Utrecht University

Enabling multitenant software solutions for online business software.

Author:

Lucian Cancescu

Student Number: 3609707

Master of Business Informatics

Utrecht University

l.cancescu@uu.nl

Supervisors:

Dr. Slinger Jansen	Jaap Kabbedijk, Msc
Department of Information and Computing Sciences	Department of Information and Computing Sciences
Utrecht University	Utrecht University
slinger.jansen@uu.nl	j.kabbedijk@uu.nl

Table of contents

Table of figures

Table of tables

Chapter 1 - Introduction

Abstract

Risk management is crucial for business software developers. In order to minimize the risk of choosing the wrong multitenancy implementation pattern for a software project, four case studies were conducted that resulted in four architectural patterns. Two of the patterns were derived from interviews while the other two from literature research. For each pattern, their advantages, their liabilities and possible solutions to mitigate them were determined.

To help business software developers to choose the most suitable multitenancy implementation patterns, this work provides an overview of the four implementation patterns. The overview consists of a number of multitenancy specific problems and offers solutions on how the patterns can solve these problems, and it also consists of a comparison of the four implementation patterns by showing their consequences and linking them to possible mitigation solutions.

1.1 The change in business software

The trend for business software has changed from running as multiple instances on clients' servers to running on one instance and serving multiple clients (Aulbach et al., 2008). For business software developers it is more efficient to deploy their applications in the cloud and thus enable it for any number of clients. However, by doing so, the complexity increases because all clients are actually accessing the same instance of the software. Since business software deals with important data, often confidential, the question whether this approach is secure is raised (Wainewright, 2012). Not only business software developers have to solve the complexity of serving their application to multiple tenants from a single instance, but they also need to convince their clients that their solution is secure, reliable and their data is stored in a safe place.

1.2 Problem statement

The aim of the research is to reduce the risk of choosing the wrong multitenancy implementation pattern for a project. There is no research that gives an overview of current multitenancy implementation patterns and shows which of them are most suitable for certain situations.

This will help business software developers to deploy their software in the right way, to carefully plan how their tenants will communicate, to ensure that clients' data and privacy are protected and to make sure the right network architecture is used.

The aim of the research is to reduce the risk of choosing the wrong multitenancy implementation pattern for a project.

To illustrate the problem statement, Figure 1 shows the Use Case Diagram for online business software. The figure shows where the problem appears, namely when the software developer has to choose the right multitenancy implementation pattern.

Actor	Main Activities
Software Developer	Choose the right MT implementation pattern. Build and deploy in a SaaS infrastructure the business software.
Tenant	Use the online business software.

Table 1. The actors and the main activities for online business software

Figure 1 below illustrates the actors and the main activities (table 1) for online business software.

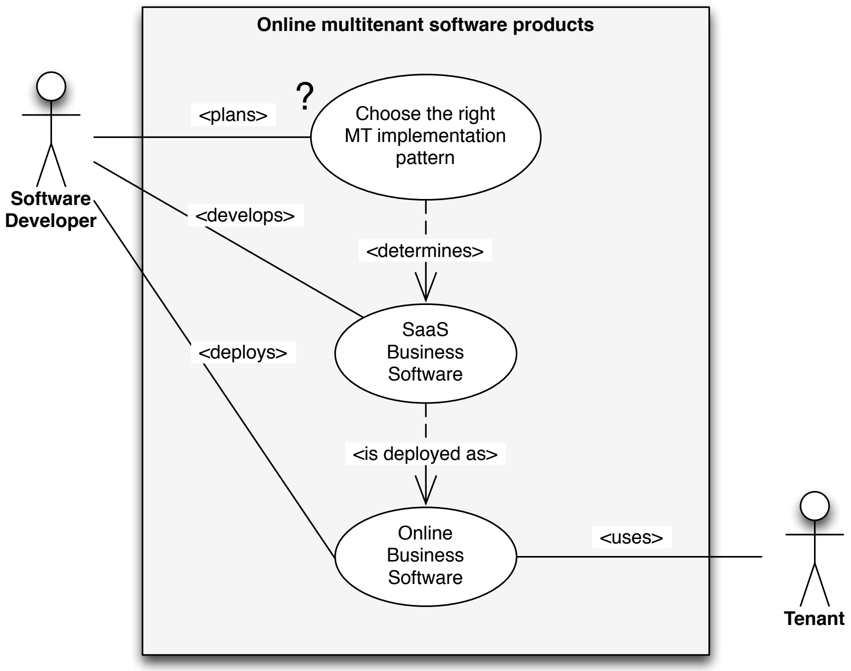


Figure 1. Use Case Diagram of the Online Business Software

The problem statement is illustrated in two examples in the following sub-chapter. The two examples show why it is unclear for a software developer which multitenancy business software pattern to use.

1.2.1 Example of problem statement

In our research we aim to help multitenant software product developers to reduce the risk of choosing the wrong implementation pattern. In order to illustrate this problem, we will present one simple example that shows a scenario where the software developer needs to make a decision on what pattern to choose (figure 2).

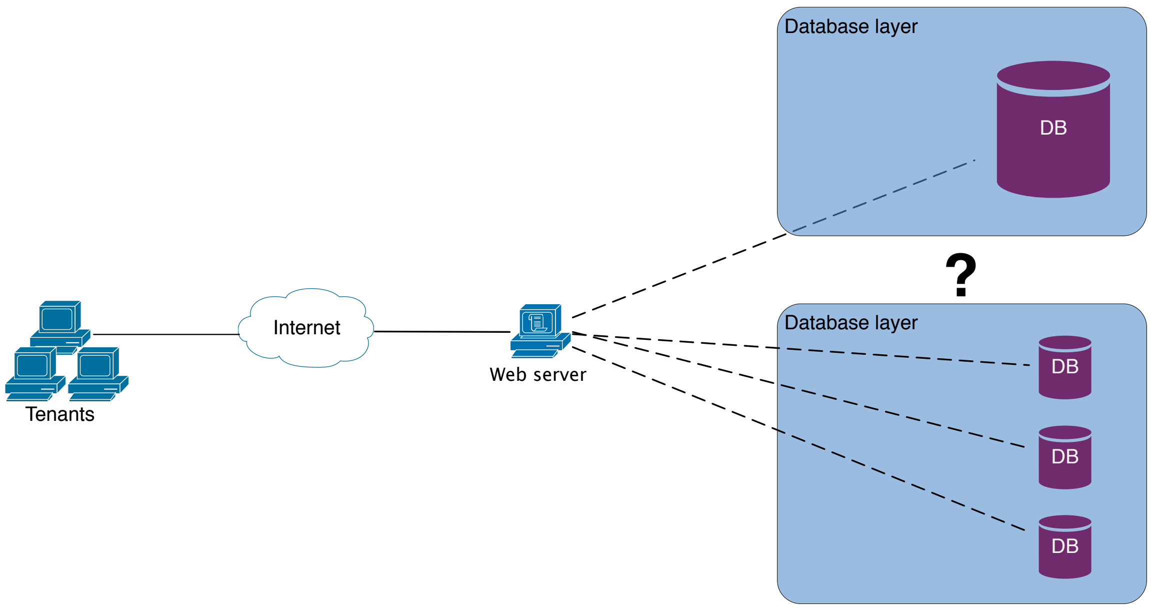


Figure 2. Illustration of the problem statement

As it can be seen in figure 2, there are two different approaches that the software developer can choose to implement. The effect for the end-user is the same. However, one approach assumes one big database to store all tenants data, while the other approach assumes individual databases per tenant.

Each approach has its own advantages but also brings risks. If for instance, if the software developer would naively design the application to run on a single database per tenant (the second approach) then the costs to maintain such an application will increase exponentially with every new tenant. It is a liability unless such an approach is specifically required by the project. On the other hand, if the software developer chooses the first approach and prepare one database for all tenants data, then he will run into problems when the number of tenants increases and they start demanding different features or possibly an own database instance to store their data.

The example presented above is very simplistic, yet very real. If the wrong implementation pattern is chosen from the beginning of a project and later on it is discovered that another pattern would have worked better, faster and even on a lower budget, then the project can become a failure, or the error could be fixed at a high cost.

As we can see, risk is involved when it comes to chose the right implementation pattern. In the current research we aim to manage the risk by clearly showing a number of multitenancy implementation patterns, pointing out their strengths, their liabilities, offering mitigation options and, in the end offering an overview of different problem scenarios and what patterns help solving them.

1.3 Multitenancy implementations

While small organizations may use public cloud solutions with software multitenancy solutions, in large organizations and enterprises private clouds are used and hardware based multitenancy solutions are present. There are many software and hardware vendors that offer multitenancy solutions. In the case of a small organization, finding the right implementation pattern might be limited by the budget and the objectives the company needs to achieve with it. However, in the case of large organizations, the problem of choosing the right multitenancy implementation pattern can become more complicated.The balance between security and flexibility for tenants to interact has a greater weight for large organizations. They can afford paying more to have their own multitenancy solution deployed in a private cloud, running as an instance specially configured for their needs. Thus, it becomes even more confusing when it comes to choosing the right implementation pattern.

1.4 Roles tenants can have

Access to the resources of a business software is granted to the tenants by using roles. Therefore, tenants can have one predefined role, such as “Purchaser” or “Approver”, or even more complex roles in which abilities are defined (Chong and Carraro, 2006).

As it can be seen in the example in Figure 3 below, tenants are given roles which define abilities, which further define the actions they can make.

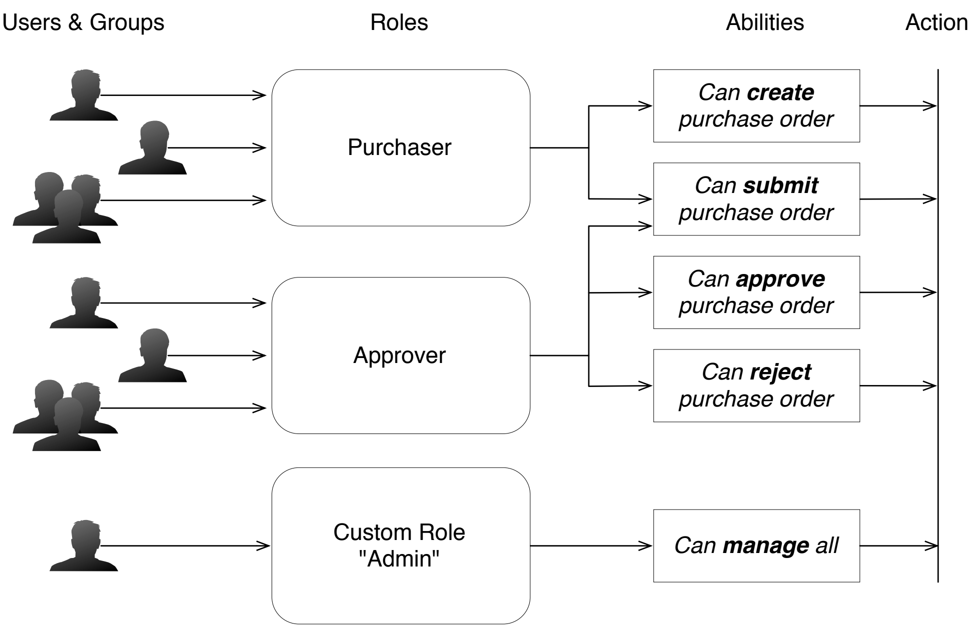

Users & Groups
Roles
Abilities
Action
Purchaser
Approver
Custom Role "Admin"
Can **create** purchase order
Can **submit** purchase order
Can **approve** purchase order
Can **reject** purchase order
Can **manage** all

Figure 3. Example on the roles tenants can have

As the business software moves to the cloud, all tenants access the same instance of the software. Software developers need to design the software in such a way that it can be served to more than one customer. Creating a business software that will be deployed as SaaS, is differently from creating and delivering it to each customer, with the main difference being the way the software is accessed (Ma, 2007).

1.5 The influence of SaaS on business software

There is no comprehensive guide that focuses on the issues mentioned above. With the evolution of on-demand software delivery models such as Software as a Service (SaaS) and unlimited computing power, the costs can be highly reduced and the flexibility is increased.

However, the complexity to deploy and to start serving online the business software to multiple tenants increases (Mietzner, Metzger, Leymann, 2009). It increases because all the resources are virtually in the same place and the tenants are accessing them. Not only the roles and privileges must be clearly defined, but the tenants have their own data that needs to be protected. Some of this data is shared between them, something that was not possible in the distributed model where the software was delivered to customers and run on their local premises, or if it was, at high costs. With the business software being delivered as

SaaS, all the data is stored in the same place, which makes it easier to share resources between tenants.

1.6 Terminology

Several terms are used in the title, problem definition, objective, problem statement, research model and in the research question. Each term is explained and defined in the context of the thesis.

Multitenancy (MT) is a software architecture, where a single instance of the software runs on a software-as-a-service (SaaS) vendor's server, while serving multiple client organizations (tenants). MT is contrasted with a multi-instance architecture, where separate software instances (or hardware systems) are set up for different client organizations. With a multitenant architecture, a software application is designed to virtually partition its data and configuration, so that each client organization works with a customized virtual application instance.

Business Software is a software program used for administrative tasks, that helps businesses to increase or measure the productivity.

Online Business Software is business software that is run online, in a SaaS cloud infrastructure. Clients are required to have an internet connection in order to access it.

Software as a Service (SaaS) refers to software delivery model where the software and its data are hosted in the cloud and the clients access them through a client (web browser or thin application).

1.7 Research questions

The main research question is:

What architectural patterns are most suitable for multitenant software products?

To answer the main research question, the following research subquestions will be addressed in this research:

1. **What are current multitenancy implementation patterns?**

The first subquestion will determine four multitenancy implementation patterns;

2. **What are the advantages and liabilities of these patterns?**

The second subquestion will analyze the four multitenancy implementation patterns determined in the first research subquestion and determine their advantages and liabilities*;*

3. **What is the appropriate multitenant business software pattern that should be chosen for developing multitenant business software?**

The last subquestion will give an overview of multitenant specific problems and show how the four implementation patterns can solve them.

Research Question	Research Methods	Deliverables
What are current multitenancy implementation patterns?	Literature study, Expert interviews	Multitenancy implementation patterns;
What are the advantages and liabilities of these patterns?	Interview and Literature analysis	Analysis on multitenancy implementation patterns based on the literature review.
What is the appropriate multitenant business software pattern that should be chosen for developing multitenant business software?	Case study	Overview of multitenant specific problems and solutions using the implementation patterns.

Table 2. The research subquestions, research methods and their deliverables

In table 2 above we show what research methods the research subquestions use and what the deliverables are.

1.8 Research question operationalization

Below the research subquestions will be stated together with the activities that will be performed to answer them. At the end of the chapter the bigger picture is presented (the PDD) which shows exactly how the deliverables from the research subquestions are connected in order to answer the main research question. Moreover, each of the deliverables is described, as it is easier to see how they are part of the bigger picture.

Subquestion 1

What are the current multitenancy implementation patterns?

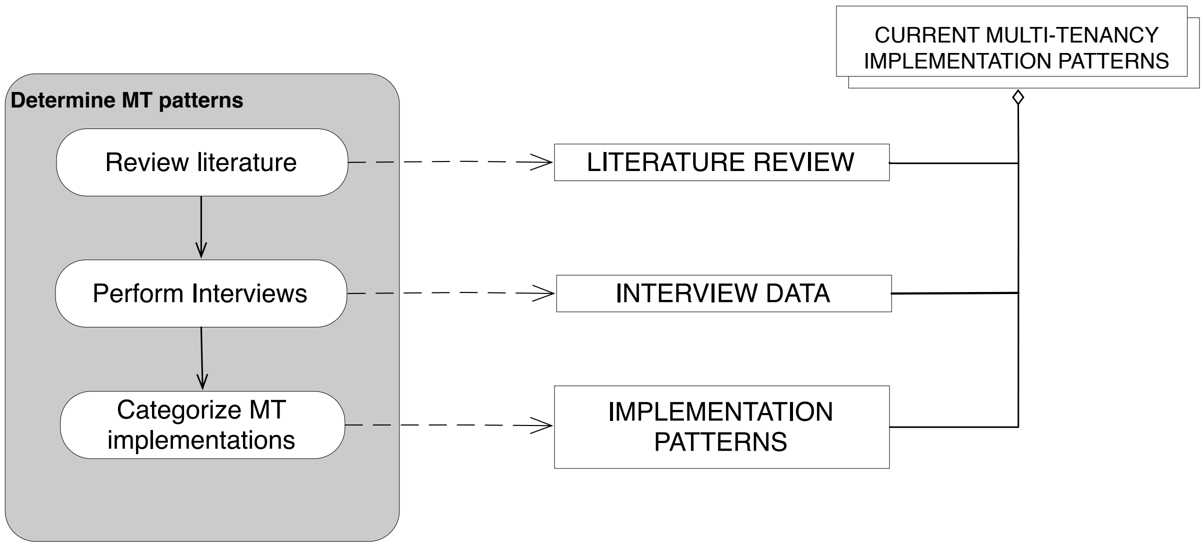


Figure 4. Activities and deliverables for subquestion 1

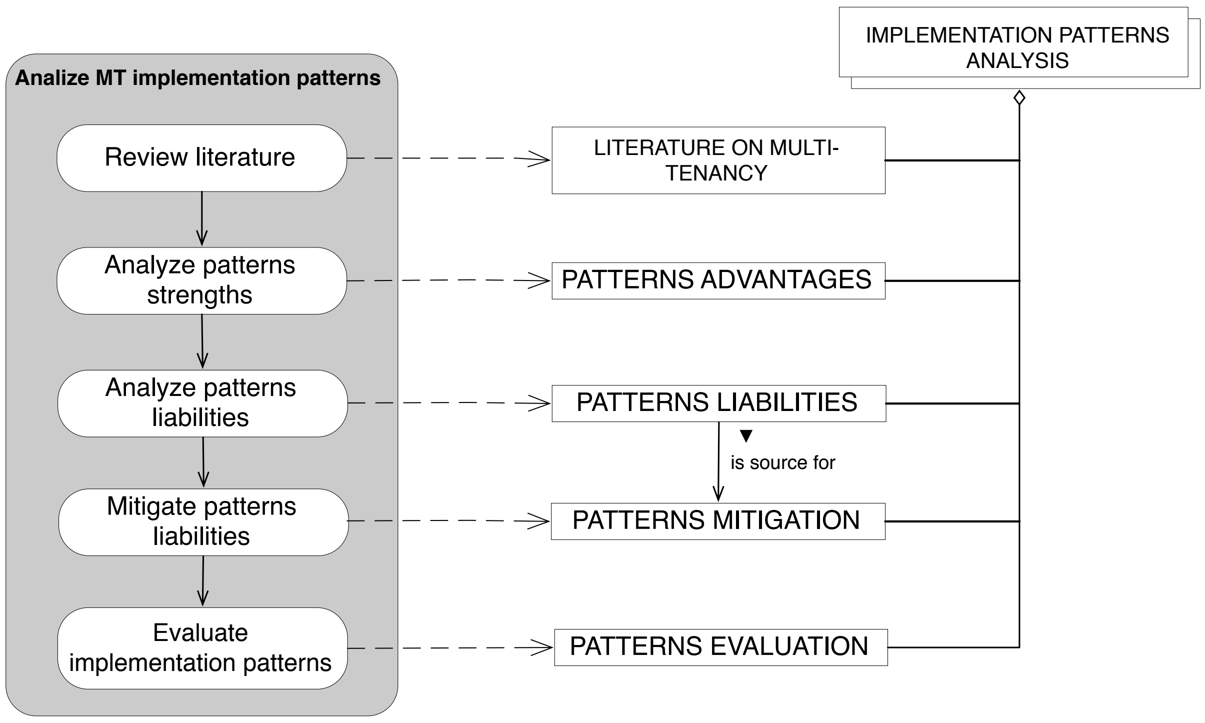
The activities in Figure 4 are described in table 3 below.

Activity	Sub-activity	Description
Analyze current Multitenancy implementations	Review literature	Prepare a LITERATURE REVIEW for two current multitenancy implementations.
	Perform Interviews	Perform two case-studies based on interviews to determine MT implementation patterns. Data will be collected as INTERVIEW DATA.
	Categorize MT implementations	The current multitenancy implementations identified by the previous activity are categorized in IMPLEMENTATION PATTERNS.

Table 3. Research subquestion 1, activity, sub-activities and descriptions

Subquestion 2

What are the advantages and liabilities of these patterns?



Analize MT implementation patterns
Review literature
Analyze patterns strengths
Analyze patterns liabilities
Mitigate patterns liabilities
Evaluate implementation patterns
IMPLEMENTATION PATTERNS ANALYSIS
LITERATURE ON MULTI-TENANCY
PATTERNS ADVANTAGES
PATTERNS LIABILITIES
is source for
PATTERNS MITIGATION
PATTERNS EVALUATION

Figure 5. Activities and deliverables for research subquestion 2

The activities in Figure 5 are described in table 4 below.

Activity	Sub-activity	Description
Analyze MT implementation patterns	Review literature	Study LITERATURE ON MULTI-TENANCY that introduces the relevant work on the topic of multi-tenancy such as databases and security.
	Analyze patterns advantages	The PATTERNS ADVANTAGES are determined based on the CURRENT MUILTI-TENANT IMPLEMENTATION PATTERNS and the LITERATURE ON MULTI-TENANCY
	Analyze patterns liabilities	PATTERNS LIABILITIES are determined discovered in the previous two activities are put together and the TENANTS CATEGORIES list is formed.
	Mitigate patterns liabilities	PATTERNS MITIGATION techniques are offered based on the PATTERNS LIABILITIES and taking into account the LITERATURE ON MULTITENANCY
	Evaluate implementation patterns	Send the implementation patterns to multitenancy experts and ask for their feedback.

Table 4. Activity, sub-activities and description for research subquestion 2

Subquestion 3

What is the appropriate multitenant business software pattern that should be chosen for developing multitenant business software?

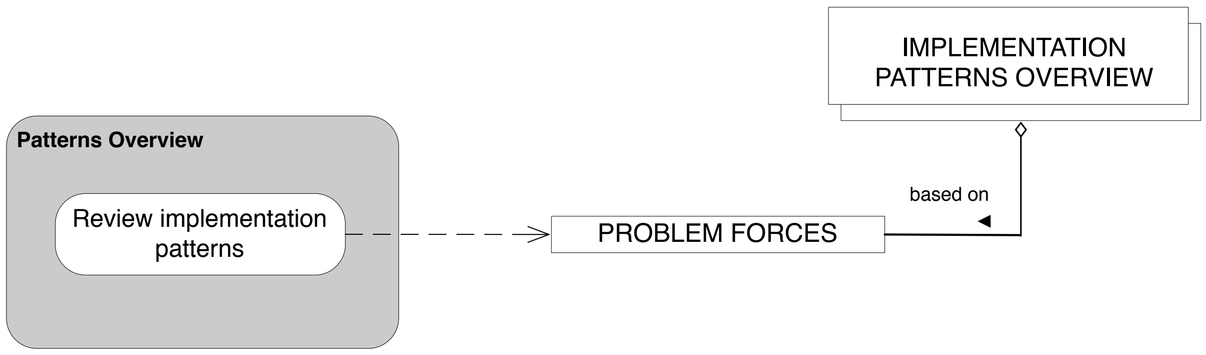

IMPLEMENTATION PATTERNS OVERVIEW
Patterns Overview
Review implementation patterns
based on
PROBLEM FORCES

Figure 6. Activities and deliverables for subquestion 3

The activities in Figure 6 are described in table 5 below.

Activity	Sub-activity	Description
Patterns Overview	Review implementation patterns	Gives an overview for the CURRENT IMPLEMENTATION PATTERNS based on the PROBLEM FORCES and the PATTERNS ANALYSIS.

Table 5. Activity, sub-activities and description for research subquestion 3

1.9 Scientific contribution

From a scientific perspective, this research adds to the body of knowledge on the topics of multitenancy, business software, and multitenant architectures.

Firstly, this research will bring together a part of the existing literature that is relevant to the subject of multitenancy software in the business software context. The first research sub question will make an overview of the current multitenancy implementations resulting in four multitenancy implementation patterns. The second research sub question will provide an analysis of the four multitenancy implementation patterns determined in the first research sub-question.

Secondly, there is little research that gives a complete overview of four multitenancy implementations and shows their their pluses and the problems the patterns caused. The method developed in this research will add to the body of knowledge since it will map the current multitenancy implementations with an extensive literature research and this could be used in future research, such as designing new multitenancy business software or figuring out the advantages and liabilities of multitenancy implementation patterns.

Finally, the research addresses the concept of multitenancy from a business software perspective, making it easier for researchers to analyze the subject of multitenancy. By doing literature research, further research could have the grounds on the current literature research that will include the current multitenancy implementations and the tenants interaction analysis.

1.10 Business relevance

The results of this research are beneficial for companies that start building and designing their software and need to offer it to multiple tenants because the findings of this research will help them in creating their business software that works on the existing proven implementations. In order to provide a benefit for multitenant software developers, current multitenancy implementations are analyzed, and their advantages and disadvantages are determined.

Overall, the work is beneficial for everyone that needs to offer a SaaS multitenant business application.

Chapter 2 - Research approach and methods

In the current chapter, the research framework, and methods are introduced.

2.1 Research framework

Figure 7 shows how this thesis project fits within the Information Systems Research framework designed by Hevner et al. (2004).

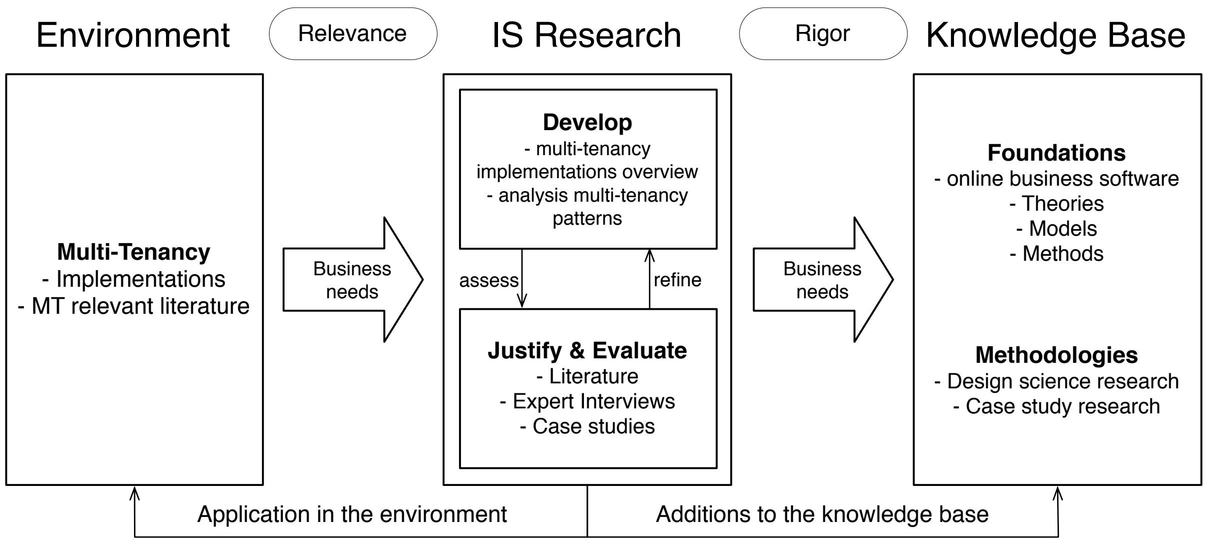


Figure 7. The thesis project within the Information Systems Research framework, adapted from Hevner et al. (2004)

2.2 Research methods

This research uses design research (Vaishnavi & Kuechler, 2004 and 2007) as the main research method. The five steps the design research cycle consists of. are applied as follows in this paper:

- **Problem awareness** - the lack of research that gives an overview of multitenancy implementation patterns;
- **Suggestion** - the solution will consist of a method that maps the current multitenancy implementations with their advantages and disadvantages;
- **Development** - the actual method is developed by answering the research questions from 1 to 3.

- **Evaluation** - the method is evaluated by performing a validation case-study with an expert in multitenancy business software.
- **Conclusion** - once the method has been validated, the results are summarized and the design cycle is complete.

To validate the method, this research will use case studies on multitenant web applications. The case study will be performed according to the guidelines defined by (Höst & Runeson, 2009). In their paper, they mention that in order to plan a case study, the elements mentioned in the list below should be present in the plan (Robson 2002):

- **Objective** - to create a method that solves the problem, namely it gives a complete overview of four multitenancy implementation patterns and analyzes them.
- **The case** - this research studies the current multitenancy implementations, and structures that information in a meaningful way that will be used in the multitenancy method.
- **Research questions** - are mentioned in the previous chapter.
- **Methods** - this research will use several data sources, namely: study of the related literature and conducting interviews with field experts.
- **Selection strategy** - data will be collected mainly from the case study companies, through expert interviews.

In order to answer the main research question and the research subquestions, literature research and case studies will be used.

2.3 Literature Research

Literature research was performed in order to collect information to identify multitenancy patterns and to analyze them and in the end offer mitigation solutions to their liabilities. First, data was collected by consulting the search engines, mainly Google Scholar (http://scholar.google.com). Other search engines were used as well such as CiteSeerX (http://citeseerx.ist.psu.edu), Springer (http://www.springerlink.com) and the Utrecht University's Library (http://www.uu.nl/university/Library). The ones listed above were the main places where literature was searched, although, books on Amazon, Google Books, or publications on IBM, Cisco, HP, Microsoft were used as well and in some cases even blogs and wiki pages for fine-grained search results.

The literature study begins with an introduction to multitenancy and its characteristics. After the multitenancy topic was introduced, and the characteristics were defined, search was performed on multitenancy database solutions, customization, variability, service performance, and security.

Topic(s)	Search Keywords
Multitenancy definition and characteristics	• Multitenancy definition • Multitenancy characteristics • Multitenancy business software • Multitenancy SaaS • Multitenancy challenges
Multitenancy database solutions	• Multitenancy database scalability • Multitenancy database security • Multitenancy customization • Multitenancy variability

Table 6. Search keywords to find related literature

Table 6 above lists some of the keywords that were used to find the relevant information. In some cases search queries were quoted in order to help the search engine to look for the exact keyword that we wanted. For instance, when scalability was the current topic, keywords such as: multitenancy database “scalability” were used to ensure that the results are related to multitenancy databases but also include scalability. Quotations were needed because database scalability is such a broad term that often the search engines would ignore the multitenancy part of it, returning irrelevant search results.

