Issue 38 | Sep-Oct 2018

DNC Magazine

www.dotnetcurry.com

WRITING PURE CODE IN C#

UNIT TESTING IN ANGULAR COMPONENTS

BLAZOR- .NET IN THE BROWSER

SERVERLESS APPS WITH AZURE FUNCTIONS

CQS A SIMPLE BUT POWERFUL PATTERN

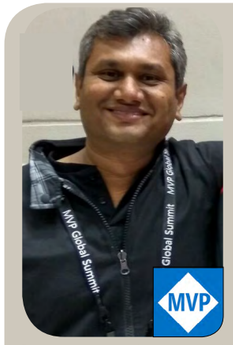
THE EVOLUTION OF C#

NON-FUNCTIONAL REQUIREMENTS: THE UNSUNG HEROES

FIND US HERE

www.dotnetcurry.com/magazine/

EDITORIAL



Suprotim Agarwal
Editor in Chief

Hello Friends,

Let me kickstart this edition by saying 'Thank You' for the overwhelming response to our 6th Anniversary Edition. Over 140K downloads, social media shoutouts, emails and comments has all made it worthwhile!

For this 38th Edition, our authors have covered Software Development practices, Design Patterns, as well as the latest in .NET, C#, Azure and Angular.

Rahul kickstarts a new series on Project Management as he discusses about Non-Functional Requirements (NFRs) and how it defines the key design aspects of any product. Yacoub shows us how to write Pure Code in C#, while Tim talks about CQS, an architecture that structures your app in a clean, reusable way.

Damir takes us through a whirlwind tour of how C# has evolved through these years. Gerald explains what it means to go Serverless in Azure, and Ravi talks about Unit Testing Angular Components.

Last but not the least, Daniel introduces us to Blazor, a new exciting experiment by Microsoft that allows .NET Core to be run in the browser.

I hope you will enjoy reading this edition. Reach out with your comments to any of us on Twitter with our handle @dotnetcurry or email me at suprotimagarwal@dotnetcurry.com.

THE TEAM

Editor In Chief : Suprotim Agarwal (suprotimagarwal@dotnetcurry.com)

Art Director : Minal Agarwal

Contributing Authors :
Damir Arh
Daniel Jimenez Garcia
Gerald Versluis
Rahul Sahasrabuddhe
Ravi Kiran
Tim Sommer
Yacoub Massad

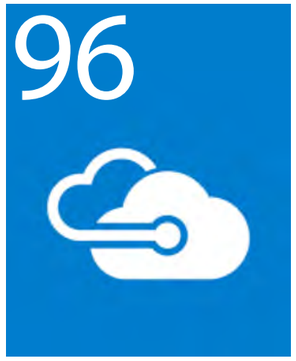
Technical Reviewers :
Damir Arh,
Keerti Kotaru,
Subodh Sohoni,
Tim Sommer
Yacoub Massad

Next Edition : Nov 2018

a2z | Knowledge Visuals

et curry.com

CONTENTS



THANK YOU

FOR THE 38th EDITION


@dani_djg


@jfversluis


@yacoubmassad


@damirarh


@sravi_kiran


@Rahul1000Buddhe


@sommertim


@keertikotaru


@suprotimagarwal


@subodhsohoni


@saffronstroke

WRITE FOR US

mailto: suprotimagarwal@dotnetcurry.com

Daniel Jimenez Garcia

AN OVERVIEW OF BLAZOR - .NET IN THE BROWSER

JavaScript took over web development and became the de-facto standard language of the web. Technologies like Flash or Silverlight succumbed to the unstoppable rise of JavaScript.

Since then, JavaScript has expanded into even further areas like mobile applications, servers and desktop applications.

This doesn't mean JavaScript will stay unchallenged forever.

First announced in 2015, WebAssembly (aka WASM) is a new portable binary format designed to be efficient in size, as well as in parsing and execution time. It finally got its Minimum Viable Product (MVP) released and supported by all major modern browsers during 2017, with older browsers relegating to polyfills.

Since the WebAssembly standards are being developed by a W3C group with engineers from Google, Mozilla, Microsoft and Apple, it should come as no surprise that Microsoft saw it as an opportunity to explore running .NET in the browser.

This is where Blazor comes into play, the current Microsoft experiment that allows .NET Core to be run in the browser.

In the rest of the article, we will explore how Blazor brings these technologies together and the new possibilities it brings to the table.

INTRODUCTION

A 5 minutes introduction to WebAssembly

WebAssembly is a portable binary format that has been designed to be:

Compact and fast to parse/load so it can be efficiently transferred over the wire, loaded and executed by the browser

Compatible with the existing web platform. Can run alongside JavaScript, allows calls to/from it, can get access to the Browser APIs and runs in the same secure sandbox than JavaScript code

A compiler target for multiple languages with C/C++ as the initial focus, but more to follow

It is important to highlight that WebAssembly does not attempt to replace JavaScript. It has been designed to complement and integrate with JavaScript, as well as with other existing web technologies like HTML and CSS.

Higher level languages can be compiled to WebAssembly, which is then run by the browser in the same sandboxed environment as JavaScript code.

Let's briefly see how WebAssembly *execution* compares to JavaScript, as the efficiency and performance gains are one of its more promising features.

- At a high level, JavaScript code gets executed by a runtime or VM where it goes through a process of parsing, compilation/optimization and finally the JIT (Just in time compiler) monitors and performs speculative optimizations of the binary code being executed.

- WebAssembly modules are compiled from higher level languages and have been already parsed and compiled/optimized so they need to go through a fast decoding phase (as the module is already in bytecode format close to machine code) before being injected into the final JIT stage of this pipeline.

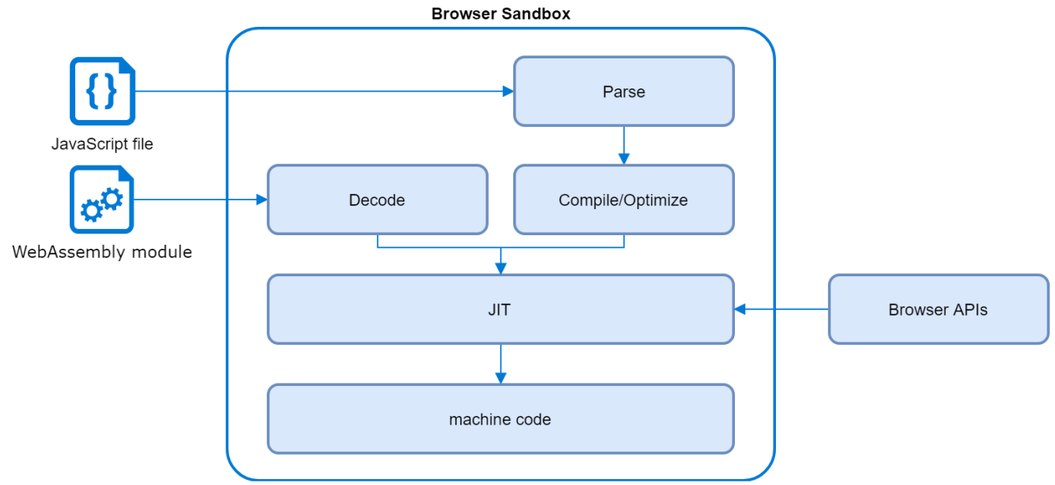


Figure 1, executing JavaScript code and WebAssembly modules in the browser

This is admittedly a very brief overview of WebAssembly since we still have lots more ground to cover in the article. For a more detailed discussion, apart from the official website you can check this article from Dave Glick, this series of articles from Lin Clark and this article from William Martin.

In a nutshell, WebAssembly promises a more efficient/faster way of running code in the browser, using any higher-level language that can target it for development, while being compatible with the existing web technologies.

As you can imagine, it is still early days for WebAssembly.

The documentation and tooling available are pretty rough, including areas as crucial as browser and development tools.

In terms of the features supported today, one of its main current limitations is the lack of support for direct access of the DOM and browser APIs, for which JavaScript interop is the only approach. It is early days and proposals to extend the current MVP are being studied.

Executing .NET code in the browser with Mono

Once a new bytecode format for running code in the browser is available, one simple step to run .NET code in the browser would be using a runtime compiled to WebAssembly.

This is where Mono fits in the puzzle.

Mono is an open source implementation of the .NET Framework that provides .NET Standard 2.0 support and has recently been compiled to WebAssembly.

The way .NET in the browser currently works is by compiling the Mono runtime to WebAssembly which can then use Mono's own IL interpreter to load and execute .NET assemblies. A JavaScript file bootstraps this runtime and downloads the dlls which are then provided to the runtime. This JavaScript file also provides access to any browser APIs required.

Overall this means your code gets compiled to .NET Standard dlls which are downloaded by the browser and interpreted by Mono's runtime compiled to WebAssembly, which is the only part actually compiled to WebAssembly. This process is known as *Interpreted Mode* and it can be roughly depicted by the following diagram:

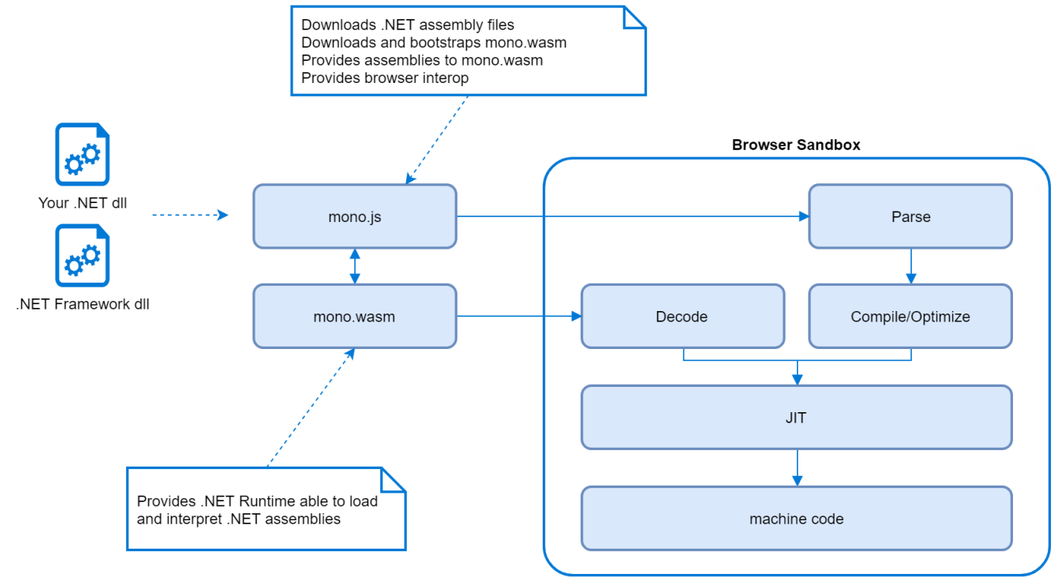


Figure 2, The WebAssembly Mono runtime in interpreted mode

As you can imagine the only bit optimized for WebAssembly is the runtime, which then needs to use its own IL interpreter to run the actual application assemblies. This is not very optimal, losing some of the benefits WebAssembly provides in terms of efficiency and performance.

However, the Mono team has been exploring an alternative Ahead of Time (AoT) mode where all of your .NET application code would get compiled to WebAssembly and executed with no previous interpretation of .NET assemblies needed.

You can find an interesting discussion of both modes in Steve Sanderson introduction to Blazor.

Blazor

The last piece of the puzzle is Blazor (Browser + Razor), which is a **framework for building Single Page Applications (SPA) in .NET Core** which uses the Mono WebAssembly runtime.

Blazor aims to provide typical SPA features like components, routing and data binding while leveraging the .NET Framework and its tooling.

It provides a familiar development experience for .NET developers who can now use the same language across the entire stack. At the same time, it is a true client framework which imposes no restrictions on server-side technologies and can be deployed from any server capable of serving static files.

The framework is based around Razor pages which are used to create Pages and Components. These support dynamic HTML, one-way and two-way data binding and DOM event handling.

It also provides a JavaScript interop service that can be used to either call JavaScript functions or receive a call from them.

The following diagram shows how these pieces fit together in the context of a Blazor project:

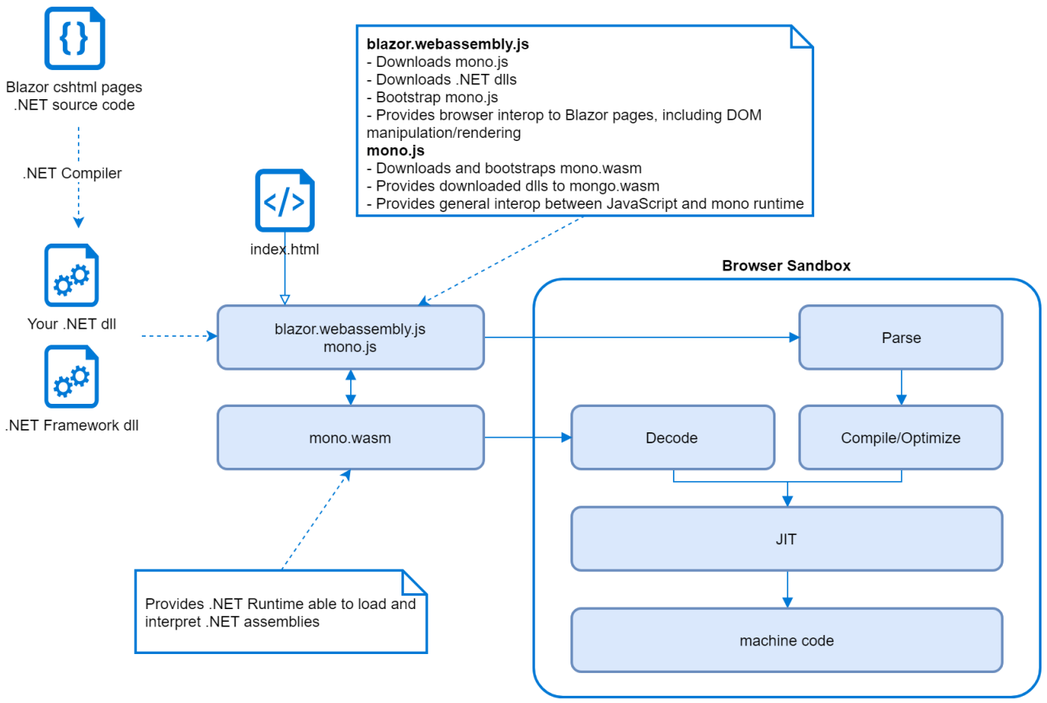


Figure 3, Blazor, Mono and WebAssembly

It is still very early days in the Blazor project, which is still officially considered an experiment by Microsoft. Whether it will become a supported product with enough adoption by the developers is yet to be seen. However, it is certainly promising and while the setup seems complicated, the development experience is surprisingly smooth where all the pieces needed for running .NET code in the browser work out of the box.

It also presents a different philosophy from past failed attempts like Silverlight, in the sense it doesn't need any specific plugins to be installed in the browsers. It is fully based on technologies provided by (modern) browsers.

Older browsers like IE can be supported using WebAssembly polyfills, although they are not a priority for the team and support is partial since Blazor itself has dependencies like Promise which are not polyfilled yet by the project templates.

If you want to read more, the official docs should be the first stop to read about Blazor. Then make sure you check out the Awesome Blazor site which contains a curated list of videos, articles, examples and much more.

Getting started

In order to get started with Blazor you need to install a few prerequisites:

- Install Visual Studio 2017, version 15.7 or later.
- Install the .NET Core 2.1 SDK, version 2.1.300 or later
- Install the Blazor Language Services extension for VS
- Optionally install the templates for the `dotnet new` command by running `dotnet new -i Microsoft.AspNetCore.Blazor.Templates::*` in the console.

Pay close attention to the versions of Visual Studio and the .NET Core SDK since earlier versions won't work!

You can find the version of the .NET Core SDK by running `dotnet --version` in the console (mine is 2.1.302 at the time of writing). The version of Visual Studio can be found in the *Help > About Microsoft Visual Studio* window.

Everything related to Blazor is still in flux, so I would also recommend you to check out the official docs in case things changed since I wrote the article.

You will also need a modern browser like Chrome, Firefox, Safari or Edge.

While support for older browsers is possible through polyfills, browsers like IE11 are not a priority for the team (and might never be) and don't currently work. You can see some community-driven polyfills to address the issue but there is no guarantee these will keep working with newer versions.

Now that you have everything in place, open Visual Studio and create a new project selecting ASP.NET Core Web Application. A new window will open where you should see three different Blazor projects:

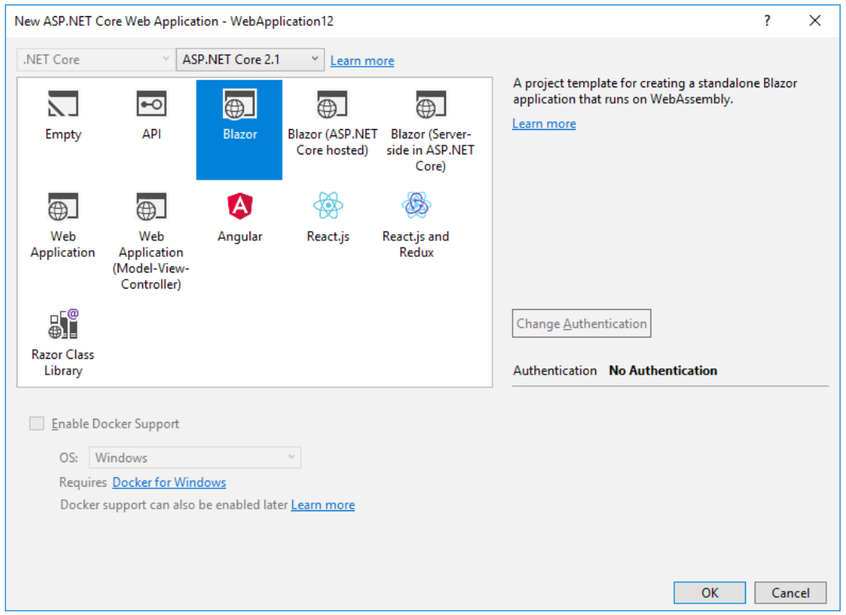


Figure 4, Blazor project templates

We will discuss these in more detail later, select the Blazor one and create the new project. Once the project is generated, you should see the following folder structure:

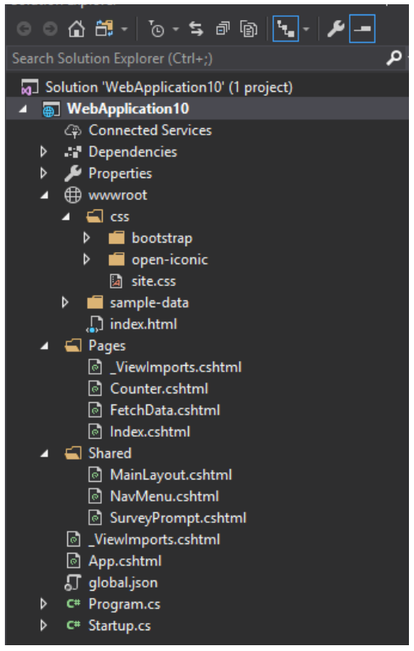


Figure 5, new Blazor project structure

You can now go ahead and run the project in the *Debug* mode. A browser window should appear, display a *Loading* message for about a second and then your first Blazor app should appear:

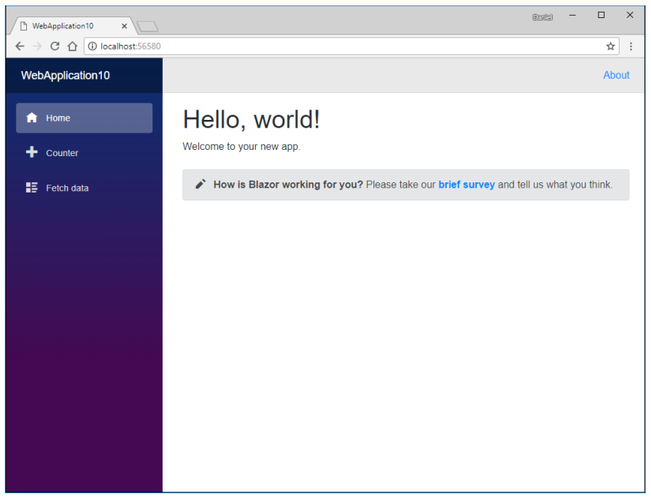


Figure 6, your first Blazor application

Let's see if we can understand how Blazor is using Mono and WebAssembly to run this project.

Open the `index.html` file of your project and you will see it is surprisingly simple.

The header loads the styles while the body contains an `<app>` element where the application will later be mounted, plus a script reference for `blazor.webassembly.js` which is the one starting the bootstrap process of the app in the browser.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset=”utf-8” />
  <meta name=”viewport” content=”width=device-width”>
  <title>WebApplication10</title>
  <base href=”/” />
  <link href=”css/bootstrap/bootstrap.min.css” rel=”stylesheet” />
  <link href=”css/site.css” rel=”stylesheet” />
</head>
<body>
  <app>Loading...</app>
  <script src=”_framework/blazor.webassembly.js”></script>
</body>
</html>
```

The script `blazor.webassembly.js` is quite important as it is the one in charge of starting the application, including downloading and starting the Mono WebAssembly runtime. If you open the network tab of your browser, you will see the following sequence:

- The blazor.webassembly.js is downloaded
- The blazor.boot.json is downloaded by blazor.webassembly.js
- The mono.js script is downloaded by blazor.webassembly.js
- The mono.wasm WebAssembly module is downloaded by mono.js
- All the dlls needed for the project are downloaded by blazor.webassembly.js

The information contained in blazor.boot.json is generated at compile time and basically describes the assemblies needed by your application, your main assembly and the entrypoint function:

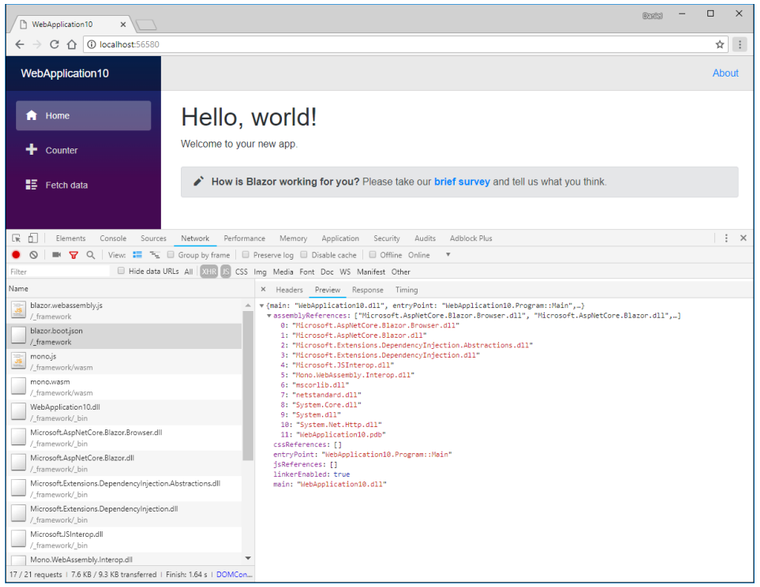


Figure 7, Bootstrap process of the application

Once everything is downloaded, the Mono runtime is started with the given assemblies and the entrypoint function is invoked:

- The entrypoint is the `Main` function of the `Program` class, which simply starts executing the Blazor framework.

- The Blazor framework, which is now being executed in the browser by the Mono runtime, will take over. Blazor will then render the root component inside the `<app>` element. The root component is configured in the `Startup` class and by default will be the `App.cshml` component.

- The root component simply includes a `<Router>` Blazor component which starts the client-side routing. The Blazor page matching the current route will be rendered, which will be the `Index.cshtml` page for the root URL.

This whole process comes at a cost and it currently takes about 1.4 seconds before you see the application rendered.

If you profile the app, you will notice that about 300ms from the start the mono.wasm is downloaded (which is about 700Kb gzipped). It then takes another 700ms for it to be processed before the download process of all the required assemblies begins, followed by the Mono runtime starting the application. In about 1.4 seconds, you will see the first paint of the application:

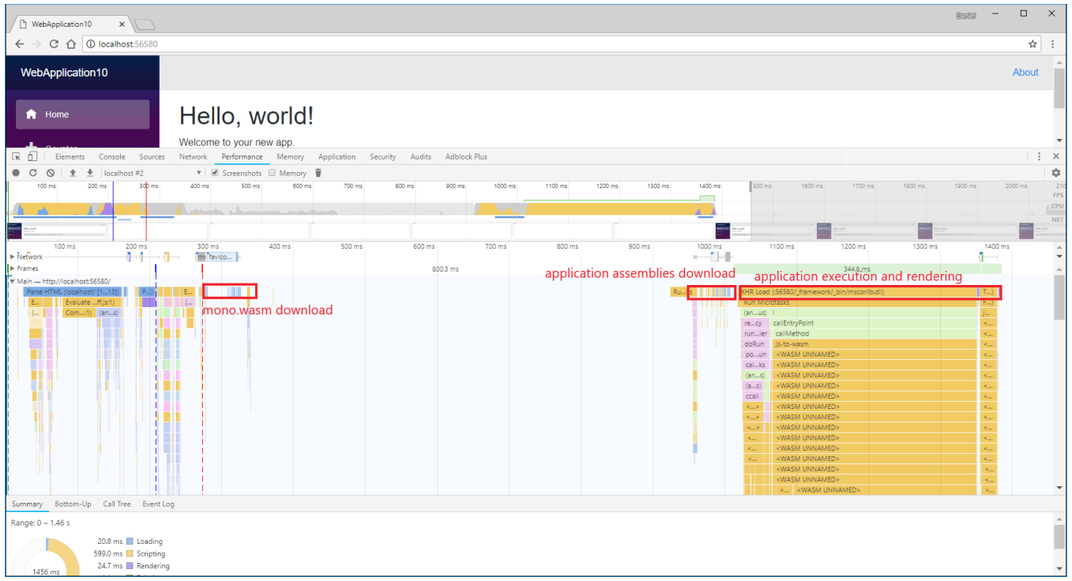


Figure 8, Monitoring application start

We also need to reinforce the idea that currently the only WebAssembly module is the Mono runtime.

- Every other assembly needed by your application is compiled as a regular .NET Standard 2.0 dll, downloaded by the browser and interpreted by the Mono runtime's own IL interpreter.

- This means your assemblies won't be compiled to WebAssembly and it is up to the runtime's IL interpreter to do a good job at executing assemblies in WebAssembly (it currently doesn't).

Remember this is the current mode of how things work, but an alternate *Ahead of Time* mode is being explored where the assemblies would also be compiled to WebAssembly and promises much better performance.

There is another performance problem when it comes to updating the DOM so Blazor can render your app and any updates that come after:

- Since WebAssembly code cannot directly access the DOM, for Blazor to render anything and update the current DOM, it needs to go through the interop bridge provided by blazor.webassembly.js.

- Blazor .NET code will try to figure out the DOM elements that need to be updated which are then passed to blazor.webassembly.js for them to be rendered.

This process is better explained in the Learning Blazor website, but together with the issue described above, remains one of the current major performance bottlenecks as you end up relying on JavaScript code for any rendering.

There are proposals for WebAssembly to gain better access to the DOM but it will take some time before they realize.

A TOUR OF BLAZOR FUNCTIONALITY

In this section, we will take a brief look at some of the functionality you can find today in Blazor, much of which will appear familiar if you ever worked with Razor in ASP.NET.

You can read through the official docs for a more detailed (and possibly up to date given the experimental and fast paced nature of Blazor) overview of the features. The Learn Blazor website is also worth a visit, as well as the Awesome Blazor curated listing.

Pages and components

Blazor leverages the Razor templating engine extensively, which means if you are already familiar with it from writing ASP.NET applications, you will have an easier time writing Blazor pages and components.

- Note that the directives available vary with respect to the ones in an ASP.NET Core application (for example the `@model` directive doesn't make sense here and there is no concept of "Razor pages").

- For a full list of the Razor directives and features supported, check out the official docs.

Razor syntax

Let's start by taking a look at the simplest of the examples, the `Index.cshtml` page located inside the Pages folder:

