Using Intrusive Microservices to Enable Deep Customization of Multi-tenant SaaS

Franck Chauvel, Arnor Solberg

Citation:
Franck Chauvel, Arnor Solberg(2018). Using Intrusive Microservices to Enable Deep Customization of Multi-tenant SaaS. In 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC), Coimbra, Portugal, 4-7 Sept. 2018, pp 30-37 DOI: 10.1109/QUATIC.2018.00015

Using Intrusive Microservices to Enable Deep Customization of Multi-Tenant SaaS

Franck Chauvel, Arnor Solberg
SINTEF, Norway.
{Franck.Chauvel, Arnor.Solberg}@sintef.no

***Abstract*—Enterprise software applications need to be customized in order to meet special requirements from customers. When the customization requirements are beyond the prediction of vendors, deep customization is needed, and traditionally customers do deep customizations by directly modifying the application source code. When the applications are moving from on-premises to multi-tenant Software as a Service, directly changing code is not feasible because many customers are sharing one instance of the application code. In this paper, we present a new approach to enable deep customization on multi-tenant SaaS, using intrusive microservices. The custom code is implemented as an isolated and self-contained microservice running beside the main service, and it uses callback code to intrusively execute queries or commands inside the main service. We present the key techniques behind intrusive microservices and illustrate how turned an open source online shopping application into a deeply customizable multi-tenant service.**

I. Introduction

Modern companies rely more and more on software applications to support their day-to-day management on sales, human resource, financial, etc. Since every company is unique in terms of organization, process or culture, off-the-shelf software cannot perfectly fit every company. Some company eventually needs to *customize* the software to meet their requirements. Customization is a special type of software development. It is performed either by the in-house developers of the customer or by third-party consultants hired by the customer.

Customization is often done in a controlled way. The software vendors predict where and how their applications may be customized, and provide their customers with the APIs, extension points or configuration choices. However, it is not possible for the vendors to predict all the potential customization requirements, and therefore, there are always customers whose requirements cannot be met by using the provided customization supports. These customers have to do *deep customization*, beyond the vendor's prediction.

Traditionally, software vendors support deep customization by allowing the customers to directly modify their source code. The customers acquire a special license of the application, re-develop it based on the original source code, and then deploy it on their own premises. In an empirical study on 8 companies deploying ERP systems, Rothenberger et al. [1] observed that in most cases ERP adopters end up implementing heavy customizations involving code modifications. The two software vendors that commissioned this research also have many customers who have made deep customizations to their products [2]. In addition to the flexibility, deep customization also has benefited of remaining the customized product as an integral piece. Moreover, vendors do not need to invest a lot of resource to design and implement the sophisticated API or extension points.

However, as the enterprise software industry is moving from single-tenant on-premises applications to cloud-based multi-tenant Software as a Service (SaaS), deep customization by directly modifying the source code is not feasible any more. In a multi-tenant SaaS, one instance of the software application is shared by many customers (also known as tenants). No customer monopolizes the source code, and therefore the customer's modification on the source code will immediately affect the other customers. Due to this limitation, the mainstream multi-tenant SaaS vendors such as Salesforce and Oracle NetSuite only support controlled customization.

In this paper, we present a novel approach to implement deep customization on multi-tenant SaaS, using *intrusive custom microservices*. Customers have the "read-only" access to the source code of the main service, and can choose fine-grained pieces in the code base, such as a C# method, to customize. Instead of directly modifying the code, they re-develop the code and wrap it as a self-contained microservice running alongside the main service. The execution to the original pieces will be redirected to the custom microservice at runtime. These custom microservices are intrusive to the main service, by sending callback code to the main service. The main service executes the callback code under the same context as the original code. In this way, the custom code is theoretically capable of doing anything the original code can do.

This approach achieves both the *isolation* required by multi-tenancy and the *assimilation* required by deep customization. On the one hand, the custom code is running separately from the main service, and can be deployed dynamically without rebooting the main service. On the other hand, the custom code has the same expression power as the original source code. Therefore customers can implement the customization in the same way as if they

are vendor developers.