Search engines return many pages of results, but in the current research we only used papers that were displayed in the first two pages of results, that means we only used the first 20 results. The reason is that we often noticed that from page two onwards the results were no longer relevant.

Besides the use of search engines, some of the papers had good references to other papers. This was also a gateway to find new papers that were on the same topic. For instance, while reading a paper on database scalability solutions, before the authors proposed their solution, they analyzed some of the solutions proposed by other authors, and referenced their work as well. We used these references to find relevant work on the current topic.

An important aspect is that there is not much literature available on the multitenancy topic. Also, we did not review all the publications available, but after studying a couple of papers, we started recognizing references to papers already studied. For instance, there were many papers that were using some definitions from another paper or studied the same problem referencing to others work but ended up with different solutions. At that point, reviewing more work did not add more value because we considered the subject saturated. Something similar happened with the multitenancy database solutions subchapter. Overall, around 100 publications (that includes papers, white papers, journal publications, articles, chapters from books) were studied, although, not all of them were summarized or referenced in the research.

In order to make sure that the literature research that we perform is correctly done, we followed the principles described by Webster and Watson, (2002) on how to write a literature review and how to identify the relevant literature. As the authors say in their paper, “Studies of the IS literature have consistently been limited by drawing from a small sample of journals.”, in our research we followed a structured approach as the one the authors recommend. The first step was to identify the major contributions from the leading journals. Then we went “backwards” and reviewed the citations for the articles identified at step one in order to determine prior articles to be considered. Finally, we did “forward” research as the authors say, and we used the search engines to identify articles that use the same keywords and included the relevant ones in our review. After completing these three steps, we started to recognize patterns on a certain topic such as repeating references, point at which we considered the subject being saturated.

Also, it is important to notice that multitenancy is a broad term. Multitenancy can be associated with almost every topic that has to do with software. The reason is because multitenancy has to do with databases, web applications, security, authorization, scalability, and many others. In our research, we only cover a couple of these topics. After the we performed the case studies, and also while studying the first chapters, we noticed that the database plays an important role in multitenancy, and we used it as a leading topic for our patterns. Then security with authorization and authentication were also widely discussed, not only in papers but also on web blogs and we considered them important as well.

Concept-centric literature review
Multitenancy Concept X: (author A, author B, ...)
Multitenancy Concept Y: (author A, author C, ...)
...

Table 7. Concept-centric approach to literature review

The structure of the literature review is concept-centric, as opposed to author-centric (Webster and Watson, 2002). At the end of chapter 3, the concept matrix can be found. It shows that the review is structured and the literature is well synthesized. The structure of the component matrix is similar to table 7.

In the end, the review ends with a conclusion which is represented in a table at the end of the chapter. The conclusion walks through the topics that were discussed, briefly explains what we learned from them and shows how they are used in the research and to what particular research question they are part of the answer.

2.4 Case studies

As described in the section above, four case studies on multitenant business software applications are performed. They will be used as data collection to determine the four multitenant implementation patterns. Two of the case studies will be done through extensive literature research and the other two will be done with expert interviews. In the end, an evaluation interview with one expert will be performed, in order to evaluate one implementation pattern. According to Yin, (2003) , four tests are defined to establish the quality of the research by four tests: construct validity, internal validity, external validity, and reliability.

According to Yin (2003) the following four phases are defined for case studies:

1. **Design case study**
   This research will perform use an embedded case study design. The embedded case study will use a single-case design. The case study will be performed on multitenant web applications and the unit of analysis consists of multitenancy implementations.

2. **Conduct case study:**
   The case studies are performed by conducting interviews over the phone, video calls or in person meetings. The interviews aim to get more information about the multitenancy implementations, for research subquestion two or for finding out the interviewees' opinion about the multitenancy method that is developed as the output of this thesis.

3. **Collect data**
   Data will be collected from the interviews and literature study. The questionnaires will be either discussed or filled-in by the interviewees. Besides the questionnaires, a source of data is the survey from research subquestion three in which we validate and test our method. According to O Leary (2004): “Collecting credible data is a tough task, and it is worth remembering that one method of data collection is not inherently better than another.” Therefore in our research, we will use two sources of

data collection, namely literature research (as described in chapter 2.3) and expert interviews (as described in chapter 2.5).

4. **Analyze case study evidence**

   The last step of the case study is to analyze the case study evidence. Here data from the survey is analyzed and further corrections are made if the multitenancy implementation method still needs to be updated.

In this research, the case studies will follow the four phases by Yin (2003), discussed above.

2.5 Expert interviews

Interviews were conducted with experts in multitenancy business software. The scope of the interviews is collect data and to gain knowledge from multitenancy experts about the multitenant software products they develop.

According to Kajornboon, (2005), "the interview starts before the interview actually begins". It means that the interview needs to be prepared and the paper by Kajornboon points the steps that need to be taken, which we have followed in our preparations, namely to offer a clear idea: about why they have been contacted, about the research and why the interview takes place, about the length of the interview, and finally, about the location and time the interview will take place.

In our research we performed two semi-structured interviews because the same questions were asked of all respondents, however the discussion was opened and the order of questions changed depending on the direction of the interview. We prepared a list of questions the same as for a structured interview, however we preferred an open discussion during the interview and used it as a general guideline.

The questions were divided in two topics, namely: application characteristics and multitenancy implementation types. The questions in the first section were more general, regarding the business side of the application, such as *how many tenants use the software* or *can tenants customize how the application looks (templates, personal logo)*. The second part of the interview, *multitenancy implementation types*, was more technical and we divided it in six categories:

1. **General**: these questions are about the releases of the application, if the application maintains backwards compatibility and other facts such as if the application allows offline working and sharing of resources. Also, we asked questions whether the company that produces the software is using it for its own needs of if they consider the implementation was successful.

2. **Security**: questions regarding the security of the application. Security plays an important role in multitenant business software and after studying the literature, we asked questions that we knew were important subjects in existing research. Such questions included: *Where there any successful attacks?* or *Was ever data lost?* and *Were there any security breaches discovered?*

3. **Architecture and API** questions: these were more technical questions. We asked for details about the architecture of the application and if it offers an application programming interface (API) for outside developers. One of the most important questions was if there is one or many instances of the application that serve the tenants. The answer of this question plays an important role to figure out the multitenancy implementation pattern. Also the presence of an API is important in figuring out the implementation pattern. Other questions included in this category,

does the software work with public cloud service providers and *does the application use background processes for requests that take long* and others of the kind.

4. **Database**: the questions in this category refer to the database of the application. We have learned from the literature review that database is one of the most important parts of the multitenancy application, thus it deserved a separate category of questions in our interview. The questions about the database varied from the type of database server they use, to the advantages or disadvantages of these servers, to how metadata is stored, whether or not data is encrypted and most importantly if data of multiple tenants is stored in the same database. Other questions included the location of the database servers, if they use cloud providers to store data, and even an estimate average of number of queries per database request. Moreover, because it is related to storage, questions about static files storage (such as images, and documents that are not usually stored in the database) were asked in this category.

5. **Authentication and authorization**: questions were asked about the authentication and authorization mechanisms. Another aspect that we asked about was whether the application allowed tenants to authorize third parties to access data on their behalf, which answer would influence the implementation pattern.

6. **Accessibility** questions were related to how tenants can access and use the application. Some of the questions were related to offline working, since the online business applications require an Internet connection, we were interested to see if tenants could also use the application without being online. Moreover, we asked if the application is available on mobile devices and if the tenants required special connectivity settings such as VPNs to access the software outside their organizations or even install additional software to run the multitenant application. Another aspect that we were interested about is where the multitenant application is deployed and if the experts chose public cloud providers to host their applications.

It is important to notice that the six categories defined above are not clearly delimited. For instance there were questions in the database category because they were mostly about the database of the application but they could also fit in the security category, such as the question that asked if tenants data is stored encrypted.

The interview did not follow the questions in any particular order. The list of questions was sent to the experts a couple of days before the interview took place so they could read them and prepare some of the answers. Then the interview was more of a discussion than simply following the questions one by one. Because there were more open questions, the whole interview was recorded for later processing. After the interview ended, the recording was transcribed (see Appendix 1 and 2 for transcriptions).

After the interviews were processed, the case studies (chapter 4) and the multitenancy implementation patterns (chapter 5) were sent back to the interviewees to ask for their comments. For one of the interview some parts had to be removed because they exposed sensitive information about different aspects of the architecture or the application. Having the approval of the experts, the interview process was complete.

For this research, two interviews were performed at Dutch companies that build multitenant business software. The interviews were held in person. One of the interviews was informal while the other one was formal taking place at the case company. Both interviews were recorded and the length of the interviews were 38 minutes for Ledensite and 59 minutes for Afas.

Company	Interviewee’s position	Case study	MT implementation pattern derived from case study
Ledensite	Lead developer	Ch. 4.1	Ch. 5.1
Afas	Lead of internal IT department as well as SaaS platform	Ch. 4.2	Ch. 5.2

Table 8. List of companies that were interviewed, resulting case studies and implementation patterns

In table 8 above, we list the two companies that were interviewed. The interviews were processed in case studies and the information in the case studies was used as input for the multitenancy implementation patterns that derived from these case studies.

As it was mentioned above, semi-structured interviews were performed (Kajornboon, 2005). We have chosen semi-structured interviews as opposed to structured and unstructured because it allowed us to adapt to the situation. There were some general questions in the interview, such as if the application provides an API. If the answer was no, then the other questions that were related to API were skipped. At the same time, during the interview the discussion lead to more answers than the questions on the list, so it was possible to ask for more details and to be flexible to have a conversation rather than a checklist of questions.

Another advantage comes from the fact that semi-structured interviews work well if the researcher does not test a specific hypothesis (Kajornboon, 2005). In our research, we are looking for as much information from the interviewee to collect in order to create a multitenancy implementation pattern, and not to validate a hypothesis. Therefore, semi-structured interviews are suitable for the needs of this research.

However, there are some drawbacks to the semi-structured interviews. According to Kajornboon (2005), the main drawback to semi-structured interviews lays is the inexperience of the interviewer to ask prompt questions. However we believe that this was not the case since all the data that we gathered from them was enough to build the multitenancy implementation patterns and there were no further questions after the interview ended.

Overall, besides literature study, we use semi-structured expert interviews as well, to collect data and to gain knowledge on multitenancy business software products from experts in the field. In this research two semi-structured interviews were performed resulting in two case studies on the top of which two multitenancy implementation patterns derived.

2.6 Validity

In order to establish the quality of our research, we are using the four tests recommended by Yin (2003). The four tests and how we are applying them are:

1. **Construct validity**

   Confirms that correct operational measures are used for the concepts being studied;

   The test assumes that the following two steps are covered, namely: (1) we “select the specific types of changes that are to be studied (and relate them to the original objectives of the study)”, and (2) we “demonstrate that the selected measures of these changes do indeed reflect the specific types of change that have been selected.”

In order to satisfy the first step, we study the changes (differences) between four multitenancy implementation patterns by analyzing the multitenant applications. We use multiple sources of evidence, namely two from interviews with experts and other two from literature study. The changes that we identify will form the chain of evidence that will be the base for creating the implementation patterns. For the second step, the chain of evidence from step one is used to create the multitenancy implementation patterns. To satisfy the second step, we need key informants to review the draft case study report. Therefore, we plan to send a draft of the case study report for review to one expert in multitenancy.

2. **Internal validity**

   Is used for making sure that the casual relationships (where applicable) are properly linked.

   As the author says in the book, the internal validity assumes that if an event *x* led to an event *y* and if we incorrectly assume that there is a relationship between *x* and *y* then we need to assure there is no external factor *z* that caused *y*, otherwise the internal validity fails due to missing threats.

   In our research we apply the internal validity in the sense that if we have a set of multitenancy implementation patterns and we try to create a new one, every component that we want to add in the new pattern needs to be compared against the components of the existing patterns. As long as components match and their connections too (this defines the relationship between two components) then we cannot create a new pattern, otherwise the internal validity fails.

   If we were to write an equation for the internal validity, we would model it as follows:

$$MTpatterns = \begin{bmatrix} P1 \\ \ldots \\ Pn \end{bmatrix}, Px = \{\mathrm{C}1, \mathrm{C}2, \ldots, \mathrm{C}z\}, Pnew = \{A1, A2, \ldots, Az \mid (A1, \ldots, Az) \notin (\mathrm{Ci}1, \ldots, \mathrm{Ci}z), 1 \le i \le n|\}$$

   Where MT models a matrix of patterns and each pattern models a set of components that are in a relationship with each other, and between any components Ci and Cj there is no Ci’ that influences Cj. For each new pattern identified by P_{new}, we must do pattern matching, in fact we will do pattern component matching to assure that the relationship between its components A_i is unique.

3. **External validity**

   In order to pass this validity test, we must make sure that the findings are generalizable beyond the case study.

   The findings of our case studies are given by the four multitenancy implementation patterns. We plan to satisfy this third step by making sure the patterns are general and they do not include specific components of the pattern that they derived from.

   By using general components in the patterns, such as *database layer* or *business logic layer*, it will generalize our findings, thus satisfying the external validity.

4. **Reliability**

   The final step assumes that if someone else wants to continue our work, this should be possible. In other words, should someone else follow the same procedures as we did and conduct the same case studies, then the findings and conclusions should be the same.

In order to satisfy the fourth validity, we have created a process deliverable diagram (figure 8) in which each step performed in the research is clearly documented by a process, and a concept. Because we designed the process deliverable diagram in an optimal way, to use a minimum number of steps in order to get to the results, then when when someone else conducts the same case studies again, by following the steps in the PDD, then the deliverables will be the same. Moreover, in order to for someone else to continue our work, the only thing that needs to be done is to follow the flow of actions in the PDD (figure 8) like we did, and it will lead to new findings with the same conclusions.

This research is structured in such a way so that in the end all, or at least most of the validity tests will pass. Refer to the discussion chapter for a more detailed analysis of the threats and for a conclusion of which validity tests have fully or partially passed.

2.7 Overall research method

In the previous chapter, the two research methods used by this research were introduced, namely: design science research and case study research. Design science research is used because in the end a multitenancy method is created, while the case study research is performed to get more information in the process of creating the method.

In order to perform the two research methods, there are many steps that will be followed. To illustrate them, they are grouped in activities (processes) with corresponding deliverables. The process-deliverable diagram (PDD) in the figure 8 below offers a brief overview of how this research will be conducted. Process-deliverable diagrams (Brinkkemper, 1996; Weerd & Brinkkemper, 2008) are used to model the different research activities and the resulting deliverables.

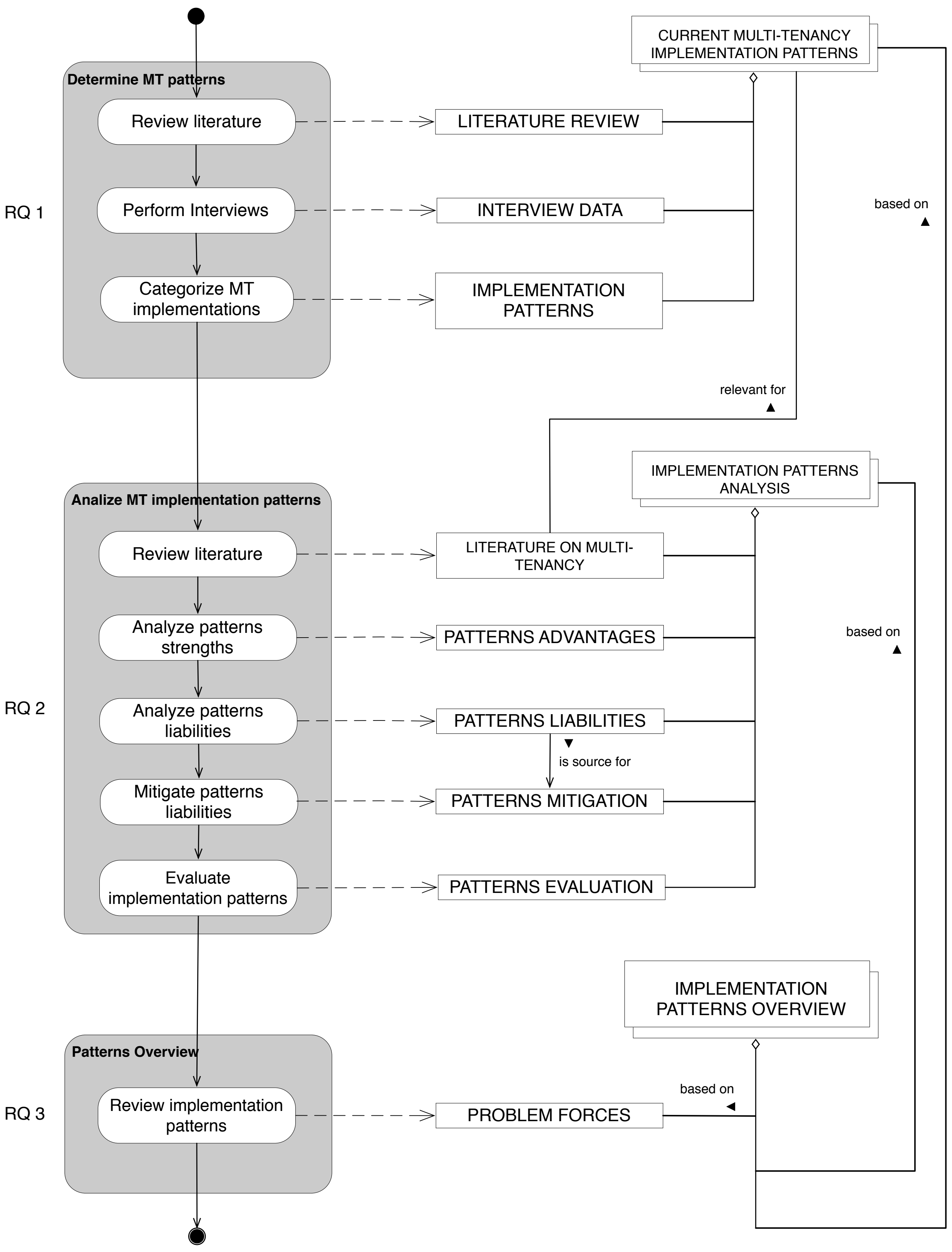

Determine MT patterns
Review literature
Perform Interviews
Categorize MT implementations
RQ 1
LITERATURE REVIEW
INTERVIEW DATA
IMPLEMENTATION PATTERNS
CURRENT MULTI-TENANCY IMPLEMENTATION PATTERNS
based on
relevant for
Analize MT implementation patterns
Review literature
Analyze patterns strengths
Analyze patterns liabilities
Mitigate patterns liabilities
Evaluate implementation patterns
RQ 2
LITERATURE ON MULTI-TENANCY
PATTERNS ADVANTAGES
PATTERNS LIABILITIES
is source for
PATTERNS MITIGATION
PATTERNS EVALUATION
IMPLEMENTATION PATTERNS ANALYSIS
based on
Patterns Overview
Review implementation patterns
RQ 3
PROBLEM FORCES
based on
IMPLEMENTATION PATTERNS OVERVIEW

Figure 8. Process Deliverable Diagram (PDD) of this research

Table 9 below describes the concepts for the Process Deliverable Diagram (PDD) in figure 8.

Concept	Description
CURRENT MULTITENANCY IMPLEMENTATION PATTERNS	**Answer to RQ 1.** Four multi-tenancy implementation patterns: two based on literature research and the other two on expert interviews
LITERATURE REVIEW	Overview of a part of the relevant literature to determine two current multi-tenancy implementation patterns.
IMPLEMENTATIONS PATTERNS	Four figures illustrating the four multi-tenancy implementation patterns.
MULTI-TENANCY PATTERNS ANALYSIS	**Answer to RQ 2.** Table that lists the advantages and liabilities of each of the four multi-tenancy implementation patterns.
LITERATURE ON MULTI-TENANCY	Overview of a part of the relevant literature on multi-tenancy. The literature that is studied is relevant to the four multi-tenancy implementation patterns from research subquestion 1, such as database solutions, and security.
PATTERNS ADVANTAGES	The advantages of the four multi-tenancy implementation patterns are listed.
PATTERNS LIABILITIES	The liabilities of the four multi-tenancy implementation patterns are listed.
PATTERNS MITIGATION	Based on LITERATURE ON MULTI-TENANCY and the PATTERNS LIABILITIES mitigation options are offered to the four patterns.
PATTERNS EVALUATION	Experts' evaluation on the multi-tenancy implementation patterns.
IMPLEMENTATION PATTERNS OVERVIEW	**Answer to RQ 3.** The overview determines what the appropriate multi-tenant business software pattern should be chosen for specific multitenancy problems.
PROBLEM FORCES	The problem forces are given as examples for what pattern is more suitable in which case.

Table 9. The concepts from the PDD and descriptions

The project thesis consists of a number of main phases as illustrated in the PDD in figure 8. Method engineering as described by Weerd & Brinkkemper (2008) is used to construct the deliverables for this project.

Chapter 3 - Theoretical background

The theoretical background chapter presents a summary of a part of the existing literature that is relevant to multitenant online business software. The chapter begins with a short introduction in online business software, it continues with a brief description of the cloud models where the online business software is deployed. After that, the architecture of some of the popular multitenancy solutions is analyzed. Since multitenancy is mostly present at the database layer, several multitenancy database solutions and schema mapping techniques are briefly described. In the end, multitenancy is all about meta-data stored in the database and customization, therefore some examples of how customization is realized in multitenant web applications are given.

The current chapter matches the PDD from figure 8 as it is seen in figure 9 below. The current action is the “Review literature” and the deliverable, the current chapter, is the LITERATURE REVIEW, both filled in with black and white text.

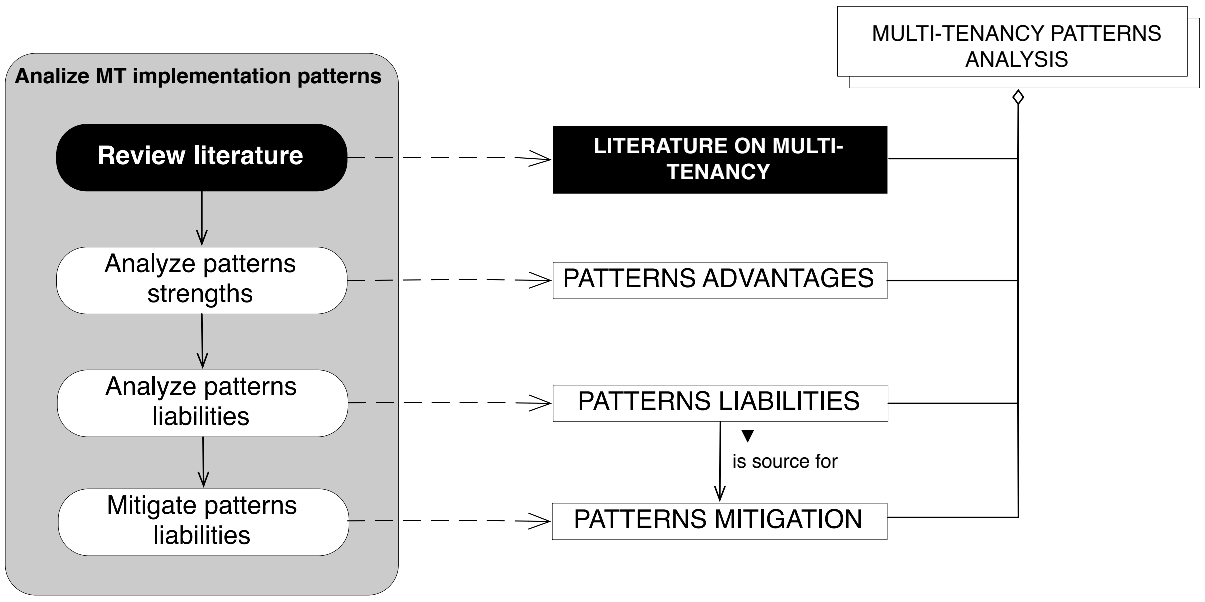

Analize MT implementation patterns
Review literature
Analyze patterns strengths
Analyze patterns liabilities
Mitigate patterns liabilities
MULTI-TENANCY PATTERNS ANALYSIS
LITERATURE ON MULTI-TENANCY
PATTERNS ADVANTAGES
PATTERNS LIABILITIES
is source for
PATTERNS MITIGATION

Figure 9. The current chapter mapped on the PDD from figure 8

A graphical representation of the current chapter’s structure is illustrated in the figure 10 below.

Note that all the subchapters of chapter 3, from 3.3 until 3.8, are references in one of the case studies in Chapter 4. It might not be clear to the reader why they relevant at the moment but after reading them, and diving into Chapter 4, they will make sense. These summaries are here because they all provide solutions to multitenancy problems (such as the database and variability subchapters) or they raise known problems (such as the ‘Security’ subchapter) and are required for the advantages, liabilities and solutions to the four multitenancy implementation frameworks in Chapter 4.

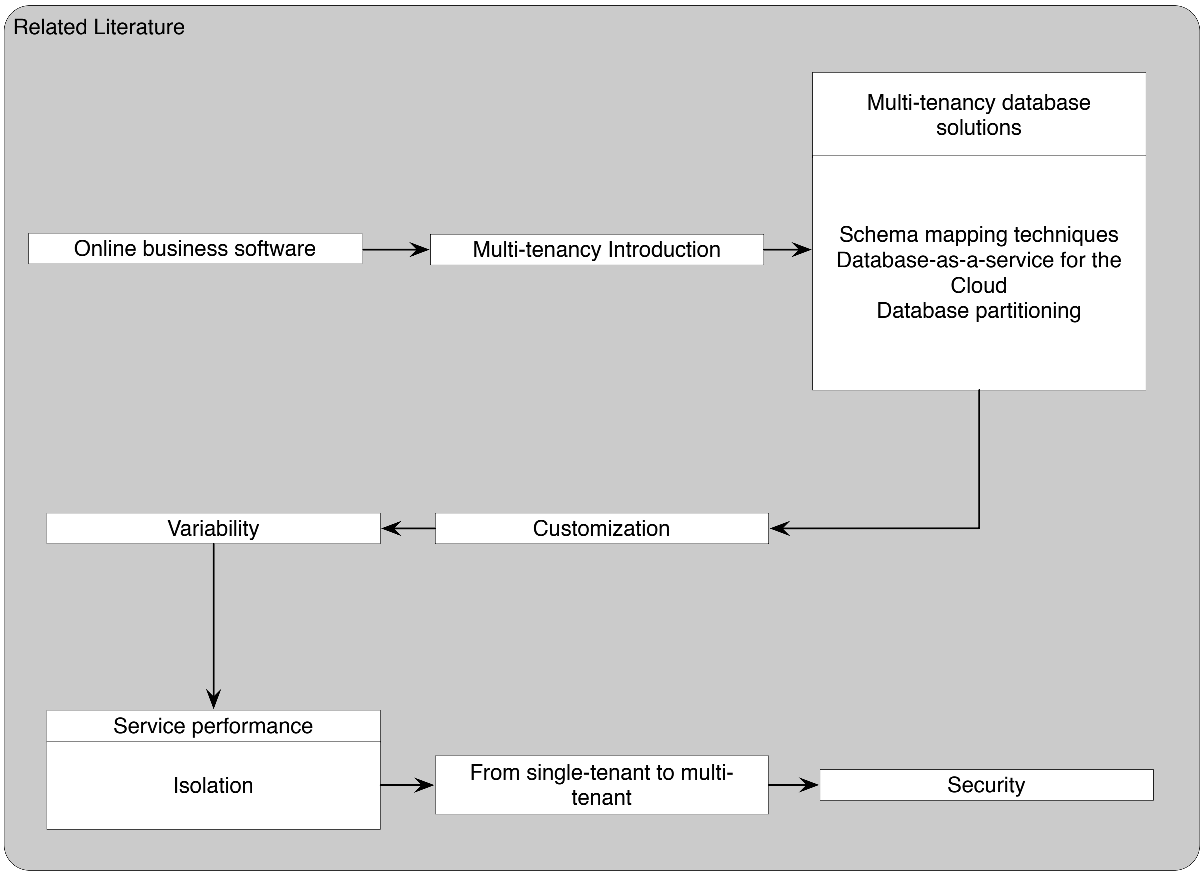

Related Literature
Online business software
Multi-tenancy Introduction
Multi-tenancy database solutions
Schema mapping techniques
Database-as-a-service for the Cloud
Database partitioning
Customization
Variability
Service performance
Isolation
From single-tenant to multi-tenant
Security

Figure 10. Graphical illustration of the related literature structure

3.1 Online business software

The trend for business software has changed, from running as multiple instances at clients premises, to a more centralized approach where one instance of the software serves multiple clients (tenants) which have or are connected to other clients themselves (figure 11). (Aulbach et al., 2008). However, in order to serve multiple tenants from one instance of the software, just a central server that runs the business software is not enough.

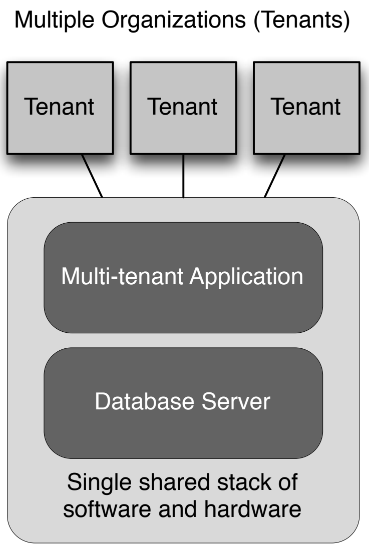

Multiple Organizations (Tenants)
Tenant
Tenant
Tenant
Multi-tenant Application
Database Server
Single shared stack of software and hardware

Figure 11. Simplified example of a deployed multitenant application

The increasing popularity of cloud providers such as Amazon Web Services and Rackspace have enabled software developers to achieve things that were not possible before, namely: have access to infinite resources at a very low price, scale only when needed, store multiple replicas of the critical data and a break-through reliability offered from the hosting provider. All these summed up enabled the creation of services that are used by millions of tenants daily without them having to think where their resources are located, having to think whether the service provider will fail or whether their data is stored secure enough. Among these services GitHub, a popular code repository management system, is described in great detail in the following subchapters. It is worth to mention that all offer multitenancy and while individuals can use them, companies use them too for their teams. These services exist because of the advances made in technology that led to the creation of computing clouds.

