



# Event-Based Customization of Multi-tenant SaaS Using Microservices

Espen Tønnessen Nordli, Phu H. Nguyen, Franck Chauvel, Hui Song

## ► To cite this version:

Espen Tønnessen Nordli, Phu H. Nguyen, Franck Chauvel, Hui Song. Event-Based Customization of Multi-tenant SaaS Using Microservices. 22th International Conference on Coordination Languages and Models (COORDINATION), Jun 2020, Valletta, Malta. pp.171-180, 10.1007/978-3-030-50029-0\_11 . hal-03273988

**HAL Id: hal-03273988**

**<https://inria.hal.science/hal-03273988>**

Submitted on 29 Jun 2021

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

# Event-based Customization of Multi-tenant SaaS Using Microservices<sup>\*</sup>

Espen Tønnessen Nordli<sup>1</sup>, Phu H. Nguyen<sup>2</sup>[0000–0003–1773–8581], Franck Chauvel<sup>2</sup>, and Hui Song<sup>2</sup>

<sup>1</sup> University of Oslo, Oslo, Norway

`espentno@ifi.uio.no`

<sup>2</sup> SINTEF, Oslo, Norway

`firstname.lastname@sintef.no`

**Abstract.** Popular enterprise software such as ERP, CRM is now being made available on the Cloud in the multi-tenant Software as a Service (SaaS) model. The added values come from the ability of vendors to enable customer-specific business advantage for every different tenant who uses the same main enterprise software product. Software vendors need novel customization solutions for Cloud-based multi-tenant SaaS. In this paper, we present an event-based approach in a non-intrusive customization framework that can enable customization for multi-tenant SaaS and address the problem of too many API calls to the main software product. The experimental results on Microsoft’s eShopOnContainers show that our approach can empower an event bus with the ability to customize the flow of processing events, and integrate with tenant-specific microservices for customization. We have shown how our approach makes sure of tenant-isolation, which is crucial in practice for SaaS vendors. This direction can also reduce the number of API calls to the main software product, even when every tenant has different customization services.

**Keywords:** Microservices · Architecture · Event-based · Cloud · SaaS · Customization · IoT · Edge · Security

## 1 Introduction

Most businesses and public services rely on enterprise software such as enterprise resource planning (ERP) or customer relationship management (CRM), to name a few. Because every company has its unique organization, processes and culture, no off-the-shelf software directly fits. Companies eventually *customize* these software systems to meet their specific requirements. For simple scenarios, software vendors predict where and how their software products may

---

<sup>\*</sup> The research leading to these results has received funding from the European Commission’s H2020 Programme under the grant agreement number 780351 (ENACT), and from the Research Council of Norway under the grant agreement numbers 296651 (ASAM) and 256594 (Cirrus).

be customized, and provide their customers with application programming interfaces (API), extension points or configuration choices. However, there are always customers whose requirements overstep the embedded customization capacity. These customers need the vendors to provide mechanisms for performing *deep customization*, that goes beyond the vendors’ prediction.

Deep customization may affect any parts of a software product, including the user interface (UI), the business logic (BL), the database schemas (DB) or any combination thereof. When a software product used to be deployed on the customers’ premises, each customer naturally ran its own customized version, in full isolation. Nowadays, software vendors are migrating their software products to the Cloud. In the Cloud-based multi-tenant software-as-a-service (SaaS) model, however, every customer must run the same code base (main product), which cannot be directly modified for one customer without affecting other customers. Software vendors desperately need novel deep customization solutions for the Cloud-based multi-tenant SaaS model.

More recently, leveraging the microservices architecture [14, 6, 1] for enabling deep customization of multi-tenant SaaS is a very promising direction as presented in [10, 12, 9, 8, 11]. These microservices-based customization approaches vary in how they balance *isolation* and *assimilation*. Isolation guarantees tenant-specific customization only affects that one single tenant, whereas assimilation guarantees that customization capability can alter anything in the main software product. Intrusive microservices [10, 12, 9] provide tight assimilation at the cost of security (tenant isolation), whereas the non-intrusive approach called **MiSC-Cloud** [7, 8, 11] trades assimilation for higher security. **MiSC-Cloud** orchestrates customization using microservices via API gateways.

