Issue 29 | Mar-Apr 2017

DNCMagazine

www.dotnetcurry.com

Reduce Bad Code
with Code Reviews

Getting to know the Redux Pattern

Using
ElasticSearch, Kibana, .NET Core and Docker
to discover and visualise data

5 star Xamarin Apps

Singleton
Pattern or Antipattern

Concurrent Programming
in .NET Core

Choosing a JavaScript Framework

Getting started with React JS

The ascendance of JavaScript in the upcoming wave of Microsoft products and services, is huge. As a .NET developer, you just can't ignore the capabilities of JavaScript. This edition of the DNC Magazine helps you to get up and running with JavaScript libraries like React and Redux. We also have a shopping guide to help you choose a JS framework for your next ASP.NET MVC app.

Microsoft has empowered its developers with cross-platform development frameworks such as Xamarin. In this edition, we demonstrate how to create 5-star apps using Xamarin Test Cloud. In an end-to-end app, we also showcase how to integrate .NET Core with open source tools like Elastic Search and Kibana. For our C# developers, we talk about multithreaded concurrent programming in .NET Core.

This edition rounds off with a discussion on why Code Reviews are an important part of the development process. In a separate article, we also discuss why Singleton is an antipattern.

So how was this edition? E-mail me at suprotimagarwal@dotnetcurry.com.

Suprotim Agarwal
Editor in Chief

Editor In Chief
Suprotim Agarwal
suprotimagarwal@dotnetcurry.com

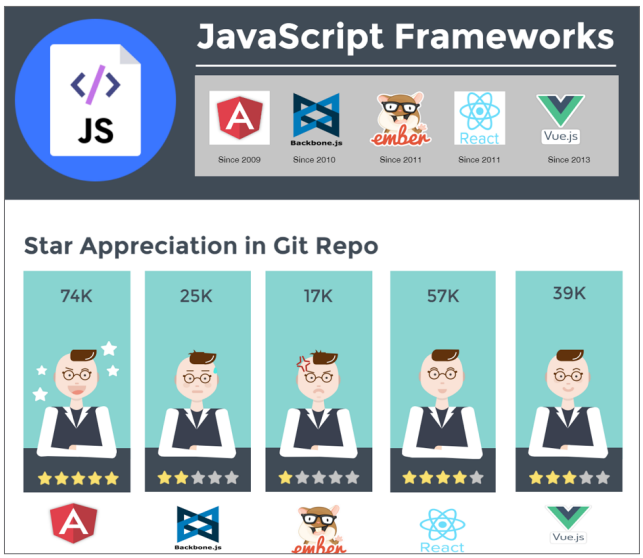
Art Director
Minal Agarwal

Contributing Authors
Craig Berntson
Damir Arh
Daniel Jimenez Garcia
Gerald Versluis
Gil Fink
Rahul Sahasrabuddhe
Ravi Kiran

Technical Reviewers
Benjamin Jakobus
Damir Arh
Francesco Bianchi
Gil Fink
Keerti Kotaru
Suprotim Agarwal
Yacoub Massad

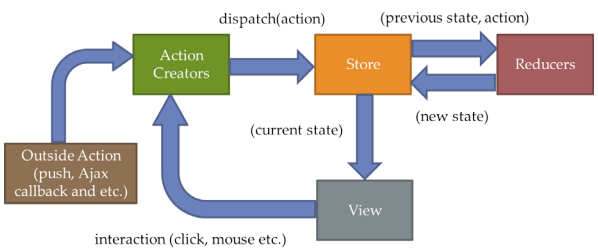
Next Edition
May 2017

Contents



Javascript Frameworks
A Shopper's Guide for the Modern ASP.NET MVC Developer

66



THANK YOU

FOR THE 29th EDITION


@damirarh


@gilfink


@ravikiran


@dani_djg


@yacoubmassad

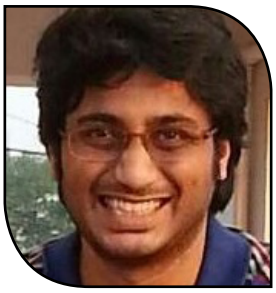

@craigber


@suprotimagarwal


@jfversluis


@rahul1000buddhe

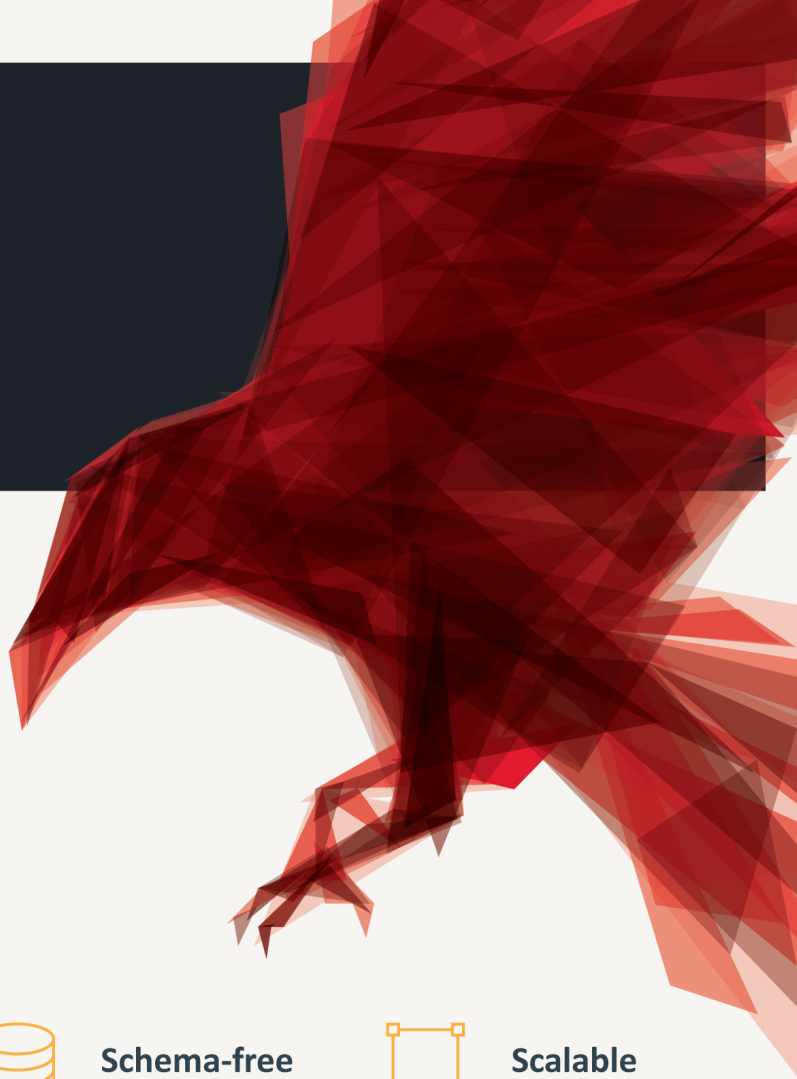

@benjaminjakobus


@keertikotaru


@ MyTwo_0Cents

WRITE FOR US

Why **Fortune 500** companies choose **RavenDB**?



In a world where data is one of the most important assets of any business the database technology should not only be protecting its data but also enhancing its business.

To address both of those needs, Hibernating Rhinos has introduced its NoSQL database called RavenDB and for the past few years, due to enhanced capabilities, it has become the choice of Fortune 500 companies.

The protection of data comes with meeting all the ACID parameters, being fully transactional and having extended failover support to guarantee you that the data will be safe and sound even when node failure happens. Moreover, the extended replication features allow businesses to setup complex failover clusters to move their protection to the next level and ensure availability or enhance their work by enabling sophisticated sharding and load balancing capabilities.

The out of the box querying features, high-performance and self-optimization assure that the database will not stand in the way of company growth.

All this is provided with user-friendly HTML5 management interface, ease of deployment and top-notch C# and Java client libraries.

- **Schema-free**
- **Scalable**
- **RavenFS**
- **Easy to use**
- **Transactional**
- **High Performance**
- **Extensible**
- **Designed with Care**

NEW

- **Monitoring**
- **Hot Spare**
- **Clustering**

RAVENDB 3.5 RELEASED

ravendb.net

Damir Arh

Concurrent Programming in .NET Core

Concurrency can help us improve performance of individual applications with asynchronous I/O operations and parallel processing. In .NET Core, "tasks" are the main abstraction for concurrent programming, but there are other helper classes as well that can make our job easier. This article will explore multithreaded concurrent programming in .NET Core.

.NET Core

ASYNCHRONOUS VS.MULTITHREADED CODE

Concurrent programming is a broad term and we should start with it by examining the difference between asynchronous methods and actual multithreading. Although .NET Core uses `Task<T>` to represent both concepts, there is a core difference in how it handles them internally.

Asynchronous methods run in the background while the calling thread is doing other work. This means that these methods are I/O bound, i.e. they spend most of their time in input and output operations, such as file or network access.

It makes a lot of sense to use asynchronous I/O methods in favor of synchronous ones, whenever possible. In the meantime, the calling thread can handle user interaction in a desktop application or process other requests in a server application, instead of just idly waiting for the operation to complete.

You can read more about calling asynchronous methods using `async` and `await` in my Asynchronous Programming in C# using Async Await – Best Practices article for the September edition of the DNC Magazine.

CPU-bound methods require CPU cycles to do their work and can only run in the background using their own dedicated thread. The number of available CPU cores, limits the number of threads that can run in parallel. The operating system is responsible for switching between the remaining threads, giving them a chance to execute their code. These methods still run concurrently, but not necessarily in parallel. This means that although the methods do not execute at the same time, one method can still execute in the middle of the other, which is paused during that time.

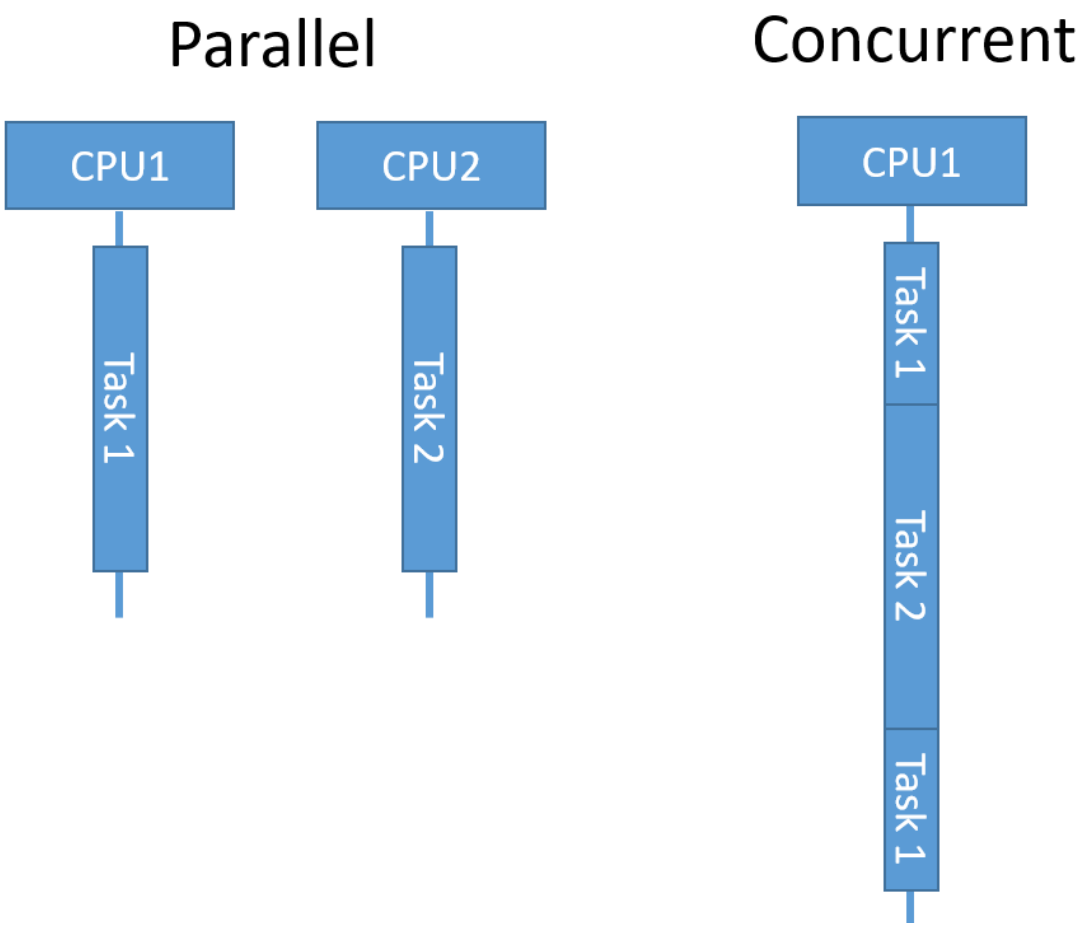


Figure 1: Parallel versus concurrent execution

This article will focus on **multithreaded concurrent programming in .NET Core** as described in the last paragraph.

Task Parallel Library

.

NET Framework 4 introduced Task Parallel Library (TPL) as the preferred set of APIs for writing concurrent code. The same programming model is adopted by .NET Core.

To run a piece of code in the background, you need to wrap it into a *task*:

```
var backgroundTask = Task.Run(() => DoComplexCalculation(42));
// do other work
var result = backgroundTask.Result;
```

`Task.Run` method accepts a `Func<T>` if it needs to return a result, or an `Action` if it does not return any result. Of course, in both cases you can use a lambda, just as I did in the example above to invoke the long running method with a parameter.

A thread from the thread pool will process the task. The .NET Core runtime includes a default scheduler that takes care of queuing and executing the tasks using the thread pool threads. You can implement your own scheduling algorithm by deriving from the `TaskScheduler` class and using it instead of the default scheduler, but this discussion is beyond the scope of this article.

In the sample we just saw, I accessed the `Result` property to merge the background thread back into the calling thread. For tasks that do not return a result, I could call `Wait()` instead. Both will block the calling method until the background task completes.

To avoid blocking the calling thread (i.e. in an ASP.NET Core application), you can use the `await` keyword instead:

```
var backgroundTask = Task.Run(() => DoComplexCalculation(42));
// do other work
var result = await backgroundTask;
```

By doing so, the calling thread will be released to process other incoming requests. Once the task completes, an available worker thread will resume processing the request. Of course, the controller action method must be asynchronous for this to work:

```
public async Task<IActionResult> Index()
{
  // method body
}
```

Handling Exceptions

Any exceptions thrown by the task will propagate to the calling thread at the point of merging the two threads back together:

- If you use `Result` or `Wait()`, they will be wrapped into an `AggregateException`. The actual exception thrown will be stored in its `InnerException` property.

- If you use `await`, the original exception will remain unwrapped.

In both cases, the call stack information will remain intact.

Cancelling a Task

Since tasks can be long running, you might want to have an option for cancelling them prematurely. To allow this option, pass a cancellation token when creating the task. You can use it afterwards to trigger the cancellation:

```
var tokenSource = new CancellationTokenSource();
var cancellableTask = Task.Run(() =>
{
  for (int i = 0; i < 100; i++)
  {
    if (tokenSource.Token.IsCancellationRequested)
    {
      // clean up before exiting
      tokenSource.Token.ThrowIfCancellationRequested();
    }
    // do long-running processing
```

```
  }
  return 42;
}, tokenSource.Token);
// cancel the task
tokenSource.Cancel();
try
{
  await cancellableTask;
}
catch (OperationCanceledException e)
{
  // handle the exception
}
```

To actually stop the task early, you need to check the cancellation token in the task and react if a cancellation was requested: do any clean up you might need to do and then call `ThrowIfCancellationRequested()` to exit the task. This will throw an `OperationCanceledException`, which can then be handled accordingly in the calling thread.

Coordinating Multiple Tasks

If you need to run more than one background task, there are methods available to help you coordinate them.

To run multiple tasks concurrently, just start them consecutively and collect references to them, e.g. in an array:

```
var backgroundTasks = new []
{
  Task.Run(() => DoComplexCalculation(1)),
  Task.Run(() => DoComplexCalculation(2)),
  Task.Run(() => DoComplexCalculation(3))
};
```

Now you can use static helper methods of the `Task` class to wait for their execution to complete synchronously or asynchronously:

```
// wait synchronously
Task.WaitAny(backgroundTasks);
Task.WaitAll(backgroundTasks);
// wait asynchronously
await Task.WhenAny(backgroundTasks);
await Task.WhenAll(backgroundTasks);
```

The two methods at the bottom actually return a task themselves, which you can once again manipulate like any other task. To get the task results, you can inspect the `Result` property of the original tasks.

Handling exceptions when working with multiple tasks is a bit trickier. `WaitAll` and `WhenAll` methods will throw an exception, whenever any of the tasks in the collection have thrown. However, while `WaitAll`'s `AggregateException` will contain all the thrown collections in its `InnerExceptions` property, `WhenAll` will only throw the first exception thrown by any of the tasks. In order to determine which task has thrown which exception, you will need to check `Status` and `Exception` properties of each individual task.

You need to be even more careful when using `WaitAny` and `WhenAny`. Both of them wait for the first task to complete (successfully or not), but do not throw an exception even if the task has thrown one. They only

return the index of the completed task or the completed task itself, respectively. You will need to catch the exception when awaiting the completed task or when accessing its result, e.g.:
var completedTask = await Task.WhenAny(backgroundTasks);

```
try
{
  var result = await completedTask;
}
catch (Exception e)
{
  // handle exception
}
```

If you want to run multiple tasks consecutively instead of in parallel, you can use continuations:

```
var compositeTask = Task.Run(() => DoComplexCalculation(42))
  .ContinueWith(previous => DoAnotherComplexCalculation(previous.Result),
TaskContinuationOptions.OnlyOnRanToCompletion)
```

The `ContinueWith()` method allows you to chain multiple task to be executed one after another. The continuing task gets a reference to the previous task to use its result or to check its status. You can also add a condition to control when to run the continuation, e.g. only when the previous task completed successfully or when it threw an exception. This adds flexibility in comparison to consecutively awaiting multiple tasks.

Of course, you can combine continuations with all the previously discussed features: exception handling, cancellation and running tasks in parallel. This gives you a lot of expressive power to combine the tasks in different ways:

```
var multipleTasks = new[]
{
  Task.Run(() => DoComplexCalculation(1)),
  Task.Run(() => DoComplexCalculation(2)),
  Task.Run(() => DoComplexCalculation(3))
};
var combinedTask = Task.WhenAll(multipleTasks);

var successfulContinuation = combinedTask.ContinueWith(task =>
  CombineResults(task.Result), TaskContinuationOptions.OnlyOnRanToCompletion);
var failedContinuation = combinedTask.ContinueWith(task =>
  HandleError(task.Exception), TaskContinuationOptions.NotOnRanToCompletion);
await Task.WhenAny(successfulContinuation, failedContinuation);
```

Task Synchronization

If tasks are completely independent, the methods we just saw for coordinating them will suffice. However, as soon as they need to access shared data concurrently, additional synchronization is required in order to prevent data corruption.

Whenever two or more threads attempt to modify a data structure in parallel, data can quickly become inconsistent. The following snippet of code is one such example:

```
var counters = new Dictionary<int, int>();
if (counters.ContainsKey(key))
{
  counters[key] ++;
```

```
}
else
{
  counters[key] = 1;
}
```

When multiple threads execute the above code in parallel, a specific execution order of instructions in different threads can cause the data to be incorrect, e.g.:

- Two threads both check the condition for the same key value when it is not yet present in the collection.
- As a result, they both enter the `else` block and set the value for this key to `1`.
- Final counter value will be 1 instead of 2, which would be the expected result if the threads would execute the same code consecutively.

Such blocks of code, which may only be entered by one thread at a time, are called **critical sections**. In C#, you can protect them by using the `lock` statement:

```
var counters = new Dictionary<int, int>();

lock (syncObject)
{
  if (counters.ContainsKey(key))
  {
    counters[key]++;
  }
  else
  {
    counters[key] = 1;
  }
}
```

For this approach to work, all threads must share the same `syncObject` as well. As a best practice, `syncObject` should be a private `Object` instance that is exclusively used for protecting access to a single critical section and cannot be accessed from outside. The lock statement will allow only one thread to access the block of code inside it. It will block the next thread trying to access it until the previous one exits it. This will ensure that a thread will execute the complete critical section of code without interruptions by another thread. Of course, this will reduce the parallelism and slow down the overall execution of code, therefore you will want to minimize the number of critical sections and to make them as short as possible.
The lock statement is just a shorthand for using the Monitor class:

```
var lockWasTaken = false;
var temp = syncObject;
try
{
  Monitor.Enter(temp, ref lockWasTaken);
  // lock statement body
}
finally
{
  if (lockWasTaken)
  {
    Monitor.Exit(temp);
  }
}
```

Although most of the time you will want to use the lock statement, Monitor class can give you additional

control when you need it. For example, you can use `TryEnter()` instead of `Enter()` and specify a timeout to avoid waiting indefinitely for the lock to release.

Other Synchronization Primitives

`Monitor` is just one of the many synchronization primitives in .NET Core. Depending on the scenario, others might be more suitable.

`Mutex` is a more heavyweight version of `Monitor` that relies on the underlying operating system. This allows it to synchronize access to a resource not only on thread boundaries, but even over process boundaries. `Monitor` is the recommended alternative over `Mutex` for synchronization inside a single process.

`SemaphoreSlim` and `Semaphore` can limit the number of concurrent consumers of a resource to a configurable maximum number, instead of to only a single one, as `Monitor` does. `SemaphoreSlim` is more lightweight than `Semaphore`, but restricted to only a single process. Whenever possible you should use `SemaphoreSlim` instead of `Semaphore`.

`ReaderWriterLockSlim` can differentiate between two different types of access to a resource. It allows unlimited number of readers to access the resource in parallel, and limits writers to a single access at a time. It is great for protecting resources that are thread safe for reading, but require exclusive access for modifying data.

`AutoResetEvent`, `ManualResetEvent` and `ManualResetEventSlim` will block incoming threads, until they receive a signal (i.e. a call to `Set()`). Then the waiting threads will continue their execution. `AutoResetEvent` will only allow one thread to continue, before blocking again until the next call to `Set()`. `ManualResetEvent` and `ManualResetEventSlim` will not start blocking threads again, until `Reset()` is called. `ManualResetEventSlim` is the recommended more lightweight version of the two.

`Interlocked` provides a selection of atomic operations that are a better alternative to locking and other synchronization primitives, when applicable:

```
// non-atomic operation with a lock
lock (syncObject)
{
  counter++;
}

// equivalent atomic operation that doesn't require a lock
Interlocked.Increment(ref counter);
```

Concurrent Collections

When a critical section is required only to ensure atomic access to a data structure, a specialized data structure for concurrent access might be a better and more performant alternative. For example, by using `ConcurrentDictionary` instead of `Dictionary`, the `lock` statement example can be simplified:
var counters = new ConcurrentDictionary<int, int>();

```
counters.TryAdd(key, 0);
lock (syncObject)
{
  counters[key]++; }
```

Naively, one might even want to use the following:

```
counters.AddOrUpdate(key, 1, (oldKey, oldValue) => oldValue + 1);
```

However, the update delegate in the above method is executed outside the critical section, therefore a second thread could still read the same old value as the first thread, before the first one has updated it, effectively overwriting the first thread's update with its own value and losing one increment. Even concurrent collections are not immune to multithreading issues when used incorrectly.

Another alternative to concurrent collections, is immutable collections.

Similar to concurrent collections they are also thread safe, but the underlying implementation is different. Any operations that change the data structures do not modify the original instance. Instead, they return a changed copy and leave the original instance unchanged:

```
var original = new Dictionary<int, int>().ToImmutableDictionary();
var modified = original.Add(key, value);
```

Because of this, any changes to the collection in one thread are not visible to the other threads, as they still reference the original unmodified collection, which is the very reason why immutable collections are inherently thread safe.

Of course, this makes them useful for a different set of problems. They work best in cases, when multiple threads require the same input collection and then modify it independently, potentially with a final common step that merges the changes from all the threads. With regular collections, this would require creating a copy of the collection for each thread in advance.

Parallel LINQ

Parallel LINQ (PLINQ) is an alternative to Task Parallel Library. As the name suggests, it heavily relies on LINQ (Language Integrated Query) feature, LINQ to Objects to be exact. As such, it is useful in scenarios, when the same expensive operation needs to be performed on a large collection of values. Unlike ordinary LINQ to Objects, which performs all the operations in sequence, PLINQ can execute these operations in parallel on multiple CPU cores.

To take advantage of that, the code changes are minimal:

```
// sequential execution
var sequential = Enumerable.Range(0, 40)
  .Select(n => ExpensiveOperation(n))
  .ToArray();

// parallel execution
var parallel = Enumerable.Range(0, 40)
  .AsParallel()
  .Select(n => ExpensiveOperation(n))
  .ToArray();
```

As you can see, the only difference between the two snippets of code is a call to `AsParallel()`. This converts an `IEnumerable<T>` to `ParallelQuery<T>`, causing the rest of the query to be run in parallel. To switch back to sequential execution in the middle of the query, you can call `AsSequential()`, which will return an `IEnumerable<T>` again.

By default, PLINQ does not preserve the order of the items in the collection to make the process more efficient. However, you can change that by calling `AsOrdered()`, when the order is important:

```
var parallel = Enumerable.Range(0, 40)
  .AsParallel()
  .AsOrdered()
  .Select(n => ExpensiveOperation(n))
  .ToArray();
```

Again, you can switch back by calling its counterpart: `AsUnordered()`.

Concurrent Programming in Full .NET Framework

Since .NET Core is a stripped-down reimplementation of the full .NET framework, all the described approaches to concurrent programming in .NET Core are also available in full .NET framework. The only exception to this are immutable collections, which are not an integral part of the full .NET framework. They are distributed as a separate NuGet package, System.Collections.Immutable, which you need to install in the project to make them available.

Conclusion

Whenever your application contains CPU intensive code which can run in parallel, it makes sense to take advantage of concurrent programming to improve performance and use the hardware more efficiently. The APIs in .NET Core abstract away many details and make writing concurrent code much easier. Still, there are potential issues to be aware of; most of them related to accessing shared data from multiple threads. If you can, you should avoid such situations altogether. When you cannot, make sure to select a synchronization method or data structure that is the most appropriate for your case ■

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.

A MAGAZINE FOR .NET AND JAVASCRIPT DEVS

- ☑ AGILE
- ☑ ASP.NET
- ☑ MVC, WEB API
- ☑ ANGULAR.JS
- ☑ NODE.JS
- ☑ AZURE
- ☑ VISUAL STUDIO
- ☑ .NET
- ☑ C#, WPF

We've got it all!

