Issue 40 | Jan-Feb 2019

DNC Magazine

www.dotnetcurry.com

USING
ASP.NET CORE SIGNALR
WITH VUE.JS

PARALLEL WORKFLOW
WITH THE .NET TPL

XAMARIN.FORMS 4
WHATS NEW?

INTRODUCTION TO
REACTIVE FORMS

DESIGNING DATA OBJECTS:
MORE EXAMPLES

UNIT TESTING
ANGULAR
DIRECTIVES
AND PIPES

WHAT WAS NEW
FOR .NET DEVELOPERS
IN **2018**

www.dotnetcurry.com/magazine/

THE TEAM

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

Art Director : Minal Agarwal

Contributing Authors :
Yacoub Massad
Ravi Kiran
Riccardo Terrell
Keerti Kotaru
Gerald Versluis
Daniel Jimenez Garcia
Damir Arh

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

Next Edition : Mar 2019

LETTER FROM THE EDITOR

Suprotim Agarwal
Editor in Chief

Hey Readers, Happy New Year 2019!

What an engaging year for the developer it was! New standards, platform changes, frontend frameworks and other tools and libraries had a lot to offer in 2018. WebAssembly made good progress with 3 major releases. Angular, React and Vue continued to dominate web development by actively releasing new features and versions. React released v16, Angular released v6 and v7, and TypeScript made good progress with v3. Watch out for React-JSS, JAMStack, WebPack, and my favourite, the upcoming Vue.js v3.

On the other hand, Microsoft was busy focusing on platform changes and new updates (Azure AI, .NET Core, .NET Standard, VS 2019, Open Source initiatives), acquisitions (Github), and partnerships (Amazon, Google, Linux etc). VS Code also became the most used source code editor in 2018 (yay to that)!

The DotNetCurry team was busy consuming these rapid changes and demystifying it for you. Our authors also worked tirelessly to promote patterns and practices and to help improve your coding skills. We will continue delivering fresh and cutting edge content in 2019 too! I also want to give a shout out to all those who are supporting a low-cost initiative by pre-ordering ***"The Absolutely Awesome Book on C# and .NET"****. Thank you Patrons!*

What topics and technologies excite you? What else do you want us to cover in this magazine? Reach out to me directly with your comments and feedback at suprotimagarwal@dotnetcurry.com or via my twitter handler @suprotimagarwal.

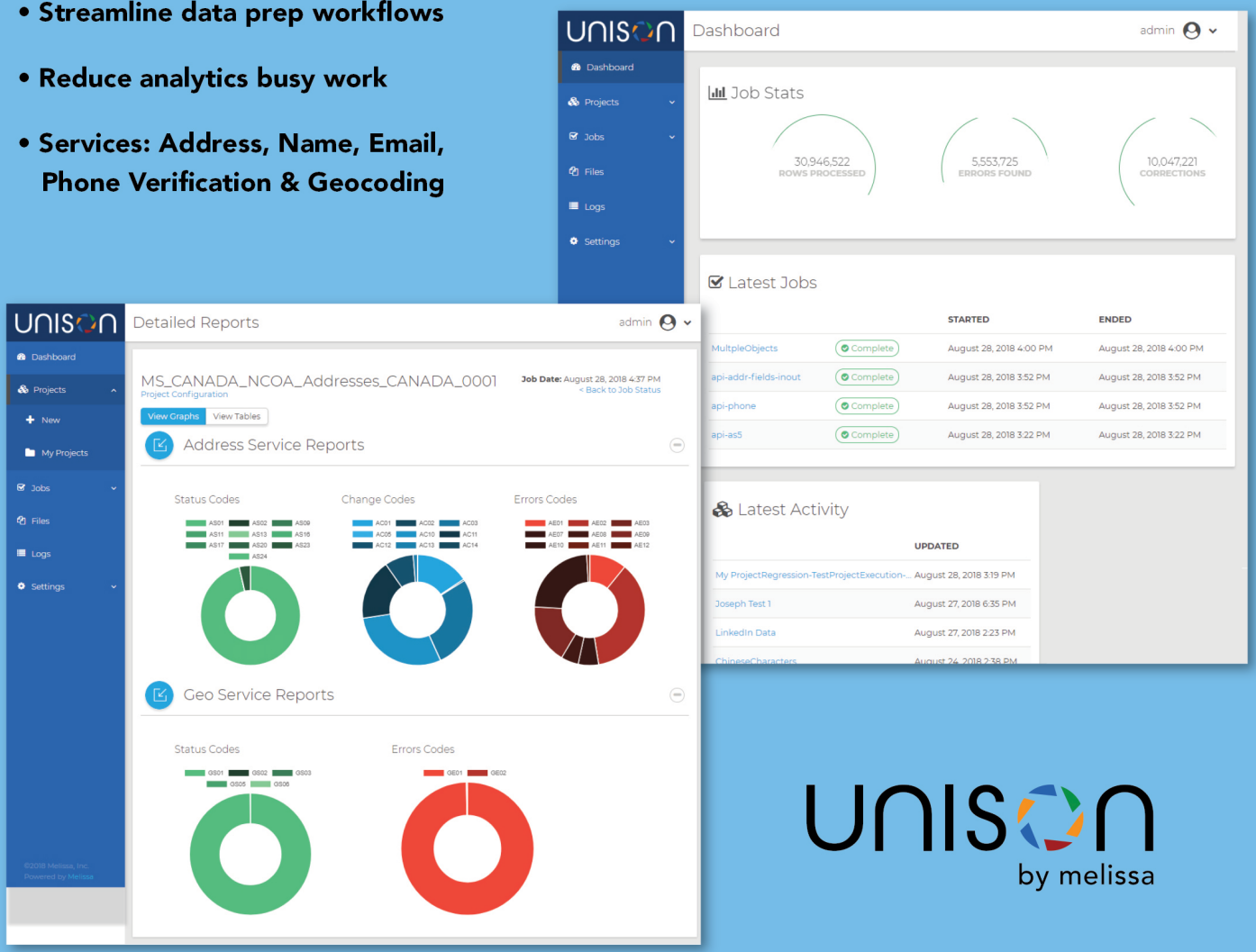
CONTENTS

Multiplatform Data Quality Management

Unison – Speedy, Secure, Scalable

Melissa's Unison is a data steward's best friend. Unison is a unique multiplatform solution that establishes and maintains contact data quality at lightning speeds – processing 30+ million addresses per hour – while meeting the most stringent security requirements. With Unison, you can design, administer and automate data quality routines that cleanse, validate and enrich even your most sensitive customer information – name, address, phone, email address – as data never leaves your organization.

- **Lightning fast processing (30M records/hour)**
- **Streamline data prep workflows**
- **Reduce analytics busy work**
- **Services: Address, Name, Email, Phone Verification & Geocoding**



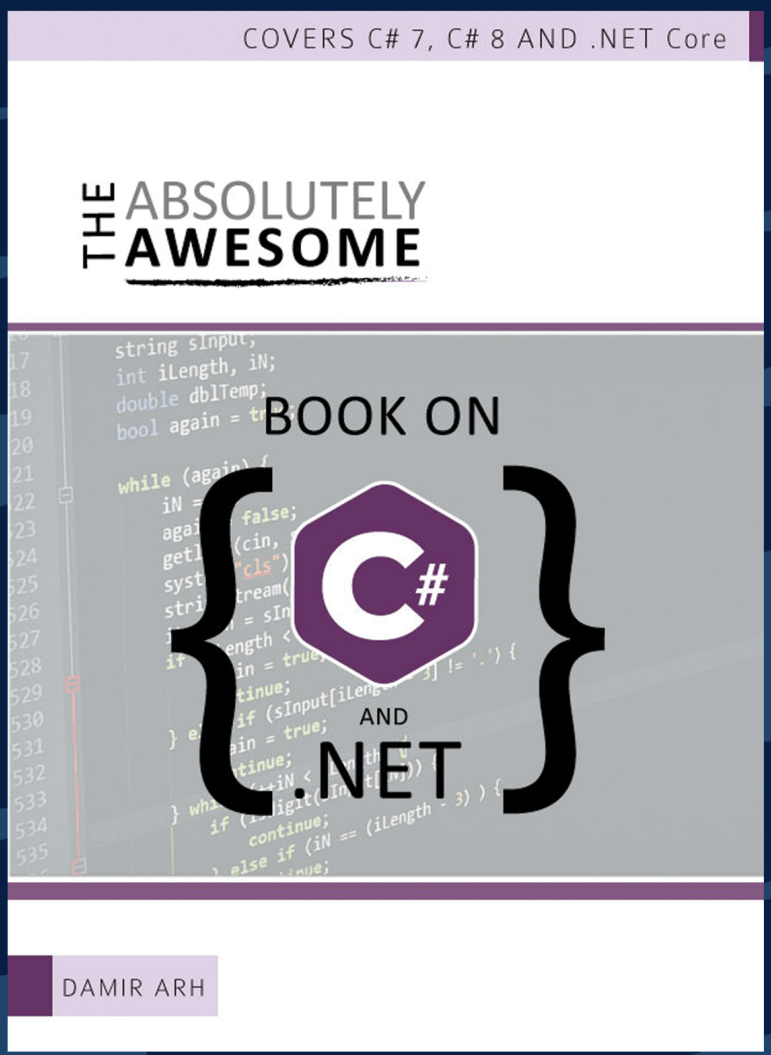
unison by melissa

Request a Demo

Melissa.com/unison
1-800-MELISSA

melissa

THERE'S A HOT NEW BOOK IN TOWN!



Features

- .NET Framework and CLR
- New features in .NET
- Type System
- Generics and Collections
- C# 6,7 and 8
- Parallel Programming
- Async Programming
- LINQ

It's got it all!

Crack your next .NET Interview

Build a Solid Foundation

Strengthen Concepts

THE ABSOLUTELY AWESOME BOOK ON

C# and .NET

ORDER NOW !

PDF, EPUB and MOBI

Damir Arh

WHAT WAS NEW FOR .NET DEVELOPERS IN 2018 & THE ROAD AHEAD

2018 was a busy year both for Microsoft, as well as for the developer community working on Microsoft & Open Source technologies. Many new important releases and announcements were made at different Microsoft events in 2018. This article covers the most important ones, as well as the other releases which didn't coincide with a specific event.

.NET CORE AND .NET STANDARD

During the year 2018, the development of .NET Core continued at full speed and several important releases were made available.

.NET CORE 2.1

In May 2018, .NET Core 2.1 was released. Its main focus was performance:

- Build times were drastically improved. Build time for a Large project got 10 times faster than before.
- A new memory efficient type `Span<T>` was introduced which provided an isolated view of a part of a larger array without any memory operations. It is very useful for efficient parsing and other processing of large inputs.
- `Span<T>` was used to implement a new `HttpClient` class for better networking performance.
- A lot of work has been done on the JIT (just-in-time) compiler to improve the runtime performance of applications.

New features were also introduced. The most important ones are:

- The Windows compatibility pack includes 20 thousand APIs from the .NET framework which are not included in .NET Core itself (e.g. `System.Drawing` namespace, support for Windows services, registry, EventLog etc.). It can primarily be used to make porting of existing .NET framework applications to .NET Core, easier. Despite its name, it's available not just for Windows but about half of the APIs are also implemented on other platforms. The rest throw an exception (as of this writing) when invoked on a non-Windows OS.

- .NET Core Tools represent a new (NuGet based) way for deploying command line (CLI) tools written in .NET Core. They are modelled after NPM (Node Package Manager) global tools. A tool can be installed using the following command:

```
dotnet tool install -g dotnet-serve
```

It will be added to the path and can later be invoked using only its name:

```
dotnet-serve
```

The `dotnet-serve` tool mentioned here is a simple HTTP server. A list of available tools is maintained on GitHub.

.NET CORE 2.2

Just before the end of the year, in December 2018, .NET Core 2.2 was released. It didn't have as many improvements as .NET Core 2.1. Most changes were introduced to ASP.NET Core, e.g.:

- Better support for Open API (Swagger) descriptors for HTTP (REST) services. Customizable conventions were added as a simpler alternative to individual documentation attributes. A diagnostic analyzer is

included for detecting mismatches between the code and the documentation attributes (you can read more about diagnostic analyzers in my DNC Magazine article "Diagnostic Analyzers in Visual Studio 2015").

- A new router with improved performance and a better globally available link generator.

Entity Framework Core was also expanded with support for spatial data.

Although .NET Core 2.2 was released after .NET Core 2.1 and includes several new features, **it's recommended to still use .NET Core 2.1** for most applications because it has a different support policy. .NET Core 2.1 was released as an LTS (Long Term Support) version which means that it will be supported for 3 years after its initial release (August 2018 when .NET Core 2.1.3 was released) or one year after the release of the next LTS version (whichever happens later).

In contrast, .NET Core 2.2 was released as a Current version. As such it will only be supported for 3 more months after the next version of .NET Core is released (either Current or LTS).

.NET CORE 3.0

Also, in December 2018, .NET Core 3.0 preview 1 was released. It's the first public preview of the next major version of .NET Core.

Its main new feature is the ability to create Windows desktop applications using Windows Forms or WPF (Windows Presentation Foundation). Both UI frameworks are extended with support for hosting UWP (Universal Windows Platform) XAML controls when targeting Windows 10 only. These controls provide access to modern browser and media player controls, as well as better touch support. .NET Core implementations of all three UI frameworks (Windows Forms, WPF and UWP) were open sourced.

The main use case for .NET Core Windows desktop applications is porting of existing .NET framework desktop applications, to .NET Core. To make it even easier, the latest version of Entity Framework 6 is being ported to .NET Core as well.

Although recompiling existing .NET framework applications for .NET Core won't make them cross-platform because the UI frameworks are only available on Windows, it will still allow them to take advantage of other .NET Core benefits:

Improved performance of .NET Core in comparison to the .NET framework.

- Support for application local deployment (in addition to global deployment) of the .NET Core version that the application uses. This will allow different applications to use different versions of .NET Core which is currently not possible with the .NET framework.

.NET Core 3.0 will also bring other new features:

- Further performance optimizations (improved `Span<T>` and `String` types, faster JSON library based on `Span<T>`, JIT compiler optimizations).

- Local installation of .NET tools which will make it easier to restore local development environment and use a specific version of a tool with a project.

Improvements based on new C# 8 features, such as asynchronous streams and default implementations of interface members (you can read more about new features of C# 8 in my DNC Magazine article "C# 8.0 – New Planned Features").

.NET STANDARD 2.1

According to the currently available information, the final release of .NET Core 3.0 will also be accompanied with the release of .NET Standard 2.1 (you can read more about .NET Standard in my DNC Magazine article ".NET Standard - Simplifying Cross Platform Development"). It is planned to include around 800 new APIs which were added to .NET Core 2.1 and later versions. The most important additions will be:

- `ValueTask` and `ValueTask<T>` for more efficient asynchronous code (you can read more about these types in my DNC Magazine article "C# 7 - What's New").
- `Span<T>` and new members in existing types which support it.
- Dynamic generation of code through reflection emit (it will not be supported on platforms which don't allow dynamic code generation).

.NET Standard 2.1 will be implemented by .NET Core 3.0 on release. It will also be supported by future releases of Xamarin, Mono and Unity. There are no plans to support .NET Standard 2.1 in future versions of the .NET framework.

VISUAL STUDIO

Visual Studio 2017 minor updates continued to be released in the year 2018. However, the current minor version (15.9) is the last minor update for Visual Studio 2017. From now on, only servicing updates will be released for it with bug fixes only and no new features.

New features will now target Visual Studio 2019. Its final release date has not been announced yet, but the first preview version is already available for download.

VISUAL STUDIO 2017

Four new minor updates for Visual Studio 2017 were released throughout the year (versions 15.6 - 15.9). Like the previous updates, they were mostly maintenance releases with a small set of new features.

Most importantly, support was added for new versions of other products which were released during that time: C# 7.3, .NET framework 4.7.2, .NET Core 2.1 and 2.2, and UWP (Universal Windows Platform which was part of the Windows 10 update).

Other important features were:

- Improved F# support for .NET Core.
- Built-in Continuous Delivery configuration for Azure DevOps.
- Support for Hyper-V based Android emulator.

- Improvements to Visual Studio Installer for better upgrade experience.

Performance improvements were included as well, most notably for solution loading and testing.

The released minor versions also incorporated many bug fixes. Therefore, it makes sense to regularly install the updates even if you don't need any of the new features. The notifications inside Visual Studio make sure you don't forget to do that, and the Visual Studio Installer makes the update process quick and simple.

VISUAL STUDIO 2019

At the Microsoft Connect() 2018 event in December, Visual Studio 2019 Preview 1 was released. This will be the first major new version of Visual Studio since the release cadence of minor versions increased with Visual Studio 2017. This means that Visual Studio 2019 is introducing several large new features which required more work and couldn't be safely added to Visual Studio 2017.

User interface redesign

As soon as you start Visual Studio 2019, you will notice the first major change. The ***Start Page*** from Visual Studio 2017 has been reworked into a completely new ***Start Window*** which opens before the main window of Visual Studio 2019. The only new feature of the Start Window in comparison to the Start Page is the option to open a new project from an existing source control repository.

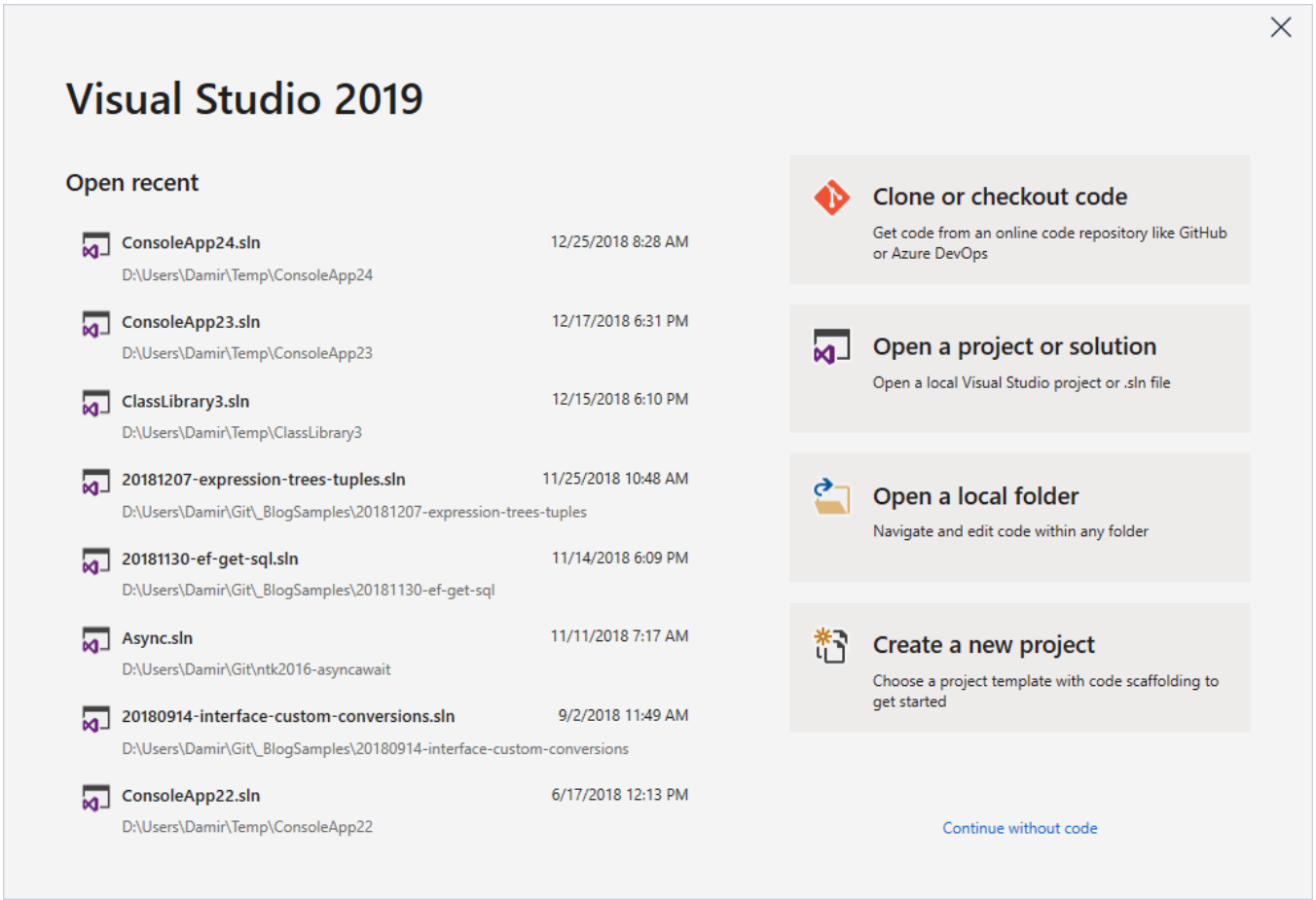


Figure 1: Visual Studio 2019 Start Window

The next change awaits you when you decide to create a new project. After many years, the wizard for creating a new project has been completely redesigned. The hierarchical view of templates has been replaced by a more flexible search as well as some filtering options.

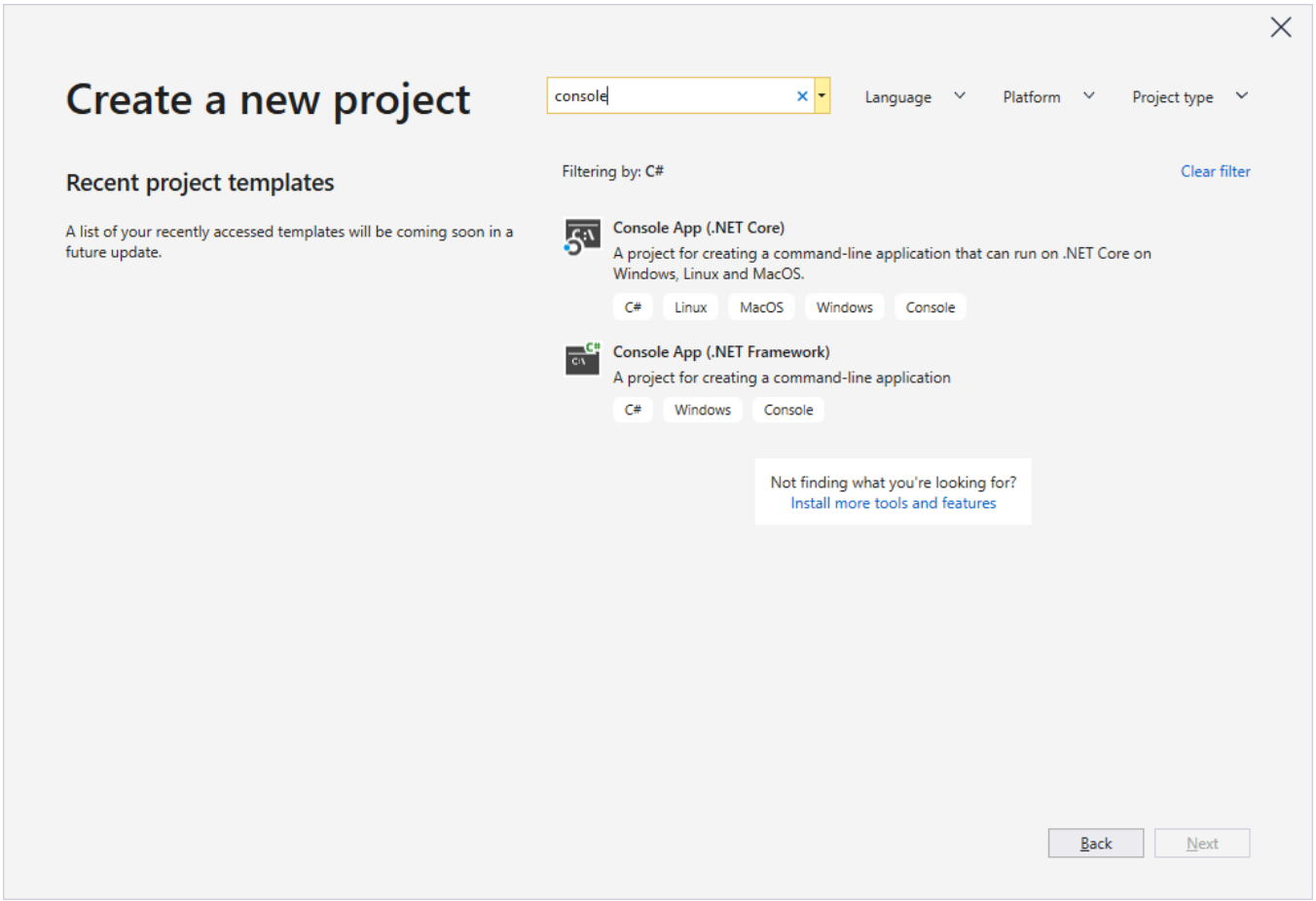


Figure 2: New project wizard in Visual Studio 2019

The final big user interface change will reveal itself when you reach the main window of Visual Studio 2019. The title bar at the top has been removed to leave more space for the code editor. The menu bar and toolbar have been reorganized accordingly to keep all existing functionality.

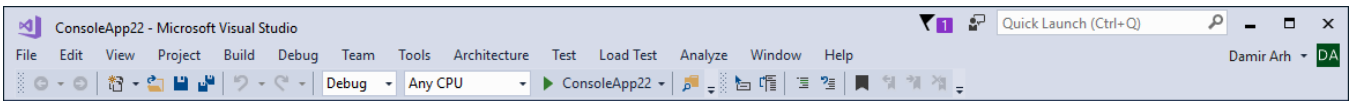


Figure 3: Title bar, menu bar and toolbar in Visual Studio 2017

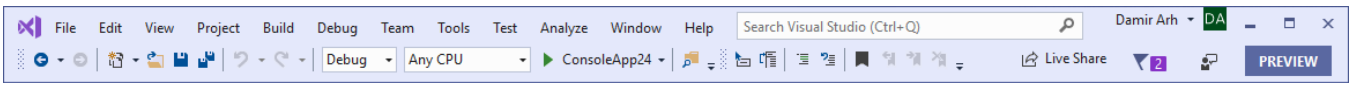


Figure 4: Menu bar and toolbar in Visual Studio 2019

The quick launch feature has also been improved to better handle typos and show more information about the displayed search results.

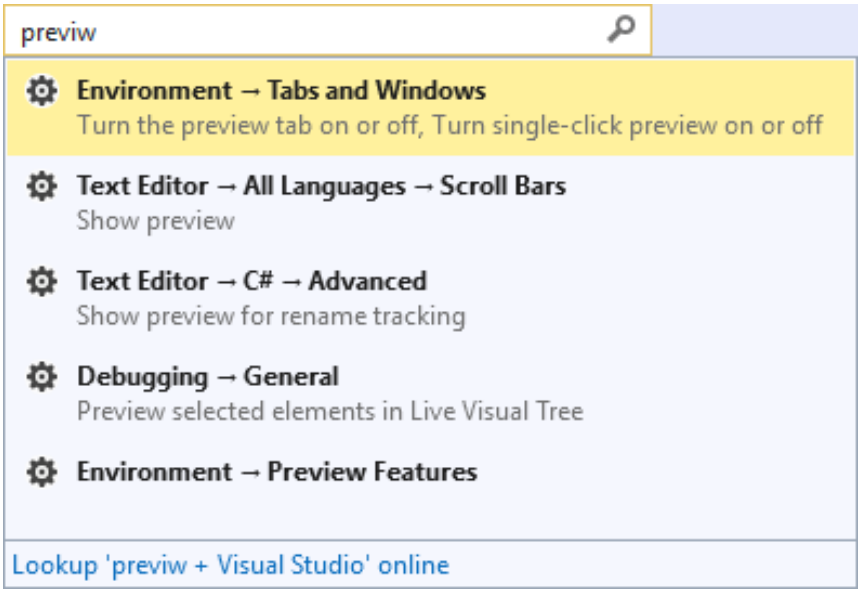


Figure 5: Improved quick launch feature in Visual Studio 2019

All these features are still experimental and subject to change based on the feedback received from the users. For some reason, if a feature doesn't work for you or you do not want it; but you would like to keep using Visual Studio 2019, there is now an option to opt-out of a specific experimental feature.

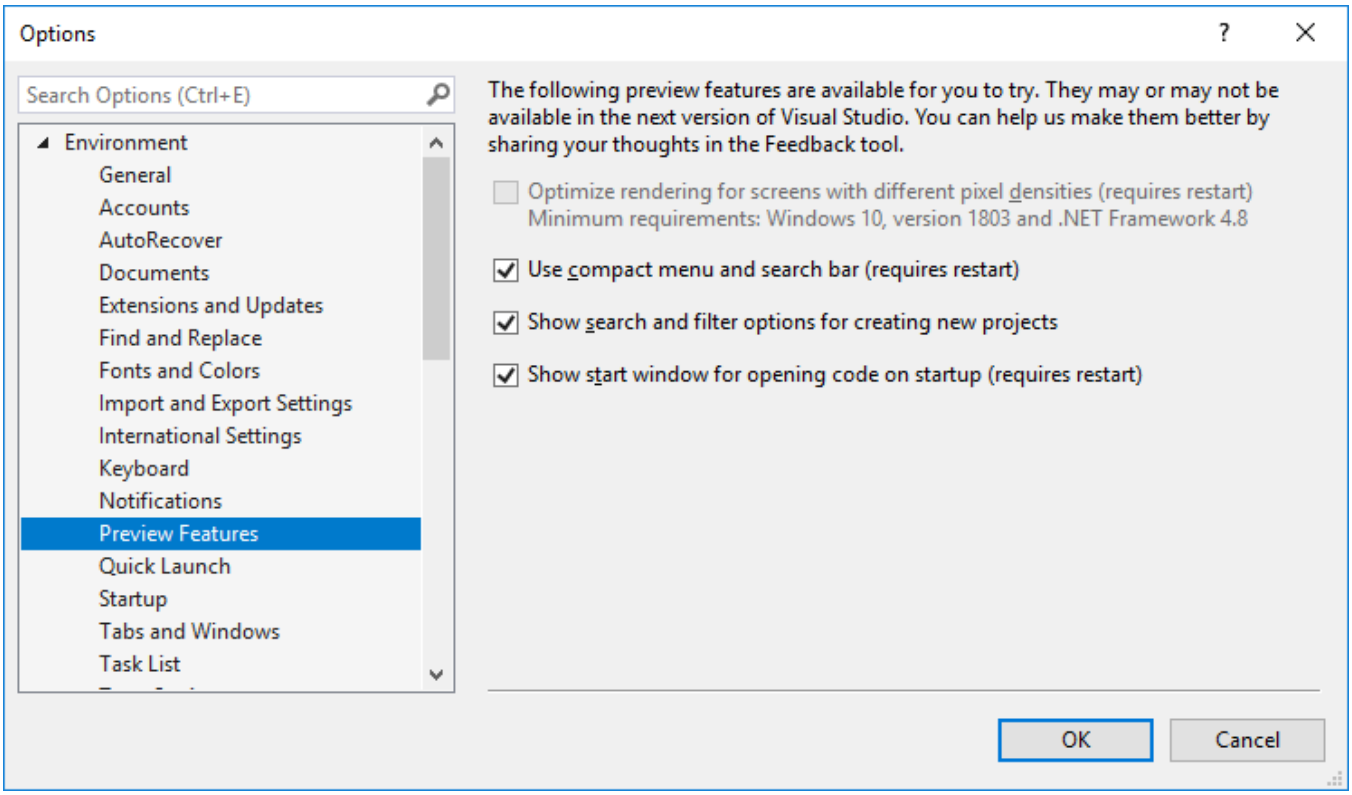


Figure 6: Opt-out options for experimental features

Productivity

There are also a lot of productivity improvements in Visual Studio 2019 which aren't immediately visible but should make you more efficient in all phases of the development process.

