to explore some JavaScript editors and replace the raw text areas.

The next step is to create a new controller **RunController** with a single action that will receive POST requests from the client, run the code specified in the request, and send the results back to the browser:

```
[HttpPost]
public IActionResult Index([FromBody]ProgramSpecification model)
{
  if (!ModelState.IsValid) return StatusCode(400);
  return Json(new { result = "Hello World!" });
}
```

Don't worry, this is just a placeholder implementation that will be revisited in the next sections.

The final piece needed is a bit of JavaScript to be executed when the user clicks the submit button. It should just send a POST AJAX request and display the results. I feel like honoring Jose Aguinaga's post How it feels to learn JavaScript in 2016 and will just drop some of that old fashioned jQuery code into the site.js file:

```
$(function () {
  var resultEl = $("#program-result");
  var resultContainerEl = $("#program-result-container");
  $("#run-program-btn").click(runProgram);

  function runProgram() {
    if (!$("#program-form").valid()) return false;

    var data = {
      program: $("#Program").val(),
      projectjson: $("#ProjectJSON").val()
    };

    $.ajax({
      method: 'POST',
      url: '/run',
      data: JSON.stringify(data),
      contentType: 'application/json',
      dataType: 'json',
      success: showResults
    });
    return false;
  }

  function showResults(data, status, xhr) {
    if (data && data.result) {
      resultEl.text(data.result);
      resultContainerEl.removeClass("hidden");
    }
  }
});
```

Feel free to use a different and better approach for the frontend code. For the time being, I just need something functional that can be used through the following sections without overcomplicating the article, and losing the focus on docker.

It's now time to implement the logic that will actually run the received `ProgramSpecification` and send the results back to the browser.

A simple initial approach

A very simple way of running the received program inside a container is basically replicating some of the steps we went through in the introduction:

- Create a temporal directory
- Save Program and ProjectJSON properties into files named Program.cs and project.json respectively.
- Execute a command to start a new docker container mounting the temp directory into /app, using the sequence `cd /app && dotnet restore && dotnet build` as the argument for the container.

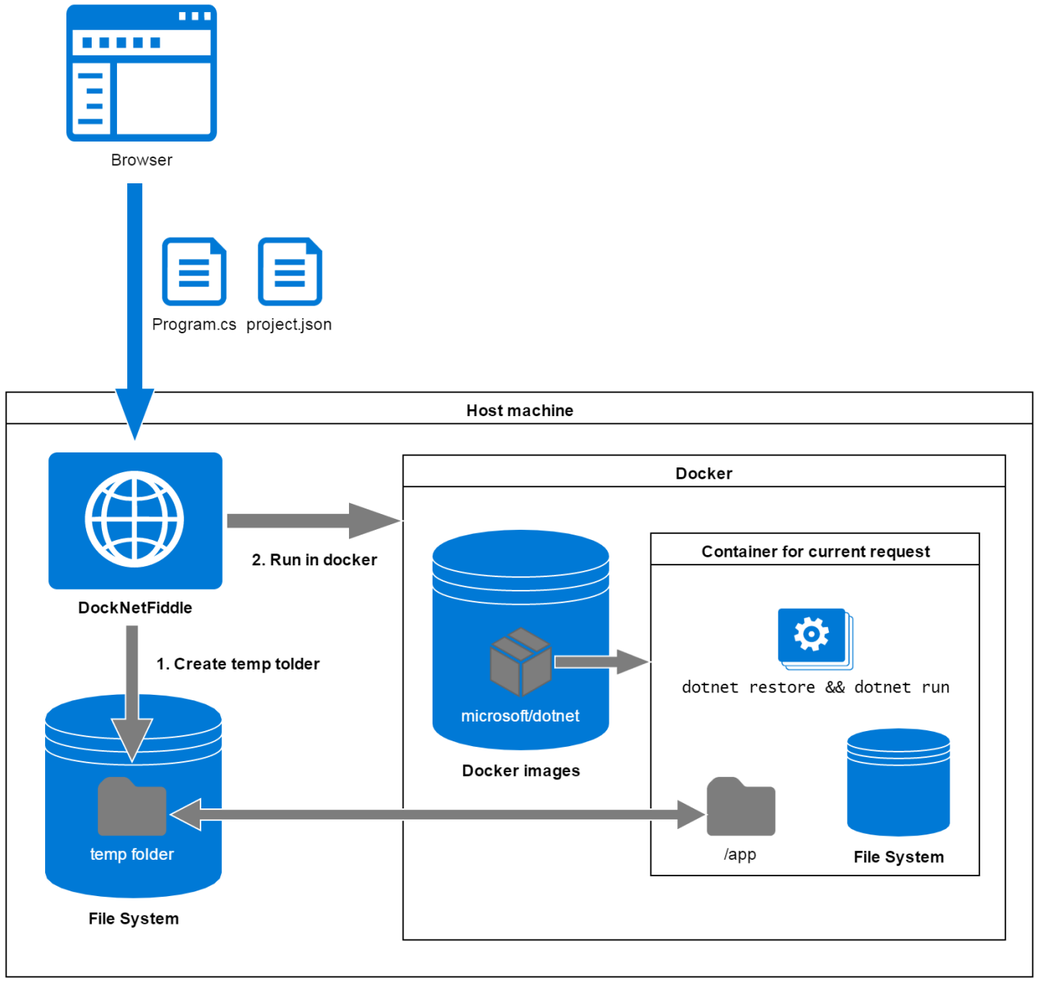


Figure 6: The container per request approach

We can try this easily; just create a folder in your machine containing the two files. Assuming that folder is *C:\Users\Daniel.Garcia\helloworld*, execute the following command (Remember in windows we still had to specify the volume path in Linux fashion):

```
>docker run --rm -t -v /c/Users/Daniel.Garcia/helloworld:/app -w /app microsoft/
dotnet /bin/sh -c "dotnet restore && dotnet run"
```

Let's take a look at the command:

- **--rm**: remove the container once it finished building and running the program.
- **-t**: attach the terminal so we can see the output from compiling and running the program.
- **-v**: mount the folder containing the program files into the /app folder inside the container
- **-w**: set the working directory inside the container to /app
- **microsoft/dotnet**: create the container from Microsoft's default image. You can specify a version here too
- **/bin/sh -c "dotnet restore && dotnet run"**: open a shell inside the container and run dotnet restore and dotnet run

So all we need to do is create a temporal folder with the program files, craft that command from DockNetFiddle, execute it, and return the command output back to the browser. The good old classes **Process** and **ProcessStartInfo** are available in .Net Core, and can be used to run a command. We just need to make some considerations mostly for windows users:

- Path to local folders being mounted as volume should use forward slashes. If you are not using Windows 10, you also need to specify your drives using /c instead of c:

- If you don't use Windows 10, docker isn't available in the command line by default and some environment variable needs to be set. The command to set all the environment variables can be obtained by running `docker-machine env default` in another cmd window, and copying the command from the last line which will look like:

```
@FOR /f "tokens=*" %i IN ('docker-machine env default') DO @%i
```

I will implement it as a windows user, so I will just create a bat file executeInDocker.bat that will be invoked from the DockNetFiddle application with the path to a temp folder as the argument:

```
@ECHO off
FOR /f "tokens=*" %%i IN ('docker-machine env default') DO @%%i
docker run --rm -t -v %1:/app -w /app microsoft/dotnet /bin/sh -c "dotnet restore
&& dotnet run"
```

I will then create a new service interface **IProgramExecutor** and class **ProgramExecutor** (remember to register it as a transient service in your **Startup.ConfigureServices** method). The service will expose a single method that we will use from the **RunController**:

```
public interface IProgramExecutor
{
  string Execute(ProgramSpecification program);
}

//In RunController.Index:
return Json(new {
  result = executorService.Execute(model)
});
```

The implementation of **ProgramExecutor** is quite straightforward:

```
private IHostingEnvironment env;
public ProgramExecutor(IHostingEnvironment env)
{
  this.env = env;
}

public string Execute(ProgramSpecification program)
{
  var tempFolder = GetTempFolderPath();
  try {
    CopyToFolder(tempFolder, program);
    return ExecuteDockerCommand(tempFolder);
  }
  finally {
    if (Directory.Exists(tempFolder))
      Directory.Delete(tempFolder, true);
  }
}
```

There are a couple of private methods to create a temp folder and to create the required files using the received program specification:

```
private string GetTempFolderPath() => Path.Combine(Path.GetTempPath(), Guid.
NewGuid().ToString());


private void CopyToFolder(string folder, ProgramSpecification program)
{
  Directory.CreateDirectory(folder);
  File.WriteAllText(
    Path.Combine(folder, "Program.cs"),
    program.Program);
  File.WriteAllText(
    Path.Combine(folder, "project.json"),
    program.ProjectJSON);
}
```

And another private method that will execute the bat file with the temp folder as argument (taking care of the path format) and get the results:

```
private string ExecuteDockerCommand(string folder)
{
  var dockerFormattedTempFolder = folder
    .Replace("C:", "/c")
    .Replace('\\', '/');
  var proc = Process.Start(new ProcessStartInfo
  {
    FileName = Path.Combine(env.ContentRootPath, "executeInDocker.bat"),
    Arguments = dockerFormattedTempFolder,
    RedirectStandardOutput = true,
    RedirectStandardError = true
  });
  var result = proc.StandardOutput.ReadToEnd();
  var error = proc.StandardError.ReadToEnd();
  proc.WaitForExit();

  return String.IsNullOrWhiteSpace(error) ? result : error;
}
```

Once you have everything in place, you should be able to enter any piece of code in your site and execute it!

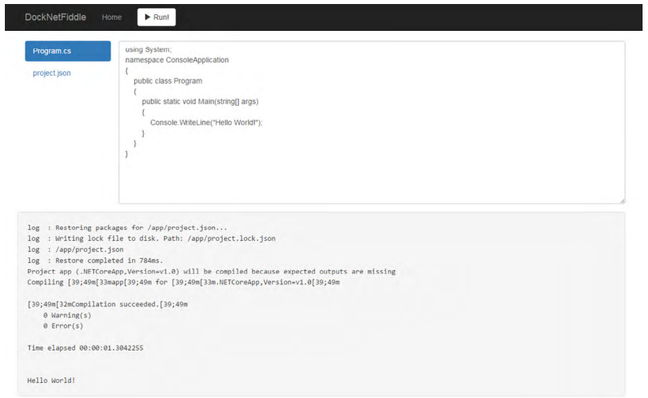


Figure 7: Testing the first implementation

Take a moment and play with your site. While this current simple approach works, it requires a new container to be created for each request. This is a waste of resources and provides poor performance. There surely is a better approach, right?

Use a long lived container

So here is another approach that takes advantage of the fact that docker allows you to send commands to already running containers using docker exec.

We can share a folder between our machine and the container (mounting it when starting the container), copy the program files there, and use docker exec to compile and run the program inside the container!

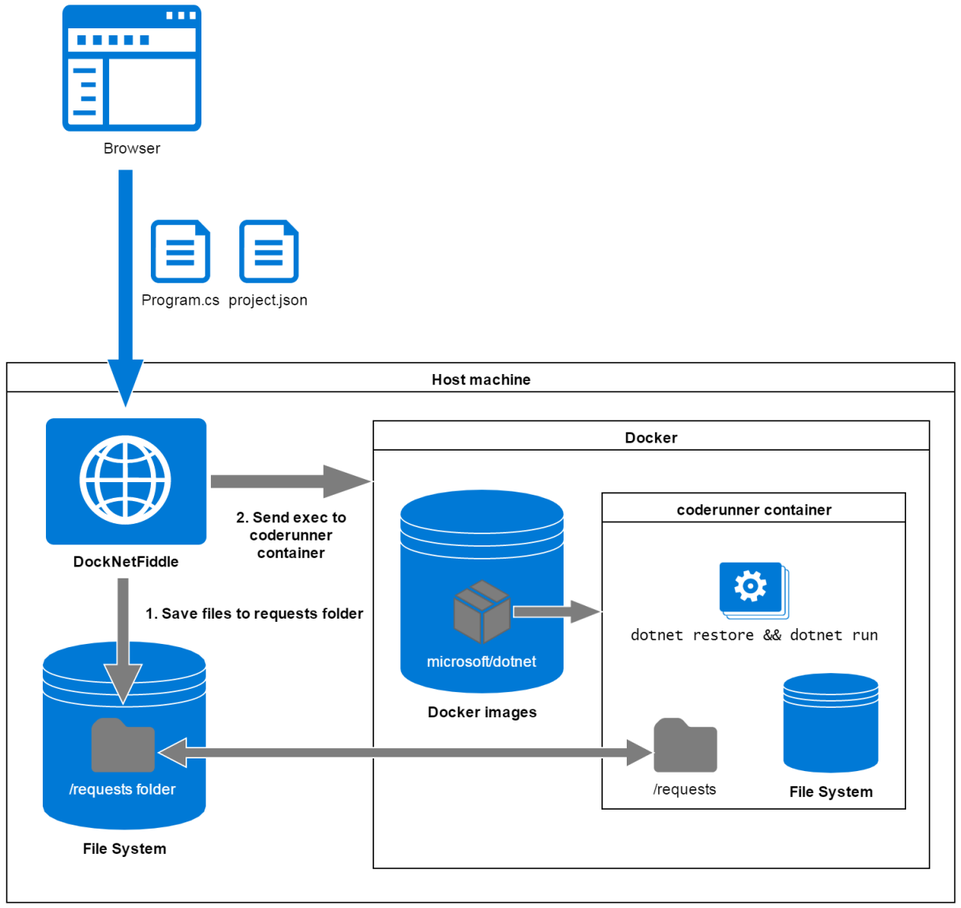


Figure 8: Using the long lived coderunner container

Let's give it a try. Create a folder named **requests** somewhere in your machine and then start a new container using the following command (notice we give it a name so we can later send commands using docker exec):

```
>docker run --rm -it --name coderunner -v /c/Users/Daniel.Garcia/requests:/requests
microsoft/dotnet
```

Now in your machine, create a new folder named **test** inside the **requests** folder, and put the default Program.cs and project.json files there. If you switch to your container, you should be able to run ls /requests/test and see both files.

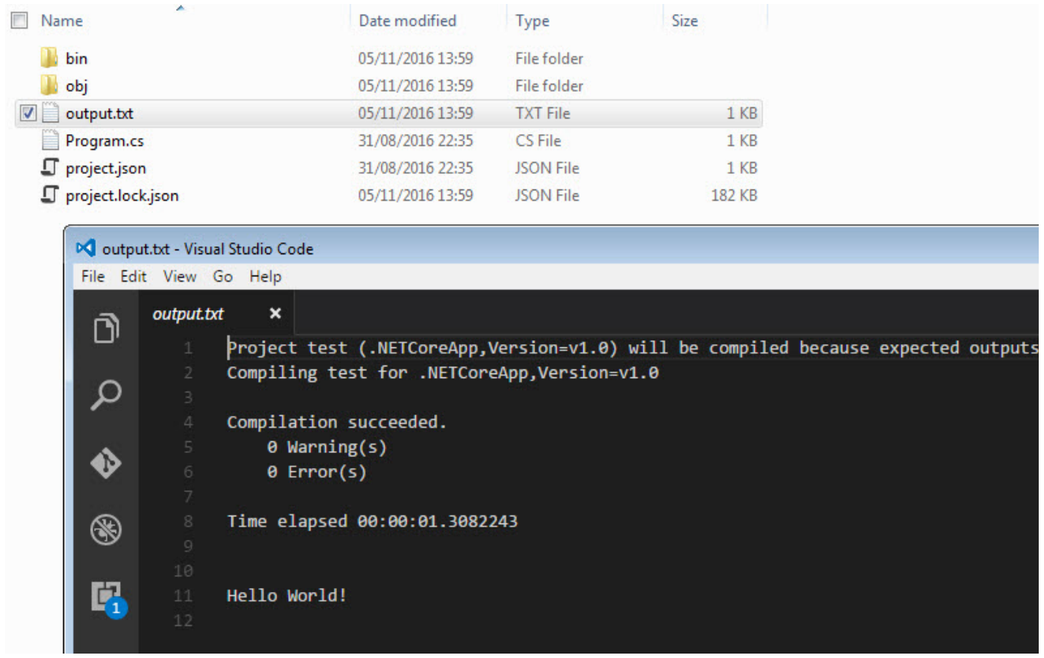
Now open a different command line and execute the following docker exec command:

```
>docker exec coderunner /bin/sh -c "cd /requests/test && dotnet restore && dotnet
run"
```

You should see the output of the program being compiled and run! Now run the same thing again but redirect the output to a file:

```
>docker exec coderunner /bin/sh -c "cd /requests/test && dotnet restore && dotnet
run >> output.txt"
```

If you check the test folder on your machine, you will see that the outputs were compiled there, and our file output.txt contains the output from compiling and running the application:



Let's quickly adapt this technique in our application. Update your bat file so it looks like the following:

```
@ECHO off
FOR /f "tokens=*" %%i IN ('docker-machine env default') DO @%%i
SET folderName=%1
SET "dockerCommand=cd /requests/%folderName% && dotnet restore && dotnet run"
docker exec coderunner /bin/sh -c "%dockerCommand%"
```

Then just update the ProgramExecutor class to:

- copy the files into a new folder inside the same requests folder that was mapped as a volume for the coderunner container. I will just hardcode the folder but feel free to use a configuration for it.

- provide the name of the new folder as argument to the bat file instead of the full path

```
public string Execute(ProgramSpecification program)
{
 var tempFolderName = Guid.NewGuid().ToString();
 var tempFolder = Path.Combine(@"C:\Users\Daniel.Garcia\requests", tempFolderName);

 try
 {
  CopyToFolder(tempFolder, program);
  return ExecuteDockerCommand(tempFolderName);
 }

 finally
 {
  // no changes
 }
}

private string ExecuteDockerCommand(string tempFolderName)
{
 var proc = Process.Start(new ProcessStartInfo
 {
  FileName = Path.Combine(env.ContentRootPath, "executeInDocker.bat"),
        Arguments = tempFolderName,
        RedirectStandardOutput = true,
        RedirectStandardError = true
    });

    // no more changes in this method
}
```

This is better than our initial approach, but we are not quite there yet. There are still quite a few things that are less than ideal like:

- We need to manually start the coderunner container separately from the website.
- The website shouldn't know about the commands needed to run the program inside the container
- We rely on sending docker commands from the website. What if the website doesn't have access to docker?
- We rely on a host folder shared between the website and the container. What would happen if you want to also host the website inside docker?

Create an image for the code runner

We will start improving things by creating a shell script for the coderunner container. This script will receive the path to a zip file containing the program files and will proceed to unzip the file, build the program, and run it. This way the website doesn't need to know the linux commands needed to compile and run the program.

Create a new folder coderunner inside your project and populate the coderunner.sh bash script with the following contents (please ignore my newly acquired bash scripting skills!):

```
#!/bin/sh

zipFilePath=$1
zipFileDir=$(dirname $1)
zipFileName=$(basename $1)
tempAppFolder="/tmp/$zipFileName"

# unzip program into temp folder
unzip -o $zipFilePath -d $tempAppFolder
cd $tempAppFolder

# restore and run program, saving output in same folder as original file
dotnet restore && dotnet run > "/$zipFileDir/$zipFileName.output" 2>&1

# remove temp folder
rm -rf $tempAppFolder
```

If you use windows, make sure you save the file with linux line endings. For example, in Visual Studio, click File, Advanced Save Options and select Linux line endings.

Now create a docker **coderunner.dockerfile** file inside the same folder, which will basically create an image from Microsoft's default image that additionally copies the coderunner.sh script, and installs zip inside the container:

```
FROM microsoft/dotnet:latest
RUN apt-get -qq update && --assume-yes install zip
COPY coderunner.sh /coderunner.sh
```

Now we can run the following commands to build our image and start a new container with our image:

```
>docker build -t coderunner -f ./coderunner/coderunner.dockerfile ./coderunner/
>docker run --rm -it -v /c/Users/Daniel.Garcia/requests:/requests coderunner
```

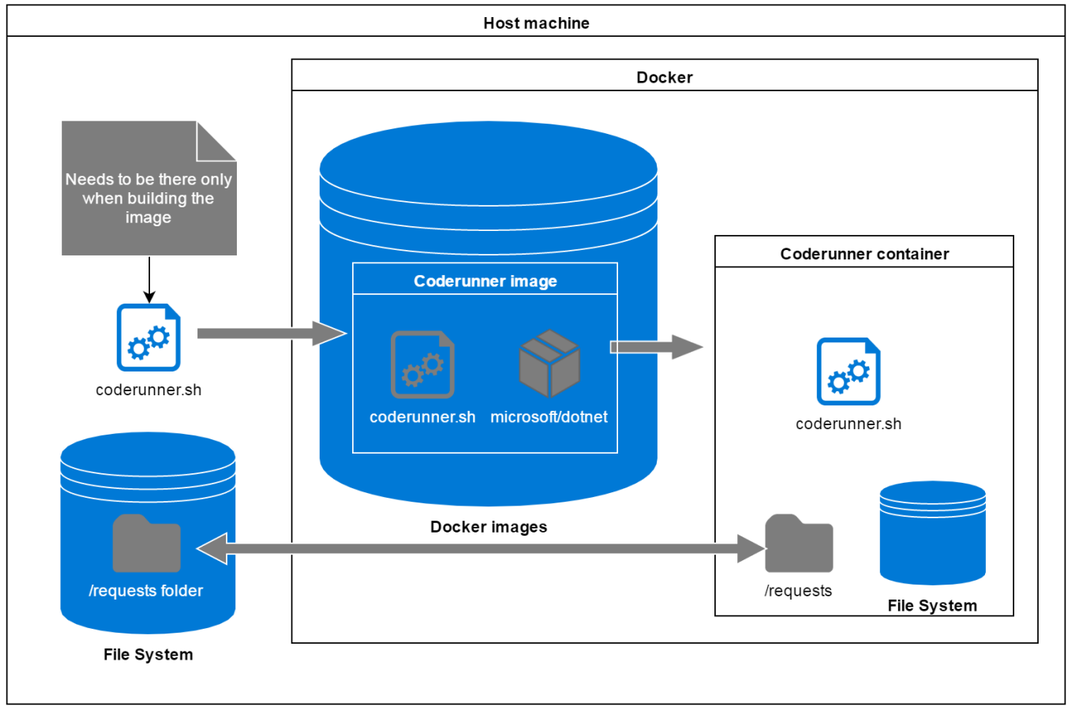


Figure 10: Creating an image for the coderunner container

Now move back to the website as we need to change the ProgramExecutor service. We now need to generate a zip file inside the requests folder and also need to monitor the folder until we see the output file. This will convert our method into an async method:

```
public async Task<string> Execute(ProgramSpecification program)
{
  var zipFileName = Guid.NewGuid().ToString() + ".zip";
  var zipFilePath = Path.Combine(
  @"C:\Users\Daniel.Garcia\requests", zipFileName);
  var expectedOutputFile = zipFilePath + ".output";
  try
  {
    DropToZip(program, zipFilePath);
    ExecuteDockerCommand(zipFilePath);
    await WaitForOutputFile(expectedOutputFile);
    return File.ReadAllText(expectedOutputFile);
  }
  finally
  {
    if (File.Exists(zipFilePath))
      File.Delete(zipFilePath);
  }
}
```

The DropToZip utility uses the old .Net Stream to create a zip file that contains the two program files inside it:

```
private void DropToZip(ProgramSpecification program, string zipFilePath)

{
    using (var zipFile = File.Create(zipFilePath))
    using (var zipStream = new System.IO.Compression.ZipArchive(zipFile, System.
IO.Compression.ZipArchiveMode.Create))
    {
        using (var writer = new StreamWriter(zipStream.CreateEntry("Program.cs").
Open()))
        {
            writer.Write(program.Program);
        }
        using (var writer = new StreamWriter(zipStream.CreateEntry("project.json").
Open()))
        {
            writer.Write(program.ProjectJSON);
        }
    }
}
```

The ExecuteDockerCommand is now just firing the bat file:

```
private void ExecuteDockerCommand(string zipFilePath)
{
  var proc = Process.Start(new ProcessStartInfo
  {
    FileName = Path.Combine(env.ContentRootPath, "executeInDocker.bat"),
    Arguments = Path.GetFileName(zipFilePath)
  });
  proc.WaitForExit();
}
```


..whereas the **WaitForOutputFile** utility uses the **FileSystemWatcher** events combined with **Tasks**:

```
private Task WaitForOutputFile(string expectedOutputFile)
{
  if (File.Exists(expectedOutputFile))
    return Task.FromResult(true);

  var tcs = new TaskCompletionSource<bool>();
  var ct = new CancellationTokenSource(10000);

  ct.Token.Register(
    () => tcs.TrySetCanceled(),
    useSynchronizationContext: false);

  FileSystemWatcher watcher = new FileSystemWatcher(
    Path.GetDirectoryName(expectedOutputFile));
    FileSystemEventHandler createdHandler = null;
    createdHandler = (s, e) =>
    {
      if (e.Name == Path.GetFileName(expectedOutputFile))
      {
        tcs.TrySetResult(true);
        watcher.Created -= createdHandler;
        watcher.Dispose();
      }
    };

  watcher.Created += createdHandler;
  watcher.EnableRaisingEvents = true;
  return tcs.Task;
}
```

The final piece of the puzzle is the bat file. Now we just need to start the **sh** script inside the container, which takes a path to the zip file inside the requests folder as an argument:

```
@ECHO off
FOR /f "tokens=*" %%i IN ('docker-machine env default') DO @%%i
SET zipFileName=/requests/%1
SET "dockerCommand=./coderunner.sh %zipFileName%"
docker exec coderunner /bin/sh -c "%dockerCommand%"
```

Run everything in docker!

