Task 1

Software Design Specification

Made by Gjorge Karakabakov

Table of Contents

Introduction

This software design specification is made with the purpose of outlining the software architecture and design of Task 1 in details. The document will provide developers an insight in meeting client's needs efficiently and effectively. Moreover the document facilitates communication and understanding of the system by providing several views of the system design.

Scope

The software design document will demonstrate how the design will accomplish the functional and non-functional requirements. This document will provide a framework to the programmers through describing the high level components and architecture, database design and algorithm design.

Intended audience

This document is mainly intended for developers.

Document Overview

The next chapter of the document describes the architectural design of Task 1. The high level components and their interactions, suitable architectural patterns and design descisions applied to the whole system.

Architectural Design

Components

Document component

This is the key component of the application. This is where the user can create (upload), read, update or delete documents.

Category component

This is the component that implements the management of categories. Categories are used in the document component.

Architectural Patterns

Development of the Task 1 project will be done in MVC architectural style. The client will be able to browse the Internet and access Task 1.

MVC Architecture Style

MVC Style separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other.

- Model component – Manages the system data and associated operations with it;
- View component – Defines how data is presented to the user;
- Controller component – Manages user interaction and passes these interactions to the View and the Model.

MVC Style is used for Task 1 because there are multiple ways to view and interact with data. Also it is useful when the future requirements for interaction and presentation of data are unknown.

Component Design

Algorithm Design

Upload/Create documents

1. Get attached file
2. If file exists
3. Then add attached file to object containing other fields (name, description and category)
4. Create document
5. Redirect to documents page

```
formData = { Name: Name, Description: Description, Category: Category }
IF file EXISTS
    ADD file TO formData
END IF
CREATE DOCUMENT USING formData
IF CREATE IS SUCCESSFULL
    REDIRECT TO DOCUMENTS LIST PAGE
ELSE
    THROW ERROR
END IF
```

Download documents

1. Find requested file by document id
2. If file does not exist
3. Then redirect to documents list page
4. Else return file

```
FIND DOCUMENT WHERE FILE id == Entered id
    IF NO DOCUMENTS FOUND
        REDIRECT TO DOCUMENTS LIST PAGE
    END IF
    READ FILE FROM DISK AND RETURN RESULT
```

Get documents

1. Get search query
2. If search query exists
3. Then parse search query
4. And set search by document name or created date
5. Find documents with set search
6. Return result

```
GET query
IF query IS SET
    PARSE query
    SET SEARCH BY name OR date
END IF
FIND DOCUMENTS WITH SET SEARCH
RETURN RESULT
```

Delete document

1. If document id parameter is not set
2. Then redirect to document list
3. Else remove document from database
4. Redirect to document list

```
IF document.id IS NOT SET
    REDIRECT TO DOCUMENTS LIST PAGE
ELSE
    REMOVE DOCUMENT FROM DATABASE
    REDIRECT TO DOCUMENTS LIST PAGE
END IF
```

Edit document

1. If document id parameter is not set
2. Then redirect to document list
3. Else find document where database document id == id parameter
4. Return document data

```
IF document.id IS NOT SET
    REDIRECT TO DOCUMENTS LIST PAGE
ELSE FIND DOCUMENT  WHERE FILE id == parameter.id
    READ FILE FROM DATABASE AND RETURN RESULT
```

Create category

1. If all parameters sent from form are OK
2. Then create category
3. Redirect to category list page
4. Else throw error

```
IF params ARE SET
    CREATE CATEGORY FROM params
    REDIRECT TO CATEGORY LIST PAGE
ELSE
    THROW ERROR
```

Get categories

1. Get search query
2. If search query exists
3. Then parse search query
4. And set search by category name
5. Find categories with set search
6. Return result

```
GET query
IF query IS SET
    PARSE query
    SET SEARCH BY name
END IF
FIND CATEGORIES WITH SET SEARCH
RETURN RESULT
```

Delete category

1. If category id parameter is not set
2. Then redirect to category list
3. Else remove category from database
4. Redirect to category list

```
IF category.id IS NOT SET
    REDIRECT TO CATEGORIES LIST PAGE
ELSE
    REMOVE CATEGORY FROM DATABASE
    REDIRECT TO CATEGORY LIST PAGE
END IF
```

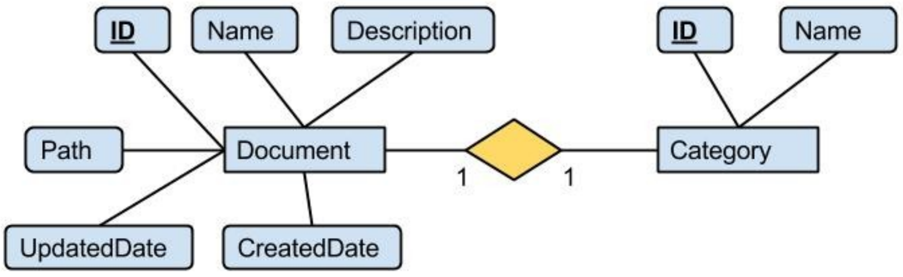
Edit category

5. If category id parameter is not set
6. Then redirect to category list
7. Else find category where database category id == id parameter
8. Return category data

```
IF category.id IS NOT SET
    REDIRECT TO CATEGORIES LIST PAGE
ELSE FIND DOCUMENT  WHERE FILE id == parameter.id
    READ FILE FROM DATABASE AND RETURN RESULT
```

Database design

ER Diagram



Normalization

Database normalization is the process of organizing the fields and tables of a relational database to minimize data redundancy and dependency. Normalization usually involves dividing a database into two or more tables and defining relationships between the tables. Since we are using non-relational database (MongoDB) normalization is not that important but in case we need to migrate the system to a relational database it is required.

There are three main normal forms:

- First Normal Form – Each filed in a table contains different information;
- Second Normal Form – Each field in a table that is not a determiner of the contents of another field must itself be a function of the other fields in the table;
- Third Normal Form – No duplicate information is permitted. For example, if two tables require an email field then the email information would be separated into separate table and the other two tables would access that email information via a key. Any change to the email would automatically reflect in all tables linked to it.

Normalization in Task 1

One significant normalization which can be mentioned in Task 1 is the category field in the documents table. Since the association between Document and Category is one-one we would only need to add the Category id in the Document table.

Document (<u>id</u>, name, description, path, createdDate, updatedDate)
Category (<u>id</u>, name)

normalized the table would look like:

Document(id, name, description, path, createdDate, updatedDate , categoryId)*
Category (id, name)

No further normalization will be required to the database.

Data Types Identification

Indexes

Document	
id	Int AUTO_INCREMENT
name	Varchar(128)
description	Varchar(255)
path	Text
createdDate	Varchar(255)
updatedDate	Varchar(255)
categoryId*	Int

Category	
id	Int AUTO_INCREMENT
name	Varchar(255)

Note: For better integration with the NoSQL database the framework used in this project automatically generates the ids as strings.