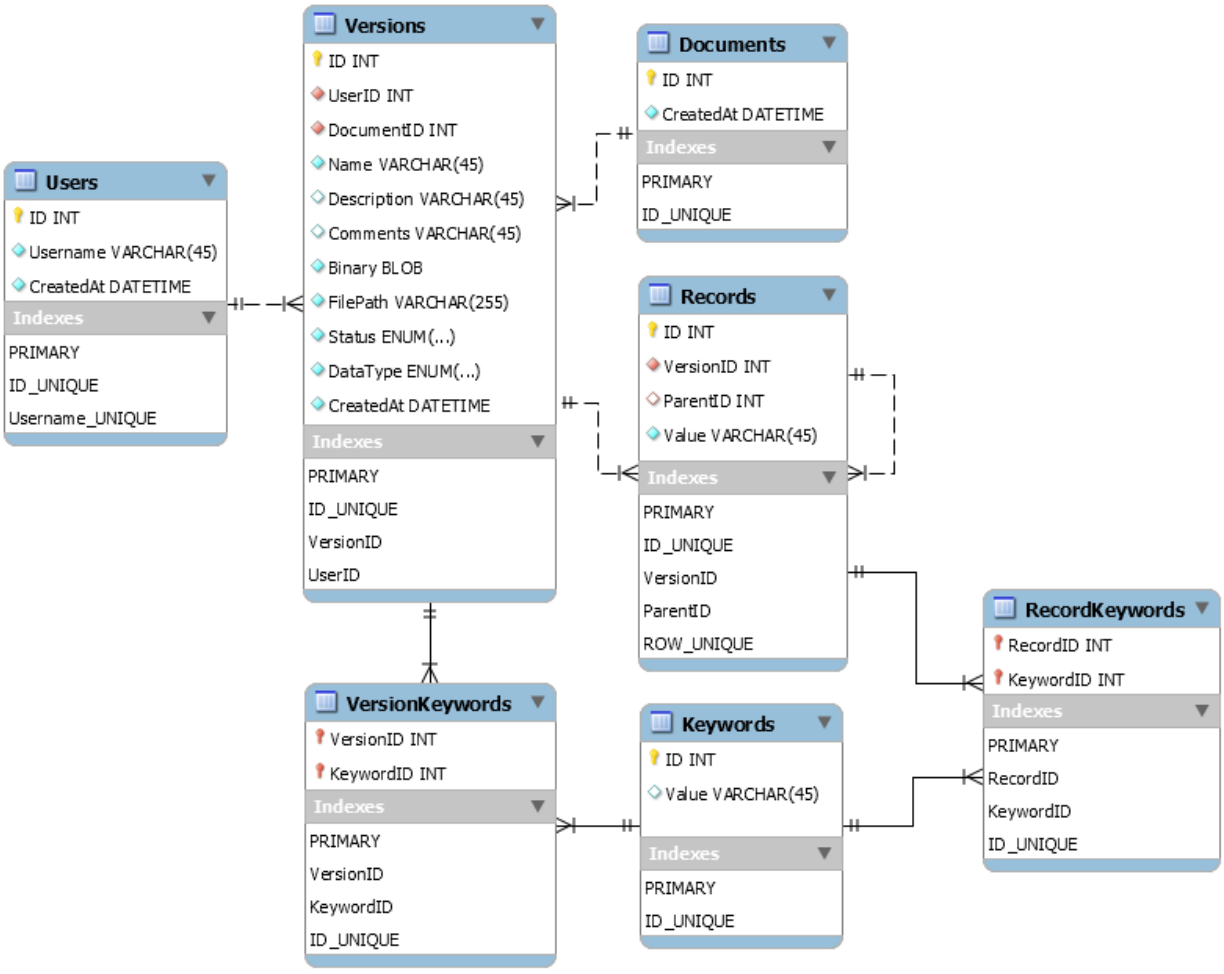
Ronald Rey – reyronald@gmail.com

Novosit's Obi-Wan Challenge No. 02 – Document Manager

Data Layer



The proposed data layer design has the following characteristics:

- The *Version* table holds the responsibility of storing all of the document data.
- The *Document* table only serves the purpose of stringing together all the versions associated to it. If we need the data, we go to its latest version (or any other).
- A *Users* table is included to track who changes what in the database.
- A document (actually a *Versions* instance, from the perspective of the database) can have a set of Records and a set of Keywords associated to it simultaneously.
- A record can be the father of another record.
- The *Records* table is single-purpose. In other words, its purpose is to be associated to a document. A foreign key relationship was added between Versions and Records table to achieve this.
- Also, in this design a record is only allowed to be related to one version, and one only (one-to-many relationship). If this is not desired, it could be fixed by adding an additional table (a junction table) to

hold the relations between the *Records* and the *Versions* table (like the case with the *Keywords* relationships, see below).

- The Keywords table is multi-purpose. It can be associated to both a Version and/or a Record at the same time. Instead of having two additional columns named VersionID and RecordID in the Keywords table, two additional tables named VersionKeywords and RecordKeywords were added. This allows us to have a many-to-many relationship and to reuse a *Keywords* instance in multiple *Versions* or multiple *Records*. In other words, keyword 'age' could appear as a keyword of versions of different or equal documents. This could not be possible in the other scenario described. In this design, there is an entry in the VersionKeywords/RecordKeywords table for every association between a Version/Record and a Keyword, if there is one. Also, if a version or a record is deleted, its atomic keywords instances would be unaffected.
- Each table has a UNIQUE index that sometimes spans more than just one column, to ensure the uniqueness of its data. The *Records* table, for instance, has a UNIQUE index composed its *VersionID*, *ParentID* and *Value* column.

Web Endpoints

A RESTful Web API would be developed on the server to work as the middle layer between the database and the web application that serves the UI. The exposed routes to do what has been requested would be as follows:

HTTP Verb	URI	Action Description	Action Methods
GET	/api/documents/:id/published	Fetches the 'Published' version of a document by its id.	Document.GetPublishedById(id)
GET	/api/documents/:id/latest	Fetches the latest version of a document by its id.	Document.GetLatestById(id)
GET	/api/documents/:id/version/:id	Fetches the specific version of a document by its document id and version id.	Document.GetVersion(docId, versId)
GET	/api/documents/:id	Fetches all the version of a specific document by its id.	Document.GetVersions()