The remainder is organized as follows. Section II uses a sample on-line shopping system to explain the requirement for deep customization. Section III gives an overview of the approach, and Section IV elaborates it with key techniques. Section V evaluates the approach by applying it on the sample open source shopping application. Section VI discusses related approaches and Section VII concludes the paper.

II. Motivating Example

In this section, we present an open source online shopping application, the MusicStore, as an exemple of the challenges that obstacle the development of deep customization on multi-tenant SaaS.

The Microsoft MusicStore [3] is an official test application for ASP.NET Core, the next generation web development framework by Microsoft. MusicStore simulates the essential functionality of online shopping, such as user management, catalogue, shopping cart and checkout, using the music albums as its sample products. We use MusicStore as the backbone software to provide a multi-tenant music selling SaaS: The owner of a small music shop can apply to be a tenant of the service, and starts selling its albums through the service, without deploying and maintaining a MusicStore instances on their own premise. This example well represents the current transition that is undertaken by both companies that commission this research.

A major challenge to provide the multi-tenant MusicStore SaaS is to support the customization by different tenants. Let us consider a simple customization use case, where the tenant wants to introduce a donation feature into her own shop: When an end-user buys an album, the shop will ask him how much he wants to donate to the charity, and the donation is automatically added to the total price of her shopping cart. The use case comprises the following three sub use cases: 1) *Add donation.* When the user adds an item to the shopping cart, a new page should pop up to let her choose how much she would like to donate, from 0 to 100% of the album's price. 2) *Display donations.* In the shopping cart overview page, the amount of donations should be shown for each shopping cart item. 3) *Get Total Price.* When the system calculates the total price of a shopping cart, the donations should be accounted for.

These use cases implies changes of the MusicStore across three layers. In the UI layer, we need to create a new page and change the component in an existing page (adding a "donation" column in a table). In the business logic layer, we need to change the logic of total price calculation. In the data layer, we need to store the donation amount for each shopping cart item. These changes are beyond the predication of the MusicStore developers, and deep customization is needed. As we are offering MusicStore as a multi-tenant SaaS, direct code modification is not feasible, because the same MusicStore instance will be shared by this tenant together with many others.

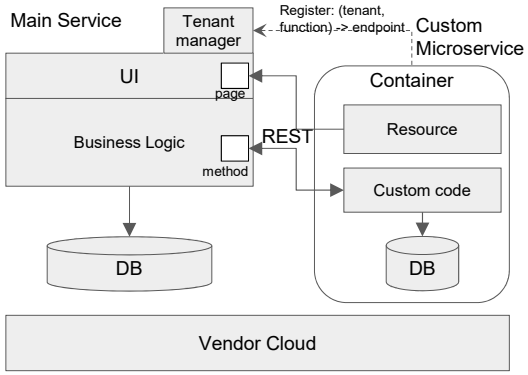

Fig. 1. The structural overview of intrusive custom microservices

We detail below our deep customization approach and how it turns the MusicStore into a customisation multi-tenant SaaS.

III. Approach Overview

We rely on intrusive custom microservices to realize deep customization on multi-tenant SaaS. Figure 1 illustrates the basic structure.

The main service is a running instance of the standard product, provided by the service vendor. The service is hosted in a public or private cloud, managed by the service vendor. The customization by each tenant is running as one or more self-contained microservices, hosted in the same vendor cloud. Each custom microservice re-implements a small number of fine-grained structures within the main service source code, such as C# methods in the business logic level, or HTML templates in the UI level. The custom microservice can choose to maintain a lightweight database within itself. The main service and a custom microservice communicate via REST invocations, and do not share data or resources directly.

A *tenant manager* registers which microservice supersedes which part of the main product and for which tenant, so that when a tenant request arrives, the service will forward it to the registered microservice, instead of executing the original standard code.

These custom microservices are intrusive to the main service through callback code, which are small code snippets sent from custom microservice to main service in plain text within the REST invocation, that are executed by the latter under the same execution context as the to-be-replaced standard code. The microservices use callback code to query data from the main service and to modify the behaviour of the service.

IV. Supporting Intrusive Custom Microservices

We explain below the key techniques according to the .NET Core stack, but the challenges and solutions are generic to other stacks.