To speed up the initial **loading of large solutions**, you can now configure which projects do you want to load automatically and which you want to keep unloaded initially. You can also choose not to restore the

state of tool windows and the Solution Explorer tree view to further reduce the startup time.

To save you time spent on writing code, several **code fixes** were added for refactoring code, e.g. for converting a local function to a method, a tuple to a struct, an anonymous type to a class or struct, etc. You can now get a quick overview of all the errors, warnings and suggestions in each file by looking at the **document health indicator** in the bottom right corner of the code editor window. Its tooltip gives a more detailed overview and allows you to navigate between issues. From its context menu, you can configure or invoke code cleanup to automatically fix some of these issues.

If you're using **regular expressions** a lot, you'll appreciate the new built-in support for them in the code editor, such as syntax highlighting, brace matching and diagnostics. Regular expressions are automatically recognized when passed to the `Regex` constructor, but you can also tag them yourself using a special comment syntax.

During **debugging** you can take advantage of improvements in the Auto, Watch and Locals windows. The search functionality will help you find the correct variable by name or value stored in it. You can use format specifiers to format the values differently. For .NET Core projects, support for custom visualizers has been added.

A subset of **tests** can now also be run from the Solution Explorer context menu. **Code metrics** can finally be used with .NET Core projects. You don't need a third-party add-on for that anymore.

Team Explorer now supports Git stashes. The experience with Azure DevOps (formerly named Visual Studio Team Services or VSTS) work items has been completely redesigned. It's now focused around the developer workflow with built-in filters and automatic association of work items to local branches. For better experience with Azure DevOps pull requests, you can install an extension for reviewing, running and debugging them inside Visual Studio.

If you're curious about the new upcoming C# 8 features, you can try some of them out (nullable reference types, ranges and asynchronous stream, to be exact). For all of them to work, you will need to create a .NET Core 3 project and manually select the C# 8 beta language version in the Advanced Build Settings of Project Properties (you can find detailed instructions on changing the language version in my DNC Magazine article "C# 7.1, 7.2 and 7.3 - New Features (Updated)").

VISUAL STUDIO LIVE SHARE

At Build 2018 in May, a public preview of Visual Studio Live Share was released as an extension for Visual Studio 2017 and Visual Studio Code. The functionality is built into Visual Studio 2019 Preview 1.

Visual Studio Live Share was designed as an improved experience over screen sharing for remote collaboration between developers. Instead of both developers looking at the same screen with only one of them having control at the same time, each of them can now work on the same code using their screen.

The guest gets full access to the host's project without having to download the project code locally. As the host and the guest independently navigate the code, they both see the changes either one of them makes, in real time. The guest can also collaborate in the debugging process although the code is running on the host's computer.

The tool can be very useful for remote pair programming, code reviews and in educational scenarios.

VISUAL STUDIO INTELLICODE

Another feature introduced at Build 2018 was Visual Studio IntelliCode. This extension for Visual Studio 2017/2019 and Visual Studio Code helps the developer with AI assisted functionalities.

IntelliSense suggestions are improved by using machine learning on the publicly available code repositories. This allows the IntelliSense feature to suggest the most relevant API and overload based on the surrounding code. The supported languages depend on the IDE chosen:

- In Visual Studio 2017 and 2019 C#, XAML and C++ are supported.
- In Visual Studio Code TypeScript/JavaScript, Python and Java are supported.

For C#, you can even train your own model on private code repositories.

Visual Studio 2017/2019 can also infer the formatting and code style conventions from existing code in the solution. It can generate an .editorconfig file for you which can be used to enforce conventions from then on.

Since the original release of the extension in May 2018, new features have been constantly added to it. It is very likely that additional features will be added in the future.

Conclusion:

With the announcement of support for Windows desktop applications in .NET Core 3.0, it became clear that .NET Core is replacing .NET framework as Microsoft's main development platform. Like it or not, .NET framework is now a legacy product. The upcoming version 4.8 is only a maintenance release with no major new features. For all new projects, .NET Core should be used unless the scenario is not (yet) supported by it.

The development of Visual Studio continues steadily. The announcement of Visual Studio 2019 means that after two years of minor updates with only a minimum number of new features, we can again expect some larger changes in 2019. You can take some of these features for a spin in the recently released preview.

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 Suprotim Agarwal for reviewing this article.

DocuVieware

Universal HTML5 and Document Management Kit

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
60-day Free Trial Support Included at www.docuvieware.com

Powered by **GdPicture**.NET

DocuVieware is an ORPALIS imaging technologies product

Gerald Versluis

XAMARIN.FORMS 4 – WHAT IS NEW?

If you have read my previous article(s) right here in the DotNetCurry(DNC) magazine, then you might have read one about the cool new things in Xamarin.Forms version 3. Xamarin.Forms v3 was released in May of 2018. At the time of this writing, it is just December and the next major version is already upon us.

One thing is for sure: the release cycle is going fast, but what are the new features in store for us? In this article, I will show you all the latest bits!

WHAT IS XAMARIN.FORMS?

Let's very briefly touch on what Xamarin.Forms is.

With Xamarin, you can build cross-platform apps based on .NET and C#. No more Java and Objective-C or Swift. Everything you can do with these languages and platforms, can be done with Xamarin in C#.

This is possible because of the Mono project.

If you have been around long enough, you might remember Mono. It was the Linux variant of the .NET framework. Basically, what the Mono project did was implement each new .NET API or feature in their own Mono framework, making it the first fully featured, cross-platform .NET solution. The people who have worked on Mono are mostly also the people that founded Xamarin.

The solutions Xamarin offers you is based on binding libraries that project the native APIs onto their .NET equivalents. This allows you to write cross-platform apps based on technology that you already know and love. The remaining problem that was left when Xamarin was introduced is that the user interface still needed to be implemented in a native way.

For Android, this means with AXML files which contain some kind of XML dialect, and for iOS you have to use Storyboards, which are also XML files under the hood. To overcome this last hurdle in true cross-platform development, **Xamarin.Forms** was created.

With Xamarin.Forms, you can define your UI through C# code or XAML in an abstract way. For instance, you define a Button object and the Xamarin.Forms libraries will translate this to their native equivalent. This way, you can develop your interfaces cross platform, but that same interface will still look native on each platform.

If you still need more information on what Xamarin.Forms is exactly and how it works internally, I would recommend to read up in the Microsoft Docs: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/.

The Forms libraries have been around for quite some time now and developers began to love or hate it. Either way, Xamarin.Forms is here to stay and Microsoft will release the stable version of this new major version over the coming months.

NEW IN XAMARIN.FORMS 4

Of course, this new major version will contain a lot of stability fixes, bugfixes and minor improvements across the board, but version 4 will also have some real big changes. Let me walk you through the most exciting ones.

Everything described here is in preview at the time of writing and will require you to download preview NuGet packages or even preview versions of tooling and templates. When Xamarin.Forms 4 is generally available, using it will be a lot easier. If you want to play around with this while still in preview, make sure to check the prerequisites.

XAMARIN.FORMS SHELL

To make it easier for newcomers to get started, Shell was invented. Basically, what Shell does is provide you with a whole lot of bootstrapping so you can quickly jump in and start focusing on the content and features.

Shell offers you out of the box navigation with flyout menus, bottom tabs and top tabs. And with the URL based navigation, it makes it very easy to deep link to anywhere in your app. With Shell you can define the structure and navigation of your app in a single file. Microsoft described it as “the evolution of MasterDetailPage, NavigationPage and TabbedPage”.

Everything is template-based so everything will look great from scratch, but you still have the ability to customize all the elements to your needs. All of this is based on existing Xamarin.Forms technology, so you don’t have to learn any new skills or tooling, just install the new templates and you can get going.

A sample Shell page might then look like this in XAML:

```
<Shell xmlns="http://xamarin.com/schemas/2014/forms"
   xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
   xmlns:local="clr-namespace:MyStore"
   FlyoutBehavior="Disabled"
   x:Class="MyStore.Shell">

  <ShellItem>

    <ShellSection Title="Home" Icon="home.png">
      <ShellContent>
        <local:HomePage />
      </ShellContent>
    </ShellSection>

    <ShellSection Title="Notifications" Icon="bell.png">
      <ShellContent>
        <local:NotificationsPage />
      </ShellContent>
    </ShellSection>

  </ShellItem>
</Shell>
```

Depending on if something is a `ShellItem` or `ShellSection` for instance, it will determine if something is a top-level section or a section that lies deeper, as well as if it will show up in the flyout menu or in the top or bottom tabs. It is a bit hard to explain in text, but once you get to play with it, it will soon become clear.

An example of what a Shell page with some items might look like, can be seen in the iOS screenshot here.

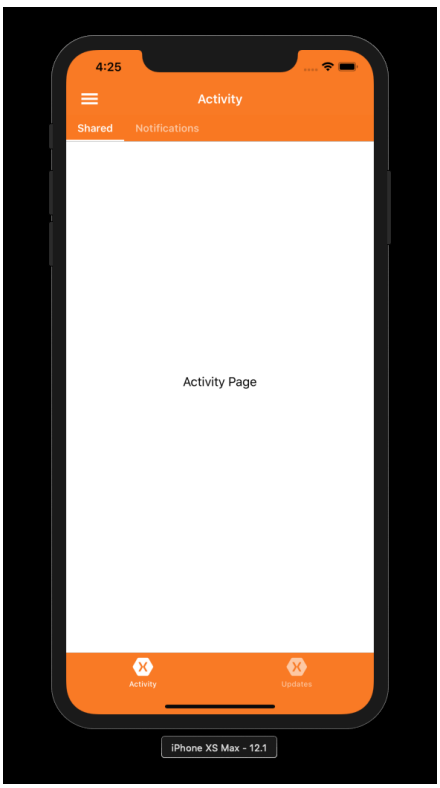


Figure 1: iOS screenshot with a Shell page containing items

At the time of this writing you will have to enable a feature flag in your project to get started with Shell. Also, don't forget to download the (pre-release) Xamarin.Forms 4 NuGet package into your project. To enable the feature flag, go to the *App.xaml.cs* file and make sure to add the lines from the code block below.

```
public partial class App : Application
{
  public App()
  {
    InitializeComponent();
    // Set the feature flag
    Forms.SetFlags("Shell_Experimental");
    // Instantiate new shell object AFTER setting the feature flag
    MainPage = new Shell();
  }
}
```

All of this should help you increase your productivity while decreasing complexity of the app. On top of that, Shell has been developed with rendering speed and memory footprint in mind, so it should stay clear of much heard complaints when using Xamarin.Forms.

To read about all the details of Shell, head over to the Microsoft Docs: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/shell.

VISUAL

As I mentioned in the introduction of this article, Xamarin is focused on creating a native look-and-feel. On the other hand, you see a lot of apps and developers of apps shift towards a more consistent look across platforms. Solutions like React Native and Flutter are leaning more towards drawing their own controls and this way, achieving the exact same look on iOS as well as Android.

With ***Visual*** Xamarin is now trying to do more or less the same thing.

You can think of *Visual* as a templating engine. When enabling *Visual*, you can specify what theme you would like your app to show. Right now, only *Material* is implemented which is based on the Google Material Design styling. In the future, other design languages including Fluent, Fabric and Cupertino will be supported.

What happens when you enable Visual is that the Forms libraries will then use different renderers to draw the controls. So, depending on which theme you will select in the future, you will just get alternate renderers that will help you achieve consistency across platforms. Not just in looks, but also in user-experience.

However, the controls are still rendered to native controls so minor differences exist mostly in terms of font, shadows, colors and elevation. In the image here, you can see a screenshot of iOS and Android side-by-side with the Material Visual enabled.

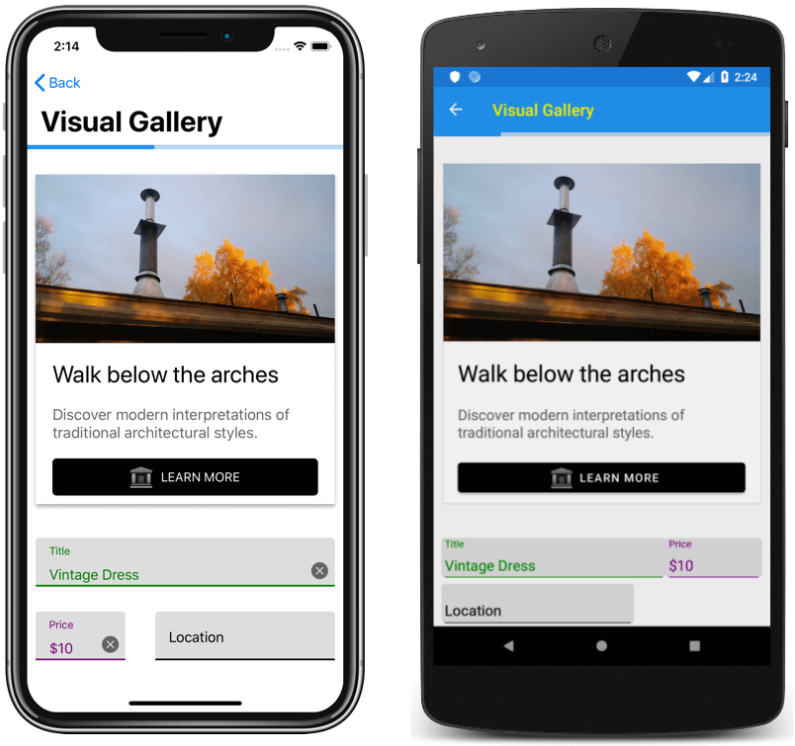


Figure 2: iOS and Android side-by-side with the Material Visual enabled

To enable Visual today, download the Forms 4 NuGet package into your projects and enable the feature flag. For this, add this line in your Android and/or iOS project in the *MainActivity* or *AppDelegate* of the respective projects: `Forms.SetFlags("CollectionView_Experimental");`

COLLECTIONVIEW CONTROL

If you have worked with Xamarin.Forms before, then you will know the `ListView` control for sure. You can hardly create any app without using it, but it can also be a bit of a pain to work with. When not implemented the right way, you can easily create performance issues with the `ListView`. In addition, some often requested features are not, and cannot be implemented on the current control.

There have been several attempts to overcome this, or at least provide some alternatives. One control that has also been wrestled with, is the *carousel*. This has been a separate page, but also a control and is incorporated in the standard toolbox of Xamarin.Forms right now. But the development on this control isn't very active, so its features are limited.

You might also know about the *FlexLayout*, which was introduced in the last major Forms update described in a previous article.

Therefore, a new control had to be created which does basically the same thing as the `ListView`, only better. Because of that, the team at Xamarin has now implemented the `CollectionView`.

Things that are easily possible with the `CollectionView`, but not with the `ListView` for instance are:

- **Horizontal list:** you can easily create a list that scrolls horizontal instead of just vertical. You can even implement your own layout if you wish.

- **Columns:** specify columns in your list. Have items shown side-by-side instead of only on top of each other. You can see an example of this in Figure 3.

- **Granular scrollbar progress:** with the `ListView`, you would just know if the user had scrolled yes or no. With the `CollectionView`, it is also possible to interact whenever the user is actually scrolling. It seems like a minor thing, but when you want to do some advanced UX and animations, this is a must-have.

- **Empty state:** you can specify a `DataTemplate` to show for whenever your collection is empty. No more swapping visibility of views to tell the user that there is nothing to show, it is built-in, right into this new control!

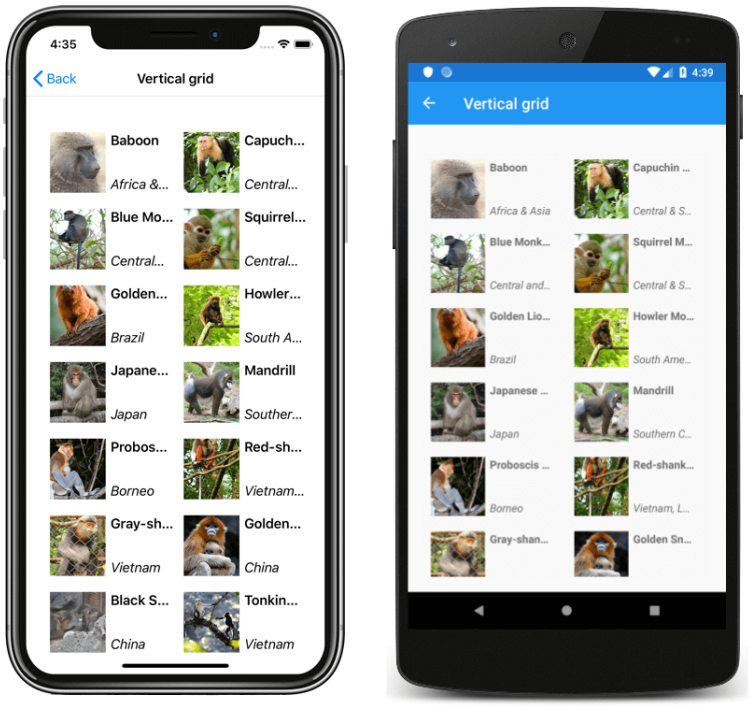


Figure 3: The new CollectionView control with two columns

The XAML for the image in Figure 3 is very simple and straight-forward. You can see it here.

```
<CollectionView ItemsSource=”{Binding Monkeys}”>
 <CollectionView.ItemsLayout>
  <GridItemsLayout Orientation=”Vertical” Span=”2” />
 </CollectionView.ItemsLayout>
 <CollectionView.ItemTemplate>
  <DataTemplate>
   <Grid Padding=”10”>
    <Grid.RowDefinitions>
     <RowDefinition Height=”35” />
     <RowDefinition Height=”35” />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
     <ColumnDefinition Width=”70” />
     <ColumnDefinition Width=”80” />
    </Grid.ColumnDefinitions>
    <Image Grid.RowSpan=”2” Source=”{Binding ImageUrl}” Aspect=”AspectFill”
    HeightRequest=”60” WidthRequest=”60” />
```

```
    <Label Grid.Column="1" Text="{Binding Name}" FontAttributes="Bold"
    LineBreakMode="TailTruncation" />
    <Label Grid.Row="1" Grid.Column="1" Text="{Binding Location}"
    LineBreakMode="TailTruncation" FontAttributes="Italic"
    VerticalOptions="End" />
   </Grid>
  </DataTemplate>
 </CollectionView.ItemTemplate>
</CollectionView>
```

If you have worked with Xamarin.Forms and the `ListView`, you might see some similarities, but also differences. You can still specify the `ItemSource` and work with `ItemTemplates` but the template will not be a *Cell* anymore.

A lot of this is possible since the `CollectionView` does not work with Cells anymore. Instead, it works with `DataTemplates`, that you might already know from the `ListView`, but then in terms of how you would like each `Cell` to look like.

There are a lot of nice things coming to the `CollectionView,` but today not all of it is available yet. Also, there might be changes in the API down the road so take that into account when choosing to start developing with this right away.

To read up on this promising control, have a look at the corresponding Microsoft Docs page: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/collectionview.

OTHER IMPROVEMENTS

A lot of improvements bigger or smaller are included in this release as well. There has been a great community effort to fix a whole lot of bugs or add small features that have been annoying developers for a long time now. Think of things like a read-only `Entry`, an `ImageButton` or the ability to set a maximum amount of lines for your `Label`.

These changes are not specific to this new 4.0 version, but are mostly included in the releases that are stable today, still they are worth mentioning since there is a big, active and very nice community behind Xamarin.Forms and the team at Microsoft is really friendly and supportive. If you ever thought to yourself: why isn't this included or why isn't this working, consider contributing it yourself and doing something back for the framework you love.

Another notable issue that the team at Microsoft has been working on, and *is* still working on **is the performance on Android**. It has already improved a lot, and they are working on making it even better.

YOU CAN USE ALL OF THIS, RIGHT NOW!

As I mentioned a couple of times before, you will need the previewing package of Xamarin.Forms 4 at the time of writing and also set a feature flag in your project manually. If you are reading this by the time that this new version is generally available, then it should just work by installing the right Forms package.

Editorial Note: *Readers are requested to check the current status of Xamarin Forms by looking up https://blog.xamarin.com or checking the Github page.*

If you need to install the preview packages, simply find the "show prerelease versions" checkbox in the screen where you find and add the NuGet packages, the ones tagged 4.0.0-beta will pop up then.

It goes without saying that there might be some bugs and instabilities here and there, but it is cool to get a peek inside the new upcoming version of this great framework. Xamarin.Forms is open-source these days, so you should be able to see some of the work on the repository as well or maybe even contribute yourself: https://github.com/xamarin/Xamarin.Forms/

SUMMARY

In this article you have seen the most important highlights of what is to come in Xamarin.Forms 4. There are a few resources that might take you a step further. Have a look at this blog post for example: https://blog.xamarin.com/xamarin-forms-4-0-preview/. This will refer you to some great sample apps that show off these new goodies.

I'm not sure if there is a target date for this to become generally available, but I expect the wait to be short. The introduction of Xamarin.Forms 4 has been done at the Microsoft Connect(); event, where a lot more news around Xamarin was announced. You can read more on that in this blog post here: https://blog.xamarin.com/connect-2018-xamarin-announcements/.

You might also want to look into the new Visual Studio 2019 for Mac that is in preview right now and will align the experience even more with the Windows Visual Studio variant. I have written a short blog post with my first impressions here: https://blog.verslu.is/tools/visual-studio/visual-studio-mac-2019-preview-glance/.

I am very much looking forward to this new version of Xamarin.Forms and all the new stuff that we get to work with. Especially the `CollectionView` allows for a lot greater UI tweaks that were not possible earlier. I am looking forward to what you will create with it, if you have anything to show, please do let me know!

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

Thanks to Mayur Tendulkar for reviewing this article.

Daniel Jimenez Garcia

USING ASP.NET CORE SIGNALR WITH VUE.JS

TO CREATE A MINI **STACKOVERFLOW.COM RIP-OFF**

ASP.NET Core 2.1 included the first official release of the redesigned SignalR. It's been completely rewritten for ASP.NET Core, although it provides a model similar to the one in the previous version of SignalR.

This article introduces the main concepts and building blocks of SignalR by implementing a minimalistic version of StackOverflow.com using an ASP.NET Core backend and a Vue.js frontend. A toy version compared to the full site, this example will be enough to explore the real-time functionality provided by SignalR.

We will also explore how to integrate its JavaScript client with the popular Vue frontend framework, as we add bi-directional communication between the client and the server through the SignalR connection.

I hope you will enjoy this article and find it a useful introduction to SignalR and a practical example on how to integrate it with one of the fastest growing frontend frameworks. *The companion source code for the article can be found on GitHub.*

1. SETTING UP THE PROJECT

Before we can start introducing the functionality provided by SignalR, we need a project that we can use as an example. In this section, we will create a new project that provides a minimalistic version of StackOverflow.com. It will be a fairly basic site that simply allows creating questions, voting on them and adding answers to the questions. However, this will be enough to later introduce some real-time features using SignalR.

Let's start by creating a new folder called **so-signalr** (or any other name you like). During the next sections, we will work through the frontend and backend projects of our mini Stack Overflow site.

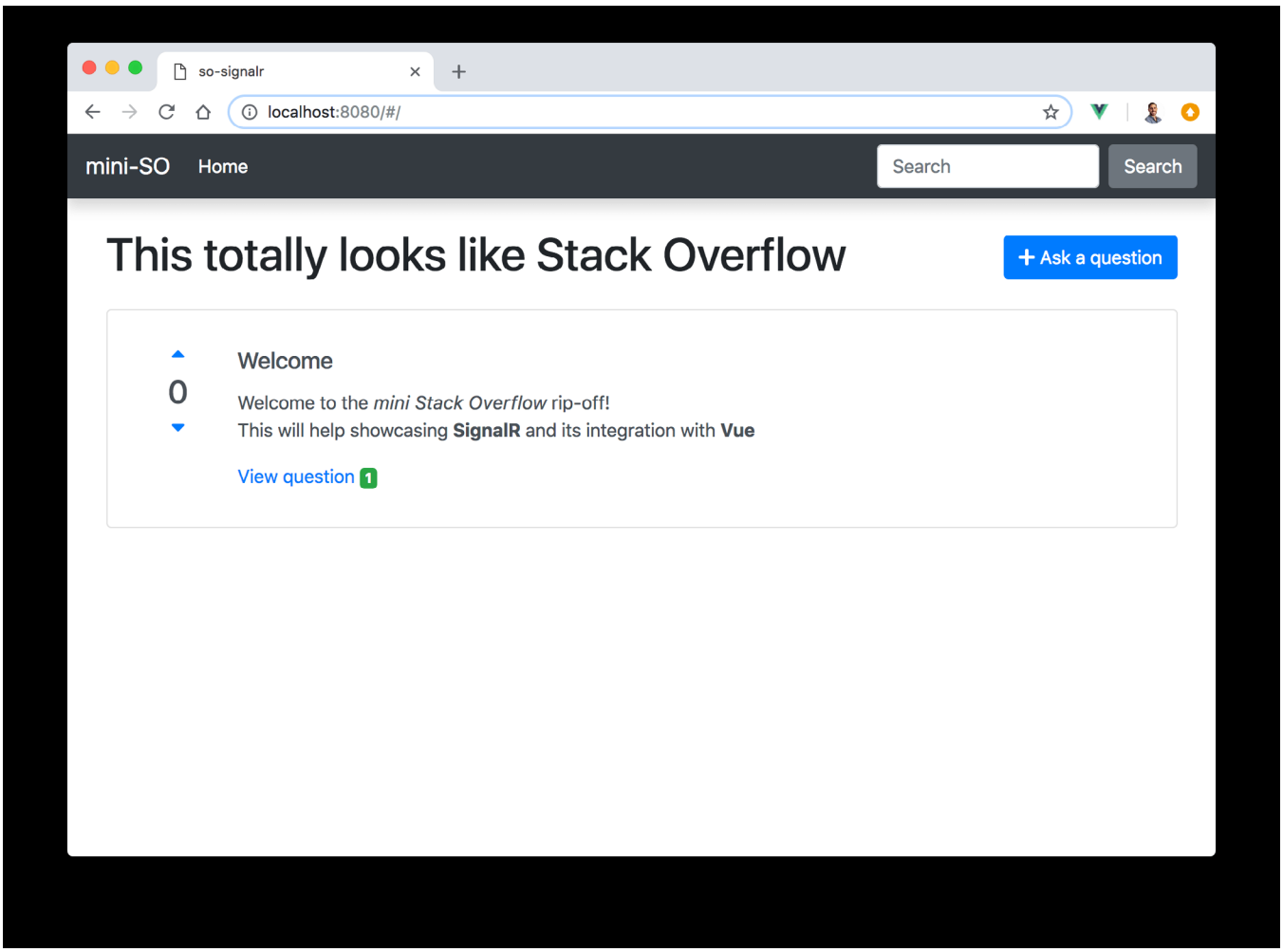


Figure 1, The Stack Overflow killer once we are done

It is worth mentioning that the main purpose of the article is to explore SignalR, and not so much how to build an API with ASP.NET Core, or a frontend with Vue. I won't get into much detail when building the example application. If you are unfamiliar with any of these or have trouble following along, I suggest you to check their official documentation sites and the previous articles published in this magazine, then come back to this article.

If you are already familiar, jump into section 2 and download the branch **starting-point** from GitHub.

Disclaimer. I prefer to think in terms of events and listeners while the official SignalR documentation thinks in terms of RPC calls and methods. Through this article, you will keep reading about events and listeners, but you can make the mental translation into RPC calls and methods if you prefer so.

CREATING THE BACKEND

Let's begin by making sure you have version 2.2 of ASP.NET Core installed by running `dotnet --version`. If you have an older version, download the latest SDK from the official site.

Next, open a terminal inside the so-signalr folder and run the following command to initialize a new ASP.NET Core project which will provide the REST API for our site:

```
dotnet new webapi –n server
```

You should now have a new folder **so-signalr/server** containing the ASP.NET Core project with our backend.

All we have to do now is to replace the default example API with one that we can use for the questions and answers (Q&A) of a site like Stack Overflow. Create a new folder named **Models**, and inside add two new files named **Question.cs** and **Answer.cs**. These will be simple POCO classes defining the base entities of our site:

```
public class Question {
  public Guid Id { get; set; }
  public string Title { get; set; }
  public string Body { get; set; }
  public int Score { get; set; }
  public List<Answer> Answers { get;set; }
}

public class Answer {
  public Guid Id { get; set; }
  public Guid QuestionId { get; set; }
  public string Body { get; set; }
}
```

These are intentionally simple. As mentioned before, all we need is a simple site inspired by Stack Overflow where we use some of the features provided by SignalR.

Next, replace the example controller with a new one that allows adding new questions and answers, as well as voting on them. A simple in-memory storage will be enough for the purposes of this article, but feel free to replace this with a different storage approach if you are so inclined to.

```
[Route("api/[controller]")]
[ApiController]
public class QuestionController : ControllerBase {
  private static ConcurrentBag<Question> questions = new ConcurrentBag<Question> {
   new Question {
    Id = Guid.Parse("b00c58c0-df00-49ac-ae85-0a135f75e01b"),
    Title = "Welcome",
    Body = "Welcome to the _mini Stack Overflow_ rip-off!\nThis will help showcasing
    **SignalR** and its integration with **Vue**",
    Answers = new List<Answer>{ new Answer { Body = "Sample answer" }}
   }
  };
  [HttpGet()]
```

```
  public IEnumerable GetQuestions()
  {
    return questions. Select(q => new {
     Id = q.Id,
     Title = q.Title,
     Body = q.Body,
     Score = q.Score,
     AnswerCount = q.Answers.Count
   });
  }
  [HttpGet("{id}")]
  public ActionResult GetQuestion(Guid id)
  {
   var question = questions.SingleOrDefault(t => t.Id == id);
   if (question == null) return NotFound();

   return new JsonResult(question);
  }
  [HttpPost()]
  public Question AddQuestion([FromBody]Question question)
  {
    question.Id = Guid.NewGuid();
    question.Answers = new List<Answer>();
    questions.Add(question);
    return question;
  }
  [HttpPost("{id}/answer")]
  public ActionResult AddAnswerAsync(Guid id, [FromBody]Answer answer)
  {
    var question = questions.SingleOrDefault(t => t.Id == id);
    if (question == null) return NotFound();

    answer.Id = Guid.NewGuid();
    answer.QuestionId = id;
    question.Answers.Add(answer);
    return new JsonResult(answer);
  }
  [HttpPatch("{id}/upvote")]
  public ActionResult UpvoteQuestionAsync(Guid id)
  {
    var question = questions.SingleOrDefault(t => t.Id == id);
    if (question == null) return NotFound();

    // Warning, this increment isnt thread-safe! Use Interlocked methods
    question.Score++;
    return new JsonResult(question);
  }
}
```

This is a fairly straightforward controller managing a list of questions kept in memory, where every question has a list of answers and a score. The API allows questions to be voted (a similar endpoint for down voting can be easily implemented. If you want, you can check out the source in GitHub) as well as adding new questions and answers.

