

Master's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

3D Atlas of Human Bones

Bc. Michal Šmrha

January 2015

/ Declaration

TODO

Abstrakt / Abstract

TODO

TODO



Contents /

1 Introduction	1
1.1 Project Background.....	1
1.2 Thesis Content	1
2 Problem Analysis	2
2.1 Role Definition	2
2.2 Requirements.....	2
2.2.1 Functional Require- ments – User	2
2.2.2 Functional Require- ments – Editor	2
2.2.3 Functional Require- ments – Administrator.....	3
2.2.4 Non-functional Re- quirements	3
2.3 Target Group	3
2.4 Existing Solutions and Al- ternatives.....	3
2.5 Analysis of the Current Ver- sion of the Atlas	4
2.5.1 Bone Selector	4
2.5.2 3D Viewer	4
2.5.3 PHP Parts	5
2.5.4 Editing	5
2.5.5 Identified Flaws	5
2.5.6 Conclusion	5
2.6 Project's Contributions	5
3 Relevant Technology	7
3.1 3D Technology for the Web	7
3.1.1 CSS 3D Transforms.....	7
3.1.2 WebGL	7
3.1.3 Flash	8
3.1.4 Other Plug-in Tech- nologies.....	9
3.1.5 Performance Test 1: Cubes (WebGL and Flash)	9
3.1.6 Performance Test 2: Aquarium (WebGL)	10
3.1.7 Frameworks (WebGL) ...	10
3.1.8 Conclusion	10
3.2 Server-Side Language	11
3.2.1 IEEE Spectrum Rank- ing	11
3.2.2 Personal Experience	12
3.2.3 Conclusion	12
3.3 Server-side Framework.....	12
3.3.1 JavaServer Faces	12
4 Application Design	13
4.1 Use Cases	13
4.1.1 User: Select Language ...	14
4.1.2 User: Navigate to Page ..	14
4.1.3 User: View Page	15
4.1.4 User: View Model	15
4.1.5 Registered User: Login ..	15
4.1.6 Registered User: Lo- gout	15
4.1.7 Registered User: Change Password	16
4.1.8 Editor: Create page	16
4.1.9 Editor: Edit Page.....	16
4.1.10 Editor: Delete Page.....	17
4.1.11 Editor: Managing Models.....	17
4.1.12 Editor: Manage Images..	17
4.1.13 Editor: Manage Mod- el's Labels	17
4.1.14 Administrator: Man- age Users	18
4.2 Graphical User Interface.....	19
4.3 Business Objects	19
4.4 Overall Architecture and Classes.....	21
4.4.1 Client Side	21
4.4.2 Servlet	21
4.4.3 Managed Beans	21
4.4.4 Entities	22
4.4.5 Persistence	23
4.5 Architecture and Classes - Mature	23
4.5.1 Service Layer	23
4.5.2 Entities and Page Components.....	24
4.5.3 Special View Classes.....	25
5 Implementation	26
5.1 Data Model	26
5.1.1 Setting Up a Local Database	26
5.1.2 Creating Tables	26
5.1.3 Generating Entities	26
5.1.4 PageComponent Ab- stract Class.....	27
5.2 Service Layer	27

5.2.1 BasicService	27	5.5.11 Using the Script, Re-	
5.2.2 LanguageService – Ex-		quired Variables.....	42
ample Service	27	5.6 Data Transfer	43
5.2.3 UserService	30	5.7 Deployment	43
5.2.4 CategoryService.....	30	5.8 Documentation.....	44
5.2.5 PageService	30	6 Testing	45
5.2.6 PageContentService.....	30	6.1 Use Case Validation	45
5.2.7 ModelService	31	6.2 Cooperation with Editors	45
5.2.8 ImageService	31	6.3 User Feedback.....	45
5.2.9 LabelService.....	31	7 Conclusion	47
5.2.10 LabelContentService	31	References	48
5.3 Backing Beans	31		
5.3.1 BasicController	32		
5.3.2 LanguageController	32		
5.3.3 LoginController	33		
5.3.4 PathController	33		
5.3.5 CategoryController	33		
5.3.6 SearchController	33		
5.3.7 PageController	33		
5.3.8 ModelController	34		
5.3.9 ImageController.....	34		
5.3.10 Converters.....	34		
5.4 View – Facelets and HTML ...	34		
5.4.1 Templating	35		
5.4.2 Page URLs	35		
5.4.3 Style – CSS	35		
5.4.4 Ajax	36		
5.4.5 Localization	36		
5.4.6 Language Component			
– Example Component ..	36		
5.4.7 Other Common Com-			
ponents	37		
5.4.8 Page Specific Compo-			
nents.....	37		
5.5 Model Viewer	38		
5.5.1 Basics, Rendering	38		
5.5.2 Materials and Lighting ..	38		
5.5.3 3D Models.....	39		
5.5.4 Labels	39		
5.5.5 Label Selection.....	40		
5.5.6 Camera Controls.....	40		
5.5.7 UI Components	41		
5.5.8 Window Resize.....	41		
5.5.9 Editing	41		
5.5.10 Mouse Mode and			
Mouse Events	42		

Tables / Figures

2.1. Identified Flaws	5
3.1. WebGL vs. Flash rendering frequency	9
3.2. IEEE Spectrum - Top Web Development Languages	11
4.1. Use Case Diagram	13
4.2. ER Diagram	20
4.3. Preliminary Design of the Application	22
4.4. Mature Design of the Server Side	24
6.1. Use Case Diagram - Valida- tion	46

Chapter 1

Introduction

The topic of this thesis is the development of the Atlas of Human Bones, an online database of bones including 3D models with labels and web pages with additional information. The Atlas will serve as a source of information and a study material, primarily for students of the Third Faculty of Medicine at Charles University in Prague.

1.1 Project Background

The Third Faculty of Medicine at Charles University in Prague is a successful, established educational institution. Its Department of Anatomy, as many others before, have realized the great potential of computer technologies in education and started numerous projects to transfer knowledge from traditional printed sources to a much more flexible online domain. One of these projects was an online atlas of human bones.

This project was initiated in 2012 by doc. MUDr. Václav Báča, Ph.D., at the time a faculty member of the Department of Anatomy. The project aimed to create an online 3D database of human bones, which would be an exhaustive yet illustrative source of information for medical students. The goal was not only to display 3D models, but also to provide numerous labels with descriptions of relevant structures of each bone.

I developed the first, rather imperfect (see 2.5) version of the Atlas in 2013 and the Third Faculty of Medicine provided the content.

Working on this project as a part of my master's thesis finally gave me the necessary amount of time to give the Atlas a long due redesign.

1.2 Thesis Content

In this thesis we first define the features of the Atlas, compare the concept with existing solutions to see if it brings anything new and analyze the existing version to see if a complete redesign is necessary. We research possible technologies and choose those best suited for various parts of our application. We design a sustainable web application whose content is not only accessible to medical students, but also editable by users without a technical background. We describe the actual implementation of the design, acquisition of content and deployment to a live server. Finally we collect feedback from potential users to see what needs to be improved in future versions.

Chapter 2

Problem Analysis

In this chapter we define the requirements and features of the application, compare the basic concept to existing solutions including the previous version of the Atlas and evaluate possible contributions of the project.

2.1 Role Definition

The application will support three basic user roles:

- **User** – Any visitor of the website. Users can browse all public content but not alter it in any way.
- **Editor** – An authenticated user responsible for uploading and editing the content.
- **Administrator** – An authenticated user responsible for managing user accounts.

2.2 Requirements

2.2.1 Functional Requirements – User

3D Viewer with following features:

- Display 3D models of bones
- Display labels for significant parts of the bones
- Display detailed description of a part when label is selected
- 3 display modes for labels: Full labels / Labels with hidden names / No labels
- Adjustable view (move, rotate, zoom)

Website with following features:

- Navigation to specific bones using a hierarchy of categories
- Navigation using a **Search** function
- Web page for each bone with a 3D model, relevant images, text and references

2.2.2 Functional Requirements – Editor

- Login into editorial system
- Manually add, edit and delete pages (bones)
- Upload 3D model files
- Upload image files
- Add uploaded 3D models and images to pages
- Manually add, edit and delete textual information related to each bone
- Manually add, edit and delete labels and descriptions of 3D models

■ 2.2.3 Functional Requirements – Administrator

- Login into account management system
- Create, edit or delete editor accounts

■ 2.2.4 Non-functional Requirements

- Browser-based online application
- Multiplatform
- Support for bilingual content (Czech, English)

■ 2.3 Target Group

The application is targeted at students of the Third Faculty of Medicine at Charles University in Prague. Other users are to be expected, but the focus should remain on medical students.

Medical students are young, non-technical users. However, high working knowledge with a web browser can be expected.

■ 2.4 Existing Solutions and Alternatives

The following list is by no means complete. These are just several examples of the most relevant solutions and alternatives found online.

- Skelet 3D (available at [1]) is the previous version of our project, further discussed in the next section. Our intention is to redesign and upgrade this solution.
- Skeletopedia (available at [2]) seems to be the independent solution closest to our specification. It provides interactive 3D models of bones in browser environment as well as simple labeling of certain parts, unfortunately not in great detail. The contribution of our project in comparison to this solution should be more detailed and far more numerous labeling of the models as well as further information about each label and each bone in general.
- Zygote Body, Anatronica (available at [3–4]) and many other systems include 3D models of the human skeleton. However, they provide little to no additional information besides the name of each bone.
- 3D Science provides extremely detailed 3D models (available at [5]). However, these are not free, most likely not suitable for online display and most importantly provide no additional information.
- Palacký University in Olomouc provides information on most bones in their online human anatomy as well as photography of human bones, sometimes including labels with short descriptions (available at [6]). The information, however, seems incomplete. Moreover, there is no 3D material available, which is the key feature of our project.
- Wikipedia provides rather detailed information on human bones (available at [7]). English version includes simple 3D animations of most bones, but it doesn't provide any connection of 3D models with labels and descriptions, which is the intended contribution of our project.

Overall, each existing solution provides some interesting features, but none of them offers everything our project is aiming for. We will incorporate the best of each existing solution and add something extra: Interactive 3D models comparable to Zygote/Anatronica, labels more detailed than those of Skeletopedia, information comparable to Wikipedia but focused on medical students and up-to-date content in Czech.

■ 2.5 Analysis of the Current Version of the Atlas

The Atlas project is not beginning with this thesis. It was initiated in 2012 and most of the development of the existing version happened in 2013.

The application was developed on the fly and went through a series of overhauls without proper planning. Although it works, it suffers from a variety of flaws. From a developer's point of view, the internal structure is unorganized and undocumented. While it might have been sufficient, the lack of order and proper organization makes sustainability and further development a complicated issue.

The current version is a publicly available website running on the servers of the Third Faculty of Medicine, Charles University [1] *Note: In December 2014 that version was replaced by the product of this project and is no longer available.*

It consists of a list of bones sorted in a hierarchy of groups based on parts of the human body, a 3D viewer of models of bones and a very simple editing page which isn't public. An as-of-yet unreleased bilingual version allows editing in Czech and English (as opposed to Czech only).

■ 2.5.1 Bone Selector

The list of available bones is a Flash application written in ActionScript 3 that allows selection of a model and filtering using a hierarchy of predefined groups. It does what it's supposed to do, although the layout might not be the most intuitive and there are no additional features such as "search". The loading time is longer than expected due to inefficient use of XML files that define groups and bones.

■ 2.5.2 3D Viewer

The model viewer is another Flash (ActionScript 3) application using a simple 3D engine called Sandy with two basic modes: view and edit. It allows the user to manipulate view freely, although not in the most intuitive way, utilizing only one mouse button. It allows users to display or hide labels and pins appropriately.

However, it suffers from graphical errors, most notably seeing labels through a bone while they should be behind it and vice versa. This is caused by Z-sorting algorithms used in Sandy 3D engine, which are too simple for the task.

It is also missing some minor tweaks such as adjusting the label width to fit the length of the text.

The edit mode is working, although its graphical design is poor.

The rendering speed of the application is unsatisfactory, greatly limiting the use on mobile devices and the quality of models. Some of the more complex models are reaching low FPS even on average desktop computers, while being virtually unusable on older machines and mobile devices. A tradeoff between model complexity and rendering speed is to be expected, but current application's performance is nevertheless underwhelming. The technology used doesn't provide easy solutions to some of the identified flaws (see 2.1).

2.5.3 PHP Parts

The Flash application is nested in a simple PHP page. The web page which is used to upload new models and enter the editing mode is implemented as a series of simple PHP scripts. Other PHP scripts are called by the Flash application to handle XML files on the server.

2.5.4 Editing

There are two versions of the application, a public version with editing disabled and a separate version for editing only, whose location is not known to public, but which is not protected in any other way. The live version is updated manually by transferring data files from the editable version. Introduction of a login system would allow better sustainability through direct editing by privileged users as opposed to transfers by the website administrator.

2.5.5 Identified Flaws

Identified Flaw	Affects	Importance	Suggested Solution	Difficulty
No images / text	Content value	High	Add editorial system	High
No authentication	Safety, sustainability	High	Add login system	Medium
Hardware demands	Availability, details	High	Change of 3D engine	High
Graphical errors	User experience	Medium	Change of 3D engine	High
Imperfect GUI	User experience	Medium	Rework GUI	Medium
Bad internal design	Future development	Medium	Proper design	Medium
No documentation	Future development	Medium	Add documentation	Low
No search	User experience	Low	Implement search	Low
Slow loading of lists	User experience	Low	Use DB over XML	Low

Table 2.1. Identified flaws of the current version of the Atlas. Author's subjective evaluation of their impact and solutions.

