Versioning for Software as a Service in the context of Multi-Tenancy

Maximilian Schneider and Johan Uhle

July 2013
University of Potsdam, Hasso-Plattner-Institute
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
{maximilian.schneider,johan.uhle}@student.hpi.uni-potsdam.de

Abstract. In this report we present our findings on how to provide different versions of a SaaS to multiple tenants at the same time. We present a categorization of versioning approaches based on the layer of the SaaS application stack they are implemented in. We then evaluate them against our own experiences in developing SaaS applications.

Keywords: computer science, cloud computing, multi-tenancy, saas, versioning, web application

1 Introduction

The recent trend of providing Software as Software as a Service (SaaS) instead via the classical retail channels has changed the way that software is versioned. Previously software was released infrequently and the customer would choose when to switch to a different version after it was released. But today a Software as a Service provider is able to release software updates continuously and deploy them without further action from the customer.

Improving on that, providers may even choose to provide multiple versions at the same time in order to compare these. A common criterion is testing changes in the user behavior towards business goals. An example for this is A/B testing [7]. In these test a subset of the provider's customers is confronted with a different version of the interface. This enables the provider to measure statistically which influence the difference between the versions has on the business based on the behavior of the different split groups. Another criterion is the technical impact of a change. This might be regarding the correct behavior of new changes as well as potential performance degradation. Both can be tested on a subset of the users in order to verify that they are actually feasible in the production environment.

But SaaS providers do not only provide multiple versions of the same software for their own benefit. It is a common phenomenon that users of a software may be hesitant or even reluctant to adopt a new version. Among the reasons are that a certain version does not provide enough value to the customer to justify switching. Further reasons are technical incompatibilities between interfaces for example if a customer uses an extension which depends on deprecated Application Programmable Interface (API) functionality. In these cases the SaaS

provider will have to keep the old version available for the customers reluctant to switch to the newer one.

All these reasons lead us to our research question: How might a SaaS provider serve different versions of their software to multiple tenants at the same time? In the following, we discuss terminology (Section 2) and related work (Section 3), explain our different approaches regarding the architecture (Section 4) and evaluate them against our previous experience (Section 5). We finish with an outlook on future work (Section 6) and the conclusion (Section 7).

2 Terminology

2.1 Multi-Tenancy

Software as a Service products are usually built as multi-user systems. In multi-tenancy, these users belong to one tenant and need to be isolated from other tenants' users. Each tenant has hereby access to an isolated multi-user system [3]. To sell the products to a wide variety of tenants the software is usually heavily customizable and secured by Service Level Agreements [2].

Multi-Tenancy itself should not be confused with multi-tenancy architecture which differs from multi-instance architecture by servicing multiple tenants from a single instance of the software [12] in contrast to providing a dedicated instance of the software to each tenant.

2.2 Version

Considering software versioning two different definitions exists. One is the *product version*, which identifies a certain stage of the software in a release lifecycle. The other is a *code revision*, which is managed by a version control system as part of software configuration management [11].

As the latter definition in contrast to the former is mainly dealing with changes to the source code, differences in code revisions might not be distinguishable by the users, whereas product versions are inherently visible to the users. The distinction between product version and code revision will be made again in Section 4.5. When we talk about version without specifying the kind, we refer to product versions.

3 Related Work

As introduction to the topic of Software as a Service, reading the paper [3] is advised as it discusses the underlying business model and proposes attributes to classify SaaS.

An argument for the need to run multiple versions of a SaaS due to legislative reasons is made in [2]. The authors also discuss that for a multi-tenant architecture the software itself has to be written version-aware in order to support customizing. This in turn increases the code complexity, compared to the

possibility of creating a new instance of the application for each version in a multi-instance architecture.

The topic of schema versioning has been discussed by [10]. Schema evolution has also been implemented in the application layer in [14] [6].

A schema versioning scheme has been implemented prototypically in a main-memory database management system by [1]. Their approach is motivated by the idea that schema handling should be the responsibility of the database layer instead of the application layer.

4 Architecture

In this section we first clarify our assumptions about the *application stack* of a SaaS, then we explain the different architectures to implement SaaS versioning.

One important assumption we make for versioning is, that each tenant and therefore also all users of a tenant use the same version at the same time. In practice this means that each tenant has a dedicated migration point in time at which they decide to switch the version. This switch affects all their users at the same time. Users aren't allowed to individually choose their version.

