Issue 37 | July-Aug 2018

DNC Magazine

www.dotnetcurry.com

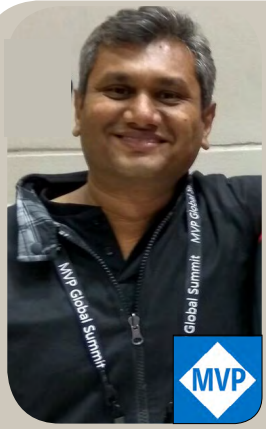
6th

Interview with

Mads Torgersen

Anniversary Issue

EDITORIAL



Suprotim Agarwal
Editor in Chief

I have said this earlier, and I will say it again. "If you want to grow old building communities and help people; you can't do it alone". It takes collaboration across a community to help each other achieve potential.

Today when we reach this milestone of 6 years and over 100K magazine subscribers, I want to express my gratitude to my family, to you the reader, and to my DotNetCurry team of tireless and extraordinary professionals. It takes a good deal of self-sacrifice, efforts, and perseverance to digest the latest developments in technologies, create detailed walkthroughs and guides, and publish it. And while it feels great to be recognized by the community for these efforts, I think none of us do what we do to get adulation. We do it for ourselves because these activities make us whole and complete. Most of all, it's fun! I salute this spirit of my team, and the spirit within you for giving us this opportunity!

As we begin another year, we will continue to stay true to the purpose of this magazine - to help you learn, prepare and stay ahead of the curve. So sit back, relax and enjoy the learning. Reach out to any of us on Twitter with our handle @dotnetcurry or email me at suprotimagarwal@dotnetcurry.com.

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

Art Director : Minal Agarwal

Contributing Authors : Damir Arh, Daniel Jimenez Garcia, David Pine, Gerald Versluis, Jeffrey Rennie, Keerti Kotaru, Ravi Kiran, Suprotim Agarwal, Tim Sommer, Yacoub Massad

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

Next Edition : Sep 2018

CONTENTS

Angular

06

Angular Evolution

Xamarin

20

Xamarin.Forms 3 Cheat Sheet

Vue

26

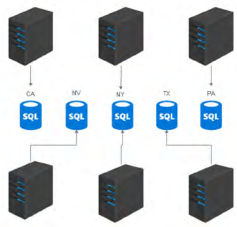
Managing **Vue state with Vuex**

SQLLOCALDB

62

Integration Testing done Simple with **SqlLocalDB**

Sharding



74

A Deep Dive into **Sharding and Multithreading**

C# 8.0

84

Writing Honest Methods in **C#**

VS Code

98

.NET Core Development in Visual Studio Code

Angular

110

Ahead of Time Compilation (AoT): How it Improves an Angular Application

...Interview

Mads Torgersen

RavenDB 4.0

Your Fully Transactional NoSQL Database

The amount of data your organization needs to handle is rising at an ever-increasing rate. We developed RavenDB 4.0 so you can handle this tougher challenge, and do it while improving the performance of your application at the same time.

RavenDB 4.0 is the premiere choice of Fortune 500 companies because it offers you the best of both worlds. It is a NoSQL database that is fully transactional. You can get the benefits of using next generation NoSQL while keeping the best value that relational databases offer. Our open source document database has reached over 100,000 writes and 1 million reads per second using low cost commodity hardware. We ramp up your performance right up until you hit the limits of your hardware.

It's easy to set up a RavenDB 4.0 database cluster and even easier to use. The ramp up time to become an expert is quick. Our RQL query language is similar to SQL, and very familiar with what you are used to. The management studio GUI makes using RavenDB 4.0 convenient for both developers and non-developers. We've automated many of the functions normally assigned to DBAs, freeing up time and resources for other priorities.

Your data never sleeps, so why should your database? Our multi-node data cluster eliminates the single point of failure vulnerability that comes with the relational SQL approach. You can create a distributed data cluster with multiple nodes, all holding a copy of your database and replicating to each other in real time. This gives you high-availability while effectively balancing load, reducing latency, and maximizing your performance. Your customers have the technology to seek information about your business 24 hours a day. With a RavenDB 4.0 data cluster you have the ability to give it to them anytime they want.

Our in-house storage engine pushes performance to the max and minimizes the need for third party plugins. With everything at your fingertips, RavenDB 4.0 gives you all the tools to take your database technology to the next level.

Enjoy Top Performance

1 million reads per second
100,000 writes per second

Easy to Install
Easy to Use

Scale Up Fast with a High Availability Data Cluster

Performs Well on Smaller Servers

Raspberry Pi, ARM Chip

Supports Multi-Model Architecture

Works well alongside SQL solutions

All-in-One Database

Map-Reduce and Full-Text Search are part of your Database

The DBAs Dream

The most functional GUI on the market

Grab a FREE License

3-node database cluster with GUI interface,
3 cores and 6 GB RAM

www.ravendb.net/free

RAVENDB 4.0
DATA MADE FASTER

THANK YOU

FOR THE 37th EDITION

@damirarh


@dani_djg


Jeffrey Rennie

@sommertim

@sravi_kiran


@yacoubmassad

@keertikotaru


@mayur_tendulkar

@davidpine7

@jfversluis

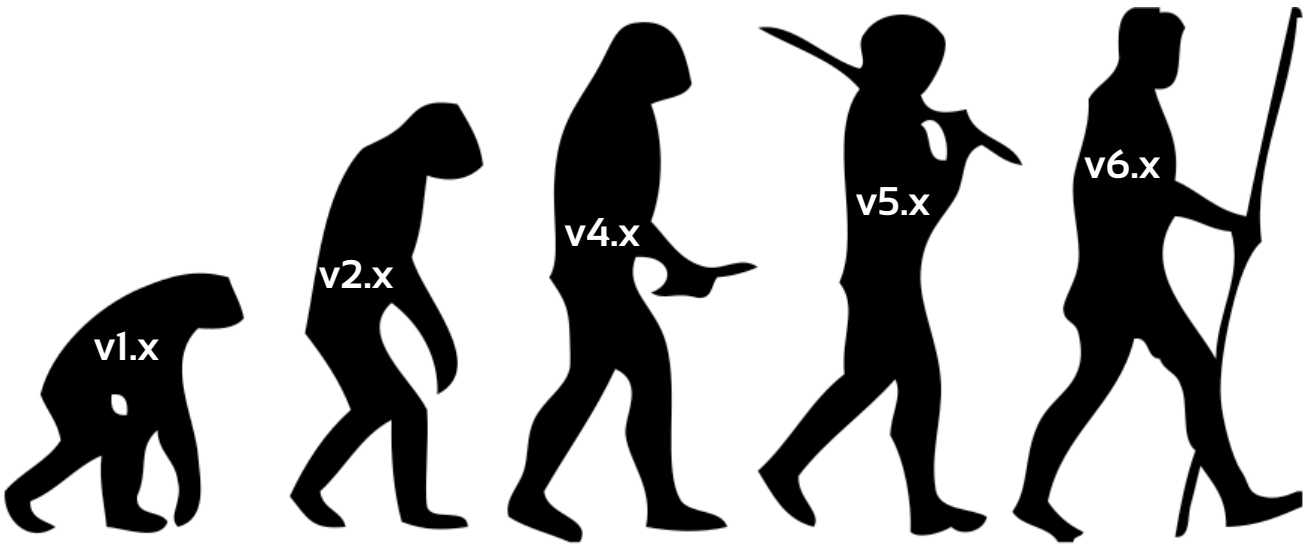
@suprotimagarwal


@saffronstroke

WRITE FOR US

mailto: suprotimagarwal@dotnetcurry.com

Keerti Kotaru



ANGULAR Evolution

Evolution Image source: https://en.wikipedia.org/wiki/File:Human_evolution_scheme.svg

Angular.js, one of the famous JavaScript frameworks for developing rich web applications using HTML5 and latest versions of JavaScript (ES5 and above), has evolved over time.

Angular started as a framework aimed to develop rich web applications, faster and better. It provided structure to the not so evolved JavaScript. As browsers got better, new language features were added to JavaScript. To keep up with these changes, Angular has evolved too.

This article tries to summarize this evolution process. It tries to look back at the original context of Angular application development and its progress.

This article **does not** intend to be a tutorial for all the features over time, rather it tries to summarize and provide a bucket list of enhancements over time.

AngularJS 1.x Context

AngularJS 1.x is seen as MV* framework.

The Model-View-Controller (MVC) and its variations Model-View-Presenter(MVP) and Model-View-View-Model(MVVM) frameworks have been used with applications for a very long time. Be it web applications developed using Java, Microsoft ASP.NET or iOS apps using Objective C, they have all used an architectural pattern. It is a good fit for all applications that have a view to present to the user.

AngularJS's MV* approach was to support any variations of these famous patterns. In an AngularJS application, that runs in a web browser,

1. Controller is a function defined using the API. An object `$scope` is injected in the function.
2. View is defined by HTML templates. Templates can bind data on `$scope` from the controller.
3. Objects on `$scope` could be either Model or View-Model, depending on overall architecture including server-side.

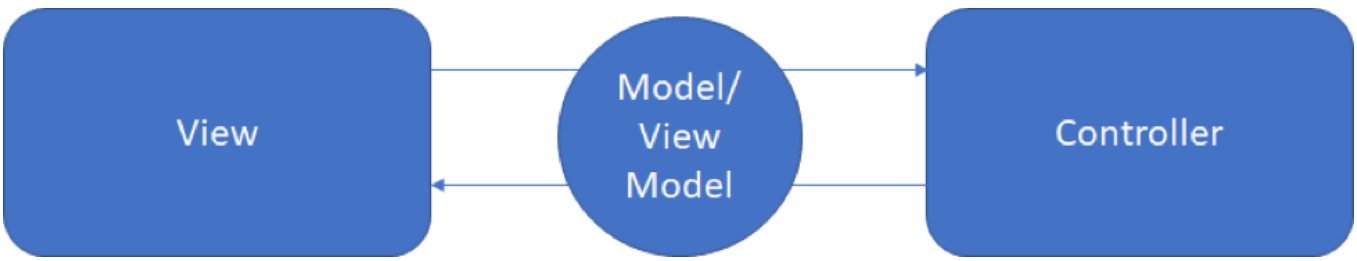


Figure 1: MV* framework

Even though this is a proven architectural pattern and is the standard solution for an application with view, there are certain disadvantages.

Modern web applications required reusable components or custom HTML elements. One should be able to reuse these elements by adding it in the markup and providing state on the fly. Largely, AngularJS 1.x applications achieved it with Directive, which allowed creating custom HTML elements with AngularJS. They allowed manipulating DOM.

Angular 2 application created a category of directives called Components, which allowed creating custom HTML elements and reusing them.

Angular 2 onwards, there is no explicit controller. The framework is not seen as MV* framework anymore.

An Angular application is now seen as composition of components. See Figure 2 where each component has its own view. It may use HTML elements to render data. It may also have multiple child components providing their own view and functionality.

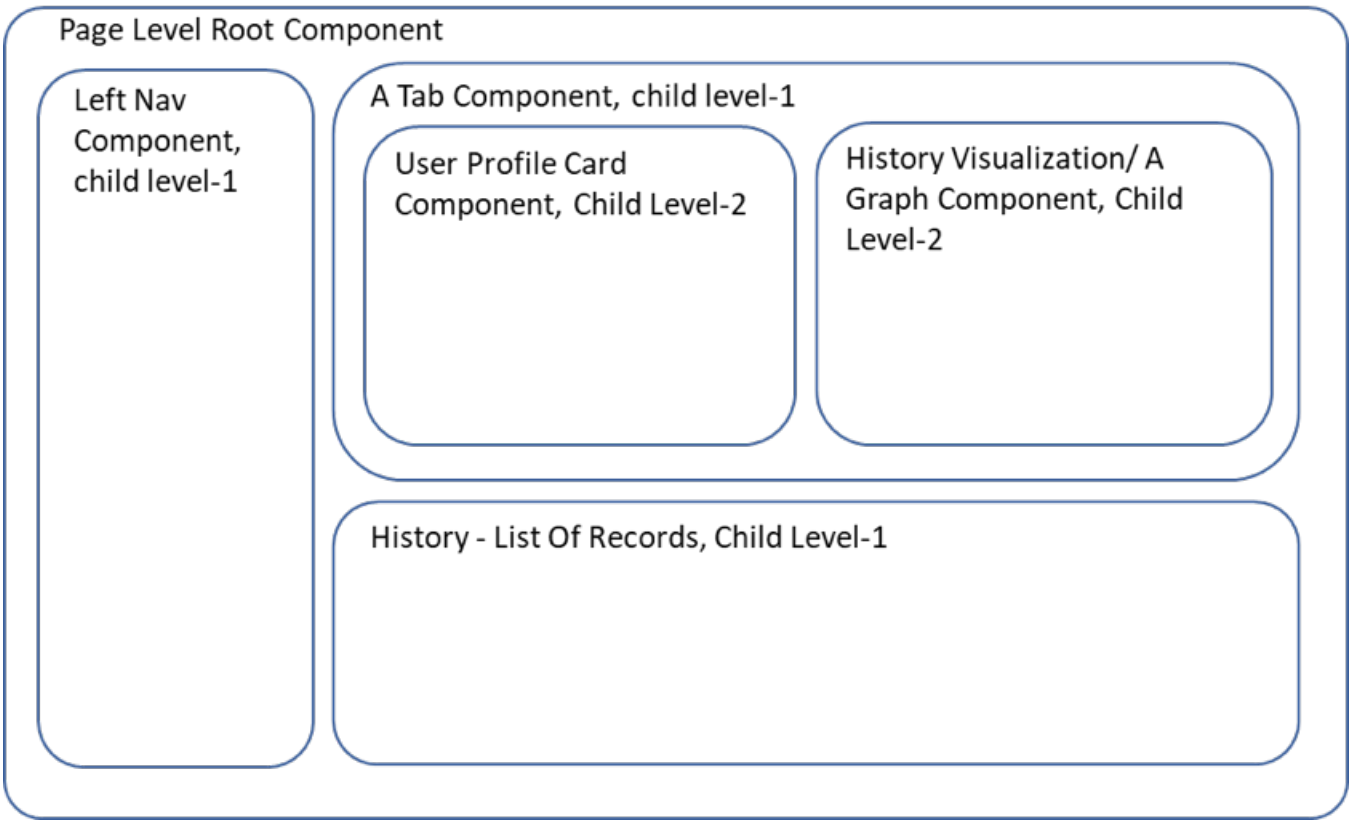


Figure 2: Composition of components

Unidirectional data flow

With AngularJS 1.x, two-way data binding is a powerful feature. The data on `$scope` object in the controller is bound to the template/view. If a change happens to this object, it's updated in the view. If the view changes values, it's updated on the `$scope` object, as well as all other view references.

There could be child elements, views or components (AngularJS 1.5 onwards components were introduced) that are tied to the object. When a change occurs in the child component, it's reflected in the parent and everywhere else that's maintaining reference to this object.

This incurred high performance cost.

Angular 2 encourages using **unidirectional data flow**. State propagation (when data changes) always flows from parent component to the child, not the other way around.

If child component has to change parent's state, it will send an event to the parent. The parent component propagates state change and updates itself as well as all child elements dependent on the state.

Note: *There are patterns like Redux that ease maintaining application state and ensure unidirectional data flow between parent and child components. Moving to the component model of developing UI applications makes it*

easy to use such a pattern. It's difficult to achieve the same results with MVC and AngularJS 1.x.

Change Detection

Continuing on the data binding topic we discussed in a previous section, it has been one of the important features of AngularJS 1.x.

The *Model* object is bound to the view in the HTML template. With two-way data binding, changes to the model are reflected in the view and vice-versa. It means model (object structure) is in sync with the DOM (Document Object Model), which is nothing, but the view.

This feature is very beneficial but is also challenging. For the view to be in sync with the model objects, change detection needs to happen.

AngularJS 1.x framework addressed it with dirty checks. The process is asynchronous.

However, the framework looped through the entire DOM structure to identify changes and synchronize view and the model. The process was costly and relatively slow.

Certain other frameworks like ReactJS addressed its Virtual DOM that differed changes and updated only the changed sections in the view.

This change detection process saw a major improvement with Angular 2 and above. Following are some of the improvements in Angular 2 and above.

Improvements in Angular 2

ZoneJS:

Angular included ZoneJS, which provides execution context for a piece of JavaScript code.
Angular uses it for change detection. Angular extended it (forked the library) and called it ngZone.

With the help of this API, Angular identifies asynchronous operations that might have caused a change. The asynchronous operations could include DOM events like button click, text field value change or an XHR/fetch API call to retrieve data from the server. This API has hooks to trigger synchronizing view.

Change Detection Strategy:

Angular 2 and above has two change detection strategies:

Default

Each component in Angular has its own change detector. It is created during the compilation process and hence is specific to each component. Making it specific helps improve performance.

DOM is organized as a tree while it's rendered on the view. Components primarily render the elements and nodes creating a component tree in an Angular application. As each component has its own change

detector, we have a tree of change detectors.

NgZones API, specifically ApplicationRef's `onTurnDone` event triggers change detector process. As mentioned earlier, NgZones provides execution context and hence it can identify events that can cause change, be it a button click or successful XHR call response.

The change detectors run through component tree and provide the resultant DOM to update the view. This process is fast primarily due to component specific change detectors. See Figure 3.

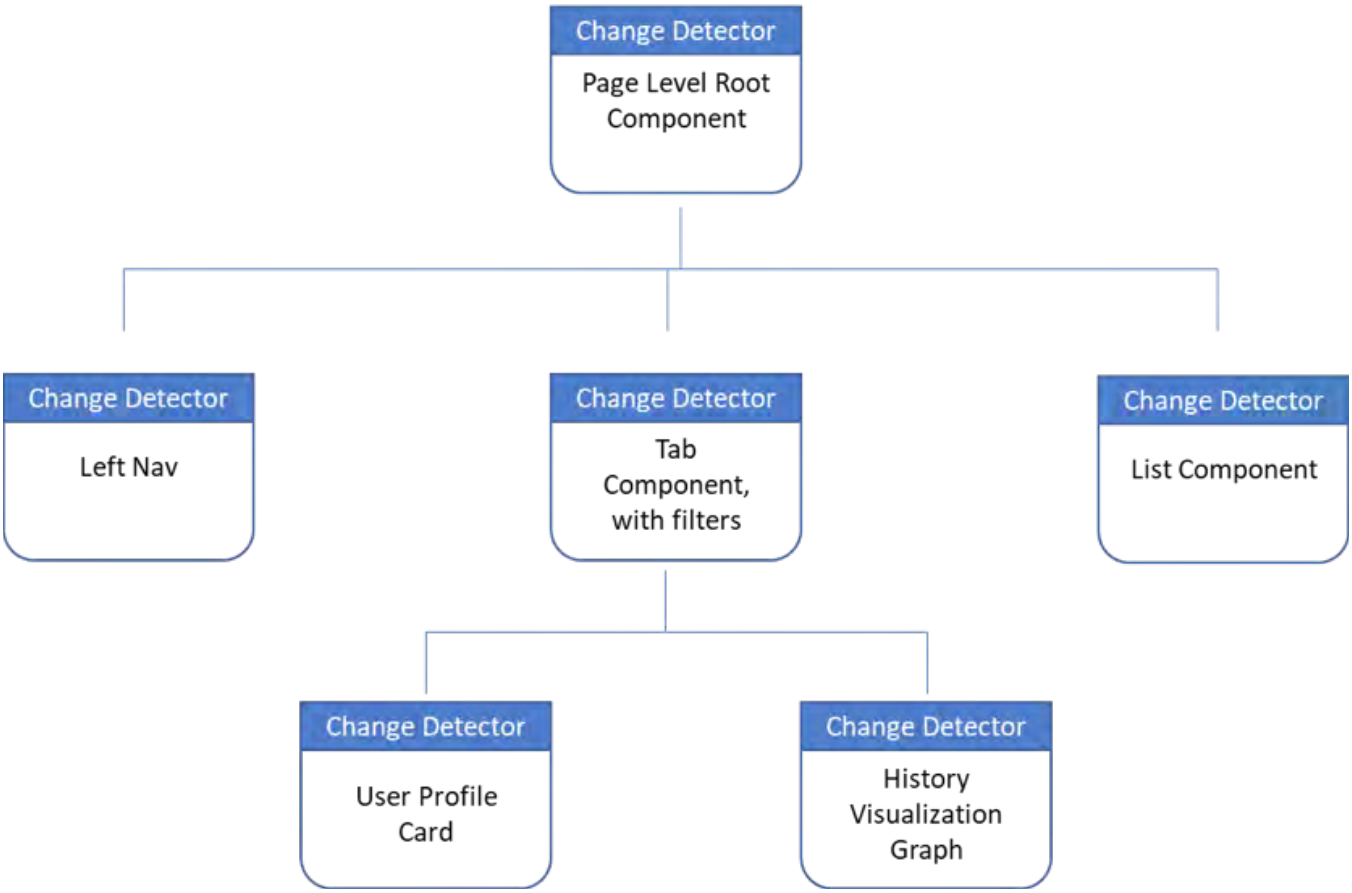


Figure 3: Component Tree

onPush

With Angular 2 and above, performance can also be improved with `onPush` change detection strategy.

Immutable objects: JavaScript objects (non-primitive types) are *call by reference*. When we compare two objects, in reality we are comparing references, not the data or values.

With mutable objects, data might get updated with a change, but the reference will remain the same. Here a simple comparison will not be enough to detect a change. Hence the Angular framework will have to perform a deep comparison, which is a costly operation. The default change detection strategy incurs this additional cost in performance.

As we use immutable objects in a component, we can instruct the component to check for value change only when object reference changes. It will perform better as it reduces work.

Consider Figure 3. Tab Component *with filter* has two child components. The child components *User Profile*

Card and History Visualization Card uses filter input from the parent component.

When user changes filter condition, the `onPush` strategy on the child components and immutable objects are passed-in through input attributes. Angular compares the object references and change detector for just the components invoked. It makes the change detection even more efficient. See Figure 4.

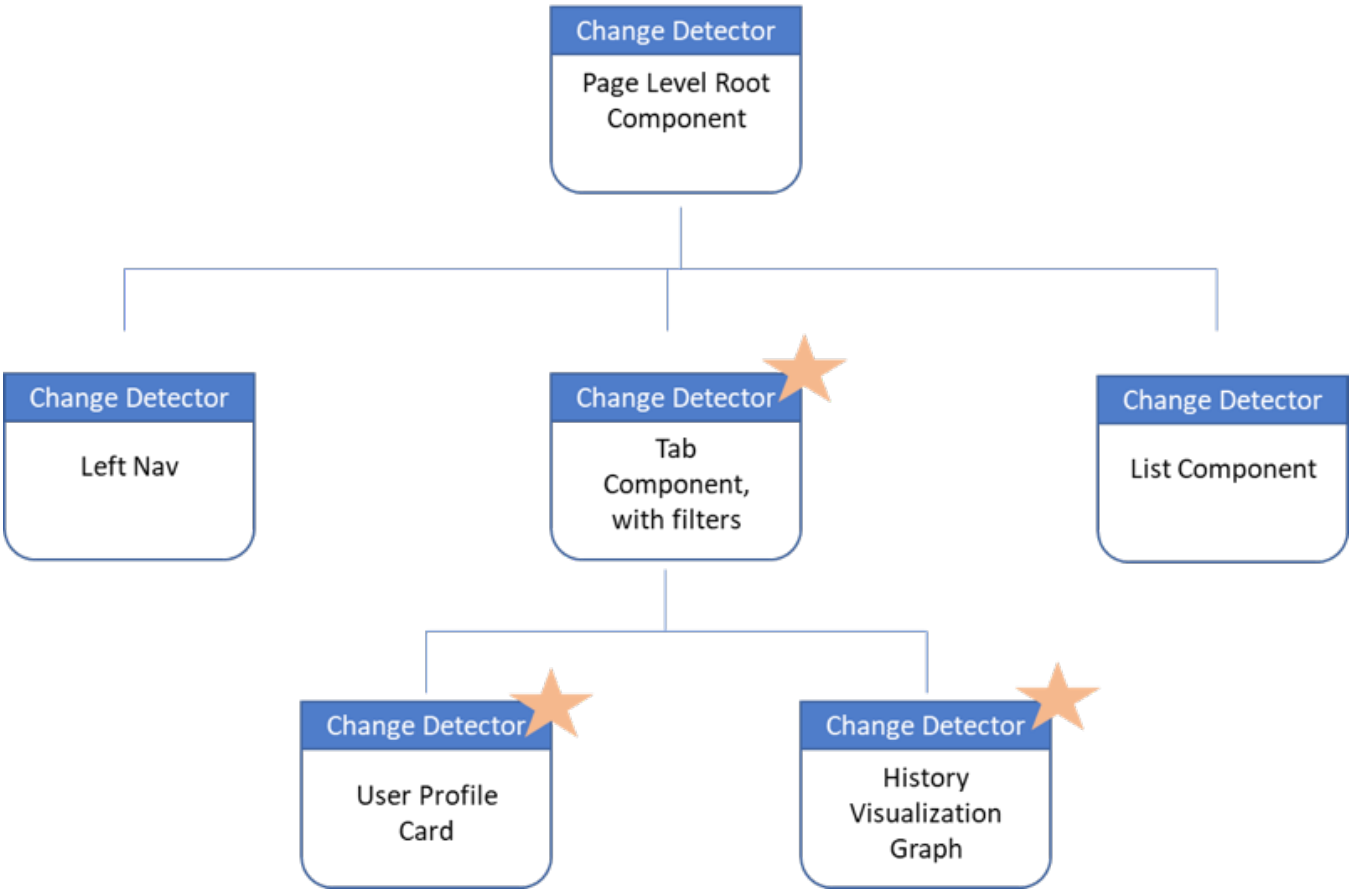


Figure 4: Efficient Change Detection

RxJS

Traditionally, Promise API has been very effective with JavaScript applications. They are asynchronous API for long running operations. The long running operation could be an XHR call or iterating through a large result-set in the browser.

Promises help invoke success or failure callback once the long running operation is ready with the results. However, it's a one-time operation. The promise is closed once result or error is obtained. There is no possibility to stream multiple results. Chaining multiple promises is clunky.

An observable is a next step to promises. RxJS is an implementation of Observables and much more. Observable can stream multiple results until it's closed. It is easy to create an observable from multiple regular operations like looping through an array or converting to an observable from a promise associated with a HTTP call etc.

An observable could also be UI interactions. Actions emitted from a text field as the user is typing-in the text, is a stream, and could be handled effectively with an observable.

Operators - RxJS also comes with ready-made solutions for recurring scenarios working with observables.

Following are some examples.

- Retry - We could attempt to retry a failed HTTP call
- Debounce - Certain observables like UI actions typing in a text field could emit results too quickly, with-in tens of milliseconds. If we are using the data real-time to make a HTTP call, too many calls are triggered. An operator like debounce could be used to wait for certain milliseconds before triggering the next call.

Follow the link for list of RxJS operators by example.

TypeScript

TypeScript is an extension to JavaScript, in other words, it's a superset to JavaScript. It is an open source project backed by Microsoft. Angular takes advantage of additional features and benefits with the language.

- **Data Types** - Supports defining data type while creating a variable
- **Decorator** - Supports decorating additional behavior to a class or a function.
- Generics – Supports building a function or a class that defines data type dynamically. Makes it future proof, as a function parameter's data type or return type need not be defined while creating the function. Similarly, class field type or parameters for a method on the class need not be defined while creating the class. The function creating an object of generic class or calling generic function can specify the type on the fly.
- **Classes** - Similar to ES6, classes can be created in TypeScript code. However, it includes future JavaScript features like controlling visibility of a class field or a method with public, private or protected. A "similar" proposal has been made for JavaScript for creating private fields. It is a Stage 3 proposal in ECMA Script at the time of this writing. Click here to view the proposal.
- **Interfaces** - An interface allows defining structure or shape an object. It helps define contacts for the API.
- **Enum** – TypeScript provides an easy way to create named constants with enums.
- **IDE friendly** – TypeScript with types, compiler and other related features makes it easy for IDEs like Visual Studio Code to provide developer friendly help on APIs and function signatures, navigating to the definitions and references. It will also be possible for IDEs to identify and show errors & warnings better and faster as we type the code.

While Angular was being built, the Angular team announced developing a language extension on top of TypeScript, called *AtScript*. However, TypeScript team driven by Microsoft chose to implement the additional AtScript features in TypeScript version 1.5. With it, Angular chose to make TypeScript a primary language for Angular. However, there are variations to use Angular with basic JavaScript or Dart.

Changes with each version of Angular

The sections we discussed so far have detailed AngularJS 1.x pros and cons and the purpose of Angular 2.

Upgradation to Angular 2 was a major shift in approach and application architecture. The upgradation process was relatively challenging.

Beyond Angular 2 (v4, v5 and now v6), it has been an incremental update to the framework. Majority of the changes are non-breaking. They are enhancements, new features and bug fixes.

The rest of this article will take a different approach going forward. **The latest release will be discussed first, going down to connect with Angular 2 at the end.**

Angular 6

Angular 6 released early May 2018 has considerable improvements to Angular CLI and the Angular eco-system. Following are some of the major improvements.

Support For Angular Elements

Angular Elements is a step in the direction to support Web Components. With the update, an Angular component can be registered as a custom component. A *custom component* can be used outside Angular context. We may include the script and use the component in a React, Angular or even a simple HTML page.

It enables Angular to be used in a smaller context. It doesn't have to be all-or-nothing for Angular applications. Sections of the applications, especially reusable components, could be built in Angular and reused elsewhere.

Angular CLI updates

Angular CLI version has been made consistent with rest of the libraries in Angular Monorepo. The version number has a jump to 6.x from 1.7.x. Some of the important features are as follows.

ng add
Enables adding features to an existing project. For example, a pre-existing Angular application can add material design features with `ng add @angular/material` command.

ng update
With each version update, this feature enables the project to be in sync. It helps project dependencies synchronize to the latest versions. Third party libraries too can provide this feature for applications using the libraries. It can be done with schematic, which needs to be provided by a third-party library.

New starter components with ng generate

Traditionally `ng generate` helps add Angular artifacts like components or services to the project. With the update, additional starter components can be added to the project.

Specifically, with Angular Material:

1. Use `ng add @angular/material` to register new starter components
2. To add a side nav starter components to the project use ng generate `@angular/material:material-nav --name:a-custom-name`

Support for workspace and library

A CLI workspace supports more than one project at one place. These projects can be applications or libraries. A library is a special type of project that could readily be published as a package in NPM or Yarn. Applications consume libraries.

Traditionally we install libraries from an NPM registry (into node_modules) and import in the code file using the library. If the library is part of the application we are developing, it helps identify it as a separate library. The Module system will look for the built library in tsconfig, if not found, it will look for it in node_modules

We can build the library using `ng build library-name` command. Use `--prod` flag if the intent is to publish the build to a registry.

Ivy Renderer

The Angular team is working on a new renderer that could under the hood decrease the bundle size, render the application faster and boost change detection so much so that it is faster compared to the current view engine. It is expected to be a non-breaking change with no changes required for the current applications once upgraded to a version that supports the new renderer.

The render works in a pipeline. It is designed by keeping **tree-shaking** in mind, which helps reduce size of the bundles. Ivy also helps debug template code by adding debug points and pointing to the line of HTML code that caused an error. This is a powerful feature.

Ivy renderer is officially not included in Angular 6. However, Ivy runtime is in active development followed by compiler support. New features are being included in beta releases.

Provide with Injectable

Angular 6 now allows providing a service with injectable decorator. A service could be provided at module level or at component level. Angular creates an instance of the service based on it.

A service provided in component tree has a smaller life cycle. As new instances of component are created, the Service is instantiated multiple times. However, providing a Service at module level reuses the same service instances in many components.

For better tree-shaking capabilities, Angular 6 allows for providing a service at module level with injectable decorator. *Tree shaking* is a bundle optimization process that helps reduce the bundle size. Smaller bundles help browser load, bootstrap and run an application faster. Earlier, service could be provided with providers parameter on `@ngModule` decorator for the module. This approach will continue to work even now.

The decorator injectable is used on the Service class. A parameter `providedIn` can use the string root or a module name that should be providing the service. As the name suggests, root provides the service with the root module. Providing with a module name works better in lazy load scenarios.

