Issue 36 | May-June 2018

DNCMagazine

www.dotnetcurry.com

- C# 8.0 NEW PLANNED FEATURES
- AZURE CUSTOM VISION AND XAMARIN
- UNIT TESTING WITH VUE.JS
- ANGULAR HTTP CLIENT - ADVANCED SCENARIOS
- AOP IN C# VIA FUNCTIONS
- DYNAMIC BINDING IN C#
- DI TECHNIQUES IN ANGULAR
- TYPESCRIPT - A TOUR OF GENERICS

@SuprotimAgarwal

letter from the editor.

It's been a busy first May week in the tech world!

Microsoft and Google recently hosted their biggest tech conferences of the year - Build 2018 and Google I/O 2018, respectively. With the two conferences overlapping each other, the tech world couldn't help but compare the two. But let's not get into "the who did better" conversation.

With both the conferences giving competing views of how they will focus on Cloud, AI, ML and IoT, I feel there's much for the developer ecosystem to rejoice about.

Here are my top "developer" picks and annoucements from the Build conference:

> Azure commoditization, giving plenty of opportunities for developers and enterprises to tap into

> 'AI for Accessibility' with seed grants for startups, researchers and non-profits

> Azure Sphere, Microsoft Layout, AI-based speech API

> Azure IoT Edge for low-power devices to perform AI tasks locally

> Custom cognitive vision services running on Azure IoT Edge

> Open sourcing Azure IoT edge runtime

> Project Kinect for enabling Azure AI-enabled edge devices

> Captivating examples of innovative, data-driven, intelligent systems

> Connecting apps and platforms using Microsoft Graph, and the possibility to connect it to a Google graph

> Azure Blockchain Workbench

> .NET Core 3 and support for WPF, Windows Forms and UWP Xaml

> Microsoft offering developers as much as 95% cut from sales of their apps on the Microsoft Store

Amidst these exciting annoucements, the drumroll went to Amazon's Alexa when it stated "I like Cortana. We both have experience with light rings, although hers is more of a halo".

So which announcement got the developer in you excited about? Mail me your views at suprotimagarwal@dotnetcurry.com or tweet them to @dotnetcurry.

the team.

Editor In Chief : Suprotim Agarwal

Art Director : Minal Agarwal

Technical Reviewers : Yacoub Massad, Suprotim Agarwal, Ravi Kiran, Keerti Kotaru, Damir Arh

Contributing Authors : Yacoub Massad, Ravi Kiran, Keerti Kotaru, Gerald Versluis, David Pine, Daniel Jimenez Garcia, Damir Arh.

Next Edition : July 2018

et curry.com

THANK YOU

FOR THE 36th EDITION

@damirarh

@yacoubmassad

@dani_djg

@sravi_kiran

@davidpine7

@jfversluis

@keertikotaru

@suprotimagarwal

@saffronstroke

WRITE FOR US

mailto: suprotimagarwal@dotnetcurry.com

RAVENDB 4.0

Your Fully Transactional NoSQL Database

LEARN IT LIVE

Workshops '18

Join RavenDB CEO Oren Eini for a first look at the exciting features of newly released RavenDB 4.0. He will take you from the starting point, assuming this is your first experience with a NoSQL database, and rapidly ramp up your skills to use advanced functionality new to the world of NoSQL. This workshop is for developers and their operations teams who want to advance their expertise in RavenDB. You will discover how to:

SIGN UP AT

workshops.ravendb.net

- Set up and secure your database in minutes
- Establish constant availability by deploying nodes to a distributed cluster
- Achieve super-fast aggregation with Map-Reduce Indexes
- Code with RavenDB 4.0 and our SQL-like query language
- Query efficiently by taking advantage of intelligent Auto Indexes
- Model Data in a NoSQL Document Database

Hibernating Rhinos

www.ravendb.net

contents.

Damir Arh

The release cadence for C# has changed after the release of C# 7. New minor versions of the language are released as part of Visual Studio 2017 updates. Regardless of this release cadence, the C# team continues working on the next major version of the language – C# 8.0.

C# 8.0 New Planned Features

So far, all the new language features introduced in minor language versions of C# were designed so as not to have a large impact on the language. These features are mostly syntactic improvements and small additions to the new features introduced in C# 7.0.

This was an intentional decision and it's going to stay that way.

Larger features which require more work in all development phases (design, implementation and testing) are still only going to be released with major versions of the language. Although the final minor version of C# 7 has not been released yet, the team is already actively working on the next major version of the language: C# 8.0.

In this C# 8 tutorial, I am going to introduce a selection of currently planned features which will most probably end up in the final release. All of them are still early in development and are likely to change.

C# 8 – New Planned Features

Nullable Reference Types

This feature was already considered in the early stages of C# 7.0 development but was postponed until the next major version. Its goal is to help developers avoid unhandled `NullReferenceException` exceptions. The core idea is to allow variable type definitions to specify whether they can have `null` value assigned to them or not:

```
IWeapon? canBeNull;
IWeapon cantBeNull;
```

Assigning a `null` value or a potential `null` value to a non-nullable variable would result in a compiler warning (the developer could configure the build to fail in case of such warnings, to be extra safe):

```
canBeNull = null;       // no warning
cantBeNull = null;      // warning
cantBeNull = canBeNull; // warning
```

Similarly, warnings would be generated when dereferencing a nullable variable without checking it for `null` value first.

```
canBeNull.Repair();       // warning
cantBeNull.Repair();      // no warning
if (canBeNull != null) {
  cantBeNull.Repair();  // no warning
}
```

The problem with such a change is that it breaks existing code: it is assumed that all variables from before the change, are non-nullable. To cope with this, static analysis for null safety could be enabled selectively with a compiler switch at the project level.

The developer could opt-in to the nullability checking when he/she is ready to deal with the resulting warnings. This would be in best interest of the developer, as the warnings might reveal potential bugs in his code.

There's an early preview of the feature available to try out as a download for Visual Studio 2017 15.6 update.

Records

Currently, a significant amount of boilerplate C# code has to be written while creating a simple class which acts as a value container only and does not include any methods or business logic.

The records syntax should allow standardized implementation of such classes with absolute minimum code:

```
public class Sword(int Damage, int Durability);
```

This single line would result in a fully functional class:

```
public class Sword : IEquatable<Sword> {
  public int Damage { get; }
  public int Durability { get; }

  public Sword(int Damage, int Durability) {
    this.Damage = Damage;
    this.Durability = Durability;
  }

  public bool Equals(Sword other) {
    return Equals(Damage, other.Damage) && Equals(Durability, other.Durability);
  }

  public override bool Equals(object other) {
    return (other as Sword)?.Equals(this) == true;
  }

  public override int GetHashCode() {
    return (Damage.GetHashCode() * 17 + Durability.GetHashCode());
  }

  public void Deconstruct(out int Damage, out int Durability) {
    Damage = this.Damage;
    Durability = this.Durability;
  }

  public Sword With(int Damage = this.Damage, int Durability = this.Durability) =>
    new Sword(Damage, Durability);
}
```

As you can see, the class features *read-only* properties, and a constructor for initializing them. It implements value equality and overrides `GetHashCode` correctly for use in hash-based collections, such as `Dictionary` and `Hashtable`. There's even a `Deconstruct` method for deconstructing the class into individual values with the tuple syntax:

```
var (damage, durability) = sword;
```

You probably don't recognize the syntax used in the last method of the class.

Method parameter's default argument will additionally allow referencing a class field or property using such syntax. This is particularly useful for implementing the `With` helper method dedicated to creating modified copies of existing immutable objects, e.g.:

```
var strongerSword = sword.With(Damage: 8);
```

Additionally the `with` expression syntax is considered as well, as syntactic sugar, for calling this method:

```
var strongerSword = sword with { Damage = 8 };
```

Recursive Patterns

Some pattern matching features have already been added to C# in version 7.0. There are plans to further extend the support in C# 8.0.

Three new features are currently planned:

- **Recursive patterns** will allow deconstruction of matched types in a single expression. It should work well with the compiler generated Deconstruct() method for records:

```
if (sword is Sword(10, var durability)) {
  // code executes if Damage = 10
  // durability has value of sword.Durability
}
```

- **Tuple patterns** will allow matching of more than one value in a single pattern matching expression:

```
switch (state, transition) {
  case (State.Running, Transition.Suspend):
   state = State.Suspended;
   break;
}
```

- An **expression version of the switch statement** will allow terser syntax when the only result of pattern matching is assigning a value to a single variable. The syntax is not yet finalized but the current suggestion is as follows:

```
state = (state, transition) switch {
  (State.Running, Transition.Suspend) => State.Suspended,
  (State.Suspended, Transition.Resume) => State.Running,
  (State.Suspended, Transition.Terminate) => State.NotRunning,
  (State.NotRunning, Transition.Activate) => State.Running,
  _ => throw new InvalidOperationException()
};
```

Default Interface Methods

Interfaces in C# are currently not allowed to contain method implementations. They are restricted to method declarations:

```
interface ISample {
  void M1();                                    // allowed
  void M2() => Console.WriteLine("ISample.M2"); // not allowed
}
```

To achieve similar functionality, abstract classes can be used instead:

```
abstract class SampleBase {
  public abstract void M1();
  public void M2() => Console.WriteLine("SampleBase.M2");
}
```

In spite of this, there are plans to add support for default interface methods to C# 8.0, i.e. method implementations using the syntax suggested in the first example above. This would allow scenarios not supported by abstract classes.

A library author could **extend an existing interface** with a default interface method implementation, instead of a method declaration.

This would have the benefit of not breaking existing classes, which implemented an older version of the interface. If they didn't implement the new method, they could still use the default interface method implementation. When they wanted to change that behavior, they could override it, but with no code change just because the interface was extended.

Since multiple inheritance is not allowed, a class can only derive from a single base abstract class.

In contrast to that limitation, a class can implement multiple interfaces. If these interfaces implement default interface methods, this effectively allows classes to **compose behavior from multiple different interfaces** – a concept known as trait and already available in many programming languages.

Unlike multiple inheritance, this feature avoids the so called diamond problem of ambiguity when a method with the same name is defined in multiple interfaces. To achieve that, C# 8.0 will require each class and interface to have a most specific override for each inherited member:

- When a member with the same name is inherited from multiple interfaces, one override is more specific than the other when its interface is derived from the other one.

- When neither interface directly or indirectly inherits from the other interface, the developer will need to specify the override he/she wants to use or write his own override. By doing so, he/she will explicitly resolve the ambiguity.

Asynchronous Streams

C# already has support for iterators and asynchronous methods. In C# 8.0, it's planned to combine the two in asynchronous iterators. They would be based on the asynchronous versions of `IEnumerable` and `IEnumerator` interfaces:

```
public interface IAsyncEnumerable<out T> {
  IAsyncEnumerator<T> GetAsyncEnumerator();
}

public interface IAsyncEnumerator<out T> : IAsyncDisposable {
  Task<bool> MoveNextAsync();
  T Current { get; }
}
```

Additionally, an asynchronous version of `IDisposable` interface would be required for consuming the asynchronous iterators:

```
public interface IAsyncDisposable {
  Task DisposeAsync();
}
```

This would allow the following code to be used for iterating over the items:

```
var enumerator = enumerable.GetAsyncEnumerator();
try {
  while (await enumerator.WaitForNextAsync()) {
    while (true) {
      Use(enumerator.Current);
    }
  }
}
```

```
finally {
  await enumerator.DisposeAsync();
}
```

It's very similar to the code we're currently using for consuming regular synchronous iterators. However, it does not look familiar because we typically just use the `foreach` statement instead. An asynchronous version of the statement would be available for asynchronous iterators:

```
foreach await (var item in enumerable) {
  Use(item);
}
```

Just like with the `foreach` statement, the compiler would generate the required code itself.

It would also be possible to implement asynchronous iterators using the `yield` keyword, similar to how it can currently be done for synchronous iterators:

```
async IAsyncEnumerable<int> AsyncIterator() {
  try {
    for (int i = 0; i < 100; i++)
    {
      yield await GetValueAsync(i);
    }
  }
  finally {
    await HandleErrorAsync();
  }
}
```

Support for cancellation tokens and LINQ is considered as well.

Ranges

There are plans to add new syntax for expressing a range of values:

```
var range = 1..5;
```

This would result in a `struct` representing the declared range:

```
struct Range : IEnumerable<int> {
  public Range(int start, int end);
  public int Start { get; }
  public int End { get; }
  public StructRangeEnumerator GetEnumerator();
  // overloads for Equals, GetHashCode...
}
```

The new type could be effectively used in several different contexts:

- It could appear as an argument in indexers, e.g. to allow a more concise syntax for slicing arrays:

```
Span<T> this[Range range] {
  get
  {
    return ((Span<T>)this).Slice(start: range.Start, length: range.End - range.
```

```
    Start);
  }
}
```

- Since the struct implements the `IEnumerable` interface, it could be used as an alternative syntax for iterating over a range of values:

```
foreach (var index in min..max) {
  // process values
}
```

- Pattern matching could take advantage of the syntax for matching a value to a specified range:

```
switch (value) {
  case 1..5:
    // value in range
    break;
}
```

It's still undecided whether the range operator would be inclusive or exclusive, i.e. whether the resulting range would include the `End` value or not. It's even possible that both an inclusive and an exclusive syntax will be available.

Generic Attributes

Support for generic attributes would allow nicer syntax for attributes which require a type as their argument. Currently, types can only be passed to attributes using the following syntax:

```
public class TypedAttribute : Attribute {
  public TypedAttribute(Type type)
  {
    // ...
  }
}
```

With generic attributes, the type could be passed in as a generic argument:

```
public class TypedAttribute<T> : Attribute {
  public TypedAttribute()
  {
    // ...
  }
}
```

Apart from the syntax being nicer, this would also allow type checking of attribute arguments which must match the supplied type, e.g.:

```
public TypedAttribute(T value) {
  // ...
}
```

Default Literal in Deconstruction

To assign default values to all members of a tuple in C# 7, the tuple assignment syntax must be used:

```
(int x, int y) = (default, default);
```

With support for `default` literal in deconstruction syntax, the same could be achieved with the following syntax:

```
(int x, int y) = default;
```

Caller Argument Expression

In C# 5, caller information attributes (`CallerMemberName`, `CallerFilePath` and `CallerLineNumber`) were introduced into the language, so that the called method could receive more information about the caller for diagnostic purposes.

The `CallerMemberName` attribute even turned out to be very useful for implementing the `INotifyPropertyChanged` interface:

```
class ViewModel : INotifyPropertyChanged {
  public event PropertyChangedEventHandler PropertyChanged;

  private void OnPropertyChanged([CallerMemberName] string propertyName = null)
  {
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
  }

  private int property;

  public int Property
  {
    get { return property; }
    set
    {
      if (value != property)
      {
        property = value;
        OnPropertyChanged();
      }
    }
  }
}
```

C# 8 might additionally support a similar `CallerArgumentExpression` attribute which would set the target argument to a string representation of the expression that was passed in as the value for another argument in the same method call:

```
public Validate(int[] array, [CallerArgumentExpression(“array”)] string
arrayExpression = null)
{
  if (array == null)
  {
    throw new ArgumentNullException(nameof(array), $”{arrayExpression} was null.”);
  }
  if (array.Length == 0)
  {
    throw new ArgumentException($”{arrayExpression} was empty.”, nameof(array));
  }
}
```

Just like with the caller information attributes, the compiler would set the variable value at the call site to

the correct literal value in the compiled code.

Target-typed new Expression

When declaring local variables, the `var` keyword can already be used to avoid repeating (potentially long) type names in code:

```
Dictionary<string, string> dictionary = new Dictionary<string, string>(); //
without var keyword
var dictionary = new Dictionary<string, string>(); // with var keyword
```

The same approach cannot be used for class members (e.g. fields) because they require the type to be explicitly stated:

```
class DictionaryWrapper {
  private Dictionary<string, string> dictionary = new Dictionary<string, string>();
  // ...
}
```

The target-typed `new` expression, as planned for C# 8, would allow an alternative shorter syntax in such cases:

```
class DictionaryWrapper {
  private Dictionary<string, string> dictionary = new();
  // ...
}
```

The syntax would of course not be limited to this context. It would be allowed wherever the target type could be implied by the compiler.

Ordering of ref and partial Modifiers on Type Declarations

C# currently requires the `partial` keyword to be placed directly before the `struct` or `class` keyword. Even with the introduction of the `ref` keyword for stack-allocated structs in C# 7.2, the restriction for the `partial` keyword remained in place, therefore the `ref` keyword must be placed directly before the `struct` keyword if there is no `partial` keyword; and directly before the `partial` keyword, if the latter is present.

Hence, these are the only two valid syntaxes:

```
public ref struct NonPartialStruct { }
public ref partial struct PartialStruct { }
```

In C# 8, this restriction is planned to be relaxed, making the following syntax valid as well:

```
public partial ref struct PartialStruct { }
```

Conclusion:

There are many new features already in the works for C# 8. This tutorial does not list all of them. However, we're probably still quite far away from the final release of C# 8.0, as the release date has not been announced yet. Until then, we can expect the plans to change: not all features might make it into

the release. And even those that will make it to the final release, might still change syntactically or even semantically.

Of course, other new features not mentioned in the article might be added to the language as well. With all that in mind, you should regard the information in this article only as an interesting glimpse into the potential future of the language ■

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

ASPECT ORIENTED PROGRAMMING IN C#
via Functions

In this article I will describe a new approach for creating and applying aspects in C# which I call AOP via functions

Introduction

In a previous article, Aspect Oriented Programming (AOP) in C# with SOLID, I talked about AOP and how it helps with reducing code duplication and code tangling that is encountered while addressing cross-cutting concerns. I also mentioned different approaches for doing AOP in C#. These approaches are:

1. AOP by using the `IQueryHandler` and `ICommandHandler` abstractions. I will call this approach the *Command Query Separation* approach or simply the CQS approach.
2. AOP via dynamic proxies
3. AOP via compile-time weaving
4. AOP via T4

NOTE: Please refer to the mentioned article for more details about AOP and these approaches. I recommend that the reader reads that article before reading this once - at least read it up to the "The First Problem: Specificity" section. This will enable me to assume that you already understand the CQS approach to AOP.

In a nutshell, we can model all behavior in the system as classes that implement either the `IQueryHandler<TQuery, TResult>` generic interface or the `ICommandHandler<TCommand>` generic interface. Then, to create an aspect, we create a generic decorator for each one of these two interfaces. To apply the aspect to an object, we simply decorate it using the aspect decorator.

For a complete example on how to do this, please refer to the mentioned article Aspect Oriented Programming (AOP) in C# with SOLID.

There are two problems with the CQS approach:

1. For each aspect, we need to create two decorators instead of one. These two decorators are almost identical. This is not a big problem though.

2. Verbosity - For example, in the case of a command handler, for each behavior class, we need to create a data object to represent a command. In the case of a query handler, we need two data objects to represent the query and the query result.

Additionally, when a behavior class has a dependency on another handler, it looks like this:

```
private readonly IQueryHandler<ParseQuery, ParseResult> parser;
```

..instead of simply:

```
private readonly IParser parser;
```

The same thing is also true for when we specify the implemented interface when we define a behavior class. That is, we specify the implemented interface like this:

```
public class CustomerReportGenerator
    : IQueryHandler<CustomerReportQuery, CustomerReportResult>
{

}
```

..instead of simply doing this:

```
public class CustomerReportGenerator : ICustomerReportGenerator
{

}
```

When invoking a dependency, it would be much simpler to do the following:

```
var result = parser.Parse(document);
```

..instead of doing the following:

```
var result = parser.Handle(new ParseQuery(document));
```

Still, the CQS approach to AOP has many advantages:

1. It requires no special tools.
2. Writing an aspect is done simply by writing two decorators. (compare this to writing the aspect in T4 for example)
3. Applying an aspect to an object is done simply by decorating the object.
4. It enables the aspect to have access to input/output data in a strongly typed way. I will explain this point in more details later in this article after I speak about AOP via functions.

In the next sections I will talk about an approach to AOP that has most of the advantages of the CQS approach, but that attempts to solve its issues.

AOP via functions

A function in mathematics is a relation between a set of inputs and a set of outputs.

In programming, a function is a unit of code that takes some input and produces some output. In .NET, we can use the Func delegate to represent a function. For example, `Func<Customer, Discount>` represents a function that takes a Customer data object as input and returns a Discount data object as output. We can also create a special `IFunction<TInput,TOutput>` interface to represent a function like this:

```
public interface IFunction<in TInput, out TOutput>
{
  TOutput Invoke(TInput input);
}
```

This generic interface has a single method named `Invoke` that takes a single parameter of type `TInput` and returns a value of type `TOutput`.

If all our behavior classes implement this interface (similar to the CQS approach), then we can create an aspect by simply creating a single decorator for `IFunction<TInput,TOutput>`. This would solve the first disadvantage I mentioned above about the CQS approach. Still this approach would be as verbose (the second disadvantage) as the CQS approach, if not more.

Instead I propose the following:

- Create aspects as decorators for `IFunction<TInput,TOutput>`

- To apply the aspect to a behavior object that implements some interface, say `IParser`, we first convert `IParser` to `IFunction`, we then decorate the resulting function, then we convert the result back to `IParser`.To convert from and to `IFunction`, we can create two adapters. One adapter would implement `IFunction` and take a dependency of type `IParser`, and the second adapter would implement `IParser` and take a dependency of type `IFunction`.

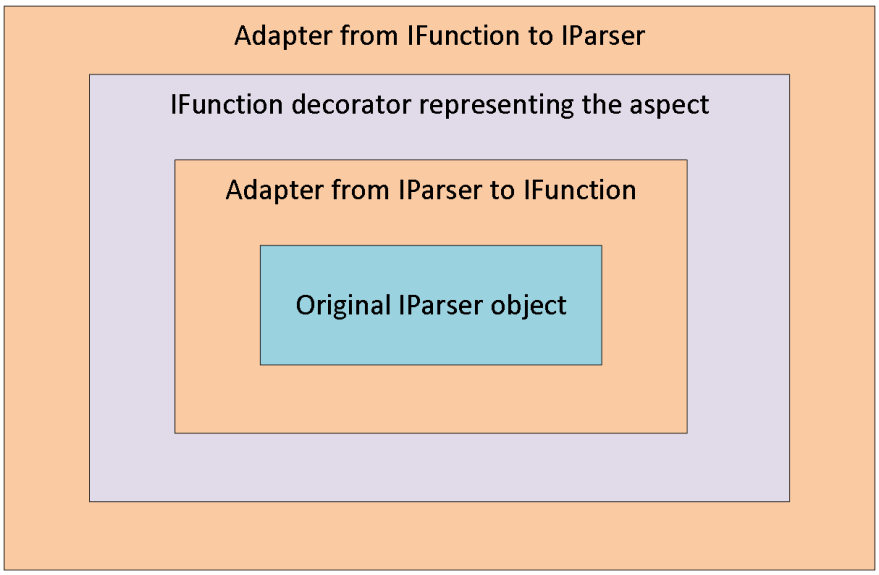


Figure 1: applying an aspect to an object of type IParser

Figure 1 shows how an original `IParser` object was first adapted to `IFunction`, then the adapter was decorated by the aspect decorator, which was then adapted back to `IParser`.

