Developing SharePoint add-ins

Björn Jansson
Sebastian Maghsoudi

Handledare: Mahder Gebremedhin, IDA, Linköpings Universitet
Maria Fredriksson, Solution Xperts, Linköping
Examinator: Peter Fritzson, IDA, Linköpings Universitet
Upphovsrätt
Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright
The publishers will keep this document online on the Internet – or its possible replacement – from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Björn Jansson and Sebastian Maghsoudi.
Summary
Developing a Microsoft SharePoint-based site is, according to marketing, simple and efficient. Efficiency is increased by giving the developer the option to easily add functionality to the site. One way to add functionality is through so called add-ins. An add-in extends the functionality of the site by running independent code on the client, cloud or a server outside the SharePoint farm. Add-ins can be added to the site through a web-based market or by creating new ones. The general question is if creation of new add-ins is worth the effort. This discussion can begin by investigating what implementation detail is reoccurring when developing this add-ins as well as what part of the process the most time is consuming. By attempting to create add-ins and the same time documenting the progress one can make conclusion regarding said questions. The result shows that the most recurring parts of development is how to manage the incoming data. It also seems as if the most time consuming part of add-in development is the research and construction of functionality that fetch data from the SharePoint-based site. Another part of the result is technical and contains information necessary to understand how the example add-ins are created. The add-ins that is presented in this paper concern document management which is a significant aspect of Microsoft SharePoint.
## Contents

Summary .......................................................................................................................... 3  

1. Introduction .................................................................................................................. 6  
   1.1 Motivation ................................................................................................................ 6  
   1.2 Purpose .................................................................................................................... 6  
   1.3 Limitations ............................................................................................................... 6  
   1.4 Related work ........................................................................................................... 7  
   1.5 Target audience ...................................................................................................... 7  
   1.6 Keywords ................................................................................................................ 7  

2. Theory ........................................................................................................................... 8  
   2.1 SharePoint 2013 Architecture ................................................................................. 8  
   2.2 SharePoint add-ins .................................................................................................. 8  
   2.3 Hosting Options ...................................................................................................... 9  
   2.4 Client-Side Programming ....................................................................................... 10  
      2.4.1 CSOM ............................................................................................................... 10  
      2.4.2 REST .............................................................................................................. 11  
      2.4.3 MVVM ........................................................................................................... 13  
   2.5 Add-in security ....................................................................................................... 14  
      2.5.1 Authentication .............................................................................................. 14  
      2.5.2 Permissions ................................................................................................. 15  

3. Method .......................................................................................................................... 17  
   3.1 Environment .......................................................................................................... 17  
   3.2 Logging .................................................................................................................. 17  

4. Implementation ............................................................................................................. 19  
   4.1 Checked out documents ....................................................................................... 19  
   4.2 My drafts .............................................................................................................. 22  
   4.3 Most popular documents ...................................................................................... 23
1. Introduction

1.1 Motivation
Microsoft SharePoint is a web-based developer’s platform used by organizations and companies around the world. SharePoint has evolved gradually since its launch in 2001 and by every version, new functionality has been added. SharePoint has since become a more or less mandatory presence in any given Microsoft Windows based corporate workplace. One of the most important (and most used) feature of computers is their ability to create and manage documents. SharePoint focuses on document management by allowing online possibilities between co-workers. Considering that there are almost endless needs in this context, there should by implication exist an endless amount of solutions. SharePoint allows developers to extend the functionality of a site developed in SharePoint through several approaches. Most of these approaches can be categorized as using built in functionality to achieve desired results. Another approach is to extend current functionality with custom developed software depending on the needs at hand. This is done by adding so called add-ins (apps) to the SharePoint site. An add-in is, in the context of SharePoint, completely independent code that runs externally and is therefore not harmful. The add-in can use the API's of REST and CSOM.

1.2 Purpose
The purpose of this work is to investigate the efficiency in regards to the resulting power given to the developer and the corresponding investment of effort (time and understanding). This task is given to the authors by a consulting firm (Solution Xperts AB). Consulting firms that want to offer SharePoint solutions to clients may find it efficient to possess reusable add-ins that can be reused in multiple projects. The firm has proposed that this can be done by developing four add-ins in order to make an assessment and conclusion based on experiences.

Add-ins that are being developed:

- Viewing and checking in checked out documents.
- Viewing drafts available to the user.
- Show most popular documents.
- Show documents related to the user.

During the development of these four add-ins the following research questions are investigated:

- What implementation details present themselves repeatedly during development of add-ins for SharePoint?
- Which parts of add-in development are most time consuming?

These research questions are asked and answered through the perspective of the authors, whom have a three-year education in computer engineering and no earlier experience with SharePoint development.

1.3 Limitations
This paper only focuses on SharePoint-hosted add-Ins later described). Also, there will not be a lot of explanation in regards to how the server and environment was configured. This is because this is not relevant to the actual purpose which if anything revolves around finding general patterns rather the specific technicalities.
1.4 Related work
A closely related study was made 2013 at KTH Royal Institute of Technology in which SharePoint 2013 was used also. The purpose of that study, as in this study, was to evaluate the SharePoint add-in model [2]. The result was a task management system in the form of an add-in. The results showed the following advantages and disadvantages of the add-in model:

Advantages:
- Add-ins are easily available for the end-user.
- The end-user can expand the functionality in a cheap or free manner.
- The reusability of existing code improves.

Disadvantages:
- The documentation provided by Microsoft is good but the concept is new and knowledge from other developers is hard to find.
- Setting up certificates and configure the add-in to communicate with SharePoint is a complicated process which is time consuming.
- An add-in from an unknown source could contain malware and could therefore be unsafe to use.

One concrete difference between the study mentioned above and this one is the hosting option. This study focus on the SharePoint-hosted option while that study uses the Provider-hosted option.