In this paper, we present an event-based non-intrusive deep customization approach for multi-tenant SaaS using microservices as part of the **MiSC-Cloud** framework [8]. The event-based approach, in combination with the synchronous way of customization in [8], shows how the **MiSC-Cloud** framework can coordinate the execution of the BL components (microservices) of the main product as well as the customization microservices of tenants to obtain the desired customization effects in the multi-tenant context.

The remainder of this paper is structured as follows: Section 2 defines deep customization. Then, Section 3 presents the event-based customization approach with key techniques. In Section 4, we show a proof-of-concept for the proposed approach by applying it on a reference application for microservice architecture by Microsoft. Section 5 discusses related work. Finally, we provide in Section 6 our conclusions and possible future research directions.

## 2 Deep Customization

By contrast with other customization means such as settings, scripting languages or API, deep customization demands that one can possibly make *any* change to the system, as one can do with direct access to the source code. Changes can, therefore, affect the user interface (UI), the business logic (BL), the database

schema (DB), or any combination thereof. Deep customization turns out difficult in multi-tenant SaaS environments, where all tenants originally run the same code (UI, BL and DB). Tenant-specific customization must affect only one single tenant. This work focuses on the customization of BL, especially based on events. In this way, customization microservices communicate with the main product, either in a synchronous way by requesting data and waiting for the response (RPC-like), or in an asynchronous way, by publishing and subscribing to events (pub/sub). The customization of UI and DB can be found in [10, 12, 9, 8, 11].

### 3 Event-based Customization Approach

In this section, we first present the main components for enabling event-based customization of multi-tenant SaaS in Section 3.1. Then, Section 3.2 details how the event-based customization approach works. In Section 3.3, we discuss how the event-based customization fulfils the requirements of tenant isolation.

#### 3.1 Main Components for Enabling Event-based Customization

Among the five main components of the MiSC-Cloud framework as presented in [7, 8], we focus on presenting the **Tenant Manager** and the Event Bus as the key parts of the event-based customization approach. The **API gateways**, **IAM Service**, and **WebMVC Customizer** are the same as we described in [8].

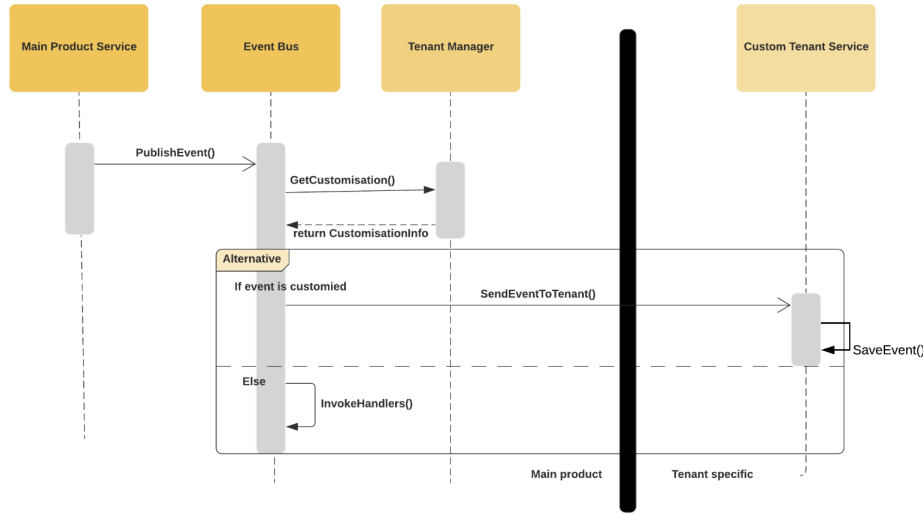
The **Tenant Manager** is a service that manages the registration of customization microservices including the events registered for customization for different tenants. The service has a simple database that stores all the tenants that are using the application, all the different events that exist in the main product and finally all the customization microservices that exist for tenants and specific events. Additionally, it stores an endpoint for each customization that is used for halting the flow of events to be discussed further in the next section.

