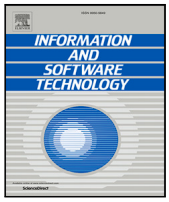
Information and Software Technology 160 (2023) 107230

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Migrating monoliths to cloud-native microservices for customizable SaaS ☆

Espen Tønnessen Nordli [a], Sindre Grønstøl Haugeland [a], Phu H. Nguyen [b,*], Hui Song [b], Franck Chauvel [c]

[a] TietoEvry, Norway
[b] SINTEF, Norway
[c] Axbit AS, Norway

ARTICLE INFO

Keywords:
Microservices
Architecture
Cloud native
Migration
Multi-tenancy
Event-based
Customization

ABSTRACT

Context: It was common that software vendors sell licenses to their clients to use software products, such as Enterprise Resource Planning, which are deployed as a monolithic entity on clients' premises. Moreover, many clients, especially big organizations, often require software products to be customized for their specific needs before deployment on premises.
Objective: However, as software vendors are migrating their monolithic software products to Cloud-native Software-as-a-Service (SaaS), they face two big challenges that this paper aims at addressing: (1) How to migrate their exclusive monoliths to multi-tenant Cloud-native SaaS; and (2) How to enable tenant-specific customizations for multi-tenant Cloud-native SaaS.
Method: This paper suggests an approach for migrating monoliths to microservice-based Cloud-native SaaS, providing customers with a flexible customization opportunity, while taking advantage of the economies of scale that the Cloud and multi-tenancy provide. We develop two proofs-of-concept to demonstrate our approach on migrating a reference application of Microsoft called SportStore to a customizable SaaS as well as customizing another Microsoft's microservices reference application called eShopOnContainers.
Results: We have shown not only the migration to microservices but also how to introduce the necessary infrastructure to support the new services and enable tenant-specific customization.
Conclusions: Our customization-driven migration approach can guide a monolith to become SaaS having (synchronous and asynchronous) customization power for multi-tenant SaaS. Furthermore, our event-based customization approach can reduce the number of API calls to the main product while enabling different tenant-specific customization services for real-world scenarios.

1. Introduction

Following the trend of cloud computing, enterprise software vendors are moving from single-tenant on-premises monolithic applications to multi-tenant (cloud native) SaaS [1]. Migrating to microservices architecture is the right way forward for legacy systems to be modernized [2–4]. There are huge benefits for migrating to microservices architecture such as maintainability and scalability in the long run [5], e.g., by adopting DevOps and benefiting from cloud-native elasticity [6]. However, software vendors are facing two big intertwined challenges: (1) How to migrate their monoliths to cloud-native microservices; and at the same time (2) How to enable tenant-specific (deep) customization in the multi-tenant Cloud native SaaS context. A 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 that goes beyond the vendors' prediction [7,8]. Moreover, in the cloud-based multi-tenant SaaS model, every customer must run the same code base (main product), which cannot be customized for one customer without affecting other customers. Running a different version for each customer would directly negate any economies of scale of the multi-tenant SaaS model [9]. Software vendors desperately need a novel approach to migrate their monoliths to cloud-native microservices while still empowering deep customization for the multi-tenant SaaS model.

☆ The research leading to these results has partially received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement No. 958363 (DAT4.Zero), and from the Research Council of Norway under the grant agreement number 309700 (FLEET).

* Corresponding author.
E-mail addresses: espen.nordli@tietoevry.com (E.T. Nordli), sindre.haugeland@tietoevry.com (S.G. Haugeland), phu.nguyen@sintef.no (P.H. Nguyen), hui.song@sintef.no (H. Song), franck.chauvel@axbit.com (F. Chauvel).

https://doi.org/10.1016/j.infsof.2023.107230
Received 15 February 2022; Received in revised form 3 February 2023; Accepted 12 April 2023
Available online 24 April 2023

In this paper, we first explore the current approaches used in the industry when migrating enterprise applications from a monolithic architecture to the microservices architecture and the different approaches used when transitioning an application from single to multi-tenant. Through reviewing the existing literature we found a number of different approaches that all accomplish one of these two goals, either focusing on the migration from one architecture type to the other or transitioning from single- to multi-tenant. Both microservices architecture and multi-tenancy offer additional benefits to the end-users of the application and the developers. Combining these two principles allows us to better utilize the economies of scale and the resource sharing found in SaaS applications. To this end, **our first contribution** focuses on a migration approach (extended from [10]) for cases where the target application follows a microservice architecture while also allowing tenants to customize the business logic to better fit their needs in a multi-tenant context. Our approach focuses on three stages during the migration, analyzing and breaking down the application into small bounded contexts, transforming the existing infrastructure to fit the new architecture and implementing functionality from the contexts as separate microservices, and finally adding the necessary components to support tenant-specific customization in the multi-tenancy context.

Most existing approaches for customizable SaaS such as dependency injection, software product lines, middleware [9,11–14] can only provide predefined customization capacity at design time. The mainstream multi-tenant SaaS vendors such as Salesforce and Oracle NetSuite provide built-in scripting languages for more fine-grained, code-level customization [15–17]. To maximize customization capability, both vendors have developed very extensive, sophisticated APIs and platforms. This way requires huge up-front investments and extensive training for developers, which are not affordable by smaller software vendors. More recently, leveraging the microservices architecture [18–20] for enabling deep customization of multi-tenant SaaS is a very promising direction as presented in [7,8,21–23]. These microservices-based customization approaches vary in how they balance *isolation* and *assimilation*. In general for any customization approach, isolation guarantees that 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 [8,21,22] provide tight assimilation at the cost of security (tenant isolation), whereas the non-intrusive approach called MiSC-Cloud [7,23,24] trades assimilation for higher security. MiSC-Cloud orchestrates customization using microservices via API Gateways. Via API Gateway(s), the APIs of the main product and the APIs of tenant-specific microservices implementing customization are exposed for tenant-specific authorized access.

In this paper, **our second contribution** focuses on an event-based non-intrusive deep customization approach (extended from [25]), in combination with the synchronous way of customization as described in [7]. By enabling event-based non-intrusive deep customization, 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. 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. This asynchronous way of customization means that customization microservices can have event-based communication with the main product BL components for customization purposes. Therefore, this event-based deep customization approach can help reducing the number of API calls to the main product, which is a big concern for software vendors. The main extensions in this paper compared to our previous work [10,25] are:

- We present in this paper not only the migration approach but also the full customization approach to make a more complete solution, from migrating monoliths to multi-tenant SaaS, and then customizing the target SaaS.
- We apply our full migration-customization approach to two case studies. Firstly, we have migrated the monolith SportStore to multi-tenant SportStore-aaS and then customized the multi-tenant SportStore-aaS. Secondly, we have further customized the microservices-based eShopOnContainers.
- We present a complete customization approach that combines synchronous customization and asynchronous customization to offer flexible ways to implement deep customization. Synchronous customization via API calls is suitable for UI customization, and for querying more "contexts" from the main software product for customization if needed. Asynchronous customization via events orchestration is suitable for BL customization, especially when the microservices of the main software product communicating via events. Event-based customization can help reducing the number of API calls to the main software product. Synchronous customization via API calls can also help triggering event-based customization for the customization scenarios where no events from the main product can be used as triggers.

In the remainder, Section 2 gives the background concepts of this work. In Section 3, we present a motivational example. Our approach is given in Section 4. In Sections 5 and 6, we show the application of the proposed approach to two different reference applications of Microsoft called SportStore [26] and eShopOnContainers.[1] Section 7 discusses related work. Finally, we provide in Section 8 our conclusions and future research directions.

2. Background

In this section, we first recall the main concepts for the migration. Then, we explain our technical objective, i.e., to achieve deep customization of SaaS. After that, we briefly present our previous approach [7] as the baseline of this work.

2.1. Monolithic applications

A monolithic architecture is a simple software architecture, where all parts of the applications (i.e., presentation, business logic and storage) are packaged into one unique deployment unit. This architecture is simple to develop, deploy and operate, but often difficult to scale. A single component implies a single technology that will seldom (if ever) fits all the newest features request. Over the time, monoliths may becomes difficult to scale, without simply over dimensioning the underlying infrastructure. On the human side, having multiple teams working on a single code base requires difficult team-synchronization to avoid half-backed features to land in production.

2.2. Microservice applications

At the other end of the complexity spectrum is the microservices architecture. It divides the application into dozen (or hundreds) of independent microservices that are not only independent deployment units, but also independent execution units that interact through the network. Each has a specific software stack and storage technology, an independent lifecycle, and dedicated development and operation team. These services do not face the customer and operate behind an unified API or proxy such as an API gateway for example.

[1] https://github.com/dotnet-architecture/eShopOnContainers

2.3. Migration

Software migration (also called modernization) refers to any changes meant to run the system on a different execution environment. Legacy mainframe systems, enterprise systems and other long lived systems will generally require one or more "migrations" through their lifespan. Migration remains notoriously difficult, error prone and expensive and architects must carefully weight migration cost against a simpler "rewrite" alternative.

A migrating software often includes three main phases [27]: Recovering existing functionality, transformation of the current architecture, and re-implementation following the target architecture. We outline below the Strangler and Blueprint approaches, two well-documented migration techniques.

2.3.1. Strangler

The Strangler approach [28] builds the new logic literally "around" the old one. At first the new logic does nothing and merely delegates incoming requests to the old system. Gradually, one can replace some delegations with a new implementation until the old system becomes unreachable and can be decommissioned. This approach includes different ways of diverting the calls to the old system, either by event interception, where the edge system taps into the message-stream intended for the original system and redirecting calls as new services are implemented. The alternative is to use asset capture, working with simple orders or specific customers.

2.3.2. Blueprint

The blueprint approach serves as a template for further adjustment, depending on the goals of the migration. The approach consists of two parallel tasks, building the required infrastructure to support the new system, and the actual migration. This approach uses aspects from domain-driven design (DDD) [29] to separate different functionality into bounded contexts. These contexts are then repeatedly migrated into services or a set of services in the new architecture. Identifying these contexts is normally done by analyzing source-code, technical documentation, and in some cases, from interviews with developers that have worked on the pre-existing system [30]. The services migrated should ideally include everything except the UI, implementing the logic of the bounded context and a form of storage for the data related to the service.

In parallel to this process, the required infrastructure for the new architecture should be set up. While the existing infrastructure might be able to support a small number of services, future migration and expansion might require a more specialized infrastructure that better support the system.

2.4. Multi-tenancy

A multi-tenant application serves multiple customers or tenants through an application shared by all the users [31]. Multi-tenancy is prevalent, particularly in cloud-hosted software. Since the application instance is shared among the different users, the software only solves a common set of problems for the users or a problem that the majority of the different users have [32]. As the application is shared among multiple tenants, costs associated with the infrastructure and operations of the servers are also shared between the tenants, resulting in lower overhead for the application compared to running individual instances for each customer.

2.5. 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.

- As for the UI, developers must have means to modify existing screens, that is reorder UI elements (labels, text fields, etc.), add new UI elements (or remove existing ones), or modify the related validation code. They should also be able to add new screens or to remove existing ones.
- As for the BL, developers must have the means to override existing logic (i.e., code) but also to remove or add new logic. In addition, they must be able to emit events and to create new type of events (or delete existing ones). Finally, they must be able to integrate with external services.
- As for the DB, developers must be able to add new columns to tables (or delete existing ones), or create new tables including foreign keys (or delete existing ones). In addition, they should also be able to override the whole data source, with a new dedicated one.

Deep customization turns out difficult in multi-tenant SaaS environments, where all tenants originally run the same code (UI, BL and DB) [21]. Tenant-specific customization must affect only one single tenant. From an extra functional standpoint, the challenge is to offer tightly assimilated customization while preserving tenant isolation, and, in turn, privacy and security.

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 [7,8,21–23].

2.6. TheMiSC-Cloud framework

Fig. 1 shows an overview of the MiSC-Cloud framework [7] using microservices and API gateways to approximate deep customization without sacrificing on security. In this framework, the WebApp MVC and the Business Logic Components are the main SaaS product. Five main modules have been introduced to enable deep customization of the main product in a non-intrusive way: WebMVC Customizer, API Gateways, Tenant Manager, Identity & Access Management (IAM) Service, and Event Bus. The non-intrusive customization approach presented in [7] shows that there are two possibilities for enabling tenant-specific customization for the main SaaS product. The first way is called "synchronous customization" using synchronous API calls via the API gateways. We have presented in details the "synchronous customization" approach in [7]. The synchronous customization approach orchestrates customization using API Gateways to which the APIs of the main product and the APIs of tenant-specific microservices implementing customization are exposed. As presented in [7], the key point in the non-intrusive customization approach is that it enables the authorized access of the tenants' customization microservices to the main product BL via the API gateways. In this way, the tenants' customization microservices can have access to the necessary execution context of the main product BL if needed. Deep customization is also possible because using API gateways even allows a customization service to replace a BL component of the main product if needed.