We have improved on the previous setup, but this really was just an intermediate step. Now let's take the final step so as to run the website too inside docker.

- This will allow us to share the folders without depending on the host machine
- It will also allow us to stop sending *docker exec* commands from the website. Instead we will install **incron** inside the coderunner image, which will start monitoring the requests folder and automatically run the script for any new zip file!
- **Docker compose** will allow to start/stop both containers with a single command

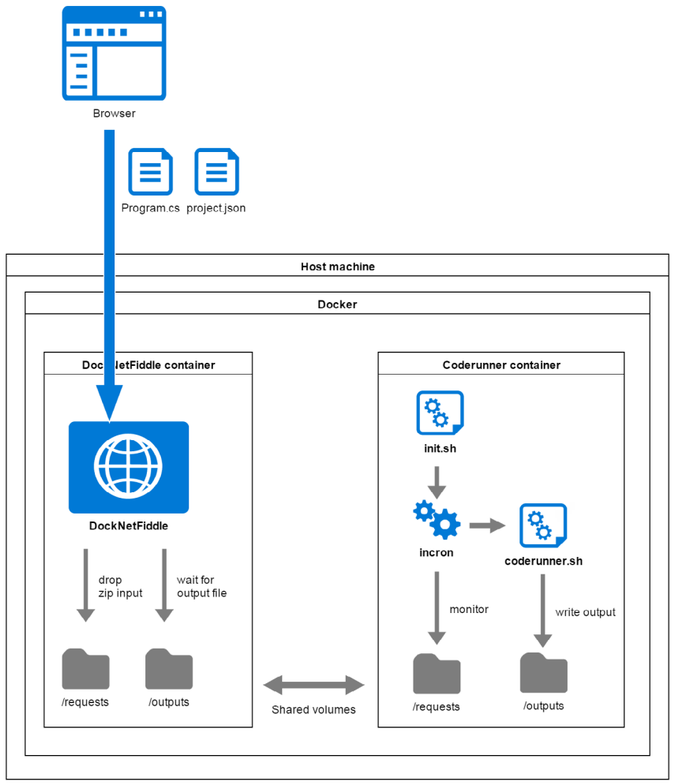


Figure 11: Updated approach, hosting everything in docker

Let's start with the coderunner container. Update its dockerfile so:

- It exposes a couple of folders **/requests** and **/outputs** that can be shared with other containers. (But they are completely independent from the host machine)

- Installs **incron** and creates a rule for monitoring the new files in **/requests** that will run the **coderunner.sh** script.

There are a few caveats to take into account before we can complete this step.

- The first is that incron runs the script on a different context that doesn't have the same environment variables, so we will create an intermediate script that will fire coderunner.sh as the root user. Create a script **launcher.sh** inside the coderunner folder with these contents:

```
#!/bin/sh
su - root -c "/coderunner.sh $1"
```

- The second is that when building an image on windows machines, copied files are not given executable permissions, so we will need to manually do that.

- The third is that since we are monitoring the **/requests** folder, we cannot create the output file in that folder, or else the **coderunner.sh** will enter an infinite loop. Update coderunner.sh to create the output file inside the **/outputs** folder at the end of the process:

```
dotnet restore && dotnet run > "/$tempAppFolder/$zipFileName.output" 2>&1
#copy entire output
cp /$tempAppFolder/$zipFileName.output /outputs/$zipFileName.output
```

- The final caveat is that docker images save the file system, but not the status of running processes! This means we need to run another script when the container starts that will start the **incron** service. We will also use that script to keep the container alive. Create another script **init.sh** with these contents (remember to save the file with linux line endings):

```
#!/bin/sh
service incron start
echo "Pres CTRL+C to stop..."
while true
do
 sleep 1
done
```

Once you have done this, update the docker file for the coderunner container:

```
FROM microsoft/dotnet:latest

# Install zip and incron
RUN apt-get -qq update && --assume-yes install zip incron
RUN echo 'root' >> /etc/incron.allow

# Expose required volumes
VOLUME /requests /outputs

# Copy script files
COPY /init.sh coderunner.sh launcher.sh ./

# Monitor folder where new zip files will be copied
# executing the launcher.sh script when a new file is created.
RUN touch incron.rules \
    && echo '/requests IN_CREATE /launcher.sh $@/$#' >> incron.rules \
    && incrontab incron.rules \
    && chmod +x /launcher.sh \
    && chmod +x /coderunner.sh \
    && chmod +x /init.sh

# Start incron service when container starts and keep container alive
CMD /init.sh && bash
```

With these changes, our coderunner container is ready!

Let's now create a dockerfile for the website. I am going to follow the quick and lazy approach of copying our entire application inside the container, then build and run the application.

For a proper way of optimizing your images for ASP.Net Core applications, check Steve Lasker's post.

You might also want to check the VS tools for docker as it easily provides you with docker files for an ASP.Net Core application that even supports running with F5 and debugging.

If you are developing .Net core apps on Mac/Linux, you might also want to check out the Docker Support extension for VS Code that gives linting, code snippet support for docker file. Combine that with the generator-docker yeoman package to debug .net core apps from VS Code

That said; create a new docknetfiddle.dockerfile inside the root folder of your ASP website with the following contents:

```
FROM microsoft/dotnet
WORKDIR /app
ENV ASPNETCORE_URLS http://+:80
ENV ASPNETCORE_ENVIRONMENT development
EXPOSE 80
COPY . .
RUN dotnet restore
RUN dotnet build
ENTRYPOINT ["dotnet", "run"]
```

We are almost ready to try our new setup. Before you do that, adjust the **ProgramExecutor** service so it creates/expects files in the **/requests** and **/outputs** folders:

```
var zipFilePath = Path.Combine("/requests", zipFileName);
var expectedOutputFile = Path.Combine("/outputs", zipFileName + ".output");
```

Note: *I have seen issues with the FileSystemWatcher events when running inside docker. Sometimes the event got fired, the file was there but it was still empty. Adding a small 'wait' before reading the file contents with* `Thread.Sleep(10)` *sufficed to fix the issue.*

Finally remove the **ExecuteDockerCommand** function from **ProgramExecutor**. Now that the coderunner container will use incron to monitor the **/requests** folder, there is no need for the website to manually invoke coderunner.sh!

Once done with the changes, run the following commands from the website root folder. These commands will build the coderunner image and start a new container:

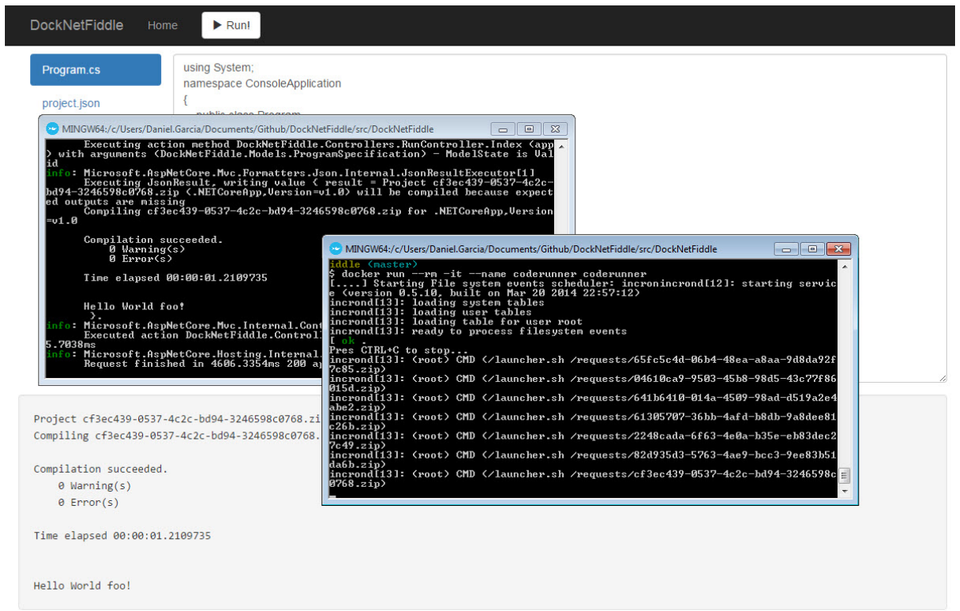
```
>docker build -t coderunner -f ./coderunner/coderunner.dockerfile ./coderunner
>docker run --rm -it --name coderunner coderunner
```

Open a second command line and repeat the same to start the website container. Notice how port 8080 of the host machine is mapped to port 80 inside the container, and how we mount the folders exposed by the coderunner container, so both containers share **/requests** and **/outputs**:

```
>docker build -f docknetfiddle.dockerfile -t docknetfiddle .
>docker run --rm -it --volumes-from coderunner -p 8080:80 docknetfiddle
```

Now direct your browser to localhost:8080 and you will see your site running on docker. It is still fully functional, but everything is running inside docker and is completely independent from your host machine!

Note: *If you are in windows using docker toolbox, instead of localhost, you need to use the IP of the VM hosting docker. For example* http://192.168.99.100:8080/



A Quick Improvement

There is a quick improvement you can make if you want. As you might have already realized, most of the times, you don't add new dependencies to the default project.json.

We could set the project.json as an optional input in the website, which means we use the default project.json unless a different one is entered. Then we could create a precompiled app inside the coderunner container and use it to avoid the dotnet restore step. Basically inspect the zip file to see if it contains a project.json file and if it doesn't, clone the precompiled app, replace its Program.cs file and `run dotnet build && dotnet run` instead of `dotnet restore && dotnet run`.

If you have any trouble making these changes, check the accompanying code in GitHub.

Introducing Docker Compose

Before wrapping up, let's briefly introduce **docker compose**. If you take a look at how the application currently needs to be started, you need to open a couple of different command line windows, and run at least one command on each to start the containers (more if you also need to build the images first). You also need to remember to start the coderunner first, and then the docknetfiddle container; as the latter mounts the volumes exposed by the former.

Docker compose is an orchestration tool that lets you easily manage setups that require multiple containers. Docker compose is installed when you install Docker for Windows/Mac. It just needs a single file describing the different containers required by your system, and their relationships. With that single file, you are able to run and stop everything with a single command!

Enough talking, let's just create a file named **docker-compose.yml** within the web application folder, the same folder that contains docknetfiddle.dockerfile. Add the following lines to the file, **making sure indentation uses spaces and not tabs:**

```
version: '2'
services:
  webapp:
    build:
      context: .
      dockerfile: docknetfiddle.dockerfile
    ports:
      - "8080:80"
    volumes_from:
      - coderunner
  coderunner:
    build:
      context: ./coderunner
      dockerfile: codeRunner.dockerfile
```

I think the commands are self-explanatory. The file basically describes that our system is composed of two different containers, both being built from docker files. You can also see the ports and volume mapping instructions for the docknetfiddle container, same as we manually did to add to a docker run command.

Now open a new command line from the folder where the docker file is located and run the following command: `>docker-compose up`

If your file name is different from docker-compose.yml, use the option `-f yourfile.yml`

That's it, your DockNetFiddle site is up and running! When you are done playing with it, run the following command to stop and delete the containers:

```
>docker-compose down
```

As you can see, Docker Compose makes it easier working with non-trivial docker setups involving multiple containers.

Conclusion

I truly hope you are now intrigued by Docker and its potential. I am no expert in Docker by any means, and have only recently started exploring and learning about it. However I think it is full of potential. As soon as you start with Docker, you will start seeing how it could add value in many situations!

It also comes with its own set of challenges like debugging, security or a different approach for operations. Not to mention, Windows systems other than Windows 10 are less than ideal for serious work with Docker. Make sure you investigate these (and many other) topics before embarking on serious projects.

This article should be considered as a learning exercise rather than a guide to create a production ready application. Concerns like scalability, security or performance were not fully covered. However it demonstrates the multi-platform capabilities of .Net Core (remember, the site built in the article is running inside a docker container, which is running itself on a Linux host) and the possibilities it opens. You now have a platform that makes it really easy to decouple your system into different containers.

I would encourage you to fork this project and add some additional features to it. For example, let's say you want to continue working on the sample DockNetFiddle application, how hard would it be to replace the communication based in shared folders with a proper messaging/queueing solution like Kafka or RabbitMQ hosted on a 3rd container? Wouldn't you be able to easily scale up by adding multiple containers of the coderunner instance? What about adding the ELK stack for monitoring the site? And you can run everything locally the same way it would run in production!

Now think about doing the same without containers!! ■

Download the entire source code from GitHub at
bit.ly/dncm28-docker

Daniel Jimenez Garcia
Author

Daniel Jimenez Garcia is a passionate software developer with 10+ years of experience. He started as a Microsoft developer and learned to love C# in general and ASP MVC in particular. In the latter half of his career he worked on a broader set of technologies and platforms while these days is particularly interested in .Net Core and Node.js. He is always looking for better practices and can be seen answering questions on Stack Overflow.

Thanks to Swaminathan Vetri and Suprotim Agarwal for reviewing this article.

Gerald Versluis

AUTOMATED CI/CD FOR YOUR XAMARIN APP

Besides the regular development work, there are tons of tools and practices that can make your life a lot easier as a developer. One of it is Continuous Integration and Continuous Delivery (CI/CD), a term you seem to see everywhere nowadays. And for a reason!

CI/CD can save you a lot of (repetitive) work and time that you can instead invest in what you love to do the most - developing. Moreover the chances of things going wrong when you automate a process, is reduced to a minimum.

In this article, you will learn how to set up CI/CD for your Xamarin app with Visual Studio Team Services (VSTS), Bitrise and HockeyApp - **completely for free**!

Thanks to Subodh Sohoni for reviewing this article.

As a developer, receiving feedback on your work early on, helps. Depending on what your requirements are, you want to know if your (unit) tests still succeed, or if your code plays well with the rest of the application, or the code builds not just 'on my machine', but on everyone else's too.

With the acquisition of Xamarin and HockeyApp, Microsoft now has a complete and fully automated build pipeline which we developers can benefit from.

In this article, we will discuss how to setup a mature and extensive building platform, without spending a dime on it! Each service in use will be discussed, followed by how they can all be combined to build and deliver apps in a continuous fashion.

Image Courtesy: Wikimedia Commons

Visual Studio Team Services

If you have been developing in .NET, I'm going to assume that you have worked with Team Foundation Server (TFS) before. It has been a couple of years since TFS got an online cousin called Visual Studio Team Services (VSTS). This version offers everything that the on-premises TFS has to offer, and probably even more! **When you are using it with a small team up to five people, you can use it completely for free**. If you have an MSDN subscription, these users do not count towards that limit of five, so you can have even bigger teams.

You will probably know that VSTS can be used as a code versioning system, but if you login to the web portal, you will see that it can be used for a lot more. Take a look at Figure 1 for instance.

Figure 1: menu structure within VSTS

If we just look at the tabs that are available to us, we will find that everything that we would need, can be done from within VSTS. From the options we see, we could guess that we can manage our code, and even manage our work with work items on digital Scrum boards etc. So much so, even building and releasing can be done (the focus of this article), and last but not the least, we can define our tests and results too.

If we were building a fairly standard .NET or Azure application, this would all be true without any limitations. But in the case of a mobile (Xamarin) app, not so much. Especially when we focus on iOS for which Apple still requires you to have a Mac to build your app on.

While Microsoft now does offer some templates to setup builds for Xamarin, you still need to supply your own Mac hardware as a build agent to get it to work. Since I promised you we would do it for free, I won't get into this option. Another option would be to rent a Mac with services like Macincloud.com, but again, its not free.

For Android and Windows Phone (or Universal Windows Platform) apps, you can work with the default build agent installation that Microsoft provides you with.

Another service that does provide you with a Mac to build your apps on, is Bitrise. I will talk about this awesome service in a little bit. I just have to mention that before you can integrate a third-party service into VSTS, I would recommend you to use Git as a versioning system. Git option is now available from VSTS and they even default to it.

By using the very popular Git, you are now able to let other services access the repository easily and work with your code. You can also swap out VSTS for Github, or any other popular Git service.

Bitrise

Bitrise is a new PaaS Continuous Integration and Deployment platform specializing in apps. They offer a free tier which should be enough for every hobby developer. There are of course some limitations like: 2 team members, max. 10 minute builds and max. 200 builds per month, but to start with, it is more than enough. Should you need more over a period of time, there is just one paid tier which is 50 dollars (at the time of this writing) and gives you pretty much everything unlimited. If you want more concurring builds, just add 50 dollars for each concurrency you would like to add.

You can find them at https://bitrise.io, and sign up easily with the 'Sign Up' button in the upper-right. While signing up, you can use your GitHub, Bitbucket or GitLab account, and your repositories will be linked automatically. If you do not want that and want to use VSTS instead, just create a new stand-alone account by simply specifying a username, email and password.

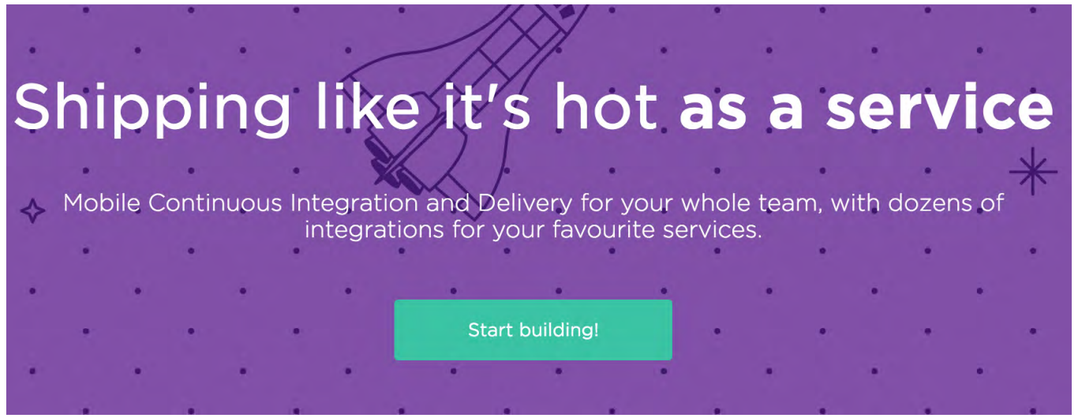


Figure 2: everything is a service at Bitrise

Everything at Bitrise is web-based, carries a modern look and is easy to use. You can easily create your app, compile a workflow with the steps that are needed to build your app, and of course trigger a build. Everything is open source so you can even contribute something to the workflow. A lot of developers already have contributed, therefore a lot of steps for your workflow are already available. For instance, you can put your app through Xamarin Test Cloud, send it to HockeyApp to be distributed, automatically set up your version number, or upload your app directly to the Google Play Store and/or App Store.

HockeyApp

If you have already developed an app before, you will probably know that Google and Apple both have possibilities to distribute it as a beta version.

You could also do that through a third-party - HockeyApp (https://hockeyapp.net).

HockeyApp has a couple of advantages over other vendors. It has more extensive features, offers your users a more unified experience and besides testing, you can also collect crash reports and statistics.

Just like Bitrise, HockeyApp offers a free plan to start with. You can add two apps, but with unlimited versions and collecting crash reports etc. Plans after that start at 10 dollars per month for personal licenses and 30 dollars for businesses.

Putting it all together

How can we put these services together to create an integrated, automated pipeline that you do not need to worry about?

If we put it into an image, this is how we would like to achieve delivery.

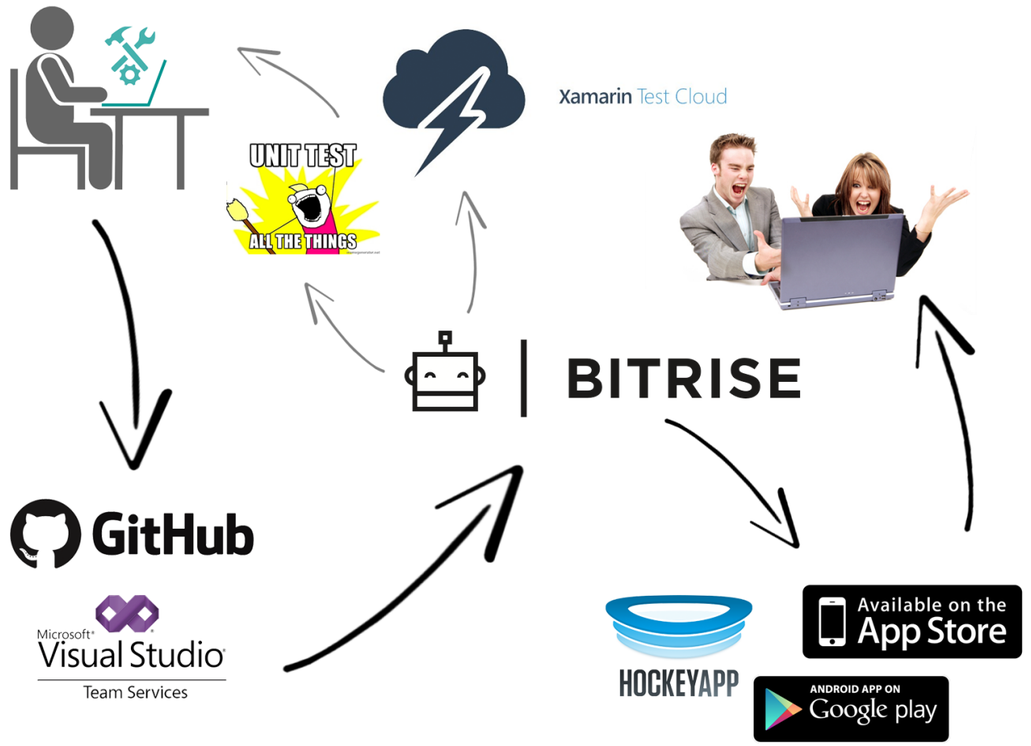


Figure 3: how we would like to handle delivery

As a developer, we commit out code to VSTS or GitHub, which sends a request to Bitrise, which in turn starts a build. Depending on how we have configured that build, it will generate a binary, put it through Test Cloud or any other kinds of tests, and distribute it through HockeyApp of the app stores.