The **Event Bus** is key to enable event-based customization. Therefore, the prerequisite for enabling event-based customization is that the main product already has (part of) its logic flow orchestrated via events. If the main product already has an **Event Bus**, such an **Event Bus** can be extended to enable event-based customization. If the main product does not have an **Event Bus**, a new one can be introduced as presented in [8]. It is important to note that a software product can be re-engineered to enable event-based logic orchestration at the back-end via an **Event Bus**. Different migration approaches from monolithic to microservices architecture already show some patterns and practices to migrate from synchronous calls into event-based communication between microservices [4, 13]. Moreover, software vendors can also create user or system events within their software product to allow authorized event-based integration with external systems (of their customers). This event-based integration is similar to the traditional way of offering a rich REST API for synchronous integration, e.g., using traditional GET-PUT-POST statements.

### 3.2 Event-based Customization Flow

A customization microservice can subscribe to an event that is published to the Event Bus when something notable happens, such as when another microservice (of the main product or another tenant-specific customization) updates a business entity. When a microservice receives an event, it can update its business entities, which might lead to the publishing of more events. We design the event bus as a multi-tenant interface with the tenant-specific APIs needed to subscribe and unsubscribe to events and to publish events.

The flow of processing events in the original **Event Bus** implementation must be changed for customization purposes. Before publishing events to the consumers, it checks with the **Tenant Manager** for any customization that has been registered for any event and tenant (see Figure 1). If an event is not customized, then the event is processed in the standard fashion. In the case that an event is customized, the event is sent to the endpoint that is part of the response from the **Tenant Manager**. At this point, the tenant's microservice is responsible for storing the event until the required customization has been achieved. Then, the tenant's microservice can republish the event to the Event Bus, along with a flag that instructs the **Event Bus** to not check for customization again, to avoid an infinite loop.



**Fig. 1.** Event-based customization flow.

In some cases, customization microservices would require some execution context from the main product that does not exist in the events that they receive. To obtain such context, customization microservices can make authorized synchronous calls to the APIs of the main product as presented in [8]. In fact, events

often contain enough execution context for customization microservices to execute customization scenarios. This means that only a few special customization scenarios would require such synchronous calls from customization microservices to the API of the main product. Combining the synchronous and asynchronous ways of customization can offer a more complete non-intrusive customization approach for multi-tenant SaaS. However, we recommend the use of event-based customization for as many customization scenarios as possible to reduce the traffic of API calls to the main product, which often leads to performance bottleneck when there are many customized tenants with unpredictable loads.

### 3.3 Tenant-isolation and Tenant-specific Event-Handlers

The **Event Bus** implementation and architecture in the main product must ensure that tenant isolation is still preserved. Instead of having one connection to a single event bus, there must be multiple connections, one per tenant. One example of such an event bus implementation is based on RabbitMQ that can make use of virtual hosts<sup>3</sup>. This way allows us to have a logical separation per tenant, and the permission can easily be set so that each tenant is only allowed to interact with its own virtual host.

## 4 Proof-Of-Concept and Evaluation

In this section, we show a proof of concept of our approach for enabling deep customization of the **eShopOnContainers** by extending the Event Bus in the application. The .NET Microservices Sample Reference Application **eShopOnContainers**<sup>4</sup> has been chosen for a couple of reasons. First, **eShopOnContainers** has a clear separation between the user interface and the business logic of the application as a prerequisite of the **MiSC-Cloud** framework. Secondly, the application follows the microservices architecture, and as such, has loose coupling as compared to a monolithic application. Finally, the collaboration between the microservices that the application as a whole is made up of is done using events and a publish/subscribe system.