There is not a formal definition of clouds. Jin, Ibrahim, Bell, Gao, Huang and Song (2010) make a very clear introduction, stating that: "a Cloud is essentially a class of systems that deliver IT resources to remote users as a service. The resources encompass hardware, programming environments and applications. The services provided through cloud systems can be classified into Infrastructure as a service (IaaS), Platform as a Service (PaaS) and Software as a service (SaaS)." Throughout the paper we will use the definition of clouds by Jin et. al.

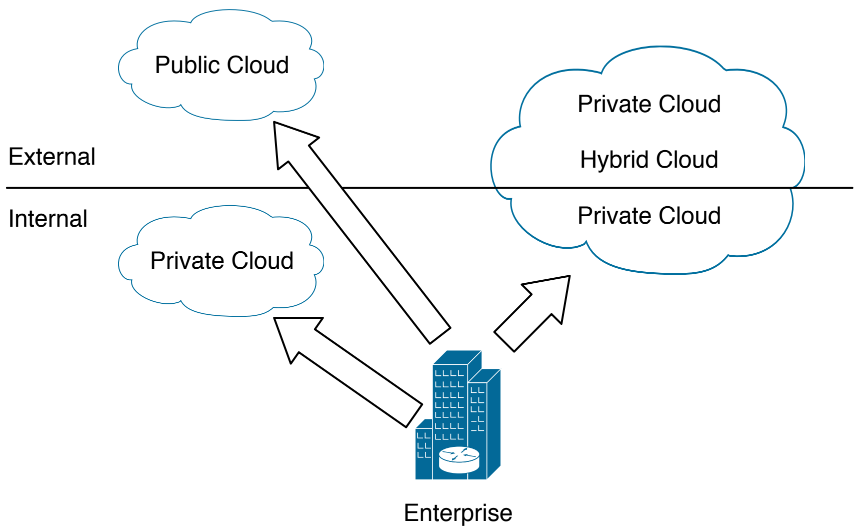


Figure 12. Three cloud types: public, private, and hybrid

According to Jin et. al., there are four types of clouds, namely: public, private, hybrid. Each of these can be used to run a business software, therefore a short description of each is provided below in order to make it more clear how and why a specific cloud type is chosen for a specific business software. In figure 12 above, three types of clouds are mentioned. It shows clearly how the organization stands compared to the cloud.

3.2 Multitenancy introduction

The aim of this sub-chapter is to give an introduction to multitenancy, and present some of the relevant literature that introduces the basic concepts and definitions of multitenancy. In this sub-chapter multiple definitions of multitenancy will be presented together with their sources. Also the context of multitenancy will be clearly explained as it is described in the existing literature. It is important to clearly specify what multitenancy is in order to eliminate the confusion. All the definitions are quoted and listed exactly as they appear on the source.

3.2.1 Definitions

According to Wikipedia's page about multitenancy, "Multitenancy refers to a principle in software architecture where a single instance of the software runs on a server, serving multiple client organizations (tenants). Multitenancy is contrasted with a multi-instance architecture where separate software instances (or hardware systems) are set up for different client organizations. With a multitenant architecture, a software application is designed to virtually partition its data and configuration, and each client organization works with a customized virtual application instance."

However, Warfield (2007) wrote an article where he argues that Wikipedia's definition is clear to the point where it uses the term "partition" in the definition which is linked to "Logical partition (virtual computing platform)". The definition Wikipedia uses for partition comes from the Redbook from IBM by Singh, Castiglion, Dymoke-Bradshaw, Hanninen, Ranieri, and Kappeler (2010), and states that "a logical partition, commonly called an LPAR, is a subset of computer's hardware resources, virtualized as a separate computer. In effect, a physical machine can be partitioned into multiple logical partitions, each hosting a separate operating system."

In Warfield's view, there is a lot of confusion that comes from using the particular notion of how multitenancy can be implemented (such as logical partitions) with the concept itself. Because, Warfield does not agree with the definition on Wikipedia, stating in his article that the previous one together with other definitions found on the Internet are implementation-specific, he is makes his own definition for multitenant software as a piece of software that has a list of properties, viewed from the involved parties, namely: the SaaS vendor, the tenant (organization), and the user (the actual user belonging to that tenant). For the SaaS vendor, multitenancy permits serving multiple customers from a single software instance installed on multiple servers. For the tenant, multitenancy is transparent in the sense that the instance that the tenant runs looks as if it was deployed separately for that tenant, and there is no way to figure out that other people are using the same software from the same instance. Finally, the end-user sees just an application as any other application, without having to know exactly what happens behind and if other tenants / organizations are using it as well. For the end-user it should just work.

In the following chapters of the related-literature, most of the papers use a definition of multitenancy that is very simplified and similar to what Wikpedia defines. While this definition is enough for a specialized paper (for instance a paper that researches database configurations for multitenancy), in this research, the term multitenancy should be rigorously defined and tackle the term from all angles, like Warfield did and explained in the previous paragraph.

The following paper summarized below was chosen because it gives a very clear introduction to multitenancy, it defines the term and shows its main characteristics. In order to do research on multitenancy, we think that a deep understanding of the term and its main characteristics is required. Therefore, until the end of the sub-chapter the main paper in discussion is the paper written by Bezemer and Zaidman (2010).

In the paper by Bezemer and Zaidman (2010), the authors aim to introduce the term multitenant and compare it against multi-user and multi-instance. They use two definitions for multitenancy. The first definition states: "a multitenant application lets customers (tenants) share the same hardware resources, by offering them one shared application and database instance, while allowing them to configure the application to fit their needs as if it runs on a dedicated environment.". The second definition states that "a tenant is the organizational entity which rents a multitenant SaaS solution. Typically, a tenant groups a number of users, which are the stakeholders in the organization."

Because it is unclear at this point where the border is between multitenant, multi-user and multi-instance, the authors make two short comparisons between each versus multitenant to make the difference between them clear. Firstly, they compare multitenant versus multi-user and state that in a multi-user application, are using the same application and are limited when it comes to configuring it. However, in a multitenant application, each tenant can configure the application and the options to do so are not restrictive as in a multi-user environment. That means besides how it looks, the application can behave differently for multiple tenants while it will always behave the same for multi-user. The authors say that even the agreements with the service provider can differ from two tenants (such as one requiring high availability of the software and the other not) while for multi-user the service provider will always use the same service level agreements with its users. Secondly, multitenant is compared versus multi-instance. Multi-instance means that virtualization is used and a virtual instance of the application is deployed per tenant. On the other hand, a multitenant application serves all tenants from the same instance of the software.

3.2.2 Characteristics

The paper (Bezemer et. al.) continues with some characteristics of multitenancy. The hardware is shared in a multitenancy environment. Compared to a single-tenant application where each tenant would install the software on their premises, in a multitenant application, a SaaS solution is used to deploy the application. The software consists of the application itself and the database where tenants' data is stored. There are many approaches to this, all sharing the application while the database can be configured differently. The different configurations for the database server for a multitenant application are: separate database per tenant, shared database but separate tables per tenant and finally shared database with shared tables which the authors describe as "pure multitenancy".

The second characteristic that is mentioned in this paper (Bezemer et. al.) is the high degree of configurability. Again, in a single-tenant environment, the application can be customized for the needs of the tenant. However, the same should be possible in a multitenant environment, to really have a multitenant application. This has larger implications, since the application should load many configuration files, for each tenant and its behavior should be the expected one regardless of the client that is using it. However, the authors mention that some multitenant applications require major configuration changes from two tenants thus making it necessary to keep running multiple versions of the application at the same time. Later in this chapter, the paper by Mietzner, et al. (2009) describes how to achieve such things with variability, and how to scale up and down, both deploy other versions of the application and un-deploy them as the tenant leaves. In this scenario, tenants are identified by the version of the application that they use, because some versions could support different configuration(s) files than others. The third characteristic is the shared application and database instance. The contrast begins with the single-tenant case where a tenant can run many instances of the application with different configurations. In the multitenant environment, normally one instance of the application should be deployed. It could be that many instances are deployed if the tenants require big configuration changes or the application needs to scale. The authors say that having the instance shared for all tenants, introduces complexity for development.

The paper (Bezemer et. al.) analyzes different parts of the multitenant application, namely, the authentication, the configuration and the database. The authentication plays an important role in multitenant applications, because all tenants' users use the same interface authentication mechanism to login on the application. Moreover, based on the user's credentials, the right access to resources is given, so any error in the authentication process will expose data of the other tenants which is a security issue. In contrast to the multi-user environment, the tenant specific authentication mechanism assures that the user belongs to the correct tenant and it also allows a user to be part of many organizations (read tenants). Next, the authors analyze the configuration which is mandatory for any multitenant application. Tenants can have different requirements from the application and besides having a different look, they can also require a different behavior. For instance if the multitenant application allows tenants to create events that users attend, while for some organizations the defaults would be enough, for other tenants, they may require special fields which the participants who subscribe to the event must fill-in before rsvp. Other changes the tenants may demand, besides layout and general configuration, are file I/O that sets different tenant-specific paths and workflow for ERP software. The last discussion is about the database which is also different than the database of a single-tenant and multi-user application. The difference lays in data isolation which is required in order to have a trustworthy application and convince tenants their data is safe in such an environment. Jacobs, and Aulbach (2007) state in their paper that the current database systems do not understand multitenancy, therefore, the multitenant application needs to be able to create tenants to the database and associate every record with the tenant id. Moreover, the application needs to adapt queries in order to only fetch data for a specific tenant, and to restrict the logged-in user from accessing data from other tenants.

There are several challenges in order to create a multitenant application and Bezemer et. al. mention them in the paper. The performance of the application is one of them. The application should not slow down if the number of tenants increase. Put it differently, the tenant should expect the application have at least the same performance as if it were deployed on his premises. Moreover, the application must make sure one tenant cannot use more resources than another or make another tenant to wait until he frees the used resources. The authors give also the example of achieving multitenancy using virtualization where each tenant is given the same configuration of the virtual machine where the application runs, thus the same resources and no tenant can use more that it has been configured with (Li, Liu, Li, Chen, 2008). The next major challenge is the scalability. In a single-tenant multi-user environment scalability was not a problem since most often the number of tenants would not vary, because the application is deployed per tenant (read organization).

Bezemer et. al. end the discussion about multitenancy with some conclusions regarding multitenancy, namely: deploying the application becomes easier and faster but with the expense of increasing the complexity in the code. Moreover, the authors state that there is not much research available on multitenancy which is perhaps the reason why there is not a clear definition on what it actually is and whether it is a “dream or a nightmare”.

3.3 Multitenancy database solutions

In order to determine the multitenancy types, a good understanding of the multitenancy databases is required. In this sub-chapter we will cover several multitenant database solutions that multitenancy software developers could consider when planning to build their multitenancy applications. In the following sub chapters, different multitenancy database solutions are introduced to overcome the limitations of DBMSs’ for dealing with large number of tables and columns, to solve the scaling issues and ways of achieving efficient multitenancy.

3.3.1 Schema mapping techniques

In the paper by Aulbach, Grust, Jacobs, Kemper, and Rittinger (2008), the authors introduce the “Chunk Folding” multitenancy schema-mapping technique. Chunk Folding folds together partitions of the logical tables into different physical multitenant tables and joins them as needed. The authors say that multitenancy occurs at the database layer. They argue this relating to Hamilton (2007) who stated that the servers of highly scalable web applications are often stateless.

For online business software data of multiple tenants is often consolidated into the same database to reduce the total cost of having multiple databases for each tenant. The authors say that it is often when developers choose to map many single-tenant logical schemas in the application to one multitenant physical schema in the database. Despite the fact that it is not easy to do this mapping, the benefits of consolidating hundreds of databases into one will save millions of dollars per year, say the authors. There is however a downside of multitenancy, which is the sharing of resources (Media Temple, 2007). Moreover, in the authors' view, there are cases when single tenant hosting is preferred over multitenancy, such applications like ERP and Financial software. Their reason is that this kind of software require more computational power, execute more complex operations, and often organizations prefer to have control over the updates.

The aim of Chunk Folding is to reduce the complexity of scaling a SQL database. The authors say that the performance begins to degrade when over about 50,000 tables are used on one server. An option to mitigate the performance downgrade, is to share the tables among tenants. While this could work in some cases, it limits the tenants to extend the application. Perhaps some of the clients use certain features that others are not, and using all tenants data in a shared table. Another approach that is used by many hosting providers, maps the logical tables into fixed generic structures, namely the Universal Pivot Tables (Florescu, Kossman, 1999). However, the authors do not fully agree with this approach either because in their view "the mapping techniques used by most hosted services today provide only limited support for extensibility and/or achieve only limited amounts of consolidation". Given the limitations of these three approaches, the authors propose a fourth approach, which they call Chunk Folding. Chunks are partitions of logical tables. The folding takes two or more chunks and puts them into different physical multitenant tables which are joined as needed.

Account	Profile Information	Chunk Table 1	Chunk Table 2

Figure 13. An example of Chunk Folding

As it can be seen from figure 13 above, the row has four chunks, the first one being the account, the second one, an extension for the account while the third and the fourth one are stored in differently shaped chunk tables. The main advantages of this approach are that chunk folding does not impose any limitation on extensibility, and it allows logical schema changes to occur while the database is online.

The authors give a brief description of the common schema-mapping techniques for multitenancy. They are summarized in the following table 10:

Schema-mapping technique	**Description**

Basic Layout	Adds a tenant / organization ID column (Tenant) to each table;
	Tables are shared among tenants;
	Advantage: very good consolidation;
	Disadvantage: no extensibility;

Private Table Layout	Each tenant has their own private tables; Advantage: Metadata is managed by the database; Disadvantage: moderate consolidation;
Extension Table Layout	Extensions are split into separate tables with a tenant / organization column and a row column (for reconstruction); Advantage: better consolidation and the performance is increased. Disadvantage: the number of tables grows proportionally with the number of tenants;
Universal Table Layout	The table contains a tenant ID, a table ID, an index ID for the table and the columns with the data for that specific index in the table that belongs to the tenant; Advantage: different tenants can extend the same table; Disadvantage: the rows become very wide and there are many null values;
Pivot Table Layout	Each table has a tenant, table, column, row ID and a value. The tables are grouped by the type of the value. Advantage: there are no null values stored; Disadvantage: more columns of meta-data than actual data.
Chunk Folding	Tables are partitioned into chunks; Chunks are folded together into different physical tables; Chunks are joined as needed; Advantage: metadata is stored more efficiently than the previous techniques; good performance;

Table 10. Schema-mapping techniques

The paper continues with several experiments that measure the efficiency of the Chunk Folding schema-mapping technique for implementing multitenancy on the top of a standard relational database.

3.3.2 Database-as-a-Service for the cloud

The paper by Curino, Jones, Popa, Malviya, Wu, Madden, Balakrishnan, Zeldovich (2011) coins the term “Relational Cloud”, a new transactional “database-as-a-service” DBaaS. The challenge to design the relational cloud to be efficient is to minimize the hardware resources. In their view, multitenancy is efficient when giving a set of databases and workloads, it can be determined what the best way is to serve them from a given set of machines. Relational Cloud stores the data belonging to different tenants within the same database server, but does not mix data of two different tenants into a common database or table. It uses existing unmodified DBMS engines in the back-end for query processing and node storage. Each backend node is running a single database server which is changing dynamically depending on the load variation. The backend nodes are communicating with a router which analyzes the queries. The SQL statements from clients are received by the router which decides which backend nodes to trigger for processing the query. The router is backed in the front-end nodes which communicate directly with the client nodes. Thus clients, from a trusted platform (private or secured) are running their applications which encrypt the queries and send them to the front-end nodes of the Relational Cloud. Then, the front-end nodes analyze the queries with the help of the router and decide which of the back-end nodes to trigger to perform the query. All these communication flows are illustrated in figure 14 below.

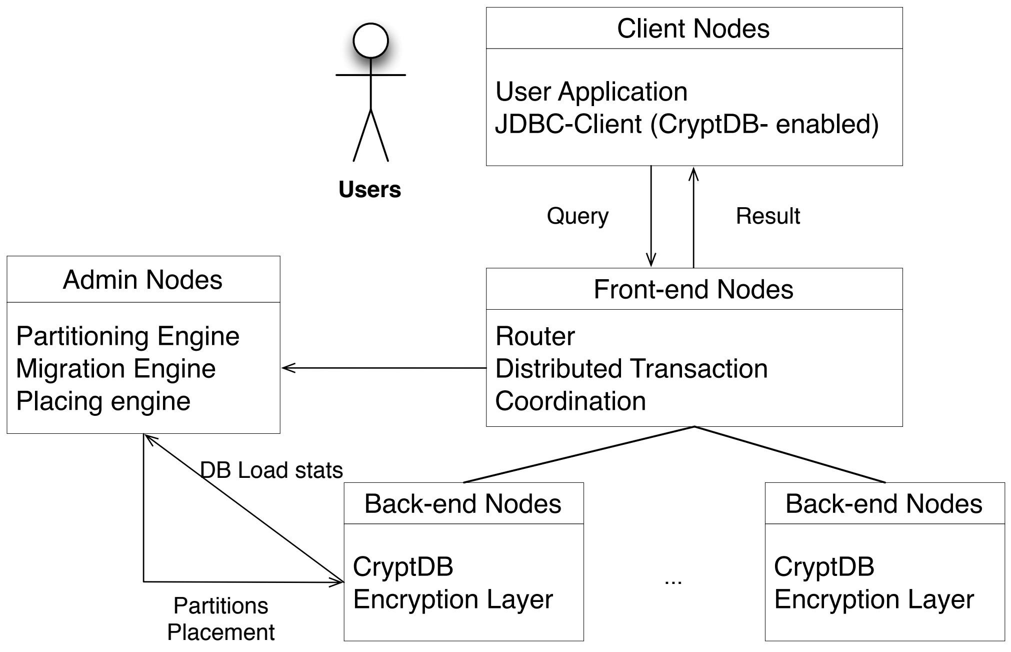


Figure 14. The architecture of Relational Cloud

The architecture of Relational Cloud permits to scale a single database to multiple machines and load balance the traffic on the back-end nodes. All these are possible because, as it is shown in figure 14, Relational Cloud is using database partitioning.

3.3.3 Database partitioning

The paper by Tsai, Shao, Huang and Bai (2010) describes a hybrid test database design for multitenancy applications that need to be customized in a SaaS architecture with two-layer database partitioning. The authors proposes a SaaS framework, introduce a three step scheme to embed recoverability on that framework, adds a feature for continuous testing which provides embedded testing support and finally examines the support for customization, recovery and continuous testing that is required for a multitenancy applications.

The paper begins with the statement that a multitenancy architecture needs to address a number of issues, namely:

- Scalability: the application should scale, heavy-computational tasks (such as delivering a newsletter) of one tenant should not affect the availability and the usability of the application for another tenant;
- Database partitioning and consistency: the queries must be executed fast, thus tenant specific data should be partitioned carefully. For instance, if the application requires a

certain type of data in each query, that data should be stored on the same table so that it can be fetched with one query;

- Fault-tolerant aspects: if the application fails or other processes (such as a background processing queue) fail, then the data of tenants should not be affected or destroyed.
- Security and fairness: data of different tenants must be isolated from each other and if the application supports prioritizing, tenants of the same priority should receive the same resources;
- Parallel processing: regardless of the tenant, tasks should be processed in parallel;
- Isolation: if a tenant requires a change, that change should not affect the other tenants;
- Performance and availability: backups and data migration for one tenant don't affect other tenants.

In order to scale a multitenant application, Tsai et. al. state that the number of tenants should increase proportional to the resources that are used. While this is the ideal case, there are two types of scaling, namely: scaling-up and scaling-out. By scaling-up, the authors mean to add resources to the application, such as CPU power, memory, additional storage disks. For database partitioning, the scale-up scenario offers the possibility to have more than one database partition on the same physical machine. By scaling-out, additional nodes are added in the cluster where the multitenant application runs, improving the availability of the application by increasing the redundancy. With regards to database partitioning, the scale-out scenario enables the creation of partitions on multiple physical machines.

Tsai et. al. suggest a new approach to database partitioning that is optimized for both writes and reads. They call it the P2, the two-layer partitioning model. Figure 15 below illustrates the two-layer partitioning model. The tenant level consists of partitions for each tenant. Because it is a horizontal partition, it is optimized for inserts and updates. Data from the tenant level is injected after in the chunk level. The chunk level is a vertical partition, that means it is optimized for reads. The central server's responsibility is to handle the queries and forward them to the right server that holds the partition which is being queried. It manages the storage keys that are located on the partition data servers. The data storage servers maintain a part of the keys and thus the time to get a response for a query is reduced. The authors build a "global index" on the central server to manage the storage keys. B-trees (Bayer, 1997) are used for the position of the local chunks. Moreover, the authors use a distributed hash table index in order to keep a uniform distribution between the servers.

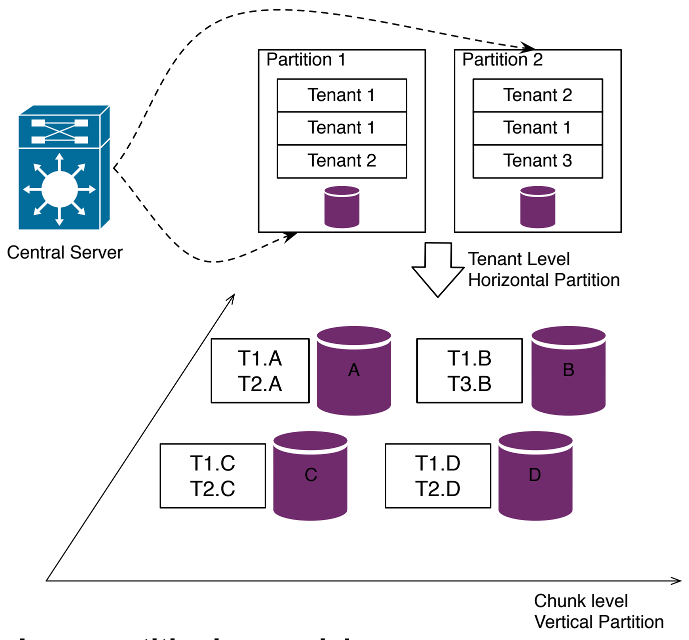


Figure 15. P2. Two-layer partitioning model

The four providers in figure 15 (Tsai et. al.) store the data partitions and the central server optimizes the global index queries for fast response times. In order to overcome the situation when one of the data providers is full of data while another has more free space, the authors are using a load balancing algorithm that can duplicate, migrate tenants across data partitions. The model the authors introduce in this paper avoids the problem of data redundancy (thus reducing the costs) and also the problem of query response times, having data distributed across partitions, while avoiding storing all data on the same server. They call it a two-layer index for the P2 model. The index relies on a distributed hash table to map keys to tenants' data. An example of how exactly the distributed hash table works is given in figure 16. Essentially, the central server holds the global index with the keys, the keys are hashed and then they point to the values.

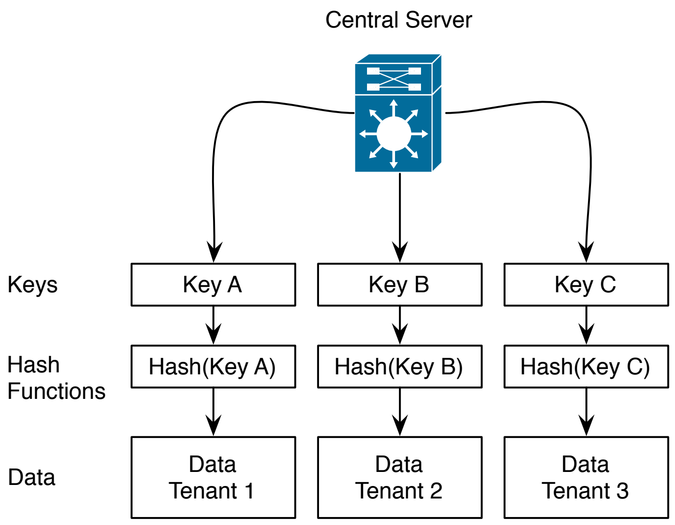


Figure 16. Example of a distributed hash table (DHT)

At the chunk partitioning level, the authors use a b-tree index to allocate tenants' data. There are two approaches in which this can be achieved, namely by allocating tenants' data periodically with fixed partitions, or by using flexible partitioning and re-partitioning.

The paper by Tsai et. al. continues with more technical details how to implement various algorithms (such as metadata recovery) and even gives testing results of their model. However, these are not summarized in this chapter since it is beyond the scope of getting an insight on data partitioning techniques.

3.4 Customization

The paper by Jansen, Houben, and Brinkkemper (2010) shows how customization and configuration is realized in multitenant web applications. They mention that there are three research areas that are related to customization in multitenant web applications, namely: multitenancy architectures, variability in software product lines and end-user personalization in web applications. The authors of this paper want to help developers of multitenant web applications with a catalog of information on how some multitenancy web applications are built.

In this paper, Jansen et. al. analyze five customization realization techniques (CRTs), namely: Model View Controller (MVC) customization and system customization. For MVC customization, three changes are examined: model changes, controller changes and view changes. For the system customization, they analyze connector changes and component changes. All these changes were identified on two case studies that were performed on multitenancy web applications. The findings of the case studies show that most of the changes happen in the controller, while the models and the views stayed the same. The advantage of designing the systems for multitenancy is that there is one deploy that serves the tenants. However, the two companies faced several performance issues, having to deal with queries that were too large. The solutions that were applied were to use dedicated servers for specific services and query optimization. Besides the controller changes, the second case study required interface changes in the sense that more than one interface was required to access the different systems. Moreover, the second web application required component changes for integrating with existing e-learning environments. The developers had to build an API to overcome integration limitations. In the end, the authors provide a list of customizations that they identified in the case studies they performed. Besides the customizations, the authors mentioned the problems the developers of the two multitenancy applications faced, which is helpful for other developers, to find solutions quicker to similar challenges.

3.5 Variability

Variability is what enables SaaS environments to serve a large number of tenants. Each tenant has its own requirements and a variable multitenant application can suit their needs. The following two papers presented next analyze variability for multitenant applications deployed in a SaaS environment.

In the paper by Kabbedijk and Jansen (2011) the authors give a pragmatic approach to variability for multitenant applications in SaaS environments and also introduce three architectural patterns that support variability in multitenant SaaS environments. By using variability, multitenant applications can offer more functionality than the basic one to the tenants by using modules. The authors use in their paper the definition of variability from Svahnberg, van Gurp, and Bosch (2005) which state that variability is “the ability of a software system or artefact to be efficiently extended, changed, customized or configured for use in a particular context”. In this paper, two different types of variability are identified, namely segment variability and tenant-oriented variability.

Kabbedijk and Jansen (2011) classify variability on three levels, namely, low, medium and high. The low level is the “look and feel” and it consists of changes on how the application looks. Using the Model View Controller architecture (MVC), (Krasner, Pope, 1988) the low level is consists of changes in the Views. The medium level, the “feature” consists of changes at the controller level in MVC. Finally, the “full” level, categorized as high, can influence all three levels of the MVC and are more specific. Using the previous classification, the authors focus in their paper only on low and medium variability levels.

	Number of customers Low	Number of customers High
Variability Low	Custom Software Solution	Standard Multitenant Solution
Variability High	Software Product Line Solution	Configurable Multitenant Solution

Table 11. Variability versus the number of customers

In table 11, the Kabbedijk et. al. present four different solutions that can be chosen, depending on the number of customers the application has and on the level of changes they require (the variability). The custom software solution is sufficient for software vendors with a low number of customers. However, if the number of customers increases, the standard multitenant solution is recommended. In the case of customers with specific requirements, the variability is high and the software product line solution is recommended if the application has a low number of customers. As in the previous case, if the number of customer increases and they all require a high level of variability, then the authors recommend the configurable multitenant solution.

The paper by Kabbedijk and Jansen (2009) introduces three variability patterns that the authors identified by conducting case studies: “customizable data views”, “module dependent menu”, and “pre/post update hooks”. Their conclusion is that the three variability patterns are helpful for companies to offer a better service to their customers. Moreover, the multitenant applications are maintainable and also scalable by using the modules.

The following paper, by Mietzner, et al. (2009), explains how to use explicit variability models to derive customization and deployment information for individual SaaS tenants. Like in the previous paper, the authors stress the importance of customization for SaaS applications. By customization, they mean that tenants are able to customize data fields used by the application and also define and modify workflows, using configuration metadata that is specific for each tenant. There are three patterns that developers can follow to deploy a multitenant application.

- The single instance pattern means that all tenants use the application through the same instance. The authors mention that a single instance means that the tenants use the same workflow, the same code and the same infrastructure.
- In the single configurable instance pattern, tenants still use the same instance of the application, however, the application is configurable per tenant. That means that at runtime, the application loads tenant specific metadata, configuration files or configuration entries in the database.

- The multiple instances pattern assumes that each tenant has its own instance of the service. It offers the highest flexibility in terms of customization. However, separate code needs to be deployed per tenant.

Mietzner et. al. mention that the two single instance patterns correspond to the idea of multitenancy. The third one although it is designed for many tenants, doesn't correspond to the definition of software multitenancy.

Multitenant SaaS applications are dependent on variability which can be of many types. The customer-driven variability consists of the changes that are made per tenant. Such changes include a custom logo on the tenant's webpage or a different template. The realization-driven variability consists of the ways in which a certain service that the multitenant application offers. The example the authors give explains clearly what this variability stands for: in the case of an online multitenancy business application, there are often services that many tenants use. One of them is the email service which instead of being a separate service per tenant, it could be a separate service for the application which each tenant uses. There is also a binding variability, which consists of knowing what parts of the application have already been deployed and for which tenants. The authors give an example of a tenant that requires high-availability and for whom a high-availability engine has been deployed. Then the authors say that operation costs are reduced when a new tenant that requires high-availability too, uses the same high-availability engine that the other tenant is using, instead of a newly deployed one.

