# Broadcast real-time notifications on database record change with SignalR, Knockout JS and SqlTableDependency

SignalR is a .NET library for that enables real-time communication from server to clients. It does this via web-sockets, which is typically available in modern browsers, but also degrades gradually to other methods that are supported by the older browsers. With SignalR your application can push content to connected clients instantly, rather than having the server wait for a client requesting new data.

But what about push content from SQL Server to server application?

## Scenario

Let's assume we are developing a Web application used to book flight tickets. Most probably the tickets availability is stored in a database used by different booking terminals. Those terminals have no idea about availability and the only way for you to be aware of any change, is to continuously poll the database for changes and refresh the display. This approach (often referred as pull approach) has some disadvantages, such as a constant performance hit on the server, even when there is no change in data.

## SignalR to instant notification from server to clients

For such applications, we need the server to take the initiative and be capable of sending information to the client when a relevant event occurs, instead of waiting for the client to request it.

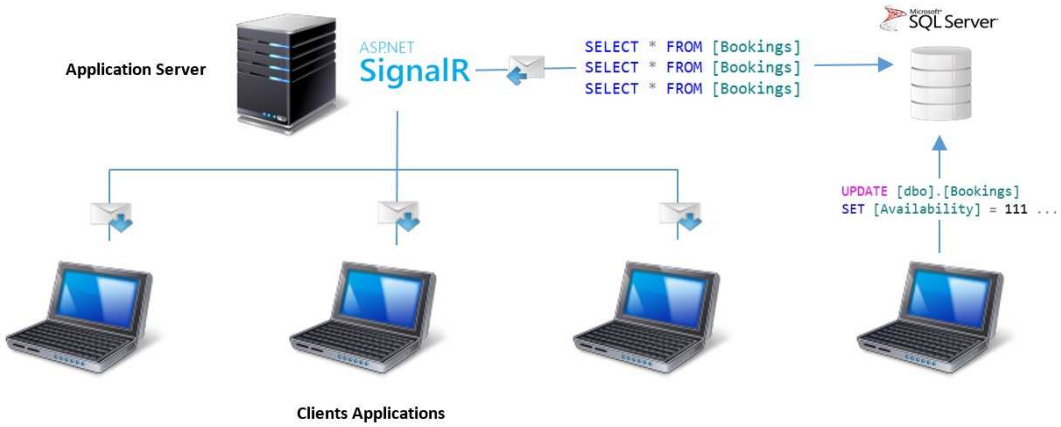



Fig.1: Notification from Server to client, where server application needs to polling database to keep.

SignalR includes an "out-of-the-box" set of transports and it determines which one it should use based on certain factors, such as the availability of the technology at both ends. SignalR will always try to use the most efficient transport and will keep falling back until selecting the best one that is compatible with the context. This decision is made automatically during an initial stage in the communication between the client and the server.

But what about database? How keep our server code aware of database changes, possibly preventing query it every tot seconds?

Starting from .NET Framework 2.0, Microsoft supply SqlDependency: this class receive notifications from SQL server every time the result set returned from the registered query is modified. However this

notification doesn't tell you which record has changed once a notification is delivered, it is necessary to fetch again the data executing a new database query.

Wouldn't it be better if this database notification returned us updated, inserted or deleted record fields, avoiding us to execute a new select?

## SqlTableDependency to instant notification from database to server

SqlTableDependency is an open source component able to create a series of database objects used to receive notifications on table record change. When any insert/update/delete operation is detected on a table, a change notifications containing record status is sent to SqlTableDependency. This avoid our server code to execute a further select to update application's data.