The second way is called "asynchronous customization", which orchestrates tenant-specific customization using tenant-specific events via the (customization-empowered) Event Bus. Using event-based communication between customization microservices and the main product

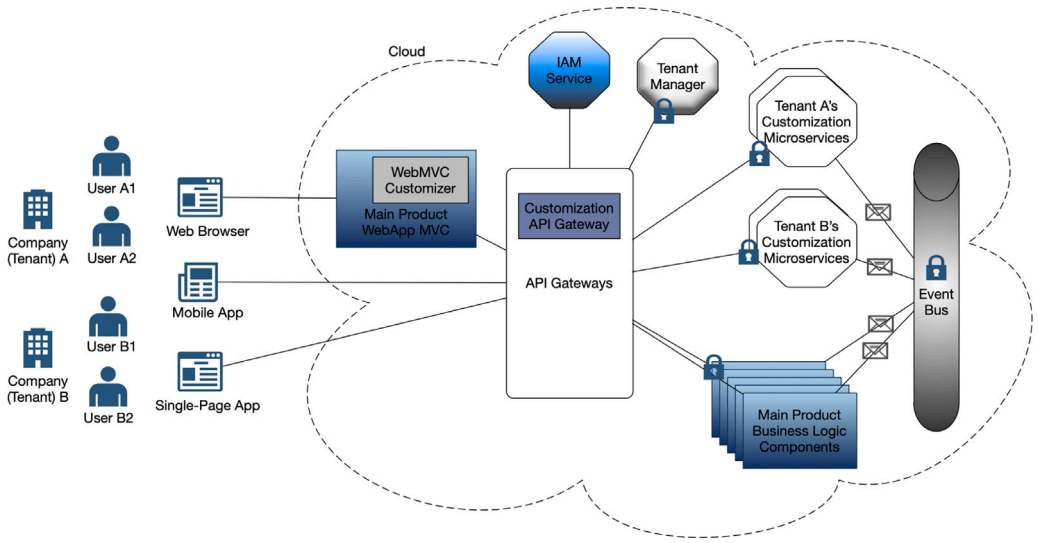


Fig. 1. The MiSC-Cloud framework [7].

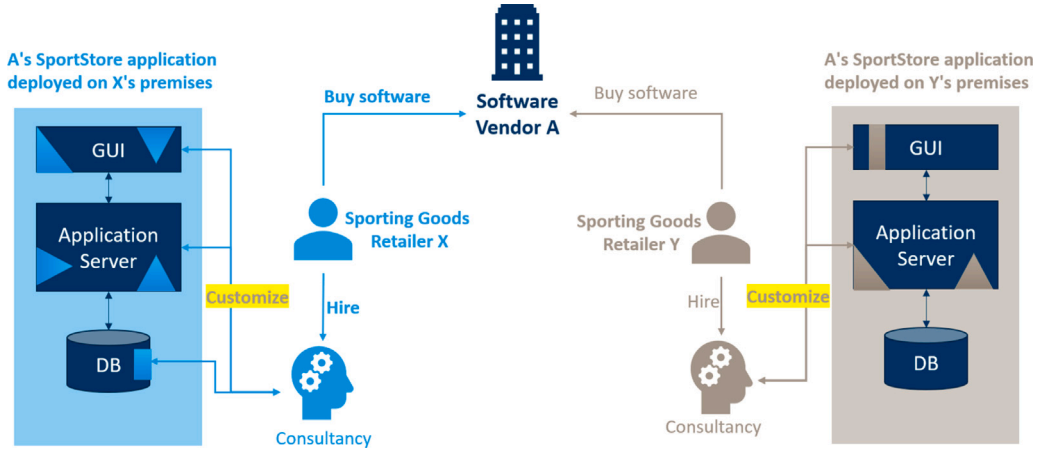


Fig. 2. Companies often need to customize software deployed for them.

BL components is one of the key parts of our customization solution that has not been detailed in [7]. In Section 4, we detail our event-based customization approach.

3. A motivational example

In this section, we present a motivational example based on a SportStore application, to show why we need to migrate a monolith to customization-ready microservice architecture, and what are the requirements for this migration.

SportStore, a web-based store for sports equipment, is a software product of software vendor A. It provides many of the essential features of an online shopping system such as user management, catalogue, shopping cart and checkout. It was implemented in .NET Core with Views, Models, and Controllers for ordering, product catalogs, and a session-based shopping cart. Software vendor A has sold the SportStore product to many sporting good retailers, who deployed the product as separate instances. Among them, retailer X and retailer Y are the big ones (Fig. 2). Such big retailers often do not use the SportStore as-is but hire either software vendor A or a third-party consultant to customize or redevelop the SportStore product further according to their own specific needs. Retailers could have different business models leading to different requirements for customization.

Following the trend of cloud computing [1], software vendor A is migrating their software products such as SportStore to become multi-tenant (Cloud-based) Software as a Service (SaaS). Customer companies such as sporting goods retailers no longer buy a license from software vendor A and install it in their own premises. Instead, they subscribe to an online service, which is also used by other customers, known as *tenants* of the service. From each tenant's perspective, they still have the SportStore product as their own even though this SportStore-aaS is also used by other tenants simultaneously. The SaaS model brings new challenges to software vendor A with regard to enabling customization, which is often required by big retailers like retailer X and retailer Y. It is not possible for any tenant to directly edit the source code, since it is shared by other tenants. A major challenge is to ensure tenant-isolation while enabling tenant-specific customization, which means that no customization specific to a tenant shall ever affect any other tenants. What software vendor A must do to keep their business in the cloud computing era is finding a method to refactor and migrate their monolithic SportStore product to become **multi-tenant customizable (Cloud-based) SaaS**.

4. Our approach for migrating to customizable saas

In this section, we give a brief overview of our migration approach (Section 4.1) and how it relates to multi-tenancy and the ability to provide deep customization for tenants (Section 4.2). We present the

main components for enabling (event-based) customization of multi-tenant SaaS in Section 4.3. Then, Section 4.4 gives the details of how the event-based customization approach works. In Section 4.5, we discuss how the event-based customization approach fulfill the requirements of tenant isolation.

4.1. Overview of our migration approach

Our approach focuses on migrating applications that follow the MVC design pattern. It draws inspiration from the migration approach proposed by E. Wolff [33] in a survey about migration approaches, and the generic re-engineering tool in [27]. Note that in this paper we do not address the constraints of migrating currently in-use applications, which the Strangler approach [28] can do best. We rather focus on how to logically migrate a monolithic application to become customizable in a multi-tenant context. Our approach can be adopted to be part of the Strangler approach for migrating currently in-use applications.

The three different phases of our approach are the *analysis*, *transformation*, and *implementation*. Each of these different phases focuses on a separate aspect of the migration. We use Fig. 3 to demonstrate this process. During the first phase, we analyze the application we are migrating in order to determine how the internals of the application works, and how we can split up the different modules (e.g., Cart, ProductionCatalog and Ordering) in the application into separate microservices. The goal is to identify and group these modules into domains or contexts that focus on specific areas of the application.

The second phase consists of transforming the existing infrastructure of the application to fit the new architecture target. If the existing infrastructure cannot be transformed, or if more effort needed to transform the infrastructure, we implement additional infrastructure to support the new application architecture. Fig. 3 shows that we added additional infrastructure to the relevant modules, includes isolated storage for the different services, a gateway that connects external clients, like a web-application or external third party applications, to the microservices, and a back-end communication system (e.g., an event bus for enabling event-based customization [25]). The migration of existing components into microservices could be partially automated, e.g., using the approach by Christoforou et al. [34]. The final phase consists of implementing the services we have identified during the initial phase, and connecting them to the infrastructure we added during the second phase.

4.2. Migrating to multi-tenancy and deep customization

Our approach aims at enabling the target architecture to be customizable for multi-tenant context as presented in [7,23]. The approaches in [7,8,22,23] offer tenants a way to (deeply) customize the functionality of the multi-tenant application without interfering with behavior for other tenants. The customization-driven aspect makes our approach different from other migration approaches. Adding customization support for tenants can be done using the tenant-manager as a lookup table (Fig. 3). Tenants register their tenant-specific customizations with the tenant manager. These customizations can be standalone services outside the main application, exposing endpoints that the service can redirect calls to. The services and the customized functions that they provide need to adhere to a predetermined stable interface defined by the developers of the main application. This interface serves as a contract between the service and the customized endpoint, describing the expected result that the service needs to continue operations after the customized function has been called.

First, we focus on introducing multi-tenancy to the application. To support multi-tenancy, we need a system for Identity Access Management (IAM) to work in tandem with the tenant manager to support customization of the application for the tenants with application-level tenant-isolation (via tenant-specific authorization mechanisms), and to configure the storage to isolate tenant data. The tenant manager provides all the registered customizations and endpoints for the logged-in tenant user, which is retrieved using a bearer token issued by the IAM system. Tenant isolation at application level is crucial to avoid data leaks problem between tenants as raised in [35]. With the tenant manager and the IAM system in place, we start adding support for customization. We use the tenant manager to return external endpoints to customized functionality in cooperation with the IAM system to ascertain the "tenantID" of the user. The main service then reroutes the request to the external endpoint along with the information required for customization.

To support the customization of the services, they need to use both the identity manager and tenant manager. The tenant manager keeps a record of all the customizations associated with a specific tenant, which can be looked up by services after querying the identity server for the user profile of the token attached to the request. We have two different scenarios for how the tenant manager is used by the services. One scenario is where the tenant has registered customization for some of the functionality, and another where the tenant uses the default functionality implemented in the service already. For both of these scenarios, the configurations are retrieved from the tenant-manager by the service. The response is cached for quick access and reduced network traffic. The tenant manager then push updates to the services when configurations are updated. In the end, the target architecture conforms to the MiSC-Cloud framework as presented in Section 2.6. In the next section, we present how we make the MiSC-Cloud framework more complete by enabling event-based customization.

4.3. Enabling customization ofMulti-Tenant SaaS

Among the five main components of the MiSC-Cloud framework in Fig. 1, 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 [7].

4.3.1. 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 tenant-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 Tenant Manager provides an administration API that can be used to add, remove or alter customization registrations without having to redeploy the Tenant Manager or accessing the database directly. We keep the tenant registration data in a small scale, so that even with a lot of tenants, all data could be cached in memory in the Tenant Manager to increase performance. In addition, the microservices could also store the same customization information in a hot cache or cold cache depending on performance requirements.

4.3.2. Event bus

There are two scenarios regarding the Event Bus, depending on whether or not the main product already has an Event Bus. 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 [7]. 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 showed some patterns and practices to migrate from synchronous calls into event-based communication between microservices [3,36]. Moreover, software vendors can also create user or system events within their software product to allow authorized

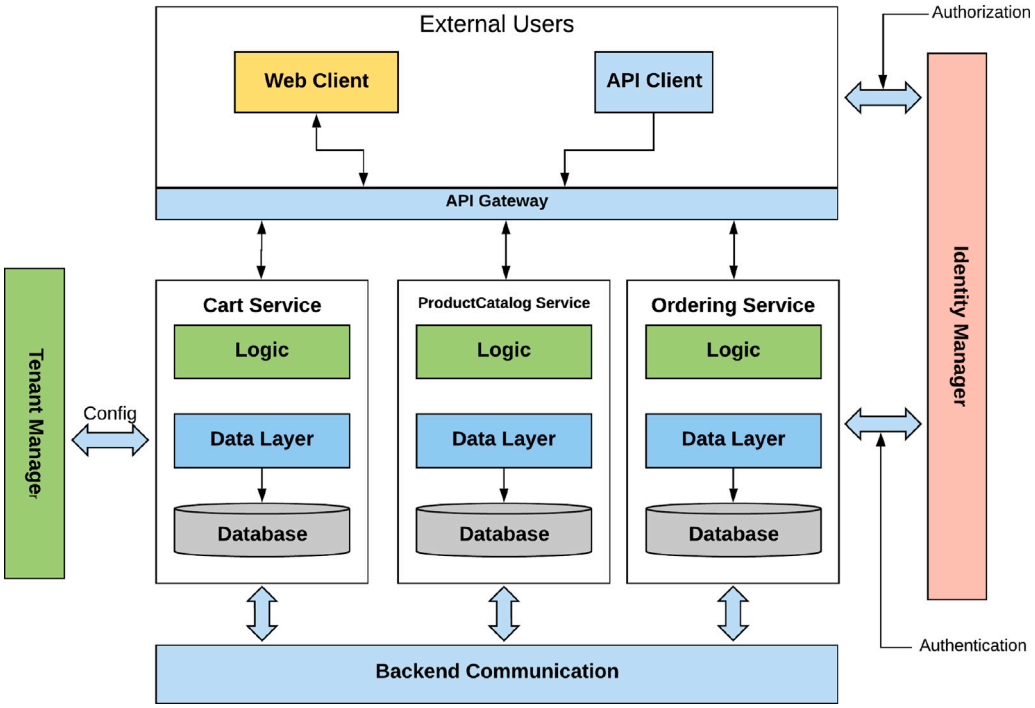


Fig. 3. Target architecture.

