Deep Customization of Multi-Tenant SaaS Using Intrusive Microservices

Hui Song
SINTEF Digital
Oslo, Norway
hui.song@sintef.no

Franck Chauvel
SINTEF Digital
Oslo, Norway
franck.chauvel@sintef.no

Arnor Solberg
SINTEF Digital
Oslo, Norway
arnor.solberg@sintef.no

ABSTRACT

Enterprise software needs to be customizable, and the customization needs from a customer are often beyond what the software vendor can predict in advance. In the on-premises era, customers do *deep customizations* beyond vendor's prediction by directly modifying the vendor's source code and then build and operate it on their own premises. When enterprise software is moving to cloud-based multi-tenant SaaS (Software as a Service), it is no longer possible for customers to directly modify the vendor's source code, because the same instance of code is shared by multiple customers at runtime. Therefore, the question is whether it is still possible to do deep customization on multi-tenant SaaS. In this paper, we give an answer to this question with a novel architecture style to realize deep customization of SaaS using intrusive microservices. We evaluate the approach on an open source online commercial system, and discuss the further research questions to make deep customization applicable in practice.

KEYWORDS

Customization, SaaS, Multi-tenancy, Architecture style

ACM Reference Format:
Hui Song, Franck Chauvel, and Arnor Solberg. 2018. Deep Customization of Multi-Tenant SaaS Using Intrusive Microservices. In *ICSE-NIER'18: 40th International Conference on Software Engineering: New Ideas and Emerging Results Track, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183399.3183407

1 INTRODUCTION

Independent software vendors (ISVs) develop enterprise software products that support daily business activities such as sales, accounting, human resources, project management, etc. Since ISVs' customers are companies that must differentiate themselves from competitors to secure their market share, the *customizability* of ISVs' products is essential.

The software engineering research community has contributed several advanced techniques for customization, but they all somehow rely on ISVs to predict all the possible customization needs. For example, both software product-line (SPL) and customer-defined business process force customers to use only the features or functionalities implemented by the vendor. Dependency injection (DI), plug-ins and scripting languages support customers in developing new functionalities, but within the ISV's predefined APIs. In practice, however, ISVs cannot always predict all possible customization needs, and customers often need a *deep customization* that goes beyond provided functionalities or APIs.

Before SaaS, customers, or the consultants hired by them, implement such *deep customization* by directly modifying the source code of the ISV's product. In this way, they are able to customize anything in the product, and often with a low cost, especially when their customization needs are small but scattered in the product. An empirical study on 8 big companies shows that in most of the cases customization on Enterprise Resource Planning (ERP) systems end up with code modifications [3]. Our study on the two companies that commissioned this research also reveals that many of their on-premises customers have more or less deep customizations [4]. In the on-premises era, such deep customization is possible because each customer deploys and maintains their own copy of the product.

Unfortunately, as ISVs move their product to multi-tenant SaaS, deep customization by directly modifying code is no longer possible, because all the customers *share a single instance of code* and physical resources. Allowing one customer to bring their own code directly endangers the quality of service (QoS) delivered to other tenants. Existing research and practice mainly focus on the application of non-intrusive mechanisms on SaaS customization, and thus it is still an open question whether *it is possible or not to realize deep customization on multi-tenant SaaS*, so that customers can still do any customization beyond the vendor's prediction.

In this paper, we explore a new architecture style to enable deep customization on multi-tenant SaaS, using intrusive microservices. The main body of custom code runs in a separate microservice, isolated from the main service, whilst specific parts of the custom code are sent back to the main product and dynamically compiled and executed within the execution context of the main service. We illustrate the approach using an open source shopping application.

This approach answers the question that deep customization is indeed possible on multi-tenant SaaS, but it raises

ICSE-NIER'18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5662-6/18/05…$15.00
https://doi.org/10.1145/3183399.3183407

the issue of tight coupling. The issue is inherent to deep customization from the on-premises era, but have not attracted enough attention to researchers from the software engineering community, because on-premises customers tend to tolerant and ignore the problems caused by deep customization. However, the consequence of tight coupling are magnified by SaaS, and thus we will discuss the potential research to mitigate the consequences in the SaaS context.

We organized the remainder as follows. Section 2 introduces an example where customization conflicts with multi-tenancy. Section 3 defines the architecture style and Section 4 evaluates it on a prototype customizable SaaS. Section 5 discusses future research directions. Section 6 discusses related work before Section 7 concludes the paper.