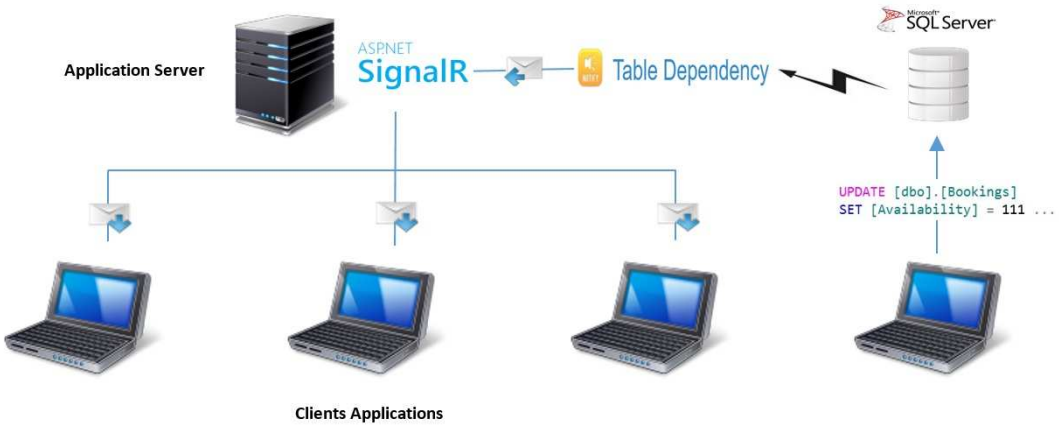



Fig.2: Notification from Database to Server.

SqlTableDependency class represents a notification dependency between an application and a SQL Server table. When you use SqlTableDependency to get notifications, this component provides the low-level implementation creating an infrastructure composed of table Trigger, Contracts, Messages, Queue, Service Broker and a clean-up stored procedure. In this way SqlTableDependency class provides access to notifications without knowing anything about the underlying Service Broker infrastructure. Once a record change happens, this infrastructure notify SqlTableDependency that in turn raise a .NET event to subscribers providing the record values.

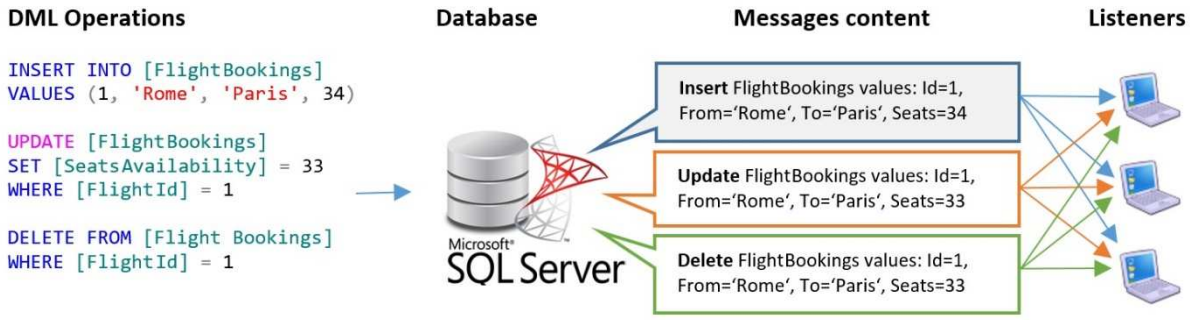
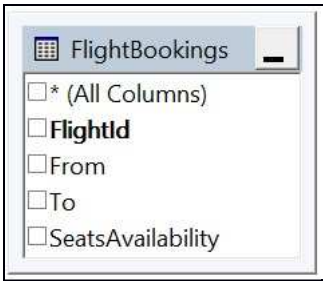



Fig.3: Database notification to SqlTableDependency.