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. Therefore, the prerequisite for enabling event-based customization is that the main product already has (part of) its logic flow orchestrated via events. We assume this prerequisite can be fulfilled for enabling event-based customization, which is detailed as follows.

4.4. 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 another microservice (of the main product or another tenant-specific customization) updates a business entity. When a microservice receives an event, it updates its own business entities, which might lead to more events being published. 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.

Therefore, the flow of publishing events in the original Event Bus implementation must be changed for customization purposes. Instead of merely publishing to all the consumers of the event, it instead checks with the Tenant Manager, whether the event maps to a handling logic that is customized for the specific tenant (see Fig. 4). If the event is not customized, then the event is processed in the standard fashion by calling the EventHandler for the different consumers of the event. In the case that the 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 EventBus to not check for customization again, so as to avoid an infinite loop.

Note that in some cases when customization microservices would require some execution context from the main product that does not exist in the events that they receive, they still can make authorized synchronous calls to the APIs of the main product to obtain such context as presented in [7]. 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 APIs of the main product. On the other hand, there is a trade-off decision of tenant-specific customization about whether to give its end users an experience in the UI customization which is as synchronous as possible. We will provide such an example in Section 6.1. Not only for synchronous experience in the UI customization, synchronous calls between the main product and customization microservices are still important for enabling deep customization as we discussed in [7]. By combining the synchronous and asynchronous ways of customization, the MiSC-Cloud framework 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.

4.5. Tenant-isolation and tenant-specific event-handlers

Tenant-isolation is a key challenge for multi-tenant customization. For event-based microservice applications, the isolation of events is an important part of tenant isolation. There are three different sources that can create a new event. The first type is whenever there is an HTTP request from one of the clients (WebMVC, WebSPA, Mobile app, as shown in Fig. 1). The second type is when the source of one event being created is the event handler for another event. Finally, there are events that are created in background tasks, which are primarily scheduled tasks that run at a set interval.

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.[2] This way allows us to have a logical separation per tenant, and

[2] https://www.rabbitmq.com/vhosts.html

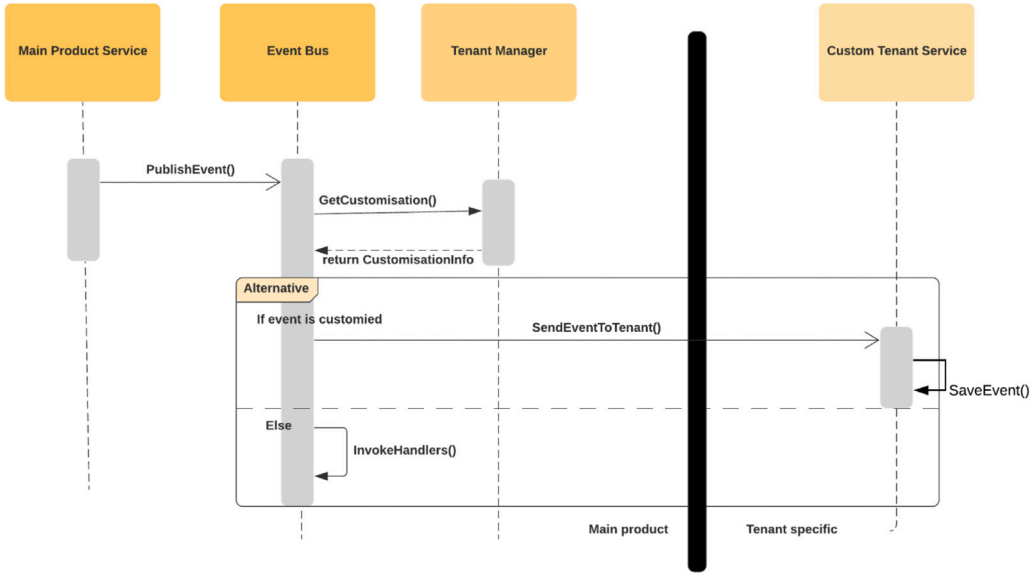


Fig. 4. Event-based customization flow.

the permission can easily be set so that each tenant is only allowed to interact with their virtual host. The tenant's own microservices, however, only need one single connection to their own event bus.

Because there are now multiple connections to different event buses in the main product, all the events must have the tenant information set, so that the main product knows which event bus the event should be published to. The current approach is a relatively simple one, in which the tenant information is simply passed along the flow in the program until a new event is created and published to the event bus. For the events where the source is an HTTP Request, the tenant information is extracted from HttpContext, wherein the tenant information has been set by the IAM service API. For the second case, where the source of the event is another event handler, then the tenant information is simply passed along the low of the program. Finally, for the events that are created by a background task, the user name is retrieved from the database that the background task makes use of, after which the IAM service API is queried so that the tenant information can be set correctly before the event is published.

One possible improvement is to add the tenant information as a scoped object that is set per request, event handler and background task. This could easily be done for HTTP Request by using the HttpContext, and for event handlers, the same tenant information could be set in the header of the AMQPContext. For the background task/batch jobs, it is possible to start one per tenant. These jobs could be run in parallel if necessary, wherein then having access to the necessary tenant information to be able to publish to the correct event bus without having to query the IAM service. To reduce the complexity of the main product, this could be taken further by making use of aspect-oriented-programming that could set the tenant information whenever the constructor of an event is called.

Tenant-isolation is especially important at the application layer using the IAM service. In a system-to-system communication based on Http, for instance a call from any of the in-house services to a third party customization service, it is natural to use token based authentication like OAuth2. This means that the client and the customization service need to access the same token authority (the IAM service). The client calls the IAM service to request a token by identifying itself as a client. The IAM service verifies and sends back a token. Client uses this token in its calls to the customization service. The customization service calls the IAM service to verify the validity of the token. The token contains information about who the client is and scopes that define the usage access. For a customization, the tenant manager can have information about the customized service api, and also information about the idserver/token authority.

5. Use case 1: Migrating and customizing SportStore-aaS

We applied our migration approach to the SportStore application [26], whose monolithic architecture is simplified in Fig. 5. We first present in Section 5.1 how the SportStore application is migrated to multi-tenancy microservices-based SaaS. Then, we show how the SportStore application empowered with event-based customization capability can support real-world customization scenarios.

5.1. Migration

The SportStore application following the MVC pattern [26] is a web-based store for sports equipment. We identify the different groupings in the application during the first phase of the migration. The SportStore application consists of three different groups, i.e., products, carts and orders. Each of these groupings has their own models, views and controllers. The functionality in these groups are closely related to each other. After the initial analysis we introduce the additional infrastructure needed to support the new architecture. In this case, we add an API gateway that connects the user interface to the services through a single entry point, and a message-broker that the services use to communicate. We then implement the groupings we identified during the first phase as separate micro-services with isolated storage. Once we have extracted the services from the old application we connect them to multi-tenant specific infrastructure.

SportStore is implemented in .NET Core with Views, Models, and Controllers for ordering, product catalogs, and a session-based shopping cart. We use these "groupings" as our bounded contexts during the analysis and extract functionality during the decomposition part of the migration. After this, we start implementing services to cover the functionality of the existing application and set up the infrastructure to support it (Fig. 6). The infrastructure includes typical components like the API-gateway and a form of back-end communication for the services. Fig. 3 shows the target architecture of the SportStore application that we have used for our migration experiments. We present the details of our migration process in the following subsections.

The MVC pattern offers a natural separation between the different layers. During the analysis and implementation we extract or replicate from the controllers in their own microservices. The application still

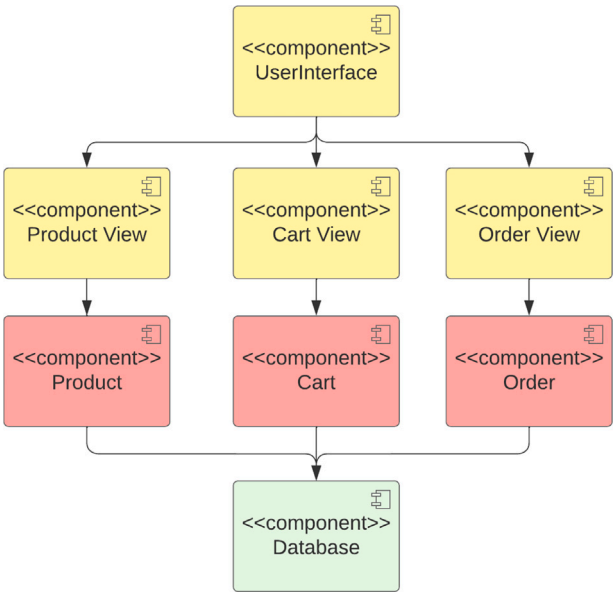


Fig. 5. The monolithic application.

has controllers; however, these only serve as a way to make calls to the services. In a way, they hide the fact that the back-end of the system is spread out into microservices. The goal is to first analyze and break down the modules in the SportStore application into separate bounded contexts following domain-driven design [29] and implementing these contexts as microservices.

5.1.1. Analysis and decomposition:

Once the structure and architecture of the existing application are analyzed and mapped out, we start decomposing the application into separate domains based on domain driven design method [29]. Moreover, it is also possible (not addressed in our paper) to use coupling measures to drive the decomposition process [37] . During the analysis, we found three different domains within the application—the product domain, the cart domain, and the order domain. The product domain contains a template for the products. The product template consists of a productID, product name, a short description of the product, the price, and the category of the product. The application stores the model in an MSSQL database, which is stored in an Entity Framework repository, with supporting methods for retrieving, updating, adding, and deleting products in the repository. The shared resource between the product domain and the cart is the product (Fig. 6). Each of these domains needs to have a shared understanding of what a product is. The cart domain contains the cart for the current session, and it consists of a list of cartlines, as well as methods for adding and removing items to the cart. Additionally, there are methods for computing the total cost of all the items in the cart and clearing the cart. The cartline class represents an item that has been added to the cart. It is made up of a productId, quantity, and the id for that specific cartline. The cartline resource is shared with the third domain, the orders-domain (Fig. 3).

5.1.2. Additional infrastructure

The migration to microservices requires some additional supporting infrastructure. The introduction of an API gateway and back-end communication is an integral part of the second phase of the migration.

API Gateway serves as a connecting layer for clients and other consumers of the services, rerouting requests to the right microservice, serving as a proxy for the different services.

Message Broker/Event Bus: In the prototype, the back-end communication moved through different stages. Each of these stages aimed to further decouple the microservices from each other. The first iteration of the back-end communication implemented direct synchronous calls from service to service. The calls are made to endpoints that the services expose to each other. The second iteration involves adding a message broker to the application to decouple the services further and adding an asynchronous way for them to orchestrate events affecting multiple services. The message broker is implemented using RabbitMQ. Messages are tagged with a specific topic, for instance, orderCreate, when the cart of a session is checked out in the SportStore. The new message is then routed to all the queues matching the orderCreate tag, and the productOrder service then consumes the messages and creates the order in a FIFO order. The topic message broker was also chosen to facilitate customization for the different services further down the line, where messages of a specific tenant would be published to a queue being consumed by that tenant's customized microservice.

