Appendix

May 21, 2013

Contents

1 Web service API-specifications

1.1 M6, Movie Distribution Web Service API Specs

The web service uses RESTful concepts to provide a standardized and stateless programming interface, for interacting with the system.

Standard response
All responses consists of the following data:

Table 1: Response message objects

Message	Description
error_code	0 on success, otherwise a number indicating the relevant error. NOT TO BE CONFUSED WITH HTTP-Response-code! This is assuming the HTTP-Response-code is 200.
error_message	A message in English, describing the eventual error.
Data	Array of data. Usually contains some data object (such as users, media etc ..).
MetaData	Metadata about the data. Examples are `limit`, `page offset` etc ..

Security and encoding
All requests should be made with a valid API-key, and values hashed into a "Checksum". Requests should be accompanied by a UTC timestamp and a nonce. All authenticated requests should contain an access token.
So every request should look something like:

```
[Resource]?auth=[AUTH-STRING]&parama=xxx&paramb=yyy
```

Where [AUTH-STRING] is something similar to:

```
{ ‘‘api-key’’ : ’’AB14’’ , ’’hmac’’ : ’’XZ45’’ , ’’time’’ : ’’213’’ ,
‘‘nonce’’ : ‘‘XC98’’ ; ‘‘token’’ = ‘‘DE95’’ }
```

The only optional part of this auth object, is the token, which is only used after the user has successfully logged in.

Objects
Every area of the application has some certain objects with a well defined

structure. These are described initially in the relevant section, and utilized throughout the API. This allows for client to receive and process these objects in a uniform manner.

Parameters
Are GET or POST arguments, depending on the request-type. The parameters documentation consists of a argument name along with a description of the parameter

Response
Is an associative JSON object, with fields corresponding to the left side column of the table. The right side of the column describes the data in the field of the returned response. All responses are wrapped in `Response<TYPE>` objects. For example when calling the `GET user` method, it returns `Response<User>`.

1.2 Users (users, tokens, user data & user groups)

The users are at the heart of the system. All actions are performed by users, and their identities are associated with both actions and data entities within the system. Users are very simple, but can be extended to application specific needs. This is done by creating new user data types (unique text indexes, that can be used to store data for each user), and assigning data to the users. Tokens are used for accessing the application as a specific user, and is given upon submission of a valid email and password combination.

Table 2: User objects

Field	Description
id	The users id
email	E-mail of the user. False if currently logged in user is not permitted to read.
user_data	All data that exists for this user as an associative structure

Table 3: Token objects

Field	Description
token	A token string, to be used in further queries
issued	Date and time for issuance
expires	Date and time for expiration

Get user

Url structure: /user/{ID}
Description: Who is the user with the id {ID}?
Method: GET
Parameters: **id** Id of the user
Response: **Response<User>**
Response object containing the user object.

Get currently logged in user

Url structure:	/user/me
Description:	Who is the currently logged in user?
Method:	GET
Parameters:	None
Authorization:	Token
Response:	**Response<User>** Response message containing the user object.

Get all users (with parameters)

Url structure:	/user?group_id={group_id}&emailFilter={emailFilter} &limit={limit}&page={page}&order_by={order_by} &order={order}
Description:	Who are the users that match the given parameters
Method:	GET
Parameters:	**group_id** Id of a user group. Only show users who are members of this/these group(s) **emailFilter** Filter by email **limit** How many users to return? (default = 10) **page** Should there be an offset? Default = 1 means no offset. **order_by** Order by what column? Default = e-mail. **order** Order which way? Default = ASC.
Authorization:	Token
Response:	**Response<User>** Response object containing an array of user objects along with page number and count.

Post user access token

Url structure:	/user/token/{email}/{password}
Description:	Can i have an access-token with these credentials?
Method:	POST
Parameters:	**email** The users e-mail **password** An SHA-1 hash of the users password.
Response:	**Response<Token>** Response object containing the token.

Renew user access token

Url structure:	/user/token/renew
Description:	Can I renew this token?
Method:	POST
Parameters:	None
Response:	**Response<Token>**
	Response object containing the renewed token.

Create new user

Url structure:	/user
Description:	Create a new user with this data
Method:	POST
Content-Type	**application/json**
	User User object.
Response:	**Response<User>**
	Response object containing the new user.

Delete user

Url structure:	/user/{ID}
Description:	Delete the user with this id
Method:	DELETE
Parameters:	**ID** Id of the user to delete.
Response:	**Response<User>**
	Response object containing an empty user.

Update user

Url structure:	/user/{ID}/{oldPassword}
Description:	Update this user with this data
Method:	PUT
Parameters:	**ID** ID of the user to update.
	old-password The users current password.
	SHA-1 hashed.
Content-Type:	**application/json**

User User object with updated user information.

Response: **Response<User>**
Response object containing the updated user.

Get user data

Url structure: /userData/{userId}

Description: Get user data for the user with the given user id.

Method: GET

Parameters: **userId** Id of the user.

Response: **Response<User_Data>**
Response object containing the user data.

Create a new user data

Url structure: /userData

Description: Make a new user data.

Method: POST

Parameters: None.

Content-Type: application/json.

User_Data User data object.