An Event Bus implementation must be associated with the authentication and authorization mechanisms of the **IAM** service for multi-tenant SaaS-based on Open ID Connect or OAuth 2.0. As an implementation of RabbitMQ already exists in the **eShopOnContainers**, we have extended it to enable event-based customization.

Let us consider the original **eShopOnContainers** in the GitHub repository as the main product being customized. We show how our event-based customization approach can enable different customization scenarios for two tenants as the representatives of multi-tenant context<sup>5</sup>. The first use case in Section 4.1 adds new logic to the main flow of the ordering process, without altering any of the

<sup>3</sup> <https://www.rabbitmq.com/vhosts.html>

<sup>4</sup> <https://github.com/dotnet-architecture/eShopOnContainers>

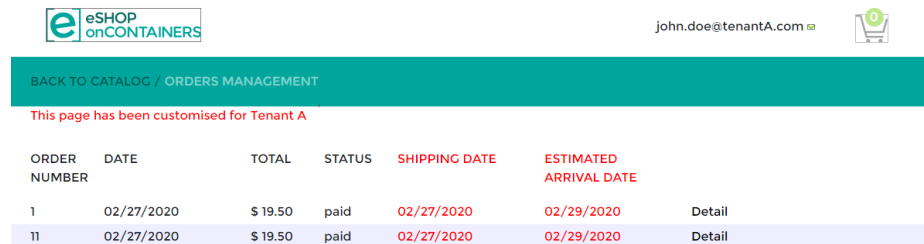
<sup>5</sup> <https://github.com/Espen1004/eShopOnContainersCustomised>

existing functionality. The second use case in Section 4.2 requires a modification of the existing logic of the ordering process by halting the flow of the order.

#### 4.1 Tenant A's Customization of the Ordering Process

The original ordering process is straightforward. After having logged in, a customer can add items in the shopping cart and then create an order with card payment and shipping address. What happens at the back-end is that the **Basket** service of the **eShopOnContainers** publishes a **UserCheckoutAcceptedIntegrationEvent**, which is consumed by the **Ordering** service to create and process the order, e.g., generating **OrderSubmittedIntegrationEvent**. **Tenant A** wants to change the original ordering process of **eShopOnContainers** to incorporate the shipping information from external (third-party) systems. This means that after the **Basket** service has published a **UserCheckoutAcceptedIntegrationEvent**, the **Ordering** service validates the order request before creating an order and an **OrderSubmittedIntegrationEvent** to trigger this customization. Here, we demonstrate the customization of **Tenant A** using the asynchronous way. The synchronous way of customization has been presented in [8].

The asynchronous way of customization has been used for the customization scenario in which the user has checked out (**UserCheckoutAcceptedIntegrationEvent**), and the corresponding order has been made (**OrderSubmittedIntegrationEvent**). The customization microservice **Shipping** of **Tenant A** intercepts the **OrderSubmittedIntegrationEvent** and queries an external system for an estimated time for delivery. This information is then stored in the microservice's database, which can then be retrieved whenever the **My Orders** page is displayed. The customization result can be seen in Figure 2. The parts in red, e.g., **SHIPPING DATE**, are the customized content, which are only available for the users of **Tenant A**. What happens in the background is that we have added a new **Event Handler** that consumes the **OrderSubmittedIntegrationEvent**. Whenever this event is published by the main product to the event bus of **Tenant A**, the **Event Handler** consumes the event and calls the customization microservice **Shipping**, which is responsible for calculating the shipping information by integrating with an external system.



