SOA*dapt*: A Framework for Developing Service-Oriented Multi-Tenant Applications

Sascha Alda, Rüdiger Buck-Emden
Department of Computer Science
Bonn-Rhine-Sieg University of Applied Sciences
Sankt Augustin, Germany
e-mail: sascha.alda@brsu.de, ruediger.buck-emden@brsu.de

***Abstract*— A plethora of architectural patterns and elements for developing service-oriented applications can be gathered from the state-of-the-art. Most of these approaches are merely applicable for single-tenant applications. However, less methodical support is provided for scenarios, in which multiple different tenants with varying requirements access the same application stack concurrently. In order to fill this gap, both novel and existing architectural patterns, architectural elements, as well as fundamental design decisions must be considered and integrated into a framework that leverages the development of multi-tenant application. This paper addresses this demand and presents the SOA*dapt* framework. It promotes the development of adaptable multi-tenant applications based on a service-oriented architecture that is capable of incorporating specific requirements of new tenants in a flexible manner.**

Keywords: Service-Oriented Architecture; Architectural Patterns; Multi-Tenant Application; Adaptation of Software.

I. Introduction

Service-oriented architectures [1] are nowadays used as a way to encapsulate and to integrate databases and applications being part of an enterprise's software landscape in terms of semantically enriched and re-usable *business services*. These business services can be orchestrated to more meaningful *workflows* that serve as executable software parts of business processes. On top of that, a *user interface* layer enables the involvement of human stakeholders during the execution of a workflow, for instance, to request initial or intermediate user inputs, as well as to represent final outputs.

The success of a service-oriented application depends on a number of factors. One important factor is the accurate modeling of the workflows including the regular flow of the activities and the potential alternate control flows. Another factor is the accurate design of the business services. These factors become even more relevant in application scenarios, in which different independent organizational units, hereafter called *tenants* (e.g., other organizations, subsidiaries of a company, faculties of a university), intend to share the same service-oriented application. The result is a *multi-tenant application* [2], in which different tenants access the same instance of an application's service stack concurrently. This concept is the foundation for latest software consumption models like software as a service (SaaS) [3] leading to high cost-effectiveness for each tenant. However, tenants often require control flows different to standardized, already deployed workflows in order to respect individual requirements. Providing a new workflow model for each tenant would be straightforward, but it breaks the idea of multi-tenancy. In addition, tenants often demand for alternate functionality to already deployed business services or even require completely new services that need to be flexibly deployed. The common approach of extending the original interfaces for each new tenant coming into play results in expanded interfaces and the risk of violating existing dependencies to other tenants.

Although a number of architectural approaches for service-oriented architectures are available [1] [4], none of these offer sufficient support for adapting both workflows and business services according to the needs of a multi-tenant application. Existing patterns and methods [5] [6] for adaptable service-oriented architectures are applicable to single-tenant applications, but are less appropriate to support the adaptation and management of multi-tenant applications.

This paper features the SOA*dapt* framework serving as a guideline for constructing a multi-layered service-oriented architecture that defines the structural decomposition of multi-tenant applications. With respect to this framework, the resulting software architecture offers a layered application stack, which is shared by multiple tenants at the same time. On the business process layer, the framework features a minimal set of *basic workflow patterns* that is suitable to model the various requirements of the tenants' workflows. On a business service layer, three types of business services can be deployed: shared business services suitable to all tenants, business services with dedicated tenant-specific service extensions, as well as fully self-contained services. Service extensions add both additional interfaces and internal behavior to the original service component that have not been anticipated and integrated in its original design. The framework contains further *architectural elements,* such as a business rule engine and a tenant context data registry that completes the architecture. Important *architectural design decisions,* such as a workflow instance model are made. Furthermore, *recommendations* for the *implementation* of the architecture are provided based on modern technologies.

This paper is structured as follows: Section 2 summarizes the related work. Section 3 describes the structure and the principles of the SOA*dapt* framework. Section 4 describes the prototype of the framework and outlines pieces of future work. Section 5 concludes this paper.

II. Related Work

The often cited SPOSAD architecture style by Koziolek [7] provides an abstract perspective on existing multi-tenant applications, such as Force.com, merely discussing the design decisions, as well as the architectural trade-offs related to this style. The resulting multi-tier architectural style is similar to our framework. It features a context-data manager that is responsible for adapting the application logic for tenant-specific business logic and computations. Unlike in the SOA*dapt* framework, no further details are provided how the context-data for a specific tenant is organized and in what way the business logic can actually be adapted for a given tenant. A clear distinction between business service and workflow is not handled in the SPOSAD architectural style.