Of course, you can customize this any way you like, but this setup will suffice for now. For this article, I will leave testing out of scope because you could write multiple articles on just that subject. Instead I will focus on the bare minimum: commit, build, distribute.

The start of our journey is to create a new build definition on Bitrise. This can be done by simply pushing the 'Add new app' button. After doing so, you are taken to a screen where you must configure your repository.

If you signed up using Github or similar, you will probably see all your repositories there and it should be pretty straight-forward.

We will be focusing on making it work with VSTS.

Since you can only make private repositories in VSTS, you need a way to make it accessible from third-party solutions. This can be done by either generating alternative credentials, or using SSH keys. We will be using the latter. Go to the project you want to build with Bitrise, and get the clone URL. You can do this by going to the 'Code' screen in your Team Project and find the 'Clone' button in the upper-right corner. Make sure you get the SSH url by clicking the 'HTTPS | SSH' button. You can recognize it because it starts with 'ssh://'.

Now when we go back to Bitrise, we notice VSTS is not one of the pre-configured providers, so select the 'Other/manual' option and paste in the clone URL you just found in VSTS and click 'Next'. To authorize access to the private VSTS repository, you need to verify access with an additional SSH key. Since it cannot be added automatically, click the 'I need to' button as seen in Figure 4.

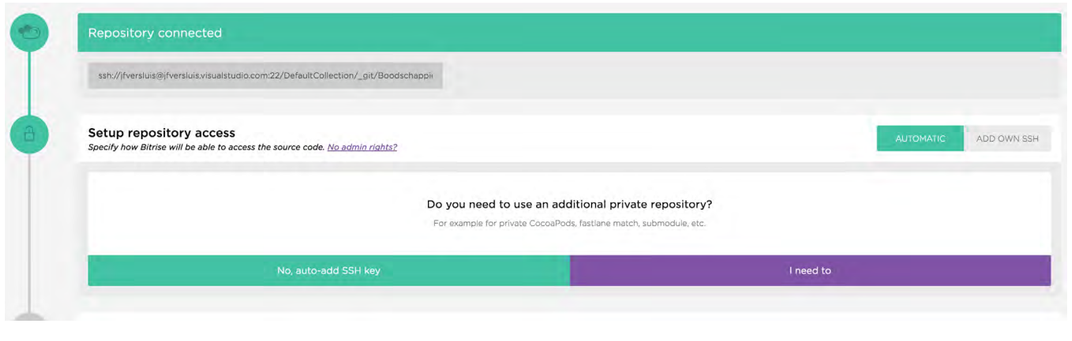


Figure 4: add the SSH key

After you press the button, you will be presented with an SSH key. You have to add this to VSTS to confirm that you have admin access to your private repository. Copy the text and go back to your VSTS account.

Remember where you got the SSH url for cloning the repository? Go back there and click 'Manage SSH keys' this time. In the new screen that comes up, click 'Add' and give the key a description, and paste the copied value from Bitrise in the 'Key Data' field. Save the new key and go back to Bitrise. Click 'I've added the SSH key' to continue.

After selecting the repository and having scanned it, Bitrise will automatically detect that it is an Xamarin app, as you can see in Figure 5. When clicking the 'Manual' button, you can also see that apps in their native languages are supported. But since we want to use the Xamarin app – that should be detected – just go back to the detected screen, and select the solution you want to build.

After you have selected the solution, choose the build configuration – Debug or Release – and platform.

Note, that you need to create a release build to be able to distribute it. These configurations are all extracted from your solution file which you have created from within Visual Studio or Visual Studio for Mac (formerly known as Xamarin Studio). You will be asked for Xamarin credentials. Since Xamarin is free nowadays, you do not need to enter these anymore. Just click the 'Skip this step' button.

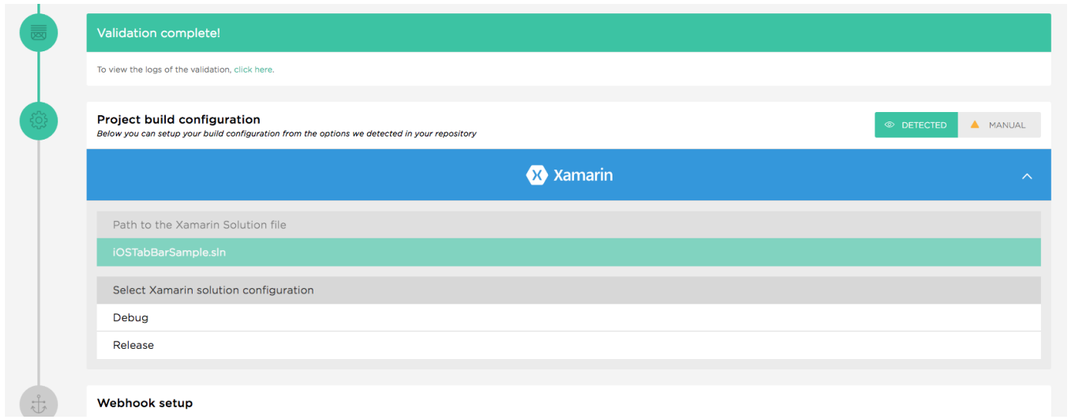


Figure 5: selecting the build configuration

As a final step to initially configure our build, we can create a webhook. This installs a webhook in VSTS which calls Bitrise. This is needed as when you commit new code, VSTS will know it has to call the webhook to notify Bitrise that an action is needed. In the case of Github and others, it can automatically register a webhook for you, although for VSTS, you need to set it up yourself. We will see how to do this in a little bit.

Including this method, there are three ways to trigger a build: manually from the web interface, calling the 'Trigger API' they made available, or through the webhook in your repository.

When your configuration is done, a first build is automatically triggered to see if everything is working as it should.

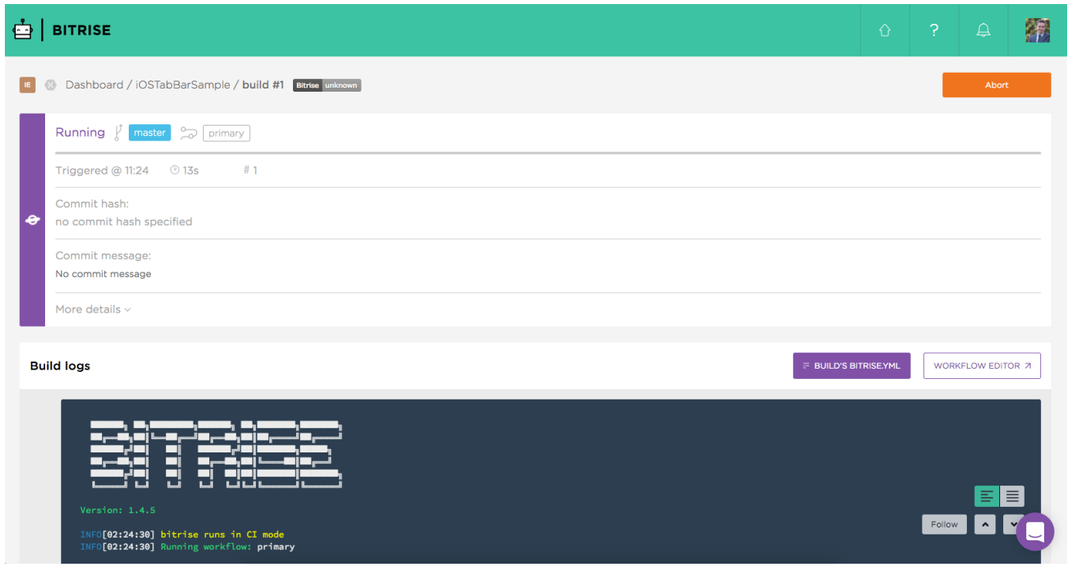


Figure 6: our very first build

During build, a very verbose log scrolls over your screen, with each step that runs. This way you can trace what is happening, or more important; what went wrong at what stage. The point of all this is of course that you do not want to wait for the build. So you can just carry on working, and an email with the results

will be sent to you once the job is done.

Building your app will most likely succeed right away (hooray!) however, to actually distribute or run it on a device, you need to sign your binary file, especially with iOS. Since we didn't do any actual configuration on what our build does, it has not been signed yet. Before we will look at how to do this, let's go back to setting up the webhook.

To do so, go to VSTS and navigate to the project you are building. Go to the cogwheel in the upper bar and select 'Service Hooks'. Click the plus button to add a new one and select 'Web Hooks' as a provider. After clicking 'Next', set the trigger to 'Code pushed' and under filters, select the repository you want to build. Additionally you can specify the branch or even the user.

After you have configured it as you want to, go to the next screen. The last thing you need to configure here is the url that needs to be called when the trigger is fired. You can find this in your Bitrise build definition. Go to the app you have created, and into the 'Code' tab. Under the 'Webhooks' title, you can select VSTS as a provider, and Bitrise will give you the url that is to be called from the webhook. Paste this back into VSTS and you're done! A build is now triggered when you push code to the repository.

Back to signing our binaries. I am going to assume that you have some knowledge on how to sign and distribute an iOS and Android app. This can be a somewhat complex procedure and is beyond the scope of this article. If you do not have any experience with signing binaries, you can learn how to do this with the Xamarin documentation here for iOS and Android here.

To customise the workflow of a build, go to your app and click on the 'Workflow' tab. Here you will see the default workflow that has been compiled for you by creating this app. The concrete implementation can differ depending on the template that has been selected when creating the app in Bitrise. But roughly it will be retrieving the code, restoring the required (NuGet) packages, building the actual app, and deliver the resulting binary. Files that are delivered are referred to as artifacts.

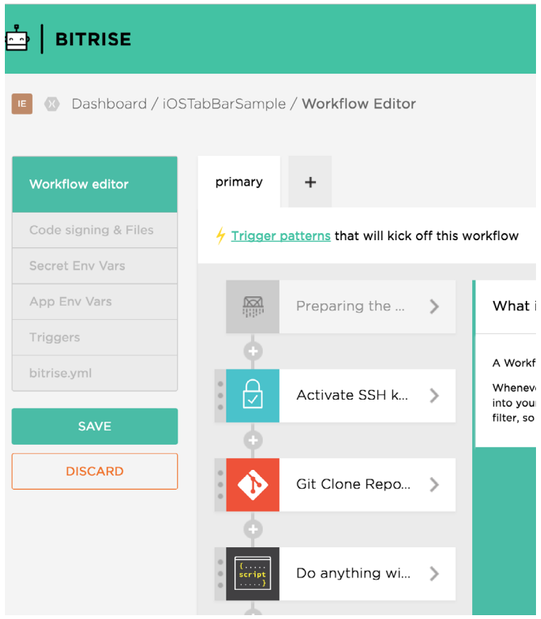


Figure 7: steps in the workflow

If you want to customise your build, you can just click on any plus sign and add a step there. You can also rearrange them once you have added them. Since everything is open-source and the developers of Bitrise allow for other developers to add their own build steps, there is already a big library with all kinds of steps you can add. There are very useful steps in there - like for versioning your app, incorporating tests but also more fun steps like getting a quote of the day in your build log.

On the far left, you will also see an option called 'Code signing & Files'. This is where you need to upload your Provisioning Profile and p12 certificate for your iOS app. In case of an Android app, you will need to add a 'Sign APK' step to your workflow and configure it with the keystore file.

After you have configured this right, and your build still works, we will now have binaries we can distribute through HockeyApp!

If you do not already have an account there, sign up for one. To start off, it is completely free if you do not need more than two apps. When you are creating multi-platform apps with Xamarin, unfortunately it still counts as one app per platform. So if you are targeting Android and iOS, that is two apps.

When you log into HockeyApp, you can create an app, just like you did with Bitrise. Click the 'New App' button and then choose to create it manually.

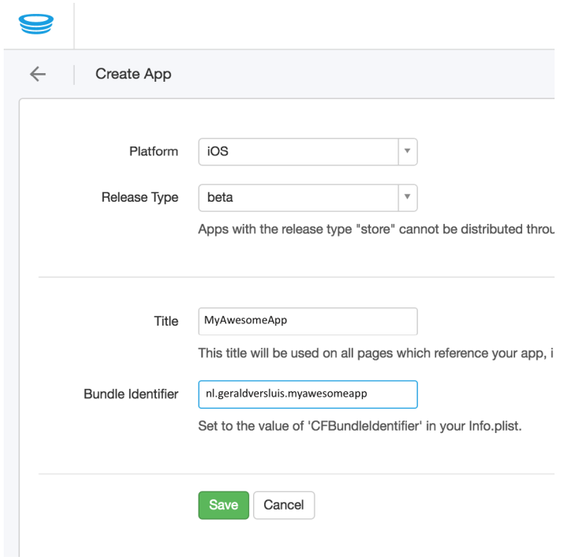


Figure 8: creating your app in HockeyApp

After specifying the necessary details and saving your app definition, you will be taken to the dashboard. There are two things we need from HockeyApp - the *App ID* which helps determine which HockeyApp definition is coupled with which Bitrise build; and the *API key* needed to make a connection between Bitrise and HockeyApp.

You can find the App ID in HockeyApp right on the dashboard of your app, in the top left corner. Note it for now and then go to the upper-right corner, click your avatar and choose 'Account Settings'. Under 'API Tokens,' you can create a new token which you can use with Bitrise.

Configure it however you want to and also note this API key.

Of course there are a lot more things you can configure in HockeyApp, like your team or user groups to distribute your app to, etc. I invite you to explore that for yourself. To get versions over for distribution, this is enough for now. For now go back to Bitrise and edit your workflow again. Just before the last step (Cleaning up the virtual machine) add one new step (or two if you're doing both iOS and Android from one build). The step is called 'HockeyApp <platform> Deploy', where platform is of course iOS or Android. Add the API token and App ID to the configuration and save it. Figure 9 shows how it looks at the end.

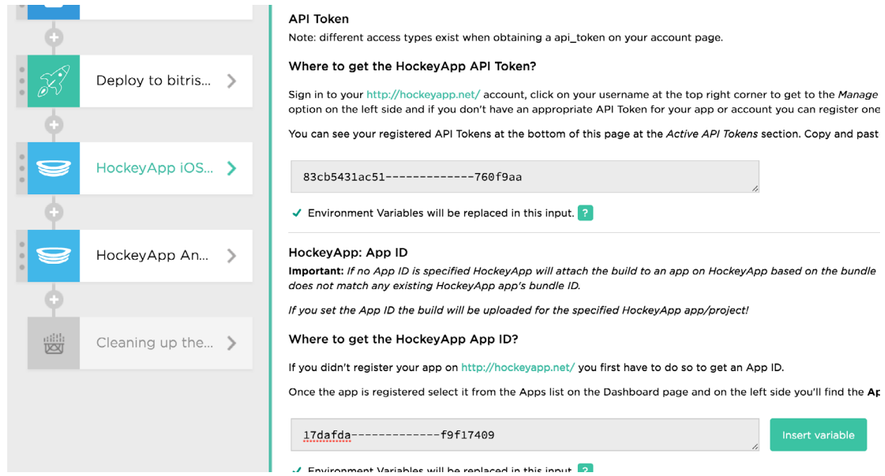


Figure 9: complete HockeyApp configuration in Bitrise

By saving the configuration, you have now completed your pipeline. Congratulations! Whenever you do a commit to your repository, a build is triggered on Bitrise, which will go through this workflow. If all goes well, it signs your binaries and sends them over to HockeyApp. Depending on the configuration there, your testers and/or end-users will automatically get notified and can download the new version of your app.

All of this, without any direct input from you.

Of course there are a lot more configuration options available for you to play with, you can completely shape everything according to your requirements.

While writing this article, Microsoft has announced the Visual Studio Mobile Center at Connect();. This solution is very similar to what Bitrise offers (including a Mac build machine!) but takes it even a step further with deeper integration with Xamarin Test Cloud and HockeyApp, since these are now Microsoft products. Although the mobile center is at an early stage right now and works only with GitHub repositories, a lot of exciting features are still to come. You can sign up right now using the 'Get started for Free' button on the website.

Conclusion

This article demonstrated how to set CI/CD up for your Xamarin app for a small team, with Visual Studio Team Services (VSTS), Bitrise and HockeyApp, completely for free! ■

Microsoft® MVP Most Valuable Professional

Gerald Versluis
Author

Gerald Versluis (@jfversluis) is a full-stack software developer and Microsoft MVP (Xamarin) from Holland. After years of experience working with Xamarin and .NET technologies, he has been involved ina number of different projects and has been building several apps. Not only does he like to code, but he is also passionate about spreading his knowledge - as well as gaining some in the bargain. Gerald involves himself in speaking, providing training sessions and writing blogs (https://blog.verslu.is) or articles in his free time. Twitter: @jfversluis Email: gerald@verslu.is Website: *https://gerald.verslu.is*

DOTNETCURRY TEAM AT THE MICROSOFT MVP SUMMIT 2016

It was that time of the year again, when MVPs from over 70 countries travelled to attend the yearly MVP Global Summit at Microsoft's headquarters in Redmond. We got a chance to present a hard copy of the DNC Magazine to a selected few, including the three famous Scott's of Microsoft. We even got featured on Channel 9 (Link: dnc-mag-channel9). Check out what we were upto:

At Channel 9 for an Interview

With Immo Landwerth

"Glowing Red" With Scott Guthrie

Pizza Party with DNC Mag TEam

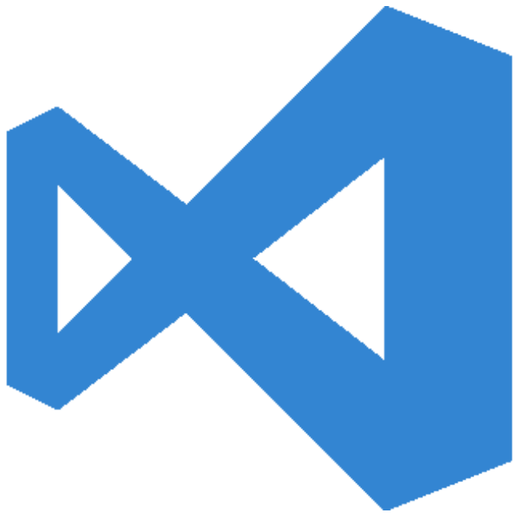
with Scott Hanselman

DNC team: Subodh, Suprotim, Vikram and Craig

With Brian Harry

Damir and Suprotim

Damir Arh



VISUAL STUDIO CODE CONDENSED

Visual Studio Code is a lightweight programming editor, available for Windows, Linux and Mac. It is open source and very actively developed. The monthly updates always include many new features and improvements. Thanks to a lively community ecosystem, numerous additional functionalities are available as free extensions.

This article should help you get started, or make you more productive at this increasingly popular text editor.

Getting Started

The Visual Studio Code installer for all supported platforms (Windows, Linux, macOS) can be downloaded from https://code.visualstudio.com/. You can also download the Insiders build, i.e. a preview version that can be installed in parallel to the stable version. Their icons are of different colors (blue icon for release build, green for the Insiders build), to easily differentiate between the two. Both builds automatically download and install updates, when the editor is restarted.

Once you open VS Code, you will observe most of the window is dedicated to the editor. There are tabs for switching between the currently opened files above it, and a status bar with basic information about the currently edited files, below it. To the left of the editor, there is a vertical view bar with five icons (See Figure 1). By clicking them, the following side bars open, from top to bottom:

- *Explorer* side bar (keyboard shortcut `Ctrl+Shift+E`) lists the currently opened files, and the contents of the open folder. The number in the icon badge (not shown) indicates how many files have unsaved changes.
- *Search* side bar (`Ctrl+Shift+F` or `Ctrl+Shift+H`) allows searching and replacing across all files in the opened folder, and provides a preview of the results.
- *Git* side bar (`Ctrl+Shift+G`) allows basic interaction with Git version control. The number in the icon badge indicates how many files have uncommitted changes.
- *Debug* side bar (`Ctrl+Shift+D`) contains the main debugger user interface.
- *Extensions* side bar (`Ctrl+Shift+X`) allows management of extensions. The number in the icon badge indicates how many installed extensions do newer versions have available.

The currently open side bar can be hidden using `Ctrl+B`.

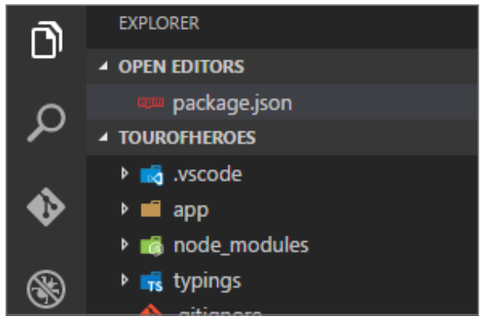


Figure 1: View bar with Explorer side bar opened

Although the editor allows opening of individual files, many of its more advanced functionalities requires the user to open a folder, which takes on the role of a workspace. This can be done by selecting the *File* > *Open Folder...* menu item, or by clicking the *Open Folder* button in the Explorer side bar (see Figure 2). By default, Visual Studio Code will reopen the last open folder when it starts the next time. If the corresponding option is selected in the installer, the *Open with Code* shell extension can also be used when right clicking a folder in Explorer.

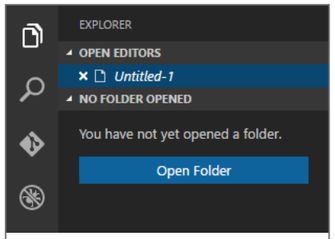


Figure 2: Open folder from Explorer side bar

The editor keeps a track of recently opened files and folders. They can be accessed by navigating to *File* > *Open Recent* menu item. In Windows, both the lists are also available from the taskbar jump list, which can be accessed by right clicking the application icon in the Windows taskbar. The installer automatically adds Visual Studio Code executable to PATH (except on the Mac, where it can be done manually from within the editor), so that files and folders can easily be opened in the editor from the command line:

```
# open current folder in a new Visual Studio Code window code .
# open current folder in existing Visual Studio Code window code . -reuse
# open the listed files in a new Visual Studio Code window code readme.md .gitignore
package.json
# open the listed folders, each one in its own Visual Studio Code window code
project1 project2
```

There are additional command line arguments available for even more control over the editor.

Becoming Productive

Keyboard shortcuts in a text editor plays a big role in making users productive, therefore shortcuts will be mentioned throughout the remainder of the article. It is also strongly recommended that you print out the keyboard shortcuts reference for your OS, and keep it on your desk. You can download a copy via the *Help* > *Keyboard shortcut Reference* menu item.

Command Palette

Probably the most important part of the editor to learn about is the Command Palette (keyboard shortcut `F1` or `Ctrl+Shift+P`). As an alternative to menus and keyboard shortcuts, it provides access to all available commands - both built-in, and those added by extensions. Each command in the list also displays the associated keyboard shortcut, if assigned. By typing in text, you can filter the sorted list for quicker access to the command.