Finally, let's enable CORS, since our frontend will be a Vue application served independently from our backend API. To do so, we need to update the Startup class. Add the following line to the `ConfigureServices` method:

```
services.AddCors();
```

Finally add the following line to the Configure method, before the `UseMvc` call, so we allow any CORS request coming from our frontend, with a fairly broad set of permissions. In a real application, you might want to be more restrictive with the methods and headers you allow. It also hardcodes the address of the frontend, which you might want to store as part of your configuration:

```
app.UseCors(builder =>
  builder
    .WithOrigins("http://localhost:8080")
    .AllowAnyMethod()
    .AllowAnyHeader()
    .AllowCredentials()
);
```

That's it, we have a REST API that can be targeted from our frontend and provides the functionality we need to disrupt the Q&A market. You should be able to start the project with `dotnet run`, which should start your backend listening at http://localhost:5000 (where **5000** is the default port unless you change it in **Program.cs**) and finally navigate to http://localhost:5000/api/question. You should see a JSON file with the list of questions, containing the one initialized in our in-memory collection.

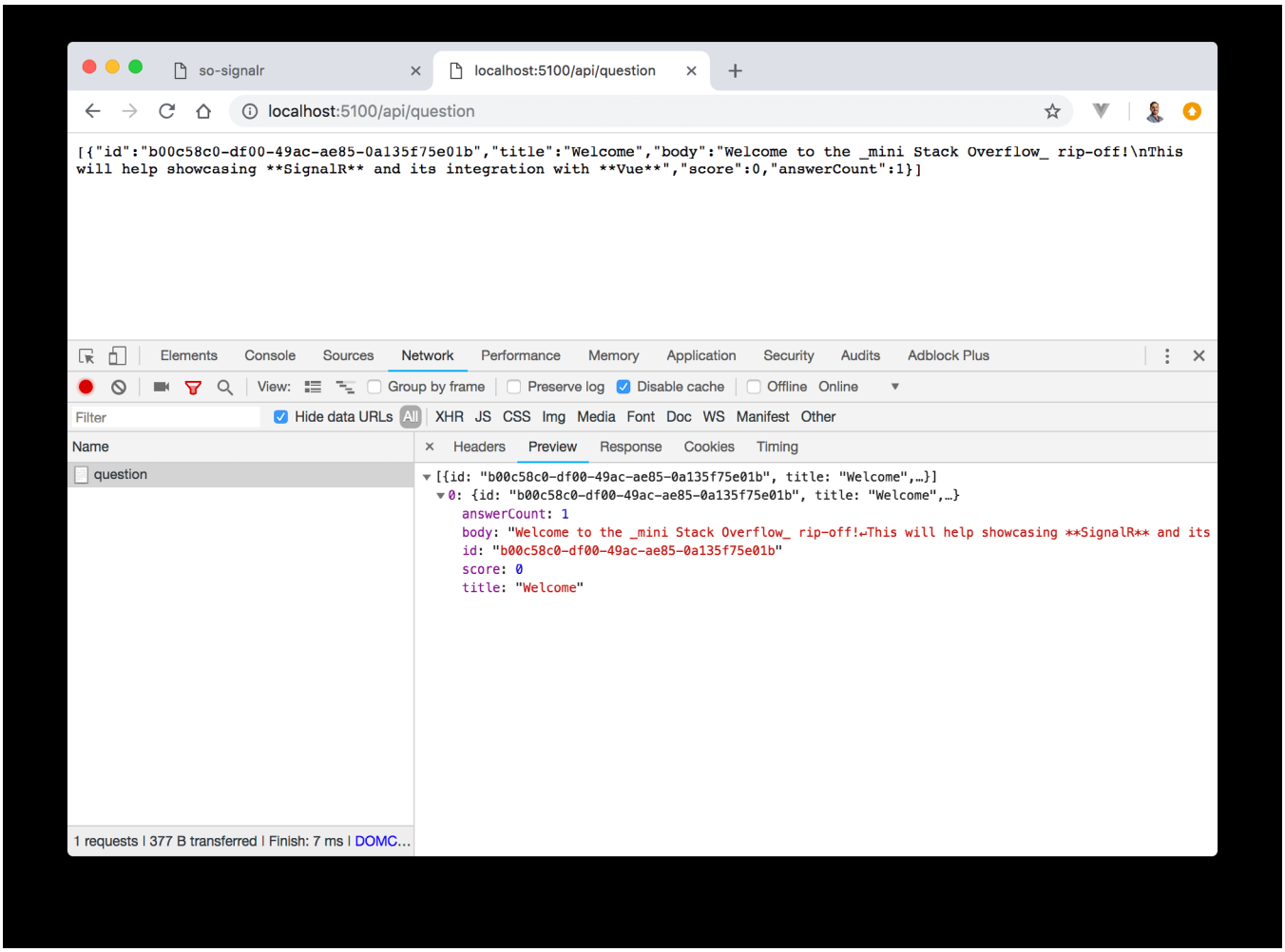


Figure 2, testing the API

It is now time to turn our attention to the frontend.

CREATING THE FRONTEND

Our next task is to create a Vue project that provides the user interface of our minimalistic Stack Overflow

site, and uses the backend we just finished in the earlier section. We will use the Vue CLI 3, which you can install as a global npm package.

Open a terminal inside the root folder and initialize a new Vue project with the command:
vue create client

When prompted to pick a preset, select "Manually select features". Leave Babel and Linter selected, then select Router option with the space bar before pressing enter. Check the screenshot for the remaining options, but feel free to use your preferred ones as long as you add the router.

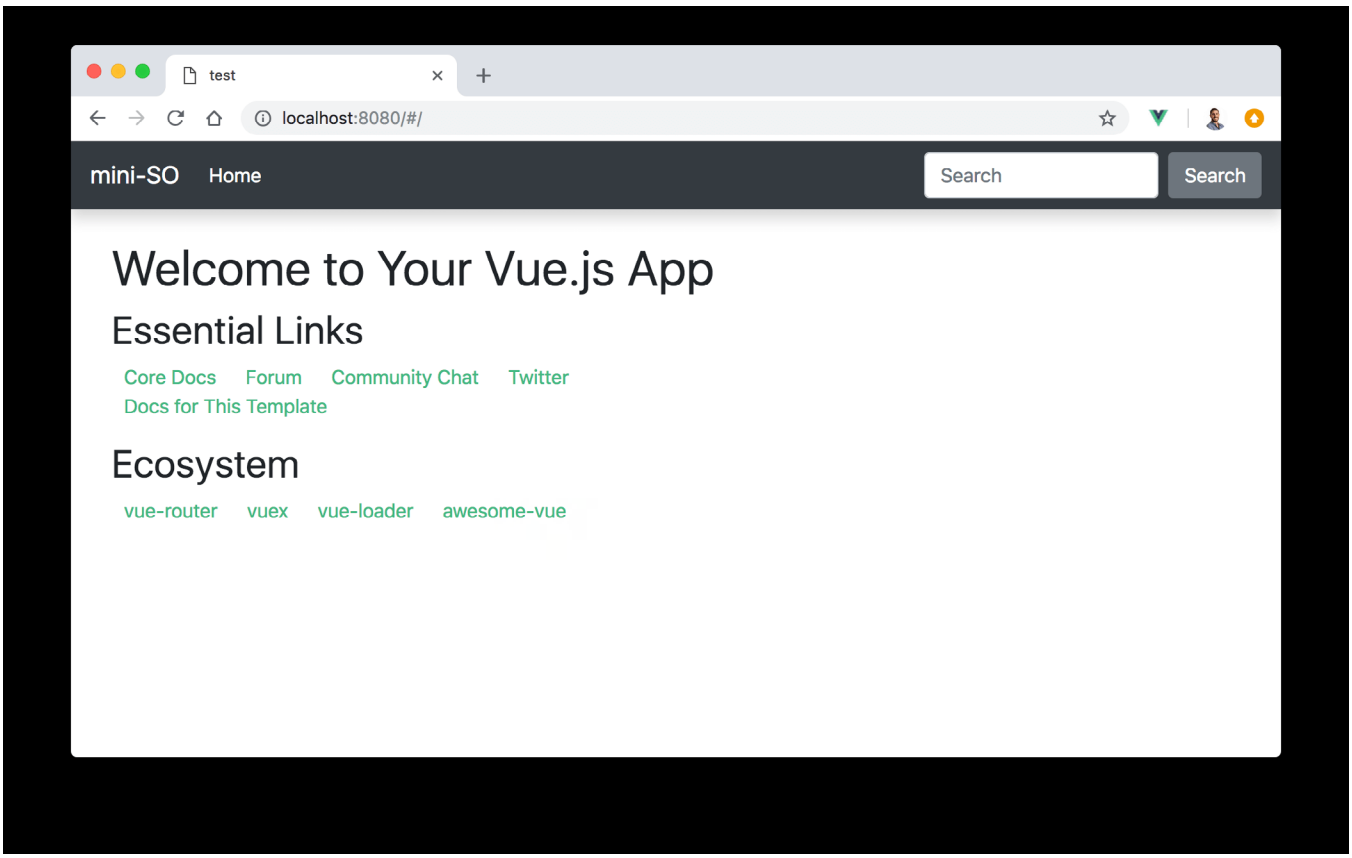


Figure 3, creating the client project with the Vue CLI 3

If you have version 2 of the cli, you should be able to follow along, use `vue init` to create the project and `npm run dev` to run it.

We will complete the setup by installing a few extra libraries that will help us build a not so barebones frontend:

- Axios provides the HTTP client we will use to send requests to our backend
- Bootstrap-vue and font-awesome provide the styling and UX components
- Vue-markdown allows us to render questions and answers markdown text as HTML

Open a terminal in the client folder or `cd` into it. Then install them all with a single command:

```
npm install --save-dev axios bootstrap-vue @fortawesome/fontawesome-free vue-markdown
```

If you would rather use different libraries or none at all and build the simplest frontend possible, you can still do so and follow along.

In order to run the frontend, you just need to open a terminal, navigate to the client folder and run the command npm run serve. This will start a webpack dev server with auto reload, so as soon as you change your code, the frontend is refreshed. You should see a message telling you that it is available on http://localhost:8080.

Let's now update the contents of the generated project so it resembles something similar to Stack Overflow. First replace the contents of the **src/main.js** file so it imports and wires the extra libraries we installed:

```
import Vue from 'vue'
import App from './App'
import router from './router'
import axios from 'axios'
import BootstrapVue from 'bootstrap-vue'
import 'bootstrap/dist/css/bootstrap.css'
import 'bootstrap-vue/dist/bootstrap-vue.css'
import '@fortawesome/fontawesome-free/css/all.css'

Vue.config.productionTip = false

// Setup axios as the Vue default $http library
axios.defaults.baseURL = 'http://localhost:5000' // same as the Url the server
listens to
Vue.prototype.$http = axios

// Install Vue extensions
Vue.use(BootstrapVue)

new Vue({
  router,
  render: h => h(App)
}).$mount('#app')
```

Nothing extraordinary, we are just importing the extra libraries we have installed and performing any initialization required with Vue. Next update the contents of App.vue so our project contains a navbar with a main area where each of the pages will be rendered:

```
<template>
 <div id="app">
  <nav class="navbar navbar-expand-md navbar-dark bg-dark shadow">
   <a class="navbar-brand" href="#/">mini-SO</a>
   <button class="navbar-toggler" type="button" data-toggle="collapse" data-
   target="#main-navbar" aria-controls="main-navbar" aria-expanded="false" aria-
   label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
   </button>

   <div class="collapse navbar-collapse" id="main-navbar">
    <ul class="navbar-nav mr-auto">
     <li class="nav-item active">
      <a class="nav-link" href="#/">Home <span class="sr-only">(current)</span></a>
     </li>
    </ul>
    <form class="form-inline my-2 my-lg-0">
     <input class="form-control mr-sm-2" type="text" placeholder="Search" aria- \
```

```
      label="Search">
      <button class="btn btn-secondary my-2 my-sm-0" type="submit">Search</button>
     </form>
    </div>
   </nav>

   <main role="main" class="container mt-4">
    <router-view/>
   </main>
  </div>
</template>

<script>
export default {
 name: 'App'
}
</script>
```

This looks complicated but it is a basic navbar using bootstrap styling. Feel free to ignore the navbar altogether, the important part is to have a `<router-view />` element where each of the client pages will be rendered.

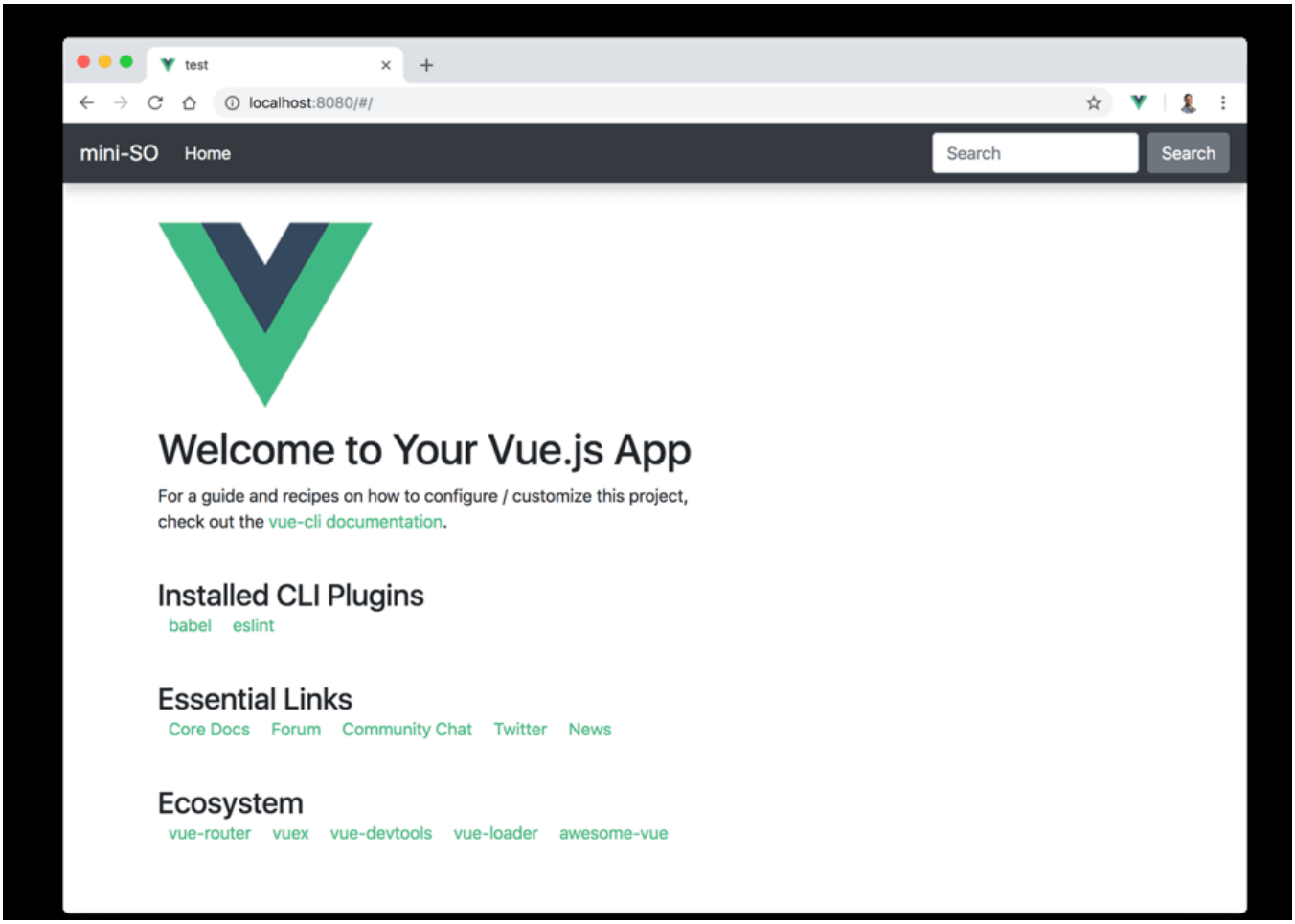


Figure 4, we now have a navbar!

Let's now replace the default page with the home page for our project, one where we will display the list of questions. Remove the existing files inside the **src/views** folder, then create a new file **home.vue** inside:

```
<template>
 <div>
  <h1>
```

```
   This totally looks like Stack Overflow
   <button v-b-modal.addQuestionModal class="btn btn-primary mt-2 float-right">
    <i class="fas fa-plus"/> Ask a question
   </button>
  </h1>
   <ul class="list-group question-previews mt-4">
    <question-preview
     v-for="question in questions"
      :key="question.id"
      :question="question"
      class="list-group-item list-group-item-action mb-3" />
   </ul>
  <add-question-modal @question-added="onQuestionAdded"/>
 </div>
</template>

<script>
import QuestionPreview from '@/components/question-preview'
import AddQuestionModal from '@/components/add-question-modal'

export default {
 components: {
  QuestionPreview,
  AddQuestionModal
 },
 data () {
  return {
   questions: []
  }
 },
 created () {
  this.$http.get('/api/question').then(res => {
   this.questions = res.data
  })
 },
 methods: {
  onQuestionAdded (question) {
   this.questions = [question, ...this.questions]
  }
 }
}
</script>

<style>
.question-previews .list-group-item{
 cursor: pointer;
}
</style>
```

It simply retrieves a list of question previews from our backend API and renders them in a list, as well as provides a button and a modal to create a new question. The component data contains a list of questions which is initially empty and is updated with the list retrieved from the backend as soon as the component gets added to the page. As you can see, it uses two components:

- The `<question-preview />` component is used to render a single question preview.
- The `<add-question-modal />` component provides a modal with a form for adding a new question. This component will submit the new question to the server and emit an event 'question-added' that can be listened to, so the new question preview is added to the list.

Of course, these two components don't exist yet, so let's add them. Start by adding a new file `add-question-modal.vue` inside the **src/components** folder with the following contents:

```
<template>
 <b-modal id="addQuestionModal" ref="addQuestionModal" hide-footer title="Add new
 Question" @hidden="onHidden">
 <b-form @submit.prevent="onSubmit" @reset.prevent="onCancel">
  <b-form-group label="Title:" label-for="titleInput">
   <b-form-input id="titleInput"
     type="text"
     v-model="form.title"
     required
     placeholder="Please provide a title">
   </b-form-input>
   </b-form-group>
   <b-form-group label="Your Question:" label-for="questionInput">
   <b-form-textarea id="questionInput"
     v-model="form.body"
     placeholder="What do you need answered?"
      :rows="6"
      :max-rows="10">
   </b-form-textarea>
  </b-form-group>

  <button class="btn btn-primary float-right ml-2" type="submit" >Submit</button>
  <button class="btn btn-secondary float-right" type="reset">Cancel</button>
 </b-form>
 </b-modal>
</template>

<script>
export default {
 data () {
  return {
   form: {
    title: '',
    body: ''
   }
  }
 },
 methods: {
  onSubmit (evt) {
   this.$http.post('api/question', this.form).then(res => {
    this.$emit('question-added', res.data)
    this.$refs.addQuestionModal.hide()
   })
  },
  onCancel (evt) {
   this.$refs.addQuestionModal.hide()
  },
  onHidden () {
   Object.assign(this.form, {
    title: '',
    body: ''
   })
  }
 }
}
</script>
```

This component provides a **bootstrap-vue** modal component and some form controls to enter the title and

body of a new question. When the user submits the form, the question is sent to the server and an event is emitted with the response from the server so it can be added to the list of previews displayed in the **home.vue** component.

Next add a new file **question-preview.vue** inside the same components folder with these contents:

```
<template>
 <li class="card container" @click="onOpenQuestion">
  <div class="card-body row">
   <question-score :question="question" class="col-1" />
   <div class="col-11">
    <h5 class="card-title">{{ question.title }}</h5>
    <p><vue-markdown :source="question.body" /></p>
    <a href="#" class="card-link">
     View question <span class="badge badge-success">{{ question.answerCount }}
     </span>
    </a>
   </div>
  </div>
 </li>
</template>
<script>

import VueMarkdown from 'vue-markdown'
import QuestionScore from '@/components/question-score'

export default {
 components: {
  VueMarkdown,
  QuestionScore
 },
 props: {
  question: {
   type: Object,
   required: true
  }
 },
 methods: {
  onOpenQuestion () {
   this.$router.push({name: 'Question', params: {id: this.question.id}})
  }
 }
}
</script>
```

This is a simple component that renders a bootstrap card with the question preview. Whenever the user clicks on the question, it will navigate to the question details page which we will add in a minute. It also uses one more component `<question-score />` that will render the current score, as well as buttons to up/down vote.

Continue adding the **question-score.vue** file inside the components folder with these contents:

```
<template>
 <h3 class="text-center scoring">
  <button class="btn btn-link btn-lg p-0 d-block mx-auto" @click.stop="onUpvote"><i
  class="fas fa-sort-up" /></button>
  <span class="d-block mx-auto">{{ question.score }}</span>
```

```
  <button class="btn btn-link btn-lg p-0 d-block mx-auto" @click.
  stop="onDownvote"><i class="fas fa-sort-down" /></button>
 </h3>
</template>
<script>
export default {
 props: {
  question: {
   type: Object,
   required: true
  }
 },
 methods: {
  onUpvote () {
   this.$http.patch(`/api/question/${this.question.id}/upvote`).then(res => {
    Object.assign(this.question, res.data)
   })
  },
  onDownvote () {
   this.$http.patch(`/api/question/${this.question.id}/downvote`).then(res => {
    Object.assign(this.question, res.data)
   })
  }
 }
}
</script>

<style scoped>
.scoring .btn-link{
  line-height: 1;
}
</style>
```

We have some bootstrap styling to render the question score along with the voting buttons. When the buttons are clicked, a request is sent to the server and the question preview is updated with the server response.

Now update the router inside **src/router.js** so it renders our completed home page:

```
import Vue from 'vue'
import Router from 'vue-router'
import HomePage from '@/views/home'

Vue.use(Router)

export default new Router({
 routes: [
  {
   path: '/',
   name: 'Home',
   component: HomePage
  }
 ]
})
```

The finished site should be reloaded automatically (if not, manually reload). You should see the list of questions and will be able to add new ones:

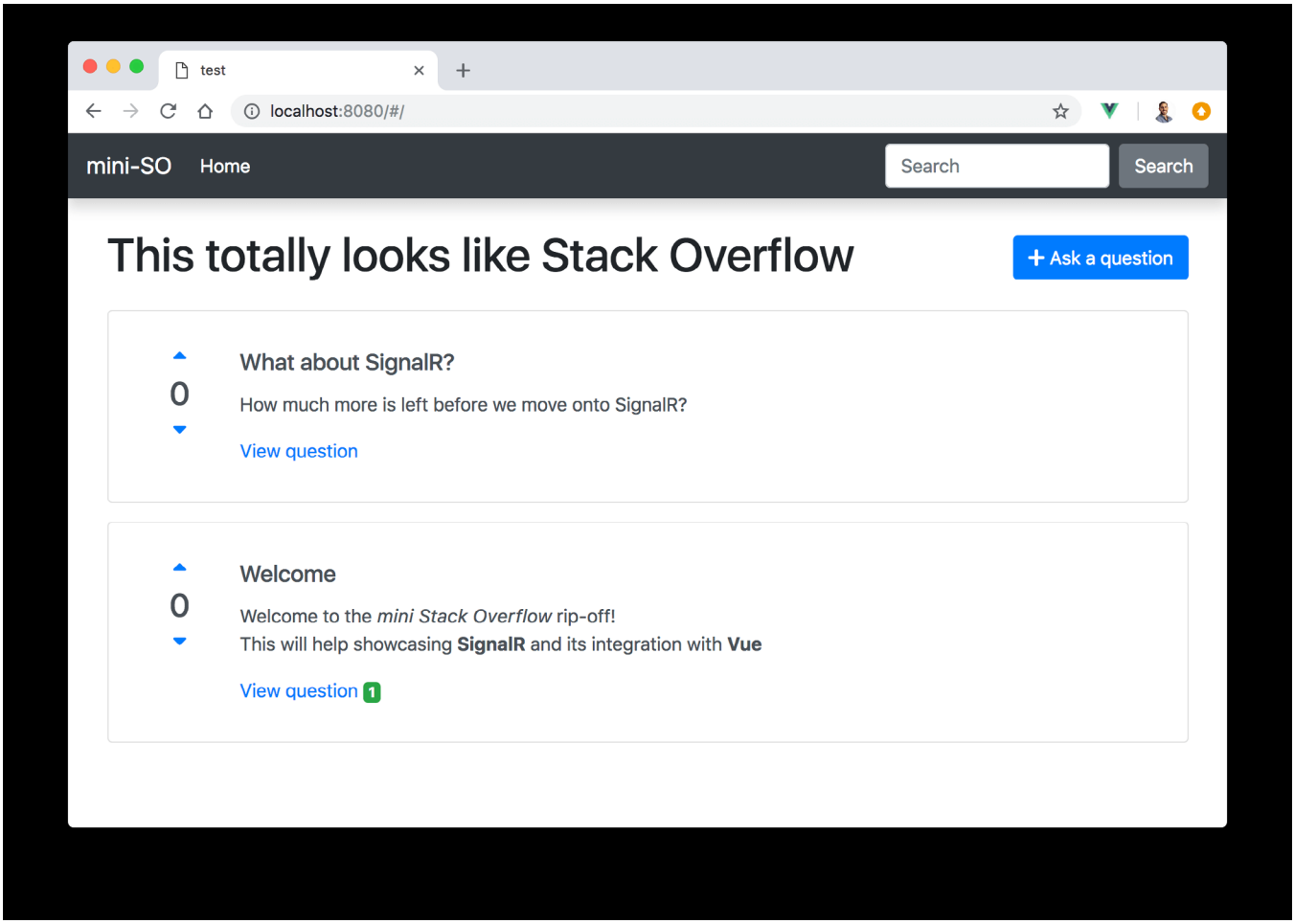


Figure 5, our site now lists questions and allows users to create them

All that's left now is to add the question details page. Users will navigate to this page whenever they click on one of the previews. The click handling code is already part of the `<question-preview />` component, but the page does not exist yet.

Let's add this page next. Create a new file **question.vue** inside the **src/views** folder with the following contents:

```
<template>
 <article class="container" v-if="question">
  <header class="row align-items-center">
   <question-score :question="question" class="col-1" />
   <h3 class="col-11">{{ question.title }}</h3>
  </header>
  <p class="row">
   <vue-markdown class="offset-1 col-11">{{ question.body }}</vue-markdown>
  </p>
  <ul class="list-unstyled row" v-if="hasAnswers">
   <li v-for="answer in question.answers" :key="answer.id" class="offset-1 col-11
   border-top py-2">
    <vue-markdown>{{ answer.body }}</vue-markdown>
   </li>
  </ul>
  <footer>
   <button class="btn btn-primary float-right" v-b-modal.addAnswerModal>
   <i class="fas fa-edit"/> Post your Answer</button>
   <button class="btn btn-link float-right" @click="onReturnHome">Back to list
   </button>
```

```
  </footer>
  <add-answer-modal :question-id="this.questionId" @answer-added="onAnswerAdded"/>
 </article>
</template>

<script>
import VueMarkdown from 'vue-markdown'
import QuestionScore from '@/components/question-score'
import AddAnswerModal from '@/components/add-answer-modal'

export default {
 components: {
  VueMarkdown,
  QuestionScore,
  AddAnswerModal
  },
  data () {
   return {
    question: null,
    answers: [],
    questionId: this.$route.params.id
   }
  },
  computed: {
   hasAnswers () {
    return this.question.answers.length > 0
   }
  },
  created () {
   this.$http.get(`/api/question/${this.questionId}`).then(res => {
    this.question = res.data
   })
  },
  methods: {
   onReturnHome () {
    this.$router.push({name: 'Home'})
   },
   onAnswerAdded (answer) {
    if (!this.question.answers.find(a => a.id === answer.id)) {
     this.question.answers.push(answer)
    }
   }
 }
}
</script>
```

This page gets the question Id from the route parameters, loads the full question details with the list of answers, and proceeds to display them on that page. It also contains buttons to get back to the home page or add a new answer. The `<add-answer-modal />` component is really similar to the one we created for adding questions, except for the fact an answer doesn't have a title. I won't copy it here but if you have trouble adding it, check the source on GitHub.

All that's left is to update the vue-router with a new route for the question page. If you remember, the code handling the click on a question preview had the following navigation:

```
this.$router.push({name: 'Question', params: {id: this.question.id}})
```

We just need to add a new route inside the **src/router.js** file like:

```
export default new Router({
 routes: [
  … // previous route
  {
   path: '/question/:id',
   name: 'Question',
   component: QuestionPage
  }
 ]
})
```

..where the QuestionPage is imported as `import QuestionPage from '@/views/question'`. If you navigate to your site again in the browser, you should now have a very simple questions and answers site that lets you see a list of questions, open one of them to see its answers, add new questions/answers and vote on questions.