Another difference is that this study tries to identify common problems that occur during the development of an add-in and also what part of development is most time consuming. The study made by Haddad, E. and D. Abdelke aims to identify the positive and negative effects of the add-in model.

1.5 Target audience
The reader is assumed to be knowledgeable in basic web-programming in order to completely absorb the content of this paper. Nevertheless, this paper will include a list of keywords that the reader will encounter multiple times. The theory is designed to give full coverage of more specific information.

1.6 Keywords

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CSOM</td>
<td>Client-Side Object Model</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
</tr>
<tr>
<td>JSON</td>
<td>Javascript Object Notation</td>
</tr>
<tr>
<td>MVVM</td>
<td>Model View ViewModel</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>URL</td>
<td>Unified Resource Location</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
2. Theory

2.1 SharePoint 2013 Architecture

The SharePoint service itself is applied in so-called farms that can be considered the lowest level of server software [3]. A farm consists of one or multiple servers that have the same SharePoint system installed. These servers can be classified into three different types. These types are web, application and database servers. A web server will store the actual SharePoint site and is therefore the only required server type in a SharePoint-farm. Because the classification is focused on what the server is dedicated to do and because the dedication require cohesion in order to reach maximum efficiency, each classification can be considered a level of abstraction within the farm. The web server level is the one closest to the end-user. The next level is that of application servers which consists of servers that is dedicated to providing SharePoint associated service applications. According to Alirezaei, Schwartz et al. (2013) application servers are usually connected to larger network farms with need for high computing capacity. The final and lowest level consist of database servers that is dedicated to managing databases through SQL queries. Database servers can take requests from both web and application servers.

2.2 SharePoint add-ins

An add-in is an independent and self-contained extension of SharePoint [7]. An add-in is made up of several properties. Primarily an add-in attempt to satisfy a specific need by extending the functionality of the SharePoint-site. Another important property is that the risk of the add-in code potentially harming the site is next to nothing since the code runs either on the client, cloud or on another server outside of the SharePoint-farm. The language used depends on the hosting option which is later explained, the web-programming is done in JavaScript. The third property is that there is a possibility to access SharePoint data through API's (REST or CSOM which is later explained). There are some other properties an add-in have but only one of them will be presented due to the context. The final property is how the add-in is connected to the interface (see figure 2.1).
2.3 Hosting Options
One of the first things to decide when developing a SharePoint add-in is the way it should be hosted. There are two alternatives [1], the first is called SharePoint-hosted, meaning when SharePoint itself hosts the add-in. The second alternative is called cloud-hosted which is when the application is hosted remotely, both provider-hosted and auto-hosted add-ins are contained within the notion cloud-hosted.

- **SharePoint-hosted add-in**: This kind of add-in as said will be hosted by SharePoint itself but this also means that it will be restricted to client-side code because it is executed on the client’s browser.

- **Provider-hosted add-in**: This alternative gives the developer a lot more freedom because he/she is in control of the system. The drawback is that the add-in needs to be hosted by the provider and made accessible to the internet or a local network.

- **Autohosted add-in**: This type of add-in is hosted by Microsoft Azure, the great thing about this is that the provider does not need to host it, the drawback is that Microsoft’s servers restricts the system. This alternative is only available when using the Office 365 environment.
2.4 Client-Side Programming
The Microsoft SharePoint 2013 app model does not support running server-side code within the SharePoint host environment [1], unfortunately this means that the developer will not have direct access to the site data. This is where the Representational State Transfer (REST) and Client-Side Object Model (CSOM) comes in handy. These are Application Programming Interfaces (API) which are designed to request the information from the SharePoint host environment either from the client’s browser or from another host.

2.4.1 CSOM
CSOM was introduced in SharePoint 2010 and offered a way for developers to access some of the core SharePoint functionality [1], before this could developers only access a small part of it. There are two different ways of using CSOM (actually there are three but the Silverlight version is mostly used for mobile development), with C# (then called managed client object model) or JavaScript. They are kept in different libraries and offer different programming interfaces but both communicates with SharePoint using the framework Windows Communication Foundation (WCF).
To request information from SharePoint, developers use the object model which is then translated to XML by a proxy (as seen in figure 2.3) and then sent to the host environment. The reply is delivered in JavaScript Object Notation (JSON) format.

2.4.2 REST
REST stands for Representational State Transfer and is defined as design principles that highlights web service functionality [8]. The principles are defined as follows:

- Only use HTTP methods defined by RFC 2616 (GET, POST, PUT etc.)
- Use complete and independent requests to achieve statelessness.
- The structure of URI’s should be similar to a directorial structure.
- The type that is transferred should be either in the form of XML and/or JSON.

What follows now is how SharePoint have implemented REST.

The REST API uses the standard HTTP GET, POST, PUT and DELETE verbs to perform CRUD (Create, Read, Update and Delete) operations on the SharePoint host environment, they are most easily sent using the JavaScript API jQuery [1]. Operations are created using Uniform Resource Identifiers (URI) in the URL of the HTTP request.
The code in listing 2.1 shows a quite easy example of how client-side code can make a request to get the object with information about the current user, note that this is not a cross-domain request (user object is stored at the add-in site).

- `_spPageContextInfo.siteAbsoluteUrl` will return the actual URL of the Add-In site.

- `/_api/web/currentuser` specifies the information requested, the current users object.

- The request is of type `GET`, this means that already existing information is requested, no information is to be added or changed.

- If the SharePoint site answers the request successfully will the text property of the `message` label be changed to the current user's title.