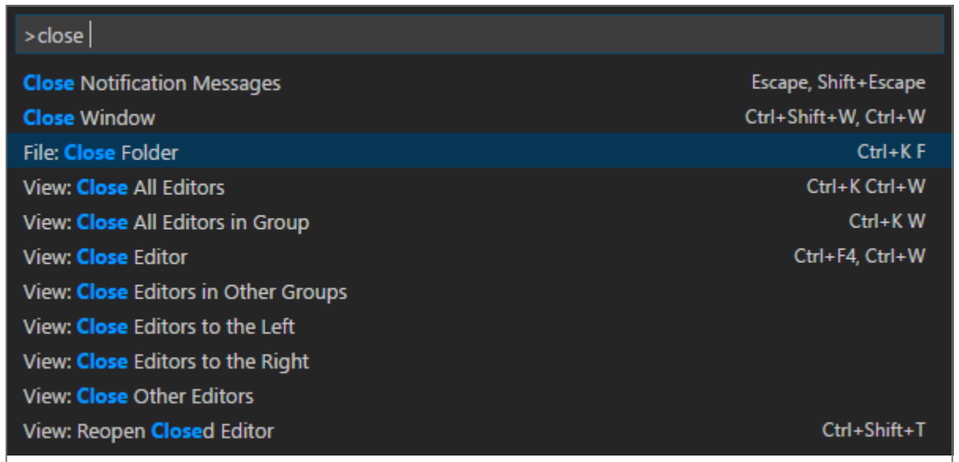


Figure 3: Invoking a command from the Command Palette

Depending on the first character in the input box, the Command Palette supports different modes. Each one of them is also directly accessible with its own keyboard shortcut.

First Character	Keyboard Shortcut	Action
	`Ctrl+P`	Quick open a file in workspace
`>`	`F1` or `Ctrl+Shift+P`	Command Palette
`:`	`Ctrl+G`	Go to line number in current file
`@`	`Ctrl+Shift+O`	Go to a symbol in current file
`#`	`Ctrl+T`	Go to a symbol in workspace

Table 1: Supported actions in Command Palette

As with commands, you can filter the resulting list by typing additional text in the input box. If you enter only the '?' character, help with all the options will be shown.

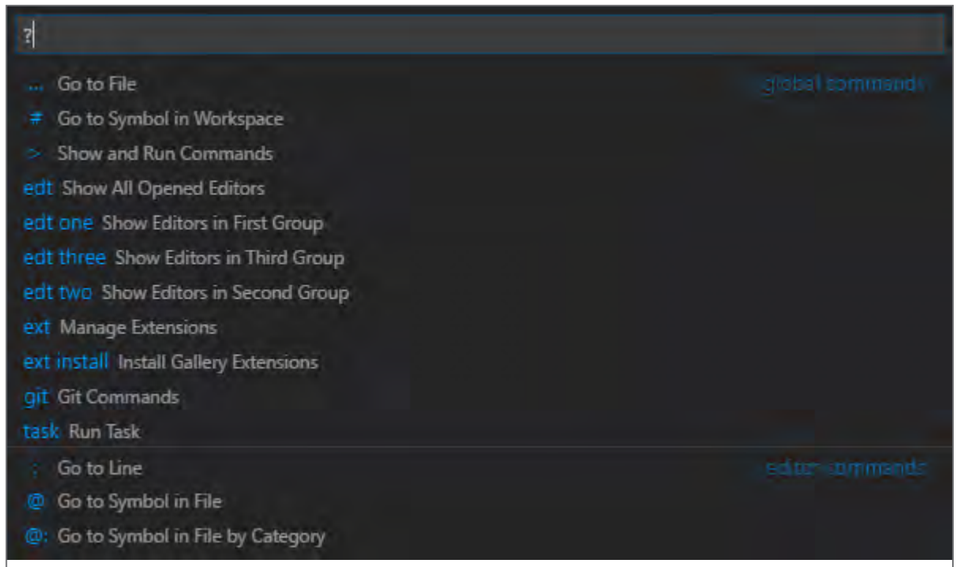


Figure 4: Built-in Command Palette help

Tabs and Groups

By default, files will open as new tabs in the editor, unless there is already a file in the editor in preview mode. In this case, the newly opened file will replace it. Every opened file is in the preview mode, until you edit it or double click on it. Files in preview mode have their filename in italics.

The editor also supports side by side editing, by organizing the files in up to three groups. You can arrange the groups vertically or horizontally, and switch between the two modes with a button in the Open Editors section of Explorer side bar, or with `Shift+Alt+1` keyboard shortcut. To open another view of the current file in a new group, click on the Split Editor icon on the top right corner or press `Ctrl+\` . To open a new file in a new group, `Ctrl` click on it in Explorer side bar, or press `Ctrl+Enter` in quick open dialog. Use `Ctrl+1`, `Ctrl+2` or `Ctrl+3` to switch to first, second or third group respectively. Use `Ctrl+Tab` to switch to

a different file in the same group.

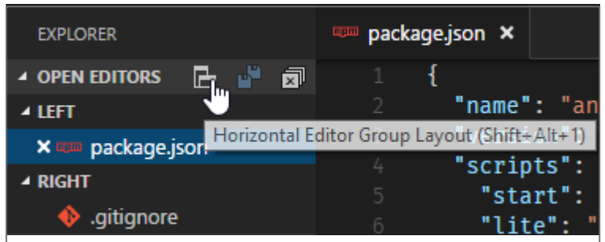


Figure 5: Switch between vertical and horizontal group layout

Editing Text

Being foremost a programming editor, Visual Studio Code includes all the standard functionalities you would expect:

- Bracket matching (`Ctrl+Shift+\`)
- Code block commenting (`Ctrl+K Ctrl+C`) and uncommenting (`Ctrl+K Ctrl+U`)
- Smart selection growing (`Shift+Alt+right`) and shrinking (`Shift+Alt+left`)
- Code folding (`Ctrl+Shift+[`) and unfolding (`Ctrl+Shift+]`), including folding the full file to a specific level (`Ctrl+K Ctrl+0` to `Ctrl+K Ctrl+5`) and unfolding the full file (`Ctrl+K Ctrl+J`)
- Zooming the editor in (`Ctrl++`) and out (`Ctrl+-`)
- Find (`Ctrl+F` in file, `Ctrl+Shift+F` in workspace) and replace (`Ctrl+H` in file, `Ctrl+Shift+H` in workspace) with regular expression support (including referencing matched groups with `$1`, `$2`... when replacing) and filtering files by filename when searching in workspace (`Ctrl+Shift+J` when *Search* side bar is opened)
- Multi cursor editing with block selection (`Ctrl+Shift+Alt+cursor` or `Shift+Alt+click`), custom adding of cursors (`Alt+click`) and adding find results to selection (`Ctrl+D`)

For programming languages with language server support (built-in or extension-based), there are additional functionalities:

- IntelliSense code suggestions (automatically as you type and manually invoked with `Ctrl+Space`)
- Method parameter hints (automatically when you open parenthesis and manually invoked with `Ctrl+Shift+Space`)
- Go to symbol definition (`F12`, opens a new file or moves cursor in same file) and peek symbol definition (`Alt+F12`, opens inline view with full editing support, `Esc` closes it)
- Find all references (`Shift+F12` opens inline view with full editing support, `Esc` closes it)
- Rename symbol (`F2`)
- Code formatting (`Shift+Alt+F` to format the full file, `Ctrl+K Ctrl+F` to format the current selection only)

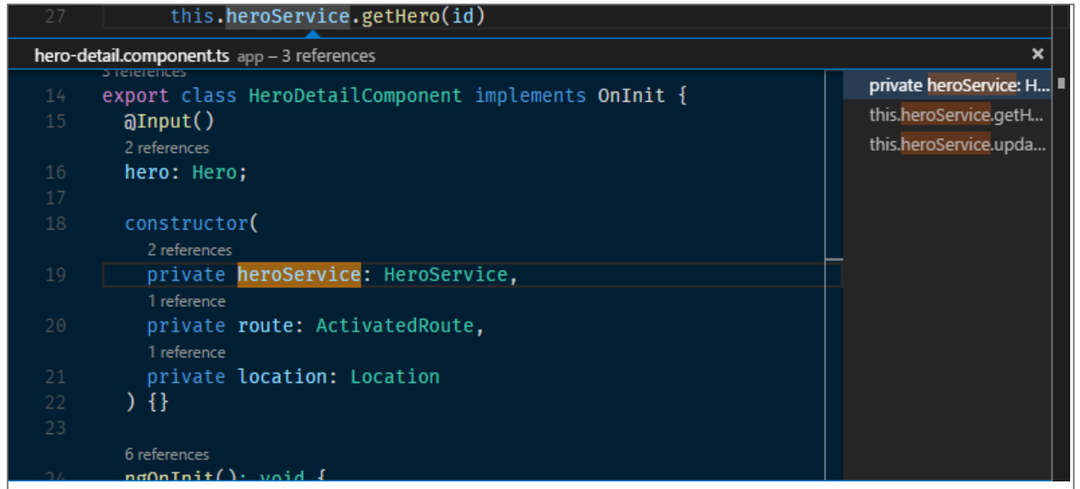


Figure 6: All symbol references in inline view

Hovering over a symbol shows context dependent information in a popup, e.g. namespace for a type, declaration for a variable or field, parameters and documentation for method, etc.

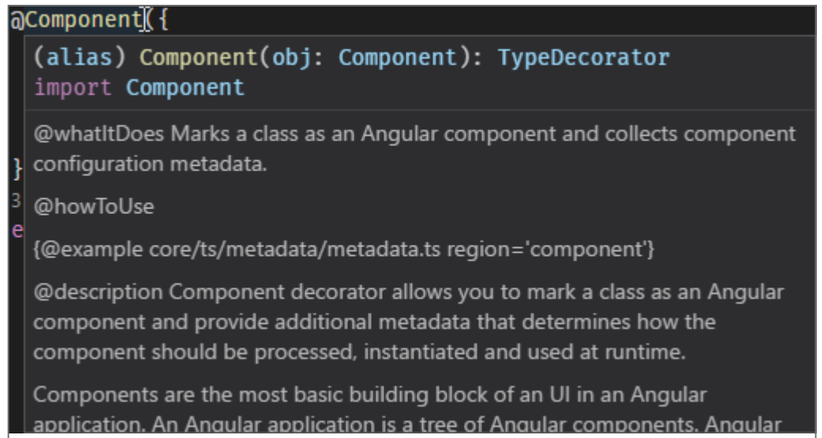


Figure 7: Popup for Angular Component declaration

Customizing the Editor

Extensions

Once you get familiar enough with the editor, you will want to customize it to your tastes. You can start by installing an extension. The *Extensions* side bar (`Ctrl+Shift+X`) will show the list of installed extensions when opened (initially empty). To find an extension to install, simply enter the search query in the input box or select one of the provided categories to list popular extensions.

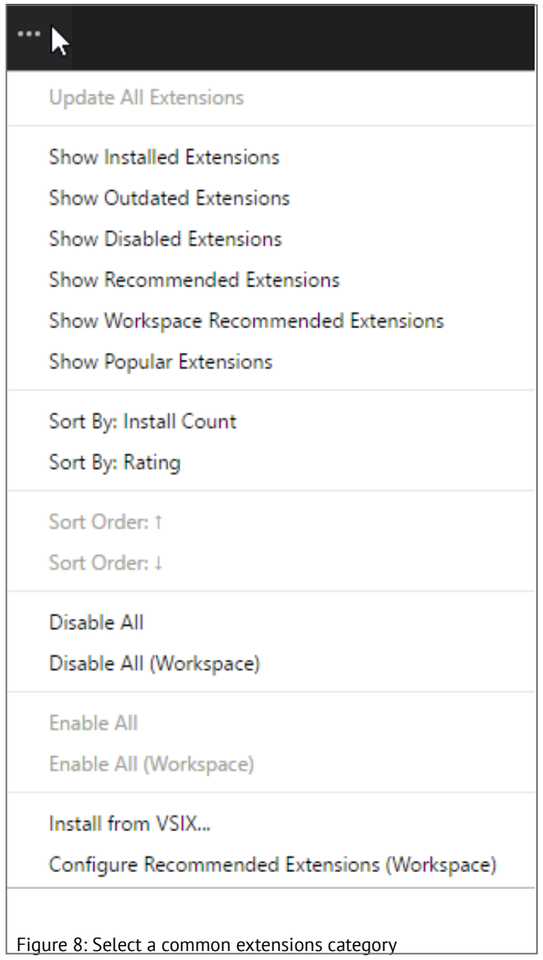


Figure 8: Select a common extensions category

Before installing an extension, you can review its details by clicking on it in the list:

- Its description by the author(s)
- A list of contributions it adds to the editor: settings, commands, debuggers, etc.
- Number of downloads and average rating as an estimate of its popularity and quality.

Enabling a newly installed or updated extension and disabling existing extensions will require you to restart the application.

There are many different categories of extensions and I will give more attention to many of them in the remainder of the article. To get you started, here are some of my favorite extensions, which I will not have an opportunity to mention later:

- Bookmarks allows you to add custom bookmarks to lines and quickly jump to them later.

- Contextual Duplicate adds a command to duplicate any text selection in the editor.

- Camel Case Navigation adds the ability to navigate through text by stopping at words separated by camel case humps, not only by whitespace or by punctuation.

Themes and Icons

There are two options for customizing the appearance:

- Color themes (*File* > *Preferences* > *Color Theme* menu or `Preferences: Color Theme` from *Command Palette*) allows you to change the colors used for the windows and text including syntax highlighting.

- File icon themes (*File* > *Preferences* > *File Icon Theme* menu or `Preferences: File Icon Theme` from *Command Palette*) allows you to add icons based on file and folder type to the *Explorer* side bar and elsewhere in the application.

Both commands provide a list of available themes with a live preview when you select them. There is also an option to search for additional themes in the extensions marketplace if none of the available alternatives suits you. My current favorites are the built-in *Dark+ (default dark)* color theme and *VS Code Icons* file icon theme from the vscode-icons extension. The latter is currently probably the largest collection of icons for Visual Studio Code. Unfortunately, it only works well with dark color themes.

Snippets

Snippets are a special type of item that appears in value completion lists (invoked with `Ctrl+Space`). Snippets are language dependent, and allow you to insert preconfigured custom blocks of code with optional tab stops and variables to modify the default values. Many snippets are already built-in. You can install some additional ones as extensions. To see which snippets are available for the language of your current file, you can invoke the `Insert Snippet` command from the *Command Palette*.

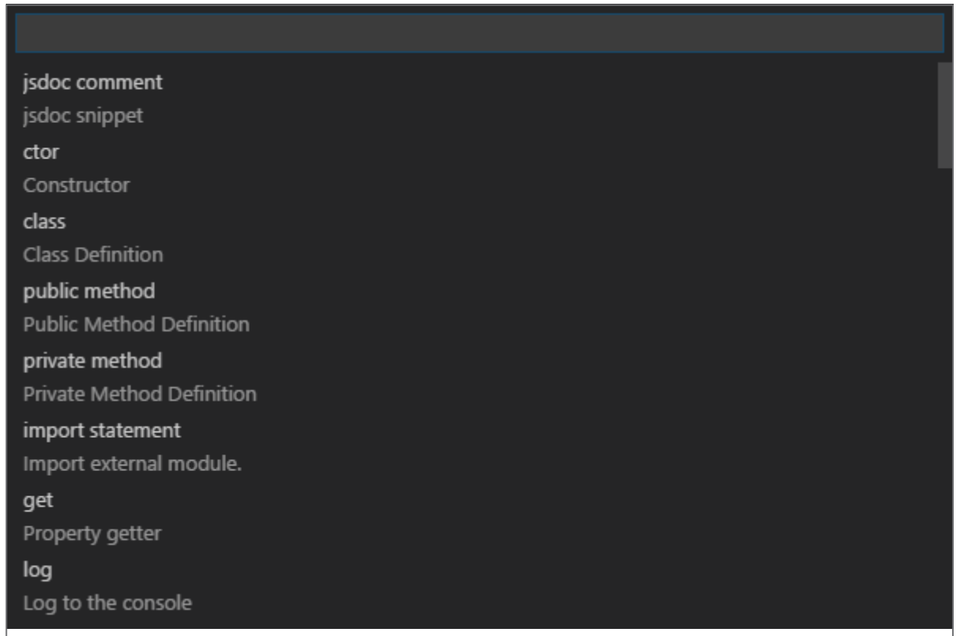


Figure 9: TypeScript snippet suggestions in Visual Studio Code

If you cannot find a snippet that suits your needs, you can also create your own. Open *File* > *Preferences* > *User Snippets* from the menu or invoke `Preferences: Open User Snippets` from the *Command Palette* and select the target language to open the dedicated JSON file for editing. Comments at the top of the file provide an explanation of the format and a sample snippet for the language, to get you started.

Keyboard Bindings

To modify keyboard bindings, open the configuration file from the *Command Palette* (`Preferences: Open Keyboard Shortcuts`) or from the menu (*File* > *Preferences* > *Keyboard Shortcuts*). The command will open two JSON files side by side: *Default Keyboard Shortcuts* with all built-in shortcuts defined in read-only mode, and `keybindings.json` with your customized shortcuts, which will be empty by default. In both files, you can use text search to find commands of interest. You can even use symbol search to navigate the files by keyboard shortcuts.

With `Ctrl+K Ctrl+K` you can invoke a simple wizard for adding a placeholder for a new shortcut. You will only need to modify the `command` value in most cases. To help you with that, there is value completion available (`Ctrl+Space`) for built-in commands. Alternatively, you can copy a keybinding definition from *Default Keyboard Shortcuts* and remove (by adding a '-' character in front of the `command` value) or add a shortcut (by changing the `key` value) for the command.

The `when` part of the keybinding definition specifies the condition, when the shortcut is valid. Usually, you will not need to modify it. Nevertheless, the supported values are well documented.

Extensions will typically include suggested keybinding definitions in their description, so that you can simply copy and paste them to your `keybindings.json` file. Visual Studio Code supports JSON comments in configuration files, which you can use to document your intentions:

```
[
  {
    // remove default binding
    “key”: “ctrl+d”,
    “command”: “-editor.action.addSelectionToNextFindMatch”,
    “when”: “editorFocus”
  },
  {
    // add alternative binding
    “key”: “ctrl+shift+d”,
    “command”: “editor.action.addSelectionToNextFindMatch”,
    “when”: “editorFocus”
  },
  {
    // rebind shortcut to an extension command
    “key”: “ctrl+d”,
    “command”: “lafe.duplicateCode”,
    “when”: “editorTextFocus”
  }
]
```

If you want to reconfigure the keybindings to match your favorite text editor, there are extensions available, which will do that for you. Search for `keymap` in *Extensions* side bar and you can choose from most current editors: ReSharper, Sublime, Atom, Emacs, and others. There is even a Vim extension available.

Application Settings

You can change general application settings by modifying the `settings.json` file. You can access it from the menu (*File* > *Preferences* > *User Settings*) or from the *Command Palette* (`Preferences: Open User Settings`). Similar to keyboard bindings, *Default Settings* file will open in read-only mode on the left, and the initially empty `settings.json` file will open on the right.

Again, you can enter new values to the latter by using value completion (`Ctrl+Space`) or by copying settings from the left file. They are neatly grouped together by the subsystem they affect or the extension they originate from. Of course, you can also search for specific text in the file.

Here are some of the settings you might want to change:

- Automatic saving of files, file formatting and whitespace trimming on save: `files.autoSave`, `files.autoSaveDelay`, `editor.formatOnSave`, `files.trimTrailingWhitespace`
- Indentation guides, rulers and visible whitespace: `editor.renderIndentGuides`, `editor.rulers`, `editor.renderWhitespace`
- Font ligatures (you will need to use a font, which supports ligatures, e.g. FiraCode): `editor.fontLigatures`, `editor.fontFamily`
- Positioning of snippets in the value suggestion popup: `editor.snippetSuggestions`
- Automatic updating of extensions: `extensions.autoUpdate`
- Shell to use in the integrated terminal (which can be invoked with `Ctrl+``): `terminal.integrated.shell.windows`. In Windows, you can use one of the following values, depending on your favorite shell:
  - `“C:\\WINDOWS\\system32\\cmd.exe”` to use the classic cmd shell
  - `“C:\\Windows\\system32\\WindowsPowerShell\\v1.0\\powershell.exe”` to use PowerShell
  - `“C:\\Windows\\system32\\bash.exe”` to use Bash if you installed Windows Subsystem for Linux in Windows 10
  - `“C:\\Windows\\SysWOW64\\cmd.exe”` for terminal.integrated.shell.windows and `[“/c”, “C:\\Program Files\\Git\\bin\\sh.exe”]` for terminal.integrated.shellArgs.windows to use Git Bash

There are many other settings available, though. You will want to do some exploration on your own.

Configuring the Workspace

Workspace settings

As the name implies, user settings are tied to the user profile. However, you can also configure settings at the workspace (i.e. folder) level and override the user settings. These settings will be stored in `.vscode\settings.json` file inside your folder. You can put the file in version control and share it with other developers working on the project. To create or edit the file, navigate to *File* > *Preferences* > *Workspace Settings* or invoke `Preferences: Open Workspace Settings` from the *Command Palette*. The file format is identical to the one for user settings.

Here are some good candidates for workspace settings:

- Files and folders to be hidden from the Explorer sidebar (e.g. `.git` and `node_modules`): `files.exclude`
- Code indentation settings: `editor.detectIndentation`, `editor.insertSpaces`, `editor.tabSize`
- Custom JSON schemas for files in the project: `json.schemas`

Version Control

Visual Studio Code has extensive built-in support for Git. However, you must have Git already installed on the machine. Most of the interaction with Git occurs in the *Git* side bar. It provides an overview of uncommitted changes in the repository with the following functionalities:

- View the changes in individual files
- Clean the changes
- Stage the changed file
- Enter the commit message, and commit the staged files (or all changed files, when none are staged)

If remotes are configured in the repository pull, push and sync commands are available as well. If any merge conflicts occur, basic syntax highlighting is available to help you resolve them. In the status bar, there is information about the current branch. By clicking on it, you can check out another branch.

Figure 10: Git information in status bar

In the marketplace, there are already many extensions available, which expand the existing Git support. Some of the most popular ones are:

- GitLens adds CodeLens information to code blocks as introduced in Visual Studio, and includes blame and history views.

- Git History (git log) implements a history graph with the ability to inspect details of each commit.
- Git Blame shows blame information about the current line in the status bar.

Other version control systems are not supported out of the box, but there are extensions available to provide basic support for most common ones: Mercurial, Subversion, Team Foundation Version Control, Perforce, etc.

Running Tasks

Most software development today includes some kind of building process, usually involving a task runner, such as Grunt or Gulp. Visual Studio Code has support for many task runners out of the box. The configuration is stored in `.vscode\tasks.json`. Trying to invoke the `Tasks: Run Build Task` command (`Ctrl+Shift+B` by default) will open a list of supported runners if the file does not exist yet.

