**SignalR Core and SqlTableDependency - Part Two**

August 16, 2017 | 21 Minute Read

**Note:** I strongly recommend you to read [this post](http://elvanydev.com/SignalR-Core-Alpha/) when you finish reading this one, in order to get know the latest changes with the new SignalR Core Alpha version.

In the [previous post](http://elvanydev.com/SignalR-Core-SqlDependency-part1/) we talked about the things what doesn’t support anymore, the new features and SignalR Core’s Architecture. We realized that SignalR Core’s building block is Asp.Net Core Sockets and now SignalR Core doesn’t depends on Http anymore and besides we can connect through TCP protocol. In this post we gonna talk about how SqlDependency and SqlTableDependency are a good complement with SignalR Core in order to we have applications more reactive. Finally I’ll show you a demo using [.NET Core 2.0 Preview 1](https://www.microsoft.com/net/core/preview#windowscmd) and Visual Studio 2017 [Preview version 15.3](https://www.visualstudio.com/vs/preview/)

**SqlDependency**

In a few words SqlDependency is a SQL Server API to detect changes and push data from data base and it’s based on SQL Service Broker. You can take a look [this basic example](https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/detecting-changes-with-sqldependency).

**SqlTableDependency**

SqlTableDependency is an API based on SqlDependency’s architecture that improves a lot of things. SqlTableDependency’s record change audit, provides the low-level implementation to receive database notifications creating SQL Server trigger, queue and service broker that immediately notify us when any record table changes happen. You can read more about SqlTableDependency [here](https://github.com/christiandelbianco/monitor-table-change-with-sqltabledependency)

*SqlTableDependency is not a wrapper of SqlDependency.*

As I said earlier, SqlTableDependency has a lot of improvements over SqlDependency, some of the coolest ones are:

* Supporting Generics
* Supporting Data Annotations on model
* Returning modified, inserted and deleted values
* Specifies column’s change triggering notification

**Demo**

**Prerequisites and Installation Requirements**

1. Install [.NET Core 2.0 Preview 1](https://www.microsoft.com/net/core/preview#windowscmd)
2. Install Visual Studio 2017 [Preview version 15.3](https://www.visualstudio.com/vs/preview/) (Previous versions of Visual Studio 2017 doesn’t support .NET Core 2.0 Preview 1)
3. Create a SQL Server database.
4. Create Products table:

CREATE TABLE [dbo].[Products](

[Name] [varchar](200) NOT NULL,

[Quantity] [int] NOT NULL,

CONSTRAINT [PK\_Products] PRIMARY KEY CLUSTERED

(

[Name] ASC

)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

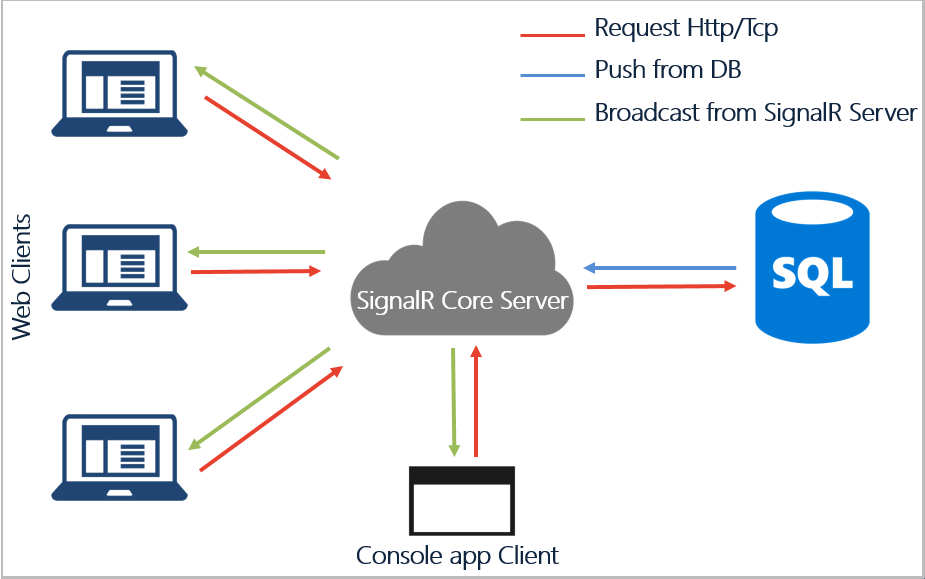
) ON [PRIMARY]