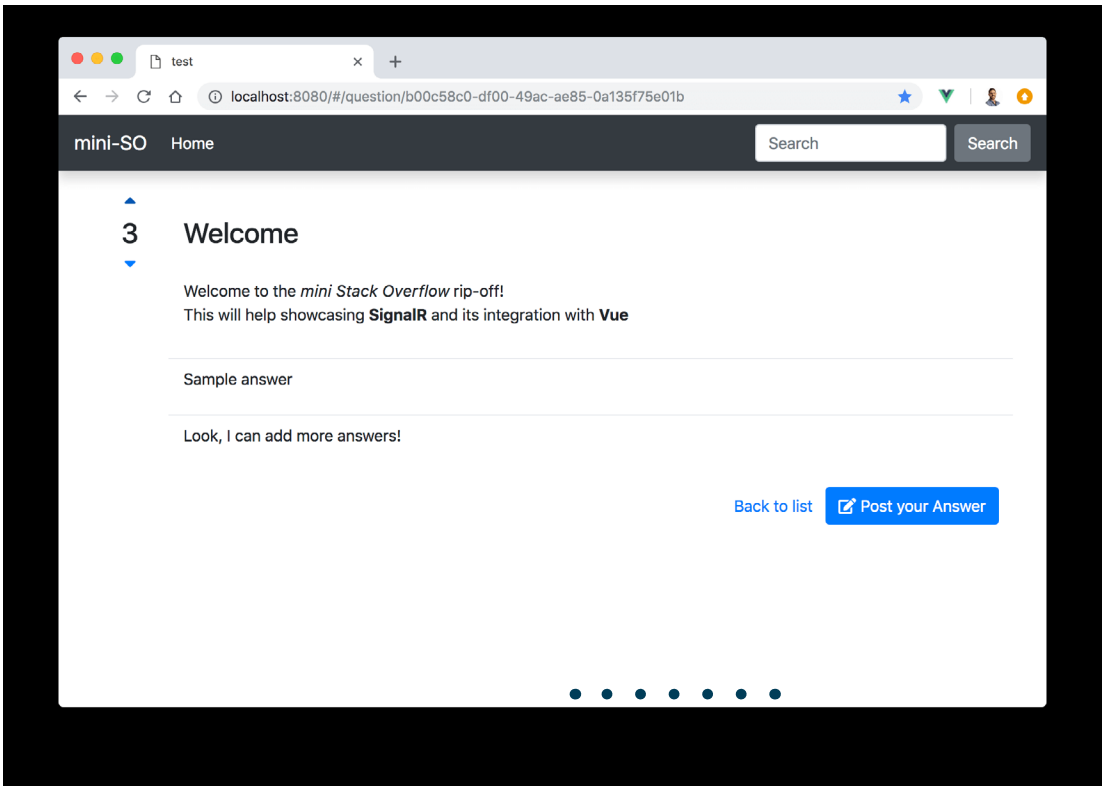


Figure 6, the question details page listing answers and allowing new answers to be added

At this point we have a simple site but interesting enough to build upon with the real-time functionality provided by SignalR.

2. CREATING AND CONNECTING TO SIGNALR HUBS

It is now time to start digging into the main point of the article, which is SignalR. The library is already part of the `Microsoft.AspNetCore.App` meta package which comes installed by default when using any of the default ASP.NET Core templates.

Let's start by adding a SignalR Hub to our backend ASP.NET Core application. A **Hub** is the main building block in SignalR. It abstracts a connection between client and server and provides the API for sending events in both directions.

Create a new folder named **Hubs** inside the server project and add a new **QuestionHub.cs** file. There we will create our hub which is just a class that inherits from the base `Hub` class:

```
using Microsoft.AspNetCore.SignalR;

namespace server.Hubs
{
  public class QuestionHub: Hub
  {
  }
}
```

Now let's modify our server `Startup` class to enable SignalR and register our Hub. Update the `ConfigureServices` method with:

```
services.AddSignalR();
```

Then update the `Configure` method in order to make our Hub available to clients:

```
app.UseSignalR(route =>
{
  route.MapHub<QuestionHub>("/question-hub");
});
```

Our clients will then be able to connect with our hub at http://localhost:5000/question-hub. Let's probe that this really works by updating our client application so it establishes a connection to the Hub.

Like we mentioned before, the Hub abstracts the connection between the client and the server and provides the necessary API to send/receive events at both ends. There is a number of clients available in different languages, including JavaScript, which will manage establishing a connection to a Hub, as well as sending/receiving events.

Install the JavaScript client by opening a terminal in the client folder and running the command:

```
npm install --save-dev @aspnet/signalr
```

Now add a new file **question-hub.js** inside the client **src** folder. Using the SignalR client to connect to the Hub is as simple as:

```
import { HubConnectionBuilder, LogLevel } from '@aspnet/signalr'

const connection = new HubConnectionBuilder()
  .withUrl('http://localhost:5000/question-hub')
  .configureLogging(LogLevel.Information)
  .build()

connection.start()
```

Then update your src/main.js file to simply import the file we just created:

```
import ‘./question-hub’
```

Make sure the client is running, restart it if you stopped it when installing the SignalR client library. You should now be able to open the developer tools, reload the page and see several requests if you filter by */question-hub*. There will be an ajax call used to negotiate the protocol to be used in the connection, and assuming your browser support web sockets, then you will see a web socket opened.

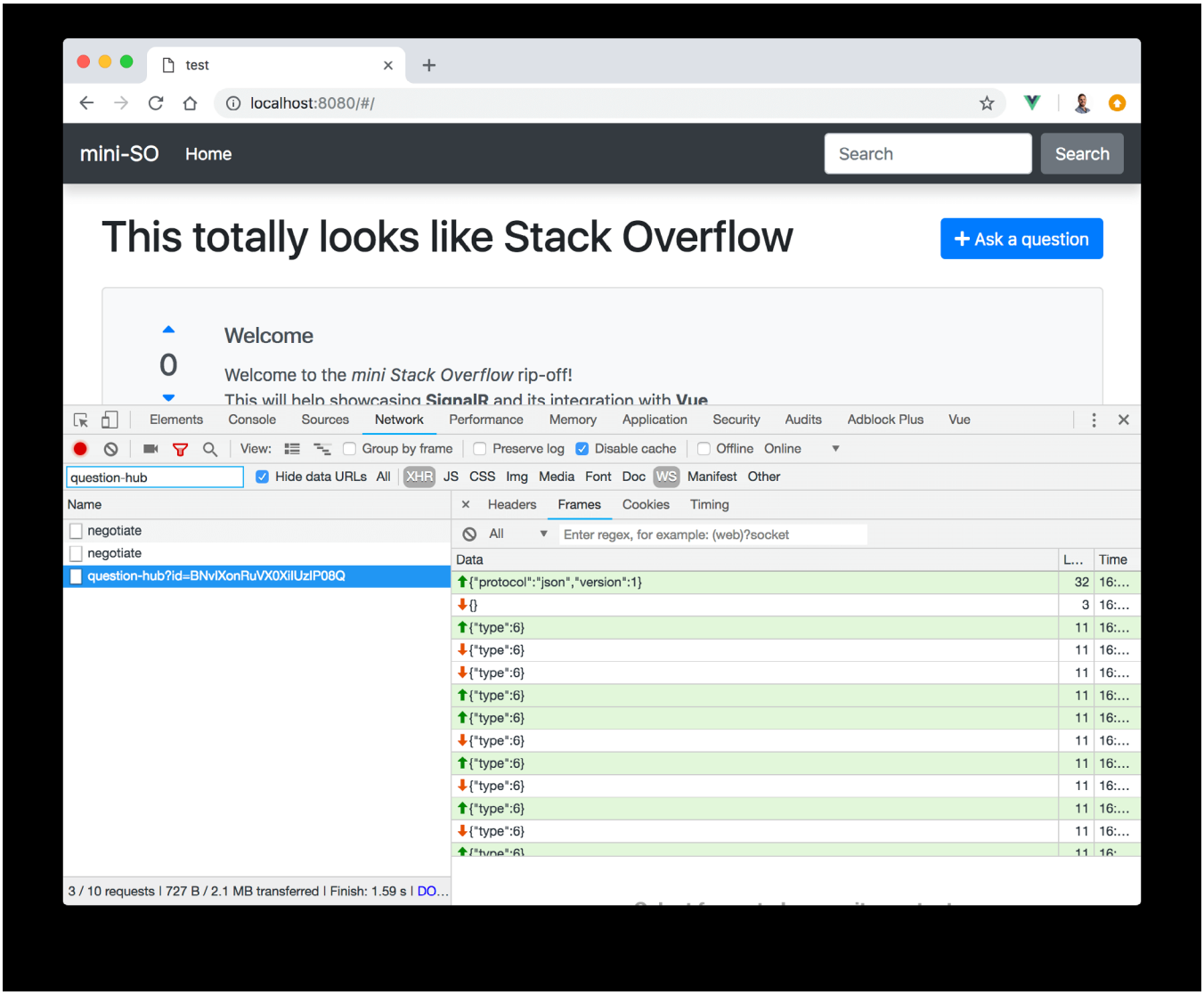


Figure 7, establishing a connection to the question Hub

CREATING A VUE PLUGIN

Our frontend is a Vue application, with an objective to send and receive SignalR events inside the Vue components. To better achieve this objective, we will need to eventually integrate the SignalR JavaScript client with Vue. A Vue plugin is the best way to add new global functionality to Vue.

Let’s convert the code inside question-hub.js into a Vue plugin. Right now, it will simply establish the connection, but we will keep coming back to this file as we add the functionality needed to send/receive events. In order to create a plugin, we just need to export an object with an `install` function, where the first argument is the Vue object:

```
import { HubConnectionBuilder, LogLevel } from '@aspnet/signalr'
export default {
 install (Vue) {
  const connection = new HubConnectionBuilder()
  .withUrl('http://localhost:5100/question-hub')
  .configureLogging(LogLevel.Information)
  .build()

  connection.start()
 }
}
```

Then update the **src/main.js** file so our plugin is installed before the vue application is started. Replace the previous import line with `import QuestionHub from './question-hub'`, then add the following line right before initializing the Vue app:

```
Vue.use(QuestionHub)
```

That's it, we now have a Vue plugin that initializes the connection to the question Hub provided by our backend. Before we start sending events through the Hub connection, there are a couple of improvements we can do.

Firstly, we can remove the hardcoded host from the `install` function of our Vue plugin. We can read this from the `baseURL` we configured for axios, where axios is available as `Vue.prototype.$http`. We can then replace the line

```
.withUrl('http://localhost:5000/question-hub')
```

with

```
.withUrl(`${Vue.prototype.$http.defaults.baseURL}/question-hub`)
```

Secondly, we can provide some simple strategy to reconnect if the Hub connection is closed. This is something that the JavaScript client doesn't do by default as advised by the official documentation.

The simplest approach is to try and reconnect if notified of the connection close event. Replace the line connection.start() with the following block:

```
let startedPromise = null
function start () {
 startedPromise = connection.start().catch(err => {
  console.error('Failed to connect with hub', err)
  return new Promise((resolve, reject) =>
  setTimeout(() => start().then(resolve).catch(reject), 5000))
 })
 return startedPromise
}
connection.onclose(() => start())

start()
```

If the connection is closed, we simply try to connect again. In the event we couldn't establish the connection, it retries after 5 seconds. This is admittedly simple! As advised in the docs, you might end up adding some throttling that increases the time between retries.

It is worth highlighting that we are keeping a variable `startedPromise`, which is a promise resolved when the connection is established. This will be very useful later, as it will let us wait for the connection to be established before trying to send an event to the server!

Note: *if you are using `dotnet watch run` to run the server application, you will start experiencing the watch process crashing once you add the reconnect code. This seems to happen because of the Hub connection keeping the port in use so the restarted server process cannot start listening on the same port.*
I haven't found any solution other than restarting the watch process. The alternative is to disable the reconnect code in development, but then you will need to manually reconnect after server-side changes by reloading the page in the browser.

3. SERVER TO CLIENT EVENTS

So far, we have created a Hub on the server and added a plugin to our Vue app that establishes a connection to the Hub. It is time to start sending events through the connection, beginning with events from the server to the client. Within the context of our application, notifying the client whenever there is a question score change is a great candidate to test this part of SignalR.

SENDING AN EVENT FROM THE CONTROLLER

The `Hub` class we inherit from in `QuestionHub` provides the API needed to send events to clients through the Hub connection. A Hub method like the following one will send an event named "MyEvent" to the client:

```
public async Task SendMyEvent() {
  await Clients.All.SendAsync("MyEvent");
}
```

You can add any extra parameters after the event name, and they will be serialized and received on the client. If you need a more granular control over whom to send the events to, the `Clients` object provides methods that lets you decide whom to send the event to. Check the documentation for more info.

However, in order to notify the client whenever the question score changes, we will need to send the events from the controller's action handling the /upvote and /downvote endpoints. Whenever you need to send events from a place other than the Hub itself, SignalR provides the HubContext. Inject a `HubContext<THub>` into the controller as in:

```
private readonly IHubContext<QuestionHub> hubContext;
public QuestionController(IHubContext<QuestionHub> questionHub) {
    this.hubContext = questionHub;
}
```

We could now add the following line to our `UpvoteQuestionAsync` method, which will send the event QuestionScoreChange to any client connected to the question Hub (after converting the method into an `async Task<ActionResult>` method):

```
await this.hubContext
  .Clients
  .All
  .SendAsync("QuestionScoreChange", question.Id, question.Score);
```

We could do the same from the `DownvoteQuestionAsync` which would finish our server side changes to notify clients of question score changes. However, we can still do better and avoid having to hardcode the

event name by using a strongly typed Hub/HubContext.

Create the following interface:

```
public interface IQuestionHub {
  Task QuestionScoreChange(Guid questionId, int score);
}
```

Then update our Hub so it inherits from Hub<T> as in:

```
public class QuestionHub: Hub<IQuestionHub>
```

Next update the controller to receive a HubContext<Thub, T> as in:

```
private readonly IHubContext<QuestionHub, IQuestionHub> hubContext;
public QuestionController(IHubContext<QuestionHub, IQuestionHub> questionHub)
{
  this.hubContext = questionHub;
}
```

Now, rather than using the `SendAsync` method of the Hub, you will be able to use the methods defined by the interface. The name of the event received in the client will be the same as the name of the interface method, in this case `QuestionScoreChange`. Now update the `UpvoteQuestionAsync` and `DownvoteQuestionAsync` methods with the following line to send the event:

```
await this.hubContext
  .Clients
  .All
  .QuestionScoreChange(question.Id, question.Score);
```

Let's now see how to receive the event on the client and have the components update with the new question score.

RECEIVING EVENTS ON THE CLIENT

On the client side, we have established a connection inside the `src/question-hub.js` file, but this isn't available to client components. In order to make it available, we will update the Vue prototype so every component has a `$questionHub` property they can use to listen to events.
We can easily achieve this by creating a new Vue instance, which provides a powerful events API by default. We can then listen to server-side events coming from the SignalR connection and propagate them through the `$questionHub` internal event bus. Add the following lines to the install method of our question-hub Vue plugin, right after creating the connection:

```
// use new Vue instance as an event bus
const questionHub = new Vue()
// every component will use this.$questionHub to access the event bus
Vue.prototype.$questionHub = questionHub
// Forward server side SignalR events through $questionHub, where components will
listen to them
connection.on('QuestionScoreChange', (questionId, score) => {
  questionHub.$emit('score-changed', { questionId, score })
})
```

As you can see, we are listening to the `QuestionScoreChange` event coming from the server, and we are forwarding it as a new `score-changed` event through the internal event bus $questionHub that our components have access to.

We could have directly added the SignalR connection object as `Vue.prototype.$questionHub = connection`, so we could directly use the SignalR client API within our Vue components. Although this could be a simpler approach, I instead decided to follow the one described earlier due to the following considerations:

- The Vue components will be completely ignorant of the SignalR JavaScript API. Instead they will use the familiar events API already present in Vue.

- The translation between SignalR events and the internal events through the `$questionHub` event bus allows us to use event names following a more idiomatic Vue style like `score-changed` rather than the method names of the `IHubContext` interface like `QuestionScoreChanged`. We can also wrap the arguments into a single object without having to define a DTO class to be used in the `IHubContext` interface.

- The small layer decoupling Vue code from the SignalR API provided by our plugin is the natural place to add more advanced functionality - like ensuring the connection is established before trying to send an event to the server (more on this later) or automatically cleaning up event handlers (outside the scope of the article, but we will see how to manually clean them up).

Making the SignalR connection directly available to Vue components might work better for you or you might prefer to do so. In that case, simply add `Vue.prototype.$questionHub = connection` as part of the plugin installation. You would then directly use the SignalR JavaScript API through `this.$questionHub` inside any component. You should be able to follow the rest of the article even if you use this approach.

The event is now being received on the client and any component could listen to it using the `$questionHub` event bus. Let's update the `question-score` component so it listens to any `score-changed` events coming through the `$questionHub`. Vue's lifecycle provides the perfect place to start listening to the event as part of its `created` hook:

```
export default {
 props: {
  question: {
   type: Object,
   required: true
  }
 },
 created () {
  // Listen to score changes coming from SignalR events
  this.$questionHub.$on('score-changed', this.onScoreChanged)
 },
 methods: {
  … existing onUpvote and onDownvote methods

  // This is called from the server through SignalR
  onScoreChanged ({questionId, score }) {
   if (this.question.id !== questionId) return
   Object.assign(this.question, { score })
  }
 } }
```

The component now receives the score-change event, unwraps the questionId and score properties, and updates its score, assuming questionId of the event matches the one for the question in the component props.

The final piece is to clean up the event listener once the component is destroyed. Simply add a new `beforeDestroy` lifecycle method, similar to the `created` one where we stop listening to the event:

```
beforeDestroy () {
  // Make sure to cleanup SignalR event handlers when removing the component
  this.$questionHub.$off('score-changed', this.onScoreChanged)
},
```

That's it! Try opening two different browser windows and vote on a question in one window to see the updated score in the other window:

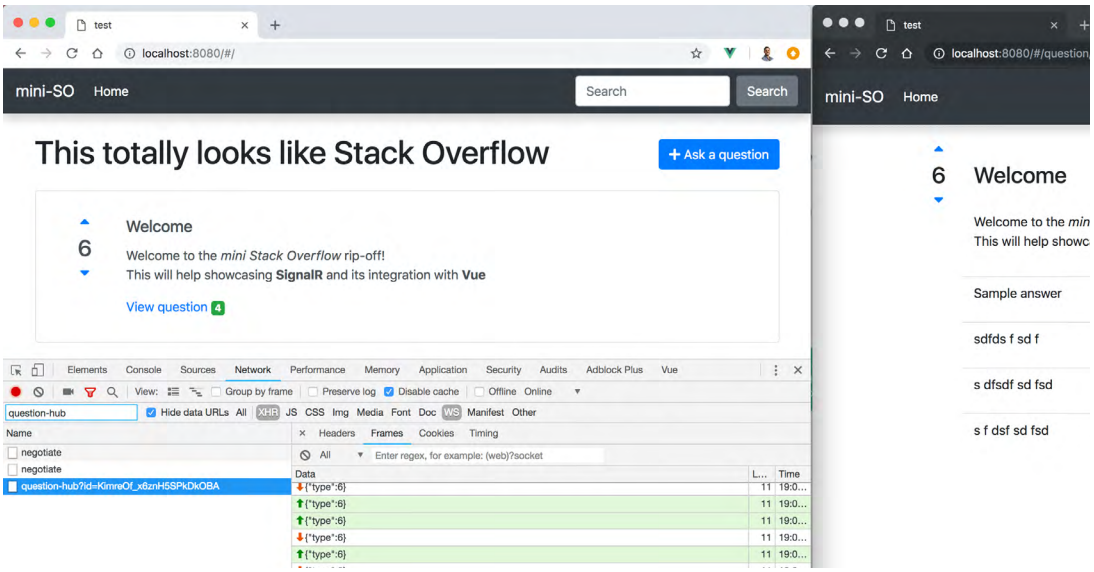


Figure 8, receiving score change events from the server

Adding another event with changes to the answers count would be very similar. I won't go through it here so we can move on, but you can check the source on GitHub if you want.

4. CLIENT TO SERVER EVENTS

As we have seen so far, receiving events from the server in the client is very straightforward. Let's now see if the opposite i.e. sending events from the client to the server, is equally easy.

We will add the following events from the client:

- Whenever the client navigates to the details page of a particular question, an event will be sent to the server. This event notifies the server that a specific client is actively looking at a specific question.
- Whenever navigating away from the details page, another event will be sent to the server. This second event notifies the server that a particular client is no longer looking at a specific question.

Apart from demonstrating how to send an event from the client, these will let us explore yet another SignalR feature, the groups. We will use the **groups** for sending an event from the server whenever a new answer is added to a question. However instead of sending it to every client connected to the Hub, we will only send it to clients actively looking at that particular question, which will be associated with a group.

SENDING EVENTS FROM THE CLIENT

The SignalR JavaScript API allows sending events as well as receiving them. If events were received using the `on` method, they are sent using the `invoke` method. The name reflects the fact that the event name provided needs to match the name of a method in the `QuestionHub` class.

Since we have the intermediate layer provided by our question-hub Vue plugin, we will add two new methods to the `$questionHub` object hiding the details of the SignalR API:

```
questionHub.questionOpened = (questionId) => {
  return connection.invoke('JoinQuestionGroup', questionId)
}

questionHub.questionClosed = (questionId) => {
  return connection.invoke('LeaveQuestionGroup', questionId)
}
```

These methods send each an event to the server, named `JoinQuestionGroup` and `LeaveQuestionGroup` respectively. The data provided with each event is simply the questionId.

Adding a listener on the server side is pretty straightforward.

SignalR expects the event names to exactly match a method in the Hub class (hence the event names like C# methods), which will be invoked whenever the event is received. Let's add them to our `QuestionHub` class. At this point, they won't do much other than writing to the log:

```
private readonly ILogger logger;
public QuestionHub(ILogger<QuestionHub> logger) {
  this.logger = logger;
}
public Task JoinQuestionGroup(Guid questionId) {
  this.logger.LogInfo($"Client {Context.ConnectionId} is viewing {questionId}");
}
public Task LeaveQuestionGroup(Guid questionId) {
  this.logger.LogInfo($"Client {Context.ConnectionId} is no longer viewing
  {questionId}");
}
```

Next update the **question.vue** page to send these events whenever the page is opened or closed. We just need to use the `created` and `beforeDestroy` lifecycle events like we did before:

```
created () {
 // Load the question and notify the server we are watching the question
 this.$http.get(`/api/question/${this.questionId}`).then(res => {
  this.question = res.data
  return this.$questionHub.questionOpened(this.questionId)
 })
},
beforeDestroy () {
 // Notify the server we stopped watching the question
 this.$questionHub.questionClosed(this.questionId)
},
```

Run the application, open a question and go back to the home page. You should see the entries we added

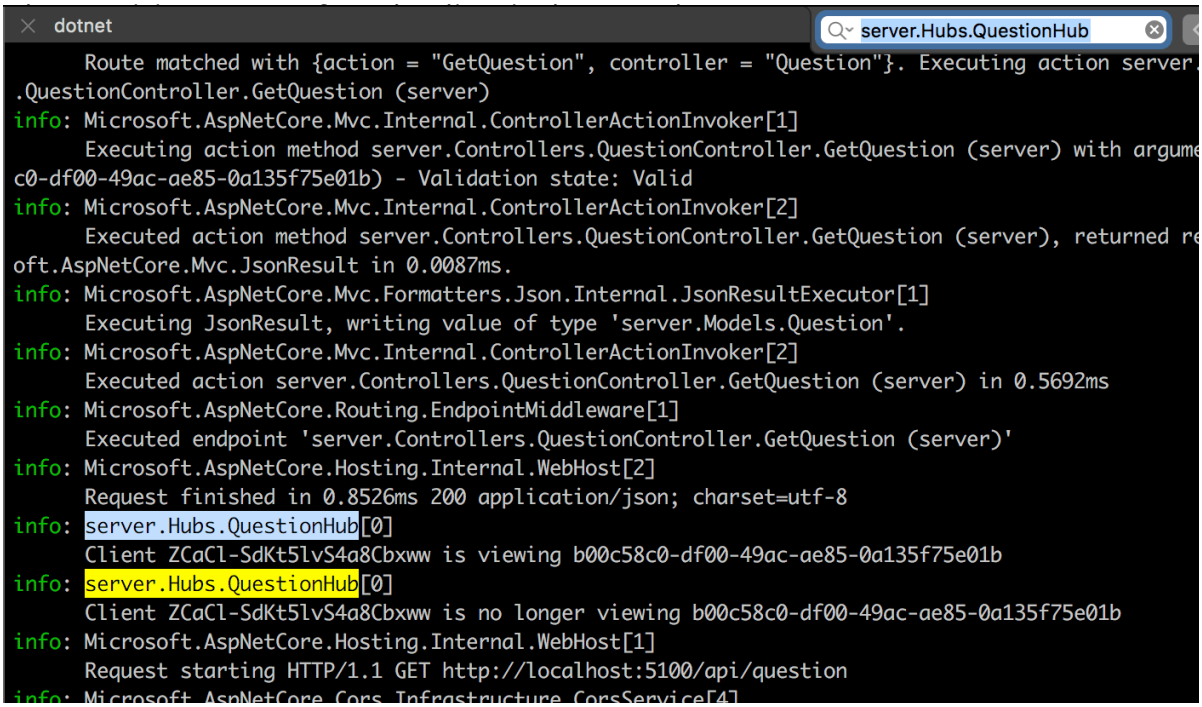


Figure 9, receiving events on the server

Before we move on, let's solve a problem with the current code inside **question-hub.js** for sending the events. To illustrate, open the details page of a question and then refresh the browser page, you will see the following error in the browser console:

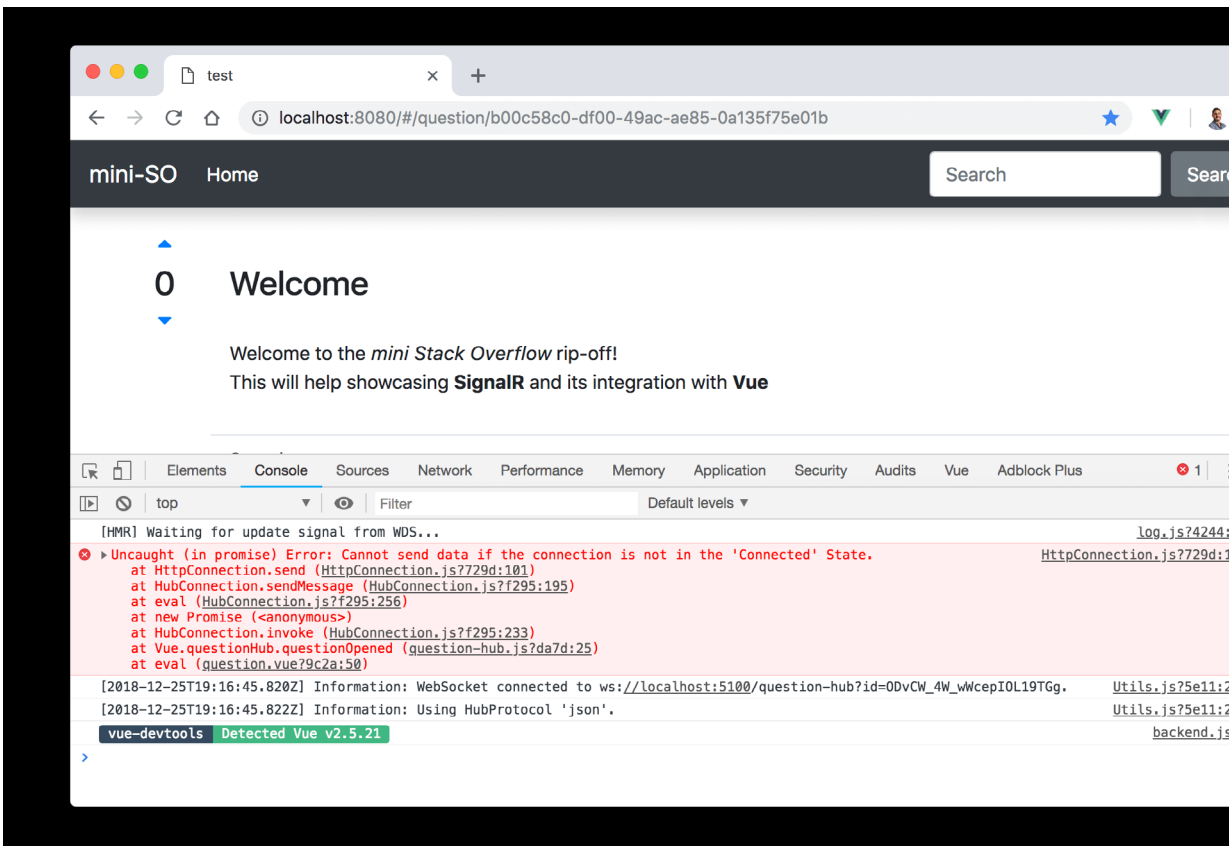


Figure 10, trying to send an event before the connection is established

This is because we are starting the application directly on the question page, which tries to send the event as soon as the page component is created. However, most likely the connection to the Hub hasn't been established yet. We can solve this problem by using the `startedPromise` that gets assigned a promise

that's only resolved once the connection is established.

Update the methods added to `$questionHub` as follows:

```
questionHub.questionOpened = (questionId) => {
 return startedPromise
  .then(() => connection.invoke('JoinQuestionGroup', questionId))
  .catch(console.error)
}

questionHub.questionClosed = (questionId) => {
 return startedPromise
  .then(() => connection.invoke('LeaveQuestionGroup', questionId))
  .catch(console.error)
}
```

Now these methods will wait for the connection to be established before trying to send any event, getting rid of the error. This is also one of the reasons why we don't let components directly access the SignalR connection! (As discussed in Section 3)

USING SIGNALR GROUPS

Up to this point, the server receives an event whenever the client opens and closes one of the question pages. We will use these events to add the client to a group specific for the question, using the `questionId` as the **group name**.

This way, by sending an event to all clients in a group, we will be able to send the event only to clients viewing a specific question, as opposed to every client connected to the Hub. Another classic example for groups would be creating a group per each room in a chat application.

Using groups is really easy, but in order to add a client to a group we need to know its `connectionId`. This is really simple from within the Hub class, since we have access to the `Context.ConnectionId`. Updating the `JoinQuestionGroup` and `LeaveQuestionGroup` methods to add/remove the current client from the question group is as easy as:

```
public async Task JoinQuestionGroup(Guid questionId)
{
  await Groups.AddToGroupAsync(Context.ConnectionId, questionId.ToString());
}

public async Task LeaveQuestionGroup(Guid questionId)
{
  await Groups.RemoveFromGroupAsync(Context.ConnectionId, questionId.ToString());
}
```

Doing the same from outside the Hub class would be much more complicated since you won't have access to that `Context` property. The client-side API also hides the connectionId. This seems to be deliberate and I can only guess why they decided to do so. Rethink your approach if you find yourself having to use the connectionId outside the Hub class!

We can now send a new event whenever a new answer is added, but only to clients currently part of the group for that question.

Begin by adding a new method to the IQuestionHub interface

```
Task AnswerAdded(Answer answer);
```

..then update the `AddAnswerAsync` action of the `QuestionController` so we send that event to the question group:

```
await this.hubContext
  .Clients
  .Group(id.ToString())
  .AnswerAdded(answer);
```

This completes the server side.