Response: **Response<User_Data>**
Response object containing the new user data.

Update user data

Url structure: /userData/{id}

Description: Updates existing user data.

Method: PUT

Parameters: **id** Id of the user data to update.

Response: **Response<User_Data>**
Response object containing the updated user data.

Delete user data

Url structure: /userData/{id}

Description: Delete user data with this id

Method:	DELETE
Parameters:	**id** Id of the user data to delete.
Response:	**Response<User_Data>**
	Response object containing empty user data.

Get user group by id

Url structure:	/userGroup/{id}
Description:	Gets a user group based on it's id.
Method:	GET
Parameters:	**id** Id of the user group to get.
Response:	**Response<UserGroup>**
	Response object containing the user group.

Create a new user group

Url structure:	/userGroup
Description:	Creates a new user group.
Method:	POST
Parameters:	None.
Content-type:	**application/json**
	UserGroup UserGroup object
Response:	**Response<UserGroup>**
	Response object containing the new user group.

Update a user group

Url structure:	/userGroup/{id}
Description:	Updates an existing user group based on it's id.
Method:	PUT
Parameters:	**id** Id of the user group to update.
Response:	**Response<UserGroup>**
	Response object containing the updated user group.

Delete a user group

Url structure:	/userGroup/{id}

Description:	Deletes a user group based on it's id.
Method:	DELETE
Parameters:	**id** Id of the user group to delete.
Response:	**Response<UserGroup>**
	Response object containing an empty user group object.

Get user's groups

Url structure:	/groupOfUser/{userId}
Description:	Gets all groups that a user is member of.
Method:	GET
Parameters:	**userId** Id of the user.
Response:	**Response<UserGroup>**
	Response object containing an array of the user groups.

Insert user into group

Url structure:	/userInGroup/{userId}/{groupId}
Description:	Inserts a user into a user group.
Method:	POST
Parameters:	**userId** Id of the user to insert
	groupId Id of the user group to insert into
Response:	**Response<UserGroup>**
	Response object containing the user group.

Remove user from group

Url structure:	/userInGroup/{userId}/{groupId}
Description:	Deletes a user from a user group.
Method:	DELETE
Parameters:	**userId** Id of the user to delete.
	groupId Id of the group to delete from.
Response:	**Response<UserGroup>**
	Response object containing an empty user group.

1.3 Actions (user actions & user group actions)

Get an action

Url structure:	/action/{id}
Description:	Gets an action based on it's id
Method:	GET
Parameters:	**id** Id of the action to get.
Response:	**Response<Action>** Response object containing the action.

Get all actions

Url structure:	/action
Description:	Gets all actions.
Method:	GET
Parameters:	None.
Response:	Response object containing an array of action objects.

Get user's actions

Url structure:	/userActions/{userId}
Description:	Gets all actions available for a specific user.
Method:	GET
Parameters:	**userId** The id of the user.
Response:	Response message containing an array of action objects.

Give action to user

Url structure:	/userActions/{userId}/{actionId}/{contentId}/{allow}
Description:	Assign an action to a user.
Method:	POST
Parameters:	**userId** The id of the user **actionId** The id of the action. **contentId** Id of the content the action allows. **allow** True if action is allowed, false otherwise.
Response:	Response object containing the new action object.

Remove action from user

Url structure:	/userActions/{userId}/{actionId}
Description:	Removes a specific action from a specific user.
Method:	DELETE
Parameters:	**userId** Id of the user
	actionId Id of the action to remove.
Response:	Response object containing an empty action object.

Get user group actions

Url structure:	/groupActions/{groupId}
Description:	Get all actions available for a specific user group.
Method:	GET
Parameters:	**groupId** The id of the user group.
Response:	Response object containing an array of action objects.

Post group action

Url structure:	/groupActions/{groupId}/{actionId}/{contentId}/{allow}
Description:	Posts an action to a user group.
Method:	POST
Parameters:	**groupId** Id of the user group.
	actionId Id of the action to assign to the group.
	contentId Id of the content the action allows.
	allow True if the action is allowed, false otherwise.
Response:	Response object containing the new action object.

Remove action from group

Url structure:	/groupActions/{groupId}/{actionId}
Description:	Removes a specific action from a specific user group.
Method:	DELETE
Parameters:	**groupId** The id of the user group to remove from.
	actionId The id of the action to remove.
Response:	Response object containing an empty action object.

1.4 Media (Media & Media Category)

Table 4: Media objects

Field	Description
id	A unique id of the media
media_category	The id of the media's category
media_category_name	The name of the media's category
user	The id of the user who uploaded
file_location	The location of the connected file
title	The title of the media
description	The description of the media
media_length	The length of the media in minutes
format	The format of the file
tags	A list of tags connected to the media

MediaCategory

Field	Description
id	A unique id
name	The name of the media category

Get media with specific id

Url structure: /media/{ID}
Description: Get a specific media, based on it's id
Method: GET
Parameters: **id** Id of the media to get.
Response: Response object containing the media object.