The three variability models mentioned in the previous paragraph can be categorized as external and internal variability. Mietzner et. al. continue with the Orthogonal Variability Model (OVM) to describe variability. With OVM, tenants can choose non-functional properties, such as if his data is shared or not, and even GUI customization with configuration files. The paper continues with an example of a multitenancy application that is deployed for three tenants. Each tenant has its own configuration file and the settings vary. For instance, one of the clients requires data separation and multiple instances of the database and high-availability of the service. The authors deploy an instance of the application per tenant. They mention that once the tenant stops using the application, his instance needs to be un-deployed. The definition of multitenancy that we use in this paper states that multitenancy software serves all tenants from the same instance of the software. However, as the software becomes more complex, several components require special deploys being the reason why the example the authors give in the paper stands for achieving multitenancy through variability. It is also the reason why a special deploy needs to be made for that third tenant which requires multiple instances of the database and high-availability of the service.

There is one interesting point Mietzner et. al. make in this paper, namely optimization of the variability. They have noticed several patterns among tenants, such as: all tenants that require high-availability of the service, also required non-shared databases. The authors say that by capturing the configurations that tenants have used in the past, the costs of developing, testing and maintaining a multitenancy application can be reduced. By removing variants that are relevant only for a small number of tenants, the software provider can reduce costs, since such configurations do not pay off in the long run.

3.6 Service performance

The current sub-chapter deals with the performance of multitenant applications. According to the definition, a multitenant environment serves all tenants from the same instance of the application. However, tenants don’t all use the application in the same way, attention needs to be spent to assure that one tenant cannot eat all the resources available on the server, thus making the application unavailable for the other tenants. While virtualizing the environment could solve this problem by allocating a fixed amount of resources per tenant, in this research we are concerned about the software multitenancy where all tenants share the resources of the server that keeps up the instance of the application.

The paper by Li, Liu, Li and Chen (2008) introduces SPIN, a service performance isolation infrastructure in multitenancy environment. SPIN analyzes the performance of the machine where the multitenant application runs, gives anomaly reports, identifies aggressive tenants and has an auto-adaptive behavior to normalize the system and remove the tenants' negative impact on others. The paper introduces the requirements for achieving performance isolation in a multitenant environment, namely: isolation, efficiency, self-adaptability. By *isolation* the authors mean that a malicious tenant is not allowed to affect the others. Because all tenants use the same instance of the software, modules should be monitored in order to identify the tenants who try to abuse the resources available. By *efficiency* the infrastructure should make the most of the available resources available on the hosting platform. And by *self-adaptability* it means that because the multitenant application has a large number of tenants, the infrastructure should not expect human intervention to stop a malicious tenant. The infrastructure should be able to identify the malicious tenant by the actions he performs and then block or limit it.

The paper (Li et. al. 2008) continues with listing the three main functionalities of SPIN, namely: the anomaly detection functionality, the monitoring functionality, and the adaptation decision functionality:

- The *anomaly detection functionality* signals when an the system has reached an unstable state in the execution environment. The performance metrics are analyzed, especially those metrics that can seriously affect the state of the system over a short period of time.

- The second functionality described in this paper is the *monitoring functionality* which is responsible for keeping the logs of the application and its modules. Data from logs can be used for detecting any performance downgrades on the server and to see whether one tenant has tried to abuse the available resources.

- Finally, the *adaptation decision functionality* of the infrastructure applies the right solution for the detected malicious tenants. The system uses an "optimal moderation policy" to isolate the malicious tenant and to prevent him from interfacing with other tenants.

The paper (Li et. al. 2008) continues with a mathematic proof of concept for SPIN and after that they give details on how to implement it. SPIN was summarized in this paper because for some multitenant applications with a high number of tenants problems like performance isolation could appear and if the service provider is not using the right tools to monitor what is happening, a malicious tenant could easily load the server and make the application unavailable for the rest of the tenants. Even more, it is very complicated to find out the problem of why some tenants cannot access the application while for some other works if tools based on thresholds (like mentioned in the paper) are used because the application could still run between the allowed parameters. Thus the monitoring tools need to understand the presence of a tenant and need to differentiate traffic from different tenants.

3.7 From single-tenant to multitenant

While this paper aims to identify multitenancy types and associate them with tenants-interaction types, this sub-chapter presents the findings of Bezemer, Zaidman, and Platzbeecker (2010) that reengineered an industrial single-tenant application to a multitenancy. Creating a multitenant application does not mean one has to start from scratch with multitenancy in mind. It could be, like the case of Exact, (a Dutch-based software company that the Bezmer et. al. used in their case study) that the software was first created single-tenant and later reengineered to support multiple tenants. When the authors planned the reengineering, they had in mind several goals, namely: to share hardware resources, to make the application highly customizable, to share the application and database instances for easier maintenance, to migrate the application with minimum adjustments for the existing business logic, to make it transparent for the application developers that the application is multitenant and to separate the multitenant components for load balancing and monitoring services.

In order to reach the goal, Bezemer et. al. mention three components in the new application that they had to work on, namely: authentication, configuration and database. The paper continues with specific details on how the three components were implemented in the code of the project. However, these details are specific to that project and will not be listed in this chapter since for other projects written in other languages or using other database services they will differ. After the new multitenant software was implemented, the authors performed tests to check if the existing functionality was broken during the update. The interesting part is the lessons learned which could be applied in any project, namely: the lightweight reengineering approach, the importance of the architecture, the difficulty of automated reengineering, transparency for the end user and little effect for the developer. The lightweight reengineering approach was used in their migration. The authors concluded that they managed to do all changes in approximately 100 new lines of code. The fact that they managed to migrate the project with such a less effort, makes them confident that any company that needs to reengineer a project, should not see the process as a barrier in order to get multitenancy. The authors also note the importance of the architecture in which they say that the layered architecture was essential for keeping their work lightweight and for finishing their work quickly and in an efficient way. Moreover, the authors discovered that automated reengineering was difficult. They agree it was easy to achieve their goal, but they mention that it is very difficult to automate the whole process. The reason is that each application has its own characteristics, and in order to reengineer a whole application from single-tenant to multitenant requires architectural and domain knowledge.

Despite the fact that the case study presented in this sub-chapter is not complete (no workflow configuration was implemented, tenants cannot be automatically created, and automated unit tests and integration tests for the multitenancy environment were missing), the summary gives a good insight for any developer that needs to reengineer a single-tenant application to a multitenant one.

3.8 Security

Security plays an important role in multitenant applications. Tenants that were used to run the application on their premises (single-tenant applications) need to be convinced that their data will be safe in a shared environment where data from other tenants is present. In the article by Wainewright (2012), he acknowledges the real problem of security concerns for multitenant environments. Wainewright states that: “Intuitively, we feel that if our data is physically on the same computer system — or, in a fully multitenant stack, actually in the same database — then there has to be a higher risk of data being exposed. Either inadvertently, when for example a software bug or system mulfunction gives access to a user of another system on the same shared infrastructure. Or maliciously, when someone exploits some weakness in the architecture to gain illicit access to data”. While theoretically data exposure should not happen, unfortunately, it just happened to Github in March 2012 (Preston-Werner, 2012 and Homakov, 2012) when a user was able to add his public key to the credentials table of a project and to demonstrate his success, and published a file inside the Git repository. Translated in multitenancy terms: a tenant got access to other tenant's data and was able to alter it, proven by the fact that he uploaded a file in the other tenant's data. While there was no damage after the investigations, the question whether storing your data in a multitenant environment remains valid.

Returning to the article written by Wainewright, he mentions that there are two main risks to be aware of, namely: the virtualized environments and the exposure of data to other users. The risk of virtualized environments comes from using one physical machine that runs many virtual machines inside it. This is one approach to achieve a multitenant environment, where each application is deployed on one machine, and all the virtual are connected to one database. The risk lays in improper configuration of the virtual machines and the possibility of one virtual machine to monitor the state of the others. A study by Gartner (MacDonald, 2010) says that throughout 2012, 60 percent of virtualized servers will be less secure than the physical servers they replace. The article does mention the fact that not virtualization is insecure, but the people that are configuring it are not trained well enough or lack the right tools and this is why, theoretically, multitenancy environments that use virtualization are still considered insecure.

The second risk Wainewright mentions in his article is that data will be exposed to other users if there are bugs in the implementation. Despite the fact that it happened and the author gives the example of the banking websites where users who logged in got access to other people's accounts, the author mentions that such errors have not stopped users from using the applications.

Another paper, by Parlione and Charlesworth (2008) is concerned about multitenancy security issues and introduces an approach for securing a multitenant Java application with the open-source Spring Security (Wiesner, 2011) framework, combined with Apache Directory Server. The authors mention that the two main security concerns for multitenant applications are authentication and authorization. They offer solutions to these concerns, however because the solutions are coupled to specific technologies, we point the reader to the actual paper.

3.9 Chapter summary

In the current chapter we reviewed literature on various multitenancy topics. We first introduced the terms *online business software* and *multitenancy* with references to existing literature. Then we analyzed and summarized papers on different topics, namely: papers that offer solutions for multitenancy database scalability problems, that analyze customization, and variability, that offer a service performance analysis tool, that offer insights on how to reengineer an existing application to multitenancy and in the end papers that study security for multitenancy applications. Table 12 summarizes our findings in this chapter and shows how they are are related to the other chapters in the research.

Concepts	Findings and relevance
Schema mapping techniques	The Chunk Folding multitenancy schema-mapping technique was introduced. It will be used as part of RQ 2 on patterns mitigation techniques, for those patterns for which database scalability is a liability.
Relational cloud	It introduces a new transactional “database-as-a-service” DBaaS. It will be used as solution for patterns with database scalability issues, part of PATTERN MITIGATION concept in RQ 2.
Database partitioning	The paper offers a hybrid test database design for multitenant applications that need to be customized in a SaaS architecture. It will be used as part of RQ 2 on patterns mitigation techniques, for those patterns for which database scalability is a liability.
Customization	Five customization realization techniques: Model View Controller (MVC) customization, and system customization (connector and component changes). We will use the customization techniques to determine the multitenancy implementation patterns that is part of RQ 1, IMPLEMENTATION PATTERNS concept.
Variability	Three variability patterns: • customizable data views • module dependent menu • pre/post update hooks We will use the variability patterns to determine the multitenancy implementation patterns that is part of RQ 1, IMPLEMENTATION PATTERNS concept.
Service performance	SPIN was introduced, a service performance isolation infrastructure in multitenancy environments. It will be used as part of the PATTERN MITIGATION concept in RQ 2 in chapter 5.
Security	The following security issues are discussed: • Storing data of multiple tenants in the same database; • Data exposure • Virtualized environments • Authentication and authorization These issues are used as part of RQ1 IMPLEMENTATION PATTERNS concept and RQ2 PATTERN MITIGATION concept.

Table 12. Summary of the literature topics discussed and their connection with the research questions

In table 12, we listed the topics that were discussed in this chapter. For each topic we offered a short description and how they contribute to answering part of the research subquestions.

After analyzing the literature on multitenancy definitions, our research will be based on the following definition: *Multitenancy is a principle in software architecture where a single instance of the software runs on a server, serving multiple client organizations (tenants).*

The following characteristics of multitenancy are used in this research based on literature study, namely: sharing of hardware resources, high degree of configurability, performance of the application, and degree of scalability;

Articles	Concepts				
	MT database solutions	Customization	Variability	Service Performance	Security
Aulbach, Grust, Jacobs, Kemper, and Rittinger (2008)	X				
Florescu, Kossman (1999)	X				
Curino, Jones, Popa, Malviya, Wu, Madden, Balakrishnan, Zeldovich (2011)	X			X	
Tsai, Shao, Huang and Bai (2010)	X		X	X	
Jansen, Houben, and Brinkkemper (2010)		X			
Kabbedijk and Jansen (2011)			X		
Svahnberg, van Gurp, and Bosch (2005)			X		
Mietzner, et al. (2009)			X		
Li, Liu, Li and Chen (2008)				X	
Bezemer, Zaidman, and Platzbeecker (2010)		X			
Wainewright (2012)					X
Preston-Werner, (2012)					X
Homakov, (2012)					X
MacDonald, 2010					X
Parlione and Charlesworth (2008)					X

Table 13. Concept Matrix

In table 13 we illustrate the concept matrix (Webster and Watson, 2002). The concepts in the previous table, together with the findings in table 12 form the base for the multitenancy implementation patterns analysis that is performed in chapter 5.

Chapter 4 - Case Studies

In this chapter we will determine four multitenancy implementation patterns. Two of them are determined based on interviews with multitenant experts and the other two are determined based on literature study. The two implementation patterns that result through literature study represent Salesforce and GitHub. The other two implementation patterns which are based on interviews are from two Dutch companies that have developed and use online multitenant business software. They are Ledensite and AFAS and the transcription of the interviews is in Appendix.

The chapter is structured as follows: in the first part we will introduce Ledensite, analyze the interview and determine the multitenant implementation characteristics that it uses. Next we introduce AFAS Online, analyze the interview and determine the multitenancy implementation characteristic. Finally, the two implementation characteristics will be compared based on criteria such as programming language, database, and the results will be structured in a table.

The following two subchapters are based on interviews with experts in the field of multitenancy. Therefore, figure 17 shows their mapping on the PDD from figure 8.

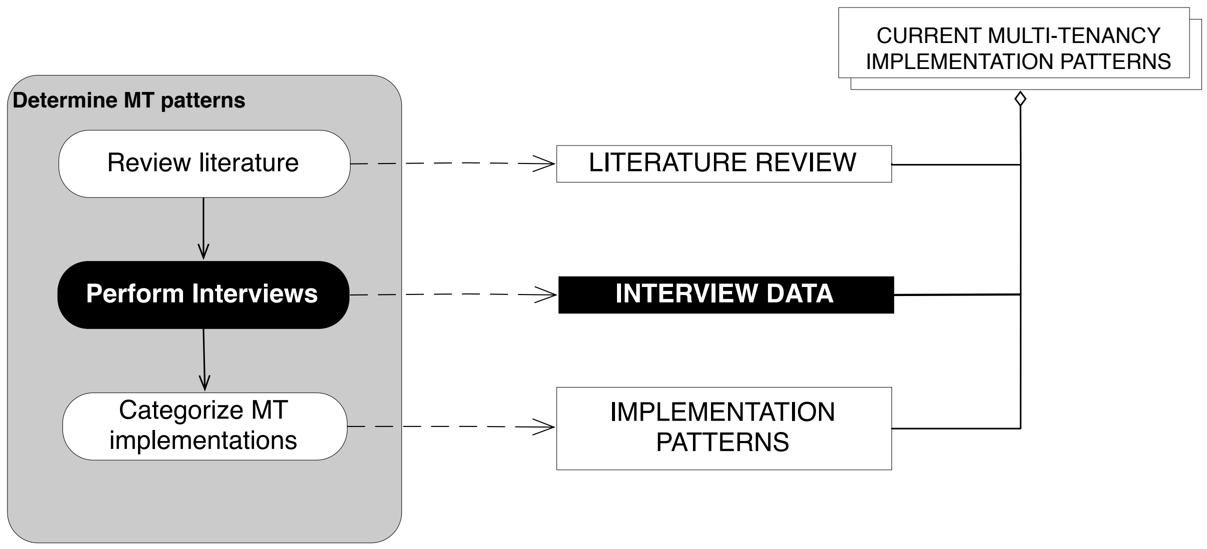
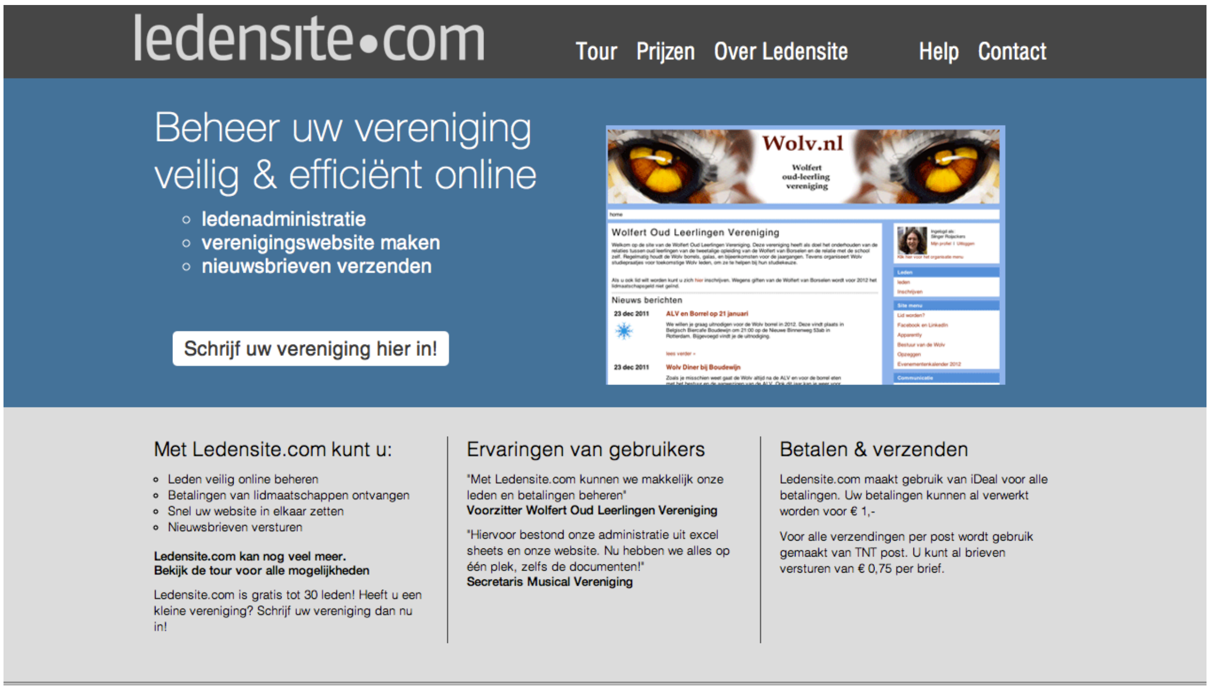


Figure 17. The following two sub-chapters mapped on the PDD from figure 8

4.1 Ledensite

Ledensite is a website for associations. It is meant to fully administer members of small, medium and large associations. Mainly, it takes over the administrative tasks from the association itself, such as invoices, newsletters, events and also communication between the members. Currently Ledensite has 6 tenants and they are all served from one instance of the software (one deploy). Ledensite is in the startup phase, the idea was born 3 years ago but the actual implementation started around 2 years ago.

Figure 18 shows how Ledensite’s main page looks like in the browser and figure 19 shows how the website looks like for one of their tenants.



Beheer uw vereniging veilig & efficiënt online

- ledenadministratie
- verenigingswebsite maken
- nieuwsbrieven verzenden

Met Ledensite.com kunt u:

- Leden veilig online beheren
- Betalingen van lidmaatschappen ontvangen
- Snel uw website in elkaar zetten
- Nieuwsbrieven versturen

Ledensite.com kan nog veel meer.
Bekijk de tour voor alle mogelijkheden

Ledensite.com is gratis tot 30 leden! Heeft u een kleine vereniging? Schrijf uw vereniging dan nu in!

Ervaringen van gebruikers

"Met Ledensite.com kunnen we makkelijk onze leden en betalingen beheren"
Voorzitter Wolfert Oud Leerlingen Vereniging

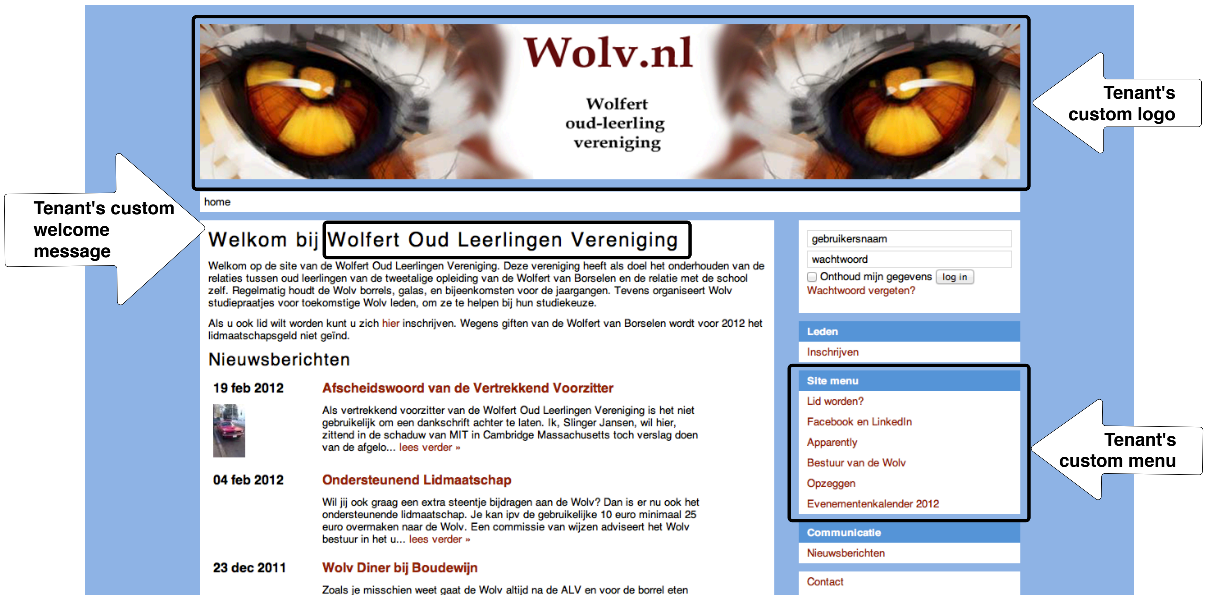
"Hiervoor bestond onze administratie uit excel sheets en onze website. Nu hebben we alles op één plek, zelfs de documenten!"
Secretaris Musical Vereniging

Betalen & verzenden

Ledensite.com maakt gebruik van iDeal voor alle betalingen. Uw betalingen kunnen al verwerkt worden voor € 1,-

Voor alle verzendingen per post wordt gebruik gemaakt van TNT post. U kunt al brieven versturen van € 0,75 per brief.

Figure 18. The main page for Ledensite.com



Welkom bij Wolfert Oud Leerlingen Vereniging

Welkom op de site van de Wolfert Oud Leerlingen Vereniging. Deze vereniging heeft als doel het onderhouden van de relaties tussen oud leerlingen van de tweetalige opleiding van de Wolfert van Borselen en de relatie met de school zelf. Regelmatig houdt de Wolv borrels, galas, en bijeenkomsten voor de jaargangen. Tevens organiseert Wolv studiepraatjes voor toekomstige Wolv leden, om ze te helpen bij hun studiekeuze.

Als u ook lid wilt worden kunt u zich hier inschrijven. Wegens giften van de Wolfert van Borselen wordt voor 2012 het lidmaatschapsgeld niet geïnd.

Nieuwsberichten

19 feb 2012 **Afscheidswoord van de Vertrekkend Voorzitter**

Als vertrekkend voorzitter van de Wolfert Oud Leerlingen Vereniging is het niet gebruikelijk om een dankschrift achter te laten. Ik, Slinger Jansen, wil hier, zittend in de schaduw van MIT in Cambridge Massachusetts toch verslag doen van de afgelo... lees verder »

04 feb 2012 **Ondersteunend Lidmaatschap**

Wil jij ook graag een extra steentje bijdragen aan de Wolv? Dan is er nu ook het ondersteunende lidmaatschap. Je kan ipv de gebruikelijke 10 euro minimaal 25 euro overmaken naar de Wolv. Een commissie van wijzen adviseert het Wolv bestuur in het u... lees verder »

23 dec 2011 **Wolv Diner bij Boudewijn**

Zoals je misschien weet gaat de Wolv altijd na de ALV en voor de borrel eten

Figure 19. Multitenancy aspects visible on the main page of one tenant using Ledensite.com

In figure 19 above, we observe several multitenancy aspects that are visible on the front page of an association that is using Ledensite. The multitenancy elements we notice are the tenants' custom logo, custom main menu and custom welcome message with introduction text.

The interview was held with Peter Duijnste, the lead developer at Ledensite.

Tenants on Ledensite are, as stated above, organizations. The users are the actual members from the organizations. When an organization registers, it receives an own URL (such as: http://organization-name.ledensite.com). The custom url is used by all the members of the organization to login and interact. Besides the url, organizations can customize their own logo on the website, and even how the site is structured, such as where the menu is placed, what color the website should have, and also define a custom front-page.

In terms of software requirements, Ledensite is a website and is accessible over the Internet. Tenants run it in the browser and they are not restricted to operating system. The whole website is HTML4 and CSS. It requires an Internet connection in order to work. The software is currently deployed on one server and serves all tenants.

The application is build in Ruby on Rails and the database that it uses is MySQL. One interesting remark Peter has made about the technology stack is the following:

"For this project and the others projects that I've been working on, the database has never been the bottleneck. The bottleneck has always been Rails that has more of a problem of serving large volumes than database queries."

When asked about an average number of database queries per request, the answer was that customers tend to ignore the default fields and make a huge list of custom fields. They were expecting sort of expecting 2,3,4 custom fields but it turned out that standard details like name, address, were not so interesting and they tend to make more custom fields, and sometimes they even duplicate the default fields because they want to have everything in the custom ones. As stated above, an organization (tenant) can customize the fields the members can fill-in and these customizations are translated in expensive (read: large number of) queries in the database. Therefore, if a profile page is visited, this could generate up to 15 queries, but the average number of queries is around 5. However there are pages, such as the ones that display the list of members together with some of their details, where the n+1 problem exists, so for 50 profiles in the list, at least 51 queries will be executed.

In terms of security, there have not been any attacks and no data was lost. The tenants are not concerned that their data is stored in the same database with the others. The website runs over HTTP with no encryption and most of the information that is stored in the database is not encrypted. The only encryption the database uses is for sensitive data, like account information, such as the password. The rest of the data is mainly open on the website (news page, events) so it is unencrypted.

In terms of API, Ledensite provides an API for one organization. The organization allows their members to login on their website and at the same time be logged in on their account on Ledensite.

Besides the customization options mentioned in the first paragraphs of the chapter, for some of the resources the tenants can define thier own fields. Also, the layout can be customized through templating. Tenants can install HTML templates, Ledensite uses Liquid templating engine and allows them to define a few custom tags for their organizations. There is a list of standard settings, such as where to place the user menu, where to place the login box and there are liquid tags for them. All the metadata is stored in the database. One feature that is not yet implemented but the some tenants asked for it is per-tenant localization that would enable tenants to call their members 'members' or 'participants' which is important for the views of the website. In order to achieve per-tenant localization, the current default yml file will be moved to the database so that each tenant can customize it the way he wants or fall-back to the default. Other meta-information that tenants can customize are the keywords and meta-description of the individual page and also tenants can configure custom trackers such as Google Analytics. All these metadata is stored in the database.

In order to find out how tenants use the website and maybe optimize it for a better experience, besides using Google Analytics, they are also doing internal log analysis. They run the Rails logs through a profiler to see if there are problems that are causing big delays of if the database connection is slow for some pages. They do not use A/B testing because in the time it has been online there were only 2 requests from unknown organizations through the contact form while the rest were convinced through direct selling rather than waiting for them to come and analyze how they interact with the front page. When asked if they did optimize the website based on the results from Google Analytics or the results of the profiler that analyzed the Rails logs, they did discover some n+1 (large number of queries) problems and moved those actions to separate rake tasks. One such action was the CSV import feature which originally was in real-time, that means the website was waiting (blocking) until the server processed the imported CSV file. Because of the large number of custom fields that some of the tenants added, the CSV import action became very slow and it was moved to a background job. Now when a CSV file is imported, the website only uploads it and triggers the background job to start processing it while it immediately becomes responsive to the user.

About the authentication, it is all software based. Ledensite identifies which tenant it is based on the custom URL. When a user wants to login, the login form asks for the email/username and password. Together with the url that sends the request, the web server knows how to identify the user, and to which tenant it belongs. Besides the web-based authentication, it offers an API that 3rd parties can use to authenticate the users. Moreover, the API for 3rd party authentication introduced another issue that is the backwards compatibility. Before the API, just upgrading the website would not affect the tenants in terms of connectivity. However, the API needs to be version maintained that means once a 3rd party starts implementing the API, there are already restrictions set with regards to future developments since they need to keep the previous version(s) of the API stable.

When asked about the infrastructure, Ledensite currently runs on a server inside a VPS, a virtual isolated space. However, as it will grow, in the future once they get to the point of having enough tenants to make the transition to cloud services from Amazon or Rackspace cost-efficient, they are going to move it there. Currently they don’t use content distributed networks (CDNs) but when they will migrate to one of the big cloud providers they will start using CDNs as well.

Within Ledensite, the organization is the base entity and everything is inherited from there. Currently the website uses a flat structure, that means tenants are individually and separated from each-other and their members cannot connect or interact with each-other except when they have two accounts on two different organizations. One of the requirements was to enable a hierarchical structure of the tenants, enabling tenants to become administrator of other tenants (such as when an organization manages a smaller organization). However, at the time of the interview, it was not implemented yet.

In the current chapter we summarized the main findings after the interview. Refer to Appendix 1 for the full transcription of the interview. The interview questions were structured on the six categories that were introduced in chapter 2.5, namely: general, security, architecture and API, database, authentication and authorization and accessibility. The information we gathered during the interview is used to partially answer RQ 1 where the implementation patterns are extracted. We use the findings from the interview to create a multitenancy implementation pattern with general components.

Item	Comments
Central entity	The organization;
Programming Language	Ruby on Rails, open-source framework based on Ruby language; Scalability issues;
Database	MySQL;
Customizations	URL, logo, template, trackers, page meta-keys;
API	Yes, login only;

Accessibility	Internet, connection required;
	
	Browser only;

3rd party access	No. There is only the login functionality enabled for one tenant but other than logging users in, no 3rd parties can access tenants data on their behalf.