The event raised implements the generic <T> pattern, this means a subscriber receive a C# object defined based on our needs. Let's do a very simple example. Supposing a database table as shown, we define a model that maps interested table columns:




```
public class SeatsAvailability
{
   public string From { get; set; }
   public string To { get; set; }
   public int Seats { get; set; }
}
```

After that we can create our dependency:

```
var conString =
   ConfigurationManager.ConnectionStrings["connectionString"].ConnectionString;

var mapper = new ModelToTableMapper<SeatsAvailability>();
mapper.AddMapping(c => c.Seats, "SeatsAvailability");

using (var tableDependency =
   new SqlTableDependency<SeatsAvailability>(conString, "FlightBookings"))
{
   tableDependency.OnChanged += TableDependency_Changed;
   tableDependency.Start();

   Console.WriteLine(@"Waiting for receiving notifications...");
   Console.WriteLine(@"Press a key to stop");
   Console.ReadKey();

   tableDependency.Stop();
}
```

And finally we subscribe to change event:

```
private static void TableDependency_Changed(
   object sender,
   RecordChangedEventArgs<SeatsAvailability> e)
{
   var changedEntity = e.Entity;

   Console.WriteLine(@"DML operation: " + e.ChangeType);
   Console.WriteLine(@"From: " + changedEntity.From);
   Console.WriteLine(@"To: " + changedEntity.To);
   Console.WriteLine(@"Seats free: " + changedEntity.Seats);
}
```

For every inserted, deleted or update record, our code will receive a typed event within interested record information. To achieve this, SqlTableDependency create the following database objects:

- Trigger: used to catch record change on the monitored table and prepare a message to put on the queue.
- Conversation timer: When the time-out expires, Service Broker puts a message of type http://schemas.microsoft.com/SQL/ServiceBroker/DialogTimer on the queue.
- Stored procedure: used to clean up all SqlTableDependency objects.
- Queue: used to store records change messages.
- Service broker's objects: used to populate queue.

Those database objects are dropped when SqlTableDependency is disposed. For more info to https://tabledependency.codeplex.com.

## Put all together with a practical example using Knockout JS

Now it's time to put in practice. We are going to create a web application simulating a Flight Booking system.

### Initial settings

Created an ASP.NET MVC Web application and install the following packages:

```
PM> Install-Package Microsoft.AspNet.SignalR
PM> Install-Package Knockoutjs
PM> Install-Package Knockout.Mapping
PM> Install-Package SqlTableDependency
```

Then, we customize the layout view, in order to include the needed library:

```
<!DOCTYPE html>
<html>
<head>
    <title>SignalR, Knockout JS and SqlTableDependencly</title>
</head>
<body>
    <div class="container" style="margin-top: 20px">
        @RenderBody()
    </div>

    <script src="~/Scripts/jquery-1.10.2.js"></script>
    <script src="~/Scripts/jquery.signalR-2.2.0.js"></script>
    <script src="~/signalr/hubs"></script>
    <script src="~/Scripts/knockout-3.4.0.js"></script>
    <script src="~/Scripts/knockout.mapping-latest.js"></script>

    @RenderSection("scripts", required: false)
</body>
</html>
```

Also, we have to initialize SignalR, creating the following class:

```
using Microsoft.Owin;
using Owin;

[assembly: OwinStartup(typeof(FlightBooking.Startup))]

namespace FlightBooking
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            app.MapSignalR();
        }
    }
}
```

For more info about SignalR and how to configure it, please refer to http://www.asp.net/signalr.

### Server side code pushing seats availability changes to clients

Not we are going to create the server functionalities. To recap our target, what we want to achieve is a notification's chain that, due to any change executed on a database table, forward this information

to all connected clients. So, assuming an update DML operation on a monitored table is executed, we want to broadcast messages from database up to clients:


Fig.4: Notification chain.