From the client point of view, there is no difference between listening to the new `AnswerAdded` SignalR event or the previous `QuestionScoreChange` event. Add a new listener to the **question-hub** Vue plugin like:

```
connection.on('AnswerAdded', answer => {
  questionHub.$emit('answer-added', answer)
})
```

And then update the **question.vue** page so that it listens to the new event and includes the answer in the list (where the function `onAnswerAdded` already exists in the component):

```
created () {
  // previous code
  …
  this.$questionHub.$on('answer-added', this.onAnswerAdded)
},
beforeDestroy () {
  // previous code
  …
  this.$questionHub.$off('answer-added', this.onAnswerAdded)
},
```

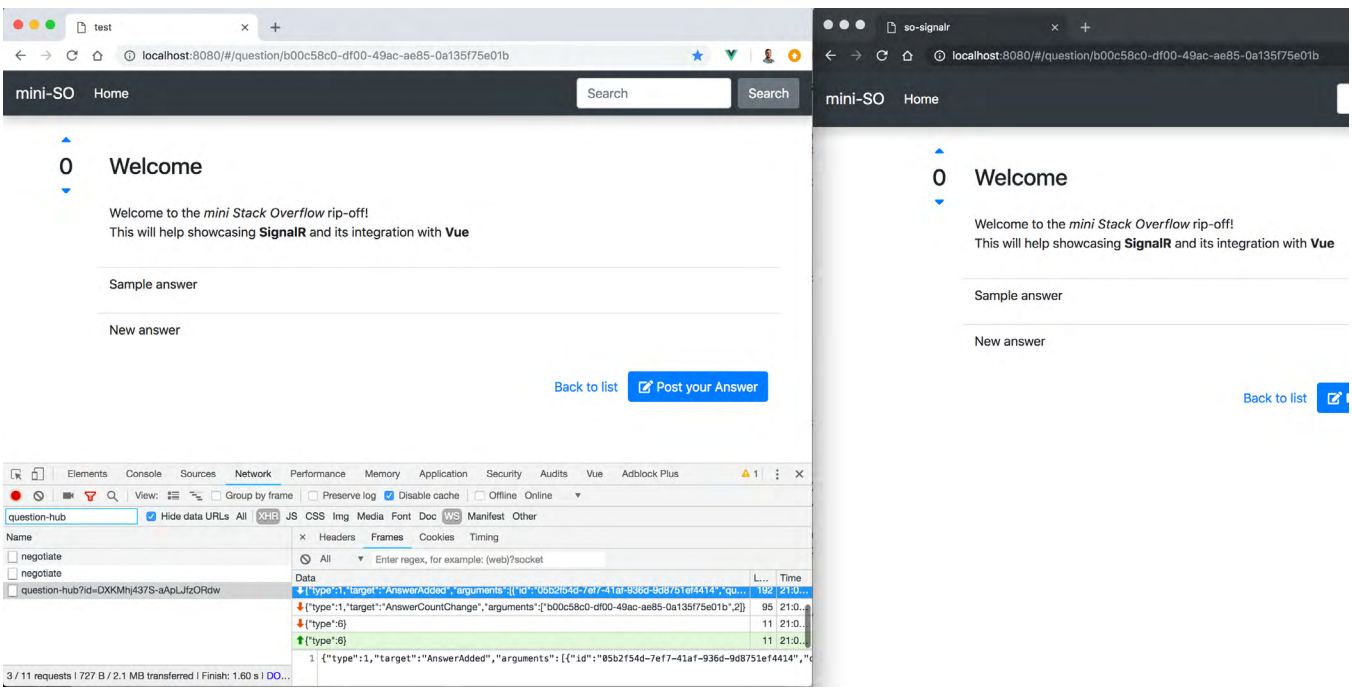


Figure 11, receiving new answer events when viewing a question

That's it, now clients viewing a question will be notified if an answer is added to that question. Any other client who isn't viewing that same question won't be notified.

CONCLUSION

SignalR is a simple but powerful library for implementing real time functionality in ASP.NET Core applications. This article, although lengthy, only scratched the surface of what's available, without having even gotten near concerns like hosting, authentication or scaling.

The main building blocks like the `Hub`, `HubContext` or individual events are relatively straightforward to understand and use. Perhaps the strongly typed interfaces that can be combined with the `Hub` and `HubContext`, like the `IQuestionHub` created in this article, are one of the most confusing pieces (For example, it took me a while to realize those interfaces don't need to be implemented at all by the Hub class).

Reading the documentation is highly recommended as it provides enough detail on how to get started and move onto advanced topics. I still find it easier to think in terms of events rather than methods though. If you are too used to libraries like socketio, doing the same mental translation might help you when digging into SignalR, which talks about RPC calls and methods, rather than events and listeners.

Personally, I also appreciate the JavaScript client being a minimal API, as opposed to trying to do everything for you. It does little but what it does, it does well - like connecting to a hub, sending/receiving events and serializing/deserializing parameters. How to integrate with your frontend framework of choice or add robustness like auto-reconnect, is up to you. Being a minimal API also gives you greater freedom when integrating into your app/framework. The discussion we had in section 3 is an example of this freedom.

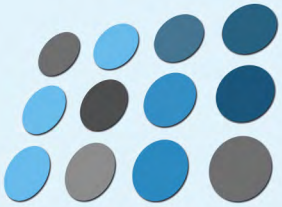
Download the entire source code from GitHub at
bit.ly/dncm40-so-signalr

Daniel Jimenez Garcia

Author

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

Thanks to Damir Arh for reviewing this article.



nopCommerce

open-source Shopping cart

N1. ASP.NET Open-source & Free eCommerce Solution

10 years

on the eCommerce market

50,000+ live shops

from small to large business with 10 000+ new stores every year

Modern tool stack

ASP .NET Core 2.1 Compatible with Azure

2,000+ integrations

third-party plugins, themes, language packs

2.5 million+

times downloaded

Stable and fast, trusted by big names:

TURKISH AIRLINES · VOLVO · CASIO · HARMAN · INTERSPORT · IKEA · HERBALIFE NUTRITION

LEARN MORE

Yacoub Massad

DESIGNING DATA OBJECTS: MORE EXAMPLES

In this article I am going to go through some examples of data object design and discuss some issues with them.

INTRODUCTION

In a previous article, Designing Data Objects in C# and F#, I gave some recommendations on how to design data objects.

In this article, I am going to go through some examples of data object designs that have some issues. I will discuss the issues with such designs and propose solutions for them.

The issues I discuss here are based on real issues that I have seen in the code bases that I work with. This

does not mean that the code in the examples here is real code. The problem domains I use in the article are different. The examples have also been simplified to fit into the article.

EXAMPLE #1 - TWO RELATED OPTIONAL PROPERTIES

```
public sealed class Customer
{
  public string Name { get; }

  public Maybe<DateTime> LastPurchaseDate { get; }

  public Maybe<decimal> LastPurchaseAmount { get; }

  //Constructor...
}
```

This class represents a customer in some shopping application. The `LastPurchaseDate` property is optional because its type is `Maybe<DateTime>`. The `LastPurchaseAmount` property is also optional.

In a nutshell, `Maybe<T>` can either have no value (NoValue) or have a specific value of type `T`.

For more information about Maybe, see the Designing Data Objects in C# and F# article.

The class also has a constructor that simply allows us to specify a value for each of the class properties. My issue with this design is that the customer object is modeled in a way that allows us to specify a value for `LastPurchaseDate` and specify a NoValue for `LastPurchaseAmount`, and vice versa.

In reality, either both should have values, or both should have no values. If this customer has at least purchased one item, both will have a value. If the customer has purchased nothing, both will have a NoValue.

Consider this updated design:

```
public sealed class Customer
{
  public string Name { get; }

  public Maybe<LastPurchaseDetails> LastPurchaseDetails { get; }

  //Constructor...
}

public sealed class LastPurchaseDetails
{
  public DateTime LastPurchaseDate { get; }

  public decimal LastPurchaseAmount { get; }

  //Constructor...
}
```

Now, the invalid states are no longer representable.

EXAMPLE #2 – CONVENIENT BUT BAD DATA OBJECT REUSE

Consider this interface that represents a service in an application that allows an administrator to manage a set of machines (e.g. PCs):

```
public interface IService
{
  ImmutableArray<Machine> GetMachines();

  void UpdateMachine(Machine machine);
}
```

The `Machine` class represents a single machine. It has some properties defined inside it.

```
public sealed class Machine
{
  public string Name { get; }

  public ImmutableArray<User> UsersLoggedInTheLast24Hours { get; }

  public bool GamingIsEnabled { get; }

  public bool PowerSavingIsEnabled { get; }

  public bool TurboSpeedIsEnabled { get; }

  //Constructor...
}
```

The `GetMachines` method is called by some GUI application to get details about the machines so that it can show them to the administrator. It displays the name of each machine, the list of users logged into that machine in the last 24 hours, and whether some features (gaming, power saving, and turbo speed) are enabled on that machine.

The `UpdateMachine` method is invoked by the GUI application when the administrator wishes to update certain settings on a specific machine. It takes a `Machine` object as a parameter.

The developer who designed this contract saw that it was convenient to use the same Data Object (i.e., the `Machine` class) to allow the administrator to view the machine information, and to also update machine settings.

However, such convenience comes at a cost.

For one, the reader of this contract can become confused about the `UsersLoggedInTheLast24Hours` property. What does it mean to update a specific machine with an updated value of this property? We sure can't change history and change the users who logged into the machine in the last 24 hours (we don't have a time machine). Maybe updating this value will change the logs recorded on that machine that contain the list of logged in users? Or maybe this property is ignored by the implementor of the `UpdateMachine` method?

After the developer who is reading the code goes to the implementation of the `UpdateMachine` method,

they find out that this property is completely ignored. The developer who wrote the code just used the same Data Object for convenience.

What about `GamingIsEnabled`, `PowerSavingIsEnabled`, and `TurboSpeedIsEnabled`? After reading the code, the developer finds out that only the first two of these properties can be changed, and the third one is simply ignored. Again, the developer who wrote this code used the same Data Object for convenience.

Although it means more code, the following design is better:

```
public interface IService
{
  ImmutableArray<Machine> GetMachines();

  void UpdateMachine(string machineName, MachineChanges changes);
}

public sealed class Machine
{
  //Machine class members here...
}

public sealed class MachineChanges
{
  public bool GamingIsEnabled{ get; }

  public bool PowerSavingIsEnabled { get; }

  //Constructor...
}
```

EXAMPLE #3 – USING AN EMPTY BYTE ARRAY TO MODEL A NO VALUE

Consider this `User` class:

```
public sealed class User
{
  public string Name { get; }

  public ImmutableArray<byte> Image { get; }

  public bool HasImage => Image.Length > 0;

  //Other properties...

  //Constructor...
}
```

One of the properties of the `User` class is the `Image` property which is of type `ImmutableArray<byte>`. This property contains a JPEG image of the user. Such image is displayed beside the user name in the application.

Notice the `HasImage` property. It returns `true` if the Image array is non-empty. Reading this code, one can understand that the image is optional. When the user has no image, the `Image` property would contain an empty byte array.

Although the HasImage property can help make the reader understand this design, it would be better if we use the Maybe type here to model the fact that the image of the user is optional:

```
public sealed class Customer
{
  public string Name { get; }

  public Maybe<ImmutableArray<byte>> Image { get; }

  //Other properties...

  //Constructor...
}
```

Note: *The Image property design can still be enhanced. For example, currently the property does not tell us that the contents of the byte array is actually a JPEG image. One enhancement for example is to encapsulate the byte array inside a JpegImage class.*

EXAMPLE #4 – A PROPERTY THAT HAS TWO SLIGHTLY DIFFERENT MEANINGS

Consider this data object:

```
public sealed class MobileDevice
{
  public bool IsOnline { get; }

  public GpsLocation GpsLocation { get; }

  public Brand Brand { get; }

  public string Name { get; }

  //Constructor...
}
```

The MobileDevice class models a mobile device in an application that allows users to monitor mobile devices. For example, employers can use the application to monitor special mobile devices they gave to their employees. Or parents can use the application to monitor the mobile devices of their children.

The IsOnline property is true if the mobile device is reachable, i.e., the application can communicate with the device. The GpsLocation property contains the location of the device based on the GPS device inside the mobile.

The Brand property contains information about the maker of the mobile device (e.g. Samsung, Apple, etc.).

The Name property is used by the user of the application to identify the device.

When reviewing the class, a code reviewer asked the developer who wrote the class the following question:

Reviewer: "If the device is online, the GpsLocation property would return the location of the device. That is understandable. However, when the mobile device is offline (IsOnline is false), what would be the value of

the GpsLocation property? Would it be null? If so, shouldn't you use the Maybe type to model the fact that it is optional?"

Developer: "No, it would contain the last known GPS location of the device. Do you think I should rename the property to CurrentOrLastKnownGpsLocation?".

Reviewer: "That would be better. Still, this name does not tell us when this property would contain the current location and when it would contain the last known location. In other words, it doesn't tell the reader about the relationship between this property and the `IsOnline` property. I have a better idea".

The reviewer suggests the following changes to the code:

```
public sealed class MobileDevice
{
  public ReachabilityStatus ReachabilityStatus { get; }

  public Brand Brand { get; }

  public string Name { get; }

  //Constructor...
}

public abstract class ReachabilityStatus
{
  private ReachabilityStatus(){ }

  public sealed class Online : ReachabilityStatus
  {
    public GpsLocation GpsLocation { get; }
    //Constructor...
  }

  public sealed class Offline : ReachabilityStatus
  {
    public GpsLocation LastKnownGpsLocation { get; }
    //Constructor...
  }
}
```

The `IsOnline` and the `GpsLocation` properties were removed. And a `ReachabilityStatus` property was added. This property contains a `Sum` type.

I explained Sum types in the Designing Data Objects in C# and F# article. In a nutshell a Sum type is a type whose values can be of one of a fixed number of types. For example, the values of the `ReachabilityStatus` type can either be of type `Online` or of type `Offline`. The Online type contains a single property called `GpsLocation` that contains the current location of the device. The Offline type's single property is correctly named `LastKnownGpsLocation`.

One could argue that the current code is verbose. This is true because C# does not have a way to model Sum types in a concise manner. In the Designing Data Objects in C# and F# article, I talked about how you can model Sum types concisely in F#. I also talked about how developers can model their data objects in F# and then write the behavior code that uses these data objects in C#. I leave the F# implementation of MobileDevice and ReachabilityStatus for the reader to write as an exercise. Refer to the mentioned article for help.

EXAMPLE #5 – USING DATETIME TO MODEL A DATE

This is a very simple example to explain. The `DateTime` struct is in the Base Class Library (BCL), and it models both a date and a time. Sometimes, we want to have a property in a Data Object to hold just a date. Consider this example:

```
public sealed class Member
{
  public DateTime SubscriptionDate { get; }
  //Other properties and constructor...
}
```

The member class represents a member in some club. The `SubscriptionDate` property represents the date when the member joined the club. In a particular application where this class exists, only the date when the member subscribed is important. Users of the application don't care if the member joined the club at 10 AM or 5 PM, they only care about the date.

The type of the property however consists of both a date and a time. What would the time part be in this case? 00:00? Or did the developer who developed this application just in case also included the time when the member joined?

It is better in this case to have a `Date` type to represent a date like this:

```
public sealed class Date
{
  public int Year { get; }
  public int Month { get; }
  public int Day { get; }
  //Constructor..
}
```

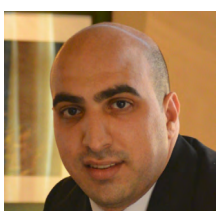
Conclusion:

In this article I gave examples of some Data Object designs and discussed some issues with them. I also provided sample solutions.

Yacoub Massad

Author

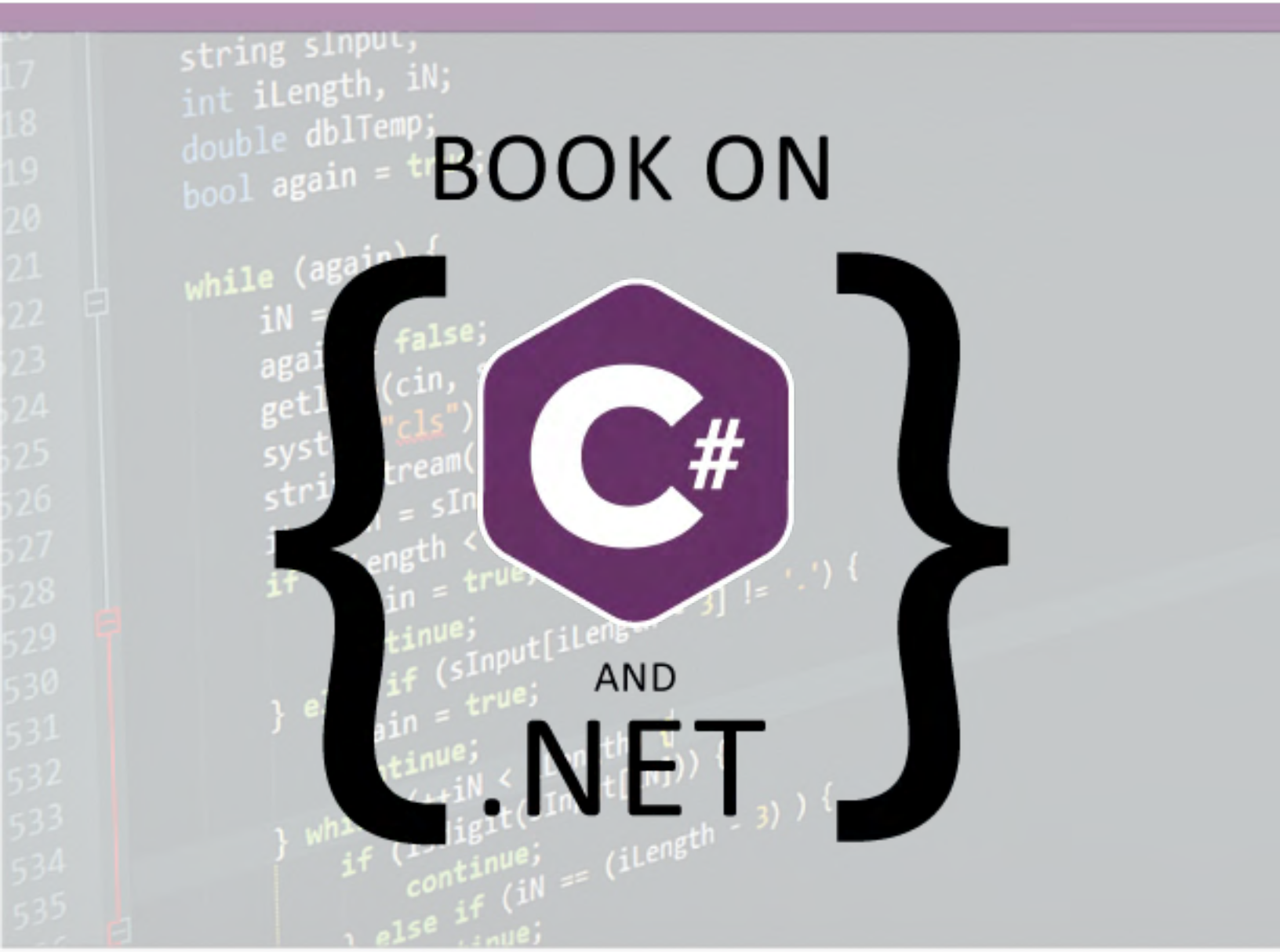
Yacoub Massad is a software developer who works mainly with Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable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



Thanks to Damir Arh for reviewing this article.

COVERS C# 7, C# 8 AND .NET Core

THE ABSOLUTELY AWESOME



PREORDER NOW!

Ravi Kiran

This tutorial continues the series on Unit Testing in Angular. At the end of this tutorial, you will get to know how to unit test directives and pipes in Angular.

UNIT TESTING **ANGULAR** DIRECTIVES AND PIPES

Introduction

Custom Directives and Pipes play a vital role in controlling the content displayed on the page. Unit testing them ensures the correctness of their behavior in different scenarios. As unit tests are done in isolation, the developer gets the freedom to replicate all the scenarios the directive or pipe would encounter. Unit tests also help in refactoring the code under test.

While writing unit tests, one could easily find if the piece of code is doing more than what it should do or if it is doing its job the right way. These scenarios could be otherwise difficult to find.

Angular makes the code written on top of it, unit testable. Angular also provides a lot of helper APIs to ease the job of developers to unit test Angular code. The article on Unit Testing Angular Components explains the testing setup in Angular CLI, shows how to structure a unit test file and discusses how to unit test Angular components. If you are new to unit testing in Angular, I would recommend reading that article first.

UNIT TESTING DIRECTIVES

Directives change the DOM of an element on which they are applied. Directives have a control over the UI that is finally rendered on the page. Having a set of unit tests for the directives would ensure that their functionality is correct.

Angular directives can't exist independently; they have to be used inside a component. To test a directive, it has to be hosted inside a component and the component's behavior has to be tested by manipulating the data.

Consider the following directive:

```
@Directive({
  selector: '[applyStyle]'
})
export class ApplyStyleDirective {
  constructor(private el: ElementRef, private renderer: Renderer2) {
    this.renderer.addClass(el.nativeElement, 'sample-text');
    this.renderer.setProperty(el.nativeElement, 'title', 'This is applied from the
    directive!');
  }
}
```

This directive sets a class and the title property to the element on which it is used. To test this, it has to be used on an element inside a test component and the behavior can be asserted by reading values of these style attributes. The following snippet shows a test component that uses this directive:

```
@Component({
  selector: 'test',
  template: `<div applyStyle>Some random text</div>`
})
class TestComponent { }
```

A testing module has to be created where the directive `ApplyStyleDirective` and the component `TestComponent` would be declared in the `declarations` block. A component fixture can be created using the testing module. A reference of the `div` element using the directive can be obtained using the fixture. The following snippet shows the `beforeEach` block with this setup:

```
import { Component, DebugElement } from '@angular/core';
import { ApplyStyleDirective } from './app-style.directive';
import { async, TestBed, ComponentFixture } from '@angular/core/testing';
import { By } from '@angular/platform-browser';

@Component({
  selector: 'test',
  template: `<div applyStyle>Some random text</div>`
})
class TestComponent { }

describe('AppStyleDirective tests', () => {
  let fixture: ComponentFixture<TestComponent>;
  let directiveElement: DebugElement;

  beforeEach(async(() => {
```

```
    fixture = TestBed.configureTestingModule({
      declarations: [
        ApplyStyleDirective, TestComponent
      ],
    }).createComponent(TestComponent);
    fixture.detectChanges();

    directiveElement = fixture.debugElement.query(By.
directive(ApplyStyleDirective));
  }));
});
```

Notice the way the directive is queried in the component. The `fixture.debugElement.query` queries the component and returns a `DebugElement` instance of the first occurrence of the directive. If there are multiple elements using the directive, then it can be replaced with the method `fixture.debugElement.queryAll`; this method returns an array of `DebugElements`.

The variable `directiveElement` can be used to check if the CSS class and the title are applied on the `div` element. The following test shows this:

```
it('should have applied the CSS class and title', () => {
  expect(directiveElement.classes['sample-text']).toEqual(true);
  expect(directiveElement.properties.title).not.toEqual('');
});
```

On running this test using the `npm test` command, you will see the following output on the browser running karma:

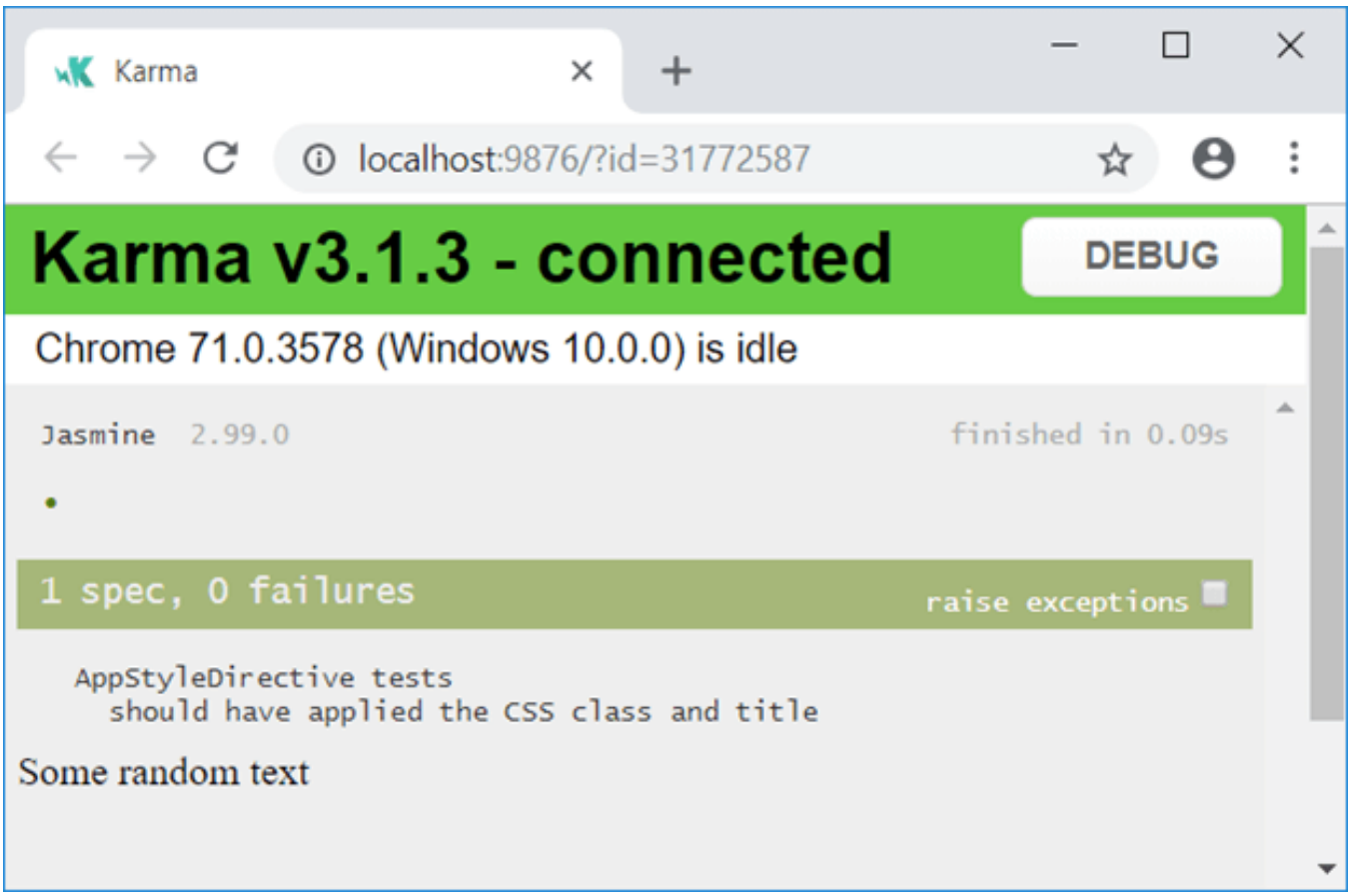


Figure 1 – Test results of applyStyle directive

DIRECTIVE WITH @INPUT

Directives often receive inputs from the containing component and their behavior would change based on value of the input object. The following snippet shows a modified version of the `ApplyStyleDirective` with the `Input` property:

```
import { Directive, OnInit, Input, ElementRef, OnChanges, Renderer2 } from '@
angular/core';

@Directive({
  selector: '[myAttr]'
})
export class MyAttrDirective implements OnInit, OnChanges {
  @Input() myAttr: any;

  constructor(private el: ElementRef,
    private renderer: Renderer2){ }

  ngOnInit() {
    this.applyStyle();
  }

  ngOnChanges() {
    this.applyStyle();
  }

  applyStyle() {
    this.renderer.setStyle(this.el.nativeElement, "color", this.myAttr.color);
    this.renderer.setStyle(this.el.nativeElement, "fontFamily", this.myAttr.
fontFamily);
  }
}
```

As you can see, this directive applies the styles it accepts from the component and modifies the styles when the input changes. The test component to be used for this directive has to create the object to be passed to the directive. The following snippet shows the test component:

```
import { MyAttrDirective } from './my-attr.directive';
import { Component, DebugElement } from '@angular/core';
import { ComponentFixture, async, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';

@Component({
  selector: 'test', template: `<div [myAttr]="styles">Some random text</div>`
})
class TestComponent {
  styles = {color: "white", fontFamily: "Verdana"};
}
```

The initial setup of the test is going to be similar to the `ApplyStyleDirective`. The following snippet shows this:

```
describe('MyAttrDirective', () => {
  let fixture: ComponentFixture<TestComponent>;
  let directiveElement: DebugElement;
```

```
  beforeEach(async(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [
        MyAttrDirective, TestComponent
      ],
    }).createComponent(TestComponent);
    fixture.detectChanges();

    directiveElement = fixture.debugElement.query(By.directive(MyAttrDirective));
  }));
});
```

The behavior of this directive can be tested using two tests; one to test if it binds the initial value; and the other to test if it updates the bindings after changing value of the input object. The following snippet shows these tests:

```
it('should create an instance', () => {
  expect(directiveElement.styles.color).toBe('white');
  expect(directiveElement.styles.fontFamily).toBe('Verdana');
});

it('should change style', () => {
  fixture.componentInstance.styles = {color: 'green', fontFamily: 'Arial'};
  fixture.detectChanges();

  expect(directiveElement.styles.color).toBe('green');
  expect(directiveElement.styles.fontFamily).toBe('Arial');
});
```

The second test makes a call to `fixture.detectChanges()` after modifying the `styles` object on the component instance. This will make the change detectors run on the component and its sub tree. This is when the directive's `ngOnChanges` method would be called and the new styles are applied on the element. On running these tests, Karma would produce the following output:

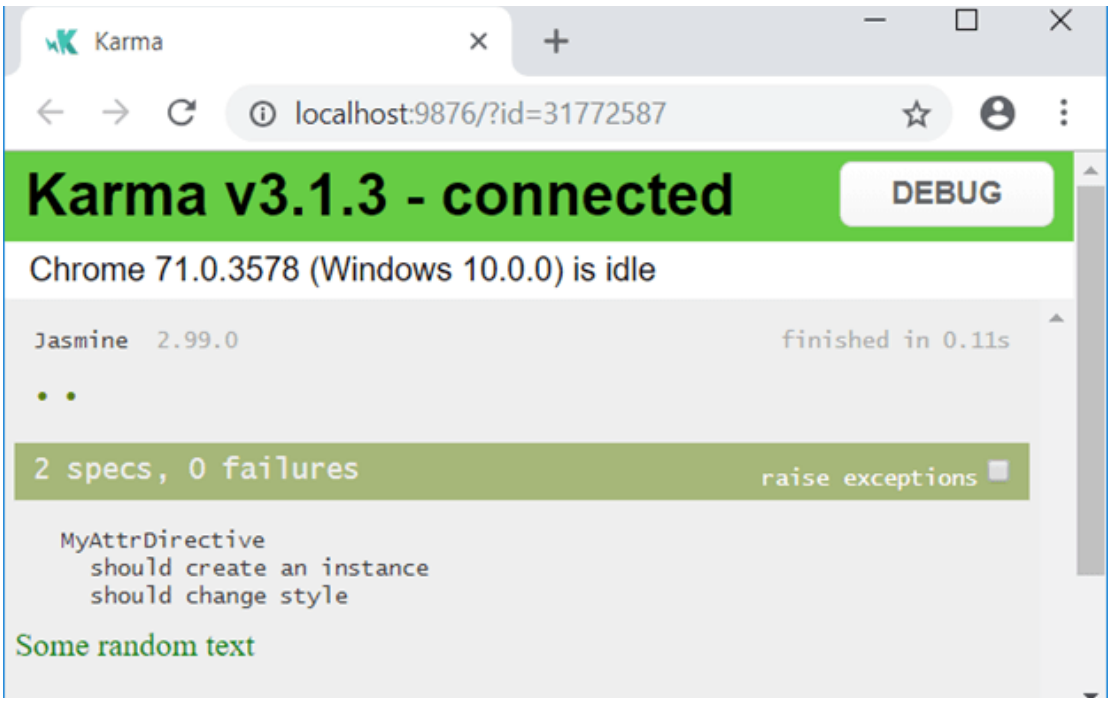


Figure 2 – Test results of myAttr directive

TESTING HOSTBINDINGS AND HOSTLISTENER

Directives can bind values to properties of the applied element using `HostBinding` and can listen to events on the element using `HostListener`. Testing whether the bindings are applied would be similar to the tests in the examples discussed earlier. To test the `HostListeners`, the event has to be triggered on the element and then the element has to be asserted for changes.