The code to apply the aspect would look like something like this:

```
IParser originalParser = ...

IParse parserWithAspectApplied =
  new FunctionToParserAdapter(
    new SomeAspect( //The aspect is applied here
      new ParserToFunctionAdapter(originalParser)));
```

`SomeAspect` represents an aspect. It is a generic decorator class that implements the generic `IFunction` interface, i.e., `IFunction<TInput,TOutput>`. It also has a dependency of type `IFunction<TInput,TOutput>`. I will show you an example of aspects later in this article.

While this approach may fix the issues with the CQS approach, there are still some issues that we need to address:

- How to convert any interface to `IFunction`? For example, while some methods return a specific type that we can use for `TOutput`, others return `void`. What would be the type of `TOutput` in this case? Also, methods can have multiple parameters, what would be the type of `TInput` of this case? What about interfaces with multiple methods?

- Assuming we know how to map between any interface and `IFunction`, creating the adapters is a time-

consuming process and is hard in many cases. Also, such adapters are code that we need to maintain. Any plausible approach to AOP must allow developers to apply aspects in a practical way.

- The code for applying the aspect is also verbose.
- What about performance? How much performance are we sacrificing by having these adapters?

I will address these issues next.

Mapping between any method and IFunction

Let's assume for a moment that all interfaces have a single method. We will deal with multi-method interfaces later.

To create the two adapters between some single-method interface and `IFunction`, we need to consider the following:

1. The method either returns a value of some type. Or its return type is `void`.
2. The method has either no parameters, a single parameter, or multiple parameters.

For methods that return a specific type, `TOutput` would simply be that type. For methods that return `void`, we can use a special type called the Unit type as `TOutput`.

Unit is simply a type that contains no data. It's only purpose is to represent the output of functions that actually have no output. You can read more about this type here: https://en.wikipedia.org/wiki/Unit_type

For methods that take no parameters, we can use Unit again for `TInput`. If the method takes a single parameter, we can use the type of this parameter as `TInput`. If the method takes more than one parameter, we can use a tuple as `TInput`. Value tuples introduced in C#7 are very convenient to use in this case.

Let's look at some examples:

```
public interface IEmailSender
{
  void SendEmail(
    EmailAddress from, EmailAddress to, string subject, EmailBody body);
}
```

This `SendEmail` method has four parameters but returns nothing. The `IEmailSender` interface would be adapted to this version of `IFunction`:

```
IFunction<(EmailAddress from, EmailAddress to, string subject, EmailBody body),
Unit>
```

`TInput` in this case is a C#7 tuple consisting of four items representing the original parameters, and `TOutput` is Unit because the original method returned void.

Here is how the adapter from `IEmailSender` to `IFunction` looks like:

```
public class EmailSenderToFunctionAdapter
: IFunction<(EmailAddress from, EmailAddress to, string subject, EmailBody body),
Unit>
{
  private readonly IEmailSender instance;

  public EmailSenderToFunctionAdapter(IEmailSender instance)
  {
    this.instance = instance;
  }

  public Unit Invoke(
  (EmailAddress from, EmailAddress to, string subject, EmailBody body) input)
  {
    this.instance.SendEmail(input.from, input.to, input.subject, input.body);
    return Unit.Default;
  }
}
```

This adapter takes `IEmailSender` as a dependency in the constructor and implements `IFunction`. When the `Invoke` method is called, the `IEmailSender` dependency is invoked. Then to satisfy the signature of method, we return `Unit.Default`. Unit.Default is a static property of the `Unit` type that returns the only instance of the `Unit` type. Here is how the `Unit` type looks like:

```
public class Unit
{
  private Unit()
  {

  }

  public static Unit Default { get; } = new Unit();
}
```

To adapt back from `IFunction` to `IEmailSender`, we create the following adapter:

```
public class FunctionToEmailSenderAdapter : IEmailSender
{
  private readonly IFunction<(EmailAddress from, EmailAddress to, string subject,
  EmailBody body), Unit> function;

  public FunctionToEmailSenderAdapter(
  IFunction<(EmailAddress from, EmailAddress to, string subject, EmailBody body),
  Unit> function)
  {
    this.function = function;
  }

  public void SendEmail(EmailAddress from, EmailAddress to, string subject,
  EmailBody body)
  {
    function.Invoke((from, to, subject, body));
  }
}
```

By the way, there is an interesting article series by Mark Seemann called Software Design Isomorphisms where he talks about how some concepts in programming can be expressed in multiple ways.

In particular, the *Unit isomorphisms* article talks about using the Unit type as a different way to represent

the return type for methods that return nothing. And the *Argument list isomorphisms* article talks about different ways to represent an argument list.

You can find the article series here: http://blog.ploeh.dk/2018/01/08/software-design-isomorphisms/

The InterfaceMappingHelper Visual Studio extension

Creating the adapters manually is a hard process. You need first to figure out the correct values for `TInput` and `TOutput`, and then you need to write two adapters correctly. This gets even harder if the original interface itself is generic. It could also have type parameter constraints. What if the developer decided to add a new parameter to the interface method? He/she needs to update the adapters accordingly.

To make it easy to generate and maintain these adapters, I created a Visual Studio extension called InterfaceMappingHelper. You can download this extension from inside Visual Studio from the Extensions and Updates tool found in the Tools menu.

Once installed, you can hover over any interface, click on the Rolsyn based refactorings bulb icon, and then click on the "Create interface-mapping extension methods class".

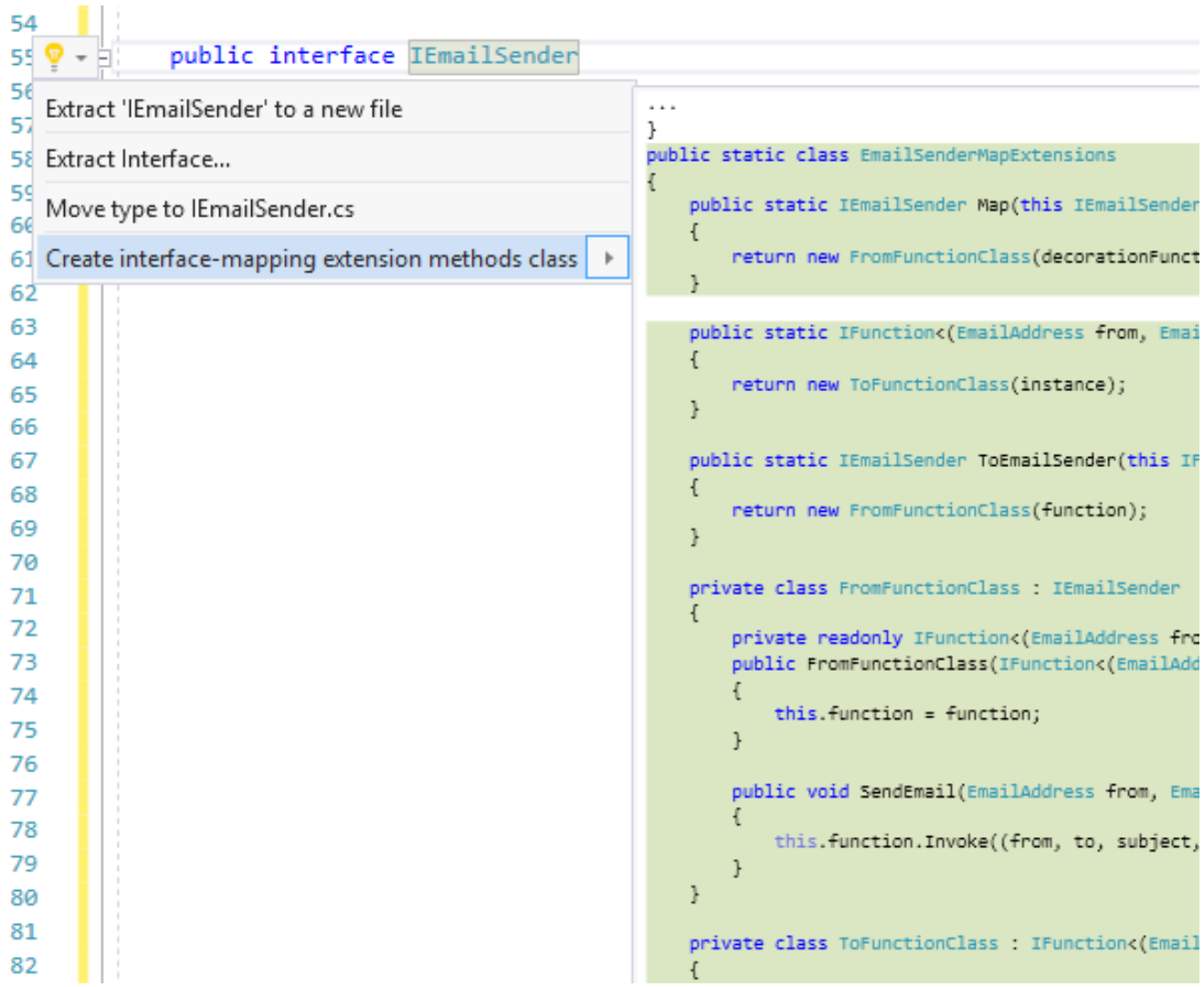


Figure 2 The InterfaceMappingHelper extension

This will create a static class right after the interface. This static class will have five members:

1. A static extension method called `Map`. I will talk about this method in the next section.

2. A private (nested) class representing the adapter from the original interface to its `IFunction` equivalent. This class is called `ToFunctionClass`.

3. Another private class representing the adapter from `IFunction` to the original interface. This class is called `FromFunctionClass`.

4. A static extension method called `ToFunction`. This method extends the original interface to allow you to adapt to `IFunction` by simply invoking `.ToFunction()`.

5. A static extension method that named ToXXXXXX where XXXXXX is the name of the original interface with the potential prefix "I" removed from the interface name. In the example we just saw, the method name is `ToEmailSender`. This method is an extension method of `IFunction`. It makes it easier to adapt back from `IFunction` to the original interface.

Next, I will talk about how the Map method can be used to apply an aspect easily.

The Map method

The `Map` method generated by the extension has the following signature:

```
public static IOriginalInterface Map(
  this IOriginalInterface instance,
  Func<IFunction, IFunction> decorationFunction)
```

`IOriginalInterface` is the name of the original interface. I use `IFunction` here as an abbreviation for `IFunction<TInput, TOutput>` for whatever `TInput` and `TOutput` that match the method signature in the interface. Here is how the Map method generated for the `IEmailSender` interface looks like:

```
public static IEmailSender Map(
  this IEmailSender instance,
  Func<
   IFunction<(EmailAddress from, EmailAddress to, string subject, EmailBody body),
   Unit>,

   IFunction<(EmailAddress from, EmailAddress to, string subject, EmailBody body),
   Unit>
  >
  decorationFunction)
{
  return new FunctionToEmailSenderAdapter(
    decorationFunction(
      new EmailSenderToFunctionAdapter(instance)));
}
```

The Map method is an extension method for the original interface, and its return type is also the original interface. It takes a `Func` from `IFunction` to `IFunction`.

So basically, this method allows you transform an object implementing some interface by telling it how to transform a function that somehow looks like it.

For example, we can use the Map method to apply an aspect (a class that decorates the generic `IFunction`

interface) like this:

```
IParser originalParser = ...

IParser parserWithAspectApplied =
  originalParser.Map(x => x.ApplySomeAspect());
```

where `ApplySomeAspect` is an extension method that decorates `IFunction`.

Let's see a real example.

I have created a repository on Github called AOPViaFunctionsExamples, you can find it here.

Let's first look at the RetryAspect class. The retry aspect can be applied to behavior units to make them retry the operation in case of failure.

The `RetryAspect` static class contains two members; a nested private Decorator class, and static extension method called `ApplyRetryAspect`.

Take a look at the nested Decorator class. This generic class decorates `IFunction<TInput, TOutput>`. It takes three parameters in the constructor; the `IFunction<TInput, TOutput>` instance to decorate, the number of times to try/retry the operation, and a `Timespan` representing the time that we should wait before we retry the operation.

The `ApplyRetryAspect` static method is an extension method for `IFunction<TInput, TOutput>`. This method is created for convenience as it will make applying the aspect much easier.
Next, take a look at the IEmailSender and IReportGenerator interfaces. I have already used the `InterfaceMappingHelper` extension to generate the mapping static class for each of these interfaces. You can see each one of these mapping classes immediately after each of the corresponding interface.

I have created very simple implementations of these interfaces. Take a look at the EmailSender, and the ReportGenerator classes.

Next, go to the Main method in the Program class to see how the aspect is applied. Look for example how I create an instance of the `EmailSender` class, and then call the `Map` method on it as an extension method. Notice how the Map method allows me to call the ApplyRetryAspect extension method on the lambda parameter which is of type `IFunction`.

I hope that you see how easy it is to apply an aspect using this approach.

Multi-method interfaces

So far, we have been dealing with interfaces that have a single method. Although when we apply the SOLID principles (the SRP and ISP principles in particular), most interfaces will have a single method, still some interfaces will have more than one method. Any practical approach to AOP should support multi-method interfaces.

To apply aspects to multi-method interfaces, we first adapt the original interface to multiple functions. We then apply the aspects to the individual functions. Then, we adapt these functions back to the original interface.

The `InterfaceMappingHelper` extension supports multi-method interfaces. It will generate all the necessary adapters and the `Map` method. Consider this interface that contains two methods:

```
public interface ITwoMethodsInterface {
  void Method1();
  void Method2();
}
```

Here is the signature of the generated Map method:

```
public static ITwoMethodsInterface Map(
  this ITwoMethodsInterface instance,
  Func<
    (IFunction<Unit, Unit> method1, IFunction<Unit, Unit> method2),
    (IFunction<Unit, Unit> method1, IFunction<Unit, Unit> method2)
  > decorationFunction)
```

Notice how the `Func` parameter of the `Map` method maps between a tuple containing two functions (method1 and method2) to a tuple containing two functions. This allows the caller to apply any aspects to each of the methods. For example, you might decide that you want to apply some aspect to the first method, but leave the second method as is. Consider this example:

```
ITwoMethodsInterface instance =
 new TwoMethodsClass()
  .Map(methods => (
   methods.method1.ApplyRetryAspect(
    numberOfTimesToRetry: 4,
     waitTimeBeforeRetries: TimeSpan.FromSeconds(5)),
      methods.method2));
```

In the above example, we call the map method on an instance of a `TwoMethodsClass` class (which implements the `ITwoMethodsInterface` interface). Notice how we apply the retry aspect on method1 but leave method2 intact.

Performance

In this section, I will examine the performance implications of the AOP via functions approach.

I will compare three approaches to AOP:

- The CQS approach
- AOP by using DynamicProxy
- The AOP via Functions approach

The aspect that we will apply will be a do-nothing aspect. That is, it simply calls the original object immediately.

The cost of the aspect should be measured relatively to the cost of the original operation. Here are the operations that I will consider:

- An operation that immediately returns some value.
- An operation that does some relatively complex mathematical operations. In specific, the operation calculates $\frac{\sum_{x=0}^{99} x^2}{100}$
-

- An operation that uses entity framework to read data from a single row table.

AOP approach	Operation	Mean	Error
No aspect/Direct call	Return value immediately	5.910 ns	0.0122 ns
	Math operation	4,397.312 ns	12.1029 ns
	Single row database access	333,368.087 ns	6,380.3390 ns
AOP via functions	Return value immediately	10.092 ns	0.0678 ns
	Math operation	4,391.248 ns	5.9884 ns
	Single row database access	334,411.618 ns	6,615.7627 ns
DynamicProxy	Return value immediately	39.987 ns	0.1860 ns
	Math operation	4,439.392 ns	16.4938 ns
	Single row database access	339,364.239 ns	6,706.4756 ns
CQS directly (without applying an aspect)	Return value immediately	10.551 ns	0.0551 ns
	Math operation	4,404.383 ns	13.0967 ns
	Single row database access	336,004.635 ns	4,839.2869 ns
CQS with aspect	Return value immediately	14.127 ns	0.2420 ns
	Math operation	4,388.645 ns	9.1133 ns
	Single row database access	330,025.458 ns	6,529.9339 ns

Note: *ns means nanosecond*

Consider the following:

- For any non-trivial operation, e.g. the operation that did some mathematical calculations and the database access operation, any AOP approach does not add any significant overhead relatively and its performance cost can be ignored. I think that the error in measurement is actually more than the overhead introduced. I think that this is the most important result.

- For very simply operations, e.g. the operation that simply returns a value immediately, AOP via functions preforms well compared to other approaches.

- AOP via functions approach seems to be as fast as the CQS approach or even faster. This was against my expectations before measurements. The reason might be that when invoking a query handler in the CQS approach, we have to create instances of the query and query result objects (this comparison is only relevant for the very simply operation).

More examples

In this section I will show two examples: an authorization aspect example, and a logging aspect example.

Authorization aspect example

Let's consider an authorization aspect example. Let's say we are developing a public API for some file system service that allows clients to manage files in some remote files system. Consider the following interface:

```
public interface IFileSystemService {
```

```
  FileContent GetFileContent(FileId fileId);
  FolderContents GetFolderContents(FolderId folderId);
  void CopyFileToFolder(FileId fileId, FolderId folderId);
}
```

This interface has three methods, one to get the contents of a file, one to get the contents of a folder, and one to copy some file into some folder.

Consider the following authorization rules:

- `GetFileContent` requires that the caller has read access to the file.
- `GetFolderContents` requires that the caller has read access to the folder.
- `CopyFileToFolder` requires that the caller has both read access to the file and write access to the folder.

Let's look at the authorization aspect. Take a look at the AuthorizationAspect class. The ApplyAuthorizationAspect extension method has three parameters:

1. The decorated parameter representing the function we are trying to decorate.

2. The authorizer parameter is an instance of the `IResourceAuthorize<TResourceId>` interface where `TResourceId` is a type parameter representing the type of the resource id. For example, this could be `FileId` or `FolderId`. The aspect would invoke such authorizer to determine if the current user has access to the specified resource.

3. The `resourceIdSelector` parameter is of type `Func<TInput, TResourceId>` and allows the caller to tell the aspect how to get the resource id from the input. For example, this could be used to select the `FolderId` parameter.

Take a look at the `Main` method in Program.cs. Take a look at the statement where I apply the authorization aspect on an instance of the `FileSystemService` class. See how I call the `Map` method to apply the authorization aspect on each of the three methods. For the `GetFileContent` method, I apply the aspect specifying the input itself as the resource id. Remember that for single-parameter methods, `TInput` is the type of a single parameter. For `GetFolderContents`, I do a similar thing. What I do for `CopyFileToFolder` is more special, though. Notice how I apply the aspect twice, once to make sure that the caller has read access to the file, and another time to make sure the caller has write access to the folder. Notice how I select the resource identifier in each case. Remember that `TInput` in this case is a tuple, i.e., (FileId fileId, FolderId folderId).

This is what I meant when I said that one of the advantages of the CQS approach to AOP is that it enables the aspect to have access to input/output data in a strongly typed way.

Logging aspect example

Let's take a look at another example: Logging. Take a look at the static LoggingAspect class. The nested Aspect class takes four dependencies in the constructor:

- The decorated function

- A dependency of type `ILogger`. This is the logger that will receive the logging data and record them

somewhere. As this is just an example, I have created a simple implementation that simply writes to the console; the ConsoleLogger class.

- A dependency of type `ILoggingDataExtractor<TInput, TOutput>`. Notice how this interface is generic. Take a look at the definition of this interface. It has three methods; a method to extract logging data from the input, a method to extract logging data from the output, and a method to extract logging data from exceptions in the case of error. Being generic, this will enable interesting ways of selecting logging data. I will show you this very soon.

- Logging data to always include. This allows us to supply constant logging data at the time of applying the aspect.

Look now at the ApplyLoggingAspect extension method in the LoggingAspect static class. There are two overloads of this method for convenience. One interesting thing to note about this method is that the last parameter is of type `Action<LoggingDataExtractorBuilder<TInput, TOutput>>`. This allows for an interesting way of selecting logging data when applying the aspect. Take a look at the `LoggingDataExtractorBuilder<TInput, TOutput>` class if you want. But I am more interested in showing you an example of applying the logging aspect. Take a look at the `Main` method in Program.cs. Take a look at the way I called the `ApplyLoggingAspect` method to apply the logging aspect on an object of type `EmailSender`. Note how I select which input parameters to include using the IncludeInputs method in the `LoggingDataExtractorBuilder` class.

Conclusion:

In a previous article, I talked about the CQS (Command Query Separation) approach to AOP among many things. This approach has many advantages including the easiness to create and apply an aspect and the ability to have the aspect access input/output data in a strongly typed way. Still this approach requires the developer to design behavior classes in a certain way and makes the code more verbose.

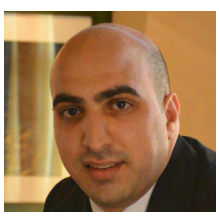
In this article, I have presented a new approach to AOP which I call AOP via functions as an attempt to mitigate the issues of the CQS approach. The basic idea of this new approach is that we adapt the object we need to apply an aspect on to single or multiple functions, then we decorate this function/these functions with the decorator that represents the aspect, then we adapt this function/these functions back to the original interface.

To make the process of creating these adapters and using them a painless process, I have created the InterfaceMappingHelper Visual Studio extension. This extension allows the developer to auto-generate the adapters and a Map method that makes applying aspects easy ■

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.