Following the common practice of microservices, we made the following high-level design decisions. 1) The main service and the custom code unit communicate with each other only via REST invocations. 2) The data exchanged between the product and custom code unit are JSON documents, and should be small in size. 3) The main service does not make any assumption on what the custom code will need and what it will do.

Deep customisation has to provide ultimate flexibility to custom code developers, which means that if a custom code unit is to replace a standard method, then the custom code should be able to query or manipulate any data that the original method body can query or manipulate. In other words, the custom code is exposed to the same context as the standard source code. However, the design decisions we identified above determines that the custom cannot directly query or manipulate the context, as it is not practical to transfer the entire C# objects in the context through JSON. We address this issue by employing a callback code mechanism and a multi-step communication with on-demand data transfer.

In the rest of this section, we first give an overview of the interactions between the product and custom code. After that, we detail the syntax and execution of callback code and how it works on the context. Then, we define the concrete REST API of a custom code unit. Finally, we describe the deployment and lifecycle of the microservices.

We present the customization mechanism based on a sample customization, i.e., the Step 3 in Section II, to compute the total price of the shopping cart.

A. Communication between main service and custom code

The original behaviour of a method is overridden by a custom code unit through a sequence of REST communications between the main product and the custom microservice.

The communications are initiated and driven by the interceptor, which is injected into every method in the main product. Whenever a method is invoked, the interceptor will pause the execution of the original method body, and check first with the tenant manager if the current tenant has registered a custom code unit registered for this method. If so, the tenant manager will return an endpoint. The interceptor then starts the communication with the custom code by invoking this endpoint with a POST request. As the first step of the communication, this request is empty, because the product does not know what data is required by the custom code. On receiving the POST request, the custom microservice executes the custom code corresponding to the endpoint, and sends back the response based on the execution. In the response, the custom code instructs the interceptor on how to change the context of the original method, what is the next step, and what data it requires in the next step. The interceptor then operates the context as indicated, prepares the required data and uses them as parameters

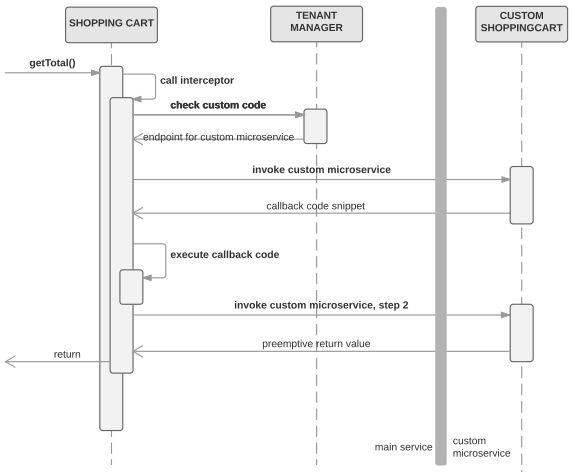

Fig. 2. Communication between main service and custom code

to invoke the next step. The communication terminates when the custom code responses without a subsequent step. Moreover, as the response of the last step, the custom microservice can provide a pre-emptive return value. If so, the interceptor will return this value without executing the original method body. On the other hand, if the last response does not have a return value, the interceptor will continue to execute the original method body, under the manipulated context.

Figure 2 illustrates the communication based on the"get total price" use case. When an end-user checks out, the main service processes the request and finally invokes the `GetTotal` method defined in the standard product to calculate the total price. This method invocation is intercepted, and by consulting the `TenantManager` it receives an endpoint to the `Donation ShoppingCart` microservice. The interceptor invokes this endpoint with an empty POST request. In the first step, the custom code does nothing but directly sends back a response with callback code to ask the main product what are the items in the current shopping cart. It also provides the endpoint to the second step. The interceptor executes the callback code to query the items, and send them as JSON objects to the custom microservice by invoking the endpoint for the second step. In the second step, the custom microservice receives the request and obtains the current items and their prices. Then it queries its own database to get the donation amount recorded for each item, and sums them up into a total price for the shopping cart. As the second response, the custom microservice instructs the interceptor to use the new total price as the return value. The interceptor returns this value to its caller without executing the original method body.