- An error object will be sent from the SharePoint site if it cannot answer the request successfully. The message of the error object will be printed out in an alert box to the user.
In order to make a cross-domain request a few things have to be added, one of them is the function `getQueryStringParameter` shown in listing 2.1. The function returns certain URLs when given correct keywords. The function is called two times with two different keywords. The function will return the URL for the SharePoint site when called with the keyword `SPHostUrl`. The URL of the site which hosts the add-in will be returned when it is called with the keyword `SPAppWebURL`, this will also include the ID which makes the site specific for each add-in.

```javascript
function getQueryStringParameter(param) {
    var params = document.URL.split("?")[1].split("&");
    var strParams = "";
    for (var i = 0; i < params.length; i = i + 1) {
        var singleParam = params[i].split("=");
        if (singleParam[0] === param) {
            return singleParam[1];
        }
    }
}

hosturl = decodeURIComponent(getQueryStringParameter("SPHostUrl"));
appurl = decodeURIComponent(getQueryStringParameter("SPAppWebUrl"));
```

**Listing 2.2: Example of a function needed to retrieve certain URL’s**

The request made with the URL in listing 2.2 asks for the `SimpleLibrary` items, because the `SimpleLibrary` is stored on the SharePoint site is a cross-domain request is needed. An important thing to remember is that cross-domain requests cannot be synchronous.

### 2.4.3 MVVM

When developing add-ins you generally use very much client-side code, generally JavaScript. Hillier Scott[1] said “The biggest challenge in implementing apps with JavaScript is the intimate relationship between the webpages and the libraries”. To solve this problem, developers often use the Model-View-ViewModel (MVVM), especially when developing with JavaScript and REST. MVVM aims to separate the displaying of data, the logic and also the actual data.

![Figure 2.4: Illustration of Model-View-ViewModel][1]

---

[1]: 1

**Figure 2.4:** Illustration of Model-View-ViewModel [1]
MVVM consists of three parts all of which can be seen in figure 2.4, View, Model and ViewModel. View consists of the actual webpages that defines the design of the add-in. Model are the libraries, lists along with other stuff that SharePoint store data in. Last but definitely not least is the ViewModel which connects these two, this is where most if not all JavaScript will be placed.

Knockout is a library which is really good for achieving the much needed loose coupling. Knockout uses two key items, declarative bindings and dependency tracking [1]. Declarative bindings make it possible to bind elements in the HTML to JavaScript functions instead of doing it the other way around. Information changes and when it does will dependency tracking make sure that it is updated on all the other places it is used. This means that if an application is dependent on information in an array and that information changes it will be notified with the new data.

2.4.3.1 Promises
Many times it is needed to make many request to SharePoint using REST, the problem is that many times they depend on each other and will therefore need to be sent sequentially even though REST works asynchronous. To fix this one can add the following request in the success part of the last request, the problem with this is that you will get a lot of code in a big hierarchy which will be hard to manage. A promise will make it possible to write the code at another location, this will make the hierarchy a lot more readable.

When making a request and use a promise along with it will the promise return an empty object immediately. This empty object will be filled with information when the response is received[1]. This simplifies the code a lot because you can structure it in a much better way. To create a promise one can use the jQuery method “$.Deferred”. A promise also avoids unnecessary REST request by returning success or failure if the promise already has been made.

2.5 Add-in security
In order to understand add-in security, there are first some aspects that must be clarified. First and foremost, add-ins has dedicated functionality that can be integrated to SharePoint and are themselves to greatest extent a separate software and should also be considered as such. But because the current functionality of SharePoint, some of its properties can partly be extended to the integrated add-in. With this taken into consideration the next issue of order is to explain what security in the context of add-ins is. In SharePoint the security revolves almost exclusively around protection of data. The report will henceforth name an attempt to access data a request.

2.5.1 Authentication
Authentication is the process that verifies the identity of the requester [1]. Basically the authentication process answers the question “Who you claim to be?”. since the user themselves has to make some sort of identification of themselves. SharePoint does not itself contribute any specific software for authentication but instead relies on external components. An example of such a component is Windows security token which is generated for each user and then stored, mapped to his/her logon name. However, later versions of SharePoint have implemented a so called SAML token system which is a XML based standard markup language. This is a move towards a more claim-based standard, meaning that the system can identify what a user actually is within the network. The Windows security token is still used or at least replaced by a FBA-token. However, this needs to be converted to a SAML-token because of that recent SharePoint version preferably use them, and this is done by a service application integrated within SharePoint called Security Token Service. The tokens are usually stored and cached on the web-server in order to reuse the same token for the same user.

Up until now the Authentication process has be described generally for users. But since the focus in this paper concerns add-ins, will the following piece concern how SharePoint authenticates
these. The process is significantly different from a user authentication process. An add-in authentication can either be done internally or externally [3]. The host environment is called through either CSOM or REST and these are therefore necessarily used for proper authentication. The process starts by determining if the call was made by an add-in, this is followed by determination to authenticate it internally or externally. In order to decide if the call was initiated by an add-in, the SharePoint host observes what kind of token is used within the call. Add-ins use a special token type called *access token* for external authentication which means it is rather easy to determine if the call was made by an add-in. However, the host cannot assume that the call was made by a user simply by determining that it was made using a SAML-token. To conclude the source, one need to inspect the calls target URL by checking if it corresponds to an app-web.

![Diagram](image)

**Figure 2.5: Determining Authentication Type [1].**

### 2.5.2 Permissions

Permissions is applied towards SharePoint objects such as Top-level sites, Child-sites, lists and list-items [1]. On a lower abstraction layer, each previously mentioned securable object has the possibility to include an access control list (ACL). Each ACL informs what each user or app is permitted to with the object (read, write and execute). It is not mandatory for a securable object to possess an ACL string which causes another problem. The issue that arises is that there is no apparent way to know what permission is associated with an object that does not have an ACL, however, this is solved by the structure of the permission hierarchy in SharePoint. The hierarchy is site based, so the Top-level securable object corresponds to the Top-level site. If there is any children-sites, these also has their corresponding securable object. Because of this implementation, the hierarchy is rather obvious (Top-level -> Children-level -> Library -> Item). Because not all objects must include an ACL string it can create a problem which is now easily solved. If the object has its own ACL, the system should simply apply this information. Otherwise should simply apply the object higher in the hierarchy.

