# Semester Project
# *Slice of Pie*

**Group 8**
**Jacob Stenum Czepluch (jstc@itu.dk),**
**Michael Mungkol Storgaard (mmun@itu.dk),**
**Niclas Tollstorff (nben@itu.dk),**
**Niels Roesen Abildgaard (nroe@itu.dk),**
**Sigurt Bladt Dinesen (sidi@itu.dk)**

**Semester Project**
**Bachelor in Software Development**
**IT-University of Copenhagen**
**Jacob Bardram (bardram@itu.dk)**
**Dario Pacino (dpacino@itu.dk)**
**December 17, 2012**

# Contents


**1 Introduction** **3**

**2 User manual** **4**
- 2.0.1 Projects . . . 4
- 2.0.2 Folders . . . 4
- 2.0.3 Documents . . . 4
- 2.1 Local client . . . 4
  - 2.1.1 Projects . . . 5
  - 2.1.2 Folders . . . 5
  - 2.1.3 Documents . . . 5
- 2.2 Web client . . . 6
  - 2.2.1 Projects . . . 7
  - 2.2.2 Folders . . . 8
  - 2.2.3 Documents . . . 8

**3 Software Analysis** **9**
- 3.1 Use Cases . . . 9
- 3.2 Domain Model . . . 9
- 3.3 System Sequence Diagrams . . . 10
- 3.4 FURPS+ . . . 10

**4 Software Design** **11**
- 4.1 Class Diagrams . . . 11
- 4.2 Interaction Diagrams . . . 11
- 4.3 Design Patterns . . . 11
  - 4.3.1 Asynchronous Calls . . . 11

**5 Software Architecture** **12**
- 5.1 Decisions . . . 12
- 5.2 Logical View . . . 12
- 5.3 Process View . . . 13
- 5.4 Deployment View . . . 14
- 5.5 Data View . . . 16
  - 5.5.1 Remote Database View . . . 16
  - 5.5.2 Filesystem View . . . 17
- 5.6 Implementation View . . . 17
- 5.7 Development View . . . 17
  - 5.7.1 Preliminary Requirements . . . 18
- 5.8 Use Case View . . . 18

**6 SCRUM** **19**
6.1 Distribution of SCRUM roles . . . . . 19
6.2 The Product Backlog . . . . . 20
6.2.1 Definition of Done . . . . . 20
6.3 The Sprint cycle . . . . . 20
6.3.1 Sprint Planning Meetings . . . . . 20
6.3.2 Daily Scrums . . . . . 21

**7 Testing** **22**
7.1 Test Strategy . . . . . 22
7.1.1 Unit tests . . . . . 22
7.1.2 System tests . . . . . 22
7.2 Test Results . . . . . 22

**8 Discussion** **23**
8.1 Security Concerns Regarding the use of Remotely Accessible MySQL Databases . . . . . 23

**9 Reflection** **24**

**10 Conclusion** **25**

**A Supplementary Requirements** **27**
A.1 Functionality . . . . . 27
A.2 Usability . . . . . 27
A.3 Reliability . . . . . 27
A.4 Performance . . . . . 27
A.5 Supportability . . . . . 27
A.6 Security . . . . . 28
A.7 Implementation Constraints . . . . . 28

# 1 Introduction

This report covers our semester project for the third semester at the BSc in Software Development at the IT-University of Copenhagen. This project took place in November and December in the year 2012.

To make this project, we were asked to form groups of the size of five persons.

The main purpose of this project is to make an interactive document sharing system, called Slice of Pie, which merges some of the functionality of Google Drive with the synchronizing functionality of Dropbox.

Like Google Drive, Slice of Pie allows the user to create, edit, and share documents through a web interface. Documents can be arranged into projects and folders. The modifications of shared documents are merged together.

Like Dropbox, Slice of Pie, gives the user the possibility to synchronize the entire Slice of Pie library into a local folder. A stand-alone client application is provided which can perform the same operations as the web client. The difference is that now the user can also work offline.

Unlike Dropbox, Google Drive and the likes, our application is limited to simple text files and the client is built into a text editor.

Our advisors throughout this project were been Jacob Bardram and Dario Pacino, We have also received feedback from our teachers assistant, Simon Bang Terkildsen, who has been of great help to us.

The source code for the project may be found at
http://github.com/hypesystem/SliceOfPie/app.

# 2 User manual

### 2.0.1 Projects

Projects are the primary container entity in the program and can contain folders or documents, but not other projects.

### 2.0.2 Folders

Folders are a container entity in the program, which can contain folders or documents. Folders has to be created inside either a project or another folder.

### 2.0.3 Documents

Documents are the primary entity in the program, as it holds all textual content.

## 2.1 Local client

The local client is mostly working offline, but some features requires an internet connection established like share project and synchronize. Changes are only applied globally, when the data is synchronized.

You are not logged in, when you are using the local client, but it does require authentication, whenever you wish to interact with online data.

