Contents

1 Introduction

Through the course, Software Development in Large Teams with International Collaboration, students at the IT-University of Copenhagen (ITU) are tasked with creating a digital rental system in collaboration with students from the Singapore Management University (SMU).

The team of students from the ITU (the Danish team) consists of five people, whereas the Singaporean team consists of three.

The digital rental system that is to be created should consist of

- a server program published through an IIS web server, developed by the Danish team;
- a client program, using the server, developed by the Singaporean team; and
- a client program, using the server, developed by the Danish team.

This report concerns our experiences with developing such an application, as well as how transnational development affected the process. As these two aspects of the development are easily discussed separately, the report is split in two parts.

In the first part we discuss the process of designing the software as well as the final result. We touch on alternatives that could have been used and why we – in this particular case – designed it as we did. We touch upon the design of requirements, use cases, and APIs as well as the underlying data model.

In the second part we concentrate on the actual collaboration with the Singaporean team, difficulties faced in general Global Software Development, and how these manifested in our work. We explore different technologies for conducting virtual meetings, and chronicle our experiences with shared documentation.

2 Analysis and Design

2.1 Requirements

As part of the analysis, we need a set of requirements for our product to establish what users will and will not be able to do with the system. This section describes and discusses our choice of requirements and use cases.

There are two types of users: users and managers. A manager's rights are a superset of a regular user's rights. In addition, managers do not have to purchase anything.

Our use cases, which describe in more detail what users and managers must be able to do with the system, are shown in Appendix A.2.

2.1.1 Security and other Design Decisions

When deciding on the requirements for the system, several criteria are considered, and some requirements which might be crucial in a real-life product are left out. These decisions are mainly made on the basis that this is an educational project, making these requirements more or less irrelevant to the project:

- We decided to leave out user authentication in our web services, so given the URL, anyone can download a movie our song without authenticating. The data structure does, however, support user authentication with each user having a password.
- In a real-life renting service such as iTunes, once a user starts playing the movie that she has just rented, the movie will only be available for viewing in the next 48 hours. When a user rents and downloads a movie or song using our service, they can keep the file for as long as they want. However, the file is only available for download for a limited period of time.
- Naturally, we have not implemented a real payment service but rather a "fake" balance where people can deposit money to their account, for use when purchasing or renting movies and songs.

2.2 Application Architecture

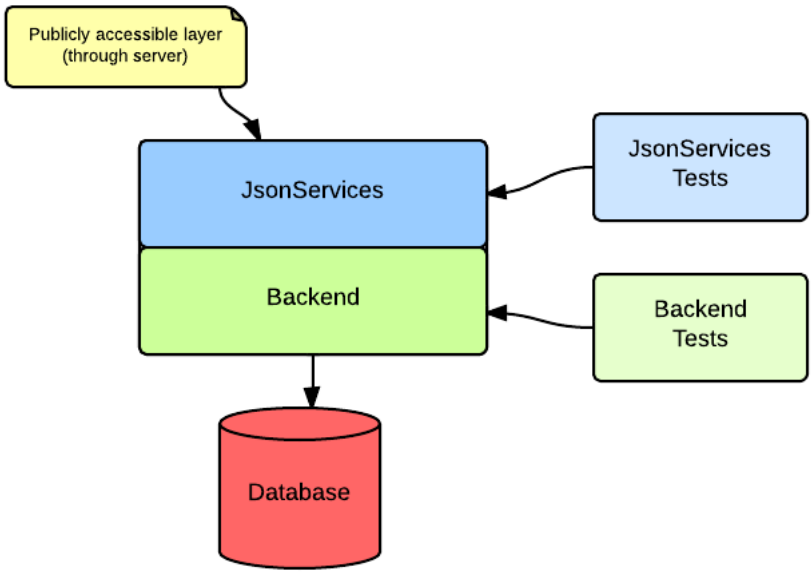

Figure 1: The different layers in the application, showing how tests interact with the different layers, and how JsonServices builds on Backend and is the accessible layer.

