

ISSUE 49 JAN 2021

DNC MAGAZINE

www.dotnetcurry.com

Architecture of Cloud
Applications

Azure Data Studio

Full-stack Real time
Applications using Blazor

Tic Tac Toe in
F#
(Part II)

Nullable Reference
Types in C# -
Best Practices

Continuous Deployment
for Serverless
Application on Azure

WHAT IS
MACHINE LEARNING?

EDITOR'S NOTE



@suprotimagarwal
Editor in Chief

Developers! Developers! Developers! A decade ago, bathed in sweat, yelled the one and only Steve Balmer as if he was addressing a sect that's unlike any other.

But why am I bringing this up now?

In the Software ecosystem, change has always been disruptive and has occurred more frequently than ever.

Organizations have had the best of intentions while adapting to these frequent changes and have often pondered seriously accelerating their digital transformation journey. The actual transformation though has been quite slow due to restraints like time and budget.

That is, until now.

The pandemic that began in 2020 forced companies to reformulate their plans and pivot by setting up remote working environments. Years' worth of digital transformation, happened in just a matter of months! And at the center of this were the Developers and IT staff, playing a crucial role in this transformation. Sitting in remote locations, and dealing with unprecedented challenges, developers have been working asynchronously to skill, reskill and upskill themselves, and make their organizations more agile.

Nobody knows for sure what 2021 and the coming years has in store for us. I sincerely hope it's good for everyone. The most important thing to remember is that adapting to trends and circumstances is what has helped our ancestors survive, and ultimately, this applies to the software industry too. Companies that can fast-track their digital transformation and enable creative thinking, co-creation with services, tools (low-code) and technologies, to pivot as the situation demands, will thrive.

..and Developers, Developers and Developers will be at the helm of this journey. So yes Steve, we are a sect that's unlike any other!

Editor In Chief :

Suprotim Agarwal
(suprotimagarwal@dotnetcurry.com)

Art Director :

Minal Agarwal

Contributing Authors :

Yacoub Massad
Gouri Sohoni
Darren Gillis
Daniel Jimenez Garcia
Damir Arh
Benjamin Jakobus

Technical Reviewers :

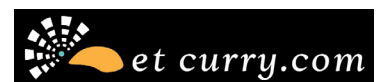
Damir Arh
Daniel Jimenez Garcia
Gouri Sohoni
Subodh Sohoni
Suprotim Agarwal
Yacoub Massad

Next Edition :

April 2021

Windows, Visual Studio, ASP.NET, Azure, TFS & other Microsoft products & technologies are trademarks of the Microsoft group of companies. 'DNC Magazine' is an independent publication and is not affiliated with, nor has it been authorized, sponsored, or otherwise approved by Microsoft Corporation. Microsoft is a registered trademark of Microsoft corporation in the United States and/or other countries.

 | Knowledge Visuals



Copyright @A2Z Knowledge Visuals Pvt. Ltd.

Reproductions in whole or part prohibited except by written permission. Email requests to "suprotimagarwal@dotnetcurry.com". The information in this magazine has been reviewed for accuracy at the time of its publication, however the information is distributed without any warranty expressed or implied.

CONTENTS



Architecture of Cloud Applications

06

Real Time Apps with Blazor
WebAssembly and SignalR

20

Diving into Azure Data Studio

50

Tic Tac Toe in F# - Part 2

68

What is Machine Learning?

78

Best Practices for Nullable reference
types in C#

90

Continuous Deployment for Serverless
Applications using Azure

100



Turn Your Application into a Barcode Powerhouse

Combine accurate barcode recognition with world-class image processing capabilities.

Companies rely on barcodes to manage their records and supply chains. They need applications that can quickly recognize and decode multiple barcode types as well as read and repair damaged, broken, and incorrect barcodes.



Detect, read, and write over 30 unique barcode types.



Clean up and repair damaged, incomplete, or poorly printed barcodes prior to scanning.



Set minimum confidence values scores for improved accuracy.



Locate and scan multiple barcodes on a page at speeds up to 1,000 pages per minute.



BarcodeXpress

by Accusoft

Barcode Xpress is a .NET SDK integration that allows your applications to scan, clean up, and process barcodes quickly and accurately. With over 80 functions for image processing and editing, it can identify and repair common and uncommon barcode types in milliseconds to keep your workflow moving smoothly.

Learn More About
How to Integrate High-Speed
Barcode Recognition Into
Your Applications

[Get Started](#)

COVERS C# v6, v7 AND .NET Core

THE ABSOLUTELY THE AWESOME

NOW INCLUDES
CHAPTERS ON
.NET Core 3.0
& C# 8.0

BOOK ON



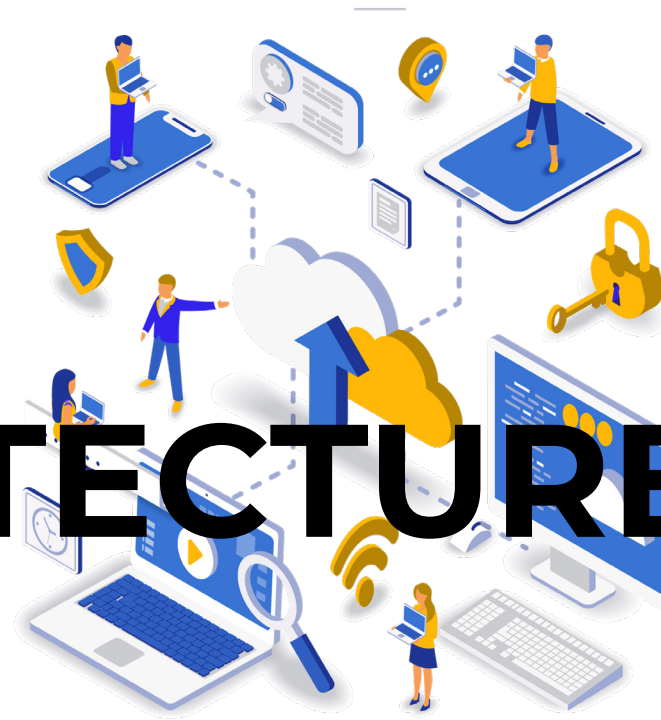
AND

.NET

DAMIR ARH



Damir Arh



ARCHITECTURE

OF

CLOUD

APPLICATIONS

In this article, I look at the definition of cloud applications and describe several design patterns that are especially useful in such applications.

*There's no unanimous opinion on what cloud applications really are. For the purpose of this article, I'm going to use the term to describe applications which are developed with the intention to be hosted on the cloud. Although often, another term is used to name such applications: **cloud-native applications**.*

Cloud-native applications

The official definition of cloud-native applications comes from the Cloud Native Computing Foundation (CNCF):

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

It's important to notice that the definition not only describes the internal architecture of cloud-native applications (microservices, declarative APIs), but also the way these applications are deployed (containers, service meshes, immutable infrastructure) and maintained (robust automation, frequent high-impact changes).

In this tutorial, I'm going to focus on the **internal application architecture of Cloud-Native Applications**.

For a more high-level overview of microservices and their deployment, you can read the [Microservices Architecture Pattern](#) article by [Subodh Sohoni](#).

The architecture of cloud-native applications or microservices has many similarities to the [Architecture of Web Applications](#) which I wrote about in my previous article from this series: [Architecture of Web Applications](#).

There are two main differences:

- The web applications as described in my [previous article](#) have their own user interface. The interface of cloud native-applications are predominantly APIs (typically REST based).
- Cloud-applications are implemented as a collection of services; each one of them running as a separate process and usually also in its own isolated environment. In contrast to that, monolithic web applications run as a single process (unless multiple instances are deployed for the purpose of load balancing).

The scope and size of these individual services varies across projects, but it is not uncommon for them to be as small as possible, limited to a well-defined bounded subset of the full application functionality. That's why the term **microservices** is often used for them.

The internal architecture of a single microservice is very similar to the internal architecture of a monolithic web application. They both include all parts of a typical multilayered application:

- Presentation layer (web page or API)
- Business logic layer
- Data access layer

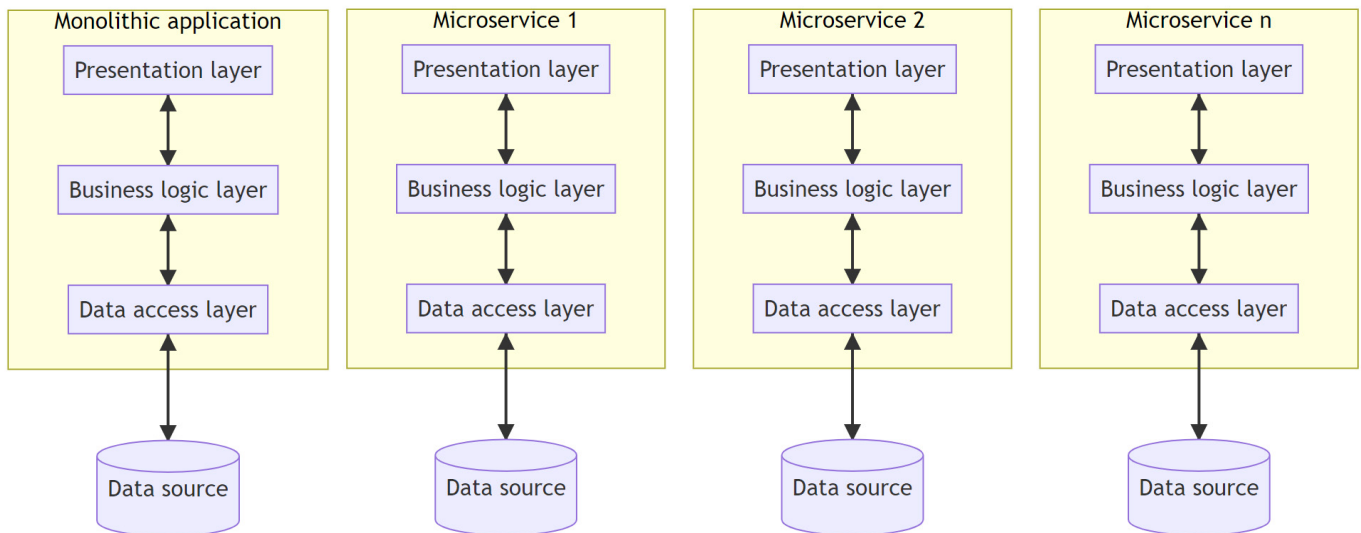


Figure 1: Layers in monolithic and microservices architecture

It's important that each microservice has not only its own data access code but also its own separate data storage (relational database or other). This not only allows it complete control over the data but also increases the need for communication between individual microservices.

From an architectural point of view, these properties are more important than the size of the microservice itself. That's why I'll simply use the term *service* for the rest of the article.

Communication between services

When one service needs access to data that's in the domain of another service, it can't simply read it from the common data store (because there is none), nor does it have direct access to the data store of another service. The only way to get to that data is through communication between the two services.

Synchronous communication

The most intuitive way for two services to communicate is most likely the [request-response pattern](#). In this case, the communication between the services consists of two messages:

- The request sent by the calling service to the called service.
- The response sent back by the called service to the calling service.

The interaction is fully synchronous: the calling service waits for a response from the called service before it can continue its processing.

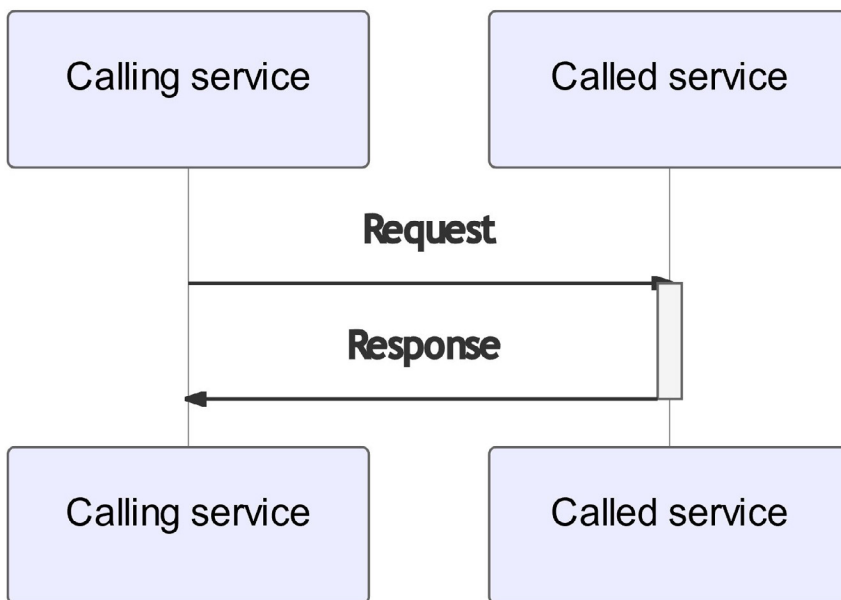


Figure 2: Request response pattern

The most common implementation of the request-response pattern is the HTTP protocol: the client (usually the browser) sends a request to the server (usually the web server) and waits for the response before it can render it.

RESTful services use the same protocol: the client (usually an application) sends an HTTP request to the server (usually a web service). When services expose their functionality as a (RESTful) API they can also use the same approach for communicating between each other: one service acts as the client and another service acts as the server.

In the .NET ecosystem, the recommended stack for implementing RESTful services is [ASP.NET Core Web API](#).

The framework is very similar to ASP.NET Core MVC (and not just by its name). Individual endpoints are implemented as action methods in controllers which group multiple action methods with the same base URL (and usually also related functionality):

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    private static readonly string[] Summaries = new[]
    {
        "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot",
        "Sweltering", "Scorching"
    };

    [HttpGet]
    public IEnumerable<WeatherForecast> Get()
    {
        var rng = new Random();
        return Enumerable.Range(1, 5).Select(index => new WeatherForecast
        {
            Date = DateTime.Now.AddDays(index),
            TemperatureC = rng.Next(-20, 55),
            Summary = Summaries[rng.Next(Summaries.Length)]
        });
    }
}
```

```
    .ToArray();  
  }  
}
```

The main difference is that there is no view in RESTful services. The response is a DTO (data transfer object) that's automatically serialized as JSON (JavaScript Object Notation), and deserialized back again on the client. If needed, the serialization and deserialization process can be customized through serialization options.

The available endpoints and the structure of requests and responses for each one of them can be described using the [OpenAPI specification](#). [Swashbuckle](#) is the most common library that's used with ASP.NET Core Web API to generate OpenAPI descriptions of RESTful APIs.

When implementing the ASP.NET Core Web API action methods, the same patterns can be used as for ASP.NET Core MVC applications: **Dependency Injection**, **Repository**, **Unit of work**, etc.

I described them in more detail in my previous article from the series: [Architecture of Web Applications](#). When calling the RESTful APIs from another service, the same remote proxy pattern can be used as described in another article from this series: [Architecting .NET Desktop and Mobile applications](#). Other useful patterns **Retry** and **Circuit breaker** are covered later in this article.

Although RESTful services are the most common API implementation today, they are not the only way to implement an API using the request response pattern.

The predecessor of RESTful services is [SOAP](#) web services. They were different from RESTful services in many ways:

- They used XML as the serialization format instead of JSON.
- They used the [RPC \(remote procedure call\)](#) model instead of a resource-based model.
- The protocol was much more complicated, with [many extensions](#).

In .NET Core, there's no framework or library available for implementing SOAP services. In .NET framework, WCF (Windows Communication Foundation) was used for that purpose but it wasn't fully ported to .NET Core (nor are there any plans for that).

There's only a limited client for SOAP services available which doesn't support all the protocols. For implementing the services, the only option is [Core WCF](#) – a far from complete open-source WCF implementation for .NET Core.

The main disadvantages of SOAP were large messages and incomplete support for all extensions on different development platforms which made cross-platform compatibility difficult. It doesn't make much sense to develop new SOAP services today.

A better alternative for RPC-based APIs is [gRPC](#) which doesn't have either of the above-mentioned disadvantages of SOAP services:

- Messages are serialized using [protocol buffers](#) which allow small binary messages.
- The protocol has [wide support across many development platforms](#).

.NET Core provides full support for gRPC services and clients. The service interface and messages are described using a protocol buffers' `.proto` file:

```
syntax = "proto3";

option csharp_namespace = "GrpcService";

package greet;

service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}

message HelloRequest {
  string name = 1;
}

message HelloReply {
  string message = 1;
}
```

The file is used as an input to autogenerate a base class for the service implementation. The service class derives from that class and implements the service methods:

```
public class GreeterService : Greeter.GreeterBase
{
  public override Task<HelloReply> SayHello(HelloRequest request, ServerCallContext
context)
  {
    return Task.FromResult(new HelloReply
    {
      Message = "Hello " + request.Name
    });
  }
}
```

For the client, the `Grpc.Net.Client`, `Google.Protobuf`, and `Grpc.Tools` NuGet packages must be installed in the project. The latter will autogenerate a remote proxy from the service `.proto` file which can then be used from the application code:

```
var channel = GrpcChannel.ForAddress("https://localhost:5001");
var client = new Greeter.GreeterClient(channel);
var response = client.SayHello(new HelloRequest
{
  Name = "World"
});
```

To learn more about gRPC in .NET Core, read the [gRPC with ASP.NET Core 3.0](#) article by Daniel Jimenez Garcia.

Asynchronous communication

No matter how efficient the chosen underlying protocol is, Synchronous communication between services has its disadvantages:

- The calling service can't complete its operation until it gets the response from the called service. Because of network latency, IO operations are always much slower than local processing and should be kept to a minimum.
- The calling service becomes dependent on the called services. If any of them fail, the calling service will fail as well. This will make the whole application less reliable. If only one service stops working, any services depending on it will stop working as well.

Although these are inherent to all distributed systems, switching to asynchronous communication between the services can to some extent help with both these problems because the calling service doesn't have to wait for the response from the called service anymore.

In the **Resilience section** later in the article, I will introduce additional patterns for handling challenges of communication between multiple services.

A common pattern for asynchronous communication is [publisher-subscriber](#).

When implemented effectively, instead of directly calling another service, the publisher service sends a message to a central event bus or message broker which is typically a separate service with the only responsibility of delivering the messages.

Examples of such services in Azure are [Service Bus](#), [Event Grid](#), and [Event Hubs](#). The other services can subscribe to receive those messages based on their types. The event bus persists messages and guarantees that the subscriber will be delivered the message even if it was busy or not operational at the time when the message was published.

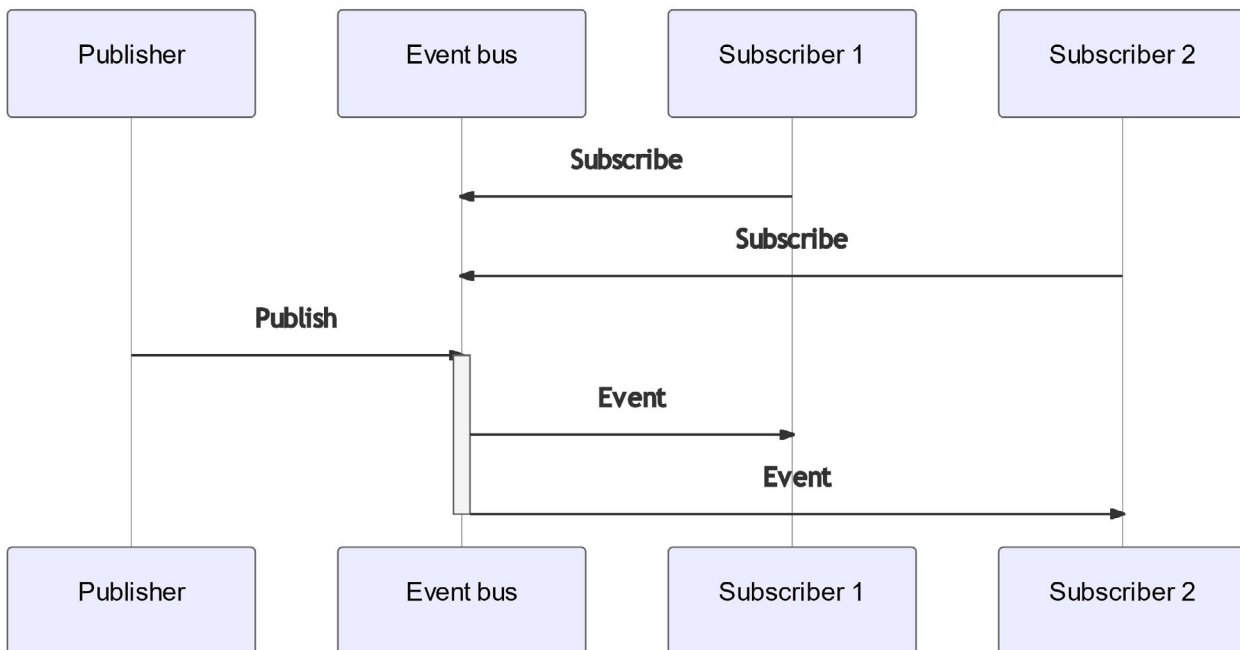


Figure 3: Publisher-subscriber pattern

This approach can be expanded further into [event sourcing](#) with event stream processing. In comparison to the publisher-subscriber pattern, the events in this approach are typically more granular and persisted permanently. This allows them to be replayed in full when a new consumer is introduced which processes the same stream of events in a different way to bring new insights into data.

The asynchronous nature of this communication affects the overall behaviour of the application:

- The services aren't directly dependent on each other anymore. Consequently, the calling service will not fail if the called service isn't working. The event will still be published, and the calling service will successfully complete its operation. Due to event persistence, the called service will process the event eventually. This drastically reduces the impact of a single failing service on the overall application.
- When the calling service completes its operation, the processing most likely hasn't been completed yet. It gets processed later when all the subscribers complete their own part of processing as well.

In a real-world application this means that a confirmation for the order could be sent out before it was fully processed. In a worst-case scenario, this could mean that one of the items ordered will be out of stock and the order will not be delivered in full within the original time estimate. To ensure consistency in data stores where the operation has already been processed, a [compensating transaction](#) might be needed to update the state according to a failure that occurred later. In more complex scenarios the [Saga](#) design pattern can help with orchestrating the transactions for different outcomes.

- The calling service can't get any data from other services to do its own processing. Since it's also not supposed to directly access data stores of other services, it must have a local copy of any data it needs. This copy of data will have to be kept in sync with the representative data source.

Client interaction

Of course, client applications (web, mobile, etc.) also need to communicate with several of the services that comprise the application. This communication is synchronous most of the time (clients expect direct response to their requests). APIs exposed as RESTful services are most widely supported across different client technologies.

API gateway

Clients that directly communicate with a multitude of services end up being very closely coupled to them. Many implementation details must be exposed to them for such communication to work; for example, how the load balancing of each service is handled.

To reduce this complexity, an [API gateway](#) can be introduced between the clients and the services. This means that the client only communicates with the API gateway which then forwards these calls to the appropriate service. Underlying changes in implementation can often be hidden from the clients. The API gateway can effectively serve as a type of a [façade](#) for the services.

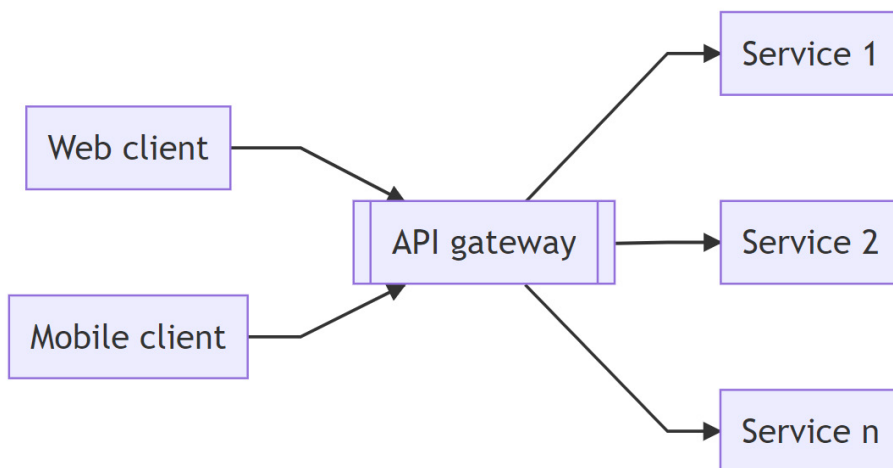


Figure 4: API gateway as a façade for services

In addition to simply forwarding the request to the appropriate service, the API gateway will often also be responsible for many cross-cutting concerns, such as authentication, [SSL termination](#), caching, logging, load balancing, etc.

In simple scenarios, a reverse proxy such as [NGINX](#) can take the role of an API gateway. For more complex requirements, dedicated solutions are available, such as [Azure API Management](#).

Backend for frontend

The role of the API gateway can be pushed another step further. Instead of simply exposing the APIs of individual services to the clients, these individual APIs can be combined into a tailored API for a specific client, such as a mobile application.

This approach of per-client APIs is described by the [backend for frontend \(BFF\) pattern](#). Multiple clients will each have its own BFF service although the calls from all of them are in the end handled by the same underlying services.

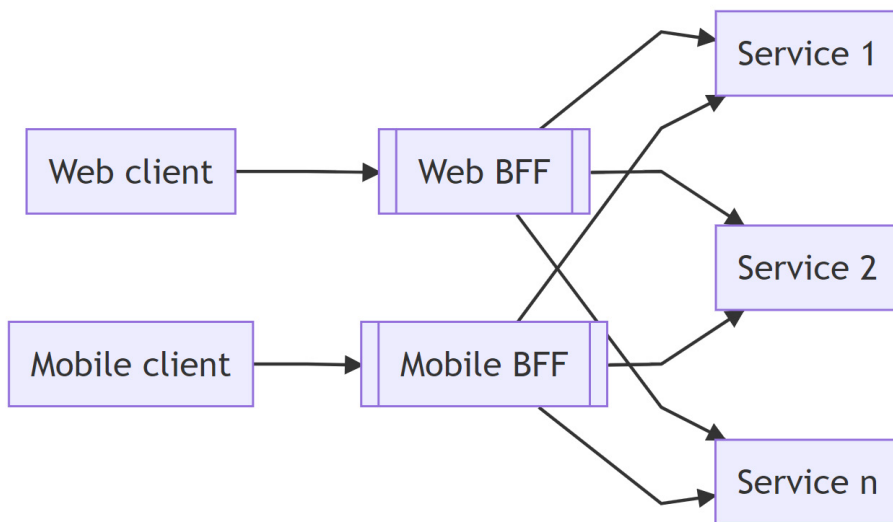


Figure 5: Each client has a separate backend for frontend

The scope of BFF services can vary. They might only orchestrate APIs of several underlying services, i.e., they call multiple services synchronously, wait for their responses and then combine them in a single response to the client request.

To improve performance, they can include a caching layer. This can be basic [response caching](#) for replaying responses to identical request for the time the cache is valid. But it can also include more sophisticated techniques.

A common pattern for more advanced caching of data is [materialized view](#). Instead of caching responses or data received from underlying services, the service has its own data store which can be used to efficiently retrieve requested data, removing the need for calling other services. This data is a duplication of data stored elsewhere and doesn't represent a source of truth.

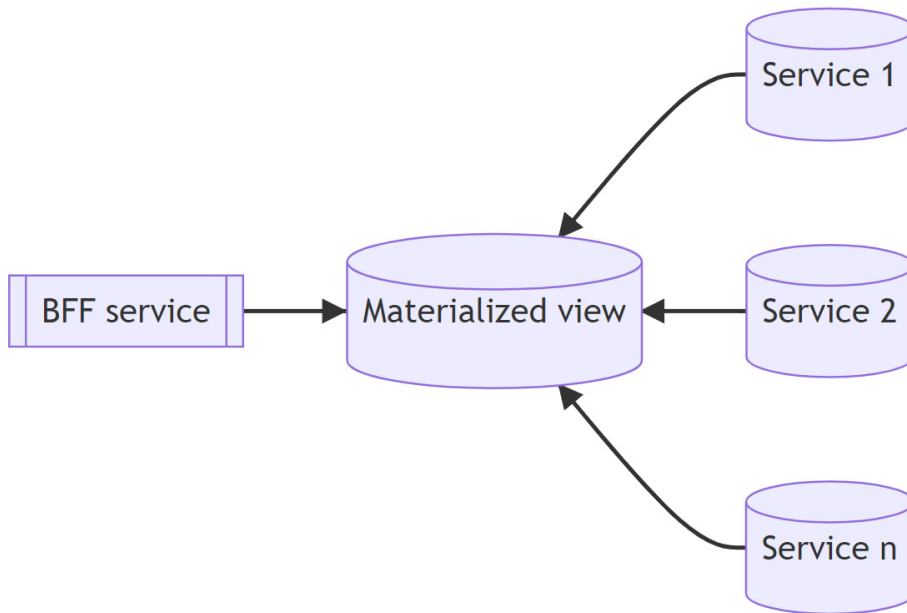


Figure 6: Materialized view is updated with data from multiple sources

The important part is, how the data is updated after the initial data synchronization or migration. This depends on the requirements and how stale the data may be. In an ideal scenario, it will automatically be updated in response to the changes of the source data with minimum delay. This can be achieved using the previously described **publisher-subscribe pattern** or **event sourcing**.

The materialized view can be subscribed to all events related to changes of data, that it persists. If other services publish such events for all data, this ensures that the data in the materialized view will be up to date most of the time.

Resilience

Cloud applications are essentially distributed applications. This changes how communication between different parts of application is performed.

Instead of it being predominantly in-process communication, which is inherently reliable, most of the intra-application communication is taking place over network, which is more often subject to failure.

When working on a distributed application for the first time, this aspect is often overlooked due to common false assumptions that were listed as the [fallacies of distributed computing](#) by L. Peter Deutsch and James Gosling in the 1990s.

To make the application more resilient to such failures and keep it working, additional measures must be taken. There are proven patterns available that can be used.

Retry and exponential backoff

The [Retry pattern](#)'s main concern is how to respond to a failed network call.

Of course, the caller can always immediately give up and propagate the failure as a response back to its caller. However, there is a possibility that the failure was transient, i.e., the next identical call will succeed.

In that case, it makes sense to retry the request before giving up. This quickly raises new questions:

- How many times should the call be retried before finally giving up?
- How long should be the delay between the retries?

There is no universal answer to these questions.

The best approach strongly depends on the specific request and the type of the error returned. The latter should make it clear at least whether the failure was transient or not.

A special category of errors are timeouts because they don't necessarily mean that the called service has given up on processing. It could be that a client in the call chain has simply given up on waiting. This means that the request might still be successfully completed. This makes it particularly dangerous to repeat the request if it isn't idempotent, i.e. the end state will be different if it's executed multiple times.

In general, a common approach to handling delays between retried requests is [exponential backoff](#), i.e., the delay between the requests is increased exponentially. This ensures that the delay will be minimal if it was indeed a transient failure and the second request will succeed.

On the other hand, if the failure persists for a longer time, it prevents the called service from being overwhelmed by the increasing rate of repeated requests from multiple clients. Of course, after a certain threshold is reached, the caller will give up retrying and report a failure itself as well.

In the .NET ecosystem, the most popular library for handling transient failures is [Polly](#). It revolves around creating policies for handling individual types of exceptions:

```
var retryPolicy = Policy
    .Handle<HttpRequestException>()
    .Retry();
```

This policy can then be used when executing an action that could throw such an exception:

```
var result = retryPolicy.Execute(() => CallService());
```

The policy above will simply immediately retry a single time after a matching exception (considered to be transient) is thrown. But it's just as easy to create a policy for multiple retries with exponential backoff:

```
var retryPolicy = Policy
    .Handle<HttpRequestException>()
    .WaitAndRetry(new[] {
        TimeSpan.FromSeconds(1),
        TimeSpan.FromSeconds(2),
        TimeSpan.FromSeconds(4)
    });
```

Instead of explicitly specifying the delays, they can also be calculated:

```
var retryPolicy = Policy
    .Handle<HttpRequestException>()
    .WaitAndRetry(5, retryAttempt =>
        TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)));
```


The library makes it very easy to define detailed policies matching specific requirements. It's a great alternative to implementing such behaviour manually.

Circuit Breaker

The **Retry pattern** is localized to a single request.

This means that if a service is sending multiple requests to a single service, each one of them is handled independently of the others. However, if a request to a service is failing, it's very likely that other requests to the same service will be failing as well.

This is where other patterns come into play.

The **Circuit Breaker pattern** acts as a common proxy for all requests to a particular service. It monitors for failing requests and based on preconfigured rules, transitions between three states:

- **Closed:** The called service is operating normally. All requests are passed to it.
- **Open:** The called service is currently failing. Any requests to it are not passed to the service and fail immediately.
- **Half-open:** After a certain period in the open state, the requests to the service are again passed to it. However, the tolerance for failure is reduced. If any requests fail, the state will switch back to open. Only after the service seems to operate normally for some time, the circuit breaker returns into the closed state.

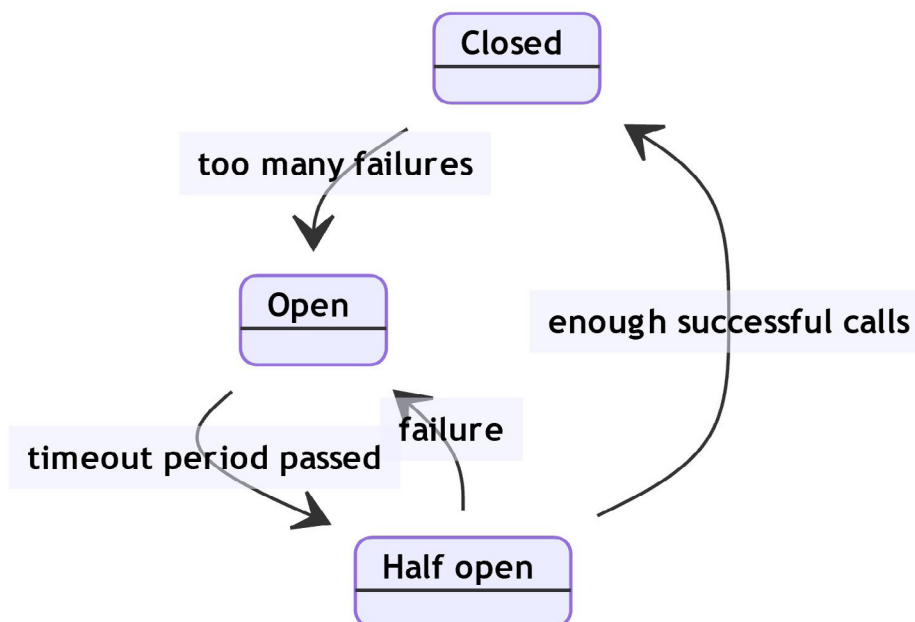


Figure 7: State diagram for circuit breaker pattern

A pattern like circuit breaker can significantly reduce the number of requests sent to a service with transient issues. This can be helpful in its recovery as it isn't overwhelmed with incoming requests which it can't handle.

As for the retry pattern, the **Polly** library includes an easy-to-use implementation of the circuit breaker

pattern:

```
var circuitBreakerPolicy = Policy
    .Handle<HttpRequestException>()
    .CircuitBreaker(2, TimeSpan.FromMinutes(1));
```

It makes perfect sense to combine both patterns: retry the requests as needed, but also track the failures and eventually switch the circuit breaker to open state. The library supports that as well:

```
var combinedPolicy = Policy
    .Wrap(retryPolicy, circuitBreakerPolicy);
```

Using appropriate error handling mechanisms can noticeably contribute to the overall reliability of a distributed cloud application.

Conclusion

The article covers some of the patterns that are especially useful in distributed cloud applications. It starts with an overview of different approaches to communication between services, both synchronous and asynchronous. It continues with some specifics of communication between clients and the services, introducing the API gateway and backend for frontend patterns.

The final part addresses the issue for reliability and transient failures in cloud environments. It introduces the retry and circuit breaker patterns as useful tools for dealing with them.



Damir Arh
Author



Damir Arh has many years of experience with software development and maintenance; from complex enterprise software projects to modern consumer-oriented mobile applications. Although he has worked with a wide spectrum of different languages, his favorite language remains C#. In his drive towards better development processes, he is a proponent of Test-driven development, Continuous Integration, and ContinuousDeployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and writing articles. He is an awarded Microsoft MVP for .NET since 2012.



Technical Review
Daniel Jimenez Garcia



Editorial Review
Suprotim Agarwal

Modern monitoring & analytics



Aggregate metrics and events from 400+ technologies including .Net, Azure, and AWS



Seamlessly pivot between correlated data for rapid troubleshooting



Search, analyze, and explore enriched log data



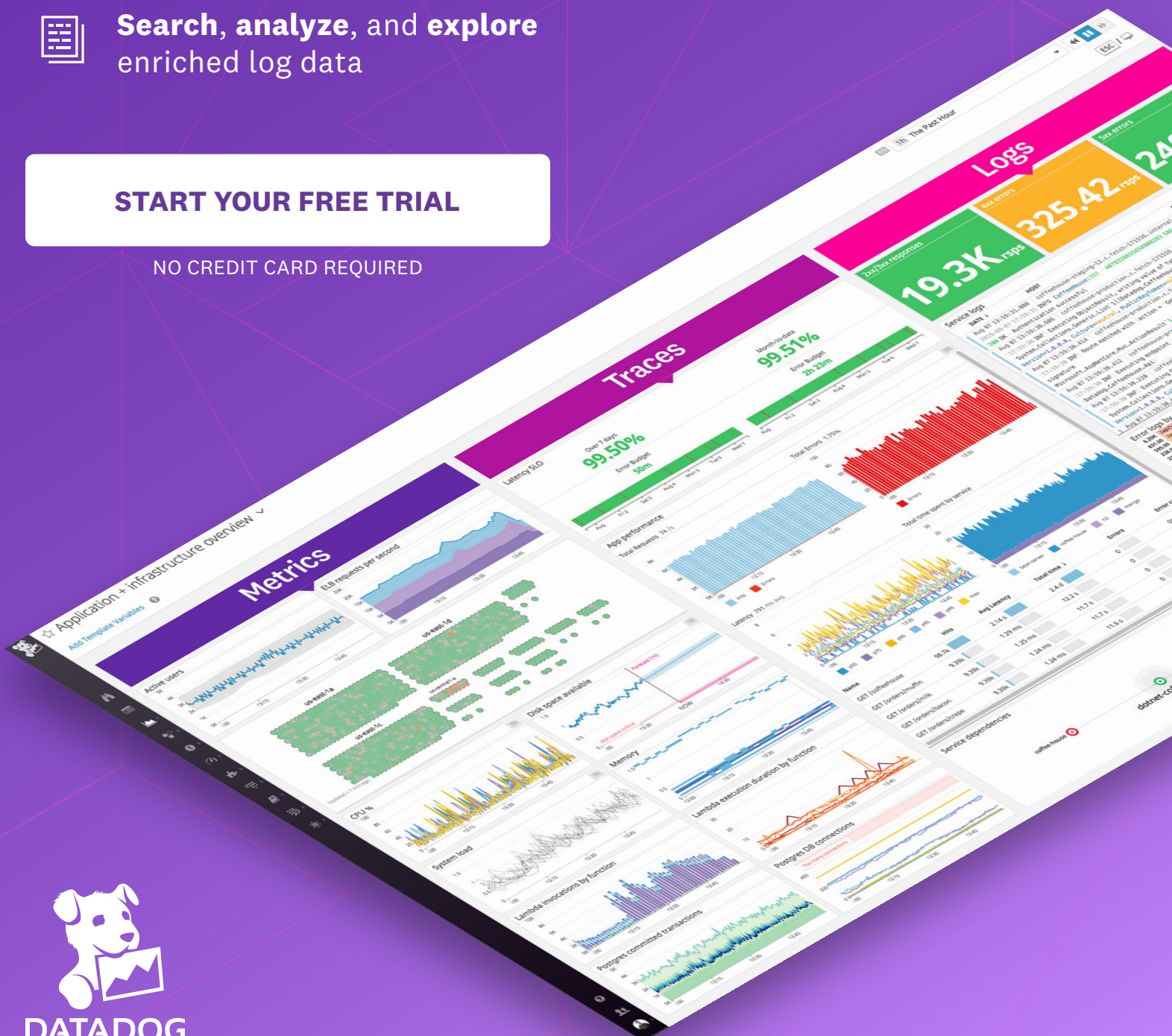
Trace requests across distributed systems and alert on app performance



Monitor your applications and API endpoints via simulated user requests

START YOUR FREE TRIAL

NO CREDIT CARD REQUIRED





FULL-STACK
REAL TIME
APPLICATIONS USING
BLAZOR
WEBASSEMBLY,
SIGNALR AND C# 9

During the Build 2020 event, Microsoft described their journey towards [One .NET](#). This vision can be summarized as being able to **use .NET to build any kind of application that runs anywhere**.

Blazor is recognized as an important piece of that vision, enabling full-stack web applications using .NET. After the WebAssembly hosting model for Blazor graduated from its experimental phase, teams finally have a way to run .NET in the browser. Using Blazor, allows you to develop across your frontend and backend applications using the same language and much of the same tooling.

In this article, **we will put Blazor WebAssembly under test by building a small real time application** where users can create and participate in surveys. The application will leverage SignalR for its real time functionality across both frontend and backend.

And given that .NET 5 and C# 9 are just released, we will use the latest ASP.NET libraries and check out some of the new language features like record types.

Building small sample applications can be a great exercise for anyone to explore whether Blazor delivers on its promise. Some of you will find using the same language and tooling makes you more productive and reduces friction. Others might miss tools, libraries and developer features taken for granted in the JavaScript ecosystem.

Hopefully this article will let you form your own idea or prompt you to try it out and explore for yourself! You can download the article code from [GitHub](#).

Much of this article applies to Blazor Server as well. If you are considering Blazor Server, just introduce into your evaluation its own benefits and downsides. For more info check the [official docs](#).

Getting started with Blazor WebAssembly

To begin with, there is some foundational work we need to go through in order to get a working solution with a Blazor WebAssembly frontend and an ASP.NET Core backend.

The very first thing we need is to create a solution with the necessary projects. This is straightforward enough using the standard ASP.NET templates. We will use the *hosted model* of Blazor WebAssembly since that gives us both frontend and backend projects out of the box.

Run the following command to create the new solution, which we will name **BlazorSurveys**:

```
dotnet new blazorwasm --hosted --output BlazorSurveys
```

This will create a new folder aptly named BlazorSurveys, containing the new solution and the generated .NET projects. I would also recommend initializing a git repository, so you can safely make changes, stage/commit them and go back in your history if necessary:

```
cd BlazorSurveys
dotnet new gitignore
git init
```

You should have a solution with three different projects, BlazorSurveys.Client, BlazorSurveys.Server and BlazorSurveys.Shared. Each is found in the respective Client, Server and Shared folders.

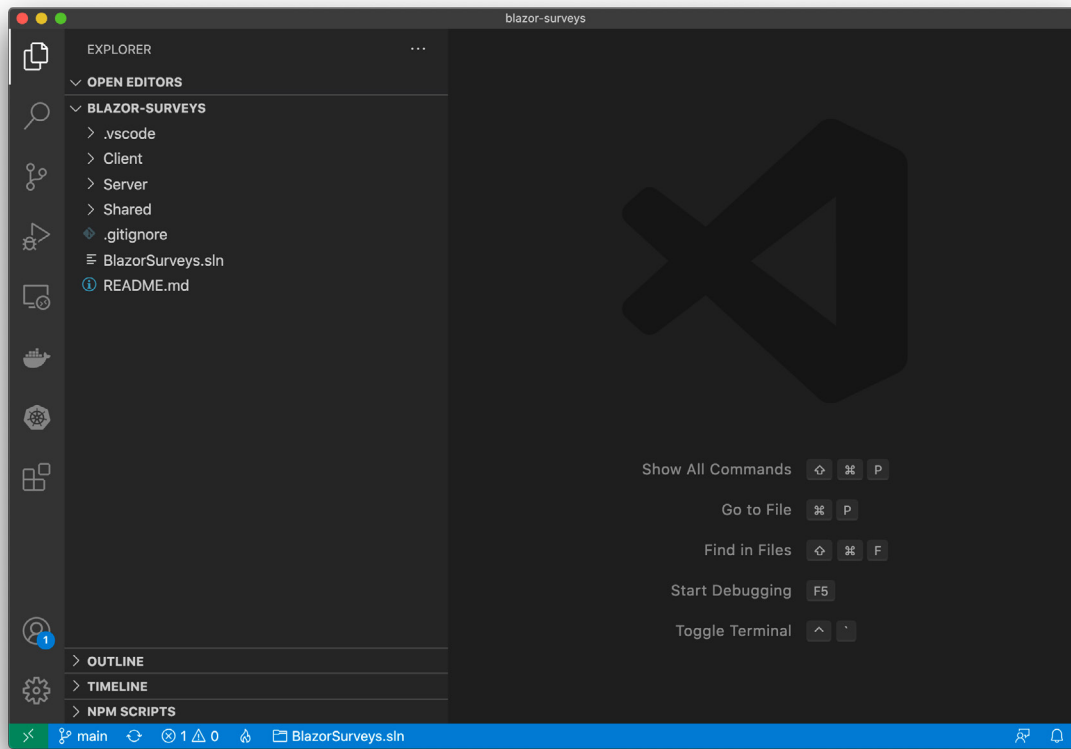


Figure 1, generated solution

Make sure everything is working fine by building and running the solution by using either of the following commands from the solution root folder:

```
# manually build and run the app  
dotnet run -p Server  
# automatically rebuild and restart the app with code changes  
dotnet watch -p Server run
```

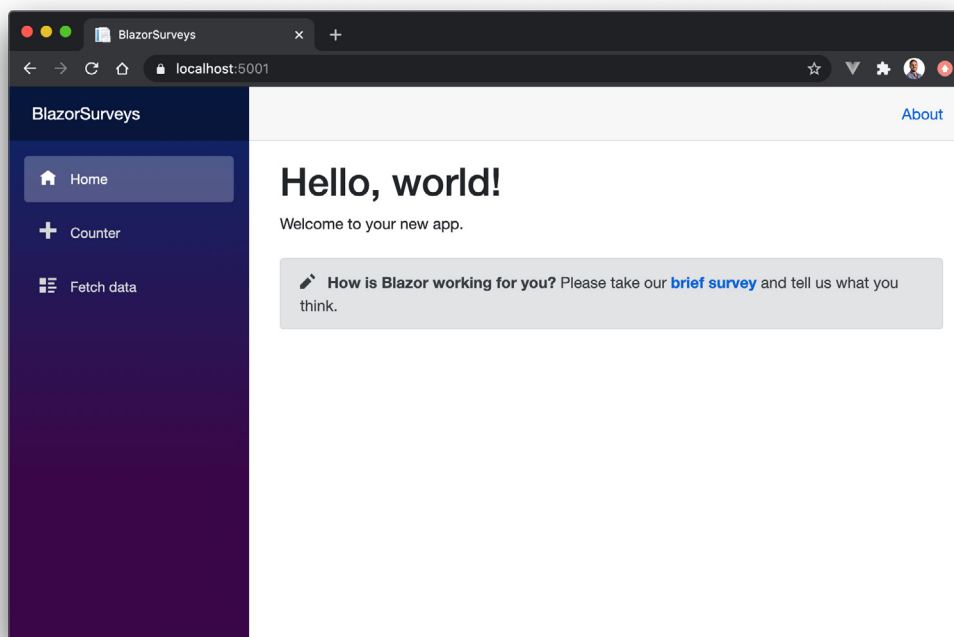


Figure 2, running the generated project greets you with its default home page

These commands just scratch the surface on how to run Blazor WebAssembly projects during development. For more information, and particularly if you want to learn about how to debug the Client project, check the [official documentation](#).

Note we use the Server project as the startup project. Blazor WebAssembly generates a set of static files, namely its DLLs and the necessary HTML/JS/CSS files to start the application, style it and interact with browser APIs. Browsers need to load these static files from a web server. Since we chose the *hosted model* of the WebAssembly template, the Server project has a double purpose in our solution:

- It provides the ASP.NET Core web API and SignalR hubs that the Client application interacts with.
- It also hosts the static files resulting from the Client application, including the index HTML page loaded at the root path as in <https://localhost:5001>, and the HTML/JS files referenced by the index page.

For a deeper discussion on how Blazor WebAssembly works and its startup process, check out my [previous article \(Blazor – Getting Started\)](#) in the DotNetCurry magazine.

Defining the application

The application we are going to build in this exercise is a web application where users can create and participate in surveys. A survey is defined as a question, a list of options that can be chosen as the answer, and an expiration date.

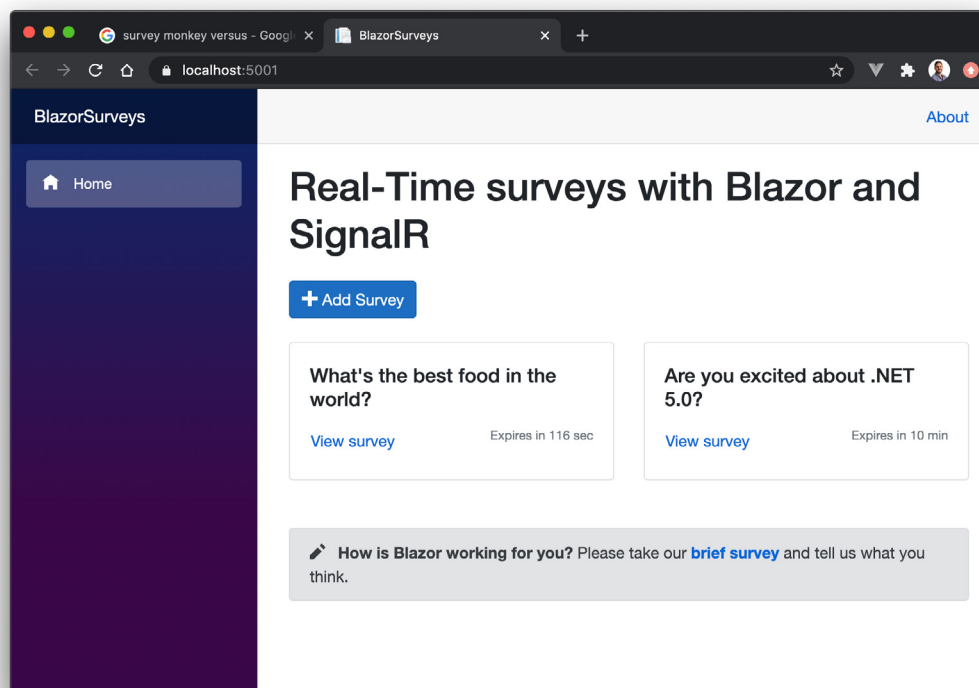


Figure 3, the home page provides a summary of all the surveys in the system

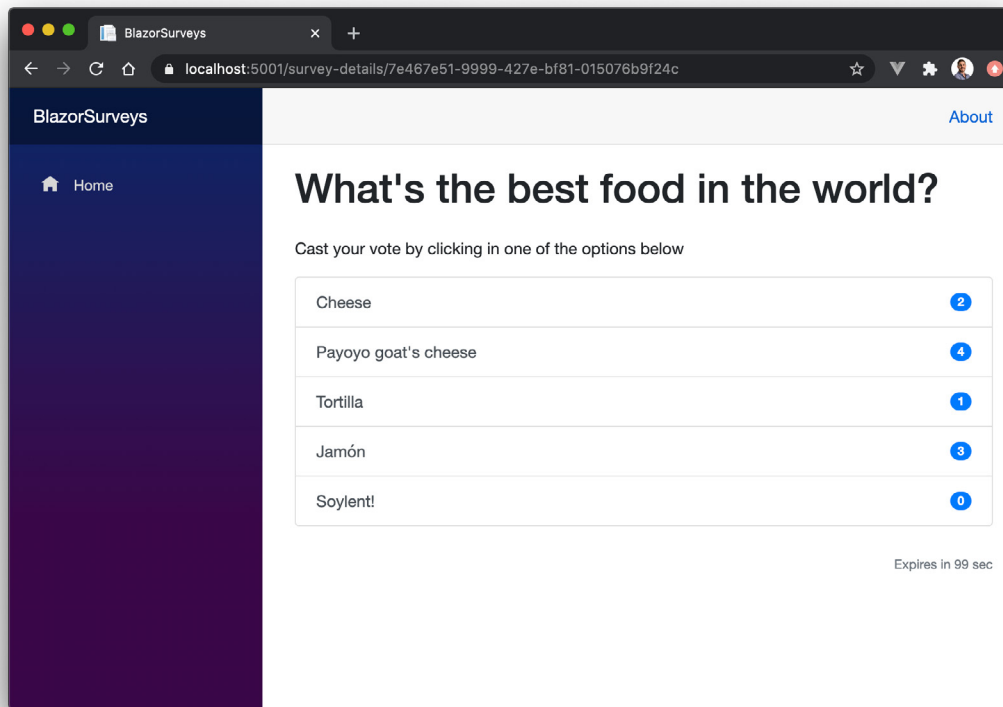


Figure 4, An example of one of the current surveys

Surveys can be a very complex field, with entire companies like SurveyMonkey and Qualtrics dedicated to it. Let's define a simple and manageable scope for the purposes of this article.

Our aim will be to create an application where users can:

- See a list with all the surveys in the system
- Create a new survey
- See the status of a survey, including how many users selected each option
- Participate in any of the open surveys (i.e., not yet expired) by selecting or *voting* for one of its answers

The real time elements of the application will provide a little extra functionality for our users:

- When browsing the list of surveys, any newly created survey will immediately show up
- When viewing the status of a given survey, the number of votes for each option will be automatically updated as other users cast their vote

This should give us enough of a real-world application feeling while allowing us to focus on the elements we want to explore like Blazor and SignalR. Of course, you are welcome to implement any other features you might want or be interested into, like authentication, charts, multiple questions per survey, question types other than multiple choice, etc.

Adding the shared models

Now that we have a better idea on what we are trying to build, we can start writing some code. We will begin by shifting our attention to the `BlazorSurveys.Shared` project in the Shared folder.

A shared project between the client and server projects is a fitting place for code used by both client and server. Since we are using Blazor, we can now define model classes that will be:

- Used as input by the REST API provided by the server project
- Used as output by the same REST API
- Used as page/component models by the pages and components defined in the client project

The model classes provide a basic shared contract between the client and the server. It is one of the benefits you get out of the box when using Blazor, having strongly typed models shared by both your client and server code.

Note the models provide a *contract* in a looser way than alternatives like `gRPC`. We are only defining shared models, there is no shared definition of the methods or endpoints provided by the server REST API. This means the compiler will guarantee you only access valid properties, and these will have the expected types. However, the compiler won't be able to guarantee that you call the right HTTP endpoint, nor that you use the right type for its input(s) and output!

You can actually use `gRPC` within your Blazor WebAssembly application, with the same general limitations that using `gRPC` from the browser has. For more information on using `gRPC` from Blazor WebAssembly, check the [official documentation](#).

For a general introduction to `gRPC`, check my [previous article](#) (`gRPC` with ASP.NET Core 3.0) in the DotNetCurry magazine.

Exploring immutable record types as models

The new C# 9 record types are a very interesting addition, which lets us easily define immutable classes. These are like normal classes, except the compiler enforces their immutability. i.e., you cannot change the values of their properties once an instance is created. If you want to modify them in some way, you need to create a new instance.

Read more about [record types](#) from the C# main designer and program manager, Mads Torgersen.

While the benefits of immutability are outside the scope of this article, the new record types are a good candidate to implement read operations in our REST API like retrieving a list of surveys or the full details of a given survey. And since this is a Blazor application, they are also good candidates as the page models for read-only pages such as the survey listing or survey details.

Let's define the survey model as an immutable record type. Add a new file to the Shared project and define the following types:

```

public record Survey
{
    public Guid Id { get; init; } = Guid.NewGuid();
    public string Title { get; init; }
    public DateTime ExpiresAt { get; init; }
    public List<string> Options { get; init; } = new List<string>();
    public List<SurveyAnswer> Answers { get;init; } = new List<SurveyAnswer>();
}

public record SurveyAnswer
{
    public Guid Id { get; init; } = Guid.NewGuid();
    public Guid SurveyId { get; init; }
    public string Option { get; init; }
}

```

As you can see, defining a model using a record type is not much different from using a normal class. However, the compiler will ensure the values are never modified after initialization.

Now let's do the same, providing a simpler version of the survey model. We will use this model when listing all the existing surveys, rather than using the full survey model with all of its properties.

```

public record SurveySummary
{
    public Guid Id { get; init; }
    public string Title { get; init; }
    public DateTime ExpiresAt { get; init; }
    public List<string> Options { get; init; }
}

```

Our API controller will have to convert between the Survey and the SurveySummary models. To keep things simple, add a mapping method like the following to the Survey record:

```

public SurveySummary ToSummary() => new SurveySummary{
    Id = this.Id,
    Title = this.Title,
    Options = this.Options,
    ExpiresAt = this.ExpiresAt
};

```

Mutable models and client-side form binding

The final model we need to create will be used for the *add survey* page and API. Rather than defining the model using a record type, we will use a standard class.

This will allow us to use the [form bindings](#) that Blazor provides out of the box, so we can bind specific properties of this model to HTML form inputs. If we were to use a record type and associate one of its properties with an HTML input field (like a textbox), we would get an Exception every time the user modified the value.

In addition, this lets us tailor the model to the UX for adding a survey. For example, we can ask users to define the survey duration in minutes, rather than asking them to enter a DateTime for the expiration date.

Add to the Shared project a new `AddSurveyModel` class like the following one:

```

public class AddSurveyModel
{
    public string Title { get; set; }
    public int? Minutes { get; set; }
    public List<OptionCreateModel> Options { get; init; } = new
List<OptionCreateModel>();

    public void RemoveOption(OptionCreateModel option) => this.Options.
Remove(option);
    public void AddOption() => this.Options.Add(new OptionCreateModel());
}

public class OptionCreateModel
{
    public string OptionValue { get; set; }
}

```

This should look very familiar if you have ever created a model for an ASP.NET Core web API or MVC controller. However, you might be missing some validation attributes. Don't worry we will come back to forms validation later in the article.

Note the **main limitation of record types** prevents us from using them **together with client-side form binding**. We could actually define a record type and use it in the server-side controller, while the client-side page could use its own mutable model class bound to the form fields.

This would allow us to keep the API and server-side immutable, while getting the benefit of client-side form binding. The downside would be having to maintain two models rather than a single one, even if the page model could be private to the Blazor component.

For the purposes of this application, I find this would be an overkill. However, in a larger application you might be interested in keeping the API and your controller using immutable record types. Even in the client-side you might want to adopt a read only immutable store using the Flux pattern and a library like [Fluxor](#).

Creating the Server application

In this section, we will update the generated server project, so it implements both a standard REST API and a SignalR hub for real time functionality.

If you have trouble or if you are already familiar with server side ASP.NET and SignalR, feel free to download the source code from [Github](#).

Adding the REST API

Now that we have our models defined, let's implement a very simple `SurveyController` class that provides the necessary REST API to manage the list of surveys and their answers. This is no different from any other API implemented with ASP.NET Core, so I won't spend much time on it. If you are new to it, check existing resources like this tutorial from the [official docs](#).

We will add a standard web API controller class, decorated with the `[ApiController]` attribute and inheriting from `ControllerBase`. For the purposes of the article, it will contain an in-memory list of

surveys rather than a connection to a database. Feel free to replace this with a persistent database, for example using Entity Framework Core as in the tutorial linked above.

```
[ApiController]
[Route("api/[controller]")]
public class SurveyController: ControllerBase
{
    private static ConcurrentBag<Survey> surveys = new ConcurrentBag<Survey> {
        // feel free to initialize here some sample surveys like:
        new Survey {
            Id = Guid.Parse("b00c58c0-df00-49ac-ae85-0a135f75e01b"),
            Title = "Are you excited about .NET 5.0?",
            ExpiresAt = DateTime.Now.AddMinutes(10),
            Options = new List<string>{ "Yes", "Nope", "meh" },
            Answers = new List<SurveyAnswer>{
                new SurveyAnswer { Option = "Yes" },
                new SurveyAnswer { Option = "Yes" },
                new SurveyAnswer { Option = "Yes" },
                new SurveyAnswer { Option = "Nope" },
                new SurveyAnswer { Option = "meh" }
            }
        },
        new Survey { ... omitted ... },
    };
}
```

The controller needs to implement the following methods, each exposing its own HTTP endpoint that the client can send a request to:

- getting a summary of all the surveys, exposed as GET /api/survey
- getting the full details of a single survey, exposed as GET /api/survey/{id}
- adding a new survey, exposed as PUT /api/survey/{id}
- answering a survey, exposed as POST /api/survey/{id}

The implementation can be as simple as:

```
[HttpGet()]
public IEnumerable<SurveySummary> GetSurveys()
{
    return surveys.Select(s => s.ToSummary());
}

[HttpGet("{id}")]
public ActionResult GetSurvey(Guid id)
{
    var survey = surveys.SingleOrDefault(t => t.Id == id);
    if (survey == null) return NotFound();
    return new JsonResult(survey);
}

[HttpPut()]
public async Task<Survey> AddSurvey([FromBody]AddSurveyModel addSurveyModel)
{
    var survey = new Survey{
```

```

        Title = addSurveyModel.Title,
        ExpiresAt = DateTime.Now.AddMinutes(addSurveyModel.Minutes.Value),
        Options = addSurveyModel.Options.Select(o => o.OptionValue).ToList()
    };
    surveys.Add(survey);
    return survey;
}

[HttpPost("{surveyId}/answer")]
public async Task<ActionResult> AnswerSurvey(Guid surveyId, [FromBody]SurveyAnswer
answer)
{
    var survey = surveys.SingleOrDefault(t => t.Id == surveyId);
    if (survey == null) return NotFound();
    // WARNING: this isn't thread safe since we store answers in a List!
    survey.Answers.Add(new SurveyAnswer{
        SurveyId = surveyId,
        Option = answer.Option
    });
    return new JsonResult(survey);
}

```

You should now have a functional API. If you want to make sure it works as expected, you can build and run the application, then test the API using a tool such as [Postman](#). See the official docs for examples on testing different methods like [Get](#) or [Put](#).

Defining the SignalR Hub

Let's now update the server project with a SignalR hub. As with the API controller before, since this is a standard Hub, I won't spend much time on it. If you want to read more, check the [official docs](#).

Add a new **Hubs** folder and create a new **SurveyHub.cs** file inside. Inside, create a new **ISurveyHub** interface. This interface defines the methods that our server-side controller will call, and the client-side will listen to. In our case, we want to notify clients when a survey is either added or updated:

```

public interface ISurveyHub
{
    Task SurveyAdded(SurveySummary survey);
    Task SurveyUpdated(Survey survey);
}

```

Next create a **SurveyHub** class that inherits from the base SignalR's **Hub** class. Note we don't have to implement the **ISurveyHub** interface, the interface is only necessary to provide a strongly typed interface to our server-side code.

```

public class SurveyHub: Hub<ISurveyHub>
{
}

```

Let's make our hub a little bit more interesting by defining two methods that clients can *invoke* rather than *listen* to. We will use them so clients can tell the server when they are viewing a particular survey. The server will subscribe those clients to the update events for that particular survey.