The work by Mietzner et al. [8] provides fundamental research on instance management for multi-tenancy and describes a set of so-called service tenancy patterns. The pattern catalogue features basic architectural elements, such as the invocation of a service or a process *under tenant context*. The tenant context is actually a piece of context-data provided by a runtime environment that describes the current tenant accessing the application. Tenant-specific customizations of business processes, however, result in the deployment of a new workflow model. Dedicated workflow patterns are not outlined. A similar approach for identifying the current tenant ("tenant context object") can be found in [9]. The SOA*dapt* framework adopts this concept for identifying a tenant and explains precisely, where it should be used.

Further studies on multi-tenant Web applications are presented by Jansen et al. [10] and Bien and Thu [9]. Both papers rely on the MVC architectural pattern for describing the global structure of a multi-tenant application. Further fine-grade models (e.g., class models) can be found. Although service-oriented applications are typically Web-based, these studies can hardly by mapped to the demands of a multi-layered architecture. A workflow perspective is ignored in both studies.

Kabbelijk [11] discusses the adoption of various combinations of multi tenancy patterns to a multi-layered architecture. These patterns are based on the number of instances e.g., of an application server or of a database necessary to serve the tenants. Owing to the rather technical perspective of the architecture, a business-driven workflow layer is omitted. So, no dedicated workflow patterns can be found in his work. However, our workflow instance model can be compared with his work and further properties and constraints of it might be extracted from his contribution.

A multi-tenant approach for business process execution can be found in the article by Pathirage et al. [12]. The authors describe an architecture based on Axis2 runtime environment and the Apache ODE workflow engine. The authors mainly discuss how the runtime environment can guarantee isolation of the running processes. Workflow patterns are not described in the paper either.

Fundamental methods for adaptable software architectures have been elaborated by Svahnberg et al. [13]. He proposes five categories of adaptation methods. Our approach corresponds to the second category "variant component specialization": additional behavior is introduced in the same component or workflow models for different tenants.

A number of academic approaches for adapting (service-oriented architectures) can be found from the state-of-the-art (e.g., [5] [6]). In the majority of cases, these approaches rely on replacing entire components on-the-fly (category one w.r.t. [13]). This approach is straightforward, but leads to dependency issues as elaborated in the introduction section.

In our approach, a (business) rule engine is used to alter the control flow at gateway elements with respect to the demands of the involved tenants. Rule engines are primarily used for evaluating complex rules that cannot be incorporated in a workflow model in a manageable way. We adopt the idea of Doehring et al. [14] to use a rule engine also for control flow management. Doehring's article, however, is not based on the multi-tenancy approach.

III. The SOA*dapt* Framework

The SOA*dapt* framework introduces architectural elements and decisions, as well as architectural patterns in order to implement adaptable multi-tenant applications. An architectural overview of the framework is given in Fig. 1. The framework is merely based on a layered architecture model. A vertical enterprise service bus (ESB) component provides for a loosely coupled interaction among the components of these layers. In the following, the layers are described in detail. Special attention is drawn to both the business process and the business service layer as both layers consists of the main concepts of the framework. The user interface, as well as the application and data layer are introduced briefly. Implications for an implementation of the concepts are provided as well.