```
@page "/"
<h1>Hello, world!</h1>
Welcome to your new app.
<SurveyPrompt Title="How is Blazor working for you?" />
```

The Razor syntax should be familiar, with its mixture of HTML, directives like `@page` and custom

components like SurveyPrompt.

- The @page enables client-side routing for the page, associating the route "/" with this page.
- The SurveyPrompt component is another Blazor component defined inside the Shared folder. From the Index page we are providing a value for its Title property

So, what does the SurveyPrompt component looks like? If you open the SurveyPrompt.cshtml file you will see:

```
<div class="alert alert-secondary mt-4" role="alert">
  <span class="oi oi-pencil mr-2" aria-hidden="true" />
  <strong>@Title</strong>
  …
</div>

@functions {
  [Parameter]
  string Title { get; set; }
}
```

It is similar to the Index page except that it contains the simplest of examples on how the pages/components can include logic written C#.

- The @functions directive allows you to define the logic of the component, where in this case we are just declaring a parameter named Title by just adding a property of type String with that name. This is the parameter provided from the Index page.

- The template section shows how the parameter can be used as part of the HTML that will be rendered for the component, using @Title. This is nothing but standard Razor syntax, where @ symbol transitions from HTML into C# code and implies the expression that follows should be evaluated at runtime, converted to a string and inserted into the resulting HTML.

Reactivity

One of the main differences when working with Blazor is that you are now writing a client-side SPA application, which runs in the browser and should react and re-render its components whenever the user interacts with it.

This is different from server-side Razor, which renders an HTML page for a given request, and is done as soon as the HTML page is ready and sent to the browser.

What this means is that your pages and components are now **reactive**.

This is a concept familiar to anyone working with client-side frameworks like Angular, React or Vue and can be easily illustrated with an example.

Let's update the Index page so it includes an input for the user to enter his/her name, which will then be used to update the Title property provided to the SurveyPrompt component:

```
@page "/"
<h1>Hello, world!</h1>
<p>Welcome to your new app @Name.</p>
<input bind="@Name" type="text" class="form-control" placeholder="Name" />
<SurveyPrompt Title="@Title" />
@functions{
  public String Name { get; set; }
  public String Title => $"How is Blazor working for you, {Name}?";
}
```

When you run the updated project (you will need to restart it since it does not automatically do so when the code changes), you should see the following updated Index page. Notice how the name you enter is used both for the welcome message in the Index page and the Title of the SurveyPrompt component:

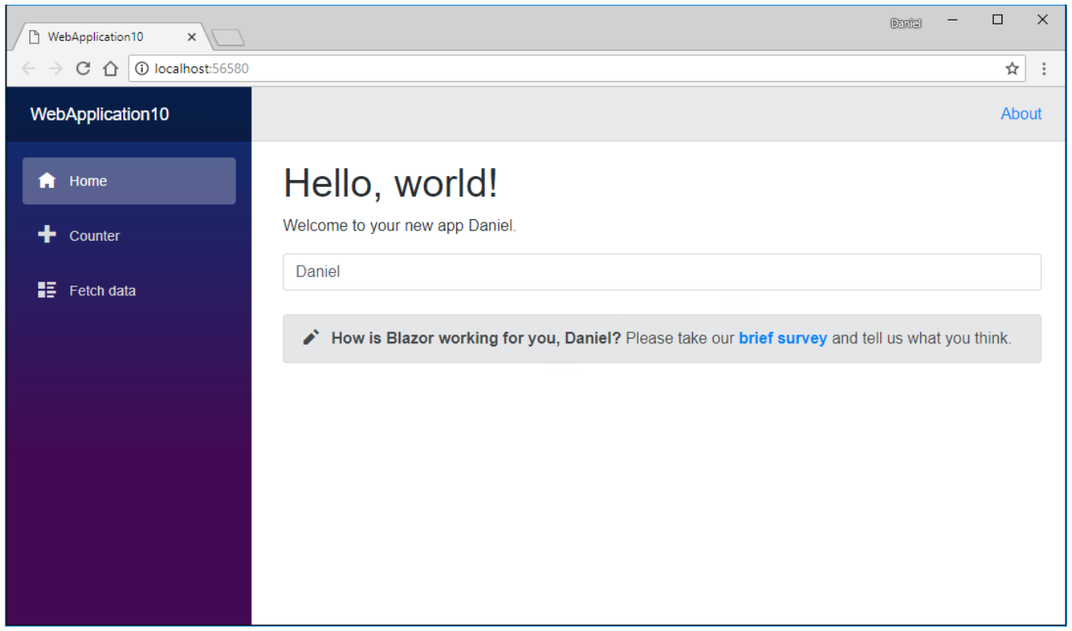


Figure 9, reactivity and data binding

If you update the value of the input, both the Index page and the SurveyPrompt will be re-rendered, updating the DOM to reflect the changes.

And not only that, it will update only the DOM elements that have been affected, since it knows that only the <p> of the Index page and the element of the SurveyPrompt component were affected by the change to the Name value.

As you can see, the reactivity means your components are instantiated and alive while they are part of the current page, and they will react to DOM events and data changes accordingly.

You can tap into this lifecycle of a component by implementing any of the lifecycle methods. This will be called by Blazor at specific points in the life of a component, for example let's update the SurveyPrompt component so it renders a list with an entry for each event:

```
<div class="alert alert-secondary mt-4" role="alert">
  …
</div>
<ul>
@foreach (var evt in LifecycleEvents) {
  <li>@evt</li>
}
```

```
</ul>
@functions {
  [Parameter]
  string Title { get; set; }
  List<string> LifecycleEvents { get; set; } = new List<string>();
  protected override void OnInit() =>
  LifecycleEvents.Add("OnInit");
  protected override void OnAfterRender() =>
  LifecycleEvents.Add("OnAfterRender");
}
```

Now run the application and change the Name using the input. Notice how the SurveyPrompt component was initialized once but rendered every time the Name changed:

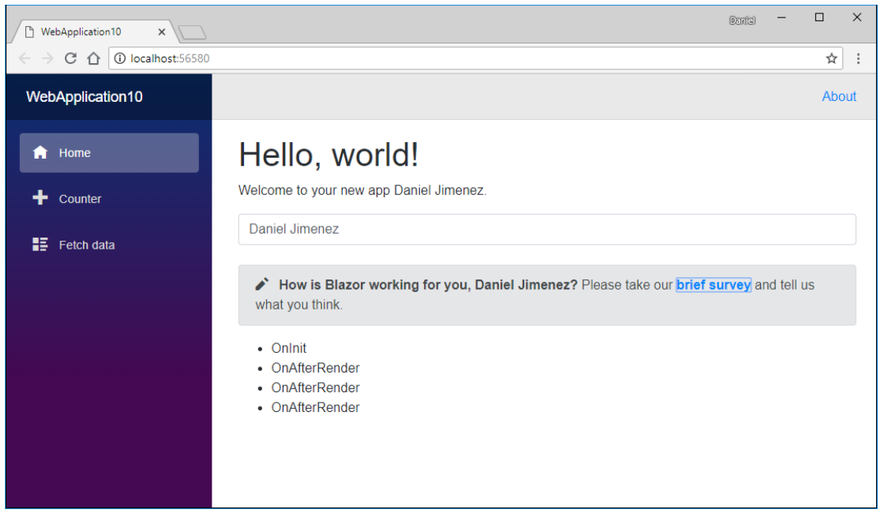


Figure 10, Some of the lifecycle events of a component

For a more detailed discussion of the lifecycle methods available see the official docs.

Data binding and event handling

We have already seen an example of how to bind a C# property to an attribute of a DOM element, however it might have gone unnoticed. When we added the input to the `Index` page using the `bind` attribute we were in fact adding 2-way data binding between our `Name` property and the `value` property of the input DOM element.

Under the hood the `bind` attribute is setting one-way binding between the C# property and the DOM property, plus an event handler for the DOM event so the C# property gets updated too.

Let's start by seeing how you can bind to an event of any DOM element using the `on{eventName}` attribute. All you have to do is set the attribute to a method with the right parameters (or none if you don't need to use them). Replace your previous changes to the Index.cshtml with:

```
<input value="@Name"
  onchange="@OnChange"
  oninput="@OnInput"
  onfocus="@OnFocus"
  onblur="@OnBlur"
  onclick="@(() => DOMEvents.Add("on click. Inline lambdas are possible too!"))"/>
<ul>
@foreach (var evt in DOMEvents)
{
  <li>@evt</li>
}
</ul>
@functions{
  public String Name { get; set; }
  List<string> DOMEvents { get; set; } = new List<string>();

  void OnChange(UIChangeEventArgs e) => DOMEvents.Add($"on change: {e.Value}");
  // Current Blazor issue prevents the updated input value
  // to be sent in the input event: https://github.com/aspnet/Blazor/issues/821
  void OnInput(UIEventArgs e) => DOMEvents.Add($"on input");
  void OnFocus() => DOMEvents.Add("on focus");
  void OnBlur() => DOMEvents.Add("on blur");
}
```

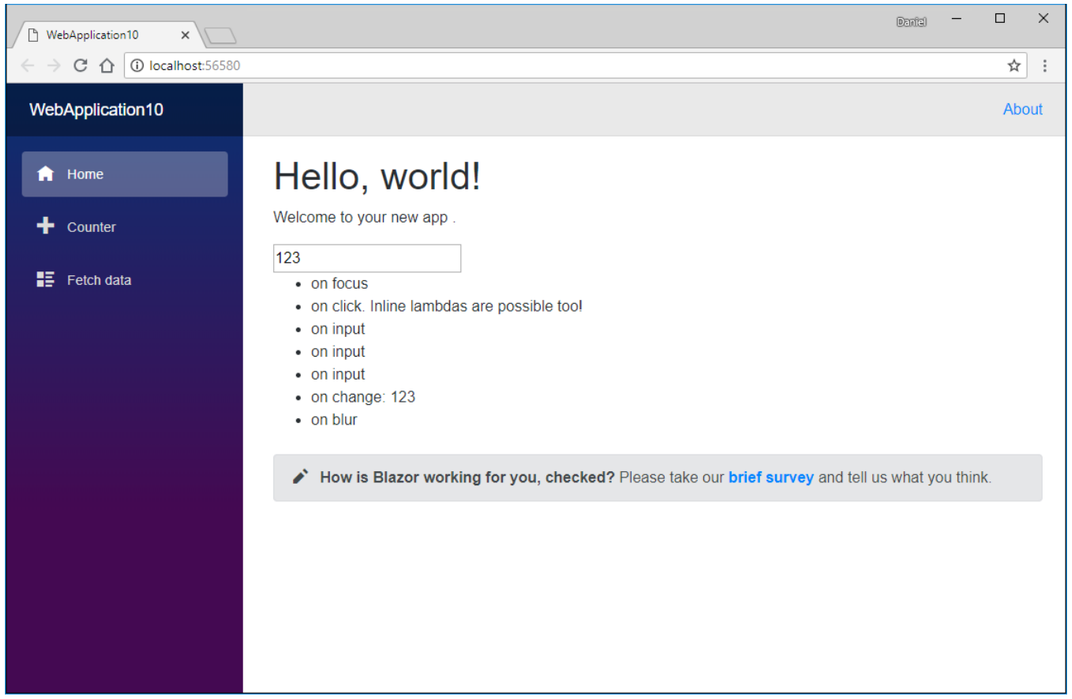


Figure 11, Data binding and event handling

As the example shows, you can listen to any DOM event using the `on{eventname}` attribute and assigning to it a component function with the required signature. Feel free to study the `Counter.cshtml` page in the project template which shows another event handling example.

Let's move onto data binding next.

By now it should be clear that you can perform one-way data binding by simply setting any valid DOM property to a C# expression. For example:

- Set the `value` property of a text input element with `<input type="text" value="@MyValue" />` where `MyValue` is a property of your page/component of type `string`.

- Set the `checked` property of a checkbox element with `<input type="checkbox" checked="@MyValue" />`, where `MyValue` is a property of your page/component of type `bool`.

However, this only gives us one-way data binding from C# to the DOM. If the value of the C# property changes, the property of the DOM element will be updated but not the other way around.

This might be fine for setting DOM properties like `disabled`, `readonly` or `class`, but in many cases and particularly with inputs, you will need two-way data binding with DOM properties like `value` or `checked`.

This is what the `bind` attribute achieves, by automatically setting one-way binding from C# to DOM plus an event handler to update from DOM to C#. So, the bind directive used here:

```
<input bind="Name" type="text" class="form-control" placeholder="Name" />
```

...is the same as manually setting the one-way binding with the input's value property and listening to its change event to update:

```
@using Microsoft.AspNetCore.Blazor;
<input value="@Name"
  onchange="@((UIChangeEventArgs e) => Name = (string)e.Value)" />
```

In its more general form, you specify both the DOM property and event you want to use:

```
<input bind-value-onchange="@Name" type="text" />
<input bind-checked-onchange="@MyValue" type="checkbox" />
```

But you can simply bind to a C# property and let the attribute figure out both the DOM property and event:

```
<input bind="@Name" type="text" />
<input bind="@MyValue" type="checkbox" />
```

Before we move on, do you remember the JavaScript library `blazor.webassembly.js`? That library plays a crucial role in making event handling possible.

The DOM events need to be handled by JavaScript code part of that library (since WebAssembly doesn't have direct access to the DOM) which will in turn call a method in the .NET code of Blazor that will finally dispatch the event to your component.

You can see this in the Blazor source code of its JavaScript library which invokes a .NET method using the interop method `DotNet.invokeMethodAsync` (more on this later):

```
return DotNet.invokeMethodAsync(
  'Microsoft.AspNetCore.Blazor.Browser',
  'DispatchEvent',
  eventDescriptor,
  JSON.stringify(eventArgs.data));
```

This means there is another penalty hit in handling events when compared to JavaScript frameworks which can directly interact with the DOM!

Structuring component's code

So far, we have only seen components whose code is declared inline, with its code inside a `@functions` directive. This isn't the only option and you can extract the code for a component to its own class using the `@inherits` attribute.

Let's update the `Counter.cshtml` page so we move its code to a separate file.

Add a new class `CounterBase` into a file `CounterBase.cs` file next to the page. Update the new class so it from `BlazorComponent` (you will need to use the namespace `Microsoft.AspNetCore.Blazor.Components`)and then move the code from the `@functions` directive into the new class. You will need to make elements protected so they can be accessed from the page:

```
public class CounterBase: BlazorComponent
{
  protected int currentCount = 0;
  protected void IncrementCount()
  {
    currentCount++;
  }
}
```

Next remove the `@functions` directive from the page and include an `@inherits CounterBase` directive. The page should end up looking like:

```
@page "/counter"
@inherits CounterBase

<h1>Counter</h1>
<p>Current count: @currentCount</p>
<button class="btn btn-primary" onclick="@IncrementCount">Click me</button>
```

There are a couple of small caveats with this approach.

The first one is that we need to use a different class name than the page, since the page will always be compiled into its own class which will inherit the one we create with the logic. The second is that since we rely on inheritance, we need to make members public/protected or they won't be accessible from the page.

However, it is possible to avoid having a cshtml file and rely on the POCO class which will then contain regular C# code to render the template. This could be an interesting approach, particularly when the HTML of the template will be simple or highly dynamic. You can even mix and match this approach with parts of the template generated in the C# code.

Read more about it in the Learn Blazor website.

Dependency injection

The logic of your components and pages will not always be so simple that it can be self-contained.

Often you will need to use other classes that provide access to functionality your components need. A very typical example is making HTTP requests, for which you need to use the `HttpClient` class.

Since Blazor is built on top of .NET Core, it comes with dependency injection support out-of-the-box. If you are familiar with dependency injection in ASP.NET Core, the support here is very similar to the default container of ASP.NET Core.

You register services in the `ConfigureServices` method of the `Startup` class, this will make them available within the framework and injected into any Blazor component using the `@inject` directive. The default project template shows an example of this approach in the `FetchData.cshtml` page which gets an instance of the `HttpClient` injected so it can send an HTTP request. (This is a framework service which is always available so you don't need to register it)

```
@page "/fetchdata"
@inject HttpClient Http

<h1>Weather forecast</h1>
<p>This component demonstrates fetching data from the server.</p>
…

@functions {
  WeatherForecast[] forecasts;

  protected override async Task OnInitAsync()
  {
    forecasts = await Http.GetJsonAsync<WeatherForecast[]>("sample-data/weather.
    json");
  }

  class WeatherForecast
  {
    public DateTime Date { get; set; }
    public int TemperatureC { get; set; }
    public int TemperatureF { get; set; }
    public string Summary { get; set; }
  }
}
```

If you use the inheritance approach to move your logic into a class inheriting from BlazorComponent, you will need to use property injection (remember the class generated from the cshtml file actually inherits from it, so it won't know how to pass any constructor properties!):

```
public class MyComponentBase: BlazorComponent
{
  [Inject]
  protected HttpClient client { get; set; }

  …
}
```

Component libraries

Blazor being a client-side framework for SPA applications, it is no wonder developers would like to create libraries of reusable components. While the templates available from Visual Studio don't show this, there is a template for the `dotnet new` CLI that allows you to create component libraries which can then be referenced from other projects.

Simply run `dotnet new -i Microsoft.AspNetCore.Blazor.Templates::*` in the console and you should now see a new template `dotnet new blazorlib` which allows you to create a new project with a Blazor library of components.

There is a great introduction to creating component libraries in this article from Chris Sainty. And if you take a look at the Awesome Blazor curated list of components, you will see the community has started to provide both specific components and general libraries providing Bootstrap4 or Material Design components!

Client routing

Blazor provides simple client-side routing. Your root `App.cshtml` component will include a `<Router>` Blazor component which will be in charge of listening to URL changes and rendering the page that matches the new URL.

- Each page is associated with one (or multiple) URL(s) through the usage of the `@page` directive. For example, the Index page uses `@page "/"` while the Counter page uses `@page "/counter"`.

- Behind the scenes, the generated classes for those cshtml files just include a `[Route]` attribute as in `[Route("/counter")]`.

The route segment associated with each route can include route parameters, which you can later access in the component code.

It is worth mentioning that currently there is no support for optional route parameters, so you would need to either use two `@page` directives or two `[Route]` attributes, one with the parameter and the other one without:

```
@page "/counter"
@page "/counter/{Step:int}"

<h1>Counter</h1>
<p>Current count: @currentCount</p>

<button class="btn btn-primary" onclick="@IncrementCount">Click me</button>

@functions{
    int currentCount = 0;
    [Parameter]
    int Step { get; set; } = 1;
    void IncrementCount() => currentCount += Step;
}
```

One caveat with routing is that it uses normal URLs and there is no hashed mode.

That is, a `@page "/counter"` will be accessible at the `/counter` URL and there is no option to use `#/counter`.

While this is something desirable in most cases, it means that the server hosting the Blazor app should be configured to redirect all URLs that don't match a static file to the index.html file, which will then bootstrap the application in the browser and will ultimately perform the navigation.

If you use the *Blazor (ASP.NET Core Hosted)* template, this is something taken care of by default, as the ASP.NET Core server includes middleware that handles this. The project also comes with a web.config to set things up in IIS which is how the non-hosted project template can be run in development with IIS Express.

However, if you plan on hosting the application yourself using a different server (which is totally possible since all the files are static files which will be downloaded by the browser), then you need to be aware of

this caveat.

See the official docs for more information on hosting Blazor apps.

HTTP requests

The `HttpClient` class is available in the dependency injection container by default and is the recommended way of performing HTTP calls from Blazor components. You can see an example in the `FetchData.cshtml` page which we discussed briefly in the section about dependency injection.

```
@page "/fetchdata"
@inject HttpClient Http
<h1>Weather forecast</h1>
…
@functions {
  WeatherForecast[] forecasts;
  protected override async Task OnInitAsync()
  {
    forecasts = await Http.GetJsonAsync<WeatherForecast[]>("sample-data/weather.
    json");
  }
  class WeatherForecast { … }
}
```

You might be wondering about the JSON specific methods available in the HttpClient since those are not part of the standard HttpClient class.

- These are extension methods provided by Blazor, in order to make it easier dealing with requests sending/receiving JSON objects, which will happen very often when dealing with REST APIs from your client-side application.

- These extension methods use under the hood the SimpleJson library (which is embedded in the source code, rather than referenced as a Nuget package and had some changes applied) rather than JSON.Net which is the one used in ASP.NET Core applications.

You should also be aware of how HTTP requests are possible from Blazor.

- It is important to understand that even if we have .NET code running in the browser, not all the libraries will work since many will depend on APIs provided by the OS which won't be available in the browser. An example would be using the `PhysicalFileProvider` class to try and access local files and folders.

- This also applies to the class available in .NET for networking. The reason why `HttpClient` works and is the recommended way of making HTTP requests from Blazor is that behind the scenes it has been configured so the actual request happens in JavaScript using the browser fetch API. Read more about this on the Learn Blazor website.

Both points might introduce further friction and gotchas compared to ASP.NET Core so you might want to stay wary.

JavaScript interop

When working on Blazor applications there will be times where you will need to integrate with existing browser APIs and/or JavaScript libraries, particularly in these early days while functionality still needs to be added to Blazor and WebAssembly.

Blazor provides Interop functionality to call JavaScript from C# and vice versa. Calling a JavaScript method from C# is simple, as long as the method is accessible from the window object (that is, it needs to be accessible as `window.some.method`)

- Add the JavaScript function into an existing JS file part of the project or a new `<script>` block in the index.html:

```
window.MyModule = window.MyModule || {};
window.MyModule.alert = (message) => {
 return alert(message);
};
```

- In the component, use the `InvokeAsync` method of the IJSRuntime interface, passing the name of the JavaScript function and any arguments:

```
@page "/interop"

<h1>Interop</h1>
<input bind="MessageToAlert" class="form-control" />
<button class="btn btn-primary mt-2" onclick="@ShowAlert">Show alert</button>

@functions{
    string MessageToAlert { get; set; }
    Task<object> ShowAlert()
    {
        return JSRuntime.Current.InvokeAsync<object>("MyModule.alert", this.
MessageToAlert);

