Context Project: Cultural Heritage

Monumentzo

Architectural design

Group 2

Date
28 March 2012

Group members
Bojana Dumeljic (4103246)
Mick van Gelderen (4091566)
Kevin de Quillettes (4077482)
Salim Salmi (4089715)

Advisors
Martha Larson
Christoph Kofler (Teaching Assistant)

Table of contents

1. Introduction

1.1. Purpose of the System

The purpose of Monumentzo is learning about monuments in an interactive, fun and intuitive way. The interactive part means that the user can browse through the monuments searching for information that he/she is interested in and meanwhile learn about the history from the time period, that area or historical figures associated to the monument. The user can also learn about the architectural attributes that the monuments has. The user can also find books about the monument which they can save in a read list to read online later or order online. They can also keep a list of favorite monuments, monuments they have already visited and the ones they still want to visit.

Monumentzo will make learning about the history of monuments fun and exciting because of its new and fresh 3D browsing interface. Our system will also be social, because it is possible to interact with other users by discussing monuments by leaving comments.

1.2. Design Goals

The system should be easy and flexible for beginners but also be intricate enough for advanced users. Thus it should be easy to handle but powerful.

As stated in 1.1 it was decided early on that the system would be for informational purposes. In order to improve on what was already available the decision was made to strive to engage users during their learning experience. To make the user engage the system should support interactivity. The option of browsing in 3D makes this possible.

The system will also be intuitive. When the information is just flat out displayed on a screen the user gets the feeling as if he was just manipulating a simple spreadsheet.
The system overcomes this by means of spatial interaction. Instead of just watching the data appear on the screen it is represented in a 3D environment. This gives the user an immediate feedback as to what he is doing in the system.

1.3. Definitions, acronyms and abbreviations

- **Monumentzo** - The name of this system.
- **ER Diagram** - Explanation of the database behind the system.
- **Lire** - A tool used for content based image retrieval. (http://www.semanticmetadata.net/lire/)
- **MVC** - Model, view, controller architectural design. (http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)
- **HTML5** - A recent version of HTML, the standard website element scripting language.
- **CSS** - A programming language for visual styling.
- **JavaScript** - A web programming language to dynamically modify web pages.
- **mySQL** - A database management system (http://www.mysql.com/)

- **PHP** - A server programming language to generate web pages and interact with a database.
- **three.js** - An open source JavaScript 3D library. (http://mrdoob.github.com/three.js/)

1.4. References

- http://www.deventeropdekaart.nl
- http://rijksmonumenten.info

1.5. Overview

This document contains the proposed software architecture in chapter two. The second chapter discusses the decomposition of the subsystems, hardware/software mapping, persistent data management, global resource handling, access control, concurrency and boundary conditions are discussed in detail. For a more detailed overview of the chapter read 2.1.

2. Proposed software architecture

2.1. Overview

This document will outline the technical design of Monumentzo. Chapter 2 will discuss the following parts of the design of the components of Monumentzo:

- Section 2.2 will discuss all the subsystems that make up the complete software suite that we call Monumentzo.
- In section 2.3 you can find everything about the mapping between the hardware and the software.
- For the information about the all the data that Monumentzo is going to store and how is splitted in different databases we will refer you to section 4 of chapter 2.
- The next section, section 2.5, will discuss the how the information that Monumentzo contains will be presented to the different kinds of users that will play a role in the day to day operations of the system.
- Section 2.6 will discuss the concurrency of the subsystems and how Monumentzo deals with any concurrency issues that might occur.
- The last section, section 2.7, of this document will talk about the boundary conditions that the system has to deal with.

2.2. Subsystem Decomposition

Monumentzo contains three major subsystems of which two are described in detail here. The third subsystem is described in more detail in section 2.4. The two subsystems that are described in this section are the frontend of Monumentzo, the part that the users will interact with, and the information gatherer that we will use to gather information from external sources.

Diagram 2.1 shows the subsystem decomposition of Monumentzo. The Monumentzo software uses the Model View Controller (MVC) architecture pattern. This architecture allows for a lot of flexibility and reusability because various versions of components can be switched easily without breaking functionality. Next to these MVC components there is a database, an information gatherer and various library. These components will be explained in more detail in this chapter.