Get all medias (with parameters)

Url structure: /media?tag={tag}&mediaCategoryFilter={mediaCategoryFilter}&nameFilter={nameFilter}&page={page}&limit={limit}
Description: Get all media matching the giver criteria.
Method: GET
Parameters: **tag** Tag connected to the media.
mediaCategoryFilter Media category that the media might be member of.

	nameFilter Filter media by name **page** Page offset. Default is 1 (no offset). **limit** How many results to return. (default = 10)
Response:	Response object containing an array of media objects along with meta information such as page offset and limit.

Create a new media

Url structure:	/media
Description:	Create a new media and get a path for your upload. This will only create an entry in the database with the meta data provided.
Method:	POST
Parameters:	None
Content-Type	**application/json** **Media** Media object containing all information.
Response:	Response object containing the newly inserted media object.

Upload a media file associated with a media

Url structure:	/mediaFiles/{ID}
Description:	Upload a media file. You give the ID connected to the posted meta data and the file you want to upload.
Method:	POST
Parameters:	**ID** The id of the media that belongs to the file.
Content-Type	**File Stream** The file to upload
Response:	Response message

Update media

Url structure:	/media/{ID}
Description:	Update the metadata of a media.
Method:	PUT
Parameters:	**ID** Id of the media to update.
Content-Type	**application/json** **Media** Media object containing the updated media information.
Response:	Response object containing the updated media object.

Delete media

Url structure:	/media/{ID}
Description:	Delete a media. This will also delete the file connected to the media.
Method:	DELETE
Parameters:	**ID** Id of the media to delete.
Response:	Response object containing an empty media object.

Get all media categories

Url structure:	/mediaCategory
Description:	Get a list of all media categories
Method:	GET
Parameters:	None
Response:	Response object containing a list of all media categories.

Get specific media category

Url structure:	/mediaCategory/{ID}
Description:	Get a media specific category
Method:	GET
Parameters:	**ID** Id of the media category to get.
Response:	Response object containing the media category object.

New media category

Url structure:	/mediaCategory
Description:	Creates a new media category
Method:	POST
Parameters:	None
Content-Type	**application/json**
	MediaCategory MediaCategory object containing all related information.
Response:	Response object containing the new media category object.

Update media category

Url structure:	/mediaCategory/{ID}
Description:	Update media category
Method:	PUT
Parameters:	**ID** Id of the media category to update.
Content-Type	**application/json**
	MediaCategory Media category object containing information of the updated media category..
Response:	Response object containing the newly updated media category object.

Delete media category

Url structure:	/mediaCategory/{ID}
Description:	Delete a media category
Method:	DELETE
Parameters:	**ID** Id of the media category to delete.
Response:	Response object containing a empty media category object.

1.5 Tags

Table 5: Tag objects

Field	Description
id	A unique id
name	The name of the tag
simple_name	The short version of the name
tag-group	Tag group

Table 6: Tag group objects

Field	Description
id	A unique id
name	The name of the tag group
description	The tag group description

Get all tags

Url structure: /tags?tagGroupFilter={tagGroupFilter}&limit={limit}&page={page}

Description: Get a list of all tags

Method: GET

Parameters: **tagGroupFilter** The id of the tag group you want to filter by
limit Amount of tags per page. Default = 10
page Page offset. Default = 1 (no offset)

Response: Response object containing an array of tag objects.

Get single tag

Url structure: /tags/{ID}

Description: Get a single tag based on it's id.

Method: GET

Parameters: **ID** Id of the tag.

Response: Response object containing the selected tag object.

New tag

Url structure:	/tags
Description:	Create a new tag
Method:	POST
Parameters:	None
Content-Type	**application/json**
	Tag Tag object with the related information.
Response:	Response object containing the new tag object.

Update tag

Url structure:	/tags/{ID}
Description:	Update a tag
Method:	PUT
Parameters:	**ID** Id of the tag to update.
Content-Type	**application/json**
	Tag Tag object containing the updated tag.
Response:	Response object containing the newly updated tag object.

Delete tag

Url structure:	/tags/{ID}
Description:	Delete a tag
Method:	DELETE
Parameters:	**ID** Id of the tag to delete.
Response:	Response object containing an empty tag object.

Get all tag groups (with parameters)

Url structure:	/tagGroups?limit={limit}&page={page}
Description:	Get a list of all tag groups.
Method:	GET
Parameters:	**limit** How many results to return (default = 10)
	page Page offset (default = 1, no offset)
Response:	Response object containing an array of tag group objects.

Get single tag group

Url structure:	/tagGroups/{ID}
Description:	Get a single tag group
Method:	GET
Parameters:	**ID** Id of the tag group to get.
Response:	Response object containing the tag group object.

New tag group

Url structure:	/tagGroups
Description:	Create a new tag group
Method:	POST
Parameters:	None
Content-Type	**application/json**
	TagGroup TagGroup object containing all relevant information.
Response:	Response object containing the new tag group object.

Update tag group

Url structure:	/tagGroups/{ID}
Description:	Update a tag group
Method:	PUT
Parameters:	**ID** Id of the tag group to edit.
Content-Type	**application/json**
	TagGroup TagGroup object with the new information.
Response:	Response object containing the newly updated tagGroup object.