Identity Server: We describe how the different features provided by Identity Server and OpenId connect help with authentication and authorization for the different tenants. Clients represent the applications that can request tokens from the identity server. In our case, only the web store of the SportStore application uses the tokens on behalf of the user. The grant types we define, specify how the clients can interact with the Identity Server. The tokens issued allow both services and users to interact directly with the identity server, because of the grant types we use. If we were to define individual grants for the service and users, we would use the Client Credentials type for the service, and OpenID grant type for the users to interact with the server. The identity resources define the functionality enabled by the identity server. The OpenID identity resource allows users to log in via the OpenIDConnect login screen, while the profile resource type allows the services to retrieve the claims of the users to check for customizations later on. The API resource allows the client to access the gateway by defining and associating access to a specific scope. In our case, we only need one resource since all the services are hidden behind the API gateway, and access to all of them on the user's behalf is necessary to provide the full functionality of the system.

Tenant Manager is an essential component of the multi-tenant aspect of the application. We use it to configure the persistency layer for the tenants and as a lookup for customized endpoints. Using the "/userInfo" endpoint of the identity server, the tenant manager can retrieve all the claims belonging to the logged-in user. The claims contain information about where the user belongs and what right he or she has for the services. With the "tenantID", we can look up the customization for the called functionality on the main service. The tenant manager uses the token from the initial request and sends a request to the "/userinfo" endpoint of the identity server, which returns all the claims associated with the token.

5.1.3. Implementation

In the following, we go through the process following our migration approach and applying it to the migration of the SportsStore application, first to the microservice architecture, and then implementing multi-tenancy for the application. We split the migration up into different phases. Each phase includes the extraction of a single service from the pre-existing system, as well as adding the necessary infrastructure to support the new migrated functionality from the monolith. After all the different services and infrastructure components have been implemented or migrated, the final architecture of the application is the same as the targeted architecture in Fig. 3.

We split the migration itself into different phases, related functionality from the existing application during each of these phases and adding the necessary infrastructure to the new application needed to support the extracted components from the pre-existing application.

The initial phase of the migration consists of the analysis and reverse engineering of the pre-existing application. During phase one, the application is still in a single monolithic piece. At this stage, the application consists of three different layers typically found in MVC

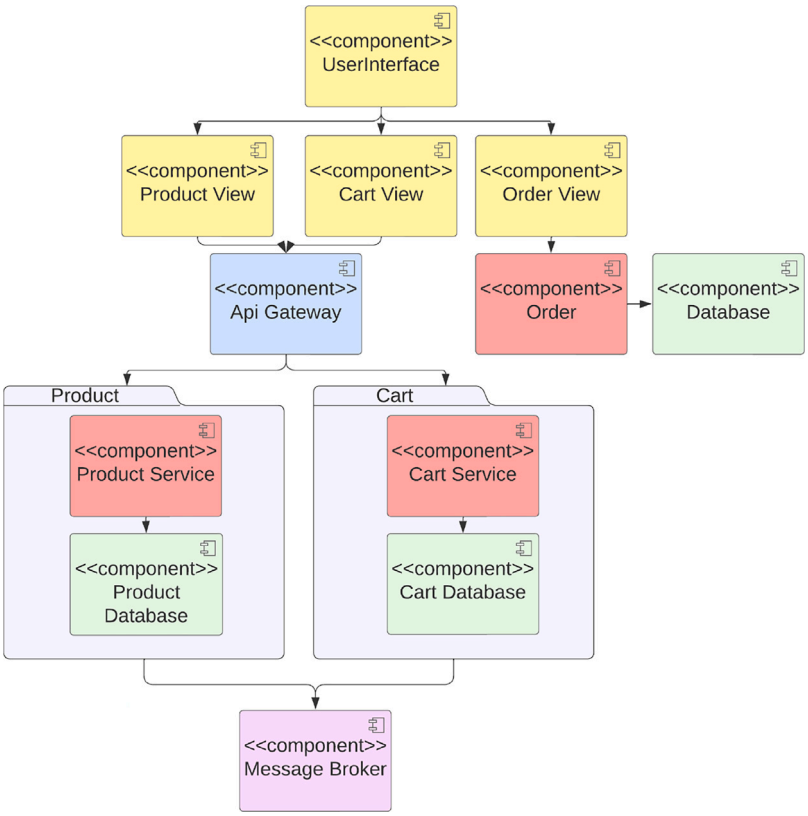


Fig. 6. Migrating functions to microservices with API gateway and Message Broker: The Product component and the Card component have been migrated into the corresponding microservices, while the Order component is not migrated yet.

applications. A user-interface that represents the view, controllers that contain the application logic, and a persistent storage layer that handles the storage of the models in different databases.

The second phase of the migration starts by picking a service or some functionality for migration. Ideally, this functionality should already be loosely coupled to the rest of the code in the monolithic application to limit any dependency back to the monolith. For this phase, we chose to focus on the product module of the SportsStore application. The product module contains all the logic associated with displaying products from the database, adding and updating products in the database, and adding or removing products to a customer cart. We add the additional infrastructure pieces associated with a microservice architecture before we migrate the service. The API-Gateway forwards and redirects calls originating from the web applications to the specific services and act as a unifying endpoint for the View instead of having the calls directed to the specific service it targets an endpoint at the gateway.

In Phase 3 of the migration, we extract another service from the monolith. With two services extracted, we need a way to orchestrate how they cooperate. We introduce the message broker as an additional piece of infrastructure to help with orchestration. The message broker keeps a queue of messages published by all services. Services can then subscribe to the message queue to consume the messages and perform actions with the content of the message. The new service is added to the API-gateway behind the downstream endpoint "/cart". Collaboration and orchestration between the cart service and product service use the message broker to add and remove products to the customer cart. The primary use of the broker at this time is to get information about the products in the cart. Using the session data, we publish a message to the broker requesting a lookup in the products database for the items in the cart. Once the product service retrieves the message from the queue, it aggregates all the products from the cart into a list before returning it to the cart service

After the fourth phase of the migration, the application is now following the microservices architecture. The functionality from the monolith has moved into individual services decoupled from each other. All calls from the web client are passed through the gateway and forwarded to the appropriate service. Orchestration and communication between the services happen through the use of a message broker. The development and deployment of different services are isolated from each of the other services. There is also a clear separation of the different layers of the application allowing tenants to customize different services.

The final phase of the migration introduces more infrastructure to support multi-tenancy. We add an IdentityManager to support login, authentication, and authorization of users and a TenantManager to provide the services with endpoints for tenant-specific customizations. To support the customization of the services, they need to use both the identity manager and tenant manager. The tenant manager keeps a record of all the customizations associated with a specific tenant, which can be looked up by services after querying the identity server for the user profile of the token attached to the request.

5.2. Customization

SportStore is a simple online shopping application, that allows the customer to purchase various sporting goods. The main flow of the program consists of the user adding items to the shopping cart, before checking out and placing the order. The checkout process then uses the items contained in the basket to create an order in the backend system. Following the approach presented in Section 4, we continue the migration of the SportStore by focusing on implementing event-based communication among the microservices via a Message Broker/Event Bus (RabbitMQ). To enable event-based customization in the SportStore application at a multi-tenant level we have to add some additional changes. To enable event-based customization we are reliant of the

product itself having most of it is logic and non-idempotent operations as events. As such we move all the business logic of handling the creation of an order to events. These events are then published to RabbitMQ fanout topics, which the various microservices of the main product can subscribe to. Additionally, we also introduce a new service, product-shipping-service, that is responsible for handling changes in the shipping state of an order. We consider this new service as a natural evolution of the SportStore application after the migration. We do this to more accurately represent a real-world system, which would be a lot more complex than the one in SportStore. Fig. 7 shows the final architecture of the main product without any customizations. The architecture is quite similar to the one in Fig. 3, with the mentioned addition of the product-shipping-service, as well as more of the logic being handled in an event-based manner.

To show a proof of concept of different customizations that can be done for the SportStore using this approach, we used two different customization requirements given by our industrial partners (two software vendors in Norway):

1. Tenant X wants to customize the ordering process to include a third-party service that calculates the arrival date for the order to the customer. Such a third-party service is often an external service provided by shipping companies, e.g., DHL. Only the users of tenant X can have this feature.
2. Tenant Y wants to customize the ordering process with an additional step. No order should have the "shipped" status until all the items in the order have had their RFID tags scanned. This step is important for luxury goods that must be verified before shipping. Only the users of tenant Y can have this feature.

The flow in the main product that leads to the creation of events starts whenever a customer performs the checkout action. This checks out the items in the basket and creates an order in the backend system. In this implementation, we have the following events: CartCheckoutEvent, OrderCreatedEvent and OrderShippedEvent and they are all part of the order process. CartCheckoutEvent is initiated whenever a call is made to the checkout endpoint in the shopping-cart-service and the event is published to a RabbitMQ exchange to which the product-ordering-service subscribes. Next, we have the OrderCreatedEvent which occurs when an order is successfully saved to the database of the product-ordering-service, the creation of the order is triggered by the previously mentioned CartCheckoutEvent. Finally, we have the OrderShippedEvent, the product-shipping-service subscribes to the OrderCreatedEvent and creates an OrderShippedEvent whenever an order is shipped. This final event is also consumed by the product-ordering-service, which then uses that event to update the order's shipped status in its database.

Fig. 8 shows the flow that happens whenever a checkout from the GUI is initiated. First a REST call is made to the shopping-cart-service, this first uses the Authorization token in the HTTP header to make a REST call to the Identity Manager to retrieve the user information and tenant. Then a CartCheckoutEvent is published to the sportstore.cart.checkout fanout exchange. This event is consumed by the product-ordering-service which creates an order in its database using the cart information contained in the CartCheckoutEvent. Then it publishes an OrderCreatedEvent. This event is subscribed to by the product-shipping-service which then ships the order, and publishes an OrderShippedEvent. This final event is subscribed to by the product-ordering-service which then stores the shipped status in its database.

There are additional changes that have to be made to ensure tenant isolation. Specifically, as we allow tenants to create their own customization microservices and consume events from the Message Broker, it is important that they are only able to access events that belong to their organization. To facilitate this we use the virtual host functionality of RabbitMQ to ensure a logical separation between the tenants. Then in the codebase, we simply have to configure the main product to connect to all the virtual hosts specified in the configuration. Further, we set up a Spring Filter that is executed before each HTTP Request to the services. This fetches the user information along with which tenant the current user belongs to from the Identity Manager and stores this in a scoped variable that is available to the code during the context of the HTTP Request. This scoped variable is the used to determine which virtual host that any events created during the HTTP Request context should be published to.

As we now have a separate event bus connections per tenant, tenant customizations can subscribe to the different events in the application and implement additional logic and functionality. However, in some cases this does not add enough customization options that a tenant might require, therefore we also have to implement a way for tenants to capture events. This allows them implement additional logic, checks and functionality that can be executed before a certain condition is met and the tenant can republish the original event to the Message Broker to allow the flow of the main product to complete. The Tenant Manager allows tenants to which events they would like to customize. Then whenever the main product publishes any event to the message broker it queries the Tenant Manager to check whether the event is customized for the current tenant. This is done via a REST call to the Tenant Manager and should be cached by the client to reduce network traffic. In the case that the event is customized the event is sent to the tenant rather than being published as normal. This send event to tenant has been implemented both with REST and event-based communication as a CustomisationEvent (swapped via configuration). In both cases the entirety of the original event is sent to the tenant at which point the tenant are responsible for republishing the event to the message broker. The tenant can then process the event and perform their customizations and at some point republish the event to the message broker to handle the rest of the flow in the main product.

5.2.1. *Tenant X's customization with shipping information:*

For tenant X, we have to add a new microservice for customization purpose, tenant-x-shipping-information, which subscribes to the OrderCreatedEvent published by the main product. Then this tenant X's customization microservice can calculate the shipping information by calling a third party service (just a mock-up, not shown here). Fig. 9 shows the flow of the program with the customization microservice in a sequence diagram. After the order has been created, the frontend then fetches the shipping information from the customization service and shows it in the GUI along with the rest of the order information as shown in Fig. 10

Tenant Y's customization with RFID tags scanned: Tenant Y's customization requires the use of the event capturing functionality that has been added to the main product. Therefore, tenant Y registers that they want to customize the OrderShippedEvent in the Tenant Manager. They also create two new microservices, tenant-y-event-service and tenant-y-rfid-service. Tenant-y-event-service is responsible for receiving the event from the main product, either via REST or as a CustomizationEvent that is published to the Message Broker. Then tenant-y-event-service publishes an OrderShippedSavedEvent to RabbitMQ after saving the necessary information in its database. Tenant-y-rfid-service consumes this event and saves some further information in its own database. The next step of the process is triggered whenever all the RFID tags of the order have been scanned (triggered by a mock endpoint in tenant-y-rfid-service in this proof-of-concept). Tenant-y-rfid-service then publishes an RFIDTagScannedEvent to the Message Broker which is consumed by tenant-y-event-service and causes the original event, OrderShippedEvent to be republished and the rest of the main product flow will execute. Fig. 11 shows the flow of the program with the customization microservice in a sequence diagram

The results in the GUI can be seen in Fig. 12.