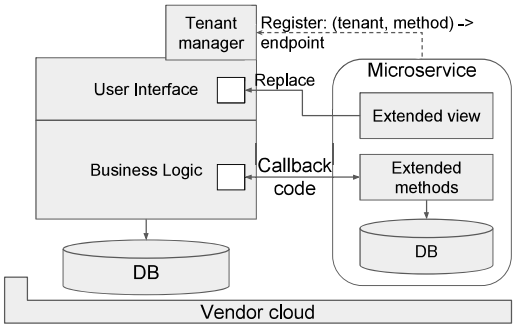

Figure 1: Custom code as intrusive microservices

2 MOTIVATING EXAMPLE

Let us consider *MuTeShop.com* (Multi-Tenant Shops) as a made-up example that captures the situation of our industrial partners. *MuTeShop.com* now offers online shopping SaaS: Customers can quickly set up their e-commerce website, including catalog, shopping cart, etc. From the *MuTeShop.com* standpoint, each customer is a *tenant* with a separate website for their *end-users* to browse and buy goods.

MuTeShop.com has to be customizable. For example, one of their key customer/tenant—say *Music.MuTeShop.com*—requires that their shopping cart includes a charity donation option. Whenever an end-user adds an album into her shopping cart, she can donate money to a designated charity, which eventually adds-up on the total checkout price. This customization need is beyond the prediction of *MuTeShop.com*: As a vendor, they would not predict the requirement for the end-users determine the price.

If *Music.MuTeShop.com* were using the "on-premise" version of a shopping product, they would have a customization consultant to directly modify their own copy of the main source code to implement the donation process. Then they only need to change the database schema to record the amount of donation for each item in each shopping cart, the business logic to account for these donations and finally the user interface for their end-users to choose/see how much they donate, within a few lines of new code.

However, as a multi-tenant SaaS, *MuTeShop.com* cannot allow such direct code modification, because the same database schema and the price accounting source code are shared by multiple tenants. Modifying the code for one tenant would interfere the service to other tenants. *MuTeShop.com* also could not provision a dedicated resource for *Music.MuTeShop.com* to host their modified code, because this would drive down the economies of scale they aim at their SaaS product.

The example illustrates the problem that many SaaS vendors face: Their services are successful for some customers only if they can do deep customization. But the vendor cannot allow the same way of deep customization as for on-premises products, because they have to keep the service multi-tenant. In summary, they need a customization solution that achieves both *isolation* and *assimilation*. Isolation shields each tenant from possible misbehaviors of other customizations, including performance, resource consumptions, but also privacy. Assimilation ensures that customers can customize every aspect of the main service, as if they were vendor developers with full control to the entire source code.

3 CUSTOM CODE AS INTRUSIVE MICROSERVICES

We propose a new architecture style to enable deep customization on multi-tenant SaaS, using intrusive microservices.

The customization by each tenant is running as one or more self-contained microservices, hosted in the same vendor cloud as the main service, as is shown in Figure 1. Each custom microservice re-implements a small number of fine-grained structures in the main service source code, such as Java methods or HTML templates, and manages its own data using a light-weight database. A *tenant manager* registers which microservice supersedes which part of the main product and for which tenant, so that when a tenant request reaches this part, the service will redirect it to the registered microservice, instead of executing the original code.

The custom microservices are intrusive to the main service through callback code snippets, which are exposed to the same execution context as the to-be-replaced standard code. Take a Java method as an example, its context includes the method arguments, the host object, and any accessible static values. When required, the microservice sends callback code to the main service to query data from these contexts or to manipulate the context in order to alter the behavior of the main service. In this way, the customization power of the microservice is not limited by any pre-defined API, but is as expressive as directly modifying the code.

Figure 2 illustrates the interaction between the main service and a custom microservice, using an example described in Section 2. When an end-user of *Music.MuTeShop.com* checks out, the main service processes the request and finally invokes the `GetTotal` method to calculate the total price. The method invocation is intercepted, and redirected to the tenant manager to check if there is any custom microservice registered on the method for this tenant, and in this case

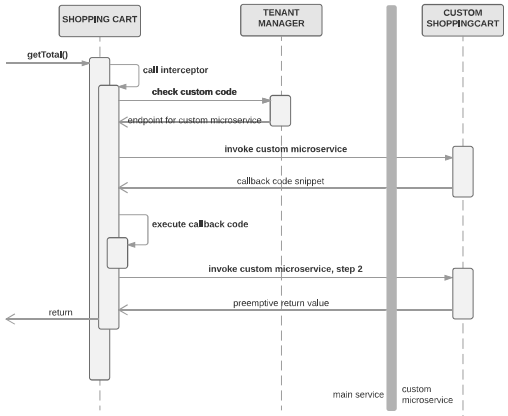

Figure 2: Interactions at runtime

it receives the endpoint to the `Donation ShoppingCart` microservice. The service invokes the obtained endpoint instead of executing the standard method body and transfers the execution to the microservice.

