IT Scenarios for Service Component Architecture (SCA)

Applies to:

Service Component Architecture technology is applicable to SAP NetWeaver Platform in general and in particular it is relevant to the future roadmap of SAP NetWeaver Process Integration and NetWeaver Composition Environment products.

Summary

The Service Component Architecture (SCA) standard comprises of different technical areas and many specifications. In reading the details of the various SCA specifications, it is possible to miss the big picture about the overall value proposition of SCA. This article describes a set of high-level IT scenarios that highlight the problems SCA attempts to solve, and thereby contextualize the various SCA technical areas.

Author(s): Sanjay Patil
Company: SAP Labs – Palo Alto
Created on: 30 December 2007

Author Bio

Sanjay Patil is a standards architect at SAP Labs, Palo Alto, and is an active member of many standards bodies including OASIS, WS-I and JCP. He currently co-chairs the OASIS Web Services Reliable Exchange (WS-RX) Technical Committee and the OASIS Service Component Architecture BPEL (SCA-BPEL) Technical Committee. His areas of interest include web services, SOA, enterprise service bus and composition technologies.
Introduction

The Service Component Architecture (SCA) standard comprises of different technical areas and many specifications. In reading the details of the various SCA specifications, it is possible to miss the big picture about the overall value proposition of SCA. This article describes a set of high level IT scenarios which highlight the problems SCA attempts to solve, and thereby contextualize the various SCA technical areas.

Note that these scenarios are not exhaustive but they are merely exemplary.

Let us first clarify some of the different IT roles involved in these scenarios:

- Component Developer – This role is responsible for writing the implementation code with the actual business logic, which can be configured and composed with other implementation artifacts.
- Composite Assembler – This role is responsible for configuring the different implementation artifacts and composing them together.
- Composite Deployer – This role is responsible for configuring the composite artifact with metadata related to utilization of infrastructure e.g. messaging.

Bottom-Up Composition

This scenario is about composing an application by assembling different existing implementation artifacts. Needless to say, any business would welcome an ability to repurpose existing implementation assets to quickly meet new business demands. Note that this is not just about consuming existing deployed service endpoints, but it is primarily about consuming the same implementation artifacts with potentially different configurations to meet the high level composite specific settings. For example, an implementation of a product price calculation component can be configured differently depending upon the target geographic location of the deployment. The different configurable properties of the generic implementation may include the local currency code, tax rate, etc.

Let us now take a look at the high level steps involved in this scenario
**Bottom-up Composition**

1. **Select a set of existing component implementations for building the new composite**
2. **Configure the component properties**
3. **Draw internal wires**
4. **Wrap the components in a composite and configure external services/references**
5. **Hand off the composite to Deployer**

The Assembler is tasked with composing an application to meet certain business requirements by using existing implementation assets. There may be a repository of implementation assets which the Assembler can browse and pick a set of implementations relevant to the target composite.

For each selected implementation artifact, the Assembler analyzes and sets values for the configurable properties e.g. set the currency code of the product price calculation component to USD if the deployment is targeted for geographic areas with USD as the currency.

The different components comprising the high level composite may be invoking services of each other and possibly some external services also. The Assembler models such service invocation dependencies by configuring what is called as ‘wires’ in the SCA parlance.

In the process of configuring the various components and the wires, the Assembler is actually creating the high level composite, for which the Assembler can define the services exposed to the consumers of the composite, the external services the composite depends upon and the configurable properties of the composite itself.

After the composite is created by the Assembler, a Deployer can then add metadata to the composite that is specific to utilization of the infrastructure. For example, certain component may need transactional environment, certain service invocation may need to use reliable messaging, etc.

The technical requirements to support this scenario would include - a language to describe the configurable aspects of the implementation artifacts in a uniform manner such that an Assembler can compose them together, and a model and methodology for the Assembler to create the composite itself. SCA Assembly Specification [1] aims to meet these technical requirements.
Top-Down Composition

As indicated by the title, this scenario takes the opposite approach for creating the composites when compared with the previous scenario of Bottom-up composition. Here, the goal of an Assembler is to analyze the high level business requirements and create composites on top of components that are yet to be implemented. Most practical cases would involve a combination of Bottom-up and Top-down scenarios, as certain implementation artifacts may already exist while a set of new implementation artifacts may have to be created in order to materialize the overall composite. For the sake of simplicity, the Top-down scenario here assumes that the composite and the involved component implementation are to be built from scratch.

Let us now look at the typical steps of this scenario.

Given the business requirements, the Assembler first defines the skeleton of the high level composite in terms of the services offered by the composite, the external services the composite is going to consume and the configurable business properties.

The Assembler then identifies the major chunks of functionalities needed to materialize the composite, and for each chunk, the Assembler defines the component metadata -- the offered services, consumed services and configurable properties.

For each of the component, the Assembler further analyzes whether there is a need to break down that component into even smaller chunks. By applying the same methodology (as applied for the high level composite) in a recursive manner, the Assembler defines the deeper level components.

At the end of the above exercise, there should be a set of component definitions which can then be handed off to the Developers for actual implementations. Depending upon the nature of the business logic involved, different implementation technologies may be used for the various components. For example, if a component involves mapping of document types, its developer may resort to using a mapping tools and create artifacts.
that can be deployed in some mapping engine. On the other hand, if a component involves algorithm intensive logic, a suitable programming language may be used to develop its implementation.

Similar to the previous scenario of Bottom-up composition, the Top-down scenario requires a technology to describe components in a manner that is independent of their implementations, and a mechanism to compose the components together by configuring their implementations and wiring them together. The SCA Assembly Specification [1] aims to provide a model and syntax for the language-neutral component descriptions and their composition. The various SCA Client and Implementation specifications (for Java, BPEL, C++, etc) [2] define the mapping of the language neutral component metadata to the specifics of the implementation languages.

