LAMK Lahden ammattikorkeakoulu
Lahti University of Applied Sciences

Push Service with ASP.NET SignalR

Case ModulErp

LAHTI UNIVERSITY OF APPLIED SCIENCES
Faculty of Technology
Information Technology
Software Engineering
Bachelor’s Thesis
Autumn 2017
Maija Kekkonen

Lahti University of Applied Sciences
Degree Programme in Information Technology

KEKKONEN, MAIJA:

Push Service with ASP.NET SignalR
Case ModulErp

Bachelor's Thesis in Software Engineering, 51 pages

Autumn 2017

ABSTRACT

The goal of this thesis was to design and implement a prototype of a push service to the ModulERP enterprise resource planning solution. The push service would be used in the browser interface of the system. A push service enables real-time data transfer from a server to a client without the client having to specifically request the data.

The research problem was to implement the push service prototype by using the SignalR library. SignalR enables push services to be quickly implemented in ASP.NET environments. The Visual Studio development environment and the SQL Server Management Studio software were used in the implementation. The programming languages used were C#, Visual Basic and JavaScript.

The result was a prototype of a push service that could be used to transmit real-time data into the browser interface of the ModulERP system. Popups based on the Telerik user interface component library, a status bar, and a message drop-down menu were added to the browser interface. The prototype has been used in informing users about the zipping of possibly large amounts of files and in informing key users about large attachment files in the system.

Key words: ASP.NET, push service, SignalR, web development

Lahden ammattikorkeakoulu
Tietotekniikka

KEKKONEN, MAIJA:

Push Service with ASP.NET SignalR
Case Modulerp

Ohjelmistotekniikan opinnäytetyö, 51 sivua

Syksy 2017

TIIVISTELMÄ

Opinnäytetyön tavoitteena oli suunnitella ja toteuttaa push-palvelun prototyyppi JL-Soft Oy:n ModulERP-toiminnanohjausohjelmistoon sen selainkäyttöliittymässä käytettäväksi. Push-palvelu mahdollistaa reaaliaikaisen tiedonvälityksen palvelimelta asiakkaalle ilman, että asiakkaan tarvitsee erikseen pyytää palvelimelta dataa.

Tutkimusongelma oli push-palvelun prototyypin toteutus käyttäen avoimen lähdekoodin SignalR-kirjastoa, joka mahdollistaa push-palveluiden nopean toteutuksen ASP.NET-ympäristöihin. Toteutuksessa käytettiin Visual Studio –kehitysympäristöä ja SQL Server Management Studio –ohjelmaa. Ohjelmointikieliä olivat C#, Visual Basic ja JavaScript.

Tuloksena oli push-järjestelmän prototyyppi, jolla voitiin välittää reaaliaikaista dataa ModulERP-järjestelmän selainkäyttöliittymään. Push-palveluun liittyy selainkäyttöliittymään lisätty statuspalkki ja pudotusvalikko, jossa push-viestit näkyvät, sekä Telerik-käyttöliittymäkomponenttikirjastolla toteutetut ponnahdusviestit. Prototyyppiä on alustavasti hyödynnetty mahdollisesti suurten tiedostomäärien paketoinnissa ZIP-tiedostoihin sekä avainkäyttäjien informoinnissa järjestelmässä olevista suurista liitetiedostoista.

Asiasanat: ASP.NET, push-palvelu, SignalR, web-kehitys

TABLE OF CONTENTS

1 INTRODUCTION 1

2 THE OPERATING ENVIRONMENT 2

2.1 General Description 2

2.2 Customers 3

2.3 The Need for a Push Service 3

3 SIGNALR 4

3.1 SignalR Basics 4

3.1.1 Transports 4

3.2 Architecture of a System Using SignalR 5

3.2.1 Communication Models 6

3.3 The SignalR Hub Class 7

3.3.1 The OWIN Startup Class 9

3.3.2 Methods That Clients Can Call 12

3.3.3 Calling Client Methods From the Hub 15

3.3.4 Managing Group Membership 17

3.3.5 Connection Lifetime Events 18

3.3.6 The Context Property and the State Object 18

3.3.7 Error Handling 20

3.3.8 Customizing the Hubs Pipeline 22

3.4 The JavaScript Client 23

3.4.1 The Proxy Generated by SignalR 23

3.4.2 Establishing a connection 25

3.4.3 Connection Lifetime Events on the Client 26

3.4.4 Configuring the connection 27

3.4.5 Error Handling and Client-Side Logging 29

3.5 The .NET Client 31

3.5.1 Basics 31

3.5.2 Configuring the Connection 32

3.5.3 Methods That the Server Can Call 33

3.5.4 Invoking Methods on the Server 37

3.5.5 Error Handling and Client-Side Logging 38

4 IMPLEMENTATION: CASE MODULERP 40

4.1 The OWIN Startup Class 40

4.2 Settings 40
4.3 The Database 41
4.4 The Hub Class 42
4.4.1 Connection Lifetime Events 42
4.5 Broadcasting from the Web Service 43
4.6 SignalR on the Client 44
4.6.1 Notifications and Popups 44
4.6.2 Client-side methods callable by the server 46
4.7 Implementations 46
5 CONCLUSION 48
SOURCES 49
APPENDIXES 52

1 INTRODUCTION

The goal of this thesis was to design a push service for the ModulERP enterprise resource planning solution maintained by JL-Soft. The library used was ASP.NET SignalR. Reaching the goal involved two sub-goals. The first one was researching the topic in order to find out how to implement a push service with the library. The second one was the actual design and creation of a prototype of the push service. The results of the research are presented in chapter 2. The design and implementation are described in chapter 3. The subject of the thesis was narrowed down to the SignalR library and the implementation of the prototype with it.

A push service delivers messages to clients without specific requests from them (Rouse & Steele 2014). This can be contrasted with more traditional HTTP connections, where the client always initiates them (Fielding 1999). Push technology enables clients to be notified instantly as the data becomes available without them needing to, for example, periodically poll for the data. SignalR, the library on which this thesis focuses on, is designed to enable and simplify the creation of a push service in ASP.NET applications.

The ModulERP solution is actively developed and maintained by JL-Soft Oy. The company is based in Lahti and was founded in 1990. It employs a little over ten people – the headquarters and developers are in Finland, but there are other employees in other cities and countries. The company has customers around the world. (JL-Soft Oy 2017.)

2 THE OPERATING ENVIRONMENT

2.1 General Description

ERP solutions, in general, help organizations manage their business by enabling decisions to be data-driven. They often gather and present data about finance, human resource management, inventory management etc. (Rouse 2017.) JL-Soft's ModulERP solution is based on modules that focus on these and other facets of business. These modules are described next.

- The **sales** module includes features for managing sales orders, shipping, invoicing, products, and offers. The module can be integrated with web stores. Transport documents can be created with it, and the module can also generate reports and statistics.
- The **purchasing** module can generate automatic, timed material purchasing proposals. It also handles invoices and their verification, linking to orders etc. The module generates reports, statistics and forecasts.
- The **manufacturing** module includes features for production planning, work time-tracking, workgroups and work roster. The module can be used to support preventive machine maintenance. An electronic work list can be maintained with the module. Product structure and data can be imported from external product design software. The module generates reports, statistics, and production forecasts. Production can be managed and monitored through the system.
- The **warehouse** module can be used to manage inventory, reclamations, and picking and receiving goods. Inventory can be tracked by batches. This module, like the other modules, also generates reports and statistics.
- The **accounting** module can be used to maintain a sales and a purchase ledger. It can also be used for managing payments and a payroll.

- The **maintenance** module includes features for scheduled maintenance, maintenance contracts, inputting and managing work orders, managing spare parts etc. The module includes a web interface for employees and customers. (JL-Soft Oy 2017.)

2.2 Customers

JL-Soft Oy has customers in over 30 countries. The customers represent a range of fields, such as electronics, metal and plastics industry, and import, installation and maintenance. The solution is used in companies of different sizes. (JL-Soft Oy 2017.)

Each customer's ERP solution is separate from the others. The solution can be delivered either as a standalone Windows application, or as a web-based cloud service. Solutions for multiple customers can be located on the same server.

2.3 The Need for a Push Service

A push service was identified as a solution to the need for real-time information. This need was apparent especially in the browser-based client interface. The push messages would be sent from JL-Soft's web service, and received by individual users when using the browser-based client. Each message could be directed to specific users only. There were other requirements in addition to real-time data, such as the buffering of messages when the client application is not running, and the logging of whether the messages have been read or not.

3 SIGNALR

3.1 SignalR Basics

SignalR is a library that simplifies adding real-time functionality to ASP.NET applications. This means that the server pushes data to clients, instead of clients requesting it. SignalR manages connections automatically. The connection between the client and the server is persistent, in contrast to classic HTTP connections that are re-established every time. (Fletcher 2014a.)