2.5.6 Conclusion

Overall, the technologies used seem to be inefficient and partly outdated. A complete reworking of the system seems to be the best solution considering the extent of individual improvements, especially the change of 3D engine and introduction of an editorial system.

The models and labels entered by medical students into the existing version should be valid and reusable in the new implementation.

2.6 Project's Contributions

The main goal is to help medical students acquire knowledge of human bones. There are numerous sources available, ranging from lectures and printed textbooks to aforementioned online solutions. Our application cannot compete with the experience and insights of lecturers and it is not likely to replace textbooks because of the sheer volume of information included in those. But it can be a great study material, available anytime, hopefully better than other online sources.

None of the known sources (other than the previous iteration of our Atlas) provide what we hope to achieve in this project: Exhaustive information linked to illustrative 3D models. There are 3D models and there is information. This project's contribution

is linking the two in an interactive manner, allowing students easy navigation and a mental link between spatial and textual information.

This was attempted in the current version of the Atlas in a clumsy and imperfect manner. This version strives to be a more professional solution of the problem without all the flaws of the last version.

The aim is to develop the application in an organized, orderly and documented way, using the best available technologies and methods chosen after careful consideration. We need to avoid the glitches and imperfections of the previous iteration, to make the display module more efficient, the interface more user-friendly. We need sustainability, most notably a proper editorial system with authentication to allow the application to grow and fill up with useful information.

Moreover, there will be the new feature of web pages with additional information. They will contain any information seen fit by the editors, giving this project the potential to be a truly exhaustive source of knowledge regarding human bones, ranging from anatomy to pathology. Other new features might be added as well, given a time reserve.

Chapter 3

Relevant Technology

This chapter compares technologies which can be used to build our application. We then choose the candidates best suited for our task.

3.1 3D Technology for the Web

There are two main directions in development of hardware accelerated 3D for the web: Rendering directly in the browser with HTML5 and against it the traditional plug-in based approach.

Certain technologies allow the application to be a built-in part of an HTML5 web. The main advantage is that users do not need to install any additional software. The application is fully integrated into the web page and its execution is managed by the browser, which might result in slight variations across browsers. In case of 3D technologies, even the use of underlying graphics API depends on the environment, such as browsers using ANGLE with WebGL to utilize DirectX over OpenGL on Windows.

The advantage of plug-ins is that the application will look the same on every device and doesn't need to be tweaked for different browsers. The reason behind that is that the application is executed by the plug-in, not the browser. The disadvantage is that users have to install a plug-in, which is usually an inconvenience and sometimes a real problem, especially on mobile devices. Also, the application is not fully integrated in the web page, which might result in behavior inconsistent with the rest of the web page.

3.1.1 CSS 3D Transforms

CSS3 can apply 3D transforms to elements directly without using HTML5 Canvas. This can be used to create 3D objects and animations.

All geometry is created by transforming rectangular elements, which is very restrictive compared to triangular faces created between vertices. This obstacle can be bypassed by using alpha textures and 3D transforms to create arbitrary shapes. Although it is possible to get creative and assemble complex 3D scenes this way, I feel like this was not the intended purpose and there are tools better suited for the task.

I was not able to find examples demonstrating the use of CSS 3D transforms for displaying complex models comparable to those in our project. Even much simpler scenes often contained graphical errors and did not run smoothly in either Internet Explorer 11 or Firefox 32. The only advantage is superior availability.

3.1.2 WebGL

WebGL (Web Graphics Library) is a JavaScript API for development of interactive 3D scenes for the web. It uses underlying low-level graphics API, typically OpenGL ES 2.0, to make full use of GPU acceleration. It is a royalty-free, cross platform standard maintained by the non-profit Khronos Group. There are frameworks to simplify the use of WebGL, which itself can be considered a low-level API.

WebGL uses HTML5 Canvas to display its content and does not require the use of a plug-in. It will work on any browser with WebGL support without additional software (although it might require appropriate GPU drivers).

The required features such as advanced mouse controls and 3D acceleration are all present and comparable to those of Flash with possible minor benefits (such as consistency of user controls throughout the page).

WebGL is a new technology compared to Flash and other plug-ins. WebGL 1.0 standard was issued in 2011 and in the following years, support by vendors and developers has been growing. Firefox, Google Chrome and Safari have supported WebGL for some time as shown at [8]. Microsoft support starts with Internet Explorer 11 and according to [9] “WebGL is available on all IE11 devices”. Apple announced support on their new iOS 8 (related article at [10]). Default Android browser doesn’t support it, but Firefox and Chrome for Android do.

■ 3.1.3 Flash

Adobe Flash is a multimedia and software platform widely used throughout Internet.

Flash uses vector graphics, static or animated, supports streaming of videos, and offers other multimedia related features. The language typically associated with online Flash is ActionScript, although Haxe can be compiled into Flash applications as well. Flash is a proprietary platform of Adobe. Freeware editing software exists, but arguably inferior to licensed Adobe Flash Professional.

Flash requires a plug-in (Flash Player) to run in web browsers, which is enough of a reason to reject many competing technologies. However, the Flash Player is so prevalent today that most users have it installed regardless of our application. Adobe claimed to have 99% penetration on desktop computers in 2011 [11]. That makes the necessity of a plug-in a much smaller issue for our project.

Since version 11.2 (2012), Flash Player allows advanced mouse control such as scrolling or right click [12], which was one of the main features missing in previous versions. Nevertheless, user input might be clashing with the rest of the HTML page, resulting in poor user experience.

Flash Player 11 also introduced Scene3D, an API for development of hardware accelerated 3D content. Scene3D is one of the major candidates for our project’s 3D API. Previous Flash technologies relied on CPU rendering and were significantly slower than GPU accelerated alternatives. Such slower technologies were used in previous implementations of the Atlas.

For several years, numerous writers and developers have claimed that Flash is insecure, a dying technology and a developmental dead end (examples at [13–14]). Despite these voices, Flash is still alive in 2014 and here to stay for some time. Although still widely used, the future of Flash is not all bright. According to statistics at builtwith.com [15], the usage of Flash on major websites has been on the decline lately.

According to [16] Adobe Flash is currently available on most desktop operating systems (Windows, OS X, Linux, Solaris), although the development for Linux and Solaris has been discontinued since Flash Player 11.2 (outside of Google Chrome).

Flash Player for mobile browsers availability is controversial. It is not available on iOS devices (iPhone, iPad...), but it was officially available on Android 2.2-4.0 and BlackBerry (Tablet OS, BB10). However, in 2011 Adobe announced at [17]: “We will no longer continue to develop Flash Player in the browser to work with new mobile device configurations” and admitted that growing support makes “HTML5 the best solution for creating and deploying content in the browser across mobile platforms”. A blog post at

[18] confirms that Flash will not be installed on new Android devices and Flash Player for Android will no longer be updated. Apple's (and formerly Steve Job's) attitude and lack of support (more information at [19]) also speaks against mobile Flash.

3.1.4 Other Plug-in Technologies

There are several other plug-in based systems worth mentioning. All following examples suffer from the necessity of installing a plug-in which is not likely to be pre-installed.

Unity [20] and ShiVa3D [21] are fine examples of multiplatform 3D engines that allow development for desktops, mobile devices and browsers. However, the browser version is not supported on mobile devices, forcing the development of native applications for each mobile operating system (often at a substantial price). That is beyond the scope of this project and browser-based solutions for all platforms are preferred.

Markup languages such as X3D (and its predecessor, VRML) used to be strictly plug-in based, but lately benefit from GPU acceleration in the form of WebGL through integration models such as X3DOM. When using X3D now, there is little reason to choose a plug-in viewer over WebGL rendering. The support for WebGL (see below) is comparable to that of plug-ins and there is no need of additional installation.

3.1.5 Performance Test 1: Cubes (WebGL and Flash)

I carried out a simple performance test to compare the rendering speed of equivalent Flash and WebGL applications.

A demo application (available at [22]) displaying a number of semi-transparent cubes was used, implemented once using Flash Scene3D and once using WebGL. The experiment was run on a Dell Vostro 3460 machine.

The number of cubes was gradually increased while noting FPS values for current number of cubes. There were minor fluctuations in the FPS and recorded values represent estimated medians over a period of several seconds.

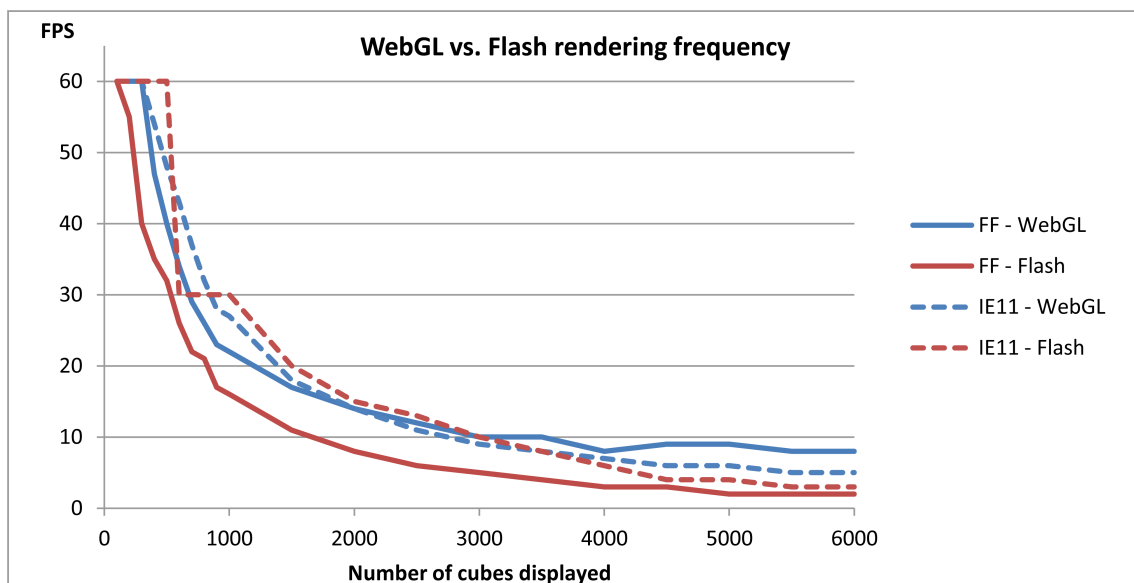


Figure 3.1. Performance of WebGL (blue) and Flash (red) when rendering a demo application in Firefox 32 (solid) and Internet Explorer 11 (dashed). Frames per second on vertical axis, number of displayed translucent elements on horizontal axis.

It is apparent from the results shown in figure 3.1 that the performance is greatly dependent on the browser.

WebGL in Firefox is by far the fastest combination in very complex scenes. Flash in Firefox is always slower than any other tested combination. In Internet Explorer, both Flash and WebGL show comparable performance with WebGL gaining in more complex scenes.

Overall, WebGL did better in this performance test, especially with a large number of displayed elements.

It is important to mention that this is an illustrative example rather than a rigorous experiment. The implementation used two different frameworks (Away 3D 4 for Flash, Three.js for Web GL) which might or might not be a source of bias. The performance might also be affected by hardware usage fluctuations in the testing machine caused by other running applications and background processes.

■ 3.1.6 Performance Test 2: Aquarium (WebGL)

To further verify that WebGL can handle complex geometry at reasonable frame rates, a public WebGL demo “Aquarium” (available at [23]) was run at maximum complexity. It displays 4000 models of fish using normal maps, reflections and other effects. The scene geometry easily exceeds 100,000 vertices. Testing on Dell Vostro 3460, the scene was sufficiently smooth, displaying at 12 FPS in Internet Explorer 11 and 14 FPS in Firefox 32. For comparison, current implementation of the Atlas shows significantly worse performance when displaying models of 10,000 vertices.

■ 3.1.7 Frameworks (WebGL)

WebGL is a low-level graphics API. Though it is possible to work with it directly, it might be more convenient to use a higher level framework. A list of many available frameworks has been assembled at [24], although not all of them are relevant for our project.

Upon brief research, I came to the conclusion that many of them were interesting solutions, but very few had a large and active community. In fact, out of those frameworks relevant to our project only *three.js* (available at [25]) showed actual signs of active use throughout Internet. While several of the frameworks have tutorials on their websites, only *three.js* can boast a number of external tutorials as well. To verify my assumption, I searched for the frameworks on *StackOverflow* [26], arguably the biggest question & answer site for programmers. The fact that *three.js* was tagged in 4775 questions while no other framework was tagged more than 60 times confirms that *three.js* is likely the most prevalent.

The API of *three.js* seems suitable for our cause. It provides easy access to scenes, objects, cameras, materials, lightning, shaders and more, making the use of WebGL a lot less complex for the developer. The API is documented online.

Considering all this, *three.js* seems to be the perfect framework for our project and there is no need to seek for a novelty solution.

■ 3.1.8 Conclusion

CSS3 is an interesting approach with great availability, which is however not very well suited for a complex 3D project. The technology clearly wasn’t meant to render detailed 3D models.

The other technologies offer us what we need: GPU acceleration, sufficient interactivity, quality of user controls and an API suitable for displaying 3D scenes (sometimes

using additional frameworks). All options seem to be perfectly viable for our purpose in terms of available features, so the main points to consider are performance and availability.