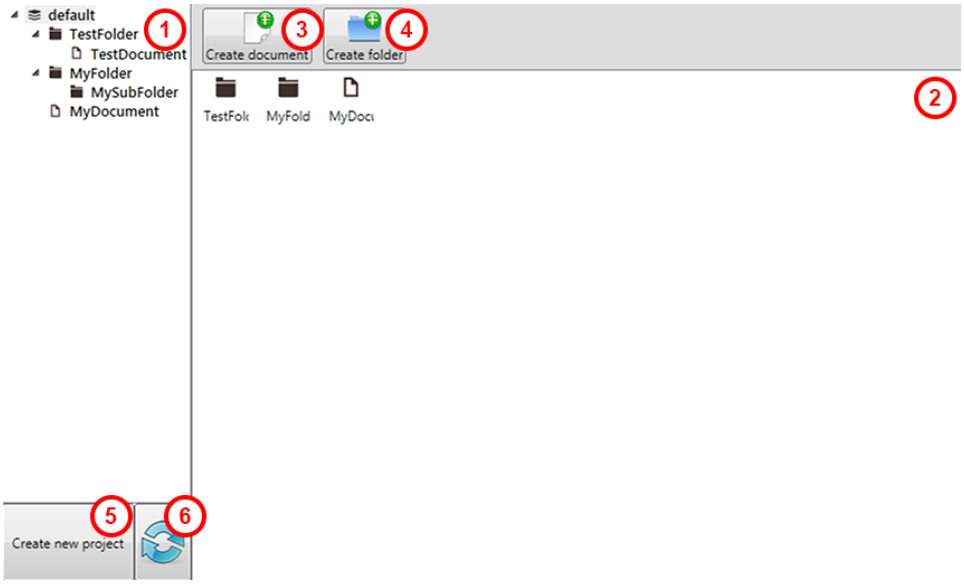


Figure 1: Screenshot of the local client

### 2.1.1 Projects

**Create project** To create a new project, click the Create new project button (5 in figure 1) and enter the desired name in the popup window. You can now find your new project in the explorer (1 in figure 1).

**Share project** In order to share a project in the local client, you need to have an internet connection established and your data has to be synchronized at some point before, meaning you do not have to have an up-to-date project, folder or document, but the project has to exist online. When these requirements are fulfilled, you can right-click a project in the explorer (1 in figure 1) and click Share project. This will open a popup window with a text field, which can handle both single email addresses or comma-separated email addresses.

**Remove project** To remove a project, right-click the project in the explorer (1 in figure 1) and click Remove project. Be aware that all folders, subfolders and documents will also be removed.

**Synchronize** In order to synchronize your data, you need to have an internet connection established, when that requirement is fulfilled, you can synchronize your data, by clicking the synchronize button (6 in figure 1) and entering your account information in the popup window. If the user does not exist, a new one will be created with the desired account information. All data will then be synchronized, which can take several moments.

### 2.1.2 Folders

**Create folder** To create a new folder, click the Create folder button (4 in figure 1) or right-click a project or folder in the explorer (1 in figure 1) and click Create folder. Then enter the desired name in the popup window. You can now find your new folder in the explorer as a subitem under the project or folder under which it was created.

**Remove folder** To remove a folder, right-click the folder in the explorer (1 in figure 1) and click Remove folder. Be aware that all subfolders and document will also be removed.

### 2.1.3 Documents

**Create document** To create a new document, click the Create document button (3 in figure 1) or right-click a project or folder in the explorer (1 in figure 1) and click Create document. Then enter the desired name in the popup window. You can now find your new document in the explorer as a subitem under the project or folder under which it was created.

**Edit document** To edit a document, click on the document in the explorer (1 in figure 1). This will open the document with a large editing area (1 in figure 2). When editing the the document, you can use the *HyperText Markup Language*[4] (HTML) to format your content. To save the document, simply click the Save document button (2 in figure 2).

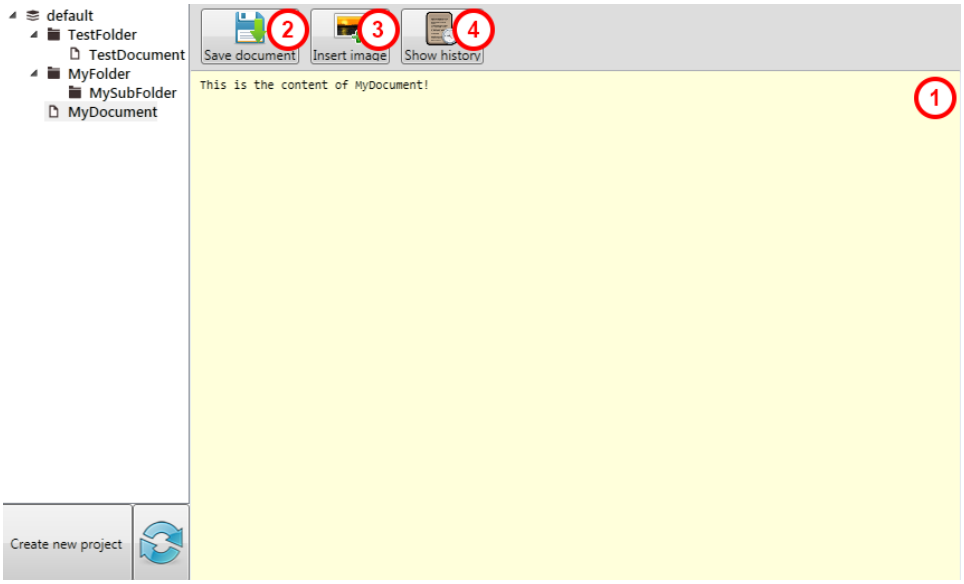


Figure 2: Screenshot of the document view in the local client

**Insert image** To insert an image, click the Insert image button (3 in figure 2) and enter the URL to the image in the popup window. This will insert an HTML image tag at the cursors position.

**Show history** To view the revisions of a document, click the Show history button (4 in figure 2). This will open a popup window with most recent revision visible, while rest of the history is available from the list in the left side of the window. You can copy the visible revision or part of it and paste it to your editor window.

**Remove document** To remove a document, right-click the folder in the explorer (1 in figure 1) and click Remove document. Be aware that all revisions will also be removed.

## 2.2 Web client

The web client requires a persistent internet connection and is making all changes in real-time. You are required to log in before using the web client.

You will however create a new user, if you enter an email address not already known.

