Office 365 Timesheet

Extensibility Guide for Partners

Contents

[1 Overview 1](#_Toc519502667)

[2 Key Scenarios 1](#_Toc519502668)

[2.1 Microsoft 365 scenarios 1](#_Toc519502669)

[2.2 Other use cases 1](#_Toc519502670)

[3 Scenario Walkthrough 2](#_Toc519502671)

[3.1 Add more categories 2](#_Toc519502672)

[3.2 Connecting from existing UI 4](#_Toc519502673)

[APPENDIX: Extensibility Example 4](#_Toc519502674)

# Overview

Timesheet is designed to be easily extensible to support potential new requirements for partners and customers across multiple industry verticals and regions.

# Key Scenarios

Partners can take Timesheet to the next level by making use of the different kinds of data that is available to a Microsoft 365 user, that is accessible via Graph API.

# Microsoft 365 scenarios

Here are some scenarios that extend the capability of the solution by leveraging Office 365 features or workloads:

* Add activity data of other Office 35 workloads to compute work hours e.g Teams, Yammer activity data using Reports API (in this case there will be an average delay of 48 hours in the data)
* Include workhours data from external sources by importing it to SharePoint and compute total work hours for users
* Access Timesheet functionality (computation logic and calculated hours) by creating SharePoint UI and make it available to the organization as an internal site

Note that though it is possible to integrate additional work hours categories such as Skype, Yammer, Teams, Windows Login information and time spent on Office applications and other systems that use AAD authentication, not all of this information is currently accessible via the Graph API in the same format as for email and calendar items.

For example

* Skype activity is available only as aggregate reports that accepts timeframe as 7, 30 or 90 days instead of a specific date range
* Tasks activity is accessible with a date filter, but is not supported to run in the application context.

# Other use cases

Here are some other potential scenarios that could be taken up by partners in the future, for which the base is there in the timesheet application which can be used to build up advanced capabilities using available data and access:

* Access Timesheet functionality (computation logic and calculated hours) via an existing UI such as from a customer's web site, existing time tracking system or a Power BI dashboard
* Advanced notifications scenarios such as time reporting delays, missed time submissions and proactive communication of overtime to manager or HR based on computed hours without waiting for submission
* Support for invoking HR processes such as triggering an overtime mitigation workflow when an employee reports overtime more than once a week
* Analytics dashboard based on Timesheet data, with role-based views for IW, Manager, HR

# Scenario Walkthrough

This section gives an overview of the extensibility process and goes into more detail on two specific scenarios.

At a high level, once a scenario has been identified, such as adding a new category, a proposed plan of action is as follows:

|  |  |  |
| --- | --- | --- |
| Step | Description | Estimate |
| Planning | Research available options to get the required data and confirm feasibility of integration | 1 – 2 Days |
| Design | Identify areas that will need update in the middle-tier, Azure Functions and UI. Take note of exception scenarios such as handling of reports and where app context is not supported and do a PoC if needed | 1 - 3 Days |
| Implementation | Complete the implementation by making changes in the appropriate services, background tasks and UI touch-points | 2 – 3 Days |
| Validation | Do a full functional test pass of the solution across all personas | 1 Day |
| Deployment | Deploy the solution, taking care to update the lists, app settings and any new Azure functions especially when updating an existing instance | 1 Day |

# Add more categories

One key area of extensibility is support for additional categories. At this time, the solution supports three different categories of work hours:

* Email
* Meeting
* Other (currently used to enable user to account for any additional time under other categories)

This can be extended to include other categories such as activity in Teams or Yammer, Skype Meetings, Office application usage and Login activity information.

Adding support for a new category involves changes in two areas:

* Compute Logic

Computation logic for calculating work hours for a user will have to be updated to include the additional category, which may necessitate adding new settings in the application configuration to customize as needed for each deployment instance.

* User Interface

The three supported categories (Email, Meeting, Other) are clearly differentiated in the Edit Hours interface to provide further insights into the total hours at a day level. When adding a new category, one should consider adding the information as a separate item here or in other areas such as in the Dashboard, or alternately add them to the Other category from a visualization standpoint

As an example, let us consider the case where a partner wants to add a new service, say, Tracker, to the work hours compute logic.

To accomplish this, the first step is to update Settings files (Appsettings.json in the Dashboard and host.json in Azure Functions) to add a new property for time calculation configuration for Tracker activity. Following this, one will need to update the following files in both middle tier and Azure Functions:

**SharePointListsSchemaHelper.cs**

Add the following columns to the GetWorkHoursJsonSchema method:

* TrackerHours
* TrackerMinutes
* TrackerAdjustedHours
* TrackerAdjustedMinutes

**GraphAppYammerService.cs**

Add new file under /services with all the logic needed to get the Tracker data using Graph API (once it supports application context)

**WorkHoursRepository.cs**

Add the logic needed in the ComputeHours, ComputeHoursforWeek and ConvertToWorkHours methods to calculate and update the SharePoint list with the Tracker values using the configuration in the app settings.

**WorkHours.cs**

Add the Tracker fields added to the SharePoint list to the WorkHoursFields class

# Connecting from existing UI

A partner can integrate an existing application with Timesheet by directly connecting to the interfaces in the middle tier which will help fetch the data using Graph API, persist the data in SharePoint and access them in another user interface or use the data for additional calculations and reporting. The compute logic will also need to be updated in associated Azure functions.