We furthermore assume that all requests to the SaaS are authenticated, thus there is always a user as well as a tenant and its version associated with each request.

4.1 Application stack

To understand the architecture better, we first want to look at the SaaS application stack as displayed in Figure 1. We derived this stack from our experience with developing SaaS ourselves as well as stacks we have seen in related papers as depicted earlier in Section 3.

We assume a separation between the front end, back end and database layer. The *front end layer* is mainly concerned with the user interface. In a SaaS it will usually be delivered to the user's browser within a HTTP request/response cycle. Another possible implementation for the front end layer is a webservice API that enables external applications (e.g. third-party or native applications) to communicate with the SaaS. We handle this more in-depth in Section 4.4. The *back end layer* is concerned with the application and business logic. It handles requests of the front end layer and consequently communicates with the *database layer*. To allow for horizontal scaling, the front end and back end layers are usually implemented stateless. All state is then kept in the database layer.

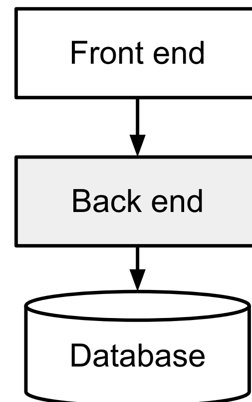

Fig. 1. Simplified Application Stack

To focus our research, we omitted some details of the stack. We were not concerned with neither caching nor load balancing. Furthermore we omitted how user management is done, especially with regards to authentication.

4.2 Multi-instance

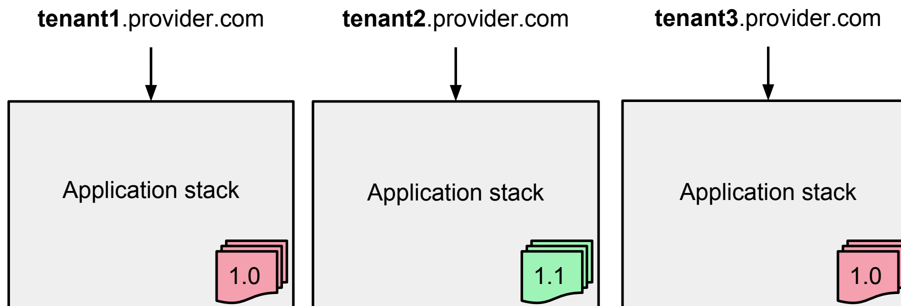

Fig. 2. Example for multi-instance with the respective software version for each tenant deployed on their instance

In a multi-instance architecture the whole application stack is deployed several times on dedicated resources. You can see an example in Figure 2. The multi-instance architecture serves the purpose of physical isolation of the tenants by provisioning resources for each tenant individually. Also it allows for developing a single-tenant application instead of building multi-tenancy awareness into the application.

This approach can also be used for versioning, where we see two variants:

Single instance per tenant When every tenant has their own application stack instance running, these instances can be versioned by deploying the respective software version on the tenant's instance. See Figure 2 for reference.

Single instance per version with multiple tenants This is a hybrid approach where each instance has a specific version and the instances in turn can be hosts for multiple tenants by using the shared-instance mechanisms explained in the following section. See Figure 3 for reference.

Using a multi-instance architecture has the benefits that the developers do not have to deal with issues concerning multi-tenancy in the application code. One of the drawbacks is the bad resource consolidation factor, thus if e.g. one tenant uses many resources but another uses none, the unused resources can't be allocated to the spiking tenant easily. The maintenance cost increases with the number of versions and tenants and also there is then no economy of scale working in favor of the SaaS provider.

Versioning-wise the actual application stack is not version-aware and thus can be built without versioning as a concern. Instead versioning is handled on

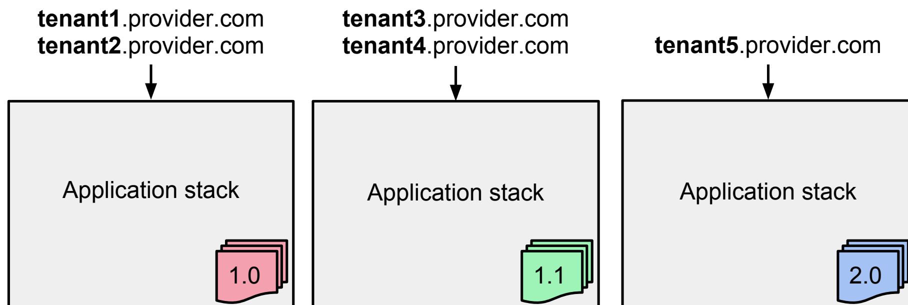

Fig. 3. Example for a hybrid multi-instance approach which is multi-instance regarding the version, but shared-instance regarding the tenants

the deployment level. Migrations between versions might then need significant operations engineering effort. Especially in the hybrid approach when data has to be migrated across application stacks.