Table 14. Multitenancy implementation characteristics for Ledensite

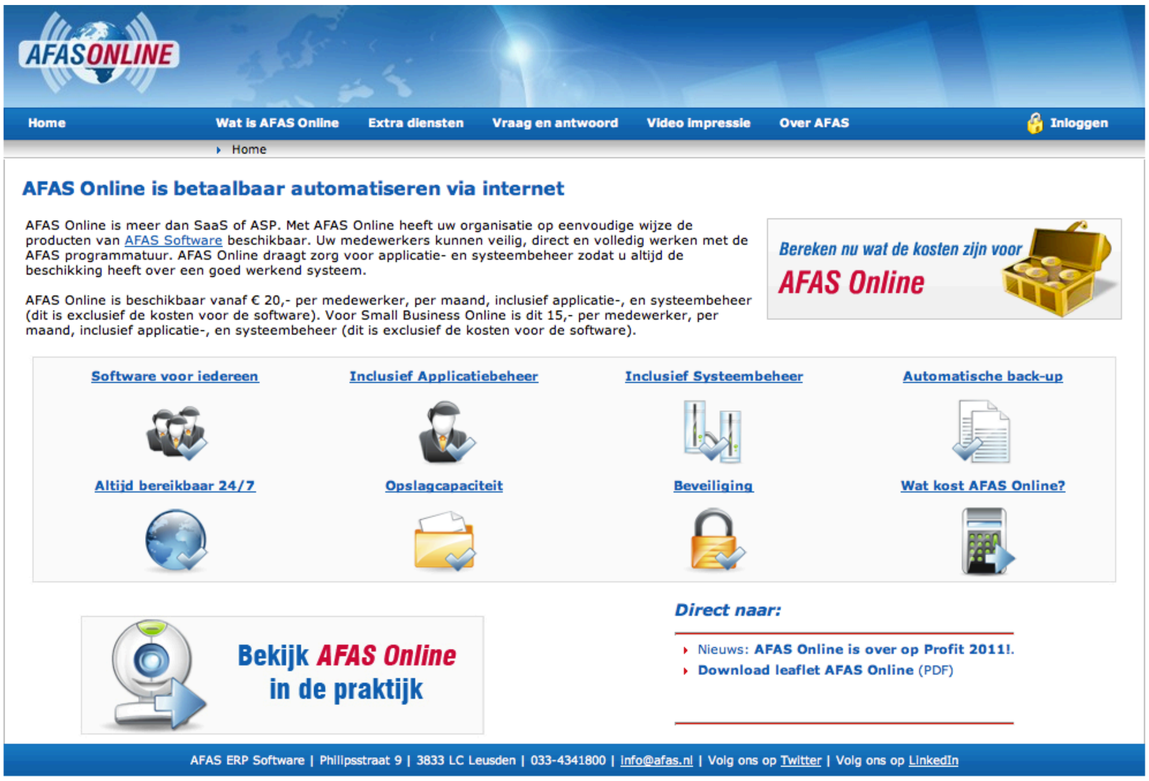
Table 14 above summarizes the implementation characteristics that were detected for Ledensite after the interview.

4.2 AFAS Online

The interview was held with Jeroen van Stokkum, the leader of the internal IT department, and the leader of the SaaS platform at AFAS. He works at the company for over 10 years.

Some facts about AFAS Online:

- 130.000 users currently have access.
- 15.000 unique users per month.
- 2.500/3.000 users concurrent during business hours.
- 40 physical servers running 120 virtual servers all running Windows Server 2008 R2.
- 40 TB storage.
- 12.500 databases (MS SQL).
- From 0 to >4000 customers in 3 years.



AFAS Online is betaalbaar automatiseren via internet

AFAS Online is meer dan SaaS of ASP. Met AFAS Online heeft uw organisatie op eenvoudige wijze de producten van AFAS Software beschikbaar. Uw medewerkers kunnen veilig, direct en volledig werken met de AFAS programmatuur. AFAS Online draagt zorg voor applicatie- en systeembeheer zodat u altijd de beschikking heeft over een goed werkend systeem.

AFAS Online is beschikbaar vanaf € 20,- per medewerker, per maand, inclusief applicatie-, en systeembeheer (dit is exclusief de kosten voor de software). Voor Small Business Online is dit 15,- per medewerker, per maand, inclusief applicatie-, en systeembeheer (dit is exclusief de kosten voor de software).

Nieuws: AFAS Online is over op Profit 2011!.
Download leaflet AFAS Online (PDF)

AFAS ERP Software | Philipsstraat 9 | 3833 LC Leusden | 033-4341800 | info@afas.nl | Volg ons op Twitter | Volg ons op LinkedIn

Figure 20. The main page of AFAS Online

Currently AFAS Profit, the main product of the company can be downloaded and installed on customers premises, but the same version is deployed in the cloud and is available to tenants who want to access it online, over the internet. The company has in works a new product, AFAS Profit 2016 which is completely built from scratch with cloud in mind. The current version started as a multi-user application that was not meant to be deployed in the cloud and changed later to make the transition to the cloud possible. The new product the company is building is also completely designed with multitenancy in mind. The interview was about the current deployed version of AFAS Online (figure 20) which is multitenant as well. Tenants are the companies that buy a license to use the product.

Business	Online	On premise
Using a part of the ERP suite	57%	43%
Using the full ERP suite	20%	80%
Overall	40%	60%

Table 15. The percentage of online versus on premise for AFAS Profit

Currently, the percentage of online usage is less than the one for on premise (table 15). However, they expect the numbers to change in the future more in the favor of online software.

The application is completely available over the internet. An internet connection is needed to use AFAS Online. It was developed using the Microsoft .Net framework and it uses Microsoft SQL Server for databases.

The application is customizable. Tenants can customize how the application looks, how the tables are displayed, which columns should be sorted in which way, what columns should be displayed in certain views and even the color of the column can be configured. Moreover, custom workflows can be defined. For instance, AFAS, the company uses defines workflows that are used within the company. Some examples of workflows are holiday booker, and an internal ticketing system. They mentioned that the workflows module is very flexible.

In terms of security, they have not experienced any successful attacks. They are very strict with the security of their product and therefore they test it and do penetration tests with external companies, and they run automated tests to check for known vulnerabilities. They said that because of these tests, they are constantly under attack of their own company. Since 100% bug-free/100% safe software does not exist and threat levels raise every day, AFAS is aware of the fact that it’s not the question if, but more likely when a (partly) successful hack will take place. Of course, there are all kinds of safeguards in place to prevent this and to keep damage as small possible.

AFAS Online is deployed at the professional datacenter Leaseweb in Haarlem. They lease different machines from them, using the Infrastructure as a Service. They get complete machines with the operating system (OS) on them and they are further responsible for the machines. AFAS has their own private cloud. They made agreements on what type of hardware and servers the virtual machines they use are installed and also on how many virtual machines can be run on one physical machine.

AFAS Online has some scalability issues. While the company is trying to predict when the clients will use more the application, such as processing the payroll because next week is the pay-slip week. They acknowledged that the scalability problems are their fault that they had to scale up before it happened. Their platform is ok, however sometimes it happened because the hardware under it was not. What they noticed was that sometimes there were differences in the processing power that was because there were other types of CPUs installed on the physical machines with less caching, or other times the storage was not growing fast enough. These caused the scalability issues when the demand on the website was high. However, they did not experience any data loss.

In terms of storage, they see all data as equal important. Moreover, they use highly available and expensive storage.

When asked about API, AFAS Online offers XML webservices for SOAP calls. If customers have an API license, then can use the API service. The webserver is developed to go through the usual controls, but developers can tune the way it handles the controls, such as for instance changing a field that in the original web service was not required to required. The API permits alto to customize what kind of data is returned, and customized views can be designed as well, for both SQL and the data that is seen on the screen. Most of the requirements the clients have asked for are available through the API.

AFAS developed an own batch service to process jobs that take a long time. For instance, the payroll job is using the built-in queuing system. The queuing system can pick a number of jobs at the same time and handle it and notify the customer when the job is ready.

In terms of database, AFAS Online is designed to use a database for every customer. If a customer requires a test database or if they have another company, AFAS will create a new database instance for them. This explains why they currently have around 4,000 customers but over 12,000 databases. So for instance if there is an accountant that does the administration for 80 companies, there would be 80 databases. They say the approach is very flexible because new databases can easily be added.

“The software is designed to use a database for every customer. That is very flexible, because new databases can be easily added.”

Because the whole application is built on Microsoft technology, AFAS Online uses Microsoft SQL database server. Given the large number of databases they have, they only have a replica, so there is no master / slave, which makes the database a single point of failure. They can load-balance web services and other services but they said they cannot load-balance the database with Microsoft SQL. From the performance point of view, it is not possible to load-balance it. They said that for failover they could do clustering but for performance when the customers become really big they cannot spread it over more services. They mentioned the single point of failure is a big issue that becomes more relevant every day. They even agreed that MySQL is more flexible than MicrosoftSQL because with MySQL you can use masters and slaves and one for read and one for write which is highly tunable. However, given the limitations of Microsoft SQL, they are happy to be a client of Microsoft, they have good contacts with the company and they are always informed on the new developments and they always use the latest developments, taking advantage of all the new features.

AFAS deals with tenants metadata by synchronizing it to the database of the customer. Except from the database they have some other files that they view as metadata. These files are maintained so the tenants can always go back to the default.

When asked about content delivery networks, they were in some tests with Akamai for the kind of connection SSL they offer, but not for the content delivery. They were not convinced yet because from a security point, AFAS does everything over SSL and if they decide to use Akamai, they would have to give them their own SSL keys to really have benefit from it. However, they said that because most of the tenants are from Netherlands, the connection is good and they do not quite need it yet, but they will keep an eye on it.

The authentication on the website is done as follows: after the user signs up, they receive an email with the login credentials. For authentication AFAS has different types: HTML authentication, and Microsoft forms management gateway. For the RDP (remote desktop protocol) sessions over SSL, they use also HTML authentication. At the moment the interview was taken they only used username & password but they plan in the future to use multi-factor authentication, such as tokens, one-time passwords and even enable single sign-on federation features for customers, that means one-time sign on and they have access to their own environment.

When asked about data sharing between tenants, Jeroen answered that Profit has a complete authorization model in itself and they don't offer anything else than the application online. They see data that a company shares with other tenants as data of other tenants within the company. Therefore, the application allows 3rd parties to access data on tenants behalf. They have a "collaborative license" which they use for instance for other tenants. A tenant can delegate an accountant (another tenant) to do the financial administration for them. In order to do that, they simply add the other tenant (the accountant for instance) number and give them access to the application. The accountant uses the same login but after logging in he sees the new database in his list and can start working on it.

In terms of accessibility, the software can be used on Linux and OSX too, but because of using the RDP sessions over SSL, clients need to install a tool that enables the Microsoft Remote Desktop Gateway which terminates the SSL session for the RDP sessions. The clients need RDP 6.1 at least for support for the SSL connection. For the printing features AFAS Online provides, the clients need .Net 3.5. The RDP protocol 6.1 offers the terminal server Easy Print driver which makes it easy to print based on the .NET framework, so AFAS does not need to install printer drivers on their side, allowing the customers to use what printer they have and the software client will tunnel the data to the right printer server.

Normally the software is available from everywhere where is an Internet connection, but they had some tenants who required them to restrict access only within their company so they added IP restrictions for them.

In the current chapter we summarized the main findings after the interview. Refer to Appendix 2 for the full transcription of the interview. The interview questions were structured on the six categories that were introduced in chapter 2.5, namely: general, security, architecture and API, database, authentication and authorization and accessibility. The information we gathered during the interview is used to partially answer RQ 1 where the implementation patterns are extracted. We use the findings from the interview to create a multitenancy implementation pattern with general components.

Item	Comments
Central entity	The company;
Programming Language	.Net, Microsoft - proprietary;
Database	Microsoft SQL - proprietary;
Customizations	Views, can be created and adjusted; Tables that are displayed on the screen can be customized;
API	Yes, sold as a separate license; Allows creating and customizing views;
Accessibility	Internet, connection required; Clients need RDP over SSL to use the full AFAS Online functionality. Through the browser only some small tasks can be done for Employee/Manager Self Service (ESS/MSS) and mostly seek stuff, not alter data.
3rd party acceess	Yes, 3rd parties (that could be other tenants) can access data on tenants behalf.

Table 16. Multitenancy implementation characteristics for AFAS Online.

Table 16 above summarizes the implementation characteristics that were detected for AFAS Online after the interview.

Implementation characteristics discussion - Ledensite and AFAS Online

Here we bring together the multitenancy implementation patterns from the two companies that were interviewed (table 17).

Item	AFAS Online
Central entity	**AFAS Online:** The company;
	Ledensite: The organization;
Programming Language	**AFAS Online:** .Net, Microsoft - proprietary;
	Ledensite: Ruby on Rails, open-source framework based on Ruby language; Scalability issues;
Database	**AFAS Online:** Microsoft SQL - proprietary;
	Ledensite: MySQL - open source
Customizations	**AFAS Online:** Views, can be created and adjusted; Tables that are displayed on the screen can be customized;
	Ledensite: URL, logo, template, trackers, page meta-keys;
API	**AFAS Online:** Yes, sold as a separate license; Allows creating and customizing views;
	Ledensite: Yes, login only;
Accessibility	**AFAS Online:** Internet, connection required; Browser & RDP over SSL.

	Ledensite:
	Internet, connection required;
	Browser only;

Freemium service	**AFAS Online:** No. Demo account available but everything is paid.
	Ledensite: Yes.
3rd party acceess	**AFAS Online:** Yes, 3rd parties (that could be other tenants) can access data on tenants behalf.
	Ledensite: No. There is only the login functionality enabled for one tenant but other than logging users in, no 3rd parties can access tenants data on their behalf.

Table 17. Multitenancy implementation characteristics for AFAS Online and Ledensite.com

Below we will point out some remarks for the data in the previous chapter, that could be useful advice for any programmer willing to start a multitenant application:

A first remark would be that while AFAS Online is a mature product, it has already an API built-in for developers and offers 3rd parties access to other tenants resources on their behalf. While Ledensite does not currently have these options, it is still considering them as the product matures and tenants requirements keep coming, as it happened with AFAS too. If the customers would not have asked for an API with this and that functionality, then probably AFAS Online would not have an API available. This may happen to Ledensite too, so the fact that currently the API is limited should not be seen as a limitation of the web framework or the database.

In terms of scalability, even the biggest websites have such issues. AFAS Online has scalability problems because they failed to scale when the demand was up. Ledensite currently does not have such a high volume of traffic but given the remark that Peter has already made during the interview, it will certainly run into scalability problems when traffic increases. So, scalability is important, especially for a multitenant application where tenants use the same “instance” of the application.

Also interesting to mention is that both applications are only available over the Internet. So if there is no internet connection the applications cannot be accessed. It is interesting to follow-up in the future to see if they will develop mobile applications that allow offline working or start using HTML5 storage features that would allow some offline working with online synchronization.

Lastly, it is interesting to observe how they implemented the tenants and the connectivity. Ledensite is using a flat structure for the tenants. That means that tenants are not interconnected with each-other and if a member of one organization wants to access another organization, he needs an account for the second organization too. Moreover, Ledensite identifies the tenants based on the unique customizable URL. In the case of AFAS Online, they use the login box to identify the tenant and from there they display the databases which the tenant can access. Also, for AFAS Online, they allow 3rd parties to access data on behalf of the tenant.

For the following two multitenancy implementations discussed in this chapter we will analyze the existing literature. Therefore, figure 21 shows with black where we are situated now in the PDD from figure 8.

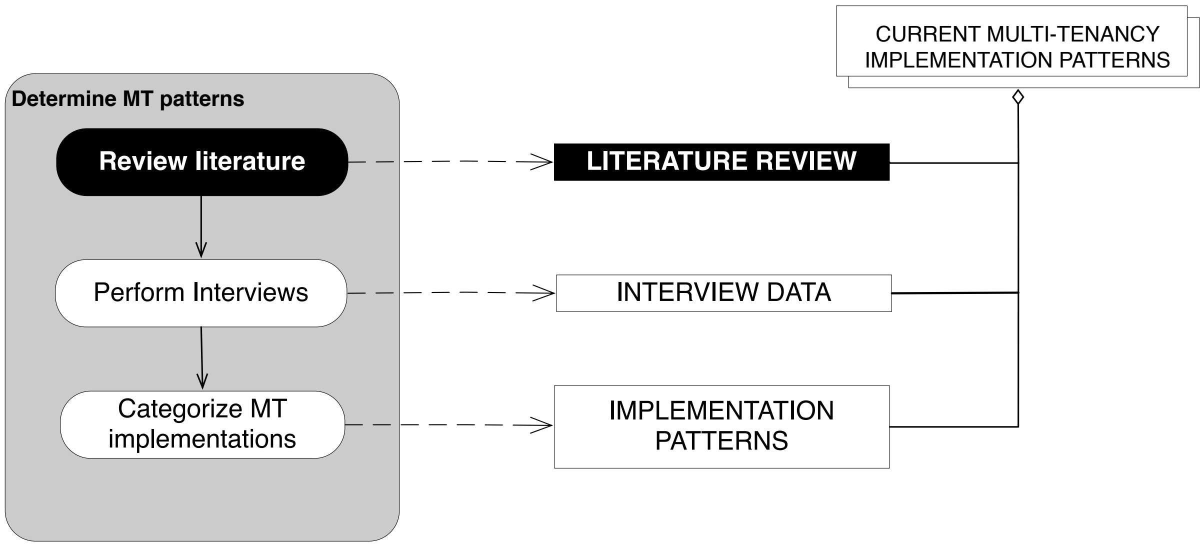


Figure 21. Current sub-chapters in the PDD from figure 8

4.3 Github

Github, the popular code sharing platform, is another interesting application that can be viewed as being multitenant but with a little variability (Dabbish, Stuart, Tsay, Herbsleb, 2012). The reason we consider in this research Github as being multitenant is because it is the same instance of the software that serves multiple organizations and open-source communities. Developers of organizations collaborate on projects that can are the tenant's (read organization's) data. Moreover, developers can work on projects from different organizations which enables tenant collaboration and tenant data sharing since data can be shared with developers of other organizations as well. Also, custom landing pages can be created for tenants applications. Organizations can use GitHub as their homepage which has links to their projects (tenant data).

In order to fully argue whether Github is multitenant or not, we can contrast it with Gitlab which is a self-hosted git management application. Gitlab can be seen as the single-tenant version of Github. Organizations can download Gitlab, personalize it (views, and even workflow) and use it internally for their own projects. That means, one instance of Gitlab is serving one single tenant.

In this subchapter we will be focusing on Github's architecture which has a different approach than Salesforce on managing tenants, tenants data and the users. Not only the architecture is interesting to follow, but also the fact that this is a highly used application and has to deal with scaling.

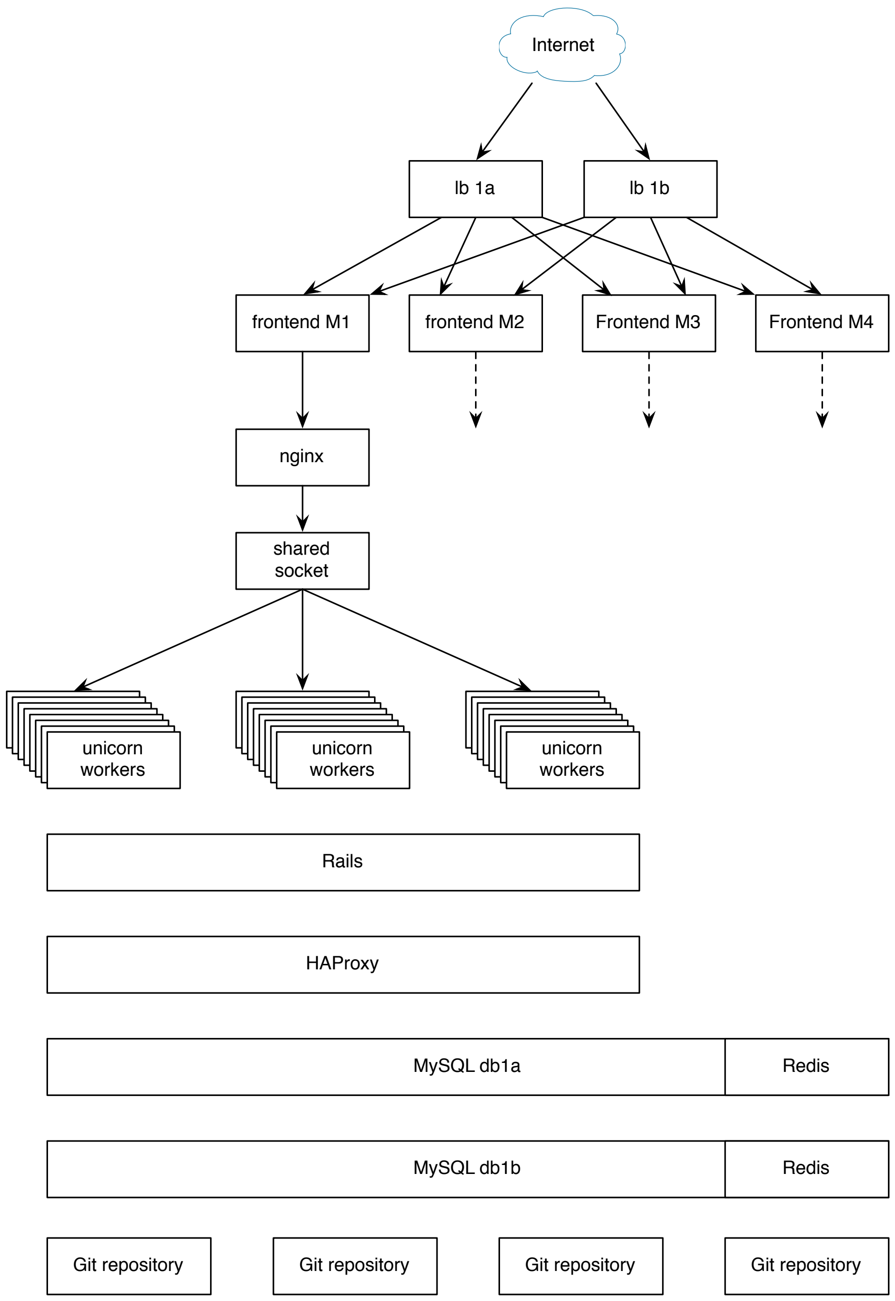


Figure 22. Part of Github's architecture according to Preston-Werner, (2009)

The article written by Preston-Werner, (2009) is very technical and explains the architecture behind Github and how they scaled it. The article begins by noticing that Github (compared to other multitenant applications such as Salesforce that only use HTTP) can be accessed via HTTP, SSH and Git. From the start, the complexity increases since there are three protocols that tenants use to interact with the application, to manage their data and to interact with each other. The author mentions that the easiest way to understand the architecture is to trace how the requests propagate through the system. When a HTTP request comes to one of Github's URLs, it hits an active load balancer. For the load balancing, Github uses Xen instances running ldirecord (Rief, Horman 2010). Xen is "a high performance resource-managed virtual machine monitor (VMM) which enables applications such as server consolidation, co-located hosting facilities, distributed web services, secure computing platforms and application mobility" (Barham, Dragovic, Fraser, Hand, Harris, Ho, Neugebauer, 2003). Github uses two load balancers (lb1a and lb1b). The load balancers forward TCP packets to one of the desired serves based on the port number of the packet and can also remove from the pool of servers those that are no longer responsive or are refusing connections. They use four front-end machines, each running a light HTTP server such as Nginx which accepts the connection and sends the request to a shared socket that calls Unicorn (Wanstrath, 2009) workers which ultimately load Rails (Ruby, S. 2010) to perform the required action. Figure 22 illustrates a part of Github's architecture. Github uses caching for fast retrieval of data. Also they use two MySQL servers for database lookups. For requests about Git repositories, if the data is not cached, they use Grit which is a Ruby plugin used for retrieving information about Git repositories, which are the actual tenants data. When Unicorn finishes the Rails action, they send back the response to the Nginx server and from there directly back to the client, skipping the load balancer. All the above were for a HTTP request.

Before going to explain the next two communication requests, it is worth mentioning that each tenant on Github, besides having an username/password pair to login on the website, they send to Github a public key which is used for identification in projects. So, when a tenant is added to a project, just a coupling between the tenant id and the public key is added to the authorization table for that project (git repository). This enables a secure way of sharing data between tenants.

Github is using SSH requests for communication between the users and the server to transfer tenants' data (Preston-Werner, 2009). The data is structured as Git repositories. For every request to fetch data from a Git repository, a SSH connection is made to one of their front-end servers. Users are authenticated by their public key, so they can connect to Github's ssh servers. Also, Github is restricting users to only allow them to execute git-related commands. That means, tenants are only allowed to work (upload, download, change) with tenants-data. Tenants' public keys, settings, payment info and other personal information are stored in the MySQL database and when the SSH connection is initiated, Github changed the frontend ssh daemon to perform a public-key lookup in the database to identify the user. If the public key is found and the user is allowed to establish the ssh connection, then they use a special service that checks if the tenant is allowed to access the data he requests, meaning that he is not trying to access someone else's git repository. If the user is granted access, the path to the server that stores the git repository must be determined. Github is using HAProxy for that. When the path to the Git repository is found, another ssh connection is executed to the server that stores the actual data and the user's command is performed there. So, after the permission has been granted, the frontend server becomes just a transparent proxy between the tenant and the data server.

Finally, the last type of request is the Git request. It is worth noting that this type of call is only used for accessing public Git repositories that are available on the Internet and read-only access (in Git's language is called cloning) is allowed. This can be viewed as tenants' public data which can be accessed without credentials by anyone on the Internet. The same as in the previous case, the Git proxy is a transparent proxy between the real file server where the git repository is stored and the tenant.

In the end, the author concludes that this architecture has helped Github to reduce the average response time that was decreased from more than 500 ms per request to less than 100 ms when they migrated the application.

While there is not much detail about how the database is designed, the article provides an excellent description on how to design a complex application with scaling in mind. Given the fact that by tenants' data Github uses Git repositories, it is not even important how the MySQL tables are designed, but how the authentication takes place and how the users get access to their data which could be public and / or private. It is interesting to learn how

Github managed to use the same infrastructure for all three types of requests that it can handle.

The information we gathered from literature study is used to partially answer RQ 1 where the implementation patterns are extracted. We use the findings from this chapter to create a multitenancy implementation pattern with general components.

4.4 Salesforce - Force.com

There are over 185,000 multitenant applications that have been built and are hosted in their cloud computing platform as a service (Force.com success stories).

The Force.com cloud platform runs the Salesforce CRM services, where each tenant has access to only his private data, and the activities are kept isolated. Being "a multitenancy oriented metadata-driven architecture" (Force.com, 2009), enables applications' functionalities and configurations to be described with metadata, allowing users to customize the applications as they want.

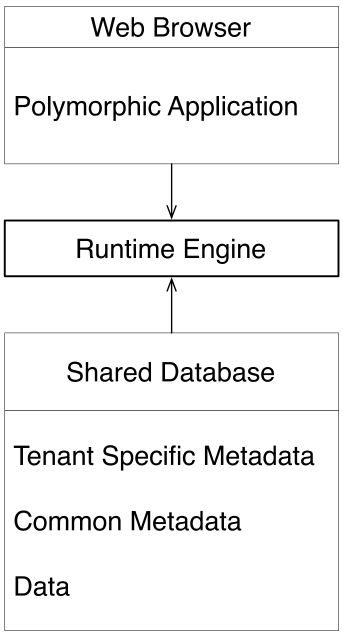


Figure 23. Metadata-driven architecture

Architecture

The rest of the subchapter is based on the paper by Weissman and Bobrowski (2009) and the whitepaper (The Force.com Multitenant Architecture, 2008).

According to Weissman and Bobrowski (2009) and (The Force.com Multitenant Architecture, 2008), the Force.com architecture is based on metadata. Everything that developers see and the users of the application is internally represented as metadata. Metadata are records in the database that store tenants' settings such as how the application should look, how the menu should be organized, what logo to use, but also metadata can define custom workflows. More about metadata is in said in the following subchapters that describe database architectures for multitenancy. Figure 23 above shows an example of how an application uses tenant-specific metadata, common metadata and ultimately data for its users. "Forms, reports, work flows, user access privileges, tenant-specific customizations and business logic, even the definitions of underlying data tables and indexes, are all abstract constructs that exist merely as metadata in Force.com's Universal Data Dictionary

(UDD)". An example of how the metadata-driven architecture works is illustrated in Figure 24.

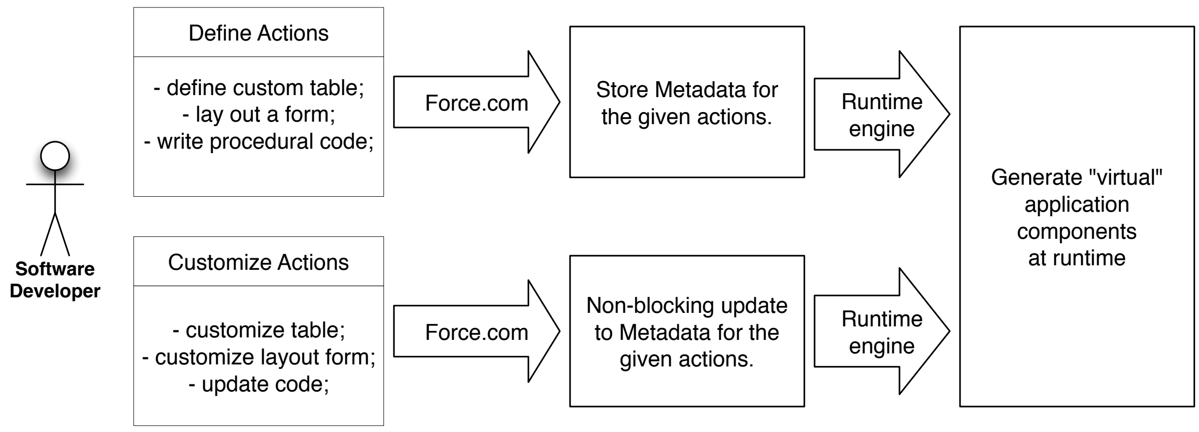


Figure 24. An example of how the metadata-driven architecture works

As it can be seen in figure 24, every action of the software developer is translated by Force.com in metadata. For updates or customized actions, an non-blocking update is required to modify the metadata which will be executed at runtime. As it can be concluded from figure 24, the runtime engine is the extra step that Force.com introduced in order to run the applications. This could be a serious bottleneck and could prevent the applications from scaling if not optimized. In order to prevent this, Force.com use metadata caches that maintain the most recent metadata in memory. Not only the metadata doesn't have to be read again since it is already in the memory, but also the compiled code is present there. This improves the multitenancy applications' response times and makes them easy to scale.