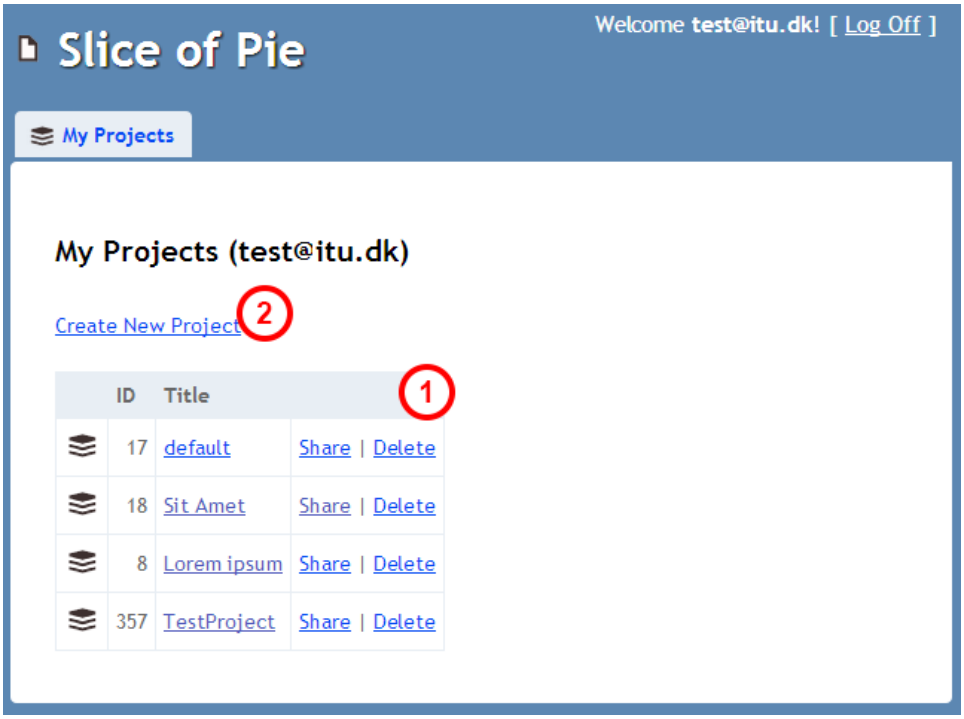


Figure 3: Screenshot of the web client

### 2.2.1 Projects

**Create project** To create a new project, click the Create New Project button (2 in figure 3) and enter the desired name on the new page. Your new project is now open and you can access an overview of folders and documents (like seen at 1 in figure 3).

**Share project** To share a project, click the Share button in the overview (1 in figure 3). This will lead you to a new page with a text area, which can handle both single email addresses or comma-separated email addresses.

**Remove project** To remove a project, click the Delete button in the overview (1 in figure 3) and confirm. Be aware that all folders, subfolders and documents will also be removed.

### 2.2.2 Folders

**Create folder** In order to create a new folder, you have to be inside a project or another folder, from where you can click the Create New Folder button (like seen at 2 in figure 3). Then enter the desired name on the new page. Your new folder is now open and you can access a list of folders and documents (like seen at 1 in figure 3).

**Remove folder** To remove a folder, click the Delete button in the overview (1 in figure 3) and confirm. Be aware that all subfolders and documents will also be removed.

### 2.2.3 Documents

**View document** To view a document, click the Show button in the overview (same location as the Share button in 1 in figure 3). This will open a new page with the contents of the document formatted as HTML (1 in figure 4).

**Show history** To view the revisions of a document, click the Display revisions button (3 in figure 4) in the view of the document. This will show the revisions below the view of the document in a list with latest revision at the top.

**Create document** In order to create a new document, you have to be inside a project or another folder, from where you can click the Create New Document button (like seen at 2 in figure 3). Then enter the desired name on the new page. Your new document is now open in editing mode.

**Edit document** To edit a document, either click on the document name in the overview (1 in figure 3) or click the Edit button (2 in figure 4) in the view of the document. When editing the the document, you can use the *HyperText markup language*[4] (HTML) to format your content. To save the document, click the Edit button below the text box.

**Remove document** To remove a document, click the Delete button in the overview (1 in figure 3) and confirm. Be aware that all revisions will also be removed.