4.3 Shared-instance

The shared-instance architecture consist of one software stack deployment that serves all tenants at the same time. This architecture is the one that is generally referred to when talking about multi-tenant SaaS applications, since it uses the economy of scale well due to its high consolidation factor [8] [2] [4].

To investigate versioning in the shared-instance architecture, we will individually inspect the layers of the application stack as outlined in the previous section, consisting of the front end, back end and database layer. For each layer we will outline how versioning can happen. We are closing this section with a look at how to migrate between versions.

4.4 Shared-instance: Front end layer

We split the *front end layer* into two categories: a *web application* running in the user's browser and an *API consumer* e.g. running on the user's mobile phone.

4.4.1 Web application In web applications the user's browser is usually reloading the whole web page on each request, which mostly is triggered by an interaction of the user with the system[1]. Thus there is a tight coupling between the front end and the back end layer, from which we conclude that they are both versioned simultaneously. Since the front end software is delivered to the user's browser on each request, the versioning concern can be completely handled on the *back end layer* as outlined in the next section.

[1] We assume that caching works transparently and perfectly as well as "One page applications" reloading their assets automatically ("hot code reload") in case something on the back end layer changes, as explained in `http://www.meteor.com/blog/2012/02/09/hot-code-pushes`

4.4.2 API consumer To allow programmatic access to the SaaS functionality, SaaS providers usually implement a webservice. A common occurrence are HTTP APIs following the REST principles [5]. These APIs allow native applications (e.g. mobile phone applications) or other webservices to build on top of the SaaS. The SaaS and their API consumers are loosely coupled. They follow their own product cycles. In case of compatibility breaking changes in the SaaS API, the API consumers have to adapt to the changes and deploy a new version to their users. This might be hard to do e.g. because users might be reluctant to updates, delivery cycles might be too long or the API consumer developer might not have the resources to update their product. This leads us to the conclusion that SaaS APIs should be able to provide several versions at the same time. The API consumers choose which version of the API they want to use. There are many ways to handle versioning APIs and explaining them in-depth is out of the scope of this report. Examples are version numbers in the URL or *HTTP Accept Header* of a request [9], or feature flags for the consumer application in the SaaS [13] (as explained further in Section 4.5.2).

4.5 Shared-instance: Back end layer

For versioning the back end layer we explore two variants: *1:1* and *1:n*.

4.5.1 Shared-instance with 1:1 mapping The *1:1 mapping* depicts that each product version is mapped to one code revision. Figure 4 shows the architecture. Thus to support different product versions at the same time, the respective code revisions have to be deployed on different back end servers. On each request, the user management decides to which back end servers the request is routed, depending on the tenant the user belongs to.

The benefit of this approach is, that the versioning is happening outside of the application code, thus the code does not have to be version-aware. On the downsides, the consolidation factor of this approach is low, given that the load is split between non-consolidateable versions. This deficit increases with the increase of versions running in parallel. Furthermore this leads to a heterogeneous deployment which is more complex than a homogeneous deployment. Also this can lead to problems like bugfixes, that if written once have to be merged and deployed into every running version. Also this approach needs an intelligent routing layer (in our example above the user management) which decides on which app server to map which request.

4.5.2 Shared-instance with 1:n mapping The *1:n mapping* depicts that each code revision maps to all available n product versions. As shown in Figure 5 each application server has the same code revision deployed and is therefore able to decide which product version to serve for each request. One way to implemented this is with feature flags. Listing 1.1 shows a code example in Ruby. In the beginning of this section we assumed that every request has a user attached to it. Furthermore we assumed that all users of a tenant share the

