External Communication

# Beschreibung

External Communication dient als Komponente Senden und Empfangen von SignalR Nachrichten. Der XClient als die Schnittstelle zum Benutzer, verbindet sich via SignalR auf den von External Communication bereitgestellten Endpunkt.

Über die bereitgestellten Endpunkte ist es dem Benutzer möglich, die KPUs in einer generischen, modularen Weise zu steuern, je nach dem welche Schnittstellen, Modelle und Ansichten diese selbst bereitstellen.

Dazu kommuniziert External Communication aktiv mit den folgenden Diensten:

* Core Service
  + Aktiv: zur Interaktion mit den ‚dahinter‘ liegenden KPUs
  + Passiv: zur Rückgabe von angefragten KPU-Packages die zum ‚Anzeigen einer KPU‘ im Client benötigt werden
* Security Service
  + Aktiv: zur Authentifizierung des Benutzers und zur immanenten Berechtigungsüberprüfung bei jeder Benutzeranfrage
* Presenter Service
  + Aktiv: zur Anmeldung einer Verbindung für die Model Updates eines bestimmten Models
  + Passiv: zur Entgegennahme und Weitergabe von paketierten Model Updates an Clients

# Benutzung

External Communication stellt die einzige Komponente dar, die von außerhalb des Systems direkt angesprochen werden kann. Als Technologie für die Kommunikation kommt dabei SignalR Core zum Einsatz, welches netstandard-kompatibel ist und sowohl in .Net Framework, als auch .Net Core Applikationen eingesetzt werden kann.

Die Schnittstellen Client -> Server und Server -> Client, die für External Communication zum Einsatz kommen, sind im Projekt „AssistantViewerInterfaces“ der Assistant-Solution definiert.

* IAssistantViewer: Dies ist die Schnittstellendefinition für die Methoden, die im Client zur Verfügung stehen müssen. Zum tatsächlichen Ansprechen ist etwas mehr nötig, als nur das Interface zu implementieren. Dazu in Folge mehr.
* IExternalCommunicationHub: Dies ist das Interface für Methodenaufrufe mit denen ein Client beim Server Anfragen absetzt. Die wichtigsten Methoden hierbei sind
  + Login: Anmelden eines registrierten Benutzers
  + RequestExecute: Ausführen einer Aktion bei einer KPU
  + ReceiveViewRequest: Anfrage für ein Paket zur Instanziierung im Client für die spätere Anzeige einer KPU.
  + ReceiveSubscribeRequest / ReceiveUnsubscribeRequest: Anfrage für die Registrierung auf Model Updates einer bestimmten KPU

Um im Client „Nachrichten“ vom Server zu erhalten ist es nötig, einerseits die entsprechenden Methoden zu implementieren, andererseits diese dem SignalR-Clientobjekt bekannt zu geben.

In Fig 1 wird gezeigt, wie man in einer selbst geschriebenen Klasse „SignalRClient“ die entsprechenden Konfiguration und Registrierung vornimmt.

|  |
| --- |
| (…)  using Microsoft.AspNetCore.SignalR.Client;  (…)  class SignalRClient  {  private HubConnection Hub;  public async Task CreateConnection()  {  var connectionString = "http://127.0.0.1:4242/ecom";  Hub = new HubConnectionBuilder()  .WithUrl(connectionString)  .Build();  RegisterMethods();  await Hub?.StartAsync();  }  private void RegisterMethods()  {  Hub?.On(nameof(OnReceiveBadLogin), async () => await OnReceiveBadLogin());  Hub?.On(nameof(OnReceiveGoodLogin), async () => await OnReceiveGoodLogin());  Hub?.On(nameof(OnBatchDataReceived), async (string viewId, string[] items, string[] data, DateTime[] dateTimes) => await OnBatchDataReceived(viewId, items, data, dateTimes));  Hub?.On(nameof(OnMessageReceived), async (string message, string[] buttons) => await OnMessageReceived(message, buttons));  Hub?.On(nameof(OnPackageReceived), async (string viewId, string data) => await OnPackageReceived(viewId, data));  Hub?.On(nameof(OnDisconnect), async () => await OnDisconnect());  }  (…)  } |