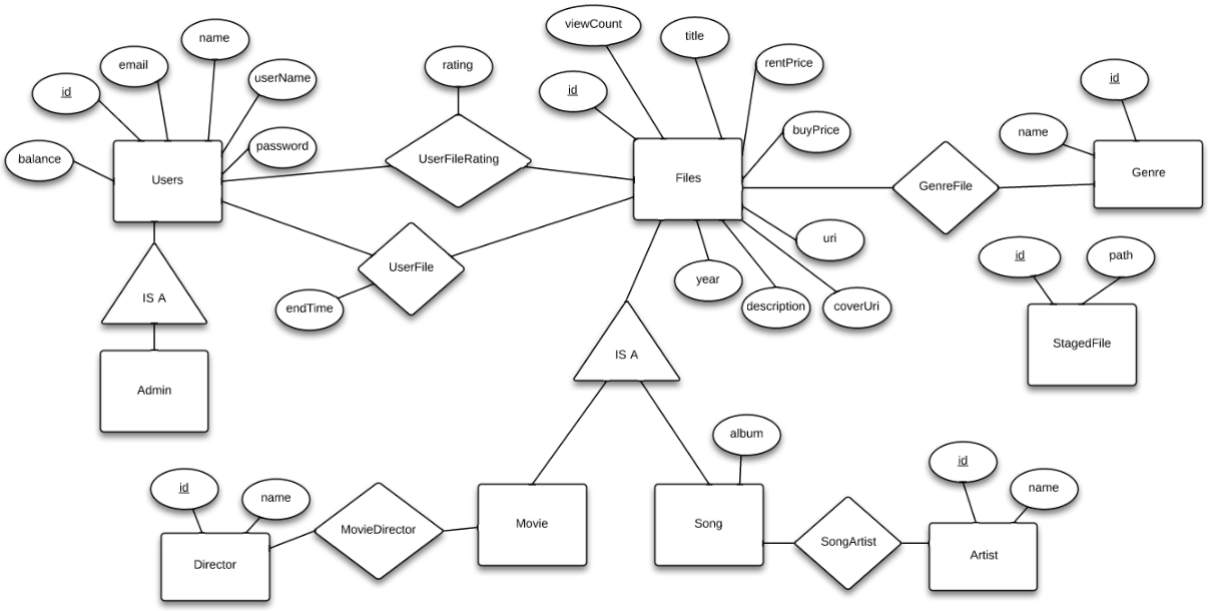

Figure 2: Our Entity Relationship Model

When designing the general architecture of the application, we decided on a simple, layered approach. This was done to separate any business logic from actual presentational code. In the case of an online API, the actual API is the presentational part (not to be confused with the client, which was developed separately from the server).

The `JsonService` project uses the logic availible through `Backend` library. Each of these has a testing library tied to it. The `Backend` utilizes a MSSQL database.

2.3 Data Model

2.3.1 Data Model and Back-end Structures

With the overall requirements set we began working on the data model of the system with an initial focus on the database entities and relationships. The database entity-relationship model was sketched in LucidChart [1] using the notation from Database Management Systems [5]. The complete E-R model is shown in Figure 2.

Our E-R model follows a naming scheme, where each table is named after the real domain object it represents (singularized), with junction tables named by a concatenation of the tables linked. The naming scheme of junction tables are shown in Figure 3.

[1] Lucidchart is a web-based diagramming software which allows users to collaborate and work together in real time to create flowcharts, organisational charts, website wireframes, UML designs, mind maps, software prototypes, and many other diagram types.