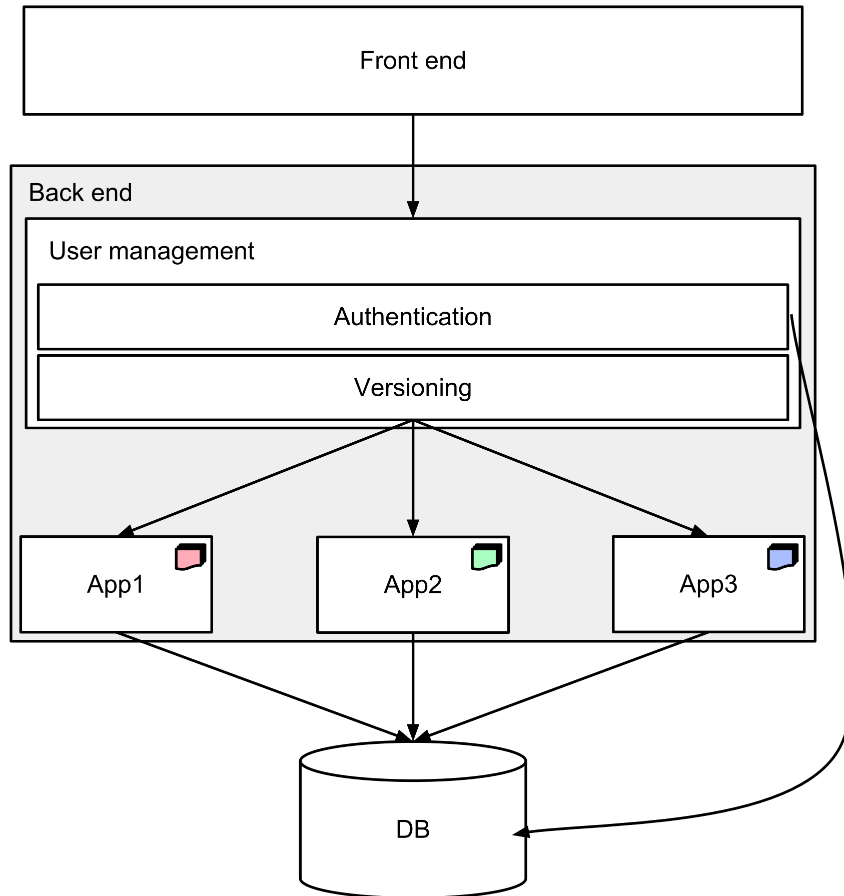

Fig. 4. Example for a shared-instance architecture with mapping each back end server to a code revision which in turn 1:1 maps to a product version

same version. This leads us to the conclusion, that the backend can determine the version for each incoming request.

The *1:n mapping* approach has several benefits: The consolidation factor is high, since all app servers can serve all requests. The deployment is homogeneous over the app servers, thus making operations easier. Also it is possible to version more fine-grained, not only based on product versions but actually on feature level and even feature version. This opens up a tenant-feature matrix which allows versioning which is more flexible than the traditional linear versioning. On the downside, the feature flags in the code increase code complexity, which might increase development time and increases the probability of bugs. Also the feature flags are spread over the code and removing them needs software development effort. Thus abandoning old versions is connected with extra cost.