100K PLUS READERS

230 PLUS AWESOME ARTICLES

27 EDITIONS

FREE SUBSCRIPTION USING YOUR EMAIL

EVERY ISSUE DELIVERED

RIGHT TO YOUR INBOX

NO SPAM POLICY

SUBSCRIBE TODAY!

JAVASCRIPT

Ravi Kiran

getting started with React.js

JavaScript libraries and frameworks have taken the world of full stack development by storm. Because of the capabilities these frameworks provide, the JavaScript language in itself has been getting a lot of attention and as a result, is now being used in non-browser platforms like server, mobile apps and devices.

The usage of the language at this level has brought in a lot of challenges and as a result, has opened up new doors for innovation. Several open source developers and even big companies are investing heavily in contributing to this growing community, and helping to find solutions to the common problems developers face.

One of the popular front end libraries that solves problems related to the "View" or UI of an application is React.js from Facebook.

Why React js?

Facebook created the React library to address the age-old challenge of efficiently dealing with the View part of large-scale websites built using the Model-View-Controller (MVC) architecture. React js understands that DOM manipulation is an expensive operation, so it provides the developer a Virtual DOM. Whenever a change is made to a portion of the UI, React.js calculates the minimum number of DOM operations needed to achieve the new state. This allows you to render the page on every change without worrying about DOM performance.

React provides a declarative way to build custom UI components and render them on the page. It is built with simplicity and unidirectional data flow in mind. This is also called top-down approach, as the data always flows from the top UI element to the bottom. The library is also easy to learn and use.

A simple trend search shows its growing popularity in comparison to other JavaScript languages and frameworks:

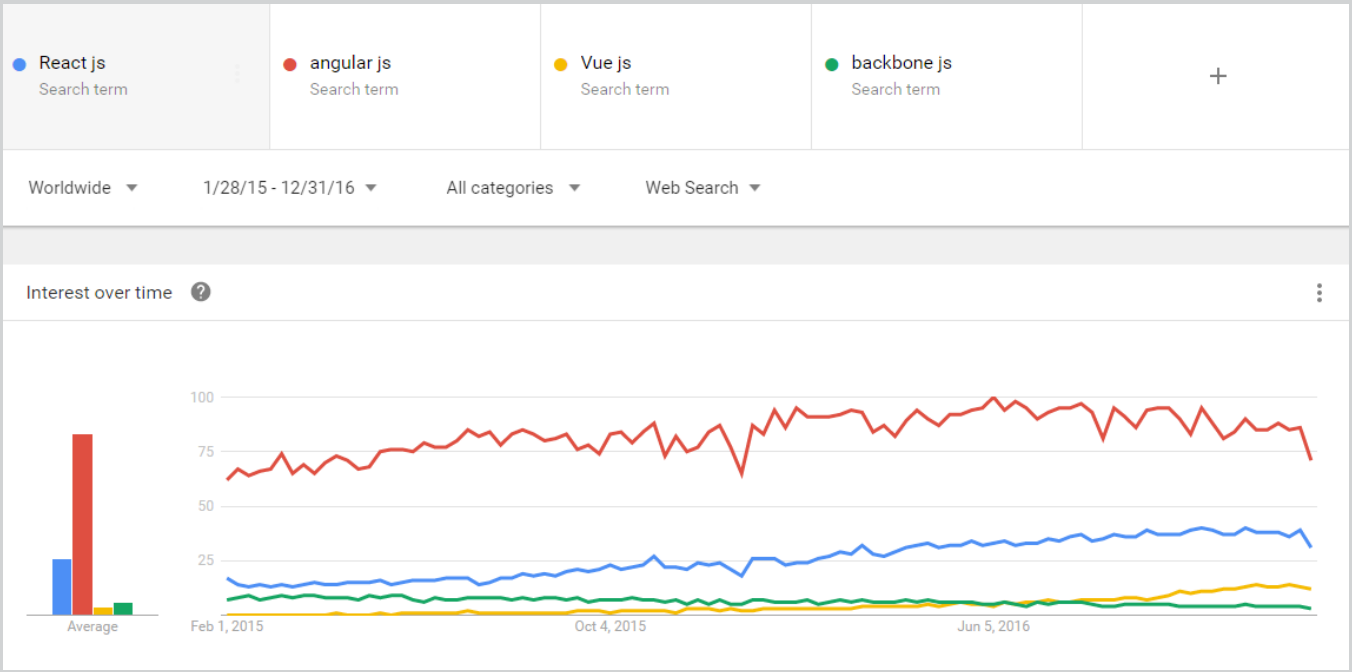


Figure 1: React js interest over time

This article will introduce you to the basic concepts of React js and will get you started with the library through a few simple examples.

Note: *It is very important to remember that React deals with only the UI (view). It is not a complete framework like Angular that deals with UI, Data, Bindings, Routing, HTTP Calls etc. To manage data flow and data in React apps, Facebook created Flux and Redux. Both Flux and redux have libraries to support the usage of these patterns in React applications. There are third party libraries or react plugins to get other features like Routing. To build a full SPA using React, we need to make these libraries work together. Discussing these features is out of scope for this article. Redux is covered in another article by Gil Fink. We will discuss the other libraries mentioned in this article in the near future.*

React.js - Basic Concepts

Before we write a basic React.js application, let's spend a moment to understand the concepts around the React library.

Virtual DOM

We are all aware of the DOM created and managed by browsers. React creates another DOM, called **Virtual DOM** to work with the application. The Virtual DOM is updated whenever the data of the component is modified. After updating the Virtual DOM, React finds the difference between the states of previous Virtual DOM and the current one. Then it renders only the portions where it finds difference between the two and not the whole UI. This makes the applications running on React more performant, as only the required portions of the UI are updated when there is a change in data.

Components

Components are custom HTML elements with business specific behavior. In React, every component is created as a component class. A component has a `render` method, which returns the view to be rendered when the component is used on the page. The component also manages the data used for binding the view.

A component in React can be created either using the `React.createClass` method or as a JavaScript class by extending the `Component` class in React. The following snippet shows a simple component:

```
var First = React.createClass({
  render(){
    return <div>This is my first component!</div>;
  }
});
```

JSX

JSX adds XML to JavaScript. The views in the react components are defined using JSX. The HTML elements and the other react components to be rendered inside the component have to be mentioned in the component using JSX. It allows us to bind data on the HTML elements, handle events using the methods on the component and bind values to the properties of the HTML elements. Every component in React has to implement a method named `render`, this method has to return the JSX to be presented on the view.

The component example in the previous section shows a simple example of JSX. We will see more examples later.

React Component Lifecycle Methods

Every React component goes through a series of lifecycle events. React provides us with lifecycle methods to run a piece of logic when these events occur. The lifecycle events occur when an instance of the component gets created, when the component has to be updated and when the component has to be removed from the page. The following are some of the lifecycle methods provided by react:

- componentWillMount: Gets invoked when the component is about to mount on the page
- componentDidMount: Gets invoked after the component is rendered on the page
- componentWillUpdate: Gets invoked before a component is updated after a change in data
- componentWillReceiveProps: Gets invoked when the component receives properties or when the properties are updated
- componentDidUpdate: Gets invoked immediately after the component is updated because of change in the data of the component
- componentWillUnmount: Gets invoked before a component is removed from the DOM. It can be used to perform any clean up tasks on the component

State, Props and Unidirectional data flow

Every component has a state object that plays the role of view model for the component. React works

on immutable data, so the state object has to be re-created whenever there is a change in the value. The lifecycle methods `componentWillUpdate` and `componentDidUpdate` are invoked when the value of state changes.

A component can receive inputs from another component through attributes on the component element. They are called Props. The lifecycle method `componentWillReceiveProps` is invoked when the props are set or changed. As the values of the props are passed from the component containing the current component, values of the props can be modified only in the parent component. When these values change, it triggers the `componentWillReceiveProps` life cycle method of the child component.

The data in a React application flows from top to bottom. A typical React application can be seen as a tree of components. The data always flows from the components at the higher level, to the components at the lower level. React doesn't support flow of data from the lower level components to higher level components. The unidirectional data flow is a performance booster for applications, as the library has to now perform lesser number of checks to update the UI. The unidirectional data flow also makes the behavior of the application predictable, as there won't be any cyclic dependencies between the components.

Building a React js Hello World Example on Codepen

Now that we are familiar with the core concepts of React, let's explore the usage of these concepts through an example. Like any other front-end technology, the development environment for React needs Node.js to be installed.

Download and install Node.js from the official website, if you don't already have it installed. This installer installs Node.js and the package manager npm. A React application can be written using any version of JavaScript as well as JavaScript preprocessors like TypeScript. The JavaScript static type checker Flow can also be used in React applications.

For this article, we will stick to JavaScript and use the ES2016 version of JavaScript, as it is the current version of JavaScript and is the recommended version to be used these days.

To get familiar with the minimum set of things needed to write a react application, let's build a small demo using Codepen *which is a very useful website to share front-end based demos. You can quickly write your demo using HTML, CSS and JavaScript and share it with others. It also supports JavaScript transpilers like Babel, TypeScript, CoffeeScript and CSS preprocessors like LESS, SCSS to allow for demos using the tools we are comfortable with.*

Note: *In addition to npm, we have one more package manager for JavaScript packages called* **yarn**. *Yarn is created by Facebook to address most of the issues developers face while working with npm. Yarn is a fast, secured and a reliable package manager. It doesn't have a new package repository, it works on npm and helps in maintaining the right versions of the packages. It is gaining a lot of popularity these days because of these features. To learn more about yarn, you can visit the official site https://yarnpkg.com/en/.*

Open the site codepen.io in your favorite browser. Click on the New Pen button on the top right corner to create a new demo. To write a react application, we need to apply a few settings to codepen. Click the Settings button located in the menu on the top right corner of the page. The settings dialog has multiple tabs to apply different settings for HTML, CSS and JavaScript. Switch to the JavaScript tab. Here we need to set the transpiler and apply the libraries required to work with React. Apply the settings as shown in the following figure:

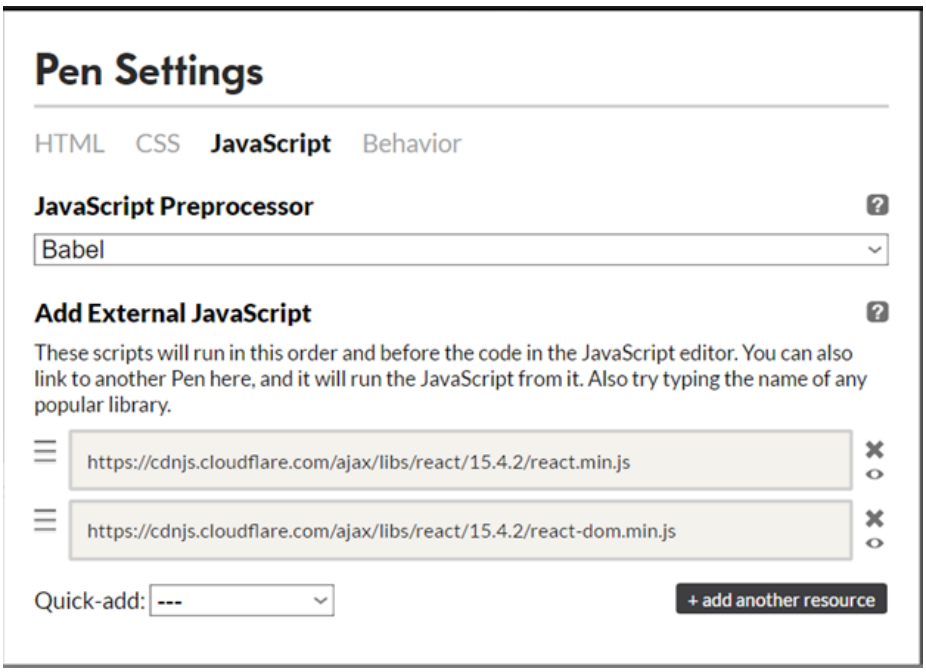


Figure 2: CodePen Settings for React js demo

The following changes are made in the above dialog:

- Changed the JavaScript Preprocessor to Babel
- The JavaScript libraries of React (https://cdnjs.cloudflare.com/ajax/libs/react/15.4.2/react.min.js) and React DOM (https://cdnjs.cloudflare.com/ajax/libs/react/15.4.2/react-dom.min.js) are added

Click the *Save and Close* button below the dialog to apply these changes. Now this pen has all the required resources. Let's write a hello world kind of a sample first. Add the following div element to the HTML template of the pen:

```
<div id="root"></div>
```

We will render a React component inside this div. Add the following code to the JavaScript pane in the pen:

```
var Hello = React.createClass({
  render: function(){
    return (
      <div>This is my first react component!</div>
    );
  }
});
```

The above component is created using ES5 syntax. As you can see, it uses the `createClass` method on the `React` object to create the component. The `render` function in the component object returns the JSX object to be rendered inside the component. Though it looks like HTML now, it is not pure HTML. We will see more examples of JSX shortly.

Now this component has to be rendered on the page. To do so, add the following statement to the JavaScript pane:

```
ReactDOM.render(<Hello />, document.getElementById('root'));
```

Observe the way the `Hello` component is used in the template. The object of the component is used in JSX passed to the `ReactDOM.render` method. The second argument passed to this method is the target DOM element where the component has to be rendered. Here, we are selecting the div added to the template and passing it to the `render` method. The object itself is the selector for the component. Now you will see the output of this code below the code section. The following screenshot shows how codepen looks like after these changes:

Figure 3: React Hello Word component

The message shown in the component is now hard coded in the div element. Let's refactor this code to move the message into the state object and use it from there. The following snippet shows the modified Hello component:

```
var Hello = React.createClass({
  getInitialState: function(){
    return { message: “This is my first react component!” };
  },
  render: function(){
    return (
      <div>{this.state.message}</div>
    );
  }
});
```

The component now has a new method, `getInitialState`. This method returns the `state` object of the component. The `state` object can be used to bind data and events on the component's view. The div element in JSX is now modified to bind the message using the `state` object. The output of this code still remains the same.

Let's set a new value to the message after 5 seconds. Doing so involves changing the value in the `state` object. As mentioned earlier, the state object is immutable and any modifications to this object will create a new `state` object. React provides us with the `setState` method on every component to modify the value of the `state` object. And it is better to set the timeout to change the message after the component is initialized. We need to use the `componentDidMount` lifecycle method for this. Add the following method to the component:

```
componentDidMount: function(){
  setTimeout(() => {
    this.setState({ message: “This message is modified after 5 seconds!” });
  }, 5000);
}
```

Now you will see that the message displayed inside the div element gets modified after 5 seconds. The

`setState` method accepts the change to be applied to the `state` object and it creates a new `state` object and modifies the change it received.
You can find the codepen containing this demo over at this link - codepen.io/sravikiran/pen/wgZRRB?editors=1010

Building a More Meaningful React js Sample

Now that we have completed the hello-world tradition, let us build a more meaningful example to explore a few more features of React.

As we saw in the codepen sample, to build a React application, we need a transpiler like babel. For most React applications, babel is the de facto transpiler as Facebook uses babel quite extensively and there are a number of babel plugins built to support the development of React. These plugins are available as a babel preset, named `babel-preset-react`. Along with these babel packages, we need a good development setup.

The task of setting up a development environment for React takes some time and effort. To save some setup time, we have the npm package `create-react-app`. We will look at the process of environment setup later in a separate article. To keep this article focused on the features of React, we will use the generator `create-react-app`. This package has to be installed globally using npm. The following command does this:

```
> npm install –g create-react-app
```

Once the above command runs successfully, we can create a new React application by running the following command from any location in the system:

```
> create-react-app my-app
```

This command creates a new folder with the name `my-app` and adds a few files and folders inside it. It also installs all the required dependencies for the project. We can start writing react application directly without worrying too much about the setup. The application created already has some basic code in it. The following image shows the folder structure of the application created in Visual Studio Code:

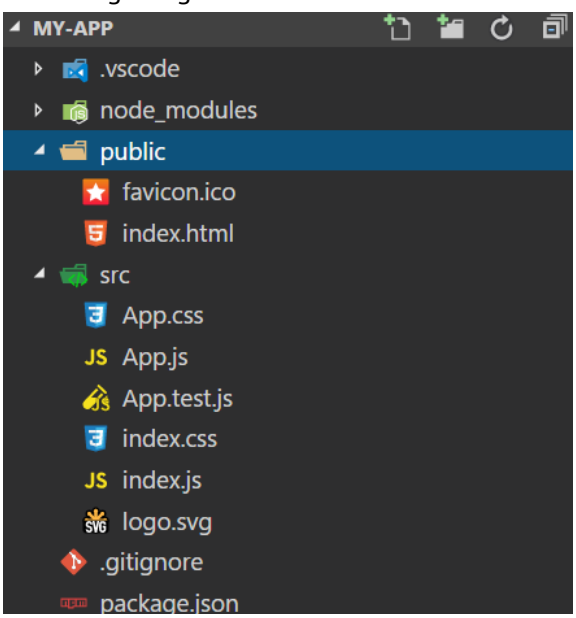


Figure 4: React js VS Code folder structure

The generated application already has some React code. It has a component named `App` and the component is rendered on the page in the file `index.js`. To start this application, open a command prompt and run the following command:

```
> npm start
```

This command starts the application and launches it in your default browser. You will see the following screen on the browser:

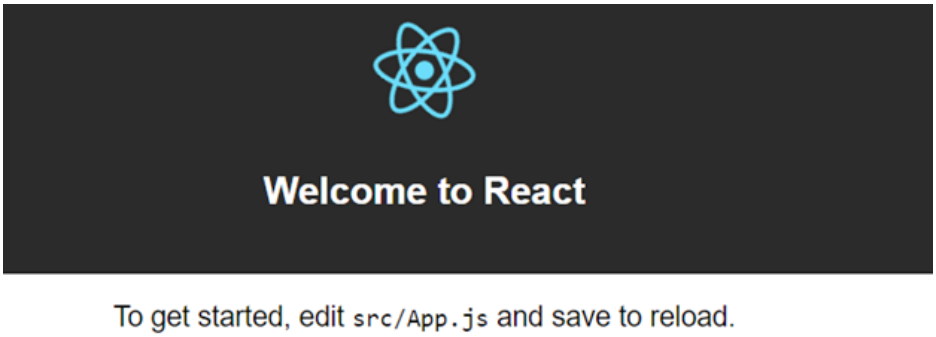


Figure 5: React js app running in browser

We will modify this sample to add two new components and use them in the main App component. We are going to display a list of people working in an imaginary software company and will allow the user to filter the data based on different parameters.

Add a new file to the application and name it ListView.js. This file will contain a component displaying the list of employees. The following snippet shows the code of this component:

```
import React, { Component } from 'react';
export class ListView extends Component {
  constructor(props) {
    super(props);
    this.state = {
      employees: [
        {
          name: "Alex Kesler",
          designation: "Software Architect",
          project: "URR"
        },

        {
          name: "Raghu Shah",
          designation: "Software Engineer",
          project: "SHU"
        },

        {
          name: "Kim Lee",
          designation: "Intern",
          project: "URR"
        },

        {
          name: "Joe Walsh",
          designation: "Manager",
          project: "SHU"
        },
```

```
        {
          name: "Christine Sam",
          designation: "QA Engineer",
          project: "FHD"
        },

        {
          name: "Tim Asermeley",
          designation: "UX Designer",
          project: "FHD"
        },

        {
          name: "Raji Sinha",
          designation: "Tech Lead",
          project: "SHU"
        }
      ]
    };
  }

  render() {
    let jsx = <table style={{ display: "inline-table" }}>
      <thead>
        <tr>
          <th>Name</th>
          <th>Designation</th>
          <th>Project</th>
        </tr>
      </thead>
      <tbody>
        {
          this.state.employees.map(function (employee) {
            return <tr key={employee.name}>
              <td>{employee.name}</td>
              <td>{employee.designation}</td>
              <td>{employee.project}</td>
            </tr>
          })
        }
      </tbody>
    </table>;

    return jsx;
  }
}
```

This component looks different from the component we built earlier. As we are using the ES2015 syntax now, we are creating a class for the component rather than calling the `React.createClass` method on the `React` object. React supports this way to create the component while using the ES2015 syntax to write JavaScript.

To set the `state` object in the class, we don't have to write the `getInitialState` method as the constructor of the class has to initialize the state. So we are setting the state object directly in the constructor. One more thing to be noticed in the constructor is the call to the `super()` class constructor. This call passes the properties of the current component.
Some of you may have a question on support of this special mix of JavaScript and XML in Visual Studio

code. Thankfully, the editor has good support for the JSX syntax. It also supports emmet, also known as zen coding of HTML inside the JSX blocks.

Take a close look at JSX of this component. It has a table that iterates through the entries in the `employees` array and prints rows on the screen. The table element has a style applied. The following snippet shows the table element:

```
<table style={{ display: “inline-table” }}>
```

The value of the style is applied in an expression. As the display type of the table is set to a static value, it is assigned using double quotes. The value to the display type can be set using a variable as well.
To loop through the `employee`*s* array, we are using the `map` method of the array in JSX and the `map` method builds the row to be printed for every record. The following snippet shows the loop that goes through the `employees` list:

```
this.state.employees.map(function (employee) {
  return <tr key={employee.name}>
    <td>{employee.name}</td>
    <td>{employee.designation}</td>
    <td>{employee.project}</td>
  </tr>
})
```

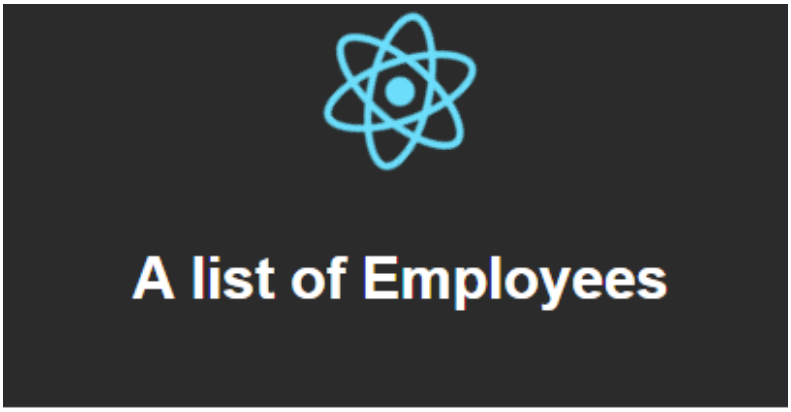
If you worked on jQuery before, this loop looks similar to the way we used to hand build the markup inside the loops. But now we are building the markup to be rendered more natively, rather than doing it inside a string. And we are using data binding to bind the values of the objects rather than appending them using string concatenation. This snippet shows the essence of JSX, the right mix of markup and the JavaScript functionality.

To use this component in the `App` component, we need to import the component there and use it in the JSX of the `App` component. The code is shown below:

```
import React, { Component } from ‘react’;
import { ListView } from ‘./ListView’;
import logo from ‘./logo.svg’;
import ‘./App.css’;

class App extends Component {
  render() {
    let jsx = <div className=”App”>
      <div className=”App-header”>
        <img src={logo} className=”App-logo” alt=”logo” />
        <h2>A list of Employees</h2>
      </div>
      <ListView />
    </div>;
    return (
      jsx
    );
  }
}
export default App;
```

Now the page looks like the following image:



Name	Designation	Project
Alex Kesler	Software Architect	URR
Raghu Shah	Software Engineer	SHU
Kim Lee	Intern	URR
Joe Walsh	Manager	SHU
Christine Sam	QA Engineer	FHD
Tim Asermeley	UX Designer	FHD
Raji Sinha	Tech Lead	SHU

Figure 6: List of Employees

React Filter example

Now let's add capabilities to filter this data. For this, we will create another component to show the controls required to filter the data and communicate the applied filters to the `ListView` component. Let's build the component to filter the data. The following snippet shows this component:

```
import React, { Component } from 'react';
import { ListView } from './ListView';
export class Filter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      field: "name",
      value: ""
    };
    this.filter = this.filter.bind(this);
    this.setField = this.setField.bind(this);
  }
  render() {
    let jsx = <div>
      <div>Field: <select onChange={this.setField}>
        <option value="name">Name</option>
        <option value="designation">Designation</option>
        <option value="project">Project</option>
      </select></div>
      <div>Value: <input type="text" onChange={this.filter} /></div>
      <ListView field={this.state.field} value={this.state.value} />
    </div>;
    return (
      jsx
    );
```

```
  }
  setField(event) {
    this.setState({ field: event.target.value });
  }
  filter(event) {
    this.setState({ value: event.target.value });
  }
}
```

This component has a drop down to select the field on which the filter has to be applied. It also has a textbox to accept the value to be used for filtering. Notice that the dropdown and the text box have their `onChange` event bound with the methods in the component class. These methods are invoked when the change event is triggered on the HTML elements.

Notice the last two statements inside the constructor where the current object of the component is bound with the methods `filter` and `setField`:

```
this.filter = this.filter.bind(this);
this.setField = this.setField.bind(this);
```

This is done to ensure that meaning of this remains the current component inside the methods handling events. Otherwise, the event handling methods are called in the global context and we won't have access to the members defined inside the component class.

The JSX of the `Filter` component uses the `ListView` component and it passes the values of `state.field` and `state.value` to the `ListView` component. These values will be accessible inside the `ListView` component through `props`. When these values are modified, React will invoke the `render` method of the `ListView` component. So, we need to use these values in the `render` method to filter the data and create the JSX of the component. The following snippet shows the modified `render` method of the `ListView` component and a method to filter the data:

```
filterEmployees() {
  let employees = this.state.employees;
  if (this.props.field && this.props.value) {
    employees = this.state.employees.filter((e) => {
      return e[this.props.field].toLowerCase().indexOf(this.props.value.
      toLowerCase()) >= 0;
    });
  }
  return employees;
}

render() {
  let employees = this.filterEmployees();
  let jsx = <table style={{ display: "inline-table" }}>
    <thead>
      <tr>
        <th>Name</th>
        <th>Designation</th>
        <th>Project</th>
      </tr>
    </thead>
    <tbody>
      {
        employees.map(function (employee) {
          return <tr key={employee.name}>
            <td>{employee.name}</td>
```

```
            <td>{employee.designation}</td>
            <td>{employee.project}</td>
          </tr>
        })
      }
    </tbody>
  </table>;
  return jsx;
}
```

Now all we need to do is, update the App component to use the Filter component. The following snippet shows the modified App component:

```
import React, { Component } from 'react';
import { Filter } from './Filter';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    let jsx = <div className="App">
      <div className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <h2>A list of Employees</h2>
      </div>
      <Filter />
    </div>;
    return (
      jsx
    );
  }
}
export default App;
```

Now run the application after saving these changes. You will see the filters on the page and the data in the table getting updated when the filters are modified. The following screenshot shows an instance of the page when the filters are applied:

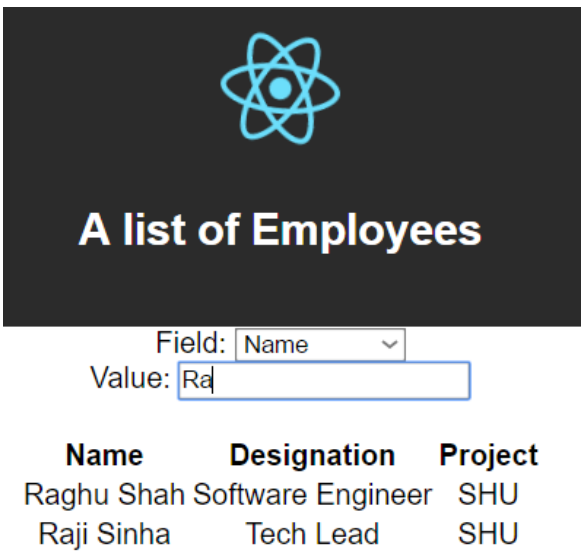


Figure 7: React js filtering data

Conclusion

Component based declarative programming is the approach developers want to use in their daily work these days. React is a library that supports this model of development. This article is an attempt to let you understand what React is and to get you started with it. We will explore more features of React in our forthcoming tutorials ■

Download the entire source code from GitHub at
bit.ly/dncm29-reactjs

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 and Suprotim Agarwal for reviewing this article.