GO

**Instructions**

1. Clone [this](https://github.com/vany0114/SignalR-Core-SqlTableDependency) repository.
2. Compile it.
3. In order to use the SQL Broker, you must be sure to enable Service Broker for the database. You can use the following command: ALTER DATABASE MyDatabase SET ENABLE\_BROKER
4. Execute the SignalRCore.Web project.
5. Execute the SignalRCore.CommandLine project. You can use dotnet run command.

**Explanation**

Fig1. - Demo

As you can see in the image above, there is a SignalR Core server that is subscribed to the database via SqlTableDependency. Also there is a console app client that is connected to the SignalR Core server through TCP protocol and the web clients are connected via HTTP protocol. The SignalR Core server performs the broadcast to all clients when any client perform a request or even when the database change.

**Understanding the Code**

First of all, in order to use SignalR Core we must reference the nuget package source for Asp.Net Core and Asp.Net Core Tools.

<?xml version="1.0" encoding="utf-8"?>

<configuration>

<packageSources>

<add key="AspNetCore" value="<https://dotnet.myget.org/F/aspnetcore-ci-dev/api/v3/index.json>" />

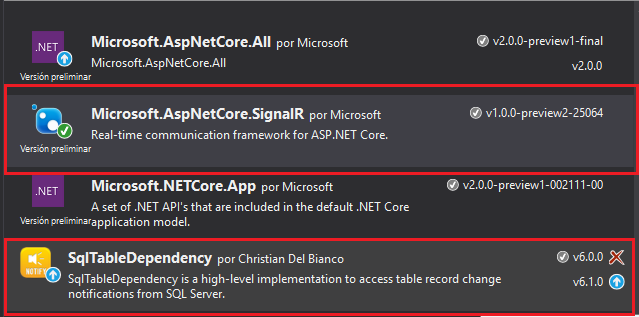
<add key="AspNetCoreTools" value="<https://dotnet.myget.org/F/aspnetcore-tools/api/v3/index.json>" />

<add key="NuGet" value="<https://api.nuget.org/v3/index.json>" />

</packageSources>

</configuration>

Now we can reference the SignalR Core nuget package. Besides We need to reference the SqlTableDependency nuget package that we gonna need later.

Fig2. - Nuget Packages

**Server side:**

Once configured the nuget packages we can start to use SignalR Core, the first thing is create the Hub.

public class Inventory : Hub

{

private readonly IInventoryRepository \_repository;

public Inventory(IInventoryRepository repository)

{

\_repository = repository;

}

public Task RegisterProduct(string product, int quantity)

{

\_repository.RegisterProduct(product, quantity);

return Clients.All.InvokeAsync("UpdateCatalog", \_repository.Products);

}

public async Task SellProduct(string product, int quantity)

{

await \_repository.SellProduct(product, quantity);

await Clients.All.InvokeAsync("UpdateCatalog", \_repository.Products);

}

}

There you go, we got a Hub, naked eye is the same Hub like an old SignalR version, but there are a couple of significant differences, the first one is that SignalR Core doesn’t use anymore Dynamic types to invoke the client methods, instead uses a method called ***InvokeAsync***, that receives the name of the client method and the parameters. The other difference is the dependency injection, even thought is not a Hub improvement itself, but it’s a great improvement of SignalR Core and Asp.Net Core in general, because in Asp.Net SignalR is necessary to do a work around in order to inject something to Hub, because SignalR application does not directly create hubs; SignalR creates them for you. By default, SignalR expects a hub class to have a parameterless constructor. So with Asp.net SignalR we must to modify the IoC container to solve this problem, luckily now is simpler.

