



Leo Novosel, BSc

Usage of Cloud Services in Modern Bookkeeping Applications

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Nikolai Scerbakov
Institute of Interactive Systems and Data Science

Graz, April 2019

This document is set in Palatino, compiled with [pdfL^AT_EX2_ε](#) and [Biber](#).

The L^AT_EX template from Karl Voit is based on [KOMA script](#) and can be found online: <https://github.com/novoid/LaTeX-KOMA-template>

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Graz, _____

Date

Signature

Kurzfassung

Der Digitalisierungstrend ist schon seit langer Zeit stark present, aber die Möglichkeiten die es mit sich bringt sind noch nicht vollständig erschöpft. In den letzten Jahren neigt die Industrie immer mehr dazu die kosteneffiziente und skalierbare Lösungen einzusetzen, und versucht immer öfter die Aktivitäten, die nicht teil des Kerngeschäftes sind, auszulagern, besonders die IT-Infrastruktur. Die Cloud-Dienste und das Software as a Service-Modell spielen dabei eine wichtige Rolle. Bei Verwendung von Cloud-Diensten müssen sich die Unternehmen nicht mehr um die Kosten und Instandhaltung von Hardware und Software kümmern, während das Software as a Service-Modell hoch skalierbare und einfach verteilbare Software Lösungen ermöglicht. Das Ziel dieser Arbeit ist zu analysieren wie diese Technologien verwendet werden können um die buchhalterische Abläufe zu vereinfachen indem sie in eine Webanwendung für Buchhaltung integriert werden. Die Arbeit erforscht die typische Anforderungen im Buchhaltungsbereich, und diskutiert die technische Implementation entsprechender Funktionalität der notwendig ist um digitale Handelsdokumente (z.B. Rechnungen) zu bearbeiten und zu speichern, weil diese oft dem Buchhaltung in Papierform zur Verfügung gestellt werden müssen, was sowohl Zeit- als auch Platzaufwendig ist. Ein grosser Teil dieser Arbeit ist fokussiert auf die Implementation, während zusätzlich Quellcode Teile der entwickelten Webanwendung bereitgestellt und diskutiert werden.

Abstract

The trend of digitalization has been present for a long time now, but its possibilities have not yet been utilized to their full potential yet. In the past few years the industry has started to turn to more cost efficient and scalable solutions, and also often tries to outsource the non-core activities, especially the ones concerning the IT infrastructure. The cloud services and the Software as a Service model play a great role in this transformation. By using the cloud services, companies do not have to worry about the cost and maintenance of the hardware and the software anymore, while the Software as a Service model facilitates highly scalable and easily distributable software solutions. This thesis aims to analyze how these technologies can be utilized to simplify the bookkeeping processes by integrating them in a bookkeeping web application. The thesis investigates the typical business requirements in a bookkeeping environment and discusses the technical implementation of the corresponding functionality needed to process and store commercial documents in a digital format, due to the fact that the commercial documents such as invoices often need to be made available to the bookkeeper or accountant in a paper form, which is both space- and time-consuming. The majority of the thesis focuses on the implementation details while additionally providing code pieces taken from the web application that was developed within the scope of this thesis.

Contents

Abstract	v
1 Introduction	1
1.1 Document Management System	3
1.2 Technology Overview	4
1.2.1 Internet and the World Wide Web	4
1.2.2 Hypertext Markup Language	5
1.2.3 Cascading Style Sheets	5
1.2.4 Client-side Scripting with JavaScript	6
1.2.5 Server-side Scripting with PHP	7
1.2.6 Asynchronous Web Applications with AJAX	7
1.3 Cloud Services	8
1.3.1 Dropbox	10
1.3.2 Google Drive	11
1.3.3 Microsoft OneDrive	12
1.4 Bookkeeping	13
1.4.1 Double-entry Bookkeeping	15
1.4.2 Legal Framework	20
1.5 Business Requirements and Integration of Cloud Services	21
1.5.1 Creation of shared Document Repositories	21
1.5.2 Uploading Documents	22
1.5.3 Viewing and Downloading Documents	23
1.5.4 Collaborative Editing of Accounting Data	24
2 Architecture and Functionality	25
2.1 Presentation Layer	25
2.2 Business Logic Layer	28
2.2.1 User Management	29
2.2.2 Repository Management	30

Contents

2.2.3	Utility Services	36
2.3	Data Layer	37
3	Implementation	39
3.1	Database	39
3.1.1	Data Model	39
3.1.2	Database abstraction with PHP Data Objects	51
3.2	User Management	53
3.2.1	Roles	53
3.2.2	Privileges	55
3.2.3	Account Creation	59
3.2.4	Signing In to Application	61
3.2.5	Signing Out of Application	63
3.3	Repository Management	64
3.3.1	Creation of Document Repository	65
3.3.2	Connecting to a Document Repository	66
3.3.3	Repository User Management	68
3.3.4	Accounting Data Fields	71
3.3.5	Accounting Data Layouts	75
3.3.6	Parameter Tables	77
3.4	Document and Accounting Data Management	81
3.4.1	Fetching the contents of a Document Repository	82
3.4.2	Directory Creation	85
3.4.3	Document Upload	86
3.4.4	Viewing the Document	88
3.4.5	Document and Directory Deletion	90
3.4.6	Editing the Accounting Data	91
3.4.7	Versioning of the Accounting Data	93
3.4.8	Creating the Accounting Records	94
3.4.9	Viewing the Books	96
3.4.10	Using the Comment System	98
3.5	Settings	100
3.5.1	General Settings	100
3.5.2	User, Repository and Authorization Overview	101
3.5.3	Application Log	101
4	Conclusion	103

Contents

Bibliography

105

List of Figures

1.1	Dropbox web interface	11
1.2	Microsoft OneDrive web interface	13
1.3	Basic structure of a balance sheet	15
1.4	Basic account structure	16
1.5	Basic structure of the balance sheet accounts	17
1.6	Basic structure of the profit and loss accounts	18
1.7	General outline of the Austrian unified account system (EKR)	19
2.1	OAuth 2.0 authorization process flow	32
3.1	Repository privileges dashboard	58
3.2	Account creation form	60
3.3	Password reset form	63
3.4	Sign out icon	64
3.5	Repository placeholder	65
3.6	Repository dashboard	68
3.7	Repository user management interface within the repository dashboard	69
3.8	Repository user authorization dialog	70
3.9	Repository accounting data fields dashboard	72
3.10	Repository data field creation dialog	73
3.11	Repository accounting data layouts dashboard	75
3.12	Repository accounting data layout creation dialog	76
3.13	Repository parameter tables dashboard	79
3.14	Repository parameter tables creation dialog	80
3.15	Repository page	82
3.16	Directory creation dialog	85
3.17	File upload dialog	87
3.18	Directory deletion	91

List of Figures

3.19	Accounting data panel	92
3.20	Accounting data revisions dialog	93
3.21	Accounting records panel	94
3.22	General ledger on the books page	97
3.23	Comments panel	99
3.24	User settings	100
3.25	User overview	101
3.26	Application log	102

1 Introduction

Bookkeeping concerns itself with recording of every financial transaction within a business (Bragg, 2011). These transactions include, amongst others, payments, purchases and sales. The basic principle states that no record should be made without the existence of the related commercial documentation. The quality of the recorded information about the transactions is not only important because of the potential legal obligation to keep business records, but also because they are summarized into financial reports that can give an important insight into the economic situation of the business. Depending on the size and needs of an organization, the complexity of the bookkeeping process and the recorded information can vary. Bigger companies usually do their own bookkeeping within their accounting departments, but a smaller businesses often use services of an external accountant (Quinn, 2010). Nowadays, almost all bookkeeping processes and tasks are carried out by using accounting applications, but a great deal of commercial documents such as invoices are still being issued only in a paper form, so there is a great effort involved in storing them and making them available to an accountant.

With the rapid progress in information technologies over the last decade, the interest in their utilization within enterprises has also increased immensely. The trends in the information industry have not only made way for the development of new sales and marketing channels (Chao, 2016), but they have also allowed the enterprises to become more adaptive to the constantly changing conditions on the market. Although the information systems have become an integral and irreplaceable part of the business, and much of the business processes are already digitalized, the issuing of digital commercial documents is not yet widely present despite the fact that the necessary technologies have been available for quite a while now.

1 Introduction

In order to tackle this issue, the legislators have undertaken the endeavor to support the digitalization of commercial documents. In the European Union, the foundation for this trend was laid in the Directive 2010/45/EU. Its purpose is to provide the legal guidelines that allow the equal treatment of paper and electronic invoices, and therefore increase the efficiency and decrease the complexity of business transactions within the European Single Market (Bundesministerium für Finanzen, 2017b). The Austrian legislator has implemented the directive with the 2012 Tax Amendment Act, and since 1. July 2013 allows issuing of electronic invoices either as a PDF or a text document, or even as scan of a paper invoice. Such invoices can be then made available either per email or as a download. The prerequisites for a validity of these commercial documents include the authenticity of the origin, content integrity and human readability. The methods and technologies used to ensure the prerequisites are basically freely selectable, so the authenticity and integrity can, for instance, be ensured by using the digital signature. When considering the benefits of electronic invoices over their paper counterpart, the legal retention period also needs to be taken under consideration. In Austria, the invoices must be kept available for 7 years from the date of issue (Bundesministerium für Finanzen, 2017a). The retention period may vary from country to country, but the mere fact that the invoices must be kept for a longer period of time introduces another challenge, namely providing the necessary storage space. Therefore, the fact that the electronic invoices can be easily made available over the internet and do not require practically any physical storage space, makes them a great alternative to a more traditional use of paper invoices.

Due to the low cost and high availability, the cloud storage services have become increasingly popular over the last few years, so they offer themselves as a logical choice when considering the digital storage possibilities. What makes them especially attractive is the fact that through their usage the storage service can be outsourced, so the provision of the necessary infrastructure or security and reliability issues are all being managed by an external provider. In addition to this, a combination of a cloud storage and a web based application can provide a software solution that is accessible solely through the client's web browser, and needs no local installation.

The goal of this thesis is not to reinvent the, often very complex, book-keeping or accounting applications, but to investigate how especially cloud

1.1 Document Management System

storage services in combination with a Software as a Service (SaaS) delivery model can be used to improve the daily bookkeeping processes by providing means to process and store commercial documents. It will also be investigated which interfaces modern cloud storage services offer in order to integrate them into a web application. To this purpose, the necessary business requirements will be analyzed, and a web application developed, which demonstrates, how cloud service functionality can be integrated to fulfill them. The application will be designed primarily to serve as a digital document archive and a interface between the accountant and the client, and in its basic form is not meant to replace a fully fledged bookkeeping application.

1.1 Document Management System

Document management system (DMS) is, in a context of information technology, a software application that provides functionality needed to store, search and process digital documents. The documents that are issued in a paper form can also be introduced into the system, but they have to be digitalized by using a scanner (Brooks, 2004), or some other method first.

When considering a web based DMS, the overall required functionality can be clearly divided between the document management and the web technologies (Balasubramanian and Bashian, 1998). The requirements for the efficient document management include managing of a large amount of documents, supporting of roles, provision of access control, recording of attributes as well as enabling of the workflow and version control, while the web technologies take responsibility for the provisioning of the user interface including the navigation, delivery of multiple media formats and provisioning of the attribute searching functionality.

Ginsburg divides the processes within a DMS into the foreground and the background ones (Ginsburg, 2000). The foreground processes are defined by the actors and their actions, while the data processing functionality resides in the background. The acting roles can be the author, the editor and the reader. The author typically performs actions such as creation, upload and update of the documents, the editor can give a clearing for the

1 Introduction

documents before publishing, and the reader usually performs actions such as searching for the document and reading it, as well as adding annotations and metadata. The document metadata is a very important concept in the document management because it adds additional information and thereby enhances the data content, while the annotation system can be used to provide appraisals or additional notes.

1.2 Technology Overview

This chapter aims to provide an overview of the terms that are used throughout the thesis, and give an insight into technologies that was used for implementation.

1.2.1 Internet and the World Wide Web

Internet is a global computer network of networks in which the data is interchanged by using the Internet Protocol Suite (W3C, 2017a). The two most important protocols within the suite are the Transmission Control Protocol and the Internet Protocol (TCP/IP). The World Wide Web (WWW), on the other hand, is only one of many services available on the internet. It can be defined as an information space in which the resources are identified by the Uniform Resource Identifiers (URI) and interconnected by means of hypertext links (Jacobs and Walsh, 2004). A resource can be anything that is identifiable by a URI, for instance, a web page or an image. The most important protocol that is used for communication purposes and data transfer on the WWW is the Hypertext Transfer Protocol.

The Hypertext Transfer Protocol, or shorter HTTP, is a stateless protocol for distributed hypertext information systems (Fielding and Reschke, 2014). The protocol is based on the request-response principle which operates by exchanging messages between the HTTP client and HTTP server over a previously established connection. Being stateless means that there is no interdependency between the requests, so each request is handled separately

and without regard to any of the previous ones. Typical HTTP communication consists of a client (e.g. web browser) sending a request message to the server in order to retrieve a resource identified by a Uniform Resource Identifier. The server receives the request and sends the requested resource to the client in response.

This thesis uses the term web page (or just page) to refer to any HTML document, from a simple static web page to an interactive web application that is using both client-side as well as server-side scripting.

1.2.2 Hypertext Markup Language

The Hypertext Markup Language (HTML) is the standard markup language when it comes to creating web pages and web applications. It is used to define the structure of a document by using markup. Each HTML document consists of a tree of elements, and each element is denoted with a start tag and an end tag. The available elements provide means to include text, tables, lists, images, forms and even video and audio (W3C, 2017b). The documents can be linked to other documents or resources by means of hypertext links. Once the HTML document is received from the server, the web browser displays it's content to the user and parses it's code into a Document Object Model (DOM) tree, which is an in-memory representation of the document (Faulkner et al., 2016). The usage of the DOM interface is very crucial when developing complex web applications, this topic will be discussed more detailed in chapters 1.2.4 and 1.2.6.

1.2.3 Cascading Style Sheets

Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of an HTML document. More precisely, it's main goal is to provide a style for structured documents and thus separate the structure of the document from it's presentation (Bos, 2016). With the rapid increase of different devices that are nowadays used to access the web, it has become increasingly important to provide an appropriate presentation for each of them, especially in regard to the differences in screen sizes and resolution,

1 Introduction

and this is exactly what CSS facilitates (W3C, 2017b). By using a wide variety of CSS selectors each element of the HTML document can be individually styled. The CSS properties that can be modified include layout (e.g. position), colors (e.g. background-color), and font properties.

Together with HTML, CSS forms a solid basis for creating web applications, but in order to implement the advanced functionality, addition of both client-side and server-side scripting is needed.

1.2.4 Client-side Scripting with JavaScript

With Dynamic HTML (DHTML), a concept was introduced that uses a set of existing technologies in order to allow creation of dynamic web content. This concept usually consist of HTML, CSS, ECMAScript - a scripting language widely known as JavaScript, and the Document Object Model. The Hypertext Markup Language is used for the page layout, Cascading Style Sheets for presentation, JavaScript as a client-side scripting language, and the Document Object Model interface allows the dynamic manipulation of the page content, layout and presentation (Goodman, 2006).

One of the key technologies within DHTML is the scripting language, and the JavaScript is the most commonly used one (W3C, 2017c). It is an interpreted high-level scripting language that is also known as ECMAScript due to the fact that the standardization of the language was submitted to European Computer Manufacturer's Association (ECMA) (Flanagan, 2011). Being a client-side scripting language, the Javascript code is sent by the server alongside HTML document and executed on the client-side i.e. in the user's web browser. The code itself can be embedded in a HTML document, or loaded from the external source files. One of the most important features of JavaScript is the ability to register an event handler function that is bound to a certain event such as a mouse click or a key press. So by using the abilities of a client-side scripting language such as JavaScript in combination with DOM and CSS, a new functionality can be added to static HTML documents making them more interactive. This facilitates creation of web applications with a responsive and customizable user interface.

1.2.5 Server-side Scripting with PHP

Scripts executed on the client-side do not always provide the necessary functionality needed in a web application, such as access to the file system or a database. It is only the feature set of the server-side scripting languages that allows the creation of web applications with such level of complexity. As the name implies, the code of the server-side scripting languages is executed on the server and stays hidden from the user at all times, only the result is sent to the client once the server is done with processing. This result can be anything from a HTML or XML document to a graphic or a PDF file.

PHP, a recursive acronym of Hypertext Preprocessor, is an open source scripting language. It was initially developed by Rasmus Lerdorf in 1994, and has been improved and expanded ever since. The language is most commonly used for server-side scripting i.e. development of web applications, but can also be used for command line scripting or development of desktop applications by using the GIMP-Toolkit (GTK). A major advantage is the fact that PHP is both cross-platform compatible as well as supported by many modern servers such as Apache or Microsoft's Internet Information Server. It can be installed either as a server module where supported, or executed by using the Common Gateway Interface (CGI) and FastCGI respectively (Ahour, Betz, and Dovgal, 2017).

Some of the PHP's most important features include a support for a wide range of databases, including all the major ones such as MySQL, Oracle or PostgreSQL, and an existence of a standard PHP library (SPL). The standard PHP library contains a collection of classes and interfaces that implement often used functionality such as database layer abstraction and error handling (Tatroe, MacIntyre, and Lerdorf, 2013).

1.2.6 Asynchronous Web Applications with AJAX

Both client- and server-side scripting can be combined together in order to provide more functionality in a web application and a better user experience. One potential drawback of combining these two techniques is the fact that

1 Introduction

each time a script needs to be executed on the server-side, the client has to send a new HTTP request, causing the page the user is currently viewing to reload. This problem can be tackled by sending the request to the server asynchronously. The technology, or more precisely, a set of technologies that allows this kind of approach is called AJAX, which is short for asynchronous JavaScript and XML.

The term AJAX was originally devised by Jesse James Garrett, and it initially contained following technologies (Powers, 2007):

- Extensible HTML (XHTML) for page structure and Cascading Style Sheets for presentation
- Extensible Markup Language (XML) as a data format and Extensible Stylesheet Language Transformation (XSLT) for data presentation
- Document Object Model for dynamic manipulation of page elements
- XMLHttpRequest object for the client-server communication
- JavaScript as a client-side scripting language that binds all of the components together

The central element is the XMLHttpRequest object. It allows the client to send the HTTP request asynchronously i.e. in the background, without blocking the user interaction and having to reload the entire page once the HTTP response comes back from the server. The other important component is the JavaScript. On one side, it is used for client-server communication by means of XMLHttpRequest, and on the other side, by using JavaScript and DOM manipulation, each of the elements on the page can be populated individually and on demand, which makes the web application more effective and more responsive. The last core component of AJAX is the data format. Although XML was initially designated as the data format, there are other formats that can be used as well, such as JavaScript Object Notation (JSON) or HTML.

1.3 Cloud Services

Cloud computing can be considered to be a result of the advancement in the technology, and the change in the user behavior and expectations (Barton,

1.3 Cloud Services

2014). It has become very popular over the last decade, not only because of the fact that the technology has allowed the users to become more mobile so they prefer to have mobile access to their data and applications, but also because of the demand for more scalable solutions that do not require much investment in the infrastructure. Therefore, the key characteristics of the cloud computing include outsourcing of the IT resources and their usage as a service, easy online access, high scalability and usage-based pricing.

Generally, cloud computing services can be divided into three service levels regarding the resource that is being provided to the client, and four delivery models that define how that resource is being provided.

A widely accepted cloud computing service model consists of three levels (Barton, 2014):

1. Software as a Service
2. Platform as a Service
3. Infrastructure as a Service

Infrastructure as a Service (IaaS) represents the bottom level, which consists of services that provide infrastructure resources such as processing, storage and network capacities, while the client has a freedom to decide which operating system and applications are going to be deployed on top of the provided service. The availability of the resources, which can be defined in the service level agreement (SLA), lies within the responsibility of the service provider. The middle level is represented through Platform as a Service (PaaS). This service model aims to provide the client with the software environment that can be used for application development or customization, while the top level model - Software as a Service (SaaS), provides the client with an access to a standardized application which is provided as a service and charged for a particular time-period or on a per-use basis.

The cloud service delivery, on the other hand, can be divided into four models (Barton, 2014):

1. **Public Cloud** - the cloud service infrastructure is both owned and operated by an external service provider.
2. **Private Cloud** - the cloud service is in-sourced and operated by the service user. Alternatively can either infrastructure be outsourced

1 Introduction

(Hosted Private Cloud) or both infrastructure and operation (Managed Private Cloud), but the access is usually very restricted.

3. **Hybrid Cloud** - a combination of multiple private and public cloud infrastructures.
4. **Virtual Private Cloud** - a public cloud which is configured for an individual user in order to provide a certain degree of isolation, for instance by using a VPN connection.

The cloud services used for the implementation of the web application within the scope of this thesis provide storage resources through the IaaS model, and are being delivered through a public cloud service. The implemented web application, on the other hand, can be delivered by using the SaaS model, or more precisely, by providing the access to the application through a web browser.

1.3.1 Dropbox

Dropbox is a cloud storage service operated by the company Dropbox, Inc. (Dropbox Inc., 2017a). The service offers both file system synchronization for the offline access as well as online access over the web browser (Figure 1.1). In case of the file system synchronization, the Dropbox client application needs to be downloaded and installed locally. The installed application creates a designated folder in the local file system, and every file that is moved into that folder gets automatically synchronized with the cloud storage. This way the user can easily synchronize his or her files over multiple devices. Additionally, the service also offers a version history in case a file was deleted or it needs to be restored to a previous version.

For the purpose of integration of the cloud service into existing applications, the Dropbox provides application programming interface (API) and software development kits (SDK) for multiple programming languages such as Java, Python and Objective-C. Additionally there are also third-party libraries available that support other programming languages as well (Dropbox Inc., 2017b). By using the API, the applications can use Dropbox functionality such as file upload, download, deletion, folder listing and other. For the authentication of the third-party applications with the Dropbox service, the

1.3 Cloud Services

OAuth2 protocol is used. The OAuth2 authorization process and the usage of the Dropbox application programming interface (API) will be discussed in more detail in the chapter 3.

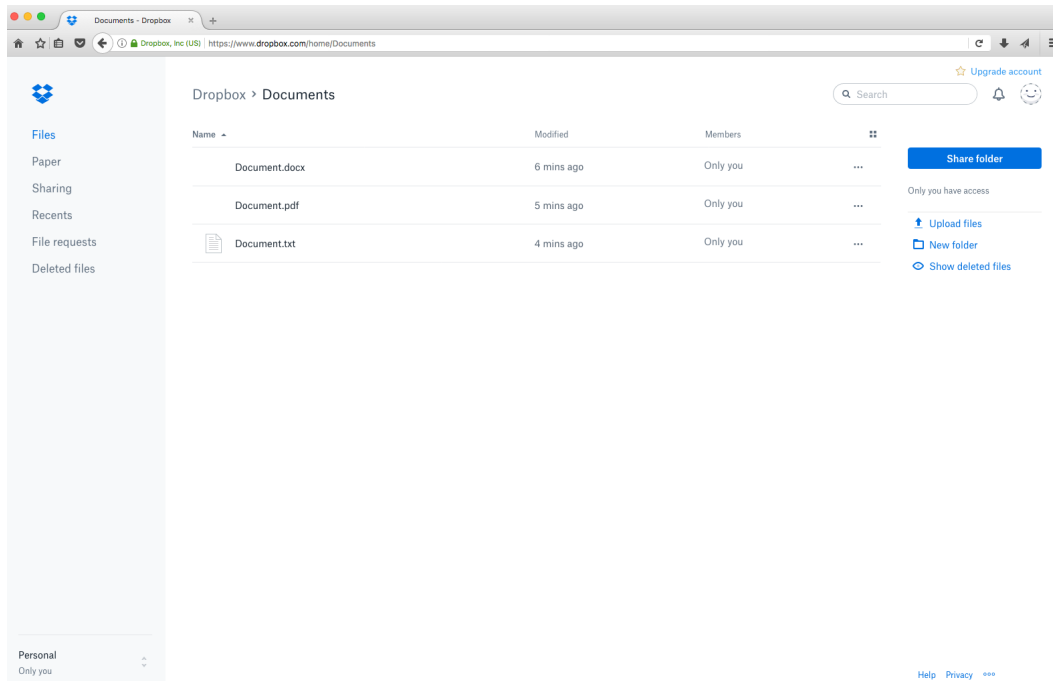


Figure 1.1: Dropbox web interface

1.3.2 Google Drive

Drive is a cloud storage service offered by Google (Google, 2017b). Just as the service from Dropbox, the Google Drive offers both file system synchronization for offline access by using the client application, as well as an online web interface for the file management. By using the synchronization, the uploaded files become available across all of the connected devices. Drive also offers file versioning, which allows the user to revert to a previous file version if needed. Additionally, Google Drive features an intelligent search functionality that can recognize objects in the images and text in the scanned documents.

1 Introduction

One further interesting feature is the possibility to scan documents with a mobile device. A photo of a document that is taken with the mobile device gets hereby instantly uploaded to the cloud storage and converted to a PDF format.

Google also extends the functionality of the Drive by seamlessly integrating the Google applications Docs, Sheets and Slides, which allow creation and collaborative editing of documents, spreadsheets and presentations which are then automatically saved to the cloud storage.

For the integration of the Drive into existing third-party applications, Google offers API client libraries for several programming languages including Java, Python, Objective-C and PHP (Google, 2017a). The authentication of the application is conducted by using the OAuth2 process.

1.3.3 Microsoft OneDrive

OneDrive is a cloud storage service from Microsoft (Microsoft, 2017a). For the upload of the files into the cloud storage, Microsoft offers local file system synchronization much like Dropbox by using the desktop application currently available for Windows and OS X, and an online web access (Figure 1.2) where the files can also be uploaded by a drag and drop. For the offline integration, the OneDrive application uses a dedicated folder. The contents of the local dedicated folder can be synchronized across multiple devices.