DocuVieware 3

HTML5 Viewer & Document Management Kit

NEW RELEASE

Easy integration

Full support for custom snap-in

Zero-footprint solution

Fully customizable UI

Mobile devices optimization

Fast & crystal-clear rendering

Check the **New Features** and the **Online Demos**

DOWNLOAD YOUR FREE TRIAL www.docuvieware.com

Powered by GdPicture.NET 14

DocuVieware is an ORPALIS imaging technologies product

Daniel Jimenez Garcia

UNIT TESTING WITH VUE

USING VUE-TEST-UTILS, MOCHA-WEBPACK AND THE VUE-CLI

I have been using Vue.js as my default frontend framework since last year. I have been positively surprised by this framework which strikes a great balance between ease of use, simplicity, performance, and extensibility while leveraging modern frontend technologies like webpack, babel or ES6 to name a few.

However, this might intimidate the uninitiated, as a certain understanding of such technologies is needed in addition to the knowledge of the framework itself. This is even worse when it comes to testing, which is an intimidating topic per se.

Now you not only need to learn about Vue.js, the tools/technologies it is built upon, as well as testing; but you also need to deal with the specifics of writing tests for Vue in a modern JavaScript environment!

In this article, we will see how easy it is to write tests using the awesome vue-test-utils together with mocha-webpack. We will also see how the vue-cli does a great job at wiring all these technologies, so a new project has everything necessary to write unit tests from the very beginning.

Editorial Note: *If you are new to Vue.js, here's an introduction www.dotnetcurry.com/javascript/1349/introducing-vue-js-tutorial*

I hope you will find the article interesting and relevant to your current and future Vue projects! Even if you are not using the vue-cli, you should still find the information relevant once you manually setup mocha-webpack and vue-test-utils.

You can find the companion code on github.

UNIT TESTING WITH VUE - SETTING UP A NEW PROJECT

First of all, make sure you have Node and npm installed on your machine. If you need to, you can download the installer from its official page. Other than that, you will just need your favorite editor and a browser.

With the prerequisites sorted, the first thing we are going to do is to create a project using the vue-cli as it will give us a good starting point with most of what we need already wired out of the box. That is, with one single command, we will be able to generate a project with Vue, webpack, and mocha-webpack ready to go! Installing the vue-cli is as simple as running the following command in your favorite shell:

```
npm install -g @vue/cli
```

Once you have the cli installed, you can start the process for creating a new project by running its create command:

```
vue create unit-test-example
```

You will immediately get prompted with the first option of the creation process, which once completed will create the project inside the specified folder (defaulted from the project name, as in `./unit-test-example` in the example above)

Make sure to select "*Manually select features*" in the first prompt:

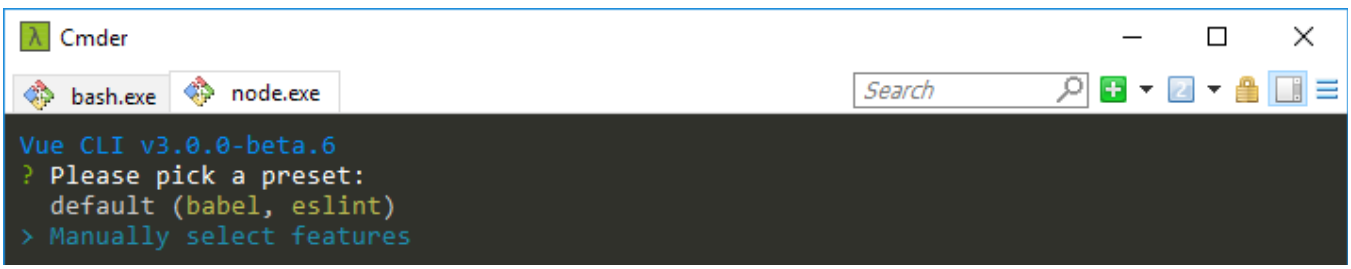


Figure 1, Creating a new vue project

When you manually select the features, the vue-cli will let you customize many of the tools and technologies that you can use. Following the creation process, make sure you select:

- Selected features: Router, Vuex, Linter, Unit test
- Pick a unit testing solution: Mocha

You can combine these with other options but if you prefer using the same selection I made, you can check the following screenshot:

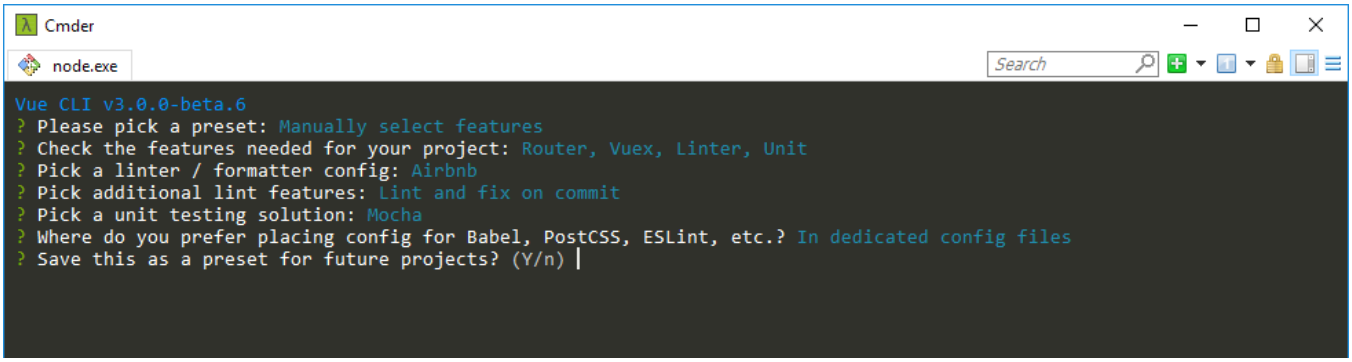


Figure 2, summary of the selected features for the new project

Once the project is generated, switch to the folder where it has been created. You should now be able to:

- Run the npm test command and see an example unit test passing.
- Run the npm run serve command and see your website on localhost:8080

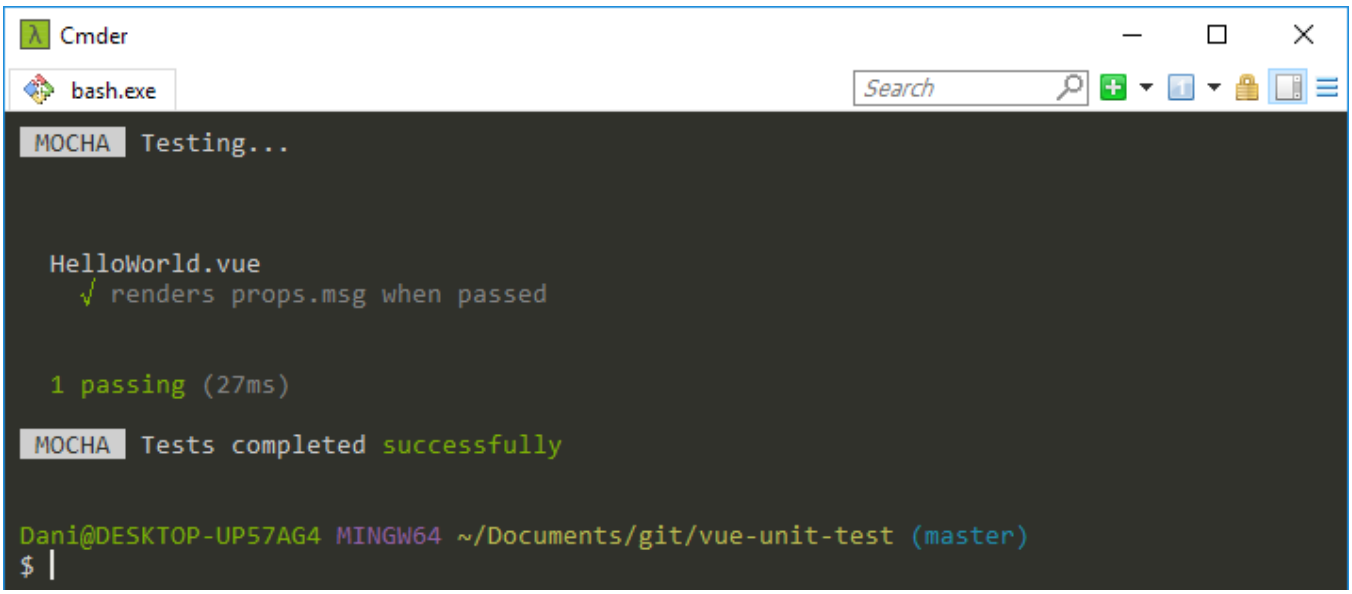


Figure 3, running npm test

Figure 4, running npm run serve

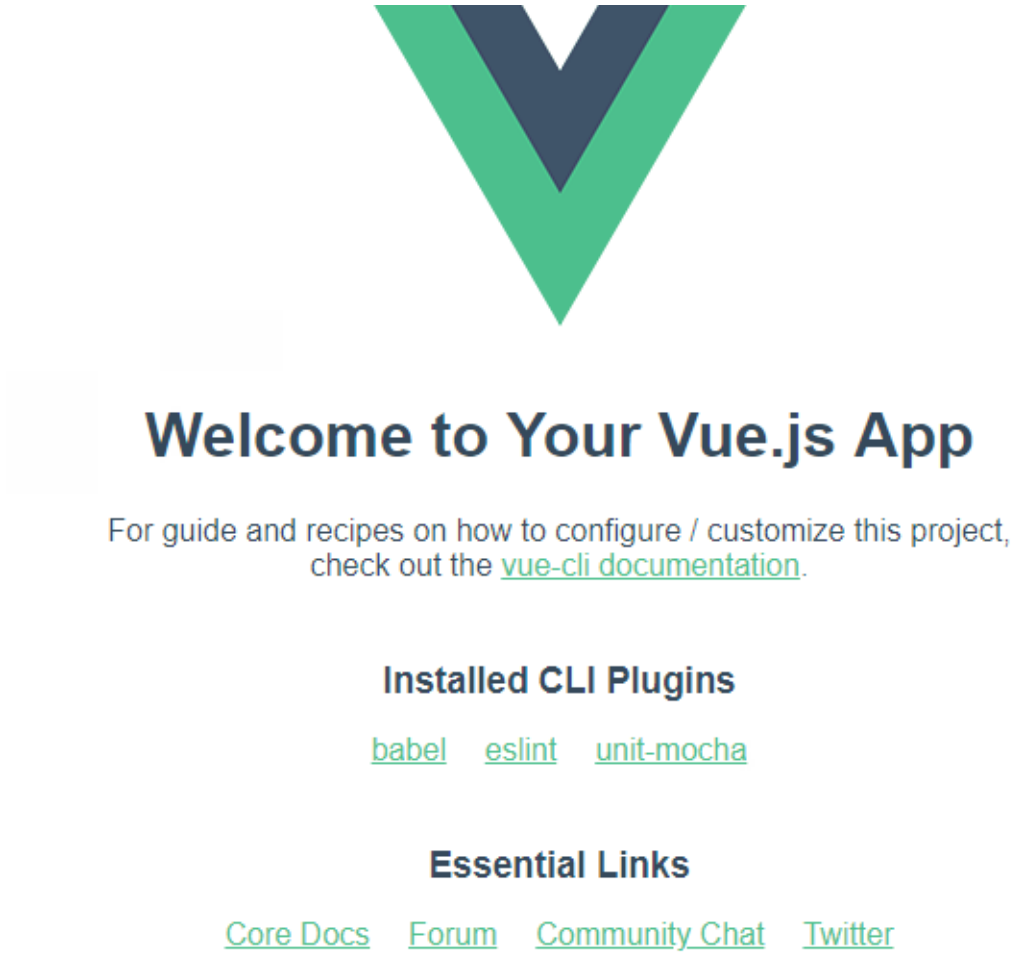


Figure 5, the generated app running locally

That's it, the project is ready for us to start adding more code and writing tests for it! Before we move on, let's take a deeper look at how things work under the covers when using the vue-cli.

UNDERSTANDING HOW THE VUE-CLI WORKS

When you open your project in an editor like VS Code and inspect the generated files, a couple of things will take you by surprise:

- There is no webpack.config.js file(s) anywhere in the project. However, when you run `npm run serve`, the app is bundled with webpack and served with the webpack-dev-server!

- The dependencies listed in package.json do not include the expected webpack, webpack-web-server, various webpack loaders, and mocha-webpack. Instead you will see the `@vue/cli-service` and various `@vue/cli-plugin-*` dependencies.

- The scripts defined in package.json all defer to the vue-cli-service as in:

```
"scripts": {
  "serve": "vue-cli-service serve",
  "build": "vue-cli-service build",
  "test": "vue-cli-service test",
  "lint": "vue-cli-service lint"
},
```

This is all fine because the vue-cli uses a plugin-based architecture in which plugins can define their own dependencies, command and webpack settings!

Certain base commands and settings are part of the "default plugins" and so are available on every project. You can check for yourself if you open the source code of the main Service object of the **@vue/vue-cli-service** and inspect the list of default plugins which at the moment looks like:

```
const builtInPlugins = [
  './commands/serve',
  './commands/build',
  './commands/inspect',
  './commands/help',
  // config plugins are order sensitive
  './config/base',
  './config/css',
  './config/dev',
  './config/prod',
  './config/app'
].map(idToPlugin)
```

..where each of the command files contains the definition of the command itself (i.e., its name and options) and the function invoked when running the command. This way, if you inspect the source code of the serve command you will see this is the one internally invoking the webpack-dev-server:

```
module.exports = (api, options) => {
 api.registerCommand('serve', {
  description: 'start development server',
  usage: 'vue-cli-service serve [options]',
  options: {
   '--open': `open browser on server start`,
   '--mode': `specify env mode (default: ${defaults.mode})`,
   '--host': `specify host (default: ${defaults.host})`,
   '--port': `specify port (default: ${defaults.port})`,
   '--https': `use https (default: ${defaults.https})`
  }
}, args => {
  info('Starting development server...')

  // Lots of code omitted here for brevity

  const server = new WebpackDevServer(compiler, /* options omitted */)
```

In a similar fashion, you can inspect the definition of the base webpack config by opening the files inside the config folder. You will notice the definition of the webpack configuration relies heavily on webpack-merge and webpack-chain. This way optional plugins added to your project can easily extend/override the default webpack configuration!

When an optional plugin is installed to your project like the **@vue/cli-plugin-unit-mocha**, this can use the same API to define additional commands (in this case the test command) and additional webpack configuration. Again, this can easily be seen by inspecting its source code:

```
module.exports = api => {
 api.chainWebpack(webpackConfig => {
  if (process.env.NODE_ENV === 'test') {
   webpackConfig.merge({
   target: 'node',
```

```
   devtool: 'inline-cheap-module-source-map',
   externals: // omitted
  })

  // additional settings omitted
 }
})

api.registerCommand('test', {
 description: 'run unit tests with mocha-webpack',
 usage: 'vue-cli-service test [options] [...files]',
 options: {
  '--watch, -w': 'run in watch mode',
  // additional options omitted
  },
   details: // omitted
 }, (args, rawArgv) => {
  api.setMode('test')
  // for @vue/babel-preset-app
  process.env.VUE_CLI_BABEL_TARGET_NODE = true
  // start runner
  const execa = require('execa')
  const bin = require.resolve('mocha-webpack/bin/mocha-webpack')
  const argv = [
  // omitted for brevity
]
```

So how is this invoked when you run `npm run serve` or `npm test`? The answer is that the vue-cli-service and the plugins needed for the additional features you selected while creating your project, have all been added as **devDependencies** to your project:

```
"devDependencies": {
 "@vue/cli-plugin-babel": "^3.0.0-beta.6",
 "@vue/cli-plugin-eslint": "^3.0.0-beta.6",
 "@vue/cli-plugin-unit-mocha": "^3.0.0-beta.6",
 "@vue/cli-service": "^3.0.0-beta.6",
 "@vue/eslint-config-airbnb": "^3.0.0-beta.6",
 "@vue/test-utils": "^1.0.0-beta.10",
 "chai": "^4.1.2",
 "lint-staged": "^6.0.0",
 "vue-template-compiler": "^2.5.13"
},
```

They will all be found inside your node_modules folder, so the following sequence happens:

1. You run a command like `npm test`, which is defined as `test: "vue-cli-service test"` inside the project's **package.json** file

2. The **@vue/vue-cli-service** installed inside node_modules is loaded

3. The **@vue/vue-cli-service** inspects your dependencies and devDependencies for any additional plugins added to your project (**like @vue/cli-plugin-unit-mocha**). It then registers any new commands provided by them (so they are available in the CLI) and their hooks which extend the default webpack configuration.

4. The vue-cli-service finds the **test command**, which in this case is defined by the **@vue/cli-plugin-unit-mocha** .

5. The **test command** invokes mocha-webpack and provides the default arguments for it like the test file pattern (defaulted as test/unit/**/*.spec.js) and where to find the webpack configuration.

6. As webpack configuration, it is pointed to a special file provided by the @vue/cli-service (found in **@vue/cli-service/webpack.config.js**) that returns the result of combining the base webpack configuration with the chained bits from all the different cli plugins, including the **@vue/cli-plugin-unit-mocha** plugin.

That was quite an in-depth look at the end, but I hope it helped to demystify the vue-cli a bit. In my view, it is a very powerful tool, but it might feel hard to understand since there is a lot going on under the covers without it being explicitly part of your project.

WHAT IF I DON'T USE VUE-CLI?

While the vue-cli is a handy and powerful tool, you might not want or be able to use it in your project. In such a case, you will need to manually install and configure all the required dependencies, including:

- Vue and related libraries like vue-router or vuex
- webpack and webpack-dev-server
- the various loaders and webpack plugins like vue-loader, css-loader or uglifyjs-webpack-plugin (just to name a few)

Then you will need to wire the scripts for generating webpack bundles for production and generate/serve them for local development.

This is not a simple process and requires a good (and up to date!) understanding of webpack, Vue and modern JavaScript. As such it is outside the scope of this article which is to show how to write and run Vue unit tests using vue-test-utils and mocha-webpack.

If you already have a Vue project and you just want to add the minimum needed to unit test with the vue-test-utils and mocha-webpack, the official vue-test-utils has a section about it.

Now that we have a project with all the setup needed to test Vue using the vue-test-utils and mocha-webpack, let's focus on understanding how to write the tests themselves.

UNIT TESTING BASICS WITH VUE-TEST-UTILS AND MOCHA-WEBPACK

The project generated by the vue-cli includes a simple unit test for one of the Vue components in the project. Let's use this example test to introduce the basics of mocha, mocha-webpack, and vue-test-utils.

By default, this test is located in **$/test/unit/HelloWorld.spec.js** and it is unit testing the component **$/src/components/HelloWorld.vue**. Its contents should look as follows:

```
import { expect } from 'chai';
import { shallow } from '@vue/test-utils';
import HelloWorld from '@/components/HelloWorld.vue';

describe('HelloWorld.vue', () => {
```

```
  it('renders props.msg when passed', () => {
    const msg = 'new message';
    const wrapper = shallow(HelloWorld, {
      propsData: { msg },
    });
    expect(wrapper.text()).to.include(msg);
  });
});
```

The first thing I will point is the functions `describe` and `it`. When we write unit tests using mocha-webpack, we are still using the mocha testing framework to define our tests. As any other framework, it provides functions to define a test module and each of the individual tests:

- The function `describe` allows you to define tests grouped into modules. You provide the name for the module and a callback function where you define the individual tests. In this callback, you can also define submodules by adding further calls to describe.

- The function `it` allows you to define a single test. Again, you provide the name for the test and a callback with the test method itself, where you exercise your code and perform assertions. A test passes if all the assertions succeed.

- When mocha runs these tests, it will print the names of the modules and tests following the same nesting order in which they were defined. Each test will be clearly marked as a success or a failure.

```
describe('HelloWorld.vue', () => {
 // This is the definition of the HelloWorld.vue test module
 // Define individual tests with the 'it' function
 // Define submodules with nested 'describe' calls

 it('renders props.msg when passed', () => {
   // This is the definition of the test
   // exercise your code and perform assertions
 });
 it('a second test', () => {
   …
 });
 describe('a submodule', () => {
   it('a third test inside the submodule', () => {
     …
   });
 });
});
```

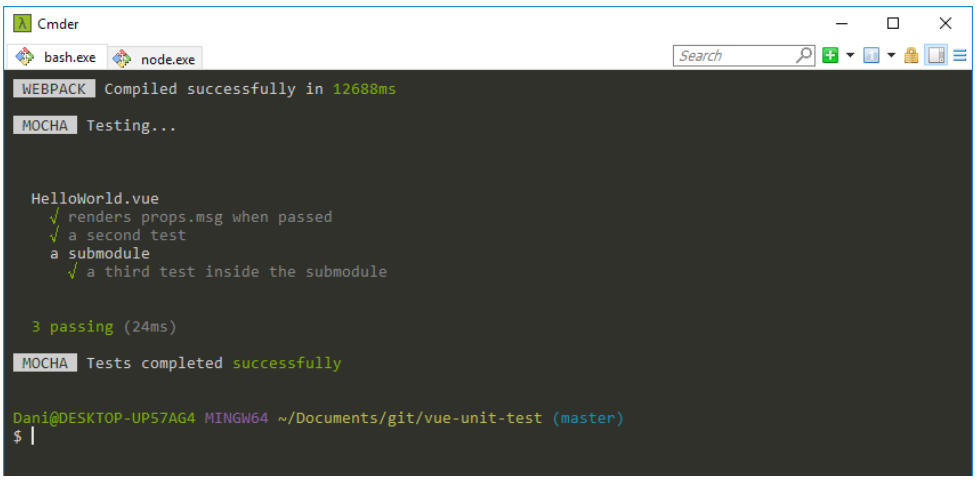


Figure 6, defining tests and modules with mocha

This should all sound familiar as it is the same across most test frameworks. Of course, mocha offers other features, some of which you will see during the rest of the article like the usual before/beforeAll/after/afterAll test hooks or support for asynchronous tests.

Let's now take a look at the assertions, since we said a test will pass if all its assertions succeed. In this simple test, there is one single assertion:

```
import { expect } from 'chai';
...
describe('HelloWorld.vue', () => {
  it('renders props.msg when passed', () => {
    ...
    expect(wrapper.text()).to.include(msg);
  });
});
```

As you can see, the test uses the function `expect` provided by the assertion library chai. The 'expect' flavor of this assertion library provides a BDD (Behavior-driven development) style of assertions which read like natural language. It's also worth mentioning that mocha doesn't care which assertion library you use, any library that throws instances of "Error" will be ok.