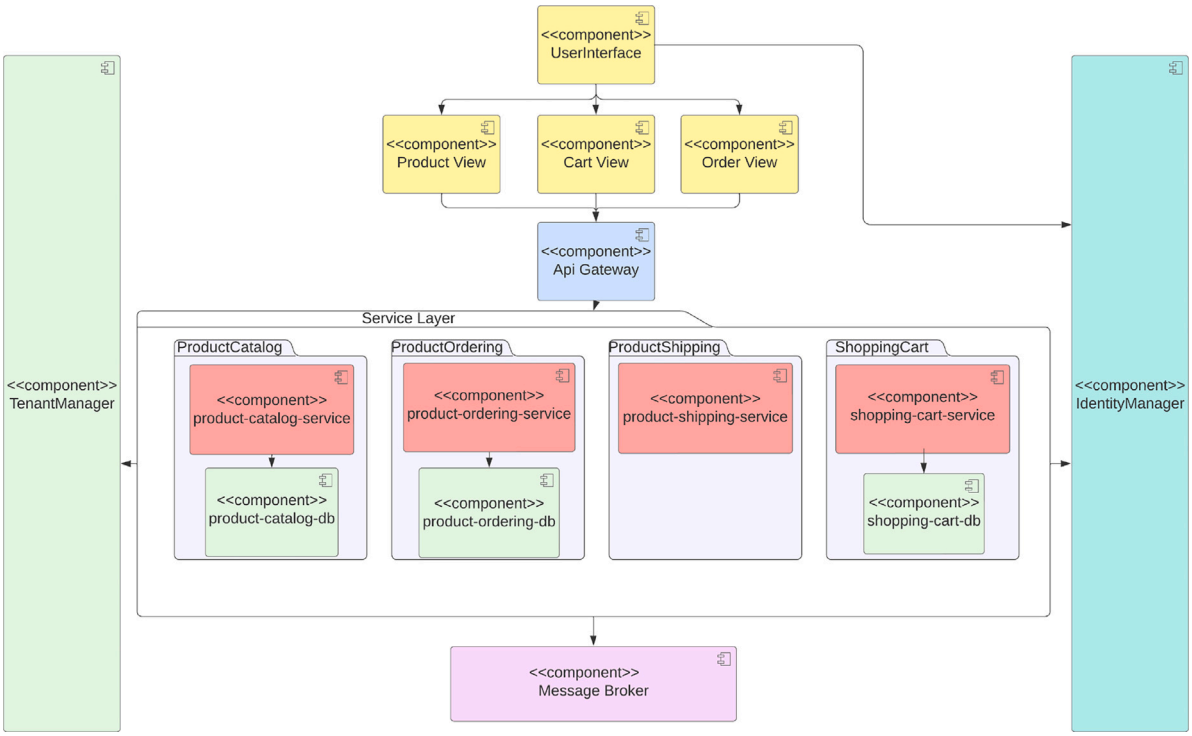


Fig. 7. Event-based SportStore architecture.

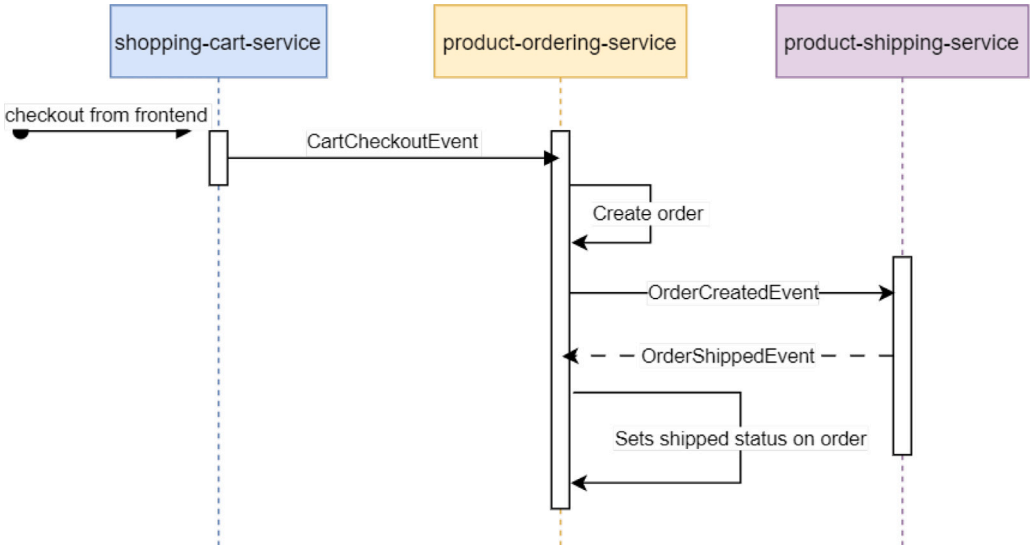


Fig. 8. Main product sequence diagram.

5.2.2. Performance:

We implemented two different ways of sending events to tenant and event capturing functionality. The first way uses REST to send the original event to an endpoint specified in the Tenant Manager. The second implementation uses RabbitMQ to send the event to an exchange that a customizing microservice can then subscribe to. We measured the performance of these two different implementations, the results of which can be seen in Table 1. We measured ten instances of each version. All numbers are in milliseconds. Row one shows the attempt number, row two shows the performance for the Event version and row three shows the performance for the REST calls. The final column shows the average of 44 ms for the Event-based way and 48 ms for REST.

The reason for the first call being quite bit slower than all of the others for both cases is the lazy initialization used by Spring Boot, and could easily be resolved by adding a warm-up request as a part of CI/CD. As we can see from the table, the performance of both REST and Event based communication is similar in this scenario. Meaning that there is no drawback in using an Event based way of communication in this scenario as it allows looser coupling, along with other benefits of asynchronous communication. Additionally, as it is an asynchronous request it is non-blocking and will not slow down the main product in any way, as compared a synchronous way of communicating which would slow down the main product, especially if there is a large amount of customizations.

As the Tenant Manager is a service that could experience a lot of traffic with this approach, we also load tested the service with

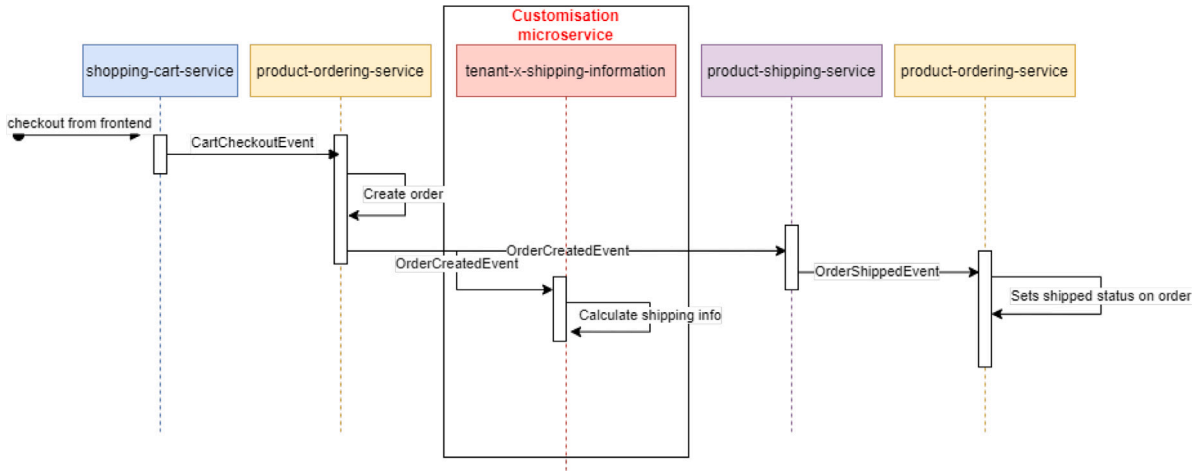


Fig. 9. Shipping information sequence diagram.

Orders

Log Out

Name	Zip	Details				
Espen Nordli	0680	**Product**	**Quantity**	Shipped	Shipping date: 11.02.2022 21:36:27	Arrival date: 13.02.2022 21:36:27
		Kayak	1			
Espen Tønnessen	0681	**Product**	**Quantity**	Shipped	Shipping date: 11.02.2022 21:36:51	Arrival date: 13.02.2022 21:36:51
		Kayak	1			

Fig. 10. Tenant X customization result.

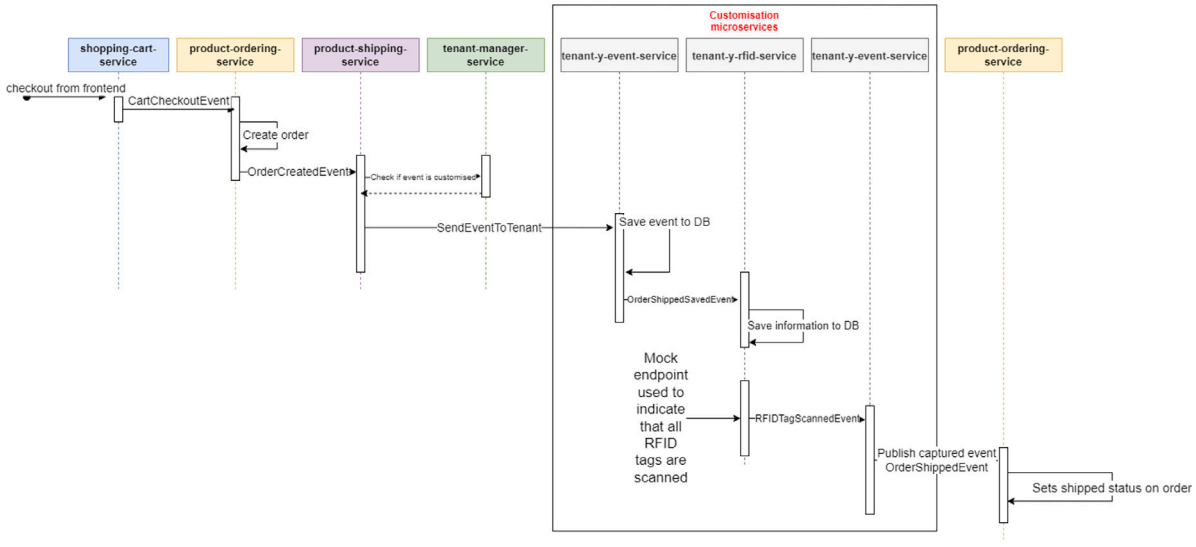


Fig. 11. RFID sequence diagram.

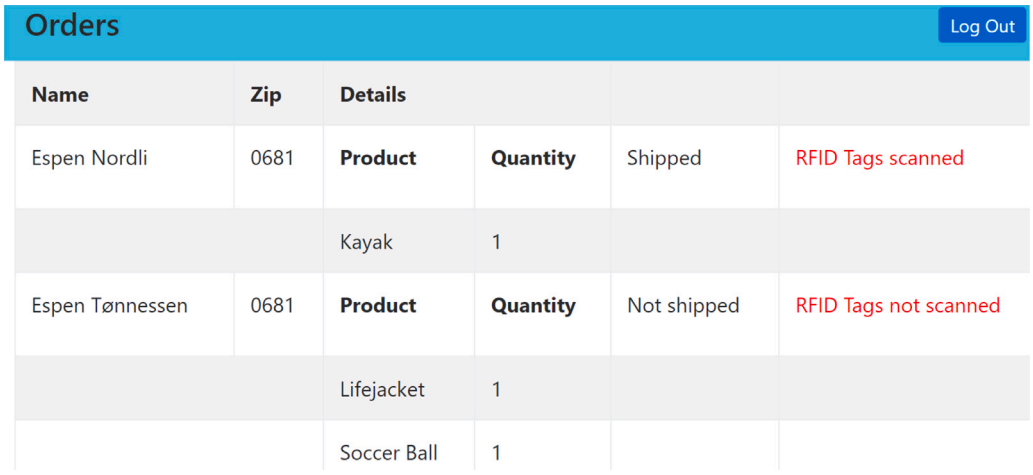


Fig. 12. Tenant Y customization result.