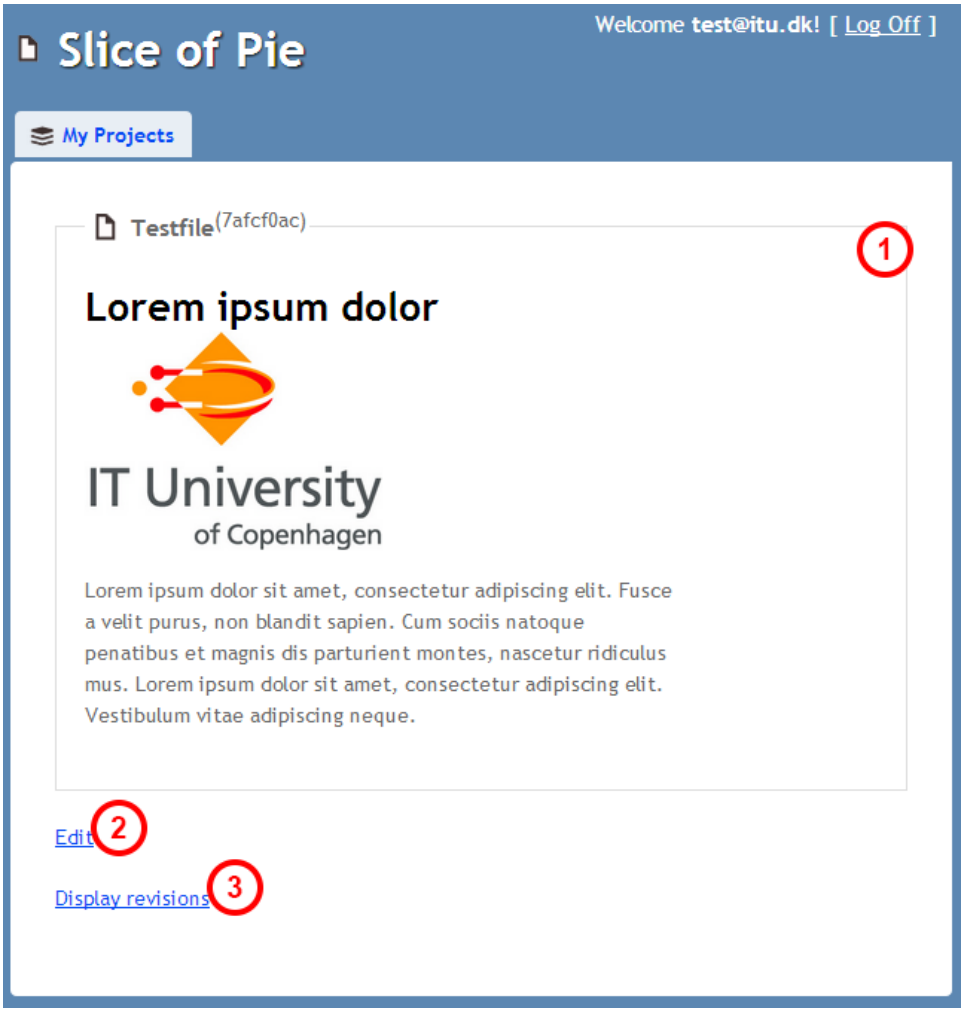


Figure 4: Screenshot of the document view in the web client

# 3 Software Analysis

This is the part where the analysis part of the project is described.

## 3.1 Use Cases

This is where the Use Cases are going to be.

## 3.2 Domain Model

This is for the Domain Model.

### 3.3 System Sequence Diagrams

This is for Software Sequence Diagrams.

### 3.4 FURPS+

This is FURPS+

# 4 Software Design

This is the main file for Software Design.

## 4.1 Class Diagrams

This is where the Class Diagrams will be described.

## 4.2 Interaction Diagrams

This is where the Interaction Diagrams will be described.

## 4.3 Design Patterns

This is where the use of Design Patterns will be described. -Singleton -Factory -GRASP -MVC -Layered

### 4.3.1 Asynchronous Calls

For a desktop UI to be repsonsive, it is important that heavy loads are moved away from the UI thread. There are several possible ways of doing this: threading, abstracting to using tasks, or asynchronous methods. To reduce the amount of boilerplate code in the UI layer, the responsibility for delegating to threads was moved to the controller.

The *Asynchronous Programming Model*[1] (abbrevated *APM*) allows for easy, asynchronous calls to heavy-duty methods at a nice level of abstraction, substrantially simplifying UI programming. The support for callbacks lets the UI react as soon as a job finishes, without having to continuously poll the model.

There do not seem to be any obvious building blocks for implementing the APM, however the MSDN Magazine[2] has an article by Jeffrey Richter guiding the process. Based on these ideas, building custom APM comes easier and this is the way we chose to go with the controller. Instead of just using Richter's classes and implementations, we built an automated factory, which quickly wraps a blokcing method in APM, making it non-blocking.

Another advantage with the APM pattern is that it is nicely supported by the *Task Parallel Library*[3, p. 656] (TPL), allowing for them to be used in a simple interface with great concurrency and error-handling support.

# 5 Software Architecture

The following is akin to a Software Architecture Document (SAD) [5, p. 655-659] and as such it contains different perspectives (views) on the most important parts of the overall architecture of the SliceOfPie system.

There are additional requirements for the application that are taken into account during the development of the system. An overview of these may be found in Appendix A.

## 5.1 Decisions

In SliceOfPie, there is a deeply founded MVC structure, in that the model and controller are compiled to a library that can be used by any user interface that wishes to do so, though a simple API (in the Controller).

There has been an emphasis on simplicity, and prooving that the concept works. This means that not all aspects of the application are currently at a production-grade level, but the infrastructure is there to support changes that would bring them to that required level.

Several other decisions and the reasoning behind them are detailed in section 8.

## 5.2 Logical View

The architecture is based around a reusable application logic layer, that connects to the local file system and a MySQL database as needed. Additionally, it can easily be used by both the Web-Only UI and the Local UI.

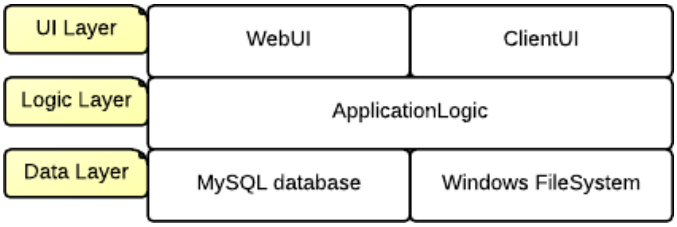


Figure 5: Layers of the application

Because of this structure, we have an architecture that is split in that it supports two parallel UI layers and two parallel data layers (see Figure 5).

From these layers our package diagram almost directly follows (Figure 6): To communicate with the different data layers, we have two separate models. It is important to note that the *LocalFileModel* connects to both the file system (in most situations) and to the MySQL database (when this is explicitly requested).

In using the Model-View-Controller pattern to separate to visuals (WebUI, ClientUI) from the data behind (LocalFileModel, WebFileModel) by

communicating through the Controller, we achieve low coupling and high cohesion. We focus on extensibility and easy replacement of different parts without it affecting other parts of the software. This is useful in a proof-of-concept as more sturdy, production-level classes should be put in place before the system is distributed.