Availability is probably the biggest concern. Most plug-in based solutions require an inconvenient installation and do not run in mobile browsers. That is a major drawback that hurts availability and a reason not to choose such solutions. The exception is Flash which is often pre-installed and has partial support in mobile browsers. The other suitable candidate left is WebGL.

Both Flash and WebGL are available on most if not all up-to-date desktop browsers. Neither is available on every mobile device. The trend seems to be obvious though, the use of Flash is on a decline and Flash development shouldn't focus on mobile browsing. On the other hand, more and more companies (including Microsoft and Apple) declare and implement support for WebGL. That is the main reason why WebGL seems to be the better candidate: Its future seems bright, which cannot be said about Flash.

As for performance, simple tests on a laptop favored WebGL over Flash, although the results depend greatly on chosen browser. WebGL also proved its ability to render very complex scenes at reasonable frame rates in an "Aquarium" demo.

After careful consideration, WebGL was chosen as the preferred technology for the 3D model viewer, even though it is not universally available. To simplify the implementation process, *three.js* framework will be used as a higher level API.

3.2 Server-Side Language

There are many languages to choose from when developing the server side of a web application.

3.2.1 IEEE Spectrum Ranking

To help evaluate the quality of the many languages, I consulted the results of a recent IEEE Spectrum ranking (available at [27]). It takes into consideration *Google* search results, *Google Trends* data, *StackOverflow* questions, demand for jobs on several job sites, *IEEE Xplore* journal articles and more. The top 10 results for web development are shown in figure 3.2 with Java, Python and C# taking the top (in this order).

Language Rank	Types	Spectrum Ranking
1. Java	🌐 📱 🖥	100.0
2. Python	🌐 🖥	93.4
3. C#	🌐 📱 🖥	92.3
4. PHP	🌐	84.7
5. Javascript	🌐 📱	84.4
6. Ruby	🌐	78.8
7. PERL	🌐 🖥	70.3
8. HTML	🌐	65.3
9. Scala	🌐 📱	63.0
10. Go	🌐 🖥	60.5

Figure 3.2. Top 10 web development languages, ranked by IEEE Spectrum in 2014 (taken from [27]).

3.2.2 Personal Experience

An important part of choosing the right language is my prior experience: I was introduced to PHP several years ago and have not used it much since. Later I worked on a desktop project in .NET C#. Throughout university, a lot of my education involved Java, although I never used it in non-academic projects or web applications.

PHP is subject to criticism for its frequent API changes, loose typing, security flaws or poor backwards compatibility. My brief personal experience made it seem inconsistent, full of small surprises and traps for an inexperienced user. A more elaborate criticism by a more experienced developer is available at [28].

On the other hand, Java and C# felt like professional tools and I did not mind working with either, although I preferred Java for subjective reasons. An objective reason would be that it is multi-platform and open-source with several free IDEs to choose from.

However, I have not developed a web application in either, so I looked for educated opinions online. Most comparisons and reviews agree that Java and .NET for web are comparable and the choice is mostly preferential (examples at [29–30]).

3.2.3 Conclusion

Considering that Java was ranked the best language for web development by IEEE Spectrum and at the same time is my preferred language from personal experience, the site will be developed in Java.

3.3 Server-side Framework

3.3.1 JavaServer Faces

JavaServer Faces [31] (JSF) is a component based Model-View-Controller (MVC) framework which was specifically created for Java web applications. It has many great features, such as binding of business objects to generated view (HTML pages), composite components to build pages, easy to use Ajax calls and much more. It has a large user base and high-quality support on *StackOverflow* [26].

JSF is a formalized Java EE standard and will be used as the main framework for our application. The alternative of building from scratch or with simple frameworks is too tedious and there seems to be little reason not to use such a framework.

Chapter 4

Application Design

In this chapter we describe the application's design, starting from use case scenarios and making our way to application architecture and classes.

4.1 Use Cases

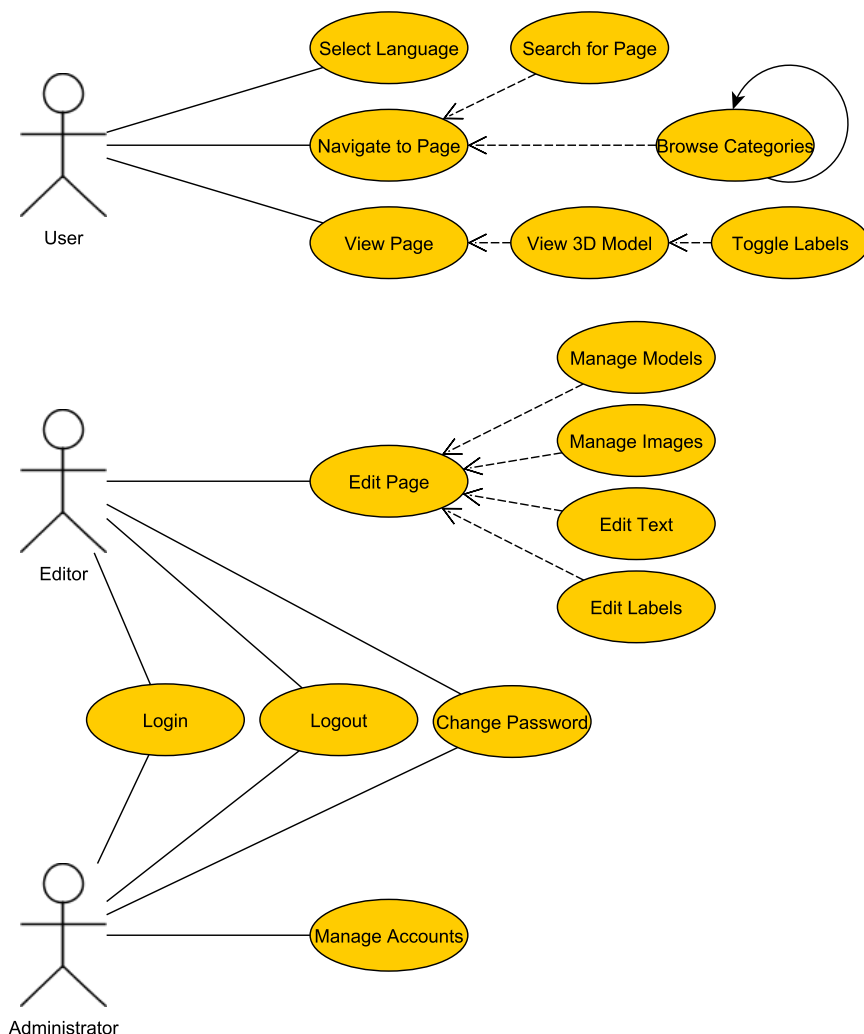


Figure 4.1. Simple use case diagram of the Atlas

This project's design phase begins with a simple use case diagram to better visualize our goals. It can be seen in diagram 4.1. It shows basic actions that can be taken by

the users in different roles: Unregistered users can browse all the content of the website, while registered editors are in charge of providing the content. Administrators get to manage editors' accounts.

Individual more detailed use case scenarios are following.

■ 4.1.1 User: Select Language

Basic flow of events:

1. User clicks a flag representing his preferred language
2. Client informs server about changed preferences for this session
3. Page is refreshed and displayed in selected language

■ 4.1.2 User: Navigate to Page

Sub-scenarios:

- Go to a subcategory:
 1. User selects a subcategory from a list of items
 2. Content of selected subcategory is displayed
- Return to an ancestor category:
 1. User selects a category in the “path” to current category / page
 2. Content of selected ancestor category is displayed
- Return to main page:
 1. User clicks the main logo
 2. Main page is displayed

Basic flow of events:

1. User selects a main category
2. Content of the category is displayed
3. User goes to a subcategory
4. Step 3 is repeated until the category containing desired page is reached
5. User selects desired page from a list of items
6. Desired page is displayed

Alternative flow: 1a. User is already browsing a category

1.
 - a) User continues by step 3
 - b) User returns to an ancestor category
 - c) User returns to main page
2. User continues by step 3

Alternative flow: 1b. User uses search function instead

1. User enters a search term
2. List of matching pages and categories is displayed
3. User selects desired item
4. Desired page is displayed

Alternative flow: 3a. User goes to wrong subcategory

- a) User returns to an ancestor category
- b) User returns to main page

■ 4.1.3 User: View Page

Basic flow of events:

1. User clicks a link to a content page
2. Content of the page is displayed, 3D model is eventually loaded
3. User scrolls through the page, views content
4. User interacts with 3D scene

■ 4.1.4 User: View Model

Starting condition: User is viewing a page with 3D content

Sub-scenarios:

- Rotating view:
 1. User drags with left mouse button inside 3D area
 2. 3D scene responds in real time and rotates
- Panning view:
 1. User drags with right mouse button inside 3D area
 2. 3D scene responds in real time and pans
- Zooming:
 1. User rolls mouse wheel inside 3D area
 2. 3D scene responds in real time and zooms
- Selecting labels:
 1. User clicks a label
 2. Details of the label are shown
 3. User clicks selected label again
 4. Details of the label are hidden
- Switching label mode:
 1. User selects mode from a dropdown menu
 2. Labels are shown, hidden or redrawn depending on mode

■ 4.1.5 Registered User: Login

Starting condition: User is currently logged out

Basic flow:

1. User enters username and password, clicks login
2. System verifies user's account
3. User is logged in, page is reloaded

Alternative flow: 2a. Username and password do not match an account

1. User is not logged in, informative message is displayed

■ 4.1.6 Registered User: Logout

Starting condition: User is currently logged in

Basic flow:

1. User clicks logout
2. User is logged out, page is reloaded

■ 4.1.7 Registered User: Change Password

Starting condition: User is currently logged in

Basic flow:

1. User clicks a link to password change
2. User enters old and new password and confirms
3. System verifies the input
4. Password is changed

Alternative flow: 3a. Input is incorrect

1. Password is not changed, informative message is displayed

■ 4.1.8 Editor: Create page

Starting condition: Editor is currently logged in

Basic flow:

1. Editor navigates to a category
2. Editor clicks “Add new page”
3. System verifies user’s rights to edit, continues if they match
(Abort operation and display warning otherwise)
4. New unpublished blank page is created in the category

■ 4.1.9 Editor: Edit Page

Starting condition: Editor is currently logged in

Basic flow:

1. Editor navigates to a category
2. Editor clicks “Edit page” at selected page
3. Editable page with selected page’s content is displayed
4. Editor changes page’s basic properties
5. Editor clicks “Add component” to add a component of chosen type
6. Component is added to content
7. Editor changes component’s properties
 - a) Text component: Text content
 - b) Headline component: Text content
 - c) Image component: Image from database, description
 - d) Model component: 3D Model from database, description
 - e) *Common for all:* Position in page
8. Repeat from step 5 until content is ready
9. Editor clicks “Saves changes”
10. System verifies user’s rights to edit, continues if they match
(Abort operation and display warning otherwise)
11. Changes are saved

Alternative flow: 5a. Editor works with existing components

1. Continue by step 7

Alternative flow: 5b. Editor clicks “Delete” at selected component

1. Component is removed
2. Continue by step 8

■ 4.1.10 Editor: Delete Page

Starting condition: Editor is currently logged in

Basic flow:

1. Editor navigates to a category
2. Editor clicks “Delete” at selected page
3. System asks for confirmation
4. Editor confirms deletion
(Abort operation otherwise)
5. System verifies user’s rights, continues if they match
(Abort operation and display warning otherwise)
6. Page is deleted

■ 4.1.11 Editor: Managing Models

Sub-scenarios:

■ Enter Model Manager

Starting condition: Editor is currently logged in

1. Editor clicks Model Manager
2. Model Manager is displayed

■ Upload model in Model Manager

1. Editor selects a file from his computer
2. Editor enters a name to identify the model in the future
3. Editor hits upload
4. System verifies user rights, continues if they match
(Abort operation and display warning otherwise)
5. File is uploaded into the system

■ Delete model in Model Manager

1. Editor clicks “Delete” next to an existing model
2. System verifies user rights, continues if they match
(Abort operation and display warning otherwise)
3. System checks if model is used in a page, continues if not
(Abort operation and display warning otherwise)
4. File is deleted from the system

■ 4.1.12 Editor: Manage Images

Images are managed the same way models are, so the scenarios would be analogical to “Manage Models”.

■ 4.1.13 Editor: Manage Model’s Labels

Starting condition: Editor is logged in and viewing a page with 3D content

Sub-scenarios:

■ Create new label

1. Editor clicks “New label”
2. Editor is prompted to choose a location

3. Editor chooses a location by clicking on the 3D model's surface
4. New blank label is created in that location and selected

■ Edit a label

1. Editor selects a label
2. Editor changes label's properties
 - a) Title by overwriting old title in its textbox
 - b) Text content by overwriting old content in its textbox
 - c) Position of "nametag" by dragging it with left mouse button

■ Delete a label

1. Editor selects a label
2. Editor clicks "Delete label"
3. System asks for confirmation
4. Editor confirms deletion
(Abort operation otherwise)
5. Label is deleted

■ Save changes

1. Editor clicks "Save changes"
2. System verifies user rights, continues if they match
(Abort operation and display warning otherwise)
3. Changes are saved

■ 4.1.14 Administrator: Manage Users

Starting condition: Administrator is currently logged in

Sub-scenarios:

■ Enter User Manager

1. Administrator clicks User Manager
2. User Manager is displayed

■ Create new user in User Manager

1. Administrator fills in name, username, default password, role
2. Administrator clicks "Add user"
3. System verifies user rights, continues if they match
(Abort operation and display warning otherwise)
4. New user is added to the system

■ Edit user in User Manager

1. Administrator changes name, password or role of an existing user
2. Administrator clicks "Save changes"
3. System verifies user rights, continues if they match
(Abort operation and display warning otherwise)
4. Changes are saved

■ Delete user in User Manager

1. Administrator clicks "Delete" next to an existing user

2. System asks for confirmation
3. Administrator confirms deletion
(Abort operation otherwise)
4. System verifies user rights, continues if they match
(Abort operation and display warning otherwise)
5. User is deleted from the system

4.2 Graphical User Interface

A simple mockup of a possible graphical user interface (GUI) was created in early stages of the design. It was a series of simplified images and it focused on “what features” rather than actual graphical design. It helped define what we want to implement in order to cover our use cases.