Craig Berntson

Reduce Bad Code with Code Reviews

We all agree, bad code negatively affects applications. It introduces bugs. It is hard to maintain. It is difficult to enhance and extend. In the January 2017 issue of DNC Magazine I discussed code standards as one way to improve the code you and your team produce. We've also heard for years that unit tests are a key step for improving code quality. I agree with this.

Unit tests are very important.

However, you can do better. It's by using something that most teams either don't do or don't do effectively. I'm talking about **Code Reviews**.

IBM engineer Michael Fagan first documented the idea of code reviews in 1976 when he wrote about them in the magazine article "Design and Code Inspections to Reduce Errors in Program Development." (bit.ly/dncm29-fagan)

The importance of code reviews continued to be documented. In the 2011 Dr. Dobbs article "Do You Inspect?", Capers Jones and Olivier Bonsignour wrote

Recent work by Tom Gilb, one of the more prominent authors dealing with software inspections, and his colleagues continues to support earlier findings that a human being inspecting code is the most effective way to find and eliminate complex problems that originate in requirements, design, and other non-code deliverables. Indeed, to identify deeper problems in source code, formal code inspection outranks testing in terms of defect-removal efficiency levels." bit.ly/dncm29-drdobb

Let me repeat the most important part of this statement,

> *"a human being inspecting code is the most effective way to find and eliminate complex problems".*

Notice they didn't say code standards. They didn't say functional testing. They didn't even say unit testing. They said, "a human being."

This is backed up elsewhere. In the book **Code Complete** (bit.ly/dncm29-codecomplete), Steve McConnell writes, *"software testing alone has limited effectiveness – the average defect detection rate is only 25 percent for unit testing, 35 percent for function testing, and 45 percent for integration testing. In contrast, the average effectiveness of design and code inspections are 55 and 60 percent."*

More recently, in 2016, SmartBear Software conducted a survey on improving code quality then proclaimed that code reviews are "The #1 Way to Improve Software Quality" (bit.ly/dncm29-smartb). When asked "What do you feel is the number one thing a company can do to improve code quality?", 34% of respondents said, "Code Review".

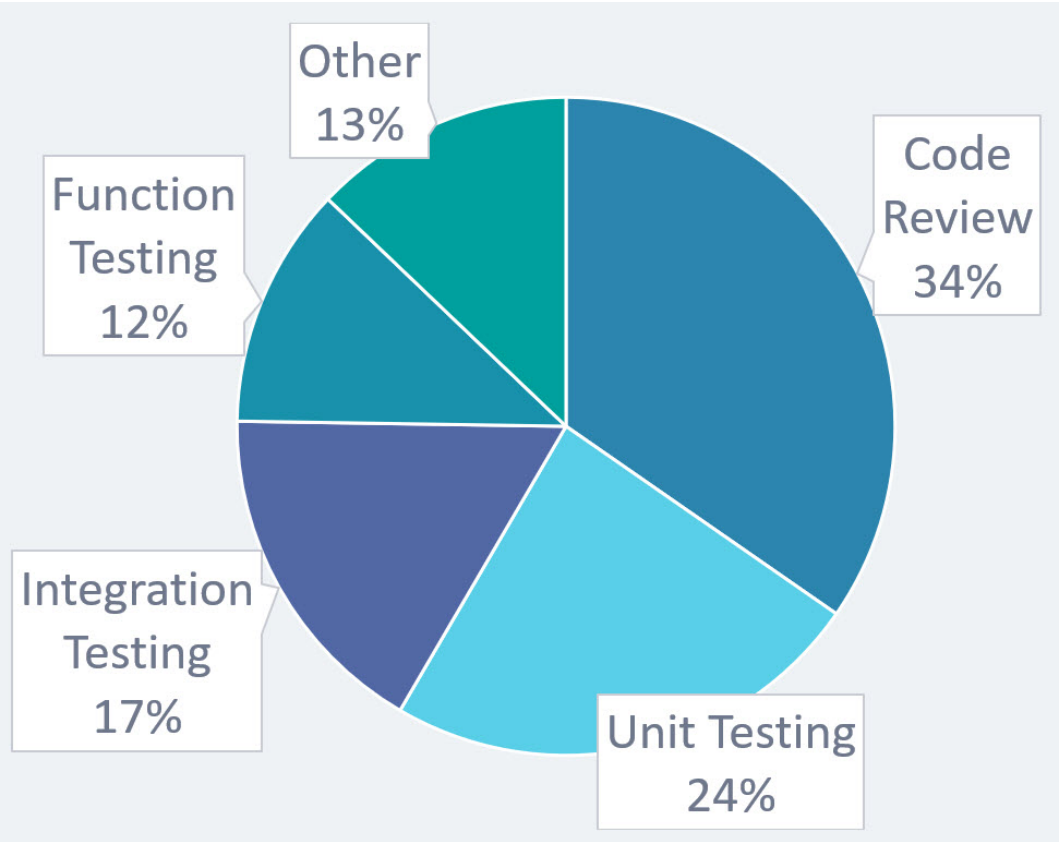


Figure 1: Code Quality Survey

It's pretty clear that code reviews are important.

IMPACT OF POOR QUALITY

Let's now look at how poor quality affects software. Way back in 2002, the National Institute of Standards and Technology, a division inside the US Department of Commerce, produced a 203 page document called *The Economic Impact of Inadequate Infrastructure for Software Testing* (http://www.nist.gov). Among other things, this study documented the number of hours it took to fix a bug that is found at different phases in the application lifecycle. They showed that a bug introduced in the coding phase but discovered in production, took an average of 14.8 hours to fix. But if found during the coding stage, it took 3.2 hours.

	Stage Found				
Stage Introduced	**Requirements**	**Coding/Unit Testing**	**Integration**	**Beta Testing**	**Production**
Requirements	1.2	8.8	14.8	15.0	18.7
Coding/Unit Testing	NA	3.2	9.7	12.2	14.8
Integration	NA	NA	6.7	12.0	17.3

Figure 2: Time taken to fix a bug at different phases in the application lifecycle

The software company Parasoft analyzed the effect of publicly announced bugs from publicly held companies. Looking at only 2014, they found the average value of the company dropped 3.75% the day

the failure was announced. That may not sound like much when viewed as a percentage, but in dollars, it is a whopping $2.3 billion loss of value. That's just the day of the announcement. Cumulative effects were worse.

FullStory, a Silicon Valley startup created by former Google engineers called code reviews "a bionic cultural hammer" meaning that when you introduce code reviews to your company, it has a huge impact on the culture. They outlined five ways this happens:

- Promotes Openness – Sets the tone for the entire company. Work is scrutinized by other team members and is welcome.

- Raises Team Standards – Makes sure all engineers share a similarly high level of standards.

- Propels Teamwork – Team members reconcile different viewpoints between reviewer and reviewee. In these cases, the manager doesn't have to step in and "be the adult." It also improves team communication.

- Keeps Security Top of Mind – Team members learn security hands-on and in context of their domain. Additionally, reviewers and reviewees constantly update their security knowledge.

- Shapes the Culture – People start to think in terms of quality and secure code.

Finally, 84% of participants in the SmartBear survey answered with Improved Quality when asked, "What do you think are the most important benefits of code review?"

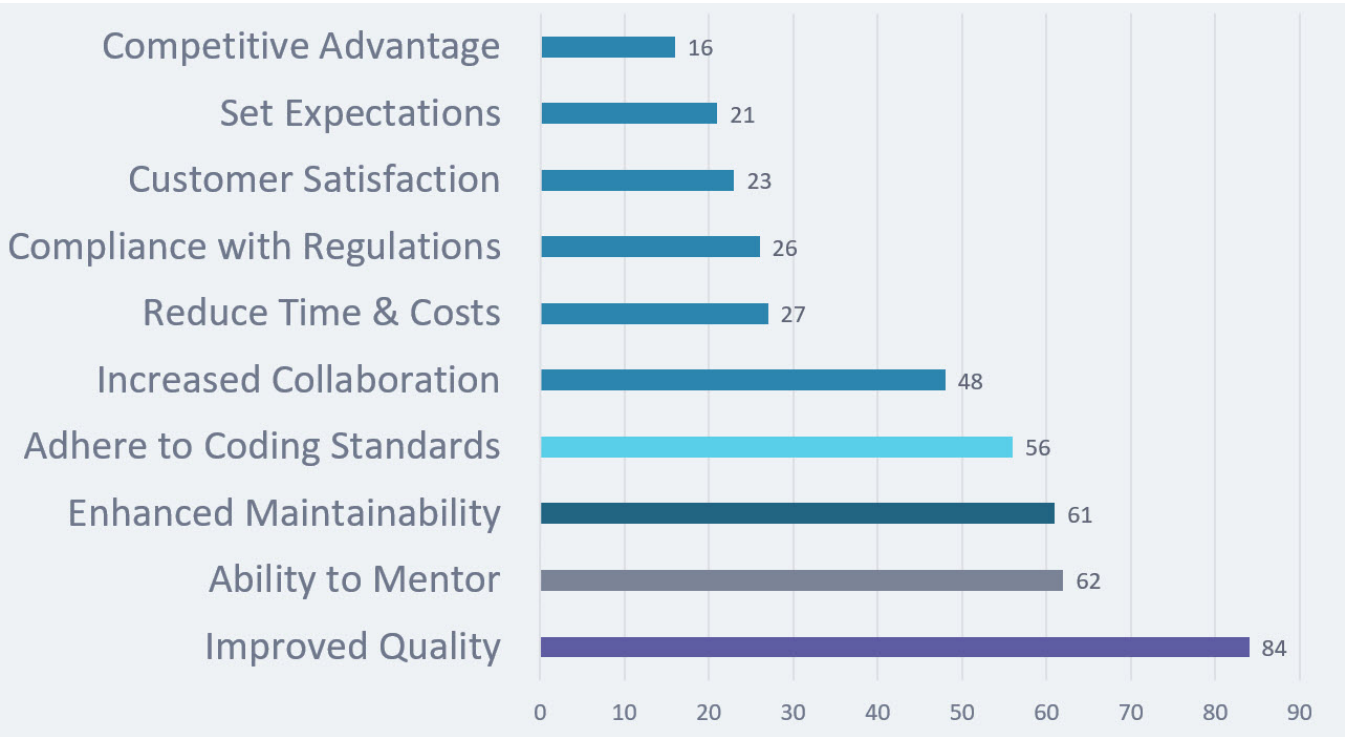


Figure 3: Most important benefits of Code Review

When you attempt to introduce code reviews to your team, you may get some push back. First, some may say that it adds time to the schedule. Go back to the NIST analysis of when bugs are discovered and you'll see that's not true. If a bug is found earlier, it takes less time and money to fix.
Second, it can become a battle of who is a better coder. A code review is not a contest. It is used to find

areas the code and coder can improve. Even the best coders can write poor code. I know this from personal experience as both the reviewer and reviewee.

At this point, I have laid out a good case for conducting code reviews but have not defined what a code review is.

WHAT EXACTLY IS A CODE REVIEW?

Wikipedia provides the following definition: *"A code review is systematic examination (sometimes referred to as peer review) of computer source code. It is intended to find mistakes overlooked in the initial development phase, improving the overall quality of software."* That falls in line with what you've seen so far.

In his Pluralsight course, "Lessons from Real World .NET Code Reviews" (bit.ly/dncm29-ps-course), Shawn Wildermuth says that a code review determines what is being done well and what can be done better, it is not a witch hunt, sharing findings makes all developers involved better, and no one size fits all.

This last point is interesting as it implies there are different ways to do code reviews. And, in fact, there are.

At a high level there are four primary types of code reviews:

- Formal – A formal code review meeting is held. All participants gather around a conference room table to go through the code. Reviewers get the code ahead of the meeting and go through, noting things that are good and bad.

- Informal – Rather than a meeting, reviewers are generally emailed the code. They then reply with their comments.

- Automated – Software analyzes code and flags suspect code that doesn't follow guidelines or has other potential issues. Human reviewers then enter additional comments that are sent back to the reviewee.

- Pull Request – Tools like GitHub allow reviewers to look at the Pull Request and comment on it before it's merged. However, it can be difficult to see the changed code in context of the entire method or class.

You may find that one of a combination of these types of code reviews work best for your team. When I do my Code Reviews presentation at conferences I am often asked how often should you do a code review. The correct answer is before every merge or check-in. Now let's turn to the actual practice of a code review.

WHAT TO DO WHEN YOU'RE THE REVIEWER

So, you have in front of you code that a team member has written and you need to code review it. The things you do are the same no matter if you're doing a formal review or a pull request. What do you do? Where do you start? First step is to try to understand the problem being solved and then ask, "Does this code solve the problem?" A good second question is, **"Where are the unit tests and do they pass?"**

After this, it's time to look how the problem is solved. By this, I don't mean to analyze every little thing and compare it to your coding standards. For example, your coding guidelines may say that closing braces should be on their own line. If this is violated once in the code, it was probably overlooked. But if the reviewee consistently violates this standard, you should probably call it out.

Other things to look for include patterns and anti-patterns, what the reviewee is doing well and what they consistently do wrong, are there just enough comments, does the code belong where it is or should it be in its own method or class? This is big picture things that make the code easier to modify.

In his Pluralsight course, Shawn Wildermuth points out several ideas:

- Take your experience to bear. You may have more experience than the reviewee. The review can become mentoring time. The opposite can also be true. If you are less experienced than the reviewee, you can learn some new techniques.

- Find things to laud and things to fix

- No code is perfect. Maintainability is a feature.

- Don't review too much code at a time.

MSDN.com (bit.ly/dncm29-msdn) also give some good advice. *"Don't review for code style; there are much better things on which to spend your time, than arguing over the placement of white spaces. Looking for bugs and performance issues is the primary goal and is where the majority of your time should be spent. Suggestions on maintenance best practices can be brought up after the bugs have been discussed."*

One final thing to keep in mind is what I call the "1 to M" rule which is, "Code is written once but read many times." This means that fancy or tricky coding should be avoided. If you can't figure out what the code is doing by reading it, it wasn't written well.

Before moving on, there are also things you should not do as the reviewer:

- Don't get emotional – We all have team members that are more difficult to work with than others. If you're reviewing code from that person, keep emotion out of the review.

- Don't focus blame – Instead of saying things like "Why did you not follow the Single Responsibility Principle?" say "How does this code comply with the Single Responsibility Principle?"

- Don't redesign the code – If you're redesigning the code, you should probably write it in the first place. If the code is really, really bad, make this a mentoring moment.

- Don't judge how you would have done it – Often a more senior developer reviews the code written by a junior developer. Because of having more experience, you tend to look at the problem with different eyes. Again, maybe a good candidate to mentor.

- Don't include one-off problems – This comes back to the earlier comment about closing braces. Look for things the reviewee does frequently.

Now that we've addressed what the reviewer should and should not do, let's flip the table and talk about the person who wrote the code.

WHAT TO DO WHEN YOU'RE THE REVIEWEE

There are some things that should be done by the person who wrote the code before submitting the code for review.

• Have a checklist of review items – Keep a list of items that you frequently do wrong and correct them ahead of time. Also, have a list of items that reviewers often look for and check for these things too. In other words, do your own personal code review.

• Help maintain the coding guidelines – This gives you the opportunity to learn how to write better code. Are there things missing that should be there? Do things need to be removed or added?

• Remember the code isn't you – We get attached to our code and tend to take criticism of it personally. We can learn from reviewers even if we have more experience than they do.
Code reviews are setup to make sure the code is in good condition. But we can also learn from them.

WRAP-UP

Whether you are the reviewer or the reviewee, there are things you need to consider in the review process and not make it about the coder, but rather about the code. Code reviews can sometimes become a mentoring experience.

Code reviews are an important part of the development process. If you're not doing reviews, you should talk to your team about getting them started. If you are doing reviews, when was the last time you reviewed the code review process to determine if it's effective or should be tweaked to improve it. By doing reviews, you can help ensure your application is green, lush, and vibrant.

ABOUT SOFTWARE GARDENING

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

Microsoft® MVP Most Valuable Professional

Craig Berntson
Author

Craig Berntson works for one of the largest mortgage companies in the US where he specializes in middleware development and helping teams get better. He has spoken at developer events across the US, Canada, and Europe for over 20 years and is a Grape City Community Influencer. Craig is the coauthor of 'Continuous Integration in .NET' available from Manning. He has been a Microsoft MVP since 1996. Craig lives in Salt Lake City, Utah. Email: dnc@craigberntson.com Twitter: @craigber

Thanks to Damir Arh and Suprotim Agarwal for reviewing this article.

Daniel Jimenez Garcia

Using ElasticSearch, Kibana, .NET Core and Docker

to Discover and Visualize data

Can you easily perform queries over your data in many different ways, perhaps in ways you have never anticipated? Are you able to visualize your logs in multiple ways while supporting instant filtering based on time, text and other types of filters?

These are just 2 examples of what can be easily achieved with the Elastic stack in a performant and scalable way.

In this article, I will introduce the popular search engine **Elasticsearch**, its companion visualization app **Kibana** and show how **.Net Core** can easily be integrated with the Elastic stack.

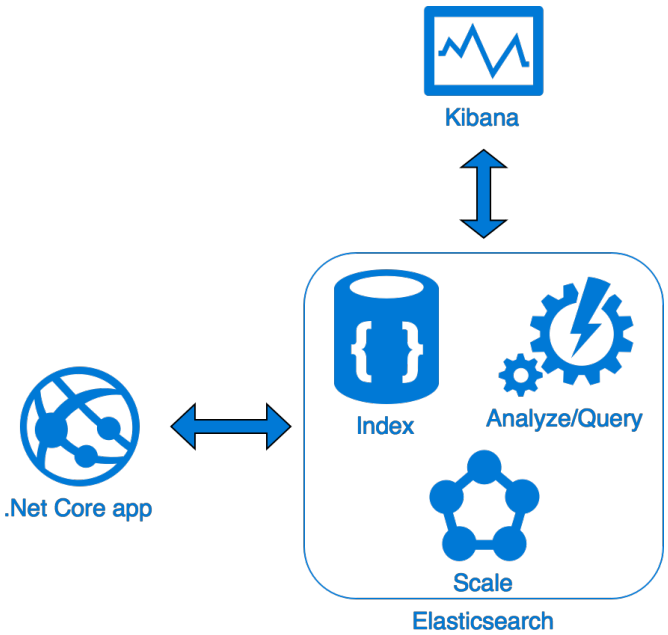


Figure 1, Elasticsearch and .Net

We will start exploring Elasticsearch through its REST API, by indexing and querying some data. Then we will perform a similar exercise using the official Elasticsearch .Net API. Once familiarized with Elasticsearch and its APIs, we will create a logger which can be plugged within .Net Core and which sends data to Elasticsearch. Kibana will be used along the way to visualize the data indexed by Elasticsearch in interesting ways.

I hope you will find the article interesting enough to leave you wanting to read and learn more about the powerful stack that Elastic provides. I certainly think so!

This article assumes a basic knowledge of C# and of REST APIs. It will use tools like Visual Studio, Postman and Docker but you could easily follow along with alternatives like VS Code and Fiddler.

ELASTICSEARCH - BRIEF INTRODUCTION

Elasticsearch at its core is a document store with powerful indexing and search capabilities exposed through a nice REST API. It is written in Java and based internally on Apache Lucene, although these details are hidden beneath its API.

Any document stored (or indexed) can get its fields indexed - fields which automatically can be searched for and aggregated in many different ways.

But ElasticSearch doesn't stop at just providing a powerful search of these indexed documents.

It is fast, distributed and horizontally scalable, supporting real time document store and analytics with clusters supporting hundreds of servers and petabytes of indexed data. It also sits at the core of the Elastic stack (aka ELK), which provides powerful applications like LogStash, Kibana and more.

Kibana specifically provides a very powerful querying and visualization web application on top of Elasticsearch. Using Kibana, it is very easy to create queries, graphs and dashboards for your data indexed in Elasticsearch.

Elasticsearch exposes a REST API and you will find many of the documentation examples as HTTP calls, which you could try using tools like curl or postman. Of course, clients for this API have been written in many different languages including .Net, Java, Python, Ruby and JavaScript amongst others.

If you want to read more, the official elastic website is probably the best place to start.

Docker, the easiest way of getting up and running locally

During this article, we will need to connect to an instance of Elasticsearch (and later Kibana). If you already have one running locally or have access to a server that you can use, that's great. Otherwise you will need to get one.

You have the option of downloading and installing Elasticsearch and Kibana either in your local machine or a VM/server you can use. However, I would suggest using Docker as the simplest and cleanest way for you to explore and play with Elasticsearch and Kibana.

You can simply run the following command and get the container up and running which contains both Elasticsearch and Kibana.

```
docker run -it --rm -p 9200:9200 -p 5601:5601 --name esk nshou/elasticsearch-kibana
```

- `-it` means starting the container in interactive mode with a terminal attached.
- `--rm` means the container will be removed as soon as you exit from the terminal.
- `-p` maps a port inside the container with a port in the host
- `--name` gives the container a name in case you don't use --rm and prefer to manually stop/remove
- `nshou/elasticsearch-kibana` is the name of an image in Docker Hub that someone already prepared with Elasticsearch and Kibana inside
- If you prefer so, you can start it in the background using the argument -d instead, and manually stopping/removing it using `docker stop esk` and `docker rm esk`.

Running multiple applications in the same container similar to what we are doing is great for trying them locally and for the purposes of this article, but isn't the recommended approach for production containers!

You should also be aware that your data would be gone once you remove the container (as soon as you stop it if you used the --rm option). While this is fine for experimenting, on real environments, you don't want to lose your data, so you will follow patterns like the "data container" instead.

Docker is a great tool and I would encourage you to learn more about it if you want to use it for something more serious than just following the article and quickly trying Elasticsearch locally. I have a nice introduction to docker for .Net Core in my previous article Building DockNetFiddle using Docker and .NET Core (bit.ly/docknetfiddle).

Simply open `http://localhost:9200` and `http://localhost:5601` and you will see both Elasticsearch and Kibana ready to use. (If you use docker toolbox, replace localhost with the ip of the VM hosting docker which you can get running `docker-machine env default` on the command line).

Figure 2, Elasticsearch up and running in Docker

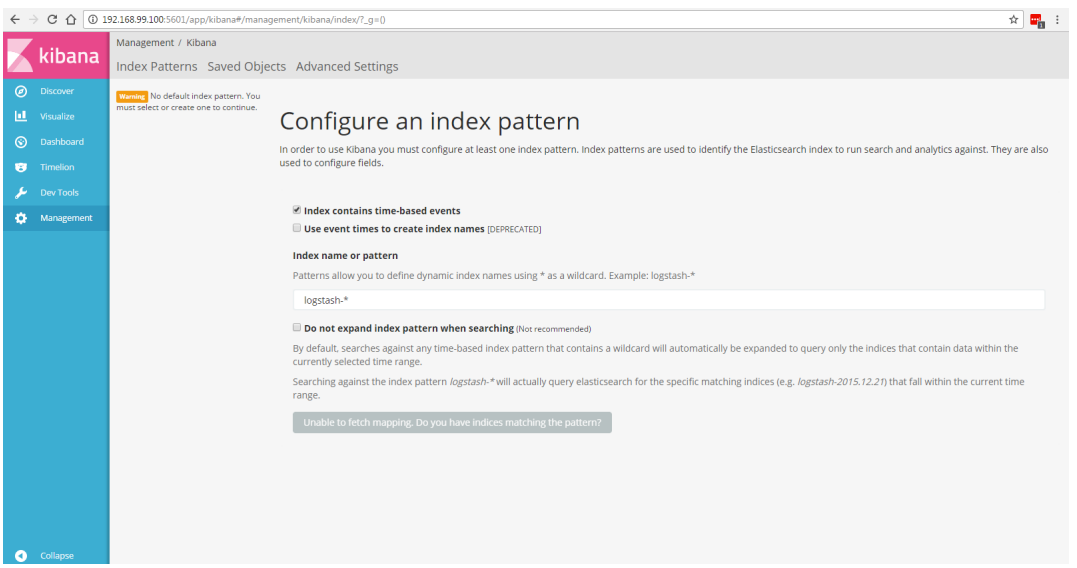


Figure 3, Kibana also ready to go

Indexing and querying documents in Elasticsearch

Before we start writing any .Net code, let's try out some basic features of our new and shiny environment. The objective will be to index some documents (akin to storing data) which will be analysed by Elasticsearch, allowing us to run different queries over them.

- Here I am going to use Postman to send HTTP requests to our Elasticsearch instance, but you could use any other similar tool like Fiddler or curl.

The first thing we are going to do is to ask Elasticsearch to create a new index and index a couple of documents. This is similar to storing data in a table/collection, the main difference (and purpose!) being that the Elasticsearch cluster (a single node in a simple docker setup) will analyze the document data and make it searchable.

Indexed documents in Elasticsearch are organized in indexes and types. In the past, this has been compared to DBs and tables, which has been confusing. As the documentation states, indexes and types are closely related to the way the data is distributed across *shards,* and indexed by Lucene. In short there is a penalty for using and querying multiple indexes, so types can be used to organize data within a single index.