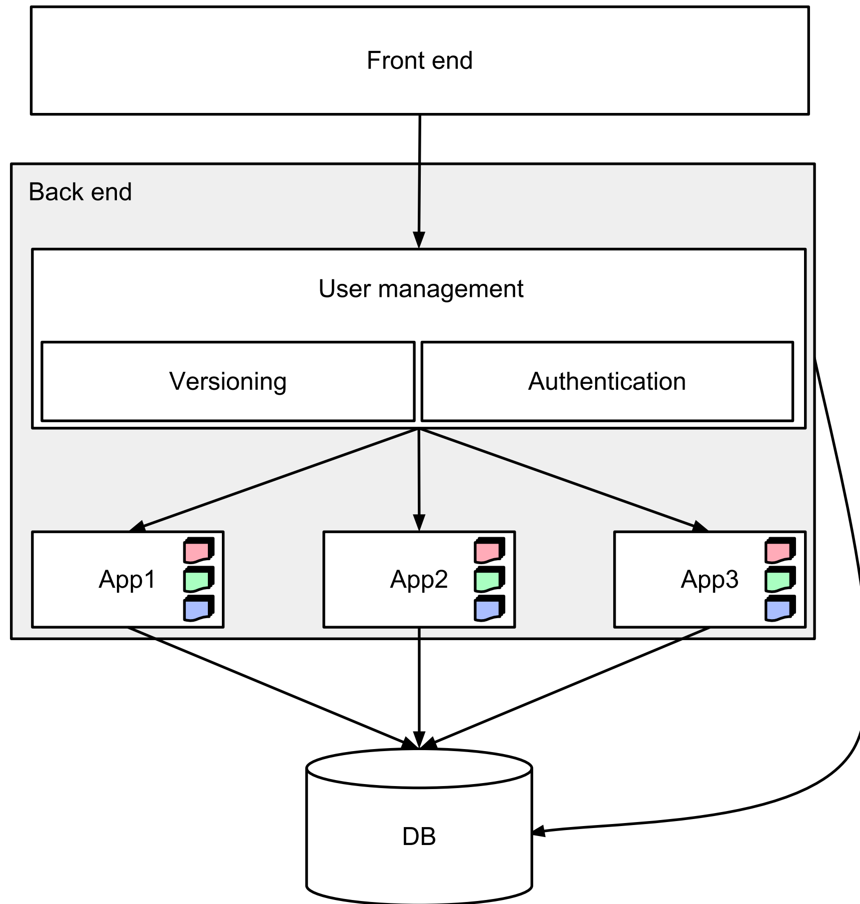

Fig. 5. Example for a shared-instance architecture with mapping each back end server to a code revision which in turn can server several product versions

Listing 1.1. Example code for *1:n mapping* on an application server

```
if user.has_version?("1.0")
  do_something()
end

if user.has_version?("1.1")
  do_something_a_bit_different()
end
```

4.6 Shared-instance: Database

The database is where state is kept. We assume that the database is a relational database, in that it keeps the data in tables with specified schemas to ensure the format and validity of the data.

In this section we will differentiate between two major cases regarding the database when versioning SaaS: *Different versions share a compatible database schema* and *different versions require different incompatible schemas.*

4.6.1 Different versions share a compatible schema When different versions share the same database schema, the database is actually agnostic to versioning. The changes of the versions happen in the other layers and thus never propagate down to the database. The same happens if changes are needed for one version, but the changes are compatible with the other versions. An example for this is the addition of new columns or tables. These can usually be ignored by versions that don't need these columns or tables, thus enabling compatibility with the changed schema.

4.6.2 Different versions require different incompatible schemas When schemas change between versions in an incompatible way, the database layer has to be versioned. Current database management systems (DBMS) only support one schema at a time per table. The concept of an updatable view is theoretically able to express this. However in our research we learned that implementations in current DBMS are not, e.g. when using aggregate functions.

As we did not find a DBMS that provides version-aware schemas, though this topic has been already researched [1], we investigate this option more in Section 6. Since the database layer can currently not handle the versioning concern, the responsibility lies with the back end layer. Next we will investigate two approaches for that.

Different tables for each schema To store different schemas, a new database or table can be created for each version. An example with different tables can be seen in Listing 1.2. Each tenant is on a specific version, thus their data is stored in the corresponding table for their respective version.

Listing 1.2. Example SQL code to create queries for supporting three versions of the users table at the same time

```
CREATE TABLE users_v1(id int(11));
CREATE TABLE users_v2(id int(11), username varchar(150));
CREATE TABLE users_v3(id int(11), user_name varchar(200));
```

With this approach migrations between versions require moving the data of a whole tenant from the tables of one version to the tables of another version. This might include columns or even whole tables that were not changed between versions. This copying introduces a significant overhead which current DBMS are not optimized for.

pivot_int

tenant_id	table	col	row	int
1	1	0	0	100
1	1	3	0	02123456789
1	1	4	0	25
1	1	0	1	101
1	1	3	1	02123456788
1	1	4	1	34
2	2	0	0	200
2	2	1	0	123456
3	3	0	0	150
3	3	3	0	07123456789
3	3	4	0	28

pivot_str

tenant_id	table	col	row	str
1	1	1	0	Joseph
1	1	2	0	Richard
1	1	5	0	male
1	1	1	1	Sarah
1	1	2	1	Smith
1	1	5	1	female
2	2	1	0	David
2	2	2	0	John
3	3	1	0	Sam
3	3	2	0	Zen
3	3	5	0	male

Fig. 6. Schema of pivot tables as shown in [15]

Pivot tables Pivot tables as explained in [15] [1] [14] follow the idea of not using a fixed schema in the database but keeping a flexible schema in the application. An example table layout can be seen in Figure 6. With pivot tables the DBMS is used more like a key-value store and in turn the application has to handle concerns that traditionally belonged to the DBMS, like query optimization or caching.