The mockup is not displayed here because of space constraints. Images from final application will be shown in an appendix.

4.3 Business Objects

The application is controlled by numerous business objects, also known as Entities. These objects represent mostly the content of the application, but also miscellaneous entities such as users and languages.

All our business objects are persistent in the sense that they should be available in all instances of the application, survive server restarts and so on. This means we will be persisting these object and the obvious choice for persistence is a relational database for most records and a file system for binary files such as 3D models and images.

It is a common theme that localizable “real life” objects are represented by two Entities – one contains properties shared by all language variants such as ID, parent category or Latin name. The other Entity contains localized properties for a given language such as title or text content.

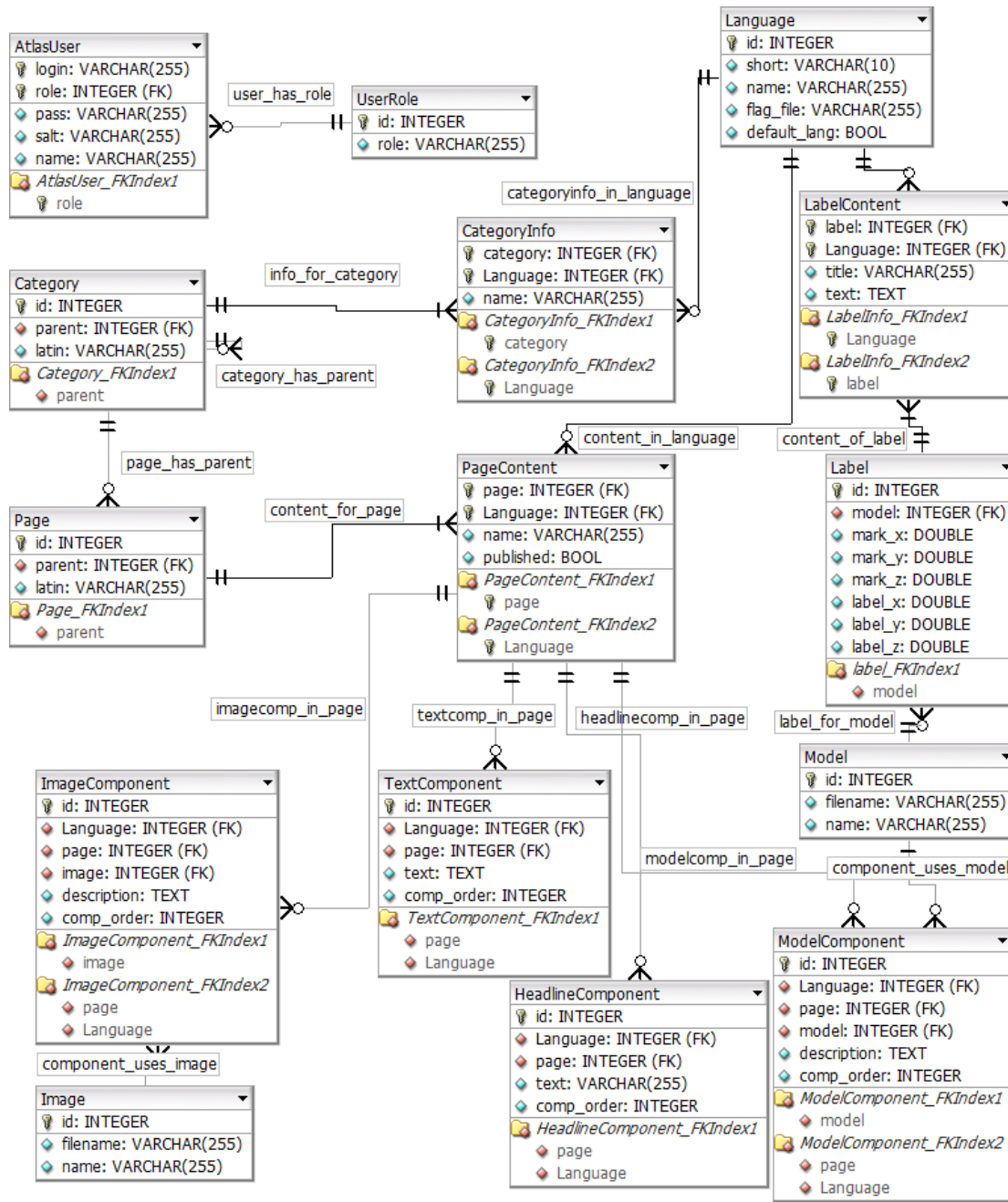


Figure 4.2. ER diagram of application's Entities in crow's foot notation.

These are the Entities used in our application as shown in ER diagram 4.2:

- **User** represents a registered user of the application (editor, administrator) and provides properties relevant for authentication. It also has a **UserRole** which determines its rights and privileges.
- **Language** represents an available language for the website and its content.
- **Category** represents a group of pages and other categories (presumably a body part). It can point to its parent category, creating a tree of categories.
- **CategoryInfo** represents localized properties of a Category for a given language.
- **Page** represents a content page (presumably a bone in the Atlas).

- **PageContent** represents the localized content of a Page. It consists of any number of “Content Components”
- **TextComponent, HeaderComponent, ImageComponent and ModelComponent** are “Content Components”, localized units of content
- **Image** represents an image file (used by ImageComponent)
- **Model** represents a 3D model file (used by ModelComponent)
- **Label** represents a label pointing out an important location in a Model
- **LabelContent** represents the localized content of a Label

All primary keys as well as foreign keys in the database are indexed to allow more efficient searching.

4.4 Overall Architecture and Classes

The preliminary design of the application is shown in diagram 4.3. Individual components are described below.

4.4.1 Client Side

The client side (leftmost grey box in diagram 4.3) represents the View in our MVC architecture.

It will consist of XHTML pages generated by JSF Facelets. JSF supports reusable components, so we can create a template file and fill it with common components such as login bar, language bar or navigation bar as well as custom content for the page.

The 3D viewer will be a JavaScript file attached to one of the content components.

4.4.2 Servlet

A Servlet (labeled as “JSF Binding” in diagram 4.3) represent the Controller in our MVC architecture.

A server communicates with a client by generating HTTP responses based on client’s HTTP requests. In Java, the unit responsible for this communication is called a Servlet. JSF takes care of this for us by implementing a FacesServlet.

The FacesServlet intercepts HTTP requests and generates the proper HTTP responses to present the correct View (XHTML page including data from Model).

4.4.3 Managed Beans

Managed beans (first darker grey block inside the central block in diagram 4.3) also belong to the Controller, although the “real” controller is indeed a FacesServlet.

Managed beans are Java classes registered with the JSF framework. This allows binding between the bean’s properties and a View. An example of this would be always displaying current user’s name in the login bar, while current user is a bound property of LoginManager (a session scoped managed bean).

The link between Model and View that a managed bean provides works both ways. It fetches entities for the View to display and it notifies the Model when changes should be made to it (initiated by user interaction with the View).

These managed beans are also called “backing beans” and each of them should be associated with a View page or component (rather than a business entity).

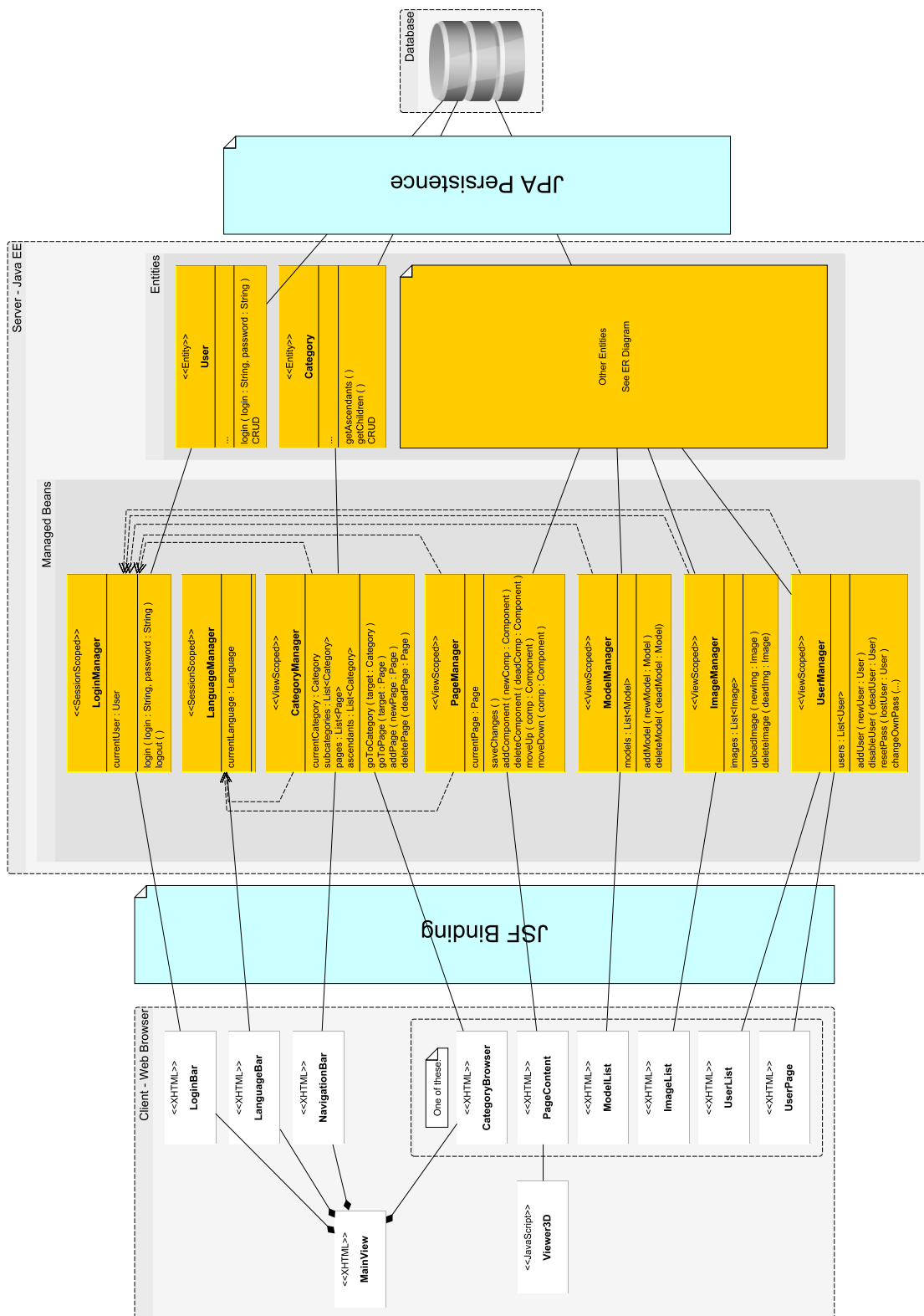


Figure 4.3. Preliminary design of the application.

4.4.4 Entities

Entities (second darker grey block inside the central block in diagram 4.3) are the business objects of our application, the Model. These objects include users, pages, 3D models, labels, languages and many others.

They are Java classes implementing all properties of the object (such as name, user-name, password... for user) and providing methods to Create, Read, Update and Delete (CRUD) their respective objects in the database.

These data-representing classes can travel through many layers of the application, from persistence to View and back.

■ 4.4.5 Persistence

Persistence (rightmost part in diagram 4.3) will be realized mostly by a database. The default Java DB (formerly Derby) will be sufficient for our needs as no non-standard requirements are known and the expected workload is small. If necessary, the database engine can be easily switched later.

The database connections, transactions and everything related will be left to Java Persistence API (JPA) as opposed to manually using JDBC. JPA with its EntityManager is a great (and ready to go) way to ensure that everything works and nothing clashed in our multi-user environment. What is more, entities properly mapped to database tables save us the hassle of writing SQL queries manually.

■ 4.5 Architecture and Classes - Mature

During the development of our application, it became apparent that while the original architecture might be sufficient, there is a lot of space for improvement.

The overall design remained quite similar to the initial plan. Client side, managed beans and JPA persistence were mostly unchanged, subject to minor updates.

The most notable changes happened between Controller and Model, where a new service layer was introduced to better separate model and logic (further discussed in 4.5.1). A more mature version of the Java server's class diagram is shown in diagram 4.4.

■ 4.5.1 Service Layer

The original design used Entities which provided their own CRUD methods. That way Controller classes (previously called Managers) would call CRUD methods directly on the instances of business objects.