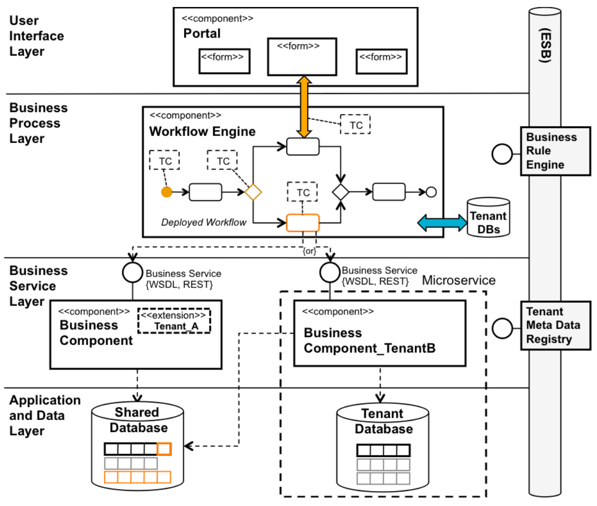

Figure 1. Architecture of the SOA*dapt* framework

A. *Business Process Layer*

This layer features a workflow engine, where an executable workflow model can be deployed. Although an object-oriented language can implement such a workflow, we assume a modeling language like BPMN 2.0 [15] as the preferred way to implement such workflows. In contrast to a business service, the functional behavior of a workflow model is said to be more comprehensive and may integrate user interactions through an associated portal component (see User Interface layer in Section III C). Workflow models are potentially stateful, that is, they can maintain a state across many workflow steps. Next, further important architectural decisions and elements of that layer will be outlined: the instance model, tenant context object, as well as the minimal set of workflow patterns.

1) *Workflow Instance Model*

A fundamental architectural property and design decision to be made for a multi-tenant application is the underlying workflow instance model. An overview is given in Fig. 2.

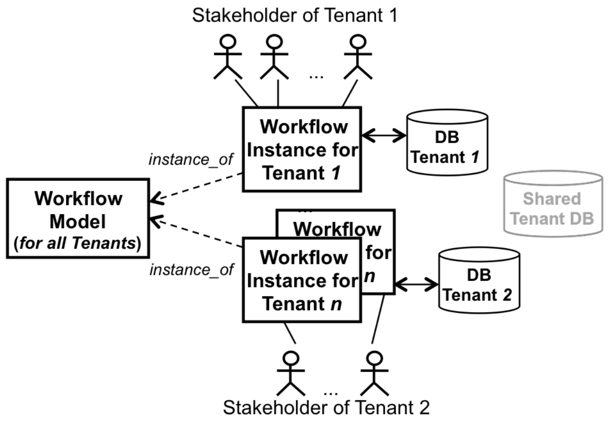

Figure. 2: Workflow instance model

It is assumed that both the structure and the course of actions of a workflow are described as a template that is referred as the *workflow model*. This could for instance be a graphical BPMN 2.0 model of a complaint management process. In a multi-tenancy application, a workflow model is said to be available for *all* tenants. Tenant-specific differences in the course of actions within a workflow model, such as different control flows or additional tasks for handling a complaint can be integrated by means of dedicated workflow patterns (see below). Given a predefined workflow model, an infinite number of *workflow instances* can be generated after the starting events has been fulfilled (e.g., triggered by the request of a user belonging to a tenant).

It is conceivable that in a strict multi-tenant application varying tenants can even share a single workflow instance. However, this approach might lead to typical technical issues as often faced on shared resources, that is, isolation problems of tenant-sensitive data, concurrency issues, or reduced scalability properties. Apart of that, such a shared instance model could hardly be appreciated from a tenant perspective. Henceforth, a single workflow instance is assigned to exactly one tenant. To further reduce the number of workflow instances, stakeholders of the same tenant could share a tenant's workflow instance (upper half of Fig. 2) – assumed that this accords to given tenant governance rules. For more restrictive scenarios, an instance could be generated for each stakeholder belonging to a tenant (lower half of Fig. 2).

The state of a workflow instance (i.e., variables, current execution state) can be stored temporarily in a corresponding tenant database, thus, guaranteeing rigidity and recoverability of the whole application. For statistical analytics, tenant data could also be stored permanently. Again, depending on given governance rules, tenant data could be stored in isolated databases (see Fig. 2). For less sensitive data or data that can trustworthily be shared among tenants (e.g., postal codes or standardized product numbers), shared databases can be integrated (see Section III.D for further details).

A workflow engine should be able to support all possible variations for workflow instance management. For the sake of scalability, separate instances of a given workflow engine could be installed on varying nodes (e.g., in a Cloud or on-premise in a local architecture). Each workflow engine instance could accommodate a dedicated workflow model or a cluster of coherent workflow models. The provision of a new workflow engine instance for each new workflow instance is a theoretical model achieving maximum scalability. However, this must be clearly contemplated from both a management and an economic perspective.