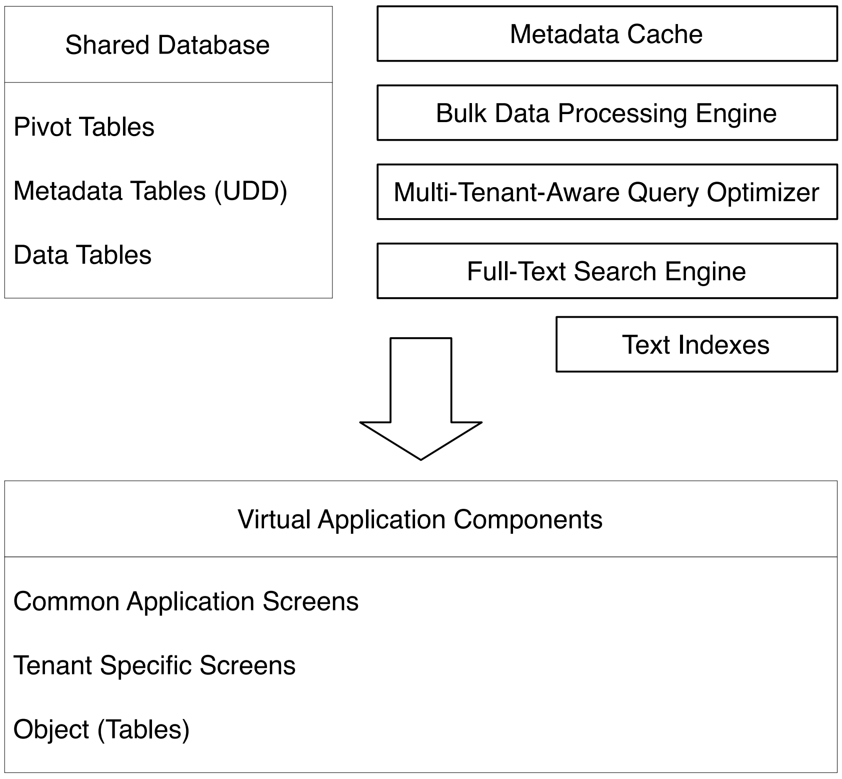


Figure 25. Force.com's metadata-driven architecture

In figure 25, the metadata-driven architecture of Force.com is presented. The users' actions are translated to metadata which are picked up by the runtime application generator that builds the applications. The internal operations are executed fast due to the multitenant-aware query optimizer which is one of the engine's most important parts. The query optimizer finds out which user is executing a given function, and "using related tenant-specific metadata maintained in the UDD along with internal system pivot tables, builds and executes data access operations as optimized database queries."

The application's response times are tuned by using an external search service which is optimized for full-text searching and indexing. It runs as a background process, not interfering with the running instance of the application. Thus data is processed and indexed in almost real-time while the application continues to deliver up-to-date and accurate information to the tenants.

Large data that needs to be processed and online transaction processing applications use the data processing engine that performs data modification operations in bulk. All applications' data from virtual tables is stored in large database tables that represent the heap storage. The platform's engine populates the virtual table with data from the large database tables after considering the corresponding metadata. In order to optimize data processing times (read, write), Force.com's engine uses special pivot tables which hold data that is used, for indexing, and relationships.

Figure 26 (Weissman et. al.) presents Force.com's storage model which manages the "virtual" database structure using metadata, data and pivot tables. In this way, tenants' data is stored in one place, being isolated at the same time from other tenants data. The database design plays an important role in the multitenancy software design. Therefore, an analysis on Force.com's database is presented in the following paragraphs.

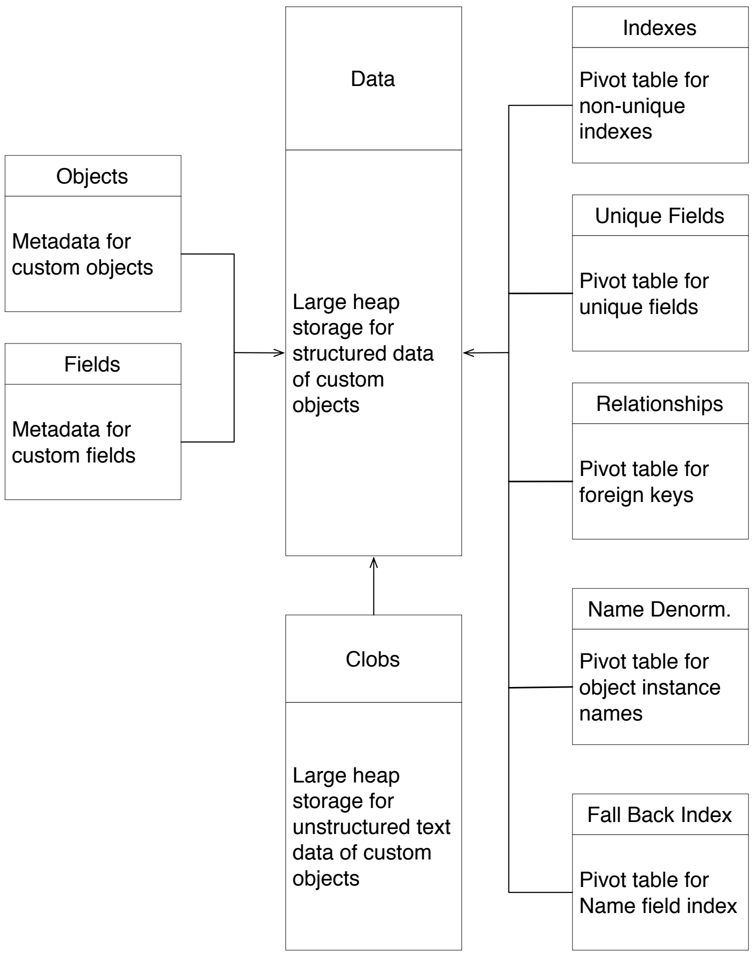

Data
Large heap storage for structured data of custom objects
Objects
Metadata for custom objects
Fields
Metadata for custom fields
Clobs
Large heap storage for unstructured text data of custom objects
Indexes
Pivot table for non-unique indexes
Unique Fields
Pivot table for unique fields
Relationships
Pivot table for foreign keys
Name Denorm.
Pivot table for object instance names
Fall Back Index
Pivot table for Name field index

Figure 26. Force.com's data definition and storage model

In the following paragraph, the objects, fields and data tables are described. For a better understanding, Figure 27 shows how they are connected.

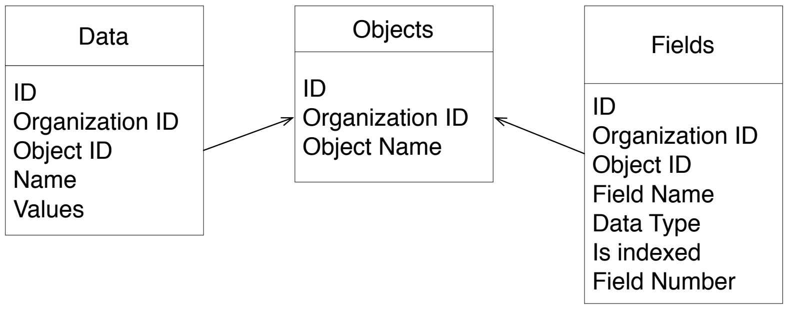


Figure 27. The relationship between the Data, Objects and Fields tables

The *objects metadata* table stores information about tables or entities that organizations define for their application (Weissman et. al.). The *data table* stores the application-accessible data. Each row includes fields for identification, such as the global identification field, the id of the organization that owns the row, and the object identifier.The *fields table*, as it can be seen in figure 27, stores information about fields that the organization defines for objects. The *data table* stores the application-accessible data which is mapped to objects and their fields. The values in the data table are all strings which gives flexibility for storing string, numbers and even dates. The *clobs table* contains character large objects (CLOBs) which allow a maximum size of 32,000 characters to be stored in the record. Entries in the clobs table are mapped to entries in the data table using a pivot clobs table which records the links between the two tables. In order to facilitate fast text searches, clobs are indexed outside the database as well. The *unique fields* table stores the fields of an object that an organization defines as having unique values (case-sensitive or case-insensitive). These fields are often required by multitenant business software, for instance when users sign up for the service and the email addresses of the accounts need to be unique. When an attempt is being made to enforce uniqueness of an existing field that contains duplicate values by an administrator or an application, an error message is raised to the application. This assures that the fields are unique and there is no way duplicates can be inserted. The *relationships pivot table* stores the relationships among the application objects. It helps optimize object joins. The *fallback index table* is used when the search engine is not available to respond to a search request in time. The application uses data in this table to perform the search, instead of returning an unsuccessful search error message to the tenant. Applications use the *name denorm* table to execute queries that fetch the names of objects instances for display as part of a hyperlink. The *history tracking table* is used by Force.com for audits. It keeps the history for any field about the changes made to the field, the old values, the new values, and the change date.

In the same way Amazon hosts its website on AWS, Salesforce has Salesforce.com deployed on Force.com (Liu, 2011). That proves that their concept works, is secure and assures a level of trust to new customers who build their multitenancy software on Force.com.

The information we gathered from literature study is used to partially answer RQ 1 where the implementation patterns are extracted. We use the findings from this chapter to create a multitenancy implementation pattern with general components.

Chapter 5 - Implementation patterns

In this chapter the four implementation patterns are presented. Figures 28 and 29 below show where the chapter is positioned in the PDD.

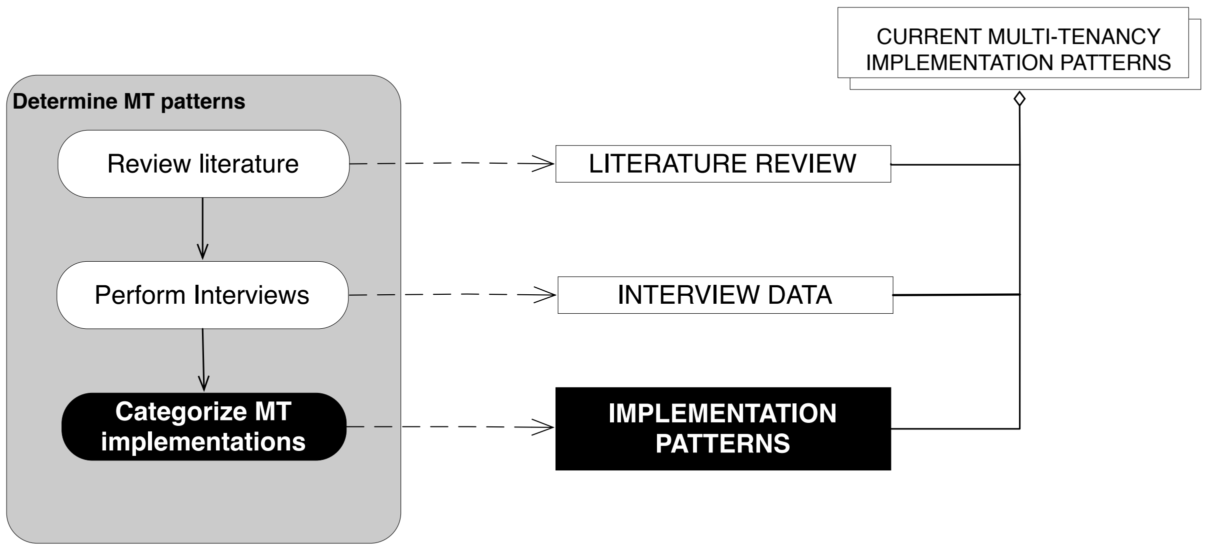


Figure 28. The multitenancy implementation patterns are identified in this chapter

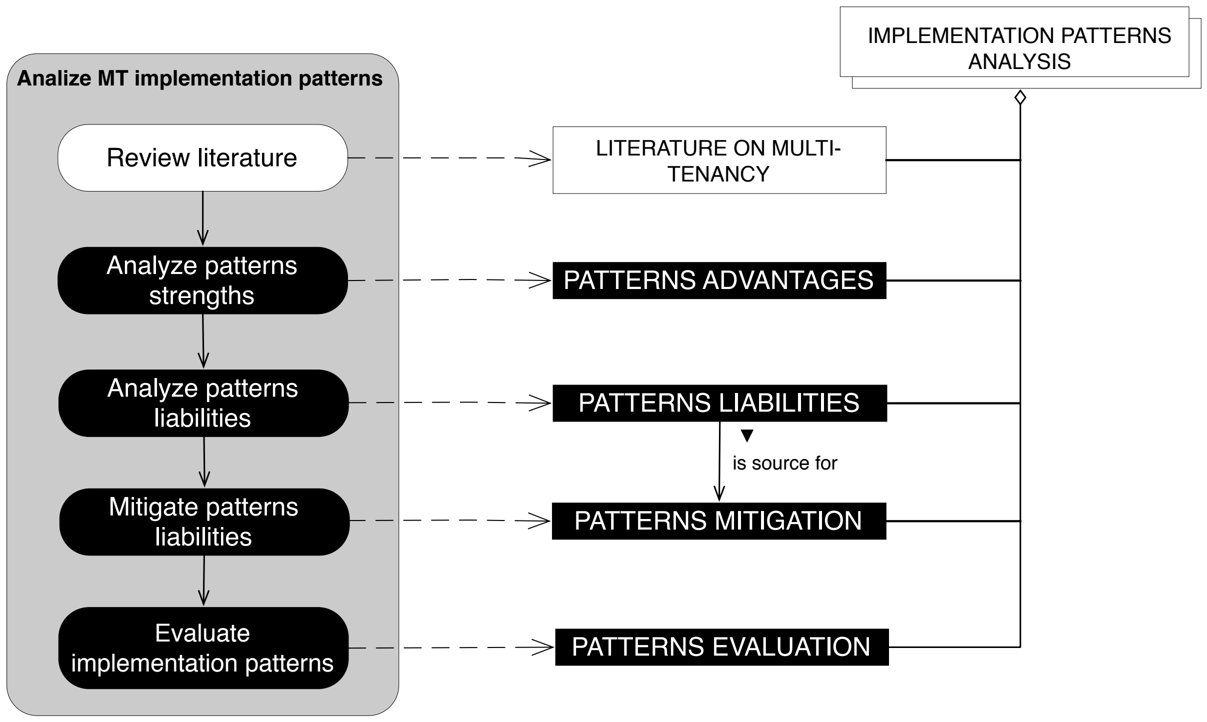

Analize MT implementation patterns
Review literature
Analyze patterns strengths
Analyze patterns liabilities
Mitigate patterns liabilities
Evaluate implementation patterns
IMPLEMENTATION PATTERNS ANALYSIS
LITERATURE ON MULTI-TENANCY
PATTERNS ADVANTAGES
PATTERNS LIABILITIES
is source for
PATTERNS MITIGATION
PATTERNS EVALUATION

Figure 29. The strengths and liabilities are listed and solutions to overcome them are given for each of the four patterns

5.1 The simple implementation pattern

The simple implementation pattern (figure 30) is created based on the case study on Ledensite (chapter 4.1).

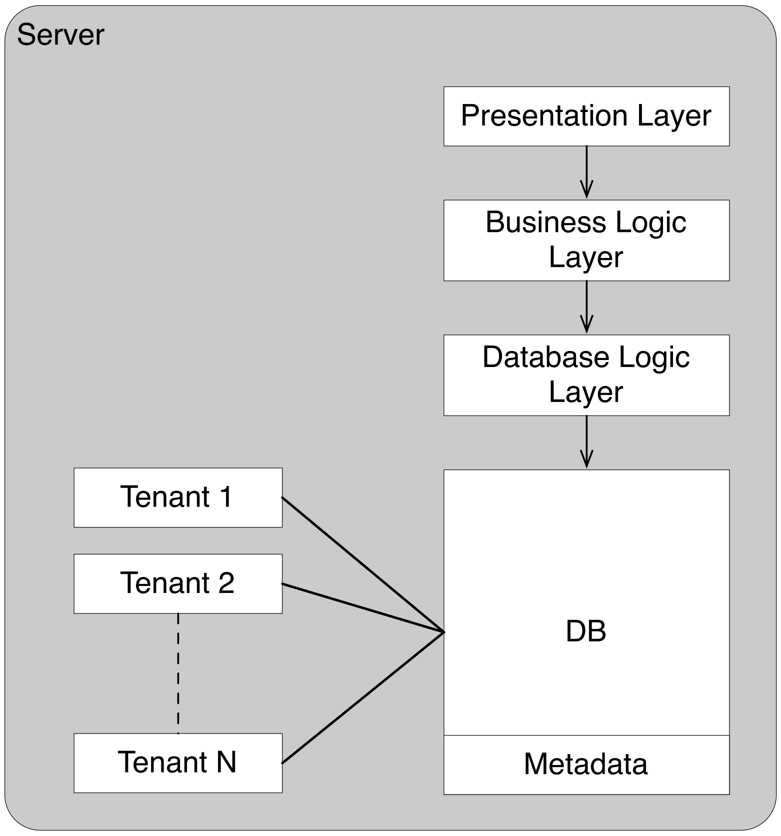


Figure 30. The simple implementation pattern

Advantages

If we look at the *simple implementation pattern* in figure 30 above and compare it with the other 3 patterns (see figures 31, 32, and 33), one of the first things that is visible is the simplicity of the pattern. It consists of one web server and one database server for all tenants. For a software developer wanting to create something really fast or start with a proof of concept, we recommend this pattern. Using one database to serve all tenants increases the flexibility. One change in the database will affect all tenants, thus no custom deploys and migrations per tenant. Even more, for a small application, using only one web server helps both in terms of costs as of configuration and deploy flexibility.

The second advantage relies in the framework architecture which in our case is the MVC (Model, View, Controller) model. Our pattern maps the Presentation Layer to the Views in MVC, the Business Logic Layer to Controllers and the Database Logic Layer to Models. Thus every web development framework that uses MVC is suitable for this pattern. Even for new developers, this structure helps keeping things simple and clear and on the long run it makes the project maintainable.

A third advantage is given by the high degree of customization. As Jansen, Houben and Brinkkemper (2010) note in their paper, most of the changes happen in the controller. Refer to Chapter 3, section 3.4, for more details about customization.

Finally, the fourth advantage of this pattern, is the ease to re-engineer a multi-user and single-tenant application to a multitenant application. As Bezemer, Zaidman, and Platzbeecker (2010) noted in their paper (see Chapter 3, section 3.7, 'From single-tenant to multitenant') for an existing single-tenant application to be upgraded to multitenant, there are three components that require minimal changes, namely: the authentication, the configuration and database. Refer to the summary in section 3.7 for more details on how to create a multitenant application starting from an existing single-tenant application.

Liabilities and mitigation

The simplicity of the multitenancy implementation pattern in figure 30 brings some liabilities. The first liability is scalability. A multitenant application using this pattern cannot simply scale. Although one server can handle quite a large number of tenants and workload, when the application grows, adding more servers will become a necessity. The solution is to split the server in two servers, one for the web server and the second for the database server. If the application is deployed in the cloud, the approach will work as long as the application can handle the requests by using the two servers with maximized resources the cloud provider offers (RAM, CPU and storage). However, the moment a third server becomes necessary, the multitenancy implementation pattern needs to be adapted or changed with one that is more scalable (see the other three mentioned below).

A second liability is using one database to store all tenants data. Some tenants might see it as a security issue if they know their data is stored on the same database with other tenants' (refer to Chapter 3, section 3.8, Security). While it might not be an issue at the beginning, it could be possible that one tenant will explicitly require to keep his data in a separate database. When this happens, it is not easy to offer a solution that keeps data from the other tenants in the same database and the data for the new tenant in a separate database. The problem only appears when the number of tenants increase, the database will eventually need to be split into many. One approach would be to use a database server per tenant and that would change the current pattern to the *extensible implementation pattern* (discussed in chapter 5.2). The other approach would be to use techniques to partition the database, like the ones proposed by Tsai, Shao, Huang and Bai, 2010 (see Chapter 3, subchapter 3.3.3, 'Database Partitioning'). The authors give three techniques that can be used to partition the database, namely the "row stores and horizontal partitioning", optimized for reads, the "column store and vertical partitioning", optimized for writes, and the "two layer partitioning model, P2", that is optimized for both reads and writes. Refer to the summary on subchapter 3.3.3 for more details. Additionally, there is a different approach defined by Curino, Jones, Popa, Malviya, Wu, Madden, Balakrishnan, Zeldovich (2011), the Relational Cloud, which is summarized in Chapter 3, section 3.3.2, 'Database-as-a-Service for the Cloud'. Their solution minimizes the hardware resources and scales automatically when the workloads increase, allowing to scale a single database to multiple machines and load balance the traffic on the back-end nodes.

Finally, another liability is at the database layer in the way metadata is stored. Using one database to store all tenants metadata works for a small number of tenants. However, when the metadata that needs to be stored increases, then using one row per tenant's metadata will not be feasible and having to split the metadata table into many tables for all tenants will increase the costs of queries. One of the papers that was summarized in chapter 3, section 3.3.1, introduces the Chunk Folding technique (Aulbach, Grust, Jacobs, Kemper, and Rittinger, 2008). We point the reader to the subchapter "Schema mapping techniques" of chapter 3 for a solution to minimize the query times and store metadata more efficiently and obtain a good performance.

Evaluation

In order to evaluate the current pattern we sent it to the multitenancy expert at Ledensite and asked for his feedback. The feedback we received was positive and the small comments the expert had were updated in the pattern description.

5.2 The extensible implementation pattern

The *extensible implementation pattern* (figure 31) is created based on the case study at Afas (chapter 4.2). In terms of complexity is the second after the *simple implementation pattern* described in chapter 5.1, and it is more suitable for large multitenant applications that go up to thousands of customers.

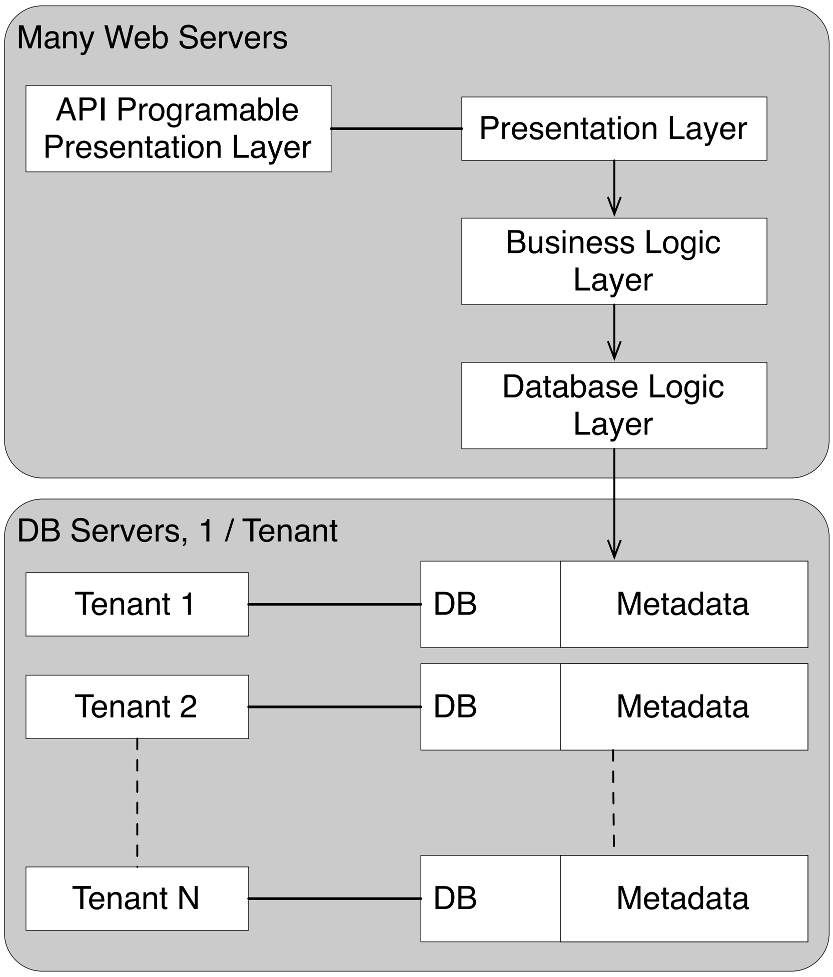


Figure 31. The extensible implementation pattern

Advantages

The first strength that is visible in the picture is the clear separation between the web servers and the database servers.

Speaking of tenant's database, a second strength is the pattern is using one database per tenant. This eliminates the security issues mentioned by Wainewright (2012) that some people have with storing multiple tenants' data in the same database (see Chapter 3, the Security topic).

A third strength is the ease of extensibility. An application can easily scale up by simply adding database servers and web servers as needed.

A forth benefit, the same as for the *simple implementation pattern* is that the current pattern is built on the same MVC structure. However, the current one includes a programable presentation layer that allows external developers to create custom views.

A fifth benefit is that it is easier than for the previous pattern to measure the service performance (Li, Liu, Li and Chen 2008). Refer to Chapter 3, subchapter 3.6.1, ‘Isolation’, that introduces a service performance isolation infrastructure in multitenancy environment.

Finally, the sixth strength of this pattern which is also a strength of the previous pattern, is the ease to re-engineer a multi-user and single-tenant application to a multitenant application. As Bezemer, Zaidman, and Platzbeecker (2010) noted in their paper (see Chapter 3, section 3.7, ‘From single-tenant to multitenant’) for an existing single-tenant application to be upgraded to multitenant, there are three components that require minimal changes, namely: the authentication, the configuration and database. Refer to the summary in section 3.7 for more details on how to create a multitenant application starting from an existing single-tenant application.

Liabilities and mitigation

The first liability that is also visible from figure 31 is that there is only one database per tenant. If that database goes offline, the application becomes unavailable for the tenant. There are two ways in which this liability can be mitigated. The first one is to add a slave database that takes the primary role when the master is not available. However, for a large number of tenants this solution would become expensive. On the other hand, it is possible to make the system available for all tenants regardless of the state of their primary databases. The solution consists in creating a global database that maps all tenants in one database and keeps itself synchronized with all the individual tenants’ databases. The global database takes the role of a secondary database and it is only used when the primary database for a tenant becomes unavailable. By doing so, the extra costs to maintain such a database can be neglected even for a large number of tenants.

In order to avoid security concerns of keeping data of multiple tenants on a failover database, when the primary database becomes unavailable, then the tenants can continue using the application but only in read-only mode. Then tenants will not complain that their data stays on the same database with other tenants (Chapter 3, section ‘Security’, Wainewright 2012) because it cannot be altered by other tenants. And so, the application will always be available to the tenants and the problem of the database as a single point of failure is partially eliminated. To totally eliminate the problem, then the global database would have to support writes as well but this depends on the business policies.

A second liability is the high cost to create a failover database system when a large number of databases exist. It depends on the database server if it supports a failover mechanism. The solution proposed above is a mixture of the database layer of the *simple implementation pattern* in figure 30 and the *extensible implementation pattern* in figure 31.

5.3 The protocol-based implementation pattern

The *protocol-based implementation pattern* (figure 32) is created on the case-study on Github (chapter 4.3).

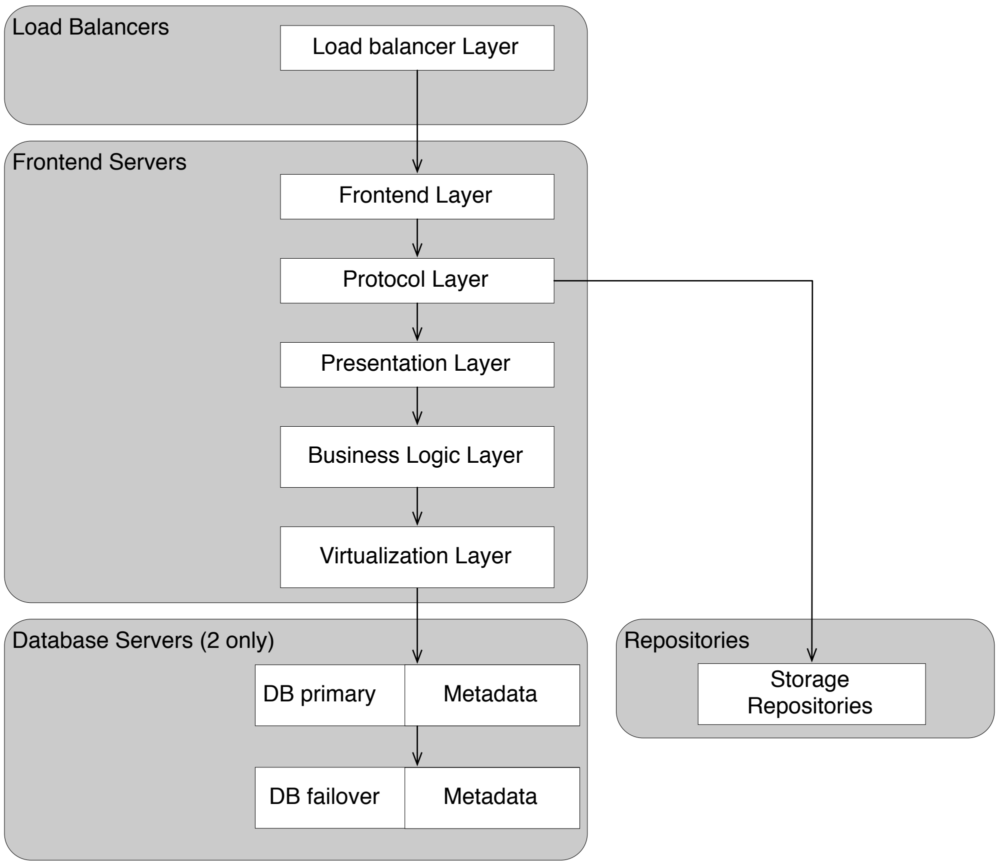


Figure 32. The protocol-based implementation pattern

Advantages

The first advantage that is visible in figure 32 if we compare it with figures 30, 31, and 33 is the fact that the *protocol-based implementation pattern* is the only that supports multiple protocols (HTTP, SSH and Git) through the Protocol Layer.

The second advantage is that the pattern can handle high traffic loads with the help of the Load balancer Layer that spreads requests across multiple servers. That implies there are more frontend servers, each using the same architecture, as it is visible in figure 32.

The next advantage comes from the multiple front-end servers, and is more a consequence of the load-balancer layer: the *protocol-based implementation pattern* is very good to scale horizontally. Whenever the application needs to handle more requests, the only thing needed is to connect more front-end servers to the load-balancers and the scalability problems are solved.

Another advantage is at the database layer. The pattern uses only two SQL servers for all their users, one server being the failover for the primary. This is an advantage not only because it provides a failover mechanism, but also data is being synchronized at a minimum cost (only two SQL servers) all front-end servers talk to the same database instance. The pattern is simply plug and play for the front-end servers.