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class CustomerSampleService {
  constructor() { }
}
```

Angular 5

Angular 5 was released in Nov 2017. While the version before it (Angular 4) provided features and new syntax for developers, this version focused on performance and reducing bundle size for better bootstrap time. In fact, the first few releases (v4, v5, v6) after a major version change (v2) has considerable improvements to performance and Angular bundle size.

Efficient bundles

Angular 5 made improvements to generating efficient production bundles. The size of the bundle was reduced. Smaller bundle size meant faster load time for an Angular application.

For the projects generated using Angular CLI, a *Build Optimizer* was included. It achieved efficient bundle size because of the following reasons:

- The tree-shaking was improved with the Build Optimizer in Angular 5. Tree-shaking is a process by which unused and dead code can be removed from a bundle that is being generated. The build optimizer marks parts of the application pure, hence implementing effective tree shaking.

- Angular 5 was updated to use RxJS 5.5. It introduced pipeable operators (earlier called lettable operators), which allowed importing one of many, non-default operators with traditional ES6 syntax. This helped tree shaking capabilities, when used with Angular CLI.

  Note: *Projects setup to use Web Pack directly need to update build process to include ModuleConcatenationPlugin, which is available for WebPack version 3 and above.*

  Following need not be done any more

  ```
  Import 'rxjs/add/operator/scan'
  Import 'rxjs/add/operator/filter'
  ```

  We may use the following syntax,

  ```
  Import {scan, filter} from 'rxjs/operators'
  ```

  Note: *Also, it's plural operators (rxjs/operators) as opposed to singular operator (rxjs/add/operator/xyz) now.*

- It is a common practice in Angular to use many decorators. They add dynamic capabilities to the code units. The Compiler would use this code. With Ahead of Time compilation (AoT), one wouldn't need the decorator to be still part of the bundle. Hence, they were cleaned-up from a generated bundle, thereby

reducing size of the bundle.

- Multiple polyfils are not required anymore with Angular 5. Following are a couple of examples:

- There were new pipes added to handle internationalization of numbers, dates and currency values. Before this change, Angular was using browser's i18n API. It would mean there are subtle changes on how browsers are providing values to the application and it needed polyfills as well.

- Angular 5 updated the pipes with its own implementation instead of depending on browser. It would also mean, we wouldn't need polyfills to be included.

- Angular had been using Reflective Injector for instantiating and injecting classes. Reflective injector is dynamic, needed additional map files and polyfills. It was replaced by Static Injector, which is performant, primarily because there is no dynamic decision making. It instantiates and injects Angular artifact configured by developer. It also allowed the need of polyfills and map files.

Enhancement to compilation at development time

- `ng serve` used for live compilation and reloads at the development time had an addition to support Ahead of Time compilation. Use `ng serve --aot`. This will eliminate variation when publishing a build, which would have been done with AOT.

- Angular compiler is hooked into TypeScript transform improving speed of live compilation and reload at the development time. It will make it easy for developers to see the changes reflected in the app being debugged.

Deprecated @angular/http module

HttpClientModule and HttpClient in @angular/common/http introduced little earlier will become mainstream client API to access services over HTTP. It would mean @angular/http is deprecated. Applications will have to upgrade and remove references to @angular/http.

HttpClient comes with more features,

- Additional progress events while making Http calls.
- Request and response objects are immutable
- JSON response doesn't required to be parsed. JSON objects are readily available.
- Use interceptors to add middleware logic in the Http calls' pipeline

Improved Decorator Support

Angular 5 enhancement to decorators allow lambdas and dynamic values to be used in the decorators. Lambdas allow using unnamed functions instead of traditional function syntax for callbacks and values provided in a decorator.

Angular 4

Angular 4.0.0 was released in March 2017. It was the first major version after Angular 2 and considering that, many developer focused features and enhancements were included. However, most of these changes were non-breaking.

A major version update between 1.x to 2.x was an architecture shift in building rich UI applications with Angular. Many applications were rewritten in Angular 2. Organizations, developers and projects spent a lot of time and resources in upgrading their apps from Angular 1.x to 2.x.

However Angular 4 was a totally different scenario. It had incremental changes, and improvements to the framework and applications that were written using the framework.

No major version 3.x

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

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

Following are some more improvements in Angular 4.x

Move animations out of @angular/core
One of the problems Angular had been facing is with size of the bundles. For projects, the framework and all vendor dependencies could grow pretty big. In an effort to reduce the bundle footprint, animations were moved out. Any project migrating from Angular 2 to 4 most likely would install and import animations package explicitly.

```
import BrowserAnimationsModule from @angular/platform-browser/animations.
```

Note: reference the module in imports array of @NgModule

Improved View Engine
Under the hood, AOT compilation process improved with version 4. It reduced size of the compiled component by almost half. A smaller bundle size was achieved with this change.

Angular Universal
Server-side rendering capabilities were made mainstream with version 4. Angular Universal allowed server-side rendering for better search engine crawling/indexing capabilities.

Improved conditional statements in the template
`ng-if` supported `else` conditions with Angular v4 onwards. The ng-if and else statements supported

conditional rendering in the view. It was an enhancement to this feature.

Read the following DotNetCurry article for comprehensive list of Angular 4 features- www.dotnetcurry.com/angular/1385/angular-4-cheat-sheet

Conclusion

Angular brought a major shift with the version 2. However, there are many applications today running on AngularJS 1.x probably due to the cost and effort associated with migrating to Angular 2.

However, considering the improvements, adoption of latest technologies like TypeScript, RxJS etc., I feel it is worth the effort. The migration not only helps with clearing the tech debt, but also helps make the application faster, leaner and better.

References and further reading

Angular Change Detection Explained by Thoughtram

Angular Documentation on RxJS Library

Angular 5 Release Notes and the Blog

Angular 4 Release Notes and Blog

Stackoverflow discussion on differences between Http and HttpClient

Angular 4 development cheat sheet, a DotNetCurry article

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.*

DocuVieware 3

HTML5 Viewer & Document Management Kit

NEW RELEASE

- Easy integration
- Full support for custom snap-in
- Zero-footprint solution
- Fully customizable UI
- Mobile devices optimization
- Fast & crystal-clear rendering

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

DOWNLOAD YOUR FREE TRIAL www.docuvieware.com

Powered by GdPicture.NET 14

DocuVieware is an ORPALIS imaging technologies product

Gerald Versluis

XAMARIN.FORMS 3 CHEAT SHEET

There has been talk around the new version of Xamarin.Forms for a while now. In May 2018, Microsoft released the new major version of Xamarin Forms, and it has great new features.

Much work has gone into stabilizing the whole thing, but that didn't stop the Xamarin team from implementing some new features. In this article I will provide you with a quick overview in the form of a cheat sheet.

This guide will help you have a good overview of the most important new features of Xamarin Forms 3, and a reference on how to use them!

#1. VISUAL STATE MANAGER

You might know the Visual State Manager (VSM) from other XAML platforms, but with version 3.0, it is now available on Xamarin.Forms. With a VSM, you can change XAML elements based on visual states. These changes can be triggered from code. This way you could, for instance, change your form layout whenever the orientation of a device changes.

A visual state, defined in XAML could look something like this:

```
<Style TargetType="FlexLayout">
 <Setter Property="VisualStateManager.VisualStateGroups">
  <VisualStateGroupList x:Name="CommonStates">
   <VisualStateGroup>
    <VisualState x:Name="Portrait">
     <VisualState.Setters>
      <Setter Property="Direction" Value="Column"/>
      <Setter Property="Margin">
       <OnPlatform x:TypeArguments="Thickness" Default="0">
        <On Platform="iOS" Value="0,30"/>
       </OnPlatform>
      </Setter>
     </VisualState.Setters>
    </VisualState>
    <VisualState x:Name="Horizontal">
     <VisualState.Setters>
      <Setter Property="Direction" Value="Row"/>
       <Setter Property="Margin">
        <OnPlatform x:TypeArguments="Thickness" Default="0">
         <On Platform="iOS" Value="30,0"/>
        </OnPlatform>
       </Setter>
      </VisualState.Setters>
     </VisualState>
    </VisualStateGroup>
   </VisualStateGroupList>
  </Setter>
 </Style>
```

As you can see, define a `Style` at the highest level and specify a target type. Inside the style, you can have different groups and a `group` can have states, each `state` defined with a different name.

These states are the key here. In a `state`, you can define values for properties that refer back to the target type, in our case a `FlexLayout`. You can even use the `OnPlatform` conditions to specify different values for different platforms.

Style can be defined in the applications `ResourceDictionary`. Just execute this line: `VisualStateManager.GoToState(Container, (width > height) ? "Horizontal" : "Portrait");` to set a certain state from code.

For example, this line of code would go to an event handler that detects if the orientation of our device has changed, and will apply the desired visual state as necessary. The "Horizontal" and "Portrait" names here refer to the names of the states we defined.

You can find more information on the Visual State Manager in the Microsoft Docs:

https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/visual-state-manager

#2. FLEXLAYOUT

Another big feature added in this major release of Xamarin.Forms is the `FlexLayout`.

With this new layout, you can stack and wrap child views. It works very similar to the concept you might know from CSS as the Flexible Box Layout (or flex layout or flex box).

The `FlexLayout` is related to the `StackLayout` you might already know from working with Xamarin.Forms. The big advantage of the `FlexLayout` is that it will wrap its children where needed. When there is no more space in the row or column where the children are defined, it will move the rest of the items to the next row/column.

A FlexLayout is very easy to define, as you can see in the code block underneath.

```
<ContentPage xmlns=”http://xamarin.com/schemas/2014/forms”
 xmlns:x=”http://schemas.microsoft.com/winfx/2009/xaml”
 xmlns:local=”clr-namespace:FlexLayoutDemos”
 x:Class=”FlexLayoutDemos.SimpleStackPage”
 Title=”Simple Stack”>
<FlexLayout Direction=”Column” AlignItems=”Center”
 JustifyContent=”SpaceEvenly”>
  <!—Your controls here -->
 </FlexLayout>
</ContentPage>
```

Define it as any other regular layout element you are used to. With the properties you set on the FlexLayout, you can determine how to layout the children and how they should align and wrap.

To learn more in-depth about the `FlexLayout`, please refer to the Microsoft Docs:
https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/layouts/flex-layout

#3. STYLESHEETS (CSS)

There has been a lot of talk about implementing CSS in Xamarin.Forms. You can love or hate it, but it is available for you to use!

XAML already bears similarity with HTML so it makes sense to use CSS to apply styling .

To use CSS in your app, you need to take three steps:

- Define your CSS
- Add it to your (shared) project, set the build action to **EmbeddedResource**
- Consume your CSS file

A simple example of a CSS file could look like the code underneath.

```
^contentpage {
  background-color: lightgray;
}
#listView {
  background-color: lightgray;
}
stacklayout {
  margin: 20;
}
.mainPageTitle {
  font-style: bold;
  font-size: medium;
}
listview image {
  height: 60;
  width: 60;
}
stacklayout>image {
  height: 200;
  width: 200;
}
```

If you are familiar with CSS, you can see it looks like the real deal and has the same capabilities. The ^ selector might seem strange since it's not available in regular CSS. With this selector you can target a base type. The quick overview of all selectors:

- Type selector: specify the name of any type to target that type
- ^ selector: select a base type
- # selector: targets an element by name, specified with the `x:Name` attribute
- . selector: select an element with a specific class, specified with the `StyleClass` attribute

And of course, you can select child elements in different ways.

To consume a stylesheet, there are different ways to do that. From XAML, you can reference an external file by declaring it like this:

```
<ContentPage.Resources>
        <StyleSheet Source=”/Assets/styles.css” />
</ContentPage.Resources>
```

You can also define your CSS inline. To do so, do this:

```
<ContentPage.Resources>
  <StyleSheet>
    <![CDATA[
      ^contentpage {
         background-color: white;
      }
   ]]>
   </StyleSheet>
</ContentPage.Resources>
```

There are also ways to load css files dynamically from code or interpret them from a string.

If you want to learn about all the nooks and crannies, head over the Microsoft Docs page:
https://docs.microsoft.com/en-us/xamarin/xamarin-forms/user-interface/styles/css/

#4. RIGHT-TO-LEFT LOCALIZATION

With Xamarin.Forms 3, it is now possible to apply a `FlowDirection` on any `VisualElement`, which is basically all controls. Simply call upon `Device.FlowDirection` to retrieve the direction of the device your user is using.

In the ideal case, you can update your app by implementing this piece of code on all of your pages: `FlowDirection="{x:Static Device.FlowDirection}".`

More information about RTL support is in this great blog post:
https://blog.xamarin.com/right-to-left-localization-xamarin-forms/

#5. MAXLENGTH ON ENTRY AND EDITOR

One of my personal favorites is a tiny addition - the new `MaxLength` property on the `Entry` and `Editor` control.

I might be biased because I have added this to Xamarin.Forms myself. This new version of Forms has had a lot of help from the community. Since a while now, Xamarin.Forms is open-source and on Github and they are accepting any good pull-requests you open.

Since I love Forms so much and use it all the time, I thought it would be great to contribute something back. This resulted in the addition of the `MaxLength` property. Using it is easy.

If you have an `Entry` of `Editor` control and you want to restrict the amount of characters that the user can enter, you can apply the `MaxLength` property with an integer value. This could, for example, look like this: `<Entry TextColor="Red" MaxLength="15" />`. This will cause an Entry that shows its characters in red and limits the number of characters to a maximum of 15.

The `MaxLength` property is bindable, so you can bind a value to this to make it more dynamic.

Other properties that were added are `ProgressBar.ProgressColor`, `Picker.FontFamily` and much, much more.

FINAL THOUGHTS

The new version of Xamarin.Forms brings a lot of new goodies while also focusing on stability. Major additions like CSS, the Visual State Manager and Left-To-Right support were long due, and a lot of devs were waiting for it.

Also, community contributions are precious. If you have worked with Xamarin.Forms before, you might know there are a lot of small features and visuals you wish would just be there.

The team at Xamarin has heard you and is now taking on this challenge working together with us. A lot of great things are added already, and I am sure much more will come.

You can see all the release notes associated with this release on this page here: https://developer.xamarin.com/releases/xamarin-forms/xamarin-forms-3.0/3.0.0/ and if you look, you will notice that the first 3.1 prerelease versions are on there, so you get to take a peek into the future!

I hope this article/cheatsheet has given you a good overview of what is new and provided you with a quick-start on these new features.

• • • • • • •

Microsoft® Most Valuable Professional

Gerald Versluis

Author

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

Thanks to Mayur Tendulkar for reviewing this article.

Daniel Jimenez Garcia

Managing Vue state with Vuex

As you write applications with Vue and take advantage of its support for components, you will need to decide how to manage the state of your application.

You will need to answer questions such as:

- **Which component holds each piece of data?**
- **How is that data shared between components?**
- **How are changes to the data communicated across those components?**

The traditional answer to these questions would be to keep your state as part of the instance data of a component. This will be passed down as input properties to other components, establishing a hierarchy of parent and child components. In turn, child components can communicate back changes via events.

While intuitive, this approach will show its problems in large and complex applications.

Events will make it harder to reason about your state and how it changes. Passing down the same piece of data across a big hierarchy of components becomes painful.

Some pieces of data won't even adapt well to this parent-child approach, as they become part of a context shared by many components like a shopping cart in an e-commerce application.

This is when centralized state management techniques become useful.

Vuex is the officially endorsed library for Vue in these matters. During this article we will look at how state is managed with **Vuex**, which is an excellent tool at your disposal for scenarios where the default state management isn't enough.

The companion source code is available in github.

> *Are you new to Vue.js?*
>
> *Check out* www.dotnetcurry.com/javascript/1349/introducing-vue-js-tutorial.

Adding Vuex to your project

Vuex is available on both npm and yarn from where you can install it as any other dependency:

```
npm install vuex –save
yarn add vuex
```

Now in your main file for the client side, make sure Vue is aware that you will be using Vuex:

```
import Vue from 'vue';
import Vuex from 'vuex';
…
Vue.use(Vuex);
```

That's everything you need to do in order to start using Vuex in an existing Vue project. You should now be able to create a simple Vuex store and inject it into your application:

```
const store = new Vuex.Store({
  state: {
    message: 'Hello World'
  },
```

```
});

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

By providing the store to the root Vue instance, it will be injected as `$store` to any component part of your app. This way every one of your components can gain access to the centralized state provided by your store. For example, you could build a hello-world component like:

```
<template>
  <p>{{ message }}</p>
</template>
<script>
export default {
  computed: {
    message() {
      return this.$store.state.message;
    },
  },
};
</script>
```

Notice how this component declares a computed where it exposes the message property of the centralized state. Because any state property follows the Vue reactivity rules, if the state was changed in the store, the computed would be re-evaluated as well and the new message would be rendered.

We are now ready to take a deeper look at Vuex concepts that will help you access and mutate the data in your store and structure bigger applications. As a final note, if you want you can easily create a new project with Vue and Vuex using the vue-cli and selecting Vuex while initializing the project.

> *Vuex relies on ES6, which shouldn't be a problem since most project templates using Vue will include a bundler like webpack and a transpiler like Babel as the Vue ecosystem relies on ES6 as well. If you are manually adding libraries to an existing/legacy project you will want to make sure you add bundling/transpiling support. If you want to follow along, using the Vue CLI is the easiest option.*

Vuex concepts

Centralized State Management

You can think of Vuex as a global singleton where you keep the state of your application.

This global singleton is called **store** by Vuex and will be injected in your components, so they can access the bits of data they need. This being Vue, it shouldn't come as a surprise the data kept by a store is reactive. Changes to the state will re-render dependent templates and re-evaluate dependent computeds.

Imagine a simple parent-child component hierarchy where some data is passed to the child component, where it might be modified by the user. The typical Vue answer is that the data is passed from the parent to the child as *props*, while any changes are communicated back via *events*:

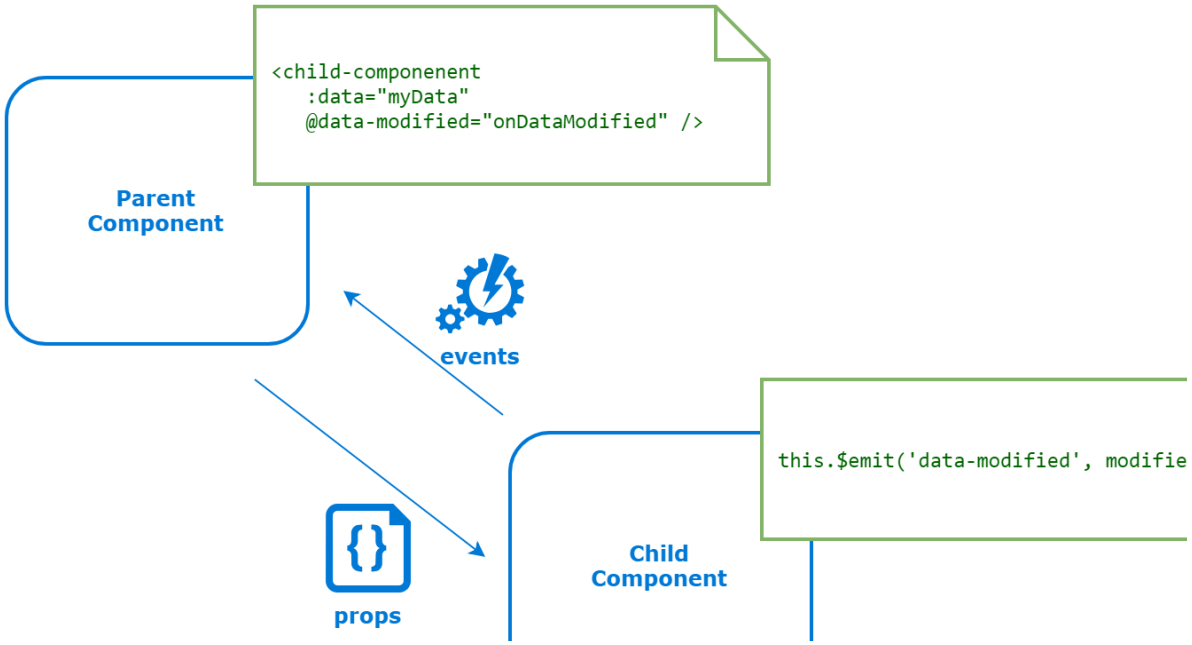


Figure 1, classic parent-child components

If we implement the same component hierarchy using Vuex, the need for events to communicate back changes disappear, as they both use the same central store where the data will be modified.

Now it is even optional to share the data using props, as both can read it from the state:

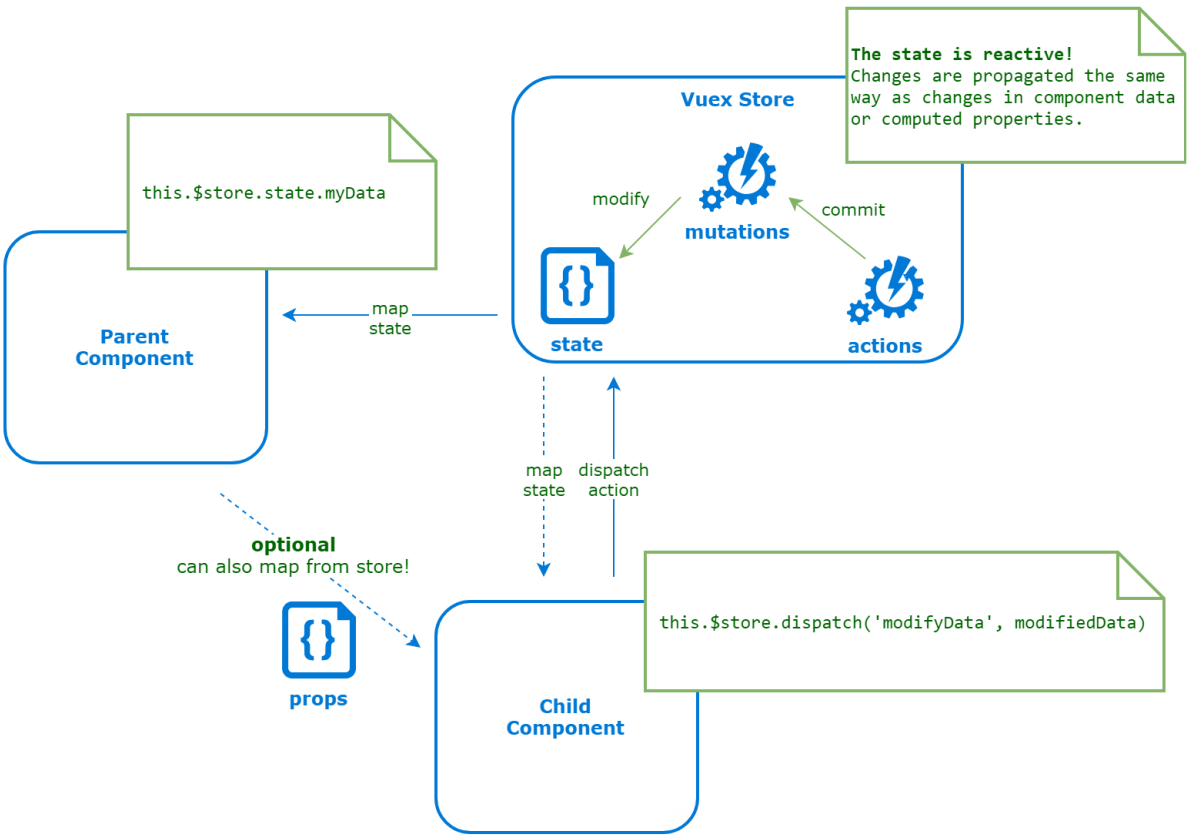


Figure 2, parent-child components with a Vuex store

One shared singleton where everyone reads from and modifies the data can become messy. This is why Vuex provides a concise number of rules and concepts that allow you to manage this complexity in a simple way:

- Data is kept as part of the store's **state**.
- The state can only be modified through **mutations**. This goes as far as Vuex providing an optional strict mode that will raise errors if state changes outside a mutation.

Those are the two core rules to have a centralized store, but Vuex provides additional concepts that will help you further deal with complexity while keeping a centralized store:

- Derived data can be provided through **getters**, the Vuex equivalent to a computed.
- **Mutations** must be synchronous.
- Asynchronous dependencies are handled through **actions**.
- The usage of **modules** will help you develop big stores and provide namespacing.

We will take a closer look to these concepts through the following sections.

State

The state is the heart of any Vuex store, it is the data kept by the store. Earlier, we have seen the definition of a Vuex store whose state contained one property called "message":

```
const store = new Vuex.Store({
  state: {
    message: 'Hello World'
  },
});
```

Any of your Vue components can now get access to the value of the message property as in `this.$store.state.message`. For example, you can declare a computed property:

```
export default {
  computed: {
    message() {
      return this.$store.state.message;
    },
  },
};
```

Since this can get very verbose and repetitive, Vuex provides you with a simpler `mapState` helper that achieves the same result:

```
import { mapState } from 'vuex';
export default {
  computed: {
    ...mapState([
      'message'
    ]),
  },
};
```

> *Not recognizing the syntax? Since mapState returns an object with a property per each mapped state's property, we can use the ES6 object spread operator (the 3 dots in front of mapState) to automatically merge those properties with any other computed property declared by our component.*

This helper is useful when mapping multiple properties from the store as it lets you map them as computed properties in a very concise way by adding more property names to the array. It can also receive

an object that lets you even more interesting mappings like renaming properties or deriving a new value:

```
import { mapState } from 'vuex';
export default {
  data(){
    prepend: 'The store message is:'
  },
  computed: {
    ...mapState({
      message: 'message' // same as before
      originalMessage: state => state.message,
      lowerCaseMessage: state => state.message.toLowerCase(),
      combinedMessage: state => this.prepend + ' ' + state.message
    }),
  },
};
```

One of the most important characteristics of the state is its **reactivity**.

Any changes to a property of the state will be detected, computed properties will be evaluated and templates rendered with the new values. This also means the state follows exactly the same reactivity rules as for component's data, with all of its caveats!

Getters

We know that data is kept in the store state and that it can be read inside components by either accessing `$store.state` or using the `mapState` helper. A common requirement will be then to generate derived data from the raw data kept in the store.

While we could generate the derived data using the `mapState` helper, or declare a computed property where we calculate it, that would not work if we need the same derived state used in multiple components. Every component would duplicate the `mapState` declaration or the computed! This is why Vuex provides **getters**, as they allow you to generate derived data which can then be used in any component.

Imagine a store that holds a user profile with its first and last name. With a getter, you could also provide its full name or a message to greet him:

```
const store = new Vuex.Store({
 state: {
  profile: {
   firstName: 'Terry',
   lastName: 'Pratchett'
  }
 },
 getters: {
  fullName: state => {
   return `${state.profile.firstName} ${state.profile.lastName}`;
  },
  welcomeMessage: (state, getters) => {
   // getters can not just use the state
   // they can use other getters
   return `Hello ${getters.fullName}`;
  }
 }
});
```

This way you can map in any component not just the profile kept in the store, but also the derived fullName or welcomeMessage. With the state, you could directly access `$store.getters.fullName` or you could use the handy `mapGetters` helper provided:

```
import { mapGetters } from 'vuex';
export default {
 computed: {
  // you can directly access any getter in your component
  // by accessing this.$store.getters
  manualAccess(){
   return this.$store.getters.fullName;
  }
  // or you can use mapGetters to create a computed with
  // the same name than the getter
  ...mapGetters([
   'fullName',
   'welcomeMessage'
  ])
  // you can also use mapGetters to create a computed
  // with a different name
  ...mapGetters({
  renamedMessage: 'welcomeMessage'
  }),
 },
};
```

Data provided through a getter is also reactive. When the state properties they depend upon change, the getters will be re-evaluated.

The getters we have seen so far depend exclusively on the store state. However, it is also possible for a getter to receive external parameters. For example, imagine the store also holds the date of birth. We might need to display the age that user had when he performed a certain action, which is something we can accomplish with a getter that receives a date parameter:

```
const store = new Vuex.Store({
 state: {
  profile: {
   firstName: 'Terry',
   lastName: 'Pratchett',
   dateOfBirth: new Date('1948-04-28')
  }
 },
 getters: {
  getAgeInDate: state => date => {
   let age = date.getFullYear() - state.dateOfBirth.getFullYear();
   const m = date.getMonth() - state.dateOfBirth.getMonth();
   if (m < 0
    || (m === 0 && date.getDate() < state.dateOfBirth.getDate())) {
     age--;
   }
   return age;
  },
 }
});
```

This getter can be accessed in a component in a very similar way to the previous getters we have seen. The main difference is that rather than being used as a computed property, it should be used as a method. This means we can either call it as `this.$store.getters.getAgeInDate(date)` or we can map it as a

method using the `mapGetters` helper:

```
import { mapGetters } from 'vuex';
export default {
 props: [
  'book'
 ],
 computed: {
  // you can directly access any getter in your component
  // by accessing this.$store.getters
  manualAccess(){
   return this.$store.getters.getAgeInDate(this.book.releaseDate);
  },
  // you can also map as a method in the methods section
  // and use it as any other method of the component
  mappingAccess(){
   return this.getAgeInDate(this.book.releaseDate);
  }
 },
 methods: {
  // as usual, pass an array when keeping the same getter name
  // or pass an object where you rename the getters
  ...mapGetters([
  'getAgeInDate',
  ])
 }
};
```

Now you might wonder how is the state modified or even initially populated? Let's shed light by looking at the mutations and the actions.

Mutations

In the examples we have seen so far, the store state already contained the data. For example, the profile store contained a hardcoded name and date of birth as state. Having hardcoded data isn't useful which means we need to set and update the store state.

Vuex provides **mutations** as **the only way of updating the store state**. If we go back to our profile example, we can initially declare the profile as an empty object and provide a mutation to set it:

```
const store = new Vuex.Store({
  state: {
    profile: {}
  },
  mutations: {
    setProfile(state, profile){
      state.profile = profile;
    }
  }
});
```

Rather than directly invoking or executing a mutation, **mutations are committed** in Vuex. All that means is that you don't have direct access to a mutation function, instead the store provides a single commit method that lets you invoke any mutation defined in the store as in `commit('mutationName', payload)`.

As you might be familiar by now, inside a component, you can directly commit the mutation through

`this.$store.commit('setProfile', profile)` or you can use the `mapMutations` helper which will map them for your as methods of your component:

```
import { mapMutations } from 'vuex';
export default {
  data(){
    return {
      profile: {
        firstName: 'Terry',
        lastName: 'Pratchett'
      }
    };
  },
  created(){
    // initialize the store state when the component is created
    // we can manually commit the mutation
    this.$store.commit('setProfile', this.profile);
    // or we can map it below and use it as if it were a component
    // method named setProfile
    this.setProfile(this.profile);
  }
  methods: {
    // as usual, pass an array if you want to keep the mutation name
    // or pass an object where you can rename it
    ...mapMutations([
      'setProfile',
    ])
  }
};
```

We have already mentioned that the store state is reactive and follows the same reactivity rules and caveats of Vue. If you go back to our mutation example, you will notice that the state was initialized with an empty profile object. This way the state's profile is made reactive by Vue and when we replace it with another object, the dependent observers like computed properties can be notified of the change.

- This is one of Vue reactivity caveats that might take you by surprise. By declaring the state as `profile: {}`, Vue will make the profile property reactive but will know nothing of whatever inner properties the profile might have, once it is set (like the firstName and lastName).

- In fact if our `setProfile` mutation was written as `Object.assign(state.profile, profile)` or even `state.firstName = profile.firstName`; etc then Vue won't be able to detect any changes! That's why the mutation is replacing the `state.profile` object with a different one, so Vue realizes the object has changed.

- For individual properties to be reactive (so we can update them individually rather than replacing the entire profile object), declare them in the initial state as in:

```
const store = new Vuex.Store({
 state: {
  profile: {
   firstName: null,
   lastName: null,
  }
 },
 mutations: {
  setProfile(state, profile){
```

```
    state.profile = profile; //ok
    Object.assign(state.profile, profile); // now ok too!
    state.profile.firstName = profile.firstName; // now ok too!
  }
 }
});
```

There is one additional characteristic about the mutations to be discussed before we move on. The mutations need to be **synchronous**!

This is a fundamental restriction on mutations, which was added by design. This way mutations are straightforward and easy to reason about, and development tools can provide a log of mutations including the state before and after the mutation.

Our application can deal with dependencies like HTTP APIs or browser events using Asynchronous code. That's the reason Vuex also provides actions.

Actions

We have seen mutations are the only way of modifying the store state, however they must be synchronous. This is a huge restriction since dealing with asynchronous dependencies is a common requirement in any web application, including fetching data from a server or interacting with browser events and APIs.

Vuex solution to this problem are **actions**, whose implementation can be asynchronous. The caveat is that, since mutations are the only ones who can modify the state, actions need to dispatch mutations themselves to modify the state (they cannot change it directly).

Their definition is very similar to that of the mutations, with the difference that instead of receiving the state as argument, they receive a context object that gives them access to the commit function, the getters and the current state (but remember they can't change it themselves).