```

public async Task JoinSurveyGroup(Guid surveyId)
{
    await Groups.AddToGroupAsync(Context.ConnectionId, surveyId.ToString());
}
public async Task LeaveSurveyGroup(Guid surveyId)
{
    await Groups.RemoveFromGroupAsync(Context.ConnectionId, surveyId.ToString());
}

```

This will let us send the survey updated events exclusively to clients that are currently viewing that survey. When implementing the client, we just need to make sure the client-side code invokes these two hub methods when entering/leaving the survey details page.

Let's finish setting up the Hub by adding the necessary services and endpoint to the server application.

Update the `ConfigureServices` method of the `Startup` class to include the SignalR services:

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

..and update the `Configure` method, mapping the Hub to the `/surveyhub` endpoint:

```

app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
    endpoints.MapControllers();
    endpoints.MapHub<SurveyHub>("/surveyhub");
    endpoints.MapFallbackToFile("index.html");
});

```

Before moving on to the next section, check the [official docs](#) on how to enable response compression for the octet-stream MIME type, which will result in smaller SignalR messages sent between your client and server.

Emitting SignalR events from the controller

In order to emit events from the controller, we can use an instance of `IHubContext<SurveyHub, ISurveyHub>`. This will let us call the events we defined in the `ISurveyHub` interface, which SignalR will then propagate to any active clients.

Update the `SurveyController` class so an instance of the Hub context is injected via its constructor:

```

public class SurveyController: ControllerBase
{
    private readonly IHubContext<SurveyHub, ISurveyHub> hubContext;

    ...

    public SurveyController(IHubContext<SurveyHub, ISurveyHub> surveyHub)
    {
        this.hubContext = surveyHub;
    }
}

```

Then modify the `AddSurvey` method so an event is sent to the clients. We can do so through the `hubContext.Clients.All` property, which will expose the methods we defined in `ISurveyHub`:

```
public async Task<Survey> AddSurvey([FromBody]AddSurveyModel addSurveyModel)
{
    var survey = new Survey{ ... omitted ... };
    surveys.Add(survey);
    await this.hubContext.Clients.All.SurveyAdded(survey.ToSummary());
    return survey;
}
```

Now let's also send an event whenever an answer is added to a survey. However rather than sending the event to all the connected clients, we will only send it to the clients that are currently viewing that survey. These are the clients who joined the SignalR group with the group name matching the survey Id:

```
public async Task<ActionResult> AnswerSurvey(Guid surveyId, [FromBody]SurveyAnswer
answer) {
    // ... omitted ...

    await this.hubContext.Clients
        .Group(surveyId.ToString())
        .SurveyUpdated(survey);
    return new JsonResult(answer);
}
```

With these changes, we have all the main elements of our server-side and shared code ready!

Creating the client application

It is time to switch focus to the client side and Blazor WebAssembly. For many, this will be the most interesting section of the article!

The final source code is available in [Github](#).

Defining a strongly typed HTTP client

At the beginning of the article, we created a few model classes and records which would be shared between the client and server. As discussed, this would give us strongly typed models shared between the client and server projects. However, it would be up to the developer to keep the client-side code calling the REST API methods in sync with its implementation in the `SurveyController`.

We can improve the situation by creating a `strongly typed HttpClient` that encapsulates the REST API methods. This way we ensure there is a single class that needs to be manually kept in sync with changes to the interface of the REST API.

Add a new file `SurveyHttpClient` to the `Shared` project. Arguably, you could add this to the Client project as well. I like the idea of encapsulating the API as a strongly typed class that any .NET client could use, not just the Blazor application. Almost as if we were defining an `HTTP interface`!

There, implement a class that encapsulates the usage of an `HttpClient` calling each of the REST API methods provided by the `SurveyController`:

```

public class SurveyHttpClient
{
    private readonly HttpClient http;

    public SurveyHttpClient(HttpClient http)
    {
        this.http = http;
    }

    public async Task<SurveySummary[]> GetSurveys()
    {
        return await this.http.GetFromJsonAsync<SurveySummary[]>("api/survey");
    }

    public async Task<Survey> GetSurvey(Guid surveyId)
    {
        return await this.http.GetFromJsonAsync<Survey>($"api/survey/{surveyId}");
    }

    public async Task<HttpResponseMessage> AddSurvey(AddSurveyModel survey)
    {
        return await this.http.PutAsJsonAsync<AddSurveyModel>("api/survey",
survey);
    }

    public async Task<HttpResponseMessage> AnswerSurvey(Guid surveyId, SurveyAnswer
answer)
    {
        return await this.http.PostAsJsonAsync<SurveyAnswer>($"api/survey/
{surveyId}/answer", answer);
    }
}

```

We have essentially created a simple library that can be used by any piece of .NET code that wants to interact with our REST API. This includes our Blazor WebAssembly project.

As an added benefit, if you ever make a breaking change to the REST API, you won't need to scan your Blazor application for usages of the `HttpClient` which call the method you modified!

To use it in the client project, you first need to install some extension methods via the following NuGet package:

```

cd Client
dotnet add package Microsoft.Extensions.Http --version 5.0.0

```

Then update the `Main` method of the `Program` class. We need to register the typed `HttpClient` as part of the services, so we can later inject it into any razor component.

```

var baseAddress = new Uri(builder.HostEnvironment.BaseAddress);
builder.Services.AddScoped(sp => new HttpClient { BaseAddress = baseAddress });
builder.Services.AddHttpClient<SurveyHttpClient>(client => client.BaseAddress =
baseAddress);

```

You can now inject an instance of the strongly typed `HttpClient` into any of the Razor components of the client project, via the standard `@inject` directive. For example, a given Razor component could load the list of surveys from the server using:


```
@inject SurveyHttpClient SurveyHttpClient
```

... template omitted ...

```
@code {  
    private SurveySummary[] surveys;  
    protected override async Task OnInitializedAsync()  
    {  
        surveys = await SurveyHttpClient.GetSurveys();  
    }  
}
```

We will use this pattern multiple times through the rest of the article.

Note you might want to consider features like error handling as part of your centralized HTTP client code. I would suggest exploring a library like [Polly](#), which lets you implement policies like retries or circuit breakers.

Establishing the SignalR connection

There is one final bit of plumbing needed before we start building the client UX. We need to add the necessary code for the client to establish a SignalR connection to the Hub we defined in the server.

The first step is adding the SignalR client library to the **Client** project:

```
dotnet add package Microsoft.AspNetCore.SignalR.Client --version 5.0.0
```

Note this is the general SignalR client library for any .NET application, not just Blazor applications!

Now we need to add the code to initialize and establish the SignalR connection. This is achieved using the `HubConnectionBuilder` class to configure the connection, and calling its `StartAsync` method.

We need to be careful not to block the initialization of the Blazor app until the SignalR connection is established. If we did so, users would see the blank loading page until the SignalR connection is established. And they would get stuck in that loading page in case it is unable to establish the connection. Since the SignalR connection does not provide any crucial functionality, this would be too restrictive.

With this in mind, let's update the `Main` method of the `Program` class in order to register a singleton `HubConnection` instance. We will use the `HubConnectionBuilder` to create the connection, but we *won't start it yet*.

```
builder.Services.AddSingleton<HubConnection>(sp => {  
    var navigationManager = sp.GetRequiredService<NavigationManager>();  
    return new HubConnectionBuilder()  
        .WithUrl(navigationManager.ToAbsoluteUri("/surveyhub"))  
        .WithAutomaticReconnect()  
        .Build();  
});
```

Instead, we will start the connection as part of the **App.razor** component. In its most basic form, this would mean calling the `StartAsync` method of the connection as in:

```

@Inject HubConnection HubConnection
...
@code {
    protected override void OnInitialized()
    {
        HubConnection.StartAsync()
    }
}

```

This code is far too simplistic!

The first issue is obvious when you compile. You will get a warning since we are not awaiting an async method! To a certain extent, that's what we want - initialize in a fire and forget fashion that doesn't block initializing the rest of the application. However, the code can be more explicit about it and disable the warning on that line.

More important is what happens when the initialization fails or when the connection gets lost. As it stands, the code performs a single attempt to establish the connection.

In fact, the automatic reconnect configured as part of the `HubConnectionBuilder`:

- does not apply at all when initializing the connection
- will only retry a given number of times. After the last attempt, it will give up and is up to the application developer to write some logic that restarts the connection process.

We can improve this initial attempt by wrapping the `StartAsync` in a method that retries the initial connection attempt. Even better, we can handle the connection closed event and attempt the same connection initialization:

Add a new `@code` section to the existing `App.razor` component with the following contents:

```

private CancellationTokenSource cts = new CancellationTokenSource();

protected override void OnInitialized()
{
    // launch the signalR connection in the background.
    #pragma warning disable CS4014
    ConnectWithRetryAsync(cts.Token);

    // Once initialized the retry logic configured in the HubConnection will
    // automatically attempt to reconnect
    // However, once it reaches its maximum number of attempts, it will give up and
    // needs to be manually started again
    // handling this event we can manually attempt to reconnect
    HubConnection.Closed += error =>
    {
        return ConnectWithRetryAsync(cts.Token);
    };
}

private async Task<bool> ConnectWithRetryAsync(CancellationToken token)
{
    // Keep trying to until we can start or the token is canceled.
    while (true)

```

```

    {
        try
        {
            await HubConnection.StartAsync(token);
            return true;
        }
        catch when (token.IsCancellationRequested)
        {
            return false;
        }
        catch
        {
            // Try again in a few seconds. This could be an incremental interval
            await Task.Delay(5000);
        }
    }
}

public async ValueTask DisposeAsync()
{
    cts.Cancel();
    cts.Dispose();
    await HubConnection.DisposeAsync();
}

```

For more information, see the [official docs](#) on how to handle the lost connection.

Listing all the surveys in the Index page

Let's begin building the UX of our sample application. The first step will be replacing the existing `Index.razor` component with one that lists the surveys in the application.

The component has an array of `SurveySummary` which is retrieved from the server using the `SurveyHttpClient`. They are then rendered using a [bootstrap card](#) per survey:

```

@page "/"
@using BlazorSurveys.Shared
@inject SurveyHttpClient SurveyHttpClient
@inject NavigationManager NavigationManager

<h1 class="mb-4">Real-Time surveys with Blazor and SignalR</h1>

<button class="btn btn-primary mb-4" @onclick="AddSurvey"><i class="oi oi-plus" />
Add Survey</button>

@if (surveys == null)
{
    <p><em>Loading...</em></p>
} else {
    <div class="row row-cols-1 row-cols-md-2">
        @foreach (var survey in surveys.OrderBy(s => s.ExpiresAt))
        {
            <div class="col mb-4">
                <div class="card">
                    <div class="card-body">
                        <h5 class="card-title">@survey.Title</h5>

```

```

        <button class="btn btn-link pl-0" @onclick="@(() => GoToSurvey(survey.
Id))">View survey</button>
    </div>
</div>
</div>
}
</div>
}
@code {
    private SurveySummary[] surveys;

    protected override async Task OnInitializedAsync()
    {
        surveys = await SurveyHttpClient.GetSurveys();
    }

    private void AddSurvey()
    {
        NavigationManager.NavigateTo("add-survey");
    }

    private void GoToSurvey(Guid surveyId)
    {
        NavigationManager.NavigateTo($"survey-details/{surveyId}");
    }
}

```

Both the “Add survey” and “View survey” buttons have an event handler for their click event. These use the `NavigationManager` to navigate to pages we haven’t yet created, so they will fail if you click them! We will add these pages in the next section.

Listening to the survey added SignalR event

We now have a page that retrieves and displays all the surveys. Let’s enhance the page so it listens to the `SurveyAdded` SignalR event and automatically displays in real time any survey added by another user.

With all the plumbing we have done, this is a matter of calling the `On` method on the `HubConnection` class. The event handler can add the survey to the component’s current list and notify Blazor of the change via the `StateHasChanged` method. The only caveat is to remember cleaning our event handler when the component is removed!

```

@using Microsoft.AspNetCore.SignalR.Client
@Inject HubConnection HubConnection
@implements IDisposable

... omitted ...

protected override async Task OnInitializedAsync()
{
    surveys = await SurveyHttpClient.GetSurveys();
    HubConnection.On<SurveySummary>("SurveyAdded", survey =>
    {
        surveys = surveys
            .Where(s => s.Title != survey.Title)
            .Append(survey)
    }
    );
}

```

```

        .ToArray();
        StateHasChanged();
    });
}

public void Dispose()
{
    HubConnection.Remove("SurveyAdded");
}

```

As you can see, we add an event handler for the `SurveyAdded` event as part of the component's initialized lifecycle event. This event handler is removed as part of the `Dispose` method when the component is removed. The event handler receives an instance of the `SurveySummary` model, as we defined in the `ISurveyHub` previously in the server project.

Render a live expiration time using a shared component and a shared `IExpirable` interface

When we defined the survey model, we added an `ExpiresAt` property. It would be useful if we use this property to display how much time remains before the survey expires. Even better, it would be great if we could live update this information.

Let's define a new `IExpirable` interface in the shared project. This interface contains an `ExpiresAt` property, and we'll use it to define three calculated properties:

- `IsExpired`, returning true if the survey is already expired
- `ExpiresInMin`, returning the number of minutes from the current time until the defined expiration time
- `ExpiresInSec`, same as `ExpiresInMin` but in seconds

Nothing particularly remarkable, except for the fact the calculated properties are directly added to the interface taking advantage of a feature added in C# 8.

```

public interface IExpirable
{
    DateTime ExpiresAt { get; }
    bool IsExpired => DateTime.Now > ExpiresAt;
    int ExpiresInMin => (int)Math.Ceiling((decimal)ExpiresInSec / 60);
    int ExpiresInSec => (int)Math.Ceiling(ExpiresAt.Subtract(DateTime.Now).
TotalMilliseconds / 1000);
}

```

Now that we have this interface, make sure to update both the `SurveySummary` and `Survey` models to implement it. No other changes are needed in these record types, since they both already defined an `ExpiresAt` property.

You can go back to the `SurveyController` for a brief change and ensure that answers are only added to surveys that haven't expired. In the `AnswerSurvey` method, add the following guard:

```

if (((IExpirable)survey).IsExpired) return StatusCode(400, "This survey has expired");

```

Back to the client project, add a new **SurveyExpiration.razor** component. The component will receive an instance of **IExpirable** as parameter and will render how long until it expires.

```
@using BlazorSurveys.Shared
<p @attributes="ExtraAttributes">
  @if (Expirable.IsExpired){
    <strong>This survey has already expired!</strong>
  } else if(Expirable.ExpiresInMin > 2) {
    <small class="text-muted">Expires in @Expirable.ExpiresInMin min</small>
  } else {
    <small class="text-muted">Expires in @Expirable.ExpiresInSec sec</small>
  }
</p>

@code {
  [Parameter]
  public IExpirable Expirable { get; set; }
  [Parameter(CaptureUnmatchedValues = true)]
  public Dictionary<string, object> ExtraAttributes { get; set; }
}
```

Note the usage of the **ExtraAttributes**. This allows users of this component to provide their own HTML attributes like **class**, which will be added to the root **<p>** element of the component.

We can immediately take advantage of this component by using it from the **Index.razor** component. Update its template so the component is added at the end of the current **card-body** element:

```
<div class="card-body">
  ...
  <SurveyExpiration Expirable=survey class="card-text float-right" />
</div>
```

This is very clean and a great usage of the **IExpirable** interface across client and server code. The only problem is that the displayed expiration time isn't updated as time goes by. i.e., we won't see the number of minutes/seconds remaining decreasing in real time.

Let's fix that by adding an old-fashioned **Timer** to the **SurveyExpiration** component. The **Timer** will fire every second and will notify Blazor that it has to re-render the component because the state has changed. We are effectively calling **StateHasChanged** every second, forcing Blazor to check the new value of the interface calculated properties and re-render the UX.

```
@using System.Threading

... omitted ...

private Timer timer;

protected override void OnInitialized()
{
  timer = new Timer((object stateInfo) =>
  {
    StateHasChanged();
  }, null, TimeSpan.FromSeconds(1), TimeSpan.FromSeconds(1));
}
```

```
public async ValueTask DisposeAsync()
{
    await timer.DisposeAsync();
}
```

If you now run the application, you will see how the time remaining is live updated until the survey eventually expires.

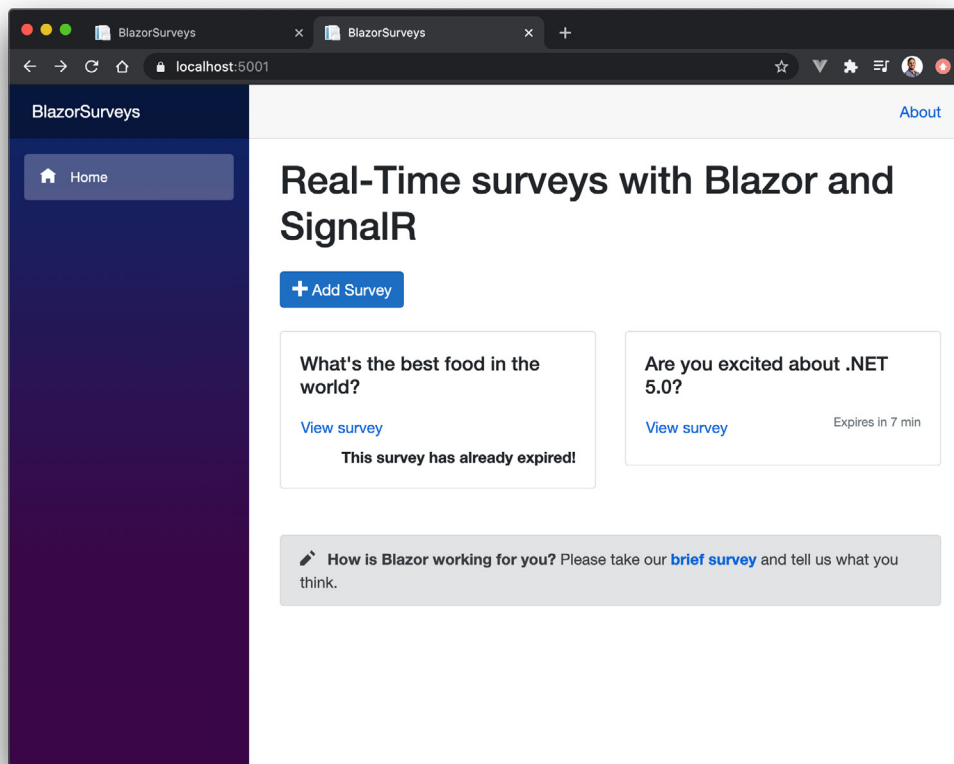


Figure 5, survey expiration time displayed and updated in real time

Answering a survey

To participate on a survey, we need to build a page that shows the full details of a survey. At the very least, users need to see the available options so they can choose one of them.

Begin by adding a new **SurveyDetails.razor** page. This new page will receive the Id of the survey to display as a parameter from the URL, it will load it from the server using the REST API and will display its full details.

```
@page "/survey-details/{Id:guid}"
@using BlazorSurveys.Shared
@inject SurveyHttpClient SurveyHttpClient

@if (survey is null){
    <p><em>Loading...</em></p>
} else {
    <h1 class="mb-4">@survey.Title</h1>

    <p>Cast your vote by clicking in one of the options below</p>

    <ul class="list-group mb-4">
```

```

    @foreach (var option in survey.Options)
    {
        <button
            class="list-group-item list-group-item-action d-flex justify-content-between
align-items-center"
            >
            @option
            <span class="badge badge-primary badge-pill">
                @(survey.Answers.Where(a => a.Option == option).Count())
            </span>
        </button>
    }
</ul>

<SurveyExpiration Expirable=survey class="text-right" />
}

@code {
    [Parameter]
    public Guid Id { get; set; }
    private Survey survey;

    protected override async Task OnInitializedAsync()
    {
        survey = await SurveyHttpClient.GetSurvey(Id);
    }
}

```

It is not too different from the previous `Index` component; except we load a single `Survey` from the server rather than a list of `SurveySummary`. We then use a bootstrap's `list group` to render each option as an actionable item that users can click on.

Note how we also reuse the same `SurveyExpiration` component in order to show how long until the survey expires.

To vote on a survey, we will need to use the `AnswerSurvey` method of the `SurveyHttpClient`. Add an event handler to each option's button, passing along that particular option:

```
@onclick="@(async () => await OnAnswer(option))"
```

Inside the code directive, implement the `OnAnswer` method as:

```

private async Task OnAnswer(string option)
{
    if (((IExpirable)survey).IsExpired) return;
    await SurveyHttpClient.AnswerSurvey(Id, new SurveyAnswer{
        Option = option
    });
}

```

That's it, users can now click on one of the options and vote on it. Note users can keep voting as many times as they want! Feel free to implement some functionality to ensure users can only participate once per survey.

Listening to the survey updated SignalR event

Now that users can answer a survey, let's leverage our SignalR hard work to ensure users can see votes from other users in real time.

If you remember, the server is already sending a `SurveyUpdated` event whenever a new answer is added. We can then update the `SurveyDetails` component to subscribe to that event in a similar way the `Index` component subscribed to the `SurveyAdded` event.

The only caveat is that the event will only be sent to clients who registered themselves within the SignalR group for that particular survey. We then need to ensure the `SurveyDetails` page calls the `JoinSurveyGroup` method of the Hub when initialized, as well as the `LeaveSurveyGroup` method when disposed:

```
@using Microsoft.AspNetCore.SignalR.Client
@Inject HubConnection HubConnection
@implements IAsyncDisposable

... omitted ...

protected override async Task OnInitializedAsync()
{
    survey = await SurveyHttpClient.GetSurvey(Id);

    // TODO: error handling, for example when not connected to the server
    await HubConnection.InvokeAsync("JoinSurveyGroup", Id);

    HubConnection.On<Survey>("SurveyUpdated", survey =>
    {
        this.survey = survey;
        StateHasChanged();
    });
}

public async ValueTask DisposeAsync()
{
    HubConnection.Remove("SurveyUpdated");
    // TODO: error handling, for example when not connected to the server
    await HubConnection.InvokeAsync("LeaveSurveyGroup", Id);
}
```

Once you have these changes, open the same survey in two separate browser windows. Note how answering in one of the windows is automatically reflected in the other.

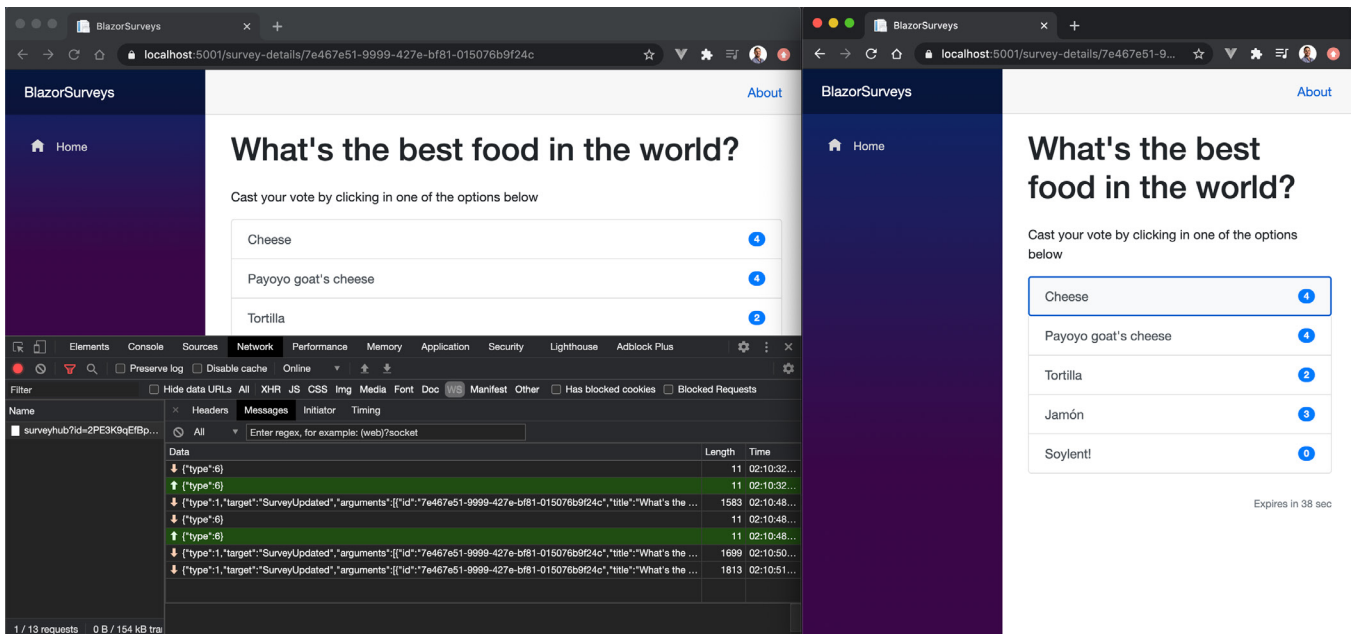


Figure 6, survey answers are propagated to clients in real time

Adding a new survey

It's time to add the last page of our application, one where users can define new surveys.

Add a new **AddSurvey.razor** page, where an instance of the **AddSurveyModel** is edited using Blazor data binding and its **forms components**. Once the form is submitted, the model will be sent to the server using the **AddSurvey** method provided by the **SurveyHttpClient**.

```
@page "/add-survey"
@using BlazorSurveys.Shared
@using Microsoft.AspNetCore.Components.Forms
@inject SurveyHttpClient SurveyHttpClient
@inject NavigationManager NavigationManager
```

```
<EditForm EditContext="@editContext" OnSubmit="@OnSubmit">
  <DataAnnotationsValidator />
  <ValidationSummary />

  <div class="form-group">
    <label for="inputTitle">Title</label>
    <InputText id="inputTitle" class="form-control" @bind-Value="survey.Title" />
  </div>

  <div class="form-group">
    <label for="inputMinutes">Minutes</label>
    <InputNumber id="inputMinutes" class="form-control" @bind-Value="survey.
Minutes" />
  </div>
```

```

<label>Options</label>
@foreach (var option in survey.Options)
{
    <div class="input-group mb-3">
        <InputText class="form-control" @bind-Value="option.OptionValue" />
        <div class="input-group-append">
            <button class="btn btn-outline-primary" type="button" @onclick="@(() =>
survey.RemoveOption(option))">Remove</button>
        </div>
    </div>
}
<p>
    <button class="btn btn-primary" type="button" @onclick="@(() => survey.
AddOption())">
        <i class="oi oi-plus" /> Add Option
    </button>
</p>

<p>
    <button type="submit" class="btn btn-primary float-right">Submit</button>
</p>
</EditForm>

@code {
    private AddSurveyModel survey = new AddSurveyModel();
    private EditContext editContext;

    protected override void OnInitialized()
    {
        editContext = new EditContext(survey);
    }

    private async Task OnSubmit()
    {
        if (!editContext.Validate()) return;

        var response = await SurveyHttpClient.AddSurvey(survey);
        if (response.IsSuccessStatusCode)
        {
            NavigationManager.NavigateTo("");
        }
    }
}

```

You should now be able to add a new survey. And with all the infrastructure we have already added, the **SurveyAdded** event will be fired and received by any client currently viewing the list of surveys. Make sure you try adding a survey with two browser windows open, where one of them stays on the Index page!

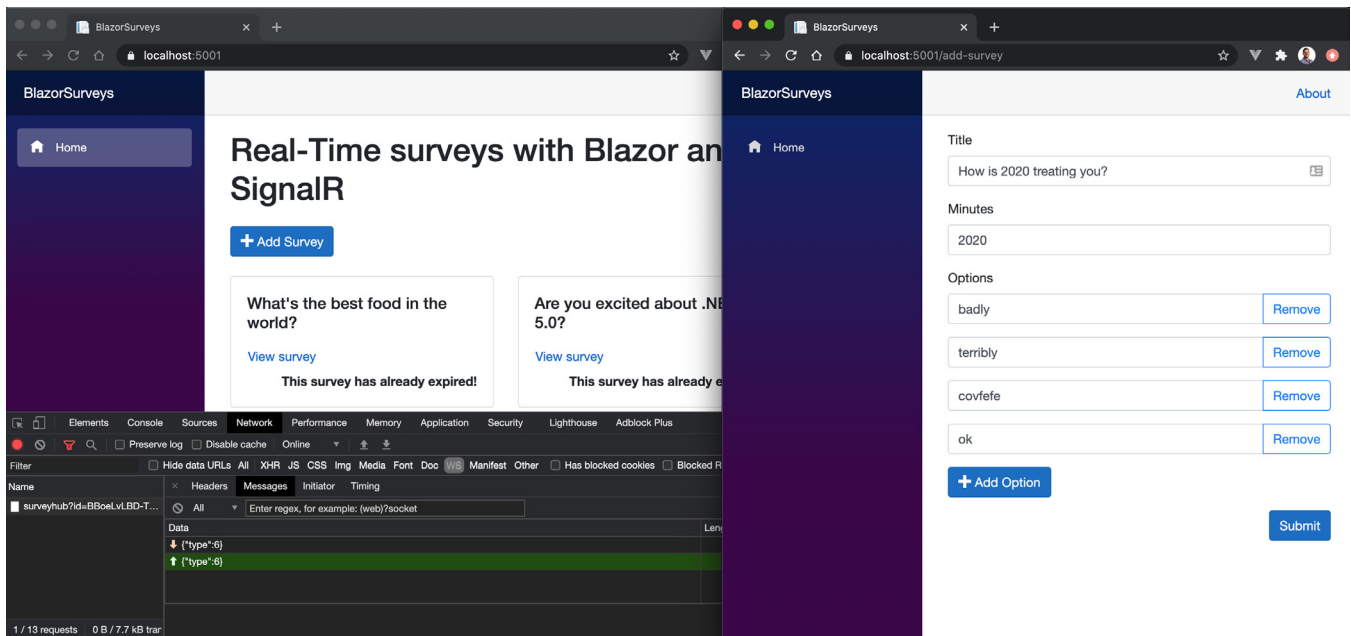


Figure 7, adding a new survey

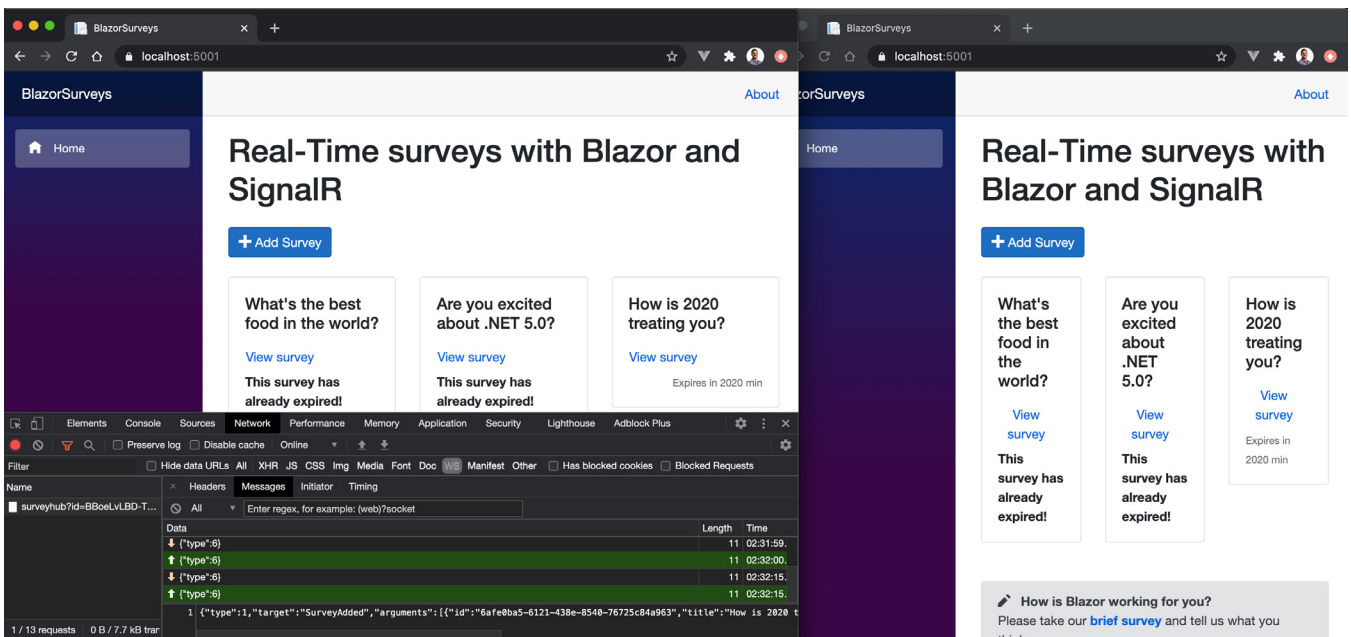


Figure 8, the new survey propagates to other clients in real time

Forms validation

Let's finish our application by updating and customizing the validation of the `AddSurveyModel`. To begin with, add some data annotation attributes like `[Required]` to the properties of the model class.

Once you have done so, let's add a *model rule*, rather than an *individual property rule*. Let's make sure that an `AddSurveyModel` is only considered valid if it has at least two options. This can be achieved by implementing the `IValidatableObject` interface:

```
public class AddSurveyModel: IValidatableObject
{
    [Required]
    [MaxLength(50)]
```

```

    public string Title { get; set; }

    ...

    public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext)
    {
        if (this.Options.Count < 2 )
        {
            yield return new ValidationResult("A survey requires at least 2
options.");
        }
    }
}

```

Now re-run the application and attempt to submit the form without entering any values. The `<ValidationSummary />` component will automatically display the failed validations, including the model level one. And the new survey won't be submitted to the server unless it's valid, due to the `if (!editContext.Validate()) return;` guard inside the `OnSubmit` method.

The `SurveyController` automatically implements server-side validation since it is decorated with the `[ApiController]` attribute. You can test this by commenting out the client-side guard and submitting an invalid model. You will notice the server replies with a 400 Invalid response, including the details of the failed validations.

This is great, we have both client and server-side validation completely in sync with very little effort!

Validation in nested models

However, the Blazor implementation is limited to top-level properties in the model. This means adding a `[Required]` property to the `OptionCreateModel` has no effect in our client-side validation!

In order to validate nested models client side, we need to install another NuGet package which is still in the experimental phase. Install it in both the **Client** and **Shared** projects:

```
dotnet add package Microsoft.AspNetCore.Components.DataAnnotations.Validation
--version 3.2.0-rc1.20223.4
```

You then have to annotate the nested model with the `[ValidateComplexType]` annotation. In our case, this means adding the annotation to the `Options` list:

```
[ValidateComplexType]
public List<OptionCreateModel> Options { get; init; } = new
List<OptionCreateModel>();
```

Finally, replace the `<DataAnnotationsValidator />` component in the Razor component with `<ObjectGraphDataAnnotationsValidator />`.

For more information, see the [official docs](#).

Customizing validation styles

Since Blazor ships with Bootstrap wired out of the box, we can customize the validation component to use Bootstrap styling.

The very first thing you want to add is your own `FieldCssClassProvider`. We will use this class to specify which class names should be added to invalid HTML input elements. To use `bootstrap styles`, we want this to be the class name `is-invalid` when invalid.