An add-in developer will have to specify which securable objects the add-in want access to, this is done in the *AppManifest.xml* file. Access to these objects must be granted with a so called permission request, this is a pop-up window which will be displayed when installing the add-in. An example of such a window can be seen in figure 2.6, the window asks the user if he or she trust the software that would run on the site by presenting what specific permissions it requests. The user who grants the add-in these permissions by pressing the *Trust It* button must also have these permissions. It is important for users and administrators which possess a high level of permission to be really careful before granting access to the add-in.
Figure 2.6: Example of a Permission Request [1].
3. Method
The issues are completely inclined towards add-in development as a process, which leads to
the demand for a well formulated reflection of the work. In order to be able to look back at the
development process of the four add-ins, logs will be authored and later reflected upon. The
add-ins themselves will also undergo evaluation in order to verify to what extent they fulfill the
requirements.

3.1. Environment
The setup became somewhat restricted because the company specifically requested
SharePoint-hosted add-ins. The option of provider-hosted add-ins was not preferred since it
would have required the company or its clients to host the add-ins and this is something they
wanted to avoid. To understand why a SharePoint-hosted add-in restricts the developers please
refer chapters 2.2 and 2.3.

The add-ins are developed on a virtual machine running the Windows Server 2012 operating
system. Visual Studio 2015 is installed on the machine and is the development tool used for
creating the add-ins. Visual Studio provides the project templates, wizards, good debugging
capability, and also a tool which makes it easy to deploy to the SharePoint site. These features
along with many more make developing add-ins convenient. TortoiseSVN¹ is used for version
control.

When developing an add-in, it is always necessary to have a SharePoint site in order to test it.
Two service applications have to be installed on the site in order for add-ins to work, the app
management service and the security token service application. The app management service
can be installed through the graphical interface but the security token service application has
to be installed using PowerShell.

Since a SharePoint hosted add-in needs its own app-web, there is need for the website to have
the app-web URL. In order to achieve maximum compatibility with the site there must be an
alias record within the DNS corresponding to the SharePoint site. This will allow the add-in
IDs to be attached to the URL.

3.2 Logging
The logs will be linked to MVVM that is previously discussed in the theory–chapter. The logs,
therefore, always categorize each log contribution as either corresponding to the model, view
or the viewmodel In order to achieve continuity, the logs will be written at each workday when
some sort of software development was made. Each contribution will be authored by both
developers so that the content is always discussed and mutually agreed upon. In addition to
increasing the overall quality of the logs, discussion helped the authors realize things that
otherwise would not have been noted if each developer had authored his own log.

All log-contributions will highlight implementation details through some sort of classification.
This means that the implementation details will be put in contrast with others observed during
the development process and similarities and differences are highlighted. This can later be
summarized and discussed in a more perspicuous manner.

The logs will also include how much time was consumed on each part. This will be done in
detail to greatest extent possible in order to be able to classify them at a later point in time. In
this regard some classification criteria are needed since it will be easier to make proper
analysis. The template for the logs can be seen in appendix B.

¹ https://tortoisesvn.net/
4. Implementation

As mentioned earlier, to get the necessary data needed to reach a well formulated conclusion, four add-ins will be developed. The add-ins are designed according to the employer’s need for everyday purposes when using MS SharePoint. There are other ways to achieve similar functionality through simply creating a so-called web-part but they have some limitations due to the fact that they can only contain already developed functionality. Add-in coding allows for better control of design, integration of different functionalities, and better access to more informative data. In order to achieve usability, the add-in should include a useable app-web and an app-part which can be added directly to the site by an administrator. An add-ins app-web is its own site; an app-part makes it possible to mount the add-in on an existing website. The difference is presented visually in appendix A.

The following subsections will motivate each add-in by giving background to the each of the problems they attempt to solve or needs they attempt to satisfy. A general description of desired architecture is also presented after the background for each add-in.

4.1 Checked out documents

Multiple issues arise when users work in a shared library environment and to access the same documents. These issues are usually solved with some sort of version control systems and SharePoint does not deviate from this. SharePoint allows users to check out documents they want to edit. When a document is checked out by one user it cannot be edited by another user. This makes it impossible for collisions to occur. However, the user that checked out the file also needs to manually check in it as well. This versioning approach can cause other issues. The most obvious is when a user unintentionally skips checking in documents thereby preventing other users within the organization from editing those documents.

This problem can be solved with an add-in that allows users to view all documents that are, by themselves, checked out. Also there will be an option to check in the documents directly through the interface. In order to make it easy for the user, the GUI should not deviate radically from the interface already implemented in SharePoint. This means that the interface should have similar lay out, containing the same values.

![Image of SharePoint list](image)

**Figure 4.1: An example of a SharePoint 2013 list within a library.**

As seen in Figure 4.1 the list does not have a lot of actual content with directly viewable metadata and other information. In fact, the folder child count is completely unnecessary for non-folders.
The checked-out column is also obsolete in this case since the user can only view items checked-out by themselves. Another thing that needs to be similar to the already existing environment is the flow of the functionality (how the user uses the functionality).

![Image of a document library and check-in window]

**Figure 4.2: Traditional Approach to Checking in a Document.**

As illustrated in figure 4.2, the first step is to actually find and select the document the user wishes to check in. Once found, the user must select to check in through the extended properties window (accessed by clicking on the triple dots). Once the user enters the check in window, they are given two options (if they are both activated properly), either to check in a minor version or to check in a major version (publish version). Both options allow the user to add a comment which can be viewed later in the version history. This process is rather straightforward and the add-in should provide a similar functionality.