Table 1
Comparison of Event versus REST performance.

#	1	2	3	4	5	6	7	8	9	10	Avg
Event	117	42	34	40	38	29	28	38	35	36	44
REST	166	42	39	37	43	37	37	27	30	26	48

Autocannon as seen in Fig. 13. As the results show a single service can handle 29k requests in 11 s. However, a real production environment could also use load balancing and auto-scaling to handle this. Additionally, client-side caching would greatly reduce the number of calls to the Tenant Manager. Another approach would be to change the way the Tenant Manager works. Instead of services querying the Tenant Manager, each microservice could store all customizations in memory, and then the Tenant Manager could publish an event to, for example, a Kafka Cluster. Then the microservices could simply read the entirety of the Kafka log containing all the customization information on startup, this also means that the Tenant Manager only has to be actively running whenever a new customization is pushed to the Kafka log.

6. Use case 2: Event-based customization of eShopOnContainers

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[3] 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.

There are different potential implementations of an event bus, each using a different technology or infrastructure such as RabbitMQ, Azure Service Bus, or any other third-party open-source or commercial service bus. 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.

[3] https://github.com/dotnet-architecture/eShopOnContainers

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. The first customization scenario in Section 6.1 adds new logic to the main flow of the ordering process, without altering any of the existing functionality. The second customization scenario in Section 6.2 requires modification of the existing logic of the ordering process by halting the flow of the order.

6.1. 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, e.g., PortNord,[4] Bring,[5] DHL.[6] 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. It is important to note that Tenant A has been approved by the software vendor of the eShopOnContainers to register its customization service(s) for subscribing to the events generated by its users. Here, we demonstrate the customization of Tenant A using both synchronous and asynchronous ways. We have used the synchronous way of customization as presented in [7] for implementing the customization of calculating the cost of delivering an order (Fig. 14). Then, we have used the asynchronous way of customization using events for implementing the customization of providing an estimated time for delivery (Fig. 15).

The synchronous way of customization has been used for the customization scenario in which the users of Tenant A can have a synchronous experience of the UI customization. More specifically, after a user of Tenant A has checked out the shopping cart, the UI for placing the order must be customized to show not only the items' prices and quantities (as in the original UI) but also the items' sizes, weights, and the corresponding shipping costs (Fig. 14). In the implementation,

[4] https://www.postnord.no/
[5] https://www.bring.no/
[6] https://www.dhl.no/no/express.html

```
EU+nordlesp@WR910ANJL MINGW64 ~
$  autocannon -c 1 -w 1 http://127.0.0.1:8084/customisations?tenant=TenantX"&"event=testing
Running 10s test @ http://127.0.0.1:8084/customisations?tenant=TenantX&event=testing
1 connections
1 workers
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	3 ms	54 ms	109 ms	110 ms	53.87 ms	31.51 ms	115 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	2385	2385	2609	2617	2578.46	66.6	2385
Bytes/Sec	636 kB	636 kB	696 kB	698 kB	688 kB	17.7 kB	636 kB

```
Req/Bytes counts sampled once per second.

29k requests in 11.01s, 7.56 MB read


EU+nordlesp@WR910ANJL MINGW64 ~
$  autocannon -c 2 -w 1 http://127.0.0.1:8084/customisations?tenant=TenantX"&"event=testing
Running 10s test @ http://127.0.0.1:8084/customisations?tenant=TenantX&event=testing
2 connections
1 workers
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	3 ms	53 ms	105 ms	108 ms	53.65 ms	31.16 ms	114 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	4667	4667	5183	5255	5134.73	157.53	4666
Bytes/Sec	1.25 MB	1.25 MB	1.38 MB	1.4 MB	1.37 MB	41.8 kB	1.24 MB

```
Req/Bytes counts sampled once per second.

57k requests in 11.01s, 15.1 MB read
92 errors (0 timeouts)
```

Fig. 13. Tenant manager load test.

when the method *Create* of the *OrderController* is called, the WebMVC Customizer module (see Fig. 1) checks with the Tenant Manager for any (synchronous) customization registered for the users of Tenant A for this method. Tenant A has already got its customization microservices registered including one customization registered for this method *Create* of the *OrderController*. The synchronous call is made via the API gateway to the corresponding endpoint of the customization microservice (namely Shipping) to trigger the customization logic for getting the items' sizes, weights, and the corresponding shipping costs from an external (third-party) system. The customization results as well as the corresponding customized UI as shown in Fig. 14 are returned to the WebMVC Customizer to render for the end users of Tenant A. Note that this scenario does not use any event. After the user has placed the order, the UserCheckoutAcceptedIntegrationEvent is created and published to the Event Bus by the Basket service, which eventually triggers the following customization.

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 will intercept the OrderSubmittedIntegrationEvent, call an external system for providing an estimated time for delivery (Fig. 15). As there is no need to alter or remove any existing logic in the application, we can simply add a new Event Handler that consumes the OrderSubmittedIntegrationEvent as seen in Fig. 16. 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 Shipping service, which is responsible for calculating the shipping information by integrating with an external system. 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 on the My Orders page in Fig. 15.

6.2. 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, and

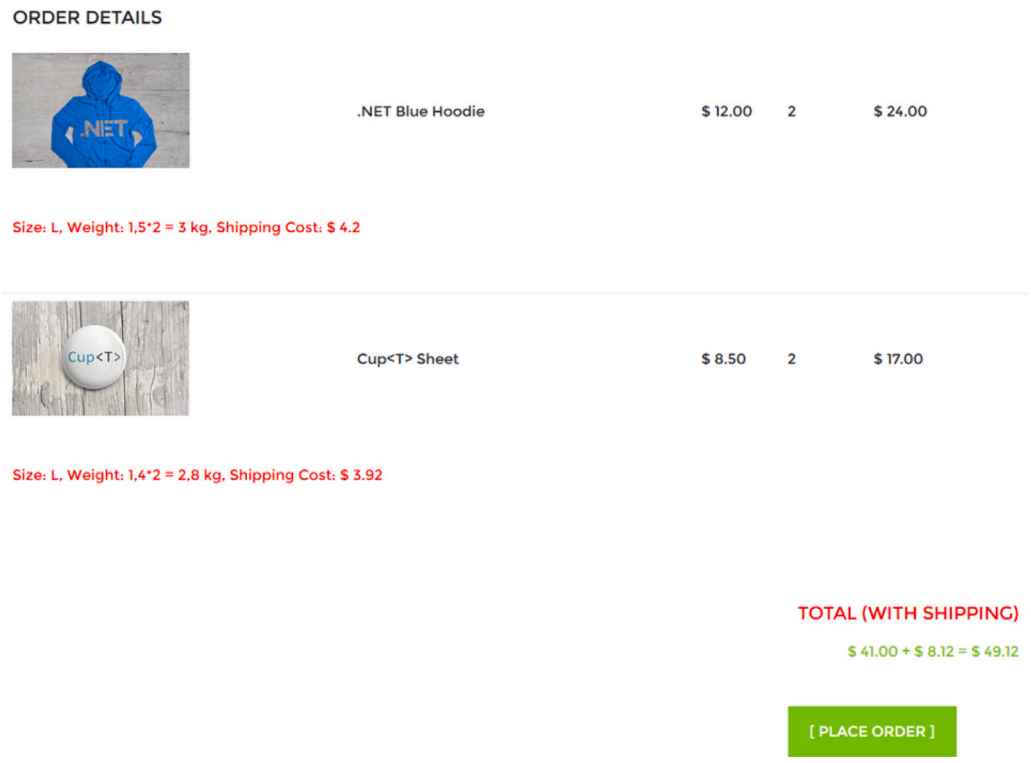


Fig. 14. Customization of Tenant A: Calculating the cost of delivering an order.

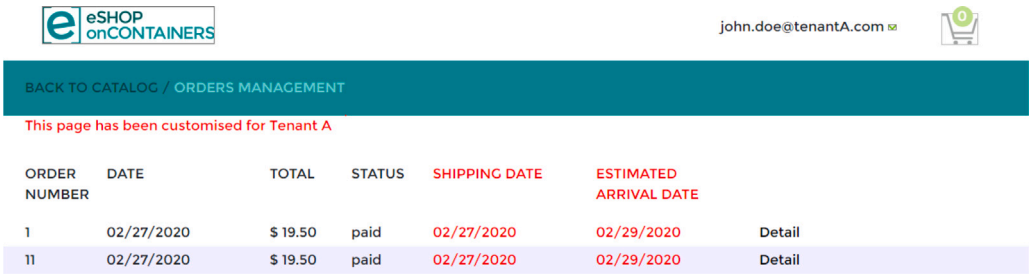


Fig. 15. Customization of Tenant A: An estimated time for delivery.

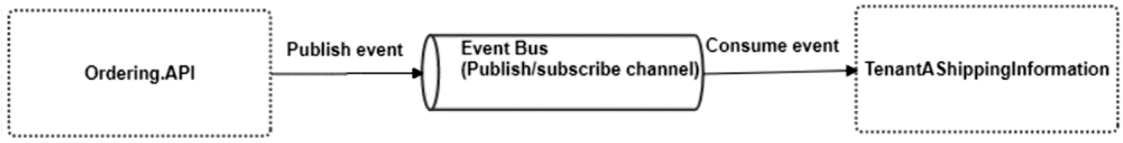


Fig. 16. Customization of Tenant A: OrderSubmittedIntegrationEvent.

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 orderliness in the order has 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. Fig. 17 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, as seen in Fig. 18. This also satisfies the requirement that tenants should be able to publish events that their own customization microservices can consume.

The customization scenario depicted in Fig. 17 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 TenantB Event Service before publishing OrderStatusChangedToAwaitingValidationEventSavedEvent to the Event Bus. The OrderStatusChangedToAwaitingValidationEventSavedEventHandler in TenantB RFID Service consumes this event, and stores the necessary data in its database.

The next step of the use case is triggered whenever the endpoint in TenantB RFID Service is used to indicate that all the order lines have

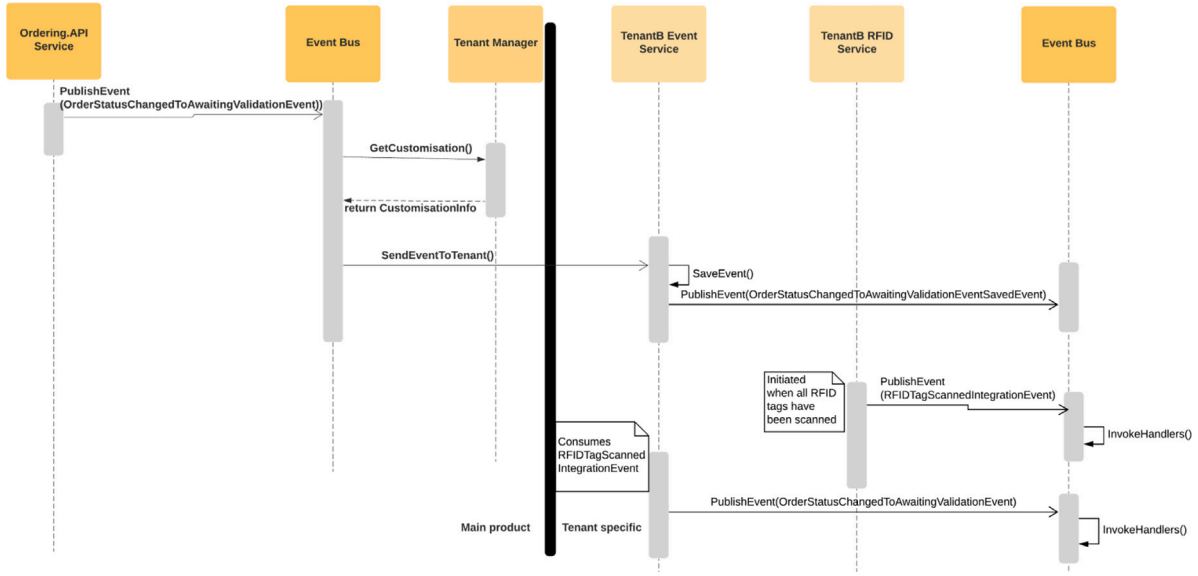


Fig. 17. Customization of Tenant B: The customization flow around the OrderStatusChangedToAwaitingValidationIntegrationEvent.