In the mature design, Entities are kept as slim as possible. They only provide access to their properties and leave out all additional business logic. It feels more natural to keep entities “dumb” as they represent data rather than logic. Separating the two concepts seems like a better practice than bundling them together.

The CRUD methods have to go somewhere, of course. Putting them in the Controllers would be poor design as this is clearly a responsibility of the Model. Because of that, a new set of business classes called Services was created. Each Service is related to an Entity type (while Controllers are related to their respective Views, all the more reason not to mix the two) and provides relevant business logic. Most notably, it implements the CRUD methods.

Because the CRUD methods mostly follow the same pattern for all Entity types, a generic `BasicService<Entity, PK>` class was created. It provides basic implementation of CRUD methods that works for all Entity types. All other Services, in addition to

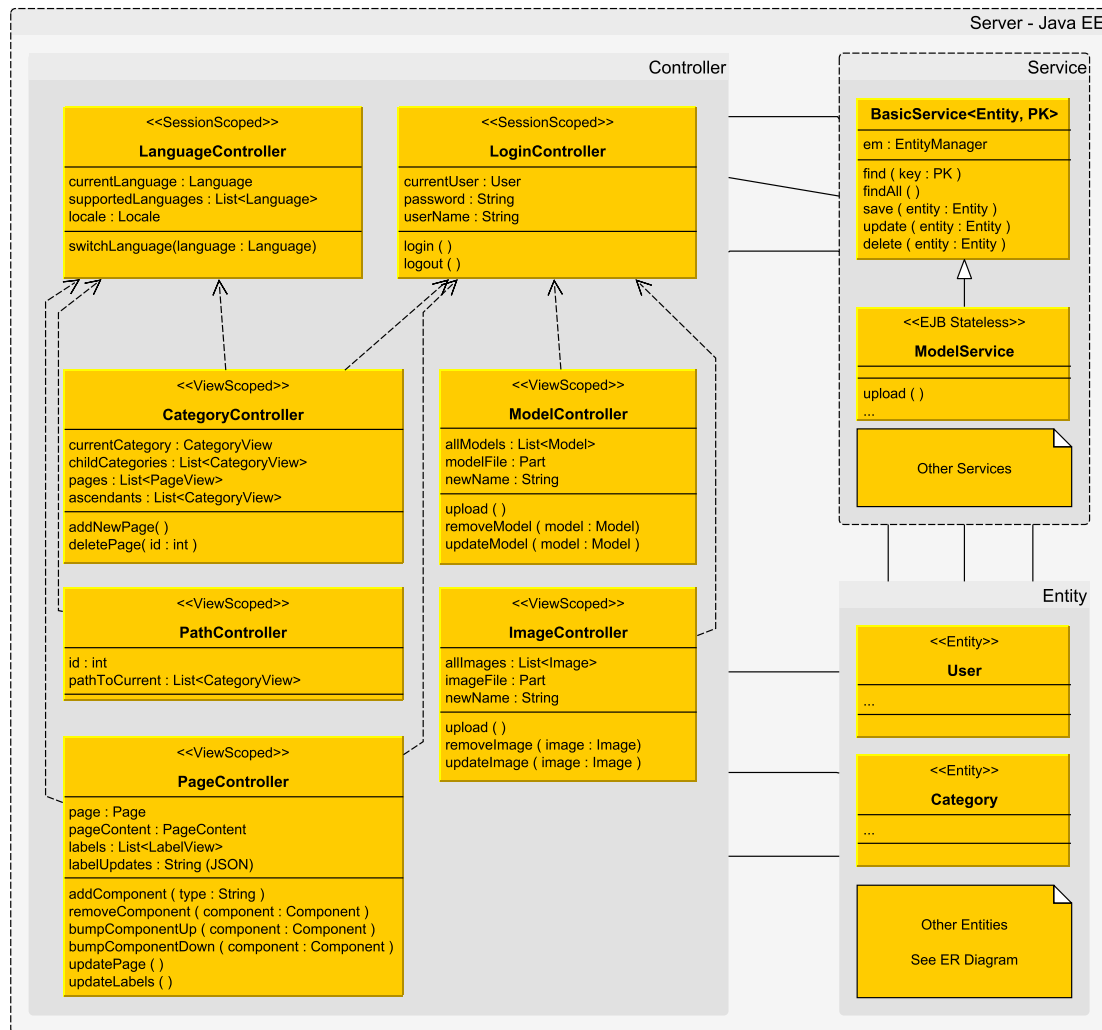


Figure 4.4. Mature design of the application's server side (without Servlets and persistence).

their own methods, inherit from this Basic Service and either use the generic CRUD implementation of their ancestor or override the methods with an implementation specific for their Entity type.

An important benefit of having a designated Service class is that it can (and should) be an Enterprise JavaBean (EJB), which among other things allows container-managed transactions. We simply inject an EntityManager and each method of the EJB will be treated as a transaction.

4.5.2 Entities and Page Components

Entity classes provide public getters and setters for all their properties, which include attributes and relations of their database counterparts. Information about a relation is held at both ends as a simple reference or collection of references. In early stages of the development, these classes also provided methods related to persistence, which were later moved to the service layer.

Our business objects include page components, namely `HeadlineComponent`, `TextComponent`, `ImageComponent` and `ModelComponent`. These components are all

used in a similar way, they belong to a page and they have a property determining their order in the page.

To allow keeping the components in a single collection and sorting them by their order, an abstract common ancestor `PageComponent` was introduced. It implements `Comparable<PageComponent>` to sort components and defines the properties for component order and component type of the instance.

All component Entities inherit from this ancestor and implement its abstract methods. Other than that, they behave just like the other Entities.

■ 4.5.3 Special View Classes

These are not shown in diagram 4.4, but they are closely related to Entities.

Several business objects are represented by a pair of Entities, one holding language-independent properties and the other holding localized data. While this makes localization easy to implement in the database, in Java working with a pair of objects rather than a single object can be inconvenient.

Our solution is to introduce special “View” entity-like classes that represent a localized view of the business object with both its localized and non-localized properties.

The more conservative alternative is to use a reference to the localized Entity and access language-independent properties through its reference to its “main” Entity (other way around is impractical because of the necessity to iterate through all language variants).

Both approaches have their merits. Only using existing Entities leads to fewer objects being created and there is no need to convert two representations. Using an additional class lets us work with a much simpler structure in Controllers (and Views) at the expense of additional work in Model. We can also filter the properties to fit the needs of Views and introduce meta-information not present in the database tables (which could however be added to the original Entities as well).

Looking back, this choice is questionable and worth revising especially if performance becomes an issue.

Three of these classes were introduced:

- **CategoryView** represents a localized summary of category’s properties. Namely its ID, localized name, Latin name and a number of pages (derived). It is mostly used to display lists of subcategories when browsing the categories.
- **PageView** represents a localized summary of page’s properties. Namely its ID, localized name, Latin name and whether it is published. It is mostly used to display lists of pages when browsing the categories.
- **LabelView** represents a localized summary of label’s properties. Namely its ID, localized title, localized text, position and “action”. It is used for two purposes: primarily displaying labels in 3D scenes, but also sending back editorial changes to be persisted (since label editing happens on the client until changes are saved). That is when the action meta-property comes into play, telling the server if a label needs to be created, updated or deleted.

Chapter 5

Implementation

In this chapter we talk about the implementation of all parts of the application, including the database, server logic in Java and 3D viewer in JavaScript. We also briefly describe the application's deployment.

Most (but not all) of the application was implemented and is ready for real use. As of yet unimplemented are the administration tools for managing registered users' accounts.

Implementation was done in NetBeans IDE 8.0.1 [32], a free open source development environment mostly focused on Java. All final source code can be seen on attached CD.

For development purposes, a local Glassfish 4.1 server was installed and integrated into NetBeans for easy "on save" deployment in real time.

5.1 Data Model

The first step of implementation was creating a data model. As mentioned in previous chapter, our business objects will be persisted in a relational database. Images and models will also be using a file system to save their possibly large binary data.

5.1.1 Setting Up a Local Database

Glassfish server comes with a pre-installed Java DB and NetBeans made it easy to setup a connection.

The only pitfall lay in Java's default security policy, which did not allow access to Java DB on standard port 1527. See StackOverflow question [33] for details.

5.1.2 Creating Tables

I used the ER diagram from design phase to generate an SQL script for table creation. I had to make syntactic changes because generated script was in MySQL, which differs from Java DB syntax. After solving all syntactic issues, all tables were successfully created. The working script was saved for later use.

5.1.3 Generating Entities

NetBeans offers a feature for generating Entity classes from a registered database. I gladly used this feature to generate properly annotated, JPA-compatible classes for all Entities.

In later stages of development, Entities returned to being thin data-holding classes without much logic of their own. Other than several simple changes (see below) and removing a small amount of generated ballast, they were largely the same as the originally generated ones.

■ 5.1.4 PageComponent Abstract Class

Page components, namely `HeadlineComponent`, `TextComponent`, `ImageComponent` and `ModelComponent` share this common ancestor.

It implements `Comparable<PageComponent>` and a `compareTo(PageComponent c)` method which sorts components by their order. It also defines an abstract getter and setter for `compOrder` and a getter for `componentType`, which is to return a string representation of component type such as “text” or “image”.

■ 5.2 Service Layer

Services are in charge of business logic, in case of our Atlas mostly handling persistence and relations between business objects. All service classes are EJBs (`@Stateless`) to allow container-managed transactions. This is important, because otherwise all transactions need to be handled manually and JPA `EntityManager` cannot be simply injected.

■ 5.2.1 BasicService

All service classes inherit from this common ancestor. It is a generic class `BasicService<E, PK>` where `E` is an Entity class and `PK` is the primary key class of the Entity.

It injects an `EntityManager` using the `@PersistenceContext` annotation, leaving it to the descendant services to make sure this injection is possible (simply being EJB fulfills this requirement) or creating their own `EntityManager`.

It provides a `BasicService(Class<E> entityType)` constructor, which takes the Entity class as a parameter. This might seem unnecessary when using generic classes, but I found no better way to access the Entity’s type in method calls other than keeping a `type : Class<E>` property and assigning it explicitly in the constructor. Inheriting classes then implement a no-argument constructor which calls `BasicService(entityType)`.

`BasicService` implements 5 default CRUD methods:

- `find(PK key) : E`
finds an Entity by its primary key, using `EntityManager`’s `find(...)` method.
- `findAll() : List<E>`
finds all Entities of a type `E` using a simple JPQL query on `EntityManager`.
- `save(E entity)`
persist a previously unpersisted Entity using `EntityManager`’s `persist(...)` method.
- `update(E entity)`
updates a persisted Entity using `EntityManager`’s `merge(...)` method.
- `delete(E entity)`
deletes a persisted Entity using `EntityManager`’s `remove(...)` method. In order for the deletion to work, it first needs to get a managed instance of the entity by referencing the output of `merge(entity)`.

■ 5.2.2 LanguageService – Example Service

Below you can see the actual code of `LanguageService`, one of the simplest service classes in our application. Note how it extends `BasicService`, calls its super-class’s

constructor and uses inherited `Entity Manager em` to perform operations related to persistence (in this case it executes “find” queries). Other methods such as `findAll()` or `save(Language lang)` are inherited from `BasicService` and not overridden, since their default implementation is sufficient for `Language`.

```

@Stateless
public class LanguageService extends BasicService<Language, Integer> {

    /**
     * Constructs a LanguageService.
     */
    public LanguageService() {
        super(Language.class);
    }

    /**
     * Finds the default persisted Language.
     *
     * @return First result of a "default" flagged Language; or null.
     */
    public Language findDefaultLanguage() {
        // get default flagged language
        try {
            TypedQuery<Language> query = em.createQuery(
                "SELECT l FROM Language l "
                + "WHERE l.defaultLang = TRUE",
                Language.class);
            return query.getSingleResult();
        } catch (NoResultException e) {
            // if none, null
            return null;
        }
    }

    /**
     * Finds persisted Language by ISO short code.
     *
     * @param code ISO code to search for. Examples: "en", "cs"
     * @return Language matching the ISO code, null if no match.
     */
    public Language findLanguageByISO639(String code) {
        // get language with matching code
        try {
            TypedQuery<Language> query = em.createQuery(
                "SELECT l FROM Language l "
                + "WHERE l.short1 = :code",
                Language.class);
            return query.setParameter("code", code).getSingleResult();
        } catch (NoResultException e) {
            // if none, null
            return null;
        }
    }
}

```

■ 5.2.3 UserService

- `login(String login, String pass) : User`

Validates login credentials, uses a JPQL query and either returns a matching user or null.

Unfortunately, advanced user management is not implemented as of yet.

Passwords are hashed using SHA-256 algorithm and salted by appending a randomized string.

■ 5.2.4 CategoryService

- `findRootCategories() : List<Category>`

Uses a JPQL query which returns all categories without a parent (top-level categories).

- `createCategoryView(Category cat, Language lang) : CategoryView`

Creates a `CategoryView` based on `cat` argument and queried `CategoryInfo`. It also calculates the number of pages.

- `countNumberOfPages(Category cat, Language lang, boolean publishedOnly) : int`

Recursively counts the number of pages inside category and its child categories. The recursion can be costly, but this will hopefully prove insignificant due to a low number of categories in the tree. If this ever becomes an issue, the solution is keeping a “numPages” property in the database. This would mean updating all ancestor categories whenever a page is created / removed, but it would speed up regular browsing. It is worth noting that thanks to JPA, this recursion accesses objects in memory and does not cause database hits.