So far there is nothing new to someone who wrote tests for Node.js applications using mocha and chai. It is time to see where the vue-test-utils and mocha-webpack get involved!

Let's begin by inspecting the Vue component we are testing, which is defined as single-file Vue component:

```
<template>
 <div class="hello">
  <h1>{{ msg }}</h1>
  // rest of the template omitted
 </div>
</template>

<script>
export default {
 name: 'HelloWorld',
 props: {
  msg: String,
 },
};
</script>

<style scoped>
// omitted
</style>
```

This is a very simple component that receives an input property which is then included in the rendered html, which is exactly what we are testing.

If you remember at the beginning of the article, I said you would need Node.js installed for running the tests. That means the tests will run in a Node.js environment, however, Node.js doesn't know:

- How to load .vue files. It only knows about JavaScript and json files.
- How to handle ES6 modules using the import/export syntax. It only knows about the commonJS modules with require/module.exports

- Even worse, it only knows about relative module imports, and we are using paths from the source code root as in `import HelloWorld from '@/components/HelloWorld.vue';`

So how are we able to run these tests in Node.js after all? With mocha-webpack we can process the tests and source code using webpack before the tests are run.

Once webpack has generated plain JavaScript that Node.js can understand (in the form of the output bundle) mocha will be invoked. It is very similar to the fact the browser wouldn't understand the source code as is written and needs the webpack bundles to be generated in a format the browser can understand.

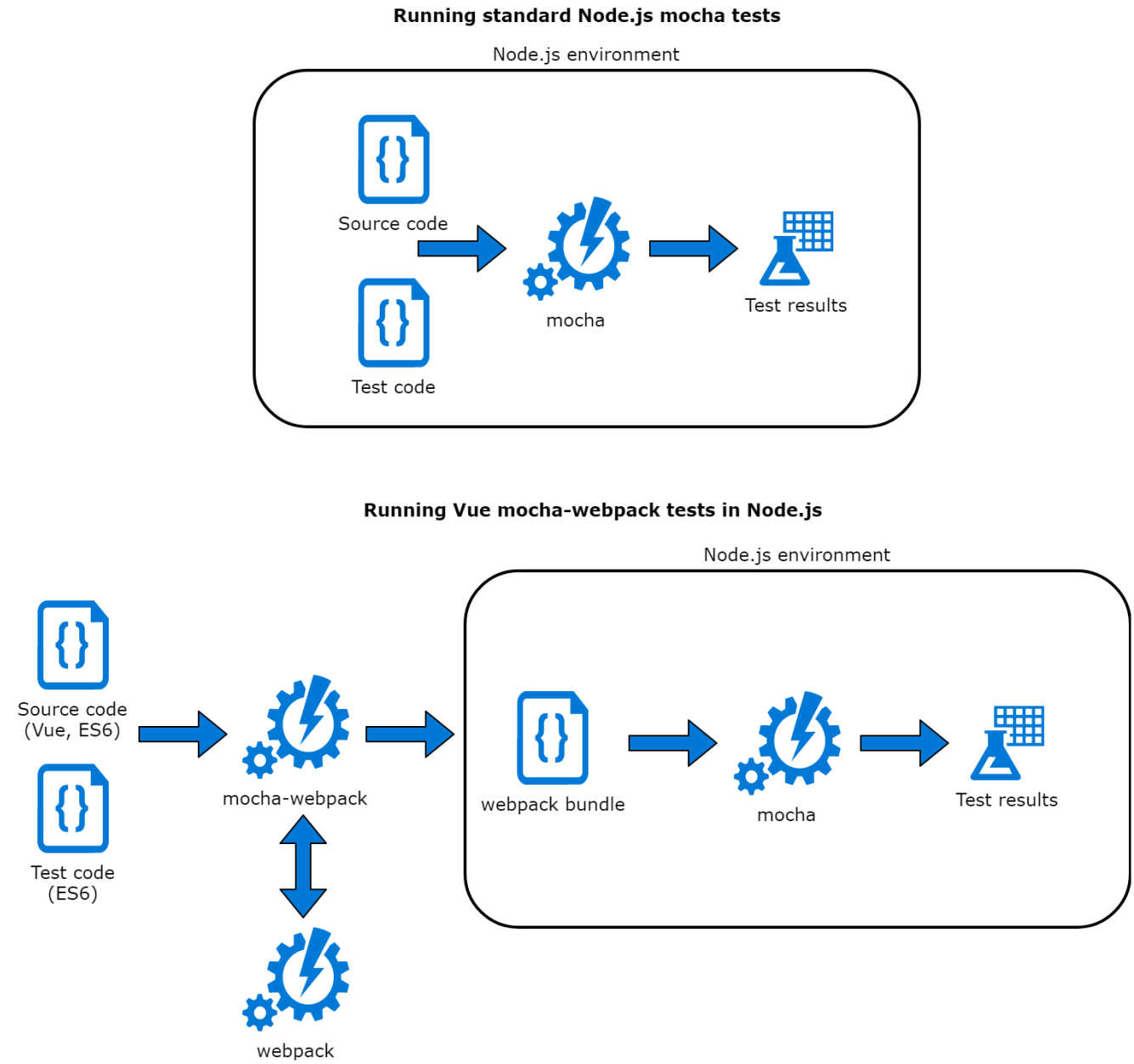


Figure 7, comparing mocha tests VS mocha-webpack vue tests

This is the reason why the tests need to be run using mocha-webpack instead of directly using mocha and the reason why we can use ES6 modules (import/export) and import .vue files, while still running the tests in a Node.js environment

```
// This would never work in plain Node
// as Node doesn't understand import/export nor vue files
import HelloWorld from '@/components/HelloWorld.vue';
```

The final piece is understanding how we have been able to instantiate the component in our test and perform an assertion over its rendered output. This is where vue-test-utils fits the puzzle.

The **vue-test-utils** library provides an API that allows you to mount a component given its definition (for example contained in a vue file) into a virtual DOM rather than a real browser (using jsdom under the covers):

- This API exposes the `shallow` method, which allows you to instantiate the HelloWorld.vue component into this virtual DOM. It returns a wrapper that you will use to interact with your component and the virtual DOM it's being mounted, for example triggering events or inspecting the resulting html.
- The `shallow` method also allows you to provide input properties to your component. We will also see through the article how it can be used to inject mocks.
- There is an alternative to `shallow` called `mount`, which is briefly discussed later in the article.
- This allows you to write a test that instantiates a component and mounts it into a virtual DOM using the `shallow` API, and then uses the `text()` method of the returned wrapper to ensure the html rendered by the component included the expected message:

```
import { shallow } from '@vue/test-utils';
import HelloWorld from '@/components/HelloWorld.vue';
...
const msg = 'new message';
const wrapper = shallow(HelloWorld, {
  propsData: { msg },
});
expect(wrapper.text()).to.include(msg);
```

With that, you should now have a decent understanding of everything involved so you can write and run that simple test! If you are still with me, we can start looking at some of the common scenarios you will find when testing Vue components.

BASIC TESTING SCENARIOS

Let's create a simple component with a counter and a button to increase it, that we can use as the target of the next tests. This is as simple as creating **$/src/components/Counter.vue** with the following contents:

```
<template>
 <div>
  Counter value: <span class="counter-value">{{ value }}</span>
  <br/>
  <button @click="onIncrease">Increase</button>
 </div>
</template>
<script>
export default {
 props: {
  initialValue: {
   type: Number,
```

```
    required: false,
    default: 0
   }
  },
  data(){
   return {
    value: this.initialValue
  };
 },
 computed: {
  stringValue(){
   return this.value.toString();
  }
 },
 methods: {
  onIncrease(){
   this.value += 1;
  }
 }
};
</script>
```

This is a rather trivial component but enough to showcase common testing scenarios. Let's begin by creating a new test file **$/tests/unit/Counter.spec.js** where we create our tests:

```
import { expect } from 'chai';
import { shallow } from '@vue/test-utils';
import Counter from '@/components/Counter.vue';

describe('Counter.vue', () => {
 // Tests to be defined here
});
```

Testing the rendered output

The example test we analyzed earlier showcased one way of testing its rendered output, by using the text function of the component wrapper. There is an alternative way of testing the rendered output, which is the html function of the component wrapper.

While `text` returns only the text that would show on a browser, the `html` includes the full html rendered by the component. The next two tests show the difference between these approaches:

```
describe('when an initial value is provided', () => {
 it('it is rendered by the component', () => {
  const wrapper = shallow(Counter, {
   propsData: { initialValue: 42 },
  });
  expect(wrapper.text()).to.include('Counter value: 42');
 });
});

describe('when omitting the initial value', () => {
 it('the default value 0 is rendered by the component', () => {
  const wrapper = shallow(Counter, {
   propsData: { },
  });
  expect(wrapper.html()).to.include('Counter value: <span class="counter-value">0</
```

```
  span>');
 });
});
```

Testing the component state

It is also possible to access the data properties of the component instance by getting the instance reference from the component wrapper. This lets you assert directly over the component state (its data properties) instead of the rendered output, letting you decide between the two approaches according to your needs.

- The component state can be seen as an internal implementation detail, and by directly accessing it in your tests, you might be coupling them too much to your implementation.
- However, testing your component rendered output might make your tests more brittle and dependent on the templates which tend to be modified more frequently.
- This is a trade-off you will need to solve by yourself.

The component wrapper exposes a `vm` property with a reference to the actual component instance, where you can directly access its state, computed blocks, methods etc.

This way the previous tests could be written as:

```
describe('when an initial value is provided', () => {
 it('it is used as the initial counter value', () => {
  const wrapper = shallow(Counter, {
   propsData: { initialValue: 42 },
  });
  const componentInstance = wrapper.vm;
   expect(componentInstance.value).to.be.equal(42);
 });
});

describe('when omitting the initial value', () => {
 it('the initial value is set as 0', () => {
  const wrapper = shallow(Counter, {
   propsData: { },
  });
  const componentInstance = wrapper.vm;
  expect(componentInstance.value).to.be.equal(0);
 });
});
```

Testing component computed properties

Computed properties can be tested in the same way as the component state, by means of getting a reference to the component instance. This way, testing a computed value is as simple as asserting that the value of the `wrapper.vm.computedName` is as expected.

However, by their own nature, computed properties depend on either the current component state or other properties. This means you probably want to use the setProps and/or setData functions of the component wrapper to update its state and verify the computed property was reevaluated as expected.

The following tests showcase this, and to make them more interesting they also show how common test initialization can be moved into a **beforeEach** test hook:

```
describe('the stringValue computed', () => {
 let wrapper;
 let componentInstance;
 beforeEach(() => {
  wrapper = shallow(Counter, {
   propsData: { initialValue: 42 },
  });
  componentInstance = wrapper.vm;
 });
 it('returns the string representation of the initial value', () => {
  expect(componentInstance.stringValue).to.be.eql("42");
 });
 it('returns the string representation of the updated value', () => {
  wrapper.setData({ value: 99 });
  expect(componentInstance.stringValue).to.be.eql("99");
 });
});
```

Testing browser event handlers

Our component binds an event handler to the click event of the button that increases the counter. Triggering the click event is straightforward using vue-test-utils, allowing us to write tests to verify the expected behavior.

We just need to locate the html element using the component wrapper, and then use the trigger method with the event name we want to trigger.

As discussed earlier, you could decide to verify how the rendered html changes as a result of the event or how the component state has changed. The following tests show both approaches:

```
describe('when the increment button is clicked', () => {
 let wrapper;
 let componentInstance;
 let increaseButton;
 beforeEach(() => {
  wrapper = shallow(Counter, {
   propsData: { initialValue: 42 },
  });
  componentInstance = wrapper.vm;
  increaseButton = wrapper.find('button');
 });
 it('the counter value is increased by one', () => {
  increaseButton.trigger('click');
  expect(componentInstance.value).to.be.equal(43);
 });
 it('the rendered output contains the value increased by one', () => {
  increaseButton.trigger('click');
  expect(wrapper.text()).to.include('Counter value: 43');
 });
});
```

Testing component methods

Many times, you will define component methods to be used as event handlers for browser events but that's not always going to be the case. For example, some methods will be used as event handlers of events emitted by nested Vue components, which cannot be triggered when mounting components using the `shallow` function.

This shouldn't stop you from testing any component method as you can invoke them directly through the reference to the component instance as in `wrapper.vm.methodName(methodArgs).`

For example, you could directly invoke the method bound to the increase button with `wrapper.vm.onIncrease()`. That way you could directly test this method without triggering the button click event as in:

```
describe('the onIncrease method', () => {
 let wrapper;
 let componentInstance;
 beforeEach(() => {
  wrapper = shallow(Counter, {
   propsData: { initialValue: 42 },
  });
  componentInstance = wrapper.vm;
 });
 it('the counter value is increased by one', () => {
  componentInstance.onIncrease();
  expect(componentInstance.value).to.be.equal(43);
 });
 it('the rendered output contains the value increased by one', () => {
  componentInstance.onIncrease();
  expect(wrapper.text()).to.include('Counter value: 43');
 });
});
```

As we discussed while testing the component data, by doing this, you are getting into the internals of the component and possibly your test knows too much about the internal implementation. Try to trigger the events through the wrapper when possible/practical.

Testing events emitted by the component

It is not uncommon for Vue components to emit events themselves under certain circumstances, that other Vue components can listen to and react. In fact, this is the recommended mechanism for communicating a child component back to its parent.

Let's update the `onIncrease` method of our simple component to emit an increased event:

```
onIncrease(){
 this.value++;
 this.$emit('increased', this.value);
}
```

The wrapper provided by vue-test-utils also provides a handy way of verifying that the expected events were emitted by the component you are testing.

We can now update the earlier tests for the increase button/method and verify the increased event was

emitted as expected:

```
it('emits an increased event with the new value', () => {
  increaseButton.trigger('click');
  expect(wrapper.emitted().increased.length).to.be.equal(1);
  expect(wrapper.emitted().increased[0]).to.be.eql([43]);
});
```

Note: you might be wondering what is the difference between `to.be.equal()` and `to.be.eql()`.

- With `equal` you perform a **strict equal** comparison which requires both values to be of the same type and for Object/Arrays to point to the same instance.

- With `eql` you perform a **deep equal** comparison which works the same for simple value but also succeeds for object/arrays if they are different instances but have the exact same properties.

INSTANTIATING COMPONENTS: MOUNT VS SHALLOW

There is a second way of instantiating components using the vue-test-utils other than the `shallow` method, which is the `mount` method.

With the `shallow` method, any child components rendered by the component under test are automatically stubbed. This way the component is tested in isolation and the tests are faster since it doesn't need to update and render the entire component tree.

On the other hand, the `mount` method will create an instance of every child component, include them in the component tree and include its rendered output in the virtual dom.

You can think of them in terms of **unit vs integration** tests. The `shallow` method is perfect for writing unit tests isolating your Vue components, while the `mount` method is needed when writing integration tests involving several components.

You can read more about this rationale in the vue-test-utils docs.

MOCKING DEPENDENCIES

Until now we have managed to test our component in isolation by instantiating them with the shallow function and providing any input prop data.

However, we have been testing a decidedly simple component with little interaction with the outside world. Let's modify it by adding an extra button that will increase the counter according to the result of an HTTP request, which will be sent using the axios library.

First, install the library with the following command:

```
npm install --save axios
```

Next update **src/main.js** to include **axios** as the `$http` property inside every component instance. This will

allow us to use axios as `this.$http` inside our components.(And as we will see later, a standard technique across different libraries like the vue-router or vuex)

```
import axios from 'axios';
Vue.prototype.$http = axios;
```

Next update the Counter.vue component with a new button and associated method where the counter is increased according to a json response from an HTTP request:

```
// on the template
<button id="increase-http-btn" @click="onIncreaseFromHttp">
  Increase from Http response
</button>

// on the script
methods: {
 ...
 onIncreaseFromHttp(){
  return this.$http.get('https://jsonplaceholder.typicode.com/users').then(response
  => {
   this.value += response.data.length;
   this.$emit('increased', this.value);
  });
 }
}
```

This adds a new challenge to our tests! In order to test the behavior of the new button, we need to mock the `$http` property of the component instance so it doesn't try to send a real HTTP request and instead uses some predefined behavior.

USING SINON TO CREATE STUBS/SPIES

Before we move onto injecting mocks for our dependencies, we first need to define them. This is where a library like sinon.js will be of great help as it provides a simple but powerful way of defining mocks for any dependency you need.

- The first thing you should know when using sinon is that they have split the api for creating stubs and spies. A spy is basically a method that returns nothing but keeps track of every call received.

- A stub is an extension of a spy. Apart from keeping track of individual calls, it lets you define the behavior for those calls. For example, you can define a stub that always returns a value or a stub that returns a particular value when some specific inputs are provided.

Before we can use sinon, we will need to add it as a devDependency to our project:

```
npm install --save-dev sinon
```

Now we can take a look at how to mock the method call `this.$http.get('https://jsonplaceholder.typicode.com/users')` so it returns a predefined json response and we can then assert the right value was increased. This is as simple as creating a sinon stub defined with the following behavior:

```
const mockResponse = {
  data: [{a: 'user'}, {another: 'user'}]
```

```
};
const mockAxios = {
  get: sinon.stub()
    .withArgs('https://jsonplaceholder.typicode.com/users')
    .returns(Promise.resolve(mockResponse))
}
```

This way we are simulating the actual HTTP API (which returns an array of users in its json response) when the GET method is invoked with the correct url.

While this is enough to continue with our article, it barely scratches the surface of what sinon can do for you. If you are going to write tests in a JavaScript environment, it is really worth getting used to it.

INJECTING MOCKS INTO THE COMPONENT INSTANCE

We know how to define a mock as either a static object or a more dynamic sinon stub/spy. Now we need to inject these mocks into the Vue component instance when using the shallow/mount methods.

Luckily for us, both methods to create a component instance accept a *mocks* object in their arguments that will be injected into the component. Thus, providing a sinon stub for the `this.$http` component property is as simple as providing the mock in the shallow/mount call:

```
wrapper = shallow(Counter, {
  propsData: { initialValue: 42 },
  mocks: {
    $http: mockAxios
  }
});
```

Now we have everything we need to verify the behavior of the new button which increases the counter by the number of objects in the HTTP response array:

```
it('the counter value is increased by the number of items in the response', (done)
=> {
  increaseHttpButton.trigger('click');
  setImmediate(() => {
    expect(componentInstance.value).to.be.equal(44);
    done();
  });
});
```

The main change with the previous test is that now our test is asynchronous as it depends on a Promise being resolved (even if we mocked `this.$http.get` it still returns a Promise). That's why our test method accepts a `done` parameter and wraps the execution of the assertion in a call to `setImmediate`. This way the promise can be resolved before the body of the `setImmediate` is executed and our assertions run.

Mocha has a nice support for asynchronous tests we could tap into if we were testing the new `onIncreaseFromHttp` method directly, as we could chain our assertions directly into the promise chain:

```
it('increases the value by the number of items in the response', () => {
  return componentInstance.onIncreaseFromHttp().then(() => {
    expect(componentInstance.value).to.be.equal(44);
  });
});
```

You can read more about the support for asynchronous testd in mocha in its docs.

MOCKING VUE-ROUTER

When using the vue-router, your component gets injected with two properties:

- The current route is available as `this.$route`
- The router object that lets you use the vue-route api to navigate or work with the history is available as `this.$router`

This means we can simply create a mock for them and provide them as mocks in the shallow/mount call. Imagine your component uses the current route in some method or computed property:

```
computed: {
 currentRoute(){
  return this.$route.name;
 }
}
```

We can just create a mock route object and inject it on the call to create the instance:

```
describe('the currentRoute property', () => {
 let wrapper;
 let componentInstance;
 beforeEach(() => {
  const mockRoute = { name: 'foo' };
  wrapper = shallow(Counter, {
   mocks: {
    $route: mockRoute
   }
  });
  componentInstance = wrapper.vm;
 });
 it('returns the name of the current route', () => {
  expect(componentInstance.currentRoute).to.be.equal('foo');
 });
});
```

Now let's see how we can deal with the router. Let's suppose a certain action triggers some programmatic navigation in your component:

```
methods: {
 onNavigate(){
  this.$router.push({name: 'home'});
 },
}
```

It's equally straightforward to create a sinon spy (since we just want to track the received calls) and create a test where we verify the expected navigation was invoked:

```
describe('the onNavigate method', () => {
 let wrapper;
```

```
 let componentInstance;
 let mockRouter;
 beforeEach(() => {
  mockRouter = {
   push: sinon.spy()
  };
  wrapper = shallow(Counter, {
   mocks: {
    $router: mockRouter
   }
  });
  componentInstance = wrapper.vm;
 });
  it('navigates to the home page', () => {
  componentInstance.onNavigate();
  sinon.assert.calledWith(mockRouter.push, {name: 'home'});
 });
});
```

Many other Vue libraries and plugins follow the same idea of injecting methods and properties into the component instance, like vuex which injects the `$store` property. This means by following this approach, you will be able to mock dependencies on more than one of Vue's plugins/libraries.

USING BABEL-PLUGIN-REWIRE TO MOCK EXTERNAL DEPENDENCIES

Not every dependency will be a friendly Vue plugin that injects methods/properties into the component instance. Sometimes we will find dependencies which are simply imported into the component code.

For example, imagine you have a component that loads a list of users through an HTTP request, but it is directly importing the axios library:

```
import axios from 'axios';
...
methods:{
 onLoadUsers(){
  return axios.get('https://jsonplaceholder.typicode.com/users').then(response => {
   this.users = response.data;
  });
 }
}
```

In this case, we cannot simply pass the mock within the call to the `shallow` function. Our component is directly loading the library with `import axios from 'axios'`, so we need a different strategy:

- With the latest versions of the vue-loader (13.0.0 or higher) the preferred way is using the babel-plugin-rewire.

- In previous versions, you could also use the inject-loader but this no longer works, so be careful when you read some"old" 2017 tutorials.

As usual, start by installing the library:

```
npm install --save-dev babel-plugin-rewire
```

Then, make sure you enable it on your **.babelrc** file, exclusively for the test environment:

```
{
 “presets”: [
  “@vue/app”
 ],
 “env”: {
  “test”: {
   “plugins”: [ “rewire”]
  }
 }
}
```

Now the plugin is wired and every time we import a Vue component (or any other file) a function named `__Rewire__` lets you replace any dependency loaded through an import statement.