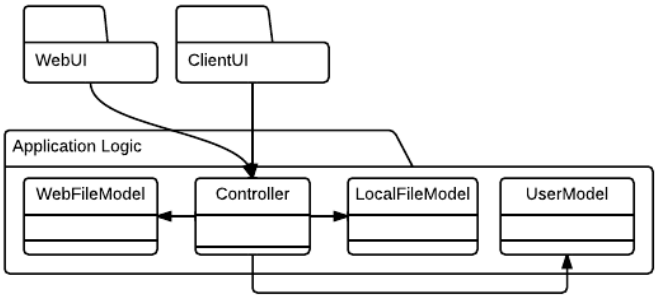


Figure 6: Package Diagram showing dependencies from the UIs to the Application Logic

The Controller additionally depends on a UserModel being present. This model handles verification and creation of user accounts.

With the overall architecture in place, the last important logical component of the architecural design consists of the basic classes that all packages use (Figure 7): *Document*, *Folder* and *Project*. These are mostly defined in their implementations of *IItemContainer* and *IItem*. *IItem* defines an "item"; something that can be contained in an *IItemContainer*. This is important as Folders may both *contain* and *be contained*, whereas Documents may only be contained and Projects may only contain

This structure, with its use of *composition* (in that folders may contain folders), results in a dynamically extensible hierachy, much like the one found in most filesystems. An additional constraint in the architecture is that a root element will **always** be a project, and as such it is easy to manage bundles of files by their parent project.

### 5.3 Process View

In Slice of Pie, concurrency is handled in the controller.

To handle concurrent calls to the controller and to keep user interfaces from blocking when methods are being called, we have implemented a pattern called the Asynchronous Programming Model (APM; see Section 4.3.1).

APM allows for asynchronous calls by allowing a callback to fire when a call finishes. This means that a client can change from using a method to using its counterparts prefixed with *Begin* and *End* to opt in for an asynchronous call. This is particularly useful in desktop clients, which would

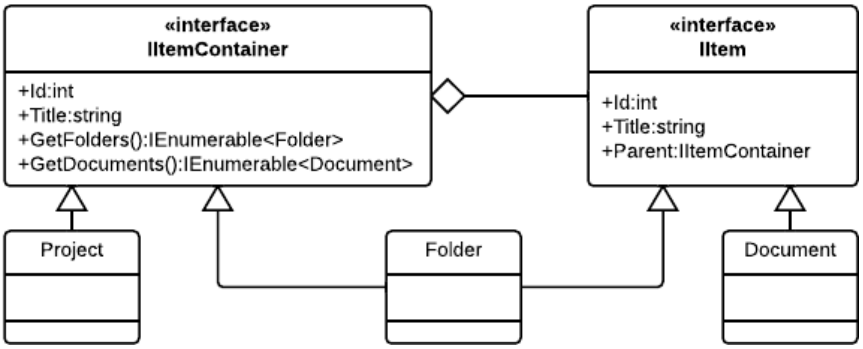


Figure 7: Basic classes sent around in the application and their relative relationship.

usually lock up (become irresponsive) while waiting for the longer blocking calls to conclude.

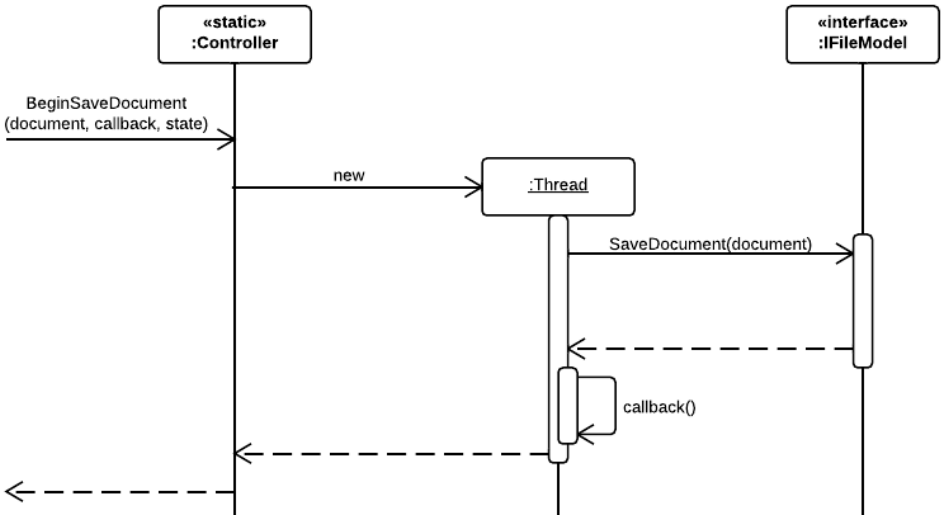


Figure 8: Sequence Diagram showing the inner workings of any APM call. Instead of using a new Thread, our implementation delegates calls to the Thread Pool, to optimize performance (reducing overhead of creating new threads).

## 5.4 Deployment View

SliceOfPie has three different parts deployed: The database server, the web server and the local desktop client.

**The database server** has a Linux-based OS and hosts a remotely-accessible MySQL database that is the central point of the application's

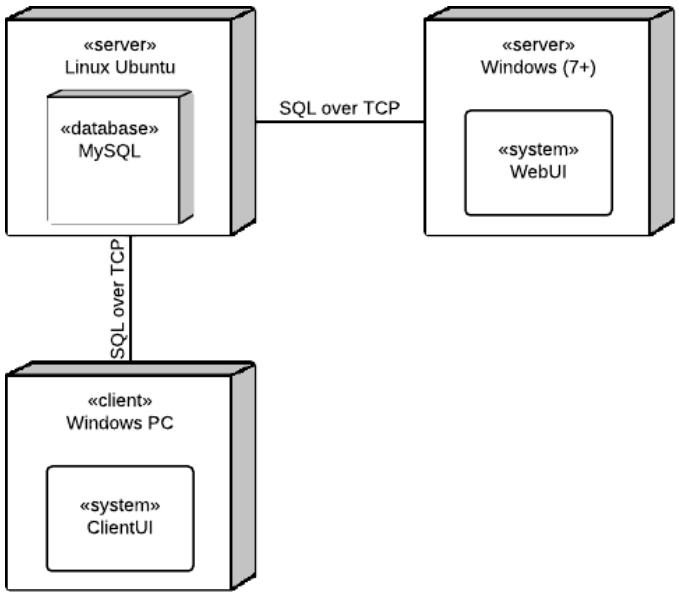


Figure 9: Deployment Diagram of the application, showing its distribution.

network, holding the current online state. Having a single database server ensures a consistent state across the platforms accessing the service. It is, however, also a single point of failure. In a production environment this could be mitigated by setting up a second server as a backup.