2) *The Tenant Context Object*

The *tenant context object* is an architectural data element representing the associated tenant of a current user accessing the workflow instance. Likewise to other context objects commonly found in Web frameworks (e.g., HttpSessionContext in the Servlet API), this object must be implemented (or rather: understood) as a global variable that is accessible in all areas of the workflow instance and corresponding objects, such as the service delegate object (see later on). The mapping of a user to a tenant is interpreted as a function that uniquely maps a given user (ID) to a tenant (ID). It is assumed that a user is derived by credentials that are passed in the beginning of the workflow execution, which is then stored e.g., in a session context object. The tenant context object tackles various parts and actors within the architecture as illustrated in Fig. 3:

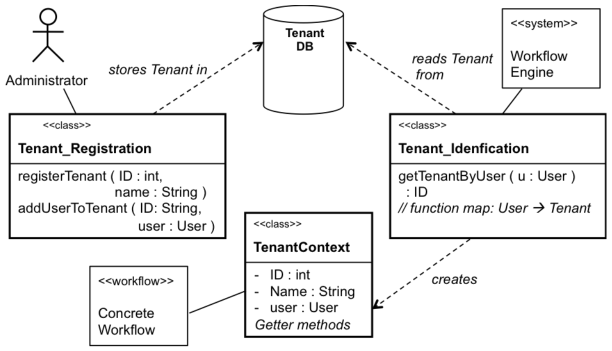

Figure. 3: Structural model of the tenant context

An administrator is able to initially register a tenant in the workflow engine. Depending on the chosen instance model, the tenant's data is stored within a single table or even in a separate database that guarantees isolation of the tenant's sensitive data. Given a unique tenant ID, the administrator is then able to add users to a tenant. For the purpose of flexibility, it is assumed that users can be added to a tenant even at runtime of the workflow engine. The workflow engine itself can request a tenant ID from a user by accessing class Tenant_Identification. This class will produce the tenant context object. The current workflow instance can access and read the attributes of that object accordingly. Attributes of the tenant context object might be used for debugging purposes, for identifying tenant-specific context information from the tenant meta data registry, or for evaluating the control flow for a tenant (see patterns below in Section III.A.3).

3) *Workflow Patterns*

Our *workflow patterns* describe solutions for recurring situations during the design of a workflow model, where the execution of a workflow might expose a different behavior based on the currently given tenant. In the layered architecture of Fig. 1, these situations are marked by the "tc" symbol surrounded by a dotted rectangle. This rectangle points out that at this stage of the workflow, the invocation of that element is *under tenant context* and, thus, it might vary. The patterns shown next provide an abstract solution and indicate implementation details. While the depicted solutions abstract from a concrete language, the implementation details will be based on BPMN 2.0. The set of patterns is considered as *minimal* and *complete*, that is, more complex and language-specific workflow elements (e.g., compensation, exception handling, sub processes) can be derived easily from this set.

Name of the pattern: Tenant-specific workflow invocation.

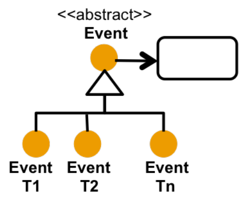

Figure. 4: Structure of the pattern "Tenant-specific workflow invocation"

Problem: The initial enactment of a workflow might vary based on the currently given tenant. That is, tenants might dictate different conditions when a workflow is to be executed. Often, additional tasks are required that have not been regarded in the standardized workflow model.
Solution: Introduce an abstract event that might be extended by concrete events that contain concrete conditions for specific tenants (see. Fig. 4). Assume a polymorphic structure: new concrete events might be introduced and bound to the abstract event at runtime (late binding).
Implementation: The introduction of an event hierarchy would result in an extension to the syntax of the BPMN 2.0 language. Consequently, a workflow engine like Activiti [16] had to be extended as well. As a trade-off solution, different starting events could be modeled and connected to the first activity in a workflow. Both BPMN 2.0 and Activiti do not support a late binding concept.

Portability: BPMN 2.0 features a bunch of different starting and intermediate event types, often leading to complex workflow models. This pattern could necessarily be ported to other events types. The usage should, however, be pondered.