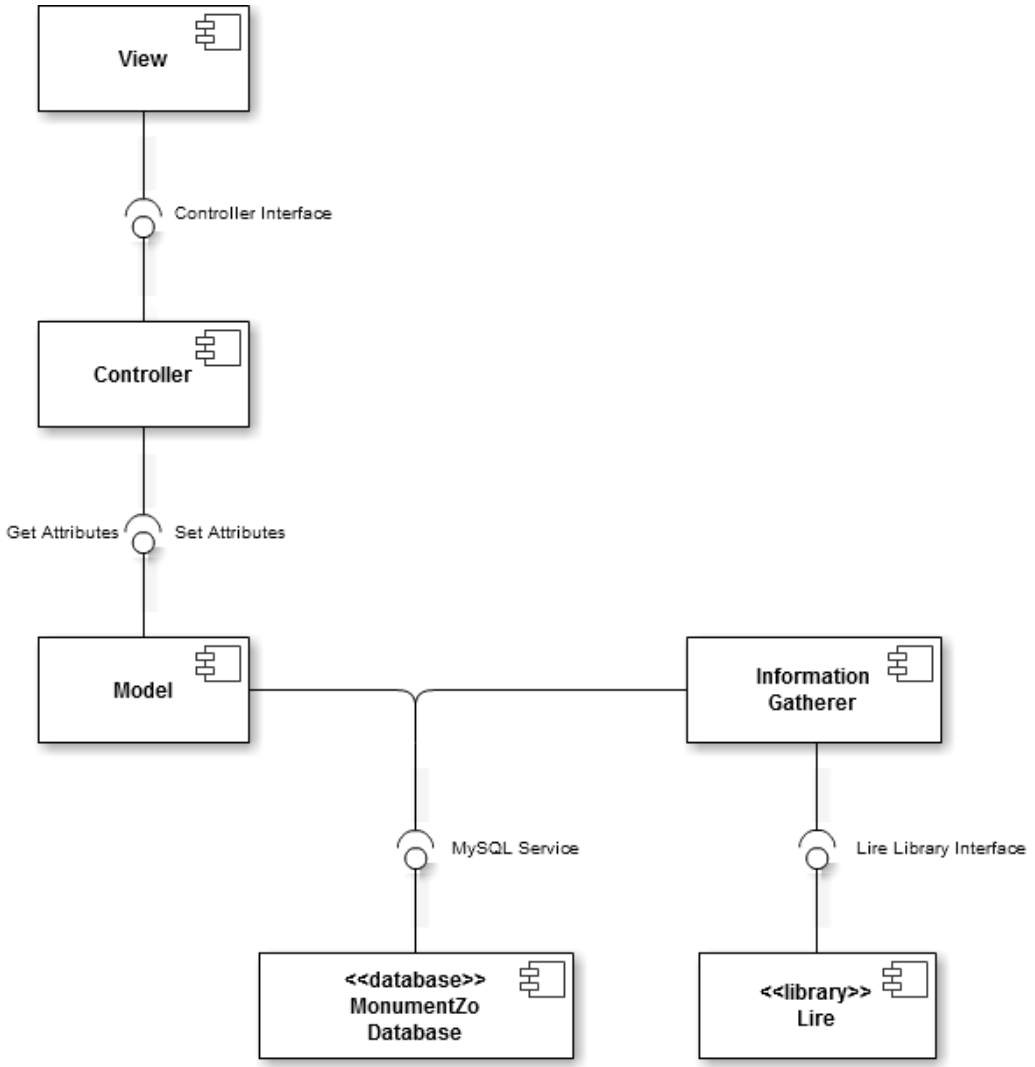

Diagram 2.1: Subsystem Decomposition Diagram

2.2.1. Web server

The first major subsystem, the frontend of the system, is set up according to the MVC architectural pattern. In this design all the requests that the users will make through the view are handled by the controllers. This means that there is a controller for each type of request that the users can make, as can be seen in diagram 2.2.

Another result from the decomposition into a MVC architecture are the models, these special classes are the only classes that interact with the data, for Monumentzo this means that there is a class for each type of external data, whether it is the database or some kind of external resource, which can also be seen in diagram 2.2.

The views are not in the class diagram, because these pieces are not classes but are made by HTML5, CSS and JavaScript and use PHP to make the page dynamic. The view also uses the three.js open source JavaScript library to make browsing in 3D possible.

2.2.2. Information gatherer

The second major subsystem is the information gatherer. This subsystem uses the pipes and filters architecture. It contains three important classes and one information class to take the information from one step in the pipeline of the gatherer to the next. The names and dependencies between the four classes can be found in diagram 2.3.

The three classes (downloader, extractor and writer) have one task each which are an exact mapping to the steps in the pipeline of the information gatherer. These tasks are downloading information from the appropriate sources, extract the useful information from the downloaded information and writing the information to the database or presenting it live to the user.

The only reason that the MonumentObject class is needed is because the information gatherer needs a way to keep track of the information in the information pipeline.

The back-end will also have a text processor that we will use to process the description of each monument to find words in it that are architectural attributes so they can be added as tags to the monument. These tags can later be used to gather more information about the monument.