The resources available in this case, as explained earlier, are very dependent on the hosting option chosen. Due to of simplicity reasons, the add-in should be SharePoint-hosted if possible. Within SharePoint add-in development the REST API is better suited for SharePoint-hosted add-in development[1]. The REST API allows the developer to access files and folders from libraries as well as to possibly check in/out the documents. This means that the REST API has enough features to satisfy the functionality for this add-in.
The development process will follow the architecture of MVVM and the project will apply this abstraction through the file structure. The Model will have functionality to fetch lists from libraries and then documents from those lists. Also the Model will be able to check in documents and fetch the user-id. The View will contain multiple HTML <div> elements which will contain a list and a graphical interface for checking in a document. The ViewModel will work on a higher abstraction level and use the other modules to create the add-in itself. The ViewModel will work in accordance with the flow chart displayed in figure 4.3. It also contains the functionality for checking in the file.

**Figure 4.3:** Flowchart for viewing list of checked out files.

**Figure 4.4:** Checking in document.
4.2 My drafts

SharePoint uses versioning for files which means that when a file is checked in it becomes a later version of itself. There are two different version types: major and minor[5]. A major version only uses a whole number while minor versions also use decimals. Both of which can be seen in figure 4.5. The first marked file, file_1, is of version 1.1 which means that it is a minor version, the version of the other marked file, file_2, is version 1.0 which is a major version.

A minor version is often called a draft. Drafts are not published which means that only people which have the appropriate permissions can read and edit them while people which do not have the extended permissions can only read the latest published (major) version.

![SampleLibrary](image)

**FIGURE 4.5: VIEW OF LIBRARY WITH VERSIONING ACTIVATED**

The client requested an add-in which simply lists all drafts. This will enable users that work on many different items to acquire an oversight of which documents that are yet to be published. The add-in is similar to the *Checked Out documents* add-in which is presented in chapter 4.1. They both will retrieve the documents in the same way. The only difference is the condition which decides what documents to view. This means that a SharePoint-hosted add-in should suffice and that the MVVM-model is appropriate.
Figure 4.6 shows the intended flowchart for the logic of the add-in. It can be seen from the flowchart of this add-in that it is similar to the Checked Out documents add-in discussed earlier.

4.3 Most popular documents
Each document in SharePoint can be downloaded as a file by any user with access to it. Depending on whether Office web apps are installed on the server or not, it is also possible to directly view them in the browser. This could be considered to be the same as opening any URL on the SharePoint farm thus using the same statistics for usage is possible. These statistics can help the user find out how many times a document was viewed. This can be a useful tool to give an insight of which documents are most accessed.

The process to reach a conclusion of what document is most popular on the site is rather lengthy. First, one has to navigate to each library after which they would then enter the most popular window for each of this libraries. The user may want to open a popular document and not necessarily open a document because it is popular and the process described forces the user to actively look for a popular document. Also, in order to find the most popular documents, the user has to visit each library and remember the total views number to compare it to the other values in other libraries.
The add-in that is developed solves all of the mentioned issues by simply viewing the most popular documents on the entire site. This will be done both in an app-web as well as a web-part. The REST API provides search queries feature. Since all the needed key values are available through a search query, the REST API will be used for this purpose. Because it is possible to use the REST API, developing a SharePoint-hosted add-in is sufficient for the needed functionality.

The add-in will follow the MVVM structure similar to the previous add-ins. This means the REST API calls will be a part of the model. The only necessary REST call will be a search query which filters away non-documents as well as sorting the results on total views. The view will simply allow listing of the relevant documents in both the app-web and a web-part. The ViewModel in this case will be rather simple since there is no more to do than get a search result of a very static query and then simply forward the result to the view.

4.4 Related documents

The users on a SharePoint farm can have the possibility to build their own MySite where they can communicate with other users, follow documents and add so called tags. A tag can be considered a property that can be added to any document by any user. A tag is supposed to correspond to the content of a document. The tags can be followed by a user through adding that tag to another document, meaning a user is using that specific tag. Every time a document is tagged with a specific tag that a user follows it is displayed on the tags information and it will be possible to reach the document through the tag.

From the window displayed in figure 4.8 users can enter the “Tags & Notes” window. They can then add tags to the selected documents.
For each tag that already exists, users can add tags from a dropdown window that comes up once they start writing something in the box that is similar to an existing tag. Once the tag is in the text box, the user simply clicks save and the tag is added and will be visible to other users as well. Any other user can now find the documents that have been tagged.

![Tags and Note Window](image)

**FIGURE 4.9: TAGS AND NOTE WINDOW.**

As seen in figure 4.10, the process consists of two steps once the user enters “Mysite” which makes it a three step process.

![Finding Documents with a Certain Tag](image)

**FIGURE 4.10: THE PROCESS OF FINDING DOCUMENTS WITH A CERTAIN TAG.**

The add-in that is developed makes it possible to view documents that are tagged with tags the user is associated with directly from an app-web or an app-part. In order to solve this, the REST API will be used in a SharePoint-hosted environment. The REST API will be used in order to get the tags a user is associated with as well as fetching the tags of the user’s documents.
Since the add-in will be SharePoint-hosted, keeping the MVVM as an abstraction is still a reasonable approach. The model will in this case contain functionality to get tags the user is associated with as well as fetching the tags associated with documents. The View will be able to list the documents both in an app-web as well as a web-part. The ViewModel will work on a higher abstraction in accordance with the flow chart presented in Figure 4.11.
5 Results

This section presents the results for each individual add-in and also the log-summary. When discussing each add-in, the first part will explain some implementation details that are very specific for the add-in. This part will appear very technical while the second part simply present the resulting add-in. The last subsection in this chapter highlights the general information that can be extracted from the logs.