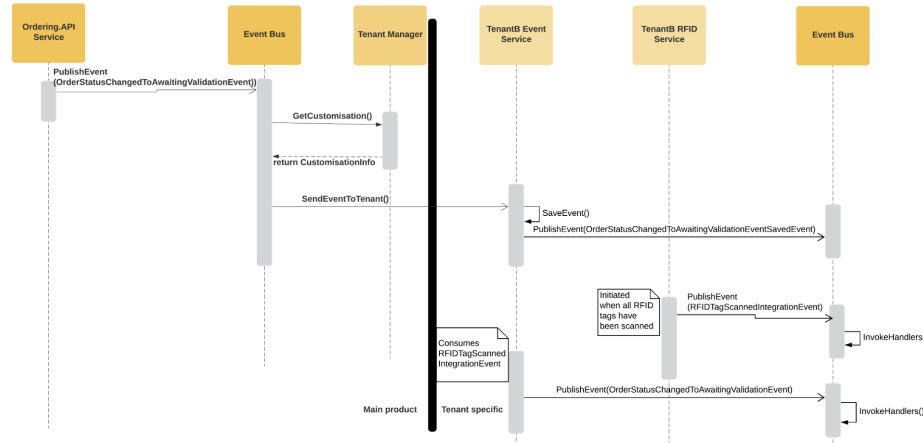
| ORDER NUMBER | DATE       | TOTAL    | STATUS | SHIPPING DATE | ESTIMATED ARRIVAL DATE |        |
|--------------|------------|----------|--------|---------------|------------------------|--------|
| 1            | 02/27/2020 | \$ 19.50 | paid   | 02/27/2020    | 02/29/2020             | Detail |
| 11           | 02/27/2020 | \$ 19.50 | paid   | 02/27/2020    | 02/29/2020             | Detail |

**Fig. 2.** Customization of **Tenant A**: An estimated time for delivery.

#### 4.2 Tenant B's Customization of the Ordering Process

Tenant B wants to customize the ordering process with some additional steps to mark all the items with RFID. Before the order status is set to confirmed, all the order lines in the order should be scanned. Further, the order status should only be set to confirmed when all the items in the order have been scanned.

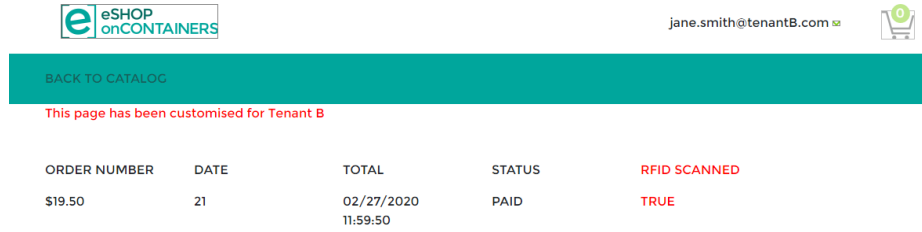
The second use case requires that the status of the order is not set to confirmed until all the items in the order have been scanned. To ensure this, we need to halt the flow of the application by capturing the `OrderStatusChangedToAwaitingValidationIntegrationEvent`. This is done by registering this event for the specific tenant in the **Tenant Manager**, as well as the endpoint that we want the event to be sent to. Figure 3 shows the customization flow triggered by the `OrderStatusChangedToAwaitingValidationIntegrationEvent`. This event is then stored in the database of the microservice for this customization until the `RFIDTagScannedIntegrationEvent` is published by the `TenantARFIDService`.



**Fig. 3.** Customization of Tenant B: The customization flow around the `OrderStatusChangedToAwaitingValidationIntegrationEvent`.

The customization scenario depicted in Figure 3 starts when the **Ordering** service publishes the `OrderStatusChangedToAwaitingValidationIntegrationEvent`. Next, the **Event Bus** implementation checks for any customization for this event by querying the **Tenant Manager**. As Tenant B has customized this event, the **Event Bus** sends the event to the endpoint specified in the response from the **Tenant Manager** rather than publishing to the RabbitMQ instance. At this point, the tenant has control of the event and can save it to the local database of Tenant B's **Event Service** before publishing `OrderStatusChangedToAwaitingValidationEventSavedEvent` to the **Event Bus**. The `OrderStatusChangedToAwaitingValidationEventSavedEventHandler` in Tenant B's **RFID Service** consumes this event, and stores the necessary data in its database.