    }
}
```

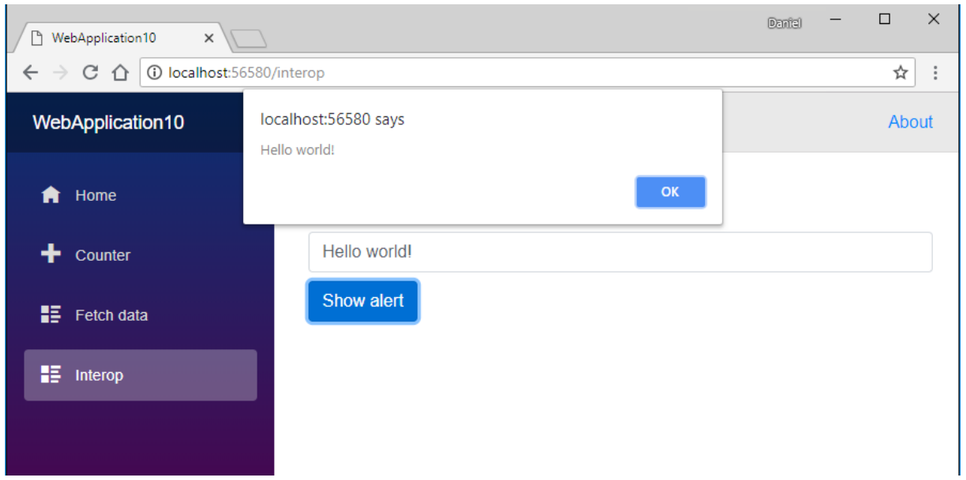


Figure 12, calling a Javascript method from .NET

Calling C# from JavaScript is equally simple. The only caveat to consider is whether you want to call a static method or an instance method, since in the latter case you will first need to pass the instance reference to JavaScript.

You can read more about it in the official docs. You can also browse the existing components in the Awesome Blazor listing, since many deal with JavaScript interop.

BLAZOR - OPEN QUESTIONS AND LIMITATIONS

As impressive as the work put into Blazor and its current results are, this is still considered an experiment by Microsoft and it's hard to know what its future will be.

In this section I will briefly discuss some of the limitations I have found which might or might not be addressed at some point (or might not be that important to you!).

Performance

The first one is the performance.

The promise of efficient code execution in the browser at close to native-code like speed is seriously affected by two issues:

- Only the Mono runtime is compiled to WebAssembly, with the rest of the .NET code relying on Mono's own IL interpreter to be executed.

- Direct access to the DOM and browser API's from WebAssembly is currently not possible which means the .NET code will find itself relying on JavaScript interop for rendering, event handling, HTTP requests and more.

You can read more about these issues and particularly the first one in several GitHub issues for the Blazor repo like this one and this one.

The first issue is expected to be improved once the AoT mode of the Mono runtime becomes a reality, but the second depends on the WebAssembly standard moving forward and being implemented by major browsers.

It is also not possible right now to compile your application into a combined bundle, instead the assemblies are downloaded individually and loaded into the Mono runtime. While the sizes of the assemblies seem small and they can be cached, the number of requests made on application startup is significant, which can be a problem with bad connections.

It seems to me that the WebAssembly promises are negated in part by these issues. And if the dependency on JavaScript interop for most of the tasks that need some sort of IO doesn't change, you might as well use a JavaScript framework unless you have some CPU intensive code to run on the client.

At the same time, I understand that for many, avoiding JavaScript will be a reason enough to use Blazor, provided it gets on par with major JS frameworks in terms of performance and functionality.

Tooling

This one is obviously caused by being so early days for Blazor. For example, live reload doesn't seem to work with the simplest of the templates, so you need to restart the project after making changes, unless I am missing something very obvious.

Debugging is also challenging, although a debugger extension for Chrome just got its first release with limited functionality. Since there are quite a few pieces involved between JavaScript libraries, .NET code and WebAssembly, and it is not always obvious how things work, good debugging support will make a huge difference.

Components and JavaScript tooling

Since WebAssembly and Blazor do not intend to replace existing web technologies but rather coexist with them, I would expect at some point better integration with existing tooling and libraries for web applications.

For example, there is no obvious way to create component style rules.

I have seen community attempts like Blazorous but I don't see the framework itself providing a clear direction for you to use LESS/SASS/Stylus, CSS pre-processors, bundling, etc. It would be great if a component could declare a style block in a specific language and these were extracted and processed by a tool like webpack (or even integration with existing tooling like webpack) into a combined CSS bundle.

A similar point can be raised about JavaScript code for components with interop needs.

The one solution available right now would be a component library where your Blazor code, CSS and JS is encapsulated. However even in such a case, when you import several of those components from libraries, you might still want to process their CSS/JS files with modern tooling like webpack.

In general, there doesn't seem to be a clean way to develop the JS/CSS part that will inevitably be needed for some of the components of your application. (at least for the foreseeable future until WASM gets better DOM support and libraries are ported to Blazor/WASM).

This can get even more annoying since it seems a non-trivial number of CSS/JavaScript libraries might be used by any given project (given the current WebAssembly and Blazor limitations), and with them comes all the modern JavaScript tooling.

You could always use tools like webpack to process all the assets in JS/CSS files outside Blazor components, but even that wouldn't be such a trivial setup to achieve for production and development purposes, so examples and guidance would be much appreciated.

SERVER SIDE BLAZOR

Just before we finish, let me briefly introduce an alternative way of running Blazor which was recently announced. It is now possible to run Blazor on the server and update the DOM and handle browser events via a thin JavaScript layer and a permanent SignalR connection.

This means you will be running your Blazor client code on the server together with the rest of your application code. Think about it, you can execute a framework in your server whose initial focus was to run .NET code in the client using WebAssembly!

This was made available with the release 0.5.0 of Blazor, and you should see an option *Blazor (Server side in ASP.NET Core)* when creating a new project:

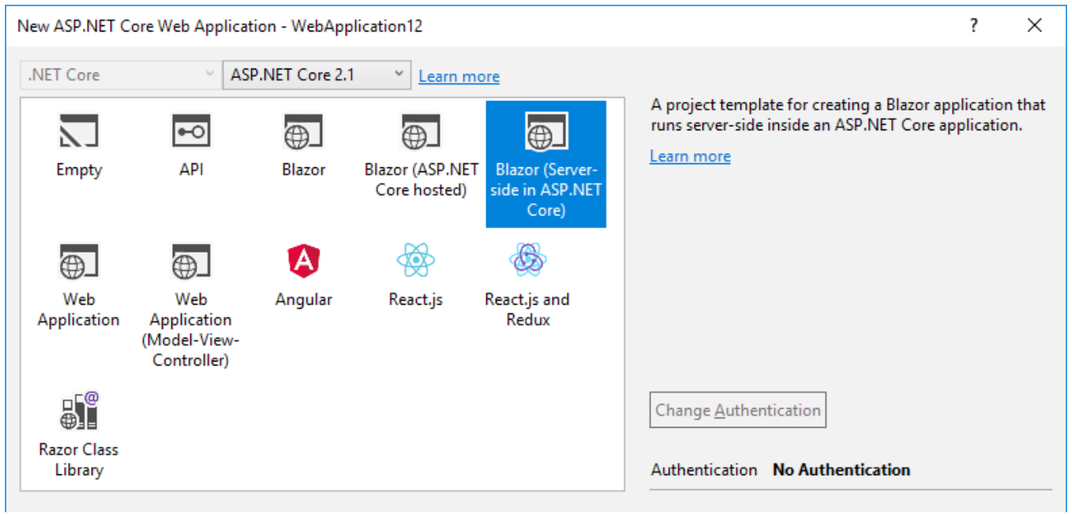


Figure 13, The server side Blazor template

If you create an application using this template, you will see that two different projects are added, a Blazor app and an ASP.NET Core server.

If you look at your index.html file you will see that `blazor.server.js` is included instead of `blazor.webassembly.js`. When your app starts, this file will establish a connection with the ASP.NET Core server, which will run the actual Blazor application (as well as the ASP.NET Core server) and any interaction with the browser and/or DOM will be propagated through a SignalR connection.

The same JavaScript functions that were used in the client side Blazor templates to update the DOM (since WebAssembly cannot access the DOM itself) are used here to render DOM changes which will be pushed through the SignalR connection.

When you inspect the network requests from the browser you will see it only needs to download the blazor.server.js script, the blazor.boot.json file and then establish a connection:

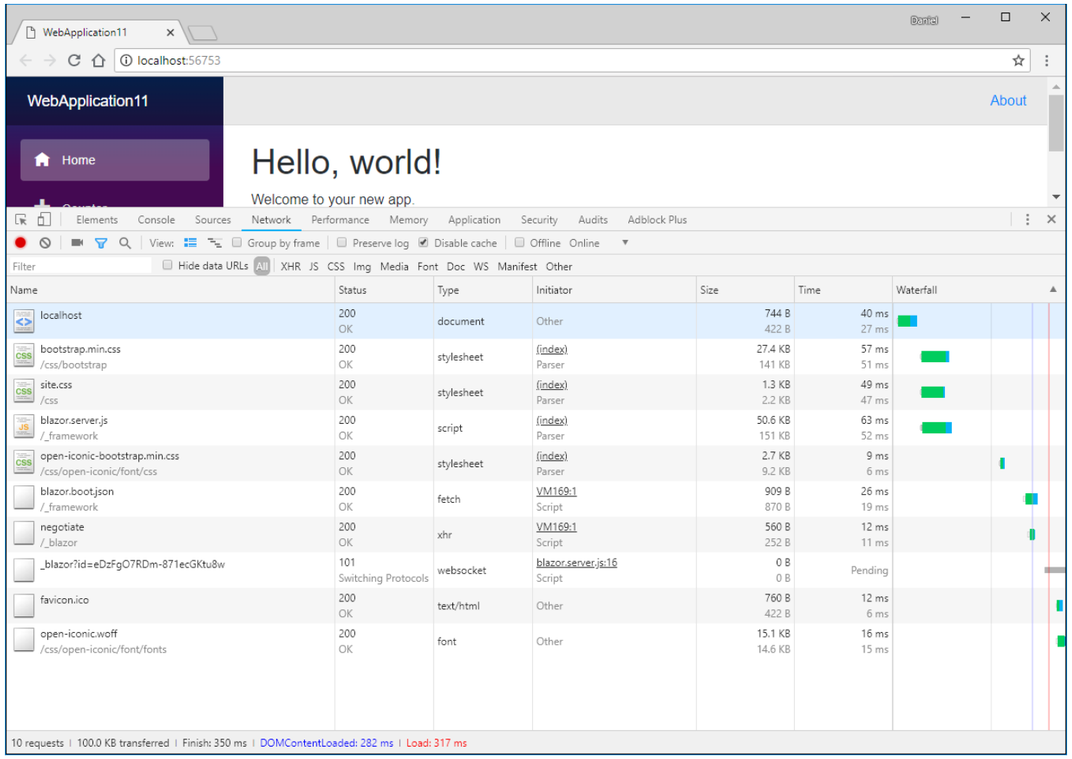


Figure 14, Faster bootstrap when using server side Blazor

This has some pros like the initial download and render of your app being really fast, immediate access to all the tooling for .NET like the debugger, or having no need for AJAX requests to fetch/update data.

Of course, it has downsides as well like every interaction with the browser requiring communication through the SignalR connection and concerns about how this model would scale with the number of concurrent users.

While the official docs contain a brief section describing the server side model , I have also found this article from Ankit Sharma quite interesting.

You can clearly see the experimental nature of the framework! I am sure we will keep seeing new ideas being implemented and some being dropped as the Blazor team gets feedback and WebAssembly/Mono keep evolving.

Conclusion

Blazor is impressive. It leverages a number of technologies in smart ways in order to provide a SPA framework that can run .NET code in the browser.

Its design is also quite flexible, something that is shown by the fact that you can decide to run it on the server and simply keep a SignalR connection to a JavaScript layer that deals with the DOM.

However, the novelty of these technologies and its experimental nature also means it is very early days for it to be a serious option when starting your new application. Expect some serious limitations both in terms of functionality and the tooling available, but nonetheless what is available today is already impressive and works better than you might expect.

Whether Blazor and WebAssembly will be able to fulfill its promises is something we will find in time! However, the pace at which the Blazor team and the community are pushing forward, is well worth staying tuned for.

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 Damir Arh *for reviewing this article.*

Tim Sommer

CQS
A Simple but Powerful Pattern

This tutorial is an introduction to designing software using the Command Query Separation (CQS) pattern. This guide helps create an architecture that structures your application in a clear way, allowing you to write code that is readable, reusable and less error prone.

Introduction

When you choose an architecture, a lot of factors come into play.

These factors, let's call them architectural goals, include (but are not limited to) : the complexity of the system

- ease of use
- simplicity
- scalability
- auditability
- performance and
- reusability.

In this article, I'll focus on how to design software using the Command Query Separation (CQS) pattern. This will allow us to tackle a lot of these goals; and to keep code readable, structured, reusable and less error prone.

Command Query Separation (CQS)

Bertrand Meyer devised CQS, or Command Query Separation, as part of his work on the Eiffel programming language. In case it's the first time you have ever heard about this term, I encourage you to read the formal definition of CQS on Martin Fowler's blog.

The pattern itself is pretty simple. It all comes down to two principles.

Every method should either be a **command** that performs an action, or a **query** that returns data to the caller, **but never both**.

Methods should return a value only if they create no side effects.

Simply put: a query should *never mutate state*, while a command *can mutate state but should never have a return value*.

So, what does that really mean? In its most basic form, you separate queries from commands by **function:**

```
interface IBookService
{
  void Edit(string name);
  Book GetById(long id);
  IEnumerable<Book> GetAllBooks();
}
```

The `Edit` function does not return a value, its only job is to edit a book by name. We are certain that this function will have *side effects* as it will mutate data and does not have a return value. So, this is a command.

The `Get` methods have no *side effects* (they do not mutate the state). These actions are responsible for fetching the data, but not changing it. These are queries.

In practice, you will most likely separate queries from commands by object. Which means you will have a `GetBookByIdQuery`, a `GetAllBooksQuery` and an `EditBookCommand`. Either a `QueryHandler` or a `CommandHandler` will typically handle these objects.

In this article, I will focus mainly on this approach.

So, the CQS pattern is fairly simple. If you follow it, your code will get more structured, more readable and more maintainable. It encourages you to write better code, that follows SOLID principles and allows the introduction of (better and more) Unit Testing.

Building an application using the CQS pattern

Now that we have a basic understanding of the CQS pattern, let's see how we would implement that into an application.

As we move from pattern to application, I will always separate Commands and Queries by an object, which if you take a high-level overview of, will look something as shown in Figure 1.

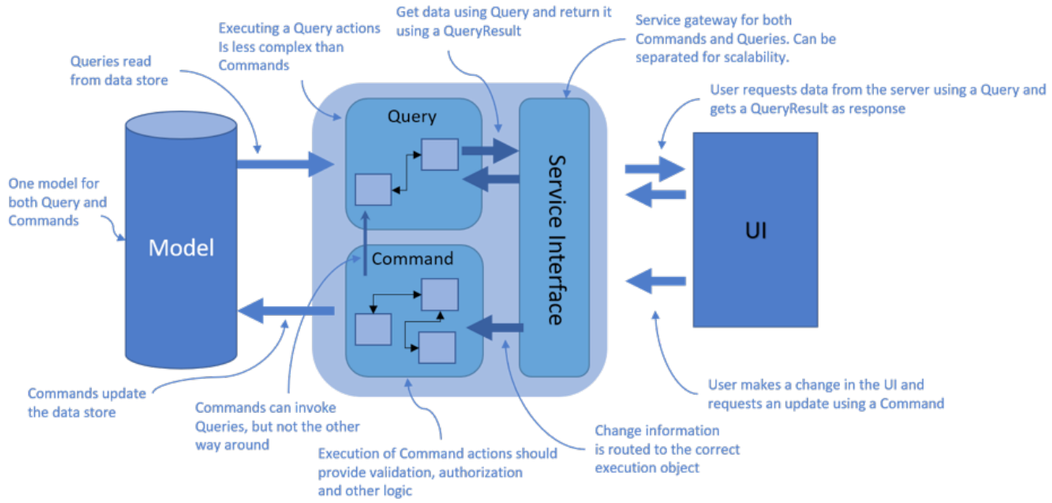


Figure 1: A high-level overview of an application built using the CQS pattern

Now, this is a bit more complex than the CQS pattern itself, but it should still be straightforward. Advantages of designing your software like this are:

- You'll get a decent level of separation of concerns (SoC). It will motivate you to work towards more SOLID code and write more unit tests.
- Code for reading and mutating data is completely separated. This logical separation (i.e. in code) automatically provides for a better structure on the method or class level. You can also implement (and enforce) performance optimizations more easily, which is especially valuable on the "read" side, as any modern application has more "reads" than "writes".
- A physical separation (i.e. different assemblies or projects that can be deployed on different servers) is possible and allows you to scale up on both "read" and "write" sides. But, this is fairly complex and leans more to CQRS (which we'll take a quick look at in the next section). But if you know what you are doing, it is possible.
- The CQS pattern is not that complex. Implementing it into your application is therefore also fairly straightforward.

While the advantages far outweigh the disadvantages, I encountered some:

- Separating Queries from Commands by the object can lead to a lot of classes. In a complex application, the number of Queries and Commands objects can get big, fast.
- If you rigidly follow the pattern, you could get stuck with unnecessary server calls. See "Some thoughts" in the "Building a CQS architecture" section for an example.

CQS vs CQRS?

Before we go further, I should state the difference between CQS (Command Query Separation) and CQRS (Command Query Responsibility Segregation). Even though they are very much related, they are not the same thing.

You could say that CQS works more on the method or class level, whilst CQRS takes the concepts of CQS and takes it to the application level. With CQS you separate methods for querying from or writing to a model. CQRS is similar, except it's more of a path through your system. A Query request takes a separate path from a Command.

Explaining CQRS in detail far outreaches the boundaries of this article as it is a much more complex subject.

But, I will leave you with this: We can say that the biggest difference between the two patterns is that, while not a requirement, CQRS allows for *separate data stores and models* (*data or ORM*) for Commands and Queries. The data between data stores is usually synchronized asynchronously using a service bus.

An architecture that is built upon the CQS pattern will follow the same logical separation, but Queries and Commands will be executed *on the same datastore and share the same model*. It should be noted though that you can use CQS in an application that does not have a data store at all.

CQRS applications are much more complex than CQS applications. The latter is much simpler, both to build and to maintain.

Building an architecture using the CQS pattern

I've written a sample application alongside this article in which you will see that implementing the CQS pattern within a basic architecture can be done easily. But, before we dive into the code, I'll give you a run-down of all the things we will be using.

The building blocks listed here are concepts I often use when designing software. But they are in no way required for building an application that follows the CQS pattern.

I've seen lots of different implementations online, which are all good! But I use the concepts listed below, which works great for me.

- **The Query, QueryResult, Command and CommandResult objects:** These are the actual classes used to make a request to the data store in the sample application. `GetBookByIdQuery`, `GetAllBooksQuery`, `UpdateBookCommand` and `DeleteBookCommand` could all be actions you'd find in a CQS architecture.
- **The Dispatcher object:** This is where the chain starts. You tell the Dispatcher to dispatch a Query or a Command object. The Dispatcher then passes your request to the correct Handler.
- **Abstract base class for both Command- and QueryHandlers:** These objects allow you to manage code that should be in all handlers. Functionalities like logging, authorization, exception handling, etc. can be implemented here.
- **The Handlers themselves:** The classes that handle your request and either retrieves data or

mutates data. There should always be one Handler mapping to one action. So, you would have a `GetBookByIdQueryHandler`, `GetAllBooksQueryHandler`, `UpdateBookCommandHandler` and `DeleteBookCommandHandler` that handle the Queries and Commands above.

- **IoC:** I use Autofac as Inversion of Control Container to glue everything together.

The execution of a Query in the sample application will go like this:

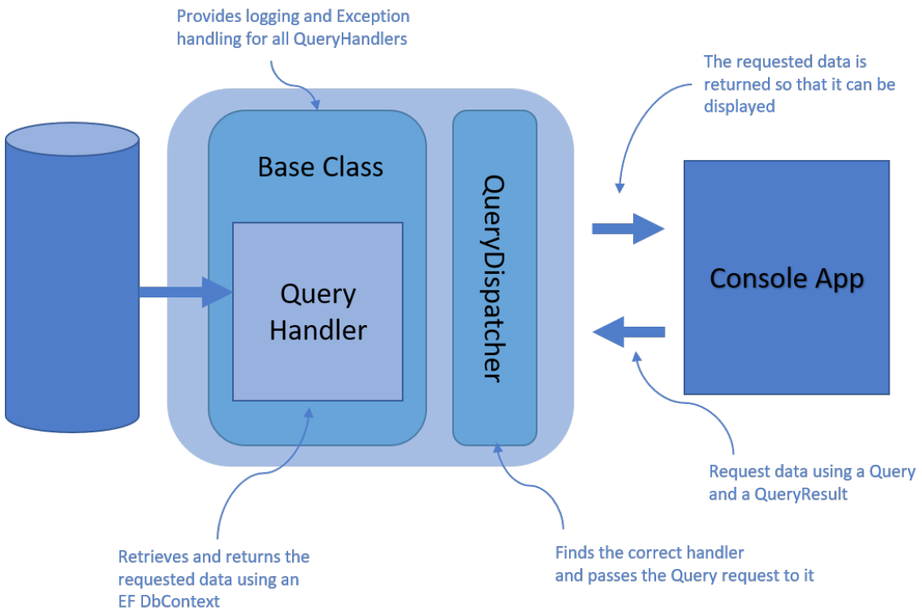


Figure 2: A representation of the execution of a Query as presented in the sample application.

For a Command, the execution will be almost the same:

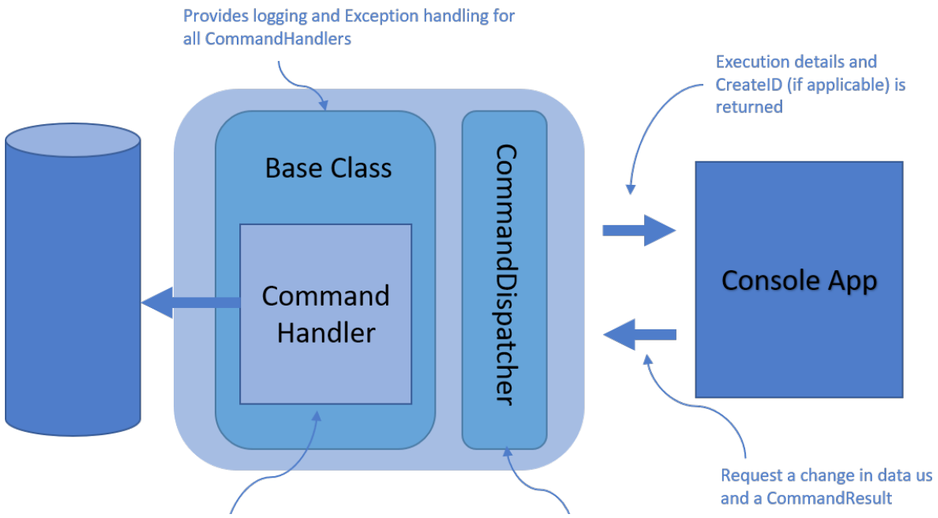


Figure 3: A representation of the execution of a Command as presented in the sample application

Some thoughts

In the sample application you'll notice I kept things as simple as possible.

A `CommandHandler` in a production application will do far more than just mutate data in your data store. There will be more logging, you'll have different layers of authorization, have Assemblers (a building block that maps Data Transfer Objects to Entity objects), have a Unit of Work (a building block that handles transactions, scopes and different contexts), perform data validation, maybe some data audits, etc.

But, in the sample, I kept the `CommandHandler` simple so you can grasp the key concept.

The CQS pattern states one thing clearly: a Command can only mutate state and cannot have a return value.

As I implement this pattern into an architecture, I allow (or even enforce) Commands to have a return value. The main reason to follow the CQS pattern is to keep code readable and reusable.

Your code gets more trustworthy as it ensures certain code will not cause unexpected side effects. A Query that changes state would be a clear and serious violation and I would never allow that.

But, for Commands I make an exception in two cases.

First, say you create a new entity. Your client now needs to know the ID of that newly created object.

Following CQS strictly, you now have two options: you could either create the ID on the client side (such as a GUID or a HiLo implementation), or the client could request the ID from the system using a Query before or after creating the object.

This is a silly constraint. I prefer to just return the ID when the create Command is executed. This makes things much less complex and allows for less server calls. In this case (and only this case), I allow the Command to return an ID.

The second case is to get details about execution. A Command should always return information about the mutation carried out by it. Did any exceptions occur? Were there any validation errors? Were there warnings or notifications the user needs to know about?

I generally include this information in my result, so my client is able to handle program flow based on the execution details.

It is not present in the sample application, but a `CommandResult` in a production application would basically look like this, including the Command Execution information:

```
public abstract class Result : IResult
{
  protected Result()
  {
    ValidationBag = new ValidationBag();
  }
  public ValidationBag ValidationBag { get; set; }
  public Exception Exception { get; set; }
  public bool Success { get; set; }
}
```

This topic is up for debate, so probably not everyone will agree with me on this.

But CQS is a pattern, a guideline. If it makes sense to deviate a bit, do it. Especially if it makes your code clearer, more readable and faster in execution! And if you don't like it, you can easily exclude it from the sample application.

The execution of a Query in a production environment would go something like the one in Figure 4:

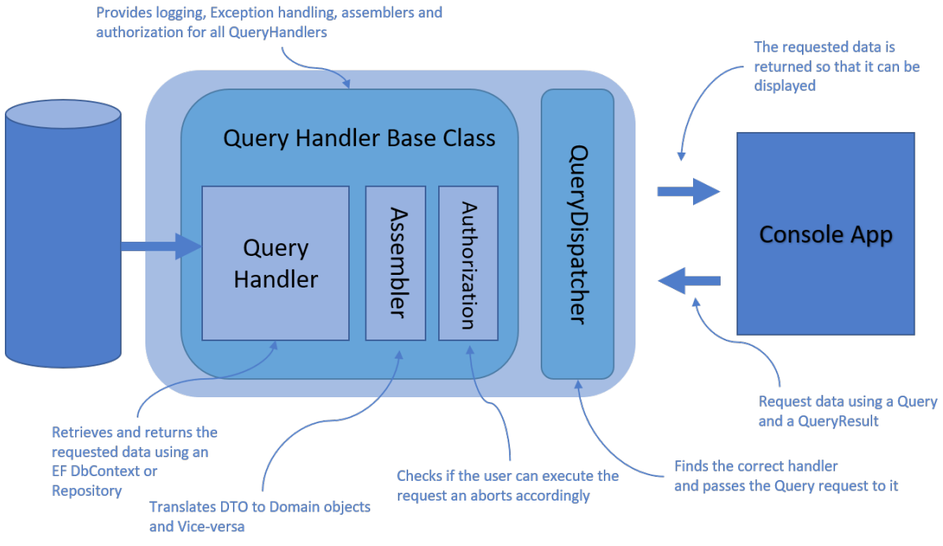


Figure 4: An example of the execution of a Query in a production environment.

A Command will have way more checks causing its execution to be a bit more complex:

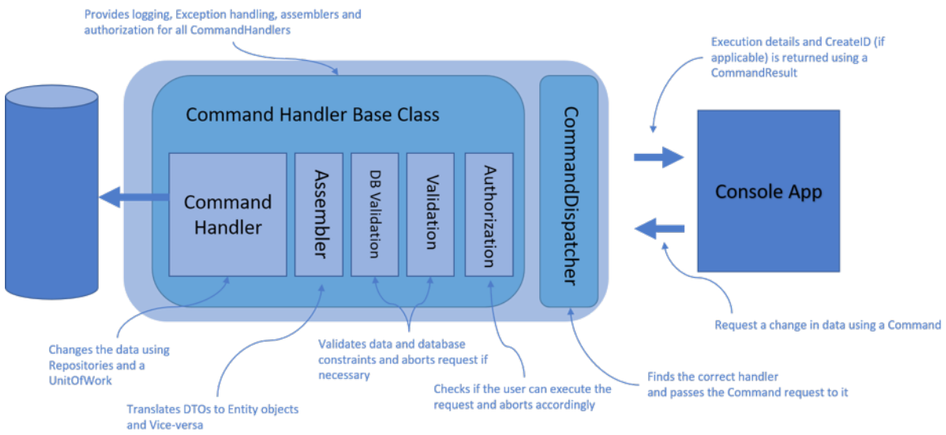


Figure 5: An example of a Command flow in a production environment.

Ok, you should now have a basic understanding of the CQS pattern, how we use it, when do we implement it in a Software Design and how it is different from CQRS.

Let's dive into the code!

The Sample

I've written a Console Application to explain the key concepts of the CQS architecture I built. It is not as fancy as a web application, but it will allow us to focus more on the key concepts.

You will also find that you can easily apply (or copy) these techniques and implement them in your own (web) application.

You can find the sample code on my github account. This article is not intended as a how-to guide, as this would require a huge amount of code samples. Instead I'll be focusing on the core concepts so you can get a better understanding on how the sample application works, and how CQS fits in the story.

A database of books

The sample application will perform basic create, read and update actions on a book database. We will be working with Entity Framework as the ORM, Autofac as the IOC Container and Log4Net as the logging framework. If you are not familiar with these topics you can find more information on EF here, and on Autofac here.

The source code can be found on the Github project under the folder 'Cqs.SampleApp.Console'. If you open the program.cs file, you will see the following:

```
private static readonly ILog _Log = LogManager.GetLogger(typeof(Program).Name);
private static void Main(string[] args)
{
  _Log.Info("Bootsrapping application..");
  var _container = Bootstrapper.Bootstrap();
  WithoutCqs(_container);
  WithCqs(_container);
  System.Console.ReadLine();
}
```

To allow you to examine and fully understand the CQS architecture, I've kept "the old way of doing things" available in the sample application. This "old way" (method `WithoutCqs`) uses the EF `DbContext` to simply get and update data.

But first, let's take a look at the static Bootstrapper class which configures the IoC container:

```
public static class Bootstrapper
{
  public static IAutofacContainer Bootstrap()
  {
    var _container = new AutofacContainer(builder =>
    {
      builder.RegisterModule<ApplicationModule>();
      builder.RegisterModule<CqsModule>();
    });

    return _container;
  }
```

```
public class ApplicationModule : Module
{
  protected override void Load(ContainerBuilder builder)
  {
    //register the EF DbContext
    builder.RegisterType<ApplicationDbContext>()
      .AsSelf()
      .WithParameter("connectionString", "BooksContext")
      .InstancePerLifetimeScope();
    }
  }

  public class CqsModule : Module
  {
    protected override void Load(ContainerBuilder builder)
    {
      var _assembly = typeof(CqsModule).Assembly;

      //register the QueryDispatcher and CommandDispatcher
      builder.RegisterType<QueryDispatcher>().As<IQueryDispatcher>();
      builder.RegisterType<CommandDispatcher>().As<ICommandDispatcher>();

      //Register all QHandlers and all CHandlers found in this assembly
      builder.RegisterAssemblyTypes(_assembly)
        .AsClosedTypesOf(typeof(IQueryHandler<,>));
      builder.RegisterAssemblyTypes(_assembly)
        .AsClosedTypesOf(typeof(ICommandHandler<,>));
    }
}
```

Here we basically glue all the components together as we register the correct implementations in our Autofac IoC Container.

We will be working with the `Book` class as our only Entity object, which is a representation of a table in the database:

```
public class Book : DbBaseModel
{
  public string Title { get; set; }
  public string Authors { get; set; }
  public DateTime DatePublished { get; set; }
  public bool InMyPossession { get; set; }
}
```

The "old way", `WithoutCqs` method looks like this:

```
private static void WithoutCqs(IAutofacContainer container)
{
  //resolve context
  var _context = container.Resolve<ApplicationDbContext>();

  //save some books if there are none in the database
  if (!_context.Books.Any())
  {
    _context.Books.Add(new Book()
    {
      Authors = "Andrew Hunt, David Thomas",
      Title = "The Pragmatic Programmer",
      Bought = true,
      DatePublished = new DateTime(1999, 10, 20),
    });

    _context.Books.Add(new Book()
    {
      Authors = "Robert C. Martin",
      Title = "The Clean Coder: A Code of Conduct for Professional Programmers",
      Bought = false,
      DatePublished = new DateTime(2011, 05, 13),
    });

    _context.SaveChanges();

    _Log.Info("Books saved..");
  }

  _Log.Info("Retrieving all books..");

  foreach (var _book in _context.Books)
  {
    _Log.InfoFormat("Title: {0}, Authors: {1}, Bought: {2}", _book.Title, _book.
    Author, _book.Bought);
  }
}
```

As you can, there's nothing complex here. We get the Database context from the IoC Container, add some data into the database, and then output the result to the console.

Our output looks like this.