Name of the pattern: Rule-based control Flow

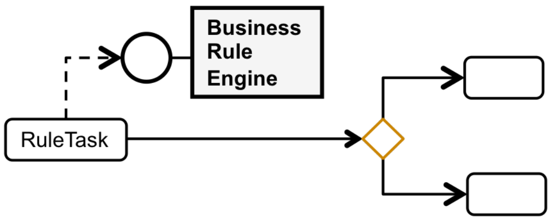

Figure. 5: Structure of pattern "Rule-based control flow"

Problem: Gateways may be used for altering the control flow. Typically, Boolean expressions are used to express conditions on the different branches that must be fulfilled for an execution. In a multi-tenant application, these conditions might differ for any of the involved tenants and, consequently, might exhibit a complex structure. The expression of conditions is not fixed, but is often subject of change (e.g., rules for expressing the credit rating of a client). The adaptation of conditions would result in the re-deployment of the workflow model.
Solution: Prior to the gateway element, a rule task is placed that uses an external rule engine to evaluate the conditions for a given tenant (see. Fig. 5). Conditions can be expressed by means of more descriptive models, such as decision tables [14]. These expressions can be adapted without re-deploying the workflow model, since both rule engine and workflow engine are completely decoupled.
Implementation: BPMN 2.0 already features rule tasks that can be used to evaluate rules from a rule engine. In the Activiti engine, the business rule engine Drools Expert can be used [16]. In order to execute the deployed rules, input variable (the so-called facts) and the result variables need to be specified in the context of this rule task. For evaluating the rules with respect to a given tenant, the tenant context object must be passed as a fact, too. The output variable will contain a list of objects that can then be evaluated at the branches of the corresponding gateway element. Depending on the state of the output variable, the control can be altered according to the demands of a respective tenant. The Drools platform offers tools for the *simple* modification of the rules e.g., within a decision table, which could even be carried out by business units with minor IT-background.
Portability: This concept cannot only be applied to exclusive, but also to inclusive gateways, in which a subset of modeled branches might be invoked concurrently. Although it is assumed that tenants share branches, it is feasible to insert branches that can exclusively be used by a dedicated tenant. Besides, additional tenant-specific activities or sub processes can be integrated into a standard workflow model.
Trade-Off: The more complex the rules and facts in a rule engine become, the more performance is needed for a thorough evaluation. So, the rule engine might result in a bottle-

neck within the whole architecture. Software architects should think of caches to store previous results.

Name of the pattern: Tenant-specific service call

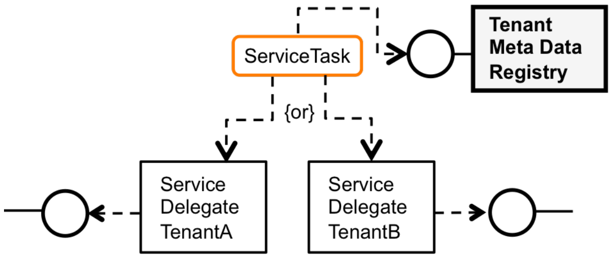

Figure. 6: Structure of pattern "Tenant-specific service call"

Problem: service call tasks implement the actual invocation of business services being placed in the business service layer. Usually, these tasks are passed input data stemming from prior tasks (e.g., a user task requesting data from a user through a portal) that serves as an input to the business service. The resulting output of that service might then be further used in subsequent tasks. Depending on the given tenant, different business services or extensions (see Section III.B) might be invoked from the business layer. Depending on the business service's interface, input and output data has to be converted (e.g., XML to JSON) or enriched (e.g., adding the postal code to a person's profile). Also, the exception handling might be different depending on the nature of the business service (e.g., transactional vs. non-transactional service call). These different service calls together with specific preliminary and subsequent tasks might be implemented by using the "Rule-based control flow" pattern. However, this would blow up the workflow model with too many technical gateway elements and different branches having no real business added value.

Solution: The invoked service tasks delegates the actual service invocation to a so-called service delegate object that is responsible of processing the whole service call including data conversion or enrichment, (remote) service invocation, and exception handling (see. Fig. 6). The correct service delegate object is instantiated by calling the tenant context data registry that stores the corresponding business services per tenant (see III.B for more details). The correct service delegate object is identified based on the current tenant context object.

Implementation: BPMN 2.0 provides service tasks as a core element of the language. However, the language itself does not support the concept of an internal service delegate element. In the Activiti engine, the invocation of a service task is handled by a service handler object, which is actually a Java object implementing a given API. The service handler object can access the input data by using a global context object, the so-called DelegateExecution object [16]. Analogously, output data can be conveyed back to the workflow instance through that context object. In this service handler object, the corresponding service delegate object can be invoked for processing the tenant-specific service call. Developer of the corresponding service handler class can fall back on the complete Java SDK, further related APIs (e.g., JAX-RS for invoking REST-based services), or frameworks (e.g., Zend for data conversion between XML and JSON). This lightweight approach is actually an improvement compared to older development models from BPEL-based engines (e.g., Apache ODE), where service calls need to be graphically bound to WSDL files of the Web Services by some proprietary and tricky development tools.

