Issue 30 | May-June 2017

DNC Magazine

www.dotnetcurry.com

Revisiting
SOLID

DATA AND ENCAPSULATION
in complex C# applications

Comparing
VUE.JS with
ANGULAR and REACT

ASP.NET CORE
Custom Binders

Implementing
Multi-Platform
Libraries with
TYPESCRIPT

ERROR HANDLING
in large projects

BDD
an in-depth look

Developing an
ANGULAR 4
application with Bootstrap 4

from the editor

I am watching Microsoft Build 2017 Live while writing this editorial note and I am as excited as I can be to catch up on the plethora of new products and annoucements Microsoft has in store for us, to set the course for the year ahead. But at the same time, I can't help but ponder over what Satya said in his keynote "***The future of computing is going to be defined by the choices that you as developers make and the impact of those choices on the world***". He also stressed the need to "build trust in technology".

But how to we build trust and woo the naysayers? In my humble opinion, we do it by giving users "more" information on how these technologies work.

Satya also enthused over using technology to bring more empowerment to more people, similar to what Microsoft researcher Haiyan Zhang did by creating a wearable prototype called "Emma" to alleviate graphic designer Emma Lawton's Parkinson tremors, helping her write again.

While we developers catch up on these radical expansions of Artifical intelligence and Cloud computing power, social responsibility and ethics must not take a back seat.

Enjoy our 30th edition and do let me know how was it. E-mail me at suprotimagarwal@dotnetcurry.com.

Suprotim Agarwal

Editor in Chief

dot net curry.com

Editor In Chief
Suprotim Agarwal
suprotimagarwal@
dotnetcurry.com

Art Director
Minal Agarwal

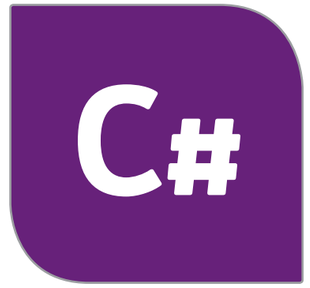
Contributing Authors
Andrei Dragotoniu
Benjamin Jakobus
Craig Berntson
Damir Arh
Daniel Jimenez Garcia
Francesco Abbruzzese
Keerti Kotaru
Yacoub Massad

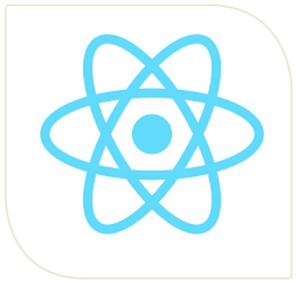
Technical Reviewers
Damir Arh
Mahesh Sabnis
Ravi Kiran
Suprotim Agarwal
Yacoub Massad

Next Edition
July 2017

www.dotnetcurry.com/
magazine

Contents





THANK YOU

FOR THE 30th EDITION


@yacoubmassad


@craigber


@ravikiran


@dani_djg


@damirarh


@benjaminjakobus


@keertikotaru


@andrei74_uk


@f_abbruzzese


@suprotimagarwal


@maheshdotnet


@ saffronstroke

WRITE FOR US

Andrei Dragotoniu

The article explains how Behavior Driven Development (BDD) works and provides a real-world example of how to use it to provide meaningful tests which can act as living documentation.

BEHAVIOR DRIVEN DEVELOPMENT (BDD) – an in-depth look

Expectations

This article expects the readers to be familiar with the testing mindset in general, it will however touch on how things can be built, taking advantage of SOLID principles and other methods of writing testable code.

Behavior Driven Development (BDD) – a quick description and example

BDD stands for **B**ehavior **D**riven **D**evelopment. The syntax used to describe the behavior is Gherkin.

The idea is to **describe what should happen in a language, as naturally as possible**.

If you are familiar with Unit Testing and are comfortable writing unit tests, then you are familiar with the way they read. Depending on how much a test needs to cover, it can be quite difficult to work out what it does, because it is after all, just code.

Only a developer can really understand what happens there.

BDD tackles the issue in a different way.

Let's hide the code and start a conversation, so much so that now anyone can read a scenario and understand what it tests.

Let's take an example:

Given a first number 4

And a second number 3

When the two numbers are added

Then the result is 7

There is no code here. This scenario can be read like a story. We can give it to a Business Analyst to make sure we're tackling the right thing, or give it to a tester, or can revisit this later and refresh our memory on how things need to work, and why did we build things a certain way.

We are describing a bit of behavior here, in this case, it could be a Math operations sub-system where we have clearly defined one of the behaviors of this system. Of course, more tests are to be written to cover the complete behavior and take care of edge cases.

If this all starts to sound like writing unit tests, then that's a good thing.
BDD and Unit testing in some respects are similar and do not prevent developers from using both, if that is appropriate.

Using the Gherkin syntax makes it very easy to explain what is being tested in a natural language, which even non-developers can read and understand.

A QA person or a Business Analyst, for example, could copy and paste such a test, change the numbers and come up with their own test cases, without having to write any code at all, or without even seeing the code.

Here is a very good writeup on Gherkin in case you are interested in details:
https://github.com/cucumber/cucumber/wiki/Gherkin

Now we have the test, how does it all work from here onwards?

Each line in the test is called a step, each step becomes a separate method, and each method gets called in the order they are written.

In our example on the previous page, the first two lines (the *Given* and the *And*) will setup the initial data, the *When* will take care of calling the method we want to test, and the *Then* is where the assert will happen.

Since each step is a separate method, hopefully by now it is obvious that we need to be able to share some *state* between steps.

Don't worry, this isn't *state* as you think of it and it doesn't break any of the testing principles, especially the one which says that *a test should never alter state or should depend on state created by another test.*

It simply means that each test needs to be able to have its own state and that state needs to be available for every step in that test.

Specflow gives us a ScenarioContext which is just a dictionary and is used to store the data needed for executing the test. This Context is cleared at the end of the test and it will be empty again when the next test runs.

With each step as a separate method, one last point to be considered here is that the step can be reused between multiple tests. Look at the first two steps in our test example. If we pass the number as an input parameter to this step method, we can reuse it wherever we reuse the steps.

The test looks more like this:

Given a first number {parameter1}

And a second number {parameter2}

When the two numbers are added

Then the result is {expected result}

Now that is much more generic and hopefully clearly shows the reusability of each step. We don't have to use the same steps in every test and they don't even need to be in the same order! We'll take a look at this a bit later.

As we keep adding tests, the actual code we write becomes smaller because for each system behavior we are testing, we will get to the point where we simply reuse the existing steps we have already coded.

So even if we spend a bit of time initially writing the testing code; as we advance, eventually the amount of time spent on writing additional steps goes down to virtually zero.

Software used for BDD

We need to see what tools can help us harness the full power of BDD. This article is written from a back-end point of view, but there are alternatives for pure front end work as well, but they won't be discussed in in this article.

I will be using:

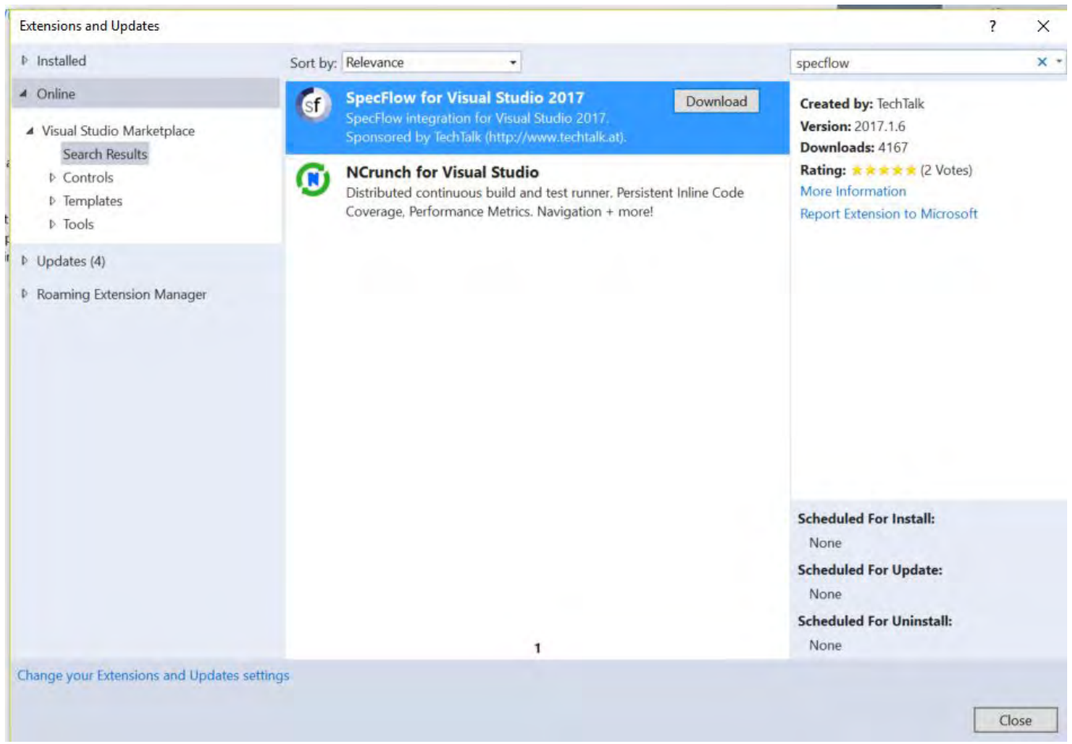
- **Visual Studio 2017** (bit.ly/dnc-vs-download)

- **Specflow** – Visual Studio extension – this will help with the Gherkin syntax and the link between the tests and the steps code.

- **NUnit** – used for Asserts. You can use something else here, **FluentAssertions** works just as well.

There is one NuGet package which installs both Specflow and NUnit, I'd use that one as it makes things easier.

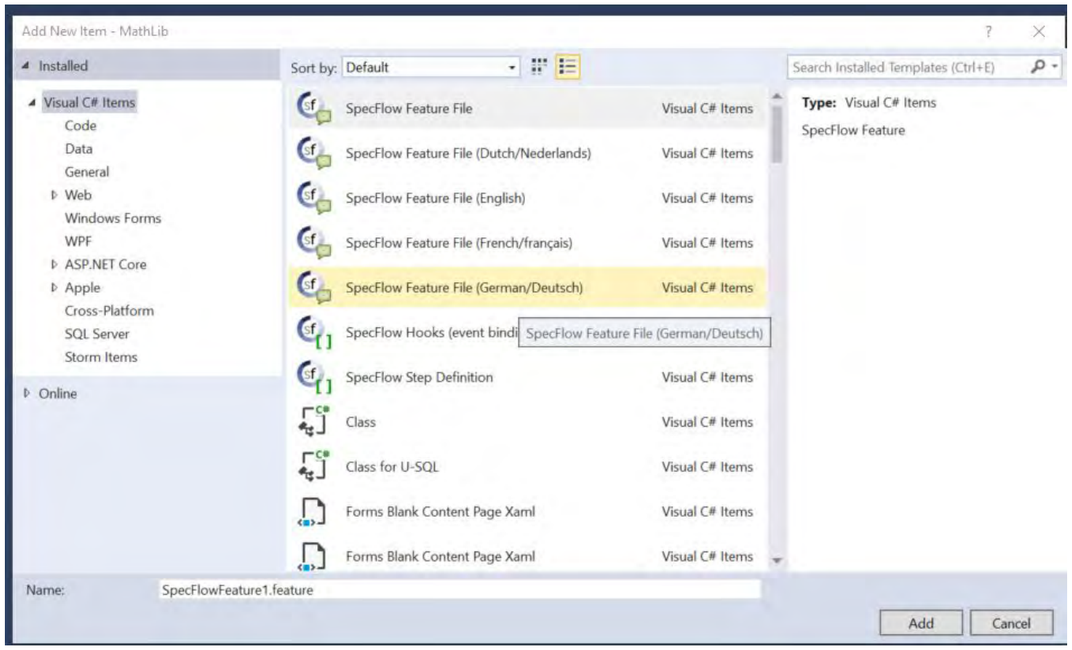
So, first install the Visual Studio Specflow extension. This will give us the file templates and the syntax coloring.



Specflow

The Specflow Visual Studio extension will allow you to create feature files. These files are the placeholder for your test scenarios.

The extension also adds syntax coloring to the feature files which is a nice visual indicator on what you have done and what you still need to do. It creates a connection between the steps of each test scenario and the test method behind them, which is quite handy especially when you have lots of feature files and lots of tests.



Once a feature file is created, it will look like this:

```
Feature: MethodAddTests
      In order to avoid silly mistakes
      As a math idiot
      I want to be told the sum of two numbers

@mytag


Scenario: Add two numbers
      Given I have entered 50 into the calculator
      And I have entered 70 into the calculator
      When I press add
      Then the result should be 120 on the screen
```

The feature text describes the problem.

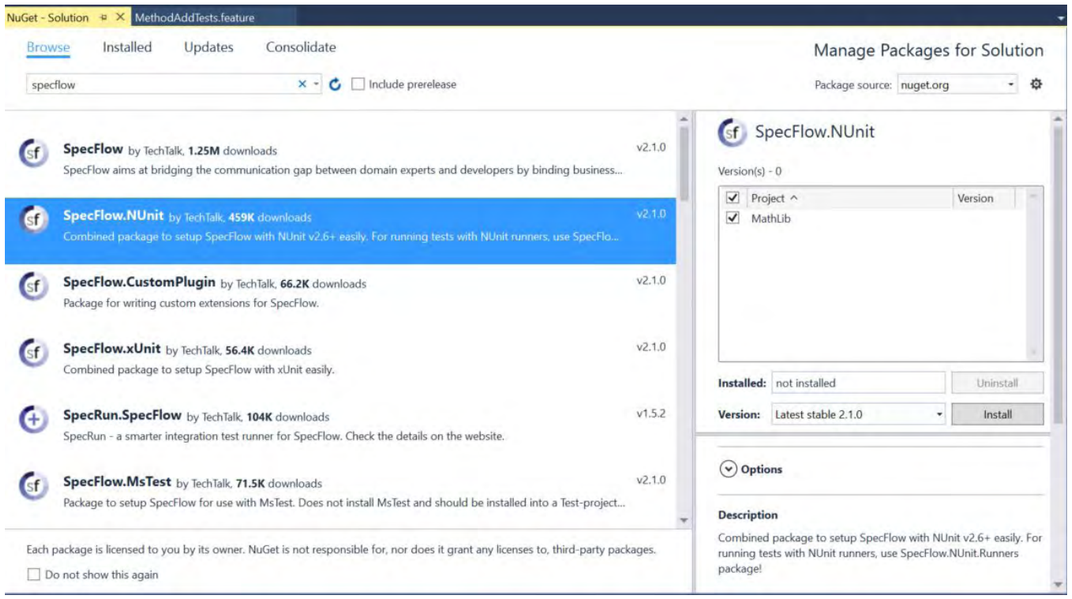
The scenario is basically one test and we can have multiple scenarios in one feature file.

The tag is used in the Test Explorer window and it allows us to group tests in a logical manner. Our initial test could look like this:

```
@MathLib @Add
Scenario: Add two numbers
      Given a first number 3
      And a second number 4
      When the two numbers are added
      Then the result should be 7
```

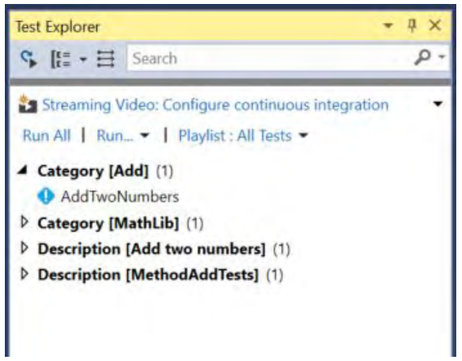
Please note how the references to UI elements have been removed. This goes back to what was said initially - **focus on functionality**, as well as on the core bits that do something; not how things are displayed and where.

In the Visual Studio solution, we still need to install Specflow and NUnit using the NuGet package SpecFlow.NUnit:



I created a MathLib class library and added this NuGet package to it.

Once we have all these packages installed, open the Test Explorer window, build the solution and you should see the following:



I filtered by Traits, which then shows the tags we created. I used two, the *MathLib* to show all the tests in the library (Add, Divide etc.), but then I can see them grouped by Math operation as well as under the *Add* tag for example. This is again just personal preference. The tags can be quite a powerful way of grouping your tests in a way which makes sense to you.

So now we have a feature file, as well as a test, but we haven't written any test code yet.

What we need next is a steps code file, where all the steps for our tests can go. We will start with one file, but we can separate the steps into multiple step files, to avoid having too much code in one file.

If you have another look at our test, you'll see that the steps are colored in purple. This is a visual indicator that there is no code yet.

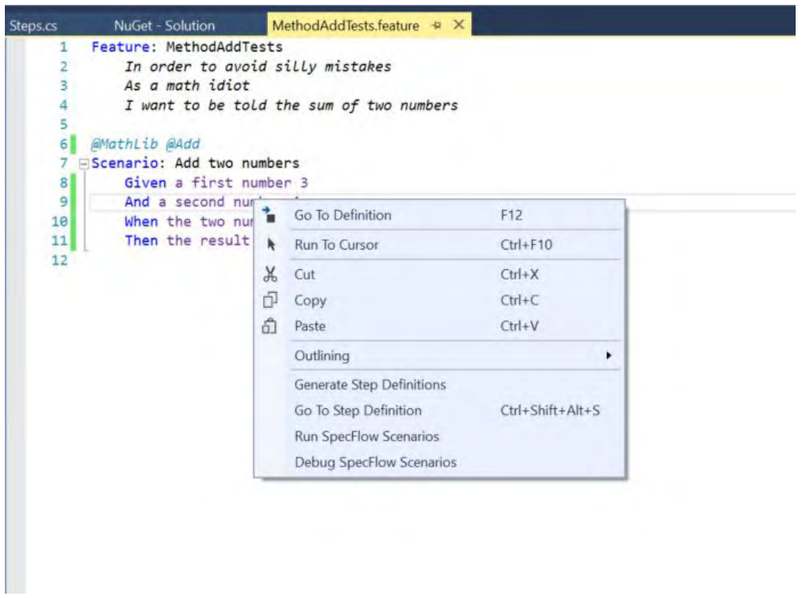
Let's create a steps code file, which is just a standard C# file.

The code looks like this:

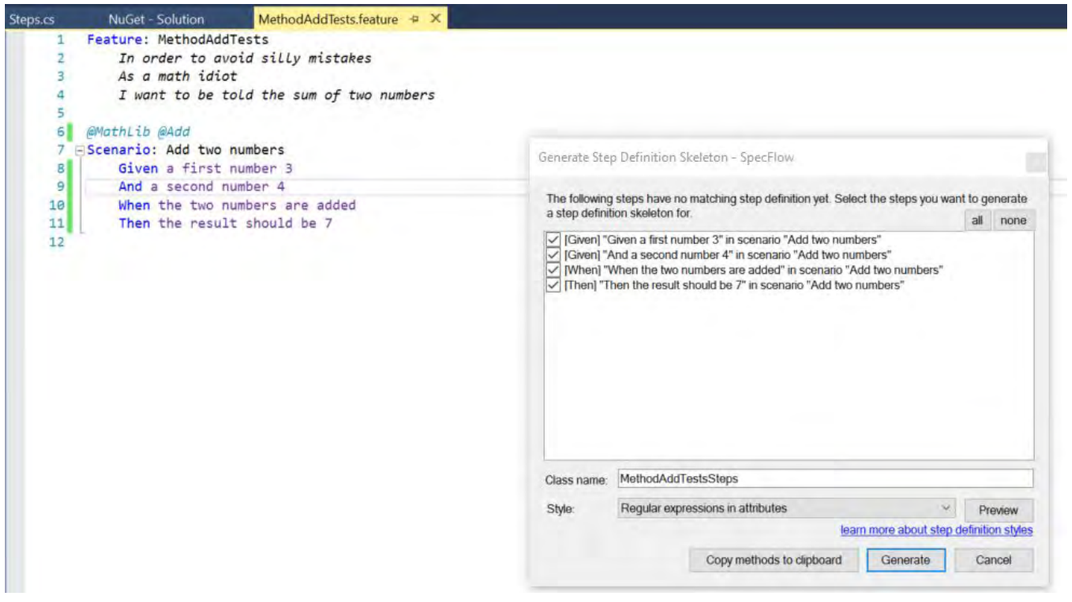
```
using TechTalk.SpecFlow;

namespace MathLibTests
{
  [Binding]
  public sealed class Steps
  {
  }
}
```

The only thing we added is the `Binding` attribute at the top of the class. This is a Specflow attribute and it makes all the steps in this file available to any feature file in this project, wherever they may be located.

Now, go back to the feature file, right click on any of the steps and you will see a *Generate Step Definitions* option in the context menu:



Click the *Generate Step Definitions* option and then *Copy methods to clipboard*:



Notice how the four steps appear in the window. Code will be generated for each one of them.
Now simply paste the code in the steps file created earlier:

```
using TechTalk.SpecFlow;

namespace MathLibTests
{
  [Binding]
  public sealed class Steps
  {
    [Given(@"a first number (.*)")]
    public void GivenAFirstNumber(int p0)
    {
      ScenarioContext.Current.Pending();
    }
    [Given(@"a second number (.*)")]
    public void GivenASecondNumber(int p0)
    {
      ScenarioContext.Current.Pending();
    }
    [When(@"the two numbers are added")]
    public void WhenTheTwoNumbersAreAdded()
    {
      ScenarioContext.Current.Pending();
    }
    [Then(@"the result should be (.*)")]
    public void ThenTheResultShouldBe(int p0)
    {
      ScenarioContext.Current.Pending();
    }
  }
}
```

Save the file and then look at the feature file again. Our initial Scenario, which had all the steps in purple, now looks like this:

```
@MathLib @Add
Scenario: Add two numbers
      Given a first number 3
      And a second number 4
      When the two numbers are added
      Then the result should be 7
```

Notice how the color has changed to black and the numbers are in italic which means they are treated as parameters. To make the code a bit clearer, let's change it a little bit:

```
using TechTalk.SpecFlow;

namespace MathLibTests
{
  [Binding]
  public sealed class Steps {
    [Given(@"a first number (.*)")]
    public void GivenAFirstNumber(int firstNumber)
    {
      ScenarioContext.Current.Pending();
    }

    [Given(@"a second number (.*)")]
    public void GivenASecondNumber(int secondNumber)
    {
      ScenarioContext.Current.Pending();
    }

    [When(@"the two numbers are added")]
    public void WhenTheTwoNumbersAreAdded()
    {
      ScenarioContext.Current.Pending();
    }

    [Then(@"the result should be (.*)")]
    public void ThenTheResultShouldBe(int expectedResult)
    {
      ScenarioContext.Current.Pending();
    }
  }
}
```

At this point, we have the steps, we have the starting point and we can add some meaningful code.

Let's add the actual math library, the one we will actually test. Create a class library, add a `MathLibOps` class to it with the `Add()` method:

```
using System;

namespace MathLib
{
  public sealed class MathLibOps {
    public int Add(int firstNumber, int secondNumber) {
      throw new NotImplementedException();
    }
  }
}
```

Now let's write enough test code to have a failing test.

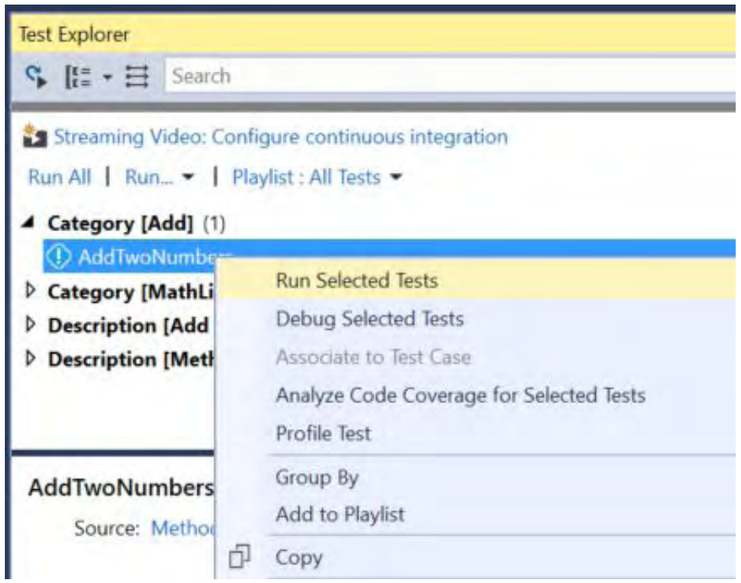
Let's look at the Steps file again. Notice all those `ScenarioContext.Current.Pending()` lines in every step? This is the Context we were talking about before. That's where we will put all the data we need. Think of it as a dictionary, with key /value pairs. The key will be used to retrieve the right data so we will give it some meaningful values to make our life easier.

```
using MathLib;
using NUnit.Framework;
using TechTalk.SpecFlow;

namespace MathLibTests
{
  [Binding]
  public sealed class Steps
  {
    [Given(@"a first number (.*)")]
    public void GivenAFirstNumber(int firstNumber)
    {
      ScenarioContext.Current.Add("FirstNumber", firstNumber);
    }

    [Given(@"a second number (.*)")]
    public void GivenASecondNumber(int secondNumber)
    {
      ScenarioContext.Current.Add("SecondNumber", secondNumber);
    }

    [When(@"the two numbers are added")]
    public void WhenTheTwoNumbersAreAdded()
    {
      var firstNumber = (int)ScenarioContext.Current["FirstNumber"];
      var secondNumber = (int)ScenarioContext.Current["SecondNumber"];

      var mathLibOps = new MathLibOps();
      var addResult = mathLibOps.Add(firstNumber, secondNumber);

      ScenarioContext.Current.Add("AddResult", addResult);
    }

    [Then(@"the result should be (.*)")]
    public void ThenTheResultShouldBe(int expectedResult)
    {
      var addResult = (int)ScenarioContext.Current["AddResult"];
      Assert.AreEqual(expectedResult, addResult);
    }
  }
}
```

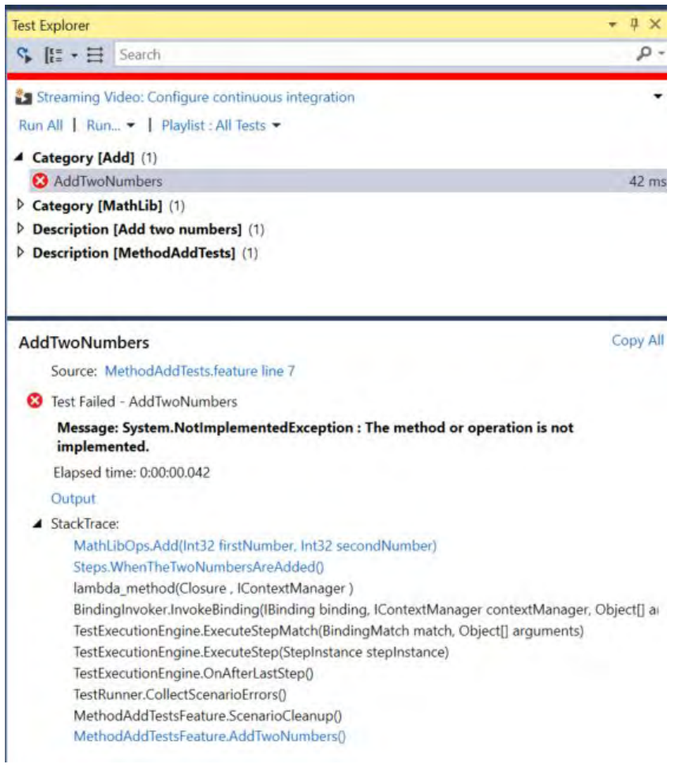
Look at the first two *Given* methods, notice how we take the parameters passed into the methods and then add them to the context with a clear key so we know what they represent.

The *When* step uses the two values from the context, instantiates the Math class and calls the Add() method with the two numbers, then it stores the result back in the context. Finally, the *Then* step takes the expected result from the feature file and it compares it to the result stored in the context. Of course, when we run the test, we will get a failure as we don't have the right code yet.