B. Intrusive Callback Code

This section explains the callback code mechanism, which enables the custom code from a remote service to

query and manipulate the data and state of the main service. In summary, a callback code is an instruction that is created by the custom code and executed by the interceptor in the main product under the same context of the original method body. The fact that the callback code and the original method body are exposed to the same context is essential to the requirement of deep customization. In the rest of this section, we first define the execution context of the callback code, and then introduce the language to write callback code and how they are executed by the interceptor.

1) Execution context of callback code: The execution context is the environment that the callback code is executed in, i.e., a set of objects that the callback code can refer to. In the .NET stack, this context is identical to the context of the to-be-replaced C# method.

When a method in the main service is invoked, the context needed to evaluate the original method body includes all the parameters passed through the invocation. If the method is not static, the host object is also passed as an implicit parameter. The method body refers to the parameters using the corresponding argument names, or the `this` identifier for the host object, in order to read or change the states of these objects. After the method body is executed, it may extend the context by a return value, which is passed by to the caller. In addition, the method body is also exposed to a global context which consists of all the classes under the current class loader, together with all the static methods and fields defined in these classes.

When the interceptor of a standard method receives a callback code, it inherits the context of the original method, and uses it to executes the callback code. This means that the callback code can refer to the parameters, the host object and all the visible classes. In this way, the callback code is able to obtain and modify the data and state of the main service, just in the same way as the original method body. For example, in the get total price use case, after the first step, the callback code can query out the current shopping cart items by calling the `GetItems` method on the host object. In the second step, the callback code add the new total price value into the context, which will be used as the return value.

2) Callback code language: A callback code snippet is sent from the custom service to the main product as plain text, and then in the product it is compiled into an executable code and evaluated at runtime. We design and implement a simple language to write callback code, based on the DynamicLinq library [4]. DynamicLinq is a .NET library that dynamically compiles a piece of C# query from plain text into an executable function (a Delegate in .NET). The C# query supported by DynamicLinq is essentially a reference to an object or a chain of method invocations from an object. We make simple extension to DynamicLinq to extend its expression power, and Figure 3 summarizes the syntax.

Query is a piece of code that returns a value. A query

```
1 Query ::= '$'ContextVar | Query'.'Method Params
2     | Query'.'Field | Query (+|-|==|>|...) Query
3     | 'IF' Query 'THEN' Query 'ELSE' Query
4     | [Query FOR '$'Id OF Query]
5 Params ::= '()' | '(' Query (, Query)* ')'
6 Instr ::= 'CALL' Query'.'Method Params
7     | 'SET' Query.Field '=' Query
```

Fig. 3. Syntax

```
context_op1 = {
  "rawItems": "$this.GetCartItems().Result",
  "items": "$rawItems.Select(i => \
     new{id=i.CartItemId, price=i.Album.Price})" }
context_op2 ={ "total": "_VAL_number 25.99",
  "msg": "_VAL_string New total price is 25.99",
  "void": "CALL System.Console.WriteLine($msg)",
  "returnx": "$total" }
```

Fig. 4. Sample context operations

can be simply a reference to a context variable, starting with a $ and followed by the variable name. From a query, we can invoke a method that is defined on the type of its returned object. As long as the method has a return value, the invocation is still a query. Similarly, we can access the field of an object or do operations of two objects. A conditional branch returns one of the alternative queries, and list comprehension iterates a collection and use its items to make a new collection.

A query must always return a value, and if we invoke void-typed methods, DynamicLinq will raise an exception. To address this issue, we introduce a new concept called "Instructions". We can use a CALL keyword to invoke a void-typed method on the returned object of a query, or use a SET keyword to assign value to a field of this object.

3) Context operation: The callback code language can be used to write a single instruction. After each step, the custom code send back a group of instructions, which we call the *context operation*. A context operation is a dictionary, and each of its items comprises a *key* in type of string and a *value* which is either an instruction or a text. We support the following types of items. When the key is the name of a context variable, the value could be either a query or a string starting with `_VAL_` followed by a type (string, number, date, boolean) and a value of this type. The query result or the direct value will be assigned to the key variable. If the specified variable does not exist in the current context, we create the variable first. The value can be an instruction, and then we will only evaluate the instruction (which has effect on the context) without assigning a value to any context variable. When the key is a keyword "returnx", we will use the resulted value of the query as the return value of the original method.