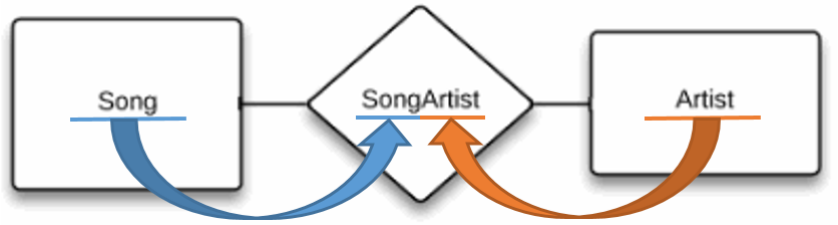

Figure 3: Example of concatenated naming for junction tables.

Where several junction tables exist between the same entities, we append an additional description string to the name (as seen in "UserFile*Rating*"). Our decision to use the domain objects' names singularized caused problems with the "User" and "File" tables, since these names are reserved keywords in MS SQL. We updated said tables by appending a 's' to the name (making it plural), but we kept the names singularized in the junction tables.

Besides creating a database structure capable of holding all the data needed for our use cases, we wanted to avoid redundant data. Our choice to avoid redundant data stems from the wish to eliminate anomalies within the database of which redundant data is a key cause. Anomalies are known to weaken the integrity of the database [5] due to irregular or inconsistent data, and we believe integrity is a key characteristic of a database dealing with users, their money, and their property. We enforced this constraint by applying a database normalization (third normal form) to our design, which is known to be free of update, insertion, and deletion anomalies.

2.3.2 Designing for Extendability

During the design of our entity relationship model, we took time to brainstorm how the design might evolve during the lifespan of the product and how the datamodel might accommodate to those changes. We settled on two plausible cases

- A need for several types of medias
- A need for several types of user roles

In the following we will explain how we designed for extendability in the context of the first case, but the principle applies to them both. The naïve solution would be to create a new table for each media type e.g. **Song (id, title, artist, album, coverURI, rentPrice, ...)** or **Movie (id, title, director, coverURI, rentPrice, ...)** This will however create a lot of duplicate columns in different tables. If we wanted to add new information to our media types later, e.g. a "viewCount" (which did actually happen) this would need to be added to every media table. The realization that several columns of information is shared across the media suggest for an inheritance based approach,

where the shared columns are placed in a “super table”, while every media table will take part in an IS-A relationship with said “super table”. This is the design that can be seen in Figure 2.

With our minds set on centralizing the common media attributes in the Files super table, we noted that both a movie director and a song artist could be gathered under a common label "Creator". The Files table would then be connected to a "person" table, which would store both song artists and movie directors.

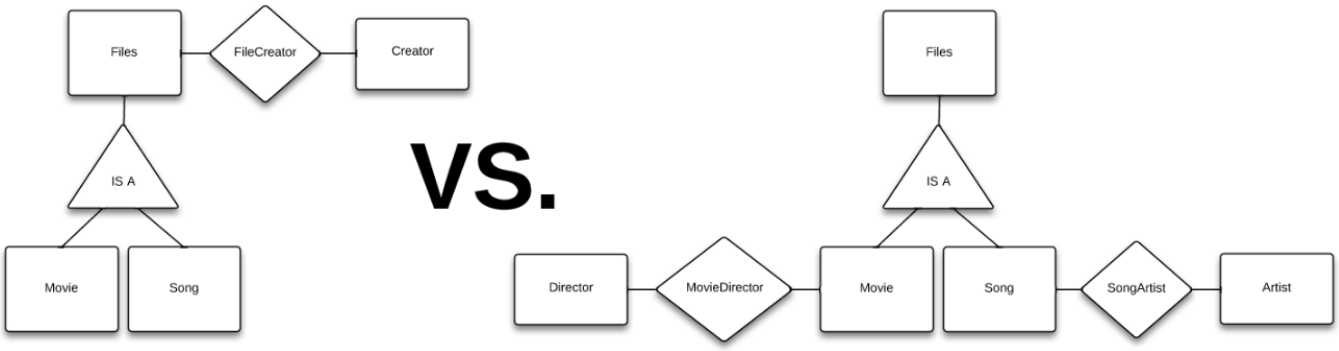

Figure 4: Example of the dilemma cases

We discussed this design and noted that it could not maintain the current constraint that only songs are linked to artists and only movies to directors. In addition we would not be able to extend the information on artists without extending it for directors too. This led to the decision to keep them seperate, although they are conceptually very close.