The backend also contains Lire. Which will be used to compare the images of the different monuments to find graphically similar images containing similar monuments.

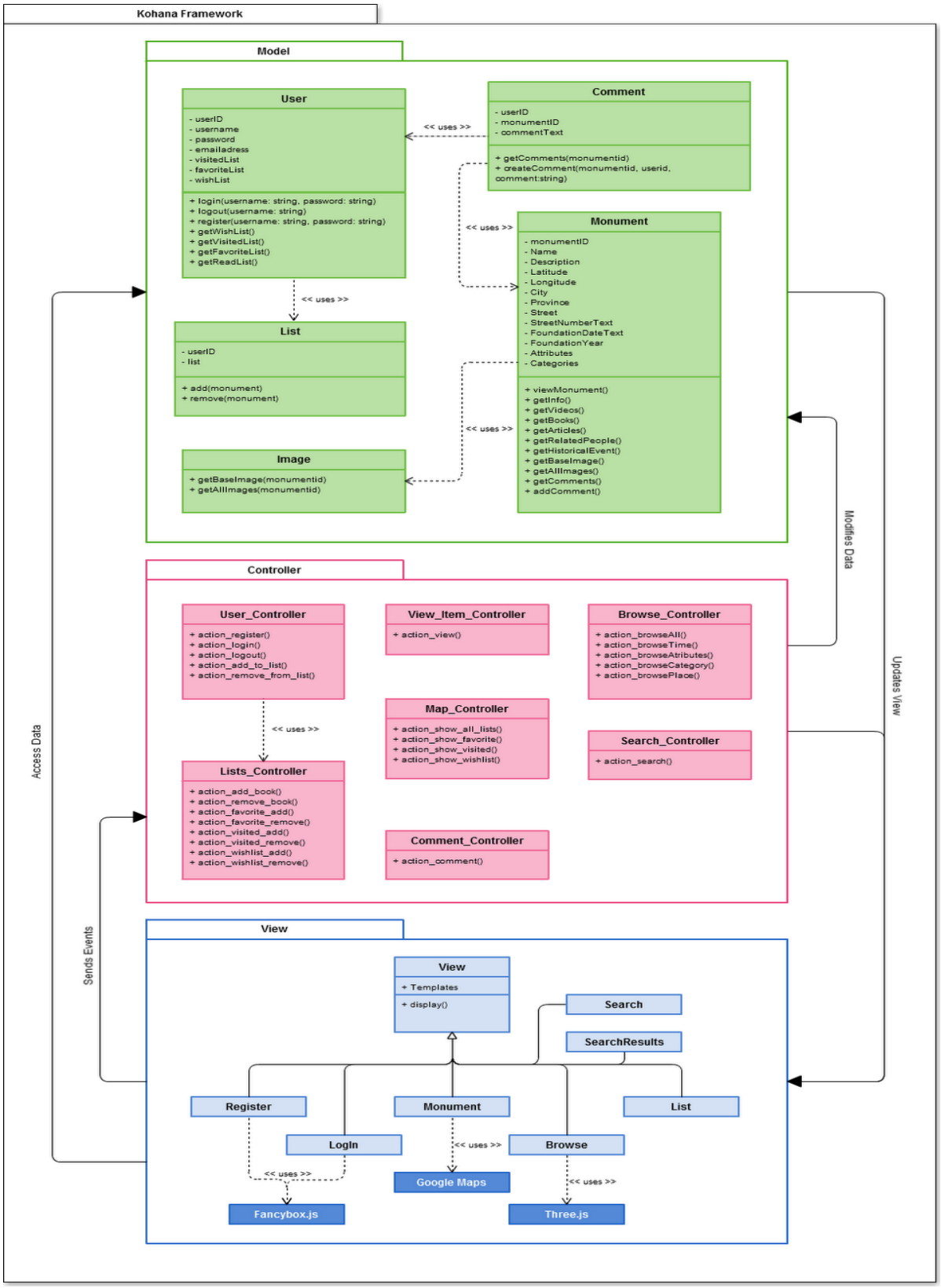
Kohana Framework
Model
User
- userID
- username
- password
- emailadress
- visitedList
- favoriteList
- wishList
+ login(username: string, password: string)
+ logout(username: string)
+ register(username: string, password: string)
+ getWishList()
+ getVisitedList()
+ getFavoriteList()
+ getReadList()
<< uses >>
Comment
- userID
- monumentID
- commentText
+ getComments(monumentid)
+ createComment(monumentid, userid, comment:string)
<< uses >>
Monument
- monumentID
- Name
- Description
- Latitude
- Longitude
- City
- Province
- Street
- StreetNumberText
- FoundationDateText
- FoundationYear
- Attributes
- Categories
+ viewMonument()
+ getInfo()
+ getVideos()
+ getBooks()
+ getArticles()
+ getRelatedPeople()
+ getHistoricalEvent()
+ getBaseImage()
+ getAllImages()
+ getComments()
+ addComment()
<< uses >>
List
- userID
- list
+ add(monument)
+ remove(monument)
<< uses >>
Image
+ getBaseImage(monumentid)
+ getAllImages(monumentid)
Modifies Data
Updates View
Access Data
Sends Events
Controller
User_Controller
+ action_register()
+ action_login()
+ action_logout()
+ action_add_to_list()
+ action_remove_from_list()
<< uses >>
Lists_Controller
+ action_add_book()
+ action_remove_book()
+ action_favorite_add()
+ action_favorite_remove()
+ action_visited_add()
+ action_visited_remove()
+ action_wishlist_add()
+ action_wishlist_remove()
View_Item_Controller
+ action_view()
Map_Controller
+ action_show_all_lists()
+ action_show_favorite()
+ action_show_visited()
+ action_show_wishlist()
Comment_Controller
+ action_comment()
Browse_Controller
+ action_browseAll()
+ action_browseTime()
+ action_browseAtributes()
+ action_browseCategory()
+ action_browsePlace()
Search_Controller
+ action_search()
View
View
+ Templates
+ display()
Search
SearchResults
Register
Monument
List
LogIn
Browse
<< uses >>
Google Maps
Fancybox.js
Three.js