Figure 4 shows two sample context operations, returned by the two steps of the GetTotal example, respectively. We present the two dictionaries in JSON format, as it is in the actual REST communication.

The first context operation is to query the items in the current shopping cart. In the first line, we query all the items and assign the list of items to a temporary context variable named `rawItems`. The query uses an existing context variable, `this`, which represents the current Shopping Cart, and invokes the `GetCartItems` method on the object. In the second line, we extract only the id and the price from each item, because these are the only information useful to the custom code.

The second context operation prints a log about the new total price, and modifies the original return value. The first two lines transfer the total price and the log message from the custom code to the main product. The two values are in types of number and string, respectively. The third line invokes a static method to print the message to the console. It utilizes both a context variable (`$msg`) and a static class (`System.Console`). Finally, the last line instructs the main service to use the value in the `total` variable as the new return value.

C. Customization protocol

Each custom code unit replaces one method in the main product, using several steps. Each step is triggered by the POST request to a unique URL. The customization protocol defines the input and output of these steps.

The input to each custom code step is a POST request, with an optional JSON document as the request body. The JSON document is a key-value map that contains data obtained from the main product. What data is carried by a request body is defined by the output of the previous step.

The output of each step is also a JSON document, in a predefined format. The whole JSON document is called a "Manual" (which means that the main service needs to work accordingly), and it contains three optional properties. The `context` property contains a context operation, as defined in the previous section. The `nextcall` property defines the next step, where `function` is the URL of the subsequent step, and `body` provides an additional context operation, where the variables and their values will be passed to the POST request for the subsequent step. The data type of the output is extensible. We will add new properties later on to introduce more functionalities into the customization.

Figure 5 shows a sample custom code unit developed in TypeScript under this protocol, which implements the GetTotal customization. The custom code unit is implemented as a TypeScript object called `gettotalcc`. In the object, the `endpoint` and `mainhandler` defines the first step. In this step, the custom code asks the main product to query out the current shopping cart items (see `context_op1` in Figure 4). After that, it sets up the next step as `compute`, and the request body contains one parameter that is the queried items. In this example, we only have one subsequent step. In this step, we first get the items from the request body, which has been automatically

```
var gettotalcc = new cirrusapi.CustomCode()
gettotalcc.endpoint = "/shoppingcartx/gettotal";
gettotalcc.mainhandler = (req, res) => {
  res.json({
    context: ... //context_op1
    nextcall: {
      body: {items: '$items'},
      function: 'compute',
    }}); }
gettotalcc.steps["compute"] = (req, res) =>{
  var items = req.body.items;
  var total = 0;
  items.forEach(item => {
    var id = item.id
    ... //query own db and compute total
    if (/*all items handled*/)
      res.json({
        context: ...#context_op2
      }); }
```

Fig. 5. Sample custom code unit for GetTotal price

decoded into an array of objects in JavaScript. For each object (a shoppingcart item) in the array, we get the item id and price, query the custom database to get the donation amount, and add it into the total value. When all the items has been counted, we return a new manual, whose context operator is the one defined as `context_op2` in Figure 4.

D. UI Customization

Our customization mechanism is focused on the business logic, i.e., how to replace a method in the main product. The customization of UI is driven by the business logic, based the MVC structure in ASP.NET.

In ASP.NET, every browser request is received and handled by a specific method in a controller class (within the Business Logic layer). If the request leads to a UI, i.e., an HTML page, the method will call the `View` method of the controller class with an identifier to an HTML template that is pre-loaded from a local file. The controller will then interpret the embedded C# code in the template to generate the HTML file, and return it to the browser.

We extend the ASP.NET Core behaviour to allow the load of HTML templates from a remote file, which can be obtained from the custom microservice through an HTTP GET request. After a remote custom template is loaded, it can be used in the same way as the original templates. After that, we can use the normal business logic customization mechanism to make a controller method use the custom template.

