**Contents**

[**1** **Overview** 2](#_Toc448877367)

[**2** **Scope of the project** 2](#_Toc448877368)

[**3** **Description** 2](#_Toc448877369)

[**3.1** **Core Concepts** 2](#_Toc448877370)

[**3.1.1** **Task Hub** 2](#_Toc448877371)

[**3.1.2** **Task Activities** 2](#_Toc448877372)

**[3.1.3](#_Toc448877373)****[Task Orchestrations](#_Toc448877373)** [2](#_Toc448877373)

**[3.1.4](#_Toc448877374)****[Task Hub Worker](#_Toc448877374)** [3](#_Toc448877374)

[**3.1.5** **Task Hub Client** 3](#_Toc448877375)

[**4**  **Module Design** 4](#_Toc448877376)

[**4.1** **Writing Task Orchestrations** 5](#_Toc448877377)

[**4.2** **Writing Task Activities** 5](#_Toc448877378)

[**4.3 Orchestration Instance Management** 5](#_Toc448877379)

[**4.4 Error Handling & Compensation** 5](#_Toc448877380)

[**4.5 Task Hub Management** 5](#_Toc448877380)

[**5 Scenarios** 5](#_Toc448877381)

[**5.1 Post On Linkedin** 5](#_Toc448877381)

[**6 Conclusion:** 7](#_Toc448877386)

# Overview

The Service Bus Durable Task Framework provides developers a means to write code orchestrations in C# using the .Net Task framework and the async/await keywords added in .Net 4.5.

Here are the key features of the durable task framework:

* Definition of code orchestrations in simple C# code
* Automatic persistence and check-pointing of program state
* Versioning of orchestrations and activities

The framework itself is very light weight and only requires an Azure Service Bus namespace and optionally an Azure Storage account. Running instances of the orchestration and worker nodes are completely hosted by the user. No user code is executing ‘inside’ Service Bus.

# **Scope of the project**

Making use of different social platforms **in the implementation of the Durable Task Framework**. Here we automate the posts on social platforms. In the Durable Task Framework project, it is mainly used to post an update to the social networking platform LinkedIn using hardcoded.

# **Description**

The Service Bus Durable Task Framework allows users to write C# code and encapsulate it within ‘durable’ .Net Tasks. These durable tasks can then be composed with other durable tasks to build complex task orchestrations.

## **Core Concepts**

There are a few fundamental concepts in the framework.

### **Task Hub**

The Task Hub is a logical container for Service Bus entities within a namespace. These entities are used by the Task Hub Worker to pass messages reliably between the code orchestrations and the activities that they are orchestrating.

### **Task Activities**

Task Activities are pieces of code that perform specific steps of the orchestration. A Task Activity can be ‘scheduled’ from within some Task Orchestration code. This scheduling yields a plain vanilla .Net Task which can be (asynchronously) awaited on and composed with other similar Tasks to build complex orchestrations.

### **Task Orchestrations**

Task Orchestrations schedule Task Activities and build code orchestrations around the Tasks that represent the activities.

### **Task Hub Worker**

The worker is the host for Task Orchestrations and Activities. It also contains APIs to perform CRUD operations on the Task Hub itself.