Since our component loads axios as:

```
import axios from ‘axios’;
```

Which means in our test we can replace the library with a mock as in:

```
mockAxios = {
 get: sinon.stub()
  .withArgs(‘https://jsonplaceholder.typicode.com/users’)
  .returns(Promise.resolve({data: mockUsers}))
};

Users.__Rewire__(‘axios’, mockAxios);
```

..where the first parameter to the rewire function should match *the name of the variable* and not the string used with the import statement!

That’s it, you should now be able to mock external dependencies which are directly imported by your components. Such a test could look like this:

```
let mockAxios;
let mockUsers;
beforeEach(() => {
 mockUsers = [{a: ‘user’}, {another: ‘user’}];
 mockAxios = {
  get: sinon.stub()
   .withArgs(‘https://jsonplaceholder.typicode.com/users’)
   .returns(Promise.resolve({data: mockUsers}))
  };
  Users.__Rewire__(‘axios’, mockAxios);
});

describe(‘when the load users button is clicked’, () => {
 it(‘updates the list of users with the HTTP response’, (done) => {
  const wrapper = shallow(Users);
  const componentInstance = wrapper.vm;
  wrapper.find(‘button’).trigger(‘click’);
```

```
  setImmediate(() => {
    expect(componentInstance.users).to.be.equal(mockUsers);
    done();
  });
 });
});
```

Conclusion

Writing tests is not easy. Particularly when you want to write good tests, the ones that give you confidence about your software without slowing you down, locking your design choices, or costing you too much time/money. In fact, it is probably one of the hardest aspects to master about software development.

That's why it's worth designing your testing strategy with different types of tests in mind (unit, integration, e2e). In the context of web applications, as more and more functionality and efforts are put into the client code, you also need to consider the client code in your testing strategy.

Client-side testing with modern frameworks like Vue gets even more complicated due to the number of tools/libraries involved. The fact that half of this lengthy article is dedicated to explaining how things are setup and work under the covers, is a testament to that.

But as I hope the second half of the article has shown, once the basic setup is in place, writing tests for your Vue components is a surprisingly pleasant and straightforward experience. And one would hope that as we get better tooling like vue-cli, it becomes easier to get started writing unit tests and more developers find themselves writing them ■

Download the entire source code from GitHub at
github.com/DanielJG/vue-unit-test

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 Ravi Kiran for reviewing this article.

Ravi Kiran

Dependency Injection: Introduction and DI Techniques in Angular

THIS TUTORIAL WILL DEMONSTRATE HOW DEPENDENCY INJECTION WORKS IN ANGULAR. IT ALSO COVERS THE DIFFERENT WAYS TO REGISTER AND INJECT THE DEPENDENCIES.

Extensibility is one of the core principles of Angular. Every feature in Angular is built with extensibility in mind. The applications written in OOPs (Object Oriented Programming Structures) based programming languages, use Dependency Injection (DI) to make applications extensible and testable.

Though JavaScript is not an Object-Oriented Programming language, it is an Object Based language. And the language features of JavaScript can be used to implement DI. AngularJS had implemented this feature and Angular (v2 onwards) continues to do so with more flexibility.

A dependency could be an external object which is used by a block of code to achieve its functionality. The dependencies can be created in the same block where they are needed, but this will make the block dependent on one type of dependency. If there is a change in the way the dependency has to be created, the block using it needs to be modified.

So, creating dependency wherever it is needed makes the code brittle and hard to maintain. It also makes it hard to test the code, as

the dependencies created in it can't be mocked. DI solves these problems, by providing a way to inject dependencies wherever they are needed. The dependencies are created at a central location, hence reducing repetition of code. And they can be mocked while writing unit tests.

This article will explain how to use DI in Angular, as well as the different ways of creating and using dependencies in an Angular application.

How DI works in Angular

A dependency in Angular can be a class, referred as a service or even a simple object. Angular's Dependency Injection is based on providers, injectors, and tokens. Every Angular module has an injector associated with it. The injector is responsible to create the dependencies and inject them when needed. Dependencies are added to the injector using the providers property of the module metadata.

Every dependency is a key-value pair, where the key is a token and the value is the object of the dependency. For Services, Angular creates the tokens by itself. For objects, we need to create the tokens. When a dependency is injected into another block (viz., a component, directive or service), the injector resolves the object from its list of dependencies using the token. Figure 1 shows how the token, provider and the injector work together.

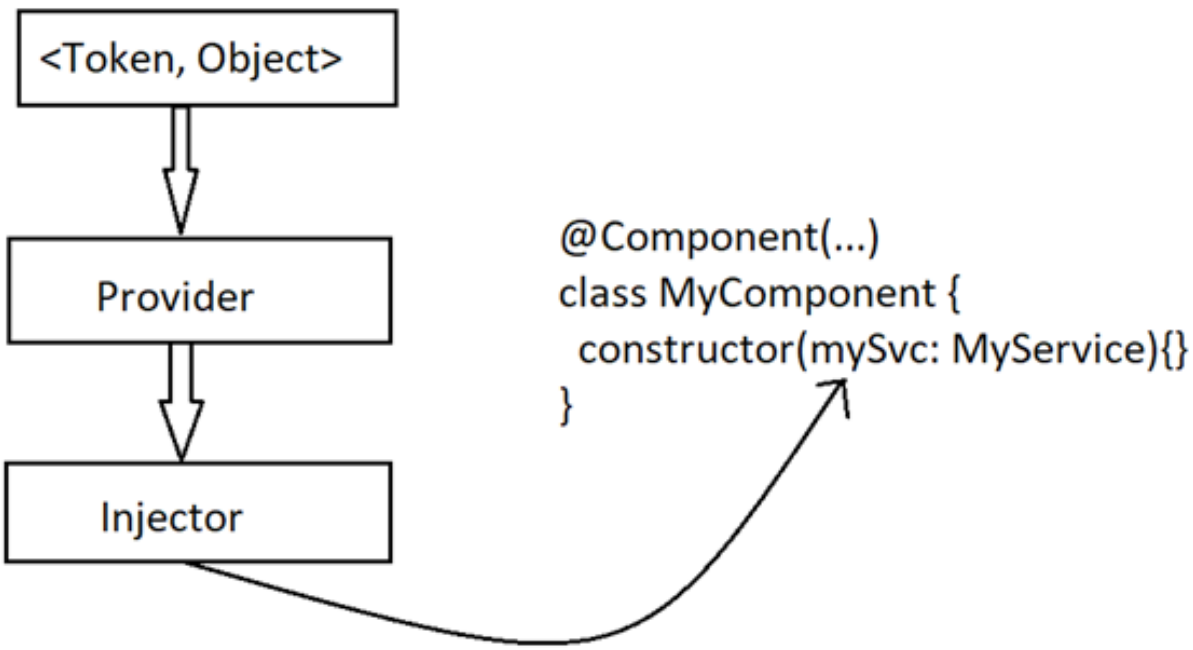


Figure 1 – DI in Angular

Similarly, providers can be added to the components as well. The dependencies registered in the providers section of a component can be used in that component, as well as the components or directives descending this component. They can't be used in the services or the directives and components, that don't descend the component providing them.

Choosing between a module and the component to register the dependency is based on need. In most cases, the dependencies are registered in the modules. The dependencies registered in the modules are registered with the application's root injector. There will be just one instance of such dependency throughout the application. The dependencies registered in the component have an instance as long as the component is alive and they get destroyed if the component is removed from the view.

Providing Classes

The simplest way to register and use dependencies in Angular is using TypeScript classes as services. Say, we have the following class:

```
import { Place } from ‘./place.model’;
import { Injectable } from ‘@angular/core’;

@Injectable()
export class PlacesService {
  places: Place[];

  constructor() {
    this.places = [];
    this.places.push({ name: ‘Charminar’, city: ‘Hyderabad’, country: ‘India’,
    isVisited: true, rating: 4 });
    this.places.push({ name: ‘London Bridge’, city: ‘London’, country: ‘UK’,
    isVisited: false, rating: 4.5 });
    this.places.push({ name: ‘Red Rocks’, city: ‘Denver’, country: ‘USA’,
    isVisited: true, rating: 3 });
    this.places.push({ name: ‘Taj Mahal’, city: ‘Agra’, country: ‘India’,
    isVisited: false, rating: 5 });
    this.places.push({ name: ‘Eiffel Tower’, city: ‘Paris’, country: ‘France’,
    isVisited: true, rating: 4 });
  }

  toggleVisited(name: string) {
    let place: Place = this.places.find(p => p.name === name);
    place.isVisited = !place.isVisited;
  }
}
```

Notice that the above class has the annotation *Injectable* applied. The annotation helps Angular in injecting any dependencies into the constructor of the class. Though the class *PlacesService* doesn’t have any dependencies, it is still useful to mark it with *Injectable*, as it will avoid any errors later when the dependencies are added to this class.

It has to be registered with the module as shown in the following snippet to be able to inject it into a component or elsewhere:

```
@NgModule({
  // Other parts of the module
  providers: [PlacesService]
})
export class AppModule {}
```

And then it can be used in a component by injecting in the constructor, as shown below:

```
@Component(...)
export class PlacesComponent {
  constructor(private placesService: PlacesService) {
  }
}
```

If you want to see this service working with a component, visit this plunk. Rest of the article modifies this sample to try the other techniques in DI.

Providing an Alternative Class

It is possible to override a service using another class that enhances its functionality. Say, we have the class *BetterPlacesService*, with the following definition:

```
import { Place } from './place.model';
import { Injectable } from '@angular/core';

@Injectable()
export class BetterPlacesService {
  private _places: Place[] = [];

  get places(){
    return this._places;
  }

  constructor() {
    this._places.push({ name: 'Charminar', city: 'Hyderabad', country: 'India',
    isVisited: true, rating: 4 });
    this._places.push({ name: 'London Bridge', city: 'London', country: 'UK',
    isVisited: false, rating: 4.5 });
    this._places.push({ name: 'Red Rocks', city: 'Denver', country: 'USA',
    isVisited: true, rating: 3 });
    this._places.push({ name: 'Taj Mahal', city: 'Agra', country: 'India',
    isVisited: false, rating: 5 });
    this._places.push({ name: 'Eiffel Tower', city: 'Paris', country: 'France',
    isVisited: true, rating: 4 });
  }

  toggleVisited(name: string) {
    let place: Place = this._places.find(p => p.name === name);
    place.isVisited = !place.isVisited;
  }
}
```

The difference between the *PlacesService* class and the *BetterPlacesService* class is, instead of providing an array of places directly from the class, it uses a getter property to expose the array. Rest of the functionality remains the same. Now we can provide this class when someone asks for *PlacesService*. To do so, change the *providers* block snippet as shown below:

```
providers: [PlacesService, {provide: PlacesService, useClass: BetterPlacesService}]
```

Now if you run the sample, you will see that *BetterPlacesService* is getting invoked when we use *PlacesService* in the *PlacesComponent*.

Providing a Registered Dependency

It is possible to override a dependency with another registered dependency. The example shown in the last section can be modified for this. Modify the *providers* array in the module as shown below:

```
providers: [PlacesService,
            BetterPlacesService,
            {provide: PlacesService, useExisting: BetterPlacesService}]
```

Make sure to register the dependency assigned to *useExisting*. Otherwise, it results in an error similar to the one shown in Figure 2:

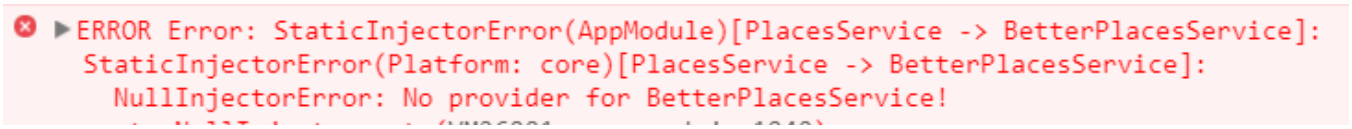


Figure 2 - Unregistered dependency error

Now the *PlacesComponent* would be still using *BetterPlacesService*.

Using a Value

We can use an object to replace a service. For this, we need to create an object following the blueprint of the service class. The following snippet shows an object that can be used to replace *PlacesService*:

```
export const placesObj = {
  places: [ { name: 'Charminar', city: 'Hyderabad', country: 'India', isVisited:
  true, rating: 4 },

  { name: 'London Bridge', city: 'London', country: 'UK', isVisited: false, rating:
  4.5 },

  { name: 'Red Rocks', city: 'Denver', country: 'USA', isVisited: true, rating: 3
  },

  { name: 'Taj Mahal', city: 'Agra', country: 'India', isVisited: false, rating: 5
  }
    ],

  toggleVisited(name: string){
    let place: Place = this.places.find(p => p.name === name);
    place.isVisited = !place.isVisited;
  }
};
```

This can be added to *providers* using *useValue*, as shown below:

```
{provide: PlacesService, useValue: placesObj}]
```

Using a Factory Function

If there is a need to execute some logic before creating the service object, it can be done using a factory function. The function has to return an object similar to the one used with *useValue*. The following snippet shows an example of a factory function:

```
export function getPlacesService(){
  return new PlacesService();
}
```

And it can be registered as follows:

```
providers: [{provide: PlacesService, useFactory: getPlacesService, deps: []}]
```

The *deps* array can be used to provide any dependencies that the function needs. Here, the factory function is not accepting any inputs, so the *deps* array is empty.

Injecting non-class Dependencies

At times, the applications need some objects to be accessible everywhere. Though we have a way to register them using *useValue*, it would be an overkill to create a service and then override it with an object. Angular provides a way to register the objects as injectables through *InjectionToken*. Say we need to inject the following object in the *PlacesComponent* to assign max and min values for the rating that a user can provide for a place:

```
export class AppConstants {
  maxRating: number;
  minRating: number;
}

export const APP_CONSTANTS: AppConstants = {
  maxRating: 5,
  minRating: 1
};
```

The class *AppConstants* is created to use the object anywhere in a type-safe way. It won't be used to register the value. Now that we have the value, let's create the token for it. The following snippet shows this:

```
export const APP_CONST_INJECTOR =  new  InjectionToken<AppConstants>('app.config');
```

You can have the class *AppConstants*, the value *APP_CONSTANTS* and the token *APP_CONST_INJECTOR* in a single file named *appconstant.ts*. The following snippet shows the portion of the module file that registers the constant:

```
// other import statements
import  {  InjectionToken }  from  '@angular/core';
export const APP_CONST_INJECTOR =  new  InjectionToken<AppConstants>('app.config');

@NgModule({
  providers:  [
    /* Other providers */,
    { provide: APP_CONST_INJECTOR, useValue: APP_CONSTANTS }]
})
export class AppModule {}
```

To inject this constant, we need the token and the Inject decorator. To be able to use the constant in the official TypeScript way, we need the class *AppConstant*. The constant can be used in *PlaceComponent*, as it has a textbox accepting a number and we can apply these values for min and max values of the input. The following snippet shows the class of the *PlaceComponent* and it uses the constant registered above:

```
import { Component, Input, Output, EventEmitter, Inject } from '@angular/core';
import { Place } from './place.model';
import { APP_CONST_INJECTOR, AppConstants } from './appconstant';

@Component({
  selector: 'place',
  templateUrl: './place.component.html'
})
export class PlaceComponent {

  constructor(@Inject(APP_CONST_INJECTOR) private appConstants: AppConstants) {

  }
```

```
  get maxRating(){
    return this.appConstants.maxRating;
  }

  get minRating(){
    return this.appConstants.minRating;
  }

  @Input('selectedPlace')
  place: Place;
  placeChanged: string;

  @Output('toggleVisited')
  toggleVisited = new EventEmitter<string>();

  togglePlaceVisited() {
    this.toggleVisited.emit(this.place.name);
  }

  get IsVisited(): string {
    return this.place.isVisited? 'Yes' : 'No';
  }
}
```

Conclusion

Dependency Injection is one of the most important features that Angular provides to build scalable web applications. As this tutorial demonstrated, the API for DI in Angular allows us to inject anything, thus making the code extensible and testable. In a forthcoming tutorial on Unit Testing in Angular, will will see how these features help in unit testing ■

Microsoft® MVP Most Valuable Professional

Ravi Kiran
Author

*Ravi Kiran (a.k.a. Ravi Kiran) is a developer working on Microsoft Technologies at Hyderabad. These days, he is spending his time on JavaScript frameworks like AngularJS, latest updates to JavaScript in ES6 and ES7, Web Components, Node.js and also on several Microsoft technologies including ASP.NET 5, SignalR and C#. He is an active **blogger**, an author at SitePoint and at DotNetCurry. He is rewarded with Microsoft MVP (ASP.NET/IIS) and DZone MVB awards for his contribution to the community.*

Thanks to Keerti Kotaru for reviewing this article.

Gerald Versluis

Content Moderation using Azure Custom Vision and Xamarin

In a previous edition of DotNetCurry Magazine I wrote about how you could make your apps smarter with the Azure Cognitive Services. In that article, I demonstrated how to leverage some simple REST APIs to have the Cognitive Services describe an image or extract the emotion of a person in the picture.

In this edition, I will go a little deeper and show you how to use the Custom Vision API and Content Moderator Services to implement content moderation.

***Warning:** this article may contain (references to) a small amount of offensive language to demonstrate content moderation.*

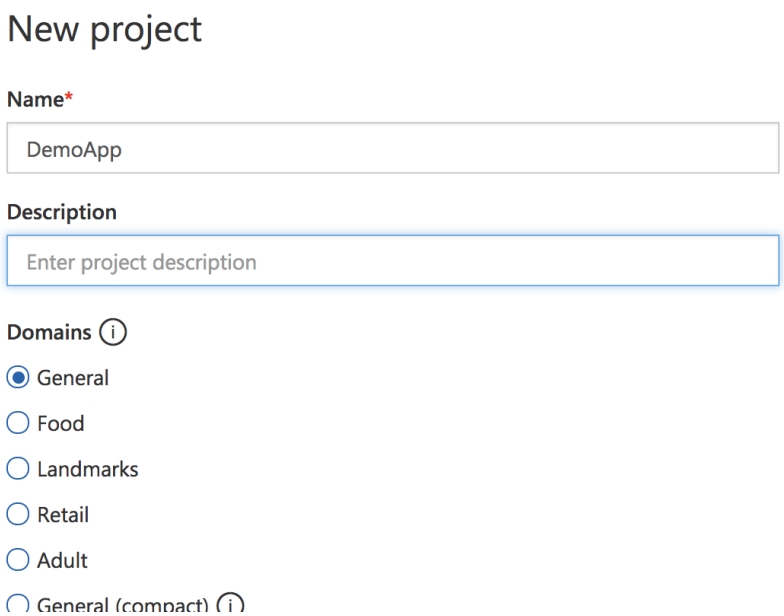
Training Your Own Model

When using Cognitive Services, you basically are using Azure's highest level of APIs. All artificial intelligence and machine learning technologies are based on trained models. These models contain information on data, that is entered, and through this data, a machine can 'learn' to tell things apart like what objects are seen in an image or detect emotions.

With Cognitive Services, these models are shared. All the data that is going through Azure is used to train the models even better. But, like with everything on Azure, you also have the ability to go a bit deeper and do more of the work manually. In this case, we will train our own model by supplying it with images and tag them with what can be seen in that image.

To do this on Azure, we need to head over to a special portal: https://customvision.ai. Login or create an account if you don't have one yet. Please note that you can start for free, but if you choose to continue to use it, some costs might be involved. For more information on that, please refer to this link: https://azure.microsoft.com/en-us/pricing/details/cognitive-services/custom-vision-service/.

After you have logged in, you will land in an empty portal where you can't do much other than create a new project, which is exactly what we will do.



To create a project, you only need to do two things: name the project and choose the domain that describes your target the best.

If you will be using the custom vision API to identify food, it's best to pick that domain since it will be optimized to better recognize food-related images. If you're unsure, choose the 'General' domain like I did for this example.

As you might have noticed there are also some domains at the bottom that are marked as 'compact'. When using one of these domains, you have the ability to export the models that you create and use them locally on your mobile device. This means that you do not have to call the Azure APIs for getting results on your image. The speed when using local models is super fast and it really is great functionality. For brevity, I will not go into details for this article, but feel free to explore.

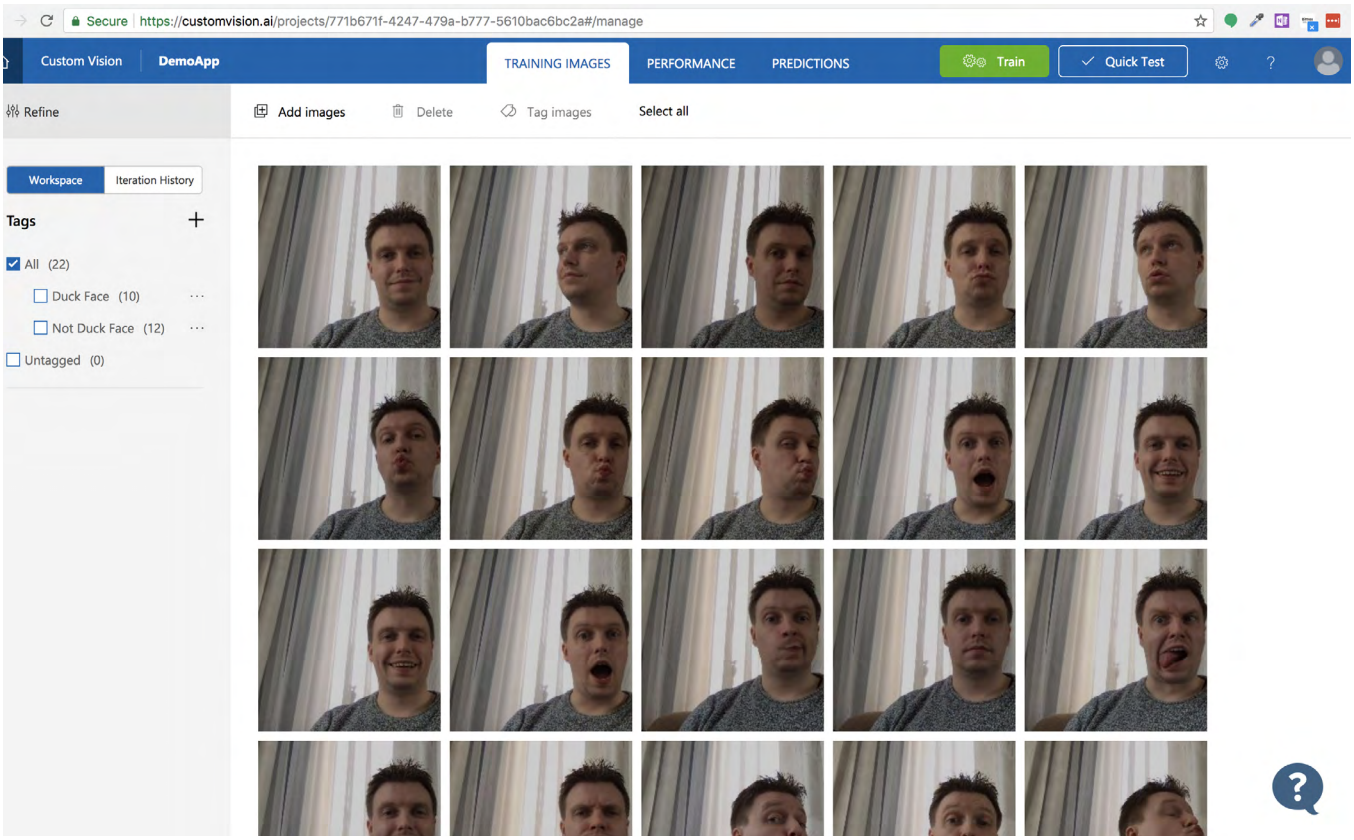
After completing the form, create the project and we're ready to start. Now, it is time to start adding images!