As you can see in our extended profile example, defining an action that loads a profile from the server and commits it to the store is straightforward:

```
import axios from 'axios';
const store = new Vuex.Store({
  state: {
    profile: {}
  },
  mutations: {
    setProfile(state, profile){
      state.profile = profile;
    }
  },
  actions: {
    loadProfile({commit}, profileId){
      return axios.get(`/my/get/profile/route/${profileId}`).then(response => {
        commit('setProfile', response.data);
      });
    }
  }
});
```

Not recognizing the syntax? The first argument is the mentioned context object which provides access to the Vuex store, i.e. it is an object like `{commit, dispatch, getters, state}`. Rather than naming its

context and use it as `context.commit()`, we can use ES6 object destructuring and directly extract the properties we need from that object, in this case the `commit()` function.

As you can see the action definition is straightforward and we are free to write any asynchronous code. This example uses the axios HTTP GET method to load the profile JSON from the server. You are free to use your preferred library. All Vuex cares is that changes to the state should be handled by committing a mutation.

Similar to mutations, the store actions are not directly invoked. Instead **actions are dispatched**, for which the store provides a central dispatch method that can be used as in

```
store.dispatch('actionName', payload).
```

And I am sure you have guessed this by now, to dispatch an action from a component, you can directly dispatch it through `this.$store.dispatch('loadProfile, profileId)` or you can use the `mapActions` helper which will map them for you as methods of your component. This way we can now update the definition of our component, where we will ask the store to load the profile from the server. The necessary `profileId` could be read from the current route parameters or received as an input property:

```
import { mapActions } from 'vuex';
export default {
 props: [
  'profileId'
 ],
 created(){
  // ask the store to load the required profile on creation
  // we can manually dispatch the action
  this.$store.dispatch('loadProfile', this.profileId);
  // or we can map it below and use it as if it were a component
  // method named loadProfile
  this.loadProfile(this.profileId);
 }
 methods: {
  // as usual, pass an array if you want to keep the action name
  // or pass an object where you can rename it
  ...mapActions([
   'loadProfile',
  ])
 }
};
```

Before we move on, lets expand on an important point when writing actions dealing with Promises for asynchronous code, which might not be immediately obvious.

Whenever you use promises in an action (as when using axios HTTP request methods which all return promises), you want to return the promise from the action:

```
loadProfile({commit}, profileId){
  return axios.get(`/my/get/profile/route/${profileId}`).then(response => {
    commit('setProfile', response.data);
  });
}
```

Notice the actions has a `return` statement, so when the action is dispatched the promise is returned to the caller. The simple inclusion of the return statement allows you to:

Chain additional behavior from your component. For example, show a spinner or progress indicator while the request is in progress:

```
data(){
 return { loading: true };
},
created(){
 this.loadProfile(this.profileId).then(() => {
  this.loading=false;
 });
}
```

Compose multiple actions. For example, we could create a second action that first loads the profile and then also loads some user preferences. Notice how the new action `loadUserContext` will itself dispatch the `loadProfile` and then perform an additional request to fetch the user preferences:

```
loadProfile({commit}, profileId){
 return axios.get(`/my/get/profile/route/${profileId}`).then(response => {
  commit('setProfile', response.data);
 });
},
loadUserContext({commit, dispatch}, profileId){
 return dispatch('loadProfile', profileId)
  .then(() => return axios.get(`/my/get/preferences/route/${profileId}`))
  .then(response => {
    commit('setPreferences', response.data);
  });
}
```

And with these last points we have finished the overview of the main Vuex concepts. We will now look at a practical example before finishing up with tools and strategies for managing big Vuex stores.

Practical example: TODO list with Vuex

Creating a Vue project with Vuex

The easiest way to generate a Vue project with Vuex as a starting point, is to use the Vue CLI. Once the CLI has been installed, create a new project as in `vue create vuex-todo`. When asked by the `create` command to pick a preset, make sure to select *manually select features* and then select **Vuex** and **vue-router**.

Next, install axios as we will use it to send HTTP requests, so run the following command `npm install --save axios` from the command line.

Once you are ready generating the project and installing the libraries, open the source code in your favorite editor and you should see a minimum Vue application including a router and a Vuex store:

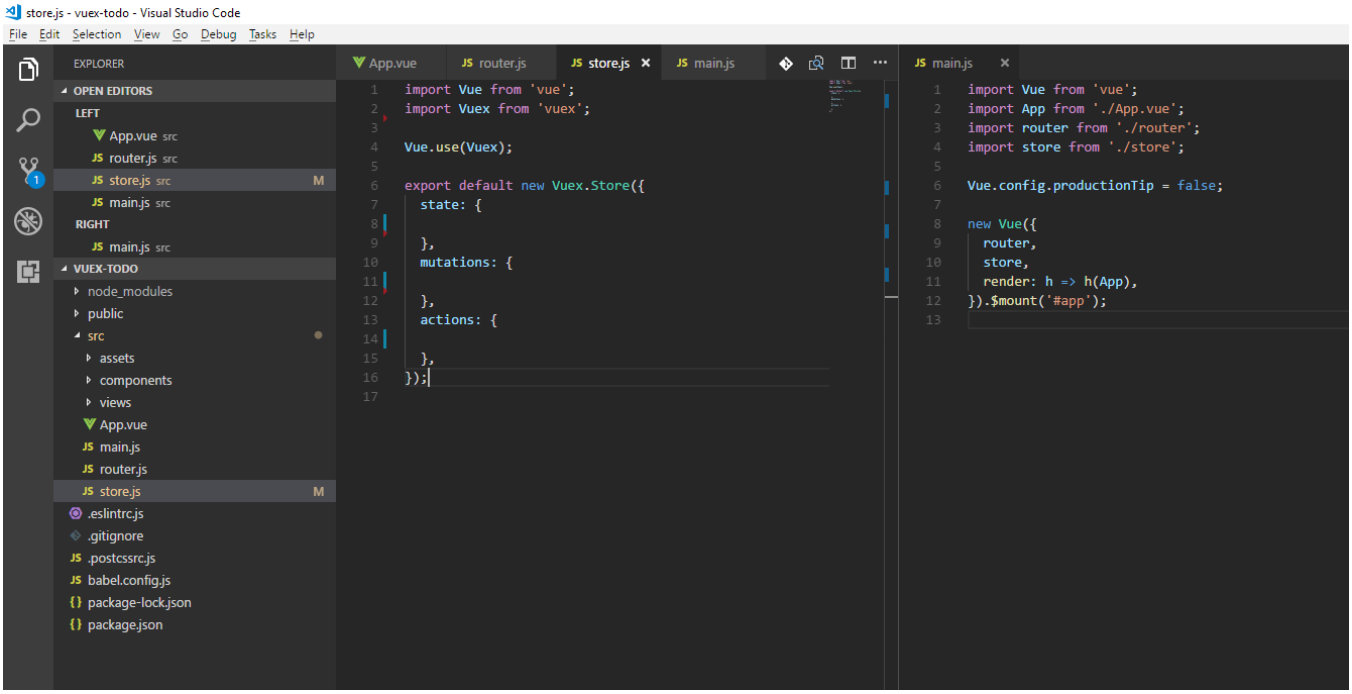


Figure 3, the project generated by the Vue CLI

Notice how **main.js** imports the store and injects it on the root Vue instance, so it is available on every component through the this.$store variable (and the Vuex mapping helpers):

```
import store from ‘./store’;
new Vue({
 router,
 store,
 render: h => h(App),
}).$mount(‘#app’);
```

The store itself will be ready for you to implement it. Right now, it imports and configures Vuex and provides empty state, actions and mutations declarations. Notice how it tells Vue you want to use Vuex and how the store itself is exported so main.js can inject it on the root Vue instance:

```
import Vue from ‘vue’;
import Vuex from ‘vuex’;

Vue.use(Vuex);

export default new Vuex.Store({
 state: {

 },
 mutations: {

 },
 actions: {

 },
});
```

With our project ready, let’s make changes. The first steps we will take are:

- Adding a new page /src/views/Todos.vue

- Adding a new route `/todos` inside src/router.js that is associated with the Todos component
- Finally updating the navigation inside App.vue with a new router-link for the todos route we just created.

It should be easy to follow the example of the existing routes/pages but if you run into trouble, feel free to check the code on github.

At this stage, our Todos.vue component will be as simple as they come, displaying a static header:

```
<template>
 <div class=”todos”>
  <h1>This is my list of things TODO</h1>
 </div>
</template>

<script>
export default {
 name: ‘todos’,
 };
</script>
```

Start the project by running `npm run serve` from the command line, you should see a message telling you the app is running at http://localhost:8080 so open that url in your favourite browser:

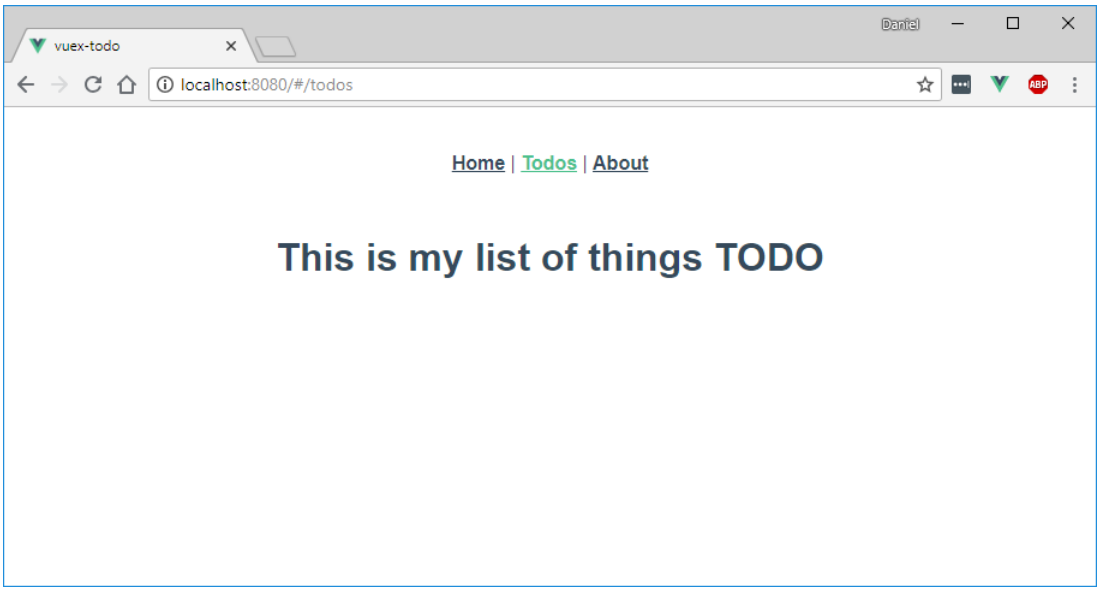


Figure 4, running the project with the new empty page and route

Now we have both an empty page and empty Vuex store where we can start implementing our TODO list.

Displaying the list of TODO items

We will start by fetching the initial list of TODO items from an online test API that comes very handy when building a front-end without a server-side API available. It has a /todo API that can be used to retrieve a list of 200 TODO items, modify them and add new ones (although changes are not persisted, it is just meant to prototype front-ends)

We will start from the Vuex store, declaring an empty array of TODO items as the initial state, providing a mutation to update such array, and an action to fetch them from our newly found API:

```
import Vue from 'vue';
import Vuex from 'vuex';
import axios from 'axios';

Vue.use(Vuex);
export default new Vuex.Store({
 state: {
  // each todo has { id, title, completed } properties
  todos: [],
 },

 mutations: {
  setTodos(state, todos){
   state.todos = todos;
  },
 },

 actions: {
  loadTodos({commit}){
   return axios.get('https://jsonplaceholder.typicode.com/todos')
    .then(response => {
      // get just the first 10 and commit them to the store
       const todos = response.data.slice(0, 10);
       commit('setTodos', todos);
    });
   },
  },
});
```

Everything so far should feel familiar as we have already discussed the role of the state, mutations and actions. The only thing worth discussing is the fact we are just taking the first 10 TODO items from the response as we don't need the 200 that test API returns.

Let's now go to our Todos.vue and integrate the store. The first thing we will do is map the `todos` array from the store state and map the `loadTodos` action. Now we can dispatch the action as soon as the component is created:

```
import { mapActions, mapState } from 'vuex';

export default {
 name: 'todos',
 created(){
  this.loadTodos();
 },
 computed: {
  ... mapState([
   'todos'
  ])
 },
 methods: {
  ...mapActions([
   'loadTodos'
  ])
 },
};
```

Next, we will create a separated component to render a single TODO item. Create a new /todos folder inside src/components and add a new file TodoItem.vue inside. All this component needs to do for now is to render a list item with the TODO title, display the completed ones with a strike through line:

```
<template>
 <li :class=”itemClass”>
  {{ todo.title }}
 </li>
</template>

<script>
export default {
 name: ‘todo-item’,
 props: [
  ‘todo’,
 ],
 computed: {
  itemClass(){
   return {
    ‘todo-item’: true,
    completed: this.todo.completed,
   };
  }
 },
};
</script>

<style scoped>
.todo-item{
 margin: 1rem 0;
 font-size: 16px;
}
.todo-item.completed{
 text-decoration: line-through;
}
</style>
```

Nothing remarkable here, a plain and boring Vue component to render a single TODO item. Now we can go back to the main Todos.vue and use it to render each of the items we load from the API. Update the template section of Todos.vue so it looks like:

```
<div class=”todos”>
 <h1>This is my list of things TODO</h1>
 <ul class=”todo-list”>
  <todo-item v-for=”todo in todos” :key=”todo.id” :todo=”todo” />
 </ul>
</div>
```

Before we can use the todo-item component in our template like we did above, we need to update the script section of Todos.vue. Import the component and provide it in the components property:

```
import TodoItem from ‘@/components/todos/TodoItem.vue’;
...
export default {
 name: ‘todos’,
 components: {
  TodoItem,
 },
  ...
}
```

That’s all we need.

Now when you go to the /todos page, the Todos.vue component is instantiated, which will call the `loadTodos` action. Once the action completes, the list of TODO items will be committed to the store, at

which point the Todos.vue template will re-render, with a `<todo-item />` component rendered for each item.

Give a try at this point!

Before moving on, let's take further advantage of our store. Let's update the store with a new `incompleteCount` getter that returns the count of items to be completed:

```
export default new Vuex.Store({
 ...
 getters: {
  incompleteCount: state => {
   return state.todos.filter(todo => !todo.completed).length;
  },
 },
 ...
});
```

Then map it on the Todos.vue component:

```
export default {
 name: 'todos',
 ...
 computed: {
  ...mapGetters([
   'incompleteCount'
  ])
 },
 ...
};
```

And finally update the template to include a message with the number of items still to be completed, including the following code right after the current `h1` header:

```
<h3>I still have {{incompleteCount}} things to be completed!</h3>
```

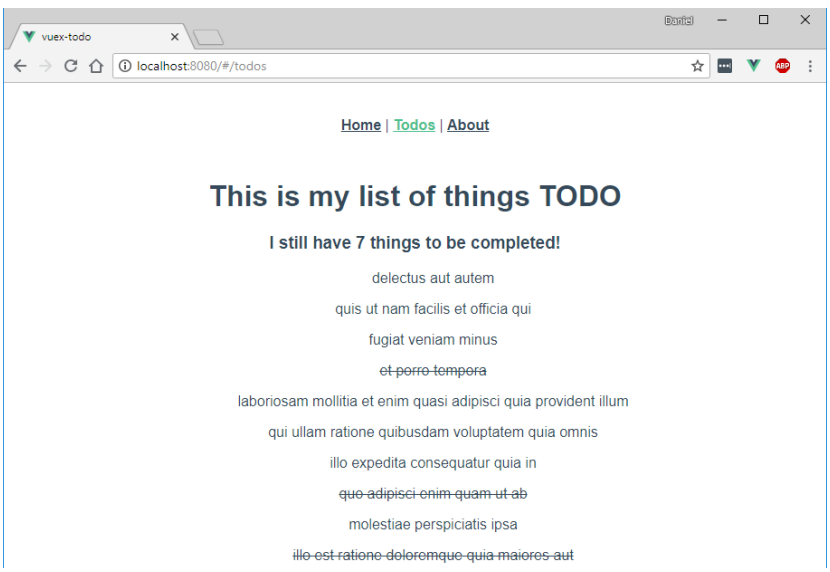


Figure 5, rendering the list of TODO items retrieved from the server

We are done with loading and rendering the list of items, lets move onto being able to complete an item.

Completing a TODO item

To complete a TODO item, we need a new action that will send a request to the server for modifying the item. Once the request succeeds, we can commit a new mutation that will update the completed flag of the item in the store:

```
export default new Vuex.Store({
 state: {
  // each todo has { id, title, completed } properties
  todos: [],
 },
 ...
 mutations: {
  completeTodo(state, todoId){
   const todo = state.todos.find(todo => todo.id === todoId);
   todo.completed = true;
  },
  ...
 },
 actions: {
  completeTodo({commit}, todoId){
   return axios.put(`https://jsonplaceholder.typicode.com/todos/${todoId}`)
   .then(() => {
    commit('completeTodo', todoId);
   });
  },
  ...
 },
});
```

Now all that's left is updating the TodoItem.vue component. We map the new completeTodo action, and we render a link besides the element which when clicked will dispatch the action:

```
<template>
 <li :class="itemClass">
  {{ todo.title }}
  <a v-if="!todo.completed" href="#" @click.prevent="onComplete">
   complete
  </a>
 </li>
</template>

<script>
import { mapActions } from 'vuex';

export default {
 name: 'todo-item',
 ...
 methods: {
  ...mapActions([
   'completeTodo',
  ]),
  onComplete(){
   this.completeTodo(this.todo.id);
  },
 },
};
</script>
```

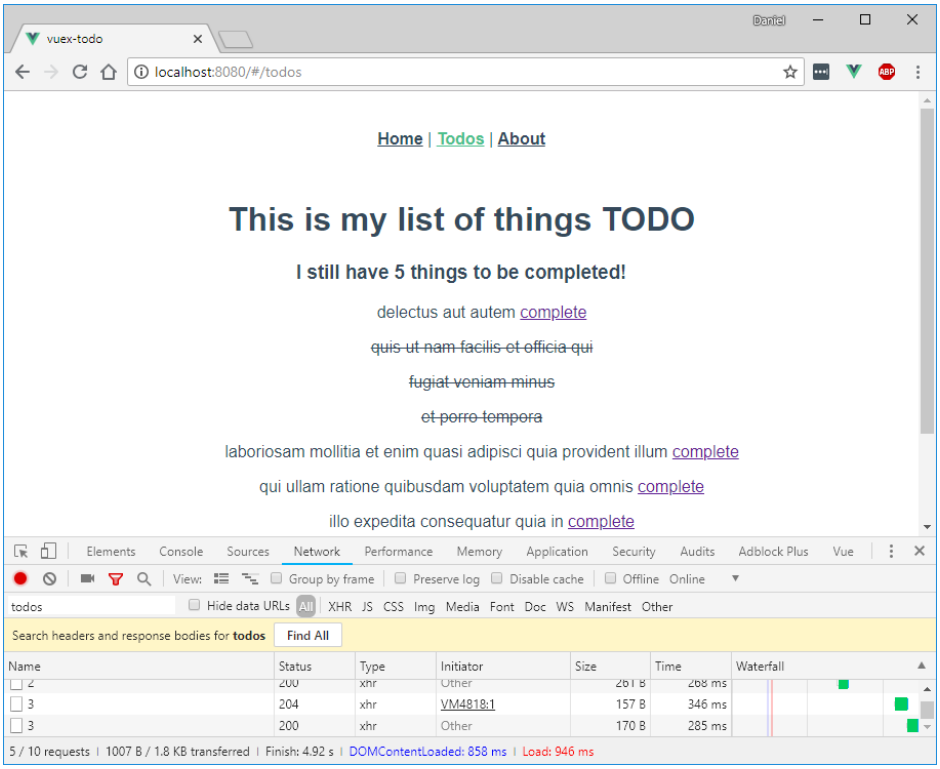


Figure 6, setting TODO items as completed

And that's it, now our TODO items can be updated as completed. Notice how the count of incomplete items automatically updates whenever an item is updated in the store!

Adding a new TODO item

We will be following the same approach as in the previous section, starting by the changes to the store. We need a new action that will post the new TODO item to the server and commit a mutation once the response comes back:

```
export default new Vuex.Store({
 state: {
  // each todo has { id, title, completed } properties
  todos: [],
 },
 ...
 mutations: {
  ...
  addTodo(state, todo){
   state.todos = [todo, ...state.todos];
  },
 },
 actions: {
  ...
  addTodo({commit, state}, todo){
   return axios.post('https://jsonplaceholder.typicode.com/todos')
   .then(response => {
    commit('addTodo', {
     title: todo.title,
     completed: false,
     // ignore the real id from the response.
     // it always returns id=201 but the item isn't really created
     // on the server so we get 404 when trying to complete it
```

```
      id: state.todos.length,
     });
    });
   },
  },
});
```

There are two points worth discussing from the code above.

- First, look carefully at the way the mutation is written. You will see the `state.todos` is assigned to a new array with the new `todo` as the first item. By reassigning the todos property of the state, we follow Vue reactivity rules and this way it will render the new `<li>`. And because we used a `:key="todo.id"` within the `v-for` statement of the template, Vue will know which items were present in the original array.

- Next, we have hit one limitation of the test API we are using. The POST endpoint is functional, but it always returns an object with id=201, which means all new TODO items will have the same id. Even worse, the items aren't created in the server, so we would get a 404 if we try to update them later as completed. To avoid this issue, since we only loaded the first ten items with ids from 1 to 10, we are assigning an id ourselves based on the list length.

Now we will create a new component called NewTodo.vue, where we will render an input to capture the title and will dispatch the `addTodo` action whenever the user clicks a link:

```
<template>
 <div class="new-todo">
  <input type="text" v-model="title" />
   <a href="#" @click.prevent="onAdd">add one more!</a>
  </div>
</template>

<script>
import { mapActions } from 'vuex';
export default {
 name: 'new-todo',
 data(){
  return {
   title: '',
  };
 },
 methods: {
  ...mapActions([
   'addTodo',
  ]),
  onAdd(){
   this.addTodo({title: this.title}).then(() => {
    this.title = '';
   });
  },
 },
};
</script>
<style scoped>
.new-todo a{
  margin-left: 10px;
}
</style>
```

All that is left is to register this new component with the parent Todos.vue component (same as we did with TodoItem.vue) and update this template to simply include `<new-todo />` right above the list.

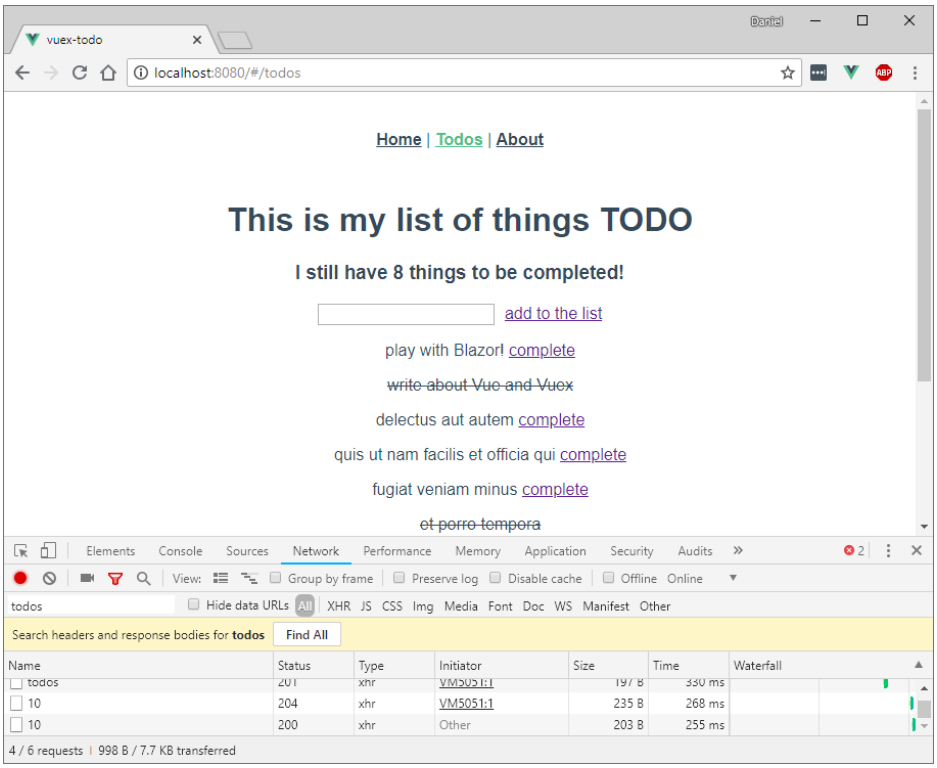


Figure 7, adding new TODO items to the list

This completes our practical example of a TODO list using Vuex! If you had any trouble following along or simply want to look at the end result, feel free to check the code in GitHub.

Structuring large stores

So far, we have seen everything you need to use Vuex, and we have put what we learned into action through the classic TODO list example.

As your application grows and you add properties to the state, mutations and actions; it might get to a point where it is hard to manage such a large store or even ensure you don't have naming clashes!

This section briefly discusses further tools and techniques that will help you scale your Vuex store as your application grows.

Using constants for getters/mutations/actions names

If you are like me, you might have been looking with certain dislike at all the mapper helpers we have used, which relied on magic strings that should match the name of the real state, getters, mutations and actions in the store.

However, we can declare all the names in separate constants file:

```
export const mutations = {
  SET_TODOS: 'setTodos'
  ...
```

```
}
export const actions = {
  LOAD_TODOS: 'loadTodos'
  ...
}
export const getters = {
  ...
}
```

Then use ES6 features when declaring the store properties:

```
import {mutations, actions} from './constants';
export default new Vuex.Store({
 ...
 mutations: {
  [mutations.SET_TODOS](state, todos){
   state.todos = todos;
  },
  ...
 },
 actions: {
  [actions.LOAD_TODOS]({commit}){
   return axios.get('https://jsonplaceholder.typicode.com/todos')
   .then(response => {
    // get just the first 10 and commit them to the store
    const todos = response.data.slice(0, 10);
    commit(mutations.SET_TODOS, todos);
   });
  },
 },
});
```

Notice how the constants are imported and how ES6 allows us to define the actions/mutations/getters using the constant names as in `[actions.LOAD_TODOS]({commit}){` rather than the standard `loadTodos({commit}){` syntax. Also notice how when the action commits the mutation, it uses the constant.

Note: *I wouldn't recommend using constants for the state properties. Doing so would make the code of the actions, getters and mutations hard to follow, since they won't be able to access state properties.*
Additionally, we can use the same constants when using the mapping helpers in your components:

```
...
import { actions } from '@/constants';

export default {
 name: 'todos',
 ...
 methods: {
  ...mapActions([
  actions.LOAD_TODOS,
  ]),
 },
};
```

With this simple trick, you can avoid hardcoded strings across your components.

Vuex Modules

Another thing that will strike you is that defining the entire store and all its properties, getters, mutations and actions within a single file is not practical and won't scale well.

You could easily split them into multiple files which will be imported and combined into one store within a root store.js file. However, this would only solve the single long file problem:

- You might still find it hard to avoid clashes with state/action/mutation names.
- From any given component, it won't be easy to find the code for an action/mutation within your multiple files unless you follow naming convention.

To help you with these issues, Vuex provides the notion of modules. A **module** is basically its own store, with state, getters, mutations and actions, which is merged into a parent store.

```
// Inside /modules/todos.js
export default {
 state: {
  todos: []
 },
 getters: { ... },
 mutations: { ... },
 actions: { ... }
}

// Inside the root store.js
import todosModule from ‘./modules/todos’;
import anotherModule from ‘./modules/another’;
export default new Vuex.Store({
 modules: {
  todos: todosModule,
  another: anotherModule
 }
});
```

In its simplest usage, they are just a formal way of splitting your store code in multiple files.

However, they can also opt-in for namespacing which helps to solve the naming issues. If a module declares `namespaced: true` as part of its declaration, the names for state properties, getters, actions and mutations will be all namespaced to the module name.

When a namespaced module is imported by the root store as in `todos: todosModule`, all the resulting state, getter, action and mutation names will be prefixed with the given module name, in this case `todos/`. That means instead of the `loadTodos` action, you will have the `todos/loadTodos` action and so on.

Since namespacing is a common usage of Vuex, all the mapping helpers support a first argument with the name of the module you want to map from:

```
// Inside /modules/todos.js
export default {
 namespaced: true,
 state: {
  todos: []
 },
```

```
 getters: { ... },
 mutations: { ... },
 actions: { ... }
}

// Inside a component
computed: {
 ...mapState('todos', [
  'todos',
 ]),
},
methods: {
 ...mapActions('todos', [
  'loadTodos',
 ]),
},
```

Notice how the mapping helpers have a first parameter with the name of the module. Plain modules and namespaced modules allow you to scale your Vuex store as your application gets developed and more functional areas are written.

Chrome dev tools

Not just when using Vuex, but when using Vue in general, it is handy to have the Vue.js dev tools chrome extension installed. This will automatically detect when Vue is being used on a page and will let you inspect the component tree, including each component data, and the flow of events.

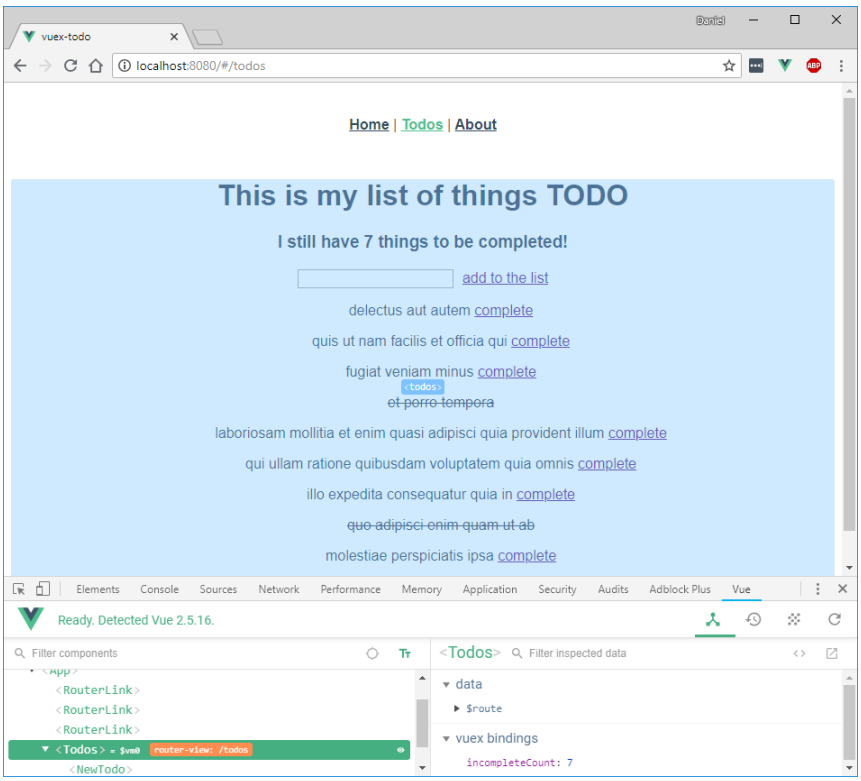


Figure 8, the Chrome dev tools

When using Vuex, the dev tools will automatically display any state/getters mapped from the store on each component. You even get a dedicated tab to the store that will display the current state and the flow of mutations that were committed.

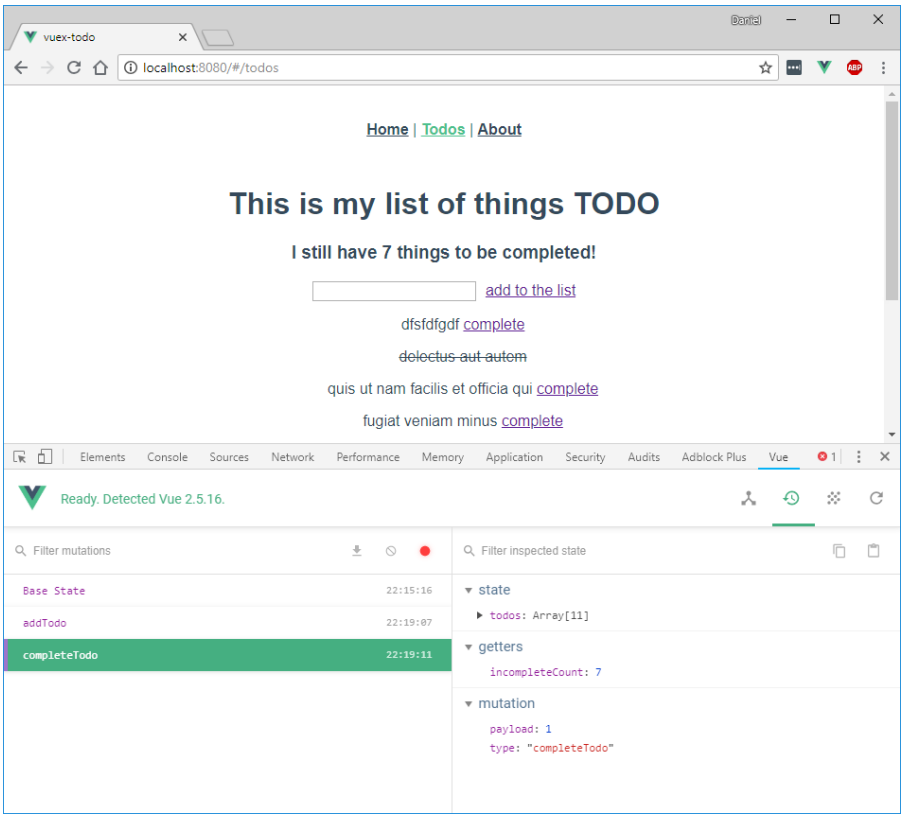


Figure 9, inspecting the vuex state

It might not be immediately obvious that within this tab, you can select any state from the *Base State* to any of the mutations that were committed and inspect the state at that time and the mutation payload!