3.1.1 Transports

SignalR is an abstraction over transports that do the actual work between the client and the server. There are four transport types:

- WebSocket: This is the only transport that establishes a true persistent, two-way connection between the client and the server.
- Server Sent Events/EventSource: This transport type receives server-sent events. It connects to a server by using HTTP and receives events in text/event-stream MIME format without closing the connection. (Mozilla Developer Network 2015a; Mozilla Developer Network 2015b.)
- Forever Frame: This transport type can be used only with Internet Explorer as the client. It creates a hidden Iframe which makes a request to the server that does not complete. The server then continually sends script to the client that is immediately executed. The server and client connections are separate.
- Ajax long polling: A persistent connection is not created. This transport type polls the server with a request that stays open until the server responds, after which the connection is reset. (Fletcher 2014a.)

3.2 Architecture of a System Using SignalR

When using SignalR, there are at least two applications – a server application and a client application. In the case of ModulERP, the .NET server application is a web service written in C# and Visual Basic. It contains a Hub class that handles connections and contains methods callable by a client, and another class that contains functionality for

- fetching message, user and client data from a database,
- creating new messages, and
- broadcasting the messages.

The parts of the client application relevant to SignalR are written in JavaScript. This code initiates a new connection with the server every time a page is loaded, and contains methods that are callable by the server on each client. The RPCs, the client-callable server methods and the server-callable client methods, are how most of the data is transferred with SignalR. This is illustrated by Figure 1, which shows how the SignalR components in each application interact with each other.

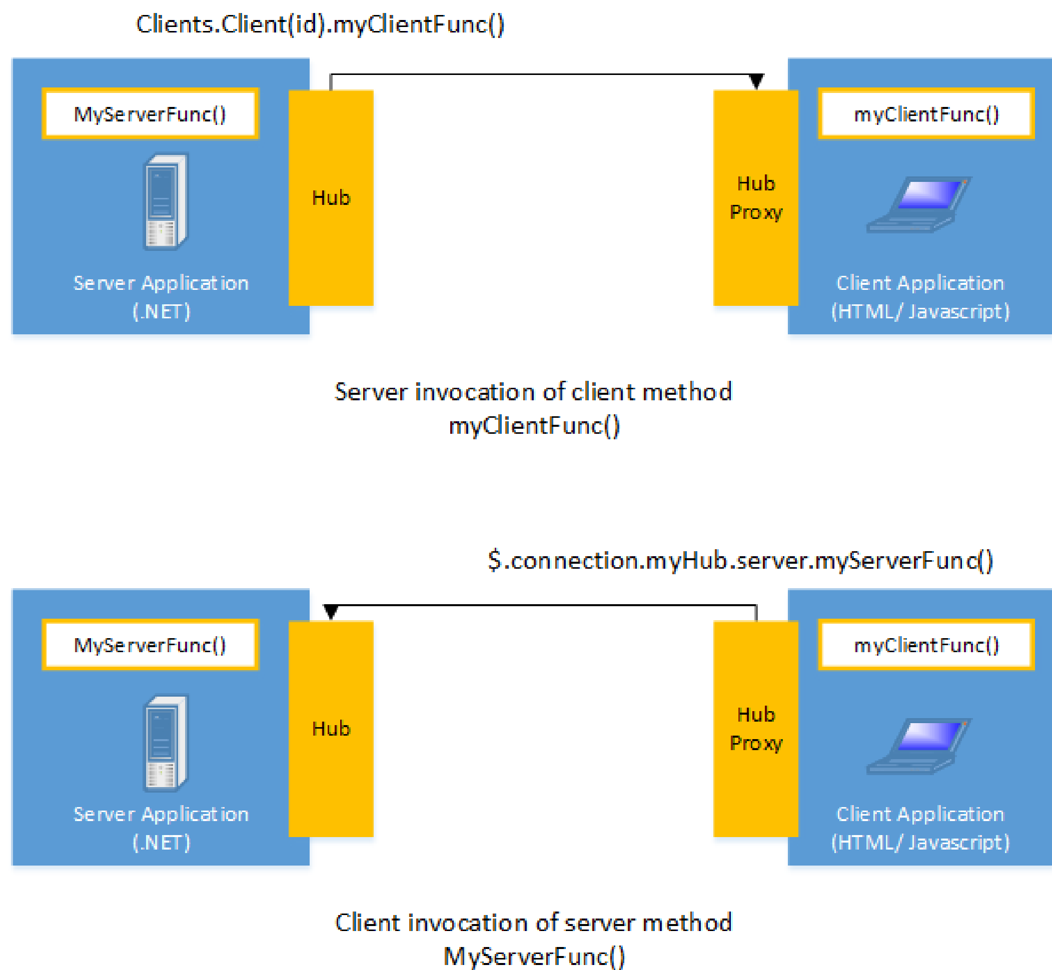

FIGURE 1. Remote procedure calls (Fletcher 2014a).

3.2.1 Communication Models

In SignalR there are two models available for communicating between clients and servers: Persistent Connections and Hubs. A Connection represents an endpoint for messages, be they single-recipient, grouped, or broadcasted. With the Connection API, the developer can directly access the low-level features of SignalR. The Hubs API is built upon the Connection API, and with the Hubs API clients can call server methods directly and vice versa. The Hubs API is used in this thesis. A diagram of the whole SignalR architecture can be seen in Figure 2. (Fletcher 2014a.)

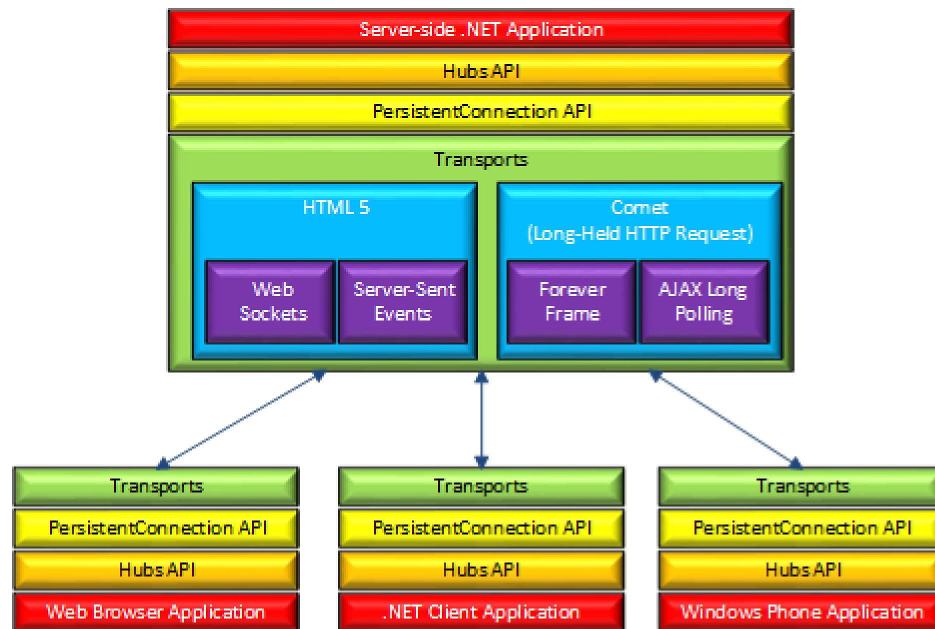

FIGURE 2. SignalR Architecture (Fletcher 2014a).

3.3 The SignalR Hub Class

The Hub class is in the center of push, or real-time, functionality. It allows remote procedure calls from client to server and vice versa. The "client-to-server plumbing" is hidden under the Hubs API. Multiple Hubs can be defined in an application. (Dykstra & Fletcher 2014b.)

Remote procedure calls are a Microsoft's technology for creating distributed client/server programs. They are intended to reduce developers' time and effort by providing a common interface between applications. They enable two-way communication between the client and the server. RPCs handle common tasks like security, synchronization and data flow. (Microsoft 2003.)

Hub classes are instantiated automatically by the SignalR Hubs pipeline. Thus, Hub classes are never instantiated from code on the server. Hub instances are transient – a new instance is created every time a client

connects, disconnects, or calls a method on the server. (Dykstra & Fletcher 2014b.)

When a client method is called from the server, a packet that contains the name and parameters of the method to be called is sent across the transport. A method call viewed from the monitoring tool Fiddler is shown in Figure 3. The method updateShape is called from the Hub MoveShapeHub. (Fletcher 2014a.)

```
10:38:03:1298 Session846.WebSocket'WebSocket #846' - Pushing 104 bytes from server WebSocket
81 66 7B 22 43 22 3A 22 42 2C 31 35 7C 43 2C 30     f{"C":"B,15|C,0
7C 44 2C 30 7C 45 2C 30 22 2C 22 4D 22 3A 5B 7B  |D,0|E,0","M":[{
22 48 22 3A 22 4D 6F 76 65 53 68 61 70 65 48 75  "H":"MoveShapeHu
62 22 2C 22 4D 22 3A 22 75 70 64 61 74 65 53 68  b","M":"updateSh
61 70 65 22 2C 22 41 22 3A 5B 7B 22 6C 65 66 74  ape","A":[{"left
22 3A 35 30 31 2E 30 2C 22 74 6F 70 22 3A 33 30  ":501.0,"top":30
32 2E 30 7D 5D 7D 5D 7D                          2.0}]}]}
```

FIGURE 3. A method call viewed from Fiddler (Fletcher 2014a).