```
Bootsrapping application..
Contructing autofac container
Resolve instance of type Cqs.SampleApp.Core.DataAccess.ApplicationDbContext
Retrieving all books..
Title: The Pragmatic Programmer, Authors: Andrew Hunt, David Thomas, Bought: True
Title: The Clean Coder: A Code of Conduct for Professional Programmers, Authors: Robert C. Martin, Bought: False
```

Figure 6: Console output of the 'WithoutCQS' sample

If you are wondering, Log4Net's ColoredConsoleAppender is configured to log to the console. You can take a look at its configuration in the sample code.

Moving towards CQS - the Query part

The read part of any CQS architecture is easier than the write part. So, it's a good idea to start there.

To get all the books from the database we will need a Query object:

```
public class GetBooksQuery : Query
{
  public bool ShowOnlyInPossession { get; set; }
    //other filters here
```

```
}
public class GetBooksQueryResult : IResult
{
  public IEnumerable<Book> Books { get; set; }
}
```

The `GetBooksQuery` has a property called `ShowOnlyInPossession`. This allows us to fetch all the books, or fetch only those that are in our possession. You can add other filters (all books published after the year 2000 for example) in the `GetBooksQuery` class. The `GetBooksQueryResult` class holds the result that will be returned from our handler, containing a list of books stored in the database.

I have not implemented an Assembler (the mapping of Entity objects to DTO objects) to simplify the solution. It is worth noting though that it is never a good idea to directly return Entity objects. You always want to use DTO (Data Transfer Objects) that correspond to a particular view for your UI.

A `QueryHandler` will handle the `GetBooksQuery`. There will always be a one-on-one mapping between your Handlers and your Queries. In this case, there will be a `GetBooksQueryHandler` which will handle the requests and fetch the books from the database.

As I've explained before, we use a Dispatcher to "route" our Query objects. The Dispatcher takes our request and passes it along. Meaning that our client application always communicates with the dispatcher, not the handlers themselves.

The dispatcher will then find and call the correct handler.

The `QueryDispatcher` interface and implementation looks like this:

```
/// <summary>
/// Dispatches a query and invokes the corresponding handler
/// </summary>
public interface IQueryDispatcher
{
  /// <summary>
  /// Dispatches a query and retrieves a query result
  /// </summary>
  /// <typeparam name="TParameter">Request to execute type</typeparam>
  /// <typeparam name="TResult">Request Result to get back type</typeparam>
  /// <param name="query">Request to execute</param>
  /// <returns>Request Result to get back</returns>
  TResult Dispatch<TParameter, TResult>(TParameter query)
    where TParameter : IQuery
    where TResult : IResult;
}

public class QueryDispatcher : IQueryDispatcher
{
  private readonly IComponentContext _Context;

  public QueryDispatcher(IComponentContext context)
  {
    _Context = context ?? throw new ArgumentNullException(nameof(context));
  }

  public TResult Dispatch<TParameter, TResult>(TParameter query)
    where TParameter : IQuery
    where TResult : IResult
  {
    //Look up the correct QueryHandler in our IoC container and invoke the retrieve
    method

    var _handler = _Context.Resolve<IQueryHandler<TParameter, TResult>>();
    return _handler.Retrieve(query);
  }
}
```

Again, no rocket science here. The dispatcher looks for the (only) matching `IQueryHandler` implementation and invokes its `Retrieve()` method.

Let's look at the `QueryHandler` Base class. This base class is not a requirement, but I use it to do general things for all QueryHandlers, like logging, authorization, etc.

```
public abstract class QueryHandler<TParameter, TResult> : IQueryHandler<TParameter,
TResult>
  where TResult : IResult, new()
  where TParameter : IQuery, new()
{
  protected readonly ILog Log;
  protected ApplicationDbContext ApplicationDbContext;

protected QueryHandler(ApplicationDbContext applicationDbContext)
{
  ApplicationDbContext = applicationDbContext;
  Log = LogManager.GetLogger(GetType().FullName);
}

public TResult Retrieve(TParameter query)
{
  var _stopWatch = new Stopwatch();
  _stopWatch.Start();

  TResult _queryResult;

  try
  {
    //do authorization and validation

    //handle the query request
    _queryResult = Handle(query);

  }
  catch (Exception _exception)
  {
```

```
      Log.ErrorFormat("Error in {0} queryHandler. Message: {1} \n Stacktrace: {2}",
      typeof(TParameter).Name, _exception.Message, _exception.StackTrace);

      //implement your Exception handling here, for example log to database, send
      mail, etc
      throw;
    }
    finally
    {
      _stopWatch.Stop();
      Log.DebugFormat("Response for query {0} served (elapsed time: {1} msec)",
      typeof(TParameter).Name, _stopWatch.ElapsedMilliseconds);
    }

    return _queryResult;
  }

  /// <summary>
  /// The actual Handle method that will be implemented in the sub class
  /// </summary>
  /// <param name="request"></param>
  /// <returns></returns>
  protected abstract TResult Handle(TParameter request);
}
```

This abstract class has some generic constraints. The `TParameter` must implement the `IQuery` interface and the `TResult` has to implement the `IResult` interface. Both generic parameters must have a public parameterless constructor.

There is a `Stopwatch` that keeps track of the execution time of the invocation of every QueryHandler. It handles errors (by logging them, but you can do other things here as well) and you could implement authorization, validation, Assemblers, etc.

Eventually our Query is handled by this line of `code _queryResult = Handle(query);`. The abstract `Handle` method will be implemented in the sub class and its return object will be returned by the super class.

So that's it! Our architecture is now ready to begin handling Queries!

Let's write our `GetBooksQueryHandler` and call it using our console application!

```
public class GetBooksQueryHandler : QueryHandler<GetBooksQuery, GetBooksQueryResult>
{
    public GetBooksQueryHandler(ApplicationDbContext applicationDbContext)
        : base(applicationDbContext)
    {
    }
    protected override GetBooksQueryResult Handle(GetBooksQuery request)
    {
        var _result = new GetBooksQueryResult();
        var _bookQuery = ApplicationDbContext.Books.AsQueryable();

        if (request.ShowOnlyInPossession)
        {
            _bookQuery = _bookQuery.Where(c => c.InMyPossession);
        }
        _result.Books = _bookQuery.ToList();

        return _result;
    }
}
```

In the program.cs file, we can now retrieve all the books from the data store. We ask our IoC container for the implementation of the QueryDispatcher. We dispatch the `GetBooksQuery` and log the returned books to the console.

```
var _queryDispatcher = container.Resolve<IQueryDispatcher>();
var _reponse = _queryDispatcher.Dispatch<GetBooksQuery, GetBooksQueryResult>(new
GetBooksQuery());

foreach (var _book in _reponse.Books)
{
  _Log.InfoFormat("Title: {0}, Authors: {1}, Bought: {2}", _book.Title, _book.
  Authors, _book.Bought);
}
```

This will (having Log4Net configured to DEBUG) output the following:

```
Resolve instance of type Cqs.SampleApp.Core.Cqs.ICommandDispatcher
Resolve instance of type Cqs.SampleApp.Core.Cqs.IQueryDispatcher
Response for query GetBooksQuery served (elapsed time: 11 msec)
Retrieving all books the CQS Way..
Title: The Pragmatic Programmer, Authors: Andrew Hunt, David Thomas, Bought: True
Title: The Clean Coder: A Code of Conduct for Professional Programmers, Authors: Robert C. Martin, Bought: False
```

Figure 7: Console output of the GetBooksQuery

Ok, that's all for the Query part!

Any type of architecture will result in more code, but I hope you realize that integrating the Command Query Separation pattern does a good job on keepings things simple and well organized.

The QueryHandler super class allows for a lot of tweaking, all in one place. If you don't like the use of super classes, you can even use decorators to introduce new behaviour. But that's too much ground to cover in one article.

Let's move on to the Command part!

Moving towards CQS - the Command part

As said before, the Command part of Command Query Separation is a bit more complex. The main difference is that, because you are doing mutations on your data store, you'll need more checks. Is the data valid? Is the data consistent? etc. But the general idea remains the same.

It starts, just like the Query part, with a Dispatcher. This class will make sure your Command is handled by the correct CommandHandler.

```
public class CommandDispatcher : ICommandDispatcher
{
  private readonly IComponentContext _Context;

  public CommandDispatcher(IComponentContext context)
  {
    _Context = context ?? throw new ArgumentNullException(nameof(context));
  }

  public TResult Dispatch<TParameter, TResult>(TParameter command) where TParameter
: ICommand where TResult : IResult
  {
    var _handler = _Context.Resolve<ICommandHandler<TParameter, TResult>>();
    return _handler.Handle(command);
  }
}
```

Just like in the Query part, we look for a CommandHandler that matches our Command and CommandResult.

The `SaveBookCommand` and its result look like this:

```
public class SaveBookCommand : Command
{
  public Book Book { get; set; }
}

public class SaveBookCommandResult : IResult
{
}
```

In the sample application, the CommandHandler base class looks exactly the same as the QueryHandler base class. The only real difference is the generic constraints. While the QueryHandler will only take objects that implement the interface `IQuery`, the CommandHandler requires objects that implement `ICommand`.

As said earlier, this super class is kept as simple as possible. It just passes the Command to its sub class, just like the QueryHandler.

A more "production worthy" CommandHandler would have a constructor like this:

```
protected CommandHandler(ICommandValidator<TRequest> validator, IMapperEngine
mapperEngine, IUnitOfWork unitOfWork)
{
  MapperEngine = mapperEngine;
  UnitOfWork = unitOfWork;
  _Validator = validator;

 _Log = LogManager.GetLogger(GetType().FullName);
}
```

The `Validator` and `MapperEngine` classes are modules that perform data validation and transform Entity Objects to Data Transfer Objects. The `UnitOfWork` handles transactions, scopes and different contexts.

But the demo code should be enough to allow you to understand the core concepts of CQS.

Let's take a look at a CommandHandler implementation:

```
public class SaveBookCommandHandler : CommandHandler<SaveBookCommand,
SaveBookCommandResult>
{
  public SaveBookCommandHandler(ApplicationDbContext context) : base(context) {
  }

  protected override SaveBookCommandResult DoHandle(SaveBookCommand request) {
    var _response = new SaveBookCommandResult();

    //get the book
    ApplicationDbContext.Books.Attach(request.Book);

    //add or update the book entity
    ApplicationDbContext.Entry(request.Book).State =
      request.Book.Id == Constants.NewId ? EntityState.Added : EntityState.Modified;

      //persist changes to the datastore
      ApplicationDbContext.SaveChanges();

      return _response;
  }
}
```

This should be pretty straightforward. The code checks if the book that is passed along with the Command is a new book or not, and adds or updates it accordingly.

We can now update the Books library!

```
var _commandDispatcher = container.Resolve<ICommandDispatcher>();

//edit first book
var _bookToEdit = _reponse.Books.First();
_bookToEdit.Bought = !_bookToEdit.Bought;
_commandDispatcher.Dispatch<SaveBookCommand, SaveBookCommandResult>(new
SaveBookCommand()
{
  Book = _bookToEdit
});

//add new book
_commandDispatcher.Dispatch<SaveBookCommand, SaveBookCommandResult>(new
SaveBookCommand()
{
  Book = new Book()
  {
    Title = "C# in Depth",
    Author = "Jon Skeet",
    Bought = false,
    DatePublished = new DateTime(2013, 07, 01)
  }
});
```

That's it! The output now shows the correct number of books:

```
Resolve instance of type Cqs.SampleApp.Core.Cqs.ICommandDispatcher
Response for query SaveBookCommand served (elapsed time: 140 msec)
Response for query SaveBookCommand served (elapsed time: 49 msec)
Response for query GetBooksQuery served (elapsed time: 6 msec)
Title: The Pragmatic Programmer, Authors: Andrew Hunt, David Thomas, InMyPossession: False
Title: The Clean Coder: A Code of Conduct for Professional Programmers, Authors: Robert C. Martin, InMyPossession: False
Title: C# in Depth, Authors: Jon Skeet, InMyPossession: False
```

Figure 8: Console output after inserting and updating a book

Conclusion

CQS, or Command Query Separation, is a very subtle, simple yet very adaptable and powerful pattern that I have been using as a baseline in my architectures for years. It has proven its worth time and time again.

Your project gets structured in a very clear way. The code gets more readable and more trustworthy as you get the assurance that unexpected side effects cannot arise. You can encapsulate a lot of complexity in the Dispatchers and/or BaseHandlers, allowing developers to focus on what they do best: Writing the application!

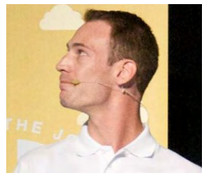
I hope you enjoyed this read and make sure to check out the **sample code on my** github page.

• • • • • • • •

Microsoft® MVP Most Valuable Professional

Tim Sommer

Author



Tim Sommer lives in the beautiful city of Antwerp, Belgium. He is passionate about computers and programming for as long as he can remember. He's a speaker, teacher and entrepreneur. He is also a Windows Insider MVP. But most of all, he's a developer, an architect, a technical specialist and a coach; with 8+ years of professional experience in the .NET framework.

Thanks to Yacoub Massad *for reviewing this article.*

Rahul Sahasrabuddhe

NON-FUNCTIONAL REQUIREMENTS:

THE UNSUNG HEROES

Non-Functional requirements or NFRs are key design aspects of any product or app. Yet, they are often forgotten during the process of design, implementation, testing and, deployment.

In this article, we'll understand various NFRs and how they play a vital role in making a product or an app robust.

NON-FUNCTIONAL REQUIREMENTS - INTRODUCTION

Consider the following scenarios:

1. You have spent around 3-4 months building a cool responsive web app using the best JavaScript framework. When you deploy it on Azure and a bunch of users start using it simultaneously, the app times out multiple times for various functionalities or pages.

2. You have built a mobile app on both iOS and Android as a fixed price project for a customer. But during UAT, you find out that both iOS and Android have gone through version updates/upgrades and now the mobile app behaves erratically for certain scenarios. Since it is a fixed price project, the customer is not ready to pay for changes (as you call them) or fixes (as termed by the customer).

3. You've built a middleware based on the micro-services principles using ASP.NET Web APIs for an Enterprise customer (let's say a big bank in US). Now the security team of that Enterprise has found various issues in the code-base since it does not comply with OWASP top 10 security risks or vulnerabilities.

And the list can go on with multiple such real-life scenarios where you as a developer or as an architect or as a scrum master have faced that moment of truth - "Wish, I could have thought of this before!".

Déjà vu?

All the examples above highlight the fact that the NFRs are the most critical yet generally ignored aspects of product/app development.

Functional Requirement (FR) vs Non-Functional Requirement (NFR)

NFRs are the requirements defined to cover the operational aspects of a product or an app while Functional requirement (FR) focus on the behavioral aspects.

Let's say you are going to build a simple shopping cart app:

1. The app should display various products matching the search criteria is a functional requirement.

2. But, the app should display the products within 2 seconds from the time the user hits the "Search" button is a non-functional requirement.

Here are some other key differences between FRs and NFRs:

1. NFRs describe the overall experience of using a product or app while FRs obviously are focused on a specific set of functionalities.

2. They are typically stated as implicit requirements (i.e. always expected than stated).

3. FRs can be captured at a user story, use case or a function point level. NFRs usually exist across the product (and also for the lifetime of the product). For example, Usability or UX is a product-wide or app-wide requirement and should be considered even in product updates/upgrades/enhancements.

4. NFRs are traced across various modules of a product/app like front-end, middleware or databases, etc.

5. NFRs are often classified as a part of software quality assurance (SQA) process and rightly so.

In short, the FRs define "what" and the NFRs define "how".

From an end-user perspective, both are important and they should work hand-in-hand.

THE CURIOUS CASE OF NFRS

Now let's look at NFRs from different perspectives.

SDLC & NFRs

9 out of 10 projects that get executed today are Agile or Scrum based.

How many of them actually follow Agile processes? Or are just called as Agile for the brand value is a totally different topic. We'll keep that for a separate discussion some other time.

Irrespective of the process you follow for developing your product or app, NFRs have to be considered from Day 1. However, the kind of process you follow will drive the way you handle NFRs.

Let's see how:

Agile/Iterative Model

In Agile development, you work in a sprint-model with theme > epic > story > task breakdown. So, when you are breaking down the functionalities, you need to also ensure that you plan for implementation of specific NFRs as a part of sprint-model.

UX/usability would come up first in terms of priority of NFRs; however, scalability, performance, security, browser compatibility will follow. For any epic/story/task, be mindful of the NFRs and come up with task breakdown and estimates.

Here are various possible ways to handle this aspect in agile development:

1. Create the acceptance criteria for story points that covers NFRs. So, for a specific story, you can add an acceptance criterion that says the *app should function well for 50 concurrent users*. So when the developer has implemented specific stories, he/she needs to also ensure that the story passes this acceptance criteria.

2. Another way is to have the DoD (definition of done) cater to specific aspects of NFR. For example, once you define UX guidelines for a project, the DoD should mandate that UX guidelines are followed in developing any new page or UI screen. This needs to be mandated for developers and also testers.

3. A more suitable and process-oriented way could be creating user stories for NFRs as well. So you can follow the typical "As an X, I want Y, so that Z happens" format but catering to NFRs. So, a story catering to high availability would be – as an end- user of the site, I want the site to be accessible 99.9% of the times so I could perform various operations seamlessly without down-time.

These are various ways of incorporating NFRs into implementation process from early on. The model or approach you would like to follow can be any of these as long as it works for the team and, of course, NFRs are a part of the delivery process.

Waterfall Model

Since the waterfall model itself forces you to do a thorough analysis of requirements followed by detailed design including logical/physical architecture design, chances are high that the NFRs are analyzed and detailed out well enough in analysis and design phase.

However, this needs to be validated by taking a sign off from the customer on things like response times, performance benchmarks etc. Else you are again falling into a trap wherein the implicit requirements or expectations from the customer (or from customers of customer or end-users) would differ greatly from what you have coded for.

Team Roles & NFRs

It is evident that NFRs are important and critical for any and every member of the team – be it scrum master, developer, QA, release manager or a support engineer. In each role however, the scope of understanding would differ:

1. For a Product Owner it is critical to have the overall picture clear in terms of goals or vision for building a product or software. After this, with help from software architect, the NFRs need to be broken down into stories or acceptance criteria (as explained earlier) and the sprints have to be executed accordingly.

2. For a Scrum Master, it is critical to understand the cost aspects of not handling NFRs in due course of the development process. What is also important for Scrum Master is to know as to how to incrementally get the NFRs implemented and delivered through the agile/sprint-based process.

3. For a business analyst (or a product owner), the end-user perspective of NFRs is more important. BA should be able to elicit the NFRs clearly so that the dev team can understand. BA/PO should be able to understand the implicit requirements by talking to end-users.

4. For a developer or QA engineer, it is important to understand the impact and relevance of each NFR on the specific functional requirement that he or she is implementing (i.e. either writing the code or test cases or automating them). And this is irrespective of years of experience the person has. The habit of thinking about NFRs needs to be inculcated from day one of their careers.

With NFR affecting so many variables in the equation, how do you ensure that you've covered them all?

One solution is to use requirement traceability matrix to track NFRs across various SDLC phases and across various product releases. So for a specific NFR – say security – you would be able to trace it well across

various stages of product development.

MEET THE SUPPORTING CAST

By now you must have realized that NFRs are essentially those important cogs in the wheel that appear to be trivial but have a bigger impact if ignored, just like a supporting actor in the movie.

You always focus on the key actors but their performance is complemented well by supporting actors. If they don't play their part well, the performance of main actors may not have the right impact.

NFRs can be your best supporting cast or best villainous performers depending on how they are envisioned, implemented and managed.

We'll be focusing on the following supporting actors:

- Scalability
- Performance
- High Availability
- Security
- Usability
- Internationalization and Localization
- Browser Compatibility
- Compliance
- Accessibility

Let's understand them one by one in depth.

Scalability

Scalability is the ability of a product or an app to scale up (or down as well) to fulfil the changes in app usage demands.

When you design a product or an app, have some idea upfront about the number of users that would use the product or the app. If you don't design for that, then the application would not perform well.

Not all the users (i.e. total number of users) use the application at a time. Hence you would also need to know how many users will use the product/app at a given time, also called as concurrent users. Concurrent users are generally 20 to 30% percent of total user base. However, it can go up to 100% for specific events – for example, your ecommerce product has launched a Christmas sale. Or your football match portal has gone online for selling tickets for a match and so on.

While total and concurrent users are the right metrics to track, you should not forget the time-factor as well. To handle such events, your app should be able to scale up/down based on needs.

Usually there are two kinds of scaling requirements:
Vertical (i.e. scaling up): When you are adding more CPUs or RAM to existing infrastructure, then you are essentially scaling up the infrastructure or vertically scaling it.

Horizontal (i.e. scaling out): When you are adding more servers to the same infrastructure, then you are scaling out the same infrastructure.

Following is a pictorial representation of both approaches:

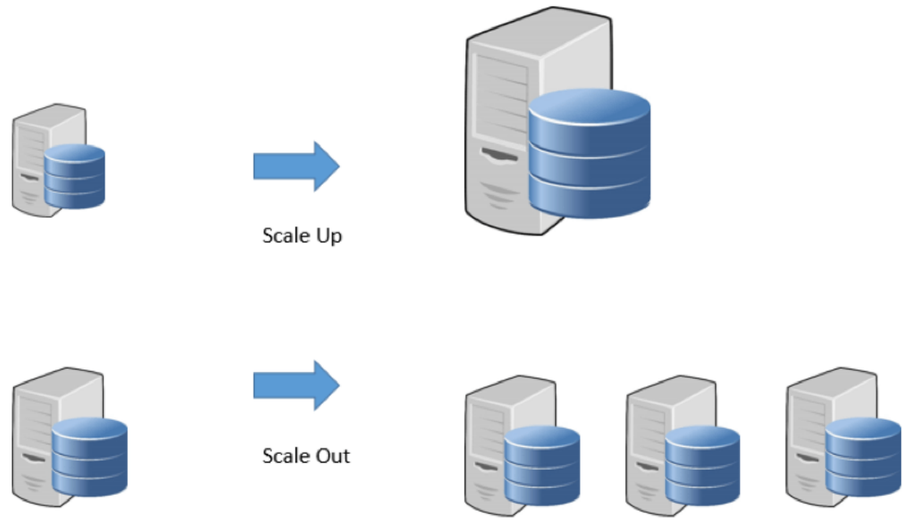


Figure 1 : Scale up vs Scale out

So, enhancing the capability of same node is scaling up or vertical scaling, whereas, adding a new node altogether is scaling out or horizontal scaling.

Most cloud providers like Azure or AWS "auto-scale" your infrastructure and have required interface to allow you to configure these things. Here are some best practices for Azure auto-scaling.

For an on-premise deployment though, you have to ensure that you understand these nuances well enough and are able to handle the scalability well. In this case, the capacity planning activity is very important. In this activity, you will need to understand the overall footprint of the product or app that you are building and then plan for hardware resources like servers, memory, CPU, database servers, web servers etc.

Software Scalability

An often-ignored aspect in scaling is the role of Software Architecture.

There are umpteen instances wherein the software scaling is not thought through during the design phase. You may scale out or scale up but if your cool data grid on a single page cannot load up in first place because it loaded data for 50,000 rows during page-load, hardware scaling will not help.

In essence, while you have the hardware to rely on for scaling, it is important that the software is written well in such a way that it produces an optimal performance in the first place.

Some key aspects to consider to make this happen are:

1. Software scaling usually is a function of the underlying technology/framework you are using. If that can scale, your app can scale. If your framework is good at doing multiple things at once (not only multi-threading), then you are good. Most modern-day technologies fulfill this aspect. However, if you are using the features of technology or framework in the wrong way, it may not work.

2. If you are using any legacy component as DLL or an API, then you need to ensure that that does not become a bottleneck. Think of some critical C++ library that your web app is using. The library also needs to scale well in order for your app to perform well.

3. If you are using a load balancer in the physical topology, then you would be able to also gracefully handle the architectural scaling problems.

4. Having right data caching policies at the web server, database level and also at data level could help in handling the scaling aspects well. However, this needs to be analyzed on a case-by-case basis.

5. In case of a database, use and define Views where needed. Also use proper indexing to avoid specific issues.

6. Session management needs to be used effectively to ensure that application performance is optimal

Even after taking some preemptive steps you would have to also take care of some corrective steps when you encounter issues after deployment.

Performance

Performance of a product or an app defines how a product/app is performing or behaving as compared to its expected behavior.

Following are the key metrics tracked in the context of performance:

1. **Response times:** This is a classic metric used for analyzing the performance of a product or an app. It defines how fast or slow your product or app is when a user tries to use it.

   An associated parameter is the latency in responses. So, some amount of latency (i.e. delay) might be acceptable to users but it cannot go beyond a threshold. That threshold is *response time*.

   You would typically define the response time per screen or for critical screens. Or you can define the requirement as 80% of the screens would have a response time of less than 3 seconds.

   Latency is attributed to resources used while providing response to user (hardware, communication protocol, data center etc.), whereas response time includes latency and the processing time (i.e. to build a response for a specific page).

2. **Throughput**: This one typically refers to the number of transactions that your app or product can handle at a time. It could be specific number of user requests and/or database transactions or API calls and so on. You would define the throughput goals as a part of requirement analysis phase and would have performance/load testing phase to validate the product or app's performance.

Strategy for Performance Analysis

> *In god we trust, all others bring date*
>
> *– W. Edwards Deming*

This quote exemplifies the strategy to be followed for performance analysis and tuning of your product or app. As long as you have data that depicts the issues in the performance of software, you can analyze and fix it. And as long as you have data that proves that your product or app is performing well on given KPIs, then you can be rest assured that all is well.

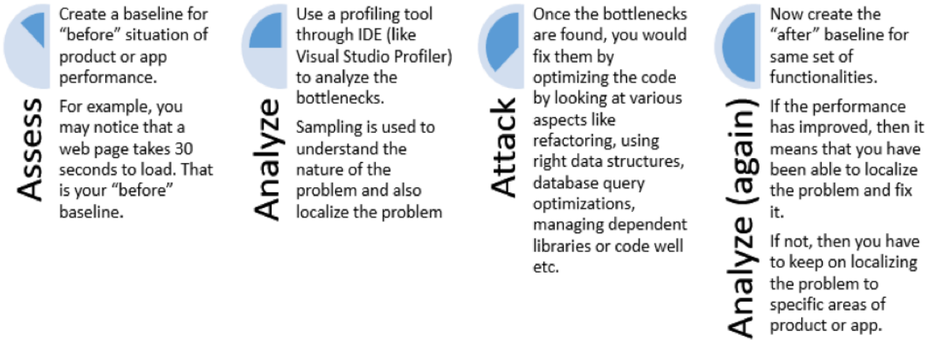
The performance aspect of a product or an app is handled in two ways:

1. You need to first ensure that your architecture or design is fool-proof enough to handle the performance metrics for which it has been designed. For this, the analysis phase needs to focus well on the desired performance benchmarks for response time, throughput etc.

2. Once done, in the system testing phase, these aspects are required to be verified and validated through test automation tools like JMeter.

However, it does not end here. Many times, when the product or apps go live or the functionalities get changed/added, the performance of the product or app may deteriorate.

In that case, you need to use the following approach:



After running this process multiple times, there are still situations when your product or app exhibits a random behavior and it is difficult to pin-point the cause of the issue. In that case, keep on using the trial-and-error methodologies to single out the problem.

However, there is no secret sauce or a well-defined formula for fixing performance problems. It's like an adventure park ride. You have to experience it to enjoy the thrill.

Let's look at a few tools used for performance testing:

Tool	Pros	Cons
JMeter	1) It is free! It is open source as well. Highly customizable and easy to use 2) Platform independent – works well on Windows and Linux. 3) Excellent configurable reporting capabilities to suit various needs. 4) Has great documentation and community support since it is widely used.	1) It needs a good infrastructure to run as it is a memory-hungry app. 2) More suitable for API testing (although UI testing can also be done).
LoadRunner	1) Versatile and can be used to perform performance, load, database and web app testing. 2) Easier to use as compared to JMeter	1) It is an expensive tool.
Gatling	1) Open source. Uses Scala 2) Easier for developers to use it since it has code-like scripting	1) Not a wide variety of protocols supported
VSTS Load Test	1) This could be a good option if your entire development tool-set is Visual Studio and TFS-based.	1) VSTS Load Test comes with Visual Studio Enterprise edition. 2) It runs on Windows only.

There are quite a few automation tools available and this list can be big but I have provided only a sampling of some popular tools. Please note that there are various UI/automation testing tools like Selenium that I have not mentioned here since we are more focused on performance testing aspects.

High Availability (HA)

If you need your web app to be available for serving users at all times (or almost at all times), then you need to design it for high availability. It is often stated in % terms like HA for a web app is 99%. What this means is that in a year, the system will be guaranteed to be available for 361.35 days (i.e. 99% of the year).

This is an indicative architecture diagram for a web app deployed in cloud (AWS/Azure or any other cloud provider) that has the following components:

1. Load balancer for balancing the varied load that the site would get.

2. App server that would serve user requests. It would host the UI and business logic layers.

3. Database servers with replication.

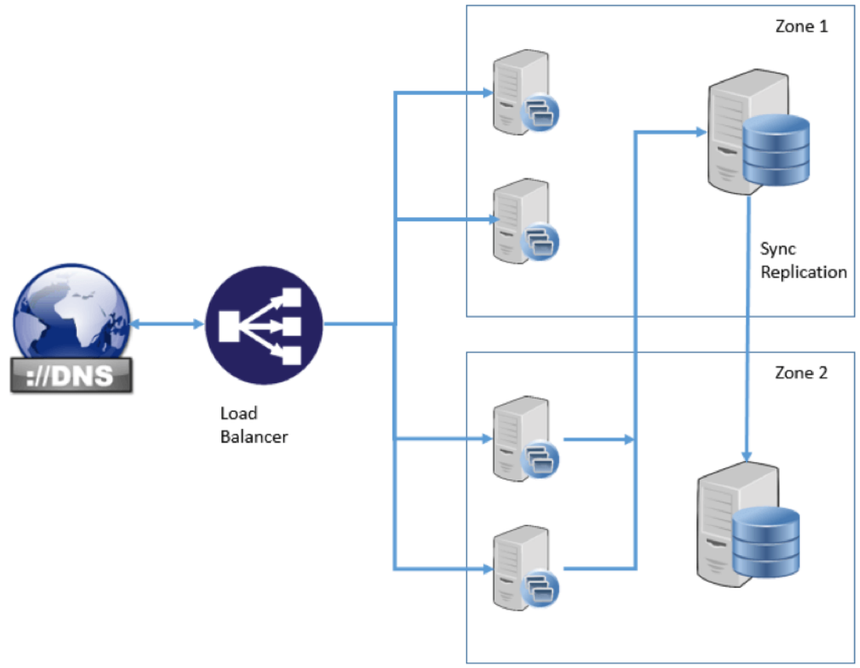


Figure 2 : 3-Tier HA architecture

Following are the key design principles followed in the HR architecture shown in Figure 2:

1. **Design for failure:** The architecture has redundancy built into it so that the failure of a server/resource does not impact operations. App servers and DB servers take care of that aspect. The load balancer itself can be a single point of failure. But this is already handled by the respective cloud providing solutions (e.g. EC2 based elastic load balancer in AWS).

2. **Availability zones:** Azure/AWS provide HA through the use of availability zones. Azure availability zones are unique physical locations providing redundancy within specific Azure regions

3. **Auto-scaling:** The cloud provider already has ability to auto-scale the infrastructure, and it is used while deploying the infrastructure.

In a nutshell, HA is a critical NFR that needs to be thought through during the design and analysis phase. It certainly cannot be an after-thought.

Security

Security has become an NFR of paramount importance of late due to the amount of data collected by various products/apps and the vulnerabilities that are being exploited by hackers for theft of data.

OWASP (Open Web Application Security Project) is an online community that focuses on improving the security aspects of web apps. It publishes a guidance every year about top 10 security vulnerabilities and how to handle them in your apps.

The following table summarizes top 10 OSWAP security vulnerability areas for 2017.

Security Vulnerability	Description
Injection	These include SQL, NoSQL, and LDAP injection by exploiting the loopholes in the way queries are coded in the applications. A most common example of such an attack is that the attacker would inject an SQL script in login/password fields and can get information out from the site.
Broken Authentication	A hacker would use a session id or token to hack into a web application and use these stolen credentials to use the application. It is required to ensure that session management is effectively handled to avoid such attacks.
Sensitive Data Exposure	If your app is not using secured protocol like HTTPS, then the hacker can steal information like credit card details. By just using HTTPS, such things can be avoided.
XML eXternal Entities (XXE)	This is relevant for applications that parse/use XMLs - API layers or SOAP-based services being such examples. If your application is not handling XML parsing well enough, then a hacker can use the loopholes to get sensitive data.
Broken Access Control	Like broken authentication, if your application has not effectively implemented access control, then the user can use session ID for one profile of the user to access data for another user. An example is a user getting access to admin-privileged information through this route and collecting information he/she is not privy to.
Security Misconfiguration	It is not only important to make your app secure but also have the frameworks, and underlying infrastructure to comply with same security standards. If you don't keep your system up to date with latest security patches, then having your app comply with all security aspects may not be sufficient.
Cross-Site Scripting (XSS)	These days almost all web apps have some or the other JS based UI. The attacker can inject JS into web pages of public facing websites or apps and can get access to sensitive information.
Insecure Deserialization	We usually serialize/deserialize objects to manage the state of various data elements in the workflow of a web app. It is required to validate the data during deserialization in order to avoid any malicious tampering done with the serialized data.
Using Components with Known Vulnerabilities	We often use various open source libraries and components without really worrying about what is "inside" them. This can lead to serious security threats to your app. Hence it is important to validate the "source" of open source libraries/components to be trustworthy enough to be used in production-grade apps.
Insufficient Logging and Monitoring	Although logging and monitoring does not directly affect the security, it is the method or level of detail in which logging and monitoring is done could have an impact on security. Chances are high that a good logging and monitoring would detect the breach faster or quicker than a system/app that does not effectively use logging and monitoring.

Certain aspects of security assessment can also be carried out using various threat modelling tools available. Microsoft has a threat modelling tool available that can be used for performing a detailed analysis of possible threats during design phase itself. Plus there are many more such tools available as well.

Usability

> *A user interface is like a joke. If you have to explain it, it is not that good.*

It is clearly evident that usability of UX is vital for your product/app to acceptable to users. Here is an in-depth article for you to understand the Usability or UX aspect well enough.

Internationalization and Localization

This is yet another NFR that surfaces fairly later in the game and would impact the overall execution in terms of additional testing efforts. Hence it is prudent to consider this aspect during the design phase.

Before we move on, you need to understand the basic difference between internationalization and localization. It is as follows:

1. Internationalization (often stated as i18n) is a process by which you enable your application to be used in various local languages. You follow various processes like using resource bundle etc. so that your app can be used in languages like German, French, Hindi, Chinese and so on.

2. Localization (stated as l10n) is making your product or app adaptable to a locale or a local language.

3. So i18n enables your app for l10n. Got it?

If your product or app will support various spoken languages like German, French, Chinese etc., then your UI should support it. Have the UI strings or labels on your screen to appear in these languages. What this means is:

1. Have the right translation resource bundles in your code projects.

2. Factor in additional testing efforts for the entire UI for the languages supported. Remember that the entire app needs to be tested for multiple languages.

3. Ensure that the UI/screens look consistent in terms of white spaces. For example, the German translation of a word occupies longer space as compared to English one.

4. For Arabic language, the screen should display from the right to left for all strings. So your app will have a different look altogether for Arabic UI.

5. Be mindful of numbering formats followed. In French, comma is used as a decimal point. A French-speaking user would need the commas for decimal points whereas an English-speaking user would need dots!

6. UI text or labels are fine but what happens to data on the UI? If you have entered a value in French with a comma as a decimal point – say 9,92 , what do you store it as in the database – 9.92 or 9.92? And how would you display it in a report that gets generated later? All these questions have to be asked and answered during analysis phase.

Browser Compatibility

Remember that friend of yours who likes Firefox over Chrome and you've fought with him about how Chrome is better than Firefox?

Aside from a nice coffee-table debate, browser compatibility becomes a key aspect in product or app development. Have you not seen banners like "best viewed in Chrome XXX"?

Well, that explains it!

It is not only about the browser you use. It is also about the form factor.

If you have designed a web app or a hybrid app that a user would use on the web or on a mobile/ smartphone, then you need to ensure that it renders well on both form factors. If your UI is responsive i.e. it adapts to the various form factors, then you have won half the battle. That is why it is critical to get this NFR sorted out in design (that too UI/UX design) phase itself.

Testing for Browser Compatibility

With so many browsers and various form factors to handle, are you worried how can you test for browser compatibility?

Well, there is a tool for that! Following is a quick summary of such tools available:

Chrome Developer Tools: I'm sure you have already used this. If you go to Chrome > Developer Mode, you can see the DotNetCurry (DNC) website:

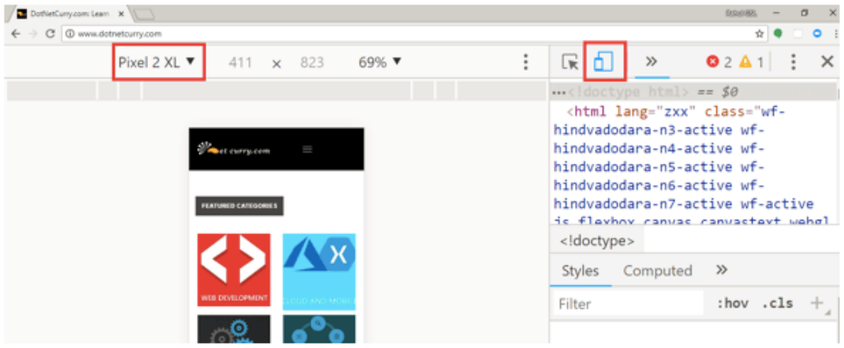


Figure 3: Various Device Views in Chrome Developer Tools.

The options shown in the highlighted boxes in Figure 3 will allow you toggle the device/web view, and various device views. This way, you can quickly check if your web app would look stunning or not so stunning for different form factors.

If you want thoroughly test your app end-to-end for browser compatiblity, then you have various tools that can be used.

Here's a quick summary of some tools:

Tool	Features
BrowserShots [1]	It is a simple yet powerful tool that takes a screenshot of how your app would look like in various browsers. The list of browsers supported is exhaustive, but it does not support IE (yes, that's true!). It has a range of options to experiment with validating things like correct functioning of JS, Flash etc. for your web app.
BrowserStack [2]	This is a paid tool that provides following features: - Supports a variety of browsers and devices of various OSes. - Used by biggies like Microsoft, Twitter, Airbnb and so on - Once you sign up/in, you get access to VMs that run browser versions and you can use them for testing. -
Browserling [3]	This is very similar to BrowserStack in terms of features and offerings. It provides real devices and browsers to test. It also has free/paid plans.

Compliance

Since there is so much user data collected and processed by various products or apps we use, the privacy and control of data by the creators of apps/products has become very critical. And that has led to a very important NFR surfacing up–the compliance requirements.

Why would this matter to you as a developer?

Well think of a situation where your product or app has been GA'ed (a.k.a. launched or General Availability) and now a specific customer wants to ensure that there are enough audit trails or audit logs to comply with regulatory requirements of a government body.

This means that you have to go to all those places in code where you are changing data (almost everywhere!) and add logs or trails.

It does not end here; you have to have a mechanism to generate reports as well. If you have not catered for any of this, this implicit new requirement is touching almost all parts of your product or app. How is that?

1. BrowserShots - http://browsershots.org/
2. BrowserStack - https://www.browserstack.com/
3. Browserling - https://www.browserling.com/

The following table summarizes such compliance standards you need to know of (the list is not exhaustive):

Compliance Standard	Areas Covered	Compliance Impact
HIPPA (Health Insurance Portability & Accountability Act)	• A must-have in any product/app that deals with healthcare industry. • Focuses on ensuring the safety of medical records data of patients (usually called as PHI – personal health information) during its handling by medical systems, doctors and other medical professionals. • Key aspects are maintaining the privacy and security of the information shared by patients with relevant entities like hospitals/doctors, etc.	If you are developing an app that deals with patient data, then you need to ensure that it complies with following HIPAA norms: • Privacy of data • Security of data • Enforcement of HIPAA norms • Implementation of system for breach notification
GDPR (General Data Protection Regulation)	This is a law created in 2016 by European Union (EU) that governs data protection and privacy for all individuals in EU. If your app is dealing with customers/users in European Union (EU), then you need to comply with GDPR norms, that cover all the following - Personal data including basic identity information, - Health data including biometric information - Web data like cookies, location/IP address etc.	In order to comply with GDPR, you need to ensure that appropriate checks and balances are put in place in your application. Following are the key aspects: • Cookie consent from user for storing and using their personal data • App should allow the users to delete their personal data. • Database should be encrypting the user data stored
HL7 (Health Level 7)	This standard governs the way data is transferred between different systems that handle medical data. It works in conjunction with HIPAA.	If your app needs to comply with HL7, then you need to use the right message formats as recommended by HL7. This* is a good library to use if you need your web app to comply with HL7 standard.

Accessibility

If you are building a consumer app, then you need to be mindful of the possibility of your app being used by someone who is hearing impaired or is blind or has some physical disability.

And this aspect needs to be analyzed during the requirement phase itself. If you intend to build a web application or any application for Federal agencies in USA, then you need to comply with Section 508 or Section 508.gov that covers the accessibility aspect well enough.

There are similar such acts in other countries (viz. in Australia Disability Discrimination Act) wherein the government mandates to have public facing apps to comply with accessibility.

The W3C consortium has Web Content Accessibility Guidelines (WCAG) detailing out various aspects of accessibility.

Here is a tool that you can use to ensure that your ASP.NET app is complying with accessibility requirements.

Conclusion:

Have you seen the batman movie, *The Dark Knight*?

It is one of top ten highest grossing movies as per this link. Ok, why am I bringing it up here? Just bear with me for a while.

Assuming that you have seen the movie (and if you have not! go see it!), do you think that the joker a.k.a. Heath Ledger is the supporting actor or the main actor?

Well, he won the Oscar for the best supporting actor that year! However you couldn't think of the movie without him. That would be an example of how the supporting actor has stolen the show! So you never know when your supporting cast could take the center stage and make or break the whole thing.

NFRs are like that.

If you pay enough attention to them, they'll win the show for you.

So next time when someone argues with you about the importance of NFRs, you can, to quote the joker, "Smile – because it confuses people!". Because only you would know how important NFRs are!

Isn't it?

Rahul Sahasrabuddhe
Author

Rahul Sahasrabuddhe has been working on Microsoft Technologies since last 17 years and leads Microsoft Technology Practice at a leading software company in Pune. He has been instrumental in setting up competencies around Azure, SharePoint and various other Microsoft Technologies. Being an avid reader, he likes to keep himself abreast with cutting edge technology changes & advances.

Thanks to Subodh Sohoni *for reviewing this article.*

* https://github.com/ewoutkramer/fhir-net-api

Damir Arh

THE EVOLUTION OF C#

Since its original release in 2002, C# has been regularly updated with new features. Today, we will look at the most important new features of each major language version and explore how the C# code we have been writing, has evolved through years.

Although it's already over 15 years since the original release of C#, the language doesn't feel that old. One reason being it has been updated on a regular basis.

Every two or three years, a new version was released with additional language features. Since the release of C# 7.0 in the beginning of 2017, the cadence has further increased with minor language versions. Within a year's time, three new minor language versions were released (C# 7.1, 7.2 and 7.3).

Read more at C# 7.1, 7.2 and 7.3 - New Features (Updated).

If we were to look at the code written for C# 1.0 in 2002, it would look much different from the code we write today.

Most of the differences result from using language constructs which didn't exist back then. However, along with language development, new classes were also added to the .NET framework which take advantage of the new language features. All of this makes the C# of today much more expressive and terser.

Let's take a trip into history with an overview of the major versions of C#.

For each version, we will inspect the most important changes and compare the code that could be written after its release, to the one that had to be written before. By the time we reach C# 1.0, we will hardly be able to recognize the code as C#.

C# 7.0

At the time of writing, the latest major language version is 7.0. With its release in 2017, it's still recent, therefore its new features aren't used often.

Most of us are still very used to writing C# code without the advantages it brings.

The main theme of C# 7.0 was **pattern matching** which added support for checking types in `switch` statements:

```
switch (weapon){
  case Sword sword when sword.Durability > 0:
    enemy.Health -= sword.Damage;
    sword.Durability--;
    break;
  case Bow bow when bow.Arrows > 0:
    enemy.Health -= bow.Damage;
    bow.Arrows--;
    break;
}
```

There are multiple new language features used in the above compact piece of code:

- The `case` statements check for the type of the value in the `weapon` variable.
- In the same statement, I declare a new variable of the matching type which can be used in the corresponding block of code.
- The last part of the statement after the `when` keyword specifies an additional condition to further restrict code execution.

Additionally, the `is` operator was extended with pattern matching support, so it can now be used to declare a new variable similar to `case` statements:

```
if (weapon is Sword sword)
{
  // code with new sword variable in scope
}
```

In earlier versions of the language without all these feature, the equivalent block of code would be much longer.

```
if (weapon is Sword)
{
  var sword = weapon as Sword;
  if (sword.Durability > 0)
  {
    enemy.Health -= sword.Damage;
    sword.Durability--;
  }
}
else if (weapon is Bow)
{
  var bow = weapon as Bow;
  if (bow.Arrows > 0)
  {
    enemy.Health -= bow.Damage;
    bow.Arrows--;
  }
}
```

Several other minor features were added in C# 7.0 as well. We will mention only two of them:

(i) Out variables allow declaration of variables at the place where they are first used as `out` arguments of a method.

```
if (dictionary.TryGetValue(key, out var value))
{
  return value;
}
else
{
  return null;
}
```

Before this feature was added, we had to declare the `value` variable in advance:

(ii) Tuples can be used to group multiple variables into a single value on-the-fly as needed, e.g. for return values of methods:

```
public (int weight, int count) Stocktake(IEnumerable<IWeapon> weapons)
{
  return (weapons.Sum(weapon => weapon.Weight), weapons.Count());
}
```

Without them, we had to declare a new type to do that even if we only needed it in a single place:

```
public Inventory Stocktake(IEnumerable<IWeapon> weapons)
{
  return new Inventory
  {
    Weight = weapons.Sum(weapon => weapon.Weight),
    Count = weapons.Count()
  };
}
```

To learn more about the new features of C# 7.0, check my article C# 7 – What's New from a previous edition of the DNC Magazine. To learn more about the minor editions of C# 7, check my article C# 7.1, 7.2 and 7.3 – New Features (Updated) in the DNC Magazine

C# 6.0

C# 6.0 was released in 2015. It coincided with the full rewrite of the compiler, codenamed Roslyn. An important part of this release was the compiler services which have since then become widely used in Visual Studio and other editors:

- Visual Studio 2015 and 2017 are using it for syntax highlighting, code navigation, refactoring and other code editing features.
- Many other editors, such as Visual Studio Code, Sublime Text, Emacs and others provide similar functionalities with the help of OmniSharp, a standalone set of tooling for C# designed to be integrated in code editors.
- Many third-party static code analyzers use the language services as their basis. These can be used inside Visual Studio, but also in the build process.

To learn more about Roslyn and the compiler services, check my article .NET Compiler Platform (a.k.a. Roslyn) – An Overview in the DotNetCurry (DNC) Magazine

There were only a few changes to the language. They were mostly syntactic sugar, but many of them are still useful enough to be commonly used today:

- Dictionary initializer can be used to set the initial value for a dictionary:

```
var dictionary = new Dictionary<int, string>
{
  [1] = “One”,
  [2] = “Two”,
  [3] = “Three”,
  [4] = “Four”,
  [5] = “Five”
};
```

Without it, the collection initializer had to be used instead:

```
var dictionary = new Dictionary<int, string>()
{
  { 1, “One” },
  { 2, “Two” },
  { 3, “Three” },
```

```
  { 4, “Four” },
  { 5, “Five” }
}
```

- The **nameof** operator returns the name of a symbol:

```
public void Method(string input)
{
  if (input == null)
  {
    throw new ArgumentNullException(nameof(input));
  }
  // method implementation
}
```

It’s great for avoiding the use of strings in code which can easily go out of sync when symbols are renamed:

```
public void Method(string input)
{
  if (input == null)
  {
    throw new ArgumentNullException(“input”);
  }
  // method implementation
}
```

- **Null conditional operator** reduces the ceremony around checking for `null` values:

```
var length = input?.Length ?? 0;
```

Not only is there more code required to achieve the same without I, it’s much more likely that we will forget to add such a check altogether:

```
int length;
if (input == null)
{
  length =0;
}
else
{
  length = input.Length;
}
```

- **Static import** allows direct invoking of static methods:

```
using static System.Math;

var sqrt = Sqrt(input);
```

Before it was introduced, it was necessary to always reference its static class:

```
var sqrt = Math.Sqrt(input);
```

- **String interpolation** simplified string formatting:

```
var output = $”Length of ‘{input}’ is {input.Length} characters.”;
```

It does not only avoid the call to `String.Format`, but also makes the formatting pattern easier to read:

```
var output = String.Format(“Length of ‘{0}’ is {1} characters.”, input, input.
Length);
```

Not to mention that having formatting pattern arguments outside the pattern makes it more likely to list them in the wrong order.

To learn more about C# 6, you can read my article Upgrading Existing C# Code to C# 6.0 in the DotNetCurry (DNC) Magazine.

C# 5.0

Microsoft released C# 5.0 in 2012 and introduced a very important new language feature: async/await syntax for asynchronous calls.

It made asynchronous programming much more accessible to everyone. The feature was accompanied by an extensive set of new asynchronous methods for input and output operations in .NET framework 4.5, which was released at the same time.

With the new syntax, asynchronous code looked very similar to synchronous code:

```
public async Task<int> CountWords(string filename)
{
  using (var reader = new StreamReader(filename))
  {
    var text = await reader.ReadToEndAsync();
    return text.Split(‘ ‘).Length;
  }
}
```

Just in case you’re not familiar with the `async` and `await` keywords, keep in mind that the I/O call to `ReadToEndAsync` method is non-blocking. The `await` keyword releases the thread for other work until the file read completes asynchronously. Only then, the execution continues back on the same thread (most of the times).

To learn more about async/await, check my article Asynchronous Programming in C# using Async Await – Best Practices in the DotNetCurry (DNC) Magazine.

Without the async/await syntax, the same code would be much more difficult to write, and to understand:

```
public Task<int> CountWords(string filename)
{
  var reader = new StreamReader(filename);
  return reader.ReadToEndAsync()
    .ContinueWith(task =>
    {
      reader.Close();
      return task.Result.Split(‘ ‘).Length;
    });
}
```

Notice, how I must manually compose the task continuation using the `Task.ContinueWith` method.

I also can't use the `using` statement anymore to close the stream because without the `await` keyword to pause the execution of the method, the stream could be closed before the asynchronous reading was complete.

And even this code is using the `ReadToEndAsync` method added to the .NET framework when C# 5.0 was released. Before that, only a synchronous version of the method was available. To release the calling thread for its duration, it had to be wrapped into a `Task`:

```
public Task<int> CountWords(string filename)
{
  return Task.Run(() =>
  {
    using (var reader = new StreamReader(filename))
    {
      return reader.ReadToEnd().Split(' ').Length;
    }
  });
}
```

Although this allowed the calling thread (usually the main thread or the UI thread) to do other work for the duration of the I/O operation, another thread from the thread pool was still blocked during that time. This code only seems asynchronous but is still synchronous in its core.

To do real asynchronous I/O, a much older and more basic API in the `FileStream` class needs to be used:

```
public Task<int> CountWords(string filename)
{
  var fileInfo = new FileInfo(filename);
  var stream = new FileStream(filename, FileMode.Open);
  var buffer = new byte[fileInfo.Length];
  return Task.Factory.FromAsync(stream.BeginRead, stream.EndRead, buffer, 0,
  buffer.Length, null)
  .ContinueWith(_ =>
  {
    stream.Close();
    return Encoding.UTF8.GetString(buffer).Split(' ').Length;
  });
}
```

There was only a single asynchronous method available for reading files and it only allowed us to read the bytes from a file in chunks, therefore we are decoding the text ourselves.

Also, the above code reads the whole file at once which doesn't scale well for large files. And we're still using the `FromAsync` helper method which was only introduced in .NET framework 4 along with the `Task` class itself. Before that, we were stuck with using the asynchronous programming model (APM) pattern directly everywhere in our code, having to call the `BeginOperation` and `EndOperation` method pairs for each asynchronous operation.

No wonder, asynchronous I/O was rarely used before C# 5.0.

C# 4.0

In 2010, C# 4.0 was released.

It was focused on **dynamic binding** to make interoperability with COM and dynamic languages simpler. Since Microsoft Office and many other large applications can now be extended by using the .NET framework directly without depending on COM interoperability, we see little use of dynamic binding in most of C# code today.

To learn more about dynamic binding, check my article Dynamic Binding in C# in the DNC Magazine.

Still, there was an important feature added at the same time, which became an integral part of the language and is frequently used today without giving it any special thought: **optional and named parameters**. They are a great alternative to writing many overloads of the same function:

```
public void Write(string text, bool centered = false, bool bold = false)
{
  // output text
}
```

This single method can be called by providing it any combination of optional parameters:

```
Write("Sample text");
Write("Sample text", true);
Write("Sample text", false, true);
Write("Sample text", bold: true);
```

We had to write three different overloads before C# 4.0 to come as close to this:

```
public void Write(string text, bool centered, bool bold)
{
  // output text
}

public void Write(string text, bool centered)
{
  Write(text, centered, false);
}

public void Write(string text)
{
  Write(text, false);
}
```

And even so, this code only supports the first three calls from the original example. Without the named parameters, we would have to create an additional method with a different name to support the last combination of parameters, i.e. to specify only `text` and `bold` parameters but keep the default value for the `centered` parameter:

```
public void WriteBold(string text, bool bold)
{
  Write(text, false, bold);
}
```

C# 3.0

C# 3.0 from 2007 was another major milestone in the language development. The features it introduced all revolve around making LINQ (Language INtegrated Query) possible:

- **Extension methods** appear to be called as members of a type although they are defined elsewhere.
- **Lambda expressions** provide shorter syntax for anonymous methods.
- **Anonymous types** are ad-hoc types which don't have to be defined in advance.

All of these contribute towards the LINQ method syntax we are all so used to today:

```
var minors = persons.Where(person => person.Age < 18)
  .Select(person => new { person.Name, person.Age })
  .ToList();
```

Before C# 3.0, there was no way to write such declarative code in C#. The functionality had to be coded imperatively:

```
List<NameAndAge> minors = new List<NameAndAge>();
foreach(Person person in persons) {
  if (person.Age > 18)
  {
    minors.Add(new NameAndAge(person.Name, person.Age));
  }
}
```

Notice how I used the full type to declare the variable in the first line of code. The `var` **keyword** most of us are using all the time was also introduced in C# 3.0. While this code doesn't seem much longer than the LINQ version, keep in mind that we still need to define the `NameAndAge` type:

```
public class NameAndAge {
  private string name;
  public string Name
  {
    get
    {
      return name;
    }
    set
    {
      name = value;
    }
  }

  private int age;
  public int Age
  {
    get
    {
      return age;
    }
    set
    {
      age = value;
    }
  }

  public NameAndAge(string name, int age)
  {
    Name = name;
    Age = age;
  }
}
```

The class code is much more verbose than we're used to because of another two features that were added in C# 3.0:

- Without **auto-implemented properties** I must manually declare the backing fields, as well as the trivial getters and setters for each property.

- The constructor for setting the property values is required because there was no **initializer syntax** before C# 3.0.

C# 2.0

We've made our way to the year 2005 when C# 2.0 was released. Many consider this the first version of the language mature enough to be used in real projects. It introduced many features which we can't live without today, but the most important and impactful one of them was certainly support for **generics**.

None of us can imagine C# without generics. All the collections we're still using today are generic:

```
List<int> numbers = new List<int>();
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);

int sum = 0;
for (int i = 0; i < numbers.Count; i++)
{
  sum += numbers[i];
}
```

Without generics, there were no strongly typed collections in the .NET framework. Instead of the code above, we were stuck with the following:

```
ArrayList numbers = new ArrayList();
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);

int sum = 0;
for (int i = 0; i < numbers.Count; i++)
{
  sum += (int)numbers[i];
}
```

Although the code might seem similar, there's an important difference: this code is not type safe. I could easily add values of other types to the collection, not only `int`. Also, notice how I cast the value I retrieve from the collection before using it. I must do that because it is typed as `object` inside the collection.

Of course, such code is very error prone. Fortunately, there was another solution available if I wanted to have type safety. I could create my own typed collection:

```
public class IntList : CollectionBase
{
  public int this[int index]
  {
    get
    {
      return (int)List[index];
    }
    set
    {
      List[index] = value;
    }
  }

  public int Add(int value)
  {
    return List.Add(value);
  }

  public int IndexOf(int value)
  {
    return List.IndexOf(value);
  }

  public void Insert(int index, int value)
  {
    List.Insert(index, value);
  }

  public void Remove(int value)
  {
    List.Remove(value);
  }

  public bool Contains(int value)
  {
    return List.Contains(value);
  }

  protected override void OnValidate(Object value)
  {
    if (value.GetType() != typeof(System.Int32))
    {
      throw new ArgumentException("Value must be of type Int32.", "value");
    }
  }
}
```

The code using it would still be similar, but it would at least be type safe, just like we're used to with generics:

```
IntList numbers = new IntList();
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);

int sum = 0;
for (int i = 0; i < numbers.Count; i++)
{
  sum += numbers[i];
}
```

However, the `IntList` collection can only be used for storing `int`s. I must implement a different strongly typed collection if I want to store values of a different type.

And the resulting code is still significantly less performant for value types because they are boxed to objects before being stored in the collection and unboxed when retrieved.

There are many other features we can't live without today which were not implemented before C# 2.0:

- **Nullable value types,**
- **Iterators,**
- **Anonymous methods**
- **...**

Conclusion:

C# has been the primary language for .NET development since version 1.0, but it has come a long way since then. Thanks to the features that were added to it version by version, it's staying up-to-date with new trends in the programming world and is still a good alternative to newer languages which have appeared in the meantime.

Occasionally, it even manages to start a new trend as it did with the `async` and `await` keywords, which have later been adopted by other languages. With support for nullable reference types and many other new features promised for C# 8.0, there's no fear that the language will stop evolving.

In case you are interested in learning about the upcoming features in C# 8, read www.dotnetcurry.com/csharp/1440/csharp-8-new-features

Microsoft 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 Yacoub Massad *for reviewing this article.*

Yacoub Massad

WRITING PURE CODE IN C#

In this article, I will show you how to use PurityAnalyzer—an experimental Visual Studio extension I wrote—to help you write pure code in C#.

INTRODUCTION

Functional programming has been gaining popularity. Languages that started as object-oriented have been getting functional programming features. C# for example, has been getting many features related to functional programming.

In the Functional Programming for C# Developers article, Damir Arh talked about some functional features of C#. In the same article, he also talked about pure functions.

A pure function is a function whose output depends solely on the arguments passed to it. If we invoke a pure function twice using the same input values, we are guaranteed to get the same output. Also, a pure function has no side effects.

All of this means that a pure function cannot mutate a parameter, mutate or read global state, read a file, write to a file, etc. Also, a pure function cannot call another function that is impure.

In the same article, the author also mentions that in C#, developers are required to have some discipline to write pure code. This is true because in C#, it is easy to mutate a parameter, get or set the value of a static field, read from or write to a file, etc.

The Haskell programming language has a different story altogether.

In Haskell, functions are pure by default and values are immutable by default. This means that writing pure functions in Haskell requires much less discipline (if any) than doing so in C#.

To help myself and other developers write and maintain pure code in C#, I have started working on a Visual Studio extension called PurityAnalyzer. Using this extension, developers can mark certain code as pure, and the analyzer would check the purity of the code and generate errors if the code is not pure.

In this article, I will work on an existing program that contains impure code. My target is to make the code pure. I will use the PurityAnalyzer extension to help me do so.

Please note that my focus in this article is on code purity. In the examples I discuss, I am not going to worry about other coding practices such as testing, good naming.

THE TIC-TAC-TOE GAME

In this article, I will walkthrough a tic-tac-toe game written in C#. I am going to assume that the reader is already familiar with this game. If you are not, please read about it here: https://en.wikipedia.org/wiki/Tic-tac-toe

Look at the first version of the game code here: https://github.com/ymassad/PureCodeInCSharpExample/tree/FirstImpureVersion

I also suggest that you run the game and see how it is played. The TicTacToeGame project has the following classes:

- The `Board` class represents the 9-cells board of the game. Internally it stores the cell contents as a jagged array of an Enum called `CellStatus`. This Enum could be one of three: Empty, HasX, or HasO. This class contains methods to get and set the contents of a cell given its row and column. It also contains a `PrintToConsole` method that prints the contents of the board to the console in a formatted way. It also contains an `IsFull` method that can determine whether all cells have been played. It also contains a property called `Winner` which is of type `Player? (Nullable<Player>)`. The caller can use this property to check who has won the game; player X or O, if any.

- The `Game` class controls the whole game. The `PlayGame` method starts a game session. It creates a new `Board` object and then allows each player to play until the game is over, either when there is a winner or when the board is full. This method calls the `PlayOneTurn` method multiple types. Each call will ask the current player to select a cell (represented by a row and a column) to play.
  The `PlayMultipleTimes` method keeps calling the `PlayGame` method until the user decides to exit.

The `Main` method simply creates a new `Game` object and invokes the `PlayMultipleTimes` method.

Is the code in these classes pure? Let's find out.

If you like to follow the steps I describe here, please install the PurityAnalyzer extension for Visual Studio 2017 from: https://marketplace.visualstudio.com/items?itemName=yacoubmassad.PurityAnalyzer

Making the Game class pure

Open the Game.cs file and annotate the class with the IsPure attribute like this:

```
1     using System;
2
3     namespace TicTacToeGame
4     {
5         [IsPure]
6         public sealed class Game
7         {
8             public static int numberOfTimesXWon;
9             public static int numberOfTimesOWon;
10
11            public void PlayMultipleTimes()
12            {
```

Figure 1: Annotating the Game class with the IsPure attribute

Because the `IsPure` attribute is not defined in code, Visual Studio shows a red squiggle underneath the annotation.

Visual Studio can help with generating the attribute. If you hover over the annotation, Visual Studio will show the Roslyn lightbulb. You can now choose to generate the attribute.

Here is the definition of the attribute if you want to add it manually:

```
public class IsPureAttribute : Attribute
{
}
```

By annotating the class with the `IsPure` attribute, we claim that all methods in the class are pure. Now, the PurityAnalyzer extension will analyze the class methods and generate errors for any code that makes the methods impure.

```
[IsPure]
public sealed class Game
{
    public static int numberOfTimesXWon;
    public static int numberOfTimesOWon;

    public void PlayMultipleTimes()
    {
        do
        {
            PlayGame();

            Console.WriteLine("Number of times X won: " + numberOfTimesXWon);
            Console.WriteLine("Number of times O won: " + numberOfTimesOWon);
        } while (PlayAgain());
    }
```

Figure 2: Errors generated because of impure code

Figure 2 shows part of the `Game` class. In particular, it shows the `PlayMultipleTimes` method. The `Console.WriteLine` method is impure because it writes to the console. Also, accessing the `numberOfTimesXWon` and the `numberOfTimesOWon` fields makes the method impure because these fields are mutable (non-readonly).

I will talk more about mutable fields later. For now, let's talk about `Console.WriteLine`.

Many methods in the class use `Console.WriteLine`. To work towards making these methods pure, we need to get rid of `Console.WriteLine`.

In a previous article, Writing Honest Methods in C#, I talked about making impure methods potentially-pure by replacing direct invocations of impure functions with invocations of functions passed as parameters. Let's apply this here.

Because we are working with instance methods, we can either have the function parameters added as method parameters, or we can add them as constructor parameters. For now, I will use the second option. Here is how the class looks like now:

```
[IsPure]
public sealed class Game
{
    public static int numberOfTimesXWon;
    public static int numberOfTimesOWon;

    private readonly Action<string> writeToConsole;

    public Game(Action<string> writeToConsole)
    {
        this.writeToConsole = writeToConsole;
    }

    public void PlayMultipleTimes()
    {
        do
        {
            PlayGame();

            writeToConsole("Number of times X won: " + numberOfTimesXWon);
            writeToConsole("Number of times O won: " + numberOfTimesOWon);
        } while (PlayAgain());
    }
```

Figure 3: Extracting a writeToConsole function as a constructor parameter

Figure 3 shows how the `Game` class now takes a `writeToConsole` function as a parameter and how the `PlayMultipleTimes` method calls it.

I have also changed other methods in the `Game` class to use the `writeToConsole` parameter instead of directly calling `Console.WriteLine`.

Notice how there are no red squiggles underneath the invocation of the `writeToConsole` function. PurityAnalyzer assumes that calls to delegates are always pure. It is the responsibility of the caller to pass a pure function if it wants to be pure.

Now, we have to go to the `Main` method and supply a value for the `writeToConsole` parameter like this:

```
static void Main(string[] args)
{
  new Game(Console.WriteLine).PlayMultipleTimes();
}
```

Of course, the `Main` method is not pure. It passes an impure function (`Console.WriteLine`) to the constructor of the `Game` class. But now, the impure dependencies of the `Game` class are no longer hidden. For more information about this, see the Writing Honest Methods in C# article.

To see how the code looks like now, take a look here:

https://github.com/ymassad/PureCodeInCSharpExample/tree/ExtractingWriteToConsole

Some methods in the class, e.g. the `PlayAgain` method, invoke the `Console.ReadLine` method to read a line from the console. Let's create a `readFromConsole` function parameter like we did for `Console.WriteLine`.

To see how the code looks like now, look here:

https://github.com/ymassad/PureCodeInCSharpExample/tree/ExtractingReadFromConsole

Let's work on the other impure method invocations. The `ReadRowAndColumnFromConsole` method invokes the `int.TryParse` method. You should see red squiggles underneath such invocation because the `int.TryParse` method is impure.

But why is it impure?

Consider the following code:

```
var result1 = int.TryParse("+1000", out var parsed1);

Thread.CurrentThread.CurrentCulture = (CultureInfo)Thread.CurrentThread.
CurrentCulture.Clone();

Thread.CurrentThread.CurrentCulture.NumberFormat.PositiveSign = "p";

var result2 = int.TryParse("+1000", out var parsed2);
```

This code invokes the `TryParse` method two times. It passes the same value ("+1000") in both invocations. If you run this code however, the first attempt to parse will be a success (result1 will be true), but the second one will be a failure (result2 will be false).

The reason the second one fails is that between the two calls, we modify the `NumberFormat.PositiveSign` property of the current `CultureInfo` object. For information about this topic, see the documentation of the CultureInfo class in the .NET Framework.

What is important to note here is that the behavior of the `TryParse` method does not depend solely on the arguments passed to it. Internally, it reads the `Thread.CurrentThread.CurrentCulture` property (which is the global state) to obtain number formatting settings that will make some decisions related to parsing.

There is another overload of the `TryParse` method that takes in an `IFormatProvider` parameter (implemented by NumberFormatInfo and `CultureInfo` for example) that we can use to specify the number formatting settings we want to use when parsing.

```
bool TryParse(string s, NumberStyles style, IFormatProvider provider, out int
result)
```

The problem with such overload though is that we can pass `null` as an argument for this parameter which will make the method fall back to using the current culture's number formatting settings. So, we cannot assume that this overload is pure.

Understandably, the .NET framework was not built with purity in mind.

What to do now? Should we extract the `TryParse` method as a function parameter like we did with `Console.WriteLine` and `Console.ReadLine`?

While we can do that, I don't think this is a good idea in the context of the TicTacToe game.

In our game, we simply need `TryParse` to parse the column and row numbers that the user entered. This is pure logic and we shouldn't have to extract it.

There are three rows and three columns so we need to parse only the following strings: "1", "2", "3".

We can create a simple method to do that like this:

```
static bool TryParse1Or2Or3(string str, out int result)
{
  if (str == "1") { result = 1; return true; }

  if (str == "2") { result = 2; return true; }

  if (str == "3") { result = 3; return true; }

  result = 0;
  return false;
}
```

...or create our own `TryParse` method from scratch if we want to.

The first solution might be acceptable in this case because we only need to handle a few cases. The second solution includes writing large amounts of code which is not a very good thing.

Had the second overload of `TryParse` been designed in a way to reject a `null` argument for the `IFormatProvider` parameter, we would have considered this method to be pure and used it here.

The PurityAnalyzer extension is not smart enough to know that the second overload of `TryParse` is pure if the argument passed for the `IFormatProvider` parameter is not null.

To overcome these issues, let's create the following methods:

```
public static class PureInt
{
  [AssumeIsPure]
  public static bool TryParse(
```

```
    this string s,
    NumberStyles style,
    IFormatProvider provider,
    out int result)
    {
      if (provider == null)
        throw new Exception($"{nameof(provider)} cannot be null");

      return int.TryParse(s, style, provider, out result);
    }

    public static bool TryParseCultureInvariant(
      this string s,
      out int result)
      {
        return TryParse(s, NumberStyles.Integer, NumberFormatInfo.InvariantInfo,
        out result);
      }
}
```

The `PureInt.TryParse` method simply invokes the second overload of the `int.TryParse` method I talked about. It also checks if the `IFormatProvider` argument is `null` and throws an exception if this is the case. This means that we can now assume that this method is pure.

This method is annotated with a special attribute, the `AssumeIsPure` attribute, to tell `PurityAnalyzer` that even if it cannot infer that the method is pure, it is actually pure.

The `TryParseCultureInvariant` method invokes the `TryParse` method passing `NumberFormatInfo.InvariantInfo` as the argument for the `IFormatProvider` parameter. This makes the method use number formatting settings that are culture-independent. See the documentation for this property for more information.

I changed the `ReadRowAndColumnFromConsole` method to call the `PureInt.TryParseCultureInvariant` method instead of `int.TryParse`.

To see how the code looks like now, look here:

https://github.com/ymassad/PureCodeInCSharpExample/tree/CreatingPureIntTryParse

Note: another solution is to consider the `int.TryParse` method pure enough and tell PurityAnayzer to consider it pure via the options page. See the last section of this article for more details.

Let's continue working on the Game class. Inside the `PlayBoard` method, PurityAnalyzer is telling us that the `PrintToConsole` method in the `Board` class is impure.

We should expect this because this method writes to the console. If we go to the implementation of the method, we can see it invokes `Console.WriteLine`.

Let's fix it really quick.

Let's add a `writeToConsole` function parameter to this method and use it there. The `PlayBoard` method can easily pass the value in the `writeToConsole` field as an argument to this method.

To see how the code looks like now, take a look here:
https://github.com/ymassad/PureCodeInCSharpExample/tree/InjectingWriteToConsoleInPrintToConsole

Let's work on the next issue. In the `PlayMultipleTimes` method of the `Game` class, two mutable static fields are read: `numberOfTimesXWon` and `numberOfTimesOWon`.

Reading mutable fields makes a method impure because when the method is called more than once, the values of such fields are not guaranteed to hold the same values. Since the method output might depend on such values, such output might be different on each invocation, even if we use the same input values each time.

"But wait a minute", you might say. "The PlayMultipleTimes method returns `void`. So, the output of the method would be the same even if we call it multiple times. And even if the values of the static fields are different each time."

Although the `PlayMultipleTimes` method returns void, it has some output; an indirect output. This method has indirect output through the `writeToConsole` constructor parameter (stored inside a field). The method invokes such function parameter to output the number of times each player won so far. For more information about direct and indirect output, see the Dependency Rejection blog post by Mark Seemann.

I hope that you see how such indirect output might be different if we invoked the `PlayMultipleTimes` method twice.

How to make the `PlayMultipleTimes` method pure? By making it not read global state.

Who updates this global state?

If we right click on the fields inside Visual Studio and select Find All References, we can see that these fields are updated inside the `SetCell` method of the `Board` class. Basically, when the `SetCell` method is invoked, it checks if there is a winner and increments one of the two static fields if any.

The following figure shows how the `PlayMultipleTimes` method calls the `SetCell` method indirectly:

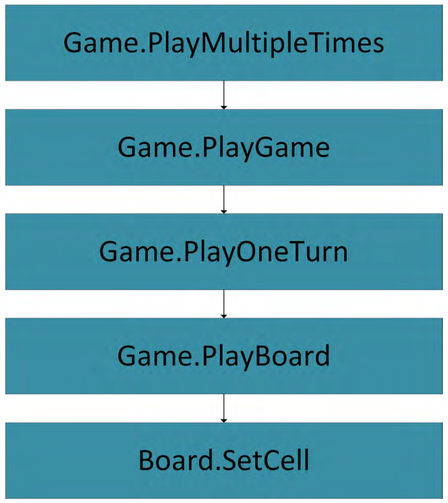


Figure 4: The call path from PlayMultipleTimes to SetCell

There are many ways to fix the issue at hand. I plan to do the following:

1. Make the `SetCell` method not update the static fields

2. Make the `PlayGame` method return `Player?` instead of void to indicate to the caller (the `PlayMultipleTimes` method in this case) which player won the game, if any. The `PlayGame` method can obtain such information from the `Winner` property of the `Board` object that is available in local scope.

3. Replace the static fields with local variables inside the `PlayMultipleTimes` method. The `PlayMultipleTimes` method will update such variables based on the output it receives from the `PlayGame` method (of type `Player?`).

Here is how the `PlayMultipleTimes` method looks like now:

```
public void PlayMultipleTimes()
{
    int numberOfTimesXWon = 0;
    int numberOfTimesOWon = 0;

    do
    {
        var winner = PlayGame();

        if (winner == Player.O)
            numberOfTimesOWon++;
        else if(winner == Player.X)
            numberOfTimesXWon++;

        writeToConsole("Number of times X won: " + numberOfTimesXWon);
        writeToConsole("Number of times O won: " + numberOfTimesOWon);

    } while (PlayAgain());
}
```

Figure 5 The PlayMultipleTimes method after replacing the fields with local variables

We still have state. The `numberOfTimesXWon` and `numberOfTimesOWon` are mutable local variables. Their values might change on each iteration of the loop. What we have done is reduce the scope of the state from global to local.

To see how the code looks like now, look here:

https://github.com/ymassad/PureCodeInCSharpExample/tree/RemovingStaticFields

Since we fixed the issue with accessing the static fields, there are no longer any red squiggles inside the `PlayMultipleTimes` method. This means that the `PlayMultipleTimes` method is pure. Or is it?

Actually, the `PlayMultipleTimes` method ***is still not pure***.

The `PlayMultipleTimes` method calls the `PlayBoard` method indirectly, via the `PlayGame` and the `PlayOneTurn` methods. The `PlayBoard` method is still impure as evident by the red squiggles underneath the call to `SetCell`.

Why are there no red squiggles underneath the invocation of `PlayGame` inside the `PlayMultipleTimes` method then?

When PurityAnalyzer analyzes the `PlayMultipleTimes` method and finds an invocation of the `PlayGame` method, it checks whether the `PlayGame` method or its containing type (the `Game` class) are marked as pure.

Since the `Game` class is marked with this attribute, `PurityAnalyzer` does not analyze the code of `PlayGame` at this moment and assumes that it is pure.

Of course, the `PlayGame` method will be analyzed on its own after the analyzer is done with the `PlayMultipleTimes` method. But again, when the analyzer sees the call to `PlayOneTurn`, it will see that the containing class is marked as pure and will not check the purity of the `PlayOneTurn` method.

This is not a problem because PurityAnalyzer is still detecting that the `PlayBoard` method is impure. Once we fix this, PurityAnalyzer's findings about `PlayMultipleTimes`, `PlayGame`, `PlayOneTurn` will be correct.

This is all done for performance reasons. Without such a way of analyzing, PurityAnalyzer will end up analyzing the same method multiple times.

If we remove the `IsPure` annotation on the `Game` class and just annotate the `PlayMultipleTimes` method with this attribute, `PurityAnalyzer` will show red squiggles underneath the call to `PlayGame` inside `PlayMultipleTimes`. This is because it will be forced to analyze the methods all the way down.

Making the Board class pure

We are now left with the impurity inside the `PlayBoard` method. PurityAnalyzer doesn't like the call to `SetCell` here. Let's go to the `SetCell` method and mark it as pure to see what's wrong with it.

```
[IsPure]
public void SetCell(int row, int column, CellStatus newValue)
{
    Cells[row][column] = newValue;

    if (newValue != CellStatus.Empty)
    {
        if (lines.Any(
            line => line.All(
                cell => Cells[cell.row][cell.column] == newValue)))
        {
            Winner = newValue == CellStatus.HasO ? Player.O : Player.X;
        }
    }
}
```

Figure 6: PurityAnalyzer complains about reading and writing mutable state in the SetCell method

Let's start with the access to the **lines** field.

Look at the definition of this field (line 56). The type of this field is `(int row, int column)[][]`. This is an array of lines on the board players can fill to win. This includes three horizontal lines, three vertical lines, and two diagonal lines (8 in total).

Each line is an array of cell locations. Each cell location is represented by a tuple containing row and column indexes.

If we hover over the lines field access inside the `SetCell` method, we would see that PurityAnalyzer is objecting to the fact that we are accessing a mutable field. There is no need for this field to be mutable since the information it contains need not change.

Let's make this field read-only.

Still, there are red squiggles underneath where it is accessed inside the `SetCell` method. However, when we hover over it again, we see a different error. PurityAnalyzer is now talking about an impure **cast**.

"What?"

What the code is doing is calling the Enumerable.Any method, as an extension method over the lines field. Here is the signature of this method:

```
public static bool Any<TSource>(this IEnumerable<TSource> source, Func<TSource,
bool> predicate);
```

The first parameter of this method is of type IEnumerable<TSource>. This means that when we call the Any method, the lines field which is of type array needs to be cast to `IEnumberable<TSource>`.

PurityAnalyzer is designed to assume that all abstract methods, e.g. interface methods, are pure. This allows us to create potentially-pure methods/classes by having them take parameters whose types are interfaces, not just delegates (Func and Action for example).

For more information about creating potentially-pure methods, see the Writing Honest Methods in C# article.

When casting from some type to `IEnumerable<T>`, PurityAnalyzer checks to see if the GetEnumerator method on that type is pure. If it is not, then PurityAnalyzer rejects the cast.

It does so because we shouldn't be able to trick PurityAnalyzer into accepting a call to an impure `GetEnumerator` method by first casting the object to `IEnumerable<T>` and then invoking the `GetEnumerator` method on that.

The `GetEnumerator` method on the array type is not pure because the array is mutable and invoking `GetEnumerator` twice might yield different results if the array contents change between the calls.

To think more concretely, think about the fact that even though the lines field is read-only, we can still mutate individual elements inside the array. For example, we can do this:

```
lines[0][0] = (5, 5);
```

Since the `SetCell` method depends on the lines field contents, its output (which is currently the new state of the current `Board` object) is not guaranteed to be the same on different calls, even if we pass the same input values.

To fix this problem, let's change the type of the field to use immutable arrays instead of mutable ones.

```
private readonly static ImmutableArray<ImmutableArray<(int row, int column)>> lines =
    ImmutableArray.Create(
        ImmutableArray.Create((0, 0), (0, 1), (0, 2)),
        ImmutableArray.Create((1, 0), (1, 1), (1, 2)),
        ImmutableArray.Create((2, 0), (2, 1), (2, 2)),
```

Figure 7: An excerpt from the updated lines field which is now an immutable array of immutable arrays

Now, the red squiggles underneath where we access the lines field inside the `SetCell` method disappeared.

The `Any` method that we are calling now is not `Enumerable.Any`. It is `ImmutableArrayExtensions.Any`. It has the following signature:

```
public static bool Any<T>(this ImmutableArray<T> immutableArray, Func<T, bool>
predicate)
```

This extension method acts directly on the `ImmutableArray<T>` type and therefore does not require a cast to `IEnumerable<T>`. This method is potentially-pure. If we give it the same immutable array, and the same pure predicate, the output will always be the same.

But even if we cast the lines field to `IEnumerabe<T>`, PurityAnalyzer will not object. The cast will be "pure". You can test this by writing the following line:

```
var enumerableLines = lines.AsEnumerable();
```

You will see no red squiggles here. However, if you change the type of the lines field back to an array, you will get the same impure cast error from before.

To see how the code looks like now, take a look here:

https://github.com/ymassad/PureCodeInCSharpExample/tree/MakingLinesArrayImmutable

Let's continue working on the other errors generated by PurityAnalyzer. There are three errors remaining:

1. Mutating the Cells array contents (instance property)
2. Reading the mutable Cells array contents (the same instance property)
3. Mutating the Winner instance property. Currently, when `SetCell` is called, we determine if there is a winner and we set the `Winner` property if so.

I am going to discuss two solutions.

Solution 1: Making the Board class immutable

The first solution is to make the `Board` class immutable. Currently, the `Board` class is mutable. When we invoke the `SetCell` method, two fields might be mutated. Here are the changes that I will make:

1. Change the type of the Cells property from `CellState[][]` to `ImmutableArray<ImmutableArray<CellState>>`. This means that once a `Board` object is created, its cell contents cannot be modified.

2. Make the `SetCell` method a static method that takes a `Board` object and returns another `Board` object that has the specified cell updated.

3. Create an additional constructor in the `Board` class that accepts an `ImmutableArray<ImmutableArray<CellState>>` array to initialize the `Cells` property. This constructor will be used by the `SetCell` method to create a `Board` object with the modified CellState values.

4. Remove the `Winner` property and instead create a static `GetWinner` method that the caller can use to determine if there is a winner based on the data in the `Cells` property. I chose to make this method

static, but it could also be an instance method.

5. Update the PlayBoard and `PlayOneTurn` methods in the `Game` class to return the new updated `Board` object.

6. Update the `PlayGame` method to keep track of the current Board object in the board variable. Mutating the board variable keeps the method pure because it is state that is local to the method.

7. In the `PlayGame` method, call `GetWinner` on the board object after each call to `PlayOneTurn` to see if there is a winner.

PurityAnalyzer now generates no errors.

To see how the code looks like now, look here:

https://github.com/ymassad/PureCodeInCSharpExample/tree/MakingBoardClassImmutable

Solution 2: Living with a mutable Board class

While the first solution is a good one for our particular game, it might not be acceptable in other programs in terms of performance.

In the first solution, each time we want to modify a single cell, we create two new immutable arrays. Let me explain.

The following figure represents the `Cells` property in the `Board` class. The red box represents the outer array, which is an array of arrays. The inner three blue rectangles represent the three arrays that are inside the outer array.

When we set a cell via `SetCell`, we create a new outer array, and then we take two of the inner arrays as-is and put them in the new outer array. We then create another array that contains the same values of the other inner array, except for the cell that we want to modify. We then put this new array in the new outer array.

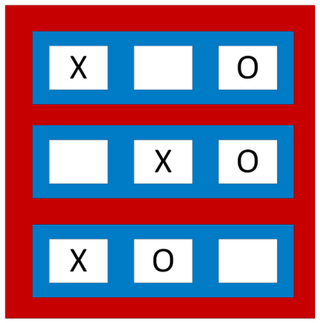


Figure 8: The outer array (in red), and the inner arrays (in blue)

The TicTacToe game is an interactive game and we expect the players to play one turn every second or so. So, the effect of creating the new arrays is negligible.

Imagine however that you are creating an application that needs to update one element in a 1000 by 1000 array 10000 times per second. Now, the performance cost of immutable objects might be significant.

The performance of immutable objects is not as bad as it might seem at first. Because such objects are immutable, we can reuse parts of one object in another object, and therefore we don't need to copy everything when we need to modify a little part of an immutable object.

Still, the performance cost of "modifying" an immutable object is larger than that of a mutable object.

Before we start working on the second solution, let's revert the code changes made in the last section.

To see how the code looks like now, take a look here:

https://github.com/ymassad/PureCodeInCSharpExample/tree/RevertingToMutableBoardClass

Here is how the `PlayBoard` method looks like now:

```
public bool PlayBoard(Player player, Board board, int row, int column)
{
    var cell = board.GetCell(row, column);

    if (cell == CellStatus.HasO || cell == CellStatus.HasX)
        return false;

    if (player == Player.O)
        board.SetCell(row, column, CellStatus.HasO);
    else
        board.SetCell(row, column, CellStatus.HasX);

    board.PrintToConsole(writeToConsole);

    return true;
}
```

Figure 9: The PlayBoard method is not pure because it invokes the impure SetCell method

The figure above shows the errors generated by PurityAnalyzer regarding the call to `SetCell`. To remind you, PurityAnalyzer is not happy because `SetCell` mutates the state inside the board object.

How to fix this?

Let's do the following changes:

1. Convert the `PlayOneTurn` method to a local function inside the `PlayGame` method. Remove the `Board` and `Player` parameters from such local function. Since `PlayOneTurn` is now inside the `PlayGame` method, it can access both the board and the `currentPlayer` local variables.

2. Convert the `PlayBoard` method to a local function inside the `PlayGame` method. Remove the `Board` and `Player` parameters from such local function. Since `PlayBoard` is now inside the `PlayGame` method, it can access both the board and the `currentPlayer` local variables.

3. Remove the arguments that are no longer needed when invoking `PlayOneTurn` and `PlayBoard`.

To see how the code looks like now, take a look here:

https://github.com/ymassad/PureCodeInCSharpExample/tree/PureExceptLocallyBoardClass

PurityAnalyzer no longer complains. But why? The `SetCell` method is still not pure!

PurityAnalyzer supports three levels of purity:

Purity Level	Description	Attribute
Pure	This is the strictest level of purity. A Pure method cannot call impure methods, cannot read or write instance or static mutable state, among other things.	IsPureAttribute
Pure-except-read-locally	Same as Pure, but methods with such purity level are allowed to read (but not write) instance mutable state in the objects that contain them. PurityAnalyzer allows such methods to be invoked on objects created in the current method and on objects passed as parameters to the current method.	IsPureExceptReadLocallyAttribute
Pure-except-locally	Same as Pure, but methods with such purity level are allowed to read or write instance mutable state in the objects that contain them. PurityAnalyzer allows such methods to be invoked on objects created in the current method.	IsPureExceptLocallyAttribute

The `SetCell` method is pure-except-locally. The only thing preventing it from being pure is the fact that it reads and mutates instance state (the `Cells` and `Winner` properties).

In the `PlayBoard` local function inside the `PlayGame` method in the `Game` class, we are calling `SetCell` on the `board` variable. PurityAnalyzer detects that this variable contains a new object, and therefore allows us to call the `SetCell` method on such a variable.

But why does `PurityAnalyzer` allow us to call pure-except-locally methods on new objects?

Because the state of such objects is local to the specific invocation of the calling method and therefore cannot affect the output of the calling method on later invocations.

Before the changes we made, the `SetCell` method was invoked on the board parameter. PurityAnalyzer does not allow us to invoke pure-except-locally methods on parameter-based objects.

The state of the PurityAnalyzer extension

The PurityAnalyzer extension is still under development. I have released version 0.6 of the extension so that the reader can experiment with it.

Currently there are 1539 automated tests written for the extension that cover a lot of the requirements that such an extension needs to support.

Still, there are cases where PurityAnalyzer will not detect impure code, and other cases where it will falsely consider pure code to be impure.

PurityAnalyzer works by analyzing code. But sometimes we call compiled methods, i.e. methods that we don't have the code for, e.g. the .NET framework base class library (BCL).

If we call a compiled method that has any of the attributes that PurityAnalyzer works with, e.g. the `IsPure` attribute, the extension will correctly handle calls to such methods.

On the other hand, if a method does not have such attributes, there are two ways for PurityAnalyzer to work with it:

- There is a list of BCL methods/types hardcoded in PurityAnalyzer that it uses to determine the purity of methods. For example, this file (embedded into the extension) contains a list of methods that PurityAnalyzer considers to be pure: https://github.com/ymassad/PurityAnalyzer/blob/master/PurityAnalyzer/Resources/PureMethods.txt

- Developers can configure PurityAnalyzer to consider some methods/types to be pure. The following figure shows the options page for PurityAnalyzer, which can be accessed using the Visual Studio menu (Tools -> Options, and then selecting Purity Analyzer on the left pane).

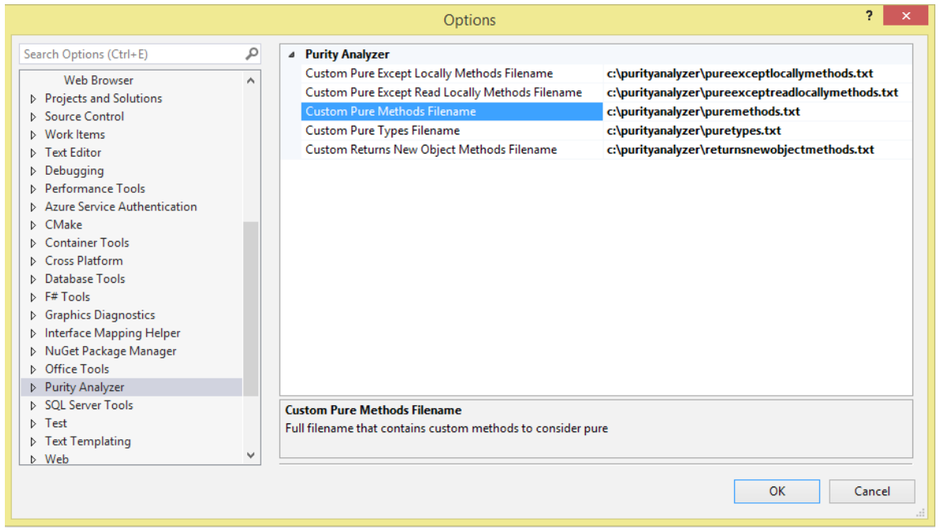


Figure 10: The PurityAnalyzer options page

Developers can create files that contain:

1. methods to consider pure

2. methods to consider pure-except-read-locally

3. methods to consider pure-except-locally

4. methods that return new objects. Objects returned by such methods are considered new objects and therefore it is legal to invoke pure-except-read-locally and pure-except-locally methods on such objects. Please note that for methods that we have the code for, PurityAnalyzer can automatically detect if the called method returns a new object. By the way, you can annotate such methods with the `ReturnsNewObjectAttribute`.

5. Which types are pure. All methods in pure types are considered pure.

The format of the files is the same as the format of the files in the Resources folder: https://github.com/ymassad/PurityAnalyzer/tree/master/PurityAnalyzer/Resources

The hardest part in making the PurityAnalyzer project succeed is to mark as many BCL methods/types as possible with the appropriate attributes.

This is one thing, you the reader can help with!

The PurityAnalyzer project is open source, you can update the files in the Resources folder and send me a pull request. This is not an easy task though. I spent some 30 minutes trying to figure out which overload of `int.TryParse` is pure to finally figure out that none of them are pure. I hope that one day the BCL will contain more APIs that are pure.

Conclusion:

In this article I presented PurityAnalyzer; a Visual Studio extension that can help you write pure code. I used the TicTacToe game as an example to show you how to start with code that is not pure and then make the code pure.

By using certain attributes to mark our methods/classes as pure, we can have PurityAnalyzer analyze the code and tell us whether the methods/classes we marked are actually pure. PurityAnalyzer generates errors for code that makes the marked methods impure.

PurityAnalyzer is still under development. I have released version 0.6 so that the reader can experiment with it.

• • • • • • •

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.*

Gerald Versluis

SERVERLESS APPS WITH AZURE FUNCTIONS

You probably have heard of the term Serverless. This new technology, which implies you don't have servers, is hot and happening and taking over the world.

In this article I will tell you what serverless is, how it is implemented with Azure Functions and how to use it with apps.

Spoiler alert: there are still servers when using serverless.

WHAT IS SERVERLESS?

As the name implies, there should be no servers. However, that is not quite the case. There *are* still servers, you just have to think about the servers, less.

The term serverless has been around for a while and there are actually some different definitions floating online.

In the earlier days, serverless meant ***Backend-as-a-Service (BaaS)*** solutions.

Think of third-party, cloud hosted services like Firebase, Auth0 or Parse (discontinued). These services provide you with a lot of out-of-the-box functionality that you can leverage without having to implement **too** much code. More importantly, you are not burdened with any code on the server-side or the infrastructure that comes with it.

Another definition that seems to be a lot more popular now is the ***Function-as-a-Service***. This is what we will be focusing on in this article.

Image Courtesy: https://blogs.msdn.microsoft.com

With functions, you *can* write your own code and logic, and nothing more. You do not need to create a big REST API project, worry about where to host it, how secure it is, etc. You just write a simple piece of code that does what you want and let the service provider do the rest.

What this essential means is - you have to think about the server, less.

What basically happens is that your function is installed inside a runtime on a webserver, and whenever a request comes in, the function is very rapidly deployed and executed.

When the execution completes, there is a small interval that waits for any subsequent call. After that, the function is destroyed until a next request comes in. This way, you don't have a long-running web application that is waiting for action, but instead is just idle most of the time.

There are several vendors that now provide this functionality: Amazon with Lambda, Google Cloud Functions and of course Microsoft Azure Functions.

STARTING WITH AZURE FUNCTIONS

Making a start with Azure Functions is very easy. Just log into the Azure Portal, create a new resource and look for the Function App. Look under Compute, or just search for it with the search box. You can see it in Figure 1.

A Function App is a container for one or multiple Azure Functions. Think of it as a web application where you will define multiple endpoints, each endpoint being a separate function.

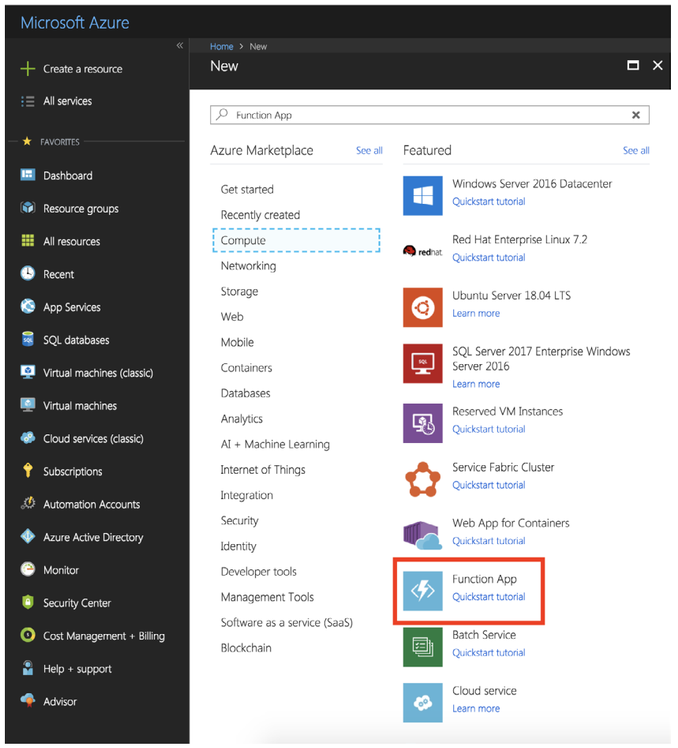


Figure 1: Creating our first Azure Function App in the Azure portal

Now, you will have to do some basic configuration of the Function App to be created. You will have to give it a (unique) name, which resource group to place it in, create an associated storage account, etc. It should all be self-explanatory.

The one option that is very interesting is the operating system option.

Here you can choose if your function should run on Windows, Linux or even Docker. Azure Functions are cross-platform and can be created with a variety of programming languages including JavaScript, PHP, Java and even Bash or PowerShell. For this example, we will just stick with Windows and C#, but it's good to know that you can leverage all of this in any other language, on any other platform.

Also, in terms of cost you might want to look at the hosting plan. The default is ***Consumption Plan***.

This means you will just pay for execution time. This is ideal, since you pay per second for the time the function is running. This is very precise and that makes it very cost effective in most cases.

However, you do have an overhead when your function is destroyed each time. You will have a cold start before each first request. Depending on how many external libraries you reference and how big your function is, this might take some time.

You can also choose the ***App Service Plan***.

This plan basically reverts your function back to the waiting REST API application that is dedicated for you, waiting for requests. It comes at a higher price, but you do not have to deal with the cold starts.

For this sample, I will stick with the Consumption Plan.

Go ahead with the configuration and click the Create button. After a bit of processing, your new Function App will be ready to go. You can verify that your function is up and running by going to https://yourfunctionname.azurewebsites.net. You will see a page that says your function app is now running as it should! See Figure 2.

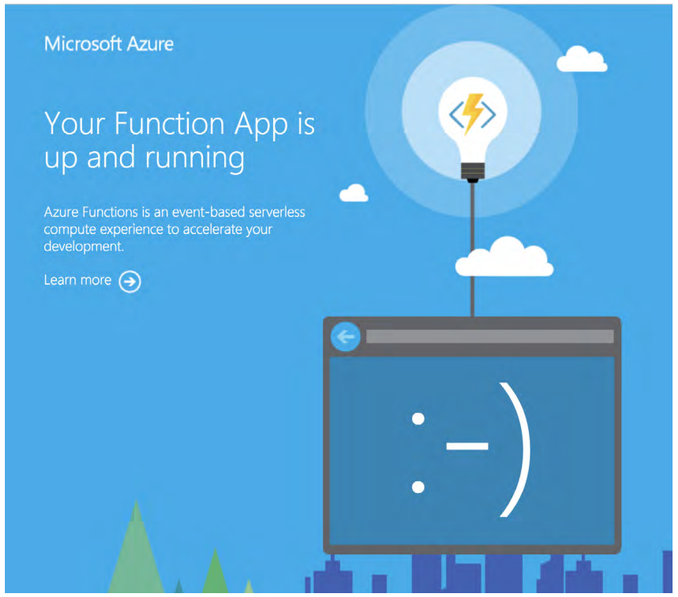


Figure 2: Status page that shows our AzureFunction App is running correctly

IMPLEMENTING A FUNCTION

There are several ways to construct your function.

This of course depends on what programming language you choose. One option you always have is to implement your function through the Azure portal. Yes, you've read that right, there is a simple code editor in the Azure portal that allows you to write code and deploy it.

In our example we'll be writing a C# function. For this we can leverage Visual Studio (VS), but you could also use VS for Mac or VS Code. They all have built-in support for Azure functions. Since I mostly develop cross-platform, I like to use my Mac with Visual Studio for Mac, but everything I will show here is also available in the other IDE's.

Let's start by creating a new Azure Functions project, no special SDK is needed as this is included in Visual Studio by default.

In Visual Studio (for Mac), create a new project and choose the Azure Functions template. You can find it under *Cloud*.

After selecting the Azure Functions template, you get to choose between a number of sub-templates. All these sub-templates provide you with different sample code to start from. This can help you get started with different kinds of triggers or implementations. You can see the selection screen in Visual Studio for Mac as shown in Figure 3.

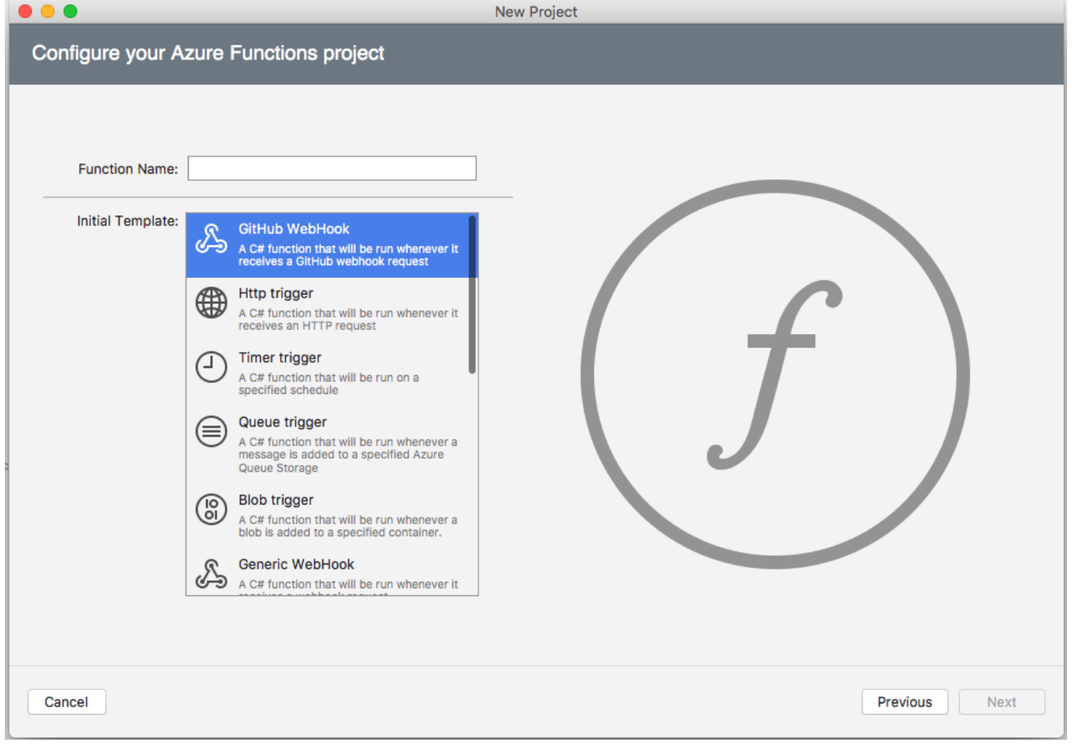


Figure 3: Selecting an Azure Functions template to get started

At the time of this writing, the Visual Studio for Windows templates aren't as complete as the ones you can find on Mac.

For this sample, I will select the Http Trigger template. And after that select Function as a value for the Access Rights when you are prompted. When the template is generated, you will get a sample implementation for free. Let's inspect what it does.

```
public static class HttpTrigger
{
  [FunctionName("HttpTrigger")]
  public static IActionResult Run([HttpTrigger(AuthorizationLevel.Function, "get",
  "post", Route = null)]HttpRequest req, TraceWriter log)
  {
    log.Info("C# HTTP trigger function processed a request.");
    string name = req.Query["name"];
    string requestBody = new StreamReader(req.Body).ReadToEnd();

    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;

    return name != null
      ? (ActionResult)new OkObjectResult($"Hello, {name}")
      : new BadRequestObjectResult("Please pass a name on the query string or in
      the request body");
  }
}
```

Overall what this function does is read the value from "name" and print that as "Hello, Gerald". The value for name can either come from the query string in your URL, or as a JSON value in the body.

There are a couple of interesting things in here.

You see references to a concept called triggers, a bit of authorization and a log. Let's look at these in a bit more detail.

TRIGGERS AND BINDINGS

A ***trigger*** is the starting point of our Azure function.

Each function has *exactly one* trigger. That trigger has some data associated to it, depending on what kind of trigger it is. This usually is the payload.

In our example above, the payload is the HTTP request, thus we are using an HTTP trigger. Other triggers can be a `TimerTrigger`, `QueueTrigger`, `BlobTrigger` and many more.

With these triggers you can respectively respond to: a timed interval, listening to an Azure Storage Queue or respond to the event when a file is added to a specified container. Chances are that a trigger that you might need is already available, but if need be, you can of course create your own custom trigger. For our example we will stick to the `HttpTrigger`, which is basically the equivalent of calling a REST API endpoint.

Then there is a concept called ***bindings*** which are a bit more extensive. With bindings, you can specify input and/or output in a declarative way. I don't know about you, but I don't like to write code that deserializes or persists data. The code is always mostly the same and just lists the properties of our objects. To overcome this, the Functions team has invented bindings.

With bindings you have a declarative way of handling incoming and outgoing data.

There are a lot of prefabricated bindings for you. For instance, you can have a binding for Cosmos DB. Because the Cosmos DB bindings can be used as an input *and* output binding, this means you can get and write data without having to write actual code for it.

Another example is the notification hub. You can easily trigger a push notification and send it to the Azure Notification Hub for sending without having to write any code to connect to the hub and send the notifications. Again, if you're missing a binding, you can implement it yourself.

For more information on triggers and bindings you can head over to the Microsoft Docs website: https://docs.microsoft.com/nl-nl/azure/azure-functions/functions-triggers-bindings. Here you will also find a great overview on which bindings are supported in which version of Azure Functions.

While bindings are a great concept, we won't go into details in this article.

AUTHORIZATION

You might not want your function to be called by everyone.

To accommodate this requirement, different authorization levels are available. At this time, you have the choice between anonymous, function, admin and system. With anonymous anyone can access the function. The other three need an API key to be able to access the function.

Then there is the matter of scope.

There are function keys and host keys. Function keys are valid for just that particular Azure Function and host keys are valid for the Function App. This means, all the functions within a Function App fall under the host key.

There is also a special _master key.

This one cannot be deleted, but it can be refreshed with a new value. This special key can be used across all functions in your function app for all access levels as it is a host key. To prevent you from locking yourself out, this _master key will always be present. Be careful with this and keep this a secret at all times!

On the Microsoft Docs pages, there is a good piece on authorization which you can read it here: https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook#authorization-keys.

LOGGING

Since you do not have to worry about the hosting of your application, this means that accessing logs like you traditionally would, is impossible. To provide you with an easy way to produce log output, the Azure Functions team has provided you with a built-in mechanism.

If you refer back to the code above, you can see that a `TraceWriter` is injected into the function. This object has different methods that correspond with different log levels like: info, trace or error. These logs

are collected in your Function App and can easily be reviewed from within the Azure portal.

If there is a need for a custom logging mechanism, you are free to do so. This will not be handled in this article.

CUSTOMIZING THE FUNCTION

Let's implement something of our own.

We will keep using the generated code that we saw in the previous steps and just remove the body of the function.

We are going to create a function that takes in an image and gives us back a description of what is in it. To achieve this, we will be using the Azure Cognitive Services. Underneath you can see the newly implemented function.

If you want to see the entire code, you can have a look at the repository I have set up for it here: https://github.com/jfversluis/ServerlessApps.

```
[FunctionName("HttpTrigger")]
public async static Task<IActionResult> Run([HttpTrigger(AuthorizationLevel.
Function, "get", "post", Route = null)]HttpRequest req, TraceWriter log) {
  log.Info("Function invoked");
  // We only support POST requests
  if (req.Method == "GET")
    return new BadRequestResult();

  // grab the key and URI from the portal config
  var visionKey = Environment.GetEnvironmentVariable("VisionKey");
  var visionEndpoint = Environment.GetEnvironmentVariable("VisionEndpoint");

  // create a client and request Tags for the image submitted
  var vsc = new ComputerVisionClient(new ApiKeyServiceClientCredentials(visionKey))
  {
    Endpoint = visionEndpoint
  };

    ImageDescription result = null;
    // We read the content as a byte array and assume it's an image
    if (req.Method == "POST") {
      try
      {
        result = await vsc.DescribeImageInStreamAsync(req.Body);
      }
      catch { }
    }
    // if we didn't get a result from the service, return a 400
    if (result == null)
      return new BadRequestResult();

    var bestResult = result.Captions.OrderByDescending(c => c.Confidence).
    FirstOrDefault()?.Text;

    return new OkObjectResult(bestResult
      ?? "I'm at a loss for words... I can't describe this image!");
}
```

I have added some comments which should clarify what happens. Overall, we start with determining if the request is a POST or a GET request.

If it's a GET, we return a HTTP 400 Bad Request result. If it's a POST, we continue, extract the image from the body and send it to the Azure Cognitive Services. If we get a result, we return that back to the caller.

The really cool thing here is, all that I had to do is install a Nuget package to communicate with the Cognitive Services.

You can use any NuGet package you like to help make your life easier. A word of warning: since your function is deployed often, this means the NuGet packages are reinstalled each time as well. The cold start-up time we have talked about earlier, comes into play here. The more packages you are using, the longer a cold start of your function will take.

You can test Azure Functions locally.

This is just as easy as debugging anything else. Just hit the run button and a command-line window will pop up, acting as a host for your Azure Function. Watch the output closely for any errors. If all goes well, you should be able to see the local URL of your function in the output as well. You can see the output on my Mac in Figure 4.

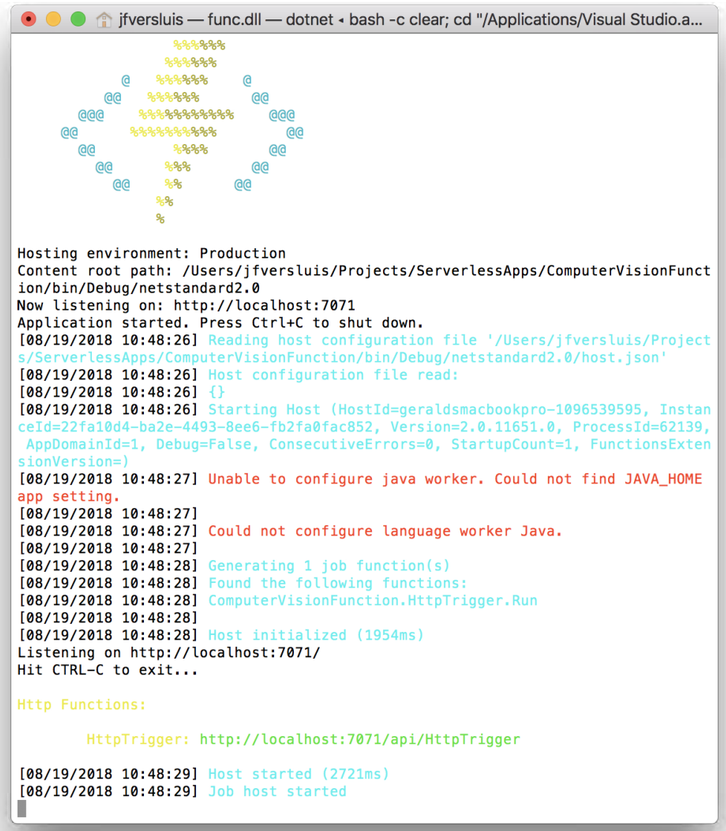


Figure 4: Debugging our Azure Function locally

When debugging locally, you can use a tool like Postman to fire HTTP requests at your local function to test the outcome. You can even setup breakpoints that will be hit.

DEPLOYMENT OF AZURE FUNCTIONS

Since we want to test it with a client, more specifically an app, we should deploy this to the Azure Function App we have set up earlier.

Depending on what IDE you're using deployment is very easy. You can just do a right-click deploy from Visual Studio for Windows, hook up your Azure subscription and you're good to go.

This functionality is not available in Visual Studio for Mac at the time of writing. Although it is available in the Beta and Alpha preview versions.

You can also setup deployment directly from your repository by going to the Azure portal. This is the route I took for now because I could not deploy from Visual Studio directly.

When you're in your Function App, you can choose to start from all kinds of premade functions. But if you look closely, at the bottom, there is the option to deploy a function from source control. You can see it in Figure 5.

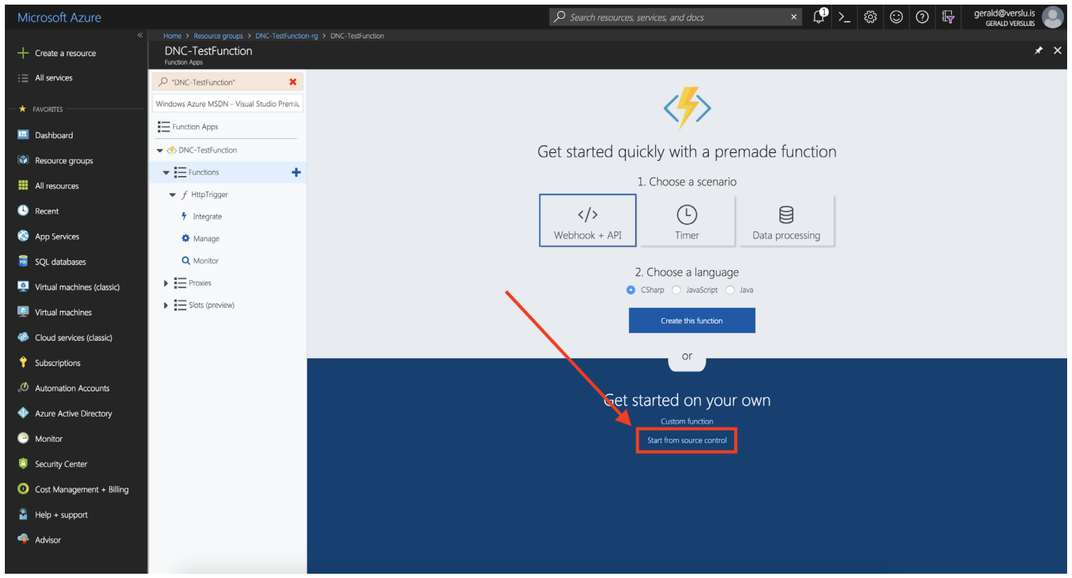


Figure 5: Setting up your Function straight from your repository

When choosing this option, you can integrate with your source control provider, GitHub in my case, and have the function be published directly from there. Doing so will setup an automated build and release for you under the hood.

This also means, when you push a new version to your repository, it gets published automatically.

When you set it up like this, your Function App turns into read-only mode, which means you can only make changes by pushing changes to your repository.

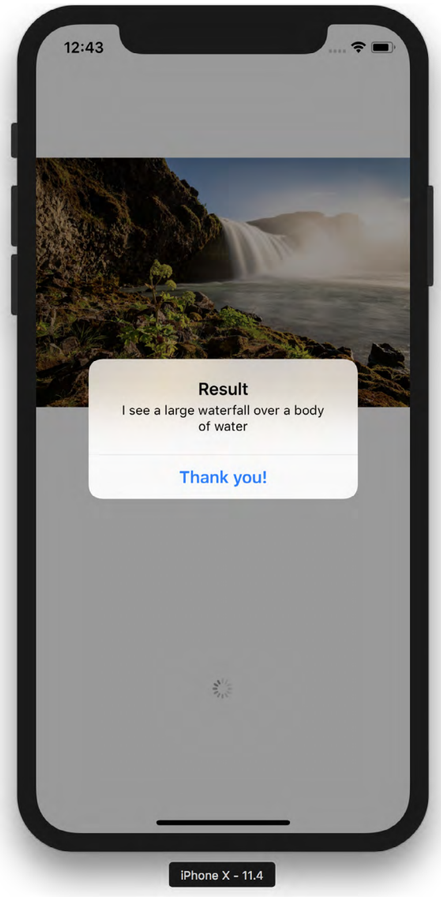
This is typically what you want. Normally, you would be able to change your function from the Azure portal. You can still do this by setting your function to read/write. When you do, remember that all changes you make will be overwritten on the next deployment from your repository.

Of course, you can also setup a CI/CD pipeline for your Azure Function from VSTS. This gives you full control over the process and how things are set up.

CONSUMING THE FUNCTION

For the last part, we will see how we can consume our function. Actually, this is the least exciting part since it is identical to any other REST server you would communicate with.

I have created a simple Xamarin app that will communicate with our Azure Function. This app allows you to pick an image from your phone or take one with your camera and upload that to our function. As we already know, our function will pick up the image, send it to the Cognitive Services and return what can be seen on the picture.



For handling the image, I have used a plugin by James Montemagno, the Xam.Plugin.Media. You can read more about it here: https://github.com/jamesmontemagno/MediaPlugin.

The most relevant part can be seen in the code underneath. Using a regular HttpClient, we do a POST request to the Azure Function, wait for the result and show that to the user.