1.4 Bookkeeping

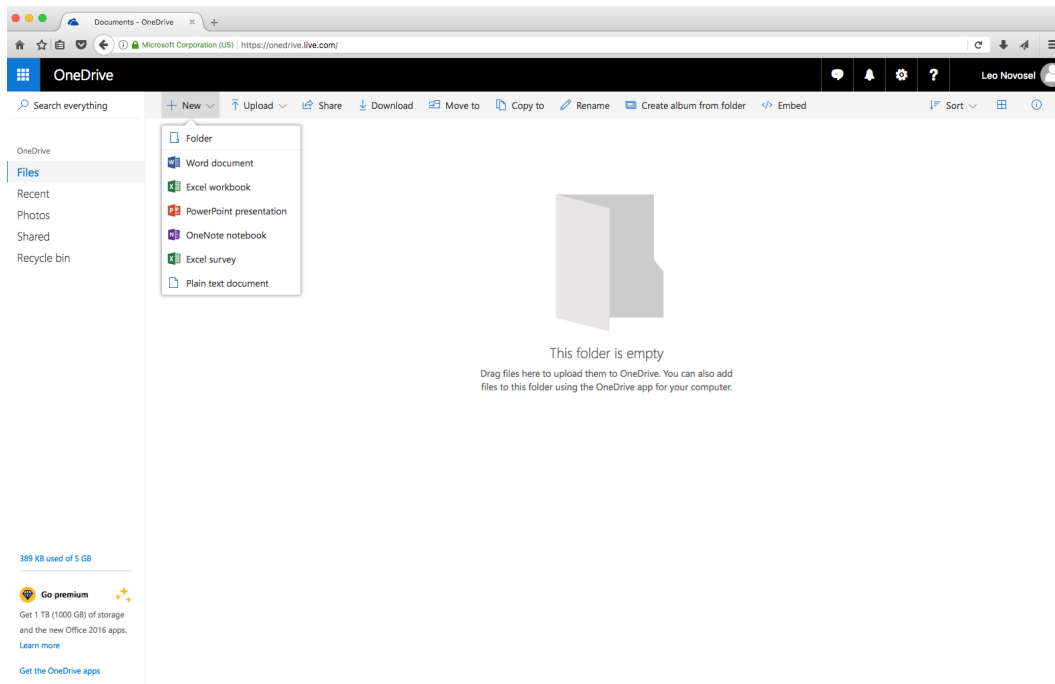


Figure 1.2: Microsoft OneDrive web interface

For OneDrive integration purposes, Microsoft offers both Graph SDKs for programming languages such as PHP, Python and Ruby, as well as a REST API (Microsoft, 2017b). The authentication of the third-party applications can be achieved by using the OAuth2 process.

1.4 Bookkeeping

Bookkeeping (also financial accounting or external accounting) is first and foremost dedicated to providing the information to the interested parties that are external to the company, with two of the most important ones being the tax authorities and the creditors (Bauer, 2017). The creditors will want to make sure that the provided capital will be returned, while the tax authorities, amongst other things, use the information about the company's profit as a basis for the tax calculation.

1 Introduction

The three constituent parts of the external accounting are the record keeping, the inventory and the annual financial statement.

Being a part of the external accounting, the record keeping needs to make sure that the provided information is uniform, and in accordance with the legal regulations that define the structure and content of the financial reports. According to section 190 of the Austrian Commercial Code (UGB), the companies are required to keep records of each individual business transaction in accordance to generally accepted accounting principles, and in such a way so that each business transaction can be traceable regarding to its origin and processing. That means that the records need to be both chronologically and systematically organized.

The inventory is the listing of all assets (both fixed and current) as well as all debts at one specific point in time. The legal framework regarding the inventory is found in the section 191 UGB. It states, amongst others, that the inventory must be created at the end of every fiscal year.

The annual financial statement is the third and last part of the external accounting. Within its context, the Austrian legislator requires the preparation of a balance sheet and a profit and loss statement (section 193 UGB). A balance sheet is a summary of all the company's assets on one side and capital (liabilities and owner's equity) on the other (Figure 1.3), while the profit and loss statement provides information regarding all the revenues and expenses within a fiscal year. In addition to balance sheet and a profit and loss statement, corporations must also provide an annex and a management report. This does not apply to companies with an annual revenue less than 700.000,- Euro (section 189 UGB).

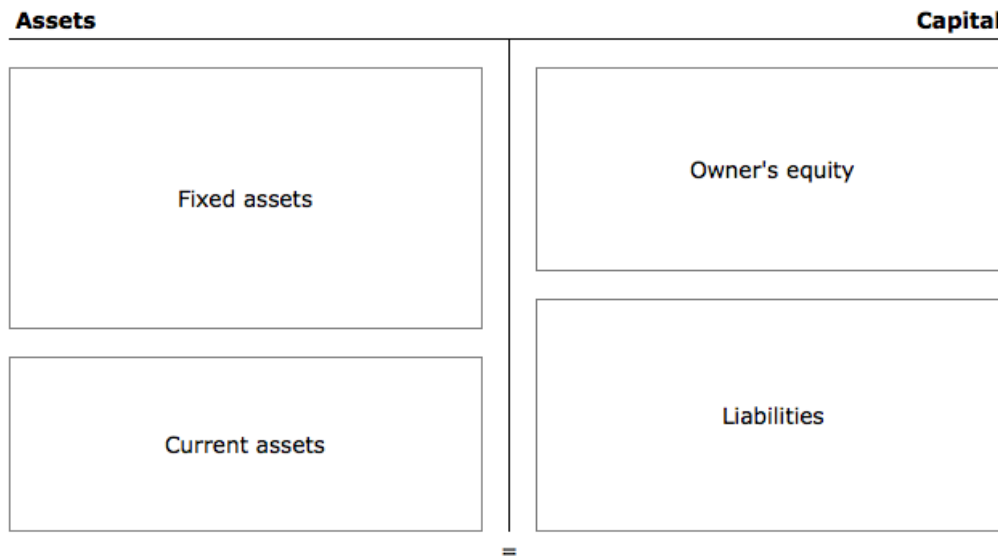


Figure 1.3: Basic structure of a balance sheet

1.4.1 Double-entry Bookkeeping

The double-entry bookkeeping is a method that allows the representation of the information required by the external accounting. The method requires that per each business transaction at least two accounting records are made, a debit entry to one account, and a corresponding credit entry to another account. As a result, the sum of credits across all accounts must always be equal to the sum of debits at all times. The accounting record contains the information regarding the accounts that have been affected by the business transaction as well as the individual amounts that have been booked to each of these accounts.

Additionally, each business transaction must also be listed twice, once in the journal, which is chronologically sorted, and once in the general ledger, which is systematically organized into accounts. A business transaction, in this context, is considered to be any transaction that changes the structure and value of assets, liabilities or owner's equity.

1 Introduction

Also, the double-entry bookkeeping allows for the profit to be calculated in two ways, either by using the profit and loss statement, or by calculating the equity difference between the closing balance and the opening balance and adding the private withdrawals while deducting the deposits. The profit calculation period is the fiscal year, which usually corresponds to the calendar year (i.e. 12 months).

One of the central concepts of the double-entry bookkeeping is the account. The account can be considered to be a two sided calculation field. This account form is called the T-account, and is used for educational purposes only. The amounts that are to be added are recorded on one side, while the amounts that are to be subtracted are recorded on the other. Hence, the left side is the debit side, and the right side is the credit side (Figure 1.4). The difference between the sum of the debit amounts and the sum of the credit amounts is called the balance. The credit balance is placed on the debit side, while the debit balance is placed on the credit side.

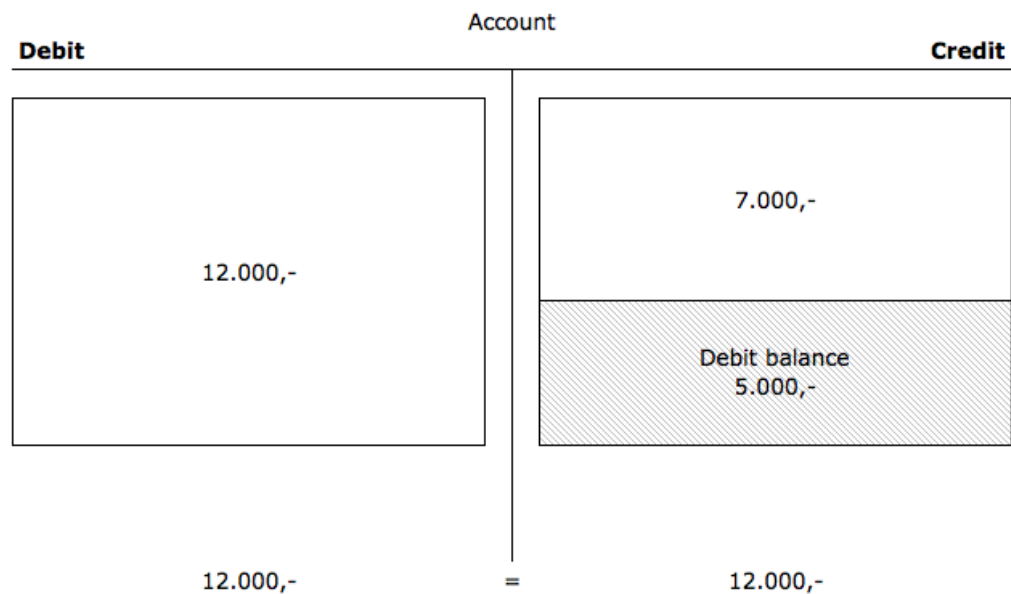


Figure 1.4: Basic account structure

The double-entry bookkeeping differentiates between two types of accounts:

1.4 Bookkeeping

the balance sheet accounts, and the profit and loss accounts. The balance sheet accounts are accounts that are derived from the individual positions in the balance sheet. The profit and loss accounts, on the other side, are used to represent all the different factors that have influence on the owner's equity and are therefore indirectly derived from the owner's equity position in the balance sheet (Bauer, 2017).

If the balance sheet account is derived from the assets, then the opening assets and the assets increases are to be recorded to the debit side of the account, while the assets decreases and the final assets are to be recorded to the credit side. If the account is derived from the liabilities then the records should be made exactly the other way around (Figure 1.5).

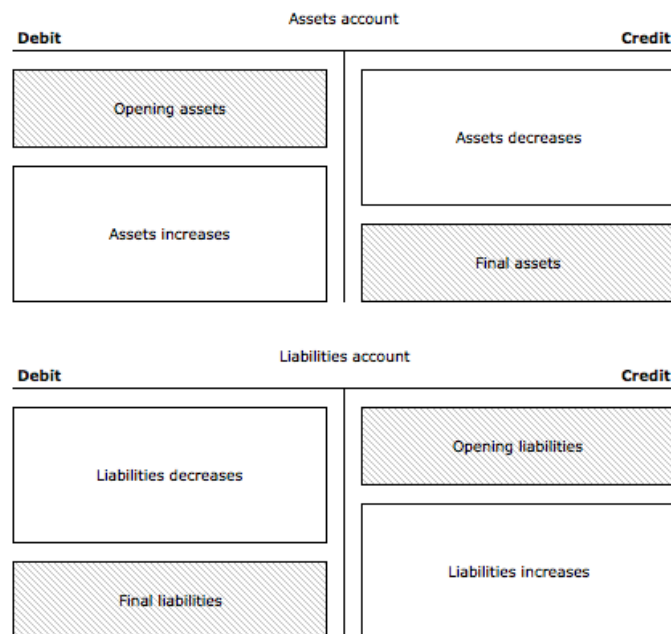


Figure 1.5: Basic structure of the balance sheet accounts

The profit and loss accounts are also divided into two groups: the expense accounts and the revenue accounts. The expenses are recorded to the debit side of an expense account, while the revenues are recorded to the credit side of a revenue account (Figure 1.6).

1 Introduction

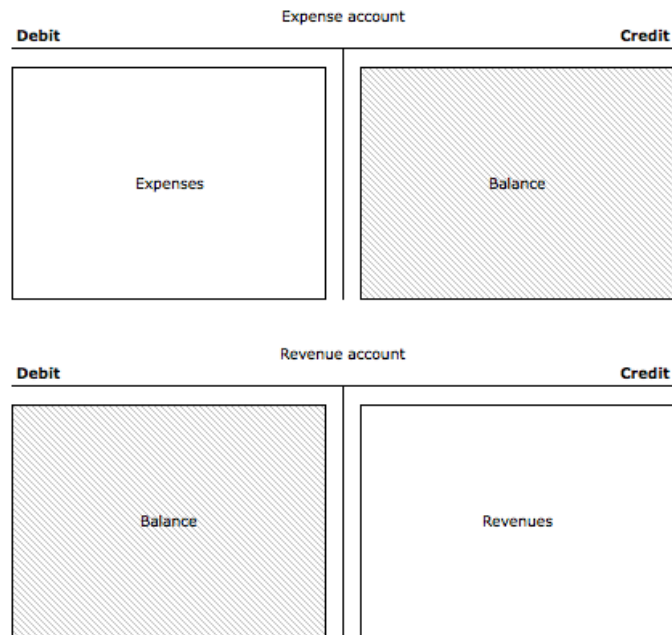


Figure 1.6: Basic structure of the profit and loss accounts

In order to unify and simplify the bookkeeping process, the Austrian unified account system (EKR) provides a proposition about how accounts can be uniformly designated and organized, hence, the EKR is a basic proposition on how the company's chart of accounts might look like. Basically, according to EKR, the accounts are divided into 10 account classes ranging from 0 to 9, and each class is further divided into 10 account groups ranging from 00 to 99 (Figure 1.7).

Class		
0	Assets accounts	Balance sheet accounts
1		
2		
3	Liabilities accounts	
4	Revenue accounts	Profit and loss accounts
5	Expense accounts	
6		
7		
8	Net financial income	
9	Owner's equity accounts, etc.	

Figure 1.7: General outline of the Austrian unified account system (EKR)

Generally, the process of Double-entry bookkeeping can be divided into five steps (Bauer, 2017):

1. **Creation of an inventory and the opening balance sheet** - The inventory contains all assets and debts, and is used to calculate the owner's equity by deducting the debts from the assets. It also forms the basis for the opening balance sheet as well as the opening accounting records. The opening balance sheet contains the assets, liabilities and owner's equity that are present in the beginning of the accounting period.
2. **Creation of the opening accounting records** - In order to be able to record the occurring business transactions within the accounting period, the positions in the opening balance sheet have to be divided into individual balance sheet accounts. If the balance sheet contains assets or liabilities at the beginning of the accounting period, than the corresponding opening account records have to be created.

1 Introduction

3. **Creation of the accounting records for the occurring business transactions** - During the accounting period, all business transactions have to be recorded. The Double-entry bookkeeping requires that at least two accounting records are made per business transaction, a debit one and a credit one. One such business transaction can affect either balance sheet accounts only (i.e. owner's equity is not affected), or one balance sheet account and one profit and loss account (i.e. the profit and loss statement is affected, and by that also the owner's equity).
4. **Creation of the final accounting records** - At the end of the accounting period all balance sheet accounts have to be closed off to the closing balance account. These records form the basis for the opening balance sheet in the next accounting period. The profit and loss accounts, on the other hand, are closed off to the special profit and loss account (P&L account). The balance of this P&L account is then transferred to the owner's equity account and then closed off to the closing balance account.
5. **Creation of the annual financial statement** - Creation of the annual financial statement containing the balance sheet and the profit and loss statement as required in the section 193 UGB.

1.4.2 Legal Framework

Generally, there are two major legal frameworks that regulate the obligatory elements of external accounting, one regarding the commercial regulations, and other regarding the tax regulations (Bauer, 2017).

The commercial regulations in Austria are primarily defined in the Austrian Commercial Code (UGB). According to the section 189 of the UGB all of the corporations and companies with an annual revenue higher than 700.000,- Euro are required to keep records.

The tax regulations concerning the external accounting, can be found primarily in the Austrian Federal Fiscal Code (BAO), the Austrian Income Tax Act (EStG) and the Austrian Turnover Tax Act (UStG).

One of the most important elements of record keeping is the receipt. The receipt is a business document that forms the foundation for an accounting

1.5 Business Requirements and Integration of Cloud Services

record, so in accordance with the basic accounting principle there should be no records made without the existence of the corresponding receipts.

The receipts can be divided into two categories: internal- and external receipts. The internal receipts originate from within the company (e.g. inventory records), while the external receipts originate from the external parties (e.g. invoices, bank statement etc.).

Due to the fact that the receipts substantiate every business transaction within the company, they need to be both well documented and well kept. The retention period for receipts in Austria, according to section 132 BAO, is 7 years.

1.5 Business Requirements and Integration of Cloud Services

In this chapter the business requirements will be analyzed, and elaborated which cloud service functionality can be used in order to meet them. The concrete implementation of the functionality and the user interface will be discussed in the chapter 3.

1.5.1 Creation of shared Document Repositories

For the purpose of storing documents, the client needs to be able to create one or more document repositories. A repository is, in this case, a web application's internal representation of the cloud storage.

In order to define a repository within the web application, the client will need to perform three steps:

1. Select a cloud storage service (i.e. Dropbox, Google Drive or OneDrive)
2. Enter a name for the new repository
3. Authenticate the web application

1 Introduction

The authentication will be conducted by using the OAuth 2.0 authentication scheme, being that all three cloud services considered in this thesis support it.

Once the application is authenticated, the user will see a new repository tile in the overview page. By selecting a repository in the overview, the web application will establish a connection to the underlying cloud service, and the user will be able to access the documents from within the repository page. The design of the overview- and the repository page will be discussed in the chapter 3.

Furthermore, the client will have to have a possibility to give clearing to other users with appropriate roles (e.g. the accountant), in order for them to be able to access the repository from within the web application.

1.5.2 Uploading Documents

As already mentioned earlier, a lot of commercial documents (e.g. invoices) are nowadays still issued in a paper form only. In order for the accountant to be able to process these documents and make appropriate records, the client needs to make them available first. One possibility would be to scan the documents and save them either in a Portable Document Format (PDF), or in one of the popular image formats such as JPEG. The advantage of this method is that the documents that are digitalized in this manner still maintain their validity, but are much easier to transfer and store for archiving purposes. The documents that are already issued in one of the supported digital formats can be uploaded directly, and without any need for a transformation.

The upload functionality for the client is given both through the web application, as well as through the regular upload channels supported by the cloud storage service. The accountant, on the other hand, can upload the documents solely through the web application, which previously needs to be authorized to access the client's cloud service account as described in chapter 1.5.1. Although the upload functionality is given through the regular cloud storage upload channels such as web interface or a local file system synchronization, this way of uploading document raises one critical issue. In

1.5 Business Requirements and Integration of Cloud Services

order to be able to keep track of a document even if it is moved or renamed, the web application will have to assign a unique identifier to the containing file. One possibility would be to generate a unique id and rename the file during the upload. Additionally the unique id can be associated with a calculated checksum of the file and its other properties such as the time of last modification. By using this approach the files would remain uniquely identifiable within the application even if they are moved to a different cloud storage.

For browsing and uploading purposes, the web application's user interface will provide a simple file manager that will represent the file system structure of the cloud storage repository. By using the file manager, the user will be able to navigate the file system, create folders if needed, and upload documents. The extent of the functionality that is provided to each user is defined through their role within the system. There are at least two roles that will initially need to be defined: the client role and the accountant role, but further roles can be added later if the need for other combination of granted privileges emerges. Roles and privileges will be discussed in the chapters 3.2.1 and 3.2.2, respectively.

1.5.3 Viewing and Downloading Documents

Once the documents have been uploaded to the repository, they are automatically visible both to the accountant and the client. In order for accountant to record the transaction, the commercial document needs to be clearly readable. With the digitally generated documents (e.g. invoices) this should not be a problem, but if the uploaded document is a scan of a paper document then it needs to be ensured that the document is readable, for this is also a prerequisite for its validity.

The uploaded documents will be made available for viewing within the web application through the cloud service's download functionality. The user can navigate to a wanted document by using the file manager, and once the document is selected, it will automatically be downloaded and displayed. For this purpose a PDF viewer will be integrated into the user interface, so an external application will not be needed. This way the user can have

1 Introduction

the file manager, the current document, and the transaction record data all displayed within a single view. The privilege of viewing the documents will be granted to both client as well as accountant role.

1.5.4 Collaborative Editing of Accounting Data

Accounting data defines which information about the business transaction are to be recorded. These can, for instance, include a unique reference number, issue date, affected account, amount, and other. In order for the web application to be able to adapt to client's needs, the accountant will have a possibility to define the extent of the information that can be recorded on a per client basis. This means that the application needs to provide a user interface that allows definition and creation of data fields that are stored in the database. In addition to this, a possibility to create views will be provided in order to be able to sort the data fields according to a preferred workflow, or not display certain fields at all if needed. For parameterization purposes, the accountant will be able to create parameter tables that are displayed as dropdown elements in the user interface. This will prevent the user from entering a possibly invalid information into accounting data field.

When the document is selected by the accountant for processing, the accounting data fields will appear in the user interface beside the document. The privilege to edit the data can be defined on a per role basis, and will be granted to the accountant role automatically. If there are more accounting clerks responsible for a single client, they must be individually cleared to access the repository as described in the chapter [1.5.1](#).

Additionally a simple commenting system will be implemented so that the accountant- client communication can occur on a per document basis. The commenting system can, for instance, be used by customer in order to leave a comment for the accountant, or by the accountant to consult the client if questions relating to document arise.

2 Architecture and Functionality

In this chapter the application architecture and the web services provided through the Remote Procedure Call (RPC) interface of the web application will be introduced and discussed.

The web application developed within the scope of this thesis is implemented using the multi-tier architecture. It is comprised of three tiers or layers: the presentation layer, the business logic layer and the data layer.

2.1 Presentation Layer

The presentation layer provides the interface for the user through a web browser. The user interface is implemented using the dynamic HTML pages that use AJAX requests in order to fetch the data from the server (i.e. Business Logic Layer). The asynchronous requests use the JSON data format in order to communicate with the web application's JSON-RPC interface.

Each request is implemented by using the jQuery's ajax object that provides the necessary properties, functions and methods needed for the asynchronous communication. jQuery is a cross-browser JavaScript library that encapsulates functionality such as DOM traversal and manipulation, and provides its own API.

For instance, in order to allow the user to browse through the repository content, the application first attaches an event handler to the DOM element that is representing the repository directory in the user interface. As shown in the listing 2.1, the application attaches a function to handle a click event on the table row HTML element.

2 Architecture and Functionality

Listing 2.1: Attaching an event handler to a DOM element

```
1 $(document).on('click', LNAPP.gui.utils.id(LNAPP.ELEM_REPOSITORY_CONTENT_TABLE) + ' tr',
2     function () {
3         LNAPP.gui.repository.selectItem($(this).closest('tr'));
4         if (LNAPP.gui.repository.currentItemIsFile()) {
5             LNAPP.xhr.download(LNAPP.gui.repository.getCurrentItemPath(), LNAPP.gui.repository.
6                 getCurrentItemName());
7         } else {
8             LNAPP.xhr.listDirectory(LNAPP.gui.repository.getCurrentItemPath());
9         }
10    });
```

When the `click` event occurs, the function checks whether the HTML table row represents a file or a directory, and if the user performed a mouse click on a directory, the function `LNAPP.xhr.listDirectory` will be called in order to present the contents of the chosen cloud storage directory to the user (Listing 2.2). The function `LNAPP.xhr.listDirectory` first generates a JSON-RPC request and then sends it to the server.

Listing 2.2: Listing for the function `LNAPP.xhr.listDirectory`

```
1 LNAPP.xhr.listDirectory = function (path) {
2     'use strict';
3     if (null === path) {
4         return;
5     }
6
7     var id = LNAPP.xhr.generateRPCRequestId(),
8         data = JSON.stringify({
9             'jsonrpc': LNAPP.JSON_RPC_VERSION,
10            'method': LNAPP.RPC_METHOD_REPOSITORY_LIST_DIRECTORY,
11            'id': id,
12            'params': {'path' : String(path)}
13        });
14
15    $.ajax({
16        type: LNAPP.REQUEST_METHOD_POST,
17        url: LNAPP.RPC_SERVER,
18        data: data,
19        success: function (data) {
20            try {
21                var response = JSON.parse(data),
22                    message;
23                if (!LNAPP.xhr.validateRPCRequestId(id, response)) {
24                    return;
25                }
26
27                message = LNAPP.xhr.checkRPCResponse(response);
28                if (null !== message) {
29                    if (LNAPP.RC_INVALID_SESSION === message.code) {
30                        window.location.href = LNAPP.PAGE_INDEX;
31                        return;
32                    }
33
34                    if (LNAPP.RC_INVALID_ACCESS_TOKEN === message.code) {
35                        window.location.href = LNAPP.PAGE_OVERVIEW + '?token=false';
36                        return;
37                    }
38                }
39                LNAPP.gui.actions.displayMessage(message.text);
40                return;
41            }
42        }
43    });
44}
```


2.1 Presentation Layer

```
41     }
42
43     if (response.hasOwnProperty('result')) {
44         if (response.result.hasOwnProperty('message') && 'AI1010' === response.
result.code) {
45             LNAPP.gui.repository.setCurrentDirectory(response.result.path);
46
47             // Clear the workspace
48             LNAPP.gui.actions.clearDocument();
49             LNAPP.gui.actions.clearAccountingData();
50             LNAPP.gui.actions.clearAccountingRecords();
51             LNAPP.gui.actions.clearComments();
52
53             // Reset document ID
54             $('#document_id').val(null);
55
56             // Disable toolbar buttons
57             $('#toolbar-button').css('pointer-events', 'none');
58
59             // Set the breadcrumbs and display the directory listing
60             $('#application-header-title').html(response.result.breadcrumbs);
61             $('#repository-content').html(LNAPP.gui.actions.generateDirectoryListing
(JSON.parse(response.result.directory_listing)));
62         } else {
63             LNAPP.gui.actions.displayMessage(response.result.message);
64         }
65     }
66     } catch (exception) {
67         LNAPP.gui.utils.logError('app_ui.js [LNAPP.xhr.listDirectory]: ' + exception.
message);
68     }
69 },
70 beforeSend: function () {
71     LNAPP.gui.overlay.displayOverlay(LNAPP.ELEM_APP_CONTENT_OVERLAY);
72     LNAPP.gui.actions.displayBusy(LNAPP.ELEM_APP_BUSY);
73 },
74 complete: function () {
75     LNAPP.gui.overlay.hideOverlay(LNAPP.ELEM_APP_CONTENT_OVERLAY);
76     LNAPP.gui.actions.hideBusy(LNAPP.ELEM_APP_BUSY);
77 },
78 error: function () {
79     LNAPP.gui.actions.handleXHRError();
80 }
81 });
82 };
```

Once the server has processed the request, it sends a response back to the client where the raw JSON data is properly formatted so that it can be inserted into the HTML page. In case of the directory listing, this occurs in the `LNAPP.gui.actions.generateDirectoryListing` function (Listing 2.3).