The final advantage is represented by the multiple storage repositories. These servers store tenants' repositories and they are accessed by the front-end servers. Their location (IP address) is stored in the database, so adding more storage repositories will not affect the performance of the system. In terms of flexibility and scalability, the approach is great because scaling-up or down is just a matter of adding or removing storage servers.

Liabilities and mitigation

The first liability of the *protocol-based implementation pattern* is the complexity. While it works well for GitHub, the pattern is complicated for a new multitenant application. The two multitenancy implementation patterns that we already presented (the simple and extensible ones, figures 30 and 31) are less complex and can be chosen for creating a new multitenant application.

The second liability is the low level of variability the pattern offers. As Kabbedijk and Jansen, (2011) mention (Chapter 3, section 3.5, 'Variability') variability enables SaaS environments to serve a large number of tenants where each tenant has its own requirements and a variable multitenant application can suit their needs. It can also be seen in figure 32 compared with the other three implementation patterns that the *protocol-based implementation pattern* has no programable layers, thus it is not possible for tenants to define their requirements and program the application to suit them. In order to mitigate this liability, the degree of variability can be increased by updating the pattern with a programable "customizable data views, module dependent menus and pre/post update hooks" (refer to Chapter 3, section 'Variability', Kabbedijk and Jansen, 2009).

5.4 The metadata-driven implementation pattern

The *metadata-driven implementation pattern* (figure 33) is created based on the case study on Salesforce - Force.com (chapter 4.4).

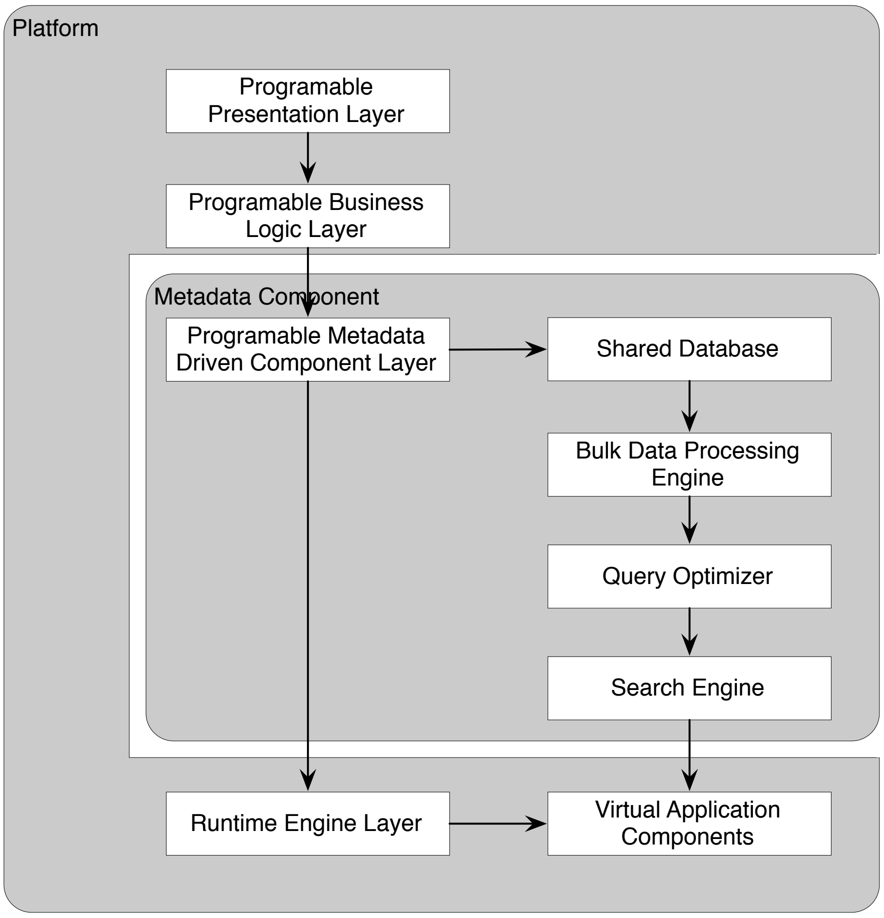

Platform
Programable Presentation Layer
Programable Business Logic Layer
Metadata Component
Programable Metadata Driven Component Layer
Shared Database
Bulk Data Processing Engine
Query Optimizer
Search Engine
Runtime Engine Layer
Virtual Application Components

Figure 33. The metadata-driven implementation pattern

Advantages

The biggest advantage the *metadata-driven implementation pattern* (figure 33) has compared to the three others (figures 30, 31, and 32) is the existence of the Metadata Component. It is an advantage because it offers extreme flexibility. Everything that is defined on the platform is treated as metadata.

The second advantage is the presence of the Runtime Engine Layer. The Runtime Engine Layer is optimized to render the views and to handle the components interactions based on metadata. The engine is built to maximize metadata processing speed.

A third advantage comes from the two programable layers at the top of the implementation pattern (figure 33). Being entirely programable offers maximum of flexibility.

Finally, we emphasize the presence of a query optimizer and the search engine. Both help in reducing the times for long queries and increasing the accuracy of search results.

Liabilities and mitigation

The current multitenancy implementation pattern is the most complex of the four described in this research. The *metadata-driven implementation pattern* should not be used by multitenant software developers unless they do extensive research on the three others implementation patterns. The current pattern has been designed to work well for Salesforce’s needs. Therefore we suggest it should only offer information for anyone who needs to create a metadata-based multitenant application.

A second liability is the high cost to maintain an application deployed on such an infrastructure. The costs are high because there are many servers that are required to support the infrastructure for the *metadata-driven implementation pattern*.

The only solution for a company willing to create a multitenant application using the *metadata-driven implementation pattern* is to take as example the way everything is stored as metadata and to understand the necessity of the runtime engine and the virtual application components. The pattern illustrated in figure 33 is on a very high-level for multitenant applications that require a similar behavior.

Chapter 6 - Patterns overview and comparison

According to Schmidt, Fayad, and Johnson (1996) a pattern must describe the contexts where problems arise, the problems that it solves and the conditions under which solutions to problems can be recommended. In the current chapter we are going to answer the third research subquestion by offering an overview of the four implementation patterns from chapter 5 and by describing the problem forces. The reader will understand what the appropriate multitenant business software patterns should be chosen for developing multitenant business software. Figure 34 shows where we currently are in the PDD.

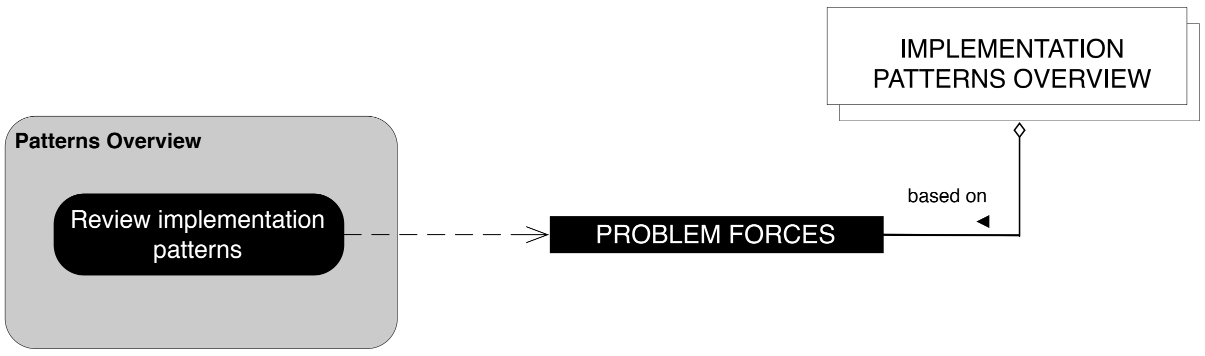

IMPLEMENTATION PATTERNS OVERVIEW
Patterns Overview
Review implementation patterns
based on
PROBLEM FORCES

Figure 34. Implementation patterns overview

In the following paragraphs we are going to point out the problems that the four implementation patterns are solving. Each of the paragraphs give a context with a problem and the implementation pattern(s) that can be chosen to solve the problem:

When a new multitenant application needs to be created and the concept is not yet proven, then we recommend the reader to start with the *simple implementation pattern* discussed in chapter 5.1. The pattern is excellent for starting a new application (small to medium business) and for quickly creating a proof of concept. Also the costs to start up are the lowest since only one server is required. The fact that the pattern only uses one server, it makes it very easy to deploy the application in a cloud environment and scale up the resources of the machine where the application runs only when needed.

Given an existing single-tenant and multi-user application exists and it needs to be reengineered to multitenancy, then we point the reader to the *simple implementation pattern* and the *extensible implementation pattern* (chapters 5.1 and 5.2). They both work because they are built on the MVC (Model - View - Controller) principle. An existing single-tenant application built on MVC can be reengineered to multitenancy with little effort if the *simple implementation pattern* is chosen because there is still one database and the only change that needs to be done is to add a *tenant id* column to the tables that store tenants' data. On the other hand, by using the *extensible implementation pattern* the only changes that need to be done is to configure the application to connect to a different database server for the tenant's data and metadata. A good example is given by Bezemer, et. al. (2010) in chapter 3.7 where a existing single-tenant application was reengineered to multitenancy.

If the business grows from medium to large then we point the reader to the *extensible implementation pattern* (chapter 5.2). The pattern is excellent because of its flexibility. Web servers and database instances can be added on the fly, without requiring changes in the application. Thus, when the business grows and one web server can no longer handle all the traffic, switching to the *extensible implementation pattern* works to scale up, because it makes it very easy to add more web servers to run the application. It is possible to place the web servers behind a load balancer to keep the application running as virtually one-instance for all tenants. An application that was already using the *simple implementation pattern* can be easily updated to the *extensible implementation pattern* by separating the web server from the database server in two separate servers. When the application runs on two separate machines, one for the web server and one for the database server, then the application is using the *extensible implementation pattern*. From that moment, more web servers can be added to scale up the application as traffic loads demand.

When an application grows on a low budget and it is using the *simple implementation pattern* and cannot afford to migrate to the *extensible implementation pattern* because the costs to have a database server per tenant would be too high, then we recommend the reader to study the *protocol-based implementation pattern* on the database servers layer. The pattern only uses two databases, one as primary and the other one as failover. However, choosing a hybrid version of this pattern, that means two database servers but only one protocol, could make it harder in the future if the application needs to allow tenants to have an own instance of their database.

In the case tenants require their own database or when the business policy requires to store tenants' data in separate databases (see the case study for Afas Online, chapter 4.2) then we point the user to the *extensible implementation pattern (chapter 5.2)*. The pattern works well for this scenario because of its flexibility that allows to simply plug-and-play database servers to the application when needed. Moreover, no changes in the application code are required to handle the creation or deletion of database servers.

Given a multitenancy application needs to support multiple protocols (such as HTTP, SSH, GIT) then we recommend the reader to get inspired from the *protocol-based implementation pattern* (chapter 5.3). The protocol-based pattern as it is shown in figure 32 is recommended for large applications. However, multiple protocols can also be supported in small applications, for instance those created using the *simple implementation pattern* by adding a Protocol Layer on the top of the Business Logic Layer.

Finally, if the situation requires all tenants' data to be stored as metadata or to allow tenants to define custom data fields, then we point the reader to the *metadata-driven implementation pattern* (chapter 5.4). The pattern is the most flexible one in terms of handling tenants' data. However, this flexibility comes with a cost that is the increased complexity. We recommend the reader to only understand how the *metadata-driven implementation pattern* works and to try to define themselves their own metadata structure, suited for their needs. Multitenant applications that need to store everything as metadata, can add a Metadata Component consisting of a Programable Metadata Component Layer (figure 33) and also the Runtime Engine Layer to resemble data from metadata and generate the Virtual Application Components.

In the paragraphs above we listed several problems that can be solved by using our multitenancy implementation patterns. We use the solutions as answer to our third research subquestion that asked what the appropriate implementation pattern is for developing multi-tenant software. The appropriate pattern depends on the problem it needs to solve, and the paragraphs above illustrate some of these problems.

Patterns comparison

In chapter 4 we show how we gathered data for the four implementation patterns, in chapter 5 we introduce the patterns with their advantages, liabilities and possible mitigation solutions, however we have not yet compared the patterns. In table 18 on the next page we compare the patterns consequences and link them to possible mitigation solutions with references to where inside the research more information can be found. To make the comparison complete, in table 20 we compare the patterns advantages, pointing out what they are excellent for and their other advantages.

In the following table, because of space limitations we indexed the implementation patterns as follows:

- MT1: The simple implementation pattern;
- MT2: The extensible implementation pattern;
- MT3: The protocol-based implementation pattern;
- MT4: The metadata-driven implementation pattern;

Next, we show the consequences and the possible mitigation solutions for each pattern:

Pattern	Consequences	Mitigation solution	References (chapter)
MT1	Server scalability	Split the server in two servers (one for web, the other for database);	N/A
		When a second Web server is needed, consider the Extensible implementation pattern;	5.2
MT1	DB scalability	When a second DB server is needed, consider the Extensible implementation pattern;	5.2
		Use Database Partitioning techniques	3.3.3
		Consider Relational Cloud Database as a Service solution	3.3.2
MT1	Metadata Scalability	Use the Chunk Folding technique	3.3.1
MT2	DB as a single point of failure	Use one database to store all tenants data. If individual databases are required per tenant, then add one failover database that holds all tenants data. Use Relational Cloud Database as a Service as starting point for the failover DB server.	3.3.2
MT2, MT3, MT4	High costs to maintain the application on such an infrastructure	Start with the simple implementation pattern and scale only when needed. Before scaling, in order to reduce costs, consider solutions from chapter 3	3.3, 3.4, 3.5
MT3, MT4	Complexity	Use the pattern as example only on how to handle metadata (MT4) or how to support storage repositories (MT3). Then consider one of the simple implementation pattern (MT1) or the extensible implementation pattern (MT2) and adapt them with the components extracted from MT4 or MT3.	5.1, 5.2
MT3	Low level of variability	Update the pattern with “customizable data views, module dependent menus and pre/post update hooks” (Kabbedijk and Jansen, 2009)	3.5

Table 18. The consequences comparison for each of the four implementation patterns with possible mitigation solutions

So far we have summarized the consequences and the possible mitigation solutions (table 18) for our four implementation patterns. In order to finalize the comparison between the four patterns, the following table, table 19, lists their advantages.

MT implementation pattern	Advantages
Simple implementation pattern	• Excellent for starting up with a new project of for quickly building a proof of concept; • It is the most lightweight pattern of the four and the most cost-efficient; • Uses minimum resources to run the application: requires only one server; • Built on MVC and there are many web frameworks available that use MVC, such as Ruby on Rails; • High degree of customization; • Suitable for reengineering a single-tenant application to multitenancy;
Extensible implementation pattern	• Excellent for scaling the application - scale only as needed by simply adding more servers or databases - plug-and-play fashion; • It is the most flexible pattern since each tenant has their own database; • Clear separation between the web and database servers; • Built on MVC and thus benefits of the existing web frameworks use MVC; • The easiest to measure service performance; • Suitable for reengineering a single-tenant application to multitenancy;
Protocol-based implementation pattern	• Excellent if the application needs to support multiple protocols (such as HTTP, SSH, Git); • Uses a load balancer layer to handle high traffic loads; • The database layer is lightweight: it only uses two SQL servers, one as primary and the other one as failover; • Storage servers can be easily added or removed, in the same plug-and-play fashion;
Metadata-driven implementation pattern	• Excellent if the application needs to handle everything as metadata; • It is highly customizable by the programable layers;

Table 19. Advantages comparison of the four implementation patterns

As it can be seen in table 19, each pattern has a list of advantages. There are patterns who have common advantages, such as the simple implementation pattern and the extensible implementation pattern who for instance both use MVC and are suitable for reengineering a single-tenant application to multitenancy. However, each pattern is excellent for something and that is given as the first advantage for all four patterns.

Chapter 7 - Discussion

In chapter 2.6 we introduced the four validation tests by Yin (2003), namely: construct validity, internal validity, external validity and reliability. As we finalized the case studies, we will analyze here each of the four validation tests to see if they are satisfied.

In order to ***construct validity*** we need to satisfy two steps. The first step was to select specific types of changes and to relate them to the original objectives of the study. We consider that the first step is fulfilled since we analyzed four multitenancy applications and determined the four implementation patterns. The patterns were determined based on the differences in the architectures from an application to another, hence the application specific types of changes.

For the second step, we had to demonstrate that the selected measures reflect the types of changes from step one. In his book, Yin (2003) says that in order to satisfy the second step we need to send a draft of the case study for review to multitenancy experts. However, we only managed to get feedback from one multitenancy expert and only for the first implementation pattern. The other expert could not validate our case study findings and for the other two case studies based on literature research we did not contact experts to validate our findings. However, the other three implementation patterns were not invalidated either so no conclusion can be taken yet whether they validate or not.

Therefore, for construct validity, we only managed to satisfy the first step while the second step is partially satisfied. However, if we were to divide the second step by the number of the patterns that were identified, 25% of it passes, giving an overall of 75% passing score combined with the first step. So, construct validity passes the validation test but not fully.

Internal validity assumes that the casual relationships that we determine are properly linked. We use casual relationships, although not explicitly, in the construction of the patterns. Each pattern consists of components that are linked together, therefore in a casual relationship. For instance, a request for an application using the extensible implementation pattern (chapter 5.2) could not just simply request metadata for Tenant 1 without first going through the presentation layer, the business layer and in the end the database logic layer. Also, our patterns include arrows between components and the arrows show the casual relationships, therefore component A→B means that component A is executed before component B. However, the test assumes that there is no component C that can cause B or the test fails.

Because we designed our patterns on a very high level and they do not contain any application specific subcomponents, we believe that for the patterns that we identified there is no component C that could be put in-between components A and B, without going into specific details. Simply, we designed the implementation patterns to be minimalistic so that nothing else could be removed without affecting the whole structure of the pattern.

Moreover, if we use the formula from chapter 2.6 (at the description of internal validity), we can easily observe that all the sets of components from our patterns are disjunct, so the formula evaluates to true and the multitenancy patterns matrix contains sets of components that are all different. It is also visible from the figures that represent the patterns that they are all different.

However, one could argue that we have only proven that nothing could be removed but more casual relationships could exist. This is true since one of the threats of the research is the low number of interviews that we performed. For each pattern we only have one application and it could be that one of the patterns could be improved by adding another component that was invisible in the application that it derived from.

Therefore, we think that the internal validity test partially passes based on the reasoning above, until the contrary is proven and given the presence of the threats.

External validity assumes that the findings are generalizable beyond the case study. Our findings from the case studies consist of four implementation patterns. The implementation patterns consist of multiple components themselves as we described in the previous paragraphs at internal validity. All components' functions are general so there is no component that is linked to a specific functionality of the application it derived from. For instance for the implementation pattern that derived from Github, we called the layer where the actual Git repositories are stored simply “Repositories” because they could mean SVN repositories or anything else that has the function of a storage repository.

Our findings from the case studies moved from specific to abstract. We began with specific details about the applications that we investigated (from interviews and literature study) and ended up with abstract implementation patterns based on the applications. So, in the end the findings are generalized from the concrete data.

Based on the reasoning above, we believe that the external validity is fully satisfied in our research. Unlikely the previous two tests that did not fully pass because of threats, we cannot identify any threats that would prevent the current test to fail.

Finally, the ***reliability*** step assumes that if someone else follows the same procedures as we did and conducts the same case studies, then the findings and the conclusions should be the same.

The procedures that we followed throughout the case studies are traceable on the PDD in figure 8. The PDD consists of the activities that we performed and outputs the deliverables. If by following the PDD by someone else the results would be the same, then we believe that performing the case studies by someone else the findings and conclusions would stay the same. Even the list of threats would remain the same in this case.

So we believe that our research is reliable and the work can be easily extended by anyone who wishes, therefore the reliability test passes.

Validity test	Results	Comments
1. Construct Validity	Partially pass	First step passes; The second step only evaluates 1 out of 4 implementation patterns. While the other three were not validated we consider it a 25% pass.
2. Internal validity	Partially pass	There are threats that can still make the implementation patterns fail this test.
3. External validity	Fully pass	No threats could be identified to make this validity fail;
4. Reliability	Fully pass	The research is extendable.

Table 20. Summary of the four validity tests based on Yin (2003) on our case studies

Table 20 summarizes our validity tests results. We can conclude that overall the tests passed, but still the result could be improved to make the first two tests fully pass. We do not have enough data yet to affirm that the first two tests are passing, but we also do not have any contrary arguments to say that they fail. So that is why we said that they partially pass because it could happen that one of the patterns is not correct or the casual relationships between the components are not correct.

Looking at the results so far, we cannot say that the list of patterns is complete. We believe there is room for more research on the field and new implementation patterns can be discovered. Even more, the current patterns are not finalized so they can be improved.

In terms of evaluation, we only managed to evaluate the first implementation pattern. It was the only one that was reviewed by the expert of the application that it derived from. Therefore, we can only be fully confident for the correctness of the first implementation pattern. However the other three have not been invalidated either, and the case studies that helped us creating them show that they match the architecture of the applications.

It is important to notice that we are aware of the following threats and limitations for our research:

- The first threat comes from the low number of interviews per case study. It would have helped to find more multitenant applications that use the same pattern, and then analyze them and extract a more accurate pattern;
- The other threat comes from the fact that two of the patterns were derived from literature research. While they might be correct as they are, it could be that the available literature did not mention all the aspects of the application and then our implementation patterns would be incomplete;
- Another limitation lays in the fact that the literature that we studied was only based on papers that were available with the student account through the University's Library proxy. There were other portals such as IEEE which our search results hinted relevant papers. While on some of the papers on these portals we managed to get access, there were papers that were simply ignored because no free access was given to them.
- The possible mitigation solutions that we offer in our research could work good enough to solve a liability, but we are aware of the fact that in some cases they might not be the best solution.
- The list of possible mitigation solutions that we offer in our research is not complete. It is limited on the amount of literature that has been studied and we are aware of the fact that more solutions, perhaps even better ones can exist.
- The list of multitenancy specific problems that we give in the first part of chapter 6 and the solutions that we offer to these problems by using the implementation patterns is not complete. There could be more problems that these patterns solve.
- The solutions to multitenancy specific problems that we give in chapter 6 might not be entirely correct or complete. Because we did not get the validation of a multitenancy expert for them we cannot guarantee their correctness or completeness.

Given the limitations and the threats, we managed to create four implementation patterns that on one hand can help companies to reduce the risk on choosing the wrong implementation type for their projects while on the other hand helps researchers to extend our work with multiple case studies and more implementation patterns.

Chapter 3 gives a review relevant topics from existing literature. As we mentioned in the list of limitations and threats, we were unable to consult all the publications on the subject. Some of them were not accessible. However, we managed to cover the two main issues that are important for multitenancy, namely database solutions and security. We also noticed during the interviews the importance of these two topics. A number of concepts have been reviewed in this chapter and at the end we created a concept matrix (table 13) to illustrate the findings of our concept-centric literature. As it can be seen in the table, many authors cover the same concepts, such it is the case with security, variability and multitenancy database solutions. All these concepts have been linked with the multitenancy implementation patterns from chapter 5. The concepts were linked with the liabilities of the patterns as solutions to mitigate them and in some cases with the advantages of the patterns, such it is the case with the simple implementation pattern which links the variability concept to its list of advantages and with the extensible implementation pattern which links the service performance infrastructure to its list of advantages.

Moreover, at the end of chapter 6 we made a comparison of the four implementation patterns. The comparison shows clearly what their consequences are and links them to possible mitigation solutions (table 18). While the information in the table gives a solution for each of the liabilities, there could be cases when the solutions that we offer are not the optimal way to solve a liability of the pattern. Besides the liabilities and the possible mitigation solutions, we also listed a number of advantages (table 19), pointing for each pattern what it is excellent for and its other advantages. We have noticed that some patterns have common advantages, such as the simple implementation pattern and the extensible implementation pattern, both have a number of advantages that are the same. The fact that the two patterns have common advantages, gives the following conclusion:

1. The patterns have a number of advantages in common, yet they are excellent for different scenarios, and:
2. Based on the fact that the complexity of the extensible implementation pattern is higher than for the simple implementation pattern, and on the comparison tables between the two (tables 12, 13, and 14), we conclude that if an application uses the simple implementation pattern it can easily be upgraded to use the extensible implementation pattern if it needs to scale up.

Therefore, we consider the extensible implementation pattern the natural transition from the simple implementation pattern if an application that with the simple implementation pattern needs to scale up.

Chapter 8 - Conclusion

Based on the findings from case studies we can conclude that there are at least four multitenancy implementation patterns. Two (derived from Ledensite and AFAS Online) were determined after interviewing multitenancy experts and the other two (derived from Github and Salesforce’s Force.com) were determined by studying the literature.

To answer the main research question, on what architectural patterns are most suitable for multitenant software products, we first need to answer the sub-questions. In the first sub-question it was asked what the current multitenancy implementation patterns are. Based on the findings of the research, we identified four multitenancy implementation patterns. The second sub-question asked what the strengths and liabilities are of the multitenancy implementation patterns that were identified above. We not only identified the strengths and the liabilities, but we also suggested how the liabilities can be mitigated so the reader can see clearly what to expect from each pattern. The third and last research subquestion was answered by reviewing the implementation patterns and by pointing out which pattern is most suitable for solving different multitenancy related problems.

The answer to the main research question **“*What architectural patterns are most suitable for multitenant software products?*”** consists in the four patterns, the analysis of their strengths and liabilities (chapters 3, 4, 5), and the overview with the problems they solve (chapter 6).

Chapter 9 - Future Research

The purpose of this research was to identify multitenancy implementation patterns and to help the reader in choosing the most suitable multitenant business software patterns for multitenant software products. We have investigated four multitenant architectures, two by interviewing experts and two by studying the literature and ended up with four multitenancy implementation patterns. For each pattern we analyzed their advantages, their liabilities and in the end we showed how the liabilities can be mitigated by studying topics of relevant literature. After, we listed a number of problems that the patterns solve and explained which patterns are most suitable for solving certain problems.

Looking at table 20, the results are generally positive and this gives us confidence to say that there is room for future research on this topic. One of the first things we would advise the reader to do is to make the first two validity tests fully pass and then continue the discovery of new implementation patterns.

Future research is needed to go more in depth on the implementation patterns. We believe there are more patterns that can be identified and thus make the method more powerful. Moreover, for the patterns that we found, it would be interesting to find other companies that are already using them and investigate if they found other advantages, liabilities and how they solved their problems. Also, only the first implementation pattern was evaluated by an expert.

Additionally, it would be very interesting to start developing a multitenant application or to reengineer an existing single-tenant application to multitenancy by choosing one of our patterns. It would confirm that the advantages and the liabilities that we found are correct and also that the solutions to problems that we offer are complete.

In the end, anything that would help extending, evaluating and validating our findings we consider useful for future research. The findings in the current research are at the first iteration and we believe there are more patterns to be discovered.

Acknowledgements

Special thanks to dr. Slinger Jansen and Jaap Kabbedijk for supervising me throughout the entire process of my research. Their feedback, support and enthusiasm helped me to increase the quality of the research, by reviewing and discussing on various issues.

Appendix

1. Interview Ledensite

Interviewee: Peter Duijnste

Peter is the lead developer for Ledensite.com

What is Ledensite?

Ledensite is an organizational website. It’s meant for fully administration for members of small-medium-large organizations. It’s basically to take over all the administrative tasks from the organization itself. We started with the idea of just having a list of members which can be updated and seen by the members. Maybe some small communication between the organization and the members. Maybe newsletters or organizing events. We also wanted to include some financial aspects, membership contributions and that sort of things.

Hot did it started? Idea first or customers demanding the products?

It started from both sides. Slinger is a member of an alumni organization from his high-school. He was running into problems organizing events, and meetings. They were having problems collecting the fees from the members. He found a lot of programs that were solving this and they were very expensive. They started with the idea for programming it for his case and then for some other people.

How many tenants?

The central entity is the organization. Right now they have 3 paying and 3 in a beta-test.

When did Ledensite start?

The first time we signed the agreement that we were going to develop it was 3 years ago. But I think we actually started coding I think 2 years ago, something around that.

Any comments about the pricing model than from what is on the website?

They followed the standard approach with the pricing model. They call it the freemium for just trying the website. Then there is a basic step where you have your own page. You can add your logo but that is it. You also have the basic features, the list of members, newsletters. The advanced feature is where the website is fully customized.

How is software distributed?

Over the internet. Tenants run it in the browser and they’re not restricted to OS. It’s all HTML4 and CSS.

Do you consider the multitenant implementation was successful?

It is. The basic multitenant layer is fine the way it is. For some of the more complex resources they can be re-engineered. Right now the base entity is the organization (tenant) and everything is inherited from there. It might be useful to have some namespaces and levels in the future. One of the things that was not planned well in the beginning was the feature set which is a problem for all these kinds of projects. We started with a very lean feature-set and over time as we started approaching customers we added a lot of new stuff. One of the things is the events which essence is a good idea and we tried to program it too quickly instead of trying to see how it fits in the application.

Did you plan to add a hierarchical structure for the customers?

There was a client who had an organization which was a collection of other organizations. They wanted to be able to manage the tenants. Currently Ledensite is managing the tenant but they wanted an admin interface for multiple tenants at the same time, like a step in between. There were already a lot of concerns of making the code base too complicated or too unstructured, also the time period that this was necessary was too short and they couldn’t guarantee the quality of the code.

Does the software require internet?

Yes, it’s fully web-based. It can be used from tablets from the browser. There were no requests from tenants to have an app. They plan to offer an adaptive design for smaller screens. They are sure there are a lot of people who would like to use it from the mobile devices.

Did you have any problems with scalability, was the server too slow?

So far, not really. They do a little bit of caching. The number of tenants is still small and they don’t have these kind of problems, but the basic infrastructure for fragment and base caching and they can expand on that quickly if it becomes an issue.

How many servers are serving the tenants?

Currently there is one server that is serving all tenants. All tenants are served from one instance of the software. We discussed the idea of having a separate instance for each tenant. Then we would have the same infrastructure but instead of Ledensite being the owner, the administrator, the tenants’ core organization would be the administrator. We are still debating the idea, but that will also be a pretty big investment in figuring out how would you deploy that, how would you keep it maintainable. Currently there are still issues in the application itself, we decided to delay that decision.