Multiple Hubs can be defined in a single application. In such a case, the SignalR connection is shared but named groups of clients are separate. Using multiple Hubs does not cause a performance difference in the application. If a query string is specified to pass data from the client to the server, different query strings cannot be specified for different Hubs. All Hubs receive the same query string. This is because all Hubs get the same HTTP request information. Also, the generated JavaScript proxies file contains all the proxies for all Hubs in a single file.

By default, clients refer to Hubs by using the camel-cased name of the Hub. A different name can be specified by using the HubName attribute. An example of this is shown in Figure 4. There, the HubName attribute is enclosed in square brackets on line 2. On line 6 there is a demonstration of referring to the renamed Hub in client-side JavaScript. (Dykstra & Fletcher 2014b.)

```
1 // C# Code
2 [HubName("MyHubName")]
3 public class MyHub : Hub
4 
5 // JavaScript
6 var myHubProxy = $.connection.MyHubName;
```

FIGURE 4. Renaming a Hub (Dykstra & Fletcher 2014b).

3.3.1 The OWIN Startup Class

When using SignalR, an OWIN startup class is required. An OWIN startup class is a class where middleware components are specified to the OWIN pipeline. A simple example could be adding logging functionality. SignalR as a whole, and the route that clients use when connecting to the Hub, are configured in the startup class. The configuration is done in the Configuration method (line 9) by using the MapSignalR method as seen in Figure 5, line 12. The OwinStartup attribute on line 4 is used to define which class is used as the startup class. (Anderson & Thiagarajan 2013; Dykstra & Fletcher 2014b.)

```
1  using Microsoft.Owin;
2  using Owin;
3
4  [assembly: OwinStartup(typeof(MyApplication.Startup))]
5  namespace MyApplication
6  {
7      public class Startup
8      {
9          public void Configuration(IAppBuilder app)
10         {
11             // Any connection or hub wire up and configuration should go here
12             app.MapSignalR();
13         }
14     }
15 }
```

FIGURE 5. The OWIN startup class. (Dykstra & Fletcher 2014b.)

OWIN is an abbreviaton of “Open Web Interface for .NET”, and it defines an abstraction between .NET servers and web applications. OWIN decouples the server from web applications and thus makes creating middleware easier (Wasson 2013).

The configurations that can be altered using the OWIN startup class include

- A custom SignalR url (the default is “/signalr”)
- Enabling cross-domain calls with CORS or JSONP
- Enabling detailed error messages for troubleshooting
- Disabling automatically generated proxy files. (Dykstra & Fletcher 2014b.)

Establishing cross-domain calls must be enabled when SignalR lies on a different server than from where clients load pages. In this case, the clients make requests across domains. CORS and JSONP are used to enable cross-domain requests. (Dykstra & Fletcher 2015.)

CORS enables cross domain calls with JavaScript on all browsers. Normally the same-origin policy prevents JavaScript from making requests across different domains. Enabling CORS is shown in Figure 6. The class

named Startup is automatically used as the OWIN startup class (line 7) (Anderson & Thiagarajan 2013). The Map method branches the pipeline for all requests that start with “/signalr” (line 11) (Dykstra & Fletcher 2015). The UseCors method adds a CORS middleware to the application pipeline (line 13) (Microsoft 2017a). The “jQuery.support.cors” flag should not be set to true in JavaScript – SignalR handles the use of CORS (Dykstra & Fletcher 2015). (Hausenblas & Hossain 2015.)

JSONP, on the other hand, creates a <script> element that then “requests to a remote data service location”. JSONP uses the feature of <script> tags being able to request data in JSON format across domains. Enabling JSONP is shown in Figure 7. The class names Startup is again automatically used as the OWIN startup class (line 7). The pipeline is again branched for all requests starting with “/signalr” (line 11). JSONP is enabled in the HubConfiguration object (line 15), which is then passed as an argument to the RunSignalR method (line 17). (Getify Solutions & Simpson 2014.)

```
1  using Microsoft.AspNet.SignalR;
2  using Microsoft.Owin.Cors;
3  using Owin;
4
5  namespace MyWebApplication
6  {
7      public class Startup
8      {
9          public void Configuration(IAppBuilder app)
10         {
11             app.Map("/signalr", map =>
12             {
13                 map.UseCors(CorsOptions.AllowAll);
14             })
15         }
16     }
17 }
```

FIGURE 6. Enabling CORS (Dykstra & Fletcher 2015)

```
1  using Microsoft.AspNet.SignalR;
2  using Microsoft.Owin.Cors;
3  using Owin;
4
5  namespace MyWebApplication
6  {
7      public class Startup
8      {
9          public void Configuration(IAppBuilder app)
10         {
11             app.Map("/signalr", map =>
12             {
13                 var hubConfiguration = new HubConfiguration
14                 {
15                     EnableJSONP = true
16                 };
17                 map.RunSignalR(hubConfiguration);
18             })
19         }
20     }
21 }
```

FIGURE 7. Enabling JSONP (Dykstra & Fletcher 2015).

3.3.2 Methods That Clients Can Call

Methods that clients can call are declared public. A return type and parameters can be specified, just as in any C# method. An example is seen in Figure 8. The method is defined on line 3 – it's referenced by name client-side. On line 5 is an example of method functionality: a chat message with a user name and content is added to all clients' pages.

Methods can also be called asynchronously, of what there is an example in Figure 9. The method is declared asynchronous (line 1). An IEnumerable containing Stock objects is created from the data read from a HttpClient. This involves the usage of asynchronous methods (lines 7 and 8). The Task object returned contains the IEnumerable. Asynchronous

methods can be used when the method will be long-running or does work that will involve waiting. When using the WebSocket transport, asynchronous methods will not block the connection, in contrast to synchronous methods that will.

Progress reporting can be done with an IProgress<T> parameter as shown in Figure 10. A long-running task is simulated in a for loop (line 3) by waiting for 200 milliseconds in each loop cycle. Progress is reported by using the Report method of the IProgress parameter (line 6).

Overloads must be defined according to the number of parameters in a method. Complex types and arrays can be used as the return type – the data between the client and the server is in JSON format, and SignalR handles the binding of (arrays of) complex objects automatically. The methods can be renamed in a way resembling the renaming of a Hub – an example of this is shown in Figure 11. The new name is specified in the HubMethodName attribute (line 2). The method can then be referenced client-side using this name (line 6). (Dykstra & Fletcher 2014b.)

```
1 public class ContosoChatHub : Hub
2 {
3     public void NewContosoChatMessage(string name, string message)
4     {
5         Clients.All.addNewMessageToPage(name, message);
6     }
7 }
```

FIGURE 8. An example of a server method that a client can call (Dykstra & Fletcher 2014b).

```
public async Task<IEnumerable<Stock>> GetAllStocks()
{
    // Returns data from a web service.
    var uri = Util.getServiceUri("Stocks");
    using(HttpClient httpClient = new HttpClient())
    {
        var response = await httpClient.GetAsync(uri);
        return (await response.Content.ReadAsAsync<IEnumerable<Stock>>());
    }
}
```

FIGURE 9. An asynchronous server method that a client can call (Dykstra & Fletcher 2014b).

```
public async Task<string> DoLongRunningThing(IProgress<int> progress)
{
    for(int i = 0; i <= 100; i += 5)
    {
        await Task.Delay(200);
        progress.Report(i);
    }
    return "Job complete!";
}
```

FIGURE 10. Reporting method progress with the IProgress<int> parameter (Dykstra & Fletcher 2014b).

```
// C# code
[HubMethodName("MyMethodName")]
public void MyMethod(string p1, string p2)

// JavaScript
hubProxy.server.MyMethodName(p1, p2);
```

FIGURE 11. Specifying a custom name for a method callable in JavaScript (Dykstra & Fletcher 2014b).

3.3.3 Calling Client Methods From the Hub

Client methods can be called by using the Clients property of the Hub class. Ways of selecting receiving clients are described in Table 1. An example of calling a client method is shown in Figure 12. There, the server application adds a chat message to all clients (line 5). The corresponding JavaScript client method is shown in Figure 13. It is defined in the client property of the Hub proxy (line 1, see chapter 3.4). The example method contains simple jQuery code for adding the message to the page (lines 3-4).