You can upload multiple images at once or one by one. When you have selected the desired images, you need to add one or more tags that apply to that set of images. With these tags, the custom vision API will be able to tell you if another image looks more like tag A or tag B. It might sound a bit cryptic right now, so let me tell you about the sample app that we will be building.

A while ago my good friend Jim Bennett (@jimbobbennett), now one of the Cloud Developer Advocates at Microsoft, created a great sample app to show the capabilities of the custom vision API.

In this app, we mimic the entry page of a social media app but apply some content moderation. More precisely, on our social media platform, we do not tolerate duck faces! Therefore, we will train our model to distinguish duck face pictures from non-duck face pictures, so we can block them. The code for this sample app can be found here: https://github.com/jfversluis/Amazing-Social-Media.

Let's quickly go back to training our model. This is where the fun starts. To properly train our model, we are going to need images of people making a duck face and people not making a duck face. Start running around, grab whoever you can find to take random images making or not making duck faces and upload them with the right tags to the portal. Unfortunately, I was alone while creating this model, but a result of my photoshoot can be seen in the screenshot below.



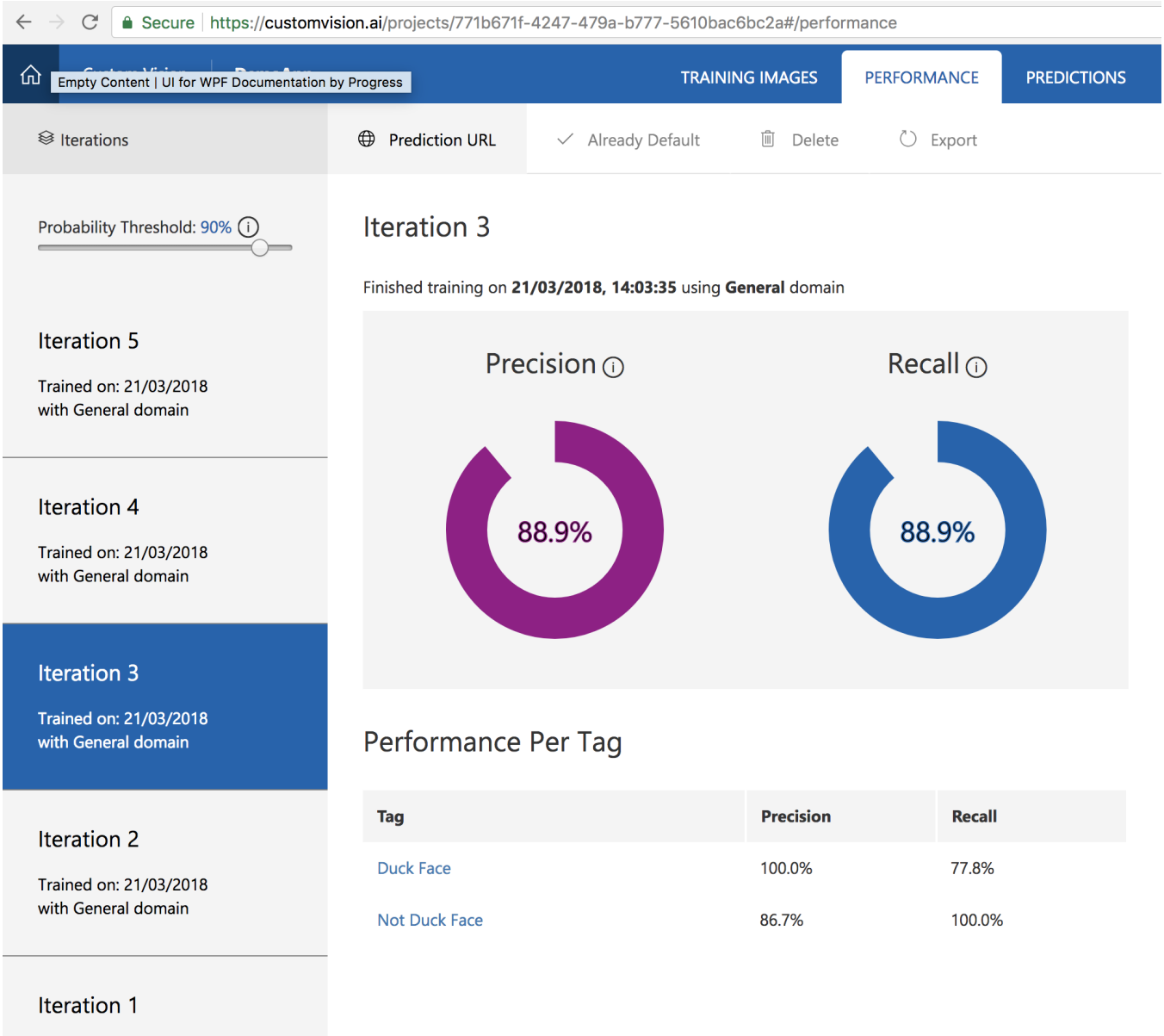
As you can see there are as little as 22 images in this project, which is enough to make a pretty accurate assumption as we will see in a little bit. When you have added some images, a minimum of five per tag, click the green 'Train' button at the top. This will turn all the images into a new model iteration.

In the portal, at the top of the screen, you can also see a tab called 'Performance'. When we go into this screen, you will see the iterations on the left side of the screen. A new iteration will be created whenever you click the train button. Each iteration has its own *precision* and *recall*. These terms are a bit tricky, so pay attention.

The **precision** tells you that whenever we make a prediction, how likely is it that we are right?

The **recall** tells you out of all images that should have been classified correctly, how many did your classifier identify correctly?

The screenshot below shows you that I have a couple of iterations and actually iteration 3 turns out to be the one with the best results.



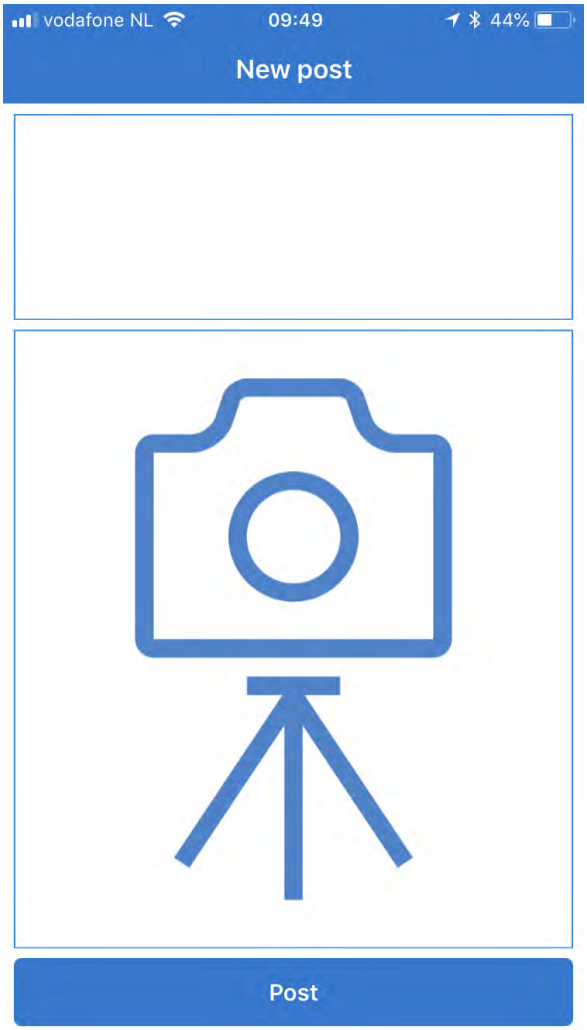
To enable you to test out different iterations and their results, you are able to reach each iteration at its own URL. You can also specify one iteration as the default one. Whenever you do not specify a specific iteration in your REST call, this iteration will be used.

At the very top of the screen, next to the train button is also a 'Quick Test' button which allows you to upload an image and test your current model.

The last tab in the top bar is 'Predictions'. Here you can review, on a per iteration basis, which images have been received and what their results were. You can use these images to further train your model. Just pick the ones that give you false positives, tag them the right way and your model will improve.

Using Custom Vision in our Apps

Everything is in place now to start leveraging all this power from our Xamarin app. In the screenshot below you can see a somewhat basic UI. As I have mentioned before; the app mimics a screen that would be doing a post on a social network. It can consist of a text at the top and image at the bottom.



Content Moderation on Text

One thing we haven't discussed is applying content moderation on a text. This is something that is very easy to do and does not require you to train your own models, since there is another service that Azure provides you out of the box. Technically speaking you could also use the Content Moderator Services for image moderation, but where's the fun in that?

To get a good overview of all the services associated with content moderation, have a look at the documentation page: https://docs.microsoft.com/en-us/azure/cognitive-services/content-moderator/overview.

As mentioned in my previous article, all cognitive (and related) services are easily accessible by REST calls which can be accessed by any tool/app that can make HTTP calls. But since I will be using a Xamarin app, I can just use the NuGet packages that are available.

For the text moderation, we will need to install the `Microsoft.CognitiveServices.ContentModerator` package. To use these services, you will need an API key from the associated portal: https://contentmoderator.cognitive.microsoft.com/. This exercise should be pretty straight-forward, so I won't get into the details.

When we look at the implementation of the text moderation in context of our app, the main code looks like this:

```
public async Task<bool> ContainsProfanity(string text)
{
  InitIfRequired();

  if (string.IsNullOrEmpty(text)) return false;

  var lang = await _client.TextModeration.DetectLanguageAsync("text/plain", text);
  var moderation = await _client.TextModeration.ScreenTextAsync(lang.
  DetectedLanguageProperty, "text/plain", text);
  return moderation.Terms != null && moderation.Terms.Any();
}
```

With this method we check if the supplied string value contains any content that we want to block. First, we need to detect the language so that Azure knows what dictionary has to be checked. By doing this, we minimize the changes of getting false-positives.

Then there is one initialization line to check whether or not the client is ready.

There are multiple sides to this, one of them is known by the name of Scunthorpe Problem (https://en.wikipedia.org/wiki/Scunthorpe_problem). As you might notice, Scunthorpe, which is a perfectly normal

name for a town in England, contains the word c*nt, which is offensive. Another problem is words that appear in multiple languages, for instance, *Mein Schwein ist Dick* means that 'my pig is fat' in German. But d*ck in English is again offensive.

The models that Azure implements, try to detect these exceptions the best they can, and with all the data that is coming in by using these services, the results will only get better. But don't be surprised when something does slip in.

There are some more helpers for text moderation that come in quite handy. They can also detect personal identifiable information (PII) like email addresses, mailing addresses, phone numbers, etc. The content moderation service can also tell you if a human review is recommended. That way you can put suspected malicious content on a separate queue for a human to review before you post it somewhere publicly.

An example of a result JSON object is shown here.

```
{
  "OriginalText": "Questions? email me at somename@microsoft.com or call me at
  1-800-333-4567",

  "NormalizedText": "Questions? email me at some name@ Microsoft. com or call me at
  1- 800- 333- 4567",

  "AutoCorrectedText": "Questions? email me at some name@ Microsoft. com or call me
  at 1- 800- 333- 4567",

  "Misrepresentation": null,

  "PII": {
   "Email": [
    {
      "Detected": "somename@microsoft.com",
      "SubType": "Regular",
      "Text": "somename@microsoft.com",
      "Index": 23
    },
    {
      "Detected": "me@somename@microsoft.com",
      "SubType": "Suspected",
      "Text": "me at somename@microsoft.com",
      "Index": 17
    }
   ],
   "IPA": [ ],
   "Phone": [
    {
      "CountryCode": "US",
      "Text": "1-800-333-4567",
      "Index": 60
    }
   ],
   "Address": [ ],
   "SSN": [ ]
  },
  "Classification": {
   "ReviewRecommended": false,
   "Category1": { "Score": 0.0065943244844675064 },
   "Category2": { "Score": 0.14019052684307098 },
   "Category3": { "Score": 0.0043589477427303791 }
```

```
  },
  "Language": "eng",
  "Terms": [
   {
    "Index": 32,
    "OriginalIndex": 32,
    "ListId": 233,
    "Term": "Microsoft"
   }
  ],
  "Status": {
   "Code": 3000,
   "Description": "OK",
   "Exception": null
  },
  "TrackingId": "bf162866-73e7-49f7-8a89-aa616a542f32"
}
```

This example is taken from: https://github.com/MicrosoftContentModerator/ContentModerator-API-Samples/.

Duck Face Moderation with Custom Vision

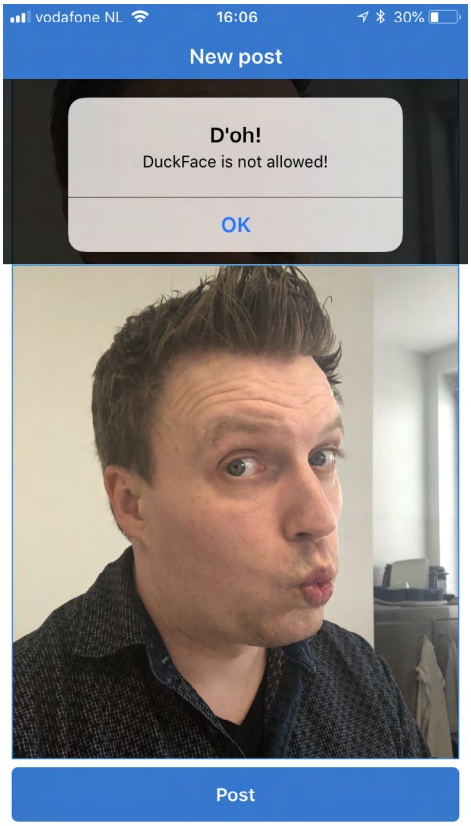
Back to our duck face filter. We also need an API key and need to know the Azure region where our instance of the service is hosted. This can be retrieved from the same portal where we trained our images.

Then getting results from the API is as easy as adding a NuGet package and retrieving the results. For this service, we will install the `Microsoft.Cognitive.CustomVision.Prediction` package.

To determine whether or not a duck face is present, the only code we need is underneath.

```
public async Task<bool> IsDuckFace(MediaFile photo)
{
  InitIfRequired();

  if (photo == null) return false;

  using (var stream = photo.GetStreamWithImageRotatedForExternalStorage())
  {
    var predictionModels = await _endpoint.PredictImageAsync(ApiKeys.ProjectId,
    stream);

    return predictionModels.Predictions
      .FirstOrDefault(p => p.Tag == "Duck Face")
      .Probability > ProbabilityThreshold;
   }
}
```

In this method, we send a picture to the custom vision APIs and retrieve back a result with predictions. From the predictions, we can check the tags that are applied to the images we have just sent to the API. If the tag is 'Duck Face', the model has determined that this picture contains an image of a duck face and we want to block this content.



This screenshot shows the image I sent to the custom vision API and the error that is shown because of the duck face that is detected on it. When a different picture, this time without duck face is sent, it should be allowed without any problems.

A fully functional app can be found at; https://github.com/jfversluis/Amazing-Social-Media. You can also see it in action in my recorded session here: https://youtu.be/tFF8T_AqnBM?t=2m31s.

Conclusion

The entire Cognitive Services Toolkit is a very powerful suite on Azure. There are different entry levels where you can hook in to. When using the Cognitive Services APIs, you can easily leverage the power of machine learning through a few simple REST calls. But when you want to have more fine-grained control over your models or generate offline models, you have the possibility to do so, for example by using Custom Vision.

In this article, we have seen a somewhat humorous usage of this powerful technology, but a more useful and world-changing example can be found here: https://github.com/Azure/ai-toolkit-iot-edge/tree/master/Skin%20cancer%20detection. With this project, you can send images of moles to detect possible skin cancer. Think about it, when a couple of great technologies are combined, you can easily detect medical conditions, possible dangers and other things with the use of a Mixed-Reality device and Azure Cognitive services.

The sky is the limit!

Microsoft® MVP Most Valuable Professional

Gerald Versluis
Author

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

Thanks to Suprotim Agarwal for reviewing this article.

Keerti Kotaru

ANGULAR HTTP CLIENT DEEP DIVE (HEADERS, HTTP EVENTS & INTERCEPTORS)

In this tutorial, we will cover some advanced scenarios. Certain requirements like Security, Logging, Auditing etc., need better flexibility and control on data exchange over HTTP. Often the requirement is overarching and needs a central piece of code addressing it, and this piece of code needs to be unobtrusive and easy to integrate.

Let's begin by looking at ways to access complete HTTP request/response.

Request, Response, Interceptors

A couple of days ago, I published an article on HTTP Client that described the basics of integrating an Angular application with a server-side API or a Web server.

The tutorial addressed the most common scenarios that developers come across while making a HTTP call from an Angular application. Some of these scenarios were:

- making HTTP calls over GET/POST/PUT/DELETE methods.
- error handling scenarios and
- solutions for separating presentation logic from service API integration details.

To learn more, read the tutorial over here, www.dotnetcurry.com/angularjs/1438/http-client-angular

Add headers to the request

Many a times, web servers need additional headers in the request.

Additional headers could include authorization tokens, content type information, session identifiers etc. HTTP Client provides an easy way to add headers to the request.

Consider the following code which adds an authorization header to a GET HTTP call.

```
return this.client.get<Traveller>(`${DATA_ACCESS_PREFIX}`,{
 headers: new HttpHeaders({
  “authorization”: aUserIdentifier
 })
})
```

The GET call has an additional object as a parameter. The object includes field “headers” of type `HttpHeaders`.

We added a new header “authorization” with a value that identifies uniquely for the server. It might perform authorization based on the provided identifier.

Figure-1 shows the network tab on Google Chrome with the additional header in the request.

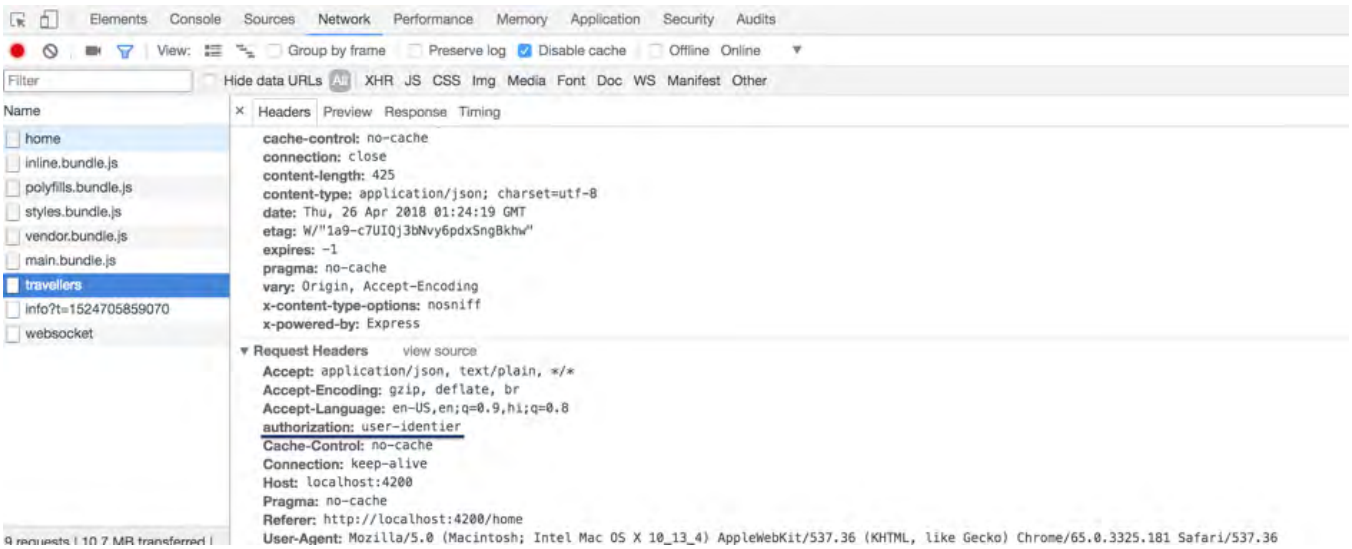


Figure-1 Additional header in the request

Add parameters to the request

Adding request parameters is another way to send additional information to the server. Key value pairs can be added to the request to send additional data. It shows as a querystring on the URL and is commonly used with GET calls. Consider the following code and Figure -2.

```
return this.client.get<Traveller>(`${DATA_ACCESS_PREFIX}`,{
 params: {
  “travellerId”: “1001”
 }
})
```

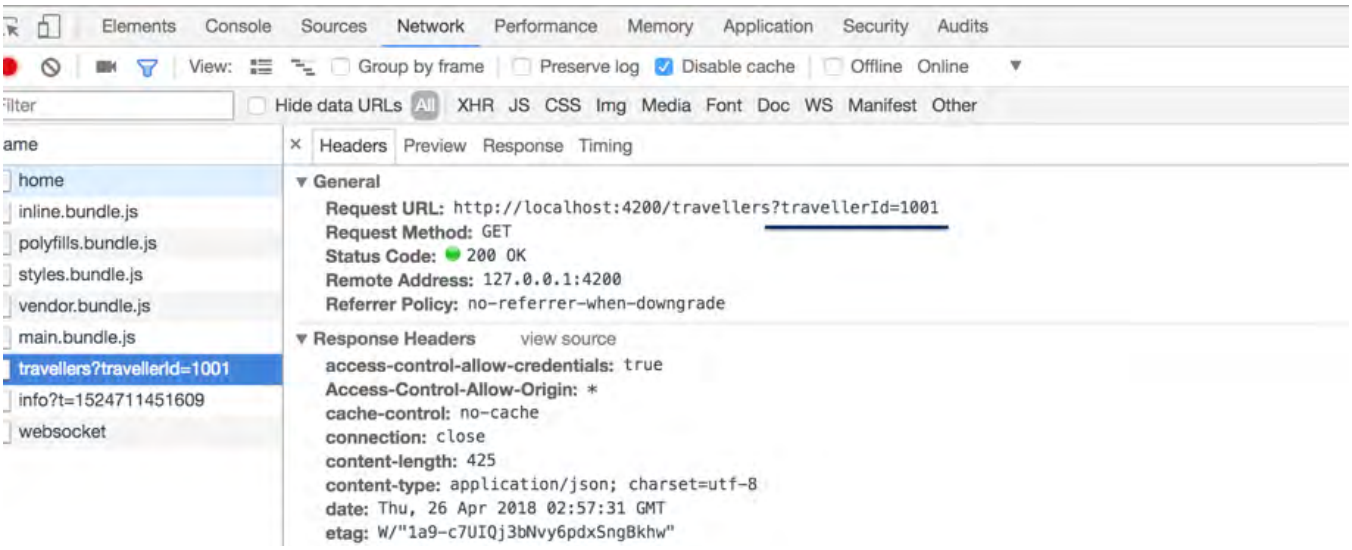


Figure -2 Adding Request parameters

Similar to the code we just saw, the parameter could also be a filter condition. Over here we are providing traveller id to be searched.