B. Business Service Layer

This layer features business components that provide business services to the upper layers. Business services provide a business value and can be reused and orchestrated in different workflow models. The corresponding interfaces of business services are described in a language-neutral format (WSDL or REST-based), from which client stubs can be generated in order to access the business service. Business components may wrap underlying applications or databases from the data and Application layer (see III.D). Business components can be implemented in an object-oriented language like Java. The deployment can be done in a conventional Java runtime environment (JRE) or in an application server, such as Axis2 or Glassfish. Business components should be stateless to ensure the scalability of the architecture and the substitutability of components within this layer. For the sake of brevity, Fig. 1 does omit execution environments on the business service layer.

Ideally, all business services can be shared among all available tenants. However, this scenario is rather seldom, since tenants are supposed to demand varying properties on the business services used in their workflows models. In order to respect different scenarios, three categories of business services are supported: (1) shared business services, (2) business services with service extensions, or (3) standalone business services deployed as Microservices. This paper mainly introduces type (2) and sketches the idea of type (3) briefly. Both variants are introduced as architectural patterns.

Name of the pattern: Service extension

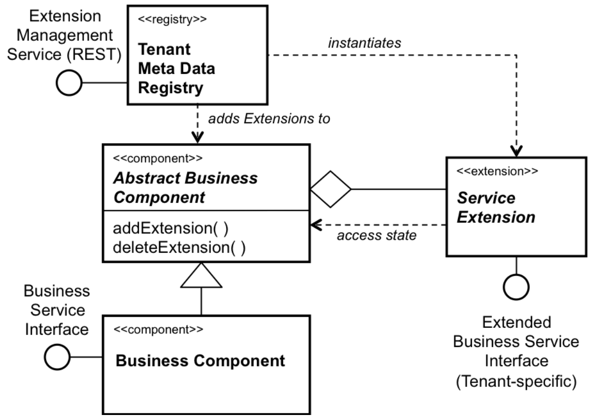
Extension Management Service (REST)
<<registry>>
Tenant Meta Data Registry
instantiates
adds Extensions to
<<component>>
Abstract Business Component
addExtension( )
deleteExtension( )
<<extension>>
Service Extension
access state
Business Service Interface
<<component>>
Business Component
Extended Business Service Interface (Tenant-specific)

Figure. 7: Structure of pattern "Service extension"

Problem: Tenants might expose varying demands on the usage of a business service. Apparently, it seems unrealistic to anticipate the complete externally visible behavior - i.e., the interface - of a business service that tenants might use in the future. This also applies to internal implementation details within the respective business components. By doing so,

the resulting service interface might be bloated with too many service operations. Integrating new operations into the interface might cause the violation of dependencies to existing components that are coupled to the original interface.
Solution: So-called service extensions can extend the interface of a business component by further service operations that are part of an extended business service. This pattern is an adoption of the extension object pattern [17], in which objects can act as a host for object extensions that can flexibly be added and removed from that hosting object. The structure of the service extension pattern (see Fig. 7) is a slight modification of the original structure as it leaves out various abstractions and concentrates on the relevant elements. An abstract business component serves as the hosting component that provides an administrative interface for adding and removing service extensions. Again, service extensions implement tenant-specific behavior. For the proper execution of this behavior, service extensions have access to the internal state of a business component. Appropriate access rights must be granted accordingly. The business component itself inherits from the abstract business component. Note that this pattern consciously abstracts from concrete implementation techniques, the inheritance relationship just points out the different responsibilities of the involved elements. In modern component models, such as OSGi, the inheritance relationship might be dissolved, thus, resulting in a business component providing both the functional interface of the business service and the administrative interface.