Now, we gonna explain the repositories. I implemented two repositories, one in memory and another one with Entity Framework in order to get the products from SQL database. The first one is because I wanted to try the SignalR Core features faster, I was really look forward.

* **In memory Repository**: (nothing fancy as you can see, except for some cool feature of C# 7.0 if you can realize)

public class InMemoryInventoryRepository : IInventoryRepository

{

private readonly ConcurrentDictionary<string, int> \_products =

new ConcurrentDictionary<string, int>(new List<KeyValuePair<string, int>>

{

new KeyValuePair<string, int>("Desk", 3),

new KeyValuePair<string, int>("Tablet", 3),

new KeyValuePair<string, int>("Kindle", 3),

new KeyValuePair<string, int>("MS Surface", 1),

new KeyValuePair<string, int>("ESP Guitar", 2)

});

public IEnumerable<Product> Products => GetProducts();

public Task RegisterProduct(string product, int quantity)

{

if (\_products.ContainsKey(product))

\_products[product] = \_products[product] + quantity;

else

\_products.TryAdd(product, quantity);

return Task.CompletedTask;

}

public Task SellProduct(string product, int quantity)

{

\_products.TryGetValue(product, out int oldQuantity);

if (oldQuantity >= quantity)

\_products[product] = oldQuantity - quantity;

return Task.FromResult(oldQuantity >= quantity);

}

private IEnumerable<Product> GetProducts()

{

return \_products.Select(x => new Product

{

Name = x.Key,

Quantity = x.Value

});

}

}

* **Database repository**: there is one important thing in this repository, look out how I inject the data context. It is because the Entity Framework context is not thread safe and in concurrence scenarios the context has a lot of issues. So using a delegate, the context is instantiated and disposed inside the class it is injected in and on every needs because Entity Framework context life cycles should be as short as possible. This is a tip what a learned when I was studying about CQRS and Event Sourcing in that great [Microsoft project.](https://github.com/MicrosoftArchive/cqrs-journey) Later I’ll show you where and how the data context’s dependency injections is configured.

public class DatabaseRepository : IInventoryRepository

{

private Func<InventoryContext> \_contextFactory;

public IEnumerable<Product> Products => GetProducts();

public DatabaseRepository(Func<InventoryContext> context)

{

\_contextFactory = context;

}

public Task RegisterProduct(string product, int quantity)

{

using (var context = \_contextFactory.Invoke())

{

if (context.Products.Any(x => x.Name == product))

{

var currentProduct = context.Products.FirstOrDefault(x => x.Name == product);

currentProduct.Quantity += quantity;

context.Update(currentProduct);

}

else

{

context.Add(new Product { Name = product, Quantity = quantity });

}

context.SaveChanges();

}

return Task.FromResult(true);

}

public Task SellProduct(string product, int quantity)

{

using (var context = \_contextFactory.Invoke())

{

var currentProduct = context.Products.FirstOrDefault(x => x.Name == product);

if (currentProduct.Quantity >= quantity)

{

currentProduct.Quantity -= quantity;

context.Update(currentProduct);

}

context.SaveChanges();

}

return Task.FromResult(true);

}

private IEnumerable<Product> GetProducts()

{

using (var context =\_contextFactory.Invoke())

{

return context.Products.ToList();

}

}

}

Now we gonna talk about how SqlTableDependency works. I created a class called ***InventoryDatabaseSubscription*** that implements an interface called ***IDatabaseSubscription*** in order to wrap the complexity about the subscriptions to database.

public class InventoryDatabaseSubscription : IDatabaseSubscription

{

private bool disposedValue = false;

private readonly IInventoryRepository \_repository;

private readonly IHubContext<Inventory> \_hubContext;

private SqlTableDependency<Product> \_tableDependency;

public InventoryDatabaseSubscription(IInventoryRepository repository, IHubContext<Inventory> hubContext)