There are some security concerns with regards to having a remotely-accessible MySQL database that are discussed in 8.1.

**The web-server** is a Windows server hosting a WebUI client. This client is accessible from all locations that have access to the host. This means taht a Web server may be set up anywhere to provide a public interface for a great amount of users. Having the data separated from the web interface allows for several such hosts to be running with the same consistent state. As handling requests, user-sessions and HTML transfers are often a bottleneck, this can be an advantage if the system receives a great amount of traffic, simply sending users to different servers.

**The desktop client** is a single client, which accesses the data-server directly. It has the advantage of allowing users to work offline, without a connection to the internet, and synchronize with the global state of the database server when they regain internet connection. Without synchronization, it also works as a simple text-editor. It has the disadvantage of not being guaranteed up-to-date (like the Web UI is).

## 5.5 Data View

There are two perspectives on data in the system: One is the interaction with the MySQL remote database and the other is the interaction with the local filesystem used in *LocalFileModel*.

### 5.5.1 Remote Database View

The mapping between the database and the logical classes is found in *MySQLModel.edmx* in the *SliceOfPie* project.

Most of the mapping is as straight forward as it gets: Properties map to fields of the corresponding value. However, in the relationship of the entities there are some important details.

The first point that is important to make is that users are defined in the data as a tuples of emails and passwords and that between users and projects there is a many-to-many relation. This manifests in the ProjectUser table, holding references to both. In effect this means that several users may have access to the same projects, and that one owner may have access to several projects.


Figure 10: The many-to-many relationship between users and project.

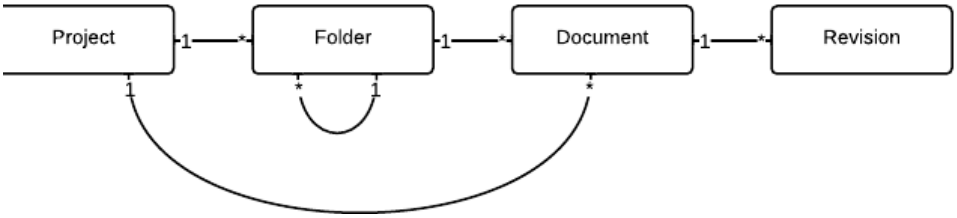


Figure 11: The one-to-many relationships between projects, folders, documents and revisions.

The Folder table contains a reference to a Folder OR a reference to a Project. This is important as it may live in either of these. It should never have both, but this is not a constraint inherent to the database. This manifests in a one-to-many relationship, where a parent project or folder may have many child folders.

Documents have the same references a Folders, and results in the same kind of relationship.

Revisions have references to Documents, resulting in a one-to-many relationship where a single document may have several revisions tied to it.

### 5.5.2 Filesystem View

The logical classes used in the system (see Section 5.2) map pretty much precisely to a traditional filesystem hierachy. In the filesystem they are represented as follows: Projects are represented as folders, found in the root of the data folder (in *<userdir>/AppData/Roaming/SliceOfPie*); Folders are folders inside the projects; Documents are files with a *.txt* ending, inside either folders or projects.

## 5.6 Implementation View

The source code of the project can be found on GitHub at http://github.com/hypesystem/SliceOfPie/app .

The solution is built into serveral different files, reusing code as best possible.

**SliceOfPie.dll** This library contains the general SliceOfPie-centric code. It is built into a DLL to be easily distributable between all interested clients. Apart from defining the common language used in the UIs (Documents, Folders, Projects), it also defines the API for the Controller, which, in effect, gives access to the data of the application.

**SliceOfPieClient.exe** This executable is simply the desktop UI. It comes bundled with SliceOfPie.dll, and provides an easy interface to use the application through.

**MvcWebApp.dll** This DLL is a MVC project, that can be hosted on so-enabled Windows servers. It depends on an MVC3 installation being present and comes bundled with SliceOfPie.dll.

**SliceOfPieTests.dll** This DLL is never actually released and contains tests for SliceOfPie.dll.

## 5.7 Development View

The software has been developed with a version control system (VCS) called Git[7], originally developed by Linus Thorvalds.

This version control system emphasises agility and the ability to concurrently work on serveral things at once, branching the source code by features, and joining the branches again only when features are done[8].

Our source code is hosted at GitHub, a Git repository host.

Our report is written in LaTeX[9] and is part of the project as its content has been developed side-by-side with the application.

### 5.7.1 Preliminary Requirements

It is recommended to use the Visual Studio 2010 IDE and the Git VCS.

Before compiling the source code of the project there are a few requirements for the setup:

1. MySQL Connector/Net[1]; allowing the application to talk with the MySQL server.
2. ASP.NET MVC 3 Tools Update[2]; enabling the WebUI to run.