Client methods execute asynchronously, and Figure 14 shows how they can be made to execute sequentially by using the “await” keyword (line 3). The keyword “async” is then needed in the method header (line 1). By using the “await” keyword, the calling client is notified *after* the method adding the message has been called (line 4). Using the keyword does not mean that the method actually has executed – it only means that SignalR has done everything necessary to attempt to execute the method. Client methods are also unable to return values. The specified method name is interpreted as a dynamic object, so there is no compile-time validation for it. If no matching method exists on the client, no error is raised. (Dykstra & Fletcher 2014b.)

```
1 public class ContosoChatHub : Hub
2 {
3     public void NewContosoChatMessage(string name, string message)
4     {
5         Clients.All.addNewMessageToPage(name, message);
6     }
7 }
```

FIGURE 12. Calling a client method from the server (Dykstra & Fletcher 2014b).

```
contosoChatHubProxy.client.addNewMessageToPage = function(name, message) {
    // Add the message to the page.
    $("discussion").append('<li><strong>' + htmlEncode(name) +
                           '<strong>:' + htmlEncode(message) + '<li>');
};
```

FIGURE 13. A JavaScript client method callable by server (Dykstra & Fletcher 2014b).

```
public async Task NewContosoChatMessage(string name, string message)
{
    await Clients.Others.addContosoChatMessageToPage(name, message);
    Clients.Caller.notifyMessageSent();
}
```

FIGURE 14. Sequential execution of client methods (Dykstra & Fletcher 2014b).

TABLE 1. Selecting clients that receive the Remote Procedure Call (Dykstra & Fletcher 2014b).

Syntax	Clients
Clients.All.clientMethod();	All clients
Clients.Caller.clientMethod();	The calling client
Clients.Others.clientMethod();	All except the calling client
Clients.Client(connectionId) .clientMethod();	A client specified by the connection ID
Clients.AllExcept(connId1, connId2).clientMethod();	All clients except the specified ones
Clients.Group(groupName) .clientMethod()	The clients in the specified group
Clients.User(userId).clientMethod();	A specific user
Clients.Clients(List<string> connIds).clientMethod();	All clients in the list
Clients.Groups(List<string> groupIds).clientMethod();	All groups in the list
Clients.Client(username) .clientMethod();	A user by name
Clients.Users(new string[] {”user1”, ”user2”}).clientMethod();	All users in the array (from SignalR version 2.1 on)

3.3.4 Managing Group Membership

Connections can be grouped in SignalR. Group membership can be managed from both the Hub on the server and from the client. Groups do not have to be explicitly created – a group is automatically created the first time it is referred to. There is no API to get a list of groups or a list of clients in groups. If a user must be in a group, all the user's connections have to be separately added to that group. The server side functionality of groups is shown in Figure 15. The current connection id is added into and removed from a group with a specific name in the respective methods (lines 5 and 10). The Add and Remove methods execute asynchronously. Client-side code is shown in Figure 16. The methods are called using the Hub proxy. (Dykstra & Fletcher 2014b.)

```
1  public class ContosoChatHub : Hub
2  {
3      public Task JoinGroup(string groupName)
4      {
5          return Groups.Add(Context.ConnectionId, groupName);
6      }
7
8      public Task LeaveGroup(string groupName)
9      {
10         return Groups.Remove(Context.ConnectionId, groupName);
11     }
12 }
```

FIGURE 15. Managing groups in the Hub (Dykstra & Fletcher 2014b).

```
1 contosoChatHubProxy.server.joinGroup(groupName);
2
3 contosoChatHubProxy.server.leaveGroup(groupName);
```

FIGURE 16. Managing groups on the JavaScript client (Dykstra & Fletcher 2014b).

3.3.5 Connection Lifetime Events

SignalR has three Hub methods for handling connection lifetime events: OnConnected, OnDisconnected and OnReconnected. The methods can be overridden in the Hub class. A new connection is established every time a browser navigates to a new page. This means that OnDisconnected is executed first, and OnConnected after that. In some cases, OnDisconnected does not get called at all – if a server goes down or if the App Domain gets recycled. (Dykstra & Fletcher 2014b.)

3.3.6 The Context Property and the State Object

Information about the client can be got from the Context property of the Hub. Types of this information are described in Table 2. The connection ID is a GUID. The same ID is used by all Hubs in the application. Query string data can be created in the JavaScript client.

The state object is provided by the client Hub proxy, and it can be used to pass data to the server with each method call. This is demonstrated in Figures 17, 18 and 19. The state data is created by the client. In the Hub, the state data is accessed from either the Clients.Caller (Figure 18) or the Clients.CallerState (Figure 19) property. The CallerState property is used when the Hub is strongly-typed or when the server code is written in Visual Basic. In Figures 18 and 19, a chat message with state data is added to all clients excluding the calling client. (Dykstra & Fletcher 2014b.)

TABLE 2. Information from the Context property (Dykstra & Fletcher 2014b).

Property	Type	Information
Context.ConnectionId	string	The connection ID of the calling client
Context.Request.Headers	NameValueCollection	HTTP header data
Context.Request .QueryString	NameValueCollection	Query string data
Context.Request.Cookies	IDictionary<string, Cookie>	Cookies
Context.User	IPrincipal	User information
Context.Request .GetHttpContext();	HttpContextBase	The HttpContext object of the request

```
1 contosoChatHubProxy.state.userName = "Fadi Fakhouri";
2 contosoChatHubProxy.state.computerName = "fadivm1";
```

FIGURE 17. Usage of the state object in the JavaScript client (Dykstra & Fletcher 2014b).

```
1 public void NewContosoChatMessage(string data)
2 {
3     string userName = Clients.Caller.userName;
4     string computerName = Clients.Caller.computerName;
5     Clients.Others.addContosoChatMessageToPage(message, userName, computerName);
6 }
```

FIGURE 18. Accessing the state object data in the Hub on the server (Dykstra & Fletcher 2014b).

```
1 public void NewContosoChatMessage(string data)
2 {
3     string userName = Clients.CallerState.userName;
4     string computerName = Clients.CallerState.computerName;
5     Clients.Others.addContosoChatMessageToPage(data, userName, computerName);
6 }
```

FIGURE 19. Using the CallerState property (Dykstra & Fletcher 2014b).

3.3.7 Error Handling

Instead of using try-catch blocks, errors can be handled by creating a Hubs pipeline module that handles the OnIncomingError method. The module must then be injected to the Hubs pipeline in the OWIN startup class. This way of handling errors is demonstrated in Figures 20 and 21. Figure 20 shows how the module can be created. The method OnIncomingError is overridden (line 3). Exception data is printed to debug output (lines 6 and 9). Last, the method of the parent class is called (line 11). Figure 21 demonstrates how the module is injected to the startup class (line 4).

Another way of handling errors is using the HubException class. Usage of HubException is demonstrated in Figures 22 and 23. Figure 22 shows how users in a chat application are prevented from sending <script> tags in their messages by throwing a HubException (line 7). Data for the exception can be created by passing an object as an argument to the HubException (line 8).

Figure 23 shows how the HubException can be handled on the client. First, the example “Send” function is called with a parameter that causes it to throw the HubException (line 1). The exception is handled in the “fail” function (line 2). In the fail callback, the type of the exception is checked to be HubException (line 3). Finally, message data is logged – user data is read from data passed to the exception (line 4). (Dykstra & Fletcher 2014b.)

```
public class ErrorHandlingPipelineModule : HubPipelineModule
{
    protected override void OnIncomingError(ExceptionContext exceptionContext,
                                            IHubIncomingInvokerContext invokerContext)
    {
        Debug.WriteLine("=> Exception " + exceptionContext.Error.Message);
        if(exceptionContext.Error.InnerException != null)
        {
            Debug.WriteLine("=> Inner Exception " + exceptionContext.Error.InnerException.Message);
        }
        base.OnIncomingError(exceptionContext, invokerContext);
    }
}
```

FIGURE 20. Creating an error handling module (Dykstra & Fletcher 2014b).

```
public void Configuration(IAppBuilder app)
{
    // Any connection or hub wire up and configuration should go here
    GlobalHost.HubPipeline.AddModule(new ErrorHandlingPipelineModule());
    app.MapSignalR();
}
```

FIGURE 21. Injecting the module into the Hubs pipeline (Dykstra & Fletcher 2014b).

```
public class MyHub : Hub
{
    public void Send(string message)
    {
        if(message.Contains("<script>"))
        {
            throw new HubException("This message will flow to the client",
                                   new { user = Context.User.Identity.Name, message = message });
        }
    }

    Clients.All.send(message);
}
```

FIGURE 22. Throwing a HubException in the Hub class (Dykstra & Fletcher 2014b).

```
1 myHub.server.send("<script>")
2     .fail(function(e) {
3         if(e.source === 'HubException') {
4             console.log(e.message + ' : ' + e.data.user);
5         }
6     });
```

FIGURE 23. Handling a HubException in the JavaScript client (Dykstra & Fletcher 2014b).