To run the test, right click it in the Test Explorer window and use the Run Selected Tests option:



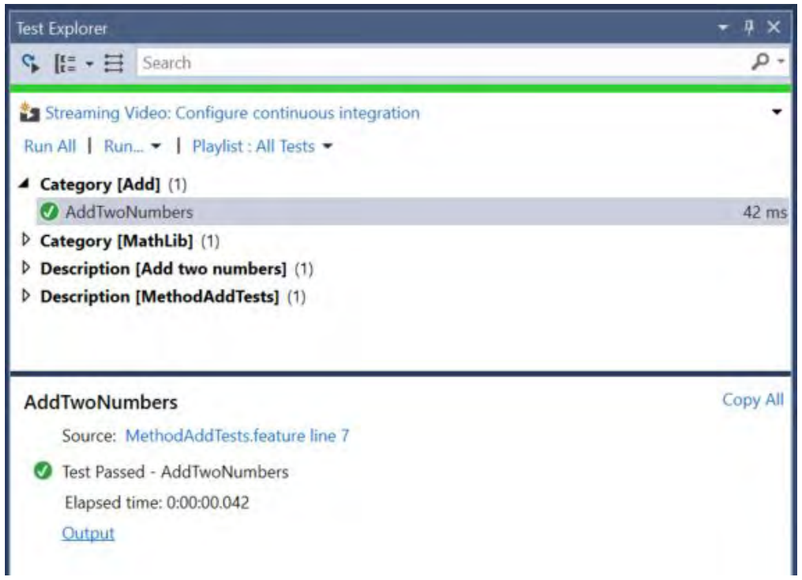
The result will look like this:



The result is as expected, so now let's fix the lib code and make it pass:

```
namespace MathLib
{
  public sealed class MathLibOps
  {
    public int Add(int firstNumber, int secondNumber)
    {
      return firstNumber + secondNumber;
    }
  }
}
```

Now, let's run the test again and we should see something a bit more cheerful:



Cool, so at this point, we should be fairly familiar with how it all hangs together.

The biggest question we need to ask now is this:

OK, this is all great, but how is this different from unit testing and what value does it actually provide? What am I getting?

I have a feature file, that's nice I suppose, but I could have easily written a unit test and be done with it. A Business Analyst is not going to care about my basic *Add two numbers* thing.

So, let's look at how we would implement something a bit more complex.

Specflow has a lot more features and we only touched on a few. A very nice feature is the ability to work with tables of data. This is important when the data is not as simple as a number. For example, imagine you have an object with five properties, which would make it more difficult to deal with, as we would now need five parameters, instead of one.
So, let's have a more serious project, let's implement an Access Framework for a website and this Access Framework will tell us if a user can perform various actions on our website.

A Complex Problem description

We have a website where people can visit and then search and apply for jobs. Restrictions will apply based on their membership type.

Membership types (Platinum, Gold, Silver, Free)
Platinum can search 50 times / day and apply 50 times / day.
Gold can search 15 times / day and apply to 15 jobs / day
Silver can search 10 times / day and apply to 10 jobs / day
Free can search 5 times / day and apply to 1 job / day.

What we need

1. We need to define the Users
2. We need to define the membership types
3. We need to define the restrictions for every membership type
4. We need a way to retrieve how many searches and applications a user has done every day.

The first three are configuration, the last one is user data. We could use this to define the ways in which we interact with the system.

Actual code

Let's create a class to represent the membership types. It could look like this:

```
namespace Models
{
  public sealed class MembershipTypeModel
  {
    public string MembershipTypeName { get; set; }
    public RestrictionModel Restriction { get; set; }

  }
}
```

The `RestrictionModel` class contains the max searches per day and the max applications per day:

```
namespace Models
{
  public sealed class RestrictionModel
  {
    public int MaxSearchesPerDay { get; set; }

    public int MaxApplicationsPerDay { get; set; }
  }
}
```

Next, we want a `UserModel`, which will hold the data we need for a user:


```
namespace Models
{
  public sealed class UserModel
  {
    public int ID { get; set; }
    public string Username { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string MembershipTypeName { get;set; }

    public UserUsageModel CurrentUsage { get; set; }
  }
}
```

The `UserUsageModel` will tell us how many searches and applications a user has already done that day:

```
namespace Models
{
  public sealed class UserUsageModel
  {
    public int CurrentSearchesCount { get; set; }

    public int CurrentApplicationsCount { get; set; }
  }
}
```

Finally, we want a class which will hold the results of the `AccessFramework` call:

```
namespace Models
{
  public sealed class AccessResultModel
  {
    public bool CanSearch { get; set; }

    public bool CanApply { get; set; }
  }
}
```

As you can see I kept this very simple, we don't want to get lost in implementation details.

We do want to see how BDD can help us with something which is not just a Hello World application.

So now we have our models, let's create a couple of interfaces, these will be responsible for the data retrieval part. First, the one dealing with generic configuration data:

```
using Models;
using System.Collections.Generic;

namespace Core
{
  public interface IConfigurationRetrieval
  {
    List<MembershipTypeModel> RetrieveMembershipTypes();
  }
}
```

The second one deals with user specific data:

```
using Models;

namespace Core {
  public interface IUserDataRetrieval {
    UserModel RetrieveUserDetails(string username);
  }
}
```

These two interfaces will become parameters to the AccessFrameworkAnalyser class and they will allow us to mock the data required for the tests:

```
using Core;
using Models;
using System;
using System.Linq;

namespace AccessFramework
{
  public sealed class AccessFrameworkAnalyser {
    IConfigurationRetrieval _configurationRetrieval;
    IUserDataRetrieval _userDataRetrieval;

    public AccessFrameworkAnalyser(IConfigurationRetrieval configurationRetrieval,
    IUserDataRetrieval userDataRetrieval) {
      if ( configurationRetrieval == null || userDataRetrieval == null) {
        throw new ArgumentNullException();
      }
      this._configurationRetrieval = configurationRetrieval;
      this._userDataRetrieval = userDataRetrieval;
    }

    public AccessResultModel DetermineAccessResults(string username) {
      if ( string.IsNullOrWhiteSpace(username)) {
        throw new ArgumentNullException();
      }

      var userData = this._userDataRetrieval.RetrieveUserDetails(username);
      var membershipTypes = this._configurationRetrieval.RetrieveMembershipTypes();

      var userMembership = membershipTypes.FirstOrDefault(p =>
      p.MembershipTypeName.Equals(userData.MembershipTypeName, StringComparison.
      OrdinalIgnoreCase));
      var result = new AccessResultModel();

      if (userMembership != null) {
        result.CanApply = userData.CurrentUsage.CurrentApplicationsCount <
        userMembership.Restriction.MaxApplicationsPerDay ? true : false;
        result.CanSearch = userData.CurrentUsage.CurrentSearchesCount <
        userMembership.Restriction.MaxSearchesPerDay ? true : false;
      }

      return result;
    }
  }
}
```

We don't do a lot here. We simply use Dependency Injection for our two interfaces, then populate the result by comparing how many searches and applications are available for the membership type of the selected user, against their current searches and applications.

Please note that we don't really care how this data is actually loaded, typically there would be an implementation for each interface which goes to a database, but for this example, we don't really care about that part.

All we need to know is that we will have a way of getting that data somehow and more than likely hook up the real implementations using an IOC of some kind in the actual UI project which needs real data. Since we don't really care for that part, we won't implement it, we will simply show some of the tests required.

Our feature file could look like this:

```
Feature: AccessFrameworkAnalyser
      In order to avoid unpaid searches and applications
      I want to be sure that the user can only search and apply up to their limit
based on their membership type

@AccessFrameworkAnalyser
Scenario: user with Free membership, already maxed searches and applications
      Given the membership types
      | MembershipTypeName | MaxSearchesPerDay | MaxApplicationsPerDay |
      | Free               | 5                 | 1                     |
      | Silver             | 10                | 10                    |
      | Gold               | 15                | 15                    |
      | Platinum           | 50                | 50                    |
      And a user
      | ID | Username | FirstName | LastName | MembershipTypeName | CurrentSearchesCount | CurrentApplicationsCount |
      | 10 | drakem   | Drake     | Moore    | Free               | 5                    | 1                        |

      When access result is required
      Then access result should be
      | CanSearch | CanApply |
      | false     | false    |

@AccessFrameworkAnalyser
Scenario: user with Free membership, still has both searches and applications
available
      Given the membership types
      | MembershipTypeName | MaxSearchesPerDay | MaxApplicationsPerDay |
      | Free               | 5                 | 1                     |
      | Silver             | 10                | 10                    |
      | Gold               | 15                | 15                    |
      | Platinum           | 50                | 50                    |
      And a user
      | ID | Username | FirstName | LastName | MembershipTypeName | CurrentSearchesCount | CurrentApplicationsCount |
      | 10 | drakem   | Drake     | Moore    | Free               | 4                    | 0                        |

      When access result is required
      Then access result should be
      | CanSearch | CanApply |
      | true      | true     |

@AccessFrameworkAnalyser
Scenario: user with Free membership, has searches but no applications available
      Given the membership types
      | MembershipTypeName | MaxSearchesPerDay | MaxApplicationsPerDay |
      | Free               | 5                 | 1                     |
      | Silver             | 10                | 10                    |
      | Gold               | 15                | 15                    |
      | Platinum           | 50                | 50                    |
      And a user
      | ID | Username | FirstName | LastName | MembershipTypeName | CurrentSearchesCount | CurrentApplicationsCount |
      | 10 | drakem   | Drake     | Moore    | Free               | 4                    | 1                        |

      When access result is required
      Then access result should be
      | CanSearch | CanApply |
      | true      | false    |
```

The pipes denote the Specflow way of dealing with tabular data.

The first row contains the headers, the rows after that contain the data. The important thing is to note how much data we setup and how readable it all is. Things are made simpler by the fact that there is no code here, nothing hides the actual data. At this point we can simply copy and paste a test, change the data and have another ready just like that.

The point is that a non-developer can do that just as well.

Let's look at the first scenario. As you can see, first we setup the membership types that we want to work with. Remember we don't care about real data, we care about the functionality and the business rules here and that's what we are testing. This makes it very easy to setup data any way we like.

The second step sets up the user and their existing counts of searches and applications.

And finally, we expect a certain result when the AccessFrameworkAnalyser class is used.

There are a few important things to mention here.

How do we load the tabular data in the steps code?

Here is an example which loads the data for the membership types:

```
private List<MembershipTypeModel> GetMembershipTypeModelsFromTable(Table table) {
  var results = new List<MembershipTypeModel>();
  foreach ( var row in table.Rows) {
    var model = new MembershipTypeModel();

    model.Restriction = new RestrictionModel();
    model.MembershipTypeName = row.ContainsKey(“MembershipTypeName”) ?
    row[“MembershipTypeName”] : string.Empty;

    if (row.ContainsKey(“MaxSearchesPerDay”)) {
      int maxSearchesPerDay = 0;

      if (int.TryParse(row[“MaxSearchesPerDay”], out maxSearchesPerDay)) {
        model.Restriction.MaxSearchesPerDay = maxSearchesPerDay;
      }
    }

    if (row.ContainsKey(“MaxApplicationsPerDay”)) {
      int maxApplicationsPerDay = 0;

      if (int.TryParse(row[“MaxApplicationsPerDay”], out maxApplicationsPerDay)) {
        model.Restriction.MaxApplicationsPerDay = maxApplicationsPerDay;
      }
    }
    results.Add(model);
  }

  return results;
}
```

It is a good idea to always check that a header exists before trying to load anything. This is very useful because depending on what you’re building, you don’t always need all the properties and objects at the same time. You might only need a couple properties for a few specific tests in which case you don’t need tables full of data. You just use the ones you need and ignore the rest and everything still works.

The actual step for loading the membership types now becomes very trivial:

```
[Given(@”the membership types”)]
  public void GivenTheMembershipTypes(Table table) {
    var membershipTypes = this.GetMembershipTypeModelsFromTable(table);
    ScenarioContext.Current.Add(“MembershipTypes”, membershipTypes);
  }
```

This is exactly like before - load the data > store in context > job done. Another interesting bit here is how we mock what we need.

I used NSubstitute for this and the code is quite simple:

```
[When(@”access result is required”)]
  public void WhenAccessResultIsRequired() {
    //data from context
    var membershipTypes = (List<MembershipTypeModel>)ScenarioContext.
    Current[“MembershipTypes”];
    var user = (UserModel)ScenarioContext.Current[“User”];
    //setup the mocks
    var configurationRetrieval = Substitute.For<IConfigurationRetrieval>();
    configurationRetrieval.RetrieveMembershipTypes().Returns(membershipTypes);

    var userDataRetrieval = Substitute.For<IUserDataRetrieval>();
    userDataRetrieval.RetrieveUserDetails(Arg.Any<string>()).Returns(user);

    //call to AccessFrameworkAnalyser
    var accessResult = new AccessFrameworkAnalyser(configurationRetrieval,
    userDataRetrieval).DetermineAccessResults(user.Username);
    ScenarioContext.Current.Add(“AccessResult”, accessResult);
  }
```

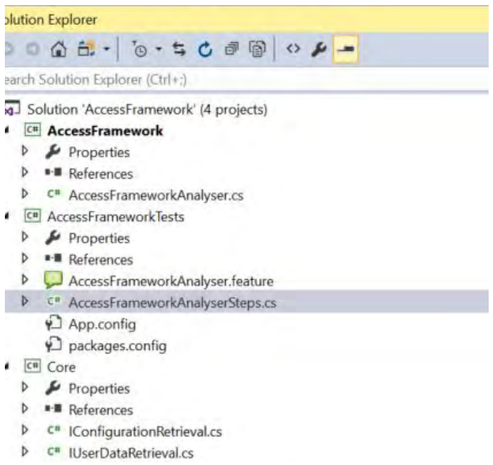
The initial data comes from steps which ran before this one, then we setup the mocks and finally call AccessFramework and store the result back in the context.

The final step, the actual assert, looks like this:

```
[Then(@”access result should be”)]
  public void ThenAccessResultShouldBe(Table table) {
    var expectedAccessResult = this.GetAccessResultFromTable(table);
    var accessResult = (AccessResultModel)ScenarioContext.Current[“AccessResult”];
    expectedAccessResult.ShouldBeEquivalentTo(accessResult);
  }
```

Here I used another NuGet package, **FluentAssertions**. This one allows me to compare objects without worrying about how many asserts I will need for every single property. I can still have just one assert.

The full code is attached, please have a look, it’s a lot easier to follow things in Visual Studio. Note the structure of the solution, everything is in a separate project, everything references exactly what it needs and nothing more:

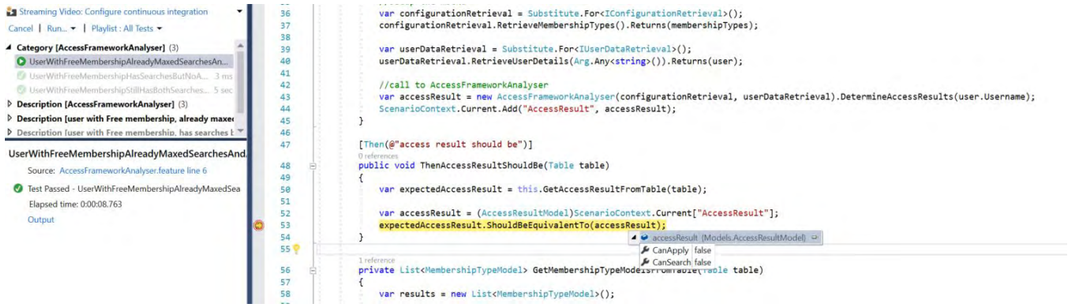


Hopefully by now you are starting to see the advantages of using BDD. The main point for me is that once the actual requirement is clear, we don’t need to look at code to work out what it does. All we need to do is look at the feature files.

It is a good idea to tag the scenarios with ticket numbers so you know which requirement each test is covering. This provides visibility to the business in terms of how much we have covered and what is left to do.

When a bug is encountered, it is a very good idea to write a test which replicates the bug and then fix it. This way you can be sure that a certain bug once fixed, it stays fixed.

If you need to debug a BDD test scenario you can simply set a breakpoint on a step and then right click the in the Test Explorer window, choose the “Debug Selected Tests” and off you go.



Downsides

So, you showed us the cake, what are the downsides of this approach?

Only one that I found so far and this is not a BDD issue specifically, but a tool issue.

Once you have several feature files and a healthy number of tests, you could potentially have quite a few steps. There is no easy way to tell when a step method is not used by any feature file. Codelens is not going to help here. You can’t tell if this particular step is called by ten scenarios. There are no counts anywhere so this could potentially mean that you could have orphan step methods.

Of course you can always delete one step method and then check if any feature file is affected but that could take a while, depending on how many feature files you have.

As I said this is not really a BDD issue, it is a Specflow issue and chances are it will only get better as more time passes.

For me, the benefits of using BDD greatly outweigh the issues with Specflow ■

Download the entire source code from GitHub at
bit.ly/dncm30-bdd

Andrei Dragotoniu
Author

Andrei Dragotoniu is a software developer from Southampton, UK. He currently works for DST Bluedoor as a Lead Backend Developer, working on a financial platform, getting involved in code standards, code reviews, helping junior developers. He is interested in architectural designs, building efficient APIs and writing testable code. In his spare time, he blogs about technical subjects at http://www.eidand.com.

Thanks to Yacoub Massad for reviewing this article.

Damir Arh

ERROR HANDLING in Large Projects

Effective error handling in any kind of an application plays an important role in providing a pleasant experience to the user, when unexpected failures occur. As our applications grow, we want to adopt a manageable strategy for handling errors in order to keep the user's experience consistent and more importantly, to provide us with means to troubleshoot and fix issues that occur.

Best Practices for Exception Handling

The idiomatic way to express error conditions in .NET framework is by throwing exceptions. In C#, we can handle them using the `try-catch-finally` statement:

```
try
{
  // code which can throw exceptions
}
catch
{
  // code executed only if exception was thrown
}
finally
{
  // code executed whether an exception was thrown or not
}
```

Whenever an exception is thrown inside the `try` block, the execution continues in the catch block. The `finally` block executes after the `try` block has successfully completed. It also executes when exiting the `catch` block, either successfully or with an exception.
In the `catch` block, we usually need information about the exception we are handling. To grab it, we use the following syntax:

```
catch (Exception e)
{
  // code can access exception details in variable e
}
```

The type used (`Exception` in our case), specifies which exceptions will be caught by the `catch` block (all in our case, as `Exception` is the base type of all exceptions).

Any exceptions that are not of the given type or its descendants, will fall through.

We can even add multiple `catch` blocks to a single `try` block. In this case, the exception will be caught by the first `catch` block with matching exception type:

```
catch (FileNotFoundException e)
{
  // code will only handle FileNotFoundException
}
catch (Exception e)
{
  // code will handle all the other exceptions
}
```

This allows us to handle different types of exceptions in different ways. We can recover from expected exceptions in a very specific way, for example:

- If a user selected a non-existing or invalid file, we can allow him to select a different file or cancel the action.

- If a network operation timed out, we can retry it or invite the user to check his network connectivity.

For remaining unexpected exceptions, e.g. a `NullReferenceExceptions` caused by a bug in the code, we can show the user a generic error message, giving him an option to report the error, or log the error automatically without user intervention.

It is also very important to avoid swallowing exceptions silently:

```
catch (Exception e)
{ }
```

Doing this is bad for both the user and the developer. The user might incorrectly assume that an action succeeded, where infact it silently failed or did not complete; whereas the developer will not get any information about the exception, unaware that he might need to fix something in the application.

Hiding errors silently is only appropriate in very specific scenarios, for example to catch exceptions thrown when attempting to log an error in the handler, for unexpected exceptions.

Even if we attempted to log this new error or retried logging the original error, there is a high probability that it would still fail.

Silently aborting is very likely the lesser evil in this case.

Centralized Exception Handling

When writing exception-handling code, it is important to do it in the right place. You might be tempted to do it as close to the origin of the exception as possible, e.g. at the level of each individual function:

```
void MyFunction()
{
  try
  {
    // actual function body
  }
  catch
  {
    // exception handling code
  }
}
```

This is often appropriate for exceptions which you can handle programmatically, without any user interaction. This is in cases when your application can fully recover from the exception and still successfully complete the requested operation, and therefore the user does not need to know that the exception occurred at all.

However, if the requested action failed because of the exception, the user needs to know about it.

In a desktop application, it would technically be possible to show an error dialog from the same method where the exception originally occurred, but in the worst case, this could result in a cascade of error dialogs, because the subsequently called functions might also fail – e.g. since they will not have the required data available:

```
void UpdatePersonEntity(PersonModel model)
{
    var person = GetPersonEntity(model.Id);
    ApplyChanges(person, model);
    Save(person);
}
```

Let us assume an unrecoverable exception is thrown inside `GetPersonEntity` (e.g., there is no PersonEntity with the given `Id`):

- `GetPersonEntity` will catch the exception and show an error dialog to the user.
- Because of the previous failure, `ApplyChanges` will fail to update the PersonEntity with the value of `null`, as returned by the first function. According to the policy of handling the exception where it happens, it will show a second error dialog to the user.
- Similarly, `Save` will also fail because of `PersonEntity` having `null` value, and will show a third error dialog in a row.

To avoid such situations, you should only handle unrecoverable exceptions and show error dialogs in functions directly invoked by a user action.

In a desktop application, these will typically be event handlers:

```
private void SaveButton_Click(object sender, RoutedEventArgs e)
{
  try
  {
    UpdatePersonEntity(model);
  }
  catch
  {
    // show error dialog
  }
}
```

In contrast, `UpdatePersonEntity` and any other functions called by it should not catch any exceptions that they cannot handle properly. The exception will then bubble up to the event handler, which will show only one error dialog to the user.

This also works well in a web application.

In an MVC application, for example, the only functions directly invoked by the user are action methods in controllers. In response to unhandled exceptions, these methods can redirect the user to an error page instead of the regular one.

To make the process of catching unhandled exceptions in entry functions simpler, .NET framework provides means for global exception handling. Although the details depend on the type of the application (desktop, web), the global exception handler is always called after the exception bubbles up from the outermost function as the last chance, to prevent the application from crashing.

In WPF, a global exception handler can be hooked up to the application's DispatcherUnhandledException event:

```
public partial class App : Application
{
  public App()
  {
```

```
    DispatcherUnhandledException += App_DispatcherUnhandledException;
  }

  private void App_DispatcherUnhandledException(object sender,
  DispatcherUnhandledExceptionEventArgs e)
  {
    var exception = e.Exception; // get exception
    // ...                          show the error to the user
    e.Handled = true;            // prevent the application from crashing
    Shutdown();                  // quit the application in a controlled way
  }
}
```

In ASP.NET, the global exception handler is convention based – a method named Application_Error in global.asax:

```
private void Application_Error(object sender, EventArgs e)
{
    var exception = Server.GetLastError(); // get exception
    Response.Redirect("error");     // show error page to user
}
```

What exactly should a global exception handler do?

From the user standpoint, it should display a friendly error dialog or error page with instructions on how to proceed, e.g. retry the action, restart the application, contact support, etc. For the developer, it is even more important to log the exception details for further analysis:

```
File.AppendAllText(logPath, exception.ToString());
```

Calling `ToString()` on an exception will return all its details, including the description and call stack in a text format. This is the minimum amount of information you want to log.

To make later analysis easier, you can include additional information, such as current time and any other useful information.

Using Exception Logging Libraries

Although writing errors to log files might seem simple, there are several challenges to address:

- Where should the files be located? Most applications do not have administrative privileges, therefore they cannot write to the installation directory.

- How to organize the log files? The application can log everything to a single log file, or use multiple log files based on date, origin of the error or some other criteria.

- How large may the log files grow? Your application log files should not occupy too much disk space. You should delete old log files based on their age or total log file size.

- Will the application write errors from multiple threads? Only one thread can access a file at a time. Multiple threads will need to synchronize access to the file or use separate files.

There are also alternatives to log files that might be more suitable in certain scenarios:

- Windows Event Log was designed for logging errors and other information from applications. It solves all of the above challenges, but requires dedicated tooling for accessing the log entries.

- If your application already uses a database, it could also write error logs to the database. If the error is caused by the database that's not accessible, the application will not be able to log that error to the database though.

You might not want to decide on the above choices in advance, but rather configure the behavior during the installation process.

Supporting all of that is not a trivial job.

Fortunately, this is a common requirement for many applications. Several dedicated libraries support all of the above and much more.

Popular Error Logging libraries for .NET

For .NET framework, the most popular logging libraries are probably log4net and NLog.

Although, there are of course differences between the two, the main concepts are quite similar.

In NLog, for example, you will typically create a static logger instance in every class that needs to write anything to the log:

```
private static Logger logger = LogManager.GetLogger("LoggerName");
```

To write to the log, you will simply call a method of that class:

```
logger.Error(exception.ToString());
```

As you have probably noticed, you have not yet specified where you want to log the errors. Instead of hardcoding this information in the application, you will put it in the NLog.config configuration file:

```
<?xml version="1.0" encoding="utf-8" ?>
<nlog xmlns=http://www.nlog-project.org/schemas/NLog.xsd
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <targets>
    <target name="logfile" xsi:type="File" fileName="errors.log"
      layout="${date:format=yyyyMMddHHmmss} ${message}" />
    </targets>

  <rules>
     <logger name="*" minlevel="Error" writeTo="logfile" />
  </rules>
</nlog>
```

The above configuration specifies that all errors will be logged to the errors.log file and accompanied with a timestamp.

However, you can easily specify a different type of target instead of a file or even add additional targets to log the errors to multiple locations.

With the `layout` attribute, you define which additional metadata you want to log along with the message. Different targets have different additional attributes to further control the logging process, e.g. archiving policy for log files and naming based on current date or other properties.

Using the rules section, you can even configure logging of different errors to different targets, based on the name of the logger that was used to emit the error.

Log levels can add additional flexibility to your logging.

Instead of only logging errors, you can log other information at different levels (warning, information, trace...) and use the `minlevel` attribute to specify which levels of log messages to write to the log and which to ignore.

You can log less information most of the time and selectively log more when troubleshooting a specific issue.

Editorial Note: *If you are doing error handling in ASP.NET applications, check out* Elmah.

Analyzing Production Log Files

Logging information about errors is only the first step.

Based on the logged information, we want to be able to detect any problems with the application and act on them, i.e. fix them to ensure that the application runs without errors in the future.

In order to do so effectively, the following tips can help us tremendously:

- Receive notifications when errors happen, so that we do not need to check all the log files manually. If errors are exceptional events, there is a high probability that we will not check the logs daily and will therefore not notice the errors as soon as we could.

- Aggregate similar errors in groups, so that we do not need to check each one of them individually. This becomes useful when a specific error starts occurring frequently and we have not fixed it yet since it can prevent us from missing a different error among all the similar ones.

As always, there are ready-made solutions for these problems available so that we do not need to develop them ourselves.

A common choice in .NET ecosystem is Application Insights. It is an Azure SaaS (Software as a Service) offering, with a free tier to get us started, and a usage based pricing model.

The product is tightly integrated into Visual Studio. There is a wizard available to add Application Insights telemetry to an existing project or to a newly created one. It will install the required NuGet package, create an accompanying resource in Azure and link the project to it with configuration entries.

This is enough for the application to send exception data to Application Insights. By adding custom code, more data can be sent to it. There is even an NLog custom target available for writing the logs to Application Insights.

Logged data can be analyzed in Azure Portal or inside Visual Studio.