The following snippet shows a directive with `HostBinding` and `HostListener`:

```
import { Directive, Input, HostBinding, HostListener } from '@angular/core';

@Directive({
  selector: '[hostBindings]'
})
export class HostBindingsDirective {

  @Input() hostBindings: any;

  @HostBinding('style.color')
  get textColor() {
    return this.hostBindings.color;
  }

  @HostBinding('style.fontFamily')
  get textFont() {
    return this.hostBindings.fontFamily;
  }

  @HostBinding('style.fontStyle')
  fontStyle: string;

  @HostListener('mouseenter')
  mouseEntered(){
    this.fontStyle = 'italic';
  }

  @HostListener('mouseleave')
  mouseLeft(){
    this.fontStyle = 'normal';
  }
}
```

The test component and the setup for the tests would remain similar to the tests written in Directive with the `Input` section. So, I am leaving this as an exercise for the reader to do the setup or you can take a look at the sample code. The test to check if the initial styles are applied is going to remain similar to the way we tested values applied to the color and font family earlier. The following snippet shows this:

```
it('should apply styles to the element', () => {
  expect(directiveElement.styles.color).toEqual('green');
  expect(directiveElement.styles.fontFamily).toEqual('Verdana');
});
```

The events *mouseenter* and *mouseleave* have to be triggered in order to check the font style changes on the element. The events can be triggered using the `triggerEventHandler` method on the `directiveElement`. As the bindings need to be updated after the event, the `detectChanges` method has to be called on the `fixture` object. The following snippet shows the tests for *mouseenter* and *mouseleave*

events:

```
it('should modify font style when mouse enters the element', () => {
  directiveElement.triggerEventHandler('mouseenter', {});
  fixture.detectChanges();
  expect(directiveElement.styles.fontStyle).toEqual('italic');
});

it('should restore font style to normal when mouse leaves', () => {
  directiveElement.triggerEventHandler('mouseleave', {});
  fixture.detectChanges();
  expect(directiveElement.styles.fontStyle).toEqual('normal');
});
```

On running these tests, Karma would produce the following output:

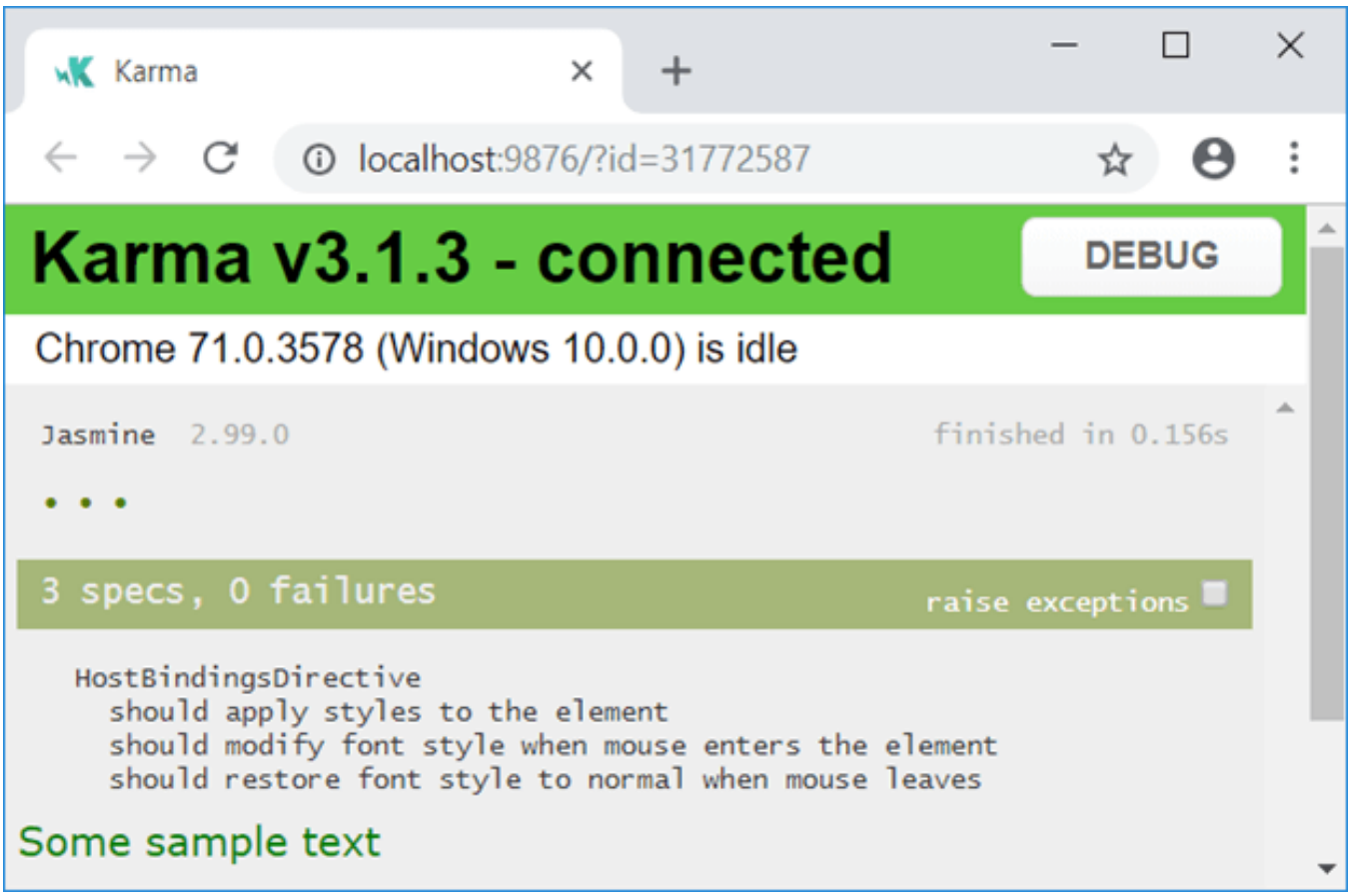


Figure 3 – Test results of hostBindings directive

UNIT TESTING PIPES

Pipes are used to transform data before it is bound to a control, so they play a vital role in formatting data in the right way for the end user. A few unit tests to ensure the correctness of the pipes would add value to the business.

Pipes can be tested either using instances of their TypeScript classes or can be tested using a test component like we did for the directives. This section will discuss both of these approaches.

Consider the following pipe:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'defaultvalue'
})
export class DefaultvaluePipe implements PipeTransform {
  transform(value: any, type: string): any {
    if (value === undefined || value === null || value === '') {
      if (type === 'number') {
        return 0;
      }
      else if (type === 'string') {
        return "-";
      }
      else if (type === 'boolean') {
        return false;
      }
      else if (type === 'object') {
        return {};
      }
    }
    return value;
  }
}
```

As you can see, the pipe class implements the `PipeTransform` interface and provides an implementation to the `transform` method. The `transform` method is called by Angular with the data on which the pipe is applied, and the additional parameters that are passed to the pipe. This `DefaultvaluePipe` checks whether the `value` is assigned and if not, it returns a default value based on the type passed to the pipe. This pipe can be used in the template as:

```
<div>{{employee | defaultvalue:'object'}}</div>
```

Let's test this pipe using an object of it. In the setup, the *spec* file has to create an instance of the pipe class. It is shown below:

```
import { DefaultvaluePipe } from './defaultvalue.pipe';

describe('DefaultvaluePipe', () => {
  let pipe: DefaultvaluePipe;

  beforeEach(() => {
    pipe = new DefaultvaluePipe();
  });
});
```

The tests will call the `transform` method of the pipe with a value and its type. The test will have an assertion around the value returned from this method. The first test will check if the pipe behaves well for the numbers. If the number doesn't have a value, it has to return 0; otherwise it has to return the number. The following tests verify this:

```
it('should return default value for number', () => {
  expect(pipe.transform(null, 'number')).toEqual(0);
});
```

```
it('should not return default value for number', () => {
  expect(pipe.transform(10, 'number')).toEqual(10);
});
```

Similarly, when the `transform` method is called with the type set to `object`, it would return an empty object; otherwise it would return the object itself. The following tests verify this behavior:

```
it('should return default value for object', () => {
  expect(pipe.transform(undefined, 'object')).toEqual({});
});

it('should not return default value for object', () => {
  expect(pipe.transform({val: 10}, 'object')).toEqual({val: 10});
});
```

Behavior of the pipe with Boolean and string type values can be asserted in the same way. You can implement them on your own or take a look at the sample code.

The best way to test a pipe is by hosting it in a test component. This is because the pipes would be eventually used in components in an Angular app. The element containing the pipe has to be inspected and assertions have to be written around the value bound in the element. The following snippet shows a test component using the `defaultvalue` pipe:

```
@Component({
  selector: 'test',
  template: `<div>{{employee | defaultvalue:'object' | json}}</div>`
})
class TestComponent {
  employee: any;
}
```

The test setup would be quite similar to the way it was done for the directives. The test module has to be created by passing the pipe `DefaultvaluePipe` and the component `TestComponent`. The only difference in the setup would be in getting a reference of the element on which the pipe is applied. The following snippet shows the setup and a test:

```
describe('DefaultvaluePipe from Component', () => {
  let fixture: ComponentFixture<TestComponent>;
  let divElement: DebugElement;

  beforeEach(async(() => {
    fixture = TestBed.configureTestingModule({
      declarations: [
        DefaultvaluePipe, TestComponent
      ],
    }).createComponent(TestComponent);
    fixture.detectChanges();

    divElement = fixture.debugElement.query(By.css('div'));
  }));

  it('should have the default value set', () => {
    expect(divElement.nativeElement.textContent).toEqual('{}');
  });
});
```

Initially, as the `defaultvalue` pipe receives an unassigned `employee` object, it returns an empty object. So the assertion compares the text content with stringified empty object. The following test asserts the behavior by assigning a value to the `employee` object:

```
it('should not show default value when the object has a value', () => {
  fixture.componentInstance.employee = { name: 'Ravi' };
  fixture.detectChanges();

  expect(divElement.nativeElement.textContent).not.toEqual('{}');
    expect(divElement.nativeElement.textContent).not.toEqual(JSON.stringify(fixture.
componentInstance.employee));
});
```

On running these tests, Karma would produce the following output:

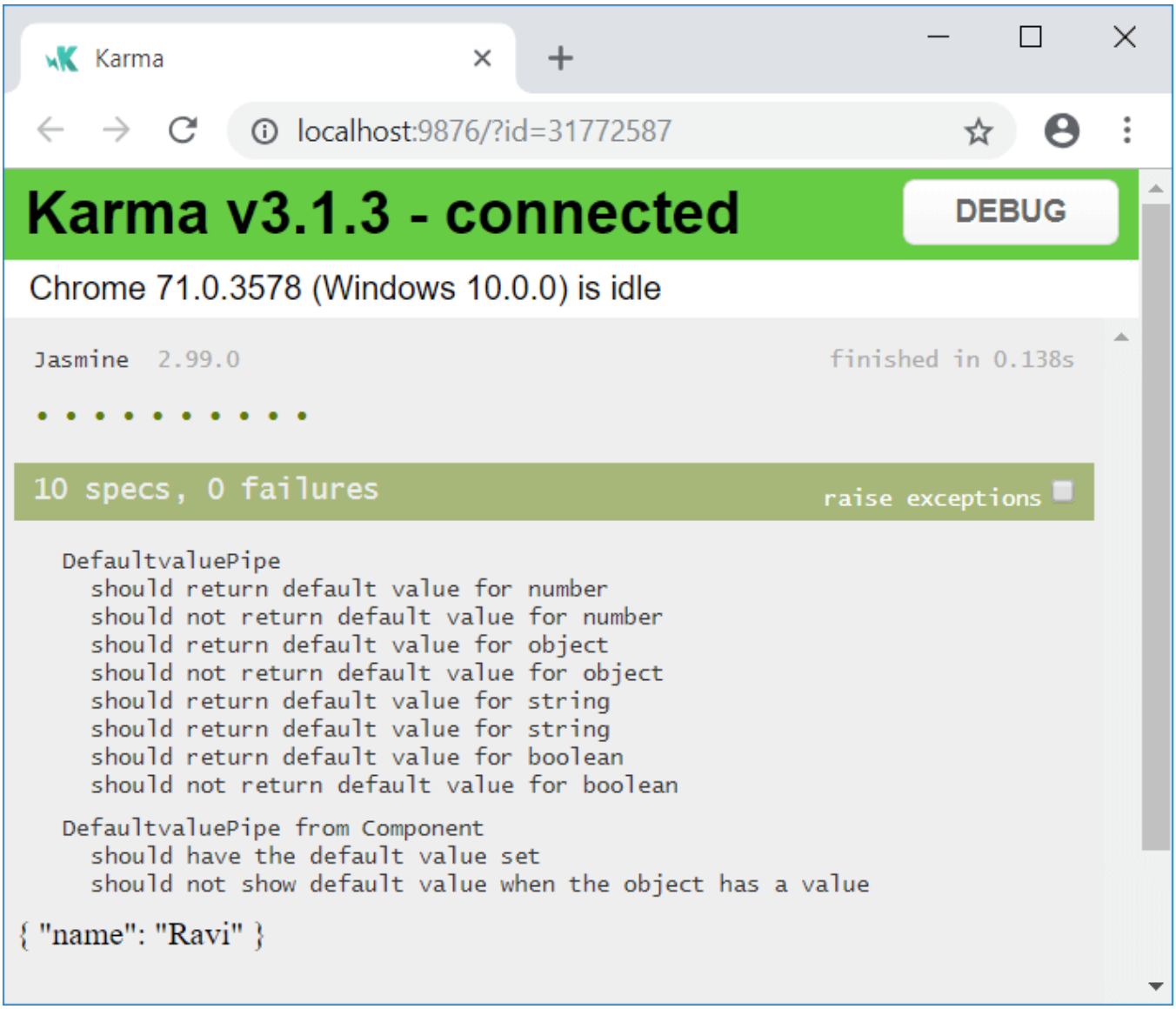


Figure 4 – Test results of defaultvalue Pipe

Conclusion

Unit testing is an important aspect of testing applications as it ensures quality of code along with reducing

possible bugs.

As shown in this tutorial, Angular makes the process of testing easier by isolating each block of code from the other, and by providing a number of helper APIs to work with the framework while testing.

Directives handle some of the vital aspects of an application that are critical to the business. Hope this tutorial got you started with testing different aspects of the directives and pipes. The future articles will continue this discussion and cover testing other parts of an Angular application.

Download the entire source code from GitHub at
bit.ly/dncm40-unittest-directives

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.

Want this magazine delivered to your inbox ?

Subscribe here

www.dotnetcurry.com/magazine/

* No spam policy

Riccardo Terrell

PARALLEL WORKFLOW WITH THE .NET TASK PARALLEL LIBRARY (TPL) DATAFLOW

Today's global market requires that businesses and industries be agile enough to respond to a constant flow of changing data. These workflows that are designed to process such streams of data are frequently large, and sometimes infinite or unknown in size.

Often, the data requires complex processing, making it a challenge to meet high throughput demands and potentially immense computational loads. To cope with these requirements, the key is to use ***parallelism*** *to exploit system resources and multiple cores.*

When designing a reactive application, it's fundamental to build and treat the system components as units of work in order to take full advantage of asynchronous execution. This design aims to exploit asynchronous message passing for the interactions and communication within the reactive components. In fact, in reactive systems, each component should embrace an asynchronous non-blocking development model for non-blocking I/O.

Reactive programming is a set of design principles used in asynchronous programming to create cohesive systems that respond to commands and requests in a timely manner.

Reactive programming is a way of thinking about systems architecture and design in a distributed environment where implementation techniques, tooling, and design patterns are components of a larger whole—a system. For more information check the online Reactive Manifesto (www.reactivemanifesto.org).

These units react to messages, which are propagated by other components in the chain of processing.

The reactive programming model emphasizes a push-based model for applications, rather than a pull-based model. This push-based strategy ensures that the individual components are easy to test and link, and, most importantly, easy to understand (for further information check this online documentation http://introtorx.com).

The **.NET Task Parallel Library DataFlow (TPL DataFlow)** helps to tackle the complexity of developing modern systems with an API that builds on Task-based Asynchronous Pattern (TAP). The TPL DataFlow fully supports asynchronous processing, in combination with a powerful compositionality semantic and a better configuration mechanism, than the TPL.

The TPL DataFlow library eases concurrent processing and implements tailored asynchronous parallel workflow and batch queuing. Furthermore, it facilitates the implementation of sophisticated patterns based on combining multiple components that talk to each other by passing messages.

THE POWER OF TPL DATAFLOW

Let's say you're building a sophisticated producer-consumer pattern that must support multiple producers and/or multiple consumers in parallel, or perhaps it has to support workflows that can scale the different steps of the process independently.

One solution is to exploit Microsoft TPL DataFlow.

Since the release of .NET 4.5, Microsoft introduced TPL Dataflow as part of the tool set for writing concurrent applications. The TPL Dataflow library is designed with the higher-level constructs necessary to tackle easy parallel problems, while providing a simple to use and powerful framework for building asynchronous data processing pipelines.

The TPL DataFlow isn't distributed as part of the .NET 4.5 framework, so to access its API and classes, you need to import the official Microsoft NuGet Package (`install-Package Microsoft.Tpl.DataFlow`).

The TPL DataFlow offers a rich array of components (also called blocks) for composing data-flow and pipeline infrastructures based on the in-process message passing semantic. This dataflow model promotes actor-based programming (https://en.wikipedia.org/wiki/Actor_model) by providing in-process message passing for coarse-grained dataflow and pipelining tasks.

The TPL DataFlow uses the task scheduler (TaskScheduler) of the TPL to efficiently manage the underlying threads and to support the TAP model (async/await) for optimized resource utilization. The TPL DataFlow

library increases the robustness of highly concurrent applications and achieves better performance, compared to the sequential version of the code, for parallelizing CPU and I/O intensive operations which have high throughput and low latency.

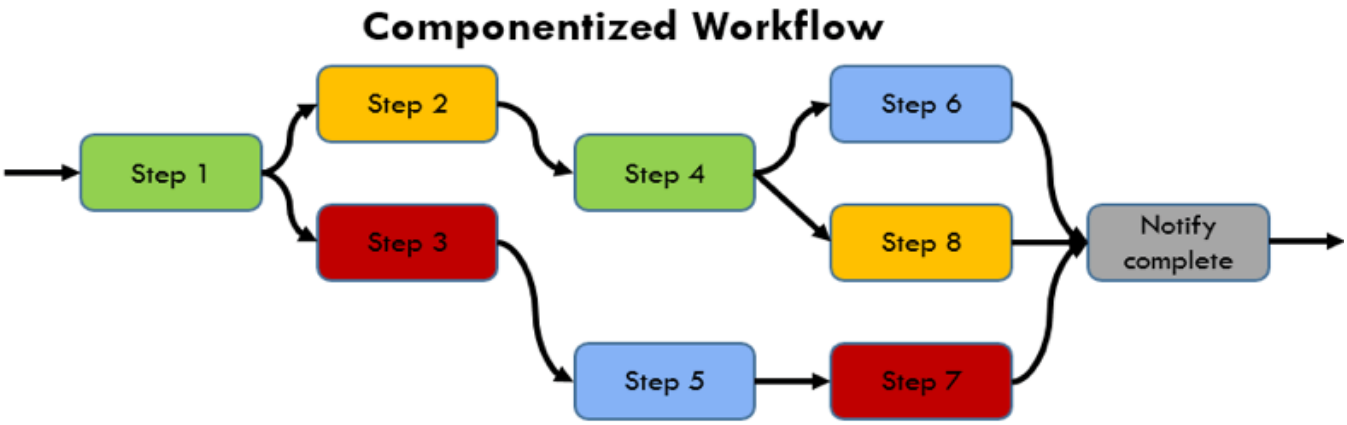


Figure 1: Workflow composed by multiple steps. Each operation can be treated as an independent computation.

The concept behind the TPL DataFlow library is to ease the creation of multiple patterns, such as:

- batch-processing pipelines
- parallel stream processing
- data buffering, or joining and
- processing in batch data from one or more sources.

Each of these patterns can be used as a standalone, or may be composed with other patterns, enabling developers to easily express complex dataflows.

DESIGNED TO COMPOSE: TPL DATAFLOW BLOCKS

Imagine you're implementing a complex workflow process composed of many different steps, such as a stock analysis pipeline.

It's ideal to split the computation in blocks, developing each block independently and then gluing them together. Making these blocks reusable and interchangeable enhances their convenience. This composable design would simplify the implementation of complex and convoluted systems.

Compositionality is the main strength of TPL DataFlow, because a set of independent containers, known as blocks, is designed to be combined. These blocks can be a chain of different tasks that constitute a parallel workflow, and are easily swapped, reordered, reused, or even removed.

The TPL DataFlow emphasizes a component's architectural approach to ease the restructure of the design. These dataflow components are useful when you have multiple operations that must communicate with one another asynchronously or when you want to process data as it becomes available.

Here's a high-level view of how the TPL Dataflow blocks operate:

1. Each block receives and buffers data from one or more sources, including other blocks, in the form of

messages. When a message is received, the block reacts by applying its behavior to the input, which then can be transformed and/or used to perform side effects.

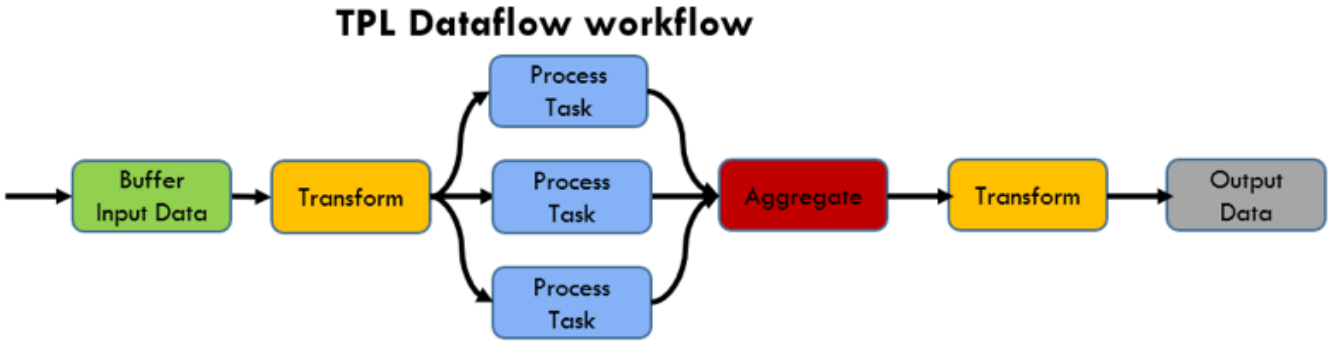


Figure 2: The TPL DataFlow embraces the concepts of reusable components. In this figure, each step of the workflow acts as reusable components. The TPL DataFlow brings few core primitives that allow you to express computations based on DataFlow graphs.

2. The output from the component (*block*) is then passed to the next linked block, and to the next one, if any, and so on, until a pipeline has completed to execute all the steps of the workflow.

The TPL DataFlow excels at providing a set of configurable properties by which it's possible, with small changes, to control the level of parallelism, the buffer size of the mailbox, and enable support for cancellation.

There are three main types of DataFlow blocks:

- *Source*—Operates as producer of data. It can also be read from.
- *Target*—Acts as a consumer, which receives the data and can be written to.
- *Propagator*—Acts as both a Source and a Target block.

The TPL DataFlow provides a set of blocks that inherit from one or more of these main DataFlow block types, each with a different purpose. It's impossible to cover all the blocks in one article. Thus, in the following sections we focus on the most common and versatile ones to adopt in general pipeline composition applications.

Note: *TPL Dataflow's most commonly used blocks are the standard BufferBlock, ActionBlock, and TransformBlock. Each is based on a delegate, which can be in the form of an anonymous function that defines the work to compute. I recommend that you keep these anonymous methods short, simple to follow, and easy to maintain.*

USING THE BUFFERBLOCK<TINPUT> AS A FIFO BUFFER

The TPL DataFlow `BufferBlock<T>` acts as a buffer for data. Such data is stored in a first in, first out (FIFO) order. In general, the `BufferBlock` is a great tool for enabling and implementing asynchronous Producer/Consumer patterns, where the internal message queue can be written to by multiple sources, or read from multiple targets.

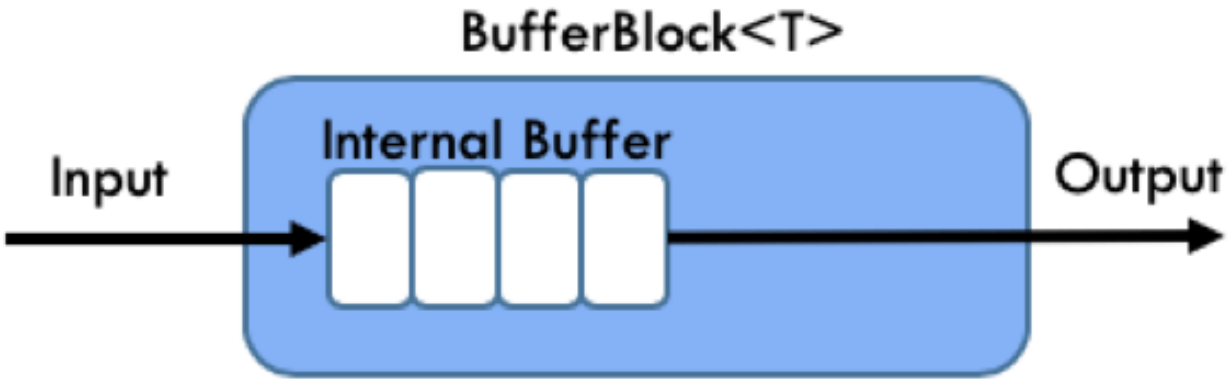


Figure 3: The TPL DataFlow `BufferBlock` has an internal buffer where the messages are queued, waiting to be pushed to the output, usually another TPL Dataflow block . The `Input` and `Output` are the same types, and this block doesn't apply any transformation on the data.

A MULTIPLE-PRODUCER/SINGLE-CONSUMER PATTERN

The producer-consumer pattern is one of the most widely used patterns in parallel programming. Developers use it to isolate work to be executed from the processing of that work.

In a typical producer/consumer pattern, at least two separated threads run concurrently: one thread produces and pushes the data to process into a queue, and another thread verifies the presence of the new incoming piece of data and processes it. The queue that holds the tasks is shared among these threads, which requires care for accessing tasks safely.

The TPL DataFlow is a great tool for implementing this pattern, because it has intrinsic support for multiple readers and multiple writers concurrently, and it encourages a pipeline pattern of programming, with producers sending messages to decoupled consumers

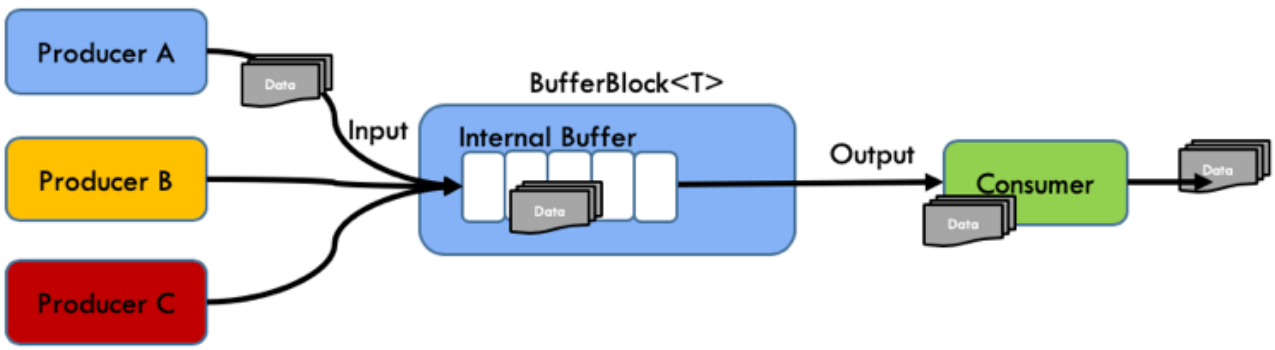


Figure 4: Multiple-producers/one-consumer pattern using the TPL DataFlow BufferBlock, which can manage and throttle the pressure of multiple producers.

In the case of a multi-producers/single-consumer pattern, it is important to enforce a restriction between the numbers of items generated and the number of items consumed. This constraint aims to balance the work between the producers when the consumer cannot handle the load. This technique is called *throttling*.

Throttling protects the program from running out of memory if the producers are faster than the consumer. Fortunately, the TPL DataFlow has built-in support for throttling, which is achieved by setting the maximum size of the buffer through the property `BoundedCapacity`, part of the `DataFlowBlockOptions`.

In the following listing, this property ensures that there will never be more than 10 items in the `BufferBlock` queue. Furthermore, in combination with enforcing the limit of the buffer size, it is important to use the function `SendAsync`, which waits intelligently and without blocking for the buffer to have available space to place a new item.

Listing 1: Asynchronous multi producer/consumer based on the TPL DataFlow BufferBlock