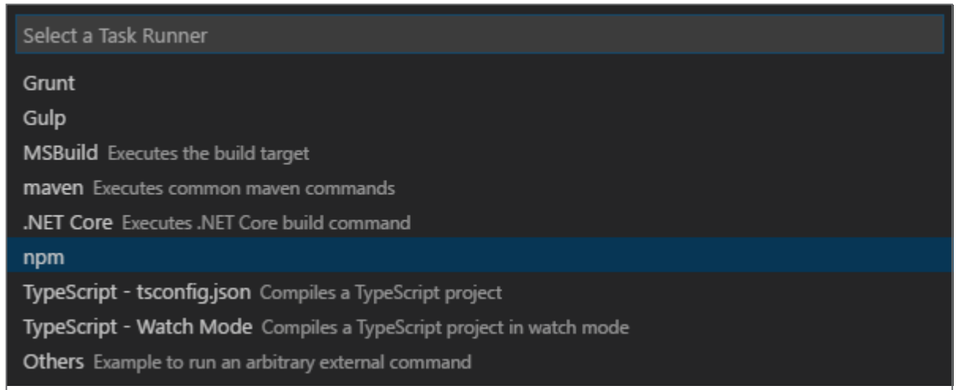


Figure 11: Task runner selection

Selecting one will generate a new file with default configuration for the runner of choice:

```
{
  // See https://go.microsoft.com/fwlink/?LinkId=733558
  // for the documentation about the tasks.json format
  “version”: “0.1.0”,
  “command”: “npm”,
  “isShellCommand”: true,
  “showOutput”: “always”,
  “suppressTaskName”: true,
  “tasks”: [
    {
      “taskName”: “install”,
      “args”: [“install”]
    },
    {
       “taskName”: “update”,
       “args”: [“update”]
    },
    {
       “taskName”: “test”,
       “args”: [“run”, “test”]
    }
  ]
}
```

The format is simple enough to understand. The generated file even includes a link to the official documentation, which you can consult for a detailed explanation. Nevertheless, a couple of things regarding this functionality are still worth mentioning:

- You can add a `problemMatcher` property to a task to enable parsing of its output. This will add the parsed errors and warnings to the *Problems* pane inside Visual Studio Code (`Ctrl+Shift+M`). Clicking individual entries will open the related file at the offending line to help you locate the issue. As always, value completion will suggest available built-in matchers. You can parse other output formats with regular expressions.

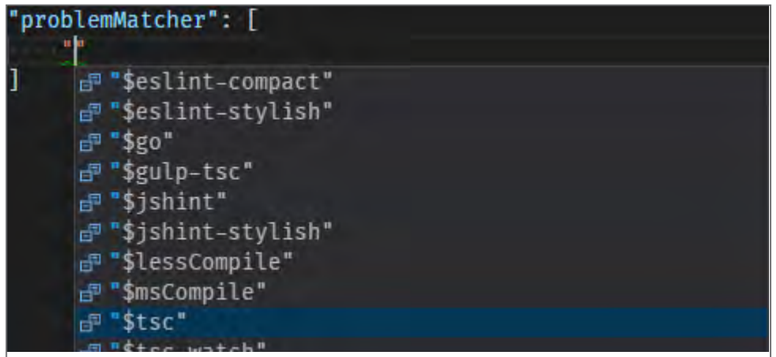


Figure 12: Problem matcher configuration

- You can use `isBuildCommand` and `isTestCommand` to mark a task as build, or test task. You will be able to run these two tasks using `Tasks: Run Build Task` and `Tasks: Run Test Task`, respectively. For the other tasks, you will need to invoke `Tasks: Run Task` command and select the task from the list. Alternatively, you can install vscode-status-bar-tasks extension to add the tasks as buttons to the status bar.

- Visual Studio Code will automatically detect tasks from Gulp, Grunt and Jake, and make them available in *Command Palette* even without specifying them in `tasks.json`. You will still need the file if you want to configure problem matchers or otherwise customize the tasks, though. Task Master extension will expand this automatic task detection to other formats, such as NPM, PowerShell, etc.

- Although you can have as many tasks as you want in a workspace, the `tasks.json` file format requires all of them to use the same task runner (specified as the `command` value). As a workaround for using more than one, you will need to use shell as `command`, or write a custom script which will delegate the calls to different task runners based on its arguments.

Debugging

Debugging configuration is similar to task runner configuration: `Debug: Start Debugging` command (`F5` by default) will open a list of supported debuggers if debugging is not configured yet. The number of debuggers available will depend on the extensions installed.

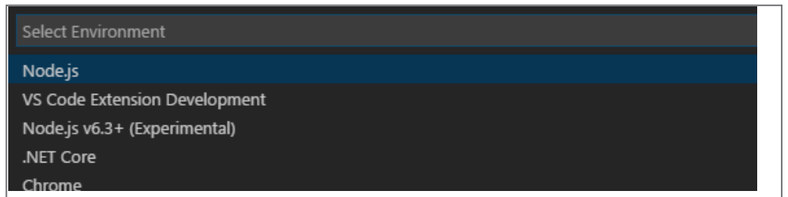


Figure 13: Debugger selection

Selecting one will initialize a new `.vscode\launch.json` configuration file:

```
{
  // Use IntelliSense to learn about possible Node.js debug attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "Launch Program",
      "program": "${workspaceRoot}/app.js",
      "cwd": "${workspaceRoot}"
    },
    {
      "type": "node",
      "request": "attach",
      "name": "Attach to Process",
      "port": 5858
    }
  ]
}
```

The link will again bring you to the official documentation of the format. However, in the case of debuggers, the available options differ a lot between debuggers, therefore you will need to consult their own documentation or make do with the sample file each one generates, and the completion suggestions.

Unlike task runners, you can configure multiple debuggers in a single workspace. The only downside is that you will not get the sample configuration for them, when `launch.json` file already exists. To work around it, you can temporarily rename the file and then manually merge in the contents of the newly generated file.

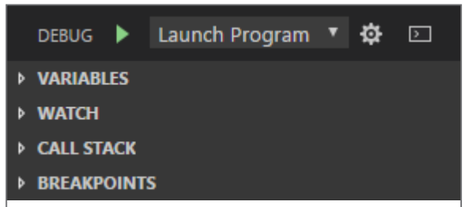


Figure 14: Debug side bar

The *Debug* side bar is the common interface of all installed debuggers. At the top, you can select a configuration and start debugging it. Below it are four sections:

- In *Variables* section, you can inspect the values found in the current stack frame.
- In *Watch* section, you can add your own variables to inspect their values.
- *Call Stack* section shows the current call stack and allows you to select a different stack frame to inspect.
- *Breakpoints* shows a list of all breakpoints set, allowing you to temporarily disable or delete them.

You can set a breakpoint by pressing F9 in a source code line or by clicking in front of its line number. Right clicking the red dot indicating the breakpoint will give you the option to set a condition when the breakpoint will hit, based either on an expression value or on the hit count. Support for this and other debugger features can vary from debugger to debugger. The built-in Node.js debugger will support all features, while other debuggers might not have yet implemented all of them.

Languages Support

Visual Studio Code only has built-in support for web development, including but not limited to:

- Languages with syntax highlighting, suggestions and snippets: JavaScript, TypeScript and CoffeeScript
- HTML with Emmet (formerly Zen Coding) support
- CSS, Sass and Less
- Node.js debugger

Many extensions can further improve the experience:

- Debuggers and more: Debugger for Chrome, Cordova Tools, React Native Tools, NativeScript
- Linters: TSLint, ESLint, jshint, stylelint
- Snippets for frameworks: Angular 2 TypeScript Snippets, AngularJs 1.x Code Snippets, Reactjs code snippets, Aurelia Snippets, ionic 2 Commands with Snippets, ionic1-snippets
- CodeLens visualizers: TypeLens, Version Lens

Thanks to extensions, Visual Studio Code is not limited to web development either. You can easily use it for development in many other languages:

- .NET: C#, Ionide-fsharp, Debugger for Unity
- PowerShell: PowerShell
- Java: Language support for Java™ for Visual Studio Code
- Python: Python
- Go: Go
- PHP: PHP Debug, PHP IntelliSense, PHP Code Format
- Ruby: Ruby
- ...

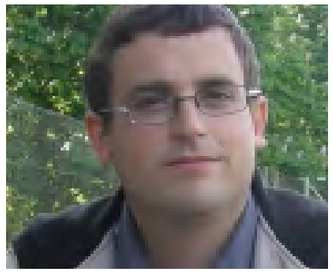
Conclusion:

I was not an early adopter of Visual Studio Code, but since I have started using it, it has grown on me. Of course, I still regularly use integrated development environments, such as Visual Studio and IntelliJ IDEA. However, Visual Studio Code is close to being the only text editor I am using on a regular basis. It has also become my tool of choice for client-side web development because of its small footprint and great debugging support.

I hope that after reading this article you will want to try it out, if you have not done so already. Who knows, may be this awesome code editor will grow on you as well!

Microsoft® MVP Most Valuable Professional

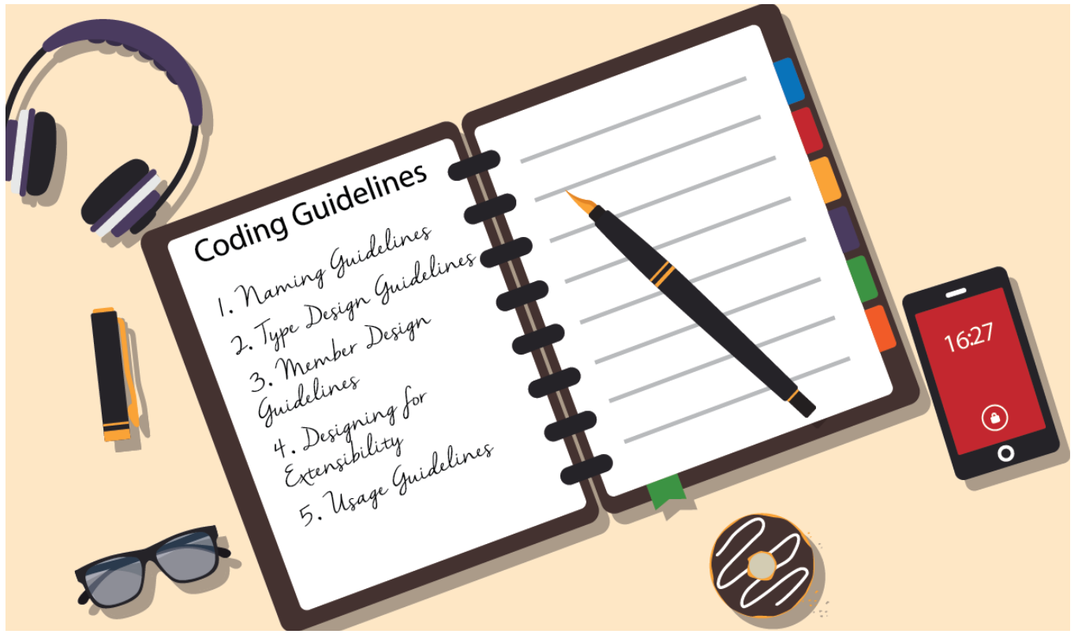
Damir Arh
Author



Damir Arh has many years of experience with Microsoft development tools; both in complex enterprise software projects and modern cross-platform mobile applications. In his drive towards better development processes, he is a proponent of test driven development, continuous integration and continuous deployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and answering questions on Stack Overflow. He is an awarded Microsoft MVP for .NET since 2012.

Thanks to Daniel Garcia and Suprotim Agarwal for reviewing this article.

Craig Berntson



CODING GUIDELINES IMPORTANT FOR EVERYONE

For years, I have used a tag line as part of my signature on online forums. It says, *"Code as if the person that will modify your code is a crazy lunatic that knows where you live. That person is probably you."* I don't know who to attribute this saying to, but adhering to it will help you and your team members from going insane.

Thanks to Damir Arh and Suprotim Agarwal for reviewing this article.

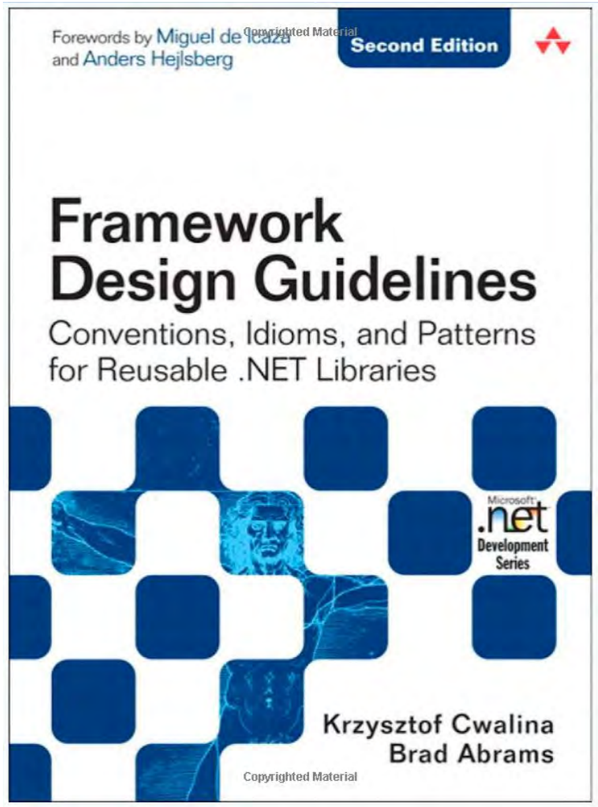
The primary thing this tag line is telling us is to use good coding guidelines. Note that I don't use the term "coding standards". I do this for a reason. **A standard is a rule. A guideline is a recommendation**. Those are very different things. A rule is strict and shouldn't be broken. A guideline is more of a suggestion, albeit, a strong suggestion.

What exactly is a coding guideline? It's a group of *suggestions* on how to write code. Some people say that if there is more than one person writing code on a project, then you should have guidelines. But if you reread the quote in the first paragraph above, you'll see that coding guidelines also apply to the solo developer. They help keep you sane when you look back at the code you wrote a year, a month, or even a week ago. You've seen this code. It makes you say, "Who wrote this?" and then you realize it was you.
Each line of code is written once and read many times. Not read by the computer, but read by a human. You should be able to understand each line of code by reading it, not by reading comments. Coding guidelines help us with this by specifying things like the amount of white space, how to name variables and classes, or what makes a proper comment.

Coding guidelines also help ensure that code does its job well, is easy to maintain, and easy to debug.

Framework Design Guidelines

One of the most comprehensive guidelines for .NET code is the "*Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*". Originally published as a book (and still available in book form), it is now also available for free at https://msdn.microsoft.com/en-us/library/ms229042(v=vs.110).aspx.



Originally published as a guide for Microsoft to create the .NET Framework, these guidelines primarily target developers creating libraries for other developers to consume. But they also contain good information that you can use in your everyday work.

The guidelines are divided into seven different categories. Each category is in turn broken down into several guidelines. Each one could have suggestions such as do this, don't do that, avoid that, or consider that. Some also have conflicting advice from one of the authors, which emphasizes these are not standards.

Here are the guidelines you'll find:

Naming Guidelines – These guidelines give ideas on how to name assemblies, namespaces, types, and members in class libraries. They are further broken down into Capitalization Conventions, General Naming Conventions, Names of Assemblies and DLLs, Names of Namespaces, Names of Classes, Structs, and Interfaces, Names of Type Members, Naming Parameters, Naming Resource.

Example: Names of Namespaces – DO prefix namespace with a company name to prevent namespaces form different companies from having the same name. DO NOT use organizational hierarchies as the basis for namespace hierarchies. CONSIDER using plural namespace names where appropriate (System.Collections instead of System.Collection).

Type Design Guidelines – These guidelines cover several areas such as Choosing Between Class and Struct, Abstract Class Design, Static Class Design, Interface Design, Struct Design, Enum Design, and Nested Types.

Example: Interface Design – DO define an interface if you need some command API to be supported by a set of types that includes value types. CONSIDER defining an interface if you need to support its functionality on types that already inherit from some other type. AVOID using interfaces with no members.

Member Design Guidelines – Suggestions for Member Overloading, Property Design, Constructor Design, Event Design, Field Design, Extension Design, Operator Overloads, and Parameter Design.

Example: Field Design – DO NOT provide instance fields that are public or protected. DO use constant fields for constants that will never change.

Designing for Extensibility – Covers Unsealed Classes, Protected Members, Events and Callbacks, Virtual Members, Abstractions, Base Classes for Implementing Abstractions, and Sealing.

Example: Events and Callbacks – CONSIDER using callbacks to allow users to provide custom code to be executed by the framework. AVOID using callbacks in performance-sensitive APIs.

Design Guidelines for Exceptions – Recommendations for Exception Throwing, Using Standard Exception Types, Exceptions and Performance.

Example: Exception and System Exception – DO NOT throw System.Exception or System.SystemException. AVOID catching System.Exception or System.SystemException except in top-level exception handlers.

Usage Guidelines – Suggestions for working with Arrays, Attributes, Collections, Serialization, System.Xml Usage, and Equality Operators.

Example: Guidelines for Collections – DO NOT use weakly typed collections in public APIs. DO NOT use Hashtable or Dictionary<Tkey, Tvalue> in public APIs.

Common Design Patterns – Recommendations for Dependency Properties and Dispose Pattern.

Example: Dispose Pattern – DO implement the Basic Dispose Pattern on types containing instances of disposable types. CONSIDER implementing the Basic Dispose Pattern on classes that themselves don't hold unmanaged resources or disposable objects but are likely to have subtypes that do.

The guidelines themselves go into explanations about each of the recommendations. These should be required reading for all .NET developers, but at a minimum, you should be familiar with where to find the guidelines for when you need a bit of help with areas they cover.

Application Guidelines

At the other end of the spectrum from Framework Design Guidelines are guidelines that more directly target every day application development. They don't provide the detail of the Framework Design Guidelines (which weigh in at 480 pages), but do cover some of the same topics.

There is no one set of guidelines here. In fact, a web search on ".NET coding guidelines" returned over 17 million results. A number that large makes it difficult to decide which guideline to use. There is one in the search results that I've used in the past. You can find the Aviva Solutions guidelines ready to use at https://csharpguidelines.codeplex.com/. Community suggestions are also accepted.

Like the Framework Design Guidelines, the Aviva guidelines cover a number of areas. However they don't go into the same amount of detail.

Class Design Guidelines – Suggestions on how to name and design a class. Examples: A class or interface should have a single purpose. Use an interface to decouple classes from each other.

Member Design Guidelines – Class members are things like properties and methods. These guidelines help you use them better. Examples: Allow properties to be set in any order. A method or property should do only one thing.

Miscellaneous Design Guidelines – Things that don't fit in the other categories. Examples: Throw the most specific exception that is appropriate. Evaluate the result of a LINQ expression before returning it.

Maintainability Guidelines – Maintainability is one of the primary reasons to have coding guidelines. This section has sage advice on things you should do to make your code maintainable. Examples: Don't use "magic" numbers. Declare and initialize variables as late as possible.

Naming Guidelines – Naming a property, method, class, variable, etc is one of the hardest things we do as developers. Here you get suggetions on how to pick good names. Examples: Name a member, parameter, or variable according to its meaning and not its type. Name methods using verb-object pair.

Performance Guidelines – Sometimes performance isn't all that critical to your application. But when it is, refer to this section. Examples: Only use async for low-intensive long-running activities. Beware of mixing up await/async with Task.Wait.

Framework Guidelines – Are you building libraries and frameworks for others? Even if you're not, you'll make your own code better. Examples: Don't hardcode strings that change based on the deployment. Only use the dynamic keyword when talking to a dynamic object.

Documentation Guidelines – Wait, do we really need documentation in today's Agile world? Well, yes, especially if other developers will consume our code. Examples: Write MSDN style documentation. Avoid inline comments.

Layout Guidelines – Laying out your code in a logical consistent manner aids readability. Examples: Order and group namespaces according to the company. Be reluctant with #region.

The Aviva Solutions guidelines total 33 pages at the time I'm writing this. A far cry from Framework Design Guidelines. This makes them much easier to adopt and target applications, something most of us work on every day.

Adopting Guidelines

At this point, you may be wondering how to adopt guidelines for your team. It's quite easy. First, your team needs to agree on a set of guidelines. I recommend you adopt an already existing set rather than adopt your own.

Once you have your guidelines in place, make sure you understand them. At first, you'll get some things wrong. Don't worry about this. Over time, you'll get better at following your guidelines.

Conduct informal code reviews. This allows other team members to help find issues and things that violate the guidelines. If you use TFS for version control, you can use the Request Code Review feature. If you use Git, then use Pull Requests. If you're already doing code reviews then you may want to add guidelines compliance. (Note that I'll discuss code reviews in more detail in my next column)

Finally, read (yes, I'm suggesting you read them) and be familiar with the Framework Design Guidelines, even for application development. Have them bookmarked in your browser so you can easily refer to them.

Visual Studio Tooling

There are several tooling options for Visual Studio that can assist with writing better code. These tools not only help you follow some coding guidelines but they also run static code analysis to help pinpoint other potential issues. Before talking about specific tools, I should probably discuss static code analysis.

Static Code Analysis

All the testing we do on code - unit, function, performance, and others, all test against code as it is executed. Static Code Analysis looks at the actual code itself to find possible errors that may not manifest themselves until runtime.

How this works from inside Visual Studio is that as you write code, it is compiled in the background and then the IL code is analyzed for these potential issues. You are probably familiar with FXCop. This tool did it's analysis externally from Visual Studio. There are now better tools as we'll see in a moment.

These tools use a set of configurable rules that typically look at things such as:

Design – Rules are applied to detect potential design flaws. These types of errors typically do not affect the execution of your code.

Globalization – Rules that detect missing or incorrect usage of information related to globalization and localization.

Naming – Detection of things like incorrect casing, cross language keyword collisions (say between C# and VB), names of types, members, parameters, namespaces, etc.

Performance – Detect elements in assemblies that will degrade performance.

Security – Identifies areas in your assemblies that could be vulnerable to malicious users or code.

Usage – Rules that detect potential flaws in your assemblies that can affect code execution.

Maintainability – These rules detect maintenance issues.

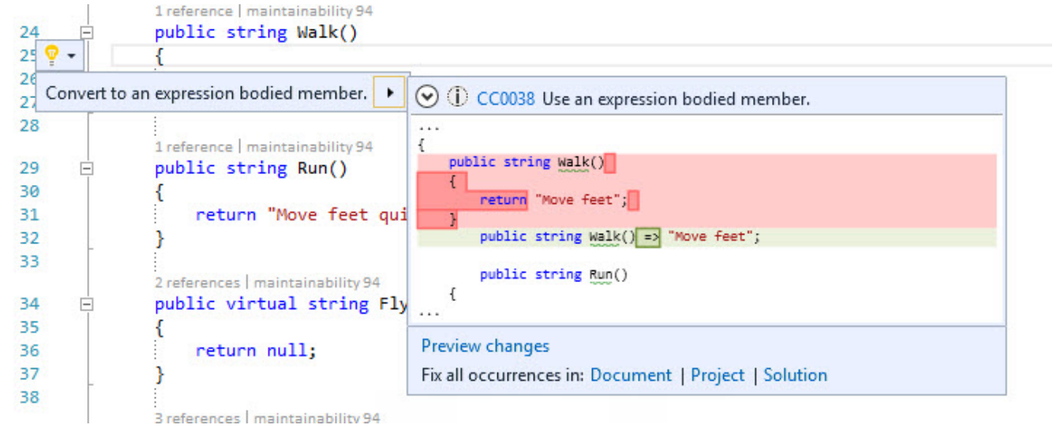
Reliability – Rules that detect incorrect thread or memory usage.

Many of the rules in these tools are based on Microsoft's Framework Design Guidelines.

Getting back to Visual Studio

Now, getting back to Visual Studio, let's discuss some of these tools. You're probably already familiar with Resharper so I won't spend any time discussing. But there is feature that was new in Visual Studio 2015 called *Diagnostic Analyzers* and are made possible by the awesomeness of the Roslyn compiler.

To install a Diagnostic Analyzer, select Tools -> Extensions and Updates then make sure Online is selected. Perhaps the most recommended analyzer is Code Cracker for C#. So, type Code Cracker into the search box then install it. However, if you are using SonarQube to do any analysis, I recommend you select SonarLint instead.



All these tools work by displaying squiggles under code that is suspect and providing a way to look at options to fix the code. The beauty of how they work is that it just happens as you type code, making the entire process quite painless. In the screenshot, the analyzer identified an issue. The suggestion dialog was displayed when I clicked on the lightbulb.