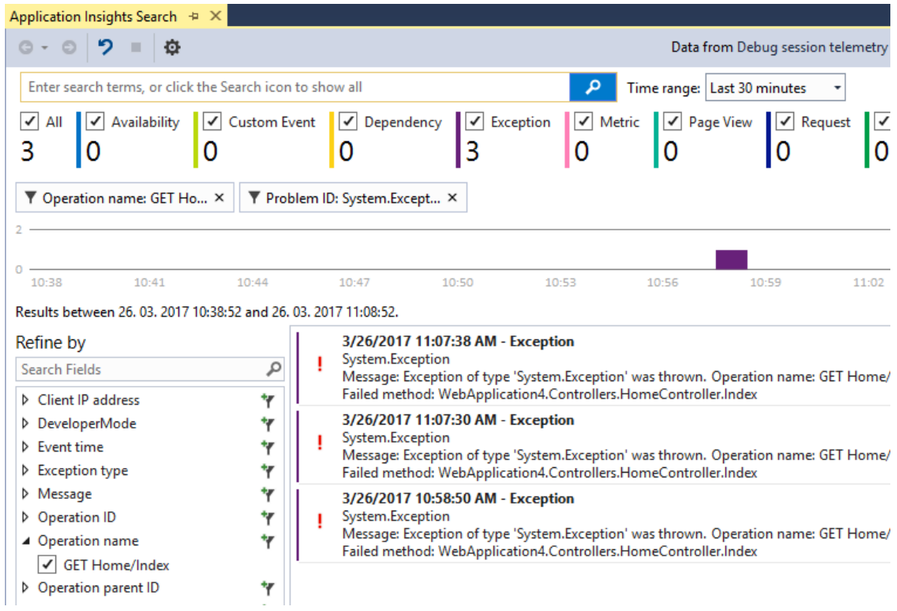


Figure 1: Application Insights search window in Visual Studio

If you want or are required to keep all the information on-premise, there is an open source alternative available: **OneTrueError**.

Before you can start using it, you first need to install the server application on your own machine. There is no wizard available for configuring OneTrueError in your application, but the process is simple:

- Install the NuGet package for your project type, e.g. `OneTrueError.Client.AspNet` for ASP.NET applications.

- Initialize automatic exception logging at application startup, e.g. in ASP.NET applications add the following code to your `Application_Start` method (replace the server URL, app key and shared secret with data from your server, of course):

```
var url = new Uri("http://yourServer/onetrueerror/");
OneTrue.Configuration.Credentials(url, "yourAppKey", "yourSharedSecret");
OneTrue.Configuration.CatchAspNetExceptions();
```

As with Application Insights, you can send more telemetry information to OneTrueError with custom logging code. You can inspect the logged information using the web application on your server.

Both Application Insights and OneTrueError, as well as other competitive products, solve another problem with log files: large modern applications which consist of multiple services (application server, web server, etc.), and are installed on multiple machines for reliability and load balancing.

Each service on each machine creates its own log and to get the full picture, data from all these logs needs to be consolidated in a single place.

By logging to a centralized server, all logs will automatically be collected there with information about its origin.

Error handling specifics for Mobile Applications

Until now, we focused on server side applications and partially desktop applications, however mobile applications are becoming an increasingly important part of a complete software product.

Of course, we would want to get similar error information for those applications as well, but there are some important differences in comparison to server and desktop applications that need to be considered:

- There is no way to easily retrieve data stored on mobile devices, not even in a corporate environment. The application needs to send the logs to a centralized location itself.

- You cannot count on mobile devices being always connected; therefore, the application cannot reliably report errors to a server as they happen. It needs to also store them locally and send them later when connectivity is restored.

Just like Application Insights and OneTrueError provide a complete solution for server application, there are dedicated products available for mobile applications.

Typically, they are not limited to error or crash reporting but also include support for usage metrics and (beta) application distribution.

Microsoft's companion to Application Insights for mobile and desktop applications is HockeyApp, but there are other alternatives available, such as Raygun and Crashlytics. They have similar feature set and mostly differentiate themselves by varying support for specific development platforms and pricing models.

Most of them require only a couple of lines of code to get started.

Conclusion:

Error handling is a broad topic, but also a very important part of application development. The actual project requirements will depend on various factors such as application size, deployment model, available budget, planned lifetime and others.

Nevertheless, it is a good idea to devise an error handling strategy for a project sooner rather than later. It will be easier to implement and will provide results sooner ■

Microsoft MVP Most Valuable Professional

Damir Arh
Author

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

Thanks to Yacoub Massad for reviewing this article.

Yacoub Massad

This article discusses the treatment of data in software applications. More specifically, it discusses runtime data encapsulation, behavior-only encapsulation, and treatment of state.

Data and Encapsulation in complex C# applications

Introduction

One important aspect of any programming paradigm is how it treats data.

In some interpretations of Object Oriented Programming (OOP), data is encapsulated inside objects. Such data is not manipulated/accessed directly by the clients of these objects, but instead, the objects expose some behavior that allows clients to indirectly query or manipulate the data.

In functional programming, data is not encapsulated or hidden inside objects. Instead, it is given as input to functions that produce other data as output.

In this sense, **functions are units of behavior.**

Such input data should be immutable, which means that it cannot be modified by the function acting on it. Also, having shared state that can be modified by different functions, is discouraged.

In procedural programming, we have procedures that are also units of behavior that act on data. But as opposed to functional programming, there is no emphasis on data immutability and statelessness. Procedures are free to modify both input data, as well as the state shared between procedures.

In large and complex applications, the ability to change software behavior easily and without breaking existing features **is not an easy task.**

Encapsulation is one mechanism that we can use to enhance software maintainability. However, there are more than one kind of encapsulations and it is important to understand how they affect software maintainability.

In this article, I am going to discuss different types of data and discuss some issues we might face if we try to encapsulate runtime data in complex application.

I will also talk about **behavior-only encapsulation** as an alternative to **data encapsulation**.

Finally, I will discuss *state* as a special kind of runtime data.

Types of data - Configuration and Runtime

When discussing how to deal with data, it is important to differentiate between different kinds of data.

One type of data is **configuration data**. Configuration data is used to customize applications.

Examples of such data are connection strings, user preferences, web service settings, etc. This data is mostly static. It is set in some configuration file, is read at application startup, and then used by the application.

One way to treat configuration data is to read it at the application startup and use it to determine which objects to create in the Composition Root. Also, we use it to inject primitive dependencies (such as connection strings) into objects.
Here is an example:

```
Settings settings = ReadSettings();

IProcessor processor =
  new IndexingProcessor(
    new Repository(settings.ConnectionString));

if (settings.DeleteFiles)
{
  processor =
    new CompositeProcessor(
      processor,
      new DeleteProcessor());
}
processor.Process();
```

In this example, we read the settings from the configuration file.

We also inject the connection string primitive dependency into the Repository class. In addition to this, based on the DeleteFiles setting, we decide whether to create a single IndexingProcessor or whether to create a CompositeProcessor that invokes both the IndexingProcessor and the DeleteProcessor objects.

Another type of data that is more relevant to the subject of this article is **runtime data**.

This is the data that the application does not know until after it completely starts. Examples of such data are user input, data queried from the database, data obtained via a web service call, data produced by processing other data, etc.

Please note that in order to change the configurations, we might need some runtime data to do this.

For example, imagine an application that allows you to change its settings. The data entered in the settings window would be runtime data. The same data is called configuration data when it is used at application startup to configure the application.

Treatment of runtime data

One way to treat runtime data is to encapsulate it inside objects.

Here is an example of runtime data that is encapsulated inside an object:

```
public class Document
{
  private readonly string identifier;
  private readonly string content;

  public Document(string identifier, string content)
  {
    this.identifier = identifier;
    this.content = content;
  }

  public string[] GetDistinctWords()
  {
    return content.Split(‘ ‘).Distinct().ToArray();
  }

  public void Print()
  {
     //...
  }

  public void Archive()
  {
     //...
  }

  public Document Translate(Language language)
  {
     //...
  }
```


This Document class represents a text document.

It has an identifier field to uniquely identify it, and a content field to hold its contents. The document identifier and its contents are run time data. They might have been read from the file system or the database at runtime.

The class contains four methods that allow the consumers to interact with it in meaningful ways.

The GetDistinctWords method for example is responsible for returning the list of distinct words in the document. The consumers of this class do not get to access the content field (the contents of the document) directly, but instead they use methods like GetDistinctWords to gain access to the contents indirectly.

This is **data encapsulation**.

Actually, this is both data and behavior encapsulation. The consumers cannot access the data directly, and also they don’t know and don’t care how the methods work internally.

The main argument for such design is that it easily allows future changes to the way the data is represented inside the object.

For example, we can change the type of the content field from a string to a string array representing the words of the document.

Or we can store a complex object graph that represents the different paragraphs of the document each with detailed information about the text like color and style, etc. We can even omit this field entirely and use the identity field to query a database or a web service every time a consumer asks the object something.

Such changes would require the implementation of the class methods (e.g. GetDistinctWords) to change, **but it wouldn’t require us to change code in other classes.**

This is the power of data encapsulation.

Let’s dig deeper.

Let’s first consider how a consumer of the Document class would get an instance of this class.

It wouldn’t make sense for the consumer to construct the Document class itself, because that would mean that the consumer already has access to the internal data of the document needed for the construction of the Document class, and that would break the whole data encapsulation concept.

The only way to hide these details from the consumer is to have the consumer get already built instances of the Document class. For example, a Document object might be passed as a parameter to a method in the consuming object. Or the consumer can invoke a repository/factory to create Document instances.

Consider the example of a DocumentProcessor class that processes documents. It has a dependency on a DocumentSource object that it uses to get the next set of Document objects to process, similar to this:

```
Document[] documents = documentSource.GetNextSetOfDocuments();
```

This way, the consumer (the DocumentProcessor class) does not need to know about the internal structure

of the Document class. We moved such knowledge to the DocumentSource class.

The DocumentSource class would for example, connect to some database, execute some database query, create instances of the Document class, and return them.

```
public class DocumentSource
{
  private readonly int documentsToTakeAtOnce;

  public DocumentSource(int documentsToTakeAtOnce)
  {
    this.documentsToTakeAtOnce = documentsToTakeAtOnce;
  }
  public Document[] GetNextSetOfDocuments()
  {
    using (var context = new DatabaseContext())
    {
       return
         context.Documents
            .Where(x => !x.Processed)
            .Take(documentsToTakeAtOnce)
            .AsEnumerable()
            .Select(x => new Document(x.Identifier, x.Content))
            .ToArray();
    }
  }
}
```

Here is how the composition code looks in the Composition Root:

```
var documentProcessor =
  new DocumentProcessor(
    new DocumentSource(documentsToTakeAtOnce: 100));
```

Now let's consider how we can modify the behavior of the Document class.

Let's first classify the behavior changes that we might want to make, into three categories:

1. Changes to handle cross cutting concerns such as performance monitoring, logging, error handling, etc.

2. Changes to the internals of existing behavior in the Document class. For example, changing the way the print functionality works or adding new supported languages for the translate functionality or changing the way we archive documents.

3. Changes to add new behavior to the Document class. For example, we might want to add a method to convert the document into a different format.

Changes to handle cross cutting concerns

Let's start by considering how we can handle cross cutting concerns in the Document class.

Let's say that we would like to record the time it takes to extract the distinct words of the document, that is, the time it takes for the GetDistinctWords() method to complete.

Our first option is to add the measurement code inside the GetDistinctWords() method in the Document class. This however will make the code in the method less readable, so we should avoid it.

It is very important to avoid adding such code to existing methods. Adding such code to existing methods will make the code in the methods very long, tangled, and thus hard to maintain.

Also, hardcoding such code into existing methods will make it hard to turn it on or off selectively. For example, what if we want to record performance only for some of the Document objects based on the source of such documents (e.g. database1 or database2)?

A better approach is to use a decorator.

To enable decoration, we need to create an interface for the Document class and make the DocumentSource.GetNextSetOfDocuments() method return an array of IDocument instead of Document.

```
public IDocument[] GetNextSetOfDocuments()
{
    //..
}
```

Next, we need to create a decorator for IDocument to record the time like this:

```
public class PerformanceRecordingDocumentDecorator : IDocument
{
  private readonly IDocument document;
    public PerformanceRecordingDocumentDecorator(IDocument document)
    {
        this.document = document;
    }

    public string[] GetDistinctWords()
    {
        Stopwatch sw = Stopwatch.StartNew();
        var distinctWords = document.GetDistinctWords();
        RecordTime(sw.Elapsed);
        return distinctWords;
    }
    private void RecordTime(TimeSpan elapsed)
    {
        //..
    }
}
```

Next, in the factory (the DocumentSource class), we decorate all Document objects using this decorator before we return them like this:

```
public IDocument[] GetNextSetOfDocuments()
{
  using (var context = new DatabaseContext())
  {
    return
      context.Documents
        .Where(x => !x.Processed)
        .Take(documentsToTakeAtOnce)
        .AsEnumerable()
        .Select(x =>
          new PerformanceRecordingDocumentDecorator(
```

```
            new Document(x.Identifier, x.Content)))
        .Cast<IDocument>()
        .ToArray();
  }
}
```

If we add more decorators to the Document object, the factory would look like this:

```
public IDocument[] GetNextSetOfDocuments()
{
  using (var context = new DatabaseContext())
  {
    return
      context.Documents
        .Where(x => !x.Processed)
        .Take(documentsToTakeAtOnce)
        .AsEnumerable()
        .Select(x =>
          new YetAnotherDocumentDecorator(
            new AnotherDocumentDecorator(
              new PerformanceRecordingDocumentDecorator(
                new Document(x.Identifier, x.Content)))))
        .Cast<IDocument>()
        .ToArray();
    }
}
```

The problem with this factory code is that code responsible for reading documents from the database, is tangled with code responsible of installing the right decorators on the read documents.

Also, we might want to decorate the document objects differently in different cases. For example, we might want to log actions on documents obtained from database 1, but not documents obtained from database 2.

Since we return IDocument instances instead of concrete Document instances, we can use any of the good things we get from programming to an interface to fix this problem.

For example, we can move the decoration logic of Document objects into a dependency like this:

```
public class DocumentSource : IDocumentSource
{
  private readonly IDocumentDecorationApplier documentDecorationApplier;
  //..
  public IDocument[] GetNextSetOfDocuments()
  {
    using (var context = new DatabaseContext())
    {
      return
        context.Documents
          .Where(x => !x.Processed)
          .Take(documentsToTakeAtOnce)
          .AsEnumerable()
          .Select(x =>
             documentDecorationApplier
               .ApplyDecoration(
                  new Document(x.Identifier, x.Content)))
          .ToArray();

    }
```



```
  }
}
public interface IDocumentDecorationApplier
{
  IDocument ApplyDecoration(IDocument document);
}

public class DocumentDecorationApplier : IDocumentDecorationApplier
{
  public IDocument ApplyDecoration(IDocument document)
  {
    return
      new YetAnotherDocumentDecorator(
        new AnotherDocumentDecorator(
          new PerformanceRecordingDocumentDecorator(document)));
  }
}
```

The DocumentSource now creates the Document objects and gives them to the IDocumentDecorationApplier dependency to apply any decorations. That would allow us to create a DocumentSource object that applies some decorations to Document objects, as well as another DocumentSource object that applies different decorations by injecting a different implementation of the IDocumentDecorationApplier interface.

To make this even more flexible, we can make the DocumentDecorationApplier accept a Func<IDocument,IDocument> in the constructor so that we can easily configure decoration from the Composition Root like this:

```
public class DocumentDecorationApplier : IDocumentDecorationApplier
{
  private readonly Func<IDocument, IDocument> decorationApplierFunction;
  //..

  public IDocument ApplyDecoration(IDocument document)
  {
    return decorationApplierFunction(document);
  }
}
var documentProcessor1 =
  new DocumentProcessor(
    new DocumentSource(
      new DocumentDecorationApplier(
        document =>
          new YetAnotherDocumentDecorator(
            new AnotherDocumentDecorator(document))),
      documentsToTakeAtOnce: 100,
      connectionString: connectionString1));

var documentProcessor2 =
  new DocumentProcessor(
    new DocumentSource(
      new DocumentDecorationApplier(
        document =>
          new AnotherDocumentDecorator(
            new PerformanceRecordingDocumentDecorator(document))),
      documentsToTakeAtOnce: 100,
      connectionString: connectionString2));
```

So far, the cost of data encapsulation seems reasonable, let's dig deeper.

Changes to modify/vary existing behavior

What if there are three different ways to print a document? Five different ways to translate a document? Two different ways to archive a document?

Different consumers of the Document class might want to print, translate, or archive the documents in different ways.

How can we configure the Document class to behave differently?

One way is to create a new Document class for each permutation a consumer might want. If there are 10 consumers who want different configurations of the Document class, then we need to create 10 Document **classes**.

A lot of the code in these classes would be duplicate.

One way to fix this problem is to create a single Document class and make it configurable by using constructor parameters like this:

```
public enum PrintAppoach {/*..*/}

public enum TranslateAppoach {/*..*/}

public enum ArchiveAppoach {/*..*/}

public class Document : IDocument
{
  public Document(
    string identifier, string content,
    PrintAppoach printAppoach, TranslateAppoach translateAppoach, ArchiveAppoach
    archiveApproach)
    {
    //..
  }
}
```

Internally, for each behavior (e.g. printing), the Document class would have a switch/if statement that will decide how to act based on the setting.

For example:

```
private void Print()
{
  if (printAppoach == PrintAppoach.Approach1)
  {
    //..
  }
  else if (printAppoach == PrintAppoach.Approach2)
  {
    //..
  }
}
```

One issue with this approach is that the Document class will get very big.

Another issue is that we need to have similar parameters in the constructor of the factory (e.g. the DocumentSource class) because it will need them when constructing Document objects.

What if we want to configure the Document object to try many different translation approaches before it fails? What if we want to control the order in which these different translation approaches execute?

What kind of complex parameters can we use to configure the Document class with all these different nuances? How complex is the Document class and its factory object going to get?

An alternative approach to solving the problem of many Document classes is to delegate each behavior to some interface. We will have a single Document class that looks like this:

```
public class Document : IDocument
{
  private readonly string identifier;
  private readonly string content;

  private readonly IPrinter printer;
  private readonly ITranslator translator;
  private readonly IArchiver archiver;
  public Document(string identifier, string content, IPrinter printer, ITranslator
  translator, IArchiver archiver)
  {
    this.identifier = identifier;
    this.content = content;
    this.printer = printer;
    this.translator = translator;
    this.archiver = archiver;
  }

  public void Print()
  {
    printer.Print(content);
  }

  public IDocument Translate(Language language)
  {
    var translatedContent = translator.Translate(content, language);
    //..
  }

  public void Archive()
  {
    archiver.Archive(identifier, content);
  //..
  }
}
```

Now, we can use all the good things we get from programming to an interface, to inject whatever translation strategy (for example) into the Document object, with any nuances that we need (e.g. retrying with different translation strategies in easily configurable order).

Although this would result in a single Document class, we have introduced more problems.

First, one could argue that data encapsulation, which is what we are going out of our way to preserve, is

broken.

The IPrinter service for example, takes a string representing the document content as a parameter. What if we changed the internal representation of the data to represent a sophisticated document that understands different fonts, different colors, etc? We would need to change the IPrinter interface to meet this new data representation.

Another problem is that these dependencies (IPrinter, ITranslator, IArchiver) are dependencies that need to be managed by the factory object (the DocumentSource) as well. They need to be injected into the factory object, just to allow the factory object to inject them into the Document objects it create.

Changes to add new behavior

As the software grows, new behavior will be added to the Document class. For example, we might add a method to convert the document to a different format.

To preserve data encapsulation, such methods need to exist in the Document class itself and cannot be moved to other classes.

The consequence of this is that if we need to decorate one method, say the GetDistinctWords() method, then our decorators will also need to implement all the other methods (which would simply delegate to the decorated instance) just to satisfy the interface.

Any time we add a new behavior to the Document class, all the decorators break and we need to go visit all the decorators and implement the new method.

The problem of complex object construction

There is yet another problem, now inside the Translate() method.

Regardless of how we translate the content, we need to create a new Document object to encapsulate the result of the translation. We cannot simply create and return a new Document object because we need to account for the many possible decorators that were used to decorate the current Document object (or maybe even different ones).

One way to fix this is to inject an IDocumentDecorationApplier dependency into the Document class and making sure that we use it to apply the decorators any time we create a new Document object:

```
public class Document : IDocument
{
  private readonly IDocumentDecorationApplier documentDecorationApplier;
//..
  public Document(
  string identifier, string content,
  IPrinter printer, ITranslator translator, IArchiver archiver,
    IDocumentDecorationApplier documentDecorationApplier)
  {
    //..
    this.documentDecorationApplier = documentDecorationApplier;
  }
//..
  public IDocument Translate(Language language)
  {
    string translatedContent = ...;
    var document = new Document(identifier, translatedContent, printer, translator,
    archiver, documentDecorationApplier);
    return documentDecorationApplier.ApplyDecoration(document);
  }//..
}
```

When the DocumentSource object creates the Document objects, it would inject the IDocumentDecorationApplier dependency into these objects.

In some sense, the Document class is now a factory of Document objects. It has the ability to create new Document objects, and thus it needs to be complex enough to do this.

This becomes even more complex when the Document class needs to construct other types of objects. For example, a Paragraph object, a Header object, etc. We would need to inject into the Document class (and its factory) some dependencies to help with the construction of such objects.

Another approach to treating run time data: Behavior-only Encapsulation

In the previous section, we explored the option of encapsulating runtime data. The advantage of such an approach is that we get to hide the internal representation of the data from consumers and therefore we get to change such representations easily.

However, there is a price that we pay for such benefits.

Another approach to treat runtime data is to separate it from behavior.

In this approach, data passes through **units of behavior** (e.g. functions or procedures) which can access the internals of such data easily.

For example, we can create a translation unit of behavior that has the following interface:

```
public interface IDocumentTranslator  {
  Document Translate(Document document, Language language);
}
```

The Document object in this case is a simple data object. Here is how it looks like:

```
public class Document {
  public string Identifier { get; }
  public string Content { get; }

  public Document(string identifier, string content)
  {
    Identifier = identifier;
    Content = content;
  }
}
```

An implementation of the IDocumentTranslator interface can simply access the content of the Document object as it wishes.

It is recommended however, that our data objects remain immutable. That is, once a data object is constructed, we cannot modify its contents. If we need to modify a Document, we would need to create another Document object and pass to its constructor, a modified version of the content.

When data objects are immutable, the code becomes easier to reason about.

We are still encapsulating. But instead of encapsulating data, we encapsulate behavior. What this means is that the consumer of the translation unit does not know *how* the translation functionality is implemented.

This is met by having consumers depend on interfaces instead of concrete types. The implementations of the interfaces are hidden from the consumer (and hence encapsulated).

Composition in this approach is much easier, we wouldn't need complex factories like the ones we saw earlier.

This is true because all runtime data exist in data objects that don't contain any behavior and thus they don't need the complex customization/configuration needed when encapsulating data.

Now, we can simply use the C# "new" keyword to create any data objects.

Also, because units of behavior don't encapsulate any runtime data, everything about them is known at application startup. Therefore, the construction/configuration/customization of units of behavior (objects that contain behavior) can be done immediately at application startup in the Composition Root.

For a complete example of this alternative approach, see the Object Composition with SOLID article.

A special kind of runtime data: State

A special kind of runtime data is state.

State is the data whose value changes from time to time.

For example, consider the case where we need to count the number of documents processed over the lifetime of an application for statistical reasons. The number of documents represents runtime data that starts with the value 0 when we run the application, and increases as the application processes more documents.

State can have different scopes.

Methods can define local variables and change the value of these variables multiple times before they complete. These variables hold state, but this state has a lifetime that starts when the method is invoked, and ends when it completes.

Therefore, it's easy to deal with this kind of state.

The number of processed documents example, is an example of state that has a larger scope. The lifetime of such a state is the lifetime of the application. It is this kind of state that requires special handling.

We cannot simply pass this kind of state through method parameters.

One reason is that the lifetime of this kind of a state is larger than the lifetime of a single method invocation or a chain of multiple method invocations.

How can we deal with this kind of state?

Some procedural programming

In procedural programming, we can define a global variable to hold some state (e.g. number of processed documents), and procedures are free to read or update such state.

This approach however has some problems.

1. Having a global variable that is modified from many places (e.g. procedures) means that a single procedure cannot be reasoned about, without understanding how other procedures modify/read this shared state.

2. Using a global variable directly means that the connection between the procedures that share the same state is not obvious.

We cannot do a lot to fix problem one. Sometimes we need state that different parts of the system need to share.

However, we can mitigate the second problem by making the dependency between the shared state and the parts of the system that access it more explicit.

Controlling shared state: The State Holder pattern

We can create an "object" whose sole responsibility is to hold some state. Here is an example of such an object:

```
public class StateHolder<T> : IStateHolder<T> {
  private T state;

  public void SetValue(T value) => state = value;

  public T GetValue() => state;
}

public interface IStateGetter<T> {
  T GetValue();
}

public interface IStateSetter<T> {
  void SetValue(T value);
}
```

```
public interface IStateHolder<T> : IStateGetter<T>, IStateSetter<T>
{

}
```

Any unit that requires to read/update the state will have a dependency on IStateGetter<T>, IStateSetter<T>, or IStateHolder<T> depending on the type of access it requires.

For example, the following DocumentProcessor class has read/write access to a state holder object that is used to store the number of processed documents:

```
public class DocumentProcessor
{
  private readonly IStateHolder<int> numberOfDocumentsStateHolder;
  //..

  private void ProcessDocuments(Document[] documents)
  {
    //..
    numberOfDocumentsStateHolder.SetValue(numberOfDocumentsStateHolder.GetValue() +
    documents.Length);
  }
}
```

..and this StatisticsService class has only read access to such state:

```
public class StatisticsService
{
  private readonly IStateGetter<int> numberOfDocumentsStateGetter;
  //..

  public Report GenerateReport()
  {
    var numberOfDocuments = numberOfDocumentsStateGetter.GetValue();
    //..
  }
}
```

This statistics service could for example be called via WCF from a monitoring web site to monitor the health of a document processing application.

Here is how we compose these objects in the Composition Root:

```
var numberOfProcessedDocumentsStateHolder =
  new StateHolder<int>();
//..
var documentProcessor =
  new DocumentProcessor(numberOfProcessedDocumentsStateHolder);
//..
var statisticsService =
  new StatisticsService(numberOfProcessedDocumentsStateHolder);
```

This will allow us to easily track which units of behavior have which type of access to such a larger-scope state.

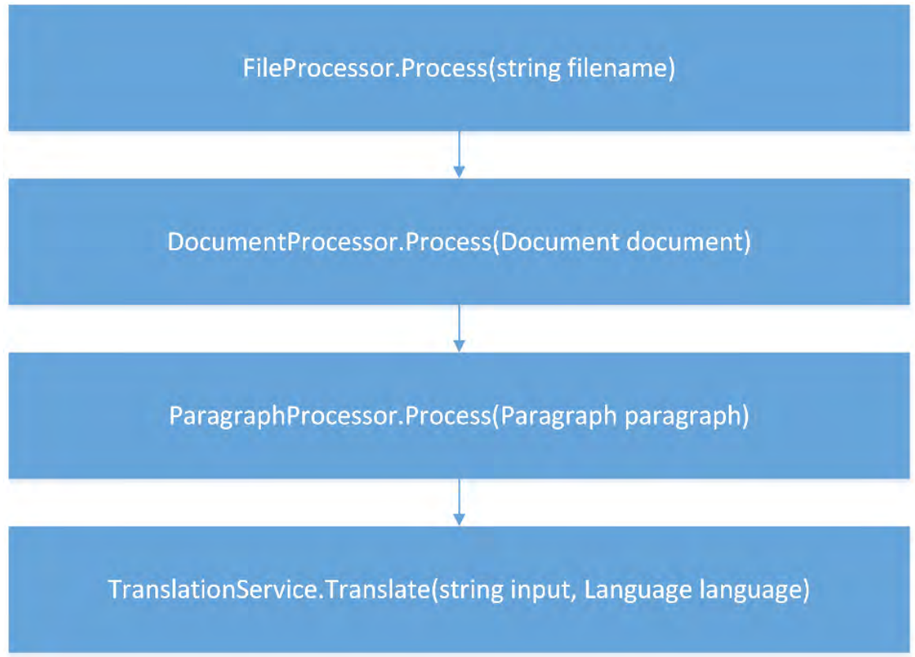
For more information about creating Composition Roots that allow developers to easily navigate and understand the structure of their applications, see the Clean Composition Roots with Pure Dependency Injection article.