Fig. 18. Customization of Tenant B: RFIDTagScannedIntegrationEvent is published by the TenantARFIDService.

been scanned. The use of this endpoint also triggers RFIDTagScanned-IntegrationEvent which is then consumed by the RFIDTagScanned-IntegrationEventHandler in TenantB 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 republished to the Event Bus, and it is processed normally by the main product. The customization result, before all the RFID tags are scanned, in the UI, can be seen in Fig. 19, and after the RFID tags have been scanned can be seen in Fig. 20.

As the external services themselves are not the focus of this work, they are just simple mock-up services. For the second customization scenario, an endpoint has been added so that whenever it is called with an order number, essentially sets that all order lines have been scanned for that order. The source code of our proof-of-concept can be found in GitHub.[7]

So far, we have shown that all the customization scenarios are based on leveraging the events in the application. Indeed, we can introduce tenant-specific events for event-based customization scenarios. All the events are isolated so that each tenant is only able to interact with their own events, which means that tenant isolation is still preserved. Normally, such event-based (asynchronous) customization is triggered by an existing event in the main product. On the other hand, the API-based (synchronous) way of customization is mainly leveraging on the APIs (methods) available in the back-end microservices of the main software product. Because customization scenarios are unpredictable, we cannot say how many APIs will be called or how many events will be used for customization purposes. Therefore, it is fair to assume all can be used for customization purposes. Table 2 shows that for the eShopOnContainers application, on average, the number of controller methods is quite similar to the number of events in the back-end microservices. Therefore, using the event-based customization approach is highly recommended whenever possible, to reduce the traffic of API calls to the back-end microservices of the main product. The synchronous way of customization via API calls is still essential for customizing the GUI as presented in [7]. Using API calls is also useful in querying more "contexts" from the main software product for customization purposes if needed. Even though, our experiments on the two scenarios above showed that using the existing "contexts" in the events themselves is already enough for those scenarios. It is also possible to use the synchronous customization approach [7] to make an API call to a tenant-specific customization service that will generate tenant-specific events for triggering the event-based customization flow as presented earlier.

Table 2
methods vs. # events in the back-end microservices of eShopOnContainers.

	Basket	*Catalog*	*Ordering*	*Payment*	*Webhooks*	Avg
# Controller Methods	5	13	7	0	5	**6**
# Events	3	5	14	3	6	**6.2**

7. Related work

7.1. Customization

7.1.1. Single-tenant to multi-tenant:

Migrating an application from single to multi-tenant is a large undertaking. Furda et al. [35] describe an approach for migrating legacy single-tenant applications to multi-tenant. From a legacy system that supports only a single tenant, the approach focuses on changing the architecture of the application into one (e.g., Model-View-Controller) that is better suited for multi-tenancy. Then, the follow-up phase is to enable multi-tenancy, e.g., migrating to multi-tenant view. In [38], the authors propose a lightweight reengineering approach

[7] https://github.com/Espent1004/eShopOnContainersCustomised

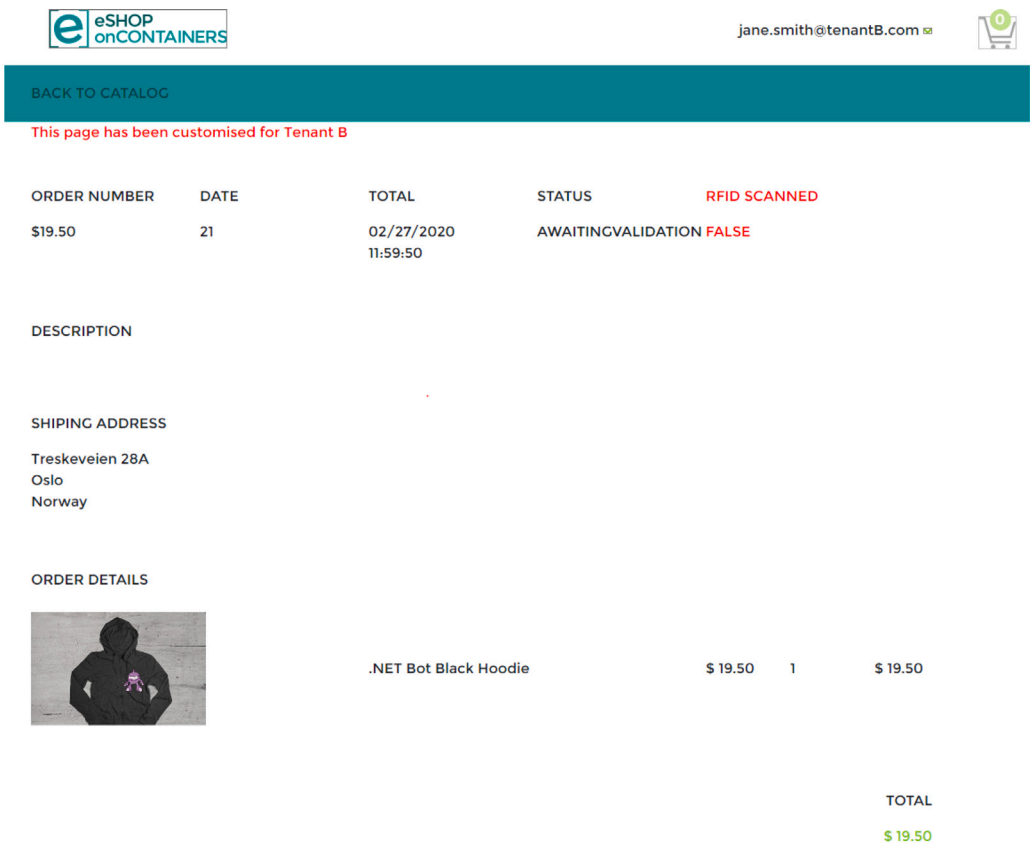


Fig. 19. Customization of Tenant B: Before all the RFID tags are scanned.

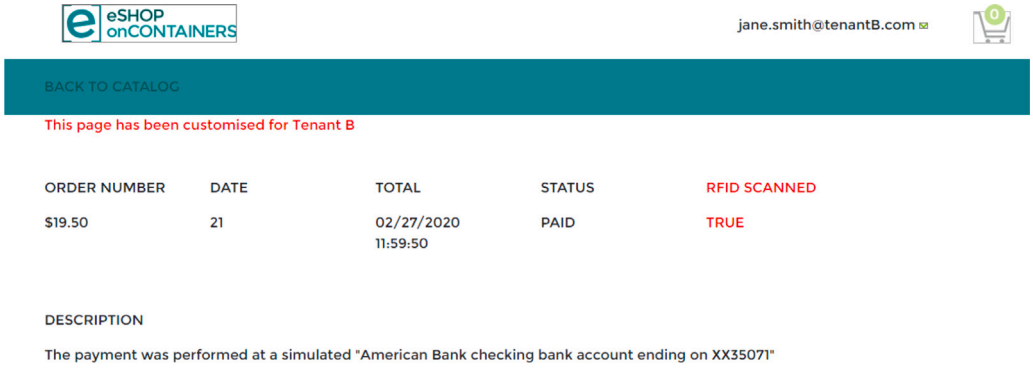


Fig. 20. Customization of Tenant B: After all the RFID tags have been scanned.

to migrate a single-tenant software system into a multi-tenant one. The targeted multi-tenant software system can provide capabilities for tenant-specific layout styles, configuration and data management. Our migration approach enables tenant-specific customization, which is beyond configuration capabilities.

7.1.2. Migrating to microservices:

A key challenge with multi-tenant applications is that the application has to deliver a shared product to multiple tenants, resulting in one-size-fits-all solutions even though the different user-groups might have different needs from the same application [31] . A way to solve this is by allowing the individual users to customize different aspects of the application for their needs, which is conflicting with multi-tenancy [32]: Multi-tenant applications allow different user groups (tenants) to share resources, but ad-hoc handling of changes related to one specific tenant can potentially affect all the tenants of the application.

The customization-driven aspect makes our approach different from other migration approaches like [2,3,36]. The approach in [2] uses migration patterns for managing service decomposition and data isolation and replication. While the approach in [3] is incremental re-engineering of a mission critical banking system that led to reduced complexity, lower coupling, higher cohesion, and a simplified integration. Each of the approaches has its specific contexts where they are the most suitable. These approaches focus on migrating live systems or systems that have been used extensively by organizations. For our prototype, we found the blueprint approach most suitable due to the "stale" state of the application. By stale, we mean that the application is no longer actively developed. All the approaches we found follow a similar pattern, made up of three-phases: Reverse-engineering, transformation, and implementation. What separates them is the focus they put on transforming and moving the existing functionality into new services.

7.2. Customization

The notion of customizable SaaS applications with explicit support for variability management has been proposed and explored extensively [11]. Many techniques are used in practice to enable customization of a SaaS application. The architectural complexity is the fact that tenant-provided customization cannot be anticipated at design-time, and the requirement of tenant isolation. There are many technical approach to addressing these complexities, such as design patterns, dependency injection (DI), software product lines (SPL), or API. To the best of our knowledge, while these approaches help to pre-defined customization at design time, they do not have sufficient support for the complex and unanticipated behavioral coordination between the custom code and the main product at runtime.

Software Product Line (SPL) [12] captures the variety of usages in a global variability model, and actual products are generated based on the configuration of the variability model. Traditional SPL approaches targets all the potential user requirements by the software vendor, and thus does not apply to our definition of customization. Dynamic SPL [39] is closer to customization,using variability models for runtime adaptation [40]. However, such model-based configuration is in a much higher abstraction level than programming [41], and does not support the introduction of new coordination behavior between custom service and the main product.

The majority of SaaS customization approaches focus on a high-level modification of the service composition. Mietzner and Leymann [13] 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 [42] follows the same assumption, but propose multiple layers of compositions. All the composite services (defined by processes) are customizable until reaching atomic services, which are, again, assumed to be provided by the vendors. Nguyen et al. [43] develop the same idea, and introduce a service container to manage the life-cycle of composite services and reduce the time to switch between tenants at runtime. These service composition approaches all support customization in a coarse-grain way, and rely on the vendors to provide the adequate "atomic services" as the building blocks for customized composite services.

As market leading SaaS for CRM and ERP, the Salesforce platform and the Oracle NetSuite provide built-in scripting languages [15–17] for fine-grained, code-level customization. Since these scripting languages are not exposed to the same execution context as the main service, the customization capability is defined by the underlying APIs of the main service. To maximize the customization capability, both vendors provide very extensive and sophisticated APIs, which is costly and not affordable by smaller vendors. In contrary, the microservices-based approach requires lower investment from the vendors in advance.

Middleware techniques can also support the customization of SaaS. Guo et al. [44] 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. [14] implemented such a customization-enabling middleware. In particular, they allow customers to develop custom code using the same language as the main product, and use Dependency Injection to dynamically inject these custom Java class into the main service, depending on the current tenant. Later work from the same group [9] develop this idea and focus on the challenges of performance isolation and latency of custom code switching. The dependency injection way for customization allows the custom code developers to introduce arbitrary coordination behavior 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, a misbehavior of the custom code only fails the underlying container, and the main product only perceives a network error, which will not affect other tenants. Besides, external microservices ease management: scaling independently resource-consuming customization and eventually billing tenants accordingly.

To the best of our knowledge, there are no commercial off-the-shelf (COTS) offerings that have support for event-based customization. The novelty of our work is an event-based customization approach for multi-tenant SaaS, which can be implemented with different COTS offerings such as Azure Service bus or Amazon EventBridge. We have demonstrated our approach using RabbitMQ, but it is also easy to switch to Azure Service Bus as shown in the eShopOnContainers architecture. The Azure Relay service can make the implementation of our approach even easier because it allows tenants to securely expose customization microservices that run in their corporate network to the main product SaaS that is running on the public cloud. However, the Azure Relay service itself does not offer any customization capability. This means that using our approach can help software vendors to implement the Azure Service Bus and Azure Relay service with event-based customization capability for multi-tenant SaaS. It should also be quite straightforward to use Amazon EventBridge[8] because it has support for connecting "applications together using data from your own applications, integrated SaaS applications, and AWS services". Our event-based customization approach must orchestrate not only events of the main product (SaaS) for multiple tenants but more importantly the events of tenant-specific microservices that are coordinated with the original events of the main product for that specific tenant only, for the customization purposes of that tenant only.