Wrap up

Having a coding standard will help ensure your code feels familiar and is easier to debug and maintain, even when written by different developers. I can tell you from experience, that guidelines work. I recommend them for teams of every size, even one-man shops.

Using static analysis tools such as Resharper or diagnostic analyzers will take your coding standards to the next level.

Coding guidelines are just one of the many tools you should have available in your toolshed. By using them, you will not become a crazy lunatic and help ensure that your code is lush, green, and vibrant.

About Software Gardening

Comparing software development to constructing a building says that software is solid and difficult to change. Instead, we should compare software development to gardening as a garden changes all the time. Software Gardening embraces practices and tools that help you create the best possible garden for your software, allowing it to grow and change with less effort ■

Craig Berntson

Author

MVP Microsoft® Most Valuable Professional

Craig Berntson works for one of the largest mortgage companies in the US where he specializes in middleware development and helping teams get better. He has spoken at developer events across the US, Canada, and Europe for over 20 years and is a Grape City Community Influencer. Craig is the coauthor of 'Continuous Integration in .NET' available from Manning. He has been a Microsoft MVP since 1996. Craig lives in Salt Lake City, Utah. Email: dnc@craigberntson.com Twitter: @craigber

Benjamin Jakobus

Migrating from Bootstrap 3 to 4

Thanks to Gil Fink and Suprotim Agarwal for reviewing this article.

Bootstrap has become the world's favorite framework for building responsive web UI. In the last edition of DNC Magazine, we took a more detailed look into the newest Bootstrap release - Bootstrap 4 Alpha - and discussed what the project has to offer. In this article, we will look into a migrating scenario from simple Bootstrap 3 website to Bootstrap 4, step by step.

Introduction

In a previous article, "Exploring Bootstrap 4", we briefly touched upon the major differences and new features that come about with the latest release of Bootstrap. To recap: Bootstrap 4 is a complete re-write of its predecessor - as such, it comes with a number of breaking changes, including:

- Dropping support for panels, wells and thumbnails
- The replacement of the badge component with labels
- The addition of a new grid tier
- Favoring root em (rem) over pixels (px) as the preferred typographic measurement
- Class name changes for the nav bar, carousel, dropdown and pagination.

(Note: A complete list of changes can be found here: https://v4-alpha.getbootstrap.com/migration/)

With these changes in mind, this article hopes to serve as a (non-exhaustive) step-by-step guide for migrating an existing Bootstrap 3 website to Bootstrap 4. To this end, we will be utilizing a small, single-page, demo website built using sample snippets provided by the official Bootstrap 3 documentation. Figure 1 below illustrates the landing page of this website. It is composed of a top nav bar, a carousel, an alert and a grid using panels, followed by a pagination component.

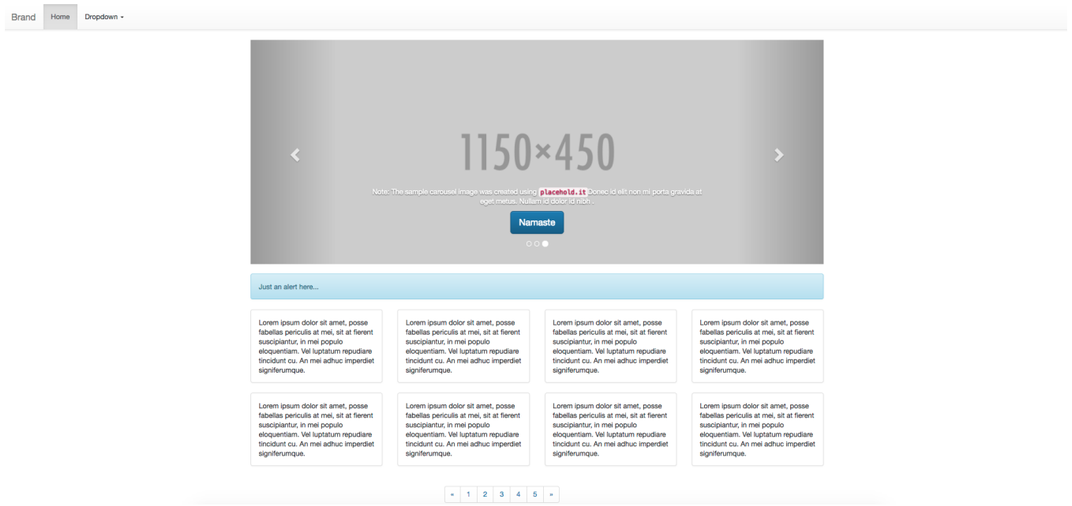


Figure 1: The landing page of our demo website, built using sample snippets found in the official Bootstrap 3 documentation.

Adding Bootstrap 4

The first step on our migration journey is to replace the Bootstrap 3 reference (stylesheet and JavaScript file) with the Bootstrap 4 equivalent. To do so, first install Bootstrap 4 using bower install bootstrap#v4.0.0-alpha.4. This should create a new folder, bower_components, in your working directory.

Replace the Bootstrap 3 stylesheet reference with:

```
<link rel="stylesheet"
href="bower_components/
bootstrap/dist/css/
bootstrap.min.css" />
```

Similarly, replace the Bootstrap 3 JavaScript reference with:

```
<script src="bower_
components/bootstrap/dist/
js/bootstrap.min.js"></
script>
```

Instead of bower, one can also use npm (`npm install bootstrap@4.0.0-alpha.5`), or composer (`composer require twbs/bootstrap:4.0.0-alpha.5`) to install Bootstrap.

Alternatively, as per the Bootstrap 4 documentation, you can use the Bootstrap 4 CDN:

```
<link rel="stylesheet"
href="https://maxcdn.
bootstrapcdn.com/
bootstrap/4.0.0-alpha.5/
css/bootstrap.min.css"
```

```
integrity="sha384-AysaV+vQoT3kOAXZkl02PThvDr8HYKPZhNT5h/CXfBThSRXQ6jW5DO2ekP5ViFdi"
crossorigin="anonymous">

<script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-alpha.5/js/bootstrap.
min.js" integrity="sha384-BLiI7JTZm+JWlgKa0M0kGRpJbF2J8q+qreVrKBC47e3K6BW78kGLrCkeR
X6I9RoK" crossorigin="anonymous"></script>
```

Save and refresh. Our demo page should now appear broken, as shown in figure 2.

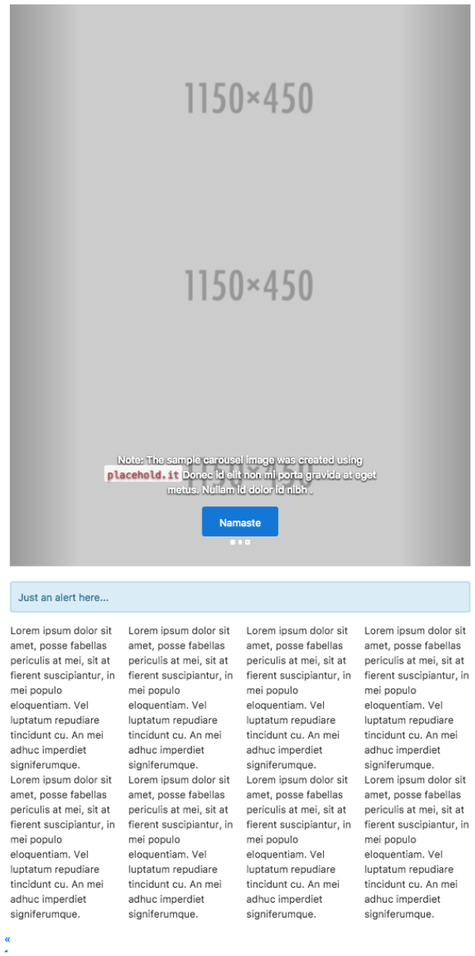


Figure 2: The landing page of our demo website rendering incorrectly after including Bootstrap 4

Fixing the navigation bar

First, let's look into fixing the navigation bar. Our Bootstrap 3 nav bar is defined as follows:

```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <a class="navbar-brand" href="#">Brand</a>
    </div>
    <div class="collapse navbar-collapse" id="navbar-collapse">
      <ul class="nav navbar-nav">
        <li class="active"><a href="#">Home <span class="sr-only">(current)
        </span></a></li>
        <li class="dropdown">
          <a href="#" class="dropdown-toggle" data-toggle="dropdown"
          role="button" aria-haspopup="true" aria-expanded="false">Dropdown <span
          class="caret"></span></a>
          <ul class="dropdown-menu">
            <li><a href="#">Item 1</a></li>
            <li><a href="#">Item 2</a></li>
            <li><a href="#">Something else here</a></li>
            <li role="separator" class="divider"></li>
            <li><a href="#">Item 3</a></li>
          </ul>
        </li>
      </ul>
    </div>
  </div>
</nav>
```

Bootstrap 4 greatly simplifies the navbar. However, this simplification will require us to make several changes to our markup, in order to get the navbar working again. Firstly, we no longer need the div, wrapping the navbar contents, to which the `container-fluid` class is applied, because navbars are now fluid by default. We also no longer need to explicitly specify that the navbar items are collapsible. Hence, we can remove the `div` to which the class collapse `navbar-collapse` was assigned. Last but not least, we also no longer require the `navbar-header` class, so we can also remove that element.

Bootstrap 4 also requires that a color scheme be specified for the navbar[1] . Let's go ahead and apply navbar `navbar-light bg-faded` to the `nav` element.

Any navbar list item now requires the `nav-item` class. Similarly, any links must have the nav-link class applied to them. Both the `nav-item` and `nav-link` classes ensure that the display type is set to `inline-block`.

Finally, we must also apply the `dropdown-item` to all dropdown items (more on this later).
Our Bootstrap 4 navigation markup now looks as follows:

```
<nav class="navbar navbar-light bg-faded">
  <a class="navbar-brand" href="#">Brand</a>
    <ul class="nav navbar-nav">
      <li class="nav-item active"><a class="nav-link" href="#">Home
      <span class="sr-only">(current)</span></a></li>

      <li class="nav-item dropdown">
        <a href="#" class="nav-link dropdown-toggle" data-toggle="dropdown"
        role="button" aria-haspopup="true" aria-expanded="false">Dropdown
        <span class="caret"></span></a>
    <ul class="dropdown-menu">
      <li class="dropdown-item"><a href="#">Item 1</a></li>
        <li class="dropdown-item"><a href="#">Item 2</a></li>

          <li class="dropdown-item"><a href="#">Something else here</a></li>
          <li role="separator" class="divider"></li>
```

[1] For more information on color schemes see https://v4-alpha.getbootstrap.com/components/navbar/#color-schemes

```
        <li class="dropdown-item"><a href="#">Item 3</a></li>
      </ul>
    </li>
    </ul>
  </nav>
```

As can be seen above, the markup required for the Bootstrap 4 navbar is much flatter and simpler, causing it to render faster, and making it easier to parse by the reader.

Replacing panels

As noted in the introduction, Bootstrap 4 dropped support for panels, wells and thumbnails. Instead, developers should make use of a new Bootstrap 4 concept: *cards*. Cards aim to merge the three aforementioned components into one, flexible, component that supports various types of content at the same time. For example, cards support images, lists, footers and headers, and as such are much less restrictive than their individual counter-parts.

Our demo website defines a series of panels within a 4x2 grid. The markup for each panel is as follows:

```
<div class="panel panel-default">
  <div class="panel-body">
    Lorem ipsum dolor sit amet, posse fabellas periculis at mei, fierent
    suscipiantur, in mei populo eloquentiam. Vel luptatum repudiare tincidunt cu.
    An mei adhuc imperdiet signiferumque.
  </div>
</div>
```

We will go ahead and replace each occurrence of these panels with a card. The simplest form of card is defined (figure 3):

```
<div class="card">
  <div class="card-block">
    <p class="card-text">
      Lorem ipsum dolor sit amet, posse fabellas periculis at mei, sit at fierent
      suscipiantur, in mei populo eloquentiam. Vel luptatum repudiare tincidunt
      cu. An mei adhuc imperdiet signiferumque.
    </p>
  </div>
</div>
```

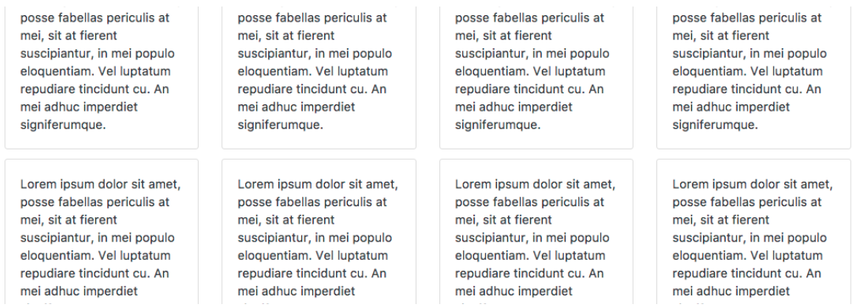


Figure 3: The simplest form of cards, containing only a card text, and no title or image.

Since cards offer various types of content, we could add a title, an image and some links, using the `card-title`, `card-img-*` and `card-link` classes respectively (figure 4):

```
<div class="card">
  <img class="card-img-top" src="beach.jpg" alt="Card image cap">
    <div class="card-block">
     <h4 class="card-title">Surf it up!</h4>
     <p class="card-text">
      Lorem ipsum dolor sit amet, posse fabellas periculis at mei, sit at fierent
      suscipiantur, in mei populo eloquentiam. Vel luptatum repudiare tincidunt cu.
      An mei adhuc imperdiet signiferumque.
     </p>
    </div>

    <div class="card-block">
      <a href="#" class="card-link">Stoked!</a>
      <a href="#" class="card-link">Hang lose!</a>
    </div>
</div>
```

Updating pagination

Bootstrap 4 introduced two new classes to use for pagination:

1. `page-item` which adjusts the display to inline, and sets the margins of the first and last children.

2. `page-link` which adjusts the border, position, margin and text decoration of an anchor element.

As such, we must update the Bootstrap 3 pagination markup, from:

```
<nav aria-label="Page navigation">
 <ul class="pagination">
  <li>
   <a href="#" aria-label="Previous">
    <span aria-hidden="true">&laquo;
    </span>
   </a>
  </li>
  <li><a href="#">1</a></li>
  <li><a href="#">2</a></li>
  <li><a href="#">3</a></li>
  <li><a href="#">4</a></li>
  <li><a href="#">5</a></li>
  <li>
   <a href="#" aria-label="Next">
    <span aria-hidden="true">&raquo;
    </span>
   </a>
  </li>
 </ul>
</nav>
```

to:

```
<nav aria-label="Page navigation">
 <ul class="pagination">
  <li class="page-item">
   <a class="page-link" href="#" aria-label="Previous">
    <span aria-hidden="true">&laquo;</span>
   </a>
  </li>
  <li class="page-item"><a class="page-link" href="#">1</a></li>
  <li class="page-item"><a class="page-link" href="#">2</a></li>
  <li class="page-item"><a class="page-link" href="#">3</a></li>
  <li class="page-item"><a class="page-link" href="#">4</a></li>
  <li class="page-item"><a class="page-link" href="#">5</a></li>
  <li class="page-item">
   <a href="#" aria-label="Next" class="page-link">
    <span aria-hidden="true">&raquo;</span>
   </a>
  </li>
 </ul>
</nav>
```

Updating the carousel

The changes to the carousel code are minor. In Bootstrap 3, a carousel is defined through the following markup:

```
<div class="carousel-inner" role="listbox">
  <div class="item active">
    <img class="first-slide" src="http://placehold.it/1150x450" alt="First slide">
    <div class="container">
      <div class="carousel-caption">
        <p>Note: The sample carousel image was created using <code>placehold.it
        </code>Donec id elit non mi porta gravida at eget metus. Nullam id dolor id
        nibh .</p>
        <p><a class="btn btn-lg btn-primary" href="#" role="button">Click me!</a>
        </p>
      </div>
    </div>
  </div>
  <!-- Other carousel items here -->
</div>
```

The only changes required are

i) Renaming the item class to `carousel-item`

ii) Replacing the Glyphicon classes with `icon-prev` and `icon-next`

Glyphicons

Although the demo project discussed in this article does not use of Glyphicons, and important change to point out is the removal of Glyphicons from Bootstrap 4. As a result of this decision, Bootstrap 4 may be more light-weight, and may have a smaller footprint than its predecessor. This means that icon sets now need to be included manually - which is advantageous as one is now no longer coupled to the default Bootstrap icon set.

Updating Dropdowns

As already noted previously, Bootstrap 4 dropdown list items are now required to have the dropdown-item class applied to them. Furthermore, the dropdown menu is no longer required to be an anchor tag - instead, one can use buttons. Aside from these two changes, the classes used to define a dropdown menu remain the same for Bootstrap 4.

The Grid System

The most striking change to the Bootstrap grid system is the introduction of a new grid tier, which contains a breakpoint at 480px. This gives Bootstrap 4, four grid tiers, instead of 3: xs, sm, md, and lg.
The addition of the sm grid tier means that Bootstrap now has five breakpoints:

- An extra-small breakpoint for handheld devices that boast a smaller screen than normal (0px)
- A small breakpoint aimed at phones / handhelds (544px)
- A breakpoint for screens of medium size, such as tablets (768px)
- A breakpoint for large screens - that is, desktops (992px)
- An extra-large breakpoint to support wide-screen (1200px)

Furthermore, the offsetting of columns has changed, from `.col-*-offset-**` to `.offset-*-**`. That is, `.col-md-offset-*` now is `.offset-md-*`. Likewise, the pushing of columns changed from `.col-push-*-*` to `.push-*-*`.

Conclusion

In this article, we took a brief look into migrating a simple, existing Bootstrap 3 website to Bootstrap 4. The website was constructed using various sample snippets encountered in the official Bootstrap 3 documentation. Although the overall task may have appeared daunting at first, the changes themselves were straight forward - with the most effort being expended by updating the navigation bar and replacing panels with cards. In summary, the major changes that one should watch out for are:

- The replacement of panels, wells and thumbnails with cards
- The renaming of classes and the introduction of a new grid tier
- The removal of Glyphicons
- The replacement of badges with labels
- The simplification of the navigation bar
- The introduction of additional classes around dropdowns and pagination

Benjamin Jakobus

Author

Benjamin Jakobus graduated with a BSc in Computer Science from University College Cork and obtained an MSc in Advanced Computing from Imperial College London. As a software engineer, he has worked on various web-development projects across Europe and Brazil.

Gouri Sohoni

AGILE DEVELOPMENT BEST PRACTICES

USING VISUAL STUDIO 2015 AND TEAM FOUNDATION SERVER 2015

This article will discuss best practices for Agile Development from a developer's perspective. While doing so, this article will discuss tools like Visual Studio 2015 and Team Foundation Server (TFS) 2015 that can help in Agile Development.

Note: Most of the functionality available with TFS 2015 is available with Visual Studio Team Services (VSTS).

Agile software development involves creating increments of working and demonstrable software within stipulated time period. It evolves through the collaborative efforts of a cross-functional motivated team. People think that there is no planning in agile, but contrary to the belief, Agile teams *do* plan features they are going to implement. But usually these plans are for the immediate future only. Agile process involves - development with new techniques and tools, on time delivery, finding bugs early, and having a positive approach for change.

We will not be delving into details of the Agile Manifesto and Agile Principles. There is plenty of material available on these topics.

Being agile involves a lot of team efforts. Getting feedback from customers or stakeholders early and quickly, reduces development efforts. Improving quality by using tools or by identifying bugs at an early stage help in making the team agile in software development.

Agile Practices – Management and Developer

There are 2 main divisions for Agile practices - one is from the management's perspective, and the other is from a developers' perspective.

The management practices are Sprint Planning, Capacity Balancing, Spring Monitoring, Kanban board and Sprint Review & Retrospective. All these management practices have excellent support when we use Team Foundation Server (TFS) 2015 or Visual Studio Team Services (VSTS).

However in this article, we will focus on the developer's perspective, and demonstrate how Visual Studio 2015 and TFS 2015 provides support for implementing agile practices.

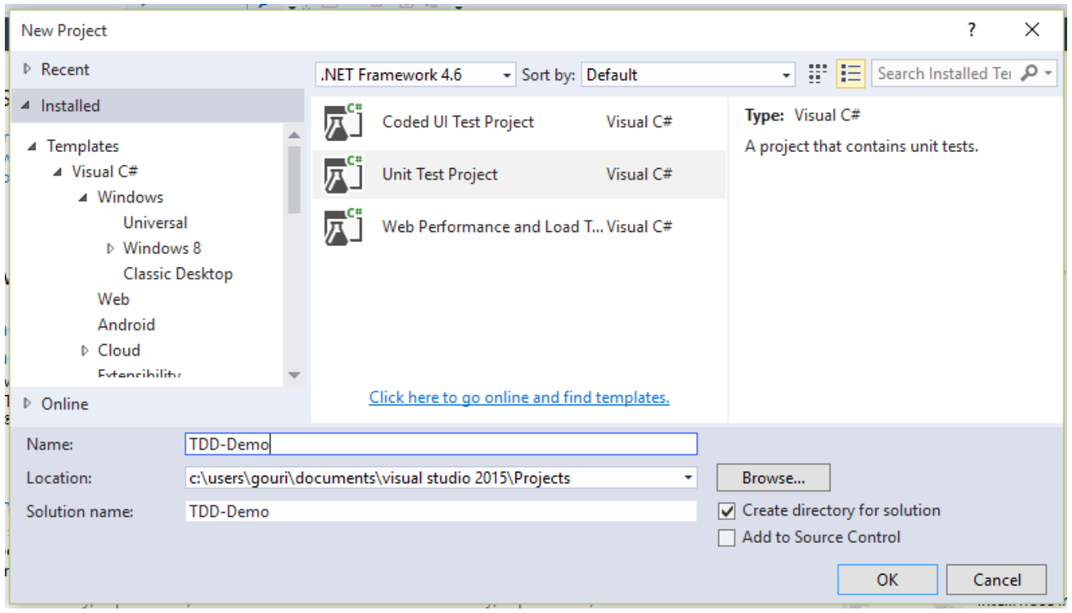
Agile Practices – Developer's Perspective

Test Driven Development or TDD

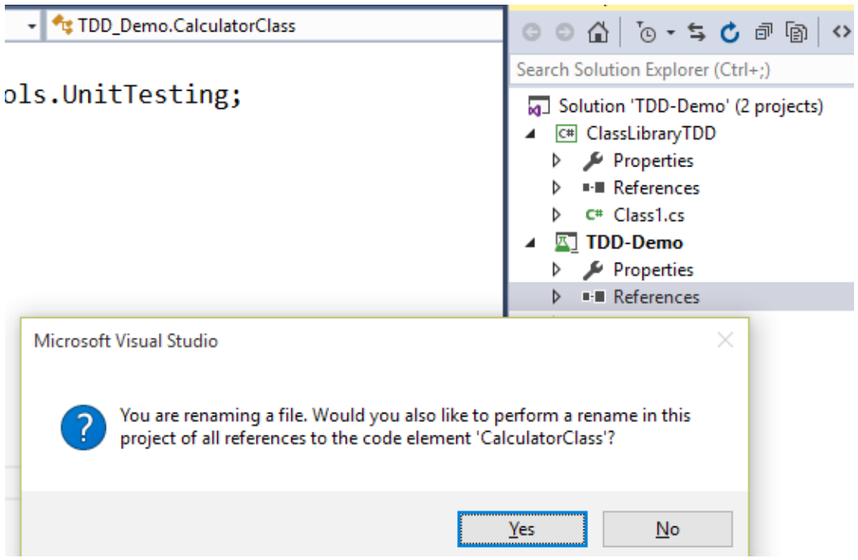
Test Driven Development (TDD) is a technique where you write the test first, and then write the functionality that fulfills the test. It is similar to defining the technical requirement first, which is represented by the test, and then create the code to satisfy that requirement. The test fails for the first time as you have not written the functional code. As we build the code, the test passes at certain points, and then we can stop adding code. This process helps in reducing development time and also reduces the bugs that may crop up. Proper usage of TDD requires that developers have knowledge of how the applications will be used in a real situation. TDD ensures that different functionality in the application is properly tested. As the tests are being executed even before writing the code, debugging at a later stage can be avoided.