```
private class BootstrapFieldClassProvider : FieldCssClassProvider
{
    public override string GetFieldCssClass(EditContext editContext, in
FieldIdentifier fieldIdentifier)
    {
        var isValid = !editContext.GetValidationMessages(fieldIdentifier).Any();
        return isValid ? "" : "is-invalid";
    }
}
```

This class is wired to the `editContext` as part of the `OnInitialized` method of the component:

```
editContext.SetFieldCssClassProvider(new BootstrapFieldClassProvider());
```

Then we can display each individual field error message right next to that field, as opposed to the validation summary displayed at the top of the page. This can be achieved using the `editContext.GetValidationMessages` method to check if there are failed validations for a specific field, and if so display them using the expected bootstrap style:

```
<div class="form-group">
  <label for="inputTitle">Title</label>
  <InputText id="inputTitle" class="form-control" @bind-Value="survey.Title" />
  @if(editContext.GetValidationMessages(() => survey.Title).Any()){
    <div class="invalid-feedback">
      @editContext.GetValidationMessages(() => survey.Title).First()
    </div>
  }
</div>
```

The problem is that this can be very verbose once you have a few fields in the form. You could explore component libraries or roll your own!

Finally, we can update the validation summary displayed at the top, so it uses a bootstrap `alert box`:

```
<div class="@editContext.GetValidationMessages().Any() ? "alert alert-danger pb-0" : """>
  <ValidationSummary class="alert alert-danger" />
</div>
```

You will also want to comment out the following rule inside `app.css` since it overrides the default bootstrap font color for alert boxes:

```
.validation-message {
  // color: red;
}
```

With these changes, your form validation will be styled using Bootstrap.

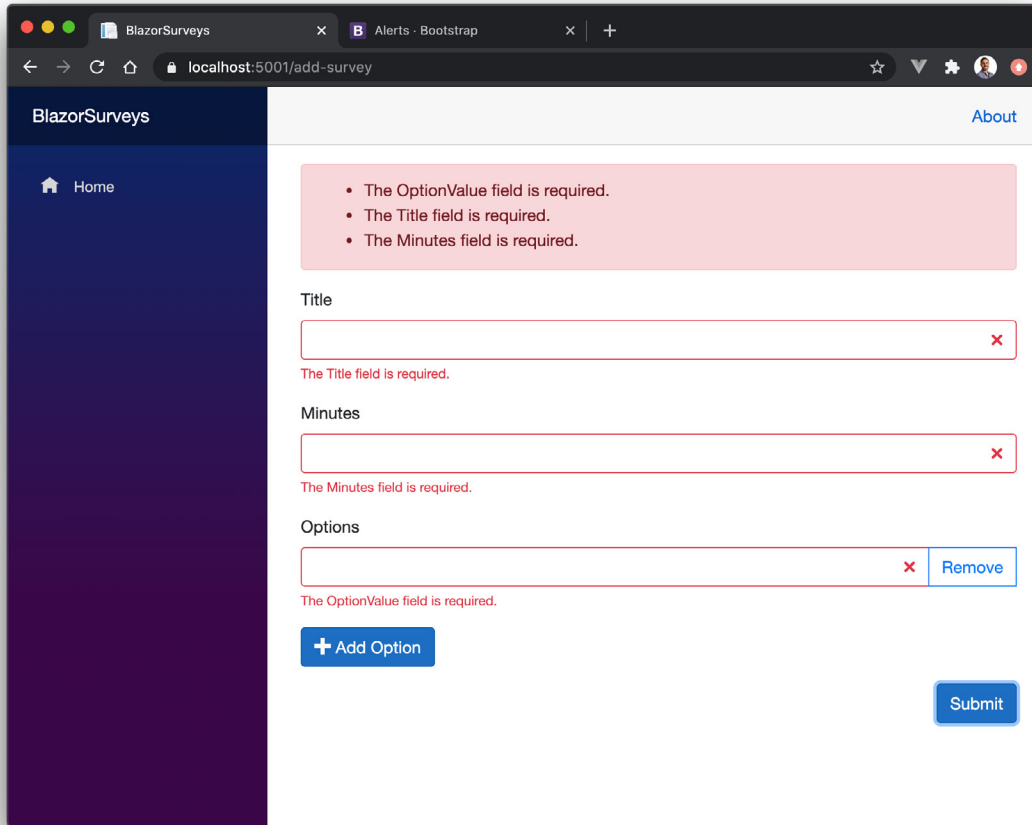


Figure 9, form validation using Bootstrap styling

Make sure to check the official docs on [forms validation!](#)

Conclusion

There is a lot to like in Blazor.

Enabling developers to create a web application with real time functionality using only .NET and C# can be a huge win for many teams and developers. Not just that, you get to leverage the best .NET framework with its latest .NET 5 release, as well as all the improvements made to the language up to C# 9.

Blazor can be a huge enabler for all of those who avoided the JavaScript ecosystem and all the nuances of modern web development. However, if you bit the JavaScript bullet a while ago and invested in the current web development skills and tools, its benefits might be less obvious.

I am really curious to see if Blazor can become an alternative to JavaScript with widespread appeal. In my opinion, it is *almost* there. I think it needs a little bit more time to receive features I now take for granted, to improve its performance and to grow an ecosystem of useful libraries and components.

One of the main features I miss while developing is stateful hot reload. That very short feedback loop as I change the code, is something I find very hard to renounce. Note this has been known since the early days of Blazor as you can see in [its issue](#), hopefully it will make the cut for the .NET 6 planning currently [happening in the open](#). Hopefully this will land together with [AoT compilation](#) for better performance and

even source maps for easier debugging!

I am also aware of how much I came to rely on flux data stores like [Vuex](#). I have used it in all the latest web applications I created using Vue.js and I found it the key ingredient to manage complex apps with many components. I should start exploring Blazor alternatives like [Fluxor](#)!

Finally, although the JavaScript ecosystem can be daunting to get into, once you become familiar with it you realize there are many truly useful tools and libraries. Blazor hasn't had the time nor the developer adoption to get there *yet*. I know you can interop from Blazor to JavaScript, which means you *could* use libraries from the JavaScript ecosystem. However, the Blazor magic breaks the moment you start adding JavaScript code and libraries into your Blazor application.

I understand these are very subjective points. What for me is a major pain point might be not so much for others. It's also likely the level of comfort with today's JavaScript and web development will drive much of your position towards Blazor. The good news is that most of these issues are solvable by either Microsoft or preferably by growing a community around Blazor.

And it's the latter what will ultimately determine the fate of Blazor!



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.



Technical Review

Damir Arh



Editorial Review

Suprotim Agarwal



et curry.com

**Want this
magazine
delivered
to your inbox ?**

Subscribe here

www.dotnetcurry.com/magazine/

* No spam policy



Darren Gillis

Diving into Azure Data Studio



In this article, I will give a brief history of the evolution of Azure Data Studio (ADS). I will also provide an overview of the features currently included in ADS and a breakdown of who the target users are and why one might want to choose ADS over SQL Server Management Studio (SSMS). I will also examine why existing SSMS users may want to add ADS to their arsenal of data administration and development tools.

Background

SQL Server Management Studio (SSMS) has been a mainstay of the SQL Server ecosystem as far back as SQL Server 2005. While the strong feature set and robust capabilities have helped DBAs and developers alike for the better part of 15 years, there remained a need for a cross-platform solution to allow additional accessibility to a new generation of developers, database administrators, data analysts, and data architects alike.

Enter **Azure Data Studio (ADS)**. Launched as an open-source initiative, ADS first appeared as “SQL Operations Studio” with pre-release public previews being available in late 2017. With the general availability release launched in September 2018, the product was renamed to what we know today as Azure Data Studio.

While relatively new, ADS is starting to mature into a quality addition to cross-platform tooling following closely in the footsteps of its cousin, Visual Studio Code. ADS is a downstream fork of VS Code and is merged regularly. For avid users of SSMS, ADS may not look like much of a competitor; however, there are some features unique to ADS that even ardent users of SSMS will come to appreciate. If the acknowledgment or awareness of ADS from SSMS users has been lacking, this will likely change since starting with the latest release of SSMS (18.7.1 on October 27, 2020), ADS will be included with the SSMS installation and installed alongside SSMS forthgoing.

The early iterations of ADS targeted SQL Server specifically. However, additional support for other databases has steadily increased as the product continues to evolve. Currently, ADS includes support for SQL Server, Azure SQL, Apache Spark, and Hive. Additional support for open-source databases is included through an extension such as PostgreSQL with pending support for MySQL currently on the roadmap.



Target Audience

The capabilities that were included with the early preview releases of Azure Data Studio were targeted at Database Administrators, Database Developers, Application Developers, and Data Analysts. With the release of SQL Server 2019 and the added support for Apache Spark and the Hadoop Distributed File System, a new architecture was born that combined these technologies into a platform known as “SQL Server Big Data Cluster”.

It was at this point where the ADS team began to consider the potential for additional support and capabilities to include user roles such as Data Engineers and Data Scientists. Interactive programming is common practice within these data roles, and the usage of live coding documents such as Jupyter Notebooks has gained widespread adoption in the data science community. As we will see later in this article, the introduction of Jupyter Notebooks support directly in ADS, has opened the tool to a wider audience across all the major OS platforms.

Relationship to SQL Server Management Studio

The developer DNA for Azure Data Studio comes directly from SQL Server Management Studio as the same team at Microsoft is responsible for both products. With the release of SQL Server 2017 and the support for SQL Server on Linux and containers, it was clear to the data team at Microsoft that a cross-platform option for SSMS features was required to provide access to developers that are heavily using platforms such as Linux, containers, and Mac OS. It was this cross-platform support that was the main catalyst to beginning work on ADS as the cross-platform option.

Quick Tour and Features

Upon launching Azure Data Studio, users of Visual Studio Code will immediately notice the resemblance and similarities.

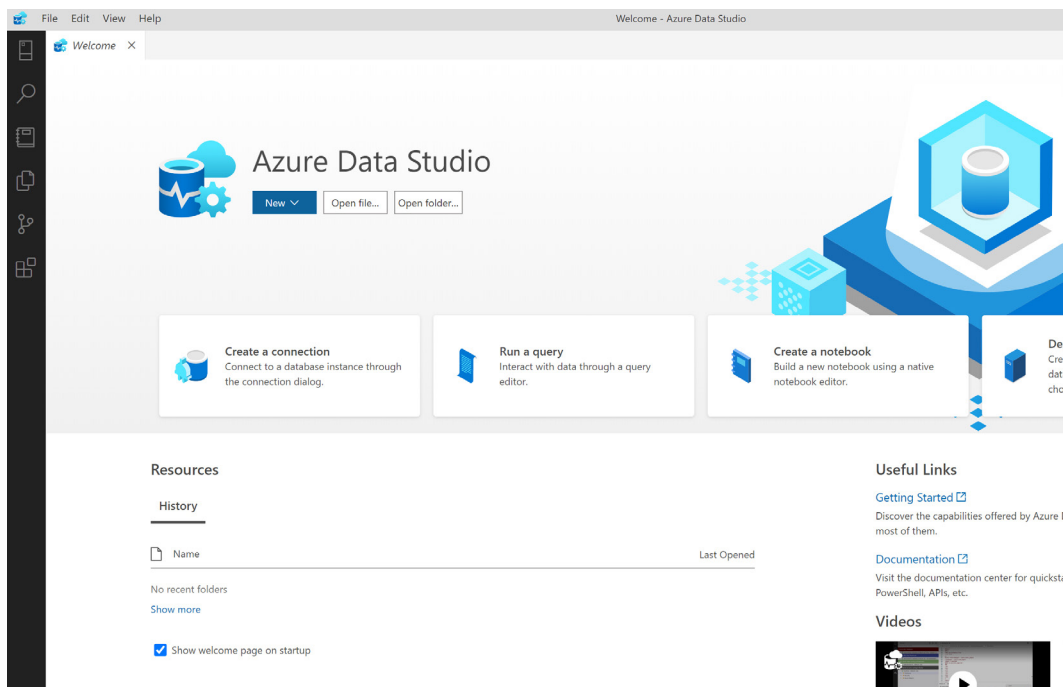


Figure 1: ADS Welcome Screen

ADS is a downstream fork of VS Code and is merged regularly. The version number of ADS as well as VS Code are both visible in the About > Help menu.

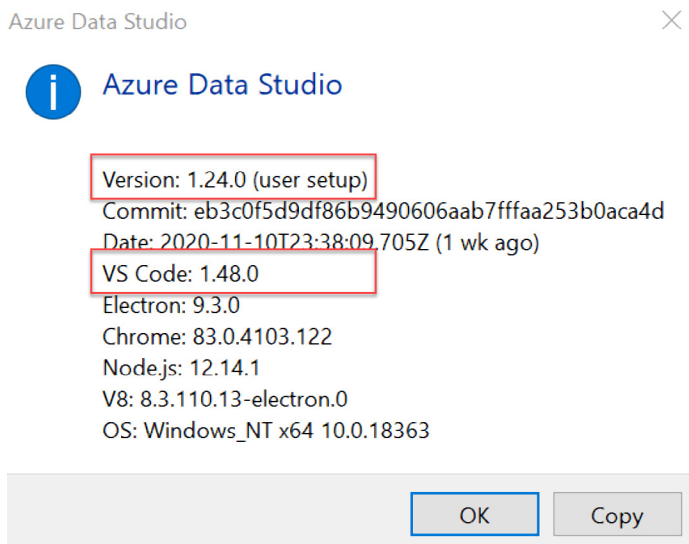


Figure2: ADS Help > About – Version highlights

Connections

Like any data tool, before data can be accessed, a connection to the relevant data source needs to be established. Azure Data Studio has many options for creating and organizing collections and supports a growing number of data sources. Current support includes SQL Server (on-prem and Azure SQL) Azure Synapse Analytics, and PostgreSQL support is available via extension (Extensions will be discussed later in this article).

Support for MySQL is expected to come via extension in an upcoming release although there is no definitive date at the time of this writing.

Connections are accessed via the “Connections” option in the vertical toolbar:

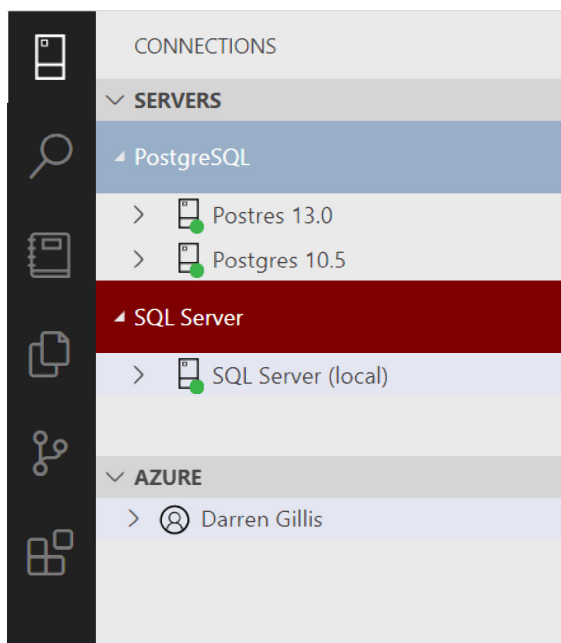


Figure 3: Main Connections section

In addition to the “SERVERS” view which offers the traditional method of setting up a data connection, you will notice additional views including “AZURE” and “SQL SERVER BIG DATA CLUSTERS” that offer the ability to connect to existing Azure accounts and SQL Big Data Clusters, respectively.

The colored groupings are fully customizable and configurable. Server groups are created using the “New Server Group” option:

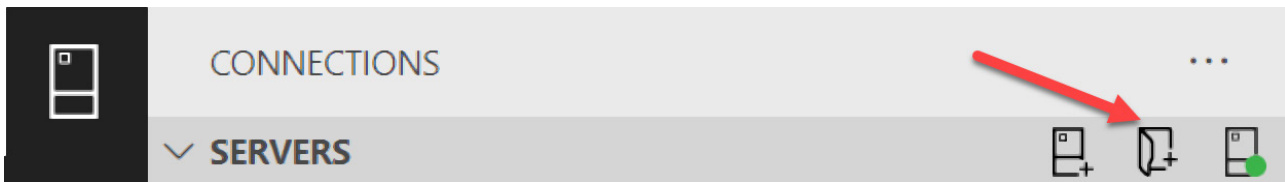


Figure 4: New Server Group option

Adding a Connection

Adding a connection is straight-forward and should be familiar to creating connections in other data tools, such as SSMS.

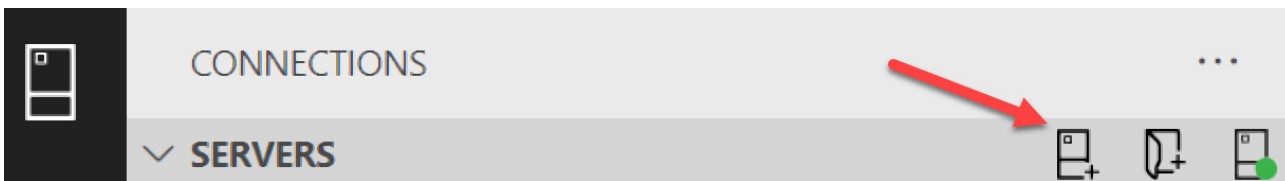


Figure 5: New Connection

The connection details present options you would expect, including “Connection type” in which the options may vary depending on any extensions that are installed. “PostgreSQL”, for instance is available with the installation of the PostgreSQL extension. You can assign the connection to any previously created server group by selecting from the “Server group” dropdown. The “Advanced” options include additional settings used to configure the connection further with settings including port number, connection pooling, timeout, encryption, etc.

Connection Details	
Connection type	Microsoft SQL Server
Server	localhost
Authentication type	Windows Authentication
User name	
Password	
	<input type="checkbox"/> Remember password
Database	WideWorldImporters
Server group	SQL Server
Name (optional)	
	Advanced...

Figure 6: Connection Details

Connections to Azure SQL can be configured either through the add connections dialog – particularly useful to include relevant connections in a server group collection. Alternatively, the option to directly connect to Azure is available and any relevant Azure SQL databases are easily accessible.

Exploring Databases

The object explorer is used to display a tree view of a server's databases and associated objects including tables, stored procedures, views, etc. This view should be familiar to users of SSMS or Server Explorer in Visual Studio.

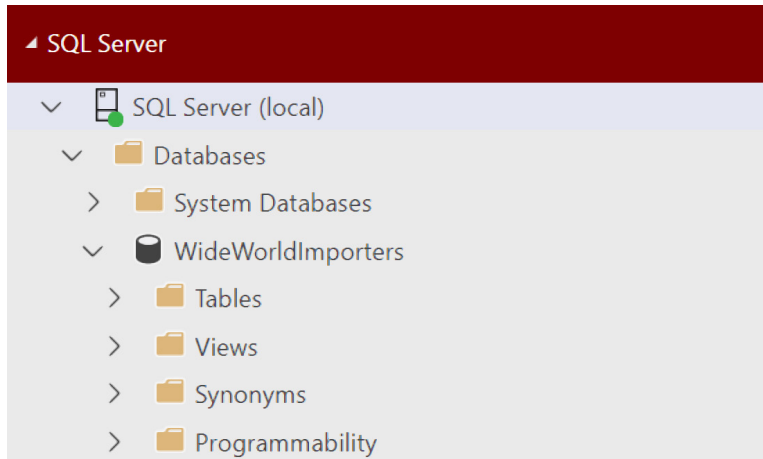


Figure 7: Exploring Database

When using the menus and options to work with the objects in the object explorer, it begins to highlight the differences with SSMS and where avid users of SSMS may be expecting to see a myriad of administrative options that are not yet available in ADS.

For example, right-clicking on the database reveals a context menu with limited options. Although, the primary options that one would expect such as “New Query” and “Backup/Restore” are available, there are many options missing when comparing to a more mature tool such as SSMS. Options such as “Detach”, “Take Offline”, “Shrink”, etc. are not found in ADS likely due to the primary focus of ADS is for accessing data and many of the administrative commands available in SSMS can be utilized using T-SQL commands. It is also worth noting that the tooling void for administrative actions is currently being filled through extensions.

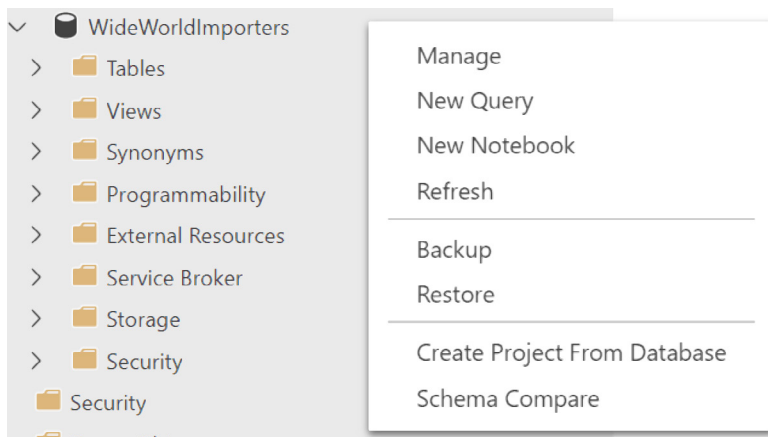


Figure 8: DB context menu

Queries

Clicking “New Query” opens a query editing window that allows for entering SQL statements.



Figure 9: New Query Editor

Full IntelliSense and syntax highlighting are included as well as support for code snippets.

Start typing “sql” and the code snippets bundled with ADS will be listed. Custom code snippets can also be added.

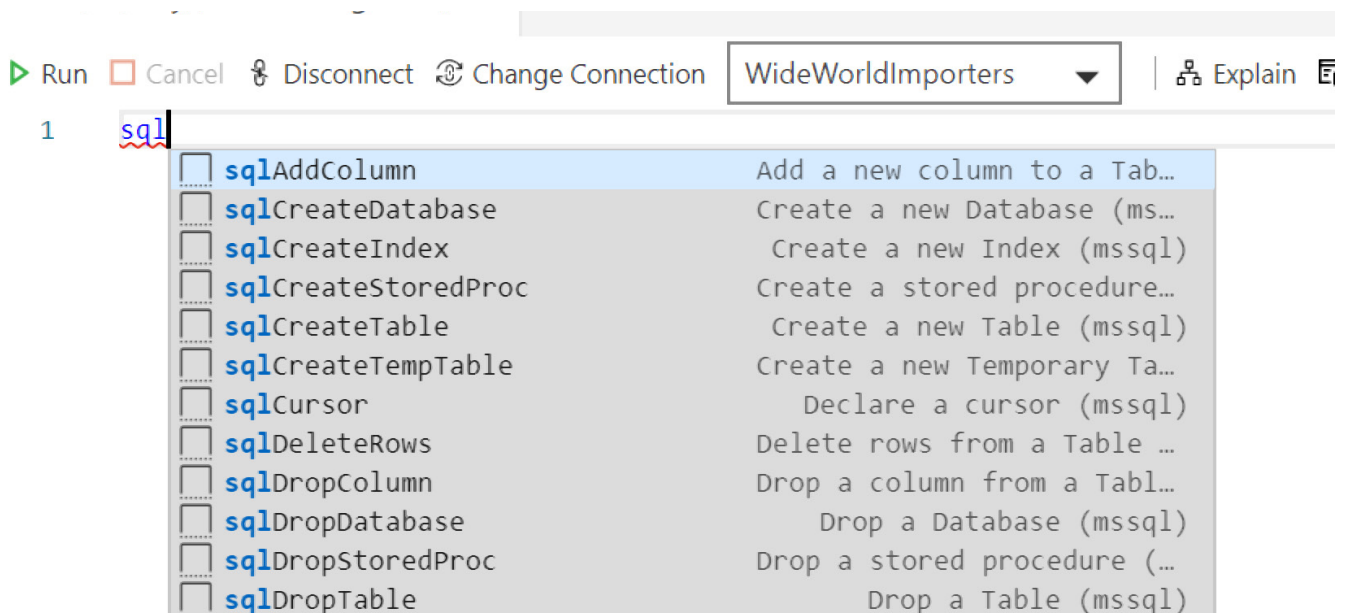


Figure 10: Code Snippets

Running the query will produce the records as rows into the results pane beneath the statements.

The screenshot shows a SQL query window with the following text:

```

1  SELECT s.CustomerID,
2      s.CustomerName,
3      sc.CustomerCategoryName,
4      pp.FullName AS PrimaryContact,
5      ap.FullName AS AlternateContact,
6      s.PhoneNumber,
7      s.FaxNumber,
8      bg.BuyingGroupName,
9      s.WebsiteURL,
10     dm.DeliveryMethodName AS DeliveryMethod,
11     c.CityName AS CityName,
12     s.DeliveryLocation AS DeliveryLocation,
13     s.DeliveryRun,
14     s.RunPosition
15 FROM Sales.Customers AS s
16 LEFT OUTER JOIN Sales.CustomerCategories AS sc
17 ON s.CustomerCategoryID = sc.CustomerCategoryID
18 LEFT OUTER JOIN [Application].People AS pp
19 ON s.PrimaryContactPersonID = pp.PersonID
20 LEFT OUTER JOIN [Application].People AS ap
21 ON s.AlternateContactPersonID = ap.PersonID
22 LEFT OUTER JOIN Sales.BuyingGroups AS bg
23 ON s.BuyingGroupID = bg.BuyingGroupID
24 LEFT OUTER JOIN [Application].DeliveryMethods AS dm
25 ON s.DeliveryMethodID = dm.DeliveryMethodID
26 LEFT OUTER JOIN [Application].Cities AS c
27 ON s.DeliveryCityID = c.CityID
    