Delete tag group

Url structure:	/tagGroups/{ID}
Description:	Delete a tag group (this will also delete tags connected to the tag group, or delete the connection)
Method:	DELETE
Parameters:	**ID** Id of the tag group to delete
Response:	Response object containing a empty tag group object.

Get tags belonging to a media

Url structure:	/tagsByMedia/{mediaId}
Description:	Gets all tags related to a specific media.
Method:	GET
Parameters:	**mediaId** The id of the media to filter by.
Response:	Response object containing a list of tags.

Post tag to media

Url structure:	/tagsByMedia/{mediaId}/{tagId}
Description:	Connects a tag to a media.
Method:	POST
Parameters:	**mediaId** The id of the media to connect to. **tagId** Id of the tag to connect.
Response:	Response object containing the media connected to the tag.

Get single user tags

Url structure:	/userTags/{id}
Description:	Gets all tags connected to a user.
Method:	GET
Parameters:	**id** Id of the user.
Response:	Response object containing a list of UserAccountTag objects.

Get all user tags

Url structure:	/userTags
Description:	Gets all user tags available in the system.
Method:	GET
Parameters:	None.
Response:	Response object containing a list of UserAccountTag objects.

Create new user tag

Url structure:	/userTags
Description:	Posts a new user tag.
Method:	POST
Parameters:	None.
Content-type:	**application/json**
	UserAccountTag UserAccountTag object containing all relevant information.
Response:	Response object containing the new UserAccountTag object.

Delete user tag

Url structure:	/userTags/{ID}
Description:	Deletes a specific user tag.
Method:	DELETE
Parameters:	**ID** Id of the user tag to delete..
Response:	Response object containing a empty UserAccountTag object.

Get media by user tag

Url structure:	/mediaByUserTag/{userId}/{userTagId}
Description:	Gets media based on user and usertag.
Method:	GET
Parameters:	**userId** The id of the user.
	userTagId Id of the user tag connected to the media
Response:	Response object containing a list of Media objects.

Get user tags by user

Url structure:	/userTagsByUser/{userId}
Description:	Returns all userTags that belong to a specific user.
Method:	GET
Parameters:	**userId** Id of the user.
Response:	Response object containing a list of all UserAccountTag objects.

Post user tag to media

Url structure:	/userAccountTag/{userId}/{mediaId}/{tagId}
Description:	Connects a user tag to a media.
Method:	POST
Parameters:	**userId** Id of the user that posts the tag.
	mediaId Id of the media.
	tagId Id of the user tag to connect to the media.
Response:	Response object containing the newly posted user tag.

Remove media from tag

Url structure:	/mediaByUserTag/{mediaId}/{tagId}
Description:	Removes a media from a user account tag.
Method:	DELETE
Parameters:	**mediaId** Id of the media to remove.
	tagId Id of the user tag to remove from.
Response:	Response object containing a empty user tag object.

1.6 Rating

Get all rating for specific media

Url structure:	/rating?media={media}&user={user}& limit={limit}&page={page}
Description:	Returns all the ratings / comments on a specific media.
Method:	GET
Parameters:	**media** Id of the media . **user** Id of the user that posted the rating. **limit** How many results to return (default = 10) **page** Page offset. (default = 1, no offset)
Response:	Response object containing a list of ratings.

New rating for media

Url structure:	/rating
Description:	Posts a new rating for a media
Method:	POST
Parameters:	None.
Content-Type:	**application/json** **Rating** Rating object containing all relevant information.
Response:	Response object containing the newly posted rating.

Edit rating

Url structure:	/rating{ID}
Description:	Edits an already existing comment.
Method:	PUT
Parameters:	**id** Id of the rating to edit
Content-Type:	**application/json** **Rating** Rating object containing the updated rating.
Response:	Response object containing the updated rating object.

Delete rating

Url structure:	/rating{ID}
Description:	Delete a rating
Method:	DELETE
Parameters:	**id** Id of the rating to delete
Response:	Response object containing an empty rating object.

2 Use cases

2.1 UC1: Create profile

Use case:	UC1: User creates profile.
Scope:	Web service application
Level:	User goal
Primary actor:	Regular user
Stakeholder and interests:	Regular user, who wants to create a new profile.
Preconditions:	None
Postconditions:	User has created a profile.

Basic flow:

1. A user decides to try out the system and wants to create a new personal profile.
2. User starts the client, and presses the "sign-up" button.
3. The user is then presented with a form which he can fill out his personal information (such as name, email etc ..)
4. After the user has inserted his personal information, he is presented with a message stating that the profile creation was successful.

Extensions:

4* If there is an error in the sign up process, the user gets an error message stating what the error is.

Special requirements: none

2.2 UC2: Log into the system

Use case:	UC2: User logs into the system.
Scope:	Web service application
Level:	User goal
Primary actor:	Regular user
Stakeholder and interests:	Registered user.
Preconditions:	User must have created a personal profile.
Postconditions:	User is logged into the system.

Basic flow:

1. The user starts the client and is presented with a log in screen.
2. User is prompted to insert the email and password of his profile.
3. The system checks if the credentials are correct, and redirects the user to the home screen of the application.

Extensions:

1. In case the user has forgotten his log in credentials, he is presented with a form where he can get his credentials sent to his email address.