Access complete response object

In the previous tutorial, we retrieved typed data objects over the network. The JSON response structure was predefined. The Angular application accessed the data that fit into a predefined TypeScript interface structure.

However, there could be a scenario where we not only need to access response body (JSON data), we also need to access response headers.

Observe Response

`HttpClient` object allows accessing complete response, including headers. In the browser, response body is a JSON object, which can be copied to a typescript interface or class type.

Response headers are key/value pairs. Consider the following code that accesses complete response object.

```
--- data-access-Service.ts---
getTravellersFullResponse():Observable<HttpResponse<Traveller>>{
 return this.client.get<Traveller>(`${DATA_ACCESS_PREFIX}`,{
  observe: “response”
 });
}
```

Notice the return type of the function is Observable of generic type `HttpResponse`, which in-turn has a generic type parameter Traveller. The Traveller object refers to the HTTP response body.

Consider the following code snippet calling the data-access-service and using the response object.

```
--- traveller-list.component.ts ---
travellers: Array<Traveller> = [];
 this.dataAccess.getTravellersFullResponse()
  .subscribe(
   (response) => {
```

```
    this.travellers = this.travellers.concat(response.body);
    response.headers.keys().map( (key) => console.log(`${key}: ${response.headers.
    get(key)}`));
   }
  );
```

As mentioned in the paragraph above, based on the generic parameter, `HttpResponse` is expected to return body of type Traveller (`HtpResponse<Traveller>`).

In the component code, instead of using the returned value from the service function, use body to bind to the UI. Notice, `travellers`, a class level object in the component. This object is available for binding to HTML controls in the template. It is an array of type `Traveller.`

Headers are key/value pairs. Hence loop through headers and retrieve values based on the key. The sample console logs the headers. Consider Figure-3.

```
Angular is running in the development mode. Call enableProdMode() to enable the production mode.
pragma: no-cache
date: Wed, 25 Apr 2018 03:22:31 GMT
x-content-type-options: nosniff
x-powered-by: Express
vary: Origin, Accept-Encoding
content-type: application/json; charset=utf-8
access-control-allow-origin: *
cache-control: no-cache
access-control-allow-credentials: true
connection: close
content-length: 425
etag: W/"1a9-c7UIQj3bNvy6pdxSngBkhw"
expires: -1
>
```

Figure-3 Header log

Observe HttpEvent

In the prior section, when response was observed, it returned a `HttpResponse` object. We had access to the whole response, instead of just the body. This approach has more details about the response that is receiving a JSON object.

However, if we need an even better control and need to access individual events or steps while making a HTTP call, observe `HttpEvent`. It works at a much lower level. It exposes events that capture progress of both the request and response.

Note: In a forthcoming section, all the six events with HttpEvent are listed.

To get started with observing events, consider the following code.

```
--- data-access-Service.ts---
getTravellersResponseByEvents(){
 return this.client.get(`${DATA_ACCESS_PREFIX}`,{
  observe: “events }); }
```

Notice, the value for observation is *events*. The `getTravellersResponseByEvents` returns `HttpEvents` of generic type object (HttpEvents<object>).

Consider the following code. The `subscribe` function is called at every stage of the HTTP call. Progress information is much more detailed. On subscribe, the sample has two out of six events. The `if` condition checks for Http Event Type received. In the sample, we update the status message when a request is sent, a response is received and is ready to use.

```
--- traveller-list.component.ts ---
this.dataAccess.getTravellersResponseByEvents()
 .subscribe( (result) => {
  if(result.type === HttpEventType.Sent){
   console.log("request sent");
   this.messages.push("request sent"); // update status message
  }else if(result.type === HttpEventType.Response){
   console.log("response obtained");
   this.messages.push("response ready"); // update status message
   this.travellers = this.travellers.concat(result.body as Array<Traveller>) //
   object is ready to use.
 }
});
```

See the following code sninppet for HTML template that shows status messages.

In the TypeScript component code we just saw, the `messages` object holds status messages. At every stage the array is pushed with new messages. The template also shows list of travellers. This list is updated to an object named `travellers` (in the component). It is done once the result type is `HttpEventType.Response`.

```
--- traveller-list.component.html ---
<div *ngFor="let msg of messages">
 <div> {{ msg }}</div>
</div>
<hr />
<div *ngFor="let traveller of travellers">
 <div> {{traveller.lastName}}, {{traveller.firstName}} is {{traveller.age}} years
old.</div>
 <div> City: {{traveller.city}}. Country: {{traveller.country}}</div>
 <div><button (click)="deleteTraveller(traveller.id)">Delete</button></div>
 <br />
</div>
```

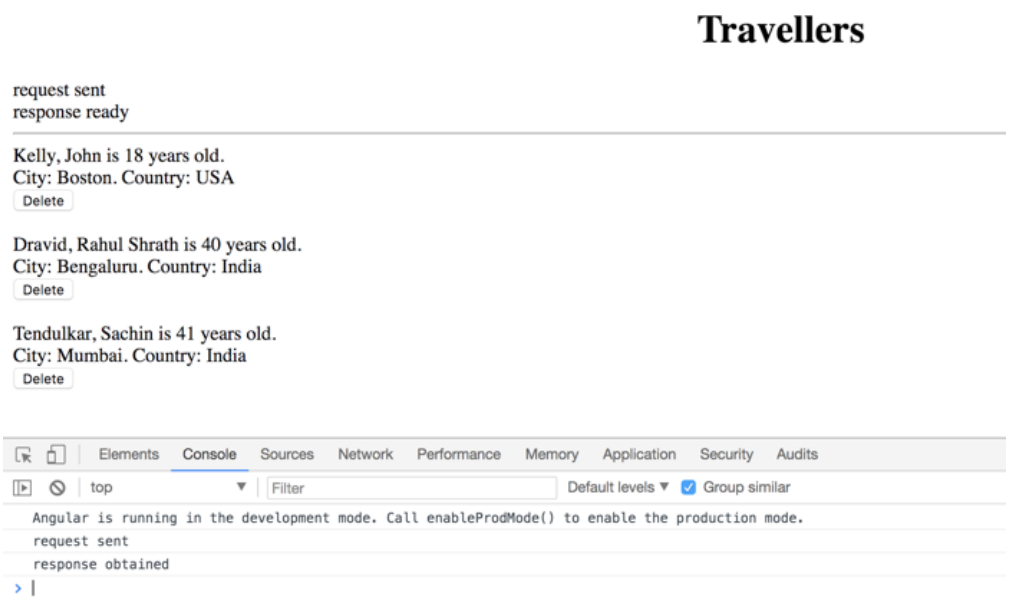


Figure -4 Status Messages

Following is the complete list of HTTP event types available.

1. HttpEventType.Sent (numeration value 0) - Request sent
2. HttpEventType.UploadProgress (Enumeration value 1) - Request upload in progress
3. HttpEventType.ResponseHeader (Enumeration value 2) - Response headers received
4. HttpEventType.DownloadProgress (Enumeration value 3) - Response download in progress
5. HttpEventType.Response (Enumeration value 4) - Response ready to use
6. HttpEventType.User (Enumeration value 5) - For a custom event.

Show non-JSON data

There are always scenarios to download files, BLOB (binary large object) or text from the remote server. An Angular application needs to support retrieving such content.

Consider the following sample that retrieves text content and shows it on the page. Notice `responseType` is set as text, anticipating text response (instead of JSON).

```
--- data-access-Service.ts---
getText(){
 return this.client.get(`${BLOBL_ACCESS_PREFIX}/1.txt`,{
  responseType: “text”
 });
}
```

A Component calls and consumes the text response. It sets in on a string variable and shows it on the template.

```
--- show-text.component.ts ---
downloadText(){
 this.dataAccess.getText()
 .subscribe((data) => {
  this.textContent = data;
 }, () => console.log(“error”));
}
--- show-text.component.html ---

<button (click)=”downloadText()”>Download Text </button>
<div>{{textContent}}</div>
```

We may also download a PDF document with a similar approach. Consider the following code where we will use `responseType` blob instead of text.

```
--- data-access-Service.ts---
 getDocument(){
   return this.client.get(`${BLOBL_ACCESS_PREFIX}/1.pdf`,{
     responseType: “blob”
   });
 }
```

The component will use the data slightly different. Will use the BLOB returned from the service and open it on the window to show the file/content.

```
--- show-pdf.component.ts ---
downloadDocument(){
 this.dataAccess.getDocument()
```

```
 .subscribe((data) => {
  console.log(data);
  let url = window.URL.createObjectURL(data);
  window.open(url); // The downloaded file is open with the help of a browser
  window.
 }, () => console.log("error"));
}
```

Interceptors

UI applications commonly need altering of request and response before it's sent on the wire and right after receiving. It could be for various reasons including logging request/responses, adding common authorization headers that the server might need, caching and so on.

Such activities are preferably carried out at a central location. Interceptors are perfect for this requirement.

In Angular, a series of interceptors can be created, which pass the request from one interceptor to the other. They are ordered and the request is passed in sequence. However, response is received in exact reverse order to the requests.

For example, consider the following interceptor

A. Caching Interceptor – stores certain details from a response in local storage or session/storage.
B. Authorization Interceptor - Adds required header to all requests matching a pattern.
C. Logging Interceptor - Logs data certain type of data being exchanged.

Let us consider that the Caching Interceptor is run first. It does required processing on the request and invokes authorization interceptor if the request needs to go to the server.

Say authorization headers are added and passes the request for logging. The logging interceptor processes it and allows the actual network call. When the response is received, it is first received by the Logging Interceptor, which passes it to Authorization Interceptor, which passes it to the Cache Interceptor that finally sends it back to the UI application and components.

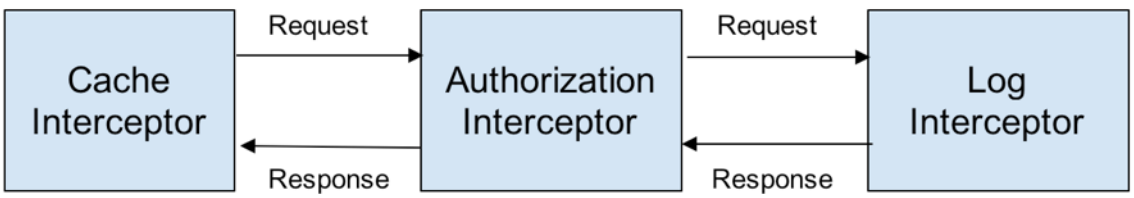


Figure -5 Interceptors in action

Create the first interceptor

To create an interceptor, create a service that implements HttpInterceptor (in module @angular/common/http). To create a service with Angular CLI, run the following command,

```
ng g service services/logging-interceptor
```

Import and implement the HttpInterceptor. The interface needs a definition for the function `intercept`.

Consider the following code. The interceptor doesn't do anything useful yet. It just logs a static message to the browser console.

```
@Injectable()
export class LoggingInterceptorService implements HttpInterceptor {
 intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>{
   console.log("request/response intercepted");
   return next.handle(req);
 }

 constructor() { }
}
```

The Request is passed in as a parameter to the interceptor (or the intercept function). This is how interceptor has access to the request. Considering the function `intercept` returns an Observable, we can access the response every time the observable is resolved or errored out.

Notice, the `intercept` function returns an Observable of generic type `HttpEvent`. That means, interceptor has access to all the six events that occur with a HTTP call. Refer to the Observe Http Event section above for details on the events.

After an interceptor does its job with the request, it needs to call the `handle` function next (which is an object HttpHandler). This process invokes the next interceptor in the chain.

Provide the Interceptor

Interceptor needs to be provided before the service making the Http call. The Injector uses all interceptors that are provided at the time of injecting a service (that makes HTTP calls).

If an interceptor is provided after the service that uses HttpClient instance, it would be ignored.

Note: Providing a service makes it available for the injector. A service is added to providers array while creating an Angular module or a component. It can be provided with a token similar to the following example. If provided without a token, it will be auto created.

In this sample, let's provide the interceptor in the AppModule. Begin by importing token for the interceptor HTTP_INTERCEPTORS (from @angular/common/http). Import the service LoggingInterceptor as well.

Provide the interceptor as shown the following sample:

```
@NgModule({
 declarations: [
   ...
 ],
 imports: [
  ...
 ],
 providers: [
   DataAccessService,
   {provide: HTTP_INTERCEPTORS, useClass: LoggingInterceptorService, multi:true}

 ],
```

```
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Please note, the field multi indicates whether or not the token being provided can be used for multiple services. With the value true, even though in this example we are just using one service, we can provide many services with the same token. See figure-6 for the result as we make HTTP calls.

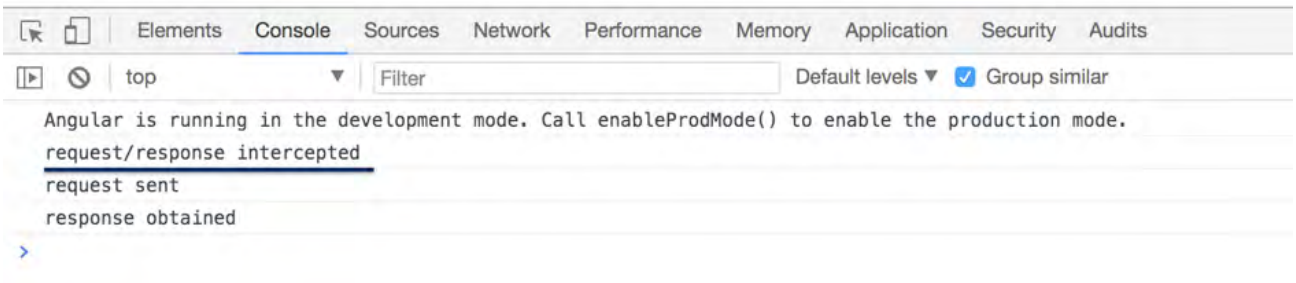


Figure-6 HTTP Calls

Access response in interceptor

As mentioned above, the `intercept` function returns an Observable of type HttpEvent. Once subscribed (in a function that consumes response - do not subscribe in interceptor), the Http call is invoked. We can use the operator tap, imported from rxjs/operators to access the success or failure response.

Consider the following code:

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>{
 console.log("request/response intercepted");
 return next.handle(req)
  .pipe(
   tap(
    (result: HttpEvent<any>) => console.log("response intercepted", result),
    (error: HttpErrorResponse) => console.error("error response intercepted",
error)
   )
  );
 }
```

Please note, the parameter to the `success` function is of type HttpEvent of a generic type. And for the error handler, the parameter is of type HttpErrorResponse.

Figure-7 shows success and error response.

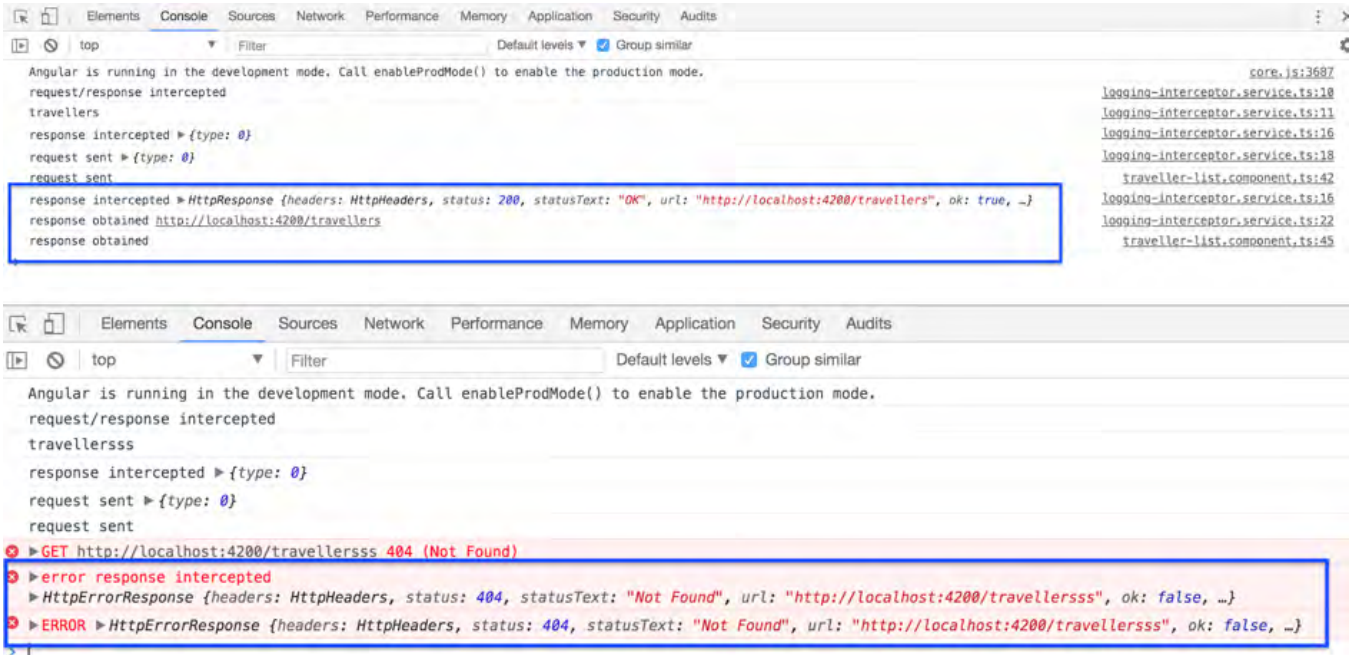


Figure-7 Success Error Response

Conclusion

Most Angular applications' server-side API integration can be achieved with basic HttpClient API. The service is ready-made and provides an easy way to integrate with RESTful services. However, there are times where the developer needs deeper control while making HTTP calls. Concepts discussed in this article are intended to address such scenarios.

We began by providing the ability to access request and response headers. It allows developers to customize the HTTP call. We later addressed writing such custom code in a reusable manner.

By creating an interceptor, we can write code that accesses and modifies headers, massages data from a central location. Without interceptors, developers would need to modify each service function to do custom editing/reading additional details ■

References

Angular documentation for HttpClient -https://angular.io/guide/http

• • • • • • •

Microsoft® MVP Most Valuable Professional

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 Ravi Kiran for reviewing this article.

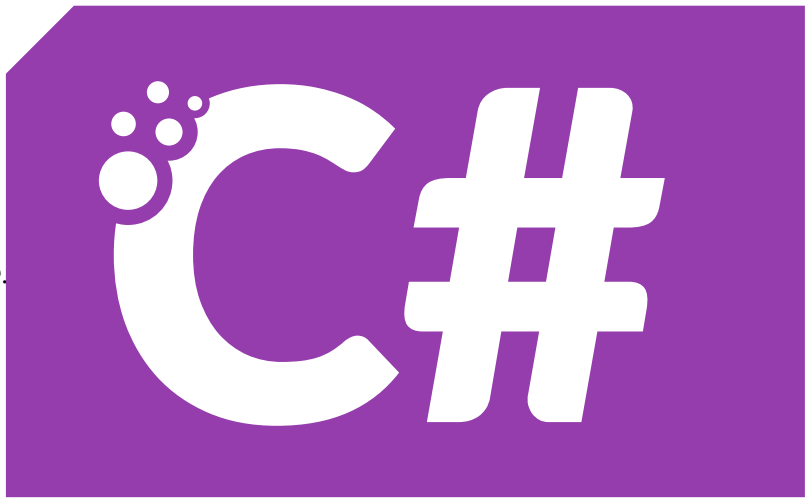
Damir Arh

Support for **Dynamic Binding in** C#

At its roots, C# is a strongly typed and type-safe language. In most of the code, the compiler will be able to perform full static type checking and fail the build if it detects any type errors. Despite that, the language also provides means to avoid compile-time type checking and even allows writing dynamic code in which the types are not yet known at compile time.

Type Safety

Type safety is a principle in programming languages which prevents accessing the values in variables inconsistent with their declared type. There are two kinds of type safety based on when the checks are performed:



- Static type safety is checked at compile time.
- Dynamic type safety is checked at runtime.

Without the checks, the values could be read from the memory as if they were of another type than they are, which would result in undefined behavior and data corruption.

Static Type Safety

C# implements measures for both kinds of type safety.

Static type safety prevents access to non-existent members of the declared type:

```
string text = “String value”;
int textLength = text.Length;
int textMonth = text.Month; // won’t compile

DateTime date = DateTime.Now;
int dateLength = date.Length; // won’t compile
int dateMonth = date.Month;
```

The two lines marked with a comment won’t compile, because the declared type of each variable doesn’t have the member we are trying to access. This check cannot be circumvented even if we try to cast the variable to the type which does have the requested member:

```
string text = “String value”;
int textLength = text.Length;
int textMonth = ((DateTime)text).Month; // won’t compile

DateTime date = DateTime.Now;
int dateLength = ((string)date).Length; // won’t compile
int dateMonth = date.Month;
```

The two lines with comments still won’t compile. Although the type we’re trying to cast the value to has the member we’re accessing, the compiler doesn’t allow the cast because there is no cast operation defined for converting between the two types.