The *tenant meta data registry* is in charge of managing the different service extensions per involved tenant for a dedicated business service component. Besides, further context data, such as form elements (see Section III C) can be added to a tenant. At design time, component assemblers can use the registry for equipping a business service with selected service extensions for a new tenant. During runtime, the application server can use the registry for querying tenant-specific service extensions. Having identified the necessary meta data describing the service extension, concrete service extensions for a tenant can be deployed in the business component. For new service extensions, an upload mechanism for both the meta data and the actual executable of that service extension (e.g., a JAR-file) needs to be provided.

Table. I POSSIBLE QUERIES FOR CONTEXT DATA REGISTRY

Explanation	URI (& HTTP method)
Returns all registered tenants (IDs) of a business component with the id *compID*	GET /comp/[*compID*]/tenants
Returns all registered service extensions (IDs) of a business component with the ID *compID* that are associated with a tenant(ID)	GET /comp/[*compID*]/tenants/ [*tenantID*]/ext/
Registers a new tenant with ID *tenantID* to the business component with ID *compID*.	POST /comp/[*compID*] /tenant/ [*tenantID*]

The registry features a hierarchical model to represent associations among service extensions, business service components, and tenants. Owing to the hierarchical nature of the data model, URIs can be used for identifying the meta data. Consequently, a REST-based interface could be used as the fundament of the extension management service. Table No. 1 shows some example queries that could be applied.

Name of pattern: Business service as Microservice
Problem: Tenants might insist of having self-contained business services that come with their own database and domain model, which conforms to a shared nothing solution. Isolation of data and services is an absolute must criterion.
Solution: Tenants are invited to deploy self-contained business services by means of Microservices [18] into the architecture. Microservices are closed units of deployment with no or a minimal set of dependencies to other services and infrastructure components (e.g., server, databases). Typically, a Microservice has its own domain model, a so-called bounded context. A Microservice contains its own internal application server, such as Glassfish. The deployment can be done in lean execution environments, such as Docker or in cloud-based environments, such as Spring Cloud. The usage of systems like Spring Clouds promotes the scalability of the business service layer and, thus, the entire multi-tenant application. More details on both the theory and implementation of Microservices can be obtained from [18].

C. *User Interface Layer*

This layer consists of user interface (UI) components for involving stakeholders within a workflow execution. In practice, a portal may take over this part allowing the provision of customizable forms. A single form consists of a coherent set of user interface elements, such as buttons or text fields, accomplishing a stakeholder to process data associated with a user task. This data can act as the initial input in the beginning or as an intermediate input during a workflow execution. At the end of the workflow execution, final output data can be displayed. The form rendering heavily depends on the tenant's usability requirements. So, the rendering process and the data exchange between the portal and the workflow engine must occur under tenant context. The specific form elements (e.g., HTML, CSS fragments) should be stored in the tenant context data registry. The portal accesses this registry upon the rendering process for a specific user.

D. *Application and Data Layer*

This layer consists of existing legacy applications and databases. Databases could be based on data models, such as the relational model or variants (e.g., object-relational). A database can be shared among the tenants. For respecting tenant-specific data, tables for each single tenants or schema extensions to common tables can be inserted. The definition of shared data models for multi-tenant applications is not part of this framework. Different approaches how to organize such data models can be found for instance in [2] or [7].

IV. PROTOTYPE, FUTURE WORK

A first prototype of the SOA*dapt* framework has been implemented on top of the Activiti workflow engine [16]. Version 6 of Activiti integrates the proposed workflow instance model (see Section III.A.1) and the tenant context object (see Section III.A.2), which was contributed to the project in the context of a joint master thesis project [19]. The workflow instance model follows the approach of having one instance of a workflow engine that accommodates all generated workflow instances. Each tenant has a separate database, thus, guaranteeing isolation of data.

From the workflow patterns (Section III.A.3), pattern “Rule-based control flow” has been implemented based on Drools expert and on top of the workflow engine Doxis4 BPM [20]. An evaluation in conjunction with the German IT-company SER GmbH – the vendor of Doxis4 BPM - confirmed the flexibility of the approach for having flexible control flows. However, the company criticized the complex user interface of Drools for editing business rule and the overall complexity of the rules themselves. Therefore, future work was recommended for improving the definition of rules. First tests revealed no critical performance issues. An in-depth performance analysis e.g., with stress tests has not been carried out so far, but is considered as future work.

A prototype for the service extension pattern is considered as future work. An ongoing project examines the appropriateness of the server Axis2 for implementing service extensions. The tenant context data registry has been implemented as a first prototype based on the Jersey framework.