{

\_repository = repository;

\_hubContext = hubContext;

}

public void Configure(string connectionString)

{

\_tableDependency = new SqlTableDependency<Product>(connectionString, null, null, null, null, DmlTriggerType.Delete);

\_tableDependency.OnChanged += Changed;

\_tableDependency.OnError += TableDependency\_OnError;

\_tableDependency.Start();

Console.WriteLine("Waiting for receiving notifications...");

}

private void TableDependency\_OnError(object sender, ErrorEventArgs e)

{

Console.WriteLine($"SqlTableDependency error: {e.Error.Message}");

}

private void Changed(object sender, RecordChangedEventArgs<Product> e)

{

if (e.ChangeType != ChangeType.None)

{

// TODO: manage the changed entity

var changedEntity = e.Entity;

\_hubContext.Clients.All.InvokeAsync("UpdateCatalog", \_repository.Products);

}

}

#region IDisposable

~InventoryDatabaseSubscription()

{

Dispose(false);

}

protected virtual void Dispose(bool disposing)

{

if (!disposedValue)

{

if (disposing)

{

\_tableDependency.Stop();

}

disposedValue = true;

}

}

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

#endregion

}

The class receives the repository and the Inventory hub context, also implements the ***Configure*** method, that basically configure the subscription with the database based on the connection string that it receives like parameter.

As you can see I subscribe to *Product* table using the Generic feature of SqlTableDependency passing the entity *Product* (by the way, it uses data annotations). There is an important thing as well, notice that the subscription only listens the delete operation on the table, because I’m passing the last parameter like this: ***DmlTriggerType.Delete***

Besides I specify a delegate to handle any change what I subscribed when database is changed. Here I perform the broadcast to all clients to notify the change through hub context. As you can see is pretty easy to use SqlTableDependency!

Now is time to take a look the configuration of ***Startup.css*** file, dependency injection and so on.

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

services.AddSignalR();

services.AddEndPoint<MessagesEndPoint>();

// dependency injection

services.AddDbContextFactory<InventoryContext>(Configuration.GetConnectionString("DefaultConnection"));

services.AddScoped<IInventoryRepository, DatabaseRepository>();

services.AddSingleton<InventoryDatabaseSubscription, InventoryDatabaseSubscription>();

services.AddScoped<IHubContext<Inventory>, HubContext<Inventory>>();

//services.AddSingleton<IInventoryRepository, InMemoryInventoryRepository>();

}

In this method we add SignalR request handler to the Asp.Net Core’ pipeline and we configure the dependency injection as well. Here we have some considerations about the data context and SqlTableDependency injection. I’ve created an extension called ***AddDbContextFactory*** in order to inject the data context as I explain earlier.

public static void AddDbContextFactory<DataContext>(this IServiceCollection services, string connectionString)

where DataContext : DbContext

{

services.AddScoped<Func<DataContext>>((ctx) =>

{

var options = new DbContextOptionsBuilder<DataContext>()

.UseSqlServer(connectionString)

.Options;

return () => (DataContext)Activator.CreateInstance(typeof(DataContext), options);

});

}

Notice that I return a delegate that returns a sentence that create an instance of *DataContext* but don’t return the instance itself. Besides notices that the injection is per request as long as it uses *AddScoped* method.

Now, about the ***InventoryDatabaseSubscription*** notice it’s injected as a singleton, because the subscription to database must performs once in order to avoid kill our database. In order to complete the configuration about the subscription to our database I’ve create another extension called ***UseSqlTableDependency*** that basically call the *Configure* method on *InventoryDatabaseSubscription* implementation. I just get the instance from Asp.Net Core service locator and then calls the method.

public static void UseSqlTableDependency<T>(this IApplicationBuilder services, string connectionString)

where T : IDatabaseSubscription

{

var serviceProvider = services.ApplicationServices;

var subscription = serviceProvider.GetService<T>();

subscription.Configure(connectionString);