```

Below the query is the 'Results' pane showing a table with 27 rows and 6 columns: CustomerID, CustomerName, CustomerCategoryName, PrimaryContact, AlternateContact, and Phone. The first few rows are:

CustomerID	CustomerName	CustomerCategoryName	PrimaryContact	AlternateContact	Phone
12	Tails핀 Toys (Biscay, MN)	Novelty Shop	Heloisa Fernandes	Amornrat Rattanaporn	(
17	Tails핀 Toys (East Fultonha...	Novelty Shop	Adam Kubat	Gulzar Naidu	(
16	Tails핀 Toys (Coney Island,...	Novelty Shop	Nitin Matondkar	Joy Dutta	(
15	Tails핀 Toys (Batson, TX)	Novelty Shop	Filips Jaunzems	Bharati Bhowmick	(
14	Tails핀 Toys (Long Meadow, ...	Novelty Shop	Tereza Valentova	Brijesh Ganguly	(
24	Tails핀 Toys (Dundarrach, N...	Novelty Shop	Intira Mookjai	Young-Tae Kim	(
13	Tails핀 Toys (Stonefort, IL)	Novelty Shop	Razeena Hosseini	Leticia Ribeiro	(
11	Tails핀 Toys (Devault, PA)	Novelty Shop	Elnaz Javan	Jayashish Ghatak	(
10	Tails핀 Toys (Wimbledon, ND)	Novelty Shop	Siddhartha Parkar	Paula Matos	(
9	Tails핀 Toys (Netcong, NJ)	Novelty Shop	Sointu Aalto	Jae-Gon Min	(
8	Tails핀 Toys (Bow Mar, CO)	Novelty Shop	Kanti Kotadia	Hoa Cu	(
7	Tails핀 Toys (Frankewing, T...	Novelty Shop	Kalidas Nadar	Filip Nedvidek	(
6	Tails핀 Toys (Jessie, ND)	Novelty Shop	Biswajeet Thakur	Radha Shah	(
5	Tails핀 Toys (Gasport, NY)	Novelty Shop	Johanna Huiting	Robert Ruutli	(
33	Tails핀 Toys (Boyden Arbor,...	Novelty Shop	Haasini Rai	Viktorie Melicharova	(
4	Tails핀 Toys (Medicine Lodg...	Novelty Shop	Daniel Roman	Leyla Radnia	(
3	Tails핀 Toys (Peeples Valle...	Novelty Shop	Bhaargav Rambhatla	Mudar Jevtic	(
2	Tails핀 Toys (Sylvanite, MT)	Novelty Shop	Lorena Cindric	Hung Van Groesen	(
1	Tails핀 Toys (Head Office)	Novelty Shop	Waldemar Fisar	Laimonis Berzins	(

The status bar at the bottom indicates: Ln 28, Col 1 Spaces: 4 UTF-8 CRLF SQL MSSQL 663 rows 00:00:00 localhost: WideWorldImporters

Figure 11: Query Results

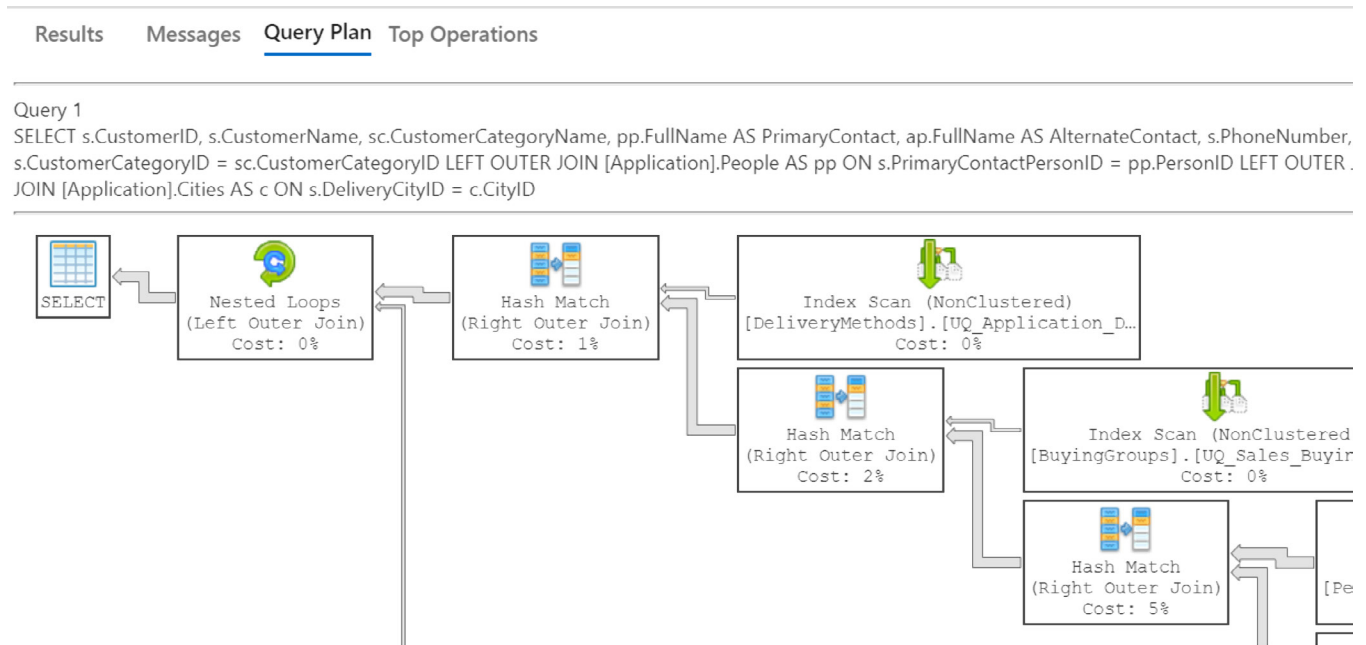


Figure 12: Query Plan

Saving the query tab will create a .sql file. As we will see later in the article, saving .sql files or exporting and saving as Notebooks, along with the built-in support for source code management, provide for better organization, version control, and safekeeping of your SQL assets.

While you do not get all the bells and whistles that SSMS offers when creating and running SQL statements, you do get some additional features that are unique to ADS. As an example, the group of menu icons to the right of the results pane offer the following options that can be applied directly against the results of the query:



Figure 13: Results Pane Options

Each of the “Save as...” options will save the results to a file of the respective format. This is helpful for taking the results and running additional analysis outside of ADS in another analytical tool or for simply saving a snapshot of the data at a point in time.

The “Chart” option will create an interactive chart based on the data in the results pane.

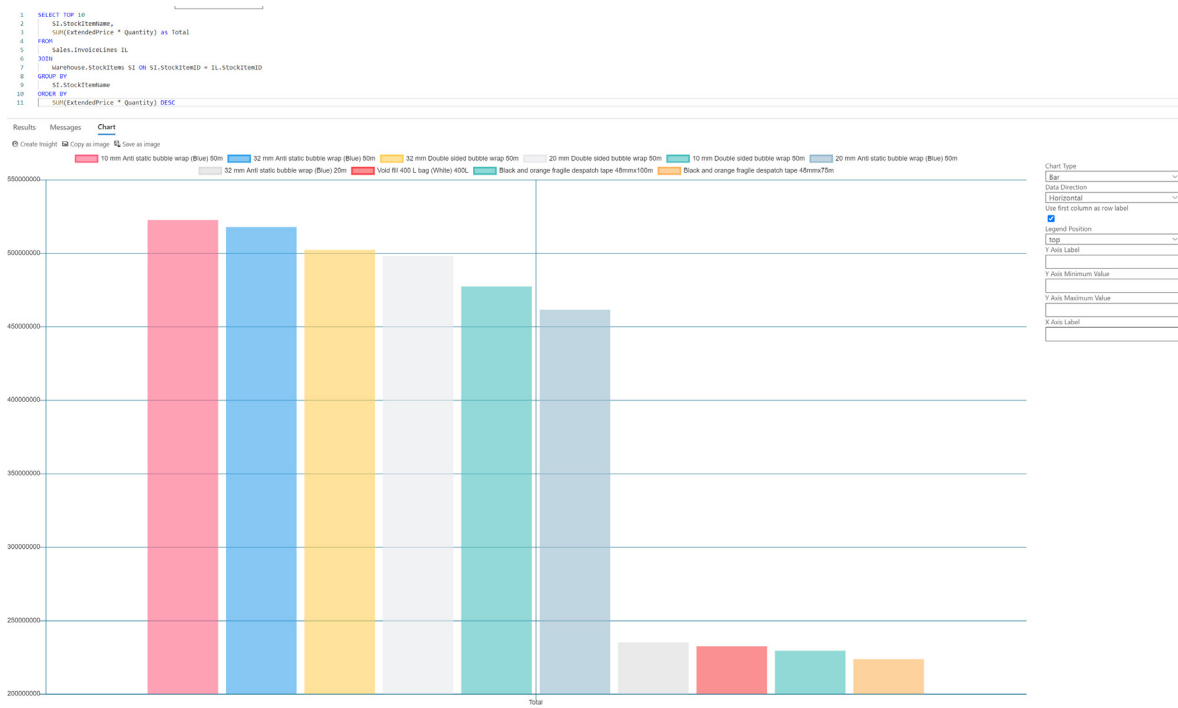


Figure 14: Chart Example

Query History

An option I have really come to appreciate is the “Query History” view (added via extension). I have often closed a file in SSMS or closed SSMS entirely without saving the current query session and regretted not being able to “recall” the unsaved file/query. To help with this, ADS includes a query history pane (accessible from within the View menu [View > Query History]) that lists recently executed queries. It also offers the ability to run the query directly from the history list or load the query into the query editor!

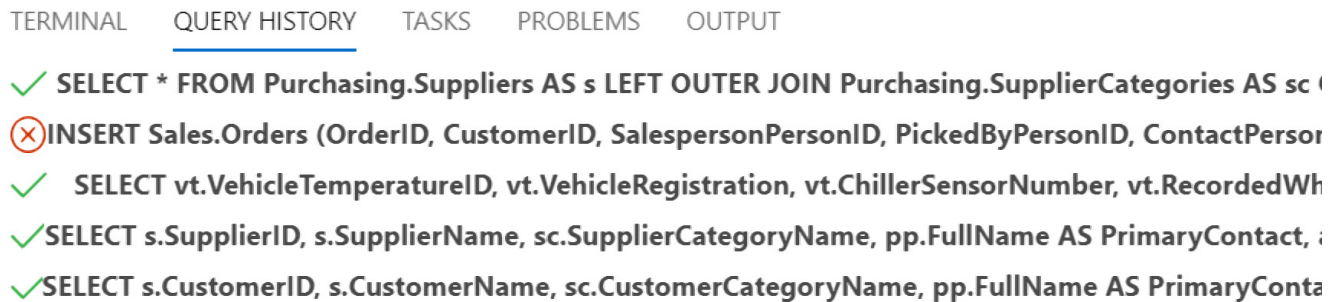


Figure 15: Query History View

Notebooks

Interactive programming has been exploding in recent years and the proliferation of this approach to producing code has been a staple in the Python community with access to products such as Jupyter Notebook developed and released by Project Jupyter. With Azure Data Studio, interactive programming is available directly in the interface and native support for Jupyter notebooks is a unique feature to ADS. Because this feature is not available in SSMS, the benefits that DBAs have been finding in the usage of creating interactive SQL code and documentation, could lead them to add ADS as a core data tool.

Let’s take a quick walk through of creating and working with a Notebook.

From the main menu, File > New Notebook or Alt+Windows+N on the keyboard will open a new Notebook editor.

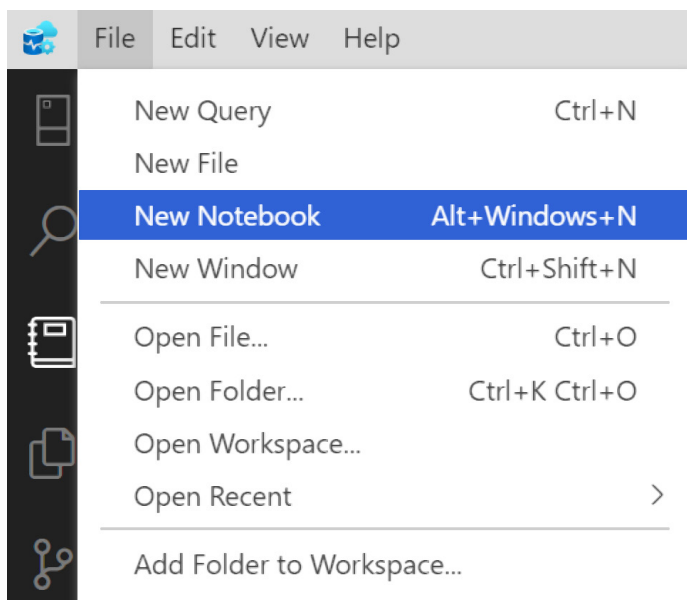


Figure 16: New Notebook

With the editor window open, the default Kernel is set to SQL. All your existing connections are accessible from the “Attach to” dropdown or you have the option of creating a new connection to use directly with the Notebook.

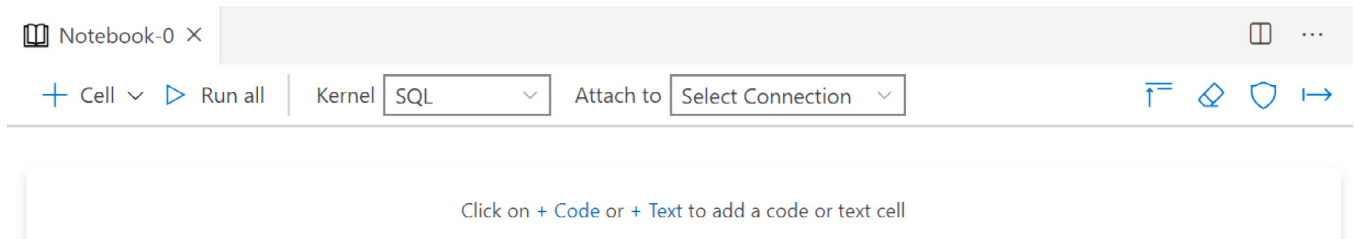


Figure 17: Notebook Editor

The list of available Kernels currently includes SQL, PySpark, Spark with Scala, Spark with R, Python, and PowerShell.

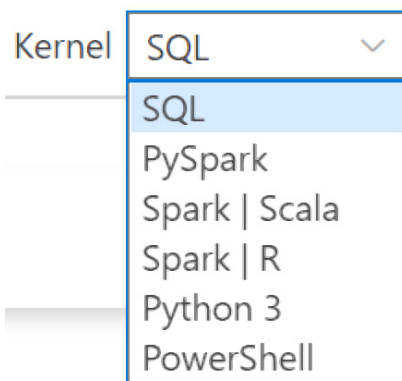


Figure 18: Notebook Kernels

It is interesting to see the mix of the other well-known data analysis languages with SQL inside a data access tool.

As mentioned earlier, one of the goals with Azure Data Studio was to support additional user roles such as Data Engineers and Data Scientists. With the addition of the SQL Kernel, it opens even more data analysis possibilities, as well as potential for use cases that are primarily targeted at DBAs.

There are two types of cells that can be created. One is a text cell that is used primarily for describing the interactive coding cells that are interweaved throughout the Notebook. Clicking to add a text cell will bring up the text editor.

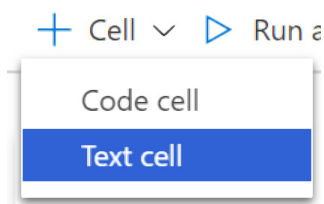


Figure 19: New Cell

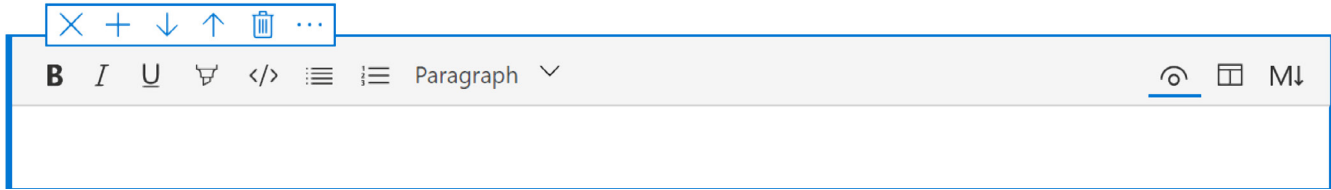


Figure 20: Text Cell

An early complaint from DBAs when working with the Notebooks, was the need to have to learn the intricacies of creating formatted documentation using Markdown directly. As such, a simple but effective rich text formatting tool was included to reduce the reliance on having to use Markdown syntax only. However, if one prefers to hand-code text formatting using the Markdown syntax, that option is still available as well.

Here is a simple example of how both text and code (and data!) can be combined to produce interactive documentation more declaratively. It starts with a text cell to build the title of the Notebook, followed by a code cell to insert the SQL statements. The generated results of the SQL statements are displayed directly in the Notebook and the results table includes options found in the standard Query results pane for exporting data and viewing the results in a chart.

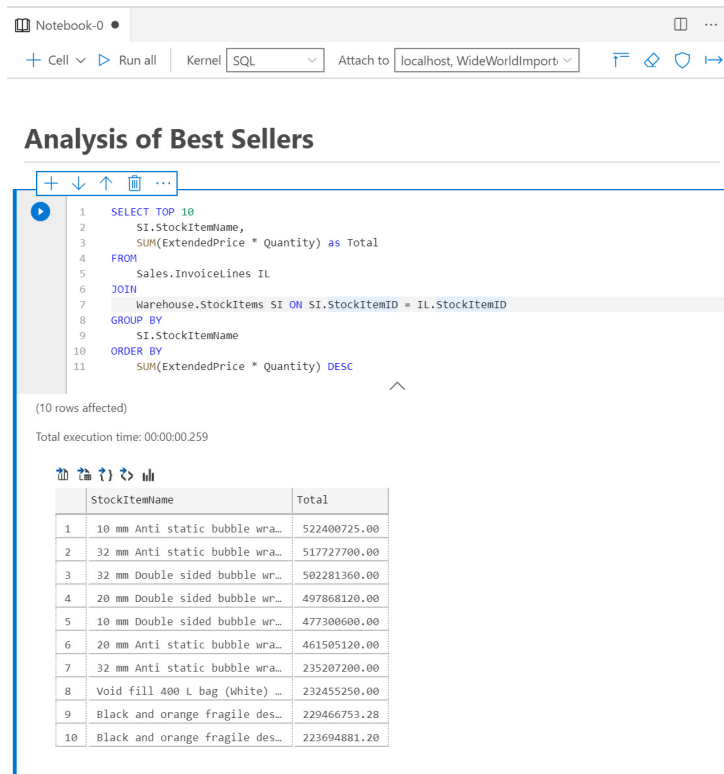


Figure 21: Code Cell

Additional text and code cells can be created throughout the Notebook document and saving the file uses the Jupyter Notebook.ipynb extension allowing for this Notebook to be shared with others to load directly into Jupyter Lab or any other data analysis tool with native Jupyter Notebook support.

Saved Notebooks are available via the "NOTEBOOKS" section.

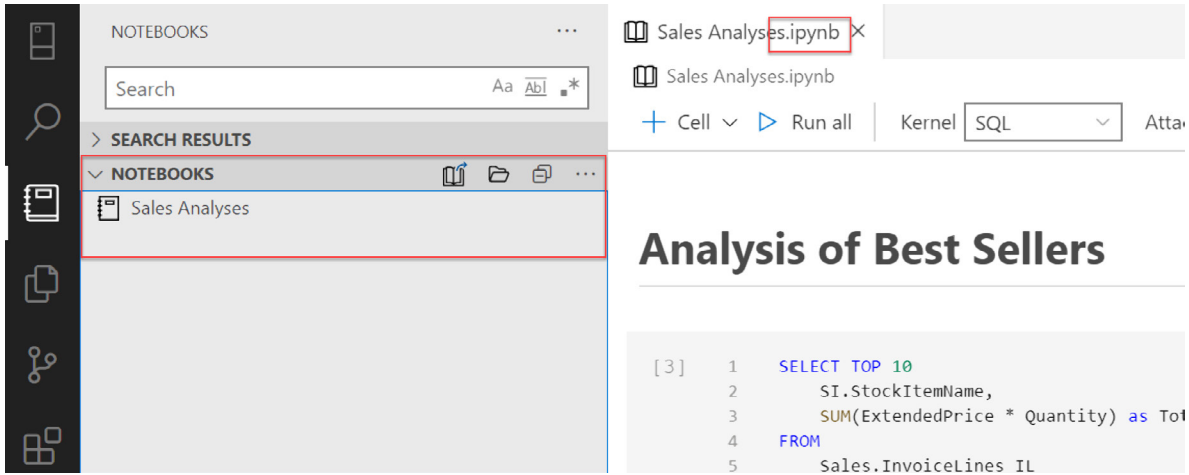


Figure 22: Save Notebook

A collection of Notebooks can be grouped together to form a “Book” with the opportunity to link directly to Notebooks within the entire book and provide a robust collection of interactive documentation that is far superior to maintaining a collection of SQL files and sharing them ad-hoc amongst your team.

Microsoft’s internal data team has been making extensive use of Notebooks for sharing key documentation. They have also started to make these Books available outside of Microsoft. As an added benefit within ADS, provided Books can be accessed and downloaded directly from within ADS via the Command Palette.

Type `Ctrl+shift+p` on the keyboard to bring up the Command Palette. Start typing “jupyter” to view a list of commands that include the “Jupyter Books: SQL Server 2019 Guide” and hit “enter”.

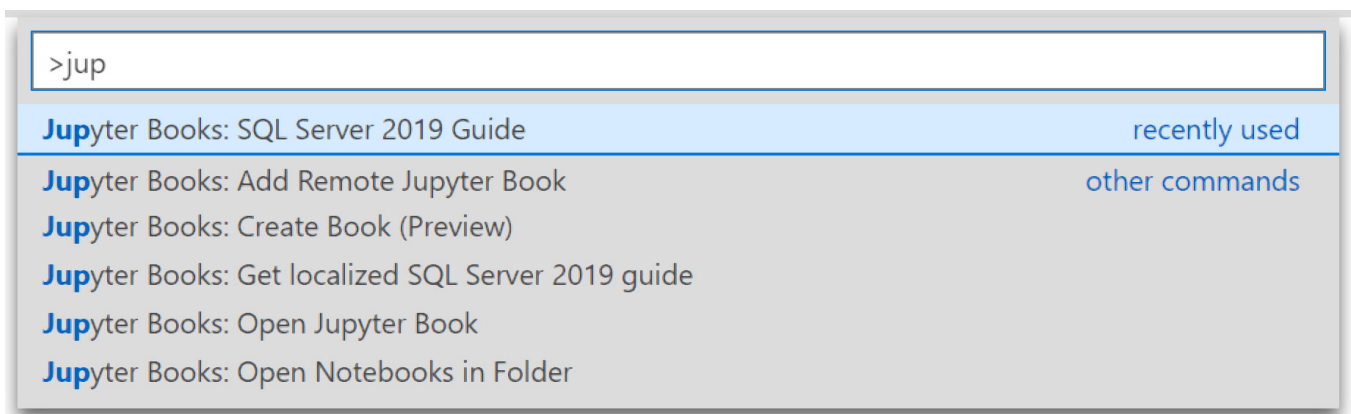


Figure 23: Accessing Provided Books

This will load an entire Book of interactive Notebooks related to managing and supporting SQL Server 2019 Big Data Clusters. This is a great example showing the power of using interactive Notebooks for documentation, troubleshooting, and knowledge sharing.

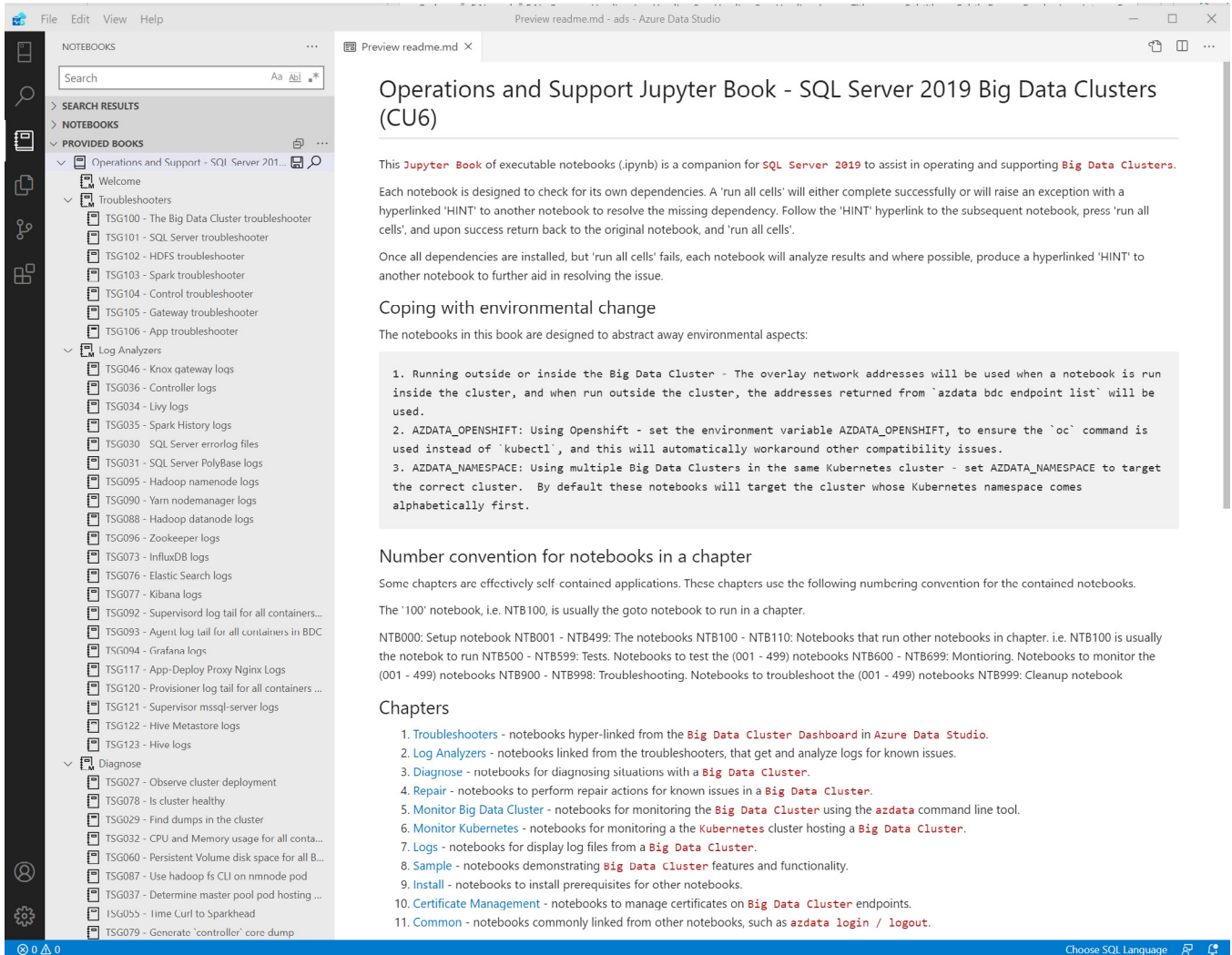


Figure 24: Provided Book Example

In addition to creating and utilizing Notebooks for operations, they can also be useful to create and share for collaboration, reporting, and data workflows including experimentation, data prep, and data analysis.

Source Code Management

Visual Studio Code offers strong tooling and support for source code management including tight integration with GitHub. Azure Data Studio adopts this feature from VS code and offers source code management for all the created .sql, notebook (.ipynb), and database project files.

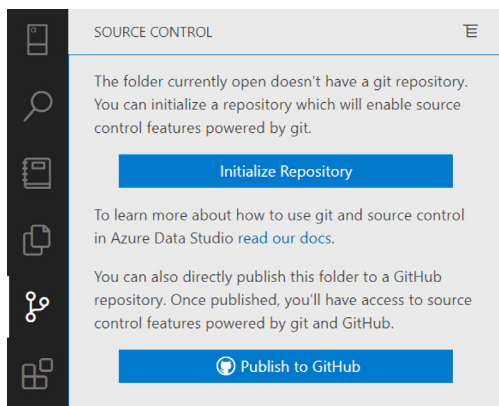


Figure 25: SCM View

Terminal

Another feature that users of VS Code will be familiar with and appreciate in ADS is the easy access to the integrated terminal window. Pressing “Ctrl+`” opens a new terminal window for applying commands directly from a command line. The default here is set to “powershell”, but any installed terminal window can be set to default including the standard Windows Command Prompt or a Linux shell such as the Bash Shell.



Figure 26: Terminal Window

Extensions

Sharing similarities to VS Code, Azure Data Studio seeks to shine as a query editor first and promotes access to viewing and analyzing data as a first-class experience. To keep the footprint light and the eye on fast editing of SQL and Notebook files, additional capabilities are handled through the growing list of extensions.

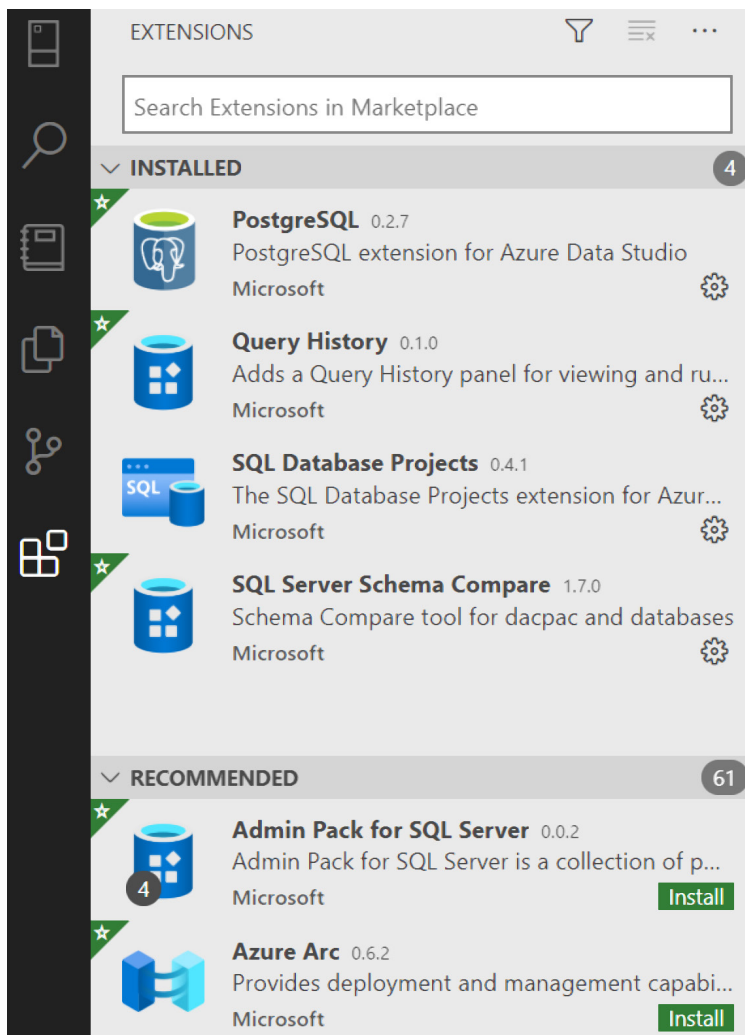


Figure 27: Extensions Panel

SQL Agent and Profiler as an example, provide extending the capabilities of ADS to bring in more administration functionality. Other notable extensions include the “Query History” view (demonstrated earlier), PostgreSQL for adding support for Postgres databases, Dacpac support, and SandDance for creating enhanced visualizations from the query results pane.

The existing list of available extensions is nowhere near that of VS Code, but this should also improve over time as more extensions are added to the marketplace.

Roadmap

While there is no definitive roadmap available, looking at the upcoming milestones and popular issues in the GitHub project, the next few releases will likely be maintenance releases for ensuring more stability in the product.

There is demand from the community for MySQL support which will likely be added via an extension (similar to support for PostgreSQL). Other high-demand features include a table design editor and increased performance for executing and viewing large datasets. I would personally like to see support for the .NET kernel added as an option in Notebooks – this is apparently in the works, but no timeline has been released for availability.

Which to Use – Azure Data Studio or SQL Server Management Studio?

Obviously, for anyone working outside of a Windows OS, Azure Data Studio is the only option when comparing the two. For ardent users of SQL Server Management Studio, the value proposition to move to ADS may not be apparent at first glance. However, the unique ability to access the SQL kernel in a lightweight editor and create notebooks and notebook collections easily, gives the nod to ADS when running data analysis tasks or building SQL scripts in the form of Notebooks to share with colleagues.

According to Microsoft, “Research has shown that users spend an order of magnitude more time working on query editing than on any other task with SQL Server Management Studio.” That’s an interesting fact considering the power that SSMS has to offer for database management capabilities but management of the database is always in the hands of a chosen few on most teams and access to the data is relevant for all.

If you are a command line user, the built-in terminal with ADS is where you will spend time sending admin commands via utilities like sqlcmd or PowerShell. If you prefer the GUI experience and lots of wizards, SSMS is better. SSMS will also be the preferred choice for security management, live query statistics, and provides tooling for viewing and creating database diagrams and table designers.

When speed is a factor? Then stick with SSMS. ADS is behind on rendering performance. Time to render the data is slower in ADS when compared directly to SSMS and scrolling longer lists of data show a noticeable lag. I am sure this will improve as the updates continue to be provided but for now, if you are working with large tables and datasets, SSMS is your best bet.

Summary Comparison:

	ADS	SSMS
Interactive Coding and Visualization	Yes	
Integrated Terminal	Yes	
GUI-Driven Administration		Yes
Diagramming Tools		Yes
Large Query Results		Yes
Extensions	Yes	
Integrated SCM	Yes	
Comprehensive Query Statistics		Yes

Conclusion:

As a product, Azure Data Studio is still early in its development and evolution. However, as a lightweight, cross-platform editor and data-viewer, it ticks a lot of boxes for accessing data and running analysis through the integrated Notebooks. For SQL Server users running on Windows, it could be a hard sell initially to buy into using ADS over SQL Server Management Studio. However, using SSMS for heavy administration-related tasks and large datasets, and using ADS for the Notebook support and integrated terminal will make a lot of sense for many users.

I like to work with PostgreSQL, and I have since abandoned other query and data editors, including some commercial options, to work exclusively in Azure Data Studio. I have noticed vast improvements in stability and feature set in the past 12 months, and I am enthusiastic about the continued potential for this product.



Darren Gillis

Author

Darren Gillis is a Toronto-based software developer and technologist with 20+ years of experience primarily with Microsoft technologies. He is a big fan of data and Microsoft Azure, having architected numerous cloud-based software projects leveraging any of the features that Azure has to offer. He is currently developing a SaaS based compliance platform using C#, React, and PostgreSQL. Feel free to connect with him on LinkedIn [linkedin.com/in/darrengillis](https://www.linkedin.com/in/darrengillis), or follow him on twitter @darrengillis



Technical Review
Gouri Sohoni



Editorial Review
Suprotim Agarwal

YOU'R INVITED TO OUR

50th

EDITION

D N C M A G A Z I N E

• APRIL 2021 •

SUBSCRIBE NOW

www.dotnetcurry.com



TIC	X	TAC
O	TOE	O
in	F#	Part II

F# is a .NET based functional-first programming language. In this part of the article series, I will continue exploring the F# language, by using an example of the Tic Tac Toe game.



Introduction

In the [Tic Tac Toe in F# - Part 1](#) (Part 1 of this tutorial series), I talked about F# and explored some of its features using the Tic Tac Toe game as an example. In this part, I am going to continue exploring some other features. Here is the link to the source code to help you follow along:

<https://github.com/y Assad/TicTacToeFSharp/blob/master/src/BoardModule.fs>

In the [previous tutorial](#), I explained the following function:

```
let getRow index board =  
    board.Rows |> Map.tryFind index |> Option.defaultValue emptyRow
```

The following expression:

```
board.Rows |> Map.tryFind index
```

..gives us the row in the board at the specified index. The `tryFind` function in this case returns a value of type `Row option`. It will either contain a `Row` value, or no value in case the map does not have a value at the specified index.

The result is then passed (using the forward pipe operator “|”) as the “second” argument to the `Option.defaultValue` function. This function is used to return an alternative value (`emptyRow` in this case) if no `Row` was found.

Here is the definition of the next function, `getCell`:

```
let getCell rowIndex columnIndex board =  
    let row = getRow rowIndex board  
    getCellInRow columnIndex row
```

It takes the location of a cell in the board as row and column indexes including the board itself, and returns the cell value in the board. Based on the knowledge of F# you learned so far, you should be able to understand this function.

Editorial Note: Reading the first part of this tutorial series is highly recommended to understand the rest of the tutorial in this series. In case you haven't already, read it here [Tic Tac Toe in F# - Part 1](#).

The next function is more interesting:

```
let allCellsInRow row =
    seq {
        yield getCellInRow One row
        yield getCellInRow Two row
        yield getCellInRow Three row
    }
```

As you might have guessed, the job of the `allCellsInRow` function is to return all cells in a row! It takes a `Row` and returns a sequence of `CellStatus` values.

More specifically, the return type of this function is `seq<CellStatus>`. A sequence in F# is like `IEnumerable` in C#. Actually, if you use an IL decompiler to view the `allCellsInRow` function, you will see that the return type of this function is `IEnumerable<CellStatus>`.

Items in an F# sequence is evaluated as the sequence is consumed. For example, if you use only a single element from the sequence returned by `getCellInRow`, only a single invocation of `getCellInRow` will be made (instead of three).

The body of this function is a sequence expression. A sequence expression is created using the `seq { }` syntax. If you ever created an `iterator method` in C#, then you might have guessed what the F# `yield` keyword does in this function. Here, `yield` describes the intention to return an item. However, only when an item is requested by the consumer, does the expression that follows the `yield` keyword gets evaluated.

It is worth noting here that the `yield` keyword in F# is not designed only for sequence expressions. A sequence expression is just one type of a `computation expression`. Computation expressions allow us to do very interesting things in F# but they are out of scope of this introductory article series.

An even more interesting function is the `allCells` function:

```
let allCells board =
    seq {
        yield! getRow One board |> allCellsInRow
        yield! getRow Two board |> allCellsInRow
        yield! getRow Three board |> allCellsInRow
    }
```

This function returns all the cells in a board as a sequence. For each row index, it obtains the corresponding Row in the board and then it calls `allCellsInRow` to obtain the cells in that row.

All of this is done inside a sequence expression. And here, `yield!` is used instead of `yield`. If we used `yield` instead of `yield!`, then the return type of `allCells` would be `seq<seq<CellStatus>>` (a sequence of sequences). This is because the return type of `allCellsInRow` is `seq<CellStatus>`.

`yield!` has the effect of flattening the yielded sequence into individual items. In C#, there is no corresponding feature. An `AllCells` function in C# would look like this:

```
public static IEnumerable<CellStatus> AllCells(Board board)
{
    foreach (var item in AllCellsInRow(GetRow(Index.One, board)))
        yield return item;

    foreach (var item in AllCellsInRow(GetRow(Index.Two, board)))
        yield return item;

    foreach (var item in AllCellsInRow(GetRow(Index.Three, board)))
        yield return item;
}
```

Here is the next function:

```
let updateCellInRow row index newStatus =
    {Cells = Map.add index newStatus row.Cells}
```

The `updateCellInRow` function “updates” the status of a cell in a specific Row. Of course, when I say updates, I mean return a new Row with the updated cell.

As explained in [part 1](#) of this tutorial series, the return value of this function is called a *record expression*. In this case, we are creating a Row value. For Cells, we are “updating” the original row cells to modify the cell in the specified index. The `Map.add` function will either add a new entry to the map or will “replace” the existing entry with a new value.

The next function `updateRowInBoard` updates a row in a board. It is very similar to `updateCellInRow`.

With the knowledge you have so far about F#, you should easily be able to understand the `updateCell` function (which updates a specific cell in a board), and the `emptyBoard` value (which represents a brand-new board).

Notice how `updateCell` uses `updateCellInRow` and `updateRowInBoard` to do its job.

Next is the `isFull` function:

```
let isFull board =
    board |> allCells |> Seq.forall (fun c -> c = HasX || c = Has0)
```

You give this function a board and it tells you whether the board is full, i.e., whether all cells have been used.

Notice the forward pipe operator (`|>`) in this function. We start with the board and give it to the `allCells` function. This will give a sequence of all cells in the board.

Then I give this sequence to the `Seq.forall` function. `Seq.forall` will test each cell in the sequence to see if it matches a certain condition. In this case, we use a *lambda expression* for such a test. The lambda returns `true` if the cell status is `HasX` or `Has0`. The keyword `fun` is used to express a lambda expression.

The `Seq.forall` function takes two parameters, a test function (or a predicate), and a sequence. In the `isFull` function, we are passing the lambda as the first parameter of `Seq.forall`. For the second parameter, we are passing the return value of `allCells` (using the forward pipe operator).

It is worth noting that instead of a lambda, we can use a function value like this:

```
let hasXOrO c = c = HasX || c = HasO

let isFull board =
    board |> allCells |> Seq.forall hasXOrO
```

The first parameter of `Seq.forall` which is the predicate parameter is of type `(T -> bool)`. The type of this parameter is a function. So, when we call `Seq.forall`, we pass a function. The first time, we did so using a lambda expression, and the second time using a function value.

It is worth noting that functions like `Seq.forall` are called higher-order functions. More specifically, higher-order functions are functions that have parameters of type function, and/or that return a function.

Let's look at the `formatCell` function:

```
let formatCell cell =
    match cell with
    | HasX -> "X"
    | HasO -> "O"
    | Empty -> "_"
```

The body of this function is a *match expression*. A match expression is very similar to the [switch expression introduced in C# 8](#). It allows for branching based on comparing a value (cell in the `formatCell` function) with multiple *patterns*. In the case of the `formatCell` function, the function evaluates to "X" when the cell value is HasX, "O" when it is HasO, and "_" when it is Empty. There is much more to pattern matching in F# than this, but this is the extent I will talk about it in this article.

Next is the `writeBoard` function:

```
let writeBoard board write =
    let writeRow rowIndex =
        let row = getRow rowIndex board
        getCellInRow One row |> formatCell |> write
        write " "
        getCellInRow Two row |> formatCell |> write
        write " "
        getCellInRow Three row |> formatCell |> write
    writeRow One
    write Environment.NewLine
    writeRow Two
    write Environment.NewLine
    writeRow Three
    write Environment.NewLine
```

The `writeBoard` function takes a board and writes it. Instead of writing it directly to the console, the `writeBoard` function takes another parameter called `write` that it will use when it wants to `write` a specific string. The caller of `writeBoard` can pass an argument for `write` that writes to the console, or to any other destination.

If you use IntelliSense to see the type of the `write` parameter, it looks like this:

```
(string -> unit)
```

This means that the `write` parameter is a function that takes a string and returns unit.

`Unit` is basically like `void` in C#. However, unlike `void`, `unit` is a real type.

`Unit` exists in many functional programming languages to make all functions return something - even if that something is nothing (pun intended)!

In C#, the type of the `write` parameter would be `Action<string>`. If C# had `unit` instead of `void`, then the type of this parameter in C# would be `Func<string, unit>` which would remove the need to have any `Action` delegates (like `Action<T>`) in the .NET frameworks.

Because the `writeBoard` function takes a parameter of type function, it is another example of a higher-order function.

Inside `writeBoard`, another function called `writeRow` is defined. This function takes a `rowIndex`, fetches the row from the board, and then writes it.

Functions defined inside functions have access to the parameters of the parent functions. They also have access to values (functions or otherwise) defined before themselves. For example, the `writeRow` function has access to (and uses) the `board` parameter of `writeBoard`.

Since version 7.0, C# has a similar feature called `local functions`.

After defining the `writeRow` function, the `writeBoard` function uses it to write the three rows of the board.

Next, I am defining the `allLinesIndexes` value:

```
let allLinesIndexes =
    seq {
        //Horizontal
        yield [One, One; One, Two; One, Three ]
        yield [Two, One; Two, Two; Two, Three ]
        yield [Three, One; Three, Two; Three, Three ]

        //Vertical
        yield [One, One; Two, One; Three, One ]
        yield [One, Two; Two, Two; Three, Two ]
        yield [One, Three; Two, Three; Three, Three ]

        //Diagonal
        yield [One, One; Two, Two; Three, Three ]
        yield [One, Three; Two, Two; Three, One ]
    }
```

This is a value, not a function. The value here is a sequence expression. Here is the type of this value:

```
seq<(Index * Index) list>
```

This is a sequence. Each element of the sequence has the following type:

```
(Index * Index) list //this can also be written as list<(Index * Index)>
```

..which is a *list*, with each element in the list having the following type:

```
(Index * Index)
```

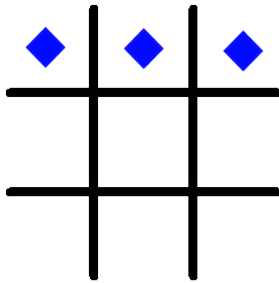
..which is a *tuple* containing two elements, each of type `Index`.

Here, I am trying to encode the 8 possible lines that a player can fill in the game in order to win. There are three horizontal lines, three vertical lines, and two diagonal lines.

I am encoding each line as a list of tuples (the list will contain three elements), each tuple contains the row and column indexes of the cell. For example, the following code:

```
yield [One, One; One, Two; One, Three ]
```

..represents the following line in the board:



The yielded expression is a list containing three elements:

```
One, One  
One, Two  
One, Three
```

A semicolon is used to separate elements of a list. Square brackets are used to denote the list.

Each of the three elements in the list is a tuple. The comma is used to separate the two items in each tuple.

Let's now look at the `allLineValues` function:

```
let allLineValues board =  
    let cell rowIndex columnIndex = getCell rowIndex columnIndex board  
    allLinesIndexes |> Seq.map (fun line -> line |> List.map (fun (r, c) -> cell r c))
```

The `allLineValues` function takes a board and returns a `seq<CellStatus list>` representing the cell values for each of the eight different lines in the board.

Inside the body of the `allLinesValues` function, the cell function is defined. This function returns the status of a cell given its row and column indexes.

The `allLineValues` function returns the following expression:

```
allLinesIndexes |> Seq.map (fun line -> line |> List.map (fun (r, c) -> cell r c))
```

The `allLinesIndexes` sequence (the 8 lines indexes) is given as the second argument to the `Seq.map` function. The `Seq.map` function is like the LINQ `Select` method in C#. The `Seq.map` function translates each line in `allLinesIndexes` to a list of `CellStatus` representing the status of cells in that line.

The first argument of `Seq.map` is a mapping function that will be used to transform each item in the source sequence. The mapping function passed to `Seq.map` is this function:

```
(fun line -> line |> List.map (fun (r, c) -> cell r c))
```

This is a lambda expression.

The lambda takes each line and then uses `List.map` to transform each cell in the line from a tuple of row and column indexes (`Index * Index`) to `CellStatus`. `List.map` is like `Seq.map` but for lists, not sequences.

In C#, we use the same `Select` method for all types of collections (e.g. `List<T>` or `T[]` or `ImmutableArray<T>`), but the result of the `Select` method is always `IEnumerable<T>`. In F# we can use specialized map methods to return a collection of the same input collection type. We can always use `Seq.map` for any kind of collection (like F# lists) if we want, but this will cause the resulting collection to be of type sequence.

The next function, `lineIsFullOf`, looks like this:

```
let lineIsFullOf line status =  
    line |> List.forall (fun s -> s = status)
```

This function takes some line as a list of `CellStatus`, and a specific status value, and then checks whether all cells in the line contain that specific status. For example, we can use it to check if one of the diagonal lines is full of Os. At least, that was my intention when I wrote this function.

However, if we use IntelliSense to see the type of this function, it looks like this:

```
val lineIsFullOf : line:'a list -> status:'a -> bool (requires equality)  
Full name: BoardModule.lineIsFullOf
```

The type of the line parameter is `'a list`. And the type of the status parameter is `'a`.

And to remind you, `'a` is a generic type parameter.

There is also an interesting “(requires equality)” at the end.

If you think about it, there is nothing in the body of the `lineIsFullOf` function to indicate to the compiler that the line parameter is a list of `CellStatus`. There is only an indication (in the body) that line must be a list of something. And for this “something”, there is an indication that it must support equality, for the following expression to make sense:

```
(fun s -> s = status)
```

That is, because in this lambda expression, we are checking equality between `s` and `status`, the type of `s` must support equality checking. Also, because we are checking equality between `s` and `status`, then `status` must have the same type as `s`.

If we are to manually write the types of the parameters and the equality constraint, the function would look like this:

```
let lineIsFullOf<'a when 'a :equality> (line: 'a list) (status : 'a) =  
    line |> List.forall (fun s -> s = status)
```

Notice how we use angle brackets to specify type parameters along with any constraints (`'a :equality` in this case).

F# tries hard to infer the types of parameters without us having to specify them manually.

Here is the last function in the `BoardModule` module:

```
let anyLineIsFullOf board status =  
    board |> allLineValues |> Seq.exists (fun line -> lineIsFullOf line status)
```

I leave this one for you to figure out.

Can you guess what this function does? What Seq.exists does? Tweet your response at @yacoubmassad or leave a comment.

This was the last function in `BoardModule`. In the next part of this tutorial series, I will talk about the `GameModule`.

Conclusion:

F# is a .NET based functional-first programming language. In this article series, I continued exploring the language by using an example of the Tic Tac Toe game.

In this part, I demonstrated sequence expressions, lambda expressions, higher-order functions, match expressions, functions defined inside functions, F# lists and tuples, the map functions, type parameters and how the F# compiler tries hard to infer the types of parameters without us having to specify them.



Yacoub Massad

Author

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



Technical Review

Damir Arh



Editorial Review

Suprotim Agarwal

.NET & JavaScript Tools



Shorten your Development time with this
wide range of software and tools

[**CLICK HERE**](#)

Benjamin Jakobus



What is MACHINE LEARNING?

“People worry that computers will get too smart and take over the world, but the real problem is that they’re too stupid and they’ve already taken over the world.”

Pedro Domingos

In a [previous article](#) of our [Machine Learning for Everybody](#) series, we gained a general overview of Artificial Intelligence (AI).

We saw that AI is concerned with solving difficult problems in dynamic environments and by examining some real-world problems, we attempt to understand that intelligent systems use very specific techniques to solve very domain-specific problems.

In this article, we are now ready to look at one of these techniques in detail: **Machine Learning**.

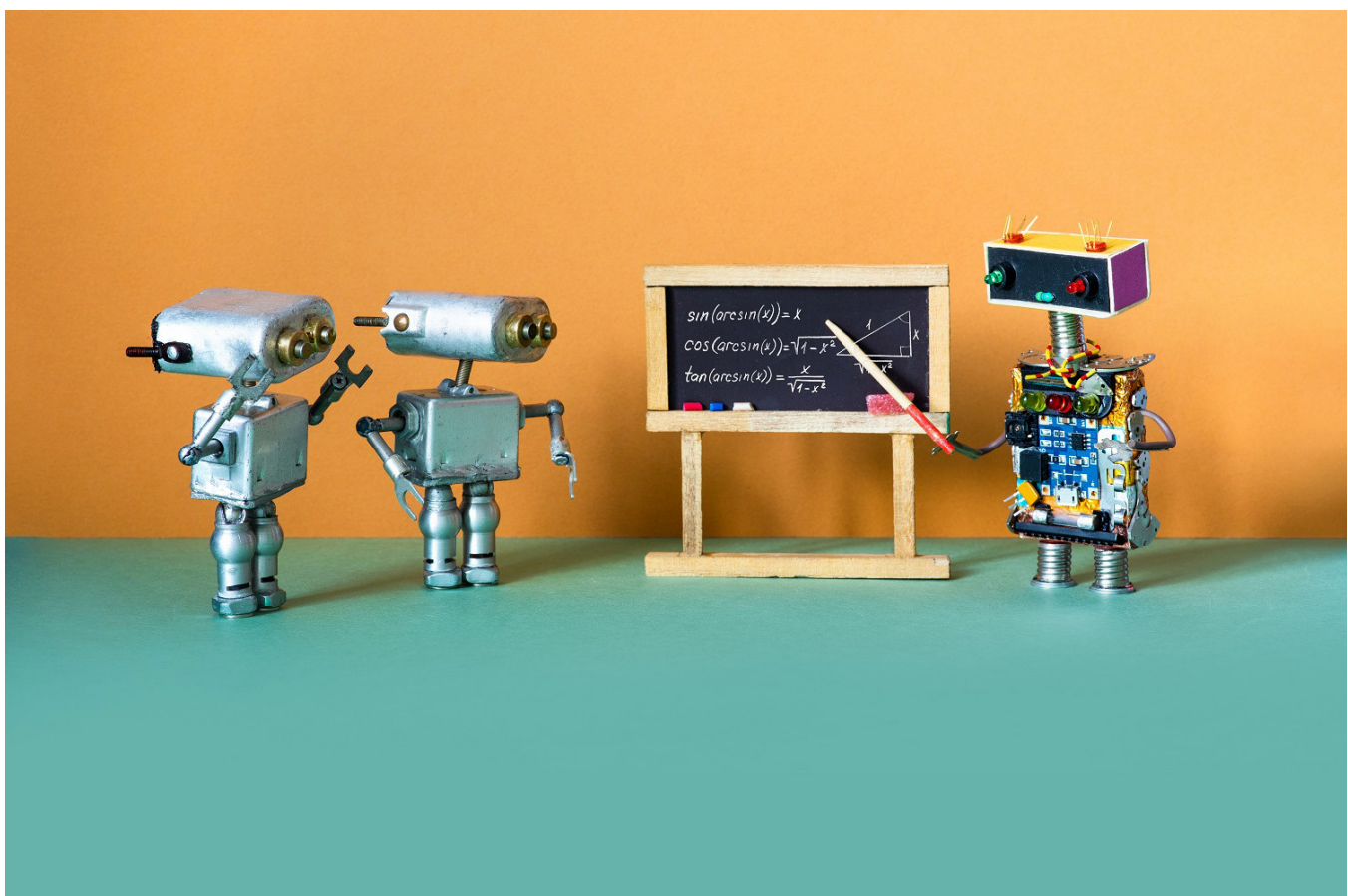
What is Machine Learning?

If there is only one message that you take away from this entire series on Machine Learning, then let it be the following: *machine learning is glorified statistics*.

That is, machine learning looks to solve problems by identifying patterns in historical data.

Having internalized these patterns, the software can then classify new, unseen data. The patterns are identified using statistics and internalized (i.e. “learned”). Therefore, any problem that lends itself well to statistical analysis, lends itself to machine learning.

It is important to remember however that not *all* types of problems lend themselves well to pattern recognition. Furthermore, using pattern recognition to build up knowledge and construct a model of the world is only a very specific type of learning. There are many other forms of learning. Forms at which humans excel at, and that we do not yet fully understand. Nevertheless, using statistical analysis, we can build programs that learn and improve over time, without requiring manual intervention.



Usually, an initial set of examples (called a “training dataset”) is used to “infuse” the software with knowledge about a very specific domain. As the software is being used, any unseen data that the software classifies is automatically added back into the training dataset. Depending on the type of problem being solved, users sometimes provide feedback as to the accuracy of the software’s prediction or classification. This feedback is then incorporated into the training cycle, resulting in software that becomes smarter and smarter, the more that it is being used.

Just like artificial intelligence in general, machine learning is inspired and influenced by a wide range of disciplines. The fusion of concepts from biology led to the development of neural networks (upcoming article) and genetic algorithms; the cognitive sciences inspired case-based reasoning (upcoming article) and advances in information theory resulted in decision trees.

At this point, these terms might sound alien. But don’t worry, we will cover them in detail in the upcoming articles. For now, just remember that machine learning draws from different concepts and builds on the understanding of the world which other fields of study brought to light.

How does machine learning differ from other problem-solving techniques?

So far, we have discovered that machine learning is a problem-solving technique that allows computer programs to learn patterns from data using methods borrowed from statistics. That is, the program developed using machine learning techniques looks at lots of data and then identifies patterns. It uses these patterns and tries to apply them to new, unseen data to determine “something” about this new data. But this all sounds very abstract. What exactly does this mean in practice?

Let’s look at a real-world use-case to answer this question: Patriot One Technologies Inc.

Patriot One Technologies Inc. is a Canadian defence company that provides covert weapons detection systems. Using a low-power impulse radar, the system creates a signal signature (think of this as simply a “digital representation” or “digital image”) of each person in range of the radar. Once created, these signal signatures are compared to the signatures in the database. The signatures in the database correspond to a wide range of signatures created of people carrying different types of concealed weapons - from knives and guns to bombs. When a new signature matches that of a signature found in the database, the system alerts a security guard or control centre, or sends a notification to law enforcement.

In classical programming, developers write computer code that describes the exact steps that the computer needs to take in order to detect each type of weapon. That is, the developers would first write code for translating the radar signals into some digital representation. For the sake of simplicity, let’s assume that this is simply an image created by translating the radar signals. They would then write a precise set of instructions that, using the image as an input, would allow the computer to detect different image features. For example, edges, straight lines, corners and certain textures.

Then, for each type of weapon, they would describe the steps involved in detecting them. For example, a rifle might consist of two long, straight lines (the barrel), followed by two smaller curved lines, ending in some lines shaped like a triangle (the stock).



Depending on the gun, the textures, positioning of edges or length of the lines may differ. The same is repeated for every single weapon type.

One doesn't need to be a programmer to realize that this approach is very cumbersome and error-prone, if not almost impossible. There is a huge variety of knives, guns and bombs. Their shapes, sizes and textures differ immensely and depending on the angle of the weapon, or the person, the weapon may only be partially visible.

Writing a precise set of steps to take into account all of the possible variations may therefore not even be feasible! And even if it were, the developers would need to write new code every time a new weapon makes it onto the market or every time a new possible angle, shape or size becomes available.

Using machine learning, the developers approach the implementation of such a system not by describing a set of steps that lead to classification, but by feeding a large set of "images" into the system, and helping the system identify the characteristics of specific threats.

That is, the developers first produce a large dataset that contains both digital signatures (or "images") of people carrying concealed weapons, and of people not carrying concealed weapons. They would then label these images according to the threat presented in them ("pistol", "rifle", "knife", "bomb" or "no threat") before feeding them into the system. The system would then use machine learning techniques to determine the characteristics of both "threat" and "no threat" images, and store and model them in such a way that they can be easily accessed. Once the radar signals create new, unseen images, the system will then check these images to see whether they match the characteristics of a known "threat" or "no threat" image.

To do so, the software will not try to measure the length of different lines or follow a series of steps for comparing the shape of different objects. Instead, it will simply take the image as a whole, and see to what degree the arrangement of the data that composes the image, corresponds to the characteristics that constitute one of the identified threats.

The advantage of this approach is many-fold.

Since the problem lends itself well to pattern recognition, this approach allows us to effectively deal with variations in image quality and account for distortions as well as variations in angle and distance.

Secondly, previously unseen images can be fed back into the training dataset once classified, allowing the system to "learn" and improve over time.

Thirdly, once new weapons reach the market, the system can be updated with minimal effort: one simply needs to record sample signatures of people carrying these weapons and then add them to the training dataset. The underlying code will not need to be updated or modified.

Defining machine learning

So far, we learned that machine learning is all about creating a model of the world using statistics and pattern recognition. By giving a real-world example of its application, we developed an understanding of how solving a problem using machine learning differs to solving a problem using traditional programming. What we do not yet have however, is a precise definition of machine learning.

What exactly do we mean when we say that a software "learns"? How can we define this process?

Luckily, defining what it means for a program to learn is a bit easier than defining intelligence or artificial intelligence. In his book "Machine Learning"¹, Tom M. Mitchell gives the following definition:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Now that's a precise definition if there ever was one. But what exactly does this mean? In essence, Mitchell's definition is a fancy way of saying that a program qualifies as a machine learning program, if, given some clear, predefined sets of tasks, the program gradually becomes better at performing these tasks over time. To know whether someone or something is getting better at performing a task, we need some way of measuring or quantifying their performance. This is what the "performance measure P" refers to in his definition. The "experience E" simply refers to the timeframe over which the program is being used. Stripping out the awkward letters, we can simplify his definition to read:

"A computer program performing a set of clearly defined tasks is said to learn from experience if, given an objective performance measure, the program improves at performing these tasks over time"

Designing a machine learning-powered program

Irrespective of what type of problem we are trying to solve using machine learning, there are several concrete design choices that developers need to make before writing even a single line of computer code. These design choices are closely linked to the definition of the term "machine learning" that we presented in the previous section. That is, one must:

1. Precisely define the objective that is to be accomplished.
2. Identify how to measure the software's performance when accomplishing these tasks.
3. Determine exactly what type of knowledge is to be learned.
4. Determine how this knowledge is to be learned.

1 Publisher: McGraw-Hill Science/Engineering/Math; (March 1, 1997), ISBN: 0070428077

This merely involves defining what it is that the program is meant to accomplish, in such a way that we can quantify (measure) it (a general description here won't suffice). In the case of the threat detection system developed by Patriot One Technologies Inc, we must precisely define what it means to detect a threat. Simply saying that the software "detects a threat" is not practical and will not help us with the implementation of such a system. Instead, we must define what a threat is.

In this case, a threat is defined as a person carrying a weapon. So, the task of the threat detection system can be stated as classifying a digital signature (i.e. "image" created by radar) using 4 labels: "no threat / no weapon", "gun", "knife", "bomb". If the person in the image is carrying a bomb, the software should apply the label "bomb" to it; in the case of a concealed knife, the label "knife", and so on, so forth.

Next, we must determine how we can measure how good the program is at performing its classification task. Luckily, in this case, the performance measurement is fairly straight forward: we simply implement a mechanism for counting the number of false positives and false negatives. That is, how many times does the program falsely classify somebody as carrying a knife/gun/bomb whilst they aren't; and how many times does the program classify somebody as not carrying a concealed weapon, whilst in fact they are. Over time, the number of incorrect classifications should diminish.

Third on our list, is the task of determining what "type of knowledge" to learn. In the case of the threat detection system, the software must be able to identify shapes and then classify them.

When defining the type of knowledge to deal with, machine learning experts define a "target function". This is a fancy way of saying that they determine a method for accepting and specifying input data, and then mapping this input to an output which maximizes the performance measure.

What do we mean by that?

Given a digital signature or image as an input, the target function acts as the formula for predicting whether or not the person represented by this input image is carrying a weapon or not (and if so, predicting the type of weapon).

Having defined the objectives, performance measure and type of knowledge to learn, one must determine *how* the knowledge that is used to make predictions or classifications is to be learned or assimilated. Experts call this the "training experience". Tom Mitchell summarizes this notion well in his book "Machine Learning" by using the example of learning how to play checkers:

"The type of training experience available can have a significant impact on success or failure of the learner. One key attribute is whether the training experience provides direct or indirect feedback regarding the choices made by the performance system. For example, in learning to play checkers, the system might learn from direct training examples consisting of individual checkers board states and the correct move for each. Alternatively, it might have available only indirect information consisting of the move sequences and final outcomes of various games played."

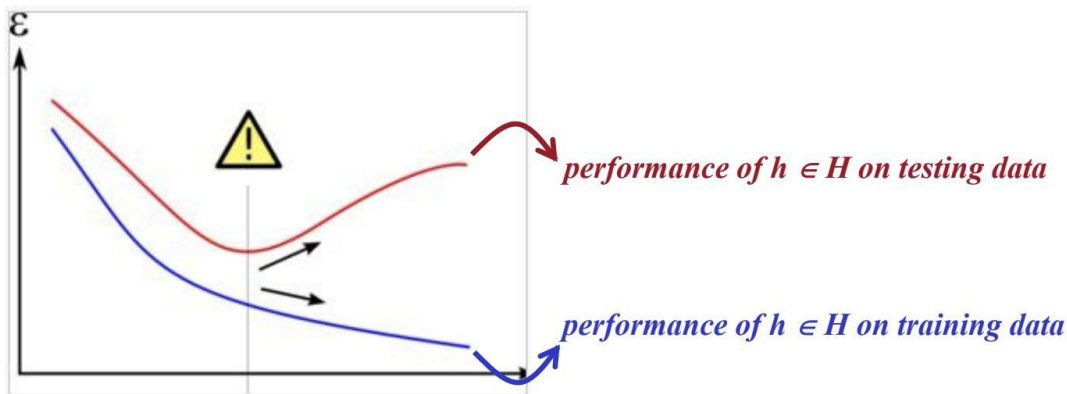
Common problems in machine learning

So far, we defined machine learning, discussed how it differs to other problem-solving techniques, and talked about how one would go about designing a machine learning-powered program. We have looked at some real-world examples, and know that machine learning does well for problems that lend themselves to statistical analysis.

If machine learning is such a great solution, then why do many systems that claim to use it, often still perform badly? Why don't we yet see self-driving cars to be readily available, and why haven't repetitive manual tasks been all replaced by robots yet?

Well, the short answer is that aside from the real-world being a very complex place, machine learning still faces many problems and challenges.

An example of one such challenge is one of building a machine learning program that works too well with the training data. That is, one that fits the training data perfectly, and therefore cannot solve problems using new or unseen data. Experts call this overfitting and can be best understood as an over-calibration of the software. That is, the software is trained so well, that it looks for the exact pattern exhibited by the training data, instead of allowing for noise introduced by unseen data. The concept is best illustrated in figure below: here we see the performance of a given program that classifies some data. The y-axis indicates the number of false classifications, whilst the x-axis the time or number of training iterations run. We, therefore, see that, over time, the number of false classifications reduces when we use the training data (thick line). If the program were to behave correctly, then we would also see the number of errors reduce with time. But instead, after a certain number of training iterations, the number of false classifications increase (red line).



How can this be?

Well, consider a walk in the woods on a rainy autumn day. Everything is wet and the ground is muddy. You see two sets of footprints in front of you. The first of these footprints consists only of a rough outline and is all smudged. The print isn't very deep or well-formed and could belong to any grown adult. The second footprint goes deep into the mud. The profile of the boot that made it is clearly visible, the imprint is deep and the outline well-preserved. As you try to replicate the footprints, you will quickly see that the first, smudged print is quite easy to replicate. You tread lightly, walked quickly and maybe wiggle your foot a bit from side to side in the process.

The second footprint, however, will be very difficult to replicate. Unless you have the exact same boot, the exact same weight and the exact same shoe size than the person that walked before you, you are unlikely to be able to replicate this footprint.

What does any of this have to do with overfitting? Well, the precise, clearly formed footprint is the imprint left on the system by the training data when overfitting took place. The imprint left by the training data is perfect, and can only be matched if we knew data precisely matches the training data. Since this is rarely the case, predictions or classifications made by the overfitted system are likely going to be incorrect.

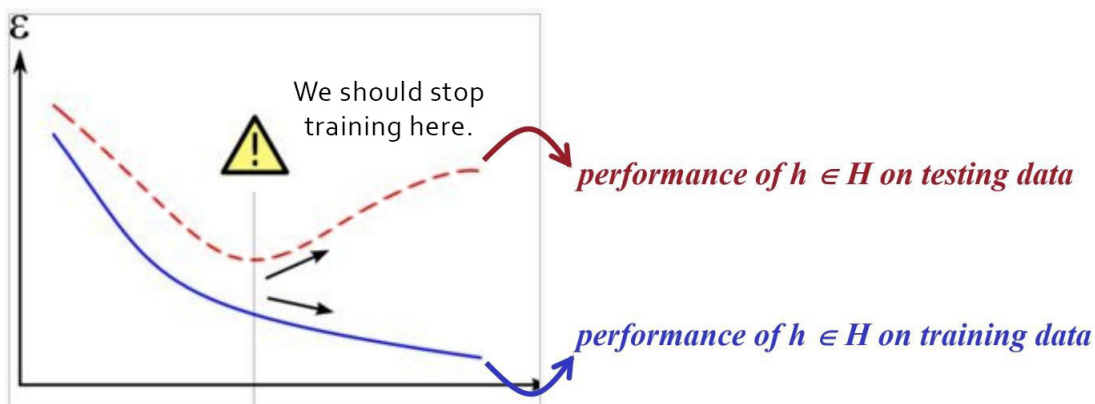
Therefore, instead of a perfect imprint, we want the training data to generate a more vague, general record of the pattern that it detected whilst training. If the imprint is too precise, then it will reflect only the

training data, but not the real world.

Luckily, there exist several ways of preventing overfitting. The first is to pay close attention to your training data, and ensure that the examples that you use for training, are an adequate representation of the "real world". Your examples should be adequately varied, to cover the possible scenarios that the program can encounter as best as possible.

In the case of the Patriot One Technologies threat detection system, this means training the system with examples that include as many different types of knives, guns and bombs as possible as well as without any threats at all. Furthermore, the sample images should record these objects at different angles and locations to capture as much noise and variation in size, distance etc as possible. By using only one size of weapon, one angle or one type, we will ensure a perfect "imprint" of this specific scenario, whilst preventing the system to recognize the threats when one or more of the aforementioned factors changes.

Another method of preventing overfitting, is to use statistical analysis to measure when to stop the training process. In other words: we split our examples into two different datasets. 80% of the examples we could use to train the system, whilst the remaining 20% we use to test how accurate the predictions are. If we reach a certain point in time during training after which the system starts to increase the number of incorrect predictions, we know that it is time to stop training. This notion is illustrated in the figure below.



Of course, there exist other ways, and even those described here are over-simplifications. Getting suitable training data, and thoroughly testing and training the system is a difficult task that should not be underestimated. Many times, neither of the two are financially feasible or realistic. As we previously discussed, often the right examples may exist in the real world, but are difficult to extract, collect or formalize (we can't stress this point enough). Data might be scattered across notes, emails, letters and different systems. Analyzing these different media, recording and organizing the required information into thousands and thousands of labelled training examples might simply not be possible.

The learning experience

The aforementioned issues are not the only problems encountered by machine learning experts. Many of the challenges around developing machine learning-powered systems are more abstract and require ways of looking at the world differently. For example, before arriving at the problem of overfitting, one common difficulty is defining the correct training experience to use in the first place.

Here, there exist three general categories, and all are fundamentally different.

The **first type of learning** is the one we have spoken about so far. It is called "supervised learning", which is a fancy name for just saying that your training data is labelled. We discussed the case of the threat detection system using supervised learning techniques, as the digital signatures/images that were fed into the system during training were labelled according to the threat which they represented.

Supervised learning has the obvious advantage of being able to produce correct and concise examples to use as the data is labelled. The obvious disadvantage of course is the fact that somebody needs to label the training data. In many cases, this involves people manually classifying, labelling or categorizing thousands upon thousands of data items, such as images or rows in Excel spreadsheets. Furthermore, by using labelling, we create a natural boundary as to the amount and type of knowledge that the system can learn. This might not be an issue for a specific problem at hand of course. Whether it is an issue or not depends largely on the problem.

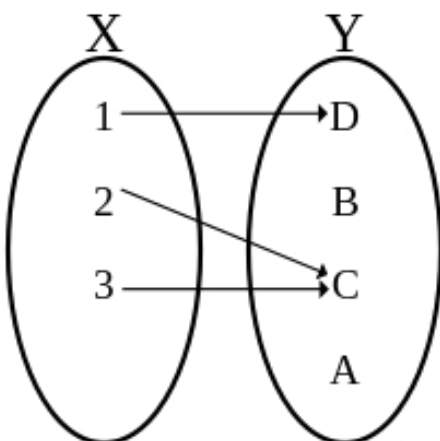
When hearing machine learning experts talk about supervised learning, they might give exotic-sounding definitions, such as:

"The problem of supervised learning involves learning a function from examples of its inputs and outputs"

They might also formally define supervised learning as:

"Given a collection of examples of f , return a function h that approximates f "

Both of the above statements were taken from the book "Artificial Intelligence - A Modern Approach" by Stuart Russell and Peter Norvig. To make sense of these statements, let's first explain what a function is. Technically speaking, a function is something that takes an input and maps it to an output (as is illustrated in figure 4.5 below)



Colloquially speaking, we could just call a function a "method" or a "formal way of doing something". This allows us to re-phrase "the problem of supervised learning involves learning a function from examples of its inputs and outputs" to: **by knowing what the expected output for a given input is, supervised learning recognizes patterns in the input data and uses these patterns to map unseen data to the desired outputs.** In other words, using labelled input data, a **hypothesis** about the data is formed (hence why the function is often labelled h in technical papers or books).

The **second "type" of learning** is called "unsupervised learning". Geoffrey Hinton, summarized this approach perfectly in just two sentences:

"When we're learning to see, nobody's telling us what the right answers are – we just look. Every so often, your mother says "that's a dog", but that's very little information."

Unsupervised learning is much closer to how humans learn. Most of the time, we learn by either observation or trial and error. But outside of a school, university or other teaching environments, we rarely learn by being presented with a list of examples along with their meaning or classification. Instead, we learn through observation. That is why we can often compare supervised learning to "book learning", and unsupervised learning to "the university of life".

With supervised learning, you are essentially taking a class with a teacher (the data doing the actual teaching), whilst with unsupervised learning, you are on your own. A surfer, for example, can learn which waves are worth catching and which waves are too steep, too flat or not worth catching, without ever needing to be presented with exact labels for each. Likewise, a farmer can identify what a good day or a bad day for working on the fields is by stepping outside his house in the morning. He developed a feeling for the weather, without ever being explicitly given a classification for it.

To put the two strategies into the perspective of our accompanying example (the threat detection system) we can summarize the two learning methods as follows:

With supervised learning, we are given examples of digital signatures or images, along with information as to what they represent, and we want to know what any unseen data represents. With unsupervised learning, on the other hand, we are given examples of digital signatures, but don't know what they represent. When running new data through the system, we simply want to categorize the data into one of 4 piles, each pile containing similar signatures.

The **third type of learning** strategies is called "reinforcement learning". This form of learning uses the **feedback** produced by the environment itself as a teacher. This way of learning is not concerned with identifying a pattern from a sample dataset but builds up knowledge by pairing up actions with positive and negative feedback.

A robot - such as the Roomba - trying to learn the layout of an apartment is a perfect example of where this learning strategy is employed in the real world. The robot itself is equipped with a sensor that sends a signal to the robot's "brain" as soon as it hits something hard. Furthermore, when the robot is first placed onto the apartment's floor, it knows nothing about its environment. No map, and no indication as to the size and shape of the apartment or the objects in it. The robot is turned on, and simply starts driving, recording its movements. As soon as it hits something hard, its sensors send a signal. This signal is a "feedback" that indicates something about the environment (in this case the fact that the robot can't proceed). Upon receiving the feedback, the robot records it in its robot brain, and then turns left, right or reverses to find a new path. Hence, over time, building a dynamic map of its surroundings.

The choice of which learning strategy to use can be difficult (sometimes mixes strategies are used), and primarily depends on three factors:

1. the nature of what is to be learned;
2. what type of feedback and performance measurements can be applied for learning; and
3. how the information learned can be modelled or represented.

Conclusion

At the beginning of this article, I wrote that “if there is only one message that you take away from this entire book, then let it be the following: **machine learning is glorified statistics**”.

As we come to the end of this article, I wanted to reiterate this statement, as it goes a long way towards understanding what machine learning really is, and what its limitations are.

Once you understand that machine learning solves problems primarily by identifying patterns in data, then you will quickly be able to see through any false claims or fishy marketing tactics when it comes to products who claim to use machine learning to make “smart decisions”. Furthermore, when you encounter a product or project that claims to use machine learning, ask yourself why it claims to use it. By using statistics and lots of data, we can build programs that learn and get better over time. Therefore, “learning” and improving over time, should be an integral part of the given product’s objective.

Aside from understanding what machine learning is, and how it differs to other problem-solving techniques, we explained some (not all) of the common problems and challenges faced by experts building machine-learning powered systems. These problems include choosing the right training experience, and getting the actual training process correct. Specifically, experts must be careful not to “over-train” their software (a process that, in technical jargon, is called “over-fitting”).

Equipped with a general knowledge of about machine learning, we are now ready to examine individual machine learning techniques in more detail. In other words, the upcoming articles in this series will become slightly more technical, as we will discover *how* exactly to “learn” from data.



Benjamin Jakobus

Author

Benjamin Jakobus is a senior software engineer based in Rio de Janeiro. He graduated with a BSc in Computer Science from University College Cork and obtained an MSc in Advanced Computing from Imperial College, London. For over 10 years he has worked on a wide range of products across Europe, the United States and Brazil. You can connect with him on [LinkedIn](#).



Technical and Editorial Review

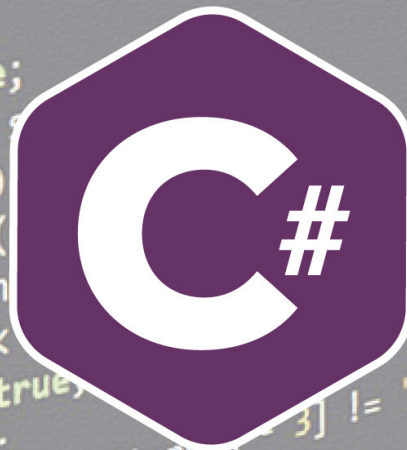
Suprotim Agarwal

COVERS C# v6, v7 AND .NET Core

THE ABSOLUTELY THE AWESOME

NOW INCLUDES
CHAPTERS ON
.NET Core 3.0
& C# 8.0

BOOK ON



AND

.NET

DAMIR ARH



Damir Arh

C#

Nullable Reference Types in Best practices

In this article, I look at the state of the nullable reference types feature one year after its initial release.

In recent years, there's been a trend of strict null checking in programming languages:

- TypeScript has the `strictNullChecks` option.
- In Kotlin (preferred language for Android development), all types don't allow null values by default.
- In Swift (Apple's language of choice), only the Optional data type allows null values.
- ...

Nullable reference types bring similar functionality to C#. The feature was originally planned for C# 7 and was finally released as part of C# 8.

Using nullable reference types

Since the introduction of generics in C# 2.0, value types can be declared nullable or non-nullable:

```
int nonNullable = null; // compiler error
int? nullable = null;
```

The `int?` notation is a shorthand for the `Nullable<int>` generic type which wraps a value type to allow assigning a null value to it.

In C# 8, nullable reference types use the same syntax to give the option of declaring reference types as nullable (i.e. allowing a null value) or non-nullable (not allowing a null value):

```
string nonNullable = null; // compiler warning
string? nullable = null;
```

Because of the language history, the decision to use the same syntax for value types and reference types changes the behavior of the language for reference types. Before C# 8, all reference types were nullable. They were however declared without a question mark:

```
// before C# 8
string nullableString;
// since C# 8
string nonNullableString;
```

To avoid such a breaking change in a new version of the language, the nullable reference types is the only feature of C# 8 that isn't enabled by default. An explicit opt-in is required for each project. The following property must be added to the project file for that:

```
<Nullable>enable</Nullable>
```

In recent versions of Visual Studio 2019, the option is also available on the *Build* page of the *Project Properties* window:

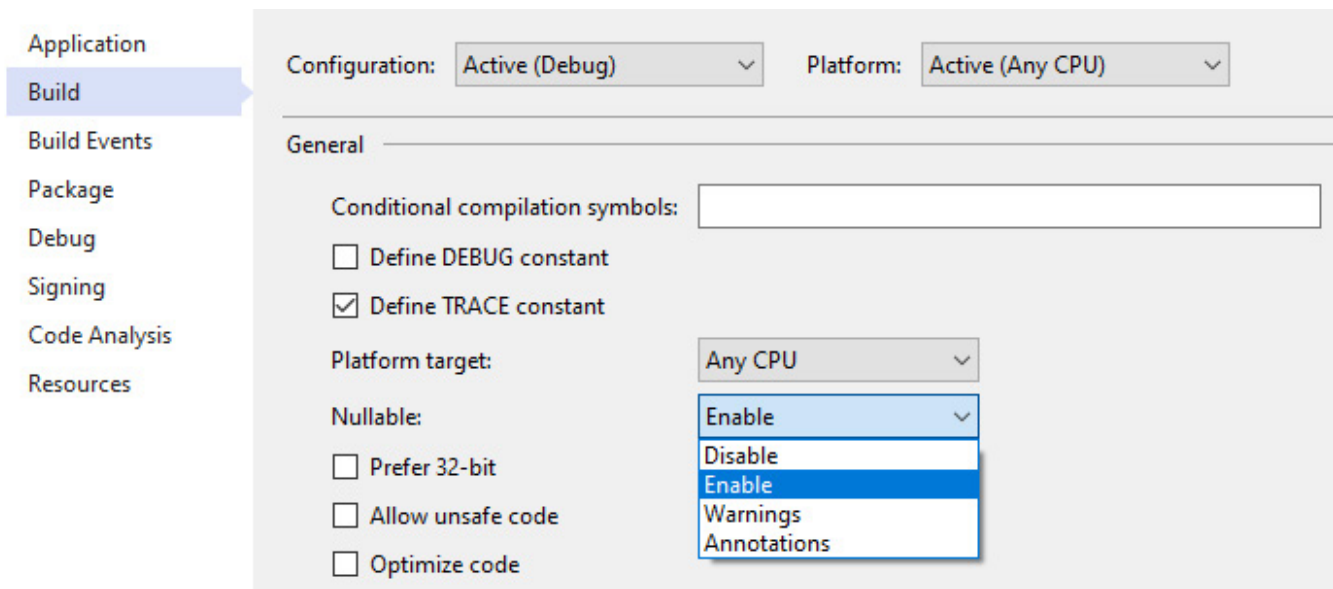


Figure 1: Nullable option on the Build page of the Project Properties window

The option is only available for projects using C# 8 or a later version. Different from the previous versions of the language, the support for C# 8 wasn't added to older .NET runtimes. The language is only supported with .NET Standard 2.1 and compatible runtimes: .NET Core 3.1 or newer, latest versions of Xamarin and Mono, and potentially future versions of UWP and Unity. There is no official support for any version of the .NET framework.

Static code analysis

There is an important difference in how null checking is implemented for value types and reference types. While assigning a null value to a non-nullable value type causes a compiler error, doing the same with a non-nullable reference type will only result in a warning. You can still treat it as an error using the *Treat warnings as errors* compiler option.

The static analysis will report warnings at compile time when a `NullReferenceException` might be thrown at run time, for example:

- When assigning a literal null value to a non-nullable variable:

```
string nonNullable = null
```

- When assigning a nullable variable to a non-nullable variable:

```
string? nullable = null;
string nonNullable = nullable;
```

When accessing members of a nullable variable:

```
var length = nullable.Length;
```

Of course, the warning won't be reported if you check for null before doing the potentially dangerous operation:

```
private int GetLength(string? nullable)
{
    if (nullable == null)
    {
        return 0;
    }

    return nullable.Length;
}
```

However, the static analysis is not perfect. Even if you get rid of all the warnings, there is still a possibility of a `NullReferenceException` being thrown at run time:

```
public class Person
{
    public Person(string firstName, string lastName, string? homeCountry = null)
    {
        FirstName = firstName;
        LastName = lastName;
        HomeCountry = homeCountry;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string? HomeCountry { get; set; }

    public void Relocate(string? country = null)
    {
        HomeCountry = country;
    }
}

var person = new Person("John", "Doe", "Unknown");
if (person.HomeCountry != null)
{
    person.Relocate(); // sets HomeCountry to null
    var countryLength = person.HomeCountry.Length; // no warning
}
```

Changes to objects in a different context (another method, different thread) are not detected. Although the above example is contrived, similar situations might occur in real code as it grows more complex.

Even if not all potential run-time exceptions are detected at compile time, there's still value in those that are. Each one of them could be a nasty bug that's now easy to fix.

The #nullable directive

When you enable the nullable reference types feature in a new project, fixing the warnings that appear as you write the code won't be a big deal most of the time.

On the other hand, if you enable the feature in an existing project with a lot of code, there is a strong possibility that an overwhelming number of warnings will be reported immediately. Fixing all of these might take a while.

To make the transition easier, a new `#nullable` directive has been introduced into the language. It can be used to enable or disable the nullable reference types feature inside a single source code file. For example, the feature can be fully disabled in a file if you put the following at the top of it:

```
#nullable disable
```

The directive supports three commands:

- `enable` enables the feature.
- `disable` disables the feature.
- `restore` reverts the feature to the project level setting.

Instead of enabling or disabling the feature in full, only part of it can be affected by adding another keyword at the end of the directive:

- `warnings` enables or disables the warnings emitted by static code analysis.
- `annotations` enables or disables support for declaring nullable or non-nullable reference types.

Hence, the following directive in a project with nullable reference types enabled will disable any warnings in the rest of the file (from the point where it is placed onward) but leave the ability to declare nullable or non-nullable reference types:

```
#nullable disable warnings
```

In my opinion, the `#nullable` directive can make it much more difficult to fully understand the behavior of the code in a project because the same exact code can have a different meaning based on the `#nullable` directives in the file:

```
#nullable enable annotations  
string doesntAllowNulls;  
#nullable disable annotations  
string allowsNulls;
```

It's difficult enough to switch between projects with the nullable reference feature enabled or disabled. Dealing with code in the same project (or even the same file) exhibiting different behavior because of the `#nullable` directive raises the complexity even more.

My suggestion is to avoid the `#nullable` directive as much as possible. While introducing nullable reference type into an existing codebase, you might need to place the directive at the top of some files with too many warnings. But as soon as possible you should remove the directives and fix those warnings instead.

Nullable reference types in libraries

To fully benefit from nullable reference types, it's important that any class libraries referenced by the project are also annotated for nullable reference types, i.e. they specify the nullability of reference types acting as their inputs and outputs.

Unfortunately, more than a year after the initial release of the feature that's not a given. Even in the .NET 5 base class library, [only 94% of the assemblies are fully annotated for nullable reference types](#). The

plan is to cover the remaining assemblies before the release of .NET 6 in November 2021. Of course, the percentage of third-party libraries with annotations for nullable reference types is even lower. The reason for that is additional work needed to add these annotations.

One aspect of this is the fact that it's in the best interest of library maintainers for their libraries to maintain compatibility with .NET Standard 2.0 (and consequently the .NET framework) even after implementing this C# 8 specific feature that depends on .NET Standard 2.1.

Fortunately, there is an officially supported way to add annotations for nullable reference types to a .NET Standard 2.0 library by manually adding the following property to the project file to enable the use of C# 8:

```
<LangVersion>8.0</LangVersion>
```

There are still some C# 8 features that won't work in such a library (asynchronous streams, for example). But most importantly, nullable reference types will be fully supported. Such a library will still work with the .NET framework as if nullable reference types weren't used in it. But any .NET Standard 2.1 compatible projects (e.g. .NET Core 3.1. or Xamarin) will get full information about the nullability of reference types.

Let's look at the following code snippet using the well-known [Json.NET library](#) to see how beneficial this can be:

```
var player = JsonConvert.DeserializeObject<Player>(null);
```

The library introduced annotations for nullable reference types in version 12. With version 11, there will be no warning for the above line of code even if nullable reference types are enabled in the consuming project. Still, at run time the code will throw an `ArgumentNullException`. However, with version 12, the same line of code will result in a warning at compile time making it easier to detect and fix the bug. However, even in version 12, code analysis won't detect all potential `NullReferenceExceptions` thrown at run time. Let's look at the next two lines of code, for example:

```
Player player = JsonConvert.DeserializeObject<Player>("null");  
var username = player.Username;
```

In the first line, a null value will be assigned to a non-nullable reference type but there will be no compile-time warning. Neither will there be one in the second line, although a `NullReferenceException` will be thrown at run time.

The only way to get a compile-time warning would be to modify the first line of code as follows:

```
var player = JsonConvert.DeserializeObject<Player?>("null");  
var username = player.Username;
```

Now, there will be a warning in the second line of code when trying to access a member of `player` without first testing its value for null. The question is, will you always use a nullable type as the generic type argument if the compiler doesn't warn you about it?

Annotation attributes

There is a way to add such warnings, though. But not in a .NET Standard 2.0 library. .NET Standard 2.1 adds a new generic constraint and a set of annotation attributes to describe the use of nullable reference types in more detail.

We will use some of them to describe the following wrapper for the `Json.NET DeserializeObject` method from above:

```
public static T DeserializeObject<T>(string json)
{
    return JsonConvert.DeserializeObject<T>(json);
}
```

First, we can add a constraint for the generic type argument `T`. Unfortunately, there's no way to require it to be a nullable reference type. So instead, we can require it to be a non-nullable reference type:

```
public static T DeserializeObject<T>(string json) where T: notnull
{
    return JsonConvert.DeserializeObject<T>(json);
}
```

While this constraint can be useful in certain scenarios, it doesn't solve the initial problem of the missing warning. Even worse. It prevents the consuming code from using a nullable reference type as the generic type argument to get the compiler warning. But it does ensure that the type argument will always be non-nullable. In combination with an appropriate annotation attribute, the final goal of having a compile time warning can still be achieved:

```
[return: MaybeNull]
public static T DeserializeObject<T>(string json) where T: notnull
{
    return JsonConvert.DeserializeObject<T>(json);
}
```

When applied to the return value, the `MaybeNull` attribute specifies that the return type value will be a nullable reference type even if the generic type argument is a non-nullable reference type. Because of that, failing to do a null check before accessing the members of the returned value or assigning that value to a non-nullable variable will result in a compile-time warning.

This is just one of the available annotation attributes for describing the reference type nullability in more detail. They can be grouped in two categories:

- Describing the output values: `MaybeNull` and `NotNull` have the exact opposite meaning. Conditional variations for both are also available: `MaybeNullWhen`, `NotNullWhen`, and `NotNullIfNotNull`.
- Describing the input values: `AllowNull` and `DisallowNull`.

Examples of use for each attribute are available in [the official documentation](#).

Support in Entity Framework Core

Of all libraries with support for nullable reference types, Entity Framework Core might be the one with the most impact on the code you write. All the specifics are [well documented](#). I'm just going to point out the most important parts that you need to be aware of.

In the `DbContext` class, the `DbSet` properties for individual tables should be non-nullable because the base class constructor will ensure that they are always initialized. However, the following line of code will result in a compile-time warning due to uninitialized non-nullable property:


```
public DbSet<Player> Players { get; set; }
```

To tell the compiler that the value is initialized without initializing it yourself, the null-forgiving operator ! can be used:

```
public DbSet<Player> Players { get; set; } = null!;
```

The modified line of code now initializes the property with the null value which on its own would still cause a warning. The null-forgiving operator tells the compiler to ignore that warning.

Although the null-forgiving operator can be used anywhere to override the static code analysis findings, it shouldn't be abused unless you're certain that the code analysis is mistaken, and you know better. Otherwise you'll just get an exception at run time instead of the compile-time warning.

In combination with Entity Framework Core, the null-forgiving operator is also useful for non-nullable navigation properties in entity classes which are again initialized by the library code:

```
public Country HomeCountry { get; set; } = null!;
```

The regular non-nullable properties should be initialized in the only public class constructor:

```
public Player(int id, string username, string emailAddress)
{
    Id = id;
    Username = username;
    EmailAddress = emailAddress;
}
```

This is important because entity classes are also instantiated in code and that's the only way to ensure that all properties are initialized. For records from the database, Entity Framework Core would initialize the properties anyway by assigning values directly to them. But it can also use the constructor for that when present.

The final and probably most important detail to be aware of is the meaning of non-nullable reference types for properties. Just like non-nullable value types, they result in a non-nullable database column. This means that the **Required** attribute is not needed anymore. Instead, the type of the property must be either nullable or non-nullable.

Special care must be taken when introducing nullable reference types in an existing codebase. Look at the following property:

```
public Country HomeCountry { get; set; }
```

Before nullable reference types, it would mean a nullable database column because there's no **Required** attribute on it. After enabling the nullable reference types, the database column will become non-nullable because the same syntax now means a non-nullable type. To keep the same database model, the type should be changed to a nullable reference type:

```
public Country? HomeCountry { get; set; }
```

Fortunately, when generating a new migration, there will be a warning because of the column data type change in case you forget to make this modification in code. Still, it requires you to pay enough attention

and review the generated migration. But you should be doing that anyway.

Conclusion

In this article, I looked at the experience of using nullable reference types one year after the initial release. I started with the basics, describing how the functionality works. I continued with a closer look at the `#nullable` directive as a tool for incremental introduction of nullable reference types into existing code. I explained how class libraries can provide information about nullable reference types to the consuming code and concluded with a closer look at support for nullable reference types in Entity Framework Core.

Although using nullable reference types can introduce its own set of problems, I still think it's beneficial because it helps you find potential bugs and allows you to better express your intent in the code. For new projects, I would recommend you enable the feature and do your best to write code without warnings. It shouldn't be too difficult, and your code will be better because of it. Enabling the feature in existing code will be more challenging and might not be worth it, especially if you don't plan to do much new development.



Damir Arh
Author



Damir Arh has many years of experience with software development and maintenance; from complex enterprise software projects to modern consumer-oriented mobile applications. Although he has worked with a wide spectrum of different languages, his favorite language remains C#. In his drive towards better development processes, he is a proponent of Test-driven development, Continuous Integration, and ContinuousDeployment. He shares his knowledge by speaking at local user groups and conferences, blogging, and writing articles. He is an awarded Microsoft MVP for .NET since 2012.



Technical Review
Yacoub Massad



Editorial Review
Suprotim Agarwal

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

THE ABSOLUTELY AWESOME

BOOK ON



AND
.NET

DAMIR ARH

Features

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

It's got it all!

Crack your next .NET Interview

Build a Solid Foundation

Strengthen Concepts

THE ABSOLUTELY AWESOME BOOK ON
C# and .NET

ORDER NOW !

PDF, EPUB and MOBI



CONTINUOUS DEPLOYMENT FOR SERVERLESS APPLICATIONS ON AZURE

Code created for every application, needs to be kept at a shared location (source control), where it can be built upon. This is usually a server build with all the latest code. Once the build is successful, the artifact required for deployment to the target is ready. And finally, we can deploy it to a Cloud platform (like Azure) using Release Pipeline.

There can be various stages involved in this process like Dev, QA, Staging and Production. In this tutorial, I will discuss the following features for serverless applications using Azure and Azure DevOps:

1. How to create and work with Serverless Applications
2. How to use Azure DevOps as Source Control
3. How to use Azure Pipelines for CI and CD
4. How to use two Serverless apps - Function App and Logic App

Overview of Azure

Microsoft Azure (formerly called as Windows Azure) is a suite of cloud services. These services range from compute, storage, analytics, networking, containers and so on.

Users can select a service of their choice suiting their project/business requirement. This choice can be based on various factors like scalability, flexibility and cost. Azure also provides us a way with which we can integrate with on-premise applications (hybrid applications). With Azure, we can start with a small application and continue to increase its scale and complexity as and when required.

Azure provides us services in following broad categories : Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

The following table shows how the difference service are managed:

	IaaS	PaaS	SaaS
Application	Application	Application	Application
Data	Data	Data	Data
Operating System	Operating System	Operating System	Operating System
Virtualization	Virtualization	Virtualization	Virtualization
Servers	Servers	Servers	Servers
Storage	Storage	Storage	Storage
Networking	Networking	Networking	Networking
Examples of services	Virtual Machines	Web App Service	Azure DevOps, Office 365
	Managed by you	Managed for you	

In case you are using an online set of features, you need to manage them including Application, Data, OS, Virtualization, Storage etc. on your own.

In the table, I have not considered Serverless Applications. Let us first take an overview for the same.

Overview of Serverless Application

With Serverless applications, developers can focus on writing code and not have to worry about:

- Infrastructure procuring

- Provisioning of resources
- Management of resources

Serverless apps allows us to write, build and deploy applications which can be scaled up or down as per requirement.

We also have the option of a hybrid solution - some part of the application like database can be on-premises and other parts like shopping cart can be on the cloud. Azure being a truly platform agnostic cloud, for Function app, allows we can write code using C# (.NET platform or .NET Core), Java, Python or PHP.

Azure Function App

Function App works as a container for storing multiple Azure Functions.

- Every function can be triggered by a specified trigger which is automatically triggered when the event occurs.
- Every function can have input as well as output bindings.
- Third party assemblies or packages can be easily referenced
 - NuGet packages can be used by adding a file named function.proj and referring to required packages via it.
- Monitoring and managing these functions is easy.
- Stateless Functions, but if required, state can be maintained by using Durable Functions.
- 3 levels for Authorization:
 - Anonymous
 - Function
 - Admin
- Security can be enforced using:
 - RBAC (Role Based Access Control) support
 - Adding Access Policy to Azure Key Vault
 - CORS (Cross Origin Support) to provide access to another domain if needed.
 - Using APIM (Azure API Management) to authenticate the requests and restrict access to the functions

Function App workflow is pretty straightforward - whenever a trigger is invoked, the function gets executed.

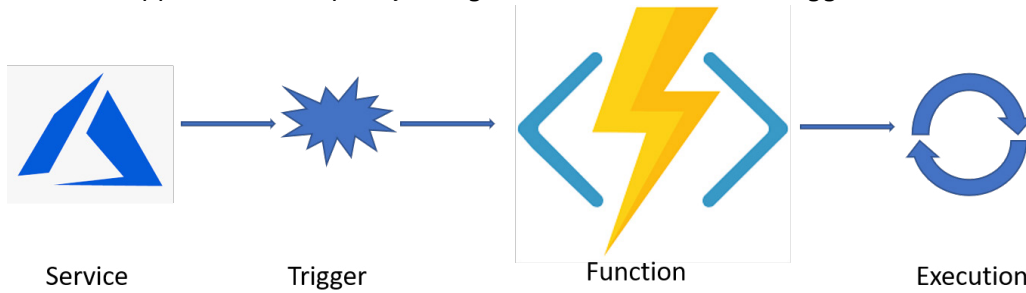


Figure 1: Working of an Azure Function

The Triggers for an Azure Function are:

- Http
- Timer
- Cosmos DB
- Blob
- Queue
- EventHub
- ServiceBus

Creating and Using Function App

Let us find out how to write a small function. We are using [Visual Studio Code](#), a free IDE from Microsoft. Visual Studio Code is getting feature rich day by day with a vast set of extensions and writing code using it is fun.

In order to see how a complete Continuous Integration (CI) and Continuous Deployment (CD) pipeline works, I will use Azure DevOps for Source Control and Pipelines for creating the required definition. You need an account to use Azure as well as to use Azure DevOps. You can create these by going to [Azure Link](#) and [Azure DevOps Link](#) (select the *Start Free* button and not *Start free with GitHub*).

A free account on Azure will be valid for a month after you provide the necessary information (you need to provide valid credit card details, but that is just to check for card validity and not to charge you). The Azure DevOps account is always free (you do not have to provide any payment information).

Let us create a Team Project in your Azure DevOps account.

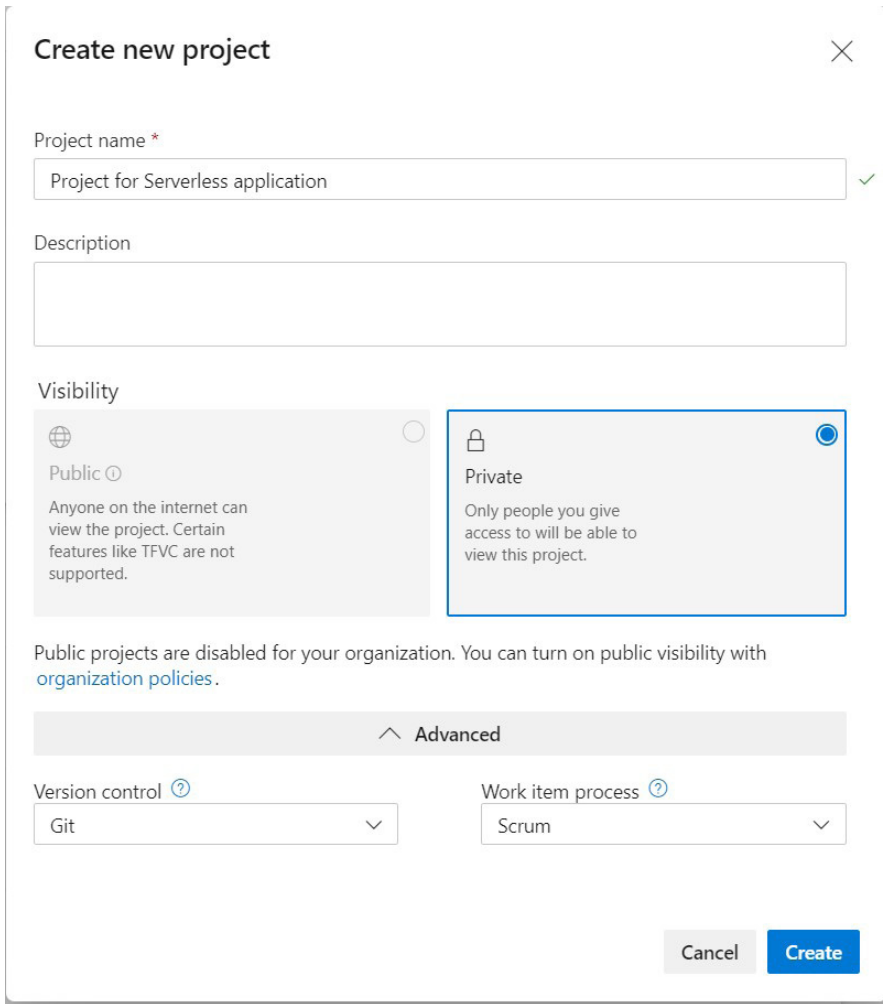


Figure 2: Create Team Project in Azure DevOps

Writing Code for Azure Function

Now provide the option for keeping our code in source control in the Team Project we created earlier. Unfortunately, there is no straight forward way to do it (because we are using Visual Studio Code). Make sure that you have installed git for windows on your machine and then start Visual Studio Code.

Download VS Code. Make sure that you have added the extension for Azure and create a project. Provide the necessary information and voila your first HTTP Trigger function code is ready. See Figure 3.

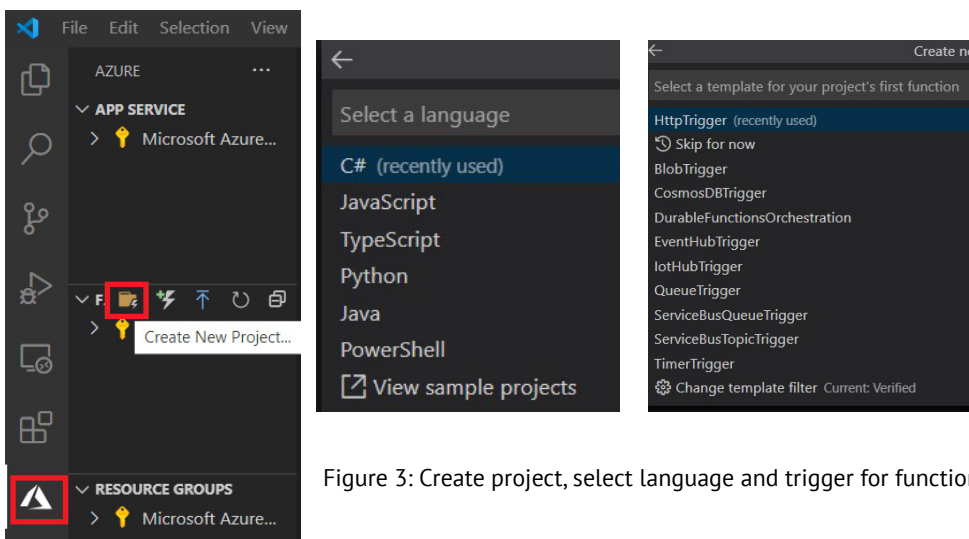
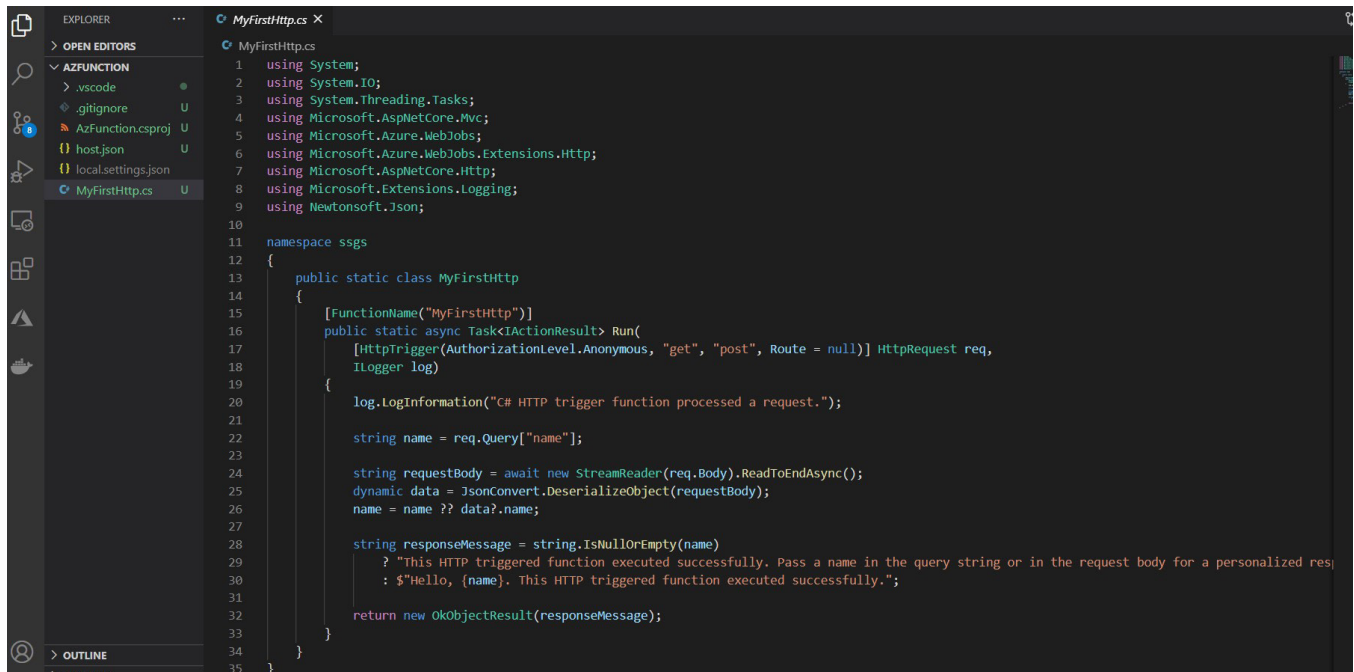


Figure 3: Create project, select language and trigger for function

The following code is automatically added to the function:



```
1 using System;
2 using System.IO;
3 using System.Threading.Tasks;
4 using Microsoft.AspNetCore.Mvc;
5 using Microsoft.Azure.WebJobs;
6 using Microsoft.Azure.WebJobs.Extensions.Http;
7 using Microsoft.AspNetCore.Http;
8 using Microsoft.Extensions.Logging;
9 using Newtonsoft.Json;
10
11 namespace ssgs
12 {
13     public static class MyFirstHttp
14     {
15         [FunctionName("MyFirstHttp")]
16         public static async Task
```

Figure 4: Code for function created with Visual Studio Code

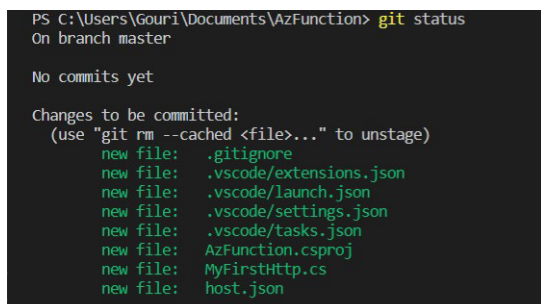
If you want to run the code locally, install [Azure Function Core Tools](#) and click on F5. The code will get built and you will get a url to view the function.

Click on Ctrl + Link to execute and view the function code in browser. Provide the parameter to view the execution. This will help in finding out if there are any changes required in the code before we actually put it in Azure.

Use Source Control from Azure DevOps

Let us add this code to source control. Start the terminal window and enter the command “*git init*” to initialize the repo for git and then “*git add*”.

The status of the operation can be checked by using “*git status*” command.



```
PS C:\Users\Gouri\Documents\AzFunction> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   .gitignore
    new file:   .vscode/extension*.json
    new file:   .vscode/launch.json
    new file:   .vscode/settings.json
    new file:   .vscode/tasks.json
    new file:   AzFunction.csproj
    new file:   MyFirstHttp.cs
    new file:   host.json
```

Figure 5: Local Git Status

Let us commit the code by giving the command “*git commit -m '<commit message>'*” If you now run the *git status* command again, you will get a message saying that there is nothing to commit and we are currently on the master branch. To know about more details about git, please use the [link](#).

Remember we have just added our code to local git and not yet to our Team Project to Azure DevOps (remote). This will require you to install extension for Azure DevOps by running the following command:

```
az extension add --name azure-devops
```

Now provide commands to connect to the Azure DevOps Team Project and commit all the code to the repository:

```
git config --global credential.helper wincred
git config --global user.name "Gouri Sohoni"
git config --global user.email <email>
az devops configure --defaults organization=https://dev.azure.com/<org
name>project=<Team Project Name>
git remote add origin <Clone Repo URL>
git push --set-upstream origin master
```

We can go to the Team Project in Azure DevOps and confirm that the repository is populated.

Once done, we need to create a pipeline which will deploy our code to Azure Function App.

Creating Azure Function App

We have already created an account on Azure Portal. Browse to <https://portal.azure.com> and login with the user name used earlier. Create a new resource from Home and search for function app.

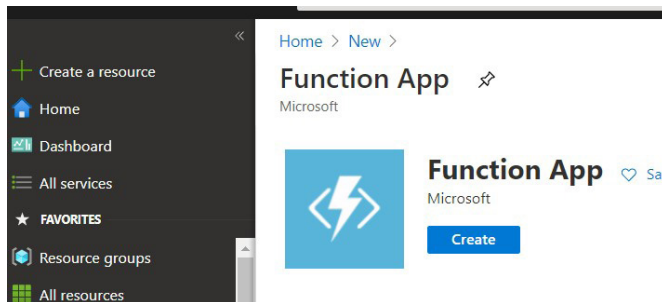


Figure 6: Create Function App in Azure

Provide information for resource group, name of function app, location, runtime stack and other details.

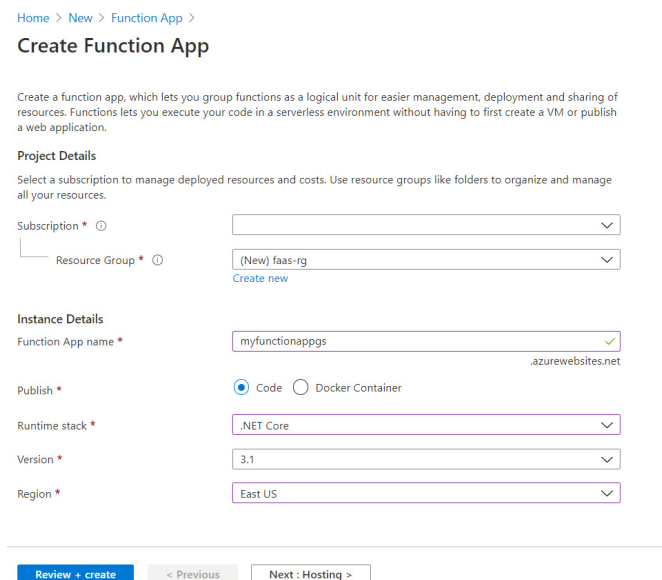


Figure 7: Configuration for Function App

Keep all the remaining options as default and click on Review + create.

Resource Group is a container which can group related resources on Azure. Each resource can be a part of a single resource group. Every resource in Azure is first reviewed for the configuration specified and then a json template is created for it. In case of Azure Function, the storage account associated with the function app, the app plan we specified with it – all of this becomes part of the same resource group.

Function App provides us with Deployment Centre for direct deployment from different sources. This feature is also available with Web App. It is a very useful feature which will implement continuous deployment. It comes in two flavors with Kudu and using Azure Pipelines. The build and release pipelines will get created with Azure Pipelines.

Once the function app is created, provide deployment for it by directly selecting Deployment Centre blade which can be seen in Figure 8.

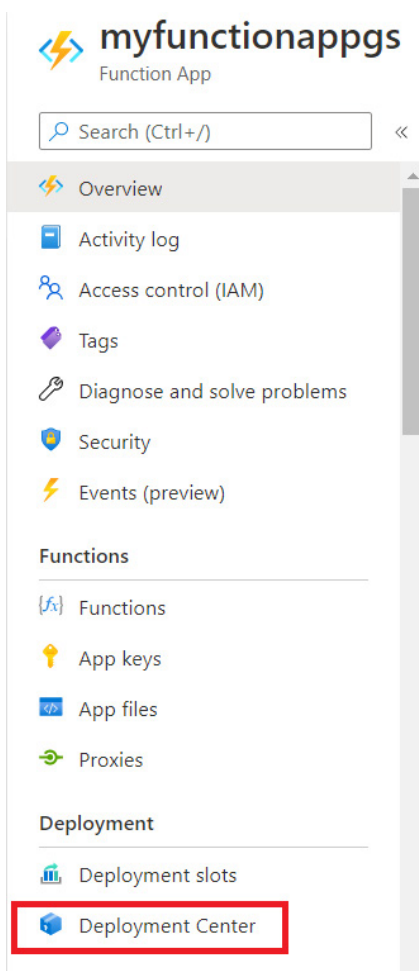


Figure 8: Deployment Center Blade in Function App

Pipelines creation for Code Deployment

Provide the necessary information like Source Control type, name of Team Project to generate Pipelines. Select Azure Repos , select Azure Pipelines and click on continue.

Deployment Center

App Service Deployment Center enables you to choose the location of your code as well as options for build and deployment to the cloud. [Learn more](#)



Continuous Deployment (CI / CD)

Three cards are displayed for selecting a source control provider:

- Azure Repos:** Configure continuous integration with an Azure Repo, part of Azure DevOps Services (formerly known as VSTS).
- GitHub:** Configure continuous integration with a GitHub repo. Status: Not Authorized.
- Bitbucket:** Configure continuous integration with a Bitbucket repo. Status: Not Authorized.

A blue "Continue" button is located at the bottom center.

Figure 9: Deployment Centre for Azure Function App

Select Azure Pipelines in the next screen and click on continue.

Deployment Center

App Service Deployment Center enables you to choose the location of your code as well as options for build and deployment to the cloud. [Learn more](#)



Two cards are displayed for selecting a build provider:

- App Service build service:** Use App Service as the build server. The App Service Kudu engine will automatically build your code during deployment when applicable with no additional configuration required.
- Azure Pipelines (Preview):** Configure a robust deployment pipeline for your application using Azure Pipelines, part of Azure DevOps Services (formerly known as VSTS). The pipeline builds, runs load tests and deploys to...

"Back" and "Continue" buttons are located at the bottom.

Figure 10: Deployment Centre for Azure Function App (cont.)

This option is available only for the Git (distributed) repository and not for TFVC (centralized).

Editorial Note: To know more about the different repos for Azure DevOps, read [Migrating Code to Azure DevOps Repos \(4 Different Scenarios\)](#).

Select the organization name and also the Team Project name:

Figure 11: Deployment Centre for Azure Function App (cont.)

View a summary for all the configurations we selected and click on Finish.

Azure Deployment Centre makes our job quite simple as we do not have to create build and release pipelines in Azure DevOps. If needed, we can also do it on our own by creating build and release pipelines by going to Azure DevOps.

Ensure that the build and release has been successfully triggered. If the account you are using for Azure Portal is different than the one for Azure DevOps, you need to create a connection. You can find more information about it via this [link](#).

There is a small hitch though. It keeps the function in read-only mode. There are a few ways to overcome this problem, one involves changing the build definition to add one more task of build for .NET Core as shown in Figure 12.

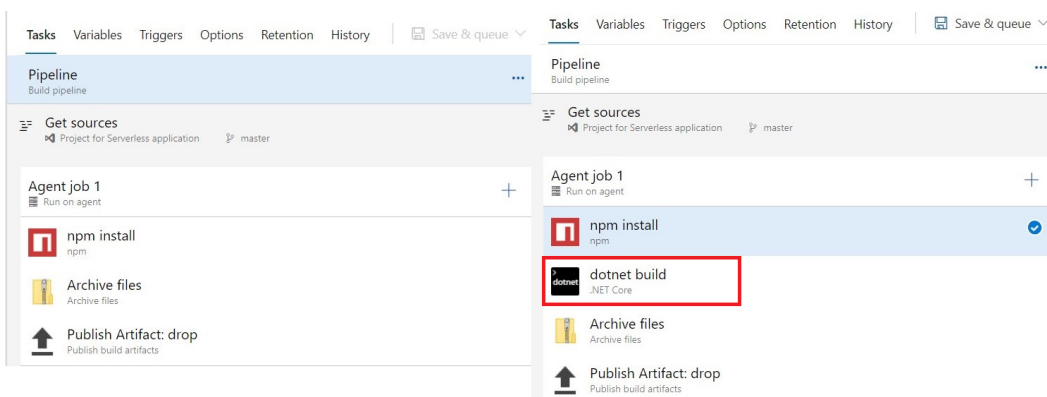


Figure 12: Build Pipeline

If we do not add this task, we will not be able to view the function. It gives following message:

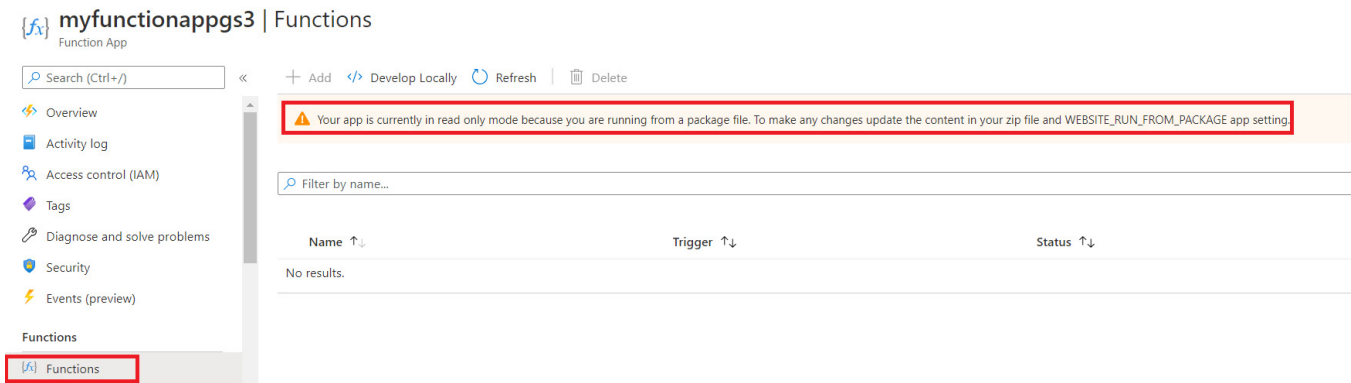


Figure 13: Warning for Deployment Centre

The other way to overcome this problem is to use Azure CLI or Azure PowerShell to deploy the function.

After successfully adding the specified task to the build definition, things should work as expected and the function gets added to the function app. This can be triggered using the url provided in Code + Test tab as follows:

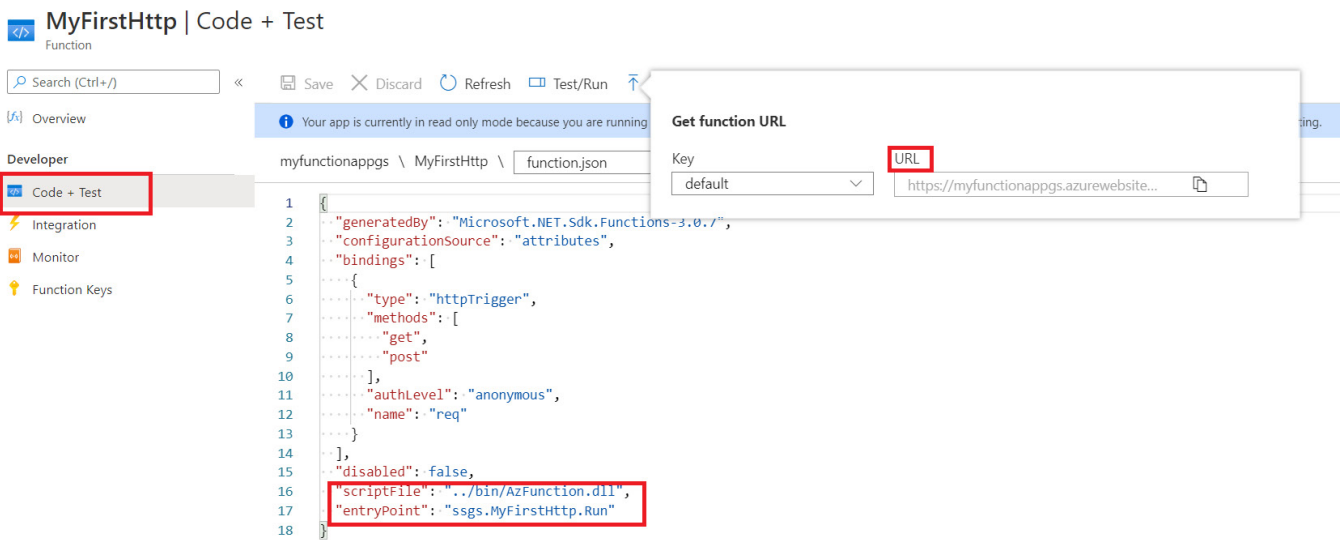


Figure 14: Trigger your Http Trigger function

You can view the script file and entry point added to the function.json file.

The other way for deployment with Deployment Centre is to use the option “App Service build service” which in turn uses Kudu for deployment.

Azure KUDU is one of the fastest and easiest ways of deploying the web site components to the Azure website or components for function app. It in turn uses Source Control with Git. We do not need to provide any other configuration except to select the organization, Team Project, repo and branch.

Deployment Center

App Service Deployment Center enables you to choose the location of your code as well as options for build and deployment to the cloud. [Learn more](#)

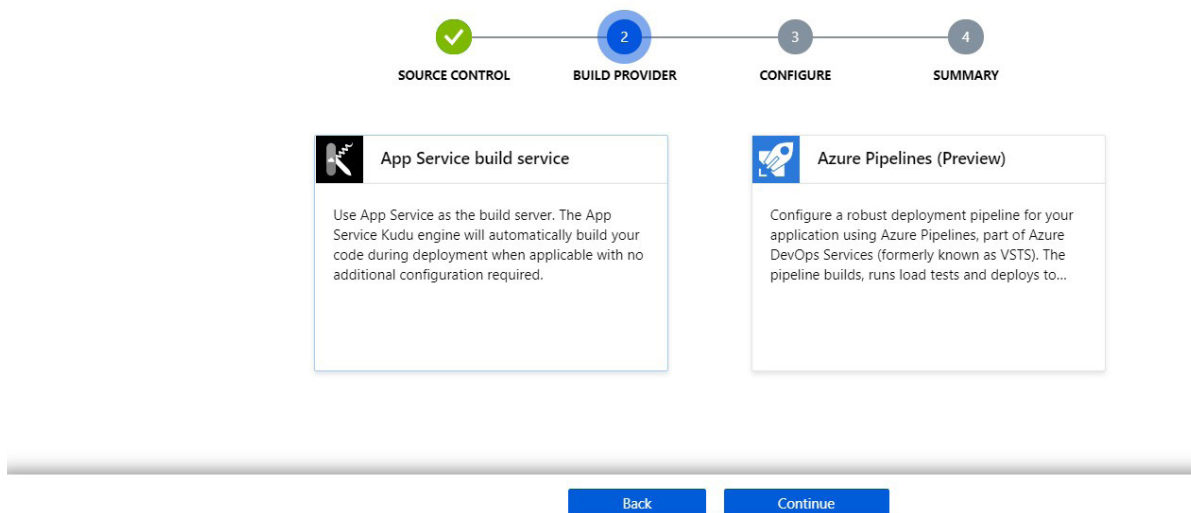


Figure 15: Deployment Centre with Kudu

When you use the Deployment Centre or directly use Visual Studio or Visual Studio Code for deployment of functions, we cannot view the code in Azure Portal for the function. Azure automatically adds PAT (Personal Access Token) to take care of the deployment.

The Azure Portal also provides us with a simple IDE to create a function with the specified trigger. I have created another function app in the same resource group to show as a demo. You can select the Function blade and click on Add to do the same.

For every function, a `run.csx` and `function.json` file gets added automatically. Any assembly can be referenced using `#r` and another `csx` file can be loaded using `#!` (`#!load`). We can add support to NuGet packages by adding a file named `function.proj` with the required packages referenced.

Editorial Note: *Here's a quick developer reference guide if you are absolutely new to this process* <https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-csharp>

I have added NuGet packages to Azure Functions and have found out some key points (comparing with the earlier version of Functions).

1. In earlier versions, the file was named as `project.json` and not `function.proj`. It still works very well with one of our extensions we had created for Azure DevOps
2. When you add the `function.proj` file with required references to NuGet packages, the code gets built automatically and another file named `project.assets.json` gets created based on the assemblies referenced. If for some reason this doesn't happen, then add the following configuration level app settings (earlier it used to happen more seamlessly, and then people have suffered while adding reference to NuGet packages, especially when adding reference to third party package named SixLabors)

- name: `DOTNET_ADD_GLOBAL_TOOLS_TO_PATH` Value: `false`
- name: `DOTNET_SKIP_FIRST_TIME_EXPERIENCE` Value: `true`

A small example of function.proj file is as shown here (for using Azure blob storage)

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Azure.Core" Version="1.3.0" />
    <PackageReference Include="Azure.Storage.Common" Version="12.4.3" />
    <PackageReference Include="Azure.Storage.Blobs" Version="12.4.4" />
  </ItemGroup>
</Project>
```

Azure Functions - Best practices

Now that we have almost come to the end of this tutorial, let's quickly see some best practices for creating Azure Functions.

- Avoid long running functions
- Write Stateless function
- Use durable functions to manage state if required to keep state
- Write defensive code
 - o Take care of any error encountered in earlier execution
- Use async code to avoid blocking

Now that we have seen how CI and CD can be used with Function App, let us quickly do the same for Logic App.

Azure Logic App

Logic App and Function App are similar as far as serverless architecture is concerned. The difference is that Logic App provides workflow, whereas Function App supports compute service.

Azure Logic App	Azure Function App
Workflow based	Our own code can be written
Automated business processes	No automation, we need to write our own
Better suitable for integration solutions	Suitable for more flexibility

There are a lot of connectors supported with Logic App with the option of creating a custom connector if required. With Logic App, we can create an application and use release pipeline to directly deploy it to Azure. For example, I have created another repository in the same Team Project I had created in Azure DevOps. Let's see how to create a Logic App.

Use Visual Studio 2019 to create Logic App

I am going to use [Visual Studio 2019](#) to create a logic app. You can add an extension for Logic App to Visual Studio Code and use it to create a Logic App. Start Visual Studio and select the resource group template. Make sure that you select the designer for logic app - it will ask for the subscription and details like name of resource group etc.

I have created a small fun workflow in a blank project where if an RSS feed is received, then the feed title and summary is sent to a queue created in storage account. The workflow looks as follows.

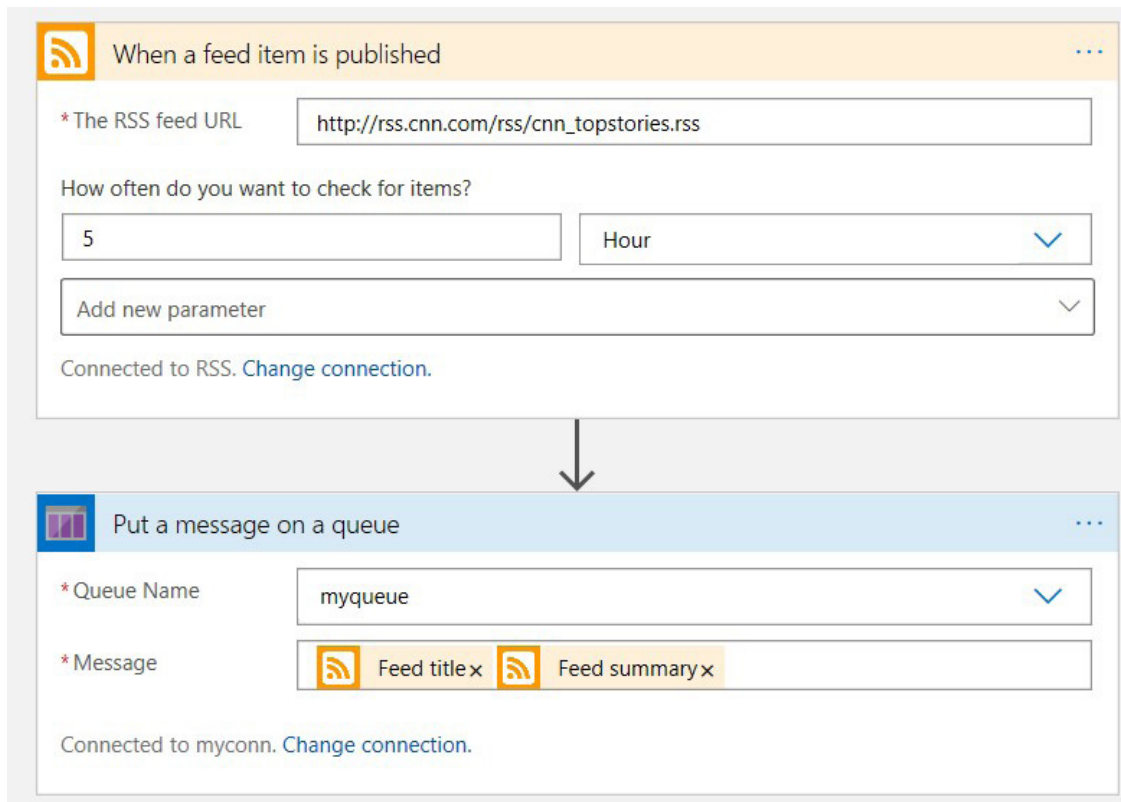


Figure 16: Workflow for Logic App

As I have already mentioned, every Azure resource has a json based template created (template.json). For the parameters in the resource, parameters.json file gets created. We will use these two files for deployment.

Now we need to send this code to source control. Follows the steps as shown earlier to commit to the remote repo using git commands. Please provide the values for the two keys for the storage account which can be copied from the Access Keys blade in Storage Account.

The code looks as follows:

```

1 {
2   "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
3   "contentVersion": "1.0.0.0",
4   "parameters": {
5     "logicAppName": {
6       "type": "string",
7       "minLength": 1,
8       "maxLength": 80,
9       "metadata": {
10        "description": "Name of the Logic App."
11      }
12     },
13     "logicAppLocation": {
14       "type": "string",
15       "minLength": 1,
16       "maxLength": 80,
17       "metadata": {
18        "description": "Location of the Logic App."
19      }
20     }
21   }
22 }

```

Figure 17: Template for Logic App

Deploying to Azure

Now we just need to create a release pipeline which takes the source code as artifact and also enables the Continuous Deployment trigger for it. So, if and when you do any changes to the json file, the changed Logic App will be deployed. We use the ARM Template deployment task to achieve this. The ARM template task will in turn use the json files as shown in Figure 18.

Figure 18: ARM Deployment for Logic App

Trigger CD for Azure Logic App

Change some value in the json file by modifying the code in Visual Studio 2019. Commit and push the changes directly to the remote repo. Observe how the deployment automatically starts!

Verify that the Logic App is successfully deployed by using the Azure Portal.

Summary

Azure provides us with various services for various requirements. In this tutorial, I discussed what are serverless applications, and their uses. Function App and Logic App were used as examples to find out how to create, use source control and how to continuously deploy services to Azure.

With these fundamentals in place, I hope you will be able to create and deploy Serverless applications with ease!



Gouri Sohoni
Author



Gouri Sohoni is a Trainer and Consultant for over two decades. She specializes in Visual Studio - Application Lifecycle Management (ALM) and Team Foundation Server (TFS). She is a Microsoft MVP in VS ALM, MCSD (VS ALM) and has conducted several corporate trainings and consulting assignments. She has also created various products that extend the capability of Team Foundation Server.



Technical Review
Subodh Sohoni



Editorial Review
Suprotim Agarwal



et curry.com

**Want this
magazine
delivered
to your inbox ?**

Subscribe here

www.dotnetcurry.com/magazine/

* No spam policy

Thank you

FOR THE 49th EDITION



@damirarh



@dani_djg



 benjamij



@yacoubmassad



@gouri_sohoni



@darrengillis



@subodhsohoni



@suprotimagarwal



@saffronstroke

WRITE FOR US

<mailto:suprotimagarwal@dotnetcurry.com>