4.6.3 Migration of data When a tenant decides to move to a new version and the new version requires a schema change, then the data of the tenant has to be migrated from the old to the new schema. Next we will describe two ways for running these migrations:

Batch-processing migration The data is migrated in one go. Depending on the database management system, the schema changes and the amount of data this process might take significant time. Depending on the algorithm used to execute the migration the database and therefore the whole application might be completely unavailable or in read-only state for the tenant's users during the migration. An example for a migration process that has low interference with the operations of the application is presented in [6].

On-the-fly migration The data is migrated when it is requested or written to or from the database. This could mean that a semantic group of data is migrated in batch when a certain event is triggered e.g. all data belonging to a user is migrated on log in.

5 Evaluation

We evaluate our findings of the previous sections with our experience in developing SaaS systems ourselves.

Project A was a multi-user SaaS with over 10 million registered users. It served a website as well as a REST API from a Ruby on Rails web application backed by MySQL servers. At one point it was decided to build a new version of the website, mostly to implement fundamental changes in the user interface. Also old features were removed or fundamentally changed as well as new features added. A team was elected to work on the new website, while another team continued to maintain the old website. Furthermore it was decided to build the new version as an API client on top of the current REST API. A new code base with own deployment independent of the current website was created. During development it showed that the REST API did not support all features needed by the new version, thus the REST API was extended to support new endpoints. Also some schema changes were necessary, but they were either additions to existing tables or new tables. The Ruby on Rails application used Active Record as an object-relational mapping (ORM) to access the database. Active Record handles additions to the schema in a forward-compatible way. Thus the schema changes needed for the new website were 100% backwards compatible with the old website.

Once deployed, the new version was gradually rolled out the the users. Users were able to opt-in to the new version, but also to switch back to the old version. Anonymous users were initially always served the old version, but once the new version was officially launched all anonymous users got served the new version. The respective version per user was saved in a cookie. When a user requested the website, the web server receiving the request decided based on the cookie information, to which application server the request should be routed.

The versioning in this example happened outside of our architecture proposals in the previous Section 4, since the new version became an API client of the old version. It still relates to the shared-instance 1:1 approach from Section 4.5.1, since it has application servers per version and a routing layer routing requests based on versions. Also one of the insights it validates is that schema changes are possible between versions if they happen in a compatible way.

Project B was a multi-tenant SaaS written in PHP using MySQL servers as DBMS. During the first years after the launch a lot of feature requests from their early customers were directly implemented. But when the SaaS reached 30,000 users, it had become too complex to appeal to new customers. It was decided to rewrite the application in Ruby on Rails in order to simplify it feature-wise and to redesign the user experience. For the purpose of isolating the two versions from each other both were deployed to different application stacks.

As the ORM Active Record could not be adapted to the old schema, the new version relied on a new schema instead. Old customers were able to migrate all their data from the old to the new system. The other direction, migrating

back to the old version, was nevertheless unsupported, because it was regarded to complex to be solved under economic constraints. The goal was to allow productive use from the first minute on, so that the customers would not have to wait in order to try out the new version. But for large customers this data transfer lasted for multiple hours. Therefore their data had to be copied asynchronously into the new schema, prioritizing important and regularly used data over the remaining majority of historic and statistical data.

The new version was in the beginning accessed over a separate subdomain and it can be classified as a hybrid multi-instance architecture from Section 4.2 with versioning implemented on the access layer. Each version was deployed to a different application stack but the data of the old version was replicated to the database servers of the new version in order to allow for fast on-the-fly migration. The resulting redundancy of data and the completely not consolidated application servers verify our findings about multi-instance architectures.

6 Future Work

In our research we made two assumptions that simplified our setup but could yield interesting future work:

Switching versions per user In Section 4.1 we assumed that a tenant chooses the version for all their users at the same time. Future research should investigate how versioning on the user-level could be implemented. Especially interesting would be handling of schema changes as users are not isolated from each other in contrast to the isolation between tenants.

Caching To build a SaaS that is able to serve many users, caching is an essential technique. In our research we omitted caching. It would be interesting to investigate caching more with regards to versioning, especially if cached objects have to be invalidated during version changes or not.

One of our insights from Section 4.6 was that versioning with schema changes can currently not be implemented as a concern on the database layer, but instead has to be handled in the application layer. Depending on the implementation this might also mean that the schema is not kept by the DBMS anymore, but rather by the application layer. This might lead to the application not benefiting from DBMS mechanisms like indices or caching anymore. We believe that further research in the direction of version-aware database management systems could be interesting for future work. An example would be that the applications could specify via a SQL extension which version of a table they want to access. The DBMS would then take care of using the correct schema as well as arrange the base data itself, thus relieving the application layer of that concern.