3* In case the given credentials are incorrect, the system gives an error and asks the user to try again.

Special requirements: none

2.3 UC3: User watches movie

Use case:	UC3: User watches a movie.
Scope:	Web service application
Level:	User goal
Primary actor:	Regular user
Stakeholder and interests:	Registered user.
Preconditions:	User must be logged into the system.
Postconditions:	None.

Basic flow:

1. After the user has logged into the system, he wants to watch a movie.
2. The user searches through the list of movies available, and select one he likes and that is free to watch.
3. The user is presented with a new screen which he can stream (watch) the movie online.

Extensions:

2.* The user selects a movie he likes and presses the link.

3.* The movie that the user selected is not a free movie, and must therefore either pay a one time fee or get a subscription.

4.* The user selects to get a subscription and is presented with a form where he can enter his payment information.

5.* After the user has inserted his payment, he is presented with a new screen which allows him to watch the movie.

Special requirements: none

2.4 UC4: User posts rating

Use case:	UC4: User posts a comment and rating for a movie.
Scope:	Web service application
Level:	User goal
Primary actor:	Regular user
Stakeholder and interests:	Registered user.
Preconditions:	User must be logged into the system.
Postconditions:	User's rating and comment is posted to the site for all to see.

Basic flow:

1. When the user is done watching a movie, he feels like writing a comment about the movie and give it a rating.
2. On the movie screen there is a form where the user can write a comment.
3. The user writes a title for the comment, and writes a few lines giving his opinion on the movie.
4. The user decides to give the movie 5 out of 5 available stars because he really liked the movie.
5. The user saves the comment and the comment is added to the list of all comments that are related to this specific movie.

Special requirements: The user must not be blocked by an admin. (see UC7)

2.5 UC5: User play list

Use case:	UC5: User creates a personal play list.
Scope:	Web service application
Level:	User goal
Primary actor:	Regular user
Stakeholder and interests:	Registered user.
Preconditions:	User must be logged into the system.
Postconditions:	Personal play list is created and stored on the system.

Basic flow:

1. A user wants to create a personal play list of movies he likes and intends to watch at a later date.
2. On the personal profile page, the user creates a new play list.
3. The user creates a tag for the play list called "movies i want to see" and saves the play list with the tag as the title.
4. The user then browses through the list of movies, adding the ones he wants to watch later.

Special requirements: None.

2.6 UC6: Distributor uploads movie

Use case:	UC6: Distributor uploads a new movie to the system
Scope:	Web service application
Level:	User goal
Primary actor:	Distributor
Stakeholder and interests:	Registered users, distributors.
Preconditions:	Distributor must be registered and logged in.
Postconditions:	New movie is uploaded to the system for everyone to watch.

Basic flow:

1. A distributor has just finished making a movie, and he wants to upload the movie to the system.
2. The distributor navigates to the "upload media" page of the system.
3. He is presented with a form in which he must fill out all meta data related to the movie.
4. After writing the information, he adds a few tags, from a list of tags, to the media, making it easier to find on the system.
5. The distributor decides that this movie will not be free to watch and adds it to the "premium movie" group.
6. After all the information is in place, he is presented with an upload form in which he can upload the media.
7. After the upload is completed, the movie is stored on the system and ready for all premium members to see.

Special requirements: Distributor must have access to upload movies to the system.

2.7 UC7: System admin blocks user

Use case:	UC7: System admin blocks user from posting ratings
Scope:	Web service application
Level:	User goal
Primary actor:	System admin
Stakeholder and interests:	Registered users, system admin.
Preconditions:	User has abused comment/rating system.
Postconditions:	User is blocked from posting comment/ratings.

Basic flow:

1. System admin has gotten complaints that a user is spamming the comment section of multiple movies.
2. System admin looks into the matter, reading all comments made by the specific user.
3. System admin removes abusing comments from the comment section.
4. System admin blocks the user from posting any more comments.
5. System admin sends a private message to the user explaining that he has been blocked, why he has been blocked, and for how long.
6. System admin also sends the user a warning that he might get blocked from the system if the abuse continues.

Special requirements: None.

2.8 UC8: System admin removes media

Use case:	UC8: System admin removes media
Scope:	Web service application
Level:	User goal
Primary actor:	System admin
Stakeholder and interests:	Distributors, system admin.
Preconditions:	Distributor has uploaded media that violates the terms of agreement.
Postconditions:	Media is removed and distributor loses his distributing license.