Later in the project we became aware that the above dilemma also applies to the concept of genres with the implication that a song could be linked to the genre "Horror", which does not make sense. As such the design of the genres-relation does not match our established design philosophies and we believe that this could (and should) be improved upon in any later revision of the project.

2.3.3 Accessing the Database

After having designed the database structure we had to decide how to access and utilize the database in the best suited way for our webservice. We quickly settled for a C#/WCF-based web service, since every group member had about the same experience in that platform from other ITU courses. With access to the .NET framework, we considered to make use of the Entity Framework for our database interactions and general object-relational mapping. However several group members had been experiencing some inconvenient performance issues in previous projects. We do believe performance is an important aspect of our service and research has shown that users of the internet have become very impatient with regards to response times [6]. We

also deemed our queries and database interaction to be fairly simple, thus having no real need for most of the functionality in the Entity Framework. After a discussion it became clear that the only real benefit of using the framework was the possibility of faster prototyping. With our emphasis on a solid performance we choose to spend a little more time developing our own database access layer in order to maintain a more direct control over the performance of our solution.

2.3.4 From Entities to Classes

Nævn ét eksempel på en entity -> objekt mapning. Husk - kun backend.

2.3.5 Lazy loading

Nævn det. Giv et eksempel.

2.3.6 Supported workflows

Nævn at alt er understøttet. Giv ét eksempel på en use cases/handling med henblik på interaktion og hvor simpel API'en (backend API) er.

2.4 Deciding on the API format

The two teams quickly agreed on using a RESTful JSON-based API as interface between frontend (client) and backend (server). This, however, still left a lot to design.

When designing an API it is important for it to be intuitive and attractive, as this allows third-party developers, as well as our overseas colleagues, to use the API for their purposes. Third-party developers in general represent a huge business opportunity in our case, as the system requires purchases in order to actually access movies. This means that third-party clients may create new revenue sources for us.

When discussing how to best attack the problem of what a "nice API" is, we found two alternatives.

Either we could let the API handle arbitrary datatypes or have specific services for every allowed datatype. The difference between the two possibilities can be illustrated through which services are offered by them:

In the high abstraction case a call to the files service may return any of the supported data types, so the client can not know this for sure at the time of calling the service. This problem can be solved in the following ways: first off, the client discovers content through a service that will also disclose the type of the content (so it is already aware

Abstraction level	High abstraction	Low abstraction
Services provided	files/	movies/ songs/

Table 1: Services offered by different abstraction levels

of the type) and, secondly, the files-service itself will return the type of the content as well as the content itself.

In contrast, the movies- and songs-services always return movies and songs, respectively.

This difference can be illustrated by looking at the services as if they were method calls. In this case, the files-service would have a return-type of "Object", whereas the movies- and songs-services would have "Movie" and "Song".

The high abstraction level works like a dynamically typed language (PHP, Javascript) would work, wheras the low abstraction level works like a type-strong language like C# or Java.

2.4.1 Cost of Change

In the high abstraction level service it is easy to expand the range of datatypes we offer, as they will be using the same API. This can with great advantage be paired with a discovery service that lets clients discover the supported data-types in real-time. The cost of change is pretty much non-existent.

This has ***a lot*** of coolness factor.

This approach is very dynamic and allows for updates to be rolled out seamlessly. It is, however, best suited for a tuple-space based database and not a relational one, such as the one we have chosen to work with.

In a relational database, using the high abstraction would require either massive scans of one huge table containing all files in the system or having multiple indexes for the same, huge table, which would be quicker, but also take up double the space.

The low abstraction level is very well suited for the relational database. Creating a table for each of the services (one for movies, one for songs) is a natural choice and lets us have the desired attributes for the types. Additionally searching is easier and fewer indexes are needed per table (so it takes up less space).