Diagram 2.2: The class diagram for the frontend

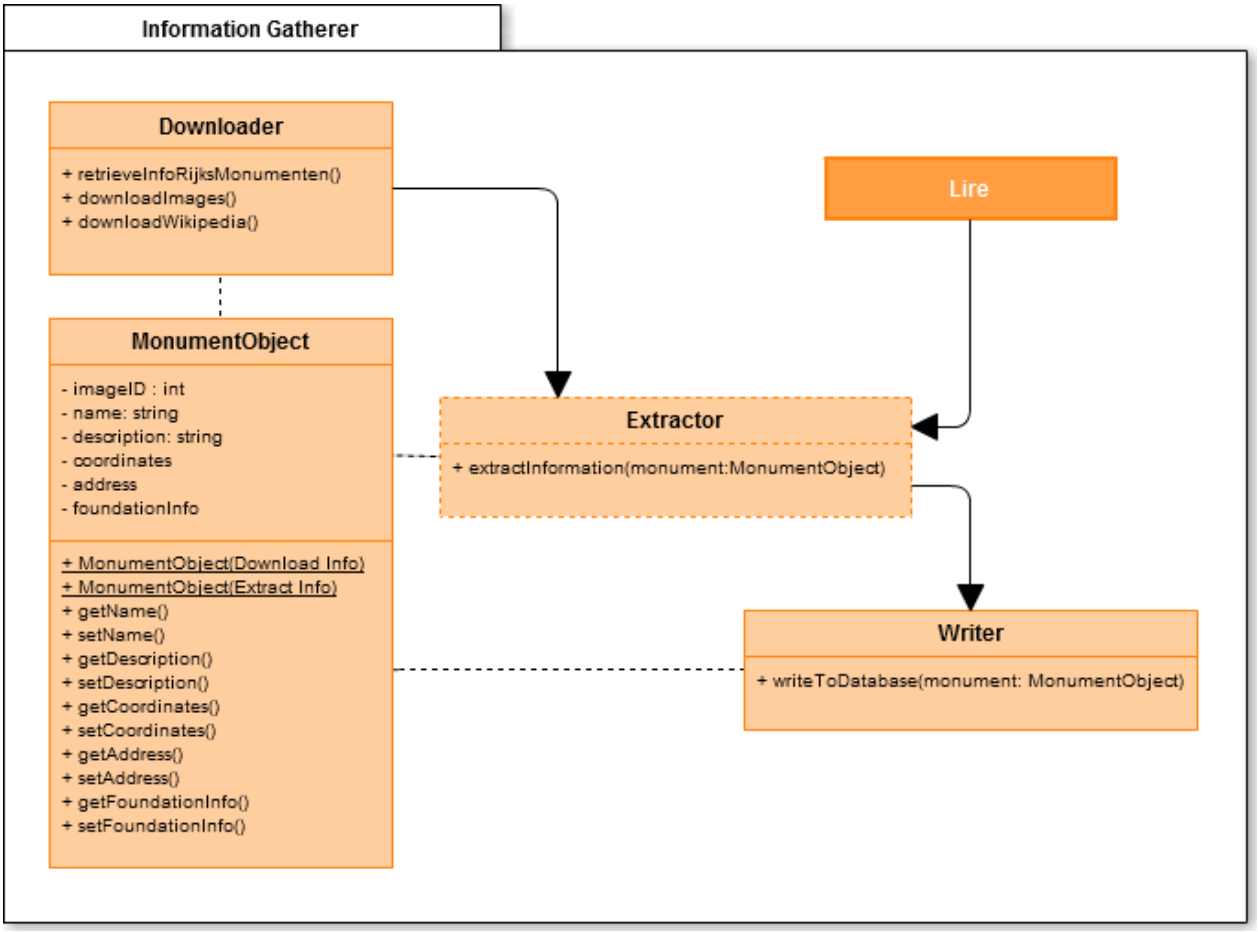

Diagram 2.3: The class diagram for the backend

2.2.3. Design Principles

The frontend and backend design takes into account a couple of design principles. Below is a list of what, why and how each design principle is taken into account.

Divide and Conquer
Monumentzo uses the MVC architecture. This architecture enforces a division of the different parts and it splits the system in a data access part, a control part and a part that defines the views for the users. This makes it possible to possible to divide the work and work in parallel. The MVC architecture also makes it possible to reuse or change components without making a lot of changes to other components.

Increase cohesion where possible
The software architecture of Monumentzo has a high cohesion where it it needed and where it makes sense. The type of cohesion with the MVC architecture is communication cohesion, because all the components in the system that operate on the same kind of data is grouped together. For example, the models and only the models communicate with the database.

Reduce coupling where possible
The architecture that Monumentzo uses, the MVC architecture, implicitly enforces this design principle, because of the separation of concerns that comes with the chosen architecture. The only coupling between the components is the transferring of data for the users.

Keep the level of abstraction as high as possible
Monumentzo is trying to keep the abstraction as high as possible. One example of this design principle are the models of the frontend. These classes hide the exact sql query's that are needed to get usable data from the database.

Increase reusability where possible
Monumentzo tries to increase reusability where possible by keeping the interaction between components as little as possible. Monumentzo makes it possible to get all the needed information in a couple function calls.

Reuse existing designs and code where possible
Monumentzo complies with this design principle because of the use of the Kohana PHP Framework. Kohana has the different kinds of functionalities that are needed for Monumentzo built-in, so Monumentzo reuses the designs and patterns for handling requests from users or accessing the database.

Design for flexibility
Monumentzo implicitly implies this principle by using the MVC architecture and the Kohana Framework. The architecture and Kohana both make it possible to keep the components separate with a low coupling. This means that the components can be easily adapted without/ with little effort to other components.

Design for testability
Monumentzo does design for testability, because the design of the system makes it more easy