The custom view templates are evaluated by the the same generation engine as the standard template. Therefore, the custom view template shares all the resources with the standard templates, such as the reusable view components, the CSS styles and the scripts. This saves customers from re-implementing the common parts and also guarantees that the custom views have the consistent

style with the standard ones. Moreover, the custom templates are also interpreted under the same context as the standard views, and has the access to the controller object. This allows the view to exchange data with the business logic using the standard way in ASP.NET, i.e., through the model and the view data carried by the controller.

E. Custom Microservice Life-Cycle

The custom microservices are hosted and managed by the same cloud vendor than the main service is hosted, for the sake of performance and manageability.

The deployment of custom code is based on the Docker container technology and follows a serverless style. Customers or their consultants deliver to the vendor the source code plus Docker specifications. The latter specifies the supportive platform (such as Node.js or Python, etc.), the auxiliary components (such as databases) and the libraries used by the custom code. The Docker engine in the vendor cloud instantiates a Docker container from the source code and the specification. After a custom microservice is up and running, the vendor cloud registers its customization endpoints to the tenant manager, in two phases.

The entire lifecyles of all the microservices are monitored and managed by the vendor cloud. The cloud monitors and controls the resource consumption of each container, pausing or scaling out the containers when necessary under the service-level agreement. It also restarts the failed containers and reports the errors to the customers. Within the main service, the callback code engine monitors the execution of each context operator and terminates the ones that spend two much resource, or falls into deadlocks.

V. Evaluation

In order to evaluate the approach, we implement the generic mechanisms in Section IV on the MusicStore, transforming it into a deeply customizable SaaS. After that, we tested its customization capability by developing a custom microservice, implementing three customization use cases. The result shows that most of the customization types are supported by our MusicStore, without dedicated APIs or extension points.

A. Implementation of the customizable SaaS

We adapted the source code of MusicStore in two steps, i.e., adding a generic library and perform a code rewriting.

The generic library implements the mechanisms in Section IV, and comprise the following components. A simple *tenant manager* is used to register the mapping from standard methods to custom code. A generic *interceptor* drives the communication between the main product and the custom code. An *callback code interpreter* executes the callback code on the local context. Finally, a *remote RAZOR file provider* loads custom HTML templates that can be obtained through REST requests. The entire library is implemented in 800 lines of C# code.

We perform an automatic code rewriting on MusicStore, adding three lines of code in the beginning of each method. The first line initializes a local context as a Dictionary object and fill it with the method parameters. The second line invokes the generic interceptor with the context and the method name. The third line checks if a return value is available in order to decide whether to skip the original method body. We choose source code rewriting rather than binary instrumentation, for the sake of simplicity.

AAll the effort is focused on generic mechanisms, without specific consideration of the actual customization requirements or features.

B. Sample custom code

On top of the customizable MusicStore, we performed three customization use cases.

- **UC1: Donation** is the one described in Section II, with three sub use cases, i.e., choosing donation, list donations and computing total price.
- **UC2: Visit Counting** records how many times each album has been visited. It has two sub use cases, i.e., recording a visit and showing global statistics.
- **UC3: Real Cover** uses the album title to search the cover picture from Bing Image and replaces the original place-holder picture.

We design these use cases deliberately, in order to achieve a good coverage of the general requirements of customization on Web-based enterprise services.

The two companies that commissioned this research have summarized the changes that their customers require when customizing their web-based ERP and CRM systems. We list these general requirements in the first column of Table I, in three different levels, i.e., the user interface, the business logic and the database. The requirements are in a high abstraction level, and each item represents a category of required changes.