As you can see, this tool can be very helpful not just during debugging sessions, but also for newcomers to better understand what's going on in their applications.

Conclusion

Vuex provides an alternate state management approach to your Vue applications that makes it very interesting on large apps once the number of components grows and it's hard to keep track of the flow of data and events between components.

In my projects, using Vuex has allowed me to keep the components focused on rendering data and reacting to user events. I have also found a natural place for shared data which doesn't have a clear owner in a pure component hierarchy. In complex pages with deep component hierarchies, I have avoided the feared mess of properties passed deep down the component hierarchy and events delegated up.

At which point does Vuex help rather than add unnecessary complexity is something that varies depending on the requirements and team expertise.

The good news is that following Vue's philosophy, you don't have to go all in with Vuex. Instead, you can mix and match Vuex with the standard props and events approach in different areas of your application. This allows you to introduce Vuex only in areas where/when you clearly see the benefit, like a shopping

cart in an e-commerce application.

If you are using Vue, I would encourage you to spend some time understanding Vuex. At first, it might seem like yet another library you don't need, but as you encounter the limitations of the components props and events, you will be grateful you have an alternative at your disposal!

• • • • • • •

Download the entire source code from GitHub at
github.com/DaniJG/vuex-todo

Daniel Jimenez Garcia

Author

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

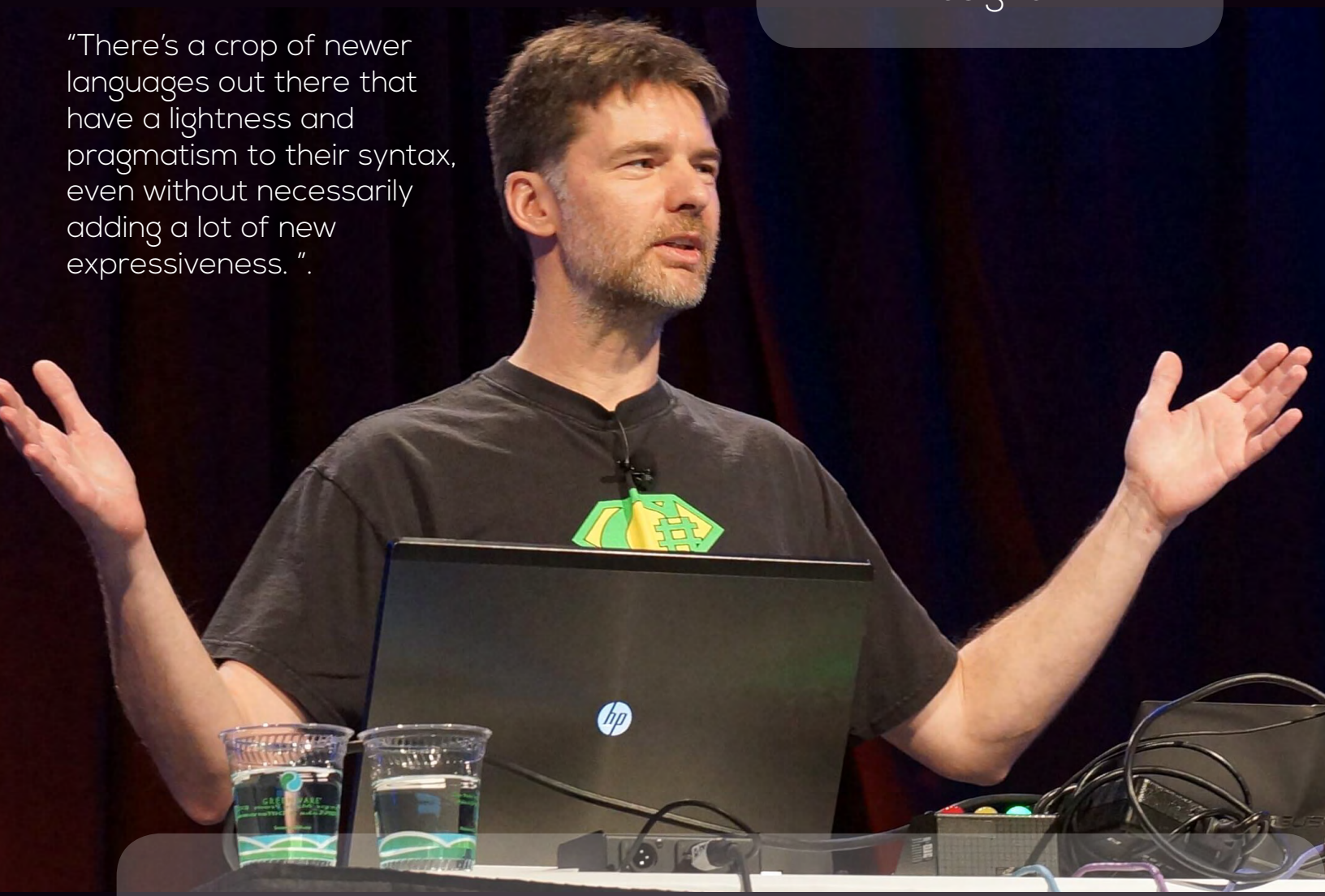
Thanks to Ravi Kiran *for reviewing this article.*

Interview with

MADS TORGERSEN

C#'s Lead Language Designer

"There's a crop of newer languages out there that have a lightness and pragmatism to their syntax, even without necessarily adding a lot of new expressiveness. ".



Someone once said "Don't choose a language for its features, choose a language for its feature building capabilities".

C# is undoubtedly one of the most modern, well designed, flexible and powerful languages in the world. Developers have been incredibly psyched about the new features and the possibilities the language has to offer.

On-stage Picture Credits - Klaus Löffelmann

Amidst C#'s ever-growing popularity, the DotNetCurry team had the opportunity to interview Mads Torgersen, C#'s Lead Language Designer, and find out what's happening in C#, and where does it go from here.

Mads needs no introduction to the C# folks out there, but for others, here's a short bio in his own words.

Tell us about yourself, and about your work at Microsoft. What does your typical day look like?

Thirteen years ago, I moved with my family to Redmond, Washington from Aarhus, Denmark to be a program manager for Microsoft. I was on Anders Hejlsberg's C# language design team from the start, and gradually took over responsibility for running the meetings, writing the notes and maintaining the language specification. About half a decade ago Anders moved to the TypeScript team, and I have taken over as the lead designer for C#.

My most important task is driving and communicating about the design of C#. Twice a week for two hours we have C# Language Design Meetings (LDMs), and that's where all the design decisions ultimately get made. Then there's prep and follow-up from those meetings, there's meetings with colleagues, partners and managers to coordinate, and of course I do a fair number of talks and conferences as well.

What are you really into outside of work?

Family! I think I may be a bit boring; I don't have any crazy hobbies or sports. I like hiking, running, reading, cooking, movies, music... But I don't identify as a Runner, or a Foodie, or a Movie Buff – they are just things I like to do.

What was your journey like to get where you are? How do you learn and approach change?

I fell in love with programming languages as soon as I realized how different they could be from each other. I started out in Basic like so many people, but discovered Forth (for the Commodore 64) by a fluke, and something went POP in my brain, causing irreversible damage!

Then for many years I thought my destiny was to be in research. I went the whole way through a PhD and some years of assistant and associate professorships before I realized the most fun I had was collaborating with industry on language design; specifically on generics for Java. Once I realized, there was a bit of serendipity and good timing, in that I almost immediately met Anders Hejlsberg at a conference where I was presenting Java generics and he was presenting C# generics. That was 13 years ago, and I've been on the C# team ever since.

For our readers getting started with C# programming, help us understand why did Microsoft create C#? How do you envision its future?

I wasn't around at the time, so this is more as seen from the outside. C# grew out of the Java and web age, where object-oriented programming was finally coming into its own, and people were looking for new, more productive tools to match a more rapidly changing landscape. For various reasons Microsoft couldn't just jump on the Java bandwagon, and that turned out to be fortuitous, as the paths of Java and C# have diverged quite a bit over the years.

C# has a target audience of professional developers, and aims to be a productive and expressive choice for your next project in many different domains, and even as the world rapidly evolves. This means keeping C# on a pretty ambitious pace of evolution. There's a risk of the language growing too big (since we can never really take anything out), and we try our best to counter that by being very deliberate about how we do new features, so they fit well within the spirit and feel of the language. We think this is better than stagnation – "good enough" now won't be in five or ten years.

Tell us more about your team and peers. How large is the team that works on the C# language/compiler?

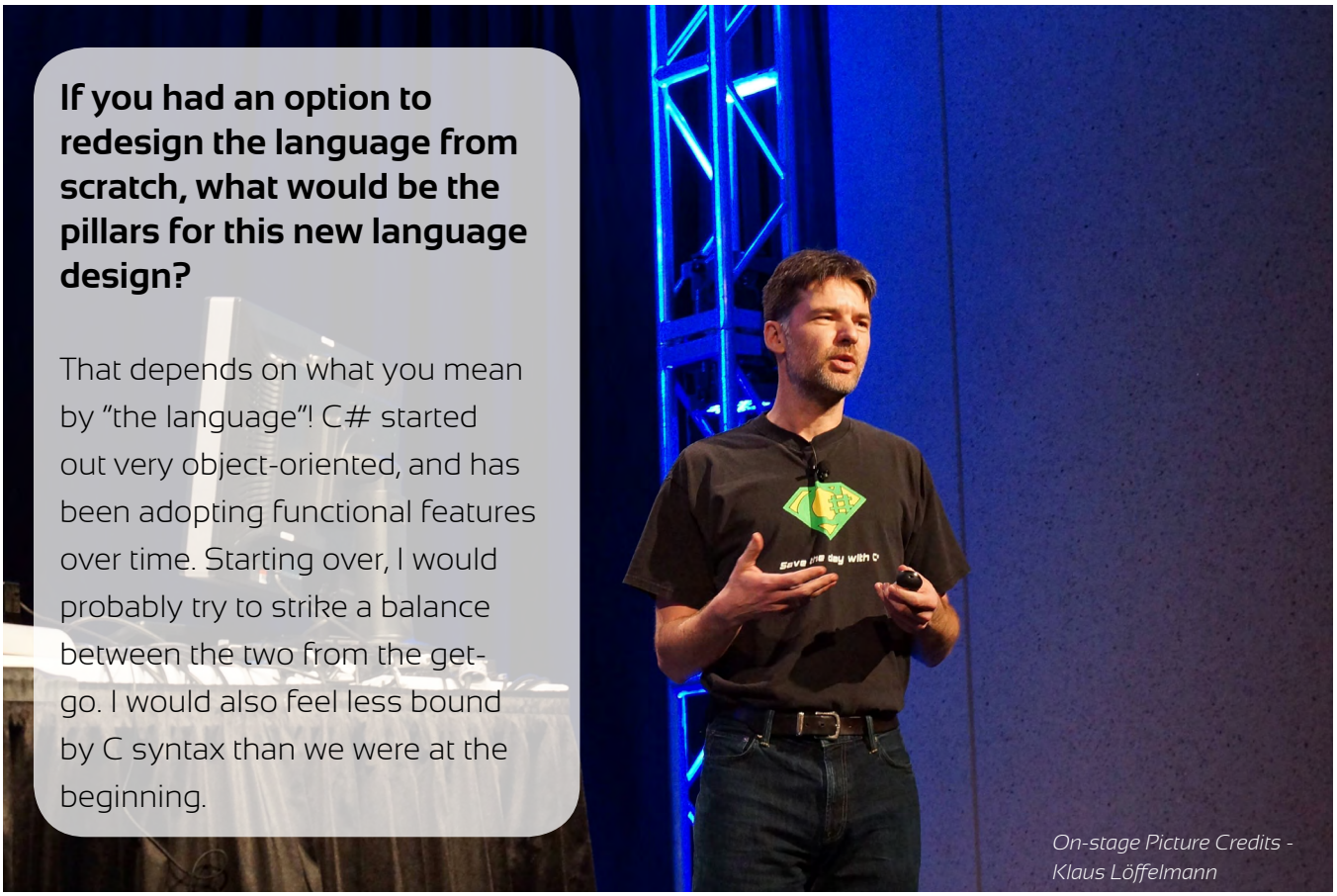
The language design team is about 8-10 people at any given time. Other than me, they aren't full time language designers though. Many of them are on the C#/VB compiler team, which is a team of another 8-10 people total. An additional 7-8 people work on the IDE, and then there's the CLI, the BCL, the CLR... it all depends on how you count.

Nowadays the community also increasingly contributes. We're open source, and even though a compiler is one of the hardest, gnarliest code bases to contribute to, folks have done it

successfully, and we are taking in language features that were implemented by people entirely outside of Microsoft.

What does it take to become a developer on your team?

A lot, frankly. It's very challenging work, and you have to be quite savvy technically, as well as being a good team player.



If you had an option to redesign the language from scratch, what would be the pillars for this new language design?

That depends on what you mean by "the language"! C# started out very object-oriented, and has been adopting functional features over time. Starting over, I would probably try to strike a balance between the two from the get-go. I would also feel less bound by C syntax than we were at the beginning.

On-stage Picture Credits - Klaus Löffelmann

What is the most annoying (or your personal least favorite) feature that exists in C# today? How and why did it get introduced in the language, and what do you plan to do about it?

There are a few! I am not fond of delegates, which are an odd blend of function and collection types. I would have preferred a more straightforward notion of function types. They were introduced primarily to support events, which I also don't think should have been a first-class feature of the language. They correspond to a very stateful design pattern, that is often useful, but not that useful.

We don't take things out of the language, so the two are there to stay. It's possible that we'll consider a more lightweight kind of function types at some point, but they'll have to be worth it.

How is progress going with C# 8? Do you have a rough estimate when it might be released?

I don't like to give even tentative dates! Progress is good, both on the design and implementation front. Ultimately, depending on when is a good time to ship, there may be a number of the features currently considered for C# 8.0 that won't make the cut, but it's too early to say which.

One potential feature for C# 8 was Type Classes or Shapes. How likely do you think this feature will make it in C# 8?

This is one that I think needs more bake time. It is unlikely to make C# 8.0. The general idea behind the proposals in this area are being able to abstract over static members (such as operators), and also be able to apply such abstractions after the fact, as opposed to today's interfaces, which must be applied when a class or struct is declared.

There are many ways to skin that cat, and we are currently exploring several of them. One thing they all share is that they are a pretty fundamental extension of C#'s expressiveness in the type system, so they need to be very carefully woven in with the current semantics and underlying implementation. This will take time, but I'm hopeful that we can design something that is worth the wait!

In the language design process, do you consult or take inspiration from other language teams in Microsoft: F# and Visual Basic, and perhaps even TypeScript?

Absolutely! We have a big overlap between the C# and Visual Basic design teams, and are in continuous coordination. Similarly, for F# we influence each other: F# is our go-to functional language for inspiration to new functional-style features such as pattern matching, and also in turn adopts many of the new C# features.

TypeScript has been a big inspiration when it comes to the nullable reference types feature we're building for C# 8.0. We have a long history of collaboration: Anders Hejlsberg and I both came from C# to help design the first version of TypeScript, and now that we drive each our own language we still meet once a week and compare notes.

What other programming languages influence the design of C# today and what features (not listed in the csharplang repo currently as a proposal) would you like to see implemented eventually?

Philosophically speaking, Scala is an inspiration, in that it set out to create a true unification of the functional and object-oriented paradigms. Though we differ a lot in the specifics, that's a

vision that helps inform the way we integrate new features from a more functional realm. We try very hard to design new features so that they mesh deeply with what's already there, the spirit and feel of C#.

There's a crop of newer languages out there that have a lightness and pragmatism to their syntax, even without necessarily adding a lot of new expressiveness. We are keen to reduce the amount of ceremony in C# as well, again while staying within the feel of the language.

Looking at GitHub issues for C#, the community is constantly submitting a rather large number of proposals. How do you review and evaluate them so that you don't get overwhelmed?

We have a championing model, where a feature proposal has to be championed by a language design team member in order to get discussed in the design meeting. This is a necessary filtering mechanism. We each spend time in the repo looking at proposals, but if an idea doesn't catch the interest of on the language's designers, then we don't bring it to the meeting.

Additional expression-bodied members in C# 7.0 were the first community member contribution that made it into the language. What was the process like in getting the contribution accepted? Are there any other contributions already considered?

It was fun! We continue to take community features; we are currently looking at one for allowing objects to be new'ed up without explicitly giving the type, when it can be inferred from context.

The DotNetCurry Team presenting a copy of the DNC Magazine to Mads at the MVP Global Summit, Redmond.

Go vs Rust vs C# vs Java. Your thoughts?

Rust is certainly interesting. By tracking lifetime in the type system they are able to guarantee memory safety without garbage collection. There are certainly low-level/systems scenarios where that is going to be interesting. Whether the trade-off with a more restrictive type system is going to pan out for higher-level code remains to be seen.

Go speaks to some as a very simple language, but for the same reason it seems to me it runs out of steam a bit. They have interesting technology with the lightweight threads and the real-time garbage collection guarantees, but most of their popularity so far has come from their association with microservices. It'll be interesting to see if they can take hold beyond that.

Java's evolution has been hamstrung for a long time by its myriad of vendors and stakeholders. It's impressive that it can still evolve at all, but it does, and it has some extremely competent people driving it. The big asset for Java beyond everything else is its enormous ubiquity; making it the default option in many scenarios.

C# and .NET have undergone a transition from closed around Windows to open and everywhere. Our relative nimbleness in language and platform evolution, and the new life it has blown into our ecosystem, are certainly a draw for many. There aren't many corners we can't reach from a technology standpoint. Will we? We'll see.

A metapoint about the language landscape is that there is room for many languages to succeed. It is a very different market from two decades ago when Java was sucking all the air out of the room, and language innovation all but ground to a halt across the board. There are cross-cutting trends for nimbleness, pragmatism, and eagerness to evolve. It's an exciting and encouraging stage for languages at this point.

In its evolution since version 1.0, C# gained a lot of new features. Those using it from the start had the benefit of learning the new features incrementally as they were being introduced. Do you think that because of all these new features, C# is a more difficult language for someone who is only starting to learn it now? Are you taking this criterion into consideration when choosing and implementing new features?

There's no question that the C# of 15 years ago was easier to learn. There's also no question that today's C# is much more useful and productive in today's landscape. We try very hard to integrate new language features in a way that is as consistent and smooth and simple as we possibly can. There is probably some complexity that comes from the language having evolved gradually over time, but there also just is more complexity facing developers today than there was twenty years ago. Languages need broader expressiveness to accommodate that, I think.

Most programming languages have a life expectancy, according to John Cook (PhD) C# will die in 2034. One common tipping point is where the language becomes too complex with too many features, could you imagine this happening with C# - if so are we approaching that tipping point?

I can certainly imagine that happening with C#, but I am not sure we are anywhere close. People have said that about C++ for ages, and yet it keeps doing just fine. Our complexity budget is considerably smaller, but I do not see signs of the community starting to fracture over complexity of the language.

We've long ago settled on a strategy here for C#. We'll do what it takes to keep it relevant and modern in the face of the changing demands of the developer landscape. The alternative is falling behind and eventually becoming a legacy language, not a good choice for new projects. That's not a path I see us taking anytime soon.

When it's all over, how do you want to be remembered?

As someone who added light and color to people's lives.

Taj Mahal Picture Credits - Kapil Rao

Thank you Mads! It was such a pleasure interacting with you.

Thank you Interviewers...

Damir Arh

David Pine

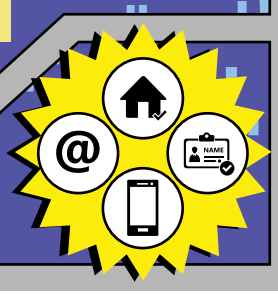
Suprotim Agarwal

Yacoub Massad

Visit Melissa Developer Portal to quickly combine our APIs (address, phone, email and name verification) into custom mashups, and enhance ecommerce and mobile apps to prevent bad data from entering your systems.

With our toolsets, you'll be a data hero – preventing fraud, reducing costs of returned mail and increasing the efficiency of your business processes.

- **Single Record & Batch Processing**
- **Scalable Pricing**
- **Flexible, Easy to Integrate Web APIs: REST, JSON & XML**
- **Other APIs available: Identity, IP, Property & Business**



Let's Team Up to Fight Bad Data Today!

Melissa.com/dnc-fab4
1-800-MELISSA

melissa

Tim Sommer

INTEGRATION TESTING DONE SIMPLE WITH SQLLOCALDB

Using SqlLocalDB to allow simple automated Integration Testing in both greenfield and brownfield projects.

Introduction

I have had multiple discussions about which automated testing process is the best for a variety of applications I have come across in my career.

When you are starting a Greenfield application, the discussion isn't that hard.

However, for Brownfield application development, the discussion gets more heated, when your application is hard to test. Legacy components and bad architectures could cause this. Or maybe the application does not have to perform complex algorithms, does not have complex or clearly defined business rules, or is a complex (or simple) CRUD Application.

In any of these cases, the application will be harder to test. This article is about that last case scenario – automated testing for CRUD application.

How do you test applications where TDD just does not feel right - when Unit Tests provide so little added value (or are so complex) that developers rule out automated testing (which is a shame!).

Because if you abandon automated testing, you abandon the positive and proven impact it has on your code.

First, some concepts – TDD, UnitTesting and Code Coverage.

Not everyone reading this article has the same technical background. So, I'll go over some core concepts, making sure that everyone is up-to-speed.

- **TDD (Test Driven Development):** While this is not an article about TDD, you cannot write about Integration Testing (or automated testing of any matter) without writing about Test Driven Development (TDD).

  TDD is a software development process that relies on the repetition of a very short development cycle. Requirements get turned into specific test cases. After this step, software is written or improved which will allow the new tests to pass.

  TDD provides a short feedback loop. It leads to a better design (SOLID, DRY and KISS principles will be easier to implement) and less production bugs. You also feel much more confident when refactoring and introducing new team members gets much easier.

- **Unit Testing:** This is an automated software testing method by which individual units of source code are tested to determine whether they are fit for use. You test particular pieces of valuable or critical code which results in better architecture, easier maintenance, and better documentation.

  A fairly contested question is: **how large is a unit**? Well, it depends, but experts commonly agreed to keep Unit Tests small and simple. So, the Unit is as small as possible in your code base.

- **Code Coverage:** This is a measure to describe the degree to which the source code of an application is executed when tests are run. If you have a high code coverage (so a lot of unit tests that span across the lines of code of your application), you have a lower chance of running into bugs, compared to applications with lower code coverage. A good deal of different metrics can be used to calculate code coverage, which is measured as a percentage. The most basic is the calculation of the percentage of application statements called during the execution of your tests.

What kind of tests are there?

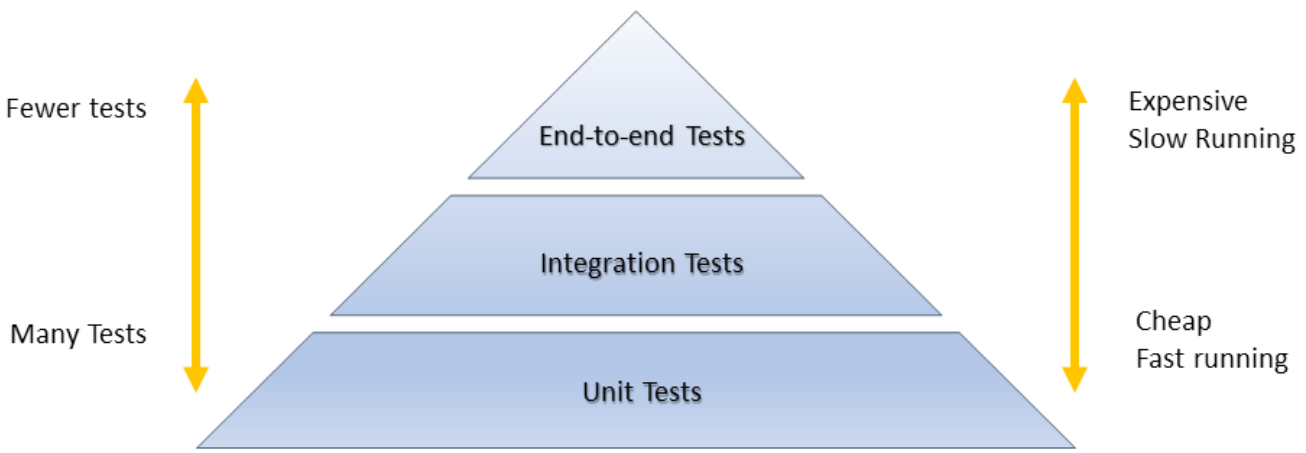


Figure 1: Different categories of tests.

Figure 1 is a simplistic visualization of three different categories of tests. Depending on who you ask, there will be more. But I'm certain almost everyone can agree with these three concepts as a bare minimum.

Starting from bottom to top, tests are cheaper to run. You should have as many tests as possible (or viable).

Where Unit Tests cover smaller code bases, Integration Tests will test multiple modules or systems as they interact together. One Integration Test will result in higher code coverage as there will be more application statements called during the execution of these tests. They are often used to test external services, like databases, APIs, etc.

So, these tests are more expensive (take longer) to run.

End-To-End tests will test the application as a whole, by testing the UI (using frameworks like Selenium, etc.), or calling an API layer. These tests are the most expensive, and you should only write them if they are necessary.

Editorial Note: If you are developing ASP.NET Core applications, read this series of articles (bit.ly/dnc-aspcore-unit) by Daniel which talks about an automated testing strategy involving unit tests, integration tests, end–to-end tests and load tests.

Integration Testing

As we go back to the example stated in the introduction, Integration Tests can be more valuable than Unit Tests. Applications with lots of legacy code, or CRUD applications, will benefit more from Integration Tests.

For the record, always work towards an architecture where TDD or Unit Tests are more viable. I am not disagreeing with that.

But sometimes you have to work with what you've got, when the architecture just doesn't allow high levels of Code Coverage and the project budget does not cover changes in that area.

In this article, I will address the issue of writing code to perform automated testing for CRUD applications, i.e. testing the database. This sample can be applied to brownfield application as well though.

To sum up, the definition of an Integration Test can be stated as follows:

- Like Unit Tests, source code is tested in an automated context.
- For Integration Tests the "Units" under test are larger.
- You test entire systems or multiple modules instead of small code units.
- Integration Tests have to run in an automated context (Continuous Integration Builds).
- Integration Tests take longer to run and can require (external) dependencies.
- Mocking is almost impossible. Because the system under test requires a lot of different modules, or because you want to test the actual results of an external service (like a database).

Can't you Mock your external services?

Well, sure you can!

But in an application as described earlier, the test results will not represent a real-life situation.
Let's say you have an AuditManager (that audits who changed an entity) or you work with "soft deletes" (when you don't physically delete records from a database but use an "IsDeleted" flag to set the deleted state of an object).

You *could* Mock these scenarios, but should you?

Do Unit Tests really provide more code safety in these situations? How will you test these managers as they interact with your repository? Or with your Business Logic layer?

At a certain point, the mocking becomes so complex and is so arbitrary that Integration Testing just makes life easier.

Providing simple Integration Testing with SqlLocalDB

The first goal for this article is to test CRUD features of our database and keep that process simple.

The challenge lies in the latter - providing a Testing Database that can be reused, shared across developers, and deployed on the Build Server.

After reading this article you will be able to add Integration Tests to any project, with only a couple of code changes.

SqlLocalDB – Getting Started

What is SqlLocalDB? Well, according to Microsoft:

"LocalDB installation copies a minimal set of files necessary to start the SQL Server Database Engine. Once LocalDB is installed, you can initiate a connection using a special connection string. When connecting, the necessary SQL Server infrastructure is automatically created and started, enabling the application to use the database without complex configuration tasks. Developer Tools can provide developers with a SQL Server Database Engine that lets them write and test Transact-SQL code without having to manage a full server instance of SQL Server."

This is perfect for what we are trying to achieve!

It is simple, it will run on a Continuous Integration (CI) build, and you don't have to create tables or change anything in your code.

SqlLocalDB Utility

There is a simple command-line tool available to enable you to create and manage instances of SQL LocalDB. I'll list the most important commands you can use below, but you can look at the Microsoft documentation too over here for a full list of features.

- `sqllocaldb create|c ["instance name"] -s` Creates a new LocalDB instance with the specified name. -s starts the new LocalDB instance after it's created.
- `sqllocaldb delete|d ["instance name"]` Deletes the LocalDB instance with the specified name.
- `sqllocaldb start|s ["instance name"]` Starts the LocalDB instance with the specified name
- `sqllocaldb stop|p ["instance name"] -i -k` Stops the LocalDB instance with the specified name. -i requests LocalDB instance shutdown with NOWAIT option. -k kills LocalDB instance process without contacting it.
- `sqllocaldb info|i` Lists all existing LocalDB instances.
- `sqllocaldb info|i "instance name"` Prints information about the specified LocalDB instance

The Sample

So, let's do this!

First, create a new project in an existing solution. Call it "[YourProject].Tests.Integration". Add a ConnectionString in the app.config file to allow our Integration Tests to connect to a new SqlLocalDB instance, which looks something like this:

```
<connectionStrings>
<add name="SomeConnection" connectionString="Data Source=(localdb)\localtestdb;
Database=application_Tests; Trusted_Connection=True; MultipleActiveResultSets=true"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

In our new Integration Test Project, add a Bootstrap class that registers all our modules (using Autofac as IoC container).

If you're not using an IoC container, you can skip this step. We will use MSTest as testing framework throughout the sample. Other popular frameworks like NUnit or xUnit will also work.