■ 5.2.5 PageService

- `createNewPage(int categoryId) : Page`

Creates a blank, unpublished `Page` including `PageContents` for all language variants. Persists all this and also updates the parent category to recognize the new page.

- `createPageView(Page page, Language lang) : PageView`

Creates a `PageView` based on `page` argument and queried `PageContent`.

- `searchByName(String searchTerm, Language lang) : List<Page>`

Searches for pages whose name (localized or Latin) contains the `searchTerm` as substring. The matching is case insensitive.

It is also worth noting that this service overrides `delete(page)` and updates the parent category’s page collection, which would otherwise still include a reference to the deleted page.

■ 5.2.6 PageContentService

- `updateWithComponents(pageContent, List<PageComponent> components)`

Replaces all `PageComponents` with new ones (passed in argument). Iterates through the `components` collection and adds each component to `PageContent`’s respective collection (text, image...). Changes are persisted using inherited `update(pageContent)`.

■ 5.2.7 ModelService

- `uploadModel(String name, Part modelFile)`

Creates and persists a `Model` entity and also saves the uploaded file to server. The filename is taken from uploaded file, but non-standard characters are replaced and in case of duplicity, numbers are appended. Models are persisted in a designated folder which the server recognizes as an “alternate document root” for easy access on the website.

- `deleteModel(Model model)`

Calls the default `delete(model)` and also deletes the model’s binary file.

- `isUsed(Model model) : boolean`

Runs a JPQL query to find out whether this model is used in any component.

■ 5.2.8 ImageService

Works very much like `ModelService` but with `Image` entities and image files.

■ 5.2.9 LabelService

- `createLabelViews(Model model, Language lang) : List<LabelView>`

Creates all `LabelViews` for a model in given language using a JPQL query. JPQL lets us create instances of any class if we map the query results to a constructor. That feature was used here so that the “JOIN” query returns a list of `LabelViews` directly.

- `updateLabelsFromJSON(String json, Model model, Language lang)`

Labels are edited in client side JavaScript and changes to be saved are sent to the server as a JSON string. The string represents an array of `LabelViews`. This method parses the string using Google’s open source library GSON [34]. Then it treats each `LabelView` appropriately to its `action` property and either creates, deletes or removes a `Label` including its `LabelContent`. *Note: It could be argued that the conversion from JSON to LabelView objects should be done in Controller rather than the Service layer.*

■ 5.2.10 LabelContentService

This service does not implement any new methods, but overrides `save(labelContent)` and `update(labelContent)` to escape all textual data before persisting. This is necessary, because labels are displayed in JavaScript, which easily breaks if unsafe characters are present. This is achieved by `StringEscapeUtils.escapeJavaScript(text)`, which is a method of the Apache Commons Lang [35] library.

■ 5.3 Backing Beans

Backing beans are the next layer of our application, one step closer to the client. In our application they are called Controllers (although the real Controller is in fact a `FacesServlet` working behind the scenes).

Each backing bean belongs to a View component (a reusable piece of HTML), provides content from the Model and methods for manipulating it.

Backing beans are annotated with their “scope”, which defines the bean’s lifespan. Most beans are either `@ViewScoped`, which means they’re recreated when a new page is

displayed, or `@SessionScoped`, which means the bean lives and holds its state as long as a session with a client is maintained.

It is important to note that proper annotation and import is required, for example `@SessionScoped @Named(loginController)`. The correct imports would be `javax.faces.view.ViewScoped` and `javax.inject.Named`. Some older tutorials use `@ManagedBean` annotation and different imports, which is an outdated approach and CDI annotations mentioned above should be used instead. If you happen to mix the wrong annotations and imports, your code will break.

The typical life cycle of a backing bean is this:

- Initialization, when relevant data is fetched from Model
- Providing methods and data for View
- End of life (depends on scope)

Most beans' initialization methods are called using `@PostConstruct` annotation, but several need an attribute from View (typically page ID) to initialize properly. This is realized by Facelets, which calls the `init(...)` method when bound properties are already set. We tell Facelets to do this by setting `f:viewAction` and `f:viewParam` tags.

Implementing “bound properties” is really easy in a CDI annotated bean. All you have to do is have a private field with public getter/setter and the property is bound. It can then be accessed in a Facelets page like `value="#{loginController.userName}"`.

Action methods are public methods which can be called from Facelets pages such as `action="#{loginController.login()}"`. These methods can have parameters and should return a `String` (redirection rule) or `void`. They are sometimes called within Ajax context and sometimes not. For this reason many action methods in our application specify a `boolean ajax` parameter which determines whether the page should be reloaded at the end of the method run. If `ajax` is `true`, returned redirection string is `null` so that Ajax can take care of the view update.

■ 5.3.1 BasicController

This is an abstract class which backing beans can inherit from.

Backing beans do not have too much in common, so this ancestor is not really necessary. It was introduced in later stages of development to make the other backing beans' code a little neater and easier to read.

It implements methods for displaying messages in the View. Messages are passed as a string argument and set as `FacesMessage` in current `FacesContext`. Because the messages can be (and typically will be) localized strings from a resource bundle, these methods evaluate their string input as an EL expression. That way, simple strings are displayed directly, but EL expressions can be used just like they would be used in a Facelets page.

Displaying messages is a common task especially during content editing. This originally created chunks of repeated, verbose code. Implementing a common ancestor reduced these chunks to one-line method calls, which look like this for localized messages:

```
showWarning("#{messages.noRights}").
```

■ 5.3.2 LanguageController

This session bean is initialized post-construct. It communicates with Model through `LanguageService`. During initialization, it takes the preferred locale from client

browser and either assigns a matching language or resorts to default language if no match is found. It also loads all supported languages.

During its life it holds the current language, current locale, a list of available languages and provides a method for switching languages, which simply sets the `currentLanguage`, locale and reloads the page.

Many other backing beans refer to this one to find out what language should be used and it is their responsibility to choose the appropriate content, not `LanguageController`'s.

■ 5.3.3 LoginController

This is the second of our two session beans and is also initialized post-construct. It communicates with Model through `UserService`. During initialization, it sets current user to `null` and user privileges to `false`.

During its life it holds the current user, a flag denoting “edit rights”, properties bound to login input fields and provides methods for login and logout.

Again, many other backing beans refer to this one to confirm user rights and it's their responsibility to take appropriate actions, not `LoginManager`'s.

■ 5.3.4 PathController

This request scoped backing bean provides a path to current page or category, simply put a page's parent, grandparent etc. all the way to a root category. It is initialized by Facelets and requires an `isPage` argument (to define whether it's working with a page or category) and the ID of current page/category (which is in fact a bound property, although we could have made it a method argument).

It uses `PageService` or `CategoryService` to find the page/category by ID and then adds its ancestors in a loop: While “current” page has a parent, add it to path and make it “current”. The path is a list of `CategoryView`.

■ 5.3.5 CategoryController

This view scoped backing bean is initialized by Facelets so it can access current category ID (bound property) during initialization. It provides content of current category (including “views” of its children, non-recursive) and methods for managing pages in this category. Management of categories as such has not been implemented yet. It also provides root categories instead of children in case current category is unset. It uses both `CategoryService` and `PageService`.

Methods `addNewPage(...)` and `deletePage(...)` check user rights, call corresponding service methods, display a message and reload the page (or not if Ajax is used). `goEditPage(...)` is just a redirect which checks user rights.

■ 5.3.6 SearchController

This view scoped backing bean is initialized by Facelets so it can access its `search term`, which is passed as an URL parameter and bound to a property. It provides a list of all pages matching the search term using `PageService`'s `searchByName(...)`.

■ 5.3.7 PageController

This view scoped backing bean is initialized by Facelets so it can access current page ID (bound property) during initialization. It provides content of the page and methods for editing the page including labels of the page's 3D model. It uses `PageService`, `PageContentService` and `LabelService` to communicate with Model.

Page components are fetched during initialization and sorted. Once sorted, their `compOrder` is set to increment by one (0, 1, 2 .. n) to allow easy manipulation of component order during editing.

3D models get a bit of special treatment because of the need to handle label editing. In current implementation only one 3D model is allowed in a page (if more are present for some reason, only first one is processed). Thanks to `PageComponent`'s interface, all components are held in a single collection and the same methods are used to manipulate all of them.

During page editing, changes are held in this backing bean and only uploaded to database when saved using `updatePage(...)`, which persists current page and its component collection. Because of this, all editing requests are required to update the page using Ajax; otherwise the view, this bean and all unsaved changes would be reset.

`addComponent(String type)` method is implemented as a switch which pre-fills a default component of given type and places it at the end of collection. Methods for changing component order swap two components in the collection and adjust their `compOrder` appropriately. Components are removed simply by removing them from the collection. No methods for editing the content of individual components are needed, because being managed properties, they can be edited directly in Facelets pages.

A method for persisting changes of 3D model's labels was implemented. It passes a JSON String generated by JavaScript to `LabelService`, where all the magic happens.

■ 5.3.8 ModelController

Note that this takes care of a "Model Manager" page, not displaying models in pages (backing beans are related to a particular view/page rather than Entities). This view scoped backing bean is initialized post-construct and fetches all 3D models from `ModelService` and sorts them alphabetically.

In addition to more standard properties, a `Part modelFile` is bound. It is used to upload files into the system and `Part` is the object generated by Facelets' upload file component. Methods to upload, edit (meta-data, not geometry) and delete models check user rights and call corresponding service methods. Deletion is preceded by making sure the model is not used in a component.

■ 5.3.9 ImageController

This bean is analogous to `ModelController`, but works with images.

■ 5.3.10 Converters

`ModelConverter` and `ImageConverter` classes implementing `Converter` interface were introduced to allow displaying Models and Images in drop-down lists. Implementation is trivial and maps Entity objects to their IDs.

■ 5.4 View – Facelets and HTML

Since we are creating a web application and using JSF, the View is realized by a series of HTML (in fact, XHTML) pages generated by JSF's presentation technology. In our case, that technology would be Facelets.

Facelets is a page declaration language which can be used for creating templates, UI components and much more. It allows effective work with XHTML and uses Expression Language (EL) to access methods and properties of managed beans. It also allows the use of basic conditional statements and iteration through collections. These features are

included in several supported tag libraries that need to be included as XML namespaces. For more details see [36].

■ 5.4.1 Templating

All our pages follow a single basic template located at `webapp/template/basic.xhtml`. A template defines a basic skeleton of the page including interchangeable components placed in their respective locations.

This is an example of inserting our main content component in the template and choosing `welcome.xhtml` as the default component (which is used unless specifically overwritten):

```
<ui:insert name="content">
    <ui:include src="../../components/welcome.xhtml" />
</ui:insert>
```

And this is how we would use this template in an actual page that uses `content_page.xhtml` as its main content (assuming other components stay default):

```
<ui:composition template="/template/basic.xhtml">
    <ui:define name="content">
        <ui:include src="components/content_page.xhtml"/>
    </ui:define>
</ui:composition>
```

Using such a template lets us reuse components and if the defaults are set correctly, we only need to define one or two components per page. This practice reduces writing a new page into writing a content component. It makes the pages easily maintainable, as changing a component affects all pages that are using it without disturbing the rest of the layout.

■ 5.4.2 Page URLs

Pages are accessed at URLs matching their name appended to the server's root URL. At a local server running on GlassFish's default port, the `index.xhtml` page would then be accessible at `http://localhost:8080/index.xhtml`.

However, all pages (or categories) are using a common `page.xhtml` webpage. To differentiate between individual content pages, an ID attribute is added to the URL. A page with ID = 1 would be accessible at `http://localhost:8080/page.xhtml?id=1`. URL IDs match database IDs of pages/categories.

This makes the pages bookmarkable and easy to navigate from server's point of view, but it does not look very pretty. In the future, a library such as PrettyFaces [37] might be used to rewrite URLs.

■ 5.4.3 Style – CSS

Pages are styled using a combination of a basic style sheet `basic.css` and an additional sheet specific for the page, such as `category.css`. Components generated by JavaScript are styled inside JavaScript.

Although the style is not overly complex, CSS presented me with a lot of issues regarding absolute/relative positioning, alignment and sometimes browser compatibility.

The final style should be compatible with all major browsers and provide scalability within reasonable limits.

■ 5.4.4 Ajax

Some pages are using Ajax to update only their necessary parts instead of reloading the whole page. Thankfully Facelets provides a `f:ajax` tag which is easy to use.

Unfortunately, this particular JSF build is suffering from an issue where view state is not retrieved upon Ajax calls. This resulted in losing page/component IDs (URL parameters) upon consequent Ajax calls. This was fixed by appending a JavaScript file which adds missing hidden `ViewState` fields to forms (see [38] for details).

■ 5.4.5 Localization

`LanguageController` is responsible for choosing the current locale. There are two levels of localization in the application:

- **Localized content** is realized in Model layer by Entities representing localized data.
- **Localized UI** is realized by inserting localized strings from a property file. Property files such as `WebStrings.en.properties` contain key-value pairs of strings for the UI. They are registered as named resource bundles in `faces-config.xml` so that they can be easily accessed in EL expressions.

■ 5.4.6 Language Component – Example Component

`language_bar.xhtml` displays flags for changing languages in all pages and is backed by `languageController`.

All available languages are taken from `languageController.supportedLanguages`.