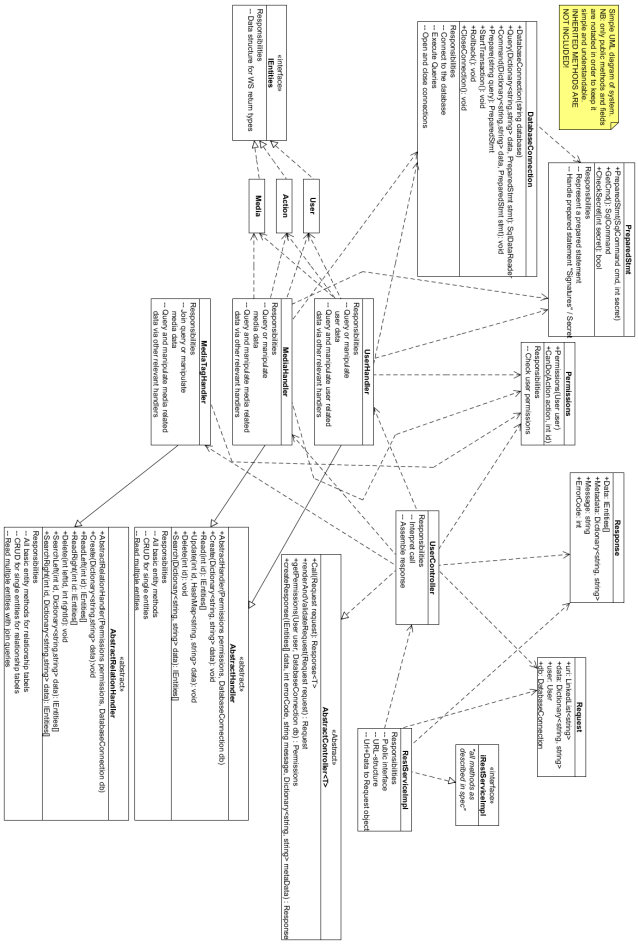
Basic flow:

1. System admin finds that a distributor has uploaded a movie that violates the terms of agreement.
2. System admin looks into the matter by checking out the video meta data and video file.
3. System admin removes the movie from the system.
4. System admin revokes the license from the distributor, degrading him to a regular user.
5. System admin sends a private message to the distributor explaining that he has lost his license and why.

Special requirements: None.

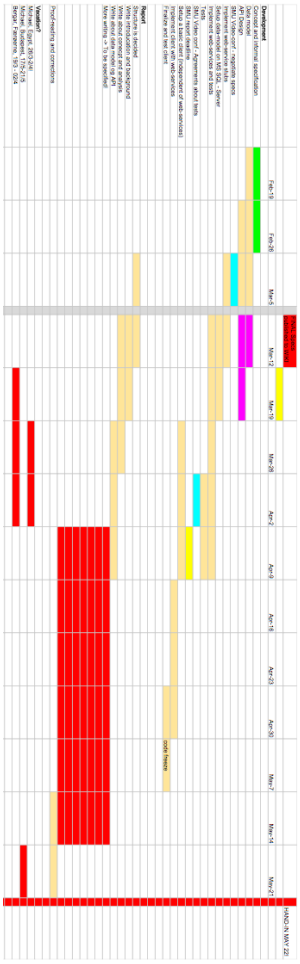
3 Class Diagram

Figure 1: Complete class diagram of the system.



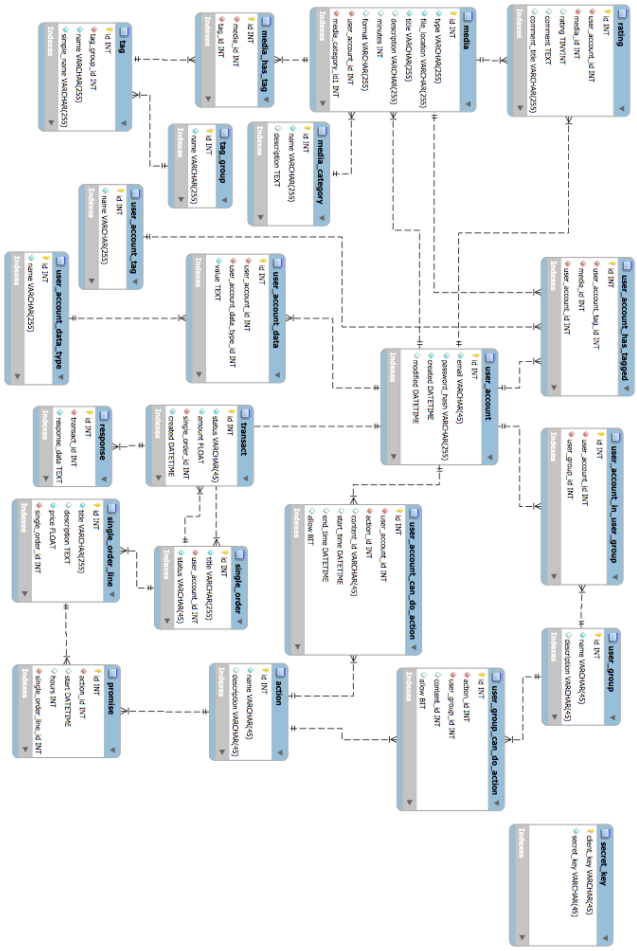
4 Project Plan

Figure 2: Project plan for the ITU group



5 Data Model

Figure 3: Complete data model for the system



6 Summary of Group Meetings

07-05-2013 - Group meeting

- Feedback -> Fine and usable. Much as expected.
- We're running behind schedule. Too ambitious.
- CODEFREEZE Tuesday morning 21/5!!!