Send these two requests in order to create an index and insert a document in that index (remember, if you use docker toolbox then use the ip of the VM hosting docker instead of localhost):

- Create a new index named "default". Indexes

```
PUT localhost:9200/default
```

- Index a document in the "default" index. Notice we need to tell which type of document are we storing (*product*) and the id of that document (*1*, although you could use any value as long as it is unique)

```
PUT localhost:9200/default/product/1
{
  "name": "Apple MacBook Pro",
  "description": "Latest MacBook Pro 13",
  "tags": ["laptops", "mac"]
}
```

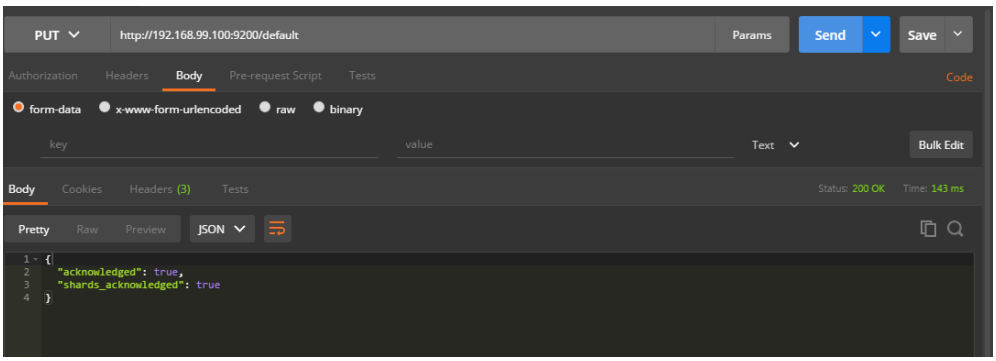


Figure 4, creating a new index

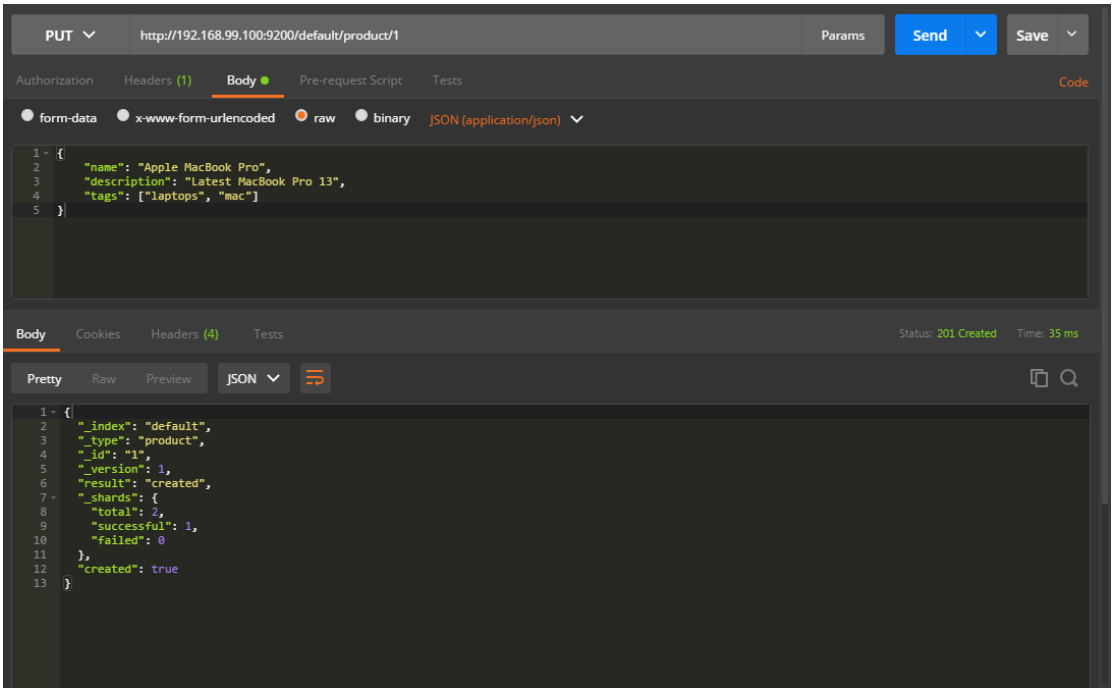


Figure 5, indexing a new document

Before we move on and verify we can retrieve and query our data, index a few more products. Try to use different tags like desktops and laptops and remember to use different ids!

Once you are done, let's retrieve all the indexed documents ordered by their names. You can either use the query string syntax or a GET/POST with body, the following two requests being equivalent:

```
GET http://localhost:9200/default/_search?q=*&sort=name.keyword:asc

POST http://localhost:9200/default/_search
{
  “query”: { “match_all”: {} },
  “sort”: [
    { “name.keyword”: “asc” }
  ]
}
```

Let's try something a bit more interesting like retrieving all the documents which contain the word “latest” in their description and the word “laptops” in the list of tags:

```
POST http://localhost:9200/default/_search
{
  “query”: {
      “bool”: {
      “must”: [
        { “match”: {“description”: “latest”} },
        { “match”: { “tags”: “laptops” } }
      ]
    }
  },
  “sort”: [
    { “name.keyword”: “asc” }
  ]
}
```

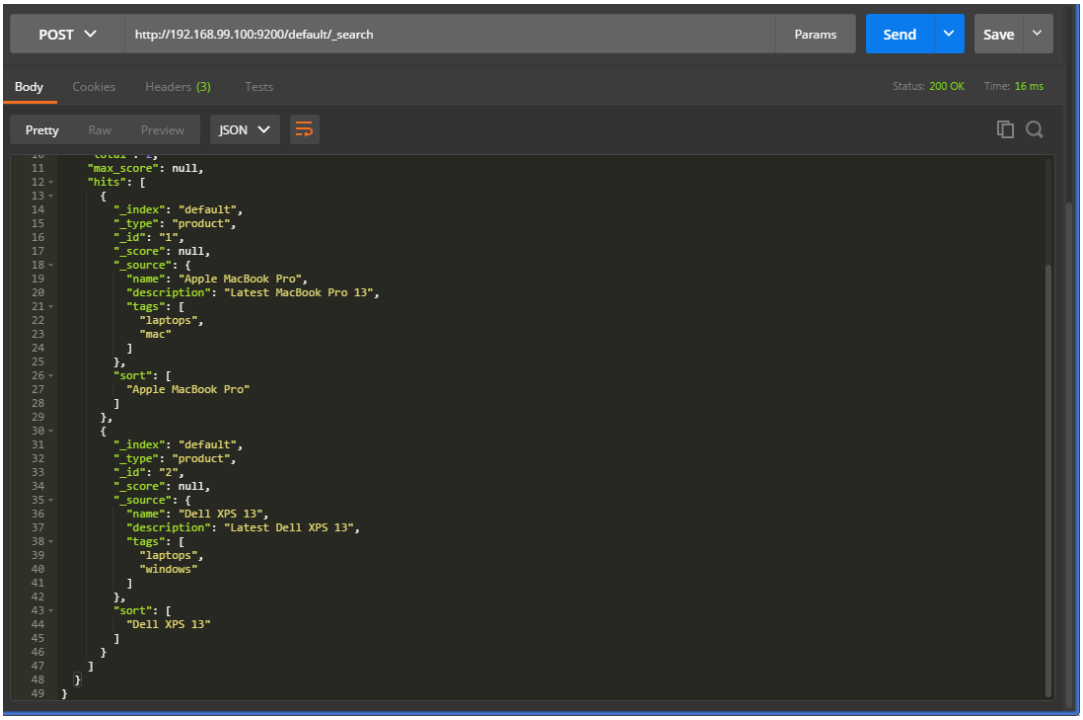


Figure 6, querying indexed documents

Visualizing data in Kibana

We are going to take a quick look at Kibana and scratch its surface in this final part of the introduction.

Assuming you have inserted a few documents while following the previous session, open your docker instance of Kibana in http://localhost:5601. You will notice that Kibana asks you to provide the default index pattern, so it knows which Elasticsearch indexes it should use:

- Since we have created a single index named "default" in the previous session, you can use "default*" as the index pattern.

- You will also need to unselect the option "Index contains time-based events" since our documents do not contain any time field.

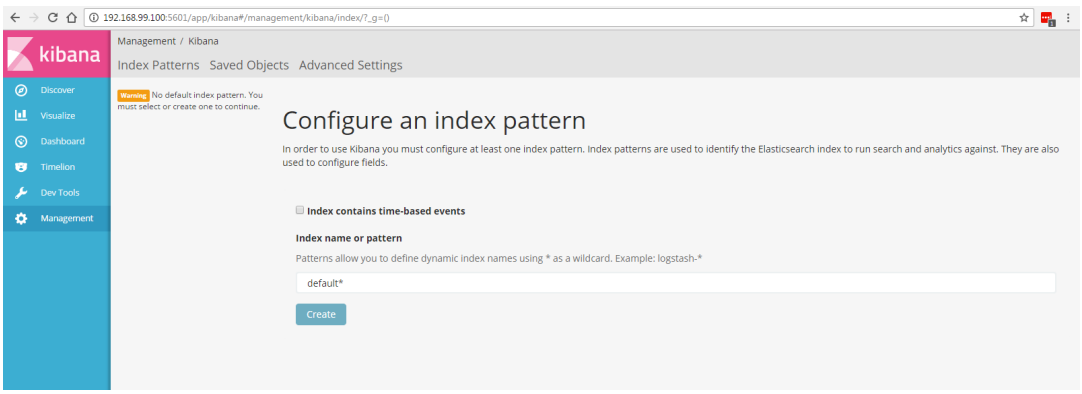


Figure 7, adding the index pattern in Kibana

Once you have done that, open the "Discover" page using the left side menu and you should see all the

latest documents inserted in the previous section. Try selecting different fields, entering search term in the search bar or individually applying filters:

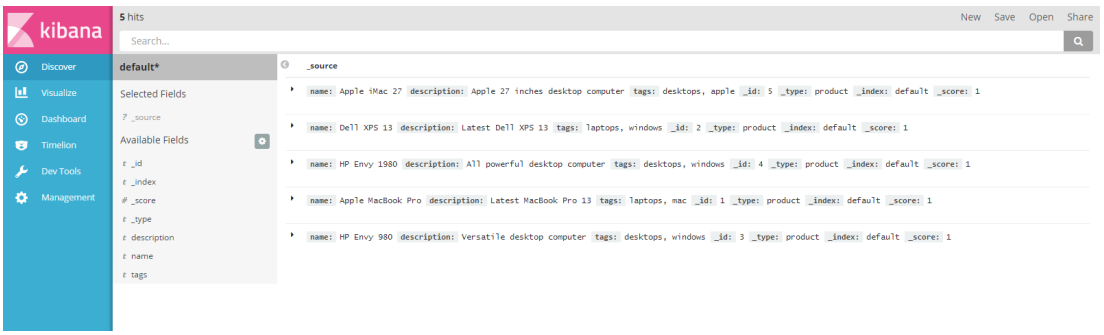


Figure 8, visualizing documents in Kibana

Finally, let's create a pie chart showing the percentage of products which are laptops or desktops. Go to the Visualize page using the left side menu and create a new "Pie Chart" visualization using the index pattern created before.

You will land on a page where you can configure the pie chart. Leave "Count" as the slice size and select "split slices" in the buckets section. Select "filters" as the aggregation type and enter 2 filters for `tags="laptops"` and `tags="desktops"`. Click run and you will see something similar to the following screenshot:

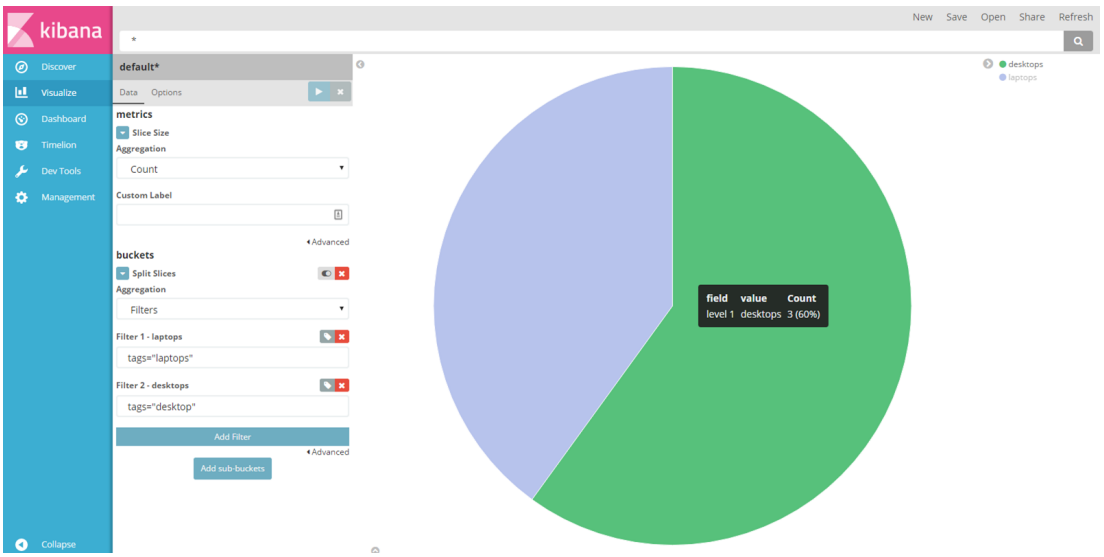


Figure 9, creating a pie chart in Kibana

Make sure you try to enter a search term in the search bar and notice how your visualization changes and includes just the filtered items!

ELASTICSEARCH .NET API

After a brief introduction to Elasticsearch and Kibana, let's see how we can index and query our documents from a .Net application.

You might be wondering why would you want to do this instead of directly consuming the HTTP API. I can provide a few reasons and I am sure you will be able to find a few on your own:

- You don't want to directly expose your Elasticsearch cluster in the open.
- Elasticsearch might not be your main database and you might need to merge or hydrate the results from the main source.
- You want to include data stored/generated server side within the documents indexed.

The first thing you will notice when you open the documentation is that there are two official APIs for .Net: **Elasticsearch.Net** and **NEST**, both supporting .Net Core projects.

- Elasticsearch.Net provides a low-level API for connecting with Elasticsearch and leaves to you the work of building/processing the requests and responses. It is a very thin client for consuming the HTTP API from .Net
- NEST sits on top of Elasticsearch.Net and provides a higher-level API. It can map your objects to/from Request/Responses, make assumptions about index names, document types, field types and provide a strongly typed language for building your queries that matches one of the HTTP REST API.

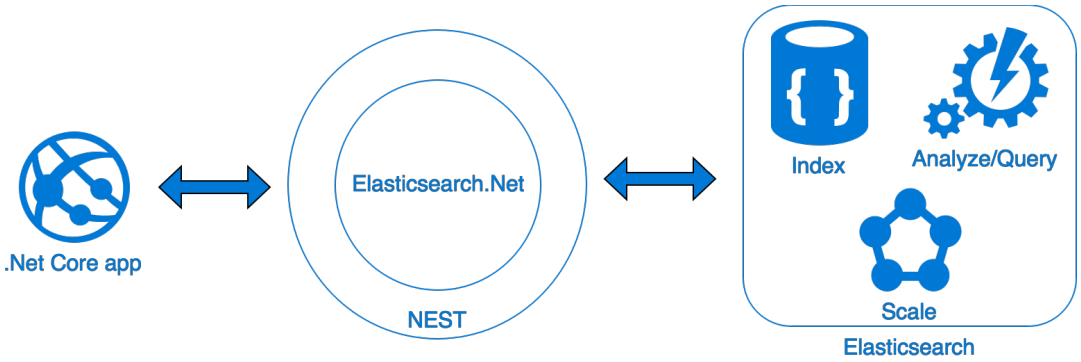


Figure 10, Elasticsearch .Net APIs

Since I am going to use NEST, the first step would be to create a new ASP .Net Core application and install NEST using the Package Manager.

Start indexing data with Nest

We are going to replicate some of the steps we took while manually sending HTTP requests in the new ASP.Net Core application. If you want, restart the docker container to clean the data or manually delete documents/indexes using the HTTP API and Postman.

Let's start by creating a POCO model for the products:

```
public class Product
{
    public Guid Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public string[] Tags { get; set; }
}
```

Next let's create a new controller `ProductController` with a method to add a new product and a method to find products based on a single search term:

```
[Route("api/[controller]")]
```

```
public class ProductController : Controller
{
  [HttpPost]
  public async Task<IActionResult> Create([FromBody]Product product)
  {

  }
  [HttpGet("find")]
  public async Task<IActionResult> Find(string term)
  {
  }
}
```

In order to implement any of these methods, we are going to need a connection to Elasticsearch. This is done instantiating an `ElasticClient` with the right connection settings. Since this class is thread-safe, the recommended approach is to use it as a singleton in your application instead of creating a new connection per request.

For the purposes of brevity and clarity I am going to now use a private static variable with hardcoded settings. Feel free to use dependency injection and configuration/options frameworks in .Net Core or check the companion code in Github.

As you can imagine, at the very minimum, you will need to provide the URL to your Elasticsearch cluster in the connection settings. Of course, there are additional optional parameters for authenticating with your cluster, setting timeouts, connection pooling and more.

```
private static readonly ConnectionSettings connSettings =
  new ConnectionSettings(new Uri("http://localhost:9200/"));
private static readonly ElasticClient elasticClient =
  new ElasticClient(connSettings);
```

Once a connection is established, indexing documents is as simple as using the Index/IndexAsync methods of ElasticClient:

```
[HttpPost]
public async Task<IActionResult> Create([FromBody]Product product)
{
  product.Id = Guid.NewGuid();
  var res = await elasticClient.IndexAsync(product);
  if (!res.IsValid)
  {
    throw new InvalidOperationException(res.DebugInformation);
  }
  return Ok();
}
```

Simple, isn't? Unfortunately, if you send the following request with Postman you will see the code fails.

```
POST http://localhost:65113/api/product
{
  "name": "Dell XPS 13",
  "description": "Latest Dell XPS 13",
  "tags": ["laptops", "windows"]
}
```

This is happening because NEST isn't able to determine which index should be used when indexing the

document! If you remember when manually using the HTTP API, the URL specified the index, document type and id of the document as: *localhost:9200/default/product/1*.

NEST is able to infer the type of the document (using the class name) and can also default how fields will be indexed (based on the field types), but needs a bit of help with the index names. You can specify a default index to be used when one cannot be determined, and specific index name for specific types.

```
connSettings = new ConnectionSettings(new Uri("http://192.168.99.100:9200/"))
  .DefaultIndex("default")
  //Optionally override the default index for specific types
  .MapDefaultTypeIndices(m => m
    .Add(typeof(Product), "default"));
```

Try again after making these changes. You will see how NEST is now able to create the index if it wasn't already there, and the document is indexed. If you switch to Kibana, you should also be able to see the document. Notice how NEST:

- Inferred the document type from the class name, Product.
- Inferred the Id as the Id property of the class.
- Included every public property in the document sent to Elasticsearch.

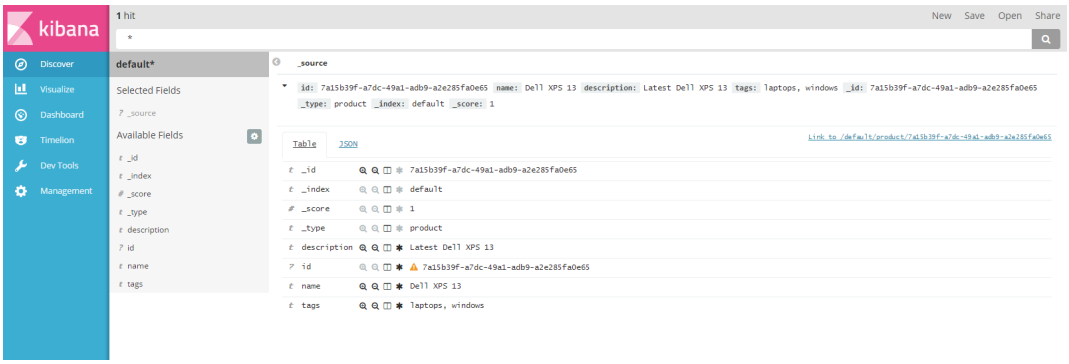


Figure 11, document indexed using NEST

Before we move into querying data, I want to revisit the way indexes are created.

How are Indexes Created

Right now, we rely on the fact that the index will be created for us if it doesn't exist. However, the way fields are mapped in the index is important and directly defines how Elasticsearch will index and analyse these fields. This is particularly obvious with string fields since Elasticsearch v5 provides two different types *"Text"* and *"Keyword"*:

- *Text* fields will be analyzed and split into words so they can tap into the more advanced search features of Elasticsearch

- On the other hand, *Keyword* fields will be taken "as is" without being analyzed and can only be searched by their exact values.

You can annotate your POCO models with attributes that NEST can use to generate more specific index mappings:

```
public class Product
{
    public Guid Id { get; set; }
    [Text(Name=”name”)]
    public string Name { get; set; }
    [Text(Name = “description”)]
    public string Description { get; set; }
    [Keyword(Name = “tag”)]
    public string[] Tags { get; set; }
}
```

However now that we want to provide index mappings, we have to manually create and define the mappings ourselves using the ElasticClient API. It is very straightforward, especially if we are just using the attributes:

```
if (!elasticClient.IndexExists(“default”).Exists)
{
    elasticClient.CreateIndex(“default”, i => i
        .Mappings(m => m
            .Map<Product>(ms => ms.AutoMap())));
}
```

Send a request directly to Elasticsearch (`GET localhost:9200/default`) and notice how the mapping settings for name and description fields are different from the one for the tags field.

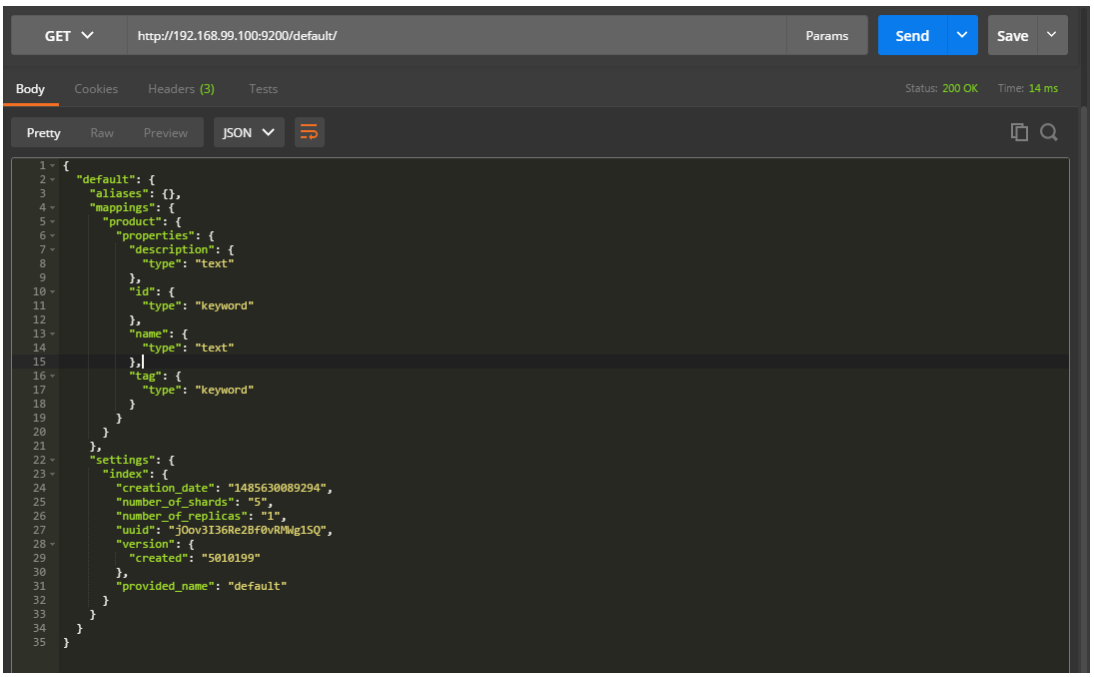


Figure 12, index mapping created with NEST

Querying data with Nest

Right now, we have a `ProductController` that can index products in Elasticsearch using NEST. It is time we implement the `Find` action of the controller and use NEST to query the documents indexed in Elasticsearch.

We are going to implement a simple search with a single term. It will look at every field and you will notice how:

- Fields mapped as "Text" were analysed. You will be able to search for specific individual words inside the name/description fields

- Fields mapped as "Keywords" were taken as-is and not analysed. You will only be able to search for exact matches of the tags.

NEST provides a rich API for querying Elasticsearch that maps with the standard HTTP API. Implementing the type of query described above is as simple as using the method Search/SearchAsync and building a SimpleQueryString as argument.

```
[HttpGet("find")]
public async Task<IActionResult> Find(string term)
{
  var res = await elasticClient.SearchAsync<Product>(x => x
    .Query( q => q.
      SimpleQueryString(qs => qs.Query(term))));
  if (!res.IsValid)
  {
    throw new InvalidOperationException(res.DebugInformation);
  }
  return Json(res.Documents);
}
```

Test your new action using Postman:

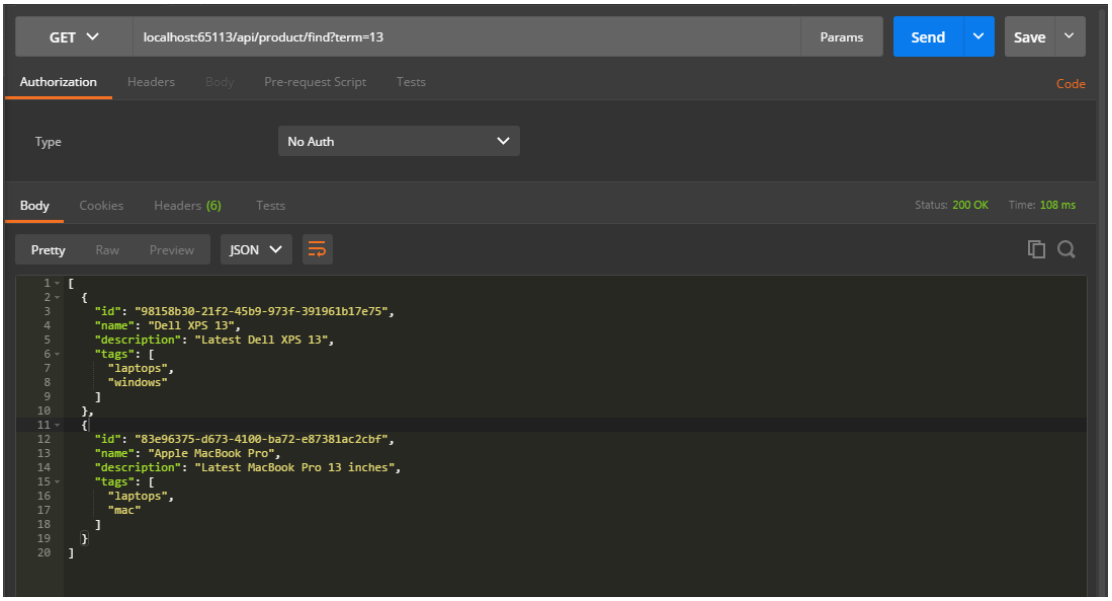


Figure 13, testing the new action using NEST for querying Elasticsearch

As you might have already realized, our action behaves exactly the same as when manually sending the following request to Elasticsearch:

```
GET http://localhost:9200/default/_search?q=*&
```

CREATING AN ELASTICSEARCH .NET CORE LOGGER PROVIDER

Now that we have seen some of the basics of NEST, let's try something a bit more ambitious. Since we have created an ASP.Net Core application, we can take advantage of the new logging framework and implement

our own logger that sends information to Elasticsearch.