to test the components and subsystems. Another things that shows that Monumentzo takes this design principle serious is the use of the Kohana PHP Framework, because it has testing functionality.

Design defensively
To a certain extent Monumentzo takes this design principle into account, because Monumentzo doesn't trust the input that users can give. This means that all the input that the users can give is checked and complies to all conditions.

2.2.4. Interface of each sub-system

The information gatherer gathers the information and adds it to the database or passes it live to the view. This way the model can use it to display the new information it to the user.

The gatherer obtains the following information offline in the backend. This is done before the launch of the system.

- Rijksmonumenten.info for the metadate about the monument and the images. Data from WikiMedia Commons and the Monument Registry is included in the metadata from this source.
- Wikipedia to enrich our own database with relevant events or people to the monuments

This information is stored in our database.

The following list is of the information that is gathered and displayed in real time. Thus whenever a user wants to view a certain item the system has to gather this information from the external resource.

- YouTube and Vimeo for relevant videos.
- Google Books and WikiBooks for relevant books.
- Wikipedia for a short description and reference link to articles on their website about the monument, attributes of the monument, etcetera.

The gatherer uses the base image of a monument from the database to find graphically similar monuments to recommend to the user.

2.3. Hardware/Software Mapping

The Monumentzo system is made to run on at least one physical server, but possibly more if needed, because the system consists of three complete and separate programs. These programs are the web server, the database management system and an information gatherer.

The web server is the most critical part of the system. It consists of three subsystems. These subsystems are the models, the views and controllers as can be seen in the diagram below. They are executed when needed and they run in the web server process.

The web server is also dependent on external resources. The external servers from Youtube, Vimeo, Google books and Wikibooks are not critical, because they only get queried for for extra information as is stated in requirement 6.

The database management system runs in its own process separately from the web server and the information gatherer. Its only task is to store and manage the information and information access for Monumentzo. It is the connection between the information gatherer and the web server and thus one of the critical components of our system.