In order to write TDD, you should first know what you want to do and how to test it. Write a small test with only a stub in it because of which the test will fail. Now write code to pass the test. Run the test and watch that is passes. If it fails, fix it, as you must have written something wrong.

There is no *direct* support for TDD in Visual Studio 2015, but we will see a small walkthrough of writing TDD. Create a project of type Unit Test Project which has a default class with the test method.



Change the name of the class so that the refactoring is automatically taken care of. See the following figure:



Observe the attribute for the class and test method which is added automatically, as shown in the following figure:

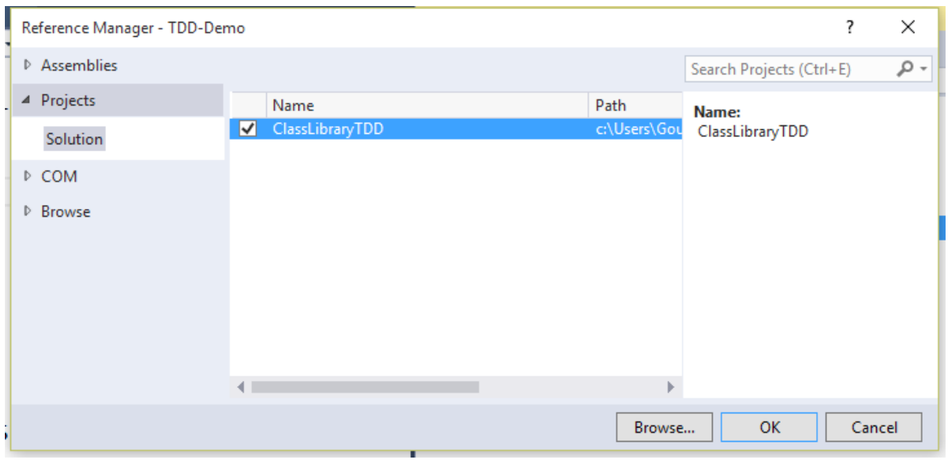
```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TDD_Demo
{
    [TestClass]
    0 references
    public class CalculatorTest
    {
        [TestMethod]
        0 references
        public void TestMethodSum()
        {
            Assert.Fail();
        }
    }
}
```

We can execute this test method using Test Explorer, and it will fail as there is no functionality. Add a new project to the solution of type class library and write some code in it.

```
public class CalculatorClass
{
  public int Sum(int[] num)
  {
    int result=0;
    for (int i = 0; i < num.Length; i++)
    {
      result += num[i];
    }
    return result;
  }
}
```

Now add a reference from unit test project to the class library project.



Change the code in the Test Method so that it tests the functionality.

```
[TestClass]
public class CalculatorTest
{
  [TestMethod]
  public void TestMethodSum()
  {
    int actual, expected = 45;
    CalculatorClass target = new CalculatorClass();
    actual = target.Sum(new int[] { 5, 10, 30 });
    Assert.AreEqual(actual, expected);
  }
}
```

Run the test from Test Explorer and see that it passes now.

Unit Tests Support in Visual Studio

Visual Studio provides excellent support for writing unit tests. The unit tests are not only limited to Microsoft framework, but third party testing is also supported.

Visual Studio 2015 supports the following automated tests - Unit Tests, Coded UI Tests, Web Performance Tests, Generic Tests, Load Tests and Ordered Tests.

Unit testing has already been discussed in this article http://bit.ly/dncm28-vs2015

Coded UI Tests (CUIT) are for testing the User Interface of an application. We can create CUIT by either recording the actions using CUIT Builder or use the already recorded actions (using Microsoft Test Manager). Another option to write CUIT is hand coding. You can find more information about CUIT at http://bit.ly/dncm28-cuit.

Web Performance Tests are used for HTTP requests and responses for a web application or a web site. You can also add validations, as well as parameters to the test.

Generic Tests are executed outside the Microsoft testing environment and returns a result as pass of fail. In case we need a particular sequence for executing tests, we can use Ordered Tests.

Once all the tests are written and executed, we can find out how these tests perform across different networks, and different browsers. We can achieve this by using **Load Tests**. We can create a scenario in which we can determine the percentage of various networks, browsers, tests. We can also specify the number of concurrent users, and if there are any think times etc. Load Testing can be executed either with Visual Studio 2015 or Visual Studio Team Services.

Pair Programming

In this technique, two programmers work together at the same workstation. One writes the code, and the other observes it, and reviews the lines of code. The reviewer gives suggestions to improve the code wherever possible. The one who writes the code is termed as *driver,* and the other as *navigator*. This increases the quality of software. Pair programming requires social communication and will fail if the pair is not working in tandem.

There is no direct support for pair programming via a tool like Visual Studio 2015. It is achievable with some experience, as one may initially feel awkward to start writing code in pair. It should be noted that two people are working together and none of them is mentoring the other. Instead, this is a collaborative effort to achieve high quality code.

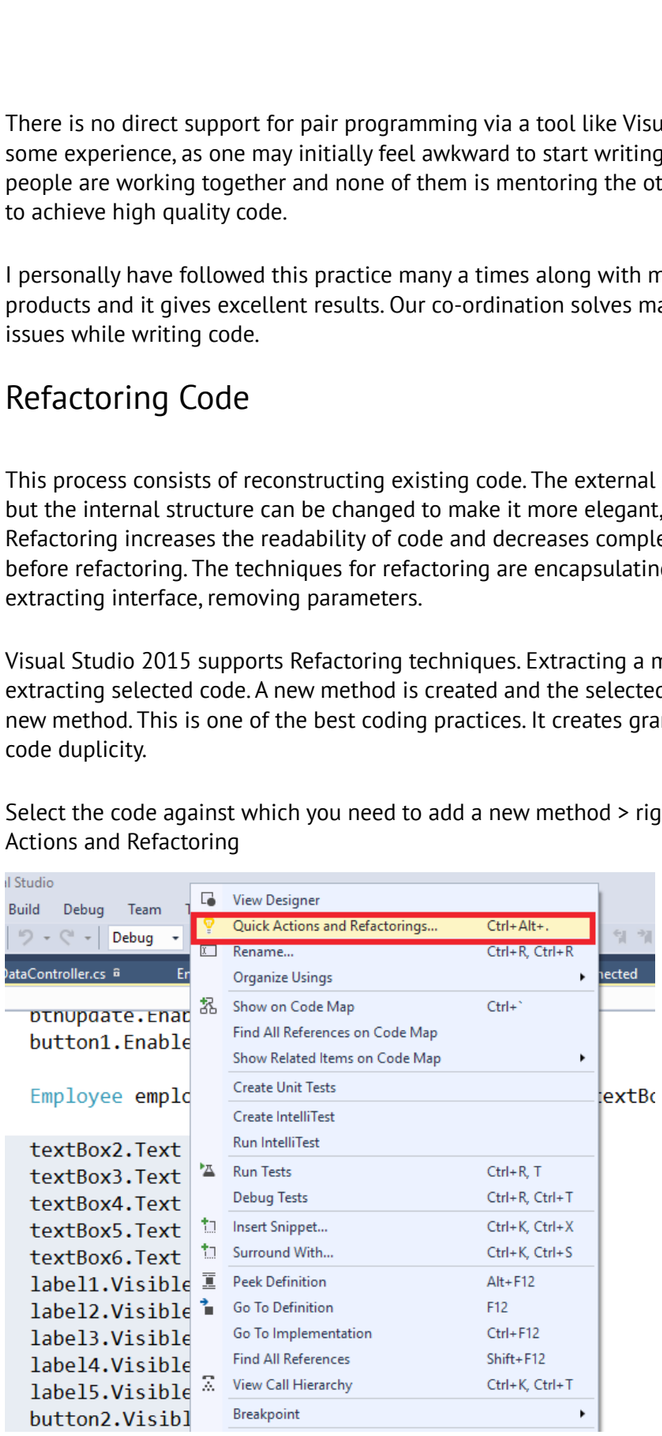
I personally have followed this practice many a times along with my husband while writing code for our products and it gives excellent results. Our co-ordination solves many problems before they become big issues while writing code.

Refactoring Code

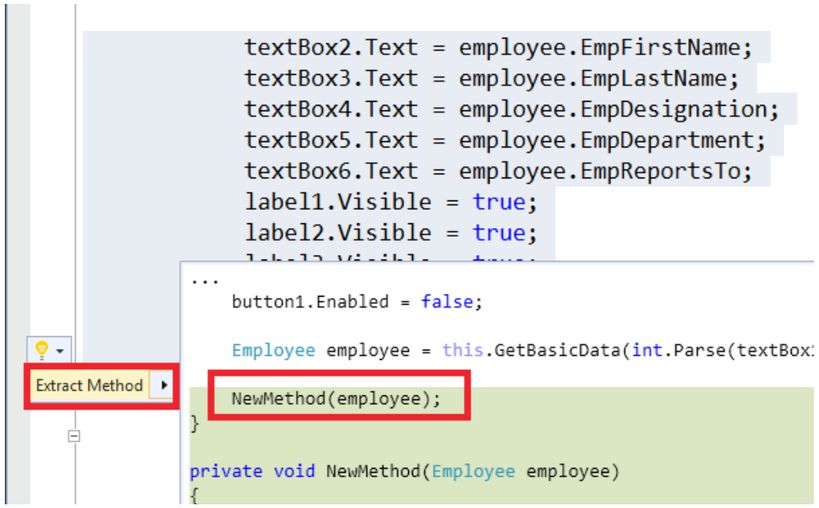
This process consists of reconstructing existing code. The external structure for the code remains as it is, but the internal structure can be changed to make it more elegant, easy to understand and maintainable. Refactoring increases the readability of code and decreases complexity. It is a good idea to write unit tests before refactoring. The techniques for refactoring are encapsulating field, extracting a method, renaming, extracting interface, removing parameters.

Visual Studio 2015 supports Refactoring techniques. Extracting a method creates a new method by extracting selected code. A new method is created and the selected code is replaced with the call to the new method. This is one of the best coding practices. It creates granular approach to the code and reduces code duplicity.

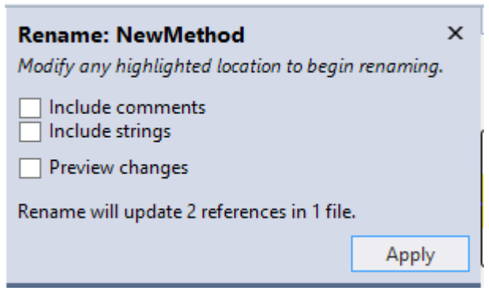
Select the code against which you need to add a new method > right click on it > select the menu for Quick Actions and Refactoring



Selecting the menu provides the option for method extraction.

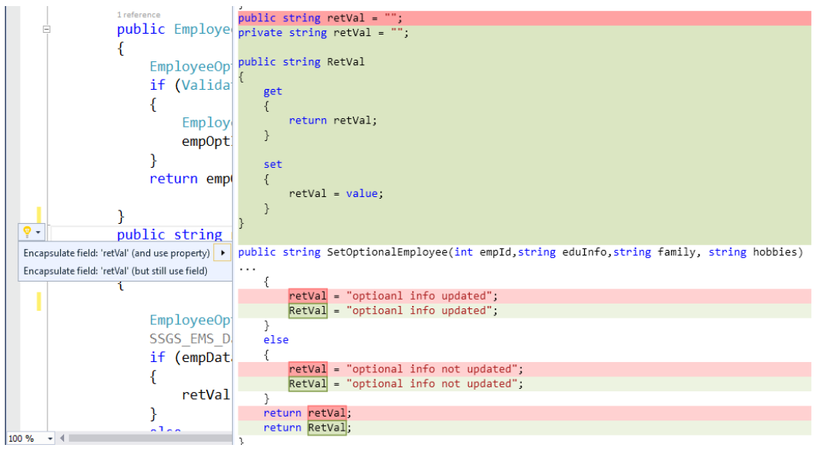


This is followed by options for renaming the method from NewMethod to a proper name, adding comments, adding any strings etc. If renaming is selected, the two references will be automatically changed.

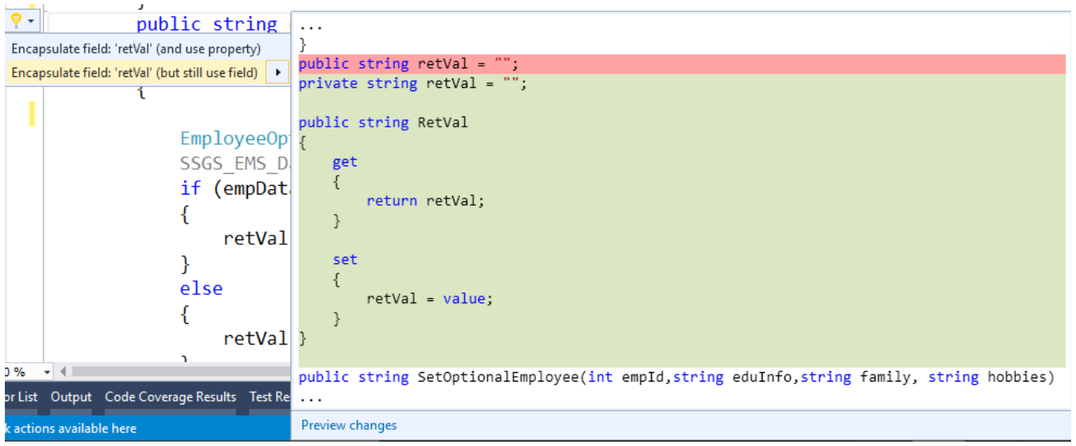


Rename Refactoring is useful in renaming identifiers like fields, variables, methods, classes, properties or even namespaces. We have already seen class renaming in TDD. When a variable is renamed, it also updates the usage of the variable. Changing the method name will update all reference to the method as seen in the previous refactoring technique. Changing the namespace will change the *using* statements and also the fully qualified names.

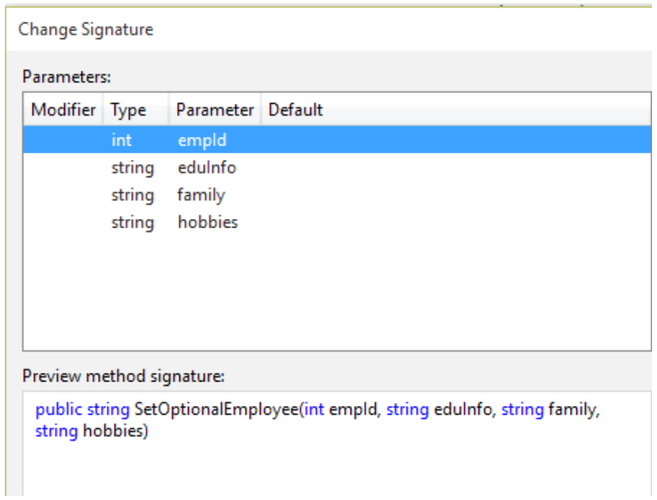
The **Encapsulating field** refactoring option lets us create a property from a field. All the code references will be automatically updated. In order to use it, right click the line in Visual Studio where the public field is declared, and select the menu for refactoring. It shows options to encapsulate the field to use a property as follows:



The other option is to encapsulate but still keep using the field:



Remove parameter refactoring technique is used to remove parameter from methods/ delegates. The parameter is removed and a new declaration is shown. Select the method from which to remove the parameter, and select the option for refactoring followed by change signature. This step removes the parameter from the signature of the method, but not from the body of the method. This will lead to errors during the build process in case you forgot to remove the parameter from the method body.



Feature Driven Development (FDD)

This is an iterative and incremental agile development process. It serves the purpose of delivering demonstrable software in a stipulated time period (sprint). This practice is more useful for large projects and teams.

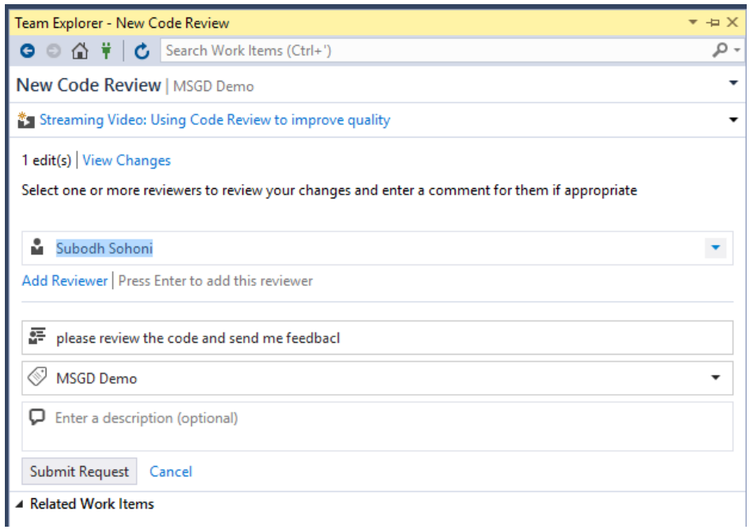
Code Review

Agile teams are cross functional. A part of this functionality can be achieved via code review. Code Review helps developers understand the code base. When one developer completes writing code, another developer also called as the reviewer looks into the code for logical errors, checks if the functionality is matching the

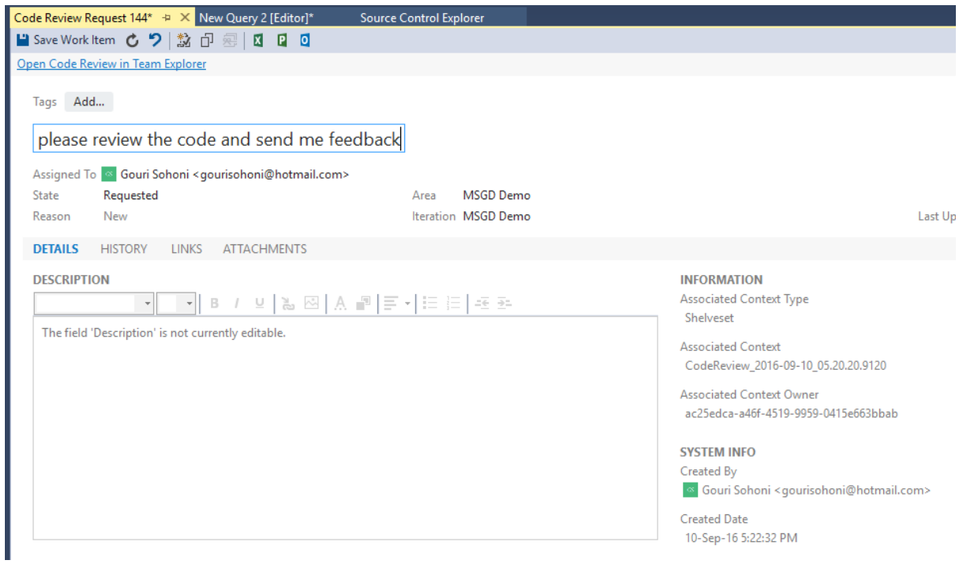
requirement, and if the unit testing suffices the functionality. He/she can also find out if the guidelines for code writing are followed or not. In Agile development, Code Review is very useful as no one person is solely responsible for the complete code base.

There is a direct support of Code Review in TFS or VSTS while writing code with Visual Studio 2015. There is a special work item of type *Code Review Request* for a developer to initiate code review. The reviewer can then send a Code Review Response in return with the comments and suggestions.

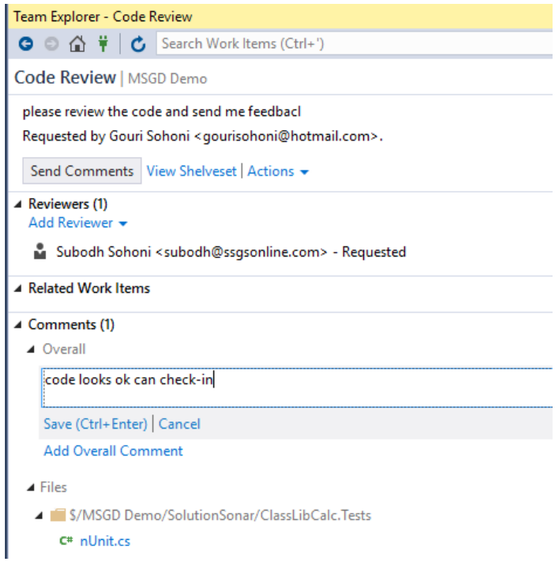
After writing code, a developer can select *Request Review* option from Team Explorer – My Work.



The Code Review Request work item looks as follows:



You can add multiple reviewers and click on Submit Request. The reviewer can review the code, provide comments and suggestions.

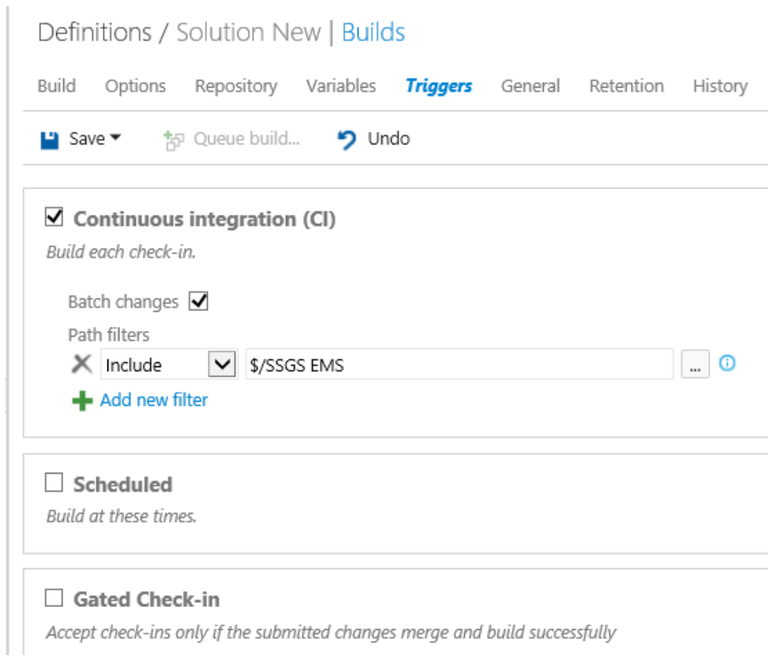


Observe that the files are included in a review, and once selected, show the changes made by the requester. Comments for an individual file can be added and finally sent by clicking on Send *Comments*. The requester then takes the necessary actions on the code and once done, the code can be checked in. The code is automatically put in the ShelveSet, which can be un-shelved and later checked in. Code Review can be closed once the code is un-shelved.

CICD (Continuous Integration Continuous Deployment)

Continuous Integration and Continuous Delivery are the two features of agile practices which help in reducing the sprint duration. As a developer, you may be working on user stories or product backlog items and writing code for the tasks assigned to you. As this is a collaborated development, other developers are also doing the same. We need to create an integrated build with all the code, from all the developers. The moment any developer checks-in code, a server side build is triggered to find out if the new code does not break the application when integrated with the latest code in the source control.
This can be easily achieved by setting the trigger as Continuous Integration for the build created with TFS 2015 or VSTS. Whenever a developer checks in code, the build will get automatically triggered with latest code in source control.