23-04-2013 - Group meeting

- Do we need custom attributes / meta data for media?
- Nope, tags cover most of it

16-04-2013

- What has been accomplished since last time?
- Bergar:
  - Functional unit tests.
  - Paused while waiting for new structure.
  - API specifications written in report.
- Alexander: ???
- Christian
  - Started on requirement specifications.
  - Not quite done.
- Morten:
  - Started working on UML
  - Overview of system code.
- What are we doing today?
  - Bergar: Report -> Prepare for review
  - Michael: Report -> Prepare for review
  - Christian: Update code to match UML.
  - Alexander: Free...? Coding
  - Morten: Updating code to match UML
- Division of assignments / responsibilities
  - Report
  - Web service
  - Architecture
  - Database
  - Client
  - Test of web service
  - Test of client (manual?)

09-04-2013 - Group meeting

- Have been working overtime in order to reach the web service specifications for the SMU deadline.
- Simplified architecture in order to reach the SMU deadline.
  - Consider to change the architecture back, is there time?
  - Found some relative easier alternatives which will improve the process.
- Spend the day to discuss the report -> Especially problem statement.
  - Have agreed on a problem statement.

22-03-2013

- Article on tokens in REST. See comments for a good explanation on why tokens are clever: http://rest.elkstein.org/2008/01/how-do-i-handle-authentication-in-rest.html
- Article on REST authentication: http://www.thebuzzmedia.com/designing-a-secure-rest-api-without-oauth-authentication/

19-03-2013

- Follow up on homework
  - We are behind.
- Plan work of the day
  - Meeting with Singapore
- WORK WORK
- Divide work for next time
  - Finish API (Michael, Kuhre & Alexander)
  - Create demo-data stubs (Kuhre & Alexander)
  - Report
    - Concept (Michael)
    - API (Bergar)
    - Datamodel (Bergar)
  - Set up database (Morten)
    - Table set up
    - Insert sample data
- Extra group meeting:
  - Friday 14-16
  - Welcome to meet earlier if possible

12-03-2013 - Group meeting

- Meeting with Singapore
- Decisions from last time

  - Generic web service.
  - Data model
  - Pay-per-view + subscription included in web service.

- Follow up of project plan.
- Work with API
- General work.
- Homework
  - API specifications
    * Tag (MediaHasTag, TagGroup) + Media (MediaCategory) **Christian**
    * Comment + UserTag (UserTagGroup) **Alexander**
    * User (UserData, UserDataTyoe) **Morten**
    * ACL (...) **Michael**
    * Order (OrderLine, Promise, Transaction) **Bergar**

05-03-2013

- Follow up on homework
  - Michael
    * See updated project plan:
      `https://docs.google.com/spreadsheet/ccc?key=0Al6hudwhcsnmdEtZb0F1TVg2Wl9EWHRWUC1TWWhjcVE#gid=0`
    * See data model (workbench-file in git-repo):
      `https://docs.google.com/document/d/1xcQ-tLMycga5sebliyYadoc2zQifpeI1Z6AlwB7IKTY/edit#heading=h.43607z2vh6ae`
    * See API
      · Not quite finished, needs more work. Will keep working on it.
  - Bergar
    * Server login / deployment
      · Server login is working without any problems.
      · UPDATE: Web service is now running on server.
    * Remote deployment
      · Visual Studio has built in web deployment function.
      · Web deployment not yet implemented on server side. Niels says it will be implemented in near future.

- Data-model
  - PPV, Subscription, money?
    - The app is in charge of products. We handle orders, orderlines and "promises"
  - Multiple sites / apps?
    - Different databases.
    - Consider doing web service interface to generate new "App" -> New database.
- API -> Not finished ...
- Report outline -> Not finished
- Division of assignments for next time:
  - Update data model based on group meetings (Michael)
  - Create UML for server (Morten + Kuhre)
  - Example of call-flow (Morten)
  - Write API specifications (Kuhre)
  - Proof-of-concept -> same as last time. (ALL -> especially Alexander + Bergar)
  - Draft of report outline (Bergar)
  - Plan for net group meeting + plan meeting with SMU (Michael)
  - Facebook message to all who weren't here today, including homework (Morten

05-03-2013 - Group meeting & Video Conference

- 08:50: Meet up and final preparations
- 09:20: Video conference with SMU:
  - Introduction (everyone) (2 minutes)
  - Outline specification (3 minutes)
    - A very standardized web service, allowing multiple types of clients.
    - Free movies, pay-per-view and subscriptions
    - Tag-based meta-data (genre / actor / director / etc)
  - What are your demands (SMU)? (3 minutes)
  - RESTful web service + agree ... (2 minutes)

19-02-2013 - Group meeting

- 09:00 - 11:00 Lecture
- Homework follow up
  - Need better definition precisely what everyone's job is.
  - Work more together