```
public static IContainer Bootstrap()
{
 //register the different modules
 var _container = new AutofacContainer(builder =>
 {
  builder.RegisterModule<IntegrationModuule>();
  builder.RegisterModule<CommonModule>();
  builder.RegisterModule<CoreModule>();
  builder.RegisterModule<DbModule>();
  builder.RegisterModule<ServiceModule>();
 });

 _container.Add(_container);

 //set up the ServiceLocator
 ServiceLocator.SetContainer(_container);

 return _container;
}

public static void SetupLocalDb()
{
 // Use a ProcessStartInfo object to provide a simple solution to create a new
LocalDbInstance
```

```
 var _processInfo =
 new ProcessStartInfo("cmd.exe", "/c " + "sqllocaldb.exe create localtestdb -s")
 {
  CreateNoWindow = true,
  UseShellExecute = false,
  RedirectStandardError = true,
  RedirectStandardOutput = true
 };

 var _process = Process.Start(_processInfo);
 _process.WaitForExit();

 string _output = _process.StandardOutput.ReadToEnd();
 string _error = _process.StandardError.ReadToEnd();

 var _exitCode = _process.ExitCode;

 Console.WriteLine("output>>" + (String.IsNullOrEmpty(_output) ? "(none)" : _
output));
 Console.WriteLine("error>>" + (String.IsNullOrEmpty(_error) ? "(none)" : _error));
 Console.WriteLine("ExitCode: " + _exitCode.ToString());
 _process.Close();
}
```

We provide an IntegrationTestBase class like so:

```
[TestClass]
public abstract class IntegrationTestBase
{
 //The Entity Framework DBContext that we will use in our Integration Tests.
 //Set to protected so that child classes can access it.
 protected IApplicationDbContext TestDbContext;

 [TestCategory("Integration")]
 [TestInitialize]
 public void Init()
 {
  try
  {
   #if DEBUG
    Bootstrapper.SetupLocalDb();
   #endif
    Bootstrapper.Bootstrap();

    TestDbContext = ServiceLocator.Resolve<IApplicationDbContext>();

    TestInit();
  }
  catch (Exception _e)
  {
   TestContextInstance.WriteLine(_e.InnerException);
   throw;
  }
 }

 public abstract void TestInit();
}
```

If you are using Autofac, you can use the ServiceLocator pattern to get your DB Context. Otherwise you can

use `Init` Method to build up your context, services you might need, etc.

If the application is run in DEBUG mode, we will also create a new SqlLocalDB TestInstance. You could use the default instance, but following this step ensures you that the instance is available, can be cleaned up, etc.
Using ProcessStartInfo for creating a SqlLocalDB Instance is not ideal!

But to keep everything simple, striving for the least amounts of required code changes, this solution is good enough, for now. Providing a LocalDB abstraction could prove useful, providing more control and abstraction to the programmer.

But, that's basically it! Now you can start writing tests!

Like so:

```
[TestCategory("Integration")]
[TestMethod]
public void Test_User_Seeded()
{
  var _userRepo = ServiceLocator.Resolve<IRepository<User>>();
  var _users = _userRepo.GetAll().ToList();

  Assert.AreEqual(_users.Count, 1);
}
```

This test will call the actual (local) database using a Repository. It fetches all the Users and ensures that exactly one user is present in the database. Since everything is already set up, we could just as easily resolve a (business) service, a controller, anything you want!

The code above assumes that all Migrations have been run prior to testing, and that the database has been seeded. If you are using Entity Framework, you can make this happen like so:

```
public class ApplicationDbContext : DbContext, IApplicationDbContext
{
 static ApplicationDbContext()
 {
   // Provides automatic migration and Seeding
   var _dbMigrator = new DbMigrator(new ApplicationDbContextConfiguration());
   _dbMigrator.Update(null);
 }
}
```

You can run this code in the IntegrationTestBase class as well.

If you are not using an IoC Container and you want to test your context directly, you could write this:

```
[TestCategory("Integration")]
[TestMethod]
public void CheckUserSeeded()
{
```

```
  using (var context = new ApplicationDbContext())
  {
    var _users = context.Users.ToList();
    Assert.AreEqual(_users.Count, 1);
  }
}
```

Or create the context in your TestInit and dispose it in the TestCleanup. This allows you to use different ConnectionStrings if needed. Don't forget to run your migrations, scripts and seeds if you use this method:

```
[TestCategory("Integration")]
[TestInitialize]
public void Init()
{
  TestDbContext = new ApplicationDbContext();
}

[TestCleanup]
public void TestFinish()
{
  TestDbContext.Dispose();
}
```

And that is it, you can now perform actual testing against your local database!

Viewing your Test Explorer gives you that automatic feeling of victory when everything starts turning into green!

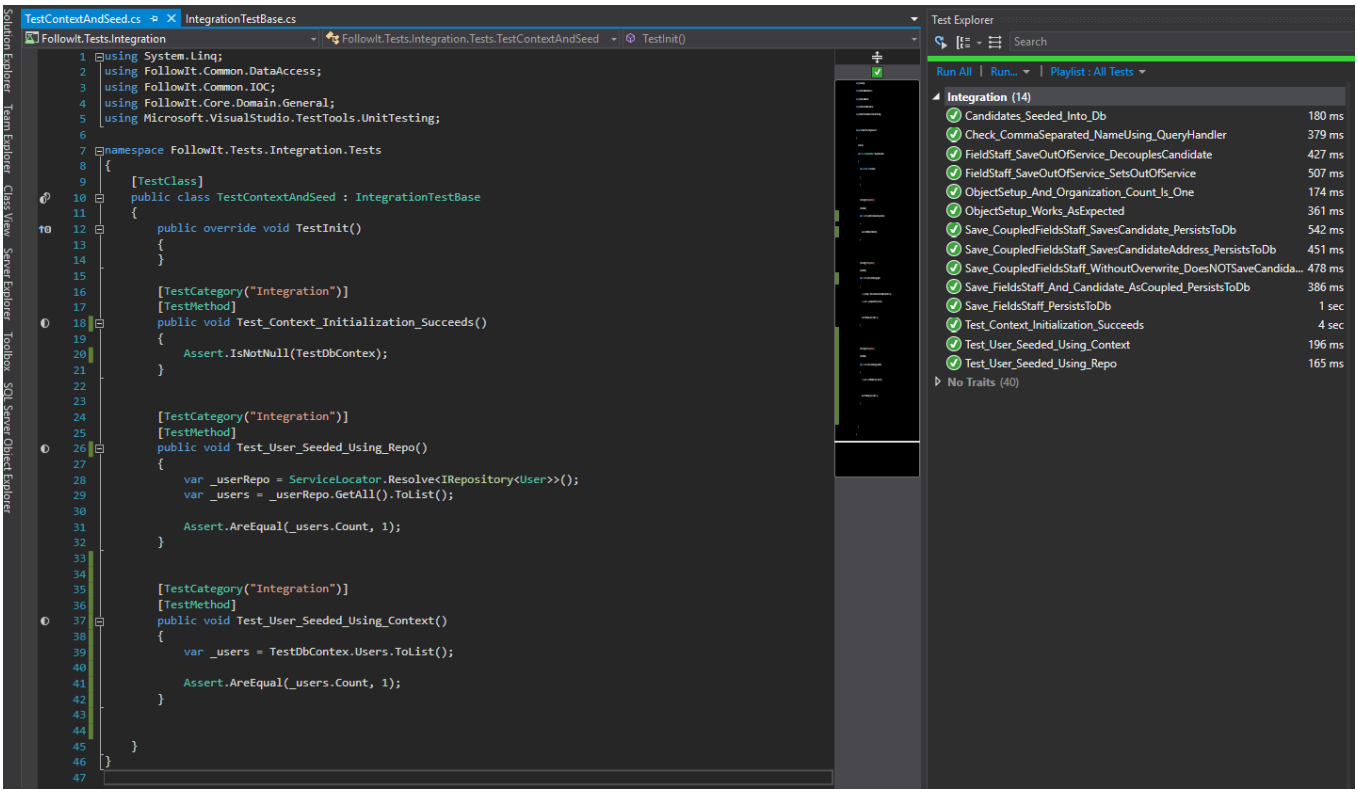


Figure 2: Test Explorer after successful Integration test run

Since we use the `TestCategory` Attribute, we can group these results by trait:

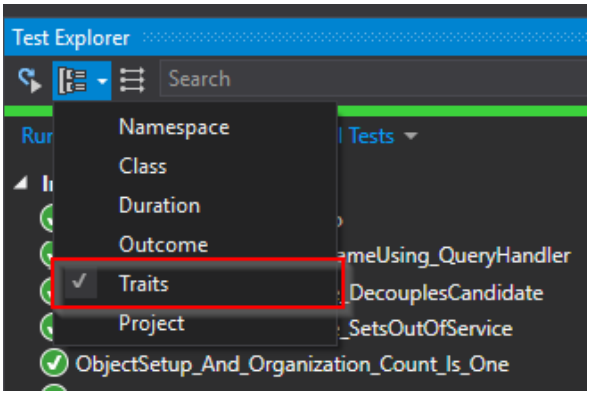


Figure 3: Group test results by Trait

Some additional thoughts

If you are using .NET Core, you can configure Entity Framework to run 'In-Memory', which would be even better and easier than the sample we just saw. But if you are not working with .NET Core, the solution above is extremely valuable.

You can change the `ConnectionString` in code. So, if you want to have a fresh database for different sub-sets of Integration Tests, you can. You could also run into problems when .mdf files already exist. This can be solved by manually deleting the files, or by attaching the database in the `ConnectionString` as shown below:

```
<add name="DataModel.Context" connectionString="Data Source=(LocalDb)\v11.0;Initial
Catalog=database;Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\
database.mdf" providerName="System.Data.SqlClient" />
```

You can view your local database using the **Sql Server Object Explorer**.

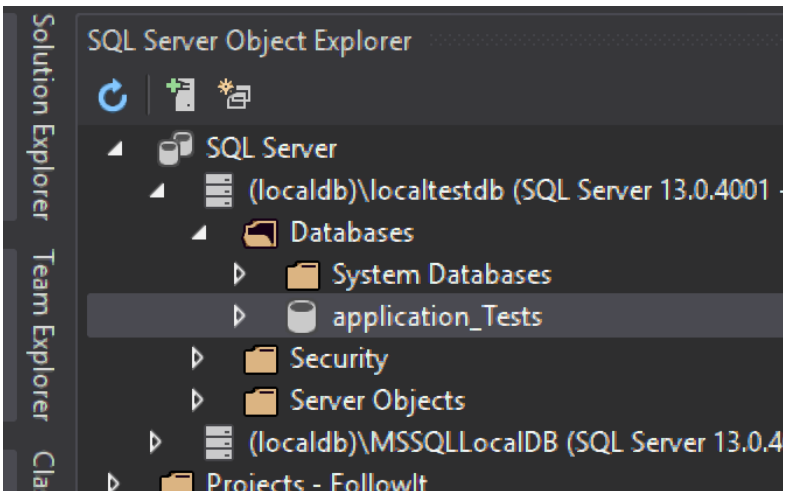


Figure 4: Using Sql Server Object Explorer to examine your local db

Continuous Integration (CI) Builds

The above code sample is a very simple example on how to set up Integration Testing. It is intended as bare-minimum, to keep the scope of this article as small as possible.

But, it provides system and database testing without having to change a lot of code to set it up. So, Goal #1 achieved!

But, we also have to be able to run our new tests in an automated context. We have to be able to configure and run a CI (or Continuous Integration) Build that has a SqlLocalDB instance available and runs the tests against it.

Let's take a look.

Changing existing CI Builds

Integration Tests are more expensive than Unit Tests. That's what we saw earlier. So, it would not be a good idea to run these tests in a "normal" CI Build process.

To prevent that, we need to exclude all Integration Tests in the Build Configuration. In the code samples earlier, you saw the following attribute: `[TestCategory("Integration")]`. This allows us to distinguish Integration Tests from other Unit Tests.

This is important because you commonly have two (or more) CI build definitions - one that runs after each check-in, and the other one that runs once a night.

In the first one, we only want to run the Unit Tests because they are quick in execution and work as a first line of defense. Integration Tests take longer, and you want to run them once a night i.e. in a "CI Nightly Build".

Run only Unit Tests

In the VSTS Build configuration, you can still use the standard "Visual Studio Test" task. You can specify to run all categories **except** the "Integration" category like so:

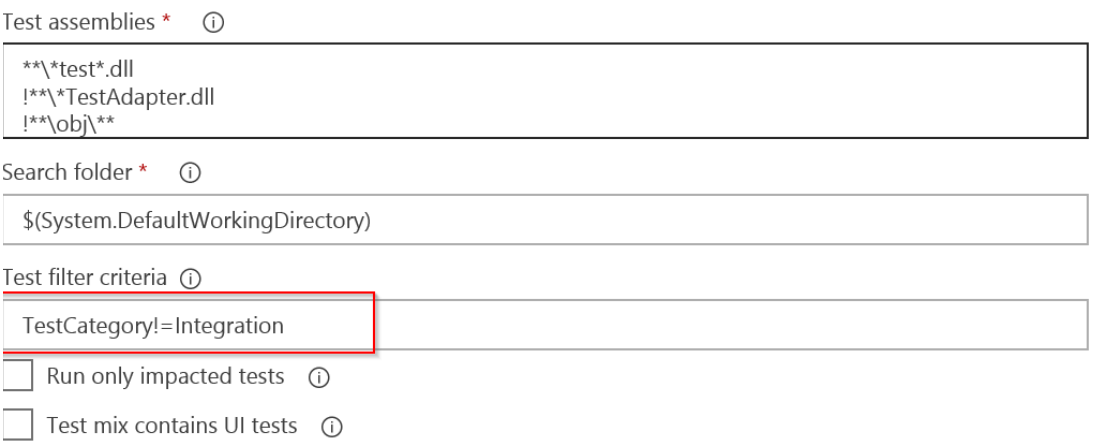


Figure 5: Exclude tests with "Integration" category

Configuring your Integration Nightly Build

For the Integration Build, we will need to do some more configuration to automate the process. The example provided below works on a hosted pipeline, so you can use it for free!

SqlLocalDB is available on VSTS build agents. So, all you need to do is make sure the instance you configured in your ConnectionString is available.

We use "Batch script" tasks to create and manage the SqlLocalDB instances. These are simple Command Line tasks that execute on your Build Agent.

Let's create the instance. Make sure you have the correct SQL Server path available. This may differ between hosted pipelines. If you are using the (free) Microsoft-hosted agents, you can check the installed software here. Depending on the installed version of SQL Server, you have to change the version number in the path `"C:\Program Files\Microsoft SQL Server\130\Tools\Binn\SqlLocalDB.exe"`.

By specifying create `localtestdb -s` we create and start the instance that will be used for our DB tests.

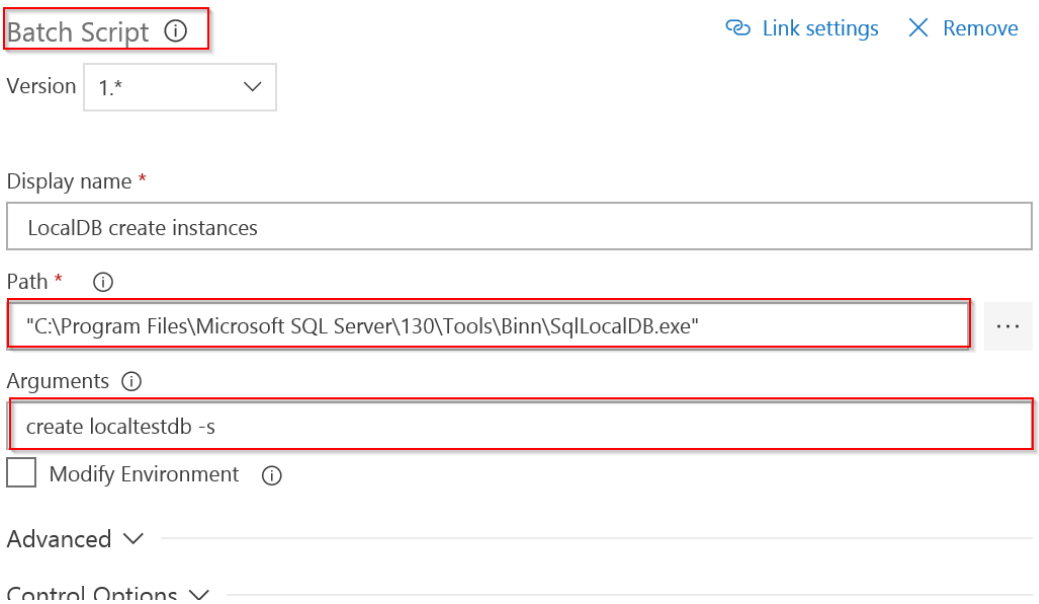


Figure 6: Create Test Instance using Batch Script Task

After this task, we can add the "Visual Studio Test" task and leave it with the default settings. This will run our Unit and Integration Tests.

After this we should clean up our LocalDb Instance. This is not really required, but I recommend it all the same.

Create a new "Batch Script" task which executes SQLLocalDB and configure `p localtestdb -k` as argument. This will stop and kill the instance process without contacting it. After that you can include the last "Batch Script" with `d localtestdb` as argument. This deletes the instance.

A complete Nightly Integration Build configuration would look something like this:

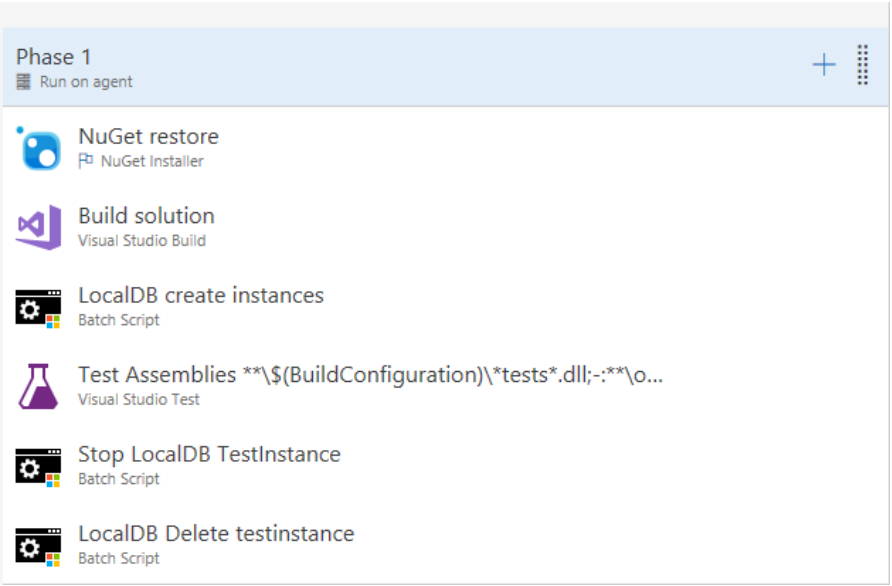


Figure 7: Complete list of tasks for Integration Test Build

Conclusion:

A common myth in development teams is that any sort of automated testing is complex and time consuming, especially in Brownfield application development.

I hope that with this example I've shown you that not only is it not complex, you can add Integration Tests without modifications to your code.

The process is simple, there is almost no pain, and the gain is substantial. The feeling you get as a developer when you see your code coverage sky-rocket, all your tests shining in green, that's just awesome!

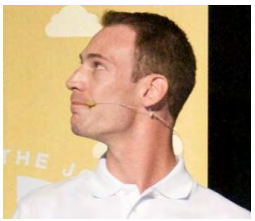
As described in the introduction, we now have a simple and automated way to run Integration Tests on any project. I hope you can integrate this way of working into your development process and experience the gains of having high code coverage!

Microsoft® MVP Most Valuable Professional

Tim Sommer

Author

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



Thanks to Damir Arh *for reviewing this article.*

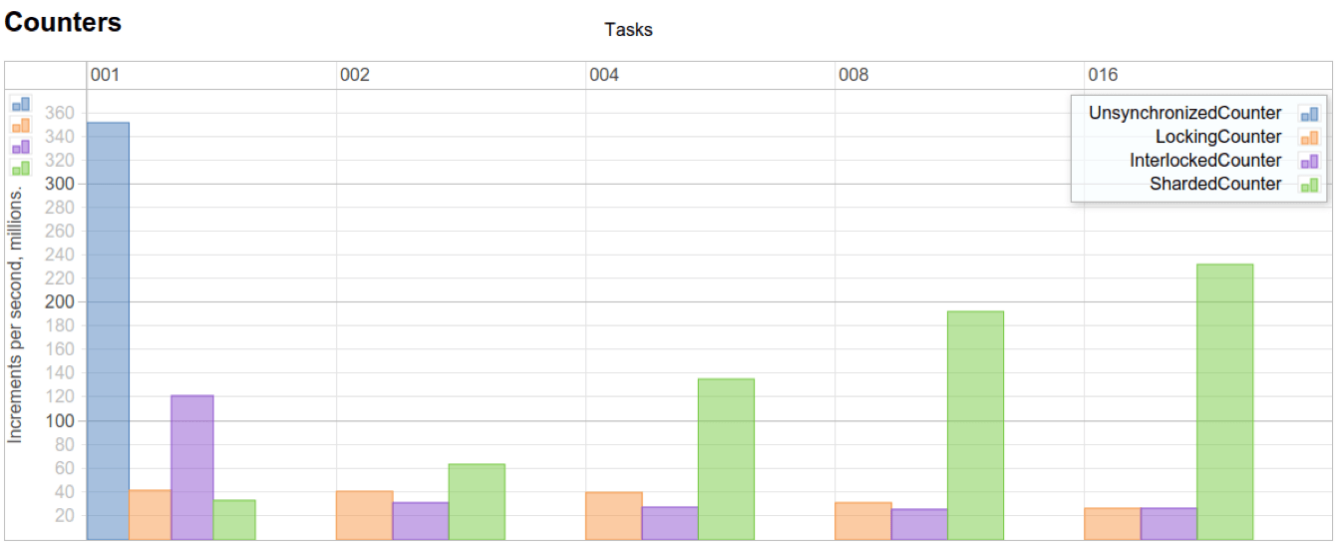
Jeffrey Rennie

A DEEP DIVE INTO SHARDING AND MULTITHREADING

Can database-style sharding improve the performance of a multi-threaded application?

I was working on a sample application that solves Sudoku puzzles. The app solves puzzles very rapidly, on multiple threads. I wanted to track how many puzzle boards had been examined in all. I needed a counter that would potentially be incremented 3,000,000 times per second, so I started thinking about performance.

The performance of the counter was probably not going to affect the overall performance of my application, but it would be fun to see how different implementations perform. To learn more, I implemented the counter in a variety of ways. Then, I ran multiple benchmarks on my machine with 6 CPU cores and 12 virtual CPU cores. To cut to the chase, here are the results. Taller is better.



Each column represents one run of the benchmark. The top axis displays the number of tasks simultaneously trying to increment the single counter. The left axis displays how many times the counter was incremented in 1 second.

So, for example, let's consider column 004. This column shows us that, with 4 concurrent tasks, the `LockingCounter` was incremented just over 40 million times per second, the `InterlockedCounter` was incremented just under 30 million times per second, and the `ShardedCounter` was incremented nearly 140 million times per second.

The `UnsychronizedCounter` only appears in column 001, because the unsynchronized counter does nothing to prevent race conditions in the code. Trying to increment an `UnsychronizedCounter` from multiple threads will result in undercounting. The total count will not be correct. Therefore, it's only appropriate to examine `UnsynchronizedCounter`'s performance when being incremented by a single thread.

The benchmark exercises the worst-case scenario for thread contention: multiple threads in a tight loop competing to read and write the same value. Here's what the inner loop for each task looks like:

```
while (!cancel.Token.IsCancellationRequested)
  counter.Increase(1);
```

So the big question is, **what is a ShardedCounter, and why does it perform better when there are more tasks competing to increase it?**

To understand the answer, let's look at each counter implementation, from simplest to most complex.

UnsynchronizedCounter

The UnsychronizedCounter is as simple as can be:

```
public class UnsynchronizedCounter : ICounter
{
  private long _count = 0;
  public long Count => _count;

  public void Increase(long amount)
  {
    _count += amount;
  }
}
```

The `Count` property returns the private `_count`, and the `Increase()` method increases the `private _count`.

Because the `UnsynchronizedCounter` has zero overhead, it can count faster than any of the other counters, but only on one thread.

If multiple threads simultaneously call `Increase()`, the final count will be less than expected, due to race conditions. Wikipedia has great description of race conditions and how they cause bugs.

LockingCounter

The `LockingCounter` prevents race conditions by holding a lock while reading and writing `_count`.

```
public class LockingCounter : ICounter
{
  private long _count = 0;
  private readonly object _thisLock = new object();

  public long Count
  {
    get
    {
      lock (_thisLock)
      {
        return _count;
      }
    }
  }

  public void Increase(long amount)
  {
    lock (_thisLock)
    {
      _count += amount;
    }
  }
}
```

Image Credit: Shutterstock.com vector ID: 229462564

The lock prevents the undercounting problem that `UnsynchronizedCounter` suffered, but as the benchmark results above indicate, `LockingCounter` is much slower than `UnsynchronizedCounter`.

InterlockedCounter

System.Threading.Interlocked provides atomic operations for values that are shared by multiple threads. For better performance, C#'s concurrent collections use interlocked operations to implement collections like ConcurrentStack. Interlocked operations cannot be used to replace locking in every case, but in the simple case of increasing a counter, they can. `InterlockedCounter` uses `Interlocked.Add()`to increase the count in a way that will never be undercounted, and without blocking while waiting to acquire a lock.

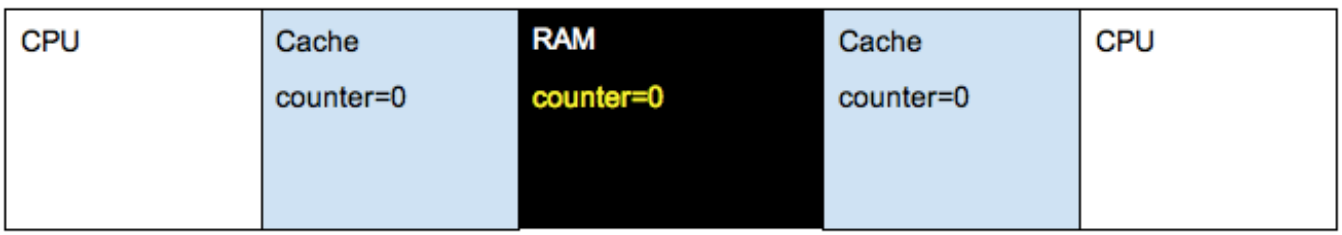
```
public class InterlockedCounter : ICounter
{
  private long _count = 0;
  public long Count => Interlocked.CompareExchange(ref _count, 0, 0);

  public void Increase(long amount)
  {
    Interlocked.Add(ref _count, amount);
  }
}
```

Looking at the benchmark results above, we see that with only a single task, `InterlockedCounter` can count more than twice as fast as `LockingCounter`. That's a huge reduction in overhead. `InterlockedCounter` is the fastest choice when a counter is being increased by multiple threads but there is not a lot of contention.

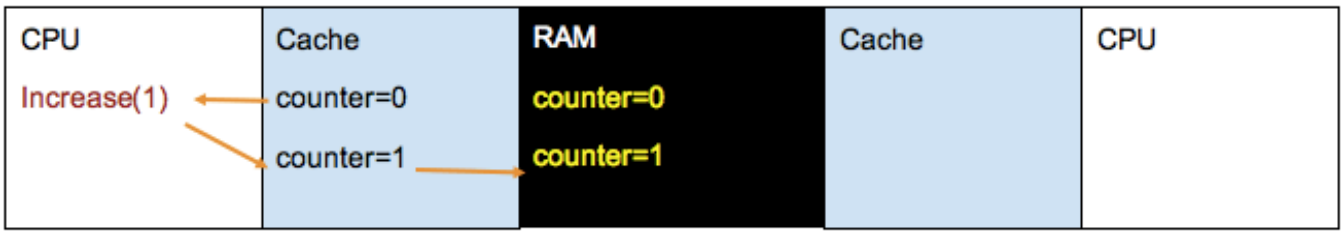
But notice that when multiple threads are trying to increment the counter very quickly, `InterlockedCounter` and `LockingCounter` perform about the same. Why is that? Because both implementations are constantly evicting the value of the counter from CPU caches and loading it again from RAM. Looking up a value in RAM takes at least 10 times as long as finding it in the cache, so evicting the value of counter from the cache is very costly.

Here's a block diagram that illustrates the problem.

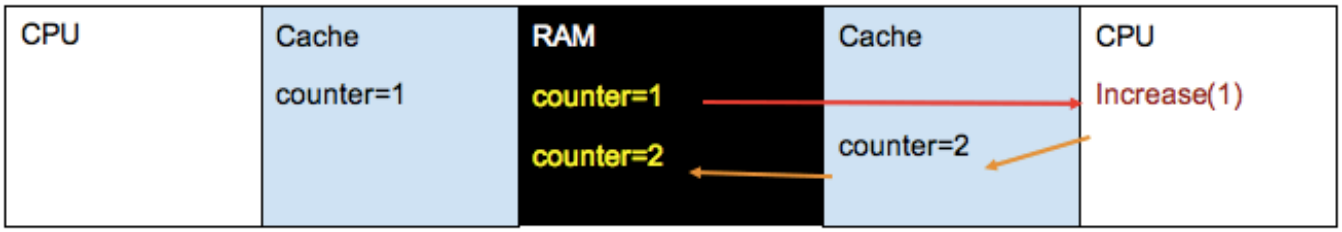
There are 2 CPUs, each with its own cache. Initially, the RAM and both caches store the value 0 for counter.



First, the CPU on the left increases the counter. It reads the counter value from its cache, and writes the new value back to its cache and to RAM. But notice also, the cache on the right no longer contains the value counter=0. The cache entry on the right was evicted because its value was out of date.



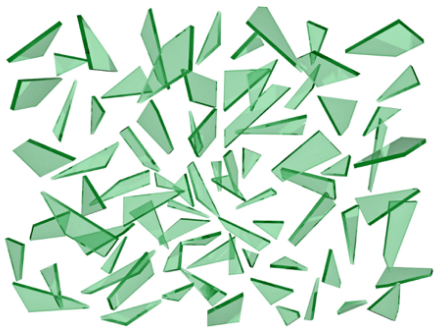
Next, the CPU on the right increases the counter. It has to retrieve the value of counter from distant RAM, as shown by the red arrow, because its cache no longer has the value.



Looking up a value in RAM takes at least 10 times as long as finding it in cache. Reading the value from RAM dominates performance, so that it doesn't matter whether the implementation uses locks or the magic of the `System.Threading.Interlocked`.

Is there anything we can do to avoid the performance bottleneck in both the `InterlockedCounter` and `LockingCounter`? Yes, there is.

ShardedCounter

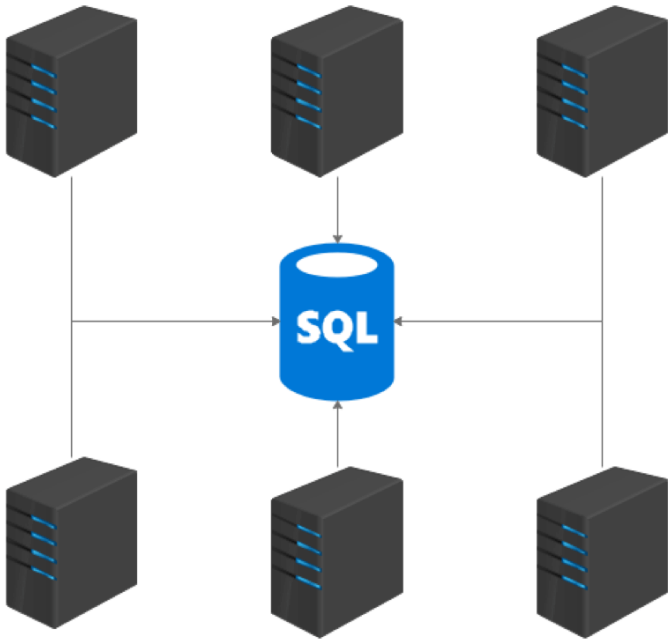


`ShardedCounter` uses the same principle as database sharding, also known as horizontal partitioning.

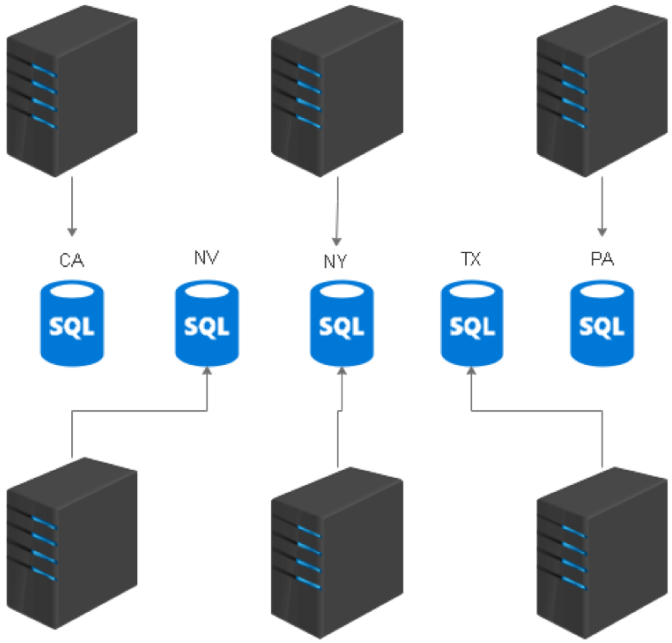
In short, when a database is performing poorly because too many clients are trying to access the same table at the same time, one solution is to break it up into multiple tables across multiple database servers. For example, consider a table of addresses:

State	First Name	Last Name
CA	Luis	Garcia
NY	Joe	Smith

The entire table is stored in one SQL Server, and the server can serve 20 queries per second. When many clients try to access the table at the same time, they are limited to 20 queries per second total. When the load exceeds 20 queries per second, then the clients' requests take longer, and performance of the whole system suffers.



In this situation, it may be possible[1] to improve performance by breaking the one Addresses table into 50 Addresses tables, one for each state:



Because each SQL Server is now handling a fraction of the load, the total throughput has increased to 20 queries per second * 50 SQL Servers = 1000 queries per second.

`ShardedCounter` applies the same strategy for increasing throughput to counters. It breaks up the one counter into multiple counters, one for each CPU core.[2] Each of the counters is called a shard, hence the

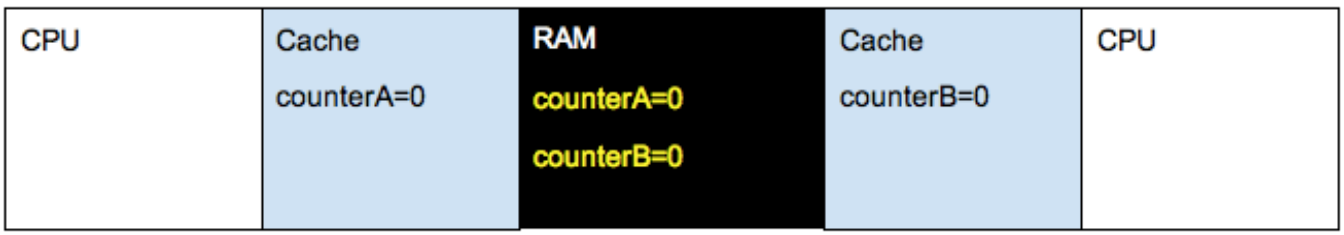
[1] Depending on what kind of queries are being run. In this case, the queries would need to be querying data by state.

[2] Actually, it breaks the counter into multiple counters, one for each *thread*. But since a single thread tends to run on the same core for a while, this approximation makes the performance easier to explain.

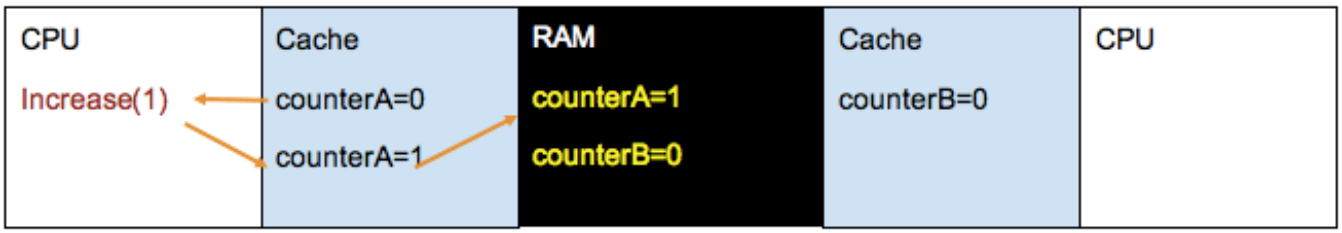
name ShardedCounter.

The idea of sharding counters is an old one. I first saw it in MapReduce. A Google search for "sharded counter" today yields a discussion of sharded counters mainly related to Google App Engine and Google Cloud Datastore. I've seen it used in SQL databases too.

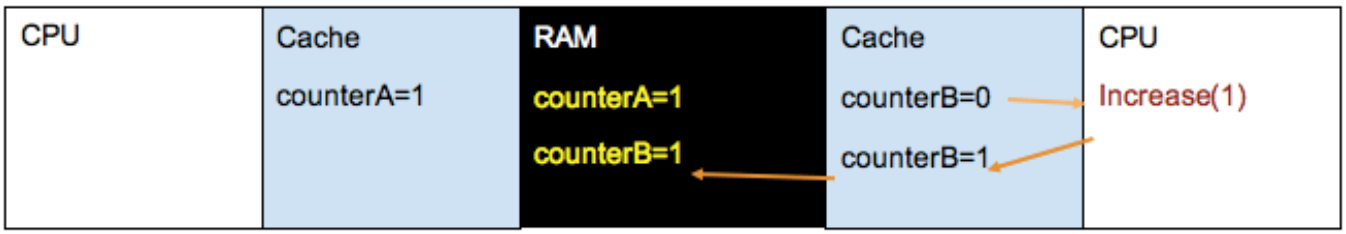
Let's replay the same steps above with the `ShardedCounter`. Initially, there are two counters. Both counters are stored in RAM, and a counter is stored in each CPU cache.



The CPU on the left increments the counter. It reads `counterA` from the cache, and writes the value back to cache and RAM.

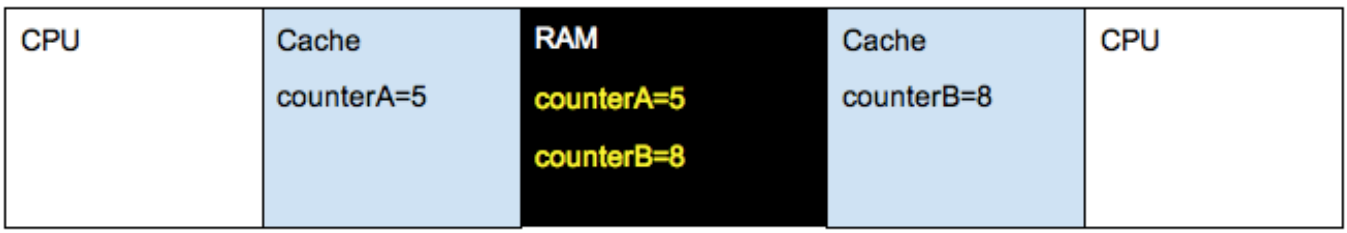


Then, the CPU on the right increments the counter. It reads counterB *from the cache*, and writes the value back to cache and RAM.

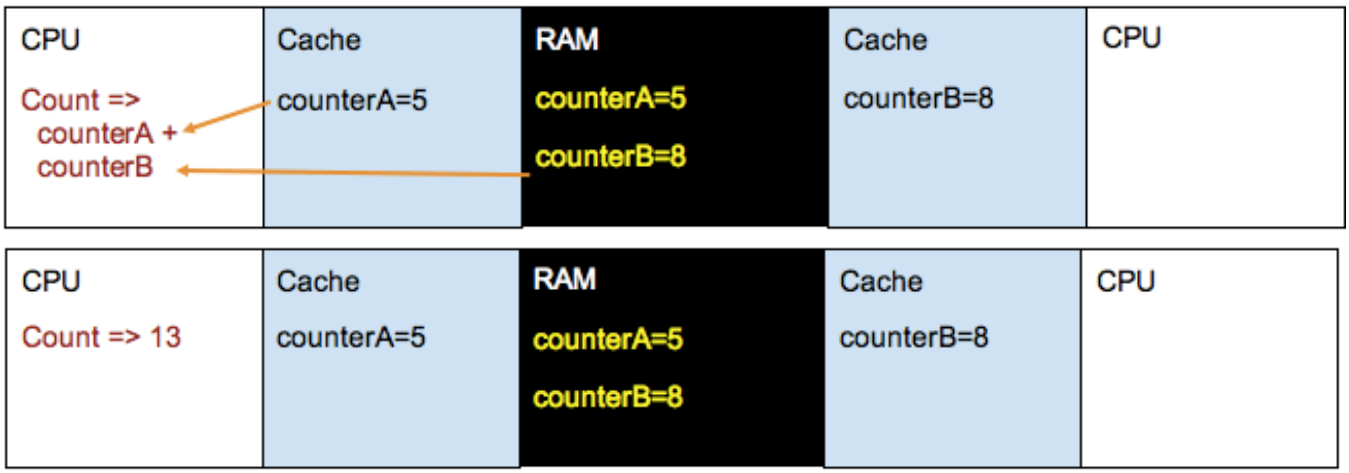


The `ShardedCounter` performs better because the `Increase` operation never[3] reads a value from RAM. It always reads the value from cache.

But of course, at some point, we need to read the total count, and to do that, we must load some values from RAM. The diagram below illustrates how the total count is computed.



[3] Approximately never. Threads get scheduled and descheduled from CPUs. When a new thread is scheduled on a CPU, it may have to read the value from RAM.



So reading the total count is still somewhat expensive. However, in my application, in the benchmark, and in many real world applications, a counter is read far less frequently than it is increased. The benchmark code reads the counter once per second but increases the counter millions of times per second. Therefore, the cost of an increase dwarves the cost of a read.

How does `ShardedCounter` create one counter per core? Technically, it doesn't. It creates one counter per thread. Because threads tend to run on the same core for a while, the effect is similar.

ShardedCounter allocates a new thread-local storage slot via Thread.AllocateDataSlot(). This creates a place to store the counter shard for each thread.