8. Conclusions

In this paper, we have presented a combined approach for migrating monoliths to microservices-based customizable Cloud-native SaaS, empowered with event-based customization ability. Our approach splits the migration into three stages, where we first analyze and break down the application into bounded contexts separating the different responsibilities and application areas. After the analysis, we start transforming the infrastructure to fit the microservice architecture. This includes migrating information from databases related to the contexts discussed above, and setting up additional components necessary to support the new services, like the API gateway and the message exchange. Finally, we implement the functionality from the contexts as separate services and connect them to infrastructure. After that, we add the infrastructure necessary for multi-tenancy and tenant-specific customization.

Our event-based customization approach is part of the 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. The primary concerns with multi-tenancy are avoiding noisy neighbors and ensuring that the tenant data is sufficiently isolated. Moving the customization outside the same execution context of the main product solves this. Customization no longer compete for computing resources with the main application, and the data of other tenants remain entirely isolated from the customization code. Our event-based customization approach makes sure of tenant-isolation, which is crucial in practice for SaaS vendors. This event-based deep customization 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.

[8] https://aws.amazon.com/eventbridge/

CRediT authorship contribution statement

Espen Tønnessen Nordli: Conceptualization, Methodology, Software, Writing – original draft, Visualization, Investigation. **Sindre Grønstøl Haugeland:** Software, Writing – original draft, Visualization. **Phu H. Nguyen:** Conceptualization, Supervision, Methodology, Software, Writing – review & editing, Visualization, Investigation, Validation. **Hui Song:** Conceptualization, Supervision, Validation, Writing – review & editing. **Franck Chauvel:** Conceptualization, Supervision, Writing – review & editing.

Declaration of competing interest

One or more of the authors of this paper have disclosed potential or pertinent conflicts of interest, which may include receipt of payment, either direct or indirect, institutional support, or association with an entity in the biomedical field which may be perceived to have potential conflict of interest with this work. For full disclosure statements refer to https://doi.org/10.1016/j.infsof.2023.107230. Phu H. Nguyen reports financial support was provided by Horizon 2020. Hui Song reports a relationship with Research Council of Norway that includes: funding grants.

Data availability

No data was used for the research described in the article.

References

[1] IDG, 2018 Cloud Computing Survey, Tech. rep., 2018, URL https://www.idg.com/tools-for-marketers/2018-cloud-computing-survey/.

[2] A. Henry, Y. Ridene, Migrating to microservices, in: A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, A. Sadovykh (Eds.), Microservices: Science and Engineering, Springer International Publishing, Cham, 2020, pp. 45–72.

[3] M. Mazzara, N. Dragoni, A. Bucchiarone, A. Giaretta, S.T. Larsen, S. Dustdar, Microservices: Migration of a mission critical system, IEEE Trans. Serv. Comput. (2018) 1, http://dx.doi.org/10.1109/TSC.2018.2889087.

[4] S.S. De Toledo, A. Martini, P.H. Nguyen, D.I.K. Sjøberg, Accumulation and prioritization of architectural debt in three companies migrating to microservices, IEEE Access 10 (2022) 37422–37445, http://dx.doi.org/10.1109/ACCESS.2022.3158648.

[5] D. Taibi, V. Lenarduzzi, C. Pahl, Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation, IEEE Cloud Comput. 4 (5) (2017) 22–32, http://dx.doi.org/10.1109/MCC.2017.4250931.

[6] A. Balalaie, A. Heydarnoori, P. Jamshidi, Microservices architecture enables DevOps: Migration to a cloud-native architecture, IEEE Softw. 33 (3) (2016) 42–52, http://dx.doi.org/10.1109/MS.2016.64.

[7] P.H. Nguyen, H. Song, F. Chauvel, R. Muller, S. Boyar, E. Levin, 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, in: ESEC/FSE 2019, Association for Computing Machinery, New York, NY, USA, 2019, pp. 905–915, http://dx.doi.org/10.1145/3338906.3340452.

[8] H. Song, P.H. Nguyen, F. Chauvel, J. Glattetre, T. Schjerpen, Customizing multi-tenant SaaS by microservices: A reference architecture, in: 2019 IEEE International Conference on Web Services, ICWS, 2019, pp. 446–448, http://dx.doi.org/10.1109/ICWS.2019.00081.

[9] S. Walraven, D.V. Landuyt, E. Truyen, K. Handekyn, W. Joosen, Efficient customization of multi-tenant software-as-a-service applications with service lines, J. Syst. Softw. 91 (2014) 48–62, http://dx.doi.org/10.1016/j.jss.2014.01.021, URL http://www.sciencedirect.com/science/article/pii/S0164121214000326.

[10] S.G. Haugeland, P.H. Nguyen, H. Song, F. Chauvel, Migrating monoliths to microservices-based customizable multi-tenant cloud-native apps, in: 2021 47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA, 2021, pp. 170–177, http://dx.doi.org/10.1109/SEAA53835.2021.00030.

[11] J. Kabbedijk, C.-P. Bezemer, S. Jansen, A. Zaidman, Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective, J. Syst. Softw. 100 (2015) 139–148.

[12] K. Pohl, G. Böckle, F.J. van Der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer Science & Business Media, 2005.

[13] R. Mietzner, F. Leymann, Generation of BPEL customization processes for SaaS applications from variability descriptors, in: Services Computing, 2008. SCC'08. IEEE International Conference on, Vol. 2, IEEE, 2008, pp. 359–366.

[14] S. Walraven, E. Truyen, W. Joosen, A middleware layer for flexible and cost-efficient multi-tenant applications, in: Proceedings of the 12th International Middleware Conference, International Federation for Information Processing, 2011, pp. 360–379.

[15] Salesforce, Apex developer guide, 2019, URL https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/ (Accessed: 14 April 2019).

[16] T. Kwok, A. Mohindra, Resource calculations with constraints, and placement of tenants and instances for multi-tenant saas applications, in: A. Bouguettaya, I. Krueger, T. Margaria (Eds.), Service-Oriented Computing – ICSOC 2008, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 633–648.

[17] Oracle, Application Development SuiteScript, 2019, URL http://www.netsuite.com/portal/platform/developer/suitescript.shtml (Accessed: 14 April 2019).

[18] J. Thönes, Microservices, IEEE Softw. 32 (1) (2015) 116, http://dx.doi.org/10.1109/MS.2015.11.

[19] S. Newman, Building Microservices: Designing Fine-Grained Systems, " O'Reilly Media, Inc.", 2015.

[20] N. Dragoni, S. Giallorenzo, A.L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, Microservices: Yesterday, today, and tomorrow, in: M. Mazzara, B. Meyer (Eds.), Present and Ulterior Software Engineering, Springer International Publishing, Cham, 2017, pp. 195–216, http://dx.doi.org/10.1007/978-3-319-67425-4_12.

[21] H. Song, F. Chauvel, A. Solberg, Deep customization of multi-tenant saas using intrusive microservices, in: Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, in: ICSE-NIER '18, ACM, New York, NY, USA, 2018, pp. 97–100, http://dx.doi.org/10.1145/3183399.3183407, URL http://doi.acm.org/10.1145/3183399.3183407.

[22] H. Song, F. Chauvel, P.H. Nguyen, Using microservices to customize multi-tenant software-as-a-service, in: Microservices: Science and Engineering, Springer International Publishing, Cham, 2020, pp. 299–331, http://dx.doi.org/10.1007/978-3-030-31646-4_12.

[23] H. Song, P.H. Nguyen, F. Chauvel, Using Microservices to Customize Multi-Tenant SaaS: From Intrusive to Non-Intrusive, in: L. Cruz-Filipe, S. Giallorenzo, F. Montesi, M. Peressotti, F. Rademacher, S. Sachweh (Eds.), Joint Post-Proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019), in: OpenAccess Series in Informatics (OASIcs), vol.78, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2020, pp. 1:1–1:18, http://dx.doi.org/10.4230/OASIcs.Microservices.2017-2019.1, URL https://drops.dagstuhl.de/opus/volltexte/2020/11823.

[24] P.H. Nguyen, H. Song, F. Chauvel, E. Levin, Towards customizing multi-tenant cloud applications using non-intrusive microservices, in: The 2nd International Conference on Microservices, 2019.

[25] E.T. Nordli, P.H. Nguyen, F. Chauvel, H. Song, Event-based customization of multi-tenant saas using microservices, in: S. Bliudze, L. Bocchi (Eds.), Coordination Models and Languages, Springer International Publishing, Cham, 2020, pp. 171–180.

[26] A. Freeman, Pro Asp. Net Core Mvc, A Press, 2016.

[27] R. Kazman, S.G. Woods, S.J. Carriere, Requirements for integrating software architecture and reengineering models: CORUM II, in: Proceedings Fifth Working Conference on Reverse Engineering (Cat. No.98TB100261), 1998, pp. 154–163.

[28] M. Fowler, Strangler Fig Application, 2004, URL https://martinfowler.com/bliki/StranglerFigApplication.html.

[29] E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2004.

[30] P. Di Francesco, P. Lago, I. Malavolta, Migrating towards microservice architectures: An industrial survey, in: 2018 IEEE International Conference on Software Architecture, ICSA, 2018, pp. 29–2909, http://dx.doi.org/10.1109/ICSA.2018.00012.

[31] T. Kwok, T. Nguyen, L. Lam, A software as a service with multi-tenancy support for an electronic contract management application, in: 2008 IEEE International Conference on Services Computing, Vol. 2, 2008, pp. 179–186, http://dx.doi.org/10.1109/SCC.2008.138.

[32] S. Walraven, D. Van Landuyt, E. Truyen, K. Handekyn, W. Joosen, Efficient customization of multi-tenant Software-as-a-Service applications with service lines, J. Syst. Softw. 91 (2014) 48–62, http://dx.doi.org/10.1016/j.jss.2014.01.021, URL http://www.sciencedirect.com/science/article/pii/S0164121214000326.

[33] E. Wolff, Migrating monoliths to microservices: A survey of approaches, in: International Conference on Microservices 2019, 2019.

[34] A. Christoforou, L. Odysseos, A.S. Andreou, Migration of software components to microservices: Matching and synthesis, in: 14th International Conference on Evaluation of Novel Approaches to Software Engineering, 2019.

[35] A. Furda, C. Fidge, A. Barros, O. Zimmermann, Chapter 13 - reengineering data-centric information systems for the cloud – a method and architectural patterns promoting multitenancy, in: I. Mistrik, R. Bahsoon, N. Ali, M. Heisel, B. Maxim (Eds.), Software Architecture for Big Data and the Cloud, Morgan Kaufmann, Boston, 2017, pp. 227–251, http://dx.doi.org/10.1016/B978-0-12-805467-3.00013-2, URL https://www.sciencedirect.com/science/article/pii/B9780128054673000132.

[36] D. Taibi, F. Auer, V. Lenarduzzi, M. Felderer, From monolithic systems to microservices: An assessment framework, 2019, arXiv preprint arXiv:1909.08933.

[37] S. Panichella, M.I. Rahman, D. Taibi, Structural coupling for microservices, 2021, arXiv preprint arXiv:2103.04674.
[38] C. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, A. Hart, Enabling multi-tenancy: An industrial experience report, in: 2010 IEEE International Conference on Software Maintenance, 2010, pp. 1–8.
[39] S. Hallsteinsen, M. Hinchey, S. Park, K. Schmid, Dynamic software product lines, Computer 41 (4) (2008).
[40] J. Lee, G. Kotonya, Combining service-orientation with product line engineering, IEEE Softw. 27 (3) (2010) 35–41.
[41] M.A. Rothenberger, M. Srite, An investigation of customization in ERP system implementations, IEEE Trans. Eng. Manage. 56 (4) (2009) 663–676.
[42] W. Tsai, X. Sun, Saas multi-tenant application customization, in: 2013 IEEE Seventh International Symposium on Service-Oriented System Engineering, 2013, pp. 1–12, http://dx.doi.org/10.1109/SOSE.2013.44.
[43] T. Nguyen, A. Colman, J. Han, Enabling the delivery of customizable web services, in: 2012 IEEE 19th International Conference on Web Services, 2012, pp. 138–145, http://dx.doi.org/10.1109/ICWS.2012.23.
[44] C.J. Guo, W. Sun, Y. Huang, Z.H. Wang, B. Gao, 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, IEEE, 2007, pp. 551–558.