To proceed with this, a developer will need to refer to the following files:

* GraphSharepointService.cs to query work hours data
* WorkHoursRepository.cs to calculate and update data in the user work hours list in SharePoint

# APPENDIX: Extensibility Example

# Scenario

Extent the Timesheet application to use an additional data source for Work Hours. This could be, for instance, hours information from an existing solution deployed within the organization. To demonstrate how this can be done, a CSV file is used to simulate the external data source.

# Approach

**Source CSV file**

* Consists of Tasks completed by an IW, tracked via an external system
* File name specified in appsettings
  + Could be published to wwwroot as a fixed location that can be updated by the admin
  + Another approach is to specify a SharePoint location, which can be updated as needed by the admin
* Static file accessed one-time for each active user
  + To extend this, the file could be saved in SharePoint or some such accessible location and read for first time login for each new user

**Computation Logic**

* Based on Task Type - Specific time can be set for Simple and Complex tasks…SimpleTaskTime(eg: 30 mins), ComplexTaskTime(60 mins)

**User - First Time login**

* As part of first time login, reads the CSV file and if there are any entries for the user, adds that to a separate SharePoint list, TimesheetUsers\_TaskHours, and updates the work hours list for the user under 'Other Hours' category

**Outcome** – User sees any tasks reported in the CSV file reflected under Other Hours for the specific date after first time login

Ignore data before prev month

**Sample CSV File**  (Note that only ObjectID, Date and TaskType are required for the Timesheet computation logic)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| UPN | ObjectID | Date | Task Type | Start Time | End Time |
| user@tenant | <id> | 1-Jun | Simple | 10:00 AM | 12:00 PM |

# Implementation

Summarized below is the set of steps involved in implementing the functionality:

* Update app settings in the Dashboard project and the Azure Functions project to include three new keys: TaskHoursFileLocation, SimpleTaskHours, ComplexTaskHours
* Add a class library to read additional tasks data from a csv file, specified in app settings. The csv file is parsed in the library and the data is sent back to the calling code wherein hours are computed based on the task types

using System;

namespace GoLocal.TimeWise.AzureFunctions.Models

{

    public class TaskHours

    {

        public string Upn { get; set; }

        public string ObjectID { get; set; }

        public DateTime Date { get; set; }

        public string TaskType { get; set; }

        public string StartTime { get; set; }

        public string EndTime { get; set; }

    }

}

* Modify TimeTrackerOptions in middle tier and Azure Functions to read the three new values in app settings at run time
* Update WorkHoursRepository.cs to read and process the Tasks data
* Implement new service, GraphAppsTaskService, in middle tier and Azure Functions to handle Tasks data

namespace GoLocal.TimeWise.AzureFunctions.Services

{

public class GraphAppTasksService : GraphAppServiceBase

{

private readonly ILogger \_logger;

private readonly TimeTrackerOptions \_timeTrackerOptions;

private readonly string \_userObjectIdentifier;

/// <summary>

/// Constructor

/// </summary>

public GraphAppTasksService(

IGraphAuthProvider authProvider,

IUserContext userContext,

// ILogger<GraphAppTasksService> logger,

TimeTrackerOptions timeTrackerOptions,

string userObjectIdentifier) : base(authProvider, userContext)

{

// \_logger = logger ?? throw new ArgumentNullException(nameof(logger));

\_timeTrackerOptions = timeTrackerOptions ?? throw new ArgumentNullException(nameof(timeTrackerOptions));

\_userObjectIdentifier = userObjectIdentifier;

}

        public async Task<List<TaskHours>> GetUserTasksAsync(DateTime startDate, DateTime endDate)

        {

            try

            {

                var userObjectIdentifier = \_userObjectIdentifier;

                var tasks = new List<TaskHours>();

                using (var reader = new StreamReader(\_timeTrackerOptions.TaskHoursFileLocation))

                {

                    reader.ReadLine();

                    while (!reader.EndOfStream)

                    {

                        var line = reader.ReadLine();

                        var values = line.Split(',');

                        var dateCheck = DateTime.Parse(values[2].ToString());

                        var identifierCheck = values[1].ToString();

if (dateCheck <= endDate && dateCheck >= startDate && identifierCheck == userObjectIdentifier)

                        {

                            TaskHours temp = new TaskHours();

                            temp.Upn = values[0].ToString();

                            temp.ObjectID = values[1].ToString();

                            temp.Date = DateTime.Parse(values[2].ToString());

                            temp.TaskType = values[3].ToString();

                            temp.StartTime = values[4].ToString();

                            temp.EndTime = values[5].ToString();

                            tasks.Add(temp);

                        }

                    }

                }

                return tasks;

            }

            catch (ServiceException ex)

            {

                switch (ex.Error.Code)

                {

                    case "RESTAPINotEnabledForComponentSharedMailbox":

                    case "Request\_ResourceNotFound":

                    case "ResourceNotFound":

                    case "ErrorItemNotFound":

                    case "itemNotFound":

                        return new List<TaskHours>();

                    case "TokenNotFound":

                        //await HttpContext.ChallengeAsync();

                        throw;

                    default:

                        throw;

                }

            }

        }

    }

}

* Update WorkHoursRepository.cs to read and process the Tasks data