The rest of Table I shows how the three use cases cover the general requirement items. The effect of the customized MusicStore can be seen by a screen-shot video [1]. In the video, we are using a MusicStore service through a fictional tenant named foo@bar.com. We first see the standard way to buy a music album through the MusicStore, i.e., browsing the album detail, add it to the shopping cart, and check the overview of the shopping cart items. After that, we deploy the custom code as a microservice and registered it into the MusicStore via the REST API. The effect of the customization is immediate: When we repeat the same shopping process, we will first see a new cover image of the album (UC3) obtained from Bing.com. The layout is altered to fit the big image, and a remark pops on when the cursor is on the image, which is driven by a new Javascript code snippet added to the page. When we add the album to shopping cart, we are led to a new page to select the donation amount, and then shown a shopping cart overview with additional columns showing donations and a different total price (UC1). Finally, we open a new

[1] https://streamable.com/1b8bh

TABLE I
COVERAGE OF GENERAL REQUIREMENTS FOR CUSTOMIZATION

Requirements	UC1	UC2	UC3	#
User Interface				
Move original control			✓	✓
Remove original control			✓	✓
Add control	✓			✓
Add new page		✓		✓
Replace page	✓			✓
Add scripts			✓	✓
Override scripts				
Business Logic				
Override logic	✓			✓
Add new logic		✓		✓
React to events		✓		✓
Trigger Events				
Link control to data	✓			✓
Execute external service			✓	✓
Database				
Override datasource				
Add field to table	✓			✓
Add new table		✓		✓
Update database	✓	✓		✓
Query database	✓			✓

page to check the statistics about the album visits (UC2). Such visits are recorded in a table with album ID and visit time stamp, and counted afterwords by a new function in the business logic layer. At the end of the video, we log off the tenant foo@bar.com, and the service immediately goes back to the standard behaviour, which shows that the customization only affect one tenant.

The custom code is deployed and registered to the MusicStore dynamically, without rebooting the main service. The customized behaviour is seamlessly integrated into the main service: The new pages and the modified ones all keep the same UI style as the original MusicStore, and are accessed through the standard MusicStore address.

We implemented the three customization scenarios in TypeScript, using the Node.js HTTP server to host the custom microservice. The first two scenarios request data storage, and we used MongoDB as the customer database. The entire custom code include 384 lines of code in 5 TypeScript files (one file for each scenario, plus 2 common files to configure and launch the HTTP server) and 175 lines of new code in 4 Razor HTML templates (of which, 2 templates are new and the other 2 are copy-and-pasted from MusicStore, with 176 lines of code that are not changed).

C. Performance

The intrusive custom microservices does not cause significant performance penalty to the main service.

We did a set of experiments to compare the latency of the user requests by the original service and the customized ones. The latency is defined as the duration from a request is sent from the browser (such as clicking a link) until the new page is ready, and is measured by the Chrome Browser. The latency comprise roughly two parts: loading the page source from the server and rendering the page in the browser. Customization only affects the first part. The loading time of original services is between 5 to 100 milliseconds (ms), and the long ones happens when a page is first requested. The additional latency caused by customization ranges from 10ms to 300ms. The customization without UI (the "get total price" example) causes in average 11ms latency. The ones that has needs a new page can cause up to 300ms additional latency when the page is first accessed, after that the additional latency is normally between 50m to 90m. The additional latency is hardly noticeable to end users, as the time to render the page is in average 2 seconds, no matter customized or not.

The memory consumption of this sample microservice remains stable at 50MB. A further experiment reveals that a 16G RAM laptop can easily host 100 instances of the same microservice.

VI. RELATED WORK

There are many technical ways to software customisation such as design patterns, dependency injection (DI), software product lines (SPL), or API. To the best of our knowledge, while these approaches help predefine customisations at design time, they fail to address the requirements of unforeseen deep customisation.

Software Product Line (SPL) [5] captures the variety of user 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 [6] is closer to customization, and some approaches, such as [7] propose the usage of variability models for customization. However, such model-based configuration is in a much higher abstraction level than programming [1], and does not fit deep customization definition, as the customization points has to predefined by the vendors.