Another example of state

Let's consider another example of state which has a scope larger than a single method, but smaller than the lifetime of the application or even the lifetime of a single request. A request in this context could mean a WCF/Web API request, but it could also mean a set of actions triggered by some event. E.g. a timer interval elapsing, the user interacting with the application, etc.

Imagine an application that translates documents.

A single request in this application starts with a file. A Document object is created from this file and then the Document is parsed into distinct paragraphs. Each paragraph is then processed by invoking some translation service.



The filename is the first piece of runtime data that we encounter in this request. The FileProcessor unit receives this data and uses it to read the file from the file system, create a Document object, and then pass it to the DocumentProcessor.

The Document object that the DocumentProcessor receives is also runtime data. The DocumentProcessor parses the Document into multiple paragraphs and for each paragraph it invokes the ParagraphProcessor unit to process it.

The ParagraphProcessor would then extract the text out of the paragraph and invoke a TranslationService to translate it to some other language.

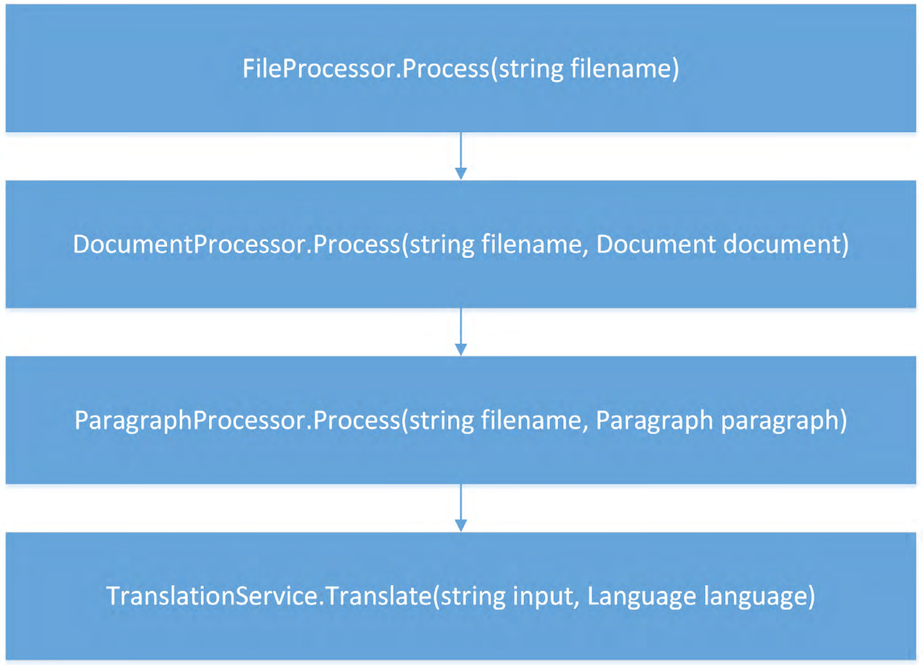
In real applications, the process will be more complex, but I use this simple example to explain the idea of

this section.

Now, let's say that a new requirement asks that we log the time a translation request is sent to the translation service, along with the name of the file.

Now, these two pieces of information can be obtained from two different places in code. The time of the request is known by the ParagraphProcessor because it is the one that invokes the translation service. And the name of the file is known only to the FileProcessor.

One solution is to change the signatures of DocumentProcessor.Process and ParagraphProcessor.Process to add a filename argument like this:



This solution introduces no state. It simply sends the name of the file to two additional units so that the ParagraphProcessor has access to the filename.

To understand the problem with this approach, imagine that in the near future, we are asked to process documents that are read from the database, not from the file system.

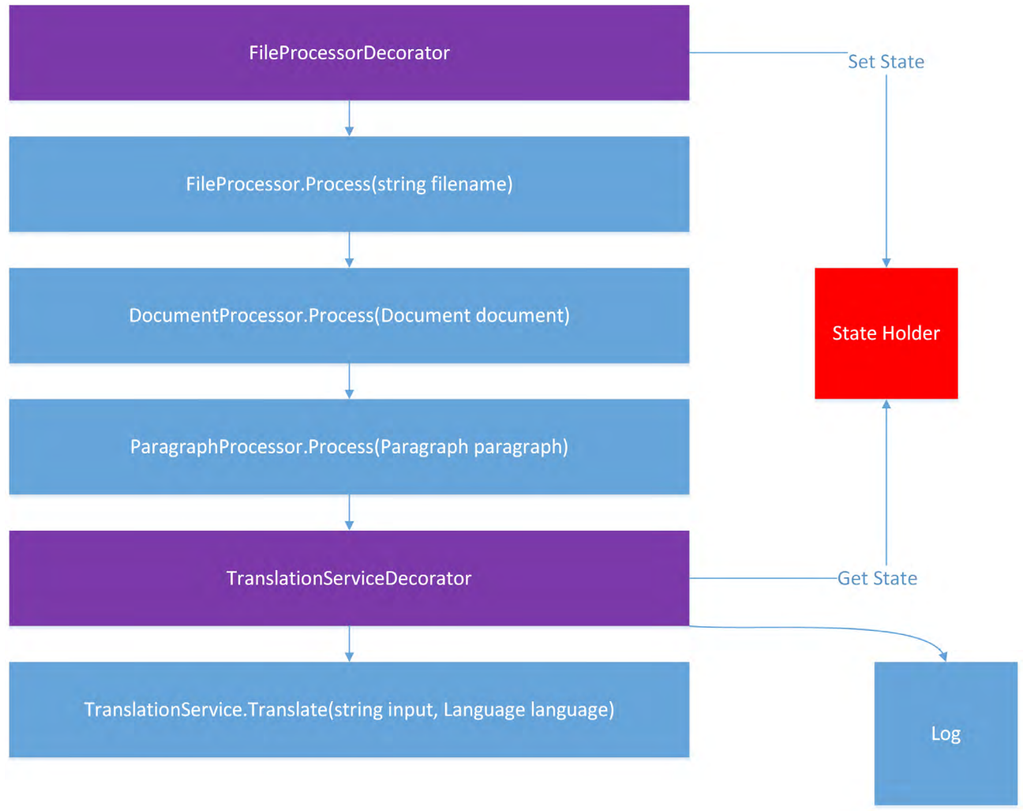
We should be able to reuse the DocumentProcessor and the ParagraphProcessor classes. By including the filename as an argument, we are explicitly saying that these two classes can only be used to process Documents/Paragraphs that are originally read from a file and therefore cannot be used by Documents/Paragraphs that originally come from someplace else.

If we insist to use them, then what should we pass for the filename parameter in case the Document comes from the database? The row id?

The problem with this approach is that the abstractions implemented by the DocumentProcessor class and the ParagraphProcessor class become leaky abstractions. Since we are using dependency injection, the FileProcessor class actually depends on IDocumentProcessor, not DocumentProcessor. Having IDocumentProcessor with a method that has a filename parameter means that this abstraction (IDocumentProcessor) leaks an implementation detail (that the implementation of this interface works with Documents that originate from files).

To help us solve this problem, we can introduce state.

We can create a decorator of IFileProcessor that stores the filename into a state holder. We also create a decorator of ITranslationService that will read the filename from the state holder and log the time and the filename.



Here is how this graph can be composed in the Composition Root:

```
var currentFilenameStateHolder = new StateHolder<string>();
var fileProcessor =
  new FileProcessorDecorator(
    currentFilenameStateHolder,
      new FileProcessor(
        new DocumentProcessor(
          new ParagraphProcessor(
            new TranslationServiceDecorator(
              currentFilenameStateHolder,
                new Logger(),
                  new TranslationServiceProxy(webServiceUrl))))));
```

Conclusion

In this article, I have discussed different types of data and discussed different ways to deal with runtime data. I have shown that although encapsulating runtime data makes it easier to later change the internal representation of data, there is a cost associated with it.

In many cases, **a better approach to handle runtime data is to pass it as arguments to units of behavior.**

A special kind of runtime data is state.

State cannot be simply passed through method arguments because its lifetime is larger than a single method or even a single request. To store state, we can use the State Holder pattern as a replacement of using global variables. The difference is that with the State Holder pattern, the dependency between the state and the units in the system that access it becomes explicit and therefore it becomes easier to understand such units ■

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

Craig Berntson

REVISITING SOLID

Last year I finished up a long series on SOLID principles. Here at DNC Magazine, we've had a couple of readers send questions about SOLID. This issue's column is my attempt to answer the questions. Before getting to the questions, I want to summarize SOLID principles.

S Single Responsibility Principle (SRP) – A class should have only one reason to be changed.

O Open-Closed Principle (OCP) – A class should be open to extension but closed to modification

L Liskov Substitution Principle (LSP) – You should be able to replace a class with a subclass without the calling code knowing about the change

I Interface Segregation Principle (ISP) – Many specific interfaces are better than a single, all-encompassing interface

D Dependency Inversion Principle (DIP) – Code should depend upon abstractions, not concrete implementations

Now, on to the questions.

SOLID for Beginners

The first question is from Eric, who writes, "***I am a Computer Science Graduate who has learnt C#. Can you explain me SOLID (like for beginners)***".

Great question, Eric. Coming out of school with a shiny degree, one can quickly realize they were taught to write code, but not taught to write software. Schools typically teach how to do things in Java, JavaScript, C#, Python and other languages but neglect the important things such as DevOps, Agile, SOLID, and other practices.

SOLID is often difficult to understand for new developers, but yet one of the most important concepts you can understand. Where I work at Quicken Loans, we typically ask about SOLID when interviewing hiring candidates and some of them cannot give a satisfactory answer.

The primary thing to ask yourself when writing code is, "Does this solution solve the problem?" If you can say yes, the code is correct. However, it may not answer other questions you need to keep in mind. These are things like:

- Does the code have adequate safe guards against invalid data?
- Is the code readable by a human?
- Is it maintainable?
- Can the code be easily extended when needs change?
- Is this code secure?
- Does the code do too much? Should I break it down into smaller pieces?

Many of these questions can be answered by applying SOLID. Let's review SOLID.
SOLID has been around for years. The concepts were first brought together by Robert "Uncle Bob" Martin. They are a set of concepts that help us create better code. You shouldn't think that code will comply with every SOLID principle the moment it leaves your fingers and enters the computer via the keyboard. Some principles are more applicable to interfaces, some to classes. And, once the code is originally written, you will typically refactor it before you are satisfied that is as close to SOLID as you can get it.

When learning SOLID, learn one principle at a time, learn it well, then move on to the next one.

Single Responsibility Principle

Single Responsibility Principle (SRP) answers many of the above questions. First, it can identify if code does too much. The code should do one thing and one thing only and do it well. And if code does one thing, it's easier to understand when you read it. It's also less likely that you'll have to modify the code and if you do, odds are that unexpected side effects won't creep in.

But, it isn't always clear what one thing code should be doing. Here's an example. For many years, developers have been creating data classes. These would be used to Create, Read, Update, and Delete (CRUD) data. One day, someone proposed that Create, Update, and Delete are all similar in that they change data in the database. Read is a different animal. It doesn't change data, but simply tells us what the data is, therefore, following SRP, it should be in a different class. After all, they said, it's far more likely we'll have to change code to read the data for different purposes than to modify the data. We may need a single row, multiple rows, summarized data, data sorted in different ways, totals, subtotals, counts, groupings, etc. This led to a concept called Command Query Responsibility Separation (CQRS).
Others counteracted this by saying that in the end, it's all data access of some kind, so it all belongs in the same class.

Which one is correct?

The answer is, both are correct. CQRS may add unneeded complexity.

As a general rule, if you have to modify code in two or more places, either in the same class or multiple classes, to apply a single modification (fix a bug or add new functionality) that section of code MIGHT be a candidate for multiple classes, hence making it compliant with SRP. What if it's only a single line of code that you move to a new class? Is the additional complexity worth it? Only you and your team can decide this.

Open-Closed Principle

The Open-Closed Principle is all about adding new functionality without modifying the existing code or even assembly. The reason behind this is that every time you modify code, you run the risk of adding bugs to the existing functionality.

Think about the String class in .NET. There are many operations that you can perform on a string. You have string.Length, string.Split, string.Substring and many others. What if there is a need to reverse all the characters in a string, so that instead of "Dot Net Curry", you wanted, "yrruC teN, toD"? If the string class is modified, there is a chance (albeit very small) that existing code can get accidentally changed. Instead, create an extension method that lives in a different assembly. BAM! Done.

Liskov Substitution Principle

Now, let's look at an oddly named principle, Liskov Substitution Principle (LSP), named after the person who first proposed it. What it says is that we should be able to replace a class we're using with a subclass, and the code keeps working.

A common example here is a logging class. Think of all the places you can write a log. It could be to a database, a file on disk, a web service, use the Windows Event Log, and others. The file on disk can take many forms. It could be plain text, XML, JSON, HTML, and many other formats. But the code that calls the logging methods should not have to call different methods for different ways to log. It should always call a similarly named method for all of them and a specific instance of the logging class handles the details. You'd have a different class for each way you support for writing a log.

Interface Segregation Principle

Have you ever looked at the different interfaces in the List<T> class in .Net? I have. List<T> inherits from eight different interfaces. Here's the definition from MSDN.

```
[SerializableAttribute]
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,
        IEnumerable, IList, ICollection, IReadOnlyList<T>, IReadOnlyCollection<T>
```

Why does it do this? The simple reason is to comply with the fourth SOLID principle, the Interface Segregation Principle (ISP). Think about the functionality defined in each of those interfaces? Now imagine if all that functionality was contained in a single interface. How complex would that be? How many different things would this single, combined interface do? It would be crazy! It would be impossible to properly maintain.

And then imagine if you had a class that didn't need to implement IReadOnlyList<T>. How would you implement that? Basically, you'd have code that didn't do anything. Why do you have code that does nothing?

The solution is to have many specialized interfaces so that consumers depend only on what they need.

Dependency Inversion Principle

Let's return to the logging example above. How do you tell a specific class which type of logging it should use? That's the job of the Dependency Inversion Principle or as some call it Dependency Injection Principle. When you instantiate the class, you pass in an instance of the class you want to use through Constructor Injection. The following code shows how to do this.

```
public interface ILogger
{
  void WriteLog(string message);
}

public class DatabaseLogger : ILogger
{
  public void WriteLog(string message)
  {
    // Format message
    // Build parameter
    // Open connection
    // Send command
    // Close connection
  }
}

public class TextFileLogger : ILogger
{
  public void WriteLog(string message)
  {
    // Format message
    // Open text file
    // Write message
    // Close text file
  }
}

public class Main
{
  public Main()
  {
    var processor = new Processor(new TextFileLogger());
  }
}

public class Processor
{
  private readonly ILogger _logger;
  public Processor(ILogger logger)
  {
    _logger = logger;
  }

  public void RunProcessing()
  {
    try
    {
      // Do the processing
```

```
    }
    catch (Exception ex)
    {
      _logger.WriteLog(ex.Message);
    }
  }
}
```

At the top, the interface is defined followed by two classes, DatabaseLogger and TextFile logger that implement the interface. In the Main class, an instance of the TextFileLogger is created and passed to the constructor of the Processor class. This is Dependency Injection. The Processor class depends on an instance of ILogger and instead of creating it every time, the instance is injected into the constructor. This also loosely couples the specific logger instance from the Processor class.

The instance of the logger is then used inside the Try/Catch block. If you want to change the logger type, all you need to do is change the calling program so that you instantiate DatabaseLogger and send it instead.

Final thoughts on learning SOLID

And there you have a simplified explanation of SOLID. I encourage you to look at my earlier columns on this topic. All are linked to above. You may benefit from finding a senior level developer on your team and using him as a mentor. Mention to him that you're interested in learning SOLID. That will help the learning curve and give you someone to rely on to help you in your early career. Now, onto the second question.

To SOLID or not to SOLID? That is the Question

With apologies to William Shakespeare, let's look at the second question.

Adam asks, "***When should I use SOLID and when should I not use SOLID?***"

Well, Adam, the easy answer is always use SOLID, but as I discussed above, it doesn't always make sense. Sometimes it adds complexity. The real answer is well, more complex.

Overly complex applications are an ugly side of software development. I've seen them over the years. I've tried to fix them. Sometimes the complexity is not in the code, but in the architectural directions taken in the application. Bad architecture is often impossible to fix without a complete rewrite.

While SOLID is mostly a coding concept, there is an architectural aspect. Should you split something into multiple classes? How many interfaces should you use? This list goes on.

Sometimes, the application of SOLID comes along as you design what the class is doing. See the above discussion on List<T>. As the designers were defining the functionality of the class, the interfaces began to take shape and they learned which ones needed to be implemented.

In my work, once I have the code working correctly, I may do additional refactoring to follow a specific Design Pattern or to make the code more self-documenting. This is also a good time to look at applying some SOLID. Asking questions like, "Does this class do too much and if it does, will I add complexity with multiple classes?" need to be asked. You may need to discuss with your team members as it may not be obvious.

Take the DatabaseLogger implementation above. There are five steps listed in the WriteLog method. Should each of those be a separate method? Should it be relegated to a different class? Or, should it all stay in that method? The answer to each of these questions is, "Maybe."

You may also find that the same SOLID questions come into play when modifying a class, either by adding new functionality (most likely SOLID comes into play) or fixing a bug (maybe SOLID should be used).

There's another situation when you may not use SOLID and that's when you're time restrained. Alright, I accept that *every* project is time constrained, but there are times you just need to get the code out and other times when you can "do it right". If you're seriously time constrained, adding the technical debt of not using SOLID when you should may be acceptable. But as with all technical debt, you should have a plan to pay it off.

One final place where SOLID won't come into play is with prototypes and throw-away code. These are both places where you don't plan to put the code into production and it will be removed soon. Don't spend the time worrying about SOLID here.

Now, having said all this, as you gain experience, you start to see how to use SOLID earlier in the architectural and code writing process. Not all the time. You'll still have programming puzzles that require you to refactor to SOLID, but because of accumulated experience, you should be able to complete the refactoring more quickly and with better results.

Conclusion

As with many software development concepts, SOLID has a place in your toolbox. It has a learning curve as steep as there is, but it's one to master, not only learning the principles but when to use them and when not to use them. And by understanding SOLID and using it properly your software can be green, lush, and growing.

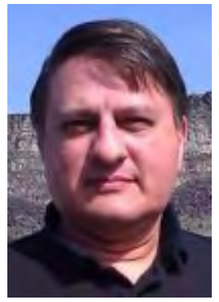
About Software Gardening

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

Microsoft MVP Most Valuable Professional

Craig Berntson
Author

Craig Berntson is a Senior Software Engineer at Quicken Loans. Craig has spoken at developer events across North America and Europe for over 20 years. He is the coauthor of Continuous Integration in .NET available from Manning Publishing. He has been a Microsoft MVP since 1996. Craig lives in Salt Lake City, Utah. Email: dnc@craigberntson.com Twitter: @craigber



Thanks to Yacoub Massad for reviewing this article.

Francesco Abbruzzese

Implementing Multi-Platform Libraries with TypeScript

TypeScript can target several JavaScript versions and several module export conventions (AMD, CommonJS, System, ES6, or simply globals + namespaces).

However, in general, each platform requires different configuration settings and a different module structure. As a result, there is no built-in option to simultaneously generate JavaScript and declaration files (d.ts) from a single TypeScript source file simultaneously for several platforms.

More specifically AMD and CommonJS conventions are covered simultaneously by selecting the UMD module target; but globals and ES6, require different settings.

In this how-to article, I will show **how to organize your TypeScript project to target UMD, globals and ES6 platforms simultaneously**, all with the help of a simple Node.js script.

This script will do the following:

- Pre-process TypeScript sources to adapt them to each target platform

- Orchestrate the whole transformations pipeline that yields the final declaration files (d.ts) JavaScript files, as well as minified JavaScript files, for all platforms.

Editorial Note: *This article assumes familiarity with TypeScript. If you are new to TypeScript, check this tutorial (bit.ly/dnc-ts-tut) to get started.*

Multi-platform libraries

Usually, modern JavaScript libraries offer their functionalities through a unique access-object (such as the jQuery, or the underscore objects for instance) and expose this access-object in a way that is compatible with several module organization standards.

More specifically, they autodetect their environment and either expose themselves as AMD / CommonJS modules or expose this access-object in a global variable.

While CommonJS ensures compatibility with Node.js and Browserify, AMD ensures compatibility with the remaining loaders (RequireJs and SystemJs) and module bundlers (SystemJs build tool, RequireJs optimization tool, WebPack).

Although the JavaScript community considers the usage of global variables as obsolete, it still remains the preferred technique for web developers that rely mainly on server side techniques and use just a small portion of JavaScript in their web pages.

As a result, compatibility with the global variable pattern is a "must". Unluckily, also the way the global variable exposes the library access-object follows two different patterns: first, where in famous free libraries, the access-object is inserted directly in a global variable (for instance `jQuery` for the jQuery library), and the second in proprietary libraries, all installed access-objects are exposed as properties of a root "company" object (something like myCompany.myLibrary).

Multi-platform compatibility and auto-detection are achieved with a JavaScript wrapper that surrounds the core code. When the library is bundled with other JavaScript code, this wrapper code might interfere with some bundler optimizations (mainly with unused code removal).

ES6 module organization doesn't need any wrapper since the imported and exported objects, are defined with declarations. Moreover, its declarative organization helps bundlers in their optimizations.

Thus, if our library is likely to be bundled with a bundler like WebPack 2 that supports and takes advantage of ES6 module organization, it is a good idea to also deliver a version based on ES6 module organization.

We can also do this if we are targeting a lower JavaScript version instead of ES6, but since all import/export statements are removed by the bundler, it merges all modules into a unique "chunk".

Implementing libraries with TypeScript

The way TypeScript is translated into JavaScript depends on the settings of the JSON configuration placed in the root of the TypeScript project.

In particular, the way modules are organized is controlled by the "module" property, whose value must be:

- "none" for a global variable based structure
- "UMD" covers both CommonJS and AMD
- "es6" for the ES6 modules structure.

Thus, having different configuration files and different versions of the library is the only way to ensure compatibility with several platforms.

Unfortunately, just accepting the idea of several compilation steps and several versions of the same library is not enough to solve the problem, since the TypeScript sources must also be slightly changed to adapt them to different platforms.

In fact, the global variable version of the library should contain TypeScript namespaces and should look like this:

```
/// <reference path="./teams.ts" />
namespace company{
```

```
  export namespace organization{
    //core code
  }
}
```

..where the reference tags declare other modules that the current module depends on, and the nested namespaces ensure a global name like “companyName.libraryName”.

The ES6 version instead would look like this:

```
import {Person, Team} from “./teams”
//core code
```

..where the import statements declare all the objects contained in other modules and used by the current module.

Finally, the UMD version should be similar to the ES6 one, but at least one module would also contain some re-exports of names imported from other modules. In fact, usually AMD and CommonJS libraries are accessed through the single access-object returned by a unique “main module” that also exposes all the needed objects contained in other modules of the library.

So atleast for the main module, we should have something like this:

```
import {Person, Team} from “./teams”
export {Person, Team}
//core code
```

As a conclusion, we need a preprocessing of our sources. Luckily, the examples above show that this preprocessing is quite simple and consists of:

- Extracting the “core code” from each source module
- Generating three different TypeScript files from each source by adding a different wrapper around the same “core code”.

So we may now proceed as follows:

1. We develop our modules in JavaScript using any of the styles outlined above (UMD, ES6, or global variable). The best candidate is ES6 since the code doesn’t depend on the way we want to deploy our library.

2. We mark the core code of each module by enclosing it between the two symbols `///{` and `///}` as outlined below:

```
import {Person, Team} from “./teams”
///{
      //core code is here
///}
```

3. We then define three wrappers for each source module, one for each destination platform. In each wrapper we mark a place where to put the “core code” with the symbol `///{}`, as outlined below:

```
namespace company{
  export namespace organization{
  ///{}  }
}
```


Once we have everything in place, that is our sources, the wrappers, and the three different TypeScript configuration files; we may launch a Node.js script that performs the needed preprocessing and compiles all preprocessed files yielding both JavaScript files, TypeScript declaration files (.d.ts), and the map files (.map).

The same script can also minify each JavaScript file with UglifyJS.

An example project

Let us analyze everything in detail with a simple example.

Preparing the project

As a first step make sure you have installed a recent version of Node.js, and a recent version of the TypeScript compiler globally (the tsc command must be available in any directory).

In the remainder of the article, I will use Visual Studio Code, but if you want, you may execute all the steps with a different IDE, since I’ll not use any specific feature of Visual Studio Code.

Create a folder for the project, and then use Visual Studio Code to open it.

If not visible, show the Visual Studio Code integrated terminal (you may also open a command prompt in the project directory).

We will need UglifyJS, so let us install it in our folder with npm:

```
> npm install uglify-js
```

Finally add a “src” folder for our sources

Defining the TypeScript configuration files

We need three different TypeScript configuration files, one for each target platform.

Since they differ just by a few settings, they may inherit most of their settings from a common configuration file.

Add a new file to the project and name it “tsconfig.base.json”, this file will contain the common configuration:

```
{
  “compilerOptions”: {
    “moduleResolution”:”classic”,
    “noImplicitAny”: true,
    “removeComments”: true,
    “preserveConstEnums”: true,
    “sourceMap”: true,
    “declaration”:true,
    “target”:”es3”,
    “strictNullChecks”: false
  }
```

```
}
```

Now we may define the ES6 configuration as “tsconfig.es6.json”:

```
{
  “extends”: “./tsconfig.base”,
  “compilerOptions”:{
    “declarationDir”: “./dest/es6”,
    “outDir”:”./dest/es6”,
    “module”: “es6”
  },
  “include”:[
    “proc/es6/**/*.*”
  ]
}
```

Where the “include” property specifies the source files, the “module” property specifies the kind of modules organization, and finally “declarationDir” and “outDir” are the folders where to place the output TypeScript declaration files, and JavaScript files respectively.

The UMD configuration as “tsconfig.umd.json”:

```
{
  “extends”: “./tsconfig.base”,
  “compilerOptions”:{
    “declarationDir”: “./dest/umd”,
    “outDir”:”./dest/umd”,
    “module”: “umd”
  },
  “include”:[
    “proc/umd/**/*.*”
  ]
}
```

In this case, the module organization is “umd”, and both input and output folders are different.

Finally, the global variable configuration as “tsconfig.global.json” is as folows:

```
{
  “extends”: “./tsconfig.base”,
  “compilerOptions”:{
    “declarationDir”: “./dest/global”,
    “outDir”:”./dest/global”,
    “module”: “none”
  },
  “include”:[
    “proc/global/**/*.*”
  ]
}
```

Here “module” “none” specifies that there are no modules at all, since we use namespaces instead.

The three different JSON files differ in the value of the “module” property, and in the location where they take their sources and output the result of the compilation. All TypeScript sources are taken form a different subfolder of the “proc” folder.
In fact, our preprocessor will take all files from the “src” folder and from each of them, it will create three new modified versions - one in the “proc/global” folder, another in the “proc/es6” folder and the last one in the “proc/umd” folder.

Analogously, all compilation results will be placed in the dest/global”, “dest /es6”, and “dest /umd” folders.

Some example TypeScript modules

Let’s add a module containing some simple classes of “person” and “team” within our “src” folder.

Let us call this file: “teams.ts”:

```
///{
export class Person
{
  name: string;
  surname: string;
  role: string;
}
export class Team
{
  name: string;
  members: Person[]
  constructor(name: string, ...members: Person[]){
    this.name = name;
    this.members=members;
  }
  add(x: Person){
    this.members.push(x);
  }
}
///}
```

In this simple module, the “core code” enclosed within the ///{ and ///} symbols is the whole code.