The new logging API differentiates between the logger and the logger provider.

- A logger is used by client code, like a controller class, to log information and events.

- Multiple logger providers can be added and enabled for the application. These will register the logged information/events and can be configured with independent logging levels from Trace to Critical.

The framework comes with a number of built-in providers for the console, event log, Azure and more, but as you will see, creating your own isn't complicated. For more information check the Logging article of the official documents.

In the final sections of the article, we will create a new logger provider for Elasticsearch, enable it in our application and use Kibana to view the logged events.

Add a new logger provider for Elasticsearch

The first thing to do is define a new POCO object that we will use as the document to index using NEST, similar to the Product class we created earlier.

This will contain the logged information, optional information about any exception that might have happened and relevant request data. Logging the request data will come in handy so we can query/visualize our logged events in relation to specific requests.

```
public class LogEntry
{
  public DateTime DateTime { get; set; }
  public EventId EventId { get; set; }
  [Keyword]
  [JsonConverter(typeof(StringEnumConverter))]
  public Microsoft.Extensions.Logging.LogLevel Level { get; set; }
  [Keyword]
  public string Category { get; set; }
  public string Message { get; set; }

  [Keyword]
  public string TraceIdentifier { get; set; }
  [Keyword]
  public string UserName { get; set; }
  [Keyword]
  public string ContentType { get; set; }
  [Keyword]
  public string Host { get; set; }
  [Keyword]
  public string Method { get; set; }
  [Keyword]
  public string Protocol { get; set; }
  [Keyword]
  public string Scheme { get; set; }
  public string Path { get; set; }
  public string PathBase { get; set; }
  public string QueryString { get; set; }
  public long? ContentLength { get; set; }
```

```
  public bool IsHttps { get; set; }
  public IRequestCookieCollection Cookies { get; set; }
  public IHeaderDictionary Headers { get; set; }
  [Keyword]
  public string ExceptionType { get; set; }
  public string ExceptionMessage { get; set; }
  public string Exception { get; set; }
  public bool HasException { get { return Exception != null; } }
  public string StackTrace { get; set; }
}
```

The next step is implementing the `ILogger` interface on a new class. As you can imagine, this will receive the logged data, map it to a new `LogEntry` object and use the `ElasticClient` to index the document in Elasticsearch.

- We will use an `IHttpContextAccessor` so we can get the current HttpContext and extract the relevant request properties.

I am not going to copy the code again to connect with Elasticsearch and create the index, as there is nothing new compared to what we did earlier. Either use a different index or delete the index with the products indexed in the previous section.

Note: *You can check the companion code in* Github *for an approach using dependency injection and configuration files.*

The main method to implement is `Log<TState>` which is where we create a LogEntry and index it with NEST:

```
public void Log<TState>(LogLevel logLevel, EventId eventId, TState state, Exception
exception, Func<TState, Exception, string> formatter)
{
  if (!IsEnabled(logLevel)) return;

  var message = formatter(state, exception);
  var entry = new LogEntry
  {
    EventId = eventId,
    DateTime = DateTime.UtcNow,
    Category = _categoryName,
    Message = message,
    Level = logLevel
  };

  var context = _httpContextAccessor.HttpContext;
  if (context != null)
  {
    entry.TraceIdentifier = context.TraceIdentifier;
    entry.UserName = context.User.Identity.Name;
    var request = context.Request;
    entry.ContentLength = request.ContentLength;
    entry.ContentType = request.ContentType;
    entry.Host = request.Host.Value;
    entry.IsHttps = request.IsHttps;
    entry.Method = request.Method;
    entry.Path = request.Path;
    entry.PathBase = request.PathBase;
```

```
    entry.Protocol = request.Protocol;
    entry.QueryString = request.QueryString.Value;
    entry.Scheme = request.Scheme;
    entry.Cookies = request.Cookies;
    entry.Headers = request.Headers;
  }

  if (exception != null)
  {
    entry.Exception = exception.ToString();
    entry.ExceptionMessage = exception.Message;
    entry.ExceptionType = exception.GetType().Name;
    entry.StackTrace = exception.StackTrace;
  }

  elasticClient.Client.Index(entry);
}
```

You also need to implement the additional `BeginScope` and `IsEnabled` methods.

- Ignore BeginScope for the purposes of this article, just return null.
- Update your constructor so it receives a LogLevel, then implement IsEnabled returning true if the level being compared is greater than or equal to the one received in the constructor.

```
public bool IsEnabled(LogLevel logLevel)
{
  return logLevel >= _logLevel;
}
```

With `ILogger` implemented, create the new `ILoggerProvider`. This class is responsible for creating instances of `ILogger` for a specific category and level. It will also receive the configured settings which maps each category to its logging level.

What's the category you might ask? This is a string that identifies which part of your system logged the event. By default, every time you inject an instance of `ILogger<T>`, the category is assigned by default as the type name of `T`. For example, acquiring an `ILogger<MyController>` and using it to log some events means those events will have "*MyController*" as the category name.

This can come in handy for e.g. to set different verbose levels for individual categories to filter/query the logged events and many more usages that I am sure you can think of.

The implementation of this class would look like the following:

```
public class ESLoggerProvider: ILoggerProvider
{
  private readonly IHttpContextAccessor _httpContextAccessor;
  private readonly FilterLoggerSettings _filter;

  public ESLoggerProvider(IServiceProvider serviceProvider, FilterLoggerSettings filter = null)
  {
    _httpContextAccessor = serviceProvider.GetService<IHttpContextAccessor>();
    _filter = filter ?? new FilterLoggerSettings
    {
      {"*", LogLevel.Warning}
    };
```

```
    }

    public ILogger CreateLogger(string categoryName)
    {
        return new ESLogger(_httpContextAccessor, categoryName, FindLevel(categoryName));
    }

    private LogLevel FindLevel(string categoryName)
    {
        var def = LogLevel.Warning;
        foreach (var s in _filter.Switches)
        {
            if (categoryName.Contains(s.Key))
                return s.Value;

            if (s.Key == "*")
                def = s.Value;
        }

        return def;
    }

    public void Dispose()
    {
    }
}
```

Finally let's create an extension method that can be used to register our logger provider in the `Startup` class:

```
public static class LoggerExtensions
{
  public static ILoggerFactory AddESLogger(this ILoggerFactory factory,
  IServiceProvider serviceProvider, FilterLoggerSettings filter = null)
  {
    factory.AddProvider(new ESLoggerProvider(serviceProvider, filter));
    return factory;
  }
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
{
  loggerFactory.AddConsole(Configuration.GetSection("Logging"))
  .AddDebug()
  .AddESLogger(app.ApplicationServices, new FilterLoggerSettings
  {
    {"*", LogLevel.Information}
  });
  …
}
```

Notice how I am overriding the default settings to assign a logging level of Information to every category. This is done so that we can easily index some events on Elasticsearch for every request.

Visualize the data in Kibana

Now that we have logged events in Kibana, let's use it to explore the data!

First of all, recreate the index pattern in Kibana and this time make sure to select "*Index contains time-based events*", selecting the field `dateTime` as the "*Time-field name*".

Next, fire up your app and navigate through some pages to get a few events logged. Also feel free to add code for throwing exceptions at some endpoint, so we can see the exception data logged as well.

After using your application a bit, go to the **Discover** page in Kibana where you should see a number of events logged, ordered by the dateTime field (By default data is filtered to the last 15 minutes, but you can change that in the upper right corner):

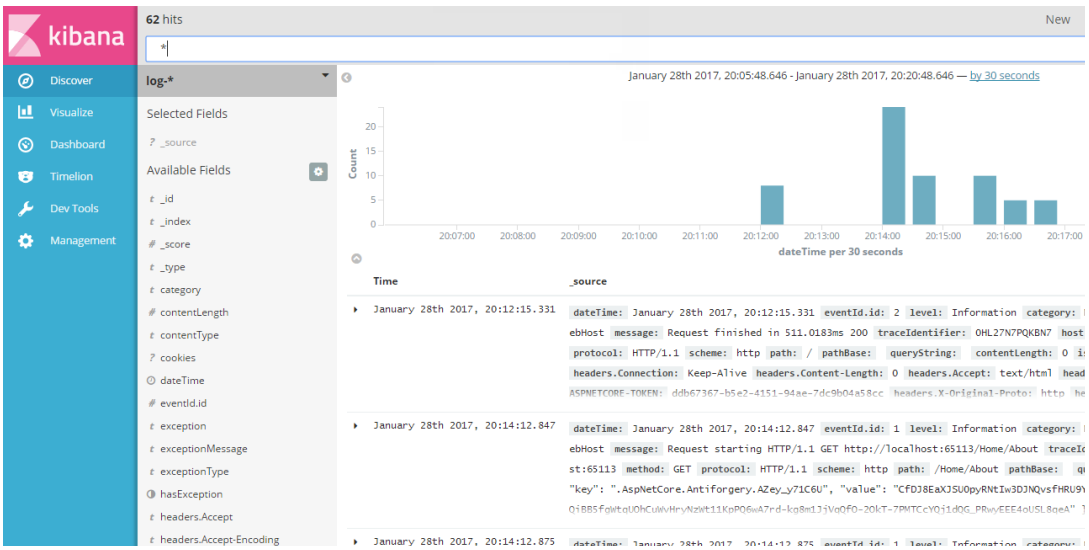


Figure 14, visualizing the logged events in Kibana. Select the time range in the upper right corner

Try entering `exception` in the search bar and notice how it filters down the events to the ones that contain the word exception in any of the analysed text fields. Then try searching for a specific exception type (remember we used a keyword field for it!).

You can also try to search for a specific URL in two different ways as in `/Home/About` and `path:"/Home/About"`. You will notice how the first case includes events where the referrer was /Home/About, while the second correctly returns only events where the path was /Home/About!

Once you have familiarized yourself with the data and how it can be queried, let's move to creating a couple of interesting graphs for our data.

First we are going to create a graph showing the number of exceptions logged per minute.

- Go to the **Visualize** page in Kibana and create a new **Vertical bar chart**.
- Leave the Count selected for the Y-axis and add a date histogram on the X-axis.
- Set the interval as minutes and finally add a filter hasException:true in the search box.

You will get a nice graph showing the number of exceptions logged per minute:

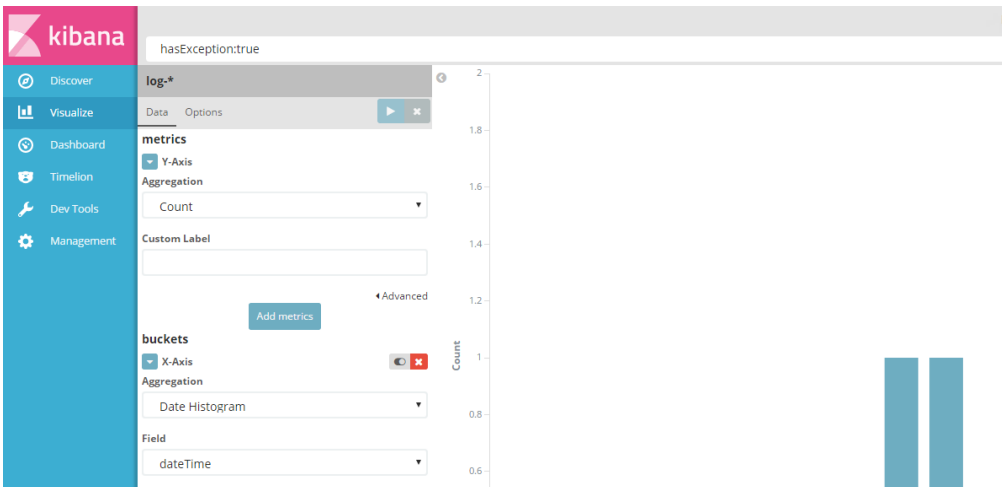


Figure 15, exceptions logged per minute

Next, show the number of messages logged for each category over time, limited to the top 5 chattier categories:

- Go to the **Visualize** page in Kibana and create a new **Line chart**.
- Again leave the Count selected for the Y-axis and add a date histogram on the X-axis, selecting dateTime as the field and minutes as the interval.
- Now add a sub-bucket and select "split lines". Use "significant terms" as the sub aggregation, category as the field and size of 5.

This will plot something similar to the following:

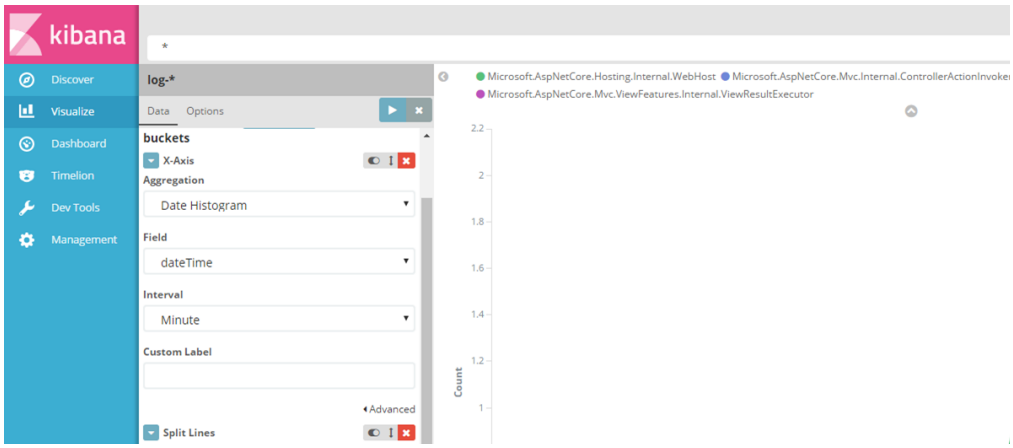


Figure 16, chattier categories over time

Try adding some filters on the search box and see how it impacts the results.

Finally, let's add another graph where we will see the top five most repeated messages for each of the top five categories with more messages.

- Go to the **Visualize** page in Kibana and create a new **Pie chart**.
- As usual, leave the Count selected for the Y-axis
- Now split the chart in columns by selecting "Terms" as the aggregation, "category" as the field, count as the metric and 5 as the limit.
- Then split the slices by selecting "Terms" as the aggregation, "message.keyword" as the field, count as the metric and 5 as the limit.

Once you have these settings in place, you will see a chart similar to mine:

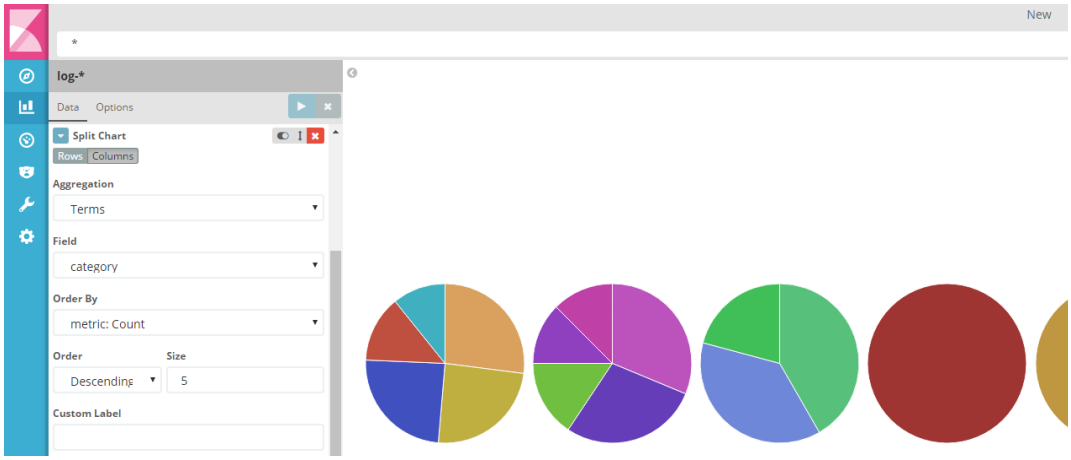


Figure 17, most frequent messages per category

Take your time and inspect the data (the percentage and actual message/category is shown hovering on the chart elements). For example you will realize that the exceptions are logged by the `DeveloperExceptionPageMiddleware` class.

Conclusion

Elasticsearch is a powerful platform for indexing and querying data. While it is quite impressive on its own, combining it with other applications like Kibana makes it a pleasure to work with in areas like data analysis, reporting and visualization. You can achieve non-trivial results almost from day one just by scratching the surface of what this platform provides.

When it comes to .Net and .Net Core, the official Elasticsearch APIs have you covered as they support .Net Standard 1.3 and greater (they are still working on providing support for 1.1).

As we have seen, using this API in an ASP.Net Core project has been straightforward and we could easily use it as the storage of a REST API and as a new logger provider added to the application.

Last but not least, I hope you enjoyed using docker as you followed along. The freedom for trying applications like Elasticsearch is a revelation, but at the same time, is just a tiny fraction of what docker can do for you and your team ■

Download the entire source code from GitHub at
bit.ly/dncm29-elklogging

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 Francesco Bianchi and Suprotim Agarwal for reviewing this article.

A MAGAZINE FOR .NET AND JAVASCRIPT DEVS

- ☑ AGILE
- ☑ ASP.NET
- ☑ MVC, WEB API
- ☑ ANGULAR.JS
- ☑ NODE.JS
- ☑ AZURE
- ☑ VISUAL STUDIO
- ☑ .NET
- ☑ C#, WPF

We've got it all!

100K PLUS READERS

230 PLUS AWESOME ARTICLES

27 EDITIONS

FREE SUBSCRIPTION USING YOUR EMAIL

EVERY ISSUE DELIVERED

RIGHT TO YOUR INBOX

NO SPAM POLICY

SUBSCRIBE TODAY!

Gil Fink

GETTING TO KNOW THE REDUX PATTERN

..With great power comes great complexities

Introduction

The trend towards Single Page Applications (SPAs) has been increasing across responsive websites. On the whole, a SPA is just a web application that uses one HTML web page as an application shell and whose end-user interactions are implemented with JavaScript, HTML, and CSS. These days, it's common to build SPAs using frameworks/libraries such as Angular or React.

With great power comes great complexity and while SPAs can help you build fast and fluid User Interfaces (UI), they can also introduce new problems that we haven't dealt in with old style web applications. Handling the data flow in SPA can be very hard and managing application states can be even harder. If you don't handle data flow and states correctly, you can expect your application behavior to be unpredictable, inconsistent and untestable.

There are many ways to tackle data flow and application state complexity. For example, you can apply the **Command Query Responsibility Segregation (CQRS) pattern**, which isolates queries that read data from commands updating that data. Another option is the **Event Sourcing pattern**, which ensures that all changes in state are stored as a sequence of events.

> IN THIS ARTICLE, WE WILL EXPLORE THE REDUX PATTERN AND HOW IT CAN HELP TACKLE THESE SPA COMPLEXITIES.

Enter the Flux and Redux Patterns

In order to understand the Redux pattern, we should start with the Flux pattern. The Flux pattern was introduced by Facebook a few years ago. Flux is a unidirectional data flow pattern that fits into the components architecture and blends with frameworks and libraries such as React and Angular 2.
Flux includes 4 main players: actions, dispatcher, stores and views. Figure 1 describes this pattern:

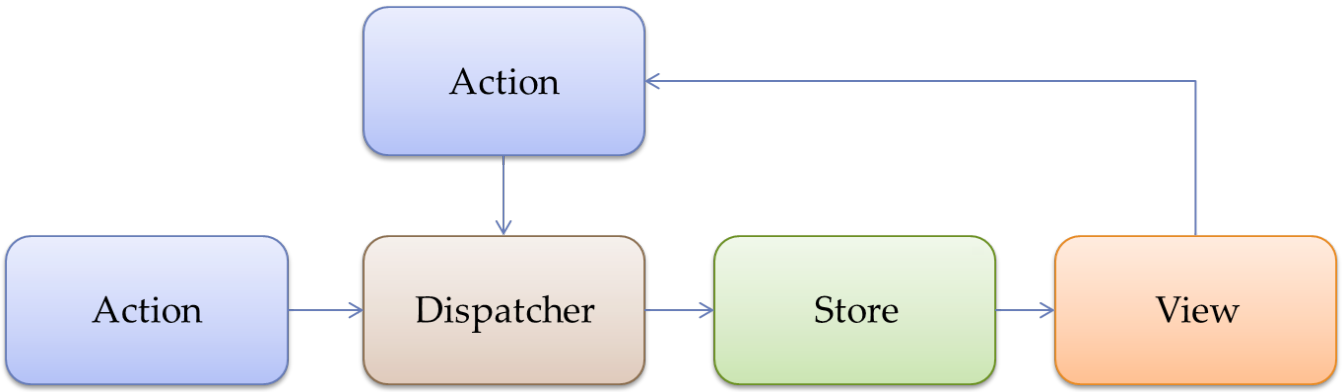


Figure 1: Flux Pattern

When a user interacts with a view, the view propagates an action to a central dispatcher. The dispatcher is responsible to propagate actions to one or many store objects.

Store objects hold the application data and business logic and are responsible to register to actions in the dispatcher. They also update the view which is affected by a specific action that indicates data/state has changed. The responsibility of the view is to update according to the new data/state, and to interact with users. Another option to enter the data flow is an outside action which is not related to a view, such as timer callbacks or Ajax callbacks.

Now that we understand how the Flux pattern works, let's drill down into Redux principles before we explore the Redux data flow and how it relates to Flux.

Redux Pattern Principles

Redux is based upon 3 major principles:
1. Single source of truth
2. State is read-only
3. Changes are made with pure functions

Single source of truth

In Redux, there is only one store object. That store is responsible to hold all the application state in one object tree.

Having only one store object helps to simplify the debugging and profiling of the application because all the data is stored in one place. Also, difficult functionality such as redo/undo becomes simpler because the state is located in one place only.

The following example demonstrates how you get the state from a store object using the `getState`

function:

```
let state = store.getState();
```

State is read-only

The only way to mutate the state that is held by the store, is to emit an action that describes what happened. State can't be manipulated by any object and that guards us from coupling problems and from some other side effects. Actions are just plain objects that describe the type of an action, and the data to change. For example, the following code block shows how to dispatch two actions:

```
store.dispatch({
   type: 'ADD_GROCERY_ITEM',
   item: { productName: 'Milk' }
});
store.dispatch({
   type: 'REMOVE_GROCERY_ITEM',
   index: 3
});
```

In the example, we dispatch an *add grocery item* action, and a *remove grocery item* action. The data in the action objects is the item to add or the index of the object to remove.

Changes are made with pure functions

In order to express how state transition occurs, you will use a reducer function. All reducer functions are pure functions. A pure function is a function that receives input and produces output without changing the inputs. In Redux, a reducer will receive the previous state and an action, and will produce a new state without changing the previous state. For example, in the next code block you can see a `groceryItemsReducer` which is responsible to react to add grocery item action:

```
function groceryItemsReducer(state, action) {
   switch (action.type) {
     case 'ADD_GROCERY_ITEM':
        return object.assign({}, state, {
           groceryItems: [
              action.item,
              …state.groceryItems
           ]
        };
     default:
        return state;
   }
}
```

As you can see, if an action isn't recognized by the reducer, you return the previous state. In case you got an add grocery item action, you return a copy of the previous state that includes the new item.
Now that we understand the main Redux principles, we can move on to the Redux data flow.

Redux Data Flow

The Redux data flow is based on Flux but it's different in a lot of ways. In Redux there is no central dispatcher. Redux includes only one store object, while Flux allows the usage of multiple stores. In order

to mutate the stored Redux state, you use reducer functions instead of inner business logic functionality that exists in Flux stores. All in all, Redux is a new pattern that took some aspects of Flux and implemented them differently.

The following figure show the Redux data flow:

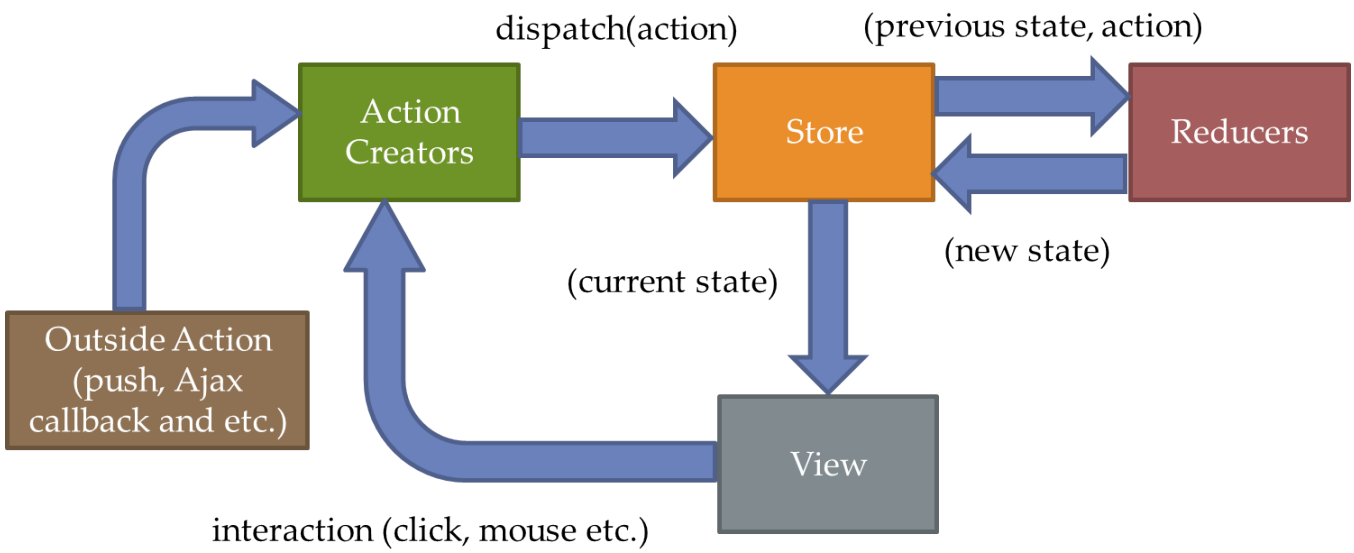


Figure 2: Redux Data Flow

As you can see, the main players really resemble the players in Flux, except for the reducers and the lack of dispatcher object.

In the Redux data flow, a user interaction or an asynchronous callback will produce an action object. The action object will be created by a relevant action creator and be dispatched to the store. When the store receives an action, it will use a reducer to produce a new state. Then, the new state will be delivered to views and they will update accordingly. This data flow is much simpler then Flux and helps to produce a predictable state to the application.