5.1 Checked out documents
5.1.1 Implementation details

Every time a REST API call is made from the ViewModel it is necessary to implement promises through `jQuery.deferred()`. The reason for using promises in the ViewModel is primarily for structural reasons. The structure in the code looks like any other program because it does not need to take into consideration that the REST calls are made asynchronously. As mentioned in Log 1.1, because the functionality in the deferred object allows the data to populate an object when it arrives. Problems can arise when multiple calls use the same deferred object. There is no obvious disadvantage to using multiple deferred objects. This is the better than using the same deferred object for all returned value.

```javascript
$.ajax({
    url: appurl +
    '/_api/SP.AppContextSite(@target)/Web/CurrentUser?$select=Id' +
    hosturl + '
    ',
    type: 'GET',
    headers: {
        "accept": "application/json;odata=verbose",
        "X-RequestDigest": digest
    },
    success: function (data) {
        deferredUserId.resolve(data)
    },
    error: function (err) {
        deferredUserId.reject(err);
    }
});
```

**LISTING 5.1: MODEL REST CALL FOR USER ID.**

Each of the returned values in the model are returned as a unique deferred object. This means there cannot be collisions because of asynchronous issues since each returned value is completely detached from any another. Even though there is an increase in coupling between the modules it will still result in a more comprehensible code because of the variable naming becomes clearer and more precise. The deferred object can now be fetched by different modules and be picked apart by simple accessing through key values. Listing 5.1 shows that the fetched data will contain the active users own ID.

```javascript
REST.Request.UserID().then(
    function (data) {
        promiseduser = data.d
        UserId = data.d;
    },
    function (err) {
        printError(err);
    }
)
```

**LISTING 5.2: FETCHING OF USER DATA FROM THE VIEWMODEL.**
Since JavaScript is not an object-oriented language, modularization has to be implemented by the developers themselves to a much greater extent. As seen in Listing 5.2 the ViewModel accesses the function through an object called request that exist in a namespace named REST.

In order to fetch the meta data for the items, one needs the name of the library where the items are located. To get the names of the available libraries, another REST call has to be done specifically targeting the metadata of the libraries.

```javascript
$.ajax({
    url: appurl + "/_api/SP.AppContextSite(@target)/web/lists?$filter=BaseTemplate eq 101&$select=Title,RootFolder/ServerRelativeUrl,EnableMinorVersions&$expand=RootFolder&@target='"+ hosturl + "'",
    type: "GET",
    async: false,  //most certainly does not work
    headers: {
        "accept": "application/json;odata=verbose",
        "X-RequestDigest": "__REQUESTDIGEST".val()
    },
    success: function (data) {
        deferredlists.resolve(data)
    },
    error: function (err) {
        deferredlists.reject(err);
    }
});
```

**LISTING 5.3: REST CALL TO FETCH LIST META DATA.**

As seen in Listing 5.3, the call is actually targeting lists on the site. This is because a library is considered a list by SharePoint. The name and title will be the same in this case and the usefulness of the other selected data will be explained at a later point. In Figure 5.3, there is need for a loop in order to analyze each library meta data in order to use the name to fetch that specific library item's metadata.

After the metadata of the list is fetched the loop can simply increment through the list titles which is the same as the names given to each library. As presented in Figure 4c there is second loop that is nested within the first one to analyze the item data. In order to fetch the metadata for the items within a library the add in use a “get-by-list-title”-type access query.

```javascript
$.ajax({
    url: appurl + "/_api/SP.AppContextSite(@target)/web/lists/getByTitle('" + ListName + "')/items?$select=FileRef,Modified,CheckoutUserId,AuthorId,OData__UIVersionString&@target='" + hosturl + "'",
    type: "GET",
    headers: {
        "accept": "application/json;odata=verbose",
        "X-RequestDigest": "__REQUESTDIGEST".val()
    },
    success: function (data) {
        deferreditems.resolve(data);
    },
    error: function (err) {
        deferreditems.reject(err);
    }
});
```

**LISTING 5.4: REST CALL TO FETCH ITEM META DATA.**
As seen in Listing 5.4 the ListName is entered to access the list’s items. In order to fetch the filename, fileref must be selected since the name is not a distinct part of the files metadata but rather its reference. CheckoutUserID will be the ID of the user that has checked out the document. This field will be null if the document is not checked out by any user.

The process of listing the documents begins with checking which documents are checked-out and by which user after comparing with the active user’s id. If the ids match the documents should be listed with all their needed values and hyperlinks. The hyperlinks are put together with the root folder information fetched in the library and the file reference from the item itself.

```javascript
$.ajax({
    url: appurl + 
    
    "/_api/SP.AppContextSite(@target)/web/GetFileByServerRelativeUrl(" + 
    
    relurl + 
    
    ")/CheckIn(comment='" + 
    
    comment + 
    
    "',checkintype="" + type 
    
    ")?@target='" + 
    
    hosturl + 
    
    "',
    type: "POST",
    headers: {
        
        "ACCEPT": "application/json;odata=verbose",
        
        "X-RequestDigest": digest//$"__REQUESTDIGEST".val(),
    },
    success: function (data) {
        deferredCheckInCall.resolve(data);
    },
    error: function (err) {
        deferredCheckInCall.reject(err);
    }
})
```

**Listing 5.5: Model REST call to check in document.**

The listed documents can now be checked-in from the interface and need of another REST call is done to achieve this. In Listing 5.5 the type is required to represent if the check-in is supposed to be of the type major or not. A comment has to be added to the URL as well.

5.1.2 Resulting add-in
The interface of the add-in can be accessed both from the site content page, meaning an actual access to the app web, or through an app-part. The interface shows all the necessary information of a document such as when the document was last modified, the location and which version it is.
As seen in both Figure 5.1 and the user has the possibility to check in the viewed documents. When the user clicks check in, they will be presented with a window shown in Figure 5c. The user can then enter a comment which can later be viewed in the version history for each document. They also have the option to either check in a major or minor version. However, if the library the document is located in does not support minor versioning, this option will not be selectable. Once the user has checked in the documents that item will be removed from the list without the app-part or the app-web having to refresh.