- REST or SOAP
  - http://msdn.microsoft.com/en-us/library/dd203052.aspx
  - SOAP is probably easier for this project.
  - REST is more interesting, go with REST
- Identify ALL requirements
  - `https://wiki.smu.edu.sg/is411/2012T2-ITU-SMU_Project_Requirements`
- KEY GOAL(s) for us?
  - Focus on web service
    * Design
    * Thorough testing
    * All necessary features
  - Minimum features:
    * **MANDATORY:**
    * 0: CRUD - Account
    * 0: Login
    * ~~0: User-levels (simpler to user ACL~~
    * 0: CRUD - Media + search (category based)
    * 0: Download / Stream
    * ~~0: Rent (for a limited)~~ Ignored ... We think that login and access control should suffice.
    * **CHOSEN**
    * 1: ACL
    * 1: Search in media (free test etc..)
    * 2: Comments / Ratings
    * 2: Favorites
    * 3: Analytic -> collection of data
    * 3: Pay-per-view / Money
    * 3: Subscriptions / Money
    * ##########
    * 4: Others who watched/liked this also watched/liked
    * 4: Share on facebook button
    * 4: Revenue shares for premium content
    * 4: Approvement of films by administrator
    * 5: Login using facebook
    * 5: Remember me feature
    * 5: Cash prizes for most seen/liked films .. (could be handled manually..)
    * 5: Administration / Moderation
- Lunch

- Homework
  - Michael: Project plan
    * Fill in relevant milestones
    * Fill in tasks
  - Michael: Data model
    * Draft for data model, including relations and attributes
    * For starter, in relation to "basics"
  - Michael: Web-service specs
    * Alexander: Rest-web-service (Proof of concept)
      · Example (HelloWorld)
      · Publish on server (evt. ask Bergar)
      · Able to call web service from simple, external client.
    * Bergar: Remote Deploy - Automatic preferred (proof of concept)
      · Try if remote deploy is working on the server.
      · Try to set up remote deployment -> Publish directly from Visual Studio (locally)
    * Kuhre: Asp.net client (proof of concept)
      · Show that a client is possible
      · Use C# -> show date for example
      · Publish to server
      · Call from browser ("HelloWorld on Dec 4th 2010")
    * Morten: MsSQL-database (proof of concept)
      · Possible to run queries
      · Remote (preferred) (outside server / remote desktop)
      · Simple demo program in relation to C#
    * Morten: Get access to WIKI
      · Login / Password, so we can update -> Test that wen can create / edit articles
    * Morden: Update to Kuhre (on facebook)
    * Write update, so he knows what is going on.. + the ones that were sleeping :)

12-02-2013 - Group meeting

- Agree on idea for system concept
  - IndeeFlix
  - Free and Premium
  - PayPerView for free on premium material

  - Division of % of profit to free artists

- Group norms
  - Expectations
    * Work time:
      · About 5 hours at every group meeting + workshops
      · 5 hours homework (for starters)
    * Ambitions
      · Learn something (new)!
      · WCF
      · REST
      · Minimum grade 7 .. 10 would be great .. 12 would be NICE
      · Writing period: Yes, but will figure this out later
  - Meeting times
    * Tuesday 9-15 is HOLY
    * Everything else is planned as we go
  - Method
    * Development paradigm -> Iterativ
    * Projectplan
    * Requirements / Scenarios / User stories
  - Tools:
    * Docs -> Documents
    * GitHub -> Code
    * Facebook -> Communications
    * VS2012 -> IDE

- Define roles:
  - Project leader: Bergar
    * Responsible for what is done
    * Handle requirements and make sure they are followed
    * Make sure there is progress
    * Meeting agenda and leading meetings
  - Contact responsible : Christian
    * Communication with SMU
    * Communication with Niels
    * Keep updated on blog, mail, wiki etc..
  - Architect: Michael
    * Overview of overall program
    * Data model and program structure
    * Make sure structure is followed.

- Choosing technology

  - Desktop or web
    - Desktop: offline
    - Web: More interesting -> Decided
  - REST or SOAP
    - Read up on this for next time
- Project plan
  - Baseline
- Requirements specifications
  - Description
  - Scenarios
  - User stories

12-02-2013 - Meeting with Niels

- Have you settled on technology?
  - NO Silverlight
  - Desktop OR (Web application -> HTML(5) / CSS / JavaScript / ASP.NET)
  - REST / SOPA / pure WCF
  - C# -> Of course ...
- Your target group and requirements?
  - Spotify-like -> Buy one at a time, or subscribe to all
  - NO DRM
  - Fairly mainstream
- Group roles?
  - Not decided yet ...
  - Project leader: ???
  - Contact responsible: ???
  - Architect: ???
  - More relevant roles, or do we have too many?? -> Niels: Fine :)
- Project description and SMU Wiki
  - Project description: Cool
  - SMU Wiki: Not done yet

Niels says:
Use as much energy on the data logical; data model, web service specs, etc .. Use as little time on the technological as possible...

05-02-2013 - Introductory meeting

- Tools
  - Silverlight = NO GO!

  - GitHub for versioning
  - Entity Framework (!!)
- Development
  - Test-Driven design
  - SCRUM
- Requirements
  - System must support multiple media types
    * Video
    * Music
    * E-books
    * ...
  - Streaming
    * Download (back-up)
  - User-authentication
    * Sign-up
    * Login
    * Multiple access levels
  - Multi-uer system
  - 3 user levels
    * Super user - Our group
    * Upload user - Companies
    * Regular users - Rent media
  - Encryption (optional)
  - Client
    * Desktop client
    * Web client
  - Subscription?
    * Subscription (monthly sub.)
    * Free (pay-per-view)

7 User Manual

8 Review