The Redux Library

The Redux library was created by Dan Abramov. On the whole, it is a small and compact library that includes a small set of API functions. In order to get started with the library, you can install it using npm:

```
npm install --save redux
```

or download it from its repository on GitHub: https://github.com/reactjs/redux.

The Redux library includes the following main API functions:

- createStore(reducer) – function to create a new store with the given reducer function.

- combineReducers(reducers) – function that helps you to combine the given array of reducers into one reducer. It can be helpful to divide your reducers into different aspect and then to use this function to produce the application main reducer.

- applyMiddleware(middlewares) – function that add middlewares to the data flow pipeline. Middlewares

are functions that wrap the store dispatch function and enables you to add functionality that runs before a dispatch occurs. Middlewares can be composed together to create interesting functionality such as logging or even dispatching of asynchronous operations. In the article, I won't cover middlewares usage but I encourage you to seek information about this concept.

- store API – Once a store is created, it includes a few API functions such as dispatch and getState.

Now that you know about the Redux library, let's see it in action.

Building a Simple App with Redux

Note: *This example assumes that you have node and npm installed on your machine. If you don't have node and npm installed on your machine, go to the following link to download and install them:* https://nodejs.org/en/download/.

It is also assumed that you have some TypeScript knowledge.

Create a new empty project and give it the name *ReduxInAction*.

Run npm init and initialize a new package.json file. In the package.json, add the following `main` and `scripts` properties:

```
"main": "src/index.js",
"scripts": {
    "test": "node src/index.js"
}
```

Once you finished editing the package.json file, run the following command in the command line:

```
npm install –save redux
```

This command will install the Redux library in your project. Add a new tsconfig.json file and add to it the following code:

```
{
  "compilerOptions": {
    "moduleResolution": "node",
    "module": "commonjs",
    "target": "es5",
    "sourceMap": true
  },
  "exclude": [
    "node_modules"
  ]
}
```

Add a new *src* folder under the root of the project. In the src folder, add two new folders: actions and reducers.

In the actions folder, create an index.ts file and add to it the following code:

```
import {Action} from "redux";
```

```
export default class CounterActions {
  static INCREMENT: string = 'INCREMENT';
  static DECREMENT: string = 'DECREMENT';
  increment(): Action {
    return {
      type: CounterActions.INCREMENT
    };
  }
  decrement(): Action {
    return {
      type: CounterActions.DECREMENT
    };
  }
}
```

The `CounterActions` class is an action creator class which also contains the action types that are included in our application.

In the *reducers* folder, create an index.ts file and add to it the following code:

```
import CounterActions from "../actions/index";
const INITIAL_STATE = 0;
export default (state = INITIAL_STATE, action) => {
  switch (action.type) {
    case CounterActions.INCREMENT:
      return state + 1;
        case CounterActions.DECREMENT:
          return state - 1;
        default:
          return state;

   }
}
```

The reducer has the implementation of how to mutate the current state and produce a new state. Once an increment action arrives, the state is incremented by 1. Once a decrement action arrives, the state is decremented by 1.

Now you can create the application shell and run it. In the `src` folder, add index.ts file and add the following code to it:

```
import {createStore} from 'redux';
import counterReducer from './reducers';
import CounterActions from "./actions/index";
const store = createStore(counterReducer);
const counterActions = new CounterActions();
console.log(store.getState()); // output 0 to the console
store.dispatch(counterActions.increment() as any);
console.log(store.getState()); // output 1 to the console
store.dispatch(counterActions.increment() as any);
console.log(store.getState()); // output 2 to the console
store.dispatch(counterActions.decrement() as any);
console.log(store.getState()); // output 1 to the console
```

At first, create a store using the *createStore* function and then give the store the *counterReducer* created earlier. Then create a new CounterActions instance. Then print to the console a set of operations to perform on the store.

Once the application is in place, run the TypeScript compiler to compile all the files in the project:

```
tsc -p
```

Then run the command `npm run test` to test the output that is written in the console.

This is a simple example of how to use Redux and it doesn't include any UI. In a follow up article, I'll cover how to combine Redux with React library.

Summary

Redux became a very popular data flow pattern and is being used in many applications. The Redux library is now a part of the efforts made by Facebook to build open source libraries. During the last year, I had the opportunity to use the pattern in several projects and it really proved itself by helping to simplify very complicated features.

While Redux can help to simplify data flow in big SPAs, it isn't suitable to every application and in simple or small applications, it can produce a lot of unnecessary coding overhead.

I encourage you to deep dive into Redux and there is a free course that was recorded by Redux library creator Dan Abramov which delves into features that weren't covered in this article: https://egghead.io/courses/getting-started-with-redux ■

Microsoft® MVP Most Valuable Professional

Gil Fink
Author

Gil Fink is a web development expert, Microsoft MVP and sparXys CEO. He is currently consulting for various enterprises and companies, where he helps to develop web based solutions. He conducts lectures and workshops for individuals and enterprises who want to specialize in web development. He is also co-author of several Microsoft Official Courses (MOCs) and training kits, co-author of "Pro Single Page Application Development" book (Apress), co-organizer of GDG Rishon Meetup and AngularUP conference.

Thanks to Ravi Kiran for reviewing this article.


Follow us on
Twitter
@dotnetcurry

Rahul Sahasrabuddhe

JavaScript Frameworks

A Shopper's Guide for the Modern ASP.NET MVC Developer

The past few years have seen a meteoric rise in SPA and MVC applications. You cannot go more than a few sentences discussing SPA without stumbling upon a JavaScript framework. This article will serve as an introduction to new as well as established JavaScript frameworks, and will help you make better decisions when you shop for a JS framework to compliment your ASP.NET MVC apps.

User experience should not be dependent on JavaScript. However JavaScript can serve to facilitate and enhance that experience.

In today's "app" world, what really matters beyond frameworks, languages, platforms and technologies, is **customer experience**[1]. With users having shorter attention span than a goldfish, your application should be able to engage with them, within no time.

It is not about focus anymore; it is about fixation.

Once users access your app – be it web/mobile, or consumer oriented/ enterprise-class; they should "stay" onto your application irrespective of all other distractions.

The user interface or in MVC parlance – **the View** plays a key role in designing applications that provide superlative customer experiences. This is where choosing the right JavaScript Framework becomes a vital decision.

As developers, you would accept nothing less than a flexible, modular, fast and rich JavaScript framework that can produce excellent quality user experiences that are user-friendly, scalable and maintainable too.

1 Customer Experience is the sum-totality of how customers engage with your company and brand, not just in a snapshot in time, but throughout the entire arc of being a customer (Src: bit.ly/dnc-cx)

The market is abuzz with plenty of JavaScript Frameworks and libraries with a new addition to it every fortnight. Which one fits your need?

Let's find out!

JavaScript Frameworks - A quick primer

A JavaScript Framework is basically a set of functionalities provided by the creator of the framework to be used/re-used and in some cases, refactored as well. A framework is built by keeping specific design goals in mind.

Here are some examples of JavaScript Frameworks and libraries that are built around specific design objectives:

1) Angular: a framework used for building complex UI applications.

2) React.JS: focuses mainly on the "V" (View) part of the Model-View-Controller (MVC) architecture. At times React js is used in conjunction with Angular JS.

3) Backbone.js: Gives structure to web applications. Specifically built for teams with less common use-cases.

4) Ember.js: Developer-centric convention-driven framework. Embraces the MV* structure and is targeted towards developers who have a MVC programming background in any Object Oriented language

5) Polymer.js: Based around leveraging web components. Uses polyfills until web components are implemented across all browsers.

6) D3.js: mainly focuses on the visualization aspects of data and is used when you want to render charts and graphs in web applications.

You can also have a complete stack built on JavaScript now – also referred to as the MEAN stack (Except M which refers to MongoDB, rest others – (E)xpress.js, (A)ngular.js and (N)ode.js are JavaScript based).

If you are new to MEAN, check out how to use Node.js, Express and MongoDB (MEAN stack) in an ASP.NET MVC application over here bit.ly/dnc-mean.

Why JavaScript Frameworks?

JavaScript Frameworks have become a quintessential ingredient of web application development these days. This is particularly true in context of Single-Page applications (SPAs) and MVC based applications. Besides speeding up the development significantly, here are some key factors that make JavaScript Frameworks a "must-have" component in any web application development strategy:

1) JavaScript frameworks usually come bundled with a lot of features like templates, data-binding, routing etc. out-of-the-box that expedite the development process and thus reduce the time to market.

2) JavaScript frameworks have come into vogue mainly due to the MV* (MVC or MVVM) pattern being used in modern web apps consistently. They provide a well-defined structure to the code.

3) They operate on client/browser side and can provide some great functionalities like instant data validations etc. for which there are no server trips needed.

4) A typical web application (and that too if it is consumer facing) ought to have a rich and at the same time, a simplistic user interface. JavaScript framework allows you to do that in combination with CSS While CSS provides excellent UI capabilities, JavaScript provides an efficient means of manipulating those capabilities.

JavaScript frameworks are helping older web apps which use heavy server-side rendering, transition to a more fluid modern UX, where a lot of action happens on the client-side without constant page reloads.

JavaScript Libraries

A library is a collection of code to perform common tasks. A framework differs from a standard library as the framework provides all of the underlying infrastructure and a collection of design patterns and best practices necessary to solve a complete task.

If the library is a tool box, framework would be the workshop.

It would be fair to say that we can focus more on the usefulness and relevance of a library or framework for a specific problem to be solved. Frameworks come in handier when the task is usually complex in nature.

In context of this article, we will focus on JavaScript frameworks that are commonly used for ASP.NET MVC based application development. There are quite a few JavaScript Frameworks focusing on Mobile development aspect (and hence are termed usually as mobile frameworks). We would not be focusing on them for now.

Let's Shop JavaScript

A quick disclaimer – what holds true for shopping of a smartphone or car, is going to hold true for the choice of a JavaScript framework too. At the end, it is a matter of personal taste & various other factors. A JavaScript framework you choose for a specific requirement may or may not fit another project requirement.

We'll follow a methodology where we will analyze the frameworks with the following parameters – a brief history of the framework & its current status, key features, and some examples where they are used commercially. At the end, we will have a comprehensive comparison of these frameworks using various parameters.

We'll focus on the following five frameworks and libraries for this comparison – Angular JS/2.0, Backbone, Ember, React and Vue, as they are the most frequently used frameworks with ASP.NET MVC.

Let's look at how these are stacked against each other when it comes to their usage. The following infographic would help us do that:

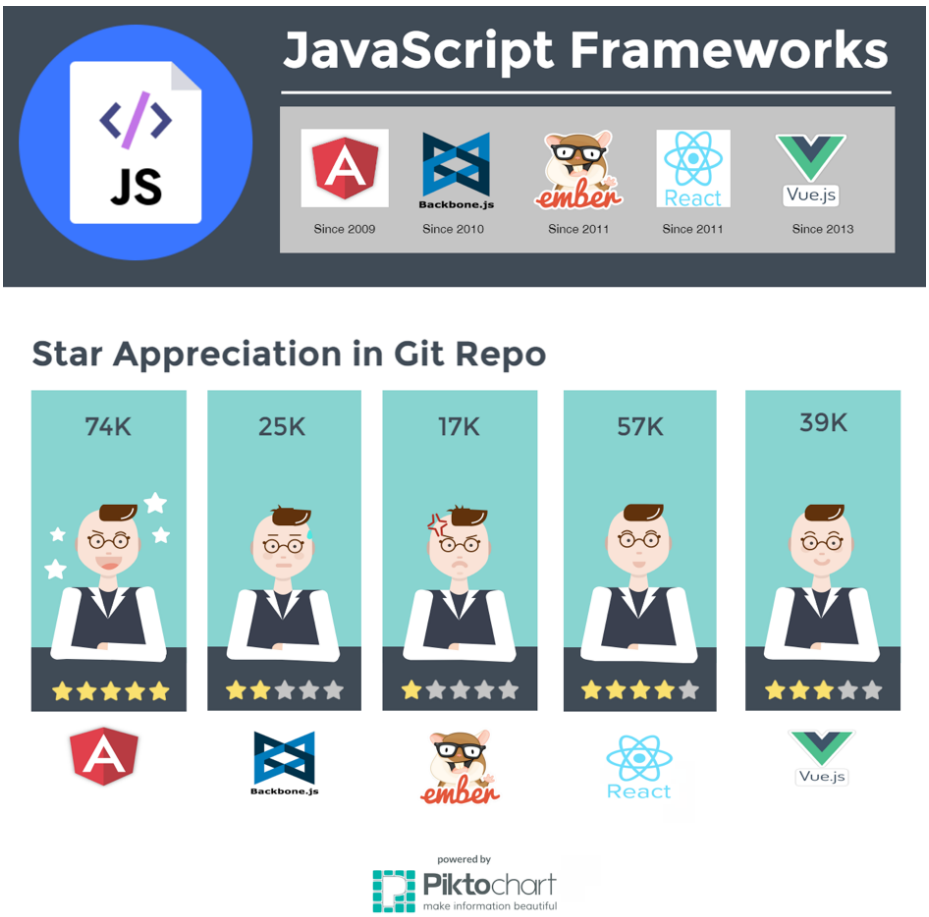


Figure 1: JavaScript frameworks Usage

Angular JS

Angular is the most-popular, most-used and most-discussed JavaScript Framework in context of web applications and that too ASP.NET MVC applications. Angular was developed by Google and was first released in 2009 under the MIT license. Since inception, it has been used widely for UI-centric web application development.

Some key features of Angular JS 1.x are:

1) Mainly UI-centric and hence focuses more on extending HTML features which makes it a right fit for web applications following MVC based architecture

2) Modular in nature

3) DOM based structure allows easy manipulation of data

4) Two-way data binding: This means that changes made to the view are instantly available to the model and vice-versa

5) Uses dependency injection to manage the dependencies of the functions used in code

6) Programmers that have coded in C# or Java (basically OOPs-based languages) find it relatively easy to work with Angular JS because of the structured approach followed.

You can find a pretty comprehensive Angular JS tutorial here.

Interestingly, while Angular JS 1.x has been extensively used, Google has been working on Angular 2.0 (note that "JS" is now dropped from the name) since around 2014.

Here are some key differences:

1) First and foremost, Angular 2.0 is NOT an upgrade of Angular JS 1.x.

2) Angular 2.0 is written in TypeScript and meets ES6 specifications. And that is why JS is dropped from the name – a small but not-to-miss detail.

3) Angular JS 1.x was not really built with mobile app form factor in mind. Angular JS 2.0 has mobile app support as one of the key design goals from the very beginning.

4) Angular 2.0 is more component-based by design as against Angular JS 1.x which is more controller-driven.

5) Angular 2.0 has better performance than Angular JS 1.x. This link provides more details. This is achieved by having some architectural tweaks while rendering HTML on server.

There are quite a few syntactical changes between Angular JS 1.x and 2.0 but we wouldn't go that deep right now. You would find ample blogs and articles explaining those. Please head over to www.dotnetcurry.com/tutorials/angularjs which has plenty of Angular tutorials.

Typically Angular JS 1.x or Angular 2.0 are used when we need to handle complex web UI requirements. Builtwithangular2.com has loads of example sites that are built using Angular 2. In addition to it, some commercial websites built using Angular JS that need a noteworthy mention are Weather.com, Upwork.com, freelancer.com, Netflix.com and the list is long.

Interestingly, Google is planning to release Angular 4 (yes, you read it right. There is no version 3!) by March 2017. Check out bit.ly/dnc-angular4 to know more about it.

To learn how to use AngularJS in ASP.NET MVC applications, check this tutorial bit.ly/dnc-angular-mvc.

Backbone.JS

Backbone was released in 2010 and is a fully-featured, yet a lightweight JavaScript framework. It is also relatively easy to learn as compared to other frameworks. If you are looking for a basic framework and you would like to add more stuff on top of it, then Backbone is the framework to use.

Here are some key features of Backbone.JS:

1) A light-weight JavaScript best suited for SPAs (Single Pages Apps)

2) It is primarily built on the MVP pattern

3) Extensive documentation making it quick and easy to learn

4) Controllers are optional in Backbone. You have Views and Models and then you have event-driven communication happening between those.

5) Events are built on top of regular DOM and they do the job of connecting models and views together.

6) Models can be easily tied to REST-ful APIs

Typically if you have a web app with most of its UI remaining the same, but only certain UI elements change based on user interaction, then Backbone.js could be the right choice. This holds true usually for SPAs and hence Backbone.js works very well for SPAs.

Some web apps that are built on Backbone.js are as follows - Pinterest, Foursquare, Walmart, Delicious and USA Today.

Here's a good tutorial introducing Backbone bit.ly/dnc-backbone.

Ember.JS

Ember.JS was initially released in 2011. It has gained popularity since then as it brings the best features of two popular JS libraries into one viz. two-way data binding (Angular JS) and server-side rendering of DOM (React.JS).

Following are some features of Ember.JS:
1) Ember provides a lot of features out-of-the-box. If you want more flexibility, you shouldn't consider this framework. However, if you want to focus more on the core development of UI/features and let JavaScript do the basic stuff, then Ember is the right choice to go with.

2) In Ember, objects can bind properties to each other. So a change in property of one object, can result in an appropriate update in the bound object details.

3) Similar to React.JS, it also provides nested views to handle complex UI.

4) You have the ability to use templates in Ember.js that would make the code more modular and easier to maintain.

5) It comes equipped with lot of tools including CLI (command line interface). This would help you add various components easily into your code.

Some examples of apps built using Ember.js are - Groupon, Vine and Apple music desktop application.

React.JS

React JS is a library, not a framework. React is focused mainly on the visual aspects of the application. It is a new frontend framework from Facebook that makes it easy to develop the V in MVC. This means you can use ReactJS similar to an Angular directive to render items and keep track of their state.

ReactJS.NET is a .NET specific JavaScript version of React.JS and can be used with ASP.NET MVC 4 or 5 and

even ASP.NET Core MVC.

Facebook and Instagram use React js very effectively. Both are UI centric applications and that is their key USP. React JS can be a good choice when you are going to develop a dynamic consumer-facing, UI-centric application that rerenders frequently. Using React JS to develop some business applications that need simple templating might not be a very good idea. Although React js has helped shape the future of JavaScript, just remember there is a time and place for every technology.

There is a time and place for great technology.

Here are some key features of React.JS:

1) It uses virtual DOM for all UI components. Essentially it creates another DOM in memory on the server-side. When user interacts with the web application, it tracks the changes in virtual DOM. Then, it does a "diff" of both DOMs and patches the browser DOM with the changes. This ensures that there is no re-rendering of DOM required.

2) React can run on the server (using Node js) and on the client. So you can build your MVC application using React instead of Razor to render a component on the server. The same piece of code can double as a client-rendering code too.

3) Nested views and loops are used often in React.JS for handling complex UI.

4) React can also work with any front-end framework you are working on. So if need be, you can smoothly replace any feature with React. For eg: switch `ng-repeat` with React code to gain its fast view rendering power.

In addition to Facebook and Instagram, Airbnb, Khan Academy, New York Times, WhatsApp web & Netflix are some of the companies that are using React.

Ravi Kiran has a more detailed article on React.js in this edition.

Vue.JS

Vue.JS (pronounced as "view") is a simple yet powerful library to build interactive web interfaces.
Vue.js works in tandem with MVC and allows you to keep your webapp untouched, while at the same time adding flairs of Vue in your views, wherever needed.

Following are some key highlights of Vue js:

1) It is relative easy to learn. The documentation is comprehensive enough and that is the only place where you need to look for help (as against forums etc.). If you have worked on Angular 1.x, you would find Vue.JS easier to adapt to.

2) It follows a declarative rendering approach when it comes to coding in Vue.JS.

3) Vue.js can be used for server-side rendering.

4) Provides the benefits of reactive data binding and reusable composable view components with ease of use.

5) Vue allows you to bind data models with your presentation layer. These models are not special in any way, they are simple JavaScript objects. You also do not need any special syntax, install any dependency or register event objects to work with these models.

6) There is no concept of virtual DOM in Vue. In Vue.JS, you can directly manipulate the DOM.

7) Templates and components are the building blocks that you can rely upon to build your web applications.

To see how Vue compares with other frameworks like React, Angular, Ember etc. check this link: vuejs.org/v2/guide/comparison.html

If you are new to Vue js, here's a good tutorial bit.ly/dnc-vuejs.

Some Noteworthy Mentions

Given that we have a new JavaScript framework popping up every now and then, it is nearly impossible to cover them all. While I have focused on some of the most popular and relevant JavaScript frameworks used with ASP.NET MVC today, here are some noteworthy mentions:

1) **Knockout.JS:** Released in 2010 under the MIT license, it follows the MVVM design philosophy. Although it appears to have lost popularity in recent times to Angular and React, you can always read about using Knockout with MVC and decide for yourself. Here's a handy tutorial - bit.ly/dnc-knockoutjs.

2) **Aurelia.JS:** It was released in January 2015. So it is relatively new. However it has ample community support already. It is a modular library and hence you can use specific independent libraries it provides, as per your needs.

3) **Bootstrap:** Bootstrap was developed by Twitter and is now made available as an open source offering. While it is certainly a JavaScript Framework that focuses on providing usual UI features, it is usually not regarded as a reliable & structured JavaScript Framework by the developer fraternity. It is typically associated mainly to UI design aspects, and is different from larger frameworks that provide features like modularity of design, best practices, and architectural robustness etc.

Some noteworthy commercial JavaScript frameworks for you to consider for your MVC apps would be KendoUI from Telerik, Wijmo from GrapeCity and IgniteUI from Infragistics.

Special Products for Your Special Needs

In the plethora of JavaScript Frameworks available, most of them focus on the UI aspect. However, there are some specific offerings that focus on a specific aspect of functional requirements. Here are some examples:

1) **Node.js:** This is a server-side JavaScript framework that you can use in conjunction with other client-side (read UI-specific) frameworks. This is a key component of MEAN stack that is gaining a lot of popularity when building entire apps on JavaScript-based stack.

Check out some Node.js tutorials at dotnetcurry.com/tutorials/nodejs.

2) **D3.js:** fully focused on chart related visualizations. This library comes with quite a lot of components that help you create cool charts. Check a tutorial on D3.js and ASP.NET Web API over here bit.ly/dnc-d3js.

3) **Babylon.js:** If you want to build some cool 3D games on browser, then you should use Babylon.JS.

JavaScript Frameworks - The Comparo

Let us compare various JavaScript Frameworks to understand its pros and cons. This is not an exhaustive list of parameters, but it will serve as a good guiding reference.

The factors considered for comparison are:

1) **Learning curve:** A JavaScript framework that is easier to learn would help in reducing your development time.

2) **Community support and documentation:** Long term support and active communities are vital for adoption. Good documentation is a cherry on the cake.

3) **Mobile-enablement:** Any web development eventually needs or leads to having a mobile (either hybrid or native) presence. So it should take minimal effort to mobile-enable your site.

4) **Efficient DOM manipulation:** A framework that excels in this would imply that it would perform well.

5) **JS Size:** In today's world, if your UI takes time to load, it immediately leads to poor UX, which eventually leads to a loss in business. You most certainly do not want your JavaScript to be the reason for it. Hence JavaScript size is an important aspect to consider. There are ways and means to mini-fy the script to reduce size. Also, the complexity of UI designed would play a part too. So although I have listed it here, you cannot compare the frameworks one-on-one just based on this parameter as there are quite a few other factors to consider too.

Certain factors that I have purposely not considered for this comparison:

1) **Performance:** This is subjective as it depends on the way a particular Framework is used, how complex the web UI is and so on. In general, all frameworks are more or less at par when it comes to performance, with few performing exceedingly well in some parameters. At the end, performance also depends on how we as developers or architects use them effectively. Just refer to stefankrause.net/wp/?p=392 which does a good job in benchmarking different JS frameworks.

2) **Reusable components:** With each framework being used so extensively, chances are very high that you would find reusable components for almost all of them without too much of a trouble. So this may not become a deciding factor. The same argument holds true for features like UI binding and as well as Routing.

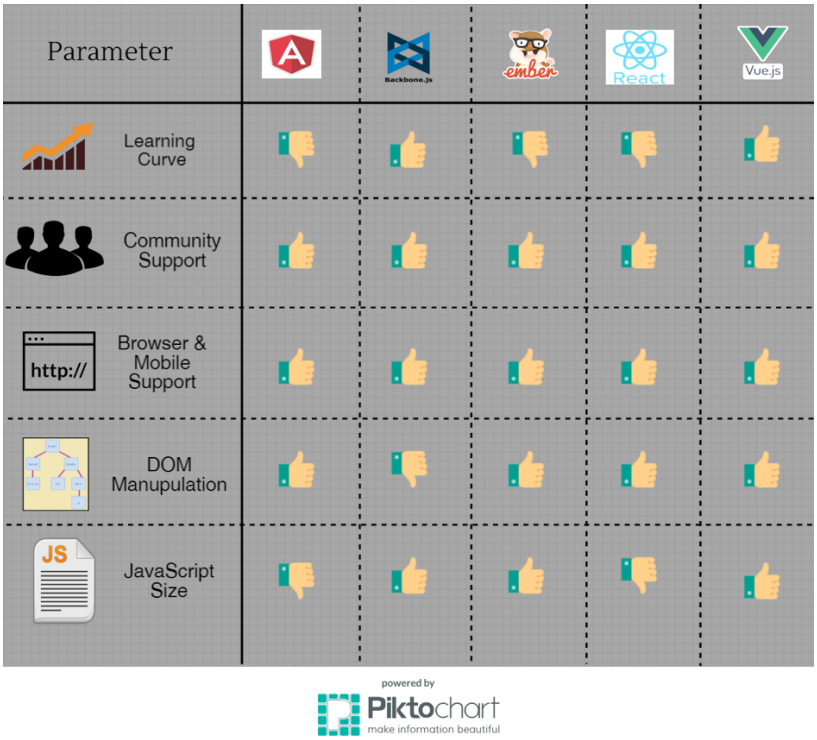


Figure 2: Comparing JavaScript frameworks

Conclusion:

We are spoilt for choices when it comes to JavaScript frameworks. The comparison above clearly shows that the frameworks are neck-to-neck when compared on key factors. In addition to this, there is a constant effort by the owners of these frameworks to optimize these frameworks, adopt the goodness of other frameworks to theirs, or release revamped versions of existing frameworks.

While we have so many choices at our disposal, we wouldn't go entirely wrong if we choose any of the mainstream, widely used JavaScript frameworks. What really matters is to have a good, thorough understanding of the business problem in hand, and then apply your known technical knowledge about frameworks to choose the apt one.

What also matters is your understanding of core JavaScript, which affects your judgement while picking a framework for your next project. Happy shopping! ■

Rahul Sahasrabuddhe

Author

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

Thanks to Suprotim Agarwal and Benjamin Jakobus for reviewing this article.