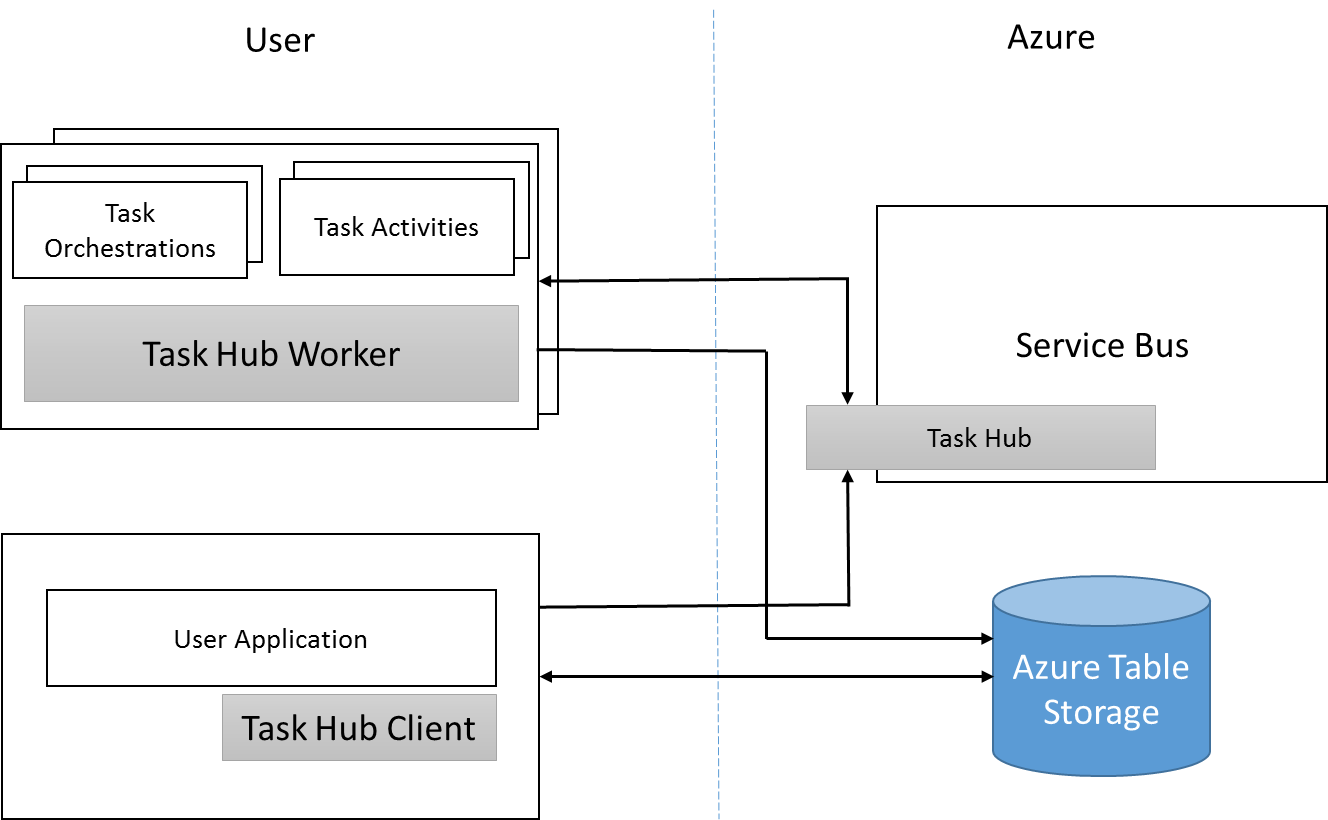
### **Task Hub Client**

The Task Hub Client provides:

* APIs for creating and managing Task Orchestration instances
* APIs for querying the state of Task Orchestration instances from an Azure Table

Both the Task Hub Worker and Task Hub Client are configured with connection strings connection strings for Service Bus and optionally with connection strings for a storage account.

Service Bus is used for storing the control flow state of the execution and message passing between the task orchestration instances and task activities. However Service Bus is not meant to be a database so when a code orchestration is completed, the state is removed from Service Bus. If an Azure Table storage account was configured then this state would be available for querying for as long as it is kept there by the user.



**For Durable Task Implementation we have**

**Task Orchestration:** TaskOrchestration.cs

**Task Activities:** AccessTokenTask, ShareLinkedinTask.

**User Applications: Linkedin**

The framework provides TaskOrchestration and TaskActivity base classes which users can derive from to specify their orchestrations and activities. They can then use the TaskHub APIs to load these orchestrations and activities into the process and then start the worker which starts processing requests for creating new orchestration instances.

The TaskHubClient APIs are used to create new orchestration instances, query for existing instances and then terminate those instances if required.

# **4** **Module Design**

Assume that user wants to build a code orchestration that will perform user authentication using access token and post an update to LinkedIn. To implement this using the Service Bus Durable Task Framework, the user will have to write different Task Activities, one for access token and one more for posting an update and one Task Orchestration that orchestrates between the Task activities.

In this orchestration, the user is scheduling one Task activity, and then waiting for the response and then scheduling the other Task activity. The framework will ensure that the state of the execution is preserved durably. E.g., if the node hosting the task orchestration above crashed before scheduling the second Task activity, on restart it will know to schedule this activity. If the node crashed after it had scheduled the activity but before the response came back, on restart it will be smart enough to know that the activity was already scheduled and it will directly start waiting for the response of the previous Task activity.