On the downside, the low abstraction level requires for entire new tables and services to be added whenever we want to support a new datatype. In turn, adding new services to the API does not make them accessible from the clients. In order to fully roll out an update, **all clients**, including third-party ones, must be updated. This all adds up, so the cost of change is massive.

2.4.2 Final decision

In our application we consider the support of a new datatype to be somewhat a rare occasion. In our business, which is centered around a very specific set of products - movies and songs - and it is with this line of products that our business will be known.

With this in mind, a high cost of change is considered acceptable. With the advantages we trade in for it, it is a more than acceptable potential loss.

2.5 Internal interfaces

In this section we will go through the interfaces between our web service implementation and the business logic of our application. Furthermore we will try to explain the descisions we made and why we made them.

In our backend system we have decided to keep an abstraction layer between our WCF service implementation and the business logic such as persistence, authentication, and data querying. This choice was inspired by the facade design pattern which is known to promote low coupling between application layers. It did also fit our wish to keep a model-view-presenter separation with the web service acting as the presenter, this allows us to have several web services based on the same data model and business logic. This could be handy if the client the SMU students required a different interface for the web service than the client we wanted to create ourselves. We also made the decision in order to keep cohesion high, both in the web service implementation class and the class handling our data, while also keeping coupling low. This is examplified by the fact that we could change the design of the database without having to change anything in web service, or we could have several web service implementations using the DBAccess class without these services having anything to do with each other. As shown in the last example this decision also allowed easier re-use of the methods accessing and editing data.

In order to prevent the DBAccess file from being over a thousand lines long we decided to split it up into several files using the partial class feature. This allowed us to have several documents with more defined areas of expertise making the code easier to look through. The different files in the class mimic the entities in the database (i.e DBMovie and DBUser).

The DBAccess uses the classes Movie, Song and User to model information stored in the database. When the web service receives instances of these classes, from the DBAccess object it queries, they have to be converted to a format which can be passed as JSON to the consumer of the service. To convert the classes in to their respective wrapper classes (i.e. MovieWrapper) we decided to use extension methods as this allowed us to keep the coupling between the service and the dbaccess low. With extension methods the service itself defines the wrapper classes it uses and how to transform the database classes into these wrappers. This allows different services to return different outputs even though they both use the same DBAccess class.

3 Working with Singapore

3.1 First Hand Impressions

When starting the semester, even before the first lecture in the *Software Development in Large Teams with International Collaboration* course, rumors were circulating, most originating from students who had gone through the same course the year before.

As this was the only source of information about what to expect from working with a Singaporean team, these rumors were taken to heart and very much shaped the actual expectations of the team.

One team (from last year) was telling the story of how they had, in practice, been doing both the work they were supposed to do and the work the Singaporean team was supposed to do. This was very much an outlier in terms of extremity, but most rumors confirmed that expectations should be kept low, and that the Danish team would very much have to take on a leading role in the relationship and guide the process.

It was with these expectations we went into the video conference room for the very first time to have a virtual meeting with the Singaporean partners that we were going to work with for the following weeks.

During this first meeting, all of our expectations were seemingly proven wrong: the Singaporean team was well prepared, and we ended up going by their agenda as they had covered all that we planned to talk about and more.

What made the greatest impression on us was that the Singaporean team had already set their sights on a technology and researched exactly what this would mean for our common interface. They wished to develop using AngularJS[2], and inquired as to whether it would be possible for us to implement a RESTful[3] JSON[4]-based interface on the server.

Leaving that meeting our expectations were through the roof: despite everything we had heard it seemed that we had been paired with a professional, dedicated, and well-prepared group.