Let’s also define also the “projects.ts” module that contains the logic to assign teams and persons to projects:

```
import {Person, Team} from “./teams”
///{
export class Project
{
  name: string;
  description: string;
  team: Team|null;

  constructor(name: string, description: string)
  {
    this.name=name;
    this.description=description;
  }
  assignTo(team: Team)
  {
    this.team=team;
  }
  addHumanResource(x: Person): boolean
  {
    if(!this.team) return false;
```

```
    this.team.add(x);
    return true;
  }
}
///}
```

Since we need to import classes from the previous module in this module, we have added an import statement that is not part of the "core code" (all import, re-export and namespace definitions are excluded from the core code).

Defining all wrappers

Wrappers are defined in files placed in the "src" directory. We give them the same name as the module they refer to but with different extensions. Namely extensions like .es6.add, .umd.add, and .global.add.

The extensions are named .es6.add, .umd.add, and .global.add.

Shown here are all the wrappers for the teams module.

teams.global.add:

```
namespace company{
  export namespace organization{
    ///{}
  }
}
```

teams.umd.add, and teams.es6.add are equal and contain just the placeholder for the "core code", since they have no import statement:

```
///{}
```

projects.es6.add instead imports the classes from the teams module, so its content is as follows:

```
import {Person, Team} from "./teams"
///{}
```

Since the projects module is also the main module of the whole library, projects.umd.add not only imports the classes defined in the projects module, but also re-exports them, so its content is:

```
import {Person, Team} from "./teams"
export {Person, Team}
///{}
```

projects.global.add does not contain just namespaces definitions like teams.global.add but also a "reference declaration" and a variable definition for each object imported from the teams module:

```
/// <reference path="./teams.ts" />
namespace company{
  export namespace organization{
    let Team = organization.Team;
    let Person = organization.Person;
  }
}
```


Variable declarations must be provided in a ".global.add" wrapper each time the source module contains imports, otherwise all references to the imported objects in the "core code" would be undefined.

Building the project

I used a simple Node.js script to perform the needed preprocessing, to call the TypeScript compiler, and to minimize all files. However, you can just take the pre-processing part of the script and organize all other tasks with Gulp or Grunt.

I placed this script in the root of the project and called it "go.js" but you may use a different name such as "build.js". In the header of the file, add all the needed *requires*:

```
var exec = require('child_process').exec;
var fs = require("fs");
var path = require('path');
var UglifyJS = require("uglify-js");
```

"fs" and "path" are needed for files and directory processing, "Uglify-js" (which was installed when preparing the project) for the minimization of the target JavaScript files and finally "child_process" to launch the "tsc" command (TypeScript compilation).

Then comes a few settings:

```
var src="./src";
var dest = "./proc";
var fdest ="./dest";
var dirs = [
  'global',
  'umd',
  'es6'
];
var except = [
  'require','exports','module', 'export', 'default'
];
```

They contain all directory names. You may change them, but then you also have to coherently change wrappers extensions, the names of the TypeScript configuration files and also all directories referred in this configuration files.

The "except" variable is passed to UglifyJS and contains all "reserved words" that must not be mangled. If needed, you may add more names, but please do not remove the existing names, since they are used in the UMD wrapper created by the TypeScript compiler.

Before starting the actual processing, I define a couple of utilities to extract "core code", and to collect input file names:

```
var extract = function(x){
  var startIndex = x.indexOf("///{");
  if(startIndex < 0) return x;
  else startIndex=startIndex+4;
  var endIndex = x.lastIndexOf("///}");
  if(endIndex>0) return x.slice(startIndex, endIndex);
  else return x.slice(startIndex);
```

```
}
```

This extracts the "core code" from a string that contain a whole source file.

```
var walk = function(directoryName, ext) {
  var res=  [];
  ext=ext || ".ts";
  var files=fs.readdirSync(directoryName);
  files.forEach(function(file) {
    let fullPath = path.join(directoryName,file);
    let f= fs.statSync(fullPath);
      if (f.isDirectory()) {
        res=res.concat(walk(fullPath));
      } else {
      if(fullPath.match(ext+"$"))
        res.push(fullPath)
      }
    });
  return res;
};
```

The "walk" function recursively collects the name of all files with extension "ext" contained in the directory "directory name" and in all its descendant directories.

With the above utilities defined the pre-processing stage is easily implemented:

```
var toProcess=walk(src);
if(!fs.existsSync(dest)) fs.mkdirSync(dest);
dirs.forEach(function(dir){
  let outDir = path.join(dest, dir);
  if(!fs.existsSync(outDir)) fs.mkdirSync(outDir);
  toProcess.forEach(function(file){
    let toAdd=file.substr(0, file.length-3)+"."+dir+".add";
    let outFile=path.join(dest, dir, path.basename(file));

    if(fs.existsSync(toAdd)){
      let input = extract(fs.readFileSync(file, 'utf8'));
      fs.writeFileSync(outFile,
        fs.readFileSync(toAdd, 'utf8').replace("///{}", input));
    }
    else{
      fs.writeFileSync(outFile,
          fs.readFileSync(file));
    }
  })
});
```

The code collects all source files with the "walk" function, then creates the root directory for all pre-processed files if it doesn't exist.

The outermost "forEach" loops through all platform specific directories. It creates them if they do not exist, and then the inner "forEach" loops through all source files.

Each source file is read into a string, its core is extracted by the "extract" function and placed in the "content area" of each wrapper file with a string replace.

The final chunk of code asynchronously invokes the TypeScript compiler and minimizes all compilation results, in the associated callback:

```
if(!fs.existsSync(fdest)) fs.mkdirSync(fdest);
for(let i=0; i<dirs.length; i++)
{
  let config = 'tsc -p tsconfig.'+dirs[i]+'.json';
  let fOutDir = path.join(fdest, dirs[i]);
  if(!fs.existsSync(fOutDir)) fs.mkdirSync(fOutDir);
  console.log("start "+dirs[i]);
  exec(config, function(error, stdout, stderr) {
    console.log(stdout);
    console.error(stderr);
    if(dirs[i] != 'es6') {
      let files = walk(fOutDir, ".js");
      files.forEach(function(file){
        let baseName=file.substr(0, file.length-3);
        if(baseName.match(".min$")) return;
        let inMap = file+".map";
        if(!fs.existsSync(inMap)) inMap=undefined;
        let outFile = baseName+".min.js";
        let outMap = baseName+".min.js.map";
        let res=UglifyJS.minify(file, {
          outSourceMap: path.basename(outMap),
          outFileName: path.basename(outFile),
          inSourceMap: inMap,
          except:except
        });
        fs.writeFileSync(outFile, res.code);
        fs.writeFileSync(outMap, res.map);
      });
    }
    console.log("finished "+dirs[i]);
  });
}
```

The TypeScript compiler is invoked three times, once for each typescript file defined. The name of the configuration file is passed as a command line argument. Before invoking the compiler, all needed destination directories are created if they do not exist, yet.

In the callback, messages and possible errors are displayed in the console. For each input file that the compiler creates: the corresponding JavaScript file, a TypeScript declaration file, and a source map is generated.

Finally, UglifyJS is invoked and the source map created by the compiler is passed to it as input map.

This way, the final source map of the minimized code will refer to the original TypeScript sources instead of the JavaScript intermediate file.

Testing the Node.js Script

You may test the Node.js script on the example project. In the Visual Studio Code integrated terminal (or in any command window rooted in the project folder), launch the command:

```
> node go
```

You should see the new “proc” and “dest” directories each containing three subdirectories: “es6”, “umd” and “global”.

The “proc” directory contains the pre-processed TypeScript files, while each “dest” folder subdirectory contains all minimization and compilation results namely : “projects.d.ts”, “projects.js”, “projects.js.map”, “projects.min.js”, and “projects.min.js.map”.

You may test the umd version of the example library with the help of the Node.js debugger integrated in Visual Studio Code. Follow these steps:

Create a “test.js” file in the root of the project and fill it with the code below:

```
let projects=require(“./dest/umd/projects.js”)

var project = new projects.Project(“test”, “test description”);
project.assignTo(
  new projects.Team(“TypeScript team”,
  new projects.Person(“Francesco”, “Abbruzzese”, “team leader”)));
var team = project.team;
```

We will use the projects module to access all the classes, since in the umd version of the library, it acts as “main module” and exports all objects exposed by the library.

Put a breakpoint on the first line of code. Go to the Visual Studio Code debugger view and click the play button to start debugging. Execute the code step-by-step with the help of the “F10” key, and inspect the value of all variables by hovering over them with the mouse.

This way, you may verify that all classes are created properly.

Conclusion:

In summary, several platforms may be targeted simultaneously by a single TypeScript library by generating several versions of each module (one for each target platform), thanks to the simple pre-processing technique described. The Node.js build-script proposed is just an example of implementation of the proposed pre-processing technique you may use as a starting point for your custom scripts

Download the entire source code from GitHub at
bit.ly/dncm30-multits

Microsoft MVP Most Valuable Professional

Francesco Abbruzzese
Author

Francesco Abbruzzese, develops Asp.net Mvc applications, and offers consultancy services since the beginning of this technology. He is the author of the famous Mvc Controls Toolkit, and his company offers tools, UI controls and services for Asp.net Mvc. He moved from decision support systems for banks and financial institutions, to the Video Games arena, and finally started his .Net adventure with the first .Net release.

Thanks to Ravi Kiran and Suprotim Agarwal for reviewing this article.

Daniel Jimenez Garcia

Understanding ASP.NET Core Model Binding: Custom Binders

If you have used previous versions of the ASP.NET MVC framework, you might have probably come across the model binding process and even created a few custom model binders. Similarly those who have used ASP.NET WebAPI, know that the binding process is different from that of MVC.

With the new .NET Core, things have changed with both the MVC and WebApi merged into the one ASP.Net Core framework.

So how does model binding now work in the new Core framework?

Interestingly, it behaves closer to the previous WebAPI framework.

This means there are scenarios that will take you by surprise if you were accustomed to the previous MVC framework.

For e.g. Submitting a JSON within the request body is particularly different now when compared to the previous MVC framework and requires you to understand these changes.

In this article, I will describe **how Model Binding in ASP.NET Core works**, how does it compare against the previous versions of the framework and how you can customize it with your custom binders.

Model Binding in ASP.NET Core

A look back at the old ASP.Net model binding process

In the previous version of MVC, the model binding process was based on model binders and value providers. When binding a model:

- The framework would iterate through the collection of `ModelBinderProviders` until one of them returned a non-null `IModelBinder` instance.
- The matched binder would have access to the request and the value providers, which basically extracted and formatted data from the request.
- By default, a `DefaultModelBinder` class was used along with value providers that extracted data from the query string, form data, route values and even from the parsed JSON body.

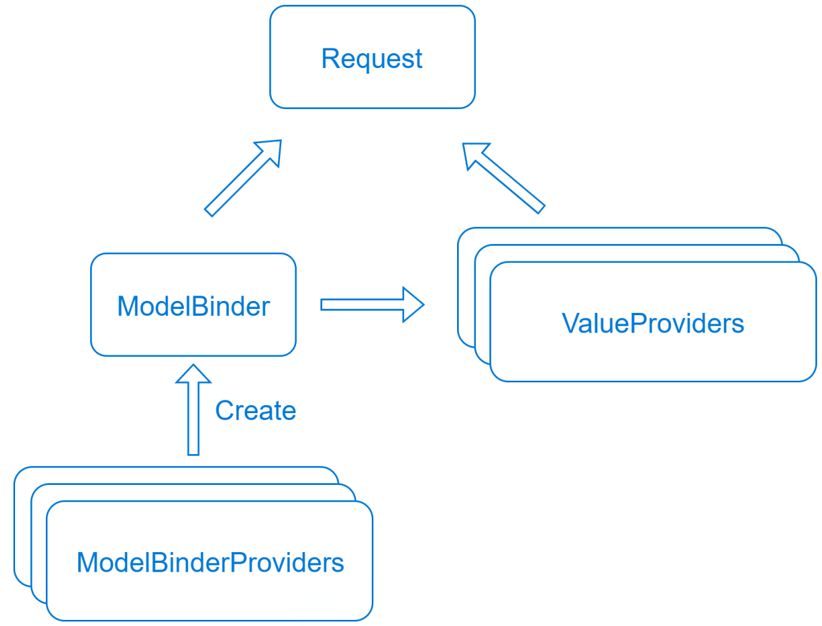


Figure 1, a simplified view of the old MVC model binding

If you wanted to extend the default MVC behavior, you would either create and register a custom model binder, or a value provider.

Along came WebApi which shared with MVC its design philosophy and many of its ideas, but used a different pipeline.

The model binding process was one of the key differences between MVC and WebApi.
It was critical whether you were binding from the Url or the Body, as there were two separate model binding processes:

• Binding from the **Url** used a similar model binding process than the MVC one, based on `ModelBinders` and `ValueProviders`, adding type converters into the mix (which convert strings into a given types).

• Binding from the **body** followed a different process which involved the added `MediaTypeFormatters`. These basically deserialized the request body into a type, and the specific formatter to be used was determined from the Content-Type header. Formatters like `JsonMediaTypeFormatter` and `XmlMediaTypeFormatter` were added, deserializing the body using **Newtonsoft.Json**, **XmlSerializer** or **DataContractSerializer**.

By default, simple types used the Url binding while complex types used the `MediaTypeFormatters`.

The framework also provided hooks for the user to be specific about which process should be used and to customize it, with the final set of rules described in its documentation.

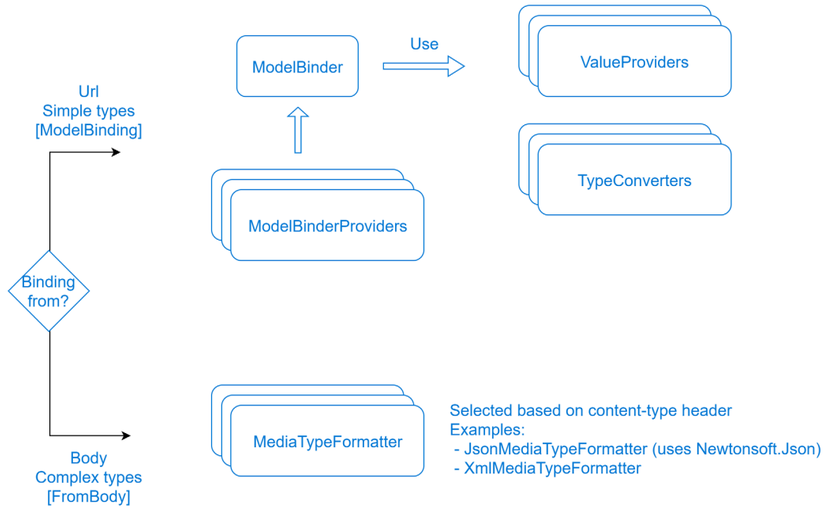


Figure 2, a simplified view of the old WebApi parameter binding

The new ASP.NET Core Model Binding process

In ASP.Net Core, both WebApi and MVC have been merged into a single framework.

However, as you read in the previous section, they had different approaches to model binding. So, you might be wondering how does model binding work in ASP.Net Core? How did it merge both approaches?

The answer is that the new framework followed the direction started by WebApi, but has integrated the formatters within a unified binding process, based on model binders.

The formatters are still there, now abstracted behind the `IInputFormatter` interface, but these formatters are now used by specific model binders like the `BodyModelBinder`.

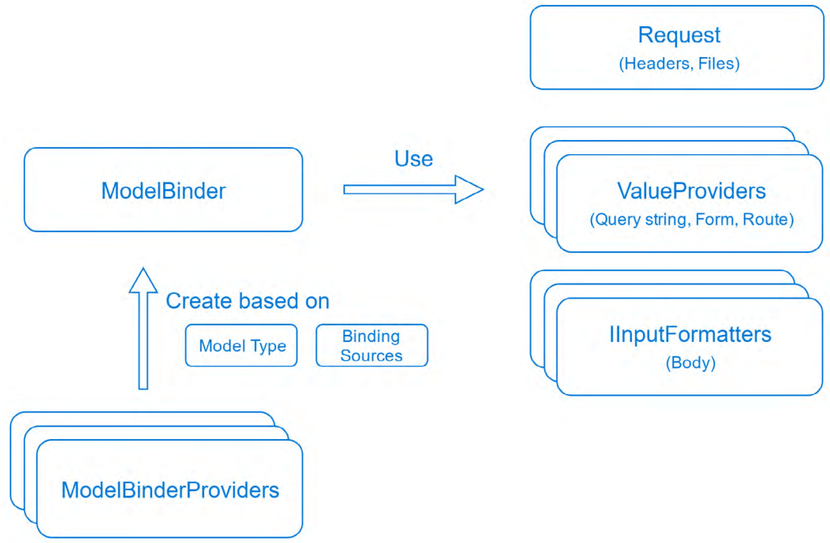


Figure 3, a simplified view of the model binding in ASP.Net Core

All of this means that there are different ways of binding your parameters and models. Depending on which model binder is selected, the framework might end up:

• using value providers and even other model binders for nested properties

• using an input formatter to deserialize the entire model from the body

Why should I care?

Knowing the model binding process followed behind the scenes is going to be critical for you to customize it.

People who are used to the binding process in the previous ASP.NET MVC framework, might get particularly surprised by the changes!

• Not every binding source is checked by default. Some model binders require you to specifically enable a binding source. For example, when binding from the Body, you need to add `[FromBody]` to your model/action parameter, otherwise the BodyModelBinder won't be used.

• In particular, the Headers, Body and Files binding sources must be specifically enabled by the user and will use specific model binders.

• There are value providers for route values, query string values and form values. Binders based on value providers can get data from any of these, but won't be able to get data from other sources like the body for example. *Although form data is posted in the body as a URL-encoded string, it is a special case parsed and interpreted as a value provider by the framework.*

• Binding from the body anything other than URL-encoded form data, will always use a formatter instead of the value providers, with the entire model being deserialized from the body. No separated binding for

each individual property will be attempted!

- A JSON formatter based on JSON.Net is added by default. All the JSON.Net hooks and options to customize how data is serialized/deserialized are available, like custom JsonConverters.

- Formatters for XML can be added, both `XmlSerializer` and `DataContractSerializer` are available.

The number of different model binders with its providers that are added by default is not a small list. You can customize that list adding/removing providers as you see fit through the `MvcOptions.ModelBinderProviders`.

The default list of binder providers might give you a better idea of how the binding process works. Bear in mind that providers are evaluated in order, the first one returning a non-null binder wins:

```
options.ModelBinderProviders.Add(new BinderTypeModelBinderProvider());
options.ModelBinderProviders.Add(new ServicesModelBinderProvider());
options.ModelBinderProviders.Add(new BodyModelBinderProvider(options.
InputFormatters, _readerFactory, _loggerFactory));
options.ModelBinderProviders.Add(new HeaderModelBinderProvider());
options.ModelBinderProviders.Add(new SimpleTypeModelBinderProvider());
options.ModelBinderProviders.Add(new CancellationTokenModelBinderProvider());
options.ModelBinderProviders.Add(new ByteArrayModelBinderProvider());
options.ModelBinderProviders.Add(new FormFileModelBinderProvider());
options.ModelBinderProviders.Add(new FormCollectionModelBinderProvider());
options.ModelBinderProviders.Add(new KeyValuePairModelBinderProvider());
options.ModelBinderProviders.Add(new DictionaryModelBinderProvider());
options.ModelBinderProviders.Add(new ArrayModelBinderProvider());
options.ModelBinderProviders.Add(new CollectionModelBinderProvider());
options.ModelBinderProviders.Add(new ComplexTypeModelBinderProvider());
```

If you look carefully at the list of the binder provider names, you can infer that some binders will be selected for specific model types, while others will be selected for specific binding sources!

For example:

- `BodyModelBinder`, will be selected when `[FromBody]` is used, and will just use the request body through the `InputFormatters`.

- `ComplexTypeModelBinder`, will be selected when binding a complex type and no specific source like `[FromBody]` is used.

When you create a custom model binder, you need to think carefully:

- Whether you will bind from sources that have associated value providers (like query string or form data) or associated input formatters (like the body)

- Whether you intend your binder to be used with nested properties. For example, if you create a model binder for properties of type decimal, it will be used when binding a parent model from the Form data, but it won't be used when binding a parent model from a JSON body. In the second case, the InputFormatter deserializes the entire model in one go.

The latter might take most people coming from MVC 5 by surprise and is the scenario we will go through in the following section.

Custom Model Binders

The scenario

Let's go through a typical example used to demonstrate custom model binders in previous versions of the framework. This is the scenario where `dateTime` fields are provided as separated date and time properties, but you still want to bind them to a single `dateTime` field in your model.

- **Important note:** I like this as an example for model binders because it is easy enough to understand and has already been used as an example in the past. However, I wouldn't use this on a real system where I would just stick with timestamps or ISO Strings and use JavaScript to combine values from multiple editors.

Let's say we have an appointment model like this:

```
public class Appointment
{
    public string Id { get; set; }
    public string Name { get; set; }
    public DateTime AppointmentDate { get; set; }
}
```

When the following form is posted, we want our custom model binder to bind the separated date and time posted values into our `Appointment.AppointmentDate` property:

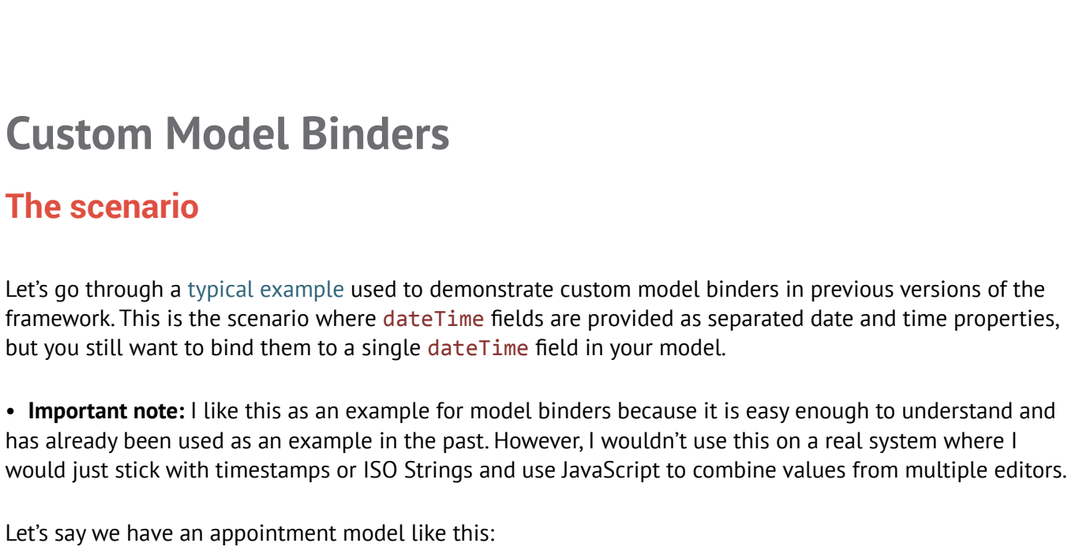


Figure 4, Form with date split in 2 fields

Following the same idea, if a JSON string like the following is posted, we also want the separated date and time values to be bound as the `Appointment.AppointmentDate` property:

```
{
  "name":"Dummy Appointment",
  "appointmentDate":{
    "date":"10/03/2017",
    "time":"11:34"
```

```
 }
}
```

As discussed in the previous sections, we will need to handle both scenarios differently.

- In the case of the form data, we can create a custom model binder.
- However, in the second case where a JSON is posted, we will need to customize the way the `JsonInputFormatter` will deserialize that property.

If you have any trouble following along, you can download the code from **Github**.

Create a new Model Binder used with forms

In this scenario, we have a view where an HTML form is rendered and submitted to our controller. This form will contain separated inputs for the date and time of the AppointmentDate property.

The following code block shows the relevant parts of the view:

```
<div class="form-group">
  <label asp-for="Name" class="col-md-2 control-label"></label>
    <div class="col-md-10">
      <input asp-for="Name" class="form-control" />
        <span asp-validation-for="Name" class="text-danger"></span>
    </div>
</div>

<div class="form-group">
  <label asp-for="AppointmentDate" class="col-md-2 control-label"></label>
    <div class="col-md-5">
      <input asp-for="AppointmentDate" name="AppointmentDate.Date" class=
      "form-control" value="@Model?.AppointmentDate.ToString("d")" />
    </div>
    <div class="col-md-5">
      <input asp-for="AppointmentDate" name="AppointmentDate.Time" class=
      "form-control" value="@Model?.AppointmentDate.ToString("t")" />
    </div>
    <span asp-validation-for="AppointmentDate" class="col-md-12 text-danger">
    </span>
</div>
```

Of course, this is accompanied by controller actions that render the view and handle the form being submitted:

```
public IActionResult CreateForm()
{
  return View();
}

[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> CreateForm([Bind("Name,AppointmentDate")]
Appointment appointment)
{
  if (ModelState.IsValid)
  {
    appointment.Id = Guid.NewGuid().ToString();
    _context.Add(appointment);
    await _context.SaveChangesAsync();
    return RedirectToAction("Index");
  }
  return View(appointment);
}
```

So far, this is all quite standard in any MVC application.

First attempt at our custom binder

Now we need to create a new custom model binder for our `AppointmentDate` property. Since this is intended for posted forms, we can use the value providers to retrieve the *AppointDate.Date* and *AppointmentDate.Time* form values.

- For the sake of simplicity, in the code that follows, I will force both date and time values to be present in the request data. While it suits the purposes of the article, in the real world, you most likely want them to be optional and proceed with the binding as long as one of the two values is found.

- A similar remark must be made about the handling of different time zones, formats and cultures. I am sticking with the invariant culture and UTC for the purposes of the article, but double check your use case and needs, in case you need to handle different ones.

The following shows a very rough version of this idea, without getting deep into different date/time formats and error handling, however it is enough for the purposes of the article and demonstrating how to use custom binders:

```
public class SplitDateTimeModelBinder: IModelBinder
{
  public Task BindModelAsync(ModelBindingContext bindingContext)
  {
    // Make sure both Date and Time values are found in the Value Providers
    // NOTE: You might not want to enforce both parts
    var datePartName = $"{bindingContext.ModelName}.Date";
    var timePartName = $"{bindingContext.ModelName}.Time";
    var datePartValues = bindingContext.ValueProvider.GetValue(datePartName);
    var timePartValues = bindingContext.ValueProvider.GetValue(timePartName);
    if (datePartValues.Length == 0 || timePartValues.Length == 0) return Task.
    CompletedTask;

    // Parse Date and Time
   // NOTE: You might want a stronger/smarter handling of time zones, formats and
   cultures
   DateTime.TryParseExact(
     datePartValues.FirstValue,
     "d",
     CultureInfo.InvariantCulture,
     DateTimeStyles.None,
     out var parsedDateValue);
     DateTime.TryParseExact(
     timePartValues.FirstValue,
     "t",
     CultureInfo.InvariantCulture,
     DateTimeStyles.AdjustToUniversal,
     out var parsedTimeValue);
```

```
    // Combine into single DateTime which is the end result
    var result = new DateTime(parsedDateValue.Year,
                  parsedDateValue.Month,
                  parsedDateValue.Day,
                  parsedTimeValue.Hour,
                  parsedTimeValue.Minute,
                  parsedTimeValue.Second);
    bindingContext.ModelState.SetModelValue(bindingContext.ModelName, result,
    $"{datePartValues.FirstValue} {timePartValues.FirstValue}");
    bindingContext.Result = ModelBindingResult.Success(result);
    return Task.CompletedTask;
  }
}
```

Now we need to tell the MVC framework when to use this model binder.

As we probably don't want to use it for every DateTime, we can just add the [ModelBinder] attribute to our model:

```
public class Appointment
{
  public string Id { get; set; }
  public string Name { get; set; }
  [ModelBinder(BinderType = typeof(SplitDateTimeModelBinder))]
  public DateTime AppointmentDate { get; set; }
}
```

A better binder with fallback for regular date time values

The approach described above requires us to explicitly tell whether a `DateTime` field in a model will be posted as separated date and time values, so we can add the `[ModelBinder]` attribute.