Following Facelets snippet demonstrates how a template component is defined (`ui:component`), how HTML components can be defined using Facelets tags (`h:panelGroup`) or plain XHTML tags (`div`), how backing bean data is accessed (`value=...`), how basic iteration through a collection is done (`ui:repeat`), how action methods are invoked (`action=...`) and how images are inserted (`h:graphicImage`):

```
<ui:component>
  <h:panelGroup>
    <ui:repeat var="lang"
      value="#{languageController.supportedLanguages}">
      <h:form>
        <div class="flagDiv">
          <h:commandLink
            action="#{languageController.switchLanguage(lang)}">
            <h:graphicImage
              alt="#{lang.name}" title="#{lang.name}"
              library="images" name="flags/#{lang.flagFile}" />
          </h:commandLink>
        </div>
      </h:form>
    </ui:repeat>
  </h:panelGroup>
</ui:component>
```

■ 5.4.7 Other Common Components

Unless stated otherwise, these components are displayed in all pages and do not have a designated backing bean.

- `login_bar.xhtml` provides typical login fields and buttons and is backed by `loginController`. There are two alternate forms, one for login and one for users who are already logged. This is implemented by setting the `rendered` attribute of each form to `loginController.logged` or its negation.
- `logo.xhtml` displays the main logo, which is also a link to the index page.
- `horizontal_menu.xhtml` displays buttons leading to Model Manager and Image Manager. It is only visible when an editor is logged in. It does not have its own backing bean, but checks `loginController` for current user's role.
- `search_bar.xhtml` provides a search field and submit button. It does not use a backing bean, just redirects to `search.xhtml` with input as URL parameter.
- `messages.xhtml` displays messages and warnings which are not paired with any particular UI component. In other words, this is the default place where messages are displayed.
- `path.xhtml` displays a path to current page, which is a series of links to higher categories for easier navigation. It is used in category and content pages and `pathController` is its backing bean.
- `footer.xhtml` contains a simple copyright note and a link to “Help” page.

■ 5.4.8 Page Specific Components

- `category.xhtml` is a component representing “content” when browsing categories. It is backed by `categoryController`. Some extra buttons are shown to editors, so it also uses `loginController` to check current user's role.

It displays general information about current category and iterates through two lists of items: subcategories and pages. Both types of items are generated by a subcomponent `category_item.xhtml`, which creates a clickable visual representation of a subcategory or page.

- `search_results.xhtml` is a component displaying search results (pages matching a search query). It is backed by `searchController` and shows a list of pages similar to the “pages” section of `category.xhtml`, also using `category_item.xhtml` subcomponents.
- `content_page.xhtml` is a component representing actual content, the pages everyone came to see. It is backed by `pageController`.

It displays general information about the page and iterates through all the “page components” using a `content_component.xhtml` subcomponent. “Page components” currently come in four types:

- **Headline** components are rendered as `<h3>` headlines.
 - **Text** components are rendered as `<p>` paragraphs.
 - **Image** components are rendered as `<image>` images followed by a line of text.
 - **Model** components are rendered as an interactive canvas generated in JavaScript (see relevant section) followed by a line of text.
- `edit_content_page` is a separate editorial interface for pages. This main component provides a static editorial toolbar and editable fields of the page such as name. Current state of the content is shown by iterating `edit_content_component` subcomponents, which provide editable fields of related “page components”. These temporary

“page components” are not persisted, they only exist in the backing bean and the page is updated by Ajax calls. It is of course possible to persist them by clicking a “Save changes” button in the toolbar.

- `models.xhtml` represents Model Manager. It is only accessible to editors and uses `modelController` backing bean. It provides a form for uploading 3D model files into the system (STL format).

Below that form is a list of existing models, realized by iterating `model_item.xhtml` subcomponents, which are editable.

- `images.xhtml` represents Image Manager and is analogical to `models.xhtml`.
- `welcome.xhtml` is a simple Home page which displays introductory textual information and a list of root categories (which means it uses `categoryController`).
- `help.xhtml` is a Help page which displays a short user guide.

5.5 Model Viewer

The model viewer runs in the client browser rather than the server and is written in JavaScript. The script can be found in `view3d.js`.

`three.js` graphical library [25] was used, as well as some of its “example” classes, namely `STLLoader.js`, `TrackballControls.js` and `Detector.js`.

5.5.1 Basics, Rendering

Typical `three.js` applications work with a `Scene` which holds a `Camera` and other 3D objects. They then use a `Renderer` to display the `Camera`’s “view” onto a `Canvas` DOM element in an HTML page.

There are several `Renderers` available, most notable being `WebGLRenderer`, which uses the WebGL technology to allow hardware acceleration. The alternative `CanvasRenderer` works without using WebGL and provides wider support at the cost of performance. WebGL is clearly superior, but we enable `CanvasRenderer` as a “fallback” option when no WebGL is available in client browser (detected by `Detector.js`). A warning message is displayed though, as `CanvasRenderer` cannot be expected to work properly.

A typical scene is statically initialized and then animated in a rendering loop. An inefficient way to do this is to call the rendering function every few milliseconds. A more efficient way is using the `requestAnimationFrame` API, which lets the browser handle rendering calls, synchronizing redraws of multiple animations and stopping the loop when the scene is not visible. That is what our application was initially doing. Since our scene includes no automated animation and every motion is triggered by the user, I decided to go one step further and abandoned the render loop altogether. Instead, the `render` function was added as a listener to controls’ `change` event, only redrawing on user interaction.

5.5.2 Materials and Lighting

`three.js` takes care of illumination and shading as long as lights and materials are set properly.

Mono-colored materials using Lambert’s lighting model were used for shaded surfaces such as the 3D model. For unshaded objects, basic mesh/line materials were used.

The scene is lighted by a weak ambient light and a stronger point light, which is always following the camera (slightly offset up and to the right). This ensures that

no part of the model is completely dark (ambient light) but surface unevenness is emphasized by shading (point light).

■ 5.5.3 3D Models

The models available to Third Faculty of Medicine come in binary STL format, so naturally that is the first format to be supported.

`three.js` provides a ready-to-use class for loading STL models called **STLLoader**. Using this class, model geometry is loaded as a **BufferGeometry** object for efficient rendering. A **Mesh** object which can be rendered is created using the geometry from **STLLoader** and a custom material. It is then placed in the center of the scene and scaled so that the longest dimension is 100 units.

There are minor problems with the STL format and currently available models: The files are unnecessarily large, shared vertices are treated as separate and no normals are provided. This results in longer loading times and a slightly “polygonal” look, because the lack of normals prevents smooth shading. Recalculating normals is non-trivial because shared vertices would need to be detected every time the model is loaded (moreover, **BufferGeometry** is not easily altered). One of the possible future solutions is the introduction of a more efficient custom file format and a converter from STL files. That way vertices would be merged and normals recalculated once during conversion, rather than every time the model is loaded.

■ 5.5.4 Labels

Labels are stored in the database and sent to the client as JavaScript variables representing an array of **LabelViews**, which have several attributes relevant for displaying them:

- Short title, which should be visible in the 3D scene
- Description, which should be displayed on demand
- Location of the relevant point on the bone
- Location of the label tag (ideally where it does not obstruct the model)

The application supports several display modes for labels, showing either named tags, blank pins or nothing.

During initialization of the scene, labels’ 3D representation is created based on the input **LabelViews**. In the 3D scene, all labels are children of an **Object3D labelHolder** and each label is represented by an individual **Object3D**. This object stores all the information about the label in custom attributes and has several children, making up the visual representation of each label:

- **Sprite**, which is the floating tag with a title. It was originally realized as a **Sprite**, which is a plane always facing the camera. Unfortunately, this approach did not allow picking (see 5.5.5). The sprites were then remade as simple planes (**Mesh** with **PlaneGeometry**) and their orientation is manually changed in the rendering function so that they always face the camera. This is done by copying the camera’s quaternion.

`three.js` does not support adding text as such, so a texture with the title is generated for each label. For this purpose a **Canvas** is created, length of title is calculated, then the canvas is stretched to fit the title, a semitransparent rounded rectangle of appropriate size is drawn on the canvas (rest is transparent) and the title is printed. A texture is created from the canvas and mapped onto the plane geometry.

- **Pin**, which is a simple ball replacing the sprite in one of the modes. They share the location with sprites and are never displayed at the same time as sprites. This is simply a **Mesh** with **SphereGeometry** and a simple material.
- **Point**, which marks the relevant point on the model's surface. It is a small **Mesh** with **SphereGeometry** and a red, unshaded material.
- **Line**, which connects the point and sprite. It is red at the starting point and fades into background color near the sprite to be less obtrusive. It would look better to fade into transparency, but that would require a custom shader, which is a lot of effort for such a small improvement.

■ 5.5.5 Label Selection

Labels are selected by picking, which is a method of interaction letting the user select objects in the 3D scene. In our application, it is realized by **three.js**'s **Projector** and **RayCaster**, which can create a ray starting at a point on the screen and going straight “through”. Intersections of the ray and 3D objects in the scene can be retrieved.

It is important to carefully map screen coordinates to viewport coordinates because the viewport (canvas) can be located anywhere on the screen due to other content, scrolling or zooming. The converted coordinates should be in $<-1; 1>$ range. It is also important to note that the vertical coordinate needs to be inverted. Following snippet shows correct conversion:

```
mouse.x = ((event.clientX -
    renderer.domElement.getBoundingClientRect().left) / width) * 2 - 1;
mouse.y = - ((event.clientY -
    renderer.domElement.getBoundingClientRect().top) / height) * 2 + 1;
```

The ray can be created using **Projector**'s **unprojectVector(...)**, which helps us transform viewport coordinates into world coordinates. We use the ray to construct a **RayCaster**, whose **intersectObjects(...)** method returns all intersections including location and object which was hit.

For label picking, we are checking if any visible parts of labels were intersected. If so, **selectLabel(label)** function is called. Previously selected label is unselected and this one is selected. In case this label was previously selected, it is simply unselected. A selected label is marked by a different color sprite and its description is shown in a special text field (see 5.5.7).

■ 5.5.6 Camera Controls

Camera position can be changed by user. Unlike some other graphical libraries, **three.js** changes view by actually manipulating the camera rather than the entire world.

To handle user interaction with the camera, **TrackballControls.js** class was used. Originally **OrbitControls.js** was used, but users objected to orbit's slightly constrained movement (vertical limit to ± 90 deg.). The natural replacement was trackball, which allows free rotation around all axes.

Trackball allows three types of view manipulation, while the camera is always looking at current target point:

- Rotation uses an imaginary “trackball”, a sphere centered in current target point. When user drags the left mouse button, camera orbits the target and the scene reacts as if the user grabbed a point on the trackball and rotated the trackball around its center.

- Panning is activated by dragging the right mouse button and it moves the camera without changing its rotation, which means the target point is also moved. The translation of both camera and target is perpendicular to camera direction.
- Zooming is activated by mouse wheel and it changes the camera's "orbit radius", or simply put the distance between camera and target.

The `TrackballControls` class was slightly altered to fire `change` events on mouse movement and only propagate the original event when we want to (so that the page does not scroll when we zoom).

Current controls work well with a mouse. They also support touch devices and basic one/two finger gestures. However, some additional tweaking might be necessary in order to intuitively combine 3D controls with default web controls on mobile devices (e.g. 3D zoom vs. page zoom).

■ 5.5.7 UI Components

Not everything we need to display is a part of the 3D scene. Certain elements such as buttons and select boxes create the 2D UI which does not change position when the camera moves.

They are created and updated in JavaScript by manipulating the page's DOM. The biggest issue is their placement, which needs to be recalculated on resize.

These UI components include:

- A box which displays selected label's description (or default text if unselected)
- A select box which allows users to switch label display mode
- A "loading" sign which is displayed until `STLLoader`'s `load` event is fired
- Buttons related to label editing, which are only displayed to authenticated editors

■ 5.5.8 Window Resize

It is important to properly handle window resize and to initialize the canvas correctly depending on window size (we call our `onWindowResize()` function during initialization too).

The size of the containing element (a `div`) is determined by the page's CSS.

On resize we need to set the size of our renderer to fit into the `div`'s width, decide on a reasonable height (4:3 ratio by default, but never higher than window), change the camera's aspect and update its projection matrix.

Then we need to position all UI components appropriately. Because of issues related to positioning and picking, the containing element's position is set as `static`. That prevents us from using relative positioning for UI components. This leads to all UI components being placed absolutely, their position and dimensions calculated from `container.offsetLeft`, `container.offsetTop` and renderer's `width` and `height`.

Lastly, the controls are updated so that the "trackball" is centered correctly.

After this setup the scene is rendered to reflect the changes.

■ 5.5.9 Editing

The script also includes everything needed for editing the labels. Following components and features are available when an editor is logged in:

- **Save changes** button, which creates a JSON string representing an array of `LabelViews` of changed labels and submits it to server.

- **New label** button, which enables the editor to select a point on bone's surface (see picking in 5.5.5, except we intersect the model). From that point a new default label (default text, set distance of tag from surface, perpendicular to surface) is created and selected.
- The box with label description is made **editable**, so that upon selection two editable fields (title and description) are dynamically created and their `oninput` events alter the label's content. Changing the title also redraws the label's texture in real time. A **Delete** button is appended to the box, letting the editor remove selected label.
- Label **sprites can be dragged** when selected. The translation is perpendicular to camera direction and realized by intersecting (see picking in 5.5.5) a large invisible plane facing the camera and placed where the sprite is placed. Theoretically just intersecting the sprite should be enough, but in practice users can drag faster than the application reacts and "lose grip" of the sprite when they quickly move their cursor out of it. The line connecting label's point and sprite needs to be recreated.