The microservice has two steps. In the first step, it sends back a callback code snippet to query the current items in the shopping cart, together with a new endpoint for the second step. The main service executes the callback code with the context and invokes the second step with the parameter as all the shopping cart items and their original prices. In the second step, the microservice queries its own database for the amount of donations, and sum a new total amount aggregating the input prices and the donations. It sends back this new total amount as a preemptive return value for the original method, and then the main service returns this customized value to the previous caller. All the communications between the main service and the microservice are through REST.

4 IMPLEMENTATION

We evaluate our approach by building a prototype deep-customizable multi-tenant shopping service, based on an open-source software, the Microsoft MusicStore.

MusicStore is the official test application for ASP.Net Core, which is Microsoft's next generation web framework for cross-platform and cloud computing. MusicStore has the essential features for online shopping, but, as is, does not support multi-tenancy nor customization.

To apply our architecture style, we adapted the MusicStore in the following three steps, and released it as an open-source project. We first equipped it with a *tenant manager*. It identifies the tenant based on the logged-in end-user, and provides a REST API to register custom code microservices. After that, we inject the source code level *pointcuts* before and after every method in MusicStore, using automatic code rewriting. Each pointcut invokes a library implementing the behavior in Figure 2. Finally, we implement the compiling and execution of *callback code* using an enhanced version of DyanmicLinq, a dialect of C#. Note that during the process, we did not predict and design where and how MusicStore can be customized, but only equip it with generic mechanisms.

We deployed the adapted application into a local machine which has a Docker daemon to host all the custom code microservices, each of which runs as a separate Docker container. A customer deploys a microservice by providing an archive of the custom source code, the new HTML templates, and the docker configuration files. Theoretically, a tenant can use any languages and techniques to write the custom code, as long as they can publish it as a REST service. Along with the prototype, we provide a simple supporting library for customers to use MongoDB as customer database, NodeJS as HTTP server, and TypeScript as a programming language. We also provide an IDE based on Visual Studio Code, with code generation and deployment support.

Acting as a tenant to the MusicStore service, we implement a complete customization example following the donation scenario described in Section 2. The custom code replaces three methods in the business logic that adds items in the shopping cart, lists items and calculate the total price, respectively. They represent 114 lines of TypeScript, with 4 lines of callback code embedded in it as text values. In addition, the custom code contains two views as HTML templates (*.cshtml files): a new one for selecting donations (14 lines) and a modified one for shopping cart overview (18 lines of new code within 129 lines of the full file). The effect of the customization is shown by a screen shot video [1].

As we can see from the demo video, the customization is *assimilated* to the main service from an end-user's perspective. The custom code is integrated seamlessly: the new views are through the main service URL and have the consistent style as the existing ones. From the tenant's point of view, the customization does not rely on any APIs specific for this donation functionality, and they can replace arbitrary methods in the source code. As for *isolation*, the custom code can be deployed on the fly, without requiring the main service to be rebooted. At runtime, the custom microservices are running within their own containers, with managed and limited resources. In this way, problematic custom code cannot take too much resource to affect the operation of the main service and any other tenants.

5 FURTHER RESEARCH QUESTIONS

Deep customization has its inherent drawbacks, and an important one is tight coupling, because it prevents the ISV and its customers from developing their solutions independently. In practice, such an inter-dependence yields a "clear" service interface upon which both agree and that should stabilize over time. Deep customization, by achieving ultimate flexibility for customers to modify the service, sacrifices with such a clear interface, because any predefined interface would mean a predefined boundary about what customers can do for customization. Such tight coupling courses problem on the security of the main service, as well as the stability of custom code during the update of the main service.

[1] https://streamable.com/oxx65

We will explore means to mitigate the consequences of tight coupling in order to enable a practical co-evolution of the main product and its customizations. We identified the following research questions:

De-coupling by generation. How to abstract all adjustments that characterize a customization, including new or alternative UI elements, business logic and data schema in a high-level domain-specific language, and get the actually custom code, as well as where the code should be injected to the main service, automatically generated?