**Heterogeneous Composition**

This scenario is aimed to highlight a very important aspect of SCA, that is – how SCA simplifies the composition in a heterogeneous environment, which would be a typical scenario for most IT departments today.

Interestingly, creating an environment where heterogeneous components are participating in the same high level application is possible today without use of any SCA technology. For example, if a high level application involves a BPEL process and an EJB, you could develop a BPEL process and deploy it to its container, and similarly you could develop an EJB and deploy it to its runtime. Separately, you could configure the connections between the BPEL process and the EJB, etc. In effect, you can make the overall integrated application work as desired. Integrating heterogeneous application is nothing new for today's industry as such. Where SCA comes into picture and adds value is that – Instead of looking at integration of applications as a separate challenge, SCA enables a consistent view of the various applications and provides a simpler environment for creating compositions as first class applications. SCA allows structuring the integration logic, and related functions (such as mapping, etc.) as first class components, whose relationship with other components is then captured as part of a well defined composite.
SCA model allows taking a holistic view of the overall project involving various heterogeneous applications, as opposed to dealing with the individual components in an isolated manner. There are many practical reasons why such a holistic view is critically important. For example, when the high-level composition is to be decommissioned, it would be preferable to issue a single undeploy command and expect that the appropriate undeploy activities for the different component runtime get triggered. Similarly, if you wanted to turn on logging/tracing for the high-level composition, it would be preferable to apply a single setting of `tracing=on` at the composite level and expect that logging/tracing is turned on for all the individual components participating in that composite. Instead of going through the painful and error-prone exercise of tweaking the settings of each component runtime, it is desirable to treat the high-level composite as a first-class application and the set of various different component runtimes as nothing but a single platform for executing the high-level composition. SCA aims to enable creation of such an environment.

**Flexible Deployment**

With the increasing dynamics in today’s businesses, it is clear that the IT applications are expected to be flexible in providing access to new types of consumers and in consuming new types of services. Mergers and acquisitions, newly forged business partnerships, adoption of software-as-a-service model, expansion of markets into new areas, etc., are some examples of why applications should remain flexible to accommodate new types of client communities and consume new types of services.

Business logic and the technologies for communication each have a separate life cycle, a separate purpose and are typically handled by separate IT roles.

This scenario involves making an implementation available to clients using Web services-based communication mechanisms as well as clients using JMS. At some later point of time, yet another communication protocol may have to be supported. If the implementation code is tied to any specific communication mechanism, it would be extremely costly and inflexible to support different communication mechanisms.
At the technical level, such flexibility means that the applications should support a clear separation of the contained business logic and the mechanics of invoking that business logic. Furthermore, the mechanics of communication with a service should remain as a configurable aspect and should not involve any implementation code changes. The various SCA Binding (for Web services, JMS, JCA, etc.) specifications are aimed to enable such a flexibility.

**Abstract Policy Declaration**

A Developer of an implementation with critical business logic may be aware that the business logic should be accessed only by authenticated users. For example, the Developer may be cognizant of the fact that execution of the code involves making important changes to some underlying database and the Developer therefore wants to express that the invokers of the business logic must be authenticated. The Developer does not care about what authentication mechanism is selected. Similarly, the Developer may want to express that the service invocation be carried on a secure communication channel, without worrying about the security technology to be used.

**Abstract policy declaration**

1. Policy Administrator authors SCA policySets with concrete policies
2. Developer specifies intents on SCA assembly
3. Developer hands over SCA assembly to Deployer
4. Deployer configures SCA assembly by assigning SCA policySets (could be automated)
5. Deployer deploys configured SCA Assembly to SCA Runtime
6. Deployer updates Registry

It is true that today's technologies such as Web Services Policy (WS-Policy) and the domain specific assertions based on WS-Policy (Web Services Reliable Messaging Policy, Web Services Security Policy, etc.) do allow services based on Web Services to express there capabilities and constraints in a declarative and implementation independent manner. However, these technologies are primarily aimed at a Deployer to express specific infrastructural choices. These technologies do not offer much for a Developer to express the QoS requirements in an abstract manner.

From a technical standpoint, this scenario requires a declarative mechanism for Developers to express their abstract policy requirements as component metadata during the development phase. At some later point of time, a Deployer would provide metadata for concrete policies that matches the abstract policies. Such concrete policy metadata would be based on the available infrastructure, for example, for an abstract intent of guaranteed_delivery, the Deployer could choose to utilize a messaging system based on Web Services
Reliable Messaging. The SCA Policy Framework [4] aims to support such a model that supports abstract policy declarations, their mapping to domain specific concrete policies, etc.

Conclusion
While open standards related to Service Oriented Architecture (SOA) exist today, their focus is largely on either interoperability of heterogeneous runtimes (e.g. Web services) or they delve too deep in specific service creation technologies (e.g. Java EE). A gap is left open for creating and composing heterogeneous services in a simplified manner. By describing some key IT scenarios that highlight this gap, this article contextualizes the applicability and value of Service Component Architecture (SCA) standard.

References
[1] OASIS Service Component Architecture / Assembly Technical Committee
  OASIS Service Component Architecture / BPEL Technical Committee
  OASIS Service Component Architecture / C and C++ Technical Committee
[3] OASIS Service Component Architecture / Bindings Technical Committee
[4] OASIS Service Component Architecture / Policy Technical Committee

Related Content
- Dependency Injection Pattern in Service Component Architecture
- Does SCA need a new Java programming model?
- Relevance of SCA to a Business Process Expert