Depending on your requirements, you might be more interested in registering a new model binder provider that will create an instance of our binder for any `DateTime` property, regardless of whether date and time are submitted as a single or separated fields.

Before using this approach, let's rewrite the binder so we can use the default `SimpleTypeModelBinder` as a fallback in case the separated Date and Time values are not found.

This isn't complicated, we just need to inject a fallback `IModelBinder` and use it when either Date or Time are missing.

The changed parts of the binder are displayed next:

```
public class SplitDateTimeModelBinder: IModelBinder
{
  private readonly IModelBinder fallbackBinder;
  public SplitDateTimeModelBinder(IModelBinder fallbackBinder)
  {
     this.fallbackBinder = fallbackBinder;
  }

  public Task BindModelAsync(ModelBindingContext bindingContext)
  {
    // Make sure both Date and Time values are found in the Value Providers
    var datePartName = $"{bindingContext.ModelName}.Date";
    var timePartName = $"{bindingContext.ModelName}.Time";
    var datePartValues = bindingContext.ValueProvider.GetValue(datePartName);
    var timePartValues = bindingContext.ValueProvider.GetValue(timePartName);

    // Fallback to the default binder when a part is missing
    if (datePartValues.Length == 0 || timePartValues.Length == 0) return
    fallbackBinder.BindModelAsync(bindingContext);

    // Parse Date and Time. From this point onwards the binder is unchanged
```

Now let's create a model binder provider that uses our updated binder for any DateTime field:

```
public class SplitDateTimeModelBinderProvider : IModelBinderProvider
{
  private readonly IModelBinder binder = new SplitDateTimeModelBinder(
  new SimpleTypeModelBinder(typeof(DateTime)));
  public IModelBinder GetBinder(ModelBinderProviderContext context)
  {
    return context.Metadata.ModelType == typeof(DateTime) ? binder : null;
  }
}
```

Finally, let's register the provider at the beginning of the list in `Startup.ConfigureServices`:

```
services.AddMvc(opts =>
{
  opts.ModelBinderProviders.Insert(0, new SplitDateTimeModelBinderProvider());
});
```

That's it, now our custom binding logic will be used when separated Date/Time values are found, while the default binding logic will be used otherwise.

If you check the code on **Github**, you can try the update view which uses a single form field for the AppointmentDate.

Testing our model binder with ajax requests

Let's say we have to create a different controller endpoint we can use with Ajax requests or as a REST API:

```
[HttpPost]
public async Task<IActionResult> CreateJson(Appointment appointment)
{
  if (ModelState.IsValid)
  {
    appointment.Id = Guid.NewGuid().ToString();
    _context.Add(appointment);
    await _context.SaveChangesAsync();
    return Ok();
  }
  return StatusCode(400);
}
```

You might wonder what will happen if you just get the previous form and post it using an ajax call.

For example, let's consider the following simple jQuery code that grabs all fields and posts the data:

```
$.ajax({
  type: 'POST',
  url: '/appointments/createjson',
  data: data,
  success: function (data, status, xhr) {
    if (xhr.status == 200) { // go to appointments index }
  }
});

// Where data can be one of these 2 options:

var data = $("#appointmentForm").serialize();
var data = {
  name: $('input[name=Name]').val(),
  appointmentDate: {
    date: $('input[name="AppointmentDate.Date"]').val(),
    time: $('input[name="AppointmentDate.Time"]').val()
  }
};
```

The code above will send an ajax request with the Content-Type header set as `application/x-www-form-urlencoded`. This is critical for the model binding process, as those values will be available in the `FormValueProvider` and our custom binder will work as expected. Everything will behave the same as when the HTML form was submitted.

Now let's change the code above to post a JSON and specify the Content-Type set as `application/json`:

```
var data = {
  name: $('input[name=Name]').val(),
  appointmentDate: {
    date: $('input[name="AppointmentDate.Date"]').val(),
    time: $('input[name="AppointmentDate.Time"]').val()
  }
};
$.ajax({
  type: 'POST',
  url: '/appointments/createjson',
  data: JSON.stringify(data),
  contentType: 'application/json',
  success: function (data, status, xhr) {
    if (xhr.status == 200) window.location('/appointments');
  }
});
```

You will notice that our model binder is not executed, in fact the model received by the controller action will be null!

The following section will shed some light on this scenario.

Updating how the posted JSON data is deserialized

When posting a JSON and specifying the Content-Type as `application/json`, the value providers will have no data available at all, as they only look at the Query string, route and form values!

In this case, we need to explicitly tell MVC that we want to bind our model from the request body where the posted JSON will be found.

Notice the usage of the `[FromBody]` attribute:

```
[HttpPost]
public async Task<IActionResult> CreateJson([FromBody]Appointment appointment)
{
  …
}
```

However, by doing this we will be using the `BodyModelBinder` instead of the binder we got before (the `ComplexModelBinder`). This new binder will internally use the `JsonInputFormatter` to deserialize the JSON found in the request body into an instance of the `Appointment` class.

As the entire model will be deserialized, **our custom DateTime binder will not be used!**

So how can we customize this scenario, and deserialize the separated date and time parts into a single DateTime value?

Since the JsonInputFormatter uses JSON.Net, we can write a custom JsonConverter and inform JSON.Net about it!

Let's start by creating a new converter that will verify that we have our separated date and time values, reading them both and creating the combined DateTime.

We will also specify that it should only be used when reading JSON:

```
public class SplitDateTimeJsonConverter : JsonConverter
{
  public override bool CanWrite => false;

  public override bool CanConvert(Type objectType)
  {
    return objectType == typeof(DateTime);
  }

  public override object ReadJson(JsonReader reader, Type objectType, object
  existingValue, JsonSerializer serializer)
  {
     // Make sure we have an object with date and time properties
     // Example: { date: "01/01/2017", time: "11:35" }
     if (reader.TokenType == JsonToken.Null) return null;
     if (reader.TokenType != JsonToken.StartObject) return null;
     var jObject = JObject.Load(reader);
     if (jObject["date"] == null || jObject["time"] == null) return null;

      // Extract and parse the separated date and time values
      // NOTE: You might want a stronger/smarter handling of locales, formats and
      cultures
      DateTime.TryParseExact(
        jObject["date"].Value<string>(),
        "d",
        CultureInfo.InvariantCulture,
```

```
        DateTimeStyles.None,
        out var parsedDateValue);
      DateTime.TryParseExact(
        jObject["time"].Value<string>(),
        "t",
        CultureInfo.InvariantCulture,
        DateTimeStyles.AssumeUniversal,
        out var parsedTimeValue);

        // Combine into single DateTime as the end result
        return new DateTime(parsedDateValue.Year,
                    parsedDateValue.Month,
                    parsedDateValue.Day,
                    parsedTimeValue.Hour,
                    parsedTimeValue.Minute,
                    parsedTimeValue.Second);
    }

    public override void WriteJson(JsonWriter writer, object value, JsonSerializer
    serializer)
    {
      throw new NotImplementedException();
    }
}
```

Then we can explicitly register it for our AppointmentDate property using the `[JsonConverter]` attribute.

```
public class Appointment
{
  public string Id { get; set; }
  public string Name { get; set; }
  [JsonConverter(typeof(SplitDateTimeJsonConverter))]
  public DateTime AppointmentDate { get; set; }
}
```

However, as we saw, when we created the custom model binder, you might prefer a single converter that we can register for all DateTime properties. Again, we will need to rewrite it so we have a fall back handler when no separated date/time values are provided. It is very similar to what we did with the custom binder, using a DateTimeConverterBase as fall back:

```
public class SplitDateTimeJsonConverter : JsonConverter
{
  private readonly DateTimeConverterBase fallbackConverter;
  public SplitDateTimeJsonConverter(DateTimeConverterBase fallbackConverter)
  {
    this.fallbackConverter = fallbackConverter;
  }

  public override object ReadJson(JsonReader reader, Type objectType, object
  existingValue, JsonSerializer serializer)
  {
    // Make sure we have an object with date and time properties
    // Example: { date: "01/01/2017", time: "11:35" }
    if (reader.TokenType == JsonToken.Null) return null;
    if (reader.TokenType != JsonToken.StartObject)
    return fallbackConverter.ReadJson(reader, objectType, existingValue,
    serializer);

    var jObject = JObject.Load(reader);

    if (jObject["date"] == null || jObject["time"] == null)
      return fallbackConverter.ReadJson(reader, objectType, existingValue,
    serializer);

  // No changes from this point onwards

}
```

We are using the default `IsoDateTimeConverter` as fall back, you might need to set its format/culture properties or use a different fallback converter as per JSON.Net date handling.

Conclusion

The model binding process has changed significantly in ASP.Net Core, especially if you compare it against the binding process used in the previous versions of ASP.Net MVC. Those of you who are used to ASP.Net WebAPI will find the new binding process closer to what you already know.

Now you need to be aware of the different binding sources, which ones of those are available by default (like the query string or form data binding sources) and which ones must be explicitly activated (like the body or header binding sources).

When customizing the model binding process, being aware of the different model binders and which binding sources these are used with, will be critical. Just creating and registering a new model binder will be enough as long as you plan using binding sources available as value providers (like query string or form data). However other binding sources like the body or the header will require different strategies and extra effort.

As an example, we have seen how to customize DateTime bindings when submitting forms and when posting JSON data. This required a new custom model binder and a new JsonConverter ■

Download the entire source code from GitHub at
bit.ly/dncm30-core-modelb

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 and Suprotim Agarwal for reviewing this article.

Keerti Kotaru

ANGULAR 4

APPLICATION DEVELOPMENT

WITH BOOTSTRAP 4 AND TYPESCRIPT

Angular is a framework for building client applications in HTML and JavaScript, or by using a language like TypeScript, which ultimately compiles to JavaScript. In this tutorial, we will build an Angular 4 application using TypeScript.

A brief Angular 4 History

So far, three major Angular versions have been released – Angular v1.x (a.k.a AngularJS), Angular 2 and the newly released Angular 4 (also known as Angular). AngularJS has had a few major releases of its own with v1.1, v1.2 and so on, but let us stick with v1.x for all purposes of discussion.

Angular v1.x and v2 are quite different in terms of architecture.

While Angular v1.x (also known as AngularJS) was based on an MVC architecture, Angular 2 is based on a component/services architecture. Considering Angular was moving from MV (Model View Whatever) pattern to components focused approach, the framework features were very different from each other and caused many application developers to rewrite a major portion of their code base.*

However, Angular v2 to v4 is a very different story. It is a rather progressive enhancement. A majority of changes are non-breaking.

Angular 4 was released on 23rd March '17.

What happened to Angular v3?

Angular 2 has been a single repository, with individual packages downloadable through npm with the `@angular/package-name convention`. For example `@angular/core, @angular/http, @angular/router` so on.

Considering this approach, it was important to have a consistent version numbering among various packages. Hence, the Angular team skipped a major version 3. It was to keep up the framework with Angular Router's version. Doing so will help avoid confusions with certain parts of the framework on version 4, while the others on version 3.

Angular 4 Enhancements

Consider the following enhancements in Angular v4,

- This release has considerable improvements in bundle size. Some have reported up to 60% reduction in Angular bundles' file size

- The ngc, AOT compiler for Angular and TypeScript is much faster

- Angular 4 is compatible with TypeScript's newer versions 2.1 and 2.2. TypeScript release helps with better type checking and enhanced IDE features for Visual Studio Code. The changes helped the IDE detect missing imports, removing unused declarations, unintentionally missing "this" operator etc.

This article and the **code sample** demonstrates an Angular 4 application using TypeScript and Bootstrap 4 (bit.ly/dnc-boot-tut). Please note that Bootstrap 4 is an alpha release at the time of writing this article.
Editorial Note: *This article assumes you are familiar with developing applications using Angular. If you are new to Angular, check some of our* Angular tutorials *(bit.ly/dnc-ang-tut). If you are new to TypeScript, read our* TypeScript *Tutorial (bit.ly/dnc-ts-tut).*

Getting Started with Angular 4 project - SetUp

Angular CLI

Angular CLI is a preferred way to get started with an Angular project (v2.x and above). It not only saves time, but also makes it easy to maintain the code base during the course of the project, with features to add additional components, services, routing etc. Refer to the appendix at the end of this article for an introduction to Angular CLI.

Get Angular CLI at https://cli.angular.io/ .

Angular CLI latest version

Since the CLI project is available as an NPM package, so you can install angular-cli globally on your system by running the following command:

```
npm install -g angular-cli@latest
```

However if you already have Angular CLI installed and want to upgrade to the latest version of Angular CLI,

run the following commands:

```
npm uninstall -g angular-cli
npm cache clean
npm install -g angular-cli@latest
```

To create a new project with Angular CLI, run the following command

```
ng new DinoExplorer
```

`DinoExplorer` is the name of the new project. You may optionally use --routing parameter to add routing to the Angular project.

However, if you prefer to configure and install Angular and Webpack manually, refer to the *Get started with Angular 4 (manual way)* section in the appendix.

The latest version of Angular CLI (v1.0) makes scaffolding possible with Angular 4. If you are using an older version, upgrade to Angular CLI. A new project created using Angular CLI now references Angular 4. Refer to figure 1.

Scaffolding makes it possible to add new components, routes or services etc. from the command line.

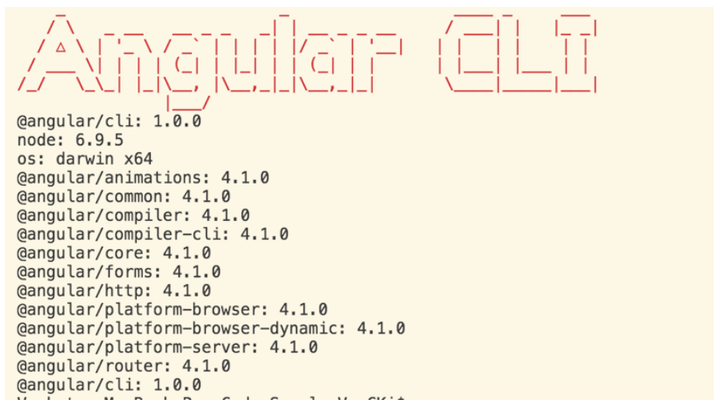


Figure 1 Version details for various libraries with ng -v command

Upgrade from Angular v2 to v4

However, to upgrade an existing project to Angular 4, run following *npm install* commands which upgrades Angular and TypeScript on a Mac or Linux machine.

```
npm install @angular/{common,compiler,compiler-cli,core,forms,http,platform-
browser,platform-browser-dynamic,platform-server,router,animations}@latest
typescript@latest –save
```

It installs the latest stable version of twelve libraries including TypeScript. At the time of writing, this article uses the latest stable version for libraries which is Angular v4.1.0 and TypeScript v2.2.0

Upgrade Angular and TypeScript on a Windows machine

The command to upgrade Angular and TypeScript on a Windows machine is as follows:

```
npm install @angular/common@latest @angular/compiler@latest @angular/compiler-cli@
latest @angular/core@latest @angular/forms@latest @angular/http@latest @angular/
platform-browser@latest @angular/platform-browser-dynamic@latest @angular/platform-
server@latest @angular/router@latest @angular/animations@latest typescript@latest
--save
```

Add Bootstrap 4

Install bootstrap Alpha release using NPM

```
npm install --save bootstrap@4.0.0-alpha.6
```

We chose a specific version to ensure future releases doesn't break the sample. Optionally, run the following command to install the latest pre-release package.

```
npm install –save bootstrap@next
```

Once the package is downloaded, add Bootstrap references in `.angular-cli.json`.

Modify styles configuration to add Bootstrap CSS

```
styles”: [
   “../node_modules/bootstrap/dist/css/bootstrap.min.css”,
   “styles.css”
],
```

Modify scripts configuration to add jQuery, Bootstrap and Tether JS files.

```
“scripts”: [
   “../node_modules/jquery/dist/jquery.min.js”,
   “../node_modules/tether/dist/js/tether.min.js”,
   “../node_modules/bootstrap/dist/js/bootstrap.min.js”
],
```

Angular 4 - Code Sample

Let us explore the Angular and Bootstrap features with the code sample provided with this article. It has three views.

a. **Basic:** For demonstrating `ng-template` feature `using *ngIf` directive. More details described later in the article.

b. **Dinosaur List:** A home page that shows dinosaur list. The list is responsive using Bootstrap 4 (Alpha). Cards and the content inside align according to screen size.

c. **Random Dino:** Randomly picks and shows a dinosaur card. By design, the response is delayed by four seconds. This is to simulate a loading message while an observable is asynchronously retrieving data.

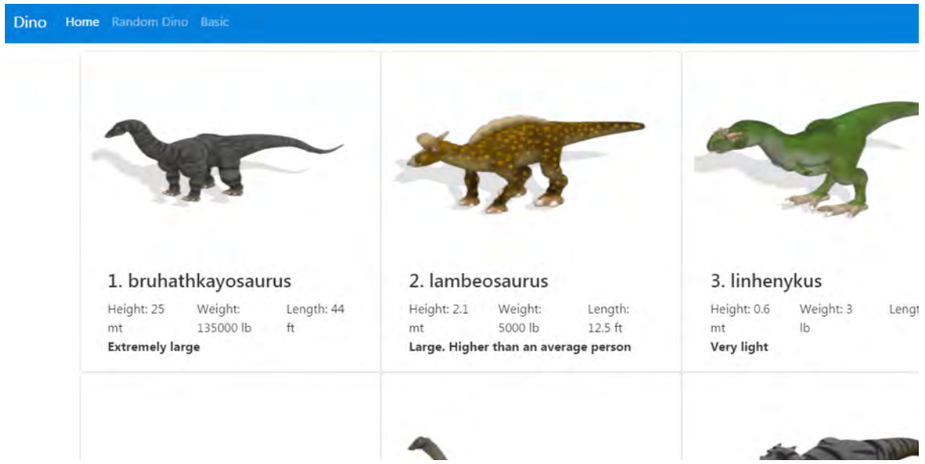


Figure 2 Demo app with dinosaur list screen

Visual Studio Code IDE

Visual Studio Code, an open source IDE from Microsoft, has good integration with TypeScript and Angular. With the new features in TypeScript 2.2, this IDE can detect missing and unused imports, missing this operator and also allows for quick refactoring.

Editorial Note: *If you are new to Visual Studio Code, here's a* productivity guide.

The Visual Studio Code extensions' eco system provides many useful features.

Angular 4 TypeScript Snippets (bit.ly/dnc-ang4-ts-snippet) is one such extension. It's a collection of snippets readily available for creating a component, service, pipe, module etc. Start typing "a-" in VS Code, and it shows list of available snippets. See figure 3 for details on Angular artifacts that we can create with the extension.

Imagine we chose to create a component. *Tab through* to provide component selector, template file name and component class name etc. The Component skeleton file is ready to use. This is an alternative to using Angular CLI for creating a new component.

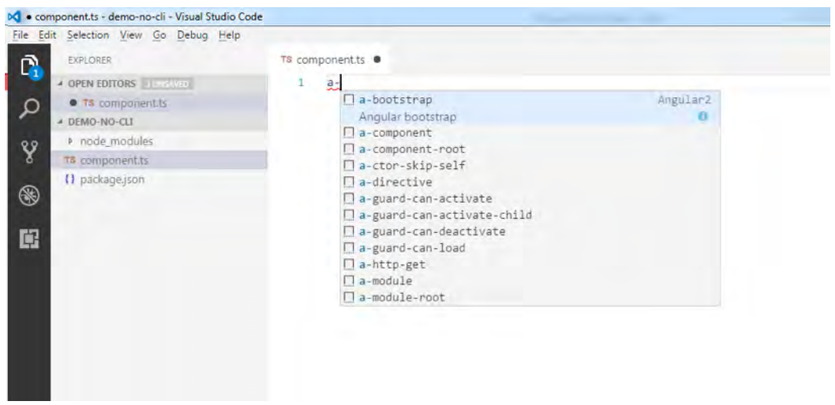


Figure 3 Angular 4 TypeScript snippet extension options

Angular 4 New Features and Angular v2 vs 4 Differences

The following sections elaborate Angular 4 features and certain differences with Angular 2.x

Template changes to ng-template

If you are upgrading from Angular 2.x to Angular 4, all template elements have to change to ng-template. The following code will result in a warning.

```
<template [ngIf]=”isVisible”> Conditional statement </template>

Template parse warnings: The <template> element is deprecated. Use <ng-template>
instead
```

Refactor it to:

```
<ng-template [ngIf]=”isVisible”> Conditional statement </ng-template>
```

TypeScript's Strict *Null Check* options is around the corner

In TypeScript 2.0, a **Strict Null Check** option has been introduced. It restricts assigning null or undefined only to variables that are defined with that type.
Consider the following example,

```
let num1: number;
num1 = null;
console.log(num1);
```

Strict null checks mode prevents assigning null to a number.

```
D:\StrictNullCheck-sample>tsc sample.ts --strictNullChecks
sample.ts(2,1): error TS2322: Type ‘null’ is not assignable to type ‘number’.
```

We can explicitly declare num1 to be of type null to allow assigning to it:

```
let num1: number | null ;
num1 = null;
console.log(num1);
```

However, this option is not fully supported yet. Angular doesn't build with `strictNullCheck` enabled in `tsconfig.json`.

Angular made relevant changes to support the feature in Angular 4.1. However, a bug in TypeScript is keeping the feature at bay. Keep an eye on these Angular and TypeScript issues for updates –

1. https://github.com/angular/angular/issues/15432 (strictNullChecks support in v4.0 is broken)
2. https://github.com/angular/angular/pull/15405 (https://github.com/angular/angular/pull/15405)
3. https://github.com/Microsoft/TypeScript/issues/10078

Angular *ngIf/then/else

With Angular 2, we could use `*ngIf` directive to conditionally show a section of the template. With Angular

4, support for `else` has been added. Consider the following template code:

```
<div *ngIf=”isTrue; else whenFalseTmpl”>
 <span>I show-up when isTrue is true.</span>
</div>

<ng-template #tmplWhenFalse > I show-up when isTrue is false </ng-template>
```

When `isTrue` value is true, instead of showing the span inline, we could offload to another template.

```
<button class=”btn btn-primary” (click)=”toggle()”> Toggle </button>
<div *ngIf=”isTrue; then tmplWhenTrue else tmplWhenFalse”></div>

<ng-template #tmplWhenTrue >I show-up when isTrue is true. </ng-template>
<ng-template #tmplWhenFalse > I show-up when isTrue is false </ng-template>
```

Consider a snippet from code sample below. It has three conditional blocks.

A dinosaur could be:
- light weight with less than 100 pounds or
- medium sized with less than 10,000 pounds or
- of extremely large size with greater than 10,000 pounds.

The `ngIf` checks dino object weight property:

1. When it’s less than 100 pounds show `#bird` template.
2. The else template has another ‘if’ condition to see if it’s less than 10000 pounds. When *true*, show the `#large` template
3. When both the above conditions fail, show the `#extraLarge` template.

```
<div *ngIf=”dino.weight < 100; then bird else large”></div>
 <ng-template #bird> <strong>Very light</strong></ng-template>

 <ng-template #large> <div *ngIf=”dino.weight<10000; else
   extraLarge”><strong>Large. Higher than an average person</strong></div>
 </ng-template>

 <ng-template #extraLarge> <strong>Extremely large</strong></ng-template>
</div>
```

Working with Observables and *ngIf/else

While rendering an observable on a template, we can show loading message or a spinner gif with the `*ngIf` directive. Specify *else* template to show while async observable is not ready with data.

The directive also supports creating a local variable. Notice a local variable `dino` (`let dino`) to refer to the async object. Consider following code.

```
<!—show else template “working” while loading observable with data. Notice async
filter for dino observable -->
 <div *ngIf=”dino | async; else working; let dino”>
  <div class=”card col-8”>
   <div class=”col-4”>
    <img class=”card-img-top” src=”assets/images/{{dino.name}}.png” [alt]=”dino.
    name”>
   </div>
   <div class=”card-block”>
    <h4 class=”card-title”>{{dino.name}}</h4>
      <!--removing dinosaur card details for readable snippet. Refer to code sample
      for complete code. -->
   </div>
   <!-- card end -->
  </div>
  <ng-template #working>
  <div>Loading...</div>
 </ng-template>
```

In the sample, to mimic delayed loading, the component calls `next()` on an observable subject after four seconds.

Refer to following code snippet,

```
this.dino = new Subject<any>();
// to mimic delayed loading the component calls next on observable subject
after four seconds.
setTimeout( () =>
this.dino.next(dataset.dinosaurs[Math.round((Math.random() * 5))]) , 4000);
```

Angular Upgrade Guide

Angular Upgrade Guide is a useful tool to review things to consider while upgrading from Angular 2.x to Angular 4 or above. Access the tool at https://angular-update-guide.firebaseapp.com/

Angular Animations

In Angular 2.x, animations were part of `@angular/core`. It was part of the bundle even if the application never used animations. With Angular 4, animations related artifacts have been moved out of `@angular/core`. It helps reduce production bundle size.

To use animations, import `BrowserAnimationsModule` from `@angular/platform-browser/animations`. Reference the module in imports array of `@NgModule`

Angular future release schedule

Refer to the major release schedule in Table 1. The details are quoted from this link (bit.ly/dnc-ang-release). Please keep a tab on the page for any changes to the schedule, considering it is still tentative.
Tentative Schedule after March 2017

Date	Stable Release	Compatibility*
September/October 2017	5.0.0	^4.0.0
March 2018	6.0.0	^5.0.0
September/October 2018	7.0.0	^6.0.0

Table 1: Release schedule taken from AngularJS blog https://github.com/angular/angular/blob/master/docs/RELEASE_SCHEDULE.md

Bootstrap 4

Bootstrap 4 is a rewrite of its previous version (v3). The newer version is built with Sass and it also supports Flexbox. One of the core features of Bootstrap is the grid system, which helps develop responsive layout.

Bootstrap 4 has enhanced breakpoints with better support for handheld mobile devices. It now supports five breakpoints as opposed to four in Bootstrap 3. Refer to following details,

< 576 px	Extra Small (col-xs-*)
>= 576 px	Small (col-sm-*)
>= 768 px	Medium (col-md-*)
>= 992 px	Large (col-lg-*)
>= 1200 px	Extra Large (col-xl-*)

Grid system changes compared to Bootstrap 3:

- In the new Bootstrap 4 grid system, the extra small (col-xs-*) breakpoint is modified to a width less than 576px. In the previous version (v3), anything less than 768 px was considered extra small.

- Bootstrap 4, anything greater than 1200px is extra-large. This width was labelled large in previous version. As mentioned in the table above, the large breakpoint (col-lg-*) is now greater than or equal to 992px.

To learn more about Bootstrap 4, refer to the articles Bootstrap 4 – New Features and Migrating from Bootstrap 3 to 4.

Consider the following code.

```
<div class=”card-text”>
 <div class=”row”>
  <div class=”col-lg-4 col-xs-12 col-sm-6 “>
   Height: {{dino.height}} mt
  </div>
  <div class=”col-lg-4  col-xs-12 col-sm-6”>
   Weight: {{dino.weight}} lb
  </div>
  <div class=”col-lg-4  col-xs-12 col-sm-6”>
   Length: {{dino.length}} ft
  </div>
 </div>
</div>
```

It uses break points for extra-small (< 578 px), small (>= 576px) and large (> 1200px). It is applied on three dinosaur fields, height, weight and length.

On an extra small screen, each field/div element takes up all twelve segments to show each field/div element in a row.

On a small (and medium) screen, it takes six segments fitting two fields/div elements in a row and pushing the third field/div element to next row.

On a large screen, each field/div element occupies four out of twelve segments. Hence it shows the three fields/div elements in one row (twelve segments in total).