| ORDER NUMBER | DATE | TOTAL                  | STATUS | RFID SCANNED |
|--------------|------|------------------------|--------|--------------|
| \$19.50      | 21   | 02/27/2020<br>11:59:50 | PAID   | TRUE         |

**Fig. 4.** Customization of Tenant B: After all the RFID tags have been scanned.

The next step of the use case is triggered whenever the endpoint in Tenant B’s **RFID Service** is used to indicate that all the order lines have been scanned. The use of this endpoint also triggers **RFIDTagScannedIntegrationEvent**, which is then consumed by the **RFIDTagScannedIntegrationEventHandler** in Tenant B’s **Event Service**. At this point, the original **OrderStatusChangedToAwaitingValidationIntegrationEvent** is re-published to the **Event Bus**, and the handlers in the main product can perform their operations. Then, the event is re-published to the **Event Bus**, and it is processed normally by the main product. The result of the customization, after the RFID tags are scanned can be seen in Figure 4.

The asynchronous customization approach is based on the events in the application. Because all the events are isolated so that each tenant is only able to interact with their own events. This means that tenant isolation is still preserved.

## 5 Related Work

The notion of customizable SaaS applications with explicit support for variability management has been proposed and explored extensively [3]. There are many technical approaches addressing these complexities, such as design patterns, dependency injection (DI), software product lines (SPL), or API. While these approaches help predefining customization at design time, they do not have sufficient support for the complex and unanticipated behavioural coordination between the custom code and the main product at runtime.

The majority of SaaS customization approaches focus on a high-level modification of the service composition. Mietzner and Leymann [5] present a customization approach based on the automatic transformation from a variability model to BPEL process. Here customization is a re-composition of services provided by vendors. Tsai and Sun [15] follow the same assumption but propose multiple layers of compositions. All the composite services are customizable until reaching atomic services, which are assumed to be provided by the vendors.

Middleware techniques can also support the customization of SaaS. Guo et al. [2] discuss, in a high abstraction level, a middleware-based framework for the development and operation of customization, and highlighted the key challenges. Walraven et al. [16] implemented such a customization, enabling middleware us-

ing Dependency Injection. The dependency injection way for customization allows the custom code developers to introduce arbitrary coordination behaviour with the main product, and thus achieve a strong expression power. However, it also brings tight coupling between the custom code and the main product. Operating the custom code as an external microservice eases performance isolation, misbehaviour of the custom code only fails the underlying container, and the main product only perceives a network error, which does not affect other tenants. Besides, external microservices ease management: scaling independently resource-consuming customization and eventually billing tenants accordingly.

## 6 Conclusions

In this paper, we have presented an event-based customization approach that is part of our non-intrusive customization framework for multi-tenant SaaS. This asynchronous way of customization means that customization microservices can have event-based communication with the main product BL components for customization purposes. Using event-based communication between customization microservices and the main product BL components is important not only for the microservices architecture but also for non-intrusive deep customization capability. Enabling customization both synchronously and asynchronously provides a more flexible way of coordinating the customization logic between the BL components (microservices) of the main product and the customization microservices of tenants to obtain the desired customization effects in the multi-tenant context. Our event-based customization approach makes sure of tenant-isolation, which is crucial in practice for SaaS vendors. This approach can also help reducing the number of API calls that may lead to performance bottleneck when there are many customized tenants with unpredictable loads. We planned to collaborate with two SaaS vendors and their customer companies for an empirical study. Enabling event-based customization is also a way to prepare for offloading custom code to the Edge devices. The event bus could be open to events from microservices on Edge devices and maybe even to “things” in the IoT context.