3.3.8 Customizing the Hubs Pipeline

Code can be injected into the Hubs pipeline. Figure 24 demonstrates a pipeline module that logs every method call both from the server and from the client. Incoming calls are logged in the overridden method "OnBeforeIncoming" (line 3), and outgoing calls in the method "OnBeforeOutgoing" (line 10). Figure 25 shows how the module is registered into the pipeline in the OWIN startup class (line 3). This is similar to the code shown in Figure 18. (Dykstra & Fletcher 2014b.)

```
1  public class LoggingPipelineModule : HubPipelineModule
2  {
3      protected override bool OnBeforeIncoming(IHubIncomingInvokerContext context)
4      {
5          Debug.WriteLine("=> Invoking " + context.MethodDescriptor.Name +
6                          " on hub " + context.MethodDescriptor.Hub.Name);
7          return base.OnBeforeIncoming(context);
8      }
9
10     protected override bool OnBeforeOutgoing(IHubOutgoingInvokerContext context)
11     {
12         Debug.WriteLine("<= Invoking " + context.Invocation.Method +
13                         " on client hub " + context.Invocation.Hub);
14         return base.OnBeforeOutgoing(context);
15     }
16 }
```

FIGURE 24. A pipeline module logging method calls (Dykstra & Fletcher 2014b).

```
1 public void Configuration(IAppBuilder app)
2 {
3     GlobalHost.HubPipeline.AddModule(new LoggingPipelineModule());
4     app.MapSignalR();
5 }
```

FIGURE 25. The logging module is registered in the startup class (Dykstra & Fletcher 2014b).

3.4 The JavaScript Client

3.4.1 The Proxy Generated by SignalR

SignalR generates a proxy that simplifies code, writes methods that the server calls, and calls methods on the server. The usage of the proxy, however, is optional. In Figures 26 and 27 it is demonstrated how the client and server (Hub) can communicate both with the proxy and without it.

Figure 26 shows how to use the generated hub proxy. The proxy is first accessed through the SignalR connection (line 1). A client method callable by the server-side Hub is defined on lines 2-4. In the example, it logs chat message data to the browser console. After the SignalR connection has been started (line 5), a Hub server method call is wired to an element on the page (lines 6-10). In the example method, a user name and a chat message are sent to the server. After the example method has been called, focus is shifted to the message input field.

Figure 27 presents client-side code without the generated proxy. The connection is accessed through the “$.hubConnection()” method (line 1). The proxy is created with a method in which the Hub name is specified (line 2). An event handler is added to provide a method the server can call

(lines 3-5). A server method is called on a click event by using the “invoke()” method (lines 8-9).

The proxy cannot be used, if the developer wants to register multiple event handlers for a single client method that the server calls. Otherwise, using the proxy can be chosen according to preference. The proxy file can be referenced through the “/signalr/hubs” URL. (Dykstra & Fletcher 2015.)

```
1  var contosoChatHubProxy = $.connection.contosoChatHub;
2  contosoChatHubProxy.client.addContosoChatMessageToPage = function(name, message) {
3      console.log(name + ' ' + message);
4  };
5  $.connection.hub.start().done(function() {
6      $('#newContosoChatMessage').click(function() {
7          contosoChatHubProxy.server
8              .newContosoChatMessage($('#displayname').val(), $('#message').val());
9          $('#message').val('').focus();
10     });
11 });
```

FIGURE 26. A client using the generated proxy (Dykstra & Fletcher 2015).

```
1  var connection = $.hubConnection();
2  var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
3  contosoChatHubProxy.on('addContosoChatMessageToPage', function(name, message) {
4      console.log(name + ' ' + message);
5  });
6  connection.start().done(function() {
7      $('#newContosoChatMessage').click(function() {
8          contosoChatHubProxy
9              .invoke('newContosoChatMessage', $('#displayname').val(), $('#message').val());
10         $('#message').val('').focus();
11     });
12 });
```

FIGURE 27. A client not using the generated proxy (Dykstra & Fletcher 2015).

3.4.2 Establishing a connection

Before establishing a connection, a connection object and a proxy must be created. Also, event handlers for methods to be called from the server must be registered. If the hub lies on a different server, its URL must be specified before starting the connection. When everything is ready, the start method can be called. The start method is asynchronous, and it returns a jQuery Deferred object. Figures 28 and 29 demonstrate this both with and without the proxy. The “$.connection.hub” object is the same as the one created by the “$.hubConnection()” method. (Dykstra & Fletcher 2015.)

In Figure 28, the generated proxy is accessed through the “connection” object (line 1). A method callable by the server is defined with the proxy (lines 2-4). The connection is started (line 5), and two callbacks are queued (lines 6-7). In the success callback, the connection id is printed to the browser console (line 6). In the failure callback, an error message is logged to the console (line 7).

In Figure 29, the connection is first accessed through the “$.hubConnection()” method (line 1). The Hub proxy is created by calling a method and specifying the Hub name (line 2). A method callable by the server is defined by creating an event handler (lines 3-5). The connection is started (line 6). Callbacks are queued as in the previous code example (lines 7-8).

A jQuery Deferred object is a chainable object. With it, callbacks can be registered into callback queues, invoke them, and relay the success or failure state of any function, whether synchronous or asynchronous. Some methods of the Deferred object are “state()”, “pipe()”, “fail()”, “done()”, etc. (The jQuery Foundation 2016.)

```
1 var contosoChatHubProxy = $.connection.contosoChatHub;
2 contosoChatHubProxy.client.addContosoChatMessageToPage = function(name, message) {
3     console.log(name + ' ' + message);
4 };
5 $.connection.hub.start()
6     .done(function() { console.log('Now connected, connection ID = ' + $.connection.hub.id); })
7     .fail(function() { console.log('Could not connect!'); });
```

FIGURE 28. Establishing a connection with the proxy (Dykstra & Fletcher 2015)

```
1 var connection = $.hubConnection();
2 var contosoChatHubProxy = connection.createHubProxy('contosoChatHub');
3 contosoChatHubProxy.on('addContosoChatMessageToPage', function(userName, message) {
4     console.log(userName + ' ' + message);
5 });
6 connection.start()
7     .done(function() { console.log('Now connected, connection ID = ' + connection.id); })
8     .fail(function() { console.log('Could not connect'); });
```

FIGURE 29. Establishing a connection without the proxy (Dykstra & Fletcher 2015).

3.4.3 Connection Lifetime Events on the Client

The client can handle several connection lifetime events. These events are described in Table 3. The events can be handled to display warning messages etc. (Dykstra & Fletcher 2015.)

TABLE 3. Connection lifetime events on the client (Dykstra & Fletcher 2015).

Event	When raised
starting	Before any data is sent over the connection
received	When any data is received on the connection, provides this data
connectionSlow	When the client detects a slow or frequently dropping connection
reconnecting	When the underlying transport starts to reconnect
reconnected	When the underlying transport has reconnected
stateChanged	When the connection state changes; provides the old and the new state
disconnected	When the connection has disconnected

3.4.4 Configuring the connection

There are two ways to configure the connection before calling the start method: specifying a query string to be sent to the server, and specifying the transport method. The transport method is normally determined by the SignalR client negotiating with the server, but this can be bypassed. The usage of a query string is shown in Figure 30. Specifying the transport method is demonstrated in Figure 31. (Dykstra & Fletcher 2015.)

In Figure 30, usage of a query string is demonstrated both client-side and server-side. Client-side, query string data can be passed to the server both with (line 2) and without the generated proxy. Without the generated proxy, the connection must be accessed through the "$.hubConnection()" method (line 5). Then, query string data can be sent through this connection (line 6). Server-side, query string data can be accessed in the "OnConnected()" event handler (line 11). It is accessed through the "QueryString" property of the "Context" object (line 13).

In Figure 31, specifying the transport method is demonstrated both with and without the generated proxy. The transport method is specified in an object that is passed as a parameter to the "start()" method of the connection – a name "transport" is used (lines 2, 6, 10 and 15). If a single

transport method is specified, it is passed as a string in the object (lines 2 and 10). If multiple fallback transport methods are specified, they are placed in an array of strings (lines 6 and 15). These transport methods are attempted to be used from left to right (web sockets first, then long polling in the example). If the hub proxy is used, the “start()” method is called on the “$.connection.hub” object (lines 2 and 6). If not, it is called on the object returned by “$.hubConnection()” (lines 9 and 14).

```
1  // JavaScript with the generated proxy
2  $.connection.hub.qs = { 'version' : '1.0' };
3
4  // JavaScript without the generated proxy
5  var connection = $.hubConnection();
6  connection.qs = { 'version' : '1.0' };
7
8  // C# server
9  public class ContosoChatHub : Hub
10 {
11     public override Task OnConnected()
12     {
13         var version = Context.QueryString['version'];
14         // ...
15         return base.OnConnected();
16     }
17 }
```

FIGURE 30. Using a query string (Dykstra & Fletcher 2015).

```
1  // JavaScript with the generated proxy
2  $.connection.hub.start({ transport: 'longPolling' });
3
4  // JavaScript with the generated proxy,
5  // fallback scheme
6  $.connection.hub.start({ transport: ['webSockets', 'longPolling'] });
7
8  // JavaScript without the generated proxy
9  var connection = $.hubConnection();
10 connection.start({ transport: 'longPolling' });
11
12 // JavaScript without the generated proxy,
13 // fallback scheme
14 var connection = $.hubConnection();
15 connection.start({ transport: ['webSockets', 'longPolling'] });
```

FIGURE 31. Specifying the transport method (Dykstra & Fletcher 2015).