We start defining our Hub class. We are going to use this class to retrieve the first set of flight seats availability and after to push seats availability change. This class establish a communication channel between the server and clients:

```
[HubName("flightBookingTicker")]
public class FlightBookingHub : Hub
{
   private readonly FlightBookingService _flightBookingService;

   public FlightBookingHub() : this(FlightBookingService.Instance) { }

   public FlightBookingHub(FlightBookingService flightBookingHub)
   {
      _flightBookingService = flightBookingHub;
   }

   public FlightsAvailability GetAll()
   {
      return _flightBookingService.GetAll();
   }
}
```

Then, we need to constitute the second channel between database and web application, in order to communicate back to client any change about reservation state. For this we are going to use SqlTableDependency:

```
public class FlightBookingService : IDisposable
{
  // singleton instance
   private readonly static Lazy<FlightBookingService> _instance =
      new Lazy<FlightBookingService>(() =>
         new FlightBookingService(
           GlobalHost.ConnectionManager.GetHubContext<FlightBookingHub>().Clients));

   private SqlTableDependency<FlightAvailability> SqlTableDependency { get; }
   private IHubConnectionContext<dynamic> Clients { get; }

   private static connectionString =
      ConfigurationManager.ConnectionStrings["connectionString"].ConnectionString;

   private FlightBookingService(IHubConnectionContext<dynamic> clients)
   {
      this.Clients = clients;

      // because our C# model has a property not matching database table name,
      // an explicit mapping is required just for this property
```

```
        var mapper = new ModelToTableMapper<FlightAvailability>();
        mapper.AddMapping(x => x.Availability, "SeatsAvailability");

        // because our C# model name differs from table name we have to
        // specify database table name
        this.SqlTableDependency = new SqlTableDependency<FlightAvailability>(
           connectionString,
           "FlightBookings",
           mapper);

        this.SqlTableDependency.OnChanged += this.TableDependency_OnChanged;
        this.SqlTableDependency.Start();
    }

    public static FlightBookingService Instance => _instance.Value;

    public FlightsAvailability GetAll()
    {
        var flightsAvailability = new List<FlightAvailability>();

        using (var sqlConnection = new SqlConnection(connectionString))
        {
           sqlConnection.Open();
           using (var sqlCommand = sqlConnection.CreateCommand())
           {
              sqlCommand.CommandText = "SELECT * FROM [FlightBookings]";

              using (var sqlDataReader = sqlCommand.ExecuteReader())
              {
                 while (sqlDataReader.Read())
                 {
                    var flightId = sqlDataReader.GetInt32(0);
                    var from = sqlDataReader.GetString(1);
                    var to = sqlDataReader.GetString(2);
                    var seats = sqlDataReader.GetInt32(2);

                    flightsAvailability.Add(new FlightAvailability {
                       FlightId = flightId,
                       From = from,
                       To = to,
                       Availability = seats
                    });
                 }
              }
           }
        }

        return new FlightsAvailability() {
           FlightCompanyId = "not used",
           FlightAvailability = flightsAvailability
        };
    }

    private void TableDependency_OnChanged(
        object sender,
        RecordChangedEventArgs<FlightAvailability> e)
    {
        switch (e.ChangeType)
        {
           case ChangeType.Delete:
              this.Clients.All.removeFlightAvailability(e.Entity);
              break;

           case ChangeType.Insert:
```

```
            this.Clients.All.addFlightAvailability(e.Entity);
            break;

         case ChangeType.Update:
            this.Clients.All.updateFlightAvailability(e.Entity);
            break;
      }
   }

   public void Dispose()
   {
      // invoke Stop() in order to remove all DB objects generated from SqlTableDependency
      this.SqlTableDependency.Stop();
   }
}
```

As you can see, SqlTableDependency trigger an event on any data table change. Within this event, we have our *FlightAvailability* model populated with *current* table values. In case of update operation, we receive the latest value. One we have that value, using our *FlightBookingHub* hub instance, we notify the javascript proxy calling the respective method that, through knockout view model, update the UI.