Gerald Versluis

App Ratings are important! Seriously! The better the reviews, the more likely it is for a mobile app to top the charts of an app store. However if your app looks bad, works bad or crashes, it will be raining one-star reviews all day long. And that is something that is very hard to recover from.

Building

5

star apps with

XAMARIN Test Cloud

Mobile application testing is quite challenging. For example, before submitting an app to the store, is it feasible to test the various form factors, connectivity types, different OS versions, device densities etc.? Is it possible to test your application on all the targeted devices?

Well now it is. Tools like the Xamarin Test Cloud enables you to test your app on thousands of *physical* devices. And it is not just for Xamarin apps!

Xamarin Test Cloud: First Look

Xamarin Test Cloud is a **UI acceptance-testing tool** for mobile apps. Introduced in 2014 with over 1,000 physical devices, Xamarin Test Cloud was unique in its kind. At Xamarin, they surveyed a large number of developers and concluded that 80% of the developers were relying on manual testing on devices. On the other hand, the survey also pointed out that 75% of these developers thought that the quality of their apps is top priority. However in all practicality, for a sole developer, or even a company, it is impossible to test everything on physical devices, especially in combination with all the different versions of Android and iOS out there.

Today, Test Cloud is located in a warehouse in Denmark and has over 2,000 different devices, while still adding about a 100 each month. These non-jailbroken devices are representative for actual devices that your users use every day.

For all these devices, you can write test scripts which can be executed automatically. The results are available to you in great detail. For each step, you can see a screenshot, how much was the memory usage at the time, as well as the CPU cycles. All this information is available for you, today!

In the future, they are even looking to expand the list of features and give you, the developer, control over a device. So not only can you run automated scripts on it, you can also remote debug your app on a device that you do not actually own. So if the app crashes or a bug is discovered on only a specific set of devices, you can reserve some time for it on Test Cloud and debug your app on it.

All of this makes up a very powerful test suite.

Xamarin Test Cloud Limitations

There are a few limitations as well. You cannot work with anything related to Bluetooth, there is no camera support, testing happens on Wi-Fi (no cellular data that means) and there is no Windows Phone support.

The other thing (which may be a limitation for some) is that all these features comes at a price. TADA!

The first tier starts at $99 per month. With this you can have unlimited apps, but you are limited to one concurrent device and one hour a day. This meaning that when you have six tests that take 10 minutes each, you can run them all once a day. The tests will be executed on one device at a time. If you have more concurrent devices, you could run on two devices at a time and the tests would only take 30 minutes. In Figure 1, you can see how Xamarin explains this concept on their website.

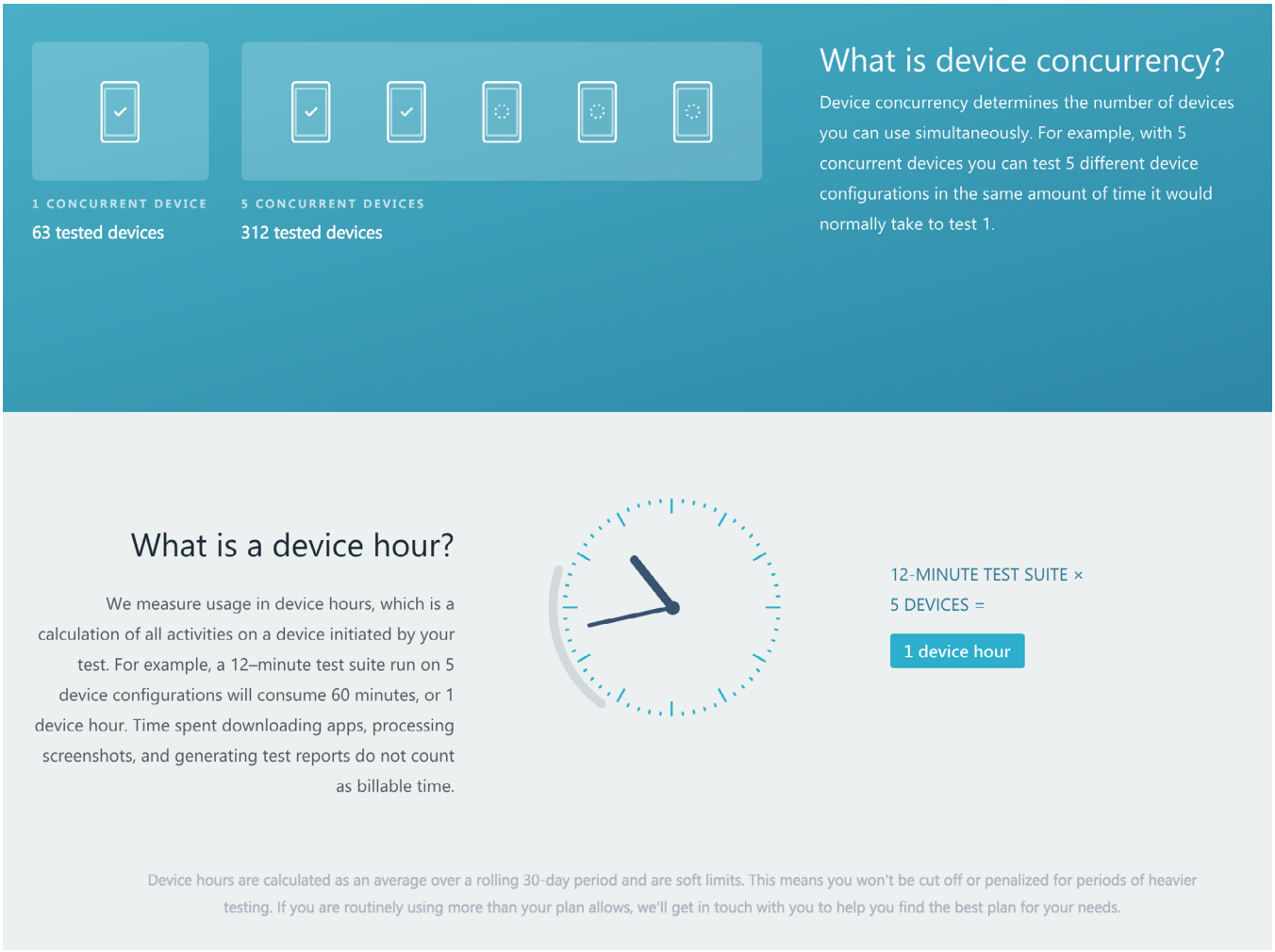


Figure 1: device hours and device concurrency on Test Cloud

For as much as $380 dollars a month, you will get three concurrent devices and five hours per day. Like I said; this can mean a lot of money especially if you are a one-man company.

But when you think about it; the number of devices and power that you get for this money is enormous. You cannot buy all those configurations yourself and find the time to run all the tests on them manually. So if you think about it the other way, it really is a bargain to keep the quality of your apps at a high level. If you need more convincing, there is a trail available. Also, if you are a MSDN subscriber, you can get 25% off the prices.

Finally, all this awesomeness can be integrated in any continuous pipeline. If you have read my article in the previous edition about setting up an automated pipeline for Xamarin apps, you can fit this part right in there. (link: bit.ly/dnc-auto-xamarin-app).

Writing Tests

Tests can be written for the Test Cloud in a couple of ways. At the time of this writing, three frameworks are supported: Calabash, Appium and Xamarin.UITests.

Calabash is an Automated UI Acceptance Testing framework that allows you to write and execute tests to validate the functionality of your iOS and Android apps. Read more about it at bit.ly/dnc-calabash.

If you already know a little bit about these frameworks, you might notice that Calabash and Appium are based on web-driver and the Ruby framework. This means you can write tests in Ruby, Java, JavaScript, Python and even PHP. This also means that not only C# Xamarin apps are supported, but also native iOS in Swift or Objective-C and Android Java apps.

Xamarin.UITest is based on the NUnit framework and has full IDE support for Visual Studio and Xamarin Studio. In the end, all these frameworks can achieve the same result. Basically, Xamarin advises to use the UITest framework when your app is a Xamarin app. The framework is easy to pick up because it is also in C# and has some small advantages like running the tests locally on your own device simulators.

If your app is a Java or Objective-C/Swift app, or for that matter a hybrid web app, you could choose Calabash or Appium. You can write the scripts in whatever tooling you want and upload the scripts along with the binaries of your app.

In this article, I will show you how to write tests for a Xamarin app in C# using the Xamarin.UITest framework in Visual Studio. By doing so, we will have an extra tool at our disposal: the Xamarin Test Recorder.

At the time of this writing, this tool has been in preview for a while. As you might expect with this application, you can record test scripts. Instead of having to code these scripts yourself, you can now tap through your app in the simulator or on your device, and it will generate the code to take these actions. The only thing that remains is to add some assertions statements to verify the results of the tests. To read more about the Test Recorder, take a look at this link: https://developer.xamarin.com/guides/testcloud/testrecorder/.

Creating a Test Run

A Test Run is what defines a test in the Test Cloud. With a Test Run, you specify a test series, the devices included in this test run and what locale the devices should have. When you log into Test Cloud, you will be taken to the dashboard. There are already a few sample apps in there for you to look at. By going through them, you can see what to expect from the test results.

Figure 2 shows you a screenshot of the dashboard.

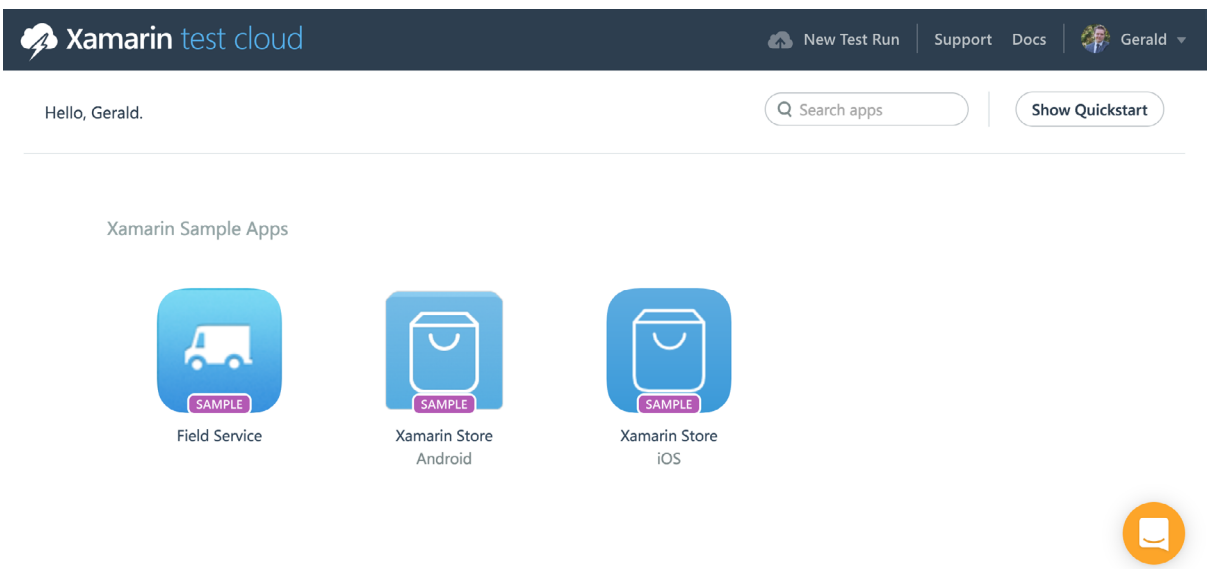


Figure 2: Xamarin Test Cloud dashboard

At the upper-right corner, you will also notice the 'New Test Run' button. Click it to create a new run. When you do so, you will be presented with a pop-up in which you get to choose if you want to create a Test Run for iOS or Android. Later, if you already have some apps in here, you can also create a new Test Run for that app.

After choosing a platform, a screen will show up where you can select the devices that are to be included in the tests. This screen in shown in Figure 3.

The screenshot shows iOS devices; if you have selected Android, there would be a lot of Android devices here.

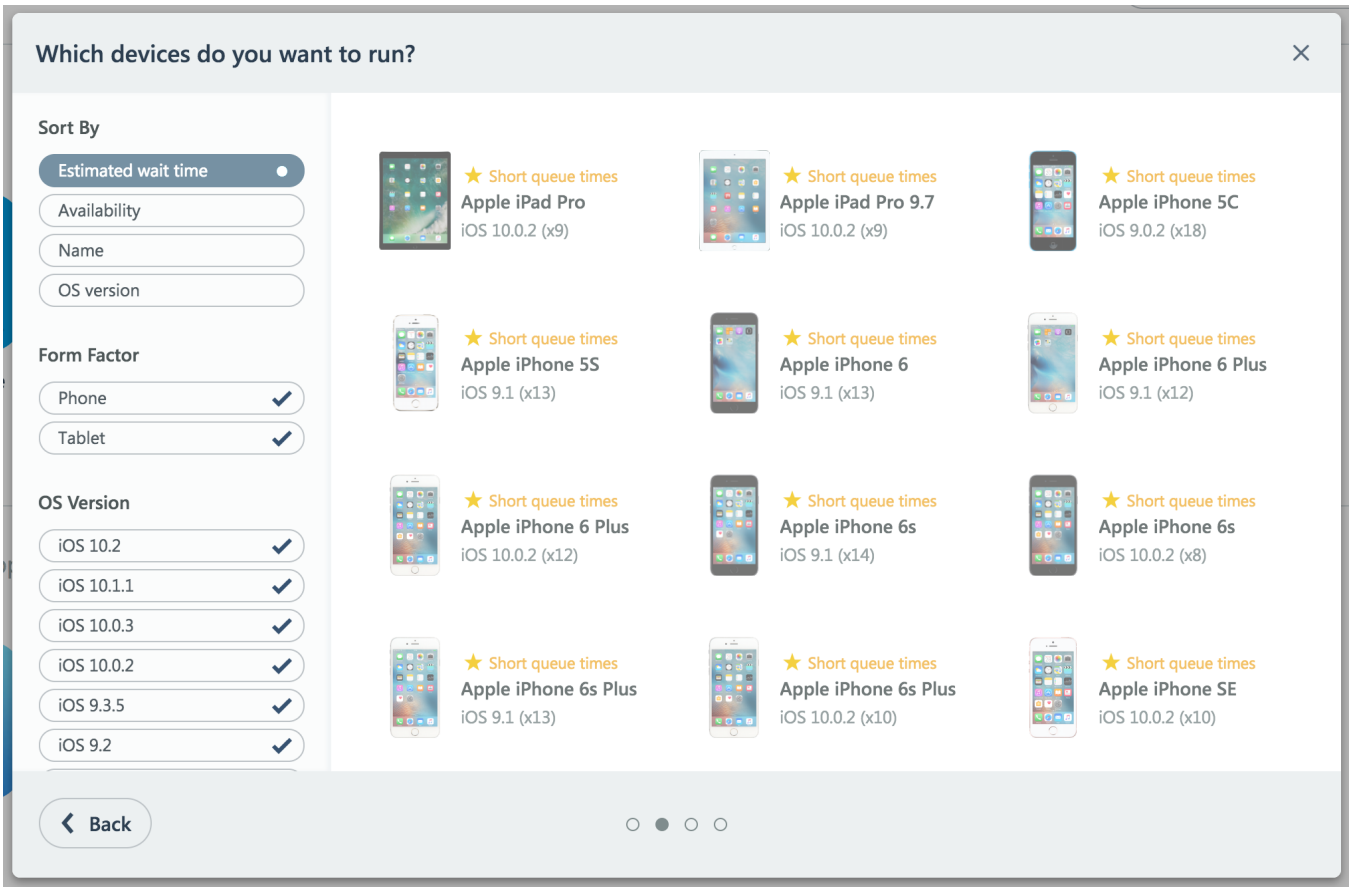


Figure 3: Selecting devices to include in the Test Run

The list of devices works very intuitively. You can sort by useful properties such as the *estimated wait time*. This time refers to the time that your test is queued in Test Cloud, in other words; how long do you generally have to wait before it is your turn. Because these are physical devices, there is a limit of how many tests can be running at one time.

Another handy field is *availability*. This enables you to show the devices based on how many of them are out there i.e. how many actual people in the world are using this device.

Other options include the ability to filter by form factor or OS version. This way you can fine-grain your selection. There is no real limit on how many devices you can select here.

After you have selected the devices that you want, go to the next screen to configure the test series that you want these devices to be in. This way you can break up your tests into multiple logical series. It is nothing more than a name that you group them by, think of them as categories. Also, you can choose a

locale. With this you can specify the language settings that are to be used before commencing your tests. This enables you to also test language specific features.

In the last step, do not just click the done button and expect it to save something for you, because it does not. In the final step, you are only presented with a console command as shown in Figure 4.

If we inspect it more closely, we can recognize some of the configuration we have just done. The locale is in there and so is the series. But there is also a switch for devices. As you can see, the devices are specified by a hash value. Depending on the devices and/or configurations you have selected, the hash will change.

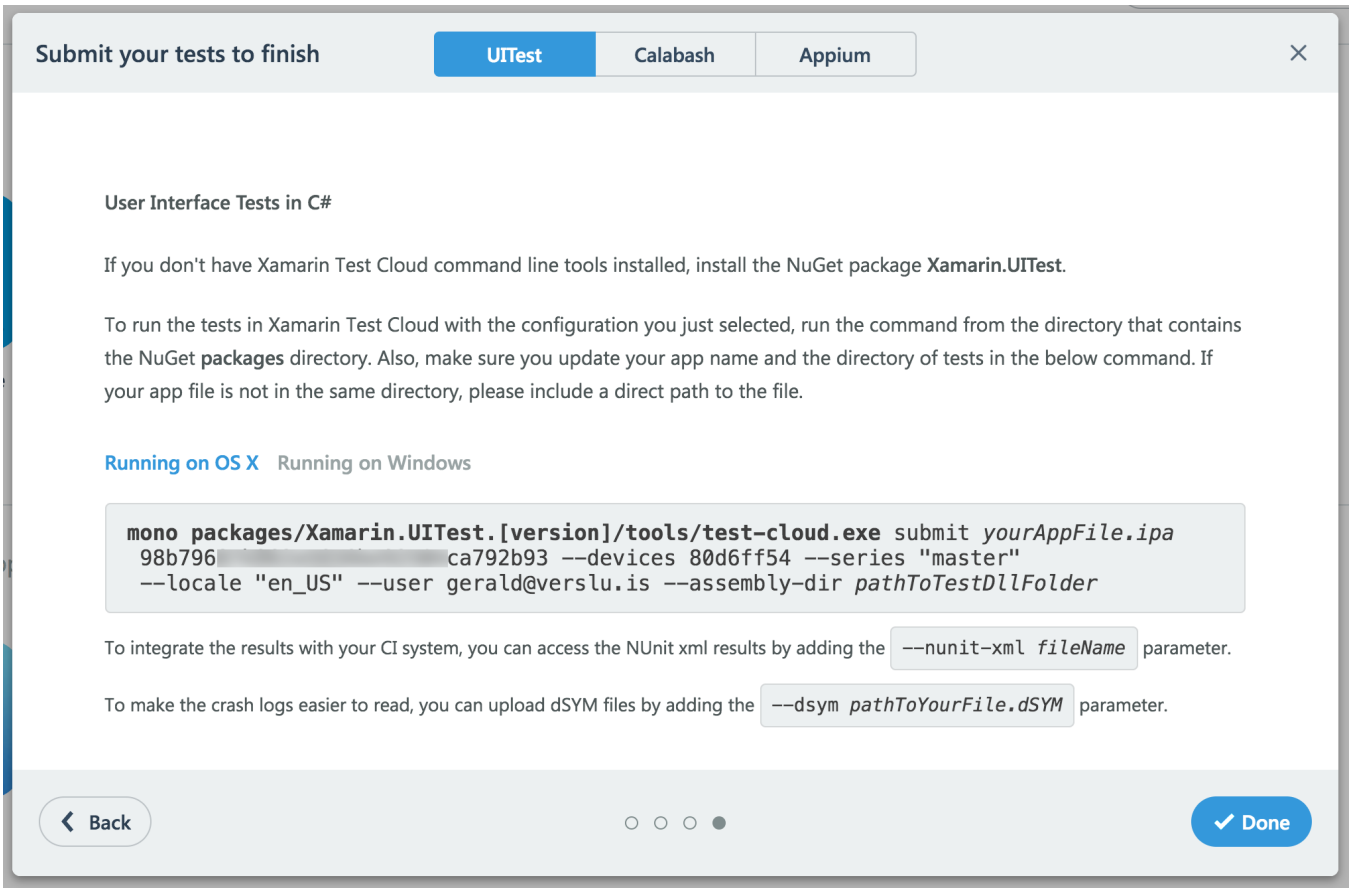


Figure 4: Final step of creating the Test Run

Also notice the longer hash value just before the devices switch. I have blurred a part of it, because this is your API key. Together with the email address that you will see later, this can be considered as your username and password.

Copy and paste this command to a notepad for later use.

Writing UI Tests

If you are already aware of writing unit tests in the .NET ecosystem, writing UI tests for Test Cloud will be a breeze. Before I show you how to write a test, let me show you the sample app that I will be using for this article. You can also find it on my GitHub here: https://github.com/jfversluis/TestCloudDemo.

For this sample, I have used a Xamarin.Forms app, using XAML. There isn't actually any difference when writing tests for the traditional Xamarin apps, besides from how to select controls. I will describe the

difference in a little bit.

The UI for this app looks similar to Figure 5. It is a very simple interface, just one label and two buttons. The green button will be a succeeding test scenario, and the red one is a failing scenario.

Figure 5: our sample test app

First observe what our XAML looks like. This is shown in Figure 6. Notice how the Label and two Buttons have an attribute called 'AutomationId'. This was a property introduced especially for Test Cloud.

With this attribute, we can easily find out controls from within the test scripts. This is also where the difference between traditional Xamarin and Xamarin.Forms is. Of course, the traditional Xamarin controls do not have the AutomationId property. In this case, you can use 'AccessibilityIdentifier' for iOS and for Android, the 'ContentDescription'. These are properties present in all controls on these platforms.

```
<?xml version="1.0" encoding="utf-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        xmlns:local="clr-namespace:TestCloudDemo" x:Class="TestCloudDemo.TestCloudDemoPage">

    <StackLayout VerticalOptions="Center" HorizontalOptions="Center" WidthRequest="300">
        <Label x:Name="ResultLabel" AutomationId="ResultLabel" Text="Test me!" />

        <Button AutomationId="SucceedButton" Text="This test will succeed" TextColor="White"
            BackgroundColor="Green" BorderColor="Green" BorderRadius="10" Clicked="Succeed_Clicked" />

        <Button AutomationId="FailButton" Text="This test will fail" TextColor="White" BackgroundColor="Red"
            BorderColor="Red" BorderRadius="10" Clicked="Failed_Clicked" />
    </StackLayout>

</ContentPage>
```

Figure 6: XAML for our UI

Actually, to make this work in Xamarin.Forms, we need some additional code which maps the 'AutomationId'

property value to the 'AccessibilityIdentifier' and 'ContentDescription' for the respective platforms. For iOS, go into the *AppDelegate.cs* and in the `FinishedLaunching` method, add this piece of code after the `Forms.Init();` line.

```
Forms.ViewInitialized += (object sender, ViewInitializedEventArgs e) => {
  if (!string.IsNullOrWhiteSpace(e.View.AutomationId))
  {
    e.NativeView.AccessibilityIdentifier = e.View.AutomationId;
  }
};
```

With this piece of code, the AutomationId property will be mapped. Do the same for Android too. Go to the *MainActivity.cs* and in the `OnCreate` method, after the `Forms.Init();` line, add this piece of code, which maps it to the 'right' property for Android.

```
Xamarin.Forms.Forms.ViewInitialized += (object sender, Xamarin.Forms.
ViewInitializedEventArgs e) => {
  if (!string.IsNullOrWhiteSpace(e.View.AutomationId))
  {
    e.NativeView.ContentDescription = e.View.AutomationId;
  }
};
```

If you examine these pieces of code closely, you will notice that it uses an event-handler at a very high level and which will be invoked for each view. As your app grows in complexity, this can become a big performance hit. To work around this, you can create a separate build configuration which holds a special compiler directive.

You are going to need this anyway, at least for iOS. Because for iOS we need to introduce some more code to make it work. By adding this code, some private iOS APIs are invoked, which is not allowed by the App Store review process. So, making a build with the Test Cloud code in place, will not be allowed in the App Store.

For iOS, install the *Xamarin.TestCloud.Agent* NuGet package. Then add some more code to the `FinishedLaunching` method, just like we did earlier. Behind the code that we have just introduced, initialize the Test Cloud agent by these lines of code:

```
  // Code for starting up the Xamarin Test Cloud Agent
#if ENABLE_TEST_CLOUD
  Xamarin.Calabash.Start();
#endif
```

Our projects are now ready to be used in the Test Cloud.

When we go back to the main page of the app and look at the code-behind for the buttons, we will see this code:

```
void Succeed_Clicked (object sender, System.EventArgs e) {
  ResultLabel.Text = "Hooray!";
  ResultLabel.TextColor = Color.Green;
}

void Failed_Clicked (object sender, System.EventArgs e) {
  ResultLabel.Text = "Whoops, that is embarrassing";
  ResultLabel.TextColor = Color.Red;
}
```

The green button will make the label value say "Hooray!" and make the color of the text green. And the red button will set the label to another text, as we will see later the text was not what it is supposed to be.

Now let's see how we can write a test for this. Add a test project to your solution by right-clicking your solution, go to 'Add' and choose 'New Project...'. In the 'Add New Project' screen go to the 'Test' category. This is shown in Figure 7.

As you can see, there are different kinds of test projects to choose from. The one that we are after is the 'UI Test App (Xamarin.UI Test | Cross Platform)'. There are separate projects for Android and iOS as well. Since my app is a Xamarin.Forms app, I will focus on the cross-platform app. However writing and running the tests aren't actually that different.