3.4.5 Error Handling and Client-Side Logging

An error event is provided by the SignalR JavaScript client for which a handler can be added. The “fail()” method can also be used to handle errors that result from a server method invocation. If detailed errors are not enabled on the server, the error message contains only minimal information about the error. Examples of error handling are shown in Figure 32. Lines 2-10 show how errors are handled both with and without the generated proxy. Lines 14-24 demonstrate the usage of the “fail()” method when invoking a server-side method with and without the generated proxy.

Client-side logging can be enabled by simply setting the logging property of the connection object to true before calling the start method. Examples of this are displayed in Figure 33. Line 2 shows how logging is enabled with the generated proxy, and lines 5-6 without.

```
// JavaScript with the generated proxy
$.connection.hub.error(function(error) {
    console.log('SignalR error: ' + error);
});

// JavaScript without the generated proxy
var connection = $.hubConnection();
connection.error(function(error) {
    console.log('SignalR error: ' + error);
});

// JavaScript with the generated proxy,
// method invocation
contosoChatHubProxy.newContosoChatMessage(userName, message)
    .fail(function(error) {
        console.log('newContosoChatMessage error: ' + error);
    });

// JavaScript without the generated proxy,
// method invocation
contosoChatHubProxy.invoke('newContosoChatMessage', userName, message)
    .fail(function(error) {
        console.log('newContosoChatMessage error: ' + error);
    });
```

FIGURE 32. Error handling (Dykstra & Fletcher 2015).

```
// JavaScript with the generated proxy
$.connection.hub.logging = true;

// JavaScript without the generated proxy
var connection = $.hubConnection();
connection.logging = true;
```

FIGURE 33. Enabling client-side logging (Dykstra & Fletcher 2015).

3.5 The .NET Client

3.5.1 Basics

The .NET client is in many ways similar to the JavaScript client. The .NET Hubs API can be used in, for example, Windows Store (WinRT) apps, WPF, Silverlight, Windows Phone and console applications (Dykstra & Fletcher 2014a).

Setting up the client requires that the Microsoft.AspNet.SignalR.Client NuGet package be installed. Also, if the SignalR version on the client is earlier than that on the server, SignalR can adapt to the difference, but not the other way around.

Before establishing a connection, a HubConnection object and a proxy must be created. Establishing the connection happens by calling the Start method on the HubConnection object. An example of this is presented in Figure 34. In the example, the HubConnection object is first obtained (line 1). Next, the Hub proxy is created (line 2). Finally, the Start method is called (line 3). The Start method executes asynchronously, and to make it execute synchronously the await keyword or the Wait method must be used. The HubConnection class is thread-safe. (Dykstra & Fletcher 2014a).

```
1 var hubConnection = new HubConnection("http://www.contoso.com/");
2 IHubProxy stockTickerHubProxy = hubConnection.CreateHubProxy("StockTickerHub");
3 await hubConnection.Start();
```

FIGURE 34. Creating the HubConnection object and proxy and starting the connection (Dykstra & Fletcher 2014a).

3.5.2 Configuring the Connection

Before establishing a connection, options can be specified. Examples of all these are presented in Figure 35:

- Query string parameters (lines 2-8)
- HTTP headers (line 11)
- Client certificates (lines 14-16)
- Concurrent connections limit (in WPF clients) (line 19)
- The transport method (line 22)

```
1  // query string data
2  var queryStringData = new Dictionary<string, string>();
3  queryStringData.Add("version", "1.0");
4  queryStringData.Add("parameter", "value");
5  var hubConnection = new HubConnection(
6      "http://www.contoso.com/",
7      queryStringData
8  );
9
10 // HTTP headers
11 hubConnection.Headers.Add("headerName", "value");
12
13 // client certificates
14 hubConnection.AddClientCertificate(
15     X509Certificate.CreateFromCertFile("MyCert.cer")
16 );
17
18 // concurrent connections
19 ServicePointManager.DefaultConnectionLimit = 10;
20
21 // transport method
22 await hubConnection.Start(new LongPollingTransport());
```

FIGURE 35. Connection configuration (Dykstra & Fletcher 2016a).

3.5.3 Methods That the Server Can Call

Methods that the server can call are defined by using the proxy's "On()" method in order to register an event handler. Examples are provided about methods without parameters (Figure 36), methods with specified parameter types (Figure 37), and methods with dynamic objects as parameters (Figure Z). Examples include code for Windows Runtime, Windows Presentation Foundation, Silverlight, and console clients (Microsoft 2017b, Microsoft 2017d). Removing an event handler happens when its Dispose method is called (Figure A). (Dykstra & Fletcher 2014a).

In Figure 36, an event handler is added in all four client types, with the function name specified (lines 2, 9, 16 and 24). The main function of the code examples is the output of text. In the first three event handlers, text box content is modified (lines 4, 11, and 19). In the console event handler, text is simply printed to the console (line 24). Each event handler is defined as a lambda expression with no input parameters. Lambda expressions are anonymous functions that are used to create e.g. delegates. They also enable passing functions as method arguments, as in the example cases. (Microsoft 2017c) On WinRT, WPF and Silverlight platforms the code is wrapped into a platform-specific function ("Context.Post()", "Dispatcher.InvokeAsync()", lines 3, 14, and 22).

In Figure 37, the code is somewhat similar to the previous example. The parameter type (Stock) is, however, specified when all the event handlers are created. Then, the parameter is made available as the generic lambda expression input parameter (lines 2, 13, 21, and 29). Figure 38 is otherwise similar to Figure 37, but the parameter types are omitted, thus making the parameters dynamic objects. Removing the event handler by using a stored reference to it is demonstrated in Figure 39, line 11. Storing the reference happens on line 2.

```
// WinRT
stockTickerHub.On("notify", () =>
    Context.Post(delegate {
        textBox.Text += "Notified!\n";
    }, null)
);

// WPF
stockTickerHub.On<Stock>("notify", () =>
    Dispatcher.InvokeAsync(() => {
        SignalRTextBlock.Text += string.Format("Notified!");
    })
);

// Silverlight
stockTickerHub.On<Stock>("notify", () =>
    Context.Post(delegate
    {
        textBox.Text += "Notified!";
    }, null)
);

// console
stockTickerHub.On("notify", () => Console.WriteLine("Notified!"));
```

FIGURE 36. Methods callable by server without parameters (Dykstra & Fletcher 2014a).

```
1  // WinRT
2  stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
3      Context.Post(delegate
4      {
5          textBox.Text += string.Format(
6              "Stock update for {0} new price {1}\n",
7              stock.Symbol,
8              stock.Price)
9      }, null)
10 );
11
12 // WPF
13 stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
14     Dispatcher.InvokeAsync(() =>
15         {
16             textBox.Text += string.Format(...);
17         })
18 );
19
20 // Silverlight
21 stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
22     Context.Post(delegate
23     {
24         textBox.Text += string.Format(...);
25     }, null)
26 );
27
28 // console
29 stockTickerHubProxy.On<Stock>("UpdateStockPrice", stock =>
30     Console.WriteLine("Symbol {0} Price {1}",
31         stock.Symbol, stock.Price);
```

FIGURE 37. Methods callable by server with parameters (Dykstra & Fletcher 2014a).

```
// WinRT
stockTickerHubProxy.On("UpdateStockPrice", stock =>
    Context.Post(delegate
    {
        textBox.Text += string.Format(
            "Stock update for {0} new price {1}\n",
            stock.Symbol,
            stock.Price)
    }, null)
);

// WPF
stockTickerHubProxy.On("UpdateStockPrice", stock =>
    Dispatcher.InvokeAsync(() =>
        {
            textBox.Text += string.Format(...);
        })
);

// Silverlight
stockTickerHubProxy.On("UpdateStockPrice", stock =>
    Context.Post(delegate
    {
        textBox.Text += string.Format(...);
    }, null)
);

// console
stockTickerHubProxy.On("UpdateStockPrice", stock =>
    Console.WriteLine("Symbol {0} Price {1}",
        stock.Symbol, stock.Price);
```

FIGURE 38. Methods callable by server with dynamic objects as parameters (Dykstra & Fletcher 2014a).

```
1  // creating the handler
2  var updateStockPriceHandler = stockTickerHubProxy
3      .On<Stock>("UpdateStockPrice", stock =>
4          Context.Post(delegate
5          {
6              textBox.Text += string.Format(...);
7          }, null)
8      );
9
10 // removing the handler
11 updateStockPriceHandler.Dispose();
```

FIGURE 39. Disposing of the handler (Dykstra & Fletcher 2014a).

