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

Maximilian Schneider and Johan Uhle

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.** TODO
How to provide different versions of a SaaS to multiple tenants at the same time?

**Keywords:** TODO

## 1 Introduction

In recent years

What is SaaS and multi-tenancy and why is it important? (better explained in 2?)

What is versioning?

Reasons for versioning Manage incompatibility Feature change (User acceptance) API compatibility (Technical incompatibility)

Rollout of features Ops: Does it scale? Biz/Users: Is it accepted by users? Does the feature 'work' (A/B testing) - Legal

+ provider pov needs flexibility to react to changed requirements reduce technical debt by retiring code/functionality

+ user pov client will complain and will want to stay on the old version

## 2 Terminology

Terminology

SaaS What is multi tenancy? How does it relate to multi user? multi-user is a subset of multi-tenant every tenant is a multi-user system in itself Multi-tenant = Multi-User + SLA + Heavily customizable [DELFT] Code-Revision / Product-Version - explain PV stop talking about CR afterwards - // api versioning is just small fraction of the application Product version Visible to the user Code revision Invisible to the user

Coupling of product version and code revision depends on architecture of application. example for "old" product version: use a specific software not just version 1 2 3 4 5 MS office versions

why only product version? if changes don't affect the user then we don't need to version When we mention 'version', we are talking about product versions.

(Frontend:UI/Backend:Busisness Logix/DB) - // how can we provide a versioning for all these layers with minimal cost. where the tenant can choose. - // how can we make sure that everything is in sync and consistent - granularity of versioning depends on how tight stack is coupled - users interface with versioning in frontend (tenant or even user specific) - backend/db is only versioned forward for whole tenant - stick with product version instead of code revision - Configuration - variables in database in shared table layout - Customization (int.) - code / templating executed in the context of the saas provider - huge changes - // Eyad does not care about config/customization - // predefined columns - // Eyad doesn't care - -¿ we handle just like configuration Customization (internal) Tenant-specific changes that run within the application context Examples: Templates, Custom DB fields, DSLs

- Extension (ext.) - // E column extension - not focused on marketplace - // truly external extensions are 2 much - // Eyad doesn't care, we don't do - -¿ we don't care Extension (external) Tenant-specific changes that run outside the application and interface with it over an API Example: Third-party add-ons in a marketplace addon is in a way just a regular customer as it is truly external we only have to consider customization in our architecture

## 3 Related Work

- SaaS and Multi Tenancy - Economy of Scale - Development vs. Fixed Cost and Maintenance - Delft paper - DB - A.Kemper "5-Schemas" - A.Kemper "Versioned Tables" - Infrastructure - Salesforce as good Example - Delft no multi-instance - BPEL multi-instance used, though not wanted - ... - Routing - Customization // how 2 either merge these approaches / try to map these approaches to something similiar or new on the application level

## 4 Architecture

In this section we first clarify the *application stack* we are assuming the SaaS application to have. Then we explain the different architectural approaches to implement versioning of SaaS, with *multi-instance* as well as *shared-instance* on the different application stack layers.

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. Practially this means that each tenant has a dedicated migration point in which they decide to switch their version. This switch affects all their users at the same time. Users aren't allowed to individually choose their version.

### 4.1 Application Stack

To understand the architecture better, we first want to look at the SaaS application stack, which basically

- Frontend -¿ Backend -¿ DB frontend: ui backend: app / business logic db: shared state - user management - Versioning layer - Auth layer engineering deatils: load balancer / caching emitted - ?web server (caching, load balancer)?

hat viele geile diagramme aus dem vortrag

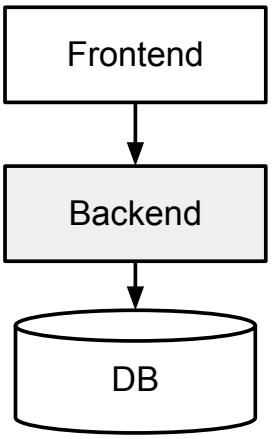


**Fig. 1.** Awesome Image

### 4.2 Multi-Instance

Not multi-tenant in itself, but usually the most inuitive way to provide saas. good if not too many clients.

multi-tenancy using multi-instance though this seems simple from an architectual perspective bad consolidation factor high maintenance cost as operation complexity increases with the number of tenants not the spirit of SaaS

### 4.3 Shared-Instance

**Frontend** Web application tight coupling between backend and frontend simultaneously versioned versioning concern in backend

API consumers (e.g. native apps) loose coupling between backend and frontend backend needs to provide different versions of API api consumers choose version

Frontend API Versioning In URL: http://api.com/v1/resource.json?version=1 Accept Header: Accept: application/json+v1 Application Flag:

**Backend**

*Shared-Instance 1:1* geiles diagram

a product ver = code rev

Pro Versioning is external to application code

Contra heterogeneous deployment worse consolidation factor intelligent routing layer needed forking of code base makes security patches difficult to apply

*Shared-Instance 1:n* geiles diagram

version choice is on app server

Pro good consolidation homogenous deployment more flexibility: fine grained feature selection

Contra increased code complexity abandoning old versions needs cleanup

inside of code base are many version checks "if version == 1.4 ..." more flexibility in choosing version, maybe not choose version but choose only features
high cost for abandoning old version to clean code base

**Listing 1.1.** code snippet

```
if(user.has_version?):
  do_something()
end
```

### Database

*Different versions share same schema* db is version agnostic

*Different versions need separate schemas* How to version schemas?
Database needs to support several schemas for a table at the same time
No DBMs supporting this is known to us
Thus versioning has to happen in the backend

**Listing 1.2.** sql

```
create table users_v1;
create table users_v2;
create table users_v3;
```

Pro: Works well if not too many versions
Con: Messy design if many versions

*Pivot tables* bild von "An Elastic Multi-tenant Database Schema for Software as a Service"
reduce dbms more to a key value store as used with salesforce

*Migration of data* version is chosen per tenant data migrations need time data migrations might need downtime on-the-fly migrations possible

## 5 Future Work

Version-aware databases
Switching versions per user. We assumed for our architecture that a tenant chooses the version for all their users at the same time 4. Future research should investigate how versioning on the user-level could be implemented.

## 6 Conclusion

Versioning is mainly a concern of one layer in architecture

Where to select the version? multi-instance: access layer 1:1 = routing layer 1:n = code level

High Level Versioning: clean separation of versioning concern low consolidation

Low Level Versioning: versioning inside of application high consolidation

Versioning is mainly an engineering problem key aspect: migrating schema & data Version-aware databases should support this