Dynamic Type Safety

However, static type safety in C# is not 100% reliable. By using specific language constructs, the static type checks can still be circumvented as demonstrated by the following code:

```
public interface IGeometricShape
{
  double Circumference { get; }
  double Area { get; }
}

public class Square : IGeometricShape
{
  public double Side { get; set; }

  public double Circumference => 4 * Side;

  public double Area => Side * Side;
}

public class Circle : IGeometricShape
{
  public double Radius { get; set; }

  public double Circumference => 2 * Math.PI * Radius;

  public double Area => Math.PI * Radius * Radius;
}

IGeometricShape circle = new Circle { Radius = 1 };
Square square = ((Square)circle); // no compiler error
var side = square.Side;
```

The line marked with the comment will compile without errors, although we can see from the code that the `circle` variable contains a value of type `Circle` which can't be cast to the `Square` type. The compiler does not perform the static analysis necessary to determine that the `circle` variable will always contain a value of type `Circle`. It only checks the type of the variable. Since the compiler allows downcasting from the base type (the `IGeometricShape` interface) to a derived type (the `Square` type) because it might be valid for certain values of the variable, our code will compile.

Despite that, no data corruption will happen because of this invalid cast. At runtime, the compiled code will not execute. It will throw an `InvalidCastException` when it detects that the value in the `circle` variable can't be cast to the `Square` type. This is an example of dynamic type safety in C#. To avoid the exception being thrown at runtime, we can include our own type checking code and handle different types in a different way:

```
if (shape is Square)
{
  Square square = ((Square)shape);
  var side = square.Side;
}
if (shape is Circle)
{
  Circle circle = ((Circle)shape);
  var radius = circle.Radius;
}
```

Since `object` is the base type for all reference types in .NET, static type checking can almost always be circumvented by casting a variable to the `object` type first and then to the target type. Using this trick, we can modify the code from our first example to make it compile:

```
string text = "String value";
int textLength = text.Length;
int textMonth = ((DateTime)(object)text).Month;

DateTime date = DateTime.Now;
int dateLength = ((string)(object)date).Length;
int dateMonth = date.Month;
```

Of course, thanks to dynamic type safety in C#, an `InvalidCastException` will still be thrown. This approach was taken advantage of in .NET to implement common data structures before the introduction of generics while still preserving type safety, albeit only at runtime. These data structures are still available today, although their use is discouraged in favor of their generic alternatives.

```
var list = new ArrayList();
list.Add("String value");
int length = ((string)list[0]).Length;
int month = ((DateTime)list[0]).Month; // no compiler error
```

`ArrayList` is an example of a non-generic data structure in the .NET framework. Since it doesn't provide any type information about the values it contains, we must cast the values back to the correct type on our own before we can use them. If we cast the value to the wrong type as in the line marked with a comment, the compiler can't detect that. The type checking will only happen at runtime.

Dynamic Binding

In all the examples so far, we used static binding. If the code tried to access a non-existent member of a type, it would result in a compilation error. Even when we cast a value to a different type, the compiler still performed the same checks for this new type which did not necessarily match the actual value.

In contrast, with dynamic binding, the compiler does not do any type checking at compile time. It simply assumes that the code is valid, no matter which member of the type it tries to access. All checking is done at runtime, when an exception is thrown if the requested member of the type does not exist.

Dynamic binding is an important feature of dynamically-typed languages, such as JavaScript. However, although C# is primarily considered a statically-typed language with a high level of type safety, it also supports dynamic binding since C# 4 .0 when this functionality was added to simplify interaction with dynamically-typed .NET languages such as IronPython and IronRuby, introduced at that time.

The Dynamic Keyword

The core of dynamic binding support in C# is the `dynamic` keyword. With it, variables can be declared to be dynamically-typed. For such variables, static type checking at compile time is completely disabled, as in the following example:

```
dynamic text = "String value";
int textLength = text.Length;
int textMonth = text.Month; // throws exception at runtime
```

```
dynamic date = DateTime.Now;
int dateLength = date.Length; // throws exception at runtime
int dateMonth = date.Month;
```

In the above example, using the `dynamic` keyword doesn't make much sense. It only postpones the checking till runtime, while it could have already been done at compile time if we used static types. It brings no advantages but introduces two significant disadvantages:

- Our code could easily be defective as in the example above because type errors are not detected at compile time.

- Even if there are no defects in code, additional type checks are required at runtime which affects performance.

Of course, there are better use cases for the `dynamic` keyword, otherwise, it wouldn't have been added to the language. For example, it can be used to avoid the restriction of anonymous types to a single method because the type name is visible only to the compiler and therefore can't be declared as the return value of a method. If the method is declared with a `dynamic` return value, the compiler will allow it to return a value of an anonymous type, although the exact type isn't (and can't be) declared explicitly:

```
public dynamic GetAnonymousType()
{
  return new
  {
    Name = "John",
    Surname = "Doe",
    Age = 42
  };
}
```

Of course, the returned value can be fully used outside of the method, with the disadvantage that no IntelliSense and compile time type checking will be available for it as neither the compiler nor the editor is aware of the actual type being returned:

```
dynamic value = GetAnonymousType();
Console.WriteLine($"{value.Name} {value.Surname}, {value.Age}");
```

Since anonymous types are declared as `internal`, this will only work within a single assembly. The code in different assemblies will still be able to get the value when calling the method, but any attempts to access its members will fail with a `RuntimeBinderException`.

However, that's only the beginning of what can be done with `dynamic` type. A great example of how much value dynamic binding can add when used correctly is the JSON.NET library for serializing and deserializing JSON. It can be best explained with the following example:

```
string json = @"
{
  ""name"": ""John"",
  ""surname"": ""Doe"",
  ""age"": 42
}";

dynamic value = JObject.Parse(json);
Console.WriteLine($"{value.name} {value.surname}, {value.age}");
```

Like the anonymous type example from before, the `Parse` method returns a value which can be dynamically bound. This allows the JSON object properties to be accessed as the properties of the parsed object although the compiler could have no knowledge about them. We can pass in any JSON object at runtime and still access its properties in code. This functionality couldn't have been implemented with anonymous types.

So, how was it implemented then? There are two helper objects in the .NET framework which we can use to implement similar functionalities.

ExpandoObject

`ExpandoObject` is the simpler one of the two. It allows members to be added to or removed from any instance of it at runtime. If assigned to a variable declared as `dynamic`, these members will be dynamically bound at runtime:

```
dynamic person = new ExpandoObject();
person.Name = "John";
person.Surname = "Doe";
person.Age = 42;

Console.WriteLine($"{person.Name} {person.Surname}, {person.Age}");
```

Members are not limited to being properties. They can be methods as well. To achieve that, a corresponding lambda can be assigned to a member, cast to a matching delegate type. Later, it can be invoked with the standard syntax for calling a method:

```
person.ToString = (Func<string>)(() => $"{person.Name} {person.Surname}, {person.
Age}");
Console.WriteLine(person.ToString());
```

To see which members were added to the `ExpandoObject` at runtime, we can cast the instance to the `IDictionary<string, object>` interface and enumerate its key-value pairs:

```
var dictionary = (IDictionary<string, object>)person;
foreach (var member in dictionary)
{
  Console.WriteLine($"{member.Key} = {member.Value}");
}
```

Using the same interface, we can also remove a member at runtime:

```
dictionary.Remove("ToString");
```

While `ExpandoObject` can be useful in simple scenarios, it gives very little control to the code instantiating and initializing it. Consuming code always has full access to the instance and can modify it to the same extent as the code creating it. Methods are added to it in a similar way to properties, i.e. as lambdas assigned to each individual instance. There is no way to define methods at the class level and automatically make them accessible from every instance.

DynamicObject

DynamicObject gives more control to the developer and avoids many of the disadvantages of ExpandoObject but at the same time requires more code to achieve similar results:

```
class MyDynamicObject : DynamicObject
{
  private readonly Dictionary<string, object> members = new Dictionary<string,
  object>();

  public override bool TryGetMember(GetMemberBinder binder, out object result)
  {
    if (members.ContainsKey(binder.Name))
    {
      result = members[binder.Name];
        return true;
    }
    else
    {
      result = null;
      return false;
    }
  }

  public override bool TrySetMember(SetMemberBinder binder, object value)
  {
    members[binder.Name] = value;
    return true;
  }
}
```

With the above implementation, `MyDynamicObject` supports dynamic adding of properties at runtime just like the `ExpandoObject` does:

```
dynamic person = new MyDynamicObject();
person.Name = "John";
person.Surname = "Doe";
person.Age = 42;

Console.WriteLine($"{person.Name} {person.Surname}, {person.Age}");
```

We can even add methods to it, using the same approach: by assigning lambdas as delegates to its members:

```
person.AsString = (Func<string>)(() => $"{person.Name} {person.Surname}, {person.
Age}");
Console.WriteLine(person.AsString());
```

We didn't use the standard method name `ToString` this time because it wouldn't work as expected. If we did, `DynamicObject`'s default `ToString` method would still be called. Unlike `ExpandoObject`, `DynamicObject` first tries to bind all member accesses to its own members and only if that fails, it delegates the resolution to its `TryGetMember` method. Since `DynamicObject` has its `ToString` method just like any other object in the .NET framework, the call will be bound to it and `TryGetMember` won't be called at all.

Thanks to this behavior, we can implement methods directly in our derived class and they will be invoked as expected:

```
public bool RemoveMember(string name)
{
  return members.Remove(name);
}
```

If we need a more dynamic behavior, we can still override the `TryInvokeMember` method. In the following example, we expose all methods of our internal dictionary without having to write specific code for each method. This gives the caller access to it, similar to what `ExpandoObject` does.

```
public override bool TryInvokeMember(InvokeMemberBinder binder, object[] args, out
object result)
{
  try
  {
    var type = typeof(Dictionary<string, object>);
    result = type.InvokeMember(binder.Name,
    BindingFlags.InvokeMethod | BindingFlags.Public | BindingFlags.Instance,
    null, members, args);
    return true;
  }
  catch
  {
    result = null;
    return false;
  }
}
```

With the methods above present in our class, we can invoke both the statically defined `RemoveMember` method and all the dynamically defined methods the same way, if we are using dynamic binding. Any methods not defined in our class will be delegated by `TryInvokeMember` method to the inner dictionary using reflection.

```
person.Remove("AsString");
Console.WriteLine(person.ContainsKey("AsString"));
```

If we wanted to, we could also directly expose the inner dictionary when casting the object to the correct dictionary type just like `ExpandoObject` does. Of course, one way would be to overload the cast operator:

```
public static explicit operator Dictionary<string, object>(MyDynamicObject
instance)
{
  return instance.members;
}
```

If we needed to implement casting in a more dynamic manner, we could always override the `TryConvert` method instead:

```
public override bool TryConvert(ConvertBinder binder, out object result)
{
  if (binder.Type.IsAssignableFrom(members.GetType()))
  {
    result = members;
    return true;
  }
  else
  {
    result = null;
    return false;
  }
}
```

Both implementations would give us access to the inner dictionary by casting an instance of the class to

the correct type:

```
var dictionary = (Dictionary<string, object>)person;
```

Both the ExpandoObject and the `DynamicObject` implement the same `IDynamicMetaObjectProvider` interface to implement the underlying dynamic binding logic. To get full control over the binding process, we can do the same. However, if we can achieve our goal by using either `ExpandoObject` or `DynamicObject`, they are a better choice because they make our implementation much simpler.

Conclusion:

Having support for dynamic binding in C# opens some possibilities which would not be available in a strictly strong typed language. We will typically be using those when interacting with other languages and runtime environments which are more dynamically typed, or when using third-party libraries which take advantage of dynamic binding to improve usability. In certain scenarios, we might even implement some dynamically bound classes ourselves. However, we should always do our best to only use dynamic binding when this is the only or the best available approach. Although we could also use it to circumvent strong typing in other cases, we should try to avoid that because the seemingly simpler code will very probably expose us to hidden bugs which could otherwise be detected by the compiler ■

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.

Want this magazine delivered to your inbox ?

Subscribe here

www.dotnetcurry.com/magazine/

* No spam policy

David Pine

TypeScript – A Tour of Generics

TypeScript is an amazing innovation with how we as developers write JavaScript applications.

TypeScript boasts its language services as a first-class citizen. These language services empower your favorite integrated development environment with advanced statement completion, refactoring, type-checking, compilation and even flow analysis.

In addition to all these capabilities, the language itself offers a more "grown up" approach to JavaScript development. As a superset of JavaScript, TypeScript really is the answer to many problems.

In this tutorial, we'll focus on **generics in TypeScript**. We'll cover how TypeScript allows developers to leverage generics, the generic syntax, limitations with generics and other key considerations.

Generics in TypeScript - Introduction

I've said it before and I'll say it again, software exists to aid with the movement and manipulation of data. When it's stated like this – it's rather boring. While the context of the data can mean all the difference in how emotionally developers are connected to their source code, it is an art form we should all take pride in. I'm an advocate of software craftsmanship and education. I believe that there is always a better way of doing something until you run out of time or priorities change. With this we consider JavaScript, a language prototyped in only ten days – which really explains a lot. Despite negativity coming from the community about JavaScript, it is one of the world's most powerful, prevalent and popular programming languages today. JavaScript runs nearly everywhere, and with such demand comes an opportunity for improvement.

Compile vs Transpile

Since we're discussing craftsmanship and education, let us clear up some misconceptions. There is still a lot of confusion in our industry as it pertains to TypeScript compilation. There are those who will argue it is not compilation and those who believe it's only transpiled.

The act of transpiling your TypeScript into JavaScript, is referred to as compilation. You are in fact compiling your TypeScript code into JavaScript. The term "transpile" is a derivative of "compile". The distinction between the two has to do with the level of abstraction, i.e. when you compile C# into IL, the code is seemingly unrecognizable from its original source – as it has been so heavily abstracted. Using this understanding, let's apply this notion to TypeScript.

The resulting JavaScript from a TypeScript compilation varies depending on the ECMAScript version you're targeting – there is an inverse correlation between the level of abstraction and the target ECMAScript version. If you're targeting ESNext in your tsconfig.json, which supports the latest proposed ES features – the level of abstraction in compiling TypeScript into JavaScript is significantly smaller than if you were targeting ES3. As such it would be fair to refer to this as "transpiling"; however, it is always safe to call this a "compilation" because transpiling is a form of compiling. It is not necessarily incorrect for someone to use these interchangeably – although one might be more accurate.

Generic Syntax

In TypeScript, the syntax for generics is rather simple. We use angle brackets to enclose the generic type-parameter. Let's take the name "TypeScript" for example, imagine that we have a generic class named `Script` and we constrain the generic type-parameter to extend from `Type`. We would end up with the following:

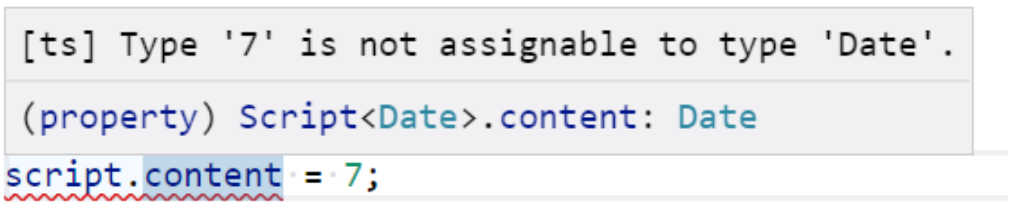
```
type Type = number | Date | string;

class Script<T extends Type> {
  constructor(
    public content: T,
    public name: string) { }
}
```

This simple class has two properties, `content` and `name`. Notice that `name` is a `string` and `content` is of type `T`. Since we have defined the constructor parameter for content as a type of `T`, this `Script` class takes on the shape of anything that satisfies its type constraint. In this case, `T` must extend `Type` and `Type` is defined as a type alias. The type alias is a union of `number`, `Date` and `string`. This means that `T` must be one of those types. Let's instantiate a new `Script` object instance with a `Date` as its parameter type

argument.

```
const script = new Script<Date>(new Date(), “Date object.”);
const date = script.content;
```

The value of the `content` member is of type `Date` and the language services enforce and recognize this as the only type for this member. If we try to reassign the `content` value of this instance to a different type, we will get an error.



It is important to note that we are even attempting to reassign the value to one of the accepted types we are constrained to, `number`.

This does not matter.

The constraint is applied when calling the constructor and lives for the life of the instance and cannot be changed afterwards. When we instantiate a Script object, the type argument is actually optional since it is inferred by the type passed into the constructor. As such we can simplify our declaration:

```
const script = new Script(new Date(), “Date object.”);
const date = script.content;
```

But it doesn't hurt to be explicit. This is more of a stylistic decision that perhaps is best mutually decided by the development team. I personally prefer being implicit as the code is less verbose and more closely resembles JavaScript.

Constraints

At its most pure and simple form, generic syntax for TypeScript is effectively a constraint on a parameter type as was demonstrated above. This is familiar to developers who are accustomed to languages such as C# that have supported generics for years now. Although it is familiar in semantics, the syntax does differ. Instead of defining a type alias we'll have a look at interfaces.

```
interface Sport {
  name: string;
  association: string;
  yearFounded: number;
  getNumberOfTeams(): number;
}
```

We now have a basic interface describing a `Sport`. TypeScript allows you to define, declare and use interfaces. However, interfaces do not exist in JavaScript. When your TypeScript code is compiled to JavaScript, the interfaces go away…they exist solely to empower the language services to reason about the shape of your objects. Likewise, they can be used for generic type constraints.

Our first implementation of this interface will be the `Football` object.

```
class Football implements Sport {
  public yearFounded = 1920;
  public name = 'Football';
  public association = 'NFL';
  public tags = [
    'Pigskin',
    'American Football',
    'Gridiron'
  ];

  public getNumberOfTeams() {
    return 32;
  }
}
```

For the sake of having multiple implementations we will also introduce a `Basketball` implementation of our `Sport` interface.

```
class Basketball implements Sport {
  public yearFounded = 1946;
  public name = 'Basketball';
  public association = 'NBA';
  public description =
    'Boring game, only excitement is the final minute of play!';

  getNumberOfTeams() {
    return 30;
  }

  deleteFromExistence(really: boolean = true): void {
    // If only it were so easy...
  }
}
```

We'll now leverage a generic lambda to output a message that relies on the Sport interface.

```
const sportWriter = <TSport extends Sport>(sport: TSport) => {
  if (sport) {
    const teamCount = sport.getNumberOfTeams();
    console.log(
    `${sport.name} the sport, better known as the ` +
    `(${sport.association}) has ` +
    `${teamCount} teams and was founded in ${sport.yearFounded}.`);
  }
}

sportWriter(new Football());
```

The following will output as follows:

```
Football the sport, better known as the (NFL) has 32 teams and was founded in 1920.
```

The benefits of this use of generics are simple, `sportWriter` will only accept implementations of `Sport`.

Generic Structures

In addition to type parameter constraints, we can build out generic lists and similar data structures that do not rely on constraints – they are simply generic to any type.

```
class List<T> {
    private items: T[] = [];

    public add(value: T): this {
        this.items.push(value);
        return this;
    }

    public remove(value: T): this {
        let index = -1;
        while (this.items
            && this.items.length > 0
            && (index = this.items.indexOf(value)) > -1) {
            this.items.splice(index, 1);
        }
        return this;
    }

    public toString(): string {
        return this.items.toString();
    }
}
```

This represents a generic list, wherein it can contain any single type. This is extremely powerful and ensures type-safety. This means that if a list is instantiated with a type parameter argument of type `string`, the list can only allow the consumer to `add` and `remove string` values.

```
const list = new List<string>();
list.add("David")
  .add("Michael")
  .add("-")
  .add("Michael")
  .add("-")
  .add("Michael")
  .add("-")
  .add("Pine")
  .remove("-");

console.log(list);
console.log(list.remove("Michael"));
```

This code will add a series of strings to the underlying array via the `add` API. Notice that we allow for a sense of functional programming by returning the instance itself after each function call – this enables a fluent API, where we can chain method invocations seamlessly together. Here's the resulting execution output.

List { items: ['David', 'Michael', 'Michael', 'Michael', 'Pine'] }
List { items: ['David', 'Pine'] }

Generics - Benefits

Generics enable common code to be reused and make encapsulation more obvious. An example of this is found during development of an application that performs common operations on different models. Often developers find existing code that acts similar to what they need. This leads to copying and pasting code, changing variables as needed - this is wrong! Instead, developers should take this opportunity to encapsulate the common sequence, steps, or functionality found within the logic they were attempting to replicate – and make it generic if possible. Doing so will encapsulate the common logic, thus reusing existing code.

Generics - Limitations

The generic syntax has a bit of a learning curve and it takes time to fully understand it. The syntax itself can be off-putting at first, but again with time it will become obvious. The one true limitation is really just readability – developers unfamiliar with the syntax might have difficulty reading it initially. However, it is easy to argue that the semantics of other languages that have adopted generic syntax are identical to how TypeScript has implemented this feature. Leveraging generic syntax is a decision the development team must embrace, but a decision nonetheless. With this in my mind, it could be perceived as a limitation.

Conclusion

TypeScript is the evolution of JavaScript that the developer community is actively adopting. The language services are far superior to any other JavaScript tool in existence today. Generics are a fantastic way to demonstrate this powerful advancement for JavaScript development. It's a feature that will help developers reuse common code and is perfect for modularity. I highly recommend that you start using TypeScript generics today!

Resources

- https://www.typescriptlang.org/docs/handbook/generics.html

David Pine

Author

David Pine is a Technical Evangelist and Microsoft MVP working at Centare in Wisconsin. David loves knowledge sharing with the technical community and speaks regionally at meetups, user groups, and technical conferences. David is passionate about sharing his thoughts through writing as well and actively maintains a blog at davidpine.net. David's posts have been featured on ASP.NET, MSDN Web-Dev, MSDN .NET and DotNetCurry. David loves contributing to open-source projects and stackoverflow.com as another means of giving back to the community. David sat on the technical board and served as one of the primary organizers of MKE DOT NET for three years. When David isn't interacting with a keyboard, you can find him spending time with his wife and their three sons, Lyric, Londyn and Lennyx. Follow David on Twitter at @davidpine7.

Thanks to Damir Arh for reviewing this article.