Listing 2.3: Listing for the function `LNAPP.gui.actions.generateDirectoryListing`

```
1 LNAPP.gui.actions.generateDirectoryListing = function (items) {
2     'use strict';
3     var html = '',
4         index,
5         id,
6         actions;
7
8     html += '<div class="repository-dashboard-title" style="text-align: left; width: 100%;
margin-left: 0px; margin-right: 0px;">REPOSITORY</div>';
9     html += '<table id="repository-content-table">';
10    for (index = 0; index < items.length; index += 1) {
```

2 Architecture and Functionality

```
11     id = '';
12     actions = '';
13     if ('..' !== items[index].name) {
14         id = ' id="' + items[index].path + '"';
15         actions = '<button class="button-action" title="Delete">Delete</button>';
16     }
17
18     if (items[index].is_dir) {
19         html += '<tr id="D_' + items[index].path + '>';
20         html += '<td class="selected-item-indicator"></td>';
21         html += '<td class="repository-item">';
22         if ('..' === items[index].name) {
23             html += '<div class="item-icon icon-parent"></div>';
24         } else {
25             html += '<div class="item-icon icon-folder"></div>';
26         }
27         html += '<div class="item-info">';
28         html += '<div class="item-top">' + items[index].name + '</div>';
29         html += '<div class="item-bottom">';
30         html += '<div class="item-status">' + items[index].size + '</div>';
31         html += '</div>';
32         html += '</div>';
33         html += '</td>';
34         html += '</tr>';
35     } else {
36         html += '<tr id="F_' + items[index].path + '>';
37         html += '<td class="selected-item-indicator"></td>';
38         html += '<td class="repository-item">';
39         html += '<div class="item-icon icon-pdf"></div>';
40         html += '<div class="item-info">';
41         html += '<div class="item-top">' + items[index].name + '</div>';
42         html += '<div class="item-bottom">';
43         html += '<div class="item-status">' + items[index].modified + '</div>';
44         html += '</div>';
45         html += '</div>';
46         html += '</td>';
47         html += '</tr>';
48     }
49 }
50 html += '</table>';
51 return html;
52 };
```

The function `LNAPP.gui.actions.generateDirectoryListing` takes the JSON formatted directory listing data and transforms it in to a valid HTML table element that is then inserted into the HTML page in place of the previous directory listing.

2.2 Business Logic Layer

The web application's functionality and business logic is provided to client through its JSON-RPC interface. The interface provides web services that offer both technical functions including, for instance, authentication and authorization, as well as business logic oriented functionality including repository-, document- and accounting data management.

2.2 Business Logic Layer

JSON-RPC is a simple stateless remote procedure call protocol that uses the JSON format for the data exchange between the client and the JSON-RPC server. The web application developed within the scope of this thesis features a JSON-RPC server that is based on the JSON-RPC 2.0 Specification.

The JSON-RPC request object contains following members (JSON-RPC Working Group, 2013):

- **jsonrpc** - JSON-RPC protocol version
- **id** - unique identifier used by the client to establish the correlation between the sent request and received response
- **method** - name of the remote method that is to be invoked
- **params** - remote method parameter values that are to be used for the remote method invocation

After the remote method has been invoked, the server responds with a response object that can contain following members:

- **jsonrpc** - JSON-RPC protocol version
- **id** - unique identifier used by the client in the request
- **result** - this member contains the result of the method invocation, and is only present if the remote method invocation was successful
- **error** - this member contains the information about the error that occurred during the invocation, and is present only if the remote method invocation was not successful

The web application's JSON-RPC interface contains web services that can be divided into three groups: user management services, repository management services, and the utility services.

2.2.1 User Management

The user management web services provide the functionality that is needed to create an application user, and manage the user's authentication data (table 2.1).

2 Architecture and Functionality

Web Service	Parameters	Result	Description
AUTH::SIGN_IN	username password	code message	Sign in
AUTH::SIGN_OUT	-	code message	Sign out
AUTH::CREATE_ACCOUNT	email first_name last_name password	code message	Create a user account
AUTH::RESET_PASSWORD	email	code message	Reset the user's password
AUTH::CHANGE_PASSWORD	password token	code message	Change the user's password

Table 2.1: User management services

The service AUTH::SIGN_IN is used to sign in an already registered user, while the AUTH::SIGN_OUT service is used to sign the user out of the application and close the session.

The AUTH::CREATE_ACCOUNT service provides the possibility to create the user's account. It takes the user's valid email address, their first name, their last name, and a password, and returns a security token needed for the account activation and a status message.

When the users forget their password, or merely want to change it, they can do so by using the web services AUTH::RESET_PASSWORD and AUTH::CHANGE_PASSWORD. The service AUTH::RESET_PASSWORD needs to be called first, because it generates a security token needed to perform the actual password change. The token is sent to the user's email address and can then be used alongside the new password as an input parameter for the AUTH::CHANGE_PASSWORD service.

The usage of the services and their integration in the web application is discussed in more detail in the chapter [3.2](#).

2.2.2 Repository Management

The repository management web services can be divided into three groups: cloud repository management services, file management services and the accounting data management services.

2.2 Business Logic Layer

The cloud repository management services are used to create a cloud repository and manage its settings and properties.

Web Service	Parameters	Result	Description
REPOSITORY::AUTHORIZE	cloud_service_id repository_name	cloud_service_auth_url code message	Starts the OAuth2 process
REPOSITORY::UPDATE	data_layout_id	code message	Update repository properties
REPOSITORY::CONNECT	authorization_id	code message	Establish a connection to a cloud service
REPOSITORY::DISCONNECT	-	code message	Close the connection to the currently connected cloud service
REPOSITORY::AUTHORIZE_USER	username role	repository_users code message	Authorize the application user for the currently connected repository
REPOSITORY::REVOKE_USER	user_id	repository_users code message	Revoke the user's authorization for the currently connected repository
REPOSITORY::SET_PRIVILEGES	privileges	code message	Set the repository user privileges
REPOSITORY::CREATE_DATA_FIELD	data_field_label data_field_type	data_fields code message	Create an accounting data field
REPOSITORY::CREATE_DATA_LAYOUT	data_layout_name data_layout	code message	Create an accounting data layout
REPOSITORY::CREATE_PARAMETER_TABLE	parameter_table	code message	Create a parameter table
REPOSITORY::POST_COMMENT	document_id comment	comments code message	Post a comment regarding a certain document
REPOSITORY::GET_COMMENTS	document_id	comments code message	Get all comments regarding a certain document

Table 2.2: Cloud repository management services

The first step towards creating a document repository is authorizing the web application for use with the chosen cloud storage service. This is done by using the REPOSITORY::AUTHORIZE service. This service generates and returns the authorization URI, where the user can sign in to the chosen cloud storage service using their credentials and grant the permissions needed to access and use the storage. The authorization process uses the OAuth 2.0 flow, being that all three cloud storage services considered in this thesis support this authorization protocol.

The OAuth 2.0 is an industry-standard authorization protocol that provides authorization flows for multiple different devices and applications (Aaron Parecki, 2019). In order for web application to obtain the token needed for service access, it uses the OAuth 2.0 authorization code grant method. With this method, the client first obtains the authorization code that is afterwards

2 Architecture and Functionality

exchanged for the access token. The figure 2.1 shows the OAuth 2.0 process flow when using the authorization code grant.

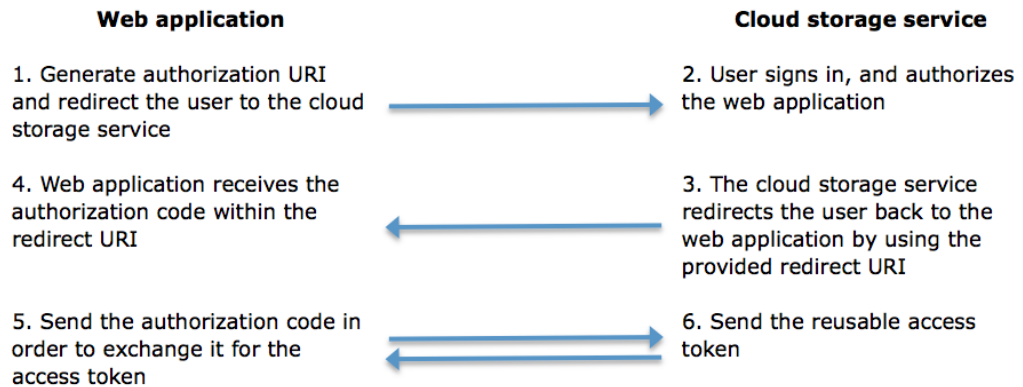


Figure 2.1: OAuth 2.0 authorization process flow

As shown in the figure 2.1, the web application first needs to generate the authorization URI and redirect the user to it. In the second step the user signs in to the cloud storage service and grants the permissions. This way the user credentials are not exposed to the web application, which is an important security feature. After the permissions have been granted, the cloud storage service redirects the user back to the web application by using the redirect URI, which has been previously communicated. In the final stage, the web application needs to extract the authorization code from the redirect URI and use it to perform a request for an access token. This request is performed in the background, and without the interaction from the user. Once the response from the cloud storage service comes in, the reusable access token is extracted and internally saved for future use.

Once the authorization has been successfully conducted, the document repository will be ready for use, and the user that has created the repository will automatically assume the role of the repository owner. This will allow them to add or remove further users to or from the repository by using the services `REPOSITORY::AUTHORIZE_USER` and `REPOSITORY::REVOKE_USER`. When authorizing a new repository user, the parameter `role` is used to assign the user an appropriate role. This will affect

2.2 Business Logic Layer

the privileges that the new user will have within the repository context. The newly created repository will have the default set of privileges assigned to it, but if the repository owner wants to define a custom set of privileges they can do so by using the `REPOSITORY::SET_PRIVILEGES` service. Roles and privileges are discussed in more detail in the chapter 3.2.

In order to be able to work with the repository, the user needs to connect to it first. This is done by using the `REPOSITORY::CONNECT` service. This service internally sets up the web application to be able to work with the repository. This includes reading the cloud storage service parameters and saving them into session variables, reading and initializing the appropriate repository permissions etc. When the user is done working with the repository, or wants to switch to a different repository, the currently active repository must be disconnected by using the `REPOSITORY::DISCONNECT` service.

The service `REPOSITORY::CREATE_DATA_FIELD` provides the possibility for user to define their own accounting data fields, while the service `REPOSITORY::CREATE_DATA_LAYOUT` enables the user to create a custom accounting data field layout that suits their workflow best. Within a layout, the user can define the order in which the accounting data fields appear in the user interface, and which fields are visible and which are hidden. Each repository is created with only two default accounting data fields, Document ID and Text, but these two services allow the web application to be expanded and modified so that it can accommodate further accounting data that some users may need. Additionally the service `REPOSITORY::UPDATE` can be used to set the default layout for the repository.

In order to simplify the workflow, the web application provides the possibility to save predefined parameter tables in form of key-value pairs that are used for the accounting data input in the user interface. These key-value pairs are displayed in the user interface as HTML select elements. The parameter tables must be defined in JSON format and can then be saved by using the `REPOSITORY::CREATE_PARAMETER_TABLE` service.

The web application's simple commenting system features two web services: `REPOSITORY::POST_COMMENT` and `REPOSITORY::GET_COMMENTS`. Former is used to post a comment regarding a certain document, while the latter can be used to fetch all comments regarding a document.

2 Architecture and Functionality

Web Service	Parameters	Result	Description
REPOSITORY::CREATE_DIRECTORY	path directory	code message	Create a directory in the cloud storage
REPOSITORY::LIST_DIRECTORY	path	directory_listing breadcrumbs path code message	List the contents of a cloud storage directory
REPOSITORY::UPLOAD	path upload_file	code message	Upload a file to the cloud storage
REPOSITORY::DOWNLOAD	path file_name	document_id download_url code message	Download a file from the cloud storage
REPOSITORY::DELETE_FILE	path	code message	Delete a file or directory from the cloud storage

Table 2.3: File management services

The file management services listed in the table 2.3 are used to interact with the cloud storage service. The web service `REPOSITORY::LIST_DIRECTORY` allows the user to navigate through the directory hierarchy of the cloud storage. This service returns the contents of a chosen directory, but it also provides the breadcrumbs that indicate where in the hierarchy is the current directory located. With the service `REPOSITORY::CREATE_DIRECTORY`, the user can create a directory in a current directory hierarchy location.

Files can be uploaded to the cloud storage by using the web service `REPOSITORY::UPLOAD`. Within the web application's context, the files are always uploaded into the directory that was previously set by using the `REPOSITORY::LIST_DIRECTORY` service. In order to display the document to the user within the web application's user interface, the file needs to be downloaded first. This is achieved by calling the `REPOSITORY::DOWNLOAD` service, and using the returned download URL to download the document.

Additionally, files and directories can be deleted from the cloud storage by using the `REPOSITORY::DELETE_FILE` web service.

2.2 Business Logic Layer

Web Service	Parameters	Result	Description
REPOSITORY::SET_ACCOUNTING_DATA	document_id [accounting data fields]	code message	Set the accounting data related to a certain document
REPOSITORY::GET_ACCOUNTING_DATA	document_id	accounting_data_id [accounting data fields] code message	Get the accounting data related to a certain document
REPOSITORY::GET_ACCOUNTING_DATA_REVISIONS	accounting_data_id	accounting_data_revisions code message	Get the accounting data revisions
REPOSITORY::CREATE_ACCOUNTING_RECORD	document_id record_date accounting_records	code message	Create accounting records related to a certain document
REPOSITORY::GET_ACCOUNTING_RECORDS	document_id	accounting_records code message	Get accounting records related to a certain document

Table 2.4: Accounting data management services

The accounting data management services provide the functionality needed to create and edit the accounting data and accounting records (table 2.4). As mentioned earlier, there are only two default accounting data fields, Document ID and Text, but with the use of the provided web services, additional accounting data fields can be created. The web service REPOSITORY::SET_ACCOUNTING_DATA is used for saving the accounting data related to a certain document. This service takes the document id, and the accounting data as an input. The accounting data parameter can also include the custom accounting data fields that were additionally created with the REPOSITORY::CREATE_DATA_FIELD service. The accounting data can be fetched with the web service REPOSITORY::GET_ACCOUNTING_DATA. This service returns both the accounting data as well as the accounting data id that is needed in order to fetch the accounting data revisions by using the REPOSITORY::GET_ACCOUNTING_DATA_REVISIONS web service.

The accounting records, on the other hand, can be created by using the REPOSITORY::CREATE_ACCOUNTING_RECORD web service. The input to this service must contain the document id, the record date, and one or more accounting records. To fetch the accounting records related to a certain document, the web service REPOSITORY::GET_ACCOUNTING_RECORDS can be used. This web service takes only a document id as an input, and returns all accounting records related to this document.

2 Architecture and Functionality

2.2.3 Utility Services

The utility services are web services created to add further functionality to the web application. They are partially intended to work only with the web application that is developed within the scope of this thesis, for some of them generate and return HTML code that is then dynamically inserted into the HTML page.

Web Service	Parameters	Result	Description
APP::CHANGE.SETTINGS	settings	code message	Change the application settings
APP::LOG.ERROR	message	reference code message	Log a client-side error
APP::UI.USER.LOG	user.id	html code message	Get the HTML formatted user log
APP::UI.USER.REQUESTS	user.id	html code message	Get the HTML formatted user requests log
APP::UI.APPLICATION.LOG	log.file	html code message	Get the HTML formatted application log
APP::UI.DIALOG	dialog.id step	dialog.id step html code message	Get the HTML formatted dialog
APP::UI.REPOSITORY.DASHBOARD	repository.id dialog.id	html dialog code message	Get the HTML formatted repository dashboard

Table 2.5: Application utility services

The web services APP::LOG_ERROR and APP::UI_APPLICATION_LOG are important for the application maintenance. The former is used to log an error that occurred on the client-side, while the latter can be used to list all the errors that have been logged, both client- and server-side.

The utility services APP::UI_USER_LOG and APP::UI_USER_REQUESTS are created for application's user management. The service APP::UI_USER_LOG returns the list of the web application sign in attempts made by the user. The list contains the information regarding the time of the attempts and if they were successful or not. The web service APP::UI_USER_REQUESTS returns the list of all of the user requests that were made. More information regarding the user requests and how they are used within the web application is available in the chapter 3.2.

2.3 Data Layer

The services `APP::UI_DIALOG` and `APP::UI_REPOSITORY_DASHBOARD` are both used to generate different user interface dialog elements. The web service `APP::UI_DIALOG` is used to create simple dialogs such as the one needed for file upload. It generates the dialog formatted in HTML and sends it to the client where the dialog is displayed. The web service `APP::UI_REPOSITORY_DASHBOARD` generates the more complex repository dashboard element for the user interface. The repository dashboard contains all the functions needed for repository management such as connecting to the repository or authorizing a new user. The repository management and usage of the repository dashboard are discussed in more detail in the chapter 3.3.

The web service `APP::CHANGE_SETTINGS` is used to set the web application's user specific settings.

2.3 Data Layer

Due to the fact that an accountant or a bookkeeping service can have more than one client, the web application needs to feature an architecture that provides data storage for multiple tenants. This means that the data model must provide means to ensure that the data belonging to multiple different clients is stored properly in regard to efficiency and security, so one of the biggest challenges in implementation of such a model will be the data isolation.

There are generally three approaches to implementing a data architecture for an SaaS delivered applications. The data can be managed by using separate databases for each tenant, by using a single database with separate schemas for each tenant, and lastly by using a single database and a schema that is shared by all tenants (Chong, Carraro, and Wolter, 2017). The application implemented within the scope of this thesis will use the second approach which houses all of the tenants in a single database with separate schemas, but due to the fact that MySQL physically does not differentiate between a schema and a database (Oracle Corporation and/or its affiliates, 2017c), the data architecture will provide each tenant with a set of separate tables that can be uniquely identified by the repository ID to store the data that should

2 Architecture and Functionality

be isolated (e.g. accounting data), while using shared tables for data that needs to be available outside the tenant's context, such as the user data. The data structures and their relationships are described and further discussed in the chapter [3.1.1](#).

3 Implementation

3.1 Database

The application will store the data by using MySQL, which is a dual-licensed relational database management system. The software can be used as open-source under the terms of GNU General Public License, or by purchasing a commercial license (Oracle Corporation and/or its affiliates, 2017a).

3.1.1 Data Model

The tables that store the client's isolated data, such as document- or accounting data, will be created using the stored procedures as soon as the client creates a repository by authenticating the application with the chosen cloud service. Additionally, the privileges for these tables will be assigned only to a new dedicated database user. Shared tables, which house the session data, user data, repository data, authorization data, user log data, user requests, user settings and the repository privileges will be created during the installation process.

Here are the listings for the database tables, procedures and functions alongside with the description of the data structures and relationships:

Table Sessions

The sessions table (Listing 3.1) stores the user session information which are used across multiple subsequent requests within the web application. The column `session.data` will hold the encrypted session data while the key that is used to encrypt it will be stored in the column `session.key`.

3 Implementation

Information about the exact time when a session was last accessed will be stored in the `last_accessed` column.

Listing 3.1: Listing for the database table sessions

```
1 CREATE TABLE 'sessions' (  
2   'session_id' VARCHAR(128) NOT NULL,  
3   'session_key' VARCHAR(128) DEFAULT NULL,  
4   'session_data' TEXT DEFAULT NULL,  
5   'last_accessed' INT UNSIGNED DEFAULT NULL,  
6   PRIMARY KEY ('session_id')  
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Table Users

The table `users` (Listing 3.2) stores all of the user related data. Each user is internally identified by a unique `user_id` which is automatically assigned by using the `AUTO_INCREMENT` attribute. The column `user_status` will be used to store the current status of the user's account. A status can be, for instance, used to lock the user and temporarily deny the access to the application as a security measure if a wrong password is entered multiple times in a row.

The column `username`, which is indexed, will store a user's valid email address which needs to be provided during the user account creation. This measure is, apart from its security purpose, also used to ensure that the username is unique within the system. The encrypted user password will be stored in the `password` column.

The columns `first_name` and `last_name` will store the user's real name. By storing the user's real name it will be easier, for instance, to track changes and identify the accounting clerk who entered the accounting data. The column `registered` will hold the date and time of the user's registration.

Listing 3.2: Listing for the database table users

```
1 CREATE TABLE 'users' (  
2   'user_id' INT UNSIGNED NOT NULL AUTO_INCREMENT,  
3   'user_status' TINYINT UNSIGNED NOT NULL DEFAULT 0,  
4   'username' VARCHAR(254) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,  
5   'password' VARCHAR(128) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,  
6   'first_name' VARCHAR(100) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,  
7   'last_name' VARCHAR(100) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,  
8   'registered' DATETIME NOT NULL,  
9   PRIMARY KEY ('user_id'),  
10  UNIQUE KEY 'idx_username' ('username')  
11 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
```

Table Repositories

All of the information related to the document repositories will be stored in the repositories table (Listing 3.3). Each repository will have a status, which can be used to mark the repository as deleted when needed, while the entry remains in the table for revision purposes.

The columns `repository_name`, `service_id` and `service_token` are all related to a cloud service which is storing the documents, so they hold the repository name which is provided by the user to identify a certain repository, a service id that is internally used to identify a cloud service (i.e. Dropbox, Google Drive or OneDrive), and the token that is received after completing the authorization process.

The columns `database_user` and `database_password` store the dedicated database user information that is internally needed to access the set of isolated tables that are created for each of the client's repositories as discussed in the beginning of this chapter.

Additionally, the default accounting data layout will be stored in the column `layout_id`.

Listing 3.3: Listing for the database table repositories

```

1 CREATE TABLE 'repositories' (
2   'repository_id' INT UNSIGNED NOT NULL AUTO_INCREMENT,
3   'repository_status' TINYINT UNSIGNED NOT NULL DEFAULT 1,
4   'repository_name' VARCHAR(50) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,
5   'layout_id' INT UNSIGNED NOT NULL DEFAULT 1,
6   'service_id' TINYINT UNSIGNED NOT NULL,
7   'service_token' VARCHAR(2048) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,
8   'database_user' VARCHAR(16) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,
9   'database_password' VARCHAR(128) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,
10  PRIMARY KEY ('repository_id')
11 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;

```

Table Authorizations

The authorizations table (Listing 3.4) stores all of the user's authorizations that are needed to access the repositories, and therefore connects the users table with the repositories table through the two foreign keys - the `user_id` and the `repository_id`. Additionally there is a deletion restriction enforced upon the two parent tables `users` and `repositories` which prevents a row from being deleted in one of these tables as long as there is an

3 Implementation

authorization present for the user or the repository in question. The column `user_role` stores the information regarding the role that a user has within the authorized repository (e.g. client or accountant).

Listing 3.4: Listing for the database table authorizations

```
1 CREATE TABLE 'authorizations' (  
2   'authorization_id' INT UNSIGNED NOT NULL AUTO_INCREMENT,  
3   'repository_id' INT UNSIGNED NOT NULL,  
4   'user_id' INT UNSIGNED NOT NULL,  
5   'user_role' TINYINT UNSIGNED NOT NULL,  
6   PRIMARY KEY ('authorization_id'),  
7   FOREIGN KEY ('user_id') REFERENCES 'users' ('user_id') ON DELETE RESTRICT,  
8   FOREIGN KEY ('repository_id') REFERENCES 'repositories' ('repository_id') ON DELETE RESTRICT,  
9   UNIQUE KEY 'idx_user_repository' ('user_id','repository_id')  
10  ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
```

Table User Log

The `user_log` table (Listing 3.5) allows the system to track the user's login attempts for security purposes. The table stores the exact time of the login attempt in the `attempt_time` column, an information indicating if the login was successful in the `success` column, and the user's id in the `user_id` column. The foreign key `user_id` also enforces the deletion restriction upon the `users` table as long as there are user log entries for a certain user present in the database.

Listing 3.5: Listing for the database table user_log

```
1 CREATE TABLE 'user_log' (  
2   'user_log_id' BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,  
3   'user_id' INT UNSIGNED NOT NULL,  
4   'attempt_time' BIGINT UNSIGNED NOT NULL,  
5   'success' TINYINT UNSIGNED NOT NULL,  
6   PRIMARY KEY ('user_log_id'),  
7   FOREIGN KEY ('user_id') REFERENCES 'users' ('user_id') ON DELETE RESTRICT  
8   ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
```

Table User Requests

The `user_requests` table (Listing 3.6) is used to keep track of the different user requests such as an account activation, or a password reset request. Each request is identified by a unique `user_request_id`, while the column `request_type` holds the information about the type of the request that was posed by the user. Columns `request_time` and `activity_time` indicate the

exact time of the request (e.g. when the user requested a password reset) and the time the requested action was actually performed by the user (e.g. the password has actually been reset by the user). The foreign key `user_id` references the the user that has posed the request.

The purpose and usage of the `user_requests` table will be discussed in more detail in the chapters [3.2.3](#) and [3.2.4](#).

Listing 3.6: Listing for the database table `user_requests`

```
1 CREATE TABLE `user_requests` (  
2   `user_request_id` BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,  
3   `user_id` INT UNSIGNED NOT NULL,  
4   `request_type` TINYINT UNSIGNED NOT NULL,  
5   `request_time` BIGINT UNSIGNED NOT NULL,  
6   `activity_time` BIGINT UNSIGNED DEFAULT NULL,  
7   PRIMARY KEY (`user_request_id`),  
8   FOREIGN KEY (`user_id`) REFERENCES `users` (`user_id`) ON DELETE RESTRICT  
9 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
```

Table Privileges

The `privileges` table (Listing [3.7](#)) stores the custom repository privileges. Each record is identified by a unique `privilege_id`, while the columns `repository_id` and `role` refer to which repository and which role the privileges apply to. The column `privileges` stores the actual privilege data.

The purpose and usage of the `privileges` table will be discussed in more detail in the chapter [3.2.2](#).

Listing 3.7: Listing for the database table `privileges`

```
1 CREATE TABLE `privileges` (  
2   `privilege_id` INT UNSIGNED NOT NULL AUTO_INCREMENT,  
3   `repository_id` INT UNSIGNED NOT NULL,  
4   `role` TINYINT UNSIGNED NOT NULL,  
5   `privileges` INT UNSIGNED NOT NULL,  
6   PRIMARY KEY (`privilege_id`),  
7   FOREIGN KEY (`repository_id`) REFERENCES `repositories` (`repository_id`) ON DELETE RESTRICT,  
8   UNIQUE KEY `idx_repository_role` (`repository_id`,`role`)  
9 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
```

Table Settings

The `table_settings` stores the application's user specific settings. The settings are stored in the `settings` column as a key - value pairs in JSON format.

3 Implementation

Being that they are user-related, each settings record will have a reference to a user (column `user_id`). The settings will be discussed in more detail in the chapter 3.5.

Listing 3.8: Listing for the database table settings