3.5.4 Invoking Methods on the Server

Calling methods on the server happens by using the Invoke method of the Hub proxy. If the server method has no return value, the non-generic overload of the Invoke method must be used. On the other hand, if the method has a return value, it must be specified as the generic type of the Invoke method. Examples of these cases are presented in Figure 40. On line 2, a method call with no return value is presented. On line 6, the method is executed asynchronously with the await keyword. On lines 11 and 12, the same method is made to execute synchronously by calling Result on the returned object.

```
1  // no return value
2  stockTickerHubProxy.Invoke("JoinGroup", "SignalRChatRoom");
3
4  // asynchronous method with return value,
5  // complex type parameter
6  var stocks = await stockTickerHub.Invoke<IEnumberable<Stock>>(
7      "AddStock", new Stock() { Symbol = "MSFT" });
8
9  // synchronous method with return value,
10 // complex type parameter
11 var stocks = stockTickerHub.Invoke<IEnumerable<Stock>>(
12     "AddStock", new Stock() { Symbol = "MSFT" }).Result;
```

FIGURE 40. Invoking server methods from the client (Dykstra & Fletcher 2014a).

3.5.5 Error Handling and Client-Side Logging

If detailed errors are not enabled, the exceptions that SignalR returns contain only minimal information about the errors. Detailed errors should of course not be used in production, but they can be enabled for troubleshooting and debugging. Handling errors happens by adding a handler for the Error event of the connection object. Handling errors from method invocations can be accomplished by wrapping the invocations in try-catch blocks.

Client-side logging can be enabled by setting values to the TraceLevel and TraceWriter properties of the Hub connection object. An example of both error handling and client-side logging is displayed in Figure 41. On lines 2 and 3, an error event handler is added as a lambda expression. On lines 6 and 7, the logging variables are set so that all trace levels are outputted to the console. (Dykstra & Fletcher 2016a).

```
1 // error handling
2 hubConnection.Error += ex => Console.WriteLine(
3     "SignalR error: {0}", ex.Message);
4
5 // logging
6 hubConnection.TraceLevel = TraceLevels.All;
7 hubConnection.TraceWriter = Console.Out;
```

FIGURE 41. Error handling and logging in the .NET client (Dykstra & Fletcher 2014a).

4 IMPLEMENTATION: CASE MODULERP

4.1 The OWIN Startup Class

An OWIN startup class was created as an entry point to start and configure SignalR. There, the default message buffer size (the number of messages in memory) was set to 500 instead of using the default value of 1000 messages (Fletcher 2014b). Thus, memory usage was reduced. CORS was also enabled in the startup class. All CORS options were allowed, which allows all headers, all methods, any origin and any supports credentials (Microsoft 2016b).

4.2 Settings

There are two types of settings in ModulERP, implementation-specific settings defined in an XML file, and other settings that can be changed at run-time.

The /signalr/ URL for the implementation is defined in the XML file – for development, it was set to “http://localhost:4275/signalr/”. There are also IDs of timed functions defined in the XML file, and the function that broadcasts push messages is included there in order for the push service to be in use. These timed functions are executed at specific intervals. The XML file also contains the download file path and URL for download-type notifications – the files are stored in the location determined by the path and made accessible by using the URL.

The runtime settings contain three fields that can be modified. First, there is a checkbox that determines whether connection data is stored in the database or just in memory. Another checkbox determines whether the notifications are sent individually or in arrays – sending the notifications in arrays is more efficient, as multiple notifications can be sent at once. There is also an input field where the maximum array size can be set – if not set, the default is maximum 50 notifications per array.

4.3 The Database

All message (and possibly also connection) data is stored in the JL-Soft Oy's Microsoft SQL Server database. There are four tables, one of which stores message data, two that store receiver data, and one that stores SignalR connection data. The message table is described in Figure 42. The receiver data tables are described in Figure 43, and the structure of the connection data table is shown in Figure 44.

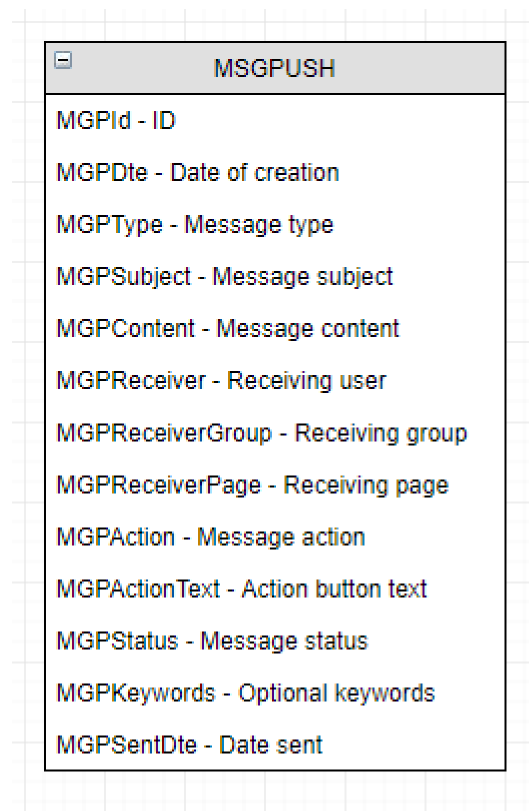

FIGURE 42. The message data table.

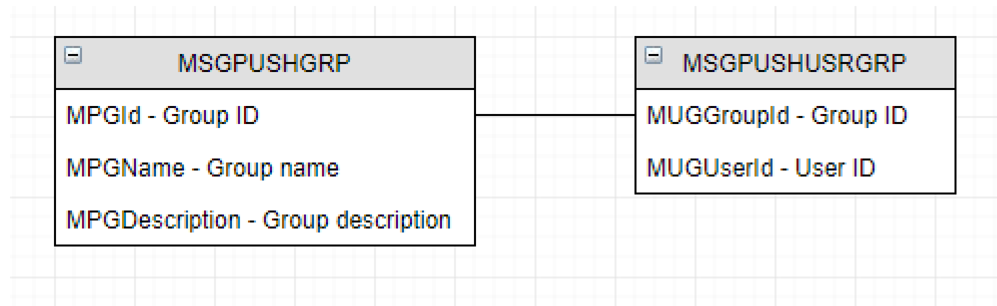

FIGURE 43. Receiver data tables.

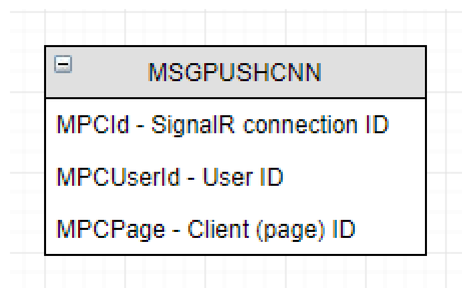

FIGURE 44. SignalR connection data table.

4.4 The Hub Class

The Hub class was created to actually implement SignalR. It manages all connections, messages etc. Connection information is stored in memory in a thread-safe ConcurrentDictionary. A ConnData class was created to store the user name and web page ID of each connection. The dictionary then maps SignalR connection IDs and ConnData objects together.

4.4.1 Connection Lifetime Events

The Hub handles two SignalR connection lifetime events: OnConnected and OnDisconnected. If in settings it has been determined that connection data will be stored in the database, the SignalR connection and user IDs and the page view name are inserted there when the OnConnected event is handled. Also, if the code runs for the first time after startup, the

connection database table is first cleared from leftover connection data. The connection information is always stored in the ConcurrentDictionary as well, for logging purposes. Last, pending messages are broadcasted to all the clients (browser windows) of the connecting user. When the OnDisconnected event fires, the respective connection information is removed from the database if the database mode is on, and from the ConcurrentDictionary.

The Hub methods callable by the clients are described in Table 3. See Chapter 3.6 for information about notifications and popups.

TABLE 3. Server methods callable by clients

Method name	What the method does
broadcastInstantPushMessages	Broadcast all messages labeled instant in the database
broadcastMultipleDismiss	Dismiss multiple notifications of one user in all the user's clients
changeArrayMsgStatus	Change the status of multiple messages in the database
changeMsgStatus	Change the status of a single message in the database
cpFileToShared	Copy a file to a download folder and download it
getArrayMaxLength	Gets the maximum message array length from the server
itemClicked	Dismisses a notification

4.5 Broadcasting from the Web Service

The messages are broadcasted from the ASMX web service of ModulErp. First, if connection data is stored in the database, all of the data is fetched. If array mode (see chapter Settings) is on, the maximum message array size is read from settings. If the maximum size is not defined there, the default is 50.

Next, a whitelist is created of users that have the push service enabled. The hub context of the Hub class is fetched in order to access Hub functionality.

There are several options regarding which messages are fetched from the database (see chapter 3.6):

- messages with the ‘instant’ flag to only speficic clients
- messages with the ‘instant’ flag to all clients
- notifications
- popups
- alerts
- messages only to a specific user
- all messages

The fetched messages are then broadcasted either in arrays or individually. The status and datetime when sent are updated after broadcasting.

4.6 SignalR on the Client

SignalR is initialized client-side every time a ModulErp web page is ready. First, the /signalr/ URL is read from the hidden field that holds it as its value. Then, the connection is started with web sockets specified as the primary transport.