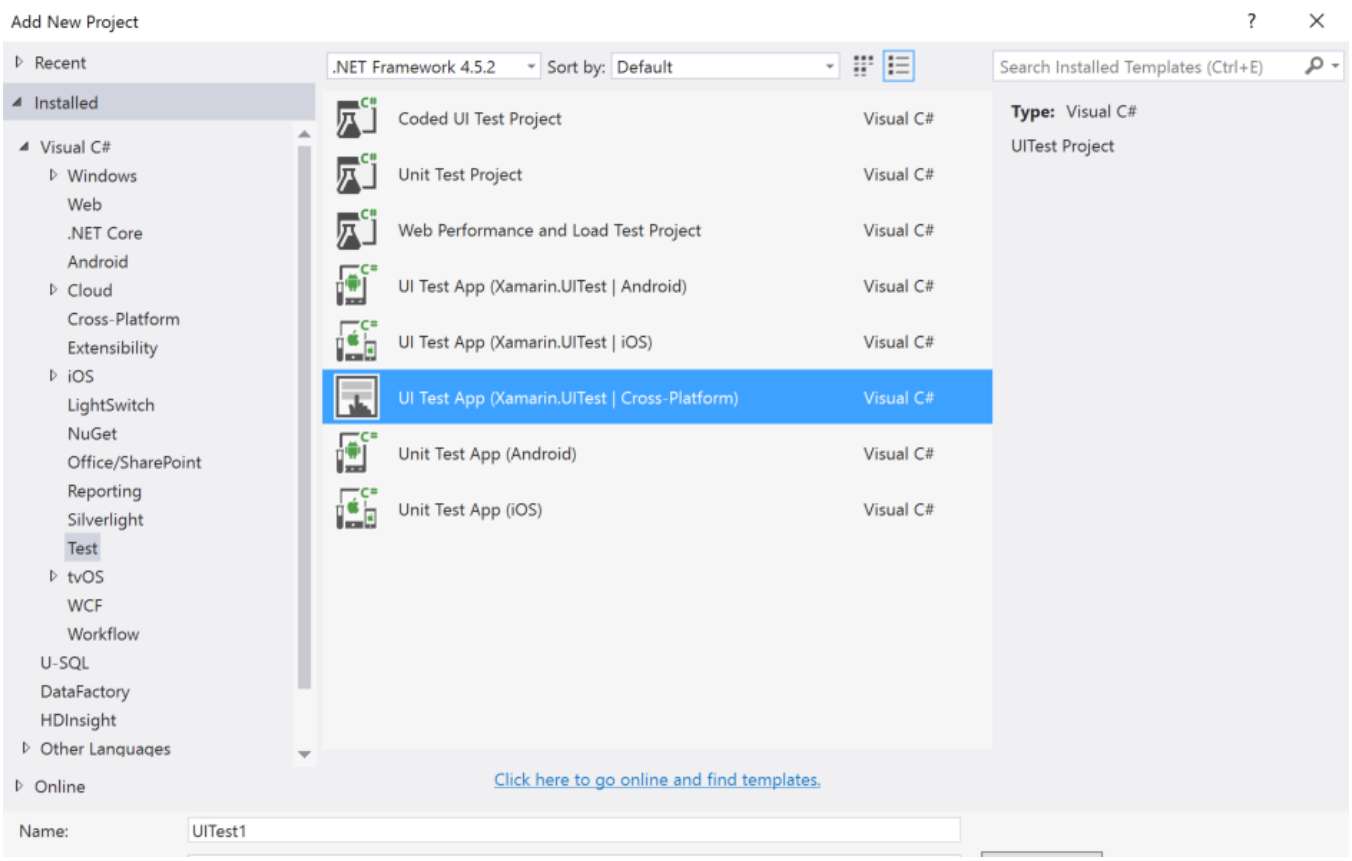


Figure 7: Adding the UITest project

Name your project appropriately and add it to your solution. If we look at the project structure, we see that it now consists of two files: *AppInitializer.cs* and *Tests.cs*. Like you might have expected, the first file contains code to initialize the test project and the latter contains the actual tests.

Initializing the tests does nothing more than create an `IApp` context which holds all kinds of methods to compose our tests with. Depending on the platform that we run it on, the interface gets a different implementation.

In the *Tests.cs* file, the real magic happens. You can see the initial contents in Figure 8. When you open it up there already is some code in there to help you get started. There is a constructor, which sets a field so that we can detect what platform we are on. Also there is a method `BeforeEachTest`, as the name tells you: the code in this method is ran before each of the separate test methods. The line of code in there refreshes the `IApp` context before each test so that the state of that object will not influence the results of another test.

And then there is the `AppLaunches` method which is handled like a test. The only thing it does is take a screenshot.

Figure 8: Initial contents of the Tests.cs file

You might have noticed that there are attributes at different levels. This way you can define which class contains tests and for which platform (`TestFixture`). You can also choose which method is the initializer method (`SetUp`) and which methods contain test code (`Test`). If you have written unit tests for .NET, this might look familiar to you as it uses the same structure.

For this sample, I will just use one class, but as you are going to expand your app and accompanying tests, you might want to break it up in different classes - at least per platform, but as the number grows, you might want to make more distinction per feature or screen. That is up to you.

Besides the code that is already here, let us create two more test methods:

```
[Test]
public void Press_Good_Button_And_Pass_Hooray()
{
  // Arrange
  // Nothing to arrange for this test
  // Act
  app.Tap(e => e.Marked(“SucceedButton”));
  app.Screenshot(“Green button tapped”);
  // Assert
  Assert.AreEqual(“Hooray!”, app.Query(e => e.Marked(“ResultLabel”)).First().Text);
}
```

```
[Test]
public void Press_Bad_Button_And_Fail_Boo()
{
  // Arrange
  // Nothing to arrange for this test
  // Act
  app.Tap(e => e.Marked("FailButton"));
  app.Screenshot("Red button tapped");
  Assert.AreEqual("Whoops, that is embarrassing...", app.Query(e =>
    e.Marked("ResultLabel")).First().Text);
}
```

First off, the tests are composed in the **Arrange-Act -Assert pattern**. You write some code to *arrange* the app to the situation that you want to test. Then write code which executes the logic that you want to test, which is the *act* part. And lastly you *assert* the outcome values by comparing the expected result to the actual result.

Like I have mentioned before, the `IApp` object, which is in the `app` variable, contains all the methods to compose our tests. For instance, `app.Tap()` allows you to tap an element on the screen just like a user would. In most of the methods, you can pass a `func` object. This will be the way to locate controls. Besides the Tap method, there is also `DoubleTap`, `PinchToZoomIn`, `ClearText` and `DismissKeyboard`, to name a few.

If we look at this line: `app.Tap(e => e.Marked("SucceedButton"));` you will see that we are looking for a control that is *marked* with the 'SucceedButton' identifier. This is the value that we put in our AutomationId property. Because we are using the AutomationId (and thus the platform-specific fields I mentioned earlier), we can use the `Marked` method. This method relies on the fact that you are using those fields. But with the `Query` method that you see in there as well, you can query controls on any property or property value.

In both the tests, you will see that I tap the button, take a screenshot and then assert the outcome. In the first test, the expected result and the actual result will match and thus pass, whereas in the second test, I have made a subtle change in the expected value by adding three dots at the end.

Before we can send this off to Test Cloud, we need to add the right references to our platform-specific app projects. Right-click the 'References' node in the test project and add the Droid and iOS project.

Running the Tests in Xamarin Test Cloud

I already told you that you can run the tests locally, but the true power lies in running them in the Test Cloud. Running the tests locally can be a bit tricky from Windows. iOS apps are not supported for local execution in Visual Studio. If you do want to do this, I would suggest to switch over to Xamarin Studio, where running tests locally is simpler. Just set the test project as the startup project and run it!

We will be focusing on running it in the Xamarin Test Cloud. We will use the command that we copied from the Test Cloud web interface when we created our Test Run.

From Visual Studio, when you have Xamarin installed, you can just right-click the test project and select the 'Run in Test Cloud' option. The rest is self-explanatory. For now, we will do this the hard way. Make sure

that you got the ipa/apk file for your iOS/Android app. In case of iOS, make sure the Test Cloud code was initialized, else you will run into an error message. For Android, do not use the Shared Mono Runtime. You can turn it off by going into your project settings. In my case, I will be running a test on Android through my Mac.

Look at the command we got earlier from the Test Cloud portal and replace the missing values. You need to specify the UITest version in the path, the path to the apk file and the path to the test assemblies. The resulting string will be something along these lines:

```
mono packages/Xamarin.UITest.2.0.5/tools/test-cloud.exe submit Droid/bin/Debug/
com.versluisit.testclouddemo.apk 98b79xxxxxxxxxxxxxxx792b93 --devices a44740d1
--series “master” --locale “en_US” --user gerald@verslu.is --assembly-dir UITests/
bin/Debug/
```

I have obfuscated my API key.

Start a console (or Terminal) window in either Windows or Mac and navigate it to the ‘Packages’ folder of your solution on your filesystem. Then paste in the command we just composed. If everything is OK, the files will be uploaded to Test Cloud and the tests will commence. In Figure 9 you will see the Terminal window uploading the files and running the tests.

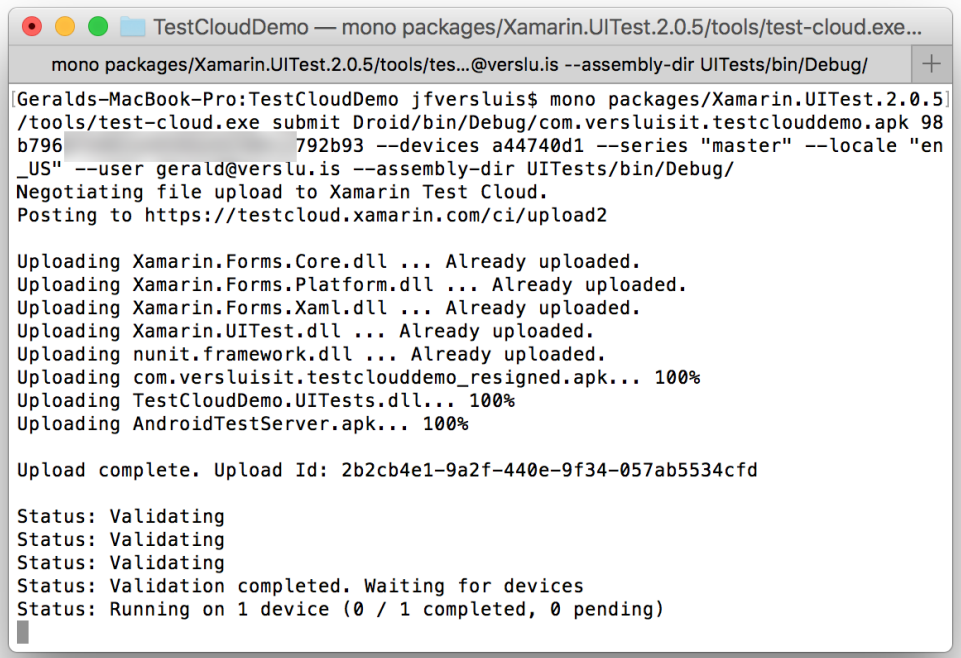


Figure 9: Sending our app to Test Cloud

If we now go back to the Test Cloud web interface, you can already see the tests in progress. Wait for the tests to be processed. You can follow the test progress using the terminal output. When the tests are completed, click through to the Test Run to see the results. An overview of the results can be seen in Figure 10.

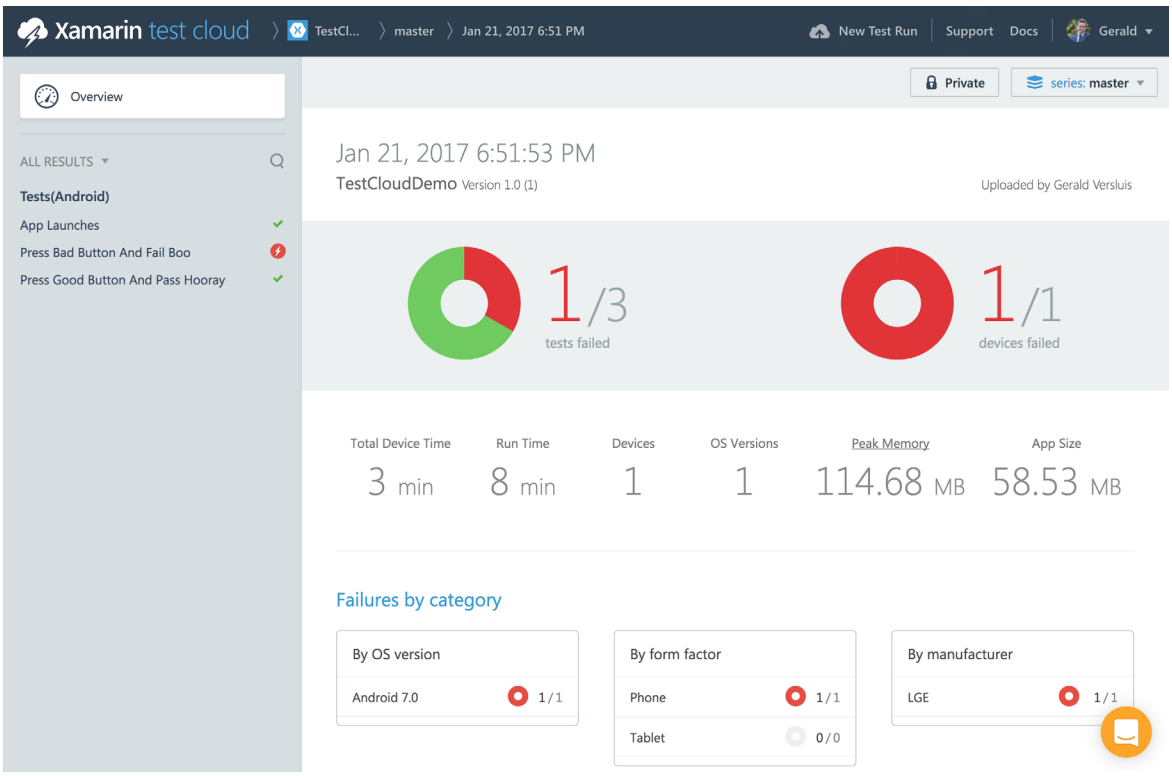


Figure 10: Overview of the test results

On the left-hand side, you see all the test methods we have defined. A fun thing to notice is that I named my test method as a sentence. I have separated the word in the method name with underscores like: `Press_Good_Button_And_Pass_Hooray`. As you can see, Test Cloud is smart enough to strip out the underscores so it looks nicer.

On the right-hand side, you see a summary of this test run. As we expected, our test failed.
When you click through on the left into the method, you will see all the steps that were taken. Take a look at Figure 11, where I have clicked through into the test where you can see all the details per step. You can see stats like what was the memory usage at the time? What was the CPU used at the time?

But probably the most important thing I find is that you can see a screenshot of the device.

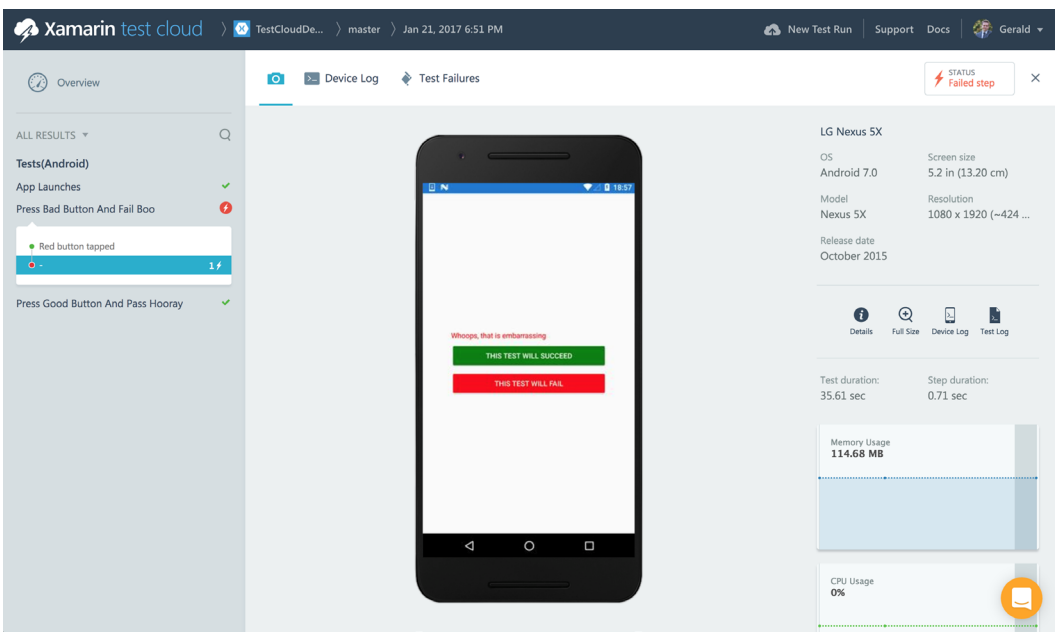


Figure 11: Details of the failed test

The complete device log is also available and you can see why a Test has failed under 'Test Failures'. In my case it says: 'Expected string length 31 but was 28. Strings differ at index 28. An ellipses (...) would have done it as the expected string could have been: "Whoops, that is embarrassing..." but instead was: "Whoops, that is embarrassing"'.

The awesome things about these tests is that it can compare values like these and check if an error snuck in there, but it also checks for usability. For instance, one time when I created a test and ran it through the Test Cloud, I placed a button near the bottom of the screen. When I composed the test, I selected a couple of devices, including some with a smaller form factor. While I did not expect any failures, some tests did fail! And the tests were failing on the smaller form factor phones. What happened was that the button near the bottom got pushed outside of the screen to a place where the user would never be able to reach it. Therefore the test failed.

So I was very happily surprised to see that Test Cloud also does these kind of checks. That way you are assured that you deliver high quality, five-star review apps.

Conclusion

In this article, we explored what the Xamarin Test Cloud is, how you can write tests for it and how to run them in the Cloud. I hope you have learned a thing or two and that you are just as enthusiastic about this as I am. I would like to encourage you to starting writing test and make your apps even better! ■

Download the entire source code from GitHub at
bit.ly/dncm29-xamarintestcloud

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 Suprotim Agarwal for reviewing this article.

Damir Arh

SINGLETON PATTERN OR ANTI-PATTERN

Singleton is one of the first and simplest software design patterns that you may encounter as a developer. However, it is often also considered an anti-pattern. In this article, we will take a closer look at it and learn how to implement it properly. We will explore some common use cases, and discuss why it is sometimes better to avoid it altogether. We will conclude with an alternative pattern you can use in most cases.

The Singleton Pattern

The Singleton pattern was originally introduced in the famous Gang of Four book (i.e. Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides). The authors described the intent of the pattern as follows:

Ensure a class only has one instance, and provide a global point of access to it.

To prevent multiple instances of the class, we need to make the constructor `private`. The class will take care of instantiation itself and provide access to the created instance via a `static` property. This is the simplest possible implementation in C#:

```
public sealed class Singleton
{
  private static readonly Singleton _instance = new Singleton();

  public static Singleton Instance
  {
    get
    {
      return _instance;
    }
  }

  private Singleton()
  {
    // place for instance initialization code
  }
}
```

The class instance is created during static type initialization, which happens when accessing any (static or instance) type member for the first time. If the type has additional members, this initialization could happen before we actually want to use the singleton instance. For types with expensive initialization, we might want to delay the instantiation until we really want to access it. The easiest way to achieve this is by using `Lazy<T>`:

```
public sealed class Singleton
{
  private static readonly Lazy<Singleton> _lazyInstance =
  new Lazy<Singleton>(() => new Singleton());

  public static Singleton Instance
  {
    get
    {
      return _lazyInstance.Value;
    }
  }

  private Singleton()
  {
    // place for instance initialization code
  }
}
```

`Lazy<T>` is a helper class introduced in .NET Framework 4 which performs the initialization when its Value

property is accessed for the first time, and ensures that the initialization will happen only once even in multi-threaded scenarios.

Singleton Pattern - Typical Use Cases

Singleton's design makes it very useful for accessing unique external resources, which require additional management. A typical example of such a resource is the application log: there is usually only one and it requires additional operations such as opening, closing, flushing, splitting into multiple files based on size or time, etc. A common approach to representing this resource in objected-oriented programming (OOP) paradigm is to implement a service with a simple interface, which transparently takes care of all the resource technicalities.

A singleton is a convenient way for accessing the service from anywhere in the application code.

The model quickly falls apart when the service not only provides access to operations but also encapsulates state, which affects how other code behaves. Application configuration is a good example of this. In the best case, the configuration is read once at the application start and does not change for the entire lifetime of the application.

However, different configuration can cause a method to return different results although no visible dependencies have changed, i.e. the constructor and the method have been called with the same parameters. This can become an even bigger problem if the singleton state can change at runtime, either by rereading the configuration file or by programmatic manipulation. Such code can quickly become very difficult to reason with:

```
var before = new MyClass().CalculateResult(3, 2);// depends on Configuration.
Instance
RefreshConfiguration(); // modifies values in Configuration.Instance
var after = new MyClass().CalculateResult(3, 2); // depends on Configuration.
Instance
```

Without comments, an uninformed reader of the code above could not expect the values of `before` and `after` to be different, and could only explain it after looking into the implementation of the individual methods, which read and modify global state hidden in `Configuration` singleton.

Let us look at the same code, this time explicitly stating its dependency on the `Configuration` class, which does not need to be a singleton any more:

```
var configuration = new Configuration();
var before = new MyClass(configuration).CalculateResult(3, 2);
RefreshConfiguration(configuration);
var after = new MyClass(configuration).CalculateResult(3, 2);
```

If values of `before` and `after` were different this time, one would immediately investigate the `configuration` that is being passed to all the methods.

Unit Testing

There are even more disadvantages to singletons when writing unit tests. Creating a dependency in the test and setting it up correctly before passing it to the method under test, is much more self-explanatory, than changing some value on a specific singleton:

```
// explicit dependency
var configuration = new Configuration();
configuration.Mode = Mode.Basic;
var result = new MyClass(configuration).CalculateResult(3, 2);

// implicit dependency
Configuration.Instance.Mode = Mode.Basic;
var result = new MyClass().CalculateResult(3, 2);
```

Additionally, the state of a singleton is actually a global state. Modifying it in one test will affect all other tests that run after it, unless they explicitly set the value back according to their requirements. This will make test results unreliable: a test will pass when it is run on its own, and will fail when it runs together with the other tests. Troubleshooting this can be difficult and time consuming.

Having no global state avoids such issues altogether.

However, even if the singleton has no state, it can still make testing more difficult in some aspects. With a singleton application log, the test code will still invoke the same logger implementation as production code. Unless we globally disable logging for tests (if that is supported), this will unnecessarily slow down the execution of tests and fill up the log files.

It will also make it almost impossible to test the logging in the method under test. Any logging calls will always end up in the real logger implementation, which probably is not and should not be designed in a way that would make it easy for tests to inspect it. If there was a way to replace the default logger in tests (e.g. by passing it explicitly to the class or method under test), we could have a different implementation for testing that would allow direct checking of logged messages.

Multi-threaded Applications

Another challenge for singletons are multi-threaded applications. Since there is only one class instance available to all threads, all the methods need to be thread-safe, i.e. they must work correctly even when called in parallel from multiple threads. This is a non-trivial requirement for a logger that logs the messages into a log file. I/O operations are not thread-safe and writing to a file needs to be synchronized (making the logging method slower) or queued (increasing the time between the method call and actual writing to the file).

Passing the logger as an explicit dependency instead of making it available as a singleton would allow each thread to use its own logger instance. This would avoid the above-mentioned issues with thread safety at the cost of having multiple log files – one for each thread. Depending on the requirements, this could be a better solution for the problem. If not, the same logger instance could still be passed to all threads.

Dependency Injection

Throughout the article, I have been advocating explicit passing of dependencies to classes in favor of implicit methods. This approach is called **dependency injection**. As the name suggests, the idea is to pass all dependencies as constructor parameters when instantiating the class, instead of creating new instances of dependencies inside the class or directly referencing global instances, such as singletons.

As we already learned, this gives full control over the dependencies to the caller. All the dependencies can be defined in a single location, i.e. the application entry point. This location is typically called the composition root. The actual location in the application code depends on the type of the application:

- In a console application, this is its startup code, i.e. the Main method.

- In web applications, this is the code handling the client requests, e.g. the controller factory in ASP.NET MVC.

- In desktop applications, this is the code responsible for setting up new views, e.g. view model locator in many WPF MVVM frameworks.

- In tests, this is each test method.

As you can see, whatever the runtime is, composition root is always the location, where new components are constructed to handle incoming requests. To learn more about composition roots and dependency injection in general, check the series of articles (bit.ly/dnc-yacoub) in DNC Magazine, written by Yacoub Massad.

The composition root is not only responsible for creating the main application components and providing dependencies to them, but also for controlling the lifetime of individual dependencies. Sometimes a new instance of a dependency will be passed to each component; at other times a dependency will only have a single instance for the entire lifetime of the application that is shared between all of the components, as is the case with singletons.

While you could handle all of that in your own code, you can usually make your job easier by using a dedicated dependency injection library, i.e. an IoC (inversion of control) container as they are often called:

```
// create the container
var container = new Container();
// configure scope
container.Options.DefaultScopedLifestyle = new WebRequestLifestyle();
// register dependencies
container.Register<ILogger, FileLogger>(Lifestyle.Singleton);
container.Register<IRepository, DbRepository>(Lifestyle.Scoped);
container.Register<HomeController>(Lifestyle.Transient);
// request composed instance
var controller = container.GetInstance<HomeController>();
```

The above snippet uses Simple Injector, one of the most popular dependency injection libraries for .NET. It demonstrates, how dependencies can be registered as implementations of interfaces (`FileLogger` and `DbRepository`) or as standalone class implementations (`HomeController`). It also sets up different lifetimes for dependencies: singleton (one instance for the duration of the application), scoped (one instance per scope, which can be configured separately – it is set to web request in this sample) or transient (new instance each time, even if the same dependency is required multiple times in a single object graph). By calling `GetInstance`, the container instantiates the requested class and provides it with required dependencies, e.g. `HomeController` could have `ILogger` and `IRepository` as its dependencies.

Modern application frameworks often have a basic dependency injection framework already built-in and provide the option of replacing it with another one for more advanced scenarios. ASP.NET Core is an example of such a framework. Using its built-in dependency injection, we can easily configure our own dependencies in `ConfigureServices` method of `Startup` class:

```
public void ConfigureServices(IServiceCollection services)
{
  services.AddMvc();
  services.AddSingleton<ILogger, FileLogger>();
```

```
  services.AddScoped<IRepository, DbRepository>();
}
```

Now we can add a constructor requiring these dependencies and the framework will automatically satisfy them, observing the configured lifetime:

```
public HomeController(ILogger logger, IRepository repository)
{
  _logger = logger;
  _repository = repository;
}
```

This allows us to define our FileLogger as a singleton in only a few lines of code, without the disadvantages brought by implementing the classic singleton pattern.

Conclusion:

Singleton is one of the basic software design patterns from the Gang of four book. It is an appropriate choice for providing managed access to unique external resources, but can quickly be abused by storing global state in it. Even without that, it can introduce difficulties when writing unit tests or developing multi/threaded applications.

As an alternative to singletons, we can pass the instance as an explicit dependency to class constructors that need it. To make this process easier, we can use a dependency injection library. Some application frameworks already have such a library built into them which can provide a complete replacement for the singleton pattern ■

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.