## 5.8 Use Case View

[1] Availible from MySQL: http://dev.mysql.com/downloads/connector/net/

[2] Availible from Microsoft: http://www.microsoft.com/en-us/download/details.aspx?id=1491

# 6 SCRUM

In this project we have been using the Scrum development framework in order to develop our software solution in an iterative and incremental way. Scrum can be very briefly summarized as follows:

A cross-functional **Team** works in timeboxed periods (called **Sprints**) throughout the development. At the beginning of these Sprints the Team chooses prioritized **items** from an exisiting **Product Backlog** that should be implemented. These items will be added to a **Sprint Backlog** and should result in a **Potentially Shippable Product Increment** by the end of the sprint. After the Sprint is initiated from two short **Sprint Planning Meetings** progress is tracked with **Daily Scrum**-meetings together with **Burndown Charts** throughout the Sprints. An important point is that Sprints are never extended. If the work on an item is not considered complete according to the Teams **Definition of Done** by the end of the sprint, it is placed back in the backlog. **Review-** and **Retrospective Meetings** at the end of each sprint makes the Team learn from any mistakes and problems, and as such Scrum is based on feedback-cycles. This section elaborates on our use of Scrum throughout the process.

## 6.1 Distribution of SCRUM roles

Using Scrum involves forming a cross-functional team consisting of three major roles: Product Owner, ScrumMaster, and Team (also called Development Team).

The **Product Owner** is interested in maximizing the return on investment (ROI) for the product. This is done by translating product features into a prioritized list, which should be continually refined. As such the Product Owner is responsible for the *value* of the work being done. In our Scrum Team the role of Product Owner was assigned to **Niels Roesen Abildgaard**.

The **ScrumMaster**'s role is to help the Scrum Team learn and apply Scrum in order to increase the business value. It is important to note that the ScrumMaster is not the manager of the Team members, but instead functions as more of a coach to the Team regarding the correct use of Scrum. As such the ScrumMaster makes sure that the team understands and follows the Scrum principles. In our Scrum Team the role of ScrumMaster was assigned to **Niclas Benjamin Tollstorff**.

The **Team** builds the product in a cross-functional manner. This means that there is no specialist role. Every team member should be able to work on any part of the system. Due to our limited team size, we had our Product Owner and ScrumMaster work as a team member outside of meetings.

## 6.2 The Product Backlog

As mentioned in the above section the Product Owner prioritizes items in a list. This list is called the **Product Backlog**. These items will primarily be user-centric features, but can also be technical improvement goals or bugs to fix.

Since the Product Backlog exists and evolve throughout the lifetime of the product we found it useful to use an online tool for creating and maintaining it. Our choice fell on **Pivotal Tracker**, which is especially suited for agile project management and collaboration. We have strived to make our product backlog DEEP (**D**etailed appropiately, **E**stimated, **E**mergent, **P**rioritized) as is good practice in Scrum. The use of Pivotal Tracker facilitates the DEEP approach by allowing the assignment of descriptions (*detailed appropiately* attribute) and estimation points (*estimated* attribute), as well as supporting ordering (*prioritized* attribute ) and continous refinement (*emergent* attribute).

### 6.2.1 Definition of Done

In Scrum the output of a Sprint should be a **Potentially Shippable Product Increment**. As such it is essential to define this term before starting the first Sprint as to avoid any misunderstandings. The Product Owner, Scrum Master and Team discuss this definition and captures it in a **Definition of Done**. Our Definition of Done can be found in the appendix.

## 6.3 The Sprint cycle

In our (close to) one month project period we managed to fit in tree one-week Sprints as well as a short preparing *Sprint Zero*, as is sometimes used in Scrum. The Sprint Zero was focused on the discussion of our vision for the product, identifying initial items for the Product Backlog, providing a minimal environment that enables the writing of quality code, and other preparation we saw fit e.g. group constitution.

Sprint Zero aside, we made use of Scrum specific meetings and progress tracking tools in our Sprints. In this section we will elaborate on the work process with these tools. It should be noted that the meetings and tools mentioned in the following sections are repeated each Sprint.

### 6.3.1 Sprint Planning Meetings

Our Sprints were prepared by two short meetings of one hour each. The first of the Sprint Planning Meetings was focused on the "what", while the second is focused on the "how".

In **Sprint Planning Part One** we discussed our priorizations of items in our Product Backlog. This part of the meeting was heavily influenced by

the Product Owner as he was responsible for the Return of Investment. As such this part of the meeting was primarily focused on the Product Owner explaining his thoughts to the Team (and ScrumMaster) regarding what items should be chosen and also *why* that is the case.

In **Sprint Planning Part Two** we discussed what items from the Product Backlog should be implemented in the current Sprint. An important aspect of this is meeting was that while the Product Owner heavily influences the priority in the Product Backlog, it was the Team that ultimately chose how many of those items to take on in the Sprint (with highest priority items chosen first). This division of work makes the each Sprint more reliable, as the Team knows how much work they can handle.

At the end of the Sprint Planning Meetings we placed the chosen items from the Product Backlog into a **Sprint Backlog** in our Pivotal Tracker online tool. In a work environment Sprint Backlogs are sometimes maintained as stickers on a big board (called a **Scrum Board**), which provides an easy overview of the current status. This was not possible (or reliable at least) at our university as we can not reserve the white boards or rooms. Fortunately our online tool has several status states for each items, which we made heavy use of.

### 6.3.2 Daily Scrums

During the Sprints we had regular very short meetings (10-15 minutes) which was focused on the team members updating each other and coordinating work according to the current status on items in the Sprint Backlog. In a work environment this meeting should happen every workday, but due to other courses we had them scheduled for tuesday, thursday and either saturday or sunday. Our Daily Scrums were based on the three points (1) What has been accomplished since the last meeting?; (2) What will be done before the next meeting?; and (3) What obstacles are in the way?

Should any obstacles be present it is the ScrumMaster's job to help resolve them.