The third and final program is the information gatherer. It consists of three subsystems, which can be found in the diagram below and its task is to download, extract and put the data, that can't be retrieved during the requests of the users to the web server, in the database. See requirement 3 for the type of information that the information gatherer retrieves. It does that in its own process and it is possible to run the information gatherer in parallel with the web server.

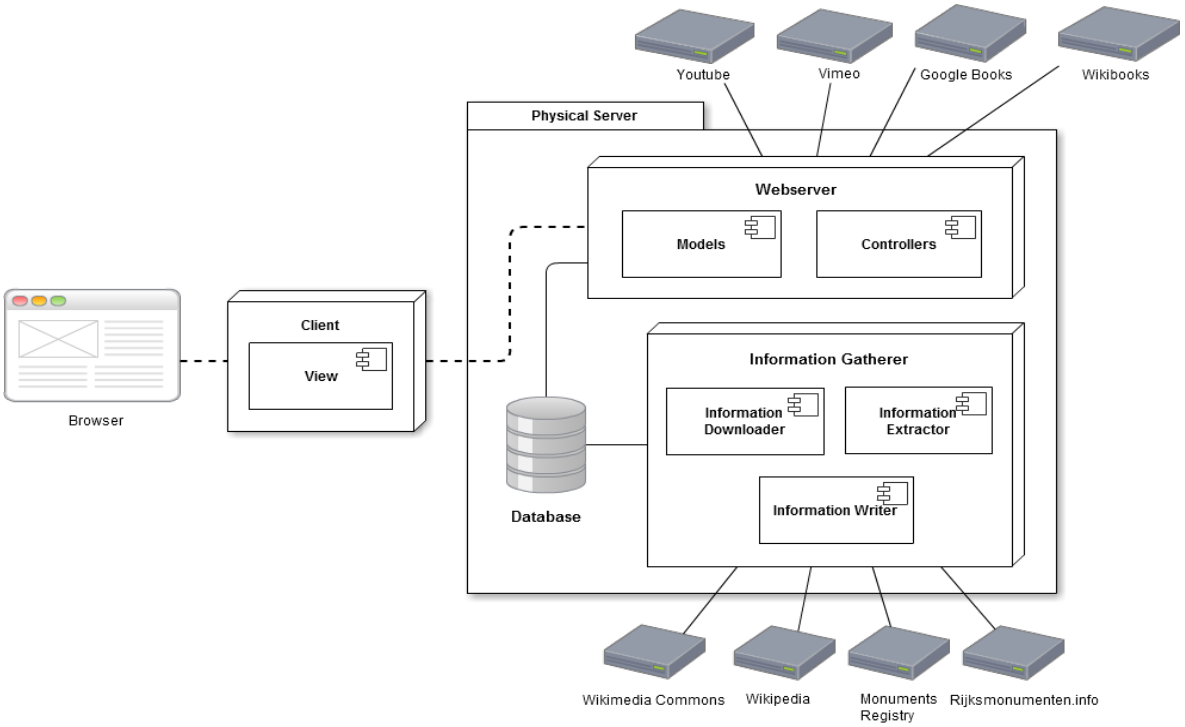

Diagram 2.4: Hardware/Software mapping

2.4. Persistent Data Management

In diagram 2.5 below the entity relationship diagram is shown. This section will explain every table of that diagram. The gray tables are used to resolve many to many relationships. The database management system will be MySQL.

Monument
The table monument contains all the relevant information that has been gathered from “Rijksmonumenten.info” and “Rijksdienst voor het cultureel erfgoed” as is stated in function 3 of the requirements. By saving this data functions like searching for a monument based on name or place will be made possible. Displaying the monument on a map will also be possible with the longitude and latitude.

TextTag and Monument_TextTag
Requirement 8 says that each monument will have one or more tags of architectural attributes. These tags will be gathered beforehand and each tag will be saved as a VARCHAR in the table TextTag. Requirement number 4.a states that the user should be able to browse on these tags. It is necessary that the system knows which tag is linked to which monument and each monument needs to know what his own tags are. Thus each monument has multiple tags and each tag has multiple monuments it belongs to. This creates a many to many relationship and will require an extra table to resolve. Monument_TextTag links the tables Monument and TextTag and remove the many to many relationship. Both are foreign keys and the combination of the two will be the primary key.

Category and Monument_Category
The same rule 4.a that applies to TextTag applies to Category so they are very similar. The table Category was also made to facilitate browsing and only contains two columns. Instead of a Tag it has a column for categories. The categories are gathered from rijksmonumenten.info. Simple examples of categories are house, castle and bridge. The same reasoning for making an extra table in TextTag applies to Category.