There are many approaches to SaaS customization in the context of service-oriented computing. However, most of approaches are focused on a high-level modification of the service composition. Mietzner and Leymann [8] present a customisation 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 [9] 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. [10] develop the same idea, and introduce a service container to manage the lifecycle of composite services and reduce the time to switch between tenants at runtime. These service composition approaches all support customization in a course grained 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 (Customer Relationship Management) and ERP (Enterprise Resource Planning), the Salesforce platform and the Oracle NetSuite provide built-in scripting languages [11] [12] [13] for fine-grained, code-level customization. Since the scripting language is not exposed to the same execution context as the main service, the customization capability is defined by the APIs of the main service. In order to maximise the customization capability, both vendors provide very extensive and sophisticated APIs, which is costly and not affordable by smaller vendors. In contrary, the intrusive microservices does not require the vendors to spend much time on designing and implementing such APIs.

Middleware techniques are also used to support the customization of SaaS. Guo et al. [14] 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. [15] 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 [16] develop this idea and focus on the challenges of performance isolation and latency of custom code switching. The dependency injection way for customization is close to our work, in terms of the assimilation between custom code and the main service. However, operating the custom code as an external microservice ease performance isolation, a misbehavior of the custonm 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 customisations and eventually billing tenants accurately.

VII. Conclusion

This paper describes an approach to use intrusive microservices for the deep customization of multi-tenant SaaS. We present the key techniques to implement this architecture style based on the .NET technical stack, and evaluate it by enabling deep customization of an open source on-line shopping application.

This paper reveals that deep customization is indeed feasible for multi-tenant SaaS. However, it is never the optimal solution for customization, but rather the last choice when customizing beyond the vendor's prediction is a must. For multi-tenant SaaS, the drawback of deep customizations is mainly twofold. On the one hand, the custom code is tightly coupled with the main service, and therefore, updates of main service may break some custom code. On the other hand, deep customization in theory allows customers to do any change on the shared service, which may cause severe security issues. Our approach at this stage does not solve these two problems. Vendors that allows deep customization has to introduce supporting facilities, such as sandbox, continuous automatic testing, vendor-involved code review, customer certification, etc. As a future plan, we will also investigate potential automatic support to mitigate these two problems, e.g., the generation of test cases to check the compatibility of custom code after main service updates, the static code analysis across main service and custom code for security purpose, etc.

References

[1] M. A. Rothenberger and M. Srite, "An investigation of customization in ERP system implementations," *IEEE Transactions on Engineering Management*, vol. 56, no. 4, pp. 663–676, 2009.

[2] H. Song, F. Chauvel, A. Solberg, B. Foyn, and T. Yates, "How to support customisation on saas: a grounded theory from customisation consultants," in *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, 2017, pp. 247–249.

[3] Microsoft. MusicStore test application that uses ASP.NET/EF Core. [Online]. Available: https://github.com/aspnet/MusicStore

[4] S. Heyenrath. The .NET Standard / .NET Core version from the System Linq Dynamic functionality. [Online]. Available: https://github.com/StefH/System.Linq.Dynamic.Core

[5] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[6] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, no. 4, 2008.

[7] J. Lee and G. Kotonya, "Combining service-orientation with product line engineering," *IEEE software*, vol. 27, no. 3, pp. 35–41, 2010.

[8] R. Mietzner and 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.

[9] W.-T. Tsai and X. Sun, "SaaS multi-tenant application customization," in *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*, 2013, pp. 1–12.

[10] T. Nguyen, A. Colman, and J. Han, "Enabling the delivery of customizable web services," in *Web Services (ICWS), 2012 IEEE 19th International Conference on*. IEEE, 2012, pp. 138–145.

[11] Salesforce. Apex Developer Guide. [Online]. Available: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/

[12] T. Kwok and A. Mohindra, "Resource calculations with constraints, and placement of tenants and instances for multi-tenant saas applications," in *International Conference on Service-Oriented Computing*. Springer, 2008, pp. 633–648.

[13] Oracle. Applicaiton Development SuiteScript. [Online]. Available: http://www.netsuite.com/portal/platform/developer/suitescript.shtml

[14] C. J. Guo, W. Sun, Y. Huang, Z. H. Wang, and 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.

[15] S. Walraven, E. Truyen, and 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.

[16] S. Walraven, D. Van Landuyt, E. Truyen, K. Handekyn, and W. Joosen, "Efficient customization of multi-tenant software-as-a-service applications with service lines," *Journal of Systems and Software*, vol. 91, pp. 48–62, 2014.