```
using (var httpClient = new HttpClient())
{
  var request = new StreamContent(file.
GetStream());

  var httpResult = await httpClient.
PostAsync("https://dnc-testfunction.
azurewebsites.net/api/HttpTrigger",
request);

  var descriptionResult = await httpResult.
Content.ReadAsStringAsync();

  await DisplayAlert("Result", $"I see
{descriptionResult}", "Thank you!");

}
```

A sample result can be seen in Figure 6.

<< Figure 6: The result of our Xamarin app communication with the Azure Function

WRAPPING UP

The complete code for the example used in this article can be found in my GitHub account: https://github.com/jfversluis/ServerlessApps. There you will find the function as well as a Xamarin.Forms app with Android and iOS support.

To try things out, you can use the Function instance that I have hosted right now. Although if you really want to get the hang of it, I would encourage you to try and set things up yourself. It is very easy and you will have a lot of fun doing it.

In this article, we have learned what serverless is and how Microsoft has implemented this concept with Azure Functions. By using Functions, you can focus on what matters the most: your business logic. You do not need to worry about how to install it onto your IIS instance and all kinds of other boilerplate code to get your REST API up and running.

You could go with a full serverless architecture where basically each function could be a microservice or use the functions for simple, out-of-process tasks like resizing images or sending push notifications.

The possibilities are endless.

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

Thanks to Tim Sommer for reviewing this article.

Ravi Kiran

UNIT TESTING ANGULAR COMPONENTS

Testing components is one of the crucial parts of an Angular application. This article will get you started with testing in Angular and will show how to test components using different examples.

Needless to say, software can't be shipped to customers without testing it thoroughly. There are multiple ways to test an application - manual tests, automated integration tests, automated unit tests, performance tests and a few other techniques. Each of these techniques test different aspects of the application.

Of these, unit tests are always written by Developers.

Unit test is the technique that tests a piece of code in isolation. The piece of code has to be independent of any possible side effects when a set of unit tests are run on it.

Unit tests play an important role in software development. They help in catching the bugs early and ensure quality of the code. Unit tests also provide documentation for the code. If a file has enough number of unit tests and each test is described well, a new developer in the team would be able to understand code in that file, without any extra effort.

Not every piece of code written is unit testable.

The code has to follow a set of conventions so that it can be isolated while testing. Testability of the code depends on the code itself and the platform on which the code is written.

Angular is built with testing in mind. The modular architecture and the way Dependency Injection works, makes any of the Angular code blocks easier to test.

This tutorial will show how Angular CLI sets up the environment for unit testing and then shows how to test Angular components.

TESTING SETUP IN ANGULAR CLI

Create a new Angular application using Angular CLI. Run the following command to get the application created:

```
> ng new visitingplaces
```

The project created using Angular CLI contains the required setup to write and run the tests. The following set of packages are installed to support testing using Karma and Jasmine:

```
"jasmine-core": "~2.99.1",
"jasmine-spec-reporter": "~4.2.1",
"karma": "~1.7.1",
"karma-chrome-launcher": "~2.2.0",
"karma-coverage-istanbul-reporter": "~2.0.0",
"karma-jasmine": "~1.1.1",
"karma-jasmine-html-reporter": "^0.2.2",
```

Figure 1 – Packages for Karma and Jasmine installed by Angular CLI

The test setup is added to the file *karma.conf.js* inside the *src* folder. The following snippet shows the

default code in the *karma.conf.js* file:

```
module.exports = function (config) {
 config.set({
  basePath: ‘’,
  frameworks: [‘jasmine’, ‘@angular-devkit/build-angular’],
  plugins: [
   require(‘karma-jasmine’),
   require(‘karma-chrome-launcher’),
   require(‘karma-jasmine-html-reporter’),
   require(‘karma-coverage-istanbul-reporter’),
   require(‘@angular-devkit/build-angular/plugins/karma’)
  ],
  client: {
   clearContext: false // leave Jasmine Spec Runner output visible in browser
  },

  coverageIstanbulReporter: {
   dir: require(‘path’).join(__dirname, ‘../coverage’),
   reports: [‘html’, ‘lcovonly’],
   fixWebpackSourcePaths: true
  },
  reporters: [‘progress’, ‘kjhtml’],
  port: 9876,
  colors: true,
  logLevel: config.LOG_INFO,
  autoWatch: true,
  browsers: [‘Chrome’],
  singleRun: false
 });
};
```

The default framework used in this project is Jasmine.

The package @angular-devkit/build-angular contains the logic of building the webpack configuration for tests. It reads the angular.json configuration file and prepares the objects for running karma.

Plugins are the set of extensions we need to run the tests. Every framework configured in karma should also have a plugin added, so the list includes a plugin for Jasmine and one for @angular-devkit.

Karma also needs providers for the browser to be used, to generate test reports and to generate the code coverage. These plugins are configured using their respective NPM packages.

The `coverageIstanbulReporter` option contains the path where the coverage report has to be stored, the type of reports to be generated.

Rest of the configuration options set the reports to be generated after running the tests, port number on which Karma has to run, to keep karma watching for changes to the files, the browser to be used to run the tests and the `singleRun` option tells whether to exit after running the tests once.

The generated project contains some default tests. Let’s run them using the following command:

```
> ng test
```

This command generates output similar to the following:

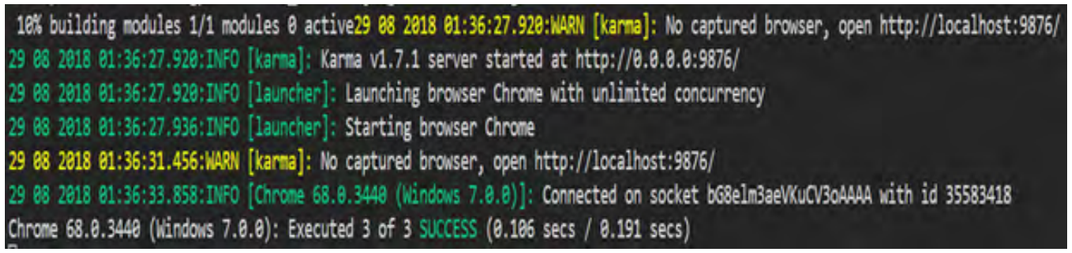


Figure 2 – Test result on the console

As the tests run on Chrome, a new instance of Chrome pops up with the URL set to http://localhost:9876 while running these tests. The page on the browser shows the list of tests executed and their status.

The following screenshot shows the output on the browser:

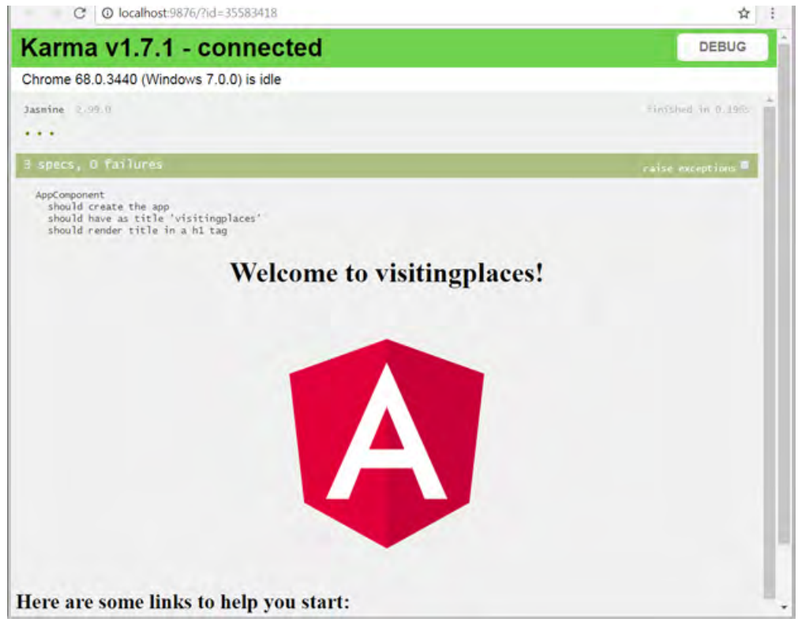


Figure 3 – Test result on the browser

Though the tests are executed in Chrome, it is preferred to use a headless browser like Chrome Headless to run unit tests in the CI/CD processes. For this, the package *puppeteer* has to be installed and ChromeHeadless has to be created as a custom launcher in the karma configuration. The following command installs the package *puppeteer*:

```
> npm install puppeteer --save -dev
```

This package has to be imported in the file *karma.conf.js* and the path has to be set to the environment. The following statements achieve this task:

```
const puppeteer = require('puppeteer');
process.env.CHROME_BIN = puppeteer.executablePath();
```

The following snippet configures the custom launcher for karma:

```
customLaunchers: {
  ChromeHeadless:  {
    base:   'Chrome',
    flags:  [
      '--headless',
      '--remote-debugging-port=9222',
      '--no-sandbox',
      '--proxy-server=\'direct://\'',
      '--proxy-bypass-list=*'
    ],
  }
},
```

Change the browser options to use ChromeHeadless.

```
browsers: ['ChromeHeadless'],
```

As the configuration shows, the custom launcher opens chrome in the headless mode. The browser won't be visible on the screen, but the Chromium engine would be used in the background to run the tests.

On running the tests now, you will find similar result on the console.

The only difference is, you won't see a new instance of the Chrome browser now. Hence, debugging would be difficult. This is OK as we won't use a CI/CD server for debugging.

TYPES OF TESTS

An Angular component can be tested in a number of different ways. This article will use the following techniques:

- **Component Class testing:** testing the component class alone in isolation
- **Shallow Integration Testing:** Testing the component using the APIs provided for tests after shallowing any other components or directives used
- **Deep Integration Testing:** Testing a component by creating stubs for the components and directives surrounding it

LEARNING FROM THE GENERATED TESTS

The code generated by the Angular CLI contains tests for the *AppComponent*. Reading the tests in this file would provide a good starting point.
Open this file in Visual Studio Code. The following snippet shows the initial statements in the file:

```
import { TestBed, async } from '@angular/core/testing';  // 1
import { AppComponent } from './app.component'; // 2
```

```
describe('AppComponent', () => {
 beforeEach(async(() => { // 3
  TestBed.configureTestingModule({ // 4
   declarations: [
    AppComponent
   ],
  }).compileComponents(); // 5
 }));
 //
});
```

The following listing explains the statements marked with numbers in the above snippet:

1. Statement 1 imports the `TestBed` and `async` objects from the package @angular/core/testing. The `TestBed` object is used to initialize the testing environment for an Angular object. The async method is used to wrap a piece of code in a test zone. It detects any asynchronous calls made in a block of code and completes the test when all the asynchronous calls are complete.

2. Statement 2 imports the component to be tested

3. Statement 3 is the beginning of the `beforeEach` block, this block is used to set up the environment and the objects required for the tests in this file. It runs before every test and for any sub-level describe blocks. The `beforeEach` block ensures that the state modified in a test doesn't cause any side effects in the next test

4. Statement 4 configures a test module using the component to be tested. It declares the component to be tested in the same way as the components are declared in the module

5. Statement 5 calls the `compileComponents` method on the test module. This is done to load the templates referred in the component using the `templateUrl` property and then to compile the components so that they can be created in the tests

Now let's understand the tests in this file. The following snippet shows the first test:

```
it('should create the app', async(() => {
  const fixture = TestBed.createComponent(AppComponent);
  const app = fixture.debugElement.componentInstance;
  expect(app).toBeTruthy();
}));
```

This test simply checks if the component instance is created. It creates the fixture of the component using the `TestBed.createComponent` method.

The `debugElement` property of the fixture object provides all the required debugging APIs of the component. The `componentInstance` property of the `debugElement` is used to get object of the component and then the assertion checks whether this object is created.

The next two tests check if the component's DOM is rendered correctly. Let's understand the third test. It is shown below:

```
it('should render title in a h1 tag', async(() => {
  const fixture = TestBed.createComponent(AppComponent);
  fixture.detectChanges();
  const compiled = fixture.debugElement.nativeElement;
```

```
  expect(compiled.querySelector('h1').textContent).toContain('Welcome to
visitingplaces!');
}));
```

This test checks if the `h1` element in the template is correctly rendered.

The `h1` element contains a binding expression that binds the `title` property of the component class. Angular updates the bindings in the template when the change detection runs. The `fixture.detectChanges` method runs the change detection on the component and updates the bindings.

Now we need the root DOM object of the component. This object will be used to query the elements containing bindings and then to check if they contain the right data.

`Fixture.debugElement.nativeElement` gives the DOM object of the component. All the standard DOM APIs can be applied on this object. The h1 element is selected using `querySelector` and then the test checks if it has the right content set.

The tests generated for the `AppComponent` are integration tests as these tests are based on the component instance. The test is not using shallows as the template of `AppComponent` doesn't contain any other directives or components. We will discuss the other techniques in the next section.

TESTING MORE SCENARIOS

The tests generated in Angular CLI provided some foundation to write tests for the components. Now let's write tests for a few components that cover more real-life scenarios.

If you want to follow along with rest of the article, use code in the folder named `before` from the sample code. This folder contains a simple Angular application with two components: `PlacesComponent` and `PlaceComponent`.

The following screenshot shows how this application looks:

The tests generated in Angular CLI provided some foundation to write tests for the components. Now let's write tests for a few components that cover more real-life scenarios.

If you want to follow along with rest of the article, use code in the folder named `before` folder from the sample code. This folder contains a simple Angular application with two components: `PlacesComponent` and `PlaceComponent`.

The following screenshot shows how this application looks:

Figure 4 – Output of the sample application

The `PlacesComponent` contains a select box with a list of places. It passes the selected place to the `PlaceComponent` as an input parameter. The `PlaceComponent` displays details of the place it accepts and raises an event when the button in the component is clicked.

In this section, we will write tests for both of these components.

Testing PlaceComponent

The `PlaceComponent` accepts a place object and displays the details of this place on the screen. It emits an event when the Button is clicked.

Let's test the functionality of this component in the three ways discussed earlier. To write these tests, add a file named *place.compone.spec.ts* to the *app* folder.

Testing as a Class

Testing a component as a class is similar to testing any TypeScript class. To test it, one needs to create an object of the class and invoke the functionality. Open the file *place.component.spec.ts* and add the following code to it:

```
import { PlaceComponent } from './place.component';

describe('PlaceComponent as class', () => {
  let placeComponent: PlaceComponent;

  beforeEach(() => {
    placeComponent = new PlaceComponent();
    placeComponent.place = {
      name: 'Charminar',
      city: 'Hyd',
      country: 'India',
      isVisited: true,
      rating: 4
    };
  });
});
```

If you are following along and writing the tests, have the `ng test` command running on a command prompt to see the test results as you make the changes.

As you see, the `beforeEach` block creates an instance of the `PlaceComponent` class and sets value to the property `place`. The `PlaceComponent` class has the property `IsVisited`. It returns Yes if the place is visited and No if the place is not visited.

The following tests verify this behavior:

```
it('should return Yes when the place is visited', () => {
  expect(placeComponent.IsVisited).toEqual('Yes');
});

it('should return No when the place is not visited', () => {
  placeComponent.place.isVisited = false;
  expect(placeComponent.IsVisited).toEqual('No');
});
```

The method `togglePlaceVisited` emits an event using the field `toggleVisited`. As this field is an `EventEmitter`, we can test if the emitted event contains the right data by subscribing to it. The following test verifies this:

```
it('should emit event when toggleVisited is called', () => {
  placeComponent.toggleVisited.subscribe(name => expect(name).
toEqual('Charminar'));
  placeComponent.togglePlaceVisited();
});
```

Testing as an Independent Component

To test as a component, we need to load a few objects from the libraries that were installed along with Angular. The following import statements load them:

```
import { NO_ERRORS_SCHEMA, Component } from '@angular/core';
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';

import { Place } from './place.model';
```

As you would have already guessed, we need to create a fixture object for the `PlaceComponent` in the `beforeEach` block. The following snippet creates the fixture:

```
describe('PlaceComponent Tests: As an independent component', () => {
 let place: Place,
  rootElement: HTMLElement;
 let fixture: ComponentFixture<PlaceComponent>;

 beforeEach(() => {
  fixture = TestBed.configureTestingModule({
   declarations: [PlaceComponent]
  })
  .createComponent(PlaceComponent);

  place = {
   name: 'Charminar',
   city: 'Hyd',
   country: 'India',
   isVisited: true,
   rating: 4
  };
  rootElement = fixture.debugElement.query(By.css('.thumbnail')).nativeElement as
  HTMLElement;
 });
});
```

The above `beforeEach` block creates the component fixture using a test module and then gets the root element of the component. To get it, it queries the `debugElement` using the *By.css* predicate.

The first test will check if the component instance is created. It is shown below:

```
it('should have initialized the component', () => {
  expect(fixture.componentInstance).toBeDefined();
});
```

A more useful test would be to check the behavior of the component after changing the data it accepts.

The `isVisited` property on the place is set to `true`, so we have the class `visited-place` set to the root element and the toggle button has the text *Mark Not Visited*. This behavior can be tested by inspecting the elements in the component. The following snippet tests this behavior:

```
it('should have applied the changes when the place set is visited', () => {
  fixture.componentInstance.place = place;
  fixture.detectChanges();

  expect(rootElement.classList).toContain('visited-place');
  expect(rootElement.querySelector("div.caption").textContent)
    .toEqual(place.name);
  expect(rootElement.querySelector('a').textContent).toEqual('Mark Not Visited');
});
```

Notice the call to `detectChanges` after setting the value of `place`. Bindings in the component are checked again and the view gets updated if state of the objects has changed.

When the place is not visited, the root element won't have the *visited-place* class set and the button will say *Mark Visited*. The following test verifies this:

```
it('should have applied the changes when the place set is not visited', () => {
  place.isVisited = false;
  fixture.componentInstance.place = place;
  fixture.detectChanges();

  expect(rootElement.classList).not.toContain('visited-place');
  expect(rootElement.querySelector('a').textContent).toEqual('Mark Visited');
});
```

When the button is clicked, the component raises the `toggleVisited` event. This can be tested by triggering event on the button using the DOM API.

The following snippet shows this test:

```
it(“should emit the event when the button is clicked”, () => {
  fixture.componentInstance.place = place;
  fixture.detectChanges();

  fixture.componentInstance.toggleVisited.subscribe((name) => expect(name).
toEqual(place.name));
  rootElement.querySelector(‘a’).click();
});
```

Testing using a Host Component

In an application using the *PlaceComponent*, the input property would be sent from the host component and the output event is also consumed by the host component.

In the sample application, the *PlacesComponent* hosts the *PlaceComponent*. It would be good to test this functionality using a mock host component, as that would really test if the *PlaceComponent* interacts correctly with the host.

The setup of this test suit will be different, as it will create a mock component. The mock component would be declared in the test module along with *PlaceComponent*. And the test will create a fixture object using the mock component instead of the component to be tested.

The following snippet sets up the required objects for the test suit:

```
describe(‘PlaceComponent Tests: Inside a test host’, () => {
 @Component({
  template: `<place [selectedPlace]=”selectedPlace”
 (toggleVisited)=”toggleVisited($event)”></place>`
})
class TestHostComponent {
 places: Place[] = [{
  name: ‘Charminar’,
  city: ‘Hyderabad’,
  country: ‘India’,
  isVisited: true,
  rating: 4
 },
 {
   name: ‘Taj Mahal’,
   city: ‘Agra’,
   country: ‘India’,
   isVisited: false,
   rating: 3
 }];
 selectedPlace: Place;
 placeName: string;
 constructor() {
  this.selectedPlace = this.places[0];
 }

 toggleVisited(name: string) {
  this.placeName = name;
 }
}
```

```
 let rootElement: HTMLElement;
 let fixture: ComponentFixture<TestHostComponent>;

 beforeEach(() => {
  fixture = TestBed.configureTestingModule({
  declarations: [PlaceComponent, TestHostComponent]
 })
 .createComponent(TestHostComponent);
 rootElement = fixture.debugElement.query(By.css(‘.thumbnail’)).nativeElement as
 HTMLElement;
 });
});
```

As you can see, the component *TestComponentHost* uses the *PlaceComponent* in its template. It binds a value to the `selectedPlace` input property and binds a method with the output property `toggleVisited`.

The fixture object is created using *TestHostComponent*.

The tests will manipulate data using the fixture object and state of the DOM would be asserted using root element of the *PlaceComponent*.

The first test will check for the initial state of the *PlaceComponent*. The *TestHostComponent* assigns a value to the field `selectedPlace` in the constructor, so the test will check if the DOM has the values in the object assigned. The following snippet shows this test:

```
it(‘should have the place component’, () => {
  fixture.detectChanges();
  expect(rootElement.querySelector(‘.caption’).textContent).toEqual(‘Charminar’);
});
```

When the event `toggleVisited` is triggered, the *TestHostComponent* should receive name of the place. The following test asserts this case by clicking the button programmatically:

```
it(‘should emit name of the place’, () => {
  fixture.detectChanges();
  let visitedLink = fixture.debugElement.query(By.css(‘a’)).nativeElement as
  HTMLElement;
  visitedLink.click();

  expect(fixture.componentInstance.placeName).toEqual(fixture.componentInstance.
  selectedPlace.name);
});
```

When *TestHostComponent* changes the place, bindings in the *PlaceComponent* have to be updated. This can be tested by assigning a new place to *selectedPlace*. The following test does this:

```
it(‘should change bindings when place is updated’, () => {
  fixture.componentInstance.selectedPlace = fixture.componentInstance.places[1];
  fixture.detectChanges();

  expect(rootElement.querySelector(‘.caption’).textContent).toEqual(‘Taj Mahal’);
});
```

Testing PlacesComponent

The *PlacesComponent* uses the service *PlacesService* and the *PlaceComponent* in its template.

Both of these have to be mocked while testing the PlacesComponent. It also uses the built-in directive *ngModel* on the select element.

As it is a built-in directive, you may load the Forms Module to the test module and test the component with the behavior of *ngModel*.

For the example test, I am going to shallow this directive, which means I will ask Angular to ignore any unknown HTML elements and attributes. For this, we need to import the schema *NO_ERRORS_SCHEMA* from *@angular/core* module and add it to schemas of the module as shown here:

```
import { NO_ERRORS_SCHEMA } from '@angular/core';
 //
 beforeEach(() => {
  //
  fixture = TestBed.configureTestingModule({
   declarations: [MyComponent],
   schemas: [NO_ERRORS_SCHEMA]
  })
  .createComponent(MyComponent);
 });
```

While testing a component that uses a service, the service has to be mocked. This is done to isolate the component from any side effects that the service could cause. The service is replaced with a mock implementation. To create the mock effectively, use the provider name of the service and assign it with a different class, factory function or an object.

The following snippet shows the setup for testing *PlacesComponent* with the mocks of *PlaceComponent* and *PlacesService*:

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';

import { NO_ERRORS_SCHEMA, Component, Output, Input, EventEmitter } from '@angular/
core';

import { PlacesComponent } from './places.component';
import { PlacesService } from './places.service';
import { Place } from './place.model';

@Component({
  selector: 'place',
  template: '<div></div>'
})
class MockPlaceComponent {
  @Input('selectedPlace')
  place: Place;
  @Output('toggleVisited')
  toggleVisited = new EventEmitter<string>();
}
```

```
describe('PlacesComponent tests', () => {
  let fixture: ComponentFixture<PlacesComponent>,
    service: PlacesService;
  let mockService;

  beforeEach(() => {
    mockService = {
      places: [{
        name: 'Charminar',
        city: 'Hyd',
        country: 'India',
        isVisited: true,
        rating: 4
      },
      {
        name: 'Taj Mahal',
        city: 'Agra',
        country: 'India',
        isVisited: false,
        rating: 3
      }],
      toggleVisited: jasmine.createSpy('toggleVisited')
    };

    fixture = TestBed.configureTestingModule({
        declarations: [PlacesComponent, MockPlaceComponent],
        providers: [{ provide: PlacesService, useValue: mockService }],
        schemas: [NO_ERRORS_SCHEMA]
      })
      .createComponent(PlacesComponent);
    service = fixture.componentRef.injector.get(PlacesService); //getting instance
of the service
    fixture.detectChanges();
  });
});
```

The service is mocked here using the `useValue` provider. The mock object created has the public members of the `PlaceService` class. It assigns a static array to the `places` property and a Jasmine spy method to the `toggleVisited` method.

Observe the statement that's used to get instance of the service. Injector of the component is used to get the service instance and the injection token is passed to the *injector.get* method. Once the component is initialized, the `select` box should be populated with data in the `places` array. The following test verifies this:

```
it('should have select box populated with all places', () => {
  expect(fixture.debugElement.queryAll(By.css('option')).length).toEqual(2);
});
```

By default, the *PlacesComponent* selects first item in the `places` array and it is passed to the place component. The next test verifies if the *place* component has received the right place.

```
it('should have assigned the selected date to place component', () => {
  let placeDebugElement = fixture.debugElement.query(By.css('place'));
  let place = placeDebugElement.componentInstance.place;
```

```
  expect(place.name).toEqual('Charminar');
});
```

Once `selectedPlace` is modified in the *PlacesComponent*, the new place has to be assigned to the *PlaceComponent*. The following test verifies this:

```
it('should have assigned the selected date to place component when the selection
changes', () => {
  fixture.componentInstance.selectedPlace = mockService.places[1];

  fixture.detectChanges();
  let placeDebugElement = fixture.debugElement.query(By.css('place'));
  let place = placeDebugElement.componentInstance.place;

  expect(place.name).toEqual('Taj Mahal');
});
```

When the `toggleVisited` method on the component is called, it should call the `toggleVisited` method on the service. The following test verifies this:

```
it('should call the toggle method in the service', () => {
  fixture.componentInstance.toggleVisited('some name');
  expect(mockService.toggleVisited).toHaveBeenCalledWith('some name');
});
```

Conclusion

Unit testing is an important aspect to test applications as it ensures quality of the code along with reducing the possible bugs.

As shown in this tutorial, Angular makes the process of testing easier by isolating each block of code from the other and by providing a number of helper APIs to work with the framework while testing. Components handle the most important aspects of an application and they have a lot of variations. Hope this tutorial got you started with testing different types of components. The future articles will discuss unit testing of other code blocks.

Download the entire source code from GitHub at
bit.ly/dncm38-unittesting-angular

Microsoft® Most Valuable Professional

Ravi Kiran
Author

Ravi Kiran (a.k.a. Ravi Kiran) is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active blogger, *an author at SitePoint and at DotNetCurry. He is rewarded with Microsoft MVP (ASP.NET/IIS) and DZone MVB awards for his contribution to the community.*

Thanks to Keerti Kotaru *for reviewing this article.*