Monument_Image
In requirement 9 is specified that each monument will have a list of images of other monuments that look similar to its own image. The information of which monument looks similar to the requested one will be gathered and stored beforehand by means of image recognition. Each monument image will have multiple similar images. This relationship will create a many to many situation in its own table so the table Monument_Image was added to resolve this.

User
Requirement 2 states that the system has to have a login function so users can make an account and leave comments. For that reason the table User was made. In this table typical login information like name, password and email address are stored. Also every user account has an ID associated with it. This ID is used to keep a list of favorite monuments and to support

the comment function.

Comment
Every user who logged in can leave a comment at a monument as is specified in requirement 11. This comment is saved in the Comment table for other users to see. When saved the date, which user posted the comment and on which monument the comment was posted are also recorded. The comment is given an ID as there is no column or combination of columns that makes a comment unique.

FavoriteList - VisitedList - WishList
Requirement number 7a says that a logged in user must be able to keep a list of monuments. There are 3 lists a user can keep. A list with his favorite monuments, a list with monuments the user visited and a list with monuments he wants to visit. Three tables were made for these list FavoriteList, VisitedList and WishList respectively. They consist of an UserID foreign key to identify to which user the list belongs and a MonumentID foreign key to refer to the monument. Neither ID's are Unique so both are used to make up the primary key.

ReadList
Lastly number 10 requires that the user has to be able to keep a list of books that he found on the system by searching monuments. Because digital copies of books will not be stored in the database a link to the google books or wikibooks page will be saved along with the ID of the user that saved it to his list.