This is how users can load these orchestration and activity classes in a worker and start processing requests to create new orchestration instances. Multiple instances of these workers can be running concurrently against the same task hub to provide load balancing as required. The framework guarantees that a particular orchestration instance code would only be executing on a single worker at one time.

## **4.1** **Writing Task Orchestrations**

Task orchestrations basically invoke Task Activities and define how the control flows from one activity to another. The code that can be written within an orchestration is plain C# but with a few constraints. These constraints exist because of how the framework replays the orchestration code. This is described in a nutshell below.

Every time new work needs to be processed by an orchestration (e.g. a Task Activity finished or a timer fired), the framework replays the user’s TaskOrchestration code from scratch. Whenever this user code attempts to schedule a TaskActivity, the framework intercepts this call and consults the ‘execution history’ of the orchestration. If it finds that the particular TaskActivity had already been executed and yielded some result, it would replay that Activity’s result immediately and the TaskOrchestration would continue. This would continue happening until the user code has executed to a point where either it is finished or it has scheduled a new Activity. If it is the latter case then the framework would actually schedule and execute the specified Activity. After this Activity is completed its result also becomes part of the execution history and the value would be used in subsequent replays.

The Task Orchestration code is always executed in a single thread. This means that if the code was awaiting multiple tasks and one of them completed followed immediately by another one, the framework is guaranteed to run the continuations for both of these tasks serially.

**4.2** **Writing Task Activities**

Task Activities are the ‘leaf’ nodes of an orchestration. This is the code which actually performs a unit of operation within the orchestration. This is plain C# code with no constraints. Task Activity code is guaranteed to be called at least once. However in error cases it might be invoked multiple times so impotence is desirable.

## **4.3** **Orchestration Instance Management**

The TaskHubClient API allows users to create new orchestration instances, query for the state of created orchestration instances and terminate these instances. The API for creating an orchestration instance will return the instance information. This information can be used in subsequent APIs to query for the state of the instance.

## **4.4 Error Handling & Compensation**

Any exception that is thrown in the TaskActivity code is marshalled back and thrown as a TaskFailedException in the TaskOrchestration code. Users can write the appropriate error handling and compensation code that suits their needs around this. Refer Failure scenario’s below for Error handling.

## **4.5 Task Hub Management**

The TaskHubWorker has APIs that can be used to perform CRUD operations on the TaskHub itself.