```
public class ShardedCounter : ICounter
{
  // Protects _shards.
  private readonly object _thisLock = new object();

  // The list of shards.
  private List<Shard> _shards = new List<Shard>();

  // The thread-local slot where shards are stored.
  private readonly LocalDataStoreSlot _slot = Thread.AllocateDataSlot();
```

Retrieving the count requires summing the counts in all the shards.

```
public long Count
{
  get
  {
    // Sum over all the shards.
    long sum = 0;
    lock (_thisLock)
    {
      foreach (Shard shard in _shards)
      {
        sum += shard.Count;
      }
    }
    return sum;
  }
}
```

In the fast, common path, increasing the counter requires no locks and only reads and writes a value that

no other thread can read or write. Therefore, there's little risk of another CPU trying to read the value and fetching it from RAM.

```
  public void Increase(long amount)
  {
    // Increase counter for this thread.
    Shard counter = Thread.GetData(_slot) as Shard;
    if (null == counter)
    {
      counter = new Shard()
      {
        Owner = Thread.CurrentThread
      };
      Thread.SetData(_slot, counter);
      lock (_thisLock) _shards.Add(counter);
    }
    counter.Increase(amount);
  }
```

Each shard is an `InterlockedCounter`, so that `Count` sees the latest values of all the counters, and thus avoids undercounting.

```
  private class Shard : InterlockedCounter
  {
    public Thread Owner { get; set; }
  }
}
```

The complete code, which is a little more complicated to clean up after threads that have finished, can be found here.

Conclusion

When concurrency issues slow down your code, sometimes using more sophisticated concurrency operations like those provided by System.Threading.Interlocked will not improve performance. The problem is that too many actors are trying to access the same value at the same time.

In this particular instance, it was a toy problem that would do little to affect the overall performance of the application. But this same technique of sharding a fought-over value can be applied to larger problems too. Sharding a value is especially easy when it is calculated with purely associative operations, like addition and multiplication, when the order of operations does not affect the result.

The full code for the counters benchmark and the sample application is available on github.com.

Jeffrey Rennie

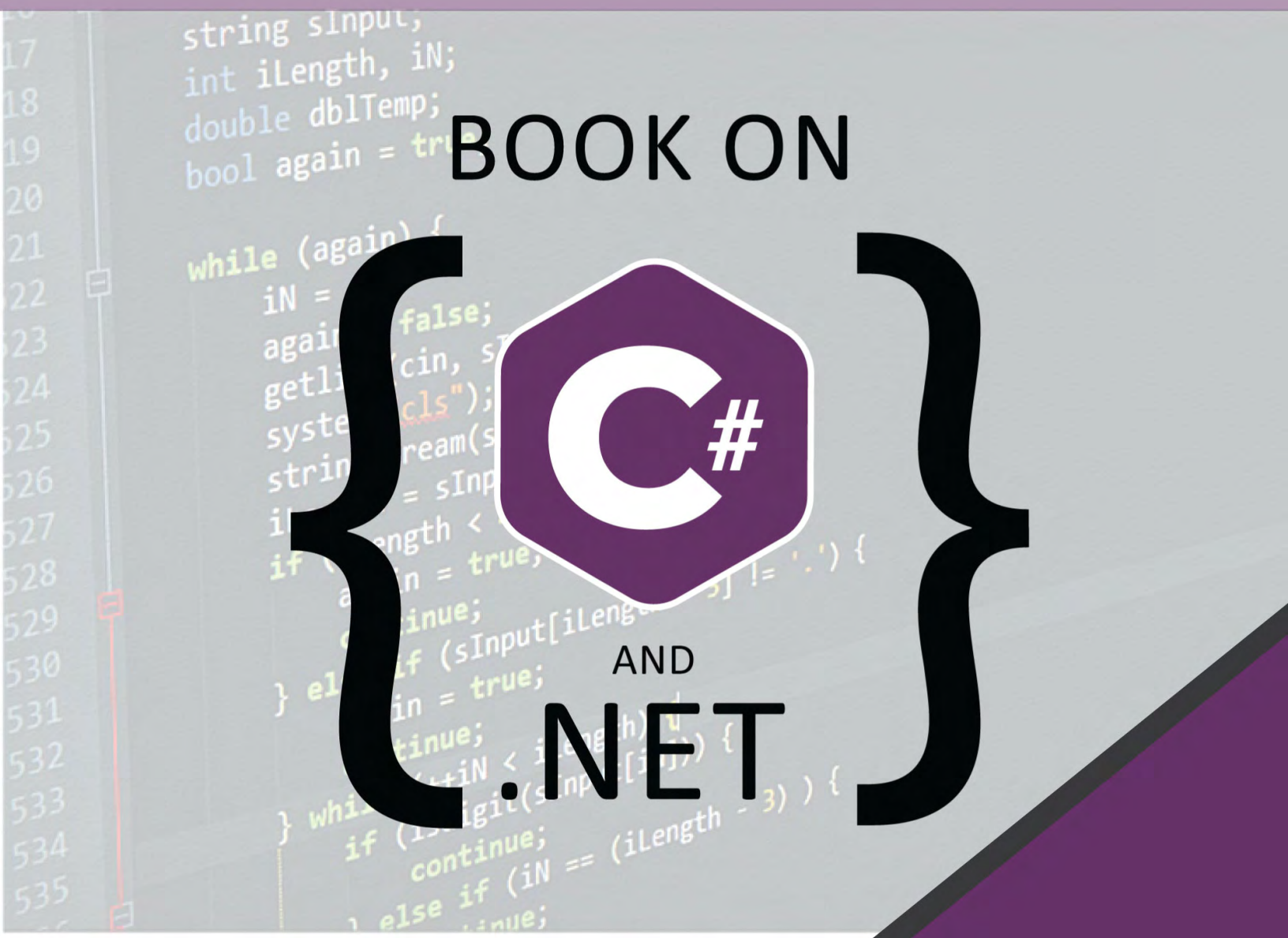
Author

Jeffrey Rennie works as a Developer Programs Engineer for Google. He frequently posts to medium, stackoverflow, and github.

Thanks to Damir Arh for reviewing this article.

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

THE ABSOLUTELY AWESOME BOOK ON C# AND .NET



DAMIR ARH

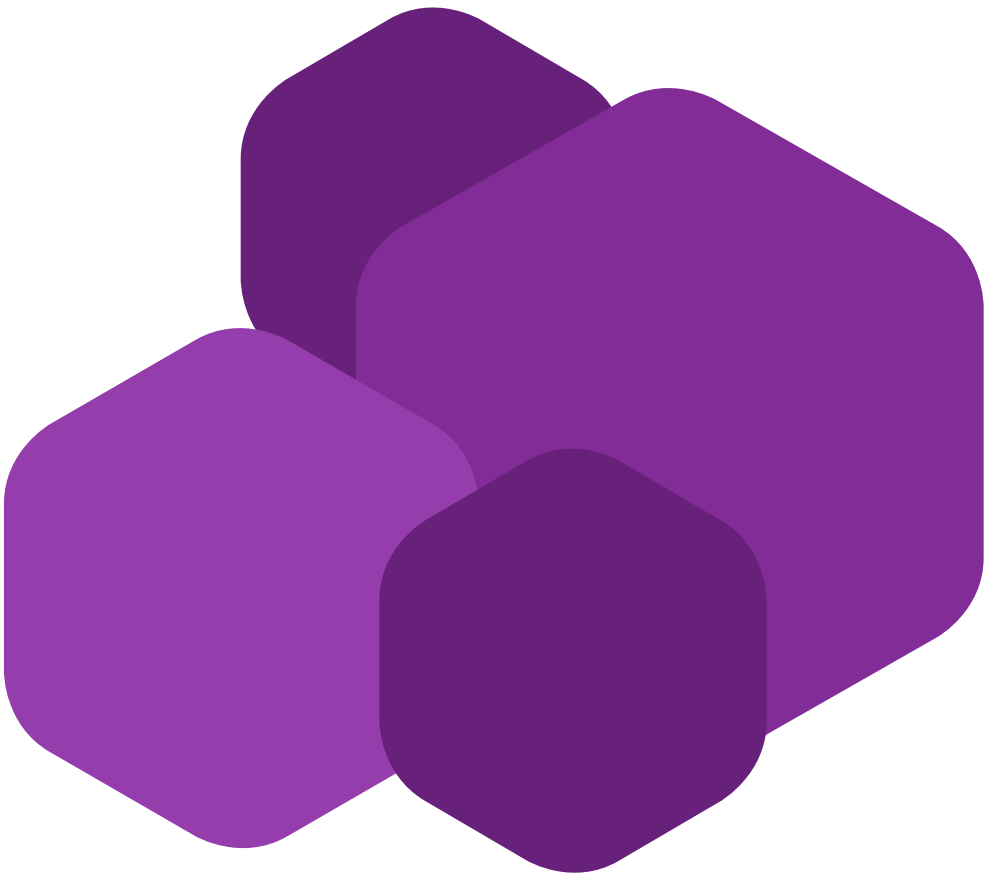
Learn more...

CODING PRACTICES

Yacoub Massad

Writing Honest Methods in C#

This tutorial describes approaches for making methods/functions more honest. A more honest method makes it easier for readers to understand what the method does by reading its signature, i.e., without reading its implementation.



Introduction

We developers spend lot of our time reading code. We read code so that we know how to change it to implement a new feature, fix a bug, etc. It is much better to work in a code base where code is easy to read and understand.

One thing that can make code readable is good naming. Giving variables, classes, and methods good names makes code easier to read.

But why is this true?

Consider the following method:

```
public static Discount CalculateDiscount(Cart cart, Customer customer)
{
  decimal discountRatio;

  if (customer.Status == CustomerStatus.Normal
    || customer.NumberOfTimesStatusUsed > 10)
  {
    discountRatio = 0;
  }
  else if (customer.Status == CustomerStatus.Silver)
  {
    discountRatio = 0.05m;
    customer.NumberOfTimesStatusUsed++;
  }
  else if (customer.Status == CustomerStatus.Gold)
  {
    discountRatio = 0.1m;
    customer.NumberOfTimesStatusUsed++;
  }
  else
  {
    throw new Exception("Invalid CustomerStatus");
  }

  var discount = discountRatio * Cart.GetTotalAmount(cart);

  return new Discount(discount);
}
```

Consider the `discountRatio` variable for example. What would we lose if we renamed it to "x"? Well, if we did so, the reader of the code will need to spend more time to understand what it represents.

Currently, as the reader reads the method and sees the statements that assign values to the `discountRatio` variable (lines 7, 11, and 16), she/he will immediately realize that the purpose of the code (lines 4-22) is to calculate the discount ratio.

Keeping the variable name as "x", will only leave the reader wondering what it means. Maybe the fact that it is assigned the values of 0, 0.05, or 0.1 will make him/her guess that this is the discount ratio. To be sure, the reader has to reach line 24 where the x variable is used. Because this variable is multiplied by the total amount in order to calculate the discount, the reader can infer that x is actually the discount ratio.

What about method names? How can they be helpful?

When the reader is reading code that invokes a method, a good method name will enable the reader to understand this code.

For example, look at the `GetTotalAmount` method in line 24 in the last example.

Reading the name of the method, the reader can understand immediately that the code is trying to get the total amount for the items in the cart. This allows the reader to continue understanding this line; that the discount is calculated by multiplying the discount ratio with the total amount.

The reader does this without the need to go to the implementation of the `GetTotalAmount` method to see how it is implemented. The name itself gave enough information about what this method does.

This ability to give the reader enough confidence about what a method does **without having to read its implementation** (by reading its signature alone) is very much desirable.

Note: *the signature of a method includes its name, its parameters, and its return type.*

Although naming is important, it is not always the best way to give the reader such confidence. Consider for example the `CalculateDiscount` method. When the reader sees this method called, for example as in this line:

```
var discount = CalculateDiscount(cart, customer);
```

..he/she will understand the purpose of this line; i.e., calculating the discount that the customer gets for the items in the cart. However, there are some missing details that the reader will not be able to know unless he goes and reads the implementation.

Can you guess what are these missing details?

If you go back and take another look at the implementation of the `CalculateDiscount` method, you will notice that if the customer has `Silver` or `Gold` status, a property on the `Customer` object named `NumberOfTimesStatusUsed` will be incremented. This property is used to keep track of how many times the customer has used Silver or Gold status. Customers with Silver or Gold status can use such status for discount for a maximum of ten times.

We can say that the `CalculateDiscount` method is dishonest. It does things that are not declared in its signature.

Of course, we can rename the method to `CalculateDiscountAndUpdateNumberOfTimesStatusUsed`. This will fix the problem. In theory, we can depend on naming alone to make methods honest. We simply state what the method does in its method name, including any hidden details.

In practice, however, we cannot rely only on method names for the following reasons:

1. Methods are easily renamed. A developer might change a method name for any reason, potentially making the method dishonest.

2. In order to be completely honest, a method name would need to declare all the things that it does. This would make some method names very long.

3. A method can call other methods, which themselves call other methods, etc. For the top-level method to be honest, its name should declare all the things that all the methods it calls (directly or indirectly) do. How long would the top-level method name become?

Before going into other techniques for making methods more honest, let's first talk about what methods do and which parts of that they need to declare, in order for us to consider them honest.

The concept of method honesty is an informal one. Having said that, I think the following is the list of things a method should declare in its signature in order to say it is honest (let's call this list the honesty list):

1. Any changes to global state. For example, this could be mutating a global variable that keeps track of the total number of documents the application has processed so far.

2. Any change to the state external to the application. For example, this could be adding a record to the database. Or modifying a file in the file system.

3. Any mutation to method parameters. The `CalculateDiscount` method is an example of a method that mutates one of its parameters.

4. How the return value is generated. This can be divided into two:
   4-a. Any effect on the return value of the method that depends on global or external state. For example, this includes reading from a global variable, reading from the file system, or reading the current time using the system timer.

   4-b. Everything but the effects described in 4.a. This is equal to:
      i. The effect of the input parameters on the return value
      ii. The knowledge that the method contains.

To explain this point, I am using the following example:

```
public static int DoSomething(int input)
{
  return input + 1;
}
```

In this method, the output is equal to the input parameter, plus one. The effect of the input parameter on the return value is that it participates in the addition operation that eventually produces the output. The "+ 1" part is the knowledge this method contains.

This method does not mutate anything, and it does not depend on anything external to generate its return value. Therefore, only 4.b is relevant to the honesty of this method.

To make this method honest, we can simply rename it like this:

```
public static int AddOne(int input)
{
  return input + 1;
}
```

Now, we can easily argue that the effect of the input parameter on the return value is clear from the signature (4.b.i); i.e., it is the value that 1 is going to be added to. We can also easily argue that the

knowledge that the method contains is also clear (4.b.ii); i.e., adding one (to the input).

Because this method mutates nothing, and its return value does not depend on any global or external state, we can consider this method to be **pure**.

For more information about purity, see the Functional Programming for C# Developers article.

I will first talk about honesty for pure methods, and will then discuss honesty related to impure methods.

Making pure methods more honest

Let's assume for a moment that when we read code and see a method invocation, we can immediately tell if the method is pure or not without needing to go and read the method implementation.

For example, imagine that there is a feature in Visual Studio that colors the method invocation with a specific color (say green) to indicate to the reader that this method is pure. Now, when you read code and see a method invocation in green, you know the method is pure, and therefore you don't need to worry about points 1, 2, 3, and 4.a from the honesty list. Although you do need to keep 4.b (the effect of the input parameters on the return value, and the knowledge of the method) in mind.

Here are some techniques that can help make pure methods honest:

Make the return type of the method declare all the possible outputs of the method.

Consider the following method signature:

```
public Customer FindCustomer(
  ImmutableArray<Customer> customer,
  string criteria)
```

This method takes in an array of customers, and search criteria, and it returns a matching customer object.

But what happens when no customer object in the array matches the specified criteria?

Does the method return null? Or does it throw an exception? Or worse, could it return a customer that has some `Id` property set to 0?

To answer these questions, we have to look at the implementation of the method.

The problem with this method is that the return type of the method does not declare all the possible outputs. It just declares that it returns a "Customer".

Here is a better signature:

```
public Maybe<Customer> FindCustomer(
  ImmutableArray<Customer> customer,
  string criteria)
```

The `Maybe` type is a type that represents an optional value; it either has a value or it does not. This is similar to the nullable type feature in C# that is available for value types.

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

Now, we can answer the above questions without the need to read the implementation of the method. We know now that in case no customer matched the criteria, the method will return `Maybe<Customer>` that does not have a value.

It is still possible that the method above throws an exception when the customer is not found, or even return a `Maybe<Customer>` that contains a customer with some Id property set to 0.

If this happens, then the method signature is lying.

The problem is no longer that the method signature does not answer some questions. Now, the problem is that the method signature provides deceitful answers. This is much worse.

This means that developers working on a codebase are required to have some minimum level of discipline as to not make such mistakes. A developer who reads a signature of the method should safely assume that it does not deceive her/him. The best thing to have is a signature that answers all the questions correctly.

However, a signature that leaves some questions unanswered is better than one that gives deceitful answers.

I don't think there is another solution to this problem. Consider for example that the method above is implemented in such a way that it returns some `Customer` object with random data without even searching the list of customers. What prevents a developer from doing so?

By the way, C# 8 is expected to have nullable support for reference types, this would allow us to write the method signature like this:

```
public Customer? FindCustomer(
  ImmutableArray<Customer> customer,
  string criteria)
```

Make the method parameter types declare all possible input values.

Consider for example that you find the following invocation of the `FindCustomer` method:

```
var customer = FindCustomer(customers, "TotalPurchases > 1000$");
```

From the signature of the method, you know that the second parameter is the search criteria. From the argument passed in this specific invocation, you know that the search criteria supports filtering based on total purchases of the customer.

But what other ways of filtering does the search criteria support?

For example, can we search by the number of purchases in a specific month? Does the search criteria support the greater than or equal to operator (>=)? Can we have multiple filtering conditions? Also, does the method throw an exception if the search criteria is not supported? Or does it silently return a `Maybe<Customer>` with no value?

The answers for these questions will probably be scattered all over the place inside the implementation of the `FindCustomer` method and the many methods that it probably calls.

To make things better, we can change the type of the second parameter (the criteria parameter) so that we cannot pass values for such parameter that are not supported.

Consider this updated signature:

```
public Maybe<Customer> FindCustomer(
  ImmutableArray<Customer> customer,
  CustomerSearchCriteria criteria)
```

Now with this change, any random string cannot be passed as the criteria. Only valid instances of the `CustomerSearchCriteria` class can be passed.

We should use the type system of .NET to make sure that the `CustomerSearchCriteria` class does not allow us to express search criteria that are not supported.

In a previous article, Designing Data Objects in C# and F#, I talked about making invalid states unrepresentable. What we need to do is make invalid search criteria unpresentable by the `CustomerSearchCriteria` class. Please read that article for more details.

Here I will show the definition of the `CustomerSearchCriteria` in F# as it makes it easier for the developer reading the code to quickly understand supported cases.

```
type TotalPurchasesConditionsOperator =
    | Equals
    | GreaterThan
    | SmallerThan

type TotalNumberOfPurchasesInPeriodOperator =
    | Equals
    | GreaterThan
    | SmallerThan
    | GreaterThanOrEquals
    | SmallerThanOrEquals

type CustomerSearchCriteria =
    | TotalPurchasesCondition of
        operator: TotalPurchasesConditionsOperator *
        amount: Money
    | TotalNumberOfPurchasesInPeriod of
        operator: TotalNumberOfPurchasesInPeriodOperator *
        startDate: DateTimeOffset *
        endDate: DateTimeOffset *
        number: int
    | Any of conditions: ImmutableArray<CustomerSearchCriteria>
    | All of conditions: ImmutableArray<CustomerSearchCriteria>
```

This code defines three F# discriminated unions:

- CustomerSearchCriteria (4 cases)
- TotalPurchasesConditionsOperator (3 cases)
- TotalNumberOfPurchasesInPeriodOperator (5 cases)

These types become part of the signature of the method because the method's second parameter is of type `CustomerSearchCriteria`. From these types, we can know that the method supports searching based on the customer's total purchases, and also on the number of purchases in a specific period.

For the total purchases condition, three operators are defined (equals, greater than, and less than). For the other condition type, five operators are supported.

The `CustomerSearchCriteria` type also shows that we can filter based on the total number of purchases in any period defined by starting and ending dates. Also, we can see from the definition that we can combine conditions using `Any`, or `All`. The method now is much more honest.

The reason I used F# here to define the types is that in C#, we would need much more code to express the valid states of `CustomerSearchCriteria`. See the article Designing Data Objects in C# and F# for more details.

Note that there could be a case where the criteria comes from the user as a string. I.e., the user types the search criteria in some text box. In this case, there should be a special method to parse such string and generate a corresponding `CustomerSearchCriteria` object like this:

```
public Maybe<CustomerSearchCriteria> Parse(string criteria)
```

This specialized method returns no value if the criteria does not represent a valid criterion, or a `CustomerSearchCriteria` object otherwise.

We cannot depend on the .NET type system here to make sure that the caller of the method passes valid search criteria. To make the `Parse` method signature more honest, we can include some documentation about the syntax of the criteria string as Xml Documentation Comments. See this reference for more details.

Make the method name describe what it does.

One question a reader might ask about the `FindCustomer` method is: What happens if there are two customers that match the criteria? Does the method return the first customer? Does it throw an exception?

We can answer this question using the method signature by updating the method name. Assuming that the method returns the first customer that matches the criteria, we can rename the method to `FindFirstCustomer`.

A method name can become long. In many cases, this is not an issue. It is better to read a method name that contains ten words and understand what it does, than to go to the implementation of the method and read ten lines of code (or even more if the method calls other methods whose names are not descriptive enough).

Please note that a method name does not need to declare things that the method parameters already declare.

For example, consider the `CalculateDiscount` method from the first example in this article. There is no need to call it `CalculateDiscountForItemsInCartForASpecificCustomer`. The `cart` and `customer` parameters already make it clear that such discount calculation is related to the cart and to a specific customer.

Another way to decrease the length of method names is to learn more about the domain. Sometimes, you can learn a new term that enables you to replace a few or several words in the method name, with one or two words.

Making impure methods more honest

Although pure methods are preferred over impure ones, without impure code, programs cannot do anything useful. We need impure code to read from and write to the console, the filesystem, the database, read the system timer, send a web request, generate a random number, etc.

Having said this, there are approaches to maximize pure code and minimize impure code. Mark Seemann has introduced one such approach which is called *Dependency Rejection*. You can read about it here: http://blog.ploeh.dk/2017/01/27/from-dependency-injection-to-dependency-rejection/

The basic idea of this approach is that we should design the core functionality of our applications to be completely pure.

When the program receives a request to do something, e.g. handle a Web API request, or handle a UI command, etc., the program does the following:

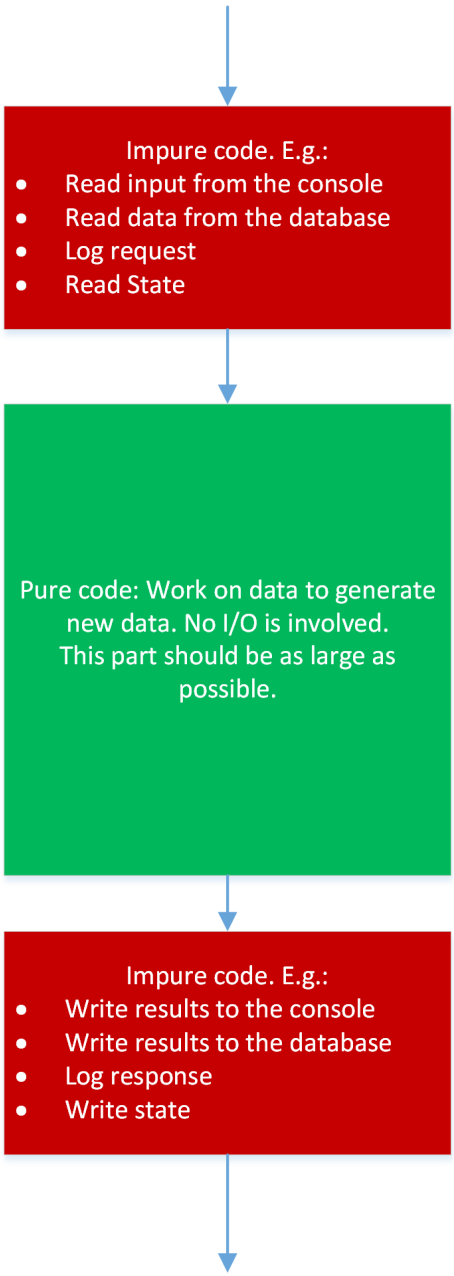


Figure 1 Dependency Rejection

1. It runs some impure code to prepare the data that will be given as input to the pure functionality. For example: read the list of orders for the customer whose id is specified in the Web API request from the database.

2. It runs the pure functionality over that input and collects the output of such pure functionality. For example: filter the orders to include only the ones whose total amount is larger than 1000$, group them by month, and then generate a report file (in memory) in PDF format from such data.

3. It runs some other impure code given the output from the pure functionality. For example: write to the log file that report XYZ has been requested for Customer X and return the report as a Web API response.

The adjacent figure represents this. Because the pure part is preceded and followed by impure parts, Mark Seemann calls this the impure/pure/impure sandwich.

The impure parts should be very simple. They should not contain much logic. All the business logic should be inside the pure part. If we can manage to have our applications designed like this, then we shouldn't care so much about the impure parts because they are very small. If the methods in the pure part are honest, then all is done.

However, there are cases where the program cannot be represented by the impure/pure/impure sandwich. In some programs, we need to run impure code frequently as we are processing the business logic.

For example, what if based on the orders a customer has placed in the last month (count, total amount, etc.), we need to talk to a different web service to retrieve some data required to generate the report?

Should we talk to all web services in the first impure part so that all data is ready for the pure part to select from? Or should we move such business logic (having different data in the report based on the orders) to the first impure part?

Both solutions have their problems.

Even in such programs, there are solutions to separate pure and impure code. Mark Seemann talks about one solution here: http://blog.ploeh.dk/2017/07/10/pure-interactions/

I am not going to talk about such a solution in this article. Instead, I am going to assume that we need to mix between impure and pure code.

Figure 2 Mixing pure and impure methods

In this figure, the green methods are pure, the red methods are impure, and the orange methods are "pure" methods that became impure because they call impure methods.

Let's explain this in more details:

- Green methods are completely pure. They contain pure logic.

- Red methods are very simple methods. They contain no business logic. They simply perform I/O, read global state, and/or update global state. An example of such methods is the `File.ReadAllBytes` method from the .NET framework. This method returns the binary content of a file, given its filename. One might argue that the `File.ReadAllBytes` method contains some logic. After all, such method and the methods it calls have to deal with a lot of concerns, like detecting if the file exists, authorizing the caller, handling the file system structure, etc. When I say that this method contains no logic, I mean it does not contain logic related to the problem the application is trying to solve. As far as the application is concerned, the `File.ReadAllBytes` "simply" reads the contents of the file.

- Orange methods contain logic as green methods do. But because they need to interact with the outside world as they are processing, they call impure methods. Going back to the example of contacting different web services based on customer orders, this logic would be in an orange method.

Let's talk about honesty of orange methods. How to make such methods honest?

The points I talked about for pure methods are still valid here. However, we need impure methods to tell us in their signatures about the *impurities* that they have.

Consider this example:

```
public XYZReport GenerateXYZReport(Customer customer, ImmutableArray<Order> orders)
```

This method generates some report (called the XYZ report) for some customer. The body of the method (not shown here) invokes some impure methods to collect some data.

Currently, the method signature does not declare that fact.

This can be partially fixed by the imaginary feature of Visual Studio that gives method invocations different colors based on whether the methods being invoked are pure, or not.

Even if you know by looking at the method invocation color that the method is impure, you don't know exactly what it does. Does it invoke a web service? Does it write to some file on the file system? Does it read the system timer? Does it update or read some global variable? Does it mutate any parameters?

What we can do is convert this method into a pure method.

Let's assume that the method above reads the current time using the system timer to include it in the report. Instead of making the method read the system timer itself, we can delegate such responsibility to the caller and make this method take the current time as a parameter like this:

```
public Report GenerateXYZReport(
  Customer customer,
  ImmutableArray<Order> orders,
  DateTime currentTime)
```

This is a simple application of the *Dependency Rejection* concept. The GenerateXYZReport method no longer depends on the system timer.

As explained before, this is not always possible.

Assume for example, that the report generation takes 30 minutes to complete. Different parts of the report will be generated at times that are minutes away from each other. Assume that it is important for business that each part includes the exact time in which it was generated. This means that during generation, we need to read the system timer at different times. We cannot simply take a simple value at the beginning of the generation process.

Consider this updated signature:

```
public Report GenerateXYZReport(
  Customer customer,
  ImmutableArray<Order> orders,
  Func<DateTime> readCurrentTime)
```

Now, the method takes a function as a parameter. In functional programming terms, this is called a higher order function. When the method needs to read the time, instead of invoking the `DateTime.Now` impure property getter, it invokes the `readCurrentTime` parameter to obtain the current time.

If the GenerateXYZReport function does not call any other impure methods (other than reading the system timer), then we can say that this function, in its current form, is a potentially-pure function.

It is potentially-pure because its purity depends on the purity of the `readCurrentTime` function passed to it. For example, if we call this method like this:

```
var result = GenerateXYZReport(customer, orders, () => DateTime.Now);
```

..then the function becomes impure because we passed a `readCurrentTime` function that reads the current system time via the impure `DateTime.Now` property getter.

However, we can also call it like this:

```
var result = GenerateXYZReport(customer, orders, () => new DateTime(2018, 6, 2));
```

Now, it is a pure function because the `readCurrentTime` function returns the same value every time.

If the method also communicates with some web services to obtain custom data based on orders, we can add another function parameter to the method like this:

```
public Report GenerateXYZReport(
  Customer customer,
  ImmutableArray<Order> orders,
  Func<DateTime> readCurrentTime,
  Func<Uri, SomeReportCustomData> obtainCustomDataFromWebService)
```

The `obtainCustomDataFromWebService` function would be used by the method to communicate with some web service to obtain some data.

Note that the logic to determine which web service to use is still part of the GenerateXYZReport method. Only the impure part is extracted as a function parameter.

Although the GenerateXYZReport method is now potentially-pure, we know that in production, we are going to pass impure functions for the `readCurrentTime` and `obtainCustomDataFromWebService` parameters.

Why is this useful then?

It is useful because now the method is honest about the impurities that it contains.

We can now configure the imaginary Visual Studio feature to treat potentially-pure functions, as pure. If a method declares all impurities as function parameters, Visual Studio would be able to tell us with certainty that the method does not do anything impure apart from the things it is declaring via the function parameters.

What about methods that read or write to global state? And methods that mutate their parameters?

- For a method that reads global state, we should prefer to make it take the value of the state as an input parameter (like how we added the `currentTime` parameter). If this is not possible, e.g. the method needs to read state at many different times, we can make such method take a `Func<State>` (like how we added the `readCurrentTime` parameter).

- For a method that updates global state, we should prefer to make it return the new state value using the return type of the method. If the method already has a return type, we can use a tuple to return both the original return value and the new state value. Again, if for some reason we need to update the state multiple times during the method execution, we can make the method take some `Action<State>` parameter that we can call to set the new state.

- For a method that mutates any of its parameters, we should make the parameter types immutable and instead return any new values via the return type of the method.

The CalculateDiscount method I displayed in the beginning of this article has the following signature:

```
public static Discount CalculateDiscount(Cart cart, Customer customer)
```

The method mutated the `Customer` parameter to increase the number of times the customer `Status` has been used. We can change the method signature like this to avoid mutation:

```
public static (Discount discount, int numberOfTimesStatusUsed)
CalculateDiscount(Cart cart, Customer customer)
```

Now, it is the responsibility of the caller to somehow store the new value of `numberOfTimesStatusUsed` to where it needs to be stored. But now the method is more honest. It tells us that one output of the method is the new number of times the status was used.

For more information about immutability, see the Designing Data Objects in C# and F# article.
All my suggestions for making impure methods honest end up making the methods pure or potentially-pure. When this happens, the method signature becomes honest, but it also adds more work for the caller.

> For example, the method that calls the GenerateXYZReport method needs to pass values for both the readCurrentTime and the obtainCustomDataFromWebService parameters. But in order for the caller method itself to be potentially-pure, it has to take these as parameters itself. This would mean that all the methods that end up calling GenerateXYZReport (directly or indirectly) would need to take such function parameters if they wanted to be honest.

This does not scale well because if we change a lower-level method to take a new function parameter, all the methods at a higher level that invoke it directly or indirectly would need to be updated to take such parameters themselves.

How can we compose honest functions in a way that scales? I will talk about this in an upcoming article.

In this article, I also talked about an imaginary Visual Studio feature that helps the reader know which methods are pure and which are not. I will expand on this in an upcoming article.

Conclusion:

This article discussed honest methods/functions.

An honest function makes it easier to understand what it does from its signature, without the need to read its implementation. We can make methods more honest by making them declare all possible outputs and all possible inputs, by using names that describe what they do, and by making methods pure or potentially-pure.

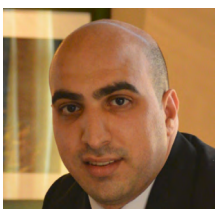
Potentially-pure methods are methods whose bodies are pure, but that invoke functions passed to them as parameters that could be pure or impure.

This tutorial ended on the note of explaining a problem related to making all impure methods in an application potentially-pure. We will see more about this problem and discuss a solution in an upcoming article.

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.*

Damir Arh

.NET Core Development

in Visual Studio Code

The official C# extension allowed developers to use Visual Studio Code from the beginning as a light-weight editor for .NET Core projects. With many other extensions created by members of the community, .NET Core development has become easier and enjoyable.

In this article, I will inspect some of these extensions and help you configure Visual Studio Code to make these extensions work together. If you're not familiar with Visual Studio Code, you can learn more about it from my previous article Visual Studio Condensed.

Manipulating Projects and Code Files

If you're used to Visual Studio, the Solution Explorer window will probably be the first thing you'll miss when you try opening your .NET Core solutions in **Visual Studio Code**.

The Solution Explorer in Visual Studio renders the projects in your solution as a tree view. This view contains the individual source code files. It also contains virtual nodes with additional important information about your projects, such as references to NuGet packages and other projects stored in the project file.

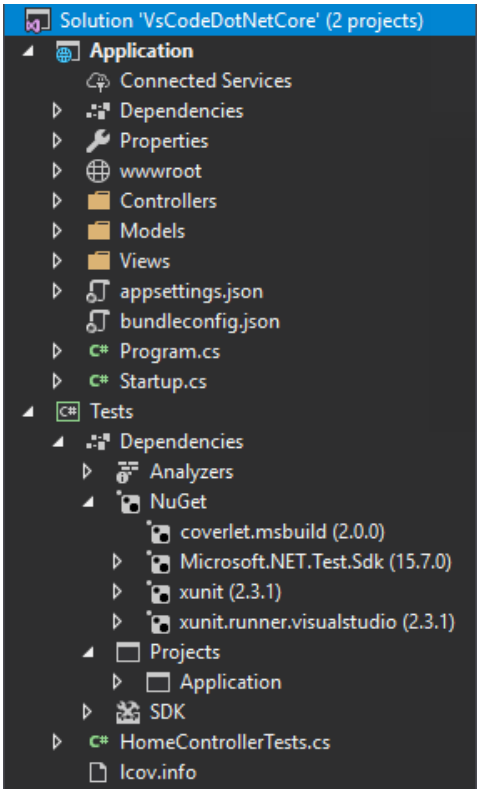


Figure 1: Solution Explorer in Visual Studio 2017

Visual Studio Code revolves around folders, not solution files. Its main navigation tool is the Explorer view which lists all the files and subfolders of the currently opened root folder. In a typical .NET Core solution all its projects are inside the same solution folder. This simple list of files does not provide the same level of information as the Solution Explorer does in Visual Studio.

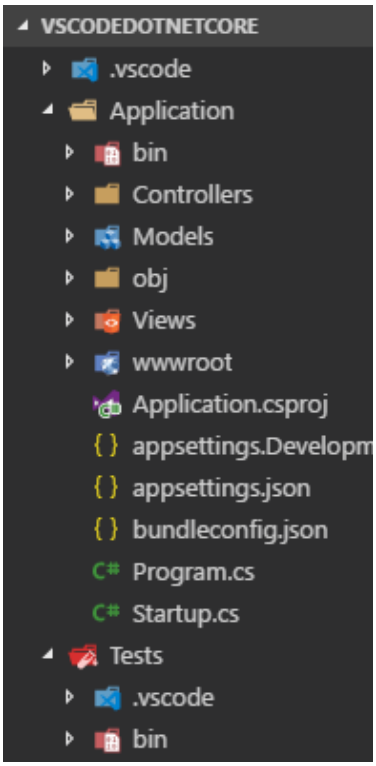


Figure 2: .NET Core solution folder in Visual Studio Code Explorer view

Fortunately, there's an extension available which can help with that.

If you install the vscode-solution-explorer extension in your copy of Visual Studio Code, it will add a Solution Explorer view to it.

It's not a fully featured equivalent to the Visual Studio Solution Explorer, but it does follow the same concept of reading the information from the solution and project files, and rendering it in a tree view with the solution and its projects as the main nodes.

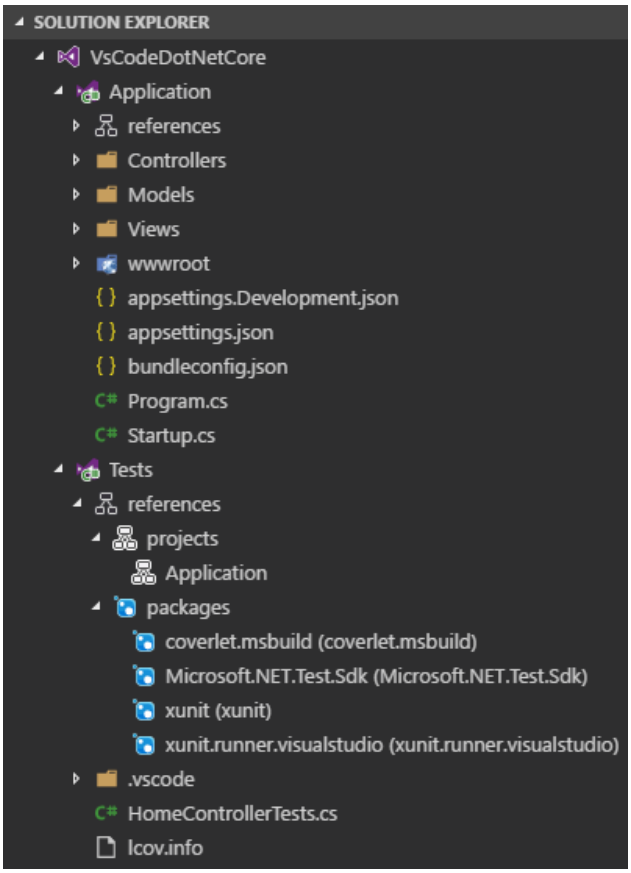


Figure 3: Solution Explorer view in Visual Studio Code

The extension not only displays the solution in a different manner but also adds context menu commands for creating solutions, projects and code files. You can use these commands instead of running `dotnet` commands directly from a terminal:

- To create a new solution when you have an empty folder open in Visual Studio Code, invoke the *Create new empty solution* command by clicking on the *No solution* found text in the Solution Explorer pane or by invoking it from the Command palette. After entering the name, the solution will be created by executing the following `dotnet` command:

  ```
  dotnet new sln -n VsCodeDotNetCore -o C:\Users\Damir\Temp\VsCodeDotNetCore
  ```

  Once the solution is created, the extension will encourage you to create a template folder.

  If you approve the request, it will create the `.vscode/solution-explorer` folder for you. Inside it, the extension will put a collection of special template files it is distributed with.

  When you'll create new code files through the extension (as explained later in the article), one of these templates will be used as the starting content for the new file. Along with the other files that Visual Studio Code itself puts in the `.vscode` folder (e.g. `launch.json` with launch configurations and `settings.json` with workspace settings), make sure to include the created template folder in the source control.

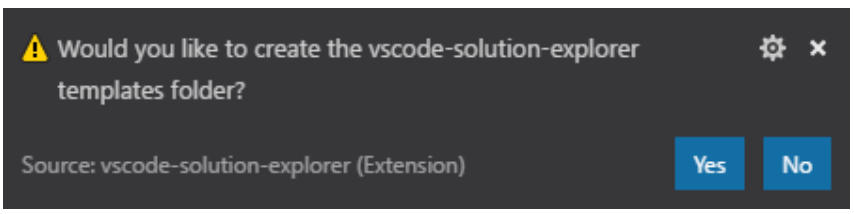


Figure 4: Notification for creating a templates folder

The same request will be displayed every time you open a folder with a solution file which does not yet have the `.vscode/solution-explorer` folder with template files created.

- You can now right-click on the solution node in the Solution Explorer and invoke the *Add new project* command. It will list the templates provided by the `dotnet` command.

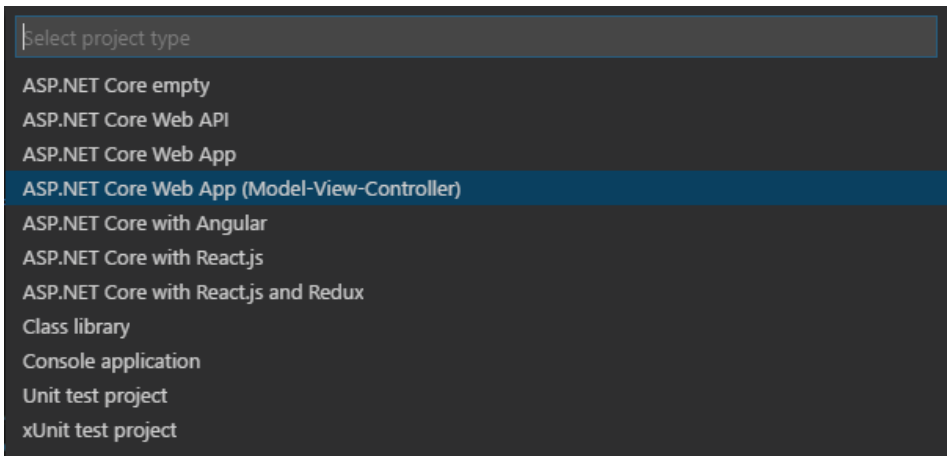


Figure 5: List of available project templates

After selecting the language (C#, F# or VB) and entering the name, it will again execute the `dotnet` command with parameters corresponding to your choices:

```
dotnet new mvc -lang C# -n Application -o Application
```

- New files can now be created by right-clicking the project or a folder inside it and invoking the *Create file* command. You will need to enter a filename (including its extension, e.g. `.cs`) and select one of the templates which were previously created in the `.vscode/solution-explorer` folder.

Figure 6: List of available file templates

As soon as you open the first C# code file in the editor, the C# extension will offer to generate build tasks and launch configurations for your project.

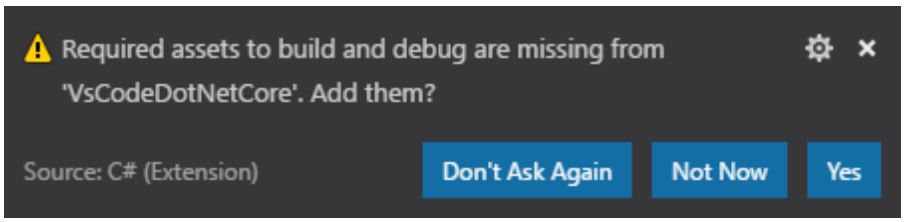


Figure 7: Notification for creating build and debug assets

These tasks will allow you to quickly build your project using the standard `Ctrl+Shift+B` keyboard shortcut, and more importantly to run and debug the project using the `F5` keyboard shortcut.

Similar to Visual Studio (VS), it will start the application. In case of a web application, it will also open the start page in your default browser. It will automatically attach the debugger to the .NET Core application

process. The execution will stop at breakpoints you place in the code, allowing you to look at the current program state.

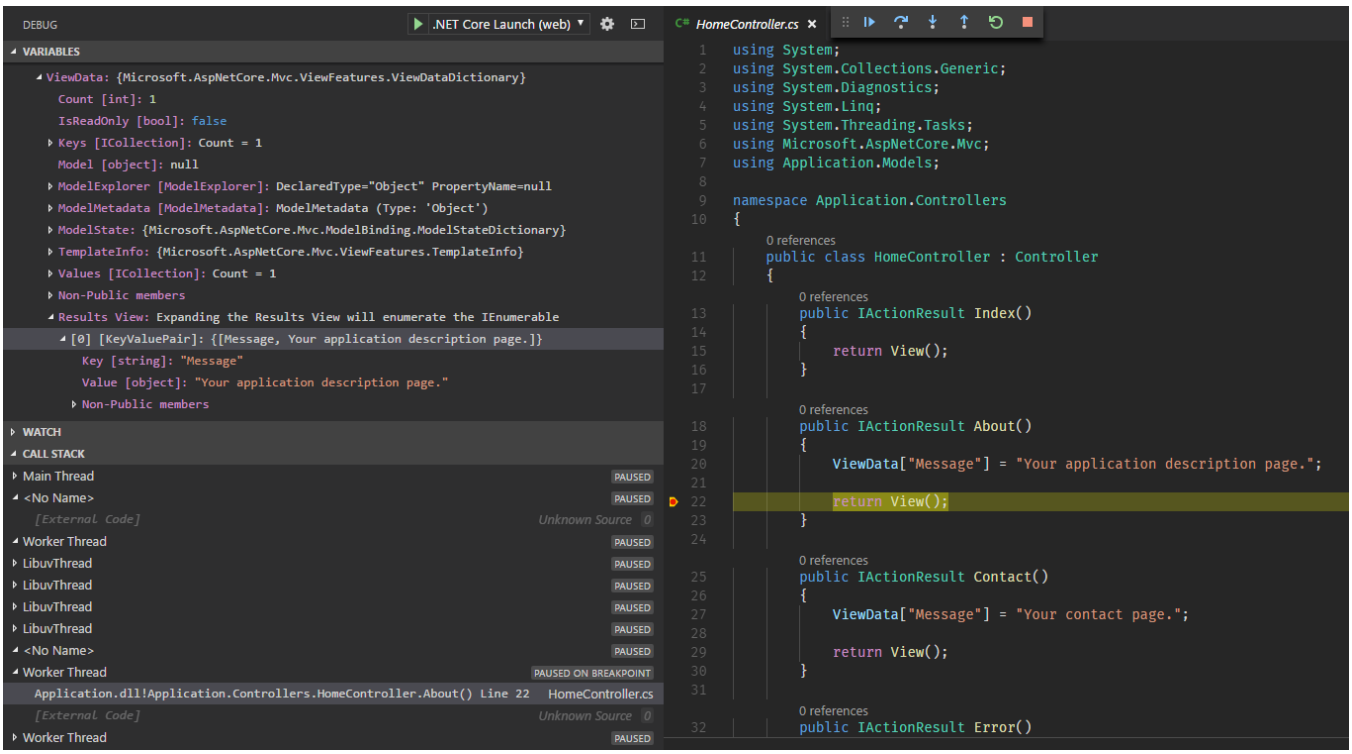


Figure 8: Debugging a .NET Core application

Working with Unit Tests

Unit tests are an important part of software development.

A test project can be created in the solution folder following the same steps as when creating the application project. You just need to select the correct project template (*Unit test project* or *xUnit test project*, depending on the test framework you want to use).

Once the test project is created, you can reference the application project from it using the Solution Explorer *Add reference* command from the project context menu. With only two projects in the solution, the application project will automatically be added as a reference. If there were more projects, you would have to select one from the list.

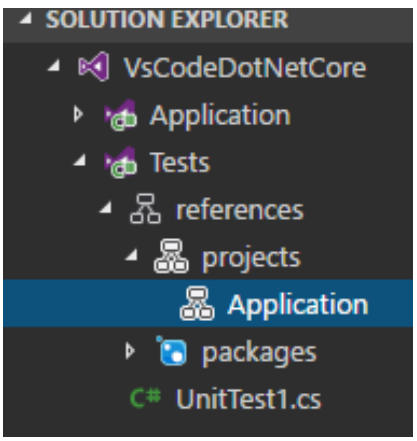


Figure 9: Project reference in the Solution Explorer view

To simplify running the unit tests and viewing the results, you should install another extension, .NET Core Test Explorer.

It provides several testing related functionalities:

- A new side bar view resembling the Test Explorer in Visual Studio (hence the name) and a corresponding icon in the activity bar (i.e. the vertical bar with icons for different side bar views) to access it.

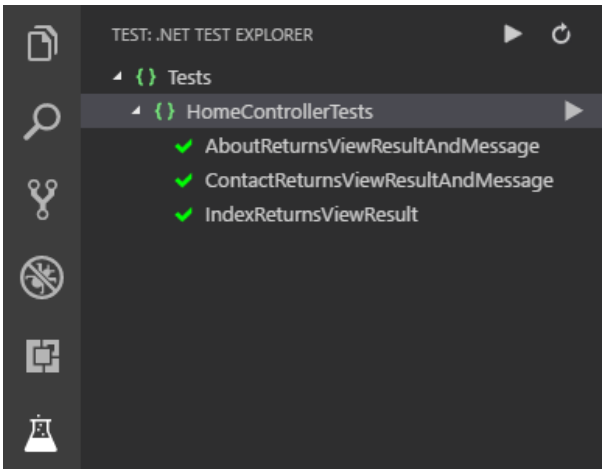


Figure 10: .NET Test Explorer side bar view

It lists all the tests in a tree view, grouped by namespace and class, along with the results of the latest test run. Each test can be navigated to using the *Go to Test* command in its context menu. Tests can be run from here at each hierarchy level: individually, by class, by namespace, or all of them.

- The latest test outcome for each test method in its Code Lens.

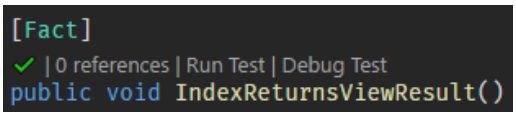


Figure 11: Test outcome in Code Lens

There's a command for running and debugging individual tests added to Code Lens as well. The latter is very useful for troubleshooting failing tests.

- Failed tests listed as problems for the files in which they are contained.

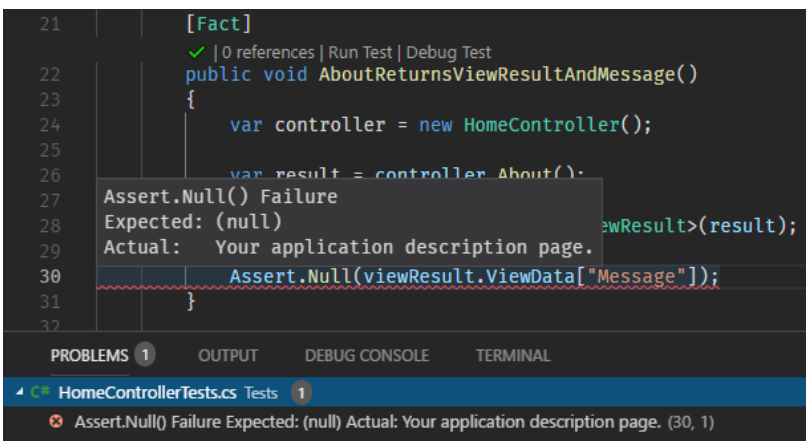


Figure 12: Failed test treated as a problem

The extension currently supports only a single test project inside the folder open in Visual Studio Code. If it finds more than one during auto-discovery, all the features will not work correctly.

There's an open issue in GitHub to add proper support for multiple test projects. Until it gets resolved, the problem can be partially worked around by specifying the path to the test project as a workspace setting. To do so, open the settings editor via the *File* > *Preferences* > *Settings* menu item, click the *Workspace settings* tab and add the following line:

```
"dotnet-test-explorer.testProjectPath": "./Tests"
```

This will not make the extension work with multiple test projects, but will restrict it to only process the test project inside the given folder. As long as there is only a single test project in that folder, the extension will function correctly with that test project and ignore any other test projects in the solution.

This way you'll be able to use it with one test project instead of not using it at all.

Continuous Testing

In my previous article Continuous Testing in .NET, I described how continuous testing for .NET Core can be configured using a command line tool. Although the .NET Core Test Explorer extension has some built-in support for continuous testing, i.e. auto running of tests whenever any source file changes, it still depends on the same command line tool.

Therefore, to get it working, you first need to add the tool reference to the test project. You can open the test project file from its context menu in the Solution Explorer view. The following line must be added inside the `ItemGroup` element which already contains other `PackageReference` and `DotNetCliToolReference` elements:

```
<DotNetCliToolReference Include="Microsoft.DotNet.Watcher.Tools" Version="2.0.2" />
```

This will make the `dotnet watch` command available to the extension. To enable the continuous testing auto watch mode in the extension, you also need to add the following entry to the workspace settings:

```
"dotnet-test-explorer.autoWatch": true
```

After changing the setting, you need to reinitialize the extension by clicking the *Refresh* button in the top right corner of its side bar view. Once it discovers the tests in the project, it will start the `dotnet watch` command line tool to watch for changes and automatically rerun the tests when a change is detected.

Code Coverage

Recently, a code coverage library for .NET Core was released, named Coverlet. By referencing it from the test project, the code coverage results can also be displayed inside Visual Studio Code.

First, you need to add the `coverlet.msbuild` NuGet package to the test project. You can use the *Add package* command from the Solution Explorer context menu.

To calculate the code coverage during the test run, you would need to add additional arguments to the command line which is used to run the tests.

Unfortunately, as of now you can't change the command which the Test Explorer extension is using to run

the tests. However, the extension will still pick up and show the test results if they are output to the same file that the extension itself is using.

This allows us to use a Visual Studio Code task to run the tests with the required command line arguments and have the Test Explorer still pick up and show the results.

Let's configure it.

If you look at the Test Explorer output in the Visual Studio Code Output window after the extension starts watching for the changes, you can find the command which was executed, e.g.:

```
d:\Users\Damir\Temp\VsCodeDotNetCore> d:\Users\Damir\Temp\VsCodeDotNetCore\Tests
d:\Users\Damir\Temp\VsCodeDotNetCore\Tests> dotnet watch test --logger
"trx;LogFileName=C:\Users\Damir\AppData\Local\Temp\test-explorer-3rLrRb\Results.
trx"
```

As you can see, the results file is placed in a temporary folder. You can change that by configuring a workspace setting:

```
"dotnet-test-explorer.pathForResultFile": "./.vscode/test-results"
```

The folder must exist prior to running the command, otherwise it will fail. Although the extension will take the setting into account, it will unfortunately still create a temporary folder inside the folder specified which will change every time Visual Studio Code is restarted:

```
d:\Users\Damir\Temp\VsCodeDotNetCore> d:\Users\Damir\Temp\VsCodeDotNetCore\Tests
d:\Users\Damir\Temp\VsCodeDotNetCore\Tests> dotnet test --logger
"trx;LogFileName=d:\Users\Damir\Temp\VsCodeDotNetCore\.vscode\test-results\test-
explorer-WkEF31\Results.trx"
```

You can still configure a task for continuously running the tests in .vscode/tasks.json which will run the above command, but with the additional arguments needed for Coverlet:

```
{
  "label": "test",
  "command": "dotnet",
  "type": "process",
  "options": {
    "cwd": "${workspaceFolder}/Tests"
  },
  "args": [
    "watch",
    "test",
    "--logger",
    "\"trx;LogFileName=../.vscode/test-results/test-explorer-WkEF31/Results.trx\"",
    // Coverlet arguments
    "/p:CollectCoverage=true",
        "/p:CoverletOutputFormat=lcov",
        "/p:CoverletOutput=./lcov"
    ],
    "problemMatcher": "$msCompile",
    "group": {  // set the task as the default test task
        "kind": "test",
        "isDefault": true
    }
}
```

Apart from enabling Coverlet, the newly added arguments also specify its output format and result file so that a Visual Studio Code extension will be able to read it. The `group` property is used to make this task the default task in Visual Studio Code so that it can be run by the *Run Test Task* command.

To display the code coverage information in Visual Studio Code, install the Coverage Gutters extension. It works with files in any language if the coverage information is available in one of its supported formats.

With the latest change in the task configuration, a `lcov.info` file will be generated in the test project folder on every test run. You can enable the Coverage Gutters extension to watch for that file by clicking on the *Watch* button in the left part of the status bar.

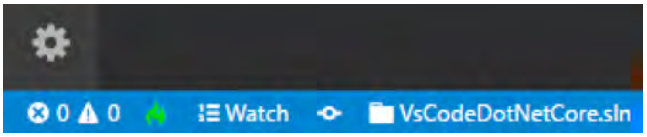


Figure 13: Coverage Gutters Watch button in status bar

This will cause the coverage information to automatically refresh in the code editor windows as the tests are continuously running in the background.

HomeController.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Application.Models;

namespace Application.Controllers
{
    3 references
    public class HomeController : Controller
    {
        1 reference
        public IActionResult Index()
        {
            return View();
        }

        1 reference
        public IActionResult About()
        {
            ViewData["Message"] = "Your application description page.";

            return View();
        }

        1 reference
        public IActionResult Contact()
        {
            ViewData["Message"] = "Your contact page.";

            return View();
        }

        0 references
        public IActionResult Error()
        {
            return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
        }
    }
}

```

Figure 14: Code coverage information in the code editor window

You will still need to run the task in `tasks.json` once to start the watcher. Since this task is configured as the default test task, you can invoke it with the special *Tasks: Run Test Task* command in Visual Studio Code. Since there's no shortcut configured for this command out-of-the-box and it's not included in any menu, you can only invoke it from the Command palette.