```
1 CREATE TABLE 'settings' (  
2   'settings_id' INT UNSIGNED NOT NULL AUTO_INCREMENT,  
3   'user_id' INT UNSIGNED NOT NULL,  
4   'settings' TEXT NOT NULL,  
5   PRIMARY KEY ('settings_id'),  
6   FOREIGN KEY ('user_id') REFERENCES 'users' ('user_id') ON DELETE RESTRICT,  
7   UNIQUE KEY 'idx_user' ('user_id')  
8 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
```

Stored Function `sf_table_suffix`

As already mentioned earlier, the client specific set of tables is created by using the stored procedures. In order to be able to identify a set of tables that belong to a certain client, the system will use the repository id as a table name suffix. This suffix will be generated with a stored function `sf_table_suffix` (Listing 3.9) by left-padding the repository id with zeros. A client specific table will therefore have a name in the format `<TABLE_NAME>_<SUFFIX>` (e.g. `documents_0000000001`).

Listing 3.9: Listing for the stored function `sf_table_suffix`

```
1 CREATE FUNCTION sf_table_suffix (repository_id INT)  
2 RETURNS CHAR(10) DETERMINISTIC  
3 RETURN LPAD (repository_id, 10, '0');
```

Stored Procedure `sp_create_table_documents`

The stored procedure `sp_create_table_documents` (Listing 3.10) is used to create the `documents` table. The `documents` table keeps record of every document that is uploaded to the cloud storage repository, and is the first of overall seven tables that are created for each repository. Each record in the table has a unique `document_id` as well as a reference to a repository that it belongs to in the column `repository_id`. The information concerning the date and time of the upload, as well as a reference to the user that has uploaded the document are kept in the columns `created_date`,

created_time and created_user_id, respectively. The columns file_name, file_last_modification and md5_file are used to identify the uploaded file. The columns file_name and file_last_modification contain the name of the uploaded file including the file extensions and the time of the last modification, while the md5_file column stores the MD5 hash or a checksum of the file that can be used both to check if a duplicate of a document is being uploaded or to identify a document if the file is renamed. The column file_size contains the size of the uploaded file in bytes.

Listing 3.10: Listing for the stored procedure sp_create_table_documents

```

1 DELIMITER $
2
3 CREATE PROCEDURE sp_create_table_documents (IN repository_id INT)
4 BEGIN
5
6 SET @statement = CONCAT (
7 'CREATE TABLE documents_', sf_table_suffix (repository_id), '(
8   'document_id' INT UNSIGNED NOT NULL AUTO_INCREMENT,
9   'repository_id' INT UNSIGNED NOT NULL,
10  'created_user_id' INT UNSIGNED NOT NULL,
11  'created_date' DATE NOT NULL,
12  'created_time' TIME NOT NULL,
13  'file_name' VARCHAR(100) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,
14  'file_size' INT UNSIGNED NOT NULL,
15  'file_last_modification' INT UNSIGNED NOT NULL,
16  'md5_file' VARCHAR(32) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,
17  PRIMARY KEY ('document_id'),
18  FOREIGN KEY ('created_user_id') REFERENCES 'users' ('user_id') ON DELETE RESTRICT,
19  FOREIGN KEY ('repository_id') REFERENCES 'repositories' ('repository_id') ON DELETE RESTRICT,
20  UNIQUE KEY 'idx_file_repository' ('file_name','repository_id')
21 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
22 );
23
24 PREPARE statement FROM @statement;
25 EXECUTE statement;
26 DEALLOCATE PREPARE statement;
27
28 END $
29
30 DELIMITER ;

```

Stored Procedure sp_create_table_accounting_data_fields

The procedure sp_create_table_accounting_data_fields (Listing 3.11) is used to create an accounting_data_fields table for each repository. This table allows the client to define a custom set of accounting data fields on a per repository basis. Each field defined by the client has a unique id which is used as a column name suffix in the accounting_data table. This means that each time a customer creates a new field, a new record will be created in the accounting_data_fields table and a new column added to the accounting_data table. Hence, the table accounting_data_fields holds

3 Implementation

the metadata for the columns in the table `accounting_data`. This metadata includes the label for the field that is visible in the user interface (column `field_label`), and an optional reference to a parameter table (column `parameter_table_id`). The table `accounting_data` will be introduced later in this chapter.

Listing 3.11: Listing for the stored procedure `sp_create_table_accounting_data_fields`

```
1 DELIMITER $
2
3 CREATE PROCEDURE sp_create_table_accounting_data_fields (IN repository_id INT)
4 BEGIN
5
6 SET @statement = CONCAT (
7 'CREATE TABLE accounting_data_fields_', sf_table_suffix (repository_id), '(
8   'field_id' INT UNSIGNED NOT NULL AUTO_INCREMENT,
9   'repository_id' INT UNSIGNED NOT NULL,
10  'field_label' VARCHAR(100) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,
11  'parameter_table_id' VARCHAR(100) CHARACTER SET utf8 COLLATE utf8_unicode_ci DEFAULT NULL,
12  PRIMARY KEY ('field_id'),
13  FOREIGN KEY ('repository_id') REFERENCES 'repositories' ('repository_id') ON DELETE RESTRICT
14 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;'
15 );
16
17 PREPARE statement FROM @statement;
18 EXECUTE statement;
19 DEALLOCATE PREPARE statement;
20
21 END $
22
23 DELIMITER ;
```

Stored Procedure `sp_create_table_accounting_data_layouts`

The procedure `sp_create_table_accounting_data_layouts` (Listing 3.12) creates the `accounting_data_layouts` table, which allows the creation of multiple accounting data field layouts. Each layout has a unique id (column `layout_id`), and holds a reference to the related repository (column `repository_id`). The column layout represents the actual layout by storing the list of the field id's that will be visible in the user interface, while the column `layout_name` stores a descriptive name that the user provided for the created layout.

Listing 3.12: Listing for the stored procedure `sp_create_table_accounting_data_layouts`

```
1 DELIMITER $
2
3 CREATE PROCEDURE sp_create_table_accounting_data_layouts (IN repository_id INT)
4 BEGIN
5
6 SET @statement = CONCAT (
```

3.1 Database

```
7 'CREATE TABLE accounting_data_layouts_', sf_table_suffix (repository_id), '(
8 'layout_id' INT UNSIGNED NOT NULL AUTO_INCREMENT,
9 'repository_id' INT UNSIGNED NOT NULL,
10 'layout_name' VARCHAR(100) CHARACTER SET utf8 COLLATE utf8_unicode_ci NOT NULL,
11 'layout' TEXT NOT NULL,
12 PRIMARY KEY ('layout_id'),
13 FOREIGN KEY ('repository_id') REFERENCES 'repositories' ('repository_id') ON DELETE RESTRICT
14 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;'
15 );
16
17 PREPARE statement FROM @statement;
18 EXECUTE statement;
19 DEALLOCATE PREPARE statement;
20
21 END $
22
23 DELIMITER ;
```

Stored Procedure `sp_create_table_accounting_data`

The procedure `sp_create_table_accounting_data` (Listing 3.13) is used to create one of the key tables in the application - the `accounting_data` table. This table holds all of the client's accounting data related to the commercial documents that are uploaded to the cloud storage.

Initially there are only three columns in the table. The primary key column `accounting_data_id` holds a unique id for every record. The columns `document_id` and `repository_id` reference the commercial document to which the accounting data relates to, and the containing repository, respectively.

Columns that hold the actual accounting data are created additionally from the user interface. This approach allows the accountant to create a custom data structure for every client individually, if needed. As already mentioned earlier, for each created data field, a new column will be added to the `accounting_data` table, while the metadata concerning the newly created column is stored in the `accounting_data_fields` table.

Listing 3.13: Listing for the stored procedure `sp_create_table_accounting_data`

```
1 DELIMITER $
2
3 CREATE PROCEDURE sp_create_table_accounting_data (IN repository_id INT)
4 BEGIN
5
6 SET @statement = CONCAT (
7 'CREATE TABLE accounting_data_', sf_table_suffix (repository_id), '(
8 'accounting_data_id' INT UNSIGNED NOT NULL AUTO_INCREMENT,
9 'document_id' INT UNSIGNED NOT NULL,
10 'repository_id' INT UNSIGNED NOT NULL,
11 PRIMARY KEY ('accounting_data_id'),
```

3 Implementation

```
12 FOREIGN KEY ('document_id') REFERENCES 'documents_', sf_table_suffix (repository_id), '' ('
    document_id') ON DELETE RESTRICT,
13 FOREIGN KEY ('repository_id') REFERENCES 'repositories' ('repository_id') ON DELETE RESTRICT,
14 UNIQUE KEY 'idx_document_repository' ('document_id','repository_id')
15 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
16 );
17
18 PREPARE statement FROM @statement;
19 EXECUTE statement;
20 DEALLOCATE PREPARE statement;
21
22 END $
23
24 DELIMITER ;
```

Stored Procedure `sp_create_table_accounting_data_revisions`

The stored procedure `sp_create_table_accounting_data_revisions` (Listing 3.14) is used to create the table `accounting_data_revisions`. This table keeps track of every change that is made to the accounting data. Each record in the table represents a version of a record from the `accounting_data` table, so it has a unique id (column `accounting_data_revision_id`) as well as a reference to the accounting data record (column `accounting_data_id`) and a reference to a repository (column `repository_id`). The information concerning the date and time of the change made to a record, as well as a reference to the user that has made the change is kept in the columns `created_date`, `created_time` and `created_user_id`, respectively. In order to keep track of the changes made to the accounting data, the columns that are created in the table `accounting_data` will also be created in the table `accounting_data_revisions`.

Listing 3.14: Listing for the stored procedure `sp_create_table_accounting_data_revisions`

```
1 DELIMITER $
2
3 CREATE PROCEDURE sp_create_table_accounting_data_revisions (IN repository_id INT)
4 BEGIN
5
6 SET @statement = CONCAT (
7 'CREATE TABLE accounting_data_revisions_', sf_table_suffix (repository_id), '(
8 'accounting_data_revision_id' INT UNSIGNED NOT NULL AUTO_INCREMENT,
9 'accounting_data_id' INT UNSIGNED NOT NULL,
10 'repository_id' INT UNSIGNED NOT NULL,
11 'created_user_id' INT UNSIGNED NOT NULL,
12 'created_date' DATE NOT NULL,
13 'created_time' TIME NOT NULL,
14 PRIMARY KEY ('accounting_data_revision_id'),
15 FOREIGN KEY ('accounting_data_id') REFERENCES 'accounting_data_', sf_table_suffix (
    repository_id), '' ('accounting_data_id') ON DELETE RESTRICT,
16 FOREIGN KEY ('repository_id') REFERENCES 'repositories' ('repository_id') ON DELETE RESTRICT,
17 FOREIGN KEY ('created_user_id') REFERENCES 'users' ('user_id') ON DELETE RESTRICT
18 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
19 );
20
```

```

21 PREPARE statement FROM @statement;
22 EXECUTE statement;
23 DEALLOCATE PREPARE statement;
24
25 END $
26
27 DELIMITER ;

```

Stored Procedure `sp_create_table_accounting_records`

The stored procedure `sp_create_table_accounting_records` (Listing 3.15) is used for the creation of the `accounting_records` table.

The table `accounting_records` stores all of the accounting records related to a certain document (column `document_id`). Each accounting record must have a record date (column `date`), indication whether it is a credit or a debit record (column `credit_debit`), amount (column `amount`), and the affected account (column `account`).

Listing 3.15: Listing for the stored procedure `sp_create_table_accounting_records`

```

1 DELIMITER $
2
3 CREATE PROCEDURE sp_create_table_accounting_records (IN repository_id INT)
4 BEGIN
5
6 SET @statement = CONCAT (
7 'CREATE TABLE accounting_records_', sf_table_suffix (repository_id), '(
8   'accounting_record_id' INT UNSIGNED NOT NULL AUTO_INCREMENT,
9   'document_id' INT UNSIGNED NOT NULL,
10  'repository_id' INT UNSIGNED NOT NULL,
11  'date' DATE NOT NULL,
12  'credit_debit' TINYINT NOT NULL,
13  'account' INT UNSIGNED NOT NULL,
14  'amount' DOUBLE NOT NULL,
15  PRIMARY KEY ('accounting_record_id'),
16  FOREIGN KEY ('document_id') REFERENCES 'documents_', sf_table_suffix (repository_id), '(('
17  document_id') ON DELETE RESTRICT,
18  FOREIGN KEY ('repository_id') REFERENCES 'repositories' ('repository_id') ON DELETE RESTRICT
19 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;'
20 );
21 PREPARE statement FROM @statement;
22 EXECUTE statement;
23 DEALLOCATE PREPARE statement;
24
25 END $
26
27 DELIMITER ;

```

Stored Procedure `sp_create_table_comments`

The stored procedure `sp_create_table_comments` (Listing 3.16) creates the `comments` table. This table stores the comments that are written by the users

3 Implementation

on a document level. All the comments that are related to a certain document will be displayed as a one single thread, so they will have a reference to a document that they relate to (column `document_id`), and will be sorted by data and time (columns `date` and `time`) representing the order in which the comments have been written. The content of the comments will be stored in the `comment_text` column.

Listing 3.16: Listing for the stored procedure `sp_create_table_comments`

```
1 DELIMITER $
2
3 CREATE PROCEDURE sp_create_table_comments (IN repository_id INT)
4 BEGIN
5
6 SET @statement = CONCAT (
7 'CREATE TABLE comments_', sf_table_suffix (repository_id), '(
8   'comment_id' INT UNSIGNED NOT NULL AUTO_INCREMENT,
9   'document_id' INT UNSIGNED NOT NULL,
10  'repository_id' INT UNSIGNED NOT NULL,
11  'user_id' INT UNSIGNED NOT NULL,
12  'date' DATE NOT NULL,
13  'time' TIME NOT NULL,
14  'comment_text' TEXT NOT NULL,
15  PRIMARY KEY ('comment_id'),
16  FOREIGN KEY ('document_id') REFERENCES 'documents_', sf_table_suffix (repository_id), '' ('
17   document_id') ON DELETE RESTRICT,
18  FOREIGN KEY ('repository_id') REFERENCES 'repositories' ('repository_id') ON DELETE RESTRICT,
19  FOREIGN KEY ('user_id') REFERENCES 'users' ('user_id') ON DELETE RESTRICT
20 ) ENGINE=InnoDB DEFAULT CHARSET=utf8 AUTO_INCREMENT=1;
21 );
22 PREPARE statement FROM @statement;
23 EXECUTE statement;
24 DEALLOCATE PREPARE statement;
25
26 END $
27
28 DELIMITER ;
```

The remaining stored procedures that are not listed here, are the procedure `sp_create_user`, procedure `sp_drop_repository_tables` and procedure `sp_initialize_repository`. The first one is used to create a dedicated database user with appropriate privileges for the newly created repository tables, while the second one allows the system to drop the created repository tables if an error occurs in the process of creating a complete set of the repository tables. So if, for instance, an error occurs after creating the first two of the eight repository tables, the two created ones will be deleted in order to keep the database in a consistent state. The use of this procedure is necessary due to the fact that Data Definition Language (DDL) statements cause an implicit commit, and therefore cannot be simply rolled back (Oracle Corporation and/or its affiliates, 2017b). Additionally the procedure

`sp_initialize_repository` is used to initialize the repository tables with default values.

3.1.2 Database abstraction with PHP Data Objects

PHP Data Objects (PDO) is a PHP extension that provides an interface for database access (Achour, Betz, and Dovgal, 2017). By using the PDO, a same set of functions can be used to define and manipulate the data, no matter which database is used in the background. The available drivers that implement the PDO interface currently include support for databases such as MySQL, Oracle, PostgreSQL, SQLite and others.

In order to add an abstraction layer between the application logic and the database, the web application will use PHP classes that represent the data model and provide methods which allow data access by using the PDO. For instance, the PHP class `Users` will be used to provide access to the database table `users`. The PDO object which is needed to access the database is passed as a constructor parameter, and kept in a protected member variable (Listing 3.17). When the instance of the `Users` class is destructed, the connection is also automatically terminated.

Listing 3.17: Creation and destruction of the PDO object within the PHP class `Users`

```

1  protected $databaseHandle_;
2
3  public function __construct(PDO $databaseHandle) {
4      $this->databaseHandle_ = $databaseHandle;
5  }
6
7  public function __destruct() {
8      if ($this->databaseHandle_) {
9          $this->databaseHandle_ = null;
10     }
11 }
```

The PDO object itself is created in the PHP class `DatabaseAdapter` (Listing 3.18). This class provides only one function `getPDODatabaseHandle`. The `DatabaseAdapter` reads the connection parameters from the `Config` class and establishes a database connection. It also checks if a connection to a cloud service (i.e. repository) is active and currently in use. It does this by checking whether the user and password parameters for the isolated

3 Implementation

repository tables are available in the session data, and if they are, it uses them to make the connection instead. This check is necessary because there are basically two categories of the database users. The first category contains only one general database user that has permissions for the shared tables only, while the second category contains all the additionally created database users that have permissions both for the shared tables as well as for the isolated repository table set. As already discussed before, one such repository-related database user is added each time a repository is created within the application alongside the corresponding set of isolated tables. The general database user is therefore used only until the user connects to a repository, and once the repository connection is established, the repository-related database user's connection parameters become available in the session data and are used for all the subsequent connections.

Listing 3.18: PHP class DatabaseAdapter

```
1 class DatabaseAdapter {
2
3     const PDO_MYSQL_DRIVER_NAME = 'mysql';
4
5     public static function getPDODatabaseHandle($sharedTableAccess = false) {
6         try {
7             $driver = Config::get ( 'PDO_DRIVER' );
8             $username = Config::get ( strtoupper ( $driver ) . '_USERNAME' );
9             $password = Config::get ( strtoupper ( $driver ) . '_PASSWORD' );
10
11             if (! $sharedTableAccess) {
12                 $connectionParameters = Session::getRepositoryDatabaseParameter ();
13                 if (null !== $connectionParameters [0] && null !== $connectionParameters [1]) {
14                     $username = $connectionParameters [0];
15                     $password = $connectionParameters [1];
16                 }
17             }
18
19             if ($driver === self::PDO_MYSQL_DRIVER_NAME) {
20                 $pdo = new PDO ( $driver . ':host=' . Config::get ( strtoupper ( $driver ) . '_HOST' ) .
                ';dbname=' . Config::get ( strtoupper ( $driver ) . '_DATABASE' ), $username, $password );
21             };
22             return $pdo;
23         }
24         return null;
25     } catch ( PDOException $exception ) {
26         Log::logException ( $exception );
27         return null;
28     }
29 }
```

The public interface of the class Users, as well as public interfaces of all the other classes that represent the data model entities, provides methods that allow data definition and manipulation. All of these methods use PDO functionality to achieve this. As already discussed in this chapter, the PDO

3.2 User Management

object is created by the DatabaseAdapter and passed to the class instance during the construction. The function `createUser` from the class `Users` shows, for instance, how the `PDOStatement` object is used to prepare and execute a prepared statement that inserts a new user record into the table `users` (Listing 3.19).

Listing 3.19: Listing for the function `createUser` from the `Users` class

```
1 public function createUser($userStatus, $username, $password, $firstName, $lastName) {
2     try {
3         $statement = $this->databaseHandle->prepare ( 'INSERT INTO users (user_status, username,
4             password, first_name, last_name, registered) VALUES (:user_status, :username, :password, :
5             first_name, :last_name, NOW());' );
6         $statement->bindParam ( ':user_status', $userStatus, PDO::PARAM_INT );
7         $statement->bindParam ( ':username', $username, PDO::PARAM_STR );
8         $statement->bindParam ( ':password', $password, PDO::PARAM_STR );
9         $statement->bindParam ( ':first_name', $firstName, PDO::PARAM_STR );
10        $statement->bindParam ( ':last_name', $lastName, PDO::PARAM_STR );
11        if ($statement->execute () ) {
12            return $this->databaseHandle->lastInsertId ();
13        } else {
14            Log::logDatabaseError( $statement, __METHOD__ );
15        }
16        return null;
17    } catch ( Exception $exception ) {
18        Log::logException ( $exception );
19        return null;
20    }
21 }
```

3.2 User Management

In this chapter different aspects of user management will be discussed, ranging from fundamental concept of roles and privileges to a more complex process of a user account creation. There will be presented both theoretical considerations, as well as some of the actual code.

3.2.1 Roles

There are two groups of roles that are available within the application: the application level roles and the repository level roles.

The application level roles contain the application level privileges and define thereby what the user is allowed to do and see within the higher-level application context. These privileges include, for instance, the permission

3 Implementation

to see the application log or access the user management interface. The privileges for both application level roles and the repository level roles will be discussed in more detail in the chapter [3.2.2](#).

The technical implementation of the application level roles will be kept simple. There will be two available application level roles: the administrator and the user. The application will be able to identify the administrators based on their username. Configuration file will keep a list of all the usernames that have the administrator privileges, and the application will search this list when the user signs in to see if there is a match with the currently signed in user. The information regarding the application level role will be stored in the session ([Listing 3.20](#)), and therefore it will be available until the user signs off.

[Listing 3.20](#): Listing for the `setApplicationLevelRole` and `getApplicationLevelRole` functions from the `Session` class

```
1 public static function setApplicationLevelRole($userName) {
2     if (Config::isAdministrator ( $userName )) {
3         $_SESSION ['APPLICATION_LEVEL_ROLE'] = Config::APPLICATION_LEVEL_ROLE_ADMINISTRATOR;
4     } else {
5         $_SESSION ['APPLICATION_LEVEL_ROLE'] = Config::APPLICATION_LEVEL_ROLE_USER;
6     }
7 }
8
9 public static function getApplicationLevelRole() {
10     if (isset ( $_SESSION ['APPLICATION_LEVEL_ROLE'] )) {
11         return $_SESSION ['APPLICATION_LEVEL_ROLE'];
12     }
13     return -1;
14 }
```

The repository level roles, on the other hand, contain the repository level privileges. These define what the user is allowed to do and see within the repository context. These privileges define, for instance, if the user is allowed to edit the accounting data, or upload documents.

The repository level roles are: repository owner, client and accountant. They are kept in the database table `authorizations`, and are assigned during the authorization of the new repository user. This enables one application user to have multiple different repository level roles for different repositories. The role repository owner will be automatically assigned to the user that is creating a new repository, but a current repository owner can also assign it to newly authorized repository users.

The repository level role is also kept within a session variable. This variable is set alongside other repository parameters when a user connects to a repository, and is cleared when the user disconnects (Listing 3.21).

Listing 3.21: Listing for the `setRepositoryLevelRole` and `getRepositoryLevelRole` functions from the `Session` class

```
1 public static function setRepositoryLevelRole($role) {
2     $_SESSION ['REPOSITORY_LEVEL_ROLE'] = $role;
3 }
4
5 public static function getRepositoryLevelRole() {
6     if (isset ( $_SESSION ['REPOSITORY_LEVEL_ROLE'] )) {
7         return $_SESSION ['REPOSITORY_LEVEL_ROLE'];
8     }
9     return -1;
10 }
```

3.2.2 Privileges

The privileges determine which functionality is accessible to the user. They are, just as roles, divided into two groups: the application privileges and the repository privileges. The application privileges can be assigned to the application level roles while the repository privileges are assigned to the repository level roles.

The application level privileges (table 3.1) are used to restrict access to certain application functionality such as user management, and are not customisable. The application level privilege checking is done directly in the code by checking if the current user has an appropriate application role to access the target functionality, and being that there will be only two available application level roles, only the administrators will be allowed to access the application areas that contain the shared data.

3 Implementation

Privilege	Description
AP_ACCESS_APPLICATION_LOG	Privilege to access the application log containing both the server-side as well as the client-side application errors
AP_ACCESS_USER_LOG	Privilege to access the user log containing the sign in timestamps
AP_ACCESS_USER_REQUESTS	Privilege to access the user requests including the account activation and password reset requests
AP_ACCESS_USER_MGMT	Privilege to access the user management interface
AP_ACCESS_REPOSITORY_MGMT	Privilege to access the repository management interface
AP_ACCESS_AUTHORIZATION_MGMT	Privilege to access the authorization management interface

Table 3.1: Application level privileges

Application privilege checking is implemented in the PHP class `Permissions` (Listing 3.22). Being that the application role is determined when the user successfully signs in to the application and stored in the session data, the application will only check if the user is an administrator in order to give clearing for the given application privilege.

Listing 3.22: Listing for the `userHasApplicationPrivilege` function from the `Permissions` class

```
1 public static function userHasApplicationPrivilege($privilege) {
2     if (! in_array ( $privilege, self::AP_PRIVILEGES )) {
3         return false;
4     }
5
6     $role = Session::getApplicationLevelRole ();
7     if (Config::APPLICATION_LEVEL_ROLE_ADMINISTRATOR == $role) {
8         return true;
9     } else {
10        return false;
11    }
12 }
```

The repository privileges, on the other hand, determine which functionality is available to the user within the repository context. These are listed in the table 3.2.

3.2 User Management

Privilege	Description
RP_VIEW_ACCOUNTING_DATA	Privilege to view the accounting data
RP_EDIT_ACCOUNTING_DATA	Privilege to edit the accounting data
RP_UPLOAD_DOCUMENT	Privilege to upload documents
RP_DELETE_DOCUMENT	Privilege to delete documents
RP_CREATE_DIRECTORY	Privilege to create new directories
RP_RENAME_DIRECTORY	Privilege to rename directories
RP_DELETE_DIRECTORY	Privilege to delete directories
RP_ADD_REPOSITORY_USER	Privilege to give repository clearance to other users
RP_REMOVE_REPOSITORY_USER	Privilege to revoke repository clearance from other users
RP_EDIT_REPOSITORY_DATA_FIELDS	Privilege to add or remove repository data fields
RP_EDIT_REPOSITORY_DATA_LAYOUTS	Privilege to add, remove or edit repository data layouts
RP_EDIT_REPOSITORY_PARAMETERS	Privilege to add or remove repository field parameters
RP_EDIT_REPOSITORY_PRIVILEGES	Privilege to modify repository privileges

Table 3.2: Repository level privileges

3 Implementation

Almost all of the repository privileges are customisable and can be set for each repository individually. The only exception is the privilege to modify the repository privileges. This privilege is reserved exclusively for the repository owner role. The user interface that allows the privilege modification is available from the repository dashboard (Figure 3.1).

REPOSITORY PRIVILEGES // MY REPOSITORY			
	Client	Accountant	Repository Owner
View accounting data	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Edit accounting data	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Upload documents	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Delete documents	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Create directory	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Rename directory	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Delete directory	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Add user	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Remove user	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Edit data fields	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Edit data layouts	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Edit parameters	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Edit privileges	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 3.1: Repository privileges dashboard

Repository privileges are checked on two levels (Listing 3.23). When the repository owner modifies the privileges for the first time, three records are created in the shared database table `privileges`, one for each repository role. Therefore the application always checks if there is a custom privilege definition for the given repository first by searching in the database table `privileges`. If the `privileges` table contains no records for the given repository then the default repository privileges are used. These are defined directly in the application code.

Listing 3.23: Listing for the function `getRepositoryPrivileges` from the `Permissions` class