7 Conclusion

In this report we investigated options on how SaaS providers might serve different versions of their software at the same time. We described in Section 4 how versioning can be implemented on different layers. We see three architectural points where the versioning concern can be handled:

- On the access layer with a multi-instance architecture as described in Section 4.2
- On the back end routing layer in a shared-instance architecture as described in Section 4.5.1
- On the back end application layer within the application code in a shared-instance architecture as described in Section 4.5.2

Implementing versioning on the access layer with a multi-instance architecture provides low consolidation factor, but also makes versioning an operational problem that can be easily handled if there is a small number of versions and users. Implementing versioning within the application code provides the highest consolidation factor, but also requires most engineering effort.

In our research we noticed that versioning is generally thought of as linearly increasing version numbers, with several feature changes bundled into each version. This concept comes from the traditional way of physically shipping software to the users. With SaaS instantaneously shipping software became possible. This technically obsoleted the need to bundle features into one big release. Instead it is now possible to ship features individually. The concept of feature flags (as explained in Section 4.5.2) then enables a fine-grained control over which tenants and users get which features, thus enabling versioning in a two-dimensional user-feature space instead of the traditional linear versioning. This approach enables faster iteration times and higher adaption to the users' needs.

All approaches to versioning that involve incompatible schema or data changes have to consider how to migrate data between versions. These migrations need a considerable engineering effort to reduce service disruption and service performance degradation.

To conclude, we think that versioning is an engineering problem that has already been solved for many cases. Still especially with regards to versioning schemas in the database, further research is needed.

References

1. Aulbach, S., Seibold, M., Jacobs, D., Kemper, A.: Extensibility and Data Sharing in evolving multi-tenant databases. 2011 IEEE 27th International Conference on Data Engineering pp. 99–110 (Apr 2011), `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5767872`
2. Bezemer, C.P., Zaidman, A.: Multi-Tenant SaaS Applications: Maintenance Dream or Nightmare? Proceedings of the Joint ERCIM Workshop on Software Evolution EVOL and International Workshop on Principles of Software Evolution IWPSE pp. 88–92 (2010), `http://dl.acm.org/citation.cfm?id=1862393`

3. Chong, F., Carraro, G., Microsoft Corporation: Architecture Strategies for Catching the Long Tail (2006), http://msdn.microsoft.com/en-us/library/aa479069.aspx
4. Chong, F., Carraro, G., Wolter, R.: Multi-Tenant Data Architecture (2006), http://msdn.microsoft.com/en-us/library/aa479086.aspx
5. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures (2000)
6. Keddo, R., Bielohlawek, T., Schmidt, T., SoundCloud: Large Hadron Migrator (2011), https://github.com/soundcloud/lhm
7. Kohavi, R., Henne, R.M., Sommerfield, D.: Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 959–967. KDD '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1281192.1281295
8. Mietzner, R., Unger, T., Titze, R., Leymann, F.: Combining Different Multi-Tenancy Patterns in Service-Oriented Applications (2009)
9. R. Fielding, J. Gettys, J.M.: HTTP Version 1.1 (1999), https://www.ietf.org/rfc/rfc2616.txt
10. Roddick, J.F.: A survey of schema versioning issues for database systems. Information and Software Technology 37(7), 383–393 (1995)
11. Scott, J., Nisse, D.: Software configuration management. IEEE Press, Piscataway, NJ, USA (2001), http://www.computer.org/portal/web/swebok/html/ch7
12. Shao, Q.: Towards Effective and Intelligent Multi-tenancy SaaS. Ph.D. thesis, Arizona State University (2011)
13. Thoughtbot: The Playbook: Feature Flags (2013), http://playbook.thoughtbot.com/validating-customers/feature-flags/
14. Weissman, C.D., Bobrowski, S.: The design of the force.com multitenant internet application development platform. Proceedings of the 35th SIGMOD international conference on Management of data - SIGMOD '09 p. 889 (2009), http://portal.acm.org/citation.cfm?doid=1559845.1559942
15. Yaish, H., Goyal, M., Feuerlicht, G.: An Elastic Multi-tenant Database Schema for Software as a Service. 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing pp. 737–743 (Dec 2011), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6118909