To make it more accessible, you can give it the same shortcut as in Visual Studio by adding the following to your keyboard shortcuts file `keybindings.json`:

```
{
  "key": "ctrl+r t",
  "command": "workbench.action.tasks.test"
}
```

Whenever you save one of the files from now on, the tests will automatically re-run. To stop the watcher, you will need to invoke the `Tasks: Terminate Task` command. It also has no shortcut configured by default.

It's also a good idea to disable the Test Explorer auto watch mode when using your own task to avoid running the tests twice and prevent any conflicts that could occur because of that.

The path for the results file in the build task will need to be modified on every restart of Visual Studio Code, which makes this solution less than ideal, but it's still the best I could come up with using the tools that are currently available.

Deployment to Azure

Microsoft Azure is Microsoft's cloud computing platform. One of the many services it provides is Azure App Service which can be used for hosting web applications.

Web application deployment will typically take place on the build server as a part of the continuous deployment configuration. However, for testing purposes, you sometimes might want to deploy your application manually. If your deployment target is an Azure App Service, the Azure App Service extension for Visual Studio Code could prove useful.

Once installed, it adds its own side bar view and a corresponding icon in the activity bar.
Before you'll be able to use it, you'll need to sign into your Azure account using the standard interface provided by the Azure Account extension which gets automatically installed as an extension dependency. Clicking the *Sign in to Azure...* button in the Azure App Service side bar view will open a notification with short instructions.

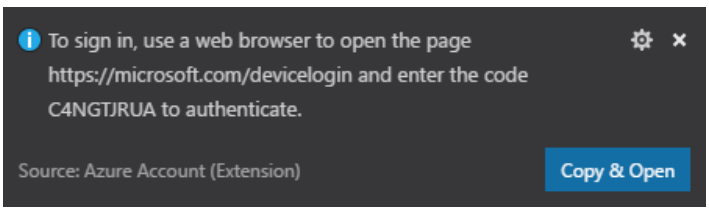


Figure 15: Azure Account extension sign-in notification

The button will open a web page where you will paste the code provided and sign into your selected Azure account. After you do that, you will be able to explore the Azure app services you have access to, grouped by subscription.

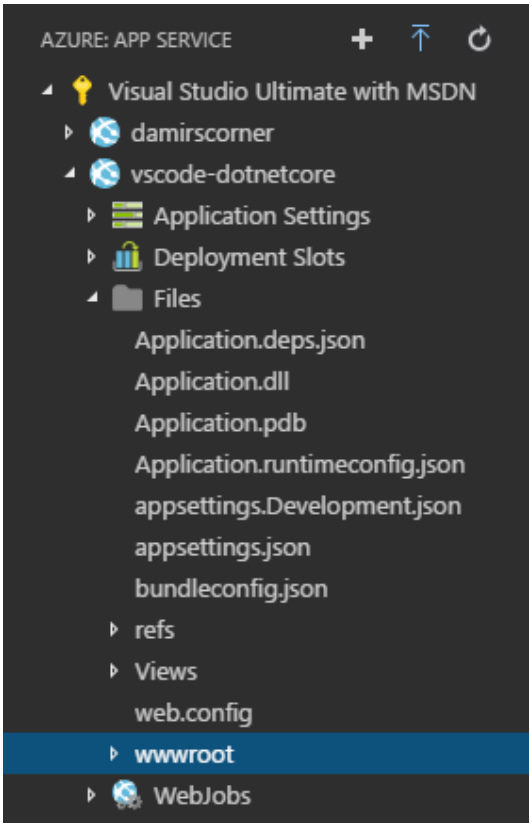


Figure 16: Azure App Service tree view in Visual Studio Code

The Azure App Service extension features are mostly focused on deployment of Node.js applications, but when used correctly, it can also be used for deploying ASP.NET Core web applications.

Since the extension doesn't have any awareness of the solution and project structure, you'll first need to create a publish folder for the project you want to deploy. You can use the *Publish* command in the Solution Explorer context menu for that.

The path to the folder will typically be the `bin/Debug/netcoreapp2.0/publish` subfolder inside the project folder, but it will also be listed at the end of the Solution Explorer output in the Visual Studio Code Output window after the command completes.

You're now ready to start the extension's deployment wizard by clicking the *Deploy to Web App* button in the top right corner of the side bar view. Following its steps, you will enter all the required information:

- path to the publish folder (click the *Browse* option in the dropdown list to select the folder),
- Azure subscription,
- web app (you can select an existing one or create a new one),
- web app name (only when creating a new web app),
- resource group (only when creating a new web app, you can select an existing one or create a new one),
- resource group name (only when creating a new resource group),
- operating system (only when creating a new web app),
- app service plan (only when creating a new web app, you can select an existing one or create a new one),
- app service plan name (only when creating a new plan),
- pricing tier (only when creating a new plan, only a subset of available tiers will be listed, you will need to create the app service plan in the portal to have all the tiers available),
- location (only when creating a new plan).

Upon the completion of the wizard, any newly created Azure resources will be created first. Then the contents of the selected folder will be deployed to the target Azure app service.

When the process is complete, the URL will be listed at the end of the Azure App Service output in the Visual Studio Code Output window. If you Ctrl-click it, a working web app should open in your default browser.

If it doesn't, you can add a *Files* node to the Azure app service tree view for basic troubleshooting by adding the following setting to the User settings tab of the settings editor:

```
"appService.showRemoteFiles": true
```

By expanding the node for an app service, you will now see the files which were uploaded to Azure and be

able to even explore their contents in the editor if you select them in the tree view.

There are other commands available in the tree view context menu, e.g. to restart or stop the service or even to delete it. Be very careful when using them if you have production services running in your Azure account.

Conclusion:

By following the instructions in this article, you can configure your free copy of Visual Studio Code to be a convenient tool for .NET Core development.

Of course, it's not as feature rich as Visual Studio 2017, but it's more light-weight, not limited to Windows and even cheaper. Also, judging by the improvements in the extensions and the enhancements to Visual Studio Code itself in the last two years, we can expect the .NET Core development experience to continue improving in the future.

Microsoft® MVP Most Valuable Professional

Damir Arh

Author

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

Thanks to Yacoub Massad *for reviewing this article.*

Ravi Kiran

THIS ARTICLE EXPLAINS HOW ANGULAR USES AHEAD OF TIME COMPILATION ON THE BROWSER AND THEN SHOWS HOW IT IMPROVES THE RUNTIME PERFORMANCE OF THE PAGE.

Ahead of Time Compilation (AoT):

How it Improves an Angular Application

Angular was created to build rich client applications that run faster everywhere. The Angular team keeps finding ways to make the applications written using the framework, perform better. There are several aspects that contribute to the performance of an Angular application. It includes:

- size of the bundled JavaScript files served to the browser
- the time Angular needs to render components on the page
- how the portions of the page are re-rendered when there is any change in the values

Angular CLI (bit.ly/dnc-ang-cli) takes care of the bundle size and eliminating unused code from the bundles, so we need not worry about the way it creates bundles. To optimize the runtime behavior and the way the components are rendered, we need to know how Angular handles the components on the browser and how Angular handles the changes happening on the objects while re-rendering the views.

Different Levels of Compilation in an Angular Application

Most of the Angular applications are written in TypeScript. The TypeScript code has to be compiled to JavaScript before loading on the browser.

If you are using Angular CLI, the built-in webpack process takes care of it. If you have your own setup for your application, you need to use a bundler with the TypeScript transpiler to compile the code and generate the bundles for deployment. Then the bundled code is loaded on the browser and Angular takes care of rendering the components starting from the root component of the application.

When Angular renders the components on the page, it converts the code into JavaScript VM-friendly code. Angular's compiler, installed using the npm package `@angular/compiler` does this in the browser before the content is shown on the page. This process is called Just in Time (JiT) compilation, as the compilation happens just before the page gets rendered. The generated code makes it easier for the JavaScript VM to work with the objects in the application thereby making change detection more efficient.

Result of the JiT compilation happening on the browser is not hidden, it is visible in the sources tab of the developer tools. The resultant code of the JiT compilation can be seen under the `ng://` folder in the sources tab. The following figure shows the contents in the `ng://` folder for the default application generated using Angular CLI:

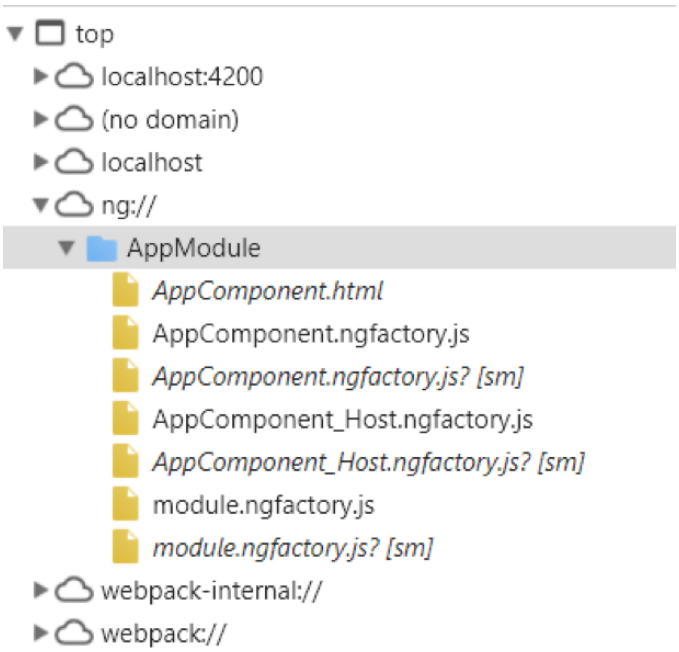


Figure 1 – Contents of ng folder

As you can see, the JiT compiler produces a **.ngfactory.js* file for every component and module. If you check the contents of any of these files, you will find that this code is highly annotated and adds a number of calls to Angular's core API.

We will take a closer look at the generated code in the next section. For now, just remember that Angular runs the code inside the **.factory.js* files to render the views.

A Closer Look at the Code Generated by Angular in the Browser

Let's see how a component printing a simple Hello World message gets compiled in the browser. Say we have the following component:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
```

```
  template: `<span>{{title}}</span>
  <br />
  <button (click)=”changeTitle()”>Click here!</button>`,
  styles: [`
  span {
    font-family: cursive
  }`]
})
export class AppComponent {
  title = ‘Hello world!’;
  private count = 0;

  changeTitle(){
    this.count++;
    this.title = `Changed! ${this.count}`;
  }
}
```

The JiT compiled version of this component looks like the following:

```
) {
    var styles_AppComponent = ['span[_ngcontent-%COMP%] {\n    font-family: cursive\n  }']; // 1
    var RenderType_AppComponent = jit_createRendererType2_0({
        encapsulation: 0,
        styles: styles_AppComponent,
        data: {}
    });
    function View_AppComponent_0(_l) {
        return jit_viewDef_1(0, [(_l()(),
        jit_elementDef_2(0, 0, null, null, 1, 'span', [], null, null, null, null, null)), (_l()(), // 2
        jit_textDef_3(1, null, ['', ''])), (_l()(),
        jit_textDef_3(-1, null, ['\n'])), (_l()(),
        jit_elementDef_2(3, 0, null, null, 0, 'br', [], null, null, null, null, null)), (_l()(),
        jit_textDef_3(-1, null, ['\n'])), (_l()(),
        jit_elementDef_2(5, 0, null, null, 1, 'button', [], null, [[null, 'click']], function(_v, en, $event) { // 3
            var ad = true;
            var _co = _v.component;
            if (('click' === en)) {
                var pd_0 = (_co.changeTitle() !== false);
                ad = (pd_0 && ad);
            }
            return ad;
        }, null, null)), (_l()(),
        jit_textDef_3(-1, null, ['Click here!']))], null, function(_ck, _v) {     // 4
            var _co = _v.component;
            var currVal_0 = _co.title;
            _ck(_v, 1, 0, currVal_0);    // 5
        });
    }
    return {
        RenderType_AppComponent: RenderType_AppComponent,
        View_AppComponent_0: View_AppComponent_0
    };
    //# sourceMappingURL=data:application/json;base64,eyJmaWxlIjoibmc6Ly8vQXBwTW9kdWxlL0FwcENvbXBvbmVudC5uZ2ZhY:
}
```

Figure 2 – Contents of AppComponent.ngfactory.js

This file is created using the metadata added to the component class. It generated two objects `RenderType_AppComponent and View_AppComponent_0`. The module is exporting these objects, Angular will use these objects to create the component. Going into details of every single minute thing in Figure 2 is beyond the scope of this article. The vital statements of the snippets are marked with numbered comments (`// 1`, `// 2` and so on) and they are described in the following listing:

Styles of the component are added to the array `styles_AppComponent`. As the component uses the default emulated encapsulation for the styles, selector of the span element’s style is appended with the unique attribute created for the component. This style array is used to construct the `RenderType_`

`AppComponent` object.

The function `View_AppComponent_0` converts the HTML template of the component to JavaScript. Comment 2 in Figure 2 shows how the span element is converted to JavaScript.

1. Creates the button element using JavaScript and wires up the event listeners. As the button has a click event handler attached to it in the template, we see the argument `[[null, 'click']]` and the callback function in this statement calls the functionality assigned to the (`click`) event in the template.

2. The callback function passed here is called whenever the component's state is updated

This statement calls a method in the @angular/core library to update the span element with the new value of `title`

The generated code has information about all the bindings and the events added to the component's template. As the whole HTML in the template is converted to JavaScript, and the factory provides information about every node in the template, it becomes easier to target a node and apply the update on it. This makes re-rendering of the bindings faster.

In addition to this, the AoT compiled code makes the application less prone to script injections. This is because the HTML templates are converted to JavaScript and hence the chances for someone to manipulate the templates to be rendered through scripts becomes difficult.

Phases in AoT

The AoT compilation process goes through the following phases where it analyzes the decorators, generates code based on the decorators and then validates the binding expressions.

Analysis

The analysis phase analyzes the decorators and checks if they obey rules of the compiler. If the decorators fail the rules, the compiler reports it and stops compilation.

- **Expression Syntax:** The metadata annotations don't support every JavaScript expression while assigning values to the metadata properties. The set of supported expressions are:

o literal objects, arrays, function calls, creation of objects using `new` keyword, values assigned from properties of objects or members of arrays

  o simple values like strings, numbers, Boolean

  o Values derived from using prefix operators (like `!isValid`), conditional operators, binary operators

Any other operators like increment/decrement operators would not work.

- **No Arrow Functions:** Arrow functions can't be used to assign values to the decorator properties. For example, the following snippet is invalid:

```
@Component({
  providers: [{
```

```
    provide: MyService, useFactory: () => getService()
  }]
})
```

To fix this, it has to be changed as following:

```
function getService(){
  return new MyService();
}
@Component({
  providers: [{
    provide: MyService, useFactory: getService
  }]
})
```

- **Limited function calls:** The compiler supports only a set of Angular decorators for code generation. Custom decorators won't be used while generating code. Also, the functions used in the metadata have to be very simple and contain only a single return statement. The built-in metadata functions used in configuring routes like *forRoot* are defined in that way. You can check Angular's documentation site for more details on the supported functions.

- **Folding:** Only exported members from a module can be used in the metadata properties. Non-exported members are folded while generating the code. For example,

```
let selector = 'app-root';
@Component({
  selector: selector
})

Gets folded into:
@Component({
  selector: 'app-root'
})
```

But not everything is foldable. The compiler can't fold spread operator on arrays, objects created using new keywords and function calls. You can check Angular's documentation for the set of supported foldable expressions.

Code Generation

The compiler uses the information saved in the *.metadata.json* file that comes as the result of the analysis phase to generate code. While the compiler uses everything that passes analysis, it may throw errors if it finds any other semantic violations.

For example, non-exported members can't be referenced and private class fields can't be used in the templates. The following snippet generates an error during compilation phase:

```
@Component({
  selector: 'app-root',
  template: `<span>{{title}}</span>`
})
export class AppComponent {
  private title = 'Hello world!';
}
```

Binding Expression Validation

In this phase, the Angular compiler checks validity of the data binding expressions in the HTML templates. The checks performed by the compiler during this phase are similar to the checks performed by TypeScript compiler on the *.ts* files.

The compiler checks for any typos in the template expressions and reports them in the errors. For example, consider the following snippet:

```
@Component({
  selector: 'app-root',
  template: `<span>{{title.toUpperCas()}}</span>`
})
export class AppComponent {
  title = 'Hello world!';
}
```

Here, template of the `AppComponent` in the above snippet has a typo. It reports the following error:

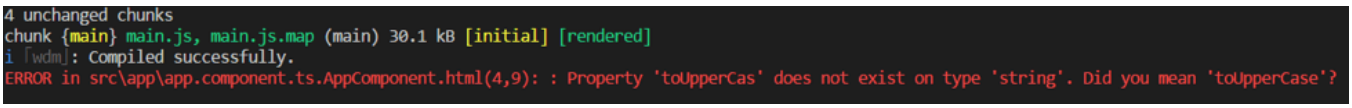


Figure 3 – Typo error in the template

Nullable objects in the component class have to be checked before they are used in the binding expressions. The following snippet may result in an error when it is loaded on the browser:

```
@Component({
  selector: 'app-root',
  template: `<span>{{name.toUpperCase()}}</span>`
})
export class AppComponent {
  name?: string;
}
```

Adding an *ngIf* to the template will prevent the error.

```
template: `<span *ngIf="name">{{name.toUpperCase()}}</span>`
```

To use an undeclared property in an object in a TypeScript file, we cast the object to `any` and use it. Similarly, in the template it can be type casted to *any* using the `$any` operator as shown below:

```
@Component({
  selector: 'app-root',
  template: `<span>{{$any(person).occupation}}</span>`
})
export class AppComponent2 {
  person: {name: string, age: number};
  constructor() {
    this.person = {
      name: 'Alex',
      age: 10
    };
    this.person["occupation"] = 'Student';
  }
}
```

The template results in an error without the `$any` operator.

Angular Compiler Options

The Angular compiler can be configured in the *tsconfig.json* file. The configuration options can be assigned using the `angularCompilerOptions` section. Angular provides a number of configuration options and going deep into them could be a topic for a separate article. The following listing describes some of the important options:

- skipTemplateCodegen: This option has to be used to suppress emitting `ngfactory` and `ngstyle` files. This option can be used to get the .metadata.json file generated and distribute it through npm package. This option is `false` by default.

strictInjectionParameters: When set to `true`, it generates an error if type of the arguments passed into classed marked with `@Injectable` can't be determined. This option is `false` by default.

- fullTemplateTypeCheck: This option tells the Angular compiler to enable the Binding Expression Validation phase. It is recommended to set this to true to avoid any undesired behavior of the application

- annotationsAs: This option enables advanced tree shaking of the Angular code. The possible values of this option are:

o decorators: It generates calls to `__decorate` helper classes and leaves the decorators in place. The code produced can't be applied with advanced tree-shaking

o static fields: It replaces decorators with static fields in the class and allows advanced tree-shaking

- preserveWhitespaces: This option can be used to preserve white spaces in the HTML templates. It is set to false by default

- enableIvy: This option enables the Ivy rendered. It is the new rendered in Angular that optimizes the Angular code even further. It is still experimental, so this option is false by default

Using AoT to Build Angular Code

It is possible to generate the code that Angular generates on the browser beforehand. This process is called **Ahead-of-Time (AoT) compilation**. For this, the Angular CLI has all the required setup.

An application can be executed with AoT during development using the following command:

```
> ng serve --aot
```

The `aot` flag supplied to the `ng serve` command tells the Angular compiler to generate the compiled code before serving the pages to the browser. If you see the code in the file `main.bundle.js` served to the browser, it contains the compiled code. So, you won't see the `ng://` folder anymore, as it is not generated in the browser.

The AoT compilation also notifies of any TypeScript errors in the template. For example, if you modify the

button element in the component as following,

```
<button (click)=”changeTitle(1)”>Click here!</button>`,
```

The AoT compiler will show a TypeScript error about the invocation of the `changeTitle` method. The following screenshot shows this error:

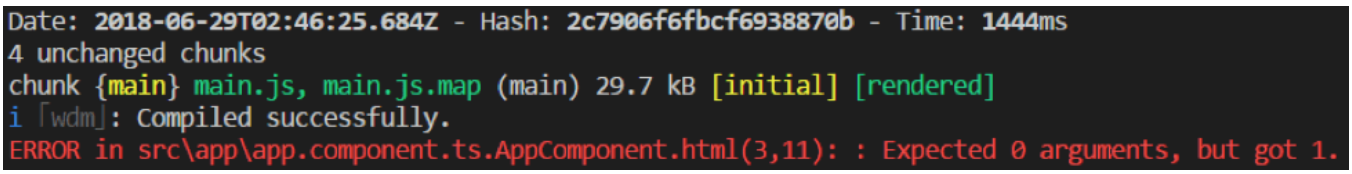


Figure 4 – TypeScript error by AoT compiler

This is quite helpful, as we get to know any potential errors during compile time itself. If an error of this type results in any odd behavior at the runtime, it becomes very hard to find the cause and fix it.

Angular CLI's command to generate application bundle, `ng build` has the `aot` option enabled by default and the Angular CLI team hasn't given an option to skip this option. So, every production bundle is AoT compiled code, which makes it effective on the browser with lesser number of runtime issues.

Comparing Bundle Size and Rendering time with and without AoT

Now that we saw how Angular compiles the code on the browser and how to do it before hand, let's see how the application performs in both cases. As first thing, let's compare size of the bundles created in both the approaches.

The *ng serve* command shows the list of bundled files to be served to the browser on the console. Figure 5 shows result of the `ng serve` command on a default Angular CLI generated application with the `AppComponent` we saw earlier in this article:

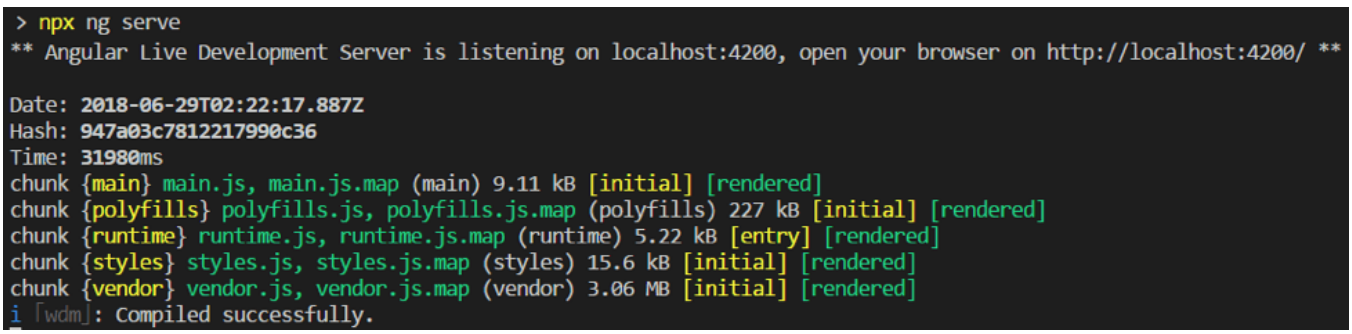


Figure 5 – Size of bundles generated in ng serve command

Figure 6 shows the result when the command is supplied with the *aot* flag:

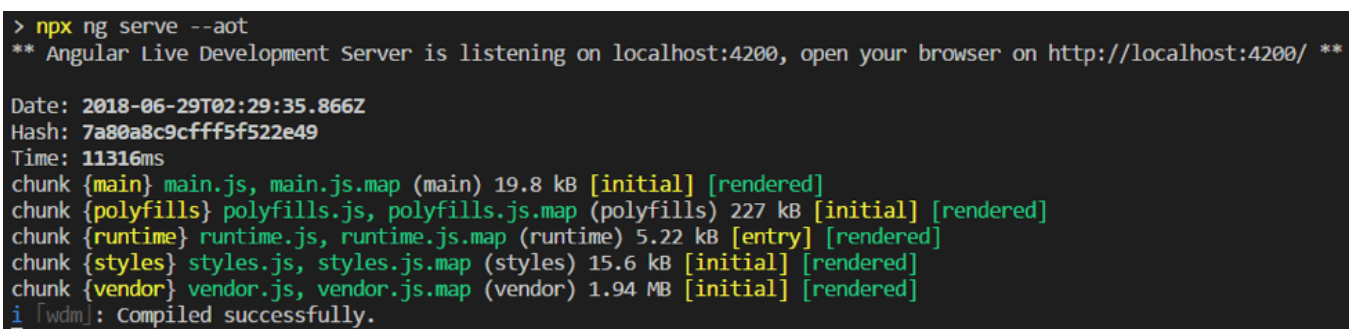


Figure 6 – Size of bundles generated in ng serve --aot command

On comparing size of the bundles created, you will see that *main.bundle.js* file is bigger when generated with the *aot* option. The *vendor.bundle.js* file is substantially smaller when generated with the `aot` flag.

This is because, the *vendor.bundle.js* file doesn't include the Angular compiler anymore. The Angular compiler runs through the JavaScript code generated after TypeScript compilation and performs JiT compilation on the browser. This process is not required when the code is *aot* compiled and hence the entire Angular Compiler gets excluded from *vendor.bundle.js*.

This reduction in the bundle size will make the scripts download faster on the page.

The reduction in the time taken to get the page ready can be checked in the Network tab of the browsers Chrome and Opera. Figure 7 & 8 show the rendered pages and the total load time when the page is served without and with the `aot` option:

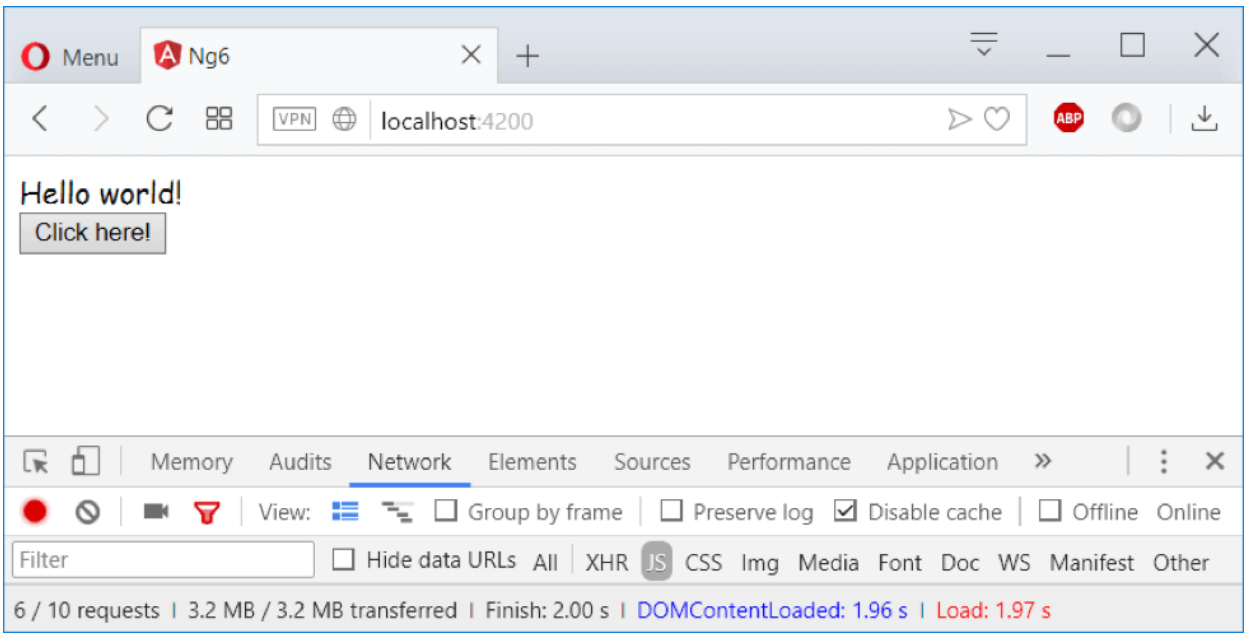


Figure 7 – Page load time without AoT

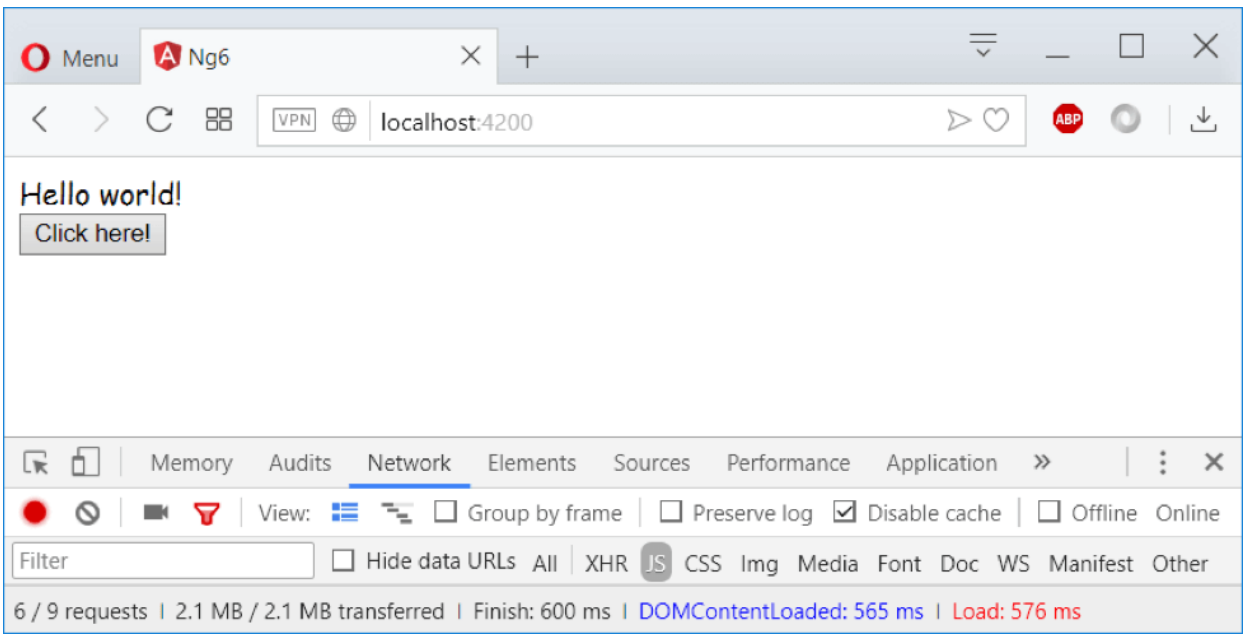


Figure 8 – Page load time with AoT

These figures clearly show the improvement in loading time and the size of content loaded. The page

loaded files of lesser size and finished rendering the page in 1.4 seconds less than the time it took before.

Using AoT with Webpack

Though Angular CLI provides all the required setup for development of a matured Angular application, you might want to use your own custom webpack based setup.

Setting up environment for Angular development with webpack is beyond the scope of this article, we will cover the packages required for the setup and configuration for AoT here. It is possible to enable AoT on the custom webpack setup using the `@ngtools/webpack` package and the `@angular/compiler-cli` package. The following command would help with that:

```
> npm install -D @angular/compiler-cli @ngtools/webpack
```

Import this file in your webpack configuration file as shown below:

```
const {AngularCompilerPlugin} = require('@ngtools/webpack');
```

This package has a loader and a plugin that have to be used. The loader takes the TypeScript, CSS and the ngfactory files and packages them. The plugin takes a bunch of options like the path to tsconfig file, path to the application module, compiler options for AoT and others. This package has to be configured in the webpack configuration.

The following snippet configures the loader:

```
module: {
  rules: [
    {
      test: /(?:\.ngfactory\.js|\.ngstyle\.js|\.ts)$/,
      loader: '@ngtools/webpack'
    }
  ]
}
```

..and in the plugins section, the `AngularCompilerPlugin` has to be configured like it is shown here:

```
plugins: [
  new AngularCompilerPlugin({
    tsConfigPath: 'path/to/tsconfig.json',
    entryModule: 'path/to/app.module#AppModule',
    sourceMap: true
  })
]
```

When to Use AoT?

AoT can be used to render the application in both development and production modes.

It is not a good option to run the application with the `aot` option during development, as the bundling process will need more time to generate the files to be served. So the amount of time one has to wait to see the changes on the browser would increase and it will make the developers less efficient.

Once most of the development is complete and any AoT validation issues have been fixed, then the AoT option can be used for development.

For production builds, it is always recommended to use AoT. Angular CLI imposes this and it doesn't provide an option to turn off AoT builds for production. If you happen to create custom setup for your project, make sure to configure AoT and use it for the production builds at least to get all the advantages.

Conclusion

Angular's compiler makes the application run better on the browser by generating JavaScript VM specific code. To make the applications run even faster, the Angular team has provided us with the option to compile the code before it loads on the browser. As we saw, the application would certainly work better on the browser with AoT enabled.

Let's make use of this feature to make our bundles smaller and the applications faster!

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.