```

1  public static function getRepositoryPrivileges($repositoryId) {
2      $defaultPrivileges = self::getDefaultRepositoryPrivileges ();
3      $repositoryPrivileges = array (
4          Authorizations::ROLE_REPOSITORY_OWNER => 0,
5          Authorizations::ROLE_CLIENT => 0,
6          Authorizations::ROLE_ACCOUNTANT => 0
7      );
8
9      $privilegesModel = new Privileges ( DatabaseAdapter::getPDODatabaseHandle ( true ) );
10     $rolePrivilege = $privilegesModel->getPrivileges ( $repositoryId, Authorizations::
11         ROLE_REPOSITORY_OWNER );
12     if (null == $rolePrivilege) {
13         $repositoryPrivileges [Authorizations::ROLE_REPOSITORY_OWNER] = $defaultPrivileges [
14             Authorizations::ROLE_REPOSITORY_OWNER];
15     } else {
16         $repositoryPrivileges [Authorizations::ROLE_REPOSITORY_OWNER] = $rolePrivilege->privileges;
17     }
18     $rolePrivilege = $privilegesModel->getPrivileges ( $repositoryId, Authorizations::ROLE_CLIENT
19         );
20     if (null == $rolePrivilege) {
21         $repositoryPrivileges [Authorizations::ROLE_CLIENT] = $defaultPrivileges [Authorizations::
22             ROLE_CLIENT];
23     } else {
24         $repositoryPrivileges [Authorizations::ROLE_CLIENT] = $rolePrivilege->privileges;
25     }
26     $rolePrivilege = $privilegesModel->getPrivileges ( $repositoryId, Authorizations::
27         ROLE_ACCOUNTANT );
28     if (null == $rolePrivilege) {
29         $repositoryPrivileges [Authorizations::ROLE_ACCOUNTANT] = $defaultPrivileges [Authorizations
30             ::ROLE_ACCOUNTANT];
31     } else {
32         $repositoryPrivileges [Authorizations::ROLE_ACCOUNTANT] = $rolePrivilege->privileges;
33     }
34     return $repositoryPrivileges;
35 }

```

3.2.3 Account Creation

In order to be able to use the application, the user needs to create an account first. The account creation process is carried out in two steps.

In the first step, the user must fill out the account creation form, which is located on the index page (Figure 3.2). In order to create an account, the user must provide a valid email address, which will serve as a username, the users first- and last-name, and a password. Upon clicking the "Create account" button the data is sent to the server by means of AJAX. On the server-side the application then creates a record in the `users` table with the status registered, as well as a record in the `user_requests` table with a request type account activation.

3 Implementation

The screenshot shows a web interface for 'cloud bookkeeping' with two main sections: 'CREATE ACCOUNT' and 'SIGN IN'. The 'CREATE ACCOUNT' section has four input fields for 'E-MAIL', 'FIRST NAME', 'LAST NAME', and 'PASSWORD', followed by a blue 'Create account' button. The 'SIGN IN' section has two input fields for 'E-MAIL' and 'PASSWORD', a blue 'Sign in' button, and a link for 'Forgot password?' below it.

Figure 3.2: Account creation form

In the second step, a verification email message is generated and sent to the provided email address. The message will contain an activation link that contains a token and redirects the user to the index page where they can sign into the application for the first time. The index page will recognize that a token is provided within the URL, and it will send it alongside the email address and the password when signing in the user. Before checking the credentials, the application will first check the user's status. If the status is registered then the token must be provided and checked for its validity. Being that the token consists of the encrypted user request data, it needs to be decrypted first and then verified (Listing 3.24). If the token data matches the data in the `user_requests` table, the user will be signed into the application and redirected to the overview page, while the status of the record in the `users` table is updated to active, and the token invalidated by updating the `activity_time` in the corresponding record of the `user_requests` table.

Listing 3.24: Listing for the functions `getTokenData` and `checkToken` from the class `Authentication`

```
1 public static function getTokenData($token) {
2     $data = Cryptography::decrypt ( $token );
3     $userRequestId = intval ( substr ( $data, 0, 10 ) );
```

3.2 User Management

```
4     $userId = intval ( substr ( $data, 10, 10 ) );
5     $requestTime = intval ( substr ( $data, 20, 20 ) );
6     $requestType = intval ( substr ( $data, 40, 1 ) );
7     return array (
8         $userRequestId,
9         $userId,
10        $requestType,
11        $requestTime
12    );
13 }
14
15 protected function checkToken($token, $requestType, $username = null) {
16     $tokenData = self::getTokenData ( $token );
17
18     if ($tokenData [2] != $requestType) {
19         Log::logError ( 'Invalid token [request type invalid]: ' . $token );
20         return false;
21     }
22
23     if (null != $username) {
24         $users = new Users ( DatabaseAdapter::getPDODatabaseHandle () );
25         $user = $users->getUser ( $tokenData [1] );
26         if (null != $user && 0 != strcmp ( trim ( $username ), trim ( $user->username ) )) {
27             Log::logError ( 'Invalid token [username mismatch]: ' . $token );
28             return false;
29         }
30     }
31
32     $userRequests = new UserRequests ( DatabaseAdapter::getPDODatabaseHandle () );
33     if (! $userRequests->authenticateUserRequest ( $tokenData [0], $tokenData [1], $requestType,
34         $tokenData [3] )) {
35         Log::logError ( 'Invalid token [user request invalid]: ' . $token );
36         return false;
37     }
38     return true;
39 }
```

3.2.4 Signing In to Application

After the user has successfully created an account, they can use the sign in form on the index page to sign into the application. The sign in form sends the user credentials to the server by means of an AJAX request. On the server-side, the user authentication is done in the php class Authentication (Listing 3.25).

Listing 3.25: Listing for the function `signIn` from the class `Authentication`

```
1 public function signIn($username, $password, $token) {
2     $pdoHandle = DatabaseAdapter::getPDODatabaseHandle ();
3     $users = new Users ( $pdoHandle );
4     $userLog = new UserLog ( $pdoHandle );
5
6     $user = $users->authenticateUser ( $username, $password );
7     if (null != $user) {
8         $activation = false;
9         if (Users::STATUS_REGISTERED == $user->user_status) {
10            if (null == $token) {
11                throw new AuthAccountNotActivatedException ( Config::get ( 'EXCEPTION_ACCOUNT_REGISTERED'
12                ) );
13            }
14        }
15    }
16 }
```

3 Implementation

```
13         if (! self::checkToken ( $token, UserRequests::REQUEST_TYPE_ACCOUNT_ACTIVATION, $username
14     ) ) {
15             throw new AuthInvalidTokenException ( Config::get ( 'EXCEPTION_INVALID_TOKEN' ) );
16         }
17         $activation = true;
18     }
19     if ( $user->success ) {
20         $session = new Session ();
21         $session->start ( true );
22         $session->setValid ( true );
23         Session::setUserId ( $user->user_id );
24         Session::setUserDisplayName ( $user->first_name, $user->last_name );
25         Session::setApplicationLevelRole ( $username );
26         $userLog->writeUserLog ( $user->user_id, UserLog::SIGN_IN_SUCCESS );
27         if ( $activation ) {
28             $users->updateUserStatus ( $user->user_id, Users::STATUS_ACTIVE );
29             self::invalidateToken( $token );
30         }
31         return true;
32     } else {
33         $userLog->writeUserLog ( $user->user_id, UserLog::SIGN_IN_FAILURE );
34         if ( Users::STATUS_LOCKED == $user->user_status ) {
35             throw new AuthUserLockedException ( Config::get ( 'EXCEPTION_AUTH_USER_LOCKED' ) );
36         } else {
37             if ( $userLog->checkBruteForceAttack ( $user->user_id ) ) {
38                 $users->updateUserStatus ( $user->user_id, Users::STATUS_LOCKED );
39                 throw new AuthUserLockedException ( Config::get ( 'EXCEPTION_AUTH_USER_LOCKED' ) );
40             }
41             throw new AuthBadPasswordException ( Config::get ( 'EXCEPTION_AUTH_BAD_PASSWORD' ) );
42         }
43     }
44 } else {
45     return false;
46 }
47 }
```

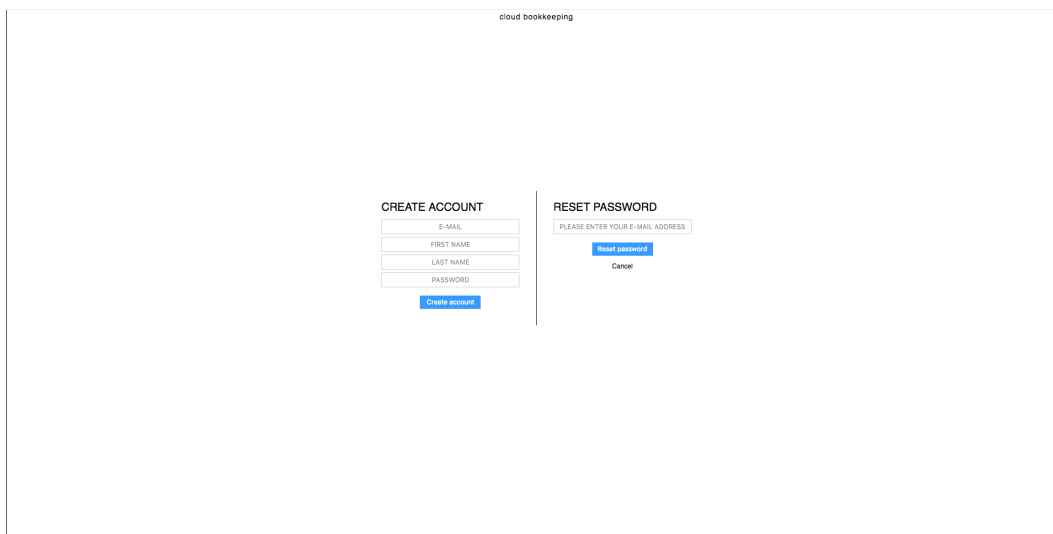
When signing in the user, the application will first check the user's status. If the status is registered, the user will not be able to sign in before completing the second step of the account creation process as described in chapter 3.2.3. If the account status is active, the user's password is verified, and if it matches the stored password, the user will be signed in by starting a session and initializing the session variables.

If the user provides a wrong password three consecutive times, the account will be locked. This is due to the brute force attack checking. On any failed sign in attempt, the applications searches the user_log table to check if there are three consecutive failure records for the given user. If so, the application locks the user by setting the user account status to locked in the users table.

Once the account has been locked, the user will need to reset the password by using the password reset request form on the index page. The form is available by clicking on the "Forgot password" link. After entering the email address, the user will receive an email that contains a token and redirects

3.2 User Management

the user to the password reset form where they can enter and confirm the new password. Once the new password is saved the user is redirected to the index page and can now sign into the application. If the account has not yet been activated then it will not be possible to perform a password reset.



The screenshot shows a web interface for 'cloud bookkeeping'. It features two main sections: 'CREATE ACCOUNT' and 'RESET PASSWORD'. The 'CREATE ACCOUNT' section has four input fields for 'E-MAIL', 'FIRST NAME', 'LAST NAME', and 'PASSWORD', followed by a 'Create account' button. The 'RESET PASSWORD' section has a single input field for 'PLEASE ENTER YOUR E-MAIL ADDRESS', followed by 'Reset password' and 'Cancel' buttons.

Figure 3.3: Password reset form

3.2.5 Signing Out of Application

When the user wishes to leave the application, it is recommended to do so by signing out of it. To sign out, the user only needs to click on the user's name or on the sign out icon. Both of them are located in the lower right corner of the application screen (Figure 3.4).

3 Implementation

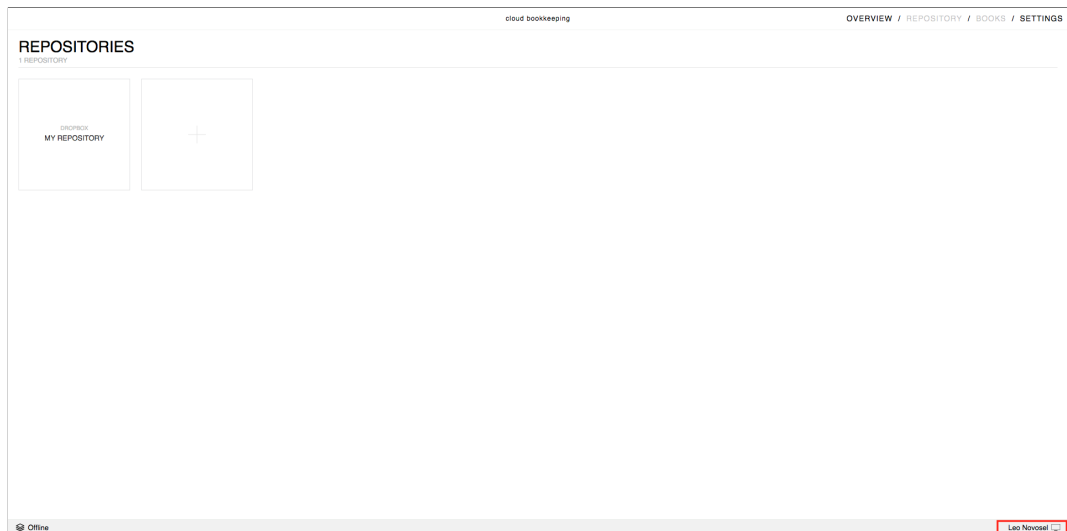


Figure 3.4: Sign out icon

When the user signs out, the application will automatically end the session. This is done in the Authentication php class (Listing 3.26).

Listing 3.26: Listing for the function `signOut` from the class `Authentication`

```
1 public function signOut() {
2     $session = new Session ();
3     $session->start ();
4     $session->end ();
5 }
```

3.3 Repository Management

The repository management contains all of the functionality needed to create and manage a document repository. This chapter discusses the concepts behind the repository management within the web application, as well as some of the implementation aspects.

3.3.1 Creation of Document Repository

In order to be able to use a document repository, the user needs to create one first. The creation of a repository within the web application is conducted in only a few simple steps. New repository can be created from the overview page by clicking on the repository placeholder (Figure 3.5). First, the user must select which cloud storage service they want to use for the repository, and then enter a name for the new repository.



Figure 3.5: Repository placeholder

In the next step the web application generates the authorization URI and redirects the user to the chosen cloud storage service where they can authorize the application by using the OAuth 2.0 flow as described in the chapter 2.2.2. The OAuth 2.0 flow is implemented for each of the cloud services individually in a separate PHP class, and each of these classes implements the `CloudServiceInterface` (Listing 3.27). This interface provides all of the functionality needed to interact with the cloud storage service.

Listing 3.27: Listing for the interface `CloudServiceInterface`

```
1 interface CloudServiceInterface {
```

3 Implementation

```
2 public function startOAuth2Process();
3 public function finishOAuth2Process(Array $response);
4 public function getAuthorizationURL();
5 public function getRootPath();
6 public function connect();
7 public function listDirectory($path);
8 public function createDirectory($path, $directory);
9 public function delete($path);
10 public function download($path, &$file);
11 public function upload($path, $filename, &$file, $localPath);
12 }
```

The OAuth 2.0 process is started by calling the function `startOAuth2Process` of the `Authorization` class (Listing 3.28), which returns the authorization URI to which the user is redirected to, and once the user authorizes the web application, they are redirected back to the overview page (i.e. redirect URI). The overview page first checks the query string and then again uses the `Authorization` class to instantiate the appropriate implementation of the `CloudServiceInterface`, and to call its `finishOAuth2Process` function. The function `finishOAuth2Process` handles the last step in the OAuth 2.0 process by exchanging the authorization code for the reusable access token.

Listing 3.28: Listing for the class `Authorization`

```
1 class Authorization {
2
3     public static function startOAuth2Process($cloudServiceId) {
4         $cloudService = CloudServiceFactory::createCloudService ( $cloudServiceId );
5         return $cloudService->startOAuth2Process ();
6     }
7
8     public static function finishOAuth2Process($cloudServiceId, $response) {
9         $cloudService = CloudServiceFactory::createCloudService ( $cloudServiceId );
10        return $cloudService->finishOAuth2Process ( $response );
11    }
12 }
```

Once the web application has attained the reusable access token, it saves all the repository related data in the database by creating a new record in the `repositories` table.

3.3.2 Connecting to a Document Repository

In order to work with the document repository, the user must connect to it first. This functionality does not actually create any type of a persistent

3.3 Repository Management

connection, but is rather used to initialize the web application's session variables that hold the information regarding the repository as well as the database connection parameters and cloud storage service access token. Besides the session variable initialization, the connect function can also be used to try to perform an action on the cloud storage service, for instance, to check if the access token is still valid.

The implementation of the connect functionality is found in the function `methodConnectRepository` from the class `RPCEndpointInterface` (Listing 3.29).

Listing 3.29: Listing for the function `methodConnectRepository`

```
1 protected static function methodConnectRepository($params, &$response) {
2     try {
3         $authorizations = new Authorizations ( DatabaseAdapter::getPDODatabaseHandle () );
4         $authorization = $authorizations->getAuthorization ( $params [self::PARAM_AUTHORIZATION_ID] )
5         ;
6         $repositories = new Repositories ( DatabaseAdapter::getPDODatabaseHandle () );
7         $authParameter = $repositories->getAuthParameter ( $params [self::PARAM_AUTHORIZATION_ID],
8             Session::getUserId () );
9
10        if (null != $authParameter) {
11            Session::setCloudServiceParameter ( $authParameter->service_id, $authParameter->
12                service_token, $authParameter->repository_id, $authParameter->repository_name );
13            Session::setRepositoryDatabaseParameter ( $authParameter->database_user, $authParameter->
14                database_password );
15            Session::setRepositoryLevelRole ( $authorization->user_role );
16            Session::setRepositoryPrivileges ( Permissions::getRepositoryPrivileges ( $authParameter->
17                repository_id ) [$authorization->user_role] );
18            Session::setRepositoryDataLayout ( $authParameter->layout_id );
19            // Check the connection to the cloud service
20            $cloudService = CloudServiceFactory::createCloudService ( Session::getCloudServiceId () );
21            if (null == $cloudService || !$cloudService->connect () ) {
22                Session::setCloudServiceParameter ( null, null, null, null );
23                Session::setRepositoryDatabaseParameter ( null, null );
24                Session::setRepositoryLevelRole ( null );
25                Session::setRepositoryPrivileges ( null );
26                Session::setRepositoryDataLayout ( null );
27                self::responseError ( $response, self::RC_FAILURE, 'CONNECTION_ERROR', TRUE );
28                return false;
29            }
30            self::responseStatus ( $response, self::RC_SUCCESS, 'REPOSITORY_CONNECTED', TRUE );
31            return true;
32        } else {
33            self::responseError ( $response, self::RC_FAILURE, 'CONNECTION_ERROR', TRUE );
34            return false;
35        }
36    } catch ( AccessTokenException $exception ) {
37        Log::logException ( $exception );
38        self::responseError ( $response, self::RC_INVALID_ACCESS_TOKEN, 'Invalid access token' );
39    } catch ( NetworkException $exception ) {
40        Log::logException ( $exception );
41        self::responseError ( $response, self::RC_FAILURE, 'NETWORK_ERROR', TRUE );
42    } catch ( Exception $exception ) {
43        Log::logException ( $exception );
44        self::responseError ( $response, self::RC_FAILURE, 'CONNECTION_FAILURE', TRUE );
45    }
46    Session::setCloudServiceParameter ( null, null, null, null );
47    Session::setRepositoryDatabaseParameter ( null, null );
48    Session::setRepositoryLevelRole ( null );
49    Session::setRepositoryPrivileges ( null );
50}
```

3 Implementation

```
46     Session::setRepositoryDataLayout ( null );  
47     return false;  
48 }
```

To connect to a repository, the user must first select a repository from the overview page. This will bring up the repository dashboard, from where the user can use the "CONNECT" button to connect to the chosen repository (Figure 3.6).

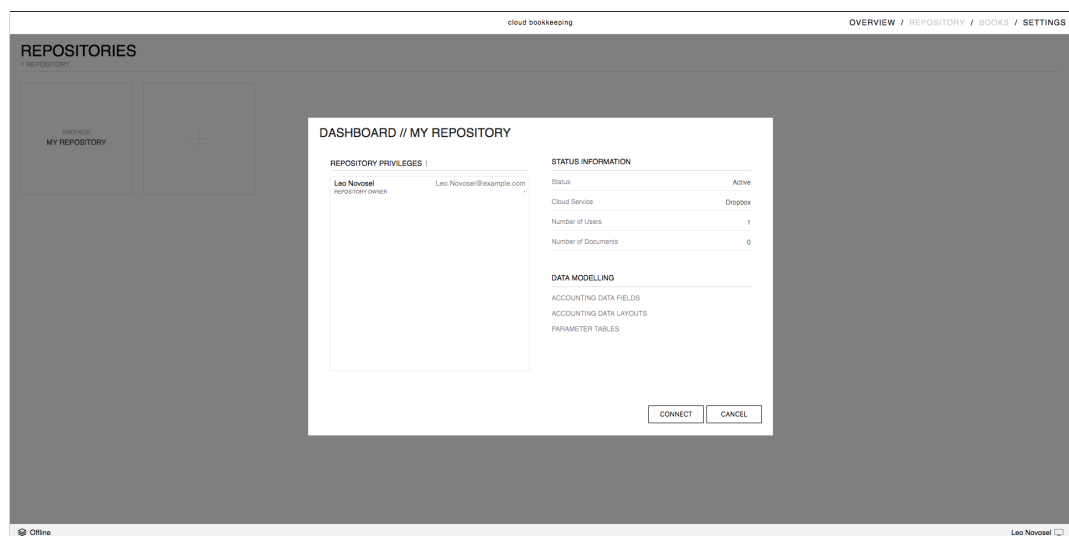


Figure 3.6: Repository dashboard

Once the repository is connected, the user can either navigate to the repository page by using the navigation bar in the top right corner of the application screen, or edit the repository properties (e.g. add repository users, create accounting data fields etc.).

3.3.3 Repository User Management

The repository user management interface is available through the repository dashboard (Figure 3.7). By default, it allows the user with a repository

3.3 Repository Management

owner role to authorize additional users to work with the repository, or to revoke their repository access, although this privilege can be assigned to any role from the repository privileges dashboard.

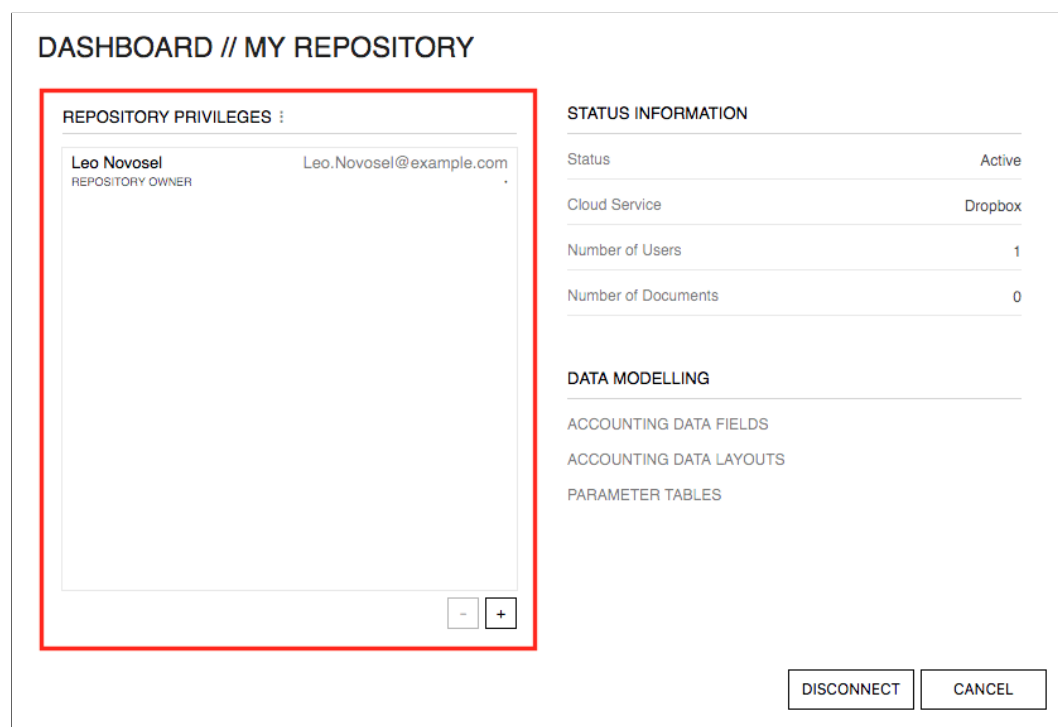


Figure 3.7: Repository user management interface within the repository dashboard

To authorize an additional user, the repository owner must click the "+" button, and then enter a username of an already registered application user. Additionally, one of the three available repository roles must be assigned to the new repository user (Figure 3.8). This role will only be valid within the currently connected repository, which means that each application user can have authorizations for multiple different repositories, each with a different role.

3 Implementation

ADD REPOSITORY USER
Please enter a user you want to authorize

USERNAME

Repository Owner
 Client
 Accountant

ADD USER CANCEL

Figure 3.8: Repository user authorization dialog

Once the needed information has been entered, and the user clicks the “ADD USER” button, the client-side function `LNAPP.xhr.authorizeRepositoryUser` generates a JSON-RPC request and sends it to the server. On the server-side the request is processed within the function `methodAuthorizeRepositoryUser` from the class `RPCEndpointInterface` (Listing 3.30).

Listing 3.30: Listing for the function `methodAuthorizeRepositoryUser`