Here is the layout for such a build:



We can also specify the trigger for Gated Check-in which will ensure that the code will be checked-in only if it can be built with all the latest code in the source control.

Continuous Delivery is taking the concept of CI further. Once the application is built, it can be taken to the next stage to QA lead for testing, or to the production depending on the requirement. This can be achieved with TFS using Release Management.

Conclusion

In this article, we discussed some features of agile process from a developers' perspective. There are a lot of best practices which can be implemented while following the agile process. We also explored how tools like Visual Studio help implement these Agile best practices ■

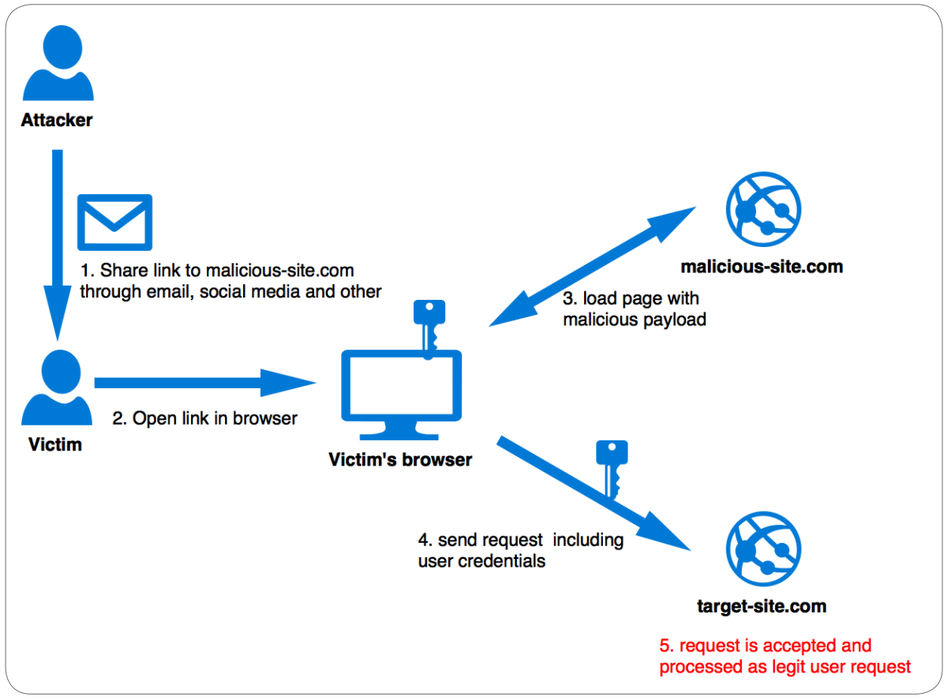
Microsoft® MVP Most Valuable Professional

Gouri Sohoni
Author

Gouri Sohoni is a Trainer and Consultant for over two decades. She specializes in Visual Studio - Application Lifecycle Management (ALM) and Team Foundation Server (TFS). She is a Microsoft MVP in VS ALM, MCSD (VS ALM) and has conducted several corporate trainings and consulting assignments. She has also created various products that extend the capability of Team Foundation Server.

Thanks to Subodh Sohoni and Suprotim Agarwal for reviewing this article.

Daniel Jimenez Garcia



ASP.NET CORE CSRF DEFENCE WITH ANTIFORGERY

Cross Site Request Forgery (aka CSRF or XSRF) is one of the most common attacks in which the user is tricked into executing an unwanted action through his browser on his behalf, in one of the sites he is currently authenticated.

ASP.Net Core contains an Antiforgery package that can be used to secure your application against this particular risk. For those who have used earlier versions of ASP.Net will see that things have changed a bit in the new framework.

Thanks to Suprotim Agarwal for reviewing this article.

Brief CSRF overview

CSRF attacks rely on the fact that most authentication systems will store credentials in the browser (such as the authentication cookie) which are automatically sent with any requests for that specific website domain/sub domain.

An attacker can then trick the user through social media, emails and other ways into a malicious or infected site, where they can send a request on their behalf to the attacked or targeted site. If the user is authenticated on the targeted site, this request will include his credentials, and the site won't be able to distinguish it from a legit request.

For example, hackers might send a link to the user which opens a malicious site. In this site, they will trick the user into clicking a button that posts a hidden form against the target site where the user is authenticated. The attacker will have forged this form to be posted against a URL like a fund transfers URL, and contain data like transferring some money to the attacker's account!

Look at the figure on the previous page to see CSRF in action.

It is important to understand that for these requests to be of any benefit to the attacker, they have to change some state in the application. Simply retrieving some data will be useless to them, as is for the users and their browsers who send the request and receive the response (even if they don't know about it).

Of course the impact of the attack depends on the particular application and the actual user privileges, but it could be used to attempt fund transfers or purchases on behalf of the user, send messages on their behalf, change email/passwords, or do an even greater damage for administrative users.

OWASP maintains a page with recommended prevention measures for this attack which include:

- Verifying the request origin
- Including a CSRF token on every request (which should be a cryptographically strong pseudorandom value so the attacker cannot forge the token itself)

You can read more about this attack and the recommended technology agnostic prevention measures on the OWASP page.

The Open Web Application Security Project (OWASP) is a worldwide not-for-profit charitable organization focused on improving the security of software, that defines its core purpose as "Be the thriving global community that drives visibility and evolution in the safety and security of the world's software." You can read more about them on their about page.

How ASP.Net Antiforgery works

ASP.Net Core includes a package called Antiforgery which can be used to protect your website against CSRF attacks. This package implements the CSRF token measure recommended by the OWASP site.

More specifically, it implements a mixture of the *Double Submit Cookie* and *Encrypted Token Pattern* described in the OWASP cheat sheet.

This basically means that it provides a stateless defence mechanism composed of two items (or token set) that should be found on any request being validated by the Antiforgery package:

- An antiforgery token included as a cookie, generated as a pseudorandom value and encrypted using the new Data Protection API

- An additional token included either as a form field, header or cookie. This includes the same pseudorandom value, plus additional data from the current user's identity. It is also encrypted using the Data Protection API.

These tokens will be generated server-side and propagated along with the html document to the user's browser. The cookie token will be included by default whenever the browser sends a new request while the application needs to make sure the request token is also included. (We will see in the next sections how to do this)

A request will be then rejected if:
- any of the two tokens is missing or have an incorrect format/encryption
- their pseudorandom values are different
- the user data embedded in the second token doesn't match the currently authenticated user

An attacker won't be able to forge these tokens himself. As they are encrypted, he would need to break the encryption before being able to forge the token.

If you are in a web farm scenario, it is important for you to check how the Data Protection API stores the keys and how they are isolated by default using an application identifier. This is important as when two individual instances of your web farm use different keys, they won't be able to understand the secrets from other instances. Probably this isn't what you want in your web farm and you will need to configure the Data Protection API so the instances share the same keys, and are thus able to understand secrets from each other. See the asp docs for an example with the authentication cookie.

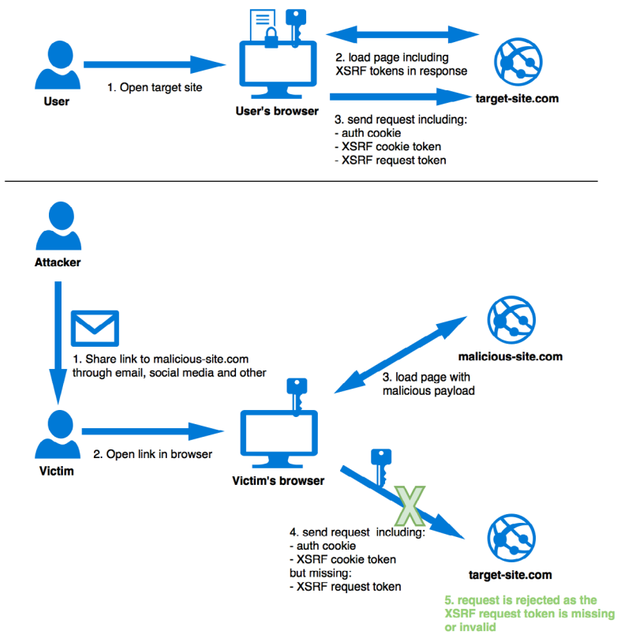


Figure 2: Antiforgery in legit requests Vs CSRF attack requests

Additionally, whenever the tokens are generated by the server and included in a response, the X-FRAME-OPTIONS header is included with the value SAMEORIGIN. This prevents the browser from rendering the page in an iframe inside a malicious website that might attempt a Clickjacking attack.

The following sections will describe in detail how to enforce Antiforgery validation in your controller actions, and how to make sure the two tokens are included within the requests.

Using Antiforgery in your application

Adding Antiforgery to the project

The **Microsoft.AspNetCore.Antiforgery** package is already included as a dependency by **Microsoft.AspNetCore.Mvc**. Similarly, the required Antiforgery services are automatically registered within the DI container by calling `services.addMvc()` in your `Startup.ConfigureServices()` method.

There are still cases where you might want to manually add Antiforgery to your project:

- If you need to override the default Antiforgery options, then you need to manually call services.`AddAntiforgery(opts => {opts setup})` so you can provide your specific option settings. These options include things like the cookie name for the cookie token, and the form field or header name for the request token.

- If you are writing an ASP.Net Core application from scratch without using the MVC framework, then you will need to manually include the Antiforgery package and register the services.
Bear in mind that this is just registering Antiforgery within your project and setting the options. It does not automatically enable any kind of request validation! You will manually need to do that as per the following sections.

Protecting controller actions

Even after Antiforgery has been added to your project, the tokens will not be generated, nor validated on any request, until you tell them to. This section describes how to enable the tokens validation as part of the request processing, while the following sections describes how to generate the tokens and make sure they are included in the requests sent by the browser.

You could manually add some middleware where you require an instance of `IAntiforgery` through dependency injection, and then call `ValidateRequestAsync(httpContext)`, catching any `AntiforgeryValidationException` and returning a 400 in that case:

```
try
{
  await _antiforgery.ValidateRequestAsync(context);
  await next.Invoke();
}
catch (AntiforgeryValidationException exception)
{
  context.Response.StatusCode = 400;
}
```

However if you are using MVC, then there are Antiforgery specific authorization filters that will basically handle that for you. You just need to make sure your controller actions are protected by using a combination of the following attributes:

- `[AutoValidateAntiforgeryToken]` – This adds the Antiforgery validation on any "unsafe" requests, where unsafe means requests with a method other than GET, HEAD, TRACE and OPTIONS. (As the other HTTP methods are meant for requests that change the server state). It will be applied globally (when added as a global filter) or at class level (when added to a specific controller class).

• `[ValidateAntiForgeryToken]` – This is used to add the Antiforgery validation to a specific controller action or class. It does not take into account the request HTTP method. So if added to a class, it will add the validation to all actions, even those with methods GET, HEAD, TRACE or OPTIONS.

• `[IgnoreAntiforgeryToken]` – Disables the Antiforgery validation in a specific action or controller. For example, you might add the Antiforgery validation globally or to an entire controller class, but you might still want to ignore the validation in specific actions.

You can mix and match these attributes to suit your project needs and team preference. For example:

• You could add `AutoValidateAntiforgeryToken` as a global filter and use `IgnoreAntiforgeryToken` in rare cases where you might need to disable it for an "unsafe" request.

• Alternatively you could add `AutoValidateAntiforgeryToken` just to your WebAPI-style of controllers (either manually or through a convention) and use the `ValidateAntiforgeryToken` in MVC-style of controllers handling many GET requests returning views with a few unsafe requests like PUT and POST actions.

Find the approach that works best for your project. Just make sure the validation is applied on every action where you need it to be.

Now let's take a look at how to make sure the tokens are included within the requests sent by the browser.

Including the tokens in server side generated forms

For the antiforgery validation to succeed, we need to make sure that **both the cookie token and the request token** are included in requests that will go through the validation.

You will be relieved to hear that whenever you generate a form in a razor view using the new tag helpers, the **request token** is automatically included as a hidden field with the name **__RequestVerificationToken** (you can change it by setting a different name in the options when adding the Antiforgery services).

For example the following razor code:

```
<form asp-controller="Foo" asp-action="Bar">
  …
  <button type="submit">Submit</button>
</form>
```

Will generate the following html:

```
<form action="/Foo/Bar" method="post">
  …
  <button type="submit">Submit</button>
  <input name="__RequestVerificationToken" type="hidden" value="CfDJ8P4n6uxULApNkzyV
  aa34lxdNGtmIOsdcJ7SYtZiwwTeX9DUiCWhGIndYmXAfTqW0U3sdSpzJ-NMEoQPjxXvx6-1V-5sAonTik
  5oN9Yd1hej6LmP1XcwnoiQJ2dRAMyhOMIYqbduDdRI1Uxqfd0GszvI">
</form>
```

You just need to make sure the form includes some of the asp-* tags, so it is interpreted by razor as a tag helper, and not just a regular form element.

• If for whatever reasons you want to omit the hidden field from a form, you can do so by adding the attribute `asp-antiforgery="false"` to the form element.

• If you use the html helpers in your razor views (as in `@Html.BeginForm()`), you can still manually generate the hidden input field using `@Html.AntiForgeryToken()`.

So that's how you generate a request token as a hidden field in the form, which will be then included within the posted data.

What about the cookie token?

Well, Antiforgery treats both the request and cookie tokens as a token set. When the `IAntiforgery` method `GetAndStoreTokens(httpContext)` is executed (which is what happens behind the scenes when generating the form hidden field), it returns the token set including both request and cookie tokens. And not only that, it will also make sure that:

• The cookie token is added as an http only cookie to the current HttpContext response.

• If the token set was already generated for this request (i.e. `GetAndStoreTokens` has already been called with the same httpContext), it will return the same token set instead of regenerating it again. This is important as it allows multiple elements of the page to get posted and also pass the validation, like having multiple forms (as the cookie would be the same for all forms, if every form had a different token, then only the form with the token matching the cookie one would pass the validation!).

Many of the cookie properties can be changed through the options:

• The cookie is always set as an http only cookie (so JavaScript doesn't have access to it)

• The default cookie **name** is generated for each application as *".AspNetCore.AntiForgery."* followed by a hash of the application name. It can be overridden by providing a **CookieName** in the options.

• No cookie **domain** is set by default, so the browser assumes the request domain. You can specify one through the **CookieDomain** option.

• The cookie **path** is defaulted from the current request path base (typically "/"), but can be forced with the **CookiePath** option.

• You can also enable the flag **RequireSsl** in the options, which will set the Secure flag of the cookie as true.

As the browser will send the cookies with subsequent request, whenever the form is posted, the request will contain both the cookie token (in the cookie) and the request token (in the form field).

In short, as long as you somehow generate the request token, the cookie token will also be generated and automatically added to the response.

Including the tokens in AJAX requests

In the previous section, we have seen how to generate a hidden field with the request token inside forms. But what about when sending data as part of an AJAX request, without any server-side generated form involved?

Well, this depends a lot on what JavaScript framework you are using for structuring your client-side code. Angular provides CSRF support out of the box and is so commonly used these days that even the Antiforgery repo contains an example for it. I will too take a look at the Angular-Antiforgery integration and later I will explain how a similar approach could be manually introduced for simple jQuery requests.

CSRF - The Angular case

In the case of Angular, you will be using their $http service for sending AJAX requests. This service will automatically include a header with the name **X-XSRF-TOKEN** if it can find the token value as a cookie with the name **XSRF-TOKEN**. So the easiest way is to play the way Angular wants us to, and create some middleware that will get the request token, and store its value as the XSRF-TOKEN cookie.

Even if it is added as a cookie, this is still the request token and not the cookie token! It might sound confusing, so let me try to clarify it:

- The application will send back to the browser a cookie XSRF-TOKEN with the request token and another cookie .AspNetCore.Antiforgery.* with the cookie token.

- Whenever Angular sends an Ajax request, the request will include a header X-XSRF-TOKEN with the request token and the cookie .AspNetCore.Antiforgery.* with the cookie token.

- The Antiforgery validation will make sure that both tokens are valid and share the same secret, etc.

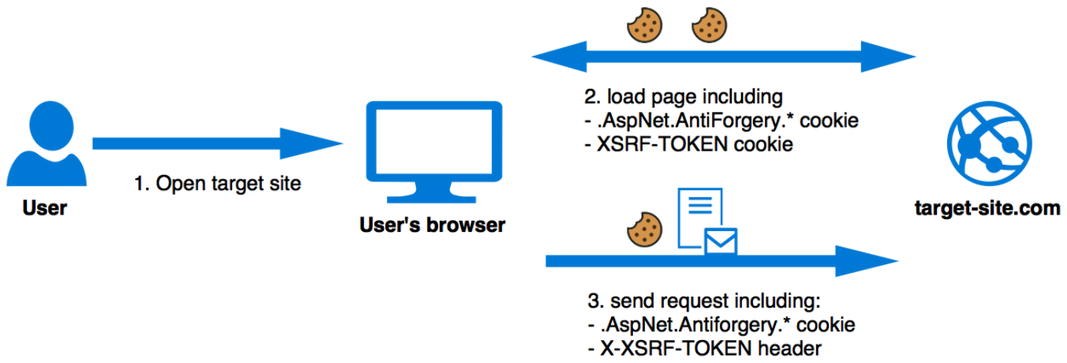


Figure 3: CSRF tokens with Angular

Since the default header name for the request token is *RequestVerificationToken*, we need to change it and make sure Antiforgery searches for the request token in a header with name X-XSRF-TOKEN. Let's just manually add `Antiforgery` and setup the options in the `ConfigureServices` method:

```
services.AddAntiforgery(opts => opts.HeaderName = "X-XSRF-Token");
```

Now we need to make sure we generate the tokens and include the request token in a cookie with name XSRF-TOKEN so Angular $http service can read it and include it as the header.

- This cannot be an http only cookie, since Angular code needs to read the cookie value so it can be included as a header in subsequent requests!

We will be interested in doing so every time we generate a full html document, so we could create a new Result Filter that basically does this if the result is a `ViewResult`:

```
public class AngularAntiforgeryCookieResultFilter: ResultFilterAttribute
{
  private IAntiforgery antiforgery;
  public AngularAntiforgeryCookieResultFilter(IAntiforgery antiforgery)
  {
    this.antiforgery = antiforgery;
  }

  public override void OnResultExecuting(ResultExecutingContext context)
  {
    if (context.Result is ViewResult)
    {
      var tokens = antiforgery.GetAndStoreTokens(context.HttpContext);
      context.HttpContext.Response.Cookies.Append("XSRF-TOKEN", tokens.
      RequestToken, new CookieOptions() { HttpOnly = false });
    }
  }
}
```

The only remaining bit is to configure it as a global filter. This way, every time we render a full html document, the response will also include cookies for both the cookie token and the request token. You will do so again in the `ConfigureServices` method:

```
services.AddAntiforgery(opts => opts.HeaderName = "X-XSRF-Token");
services.AddMvc(opts =>
{
  opts.Filters.AddService(typeof(AngularAntiforgeryCookieResultFilter));
});
services.AddTransient<AngularAntiforgeryCookieResultFilter>();
```

And that's it, now your Angular code can use the $http service and the tokens will be included within the request. You can check an example with a simple TODO Angular application in GitHub.

CSRF - The manual case with jQuery

If you are using a framework other than Angular (or no framework at all), the way tokens are handled might be different. However the underlying principles in every case will be the same, as you always need to make sure that:

- The tokens are generated server side.
- The request token is made available to the client JavaScript code.
- The client JavaScript code includes the request token either as a header or as a form field.

So let's say your client code sends AJAX requests using jQuery. Since we have already created a *Result* Filter

that includes the request token as a XSRF-TOKEN cookie, and have configured Antiforgery to look for the request token in a header named X-XSRF-TOKEN, we can reuse the same approach.

You will need to get the request token from the cookie and include it within your requests (be careful if you use something like `$.ajaxSetup()` as the token would be included on any requests, including those from 3rd party code using jQuery):

```
var token = readCookie('XSRF-TOKEN');
$.ajax({
  url: '/api/todos',
  method: 'PUT',
  data: JSON.stringify(todo),
  contentType: 'application/json',
  headers: { 'X-XSRF-TOKEN': token },
  success: onSuccess
});
```

Where *readCookie* can be something like the jQuery cookies plugin or just your own utility for reading the value of a cookie (taken from Stack Overflow):

```
function readCookie(name) {
  name += '=';
  for (var ca = document.cookie.split(/;\s*/), i = ca.length - 1; i >= 0; i--)
    if (!ca[i].indexOf(name))
      return ca[i].replace(name, '');
}
```

But we are just using those cookie and header names because we set up the server this way for Angular before. If you are not using Angular, you could use whatever name you like for the cookie with the request token (as long as the middleware adding that cookie uses the same name), and the default header name (as long as you don't specify a different one in the options):

```
var token = readCookie('AnyNameYouWant');
$.ajax({
  …
  headers: { 'RequestVerificationToken': token },
  …
});
```

In fact, you don't necessarily need to use a cookie to propagate the request token to your JavaScript code. As long as you are able to do so, and the token is included within your AJAX requests, then the validation will work. For example, you could also render an inline script in your razor layout that adds the request token to some variable, which is later used by your JavaScript code executing jQuery AJAX requests:

```
@*In your layout*@
@inject Microsoft.AspNetCore.Antiforgery.IAntiforgery antiforgery;
@{
    var antiforgeryRequestToken = antiforgery.GetAndStoreTokens(Context).
    RequestToken;
}
<script>
    //Render the token in a property readable from your client JavaScript
    app.antiforgeryToken = @Json.Serialize(antiforgeryRequestToken);
</script>
//In your client JavaScript
$.ajax({
    …
    headers: { 'RequestVerificationToken': app.antiforgeryToken },
    …
});
```

Validate the request origin

You might have noticed that OWASP recommended another protection measure - validating the request origin. The way they recommend validating it is by:

1. Getting the request origin (by looking at the Origin and Referer headers)
2. Getting the target origin (looking at the Host and X-Forwarded-Host headers)
3. Validating that either the request and target origins are the same, or the request origin is included in a whitelisted list of additional origins.

You can implement an `IAuthorizeFilter` that goes through those steps for all unsafe requests (any request with a method other than GET, HEAD, TRACE and OPTIONS) and when the validation fails, sets the result as a 400:

```
context.Result = new BadRequestResult();
```

It shouldn't be too hard writing such a filter and including it as a global filter. If you want to see an example including options for the whitelisted extra origins, check the sample project in GitHub.

Conclusion

Security is a critical aspect for any non-trivial application. Knowing the different types of attacks and how to protect against them, is an art in itself. Luckily this entire security process is simplified for us with mature frameworks that provide more and better security features.

When it comes to the particular CSRF attack, ASP.Net Core gives you the tools to protect your application but still requires some (minimum) effort to be properly configured and setup ■

Download the entire source code from GitHub at
bit.ly/dncm28-aspcore-antiforgery

Daniel Jimenez Garcia
Author

Daniel Jimenez Garcia is a passionate software developer with 10+ years of experience. He started as a Microsoft developer and learned to love C# in general and ASP MVC in particular. In the latter half of his career he worked on a broader set of technologies and platforms while these days is particularly interested in .Net Core and Node.js. He is always looking for better practices and can be seen answering questions on Stack Overflow.

THANK YOU

FOR THE 28th EDITION


@damirarh


@yacoubmassad


@gouri_sohoni


@dani_djg


@benjaminjakobus


@craigber


@suprotimagarwal


@jfversluis


@saffronstroke


@gilfink


@subodhsohoni


@svswaminathan

JOIN US