}

Finally to finish the configuration we need to configure the endpoint to the SignalR Hub. In this case the endpoint is ***/inventory*** that’s mapping with *Inventory* Hub (notice the last line use the extension explained before)

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

else

{

app.UseExceptionHandler("/Home/Error");

}

app.UseStaticFiles();

app.UseSignalR(routes =>

{

routes.MapHub<Inventory>("/inventory");

});

app.UseSockets(routes =>

{

routes.MapEndpoint<MessagesEndPoint>("/message");

});

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

app.UseSqlTableDependency<InventoryDatabaseSubscription>(Configuration.GetConnectionString("DefaultConnection"));

}

**Client side:**

Now we gonna talk about the clients, we start with web client. In order to connect with SignalR Core Server easily, we gonna use the SignalR Core javascript client that provides SignalR Core. We only need to specify the endpoint and the formats that we want to handle.

let connection = new signalR.HubConnection(`http://${document.location.host}/inventory`, 'formatType=json&format=text');

let startConnection = () => {

connection.start()

.then(e => {

$("#connetion-status").text("Connection opened");

$("#connetion-status").css("color", "green");

})

.catch(err => console.log(err));

};

startConnection();

To receive notifications from server I have the method called *UpdateCatalog* that refresh the products.

connection.on('UpdateCatalog', products => {

$('#products-table').DataTable().fnClearTable();

$('#products-table').DataTable().fnAddData(products);

refreshProductList(products);

});

And to invoke a server method from the client, we gonna use the ***invoke*** method that’s provided for the API.

$("#btn-sell").on('click', (e) => {

let product = $("#product").val();

let quantity = parseInt($("#quantity").val());

connection.invoke('SellProduct', product, quantity)

.catch(err => console.log(err));

});

Lastly we have a console application client that also receives notifications from server and invoke to server as well. This client is located on *SignalRCore.CommandLine* project and it maintain a connection with the server via ***HubConnection*** class. This class is very “similar” to the javascript API, talking about the use, at least. It has a method called ***On*** to receive notifications and a method called ***Invoke*** to invoke a server method.

public static async Task<int> ExecuteAsync()

{

var baseUrl = "<http://localhost:4235/inventory>";

var loggerFactory = new LoggerFactory();

Console.WriteLine("Connecting to {0}", baseUrl);

var connection = new HubConnection(new Uri(baseUrl), loggerFactory);

try

{

await connection.StartAsync();

Console.WriteLine("Connected to {0}", baseUrl);

var cts = new CancellationTokenSource();

Console.CancelKeyPress += (sender, a) =>

{

a.Cancel = true;

Console.WriteLine("Stopping loops...");

cts.Cancel();

};

// Set up handler

connection.On("UpdateCatalog", new[] { typeof(IEnumerable<dynamic>) }, a =>

{

var products = a[0] as List<dynamic>;

foreach (var item in products)

{

Console.WriteLine($"{item.name}: {item.quantity}");

}

});

while (!cts.Token.IsCancellationRequested)

{

var product = await Task.Run(() => ReadProduct(), cts.Token);

var quanity = await Task.Run(() => ReadQuantity(), cts.Token);

if (product == null)

{

break;

}

await connection.Invoke("RegisterProduct", cts.Token, product, quanity);

}

}

catch (AggregateException aex) when (aex.InnerExceptions.All(e => e is OperationCanceledException))

{

}

catch (OperationCanceledException)

{

}

finally

{

await connection.DisposeAsync();

}

return 0;

}

So that’s all about SignalR Core and SqlTableDependency, I hope will be useful for you all and that you keep motivated with .Net Core and Asp.Net Core. As a little gift you can take a look to ***MessagesEndPoint*** class, that’s an example about a pure socket implementation with SignalR Core. The web client is ***sockets.html***.

Download the code from my GitHub repository: <https://github.com/vany0114/SignalR-Core-SqlTableDependency>