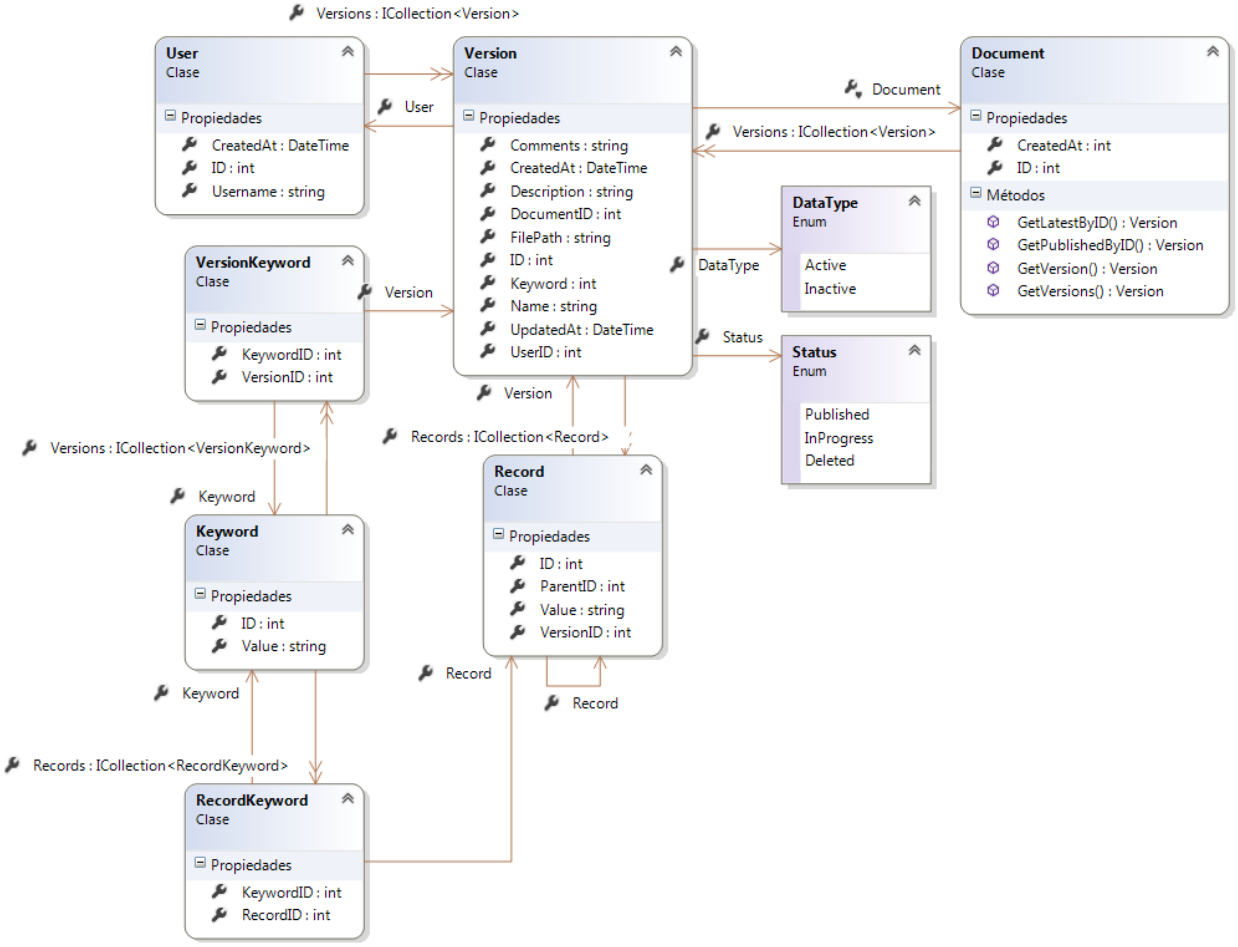
The URIs could be renamed if common consensus indicates they don't clearly reflect their purpose.

You can see a birds-eye view of the location of the Action Methods in the code by inspecting the class diagram in the following section.

In this setup, external clients could obtain, for example, the latest version of the document with id 52231 by making a GET request to /api/documents/52231/latest, which would return a JSON text with the required data.

Service Definition

The Model layer in the Server-side code would be structured as follows, mirroring our database as much as possible.



The choice between what types of loading is used when accessing the Navigation Properties will depend on the case.

Here, the Document class has the static methods used for the API. Keep in mind that they could be in the Version class instead with a slightly different name to better reflect their purpose, or in the controller that would handle the requests.

Components Overview

Such application or feature could be served under almost any modern technology stack, depending on the client's needs. On the server-side, a traditional Microsoft based stack could be as follows:

- ASP.NET Web API 2
- Microsoft SQL Server
- Entity Framework 6

Since the actual Web Application that serves the UI is a client-side Single Page Application, we can get away with merely using the ASP.NET Web API 2 framework to expose the resources. It is ideal for our scenario.

On the client-side:

- AngularJS (or BackboneJS)
- Custom styling or framework based (Bootstrap, Foundation, etc.)

Javascript MV* frameworks provide a flexibility hardly matched by other technologies, when it comes to Web Applications.

This approach is preferred, because by keeping all the UI logic on the client side, we could easily migrate the backend to other technologies if necessary with minimal effort, just by replicating the Web API (and each endpoint) on the server-side, and the whole application would be unaffected.

If the client favors Open Source technologies and is not Microsoft oriented, a PHP (Laravel based), Python (Django based), Java (Spring based) or Ruby (Rails based) could be proposed instead, with database engines like MySQL, Oracle or PostgreSql.