Our code sample follows a similar approach to align cards on the page.

```
<div class=”card col-lg-4 col-sm-6 col-xs-12” *ngFor=”let dino of dinosaurs; index
as dinoIndex;”>
<!—card definition goes here -->
</div>
```

- It shows dinosaur cards in a single column on an extra small screen with col-xs-12; takes up all twelve segments. Refer to figure 4.

Figure 4: Extra Small screen. Dinosaur cards and height, weight and length fields aligned in a single row

- Dinosaur cards are aligned to two columns on a small and medium screen with col-sm-6; takes up six segments, i.e, half a row. Refer to figure 5.

Figure 5 On a Small (and medium screen) two dinosaur cards are shown in a row. Two fields (Height, weight and length) are shown in a row

- Dinosaur cards are aligned to three columns on a large and extra-large screen with col-lg-4, takes up four

segments, i.e, 1/3rd of a row. Refer to figure 6.

Figure 6 on a large (and extra-large) screen, three dinosaur cards shown in a row. Height, weight and length fields show in a single row

Conclusion

Angular went through a good amount of transition from Angular 1.x, MV* model, to the framework we know today.

The purpose of the transition is to effectively support new features in JavaScript, as well as sync up with the latest web development standards. When used with TypeScript, it is on steroids with support for types, integration with IDE like Visual Studio Code and many other useful features.

In the process, Angular 2 upgrade included many breaking changes. However, upgrading to future versions of Angular are expected to be smooth with minimal or no breaking changes. The Angular framework will continue to evolve to support more features and make the developer's job easy.

This is a good thing. Obviously we do not want a stagnated framework. On a lighter note, dinosaurs are exciting in museums and for developers, they can be seen only in code samples.

Download the entire source code of this article (Github).

Appendix

1. Introduction to Angular CLI

Angular CLI is an easy and effective way to get started with an Angular project (v2 and above). It scaffolds an Angular project with Webpack (or JSPM/SystemJS), TypeScript, unit tests' skeleton, routing (optional) etc. During the course of the project, it also helps to add Angular artifacts to the repository. We can add components, services, directives, pipes, classes, guards, interfaces, enums and modules to the project.

The tool is in line with Angular style guide.

Install Angular CLI with the following command

```
npm i -g @angular/cli
```

Create a new Angular project with

```
ng new my-angular-project
```

After installation, to run the project execute the following command

```
ng serve
```

To add a new component use the following command

```
ng g component my-component
```

2. Get started with Angular 4 (manual way)

Consider the following steps to get started with an Angular 4 and Webpack (for bundling) incase you do not chose to go ahead with Angular CLI.

- Initialize an npm package with the following command. Provide values like package name, description, version etc. when prompted.
npm init

- Use the package.json from the code sample – manual-starter folder. Notice all Angular packages are latest and above 4.0.0 (indicated by ^)

```
"@angular/common": "^4.0.0",
 "@angular/compiler": "^4.0.0",
 "@angular/core": "^4.0.0",
 "@angular/forms": "^4.0.0",
 "@angular/http": "^4.0.0",
 "@angular/platform-browser": "^4.0.0",
 "@angular/platform-browser-dynamic": "^4.0.0",
 "@angular/router": "^4.0.0",
```

- Webpack Configuration files: The webpack.conf.js is at the root of the project. It branches to three more configuration files i) webpack.dev.js ii) webpack.prod.js and iii) webpack.common.js.

- Use the start script to run *webpack dev server* using dev configuration. Consider following script command.

```
"scripts": {
   "start": "webpack-dev-server --inline --progress --port 8080",
```

- Webpack creates three bundles, with the application bundled into app.js

```
entry: {
   'polyfills': './src/polyfills.ts',
   'vendor': './src/vendor.ts',
   'app': './src/main.ts'
 },
```

- Root for the Angular application is main.ts. It bootstraps the application. It loads the main Angular module, which in this example is located in app/app.module.ts

- app.module.ts starts with the main component app.component.ts, located in the same folder

References

For more about Angular CLI NPM repo with documentation, follow the link https://www.npmjs.com/package/angular-cli

Angular 4 release notes: http://angularjs.blogspot.in/2017/03/angular-400-now-available.html

Configuring Webpack with Angular https://angular.io/docs/ts/latest/guide/webpack.html

Bootstrap 4, breakpoint list https://v4-alpha.getbootstrap.com/layout/grid/#grid-options

Release schedule reference https://github.com/angular/angular/blob/master/docs/RELEASE_SCHEDULE.md

Angular Upgrade tool https://angular-update-guide.firebaseapp.com/

Dinosaur data for the code sample – https://dinosaur-facts.firebaseio.com/

Dinosaur images for code sample- spore.com

Keerti Kotaru

Author

V Keerti Kotaru has been working on web applications for over 15 years now. He started his career as an ASP.Net, C# developer. Recently, he has been designing and developing web and mobile apps using JavaScript technologies. Keerti is also a Microsoft MVP, author of a book titled 'Material Design Implementation using AngularJS' and one of the organisers for vibrant ngHyderabad (AngularJS Hyderabad) Meetup group. His developer community activities involve speaking for CSI, GDG and ngHyderabad.

Thanks to Suprotim Agarwal and Mahesh Sabnis for reviewing this article.

Benjamin Jakobus

Comparing VueJS to Angular and React

Angular is a popular, fully-fledged JavaScript framework used to build modern web applications. React and VueJS are up-and-coming JavaScript libraries, and are used to build web-interfaces.

In this article, we will contrast AngularJS (v1), Angular 2 and React with VueJS by building a small, single-page sample app. If you are new to VueJS, read this beginner tutorial bit.ly/dnc-vuejs

The article will consider the similarities and differences between Angular, VueJS and React in templating, component definition, directive definition, application design, performance, design flexibility and functionality.

VueJS, Angular and React – Similarities and Differences

We will begin by building a demo application using AngularJS (the first version of Angular), as AngularJS has served as an inspiration in the design of VueJS. AngularJS remains to be one of the most popular JavaScript frameworks for developing modern web-applications. Released almost seven years ago, the framework is supported by Google, and aims to provide developers with all the necessary tools, along with a strict set of patterns for front-end development.

Editorial Note: *The latest version of Angular as of this writing is v4. After AngularJS, Angular 2 was released. There was no v3. Since Angular v2 to v4 is a progressive enhancement, a majority of the changes are non-breaking and almost all the Angular 2 principles and differences discussed in this article, apply to Angular v4 too.*

VueJS is an up-and-coming JavaScript library for developing web-interfaces. Released in 2013, VueJS was inspired by AngularJS (v1), and as such borrows the framework's templating syntax. For example, and as we will see shortly, the syntax for loops, conditionals, model declarations and interpolation all look very similar to that used by AngularJS.

Editorial Note: *If you are new to VueJS, read this beginner tutorial* http://bit.ly/dnc-vuejs

Vue's focus is on doing two things well: ease of use and rendering speed.

It excels in both. The library is extremely easy to learn and use, and beats both AngularJS and React when it comes to page rendering. As such, VueJS feels almost like Angular's fitter little brother as it shares the aspects done well, but at the same time is much lighter, more performant and less opinionated than AngularJS.

However being a library, as opposed to a framework, means that VueJS does not offer the same amount of functionality that AngularJS offers. For example, whilst Angular provides things such as a HTTP request service or router, VueJS users will require to rely on third-party code if they wish to avail of such functionalities.

VueJs vs Angular

As VueJS is gaining in popularity, it is important to understand the differences, advantages and disadvantages between VueJS and AngularJS. Therefore this article sets out to explore both VueJS and AngularJS, by building a sample demo application: **SurfPics**.

The hypothetical SurfPics website will allow surfers to display their various beach and surf videos and photographs. In the first section of this article, we will outline the boilerplate code for this demo application. In the subsequent two sections we will then walk you through the development of the application, using first AngularJS and then VueJS. The similarities between the frameworks should become evident quite quickly.

Once we have established the similarity between the two frameworks, we will contrast VueJS to Angular 2, which deviates slightly from its predecessor (AngularJS). Nonetheless, here too, similarities are apparent.

Last but not least, we will discuss the advantages, disadvantages, differences and similarities between a performant alternative to VueJS and AngularJS: React.

React was developed by Facebook. Similar to VueJS, its objective is to build performant web-interfaces.

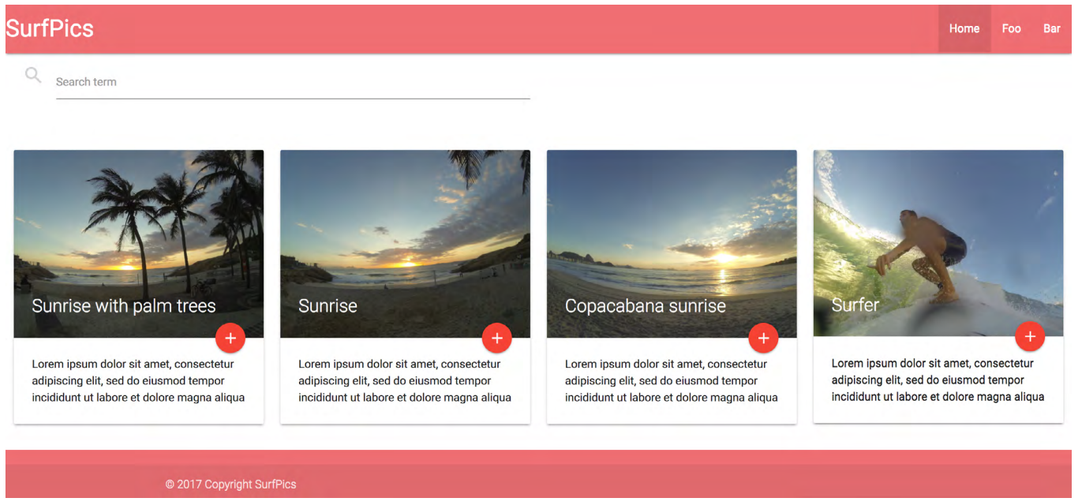


Figure 1: Our demo page - built using Materialize

Demo application - Preparation

Before we can start diving into VueJS and AngularJS, we must lay the groundwork for our demo application.

To begin with, create a directory called `src`, containing three sub-directories: `angular`, `angular2` and `vue`. These directories will contain our AngularJS and VueJS application respectively.
Inside both the `angular` and `vue` folder, create the following three, empty, files:

```
-  index.html
-  app.js
-  app.css
```

Add the following styles to app.css:

```
#app {
  min-height: 460px;
}
.text-muted {
  color: lightgrey;
}
.search-box {
  margin-left: 1em;
  margin-top: -1em;
}
```

The above CSS rules are fairly self-explanatory: they will set the colors, margins and minimum height for our application.

As this article does not concern itself with the intricacies of markup and CSS styles, you can safely ignore these boilerplate rules once you have added them to the application's CSS file.

The index.html file should contain the skeleton for our sample application:

```
<!DOCTYPE html>
<html>
 <head>
  <title>SurfPics</title>
   <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/
   materialize/0.98.0/css/materialize.min.css">
   <link href="http://fonts.googleapis.com/icon?family=Material+Icons"
   rel="stylesheet">
   <link rel="stylesheet" href="app.css">
 </head>
 <body>
  <nav>
   <div class="nav-wrapper">
    <a href="#" class="brand-logo">SurfPics</a>
     <ul id="nav-mobile" class="right hide-on-med-and-down">
      <li class="active"><a href="index.html">Home</a></li>
      <li><a href="foo.html">Foo</a></li>
      <li><a href="bar.html">Bar</a></li>
     </ul>
   </div>
  </nav>
  <div id="app">
   <!-- App content goes here -->
  </div>
  <footer class="page-footer">
   <div class="footer-copyright">
    <div class="container">
     © 2017 Copyright SurfPics
    </div>
   </div>
  </footer>
 <script src="app.js"></script>
 <script type="text/javascript" src="https://code.jquery.com/jquery-2.1.1.min.js">
 </script>
 <script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.98.0/js/
 materialize.min.js"></script>
 </body>
</html>
```

Again, the above markup is very straightforward and self-explanatory: We created a page containing a navbar and a footer, and included the relevant dependencies (jQuery and materialize).

As per best-practices, JavaScript files are loaded at the bottom of the page (as opposed to the top, which hampers download parallelization and may give the impression of poor page load times).

Note the third party dependency aside from jQuery: **Materialize**.

Materialize is *"a modern responsive front-end framework based on Material Design"* (http://materializecss.com/), and we will be using it to style our demo application ("Material Design" refers to a set of visual design guidelines created by Google). The framework provides some slick styling, and is essentially the Bootstrap equivalent for Material Design lovers.

Building a page using AngularJS 1.6

The demo application that we will be building is going to be very simple and will consist of one page, displaying a set of available photographs (see figure 1). Each photograph will be accompanied by a short description and a title. The photograph, title and description will be presented in the form of a *card*.

In addition to displaying a set of photographs, we will be providing a search bar above the photographs, which, in theory, will allow users to search for photographs whose title or description contains a certain string.

Although we will not be implementing the search functionality itself, we will be implementing a "searching" message to indicate to the user that the search is in progress as the search term is being entered.
Let's begin by first including AngularJS 1.6.1 in our page: Place the following line above the `script` tag for `app.js`:

```
<script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.6.1/angular.min.
js"></script>
```

Next, we define our application controller. Open angular/app.js and insert the following code.

```
angular.module('app', [])
 .controller('AppController', function() {
   var app = this;
 });
```

Inside this application controller, we will be hard-coding our set card information:

```
angular.module('app', [])
 .controller('AppController', function() {
   var app = this;
   this.cards = [
   {
     title: 'Sunrise with palm trees',
     description: 'Lorem ipsum dolor sit amet, aliqua',
     img_src: "img/1.jpg"
   },
   {
     title: 'Sunrise',
     description: 'Lorem ipsum dolor sit amet, consectetur aliqua',
     img_src: "img/2.jpg"
   },
   {
     title: 'Copacabana sunrise',
     description: 'Lorem ipsum dolor sit ut labore et dolore magna aliqua',
     img_src: "img/3.jpg"
   },
   {
     title: 'Surfer',
     description: 'Lorem ipsum dolor sit ame magna aliqua',
     img_src: "img/4.png"
   }
 ];
});
```

To display the cards, first define the controller: add `ng-controller="AppController as ctrl"` to the div where `id="app"`.

Next add the following markup inside this application div:

```
<div class="row">
 <div class="col s12 m3" ng-repeat="card in ctrl.cards">
  <div class="card">
   <div class="card-image">
    <img ng-src="{{card.img_src}}">
    <span class="card-title">{{card.title}}</span>
    <a class="btn-floating halfway-fab waves-effect waves-light red">
     <i class="material-icons">add</i>
    </a>
   </div>
   <div class="card-content">
    <p>{{card.description}}</p>
   </div>
  </div>
 </div>
</div>
```

The markup itself is fairly self-explanatory: we create one row, containing 4 columns. Each column contains one card. Specifically, we iterate over the cards contained in the controller using `ng-repeat="card in ctrl.cards"`. For each card, we interpolate its contents using the expression `{{card.property}}`, where *property* is the card property that we wish to access.

The various classes used in the sample code above are defined by materialize, and served as mere decoration.

Next, we are going to define the search bar.

Insert the following markup above the row containing the photographs:

```
<div class="row">
 <form class="col s12">
  <div class="row">
   <div class="input-field col s6">
    <i class="material-icons prefix text-muted">search</i>
     <input ng-change="ctrl.search()" id="icon_prefix" type="text" class="validate"
     ng-model="ctrl.search_term">
     <label for="icon_prefix">Search term</label>
   </div>
  </div>
  <div ng-if="ctrl.search_term" class="row text-muted" class="search-box">
   <i class="material-icons search">hourglass_empty</i> {{ctrl.message}}
  </div>
 </form>
</div>
```

Once again, the markup should be self-explanatory: we add a row containing an input box.

By using `ng-change`, we ensure that the controller's search function is called whenever the input changes. Using `ng-if` we conditionally display an hourglass icon along with a message (see figure 2 below) if a search term exists (i.e. if the controller's `search_term` property is truthy).

The search input is bound to `search_term` property of our controller using `ng-model`.

The search function itself is defined inside the controller as follows:

```
this.search = function () {
  app.message = 'Searching for ' + app.search_term + '...';
};
```

And that's it - we now have a basic page that:

- Dynamically generates rows using a loop and interpolates the relevant data
- Binds the data of an input
- Conditionally displays a search message

Figure 2: An hour-glass icon along with a search message appears once the search input contains text.

Building a page using VueJS

By building a VueJS equivalent of our AngularJS demo page, we will see that, upon first glance, VueJS seems like a simplified clone of AngularJS. Its directives carry similar names and the VueJS app itself requires no bootstrapping.

We begin building our demo app by including VueJS as one of our dependencies.

Open `vue/index.html`, and insert the following line above the `script` tag for `app.js`:

```
<script src="https://unpkg.com/vue/dist/VueJS"></script>
```

Next, we define our VueJS instance in `app.js` as follows:

```
var app = new Vue({
  el: '#app'
});
```

Note how little overhead is required to create this instance, and we are not forced to use the concepts of modules and controllers; instead, we simply create a Vue object that targets our application `div` (i.e. the `div` with `id="app"`).

Targeting our application `div` means that this Vue instance is available anywhere within the application `div`.

To define our cards (or any other data), we must use the `data` property. We will also initialise message to an empty string and `search_term` to `null`:

```
var app = new Vue({
  el: '#app',
  data: {
    message: '',
    search_term: null,
    cards: [
          {
            title: 'Sunrise with palm trees',
            description: 'Lorem ipsum dolor sit amet, consectetur',
            img_src: "img/1.jpg"
          },
          {
            title: 'Sunrise',
            description: 'Lorem ipsum dolor sit amet, consectetur',
            img_src: "img/2.jpg"
          },
          {
            title: 'Copacabana sunrise',
            description: 'Lorem ipsum dolor sit amet, consectetur',
            img_src: "img/3.jpg"
          },
          {
             title: 'Surfer',
             description: 'Lorem ipsum dolor sit amet, consectetur',
             img_src: "img/4.png"
          }
      ]
  }
});
```

Last but not least, we will define our search function by using the methods property:

```
methods: {
  search: function () {
    this.message = 'Searching for ' + this.search_term + '...'; }
}
```

Now it is time to update our template.

As we will see, templates in AngularJS and VueJS look almost identical: we will be replacing `ng-model` with `v-model`, `ng-if` with `v-if`, `ng-repeat` with `v-for` and `ng-bind` with `v-on:change="search"` (table 1 below lists the full set of templating differences between AngularJS and VueJS).

AngularJS 1	VueJS
ng-if	v-if
ng-repeat	v-for
ng-model	v-model
ng-class	v-class
ng-click	v-on:click
ng-change	v-on:change
ng-show	v-show
ng-bind	v-model
ng-disabled	-
-	v-bind:style

Table 1: Templating differences between AngularJS 1 and VueJS

Specifically, we will replace:

```
<div class="col s12 m3" ng-repeat="card in ctrl.cards">
```

with:

```
<div class="col s12 m3" v-for="card in cards">
```

Next, we replace:

```
<img src="{{card.img_src}}">
```

with:

```
<img v-bind:src="card.img_src">
```

And replace:

```
<input ng-change="ctrl.search()" id="icon_prefix" type="text" class="validate" ng-
model="ctrl.search_term">
```

with:

```
<input v-on:change="search" id="icon_prefix" type="text" class="validate"
v-model="search_term">
```

Last but not least, we replace:

```
<div ng-if="ctrl.search_term" class="row text-muted" class="search-box">
     <i class="material-icons search">hourglass_empty</i> {{ctrl.message}}
</div>
```

with:

```
<div v-if="search_term" class="row text-muted" class="search-box">
    <i class="material-icons search">hourglass_empty</i> {{message}}
</div>
```

And that's it! Pronto!

We now have a fully functioning VueJS version of our AngularJS demo app. In summary, here is the complete template:

```
<!DOCTYPE html>

<html>
 <head>
  <title>SurfPics</title>
   <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/
   materialize/0.98.0/css/materialize.min.css">
   <link href="http://fonts.googleapis.com/icon?family=Material+Icons"
   rel="stylesheet">
   <link rel="stylesheet" href="app.css">
 </head>
 <body>
  <nav>
   <div class="nav-wrapper">
    <a href="#" class="brand-logo">SurfPics</a>
    <ul id="nav-mobile" class="right hide-on-med-and-down">
     <li class="active"><a href="index.html">Home</a></li>
     <li><a href="foo.html">Foo</a></li>
     <li><a href="bar.html">Bar</a></li>
   </ul>
  </div>
</nav>
<div id="app">
 <div class="row">
  <form class="col s12">
   <div class="row">
    <div class="input-field col s6">
     <i class="material-icons prefix text-muted">search</i>
     <input v-on:change="search" id="icon_prefix" type="text" class="validate"
     v-model="search_term">
     <label for="icon_prefix">Search term</label>
    </div>
   </div>
   <div v-if="search_term" class="row text-muted" class="search-box">
    <i class="material-icons search">hourglass_empty</i> {{message}}
   </div>
  </form>
 </div>
 <div class="row">
  <div class="col s12 m3" v-for="card in cards">
   <div class="card">
    <div class="card-image">
     <img v-bind:src="card.img_src">
     <span class="card-title">{{card.title}}</span>
     <a class="btn-floating halfway-fab waves-effect waves-light red">
      <i class="material-icons">add</i></a>
    </div>
    <div class="card-content">
     <p v-sp-grey>{{card.description}}</p>
    </div>
   </div>
  </div>
 </div>
</div>

<footer class="page-footer">
 <div class="footer-copyright">
  <div class="container">
   © 2017 Copyright SurfPics
  </div>
 </div>
</footer>
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script src="app.js"></script>
<script type="text/javascript" src="https://code.jquery.com/jquery-2.1.1.min.
js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.98.0/js/
materialize.min.js"></script>
</body>
</html>
```

Building a page using Angular (v2)

Before we can begin building an Angular 2 application, we must first setup our development environment. As per the official Angular2 documentation:

"The QuickStart seed contains the same application as the QuickStart playground. But its true purpose is to provide a solid foundation for local development. Consequently, there are many more files in the project folder on your machine, most of which you can learn about later."

To install the QuickStart seed, navigate to https://angular.io/docs/ts/latest/guide/setup.html#download and download the QuickStart seed. Extract the downloaded archive into the `angular2` directory that we created previously. Copy the image folder (`img`) into `quickstart-master/src`. From your command line, execute the following three commands:

```
cd quickstart-master
npm install
npm start
```

A new browser window should open: http://localhost:3000/

Inside `quickstart-master/src`, update index.html to the following:

```
<!DOCTYPE html>
<html>
 <head>
  <title>SurfPics</title>
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/
  materialize/0.98.0/css/materialize.min.css">
  <link href="http://fonts.googleapis.com/icon?family=Material+Icons"
  rel="stylesheet">
  <link rel="stylesheet" href="app.css">
  <base href="/">
```

```
 <meta charset="UTF-8">
 <meta name="viewport" content="width=device-width, initial-scale=1">
 <link rel="stylesheet" href="styles.css">

 <!-- Polyfill(s) for older browsers -->
 <script src="node_modules/core-js/client/shim.min.js"></script>

 <script src="node_modules/zone.js/dist/zone.js"></script>
 <script src="node_modules/systemjs/dist/system.src.js"></script>

 <script src="systemjs.config.js"></script>
 <script>
  System.import('main.js').catch(function(err){ console.error(err); });
 </script>
</head>
<body>
 <nav>
  <div class="nav-wrapper">
   <a href="#" class="brand-logo">SurfPics</a>
   <ul id="nav-mobile" class="right hide-on-med-and-down">
    <li class="active"><a href="index.html">Home</a></li>
    <li><a href="foo.html">Foo</a></li>
    <li><a href="bar.html">Bar</a></li>
   </ul>
  </div>
 </nav>
 <my-app>Loading AppComponent content here ...</my-app>

 <footer class="page-footer">
   <div class="footer-copyright">
    <div class="container">
     © 2017 Copyright SurfPics
    </div>
   </div>
  </footer>
  <script src="https://unpkg.com/vue/dist/vue.js"></script>
  <script src="app.js"></script>
  <script type="text/javascript" src="https://code.jquery.com/jquery-2.1.1.min.
  js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.98.0/js/
  materialize.min.js"></script>
 </body>
</html>
```

As we already covered the markup in the previous sections, we won't iterate over the description here. However, take note of the following line;

```
<my-app>Loading AppComponent content here …</my-app>
```

This references the root component that will ideally consist of sets of nested components. As our application is simple, we won't be developing a tree of components, but instead just work with the application root component (called "my-app").

To this end, update `quickstart-master/src/app/app.component.ts` to contain the following:

```
import { Component } from '@angular/core';
@Component({
 selector: 'my-app',
 template: `
<div class="row">
 <form class="col s12">
  <div class="row">
   <div class="input-field col s6">
    <i class="material-icons prefix text-muted">search</i>
    <input (keyup)="search($event)" id="icon_prefix" type="text" class="validate"
    v-model="search_term">
    <label for="icon_prefix">Search term</label>
   </div>
  </div>
  <div v-if="search_term" class="row text-muted" class="search-box">
   <i class="material-icons search">hourglass_empty</i> {{message}}
  </div>
 </form>
</div>
<div class="row">
 <div class="col s12 m3" *ngFor="let card of cards">
  <div class="card">
   <div class="card-image">
    <img src="{{card.img_src}}">
    <span class="card-title">{{card.title}}</span>
    <a class="btn-floating halfway-fab waves-effect waves-light red"><i class=
    "material-icons">add</i></a>
   </div>
   <div class="card-content">
    <p>{{card.description}}</p>
   </div>
  </div>
 </div>
</div>
 `,
})
export class AppComponent  {
 message = '';
 search_term: null;
 cards = [
 {
  title: 'Sunrise with palm trees',
  description: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
  eiusmod tempor incididunt ut labore et dolore magna aliqua',
  img_src: "img/1.jpg"
 },
 {
  title: 'Sunrise',
  description: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
  eiusmod tempor incididunt ut labore et dolore magna aliqua',
  img_src: "img/2.jpg"
 },
 {
  title: 'Copacabana sunrise',
  description: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
  eiusmod tempor incididunt ut labore et dolore magna aliqua',
  img_src: "img/3.jpg"
 },
 {
  title: 'Surfer',
  description: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
  eiusmod tempor incididunt ut labore et dolore magna aliqua',
  img_src: "img/4.png"
 }
];
```

```
search(event: any) {
 this.message = 'Searching for ' + event.target.value + '...';
 }
}
```

Having developed both the AngularJS 1 app, and the VueJS application, the above code should look familiar, even if is written in TypeScript: we define our search method and the apps data.

The markup for the component goes into the component's `template`. By examining the markup, we see that there is no difference in the actual structure of the application - only in the naming of the default directives (more on this later).

In summary, here's the complete Angular template:

```
<!DOCTYPE html>
<html>
 <head>
  <title>SurfPics</title>
  <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/
  materialize/0.98.0/css/materialize.min.css">
  <link href="http://fonts.googleapis.com/icon?family=Material+Icons"
  rel="stylesheet">
  <link rel="stylesheet" href="app.css">
  <base href="/">
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="styles.css">

  <!-- Polyfill(s) for older browsers -->
  <script src="node_modules/core-js/client/shim.min.js"></script>

  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/systemjs/dist/system.src.js"></script>

  <script src="systemjs.config.js"></script>
  <script>
   System.import('main.js').catch(function(err){ console.error(err); });
  </script>
 </head>
 <body>
  <nav>
   <div class="nav-wrapper">
    <a href="#" class="brand-logo">SurfPics</a>
    <ul id="nav-mobile" class="right hide-on-med-and-down">
     <li class="active"><a href="index.html">Home</a></li>
     <li><a href="foo.html">Foo</a></li>
     <li><a href="bar.html">Bar</a></li>
    </ul>
   </div>
  </nav>
  <my-app>Loading AppComponent content here ...</my-app>

  <footer class="page-footer">
   <div class="footer-copyright">
    <div class="container">
     © 2017 Copyright SurfPics
    </div>
   </div>
  </footer>
  <script src="https://unpkg.com/vue/dist/vue.js"></script>
  <script src="app.js"></script>
  <script type="text/javascript" src="https://code.jquery.com/jquery-2.1.1.min.
   js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/materialize/0.98.0/js/
  materialize.min.js"></script>
 </body>
</html>
```

Angular 2	VueJS
*ngIf	v-if
*ngFor	v-for
[ngClass]	v-class
(click)	v-on:click
[(ngModel)]	v-model

Table 2: Templating differences between Angular 2 and VueJS

Similarities between AngularJS 1.6 and VueJS

Templating

As we have just seen, the most striking similarity between AngularJS and VueJS is templating. The syntax is almost identical, with AngularJS using the `ng` prefix, and VueJS using the `v` prefix. This similarity is by no means accidental: As noted by the VueJS developers on the official VueJS website: *"this [similarity] is because there were a lot of things that Angular got right and these were an inspiration for Vue very early in its development."*

Directives and components

Both AngularJS and VueJS offer components and directives. Whilst AngularJS makes no clear distinction between components and directives, it clearly states that *"directives are meant to encapsulate DOM manipulations only, while components are self-contained units that have their own view and data logic"*.