Is tenants information stored in the same database?

Yes, all tenants information is stored in one central database.

Did the tenants have any concerns?

That has not been a point of discussions. Security-wise I didn’t hear anybody raise any concerns. It only takes a few errors to make data from a tenant available to other tenants.

Security issues?

Just script-kiddies.

Did you experience any data loss?

So far, no. We run database backups every day so it should be limited if that happens.

Do you categorize tenants data as critical, important or less important?

We treat all the data pretty much the same way. There is some encryption for sensitive data like account information (users’ accounts) but everything else is open. Most of the information is public, there are a few things that might be considered private, like the newsletter, but it is not so critical that it might require individual encryption. We discussed of offering a secure connection over https, which was my assumption that it will be a request from one of the customers pretty soon, but so far nobody has asked for it. One of the tenants is pretty technical but he doesn’t seem to care.

Another client, an organization of lawyers have their website where they use to publish information and thew wanted single sign-in functionality. So that if you login to their website you are automatically be logged in on Ledensite. So we assumed that you need to send back and forth login details and we encrypted that thing with bcrypt and shared salt and that sort of things but we sort of assumed that they would request https for those transactions but they didn’t seem to care really. It’s something I really wanted to include but not for now.

How about software updates?

For the backend and templates each update affects all tenants. The templates need to be customized and they are consistent.

Do tenants need to pay for updates?

No, the only exception is that if we implement a feature that are specific requested by a tenant.

If a new tenant requests a feature, will that become available for all tenants?

First we judge of course if it is applicable for the broader user base. If it is, we use it to improve the feature-set.

Is the software backwards compatible?

Up until recently it wasn't an issue. The main reason it became an issue was the single sign-on api which needs to be version maintained, but other than that we don't have any interoperability with the outside. Another thing was the import feature, the csv import, but other than that there is no real need to stay compatible except for our own database.

When a tenant logs in, is there a hardware or a software decision that determines which data he is allowed to access?

It is fully software-based.

Do you use any cloud services, like Amazon or Rackspace?

No. Once we get to the point of having enough tenants to make that cost-effective, we're definitely going to move to the cloud. But for the time being, because I already run my own server for different customers, it's just more effective to give it a little bit of room. It has its own vps, it has isolated space, but it is on a shared server because the amount of resources required is just very slow right now.

Does Ledensite provide ani API for outside developers?

No. Other than the one we just talked about, so the authentication api, apart from that we don't really have any outside access api. It is planned to expand on the accounts api and also allow querying on the users table, eventually more, but that's for the future.

Do you store more replicas of tenants data?

We don't have a failover server right now. We have the backups we can restore but that's about it.

How do you deal with long processing tasks, such as newsletters?

We talked about having workers running, but for the moment we just cope with cron jobs and rake tasks. The mailing is marked as ready for delivering and the next moment it gets delivered. The cron job runs every hour.

What database do you use? Any drawbacks?

MySQL. Although we looked at PostgreSQL for a while and they have a couple of features that might be interesting, for the moment we just decided to go with MySQL because it's more or less the standard, we don have the sort of volume that requires a certain type of type of database, and of course the migration from MySQL to PostgreSQL is very simple and we just use that for the time being. For this project and the other projects that I've been working on, the database has never been the bottleneck. The bottleneck has always been Rails that has more of a problem of serving large volumes than database queries.

Can you give me an estimate about the average number of queries to the database for a request?

That's really really different per view. It varies a lot. If you look at the profile page, you have the user but also the custom user fields. Actually that's not the lower ends. We noticed that, and that's something we found out recently that customers tend to ignore the default fields and make a huge list of custom fields. We were sort of expecting like 2, 3, 4 custom fields but it turns out that standard details like name, address, is not so interesting and they tend to make some custom fields, and sometimes they even duplicating the default fields because they want to have everything in the custom fields. Right now we're working on getting rid of all the standard fields. The downside is the number of queries this generates, because it means you have a many-to-many table for one customer and this could lead to 10 or even 15 hits to a table. But of course, we try to be smart about this and join everything together before querying. So, I'd say, if you look at one of the pages like the profile page, you're probably down to 5 or so queries. If you look at the bigger pages, for example the members page, where you can customize to see 50 results per page or so, there's a couple of instances where we still have the n+1 problem so you might have a dozen of queries for one view.

How and what can tenants customize on the website? How do you deal with metadata?

For some of the resources they can define their own fields. Mostly the layout customization is done through templating. So you can just install HTML templates, we use Liquid templating engine and we allow them to define a few custom tags for their organization. We have a list of standard settings for example, place the user menu here, place the login box here, we have liquid tags for them. All the metadata is stored in the database. The list of things that are customized per tenant: base domain name, logo, menu, per-tenant localization (coming soon), such as some organizations calling their members 'members' , others 'participants' which is important for the views. We're working to move away the localization from a yml file to the database as well, so each tenant can choose the default locale or update their own. That can also be used for the contact person for instance where we have a text which they might want to change as well. Instead of making everything editable fields, which is also possible, we felt like having a customizable locale might be interesting. So, the templates, soon the static text, the newsletters can be fully customized, the details of what kind of data is stored on members can be customized. Another thing that you can customize is meta-tags, such as key-words and meta description, and we have features to include trackers such as Google Analytics for each individual tenant.

Do you use any key-memory data store, such as Redis?

We use Memcached but it is pretty limited. It is used mostly for fragment caching right now.

Do you use content delivery networks (CDNs)?

No, it's not necessary right now.

Can you tell me anything about the authentication mechanism that you provide?

It’s very simple: it’s just a form where you enter the password and the email address. There is also the authentication API that can be seen as a 3rd party accessing tenants data on their behalf.

How is the software delivered?

It’s a website, no native applications / apps. The software is available only through browsers.

Do you host the software or each tenant has an individual copy of it?

We host the software, the hosting company is Hetzner.

Do you use any tools to check tenants’ behavior on the website, and how they’re using it (such as A/B testing)?

We do click tracking. Apart of installing Google Analytics for it, we also do internal log analysis. We also run the Rails logs through a profiler to see if there are problems that are causing big delays or if the database connection is really slow for some reason. We don’t use A/B testing mostly because in the time it has been online we had 2 requests from unknown organizations through the contact, the rest were convinced through directed selling rather than hoping for the visitors to come in and become clients.

Have you decided to adapt the website on tenants behavior, that you noticed from the logs?

So far we haven’t noticed anything that requires significant changes. Mostly optimizing the views here and there. Through the logs we discovered a couple of these n+1 problems (large number of queries), we discovered all cases where stuff needed to be moved in a separate task like the rake tasks. A good example is the CSV import, was in real-time and the website was waiting until the server sent back a response. Initially it was fine but the problems appeared when we added the custom fields that really slowed it down. Mostly it has been performance stuff, not user views.

2. Interview AFAS

Interviewee: Jeroen van Stokkum

Jeroen works at AFAS for over 10 years. Jeroen is the leader of the internal IT department as well as the SaaS platform at AFAS.

What is your main product?

AFAS profit is the main product AFAS produces, AFAS profit 2011 at the moment. You can run it on premise and you can also run it in the cloud (AFAS Online), in our SaaS platform. We did some tuning on it for the SaaS platform but it's over 99% the same software as it was before. It's not specifically designed for it but it runs well.

There is a new project, AFAS profit 2016 which is completely built from scratch and it doesn't matter where you run it. It is completely designed with multitenancy in mind, but right now we can serve customers the same way.

How is the software being distributed?

You can download or get a DVD of the software that you can install on premises. The application is web-enabled. We have two kinds of web entrances and the main product is used over RDP sessions in a remote application. That's more for the users who produce stuff, for instance, something in the controlling department who is doing all kind of financial mutations, or someone in the HRM area. The other part is the employee self-service. Those are more the uses who view stuff. They're using a web-browser for instance to ask for a holiday.

Do tenants need to communicate / share data with each-other while they are using the software?

Profit has a complete authorization model in itself and we don't offer other things than the application online. We don't say you have a home directory or something like that. We see data that you share with other tenants that would always be data of other tenants within your own organization. We don't share between the organizations but if this data is related to Profit, you should put it in profit although it should be a personnel file or a financial file so they put it in there and someone who is authorized can also use the file. There are no other things than that because people would never clean it up.

Can you use the software if you are not inside the organization?

Yes. It's completely available over the Internet. You need an internet connection in order to use it and it cannot be used without internet.

Do you offer a freemium service?

No, we have some demo account which sales people use to show things or to check if it works. We have some requirements because it is fully based on a Microsoft platform. In order to get some experience with the website, we have some demo accounts for that, but for the others, this product is so big, so you have this many options that you don't know what you can try, you can try some things but you don't know what you're doing if there was a part of the product offered for free. In your previous software you would do financial mutations and you would know how they worked but this works totally different in some ways so, you can try and do something and say that you don't like it. So we can say that we give you something and you should do some course before. When we target customers is more like a direct sale to them. We also support small businesses which can start using AFAS Online right away, and use product videos to understand how it works and to train, but we don't offer it for free for the first part.

Can you tell me anything about the pricing model?

We have some different pricing for the type of companies we have. Because it's an ERP product we offer logistics, payroll, taxes, financial projects. We have different types of companies. The bigger companies in the Netherlands use the HR and payroll products and they have some pricing model that is per user. There are two different types of users - the users who use the RDP application and who can access the full application, and the employees users, you getting an email your payslip is ready for you, so you logon and that's a total different model because you logon for like 5 minutes per month and the other users work 8 hours a day in the application. The employees have only web access and the other users have full access. Most of the tenants are logged on during the business hours and it depends per business on how much they use the application.

Can tenants customize how the application looks?

Yes. In the full application there is not that much to customize. You have all kinds of settings. The first is the authorization module where you see or don't see things. Then you have all kinds of customizable views or re-ordering tools. All the views you're looking at they're all customizable. Then you have the per-user settings where you can change for instance the color or the position of a column, so it's fully customizable.

How about custom workflows? Do you allow tenants to define custom workflows?

We have a workflows module where you can use workflows where you can for instance do holiday requests. Customers can define all kinds of workflows. Here in AFAS, everything is done in workflows and the module is very flexible. We fully use inside the company our own product.

Do you consider the implementation is successful?

Absolutely. We started with about 3 customers the first day, but from the first day until now we serve more than 4000 customers in 3 years. A part of it is new business because every new customer for who we start the implementation at the online product and they sign the offering, the next day they get a new login code, or even the same day so our consultants can start implementing right away. We don't have to wait for hardware at the customer side. And because it is the same software, the point where the customer goes live they start using the online software (AFAS online). When the hardware is ready on the customer premises, if they choose an own deploy, we transfer the backup to you.

Can you give me some numbers, such as a ratio of customers who have their own deploy vs the customers who use the online software?

We have a different ratio of the small customers which we call ‘profit small business’, that’s for a few hundred euros you have complete financial administration and invoice. For them everything is done inside the cloud. We use the same platform, the same service, the same sharing of resources and the other is the profit main product which is the full ERP suite, or they use a part of it but the numbers are a bit different because for the small business we have 57% online and 43% on premise, while for the the Profit, the big product is 20% online and 80% on premise. Overall it’s 40% online and 60% on premise. We see a big movement in there because in a few years it will be completely upside-down. In the 2016 product it will be online by default, and for the other way it will be a big discussion if you want to run it in your own environment. But it will still be possible, because companies like Ernst & Young they need it because they have all kind of regulations.

Were there any security problems with the software? Any attacks, any security breaches discovered?

No. We see all kinds of scanning and stuff but we do all kinds of tests, we do penetration tests with external companies, we run automated tests to check for known vulnerabilities. Because we have some different products and different we’re almost constantly under attack of our own company who does it for us because we release something new, we have to check it again but there hasn’t been a breach yet.

Did you have any scalability issues with the online software?

Yes, we had that and that’s kind of an annual thing which we try to break-through because we should be able to predict when the customers are doing heavy usage, like for instance this week is the week when everyone is processing all their payrolling because next week is the payslip week. So we had those problems, and those were partly our fault that they did occur because we had to scale up before it happened. We think the platform was ok but the hardware under it wasn’t. We run the software at the professional datacenter Leaseweb in Haarlem. We lease different services from them, like Infrastructure as a Service. We get complete machines with OS on them and we are further responsible for the whole stuff. We have our own private cloud. We made agreements on what types of hardware and servers the virtual machines are, and also how many virtual machines they can put on a physical machine. So we agreed on this type of server, it’s a database for us, so we need this amount of cpus and memory and you can only put two virtual machine on a host because we really need the performance for those two machines on the host. One thing we found out, we had some differences in the processing power and that was because there were other types of cpu’s in it with less caching, or the storage wasn’t growing fast enough, they had some lack of storage. Because of that they were running on some de-duplication problems and they had some performance issues. These are the kind of problems we ran into.

Did you expect any data loss?

No.

Do you categorize users’ data as important and less important?

It’s all important, we put it on highly available and expensive storage.

How many developers implemented the software?

Because it is a product that we normally use, we don’t have special developers for AFAS online. AFAS is a company with its developers and the products which have been developed over the past 10 years it was getting bigger and bigger and there were some customizations for it but you can’t really say that in terms of developers.

How was the software born? Did it begin with the customers asking for it, or the company who created it for its own needs and then sold it to customers?

AFAS was built from a management buyout from a company 15 years ago. They took the financial product with them, that was a product on a DOS. And also a payrolling application. Then they started developing their own Windows application. That was developed over the years and it’s still the basic for the core product. During the years some features were redesigned and more integrated than they were before but with the management buyout they also took over some of the customers. So it was never designed for our internal use, it was always designed for the customers to use.

How, and how often do you offer updates?

We have builds and updates. Builds are once or twice a year but it’s mostly once per year. We also have the profit updates which are about 6 times a year. Right now we upgrade the online product in one time. Two weeks ago in one weekend we had one big maintenance window during the night for about 6 hours and for the rest of the weekend all the database conversions were still running. When someone logged on and the database wasn’t converted yet to the new updates it would take about 2-3 minutes and then they would be able to continue to work. So we do one big update for all clients. For the tenants that have the software developed on their premises, we only support the last and the pre-last build and most of them will always have one of the latest releases because we are also bound to some regulations and they needed to have the latest changes. If one tenant uses an old version of the product, and then they decide to upgrade, then we offer them some release in-between to cope also with the data migration.

When a client logs in, the controller that decides what data the tenant can access, is it hardware or software based?

That’s software, everything is software.

How many instances of the software do you run?

AFAS online is spread over multiple servers. There is a load-balancer that decides which server handles a request.

Do you use virtualization on the hardware?

Yes, we only get offered the virtual machines, so everything is virtualized.

Have you considered using cloud services from Amazon or Rackspace?

Not from these specific parties, but we have another feature in the software to publish portals, like for instance, our AFAS.nl website is published from the ERP application and we publish those websites in Microsoft Azure and it has a connection to the back-office which can be on premise or at AFAS online and your customers can register for the customers self-service. The website is running on Azure. We are always looking for the best way for that. Because of the legacy part of the software we won't be able to use fully public cloud like Azure or Amazon and that's why we built our own private cloud, but for the 2016 product, it won't matter anymore, we can run it anywhere and we're always looking for that because all of these services they offer will only get more and more commodity, like for instance have your data at multiple places, your backup is all arranged for you. So we continue looking to the public cloud providers offerings. In a few years we want to be able to for instance just decide that we want to change our cloud provider, to just deploy our application on the new provider and continue serving it from there. Right now it is not possible but we want to use that because we like the idea of how it is made.

Do you offer an API for outside developers?

Yes, we offer XML webservices for SOAP calls. If you have the API license then you can do whatever you want. The webserver is developed to go through all the usual controls. So for instance if you would like to enter a new employee you get all these kinds of checks such as birthdate, or social security number that can be changed to required fields. Developers can program the webserver with the API to return validation errors for instance, for some fields. It is customizable on what kind of data you get out, you can design your own views, not the SQL views but also in the application part to decide what kind of dataset you want to pull out, but to enter data into the software it's all programmed what kind of types of actions you can do. Most of the things that have been asked for are possible to do through the API.

Do you use background queues to handle requests that could take a long time?

We put more and more stuff in our batch service which does some queueing and stuff so yes. For some big jobs, like the payrolling we offer it. We use our own built-in queueing system. We have a lot of batch services which are configured for multiple types of jobs. And they can pick a number of jobs at the same time and handle it and say to the customer your job is ready, you can for instance print out.

Do you store data from multiple tenants in the same database?

No, the software is designed to use a database for every customer. If a customer wants a test database or they have a second company, it will always create a new database for them. For now we have over 4,000 customers but we have over 12,000 databases. If there is an accountant which does the financial administration for 80 companies, there will be 80 databases. That is very flexible, because new databases can easily be added.

What database do you use, and what is your experience with it?

The database is Microsoft SQL. The database is our single point of failure. We can load-balance web services and other services but we cannot load-balance the database with

Microsoft SQL. From the performance point of view, you can’t load-balance it. For failover, you can do clustering but for performance when these customers get really big you can’t spread it over more services. So that’s a big issue which gets more relevant every day. We only store one replica of tenants data. It’s not like with MySQL where you can use masters and slaves and one for read and one for write that’s highly tunable, but for Microsoft SQL that’s more difficult, and of course, with 12,000 databases you won’t be doing those things. On the other hand, we always run on Microsoft SQL, we have good contacts with Microsoft so we’re always on top on the new developments and we almost always use the new developments, or the application takes advantage of all the new stuff.

How do you handle metadata? How do you organize it and where do you store it?

With the new update, you get new metadata. This is put in a metadata database from us. When we run conversions the metadata is synched to the database of the customer. Except from the database we also have some metadata in some files and that’s also spread over to customers. I’m not very happy with that because I store this same metadata for hundreds of times but it isn’t that much. When one customer will customize it, the metadata will always will be maintained. So they can always go back to how we delivered it.

Do you use any content delivery network?

Right now I’m running some tests with Akamai but that’s more for the kind of connection SSL they offer, not for the content delivery. Because it’s highly customizable, it’s not possible to do all kinds of caching. The other thing is, more a security concern, because we do everything over SSL, we would have to give them our own SSL keys to really have benefit from it. I don’t think we will use it for that but we will keep an eye on it because it might be useful in the future. Like for instance for the website, we could use it, but since most of our customers are in Holland, the connections are good here so there is not quite required at the moment.

Can you describe a little how the users login on the website?

After they sign-up, they get an email with their login credentials. For authentication we have different types: for one web application we still use the HTML authentication, for the other one we use the form authentication from Microsoft forms management gateway. For the RDP sessions over SSL we use use also HTML authentication. At this moment it’s only username & password but in the future we’re looking into multi-factor authentication like tokens, one-time passwords, and so on, and also some single sign-on federation features for customers to have a single sign-on from their own environment.

Does the software allow 3rd parties to access data on tenants behalf?

We have this collaborate licenses that are used for instance by the accountants. The customer who does its financial administration itself could say that someone is their accountant and they can in the authorization part add the other customer with another customer number and give him access to the application. Actually the accountant uses the same login but after he logs in he sees he has access to the new database as well, so he doesn’t have to logon again, still using the same platform.

Does the application provide a native and web client?

Yes, both. The native application can be used on Linux and OSX too, using the RDP sessions over SSL. That was the hardest part, the Microsoft Remote Desktop Gateway which terminates the SSL session for the RDP session it wasn't supported from the clients from OSX and Linux but now it is, so they need some piece of software one time so they can use it. And it is the same piece for mobile devices also, for iOS and Android.

Can tenants access their data from outside the company?

Yes, we have some customers who have restriction. They asked us to block their customers from accessing the software from home, only from their network. So we have IP restrictions for them.

Do tenants need a VPN connection in order to access the online software?

No, everything is just the SSL connection.

In order to run the software, does the tenant have to install anything?

They need RDP 6.1 at least for support for the SSL connection. For the printing features they need .Net 3.5. RDP protocol 6.1 offers the terminal server Easy Print driver which makes it easy to print based on the .net framework, so we don't have to install printer drivers on our side so every customer can use whatever printer they have and the client will tunnel it to the right printer driver.

References

Aulbach, S., Jacobs, D., & Kemper, A. (2008). Multi-Tenant Databases for Software as a Service: Schema-Mapping Techniques. Techniques. In Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008, pages 1195–1206. ACM

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., et al. (2003). Xen and the art of virtualization. (M. L. Scott & L. L. Peterson, Eds.)Memory, 37(5), 164-177. ACM. doi:10.1145/1165389.945462

Bayer, R. (1997). The universal B-tree for multidimensional indexing: General concepts. In T. Masuda, Y. Masunaga, & M. Tsukamoto (Eds.), (Vol. 1274, pp. 198-209). Springer Berlin / Heidelberg. doi:10.1007/3-540-63343-X_48

Bezemer, C., Zaidman, A.: Multi-tenant SaaS applications: maintenance dream or nightmare? In: Proceedings of the International Workshop on Principles of Software Evolution (IWPSE), pp. 88–92. ACM, New York (2010)

Bezemer, C., Zaidman, A., Platzbeecker, B., & Hart, A. (2010). Enabling Multi-Tenancy: An Industrial Experience Report. Innovation, 1-8. IEEE. doi:10.1109/ICSM.2010.5609735

Briscoe, G., & Marinos, A. (2009). Towards community cloud computting. Proceedings of the IEEE Digital EcosystemS and Technologies (DEST 2009), Istanbul, Turkey.

Chong, F., Carraro, G., Wolter, R. (2006). Multi-Tenant Data Architecture. Retrieved February 10, 2012, from http://msdn.microsoft.com/en-us/library/aa479086.aspx

Chong, F., Carraro, G. (2006). Architecture Strategies for Catching the Long Tail. Retrieved February 19, 2012, from http://msdn.microsoft.com/en-us/library/aa479069.aspx

Curino, C., Jones, E., Popa, R., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., Zeldovich, N. 2011. Relational Cloud: A Database Service for the Cloud. In CIDR, pages 235–240.

Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. Social coding in github: Transparency and collaboration in an open software repository. In CSCW (2012).

Florescu, D., Kossman, D. 1999. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Technical report, Inria, France

Force.com success stories. http://www.salesforce.com/platform/success_stories. Accessed on May 14, 2012.

Force.com Whitepaper (2009a). A comprehensive look at the force.com cloud platform. http://wiki.developerforce.com/index.php/ A_Comprehensive_Look_at_the_Force.com_Cloud_Platform. Accessed on March 27, 2012.

Force.com Whitepaper (2009b). The Force.com multitenant architecture.http:// www.apexdevnet.com/media/ForcedotcomBookLibrary/ Force.com_Multitenancy_WP_101508.pdf. Accessed on March 27, 2012

Hamilton, J., R. 2007. On designing and deploying internet-scale services. In Proceedings of the 21th Large Installation System Administration Conference, LISA 2007, Dallas, Texas, USA, November 11-16, 2007, pages 231–242. USENIX

Hayes, B. (2008). Cloud Computing. Retrieved February 19, 2012 from http://doi.acm.org/10.1145/1364782.1364786

Hevner, A.R. (2004). Design Science in Information Systems Research. MIS Quarterly, 79-99

Homakov, E. (2012). How-To. http://homakov.blogspot.com/2012/03/how-to.html. Accessed on April 30, 2012.

Kabbedijk, J., & Jansen, S. (2011). Variability in Multi-tenant Environments: Architectural Design Patterns from Industry. In O. De Troyer, C. Bauzer Medeiros, R. Billen, P. Hallot, A. Simitsis, & H. Van Mingroot (Eds.), (Vol. 6999, pp. 151-160). Springer Berlin / Heidelberg. doi:10.1007/978-3-642-24574-9_20

Kajornboon, A. B. (2005). Using interviews as research instruments. E-Journal for Research.

Krasner, G., Pope, S.: A description of the model-view-controller user interface paradigm in the smalltalk-80 system. Journal of Object Oriented Programming 1(3), 26–49 (1988)

Jacobs, D., & Aulbach, S. (2007). Ruminations on Multi-Tenant Databases, 1-5.

Jansen, S., Houben, G.-J., & Brinkkemper, S. (2010). Customization Realization in Multi-tenant Web Applications: Case Studies from the Library Sector. In B. Benatallah, F. Casati, G. Kappel, & G. Rossi (Eds.), (Vol. 6189, pp. 445-459). Springer Berlin / Heidelberg. doi:10.1007/978-3-642-13911-6_30

Li, X., Liu, T., Li, Y., & Chen, Y. (2008). SPIN: Service Performance Isolation Infrastructure in Multi-tenancy Environment. In A. Bouguettaya, I. Krueger, & T. Margaria (Eds.), (Vol. 5364, pp. 649-663). Springer Berlin / Heidelberg. doi:10.1007/978-3-540-89652-4_58

Liu, D. (2011). What's the difference between Salesforce.com and Force.com? http://www.quora.com/Whats-the-difference-between-Salesforce-com-and-Force-com. Accessed on April 1, 2012

Jin, H., Ibrahim, S., Bell, T., Gao, W., Huang, D., Song, W. (2010). Handbook of Cloud Computing. Chapter 14: Cloud Types and Services

Ma, D. (2007). The Business Model of "Software-As-A-Service". IEEE International Conference on , vol., no., pp.701-702, 9-13

MacDonald, N. (2010). Gartner: Virtualized Servers Less Secure Than Physical Ones. http://news.idg.no/cw/art.cfm?id=63EB65D1-1A64-67EA-E490D909BD42DBE0. Accessed on 30 April, 2012.

Media Temple. 2007. Anatomy of MySQL on the GRID. http://weblog.mediatemple.net/2007/01/19/anatomy-of-mysql-on-the-grid/#more-192. Accessed on April 3, 2012

Mietzner, R., Metzger, A., Leymann, F., Pohl, K. (2009) Variability Modeling to Support Customization and Deployment of Multi-Tenant- Aware Software as a Service Applications, In Proc. of ICSE Worksh op on Principles of Engineering Service Oriented Systems 18- 25. IEEE Computer Society, Los Alamitos (2009)

O’Leary, A. (2004). The Essential Guide to Doing Research. London, SAGE Publications.

Parlione, M., Charlesworth, C. (2008). Securing a multitenant SaaS application. Authentication and authorization with Spring Security and Apache Directory Server.

Preston-Werner, T. (2009). How We Made GitHub Fast. https://github.com/blog/530-how-we-made-github-fast. Accessed on April 30, 2012.

Preston-Werner, T. (2012). Public Key Security Vulnerability and Mitigation. https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation. Accessed on April 30, 2012.

Poddar, I., Carew, D. (2009). Software as a Service: Create multi-tenant workflows using WebSphere sMash Assemble Flow, Part 2. http://www.ibm.com/developerworks/offers/lp/demos/summary/wesd-saaswsmash2.html. Accessed on April 30, 2012

Rief, J., Horman, S., 2010. ldirectord. http://horms.net/projects/ldirectord/. Accessed on April 30, 2012.

Robson C (2002) Real World Research. Blackwell, (2nd edition)

Ruby, S. 2012. Agile Web Development with Rails (4th edition). The Pragmatic Bookshelf.

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering , 14, 131-164.

Schiller, O., Schiller, B., Brodt, A., Mitschang, B.: Native support of multi-tenancy in RDBMS for software as a service. In: Proceedings of the 14th International Conference on Extending Database Technology, pp. 117–128. ACM, New York (2011)

Schmidt, D. C., Fayad, M.E., Johnson, R.E. 1996. Software Patterns, Communications of the ACM, Volume 39 Issue 10

Singh., K, Castiglion, T., Dymoke-Bradshaw. L., Hanninen P., Ranieri V., & Kappeler P. (2010). Security on the Mainframe. IBM Redbooks.

Shao, Q. (2011). Towards Effective and Intelligent Multi-tenancy SaaS.

Svahnberg, M., van Gurp, J., & Bosch, J. (2005). A taxonomy of variability realization techniques. Software: Practice and Experience 35(8), 705–754.

The Force.com Multitenant Architecture. 2008. http://wiki.developerforce.com/page/Multi_Tenant_Architecture. Accessed on May 14, 2010.

Tsai, W., Shao, Q., Huang, Y., Bai, X. 2010. “Towards a Scalable and Robust Multi-Tenancy SaaS,” in Proceedings of the Second Asia-Pacific Symposium on Internetware. P.8. ACM

Wanstrath, C. (2009). Unicorn!. https://github.com/blog/517-unicorn Accessed on April 30, 2012.

Webster, J., & Watson, R. (2002). Analyzing the Past to Prepare for the Future: Writing a Literature Review. Management Information Systems Quarterly, 26(2).

Weerd, I. V. D., & Brinkkemper, S. (2008). Meta-Modeling for Situational Analysis and Design Methods. (M. R. Syed & S. N. Syed, Eds.) (pp. 38 - 58). Hershey: Idea Group Publishing. doi:10.4018/978-1-59904-887-1

Weissman, C. D., & Bobrowski, S. (2009). The Design of the Force.com Multitenant Internet Application Development Platform. In Proc. of the 35th SIGMOD int. conf. on Management of data (SIGMOD), pages 889–896. ACM.

Wiegers, K., E. (2009). Software Requirements, Second Edition. Microsoft Press.

Warfield, B. 2007. Multitenancy can have a 16:1 cost advantage over single-tenant. http://smoothspan.wordpress.com/2007/10/28/multitenancy-can-have-a-161-cost-advantage-over-single-tenant/. Accessed on April 29, 2012.

Wainewright, P. (2012). Security risks of multi-tenancy. http://www.zdnet.com/blog/saas/security-risks-of-multi-tenancy/1007. Accessed on April 30, 2012

Wiesner, M. (2011). Introduction to Spring Security 3/3.1. http://www.infoq.com/presentations/Spring-Security-3. Accessed on April 30, 2012

Winters, S. (2011). Virtual Security Gateway Multi-Tenancy Example White Paper. Retrieved February 22, 2012 from http://communities.cisco.com/servlet/JiveServlet/previewBody/21169-102-1-35964/VSG_VNMCI_multi-tenancy_white_paper.pdf

Yin, R. K. (2003). Case Study Research Design and Methods, 3rd edition. (L. Bickman & D. J. Rog, Eds.)Journal of Advanced Nursing (Vol. 44, pp. 108-108). Sage Publications. doi:10.1046/j.1365-2648.2003.02790_1.x