# 7 Testing

Testing in our system is primarily based on automated unit tests and manual structured client tests. The reasoning behind this choice of test methods of course with the intention of gaining confidence in our code and program in general. With this approach we achieve a thorough testing of the individual smaller parts of the system, while maintaining confidence that all the parts works in unison and as intended, as we systematically go through each functionality in the client program.

## 7.1 Test Strategy

We use unit tests for white box testing of individual methods. Each non-trivial method should have its own tests and those tests focus should only lie on that particular methods functionality.

The other part of our testing is black box testing performed through system tests. Whenever code have been added or changed, system testing of each affected functionality should be done.

### 7.1.1 Unit tests

All non-trivial logic code in our system has one or more unit tests, this ranges from a class constructor like the LocalFileModel to merging of two documents in the Merger. As these are white box tests each test are created explicitly from the code logic and are created with only one focus, like seen in the unit tests for UserModel, which very specifically tests whether an email and password combination validates or not.

Whenever applicable we have used a test-first or test driven development approach to ensure that an existing bug does in fact exist or that specific functionality is not yet implemented and therefore fails the test. This also makes it a lot easier to test that specific functionality instead of running the complete system to only create or alter a smaller part. When mentioning applicability it is because the use of test-first is particularly useful, when writing small or encapsulated methods or functionality like creating a document, removing a project or merging two documents, while not as useful, when for example writing UI.

Our unit tests are focused on the logical code, which include the models, the controller and their subclasses.

### 7.1.2 System tests

## 7.2 Test Results

This is the results from the testing.

# 8 Discussion

## 8.1 Security Concerns Regarding the use of Remotely Accessible MySQL Databases

SliceOfPie is based on a single data-server keeping the global state of the application (see 5.4).

In the proof-of-concept (current) implementation of the application, the data-server is nothing more than a Linux-server hosting a remotely accessible MySQL database. This means that any computer with access to the internet and knowledge of the remote server's IP can try to connect to the database. Any such knowledge can be used to brute-force (or otherwise crack open) the database. This poses a security threat.

With decompilation tools, it is possible to extract the exact IP (and in the current state also the username and password) used to connect to the database. This means that the global state, that the entire application relies on, is - in effect - not secure. Tampering of data is not prevented in any way.

In a production-environment (read: not-just-proof-of-concept) this would be unacceptable. An alternative exists, however: Web-Services.

A WS hosted on the data-server could provide a publicly availible API for getting and setting data in a more controlled manner, without leaving any database passwords hidden in the source code of the application. Providing such an API in a well-known format such as XML or JSON[6] would, additionally, allow for easy development of other types of clients for the service (browser-hosted Javascript clients; PHP clients; etc.)

Changing the back-end from a database to a web sevice would only require changes in the file models currently communicating with the database

# 9 Reflection

This is the main file for this section. Fill in what ever you want.

# 10 Conclusion

This is the main file for this section. Fill in what ever you want.

## References


[1] Microsoft, *Asynchronous Programming Model (APM)*
MSDN
http://msdn.microsoft.com/en-us/library/ms228963.aspx
Visited: 2012-12-13

[2] Jeffrey Richter, *Implementing the CLR Asynchronous Programming Model*
MSDN Magazine, March 2007
http://msdn.microsoft.com/en-us/magazine/cc163467.aspx
Visited: 2012-12-13

[3] Ian Griffiths, Matthew Adams & Jesse Liberty, *Programming C# 4.0*
O'Reilly, 6th edition, 2010.

[4] World Wide Web Consortium (W3C), *HTML: The Markup Language (an HTML language reference)*
http://www.w3.org/TR/html-markup/
Visited: 2012-12-13

[5] Craig Larman, *Applying UML and Patterns*
Prentice Hall PTR, 3rd edition, 2011.

[6] Alex Rodriguez, *RESTful Web Services: The basics*
http://www.ibm.com/developerworks/webservices/library/ws-restful/
Visited: 2012-12-13

[7] Scott Chacon, *Pro Git*
http://progit.org/
Visited: 2012-12-14

[8] Slice Of Pie (Michael Storgaard), *Branching During Development*
https://github.com/hypesystem/SliceOfPie/wiki/Branching-during-development
Visited: 2012-12-14

[9] Wikibooks, *LaTeX*
http://en.wikibooks.org/wiki/LaTeX
Visited: 2012-12-14

# A Supplementary Requirements

Following is a list of requirements for the system that cannot be defined as Use Cases but are still important to the project.

## A.1 Functionality

The system must keep track of user projects and project files, including revision control.

The system must supply a web-interface, as well as a local interface for offline operations.

The system must support merging of conflicting versions of files, such as those that would occur from simultaneous offline and online editing by different users.

## A.2 Usability

Document creation, sharing, and other technical background operations must be performed within a time span of a couple of seconds ( 2).

The only operations that are allowed to take more than a couple of seconds are user operations such as editing documents and reviewing merges.

Merge reviews should be easy to understand, such that a normal page of revisions can be revised in less than 2 minutes.

## A.3 Reliability

The system must have 98% up-time. Planned down-time not taken into account.

The system should be able to reject errors in synchronization, such that document synchronization is atomic and correct.

## A.4 Performance

The system is partly based on a web interface. Performance, therefore, is dependent on external factors such as the users internet connection. This makes bandwidth consumption a prime consideration for system performance.

The local client must be light-weight enough to run on a five year old laptop with a dual-core processor.

## A.5 Supportability

The system should be self contained enough to require technical assistance only during server setup.

The system should have high version stability. No update is allowed to break backwards-compatibility.

### A.6 Security

Users are authenticated via a personal login and password.

Passwords may not be kept as plaintext, but are maintained as salted hashes.

### A.7 Implementation Constraints

The system should be written in C#. This a requirement from a very important stakeholder.