4.6.1 Notifications and Popups

There are two implementations push messages in the ModulErp system: notifications and popups. For notifications, a task bar and a drop-down menu were created in the upper side of the web page. There are four types of notifications:

- info (default)
- critical

- download
- warning

The respective icons for all the notifications are shown in Figure 45. The task bar and a sample notification is presented in Figure 46. The bar contains the amount of each type of notifications, and the actual notifications can be accessed from the drop down menu. The notifications can be dismissed by clicking on them. Files can be downloaded by clicking on the download button of the notification.

Popups were created by using Telerik's RadNotification control. There are also four types of these popups. A sample popup is presented in Figure 47.

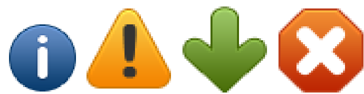

FIGURE 45. The icons for push messages.

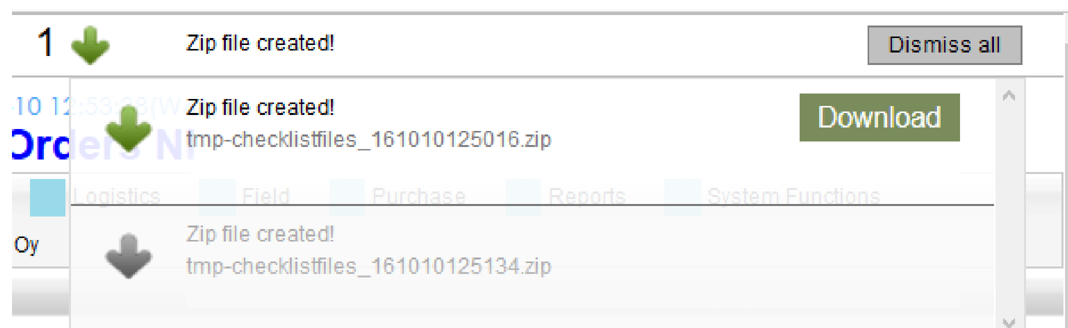

FIGURE 46. The task bar with the notification popup menu open.

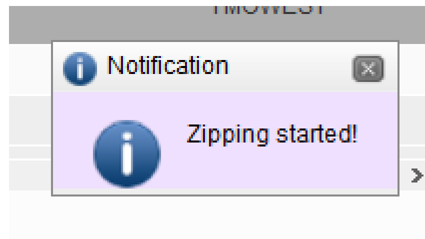

FIGURE 47. A notification popup.

4.6.2 Client-side methods callable by the server

Client-side methods that are callable by the server are described in Table 4. These were written in JavaScript.

TABLE 4. Client-side methods callable by the server

Method name	Description
appendNotif	Append a new notification to the drop-down menu
appendNotifsFromArray	Append possibly multiple notifications from an array
createPopup	Display a popup
dismissItem	Dismiss an item in the drop-down menu
dismissMultipleItems	Dismiss multiple items in the drop-down menu
downloadFile	Download a file
setArrayMaxLength	Set the maximum size of an array sent to the server

4.7 Implementations

There are two implementations of the push service in ModulERP. In the first one, notifications are sent while checking the size of attachment files in the system. These attachments always belong to a specific project. A notification is sent to the key user of the respective project every time an

attachment exceeds the maximum file size limit. The key user can then handle the file as needed.

In the other implementation, notifications are sent while zipping PDF files into archives of certain maximum size. If this size is exceeded, multiple ZIP files are created. The PDF files are end reports of work orders. The files are generated by the ModulERP system, and users can input data into them. After the compression is done, the ZIP archives are made accessible to the requesting user via download type notifications.

5 CONCLUSION

The goal of this thesis was to create a prototype of a push service with the SignalR library in the ModulERP solution, specifically its ASP.NET web interface. The goal was reached – SignalR turned out to be a versatile, easy to use library that greatly aids the creation of push services.

The prototype is ready, and there are implementations in addition to the push service itself, such as informing users of zipped files available for download. The prototype could be further developed to enhance performance and reliability. Reliability would be an issue e.g. in cases of one user having multiple clients (browser windows) open at the same time. Most user experiences were the result of development and debugging. After the prototype was showcased to actual users, small usability-related fixes were made to the task bar and dropdown menu.

The push service can be further developed – a chat, for example, could be added to the browser interface by using the service. A chat is just an example, as the possibilities are endless. If a new version would be developed, issues such as speed and performance could be addressed. Multiple Hub classes could also be created in contrast to the one single class in the prototype of the service.

SOURCES

Dykstra, T. & Fletcher, P. 2015. ASP.NET SignalR Hubs API Guide – JavaScript Client. The ASP.NET Site [cited 2nd June 2017]. Available: http://www.asp.net/signalr/overview/guide-to-the-api/hubs-api-guide-javascript-client

Dykstra, T. & Fletcher, P. 2014 a. ASP.NET SignalR Hubs API Guide – .NET Client (C#). The ASP.NET Site [cited 19th September 2016]. Available: http://www.asp.net/signalr/overview/guide-to-the-api/hubs-api-guide-net-client

Dykstra, T. & Fletcher, P. 2014 b. ASP.NET SignalR Hubs API Guide – Server. The ASP.NET Site [cited 18th December 2015]. Available: http://www.asp.net/signalr/overview/guide-to-the-api/hubs-api-guide-server

Fielding. R., et al. 1999. Hypertext Transfer Protocol – HTTP/1.1: Introduction [cited 27th April 2017]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec1.html

Fletcher, P. 2014 a. Introduction to SignalR. The ASP.NET Site [cited 12th May 2017]. Available: http://www.asp.net/signalr/overview/getting-started/introduction-to-signalr

Fletcher, P. 2014 b. SignalR Performance. The ASP.NET Site [cited 29th January 2016]. Available: http://www.asp.net/signalr/overview/performance/signalr-performance

Getify Solutions & Simpson, P. 2014. Defining Safer JSON-P [cited 11th January 2016]. Available: http://json-p.org/

Hausenblas, M. & Hossain, M. 2015. Enable Cross-Origin Resource Sharing [cited 18th December 2015]. Available: http://enable-cors.org/

JL-Soft Oy. 2017. JL-Soft [cited 14th March 2017]. Available: http://www.jlsoft.fi/fi/home_fi/

The jQuery Foundation. 2016. Deferred Object. jQuery API Documentation [cited 18th January 2016]. Available: https://api.jquery.com/category/deferred-object/

Microsoft. 2016 a. ASP.NET SignalR Reference [cited 16th February 2016]. Available: https://msdn.microsoft.com/en-us/library/dn440565(v=vs.118).aspx

Microsoft. 2016 b. CorsOptions Class. Microsoft Developer Network [cited 29th January 2016]. Available: https://msdn.microsoft.com/en-us/library/microsoft.owin.cors.corsoptions(v=vs.113).aspx

Microsoft. 2017 a. CorsExtensions.UseCors Method (IAppBuilder, CorsOptions). Microsoft Developer Network [cited 2nd June 2017]. Available: https://msdn.microsoft.com/en-us/library/owin.corsextensions.usecors(v=vs.113).aspx

Microsoft. 2017 b. Introduction to WPF in Visual Studio 2015. Microsoft Docs [cited 3rd July 2017]. Available: https://docs.microsoft.com/en-us/dotnet/framework/wpf/getting-started/introduction-to-wpf-in-vs

Microsoft. 2017 c. Lambda Expressions (C# Programming Guide). Microsoft Docs [cited 3rd July 2017]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions

Microsoft. 2015. Learn About ASP.NET SignalR. The ASP.NET Site [cited 15th December 2015]. Available: http://www.asp.net/signalr/overview

Microsoft. 2003. What Is RPC? TechNet [cited 18th December 2015]. Available: https://technet.microsoft.com/en-us/library/cc787851(v=ws.10).aspx

Microsoft. 2017 d. Windows UWP Namespaces. Windows Dev Center [cited 3rd July 2017]. Available: https://docs.microsoft.com/en-us/uwp/api/

Mozilla Developer Network. 2015 a. EventSource – Web APIs [cited 11th January 2016]. Available: https://developer.mozilla.org/en-US/docs/Web/API/EventSource

Mozilla Developer Network. 2015 b. Using server-sent events [cited 3rd October 2017]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events

Rouse, M. 2017. ERP (enterprise resource planning). TechTarget [cited 16th May 2017]. Available: http://searchsap.techtarget.com/definition/ERP

Rouse, M. & Steele, C. 2014. Push Notification Definition. TechTarget [cited 15th December 2015]. Available: http://searchmobilecomputing.techtarget.com/definition/push-notification

Anderson, R. & Thiagarajan, P. 2013. OWIN Startup Class Detection. Microsoft Docs [cited 1st June 2017]. Available: https://docs.microsoft.com/en-us/aspnet/aspnet/overview/owin-and-katana/owin-startup-class-detection

Wasson, M. 2013. Getting Started with OWIN and Katana. The ASP.NET Site [cited 18th December 2015]. Available: http://www.asp.net/aspnet/overview/owin-and-katana/getting-started-with-owin-and-katana

APPENDIXES