Fig 1: Sourcecode SignalR Client 1

In diesem Beispiel werden zwei Methoden des SignalRClient gezeigt, „Connect“ und „RegisterMethods“. „Connect“ versucht dabei, eine Verbindung zu einem SignalR Server aufzubauen und speichert das Verbindungsobjekt über das mit dem Server kommuniziert werden kann im Property „Hub“. Mit „RegisterMethods“ wird dem Hubobjekt mitgeteilt, wie es mit vom Server hereinkommenden Anfragen umgehen soll. Die Anfragen bei SignalR entsprechen in beiden Richtungen Methodenaufrufen denen Parameter mitgegeben werden können, die Daten dafür werden allerdings nur serialisiert übertragen. D.h., dass der Methodenaufruf vom Server beim Client für die Methode „OnReceiveGoodLogin“, effektiv nur den Namen der Methode als Text überträgt (vereinfacht gesagt aber für das Verständnis der Thematik hilfreich). Um zur Laufzeit im Client zu wissen, was mit dem Text vom Server „OnReceiveGoodLogin“ angefangen werden soll, muss dies beim Hub hinterlegt werden, so zu sehen in der Zeile

Hub?.On(nameof(OnReceiveGoodLogin), async ()=> await OnReceiveGoodLogin());

Die Methode mit dem gleichlautenden Namen wird hier über den lambda-Ausdruck verwendet, um das Verhalten beim hereinkommenden Aufruf abzubilden. Wie die Methode dann aussieht, ist Implementierungsdetail des Clients, denkbar wäre ein Wechsel der Anzeige vom Login-Bildschirm zu einer nenügeführten Darstellung der verfügbaren KPUs. Auch ist es Implementierungsdetail, dass die implementierte Methode genauso heißt wie in der registrierten Schnittstelle, es ist hier allerdings hilfreich um den „nameof“-Operator verwenden zu können der den Methodennamen zur Compile-Zeit durch einen gleichlautenden String ersetzt.

Die andere Richtung der Nachrichtenübertragung, also vom Client zum Server, erfolgt ebenfalls über den Hub, wie Fig 2 zeigt.

|  |
| --- |
| public async Task LoginAsync(string user, string password)  {  await Hub.InvokeAsync("Login", user, password);  } |

Fig 2: Sourcecode SignalR Client 2

Hier werden lediglich die Parameter Username und Passwort an den Hub für den Aufruf der serverseitigen Methode „Login“ übergeben.

Anm.: Für Methoden mit Rückgabewerten sind Hub.On und Hub.InvokeAsync generisch geschrieben, sodass nahezu beliebige Methoden / Lambdas als Empfänger von verwendet werden können.

# Installation

External Communication wird auf einem Knoten eines Service Fabric Application – Clusters in der Service Fabric Application “Assistant” installiert.

# Konfiguration

NLog

# Abhängigkeiten

Sämtliche anderen Dienste der Service Fabric Application „Assistant“ müssen im Cluster mit installiert werden. Die Service Fabric Application „Access Control“ muss auf dem Cluster installiert und erreichbar sein.

External Communication benötigt folgende Bibliotheken die nicht vom Standard-Nugetserver geladen werden können:

BreanosConnectors.Kpu.Communication.Common

# Anforderungen

Zum sinnvollen Einsatz von External Communication muss der Port 4042 des Knotenrechners, auf dem External Communication installiert ist, erreichbar sein.

External Communication muss derzeit auf genau einem Knoten installiert werden, da der Dienst als „zustandsloser Dienst“ in der Service Fabric Application konzipiert ist und die Logik zur Model Update – Weitergabe vom Presenter Service aus sonst nicht funktionieren würde.

Logging kann über NLog mit einer konfigurierbaren Datenbankverbindung erfolgen und benötigt dafür dann ein erreichbares DBMS mit beschreibbarer Datenbank.