5.2 My drafts
5.2.1 Implementation details
The My drafts add-in is inherently similar to the Checked out documents since both need to access the same metadata of the items. One of the only differences in the model is that there is
no need to fetch any check out information for neither the libraries or the items. The URL needs to be slightly different in this scenario since it should exclude some parts.

```
url: appurl + "/_api/SP.AppContextSite(@target)/web/lists/getByTitle(" + 
  ListName + 
  ")/items?$select=FileRef,Modified,CheckoutUserId,AuthorId,OData__UIVersionString&@target=" + hosturl + ""
```

Listing 5.6: URL for rest call to fetch item meta-data.

Another difference is in how the add-in chooses which documents to view. The add-in is supposed to view all drafts of all documents available to the user. The add-in need to extract if the document is a minor version or not. In order to do this, it uses the OData__UIVersion – element which is specified to be in string format. Simply by checking if the string contains a “.0” as two consecutive characters, the add in can assume the document is a major version. This is because there are no situations where this can occur if the document is a minor version.

5.2.2 Resulting add-in
The interface for the My drafts add-in is similarly to the Checked-out documents add-in in the sense that it needs to be accessible through its own app-web as well as an app-part.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Library</th>
<th>Modified</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage1.xlsx</td>
<td>SampleLibrary</td>
<td>2016-05-27 11:36:02</td>
<td>0.1</td>
</tr>
<tr>
<td>lol/happydays.docx</td>
<td>SampleLibrary</td>
<td>2016-05-31 12:27:28</td>
<td>0.1</td>
</tr>
<tr>
<td>internetfile.html</td>
<td>SampleLibrary</td>
<td>2016-06-02 08:20:46</td>
<td>0.2</td>
</tr>
<tr>
<td>internetfile.docx</td>
<td>SampleLibrary</td>
<td>2016-06-02 08:20:39</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Figure 5.3: App-web of the “My Drafts” – add-in

As seen in Figure 5.3, the documents that are displayed, only are of the version that does not end with “.0”. This means that none of the documents viewed are of a major version.
5.3 Most Popular
The purpose of this add-in is to display the most used files on the SharePoint system in sorted order. This was achieved using ViewsLifetime feature which is available through office web-apps.

5.3.1 Implementation details
During the first period of implementation for this add-in it seemed like the add-in had to access the SharePoint database in order to retrieve statistics for the document including how many times they had been viewed. This would not be possible using only client-side code so this forced the add-in to be provider-hosted. A lot of time was spent trying to achieve this but without success. However, when working towards this we found the property ViewsLifetime. ViewsLifetime is available through office web-apps which required a separate server, including the SharePoint server that makes two servers. A physical server was not available so a virtual machine was used instead. Setting up an office web-apps server proved to be quite difficult, especially without previous knowledge about office web-apps.

Office web-apps, along with SharePoint, creates analytical information about documents and how they are used. For example, how many times each document has been downloaded or opened. This is how ViewsLifetime acquire the information needed.

Listing 5.7 shows the URL used in the query which. Notice that the results are to be sorted using ViewsLifetime in a descending order. This will result in the most used documents first. Another thing to notice is that only the top five results will be returned due to last part of the URL, i.e., ($top=5$).

The file types have to be specified in the URL which caused some minor issues. It seemed like a bad idea to just add all file types to the URL string. Instead the function add_filetypes_to_url was created. This function stores all file types in the array file_types and then appends them to returnstring which is then returned and thus added to the URL string. This makes it easier to maintain and extend the file types. The function can also be used on other places if needed and avoids having a lot of duplicate code.
5.3.2 Resulting add-in
The add-in was successfully completed but unfortunately not tested which can be seen in Figure 5.4 where the amount of times a document has been viewed is of value null. This is due to office web-apps and SharePoint being incorrectly configured causing analytical statistics to not be generated. An obstacle on the way to solve this was that the statistics takes at least a few days to be generated when working. This means that it takes a few days to verify whether a configuration is correct or not. This study was limited in time so no working configuration was found in time.

The user can click both the filename and the location. Clicking the filename will open the file in the browser. Unfortunately this is only possible for a small number of file types. The file will be downloaded if it cannot be opened by the browser. Clicking the location will navigate to that locations view (URL).
5.4 Related documents
Tags in SharePoint are used to share information with colleagues [6]. Users can also search documents by tags. The results will then be files related to that tag. The purpose of this add-in was to display documents which the user might be interested in based on tags.

5.4.1 Logs and implementation details
There were two options which had the potential to achieve the desired functionality.

1. The tags could be found using a REST request so using these in order to find the documents seemed like a good approach. A conclusion was made after a couple of days that this was not possible because the information needed was not available through either REST or CSOM.

2. The other possibility was to find the documents by making a search using the tags. This was also showed to be impossible since this feature is outdated or at least not possible version of SharePoint.

The add-in was not possible to create because it was heavily dependent on missing SharePoint functionality. It is worth mentioning that this functionality once existed in SharePoint but has been removed. It might be possible to create this add-in using a provider-hosted but it is unlikely due to the fact that a sharepoint-hosted add-in can access the same functionality. One way might be if the provider-hosted add-in could obtain a direct access to the SharePoint database. It might be possible to develop using a web-part instead but that will not result in a add-in which makes it useless for this study.

5.5 Log summary
During the implementation of the add-ins a total of 37 logs where authored. Each log highlights (by the authors) a considered task. These tasks can sometimes be divided into sub-tasks since each task is viewed from as high abstraction layer as possible.

\[2\] A task is here defined as a piece of work that is attempted to be done. The task should in general be atomic without losing context.
As seen in Figure 5.5, more than half of the tasks completed or attempted could be classified as a part of the Model while a third are in ViewModel and finally a sixth can be connected to the View.