## References

1. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: *Microservices: Yesterday, Today, and Tomorrow*, pp. 195–216. Springer International Publishing, Cham (2017). [https://doi.org/10.1007/978-3-319-67425-4\\_12](https://doi.org/10.1007/978-3-319-67425-4_12), [https://doi.org/10.1007/978-3-319-67425-4\\_12](https://doi.org/10.1007/978-3-319-67425-4_12)
2. Guo, C.J., Sun, W., Huang, Y., Wang, Z.H., Gao, B.: A framework for native multi-tenancy application development and management. In: *e-commerce Technology and the 4th IEEE International Conference on Enterprise Computing, e-commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*, pp. 551–558. IEEE (2007)
3. Kabbedijk, J., Bezemer, C.P., Jansen, S., Zaidman, A.: Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. *Journal of Systems and Software* **100**, 139–148 (2015)

4. Mazzara, M., Dragoni, N., Bucchiarone, A., Giaretta, A., Larsen, S.T., Dustdar, S.: Microservices: Migration of a mission critical system. *IEEE Transactions on Services Computing* pp. 1–1 (2018). <https://doi.org/10.1109/TSC.2018.2889087>
5. Mietzner, R., Leymann, F.: Generation of BPEL customization processes for SaaS applications from variability descriptors. In: *Services Computing, 2008. SCC'08. IEEE International Conference on*. vol. 2, pp. 359–366. IEEE (2008)
6. Newman, S.: *Building microservices: designing fine-grained systems.* ” O'Reilly Media, Inc.” (2015)
7. Nguyen, P.H., Song, H., Chauvel, F., Levin, E.: Towards customizing multi-tenant cloud applications using non-intrusive microservices. In: *The 2nd International Conference on Microservices, Dortmund* (2019)
8. Nguyen, P.H., Song, H., Chauvel, F., Muller, R., Boyar, S., Levin, E.: Using microservices for non-intrusive customization of multi-tenant saas. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. p. 905–915. ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA (2019). <https://doi.org/10.1145/3338906.3340452>, <https://doi.org/10.1145/3338906.3340452>
9. Song, H., Chauvel, F., Nguyen, P.H.: *Using Microservices to Customize Multi-tenant Software-as-a-Service*, pp. 299–331. Springer International Publishing, Cham (2020). [https://doi.org/10.1007/978-3-030-31646-4\\_12](https://doi.org/10.1007/978-3-030-31646-4_12), [https://doi.org/10.1007/978-3-030-31646-4\\_12](https://doi.org/10.1007/978-3-030-31646-4_12)
10. Song, H., Chauvel, F., Solberg, A.: Deep customization of multi-tenant saas using intrusive microservices. In: *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. pp. 97–100. ICSE-NIER '18, ACM, New York, NY, USA (2018). <https://doi.org/10.1145/3183399.3183407>, <http://doi.acm.org/10.1145/3183399.3183407>
11. Song, H., Nguyen, P.H., Chauvel, F.: Using Microservices to Customize Multi-Tenant SaaS: From Intrusive to Non-Intrusive. In: Cruz-Filipe, L., Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S. (eds.) *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*. OpenAccess Series in Informatics (OASICS), vol. 78, pp. 1:1–1:18. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2020). <https://doi.org/10.4230/OASICS.Microservices.2017-2019.1>, <https://drops.dagstuhl.de/opus/volltexte/2020/11823>
12. Song, H., Nguyen, P.H., Chauvel, F., Glattetre, J., Schjerpen, T.: Customizing multi-tenant saas by microservices: A reference architecture. In: *2019 IEEE 26th International Conference on Web Services* (2019)
13. Taibi, D., Auer, F., Lenarduzzi, V., Felderer, M.: From monolithic systems to microservices: An assessment framework. *arXiv preprint arXiv:1909.08933* (2019)
14. Thönes, J.: Microservices. *IEEE Software* **32**(1), 116–116 (Jan 2015). <https://doi.org/10.1109/MS.2015.11>
15. Tsai, W., Sun, X.: SaaS multi-tenant application customization. In: *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*. pp. 1–12 (March 2013). <https://doi.org/10.1109/SOSE.2013.44>
16. Walraven, S., Truyen, E., Joosen, W.: A middleware layer for flexible and cost-efficient multi-tenant applications. In: *Proceedings of the 12th International Middleware Conference*. pp. 360–379. International Federation for Information Processing (2011)