# Description

The service ExternalCommunication serves as a component for sending and receiving SignalR messages. The XClient which renders the user interface, connects to the endpoint provided by External Communication via SignalR.

By using the provided endpoints a user can control the KPUs in a generic and modular way, depending on the interfaces, models and views provided by the KPUs themselves.

For this, External Communication communicates with the following services:

* Core Service
  + active: interaction with the KPUs which are logically ‘beyond’ the Core Service
  + passive: returning requested KPU-packages that are necessary in the client to ‘display a KPU’
* Security Service
  + active: user authentication and immanent permission checks on every query from a client
* Presenter Service
  + active: registration of a connection for the model updates of a model
  + passive: forwarding of packaged model updates to clients

# Usage

External Communication is the component used for communication with SignalR clients. SignalR Core is used which is compliant to netstandard so it can be used both in .Net Framework and .Net Core applications.

The interfaces client -> server and server -> client used for External Communication are defined in the project “AssistantViewerInterfaces” of the Assistant solution.

* IAssistantViewer: this is the interface for methods the client has to offer to the server on its side. For the server to actually be able to call them, a little more boilerplate code is necessary, more on that later.
* IExternalCommunication: this is the interface for methods, the server offers to the client. Notable examples include:
  + Login: user authentication and association with a certain connection
  + RequestExecute: Execution of an action on a KPU
  + ReceiveViewRequest: Request for a package that a client needs to have and use in order to be able to display a KPU
  + ReceiveSubscribeRequest / ReceiveUnsubscribeRequest: Request for the subscription to Model Updates from a certain KPU

To receive “messages” from the server, it’s necessary to both implement the corresponding methods as well as let the SignalR-client object know about them.

Fig 1 shows the class SignalRClient as an example implementation for the hub configuration and registration.

In this example, two methods are shown, “Connect” and “RegisterMethods”. Connect attempts establish the connection with the SignalR server and stores the connection object in the property “Hub”. This object can then be used to communicate with the server. “RegisterMethods” informs the connection object about how to handle certain calls that are incoming from the server.

Calls in SignalR in both directions are essentially equivalent to calling methods in a program which can have parameters. However, the data is only transferred serialized via SignalR, meaning the server wants to call on the client results in little more than the name of that method being transferred as a string to the client. (it’s a bit more complicated but this is effective in wrapping one’s head around the concept).

To know during runtime, what to do with a call from the server that starts with “OnReceiveGoodLogin”, that information must be known to the Hub as can be seen in the line

Hub?.On(nameof(OnReceiveGoodLogin), async ()=> await OnReceiveGoodLogin());

The method with the same name is being called via a lambda expression to represent the behavior for the incoming call. How exactly the method is implemented is dependent on how the client is built, for example one could switch the view from the login screen to some menu display for all available KPUs. Another one of those details of implementation is that the method implemented in the client is named equal to the SignalR call which enables the use of the “nameof”-Operator which replaces the method name with its string representation during compile time.

The opposite way of sending messages, from client to server is also done through the Hub, as shown in Fig 2.

# Installation

External Communication is installed by deploying it within the Service Fabric Cluster Application “Assistant”.

# Configuration

NLog

# Dependencies

All other services of the Service Fabric Application “Assistant” also have to be deployed onto the cluster. The Service Fabric Application “Access Control” must also be deployed and accessible on the cluster.

External Communication requires the following libraries which are not available from the standard nuget server:

* BreanosConnectors.Kpu.Communication.Common

# Requirements

For External Communication to be used, port 4042 of the Node’s system it’s deployed on must be accessible.

At present, External Communication must be installed on exactly one Node of the cluster. This is because External Communication is designed as a “stateless service” with the Service Fabric Application and the forwarding logic for model updates wouldn’t work if it were running on more than one node.

Logging is implemented via NLog which can utilize an existing database as the logging target which then needs an accessible DBMS with writable database.