Labels include a `needsUpdate` flag to determine if they need to be submitted to server. To further specify the action, `isNew` and `isDeleted` flags were introduced. This means that labels deleted in current editorial session are not in fact removed from the scene, just made invisible and marked for deletion so that we know what to delete when changes are submitted.

■ 5.5.10 Mouse Mode and Mouse Events

`TrackballControls` take care of their events, but we still need to listen to mouse events because of picking and editing.

Three custom mouse modes are used:

- `view` only enables label selection
- `moveSprite` allows the sprites to be dragged
- `addLabel` allows selection of a point for new label

And three mouse events are processed:

- `mousedown` checks if a label was hit and if it was, it remembers which one. If a sprite was hit and an editor is logged in, mouse mode is switched to `moveSprite`.
- `mousemove` is mostly needed in `moveSprite` mode (see 5.5.9), but also displays a marker at intersection point in `addLabel` mode.
- `mouseup` checks what was hit and reacts depending on current mode:
 - `view`: If the same label was hit in `mousedown`, it is selected.
 - `moveSprite`: Mode is switched back to `view`.
 - `addLabel`: If model was hit, new default label is created (see 5.5.9).

■ 5.5.11 Using the Script, Required Variables

To use the `view3d.js` script in a HTML page, two basic requirements must be met.

All required libraries/classes must be included before this script is run:

- `three.js` or `three.min.js`
- `Detector.js`
- `STLLoader.js`
- `TrackballControls.js`

Several variables need to be set before this script is run:

- `editable` determines whether editor mode is enabled.
- `modelPath` leads to the 3D model file to be displayed.
- `strings` is an object containing key-value pairs of localized UI strings.
- `labels` is an array containing localized labels in `LabelView` format.
- `labelUpdateInput` and `labelUpdateButton` are (invisible) DOM elements used to input and submit label updates. They are bound to backing beans, so they cannot be created inside the script.

5.6 Data Transfer

Previous version of the Atlas contained models and labels which are still relevant in our application as well as a hierarchy of categories which can be used.

I created a Java application called `AtlasFiller` for the sole purpose of transferring this data on a one-time basis.

Given correct paths, it parses all relevant XML files taken from the old Atlas and creates a hierarchy of data objects representing categories, pages and labels. This representation is then inserted into our application's database as the currently used structures: A set of `Models` and their `Labels`, a tree of `Categorys` and `Pages`, each page having one `ModelComponent` with the related `Model`. Localized counterparts of these Entities were created in both Czech and English.

Model files need to be copied to the application's `.../uploads/models` folder manually.

5.7 Deployment

The application was deployed to a production server belonging to Third Faculty of Medicine in December 2014 under the name **Skelet 3D**. A virtual machine running Debian OS was created specifically for our application.

To match the conditions used during development, a GlassFish 4.1 server was installed and its pre-included Java DB database was setup as well as proper connection pools (using GlassFish's Administration Console). It is worth noting that Java DB does not start automatically with the server and needs to be started manually from the command line.

Database tables were created remotely using the SQL script from early implementation stages (see 5.1.2) and populated with basic data such as `Languages` and example user accounts.

Data from previous version were remotely imported using the `AtlasFiller` application (see 5.6).

Storage for uploaded files was setup at `/home/glassfish/atlas-webapp/uploads/` and existing models were transferred manually.

The application itself could not be deployed using NetBeans for unknown reasons, so the WAR file was deployed manually using GlassFish's Administration Console.

Daily backup of the database as well as uploaded files was setup to prevent loss of data.

The application is currently running on this server at `http://195.113.62.79/`, also replacing the previous version of Atlas on the domain name `http://skelet3d.lf3.cuni.cz/` [39].

■ 5.8 Documentation

- A short, basic user guide is available on the “Help” page directly on the website.
- A thorough editor’s guide was created and was given to current editors. It can be seen on attached CD.
- JavaDoc documentation for public APIs was generated from source code’s comments. It can be seen on attached CD.

Chapter 6

Testing

This chapter describes what steps were taken to evaluate the quality of the application.

Basic functionality of the application was being tested by the author during the whole implementation process, by editors during later stages of development and finally by several uninvolved users. Both editors and regular users provided valuable feedback regarding user experience.

6.1 Use Case Validation

The first step of evaluating the application is use case validation. It tells us which functional requirements were met. Each use case was validated by walking through its scenarios and observing expected results. Colored use case diagram 6.1 shows that most use cases were successfully implemented. Two use cases were not implemented as of yet: password change and account management.

6.2 Cooperation with Editors

During later stages of implementation, three editors familiar with previous versions of the Atlas provided regular feedback on current version's features and user experience. They tested all editorial features as they became available. Many smaller and larger issues were discovered and addressed during this stage.

These are just several examples of issues the editors helped identify and their solutions:

- *Constrained camera movement*: Orbit controls replaced by trackball controls
- *Parts of bones completely white at certain angles*: Material changed and lighting reduced
- *UI components of 3D viewer misplaced (under specific conditions)*: Absolute positioning of components determined after all other content is loaded
- *Files containing special characters in filenames not working*: Filenames processed and stripped of special characters
- *Models have edges rather than smooth surfaces*: Known issue with model format, solution pending

The editors were asked for ways to improve the editorial interface and what features they were missing. I emphasized that improvements we make now would make their future work easier. Nevertheless, they did not make any additional requests.

6.3 User Feedback

TODO

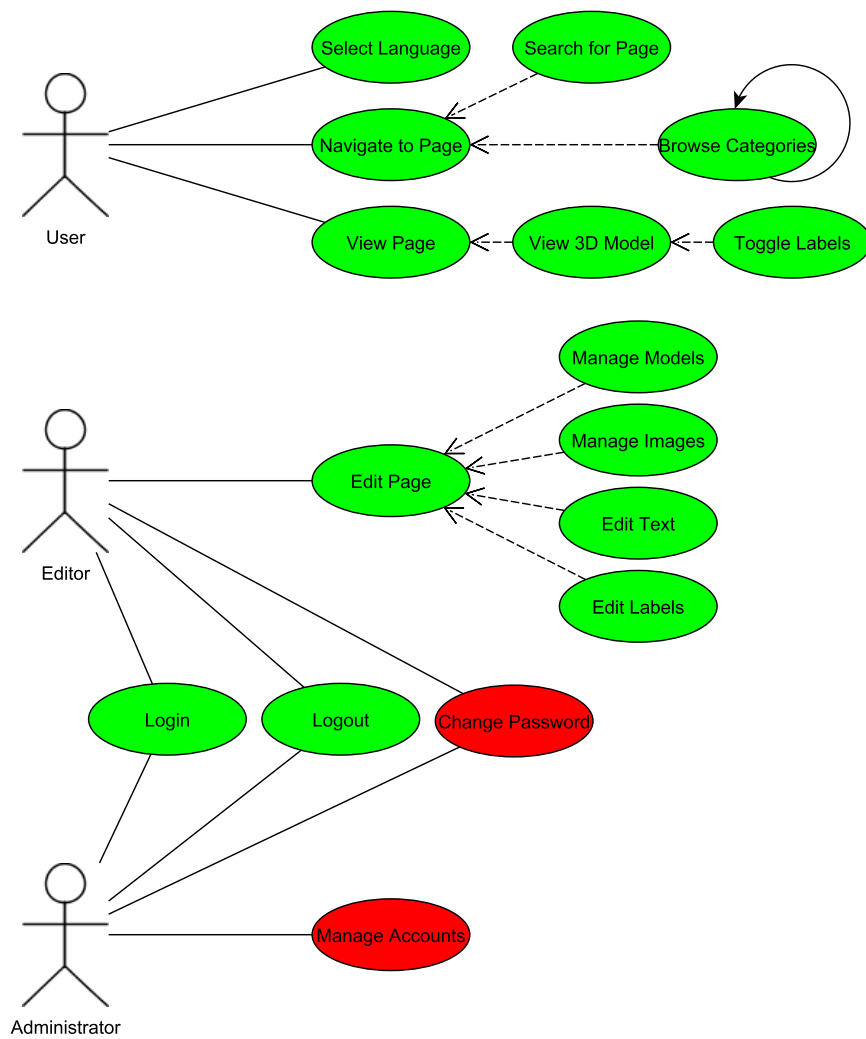



Figure 6.1. Diagram of use case validation results: Green use cases were successfully implemented, red use cases were not. It shows that the application is almost complete, only account-related features are missing.



Chapter 7

Conclusion

TODO

References

- [1] *Skelet 3D*.
<http://skelet3d.lf3.cuni.cz/>.
- [2] *Skeletopedia*.
<http://skeletopedia.sk/>.
- [3] *Zygote Body*.
<http://www.zygotebody.com/>.
- [4] *Anatronica*.
<http://www.anatronica.com/>.
- [5] *3D Science - Skeletal Models*.
http://www.3dscience.com/3D_Models/Human_Anatomy/Skeletal/.
- [6] *UPOL - Atlas člověka*.
<http://www.atlascloveka.upol.cz/cs/cs02/cs0201/cs020100.html/>.
- [7] *Wikipedia - List of Human Bones*.
http://en.wikipedia.org/wiki/List_of_bones_of_the_human_skeleton.
- [8] *Can I Use - WebGL*.
<http://caniuse.com/#feat=webgl>.
- [9] *Internet Explorer Dev Center - WebGL*.
<http://msdn.microsoft.com/en-us/library/ie/bg182648%28v=vs.85%29.aspx>.
- [10] Stephen Shankland. *CNET - iOS 8 brings big boost for Web programmers*.
<http://www.cnet.com/news/ios-8-brings-big-boost-for-web-programmers/>.
- [11] *Adobe - Flash Penetration Statistics*.
<http://www.adobe.com/cz/products/flashplatformruntimes/statistics.html>.
- [12] Tom Krcha. *Using MouseLock, Right Click, Middle Click features in Flash Player 11.2*.
<http://tomkrcha.com/?p=2621>.
- [13] Marcus Kruger. *Wired - Flash Is Dead*.
<http://archive.wired.com/insights/2014/05/flash-dead-long-live-webgl/>.
- [14] Serdar Yegulalp. *Infoworld - Adobe Flash: Insecure, outdated, and here to stay*.
<http://www.infoworld.com/article/2610420/adobe-flash/adobe-flash--insecure--outdated--and-he.html>.
- [15] *BuiltWith - Shockwave Flash Embed Usage Statistics*.
<http://trends.builtwith.com/framework/Shockwave-Flash-Embed>.
- [16] *Adobe - About Flash Player*.
<https://www.adobe.com/software/flash/about/>.
- [17] Danny Winokur. *Adobe News - Flash to Focus on PC Browsing and Mobile Apps*.
<http://blogs.adobe.com/conversations/2011/11/flash-focus.html>.

- [18] Tareq Aljaber. *Adobe Blogs - An Update on Flash Player and Android*.
<http://blogs.adobe.com/flashplayer/2012/06/flash-player-and-android-update.html>.
- [19] *Wikipedia - Apple and Adobe Flash controversy*.
http://en.wikipedia.org/wiki/Apple_and_Adobe_Flash_controversy.
- [20] *Unity 3D*.
<http://unity3d.com/>.
- [21] *ShiVa3D*.
<http://www.stonetrip.com/>.
- [22] Felix Turner. *Airtight Interactive - Stage3D vs WebGL Performance*.
<http://www.airtightinteractive.com/2011/10/stage3d-vs-webgl-performance/>.
- [23] *WebGL Samples - Aquarium*.
<http://webglsamples.googlecode.com/hg/aquarium/aquarium.html>.
- [24] *WebGL User Contributions*.
https://www.khronos.org/webgl/wiki/User_Contributions.
- [25] *three.js*.
<http://threejs.org/>.
- [26] *Stack Overflow*.
<http://stackoverflow.com/>.
- [27] *IEEE Spectrum - The Top Programming Languages*.
<http://spectrum.ieee.org/static/interactive-the-top-programming-languages>.
- [28] *PHP: a fractal of bad design*.
<http://eev.ee/blog/2012/04/09/php-a-fractal-of-bad-design/>.
- [29] Walker Rowe. *Southern Pacific Review - Choosing Java Vs .Net*.
<http://southernpacificreview.com/2013/08/08/choosing-java-vs-net-for-web-development/>.
- [30] Srinath Davu. *Segue Technologies - .NET vs Java*.
<http://www.seguetech.com/blog/2013/06/03/dotnet-vs-java-how-to-pick>.
- [31] *Oracle - JavaServer Faces*.
<http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>.
- [32] *NetBeans IDE*.
<https://netbeans.org/>.
- [33] *Stack Overflow - Unable to Start Derby...*
<http://stackoverflow.com/questions/21154400/unable-to-start-derby-database-from-netbeans-7-4>
- [34] *Google GSON*.
<https://code.google.com/p/google-gson/>.
- [35] *Apache Commons Lang*.
<http://commons.apache.org/proper/commons-lang/>.
- [36] *Java EE 6 Tutorial - What Is Facelets?*
<http://docs.oracle.com/javaee/6/tutorial/doc/gijtu.html>.
- [37] *PrettyFaces*.
<http://ocpsoft.org/prettyfaces/>.
- [38] *Balusc - Fix Missing JSF ViewState After Ajax*.
<http://balusc.blogspot.cz/2011/09/communication-in-jsf-20.html>.
- [39]