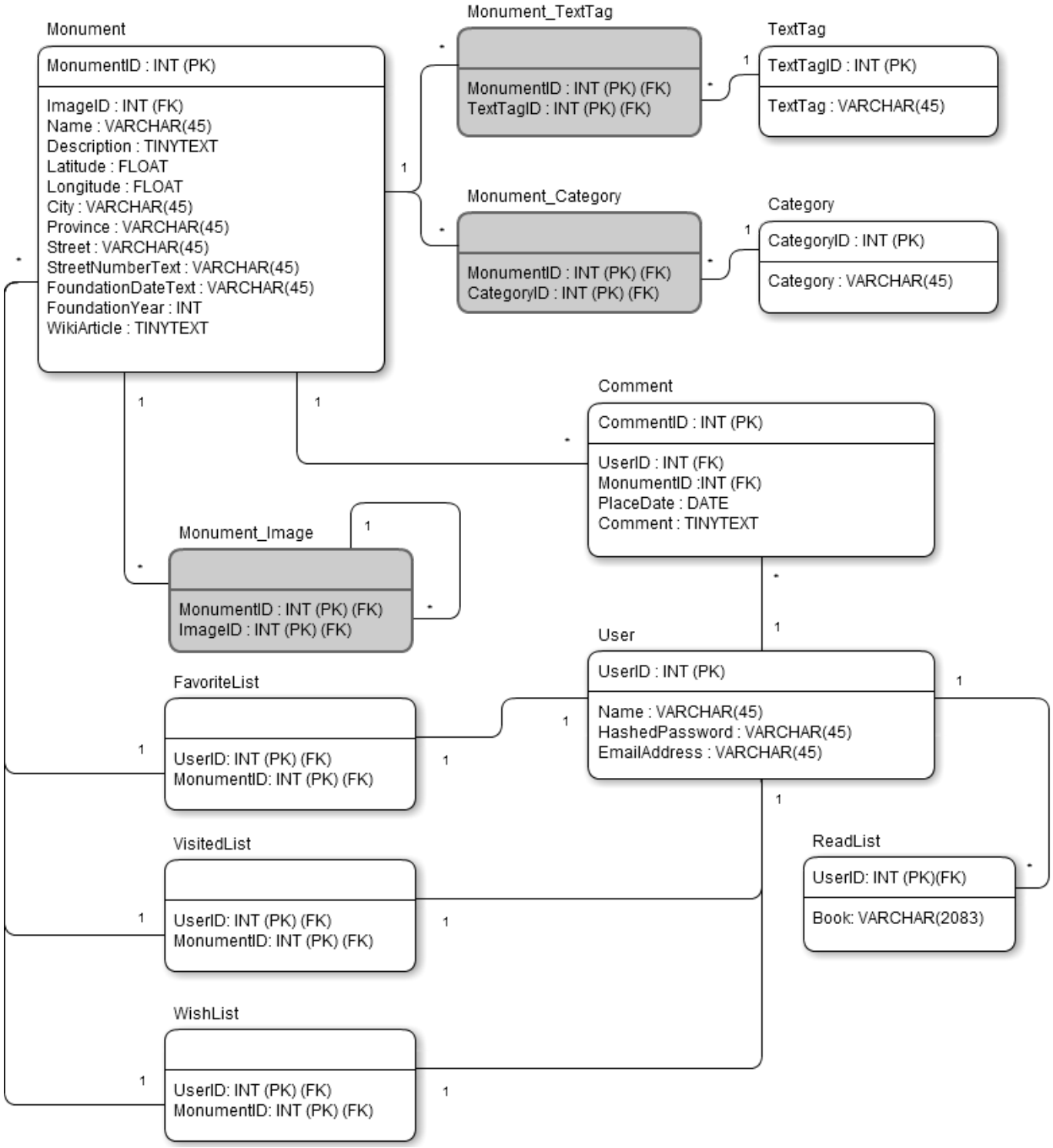

Diagram 2.5: The ER diagram

2.5. Global Resource Handling and Access Control

In this subchapter we will discuss the way the systems uses external resources in 2.5.1. Then we will discuss how we will deal with security and access control to different part of the system in 2.5.2.

2.5.1. Global Resource Handling

Various external resources are used in the system, the list of external resources can be found in requirement 6. They are all open source and accessible through their APIs. All data will be gathered by using PHP to fire queries to the various APIs and to process the received information.

We will gain access to the YouTube and Vimeo videos by embedding their players using JavaScript.

We will use the Google Maps API to gain satellite images of the monuments location using the coordinates in the metadata, if we do not have an actual image of the monument.

2.5.2. Access Control and Security

The system has two types of users, an anonymous user and a logged in user. This means that some options will be obscured for the anonymous user but all options will be visible for the logged in user. Users that are not logged in will only be able to search and browse. When a user registers and logs in they have a lot more options, see requirement 1 and 2. For example placing a comment, adding monuments their favorites list, add a book to their read list. When the view is creating the page there will be a check to see if the user is logged in to make sure not logged in user cannot access those options.

For safety and privacy issues, security will be maintained by hashing users passwords. This can be done easily with the Kohana framework we are using. But also encryption and validation.

2.6. Concurrency

Monumentzo has at least two processes that run in parallel. The possible combinations of components that run in parallel are: the database management system and the information gatherer or the database management system and the web server.

When the database management system and the information gatherer run in parallel there will only be communication from the gatherer to the database management system. All the components in the information gatherer, the download component, the extraction component and the information writer, will all run in series. Between these components there should be, under normal circumstances, one-way communications from the downloader to the extraction component and from the extractor to the information writer. This should make sure that no deadlocks occur.

If any error occurs then it will be passed back to the previous component or if that's not possible the current component will handle the error and the the component that handles it should try to recover from the error.

In the information gatherer there should only be one communication line to the database. The only component of the information gatherer that interacts with the the database is the information writer and the database management system should make sure that the concurrent access to the database is handled properly.

If the web server and the database management system run in parallel there will be lots of potential ways that a deadlock could occur. One place where a lot of deadlocks could occur is the handling of the user requests, but it is expected that the web server and the php interpreter will handle parallel request and also that they will keep the request separate so they cannot communicate between each other.

The second place where there could occur deadlocks is between each model and the database. This possible problem point is one of the things the database management system can handle perfectly without any adaptation so the database management system will handle all the parallel requests to the database.

Lastly, it is also possible to run all three components in parallel, but there won't be any direct communication between the information gatherer and the web server. All the communication between the information gatherer and the web server is via the database. So the only place where there could be a potential deadlock is in the database. To resolve these deadlocks the database management system will handle parallel access to the databases. For the rest of the components the problems fall in one of the categories discussed earlier in this section so more information can be found there.

2.7. Boundary Conditions

As was discussed in chapter 2.3 Hardware/Software mapping the system has a couple of critical components. In this section explanations will be given as to why they are or are not critical to the functioning of the system and what will be done in case of a system crash.

The web server is the most critical part of the system. The web server provides the user visual information. In the event that it crashes the user will not be able to do any more request and no visual feedback will be shown other than error messages. To combat this more servers could be added.

Data used in the system is gathered from a lot of different location, for example wikipedia, monumentenregister.info and google books. The bulk of this information will be stored beforehand so when they are for any reason not accessible anymore the data will already be on the server. Some data like audio visual material however will not be stored. The data services used in the system from which information can not be stored won't be essential components, because they provide extra information for the user to access and are not required for the system to function.

A part of the back-end, namely the part that fills our database with additional information for monuments, will be run periodically because it requires a lot of processing. This part won't be critical for the system as there will already be data stored for usage.