```
1 protected static function methodAuthorizeRepositoryUser($params, &$amp;response) {
2     try {
3         if (! Permissions::userHasRepositoryPrivilege ( 'RP_ADD_REPOSITORY_USER' )) {
4             self::responseError ( $response, self::RC_FAILURE, 'INSUFFICIENT_PRIVILEGES', TRUE );
5             return false;
6         }
7
8         $users = new Users ( DatabaseAdapter::getPDODatabaseHandle () );
9         $user = $users->getUserid ( $params [self::PARAM_USERNAME] );
10        if (null === $user) {
11            self::responseError ( $response, self::RC_FAILURE, 'USER_AUTHORIZATION_ERROR', TRUE );
12            return false;
13        }
14
15        $authorizations = new Authorizations ( DatabaseAdapter::getPDODatabaseHandle () );
16        $authorization = $authorizations->createAuthorization ( Session::getCloudRepositoryId (),
17            $user->user_id, $params [self::PARAM_ROLE] );
18
19        if (null != $authorization) {
20            $repositoryUsers = $users->getUsers ( Session::getCloudRepositoryId () );
21            foreach ( $repositoryUsers as $repositoryUser ) {
22                $authorizedUser [self::PARAM_USER_ID] = $repositoryUser->user_id;
23                $authorizedUser [self::PARAM_FIRST_NAME] = $repositoryUser->first_name;
```

3.3 Repository Management

```
23     $authorizedUser [self::PARAM_LAST_NAME] = $repositoryUser->last_name;
24     $authorizedUser [self::PARAM_USERNAME] = $repositoryUser->username;
25     $authorizedUser [self::PARAM_ROLE] = strtoupper ( View::getRepositoryLevelRoleLabel (
    $repositoryUser->user_role ) );
26     $repositoryUsersArray [] = $authorizedUser;
27 }
28 self::addResponseMember ( $response, self::PARAM_REPOSITORY_USERS, json_encode (
    $repositoryUsersArray ) );
29 self::responseStatus ( $response, self::RC_SUCCESS, 'USER_AUTHORIZED', TRUE );
30 return true;
31 } else {
32     self::responseError ( $response, self::RC_FAILURE, 'USER_AUTHORIZATION_ERROR', TRUE );
33     return false;
34 }
35 } catch ( Exception $exception ) {
36     Log::logException ( $exception );
37     self::responseError ( $response, self::RC_FAILURE, 'USER_AUTHORIZATION_FAILURE', TRUE );
38 }
39 return false;
40 }
```

3.3.4 Accounting Data Fields

As discussed in the chapter 1.5.4, the users have a possibility to define additional accounting data fields. This can be achieved through the repository accounting data fields dashboard (Figure 3.9). The dashboard contains the list of currently available data fields, and a possibility to add a new one by clicking the "NEW DATA FIELD" button.

The list of fields contains information regarding the individual data fields. The column "Label" contains the textual description that is visible in the user interface, while the column "FieldId" contains the id that is used when sending accounting data as a parameter using a JSON-RPC request. The column "DataType" indicates the application internal data type, and the column "Null" indicates whether the field is optional or obligatory. The "Parameter Table" column contains the id of the parameter table that is used for the data field in the user interface.

3 Implementation

ACCOUNTING DATA FIELDS // MY REPOSITORY				
Label	Field	DataType (Size)	Null	Parameter Table
Document ID	data_1	varchar(255)	YES	-
Text	data_2	varchar(255)	YES	-

Figure 3.9: Repository accounting data fields dashboard

In order to create an new accounting data field, the user only needs to enter a label for the new field (i.e. a name), and the datatype (Figure 3.10). The data type is then internally mapped to an SQL data type in the PHP class `DataType`.

Figure 3.10: Repository data field creation dialog

When saving a new data field, the JSON-RPC request is created in the function `LNAPP.xhr.addRepositoryDataField`, and on the server-side processed within the function `methodCreateRepositoryDataField` from the class `RPCEndpointInterface` (Listing 3.31).

Listing 3.31: Listing for the function `methodCreateRepositoryDataField`

```

1  protected static function methodCreateRepositoryDataField($params, &$response) {
2      try {
3          if (! Permissions::userHasRepositoryPrivilege ( 'RP_EDIT_REPOSITORY_DATA_FIELDS' )) {
4              self::responseError ( $response, self::RC_FAILURE, 'INSUFFICIENT_PRIVILEGES', TRUE );
5              return false;
6          }
7
8          $databaseHandle = DatabaseAdapter::getPDODatabaseHandle ();
9          $databaseHandle->beginTransaction ();
10
11         $dataType = new DataType ();
12         $parameterTableId = null;
13         if (DataType::DATA_TYPE_PARAMETER == $params [self::PARAM_DATA_FIELD_TYPE]) {
14             if (! array_key_exists ( self::PARAM_DATA_FIELD_PARAM, $params )) {
15                 self::responseError ( $response, self::RC_FAILURE, 'ADDING_DATA_FIELD_ERROR', TRUE );
16                 $databaseHandle->rollback ();
17                 return false;
18             }
19             $parameterTableId = $params [self::PARAM_DATA_FIELD_PARAM];
20             $columnDataType = $dataType->getDatatype ( $params [self::PARAM_DATA_FIELD_TYPE], $params [
21                 self::PARAM_DATA_FIELD_LENGTH], $parameterTableId );
22         } else {
23             $columnDataType = $dataType->getDatatype ( $params [self::PARAM_DATA_FIELD_TYPE], $params [
24                 self::PARAM_DATA_FIELD_LENGTH] );
25         }
26     }
27 }

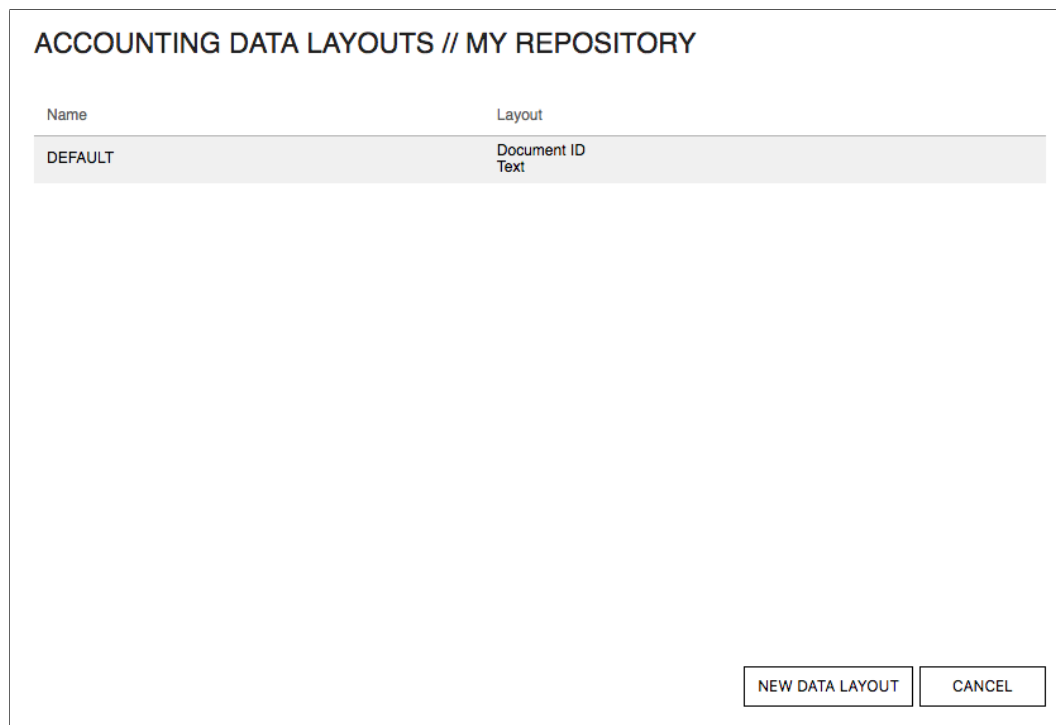
```

3 Implementation

```
24
25 if ('' != $columnDataType) {
26     // Create record in the accounting data fields table
27     $accountingDataFields = new AccountingDataFields ( $databaseHandle );
28     $fieldId = $accountingDataFields->createAccountingDataField ( Session::
getCloudRepositoryId (), $params [self::PARAM_DATA_FIELD_LABEL], $parameterTableId );
29
30     if (null === $fieldId) {
31         self::responseError ( $response, self::RC_FAILURE, 'ADDING_DATA_FIELD_ERROR', TRUE );
32         $databaseHandle->rollback ();
33         return false;
34     }
35
36     $columnId = AccountingData::DATA_FIELD_PREFIX . $fieldId;
37
38     // Create column in the accounting data table
39     $accountingData = new AccountingData ( $databaseHandle );
40     $result = $accountingData->createAccountingDataField ( Session::getCloudRepositoryId (),
$columnId, $columnDataType );
41
42     if (! $result) {
43         self::responseError ( $response, self::RC_FAILURE, 'ADDING_DATA_FIELD_ERROR', TRUE );
44         $databaseHandle->rollback ();
45         return false;
46     }
47
48     // Create column in the accounting data revisions table
49     $accountingDataRevisions = new AccountingDataRevisions ( $databaseHandle );
50     $result = $accountingDataRevisions->createAccountingDataRevisionsField ( Session::
getCloudRepositoryId (), $columnId, $columnDataType );
51
52     if (! $result) {
53         self::responseError ( $response, self::RC_FAILURE, 'ADDING_DATA_FIELD_ERROR', TRUE );
54         $databaseHandle->rollback ();
55         return false;
56     }
57
58     // Update default accounting data layout
59     $accountingDataLayouts = new AccountingDataLayouts ( $databaseHandle );
60     $result = $accountingDataLayouts->updateDefaultAccountingDataLayout ( Session::
getCloudRepositoryId (), $columnId );
61     if (! $result) {
62         self::responseError ( $response, self::RC_FAILURE, 'ADDING_DATA_FIELD_ERROR', TRUE );
63         $databaseHandle->rollback ();
64         return false;
65     }
66
67     $dataFields = $accountingDataFields->getAccountingDataFields ( Session::
getCloudRepositoryId () );
68     if (null !== $dataFields) {
69         self::addResponseMember ( $response, self::PARAM_DATA_FIELDS, json_encode ( $dataFields )
);
70     }
71
72     $databaseHandle->commit ();
73     self::responseStatus ( $response, self::RC_SUCCESS, 'DATA_FIELD_CREATED', TRUE );
74     return true;
75 } else {
76     self::responseError ( $response, self::RC_FAILURE, 'ADDING_DATA_FIELD_ERROR', TRUE );
77     $databaseHandle->rollback ();
78     return false;
79 }
80 } catch ( Exception $exception ) {
81     Log::logException ( $exception );
82     self::responseError ( $response, self::RC_FAILURE, 'ADDING_DATA_FIELD_FAILURE', TRUE );
83     $databaseHandle->rollback ();
84 }
85 return false;
86 }
```


3.3.5 Accounting Data Layouts

In addition to the possibility to create additional data fields, the users also have the possibility to define which accounting data fields are going to appear in the user interface, and in which order. They can do that from the accounting data layouts dashboard (Figure 3.11). This dashboard contains a list of the currently available layouts, as well as their content. Additionally, by clicking the "NEW DATA LAYOUT" button, the users can create their own custom layout.



ACCOUNTING DATA LAYOUTS // MY REPOSITORY

Name	Layout
DEFAULT	Document ID Text

NEW DATA LAYOUT CANCEL

Figure 3.11: Repository accounting data layouts dashboard

The accounting data layout creation is implemented by using a simple drag-and-drop user interface (Figure 3.12). The left table contains all of the data fields that are available, while the right table contains the data fields that are going to be part of the new layout. The fields can be dragged and dropped

3 Implementation

both between tables, as well as within a single table in order to define the order in which the fields will appear in the user interface. Once the user has defined which fields should appear in the layout, and in which order, they only need to enter a name for the new layout before saving it. If the user wants to change the default layout for the repository, they only need to click on it in the accounting data layouts dashboard.

ACCOUNTING DATA LAYOUTS // MY REPOSITORY

NEW DATA LAYOUT

AVAILABLE DATA FIELDS				SELECTED DATA LAYOUT FIELDS			
Document ID	data_1	varchar(255)	-	Text	data_2	varchar(255)	-

SAVE CANCEL

Figure 3.12: Repository accounting data layout creation dialog

The new data layout is saved by performing a JSON-RPC request created in the function `LNAPP.xhr.addRepositoryDataLayout`, while the server-side logic is implemented within the function `methodCreateRepositoryDataLayout` from the class `RPCEndpointInterface` (Listing 3.32).

Listing 3.32: Listing for the function `methodCreateRepositoryDataLayout`

```
1 protected static function methodCreateRepositoryDataLayout($params, &$response) {
```

3.3 Repository Management

```
2     try {
3         if (! Permissions::userHasRepositoryPrivilege ( 'RP_EDIT_REPOSITORY_DATA_LAYOUTS' )) {
4             self::responseError ( $response, self::RC_FAILURE, 'INSUFFICIENT_PRIVILEGES', TRUE );
5             return false;
6         }
7     }
8     $accountingDataLayouts = new AccountingDataLayouts ( DatabaseAdapter::getPDODatabaseHandle
9         ( ) );
10    $result = $accountingDataLayouts->createAccountingDataLayout ( Session::getCloudRepositoryId
11        ( ), $params [self::PARAM_DATA_LAYOUT_NAME], $params [self::PARAM_DATA_LAYOUT] );
12    if (null == $result) {
13        self::responseError ( $response, self::RC_FAILURE, 'ADDING_DATA_LAYOUT_ERROR', TRUE );
14        return false;
15    }
16    self::responseStatus ( $response, self::RC_SUCCESS, 'DATA_LAYOUT_CREATED', TRUE );
17    return true;
18 } catch ( Exception $exception ) {
19     Log::logException ( $exception );
20     self::responseError ( $response, self::RC_FAILURE, 'ADDING_DATA_LAYOUT_FAILURE', TRUE );
21 }
return false;
}
```

3.3.6 Parameter Tables

Parameter tables allow the user to create predefined key-value entries that can be used as options for the accounting data field's select HTML element in the user interface. By using the parameter tables the user can define allowed values for a certain data field, and by that reduce the possibility of an incorrect input value. To associate an accounting data field with a certain parameter table, the user must use the "Parameter" data type and select the wanted parameter table while creating a new data field in the repository data field creation dialog (Figure 3.10).

The handling of the parameter tables is implemented in the PHP class Parameters (Listing 3.33). When the parameter tables are loaded, the application reads both application default parameter tables, as well as the repository specific ones.

Listing 3.33: Listing for the class Parameters

```
1     class Parameters {
2
3         protected $tables_;
4
5         public function __construct($repositoryId = NULL) {
6             $this->tables_ = array ();
7             if (null === $repositoryId) {
8                 if (null !== Session::getCloudRepositoryId ()) {
9                     $repositoryId = Session::getCloudRepositoryId ();
10                }
11            }
12        }
13    }
```

3 Implementation

```
12
13     $parametersDir = array_diff ( scandir ( Config::get ( 'PARAMETERS_PATH' ) ), array (
14         '.htaccess',
15         '..',
16         '.'
17     ) );
18
19     $customParametersDir = null;
20     if (null !== $repositoryId && file_exists ( Config::get ( 'PARAMETERS_PATH' ) . 'custom_' .
21         $repositoryId )) {
22         $customParametersDir = array_diff ( scandir ( Config::get ( 'PARAMETERS_PATH' ) . 'custom_' .
23             $repositoryId ), array (
24                 '.htaccess',
25                 '..',
26                 '.'
27             ) );
28     }
29     foreach ( $parametersDir as $file ) {
30         $json = file_get_contents ( Config::get ( 'PARAMETERS_PATH' ) . DIRECTORY_SEPARATOR .
31             $file );
32         $table = json_decode ( $json, true );
33         if (null !== $table) {
34             array_push ( $this->tables_, $table );
35         }
36     }
37     if (null !== $customParametersDir) {
38         foreach ( $customParametersDir as $file ) {
39             $json = file_get_contents ( Config::get ( 'PARAMETERS_PATH' ) . 'custom_' .
40                 $repositoryId . DIRECTORY_SEPARATOR . $file );
41             $table = json_decode ( $json, true );
42             if (null !== $table) {
43                 array_push ( $this->tables_, $table );
44             }
45         }
46     }
47     public function getTable($tableId) {
48         foreach ( $this->tables_ as $table ) {
49             if ($tableId == $table ['id']) {
50                 return $table;
51             }
52         }
53         return null;
54     }
55     public function getTables() {
56         return $this->tables_;
57     }
58 }
59 }
```

All parameter tables, as well as their content, are visible in the repository parameter tables dashboard (Figure 3.13). From here, the user can click the “NEW PARAMETER TABLE” to create a new, repository specific, parameter table. Repository specific means that the table will only be available within the currently connected repository. To create a parameter table that is available globally, the JSON file containing the table must be manually placed into the “parameters” directory.

3.3 Repository Management

PARAMETER TABLES // MY REPOSITORY

PARAMETER TABLE ID: ACCOUNT // NAME: Account // KEY DATA TYPE: INTEGER NUMERICAL

Key	Value
0010	Aufwendungen für das Ingangsetzen und Erweitern eines Betriebes
0090	Kumulierte Abschreibungen
0100	Konzessionen
0110	Patentrechte und Lizenzen
0120	Datenverarbeitungsprogramme
0130	Marken, Warenzeichen und Musterschutzrechte, sonstige Urheberrechte
0140	Pacht- und Mietrechte
0150	Geschäfts(Firmen)wert
0180	Geleistete Anzahlungen
0190	Kumulierte Abschreibungen
0200	Unbebaute Grundstücke
0210	Bebaute Grundstücke (Grundwert)
0220	Grundstückleihenrechte

NEW PARAMETER TABLE CANCEL

Figure 3.13: Repository parameter tables dashboard

To create a new custom parameter table, the user must enter the JSON formatted content of the table in the parameter tables creation dialog (Figure 3.14). The table must contain attributes `name`, `id`, `key_data_type`, and a `key-value` array `table`. The value of the attribute `name` represents the name of the table that will be visible in the user interface, while the `id` is used only internally within the application logic. The value of the attribute `key_data_type` designates the data type of the table's keys, while the values must always be strings.

3 Implementation

PARAMETER TABLES // MY REPOSITORY

NEW PARAMETER TABLE

```
{
  "name": "TABLE_NAME",
  "id": "TABLE_ID",
  "key_data_type": DATA_TYPE,
  "table": {
    "KEY": "VALUE",
    "KEY": "VALUE"
  }
}
```

Figure 3.14: Repository parameter tables creation dialog

The content of the new parameter table is uploaded by performing a JSON-RPC request created in the function `LNAPP.xhr.createParameterTable`, while the server-side request handling is implemented within the function `methodCreateRepositoryParameterTable` in the class `RPCEndpointInterface` (Listing 3.34).

Listing 3.34: Listing for the function `methodCreateRepositoryParameterTable`

```
1 protected static function methodCreateRepositoryParameterTable($params, &$response) {
2   try {
3     if (! Permissions::userHasRepositoryPrivilege ( 'RP_EDIT_REPOSITORY_PARAMETERS' )) {
4       self::responseError ( $response, self::RC_FAILURE, 'INSUFFICIENT_PRIVILEGES', TRUE );
5       return false;
6     }
7
8     $parameterTable = $params [self::PARAM_PARAMETER_TABLE];
9     $json = json_decode ( $parameterTable );
10    if (null === $json) {
11      Log::logError ( json_last_error_msg () . ': ' . $parameterTable );
12      self::responseError ( $response, self::RC_FAILURE, json_last_error_msg () );
13      return false;
14    }
15  }
```

3.4 Document and Accounting Data Management

```
15
16     if (Config::get ( 'LABEL_NO_VALUE' ) === View::getAccountingDataFieldDataTypeLabel ( $json->
17         key_data_type )) {
18         self::responseError ( $response, self::RC_FAILURE, 'CREATING_PARM_TABLE_FAILURE', TRUE );
19         return false;
20     }
21
22     $path = Config::get ( 'PARAMETERS_PATH' ) . 'custom' . Session::getCloudRepositoryId ();
23     if (! is_dir ( $path )) {
24         if (! @mkdir ( $path, 0700, true )) {
25             $error = error_get_last ();
26             Log::logError ( "Coud not create directory " . $path . ": " . $error ['message'] );
27             self::responseError ( $response, self::RC_FAILURE, 'CREATING_PARM_TABLE_FAILURE', TRUE )
28             ;
29             return false;
30         }
31
32         $htaccessFile = @fopen ( $path . DIRECTORY_SEPARATOR . '.htaccess', "w" );
33         fwrite ( $htaccessFile, 'Options -Indexes' );
34         fclose ( $htaccessFile );
35     }
36
37     $parameterTableFilePath = $path . DIRECTORY_SEPARATOR . $json->id . '.json';
38     $parameterTableFile = @fopen ( $parameterTableFilePath, "w" );
39     fwrite ( $parameterTableFile, $parameterTable );
40     fclose ( $parameterTableFile );
41
42     self::responseStatus ( $response, self::RC_SUCCESS, 'PARAMETER_TABLE_CREATED', TRUE );
43     return true;
44 } catch ( Exception $exception ) {
45     Log::logException ( $exception );
46     self::responseError ( $response, self::RC_FAILURE, 'CREATING_PARM_TABLE_FAILURE', TRUE );
47 }
48 }
```

3.4 Document and Accounting Data Management

To start working with the repository, the user must use the navigation bar in the upper right corner of the web application to navigate to the repository page by clicking the "REPOSITORY" entry. This will take them to the repository page which allows them to manage the repositories's documents and the accounting data.

The repository page consist of four panels and the PDF document viewer (Figure 3.15). On the left hand side of the application window is the repository browser panel. This panel allows the user to browse the contents of the repository, and to manage the files and directories. In the middle of the window is the document viewer which allows the user to view the PDF documents without having to use an external program or a browser plugin. The accounting data and the accounting records can be found on

3 Implementation

the right hand side of the application window in the accounting data panel and the accounting records panel respectively. On the bottom, below these two panels, is the comments panel that scrolls up when the user clicks on it in order to save screen space when commenting system is not needed. Hence, the document and accounting data management contains all of the functionality needed to manage the content of a document repository, and the accounting data. This chapter discusses some of it's concepts, and also shows some of the implementation aspects.

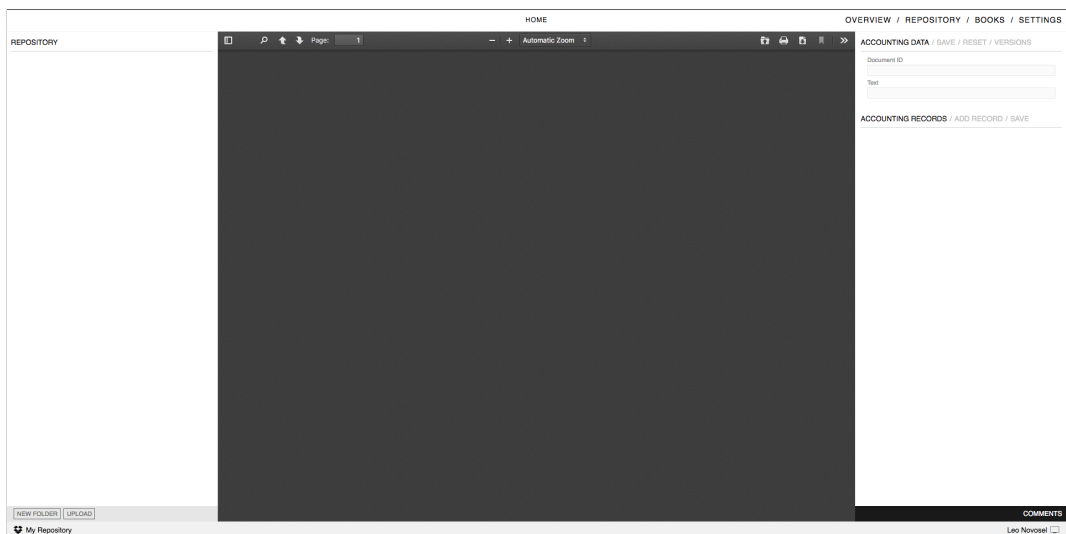


Figure 3.15: Repository page

3.4.1 Fetching the contents of a Document Repository

In order to be able to browse the contents of a document repository (i.e. of the cloud storage), the users have the repository browser panel at their disposal. The browser panel is always showing the content of the current directory as a list of files and directories that are contained within this directory. Additionally, a top entry labeled with two dots is added if the current directory has a parent directory.

The client-side implementation of this functionality has already been introduced in the chapter 2.1. The data that is sent to the client is generated in

3.4 Document and Accounting Data Management

the RPC endpoint by the function `methodListRepositoryDirectory` (Listing 3.35).

Listing 3.35: Listing for the function `methodListRepositoryDirectory`

```
1 protected static function methodListRepositoryDirectory($params, &$response) {
2     try {
3         $cloudService = CloudServiceFactory::createCloudService ( Session::getCloudServiceId () );
4
5         $requestedPath = $params [self::PARAM_PATH];
6         if (Config::get ( 'ROOT_PATH' ) == $requestedPath) {
7             if ($cloudService->getRootPath () != $requestedPath) {
8                 $requestedPath = $cloudService->getRootPath ();
9             }
10        } else {
11            $requestedPath = base64_decode ( $requestedPath );
12        }
13
14        Session::setCloudRepositoryCurrentPath ( $requestedPath );
15        $items = $cloudService->listDirectory ( $requestedPath );
16
17        $dirListing = array ();
18        foreach ( $items as $item ) {
19            $dirListing [] = array (
20                'name' => $item->getName (),
21                'path' => $item->getPath (),
22                'size' => $item->getSize (),
23                'modified' => $item->getModified (),
24                'is_dir' => $item->isDir ()
25            );
26        }
27
28        self::addResponseMember ( $response, self::PARAM_DIR_LISTING, json_encode ( $dirListing ) );
29        self::addResponseMember ( $response, self::PARAM_BREADCRUMBS, View::generateBreadcrumbs (
30            $requestedPath ) );
31        self::addResponseMember ( $response, self::PARAM_PATH, base64_encode ( $requestedPath ) );
32        self::responseStatus ( $response, self::RC_SUCCESS, 'DIRECTORY_LISTED', TRUE );
33        return true;
34    } catch ( AccessTokenException $exception ) {
35        Log::logException ( $exception );
36        self::responseError ( $response, self::RC_INVALID_ACCESS_TOKEN, 'Invalid access token' );
37    } catch ( NetworkException $exception ) {
38        Log::logException ( $exception );
39        self::responseError ( $response, self::RC_FAILURE, 'NETWORK_ERROR', TRUE );
40    } catch ( Exception $exception ) {
41        Log::logException ( $exception );
42        self::responseError ( $response, self::RC_FAILURE, 'LIST_DIRECTORY_FAILURE', TRUE );
43    }
44    return false;
45 }
```

Being that the directory listing functionality must be implemented for each of the cloud storage services individually due to the fact that they all provide their own interface, the function `methodListRepositoryDirectory` must first create an instance of the appropriate class that implements the `CloudServiceInterface` interface by using the class `CloudServiceFactory`, and then call it's `listDirectory` function.

The listing 3.36 shows the implementation of the `listDirectory` function for the Google Drive service (class `GoogleDriveCloudService`), and how the

3 Implementation

cloud storage content is internally represented by using the `CloudServiceFSItem` class.

Listing 3.36: Listing for the function `listDirectory`