## Seats availability view

Create a simple controller just to render the view. This view take advantages of knockout to display free seats and it is updated every time the availability change:

```
<table class="table table-striped">
    <thead style="background-color: silver">
        <tr>
            <th>Flight Id</th>
            <th>From</th>
            <th>To</th>
            <th>Seats Availability</th>
        </tr>
    </thead>
    <tbody data-bind="foreach: flights">
        <tr>
            <td><span data-bind="text: $data.flightId"></span></td>
            <td><span data-bind="text: $data.from"></span></td>
            <td><span data-bind="text: $data.to"></span></td>
            <td><span data-bind="text: $data.freeSeats"></span></td>
        </tr>
    </tbody>
</table>

@section Scripts {
    <script src="~/Scripts/flightBookingViewModels.js"></script>
    <script src="~/Scripts/flightBookingTicker.js"></script>
}
```

## Knockout view model and SignalR client code

What remains to do is to define the knockout view model, used to bind HTML elements with our data. This means that every time we change the number of free seats into the view model, knockout will update our UI automatically.

```
// flight view model definition
function FlightBookingViewModel(flight) {
    var self = this;

    var mappingOptions = {
```

```
        key: function (data) {
            return ko.utils.unwrapObservable(data.flightId);
        }
    };

    ko.mapping.fromJS(flight, mappingOptions, self);
};


// flights view model definition
function FlightsBookingViewModel(flights) {
    var self = this;

    var flightsBookingMappingOptions = {
        flights: {
            create: function (options) {
                return new FlightBookingViewModel(options.data);
            }
        }
    };

    self.addFlightAvailability = function (flight) {
        self.flights.push(new FlightBookingViewModel(flight));
    };

    self.updateFlightAvailability = function (flight) {
        var flightMappingOptions = {
            update: function (options) {
                ko.utils.arrayForEach(options.target, function (item) {
                    if (item.flightId() === options.data.flightId) {
                        item.freeSeats(options.data.freeSeats);
                    }
                });
            }
        };

        ko.mapping.fromJS(flight, flightMappingOptions, self.flights);
    };

    self.removeFlightAvailability = function (flight) {
        self.flights.remove(function(item) {
             return item.flightId() === flight.flightId;
        });
    };

    ko.mapping.fromJS(flights, flightsBookingMappingOptions, self);
};
```

The main view model - *FlightsBookingViewModel* - has a constructor parameter used to initialize its self with all seats availability. Moreover, it expose three methods to take care of update its self:

- removeFlightAvailability
- updateFlightAvailability
- addFlightAvailability

Those methods will be called from our server side application, using the client-side hub proxy:

```
$(function () {
    var viewModel = null;

    // generate client-side hub proxy and then
    // add client-side hub methods that the server will call
```

```
    var ticker = $.connection.flightBookingTicker;

    // Add a client-side hub method that the server will call
    ticker.client.updateFlightAvailability = function (flight) {
        viewModel.updateFlightAvailability(flight);
    };

    ticker.client.addFlightAvailability = function (flight) {
        viewModel.addFlightAvailability(flight);
    };

    ticker.client.removeFlightAvailability = function (flight) {
        viewModel.removeFlightAvailability(flight);
    };

    // start the connection, load seats availability and set the KO view model
    $.connection.hub.start().done(function() {
        ticker.server.getAll().done(function (flightsBooking) {
            viewModel = new FlightsBookingViewModel(flightsBooking);
            ko.applyBindings(viewModel);
        });
    });
});
```

We can easily test this running the application and connecting it using two or more browser. Once the flight availabilities are displayed, changing any data in database table will result in an immediate UI update.

## Test it

Source code: https://realtimenotifications.codeplex.com/. Then follow these steps to run it:

- Create a table as:
```
CREATE TABLE [dbo].[FlightBookings](
       [FlightId] [int] PRIMARY KEY NOT NULL,
       [From] [nvarchar](50) NULL,
       [To] [nvarchar](50) NULL,
       [SeatsAvailability] [int] NULL)
```
- Populate the table with some data.
- Run the web application.
- Modify any data in the table to get an immediate notification on the HTML page.

## Step by step video

See implementation on youtube: https://youtu.be/FBkkdCuTO7g