In terms of actually creating a directive, we have seen that the AngularJS way is more elaborate. VueJS on the other hand allows developers to define directives much more concisely. The same holds true for components.

For example, consider that we may want to define a directive that colors an image description text grey. In VueJS, we would first define the directive:

```
Vue.directive('sp-grey', {
 });
```

where `sp-grey` is our directive name. We then simply change the color of the bound element as soon as it is inserted into the DOM:

```
Vue.directive('sp-grey', {
 inserted: function (el) {
  el.style.color = 'lightgrey';
 }
});
```

The directive can then be used by prepending to v- prefix:

```
<p v-sp-grey>{{card.description}}</p>
```

Consider the typical AngularJS equivalent:

```
angular.module('app').directive('psGrey', function () {
 return {
  restrict: 'A',
   scope: {
    text: '@'
   },
   template: '<span style="color: lightgrey;">{{ text }}</span>'
  };
});
```

We would then use the directive as follows:

```
<p ps-grey text="{{card.description}}"></p>
```

Note: In the above, we could alternatively avoid using a template by using link to access the element directly:

```
angular.module('app').directive('psGrey', function () {
 return {
  restrict: 'A',
   scope: {
   },
   link: function(scope, element, attrs) {
    element.css({'color': 'lightgrey'});
     element.text(attrs.text);
   }
  };
});
```

Development: Similarities between Angular 2 and VueJS

Templating

Just as with AngularJS 1.6, when it comes to templating, Angular 2 and VueJS are almost identical.

Again, the Angular 2 syntax is almost identical to both VueJS and AngularJS 1.6. Whilst AngularJS 1.6 uses the `ng` prefix for in-built directives and VueJS uses the v prefix, Angular 2 uses the `*ng` prefix. So, for example, what is `v-for` in VueJS, is `*ngFor` in Angular 2.

Interpolation is identical, and uses the `{{ }}` form, as is the case for both VueJS and AngularJS 1.6.

Angular 2 uses the concept of *template statements* in order to bind an action to an event, such as a button.

As per the official Angular 2 documentation: *"A template statement responds to an event raised by a binding target such as an element, component, or directive. You'll see template statements in the event binding section, appearing in quotes to the right of the = symbol as in (event)="statement"".*

This means that what is written as `<button v-on:click="perform_task()">Click Me</button>` in VueJS, is expressed as `<button (click)="perform_task()">Click Me</button>` in Angular 2 (see table 2 above).

Directives and components

Aside from TypeScript, the most striking difference between Angular 2 and VueJS, is that Angular 2 is entirely component based. Everything is a component. For example, when creating the SurfPics equivalent in Angular 2, we are really defining a SurfPics component:

```
@Component({
 selector: 'SurfPics'
})
@View({
 templateUrl: './index.html'
})
export class AngularComparison {
 constructor() {
  this.message = '';
   this.search_term = null;
    // Remainder of app
 }
}
```

Angular 2 components themselves are just directives with a template. Much like VueJS, Angular 2 directives are concisely defined and are used for encapsulating DOM operations. Similar to VueJS, an Angular 2 attribute directive may be used to change the appearance of an element. In Angular 2, our previously introduced `ps-grey` directive is defined in a similar manner than its VueJS equivalent:

```
import { Directive, ElementRef, Input } from '@angular/core';
@Directive({ selector: '[psGrey]' })
export class PsGreyDirective {
 constructor(el: ElementRef) {
  el.nativeElement.style.color = 'lightgrey';
 }
}
```

Development: Similarities between React and VueJS

Templating

React is slightly less intuitive than VueJS.

Instead of using templating, React components are defined using JSX (an XML-like syntax for JavaScript). That is, instead of writing a HTML template, one declares a component class and then calls a special `render` method to define what it is that is to be rendered:

```
var hello_world = React.createClass({
 render: function() {
  return (
```

```
   <h1>Hello World</h1>
  )
 }
});
```

Upon first glance, this looks straightforward, however it can become more complex quickly as a lot of boilerplate code is required to develop more complex applications. For example, to render the list of cards contains the SurfPics beach photos from our previous example, we must:

- define a React component
- declare the cards and then call `React.render()`, indicating what it is that we want rendered:

```
// Code placed inside React component definition above
const cards = [
 {
  title: 'Sunrise with palm trees',
  description: 'Lorem ipsum dolor sit amet, consectetur',
  img_src: "img/1.jpg"
 },
 {
  title: 'Sunrise',
  description: 'Lorem ipsum dolor sit amet, consectetur',
  img_src: "img/2.jpg"
 },
];
const Iteration = ({ items }) => <div> {items.map(item => <span>{item}</span>)}
</div>;

ReactDOM.render(<Iteration items={cards}/>, document.getElementById('cards'));
```

Then, in our `index.html`, we would assign the `id="cards"` to the element that we want repeated:

```
<div class="col s12 m3" id="cards">
```

Directives and components

In the React universe, the functionality offered by Angular directives are supported through a set of different types of *components*. Aside from the default component, it allows developers to define stateless functional components. For example:

```
const FooButton = ({
  name,
  callback
}) => (
    <button onClick={callback}>Click me, {name}</button>
)
```

The stateless functional components are possibly the closest analogy to VueJS directives, in that they are commonly used to create purely presentational components. Also known as "dumb components", these components are not concerned with behavioral logic, but instead just focus on the UI.

Whilst allowing for a more functional-style approach, the lack of templating and directives, and the use of JSX makes React a bit less straight-forward to use than VueJS. For example, in order to express a simple conditional, one would need to develop an entire component:

```
var hello_world = React.createClass({
 render: function() {
  if (this.something) {
   return (<h1>Hello World</h1>);
  } else {
  return (<h1>Bye Bye</h1>);
  }
 }
});
```

The VueJS equivalent would amount to:

```
<h1 v-if="something">Hello World</h1>
<h1 v-else>Bye Bye</h1>
```

Comparing VueJS to other frameworks

Performance

When it comes to rendering performance, VueJS generally outperforms many existing popular frameworks such as AngularJS or React. In fact, VueJS is one of the fastest frameworks to date. For example, the AngularJS 1 equivalent of our demo application, containing one row of data, took almost 14.7 milliseconds longer to render. The significance of this performance difference becomes more evident when increasing our dataset from 1 row to 100 rows of data: VueJS now renders the page in 488.7 milliseconds, whilst AngularJS requires 534.3 milliseconds.

Figure 3 below by Stefan Krause (http://stefankrause.net) gives a more in-depth summary of the various frameworks and performance differences.

Duration in milliseconds (Slowdown = Duration / Fastest)

	angular v2.4.3-non-keyed	aurelia v1.0.7	binding.scala v10.0.1	cyclejs-dom v14.1.0	dio v3.0.5	domvm v2.0.1-non-keyed	inferno v1.2.0	nx v1.0.0-beta.1.1.0-non-keyed	polymer v1.7.0	ractive-edge	ractive v0.8.9-non-keyed	react v15.4.2-non-keyed	riot v3.0.7	simulacra v1.5.5	svelte v1.0.1	tsers v1.0.0	vue v2.1.10-non-keyed	vanillajs
create rows Duration for creating 1000 rows after the page loaded.	179.20 ± 3.15 (1.41)	178.81 ± 2.07 (1.40)	304.76 ± 7.10 (2.39)	150.71 ± 4.74 (1.18)	136.05 ± 0.91 (1.07)	140.94 ± 3.88 (1.11)	142.90 ± 1.96 (1.12)	208.67 ± 8.56 (1.64)	199.50 ± 1.61 (1.57)	297.77 ± 5.74 (2.34)	321.12 ± 3.69 (2.52)	181.98 ± 2.66 (1.43)	303.04 ± 5.28 (2.38)	292.13 ± 0.67 (2.29)	136.93 ± 1.42 (1.08)	255.52 ± 3.89 (2.01)	152.92 ± 1.49 (1.20)	127.36 ± 2.52 (1.00)
replace all rows Duration for updating all 1000 rows of the table (with 5 warmup iterations).	54.62 ± 0.58 (1.01)	83.52 ± 0.91 (1.54)	79.61 ± 3.26 (1.47)	73.60 ± 0.89 (1.36)	55.43 ± 0.89 (1.03)	59.32 ± 0.47 (1.10)	59.78 ± 1.60 (1.11)	70.10 ± 0.52 (1.30)	60.00 ± 1.42 (1.11)	64.14 ± 0.95 (1.19)	67.66 ± 1.06 (1.25)	77.90 ± 1.08 (1.44)	62.58 ± 0.46 (1.16)	62.23 ± 0.45 (1.15)	54.65 ± 2.13 (1.01)	115.49 ± 1.77 (2.14)	62.61 ± 0.56 (1.16)	54.07 ± 0.84 (1.00)
partial update Time to update the text of every 10th row (with 5 warmup iterations).	10.40 ± 0.70 (1.03)	9.97 ± 0.45 (1.00)	11.07 ± 0.61 (1.00)	25.08 ± 0.52 (1.57)	11.64 ± 0.85 (1.00)	14.87 ± 0.50 (1.00)	10.63 ± 0.36 (1.00)	10.72 ± 0.37 (1.00)	12.33 ± 0.53 (1.00)	10.79 ± 0.58 (1.00)	14.44 ± 0.53 (1.00)	15.61 ± 1.07 (1.00)	16.39 ± 0.13 (1.02)	9.72 ± 0.17 (1.00)	10.26 ± 0.46 (1.00)	33.82 ± 0.89 (2.11)	15.51 ± 0.56 (1.00)	9.59 ± 0.62 (1.00)
select row Duration to highlight a row in response to a click on the row. (with 5 warmup iterations).	8.65 ± 7.02 (1.00)	11.87 ± 0.45 (1.00)	3.11 ± 0.43 (1.00)	17.11 ± 0.25 (1.07)	4.54 ± 1.02 (1.00)	6.40 ± 0.41 (1.00)	5.97 ± 0.62 (1.00)	7.20 ± 0.78 (1.00)	4.16 ± 0.37 (1.00)	7.20 ± 0.79 (1.00)	8.88 ± 2.65 (1.00)	4.97 ± 0.25 (1.00)	9.15 ± 0.45 (1.00)	3.92 ± 1.00 (1.00)	4.38 ± 1.71 (1.00)	24.66 ± 0.70 (1.54)	7.66 ± 0.18 (1.00)	3.49 ± 2.00 (1.00)
swap rows Time to swap 2 rows on a 1K table. (with 5 warmup iterations).	7.85 ± 0.69 (1.00)	10.94 ± 0.36 (1.00)	7.08 ± 0.20 (1.00)	23.18 ± 0.71 (1.45)	8.39 ± 0.34 (1.00)	11.28 ± 0.43 (1.00)	8.02 ± 0.27 (1.00)	8.25 ± 0.27 (1.00)	7.85 ± 0.10 (1.00)	7.94 ± 0.27 (1.00)	7.51 ± 0.23 (1.00)	11.31 ± 0.59 (1.00)	13.82 ± 0.43 (1.00)	7.57 ± 0.15 (1.00)	7.55 ± 0.10 (1.00)	30.20 ± 0.62 (1.89)	13.60 ± 0.79 (1.00)	7.28 ± 0.14 (1.00)
remove row Duration to remove a row. (with 5 warmup iterations).	37.36 ± 2.95 (1.09)	61.72 ± 1.33 (1.79)	46.81 ± 0.98 (1.36)	54.56 ± 0.80 (1.59)	38.75 ± 0.75 (1.13)	44.01 ± 0.90 (1.28)	41.43 ± 1.66 (1.20)	51.66 ± 1.36 (1.50)	49.38 ± 2.35 (1.44)	67.79 ± 1.90 (1.97)	62.86 ± 2.29 (1.83)	66.01 ± 2.01 (1.92)	44.51 ± 0.41 (1.29)	48.10 ± 1.23 (1.40)	34.41 ± 1.28 (1.00)	76.90 ± 1.48 (2.24)	46.09 ± 0.43 (1.34)	38.23 ± 1.53 (1.11)
create many rows Duration to create 10,000 rows	1857.48 ± 7.96 (1.46)	1880.92 ± 34.50 (1.48)	2271.98 ± 18.13 (1.79)	1610.25 ± 31.52 (1.27)	1366.00 ± 8.70 (1.07)	1453.77 ± 36.54 (1.14)	1373.96 ± 7.52 (1.08)	2141.70 ± 5.10 (1.68)	1941.82 ± 67.09 (1.53)	2737.69 ± 47.75 (2.15)	2933.26 ± 48.58 (2.31)	1805.81 ± 11.77 (1.42)	3365.71 ± 27.76 (2.65)	3250.83 ± 10.60 (2.56)	1418.78 ± 7.76 (1.12)	2837.40 ± 32.85 (2.23)	1514.01 ± 9.09 (1.19)	1271.61 ± 9.44 (1.00)
append rows to large table Duration for adding 1000 rows on a table of 10,000 rows.	292.79 ± 5.92 (1.14)	299.99 ± 9.60 (1.17)	322.24 ± 6.19 (1.25)	500.52 ± 13.50 (1.95)	270.94 ± 6.74 (1.05)	340.23 ± 5.38 (1.32)	257.01 ± 5.24 (1.00)	332.76 ± 7.03 (1.29)	349.44 ± 6.56 (1.36)	419.58 ± 11.35 (1.63)	491.11 ± 8.04 (1.91)	326.55 ± 2.64 (1.27)	574.93 ± 32.65 (2.24)	286.44 ± 6.91 (1.11)	260.94 ± 4.65 (1.02)	662.45 ± 10.26 (2.58)	361.10 ± 3.18 (1.41)	271.47 ± 5.27 (1.06)
clear rows Duration to clear the table filled with 10,000 rows.	336.63 ± 7.52 (1.67)	291.63 ± 5.14 (1.45)	368.19 ± 5.72 (1.83)	257.14 ± 3.24 (1.28)	201.09 ± 1.60 (1.00)	230.39 ± 1.35 (1.15)	204.16 ± 3.50 (1.02)	271.58 ± 7.09 (1.35)	293.13 ± 4.22 (1.46)	390.70 ± 9.42 (1.94)	669.63 ± 8.84 (3.33)	423.06 ± 9.42 (2.10)	287.44 ± 4.07 (1.43)	350.18 ± 6.52 (1.74)	258.42 ± 2.77 (1.29)	319.47 ± 3.94 (1.59)	246.07 ± 3.18 (1.22)	204.93 ± 3.55 (1.02)
slowdown geometric mean	**1.18**	**1.29**	**1.39**	**1.39**	**1.04**	**1.12**	**1.06**	**1.28**	**1.25**	**1.50**	**1.84**	**1.35**	**1.46**	**1.38**	**1.05**	**2.01**	**1.16**	**1.02**

Figure 3: Performance benchmark of various JS libraries and frameworks, taken from http://www.stefankrause.net/js-frameworks-benchmark5/webdriver-ts/table.html

From Stefan Krause's study, we can see that Vue outperforms most major frameworks, including Angular 2 and React in most types of operations. As noted on the study's website:

"The benchmarks computes the geometric mean of the ratio between the durations of the benchmarks of the framework to the duration of the fastest framework for that benchmark. A factor X indicates that a frameworks takes on average X times as long as the fastest version (which is usually vanillajs)." (http://www.stefankrause.net/wp/?p=316)

This means that, although VanillaJS and Inferno were usually the most performant, VueJS is much faster at CRUD operations (reading, updating, appending, removing and writing rows to a table). Note: for instructions on running the benchmarks, see https://github.com/krausest/js-framework-benchmark.

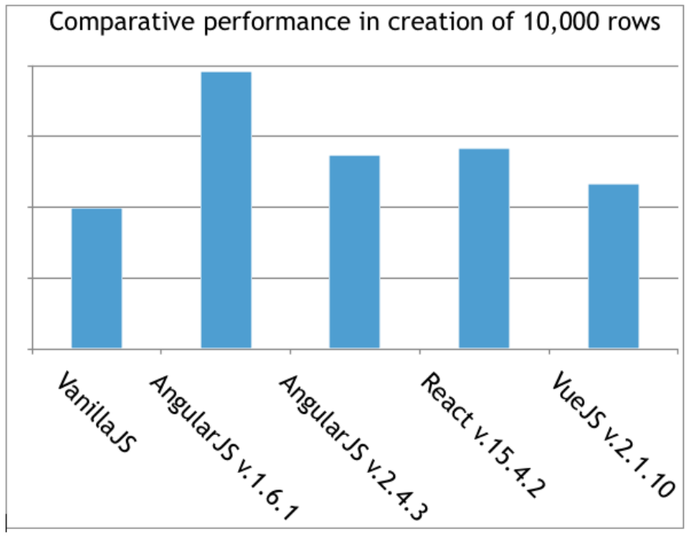


Chart 1: Comparative performance in creation of 10,000 rows. Results based on data from Frank Krause's JavaScript Framework performance benchmarks. (Round 4)

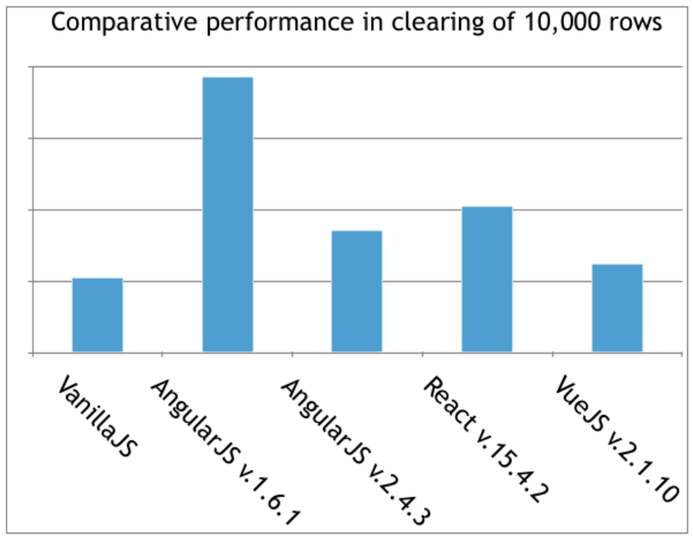


Chart 2: Comparative performance in clearing of 10,000 rows. Results based on data from Frank Krause's JavaScript Framework performance benchmarks (Round 5).

As can be seen in charts 1 and 2 above, VueJS outperforms Angular and React when it comes to the creation and clearing of elements. The framework is only beaten by VanillaJS. There do however exist operations for which React is faster than Vue. Namely, the appending of elements, partial updates and replacing of elements (see chart 3).

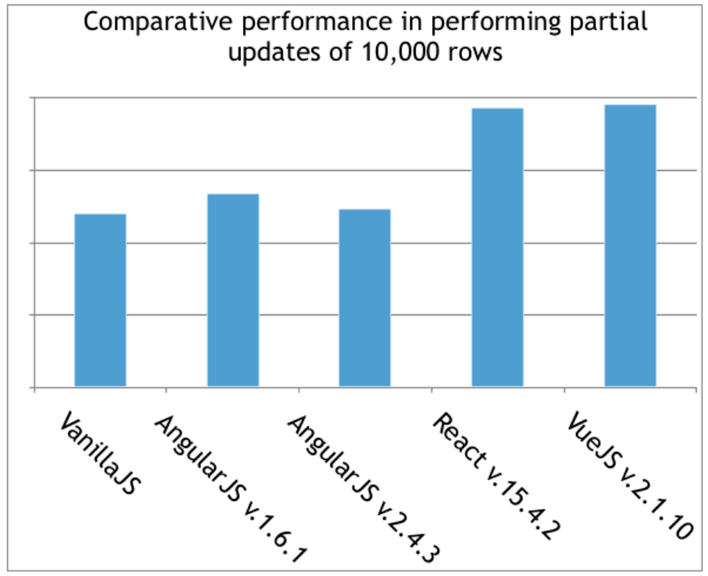


Chart 3: Comparative performance in performing partial updates of 10,000 rows. Results based on data from Frank Krause's JavaScript Framework performance benchmarks (Round 5).

The VueJS official website boasts similar performance difference between both React and AngularJS. Tested on a "Chrome 52 on a 2014 MacBook Air", the developers show a significant performance difference when rendering 10,000 list items 100 times:

	VueJS	React
Fastest	23ms	63ms
Median	42ms	81ms
Average	51ms	94ms
95th Perc.	73ms	164ms
Slowest	343ms	453ms

Table 3: Contrasting the rendering times of 10,000 list items (rendered 100 times) using both VueJS and React. Data taken from the official VueJS website: https://vuejs.org/v2/guide/comparison.html

Memory footprint

Another interesting comparison factor is the memory footprint across the different frameworks.

Krause's benchmark goes to show that, at roughly 3.6MB, VueJS has a much smaller read memory footprint (after page load) than both React (4.73MB), Angular 1 (5.18MB) and Angular 2 (5.86MB). Although this difference may be small, its significance becomes obvious when considering the memory usage after the addition of 1000 rows: AngularJS v.1.6.1 consumed 14.88MB; at 12.46MB Angular v.2.4.3 consumed only a little less than its predecessor. React v15.4.2 consumed 12.99MB whilst VueJS only required 8.89MB (see

chart 4 below).

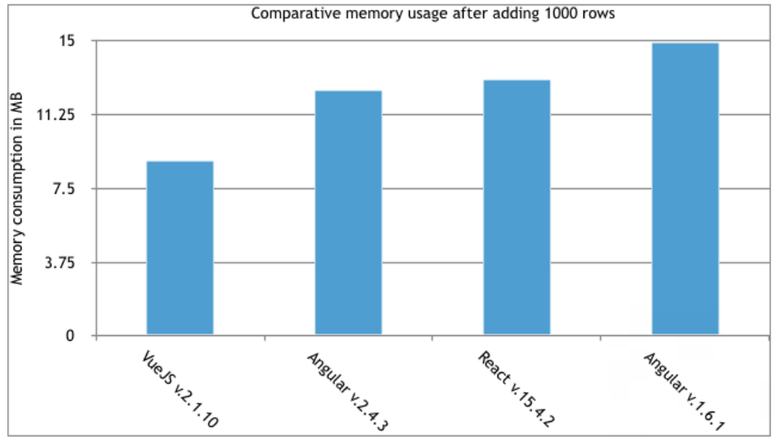


Chart 4: Comparative memory usage after adding 1000 rows. Results based on data from Frank Krause's JavaScript Framework performance benchmarks (Round 5).

Although not reflected in the benchmarks above, it is possible to further improve Vue's rendering times by utilizing its support for server-side rendering.

Server-side rendering means that the server will generate the HTML on-the-fly, reducing the number of requests required to be sent by the client (and hence speeding up loading times). A nice example of an app using server-side rendering in VueJS is the following HackerNews clone: https://github.com/vuejs/vue-hackernews-2.0/

Design flexibility

Much like React, VueJS, is not very "opinionated".

This means that the library does not dictate how it should be used, and how applications using it should be designed. Whilst other frameworks may force the developer to follow a certain pattern, VueJS does not.

For example, unlike Ember or Angular, VueJS does not force you to use modules or define a controller: instead, one just creates a VueJS object that is configured with the data and methods needed in the template.

Using an opinionated framework can be both advantageous and disadvantageous: it may save developers time by already having something that made design choices for them; or it can become a hurdle as the framework "locks" developers into doing things a specific way.

Functionality

Similar to React, VueJS focuses just on the development of the actual web-interface. Whilst it does this well, it does just that, and nothing more.

Much like React, VueJS does not come with additional tools and features. Anything outside VueJS' sole purpose must be achieved by either writing the code yourself, or using third-party libraries. The previously discussed alternatives to VueJS, namely AngularJS 1 and 2 (which are complete frameworks), offer full sets of additional functionality.

For example, AngularJS offers services for performing HTTP requests, promise handling, filters for formatting data, routing, a service for cookie handling, or a service wrapper for handling timeouts.

Size

The codebase of VueJS is much smaller and simpler than that of its competitors, such as AngularJS. VueJS 2.1.10 consists of a mere 8569 lines of code. This makes a large difference in terms of actual file size: minified VueJS 2.1.10 is a mere 74KB in size, compared to 168KB for AngularJS 1.6.1 or 520KB for Ember 2.12.0. However, in this comparison, React v.15.4.2 beats VueJS hands-down with 24KB in size.

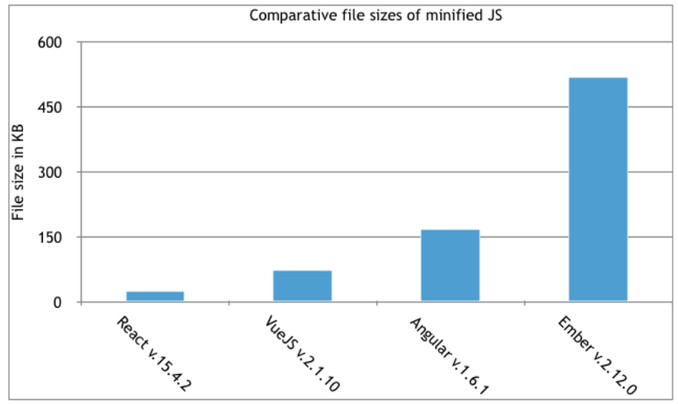


Chart 5: Comparative file sizes of minified JS

Learning curve

Another key advantage of VueJS is the fact that it works out of the box. One can load the library using a CDN and get to work immediately.

VueJS works perfectly well without having to use a build system. The same cannot be said for React or Angular 2. The latter, for example, comes with an entire command line interface tool that a developer must use to create, manage, build and deploy the project. The introduction of such tools can increase the learning curve.

Whilst such tools are necessary for large projects, forcing them upon the developer point-blank may be frustrating for some. Furthermore, learning Angular 2 requires first learning TypeScript - this might be a large stepping stone for some.

When it comes to actually learning how to use one of the aforementioned frameworks, there is little difference: **the overall design concepts between AngularJS, React and VueJS are all quite similar.**

All of the three frameworks/libraries make use of similar concepts, such as components. And whilst the exact definition may vary slightly, the big picture remains the same. If one already understands either React

or AngularJS, then grasping the basics behind VueJS will be a matter of a couple of hours.

Conclusion

In this article, we compared VueJS to three other popular JavaScript counterparts: AngularJS and React. The comparison took into account performance, functionality, design flexibility and the learning curve. In terms of the latter, we discovered that VueJS, React and AngularJS all share very similar concepts, and as such learning how to use VueJS will require minimal effort.

When considering performance, we showed that VueJS outperforms AngularJS, and, in most cases, React.

Lastly, VueJS is similar to React, in that it focuses on a core set of functionality, and as such offers fewer features than Angular. This has the result that VueJS does however provide developers with more design flexibility ■

Download the entire source code from GitHub at
bit.ly/dncm30-vuejs-vs-ang

Benjamin Jakobus
Author

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

Thanks to Ravi Kiran and Suprotim Agarwal for reviewing this article.