3.2 Means of Communication

Throughout the project we had a strong focus on continuously optimizing our means of communication. We focused on the communication being clear and responsive, but

[2] AngularJS is a Google-developed front-end JavaScript framework. AngularJS – Superheroic JavaScript MVW Framework, http://angularjs.org/

[3] REpresentational State Transfer, a type of API based on the Hyper-Text Transfer Protocol (HTTP), http://www.oracle.com/technetwork/articles/javase/index-137171.html

[4] JavaScript Object Notation, a lightweight data-interchange format based on a subset of the JavaScript programming language, http://www.json.org/

also comfortable. We would not want confusion to arise, but many issues in Software Development require internal discussions in the separate teams: before we can agree to a change of the API, we have to assert the cost of change on our side, whether it is worth it, and whether there are any readily availible and more efficient alternatives.

Our first meeting was agreed on terms we had no influence on. The format was a video conference in a for-the-purpose designed room. Being able to see and hear the other team in real-time gave some insight into when they were thinking, when they were getting ready to speak, etc. Despite the sound-quality being sub-par, this was a great way of communicating: clear, quick, and to-the-point.

For our second meeting we chose the technology resembling the video conference the most. The choice landed on Skype[5], which has many of the same features.

To our great disappointment Skype proved highly unreliable when using the internet-connection of the ITU: the video was blurry and the sound lagging behind. We had a choice between giving up Skype and giving up sitting together as a team when communicating.

As we moved to our respective residences, and called from there, the quality of our communication went up. With group calls it was possible for the Danish team to first meet (virtually) to discuss the agenda, before calling up the Singaporean team. We had to give up video for this (as it turns out to be a paid feature of Skype to have video in group-calls), but all-in-all it was an improvement.

3.2.1 The change to textual communication

Skype was especially used in the beginning of the project, when we were trying to come to an agreement on API design. It was very useful to have (more than) weekly calls where we could introduce, propose and discuss new features. Some items were agreed upon instantly, and others were put on the agenda for next meeting, allowing the teams to thoroughly discuss the feature and all that it might entail.

As the changes to the API became smaller and smaller and more irregular, we moved to a more text-based form of communication. We had previously been using e-mail as a way to prepare for meetings and discuss agendas, but as we had less and less to discuss on these meetings this became obsolete.

Using Facebook (as a medium) to communicate came as a natural solution as, in a young generation such as ours, the majority of people already use the service. It was quick to set up a group and use this as a form of general communication between the teams. This way of communicating preserved the advantage that teams could get together and discuss changes (whenever necessary) before reporting back, but at the

[5] Skype is, among other things, a Voice over Internet Protocol (VoIP) service, allowing participants to communicate with video and sound, http://www.skype.com/en/about/

same time we lost the "human factor" by no longer being able to see and hear each other.

We risked misunderstandings (not being able to see each other removes the factors of intonation and facial expressions in the communications), but won a lot of time and efficiency. This turned out to be an acceptable change, especially after realizing that some people express themselves better in text than others. As such, we had some issues with the language of one person of the other team, but writing to the others (each time) more or less resolved these misunderstandings.

3.3 Communicating with the Singaporean team

Immediately at the first meeting with the Singaporean team, the first hint of cultural differences presented itself in the form of Hofstede's first dimension of culture, adherence to hierarchical organization structures[1]. The Singaporean team wanted to know who the team leader of Danish team was. This was something the Danish team had not discussed, or even considered, as they were used to acting in small group democracies (i.e. with no leader). The project groups the Danish team are used to to participate in are small enough that everyone can be quite intimate with the project in its entirety. Arguably eliminating the need of government. In fact, the Danish contact person was chosen only shortly before the meeting, and with an air of necessity rather than desire.

For a while, the cultural differences seemed negligible, but as the project progressed communication between the two teams became increasingly troublesome. More than once it got to the point where one Danish team member felt it necessary to put down his foot[2]. The Danish team members perceived a tendency in the Singaporean team; the Singaporean team members would seemingly agree to proposals from the Danish team, but would then turn on it later. This tendency is illustrated by a couple of examples:

During the first meeting, a second meeting was arranged for two days later. Not long after, the Singaporean team announced that they would not be able to make the agreed appointment, and proposed to meet the following Saturday instead. Despites the verbally unambiguous agreement however, the Danish team's representatives held Saturdays meeting alone. [Reference needed]

Another example of apparent agreement: Shortly after having mutually expressed agreement on the API of the web service (and thereby the supported requirements for the Danish client) the Singaporean team started requesting API support for features already described in the shared documents. [reference needed] Indicated that the Singaporeans either had not read, or not been able to understand, the technical specification document supplied by the Danish team.