V. Conclusion

This paper has introduced the framework SOA*dapt* that can be used for the development of adaptable multi-tenant applications that are based on a multi-layered service-oriented architecture. The framework proposes a set of architectural patterns that allow the adaptation of a multi-tenant application in order to respect varying requirements of the involved tenants. SOA*dapt* also introduces architectural elements for setting up and running a multi-tenant application in a scalable way. First prototypes have been developed.

References

[1] M. Stal, "Using architectural patterns and blueprints for service-oriented architecture", IEEE Software, vol. 23, no. 2, pp. 54-61, 2006.

[2] F. Chong and G. Carraro, "Architecture Strategies for Catching the Long Tail", Available: http://msdn.microsoft.com/en-us/library/aa479069.aspx. April 2006. [retrieved: Jan., 2017]

[3] M. Turner and D. Budgen, "Turning Software into a service", IEEE Computer, vol. 36, no. 10, pp. 38-45, 2003.

[4] T. Erl, "Service-Oriented Architecture: Concepts, Technology, and Design", Prentice Hall, 2005.

[5] R. Mirandola, P. Potena, E. Riccobene, and P. Scandurra, "A Framework for Adapting Service-Oriented Applications based on Functional / Extra-functional Requirements Tradeoffs", 6th International Conference on Software Engineering Advances, Barcelona, Spain, pp. 146-151, 2011.

[6] H. Gomaa, K. Hashimoto, M. Kim, and e. al., "Software Adaptation Patterns for Service-Oriented Architectures", 25th Symposium On Applied Computing, Sierre, pp. 462-469, March 2010.

[7] H. Koziolek, "The SPOSAD Architectural Style for Multi-tenant Software Applications", 9th Working IEEE Conference on Software Architecture, Boulder, USA, pp. 320-327, 2011.

[8] Mietzner R., Unger T., R. Titze, and F. Leymann, "Combining Different Multi-Tenancy Patterns in Service-Oriented Applications", IEEE EDOC Conference, pp. 108-117, 2009.

[9] N. H. Bien and T. D. Thu, "Multi-tenant web application framework architecture pattern", 2nd National Foundation for Science and Technology Development Conference on Information and Computer Science, Vietnam, pp. 40-48, 2015.

[10] S. Jansen, G. Houben, and S. Brinkkemper, "Customization Realization in Multi-Tenant Web Applications: Case Studies from the Library Sector", 10th International Conference on Web Engineering, Vienna, pp. 445-459, 2010.

[11] J. Kabbedijk and S. Jansen, "Multi-tenant Architecture Comparison ", 8th European Conference in Software Architecture, Vienna, pp. 202-209, 2014.

[12] M. Pathirage, S. Perera, I. Kumara, and S. Weerawarana, "A scalable Multi-tenant Architecture for Business Process Executions", IEEE International Conference on Web Services, pp. 21-41, 2011.

[13] M. Svahnberg, J. Grup, and J. Bosch, "A taxonomy of variability realization techniques", Software Practice and Experience, vol. 35, no. 8, pp. 705-754, 2005.

[14] M. Döhring, B. Zimmermann, and E. Godehardt, "Extended Workflow Flexibility using Rule-Based Adaptation Patterns with Eventing Semantics.", Informatik 2010: Service Science - Neue Perspektiven für die Informatik, Leipzig2010.

[15] OMG, "BPMN 2.0 Specification", Available: http://www.omg.org/spec/BPMN/2.0/. 2011. [retrieved: Jan., 2017]

[16] T. Rademakers, "Activiti in Action - Executable business processes in BPMN 2.0", Manning Pub., 2012.

[17] E. Gamma, "The extension objects pattern", Pattern Languages of Programs (Plop) Conference - Writers Workshop, Illinois, USA1996.

[18] E. Wolff, "Microservices: Flexible Software Architectures", CreateSpace Publishing, 2016.

[19] J. Barrez, " Multi-Tenancy with separate database schemas in Activiti", Available: http://www.jorambarrez.be/blog/2015/10/06/multi-tenancy-separate-database-schemas-in-activiti/. 2015. [retrieved: Jan., 2017]

[20] SER, "Business process management with Doxis4", Available: http://www.ser-solutions.com/products-solutions/business-processes.html. 2016. [retrieved: Jan., 2017]