```
1 public function listDirectory($path) {
2     try {
3         $contents = array ();
4         $client = $this->getClient ();
5         if (false === $client) {
6             return $contents;
7         }
8
9         $googleDriveClient = new Google_Service_Drive ( $client );
10
11         if ($path == $this->getRootPath () && null == Session::getCloudServiceParameterValue ( '
12             GOOGLE_DRIVE_ROOT_ID' )) {
13             Session::setCloudServiceParameterValue ( 'GOOGLE_DRIVE_ROOT_ID', $path );
14         }
15
16         if ($path != $this->getRootPath ()) {
17             $params = array (
18                 'fields' => 'parents'
19             );
20             $file = $googleDriveClient->files->get ( $path, $params );
21
22             if ($this->getRootPath () == Session::getCloudServiceParameterValue ( '
23                 GOOGLE_DRIVE_ROOT_ID' )) {
24                 Session::setCloudServiceParameterValue ( 'GOOGLE_DRIVE_ROOT_ID', $file->getParents ()
25                 [0] );
26             } else {
27                 if ($file->getParents () [0] == Session::getCloudServiceParameterValue ( '
28                     GOOGLE_DRIVE_ROOT_ID' )) {
29                     $contents [] = new CloudServiceFSItem ( CloudServiceFSItem::PARENT_DIRECTORY,
30                     base64_encode ( 'root' ), true, '-', '-' );
31                 } else {
32                     $contents [] = new CloudServiceFSItem ( CloudServiceFSItem::PARENT_DIRECTORY,
33                     base64_encode ( $file->getParents () [0] ), true, '-', '-' );
34                 }
35             }
36         }
37
38         $params = array (
39             'q' => '\'. $path . \' in parents'
40         );
41         $results = $googleDriveClient->files->listFiles ( $params );
42
43         if (count ( $results->getFiles () ) == 0) {
44             return $contents;
45         } else {
46             foreach ( $results->getFiles () as $item ) {
47                 $name = '';
48                 $filePath = '';
49                 $isDir = false;
50                 $size = '';
51                 $modified = '';
52
53                 $name = htmlspecialchars ( $item->getName () );
54                 $filePath = base64_encode ( $item->getId () );
55                 if ('application/vnd.google-apps.folder' == $item->getMimeType ()) {
56                     $isDir = true;
57                 }
58
59                 $contents [] = new CloudServiceFSItem ( $name, $filePath, $isDir, $size, $modified );
60             }
61         }
62
63         return $contents;
64     } catch ( Exception $exception ) {
65         throw new Exception ( $exception->getMessage () );
66     }
67 }
```

3.4.2 Directory Creation

New directories (i.e. folders) can be created by using the repository toolbar which is located at the bottom of the repository browser panel. The toolbar contains only two buttons, the "NEW FOLDER" button and the "UPLOAD" button. In order to create a new directory within the current one, the user only needs to provide the name for the new directory after clicking on the "NEW FOLDER" button, and then confirm by clicking the "OK" button (Figure 3.16). The newly created directory will be immediately visible in the repository browser panel, because the application automatically triggers a repository listing if the directory creation was successful.

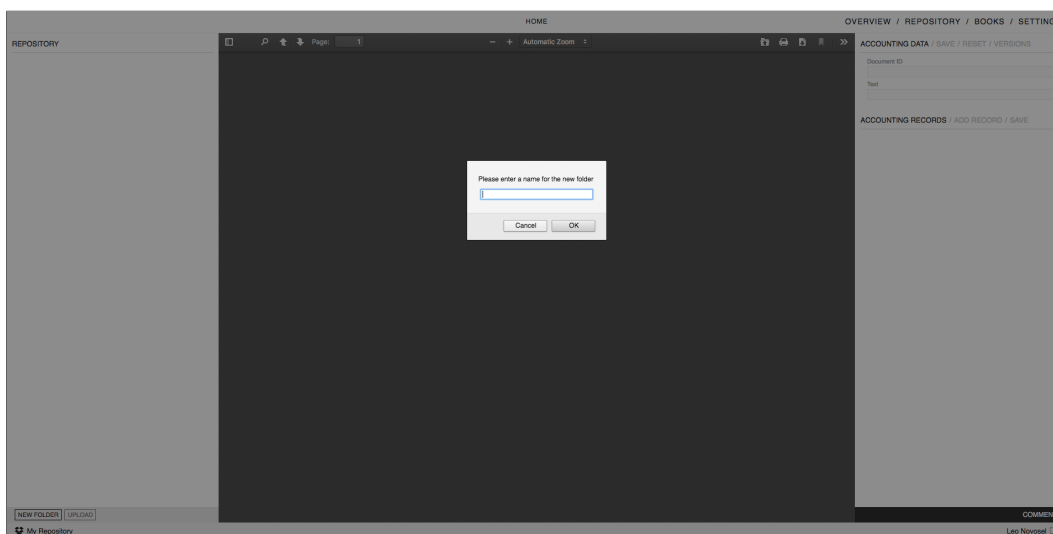


Figure 3.16: Directory creation dialog

This functionality is also implemented by creating an RPC request which is handled by the RPC endpoint class `RPCEndpointInterface`. In this case the RPC endpoint will create an instance of the appropriate class that

3 Implementation

implements the `CloudServiceInterface` interface, and then call the function `createDirectory`. The listing 3.37 shows the implementation of the `createDirectory` function from the class `GoogleDriveCloudService`.

Listing 3.37: Listing for the function `createDirectory`

```
1 public function createDirectory($path, $directory) {
2     try {
3         $client = $this->getClient ();
4         if (false === $client) {
5             return null;
6         }
7
8         $googleDriveClient = new Google_Service_Drive ( $client );
9
10        $fileMetadata = new Google_Service_Drive_DriveFile ( array (
11            'name' => $directory,
12            'parents' => array (
13                $path
14            ),
15            'mimeType' => 'application/vnd.google-apps.folder'
16        ) );
17
18        $file = $googleDriveClient->files->create ( $fileMetadata, array (
19            'fields' => 'id'
20        ) );
21        return $file->getId ();
22    } catch ( Exception $exception ) {
23        Log::logException ( $exception );
24        return null;
25    }
26 }
```

3.4.3 Document Upload

Users can upload PDF documents by using the “UPLOAD” button which is located in the repository toolbar at the bottom of the repository browser panel. After clicking the “UPLOAD” button the file upload will appear where the user can select one or more files to upload (Figure 3.17). After selecting the files and clicking the “UPLOAD” button, the files will be uploaded to the current working directory, and if the file upload was successful, the application will automatically refresh the directory content in the repository browser.

3.4 Document and Accounting Data Management

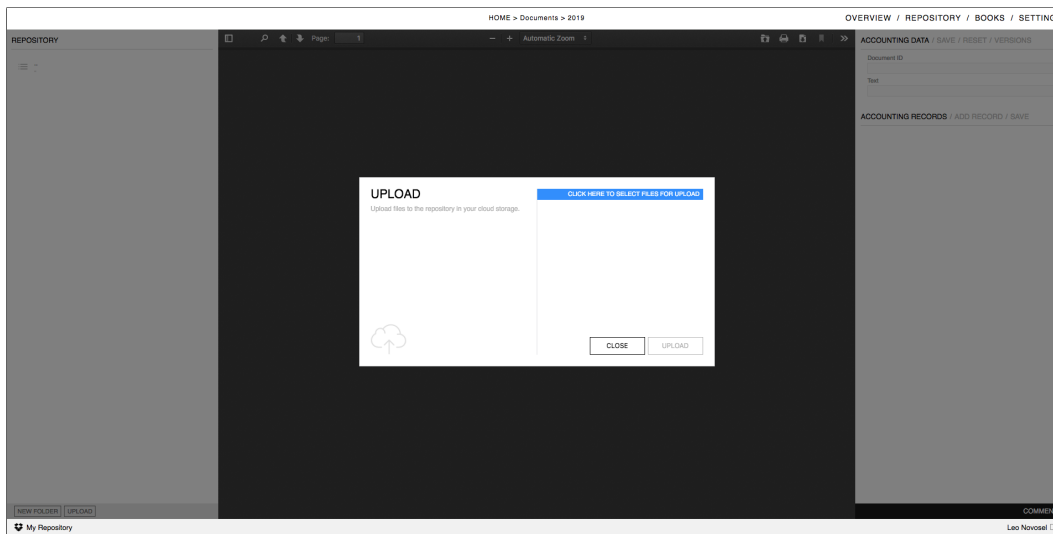


Figure 3.17: File upload dialog

On the server-side, the request is once again handled by the RPC endpoint, this time by the function `methodRepositoryUpload`. This function uses the `$_FILES` array to gather the information regarding the uploaded files, and then calls the upload function from a class that implements the `CloudServiceInterface` interface. The listing 3.38 shows the implementation of the upload function from the class `GoogleDriveCloudService`.

Listing 3.38: Listing for the function upload

```
1 public function upload($path, $filename, &$file, $localPath) {
2     try {
3         $client = $this->getClient ();
4         if (false === $client) {
5             return false;
6         }
7
8         $googleDriveClient = new Google_Service_Drive ( $client );
9
10        $fileMetadata = new Google_Service_Drive_DriveFile ( array (
11            'name' => $filename,
12            'parents' => array (
13                $path
14            )
15        ) );
16        $content = stream_get_contents ( $file );
17        $file = $googleDriveClient->files->create ( $fileMetadata, array (
18            'data' => $content,
19            'mimeType' => 'application/pdf',
20            'uploadType' => 'multipart',
21            'fields' => 'id'
22        ) );
23        return true;
24    }
25}
```

3 Implementation

```
24 } catch ( Exception $exception ) {
25     Log::logException ( $exception );
26     return false;
27 }
28 }
```

3.4.4 Viewing the Document

In order for user to be able to view the document, the document must be downloaded first. The download process is started when the user selects a document in the repository browser. First, the client sends an RPC request which is handled by the RPC endpoint function `methodRepositoryDownload`. The function saves the file path in the session variable and returns the download link to the client. The client can then use the download link to stream the content of the document into a temporary file which is displayed in the document viewer (Listing 3.39).

Listing 3.39: Listing for the class `FileDownloadStream`

```
1 class FileDownloadStream {
2
3     public static function download() {
4         $auth = new Authentication ();
5         if (! $auth->validateSession ()) {
6             exit ();
7         }
8
9         $path = '';
10        if (! isset ( $_GET ['path'] ) && null == Session::getCloudRepositoryPath ()) {
11            exit ();
12        } else if (isset ( $_GET ['path'] )) {
13            $path = base64_decode ( $_GET ['path'] );
14        } else if (null != Session::getCloudRepositoryPath ()) {
15            $path = Session::getCloudRepositoryPath ();
16        }
17
18        $cloudService = CloudServiceFactory::createCloudService ( Session::getCloudServiceId () );
19        $file = tmpfile ();
20        if (! $file) {
21            $maxmemory = 20 * 1024 * 1024; // 20 Mb
22            $file = @fopen ( "php://temp/maxmemory:$maxmemory", 'r+b' );
23
24            if (! $file) {
25                Log::logError ( 'FileDownloadStream: Temporary file creation failed! Temp directory is:
26                ' . sys_get_temp_dir () );
27                exit ();
28            }
29
30            list ( $fileName, $mimeType ) = $cloudService->download ( $path, $file );
31
32            if (null != $mimeType) {
33                header ( 'Pragma: public' );
34                header ( 'Expires: 0' );
35                header ( 'Cache-Control: must-revalidate, post-check=0, pre-check=0' );
36                header ( 'Cache-Control: public' );
37                header ( 'Content-Description: File Transfer' );
```

3.4 Document and Accounting Data Management

```
38     header ( 'Content-Type: ' . $mimeType );
39     header ( 'Content-Disposition:attachment; filename="' . $fileName . '"' );
40     header ( 'Content-Transfer-Encoding: binary' );
41     rewind ( $file );
42     echo stream_get_contents ( $file );
43 }
44 exit ();
45 }
46 }
```

Alongside the file download, which occurs when the user selects a document in the repository browser, the web application also automatically reads and displays the document related accounting data, accounting records and comments. This occurs in the client-side function `LNAPP.xhr.download` (Listing 3.40).

Listing 3.40: Listing for the function `LNAPP.xhr.download`

```
1  LNAPP.xhr.download = function (path, file) {
2    'use strict';
3    if (null === path) {
4      return;
5    }
6
7    if (null === file) {
8      return;
9    }
10
11    var id = LNAPP.xhr.generateRPCRequestId(),
12        data = JSON.stringify({
13      'jsonrpc': LNAPP.JSON_RPC_VERSION,
14      'method': LNAPP.RPC_METHOD_REPOSITORY_DOWNLOAD,
15      'id': id,
16      'params': {'path' : String(path), 'file_name' : String(file)}
17    });
18
19    $.ajax({
20      type: LNAPP.REQUEST_METHOD_POST,
21      url: LNAPP.RPC_SERVER,
22      data: data,
23      success: function (data) {
24        try {
25          var response = JSON.parse(data),
26              message;
27          if (!LNAPP.xhr.validateRPCRequestId(id, response)) {
28            return;
29          }
30
31          message = LNAPP.xhr.checkRPCResponse(response);
32          if (null !== message) {
33            if (LNAPP.RC_INVALID_SESSION === message.code) {
34              window.location.href = LNAPP.PAGE_INDEX;
35              return;
36            }
37
38            if (LNAPP.RC_INVALID_ACCESS_TOKEN === message.code) {
39              window.location.href = LNAPP.PAGE_OVERVIEW + '?token=false';
40              return;
41            }
42
43            LNAPP.gui.actions.displayMessage(message.text);
44            return;
45          }
46        }
47      }
48    });
49  }
```

3 Implementation

```
46
47     if (response.hasOwnProperty('result')) {
48         if (response.result.hasOwnProperty('message') && 'AI1012' === response.result.code) {
49             // Display the document
50             document.getElementById('pdf-viewer').src = 'tools/pdfjs-1.4.20-dist/web/viewer.html
?file=' + response.result.download_url;
51
52             // Set document ID
53             $('#document_id').val(response.result.document_id);
54
55             // Enable toolbar buttons
56             $('.toolbar-button').css('pointer-events', 'auto');
57
58             // Display the document data
59             LNAPP.xhr.getAccountingData(response.result.document_id);
60             // Display accounting records
61             LNAPP.gui.actions.clearAccountingRecords();
62             LNAPP.xhr.getAccountingRecords(response.result.document_id);
63             // Display Comments
64             LNAPP.gui.actions.clearComments();
65             LNAPP.xhr.getComments(response.result.document_id);
66         } else {
67             LNAPP.gui.actions.displayMessage(response.result.message);
68         }
69     }
70     } catch (exception) {
71         LNAPP.gui.utils.logError('app_ui.js [LNAPP.xhr.download]: ' + exception.message);
72     }
73 },
74 beforeSend: function () {
75     LNAPP.gui.overlay.displayOverlay(LNAPP.ELEM_APP_CONTENT_OVERLAY);
76     LNAPP.gui.actions.displayBusy(LNAPP.ELEM_APP_BUSY);
77 },
78 complete: function () {
79     LNAPP.gui.overlay.hideOverlay(LNAPP.ELEM_APP_CONTENT_OVERLAY);
80     LNAPP.gui.actions.hideBusy(LNAPP.ELEM_APP_BUSY);
81 },
82 error: function () {
83     LNAPP.gui.actions.handleXHRError();
84 }
85 });
86 };
```

3.4.5 Document and Directory Deletion

Documents or directories can be deleted by using the delete icon that appears when the user hovers the mouse cursor over the icon displayed on the left side of the file or directory items in the the document viewer (Figure 3.18).

3.4 Document and Accounting Data Management

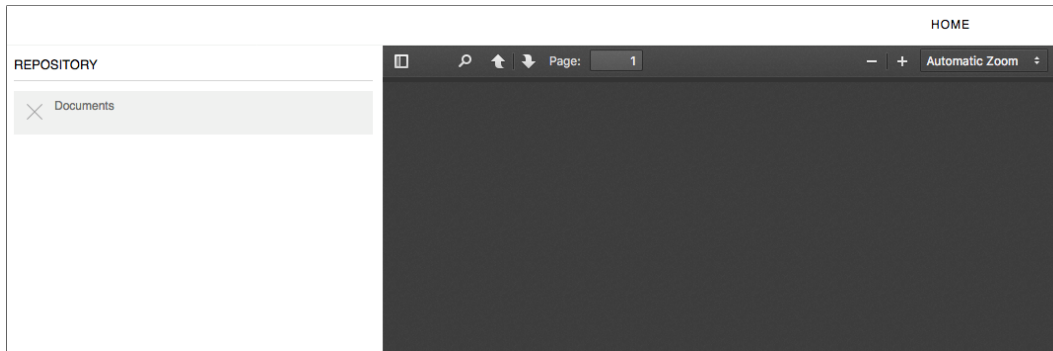


Figure 3.18: Directory deletion

Just as the upload, and the directory creation, the deletion functionality is also implemented in the RPC endpoint. When a directory or a file are to be deleted, the RPC endpoint creates an instance of the appropriate class that implements the `CloudServiceInterface` interface, and then calls the function `delete`. The listing 3.41 shows the implementation of the `delete` function from the class `GoogleDriveCloudService`.

Listing 3.41: Listing for the function `delete`

```
1 public function delete($path) {
2     try {
3         $client = $this->getClient ();
4         if (false == $client) {
5             return false;
6         }
7
8         $googleDriveClient = new Google_Service_Drive ( $client );
9         $googleDriveClient->files->delete ( $path );
10        return true;
11    } catch ( Exception $exception ) {
12        Log::logException ( $exception );
13        return false;
14    }
15 }
```

3.4.6 Editing the Accounting Data

When a document is selected in the the repository browser, the application automatically reads the accounting data and displays it in the accounting data panel. The displayed accounting data fields and the order in which they are displayed will correspond to the default accounting data layout of the current repository (Figure 3.20).

3 Implementation

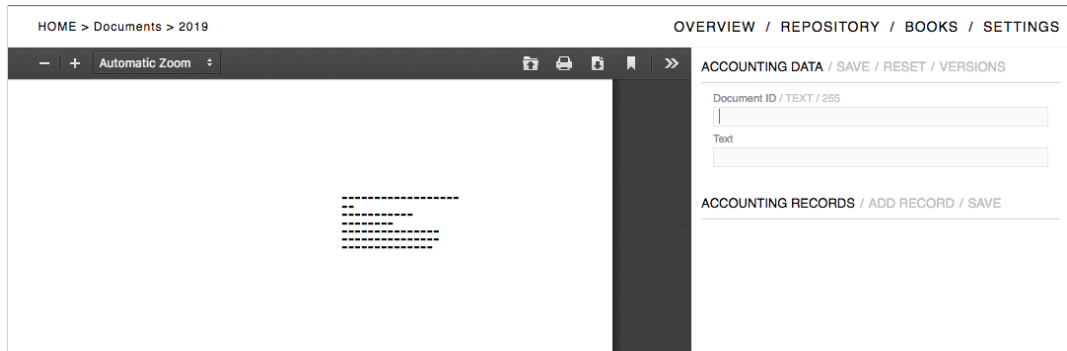


Figure 3.19: Accounting data panel

The user can edit the data by entering the values in the available data fields, and save the data by clicking the “SAVE” menu item on the top of the accounting data panel. When the accounting data is saved, the application automatically also saves the current version of the accounting data for the revision purposes. This functionality is implemented in the `methodSetAccountingData` function of the RPC endpoint (Listing 3.42).

Listing 3.42: Listing for the function `methodSetAccountingData`

```
1 protected static function methodSetAccountingData($params, &$response) {
2   try {
3     if (! Permissions::userHasRepositoryPrivilege ( 'RP_EDIT_ACCOUNTING_DATA' )) {
4       self::responseError ( $response, self::RC_FAILURE, 'INSUFFICIENT_PRIVILEGES', TRUE );
5       return false;
6     }
7
8     $accountingDataModel = new AccountingData ( DatabaseAdapter::getPDODatabaseHandle () );
9     $result = $accountingDataModel->setAccountingData ( Session::getCloudRepositoryId (),
10      $params [self::PARAM_DOCUMENT_ID], $params );
11     if (false === $result) {
12       self::responseError ( $response, self::RC_FAILURE, 'SET_ACCOUNTING_DATA_ERROR', TRUE );
13       return false;
14     }
15
16     $accountingData = $accountingDataModel->getAccountingData ( Session::getCloudRepositoryId
17      (), $params [self::PARAM_DOCUMENT_ID] );
18     if (null === $result) {
19       self::responseError ( $response, self::RC_FAILURE, 'SET_ACCOUNTING_DATA_ERROR', TRUE );
20       return false;
21     }
22
23     $accountingDataRevisionsModel = new AccountingDataRevisions ( DatabaseAdapter::
24      getPDODatabaseHandle () );
25     $result = $accountingDataRevisionsModel->createAccountingDataRevision ( Session::
26      getCloudRepositoryId (), $accountingData ['accounting_data_id'], $params );
27     if (false === $result) {
28       self::responseError ( $response, self::RC_FAILURE, 'SET_ACCOUNTING_DATA_ERROR', TRUE );
29       return false;
30     }
31
32     self::responseStatus ( $response, self::RC_SUCCESS, 'ACCOUNTING_DATA_SET', TRUE );
33   }
34 }
```

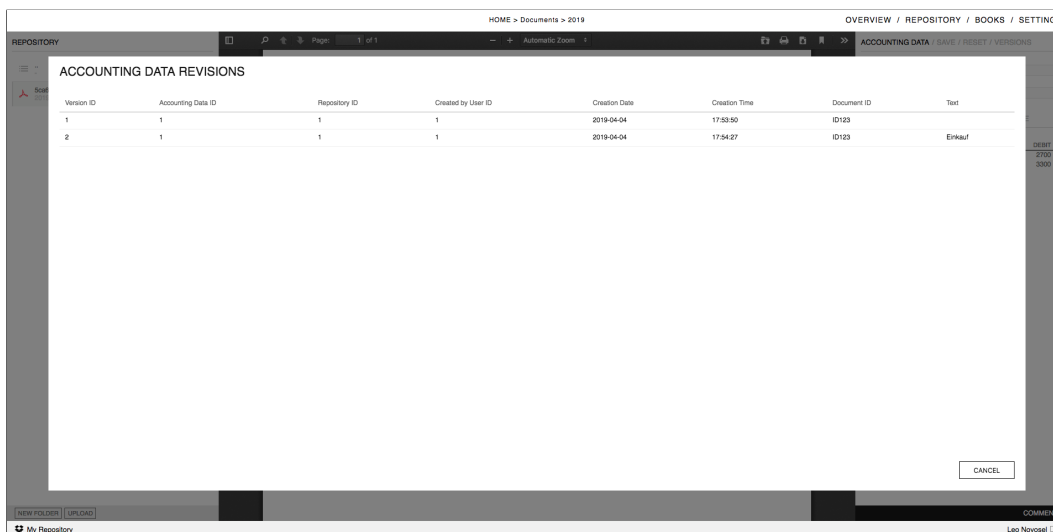
3.4 Document and Accounting Data Management

```
29     return true;
30 } catch ( Exception $exception ) {
31     Log::logException ( $exception );
32     self::responseError ( $response, self::RC_FAILURE, 'SET_ACCOUNTING_DATA_FAILURE', TRUE );
33 }
34 return false;
35 }
```

3.4.7 Versioning of the Accounting Data

Each time when the accounting data is modified, the current version of the data is saved for the revision purposes. This occurs automatically in the background and requires no action from the user.

The versions of the accounting data are visible in the accounting data revisions dialog that is available through the "VERSIONS" menu item on the top of the accounting data panel (Figure 3.20). The information visible in the accounting data panel includes the version id, accounting data id, repository id, the id of the user that has edited the accounting data, time and date when the data was edited, and the accounting data itself.



Version ID	Accounting Data ID	Repository ID	Created by User ID	Creation Date	Creation Time	Document ID	Text
1	1	1	1	2019-04-04	17:53:50	ID123	
2	1	1	1	2019-04-04	17:54:27	ID123	Erkauf

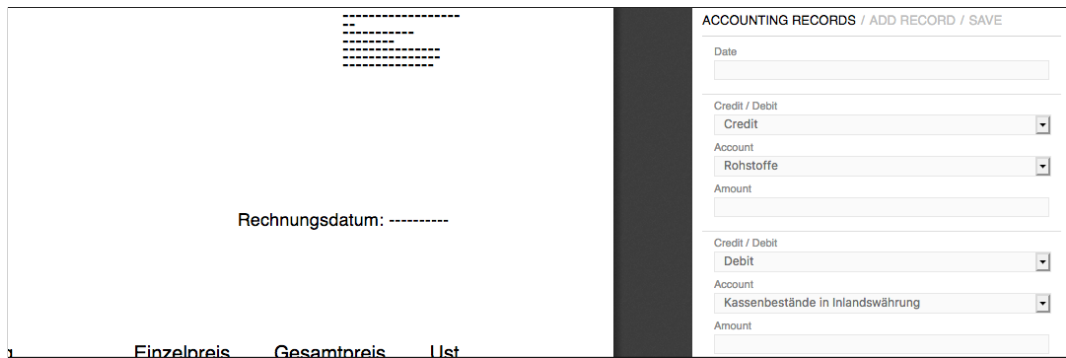
Figure 3.20: Accounting data revisions dialog

3 Implementation

As discussed in the chapter 3.4.6, the versioning of the accounting data is an integral part of the accounting data editing process, and is not separately implemented.

3.4.8 Creating the Accounting Records

The accounting records can be created in the accounting data records panel on the right hand side of the application window by clicking on the "ADD RECORD" menu item on the top of the panel (Figure 3.21). When the menu item is clicked the application will display the input field for the record date, and three fields for the record data: credit or debit selector, the account field and the amount field. In order to create two accounting records with the same record date, a credit and a debit one, the user must click the "ADD RECORD" menu item twice. This will provide the user with the possibility to enter two records before saving the data, and if the business transaction affects even more accounts, the user can create additional records simply by clicking on the "ADD RECORD" menu item. When all the accounting records are recorded, the user can save them by clicking on the "SAVE" menu item.



The screenshot shows a software interface for creating accounting records. On the left, there is a table with columns labeled 'Einzelpreis', 'Gesamtpreis', and 'List'. Above the table, there is a label 'Rechnungsdatum: -----' and a series of dashed lines representing a list of records. On the right, there is a panel titled 'ACCOUNTING RECORDS / ADD RECORD / SAVE'. This panel contains two identical forms for entering record data. Each form has a 'Date' input field, a 'Credit / Debit' dropdown menu (with 'Credit' selected in the first and 'Debit' in the second), an 'Account' dropdown menu (with 'Rohstoffe' in the first and 'Kassenbestände in Inlandswährung' in the second), and an 'Amount' input field. The top of the panel also has a breadcrumb trail: 'ACCOUNTING RECORDS / ADD RECORD / SAVE'.

Figure 3.21: Accounting records panel

When the accounting records are saved on the client-side, the function `LNAPP.xhr.createAccountingRecord` (Listing 3.43) is called. This function collects all the accounting records data from the input fields and generates an RPC request.

3.4 Document and Accounting Data Management

Listing 3.43: Listing for the function LNAPP.xhr.createAccountingRecord