```
BufferBlock<int> buffer = new BufferBlock<int>(
  // Set the BoundedCapacity to be able to manage
  // and throttle the pressure from multiple producers
  new DataFlowBlockOptions { BoundedCapacity = 10 });

async Task Produce(IEnumerable<int> values)
{
  foreach (var value in values)
  // Send the message to buffer block asynchronously.
  // The SendAsync method helps to throttle the messages sent
  await buffer.SendAsync(value);;
}

async Task MultipleProducers(params IEnumerable<int>[] producers)
{
  // Running multiple producers in parallel waiting
  // all to terminate before notify the buffer block to complete
  await Task.WhenAll(
    from values in producers select Produce(values).ToArray())
    .ContinueWith(_ => buffer.Complete());
}

async Task Consumer(Action<int> process)
{
  // Safeguard the buffer block from receiving a message
  // only if there are any items available in the queue
  while (await buffer.OutputAvailableAsync())
  process(await buffer.ReceiveAsync());
}

async Task Run() {
  IEnumerable<int> range = Enumerable.Range(0, 100);
  await Task.WhenAll(MultipleProducers(range, range, range),
  Consumer(n => Console.WriteLine($"value {n} - ThreadId
  {Thread.CurrentThread.ManagedThreadId}")));
}
```

By default, the TPL DataFlow blocks have the value `DataFlowBlockOptions.Unbounded` set to -1, which means that the queue is unbounded (unlimited) to the number of messages. However, you can reset this value to a specific capacity that limits the number of messages the block may be queueing.

When the queue reaches maximum capacity, any additional incoming messages will be postponed for later processing making the producer await before further work. Likely, making the producer slowdown (or await) is not a problem because the messages are sent asynchronously.

TRANSFORMING DATA WITH THE TRANSFORMBLOCK<TINPUT, TOUTPUT>

The TPL Dataflow `TransformBlock<TInput,TOutput>` acts like a mapping function, which applies a projection function to an input value and provides a correlated output.

The transformation function is passed as an argument in the form of a delegate `Func<TInput,TOutput>`, which is generally expressed as a lambda expression. This block's default behavior is to process one message at a time, maintaining strict FIFO ordering.

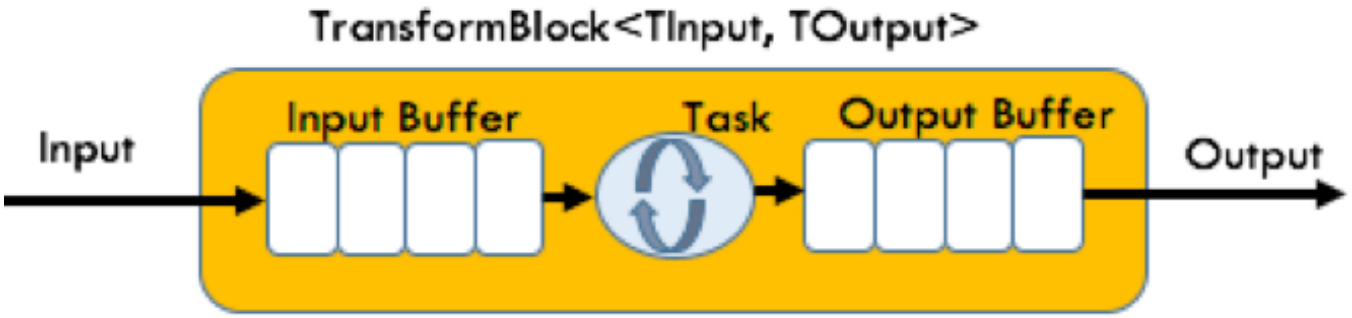


Figure 5: The TPL DataFlow `TransformBlock` has an internal buffer for both the input and output values; this type of block has the same buffer capabilities of the `BufferBlock`. The purpose of this block is to apply a transformation function on the data; the `Input` and `Output` are likely different types.

Note that `TransformBlock<TInput,TOutput>` performs as the `BufferBlock<TOutput>`, which buffers both the input and output values. The underlying delegate can run synchronously or asynchronously.

The asynchronous version takes a delegate with signature `Func<TInput,Task<TOutput>>` whose purpose is to run the underlying function asynchronously. The block treats the processing of that element completed when the returned `Task` completes. Listing 2 shows how to use the `TransformBlock` type.

Listing 2: Downloading images using the TPL Dataflow `TransformBlock`

```
var fetchImageFlag = new TransformBlock<string, (string, byte[])>(
  async urlImage => {
    using (var webClient = new WebClient()) {
      byte[] data = await webClient.DownloadDataTaskAsync(urlImage);
       Console.WriteLine($"The image {Path.GetFileName(urlImage)} has a size
       of {data.Length} bytes");
      return (urlImage, data);
    }
});

List<string> urlFlags = new List<string>{
  "Italy#/media/File:Flag_of_Italy.svg",
  "Spain#/media/File:Flag_of_Spain.svg",
  "United_States#/media/File:Flag_of_the_United_States.svg"
};

foreach (var urlFlag in urlFlags)
  fetchImageFlag.Post($"https://en.wikipedia.org/wiki/{urlFlag}");
```

In Listing 2, the `TransformBlock<string,(string, byte[])> fetchImageFlag` block fetches the flag image into a tuple of string and byte array format `(urlImage, data)`. In this case, the output isn't consumed anywhere, so the code isn't too useful. You need another block to process the outcome in a meaningful way.

COMPLETING THE WORK WITH ACTIONBLOCK<TINPUT >

The TPL Dataflow `ActionBlock` executes a given callback for any item sent to it. You can think of this block logically as a buffer for data combined with a task for processing that data. `ActionBlock<TInput>` is a target block that calls a delegate when it receives data, similar to foreach loop.

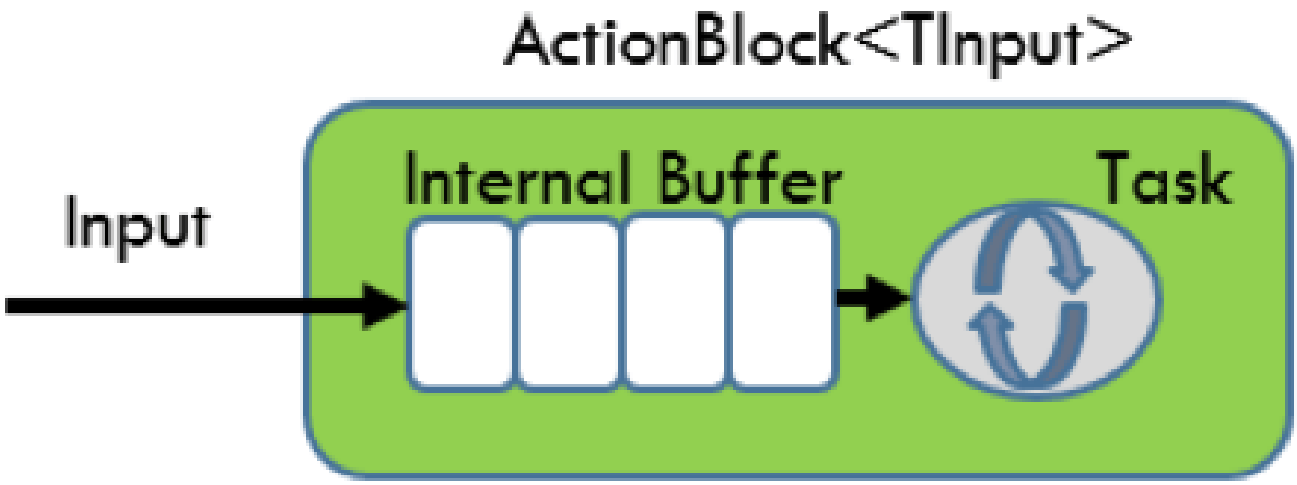


Figure 6: The TPL DataFlow `ActionBlock` has an internal buffer for input messages that are queued if the `Task` is busy processing another message. This type of block has the same buffer capabilities of the `BufferBlock`. The purpose of this block is to apply an action that completes the workflow without output and that likely produces side effects. In general, because the `ActionBlock` doesn't have an output, it cannot compose to a following block, so it's used to terminate the workflow.

`ActionBlock<TInput>` is usually the last step in a TPL Dataflow pipeline; in fact, it doesn't produce any output. This design prevents the `ActionBlock` from being combined with further blocks, making it the perfect candidate to terminate the workflow process. For this reason, the `ActionBlock` is likely to produce side effects as a final step to complete the pipeline processing.

Listing 3 shows the `TransformBlock` from Listing 2 pushing its outputs to the `ActionBlock` to persist the flag images in the local file system.

Listing 3: Persisting data using the TPL Dataflow `ActionBlock`

```
var saveData = new ActionBlock<(string, byte[])>(async data => {
  (string urlImage, byte[] image) = data;
  string filePath = urlImage.Substring(urlImage.IndexOf("File:") + 5);
  await File.WriteAllBytesAsync(filePath, image);
});

// Links the output from the TransformBlock to the ActionBlock
fetchImageFlag.LinkTo(saveData);
```

NOTE: the code in this article runs on .NET Core 2.0 (or higher). The method File.WriteAllBytesAsync, in the previus code listing, is part of the .NET Core.

The argument passed into the constructor during the instantiation of the `ActionBlock` block can be either a delegate `Action<TInput>` or `Func<TInput,Task>`. The latter performs the internal action (behavior) asynchronously for each message input (received). Note that the `ActionBlock` has an internal buffer for

the incoming data to be processed, which works exactly like the `BufferBlock.`

It's important to remember that the `ActionBlock` saveData is linked to the previous `TransformBlock` `fetchImageFlag` using the `LinkTo` extension method. In this way, the output produced by the `TransformBlock` is pushed to the `ActionBlock` as soon as available.

LINKING DATAFLOW BLOCKS

The TPL Dataflow blocks can be linked with the help of the `LinkTo` extension method.

Linking DataFlow blocks is a powerful technique for automatically transmitting the result of each computation to the connected blocks in a message-passing manner. The key component for building sophisticated pipelines in a declarative manner is to use connecting blocks.

BUFFERING AND PARALLELIZING MESSAGE PROCESSING

By default, the TPL Dataflow block processes only one message at a time, while buffering the other incoming messages until the previous one completes. Each block is independent of others, so one block can process one item while another block processes a different item.

However, when constructing the block, you can change this behavior by setting the `MaxDegreeOfParallelism` property in the `DataFlowBlockOptions` to a value higher than 1 (this property is used later in the article in a program example). You can use the TPL Dataflow to speed up the computations by specifying the number of messages that can be processed in parallel. The internals of the class handle the rest, including the ordering of the data sequence.

REACTIVE EXTENSIONS (RX) AND TPL DATAFLOW MESHES

The TPL Dataflow and .NET Reactive Extensions (Rx) have important similarities, despite having independent characteristics and strengths. In fact, these libraries complement each other, making them easy to integrate.

The TPL Dataflow is closer to an agent-based programming model, focused on providing building blocks for message passing, which simplifies the implementation of parallel CPU- and I/O-intensive applications with high throughput and low latency, while also providing developers explicit control over how data is buffered. Rx is keener to the functional paradigm, providing a vast set of operators that predominantly focus on

AN AGENT IS NOT AN ACTOR

On the surface, there are some similarities between agents and actors, which sometimes cause people to use these terms interchangeably. But, the main difference is that agents are ***in*** a process, while actors may be running ***on*** another process. In fact, the reference to an agent is a pointer to a specific instance, whereas an actor reference is determined through location transparency. ***Location transparency*** is the use of names to identify network resources, rather than their actual location, which means that the actor may be running in the same process, or on another process, or possibly on a remote machine.

Agent-based concurrency is inspired by the actor model, but, its construction is much simpler than the actor model. Actor systems have built-in sophisticated tools for distribution support, which include supervision to manage exceptions and potentially self-heal the system, routing to customize the work distribution and more.

coordination and composition of event streams with a LINQ-based API.

The TPL Dataflow has built-in support for integrating with Rx, which allows it to expose the source DataFlow blocks as both observables and observers (https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern).

The `AsObservable` extension method transforms any TPL Dataflow *Propagator* block into an observable sequence, which allows the output of the DataFlow chain to flow efficiently into an arbitrary set of Reactive fluent extension methods for further processing. Specifically, the `AsObservable` extension method constructs an `IObservable<T>` for an `ISourceBlock<T>`.

Note: *TPL Dataflow can also act as an observer. The `AsObserver` extension method creates an `IObserver<T>` for an `ITargetBlock<T>`, where the `OnNext` calls for the observer result in the data being sent to the target. The `OnError` calls result in the exception faulting the target, and the `OnCompleted` calls will result in `Complete` called on the target.*

Let's see in the following example how to integrate Rx and TPL Dataflow.

A PARALLEL WORKFLOW TO COMPRESS, ENCRYPT A LARGE STREAM

In this section, you'll build a complete asynchronous and parallelized workflow to demonstrate the power of the TPL Dataflow library.

ASYNCHRONY VS. PARALLELISM

Parallelism is primarily about application performance, and it facilitates CPU-intensive work on multiple threads at one time, taking advantage of modern multicore computer architectures. Asynchrony is a subset of concurrency, which focuses on I/O-bound rather than CPU-bound operations. Asynchronous programming addresses the issue of latency (Latency is anything that takes long time to run).

This example uses a combination of TPL Dataflow blocks and Reactive Extensions work as a parallel pipeline. The purpose of this example is to analyze and architect a real case application. It then evaluates the challenges encountered during the development of the program, and examines how TPL Dataflow can be introduced in the design of the application.

The goal of this application is to leverage the TPL Dataflow library to efficiently spread the work out across multiple CPU cores to maximize the speed of computation and overall scalability, especially when the blocks that compose a workflow, process the messages at different rates and in parallel. This is particularly useful when you need to process a large stream of bytes that could generate hundreds, or even thousands, of chunks of data.

CASE STUDY: THE PROBLEM OF PROCESSING A LARGE STREAM OF DATA

Let's say that you need to compress a large file to make it easier to persist or transmit over the network, or that a file's content must be encrypted to protect that information.

Often, both compression and encryption must be applied.

These operations can take a long time to complete if the full file is processed all at once. Furthermore, it's challenging to move a file, or stream data, across the network, and the complexity increases with the size of the file, due to external factors, such as latency or unpredictable bandwidth.

In addition, if the file is transferred in one transaction, and something goes wrong, then the operation tries to resend the entire file, which can be time and resource consuming.

In the following sections, you'll tackle this problem step-by-step.

In .NET, it isn't easy to compress a file larger than 4 GB, due to the framework limitation on the size of data to compress. Due to the maximum addressable size for a 32-bit pointer, if you create an array over 4 GB, an `OutOfMemory` exception is thrown.

Starting with .NET 4.5 and for 64-bit platforms, the option gcAllowVeryLargeObjects is available to enable arrays greater than 2 GB. This option allows 64-bit applications to have a multi-dimensional array with size `UInt32.MaxValue` (4,294,967,295) elements. Technically, you can apply the standard GZip compression that's used to compress streams of bytes to data larger than 4 GB; but the GZip distribution doesn't support this by default. The related .NET `GZipStream` class inheritably has a 4 GB limitation.

HOW CAN YOU COMPRESS AND ENCRYPT A LARGE FILE WITHOUT BEING CONSTRAINED BY THE 4 GB LIMIT IMPOSED BY THE FRAMEWORK CLASSES?

A practical solution involves using a chunking routine to chop the stream of data.

Chopping the stream of data makes it easier to compress and/or encrypt each block individually and ultimately write the block content to an output stream. The chunking technique splits the data, generally into chunks of the same size, applies the appropriate transformation to each chunk (compression before encryption), and glues the chunks together in the correct order.

It's vital and good practice to guarantee the correct order of the chunks upon reassembly at the end of workflow. Due to the intensive I/O asynchronous operations (compress and encrypt), the packages might not arrive in the correct sequence, especially if the data is transferred across the network. You must verify order during reassembly.

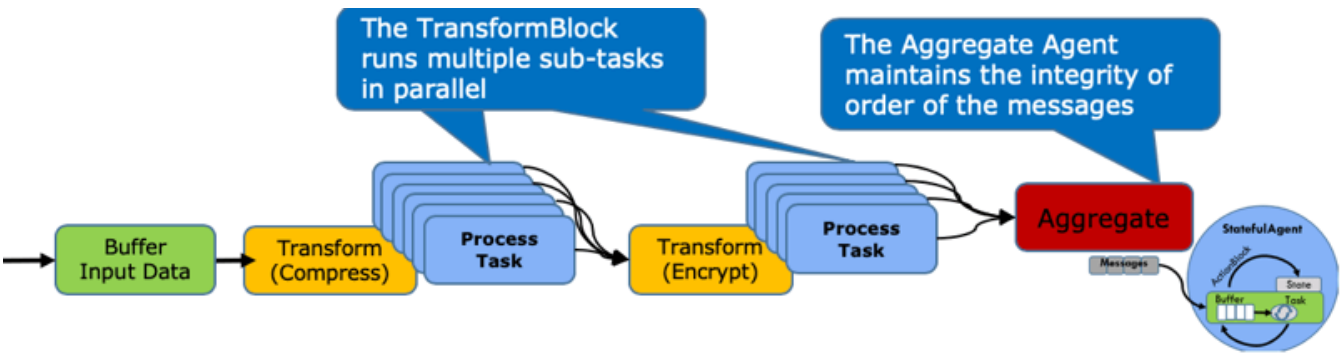


Figure 7: The `Transform` blocks process the messages in parallel. The result is sent to the next block when the operation completes. The aggregate agent's purpose is to maintain the integrity of the order of the messages, similar to the `AsOrdered` PLINQ extension method.

The opportunity for parallelism fits naturally in this design, because the chunks of the data can be processed independently.

ENCRYPTION AND COMPRESSION: ORDER MATTERS

It might seem that because the compression and encryption operations are independent of one another, it makes no difference in which order they're applied to a file. This isn't true. *The order in which the operations of compression and encryption are applied, is vital.* Encryption has the effect of turning input data into high-entropy data (Information entropy is defined as the average amount of information produced by a stochastic source of data. See https://en.wikipedia.org/wiki/Entropy_(information_theory), which is a measure of the unpredictability of information content. Therefore, the encrypted data appears like a random array of bytes, which makes finding common patterns less probable. Conversely, compression algorithms work best when there are several similar patterns in the data, which can be expressed with fewer bytes.

When data must be both compressed and encrypted, you achieve the best results by first compressing and then encrypting the data. In this way, the compression algorithm can find similar patterns to shrink, and consequently the encryption algorithm produces the chunks of data having almost the same size of the compressed ones. Furthermore, if the order of the operations is compression and then encryption, not only should the output be a smaller file, but the encryption will most likely take less time because it'll operate on less data.

Listing 4 shows the full implementation of the parallel compression–encryption workflow. Note that in the source code, you can find the reverse workflow to decrypt and decompress the data, as well as use asynchronous helper functions for compressing and encrypting bytes array.

***Note**: **the full source code of this article** is available for download over here. .NET Core 2.0 is a prerequisite.*

The function `CompressAndEncrypt` takes as an argument the source and destination streams to process, the `chunkSize` argument defines the size in which the data is split (the default is 1 MB if no value is provided), and `CancellationTokenSource` stops the DataFlow execution at any point. If no `CancellationTokenSource` is provided, a new token is defined and propagated through the DataFlow operations.

The core of the function consists of three TPL Dataflow building blocks, in combination with a Reactive Extensions operator (`Scan`) that completes the workflow. The `inputBuffer` is a `BufferBlock` type that, as the name implies, buffers the incoming chunks of bytes read from the source stream, and holds these items to pass them to the next block in the flow, which is the linked `TransformBlock compressor`.

Listing 4: Parallel stream compression and encryption using TPL Dataflow

```
async Task CompressAndEncrypt(
  Stream streamSource, Stream streamDestination,
  long chunkSize = 1048576, CancellationTokenSource cts = null)
{
  cts = cts ?? new CancellationTokenSource();

  // Sets the BoundedCapacity value to throttle the messages
  var compressorOptions = new ExecutionDataflowBlockOptions
  {
    MaxDegreeOfParallelism = Environment.ProcessorCount,
    BoundedCapacity = 20,
    CancellationToken = cts.Token
```

```
};

var inputBuffer = new BufferBlock<CompressingDetails>(
  new DataflowBlockOptions
  {
    CancellationToken = cts.Token,
    BoundedCapacity = 20
  });

var compressor = new TransformBlock<CompressingDetails,
  CompressedDetails>(async details => {
  // Compresses asynchronously the data
  // (the method is provided in the source code)
  var compressedData = await IOUtils.Compress(details.Bytes);

  // Converts the current data structure into the CompressionDetails
  // message type that is sent to the next block
  return details.ToCompressedDetails(compressedData);
  }, compressorOptions);

  var encryptor = new TransformBlock<CompressedDetails, EncryptDetails>(
    async details => {

    // Combines the data and metadata into a byte array pattern
    // that will be deconstructed and parsed during the reverse
    // operation decrypt-decompress
    byte[] data = IOUtils.CombineByteArrays(details.CompressedDataSize,
    details.ChunkSize, details.Bytes);
    // Encrypts asynchronously the data
    // (the method is provided in the source code)
    var encryptedData = await IOUtils.Encrypt(data);
      return details.ToEncryptDetails(encryptedData);
    }, compressorOptions);

  // Enables Rx integration with TPL Dataflow
  encryptor.AsObservable()
  .Scan((new Dictionary<int, EncryptDetails>(), 0),
  (state, msg) =>
  // Runs Rx Scan operation asynchronously
  Observable.FromAsync(async () => {
    (Dictionary<int, EncryptDetails> details, int lastIndexProc) = state;
    details.Add(msg.Sequence, msg);
    // Guarantee to process any chunk of data available that are
    // sequentially in correct order and succession
    while (details.ContainsKey(lastIndexProc + 1))
      {
        msg = details[lastIndexProc + 1];

        // Persists asynchronously the data; the stream could be replaced
        // with a network stream to send the data across the wire.
        await streamDestination.WriteAsync(msg.EncryptedDataSize, 0,
          msg.EncryptedDataSize.Length);
        await streamDestination.WriteAsync(msg.Bytes, 0, msg.Bytes.Length);
          lastIndexProc = msg.Sequence;

        // the chunk of data that is processed is removed
        // from the local state, keeping track of the items to perform
        details.Remove(lastIndexProc);
      }
      return (details, lastIndexProc);
```

```
      })
      .SingleAsync())
      // Rx subscribes to TaskPoolScheduler
      .SubscribeOn(TaskPoolScheduler.Default).Subscribe();

    var linkOptions = new DataflowLinkOptions { PropagateCompletion = true };
    // Links the DataFlow blocks to compose workflow
    inputBuffer.LinkTo(compressor, linkOptions);
    compressor.LinkTo(encryptor, linkOptions);

    long sourceLength = streamSource.Length;
    byte[] size = BitConverter.GetBytes(sourceLength);
    // The total size of the file stream is persisted as the first chunk of data;
    // in this way, the decompression algorithm knows how to retrieve
    // the information and how long to run.
    await streamDestination.WriteAsync(size, 0, size.Length);

    // Determines the chunk size to partition data
    chunkSize = Math.Min(chunkSize, sourceLength);
    int indexSequence = 0;
    // Reads the source stream into chunks until the end of the stream
    while (sourceLength > 0)
    {
      byte[] data = new byte[chunkSize];
      int readCount = await streamSource.ReadAsync(data, 0, data.Length);
      byte[] bytes = new byte[readCount];
      Buffer.BlockCopy(data, 0, bytes, 0, readCount);
      var compressingDetails = new CompressingDetails
      {
        Bytes = bytes,
        ChunkSize = BitConverter.GetBytes(readCount),
        Sequence = ++indexSequence
      };
      await inputBuffer.SendAsync(compressingDetails);
      sourceLength -= readCount;
      if (sourceLength < chunkSize)
        chunkSize = sourceLength;
      // Checks the current source stream position
      // after each read operation to decide when to complete the operation
      if (sourceLength == 0)
        inputBuffer.Complete();
    }
    await encryptor.Completion;
    await streamDestination.FlushAsync();
}
```

The bytes read from the stream are sent to the buffer block by using the `SendAsync` method:

```
var compressingDetails = new CompressingDetails {
  Bytes = bytes,
  ChunkSize = BitConverter.GetBytes(chunkSize),
  Sequence = ++indexSequence
};

await buffer.SendAsync(compressingDetails);
```

Each chunk of bytes read from the streamSource is wrapped into the `CompressingDetails` object , which contains additional information of byte array size.

The monotonic value is later used in the sequence of chunks generated to preserve the order. A *monotonic value* is a function between ordered sets that preserves or reverses the given value, and the value always either decreases or increases. The order of the block is important both for a correct compression–encryption operation and for correct decryption and decompression into the original shape.
In general, if the purpose of a TPL Dataflow block is purely to forward items from one block to several others, then you don't need the `BufferBlock`. But in the case of reading a large or continuous stream of data, this block is useful for taming the backpressure generated from the massive amount of data given to the process by setting an appropriate `BoundedCapacity`.

In this example, the `BoundedCapacity` is restricted to a capacity of 20 items. When there are 20 items in this block, it will stop accepting new items until one of the existing items passes to the next block. Because the DataFlow source of data originated from asynchronous I/O operations, there's a risk of potentially large amounts of data to process. It's recommended that you limit the internal buffering to throttle the data by setting the `BoundedCapacity` property in the options defined when constructing the `BufferBlock`.

The next two operation types are compression transformation and encryption transformation.

During the first phase (compression), the `TransformBlock` applies the compression to the chunk of bytes and enriches the message received `CompressingDetails` with the relative data information, which includes the compressed byte array and its size. This information persists as part of the output stream which is needed later during the decompression.

The second phase (encryption) enciphers the chunk of compressed byte array and creates a sequence of bytes resulting from the composition of three arrays: `CompressedDataSize`, `ChunkSize`, and data array. This structure instructs the decompression and decryption algorithms to target the right portion of bytes to revert from the stream.

NOTE: *Keep in mind that when there are multiple TPL Dataflow blocks, certain Tasks may be idle while the others are executing, so you have to tune the block's execution option to avoid potential starvation. Details regarding this optimization are explained in the coming section.*

ENSURING THE ORDER INTEGRITY OF A STREAM OF MESSAGES

The TPL Dataflow documentation guarantees that the `TransformBlock` will propagate the messages in the same order in which they arrived.

Internally, the `TransformBlock` uses a reordering buffer to fix any out-of-order issues that might arise from processing multiple messages concurrently. Unfortunately, due to the high number of asynchronous and I/O intensive operations running in parallel, keeping the integrity of the message order doesn't apply to your case. This is why you implemented the additional sequential ordering preservation using monotonically values.

If you decide to *send* or *stream* the data over the network, then the guarantee of delivering the packages in the correct sequence is lost, due to variables such as the unpredictable bandwidth and unreliable network connection. In addition, because these packages of data are processed (compressed and encrypted) in parallel and sent asynchronously through the network independently, the order by which these chucks of data are sent won't necessarily match the order of arrived to destination.

To safeguard the order integrity when processing chunks of data, the final step in the workflow is leveraging Reactive Extensions with the `Observable.Scan` operator. This operator behaves as a

multiplexer by reassembling the items and persists them in the local file system, maintaining the correct sequence. The order of the sequence is kept in a property of the `EncryptDetails` data structure, which is passed into the Scan operation. The particularity of this component is the presence of an internal state that keeps track of the messages received and their order.

NOTE: *To complete successfully the workflow in the previous example, we need keep the order of which the chunks are processed to be able to re-compose them correctly. For this reason, we are using Reactive Extensions, with the function Scan, to maintain the state of the exact order of the messages. This would not be possible using only the TPL Dataflow blocks, because are stateless per nature.*

THE MULTIPLEXER PATTERN

The multiplexer is a pattern generally used in combination with a Producer/Consumer design. It allows the consumer, which in the previous example is the last stage of the pipeline, to receive the chunks of data in the correct sequential order. The chunks of data don't need to be sorted or reordered. Instead, the fact that each producer (TPL DataFlow block) queue is locally ordered allows the multiplexer to look for the next value (message) in the sequence. The multiplexer waits for a message from a producer DataFlow block. When a chunk of data arrives, the multiplexer looks to see if the chunk's sequence number is the next in the expected sequence. If it is, the multiplexer persists the data to the local file system. If the chunk of data isn't the one expected next in the sequence, the multiplexer holds the value in an internal buffer (in the previous example is used a collection of `Dictionary<int,EncryptDetails>`) and repeats the analysis operation for the next message received. This algorithm allows the multiplexer to put together the inputs from the incoming producer message in a way that ensures sequential order without sorting the values.

The accuracy for the whole computation requires preservation of the order of the source sequence to ensure that the order is consistent at merge time.

The TPL Dataflow blocks and Rx observable streams can be completed successfully or with errors, and the `AsObservable` method will translate the block completion into the completion of the observable stream. But if the block faults with an exception, that exception will be wrapped in an `AggregateException` when it is passed to the observable stream. This is similar to how linked blocks propagate their faults.

NOTE: *In the case of sending the chunks of data over the network, the same strategy of writing data to the local file system is applicable by having a the Scan Rx function working as part of the receiver on the other side of the wire.*

The state of the `Observable.Scan` operator is preserved using a tuple. The first item of the tuple is a collection `Dictionary<int, EncryptDetails>`, where the key represents the sequence value of the original order by which the data was sent. The second item, `lastIndexProc`, is the index of the last item processed, which prevents reprocessing the same chunks of data more than once.

The body of `Observable.Scan` runs the `while` loop that uses this value `lastIndexProc` and makes sure that the processing of the chunks of data starts from the last item unprocessed. The loop continues to iterate until the order of the items is continuous, otherwise it breaks out from the loop and waits for the next message, which might complete the missing gap in the sequence.

LINKING, PROPAGATING, AND COMPLETING

The TPL Dataflow blocks in the compress-encrypt workflow are linked using the `LinkTo` extension method, which by default propagates only data (messages). But if the workflow is linear, as in this example, it's good practice to share information among the blocks through an automatic notification, such as when the work is terminated or eventual errors occur. This behavior is achieved by constructing the `LinkTo` method with a `DataFlowLinkOptions` optional argument and the `PropagateCompletion` property set to true. Here's the code from the previous example with this option built-in (in bold).