Approaching the Singaporean team's deadline, the Danes had their easter holidays. This wasn't exactly a surprise, and the Danish team had tried, and thought they had succeeded, to cement the service-requirements so only actual implementation work,

demanding little to no communication, was left for the holidays. The Singaporeans however, had planned differently. Due to having deadlines for other projects early in the process, they had delayed finalizing their requirements, and announced most of their change requests in this period. This resulted in great stress for both the Danish team, who had made plans for the holidays, and for the Singaporean, who became increasingly anxious they might not make their deadline.

The possibility that these disagreements are unrelated to cultural differences between the teams, and hence are products of dislike or indifference must be explored: The Danish team agrees that the Singaporean team's emails, Facebook messages, and appearance during meetings generally showed a positive demeanor. That, and the explicitly expressed wish to collaborate with the Danish team, even when facing disagreement makes this explanation unlikely. (see appendix A.1)

Let us explore the possibility of cultural differences acting as catalyst for misunderstanding. In some cultures, (commonly in Asia) public disagreement is somewhat stigmatized (**Hall's fifth dimension of culture: hierachical structures - ELABORATE ON THIS**) [1][4], and rather than explicitly saying *"no"* the Singaporean team might have, subtly, as perceived by western standards, hinted a reluctance that the other team was expected to, but did not know to, react on.

3.4 Interface Documentation

In the beginning of the project the two teams had not agreed on how to share documentation - each of the teams were left to their own devices.

In the Danish team we used a combination of in-code documentation and various diagrams to show the relationship between classes in the program, entities in the backend, and so on. As the project is fairly limited in scope, all members of the Danish team were aware of the more abstract ideas of the project, so this documentation was sufficient.

Additionally, in-code comments for the top layer, found in the `JsonServices` project, were deferred as this part of the project was subject to the most change and, additionally, was mostly self-documenting in its format.

Adding documentation comments to this code would simply require for the team to learn a *different* syntax.

With requests for changes coming in rapidly, some asking for features that had already been implemented, we realized that it would be more optimal to share updated documentation with the Singaporean team. We had previously shared the first draft with them in the context of agreeing on initial requirements for the API, and it was straightforward to move this to Google Docs[6] (as proposed by the Singaporean team) and

[6]Google Docs allows users to share and collaboratively edit documents online, http://docs.google.com/

```
[WebInvoke(Method = "POST",
           RequestFormat = WebMessageFormat.Json,
           ResponseFormat = WebMessageFormat.Json,
           BodyStyle = WebMessageBodyStyle.WrappedRequest,
           UriTemplate = "{id}/purchase")]
SuccessFlag PurchaseSong(string id, int userId);
```

Table 2: Example of the interface implementation (ISongService) of the service's action to purchase a song. As is seen, the code is rather verbose, in that it details request method, request- and response-format, as well as the actual URI used.

bring it up to date. Going forward this was our single reference-point for updated documentation of the API. Using Google Docs provided an easy resource and announcing changes via Facebook allowed all members of both teams to be aware of changes.

3.4.1 Format of Documentation

After our first discussion with the Singaporean team, the Danish team had the impression that they were relatively technically savvy: they had chosen to use a technology that was not the standard choice, and they had researched what this would require on our part.