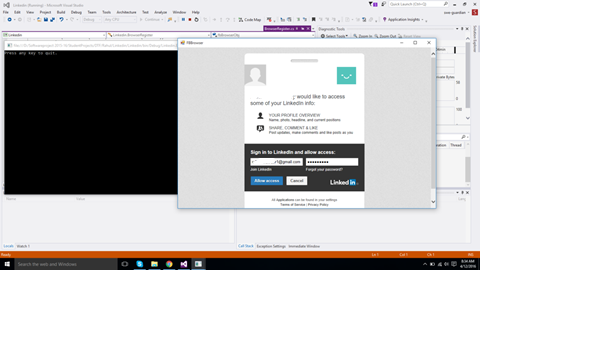
# 

# **5 Scenarios**

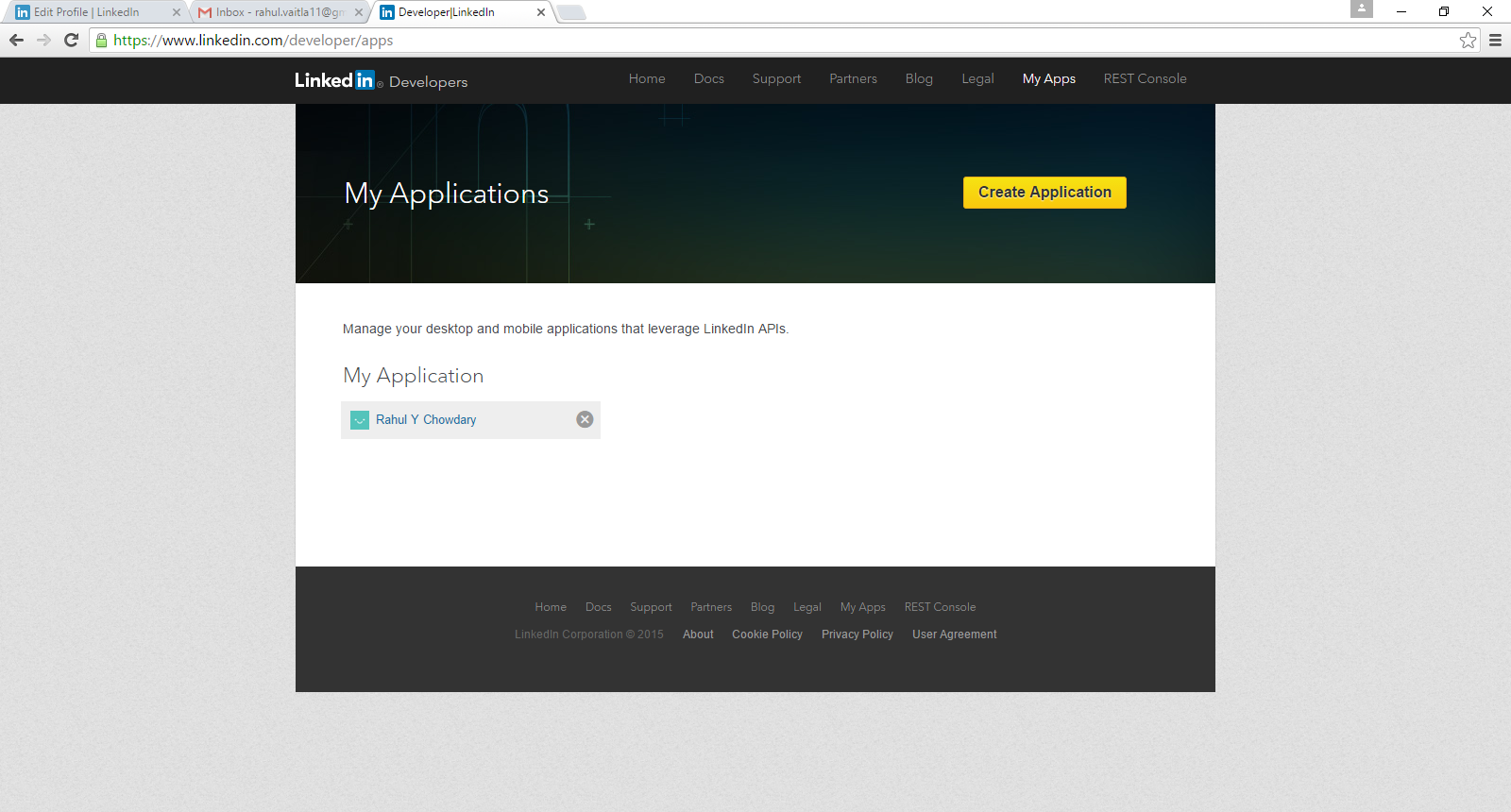
# **5.1 Post On LinkedIn.**

**LinkedIn Access Token .**

An access is opaque string for identifies a user, app and pages, can be used by app to make graph API call. For this, user have to provide his Linkedin user name and password.

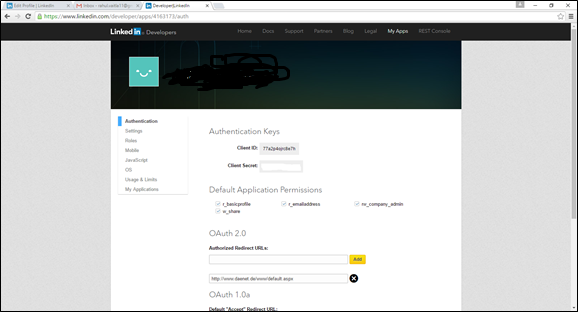


**Create app in LinkedIn**



**Use App ID(Client ID) in this link for collection LinkedIn Access Token .**

https://www.linkedin.com/uas/oauth2/authorization?response\_type=code&client\_id=77a2p4ojrc8e7h&scope=w\_share&state=STATE&redirect\_uri=http://www.daenet.de/www/default.aspx



# **6 Conclusion:**

Making use of different social platforms in the implementation of the Durable Task Framework. Here we succeeded in automating the posts on social platform LinkedIn. In the Durable Task Framework project, it is mainly used to posts an update to social platform LinkedIn.