Figure 5.6 shows the order in which part of the MVVM that required most time. The order hasn’t changed from Figure 5.6. The distribution simply suggest that the time consumed for part of the project is not proportional to the number of tasks per part. If possible it could be beneficial for a well formulated conclusion to classify the part of the model into sub categories since the effort in completing this section of the software corresponded to almost three quarters of the time consumed during the development phase.

5.5.1 Subcategories of Model
The category corresponding to the effort for completing the Model has been defined by the developers as everything that is not considered to be connected to ViewModel or View.
Therefore it seems reasonable to try to categories even further. Investigating the logs suggests the following categories:

- Server configuration
- Syntactical research
- Miscellaneous

**Figure 5.7: Distribution of Time Classified in Accordance with Model Re-Categorized**

When evaluating Figure 5.7, it is worth to mention that both View and ViewModel could also be further categorized since the task division is a lot more evenly distributed in comparison to Figure 5.5.

**Figure 5.8: Distribution of Time Classified in Accordance with Model Re-Categorized**

In Figure 5.8 it is clear that over half of actual time was spent on syntactical research and server configuration.
6. Discussion
This chapter will discuss and reflect what has been observed from the results and also investigates the method used in order to retrieve the results. It also covers areas where there is a potential for further improvement and future work.

6.1 Results
In order to get a broader picture of the project it is important to discuss the log-summary. Figure 5J show that almost a third (31%) of the time is used for server configuration. This is a lot but also rather expected for developers without any previous SharePoint experience. A more experienced developer would probably have set up a basic SharePoint environment at least a couple of times before, making it easier for said developer to do it again. Setting up the basic environment is of course extremely important but what has been discovered is that the add-ins are often very dependent on various different resources provided by different SharePoint processes. This means one also have the know-how in setting up these as well.

Figure 5J also show that another 31% of the time is spent on syntactical research. What is meant by syntactical research is simply trying to figure out how to write the program, finding the SharePoint resources needed, and how they are accessed using REST. The conclusion that can be made from this is that a familiarity with SharePoint is a lot more important than it appears at first.

Another observation to be made is that only 3% of the time was spent on the view. This can seem very minor but it depends on how the add-in is supposed to be used. The add-ins created during this study are meant to be used on an existing page in the form of an app-part which limits the design. They basically only consisted of a single HTML-table.

6.2 Method
The though behind the method used was based upon the idea that there is need to make a contribution relevant to the evaluation of the work process more than any other aspect. There is probably none or at least very few methods that wouldn’t involve some form of documentation. Since the same individuals that makes the analysis also are the persons that implements the software, the type of documentation could not be that much different to collect if it were to collect the necessary information.

The method utilized in this study has some flaws. The first flaw being the lack of evaluation. Originally one part of the method was called evaluation and it was supposed to evaluate the individual add-ins by letting people with different background test them and give feedback. It can be considered applied to a limited extent since the add-in was shown to our supervisor at the company. However it was not evaluated by anyone that was not involved in the development process in some way. Gathering up a group of people, letting them test the add-ins, and collecting feedback followed by an analysis of the feedback thoroughly would have taken a lot of time. Unfortunately, this was not possible during the period of this study.

The number of add-ins might also generate other problems. Having four add-ins with very specific functionality that is not very complex may only reveal shallow problems. It would have been very interesting to see the difference of the results if only one big add-in was created. The amount of time needed for server configuration would probably go down while the amount of syntactical research increased. The implementation problems would also be more specific to SharePoint development instead of SharePoint configuration.
6.3 Future work
The results of this study show how time is spent when developing add-ins but only based on the authors experience. Using the same method but instead with experienced SharePoint developers might show a completely different distribution of time. This would reveal a more complete picture of how a developer would spend his time.

Add-ins are reusable. It could also be beneficial to track how often features provided by a specific add-ins are used in order to get an insight into the usefulness of each add-in compared to another. This would also contribute to a general picture of add-ins where all types of add-ins are taken into consideration.

In the future one might also finish the related documents add-in using a provider-hosted hosting alternative instead of the SharePoint-hosted. The add-in will most likely be difficult to accomplish but perhaps not impossible. The time it would take to develop it can most likely be spent better.
7. Conclusion

The conclusion can be divided into two segments, one part that states the answers to the research questions. The other part summarizes to what extent the implementation was completed in accordance to the required competence. The second part can be concretized rather simply into a bulleted list.

- Checked-out documents – completed to full extent.
- My drafts – completed to full extent.
- Related documents – not completed due to lack of competence in the tools given.
- Show popular – finished project but not tested.

*What implementation details presented themselves repeatedly during development of add-ins for SharePoint?*

This question seems hard to answer but it can be very clear and specific given the information presented in this paper. Besides the fact that there is two hosting-options which vastly differs in possibilities, there are options within SharePoint-hosted context which can affect the possibilities considerably. Add-ins that deal with documents need information of said documents. There are slightly different ways to fetch the metadata for a document. The first option is to use a simple database fetch presented in checked out documents (5.1) and my drafts (5.2). The other option is to make a search query as presented in most popular. This is the most significant conclusion that can be reached about share-point hosted add-ins which are the main focus in this study.

*Which parts of add-in development is most time consuming?*

The most time-consuming parts of the development were tasks associated with the Model. Most of the work revolved around researching syntax for rest in the context of SharePoint and also setting the server environment correctly. The lack of documentation was the main reason for that these parts weighed so heavily in the time aspects. This does not only result in a lot of work but also decrease the reliability of the implementations since there may be situations where the add-in fails since there may be a missing setting etc. The actual development process however is not at all time consuming given the developer is already familiar with most implementation details.
8. Bibliography


Appendix A

The image above is an example of an add-in looks by default in its own app-web which is generated when created using Visual Studio.

This image shows the same add-in but now instead in the form of an app-part which makes it possible to include it on a website using a single HTML-tag.
The image above is the template for the logs.