Static coupling detection How to compute, by static analysis on both the service source code and the custom code, the actual scope in the main product that are impacted by a given customization? On top of this, we can further analyze if there is any security sensitive code falls into this scope (which means that the custom code is risky to the main service), and if an update of the main service has overlap with this scope (which indicates that the custom code is not stable with the updated main service).

Dynamic coupling testing. How to generate the *security test cases* to check whether the custom code breaks the main service, as well as the *stability test cases* to check if an old custom code no longer works with the main service update. The security test cases are executed every time a new customization is on board, while the stability test cases are executed on all the existing customization every time a new update to the main service is in staging. Such testing approach does not solve the coupling problem, but alarms the vendor and the customers before the coupling causes actual damage.

6 RELATED WORK

As for customization of multi-tenant SaaS, the closest work we have found comes from Walraven et al. [7], a middleware for multi-tenant systems that uses dependency injection (DI) to achieve the dynamic switch between customized logics for different tenants. Such customization is rather assimilated but still did not reach the level of deep customization: Customers develop the to-be-injected logic (in terms of Java classes) using the same language as the main product, with the same execution context. However, what can be customized in the main service is still limited by predefined injection points, which has to be predicted by the vendors. Another work from the same team [5], tackles customization using context-oriented programming(CoP), but there the custom code is developed and maintained by the ISV's team, not the third-party customizations as we are aiming.

Tsai et al. address customization of SaaS but using orthogonal variability models (OVM) [6]. They focused on the business logic and propose workflow models with explicit variation points that new tenants may replace with preexisting variants. They assumed that customizations result from the combination of existing pieces and do not entail the development of new and assimilated code, as we aim at. García-Galán et al. [1] supports user-centric adaptation of multi-tenant services by dynamically identifying the service configuration based on the user preferences. This work is also based on a defined configuration space comprise existing features, but deep customization targets the development of new features by customers.

Recently, Makki et al. [2] intercepted calls to subprocesses and dynamically branch to tenant-specific variations. An SPL defines the valid variations using a feature model that the underlying middleware queries at runtime to invoke the proper subprocesses. The SPL defines one single standard interface, beyond which customization is not possible.

7 CONCLUSION

This paper highlights the need and the challenge to realize deep customization in multi-tenant SaaS, which hinders many ISVs of enterprise software from transferring to SaaS, as some of their important customers may already have incompatible deep customizations. We have shown how the intrusive microservice architecture style helps enabling deep customization in multi-tenant SaaS, reconciling assimilation and isolation. Customers may develop separate microservices and register them as alternative end-points that then replace parts of the main product. This architecture style however tighten the coupling between the main product and its customizations, and therefore further research in different directions is required to mitigate the consequence of such tight coupling, in order to make deep customization applicable in practice.

ACKNOWLEDGMENTS

This research was funded by the Research Council of Norway under agreement No. 256594 (Cirrus), and the EU H2020 Programme under agreement No. 731529 (STAMP).

REFERENCES

[1] Jesús García-Galán, Liliana Pasquale, Pablo Trinidad, and Antonio Ruiz-Cortés. 2014. User-centric adaptation of multi-tenant services: Preference-based analysis for service reconfiguration. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 65–74.

[2] Majid Makki, Dimitri Van Landuyt, Stefan Walraven, and Wouter Joosen. 2016. Scalable and Manageable Customization of Workflows in Multi-tenant SaaS Offerings. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16)*. ACM, New York, NY, USA, 432–439.

[3] Marcus A Rothenberger and Mark Srite. 2009. An investigation of customization in ERP system implementations. *IEEE Transactions on Engineering Management* 56, 4 (2009), 663–676.

[4] Hui Song, Franck Chauvel, Arnor Solberg, Bent Foyn, and Tony Yates. 2017. How to support customisation on SaaS: a grounded theory from customisation consultants. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 247–249.

[5] Eddy Truyen, Nicolás Cardozo, Stefan Walraven, Jorge Vallejos, Engineer Bainomugisha, Sebastian Günther, Theo D'Hondt, and Wouter Joosen. 2012. Context-oriented Programming for Customizable SaaS Applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC '12)*. 418–425.

[6] W. T. Tsai and X. Sun. 2013. SaaS Multi-tenant Application Customization. In *Proceedings of the 7th IEEE International Symposium on Service-Oriented System Engineering (SOSE)*. 1–12.

[7] Stefan Walraven, Eddy Truyen, and Wouter Joosen. 2011. A Middleware Layer for Flexible and Cost-efficient Multi-tenant Applications. In *Proceedings of the 12th International Middleware Conference (Middleware '11)*. 360–379.