```
1 LNAPP.xhr.createAccountingRecord = function () {
2   'use strict';
3   var id = LNAPP.xhr.generateRPCRequestId(),
4       params = {},
5       records = [],
6       record = {},
7       index,
8       data = {
9         'jsonrpc': LNAPP.JSON_RPC_VERSION,
10        'method': LNAPP.RPC_METHOD_REPOSITORY_CREATE_ACCOUNTING_RECORD,
11        'id': id
12      };
13
14   params.document_id = String($('#document_id').val());
15   params.record_date = String($('#record-date-0').val());
16
17   for (index = 0; index < LNAPP.accountingRecordEntry_; index += 1) {
18     record.credit_debit = $('#record-credit-debit-' + index).val();
19     record.account = $('#record-account-' + index).val();
20     record.amount = $('#record-amount-' + index).val();
21     records.push(record);
22     record = {};
23   }
24
25   params.accounting_records = records;
26   data.params = params;
27   data = JSON.stringify(data);
28
29   $.ajax({
30     type: LNAPP.REQUEST_METHOD_POST,
31     url: LNAPP.RPC_SERVER,
32     data: data,
33     success: function (data) {
34       try {
35         var response = JSON.parse(data),
36             message;
37         if (!LNAPP.xhr.validateRPCRequestId(id, response)) {
38           return;
39         }
40
41         message = LNAPP.xhr.checkRPCResponse(response);
42         if (null !== message) {
43           if (LNAPP.RC_INVALID_SESSION === message.code) {
44             window.location.href = LNAPP.PAGE_INDEX;
45             return;
46           }
47
48           if (LNAPP.RC_INVALID_ACCESS_TOKEN === message.code) {
49             window.location.href = LNAPP.PAGE_OVERVIEW + '?token=false';
50             return;
51           }
52
53           LNAPP.gui.actions.displayMessage(message.text);
54           return;
55         }
56
57         if (response.hasOwnProperty('result')) {
58           if (response.result.hasOwnProperty('message') && 'AI1024' === response.result.code) {
59             LNAPP.gui.actions.displayFeedback('SAVED');
60             LNAPP.gui.actions.clearAccountingRecords();
61             LNAPP.xhr.getAccountingRecords($('#document_id').val());
62           } else {
63             LNAPP.gui.actions.displayMessage(response.result.message);
64           }
65         }
66       } catch (exception) {
67         LNAPP.gui.utils.logError('app_ui.js [LNAPP.xhr.createAccountingRecord]: ' + exception.
68           message);
69       }
70     },
71     error: function () {
72       LNAPP.gui.actions.handleXHRError();
73     }
74   });
75 }
```

3 Implementation

```
72     }  
73   });  
74 };
```

On the server-side, the request is handled by the RPC endpoint's function `methodCreateAccountingRecord` which saves the accounting records in the database (Listing 3.44).

Listing 3.44: Listing for the function `methodCreateAccountingRecord`

```
1  protected static function methodCreateAccountingRecord($params, &$response) {  
2    try {  
3      if (! Permissions::userHasRepositoryPrivilege ( 'RP_EDIT_ACCOUNTING_DATA' )) {  
4        self::responseError ( $response, self::RC_FAILURE, 'INSUFFICIENT_PRIVILEGES', TRUE );  
5        return false;  
6      }  
7  
8      $date = explode ( '.', $params [self::PARAM_RECORD_DATE] );  
9      if (count ( $date ) == 3) {  
10       $recordDate = $date [2] . '-' . $date [1] . '-' . $date [0];  
11     } else {  
12       self::responseError ( $response, self::RC_FAILURE, 'SAVING_RECORDS_ERROR', TRUE );  
13       return false;  
14     }  
15  
16     $databaseHandle = DatabaseAdapter::getPDODatabaseHandle ();  
17     $databaseHandle->beginTransaction ();  
18     $accountingRecords = new AccountingRecords ( $databaseHandle );  
19  
20     foreach ( $params [self::PARAM_ACCOUNTING_RECORDS] as $record ) {  
21       $creditDebit = $record ['credit_debit'];  
22       $account = $record ['account'];  
23       $amount = $record ['amount'];  
24       $recordId = $accountingRecords->createAccountingRecord ( $params [self::PARAM_DOCUMENT_ID],  
25         Session::getCloudRepositoryId (), $recordDate, $creditDebit, $account, $amount );  
26       if (null == $recordId) {  
27         $databaseHandle->rollback ();  
28         self::responseError ( $response, self::RC_FAILURE, 'SAVING_RECORDS_ERROR', TRUE );  
29         return false;  
30       }  
31     }  
32     $databaseHandle->commit ();  
33     self::responseStatus ( $response, self::RC_SUCCESS, 'RECORDS_SAVED', TRUE );  
34     return true;  
35   } catch ( Exception $exception ) {  
36     Log::logException ( $exception );  
37     self::responseError ( $response, self::RC_FAILURE, 'SAVING_RECORDS_FAILURE', TRUE );  
38   }  
39   return false;  
40 }
```

3.4.9 Viewing the Books

As discussed in the chapter 1.4.1, the double-entry bookkeeping requires that each business transaction must be listed twice, once in the journal, which is chronologically sorted, and once in the general ledger, which is

3.4 Document and Accounting Data Management

systematically organized into accounts. Within the web application that is developed within the scope of this thesis, the journal and the general ledger can be found on the books page that is reachable through the entry "BOOKS" in the application's navigation bar (Figure 3.22).

cloud bookkeeping					OVERVIEW / REPOSITORY / BOOKS / SETTINGS	
JOURNAL					GENERAL LEDGER	
GENERAL LEDGER					ID: 123	
1100 Rohstoffe						
Date	Document ID	Text	Credit	Debit		
2019-02-01	ID123	Erkauf	10000			
2500 Forderungen aus der Abgabenverrechnung						
Date	Document ID	Text	Credit	Debit		
2019-02-01	ID123	Erkauf	2000			
2700 Kassenbestände in Inlandswährung						
Date	Document ID	Text	Credit	Debit		
2019-02-01	ID123	Erkauf		5000		
3300 Verbindlichkeiten aus Lieferungen und Leistungen Inland						
Date	Document ID	Text	Credit	Debit		
2019-02-01	ID123	Erkauf		7000		

Figure 3.22: General ledger on the books page

Both journal and the general ledger are generated with PHP within the books page. The Listing 3.45 shows the implementation of the function `generateGeneralLedgerTable` from the class `View`.

Listing 3.45: Listing for the function `generateGeneralLedgerTable`

```
1 public static function generateGeneralLedgerTable($repositoryId) {
2     $html = '';
3     $parameters = new Parameters ( $repositoryId );
4     $accountsTable = $parameters->getTable ( 'ACCOUNT' );
5
6     $accountingRecordsModel = new AccountingRecords ( DatabaseAdapter::getPDODatabaseHandle ( ) );
7     $accountingRecords = null;
8     $account = '';
9     foreach ( $accountsTable ['table'] as $key => $value ) {
10        $accountingRecords = $accountingRecordsModel->getGeneralLedgerAccountingRecords ( $key,
11            Session::getCloudRepositoryId ( ) );
12        $account = $key . ' ' . $value;
13
14        if (null !== $accountingRecords) {
15            $html .= '<div class="data-table-header" style="text-align: left;">' . $account . '</div>';
16            $html .= '<table class="data-table" style="margin-bottom: 20px;" cellpadding="0px" cellspacing="0px">';
17        }
18    }
19    return $html;
20 }
```

3 Implementation

```
16     $html .= '<tbody>';
17     $html .= '<tr class="header">';
18     $html .= '<td>Date</td>';
19     $html .= '<td>Document ID</td>';
20     $html .= '<td>Text</td>';
21     $html .= '<td>Credit</td>';
22     $html .= '<td>Debit</td>';
23     $html .= '</tr>';
24
25     foreach ( $accountingRecords as $accountingRecord ) {
26         $html .= '<tr class="data-row selectable">';
27         $html .= '<td>' . $accountingRecord->date . '</td>';
28         $html .= '<td>' . $accountingRecord->data_1 . '</td>';
29         $html .= '<td>' . $accountingRecord->data_2 . '</td>';
30         if (0 == $accountingRecord->credit_debit) {
31             $html .= '<td>' . $accountingRecord->amount . '</td>';
32         } else {
33             $html .= '<td>&nbsp;</td>';
34         }
35
36         if (1 == $accountingRecord->credit_debit) {
37             $html .= '<td>' . $accountingRecord->amount . '</td>';
38         } else {
39             $html .= '<td>&nbsp;</td>';
40         }
41         $html .= '</tr>';
42     }
43
44     $html .= '</tbody>';
45     $html .= '</table>';
46     $html .= '</br>';
47
48 }
49 }
50
51 return $html;
52 }
```

3.4.10 Using the Comment System

The commenting system allows the user to leave a document related comment that is visible to all other repository users. Comments can be entered in the comments panel which is initially hidden, and must be clicked on in order to scroll up into the view (Figure 3.23). The system is kept very simple, so all of the comments are shown in the order in which they were entered.

3.4 Document and Accounting Data Management

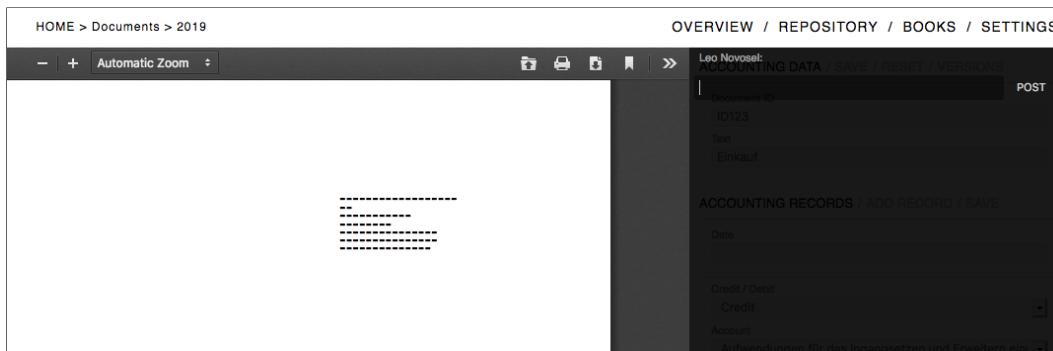


Figure 3.23: Comments panel

When posting a comment, an RPC request is sent to the RPC server and handled in the function methodPostComment (Listing 3.46) from the RPC endpoint class RPCEndpointInterface.

Listing 3.46: Listing for the function methodPostComment

```
1  protected static function methodPostComment($params, &$response) {
2      try {
3
4          $commentsModel = new Comments( DatabaseAdapter::getPDODatabaseHandle () );
5          if (null === $commentsModel->createComment ( Session::getCloudRepositoryId (), $params [self
::PARAM_DOCUMENT_ID], Session::getUserId (), strip_tags ( $params [self::PARAM_COMMENT] ) ))
        {
6              self::responseError ( $response, self::RC_FAILURE, 'CREATING_COMMENT_ERROR', TRUE );
7              return false;
8          }
9
10         $comments = $commentsModel->getComments ( Session::getCloudRepositoryId (), $params [self::
PARAM_DOCUMENT_ID] );
11         $commentsArray = array ();
12         if (null !== $comments) {
13             foreach ( $comments as $comment ) {
14                 $record [self::PARAM_COMMENT_ID] = $comment->comment_id;
15                 $record [self::PARAM_FIRST_NAME] = $comment->first_name;
16                 $record [self::PARAM_LAST_NAME] = $comment->last_name;
17                 $record [self::PARAM_COMMENT] = $comment->comment_text;
18                 $commentsArray [] = $record;
19             }
20         }
21
22         self::addResponseMember ( $response, self::PARAM_COMMENTS, json_encode ( $commentsArray ) );
23         self::responseStatus ( $response, self::RC_SUCCESS, 'COMMENT_CREATED', TRUE );
24         return true;
25     } catch ( Exception $exception ) {
26         Log::logException ( $exception );
27         self::responseError ( $response, self::RC_FAILURE, 'CREATING_COMMENT_FAILURE', TRUE );
28     }
29     return false;
30 }
```

3 Implementation

3.5 Settings

This section gives a short overview of the user settings available in the web application, the user, repository and authorization management interfaces, as well as the application log.

3.5.1 General Settings

The user can navigate to the settings page by using the "SETTINGS" item from the navigation bar in the top right corner of the application screen. The first settings available to the user are the general settings, which contain the user specific application settings (Figure 3.24). The number of settings can be expanded, but by default, the only user setting available is the repository layout setting that allows the user to select between two different panel layouts on the repository page. The horizontal layout, which is the default one, will position the repository browser on the left hand side of the application window, and the accounting data and accounting records panels on the right hand side, while keeping the document viewer in the middle. The vertical layout, on the other hand, will position all of the panels below the document viewer allowing the document viewer to stretch horizontally across the screen. This option can be beneficial when working with wide format documents.

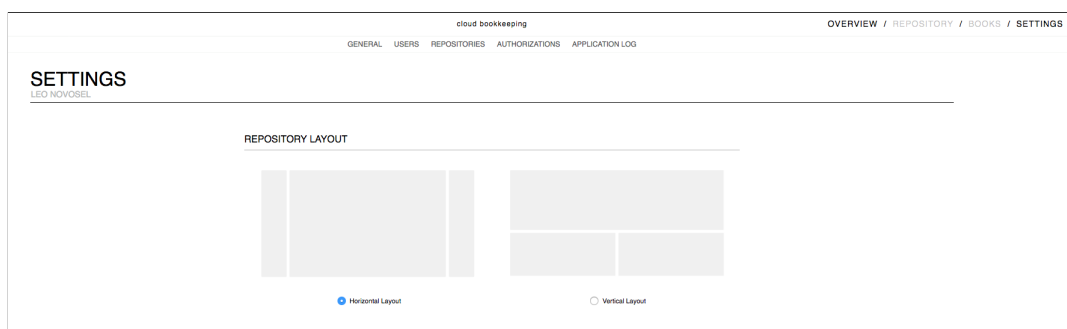


Figure 3.24: User settings

3.5.2 User, Repository and Authorization Overview

For the application administrators, the user interface offers three overview panels which are kept very simple. The user overview panel contains the list of all application users (Figure 3.25). When a user is selected, the application automatically displays the user log and the user requests log. This information can be helpful when, for instance, analyzing why a specific user has no application access.

The screenshot shows the 'USERS' overview panel. At the top, there are navigation tabs: GENERAL, USERS, REPOSITORIES, AUTHORIZATIONS, and APPLICATION LOG. The 'USERS' tab is selected. Below the tabs, the title 'USERS' is displayed, followed by a sub-header '1 USERS'. A table lists the user details:

User ID	Username	First Name	Last Name	Account Status	Registered
1	Leo.Novosel@example.com	Leo	Novosel	Active	22.03.2019 19:27:04

Below the user table, there are two sections: 'USER LOG' and 'USER REQUESTS'. Each section has a table with columns for 'User Log ID', 'Date and Time', 'Success', 'Request ID', 'Request Type', 'Requested on', and 'Carried out on'.

USER LOG

User Log ID	Date and Time	Success
1	22.03.2019 19:27:08	Success
2	27.03.2019 20:04:50	Success
3	29.03.2019 12:21:13	Success
4	31.03.2019 08:20:44	Success
5	31.03.2019 09:21:43	Success
6	01.04.2019 18:06:20	Success
7	04.04.2019 18:04:51	Success
8	05.04.2019 11:18:52	Success
9	05.04.2019 16:53:10	Success
10	06.04.2019 09:27:57	Success

USER REQUESTS

Request ID	Request Type	Requested on	Carried out on
1	Account Activation	22.03.2019 19:27:04	22.03.2019 19:27:08

At the bottom left, there is a status indicator 'Offline' and at the bottom right, the user's name 'Leo Novosel' is displayed.

Figure 3.25: User overview

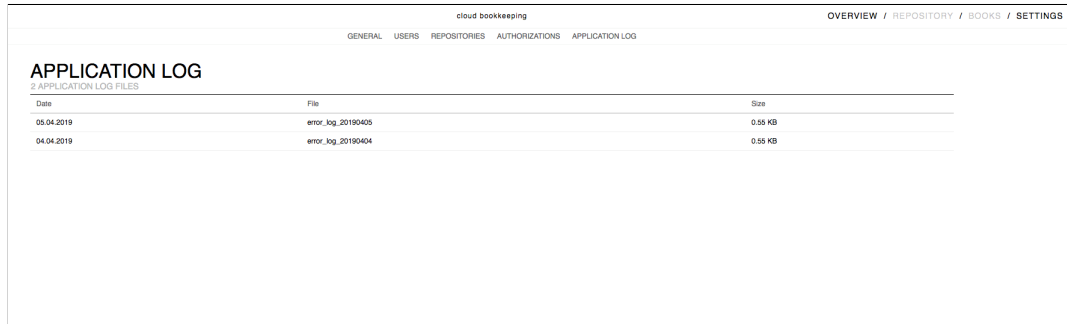
Additionally, the application offers the repository and authorization overview panels. The repository overview contains the list of all repositories, while the authorization overview contains the list of all the repository user authorizations.

3.5.3 Application Log

The application log panel contains the list of all the application log files (Figure 3.26). An individual application log file can be viewed by selecting

3 Implementation

an appropriate entry from the list. The log files contain both client-side and server-side errors.



Date	File	Size
05.04.2019	error_log_20190405	0.55 KB
04.04.2019	error_log_20190404	0.55 KB

Figure 3.26: Application log

4 Conclusion

In this thesis, the possible usage of the cloud storage services and the SaaS delivery model for the bookkeeping or accounting applications was researched. For this purpose, the business requirements were analyzed and the necessary functionality implemented in a web application by using the cloud service APIs and JSON-RPC web services. The process of integrating a cloud storage service in the web application has proven to be very straightforward, due to the fact that the APIs of the cloud storage services considered in this thesis are very well documented, and that all of the service providers offer API libraries for a variety of programming languages. The goal of this thesis was not to implement a complete bookkeeping application, but the mapping of the basic business requirements to the web service functionality was very intuitive. Additionally, the SaaS has proven to be an ideal delivery model for this kind of application, because it offers the possibility of implementing a multi-tenant architecture and supports the collaborative functionality that was needed.

Bibliography

- Aaron Parecki (2019). *OAuth 2.0*. URL: <https://oauth.net/2/> (visited on 03/14/2019) (cit. on p. 31).
- Achour, Mehdi, Friedhelm Betz, Antony Dovgal, et al. (2017). *PHP Manual*. URL: <http://php.net/manual/en/index.php> (visited on 01/17/2017) (cit. on pp. 7, 51).
- Balasubramanian, V and Alf Bashian (1998). "Document management and Web technologies: Alice marries the Mad Hatter." In: *Communications of the ACM* 41.7, pp. 107–115 (cit. on p. 3).
- Barton, Thomas (2014). *E-Business mit Cloud Computing [in German]*. Springer Fachmedien Wiesbaden (cit. on pp. 8, 9).
- Bauer, Ulrich (2017). *Externe Unternehmensrechnung/Buchhaltung und Bilanzierung [in German]*. Skriptenprojekt Hochschülerschaft an der TU Graz GmbH (cit. on pp. 13, 17, 19, 20).
- Bos, Bert (2016). *Cascading Style Sheets Level 2 Revision 2 (CSS 2.2) Specification*. URL: <https://www.w3.org/TR/CSS22/css2.pdf> (visited on 01/21/2017) (cit. on p. 5).
- Bragg, Steven M. (2011). *Bookkeeping Essentials*. John Wiley & Sons (cit. on p. 1).
- Brooks, Chad (2004). *Document Management Systems: A Buyer's Guide*. URL: <http://www.businessnewsdaily.com/8026-choosing-a-document-management-system.html> (visited on 03/27/2017) (cit. on p. 3).
- Bundesministerium für Finanzen (2017a). *Elektronische Rechnung - Überblick über die Änderungen durch das Abgabenänderungsgesetz 2012 [in German]*. URL: https://www.bmf.gv.at/steuern/selbststaendige-unternehmer/umsatzsteuer/elektronische-rechnung_abgaeg2012.html (visited on 02/27/2017) (cit. on p. 2).
- Bundesministerium für Finanzen (2017b). *e-Rechnung Rechtliche Grundlagen [in German]*. URL: https://www.erechnung.gv.at/erb?p=info_jur (visited on 02/27/2017) (cit. on p. 2).

Bibliography

- Chao, Kuo-Ming (2016). "E-services in e-business engineering." In: *Electronic Commerce Research and Applications* 16, pp. 77–81. URL: <http://www.sciencedirect.com/science/article/pii/S1567422315000769> (cit. on p. 1).
- Chong, Frederick, Gianpaolo Carraro, and Roger Wolter (2017). *Multi-Tenant Data Architecture*. URL: <https://msdn.microsoft.com/en-us/library/aa479086.aspx> (visited on 04/03/2017) (cit. on p. 37).
- Dropbox Inc. (2017a). *Dropbox*. URL: <https://www.dropbox.com/> (visited on 05/01/2017) (cit. on p. 10).
- Dropbox Inc. (2017b). *Dropbox Developers*. URL: <https://www.dropbox.com/developers> (visited on 05/01/2017) (cit. on p. 10).
- Faulkner, Steve et al. (2016). *HTML 5.1 W3C Recommendation*. URL: <https://www.w3.org/TR/html/> (visited on 01/22/2017) (cit. on p. 5).
- Fielding, Roy and Julian Reschke (2014). *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. URL: <https://tools.ietf.org/html/rfc7230> (visited on 03/18/2017) (cit. on p. 4).
- Flanagan, David (2011). *JavaScript: The Definitive Guide*. 6th Edition. O'Reilly Media, Inc. (cit. on p. 6).
- Ginsburg, Mark (2000). "Intranet document management systems as knowledge ecologies." In: *Proceedings of the 33rd Hawaii International Conference on System Sciences* (cit. on p. 3).
- Goodman, Danny (2006). *Dynamic HTML: The Definitive Reference*. 3rd Edition. O'Reilly Media, Inc. (cit. on p. 6).
- Google (2017a). *Google API Client Libraries*. URL: <https://developers.google.com/api-client-library/> (visited on 05/02/2017) (cit. on p. 12).
- Google (2017b). *Google Drive*. URL: <https://www.google.com/drive/> (visited on 05/02/2017) (cit. on p. 11).
- Jacobs, Ian and Norman Walsh (2004). *Architecture of the World Wide Web, Volume One*. URL: <https://www.w3.org/TR/webarch> (visited on 03/20/2017) (cit. on p. 4).
- JSON-RPC Working Group (2013). *JSON-RPC 2.0 Specification*. URL: <https://www.jsonrpc.org/specification> (visited on 03/03/2019) (cit. on p. 29).
- Microsoft (2017a). *Microsoft OneDrive*. URL: <https://onedrive.live.com/about/en-us/> (visited on 05/03/2017) (cit. on p. 12).

Bibliography

- Microsoft (2017b). *OneDrive API*. URL: <https://dev.onedrive.com/> (visited on 05/03/2017) (cit. on p. 13).
- Oracle Corporation and/or its affiliates (2017a). *MySQL 5.7 Reference Manual*. URL: <https://dev.mysql.com/doc/refman/5.7/en/introduction.html> (visited on 04/08/2017) (cit. on p. 39).
- Oracle Corporation and/or its affiliates (2017b). *MySQL 5.7 Reference Manual*. URL: <https://dev.mysql.com/doc/refman/5.7/en/implicit-commit.html> (visited on 04/20/2017) (cit. on p. 50).
- Oracle Corporation and/or its affiliates (2017c). *MySQL Glossary*. URL: https://dev.mysql.com/doc/refman/5.7/en/glossary.html#glos_schema (visited on 04/03/2017) (cit. on p. 37).
- Powers, Shelley (2007). *Adding Ajax*. O'Reilly Media, Inc. (cit. on p. 8).
- Quinn, Martin (2010). *Brilliant Book-keeping*. Pearson Business (cit. on p. 1).
- Tatroe, Kevin, Peter MacIntyre, and Rasmus Lerdorf (2013). *Programming PHP*. 3rd Edition. O'Reilly Media, Inc. (cit. on p. 7).
- W3C (2017a). *Help and FAQ*. URL: <https://www.w3.org/Help/#webinternet> (visited on 03/20/2017) (cit. on p. 4).
- W3C (2017b). *HTML & CSS*. URL: <https://www.w3.org/standards/webdesign/htmlcss> (visited on 01/20/2017) (cit. on pp. 5, 6).
- W3C (2017c). *JavaScript Web APIs*. URL: <https://www.w3.org/standards/webdesign/script> (visited on 01/18/2017) (cit. on p. 6).