```
var linkOptions = new DataFlowLinkOptions { PropagateCompletion = true };
inputBuffer.LinkTo(compressor, linkOptions);
compressor.LinkTo(encryptor, linkOptions);
```

The `PropagateCompletion` optional property informs the DataFlow block to automatically propagate its results and exceptions to the next stage when it completes. This is accomplished by calling the `Complete` method to trigger the complete notification upon reaching the end of the stream:

```
if (sourceLength < chunkSize)
  chunkSize = sourceLength;
if (sourceLength == 0)
  buffer.Complete();
```

..and then all the DataFlow blocks are announced in cascade as a chain that the process is completed:

```
await encryptor..Completion;
```

Ultimately, you can run the code as follows:

```
using (var streamSource = new FileStream(sourceFile, FileMode.OpenOrCreate,
  FileAccess.Read, FileShare.None, useAsync: true))
using (var streamDestination = new FileStream(destinationFile,
  FileMode.Create, FileAccess.Write, FileShare.None, useAsync: true))
  await CompressAndEncrypt(streamSource, streamDestination);
```

Note that the code runs on .NET Core 2.0 (or higher).

The following table shows the benchmarks for compressing and encrypting different file sizes, including the inverted operation of decrypting and decompressing. The benchmark result is the average of each operation run three times.

BENCHMARKS FOR COMPRESSING AND ENCRYPTING DIFFERENT FILE SIZES

File size in GB	Degree of parallelism	Compress-encrypt time in seconds	Decrypt-decompress time in seconds
3	1	524.56	398.52
3	4	123.64	88.25
3	8	69.20	45.93
12	1	2249.12	1417.07
12	4	524.60	341.94
12	8	287.81	163.72

IN CONCLUSION

A system written using TPL DataFlow benefits from a multicore system because all the blocks that compose a workflow can run in parallel.

The TPL Dataflow enables effective techniques for running embarrassingly parallel problems, where many independent computations can be executed in parallel in an evident way.

The TPL Dataflow has built-in support for throttling and asynchrony, improving both the I/O-bound and CPU-bound operations. In particular, it provides the ability to build responsive client applications while still getting the benefits of massively parallel processing.

The TPL Dataflow can be used to parallelize the workflow to compress and encrypt a large stream of data by processing blocks at different rates.

The combination and integration with Rx and TPL Dataflow simplifies the implementation of parallel CPU and I/O-intensive applications, while also providing developers explicit control over how data is buffered.

Resources

For more information about the DataFlow library, see the online MSDN documentation.

Download the entire source code from GitHub at
bit.ly/dncm40-tpldataflow

Riccardo Terrell
Author

Riccardo is an information systems and technology professional and architect specializing in software & systems development. He has over 20 years' experience delivering cost-effective technology solutions in the competitive business environment. Riccardo is passionate about integrating advanced technology tools to increase internal efficiency, enhance work productivity, and reduce operating costs. He is a Microsoft Most Valuable Professional (MVP) who is active in the .NET, functional programming, and F# communities. Riccardo believes in multi-paradigm programming to maximize the power of code and is the author of "Functional Concurrency in .NET"; which, features how to develop highly-scalable systems in F# & C#.

Thanks to Yacoub Massad for partially reviewing this article.

Keerti Kotaru

INTRODUCTION TO REACTIVE FORMS

A form is an important aspect of any web application. Data input from the user, data validation and managing the model to save and update actions is key for most web applications.

One of the strengths of Angular is the number of features and simplicity it provides while working with Forms.

In this article, we will be taking advantage of Angular features to build forms. Consider the following use case.

"In a travel management application, consider a screen that accepts passenger details. Imagine the user is presented with forms and some validations are performed on form data input. If the data is valid, passenger model objects are created and further processed to save the information and generate an itinerary."

Angular provides the following two choices for creating forms.

- Reactive Forms
- Template-driven Forms

This article focuses on creating Reactive Forms. We will follow-up with a separate article in a different edition detailing template-driven forms and form validations.

REACTIVE FORMS

- Reactive forms are model driven, that is, a data object maintained in the component representing the form or the view (or template).
- Reactive forms are immutable. Every change to the form creates a new state of the form, maintaining integrity of the model (in the component).
- Reactive forms are synchronous
- This approach is easily unit testable as data or form state is predictable and can be easily accessed through the component.

To begin with, let's look at a very simple form with a single text field. Further examples in the article work with more fields.

Taking the travel management example we discussed as our use case, imagine a form with a coupon code. The User can key-in a personal coupon received over an email and apply it on the itinerary. See figure-1.

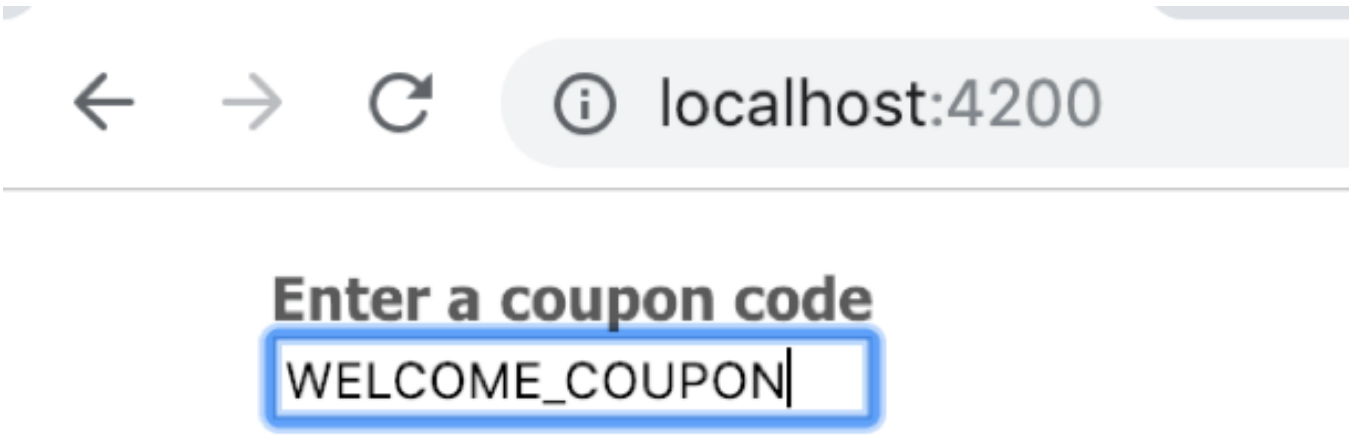


Figure 1 A very simple form with one field

STEP 1 - ANGULAR MODULE CHANGES

Reactive forms' related API (Angular service providers, directives etc.) is encapsulated in an Angular module of its own. Import it into the module where we are creating the forms.

Consider the code sample below. We are importing it into the main module.

```
// first import Angular module (ReactiveFormsModule) from the JavaScript module @
angular/forms
import { ReactiveFormsModule } from '@angular/forms';

// import to Angular module so that the services and dirctives can be accessed.
@NgModule({
  // Removing rest of the code in ngModule decorator for brevity
  imports: [
    ReactiveFormsModule,
  ],
})
export class AppModule { }
```

Let's create a new component for accepting *coupon code* input. If you are using the Angular CLI, use the following command to create the component.

```
ng generate component input-coupon
```

STEP 2- ANGULAR COMPONENT CHANGES

As mentioned in the earlier section, reactive forms are model driven. Model is part of the component class. It is bound to the component template. Rest of the section details creating a form control in the component class, as well as changes to the template.

There are multiple ways to create and manage the model in the component class. Let's begin by using `FormBuilder` service, which abstracts the nitty-gritty details of creating the form controls.

Import and inject form builder into the component class as shown in the following code snippet.

```
// import form builder from the JavaScript module
import { FormBuilder } from '@angular/forms';

@Component({
  // Removed code for brevity
})
export class InputCouponComponent implements OnInit {
  // Inject form builder
  constructor(private fb: FormBuilder) { }
...
```

Create a control representing the coupon text field in the component class. Use form builder's helper function control to create the object.

```
this.coupon = this.fb.control('Welcome Coupon');
```

The return object assigned to `this.coupon` is a FormControl object.

Note: *We can create an instance of* `FormControl` *with the traditional* `new` *keyword (see code snippet below). In the current example, using* `FormBuilder` *wouldn't add any additional value. However, as the forms gets complex, using the* `FormBuilder` *simplifies code. For consistency, always prefer using* `FormBuilder` *for Reactive Forms.*

```
this.coupon = new FormControl('Welcome form control');
```

Let's now update the component template. Use the directive `FormControlDirective` to associate class property `coupon` (instance of `FormControl`) to the text field. Consider the highlighted section in the code snippet

```
<div style="margin-left: 50px">
  <label for="couponCode">
    <h2>Enter a coupon code</h2>
    <input type="text" id="couponCode" [formControl]="coupon"
(change)="changeHandler()" />
  </label>
</div>
```

Also notice the change event handler. In a professional application, the change handler can verify the

coupon with a server-side API to calculate value of the coupon. To access the user input, use the `value` property on the coupon object. It is a snapshot of the value.

However, a form control object can stream value being modified by the user, in the text field. Consider `changeHandler` that prints user input on console. See figure-2 to see the printed value after the user has changed the text field value

```
changeHandler(){
  console.log("snapshot of coupon input,", this.coupon.value);
}
```

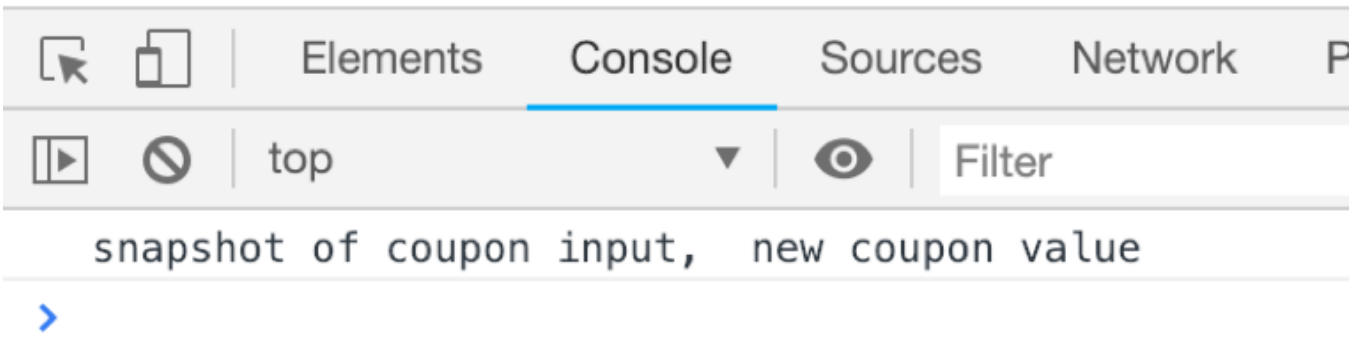


Figure 2 change handler printed value of the text field

As mentioned above, user input is a stream, which can be accessed as an observable on the form control instance (`this.coupon`). Note that the above sample has only accessed a snapshot of data. To access the observable, use `valueChanges` object as shown here.

```
this.coupon.valueChanges.subscribe(
  value => console.log("value as changed by the user", value)
);
```

See figure 3 that captures stream of user input on the observable. Notice the last line, which is a snapshot from the earlier sample.

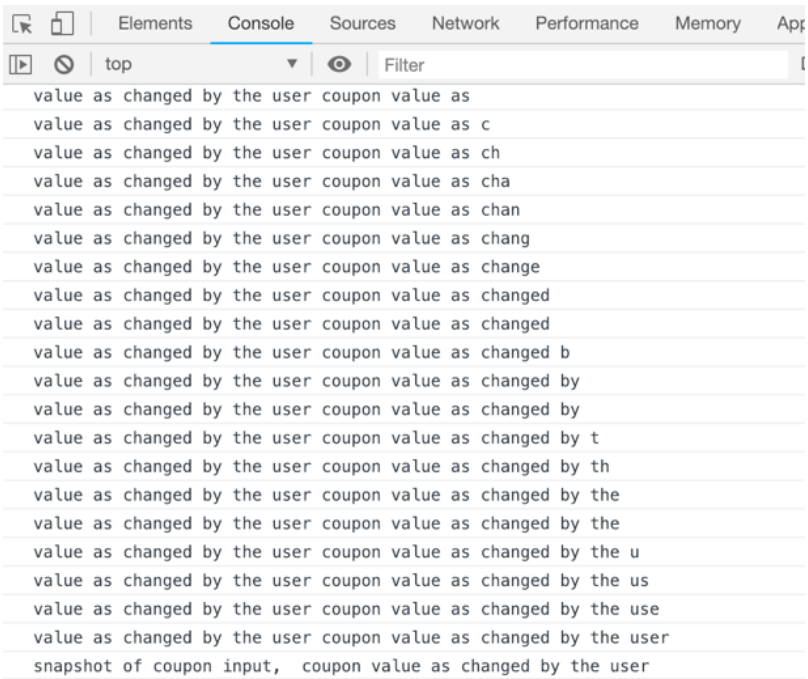


Figure 3 Observable output on form control

Note that more often than not, accessing snapshot value is sufficient to access user input. However, there could be a scenario to replay model changes triggered by the user, in which we might need observable to stream all user inputs.

So far, we have captured user inputs and acted on the model changes. It is also common that the model is updated by an event and the UI needs to reflect the change. Use `setValue` function to gracefully update the model and hence the view or template value.

The event that updates the model value could be triggered by a different component in the application, or a server-side state change observed by the Angular application and so on. For simplicity the following code snippet uses timeout to update the coupon value. The new value will be set on the view.

```
setTimeout( () => {
      this.coupon.setValue("Coupon set by an event")
}, 3000);
```

FORM GROUP FOR MULTIPLE FORM CONTROLS

Most real-world examples of forms have more than one form controls. A **form group** is used to manage such forms. It is a collection of Form Control elements.

Create a traveller form (figure-4) with traveller details.

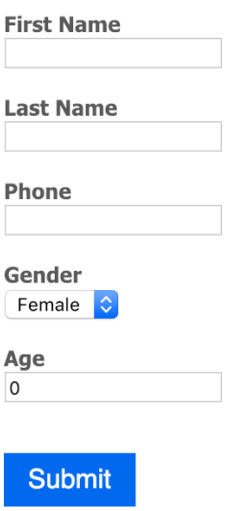


Figure 4 A sample form for traveller

Next, create a form group and the model representing the form in the Component class.

```
  @Component({
  // Removed code for brevity
})
export class InputCouponComponent implements OnInit {

  traveller: any; // We can explicitly type it to FormGroup

  constructor(private fb: FormBuilder) { }

  ngOnInit() {
```

```
    this.traveller = this.fb.group({
      firstName: '',
      lastName: '',
      phone: '',
      gender: 'female',
      age: 0,
    });
  }
```

The traveller class property represents the form group. Similar to the previous sample, this sample injects `FormBuilder` service into the component class. Now use the helper function group to create a form group. Notice the JSON object passed as a parameter. Each field represents a form element. We have provided default values that are shown on the form when it loads.

The group function on `FormBuilder` object returns a `FormGroup` object. In the above snippet, we assigned a value to the traveller, a class property (`this.traveller`). It is of "*any*" type. However, we can import the class `FormGroup` from '*@angular/forms*' module and explicitly type it as `FormGroup`.

Note: As mentioned earlier, *FormBuilder* is syntactic sugar. We can create an instance of `FormGroup` and `FormControl`. See the following code snippet. Notice how `FormBuilder` simplifies the code and is recommended while creating Reactive Forms.

```
this.traveller = new FormGroup({
  firstName: new FormControl(''),
  lastName: new FormControl(''),
  phone: new FormControl(''),
  gender: new FormControl('male'),
  age: new FormControl(0),
});
```

Next, the following changes need to be made to the template. Notice the highlighted `FormGroupDirective` and `FormControlName`. They refer to the form group object traveller and fields in the `FormGroup`.

```
<form [formGroup]="traveller" >
 <div>
  <label for="firstName">
   <h2>First Name</h2>
    <input type="text" id="firstName" formControlName="firstName" />
  </label>
 </div>

 <div>
  <label for="lastName">
   <h2>Last Name</h2>
   <input type="text" id="lastName" formControlName="lastName" />
  </label>
 </div>

 <div>
  <label for="phone">
   <h2>Phone</h2>
    <input type="text" id="phone" formControlName="phone" />
  </label>
 </div>
```

```
 <div>
  <label for="gender">
   <h2>Gender</h2>
    <select name="gender" id="gender" formControlName="gender">
     <option value="female">Female</option>
     <option value="male">Male</option>
    </select>
  </label>
 </div>

 <div>
  <label for="age">
   <h2>Age</h2>
    <input type="text" id="age" formControlName="age" />
  </label>
 </div>

</form>
```

Now let's create a handler for form submit action. The value property on `FormGroup` object provides snapshot of data (similar to `FormControl`). However, it's a good practice to assign `FormGroup` to another model object representing the person/traveller. This object could be further massaged and sent to server side API for save and other actions.

Consider the following template code snippet that configures class function on submit.

```
<form [formGroup]="traveller" (submit)="formSubmitHandler()">

  <!—removed form for brevity -->
  <button type="submit">Submit</button>

</form>
```

See the component changes for handling submitted form. (Read through the comments in the following code snippet)

```
// Person represents model object for traveller. Form Group values are assigned to
it.
type Person = {
  firstName: string,
  lastName: string,
  phone: string,
  age: number,
  gender: string
};

@Component({
 // Removed code for Brevity
})
export class InputCouponComponent implements OnInit {

  traveller: FormGroup;
  person: Person;

  constructor(private fb: FormBuilder) { }

  ngOnInit() {
```

```
    this.traveller = this.fb.group({
     firstName: '',
     lastName: '',
     phone: '',
     gender: 'female',
     age: 0,
    });
 }

 formSubmitHandler(){
  // get value from FormGroup and save to a model object
  this.person = this.traveller.value as Person

  // further process model object. Sent to Server-side API with a  post call.
  // ...

  // Console log person object for debug purposes.
  console.log("Form values,", this.person);
 }

}
```

Figure-5 shows the details of the logged person object in the console obtained from the `FormGroup`. Values are keyed in by the user.

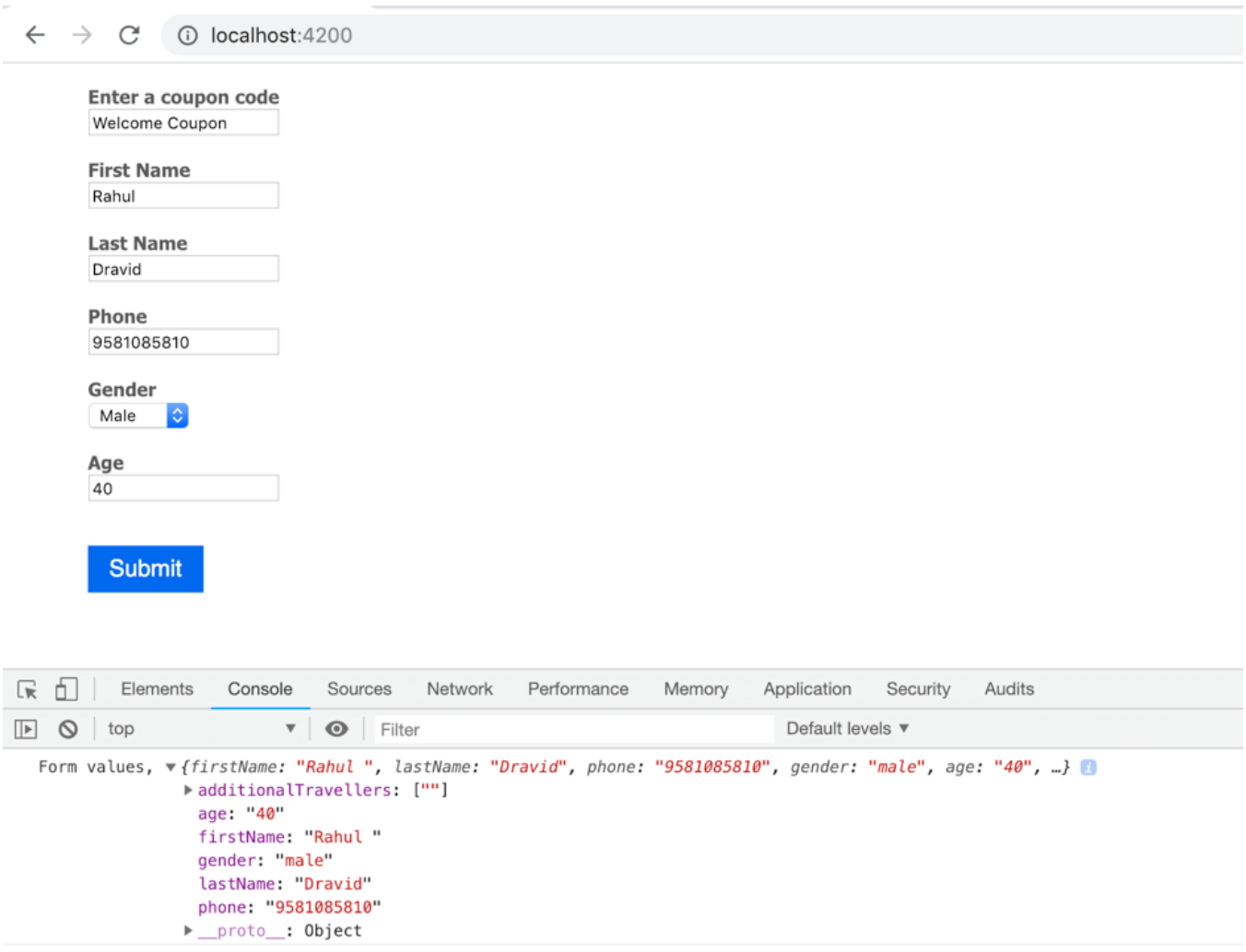


Figure 5 Form submit handler logging snapshot data

CLEAR A FORM GROUP

It is a general practice in data input forms to reset form content, possibly for a new record entry.

`FormGroup` has a function to reset the complete form. We can create a Clear Form button and reset the whole form on a button click. See the following code snippet.

```
// Component class to use the following function on click of reset button.
clearForm(){
   // this traveller is a FormGroup object
   this.traveller.reset();
}

<!-- Template file changes to include the Clear button -->
<button style=”margin-top: 30px;” (click)=”clearForm()”>Clear</button>
```

Remember `this.formGroupObject.value` provides snapshot for `FormGroup` object. We can also stream changes to the form group. The `valueChanges` is an observable on `FormGroup`. Subscribe to observe changes.

As mentioned in `FormControl` example, if required, it is useful to save and replay changes to the form.

```
this.traveller.valueChanges.subscribe(
  value => console.log(“value as changed by the user”, value),
   errorHandler,
   doneHandler
);
```

See figure-6 for the log on observable.

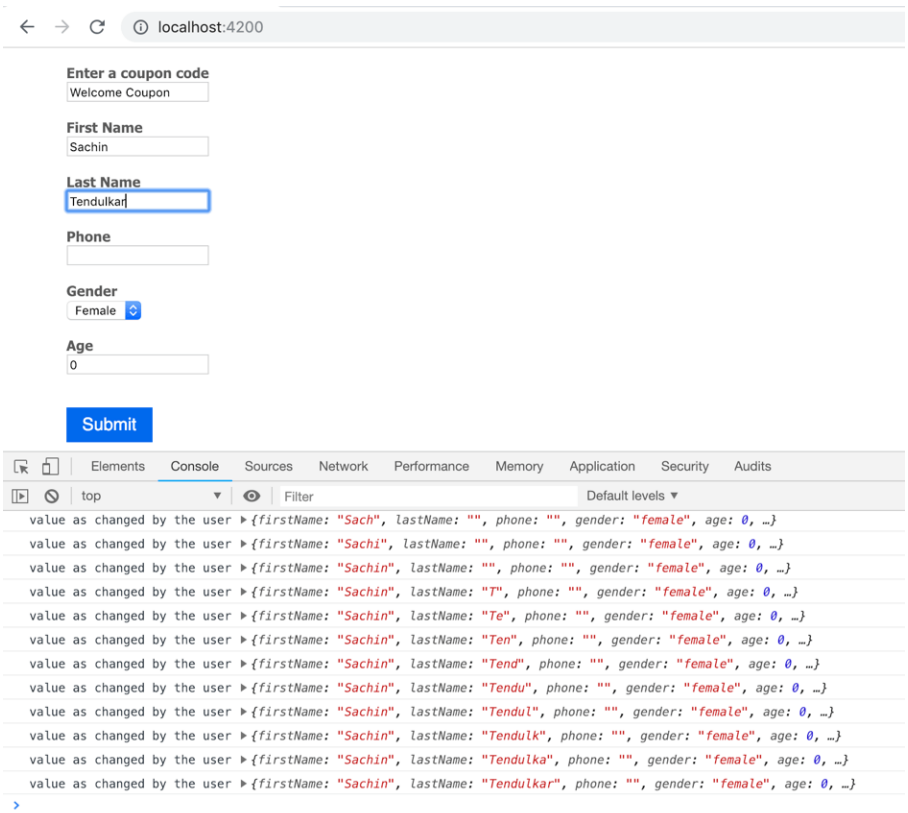


Figure 6 FormGroup’s observable. The subscribe console logging changes to the FormGroup

USING FORMARRAY

Imagine a scenario where form controls are dynamic. In the traveller sample, a primary passenger adds information about additional travellers in the family with her/him. See figure -7.

In this scenario, fields are added on the fly. User can click on the plus (+) button to add more passengers to the travel plan.

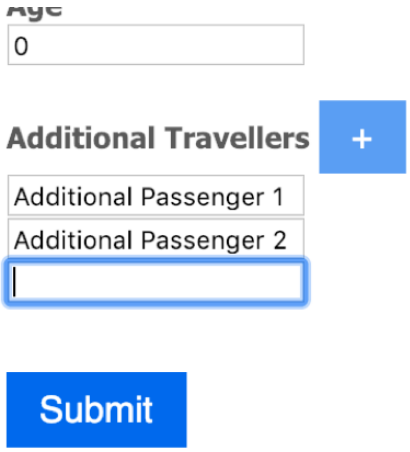


Figure 7 Additional passengers to be added dynamically.

To model it in the component class, we can use a FormArray as shown in following code snippet. We modified FormGroup to add an additional traveller array field.

```
this.traveller = this.fb.group({
  firstName: '',
  lastName: '',
  phone: '',
  gender: 'female',
  age: 0,
  additionalTravellers: this.fb.array([this.fb.control('')])
});
```

Notice the object additionalTravellers. The FormBuilder object's array function returns a FormArray instance. We are using FormArray as the additional number of travellers is not predefined. This way, more fields can be added dynamically.

Note: *For simplicity and consistency, we used the FormBuilder service function.*
Like FormControl and FormGroup, we can create an instance of FormArray with the new keyword as shown here.

```
this.traveller = this.fb.group({
  firstName: '',
  lastName: '',
  phone: '',
  gender: 'female',
  age: 0,
  additionalTravellers: new FormArray([new FormControl('')])
});
```

Consider the template code.

```
<form [formGroup]=”traveller” (submit)=”formSubmitHandler()”>
 <!—removed additional form fields for brevity -->
  <div formArrayName=”additionalTravellers”>
   <label for=”additionalTravellers”>
    <h2> Additional Travellers <button (click)=”addAdditionalTrav()”>+</button>
    </h2>
    <div *ngFor=”let tr of additionalTravellers.controls; let i=index”>
     <input type=”text” [formControlName]=”i” />
    </div>
   </label>
  </div>
 <button style=”margin-top: 30px;” type=”submit”>Submit</button>
</form>
```

Notice `FormArrayNameDirective` referring to `additionalTravellers` in the `FormGroup` from the component class. To make it available on the class, we will add a getter function and retrieve the `FormArray` object from the `FormGroup`. The following snippet calls get on `FormGroup` with `additionalTravellers` field name.

```
get additionalTravellers(){
  return this.traveller.get(‘additionalTravellers’) as FormArray;
}
```

Also notice `ngFor` on the `FormArray` object’s control field. As we have one additional traveller to begin with, one field is shown on the view.

As user adds additional travellers, we can push form controls to the model in the component class, which in turn adds fields to the view. Here’s a code snippet.

```
addAdditionalTrav(){
  this.additionalTravellers.push(this.fb.control(‘’));
}
```

Finally, as the user submits the form, the `FormGroup` will include all additional traveller’s data. Notice the console log from the submit handler in figure-8.

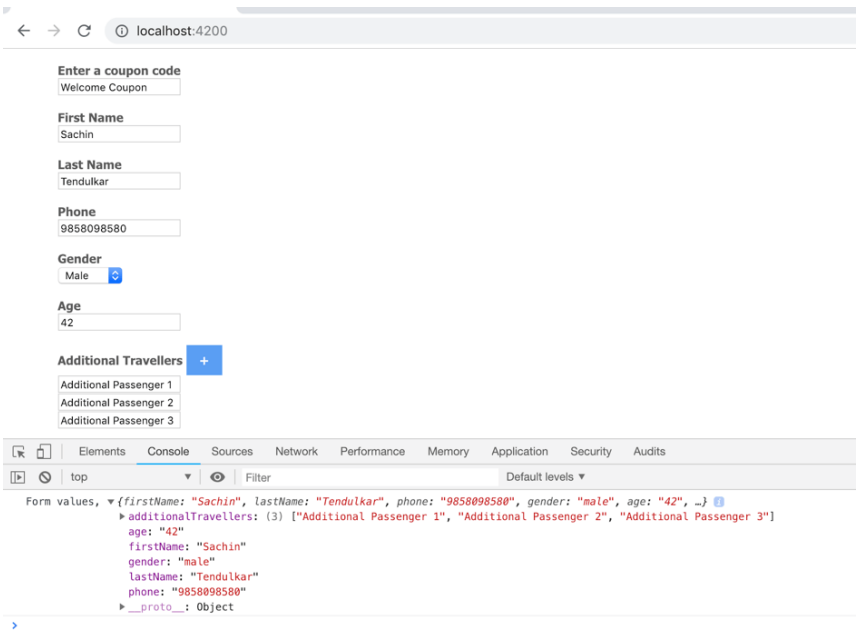


Figure 8 FormGroup includes nested FormArray values

CONCLUSION

Forms are one of the basic features of a web application. In this article, we began by listing the various features that Angular provides to work with Forms. It details working with Reactive Forms, which are most used in an input intensive application. We worked with FormControl, FormGroup and FormArray to build forms. We also used FormBuilder to help simplify the Angular API.

Future articles on the topic will detail validations and template-driven forms.

REFERENCES AND LINKS

Angular documentation- https://angular.io/

Download the entire source code from GitHub at
bit.ly/dncm40-reactiveforms

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.

THE ABSOLUTELY AWESOME BOOK ON ANGULAR

KEERTI KOTARU
RAVI KIRAN

THANK YOU

FOR THE 40th EDITION


@dani_djg

@damirarh


@yacoubmassad

@TRikace

@sravi_kiran

@jfversluis

@keertikotaru


@mayur_tendulkar

@suprotimagarwal


@saffronstroke

WRITE FOR US

mailto: suprotimagarwal@dotnetcurry.com