As JSON is a relatively easily understandable format, and as properly designed RESTful APIs should be understandable from just the URI, we decided to use the two in combination to document it: a HTTP-method and URI heading with parameters denoted in curly-brackets, accompanied by an optional JSON-format request body as well as a required response body, also in JSON, which denotes the return format if the request was a success.

```
POST users/create
{ name: /*string*/, username: /*string*/, email: /*string*/, password: /*string*/ }
{ success: /*boolean*/, message: /*string*/ }
```

Table 3: Example of API documentation showing the request and response formats for creating a new user.

In accordance with proper RESTful design, requests that fail will return the proper error codes[3] in the header of the HTTP-response.

As we had proposed this format to the Singaporean team, and they had agreed to it, this seemed like an acceptable solution. In addition to this, we expected them to be fairly aware of how the JSON format works, as they had proposed it themselves.

3.4.2 Issues with Shared Documentation

We did not encounter any massive issues with our way of sharing the documentation of the interface.

With a shared document where all members of both teams had the rights to edit, one could imagine a situation where changes were made to the documentation without being made to the actual implementation, causing a discrepancy that might take a long time to notice and fix. Such a situation did, however, not occur.

We encountered only one issue with the whole of the documentation process, which we did not find a solution for during the time of the project.

On several occasions, when the Singaporean team requested features, the features turned out to already exist, be implemented and fully documented. This was either a case of simply not reading the documentation before posting an idea, or not understanding the format of the documentation (which there were no complaints about).

This led to a growing amount of time spent checking that the feature was indeed documented and implemented, and then informing the Singaporean team of the situation. Mostly this was a minor issue, but adding up it became a source of stress for both teams.

References

[1] Judith S. Olson, Gary M. Olson
Culture surprises in Remote Software Development Teams
ACM Queue, 2003

[2] Niels Roesen Abildgaard
Should this be included?
Facebook post in response to Robert Chai

[3] Leonard Richardson, Sam Ruby
RESTful Web Services O'Reilly, 2007

[4] James D. Herbsleb, Daniel J. Paulish, Matthew Bass
Global Software Development at Siemens: Experience from Nine Projects
Carnegie Mellon University School of Computer Science, Siemens Corporate Research

[5] Raghu Ramakrishnan, Johannes Gehrke
Database Management Systems
3rd edition 2002

[6] http://www.nytimes.com/2012/03/01/technology/impatient-web-users-flee-slow-loading-sites.html
For Impatient Web Users, an Eye Blink Is Just Too Long to Wait
The New York Times, February 29, 2012

A Appendices

A.1 Email from Robert Chai

Excerpt:

"At the end of the day, I believe though as short term we are working together on this project.. The ideal is to work things out, be proactive on suggestions and just work out on potential compromises along the way.

For that, I would like to first apologize to say that there were faults of mine relying on the workings of angularjs v0.9 as my previous knowledge was based on juz 1.5 year back while I was doing my final year project. Thank you for sharing the link on the update of the code.

Also, i would like to thank you can your team for working with us on the pagination segment. [...]

Cheers! Robert Chai"

A.2 Use Cases

Users:

- A user must be able to login, register, and edit an account.
- A user may deposit money into her account, to be used for later purchases.
- A user must be able to overview movies by the following categories: lexicographically, by release date, and by genre.
- A user must be able to locate a specific movie through text search, matching movies with any property matching the text string.
- A movie must have properties describing title of movie, year of release, director(s), genre(s) of movie, and a textual description.
- A user must be able to purchase a movie, which will then be associated with her account.
- A user must be able to rent a movie, which will then be associated with her account for 2 days.
- A user must be able to watch any movie associated with her account in the browser, as long as it is still available through the service.
- A user must be able to download any movie associated with her account, as long as it is still available through the service.
- A user must be able to find all movies that are or have been associated with her account, as long it is still available through the service (ie. movies the user has purchased or rented at a given time).

- A user must be able to find all movies that are associated with her account, as long it is still available through the service (ie. movies that the user can watch at the given time).

Managers:

- A manager's rights extend those of the user, so that a manager can do everything the user can, without purchase.
- A manager must be able to upload movies as well as provide properties and prices to them.
- A manager must be able to delete movies from the service.